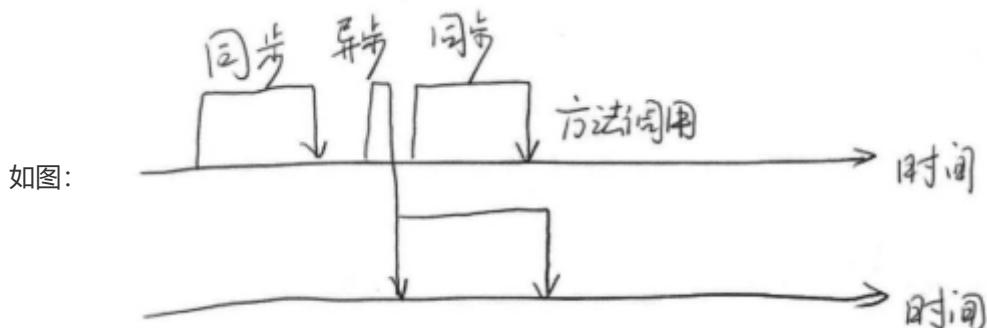


加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第1篇：必须知道的几个概念

同步 (Synchronous) 和异步 (Asynchronous)

同步和异步通常来形容一次方法调用，同步方法调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为。异步方法调用更像一个消息传递，一旦开始，方法调用就会立即返回，调用者就可以继续后续的操作。而异步方法通常会在另外一个线程中“真实”地执行。整个过程，不会阻碍调用者的工作。



上图中显示了同步方法调用和异步方法调用的区别。对于调用者来说，异步调用似乎是一瞬间就完成的。如果异步调用需要返回结果，那么当这个异步调用真实完成时，则会通知调用者。

打个比方，比如购物，如果你去商场买空调，当你到了商场看重了一款空调，你就向售货员下单。售货员去仓库帮你调配物品。这天你热的是不行了，就催着商家赶紧给你送货，于是你就在商店里面候着他们，直到商家把你和空调一起送回家，一次愉快的购物就结束了。这就是同步调用。

不过，如果我们赶时髦，就坐在家里打开电脑，在电脑上订购了一台空调。当你完成网上支付的时候，对你来说购物过程已经结束了。虽然空调还没有送到家，但是你的任务已经完成了。商家接到你的订单后，就会加紧安排送货，当然这一切已经跟你无关了。你已经支付完成，想干什么就能去干什么，出去溜几圈都不成问题，等送货上门的时候，接到商家的电话，回家一趟签收就完事了。这就是异步调用。

并发 (Concurrency) 和并行 (Parallelism)

并发和并行是两个非常容易被混淆的概念。他们都可以表示两个或者多个任务一起执行，但是侧重点有所不同。并发偏重于多个任务交替执行，而多个任务之间有可能还是串行的，而并行是真正意义上的“同时执行”，下图很好地诠释了这点。

Concurrent and Parallel Programming

05 Apr 2013

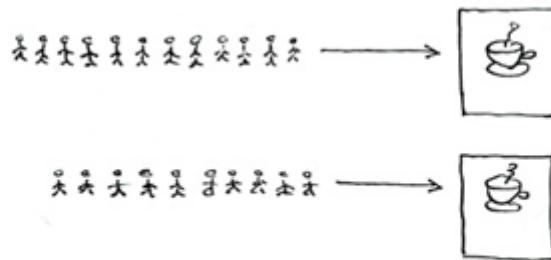
What's the difference between concurrency and parallelism?

Explain it to a five year old.

Concurrent = Two Queues One Coffee Machine



Parallel = Two Queues Two Coffee Machines



© Joe Armstrong 2013

Concurrent = Two queues and one coffee machine.

Parallel = Two queues and two coffee machines.

大家排队在一个咖啡机上接咖啡，交替执行，是并发；两台咖啡机上面接咖啡，是并行。

从严格意义上来说，并行的多任务是真的同时执行，而对于并发来说，这个过程只是交替的，一会执行任务A，一会执行任务B，系统会不停地在两者之间切换。但对于外部观察者来说，即使多个任务之间是串行并发的，也会造成多任务间并行执行的错觉。

并发说的是在一个时间段内，多件事情在这个时间段内交替执行。

并行说的是多件事情在同一个时刻同事发生。

实际上，如果系统内只有一个CPU，而使用多进程或者多线程任务，那么真实环境中这些任务不可能是真实并行的，毕竟一个CPU一次只能执行一条指令，在这种情况下多进程或者多线程就是并发的，而不是并行的（操作系统会不停地切换多任务）。真实的并行也只能出现在拥有多个CPU的系统中（比如多核CPU）。

临界区

临界区用来表示一种公共资源或者说共享数据，可以被多个线程使用，但是每一次只能有一个线程使用它，一旦临界区资源被占用，其他线程要想使用这个资源就必须等待。

比如，一个办公室里有一台打印机，打印机一次只能执行一个任务。如果小王和小明同时需要打印文件，很明显，如果小王先发了打印任务，打印机就开始打印小王的文件，小明的任务就只能等待小王打印结束后才能打印，这里的打印机就是一个临界区的例子。

在并行程序中，临界区资源是保护的对象，如果意外出现打印机同时执行两个任务的情况，那么最有可能的结果就是打印出来的文件是损坏的文件，它既不是小王想要的，也不是小明想要的。

阻塞 (Blocking) 和非阻塞 (Non-Blocking)

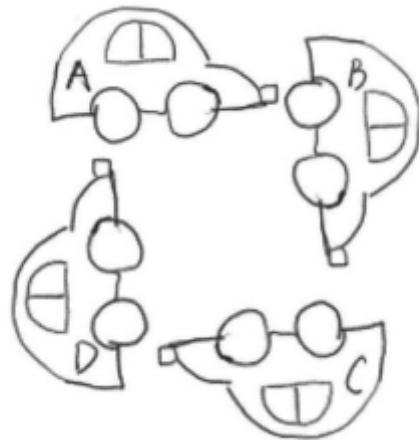
阻塞和非阻塞通常用来形容很多线程间的相互影响。比如一个线程占用了临界区资源，那么其他所有需要这个资源的线程就必须在这个临界区中等待。等待会导致线程挂起，这种情况就是阻塞。此时，如果占用资源的线程一直不愿意释放资源，那么其他线程阻塞在这个临界区上的线程都不能工作。

非阻塞的意思与之相反，它强调没有一个线程可以妨碍其他线程执行，所有的线程都会尝试不断向前执行。

死锁 (Deadlock) 、饥饿 (Starvation) 和活锁 (Livelock)

死锁、**饥饿**和**活锁**都属于多线程的活跃性问题。如果发现上述几种情况，那么相关线程就不再活跃，也就是说它可能很难再继续往下执行了。

死锁应该是最糟糕的一种情况了（当然，其他几种情况也好不到哪里去），如下图显示了一个死锁的发生：



A、B、C、D四辆小车都在这种情况下都无法继续行驶了。他们彼此之间相互占用了其他车辆的车道，如果大家都不愿意释放自己的车道，那么这个状况将永远持续下去，谁都不可能通过，死锁是一个很严重的并且应该避免和实时小心的问题，后面的文章中会做更详细的讨论。

饥饿是指某一个或者多个线程因为种种原因无法获得所要的资源，导致一直无法执行。比如它的优先级可能太低，而高优先级的线程不断抢占它需要的资源，导致低优先级线程无法工作。在自然界中，母鸡给雏鸟喂食很容易出现这种情况：由于雏鸟很多，食物有限，雏鸟之间的事务竞争可能非常厉害，经常抢不到事务的雏鸟有可能被饿死。线程的饥饿非常类似这种情况。此外，某一个线程一直占着关键资源不放，导致其他需要这个资源的线程无法正常执行，这种情况也是饥饿的一种。于死锁想必，饥饿还是有可能在未来一段时间内解决的（比如，高优先级的线程已经完成任务，不再疯狂执行）。

活锁是一种非常有趣的情况。不知道大家是否遇到过这么一种场景，当你要做电梯下楼时，电梯到了，门开了，这是你正准备出去。但很不巧的是，门外一个人挡着你的去路，他想进来。于是，你很礼貌地靠左走，礼让对方。同时，对方也非常礼貌的靠右走，希望礼让你。结果，你们俩就又撞上了。于是乎，你们都意识到了问题，希望尽快避让对方，你立即向右边走，同时，他立即向左边走。结果，又撞上了！不过介于人类的智慧，我相信这个动作重复两三次后，你应该可以顺利解决这个问题。因为这个时候，大家都会本能地对视，进行交流，保证这种情况不再发生。但如果这种情况发生在两个线程之间可能就不那么幸运了。如果线程智力不够。且都秉承着“谦让”的原则，主动将资源释放给他人使用，那么久会导致资源不断地在两个线程间跳动，而没有一个线程可以同时拿到所有资源正常执行。这种情况就是活锁。

死锁的例子

```
package com.jvm.visualvm;
```

```

/**
 * <a href="http://www.itsoku.com/archives">Java干货铺子,只生产干货,公众号:  

javacode2018</a>
*/
public class Demo4 {

    public static void main(String[] args) {
        Obj1 obj1 = new Obj1();
        Obj2 obj2 = new Obj2();
        Thread thread1 = new Thread(new SynAddRunalbe(obj1, obj2, 1, 2, true));
        thread1.setName("thread1");
        thread1.start();
        Thread thread2 = new Thread(new SynAddRunalbe(obj1, obj2, 2, 1, false));
        thread2.setName("thread2");
        thread2.start();
    }

    /**
     * 线程死锁等待演示
     */
    public static class SynAddRunalbe implements Runnable {
        Obj1 obj1;
        Obj2 obj2;
        int a, b;
        boolean flag;

        public SynAddRunalbe(Obj1 obj1, Obj2 obj2, int a, int b, boolean flag) {
            this.obj1 = obj1;
            this.obj2 = obj2;
            this.a = a;
            this.b = b;
            this.flag = flag;
        }

        @Override
        public void run() {
            try {
                if (flag) {
                    synchronized (obj1) {
                        Thread.sleep(100);
                        synchronized (obj2) {
                            System.out.println(a + b);
                        }
                    }
                } else {
                    synchronized (obj2) {
                        Thread.sleep(100);
                        synchronized (obj1) {
                            System.out.println(a + b);
                        }
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

public static class Obj1 {
}

public static class Obj2 {
}

```

运行上面代码，可以通过jstack查看到死锁信息：

```

"thread2" #13 prio=5 os_prio=0 tid=0x0000000029225000 nid=0x3c94 waiting for
monitor entry [0x0000000029c9f000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.jvm.visualvm.Demo4$SynAddRunalbe.run(Demo4.java:50)
        - waiting to lock <0x00000007173d40f0> (a com.jvm.visualvm.Demo4$Obj1)
        - locked <0x00000007173d6310> (a com.jvm.visualvm.Demo4$Obj2)
        at java.lang.Thread.run(Thread.java:745)

Locked ownable synchronizers:
- None

"thread1" #12 prio=5 os_prio=0 tid=0x0000000029224800 nid=0x6874 waiting for
monitor entry [0x0000000029b9f000]
    java.lang.Thread.State: BLOCKED (on object monitor)
        at com.jvm.visualvm.Demo4$SynAddRunalbe.run(Demo4.java:43)
        - waiting to lock <0x00000007173d6310> (a com.jvm.visualvm.Demo4$Obj2)
        - locked <0x00000007173d40f0> (a com.jvm.visualvm.Demo4$Obj1)
        at java.lang.Thread.run(Thread.java:745)

Locked ownable synchronizers:
- None

```

thread1持有com.jvm.visualvm.Demo4\$Obj1的锁，等待获取com.jvm.visualvm.Demo4\$Obj2的锁
 thread2持有com.jvm.visualvm.Demo4\$Obj2的锁，等待获取com.jvm.visualvm.Demo4\$Obj1的锁，
 两个线程相互等待获取对方持有的锁，出现死锁。

饥饿死锁的例子

```

package com.jvm.jconsole;

import java.util.concurrent.*;

/**
 * <a href="http://www.itsoku.com/archives">Java干货铺子,只生产干货, 公众号:  

javacode2018</a>
 */
public class ExecutorLock {
    private static ExecutorService single = Executors.newSingleThreadExecutor();

    public static class AnotherCallable implements Callable<String> {
        @Override
        public String call() throws Exception {
            System.out.println("in AnotherCallable");
            return "another success";
        }
    }
}

```

```

public static class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        System.out.println("in MyCallable");
        Future<String> submit = single.submit(new AnotherCallable());
        return "success:" + submit.get();
    }
}

public static void main(String[] args) throws ExecutionException,
InterruptedException {
    MyCallable task = new MyCallable();
    Future<String> submit = single.submit(task);
    System.out.println(submit.get());
    System.out.println("over");
    single.shutdown();
}
}

```

执行代码，输出：

```
in MyCallable
```

使用jstack命令查看线程堆栈信息：

```

"pool-1-thread-1" #12 prio=5 os_prio=0 tid=0x0000000028e3d000 nid=0x58a4 waiting
on condition [0x00000000297ff000]
    java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x0000000717921bf0> (a
java.util.concurrent.FutureTask)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
        at java.util.concurrent.FutureTask.awaitDone(FutureTask.java:429)
        at java.util.concurrent.FutureTask.get(FutureTask.java:191)
        at com.jvm.jconsole.ExecutorLock$MyCallable.call(ExecutorLock.java:25)
        at com.jvm.jconsole.ExecutorLock$MyCallable.call(ExecutorLock.java:20)
        at java.util.concurrent.FutureTask.run(FutureTask.java:266)
        at
java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at
java.util.concurrent.ThreadPoolExecutor$worker.run(ThreadPoolExecutor.java:617)
        at java.lang.Thread.run(Thread.java:745)

Locked ownable synchronizers:
- <0x00000007173f2690> (a java.util.concurrent.ThreadPoolExecutor$Worker)

"main" #1 prio=5 os_prio=0 tid=0x00000000033e4000 nid=0x5f94 waiting on
condition [0x00000000031fe000]
    java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00000007173f1d48> (a
java.util.concurrent.FutureTask)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
        at java.util.concurrent.FutureTask.awaitDone(FutureTask.java:429)

```

```
at java.util.concurrent.FutureTask.get(FutureTask.java:191)
at com.jvm.jconsole.ExecutorLock.main(ExecutorLock.java:32)
```

Locked ownable synchronizers:

- None

```
18
19
20     public static class MyCallable implements Callable<String> {
21         @Override
22         public String call() throws Exception {
23             System.out.println("in MyCallable");
24             Future<String> submit = single.submit(new AnotherCallable());
25             return "success:" + submit.get();
26         }
27     }
28
29     public static void main(String[] args) throws ExecutionException, InterruptedException {
30         MyCallable task = new MyCallable();
31         Future<String> submit = single.submit(task);
32         System.out.println(submit.get());
33         System.out.println("over");
34         single.shutdown();
35     }
36
37
38 }
```

堆栈信息结合图中的代码，可以看出主线程在32行处于等待中，线程池中的工作线程在25行处于等待中，等待获取结果。由于线程池是一个线程，AnotherCallable得不到执行，而被饿死，最终导致了程序死锁的现象。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

第2篇：并发级别

由于临界区的存在，多线程之间的并发必须受到控制。根据控制并发的策略，我们可以把并发的级别分为阻塞、无饥饿、无障碍、无锁、无等待几种。

阻塞

一个线程是阻塞的，那么在其他线程释放资源之前，当前线程无法继续执行。当我们使用synchronized关键字或者重入锁时，我们得到的就是阻塞的线程。

synchronize关键字和重入锁都试图在执行后续代码前，得到临界区的锁，如果得不到，线程就会被挂起等待，直到占有了所需资源为止。

无饥饿(Starvation-Free)

如果线程之间是有优先级的，那么线程调度的时候总是会倾向于先满足高优先级的线程。也就是说，对于同一个资源的分配，是不公平的！图1.7中显示了非公平锁与公平锁两种情况(五角星表示高优先级线程)。对于非公平锁来说，系统允许高优先级的线程插队。这样有可能导致低优先级线程产生饥饿。但如果锁是公平的，按照先来后到的规则，那么饥饿就不会产生，不管新来的线程优先级多高，要想获得资源，就必须乖乖排队，这样所有的线程都有机会执行。



图 1.7 非公平锁与公平锁

无障碍(Obstruction-Free)

无障碍是一种最弱的非阻塞调度。两个线程如果无障碍地执行，那么不会因为临界区的问题导致一方被挂起。换言之，大家都可以大摇大摆地进入临界区了。那么大家一起修改共享数据，把数据改坏了怎么办呢？对于无障碍的线程来说，一旦检测到这种情况，它就会立即对自己所做的修改进行回滚，确保数据安全。但如果数据竞争发生，那么线程就可以顺利完成自己的工作，走出临界区。

如果说阻塞的控制方式是悲观策略，也就是说，系统认为两个线程之间很有可能发生不幸的冲突，因此以保护共享数据为第一优先级，相对来说，非阻塞的调度就是一种乐观的策略。它认为多个线程之间很有可能不会发生冲突，或者说这种概率不大。因此大家都应该无障碍地执行，但是一旦检测到冲突，就应该进行回滚。

从这个策略中也可以看到，无障碍的多线程程序并不一定能顺畅运行。因为当临界区中存在严重的冲突时，所有的线程可能都会不断地回滚自己的操作，而没有一个线程可以走出临界区。这种情况会影响系统的正常执行。所以，我们可能会非常希望在这一堆线程中，至少可以有一个线程能够在有限的时间内完成自己的操作，而退出临界区。至少这样可以保证系统不会在临界区中进行无限的等待。

一种可行的无障碍实现可以依赖一个“一致性标记”来实现。线程在操作之前，先读取并保存这个标记，在操作完成后，再次读取，检查这个标记是否被更改过，如果两者是一致的，则说明资源访问没有冲突。如果不一致，则说明资源可能在操作过程中与其他线程冲突，需要重试操作。而任何对资源有修改操作的线程，在修改数据前，都需要更新这个一致性标记，表示数据不再安全。

数据库中乐观锁，应该比较熟悉，表中需要一个字段version(版本号)，每次更新数据version+1，更新的时候将版本号作为条件进行更新，根据更新影响的行数判断更新是否成功，伪代码如下：

```
1. 查询数据，此时版本号为w_v  
2. 打开事务  
3. 做一些业务操作  
4. update t set version = version+1 where id = 记录id and version = w_v; //此行会返回影响的行数c  
5. if(c>0){  
    //提交事务  
} else{  
    //回滚事务  
}
```

多个线程更新同一条数据的时候，数据库会对当前数据加锁，同一时刻只有一个线程可以执行更新语句。

无锁(Lock-Free)

无锁的并行都是无障碍的。在无锁的情况下，所有的线程都能尝试对临界区进行访问，但不同的是，无锁的并发保证必然有一个线程能够在有限步内完成操作离开临界区。

在无锁的调用中，一个典型的特点是可能会包含一个无穷循环。在这个循环中，线程会不断尝试修改共享变量。如果没有冲突，修改成功，那么程序退出，否则继续尝试修改。但无论如何，无锁的并行总能保证有一个线程是可以胜出的，不至于全军覆没。至于临界区中竞争失败的线程，他们必须不断重试，直到自己获胜。如果运气很不好，总是尝试不成功，则会出现类似饥饿的先写，线程会停止。

下面就是一段无锁的示意代码，如果修改不成功，那么循环永远不会停止。

```
while(!atomicvar.compareAndSet(localvar, localvar+1)){  
    localval = atomicvar.get();  
}
```

无等待

无锁只要求有一个线程可以在有限步内完成操作，而无等待则在无锁的基础上更进一步扩展。它要求所有线程都必须在有限步内完成，这样不会引起饥饿问题。如果限制这个步骤的上限，还可以进一步分解为有界无等待和线程数无关的无等待等几种，他们之间的区别只是对循环次数的限制不同。

一种典型的无等待结果就是RCU(Read Copy Update)。它的基本思想是，对数据的读可以不加控制。因此，所有的读线程都是无等待的，它们既不会被锁定等待也不会引起任何冲突。但在写数据的时候，先获取原始数据的副本，接着只修改副本数据(这就是为什么读可以不加控制)，修改完成后，在合适的时机回写数据。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第3篇：有关并行的两个重要定律

有关为什么要使用并行程序的问题前面已经进行了简单的探讨。总的来说，最重要的应该是处于两个目的。

第一，为了获得更好的性能；

第二，由于业务模型的需要，确实需要多个执行实体。

在这里，我将更加关注第一种情况，也就是有关性能的问题。将串行程序改造为并发程序，一般来说可以提高程序的整体性能，但是究竟能提高多少，甚至说究竟是否真的可以提高，还是一个需要研究的问题。目前，主要有两个定律对这个问题进行解答，一个是Amdahl定律，另外一个是Gustafson定律。

Amdahl(阿姆达尔)定律

Amdahl定律是计算机科学中非常重要的定律。它定义了串行系统并行化后的加速比的计算公式和理论上线。

加速比定义：加速比 = 优化前系统耗时 / 优化后系统耗时

所谓加速比就是优化前耗时与优化后耗时的比值。加速比越高，表明优化效果越明显。图1.8显示了Amdahl公式的推到过程，其中n表示处理器个数，T表示时间，T1表示优化前耗时(也就是只有1个处理器时的耗时)，Tn表示使用n个处理器优化后的耗时。F是程序中只能串行执行的比例。

$$\begin{aligned}
 T_n &= T_1 \left(F + \frac{1}{n} (1-F) \right) \\
 &\downarrow \quad \text{串行比例} \quad \downarrow \quad \text{并行比例} \\
 \text{处理器个数} & \\
 \text{加速比} &= \frac{T_1}{T_n} \rightarrow \frac{T_1}{\text{优化前耗时}} \rightarrow \frac{1}{\text{优化后耗时}}
 \end{aligned}$$

$$\begin{aligned}
 &= \frac{T_1}{T_1 \left(F + \frac{1}{n} (1-F) \right)} \\
 &= \frac{1}{F + \frac{1}{n} (1-F)}
 \end{aligned}$$

图 1.8 Amdahl 公式的推导

根据这个公式，如果CPU处理器数量趋于无穷，那么加速比与系统的串行化比例成反比，如果系统中必须有50%的代码串行执行，那么系统的最大加速比为2。

假设有一个程序分为以下步骤执行，每个执行步骤花费100个单位时间。其中，只有步骤2和步骤5可以并行，步骤1、3、4必须串行，如图1.9所示。在全串行的情况下，系统合计耗时为500个单位时间。

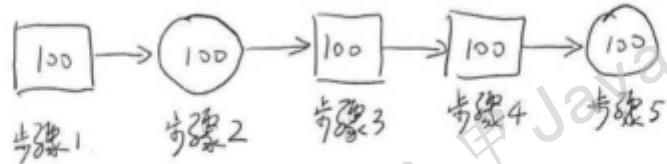


图 1.9 串行工作流程

若步骤2和步骤5并行化，假设在双核处理器上，则有如图1.10所示的处理流程。在这种情况下，步骤2和步骤5的耗时将为50个单位时间。故系统整体耗时为400个单位时间。根据加速比的定义有：

$$\text{加速比} = \text{优化前系统耗时} / \text{优化后系统耗时} = 500/400 = 1.25$$

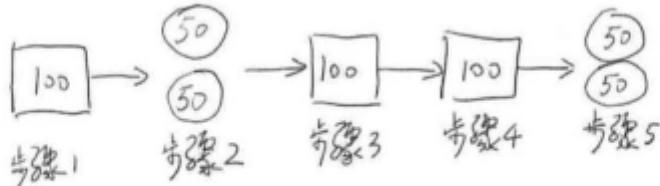


图 1.10 双核处理上的并行化

由于5个步骤中，3个步骤必须串行，因此其串行化比例为3/5=0.6，即 $F = 0.6$ ，且双核处理器的处理器个数N为2。代入加速比公式得：

$$\text{加速比} = 1/(0.6+(1-0.6)/2)=1.25$$

在极端情况下，假设并行处理器个数为无穷大，则有如图1.11所示的处理过程。步骤2和步骤5的处理时间趋于0。即使这样，系统整体耗时依然大于300个单位时间。使用加速比计算公式，N趋于无穷大，有加速比 $= 1/F$ ，且 $F=0.6$ ，故有加速比=1.67。即加速比的极限为 $500/300=1.67$ 。

由此可见，为了提高系统的速度，仅增加CPU处理的数量并不一定能起到有效的作用。需要从根本上修改程序的串行行为，提高系统内可并行化的模块比重，在此基础上，合理增加并行处理器数量，才能以最小的投入，得到最大的加速比。

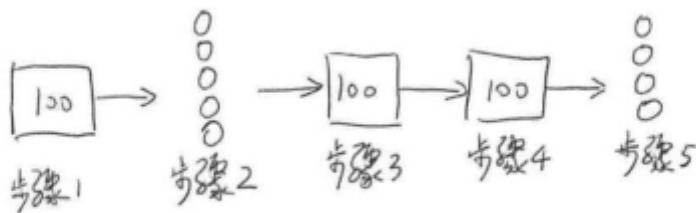


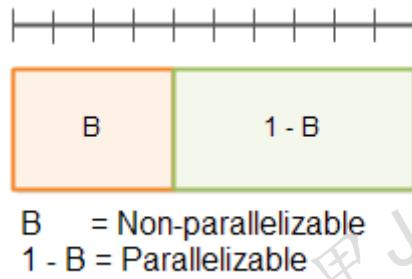
图 1.11 极端情况下的并行化

注意：根据Amdahl定律，使用多核CPU对系统进行优化，优化的效果取决于CPU的数量，以及系统中串行化程序的比例。CPU数量越多，串行化比例越低，则优化效果越好。仅提高CPU数量而不降低程序的串行化比例，也无法提高系统的性能。

阿姆达尔定律图示

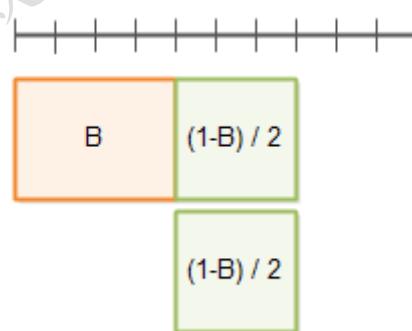
为了更好地理解阿姆达尔定律，我会尝试演示这个定律是如何诞生的。

首先，一个程序可以被分割为两部分，一部分为不可并行部分B，一部分为可并行部分 $1 - B$ 。如下图：

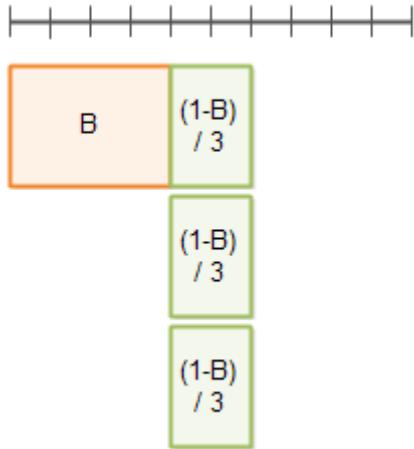


在顶部被带有分割线的那条直线代表总时间 $T(1)$ 。

下面你可以看到在并行因子为2的情况下执行时间：

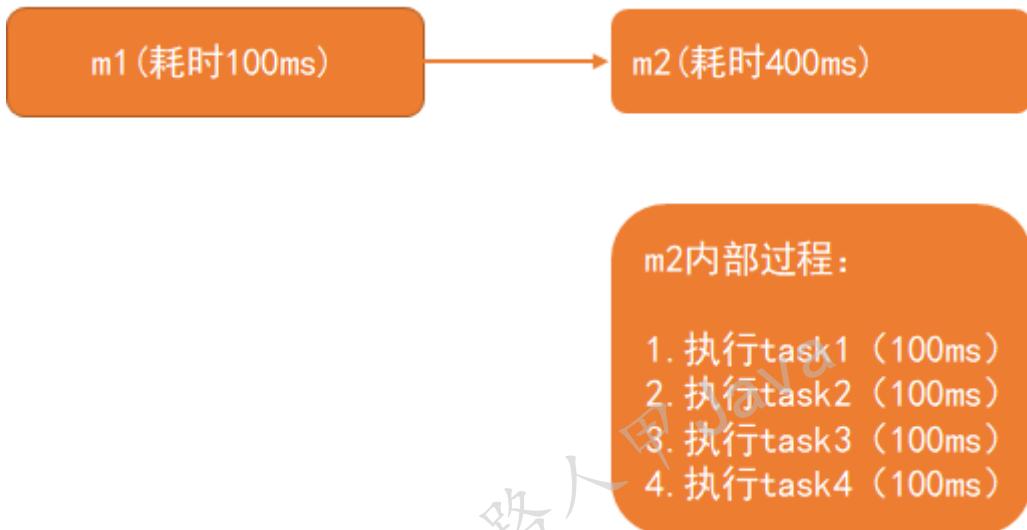


并行因子为3的情况：



举个例子

一个业务会串行调用2个方法，`m1`, `m2`, `m1`耗时100ms, `m2`耗时400ms, `m2`内部串行执行了4个无依赖的任务，每个任务100ms，如下图：



`m2`内部的4个任务无依赖的，即可以并行进行处理，4个任务同时并行，当cpu数量大于等于4的时候，可以让4个任务同时进行，此时`m2`耗时最小，即100ms，cpu为2个的时候，同时只能执行2个任务，其他2个任务处于等待cpu分配时间片状态，此时`m2`耗时200ms；当cpu超过4个的时候，或者趋于无限大的时候，`m2`耗时还是100ms，此时cpu数量再怎么增加对性能也没有提升了，此时需要提升的是任务可以并行的数量。

从阿姆达尔定律可以看出，程序的可并行化部分可以通过使用更多的硬件（更多的线程或CPU）运行更快。对于不可并行化的部分，只能通过优化代码来达到提速的目的。因此，你可以通过优化不可并行化部分来提高你的程序的运行速度和并行能力。你可以对不可并行化在算法上做一点改动，如果有可能，你也可以把一些移到可并行化放的部分。

Gustafson定律

Gustafson定律也试图说明处理器个数、串行化比例和加速比之间的关系，如图1.12所示，但是Gustafson定律和Amdahl定律的角度不同。同样，加速比都被定义为优化前的系统耗时除以优化后的系统耗时。

串行时间 $\xrightarrow{\text{并行时间}} \text{并行时间}$
 执行时间: $a+b$

总执行时间: $a+n \cdot b$ $\xrightarrow{\text{处理器个数}}$

加速比: $(a+n \cdot b) / (a+b)$

定义: $F = a / (a+b)$ 串行比例
 则 加速比 $S(n) = \frac{a+n \cdot b}{a+b} = \frac{a}{a+b} + \frac{n \cdot b}{a+b}$
 $= F + n \cdot \left(\frac{a+b-a}{a+b} \right) = F + n \left(1 - \frac{a}{a+b} \right)$
 $= F + n(1-F) = F + n - nF$
 $= n - F(n-1)$

图 1.12 Gustafson 定律的推导

根据Gustafson定律，我们可以更容易地发现，如果串行化比例很小，并行化比例很大，那么加速比就是处理器的个数。只要不断地累加处理器，就能获得更快的速度。

Amdahl定律和Gustafson定律结论有所不同，并不是说其中有个是错误的，只是二者从不同的角度去看待问题的结果，他们的侧重点有所不同。

Amdahl强调：当串行化比例一定时，加速比是有上限的，不管你堆叠多少个CPU参与计算，都不能突破这个上限。**Gustafson定律关系的是：**如果可被并行化的代码所占比例足够大，那么加速比就能随着CPU的数量线性增长。

总的来说，提升性能的方法：想办法提升系统并行的比例，同时增加CPU数量。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第4篇：JMM相关的一些概念

JMM(java内存模型)，由于并发程序要比串行程序复杂很多，其中一个重要原因是并发程序中数据访问**一致性和安全性**将会受到严重挑战。**如何保证一个线程可以看到正确的数据呢？**这个问题看起来很白痴。对于串行程序来说，根本就是小菜一碟，如果你读取一个变量，这个变量的值是1，那么你读取到的一定是1，就是这么简单的问题在并行程序中居然变得复杂起来。事实上，如果不加控制地任由线程胡乱并行，即使原本是1的数值，你也可能读到2。因此我们需要在深入了解并行机制的前提下，再定义一种规则，保证多个线程间可以有小弟，正确地协同工作。而JMM也就是为此而生的。

JMM关键技术点都是围绕着多线程的原子性、可见性、有序性来建立的。我们需要先了解这些概念。

原子性

原子性是指操作是不可分的，要么全部一起执行，要么不执行。在java中，其表现在对于共享变量的某些操作，是不可分的，必须连续的完成。比如a++，对于共享变量a的操作，实际上会执行3个步骤：

1.读取变量a的值，假如a=1 2.a的值+1，为2 3.将2值赋值给变量a，此时a的值应该为2

这三个操作中任意一个操作，a的值如果被其他线程篡改了，那么都会出现我们不希望出现的结果。所以必须保证这3个操作是原子性的，在操作a++的过程中，其他线程不会改变a的值，如果在上面的过程中出现其他线程修改了a的值，在满足原子性的原则下，上面的操作应该失败。

java中实现原子操作的方法大致有2种：**锁机制、无锁CAS机制**，后面的章节中会有介绍。

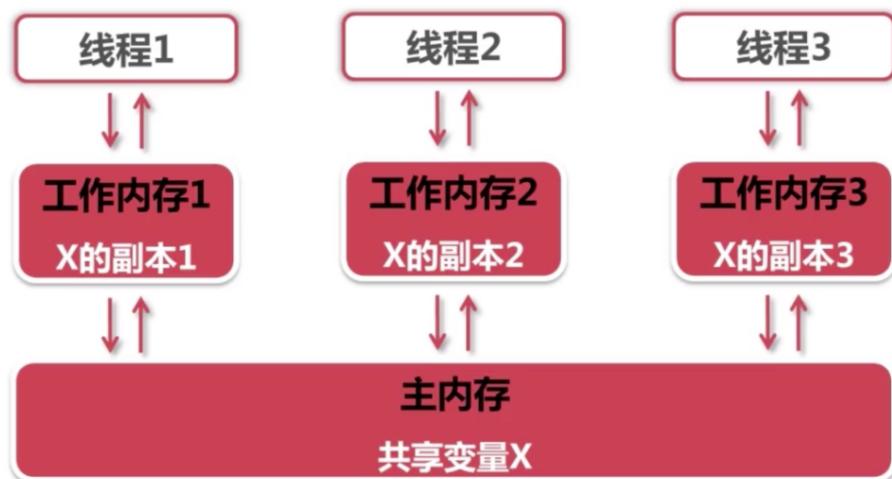
可见性

可见性是指一个线程对共享变量的修改，对于另一个线程来说是否是可以看到的。有些同学会说修改同一个变量，那肯定是可以看到的，难道线程眼盲了？

为什么会出现这种问题呢？

看一下java线程内存模型：

Java内存模型 (JMM)



- 我们定义的所有变量都储存在 主内存 中
- 每个线程都有自己 独立的工作内存，里面保存该线程使用到的变量的副本（主内存中该变量的一份拷贝）
- 线程对共享变量所有的操作都必须在自己的工作内存中进行，不能直接从主内存中读写（不能越级）
- 不同线程之间也无法直接访问其他线程的工作内存中的变量，线程间变量值的传递需要通过主内存 来进行。（同级不能相互访问）

线程需要修改一个共享变量X，需要先把X从主内存复制一份到线程的工作内存，在自己的工作内存中修改完毕之后，再从工作内存中回写到主内存。如果线程对变量的操作没有刷写回主内存的话，仅仅改变了自己的工作内存的变量的副本，那么对于其他线程来说是不可见的。而如果另一个变量没有读取主内存中的新的值，而是使用旧的值的话，同样的也可以列为不可见。

共享变量可见性的实现原理：

线程A对共享变量的修改要被线程B及时看到的话，需要进过以下步骤：

1.线程A在自己的工作内存中修改变量之后，需要将变量的值刷新到主内存中 2.线程B要把主内存中变量的值更新到工作内存中

关于线程可见性的控制，可以使用**volatile**、**synchronized**、**锁**来实现，后面章节会有详细介绍。

有序性

有序性指的是程序按照代码的先后顺序执行。

为了性能优化，编译器和处理器会进行指令冲排序，有时候会改变程序语句的先后顺序，比如程序。

```
int a = 1; //1
int b = 20; //2
int c = a + b; //3
```

编译器优化后可能变成

```
int b = 20; //1
int a = 1; //2
int c = a + b; //3
```

上面这个例子中，编译器调整了语句的顺序，但是不影响程序的最终结果。

在单例模式的实现上有一种双重检验锁定的方式，代码如下：

```
public class Singleton {  
    static Singleton instance;  
    static Singleton getInstance(){  
        if (instance == null) {  
            synchronized(Singleton.class) {  
                if (instance == null)  
                    instance = new Singleton();  
            }  
        }  
        return instance;  
    }  
}
```

我们先看 `instance = new Singleton();`

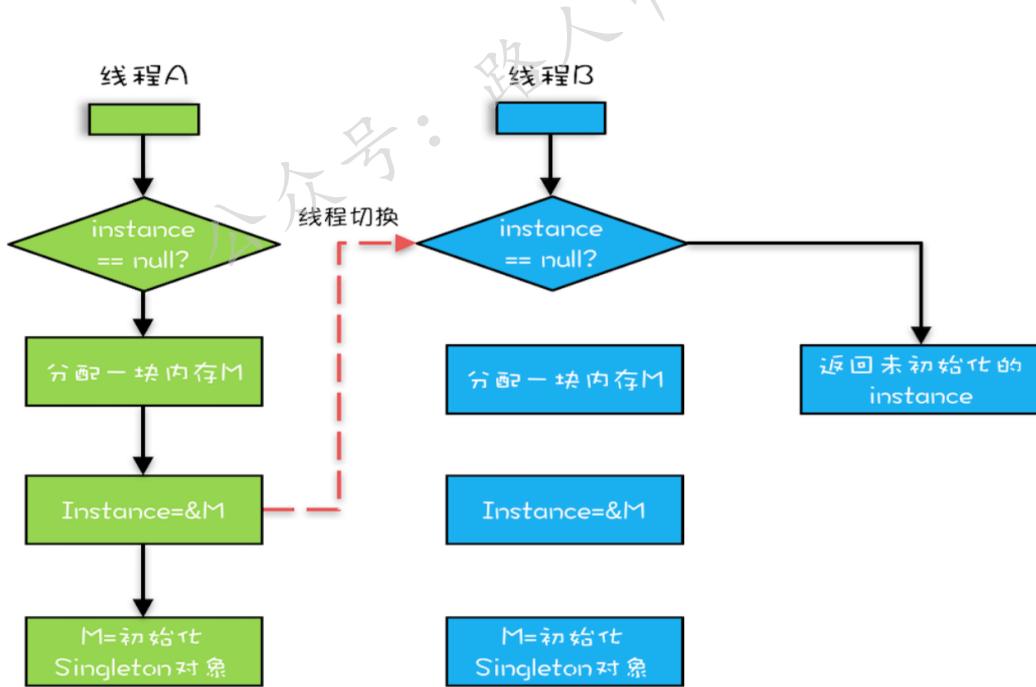
未被编译器优化的操作：

1. 指令1：分配一块内存M
2. 指令2：在内存M上初始化Singleton对象
3. 指令3：将M的地址赋值给instance变量

编译器优化后的操作指令：

1. 指令1：分配一块内存S
2. 指令2：将M的地址赋值给instance变量
3. 指令3：在内存M上初始化Singleton对象

现在有2个线程，刚好执行的代码被编译器优化过，过程如下：



双重检查创建单例的异常执行路径

最终线程B获取的instance是没有初始化的，此时去使用instance可能会产生一些意想不到的错误。

现在比较好的做法就是采用静态内部内的方式实现：

```
public class SingletonDemo {  
    private SingletonDemo() {  
    }  
    private static class SingletonDemoHandler{  
        private static SingletonDemo instance = new SingletonDemo();  
    }  
    public static SingletonDemo getInstance() {  
        return SingletonDemoHandler.instance;  
    }  
}
```

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第5篇：深入理解进程和线程

进程

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。程序是指令、数据及其组织形式的描述，进程是程序的实体。

进程具有的特征：

- **动态性：**进程是程序的一次执行过程，是临时的，有生命期的，是动态产生，动态消亡的
- **并发性：**任何进程都可以同其他进行一起并发执行
- **独立性：**进程是系统进行资源分配和调度的一个独立单位
- **结构性：**进程由程序、数据和进程控制块三部分组成

我们经常使用windows系统，经常会看见.exe后缀的文件，双击这个.exe文件的时候，这个文件中的指令就会被系统加载，那么我们就能得到一个关于这个.exe程序的进程。进程是“活”的，或者说是正在被执行的。

window中打开任务管理器，可以看到当前系统中正在运行的进程，如下图：

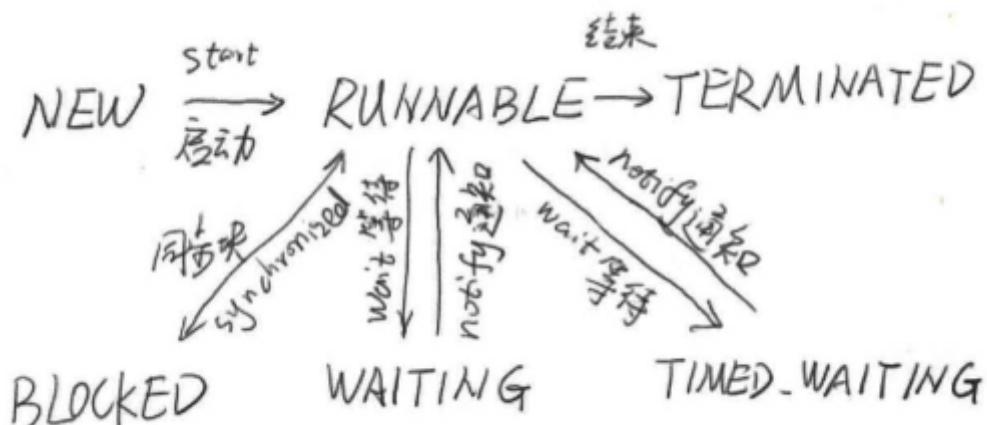
The screenshot shows the Windows Task Manager with the '进程' (Processes) tab selected. The table lists processes and their CPU usage. The 'Docker.Watchguard.exe' process is currently selected. The columns are '名称' (Name), '状态' (Status), and 'CPU' (CPU Usage). The CPU column is sorted by value, with '13%' at the top.

名称	状态	CPU
Java Update Scheduler (32 位)		0%
Java Update Checker (32 位)		0%
> Hyper-V 主机计算服务		0%
> Flash Helper Service (32 位)		0%
> dockerd.exe		0%
Docker.Watchguard.exe		0%
> Docker.Service		0%
Docker Desktop		0%
Device Association Framework		0%
Device Association Framework		0%
> Dell SonicWALL Global VPN ...		0%
CTF 加载程序		0%
> Cortana (小娜) (2)	等待	0%
> Component Host Service		0%

线程

线程是轻量级的进程，是程序执行的最小单元，使用多线程而不是多进程去进行并发程序的设计，是因为线程间的切换和调度的成本远远小于进程。

我们用一张图来看一下线程的状态图：



线程的所有状态在java.lang.Thread中的State枚举中有定义，如：

```
public enum State {  
    NEW,  
    RUNNABLE,  
    BLOCKED,  
    WAITING,  
    TIMED_WAITING,  
    TERMINATED;  
}
```

线程几个状态的介绍：

- **New**: 表示刚刚创建的线程，这种线程还没有开始执行
- **RUNNABLE**: 运行状态，线程的start()方法调用后，线程会处于这种状态
- **BLOCKED**: 阻塞状态。当线程在执行的过程中遇到了synchronized同步块，但这个同步块被其他线程已获取还未释放时，当前线程将进入阻塞状态，会暂停执行，直到获取到锁。当线程获取到锁之后，又会进入到运行状态 (RUNNABLE)
- **WAITING**: 等待状态。和TIME_WAITING都表示等待状态，区别是WAITING会进入一个无时间限制的等，而TIME_WAITING会进入一个有限的时间等待，那么等待的线程究竟在等什么呢？一般来说，WAITING的线程正式在等待一些特殊的事件，比如，通过wait()方法等待的线程在等待notify()方法，而通过join()方法等待的线程则会等待目标线程的终止。一旦等到期望的事件，线程就会再次进入RUNNABLE运行状态。
- **TERMINATED**: 表示结束状态，线程执行完毕之后进入结束状态。

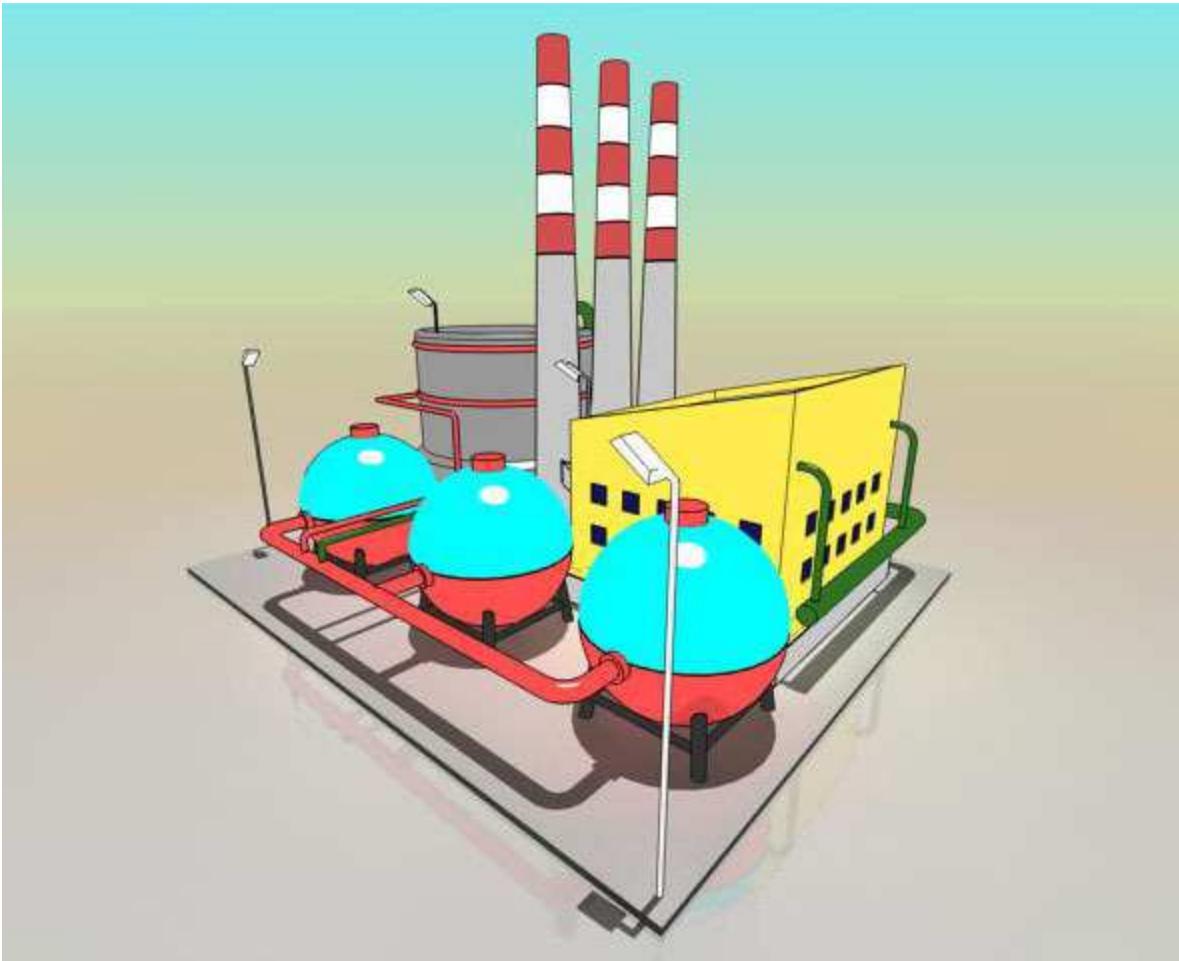
注意：从NEW状态出发后，线程不能在回到NEW状态，同理，处理TERMINATED状态的线程也不能在回到RUNNABLE状态

进程与线程的一个简单解释

进程 (process) 和线程 (thread) 是操作系统的基本概念，但是它们比较抽象，不容易掌握。

公众号：路人甲Java

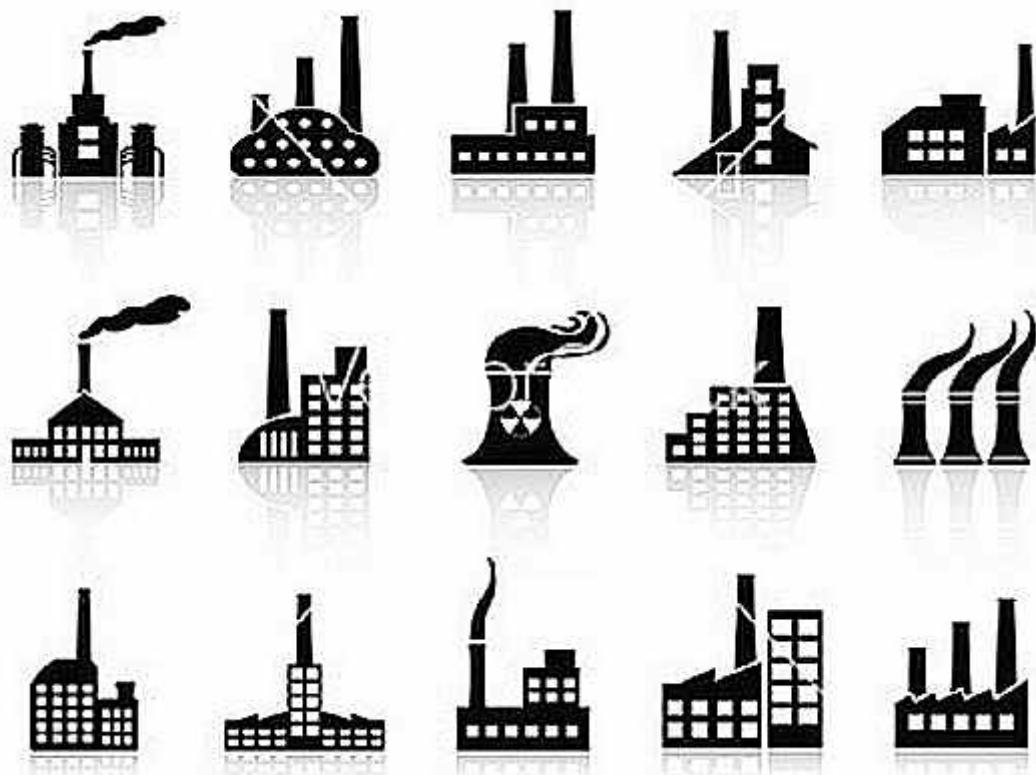
1.计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。



2.假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，其他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务。



3. 进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。



公众号：路人甲 Java

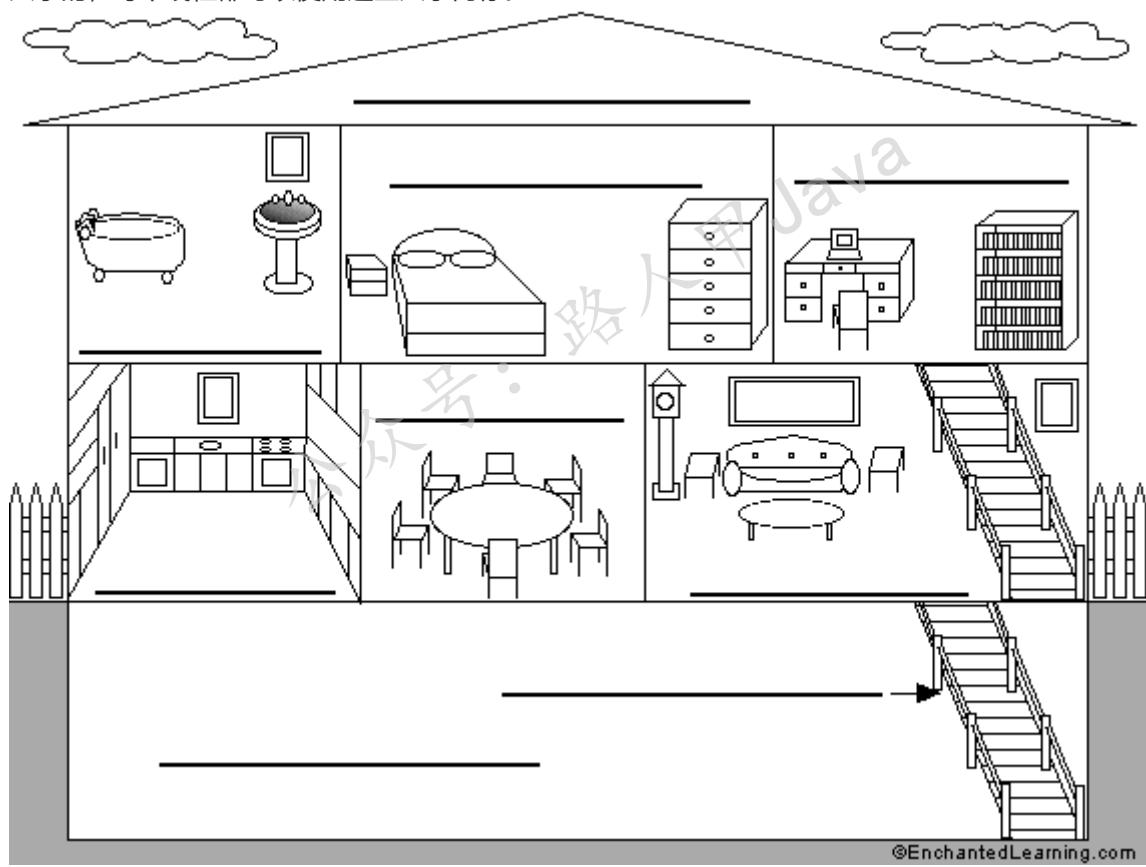
4.一个车间里，可以有很多工人。他们协同完成一个任务。



5.线程就好比车间里的工人。一个进程可以包括多个线程。



6.车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。



7.可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。



8.一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。



9. 还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。



10. 这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做"信号量" (Semaphore)，用来保证多个线程不会互相冲突。



11. 操作系统的设计，因此可以归结为三点：（1）以多进程形式，允许多个任务同时运行；（2）以多线程形式，允许单个任务分成不同的部分运行；（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)

4. [Maven高手系列 \(共10篇\)](#)
5. [Mybatis系列 \(共12篇\)](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第6篇：线程的基本操作

新建线程

新建线程很简单。只需要使用new关键字创建一个线程对象，然后调用它的start()启动线程即可。

```
Thread thread1 = new Thread1();
t1.start();
```

那么线程start()之后，会干什么呢？线程有个run()方法，start()会创建一个新的线程并让这个线程执行run()方法。

这里需要注意，下面代码也能通过编译，也能正常执行。但是，却不能新建一个线程，而是在当前线程中调用run()方法，将run方法只是作为一个普通的方法调用。

```
Thread thread = new Thread1();
thread1.run();
```

所以，希望大家注意，调用start方法和直接调用run方法的区别。

start方法是启动一个线程，run方法只会在垫钱线程中串行的执行run方法中的代码。

默认情况下，线程的run方法什么都没有，启动一个线程之后马上就结束了，所以如果你需要线程做点什么，需要把您的代码写到run方法中，所以必须重写run方法。

```
Thread thread1 = new Thread() {
    @Override
    public void run() {
        System.out.println("Hello, 我是一个线程！");
    }
};
thread1.start();
```

上面是使用匿名内部类实现的，重写了Thread的run方法，并且打印了一条信息。我们可以通过继承 Thread类，然后重写run方法，来自定义一个线程。但考虑java是单继承的，从扩展性上来说，我们实现一个接口来自定义一个线程更好一些，java中刚好提供了Runnable接口来自定义一个线程。

```
@FunctionalInterface  
public interface Runnable {  
    public abstract void run();  
}
```

Thread类有一个非常重要的构造方法：

```
public Thread(Runnable target)
```

我们在看一下Thread的run方法：

```
public void run() {  
    if (target != null) {  
        target.run();  
    }  
}
```

当我们启动线程的start方法之后，线程会执行run方法，run方法中会调用Thread构造方法传入的target的run方法。

实现Runnable接口是比较常见的做法，也是推荐的做法。

终止线程

一般来说线程执行完毕就会结束，无需手动关闭。但是如果我们要想关闭一个正在运行的线程，有什么方法呢？可以看一下Thread类中提供了一个stop()方法，调用这个方法，就可以立即将一个线程终止，非常方便。

```
package com.itsoku.chat01;  
  
import lombok.extern.slf4j.Slf4j;  
import java.util.concurrent.TimeUnit;  
  
/**  
 * <b>description</b>: <br>  
 * <b>time</b>: 2019/7/12 17:18 <br>  
 * <b>author</b>: 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */  
@Slf4j  
public class Demo01 {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread1 = new Thread() {  
            @Override  
            public void run() {  
                log.info("start");  
                boolean flag = true;  
                while (flag) {  
                    ;  
                }  
            }  
        };  
        thread1.start();  
        TimeUnit.SECONDS.sleep(2);  
        thread1.interrupt();  
    }  
}
```

```

        }
        log.info("end");
    }
};

thread1.setName("thread1");
thread1.start();
//当前线程休眠1秒
TimeUnit.SECONDS.sleep(1);
//关闭线程thread1
thread1.stop();
//输出线程thread1的状态
log.info("{}" , thread1.getState());
//当前线程休眠1秒
TimeUnit.SECONDS.sleep(1);
//输出线程thread1的状态
log.info("{}" , thread1.getState());
}
}

```

运行代码，输出：

```

18:02:15.312 [thread1] INFO com.itsoku.chat01.Demo01 - start
18:02:16.311 [main] INFO com.itsoku.chat01.Demo01 - RUNNABLE
18:02:17.313 [main] INFO com.itsoku.chat01.Demo01 - TERMINATED

```

代码中有个死循环，调用stop方法之后，线程thread1的状态变为TERMINATED（结束状态），线程停止了。

我们使用idea或者eclipse的时候，会发现这个方法是一个废弃的方法，也就是说，在将来，jdk可能就会移除该方法。

stop方法为何会被废弃而不推荐使用？stop方法过于暴力，强制把正在执行的方法停止了。

大家是否遇到过这样的场景：电力系统需要维修，此时咱们正在写代码，维修人员直接将电源关闭了，代码还没保存的，是不是很崩溃，这种方式就像直接调用线程的stop方法类似。线程正在运行过程中，被强制结束了，可能会导致一些意想不到的后果。可以给大家发送一个通知，告诉大家保存一下手头的工作，将电脑关闭。

线程中断

在java中，线程中断是一种重要的线程写作机制，从表面上理解，中断就是让目标线程停止执行的意思，实际上并非完全如此。在上面中，我们已经详细讨论了stop方法停止线程的坏处，jdk中提供了更好的中断线程的方法。严格的说，线程中断并不会使线程立即退出，而是给线程发送一个通知，告知目标线程，有人希望你退出了！至于目标线程接收到通知之后如何处理，则完全由目标线程自己决定，这点很重要，如果中断后，线程立即无条件退出，我们又会到stop方法的老问题。

Thread提供了3个与线程中断有关的方法，这3个方法容易混淆，大家注意下：

```

public void interrupt() //中断线程
public boolean isInterrupted() //判断线程是否被中断
public static boolean interrupted() //判断线程是否被中断，并清除当前中断状态

```

`interrupt()`方法是一个**实例方法**，它通知目标线程中断，也就是设置中断标志位为true，中断标志位表示当前线程已经被中断了。`isInterrupted()`方法也是一个**实例方法**，它判断当前线程是否被中断（通过检查中断标志位）。最后一个方法`interrupted()`是一个**静态方法**，返回boolean类型，也是用来判断当前线程是否被中断，但是同时会清除当前线程的中断标志位的状态。

```
while (true) {
    if (this.isInterrupted()) {
        System.out.println("我要退出了!");
        break;
    }
}
};

thread1.setName("thread1");
thread1.start();
TimeUnit.SECONDS.sleep(1);
thread1.interrupt();
```

上面代码中有个死循环，`interrupt()`方法被调用之后，线程的中断标志将被置为true，循环体中通过检查线程的中断标志是否为ture (`this.isInterrupted()`) 来判断线程是否需要退出了。

再看一种中断的方法：

```
static volatile boolean isStop = false;

public static void main(String[] args) throws InterruptedException {
    Thread thread1 = new Thread() {
        @Override
        public void run() {
            while (true) {
                if (isStop) {
                    System.out.println("我要退出了!");
                    break;
                }
            }
        }
    };
    thread1.setName("thread1");
    thread1.start();
    TimeUnit.SECONDS.sleep(1);
    isStop = true;
}
```

代码中通过一个变量`isStop`来控制线程是否停止。

通过变量控制和线程自带的`interrupt`方法来中断线程有什么区别呢？

如果一个线程调用了`sleep`方法，一直处于休眠状态，通过变量控制，还可以中断线程么？大家可以思考一下。

此时只能使用线程提供的`interrupt`方法来中断线程了。

```
public static void main(String[] args) throws InterruptedException {
    Thread thread1 = new Thread() {
        @Override
        public void run() {
```

```

        while (true) {
            //休眠100秒
            try {
                TimeUnit.SECONDS.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("我要退出了!");
            break;
        }
    }
};

thread1.setName("thread1");
thread1.start();
TimeUnit.SECONDS.sleep(1);
thread1.interrupt();
}

```

调用interrupt()方法之后，线程的sleep方法将会抛出 InterruptedException 异常。

```

Thread thread1 = new Thread() {
    @Override
    public void run() {
        while (true) {
            //休眠100秒
            try {
                TimeUnit.SECONDS.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            if (this.isInterrupted()) {
                System.out.println("我要退出了!");
                break;
            }
        }
    }
};

```

运行上面的代码，发现程序无法终止。为什么？

代码需要改为：

```

Thread thread1 = new Thread() {
    @Override
    public void run() {
        while (true) {
            //休眠100秒
            try {
                TimeUnit.SECONDS.sleep(100);
            } catch (InterruptedException e) {
                this.interrupt();
                e.printStackTrace();
            }
            if (this.isInterrupted()) {
                System.out.println("我要退出了!");
                break;
            }
        }
    }
};

```

```
    }  
}  
};
```

上面代码可以终止。

注意：sleep方法由于中断而抛出异常之后，线程的中断标志会被清除（置为false），所以在异常中需要执行this.interrupt()方法，将中断标志位置为true

等待 (wait) 和通知 (notify)

为了支持多线程之间的协作，JDK提供了两个非常重要的方法：等待wait()方法和通知notify()方法。这2个方法并不是在Thread类中的，而是在Object类中定义的。这意味着所有的对象都可以调用者两个方法。

```
public final void wait() throws InterruptedException;  
public final native void notify();
```

当在一个对象实例上调用wait()方法后，当前线程就会在这个对象上等待。这是什么意思？比如在线程A中，调用了obj.wait()方法，那么线程A就会停止继续执行，转为等待状态。等待到什么时候结束呢？线程A会一直等到其他线程调用obj.notify()方法为止，这时，obj对象成为了多个线程之间的有效通信手段。

那么wait()方法和notify()方法是如何工作的呢？如图2.5展示了两者的工作过程。如果一个线程调用了object.wait()方法，那么它就会进入object对象的等待队列。这个队列中，可能会有多个线程，因为系统可能运行多个线程同时等待某一个对象。当object.notify()方法被调用时，它就会从这个队列中随机选择一个线程，并将其唤醒。这里希望大家注意一下，这个选择是不公平的，并不是先等待线程就会优先被选择，这个选择完全是随机的。

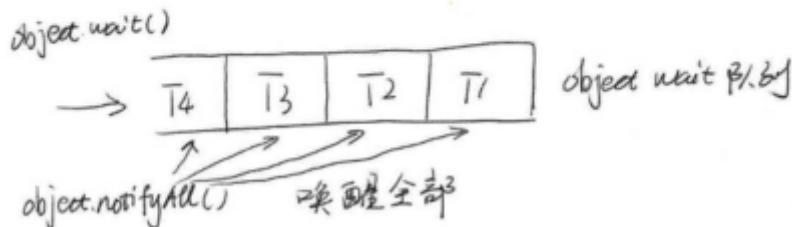
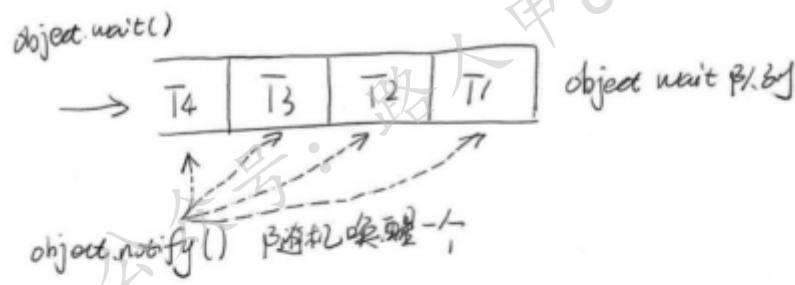


图 2.5 wait()方法和 notify()方法的工作过程

除notify()方法外，Object独享还有一个notifyAll()方法，它和notify()方法的功能类似，不同的是，它会唤醒在这个等待队列中所有等待的线程，而不是随机选择一个。

这里强调一点，Object.wait()方法并不能随便调用。它必须包含在对应的synchronize语句汇总，无论是wait()方法或者notify()方法都需要首先获取目标独享的一个监视器。图2.6显示了wait()方法和notify()方法的工作流程细节。其中T1和T2表示两个线程。T1在正确执行wait()方法钱，必须获得object对象的监视器。而wait()方法在执行后，会释放这个监视器。这样做的目的是使其他等待在object对象上的线程不至于因为T1的休眠而全部无法正常执行。

线程T2在notify()方法调用前，也必须获得object对象的监视器。所幸，此时T1已经释放了这个监视器，因此，T2可以顺利获得object对象的监视器。接着，T2执行了notify()方法尝试唤醒一个等待线程，这里假设唤醒了T1。T1在被唤醒后，要做的第一件事并不是执行后续代码，而是要尝试重新获得object对象的监视器，而这个监视器也正是T1在wait()方法执行前所持有的那个。如果暂时无法获得，则T1还必须等待这个监视器。当监视器顺利获得后，T1才可以在真正意义上继续执行。



图 2.6 wait()方法和 notify()方法的工作流程细节

给大家上个例子：

```
package com.itsoku.chat01;

/**
 * <b>description</b>: <br>
 * <b>time</b>: 2019/7/12 17:18 <br>
 * <b>author</b>: 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo06 {
    static Object object = new Object();

    public static class T1 extends Thread {
        @Override
        public void run() {
            synchronized (object) {
                System.out.println(System.currentTimeMillis() + ":T1 start!");
                try {
                    System.out.println(System.currentTimeMillis() + ":T1 wait
for object");
                    object.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```

        System.out.println(system.currentTimeMillis() + ":T1 end!");
    }
}

public static class T2 extends Thread {
    @Override
    public void run() {
        synchronized (object) {
            System.out.println(system.currentTimeMillis() + ":T2 start,
notify one thread!");
            object.notify();
            System.out.println(system.currentTimeMillis() + ":T2 end!");
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    new T1().start();
    new T2().start();
}
}

```

运行结果：

```

1562934497212:T1 start!
1562934497212:T1 wait for object
1562934497212:T2 start, notify one thread!
1562934497212:T2 end!
1562934499213:T1 end!

