Concurrent概述

2018年8月18日 星期六　　上午 11:27

1. Concurrent概述

   Concurrent包是jdk5中开始提供的一套并发编程包,其中包含了大量和多线程开发相关的工具类,大大的简化了java的多线程开发,在高并发 分布式场景下应用广泛.

2. Concurrent包

   java.util.concurrent

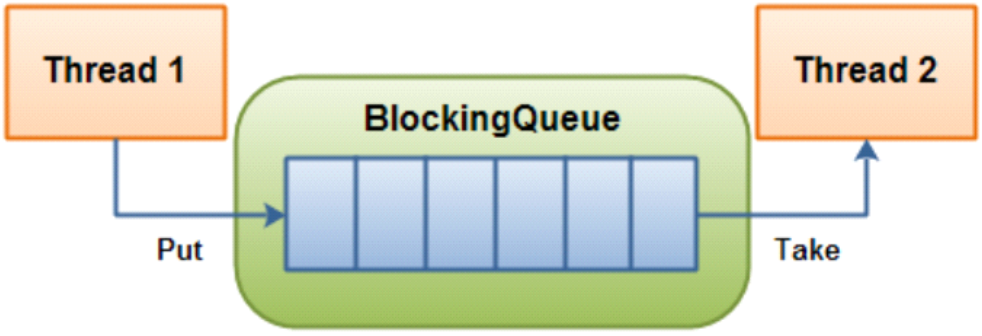
BlockingQueue-阻塞式队列

2018年8月18日 星期六　　上午 11:33

1. 阻塞式队列概述

本身是一种队列数据结构,和其他队列比起来,多了阻塞机制,从而可以在多个线程之间进行存取队列的操作,而不会有线程并发安全问题.所以称之为阻塞式队列.

可以简单的理解为,阻塞式队列是专门设计用来在多个线程间通过队列共享数据.



在阻塞式队列中,

如果队列满了,仍然有线程向其中写入数据,则这次写入操作会被阻塞住,直到有另外的线程从队列中消费了数据,队列有了空间,阻塞才会被放开,写入操作才可以执行.

同样,如果队列是空的,仍然有线程从队列中获取数据,则读取操作将会被阻塞住,直到有另外的线程向队列中写入了数据,队列不再为空,阻塞才会被放开,读取操作才可以执行

可以发现这个过程中,通过阻塞机制,协调了多个线程在一个队列上的读写操作,所以称之为阻塞式队列.

阻塞式队列可以非常方便的实现 多线程开发中 经典的 生产者-消费者模式

2. 继承结构

java.util.concurrent

接口 BlockingQueue<E>

|-

java.util.concurrent

类 ArrayBlockingQueue<E>

|-

java.util.concurrent

类 LinkedBlockingQueue<E>

ArrayBlockingQUeue 底层是数组 需要在创建之初就指定好大小

LinkedBlockingQUeue 底层是链表 可以在创建时指定大小 也可以不指定

3. 重要方法

	抛出异常	*特殊值*	*阻塞*	*超时*
插入	add(e)	offer(e)	put(e)	offer(e, time, unit)
移除	remove()	poll()	take()	poll(time, unit)
检查	element()	peek()	*不可用*	*不可用*

在BlockingQueue中 插入 移除 和 检查数据 有四组方法,这四组方法在处理 队列满的时候插入 队列空的时候获取 时 会有不同的处理机制

包括

抛出异常

返回特殊值

产生阻塞

产生阻塞 但是阻塞具有超时时间 一旦超过指定时间 阻塞自动放开

在使用BlockingQueue过程中 可以根据需要选择对应的方法

4. 案例

```
1  package cn.tedu.concurrent;
2
3  import java.util.concurrent.ArrayBlockingQueue;
4  import java.util.concurrent.BlockingQueue;
5
6  public class BlockingQueueDemo01 {
7          public static void main(String[] args) throws Exception {
8                  //1.创建阻塞式队列
9                  BlockingQueue<String> queue = new ArrayBlockingQueue<>(5);
```

```
10                  //2.存入数据
11                  new Thread(new Producer(queue)).start();
12                  new Thread(new Consumer(queue)).start();
13          }
14  }
15
16  class Consumer implements Runnable{
17          private BlockingQueue<String> queue = null;
18
19          public Consumer(BlockingQueue<String> queue) {
20                  this.queue = queue;
21          }
22
23          @Override
24          public void run() {
25
26                  try {
27                          while(true){
28                                  Thread.sleep(2000);
29                                  String s = queue.take();
30                                  System.out.println("消费者消费了"+s);
31                          }
32                  } catch (Exception e) {
33                          e.printStackTrace();
34                  }
35
36          }
37  }
38
39  class Producer implements Runnable{
40          private BlockingQueue<String> queue = null;
41
42          public Producer(BlockingQueue<String> queue) {
43                  this.queue = queue;
44          }
45          @Override
46          public void run() {
47                  try {
48                          int i = 0;
49                          while(true){
50                                  Thread.sleep(5000);
51                                  int n = ++i;
52                                  queue.put("a"+n);
53                                  System.out.println("生产者生产了a"+n);
54                          }
55                  } catch (Exception e) {
56                          e.printStackTrace();
57                  }
58          }
59  }
```

ConcurrentMap-并发Map

2018年8月18日 星期六　　下午 2:12

1. ConcurrentMap概述

ConcurrentMap是一个线程安全的Map,可以防止多线程并发安全问题.

HashTable也是线程安全的,但是ConcurrentMap性能要比HashTable好的多,所以推荐使用ConcurrentMap.

2. ConcurrentMap性能非常好的原因

a. 锁更加精细

HashTable加锁是锁在整个HashTable上,一个线程操作时其他线程无法操