```

注意下打印结果，T2调用notify方法之后，T1并不能立即继续执行，而是要等待T2释放objec投递锁之后，T1重新成功获取锁后，才能继续执行。因此最后2行日志相差了2秒（因为T2调用notify方法后休眠了2秒）。

注意：Object.wait()方法和Thread.sleep()方法都可以让现场等待若干时间。除wait()方法可以被唤醒外，另外一个主要的区别就是wait()方法会释放目标对象的锁，而Thread.sleep()方法不会释放锁。

再给大家讲解一下wait(), notify(), notifyAll(), 加深一下理解：

可以这么理解，obj对象上有2个队列，如图1，**q1：等待队列，q2：准备获取锁的队列**；两个队列都为空。

图1

obj.wait()过程:

```
synchronize(obj){  
    obj.wait();  
}
```

假如有3个线程，t1、t2、t3同时执行上面代码，t1、t2、t3会进入q2队列，如图2，进入q2的队列的这些线程才有资格去争抢obj的锁，假设t1争抢到了，那么t2、t3机型在q2中等待着获取锁，t1进入代码块执行wait()方法，此时t1会进入q1队列，然后系统会通知q2队列中的t2、t3去争抢obj的锁，抢到之后过程如t1的过程。最后t1、t2、t3都进入了q1队列，如图3。

q1队列	q2队列
	t1
	t2
	t3

2

q1队列	q2队列
t1	
t2	
t3	

图3

上面过程之后，又来了线程t4执行了notify()方法，如下：**

```
synchronize(obj){  
    obj.notify();  
}
```

t4会获取到obj的锁，然后执行notify()方法，系统会从q1队列中随机取一个线程，将其加入到q2队列，假如t2运气比较好，被随机到了，然后t2进入了q2队列，如图4，进入q2的队列的锁才有资格争抢obj的锁，t4线程执行完毕之后，会释放obj的锁，此时队列q2中的t2会获取到obj的锁，然后继续执行，执行完毕之后，q1中包含t1、t3，q2队列为空，如图5

图4

图5

接着又来了个t5队列，执行了notifyAll()方法，如下：

```
synchronize(obj){  
    obj.notifyAll();  
}
```

2. 调用obj.wait()方法，当前线程会加入队列queue1，然后会释放obj对象的锁

t5会获取到obj的锁，然后执行notifyAll()方法，系统会将队列q1中的线程都移到q2中，如图6，t5线程执行完毕之后，会释放obj的锁，此时队列q2中的t1、t3会争抢obj的锁，争抢到的继续执行，未增强到的带锁释放之后，系统会通知q2中的线程继续争抢锁，然后继续执行，最后两个队列中都为空了。

q1队列	q2队列
	t1
	t3

图6

挂起 (suspend) 和继续执行 (resume) 线程

Thread类中还有2个方法，即**线程挂起(suspend)**和**继续执行(resume)**，这2个操作是一对相反的操作，被挂起的线程，必须要等到resume()方法操作后，才能继续执行。系统中已经标注着2个方法过时了，不推荐使用。

系统不推荐使用suspend()方法去挂起线程是因为suspend()方法导致线程暂停的同时，并不会释放任何锁资源。此时，其他任何线程想要访问被它占用的锁时，都会被牵连，导致无法正常运行（如图2.7所示）。直到在对应的线程上进行了resume()方法操作，被挂起的线程才能继续，从而其他所有阻塞在相关锁上的线程也可以继续执行。但是，如果resume()方法操作意外地在suspend()方法前就被执行了，那么被挂起的线程可能很难有机会被继续执行了。并且，更严重的是：它所占用的锁不会被释放，因此可能会导致整个系统工作不正常。而且，对于被挂起的线程，从它线程的状态上看，居然还是**Runnable**状态，这也会影响我们对系统当前状态的判断。

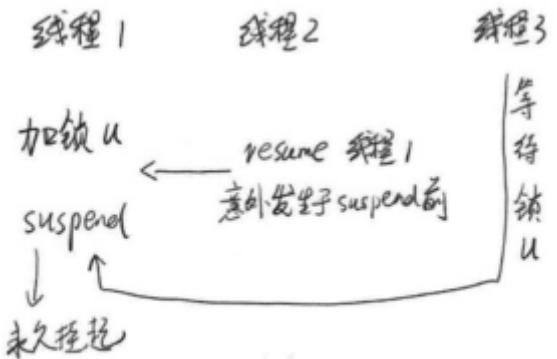


图 2.7 suspend()方法导致线程进入类似死锁的状态

上个例子：

```
/**
 * <b>description</b>: <br>
 * <b>time</b>: 2019/7/12 17:18 <br>
 * <b>author</b>: 微信公众号: 路人甲Java, 专注于java技术分享(带你玩转爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等), 喜欢请关注!
 */
public class Demo07 {
    static Object object = new Object();

    public static class T1 extends Thread {
        public T1(string name) {
            super(name);
        }

        @Override
        public void run() {
            synchronized (object) {
                System.out.println("in " + this.getName());
                Thread.currentThread().suspend();
            }
        }
    }

    public static void main(string[] args) throws InterruptedException {
        T1 t1 = new T1("t1");
        t1.start();
        Thread.sleep(100);
        T1 t2 = new T1("t2");
        t2.start();
        t1.resume();
        t2.resume();
        t1.join();
        t2.join();
    }
}
```

运行代码输出：

```
in t1
in t2
```

我们会发现程序不会结束，线程t2被挂起了，导致程序无法结束，使用jstack命令查看线程堆栈信息可以看到：

```
"t2" #13 prio=5 os_prio=0 tid=0x000000002796c000 nid=0xa3c runnable  
[0x000000002867f000]  
    java.lang.Thread.State: RUNNABLE  
        at java.lang.Thread.suspend0(Native Method)  
        at java.lang.Thread.suspend(Thread.java:1029)  
        at com.itsoku.chat01.Demo07$T1.run(Demo07.java:20)  
        - locked <0x0000000717372fc0> (a java.lang.Object)
```

发现t2线程在**suspend0**处被挂起了，t2的状态竟然还是RUNNABLE状态，线程明明被挂起了，状态还是运行中容易导致我们对当前系统进行误判，代码中已经调用**resume()**方法了，但是由于时间先后顺序的缘故，**resume**并没有生效，这导致了t2永远滴被挂起了，并且永远占用了object的锁，这对于系统来说可能是致命的。

等待线程结束 (join) 和谦让 (yield)

很多时候，一个线程的输入可能非常依赖于另外一个或者多个线程的输出，此时，这个线程就需要等待依赖的线程执行完毕，才能继续执行。jdk提供了**join()**操作来实现这个功能。如下所示，显示了2个**join()**方法：

```
public final void join() throws InterruptedException;  
public final synchronized void join(long millis) throws InterruptedException;
```

第1个方法表示无限等待，它会一直只是当前线程。知道目标线程执行完毕。

第2个方法有个参数，用于指定等待时间，如果超过了给定的时间目标线程还在执行，当前线程也会停止等待，而继续往下执行。

比如：线程T1需要等待T2、T3完成之后才能继续执行，那么在T1线程中需要分别调用T2和T3的**join()**方法。

上个示例：

```
/**  
 * <b>description</b>: <br>  
 * <b>time</b>: 2019/7/12 17:18 <br>  
 * <b>author</b>: 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */  
public class Demo08 {  
    static int num = 0;  
  
    public static class T1 extends Thread {  
        public T1(String name) {  
            super(name);  
        }  
  
        @Override  
        public void run() {  
            System.out.println(System.currentTimeMillis() + ",start " +  
this.getName());  
            for (int i = 0; i < 10; i++) {  
                num++;  
            }  
        }  
    }  
}
```

```

        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    System.out.println(System.currentTimeMillis() + ",end " +
this.getName());
}

}

public static void main(String[] args) throws InterruptedException {
    T1 t1 = new T1("t1");
    t1.start();
    t1.join();
    System.out.println(System.currentTimeMillis() + ",num = " + num);
}
}

```

执行结果：

```

1562939889129,start t1
1562939891134,end t1
1562939891134,num = 10

```

num的结果为10，1、3行的时间戳相差2秒左右，说明主线程等待t1完成之后才继续执行的。

看一下jdk1.8中Thread.join()方法的实现：

```

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()) {
            wait(0);
        }
    } else {
        while (isAlive()) {
            long delay = millis - now;
            if (delay <= 0) {
                break;
            }
            wait(delay);
            now = System.currentTimeMillis() - base;
        }
    }
}

```

从join的代码中可以看出，在被等待的线程上使用了synchronized，调用了它的wait()方法，线程最后执行完毕之后，**系统会自动调用它的notifyAll()方法**，唤醒所有在此线程上等待的其他线程。

注意：被等待的线程执行完毕之后，系统自动会调用该线程的**notifyAll()**方法。所以一般情况下，我们不要去在线程对象上使用**wait()**、**notify()**、**notifyAll()**方法。

另外一个方法是**Thread.yield()**，他的定义如下：

```
public static native void yield();
```

yield是谦让的意思，这是一个静态方法，一旦执行，它会让当前线程出让CPU，但需要注意的是，出让CPU并不是说不让当前线程执行了，当前线程在出让CPU后，还会进行CPU资源的争夺，但是能否再抢到CPU的执行权就不一定了。因此，对**Thread.yield()**方法的调用好像就是在说：我已经完成了一些主要的工作，我可以休息一下了，可以让CPU给其他线程一些工作机会了。

如果觉得一个线程不太重要，或者优先级比较低，而又担心此线程会过多的占用CPU资源，那么可以在适当的时候调用一下**Thread.yield()**方法，给与其他线程更多的机会。

总结

1. 创建线程的2中方式：继承**Thread**类；实现**Runnable**接口
2. 启动线程：调用线程的**start()**方法
3. 终止线程：调用线程的**stop()**方法，方法已过时，建议不要使用
4. 线程中断相关的方法：调用线程**实例****interrupt()**方法将中断标志置为**true**；使用**线程实例方法isInterrupted()**获取中断标志；调用**Thread**的**静态方法****interrupted()**获取线程是否被中断，此方法调用之后会清除中断标志（将中断标志置为**false**了）
5. **wait**、**notify**、**notifyAll**方法，这块比较难理解，可以回过头去再理理
6. 线程挂起使用**线程实例方法****suspend()**，恢复线程使用**线程实例方法****resume()**，这2个方法都过时了，不建议使用
7. 等待线程结束：调用**线程实例方法****join()**
8. 出让cpu资源：调用**线程静态方法****yeild()**

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第7篇：volatile与Java内存模型

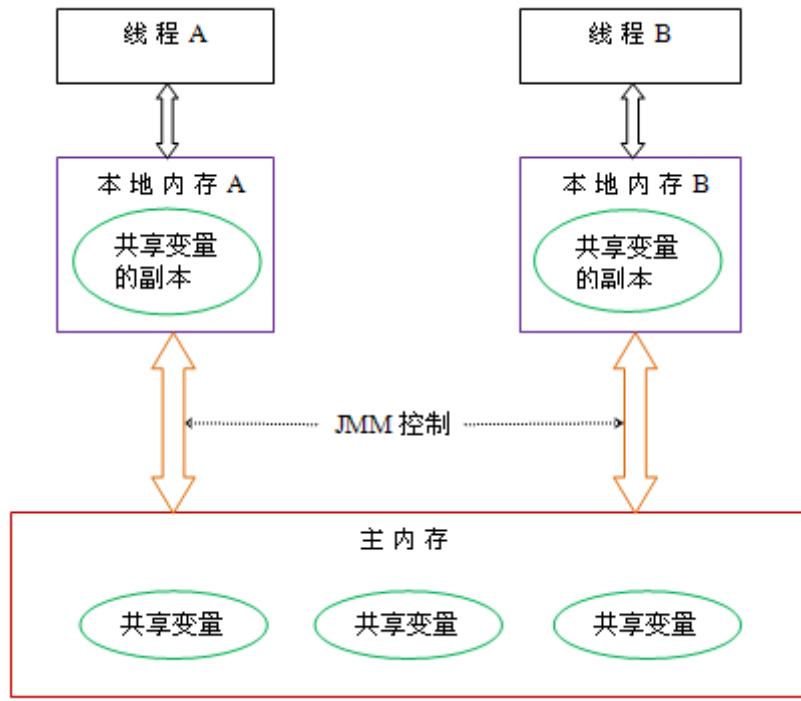
```
public class Demo09 {  
    public static boolean flag = true;  
  
    public static class T1 extends Thread {  
        public T1(String name) {  
            super(name);  
        }  
  
        @Override  
        public void run() {  
            System.out.println("线程" + this.getName() + " in");  
            while (flag) {  
                ;  
            }  
            System.out.println("线程" + this.getName() + "停止了");  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        new T1("t1").start();  
        //休眠1秒  
        Thread.sleep(1000);  
        //将flag置为false  
        flag = false;  
    }  
}
```

运行上面代码，会发现程序无法终止。

线程t1的run()方法中有个循环，通过flag来控制循环是否结束，主线程中休眠了1秒，将flag置为false，按说此时线程t1会检测到flag为false，打印“线程t1停止了”，为何和我们期望的结果不一样呢？运行上面的代码我们可以判断，t1中看到的flag一直为ture，主线程将flag置为false之后，t1线程中没有看到，所以一直死循环。

那么t1中为什么看不到被主线程修改之后的flag？

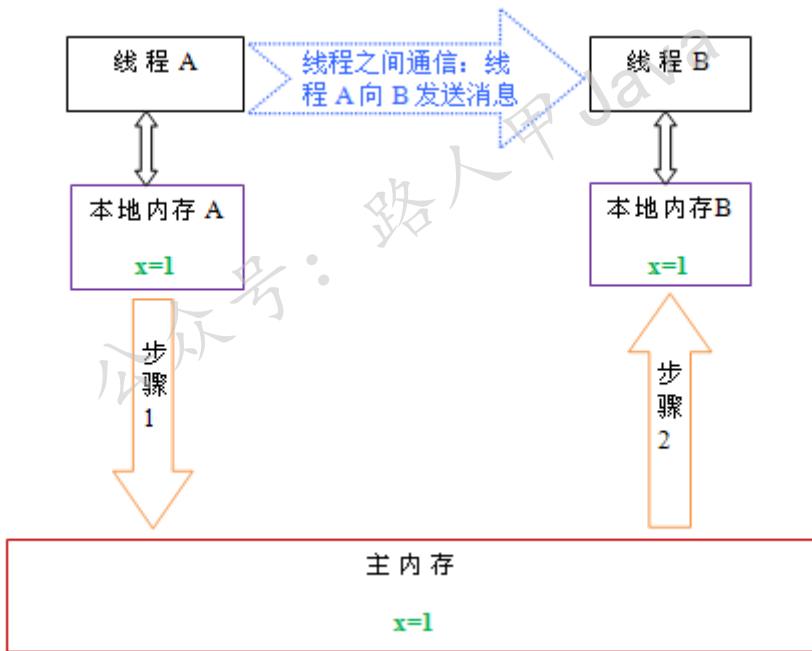
要解释这个，我们需要先了解一下java内存模型（JMM），Java线程之间的通信由Java内存模型（本文简称为JMM）控制，JMM决定一个线程对共享变量的写入何时对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他硬件和编译器优化。Java内存模型的抽象示意图如下：



从上图中可以看出，线程A需要和线程B通信，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量

下面通过示意图来说明这两个步骤：



如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

对JMM了解之后，我们再看看文章开头的问题，线程t1中为何看不到被主线程修改为false的flag的值，有两种可能：

1. 主线程修改了flag之后，未将其刷新到主内存，所以t1看不到

- 主线程将flag刷新到了主内存，但是t1一直读取的是自己工作内存中flag的值，没有去主内存中获取flag最新的值

对于上面2种情况，有没有什么办法可以解决？

是否有这样的方法：线程中修改了工作内存中的副本之后，立即将其刷新到主内存；工作内存中每次读取共享变量时，都去主内存中重新读取，然后拷贝到工作内存。

java帮我们提供了这样的方法，使用**volatile**修饰共享变量，就可以达到上面的效果，被volatile修改的变量有以下特点：

- 线程中读取的时候，每次读取都会去主内存中读取共享变量最新的值，然后将其复制到工作内存
- 线程中修改了工作内存中变量的副本，修改之后会立即刷新到主内存

我们修改一下开头的示例代码：

```
public volatile static boolean flag = true;
```

使用volatile修饰flag变量，然后运行一下程序，输出：

```
线程t1 in  
线程t1停止了
```

这下程序可以正常停止了。

volatile解决了共享变量在多线程中可见性的问题，可见性是指一个线程对共享变量的修改，对于另一个线程来说是否是可以看到的。

更多好文章

- [spring高手系列（正在连载中）](#)
- [Java高并发系列（共34篇）](#)
- [MySQL高手系列（共27篇）](#)
- [Maven高手系列（共10篇）](#)
- [Mybatis系列（共12篇）](#)
- [聊聊db和缓存一致性常见的实现方式](#)
- [接口幂等性这么重要，它是什么？怎么实现？](#)
- [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



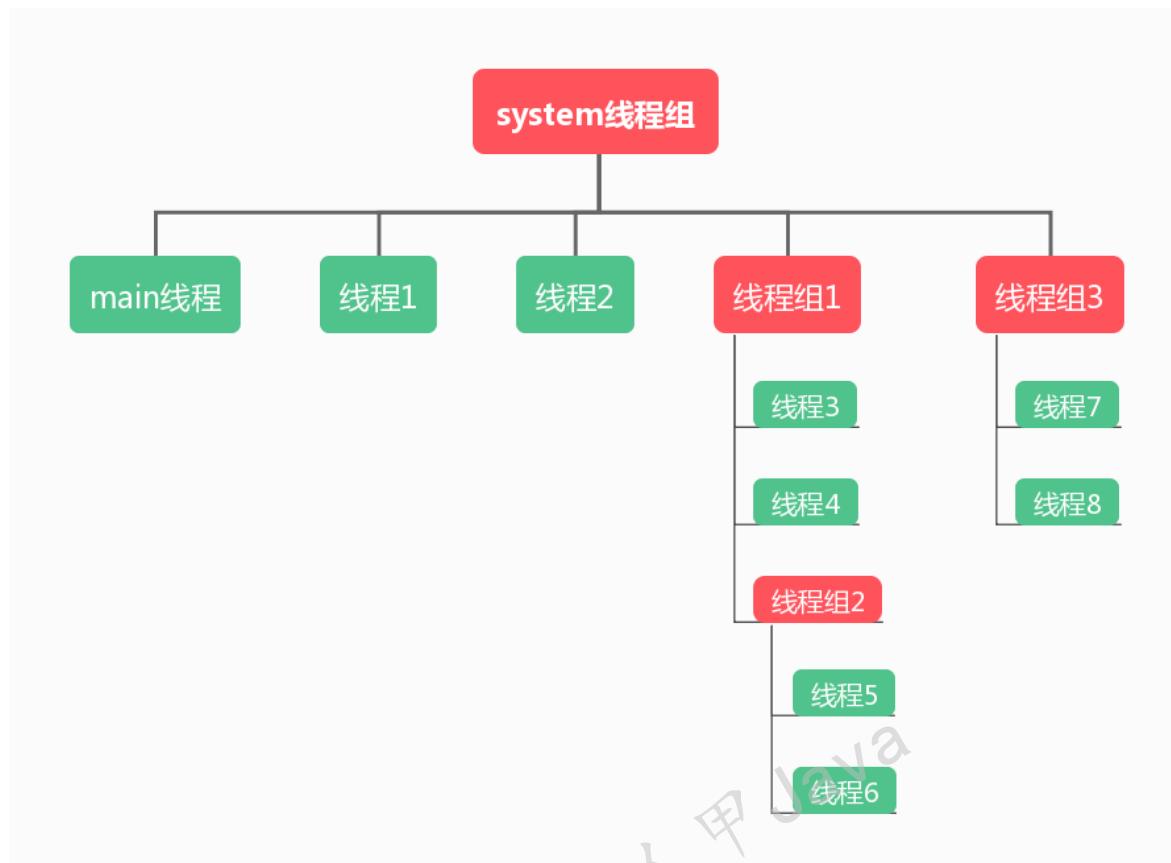
加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

第8篇：线程组

线程组

我们可以把线程归属到某个线程组中，线程组可以包含多个线程以及线程组，线程和线程组组成了父子关系，是个树形结构，如下图：



使用线程组可以方便管理线程，线程组提供了一些方法方便我们管理线程。

创建线程关联线程组

创建线程的时候，可以给线程指定一个线程组，代码如下：

```
package com.itsoku.chat02;

import java.util.concurrent.TimeUnit;

/**
 * <b>description</b>: <br>
 * <b>time</b>: 2019/7/13 17:53 <br>
 * <b>author</b>: 微信公众号: 路人甲Java, 专注于java技术分享 (带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等), 喜欢请关注!
 */
public class Demo1 {
    public static class R1 implements Runnable {
        @Override
        public void run() {
            System.out.println("threadName:" +
Thread.currentThread().getName());
            try {

```

```

        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) throws InterruptedException {
    ThreadGroup threadGroup = new ThreadGroup("thread-group-1");
    Thread t1 = new Thread(threadGroup, new R1(), "t1");
    Thread t2 = new Thread(threadGroup, new R1(), "t2");
    t1.start();
    t2.start();
    TimeUnit.SECONDS.sleep(1);
    System.out.println("活动线程数:" + threadGroup.activeCount());
    System.out.println("活动线程组:" + threadGroup.activeGroupCount());
    System.out.println("线程组名称:" + threadGroup.getName());
}
}

```

输出结果：

```

threadName:t1
threadName:t2
活动线程数:2
活动线程组:0
线程组名称:thread-group-1

```

`activeCount()`方法可以返回线程组中的所有活动线程数，包含下面的所有子孙节点的线程，由于线程组中的线程是动态变化的，这个值只能是一个估算值。

为线程组指定父线程组

创建线程组的时候，可以给其指定一个父线程组，也可以不指定，如果不指定父线程组，则父线程组为当前线程的线程组，java api有2个常用的构造方法用来创建线程组：

```

public ThreadGroup(String name)
public ThreadGroup(ThreadGroup parent, String name)

```

第一个构造方法未指定父线程组，看一下内部的实现：

```

public ThreadGroup(String name) {
    this(Thread.currentThread().getThreadGroup(), name);
}

```

系统自动获取当前线程的线程组作为默认父线程组。

上一段示例代码：

```

package com.itsoku.chat02;

import java.util.concurrent.TimeUnit;

```

```

/**
 * <b>description</b>: <br>
 * <b>time</b>: 2019/7/13 17:53 <br>
 * <b>author</b>: 微信公众号: 路人甲Java, 专注于java技术分享(带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等), 喜欢请关注!
 */
public class Demo2 {
    public static class R1 implements Runnable {
        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            System.out.println("所属线程组:" + thread.getThreadGroup().getName() +
",线程名称:" + thread.getName());
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadGroup threadGroup1 = new ThreadGroup("thread-group-1");
        Thread t1 = new Thread(threadGroup1, new R1(), "t1");
        Thread t2 = new Thread(threadGroup1, new R1(), "t2");
        t1.start();
        t2.start();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("threadGroup1活动线程数:" +
threadGroup1.activeCount());
        System.out.println("threadGroup1活动线程组:" +
threadGroup1.activeGroupCount());
        System.out.println("threadGroup1线程组名称:" + threadGroup1.getName());
        System.out.println("threadGroup1父线程组名称:" +
threadGroup1.getParent().getName());
        System.out.println("-----");
        ThreadGroup threadGroup2 = new ThreadGroup(threadGroup1, "thread-group-
2");
        Thread t3 = new Thread(threadGroup2, new R1(), "t3");
        Thread t4 = new Thread(threadGroup2, new R1(), "t4");
        t3.start();
        t4.start();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("threadGroup2活动线程数:" +
threadGroup2.activeCount());
        System.out.println("threadGroup2活动线程组:" +
threadGroup2.activeGroupCount());
        System.out.println("threadGroup2线程组名称:" + threadGroup2.getName());
        System.out.println("threadGroup2父线程组名称:" +
threadGroup2.getParent().getName());

        System.out.println("-----");
        System.out.println("threadGroup1活动线程数:" +
threadGroup1.activeCount());
        System.out.println("threadGroup1活动线程组:" +
threadGroup1.activeGroupCount());

        System.out.println("-----");
    }
}

```

```
        threadGroup1.list();
    }
}
```

输出结果：

```
所属线程组:thread-group-1,线程名称:t1
所属线程组:thread-group-1,线程名称:t2
threadGroup1活动线程数:2
threadGroup1活动线程组:0
threadGroup1线程组名称:thread-group-1
threadGroup1父线程组名称:main
-----
所属线程组:thread-group-2,线程名称:t4
所属线程组:thread-group-2,线程名称:t3
threadGroup2活动线程数:2
threadGroup2活动线程组:0
threadGroup2线程组名称:thread-group-2
threadGroup2父线程组名称:thread-group-1
-----
threadGroup1活动线程数:4
threadGroup1活动线程组:1
-----
java.lang.ThreadGroup[name=thread-group-1,maxpri=10]
    Thread[t1,5,thread-group-1]
    Thread[t2,5,thread-group-1]
java.lang.ThreadGroup[name=thread-group-2,maxpri=10]
    Thread[t3,5,thread-group-2]
    Thread[t4,5,thread-group-2]
```

代码解释：

1. **threadGroup1**未指定父线程组，系统获取了主线程的线程组作为**threadGroup1**的父线程组，输出结果中是：main
2. **threadGroup1**为**threadGroup2**的父线程组
3. **threadGroup1**活动线程数为4，包含了**threadGroup1**线程组中的t1、t2，以及子线程组**threadGroup2**中的t3、t4
4. 线程组的list()方法，将线程组中的所有子孙节点信息输出到控制台，用于调试使用

根线程组

获取根线程组

```
package com.itsoku.chat02;

/**
 * <b>description</b>: <br>
 * <b>time</b>: 2019/7/13 17:53 <br>
 * <b>author</b>: 微信公众号: 路人甲Java, 专注于java技术分享(带你玩转爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等), 喜欢请关注!
 */
public class Demo3 {

    public static void main(String[] args) {
```

```
        System.out.println(Thread.currentThread());
        System.out.println(Thread.currentThread().getThreadGroup());
        System.out.println(Thread.currentThread().getThreadGroup().getParent());

        System.out.println(Thread.currentThread().getThreadGroup().getParent().getParent());
    }
}
```

运行上面代码，输出：

```
Thread[main,5,main]
java.lang.ThreadGroup[name=main,maxpri=10]
java.lang.ThreadGroup[name=system,maxpri=10]
null
```

从上面代码可以看出：

1. **主线程的线程组为main**
2. **根线程组为system**

看一下ThreadGroup的源码：

```
private ThreadGroup() { // called from C code
    this.name = "system";
    this.maxPriority = Thread.MAX_PRIORITY;
    this.parent = null;
}
```

发现ThreadGroup默认构造方法是private的，是由c调用的，创建的正是system线程组。

批量停止线程

调用线程组**interrupt()**，会将线程组树下的所有子孙线程中断标志置为true，可以用来批量中断线程。

示例代码：

```
package com.itsoku.chat02;

import java.util.concurrent.TimeUnit;

/**
 * <b>description</b>: <br>
 * <b>time</b>: 2019/7/13 17:53 <br>
 * <b>author</b>: 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo4 {
    public static class R1 implements Runnable {
        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            thread.interrupt();
        }
    }
}
```

```

        System.out.println("所属线程组：" + thread.getThreadGroup().getName() +
", 线程名称：" + thread.getName());
        while (!thread.isInterrupted()) {
            ;
        }
        System.out.println("线程：" + thread.getName() + "停止了！");
    }
}

public static void main(String[] args) throws InterruptedException {
    ThreadGroup threadGroup1 = new ThreadGroup("thread-group-1");
    Thread t1 = new Thread(threadGroup1, new R1(), "t1");
    Thread t2 = new Thread(threadGroup1, new R1(), "t2");
    t1.start();
    t2.start();

    ThreadGroup threadGroup2 = new ThreadGroup(threadGroup1, "thread-group-
2");
    Thread t3 = new Thread(threadGroup2, new R1(), "t3");
    Thread t4 = new Thread(threadGroup2, new R1(), "t4");
    t3.start();
    t4.start();
    TimeUnit.SECONDS.sleep(1);

    System.out.println("-----threadGroup1信息-----");
    threadGroup1.list();

    System.out.println("-----");
    System.out.println("停止线程组：" + threadGroup1.getName() + "中的所有子孙线
程");
    threadGroup1.interrupt();
    TimeUnit.SECONDS.sleep(2);

    System.out.println("-----threadGroup1停止后，输出信息-----");
    threadGroup1.list();
}
}

```

输出：

```

所属线程组:thread-group-1,线程名称:t1
所属线程组:thread-group-1,线程名称:t2
所属线程组:thread-group-2,线程名称:t3
所属线程组:thread-group-2,线程名称:t4
-----threadGroup1信息-----
java.lang.ThreadGroup[name=thread-group-1,maxpri=10]
    Thread[t1,5,thread-group-1]
    Thread[t2,5,thread-group-1]
java.lang.ThreadGroup[name=thread-group-2,maxpri=10]
    Thread[t3,5,thread-group-2]
    Thread[t4,5,thread-group-2]
-----
停止线程组： thread-group-1中的所有子孙线程
线程:t4停止了!
线程:t2停止了!
线程:t1停止了!
线程:t3停止了!

```

```
-----threadGroup1停止后，输出信息-----  
java.lang.ThreadGroup[name=thread-group-1,maxpri=10]  
java.lang.ThreadGroup[name=thread-group-2,maxpri=10]
```

停止线程之后，通过list()方法可以看出输出的信息中不包含已结束的线程了。

多说几句，建议大家再创建线程或者线程组的时候，给他们取一个有意义的名字，对于计算机来说，可能名字并不重要，但是在系统出问题的时候，你可能会去查看线程堆栈信息，如果你看到的都是t1、t2、t3，估计自己也比较崩溃，如果看到的是httpAccpHandler、dubboHandler类似的名字，应该会好很多。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第9篇：用户线程和守护线程

守护线程是一种特殊的线程，在后台默默地完成一些系统性的服务，比如**垃圾回收线程、JIT线程都是守护线程**。与之对应的是**用户线程**，用户线程可以理解为是系统的工作线程，它会完成这个程序需要完成的业务操作。如果用户线程全部结束了，意味着程序需要完成的业务操作已经结束了，系统可以退出了。**所以当系统只剩下守护进程的时候，java虚拟机会自动退出**。

java线程分为用户线程和守护线程，线程的daemon属性为true表示是守护线程，false表示是用户线程。

下面我们来看一下守护线程的一些特性。

程序只有守护线程时，系统会自动退出

```
package com.itsoku.chat03;
```

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

```

/**
 * 微信公众号: 路人甲Java, 专注于java技术分享(带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等), 喜欢请关注!
 */
public class Demo1 {

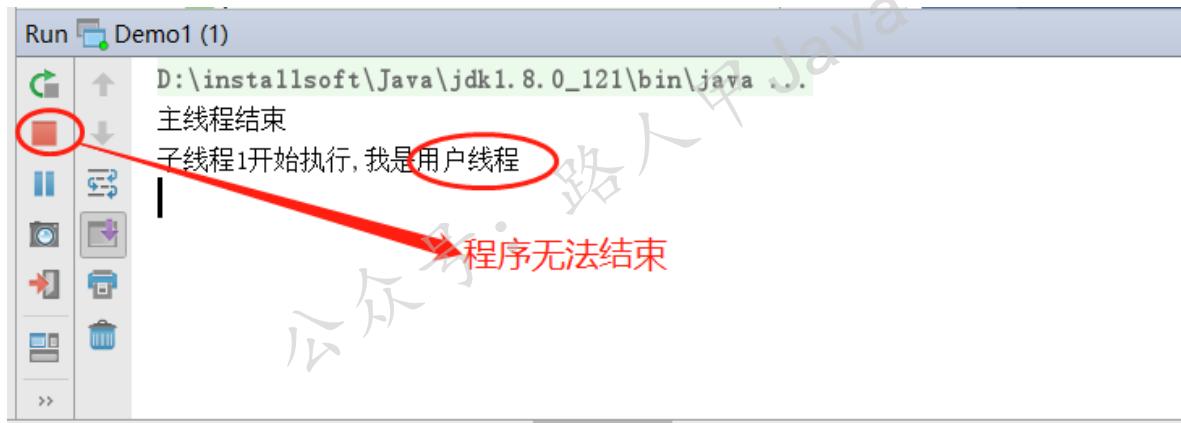
    public static class T1 extends Thread {
        public T1(String name) {
            super(name);
        }

        @Override
        public void run() {
            System.out.println(this.getName() + "开始执行," + (this.isDaemon() ?
"我是守护线程" : "我是用户线程"));
            while (true);
        }
    }

    public static void main(String[] args) {
        T1 t1 = new T1("子线程1");
        t1.start();
        System.out.println("主线程结束");
    }
}

```

运行上面代码，结果如下：



可以看到主线程已经结束了，但是程序无法退出，原因：子线程1是用户线程，内部有个死循环，一直处于运行状态，无法结束。

再看下面的代码：

```

package com.itsoku.chat03;

/**
 * 微信公众号: 路人甲Java, 专注于java技术分享(带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等), 喜欢请关注!
 */
public class Demo2 {

    public static class T1 extends Thread {

```

```

public T1(String name) {
    super(name);
}

@Override
public void run() {
    System.out.println(this.getName() + "开始执行," + (this.isDaemon() ?
"我是守护线程" : "我是用户线程"));
    while (true) ;
}
}

public static void main(String[] args) {
T1 t1 = new T1("子线程1");
t1.setDaemon(true);
t1.start();
System.out.println("主线程结束");
}
}

```

运行结果：



程序可以正常结束了，代码中通过 `t1.setDaemon(true);` 将 `t1` 线程设置为守护线程，`main` 方法所在的主线程执行完毕之后，程序就退出了。

结论：当程序中所有的用户线程执行完毕之后，不管守护线程是否结束，系统都会自动退出。

设置守护线程，需要在start()方法之前进行

```

package com.itsoku.chat03;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo3 {

    public static void main(String[] args) {

```

```

        Thread t1 = new Thread() {
            @Override
            public void run() {
                try {
                    TimeUnit.SECONDS.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        t1.start();
        t1.setDaemon(true);
    }
}

```

`t1.setDaemon(true);`是在`t1.start()`方法之后执行的，执行会报异常，运行结果如下：

线程daemon的默认值

我们看一下创建线程源码，位于**Thread类的init()**方法中：

```

Thread parent = currentThread();
this.daemon = parent.isDaemon();

```

`daemonic`的默认值为父线程的`daemon`，也就是说，父线程如果为用户线程，子线程默认也是用户线程，父线程如果是守护线程，子线程默认也是守护线程。

示例代码：

```

package com.itsoku.chat03;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo4 {
    public static class T1 extends Thread {
        public T1(String name) {
            super(name);
        }

        @Override
        public void run() {

```

```

        System.out.println(this.getName() + ".daemon:" + this.isDaemon());
    }

}

public static void main(String[] args) throws InterruptedException {
    System.out.println(Thread.currentThread().getName() + ".daemon:" +
    Thread.currentThread().isDaemon());

    T1 t1 = new T1("t1");
    t1.start();

    Thread t2 = new Thread() {
        @Override
        public void run() {
            System.out.println(this.getName() + ".daemon:" +
this.isDaemon());
            T1 t3 = new T1("t3");
            t3.start();
        }
    };

    t2.setName("t2");
    t2.setDaemon(true);
    t2.start();

    TimeUnit.SECONDS.sleep(2);
}
}

```

运行代码，输出：

```

main.daemon:false
t1.daemon:false
t2.daemon:true
t3.daemon:true

```

t1是由主线程(main方法所在的线程)创建的，main线程是t1的父线程，所以t1.daemon为false，说明t1是用户线程。

t2线程调用了setDaemon(true);将其设为守护线程，t3是由t2创建的，所以t3默认线程类型和t2一样，t2.daemon为true。

总结

1. java中的线程分为**用户线程**和**守护线程**
2. 程序中的所有的用户线程结束之后，不管守护线程处于什么状态，java虚拟机都会自动退出
3. 调用线程的实例方法setDaemon()来设置线程是否是守护线程
4. setDaemon()方法必须在线程的start()方法之前调用，在后面调用会报异常，并且不起效
5. 线程的daemon默认值和其父线程一样

更多好文章

1. [spring高手系列（正在连载中）](#)

2. [Java高并发系列 \(共34篇\)](#)
3. [MySQL高手系列 \(共27篇\)](#)
4. [Maven高手系列 \(共10篇\)](#)
5. [Mybatis系列 \(共12篇\)](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第10篇：线程安全和synchronized

什么是线程安全？

当多个线程去访问同一个类（对象或方法）的时候，该类都能表现出正常的行为（与自己预想的结果一致），那我们就可以说这个类是线程安全的。

看一段代码：

```
package com.itsoku.chat04;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo1 {
    static int num = 0;

    public static void m1() {
        for (int i = 0; i < 10000; i++) {
            num++;
        }
    }

    public static class T1 extends Thread {
        @Override
        public void run() {
            Demo1.m1();
        }
    }
}

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

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

```

T1 t1 = new T1();
T1 t2 = new T1();
T1 t3 = new T1();
t1.start();
t2.start();
t3.start();

//等待3个线程结束打印num
t1.join();
t2.join();
t3.join();

System.out.println(Demo1.num);
/**
 * 打印结果:
 * 25572
 */
}
}

```

Demo1中有个静态变量num，默认值是0，m1()方法中对num++执行10000次，main方法中创建了3个线程用来调用m1()方法，然后调用3个线程的join()方法，用来等待3个线程执行完毕之后，打印num的值。我们期望的结果是30000，运行一下，但真实的结果却不是30000。上面的程序在多线程中表现出来的结果和预想的结果不一致，说明上面的程序不是线程安全的。

线程安全是并发编程中的重要关注点，应该注意到的是，造成线程安全问题的主要诱因有两点：

1. 一是存在共享数据(也称临界资源)
2. 二是存在多条线程共同操作共享数据

因此为了解决这个问题，我们可能需要这样一个方案，当存在多个线程操作共享数据时，**需要保证同一时刻有且只有一个线程在操作共享数据**，其他线程必须等到该线程处理完数据后再进行，这种方式有个高尚的名称叫**互斥锁**，即能达到互斥访问目的的锁，也就是说当一个共享数据被当前正在访问的线程加上互斥锁后，在同一个时刻，其他线程只能处于等待的状态，直到当前线程处理完毕释放该锁。在Java中，**关键字 synchronized可以保证在同一个时刻，只有一个线程可以执行某个方法或者某个代码块(主要是对方法或者代码块中存在共享数据的操作)**，同时我们还应该注意到**synchronized另外一个重要的作用，synchronized可保证一个线程的变化(主要是共享数据的变化)被其他线程所看到(保证可见性，完全可以替代volatile功能)**，这点确实也是很重要的。

那么我们把上面的程序做一下调整，在m1()方法上面使用关键字synchronized，如下：

```

public static synchronized void m1() {
    for (int i = 0; i < 10000; i++) {
        num++;
    }
}

```

然后执行代码，输出30000，和期望结果一致。

synchronized主要有3种使用方式

1. 修饰实例方法，作用于当前实例，进入同步代码前需要先获取实例的锁
2. 修饰静态方法，作用于类的Class对象，进入修饰的静态方法前需要先获取类的Class对象的锁
3. 修饰代码块，需要指定加锁对象(记做lockobj)，在进入同步代码块前需要先获取lockobj的锁

synchronized作用于实例对象

所谓实例对象锁就是用synchronized修饰实例对象的实例方法，注意是**实例方法**，不是**静态方法**，如：

```
package com.itsoku.chat04;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo2 {
    int num = 0;

    public synchronized void add() {
        num++;
    }

    public static class T extends Thread {
        private Demo2 demo2;

        public T(Demo2 demo2) {
            this.demo2 = demo2;
        }

        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                this.demo2.add();
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    Demo2 demo2 = new Demo2();
    T t1 = new T(demo2);
    T t2 = new T(demo2);
    t1.start();
    t2.start();

    t1.join();
    t2.join();

    System.out.println(demo2.num);
}
```

main()方法中创建了一个对象demo2和2个线程t1、t2，t1、t2中调用demo2的add()方法10000次，add()方法中执行了num++，num++实际上是分3步，获取num，然后将num+1，然后将结果赋值给num，如果t2在t1读取num和num+1之间获取了num的值，那么t1和t2会读取到同样的值，然后执行num++，两次操作之后num是相同的值，最终和期望的结果不一致，造成了线程安全失败，因此我们对add方法加了synchronized来保证线程安全。

注意：m1()方法是实例方法，两个线程操作m1()时，需要先获取demo2的锁，没有获取到锁的，将等待，直到其他线程释放锁为止。

synchronize作用于实例方法需要注意：

1. 实例方法上加synchronized，线程安全的前提是，多个线程操作的是**同一个实例**，如果多个线程作用于不同的实例，那么线程安全是无法保证的
2. 同一个实例的多个实例方法上有synchronized，这些方法都是互斥的，同一时间只允许一个线程操作**同一个实例的其中的一个synchronized方法**

synchronized作用于静态方法

当synchronized作用于静态方法时，锁的对象就是当前类的Class对象。如：

```
package com.itsoku.chat04;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo3 {
    static int num = 0;

    public static synchronized void m1() {
        for (int i = 0; i < 10000; i++) {
            num++;
        }
    }

    public static class T1 extends Thread {
        @Override
        public void run() {
            Demo3.m1();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T1 t1 = new T1();
        T1 t2 = new T1();
        T1 t3 = new T1();
        t1.start();
        t2.start();
        t3.start();

        //等待3个线程结束打印num
        t1.join();
        t2.join();
        t3.join();

        System.out.println(Demo3.num);
        /**
         * 打印结果：
         * 30000
         */
    }
}
```

上面代码打印30000，和期望结果一致。m1()方法是静态方法，有synchronized修饰，锁用于与Demo3.class对象，和下面的写法类似：

```
public static void m1() {  
    synchronized (Demo4.class) {  
        for (int i = 0; i < 10000; i++) {  
            num++;  
        }  
    }  
}
```

synchronized同步代码块

除了使用关键字修饰实例方法和静态方法外，还可以使用同步代码块，在某些情况下，我们编写的方法体可能比较大，同时存在一些比较耗时的操作，而需要同步的代码又只有一小部分，如果直接对整个方法进行同步操作，可能会得不偿失，此时我们可以使用同步代码块的方式对需要同步的代码进行包裹，这样就无需对整个方法进行同步操作了，同步代码块的使用示例如下：

```
package com.itsoku.chat04;  
  
/**  
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */  
public class Demo5 implements Runnable {  
    static Demo5 instance = new Demo5();  
    static int i = 0;  
  
    @Override  
    public void run() {  
        //省略其他耗时操作....  
        //使用同步代码块对变量i进行同步操作，锁对象为instance  
        synchronized (instance) {  
            for (int j = 0; j < 10000; j++) {  
                i++;  
            }  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        Thread t1 = new Thread(instance);  
        Thread t2 = new Thread(instance);  
        t1.start();  
        t2.start();  
  
        t1.join();  
        t2.join();  
  
        System.out.println(i);  
    }  
}
```

从代码看出，将synchronized作用于一个给定的实例对象instance，即当前实例对象就是锁对象，每次当线程进入synchronized包裹的代码块时就会要求当前线程持有instance实例对象锁，如果当前有其他线程正持有该对象锁，那么新到的线程就必须等待，这样也就保证了每次只有一个线程执行i++;操作。当然除了instance作为对象外，我们还可以使用this对象(代表当前实例)或者当前类的class对象作为锁，如下代码：

```
//this,当前实例对象锁
synchronized(this){
    for(int j=0;j<1000000;j++){
        i++;
    }
}

//class对象锁
synchronized(Demo5.class){
    for(int j=0;j<1000000;j++){
        i++;
    }
}
```

分析代码是否互斥的方法，先找出synchronized作用的对象是谁，如果多个线程操作的方法中synchronized作用的锁对象一样，那么这些线程同时异步执行这些方法就是互斥的。如下代码：

```
package com.itsoku.chat04;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo6 {
    //作用于当前类的实例对象
    public synchronized void m1() {
    }

    //作用于当前类的实例对象
    public synchronized void m2() {
    }

    //作用于当前类的实例对象
    public void m3() {
        synchronized (this) {
        }
    }

    //作用于当前类class对象
    public static synchronized void m4() {
    }

    //作用于当前类class对象
    public static void m5() {
        synchronized (Demo6.class) {
        }
    }
}
```

```

public static class T extends Thread{
    Demo6 demo6;

    public T(Demo6 demo6) {
        this.demo6 = demo6;
    }

    @Override
    public void run() {
        super.run();
    }
}

public static void main(String[] args) {
    Demo6 d1 = new Demo6();
    Thread t1 = new Thread(() -> {
        d1.m1();
    });
    t1.start();
    Thread t2 = new Thread(() -> {
        d1.m2();
    });
    t2.start();

    Thread t3 = new Thread(() -> {
        d1.m2();
    });
    t3.start();

    Demo6 d2 = new Demo6();
    Thread t4 = new Thread(() -> {
        d2.m2();
    });
    t4.start();

    Thread t5 = new Thread(() -> {
        Demo6.m4();
    });
    t5.start();

    Thread t6 = new Thread(() -> {
        Demo6.m5();
    });
    t6.start();
}
}

```

分析上面代码：

1. 线程t1、t2、t3中调用的方法都需要获取d1的锁，所以他们是互斥的
2. t1/t2/t3这3个线程和t4不互斥，他们可以同时运行，因为前面三个线程依赖于d1的锁，t4依赖于d2的锁
3. t5、t6都作用于当前类的Class对象锁，所以这两个线程是互斥的，和其他几个线程不互斥

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第11篇：中断线程的几种方式

本文主要探讨一下中断线程的几种方式。

通过一个变量控制线程中断

代码：

```
package com.itsoku.chat05;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo1 {

    public volatile static boolean exit = false;

    public static class T extends Thread {
        @Override
        public void run() {
            while (true) {
                //循环处理业务
                if (exit) {
                    break;
                }
            }
        }
    }
}
```

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

```

        }
    }

    public static void setExit() {
        exit = true;
    }

    public static void main(String[] args) throws InterruptedException {
        T t = new T();
        t.start();
        TimeUnit.SECONDS.sleep(3);
        setExit();
    }
}

```

代码中启动了一个线程，线程的run方法中有个死循环，内部通过exit变量的值来控制是否退出。
`TimeUnit.SECONDS.sleep(3);`让主线程休眠3秒，此处为什么使用TimeUnit? TimeUnit使用更方便一些，能够很清晰的控制休眠时间，底层还是转换为Thread.sleep实现的。程序有个重点：**volatile**关键字，exit变量必须通过这个修饰，如果把这个去掉，程序无法正常退出。volatile控制了变量在多线程中的可见性，关于volatile前面的文章中有介绍，此处就不再说了。

通过线程自带的中断标志控制

示例代码：

```

package com.itsoku.chat05;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo2 {

    public static class T extends Thread {
        @Override
        public void run() {
            while (true) {
                //循环处理业务
                if (this.isInterrupted()) {
                    break;
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t = new T();
        t.start();
        TimeUnit.SECONDS.sleep(3);
        t.interrupt();
    }
}

```

```
    }  
}
```

运行上面的程序，程序可以正常结束。线程内部有个中断标志，当调用线程的interrupt()实例方法之后，线程的中断标志会被置为true，可以通过线程的实例方法isInterrupted()获取线程的中断标志。

线程阻塞状态中如何中断

示例代码：

```
package com.itsoku.chat05;  
  
import java.util.concurrent.TimeUnit;  
  
/**  
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */  
public class Demo3 {  
  
    public static class T extends Thread {  
        @Override  
        public void run() {  
            while (true) {  
                //循环处理业务  
                //下面模拟阻塞代码  
                try {  
                    TimeUnit.SECONDS.sleep(1000);  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                }  
            }  
        }  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        T t = new T();  
        t.start();  
    }  
}
```

运行上面代码，发现程序无法结束。

在此先补充几点知识：

1. 调用线程的interrupt()实例方法，线程的中断标志会被置为true
2. 当线程处于阻塞状态时，调用线程的interrupt()实例方法，线程内部会触发 InterruptedException异常，并且会清除线程内部的中断标志（即将中断标志置为false）

那么上面代码可以调用线程的interrupt()方法来引发InterruptedException异常，来中断sleep方法导致的阻塞，调整一下代码，如下：

```
package com.itsoku.chat05;  
  
import java.util.concurrent.TimeUnit;
```

```

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo3 {

    public static class T extends Thread {
        @Override
        public void run() {
            while (true) {
                //循环处理业务
                //下面模拟阻塞代码
                try {
                    TimeUnit.SECONDS.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    this.interrupt();
                }
                if (this.isInterrupted()) {
                    break;
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t = new T();
        t.start();
        TimeUnit.SECONDS.sleep(3);
        t.interrupt();
    }
}

```

运行结果：

```

java.lang.InterruptedException: sleep interrupted
at java.lang.Thread.sleep(Native Method)
at java.lang.Thread.sleep(Thread.java:340)
at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
at com.itsoku.chat05.Demo3$T.run(Demo3.java:17)

```

程序可以正常结束了，分析一下上面代码，注意几点：

1. main方法中调用了t.interrupt()方法，此时线程t内部的中断标志会置为true
2. 然后会触发run()方法内部的InterruptedException异常，所以运行结果中有异常输出，上面说了，当触发InterruptedException异常时候，线程内部的中断标志又会被清除（变为false），所以在catch中又调用了this.interrupt();一次，将中断标志置为false
3. run()方法中通过this.isInterrupted()来获取线程的中断标志，退出循环（break）

总结

1. 当一个线程处于被阻塞状态或者试图执行一个阻塞操作时，可以使用 `Thread.interrupt()` 方式中断该线程，注意此时将会抛出一个 **InterruptedException** 的异常，同时中断状态将被复位（由中断状态改为非中断状态）
2. 内部有循环体，可以通过一个变量来作为一个信号控制线程是否中断，注意变量需要 `volatile` 修饰
3. 文中的几种方式可以结合起来灵活使用控制线程的中断

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第12篇：JUC中ReentrantLock

本篇文章开始将juc中常用的一些类，估计会有十来篇。

synchronized的局限性

`synchronized` 是 Java 内置的关键字，它提供了一种独占的加锁方式。`synchronized` 的获取和释放锁由 JVM 实现，用户不需要显示的释放锁，非常方便，然而 `synchronized` 也有一定的局限性，例如：

1. 当线程尝试获取锁的时候，如果获取不到锁会一直阻塞，这个阻塞的过程，用户无法控制
2. 如果获取锁的线程进入休眠或者阻塞，除非当前线程异常，否则其他线程尝试获取锁必须一直等待

JDK1.5 之后发布，加入了 Doug Lea 实现的 `java.util.concurrent` 包。包内提供了 `Lock` 类，用来提供更多扩展的加锁功能。`Lock` 弥补了 `synchronized` 的局限，提供了更加细粒度的加锁功能。

ReentrantLock

`ReentrantLock` 是 `Lock` 的默认实现，在聊 `ReentrantLock` 之前，我们需要先弄清楚一些概念：

1. 可重入锁：可重入锁是指同一个线程可以多次获得同一把锁；ReentrantLock和关键字Synchronized都是可重入锁
2. 可中断锁：可中断锁时子线程在获取锁的过程中，是否可以相应线程中断操作。synchronized是不可中断的，ReentrantLock是可中断的
3. 公平锁和非公平锁：公平锁是指多个线程尝试获取同一把锁的时候，获取锁的顺序按照线程到达的先后顺序获取，而不是随机插队的方式获取。synchronized是非公平锁，而ReentrantLock是两种都可以实现，不过默认是非公平锁

ReentrantLock基本使用

我们使用3个线程来对一个共享变量++操作，先使用synchronized实现，然后使用ReentrantLock实现。

synchronized方式：

```
package com.itsoku.chat06;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo2 {

    private static int num = 0;

    private static synchronized void add() {
        num++;
    }

    public static class T extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                Demo2.add();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T();
        T t2 = new T();
        T t3 = new T();

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        System.out.println(Demo2.num);
    }
}
```

ReentrantLock方式：

```

package com.itsoku.chat06;

import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo3 {

    private static int num = 0;

    private static ReentrantLock lock = new ReentrantLock();

    private static void add() {
        lock.lock();
        try {
            num++;
        } finally {
            lock.unlock();
        }
    }

    public static class T extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                Demo3.add();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T();
        T t2 = new T();
        T t3 = new T();

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        System.out.println(Demo3.num);
    }
}

```

ReentrantLock的使用过程：

1. 创建锁： **ReentrantLock lock = new ReentrantLock();**
2. 获取锁： **lock.lock()**
3. 释放锁： **lock.unlock();**

对比上面的代码，与关键字synchronized相比，ReentrantLock锁有明显的操作过程，开发人员必须手动的指定何时加锁，何时释放锁，正是因为这样手动控制，ReentrantLock对逻辑控制的灵活度要远远胜于关键字synchronized，上面代码需要注意lock.unlock()一定要放在finally中，否则，若程序出现了异常，锁没有释放，那么其他线程就再也没有机会获取这个锁了。

ReentrantLock是可重入锁

来验证一下ReentrantLock是可重入锁，实例代码：

```
package com.itsoku.chat06;

import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo4 {
    private static int num = 0;
    private static ReentrantLock lock = new ReentrantLock();

    private static void add() {
        lock.lock();
        lock.lock();
        try {
            num++;
        } finally {
            lock.unlock();
            lock.unlock();
        }
    }

    public static class T extends Thread {
        @Override
        public void run() {
            for (int i = 0; i < 10000; i++) {
                Demo4.add();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T();
        T t2 = new T();
        T t3 = new T();

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        System.out.println(Demo4.num);
    }
}
```

```
}
```

上面代码中add()方法中，当一个线程进入的时候，会执行2次获取锁的操作，运行程序可以正常结束，并输出和期望值一样的30000，假如ReentrantLock是不可重入的锁，那么同一个线程第2次获取锁的时候由于前面的锁还未释放而导致死锁，程序是无法正常结束的。ReentrantLock命名也挺好的Reentrant Lock，和其名字一样，可重入锁。

代码中还有几点需要注意：

1. **lock()方法和unlock()方法需要成对出现，锁了几次，也要释放几次，否则后面的线程无法获取锁了；可以将add中的unlock删除一个事实，上面代码运行将无法结束**
2. **unlock()方法放在finally中执行，保证不管程序是否有异常，锁必定会释放**

ReentrantLock实现公平锁

在大多数情况下，锁的申请都是非公平的，也就是说，线程1首先请求锁A，接着线程2也请求了锁A。那么当锁A可用时，是线程1可获得锁还是线程2可获得锁呢？这是不一定的，系统只是会从这个锁的等待队列中随机挑选一个，因此不能保证其公平性。这就好比买票不排队，大家都围在售票窗口前，售票员忙的焦头烂额，也顾及不上谁先谁后，随便找个人出票就完事了，最终导致的结果是，有些人可能一直买不到票。而公平锁，则不是这样，它会按照到达的先后顺序获得资源。公平锁的一大特点是：它不会产生饥饿现象，只要你排队，最终还是可以等到资源的；synchronized关键字默认是有jvm内部实现控制的，是非公平锁。而ReentrantLock运行开发者自己设置锁的公平性。

看一下jdk中ReentrantLock的源码，2个构造方法：

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

默认构造方法创建的是非公平锁。

第2个构造方法，有个fair参数，当fair为true的时候创建的是公平锁，公平锁看起来很不错，不过要实现公平锁，系统内部肯定需要维护一个有序队列，因此公平锁的实现成本比较高，性能相对于非公平锁来说相对低一些。因此，在默认情况下，锁是非公平的，如果没有特别要求，则不建议使用公平锁。

公平锁和非公平锁在程序调度上是很不一样，来一个公平锁示例看一下：

```
package com.itsoku.chat06;  
  
import java.util.concurrent.locks.ReentrantLock;  
  
/**  
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */  
public class Demo5 {  
    private static int num = 0;  
    private static ReentrantLock fairLock = new ReentrantLock(true);  
  
    public static class T extends Thread {  
        public T(String name) {  
            super(name);  
        }
```

```

    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            fairLock.lock();
            try {
                System.out.println(this.getName() + "获得锁!");
            } finally {
                fairLock.unlock();
            }
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    T t1 = new T("t1");
    T t2 = new T("t2");
    T t3 = new T("t3");

    t1.start();
    t2.start();
    t3.start();

    t1.join();
    t2.join();
    t3.join();
}
}

```

运行结果输出：

```

t1获得锁!
t2获得锁!
t3获得锁!

```

看一下输出的结果，锁是按照先后顺序获得的。

修改一下上面代码，改为非公平锁试试，如下：

```
ReentrantLock fairLock = new ReentrantLock(false);
```

运行结果如下：

```
t1获得锁！  
t3获得锁！  
t3获得锁！  
t3获得锁！  
t3获得锁！  
t1获得锁！  
t1获得锁！  
t1获得锁！  
t1获得锁！  
t2获得锁！  
t2获得锁！  
t2获得锁！  
t2获得锁！  
t2获得锁！  
t3获得锁！
```

可以看到t3可能会连续获得锁，结果是比较随机的，不公平的。

ReentrantLock获取锁的过程是可中断的

对于synchronized关键字，如果一个线程在等待获取锁，最终只有2种结果：

1. 要么获取到锁然后继续后面的操作
2. 要么一直等待，直到其他线程释放锁为止

而ReentrantLock提供了另外一种可能，就是在等待获取锁的过程中（发起获取锁请求到还未获取到锁这段时间内）是可以被中断的，也就是说在等待锁的过程中，程序可以根据需要取消获取锁的请求。有些使用这个操作是非常有必要的。比如：你和好朋友越好一起去打球，如果你等了半小时朋友还没到，突然你接到一个电话，朋友由于突发状况，不能来了，那么你一定达到回府。中断操作正是提供了一套类似的机制，如果一个线程正在等待获取锁，那么它依然可以收到一个通知，被告知无需等待，可以停止工作了。

示例代码：

```
package com.itsoku.chat06;  
  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.locks.ReentrantLock;  
  
/**  
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */  
public class Demo6 {  
    private static ReentrantLock lock1 = new ReentrantLock(false);  
    private static ReentrantLock lock2 = new ReentrantLock(false);  
  
    public static class T extends Thread {  
        int lock;  
  
        public T(String name, int lock) {  
            super(name);  
            this.lock = lock;  
        }  
  
        @Override  
        public void run() {
```

```

        try {
            if (this.lock == 1) {
                lock1.lockInterruptibly();
                TimeUnit.SECONDS.sleep(1);
                lock2.lockInterruptibly();
            } else {
                lock2.lockInterruptibly();
                TimeUnit.SECONDS.sleep(1);
                lock1.lockInterruptibly();
            }
        } catch (InterruptedException e) {
            System.out.println("中断标志：" + this.isInterrupted());
            e.printStackTrace();
        } finally {
            if (lock1.isHeldByCurrentThread()) {
                lock1.unlock();
            }
            if (lock2.isHeldByCurrentThread()) {
                lock2.unlock();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T("t1", 1);
        T t2 = new T("t2", 2);

        t1.start();
        t2.start();
    }
}

```

先运行一下上面代码，发现程序无法结束，使用jstack查看线程堆栈信息，发现2个线程死锁了。

```

Found one Java-level deadlock:
=====
"t2":
  waiting for ownable synchronizer 0x0000000717380c20, (a
  java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "t1"
"t1":
  waiting for ownable synchronizer 0x0000000717380c50, (a
  java.util.concurrent.locks.ReentrantLock$NonfairSync),
  which is held by "t2"

```

lock1被线程t1占用，lock2被线程t2占用，线程t1在等待获取lock2，线程t2在等待获取lock1，都在相互等待获取对方持有的锁，最终产生了死锁，如果是在synchronized关键字情况下发生了死锁现象，程序是无法结束的。

我们对上面代码改造一下，线程t2一直无法获取到lock1，那么等待5秒之后，我们中断获取锁的操作。主要修改一下main方法，如下：

```
T t1 = new T("t1", 1);
T t2 = new T("t2", 2);

t1.start();
t2.start();

TimeUnit.SECONDS.sleep(5);
t2.interrupt();
```

新增了2行代码 `TimeUnit.SECONDS.sleep(5); t2.interrupt();`，程序可以结束了，运行结果：

```
java.lang.InterruptedException
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireInterruptibly(AbstractQueuedSynchronizer.java:898)
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireInterruptibly(AbstractQueuedSynchronizer.java:1222)
    at
java.util.concurrent.locks.ReentrantLock.lockInterruptibly(ReentrantLock.java:335)
    at com.itsoku.chat06.Demo6$T.run(Demo6.java:31)
中断标志:false
```

从上面信息中可以看出，代码的31行触发了异常，**中断标志输出：false**

```
28 } else {
29     lock2.lockInterruptibly();
30     TimeUnit.SECONDS.sleep(timeout);
31     lock1.lockInterruptibly();
32 }
```

t2在31行一直获取不到lock1的锁，主线程中等待了5秒之后，t2线程调用了interrupt()方法，将线程的中断标志置为true，此时31行会触发InterruptedException异常，然后线程t2可以继续向下执行，释放了lock2的锁，然后线程t1可以正常获取锁，程序得以继续进行。线程发送中断信号触发InterruptedException异常之后，中断标志将被清空。

关于获取锁的过程中被中断，注意几点：

1. ReentrantLock中必须使用实例方法 lockInterruptibly() 获取锁时，在线程调用interrupt()方法之后，才会引发 InterruptedException 异常
 2. 线程调用interrupt()之后，线程的中断标志会被置为true
 3. 触发InterruptedException异常之后，线程的中断标志会被清空，即置为false
 4. 所以当线程调用interrupt()引发InterruptedException异常，中断标志的变化是:false->true->false

ReentrantLock锁申请等待限时

申请锁等待限时是什么意思？一般情况下，获取锁的时间我们是不知道的，`synchronized`关键字获取锁的过程中，只能等待其他线程把锁释放之后才能够有机会获取到锁。所以获取锁的时间有长有短。如果获取锁的时间能够设置超时时间，那就非常好了。

ReentrantLock刚好提供了这样功能，给我们提供了获取锁限时等待的方法 `tryLock()`，可以选择传入时间参数，表示等待指定的时间，无参则表示立即返回锁申请的结果：true表示获取锁成功，false表示获取锁失败。

tryLock无参方法

看一下源码中tryLock方法：

```
public boolean tryLock()
```

返回boolean类型的值，此方法会立即返回，结果表示获取锁是否成功，示例：

```
package com.itsoku.chat06;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo8 {
    private static ReentrantLock lock1 = new ReentrantLock(false);

    public static class T extends Thread {

        public T(String name) {
            super(name);
        }

        @Override
        public void run() {
            try {
                System.out.println(System.currentTimeMillis() + ":" +
this.getName() + "开始获取锁!");
                //获取锁超时时间设置为3秒，3秒内是否能否获取锁都会返回
                if (lock1.tryLock()) {
                    System.out.println(System.currentTimeMillis() + ":" +
this.getName() + "获取到了锁!");
                    //获取到锁之后，休眠5秒
                    TimeUnit.SECONDS.sleep(5);
                } else {
                    System.out.println(System.currentTimeMillis() + ":" +
this.getName() + "未能获取到锁!");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (lock1.isHeldByCurrentThread()) {
                    lock1.unlock();
                }
            }
        }
    }
}
```

```
public static void main(String[] args) throws InterruptedException {
    T t1 = new T("t1");
    T t2 = new T("t2");

    t1.start();
    t2.start();
}
```

代码中获取锁成功之后，休眠5秒，会导致另外一个线程获取锁失败，运行代码，输出：

```
1563356291081:t2开始获取锁!
1563356291081:t2获取到了锁!
1563356291081:t1开始获取锁!
1563356291081:t1未能获取到锁!
```

可以看到t2获取成功，t1获取失败了，tryLock()是立即响应的，中间不会有阻塞。

tryLock有参方法

可以明确设置获取锁的超时时间，该方法签名：

```
public boolean tryLock(long timeout, TimeUnit unit) throws InterruptedException
```

该方法在指定的时间内不管是否可以获取锁，都会返回结果，返回true，表示获取锁成功，返回false表示获取失败。此方法有2个参数，第一个参数是时间类型，是一个枚举，可以表示时、分、秒、毫秒等待，使用比较方便，第1个参数表示在时间类型上的时间长短。此方法在执行的过程中，如果调用了线程的中断interrupt()方法，会触发InterruptedException异常。

示例：

```
package com.itsoku.chat06;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo7 {
    private static ReentrantLock lock1 = new ReentrantLock(false);

    public static class T extends Thread {

        public T(String name) {
            super(name);
        }

        @Override
        public void run() {
            try {
                lock1.tryLock(5, TimeUnit.SECONDS);
                System.out.println(Thread.currentThread().getName() + " 获取到了锁");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        System.out.println(System.currentTimeMillis() + ":" +
this.getName() + "开始获取锁!");
        //获取锁超时时间设置为3秒，3秒内是否能否获取锁都会返回
        if (lock1.tryLock(3, TimeUnit.SECONDS)) {
            System.out.println(System.currentTimeMillis() + ":" +
this.getName() + "获取到了锁!");
            //获取到锁之后，休眠5秒
            TimeUnit.SECONDS.sleep(5);
        } else {
            System.out.println(System.currentTimeMillis() + ":" +
this.getName() + "未能获取到锁!");
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        if (lock1.isHeldByCurrentThread()) {
            lock1.unlock();
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    T t1 = new T("t1");
    T t2 = new T("t2");

    t1.start();
    t2.start();
}
}

```

程序中调用了ReentrantLock的实例方法 `tryLock(3, TimeUnit.SECONDS)`，表示获取锁的超时时间是3秒，3秒后不管是否能否获取锁，该方法都会有返回值，获取到锁之后，内部休眠了5秒，会导致另一个线程获取锁失败。

运行程序，输出：

```

1563355512901:t2开始获取锁!
1563355512901:t1开始获取锁!
1563355512902:t2获取到了锁!
1563355515904:t1未能获取到锁!

```

输出结果中分析，t2获取到锁了，然后休眠了5秒，t1获取锁失败，t1打印了2条信息，时间相差3秒左右。

关于tryLock()方法和tryLock(long timeout, TimeUnit unit)方法，说明一下：

1. 都会返回boolean值，结果表示获取锁是否成功
2. tryLock()方法，不管是否获取成功，都会立即返回；而有参的tryLock方法会尝试在指定的时间内去获取锁，中间会阻塞的现象，在指定的时间之后会不管是否能够获取锁都会返回结果
3. tryLock()方法不会响应线程的中断方法；而有参的tryLock方法会响应线程的中断方法，而触发 `InterruptedException` 异常，这个从2个方法的声明上可以看出来

ReentrantLock其他常用的方法

1. isHeldByCurrentThread：实例方法，判断当前线程是否持有ReentrantLock的锁，上面代码中有使用过。

获取锁的4种方法对比

获取锁的方法	是否立即响应(不会阻塞)	是否响应中断
lock()	×	×
lockInterruptibly()	×	√
tryLock()	√	×
tryLock(long timeout, TimeUnit unit)	×	√

总结

1. ReentrantLock可以实现公平锁和非公平锁
2. ReentrantLock默认实现的是非公平锁
3. ReentrantLock的获取锁和释放锁必须成对出现，锁了几次，也要释放几次
4. 释放锁的操作必须放在finally中执行
5. lockInterruptibly()实例方法可以相应线程的中断方法，调用线程的interrupt()方法时，lockInterruptibly()方法会触发 InterruptedException 异常
6. 关于 InterruptedException 异常说一下，看到方法声明上带有 throws InterruptedException，表示该方法可以相应线程中断，调用线程的interrupt()方法时，这些方法会触发 InterruptedException 异常，触发InterruptedException时，线程的中断中断状态会被清除。所以如果程序由于调用 interrupt() 方法而触发 InterruptedException 异常，线程的标志由默认的false变为true，然后又变为false
7. 实例方法tryLock()会尝试获取锁，会立即返回，返回值表示是否获取成功
8. 实例方法tryLock(long timeout, TimeUnit unit)会在指定的时间内尝试获取锁，指定的时间内是否能够获取锁，都会返回，返回值表示是否获取锁成功，该方法会响应线程的中断

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第13篇：JUC中的Condition

本文目标

1. synchronized中实现线程等待和唤醒
2. Condition简介及常用方法介绍及相关示例
3. 使用Condition实现生产者消费者
4. 使用Condition实现同步阻塞队列

Object对象中的wait(), notify()方法，用于线程等待和唤醒等待中的线程，大家应该比较熟悉，想再次了解的朋友可以移步到[线程的基本操作](#)

synchronized中等待和唤醒线程示例

```
package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo1 {
    static Object lock = new Object();

    public static class T1 extends Thread {
        @Override
        public void run() {
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "准备获取锁！");
            synchronized (lock) {
                System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "获取锁成功！");
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

```

        }
    }
    System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "释放锁成功!");
}
}

public static class T2 extends Thread {
    @Override
    public void run() {
        System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "准备获取锁!");
        synchronized (lock) {
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "获取锁成功!");
            lock.notify();
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ " notify!");
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "准备释放锁!");
        }
        System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "释放锁成功!");
    }
}

public static void main(String[] args) throws InterruptedException {
    T1 t1 = new T1();
    t1.setName("t1");
    t1.start();
    TimeUnit.SECONDS.sleep(5);
    T2 t2 = new T2();
    t2.setName("t2");
    t2.start();
}
}

```

输出：

```

1: 1563530109234,t1准备获取锁!
2: 1563530109234,t1获取锁成功!
3: 1563530114236,t2准备获取锁!
4: 1563530114236,t2获取锁成功!
5: 1563530114236,t2 notify!
6: 1563530119237,t2准备释放锁!
7: 1563530119237,t2释放锁成功!
8: 1563530119237,t1释放锁成功!

```

代码结合输出的结果我们分析一下：

- 线程t1先获取锁，然后调用了wait()方法将线程置为等待状态，然后会释放lock的锁
- 主线程等待5秒之后，启动线程t2，t2获取到了锁，结果中1、3行时间相差5秒左右

3. t2调用lock.notify()方法，准备将等待在lock上的线程t1唤醒，notify()方法之后又休眠了5秒，看一下输出的5、8可知，notify()方法之后，t1并不能立即被唤醒，需要等到t2将synchronized块执行完毕，释放锁之后，t1才被唤醒
4. wait()方法和notify()方法必须放在同步块内调用（synchronized块内），否则会报错

Condition使用简介

在了解Condition之前，需要先了解一下重入锁ReentrantLock，可以移步到：[JUC中的ReentrantLock](#)。

任何一个java对象都天然继承于Object类，在线程间实现通信的往往应用到Object的几个方法，比如wait()、wait(long timeout)、wait(long timeout, int nanos)与notify()、notifyAll()几个方法实现等待/通知机制，同样的，在java Lock体系下依然会有同样的方法实现等待/通知机制。

从整体上来看**Object的wait和notify/notify是与对象监视器配合完成线程间的等待/通知机制，而Condition与Lock配合完成等待通知机制，前者是java底层级别的，后者是语言级别的，具有更高的可控制性和扩展性**。两者除了在使用方式上不同外，在**功能特性**上还是有很多的不同：

1. Condition能够支持不响应中断，而通过使用Object方式不支持
2. Condition能够支持多个等待队列（new 多个Condition对象），而Object方式只能支持一个
3. Condition能够支持超时时间的设置，而Object不支持

Condition由ReentrantLock对象创建，并且可以同时创建多个，Condition接口在使用前必须先调用ReentrantLock的lock()方法获得锁，之后调用Condition接口的await()将释放锁，并且在该Condition上等待，直到有其他线程调用Condition的signal()方法唤醒线程，使用方式和wait()、notify()类似。

示例代码：

```
package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo2 {
    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    public static class T1 extends Thread {
        @Override
        public void run() {
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "准备获取锁！");
            lock.lock();
            try {
                System.out.println(System.currentTimeMillis() + "," + 
this.getName() + "获取锁成功！");
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }
}
```

```

        }
        System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "释放锁成功!");
    }

    public static class T2 extends Thread {
        @Override
        public void run() {
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "准备获取锁!");
            lock.lock();
            try {
                System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "获取锁成功!");
                condition.signal();
                System.out.println(System.currentTimeMillis() + "," + this.getName()
+ " signal!");
                try {
                    TimeUnit.SECONDS.sleep(5);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "准备释放锁!");
            } finally {
                lock.unlock();
            }
            System.out.println(System.currentTimeMillis() + "," + this.getName()
+ "释放锁成功!");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T1 t1 = new T1();
        t1.setName("t1");
        t1.start();
        TimeUnit.SECONDS.sleep(5);
        T2 t2 = new T2();
        t2.setName("t2");
        t2.start();
    }
}

```

输出：

```

1563532185827,t1准备获取锁!
1563532185827,t1获取锁成功!
1563532190829,t2准备获取锁!
1563532190829,t2获取锁成功!
1563532190829,t2 signal!
1563532195829,t2准备释放锁!
1563532195829,t2释放锁成功!
1563532195829,t1释放锁成功!

```

输出的结果和使用synchronized关键字的实例类似。

Condition.await()方法和Object.wait()方法类似，当使用Condition.await()方法时，需要先获取Condition对象关联的ReentrantLock的锁，在Condition.await()方法被调用时，当前线程会释放这个锁，并且当前线程会进行等待（处于阻塞状态）。在signal()方法被调用后，系统会从Condition对象的等待队列中唤醒一个线程，一旦线程被唤醒，被唤醒的线程会尝试重新获取锁，一旦获取成功，就可以继续执行了。因此，在signal被调用后，一般需要释放相关的锁，让给其他被唤醒的线程，让他可以继续执行。

Condition常用方法

Condition接口提供的常用方法有：

和Object中wait类似的方法

1. void await() throws InterruptedException: 当前线程进入等待状态，如果其他线程调用condition的signal或者signalAll方法并且当前线程获取Lock从await方法返回，如果在等待状态中被中断会被抛出被中断异常；
2. long awaitNanos(long nanosTimeout): 当前线程进入等待状态直到被通知，中断或者超时；
3. boolean await(long time, TimeUnit unit) throws InterruptedException: 同第二种，支持自定义时间单位，false: 表示方法超时之后自动返回的，true: 表示等待还未超时时，await方法就返回了（超时之前，被其他线程唤醒了）
4. boolean awaitUntil(Date deadline) throws InterruptedException: 当前线程进入等待状态直到被通知，中断或者到了某个时间
5. void awaitUninterruptibly(): 当前线程进入等待状态，不会响应线程中断操作，只能通过唤醒的方式让线程继续

和Object的notify/notifyAll类似的方法

1. void signal(): 唤醒一个等待在condition上的线程，将该线程从等待队列中转移到同步队列中，如果在同步队列中能够竞争到Lock则可以从等待方法中返回。
2. void signalAll(): 与1的区别在于能够唤醒所有等待在condition上的线程

Condition.await()过程中被打断

```
package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo4 {
    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    public static class T1 extends Thread {
        @Override
        public void run() {
            lock.lock();
            try {
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("T1 执行完毕");
        }
    }

    public static void main(String[] args) {
        T1 t1 = new T1();
        t1.start();
        try {
            condition.signal();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("主线程执行完毕");
    }
}
```

```

        condition.await();
    } catch (InterruptedException e) {
        System.out.println("中断标志: " + this.isInterrupted());
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public static void main(String[] args) throws InterruptedException {
    T1 t1 = new T1();
    t1.setName("t1");
    t1.start();
    TimeUnit.SECONDS.sleep(2);
    //给t1线程发送中断信号
    System.out.println("1、t1中断标志: " + t1.isInterrupted());
    t1.interrupt();
    System.out.println("2、t1中断标志: " + t1.isInterrupted());
}
}

```

输出：

```

1、t1中断标志: false
2、t1中断标志: true
中断标志: false
java.lang.InterruptedException
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.reportInte
rruptAfterWait(AbstractQueuedSynchronizer.java:2014)
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Abst
ractQueuedSynchronizer.java:2048)
    at com.itsoku.chat09.Demo4$T1.run(Demo4.java:19)

```

调用condition.await()之后，线程进入阻塞中，调用t1.interrupt()，给t1线程发送中断信号，await()方法内部会检测到线程中断信号，然后触发 InterruptedException 异常，线程中断标志被清除。从输出结果中可以看出，线程t1中断标志的变换过程：false->true->false

await(long time, TimeUnit unit)超时之后自动返回

```

package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo5 {
    static ReentrantLock lock = new ReentrantLock();

```

```

static Condition condition = lock.newCondition();

public static class T1 extends Thread {
    @Override
    public void run() {
        lock.lock();
        try {
            System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",start");
            boolean r = condition.await(2, TimeUnit.SECONDS);
            System.out.println(r);
            System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    T1 t1 = new T1();
    t1.setName("t1");
    t1.start();
}
}

```

输出：

```

1563541624082,t1,start
false
1563541626085,t1,end

```

t1线程等待2秒之后，自动返回继续执行，最后await方法返回false，**await返回false表示超时之后自动返回**

await(long time, TimeUnit unit)超时之前被唤醒

```

package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo6 {
    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    public static class T1 extends Thread {

```

```

@Override
public void run() {
    lock.lock();
    try {
        System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",start");
        boolean r = condition.await(5, TimeUnit.SECONDS);
        System.out.println(r);
        System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",end");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        lock.unlock();
    }
}

public static void main(String[] args) throws InterruptedException {
    T1 t1 = new T1();
    t1.setName("t1");
    t1.start();
    //休眠1秒之后，唤醒t1线程
    TimeUnit.SECONDS.sleep(1);
    lock.lock();
    try {
        condition.signal();
    } finally {
        lock.unlock();
    }
}
}

```

输出：

```

1563542046046,t1,start
true
1563542047048,t1,end

```

t1线程中调用 `condition.await(5, TimeUnit.SECONDS)`; 方法会释放锁，等待5秒，主线程休眠1秒，然后获取锁，之后调用`signal()`方法唤醒t1，输出结果中发现await后过了1秒（1、3行输出结果的时间差），`await`方法就返回了，并且返回值是true。**true表示await方法超时之前被其他线程唤醒了。**

long awaitNanos(long nanosTimeout)超时返回

```

package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */

```

```

/*
public class Demo7 {
    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    public static class T1 extends Thread {
        @Override
        public void run() {
            lock.lock();
            try {
                System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",start");
                long r = condition.awaitNanos(TimeUnit.SECONDS.toNanos(5));
                System.out.println(r);
                System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",end");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T1 t1 = new T1();
        t1.setName("t1");
        t1.start();
    }
}

```

输出：

```

1563542547302,t1,start
-258200
1563542552304,t1,end

```

awaitNanos参数为纳秒，可以调用TimeUnit中的一些方法将时间转换为纳秒。

t1调用await方法等待5秒超时返回，返回结果为负数，表示超时之后返回的。

waitNanos(long nanosTimeout)超时之前被唤醒

```

package com.itsoku.chat09;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo8 {
    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();
}

```

```

public static class T1 extends Thread {
    @Override
    public void run() {
        lock.lock();
        try {
            System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",start");
            long r = condition.awaitNanos(TimeUnit.SECONDS.toNanos(5));
            System.out.println(r);
            System.out.println(System.currentTimeMillis() + "," +
this.getName() + ",end");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            lock.unlock();
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    T1 t1 = new T1();
    t1.setName("t1");
    t1.start();
    //休眠1秒之后，唤醒t1线程
    TimeUnit.SECONDS.sleep(1);
    lock.lock();
    try {
        condition.signal();
    } finally {
        lock.unlock();
    }
}
}

```

输出：

```

1563542915991,t1,start
3999988500
1563542916992,t1,end

```

t1中调用await休眠5秒，主线程休眠1秒之后，调用signal()唤醒线程t1，await方法返回正数，表示返回时距离超时时间还有多久，将近4秒，返回正数表示，线程在超时之前被唤醒了。

其他几个有参的await方法和无参的await方法一样，线程调用interrupt()方法时，这些方法都会触发InterruptedException异常，并且线程的中断标志会被清除。

同一个锁支持创建多个Condition

使用两个Condition来实现一个阻塞队列的例子：

```

package com.itsoku.chat09;

import java.util.LinkedList;

```

```

import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class BlockingQueueDemo<E> {
    int size;//阻塞队列最大容量

    ReentrantLock lock = new ReentrantLock();

    LinkedList<E> list = new LinkedList<E>();//队列底层实现

    Condition notFull = lock.newCondition();//队列满时的等待条件
    Condition notEmpty = lock.newCondition();//队列空时的等待条件

    public BlockingQueueDemo(int size) {
        this.size = size;
    }

    public void enqueue(E e) throws InterruptedException {
        lock.lock();
        try {
            while (list.size() == size)//队列已满，在notFull条件下等待
                notFull.await();
            list.add(e); //入队：加入链表末尾
            System.out.println("入队：" + e);
            notEmpty.signal(); //通知在notEmpty条件下等待的线程
        } finally {
            lock.unlock();
        }
    }

    public E dequeue() throws InterruptedException {
        E e;
        lock.lock();
        try {
            while (list.size() == 0)//队列为空，在notEmpty条件下等待
                notEmpty.await();
            e = list.removeFirst(); //出队：移除链表首元素
            System.out.println("出队：" + e);
            notFull.signal(); //通知在notFull条件下等待的线程
            return e;
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        BlockingQueueDemo<Integer> queue = new BlockingQueueDemo<Integer>(2);
        for (int i = 0; i < 10; i++) {
            int data = i;
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        queue.enqueue(data);
                    }
                }
            }).start();
        }
    }
}

```

```
        } catch (InterruptedException e) {  
            }  
        }  
    }).start();  
}  
for (int i = 0; i < 10; i++) {  
    new Thread(new Runnable() {  
        @Override  
        public void run() {  
            try {  
                Integer data = queue.dequeue();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }).start();  
}  
}
```

代码非常容易理解，创建了一个阻塞队列，大小为3，队列满的时候，会被阻塞，等待其他线程去消费，队列中的元素被消费之后，会唤醒生产者，生产数据进入队列。上面代码将队列大小置为1，可以实现同步阻塞队列，生产1个元素之后，生产者会被阻塞，待消费者消费队列中的元素之后，生产者才能继续工作。

Object的监视器方法与Condition接口的对比

对比项	Object 监视器方法	Condition
前置条件	获取对象的锁	调用Lock.lock获取锁，调用Lock.newCondition()获取Condition对象
调用方式	直接调用，如：object.wait()	直接调用，如：condition.await()
等待队列个数	一个	多个，使用多个condition实现
当前线程释放锁并进入等待状态	支持	支持
当前线程释放锁进入等待状态中不响应中断	不支持	支持
当前线程释放锁并进入超时等待状态	支持	支持
当前线程释放锁并进入等待状态到将来某个时间	不支持	支持
唤醒等待队列中的一个线程	支持	支持
唤醒等待队列中的全部线程	支持	支持

总结

1. 使用condition的步骤：创建condition对象，获取锁，然后调用condition的方法
2. 一个ReentrantLock支持多个condition对象
3. `void await() throws InterruptedException;`方法会释放锁，让当前线程等待，支持唤醒，支持线程中断
4. `void awaitUninterruptibly();`方法会释放锁，让当前线程等待，支持唤醒，不支持线程中断
5. `long awaitNanos(long nanosTimeout) throws InterruptedException;`参数为纳秒，此方法会释放锁，让当前线程等待，支持唤醒，支持中断。超时之后返回的，结果为负数；超时之前返回的，结果为正数（表示返回时距离超时时间相差的纳秒数）
6. `boolean await(long time, TimeUnit unit) throws InterruptedException;`方法会释放锁，让当前线程等待，支持唤醒，支持中断。超时之后返回的，结果为false；超时之前返回的，结果为true
7. `boolean awaitUntil(Date deadline) throws InterruptedException;`参数表示超时的截止时间点，方法会释放锁，让当前线程等待，支持唤醒，支持中断。超时之后返回的，结果为false；超时之前返回的，结果为true
8. `void signal();`会唤醒一个等待中的线程，然后被唤醒的线程会被加入同步队列，去尝试获取锁
9. `void signalAll();`会唤醒所有等待中的线程，将所有等待中的线程加入同步队列，然后去尝试获取锁

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第14篇：JUC中的LockSupport工具类，必备技能

这是java高并发系列第14篇文章。

本文主要内容：

1. 讲解3种让线程等待和唤醒的方法，每种方法配合具体的示例
2. 介绍LockSupport主要用法
3. 对比3种方式，了解他们之间的区别

LockSupport位于java.util.concurrent（简称juc）包中，算是juc中一个基础类，juc中很多地方都会使用LockSupport，非常重要，希望大家一定要掌握。

关于线程等待/唤醒的方法，前面的文章中我们已经讲过2种了：

1. 方式1：使用Object中的wait()方法让线程等待，使用Object中的notify()方法唤醒线程
2. 方式2：使用juc包中Condition的await()方法让线程等待，使用signal()方法唤醒线程

这两种方式，我们先来看一下示例。

使用Object类中的方法实现线程等待和唤醒

示例1：

```
package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo1 {

    static Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            synchronized (lock) {
                System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
                try {
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
            }
        });
        t1.setName("t1");
    }
}
```

```
t1.start();
//休眠5秒
TimeUnit.SECONDS.sleep(5);
synchronized (lock) {
    lock.notify();
}
}
```

输出：

```
1563592938744,t1 start!
1563592943745,t1 被唤醒!
```

t1线程中调用`lock.wait()`方法让t1线程等待，主线程中休眠5秒之后，调用`lock.notify()`方法唤醒了t1线程，输出的结果中，两行结果相差5秒左右，程序正常退出。

示例2

我们把上面代码中main方法内部改一下，删除了`synchronized`关键字，看看有什么效果：

```
package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo2 {

    static Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
        });
        t1.setName("t1");
        t1.start();
        //休眠5秒
        TimeUnit.SECONDS.sleep(5);
        lock.notify();
    }
}
```

运行结果：

```
Exception in thread "t1" java.lang.IllegalMonitorStateException
1563593178811,t1 start!
    at java.lang.Object.wait(Native Method)
    at java.lang.Object.wait(Object.java:502)
    at com.itsoku.chat10.Demo2.lambda$main$0(Demo2.java:16)
    at java.lang.Thread.run(Thread.java:745)
Exception in thread "main" java.lang.IllegalMonitorStateException
    at java.lang.Object.notify(Native Method)
    at com.itsoku.chat10.Demo2.main(Demo2.java:26)
```

上面代码中将`synchronized`去掉了，发现调用`wait()`方法和调用`notify()`方法都抛出了`IllegalMonitorStateException`异常，原因：**Object类中的wait、notify、notifyAll用于线程等待和唤醒的方法，都必须在同步代码中运行（必须用到关键字synchronized）。**

示例3

唤醒方法在等待方法之前执行，线程能够被唤醒么？代码如下：

```
package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo3 {

    static Object lock = new Object();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            synchronized (lock) {
                System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
                try {
                    //休眠3秒
                    lock.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
            }
        });
        t1.setName("t1");
        t1.start();
        //休眠1秒之后唤醒lock对象上等待的线程
        TimeUnit.SECONDS.sleep(1);
        synchronized (lock) {
            lock.notify();
        }
    }
}
```

```

        }
        System.out.println("lock.notify()执行完毕");
    }
}

```

运行代码，输出结果：

```

lock.notify()执行完毕
1563593869797,t1 start!

```

输出了上面2行之后，程序一直无法结束，t1线程调用wait()方法之后无法被唤醒了，从输出中可见，`notify()`方法在`wait()`方法之前执行了，等待的线程无法被唤醒了。说明：唤醒方法在等待方法之前执行，线程无法被唤醒。

关于Object类中的用户线程等待和唤醒的方法，总结一下：

1. `wait()/notify()/notifyAll()`方法都必须放在同步代码（必须在`synchronized`内部执行）中执行，需要先获取锁
2. 线程唤醒的方法（`notify`、`notifyAll`）需要在等待的方法（`wait`）之后执行，等待中的线程才可能被唤醒，否则无法唤醒

使用Condition实现线程的等待和唤醒

Condition的使用，前面的文章讲过，对这块不熟悉的可以移步[JUC中Condition的使用](#)，关于Condition我们准备了3个示例。

示例1

```

package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo4 {

    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            lock.lock();
            try {
                System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
                try {
                    condition.await();
                } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
} finally {
    lock.unlock();
}
});
t1.setName("t1");
t1.start();
//休眠5秒
TimeUnit.SECONDS.sleep(5);
lock.lock();
try {
    condition.signal();
} finally {
    lock.unlock();
}
}

}
}

```

输出：

```

1563594349632,t1 start!
1563594354634,t1 被唤醒!

```

t1线程启动之后调用 `condition.await()` 方法将线程处于等待中，主线程休眠5秒之后调用 `condition.signal()` 方法将t1线程唤醒成功，输出结果中2个时间戳相差5秒。

示例2

我们将上面代码中的`lock.lock()`、`lock.unlock()`去掉，看看会发生什么。代码：

```

package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo5 {

    static ReentrantLock lock = new ReentrantLock();
    static Condition condition = lock.newCondition();

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
            try {
                condition.await();
            } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
}
t1.setName("t1");
t1.start();
//休眠5秒
TimeUnit.SECONDS.sleep(5);
condition.signal();
}
}

```

输出：

```

Exception in thread "t1" java.lang.IllegalMonitorStateException
1563594654865,t1 start!
at
java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:151)
at
java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSync
hronizer.java:1261)
at
java.util.concurrent.locks.AbstractQueuedSynchronizer.fullyRelease(AbstractQueue
dSynchronizer.java:1723)
at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Abst
ractQueuedSynchronizer.java:2036)
at com.itsoku.chat10.Demo5.lambda$main$0(Demo5.java:19)
at java.lang.Thread.run(Thread.java:745)
Exception in thread "main" java.lang.IllegalMonitorStateException
at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.signal(Abs
tractQueuedSynchronizer.java:1939)
at com.itsoku.chat10.Demo5.main(Demo5.java:29)

```

有异常发生，`condition.await();`和`condition.signal();`都触发了`IllegalMonitorStateException`异常。原因：调用**condition**中线程等待和唤醒的方法的前提是必须先获取**lock**的锁。

示例3

唤醒代码在等待之前执行，线程能够被唤醒么？代码如下：

```

package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo6 {

```

```

static ReentrantLock lock = new ReentrantLock();
static Condition condition = lock.newCondition();

public static void main(String[] args) throws InterruptedException {
    Thread t1 = new Thread(() -> {
        try {
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.lock();
        try {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
            try {
                condition.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
        } finally {
            lock.unlock();
        }
    });
    t1.setName("t1");
    t1.start();
    //休眠5秒
    TimeUnit.SECONDS.sleep(1);
    lock.lock();
    try {
        condition.signal();
    } finally {
        lock.unlock();
    }
    System.out.println(System.currentTimeMillis() + ",condition.signal();执行
完毕");
}
}

```

运行结果:

```

1563594886532,condition.signal();执行完毕
1563594890532,t1 start!

```

输出上面2行之后，程序无法结束，代码结合输出可以看出signal()方法在await()方法之前执行的，最终t1线程无法被唤醒，导致程序无法结束。

关于Condition中方法使用总结：

1. 使用Condition中的线程等待和唤醒方法之前，需要先获取锁。否则会报
`IllegalMonitorStateException` 异常
2. signal()方法先于await()方法之前调用，线程无法被唤醒

Object和Condition的局限性

关于Object和Condition中线程等待和唤醒的局限性，有以下几点：

1. 2中方式中的让线程等待和唤醒的方法能够执行的先决条件是：线程需要先获取锁
2. 唤醒方法需要在等待方法之后调用，线程才能够被唤醒

关于这2点，LockSupport都不需要，就能实现线程的等待和唤醒。下面我们来说一下LockSupport类。

LockSupport类介绍

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

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

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

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

LockSupport中常用的方法

阻塞线程

- void park(): 阻塞当前线程，如果调用**unpark方法**或者**当前线程被中断**，从能从park()方法中返回
- void park(Object blocker): 功能同方法1，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查
- void parkNanos(long nanos): 阻塞当前线程，最长不超过nanos纳秒，增加了超时返回的特性
- void parkNanos(Object blocker, long nanos): 功能同方法3，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查
- void parkUntil(long deadline): 阻塞当前线程，直到deadline，deadline是一个绝对时间，表示某个时间的毫秒格式
- void parkUntil(Object blocker, long deadline): 功能同方法5，入参增加一个Object对象，用来记录导致线程阻塞的阻塞对象，方便进行问题排查；

唤醒线程

- void unpark(Thread thread): 唤醒处于阻塞状态的指定线程

示例1

主线程线程等待5秒之后，唤醒t1线程，代码如下：

```
package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.LockSupport;
import java.util.concurrent.locks.ReentrantLock;
```

```

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo7 {

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
            LockSupport.park();
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
        });
        t1.setName("t1");
        t1.start();
        //休眠5秒
        TimeUnit.SECONDS.sleep(5);
        LockSupport.unpark(t1);
        System.out.println(System.currentTimeMillis() + ",LockSupport.unpark();
执行完毕");
    }
}

```

输出：

```

1563597664321,t1 start!
1563597669323,LockSupport.unpark();执行完毕
1563597669323,t1 被唤醒!

```

t1中调用 LockSupport.park(); 让当前线程t1等待，主线程休眠了5秒之后，调用 LockSupport.unpark(t1); 将t1线程唤醒，输出结果中1、3行结果相差5秒左右，说明t1线程等待5秒之后，被唤醒了。

LockSupport.park(); 无参数，内部直接会让当前线程处于等待中；unpark方法传递了一个线程对象作为参数，表示将对应的线程唤醒。

示例2

唤醒方法放在等待方法之前执行，看一下线程是否能够被唤醒呢？代码如下：

```

package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.LockSupport;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo8 {

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

```

```

Thread t1 = new Thread(() -> {
    try {
        TimeUnit.SECONDS.sleep(5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
    LockSupport.park();
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
}
t1.setName("t1");
t1.start();
//休眠1秒
TimeUnit.SECONDS.sleep(1);
LockSupport.unpark(t1);
System.out.println(System.currentTimeMillis() + ",LockSupport.unpark(); 执行完毕");
}
}

```

输出：

```

1563597994295,LockSupport.unpark();执行完毕
1563597998296,t1 start!
1563597998296,t1 被唤醒!

```

代码中启动t1线程，t1线程内部休眠了5秒，然后主线程休眠1秒之后，调用了 LockSupport.unpark(t1); 唤醒线程t1，此时 LockSupport.park(); 方法还未执行，说明唤醒方法在等待方法之前执行的；输出结果中2、3行结果时间一样，表示 LockSupport.park(); 没有阻塞了，是立即返回的。

说明：唤醒方法在等待方法之前执行，线程也能够被唤醒，这点是另外2中方法无法做到的。Object和Condition中的唤醒必须在等待之后调用，线程才能被唤醒。而LockSupport中，唤醒的方法不管是在等待之前还是在等待之后调用，线程都能够被唤醒。

示例3

park()让线程等待之后，是否能够响应线程中断？代码如下：

```

package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.LockSupport;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo9 {

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> {

```

```

        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " start!");
        System.out.println(Thread.currentThread().getName() + ",park()之前中
断标志: " + Thread.currentThread().isInterrupted());
        LockSupport.park();
        System.out.println(Thread.currentThread().getName() + ",park()之后中
断标志: " + Thread.currentThread().isInterrupted());
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + " 被唤醒!");
    });
    t1.setName("t1");
    t1.start();
    //休眠5秒
    TimeUnit.SECONDS.sleep(5);
    t1.interrupt();
}

}

```

输出：

```

1563598536736,t1 start!
t1,park()之前中断标志: false
t1,park()之后中断标志: true
1563598541736,t1 被唤醒!

```

t1线程中调用了park()方法让线程等待，主线程休眠了5秒之后，调用 t1.interrupt(); 给线程t1发送中断信号，然后线程t1从等待中被唤醒了，输出结果中的1、4行结果相差5秒左右，刚好是主线程休眠了5秒之后将t1唤醒了。结论： park方法可以相应线程中断。

LockSupport.park方法让线程等待之后，唤醒方式有2种：

1. 调用**LockSupport.unpark方法**
2. 调用**等待线程的 interrupt() 方法，给等待的线程发送中断信号，可以唤醒线程**

示例4

LockSupport有几个阻塞放有一个blocker参数，这个参数什么意思，上一个实例代码，大家一看就懂了：

```

package com.itsoku.chat10;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.LockSupport;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo10 {

    static class BlockerDemo {

    }

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

```

```

        Thread t1 = new Thread(() -> {
            LockSupport.park();
        });
        t1.setName("t1");
        t1.start();

        Thread t2 = new Thread(() -> {
            LockSupport.park(new BlockerDemo());
        });
        t2.setName("t2");
        t2.start();
    }
}

```

运行上面代码，然后用jstack查看一下线程的堆栈信息：

```

"t2" #13 prio=5 os_prio=0 tid=0x00000000293ea800 nid=0x91e0 waiting on condition
[0x0000000029c3f000]
    java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for <0x00000007180bfeb0> (a
com.itsoku.chat10.Demo10$BlockerDemo)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
        at com.itsoku.chat10.Demo10.lambda$main$1(Demo10.java:22)
        at com.itsoku.chat10.Demo10$$Lambda$2/824909230.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)

"t1" #12 prio=5 os_prio=0 tid=0x00000000293ea000 nid=0x9d4 waiting on condition
[0x0000000029b3f000]
    java.lang.Thread.State: WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        at java.util.concurrent.locks.LockSupport.park(LockSupport.java:304)
        at com.itsoku.chat10.Demo10.lambda$main$0(Demo10.java:16)
        at com.itsoku.chat10.Demo10$$Lambda$1/1389133897.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)

```

代码中，线程t1和t2的不同点是，t2中调用park方法传入了一个BlockerDemo对象，从上面的线程堆栈信息中，发现t2线程的堆栈信息中多了一行- parking to wait for <0x00000007180bfeb0> (a com.itsoku.chat10.Demo10\$BlockerDemo)，刚好是传入的BlockerDemo对象，park传入的这个参数可以让我们在线程堆栈信息中方便排查问题，其他暂无他用。

LockSupport的其他等待方法，包含有超时时间了，过了超时时间，等待方法会自动返回，让线程继续运行，这些方法在此就不提供示例了，有兴趣的朋友可以自己动手练一练。

线程等待和唤醒的3种方式做个对比

到目前为止，已经说了3种让线程等待和唤醒的方法了

1. 方式1：Object中的wait、notify、notifyAll方法
2. 方式2：juc中Condition接口提供的await、signal、signalAll方法
3. 方式3：juc中的LockSupport提供的park、unpark方法

3种方式对比：

	Object	Condition	LockSupport
前置条件	需要在synchronized中运行	需要先获取Lock的锁	无
无限等待	支持	支持	支持
超时等待	支持	支持	支持
等待到将来某个时间返回	不支持	支持	支持
等待状态中释放锁	会释放	会释放	不会释放
唤醒方法先于等待方法执行，能否唤醒线程	否	否	可以
是否能响应线程中断	是	是	是
线程中断是否会清除中断标志	是	是	否
是否支持等待状态中不响应中断	不支持	支持	不支持

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第15篇：UC中的Semaphore（信号量）

Semaphore（信号量）为多线程协作提供了更为强大的控制方法，前面的文章中我们学了synchronized和重入锁ReentrantLock，这2种锁一次都只能允许一个线程访问一个资源，而信号量可以控制有多少个线程可以同时访问特定的资源。

Semaphore常用场景：限流

举个例子：

比如有个停车场，有5个空位，门口有个门卫，手中5把钥匙分别对应5个车位上面的锁，来一辆车，门卫会给司机一把钥匙，然后进去找到对应的车位停下来，出去的时候司机将钥匙归还给门卫。停车场生意比较好，同时来了100辆车，门卫手中只有5把钥匙，同时只能放5辆车进入，其他车只能等待，等有人将钥匙归还给门卫之后，才能让其他车辆进入。

上面的例子中门卫就相当于Semaphore，车钥匙就相当于许可证，车就相当于线程。

Semaphore主要方法

- **Semaphore(int permits)**: 构造方法，参数表示许可证数量，用来创建信号量
- **Semaphore(int permits,boolean fair)**: 构造方法，当fair等于true时，创建具有给定许可数的计数信号量并设置为公平信号量
- **void acquire() throws InterruptedException**: 从此信号量获取1个许可前线程将一直阻塞，相当于一辆车占了一个车位，此方法会响应线程中断，表示调用线程的interrupt方法，会使该方法抛出InterruptedException异常
- **void acquire(int permits) throws InterruptedException** : 和acquire()方法类似，参数表示需要获取许可的数量；比如一个大卡车要入停车场，由于车比较大，需要申请3个车位才可以停放
- **void acquireUninterruptibly(int permits)** : 和acquire(int permits) 方法类似，只是不会响应线程中断
- **boolean tryAcquire()**: 尝试获取1个许可，不管是否能够获取成功，都立即返回，true表示获取成功，false表示获取失败
- **boolean tryAcquire(int permits)**: 和tryAcquire()，表示尝试获取permits个许可
- **boolean tryAcquire(long timeout, TimeUnit unit) throws InterruptedException**: 尝试在指定的时间内获取1个许可，获取成功返回true，指定的时间过后还是无法获取许可，返回false
- **boolean tryAcquire(int permits, long timeout, TimeUnit unit) throws InterruptedException**: 和tryAcquire(long timeout, TimeUnit unit)类似，多了一个permits参数，表示尝试获取permits个许可
- **void release()**: 释放一个许可，将其返回给信号量，相当于车从停车场出去时将钥匙归还给门卫
- **void release(int n)**: 释放n个许可
- **int availablePermits()**: 当前可用的许可数

示例1：Semaphore简单的使用

```
package com.itsoku.chat12;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo1 {
    static Semaphore semaphore = new Semaphore(2);

    public static class T extends Thread {
        public T(String name) {
```

```

        super(name);
    }

    @Override
    public void run() {
        Thread thread = Thread.currentThread();
        try {
            semaphore.acquire();
            System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",获取许可!");
            TimeUnit.SECONDS.sleep(3);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaphore.release();
            System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",释放许可!");
        }
    }
}

public static void main(String[] args) throws InterruptedException {
    for (int i = 0; i < 10; i++) {
        new T("t-" + i).start();
    }
}
}

```

输出：

```

1563715791327,t-0,获取许可!
1563715791327,t-1,获取许可!
1563715794328,t-0,释放许可!
1563715794328,t-5,获取许可!
1563715794328,t-1,释放许可!
1563715794328,t-2,获取许可!
1563715797328,t-2,释放许可!
1563715797328,t-6,获取许可!
1563715797328,t-5,释放许可!
1563715797328,t-3,获取许可!
1563715800329,t-6,释放许可!
1563715800329,t-9,获取许可!
1563715800329,t-3,释放许可!
1563715800329,t-7,获取许可!
1563715803330,t-7,释放许可!
1563715803330,t-8,获取许可!
1563715803330,t-9,释放许可!
1563715803330,t-4,获取许可!
1563715806330,t-8,释放许可!
1563715806330,t-4,释放许可!

```

代码中 `new Semaphore(2)` 创建了许可数量为2的信号量，每个线程获取1个许可，同时允许两个线程获取许可，从输出中也可以看出，同时有两个线程可以获取许可，其他线程需要等待已获取许可的线程释放许可之后才能运行。为获取到许可的线程会阻塞在 `acquire()` 方法上，直到获取到许可才能继续。

示例2：获取许可之后不释放

门卫 (Semaphore) 有点呆，司机进去的时候给了钥匙，出来的时候不归还，门卫也不会说什么。最终结果就是其他车辆都无法进入了。

如下代码：

```
package com.itsoku.chat12;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo2 {
    static Semaphore semaphore = new Semaphore(2);

    public static class T extends Thread {
        public T(String name) {
            super(name);
        }

        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            try {
                semaphore.acquire();
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ", 获取许可！");
                TimeUnit.SECONDS.sleep(3);
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ", 运行结束！");
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ", 当前可用许可数量：" + semaphore.availablePermits());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 10; i++) {
            new T("t-" + i).start();
        }
    }
}
```

输出：

```
1563716603924,t-0,获取许可!
1563716603924,t-1,获取许可!
1563716606925,t-0,运行结束!
1563716606925,t-0,当前可用许可数量:0
1563716606925,t-1,运行结束!
1563716606925,t-1,当前可用许可数量:0
```

上面程序运行后一直无法结束，观察一下代码，代码中获取许可后，没有释放许可的代码，最终导致，可用许可数量为0，其他线程无法获取许可，会在 `semaphore.acquire()` 处等待，导致程序无法结束。

示例3：释放许可正确的姿势

示例1中，在finally里面释放锁，会有问题么？

如果获取锁的过程中发生异常，导致获取锁失败，最后finally里面也释放了许可，最终会怎么样，导致许可数量凭空增长了。

示例代码：

```
package com.itsoku.chat12;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo3 {
    static Semaphore semaphore = new Semaphore(1);

    public static class T extends Thread {
        public T(String name) {
            super(name);
        }

        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            try {
                semaphore.acquire();
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",获取许可,当前可用许可数量:" + semaphore.availablePermits());
                //休眠100秒
                TimeUnit.SECONDS.sleep(100);
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",运行结束!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                semaphore.release();
            }
            System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",当前可用许可数量:" + semaphore.availablePermits());
        }
    }
}
```

```

    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T("t1");
        t1.start();
        //休眠1秒
        TimeUnit.SECONDS.sleep(1);
        T t2 = new T("t2");
        t2.start();
        //休眠1秒
        TimeUnit.SECONDS.sleep(1);
        T t3 = new T("t3");
        t3.start();

        //给t2和t3发送中断信号
        t2.interrupt();
        t3.interrupt();
    }
}

```

输出：

```

1563717279058,t1,获取许可,当前可用许可数量:0
java.lang.InterruptedException
1563717281060,t2,当前可用许可数量:1
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly(AbstractQueuedSynchronizer.java:998)
1563717281060,t3,当前可用许可数量:2
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.java:1304)
        at java.util.concurrent.Semaphore.acquire(Semaphore.java:312)
        at com.itsoku.chat12.Demo3$T.run(Demo3.java:21)
java.lang.InterruptedException
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly(AbstractQueuedSynchronizer.java:1302)
        at java.util.concurrent.Semaphore.acquire(Semaphore.java:312)
        at com.itsoku.chat12.Demo3$T.run(Demo3.java:21)

```

程序中信号量许可数量为1，创建了3个线程获取许可，线程t1获取成功了，然后休眠100秒。其他两个线程阻塞在 `semaphore.acquire()` 方法处，代码中对线程t2、t3发送中断信号，我们看一下 `Semaphore` 中 `acquire` 的源码：

```

public void acquire() throws InterruptedException

```

这个方法会响应线程中断，主线程中对t2、t3发送中断信号之后，`acquire()` 方法会触发 `InterruptedException` 异常，t2、t3最终没有获取到许可，但是他们都执行了finally中的释放许可的操作，最后导致许可数量变为了2，导致许可数量增加了。所以程序中释放许可的方式有问题。需要改进一下，获取许可成功才去释放锁。

正确的释放锁的方式，如下：

```

package com.itsoku.chat12;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！
 */
public class Demo4 {
    static Semaphore semaphore = new Semaphore(1);

    public static class T extends Thread {
        public T(String name) {
            super(name);
        }

        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            //获取许可是否成功
            boolean acquireSuccess = false;
            try {
                semaphore.acquire();
                acquireSuccess = true;
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",获取许可,当前可用许可数量:" + semaphore.availablePermits());
                //休眠100秒
                TimeUnit.SECONDS.sleep(5);
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",运行结束!");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (acquireSuccess) {
                    semaphore.release();
                }
            }
            System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",当前可用许可数量:" + semaphore.availablePermits());
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T("t1");
        t1.start();
        //休眠1秒
        TimeUnit.SECONDS.sleep(1);
        T t2 = new T("t2");
        t2.start();
        //休眠1秒
        TimeUnit.SECONDS.sleep(1);
        T t3 = new T("t3");
        t3.start();

        //给t2和t3发送中断信号
        t2.interrupt();
        t3.interrupt();
    }
}

```

```
    }  
}
```

输出：

```
1563717751655,t1,获取许可,当前可用许可数量:0  
1563717753657,t3,当前可用许可数量:0  
java.lang.InterruptedException  
1563717753657,t2,当前可用许可数量:0  
    at  
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly  
(AbstractQueuedSynchronizer.java:1302)  
        at java.util.concurrent.Semaphore.acquire(Semaphore.java:312)  
        at com.itsoku.chat12.Demo4$T.run(Demo4.java:23)  
java.lang.InterruptedException  
    at  
java.util.concurrent.locks.AbstractQueuedSynchronizer.doAcquireSharedInterruptibly  
(AbstractQueuedSynchronizer.java:998)  
    at  
java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireSharedInterruptibly  
(AbstractQueuedSynchronizer.java:1304)  
        at java.util.concurrent.Semaphore.acquire(Semaphore.java:312)  
        at com.itsoku.chat12.Demo4$T.run(Demo4.java:23)  
1563717756656,t1,运行结束!  
1563717756656,t1,当前可用许可数量:1
```

程序中增加了一个变量 `acquiresuccess` 用来标记获取许可是否成功，在 `finally` 中根据这个变量是否为 `true`，来确定是否释放许可。

示例4：在规定的时间内希望获取许可

司机来到停车场，发现停车场已经满了，只能在外等待内部的车出来之后才能进去，但是要等多久，他自己也不知道，他希望等10分钟，如果还是无法进去，就不到这里停车了。

Semaphore内部2个方法可以提供超时获取许可的功能：

```
public boolean tryAcquire(long timeout, TimeUnit unit) throws  
InterruptedException  
public boolean tryAcquire(int permits, long timeout, TimeUnit unit)  
    throws InterruptedException
```

在指定的时间内去尝试获取许可，如果能够获取到，返回`true`，获取不到返回`false`。

示例代码：

```
package com.itsoku.chat12;  
  
import java.util.concurrent.Semaphore;  
import java.util.concurrent.TimeUnit;  
  
/**  
 * 微信公众号：路人甲Java，专注于java技术分享（带你玩转 爬虫、分布式事务、异步消息服务、任务调度、分库分表、大数据等），喜欢请关注！  
 */
```

```

public class Demo5 {
    static Semaphore semaphore = new Semaphore(1);

    public static class T extends Thread {
        public T(String name) {
            super(name);
        }

        @Override
        public void run() {
            Thread thread = Thread.currentThread();
            //获取许可是否成功
            boolean acquireSuccess = false;
            try {
                //尝试在1秒内获取许可，获取成功返回true，否则返回false
                System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",尝试获取许可,当前可用许可数量:" + semaphore.availablePermits());
                acquireSuccess = semaphore.tryAcquire(1, TimeUnit.SECONDS);
                //获取成功执行业务代码
                if (acquireSuccess) {
                    System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",获取许可成功,当前可用许可数量:" + semaphore.availablePermits());
                    //休眠5秒
                    TimeUnit.SECONDS.sleep(5);
                } else {
                    System.out.println(System.currentTimeMillis() + "," +
thread.getName() + ",获取许可失败,当前可用许可数量:" + semaphore.availablePermits());
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (acquireSuccess) {
                    semaphore.release();
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        T t1 = new T("t1");
        t1.start();
        //休眠1秒
        TimeUnit.SECONDS.sleep(1);
        T t2 = new T("t2");
        t2.start();
        //休眠1秒
        TimeUnit.SECONDS.sleep(1);
        T t3 = new T("t3");
        t3.start();
    }
}

```

输出：

```
1563718410202,t1,尝试获取许可,当前可用许可数量:1  
1563718410202,t1,获取许可成功,当前可用许可数量:0  
1563718411203,t2,尝试获取许可,当前可用许可数量:0  
1563718412203,t3,尝试获取许可,当前可用许可数量:0  
1563718412204,t2,获取许可失败,当前可用许可数量:0  
1563718413204,t3,获取许可失败,当前可用许可数量:0
```

代码中许可数量为1，`semaphore.tryAcquire(1, TimeUnit.SECONDS);`：表示尝试在1秒内获取许可，获取成功立即返回true，超过1秒还是获取不到，返回false。线程t1获取许可成功，之后休眠了5秒，从输出中可以看出t2和t3都尝试了1秒，获取失败。

其他一些使用说明

1. Semaphore默认创建的是非公平的信号量，什么意思呢？这个涉及到公平与不公平。举个例子：5个车位，允许5个车辆进去，来了100辆车，只能进去5辆，其他95在外面排队等着。里面刚好出来了1辆，此时刚好又来了10辆车，这10辆车是直接插队到其他95辆前面去，还是到95辆后面去排队呢？排队就表示公平，直接去插队争抢第一个，就表示不公平。对于停车场，排队肯定更好一些咯。不过对于信号量来说不公平的效率更高一些，所以默认是不公平的。
2. 建议阅读以下Semaphore的源码，对常用的方法有个了解，不需要都记住，用的时候也方便查询就好。
3. 方法中带有`throws InterruptedException`声明的，表示这个方法会响应线程中断信号，什么意思？表示调用线程的`interrupt()`方法，会让这些方法触发`InterruptedException`异常，即使这些方法处于阻塞状态，也会立即返回，并抛出`InterruptedException`异常，线程中断信号也会被清除。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

第16篇：JUC中等待多线程完成的工具类 CountDownLatch

本篇内容

1. 介绍CountDownLatch及使用场景
2. 提供几个使用示例介绍CountDownLatch的使用
3. 手写一个并行处理任务的工具类

假如有这样一个需求，当我们需要解析一个Excel里多个sheet的数据时，可以考虑使用多线程，每个线程解析一个sheet里的数据，等到所有的sheet都解析完之后，程序需要统计解析总耗时。分析一下：解析每个sheet耗时可能不一样，总耗时就是最长耗时的那个操作。

我们能够想到的最简单的做法是使用join，代码如下：

```
package com.itsoku.chat13;

import java.util.concurrent.TimeUnit;

/**
 * 微信公众号: javacode2018, 获取年薪50万课程
 */
public class Demo1 {

    public static class T extends Thread {
        //休眠时间（秒）
        int sleepSeconds;

        public T(String name, int sleepSeconds) {
            super(name);
            this.sleepSeconds = sleepSeconds;
        }

        @Override
        public void run() {
            Thread ct = Thread.currentThread();
            long startTime = System.currentTimeMillis();
            System.out.println(startTime + "," + ct.getName() + ",开始处理!");
            try {
                //模拟耗时操作，休眠sleepSeconds秒
                TimeUnit.SECONDS.sleep(this.sleepSeconds);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            long endTime = System.currentTimeMillis();
            System.out.println(endTime + "," + ct.getName() + ",处理完毕,耗时:" +
                (endTime - startTime));
        }
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        T t1 = new T("解析sheet1线程", 2);
```

```

        t1.start();

        T t2 = new T("解析sheet2线程", 5);
        t2.start();

        t1.join();
        t2.join();
        Long endTime = System.currentTimeMillis();
        System.out.println("总耗时：" + (endTime - startTime));

    }
}

```

输出：

```

1563767560271, 解析sheet1线程, 开始处理!
1563767560272, 解析sheet2线程, 开始处理!
1563767562273, 解析sheet1线程, 处理完毕, 耗时:2002
1563767565274, 解析sheet2线程, 处理完毕, 耗时:5002
总耗时:5005

```

代码中启动了2个解析sheet的线程，第一个耗时2秒，第二个耗时5秒，最终结果中总耗时：5秒。上面的关键技术点是线程的 `join()` 方法，此方法会让当前线程等待被调用的线程完成之后才能继续。可以看一下join的源码，内部其实是在synchronized方法中调用了线程的wait方法，最后被调用的线程执行完毕之后，由jvm自动调用其notifyAll()方法，唤醒所有等待中的线程。这个notifyAll()方法是由jvm内部自动调用的，jdk源码中是看不到的，需要看jvm源码，有兴趣的同学可以去查一下。所以JDK不推荐在线程上调用wait、notify、notifyAll方法。

而在JDK1.5之后的并发包中提供的CountDownLatch也可以实现join的这个功能。

CountDownLatch介绍：

CountDownLatch称之为闭锁，它可以使一个或一批线程在闭锁上等待，等到其他线程执行完相应操作后，闭锁打开，这些等待的线程才可以继续执行。确切的说，闭锁在内部维护了一个倒计数器。通过该计数器的值来决定闭锁的状态，从而决定是否允许等待的线程继续执行。

常用方法：

public CountDownLatch(int count): 构造方法，count表示计数器的值，不能小于0，否者会报异常。

public void await() throws InterruptedException: 调用await()会让当前线程等待，直到计数器为0的时候，方法才会返回，此方法会响应线程中断操作。

public boolean await(long timeout, TimeUnit unit) throws InterruptedException: 限时等待，在超时之前，计数器变为了0，方法返回true，否者直到超时，返回false，此方法会响应线程中断操作。

public void countDown(): 让计数器减1

CountDownLatch使用步骤：

1. 创建CountDownLatch对象
2. 调用其实例方法 `await()`, 让当前线程等待
3. 调用 `countDown()` 方法, 让计数器减1
4. 当计数器变为0的时候, `await()` 方法会返回

示例1：一个简单的示例

我们使用CountDownLatch来完成上面示例中使用join实现的功能，代码如下：

```
package com.itsoku.chat13;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号: javacode2018, 获取年薪50万课程
 */
public class Demo2 {

    public static class T extends Thread {
        //休眠时间（秒）
        int sleepSeconds;
        CountDownLatch countDownLatch;

        public T(String name, int sleepSeconds, CountDownLatch countDownLatch) {
            super(name);
            this.sleepSeconds = sleepSeconds;
            this.countDownLatch = countDownLatch;
        }

        @Override
        public void run() {
            Thread ct = Thread.currentThread();
            long startTime = System.currentTimeMillis();
            System.out.println(startTime + "," + ct.getName() + ",开始处理!");
            try {
                //模拟耗时操作, 休眠sleepSeconds秒
                TimeUnit.SECONDS.sleep(this.sleepSeconds);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                countDownLatch.countDown();
            }
            long endTime = System.currentTimeMillis();
            System.out.println(endTime + "," + ct.getName() + ",处理完毕,耗时:" +
            (endTime - startTime));
        }
    }

    public static void main(String[] args) throws InterruptedException {
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + "线程 start!");
        CountDownLatch countDownLatch = new CountDownLatch(2);
```

```

long starTime = System.currentTimeMillis();
T t1 = new T("解析sheet1线程", 2, countDownLatch);
t1.start();

T t2 = new T("解析sheet2线程", 5, countDownLatch);
t2.start();

countDownLatch.await();
System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + "线程 end!");
long endTime = System.currentTimeMillis();
System.out.println("总耗时:" + (endTime - starTime));

}
}

```

输出：

```

1563767580511,main线程 start!
1563767580513,解析sheet1线程,开始处理!
1563767580513,解析sheet2线程,开始处理!
1563767582515,解析sheet1线程,处理完毕,耗时:2002
1563767585515,解析sheet2线程,处理完毕,耗时:5002
1563767585515,main线程 end!
总耗时:5003

```

从结果中看出，效果和join实现的效果一样，代码中创建了计数器为2的 CountDownLatch，主线程中调用 countDownLatch.await(); 会让主线程等待，t1、t2线程中模拟执行耗时操作，最终在finally中调用了 countDownLatch.countDown(); 此方法每调用一次，CountDownLatch内部计数器会减1，当计数器变为0的时候，主线程中的await()会返回，然后继续执行。注意：上面的 countDown() 这个是必须要执行的方法，所以放在finally中执行。

示例2：等待指定的时间

还是上面的示例，2个线程解析2个sheet，主线程等待2个sheet解析完成。主线程说，我等待2秒，你们还是无法处理完成，就不等待了，直接返回。如下代码：

```

package com.itsoku.chat13;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号: javacode2018, 获取年薪50万课程
 */
public class Demo3 {

    public static class T extends Thread {
        //休眠时间（秒）
        int sleepSeconds;
        CountDownLatch countDownLatch;
    }
}

```

```

public T(String name, int sleepSeconds, CountDownLatch countDownLatch) {
    super(name);
    this.sleepSeconds = sleepSeconds;
    this.countDownLatch = countDownLatch;
}

@Override
public void run() {
    Thread ct = Thread.currentThread();
    long startTime = System.currentTimeMillis();
    System.out.println(startTime + "," + ct.getName() + ",开始处理!");
    try {
        //模拟耗时操作，休眠sleepSeconds秒
        TimeUnit.SECONDS.sleep(this.sleepSeconds);
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        countDownLatch.countDown();
    }
    long endTime = System.currentTimeMillis();
    System.out.println(endTime + "," + ct.getName() + ",处理完毕,耗时:" +
(endTime - startTime));
}
}

public static void main(String[] args) throws InterruptedException {
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + "线程 start!");
    CountDownLatch countDownLatch = new CountDownLatch(2);

    long startTime = System.currentTimeMillis();
    T t1 = new T("解析sheet1线程", 2, countDownLatch);
    t1.start();

    T t2 = new T("解析sheet2线程", 5, countDownLatch);
    t2.start();

    boolean result = countDownLatch.await(2, TimeUnit.SECONDS);

    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + "线程 end!");
    long endTime = System.currentTimeMillis();
    System.out.println("主线程耗时:" + (endTime - startTime) + ",result:" +
result);
}
}

```

输出：

```

1563767637316,main线程 start!
1563767637320,解析sheet1线程,开始处理!
1563767637320,解析sheet2线程,开始处理!
1563767639321,解析sheet1线程,处理完毕,耗时:2001
1563767639322,main线程 end!
主线程耗时:2004,result:false
1563767642322,解析sheet2线程,处理完毕,耗时:5002

```

从输出结果中可以看出，线程2耗时了5秒，主线程耗时了2秒，主线程中调用 `countDownLatch.await(2, TimeUnit.SECONDS);`，表示最多等2秒，不管计数器是否为0，`await`方法都会返回，若等待时间内，计数器变为0了，立即返回true，否则超时后返回false。

示例3：2个CountDown结合使用的示例

有3个人参见跑步比赛，需要先等指令员发指令枪后才能开跑，所有人都跑完之后，指令员喊一声，大家跑完了。

示例代码：

```
package com.itsoku.chat13;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号：javacode2018，获取年薪50万课程
 */
public class Demo4 {

    public static class T extends Thread {
        //跑步耗时（秒）
        int runCostSeconds;
        CountDownLatch commanderCd;
        CountDownLatch countDown;

        public T(String name, int runCostSeconds, CountDownLatch commanderCd,
        CountDownLatch countDown) {
            super(name);
            this.runCostSeconds = runCostSeconds;
            this.commanderCd = commanderCd;
            this.countDown = countDown;
        }

        @Override
        public void run() {
            //等待指令员枪响
            try {
                commanderCd.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Thread ct = Thread.currentThread();
            long startTime = System.currentTimeMillis();
            System.out.println(startTime + "," + ct.getName() + "，开始跑！");
            try {
                //模拟耗时操作，休眠runCostSeconds秒
                TimeUnit.SECONDS.sleep(this.runCostSeconds);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                countDown.countDown();
            }
            long endTime = System.currentTimeMillis();
        }
    }
}
```

```

        System.out.println(endTime + "," + ct.getName() + "，跑步结束，耗时：" +
(endTime - startTime));
    }
}

public static void main(String[] args) throws InterruptedException {
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + "线程 start!");
    CountDownLatch commanderCd = new CountDownLatch(1);
    CountDownLatch countDownLatch = new CountDownLatch(3);

    long startTime = System.currentTimeMillis();
    T t1 = new T("小张", 2, commanderCd, countDownLatch);
    t1.start();

    T t2 = new T("小李", 5, commanderCd, countDownLatch);
    t2.start();

    T t3 = new T("路人甲", 10, commanderCd, countDownLatch);
    t3.start();

    //主线程休眠5秒，模拟指令员准备发枪耗时操作
    TimeUnit.SECONDS.sleep(5);
    System.out.println(System.currentTimeMillis() + "，枪响了，大家开始跑");
    commanderCd.countDown();

    countDownLatch.await();
    long endTime = System.currentTimeMillis();
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + "所有人跑完了，主线程耗时：" + (endTime -
startTime));
}
}

```

输出：

```

1563767691087,main线程 start!
1563767696092,枪响了，大家开始跑
1563767696092,小张，开始跑！
1563767696092,小李，开始跑！
1563767696092,路人甲，开始跑！
1563767698093,小张，跑步结束，耗时：2001
1563767701093,小李，跑步结束，耗时：5001
1563767706093,路人甲，跑步结束，耗时：10001
1563767706093,main所有人跑完了，主线程耗时：15004

```

代码中，t1、t2、t3启动之后，都阻塞在 commanderCd.await();，主线程模拟发枪准备操作耗时5秒，然后调用 commanderCd.countDown(); 模拟发枪操作，此方法被调用以后，阻塞在 commanderCd.await(); 的3个线程会向下执行。主线程调用 countDownLatch.await(); 之后进行等待，每个人跑完之后，调用 countDownLatch.countDown(); 通知一下 countDownLatch 让计数器减1，最后3个人都跑完了，主线程从 countDownLatch.await(); 返回继续向下执行。

手写一个并行处理任务的工具类

```
package com.itsoku.chat13;

import org.springframework.util.CollectionUtils;

import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.function.Consumer;
import java.util.stream.Collectors;
import java.util.stream.Stream;

/**
 * 微信公众号: javacode2018, 获取年薪50万课程
 */
public class TaskDisposeutils {

    //并行线程数
    public static final int POOL_SIZE;

    static {
        POOL_SIZE = Integer.max(Runtime.getRuntime().availableProcessors(), 5);
    }

    /**
     * 并行处理, 并等待结束
     *
     * @param taskList 任务列表
     * @param consumer 消费者
     * @param <T>
     * @throws InterruptedException
     */
    public static <T> void dispose(List<T> taskList, Consumer<T> consumer)
            throws InterruptedException {
        dispose(true, POOL_SIZE, taskList, consumer);
    }

    /**
     * 并行处理, 并等待结束
     *
     * @param moreThread 是否多线程执行
     * @param poolsize 线程池大小
     * @param taskList 任务列表
     * @param consumer 消费者
     * @param <T>
     * @throws InterruptedException
     */
    public static <T> void dispose(boolean moreThread, int poolsize, List<T>
            taskList, Consumer<T> consumer) throws InterruptedException {
        if (CollectionUtils.isEmpty(taskList)) {
            return;
        }
        if (moreThread && poolsize > 1) {
            poolsize = Math.min(poolsize, taskList.size());
            ExecutorService executorService = null;
            try {
                executorService = Executors.newFixedThreadPool(poolsize);
                taskList.forEach(executorService::execute);
                CountDownLatch latch = new CountDownLatch(poolsize);
                taskList.forEach(task -> latch.countDown());
                latch.await();
            } catch (Exception e) {
                e.printStackTrace();
            } finally {
                if (executorService != null) {
                    executorService.shutdown();
                }
            }
        }
    }
}
```

```

        CountDownLatch countDownLatch = new
CountDownLatch(taskList.size());
        for (T item : taskList) {
            executorService.execute(() -> {
                try {
                    consumer.accept(item);
                } finally {
                    countDownLatch.countDown();
                }
            });
        }
        countDownLatch.await();
    } finally {
        if (executorService != null) {
            executorService.shutdown();
        }
    }
}
} else {
    for (T item : taskList) {
        consumer.accept(item);
    }
}
}

public static void main(String[] args) throws InterruptedException {
    //生成1-10的10个数字，放在list中，相当于10个任务
    List<Integer> list = Stream.iterate(1, a -> a +
1).limit(10).collect(Collectors.toList());
    //启动多线程处理list中的数据，每个任务休眠时间为list中的数值
    TaskDisposeUtils.dispose(list, item -> {
        try {
            long startTime = System.currentTimeMillis();
            TimeUnit.SECONDS.sleep(item);
            long endTime = System.currentTimeMillis();

            System.out.println(System.currentTimeMillis() + ",任务" + item +
"执行完毕，耗时：" + (endTime - startTime));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    });
    //上面所有任务处理完毕之后，程序才能继续
    System.out.println(list + "中的任务都处理完毕！");
}
}

```

运行代码输出：

```
1563769828130,任务1执行完毕,耗时:1000
1563769829130,任务2执行完毕,耗时:2000
1563769830131,任务3执行完毕,耗时:3001
1563769831131,任务4执行完毕,耗时:4001
1563769832131,任务5执行完毕,耗时:5001
1563769833130,任务6执行完毕,耗时:6000
1563769834131,任务7执行完毕,耗时:7001
1563769835131,任务8执行完毕,耗时:8001
1563769837131,任务9执行完毕,耗时:9001
1563769839131,任务10执行完毕,耗时:10001
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]中的任务都处理完毕!
```

TaskDisposeUtils是一个并行处理的工具类，可以传入n个任务内部使用线程池进行处理，等待所有任务都处理完成之后，方法才会返回。比如我们发送短信，系统中有1万条短信，我们使用上面的工具，每次取100条并行发送，待100个都处理完毕之后，再取一批按照同样的逻辑发送。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第17篇：JUC中的循环栅栏CyclicBarrier

本文主要内容

1. 介绍CyclicBarrier
2. 6个示例介绍CyclicBarrier的使用
3. 对比CyclicBarrier和CountDownLatch

CyclicBarrier简介

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

CyclicBarrier通常称为循环屏障。它和CountDownLatch很相似，都可以使线程先等待然后再执行。不过CountDownLatch是使一批线程等待另一批线程执行完后再执行；而CyclicBarrier只是使等待的线程达到一定数目后再让它们继续执行。故而CyclicBarrier内部也有一个计数器，计数器的初始值在创建对象时通过构造参数指定，如下所示：

```
public CyclicBarrier(int parties) {  
    this(parties, null);  
}
```

每调用一次await()方法都将使阻塞的线程数+1，只有阻塞的线程数达到设定值时屏障才会打开，允许阻塞的所有线程继续执行。除此之外，CyclicBarrier还有几点需要注意的地方：

- CyclicBarrier的计数器可以重置而CountDownLatch不行，这意味着CyclicBarrier实例可以被重复使用而CountDownLatch只能被使用一次。而这也是循环屏障循环二字的语义所在。
- CyclicBarrier允许用户自定义barrierAction操作，这是个可选操作，可以在创建CyclicBarrier对象时指定

```
public CyclicBarrier(int parties, Runnable barrierAction) {  
    if (parties <= 0) throw new IllegalArgumentException();  
    this.parties = parties;  
    this.count = parties;  
    this.barrierCommand = barrierAction;  
}
```

一旦用户在创建CyclicBarrier对象时设置了barrierAction参数，则在阻塞线程数达到设定值屏障打开前，会调用barrierAction的run()方法完成用户自定义的操作。

示例1：简单使用CyclicBarrier

公司组织旅游，大家都有经历过，10个人，中午到饭点了，需要等到10个人都到了才能开饭，先到的人坐那等着，代码如下：

```
package com.itsoku.chat15;  
  
import java.util.concurrent.BrokenBarrierException;  
import java.util.concurrent.CyclicBarrier;  
import java.util.concurrent.TimeUnit;  
  
/**  
 * 微信公众号：javacode2018，获取年薪50万java课程  
 */  
public class Demo1 {  
    public static CyclicBarrier cyclicBarrier = new CyclicBarrier(10);  
  
    public static class T extends Thread {  
        int sleep;  
  
        public T(String name, int sleep) {  
            super(name);  
            this.sleep = sleep;  
        }  
  
        @Override
```

```

public void run() {
    try {
        //模拟休眠
        TimeUnit.SECONDS.sleep(sleep);
        long starTime = System.currentTimeMillis();
        //调用await()的时候，当前线程将被阻塞，需要等待其他员工都到达await了才能
        继续
        cyclicBarrier.await();
        long endTime = System.currentTimeMillis();
        System.out.println(this.getName() + ",sleep:" + this.sleep + " "
        等待了" + (endTime - starTime) + "(ms),开始吃饭了！");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
}
}

public static void main(String[] args) throws InterruptedException {
    for (int i = 1; i <= 10; i++) {
        new T("员工" + i, i).start();
    }
}
}

```

输出：

```

员工1,sleep:1 等待了9000(ms),开始吃饭了!
员工9,sleep:9 等待了1000(ms),开始吃饭了!
员工8,sleep:8 等待了2001(ms),开始吃饭了!
员工7,sleep:7 等待了3001(ms),开始吃饭了!
员工6,sleep:6 等待了4001(ms),开始吃饭了!
员工4,sleep:4 等待了6000(ms),开始吃饭了!
员工5,sleep:5 等待了5000(ms),开始吃饭了!
员工10,sleep:10 等待了0(ms),开始吃饭了!
员工2,sleep:2 等待了7999(ms),开始吃饭了!
员工3,sleep:3 等待了7000(ms),开始吃饭了!

```

代码中模拟了10个员工上桌吃饭的场景，等待所有员工都到齐了才能开始，可以看到第10个员工最慢，前面的都在等待第10个员工，员工1等待了9秒，上面代码中调用 `cyclicBarrier.await()` 会让当前线程等待。当10个员工都调用了 `cyclicBarrier.await()` 之后，所有处于等待中的员工都会被唤醒，然后继续运行。

示例2：循环使用CyclicBarrier

对示例1进行改造一下，吃完饭之后，所有人都去车上，待所有人都到车上之后，驱车去下一景点玩。

```

package com.itsoku.chat15;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;

```

```

/**
 * 微信公众号: javacode2018, 获取年薪50万java课程
 */
public class Demo2 {
    public static CyclicBarrier cyclicBarrier = new CyclicBarrier(10);

    public static class T extends Thread {
        int sleep;

        public T(String name, int sleep) {
            super(name);
            this.sleep = sleep;
        }

        //等待吃饭
        void eat() {
            try {
                //模拟休眠
                TimeUnit.SECONDS.sleep(sleep);
                long starTime = System.currentTimeMillis();
                //调用await()的时候, 当前线程将会被阻塞, 需要等待其他员工都到达await了才能
                继续
                cyclicBarrier.await();
                long endTime = System.currentTimeMillis();
                System.out.println(this.getName() + ",sleep:" + this.sleep + "
等待了" + (endTime - starTime) + "(ms),开始吃饭了!");

                //休眠sleep时间, 模拟当前员工吃饭耗时
                TimeUnit.SECONDS.sleep(sleep);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }

    //等待所有人到齐之后, 开车去下一站
    void drive() {
        try {
            long starTime = System.currentTimeMillis();
            //调用await()的时候, 当前线程将会被阻塞, 需要等待其他员工都到达await了才能
            继续
            cyclicBarrier.await();
            long endTime = System.currentTimeMillis();
            System.out.println(this.getName() + ",sleep:" + this.sleep + "
等待了" + (endTime - starTime) + "(ms),去下一景点的路上!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void run() {
        //等待所有人到齐之后吃饭, 先到的人坐那等着, 什么事情不要干
        this.eat();
    }
}

```

```

        //等待所有人到齐之后开车去下一景点，先到的人坐那等着，什么事情不要干
        this.drive();
    }

}

public static void main(String[] args) throws InterruptedException {
    for (int i = 1; i <= 10; i++) {
        new T("员工" + i, i).start();
    }
}
}

```

输出：

```

员工10,sleep:10 等待了0(ms),开始吃饭了!
员工5,sleep:5 等待了5000(ms),开始吃饭了!
员工6,sleep:6 等待了4000(ms),开始吃饭了!
员工9,sleep:9 等待了1001(ms),开始吃饭了!
员工4,sleep:4 等待了6000(ms),开始吃饭了!
员工3,sleep:3 等待了7000(ms),开始吃饭了!
员工1,sleep:1 等待了9001(ms),开始吃饭了!
员工2,sleep:2 等待了8000(ms),开始吃饭了!
员工8,sleep:8 等待了2001(ms),开始吃饭了!
员工7,sleep:7 等待了3000(ms),开始吃饭了!
员工10,sleep:10 等待了0(ms),去下一景点的路上!
员工1,sleep:1 等待了8998(ms),去下一景点的路上!
员工5,sleep:5 等待了4999(ms),去下一景点的路上!
员工4,sleep:4 等待了5999(ms),去下一景点的路上!
员工3,sleep:3 等待了6998(ms),去下一景点的路上!
员工2,sleep:2 等待了7998(ms),去下一景点的路上!
员工9,sleep:9 等待了999(ms),去下一景点的路上!
员工8,sleep:8 等待了1999(ms),去下一景点的路上!
员工7,sleep:7 等待了2999(ms),去下一景点的路上!
员工6,sleep:6 等待了3999(ms),去下一景点的路上!

```

坑，又是员工10最慢，要提升效率了，不能吃的太多，得减肥。

代码中CyclicBarrier相当于使用了2次，第一次用于等待所有人到达后开饭，第二次用于等待所有人上车后驱车去下一景点。注意一些先到的员工会在其他人到达之前，都处于等待状态

`(cyclicBarrier.await();会让当前线程阻塞)`，无法干其他事情，等到最后一个人到了会唤醒所有人，然后继续。

CyclicBarrier内部相当于有个计数器（构造方法传入的），每次调用`await();`后，计数器会减1，并且`await()`方法会让当前线程阻塞，等待计数器减为0的时候，所有在`await()`上等待的线程被唤醒，然后继续向下执行，此时计数器又会被还原为创建时的值，然后可以继续再次使用。

示例3：最后到的人给大家上酒，然后开饭

还是示例1中的例子，员工10是最后到达的，让所有人都久等了，那怎么办，得给所有人倒酒，然后开饭，代码如下：

```

package com.itsoku.chat15;

import java.util.concurrent.BrokenBarrierException;

```

```

import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号: javacode2018, 获取年薪50万java课程
 */
public class Demo3 {
    public static CyclicBarrier cyclicBarrier = new CyclicBarrier(10, () -> {
        //模拟倒酒, 花了2秒, 又得让其他9个人等2秒
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(Thread.currentThread().getName() + "说, 不好意思, 让大家
久等了, 给大家倒酒赔罪!");
    });

    public static class T extends Thread {
        int sleep;

        public T(String name, int sleep) {
            super(name);
            this.sleep = sleep;
        }

        @Override
        public void run() {
            try {
                //模拟休眠
                TimeUnit.SECONDS.sleep(sleep);
                long starTime = System.currentTimeMillis();
                //调用await()的时候, 当前线程将会被阻塞, 需要等待其他员工都到达await了才能
继续
                cyclicBarrier.await();
                long endTime = System.currentTimeMillis();
                System.out.println(this.getName() + ",sleep:" + this.sleep + "
等待了" + (endTime - starTime) + "(ms),开始吃饭了! ");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 10; i++) {
            new T("员工" + i, i).start();
        }
    }
}

```

输出:

```
员工10说，不好意思，让大家久等了，给大家倒酒赔罪！
员工10,sleep:10 等待了2000(ms),开始吃饭了！
员工1,sleep:1 等待了11000(ms),开始吃饭了！
员工2,sleep:2 等待了10000(ms),开始吃饭了！
员工5,sleep:5 等待了7000(ms),开始吃饭了！
员工7,sleep:7 等待了5000(ms),开始吃饭了！
员工9,sleep:9 等待了3000(ms),开始吃饭了！
员工4,sleep:4 等待了8000(ms),开始吃饭了！
员工3,sleep:3 等待了9001(ms),开始吃饭了！
员工8,sleep:8 等待了4001(ms),开始吃饭了！
员工6,sleep:6 等待了6001(ms),开始吃饭了！
```

代码中创建 `CyclicBarrier` 对象时，多传入了一个参数（内部是倒酒操作），先到的人先等待，待所有人都到齐之后，需要先给大家倒酒，然后唤醒所有等待中的人让大家开饭。从输出结果中我们发现，倒酒操作是由最后一个人操作的，最后一个人倒酒完毕之后，才唤醒所有等待中的其他员工，让大家开饭。

示例4：其中一个人等待中被打断了

员工5等待中，突然接了个电话，有点急事，然后就拿起筷子开吃了，其他人会怎么样呢？看着他吃什么？

代码如下：

```
package com.itsoku.chat15;

import java.sql.Time;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;

/**
 * 微信公众号: javacode2018, 获取年薪50万java课程
 */
public class Demo4 {
    public static CyclicBarrier cyclicBarrier = new CyclicBarrier(10);

    public static class T extends Thread {
        int sleep;

        public T(String name, int sleep) {
            super(name);
            this.sleep = sleep;
        }

        @Override
        public void run() {
            long startTime = 0, endTime = 0;
            try {
                //模拟休眠
                TimeUnit.SECONDS.sleep(sleep);
                startTime = System.currentTimeMillis();
                //调用await()的时候，当前线程将会被阻塞，需要等待其他员工都到达await了才能
                继续
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
        System.out.println(this.getName() + "到了！");
        cyclicBarrier.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
    endTime = System.currentTimeMillis();
    System.out.println(this.getName() + ",sleep:" + this.sleep + " 等待了"
        + (endTime - startTime) + "(ms),开始吃饭了!");
}
}

public static void main(String[] args) throws InterruptedException {
    for (int i = 1; i <= 10; i++) {
        int sleep = 0;
        if (i == 10) {
            sleep = 10;
        }
        T t = new T("员工" + i, sleep);
        t.start();
        if (i == 5) {
            //模拟员工5接了个电话，将自己等待吃饭给打断了
            TimeUnit.SECONDS.sleep(1);
            System.out.println(t.getName() + ",有点急事，我先开干了!");
            t.interrupt();
            TimeUnit.SECONDS.sleep(2);
        }
    }
}
```

输出：

员工4到了！
员工3到了！
员工5到了！
员工1到了！
员工2到了！
员工5,有点急事，我先开干了！

```
java.util.concurrent.BrokenBarrierException
员工1,sleep:0 等待了1001(ms),开始吃饭了!
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
员工3,sleep:0 等待了1001(ms),开始吃饭了!
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
员工4,sleep:0 等待了1001(ms),开始吃饭了!
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
员工2,sleep:0 等待了1001(ms),开始吃饭了!
员工5,sleep:0 等待了1002(ms),开始吃饭了!
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
```

```
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
java.lang.InterruptedException
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.reportInte
rruptAfterWait(AbstractQueuedSynchronizer.java:2014)
    at
java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(Abst
ractQueuedSynchronizer.java:2048)
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:234)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
java.util.concurrent.BrokenBarrierException
员工6到了!
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
员工9到了!
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
员工8到了!
员工7到了!
员工6,sleep:0 等待了0(ms),开始吃饭了!
员工7,sleep:0 等待了1(ms),开始吃饭了!
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
员工8,sleep:0 等待了1(ms),开始吃饭了!
员工9,sleep:0 等待了1(ms),开始吃饭了!
Disconnected from the target VM, address: '127.0.0.1:64413', transport: 'socket'
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo4$T.run(Demo4.java:31)
员工10到了!
员工10,sleep:10 等待了0(ms),开始吃饭了!
```

输出的信息看着有点乱，给大家理一理，员工5遇到急事，拿起筷子就是吃，这样好么，当然不好，他这么做了，后面看他这么做了都跟着这么做（这种场景是不是很熟悉，有一个人拿起筷子先吃起来，其他人都跟着上了），直接不等其他人了，拿起筷子就开吃了。CyclicBarrier遇到这种情况就是这么处理的。前面4个员工都在 `await()` 处等待着，员工5也在 `await()` 上等待着，等了1秒

`(TimeUnit.SECONDS.sleep(1))`，接了个电话，然后给员工5发送中断信号后
`(t.interrupt();)`，员工5的 `await()` 方法会触发 `InterruptedException` 异常，此时其他等待中的前4个员工，看着5开吃了，自己立即也不等了，内部从 `await()` 方法中触发

`BrokenBarrierException` 异常，然后也开吃了，后面的6/7/8/9/10员工来了以后发现大家都开吃了，自己也不等了，6-10员工调用 `await()` 直接抛出了 `BrokenBarrierException` 异常，然后继续向下。

结论：

1. 内部有一个人把规则破坏了（接收到中断信号），其他人都不按规则来了，不会等待了
2. 接收到中断信号的线程，`await`方法会触发`InterruptedException`异常，然后被唤醒向下运行
3. 其他等待中或者后面到达的线程，会在`await()`方法上触发 `BrokenBarrierException` 异常，然后继续执行

示例5：其中一个人只愿意等的5秒

基于示例1，员工1只愿意等的5秒，5s后如果大家还没到期，自己要开吃了，员工1开吃了，其他人会怎么样呢？

```
package com.itsoku.chat15;

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

/**
 * 微信公众号：javacode2018，获取年薪50万java课程
 */
public class Demo5 {
    public static CyclicBarrier cyclicBarrier = new CyclicBarrier(10);

    public static class T extends Thread {
        int sleep;

        public T(String name, int sleep) {
            super(name);
            this.sleep = sleep;
        }

        @Override
        public void run() {
            long startTime = 0, endTime = 0;
            try {
                //模拟休眠
                TimeUnit.SECONDS.sleep(sleep);
                startTime = System.currentTimeMillis();
                //调用await()的时候，当前线程将会被阻塞，需要等待其他员工都到达await了才能
                //继续
                System.out.println(this.getName() + "到了！");
                if (this.getName().equals("员工1")) {
                    cyclicBarrier.await(5, TimeUnit.SECONDS);
                } else {
                    cyclicBarrier.await();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        } catch (TimeoutException e) {
            e.printStackTrace();
        }
        endTime = System.currentTimeMillis();
        System.out.println(this.getName() + ",sleep:" + this.sleep + " 等待了" + (endTime - startTime) + "(ms),开始吃饭了!");
    }
}

public static void main(String[] args) throws InterruptedException {
    for (int i = 1; i <= 10; i++) {
        T t = new T("员工" + i, i);
        t.start();
    }
}
}

```

输出：

```

员工1到了!
员工2到了!
员工3到了!
员工4到了!
员工5到了!
员工6到了!
员工1,sleep:1 等待了5001(ms),开始吃饭了!
员工5,sleep:5 等待了1001(ms),开始吃饭了!
java.util.concurrent.TimeoutException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:257)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:435)
    at com.itsoku.chat15.Demo5$T.run(Demo5.java:32)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)
员工6,sleep:6 等待了2(ms),开始吃饭了!
java.util.concurrent.BrokenBarrierException
员工2,sleep:2 等待了4002(ms),开始吃饭了!
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
员工3,sleep:3 等待了3001(ms),开始吃饭了!
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
员工4,sleep:4 等待了2001(ms),开始吃饭了!
    at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)
java.util.concurrent.BrokenBarrierException
员工7到了!

```

```

at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
员工7,sleep:7 等待了0(ms),开始吃饭了!
at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)

员工8到了!
员工8,sleep:8 等待了0(ms),开始吃饭了!
java.util.concurrent.BrokenBarrierException
at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)

员工9到了!
java.util.concurrent.BrokenBarrierException
员工9,sleep:9 等待了0(ms),开始吃饭了!
at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)

java.util.concurrent.BrokenBarrierException
员工10到了!
at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
员工10,sleep:10 等待了0(ms),开始吃饭了!
at com.itsoku.chat15.Demo5$T.run(Demo5.java:34)

```

从输出结果中我们可以看到：1等待5秒之后，开吃了，其他等待人都开吃了，后面来的人不等待，直接开吃了。

员工1调用有参 `await` 方法等待5秒之后，触发了 `TimeoutException` 异常，然后继续向下运行，其他的在5开吃之前已经等了一会的的几个员工，他们看到5开吃了，自己立即不等待了，也开吃了（他们的 `await` 抛出了 `BrokenBarrierException` 异常）；还有几个员工在5开吃之后到达的，他们直接不等待了，直接抛出 `BrokenBarrierException` 异常，然后也开吃了。

结论：

1. 等待超时的方法

```

public int await(long timeout, TimeUnit unit) throws
InterruptedException,BrokenBarrierException,TimeoutException

```

2. 内部有一个人把规则破坏了（等待超时），其他人都不按规则来了，不会等待了
3. 等待超时的线程，`await`方法会触发`TimeoutException`异常，然后被唤醒向下运行
4. 其他等待中或者后面到达的线程，会在`await()`方法上触发`BrokenBarrierException`异常，然后继续执行

示例6：重建规则

示例5中改造一下，员工1等待5秒超时之后，开吃了，打破了规则，先前等待中的以及后面到达的都不按规则来了，都拿起筷子开吃。过了一会，导游重新告知大家，要按规则来，然后重建了规则，大家都按规则来了。

代码如下：

```

package com.itsoku.chat15;

```

```

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

/**
 * 微信公众号: javacode2018, 获取年薪50万java课程
 */
public class Demo6 {
    public static CyclicBarrier cyclicBarrier = new CyclicBarrier(10);

    //规则是否已重建
    public static boolean guizhe = false;

    public static class T extends Thread {
        int sleep;

        public T(String name, int sleep) {
            super(name);
            this.sleep = sleep;
        }

        @Override
        public void run() {
            long startTime = 0, endTime = 0;
            try {
                //模拟休眠
                TimeUnit.SECONDS.sleep(sleep);
                startTime = System.currentTimeMillis();
                //调用await()的时候, 当前线程将会被阻塞, 需要等待其他员工都到达await了才能
                继续
                System.out.println(this.getName() + "到了!");
                if (!guizhe) {
                    if (this.getName().equals("员工1")) {
                        cyclicBarrier.await(5, TimeUnit.SECONDS);
                    } else {
                        cyclicBarrier.await();
                    }
                } else {
                    cyclicBarrier.await();
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (BrokenBarrierException e) {
                e.printStackTrace();
            } catch (TimeoutException e) {
                e.printStackTrace();
            }
            endTime = System.currentTimeMillis();
            System.out.println(this.getName() + ",sleep:" + this.sleep + " 等待
了" + (endTime - startTime) + "(ms),开始吃饭了!");
        }
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 10; i++) {
            T t = new T("员工" + i, i);
        }
    }
}

```

```

        t.start();
    }

    //等待10秒之后，重置，重建规则
    TimeUnit.SECONDS.sleep(15);
    cyclicBarrier.reset();
    guizhe = true;
    System.out.println("-----大家太皮了，请大家按规则来-----");
}

//再来一次
for (int i = 1; i <= 10; i++) {
    T t = new T("员工" + i, i);
    t.start();
}
}
}
}

```

输出：

```

员工1到了!
员工2到了!
员工3到了!
员工4到了!
员工5到了!
java.util.concurrent.TimeoutException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:257)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:435)
    at com.itsoku.chat15.Demo6$T.run(Demo6.java:36)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:250)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo6$T.run(Demo6.java:38)

员工6到了!
员工1,sleep:1 等待了5002(ms),开始吃饭了!
员工6,sleep:6 等待了4(ms),开始吃饭了!
员工4,sleep:4 等待了2004(ms),开始吃饭了!
员工5,sleep:5 等待了1004(ms),开始吃饭了!
员工3,sleep:3 等待了3002(ms),开始吃饭了!
员工2,sleep:2 等待了4004(ms),开始吃饭了!
员工7到了!
员工7,sleep:7 等待了0(ms),开始吃饭了!

```

```
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo6$T.run(Demo6.java:38)
java.util.concurrent.BrokenBarrierException
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo6$T.run(Demo6.java:38)
员工8到了!
员工8,sleep:8 等待了0(ms),开始吃饭了!
java.util.concurrent.BrokenBarrierException
员工9到了!
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
员工9,sleep:9 等待了0(ms),开始吃饭了!
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo6$T.run(Demo6.java:38)
java.util.concurrent.BrokenBarrierException
员工10到了!
    at java.util.concurrent.CyclicBarrier.dowait(CyclicBarrier.java:207)
员工10,sleep:10 等待了0(ms),开始吃饭了!
    at java.util.concurrent.CyclicBarrier.await(CyclicBarrier.java:362)
    at com.itsoku.chat15.Demo6$T.run(Demo6.java:38)
-----大家太皮了,请大家按规则来-----
员工1到了!
员工2到了!
员工3到了!
员工4到了!
员工5到了!
员工6到了!
员工7到了!
员工8到了!
员工9到了!
员工10到了!
员工10,sleep:10 等待了0(ms),开始吃饭了!
员工1,sleep:1 等待了9000(ms),开始吃饭了!
员工2,sleep:2 等待了8000(ms),开始吃饭了!
员工3,sleep:3 等待了6999(ms),开始吃饭了!
员工7,sleep:7 等待了3000(ms),开始吃饭了!
员工6,sleep:6 等待了4000(ms),开始吃饭了!
员工5,sleep:5 等待了5000(ms),开始吃饭了!
员工4,sleep:4 等待了6000(ms),开始吃饭了!
员工9,sleep:9 等待了999(ms),开始吃饭了!
员工8,sleep:8 等待了1999(ms),开始吃饭了!
```

第一次规则被打乱了，过了一会导游重建了规则 (`cyclicBarrier.reset()`)，接着又重来了一次模拟等待吃饭的操作，正常了。

CountDownLatch和CyclicBarrier的区别

还是举例子说明一下：

CountDownLatch示例

主管相当于 **CountDownLatch**，干活的小弟相当于做事情的线程。

老板交给主管了一个任务，让主管搞完之后立即上报给老板。主管下面有10个小弟，接到任务之后将任务划分为10个小任务分给每个小弟去干，主管一直处于等待状态（主管会调用 `await()` 方法，此方法会阻塞当前线程），让每个小弟干完之后通知一下主管（调用 `countDown()` 方法通知主管，此方法会立即返回），主管等到所有的小弟都做完了，会被唤醒，从 `await()` 方法上苏醒，然后将结果反馈给老板。期间主管会等待，会等待所有小弟将结果汇报给自己。

而CyclicBarrier是一批线程让自己等待，等待所有的线程都准备好了，自己才能继续。

好了，上面举了6个例子便于大家熟悉 `cyclicBarrier` 的用法，喜欢的帮忙转发一下，谢谢！

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第18篇：java中的线程池

java高并发系列第18篇文章。

本文主要内容

1. 什么是线程池
2. 线程池实现原理
3. 线程池中常见的各种队列
4. 自定义线程创建的工厂
5. 常见的饱和策略
6. 自定义饱和策略
7. 线程池中两种关闭方法有何不同
8. 扩展线程池

加微信itsoku，发送：1024，获取 10T 高质量计算机学习视频！！

9. 合理地配置线程池
10. 线程池中线程数量的配置

什么是线程池

大家用jdbc操作过数据库应该知道，操作数据库需要和数据库建立连接，拿到连接之后才能操作数据库，用完之后销毁。数据库连接的创建和销毁其实是比较耗时的，真正和业务相关的操作耗时是比较短的。每个数据库操作之前都需要创建连接，为了提升系统性能，后来出现了数据库连接池，系统启动的时候，先创建很多连接放在池子里面，使用的时候，直接从连接池中获取一个，使用完毕之后返回到池子里面，继续给其他需要者使用，这其中就省去创建连接的时间，从而提升了系统整体的性能。

线程池和数据库连接池的原理也差不多，创建线程去处理业务，可能创建线程的时间比处理业务的时间还长一些，如果系统能够提前为我们创建好线程，我们需要的时候直接拿来使用，用完之后不是直接将其关闭，而是将其返回到线程集中，给其他需要这使用，这样直接节省了创建和销毁的时间，提升了系统的性能。

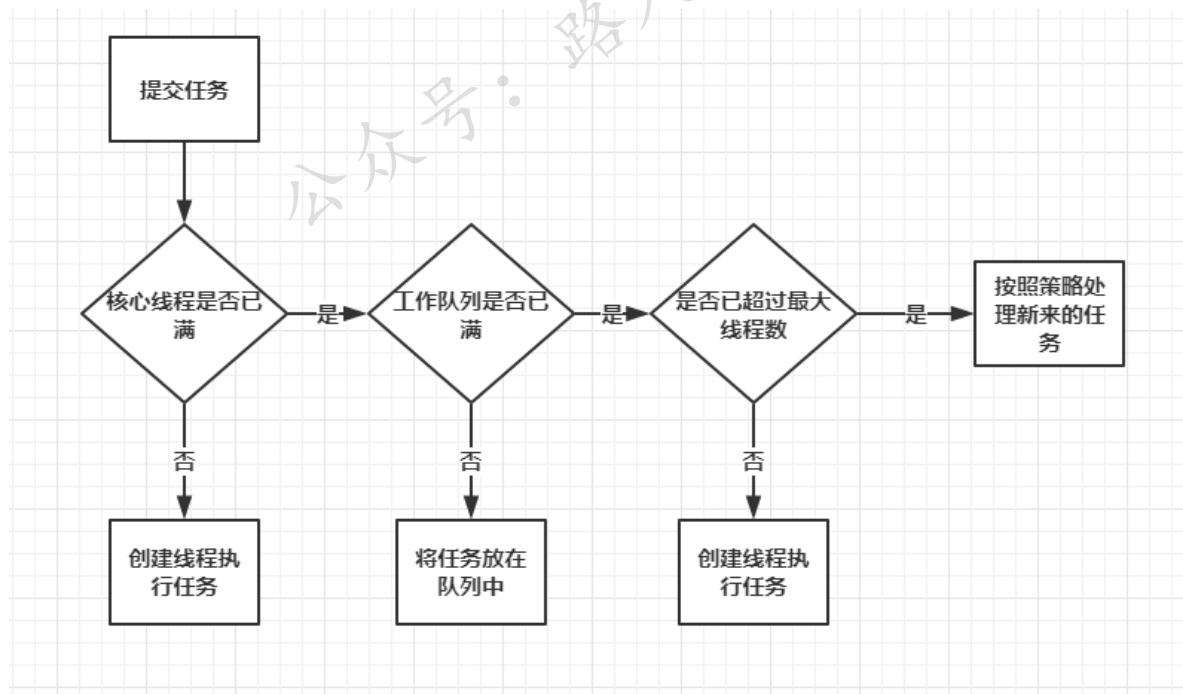
简单的说，在使用了线程池之后，创建线程变成了从线程池中获取一个空闲的线程，然后使用，关闭线程变成了将线程归还到线程池。

线程池实现原理

当向线程池提交一个任务之后，线程池的处理流程如下：

1. 判断是否达到核心线程数，若未达到，则直接创建新的线程处理当前传入的任务，否则进入下个流程
2. 线程池中的工作队列是否已满，若未满，则将任务丢入工作队列中先存着等待处理，否则进入下个流程
3. 是否达到最大线程数，若未达到，则创建新的线程处理当前传入的任务，否则交给线程池中的饱和策略进行处理。

流程如下图：



举个例子，加深理解：

咱们作为开发者，上面都有开发主管，主管下面带领几个小弟干活，CTO给主管授权说，你可以招聘5个小弟干活，新来任务，如果小弟还不到吴哥，立即去招聘一个来干这个新来的任务，当5个小弟都招来了，再来任务之后，将任务记录到一个表格中，表格中最多记录100个，小弟们会主动去表格中获取任务执行，如果5个小弟都在干活，并且表格中也记录满了，那你可以将小弟扩充到20个，如果20个小弟都在干活，并且存放任务的表也满了，产品经理再来任务后，是直接拒绝，还是让产品自己干，这个由你自己决定，小弟们都尽心尽力在干活，任务都被处理完了，突然公司业绩下滑，几个员工没事干，打酱油，为了节约成本，CTO主管把小弟控制到5人，其他15个人直接被干掉了。所以作为小弟们，别让自己闲着，多干活。

原理：先找几个人干活，大家都忙于干活，任务太多可以排期，排期的任务太多了，再招一些人来干活，最后干活的和排期都达到上层领导要求的上限了，那需要采取一些其他策略进行处理了。对于长时间不干活的人，考虑将其开掉，节约资源和成本。

java中的线程池

jdk中提供了线程池的具体实现，实现类是：`java.util.concurrent.ThreadPoolExecutor`，主要构造方法：

```
public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory,
                           RejectedExecutionHandler handler)
```

corePoolSize：核心线程大小，当提交一个任务到线程池时，线程池会创建一个线程来执行任务，即使有其他空闲线程可以处理任务也会创新线程，等到工作的线程数大于核心线程数时就不会再创建了。如果调用了线程池的 `prestartAllCoreThreads` 方法，线程池会提前把核心线程都创造好，并启动

maximumPoolSize：线程池允许创建的最大线程数。如果队列满了，并且以创建的线程数小于最大线程数，则线程池会再创建新的线程执行任务。如果我们使用了无界队列，那么所有的任务会加入队列，这个参数就没有什么效果了

keepAliveTime：线程池的工作线程空闲后，保持存活的时间。如果没有任务处理了，有些线程会空闲，空闲的时间超过了这个值，会被回收掉。如果任务很多，并且每个任务的执行时间比较短，避免线程重复创建和回收，可以调大这个时间，提高线程的利用率

unit：`keepAliveTime`的时间单位，可以选择的单位有天、小时、分钟、毫秒、微妙、千分之一毫秒和纳秒。类型是一个枚举 `java.util.concurrent.TimeUnit`，这个枚举也经常使用，有兴趣的可以看一下其源码

workQueue：工作队列，用于缓存待处理任务的阻塞队列，常见的有4种，本文后面有介绍

threadFactory：线程池中创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更有意义的名字

handler：饱和策略，当线程池无法处理新来的任务了，那么需要提供一种策略处理提交的新任务，默认有4种策略，文章后面会提到

调用线程池的`execute`方法处理任务，执行`execute`方法的过程：

1. 判断线程池中运行的线程数是否小于corePoolSize，是：则创建新的线程来处理任务，否：执行下一步
2. 试图将任务添加到workQueue指定的队列中，如果无法添加到队列，进入下一步
3. 判断线程池中运行的线程数是否小于maximumPoolSize，是：则新增线程处理当前传入的任务，否：将任务传递给 handler 对象 rejectedExecution 方法处理

线程池的使用步骤：

1. 调用构造方法创建线程池
2. 调用线程池的方法处理任务
3. 关闭线程池

线程池使用的简单示例

上一个简单的示例，如下：

```
package com.itsoku.chat16;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    static ThreadPoolExecutor executor = new ThreadPoolExecutor(3,
        5,
        10,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(10),
        Executors.defaultThreadFactory(),
        new ThreadPoolExecutor.AbortPolicy());

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            int j = i;
            String taskName = "任务" + j;
            executor.execute(() -> {
                //模拟任务内部处理耗时
                try {
                    TimeUnit.SECONDS.sleep(j);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + taskName +
"处理完毕");
            });
        }
        //关闭线程池
        executor.shutdown();
    }
}
```

```
}
```

输出：

```
pool-1-thread-1任务0处理完毕  
pool-1-thread-2任务1处理完毕  
pool-1-thread-3任务2处理完毕  
pool-1-thread-1任务3处理完毕  
pool-1-thread-2任务4处理完毕  
pool-1-thread-3任务5处理完毕  
pool-1-thread-1任务6处理完毕  
pool-1-thread-2任务7处理完毕  
pool-1-thread-3任务8处理完毕  
pool-1-thread-1任务9处理完毕
```

线程池中常见5种工作队列

任务太多的时候，工作队列用于暂时缓存待处理的任务，jdk中常见的5种阻塞队列：

ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按照先进先出原则对元素进行排序

LinkedBlockingQueue：是一个基于链表结构的阻塞队列，此队列按照先进先出排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法 `Executors.newFixedThreadPool` 使用了这个队列。

SynchronousQueue：一个不存储元素的阻塞队列，每个插入操作必须等到另外一个线程调用移除操作，否则插入操作一直处理阻塞状态，吞吐量通常要高于LinkedBlockingQueue，静态工厂方法 `Executors.newCachedThreadPool` 使用这个队列

PriorityBlockingQueue：优先级队列，进入队列的元素按照优先级会进行排序

前2种队列相关示例就不说了，主要说一下后面2种队列的使用示例。

SynchronousQueue队列的线程池

```
package com.itsoku.chat16;  
  
import java.util.concurrent.*;  
  
/**  
 * 跟着阿里P7学并发，微信公众号：javacode2018  
 */  
public class Demo2 {  
    public static void main(String[] args) {  
        ExecutorService executor = Executors.newCachedThreadPool();  
        for (int i = 0; i < 50; i++) {  
            int j = i;  
            String taskName = "任务" + j;  
            executor.execute(() -> {  
                System.out.println(Thread.currentThread().getName() + "处理" +  
taskName);  
            });  
        }  
    }  
}
```

```
//模拟任务内部处理耗时
try {
    TimeUnit.SECONDS.sleep(1);
} catch (InterruptedException e) {
    e.printStackTrace();
}
});

executor.shutdown();
}

}
```

```
pool-1-thread-1处理任务0
pool-1-thread-2处理任务1
pool-1-thread-3处理任务2
pool-1-thread-6处理任务5
pool-1-thread-7处理任务6
pool-1-thread-4处理任务3
pool-1-thread-5处理任务4
pool-1-thread-8处理任务7
pool-1-thread-9处理任务8
pool-1-thread-10处理任务9
pool-1-thread-11处理任务10
pool-1-thread-12处理任务11
pool-1-thread-13处理任务12
pool-1-thread-14处理任务13
pool-1-thread-15处理任务14
pool-1-thread-16处理任务15
pool-1-thread-17处理任务16
pool-1-thread-18处理任务17
pool-1-thread-19处理任务18
pool-1-thread-20处理任务19
pool-1-thread-21处理任务20
pool-1-thread-25处理任务24
pool-1-thread-24处理任务23
pool-1-thread-23处理任务22
pool-1-thread-22处理任务21
pool-1-thread-26处理任务25
pool-1-thread-27处理任务26
pool-1-thread-28处理任务27
pool-1-thread-30处理任务29
pool-1-thread-29处理任务28
pool-1-thread-31处理任务30
pool-1-thread-32处理任务31
pool-1-thread-33处理任务32
pool-1-thread-38处理任务37
pool-1-thread-35处理任务34
pool-1-thread-36处理任务35
pool-1-thread-41处理任务40
pool-1-thread-34处理任务33
pool-1-thread-39处理任务38
pool-1-thread-40处理任务39
pool-1-thread-37处理任务36
pool-1-thread-42处理任务41
pool-1-thread-43处理任务42
```

```
pool-1-thread-45处理任务44  
pool-1-thread-46处理任务45  
pool-1-thread-44处理任务43  
pool-1-thread-47处理任务46  
pool-1-thread-50处理任务49  
pool-1-thread-48处理任务47  
pool-1-thread-49处理任务48
```

代码中使用 `Executors.newCachedThreadPool()` 创建线程池，看一下的源码：

```
public static ExecutorService newCachedThreadPool() {  
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,  
        60L, TimeUnit.SECONDS,  
        new SynchronousQueue<Runnable>());  
}
```

从输出中可以看出，系统创建了50个线程处理任务，代码中使用了 `SynchronousQueue` 同步队列，这种队列比较特殊，放入元素必须要有另外一个线程去获取这个元素，否则放入元素会失败或者一直阻塞在那里直到有线程取走，示例中任务处理休眠了指定的时间，导致已创建的工作线程都忙于处理任务，所以新来任务之后，将任务丢入同步队列会失败，丢入队列失败之后，会尝试新建线程处理任务。使用上面的方式创建线程池需要注意，如果需要处理的任务比较耗时，会导致新来的任务都会创建新的线程进行处理，可能会导致创建非常多的线程，最终耗尽系统资源，触发OOM。

PriorityBlockingQueue优先级队列的线程池

```
package com.itsoku.chat16;  
  
import java.util.concurrent.*;  
  
/**  
 * 跟着阿里P7学并发，微信公众号：javacode2018  
 */  
public class Demo3 {  
  
    static class Task implements Runnable, Comparable<Task> {  
  
        private int i;  
        private String name;  
  
        public Task(int i, String name) {  
            this.i = i;  
            this.name = name;  
        }  
  
        @Override  
        public void run() {  
            System.out.println(Thread.currentThread().getName() + "处理" +  
this.name);  
        }  
  
        @Override  
        public int compareTo(Task o) {  
            return Integer.compare(o.i, this.i);  
        }  
    }  
}
```

```

    }

    public static void main(String[] args) {
        ExecutorService executor = new ThreadPoolExecutor(1, 1,
            60L, TimeUnit.SECONDS,
            new PriorityBlockingQueue());
        for (int i = 0; i < 10; i++) {
            String taskName = "任务" + i;
            executor.execute(new Task(i, taskName));
        }
        for (int i = 100; i >= 90; i--) {
            String taskName = "任务" + i;
            executor.execute(new Task(i, taskName));
        }
        executor.shutdown();
    }
}

```

输出：

```

pool-1-thread-1处理任务0
pool-1-thread-1处理任务100
pool-1-thread-1处理任务99
pool-1-thread-1处理任务98
pool-1-thread-1处理任务97
pool-1-thread-1处理任务96
pool-1-thread-1处理任务95
pool-1-thread-1处理任务94
pool-1-thread-1处理任务93
pool-1-thread-1处理任务92
pool-1-thread-1处理任务91
pool-1-thread-1处理任务90
pool-1-thread-1处理任务9
pool-1-thread-1处理任务8
pool-1-thread-1处理任务7
pool-1-thread-1处理任务6
pool-1-thread-1处理任务5
pool-1-thread-1处理任务4
pool-1-thread-1处理任务3
pool-1-thread-1处理任务2
pool-1-thread-1处理任务1

```

输出中，除了第一个任务，其他任务按照优先级高低按顺序处理。原因在于：创建线程池的时候使用了优先级队列，进入队列中的任务会进行排序，任务的先后顺序由Task中的i变量决定。向 `PriorityBlockingQueue` 加入元素的时候，内部会调用代码中Task的 `compareTo` 方法决定元素的先后顺序。

自定义创建线程的工厂

给线程池中线程起一个有意义的名字，在系统出现问题的时候，通过线程堆栈信息可以更容易发现系统中问题所在。自定义创建工厂需要实现 `java.util.concurrent.ThreadFactory` 接口中的 `Thread newThread(Runnable r)` 方法，参数为传入的任务，需要返回一个工作线程。

示例代码：

```
package com.itsoku.chat16;

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {
    static AtomicInteger threadNum = new AtomicInteger(1);

    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 5,
            60L, TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(10), r -> {
                Thread thread = new Thread(r);
                thread.setName("自定义线程-" + threadNum.getAndIncrement());
                return thread;
            });
        for (int i = 0; i < 5; i++) {
            String taskName = "任务-" + i;
            executor.execute(() -> {
                System.out.println(Thread.currentThread().getName() + "处理" +
taskName);
            });
        }
        executor.shutdown();
    }
}
```

输出：

```
自定义线程-1处理任务-0
自定义线程-3处理任务-2
自定义线程-2处理任务-1
自定义线程-4处理任务-3
自定义线程-5处理任务-4
```

代码中在任务中输出了当前线程的名称，可以看到是我们自定义的名称。

通过jstack查看线程的堆栈信息，也可以看到我们自定义的名称，我们可以将代码中
`executor.shutdown();`先给注释掉让程序先不退出，然后通过jstack查看，如下：

```

"自定义线程-4" #15 prio=5 os_prio=0 tid=0x0000000028b19800 nid=0x883c waiting on condition [0x000000002960e0
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x0000000717391160> (a java.util.concurrent.locks.AbstractQueuedSynchronizer
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:201)
at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:403)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)

"自定义线程-3" #14 prio=5 os_prio=0 tid=0x0000000028b15000 nid=0x6788 waiting on condition [0x000000002950e0
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x0000000717391160> (a java.util.concurrent.locks.AbstractQueuedSynchronizer
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:201)
at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:403)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)

"自定义线程-2" #13 prio=5 os_prio=0 tid=0x0000000028b14000 nid=0x73cc waiting on condition [0x000000002940f0
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x0000000717391160> (a java.util.concurrent.locks.AbstractQueuedSynchronizer
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:201)
at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:403)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1127)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617)
at java.lang.Thread.run(Thread.java:745)

"自定义线程-1" #12 prio=5 os_prio=0 tid=0x0000000028b12800 nid=0x57c4 waiting on condition [0x000000002930f0
java.lang.Thread.State: WAITING (parking)
    at sun.misc.Unsafe.park(Native Method)
    - parking to wait for <0x0000000717391160> (a java.util.concurrent.locks.AbstractQueuedSynchronizer
at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(AbstractQueuedSynchronizer.java:201)
at java.util.concurrent.ArrayBlockingQueue.take(ArrayBlockingQueue.java:403)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1067)

```

4种常见饱和策略

当线程池中队列已满，并且线程池已达到最大线程数，线程池会将任务传递给饱和策略进行处理。这些策略都实现了 `RejectedExecutionHandler` 接口。接口中有个方法：

```
void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
```

参数说明：

`r`: 需要执行的任务

`executor`: 当前线程池对象

JDK中提供了4种常见的饱和策略：

AbortPolicy: 直接抛出异常

CallerRunsPolicy: 在当前调用者的线程中运行任务，即随丢来的任务，由他自己去处理

DiscardOldestPolicy: 丢弃队列中最老的一个任务，即丢弃队列头部的一个任务，然后执行当前传入的任务

DiscardPolicy: 不处理，直接丢弃掉，方法内部为空

自定义饱和策略

需要实现 `RejectedExecutionHandler` 接口。任务无法处理的时候，我们想记录一下日志，我们需要自定义一个饱和策略，示例代码：

```
package com.itsoku.chat16;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo5 {
    static class Task implements Runnable {
        String name;

        public Task(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + "处理" +
this.name);
            try {
                TimeUnit.SECONDS.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        @Override
        public String toString() {
            return "Task{" +
                "name='" + name + '\'' +
                '}';
        }
    }

    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(1,
            1,
            60L,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<Runnable>(1),
            Executors.defaultThreadFactory(),
            (r, executors) -> {
                //自定义饱和策略
                //记录一下无法处理的任务
                System.out.println("无法处理的任务：" + r.toString());
            });
        for (int i = 0; i < 5; i++) {
            executor.execute(new Task("任务-" + i));
        }
        executor.shutdown();
    }
}
```

```
    }  
}
```

输出：

```
无法处理的任务: Task{name='任务-2'}  
无法处理的任务: Task{name='任务-3'}  
pool-1-thread-1处理任务-0  
无法处理的任务: Task{name='任务-4'}  
pool-1-thread-1处理任务-1
```

输出结果中可以看到有3个任务进入了饱和策略中，记录了任务的日志，对于无法处理多任务，我们最好能够记录一下，让开发人员能够知道。任务进入了饱和策略，说明线程池的配置可能不是太合理，或者机器的性能有限，需要做一些优化调整。

线程池中的2个关闭方法

线程池提供了2个关闭方法：`shutdown` 和 `shutdownNow`，当调用者两个方法之后，线程池会遍历内部的工作线程，然后调用每个工作线程的`interrupt`方法给线程发送中断信号，内部如果无法响应中断信号的可能永远无法终止，所以如果内部有无线循环的，最好在循环内部检测一下线程的中断信号，合理的退出。调用者两个方法中任意一个，线程池的 `isShutdown` 方法就会返回true，当所有的任务线程都关闭之后，才表示线程池关闭成功，这时调用 `isTerminated` 方法会返回true。

调用 `shutdown` 方法之后，线程池将不再接受新任务，内部会将所有已提交的任务处理完毕，处理完毕之后，工作线程自动退出。

而调用 `shutdownNow` 方法后，线程池会将还未处理的（在队列里等待处理的任务）任务移除，将正在处理中的任务处理完毕之后，工作线程自动退出。

至于调用哪个方法来关闭线程，应该由提交到线程池的任务特性决定，多数情况下调用 `shutdown` 方法来关闭线程池，如果任务不一定要执行完，则可以调用 `shutdownNow` 方法。

扩展线程池

虽然jdk提供了`ThreadPoolExecutor`这个高性能线程池，但是如果我们自己想在这个线程池上面做一些扩展，比如，监控每个任务执行的开始时间，结束时间，或者一些其他自定义的功能，我们应该怎么办？

这个jdk已经帮我们想到了，`ThreadPoolExecutor`内部提供了几个方法 `beforeExecute`、`afterExecute`、`terminated`，可以由开发人员自己去实现这些方法。看一下线程池内部的源码：

```
try {  
    beforeExecute(wt, task); //任务执行之前调用的方法  
    Throwable thrown = null;  
    try {  
        task.run();  
    } catch (RuntimeException x) {  
        thrown = x;  
        throw x;  
    } catch (Error x) {  
        thrown = x;
```

```

        throw x;
    } catch (Throwable x) {
        thrown = x;
        throw new Error(x);
    } finally {
        afterExecute(task, thrown); //任务执行完毕之后调用的方法
    }
} finally {
    task = null;
    w.completedTasks++;
    w.unlock();
}

```

beforeExecute: 任务执行之前调用的方法，有2个参数，第1个参数是执行任务的线程，第2个参数是任务

```
protected void beforeExecute(Thread t, Runnable r) { }
```

afterExecute: 任务执行完成之后调用的方法，2个参数，第1个参数表示任务，第2个参数表示任务执行时的异常信息，如果无异常，第二个参数为null

```
protected void afterExecute(Runnable r, Throwable t) { }
```

terminated: 线程池最终关闭之后调用的方法。所有的工作线程都退出了，最终线程池会退出，退出时调用该方法

示例代码：

```

package com.itsoku.chat16;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo6 {
    static class Task implements Runnable {
        String name;

        public Task(String name) {
            this.name = name;
        }

        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + "处理" +
this.name);
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

@Override
public String toString() {
    return "Task{" +
        "name='" + name + '\'' +
        '}';
}

public static void main(String[] args) throws InterruptedException {
    ThreadPoolExecutor executor = new ThreadPoolExecutor(10,
        10,
        60L,
        TimeUnit.SECONDS,
        new ArrayBlockingQueue<Runnable>(1),
        Executors.defaultThreadFactory(),
        (r, executors) -> {
            //自定义饱和策略
            //记录一下无法处理的任务
            System.out.println("无法处理的任务: " + r.toString());
        })
    }

@Override
protected void beforeExecute(Thread t, Runnable r) {
    System.out.println(System.currentTimeMillis() + "," +
t.getName() + ",开始执行任务:" + r.toString());
}

@Override
protected void afterExecute(Runnable r, Throwable t) {
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",任务:" + r.toString() + ", 执行完毕!");
}

@Override
protected void terminated() {
    System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ", 关闭线程池!");
};

for (int i = 0; i < 10; i++) {
    executor.execute(new Task("任务-" + i));
}
TimeUnit.SECONDS.sleep(1);
executor.shutdown();
}
}

```

输出：

```

1564324574847, pool-1-thread-1, 开始执行任务:Task{name='任务-0'}
1564324574850, pool-1-thread-3, 开始执行任务:Task{name='任务-2'}
pool-1-thread-3处理任务-2
1564324574849, pool-1-thread-2, 开始执行任务:Task{name='任务-1'}
pool-1-thread-2处理任务-1
1564324574848, pool-1-thread-5, 开始执行任务:Task{name='任务-4'}
pool-1-thread-5处理任务-4
1564324574848, pool-1-thread-4, 开始执行任务:Task{name='任务-3'}

```

```
pool-1-thread-4处理任务-3
1564324574850, pool-1-thread-7, 开始执行任务:Task{name='任务-6'}
pool-1-thread-7处理任务-6
1564324574850, pool-1-thread-6, 开始执行任务:Task{name='任务-5'}
1564324574851, pool-1-thread-8, 开始执行任务:Task{name='任务-7'}
pool-1-thread-8处理任务-7
pool-1-thread-1处理任务-0
pool-1-thread-6处理任务-5
1564324574851, pool-1-thread-10, 开始执行任务:Task{name='任务-9'}
pool-1-thread-10处理任务-9
1564324574852, pool-1-thread-9, 开始执行任务:Task{name='任务-8'}
pool-1-thread-9处理任务-8
1564324576851, pool-1-thread-2, 任务:Task{name='任务-1'}, 执行完毕!
1564324576851, pool-1-thread-3, 任务:Task{name='任务-2'}, 执行完毕!
1564324576852, pool-1-thread-1, 任务:Task{name='任务-0'}, 执行完毕!
1564324576852, pool-1-thread-4, 任务:Task{name='任务-3'}, 执行完毕!
1564324576852, pool-1-thread-8, 任务:Task{name='任务-7'}, 执行完毕!
1564324576852, pool-1-thread-7, 任务:Task{name='任务-6'}, 执行完毕!
1564324576852, pool-1-thread-5, 任务:Task{name='任务-4'}, 执行完毕!
1564324576853, pool-1-thread-6, 任务:Task{name='任务-5'}, 执行完毕!
1564324576853, pool-1-thread-10, 任务:Task{name='任务-9'}, 执行完毕!
1564324576853, pool-1-thread-9, 任务:Task{name='任务-8'}, 执行完毕!
1564324576853, pool-1-thread-9, 关闭线程池!
```

从输出结果中可以看到，每个需要执行的任务打印了3行日志，执行前由线程池的 `beforeExecute` 打印，执行时会调用任务的 `run` 方法，任务执行完毕之后，会调用线程池的 `afterExecute` 方法，从每个任务的首尾2条日志中可以看到每个任务耗时2秒左右。线程池最终关闭之后调用了 `terminated` 方法。

合理地配置线程池

要想合理的配置线程池，需要先分析任务的特性，可以冲以下几个角度分析：

- 任务的性质：CPU密集型任务、IO密集型任务和混合型任务
- 任务的优先级：高、中、低
- 任务的执行时间：长、中、短
- 任务的依赖性：是否依赖其他的系统资源，如数据库连接。

性质不同任务可以用不同规模的线程池分开处理。CPU密集型任务应该尽可能小的线程，如配置cpu数量+1个线程的线程池。由于IO密集型任务并不是一直在执行任务，不能让cpu闲着，则应配置尽可能多的线程，如：cpu数量*2。混合型的任务，如果可以拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这2个任务执行的时间相差不是太大，那么分解后执行的吞吐量将高于串行执行的吞吐量。可以通过 `Runtime.getRuntime().availableProcessors()` 方法获取cpu数量。优先级不同任务可以对线程池采用优先级队列来处理，让优先级高的先执行。

使用队列的时候建议使用有界队列，有界队列增加了系统的稳定性，如果采用无解队列，任务太多的时候可能导致系统OOM，直接让系统宕机。

线程池中线程数量的配置

线程池汇总线程大小对系统的性能有一定的影响，我们的目标是希望系统能够发挥最好的性能，过多或者过少的线程数量无法有效利用机器的性能。在Java Concurrency in Practice书中给出了估算线程池大小的公式：

Ncpu = CPU的数量

Ucpu = 目标CPU的使用率, $0 \leq Ucpu \leq 1$

W/C = 等待时间与计算时间的比例

为保证处理器达到期望的使用率, 最有的线程池的大小等于:

Nthreads = $Ncpu \times Ucpu \times (1+W/C)$

一些使用建议

在《阿里巴巴java开发手册》中指出了线程资源必须通过线程池提供, 不允许在应用中自行显示的创建线程, 这样一方面是线程的创建更加规范, 可以合理控制开辟线程的数量; 另一方面线程的细节管理交给线程池处理, 优化了资源的开销。而线程池不允许使用Executors去创建, 而要通过 ThreadPoolExecutor方式, 这一方面是由jdk中Executor框架虽然提供了如newFixedThreadPool()、newSingleThreadExecutor()、newCachedThreadPool()等创建线程池的方法, 但都有其局限性, 不够灵活; 另外由于前面几种方法内部也是通过ThreadPoolExecutor方式实现, 使用ThreadPoolExecutor有助于大家明确线程池的运行规则, 创建符合自己的业务场景需要的线程池, 避免资源耗尽的风险。

ThreadPoolTaskExecutor 其他知识点汇总(待补充)

1. 线程池中的所有线程超过了空闲时间都会被销毁么?

如果allowCoreThreadTimeOut为true, 超过了空闲时间的所有线程都会被回收, 不过这个值默认是false, 系统会保留核心线程, 其他的会被回收

2. 空闲线程是如何被销毁的?

所有运行的工作线程会尝试从队列中获取任务去执行, 超过一定时间(keepAliveTime)还没有拿到任务, 自己主动退出

3. 核心线程在线程池创建的时候会初始化好么?

默认情况下, 核心线程不会进行初始化, 在刚开始调用线程池执行任务的时候, 传入一个任务会创建一个线程, 直到达到核心线程数。不过可以在创建线程池之后, 调用其

```
restartAllCoreThreads
```

提前将核心线程创建好。

更多好文章

1. [spring高手系列 \(正在连载中\)](#)
2. [Java高并发系列 \(共34篇\)](#)
3. [MySQL高手系列 \(共27篇\)](#)
4. [Maven高手系列 \(共10篇\)](#)
5. [Mybatis系列 \(共12篇\)](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要, 它是什么? 怎么实现?](#)
8. [泛型, 有点难度, 会让很多人懵逼, 那是因为你没有看这篇文章!](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第19篇：JUC中的Executor框架详解1

这是java高并发系列第19篇文章。

本文主要内容

1. 介绍Executor框架相关内容
2. 介绍Executor
3. 介绍ExecutorService
4. 介绍线程池ThreadPoolExecutor及案例
5. 介绍定时器ScheduledExecutorService及案例
6. 介绍Executors类的使用
7. 介绍Future接口
8. 介绍Callable接口
9. 介绍FutureTask的使用
10. 获取异步任务的执行结果的几种方法

Executors框架介绍

Executors框架是Doug Lea的神作，通过这个框架，可以很容易的使用线程池高效地处理并行任务。

Executor框架主要包含3部分的内容：

1. 任务相关的：包含被执行的任务要实现的接口：**Runnable**接口或**Callable**接口
2. 任务的执行相关的：包含任务执行机制的**核心接口Executor**，以及继承自 Executor 的 **ExecutorService** 接口。Executor框架中有两个关键的类实现了ExecutorService接口（`ThreadPoolExecutor` 和 `ScheduledThreadPoolExecutor`）
3. 异步计算结果相关的：包含**接口Future**和**实现Future接口的FutureTask类**

Executors框架包括：

- Executor
- ExecutorService
- ThreadPoolExecutor
- Executors
- Future
- Callable

- FutureTask
- CompletableFuture
- CompletionService
- ExecutorCompletionService

下面我们来一个个介绍其用途和使用方法。

Executor接口

Executor接口中定义了方法execute(Runnable able)接口，该方法接受一个Runnable实例，他来执行一个任务，任务即实现一个Runnable接口的类。

ExecutorService接口

ExecutorService继承于Executor接口，他提供了更为丰富的线程实现方法，比如ExecutorService提供关闭自己的方法，以及为跟踪一个或多个异步任务执行状况而生成Future的方法。

ExecutorService有三种状态：运行、关闭、终止。创建后便进入运行状态，当调用了shutdown()方法时，便进入了关闭状态，此时意味着ExecutorService不再接受新的任务，但是他还是会执行已经提交的任务，当所有已经提交了的任务执行完后，便达到终止状态。如果不调用shutdown方法，ExecutorService方法会一直运行下去，系统一般不会主动关闭。

ThreadPoolExecutor类

线程池类，实现了ExecutorService接口中所有方法，该类也是我们经常要用到的，非常重要，关于此类有详细的介绍，可以移步：[玩转java中的线程池](#)

ScheduleThreadPoolExecutor定时器

ScheduleThreadPoolExecutor继承自ScheduleThreadPoolExecutor，他主要用来延迟执行任务，或者定时执行任务。功能和Timer类似，但是ScheduleThreadPoolExecutor更强大、更灵活一些。Timer后台是单个线程，而ScheduleThreadPoolExecutor可以在创建的时候指定多个线程。

常用方法介绍：

schedule:延迟执行任务1次

使用ScheduleThreadPoolExecutor的schedule方法，看一下这个方法的声明：

```
public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)
```

3个参数：

command：需要执行的任务

delay：需要延迟的时间

unit：参数2的时间单位，是个枚举，可以是天、小时、分钟、秒、毫秒、纳秒等

示例代码:

```
package com.itsoku.chat18;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        System.out.println(System.currentTimeMillis());
        ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(10);
        scheduledExecutorService.schedule(() -> {
            System.out.println(System.currentTimeMillis() + "开始执行");
            //模拟任务耗时
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "执行结束");
        }, 2, TimeUnit.SECONDS);
    }
}
```

输出：

```
1564575180457
1564575185525开始执行
1564575188530执行结束
```

scheduleAtFixedRate:固定的频率执行任务

使用 `ScheduledExecutorService` 的 `scheduleAtFixedRate` 方法，该方法设置了执行周期，下一次执行时间相当于是上一次的执行时间加上 `period`，任务每次执行完毕之后才会计算下次的执行时间。

看一下这个方法的声明：

```
public ScheduledFuture<?> scheduleAtFixedRate(Runnable command,
                                                long initialDelay,
                                                long period,
                                                TimeUnit unit);
```

4个参数：

command：表示要执行的任务

initialDelay：表示延迟多久执行第一次

period: 连续执行之间的时间间隔

unit: 参数2和参数3的时间单位, 是个枚举, 可以是天、小时、分钟、秒、毫秒、纳秒等

假设系统调用scheduleAtFixedRate的时间是T1, 那么执行时间如下:

第1次: T1+initialDelay

第2次: T1+initialDelay+period

第3次: T1+initialDelay+2*period

第n次: T1+initialDelay+(n-1)*period

示例代码:

```
package com.itsoku.chat18;

import java.sql.Time;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发, 微信公众号: javacode2018
 */
public class Demo2 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        System.out.println(System.currentTimeMillis());
        //任务执行计数器
        AtomicInteger count = new AtomicInteger(1);
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(10);
        scheduledExecutorService.scheduleAtFixedRate(() -> {
            int currCount = count.getAndIncrement();
            System.out.println(Thread.currentThread().getName());
            System.out.println(System.currentTimeMillis() + "第" + currCount +
"次" + "开始执行");
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "第" + currCount +
"次" + "执行结束");
        }, 1, 1, TimeUnit.SECONDS);
    }
}
```

前面6次输出结果:

```
1564576404181
pool-1-thread-1
1564576405247第1次开始执行
1564576407251第1次执行结束
pool-1-thread-1
1564576407251第2次开始执行
1564576409252第2次执行结束
pool-1-thread-2
```

```
1564576409252第3次开始执行  
1564576411255第3次执行结束  
pool-1-thread-1  
1564576411256第4次开始执行  
1564576413260第4次执行结束  
pool-1-thread-3  
1564576413260第5次开始执行  
1564576415265第5次执行结束  
pool-1-thread-2  
1564576415266第6次开始执行  
1564576417269第6次执行结束
```

代码中设置的任务第一次执行时间是系统启动之后延迟一秒执行。后面每次时间间隔1秒，从输出中可以看出系统启动之后过了1秒任务第一次执行（1、3行输出），输出的结果中可以看到任务第一次执行结束时间和第二次的结束时间一样，为什么会这样？前面有介绍，任务当前执行完毕之后会计算下次执行时间，下次执行时间为上次执行的开始时间+period，第一次开始执行时间是1564576405247，加1秒为1564576406247，这个时间小于第一次结束的时间了，说明小于系统当前时间了，会立即执行。

scheduleWithFixedDelay:固定的间隔执行任务

使用 `ScheduleThreadpoolExecutor` 的 `scheduleWithFixedDelay` 方法，该方法设置了执行周期，与 `scheduleAtFixedRate` 方法不同的是，下一次执行时间是上一次任务执行完的系统时间加上 `period`，因而具体执行时间不是固定的，但周期是固定的，是采用相对固定的延迟来执行任务。看一下这个方法的声明：

```
public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command,  
                                                long initialDelay,  
                                                long delay,  
                                                TimeUnit unit);
```

4个参数：

`command`: 表示要执行的任务

`initialDelay`: 表示延迟多久执行第一次

`period`: 表示下次执行时间和上次执行结束时间之间的间隔时间

`unit`: 参数2和参数3的时间单位，是个枚举，可以是天、小时、分钟、秒、毫秒、纳秒等

假设系统调用 `scheduleAtFixedRate` 的时间是 `T1`，那么执行时间如下：

第1次: $T1 + initialDelay$, 执行结束时间: $E1$

第2次: $E1 + period$, 执行结束时间: $E2$

第3次: $E2 + period$, 执行结束时间: $E3$

第4次: $E3 + period$, 执行结束时间: $E4$

第 n 次: 上次执行结束时间+`period`

示例代码：

```
package com.itsoku.chat18;  
  
import java.util.concurrent.*;  
import java.util.concurrent.atomic.AtomicInteger;
```

```

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo3 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        System.out.println(System.currentTimeMillis());
        //任务执行计数器
        AtomicInteger count = new AtomicInteger(1);
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(10);
        scheduledExecutorService.scheduleWithFixedDelay(() -> {
            int currCount = count.getAndIncrement();
            System.out.println(Thread.currentThread().getName());
            System.out.println(System.currentTimeMillis() + "第" + currCount +
"次" + "开始执行");
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "第" + currCount +
"次" + "执行结束");
        }, 1, 3, TimeUnit.SECONDS);
    }
}

```

前几次输出如下：

```

1564578510983
pool-1-thread-1
1564578512087第1次开始执行
1564578514091第1次执行结束
pool-1-thread-1
1564578517096第2次开始执行
1564578519100第2次执行结束
pool-1-thread-2
1564578522103第3次开始执行
1564578524105第3次执行结束
pool-1-thread-1
1564578527106第4次开始执行
1564578529106第4次执行结束

```

延迟1秒之后执行第1次，后面每次的执行时间和上次执行结束时间间隔3秒。

`scheduleAtFixedRate` 和 `scheduleWithFixedDelay` 示例建议多看2遍。

定时任务有异常会怎么样？

示例代码：

```

package com.itsoku.chat18;

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

```

```

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo4 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        System.out.println(System.currentTimeMillis());
        //任务执行计数器
        AtomicInteger count = new AtomicInteger(1);
        ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(10);
        ScheduledFuture<?> scheduledFuture =
        scheduledExecutorService.scheduleWithFixedDelay(() -> {
            int currCount = count.getAndIncrement();
            System.out.println(Thread.currentThread().getName());
            System.out.println(System.currentTimeMillis() + "第" + currCount +
            "次" + "开始执行");
            System.out.println(10 / 0);
            System.out.println(System.currentTimeMillis() + "第" + currCount +
            "次" + "执行结束");
        }, 1, 1, TimeUnit.SECONDS);

        TimeUnit.SECONDS.sleep(5);
        System.out.println(scheduledFuture.isCancelled());
        System.out.println(scheduledFuture.isDone());

    }
}

```

系统输出如下内容就再也没有输出了：

```

1564578848143
pool-1-thread-1
1564578849226第1次开始执行
false
true

```

先说补充点知识：schedule、scheduleAtFixedRate、scheduleWithFixedDelay这几个方法有个返回值 ScheduledFuture，通过 ScheduledFuture 可以对执行的任务做一些操作，如判断任务是否被取消、是否执行完成。

再回到上面代码，任务中有个10/0的操作，会触发异常，发生异常之后没有任何现象，被 ScheduledExecutorService 内部给吞掉了，然后这个任务再也不会执行了，
scheduledFuture.isDone() 输出 true，表示这个任务已经结束了，再也不会被执行了。所以如果程序有异常，开发者自己注意处理一下，不然跑着跑着发现任务怎么不跑了，也没有异常输出。

取消定时任务的执行

可能任务执行一会，想取消执行，可以调用 ScheduledFuture 的 cancel 方法，参数表示是否给任务发送中断信号。

```
package com.itsoku.chat18;
```

```

import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo5 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        System.out.println(System.currentTimeMillis());
        //任务执行计数器
        AtomicInteger count = new AtomicInteger(1);
        ScheduledExecutorService scheduledExecutorService =
Executors.newScheduledThreadPool(1);
        ScheduledFuture<?> scheduledFuture =
scheduledExecutorService.scheduleWithFixedDelay(() -> {
            int currCount = count.getAndIncrement();
            System.out.println(Thread.currentThread().getName());
            System.out.println(System.currentTimeMillis() + "第" + currCount +
"次" + "开始执行");
            try {
                TimeUnit.SECONDS.sleep(2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(System.currentTimeMillis() + "第" + currCount +
"次" + "执行结束");
        }, 1, 1, TimeUnit.SECONDS);

        TimeUnit.SECONDS.sleep(5);
        scheduledFuture.cancel(false);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("任务是否被取消：" + scheduledFuture.isCancelled());
        System.out.println("任务是否已完成：" + scheduledFuture.isDone());
    }
}

```

输出：

```

1564579843190
pool-1-thread-1
1564579844255第1次开始执行
1564579846260第1次执行结束
pool-1-thread-1
1564579847263第2次开始执行
任务是否被取消：true
任务是否已完成：true
1564579849267第2次执行结束

```

输出中可以看到任务被取消成功了。

Executors类

Executors类，提供了一系列工厂方法用于创建线程池，返回的线程池都实现了ExecutorService接口。常用的方法有：

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor()
public static ExecutorService newSingleThreadExecutor(ThreadFactory
threadFactory)
```

创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。内部使用了无限容量的LinkedBlockingQueue阻塞队列来缓存任务，任务如果比较多，单线程如果处理不过来，会导致队列堆满，引发OOM。

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads)
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory)
```

创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，在提交新任务，任务将会进入等待队列中等待。如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。内部使用了无限容量的LinkedBlockingQueue阻塞队列来缓存任务，任务如果比较多，如果处理不过来，会导致队列堆满，引发OOM。

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool()
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)
```

创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒处于等待任务到来）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池的最大值是Integer的最大值(2^31-1)。内部使用了SynchronousQueue同步队列来缓存任务，此队列的特性是放入任务时必须要有对应的线程获取任务，任务才可以放入成功。如果处理的任务比较耗时，任务来的速度也比较快，会创建太多的线程引发OOM。

newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize,
ThreadFactory threadFactory)
```

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

在《阿里巴巴java开发手册》中指出了线程资源必须通过线程池提供，不允许在应用中自行显示的创建线程，这样一方面是线程的创建更加规范，可以合理控制开辟线程的数量；另一方面线程的细节管理交给线程池处理，优化了资源的开销。而线程池不允许使用Executors去创建，而要通过ThreadPoolExecutor方式，这一方面是由于jdk中Executor框架虽然提供了如newFixedThreadPool()、newSingleThreadExecutor()、newCachedThreadPool()等创建线程池的方法，但都有其局限性，不够

灵活；另外由于前面几种方法内部也是通过ThreadPoolExecutor方式实现，使用ThreadPoolExecutor有助于大家明确线程池的运行规则，创建符合自己的业务场景需要的线程池，避免资源耗尽的风险。

Future、Callable接口

Future 接口定义了操作异步异步任务执行一些方法，如获取异步任务的执行结果、取消任务的执行、判断任务是否被取消、判断任务执行是否完毕等。

Callable 接口中定义了需要有返回的任务需要实现的方法。

```
@FunctionalInterface  
public interface Callable<V> {  
    V call() throws Exception;  
}
```

比如主线程让一个子线程去执行任务，子线程可能比较耗时，启动子线程开始执行任务后，主线程就去做其他事情了，过了一会才去获取子任务的执行结果。

获取异步任务执行结果

示例代码：

```
package com.itsoku.chat18;  
  
import java.util.concurrent.*;  
  
/**  
 * 跟着阿里P7学并发，微信公众号：javacode2018  
 */  
public class Demo6 {  
    public static void main(String[] args) throws ExecutionException,  
InterruptedException {  
        ExecutorService executorService = Executors.newFixedThreadPool(1);  
        Future<Integer> result = executorService.submit(() -> {  
            System.out.println(System.currentTimeMillis() + "," +  
Thread.currentThread().getName() + ",start!");  
            TimeUnit.SECONDS.sleep(5);  
            System.out.println(System.currentTimeMillis() + "," +  
Thread.currentThread().getName() + ",end!");  
            return 10;  
        });  
        System.out.println(System.currentTimeMillis() + "," +  
Thread.currentThread().getName());  
        System.out.println(System.currentTimeMillis() + "," +  
Thread.currentThread().getName() + ",结果：" + result.get());  
    }  
}
```

输出：

```
1564581941442,main  
1564581941442,pool-1-thread-1,start!  
1564581946447,pool-1-thread-1,end!  
1564581941442,main,结果: 10
```

代码中创建了一个线程池，调用线程池的 submit 方法执行任务，submit参数为 callable 接口：表示需要执行的任务有返回值，submit方法返回一个 Future 对象，Future相当于一个凭证，可以在任意时间拿着这个凭证去获取对应任务的执行结果（调用其 get 方法），代码中调用了 result.get() 方法之后，此方法会阻塞当前线程直到任务执行结束。

超时获取异步任务执行结果

可能任务执行比较耗时，比如耗时1分钟，我最多只能等待10秒，如果10秒还没返回，我就去做其他事情了。

刚好get有个超时的方法，声明如下：

```
V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException;
```

示例代码：

```
package com.itsoku.chat18;

import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo8 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(1);
        Future<Integer> result = executorService.submit(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",start!");
            TimeUnit.SECONDS.sleep(5);
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",end!");
            return 10;
        });
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName());
        try {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",结果：" + result.get(3, TimeUnit.SECONDS));
        } catch (TimeoutException e) {
            e.printStackTrace();
        }
        executorService.shutdown();
    }
}
```

输出：

```
1564583177139,main
1564583177139,pool-1-thread-1,start!
java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
    at com.itsoku.chat18.Demo8.main(Demo8.java:19)
1564583182142,pool-1-thread-1,end!
```

任务执行中休眠了5秒，get方法获取执行结果，超时时间是3秒，3秒还未获取到结果，get触发了TimeoutException异常，当前线程从阻塞状态苏醒了。

Future其他方法介绍一下

cancel: 取消在执行的任务，参数表示是否对执行的任务发送中断信号，方法声明如下：

```
boolean cancel(boolean mayInterruptIfRunning);
```

isCancelled: 用来判断任务是否被取消

isDone: 判断任务是否执行完毕。

cancel方法来个示例:

```
package com.itsoku.chat18;

import java.util.concurrent.*;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo7 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(1);
        Future<Integer> result = executorService.submit(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",start!");
            TimeUnit.SECONDS.sleep(5);
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",end!");
            return 10;
        });

        executorService.shutdown();

        TimeUnit.SECONDS.sleep(1);
        result.cancel(false);
        System.out.println(result.isCancelled());
        System.out.println(result.isDone());

        TimeUnit.SECONDS.sleep(5);
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName());
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",结果：" + result.get());
        executorService.shutdown();
    }
}
```

```
    }  
}
```

输出：

```
1564583031646, pool-1-thread-1, start!  
true  
true  
1564583036649, pool-1-thread-1, end!  
1564583037653, main  
Exception in thread "main" java.util.concurrent.CancellationException  
at java.util.concurrent.FutureTask.report(FutureTask.java:121)  
at java.util.concurrent.FutureTask.get(FutureTask.java:192)  
at com.itsoku.chat18.Demo7.main(Demo7.java:24)
```

输出2个true，表示任务已被取消，已完成，最后调用get方法会触发 CancellationException 异常。

总结：从上面可以看出Future、 Callable接口需要结合ExecutorService来使用，需要有线程池的支持。

FutureTask类

FutureTask除了实现Future接口，还实现了Runnable接口，因此FutureTask可以交给Executor执行，也可以交给线程执行执行（Thread有个Runnable的构造方法），FutureTask表示带返回值结果的任务。

上面我们演示的是通过线程池执行任务然后获取执行结果。

这次我们通过FutureTask类，自己启动一个线程来获取执行结果，示例如下：

```
package com.itsoku.chat18;  
  
import java.util.concurrent.*;  
  
/**  
 * 跟着阿里P7学并发，微信公众号：javacode2018  
 */  
public class Demo9 {  
    public static void main(String[] args) throws ExecutionException,  
InterruptedException {  
        FutureTask<Integer> futureTask = new FutureTask<Integer>(() -> {  
            System.out.println(System.currentTimeMillis() + "," +  
Thread.currentThread().getName() + ",start!");  
            TimeUnit.SECONDS.sleep(5);  
            System.out.println(System.currentTimeMillis() + "," +  
Thread.currentThread().getName() + ",end!");  
            return 10;  
        });  
  
        System.out.println(System.currentTimeMillis() + "," + Thread.currentThread().getNam  
e());  
        new Thread(futureTask).start();
```

```
System.out.println(System.currentTimeMillis()+"."+Thread.currentThread().getName());  
  
System.out.println(System.currentTimeMillis()+"."+Thread.currentThread().getName(),结果:"+futureTask.get());  
}  
}
```

输出：

```
1564585122547,main  
1564585122547,main  
1564585122547,Thread-0,start!  
1564585127549,Thread-0,end!  
1564585122547,main,结果:10
```

大家可以回过头去看一下上面用线程池的submit方法返回的Future实际类型正是FutureTask对象，有兴趣的可以设置个断点去看看。

FutureTask类还是相当重要的，标记一下。

下面3个类，下一篇篇文章进行详解

1. 介绍CompletableFuture
2. 介绍CompletionService
3. 介绍ExecutorCompletionService

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

第20篇：JUC中的Executor框架详解2

本文内容

1. ExecutorCompletionService出现的背景
2. 介绍CompletionService接口及常用的方法
3. 介绍ExecutorCompletionService类及其原理
4. 示例：执行一批任务，然后消费执行结果
5. 示例【2种方式】：异步执行一批任务，有一个完成立即返回，其他取消

需要解决的问题

还是举个例子说明更好理解一些。

买新房了，然后在网上下单买冰箱、洗衣机，电器商家不同，所以送货耗时不一样，然后等他们送货，快递只愿送到楼下，然后我们自己将其搬到楼上的家中。

用程序来模拟上面的实现。示例代码如下：

```
package com.itsoku.chat18;

import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo12 {
    static class GoodsModel {
        //商品名称
        String name;
        //购物开始时间
        long starttime;
        //送到的时间
        long endtime;

        public GoodsModel(String name, long starttime, long endtime) {
            this.name = name;
            this.starttime = starttime;
            this.endtime = endtime;
        }

        @Override
        public String toString() {
            return name + ", 下单时间[" + this.starttime + "," + endtime + "], 耗时：" + (this.endtime - this.starttime);
        }
    }

    /**
     * 将商品搬上楼
     *
     * @param goodsModel
     * @throws InterruptedException
     */
    static void moveUp(GoodsModel goodsModel) throws InterruptedException {
```

```

        //休眠5秒，模拟搬上楼耗时
        TimeUnit.SECONDS.sleep(5);
        System.out.println("将商品搬上楼，商品信息：" + goodsModel);
    }

    /**
     * 模拟下单
     *
     * @param name      商品名称
     * @param costTime 耗时
     * @return
     */
    static Callable<GoodsModel> buyGoods(String name, Long costTime) {
        return () -> {
            long startTime = System.currentTimeMillis();
            System.out.println(startTime + "购买" + name + "下单！");
            //模拟送货耗时
            try {
                TimeUnit.SECONDS.sleep(costTime);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            long endTime = System.currentTimeMillis();
            System.out.println(startTime + name + "送到了！");
            return new GoodsModel(name, startTime, endTime);
        };
    }

    public static void main(String[] args) throws InterruptedException,
ExecutionException {
    long st = System.currentTimeMillis();
    System.out.println(st + "开始购物！");

    //创建一个线程池，用来异步下单
    ExecutorService executor = Executors.newFixedThreadPool(5);
    //异步下单购买冰箱
    Future<GoodsModel> bxFuture = executor.submit(buyGoods("冰箱", 5));
    //异步下单购买洗衣机
    Future<GoodsModel> xyjFuture = executor.submit(buyGoods("洗衣机", 2));
    //关闭线程池
    executor.shutdown();

    //等待冰箱送到
    GoodsModel bxGoodModel = bxFuture.get();
    //将冰箱搬上楼
    moveUp(bxGoodModel);
    //等待洗衣机送到
    GoodsModel xyjGooldModel = xyjFuture.get();
    //将洗衣机搬上楼
    moveUp(xyjGooldModel);
    long et = System.currentTimeMillis();
    System.out.println(et + "货物已送到家里咯，哈哈哈！");
    System.out.println("总耗时：" + (et - st));
}
}

```

输出：

```
1564653121515开始购物！
1564653121588购买冰箱下单！
1564653121588购买洗衣机下单！
1564653121588洗衣机送到了！
1564653121588冰箱送到了！
将商品搬上楼，商品信息：冰箱，下单时间[1564653121588,1564653126590]，耗时：5002
将商品搬上楼，商品信息：洗衣机，下单时间[1564653121588,1564653123590]，耗时：2002
1564653136591货物已送到家里咯，哈哈哈！
总耗时：15076
```

从输出中我们可以看出几个时间：

1. 购买冰箱耗时5秒
2. 购买洗衣机耗时2秒
3. 将冰箱送上楼耗时5秒
4. 将洗衣机送上楼耗时5秒
5. 共计耗时15秒

购买洗衣机、冰箱都是异步执行的，我们先把冰箱送上楼了，然后再把冰箱送上楼了。上面大家应该发现了一个问题，洗衣机先到的，洗衣机到了，我们并没有去把洗衣机送上楼，而是在等待冰箱到货（`bxFuture.get()`），然后将冰箱送上楼，中间导致浪费了3秒，现实中应该是这样的，先到的先送上楼，修改一下代码，洗衣机先到的，先送洗衣机上楼，代码如下：

```
package com.itsoku.chat18;

import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo13 {
    static class GoodsModel {
        //商品名称
        String name;
        //购物开始时间
        long starttime;
        //送到的时间
        long endtime;

        public GoodsModel(String name, long starttime, long endtime) {
            this.name = name;
            this.starttime = starttime;
            this.endtime = endtime;
        }

        @Override
        public String toString() {
            return name + "，下单时间[" + this.starttime + "," + endtime + "]", 耗时：" + (this.endtime - this.starttime);
        }
    }

    /**
     * 将商品搬上楼
     *
     * @param goodsModel
     */
```

```

    * @throws InterruptedException
    */
    static void moveUp(GoodsModel goodsModel) throws InterruptedException {
        //休眠5秒，模拟搬上楼耗时
        TimeUnit.SECONDS.sleep(5);
        System.out.println("将商品搬上楼，商品信息：" + goodsModel);
    }

    /**
     * 模拟下单
     *
     * @param name      商品名称
     * @param costTime 耗时
     * @return
     */
    static Callable<GoodsModel> buyGoods(String name, Long costTime) {
        return () -> {
            long startTime = System.currentTimeMillis();
            System.out.println(startTime + "购买" + name + "下单！");
            //模拟送货耗时
            try {
                TimeUnit.SECONDS.sleep(costTime);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            long endTime = System.currentTimeMillis();
            System.out.println(endTime + name + "送到了！");
            return new GoodsModel(name, startTime, endTime);
        };
    }

    public static void main(String[] args) throws InterruptedException,
    ExecutionException {
        long st = System.currentTimeMillis();
        System.out.println(st + "开始购物！");

        //创建一个线程池，用来异步下单
        ExecutorService executor = Executors.newFixedThreadPool(5);
        //异步下单购买冰箱
        Future<GoodsModel> bxFuture = executor.submit(buyGoods("冰箱", 5));
        //异步下单购买洗衣机
        Future<GoodsModel> xyjFuture = executor.submit(buyGoods("洗衣机", 2));
        //关闭线程池
        executor.shutdown();

        //等待洗衣机送到
        GoodsModel xyjGoodModel = xyjFuture.get();
        //将洗衣机搬上楼
        moveUp(xyjGoodModel);
        //等待冰箱送到
        GoodsModel bxGoodModel = bxFuture.get();
        //将冰箱搬上楼
        moveUp(bxGoodModel);
        long et = System.currentTimeMillis();
        System.out.println(et + "货物已送到家里咯，哈哈哈！");
        System.out.println("总耗时：" + (et - st));
    }
}

```

输出：

```
1564653153393开始购物！  
1564653153466购买洗衣机下单！  
1564653153466购买冰箱下单！  
1564653155467洗衣机送到了！  
1564653158467冰箱送到了！  
将商品搬上楼，商品信息：洗衣机，下单时间[1564653153466, 1564653155467]，耗时：2001  
将商品搬上楼，商品信息：冰箱，下单时间[1564653153466, 1564653158467]，耗时：5001  
1564653165469货物已送到家里咯，哈哈哈！  
总耗时：12076
```

耗时12秒，比第一种少了3秒。

问题来了，上面是我们通过调整代码达到了最优效果，实际上，购买冰箱和洗衣机具体哪个耗时时间长我们是不知道的，怎么办呢，有没有什么解决办法？

CompletionService接口

CompletionService相当于一个执行任务的服务，通过submit丢任务给这个服务，服务内部去执行任务，可以通过服务提供的一些方法获取服务中已经完成的任务。

接口内的几个方法：

```
Future<V> submit(Callable<V> task);
```

用于向服务中提交有返回结果的任务，并返回Future对象

```
Future<V> submit(Runnable task, V result);
```

用户向服务中提交有返回值的任务去执行，并返回Future对象

```
Future<V> take() throws InterruptedException;
```

从服务中返回并移除一个已经完成的任务，如果获取不到，会一致阻塞到有返回值为止。此方法会响应线程中断。

```
Future<V> poll();
```

从服务中返回并移除一个已经完成的任务，如果内部没有已经完成的任务，则返回空，此方法会立即响应。

```
Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException;
```

尝试在指定的时间内从服务中返回并移除一个已经完成的任务，等待的时间超时还是没有获取到已完成的任务，则返回空。此方法会响应线程中断

通过submit向内部提交任意多个任务，通过take方法可以获取已经执行完成的任务，如果获取不到将等待。

ExecutorCompletionService类

ExecutorCompletionService类是CompletionService接口的具体实现。

说一下其内部原理，ExecutorCompletionService创建的时候会传入一个线程池，调用submit方法传入需要执行的任务，任务由内部的线程池来处理；ExecutorCompletionService内部有个阻塞队列，任意一个任务完成之后，会将任务的执行结果（Future类型）放入阻塞队列中，然后其他线程可以调用它take、poll方法从这个阻塞队列中获取一个已经完成的任务，获取任务返回结果的顺序和任务执行完成的先后顺序一致，所以最先完成的任务会先返回。

关于阻塞队列的知识后面会专门抽几篇来讲，大家可以关注一下后面的文章。

看一下构造方法：

```
public ExecutorCompletionService(Executor executor) {  
    if (executor == null)  
        throw new NullPointerException();  
    this.executor = executor;  
    this.aes = (executor instanceof AbstractExecutorService) ?  
        (AbstractExecutorService) executor : null;  
    this.completionQueue = new LinkedBlockingQueue<Future<?>>();  
}
```

构造方法需要传入一个Executor对象，这个对象表示任务执行器，所有传入的任务会被这个执行器执行。

completionQueue是用来存储任务结果的阻塞队列，默认用采用的是LinkedBlockingQueue，也支持自己设置。通过submit传入需要执行的任务，任务执行完成之后，会放入completionQueue中，有兴趣的可以看一下原码，还是很好理解的。

使用ExecutorCompletionService解决文章开头的问题

代码如下：

```
package com.itsoku.chat18;  
  
import java.util.concurrent.*;  
  
/**  
 * 跟着阿里P7学并发，微信公众号：javacode2018  
 */  
public class Demo14 {  
    static class GoodsModel {  
        //商品名称  
        String name;  
        //购物开始时间  
        long starttime;  
        //送到的时间  
        long endtime;  
  
        public GoodsModel(String name, long starttime, long endtime) {  
            this.name = name;  
            this.starttime = starttime;  
            this.endtime = endtime;  
        }  
    }  
}
```

```

    }

    @Override
    public String toString() {
        return name + ", 下单时间[" + this starttime + "," + endtime + "], 耗时:" + (this.endtime - this.starttime);
    }
}

/**
 * 将商品搬上楼
 *
 * @param goodsModel
 * @throws InterruptedException
 */
static void moveUp(GoodsModel goodsModel) throws InterruptedException {
    //休眠5秒，模拟搬上楼耗时
    TimeUnit.SECONDS.sleep(5);
    System.out.println("将商品搬上楼，商品信息：" + goodsModel);
}

/**
 * 模拟下单
 *
 * @param name      商品名称
 * @param costTime 耗时
 * @return
 */
static Callable<GoodsModel> buyGoods(String name, Long costTime) {
    return () -> {
        long startTime = System.currentTimeMillis();
        System.out.println(startTime + "购买" + name + "下单！");
        //模拟送货耗时
        try {
            TimeUnit.SECONDS.sleep(costTime);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long endTime = System.currentTimeMillis();
        System.out.println(endTime + name + "送到了！");
        return new GoodsModel(name, startTime, endTime);
    };
}

public static void main(String[] args) throws InterruptedException,
ExecutionException {
    long st = System.currentTimeMillis();
    System.out.println(st + "开始购物！");
    ExecutorService executor = Executors.newFixedThreadPool(5);

    //创建ExecutorCompletionService对象
    ExecutorCompletionService<GoodsModel> executorCompletionService = new
    ExecutorCompletionService<>(executor);
    //异步下单购买冰箱
    executorCompletionService.submit(buyGoods("冰箱", 5));
    //异步下单购买洗衣机
    executorCompletionService.submit(buyGoods("洗衣机", 2));
    executor.shutdown();
}

```

```

//购买商品的数量
int goodsCount = 2;
for (int i = 0; i < goodsCount; i++) {
    //可以获取到最先到的商品
    GoodsModel goodsModel = executorCompletionService.take().get();
    //将最先到的商品送上楼
    moveUp(goodsModel);
}

long et = System.currentTimeMillis();
System.out.println(et + "货物已送到家里咯，哈哈！");
System.out.println("总耗时：" + (et - st));
}
}

```

输出：

```

1564653208284开始购物！
1564653208349购买冰箱下单！
1564653208349购买洗衣机下单！
1564653210349洗衣机送到了！
1564653213350冰箱送到了！
将商品搬上楼，商品信息：洗衣机，下单时间[1564653208349,1564653210349]，耗时：2000
将商品搬上楼，商品信息：冰箱，下单时间[1564653208349,1564653213350]，耗时：5001
1564653220350货物已送到家里咯，哈哈！
总耗时：12066

```

从输出中可以看出和我们希望的结果一致，代码中下单顺序是：冰箱、洗衣机，冰箱送货耗时5秒，洗衣机送货耗时2秒，洗衣机先到的，然后被送上楼了，冰箱后到被送上楼，总共耗时12秒，和期望的方案一样。

示例：执行一批任务，然后消费执行结果

代码如下：

```

package com.itsoku.chat18;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.*;
import java.util.function.Consumer;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo15 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        List<Callable<Integer>> list = new ArrayList<>();
        int taskCount = 5;
        for (int i = taskCount; i > 0; i--) {

```

```

        int j = i * 2;
        list.add(() -> {
            TimeUnit.SECONDS.sleep(j);
            return j;
        });
    }
    solve(executorService, list, a -> {
        System.out.println(System.currentTimeMillis() + ":" + a);
    });
    executorService.shutdown();
}

public static <T> void solve(Executor e, Collection<Callable<T>> solvers,
Consumer<T> use) throws InterruptedException, ExecutionException {
    CompletionService<T> ecs = new ExecutorCompletionService<T>(e);
    for (Callable<T> s : solvers) {
        ecs.submit(s);
    }
    int n = solvers.size();
    for (int i = 0; i < n; ++i) {
        T r = ecs.take().get();
        if (r != null) {
            use.accept(r);
        }
    }
}
}

```

输出：

```

1564667625648:2
1564667627652:4
1564667629649:6
1564667631652:8
1564667633651:10

```

代码中传入了一批任务进行处理，最终将所有处理完成的按任务完成的先后顺序传递给 `Consumer` 进行消费了。

示例：异步执行一批任务，有一个完成立即返回，其他取消

这个给大家讲解2种方式。

方式1

使用`ExecutorCompletionService`实现，`ExecutorCompletionService`提供了获取一批任务中最先完成的任务结果的能力。

代码如下：

```
package com.itsoku.chat18;
```

```

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.*;
import java.util.function.Consumer;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo16 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        long starttime = System.currentTimeMillis();
        ExecutorService executorService = Executors.newFixedThreadPool(5);
        List<Callable<Integer>> list = new ArrayList<>();
        int taskCount = 5;
        for (int i = taskCount; i > 0; i--) {
            int j = i * 2;
            String taskName = "任务"+i;
            list.add(() -> {
                TimeUnit.SECONDS.sleep(j);
                System.out.println(taskName+"执行完毕！");
                return j;
            });
        }
        Integer integer = invokeAny(executorService, list);
        System.out.println("耗时：" + (System.currentTimeMillis() - starttime) + "，");
        执行结果：" + integer);
        executorService.shutdown();
    }
}

```

```

public static <T> T invokeAny(Executor e, Collection<Callable<T>> solvers)
throws InterruptedException, ExecutionException {
    CompletionService<T> ecs = new ExecutorCompletionService<T>(e);
    List<Future<T>> futureList = new ArrayList<>();
    for (Callable<T> s : solvers) {
        futureList.add(ecs.submit(s));
    }
    int n = solvers.size();
    try {
        for (int i = 0; i < n; ++i) {
            T r = ecs.take().get();
            if (r != null) {
                return r;
            }
        }
    } finally {
        for (Future<T> future : futureList) {
            future.cancel(true);
        }
    }
    return null;
}
}

```

程序输出下面结果然后停止了：

任务1执行完毕！

耗时:2072, 执行结果:2

代码中执行了5个任务，使用CompletionService执行任务，调用take方法获取最先执行完成的任务，然后返回。在finally中对所有任务发送取消操作（`future.cancel(true);`），从输出中可以看出只有任务1执行成功，其他任务被成功取消了，符合预期结果。

方式2

其实 ExecutorService 已经为我们提供了这样的方法，方法声明如下：

```
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
    throws InterruptedException, ExecutionException;
```

示例代码：

```
package com.itsoku.chat18;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo17 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        long starttime = System.currentTimeMillis();
        ExecutorService executorService = Executors.newFixedThreadPool(5);

        List<Callable<Integer>> list = new ArrayList<>();
        int taskCount = 5;
        for (int i = taskCount; i > 0; i--) {
            int j = i * 2;
            String taskName = "任务" + i;
            list.add(() -> {
                TimeUnit.SECONDS.sleep(j);
                System.out.println(taskName + "执行完毕！");
                return j;
            });
        }
        Integer integer = executorService.invokeAny(list);
        System.out.println("耗时：" + (System.currentTimeMillis() - starttime) + "，
执行结果：" + integer);
        executorService.shutdown();
    }
}
```

输出下面结果之后停止：

任务1执行完毕！

耗时: 2061, 执行结果: 2

输出结果和方式1中结果类似。

更多好文章

1. [spring高手系列 \(正在连载中\)](#)
2. [Java高并发系列 \(共34篇\)](#)
3. [MySql高手系列 \(共27篇\)](#)
4. [Maven高手系列 \(共10篇\)](#)
5. [Mybatis系列 \(共12篇\)](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第21篇：java中的CAS

这是java高并发系列第21篇文章。

本文主要内容

1. 从网站计数器实现中一步步引出CAS操作
2. 介绍java中的CAS及CAS可能存在的问题
3. 悲观锁和乐观锁的一些介绍及数据库乐观锁的一个常见示例
4. 使用java中的原子操作实现网站计数器功能

我们需要解决的问题

需求：我们开发了一个网站，需要对访问量进行统计，用户每次发一次请求，访问量+1，如何实现呢？

下面我们来模仿有100个人同时访问，并且每个人对咱们的网站发起10次请求，最后总访问次数应该是1000次。实现访问如下。

方式1

代码如下：

```
package com.itsoku.chat20;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    //访问次数
    static int count = 0;

    //模拟访问一次
    public static void request() throws InterruptedException {
        //模拟耗时5毫秒
        TimeUnit.MILLISECONDS.sleep(5);
        count++;
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        int threadSize = 100;
        CountDownLatch countDownLatch = new CountDownLatch(threadSize);
        for (int i = 0; i < threadSize; i++) {
            Thread thread = new Thread(() -> {
                try {
                    for (int j = 0; j < 10; j++) {
                        request();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    countDownLatch.countDown();
                }
            });
            thread.start();
        }

        countDownLatch.await();
        long endTime = System.currentTimeMillis();
        System.out.println(Thread.currentThread().getName() + "，耗时：" + (endTime - startTime) + ",count=" + count);
    }
}
```

输出：

```
main, 耗时: 138, count=975
```

代码中的count用来记录总访问次数，request()方法表示访问一次，内部休眠5毫秒模拟内部耗时，request方法内部对count++操作。程序最终耗时1秒多，执行还是挺快的，但是count和我们期望的结果不一致，我们期望的是1000，实际输出的是973（每次运行结果可能都不一样）。

分析一下问题出在哪呢？

代码中采用的是多线程的方式来操作count，count++会有线程安全问题，count++操作实际上是由以下三步操作完成的：

1. 获取count的值，记做A: A=count
2. 将A的值+1，得到B: B = A+1
3. 让B赋值给count: count = B

如果有A、B两个线程同时执行count++，他们同时执行到上面步骤的第1步，得到的count是一样的，3步操作完成之后，count只会+1，导致count只加了一次，从而导致结果不准确。

那么我们应该怎么做的呢？

对count++操作的时候，我们让多个线程排队处理，多个线程同时到达request()方法的时候，只能允许一个线程可以进去操作，其他的线程在外面候着，等里面的处理完毕出来之后，外面等着的再进去一个，这样操作count++就是排队进行的，结果一定是正确的。

我们前面学了synchronized、ReentrantLock可以对资源加锁，保证并发的正确性，多线程情况下可以保证被锁的资源被串行访问，那么我们用synchronized来实现一下。

方式2：使用synchronized实现

代码如下：

```
package com.itsoku.chat20;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo2 {
    //访问次数
    static int count = 0;

    //模拟访问一次
    public static synchronized void request() throws InterruptedException {
        //模拟耗时5毫秒
        TimeUnit.MILLISECONDS.sleep(5);
        count++;
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        int threadSize = 100;
        CountDownLatch countDownLatch = new CountDownLatch(threadSize);
        for (int i = 0; i < threadSize; i++) {
            Thread thread = new Thread(() -> {
                try {
                    request();
                } catch (InterruptedException e) {
                }
            });
            thread.start();
        }
        countDownLatch.await();
        System.out.println("耗时：" + (System.currentTimeMillis() - startTime));
    }
}
```

```

        for (int j = 0; j < 10; j++) {
            request();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        countDownLatch.countDown();
    }
});

thread.start();
}

countDownLatch.await();
long endTime = System.currentTimeMillis();
System.out.println(Thread.currentThread().getName() + "，耗时：" +
(endTime - startTime) + ",count=" + count);
}
}

```

输出：

```
main, 耗时: 5563, count=1000
```

程序中request方法使用synchronized关键字，保证了并发情况下，request方法同一时刻只允许一个线程访问，request加锁了相当于串行执行了，count的结果和我们预期的结果一致，只是耗时比较长，5秒多。

方式3

我们在看一下count++操作，count++操作实际上是被拆分为3步骤执行：

1. 获取count的值，记做A: A=count
2. 将A的值+1，得到B: B = A+1
3. 让B赋值给count: count = B

方式2中我们通过加锁的方式让上面3步骤同时只能被一个线程操作，从而保证结果的正确性。

我们是否可以只在第3步加锁，减少加锁的范围，对第3步做以下处理：

```

获取锁
第三步获取一下count最新的值，记做LV
判断LV是否等于A，如果相等，则将B的值赋给count，并返回true，否者返回false
释放锁

```

如果我们发现第3步返回的是false，我们就再次去获取count，将count赋值给A，对A+1赋值给B，然后再将A、B的值带入到上面的过程中执行，直到上面的结果返回true为止。

我们用代码来实现，如下：

```

package com.itsoku.chat20;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

```

```
/**  
 * 跟着阿里p7学并发，微信公众号：javacode2018  
 */  
public class Demo3 {  
    //访问次数  
    volatile static int count = 0;  
  
    //模拟访问一次  
    public static void request() throws InterruptedException {  
        //模拟耗时5毫秒  
        TimeUnit.MILLISECONDS.sleep(5);  
        int expectCount;  
        do {  
            expectCount = getCount();  
        } while (!compareAndSwap(expectCount, expectCount + 1));  
    }  
  
    /**  
     * 获取count当前的值  
     *  
     * @return  
     */  
    public static int getCount() {  
        return count;  
    }  
  
    /**  
     * @param expectCount 期望count的值  
     * @param newCount      需要给count赋的新值  
     * @return  
     */  
    public static synchronized boolean compareAndSwap(int expectCount, int  
newCount) {  
        //判断count当前值是否和期望的expectCount一样，如果一样将newCount赋值给count  
        if (getCount() == expectCount) {  
            count = newCount;  
            return true;  
        }  
        return false;  
    }  
  
    public static void main(String[] args) throws InterruptedException {  
        long startTime = System.currentTimeMillis();  
        int threadSize = 100;  
        CountDownLatch countDownLatch = new CountDownLatch(threadSize);  
        for (int i = 0; i < threadSize; i++) {  
            Thread thread = new Thread(() -> {  
                try {  
                    for (int j = 0; j < 10; j++) {  
                        request();  
                    }  
                } catch (InterruptedException e) {  
                    e.printStackTrace();  
                } finally {  
                    countDownLatch.countDown();  
                }  
            });  
        };  
    };
```

```

        thread.start();
    }

    countDownLatch.await();
    long endTime = System.currentTimeMillis();
    System.out.println(Thread.currentThread().getName() + ", 耗时: " +
(endTime - startTime) + ", count=" + count);
}
}

```

输出：

```
main, 耗时: 116, count=1000
```

代码中用了 `volatile` 关键字修饰了 `count`, 可以保证 `count` 在多线程情况下的可见性。关于 `volatile` 关键字的使用, 也是非常非常重要的, 前面有讲过, 不太了解的朋友可以去看一下: [volatile与Java内存模型](#)

咱们再看一下代码, `compareAndSwap` 方法, 我们给起个简称吧叫 `CAS`, 这个方法有什么作用呢? 这个方法使用 `synchronized` 修饰了, 能保证此方法是线程安全的, 多线程情况下此方法是串行执行的。方法由两个参数, `expectCount`: 表示期望的值, `newCount`: 表示要给 `count` 设置的新值。方法内部通过 `getCount()` 获取 `count` 当前的值, 然后与期望的值 `expectCount` 比较, 如果期望的值和 `count` 当前的值一致, 则将新值 `newCount` 赋值给 `count`。

再看一下 `request()` 方法, 方法中有个 `do-while` 循环, 循环内部获取 `count` 当前值赋值给了 `expectCount`, 循环结束的条件是 `compareAndSwap` 返回 `true`, 也就是说如果 `compareAndSwap` 如果不成功, 循环再次获取 `count` 的最新值, 然后 `+1`, 再次调用 `compareAndSwap` 方法, 直到 `compareAndSwap` 返回成功为止。

代码中相当于将 `count++` 拆分开了, 只对最后一步加锁了, 减少了锁的范围, 此代码的性能是不是比方式2快不少, 还能保证结果的正确性。大家是不是感觉这个 `compareAndSwap` 方法挺好的, 这东西确实很好, java 中已经给我们提供了 CAS 的操作, 功能非常强大, 我们继续向下看。

CAS

CAS, compare and swap 的缩写, 中文翻译成比较并交换。

CAS 操作包含三个操作数 —— 内存位置 (V) 、预期原值 (A) 和新值(B)。如果内存位置的值与预期原值相匹配, 那么处理器会自动将该位置值更新为新值。否则, 处理器不做任何操作。无论哪种情况, 它都会在 CAS 指令之前返回该位置的值。 (在 CAS 的一些特殊情况下将仅返回 CAS 是否成功, 而不提取当前值。) CAS 有效地说明了“我认为位置 V 应该包含值 A; 如果包含该值, 则将 B 放到这个位置; 否则, 不要更改该位置, 只告诉我这个位置现在的值即可。”

通常将 CAS 用于同步的方式是从地址 V 读取值 A, 执行多步计算来获得新值 B, 然后使用 CAS 将 V 的值从 A 改为 B。如果 V 处的值尚未同时更改, 则 CAS 操作成功。

系统底层进行 CAS 操作的时候, 会判断当前系统是否为多核系统, 如果是就给总线加锁, 只有一个线程会对总线加锁成功, 加锁成功之后会执行 cas 操作, 也就是说 CAS 的原子性实际上是 CPU 实现的, 其实在这一点上还是有排他锁的, 只是比起用 `synchronized`, 这里的排他时间要短得多, 所以在多线程情况下性能会比较好。

java 中提供了对 CAS 操作的支持, 具体在 `sun.misc.Unsafe` 类中, 声明如下:

```
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);
public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);
```

上面三个方法都是类似的，主要对4个参数做一下说明。

var1：表示要操作的对象

var2：表示要操作对象中属性地址的偏移量

var4：表示需要修改数据的期望的值

var5：表示需要修改为的新值

JUC包中大部分功能都是依靠CAS操作完成的，所以这块也是非常重要的，有关Unsafe类，下篇文章会具体讲解。

`synchronized`、`ReentrantLock`这种独占锁属于**悲观锁**，它是在假设需要操作的代码一定会发生冲突的，执行代码的时候先对代码加锁，让其他线程在外面等候排队获取锁。悲观锁如果锁的时间比较长，会导致其他线程一直处于等待状态，像我们部署的web应用，一般部署在tomcat中，内部通过线程池来处理用户的请求，如果很多请求都处于等待获取锁的状态，可能会耗尽tomcat线程池，从而导致系统无法处理后面的请求，导致服务器处于不可用状态。

除此之外，还有**乐观锁**，乐观锁的含义就是假设系统没有发生并发冲突，先按无锁方式执行业务，到最后了检查执行业务期间是否有并发导致数据被修改了，如果有并发导致数据被修改了，就快速返回失败，这样的操作使系统并发性能更高一些。cas中就使用了这样的操作。

关于乐观锁这块，想必大家在数据库中也有用到过，给大家举个例子，可能以后会用到。

如果你们的网站中有调用支付宝充值接口的，支付宝那边充值成功了会回调商户系统，商户系统接收到请求之后怎么处理呢？假设用户通过支付宝在商户系统中充值100，支付宝那边会从用户账户中扣除100，商户系统接收到支付宝请求之后应该在商户系统中给用户账户增加100，并且把订单状态置为成功。

处理过程如下：

```
开启事务
获取订单信息
if(订单状态==待处理){
    给用户账户增加100
    将订单状态更新为成功
}
返回订单处理成功
提交事务
```

由于网络等各种问题，可能支付宝回调商户系统的时候，回调超时了，支付宝又发起了一笔回调请求，刚好这2笔请求同时到达上面代码，最终结果是给用户账户增加了200，这样事情就搞大了，公司蒙受损失，严重点可能让公司就此倒闭了。

那我们可以用乐观锁来实现，给订单表加个版本号version，要求每次更新订单数据，将版本号+1，那么上面的过程可以改为：

```
获取订单信息,将version的值赋值给V_A
if(订单状态==待处理){
    开启事务
    给用户账户增加100
    update影响行数 = update 订单表 set version = version + 1 where id = 订单号 and
    version = V_A;
    if(update影响行数==1){
        提交事务
    }else{
        回滚事务
    }
}
返回订单处理成功
```

上面的update语句相当于我们说的CAS操作，执行这个update语句的时候，多线程情况下，数据库会对当前订单记录加锁，保证只有一条执行成功，执行成功的，影响行数为1，执行失败的影响行数为0，根据影响行数来决定提交还是回滚事务。上面操作还有一点是将事务范围缩小了，也提升了系统并发处理的性能。这个知识点希望你们能get到。

CAS 的问题

cas这么好用，那么有没有什么问题呢？还真有

ABA问题

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。这就是CAS的ABA问题。常见的解决思路是使用版本号。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。目前在JDK的atomic包里提供了一个类 `AtomicStampedReference` 来解决ABA问题。这个类的`compareAndSet`方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

循环时间长开销大

上面我们说过如果CAS不成功，则会原地循环（自旋操作），如果长时间自旋会给CPU带来非常大的执行开销。并发量比较大的情况下，CAS成功概率可能比较低，可能会重试很多次才会成功。

使用JUC中的类实现计数器

juc框架中提供了一些原子操作，底层是通过Unsafe类中的cas操作实现的。通过原子操作可以保证数据在并发情况下的正确性。

此处我们使用 `java.util.concurrent.atomic.AtomicInteger` 类来实现计数器功能，`AtomicInteger` 内部是采用cas操作来保证对int类型数据增减操作在多线程情况下的正确性。

计数器代码如下：

```
package com.itsoku.chat20;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {
    //访问次数
    static AtomicInteger count = new AtomicInteger();

    //模拟访问一次
    public static void request() throws InterruptedException {
        //模拟耗时5毫秒
        TimeUnit.MILLISECONDS.sleep(5);
        //对count原子+1
        count.incrementAndGet();
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        int threadSize = 100;
        CountDownLatch countDownLatch = new CountDownLatch(threadSize);
        for (int i = 0; i < threadSize; i++) {
            Thread thread = new Thread(() -> {
                try {
                    for (int j = 0; j < 10; j++) {
                        request();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    countDownLatch.countDown();
                }
            });
            thread.start();
        }

        countDownLatch.await();
        long endTime = System.currentTimeMillis();
        System.out.println(Thread.currentThread().getName() + "，耗时：" + (endTime - startTime) + ",count=" + count);
    }
}
```

输出：

```
main, 耗时: 119, count=1000
```

耗时很短，并且结果和期望的一致。

关于原子类操作，都位于 `java.util.concurrent.atomic` 包中，下篇文章我们主要来介绍一下这些常用的类及各自的使用场景。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第22篇：java中的Unsafe类

本文主要内容

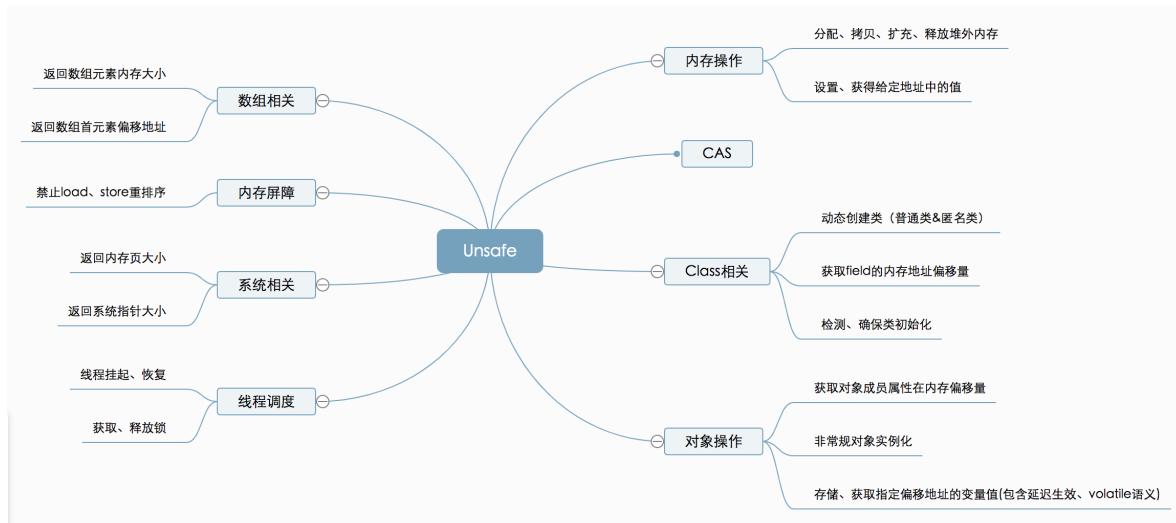
1. 基本介绍、
2. 通过反射获取Unsafe实例
3. Unsafe中的CAS操作
4. Unsafe中原子操作相关方法介绍
5. Unsafe中线程调度相关方法
6. park和unpark示例
7. Unsafe锁示例
8. Unsafe中保证变量的可见性
9. Unsafe中Class相关方法
10. 示例：staticFieldOffset、staticFieldBase、staticFieldBase
11. 示例：shouldBeInitialized、ensureClassInitialized
12. 对象操作的其他方法
13. 绕过构造方法创建对象
14. 数组相关的一些方法
15. 内存屏障相关操作
16. java高并发系列目录

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

基本介绍

最近我们一直在学习java高并发，java高并发中主要涉及到类位于java.util.concurrent包中，简称juc，juc中大部分类都是依赖于Unsafe来实现的，主要用到了Unsafe中的CAS、线程挂起、线程恢复等相关功能。所以如果打算深入了解JUC原理的，必须先了解一下Unsafe类。

先上一幅Unsafe类的功能图：



Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增强Java语言底层资源操作能力方面起到了很大的作用。但由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用Unsafe类会使得程序出错的概率变大，使得Java这种安全的语言变得不再“安全”，因此对Unsafe的使用一定要慎重。

从Unsafe功能图上看出，Unsafe提供的API大致可分为**内存操作**、**CAS**、**Class相关**、**对象操作**、**线程调度**、**系统信息获取**、**内存屏障**、**数组操作**等几类，本文主要介绍3个常用的操作：**CAS**、**线程调度**、**对象操作**。

看一下UnSafe的原码部分：

```
public final class Unsafe {
    // 单例对象
    private static final Unsafe theUnsafe;

    private Unsafe() {
    }
    @CallerSensitive
    public static Unsafe getUnsafe() {
        Class var0 = Reflection.getCallerClass();
        // 仅在引导类加载器`BootstrapClassLoader`加载时才合法
        if(!VM.isSystemDomainLoader(var0.getClassLoader())) {
            throw new SecurityException("Unsafe");
        } else {
            return theUnsafe;
        }
    }
}
```

从代码中可以看出，Unsafe类为单例实现，提供静态方法getUnsafe获取Unsafe实例，内部会判断当前调用者是否是由系统类加载器加载的，如果不是系统类加载器加载的，会抛出 SecurityException 异常。

那我们想使用这个类，如何获取呢？

可以把我们的类放在jdk的lib目录下，那么启动的时候会自动加载，这种方式不是很好。

我们学过反射，通过反射可以获取到 unsafe 中的 theUnsafe 字段的值，这样可以获取到Unsafe对象的实例。

通过反射获取Unsafe实例

代码如下：

```
package com.itsoku.chat21;

import sun.misc.Unsafe;

import java.lang.reflect.Field;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    static Unsafe unsafe;

    static {
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        System.out.println(unsafe);
    }
}
```

输出：

```
sun.misc.Unsafe@76ed5528
```

Unsafe中的CAS操作

看一下Unsafe中CAS相关方法定义：

```
/***
 * CAS 操作
 */
```

```

* @param o      包含要修改field的对象
* @param offset 对象中某field的偏移量
* @param expected 期望值
* @param update 更新值
* @return true | false
*/
public final native boolean compareAndSwapObject(Object o, long offset, Object
expected, Object update);

public final native boolean compareAndSwapInt(Object o, long offset, int
expected,int update);

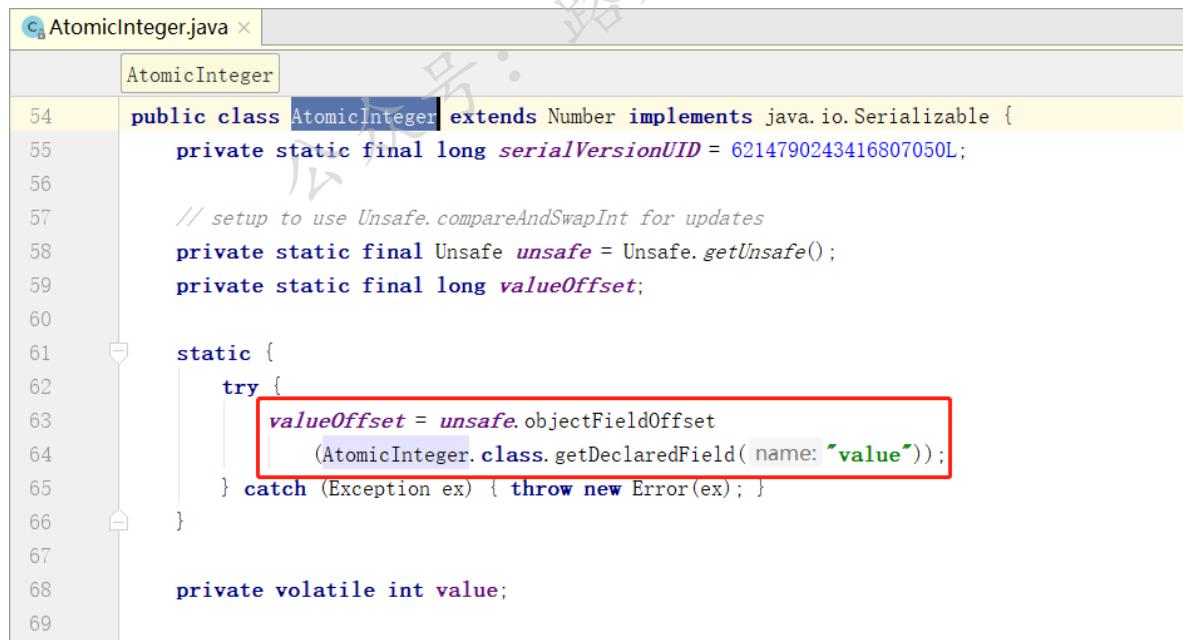
public final native boolean compareAndSwapLong(Object o, long offset, long
expected, long update);

```

什么是CAS? 即比较并替换, 实现并发算法时常用到的一种技术。CAS操作包含三个操作数——**内存位置、预期原值及新值**。执行CAS操作的时候, 将内存位置的值与预期原值比较, 如果相匹配, 那么处理器会自动将该位置值更新为新值, 否则, 处理器不做任何操作, 多个线程同时执行cas操作, 只有一个会成功。我们都知道, CAS是一条CPU的原子指令 (cmpxchg指令), 不会造成所谓的数据不一致问题, Unsafe提供的CAS方法 (如compareAndSwapXXX) 底层实现即为CPU指令cmpxchg。执行 cmpxchg指令的时候, 会判断当前系统是否为多核系统, 如果是就给总线加锁, 只有一个线程会对总线加锁成功, 加锁成功之后会执行cas操作, 也就是说CAS的原子性实际上是CPU实现的, 其实在这一点上还是有排他锁的, 只是比起用synchronized, 这里的排他时间要短的多, 所以在多线程情况下性能会比较好。

说一下offset, offeset为字段的偏移量, 每个对象有个地址, offset是字段相对于对象地址的偏移量, 对象地址记为baseAddress, 字段偏移量记为offeset, 那么字段对应的实际地址就是 baseAddress+offeset, 所以cas通过对象、偏移量就可以去操作字段对应的值了。

CAS在java.util.concurrent.atomic相关类、Java AQS、JUC中并发集合等实现上有非常广泛的应用, 我们看一下 `java.util.concurrent.atomic.AtomicInteger` 类, 这个类可以在多线程环境中对int类型的数据执行高效的原子修改操作, 并保证数据的正确性, 看一下此类中用到Unsafe cas的地方:



```

AtomicInteger.java x
AtomicInteger
54 public class AtomicInteger extends Number implements java.io.Serializable {
55     private static final long serialVersionUID = 6214790243416807050L;
56
57     // setup to use Unsafe.compareAndSwapInt for updates
58     private static final Unsafe unsafe = Unsafe.getUnsafe();
59     private static final long valueOffset;
60
61     static {
62         try {
63             valueOffset = unsafe.objectFieldOffset
64                 (AtomicInteger.class.getDeclaredField( name: "value"));
65         } catch (Exception ex) { throw new Error(ex); }
66     }
67
68     private volatile int value;
69

```

```

156     /*
157     * @return the previous value
158     */
159     public final int getAndIncrement() {
160         return unsafe.getAndAddInt(o: this, valueOffset, i: 1);
161     }
162
163     /**
164      * Atomically decrements by one the current value.
165      *
166      * @return the previous value
167      */
168     public final int getAndDecrement() {
169         return unsafe.getAndAddInt(o: this, valueOffset, i: -1);
170     }
171
172     /**
173      * Atomically adds the given value to the current value.
174      *
175      * @param delta the value to add
176      * @return the previous value
177      */
178     public final int getAndAdd(int delta) {
179         return unsafe.getAndAddInt(o: this, valueOffset, delta);
    }

```

原子+1，并返回加之前的值

原子-1，并返回减之前的值

原子+delta，并返回操作之前的值

JUC中其他地方使用到CAS的地方就不列举了，有兴趣的可以去看一下源码。

Unsafe中原子操作相关方法介绍

5个方法，看一下实现：

```

/**
 * int类型值原子操作，对var2地址对应的值做原子增加操作(增加var4)
 *
 * @param var1 操作的对象
 * @param var2 var2字段内存地址偏移量
 * @param var4 需要加的值
 * @return
 */
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while (!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

/**
 * long类型值原子操作，对var2地址对应的值做原子增加操作(增加var4)
 *
 * @param var1 操作的对象
 * @param var2 var2字段内存地址偏移量
 * @param var4 需要加的值
 * @return 返回旧值
 */
public final long getAndAddLong(Object var1, long var2, long var4) {
}

```

```
    long var6;
    do {
        var6 = this.getLongVolatile(var1, var2);
    } while (!this.compareAndSwapLong(var1, var2, var6, var6 + var4));

    return var6;
}

/**
 * int类型值原子操作方法，将var2地址对应的值置为var4
 *
 * @param var1 操作的对象
 * @param var2 var2字段内存地址偏移量
 * @param var4 新值
 * @return 返回旧值
 */
public final int getAndSetInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while (!this.compareAndSwapInt(var1, var2, var5, var4));

    return var5;
}

/**
 * long类型值原子操作方法，将var2地址对应的值置为var4
 *
 * @param var1 操作的对象
 * @param var2 var2字段内存地址偏移量
 * @param var4 新值
 * @return 返回旧值
 */
public final long getAndSetLong(Object var1, long var2, long var4) {
    long var6;
    do {
        var6 = this.getLongVolatile(var1, var2);
    } while (!this.compareAndSwapLong(var1, var2, var6, var4));

    return var6;
}

/**
 * Object类型值原子操作方法，将var2地址对应的值置为var4
 *
 * @param var1 操作的对象
 * @param var2 var2字段内存地址偏移量
 * @param var4 新值
 * @return 返回旧值
 */
public final Object getAndSetObject(Object var1, long var2, Object var4) {
    Object var5;
    do {
        var5 = this.getObjectVolatile(var1, var2);
    } while (!this.compareAndSwapObject(var1, var2, var5, var4));

    return var5;
}
```

看一下上面的方法，内部通过自旋的CAS操作实现的，这些方法都可以保证操作的数据在多线程环境中的原子性，正确性。

来个示例，我们还是来实现一个网站计数功能，同时有100个人发起对网站的请求，每个人发起10次请求，每次请求算一次，最终结果是1000次，代码如下：

```
package com.itsoku.chat21;

import sun.misc.Unsafe;

import java.lang.reflect.Field;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo2 {
    static Unsafe unsafe;
    //用来记录网站访问量，每次访问+1
    static int count;
    //count在Demo2.class对象中的地址偏移量
    static long countOffset;

    static {
        try {
            //获取Unsafe对象
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);

            Field countField = Demo2.class.getDeclaredField("count");
            //获取count字段在Demo2中的内存地址的偏移量
            countOffset = unsafe.staticFieldOffset(countField);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //模拟访问一次
    public static void request() throws InterruptedException {
        //模拟耗时5毫秒
        TimeUnit.MILLISECONDS.sleep(5);
        //对count原子加1
        unsafe.getAndAddInt(Demo2.class, countOffset, 1);
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        int threadSize = 100;
        CountDownLatch countDownLatch = new CountDownLatch(threadSize);
        for (int i = 0; i < threadSize; i++) {
            Thread thread = new Thread(() -> {
                try {
                    for (int j = 0; j < 10; j++) {
                        request();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
            thread.start();
        }
        countDownLatch.countDown();
        countDownLatch.await();
        long endTime = System.currentTimeMillis();
        System.out.println("耗时：" + (endTime - startTime));
    }
}
```

```

        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    } finally {
        countDownLatch.countDown();
    }
});

thread.start();
}

countDownLatch.await();
long endTime = System.currentTimeMillis();
System.out.println(Thread.currentThread().getName() + ", 耗时: " +
(endTime - startTime) + ", count=" + count);
}
}

```

输出：

```
main, 耗时: 114, count=1000
```

代码中我们在静态块中通过反射获取到了Unsafe类的实例，然后获取Demo2中count字段内存地址偏移量 `countoffset`，main方法中模拟了100个人，每人发起10次请求，等到所有请求完毕之后，输出count的结果。

代码中用到了 `CountDownLatch`，通过 `countDownLatch.await()` 让主线程等待，等待100个子线程都执行完毕之后，主线程在进行运行。`CountDownLatch` 的使用可以参考：[java高并发系列 - 第16天：JUC中等待多线程完成的工具类CountDownLatch，必备技能](#)

Unsafe中线程调度相关方法

这部分，包括线程挂起、恢复、锁机制等方法。

```

//取消阻塞线程
public native void unpark(Object thread);
//阻塞线程, isAbsolute: 是否是绝对时间, 如果为true, time是一个绝对时间, 如果为false, time是一个相对时间, time表示纳秒
public native void park(boolean isAbsolute, long time);
//获得对象锁(可重入锁)
@Deprecated
public native void monitorEnter(Object o);
//释放对象锁
@Deprecated
public native void monitorExit(Object o);
//尝试获取对象锁
@Deprecated
public native boolean tryMonitorEnter(Object o);

```

调用 `park` 后，线程将被阻塞，直到 `unpark` 调用或者超时，如果之前调用过 `unpark`，不会进行阻塞，即 `park` 和 `unpark` 不区分先后顺序。`monitorEnter`、`monitorExit`、`tryMonitorEnter` 3个方法已过期，不建议使用了。

park和unpark示例

代码如下：

```
package com.itsoku.chat21;

import sun.misc.Unsafe;

import java.lang.reflect.Field;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo3 {
    static Unsafe unsafe;

    static {
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 调用park和unpark，模拟线程的挂起和唤醒
     *
     * @throws InterruptedException
     */
    public static void m1() throws InterruptedException {
        Thread thread = new Thread(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",start");
            unsafe.park(false, 0);
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",end");
        });
        thread.setName("thread1");
        thread.start();

        TimeUnit.SECONDS.sleep(5);
        unsafe.unpark(thread);
    }

    /**
     * 阻塞指定的时间
     */
    public static void m2() {
        Thread thread = new Thread(() -> {
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",start");
            //线程挂起3秒
            unsafe.park(false, TimeUnit.SECONDS.toNanos(3));
        });
    }
}
```

```

        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ",end");
    });
    thread.setName("thread2");
    thread.start();
}

public static void main(String[] args) throws InterruptedException {
    m1();
    m2();
}
}

```

输出：

```

1565000238474,thread1,start
1565000243475,thread1,end
1565000243475,thread2,start
1565000246476,thread2,end

```

m1()中thread1调用park方法， park方法会将**当前线程阻塞**，被阻塞了5秒之后，被主线程调用unpark方法给唤醒了， unpark方法参数表示需要唤醒的线程。

线程中相当于有个许可，许可默认是0，调用park的时候，发现是0会阻塞当前线程，调用unpark之后，许可会被置为1，并会唤醒当前线程。如果在park之前先调用了unpark方法，执行park方法的时候，不会阻塞。park方法被唤醒之后，许可又会被置为0。多次调用unpark的效果是一样的，许可还是1。

juc中的 LockSupport 类是通过unpark和park方法实现的，需要了解LockSupport可以移步：[JUC中的 LockSupport工具类](#)

Unsafe锁示例

代码如下：

```

package com.itsoku.chat21;

import sun.misc.Unsafe;

import java.lang.reflect.Field;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {

    static Unsafe unsafe;
    //用来记录网站访问量，每次访问+1
    static int count;

    static {
        try {

```

```

        Field field = unsafe.class.getDeclaredField("theUnsafe");
        field.setAccessible(true);
        unsafe = (Unsafe) field.get(null);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

//模拟访问一次
public static void request() {
    unsafe.monitorEnter(Demo4.class);
    try {
        count++;
    } finally {
        unsafe.monitorExit(Demo4.class);
    }
}

public static void main(String[] args) throws InterruptedException {
    long startTime = System.currentTimeMillis();
    int threadsize = 100;
    CountDownLatch countDownLatch = new CountDownLatch(threadsize);
    for (int i = 0; i < threadsize; i++) {
        Thread thread = new Thread(() -> {
            try {
                for (int j = 0; j < 10; j++) {
                    request();
                }
            } finally {
                countDownLatch.countDown();
            }
        });
        thread.start();
    }

    countDownLatch.await();
    long endTime = System.currentTimeMillis();
    System.out.println(Thread.currentThread().getName() + ", 耗时: " +
(endTime - startTime) + ", count=" + count);
    }
}

```

输出：

```
main, 耗时: 64, count=1000
```

monitorEnter、monitorExit都有1个参数，表示上锁的对象。用法和synchronized关键字语义类似。

注意：

1. monitorEnter、monitorExit、tryMonitorEnter 3个方法已过期，不建议使用了
2. monitorEnter、monitorExit必须成对出现，出现的次数必须一致，也就是说锁了n次，也必须释放n次，否则会造成死锁

Unsafe中保证变量的可见性

关于变量可见性需要先了解java内存模型JMM，可以移步到：

[JMM相关的一些概念](#)

[volatile与Java内存模型](#)

java中操作内存分为主内存和工作内存，共享数据在主内存中，线程如果需要操作主内存的数据，需要先将主内存的数据复制到线程独有的工作内存中，操作完成之后再将其刷新到主内存中。如线程A要想看到线程B修改后的数据，需要满足：线程B修改数据之后，需要将数据从自己的工作内存中刷新到主内存中，并且A需要去主内存中读取数据。

被关键字volatile修饰的数据，有2点语义：

1. 如果一个变量被volatile修饰，读取这个变量时候，会强制从主内存中读取，然后将其复制到当前线程的工作内存中使用
2. 给volatile修饰的变量赋值的时候，会强制将赋值的结果从工作内存刷新到主内存

上面2点语义保证了被volatile修饰的数据在多线程中的可见性。

Unsafe中提供了和volatile语义一样的功能的方法，如下：

```
//设置给定对象的int值，使用volatile语义，即设置后立马更新到内存对其他线程可见
public native void putIntVolatile(Object o, long offset, int x);
//获得给定对象的指定偏移量offset的int值，使用volatile语义，总能获取到最新的int值。
public native int getIntVolatile(Object o, long offset);
```

putIntVolatile方法，2个参数：

- o：表示需要操作的对象
- offset：表示操作对象中的某个字段地址偏移量
- x：将offset对应的字段的值修改为x，并且立即刷新到主存中

调用这个方法，会强制将工作内存中修改的数据刷新到主内存中。

getIntVolatile方法，2个参数

- o：表示需要操作的对象
- offset：表示操作对象中的某个字段地址偏移量

每次调用这个方法都会强制从主内存读取值，将其复制到工作内存中使用。

其他的还有几个putXXXVolatile、getXXXVolatile方法和上面2个类似。

本文主要讲解这些内容，希望您能有所收获，谢谢。

Unsafe中Class相关方法

此部分主要提供Class和它的静态字段的操作相关方法，包含静态字段内存定位、定义类、定义匿名类、检验&确保初始化等。

```

//获取给定静态字段的内存地址偏移量，这个值对于给定的字段是唯一且固定不变的
public native long staticFieldOffset(Field f);
//获取一个静态类中给定字段的对象指针
public native Object staticFieldBase(Field f);
//判断是否需要初始化一个类，通常在获取一个类的静态属性的时候（因为一个类如果没初始化，它的静态属性也不会初始化）使用。当且仅当ensureClassInitialized方法不生效时返回false。
public native boolean shouldBeInitialized(Class<?> c);
//检测给定的类是否已经初始化。通常在获取一个类的静态属性的时候（因为一个类如果没初始化，它的静态属性也不会初始化）使用。
public native void ensureClassInitialized(Class<?> c);
//定义一个类，此方法会跳过JVM的所有安全检查，默认情况下，ClassLoader（类加载器）和
ProtectionDomain（保护域）实例来源于调用者
public native Class<?> defineClass(String name, byte[] b, int off, int len,
ClassLoader loader, ProtectionDomain protectionDomain);
//定义一个匿名类
public native Class<?> defineAnonymousClass(Class<?> hostClass, byte[] data,
Object[] cpPatches);

```

示例：staticFieldOffset、staticFieldBase、staticFieldBase

```

package com.itsoku.chat21;

import lombok.extern.slf4j.Slf4j;
import sun.misc.Unsafe;

import java.lang.reflect.Field;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
@Slf4j
public class Demo7 {

    static Unsafe unsafe;
    //静态属性
    private static Object v1;
    //实例属性
    private Object v2;

    static {
        //获取Unsafe对象
        try {
            Field field = unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws NoSuchFieldException {
        Field v1Field = Demo7.class.getDeclaredField("v1");
        Field v2Field = Demo7.class.getDeclaredField("v2");
    }
}

```

```

        System.out.println(unsafe.staticFieldOffset(v1Field));
        System.out.println(unsafe.objectFieldOffset(v2Field));

        System.out.println(unsafe.staticFieldBase(v1Field)==Demo7.class);
    }
}

```

输出：

```

112
12
true

```

可以看出 `staticFieldBase` 返回的就是 `Demo2` 的 class 对象。

示例：`shouldBeInitialized`、`ensureClassInitialized`

```

package com.itsoku.chat21;

import lombok.extern.slf4j.Slf4j;
import sun.misc.Unsafe;

import java.lang.reflect.Field;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo8 {

    static Unsafe unsafe;

    static {
        // 获取 Unsafe 对象
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static class C1 {
        private static int count;

        static {
            count = 10;
            System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ", C1 static init.");
        }
    }

    static class C2 {
        private static int count;
    }
}

```

```

    static {
        count = 11;
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ", c2 static init.");
    }
}

public static void main(String[] args) throws NoSuchFieldException {
    //判断C1类是需要初始化，如果已经初始化了，会返回false，如果此类没有被初始化过，返回true
    if (unsafe.shouldBeInitialized(c1.class)) {
        System.out.println("c1需要进行初始化");
        //对c1进行初始化
        unsafe.ensureClassInitialized(c1.class);
    }

    System.out.println(c2.count);
    System.out.println(unsafe.shouldBeInitialized(c1.class));
}
}

```

输出：

```

c1需要进行初始化
1565069660679,main, c1 static init.
1565069660680,main, c2 static init.
11
false

```

代码中C1未被初始化过，所以 `unsafe.shouldBeInitialized(c1.class)` 返回true，然后调用 `unsafe.ensureClassInitialized(c1.class)` 进行初始化。

代码中执行 `c2.count` 会触发C2进行初始化，所以 `shouldBeInitialized(c1.class)` 返回false

对象操作的其他方法

```

//返回对象成员属性在内存地址相对于此对象的内存地址的偏移量
public native long objectFieldOffset(Field f);
//获得给定对象的指定地址偏移量的值，与此类似操作还有：getInt, getDouble, getLong, getChar等
public native Object getObject(Object o, long offset);
//给定对象的指定地址偏移量设值，与此类似操作还有：putInt, putDouble, putLong, putChar等
public native void putObject(Object o, long offset, Object x);
//从对象的指定偏移量处获取变量的引用，使用volatile的加载语义
public native Object getObjectVolatile(Object o, long offset);
//存储变量的引用到对象的指定的偏移量处，使用volatile的存储语义
public native void putObjectVolatile(Object o, long offset, Object x);
//有序、延迟版本的putObjectVolatile方法，不保证值的改变被其他线程立即看到，只有在field被volatile修饰符修饰时有效
public native void putOrderedObject(Object o, long offset, Object x);
//绕过构造方法、初始化代码来创建对象
public native Object allocateInstance(Class<?> cls) throws
InstantiationException;

```

`getObject` 相当于获取对象中字段的值，`putObject` 相当于给字段赋值，有兴趣的可以自己写个例子看看效果。

绕过构造方法创建对象

介绍一下 `allocateInstance`，这个方法可以绕过构造方法来创建对象，示例代码如下：

```
package com.itsoku.chat21;

import sun.misc.Unsafe;

import java.lang.reflect.Field;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo9 {

    static Unsafe unsafe;

    static {
        //获取Unsafe对象
        try {
            Field field = unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            unsafe = (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static class C1 {
        private String name;

        private C1() {
            System.out.println("C1 default constructor!");
        }

        private C1(String name) {
            this.name = name;
            System.out.println("C1 有参 constructor!");
        }
    }

    public static void main(String[] args) throws InstantiationException {
        System.out.println(unsafe.allocateInstance(C1.class));
    }
}
```

输出：

```
com.itsoku.chat21.Demo9$C1@782830e
```

看一下类C1中有两个构造方法，都是private的，通过new、反射的方式都无法创建对象。但是可以通过Unsafe的`allocateInstance`方法绕过构造函数来创建C1的实例，输出的结果中可以看出创建成功了，并且没有调用构造方法。

典型应用

- **常规对象实例化方式**: 我们通常所用到的创建对象的方式，从本质上讲，都是通过new机制来实现对象的创建。但是，new机制有个特点就是当类只提供有参的构造函数且无显示声明无参构造函数时，则必须使用有参构造函数进行对象构造，而使用有参构造函数时，必须传递相应个数的参数才能完成对象实例化。
- **非常规的实例化方式**: 而Unsafe中提供allocateInstance方法，仅通过Class对象就可以创建此类的实例对象，而且不需要调用其构造函数、初始化代码、JVM安全检查等。它抑制修饰符检测，也就是即使构造器是private修饰的也能通过此方法实例化，只需提类对象即可创建相应的对象。由于这种特性，allocateInstance在java.lang.invoke、Objenesis（提供绕过类构造器的对象生成方式）、Gson（反序列化时用到）中都有相应的应用。

数组相关的一些方法

这部分主要介绍与数据操作相关的arrayBaseOffset与arrayIndexScale这两个方法，两者配合起来使用，即可定位数组中每个元素在内存中的位置。

```
//返回数组中第一个元素的偏移地址  
public native int arrayBaseOffset(Class<?> arrayClass);  
//返回数组中一个元素占用的大小  
public native int arrayIndexScale(Class<?> arrayClass);
```

这两个与数据操作相关的方法，在java.util.concurrent.atomic包下的AtomicIntegerArray（可以实现对Integer数组中每个元素的原子性操作）中有典型的应用，如下图AtomicIntegerArray源码所示，通过Unsafe的arrayBaseOffset、arrayIndexScale分别获取数组首元素的偏移地址base及单个元素大小因子scale。后续相关原子性操作，均依赖于这两个值进行数组中元素的定位，如下图二所示的getAndAdd方法即通过checkedByteOffset方法获取某数组元素的偏移地址，而后通过CAS实现原子性操作。

数组元素定位：

Unsafe类中有很多以BASE_OFFSET结尾的常量，比如ARRAY_INT_BASE_OFFSET，ARRAY_BYTE_BASE_OFFSET等，这些常量值是通过arrayBaseOffset方法得到的。arrayBaseOffset方法是一个本地方法，可以获取数组第一个元素的偏移地址。Unsafe类中还有很多以INDEX_SCALE结尾的常量，比如ARRAY_INT_INDEX_SCALE，ARRAY_BYTE_INDEX_SCALE等，这些常量值是通过arrayIndexScale方法得到的。arrayIndexScale方法也是一个本地方法，可以获取数组的转换因子，也就是数组中元素的增量地址。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。

内存屏障相关操作

在Java 8中引入，用于定义内存屏障（也称内存栅栏，内存栅障，屏障指令等，是一类同步屏障指令，是CPU或编译器在对内存随机访问的操作中的一个同步点，使得此点之前的所有读写操作都执行后才可以开始执行此点之后的操作），避免代码重排序。

```
//内存屏障，禁止load操作重排序。屏障前的load操作不能被重排序到屏障后，屏障后的load操作不能被重排序到屏障前
public native void loadFence();
//内存屏障，禁止store操作重排序。屏障前的store操作不能被重排序到屏障后，屏障后的store操作不能被重排序到屏障前
public native void storeFence();
//内存屏障，禁止load、store操作重排序
public native void fullFence();
```

Unsafe相关的就介绍这么多！

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第23篇：JUC中的原子操作类

本文主要内容

1. JUC中的原子类介绍
2. 介绍基本类型原子类
3. 介绍数组类型原子类
4. 介绍引用类型原子类
5. 介绍对象属性修改相关原子类

预备知识

JUC中的原子类都是依靠volatile、CAS、Unsafe类配合来实现的，需要了解的请移步：

[volatile与Java内存模型](#)

[java中的CAS](#)

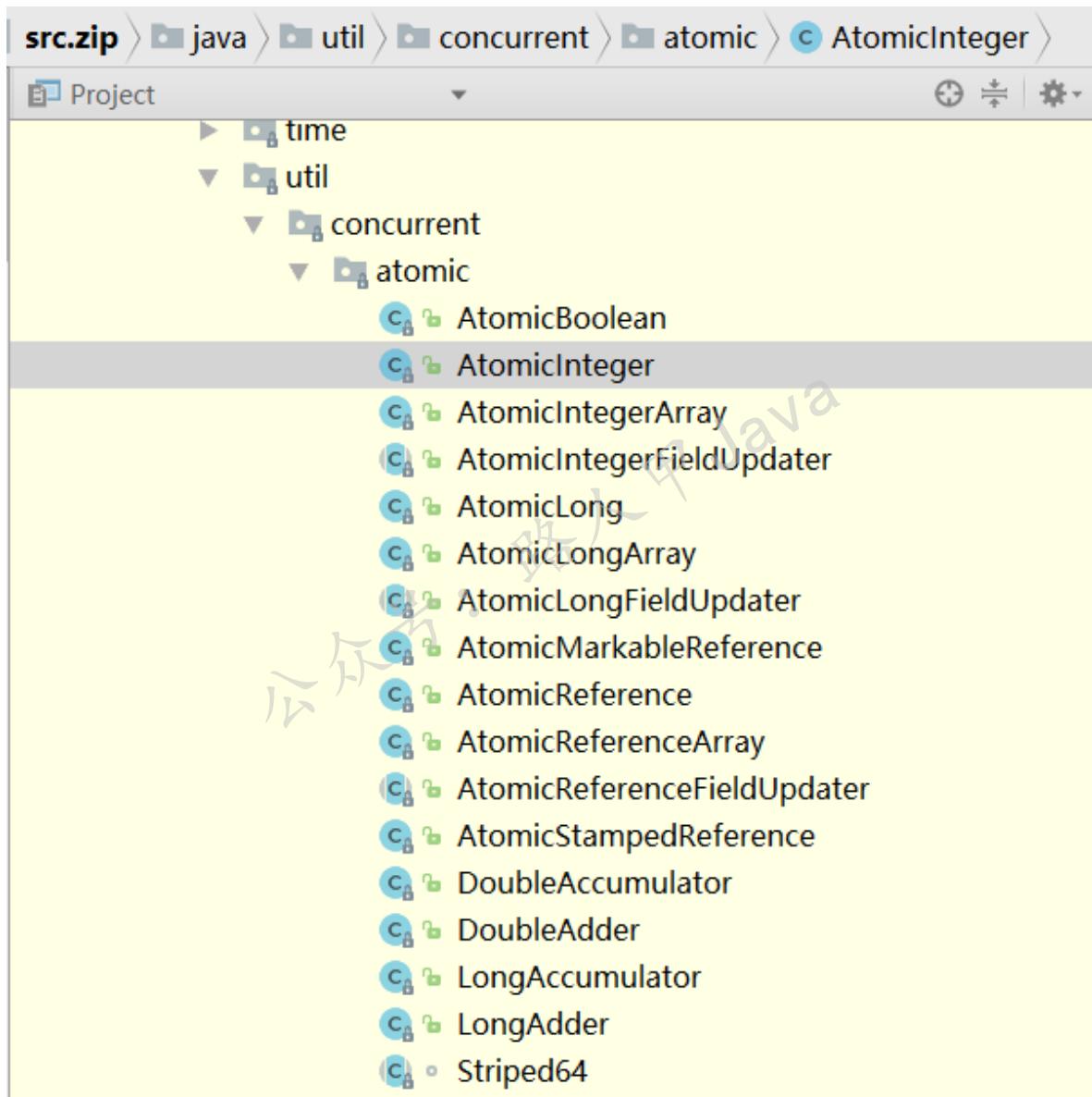
[JUC底层工具类Unsafe](#)

JUC中原子类介绍

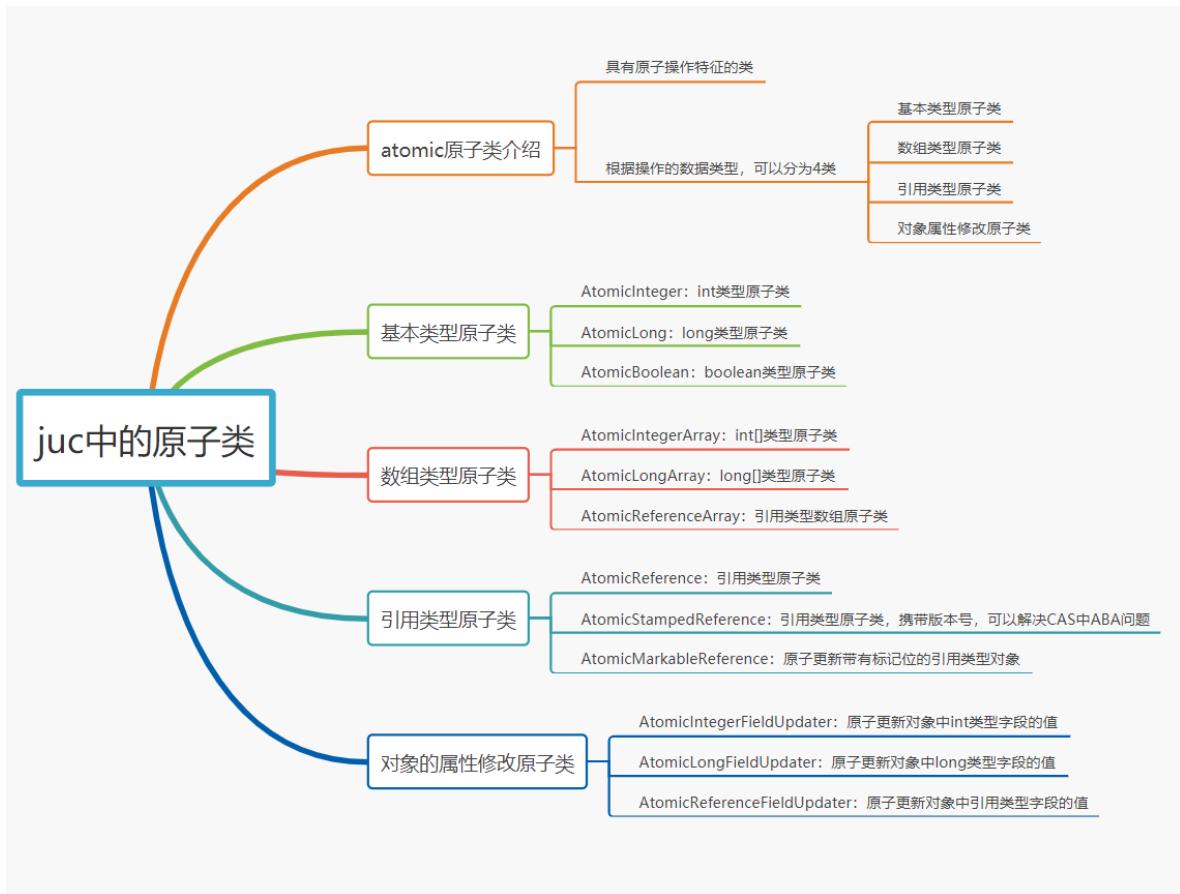
什么是原子操作？

atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里 **atomic** 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰，所以，所谓原子类说简单点就是具有原子操作特征的类，原子操作类提供了一些修改数据的方法，这些方法都是原子操作的，在多线程情况下可以确保被修改数据的正确性。

JUC中对原子操作提供了强大的支持，这些类位于java.util.concurrent.atomic包中，如下图：



JUC中原子类思维导图



基本类型原子类

使用原子的方式更新基本类型

- AtomicInteger: int类型原子类
- AtomicLong: long类型原子类
- AtomicBoolean : boolean类型原子类

上面三个类提供的方法几乎相同，这里以 AtomicInteger 为例子来介绍。

AtomicInteger 类常用方法

```

public final int get() //获取当前的值
public final int getAndSet(int newValue)//获取当前的值，并设置新的值
public final int getAndIncrement()//获取当前的值，并自增
public final int getAndDecrement() //获取当前的值，并自减
public final int getAndAdd(int delta) //获取当前的值，并加上预期的值
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式
将该值设置为输入值(update)
public final void lazySet(int newValue)//最终设置为newValue, 使用 lazySet 设置之后可
能导致其他线程在之后的一小段时间内还是可以读到旧的值。

```

部分源码

```

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

static {
    try {
        valueOffset = unsafe.objectFieldOffset
            (AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) { throw new Error(ex); }
}

private volatile int value;

```

2个关键字段说明：**value**: 使用volatile修饰，可以确保value在多线程中的可见性。
valueOffset: value属性在AtomicInteger中的偏移量，通过这个偏移量可以快速定位到value字段，这个是实现AtomicInteger的关键。

getAndIncrement源码：

```

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

```

内部调用的是Unsafe类中的**getAndAddInt**方法，我们看一下**getAndAddInt**源码：

```

public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

    return var5;
}

```

说明： this.getIntVolatile: 可以确保从主内存中获取变量最新的值。

compareAndSwapInt: CAS操作，CAS的原理是拿期望的值和原本的值作比较，如果相同则更新成新的值，可以确保在多线程情况下只有一个线程会操作成功，不成功的返回false。

上面有个do-while循环，compareAndSwapInt返回false之后，会再次从主内存中获取变量的值，继续做CAS操作，直到成功为止。

getAndAddInt操作相当于线程安全的count++操作，如同： synchronize(lock){ count++; }
count++操作实际上被拆分为3步骤执行：

1. 获取count的值，记做A: A=count
2. 将A的值+1，得到B: B = A+1
3. 让B赋值给count: count = B 多线程情况下会出现线程安全的问题，导致数据不准确。

synchronize的方式会导致占时无法获取锁的线程处于阻塞状态，性能比较低。CAS的性能比synchronize要快很多。

示例

使用AtomicInteger实现网站访问量计数器功能，模拟100人同时访问网站，每个人访问10次，代码如下：

```

package com.itsoku.chat23;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    //访问次数
    static AtomicInteger count = new AtomicInteger();

    //模拟访问一次
    public static void request() throws InterruptedException {
        //模拟耗时5毫秒
        TimeUnit.MILLISECONDS.sleep(5);
        //对count原子+1
        count.incrementAndGet();
    }

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        int threadSize = 100;
        CountDownLatch countDownLatch = new CountDownLatch(threadSize);
        for (int i = 0; i < threadSize; i++) {
            Thread thread = new Thread(() -> {
                try {
                    for (int j = 0; j < 10; j++) {
                        request();
                    }
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } finally {
                    countDownLatch.countDown();
                }
            });
            thread.start();
        }

        countDownLatch.await();
        long endTime = System.currentTimeMillis();
        System.out.println(Thread.currentThread().getName() + "，耗时：" + (endTime - startTime) + ", count=" + count);
    }
}

```

输出：

```
main, 耗时: 158, count=1000
```

通过输出中可以看出 `incrementAndGet` 在多线程情况下能确保数据的正确性。

数组类型原子类介绍

使用原子的方式更新数组里的某个元素，可以确保修改数组中数据的线程安全性。

- AtomicIntegerArray：整形数组原子操作类
- AtomicLongArray：长整形数组原子操作类
- AtomicReferenceArray：引用类型数组原子操作类

上面三个类提供的方法几乎相同，所以我们这里以 AtomicIntegerArray 为例子来介绍。

AtomicIntegerArray 类常用方法

```
public final int get(int i) //获取 index=i 位置元素的值
public final int getAndSet(int i, int newValue)//返回 index=i 位置的当前的值，并将其设置为newValue
public final int getAndIncrement(int i)//获取 index=i 位置元素的值，并让该位置的元素自增
public final int getAndDecrement(int i) //获取 index=i 位置元素的值，并让该位置的元素自减
public final int getAndAdd(int delta) //获取 index=i 位置元素的值，并加上预期的值
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将 index=i 位置的元素值设置为输入值(update)
public final void lazySet(int i, int newValue)//最终 将index=i 位置的元素设置为 newValue，使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

示例

统计网站页面访问量，假设网站有10个页面，现在模拟100个人并行访问每个页面10次，然后将每个页面访问量输出，应该每个页面都是1000次，代码如下：

```
package com.itsoku.chat23;

import java.util.Arrays;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.atomic.AtomicIntegerArray;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo2 {

    static AtomicIntegerArray pageRequest = new AtomicIntegerArray(new int[10]);

    /**
     * 模拟访问一次
     *
     * @param page 访问第几个页面
     * @throws InterruptedException
     */
    public static void request(int page) throws InterruptedException {
        //模拟耗时5毫秒
        TimeUnit.MILLISECONDS.sleep(5);
        //pageCountIndex为pageCount数组的下标，表示页面对应数组中的位置
        int pageCountIndex = page - 1;
        pageRequest.incrementAndGet(pageCountIndex);
    }
}
```

```

public static void main(String[] args) throws InterruptedException {
    long startTime = System.currentTimeMillis();
    int threadSize = 100;
    CountDownLatch countDownLatch = new CountDownLatch(threadSize);
    for (int i = 0; i < threadSize; i++) {
        Thread thread = new Thread(() -> {
            try {
                for (int page = 1; page <= 10; page++) {
                    for (int j = 0; j < 10; j++) {
                        request(page);
                    }
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                countDownLatch.countDown();
            }
        });
        thread.start();
    }

    countDownLatch.await();
    long endTime = System.currentTimeMillis();
    System.out.println(Thread.currentThread().getName() + ", 耗时: " +
(endTime - startTime));

    for (int pageIndex = 0; pageIndex < 10; pageIndex++) {
        System.out.println("第" + (pageIndex + 1) + "个页面访问次数为" +
pageRequest.get(pageIndex));
    }
}
}

```

输出：

```

main, 耗时: 635
第1个页面访问次数为1000
第2个页面访问次数为1000
第3个页面访问次数为1000
第4个页面访问次数为1000
第5个页面访问次数为1000
第6个页面访问次数为1000
第7个页面访问次数为1000
第8个页面访问次数为1000
第9个页面访问次数为1000
第10个页面访问次数为1000

```

说明：

代码中将10个面的访问量放在了一个int类型的数组中，数组大小为10，然后通过 `AtomicIntegerArray` 来操作数组中的每个元素，可以确保操作数据的原子性，每次访问会调用 `incrementAndGet`，此方法需要传入数组的下标，然后对指定的元素做原子+1操作。输出结果都是1000，可以看出对于数组中元素的并发修改是线程安全的。如果线程不安全，则部分数据可能会小于1000。

其他的一些方法可以自行操作一下，都非常简单。

引用类型原子类介绍

基本类型原子类只能更新一个变量，如果需要原子更新多个变量，需要使用 引用类型原子类。

- **AtomicReference**: 引用类型原子类
- **AtomicStampedReference**: 原子更新引用类型里的字段原子类
- **AtomicMarkableReference** : 原子更新带有标记位的引用类型

AtomicReference 和 **AtomicInteger** 非常类似，不同之处在于 **AtomicInteger**是对整数的封装，而 **AtomicReference**则是对应普通的对象引用，它可以确保你在修改对象引用时的线程安全性。在介绍 **AtomicReference**的同时，我们先来了解一个有关原子操作逻辑上的不足。

ABA问题

之前我们说过，线程判断被修改对象是否可以正确写入的条件是对象的当前值和期望值是否一致。这个逻辑从一般意义上来说是正确的，但是可能出现一个小小的例外，就是当你获得当前数据后，在准备修改为新值钱，对象的值被其他线程连续修改了两次，而经过这2次修改后，对象的值又恢复为旧值，这样，当前线程就无法正确判断这个对象究竟是否被修改过，这就是所谓的ABA问题，可能会引发一些问题。

举个例子

有一家蛋糕店，为了挽留客户，决定为贵宾卡客户一次性赠送20元，刺激客户充值和消费，但条件是，每一位客户只能被赠送一次，现在我们用 **AtomicReference** 来实现这个功能，代码如下：

```
package com.itsoku.chat22;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo3 {
    //账户原始余额
    static int accountMoney = 19;
    //用于对账户余额做原子操作
    static AtomicReference<Integer> money = new AtomicReference<>(accountMoney);

    /**
     * 模拟2个线程同时更新后台数据库，为用户充值
     */
    static void recharge() {
        for (int i = 0; i < 2; i++) {
            new Thread(() -> {
                for (int j = 0; j < 5; j++) {
                    Integer m = money.get();
                    if (m == accountMoney) {
                        if (money.compareAndSet(m, m + 20)) {
                            System.out.println("当前余额：" + m + "，充值20
元成功，余额：" + money.get() + "元");
                        }
                    }
                }
            })
            .start();
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```

        TimeUnit.MILLISECONDS.sleep(100);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}).start();
}

/**
 * 模拟用户消费
 */
static void consume() throws InterruptedException {
    for (int i = 0; i < 5; i++) {
        Integer m = money.get();
        if (m > 20) {
            if (money.compareAndSet(m, m - 20)) {
                System.out.println("当前余额: " + m + ", 大于10, 成功消费10元, 余额: " + money.get() + "元");
            }
        }
        //休眠50ms
        TimeUnit.MILLISECONDS.sleep(50);
    }
}

public static void main(String[] args) throws InterruptedException {
    recharge();
    consume();
}
}

```

输出：

```

当前余额: 19, 小于20, 充值20元成功, 余额: 39元
当前余额: 39, 大于10, 成功消费10元, 余额: 19元
当前余额: 19, 小于20, 充值20元成功, 余额: 39元
当前余额: 39, 大于10, 成功消费10元, 余额: 19元
当前余额: 19, 小于20, 充值20元成功, 余额: 39元
当前余额: 39, 大于10, 成功消费10元, 余额: 19元
当前余额: 19, 小于20, 充值20元成功, 余额: 39元

```

从输出中可以看到，这个账户被先后反复多次充值。其原因是账户余额被反复修改，修改后的值和原有的数值19一样，使得CAS操作无法正确判断当前数据是否被修改过（是否被加过20）。虽然这种情况出现的概率不大，但是依然是有可能出现的，因此，当业务上确实可能出现这种情况时，我们必须多加防范。JDK也为我们考虑到了这种情况，使用`AtomicStampedReference`可以很好地解决这个问题。

使用`AtomicStampedReference`解决ABA的问题

`AtomicReference`无法解决上述问题的根本原因是，对象在被修改过程中丢失了状态信息，比如充值20元的时候，需要同时标记一个状态，用来标注用户被充值过。因此我们只要能够记录对象在修改过程中的状态值，就可以很好地解决对象被反复修改导致线程无法正确判断对象状态的问题。

AtomicStampedReference 正是这么做的，他内部不仅维护了对象的值，还维护了一个时间戳（我们这里把他称为时间戳，实际上它可以使用任何一个整形来表示状态值），当AtomicStampedReference对应的数值被修改时，除了更新数据本身外，还必须要更新时间戳。当AtomicStampedReference设置对象值时，对象值及时间戳都必须满足期望值，写入才会成功。因此，即使对象值被反复读写，写回原值，只要时间戳发生变量，就能防止不恰当的写入。

AtomicStampedReference 的几个Api在 AtomicReference 的基础上新增了有关时间戳的信息。

```
//比较设置，参数依次为：期望值、写入新值、期望时间戳、新时间戳
public boolean compareAndSet(V expectedReference, V newReference, int
expectedStamp, int newStamp);
//获得当前对象引用
public V getReference();
//获得当前时间戳
public int getStamp();
//设置当前对象引用和时间戳
public void set(V newReference, int newStamp);
```

现在我们使用 AtomicStampedReference 来修改一下上面充值的问题，代码如下：

```
package com.itsoku.chat22;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReference;
import java.util.concurrent.atomic.AtomicStampedReference;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {
    //账户原始余额
    static int accountMoney = 19;
    //用于对账户余额做原子操作
    static AtomicStampedReference<Integer> money = new AtomicStampedReference<>(
accountMoney, 0);

    /**
     * 模拟2个线程同时更新后台数据库，为用户充值
     */
    static void recharge() {
        for (int i = 0; i < 2; i++) {
            int stamp = money.getStamp();
            new Thread(() -> {
                for (int j = 0; j < 50; j++) {
                    Integer m = moneygetReference();
                    if (m == accountMoney) {
                        if (money.compareAndSet(m, m + 20, stamp, stamp + 1)) {
                            System.out.println("当前时间戳：" + money.getStamp() +
", 当前余额：" + m + "， 小于20， 充值20元成功， 余额：" + money.getReference() + "元");
                        }
                    }
                }
                //休眠100ms
                try {
                    TimeUnit.MILLISECONDS.sleep(100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            })
        }
    }
}
```

```

        }
    }
}).start();
}

/**
 * 模拟用户消费
 */
static void consume() throws InterruptedException {
    for (int i = 0; i < 50; i++) {
        Integer m = money.getReference();
        int stamp = money.getStamp();
        if (m > 20) {
            if (money.compareAndSet(m, m - 20, stamp, stamp + 1)) {
                System.out.println("当前时间戳: " + money.getStamp() + ", 当前余额: " + m + ", 大于10, 成功消费10元, 余额: " + money.getReference() + "元");
            }
        }
        //休眠50ms
        TimeUnit.MILLISECONDS.sleep(50);
    }
}

public static void main(String[] args) throws InterruptedException {
    recharge();
    consume();
}
}

```

输出：

```

当前时间戳: 1, 当前余额: 19, 小于20, 充值20元成功, 余额: 39元
当前时间戳: 2, 当前余额: 39, 大于10, 成功消费10元, 余额: 19元

```

结果正常了。

关于这个时间戳的，在数据库修改数据中也有类似的用法，比如2个编辑同时编辑一篇文章，同时提交，只允许一个用户提交成功，提示另外一个用户：博客已被其他人修改，如何实现呢？

博客表：t_blog (id,content,stamp)，stamp默认值为0，每次更新+1

A、B 二个编辑同时对一篇文章进行编辑，stamp都为0，当点击提交的时候，将stamp和id作为条件更新博客内容，执行的sql如下：

```

update t_blog set content = 更新的内容,stamp = stamp+1 where id = 博客id and stamp = 0;

```

这条update会返回影响的行数，只有一个会返回1，表示更新成功，另外一个提交者返回0，表示需要修改的数据已经不满足条件了，被其他用户给修改了。这种修改数据的方式也叫乐观锁。

对象的属性修改原子类介绍

如果需要原子更新某个类里的某个字段时，需要用到对象的属性修改原子类。

- AtomicIntegerFieldUpdater: 原子更新整形字段的值
- AtomicLongFieldUpdater: 原子更新长整形字段的值
- AtomicReferenceFieldUpdater : 原子更新应用类型字段的值

要想原子地更新对象的属性需要两步:

1. 第一步, 因为对象的属性修改类型原子类都是抽象类, 所以每次使用都必须使用静态方法 newUpdater() 创建一个更新器, 并且需要设置想要更新的类和属性。
2. 第二步, 更新的对象属性必须使用 public volatile 修饰符。

上面三个类提供的方法几乎相同, 所以我们这里以 `AtomicReferenceFieldUpdater` 为例子来介绍。

调用 `AtomicReferenceFieldUpdater` 静态方法 `newUpdater` 创建 `AtomicReferenceFieldUpdater` 对象

```
public static <U, W> AtomicReferenceFieldUpdater<U, W> newUpdater(Class<U>
tclass, Class<W> vclass, String fieldName)
```

说明:

三个参数

`tclass`: 需要操作的字段所在的类 `vclass`: 操作字段的类型 `fieldName`: 字段名称

示例

多线程并发调用一个类的初始化方法, 如果未被初始化过, 将执行初始化工作, 要求只能初始化一次

代码如下:

```
package com.itsoku.chat22;

import com.sun.org.apache.xpath.internal.operations.Bool;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicReferenceFieldUpdater;

/**
 * 跟着阿里P7学并发, 微信公众号: javacode2018
 */
public class Demo5 {

    static Demo5 demo5 = new Demo5();
    //isInit用来标注是否被初始化过
    volatile Boolean isInit = Boolean.FALSE;
    AtomicReferenceFieldUpdater<Demo5, Boolean> updater =
    AtomicReferenceFieldUpdater.newUpdater(Demo5.class, Boolean.class, "isInit");

    /**
     * 模拟初始化工作
     *
     * @throws InterruptedException
     */
    public void init() throws InterruptedException {
        //isInit为false的时候, 才进行初始化, 并将isInit采用原子操作置为true
        if (updater.compareAndSet(demo5, Boolean.FALSE, Boolean.TRUE)) {
```

```

        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ", 开始初始化!");
        //模拟休眠3秒
        TimeUnit.SECONDS.sleep(3);
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ", 初始化完毕!");
    } else {
        System.out.println(System.currentTimeMillis() + "," +
Thread.currentThread().getName() + ", 有其他线程已经执行了初始化!");
    }
}

public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
        new Thread(() -> {
            try {
                demo5.init();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}

```

输出：

```

1565159962098,Thread-0, 开始初始化!
1565159962098,Thread-3, 有其他线程已经执行了初始化!
1565159962098,Thread-4, 有其他线程已经执行了初始化!
1565159962098,Thread-2, 有其他线程已经执行了初始化!
1565159962098,Thread-1, 有其他线程已经执行了初始化!
1565159965100,Thread-0, 初始化完毕!

```

说明：

1. `isInit`属性必须要`volatile`修饰，可以确保变量的可见性
2. 可以看出多线程同时执行`init()`方法，只有一个线程执行了初始化的操作，其他线程跳过了。多个线程同时到达`updater.compareAndSet`，只有一个会成功。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第24篇：java中的ThreadLocal、InheritableThreadLocal

本文内容

1. 需要解决的问题
2. 介绍ThreadLocal
3. 介绍InheritableThreadLocal

需要解决的问题

我们还是以解决问题的方式来引出 ThreadLocal、InheritableThreadLocal，这样印象会深刻一些。

目前java开发web系统一般有3层，controller、service、dao，请求到达controller，controller调用service，service调用dao，然后进行处理。

我们写一个简单的例子，有3个方法分别模拟controller、service、dao。代码如下：

```
package com.itsoku.chat24;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {

    static AtomicInteger threadIndex = new AtomicInteger(1);
    //创建处理请求的线程池子
```

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

```

static ThreadPoolExecutor disposeRequestExecutor = new ThreadPoolExecutor(3,
    3,
    60,
    TimeUnit.SECONDS,
    new LinkedBlockingDeque<>(),
    r -> {
        Thread thread = new Thread(r);
        thread.setName("disposeRequestThread-" +
threadIndex.getAndIncrement());
        return thread;
    });
}

//记录日志
public static void log(String msg) {
    StackTraceElement stack[] = (new Throwable()).getStackTrace();
    System.out.println("****" + System.currentTimeMillis() + ",[线程:" +
Thread.currentThread().getName() + "]," + stack[1] + ":" + msg);
}

//模拟controller
public static void controller(List<String> dataList) {
    log("接受请求");
    service(dataList);
}

//模拟service
public static void service(List<String> dataList) {
    log("执行业务");
    dao(dataList);
}

//模拟dao
public static void dao(List<String> dataList) {
    log("执行数据库操作");
    //模拟插入数据
    for (String s : dataList) {
        log("插入数据" + s + "成功");
    }
}

public static void main(String[] args) {
    //需要插入的数据
    List<String> dataList = new ArrayList<>();
    for (int i = 0; i < 3; i++) {
        dataList.add("数据" + i);
    }

    //模拟5个请求
    int requestCount = 5;
    for (int i = 0; i < requestCount; i++) {
        disposeRequestExecutor.execute(() -> {
            controller(dataList);
        });
    }

    disposeRequestExecutor.shutdown();
}
}

```

运行结果：

```
****1565338891286,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.controller(Demo1.java:36):接受请求
****1565338891286,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.controller(Demo1.java:36):接受请求
****1565338891287,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.service(Demo1.java:42):执行业务
****1565338891287,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.service(Demo1.java:42):执行业务
****1565338891287,[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo1.controller(Demo1.java:36):接受请求
****1565338891287,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:48):执行数据库操作
****1565338891287,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据0成功
****1565338891287,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据1成功
****1565338891287,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:48):执行数据库操作
****1565338891287,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据2成功
****1565338891287,[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo1.service(Demo1.java:42):执行业务
****1565338891288,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.controller(Demo1.java:36):接受请求
****1565338891287,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据0成功
****1565338891288,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.service(Demo1.java:42):执行业务
****1565338891288,[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo1.dao(Demo1.java:48):执行数据库操作
****1565338891288,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:48):执行数据库操作
****1565338891288,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据1成功
****1565338891288,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据0成功
****1565338891288,[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据0成功
****1565338891288,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据1成功
****1565338891288,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据2成功
****1565338891288,[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据2成功
****1565338891288,[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据1成功
****1565338891288,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.controller(Demo1.java:36):接受请求
****1565338891288,[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据2成功
****1565338891288,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.service(Demo1.java:42):执行业务
****1565338891289,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:48):执行数据库操作
```

```
****1565338891289,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据0成功
****1565338891289,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据1成功
****1565338891289,[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo1.dao(Demo1.java:51):插入数据数据2成功
```

代码中调用controller、service、dao 3个方法时，来模拟处理一个请求。main方法中循环了5次模拟发起5次请求，然后交给线程池去处理请求，dao中模拟循环插入传入的dataList数据。

问题来了：开发者想看一下哪些地方耗时比较多，想通过日志来分析耗时情况，想追踪某个请求的完整日志，怎么搞？

上面的请求采用线程池的方式处理的，多个请求可能会被一个线程处理，通过日志很难看出那些日志是同一个请求，我们能不能给请求加一个唯一标志，日志中输出这个唯一标志，当然可以。

如果我们的代码就只有上面示例这么简单，我想还是很容易的，上面就3个方法，给每个方法加个traceId参数，log方法也加个traceId参数，就解决了，代码如下：

```
package com.itsoku.chat24;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo2 {

    static AtomicInteger threadIndex = new AtomicInteger(1);
    //创建处理请求的线程池子
    static ThreadPoolExecutor disposeRequestExecutor = new ThreadPoolExecutor(3,
        3,
        60,
        TimeUnit.SECONDS,
        new LinkedBlockingDeque<>(),
        r -> {
            Thread thread = new Thread(r);
            thread.setName("disposeRequestThread-" +
threadIndex.getAndIncrement());
            return thread;
        });

    //记录日志
    public static void log(String msg, String traceId) {
        StackTraceElement stack[] = (new Throwable()).getStackTrace();
        System.out.println("****" + System.currentTimeMillis() + "[traceId:" +
traceId + "],[线程:" + Thread.currentThread().getName() + "]," + stack[1] + ":" +
msg);
    }

    //模拟controller
    public static void controller(List<String> dataList, String traceId) {
```

```

        log("接受请求", traceId);
        service(dataList, traceId);
    }

    //模拟service
    public static void service(List<String> dataList, String traceId) {
        log("执行业务", traceId);
        dao(dataList, traceId);
    }

    //模拟dao
    public static void dao(List<String> dataList, String traceId) {
        log("执行数据库操作", traceId);
        //模拟插入数据
        for (String s : dataList) {
            log("插入数据" + s + "成功", traceId);
        }
    }

    public static void main(String[] args) {
        //需要插入的数据
        List<String> dataList = new ArrayList<>();
        for (int i = 0; i < 3; i++) {
            dataList.add("数据" + i);
        }

        //模拟5个请求
        int requestCount = 5;
        for (int i = 0; i < requestCount; i++) {
            String traceId = String.valueOf(i);
            disposeRequestExecutor.execute(() -> {
                controller(dataList, traceId);
            });
        }

        disposeRequestExecutor.shutdown();
    }
}

```

输出：

```

*****1565339559773[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.controller(Demo2.java:36):接受请求
*****1565339559773[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.controller(Demo2.java:36):接受请求
*****1565339559773[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.controller(Demo2.java:36):接受请求
*****1565339559774[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.service(Demo2.java:42):执行业务
*****1565339559774[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.service(Demo2.java:42):执行业务
*****1565339559774[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:48):执行数据库操作
*****1565339559774[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.service(Demo2.java:42):执行业务
*****1565339559774[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据0成功

```

```
****1565339559774[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:48):执行数据库操作
****1565339559774[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据1成功
****1565339559774[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.dao(Demo2.java:48):执行数据库操作
****1565339559774[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据2成功
****1565339559774[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据0成功
****1565339559775[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.controller(Demo2.java:36):接受请求
****1565339559775[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据0成功
****1565339559775[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.service(Demo2.java:42):执行业务
****1565339559775[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据1成功
****1565339559775[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:48):执行数据库操作
****1565339559775[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据1成功
****1565339559775[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据0成功
****1565339559775[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据2成功
****1565339559775[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据1成功
****1565339559775[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据2成功
****1565339559775[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据0成功
****1565339559775[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据1成功
****1565339559775[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo2.controller(Demo2.java:36):接受请求
****1565339559776[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.service(Demo2.java:42):执行业务
****1565339559776[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:48):执行数据库操作
****1565339559776[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据0成功
****1565339559776[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据1成功
****1565339559776[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo2.dao(Demo2.java:51):插入数据数据2成功
```

上面我们通过修改代码的方式，把问题解决了，但前提是你们的系统都像上面这么简单，功能很少，需要改的代码很少，可以这么去改。但事与愿违，我们的系统一般功能都是比较多的，如果我们都一个个去改，岂不是要疯掉，改代码还涉及到重新测试，风险也不可控。那有什么好办法么？

ThreadLocal

还是拿上面的问题，我们来分析一下，每个请求都是由一个线程处理的，线程就相当于一个人一样，每个请求相当于一个任务，任务来了，人来处理，处理完毕之后，再处理下一个请求任务。人身上是不是有很多口袋，人刚开始准备处理任务的时候，我们把任务的编号放在处理者的口袋中，然后处理中一路携带者，处理过程中如果需要用到这个编号，直接从口袋中获取就可以了。那么刚好java中线程设计的

时候也考虑到了这些问题，Thread对象中就有很多口袋，用来放东西。Thread类中有这么一个变量：

```
ThreadLocal.ThreadLocalMap threadLocals = null;
```

这个就是用来操作Thread中所有口袋的东西，`ThreadLocalMap` 源码中有一个数组（有兴趣的可以去看一下源码），对应处理器身上很多口袋一样，数组中的每个元素对应一个口袋。

如何来操作Thread中的这些口袋呢，java为我们提供了一个类 `ThreadLocal`，`ThreadLocal`对象用来操作Thread中的某一个口袋，可以向这个口袋中放东西、获取里面的东西、清除里面的东西，这个口袋一次性只能放一个东西，重复放东西会将里面已经存在的东西覆盖掉。

常用的3个方法：

```
//向Thread中某个口袋中放东西
public void set(T value);
//获取这个口袋中目前放的东西
public T get();
//清空这个口袋中放的东西
public void remove()
```

我们使用`ThreadLocal`来改造一下上面的代码，如下：

```
package com.itsoku.chat24;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo3 {

    //创建一个操作Thread中存放请求任务追踪id口袋的对象
    static ThreadLocal<String> traceIdKD = new ThreadLocal<>();

    static AtomicInteger threadIndex = new AtomicInteger(1);
    //创建处理请求的线程池子
    static ThreadPoolExecutor disposeRequestExecutor = new ThreadPoolExecutor(3,
        3,
        60,
        TimeUnit.SECONDS,
        new LinkedBlockingDeque<>(),
        r -> {
            Thread thread = new Thread(r);
            thread.setName("disposeRequestThread-" +
                threadIndex.getAndIncrement());
            return thread;
        });

    //记录日志
    public static void log(String msg) {
        StackTraceElement stack[] = (new Throwable()).getStackTrace();
```

```

    //获取当前线程存放traceId口袋中的内容
    String traceId = traceIdKD.get();
    System.out.println("****" + System.currentTimeMillis() + "[traceId:" +
traceId + "],[线程:" + Thread.currentThread().getName() + "]," + stack[1] + ":" +
msg);
}

//模拟controller
public static void controller(List<String> dataList) {
    log("接受请求");
    service(dataList);
}

//模拟service
public static void service(List<String> dataList) {
    log("执行业务");
    dao(dataList);
}

//模拟dao
public static void dao(List<String> dataList) {
    log("执行数据库操作");
    //模拟插入数据
    for (String s : dataList) {
        log("插入数据" + s + "成功");
    }
}

public static void main(String[] args) {
    //需要插入的数据
    List<String> dataList = new ArrayList<>();
    for (int i = 0; i < 3; i++) {
        dataList.add("数据" + i);
    }

    //模拟5个请求
    int requestCount = 5;
    for (int i = 0; i < requestCount; i++) {
        String traceId = String.valueOf(i);
        disposeRequestExecutor.execute(() -> {
            //把traceId放入口袋中
            traceIdKD.set(traceId);
            try {
                controller(dataList);
            } finally {
                //将traceId从口袋中移除
                traceIdKD.remove();
            }
        });
    }

    disposeRequestExecutor.shutdown();
}
}

```

输出：

```
****1565339644214[traceId:1],[线程:disposeRequestThread-  
2],com.itsoku.chat24.Demo3.controller(Demo3.java:41):接受请求  
****1565339644214[traceId:2],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.controller(Demo3.java:41):接受请求  
****1565339644214[traceId:0],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.controller(Demo3.java:41):接受请求  
****1565339644214[traceId:2],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.service(Demo3.java:47):执行业务  
****1565339644214[traceId:1],[线程:disposeRequestThread-  
2],com.itsoku.chat24.Demo3.service(Demo3.java:47):执行业务  
****1565339644214[traceId:2],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:53):执行数据库操作  
****1565339644214[traceId:0],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.service(Demo3.java:47):执行业务  
****1565339644214[traceId:2],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据0成功  
****1565339644214[traceId:0],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:53):执行数据库操作  
****1565339644214[traceId:1],[线程:disposeRequestThread-  
2],com.itsoku.chat24.Demo3.dao(Demo3.java:53):执行数据库操作  
****1565339644215[traceId:0],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据0成功  
****1565339644215[traceId:2],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据1成功  
****1565339644215[traceId:0],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据1成功  
****1565339644215[traceId:1],[线程:disposeRequestThread-  
2],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据0成功  
****1565339644215[traceId:0],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据2成功  
****1565339644215[traceId:2],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据2成功  
****1565339644215[traceId:1],[线程:disposeRequestThread-  
2],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据1成功  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.controller(Demo3.java:41):接受请求  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.controller(Demo3.java:41):接受请求  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.service(Demo3.java:47):执行业务  
****1565339644215[traceId:1],[线程:disposeRequestThread-  
2],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据2成功  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:53):执行数据库操作  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.service(Demo3.java:47):执行业务  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据0成功  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:53):执行数据库操作  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据1成功  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据0成功  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据2成功  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据1成功  
****1565339644215[traceId:4],[线程:disposeRequestThread-  
3],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据0成功  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据2成功  
****1565339644215[traceId:3],[线程:disposeRequestThread-  
1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据1成功
```

```
****1565339644215[traceId:3],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo3.dao(Demo3.java:56):插入数据数据2成功
```

可以看出输出和刚才使用traceId参数的方式结果一致，但是却简单了很多。不用去修改controller、service、dao代码了，风险也减少了很多。

代码中创建了一个`ThreadLocal traceIdKD`，这个对象用来操作Thread中一个口袋，用这个口袋来存放traceld。在main方法中通过`traceIdKD.set(traceId)`方法将traceld放入口袋，log方法中通过`traceIdKD.get()`获取口袋中的traceld，最后任务处理完之后，main中的finally中调用`traceIdKD.remove()`将口袋中的traceld清除。

ThreadLocal的官方API解释为：

“该类提供了线程局部 (thread-local) 变量。这些变量不同于它们的普通对应物，因为访问某个变量（通过其 get 或 set 方法）的每个线程都有自己的局部变量，它独立于变量的初始化副本。ThreadLocal 实例通常是类中的 private static 字段，它们希望将状态与某一个线程（例如，用户 ID 或事务 ID）相关联。”

InheritableThreadLocal

继续上面的实例，dao中循环处理dataList的内容，假如dataList处理比较耗时，我们想加快处理速度有什么办法么？大家已经想到了，用多线程并行处理`dataList`，那么我们把代码改一下，如下：

```
package com.itsoku.chat24;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo4 {

    //创建一个操作Thread中存放请求任务追踪id口袋的对象
    static ThreadLocal<String> traceIdKD = new ThreadLocal<>();

    static AtomicInteger threadIndex = new AtomicInteger(1);
    //创建处理请求的线程池子
    static ThreadPoolExecutor disposeRequestExecutor = new ThreadPoolExecutor(3,
            3,
            60,
            TimeUnit.SECONDS,
            new LinkedBlockingDeque<>(),
            r -> {
                Thread thread = new Thread(r);
                thread.setName("disposeRequestThread-" +
                        threadIndex.getAndIncrement());
                return thread;
            });
}
```

```

//记录日志
public static void log(String msg) {
    StackTraceElement stack[] = (new Throwable()).getStackTrace();
    //获取当前线程存放traceId口袋中的内容
    String traceId = traceIdKD.get();
    System.out.println("****" + System.currentTimeMillis() + "[traceId:" +
traceId + "],[线程:" + Thread.currentThread().getName() + "]," + stack[1] + ":" +
msg);
}

//模拟controller
public static void controller(List<String> dataList) {
    log("接受请求");
    service(dataList);
}

//模拟service
public static void service(List<String> dataList) {
    log("执行业务");
    dao(dataList);
}

//模拟dao
public static void dao(List<String> dataList) {
    CountDownLatch countDownLatch = new CountDownLatch(dataList.size());

    log("执行数据库操作");
    String threadName = Thread.currentThread().getName();
    //模拟插入数据
    for (String s : dataList) {
        new Thread(() -> {
            try {
                //模拟数据库操作耗时100毫秒
                TimeUnit.MILLISECONDS.sleep(100);
                log("插入数据" + s + "成功,主线程: " + threadName);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                countDownLatch.countDown();
            }
        }).start();
    }
    //等待上面的dataList处理完毕
    try {
        countDownLatch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    //需要插入的数据
    List<String> dataList = new ArrayList<>();
    for (int i = 0; i < 3; i++) {
        dataList.add("数据" + i);
    }

    //模拟5个请求
}

```

```

        int requestCount = 5;
        for (int i = 0; i < requestCount; i++) {
            String traceId = String.valueOf(i);
            disposeRequestExecutor.execute(() -> {
                //把traceId放入口袋中
                traceIdKD.set(traceId);
                try {
                    controller(dataList);
                } finally {
                    //将traceId从口袋中移除
                    traceIdKD.remove();
                }
            });
        }

        disposeRequestExecutor.shutdown();
    }
}

```

输出：

```

*****1565339904279[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
*****1565339904279[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
*****1565339904279[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
*****1565339904279[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
*****1565339904279[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
*****1565339904279[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
*****1565339904279[traceId:0],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
*****1565339904279[traceId:1],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
*****1565339904279[traceId:2],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
*****1565339904281[traceId:null],[线程:Thread-3],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据0成功,主线程:disposeRequestThread-1
*****1565339904281[traceId:null],[线程:Thread-5],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据0成功,主线程:disposeRequestThread-2
*****1565339904281[traceId:null],[线程:Thread-4],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据0成功,主线程:disposeRequestThread-3
*****1565339904281[traceId:null],[线程:Thread-6],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据1成功,主线程:disposeRequestThread-3
*****1565339904281[traceId:null],[线程:Thread-9],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据1成功,主线程:disposeRequestThread-3
*****1565339904282[traceId:null],[线程:Thread-8],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据1成功,主线程:disposeRequestThread-1

```

```
****1565339904282[traceId:null],[线程:Thread-10],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据2成功,主线程:disposeRequestThread-1
****1565339904282[traceId:3],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565339904282[traceId:null],[线程:Thread-7],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据1成功,主线程:disposeRequestThread-2
****1565339904282[traceId:null],[线程:Thread-11],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据2成功,主线程:disposeRequestThread-2
****1565339904282[traceId:3],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565339904282[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565339904283[traceId:3],[线程:disposeRequestThread-3],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565339904283[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565339904283[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565339904283[traceId:null],[线程:Thread-12],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据0成功,主线程:disposeRequestThread-3
****1565339904283[traceId:null],[线程:Thread-13],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据1成功,主线程:disposeRequestThread-3
****1565339904283[traceId:null],[线程:Thread-14],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据0成功,主线程:disposeRequestThread-1
****1565339904284[traceId:null],[线程:Thread-15],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据2成功,主线程:disposeRequestThread-3
****1565339904284[traceId:null],[线程:Thread-17],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据2成功,主线程:disposeRequestThread-1
****1565339904284[traceId:null],[线程:Thread-16],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:62):插入数据数据1成功,主线程:disposeRequestThread-1
```

看一下上面的输出，有些traceId为null，这是为什么呢？这是因为dao中为了提升处理速度，创建了子线程来并行处理，子线程调用log的时候，去自己的存放traceId的口袋中拿去东西，肯定是空的了。

那有什么办法么？可不可以这样？

父线程相当于主管，子线程相当于干活的小弟，主管让小弟们干活的时候，将自己兜里面的东西复制一份给小弟们使用，主管兜里面可能有很多牛逼的工具，为了提升小弟们的工作效率，给小弟们都复制一个，丢到小弟们的兜里，然后小弟就可以从自己的兜里拿去这些东西使用了，也可以清空自己兜里面的东西。

Thread 对象中有个 `inheritableThreadLocals` 变量，代码如下：

```
ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;
```

`inheritableThreadLocals` 相当于线程中另外一种兜，这种兜有什么特征呢，当创建子线程的时候，子线程会将父线程这种类型兜的东西全部复制一份放到自己的 `inheritableThreadLocals` 兜中，使用 `InheritableThreadLocal` 对象可以操作线程中的 `inheritableThreadLocals` 兜。

InheritableThreadLocal 常用的方法也有3个：

```
//向Thread中某个口袋中放东西  
public void set(T value);  
//获取这个口袋中目前放的东西  
public T get();  
//清空这个口袋中放的东西  
public void remove()
```

使用 InheritableThreadLocal 解决上面子线程中无法输出traceId的问题，只需要将上一个示例代码中的 ThreadLocal 替换成 InheritableThreadLocal 即可，代码如下：

```
package com.itsoku.chat24;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.LinkedBlockingDeque;  
import java.util.concurrent.ThreadPoolExecutor;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.atomic.AtomicInteger;  
  
/**  
 * 跟着阿里P7学并发，微信公众号：javacode2018  
 */  
public class Demo4 {  
  
    //创建一个操作Thread中存放请求任务追踪id口袋的对象，子线程可以继承父线程中内容  
    static InheritableThreadLocal<String> traceIdKD = new  
InheritableThreadLocal<>();  
  
    static AtomicInteger threadIndex = new AtomicInteger(1);  
    //创建处理请求的线程池子  
    static ThreadPoolExecutor disposeRequestExecutor = new ThreadPoolExecutor(3,  
        3,  
        60,  
        TimeUnit.SECONDS,  
        new LinkedBlockingDeque<>(),  
        r -> {  
            Thread thread = new Thread(r);  
            thread.setName("disposeRequestThread-" +  
threadIndex.getAndIncrement());  
            return thread;  
        });  
  
    //记录日志  
    public static void log(String msg) {  
        StackTraceElement stack[] = (new Throwable()).getStackTrace();  
        //获取当前线程存放traceId口袋中的内容  
        String traceId = traceIdKD.get();  
        System.out.println("****" + System.currentTimeMillis() + "[traceId:" +  
traceId + "],[线程:" + Thread.currentThread().getName() + "]," + stack[1] + ":" +  
msg);  
    }  
  
    //模拟controller
```

```

public static void controller(List<String> dataList) {
    log("接受请求");
    service(dataList);
}

//模拟service
public static void service(List<String> dataList) {
    log("执行业务");
    dao(dataList);
}

//模拟dao
public static void dao(List<String> dataList) {
    CountDownLatch countDownLatch = new CountDownLatch(dataList.size());

    log("执行数据库操作");
    String threadName = Thread.currentThread().getName();
    //模拟插入数据
    for (String s : dataList) {
        new Thread(() -> {
            try {
                //模拟数据库操作耗时100毫秒
                TimeUnit.MILLISECONDS.sleep(100);
                log("插入数据" + s + "成功,主线程: " + threadName);
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                countDownLatch.countDown();
            }
        }).start();
    }
    //等待上面的dataList处理完毕
    try {
        countDownLatch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    //需要插入的数据
    List<String> dataList = new ArrayList<>();
    for (int i = 0; i < 3; i++) {
        dataList.add("数据" + i);
    }

    //模拟5个请求
    int requestCount = 5;
    for (int i = 0; i < requestCount; i++) {
        String traceId = String.valueOf(i);
        disposeRequestExecutor.execute(() -> {
            //把traceId放入口袋中
            traceIdKD.set(traceId);
            try {
                controller(dataList);
            } finally {
                //将traceId从口袋中移除
                traceIdKD.remove();
            }
        });
    }
}

```

```

        }
    });

    disposeRequestExecutor.shutdown();
}

}

```

输出：

```

****1565341611454[traceId:1],[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565341611454[traceId:2],[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565341611454[traceId:0],[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565341611454[traceId:2],[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565341611454[traceId:1],[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565341611454[traceId:0],[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565341611454[traceId:2],[线程:disposeRequestThread-
3],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565341611454[traceId:1],[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565341611454[traceId:0],[线程:disposeRequestThread-
1],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565341611557[traceId:2],[线程:Thread-
5],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据0成功,主线程:
disposeRequestThread-3
****1565341611557[traceId:0],[线程:Thread-
4],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据0成功,主线程:
disposeRequestThread-1
****1565341611557[traceId:1],[线程:Thread-
11],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据2成功,主线程:
disposeRequestThread-2
****1565341611557[traceId:1],[线程:Thread-
3],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据0成功,主线程:
disposeRequestThread-2
****1565341611557[traceId:1],[线程:Thread-
8],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据1成功,主线程:
disposeRequestThread-2
****1565341611557[traceId:0],[线程:Thread-
6],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据1成功,主线程:
disposeRequestThread-1
****1565341611557[traceId:0],[线程:Thread-
10],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据2成功,主线程:
disposeRequestThread-1
****1565341611557[traceId:3],[线程:disposeRequestThread-
2],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565341611557[traceId:2],[线程:Thread-
9],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据2成功,主线程:
disposeRequestThread-3
****1565341611558[traceId:2],[线程:Thread-
7],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据1成功,主线程:
disposeRequestThread-3

```

```
****1565341611557[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565341611557[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.controller(Demo4.java:42):接受请求
****1565341611558[traceId:3],[线程:disposeRequestThread-2],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565341611558[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.service(Demo4.java:48):执行业务
****1565341611558[traceId:4],[线程:disposeRequestThread-1],com.itsoku.chat24.Demo4.dao(Demo4.java:56):执行数据库操作
****1565341611659[traceId:3],[线程:Thread-15],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据2成功,主线程:disposeRequestThread-2
****1565341611659[traceId:4],[线程:Thread-14],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据0成功,主线程:disposeRequestThread-1
****1565341611659[traceId:3],[线程:Thread-13],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据1成功,主线程:disposeRequestThread-2
****1565341611659[traceId:3],[线程:Thread-12],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据0成功,主线程:disposeRequestThread-2
****1565341611660[traceId:4],[线程:Thread-16],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据1成功,主线程:disposeRequestThread-1
****1565341611660[traceId:4],[线程:Thread-17],com.itsoku.chat24.Demo4.lambda$dao$1(Demo4.java:64):插入数据数据2成功,主线程:disposeRequestThread-1
```

输出中都有traceId了，和期望的结果一致。

希望通过这篇文章可以学会使用 `InheritableThreadLocal` 和 `InheritedThreadLocal`。有问题可以加我微信 `itsoku` 交流，也可以留言，谢谢。

更多好文章

1. [spring高手系列 \(正在连载中\)](#)
2. [Java高并发系列 \(共34篇\)](#)
3. [MySQL高手系列 \(共27篇\)](#)
4. [Maven高手系列 \(共10篇\)](#)
5. [Mybatis系列 \(共12篇\)](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第25篇：JUC中的阻塞队列

本文内容

1. 掌握Queue、BlockingQueue接口中常用的方法
2. 介绍6中阻塞队列，及相关场景示例
3. 重点掌握4种常用的阻塞队列

Queue接口

队列是一种先进先出（FIFO）的数据结构，java中用 Queue 接口来表示队列。

Queue 接口中定义了6个方法：

```
public interface Queue<E> extends Collection<E> {  
    boolean add(E e);  
    boolean offer(E e);  
    E remove();  
    E poll();  
    E element();  
    E peek();  
}
```

每个 Queue 方法都有两种形式：

- (1) 如果操作失败则抛出异常，
- (2) 如果操作失败，则返回特殊值（`null` 或 `false`，具体取决于操作），接口的常规结构如下表所示。

操作类型	抛出异常	返回特殊值
插入	<code>add(e)</code>	<code>offer(e)</code>
移除	<code>remove()</code>	<code>poll()</code>
检查	<code>element()</code>	<code>peek()</code>

`queue` 从 `Collection` 继承的 `add` 方法插入一个元素，除非它违反了队列的容量限制，在这种情况下它会抛出 `IllegalStateException`；`offer` 方法与 `add` 不同之处仅在于它通过返回 `false` 来表示插入元素失败。

`remove` 和 `poll` 方法都移除并返回队列的头部，确切地移除哪个元素是由具体的实现来决定的，仅当队列为空时，`remove` 和 `poll` 方法的行为才有所不同，在这些情况下，`remove` 抛出 `NoSuchElementException`，而 `poll` 返回 `null`。

`element` 和 `peek` 方法返回队列头部的元素，但不移除，它们之间的差异与 `remove` 和 `poll` 的方式完全相同，如果队列为空，则 `element` 抛出 `NoSuchElementException`，而 `peek` 返回 `null`。

队列一般不要插入空元素。

BlockingQueue 接口

`BlockingQueue` 位于 `juc` 中，熟称阻塞队列，阻塞队列首先它是一个队列，继承 `Queue` 接口，是队列就会遵循先进先出（FIFO）的原则，又因为它是阻塞的，故与普通的队列有两点区别：

1. 当一个线程向队列里面添加数据时，如果队列是满的，那么将阻塞该线程，暂停添加数据
2. 当一个线程从队列里面取出数据时，如果队列是空的，那么将阻塞该线程，暂停取出数据

`BlockingQueue` 相关方法：

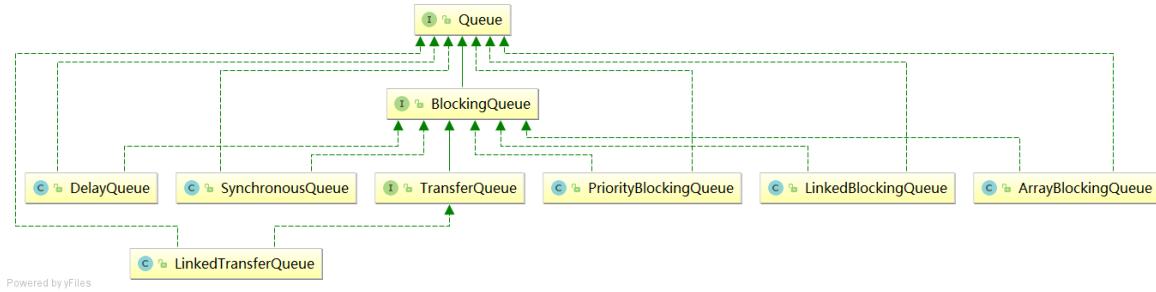
操作类型	抛出异常	返回特殊值	一直阻塞	超时退出
插入	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e,timeout,unit)</code>
移除	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(timeout,unit)</code>
检查	<code>element()</code>	<code>peek()</code>	不支持	不支持

重点，再来解释一下，加深印象：

1. 3个可能会有异常的方法，`add`、`remove`、`element`；这3个方法不会阻塞（是说队列满或者空的情况下是否会阻塞）；队列满的情况下，`add` 抛出异常；队列为空情况下，`remove`、`element` 抛出异常
2. `offer`、`poll`、`peek` 也不会阻塞（是说队列满或者空的情况下是否会阻塞）；队列满的情况下，`offer` 返回 `false`；队列为空的情况下，`poll`、`peek` 返回 `null`
3. 队列满的情况下，调用 `put` 方法会导致当前线程阻塞
4. 队列为空的情况下，调用 `take` 方法会导致当前线程阻塞
5. `offer(e,timeout,unit)`，超时之前，插入成功返回 `true`，否者返回 `false`
6. `poll(timeout,unit)`，超时之前，获取到头部元素并将其移除，返回 `true`，否者返回 `false`
7. 以上一些方法希望大家都记住，方便以后使用

BlockingQueue常见的实现类

看一下相关类图



ArrayBlockingQueue

基于数组的阻塞队列实现，其内部维护一个定长的数组，用于存储队列元素。线程阻塞的实现是通过ReentrantLock来完成的，数据的插入与取出共用同一个锁，因此ArrayBlockingQueue并不能实现生产、消费同时进行。而且在创建ArrayBlockingQueue时，我们还可以控制对象的内部锁是否采用公平锁，默认采用非公平锁。

LinkedBlockingQueue

基于单向链表的阻塞队列实现，在初始化LinkedBlockingQueue的时候可以指定大小，也可以不指定，默认类似一个无限大小的容量（Integer.MAX_VALUE），不指队列容量大小也是会有风险的，一旦数据生产速度大于消费速度，系统内存将有可能被消耗殆尽，因此要谨慎操作。另外LinkedBlockingQueue中用于阻塞生产者、消费者的锁是两个（锁分离），因此生产与消费是可以同时进行的。

PriorityBlockingQueue

一个支持优先级排序的无界阻塞队列，进入队列的元素会按照优先级进行排序

SynchronousQueue

同步阻塞队列，SynchronousQueue没有容量，与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue，每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然

DelayQueue

DelayQueue是一个支持延时获取元素的无界阻塞队列，里面的元素全部都是“可延期”的元素，列头的元素是最先“到期”的元素，如果队列里面没有元素到期，是不能从列头获取元素的，哪怕有元素也不行，也就是说只有在延迟期到时才能够从队列中取元素

LinkedTransferQueue

LinkedTransferQueue是基于链表的FIFO无界阻塞队列，它出现在JDK7中，Doug Lea 大神说LinkedTransferQueue是一个聪明的队列，它是ConcurrentLinkedQueue、SynchronousQueue(公平模式下)、无界的LinkedBlockings等的超集，
LinkedTransferQueue包含了ConcurrentLinkedQueue、SynchronousQueue、LinkedBlockings三种队列的功能

下面我们来介绍每种阻塞队列的使用。

ArrayBlockingQueue

有界阻塞队列，内部使用数组存储元素，有2个常用构造方法：

```
//capacity表示容量大小， 默认内部采用非公平锁
public ArrayBlockingQueue(int capacity)
//capacity: 容量大小, fair: 内部是否是使用公平锁
public ArrayBlockingQueue(int capacity, boolean fair)
```

需求：业务系统中有很多地方需要推送通知，由于需要推送的数据太多，我们将需要推送的信息先丢到阻塞队列中，然后开一个线程进行处理真实发送，代码如下：

```
package com.itsoku.chat25;

import lombok.Data;
import lombok.extern.slf4j.Slf4j;
import sun.text.normalizer.NormalizerBase;

import java.util.Calendar;
import java.util.concurrent.*;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    //推送队列
    static ArrayBlockingQueue<String> pushQueue = new ArrayBlockingQueue<String>(10000);

    static {
        //启动一个线程做真实推送
        new Thread(() -> {
            while (true) {
                String msg;
                try {
                    long startTime = System.currentTimeMillis();
                    //获取一条推送消息，此方法会进行阻塞，直到返回结果
                    msg = pushQueue.take();
                    long endTime = System.currentTimeMillis();
                    //模拟推送耗时
                    TimeUnit.MILLISECONDS.sleep(500);

                    System.out.println(String.format("[%s,%s,take耗时:%s],%s,发送
消息:%s", startTime, endTime, (endTime - startTime),
Thread.currentThread().getName(), msg));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }).start();
    }

    //推送消息，需要发送推送消息的调用该方法，会将推送信息先加入推送队列
    public static void pushMsg(String msg) throws InterruptedException {
        pushQueue.put(msg);
    }

    public static void main(String[] args) throws InterruptedException {
        for (int i = 1; i <= 5; i++) {
            String msg = "一起来学java高并发，第" + i + "天";
        }
    }
}
```

```
//模拟耗时  
TimeUnit.SECONDS.sleep(i);  
Demo1.pushMsg(msg);  
}  
}  
}
```

输出：

```
[1565595629206,1565595630207,take耗时:1001],Thread-0,发送消息:一起来学java高并发,第1天  
[1565595630208,1565595632208,take耗时:2000],Thread-0,发送消息:一起来学java高并发,第2天  
[1565595632208,1565595635208,take耗时:3000],Thread-0,发送消息:一起来学java高并发,第3天  
[1565595635208,1565595639209,take耗时:4001],Thread-0,发送消息:一起来学java高并发,第4天  
[1565595639209,1565595644209,take耗时:5000],Thread-0,发送消息:一起来学java高并发,第5天
```

代码中我们使用了有界队列 `ArrayBlockingQueue`，创建 `ArrayBlockingQueue` 时候需要制定容量大小，调用 `pushQueue.put` 将推送信息放入队列中，如果队列已满，此方法会阻塞。代码中在静态块中启动了一个线程，调用 `pushQueue.take()` 从队列中获取待推送的信息进行推送处理。

注意： `ArrayBlockingQueue` 如果队列容量设置的太小，消费者发送的太快，消费者消费的太慢的情况下，会导致队列空间满，调用 `put` 方法会导致发送者线程阻塞，所以注意设置合理的大小，协调好消费者的速度。

LinkedBlockingQueue

内部使用单向链表实现的阻塞队列，3个构造方法：

```
//默认构造方法，容量大小为Integer.MAX_VALUE  
public LinkedBlockingQueue();  
//创建指定容量大小的LinkedBlockingQueue  
public LinkedBlockingQueue(int capacity);  
//容量为Integer.MAX_VALUE，并将传入的集合丢入队列中  
public LinkedBlockingQueue(Collection<? extends E> c);
```

`LinkedBlockingQueue` 的用法和 `ArrayBlockingQueue` 类似，建议使用的时候指定容量，如果不指定容量，插入的太快，移除的太慢，可能会产生OOM。

PriorityBlockingQueue

无界的优先级阻塞队列，内部使用数组存储数据，达到容量时，会自动进行扩容，放入的元素会按照优先级进行排序，4个构造方法：

```

//默认构造方法， 默认初始化容量是11
public PriorityBlockingQueue();
//指定队列的初始化容量
public PriorityBlockingQueue(int initialCapacity);
//指定队列的初始化容量和放入元素的比较器
public PriorityBlockingQueue(int initialCapacity, Comparator<? super E>
comparator);
//传入集合放入来初始化队列， 传入的集合可以实现SortedSet接口或者PriorityQueue接口进行排序， 如果没有实现这2个接口， 按正常顺序放入队列
public PriorityBlockingQueue(Collection<? extends E> c);

```

优先级队列放入元素的时候，会进行排序，所以我们需要指定排序规则，有2种方式：

1. 创建 `PriorityBlockingQueue` 指定比较器 `Comparator`
2. 放入的元素需要实现 `Comparable` 接口

上面2种方式必须选一个，如果2个都有，则走第一个规则排序。

需求：还是上面的推送业务，目前推送是按照放入的先后顺序进行发送的，比如有些公告比较紧急，优先级比较高，需要快点发送，怎么搞？此时 `PriorityBlockingQueue` 就派上用场了，代码如下：

```

package com.itsoku.chat25;

import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo2 {

    //推送信息封装
    static class Msg implements Comparable<Msg> {
        //优先级，越小优先级越高
        private int priority;
        //推送的信息
        private String msg;

        public Msg(int priority, String msg) {
            this.priority = priority;
            this.msg = msg;
        }

        @Override
        public int compareTo(Msg o) {
            return Integer.compare(this.priority, o.priority);
        }

        @Override
        public String toString() {
            return "Msg{" +
                    "priority=" + priority +
                    ", msg='" + msg + '\'' +
                    '}';
        }
    }
}

```

```

//推送队列
static PriorityBlockingQueue<Msg> pushQueue = new PriorityBlockingQueue<Msg>
();

static {
    //启动一个线程做真实推送
    new Thread(() -> {
        while (true) {
            Msg msg;
            try {
                Long startTime = System.currentTimeMillis();
                //获取一条推送消息，此方法会进行阻塞，直到返回结果
                msg = pushQueue.take();
                //模拟推送耗时
                TimeUnit.MILLISECONDS.sleep(100);
                Long endTime = System.currentTimeMillis();
                System.out.println(String.format("[%s,%s,take耗时:%s],%s,发送
消息:%s", startTime, endTime, (endTime - startTime),
Thread.currentThread().getName(), msg));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
}

//推送消息，需要发送推送消息的调用该方法，会将推送信息先加入推送队列
public static void pushMsg(int priority, String msg) throws
InterruptedException {
    pushQueue.put(new Msg(priority, msg));
}

public static void main(String[] args) throws InterruptedException {
    for (int i = 5; i >= 1; i--) {
        String msg = "一起来学java高并发，第" + i + "天";
        Demo2.pushMsg(i, msg);
    }
}
}

```

输出：

```

[1565598857028,1565598857129,take耗时:101],Thread-0,发送消息:Msg{priority=1,
msg='一起来学java高并发，第1天'}
[1565598857162,1565598857263,take耗时:101],Thread-0,发送消息:Msg{priority=2,
msg='一起来学java高并发，第2天'}
[1565598857263,1565598857363,take耗时:100],Thread-0,发送消息:Msg{priority=3,
msg='一起来学java高并发，第3天'}
[1565598857363,1565598857463,take耗时:100],Thread-0,发送消息:Msg{priority=4,
msg='一起来学java高并发，第4天'}
[1565598857463,1565598857563,take耗时:100],Thread-0,发送消息:Msg{priority=5,
msg='一起来学java高并发，第5天'}

```

main中放入了5条推送信息，i作为消息的优先级按倒叙放入的，最终输出结果中按照优先级由小到大输出。注意Msg实现了 Comparable 接口，具有了比较功能。

SynchronousQueue

同步阻塞队列，SynchronousQueue没有容量，与其他BlockingQueue不同，SynchronousQueue是一个不存储元素的BlockingQueue，每一个put操作必须要等待一个take操作，否则不能继续添加元素，反之亦然。SynchronousQueue在现实中用的不多，线程池中有用到过，`Executors.newCachedThreadPool()`实现中用到了这个队列，当有任务丢入线程池的时候，如果已创建的工作线程都在忙于处理任务，则会新建一个线程来处理丢入队列的任务。

来看示例代码：

```
package com.itsoku.chat25;

import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.SynchronousQueue;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo3 {

    static SynchronousQueue<String> queue = new SynchronousQueue<>();

    public static void main(String[] args) throws InterruptedException {
        new Thread(() -> {
            try {
                long startTime = System.currentTimeMillis();
                queue.put("java高并发系列，路人甲Java!");
                long endTime = System.currentTimeMillis();
                System.out.println(String.format("[%s,%s, take耗时:%s] ,%s",
                    startTime, endTime, (endTime - startTime), Thread.currentThread().getName()));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
        //休眠5秒之后，从队列中take一个元素
        TimeUnit.SECONDS.sleep(5);
        System.out.println(System.currentTimeMillis() + "调用take获取并移除元素，" +
            queue.take());
    }
}
```

输出：

```
1565600421645调用take获取并移除元素，java高并发系列，路人甲Java!
[1565600416645, 1565600421645, take耗时:5000], Thread-0
```

main方法中启动了一个线程，调用`queue.put`方法向队列中丢入一条数据，调用的时候产生了阻塞，从输出结果中可以看出，直到`take`方法被调用时，`put`方法才从阻塞状态恢复正常。

DelayQueue

DelayQueue是一个支持延时获取元素的无界阻塞队列，里面的元素全部都是“可延期”的元素，列头的元素是最先“到期”的元素，如果队列里面没有元素到期，是不能从列头获取元素的，哪怕有元素也不行，也就是说只有在延迟期到时才能够从队列中取元素。

需求：还是推送的业务，有时候我们希望早上9点或者其他指定的时间进行推送，如何实现呢？此时 DelayQueue 就派上用场了。

我们先看一下 DelayQueue 类的声明：

```
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
    implements BlockingQueue<E>
```

元素E需要实现接口 Delayed，我们看一下这个接口的代码：

```
public interface Delayed extends Comparable<Delayed> {
    long getDelay(TimeUnit unit);
}
```

Delayed 继承了 Comparable 接口，这个接口是用来做比较用的，DelayQueue 内部使用 PriorityQueue 来存储数据的，PriorityQueue 是一个优先级队列，丢入的数据会进行排序，排序方法调用的是 Comparable 接口中的方法。下面主要说一下 Delayed 接口中的 getDelay 方法：此方法在给定的时间单位内返回与此对象关联的剩余延迟时间。

对推送我们再做一下处理，让其支持定时发送（定时在将来某个时间也可以说是延迟发送），代码如下：

```
package com.itsoku.chat25;

import java.util.Calendar;
import java.util.concurrent.DelayQueue;
import java.util.concurrent.Delayed;
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {

    //推送信息封装
    static class Msg implements Delayed {
        //优先级，越小优先级越高
        private int priority;
        //推送的信息
        private String msg;
        //定时发送时间，毫秒格式
        private long sendTimeMs;

        public Msg(int priority, String msg, long sendTimeMs) {
            this.priority = priority;
            this.msg = msg;
            this.sendTimeMs = sendTimeMs;
        }

        @Override
        public String toString() {
```

```

        return "Msg{" +
            "priority=" + priority +
            ", msg='" + msg + '\'' +
            ", sendTimeMs=" + sendTimeMs +
            '}';
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert(this.sendTimeMs -
Calendar.getInstance().getTimeInMillis(), TimeUnit.MILLISECONDS);
    }

    @Override
    public int compareTo(Delayed o) {
        if (o instanceof Msg) {
            Msg c2 = (Msg) o;
            return Integer.compare(this.priority, c2.priority);
        }
        return 0;
    }
}

//推送队列
static DelayQueue<Msg> pushQueue = new DelayQueue<Msg>();

static {
    //启动一个线程做真实推送
    new Thread(() -> {
        while (true) {
            Msg msg;
            try {
                //获取一条推送消息，此方法会进行阻塞，直到返回结果
                msg = pushQueue.take();
                //此处可以做真实推送
                Long endTime = System.currentTimeMillis();
                System.out.println(String.format("定时发送时间: %s, 实际发送时间: %s, 发送消息:%s", msg.sendTimeMs, endTime, msg));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }).start();
}

//推送消息，需要发送推送消息的调用该方法，会将推送信息先加入推送队列
public static void pushMsg(int priority, String msg, Long sendTimeMs) throws
InterruptedException {
    pushQueue.put(new Msg(priority, msg, sendTimeMs));
}

public static void main(String[] args) throws InterruptedException {
    for (int i = 5; i >= 1; i--) {
        String msg = "一起来学java高并发,第" + i + "天";
        Demo4.pushMsg(i, msg, Calendar.getInstance().getTimeInMillis() + i *
2000);
    }
}

```

```
}
```

输出：

```
定时发送时间：1565603357198, 实际发送时间：1565603357198, 发送消息：Msg{priority=1,  
msg='一起来学java高并发, 第1天', sendTimeMs=1565603357198}  
定时发送时间：1565603359198, 实际发送时间：1565603359198, 发送消息：Msg{priority=2,  
msg='一起来学java高并发, 第2天', sendTimeMs=1565603359198}  
定时发送时间：1565603361198, 实际发送时间：1565603361199, 发送消息：Msg{priority=3,  
msg='一起来学java高并发, 第3天', sendTimeMs=1565603361198}  
定时发送时间：1565603363198, 实际发送时间：1565603363199, 发送消息：Msg{priority=4,  
msg='一起来学java高并发, 第4天', sendTimeMs=1565603363198}  
定时发送时间：1565603365182, 实际发送时间：1565603365183, 发送消息：Msg{priority=5,  
msg='一起来学java高并发, 第5天', sendTimeMs=1565603365182}
```

可以看出时间发送时间，和定时发送时间基本一致，代码中 `Msg` 需要实现 `Delayed` 接口，重点在于 `getDelay` 方法，这个方法返回剩余的延迟时间，代码中使用 `this.sendTimeMs` 减去当前时间的毫秒格式时间，得到剩余延迟时间。

LinkedTransferQueue

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

`LinkedTransferQueue` 类继承自 `AbstractQueue` 抽象类，并且实现了 `TransferQueue` 接口：

```
public interface TransferQueue<E> extends BlockingQueue<E> {  
    // 如果存在一个消费者已经等待接收它，则立即传送指定的元素，否则返回false，并且不进入队列。  
    boolean tryTransfer(E e);  
    // 如果存在一个消费者已经等待接收它，则立即传送指定的元素，否则等待直到元素被消费者接收。  
    void transfer(E e) throws InterruptedException;  
    // 在上述方法的基础上设置超时时间  
    boolean tryTransfer(E e, long timeout, TimeUnit unit)  
        throws InterruptedException;  
    // 如果至少有一位消费者在等待，则返回true  
    boolean hasWaitingConsumer();  
    // 获取所有等待获取元素的消费线程数量  
    int getWaitingConsumerCount();  
}
```

再看一下上面的这些方法，`transfer(E e)` 方法和 `SynchronousQueue` 的 `put` 方法类似，都需要等待消费者取走元素，否者一直等待。其他方法和 `ArrayBlockingQueue`、`LinkedBlockingQueue` 中的方法类似。

总结

1. 重点需要了解 `BlockingQueue` 中的所有方法，以及他们的区别
2. 重点掌握 `ArrayBlockingQueue`、`LinkedBlockingQueue`、`PriorityBlockingQueue`、`DelayQueue` 的使用场景
3. 需要处理的任务有优先级的，使用 `PriorityBlockingQueue`

4. 处理的任务需要延时处理的，使用 `DelayQueue`

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

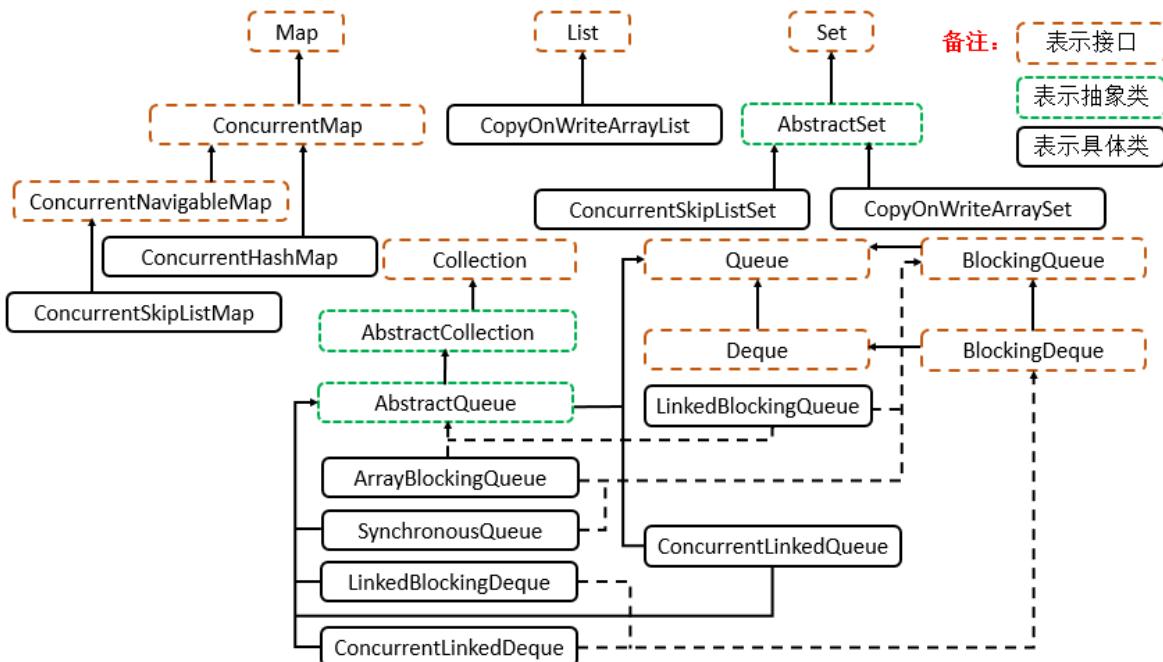
加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第26篇：JUC中一些常见的集合

本文内容

1. 了解JUC常见集合，学会使用
2. ConcurrentHashMap
3. ConcurrentSkipListMap
4. ConcurrentSkipListSet
5. CopyOnWriteArrayList
6. 介绍Queue接口
7. ConcurrentLinkedQueue
8. CopyOnWriteArrayList
9. 介绍Deque接口
10. ConcurrentLinkedDeque

JUC集合框架图



图可以看到，JUC的集合框架也是从Map、List、Set、Queue、Collection等超级接口中继承而来的。所以，大概可以知道JUC下的集合包含了一些基本操作，并且变得线程安全。

Map

ConcurrentHashMap

功能和HashMap基本一致，内部使用红黑树实现的。

特性：

- 1. 迭代结果和存入顺序不一致
 - 2. key和value都不能为空
 - 3. 线程安全的

ConcurrentSkipListMap

内部使用跳表实现的，放入的元素会进行排序，排序算法支持2种方式来指定：

1. 通过构造方法传入一个 `comparator`
 2. 放入的元素实现 `Comparable` 接口

上面2种方式必选一个，如果2种都有，走规则1。

特性：

1. 迭代结果和存入顺序不一致
 2. 放入的元素会排序
 3. key和value都不能为空
 4. 线程安全的

List

CopyOnWriteArrayList

实现List的接口的，一般我们使用ArrayList、LinkedList、Vector，其中只有Vector是线程安全的，可以使用Collections静态类的synchronizedList方法对ArrayList、LinkedList包装为线程安全的List，不过这些方式在保证线程安全的情况下性能都不高。

CopyOnWriteArrayList是线程安全的List，内部使用数组存储数据，集合中多线程并行操作一般存在4种情况：读读、读写、写写、写读，这个只有在写写操作过程中会导致其他线程阻塞，其他3种情况均不会阻塞，所以读取的效率非常高。

可以看一下这个类的名称：CopyOnWrite，意思是在写入操作的时候，进行一次自我复制，换句话说，当这个List需要修改时，并不修改原有内容（这对于保证当前在读线程的数据一致性非常重要），而是在原有存放数据的数组上产生一个副本，在副本上修改数据，修改完毕之后，用副本替换原来的数组，这样也保证了写操作不会影响读。

特性：

1. 迭代结果和存入顺序一致
2. 元素不重复
3. 元素可以为空
4. 线程安全的
5. 读读、读写、写读3种情况不会阻塞；写写会阻塞
6. 无界的

Set

ConcurrentSkipListSet

有序的Set，内部基于ConcurrentSkipListMap实现的，放入的元素会进行排序，排序算法支持2种方式来指定：

1. 通过构造方法传入一个Comparator
2. 放入的元素实现Comparable接口

上面2种方式需要实现一个，如果2种都有，走规则1

特性：

1. 迭代结果和存入顺序不一致
2. 放入的元素会排序
3. 元素不重复
4. 元素不能为空
5. 线程安全的
6. 无界的

CopyOnWriteArraySet

内部使用CopyOnWriteArrayList实现的，将所有的操作都会转发给CopyOnWriteArrayList。

特性：

1. 迭代结果和存入顺序不一致

2. 元素不重复
3. 元素可以为空
4. 线程安全的
5. 读读、读写、写读 不会阻塞；写写会阻塞
6. 无界的

Queue

Queue接口中的方法，我们再回顾一下：

操作类型	抛出异常	返回特殊值
插入	<code>add(e)</code>	<code>offer(e)</code>
移除	<code>remove()</code>	<code>poll()</code>
检查	<code>element()</code>	<code>peek()</code>

3种操作，每种操作有2个方法，不同点是队列为空或者满载时，调用方法是抛出异常还是返回特殊值，大家按照表格中的多看几遍，加深记忆。

ConcurrentLinkedQueue

高效并发队列，内部使用链表实现的。

特性：

1. 线程安全的
2. 迭代结果和存入顺序一致
3. 元素可以重复
4. 元素不能为空
5. 线程安全的
6. 无界队列

Deque

先介绍一下Deque接口，双向队列(Deque)是Queue的一个子接口，双向队列是指该队列两端的元素既能入队(offer)也能出队(poll)，如果将Deque限制为只能从一端入队和出队，则可实现栈的数据结构。对于栈而言，有入栈(push)和出栈(pop)，遵循先进后出原则。

一个线性 collection，支持在两端插入和移除元素。名称 `deque` 是“double ended queue (双端队列)”的缩写，通常读为“deck”。大多数 `Deque` 实现对于它们能够包含的元素数没有固定限制，但此接口既支持有容量限制的双端队列，也支持没有固定大小限制的双端队列。

此接口定义在双端队列两端访问元素的方法。提供插入、移除和检查元素的方法。每种方法都存在两种形式：一种形式在操作失败时抛出异常，另一种形式返回一个特殊值（`null` 或 `false`，具体取决于操作）。插入操作的后一种形式是专为使用有容量限制的 `Deque` 实现设计的；在大多数实现中，插入操作不能失败。

下表总结了上述 12 种方法：

	第一个元素 (头部)	第一个元素 (头部)	最后一个元素 (尾部)	最后一个元素 (尾部)
	抛出异常	特殊值	抛出异常	特殊值
插入	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
移除	removeFirst()	pollFirst()	removeLast()	pollLast()
检查	getFirst()	peekFirst()	getLast()	peekLast()

此接口扩展了 `Queue` 接口。在将双端队列用作队列时，将得到 FIFO（先进先出）行为。将元素添加到双端队列的末尾，从双端队列的开头移除元素。从 `Queue` 接口继承的方法完全等效于 `Deque` 方法，如下表所示：

此接口扩展了 `Queue` 接口。在将双端队列用作队列时，将得到 FIFO（先进先出）行为。将元素添加到双端队列的末尾，从双端队列的开头移除元素。从 `Queue` 接口继承的方法完全等效于 `Deque` 方法，如下表所示：

Queue 方法	等效 Deque 方法
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

ConcurrentLinkedDeque

实现了 `Deque` 接口，内部使用链表实现的高效的并发双端队列。

特性：

1. 线程安全的
2. 迭代结果和存入顺序一致
3. 元素可以重复
4. 元素不能为空
5. 线程安全的
6. 无界队列

BlockingQueue

关于阻塞队列，上一篇有详细介绍，可以看看：[掌握JUC中的阻塞队列](#)

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第27篇：实战：你的接口太慢了需要优化

开发环境：jdk1.8。

案例讲解

电商app都有用过吧，商品详情页，需要给他们提供一个接口获取商品相关信息：

1. 商品基本信息（名称、价格、库存、会员价格等）
2. 商品图片列表
3. 商品描述信息（描述信息一般是由富文本编辑的大文本信息）

数据库中我们用了3张表存储上面的信息：

1. 商品基本信息表：t_goods（字段：id【商品id】、名称、价格、库存、会员价格等）
2. 商品图片信息表：t_goods_imgs（字段：id、goods_id【商品id】、图片路径），一个商品会有多张图片
3. 商品描述信息表：t_goods_ext（字段：id, goods_id【商品id】、商品描述信息【大字段】）

这需求对于大家来说很简单吧，伪代码如下：

```

public Map<String, Object> detail(long goodsId) {
    //创建一个map
    //step1: 查询商品基本信息，放入map
    map.put("goodsModel", (select * from t_goods where id = #gooldsid#));
    //step2: 查询商品图片列表，返回一个集合放入map
    map.put("goodsImgModelList", (select * from t_goods_imgs where goods_id =
    #gooldsid#));
    //step3: 查询商品描述信息，放入map
    map.put("goodsExtModel", (select * from t_goods_ext where goods_id =
    #gooldsid#));
    return map;
}

```

上面这种写法应该很常见，代码很简单，假设上面每个步骤耗时200ms，此接口总共耗时 ≥ 600 毫秒，其他还涉及到网络传输耗时，估计总共会在700ms左右，此接口有没有优化的空间，性能能够提升多少？我们一起来挑战一下。

在看一下上面的逻辑，整个过程是按顺序执行的，实际上3个查询之间是没有任何依赖关系，所以说3个查询可以同时执行，那我们对这3个步骤采用多线程并行执行，看一下最后什么情况，代码如下：

```

package com.itsoku.chat26;

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {

    /**
     * 获取商品基本信息
     *
     * @param goodsId 商品id
     * @return 商品基本信息
     * @throws InterruptedException
     */
    public String goodsDetailModel(long goodsId) throws InterruptedException {
        //模拟耗时，休眠200ms
        TimeUnit.MILLISECONDS.sleep(200);
        return "商品id：" + goodsId + ",商品基本信息....";
    }

    /**
     * 获取商品图片列表
     *
     * @param goodsId 商品id
     * @return 商品图片列表
     * @throws InterruptedException
     */
    public List<String> goodsImgModelList(long goodsId) throws
InterruptedException {
        //模拟耗时，休眠200ms
}

```

```

        TimeUnit.MILLISECONDS.sleep(200);
        return Arrays.asList("图1", "图2", "图3");
    }

    /**
     * 获取商品描述信息
     *
     * @param goodsId 商品id
     * @return 商品描述信息
     * @throws InterruptedException
     */
    public String goodsExtModel(long goodsId) throws InterruptedException {
        //模拟耗时，休眠200ms
        TimeUnit.MILLISECONDS.sleep(200);
        return "商品id:" + goodsId + ",商品描述信息.....";
    }

    //创建个线程池
    ExecutorService executorService = Executors.newFixedThreadPool(10);

    /**
     * 获取商品详情
     *
     * @param goodsId 商品id
     * @return
     * @throws ExecutionException
     * @throws InterruptedException
     */
    public Map<String, Object> goodsDetail(long goodsId) throws
ExecutionException, InterruptedException {
        Map<String, Object> result = new HashMap<>();

        //异步获取商品基本信息
        Future<String> gooldsDetailModelFuture = executorService.submit(() ->
goodsDetailModel(goodsId));
        //异步获取商品图片列表
        Future<List<String>> goodsImgModelListFuture =
executorService.submit(() -> goodsImgModelList(goodsId));
        //异步获取商品描述信息
        Future<String> goodsExtModelFuture = executorService.submit(() ->
goodsExtModel(goodsId));

        result.put("gooldDetailModel", gooldsDetailModelFuture.get());
        result.put("goodsImgModelList", goodsImgModelListFuture.get());
        result.put("goodsExtModel", goodsExtModelFuture.get());
        return result;
    }

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        long startTime = System.currentTimeMillis();
        Map<String, Object> map = new Demo1().goodsDetail(1L);
        System.out.println(map);
        System.out.println("耗时(ms):" + (System.currentTimeMillis() -
startTime));
    }
}

```

输出：

```
{goods_imgsModelList=[图1, 图2, 图3], goodsDetailModel=商品id:1,商品基本信息....,
goodsExtModel=商品id:1,商品描述信息.....}
耗时(ms):208
```

可以看出耗时200毫秒左右，性能提升了2倍，假如这个接口中还存在其他无依赖的操作，性能提升将更加显著，上面使用了线程池并行去执行3次查询的任务，最后通过Future获取异步执行结果。

整个优化过程

1. 先列出无依赖的一些操作
2. 将这些操作改为并行的方式

用到的技术有

1. [线程池相关知识](#)
2. [Executors、Future相关知识](#)

总结

1. 对于无依赖的操作尽量采用并行方式去执行，可以很好的提升接口的性能
2. 大家可以在你们的系统中试试这种方法，感受一下效果，会让你感觉很爽

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第28篇：实战：构建日志系统

环境：jdk1.8。

本文内容

1. 日志有什么用？
2. 日志存在的痛点？
3. 构建日志系统

日志有什么用？

1. 系统出现故障的时候，可以通过日志信息快速定位问题，修复bug，恢复业务
2. 提取有用数据，做数据分析使用

本文主要讨论通过日志来快速定位并解决问题。

日志存在的痛点

先介绍一下多数公司采用的方式：目前比较流行的是采用springcloud（或者dubbo）做微服务，按照业拆分为多个独立的服务，服务采用集群的方式部署在不同的机器上，当一个请求过来的时候，可能会调用到很多服务进行处理，springcloud一般采用logback（或者log4j）输出日志到文件中。当系统出问题的时候，按照系统故障的严重程度，严重的会回退版本，然后排查bug，轻的，找运维去线上拉日志，然后排查问题。

这个过程中存在一些问题：

1. 日志文件太多太多，不方便查找
2. 日志分散在不同的机器上，也不方便查找
3. 一个请求可能会调用多个服务，完整的日志难以追踪
4. 系统出现了问题，只能等到用户发现了，自己才知道

本文要解决上面的几个痛点，构建我们的日志系统，达到以下要求：

1. 方便追踪一个请求完整的日志
2. 方便快速检索日志
3. 系统出现问题自动报警，通知相关人员

构建日志系统

按照上面我们定的要求，一个个解决。

方便追踪一个请求完整的日志

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

当一个请求过来的时候，可能会调用多个服务，多个服务内部可能又会产生子线程处理业务，所以这里面有两个问题需要解决：

1. 多个服务之间日志的追踪
2. 服务内部子线程和主线程日志的追踪，这个地方举个例子，比如一个请求内部需要给10000人发送推送，内部开启10个线程并行处理，处理完毕之后响应操作者，这里面有父子线程，我们要能够找到这个里面所有的日志

需要追踪一个请求完整日志，我们需要给每个请求设置一个全局唯一编号，可以使用UUID或者其他方式也行。

多个服务之间日志追踪的问题：当一个请求过来的时候，在入口处生成一个trace_id，然后放在ThreadLocal中，如果内部设计到多个服务之间相互调用，调用其他服务的时，将trace_id顺便携带过去。

父子线程日志追踪的问题：可以采用InheritableThreadLocal来存放trace_id，这样可以在线程中获取到父线程中的trace_id。

所以此处我们需要使用 InheritableThreadLocal 来存储trace_id。

关于ThreadLocal和InheritableThreadLocal可以参考：[ThreadLocal、InheritableThreadLocal \(通俗易懂\)](#)

如果自己使用了线程池处理请求的，由于线程池中的线程采用的是复用的方式，所以需要对执行的任务Runnable做一些改造，如代码：

```
public class TraceRunnable implements Runnable {  
    private String traceId;  
    private Runnable target;  
  
    public TraceRunnable(Runnable target) {  
        this.traceId = TraceUtil.get();  
        this.target = target;  
    }  
  
    @Override  
    public void run() {  
        try {  
            TraceUtil.set(this.traceId);  
            MDC.put(TraceUtil.MDC_TRACE_ID, TraceUtil.get());  
            this.target.run();  
        } finally {  
            MDC.remove(TraceUtil.MDC_TRACE_ID);  
            TraceUtil.remove();  
        }  
    }  
  
    public static Runnable trace(Runnable target) {  
        return new TraceRunnable(target);  
    }  
}
```

需要用线程池执行的任务使用 TraceRunnable 封装一下就可以了。

TraceUtil代码：

```

public class TraceUtil {

    public static final String REQUEST_HEADER_TRACE_ID =
"com.ms.header.trace.id";
    public static final String MDC_TRACE_ID = "trace_id";

    private static InheritableThreadLocal<String> inheritableThreadLocal = new
InheritableThreadLocal<>();

    /**
     * 获取traceid
     *
     * @return
     */
    public static String get() {
        String traceId = inheritableThreadLocal.get();
        if (traceId == null) {
            traceId = IDUtil.getId();
            inheritableThreadLocal.set(traceId);
        }
        return traceId;
    }

    public static void set(String trace_id) {
        inheritableThreadLocal.set(trace_id);
    }

    public static void remove() {
        inheritableThreadLocal.remove();
    }

}

```

日志输出中携带上trace_id，这样最终我们就可以通过trace_id找到一个请求的完整日志了。

方便快速检索日志

日志分散在不同的机器上，如果要快速检索，需要将所有服务产生的日志汇集到一个地方。

关于检索日志的，列一下需求：

1. 我们将收集日志发送到消息中间件中（可以是kafka、rocketmq），消息中间件这块不介绍，选择玩的比较溜的就可以了
2. 系统产生日志尽量不要影响接口的效率
3. 带宽有限的情况下，发送日志也尽量不要去影响业务
4. 日志尽量低延次，产生的日志，尽量在生成之后1分钟后可以检索到
5. 检索日志功能要能够快速响应

关于上面几点，我们需要做的：日志发送的地方进行改造，引入消息中间件，将日志异步发送到消息中间件中，查询的地方采用elasticsearch，日志系统需要订阅消息中间件中的日志，然后丢给elasticsearch建索引，方便快速检索，咱们来一点点的介绍。

日志发送端的改造

日志是有业务系统产生的，一个请求过来的时候会产生很多日志，日志产生时，我们尽量减少日志输出对业务耗时的影响，我们的过程如下：

1. 业务系统内部引用一个线程池来异步处理日志，线程池内部可以使用一个容量稍微大一点的阻塞队列
2. 业务系统将日志丢给线程池进行处理
3. 线程池中将需要处理的日志先压缩一下，然后发送至mq

线程池的使用可以参考：[JAVA线程池，这一篇就够了](#)

引入mq存储日志

业务系统将日志先发送到mq中，后面由其他消费者订阅进行消费。日志量比较大的，对mq的要求也比较高，可以选择kafka，业务量小的，也可以选取activemq。

使用elasticsearch来检索日志

elasticsearch（以下简称es）是一个全文检索工具，具体详情可以参考其官网相关文档。使用它来检索数据效率非常高。日志系统中需要我们开发一个消费端来拉取mq中的消息，将其存储到es中方便快速检索，关于这块有几点说一下：

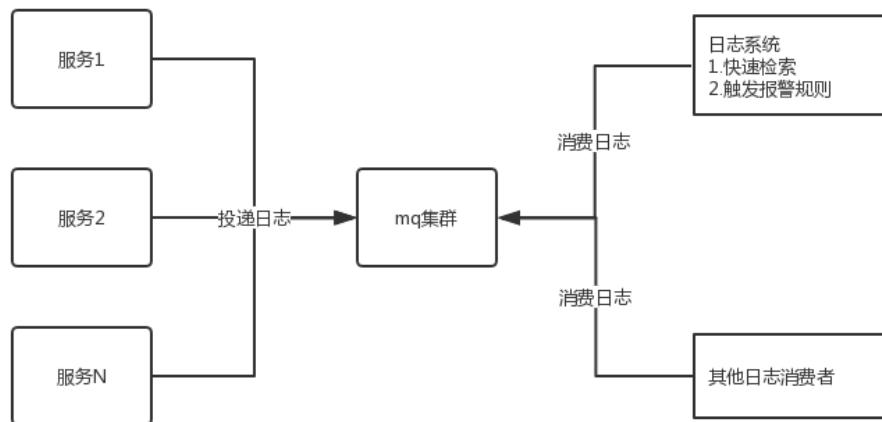
1. 建议按天在es中建立数据库，日志量非常大的，也可以按小时建立数据库。查询的时候，时间就是必选条件了，这样可以让es定位到日志库进行检索，提升检索效率
2. 日志常见的需要收集的信息：trace_id、时间、日志级别、类、方法、url、调用的接口开始时间、调用接口的结束时间、接口耗时、接口状态码、异常信息、日志信息等等，可以按照这些在es中建立索引，方便检索。

日志监控报警

日志监控报警是非常重要的，这个必须要有，日志系统中需要开发监控报警功能，这块我们可以做成通过页面配置的方式，支持报警规则的配置，如日志中产生了某些异常、接口响应时间大于多少、接口返回状态码404等异常信息的时候能够报警，具体的报警可以是语音电话、短信通知、钉钉机器人报警等等，这些也做成可以配置的。

日志监控模块从mq中拉取日志，然后去匹配我们启用的一些规则进行报警。

结构图如下



关于搭建日志中遇到的一些痛点，可以加我微信itsoku交流。

构建日志系统需要用到的知识点

1. [java中线程池的使用](#)
2. [ThreadLocal、InheritableThreadLocal（通俗易懂）](#)
3. elasticsearch，可以参考其官方文档
4. mq

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



第29篇：实战：一起来搞懂限流

环境：jdk1.8。

本文内容

1. 介绍常见的限流算法
2. 通过控制最大并发数来进行限流
3. 通过漏桶算法来进行限流
4. 通过令牌桶算法来进行限流
5. 限流工具类RateLimiter

常见的限流的场景

1. 秒杀活动，数量有限，访问量巨大，为了防止系统宕机，需要做限流处理
2. 国庆期间，一般的旅游景点人口太多，采用排队方式做限流处理
3. 医院看病通过发放排队号的方式来做限流处理。

常见的限流算法

1. 通过控制最大并发数来进行限流
2. 使用漏桶算法来进行限流
3. 使用令牌桶算法来进行限流

通过控制最大并发数来进行限流

以秒杀业务为例，10个iphone，100万人抢购，100万人同时发起请求，最终能够抢到的人也就是前面几个人，后面的基本上都没有希望了，那么我们可以通过控制并发数来实现，比如并发数控制在10个，其他超过并发数的请求全部拒绝，提示：秒杀失败，请稍后重试。

并发控制的，通俗解释：一大波人去商场购物，必须经过一个门口，门口有个门卫，兜里面有指定数量的门禁卡，来的人先进去门卫那边拿取门禁卡，拿到卡的人才可以刷卡进入商场，拿不到的可以继续等待。进去的人出来之后会把卡归还给门卫，门卫可以把归还来的卡继续发放给其他排队的顾客使用。

JUC中提供了这样的工具类：Semaphore，示例代码：

```
package com.itsoku.chat29;

import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
```

```

static Semaphore semaphore = new Semaphore(5);

public static void main(String[] args) {
    for (int i = 0; i < 20; i++) {
        new Thread(() -> {
            boolean flag = false;
            try {
                flag = semaphore.tryAcquire(100, TimeUnit.MICROSECONDS);
                if (flag) {
                    //休眠2秒，模拟下单操作
                    System.out.println(Thread.currentThread() + "，尝试下单中。。。。。");
                    TimeUnit.SECONDS.sleep(2);
                } else {
                    System.out.println(Thread.currentThread() + "，秒杀失败，请稍微重试！");
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                if (flag) {
                    semaphore.release();
                }
            }
        }).start();
    }
}

```

输出：

```

Thread[Thread-10,5,main], 尝试下单中。。。。。。
Thread[Thread-8,5,main], 尝试下单中。。。。。。
Thread[Thread-9,5,main], 尝试下单中。。。。。。
Thread[Thread-12,5,main], 尝试下单中。。。。。。
Thread[Thread-11,5,main], 尝试下单中。。。。。。
Thread[Thread-2,5,main], 秒杀失败，请稍微重试！
Thread[Thread-1,5,main], 秒杀失败，请稍微重试！
Thread[Thread-18,5,main], 秒杀失败，请稍微重试！
Thread[Thread-16,5,main], 秒杀失败，请稍微重试！
Thread[Thread-0,5,main], 秒杀失败，请稍微重试！
Thread[Thread-3,5,main], 秒杀失败，请稍微重试！
Thread[Thread-14,5,main], 秒杀失败，请稍微重试！
Thread[Thread-6,5,main], 秒杀失败，请稍微重试！
Thread[Thread-13,5,main], 秒杀失败，请稍微重试！
Thread[Thread-17,5,main], 秒杀失败，请稍微重试！
Thread[Thread-7,5,main], 秒杀失败，请稍微重试！
Thread[Thread-19,5,main], 秒杀失败，请稍微重试！
Thread[Thread-15,5,main], 秒杀失败，请稍微重试！
Thread[Thread-4,5,main], 秒杀失败，请稍微重试！
Thread[Thread-5,5,main], 秒杀失败，请稍微重试！

```

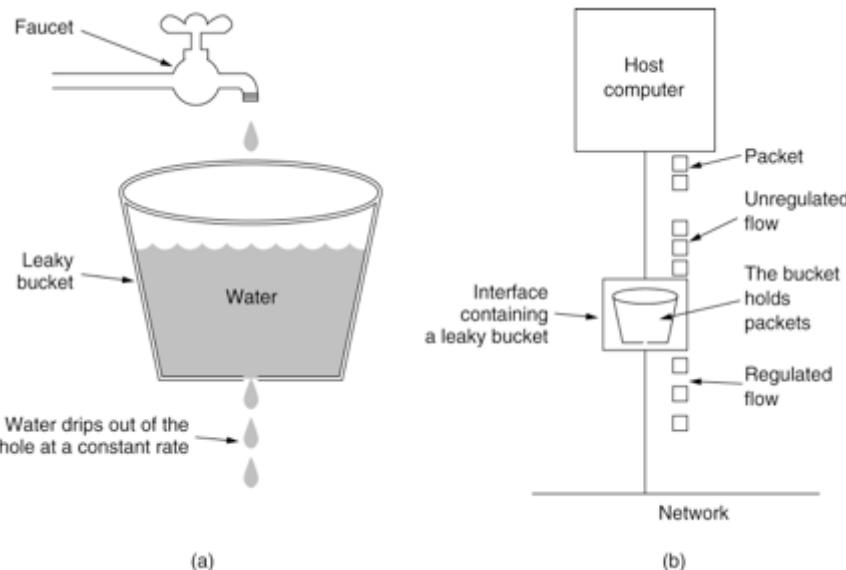
关于 `Semaphore` 的使用，可以移步：[JUC中的Semaphore（信号量）](#)

使用漏桶算法来进行限流

国庆期间比较火爆的景点，人流量巨大，一般入口处会有限流的弯道，让游客进去进行排队，排在前面的人，每隔一段时间会放一拨进入景区。排队人数超过了指定的限制，后面再来的人会被告知今天已经游客量已经达到峰值，会被拒绝排队，让其明天或者以后再来，这种玩法采用漏桶限流的方式。

漏桶算法思路很简单，水（请求）先进入到漏桶里，漏桶以一定的速度出水，当水流速过大会直接溢出，可以看出漏桶算法能强行限制数据的传输速率。

漏桶算法示意图：



简陋版的实现，代码如下：

```
package com.itsoku.chat29;

import java.util.Objects;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.LockSupport;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo2 {

    public static class BucketLimit {
        static AtomicInteger threadNum = new AtomicInteger(1);
        //容量
        private int capacity;
        //流速
        private int flowRate;
        //流速时间单位
        private TimeUnit flowRateUnit;
        private BlockingQueue<Node> queue;
        //漏桶流出的任务时间间隔（纳秒）
        private long flowRateNanosTime;

        public BucketLimit(int capacity, int flowRate, TimeUnit flowRateUnit) {
```

```

        this.capacity = capacity;
        this.flowRate = flowRate;
        this.flowRateUnit = flowRateUnit;
        this.bucketThreadwork();
    }

    //漏桶线程
    public void bucketThreadwork() {
        this.queue = new ArrayBlockingQueue<Node>(capacity);
        //漏桶流出的任务时间间隔(纳秒)
        this.flowRateNanosTime = flowRateUnit.toNanos(1) / flowRate;
        Thread thread = new Thread(this::bucketwork);
        thread.setName("漏桶线程-" + threadNum.getAndIncrement());
        thread.start();
    }

    //漏桶线程开始工作
    public void bucketwork() {
        while (true) {
            Node node = this.queue.poll();
            if (Objects.nonNull(node)) {
                //唤醒任务线程
                LockSupport.unpark(node.thread);
            }
            //休眠flowRateNanosTime
            LockSupport.parkNanos(this.flowRateNanosTime);
        }
    }

    //返回一个漏桶
    public static BucketLimit build(int capacity, int flowRate, TimeUnit
flowRateUnit) {
        if (capacity < 0 || flowRate < 0) {
            throw new IllegalArgumentException("capacity、flowRate必须大于
0！");
        }
        return new BucketLimit(capacity, flowRate, flowRateUnit);
    }

    //当前线程加入漏桶，返回false，表示漏桶已满；true：表示被漏桶限流成功，可以继续处理任
务
    public boolean acquire() {
        Thread thread = Thread.currentThread();
        Node node = new Node(thread);
        if (this.queue.offer(node)) {
            LockSupport.park();
            return true;
        }
        return false;
    }

    //漏桶中存放的元素
    class Node {
        private Thread thread;

        public Node(Thread thread) {
            this.thread = thread;
        }
    }

```

```

    }

    public static void main(String[] args) {
        BucketLimit bucketLimit = BucketLimit.build(10, 60, TimeUnit.MINUTES);
        for (int i = 0; i < 15; i++) {
            new Thread(() -> {
                boolean acquire = bucketLimit.acquire();
                System.out.println(System.currentTimeMillis() + " " + acquire);
                try {
                    TimeUnit.SECONDS.sleep(1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }
    }
}

```

代码中 `BucketLimit.build(10, 60, TimeUnit.MINUTES);` 创建了一个容量为10，流水为60/分钟的漏桶。

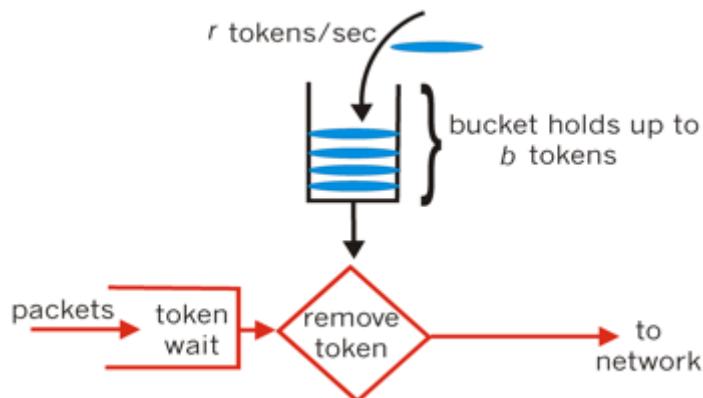
代码中用到的技术有：

1. [BlockingQueue阻塞队列](#)
2. [JUC中的LockSupport工具类，必备技能](#)

使用令牌桶算法来进行限流

令牌桶算法的原理是系统以恒定的速率产生令牌，然后把令牌放到令牌桶中，令牌桶有一个容量，当令牌桶满了的时候，再向其中放令牌，那么多余的令牌会被丢弃；当想要处理一个请求的时候，需要从令牌桶中取出一个令牌，如果此时令牌桶中没有令牌，那么则拒绝该请求。从原理上看，令牌桶算法和漏桶算法是相反的，一个“进水”，一个是“漏水”。这种算法可以应对突发程度的请求，因此比漏桶算法好。

令牌桶算法示意图：



有兴趣的可以自己去实现一个。

限流工具类RateLimiter

Google开源工具包Guava提供了限流工具类RateLimiter，可以非常方便的控制系统每秒吞吐量，示例代码如下：

```
package com.itsoku.chat29;

import com.google.common.util.concurrent.RateLimiter;

import java.util.Calendar;
import java.util.Date;
import java.util.Objects;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.concurrent.locks.LockSupport;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo3 {

    public static void main(String[] args) throws InterruptedException {
        RateLimiter rateLimiter = RateLimiter.create(5); //设置QPS为5
        for (int i = 0; i < 10; i++) {
            rateLimiter.acquire();
            System.out.println(System.currentTimeMillis());
        }
        System.out.println("-----");
        //可以随时调整速率，我们将qps调整为10
        rateLimiter.setRate(10);
        for (int i = 0; i < 10; i++) {
            rateLimiter.acquire();
            System.out.println(System.currentTimeMillis());
        }
    }
}
```

输出：

```
1566284028725
1566284028922
1566284029121
1566284029322
1566284029522
1566284029721
1566284029921
1566284030122
1566284030322
1566284030522
-----
1566284030722
1566284030822
1566284030921
1566284031022
1566284031121
1566284031221
```

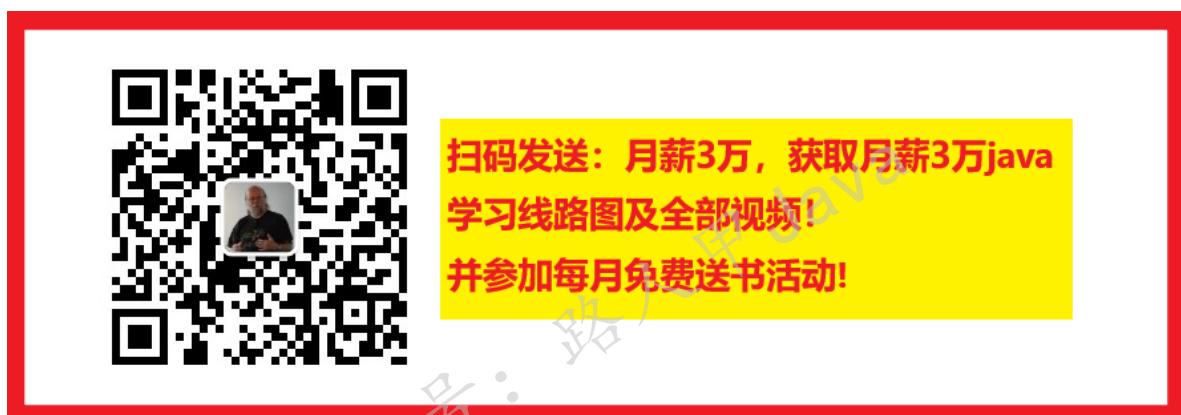
1566284031321
1566284031422
1566284031522
1566284031622

代码中 `RateLimiter.create(5)` 创建QPS为5的限流对象，后面又调用 `rateLimiter.setRate(10);` 将速率设为10，输出中分2段，第一段每次输出相隔200毫秒，第二段每次输出相隔100毫秒，可以非常精准的控制系统的QPS。

上面介绍的这些，业务中可能会用到，也可以用来应对面试。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第30篇：JUC中的CompletableFuture

环境：jkd1.8。

CompletableFuture是java8中新增的一个类，算是对Future的一种增强，用起来很方便，也是会经常用到的一个工具类，熟悉一下。

CompletionStage接口

- CompletionStage代表异步计算过程中的某一个阶段，一个阶段完成以后可能会触发另外一个阶段
- 一个阶段的计算执行可以是一个Function，Consumer或者Runnable。比如：`stage.thenApply(x -> square(x)).thenAccept(x -> System.out.print(x)).thenRun(() -> System.out.println())`

加微信itsoku，发送：1024，获取 10T 高质量计算机学习视频！！

- 一个阶段的执行可能是被单个阶段的完成触发，也可能是由多个阶段一起触发

CompletableFuture类

- 在Java8中，CompletableFuture提供了非常强大的Future的扩展功能，可以帮助我们简化异步编程的复杂性，并且提供了函数式编程的能力，可以通过回调的方式处理计算结果，也提供了转换和组合CompletableFuture的方法。
- 它可能代表一个明确完成的Future，也有可能代表一个完成阶段（CompletionStage），它支持在计算完成以后触发一些函数或执行某些动作。
- 它实现了Future和CompletionStage接口

常见的方法，熟悉一下：

runAsync 和 supplyAsync方法

CompletableFuture 提供了四个静态方法来创建一个异步操作。

```
public static CompletableFuture<Void> runAsync(Runnable runnable)
public static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor)
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)
```

没有指定Executor的方法会使用ForkJoinPool.commonPool() 作为它的线程池执行异步代码。如果指定线程池，则使用指定的线程池运行。以下所有的方法都类同。

- runAsync方法不支持返回值。
- supplyAsync可以支持返回值。

示例代码

```
//无返回值
public static void runAsync() throws Exception {
    CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
        }
        System.out.println("run end ...");
    });

    future.get();
}

//有返回值
public static void supplyAsync() throws Exception {
    CompletableFuture<Long> future = CompletableFuture.supplyAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
        }
        System.out.println("run end ...");
    });
}
```

```

        return System.currentTimeMillis();
    });

    long time = future.get();
    System.out.println("time = "+time);
}

```

计算结果完成时的回调方法

当CompletableFuture的计算结果完成，或者抛出异常的时候，可以执行特定的Action。主要是下面的方法：

```

public CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable>
action)
public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T, ? super
Throwable> action)
public CompletableFuture<T> whenCompleteAsync(BiConsumer<? super T, ? super
Throwable> action, Executor executor)
public CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn)

```

可以看到Action的类型是BiConsumer<? super T,? super Throwable>它可以处理正常的计算结果，或者异常情况。

whenComplete 和 whenCompleteAsync 的区别：

- whenComplete：是执行当前任务的线程执行继续执行 whenComplete 的任务。
- whenCompleteAsync：是执行把 whenCompleteAsync 这个任务继续提交给线程池来进行执行。

示例代码

```

public static void whenComplete() throws Exception {
    CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
        }
        if(new Random().nextInt()%2>=0) {
            int i = 12/0;
        }
        System.out.println("run end ...");
    });

    future.whenComplete(new BiConsumer<Void, Throwable>() {
        @Override
        public void accept(Void t, Throwable action) {
            System.out.println("执行完成! ");
        }
    });

    future.exceptionally(new Function<Throwable, Void>() {
        @Override
        public Void apply(Throwable t) {
            System.out.println("执行失败! "+t.getMessage());
            return null;
        }
    });
}

```

```
        TimeUnit.SECONDS.sleep(2);
    }
```

thenApply 方法

当一个线程依赖另一个线程时，可以使用 thenApply 方法来把这两个线程串行化。

```
public <U> CompletableFuture<U> thenApply(Function<? super T, ? extends U> fn)
public <U> CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn)
public <U> CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn, Executor executor)
```

Function<? super T, ? extends U> T: 上一个任务返回结果的类型 U: 当前任务的返回值类型

示例代码

```
private static void thenApply() throws Exception {
    CompletableFuture<Long> future = CompletableFuture.supplyAsync(new
supplier<Long>() {
    @Override
    public Long get() {
        long result = new Random().nextInt(100);
        System.out.println("result1="+result);
        return result;
    }
}).thenApply(new Function<Long, Long>() {
    @Override
    public Long apply(Long t) {
        long result = t*5;
        System.out.println("result2="+result);
        return result;
    }
});

long result = future.get();
System.out.println(result);
}
```

第二个任务依赖第一个任务的结果。

handle 方法

handle 是执行任务完成时对结果的处理。 handle 方法和 thenApply 方法处理方式基本一样。不同的是 handle 是在任务完成后再执行，还可以处理异常的任务。 thenApply 只可以执行正常的任务，任务出现异常则不执行 thenApply 方法。

```
public <U> CompletionStage<U> handle(BiFunction<? super T, Throwable, ? extends
U> fn);
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ?
extends U> fn);
public <U> CompletionStage<U> handleAsync(BiFunction<? super T, Throwable, ?
extends U> fn,Executor executor);
```

示例代码

```
public static void handle() throws Exception{
    CompletableFuture<Integer> future = CompletableFuture.supplyAsync(new
Supplier<Integer>() {

    @Override
    public Integer get() {
        int i= 10/0;
        return new Random().nextInt(10);
    }
}).handle(new BiFunction<Integer, Throwable, Integer>() {
    @Override
    public Integer apply(Integer param, Throwable throwable) {
        int result = -1;
        if(throwable==null){
            result = param * 2;
        }else{
            System.out.println(throwable.getMessage());
        }
        return result;
    }
});
System.out.println(future.get());
}
```

从示例中可以看出，在 handle 中可以根据任务是否有异常来进行做相应的后续处理操作。而 thenApply 方法，如果上个任务出现错误，则不会执行 thenApply 方法。

thenAccept 消费处理结果

接收任务的处理结果，并消费处理，无返回结果。

```
public CompletionStage<Void> thenAccept(Consumer<? super T> action);
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action);
public CompletionStage<Void> thenAcceptAsync(Consumer<? super T> action,Executor
executor);
```

示例代码

```
public static void thenAccept() throws Exception{
    CompletableFuture<Void> future = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        return new Random().nextInt(10);
    }
}).thenAccept(integer -> {
    System.out.println(integer);
});
future.get();
}
```

从示例代码中可以看出，该方法只是消费执行完成的任务，并可以根据上面的任务返回的结果进行处理。并没有后续的输错操作。

thenRun 方法

跟 thenAccept 方法不一样的是，不关心任务的处理结果。只要上面的任务执行完成，就开始执行 thenAccept。

```
public CompletionStage<Void> thenRun(Runnable action);
public CompletionStage<Void> thenRunAsync(Runnable action);
public CompletionStage<Void> thenRunAsync(Runnable action, Executor executor);
```

示例代码

```
public static void thenRun() throws Exception{
    CompletableFuture<Void> future = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        return new Random().nextInt(10);
    }
}).thenRun(() -> {
    System.out.println("thenRun ...");
});
future.get();
}
```

该方法同 thenAccept 方法类似。不同的是上个任务处理完成后，并不会把计算的结果传给 thenRun 方法。只是处理完任务后，执行 thenAccept 的后续操作。

thenCombine 合并任务

thenCombine 会把两个 CompletionStage 的任务都执行完成后，把两个任务的结果一块交给 thenCombine 来处理。

```
public <U,V> CompletionStage<V> thenCombine(CompletionStage<? extends U>
other,BiFunction<? super T,>? super U,>? extends V> fn);
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends U>
other,BiFunction<? super T,>? super U,>? extends V> fn);
public <U,V> CompletionStage<V> thenCombineAsync(CompletionStage<? extends U>
other,BiFunction<? super T,>? super U,>? extends V> fn,Executor executor);
```

示例代码

```
private static void thenCombine() throws Exception {
    CompletableFuture<String> future1 = CompletableFuture.supplyAsync(new
Supplier<String>() {
    @Override
    public String get() {
        return "hello";
    }
});
CompletableFuture<String> future2 = CompletableFuture.supplyAsync(new
Supplier<String>() {
    @Override
    public String get() {
        return "hello";
    }
});
```

```

    });
    CompletableFuture<String> result = future1.thenCombine(future2, new
BiFunction<String, String, String>() {
    @Override
    public String apply(String t, String u) {
        return t+" "+u;
    }
});
System.out.println(result.get());
}

```

thenAcceptBoth

当两个CompletionStage都执行完成后，把结果一块交给thenAcceptBoth来进行消耗

```

public <U> CompletionStage<Void> thenAcceptBoth(CompletionStage<? extends U>
other,BiConsumer<? super T, ? super U> action);
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? extends
U> other,BiConsumer<? super T, ? super U> action);
public <U> CompletionStage<Void> thenAcceptBothAsync(CompletionStage<? extends
U> other,BiConsumer<? super T, ? super U> action,      Executor executor);

```

示例代码

```

private static void thenAcceptBoth() throws Exception {
    CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        int t = new Random().nextInt(3);
        try {
            TimeUnit.SECONDS.sleep(t);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("f1="+t);
        return t;
    }
});

CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        int t = new Random().nextInt(3);
        try {
            TimeUnit.SECONDS.sleep(t);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("f2="+t);
        return t;
    }
});
f1.thenAcceptBoth(f2, new BiConsumer<Integer, Integer>() {
    @Override

```

```

        public void accept(Integer t, Integer u) {
            System.out.println("f1="+t+";f2="+u+";");
        }
    );
}

```

applyToEither 方法

两个CompletionStage，谁执行返回的结果快，我就用那个CompletionStage的结果进行下一步的转化操作。

```

public <U> CompletionStage<U> applyToEither(CompletionStage<? extends T>
other,Function<? super T, U> fn);
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends T>
other,Function<? super T, U> fn);
public <U> CompletionStage<U> applyToEitherAsync(CompletionStage<? extends T>
other,Function<? super T, U> fn,Executor executor);

```

示例代码

```

private static void applyToEither() throws Exception {
    CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        int t = new Random().nextInt(3);
        try {
            TimeUnit.SECONDS.sleep(t);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("f1="+t);
        return t;
    }
});
CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        int t = new Random().nextInt(3);
        try {
            TimeUnit.SECONDS.sleep(t);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("f2="+t);
        return t;
    }
});

CompletableFuture<Integer> result = f1.applyToEither(f2, new
Function<Integer, Integer>() {
    @Override
    public Integer apply(Integer t) {
        System.out.println(t);
        return t * 2;
    }
});

```

```

        }
    });

    System.out.println(result.get());
}

```

acceptEither 方法

两个CompletionStage，谁执行返回的结果快，我就用那个CompletionStage的结果进行下一步的消耗操作。

```

public CompletionStage<Void> acceptEither(CompletionStage<? extends T>
other, Consumer<? super T> action);
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T>
other, Consumer<? super T> action);
public CompletionStage<Void> acceptEitherAsync(CompletionStage<? extends T>
other, Consumer<? super T> action, Executor executor);

```

示例代码

```

private static void acceptEither() throws Exception {
    CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        int t = new Random().nextInt(3);
        try {
            TimeUnit.SECONDS.sleep(t);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("f1="+t);
        return t;
    }
});

CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(new
Supplier<Integer>() {
    @Override
    public Integer get() {
        int t = new Random().nextInt(3);
        try {
            TimeUnit.SECONDS.sleep(t);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("f2="+t);
        return t;
    }
});
f1.acceptEither(f2, new Consumer<Integer>() {
    @Override
    public void accept(Integer t) {
        System.out.println(t);
    }
});
}

```

```
}
```

runAfterEither 方法

两个CompletionStage，任何一个完成了都会执行下一步的操作（Runnable）

```
public CompletionStage<Void> runAfterEither(CompletionStage<?> other, Runnable action);
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other, Runnable action);
public CompletionStage<Void> runAfterEitherAsync(CompletionStage<?> other, Runnable action, Executor executor);
```

示例代码

```
private static void runAfterEither() throws Exception {
    CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            int t = new Random().nextInt(3);
            try {
                TimeUnit.SECONDS.sleep(t);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("f1=" + t);
            return t;
        }
    });

    CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            int t = new Random().nextInt(3);
            try {
                TimeUnit.SECONDS.sleep(t);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("f2=" + t);
            return t;
        }
    });
    f1.runAfterEither(f2, new Runnable() {

        @Override
        public void run() {
            System.out.println("上面有一个已经完成了。");
        }
    });
}
```

runAfterBoth

两个CompletionStage，都完成了计算才会执行下一步的操作（Runnable）

```
public CompletionStage<Void> runAfterBoth(CompletionStage<?> other, Runnable action);
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other, Runnable action);
public CompletionStage<Void> runAfterBothAsync(CompletionStage<?> other, Runnable action, Executor executor);
```

示例代码

```
private static void runAfterBoth() throws Exception {
    CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            int t = new Random().nextInt(3);
            try {
                TimeUnit.SECONDS.sleep(t);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("f1=" + t);
            return t;
        }
    });

    CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            int t = new Random().nextInt(3);
            try {
                TimeUnit.SECONDS.sleep(t);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("f2=" + t);
            return t;
        }
    });
    f1.runAfterBoth(f2, new Runnable() {
        @Override
        public void run() {
            System.out.println("上面两个任务都执行完成了。");
        }
    });
}
```

thenCompose 方法

thenCompose 方法允许你对两个 CompletionStage 进行流水线操作，第一个操作完成时，将其结果作为参数传递给第二个操作。

```
public <U> CompletableFuture<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn);
public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn) ;
public <U> CompletableFuture<U> thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn, Executor executor) ;
```

示例代码

```
private static void thenCompose() throws Exception {
    CompletableFuture<Integer> f = CompletableFuture.supplyAsync(new Supplier<Integer>() {
        @Override
        public Integer get() {
            int t = new Random().nextInt(3);
            System.out.println("t1=" + t);
            return t;
        }
    }).thenCompose(new Function<Integer, CompletionStage<Integer>>() {
        @Override
        public CompletionStage<Integer> apply(Integer param) {
            return CompletableFuture.supplyAsync(new Supplier<Integer>() {
                @Override
                public Integer get() {
                    int t = param * 2;
                    System.out.println("t2=" + t);
                    return t;
                }
            });
        }
    });
    System.out.println("thenCompose result : " + f.get());
}
```

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第31篇：等待线程完成的方式你知道几种？

环境：jdk1.8。

java高并发系列已经学了不少东西了，本篇文章，我们用前面学的知识来实现一个需求：

在一个线程中需要获取其他线程的执行结果，能想到几种方式？各有什么优缺点？

结合这个需求，我们使用**6种方式**，来对之前学过的知识点做一个回顾，加深记忆。

方式1：Thread的join()方法实现

代码：

```
package com.itsoku.chat31;

import java.sql.Time;
import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    //用于封装结果
    static class Result<T> {
        T result;

        public T getResult() {
            return result;
        }

        public void setResult(T result) {
            this.result = result;
        }
    }

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

加微信itsoku，发送：1024，获取10T高质量计算机学习视频！！

```

        System.out.println(System.currentTimeMillis());
        //用于存放子线程执行的结果
        Result<Integer> result = new Result<>();
        //创建一个子线程
        Thread thread = new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(3);
                result.setResult(10);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        thread.start();
        //让主线程等待thread线程执行完毕之后再继续，join方法会让当前线程阻塞
        thread.join();

        //获取thread线程的执行结果
        Integer rs = result.getResult();
        System.out.println(System.currentTimeMillis());
        System.out.println(System.currentTimeMillis() + ":" + rs);
    }
}

```

输出：

```

1566733162636
1566733165692
1566733165692:10

```

代码中通过join方式阻塞了当前主线程，当thread线程执行完毕之后，join方法才会继续执行。

关于join()方法和线程更详细的使用，可以参考：[线程的基本操作](#)

方式2：CountDownLatch实现

代码：

```

package com.itsoku.chat31;

import java.util.concurrent.*;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo2 {
    //用于封装结果
    static class Result<T> {
        T result;

        public T getResult() {
            return result;
        }

        public void setResult(T result) {
            this.result = result;
        }
    }
}

```

```

    }

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
    System.out.println(System.currentTimeMillis());
    CountDownLatch countDownLatch = new CountDownLatch(1);
    //用于存放子线程执行的结果
    Demo1.Result<Integer> result = new Demo1.Result<>();
    //创建一个子线程
    Thread thread = new Thread(() -> {
        try {
            TimeUnit.SECONDS.sleep(3);
            result.setResult(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            countDownLatch.countDown();
        }
    });
    thread.start();
    //countDownLatch.await()会让当前线程阻塞，当countDownLatch中的计数器变为0的时候，await方法会返回
    countDownLatch.await();

    //获取thread线程的执行结果
    Integer rs = result.getResult();
    System.out.println(System.currentTimeMillis());
    System.out.println(System.currentTimeMillis() + ":" + rs);
}
}

```

输出：

```

1566733720406
1566733723453
1566733723453:10

```

上面代码也达到了预期效果，使用 `CountDownLatch` 可以让一个或者多个线程等待一批线程完成之后，自己再继续；`CountDownLatch` 更详细的介绍见：[JUC中等待多线程完成的工具类CountDownLatch，必备技能](#)

方式3：ExecutorService.submit方法实现

代码：

```

package com.itsoku.chat31;

import java.util.concurrent.*;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo3 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {

```

```

//创建一个线程池
ExecutorService executorService = Executors.newCachedThreadPool();
System.out.println(System.currentTimeMillis());
Future<Integer> future = executorService.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(3);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return 10;
});
//关闭线程池
executorService.shutdown();
System.out.println(System.currentTimeMillis());
Integer result = future.get();
System.out.println(System.currentTimeMillis() + ":" + result);
}
}

```

输出：

```

1566734119938
1566734119989
1566734122989:10

```

使用 `ExecutorService.submit` 方法实现的，此方法返回一个 `Future`，`future.get()` 会让当前线程阻塞，直到 Future 关联的任务执行完毕。

相关知识：

1. [JAVA线程池，这一篇就够了](#)
2. [JUC中的Executor框架详解1](#)
3. [JUC中的Executor框架详解2](#)

方式4：FutureTask方式1

代码：

```

package com.itsoku.chat31;

import java.util.concurrent.*;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        System.out.println(System.currentTimeMillis());
        //创建一个FutureTask
        FutureTask<Integer> futureTask = new FutureTask<>(() -> {
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {

```

```

        e.printStackTrace();
    }
    return 10;
});
//将futureTask传递一个线程运行
new Thread(futureTask).start();
System.out.println(System.currentTimeMillis());
//futureTask.get()会阻塞当前线程，直到futureTask执行完毕
Integer result = futureTask.get();
System.out.println(System.currentTimeMillis() + ":" + result);
}
}

```

输出：

```

1566736350314
1566736350358
1566736353360:10

```

代码中使用 `FutureTask` 实现的，`FutureTask`实现了 `Runnable` 接口，并且内部带返回值，所以可以传递给 `Thread` 直接运行，`futureTask.get()` 会阻塞当前线程，直到 `FutureTask` 构造方法传递的任务执行完毕，`get` 方法才会返回。关于 `FutureTask` 详细使用，请参考：[JUC中的Executor框架详解1](#)

方式5：FutureTask方式2

代码：

```

package com.itsoku.chat31;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo5 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        System.out.println(System.currentTimeMillis());
        //创建一个FutureTask
        FutureTask<Integer> futureTask = new FutureTask<>(() -> 10);
        //将futureTask传递一个线程运行
        new Thread(() -> {
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            futureTask.run();
        }).start();
        System.out.println(System.currentTimeMillis());
        //futureTask.get()会阻塞当前线程，直到futureTask执行完毕
        Integer result = futureTask.get();
    }
}

```

```
        System.out.println(system.currentTimeMillis() + ":" + result);
    }
}
```

输出：

```
1566736319925
1566736319970
1566736322972:10
```

创建了一个 FutureTask 对象，调用 futureTask.get() 会阻塞当前线程，子线程中休眠了3秒，然后调用 futureTask.run(); 当futureTask的run()方法执行完毕之后，futureTask.get() 会从阻塞中返回。

注意：这种方式和方式4的不同点。

关于 FutureTask 详细使用，请参考：[JUC中的Executor框架详解1](#)

方式6：CompletableFuture方式实现

代码：

```
package com.itsoku.chat31;

import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo6 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        System.out.println(system.currentTimeMillis());
        CompletableFuture<Integer> completableFuture =
        CompletableFuture.supplyAsync(() -> {
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return 10;
        });
        System.out.println(system.currentTimeMillis());
        //futureTask.get() 会阻塞当前线程，直到futureTask执行完毕
        Integer result = completableFuture.get();
        System.out.println(system.currentTimeMillis() + ":" + result);
    }
}
```

输出：

1566736205348
1566736205428
1566736208429:10

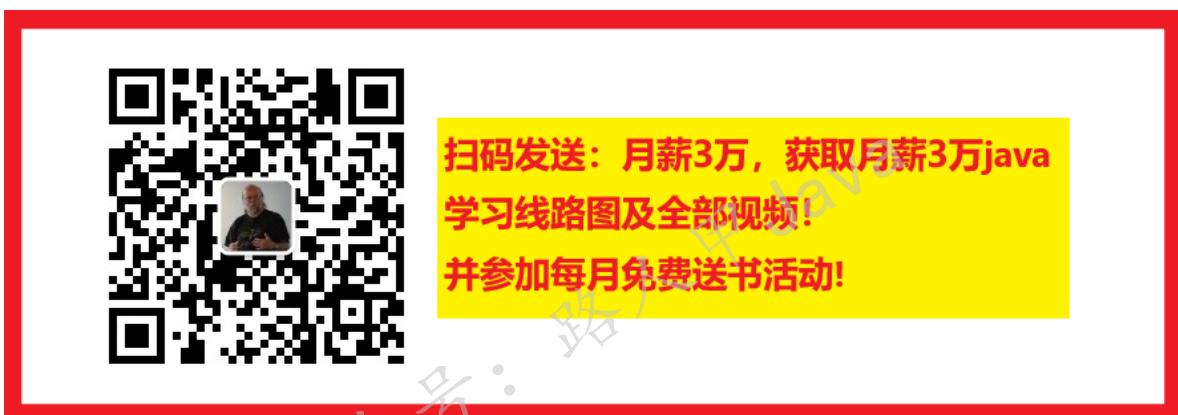
`CompletableFuture.supplyAsync` 可以用来异步执行一个带返回值的任务，调用 `CompletableFuture.get()`

会阻塞当前线程，直到任务执行完毕，`get`方法才会返回。

关于 `CompletableFuture` 更详细的使用见：[JUC中工具类CompletableFuture，必备技能](#)

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第32篇：原子操作增强类LongAdder、LongAccumulator

java环境：jdk1.8。

本文主要内容

1. 4种方式实现计数器功能，对比其性能
2. 介绍LongAdder
3. 介绍LongAccumulator

来个需求

一个jvm中实现一个计数器功能，需保证多线程情况下数据正确性。

我们来模拟50个线程，每个线程对计数器递增100万次，最终结果应该是5000万。

我们使用4种方式实现，看一下其性能，然后引出为什么需要使用 LongAdder、LongAccumulator。

方式一：synchronized方式实现

```
package com.itsoku.chat32;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.LongAccumulator;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo1 {
    static int count = 0;

    public static synchronized void incr() {
        count++;
    }

    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        for (int i = 0; i < 10; i++) {
            count = 0;
            m1();
        }
    }

    private static void m1() throws InterruptedException {
        long t1 = System.currentTimeMillis();
        int threadCount = 50;
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 1000000; j++) {
                        incr();
                    }
                } finally {
                    countDownLatch.countDown();
                }
            }).start();
        }
        countDownLatch.await();
        long t2 = System.currentTimeMillis();
    }
}
```

```
        System.out.println(String.format("结果: %s, 耗时(ms): %s", count, (t2 - t1)));
    }
}
```

输出：

```
结果: 50000000, 耗时(ms): 1437
结果: 50000000, 耗时(ms): 1913
结果: 50000000, 耗时(ms): 386
结果: 50000000, 耗时(ms): 383
结果: 50000000, 耗时(ms): 381
结果: 50000000, 耗时(ms): 382
结果: 50000000, 耗时(ms): 379
结果: 50000000, 耗时(ms): 379
结果: 50000000, 耗时(ms): 392
结果: 50000000, 耗时(ms): 384
```

平均耗时：390毫秒

方式2：AtomicLong实现

```
package com.itsoku.chat32;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.AtomicLong;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo2 {
    static AtomicLong count = new AtomicLong(0);

    public static void incr() {
        count.incrementAndGet();
    }

    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        for (int i = 0; i < 10; i++) {
            count.set(0);
            m1();
        }
    }

    private static void m1() throws InterruptedException {
        long t1 = System.currentTimeMillis();
        int threadCount = 50;
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 1000000; j++) {
                        incr();
                    }
                } catch (Exception e) {
                }
            }).start();
        }
        countDownLatch.await();
        System.out.println("耗时：" + (System.currentTimeMillis() - t1));
    }
}
```

```

        } finally {
            countDownLatch.countDown();
        }
    }).start();
}
countDownLatch.await();
long t2 = System.currentTimeMillis();
System.out.println(String.format("结果: %s, 耗时(ms): %s", count, (t2 - t1)));
}
}

```

输出：

```

结果: 50000000,耗时(ms): 971
结果: 50000000,耗时(ms): 915
结果: 50000000,耗时(ms): 920
结果: 50000000,耗时(ms): 923
结果: 50000000,耗时(ms): 910
结果: 50000000,耗时(ms): 916
结果: 50000000,耗时(ms): 923
结果: 50000000,耗时(ms): 916
结果: 50000000,耗时(ms): 912
结果: 50000000,耗时(ms): 908

```

平均耗时：920毫秒

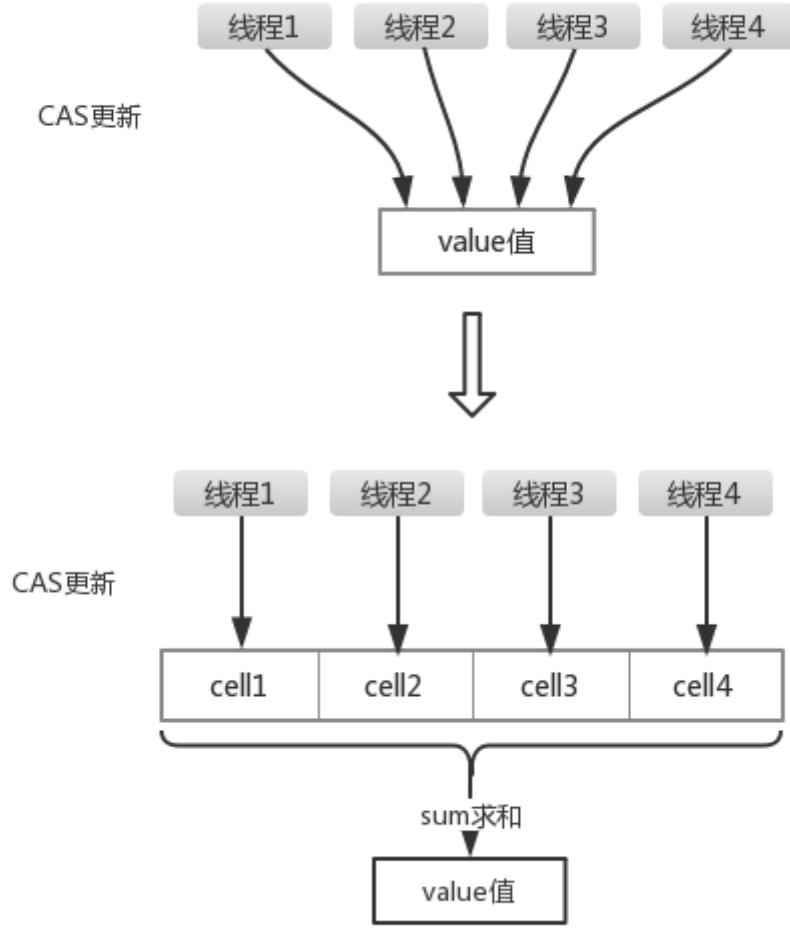
AtomicLong 内部采用CAS的方式实现，并发量大的情况下，CAS失败率比较高，导致性能比 synchronized还低一些。并发量不是太大的情况下，CAS性能还是可以的。

AtomicLong 属于JUC中的原子类，还不是很熟悉的可以看一下：[JUC中原子类，一篇就够了](#)

方式3：LongAdder实现

先介绍一下 LongAdder，说到LongAdder，不得不提的就是AtomicLong，AtomicLong是JDK1.5开始出现的，里面主要使用了一个long类型的value作为成员变量，然后使用循环的CAS操作去操作value的值，并发量比较大的情况下，CAS操作失败的概率较高，内部失败了会重试，导致耗时可能会增加。

LongAdder是JDK1.8开始出现的，所提供的API基本上可以替换掉原先的AtomicLong。LongAdder在并发量比较大的情况下，操作数据的时候，相当于把这个数字分成了很多份数字，然后交给多人去管控，每个管控者负责保证部分数字在多线程情况下操作的正确性。当多线程访问的时，通过hash算法映射到具体管控者去操作数据，最后再汇总所有的管控者的数据，得到最终结果。相当于降低了并发情况下锁的粒度，所以效率比较高，看一下下面的图，方便理解：



代码：

```

package com.itsoku.chat32;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.AtomicLong;
import java.util.concurrent.atomic.LongAdder;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
public class Demo3 {
    static LongAdder count = new LongAdder();

    public static void incr() {
        count.increment();
    }

    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        for (int i = 0; i < 10; i++) {
            count.reset();
            m1();
        }
    }
}

```

```

private static void m1() throws ExecutionException, InterruptedException {
    long t1 = System.currentTimeMillis();
    int threadCount = 50;
    CountDownLatch countDownLatch = new CountDownLatch(threadCount);
    for (int i = 0; i < threadCount; i++) {
        new Thread(() -> {
            try {
                for (int j = 0; j < 1000000; j++) {
                    incr();
                }
            } finally {
                countDownLatch.countDown();
            }
        }).start();
    }
    countDownLatch.await();
    long t2 = System.currentTimeMillis();
    System.out.println(String.format("结果: %s, 耗时(ms): %s", count.sum(), (t2 - t1)));
}
}

```

输出：

```

结果: 50000000,耗时(ms): 206
结果: 50000000,耗时(ms): 105
结果: 50000000,耗时(ms): 107
结果: 50000000,耗时(ms): 107
结果: 50000000,耗时(ms): 105
结果: 50000000,耗时(ms): 99
结果: 50000000,耗时(ms): 106
结果: 50000000,耗时(ms): 102
结果: 50000000,耗时(ms): 106
结果: 50000000,耗时(ms): 102

```

平均耗时：100毫秒

代码中 `new LongAdder()` 创建一个 `LongAdder` 对象，内部数字初始值是 0，调用 `increment()` 方法可以对 `LongAdder` 内部的值原子递增 1。`reset()` 方法可以重置 `LongAdder` 的值，使其归 0。

方式4：LongAccumulator实现

LongAccumulator介绍

`LongAccumulator` 是 `LongAdder` 的功能增强版。`LongAdder` 的 API 只有对数值的加减，而 `LongAccumulator` 提供了自定义的函数操作，其构造函数如下：

```

/**
 * accumulatorFunction: 需要执行的二元函数（接收2个long作为形参，并返回1个long）
 * identity: 初始值
 */
public LongAccumulator(LongBinaryOperator accumulatorFunction, long identity) {
    this.function = accumulatorFunction;
    base = this.identity = identity;
}

```

示例代码：

```

package com.itsoku.chat32;

import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.atomic.LongAccumulator;
import java.util.concurrent.atomic.LongAdder;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
public class Demo4 {
    static LongAccumulator count = new LongAccumulator((x, y) -> x + y, 0L);

    public static void incr() {
        count.accumulate(1);
    }

    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        for (int i = 0; i < 10; i++) {
            count.reset();
            m1();
        }
    }

    private static void m1() throws ExecutionException, InterruptedException {
        long t1 = System.currentTimeMillis();
        int threadCount = 50;
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        for (int i = 0; i < threadCount; i++) {
            new Thread(() -> {
                try {
                    for (int j = 0; j < 1000000; j++) {
                        incr();
                    }
                } finally {
                    countDownLatch.countDown();
                }
            }).start();
        }
        countDownLatch.await();
        long t2 = System.currentTimeMillis();
        System.out.println(String.format("结果: %s, 耗时(ms): %s",
        count.longValue(), (t2 - t1)));
    }
}

```

```
}
```

输出：

```
结果: 50000000,耗时(ms): 138
结果: 50000000,耗时(ms): 111
结果: 50000000,耗时(ms): 111
结果: 50000000,耗时(ms): 103
结果: 50000000,耗时(ms): 103
结果: 50000000,耗时(ms): 105
结果: 50000000,耗时(ms): 101
结果: 50000000,耗时(ms): 106
结果: 50000000,耗时(ms): 102
结果: 50000000,耗时(ms): 103
```

平均耗时：100毫秒

`LongAccumulator` 的效率和 `LongAdder` 差不多，不过更灵活一些。

调用 `new LongAdder()` 等价于 `new LongAccumulator((x, y) -> x + y, 0L)`。

从上面4个示例的结果来看，`LongAdder`、`LongAccumulator` 全面超越同步锁及 `AtomicLong` 的方式，建议在使用 `AtomicLong` 的地方可以直接替换为 `LongAdder`、`LongAccumulator`，吞吐量更高一些。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySql高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



加微信itsoku，发送：1024，获取 100G 高质量计算机学习视频！！

第33篇：怎么演示公平锁和非公平锁

加微信itsoku，发送：1024，获取 10T 高质量计算机学习视频！！

环境：jdk1.8。

今天群里面刚有有人在问这块的东西，那就拿出来说一下。

本文主要用juc中的 ReentrantLock 来说一下公平锁和非公平锁的东西。

先理解一下什么是公平锁、非公平锁？

公平锁和非公平锁体现在别人释放锁的一瞬间，如果前面已经有排队的，新来的是否可以插队，如果可以插队表示是非公平的，如果不可用插队，只能排在最后面，是公平的方式。

示例

测试公平锁和非公平锁的时候，可以这么来：在主线程中先启动一个t1线程，在t1里面获取锁，获取锁之后休眠一会，然后在主线程中启动10个father线程去排队获取锁，然后在t1中释放锁代码的前面一步再启动一个线程，在这个线程内部再创建10个son线程，去获取锁，看看后面这10个son线程会不会排到上面10个father线程前面去，如果会表示插队了，说明是非公平的，如果不插队，表示排队执行的，说明是公平的方式，示例代码如下：

```
package com.itsoku.chat32;

import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
@Slf4j
public class Demo8 {
    public static void main(String[] args) throws InterruptedException {
        //非公平锁
        test1(false);
        TimeUnit.SECONDS.sleep(4);
        log.info("-----");
        //公平锁
        test1(true);
    }

    public static void test1(boolean fair) throws InterruptedException {
        ReentrantLock lock = new ReentrantLock(fair);
        Thread t1 = new Thread(() -> {
            lock.lock();
            try {
                log.info("start");
                TimeUnit.SECONDS.sleep(3);
                new Thread(() -> {
                    m1(lock, "son");
                }).start();
                log.info("end");
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                lock.unlock();
            }
        });
    }
}
```

```

        });
        t1.setName("t1");
        t1.start();
        TimeUnit.SECONDS.sleep(1);
        m1(lock, "father");
    }

    public static void m1(ReentrantLock lock, String threadPre) {
        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(() -> {
                lock.lock();
                try {
                    log.info("获取到锁！");
                } finally {
                    lock.unlock();
                }
            });
            thread.setName(threadPre + "-" + i);
            thread.start();
        }
    }
}

```

输出：

```

10:16:02.132 [t1] INFO com.itsoku.chat32.Demo8 - start
10:16:05.135 [t1] INFO com.itsoku.chat32.Demo8 - end
10:16:05.135 [father-0] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.136 [father-1] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.136 [father-2] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.136 [son-2] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.136 [father-3] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.137 [father-4] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.137 [father-5] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.137 [son-5] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.137 [father-6] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.137 [father-7] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.137 [father-8] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [father-9] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [son-0] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [son-1] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [son-3] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [son-4] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [son-6] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.138 [son-7] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.139 [son-8] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:05.139 [son-9] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:07.129 [main] INFO com.itsoku.chat32.Demo8 - -----
-
10:16:07.129 [t1] INFO com.itsoku.chat32.Demo8 - start
10:16:10.130 [t1] INFO com.itsoku.chat32.Demo8 - end
10:16:10.130 [father-0] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.130 [father-1] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.130 [father-2] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.131 [father-3] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.131 [father-4] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.131 [father-5] INFO com.itsoku.chat32.Demo8 - 获取到锁!

```

```
10:16:10.131 [father-6] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.132 [father-7] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.132 [father-8] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.132 [father-9] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.132 [son-1] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.132 [son-0] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.132 [son-2] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.133 [son-4] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.133 [son-3] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.133 [son-5] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.133 [son-6] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.133 [son-7] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.133 [son-8] INFO com.itsoku.chat32.Demo8 - 获取到锁!
10:16:10.135 [son-9] INFO com.itsoku.chat32.Demo8 - 获取到锁!
```

运行代码可以创建一个springboot项目，需要安装lombok插件

上面代码中以 son 开头的线程在 father 线程之后启动的，分析一下结果：

`test1(false);` 执行的是非公平锁的过程，看一下 son 的输出排到 father 前面去了，说明插队了，说明采用的是非公平锁的方式。

`test1(true);` 执行的是公平锁的过程，看一下输出， son 都是在 father 后面输出的，说明排队执行的，说明采用的是公平锁的方式。

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！

加微信itsoku，发送：1024，获取100G高质量计算机学习视频！！

第34篇：谷歌提供的一些好用的并发工具类

环境：jdk1.8。

关于并发方面的，juc已帮我们提供了很多好用的工具，而谷歌在此基础上做了扩展，使并发编程更容易，这些工具放在guava.jar包中。

本文演示几个简单的案例，见一下guava的效果。

需要先了解的一些技术：[juc中的线程池](#)、[Excecutors](#)、[ExecutorService](#)、[Callable](#)、[Future](#)

guava maven配置

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>27.0-jre</version>
</dependency>
```

guava中常用几个类

MoreExecutors：提供了一些静态方法，是对juc中的Executors类的一个扩展。 **Futures**：也提供了很多静态方法，是对juc中Future的一个扩展。

案例1：异步执行任务完毕之后回调

```
package com.itsoku.chat34;

import com.google.common.util.concurrent.ListenableFuture;
import com.google.common.util.concurrent.ListeningExecutorService;
import com.google.common.util.concurrent.MoreExecutors;
import lombok.extern.slf4j.Slf4j;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
@Slf4j
public class Demo1 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        //创建一个线程池
        ExecutorService delegate = Executors.newFixedThreadPool(5);
        try {
            ListeningExecutorService executorService =
            MoreExecutors.listeningDecorator(delegate);
            //异步执行一个任务
            ListenableFuture<Integer> submit = executorService.submit(() -> {
```

```
        log.info("{}", System.currentTimeMillis());
        //休眠2秒， 默认耗时
        TimeUnit.SECONDS.sleep(2);
        log.info("{}", System.currentTimeMillis());
        return 10;
    });
    //当任务执行完毕之后回调对应的方法
    submit.addListener(() -> {
        log.info("任务执行完毕了，我被回调了");
    }, MoreExecutors.directExecutor());
    log.info("{}", submit.get());
} finally {
    delegate.shutdown();
}
}
```

输出：

```
14:25:50.055 [pool-1-thread-1] INFO com.itsoku.chat34.Demo1 - 1567491950047  
14:25:52.063 [pool-1-thread-1] INFO com.itsoku.chat34.Demo1 - 1567491952063  
14:25:52.064 [pool-1-thread-1] INFO com.itsoku.chat34.Demo1 - 任务执行完毕了，我被回调了  
14:25:52.064 [main] INFO com.itsoku.chat34.Demo1 - 10
```

说明：

`ListeningExecutorService` 接口继承于 juc 中的 `ExecutorService` 接口，对 `ExecutorService` 做了一些扩展，看其名字中带有 `Listening`，说明这个接口自带监听的功能，可以监听异步执行任务的结果。通过 `MoreExecutors.listeningDecorator` 创建一个 `ListeningExecutorService` 对象，需传递一个 `ExecutorService` 参数，传递的 `ExecutorService` 负责异步执行任务。

`ListeningExecutorService` 的 `submit` 方法用来异步执行一个任务，返回 `ListenableFuture`，`ListenableFuture` 接口继承于 juc 中的 `Future` 接口，对 `Future` 做了扩展，使其带有监听的功能。调用 `submit.addListener` 可以在执行的任务上添加监听器，当任务执行完毕之后会回调这个监听器中的方法。

`ListenableFuture` 的 `get` 方法会阻塞当前线程直到任务执行完毕。

上面的还有一种写法，如下：

```
package com.itsoku.chat34;

import com.google.common.util.concurrent.*;
import lombok.extern.slf4j.Slf4j;
import org.checkerframework.checker.nullness.qual.Nullable;

import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里p7学并发，微信公众号：javacode2018
 */
@Slf4j
```

```

public class Demo2 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService delegate = Executors.newFixedThreadPool(5);
        try {
            ListeningExecutorService executorService =
MoreExecutors.listeningDecorator(delegate);
            ListenableFuture<Integer> submit = executorService.submit(() -> {
                log.info("{}", System.currentTimeMillis());
                TimeUnit.SECONDS.sleep(4);
                //int i = 10 / 0;
                log.info("{}", System.currentTimeMillis());
                return 10;
            });
            Futures.addCallback(submit, new FutureCallback<Integer>() {
                @Override
                public void onSuccess(@Nullable Integer result) {
                    log.info("执行成功:{}", result);
                }

                @Override
                public void onFailure(Throwable t) {
                    try {
                        TimeUnit.MILLISECONDS.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    log.error("执行任务发生异常:" + t.getMessage(), t);
                }
            }, MoreExecutors.directExecutor());
            log.info("{}", submit.get());
        } finally {
            delegate.shutdown();
        }
    }
}

```

输出：

```

14:26:07.938 [pool-1-thread-1] INFO com.itsoku.chat34.Demo2 - 1567491967936
14:26:11.944 [pool-1-thread-1] INFO com.itsoku.chat34.Demo2 - 1567491971944
14:26:11.945 [main] INFO com.itsoku.chat34.Demo2 - 10
14:26:11.945 [pool-1-thread-1] INFO com.itsoku.chat34.Demo2 - 执行成功:10

```

上面通过调用 `Futures` 的静态方法 `addCallback` 在异步执行的任务中添加回调，回调的对象是一个 `FutureCallback`，此对象有2个方法，任务执行成功调用 `onSuccess`，执行失败调用 `onFailure`。失败的情况可以将代码中 `int i = 10 / 0;` 注释去掉，执行一下可以看看效果。

示例2：获取一批异步任务的执行结果

```

package com.itsoku.chat34;

import com.google.common.util.concurrent.Futures;

```

```

import com.google.common.util.concurrent.ListenableFuture;
import com.google.common.util.concurrent.ListeningExecutorService;
import com.google.common.util.concurrent.MoreExecutors;
import lombok.extern.slf4j.Slf4j;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.*;
import java.util.stream.Collectors;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
@Slf4j
public class Demo3 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        log.info("star");
        ListeningExecutorService delegate = Executors.newFixedThreadPool(5);
        try {
            ListeningExecutorService executorService =
            MoreExecutors.listeningDecorator(delegate);
            List<ListenableFuture<Integer>> futureList = new ArrayList<>();
            for (int i = 5; i >= 0; i--) {
                int j = i;
                futureList.add(executorService.submit(() -> {
                    TimeUnit.SECONDS.sleep(j);
                    return j;
                }));
            }
            //获取一批任务的执行结果
            List<Integer> resultList = Futures.allAsList(futureList).get();
            //输出
            resultList.forEach(item -> {
                log.info("{}", item);
            });
        } finally {
            delegate.shutdown();
        }
    }
}

```

输出：

```

14:26:35.970 [main] INFO com.itsoku.chat34.Demo3 - star
14:26:41.137 [main] INFO com.itsoku.chat34.Demo3 - 5
14:26:41.138 [main] INFO com.itsoku.chat34.Demo3 - 4
14:26:41.138 [main] INFO com.itsoku.chat34.Demo3 - 3
14:26:41.138 [main] INFO com.itsoku.chat34.Demo3 - 2
14:26:41.138 [main] INFO com.itsoku.chat34.Demo3 - 1
14:26:41.138 [main] INFO com.itsoku.chat34.Demo3 - 0

```

结果中按顺序输出了6个异步任务的结果，此处用到了Futures.allAsList方法，看一下此方法的声明：

```
public static <V> ListenableFuture<List<V>> allAsList(
    Iterable<? extends ListenableFuture<? extends V>> futures)
```

传递一批 `ListenableFuture`，返回一个 `ListenableFuture<List<V>>`，内部将一批结果转换为了一个 `ListenableFuture` 对象。

示例3：一批任务异步执行完毕之后回调

异步执行一批任务，最后技术其和

```
package com.itsoku.chat34;

import com.google.common.util.concurrent.*;
import lombok.extern.slf4j.Slf4j;
import org.checkerframework.checker.nullness.qual.Nullable;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

/**
 * 跟着阿里P7学并发，微信公众号：javacode2018
 */
@Slf4j
public class Demo4 {
    public static void main(String[] args) throws ExecutionException,
    InterruptedException {
        log.info("star");
        ExecutorService delegate = Executors.newFixedThreadPool(5);
        try {
            ListeningExecutorService executorService =
MoreExecutors.listeningDecorator(delegate);
            List<ListenableFuture<Integer>> futureList = new ArrayList<>();
            for (int i = 5; i >= 0; i--) {
                int j = i;
                futureList.add(executorService.submit(() -> {
                    TimeUnit.SECONDS.sleep(j);
                    return j;
                }));
            }
            ListenableFuture<List<Integer>> listListenableFuture =
Futures.allAsList(futureList);
            Futures.addCallback(listListenableFuture, new
FutureCallback<List<Integer>>() {
                @Override
                public void onSuccess(@Nullable List<Integer> result) {
                    log.info("result中所有结果之和：" +
result.stream().reduce(Integer::sum).get());
                }
            });
        }
    }
}
```

```
        public void onFailure(Throwable t) {
            log.error("执行任务发生异常：" + t.getMessage(), t);
        }
    }, MoreExecutors.directExecutor());
} finally {
    delegate.shutdown();
}
}
```

输出：

```
14:47:04.819 [main] INFO com.itsoku.chat34.Demo4 - star
14:47:09.933 [pool-1-thread-1] INFO com.itsoku.chat34.Demo4 - result中所有结果之和：15
```

代码中异步执行了一批任务，所有任务完成之后，回调了上面的 `onSuccess` 方法，内部对所有的结果进行 `sum` 操作。

总结

- 通过 guava 提供的一些工具类，方便异步执行任务并进行回调
- guava 内部还有很多好用的工具类，有兴趣的可以去研究一下

更多好文章

1. [spring高手系列（正在连载中）](#)
2. [Java高并发系列（共34篇）](#)
3. [MySQL高手系列（共27篇）](#)
4. [Maven高手系列（共10篇）](#)
5. [Mybatis系列（共12篇）](#)
6. [聊聊db和缓存一致性常见的实现方式](#)
7. [接口幂等性这么重要，它是什么？怎么实现？](#)
8. [泛型，有点难度，会让很多人懵逼，那是因为你没有看这篇文章！](#)



扫码发送：月薪3万，获取月薪3万java
学习线路图及全部视频！
并参加每月免费送书活动！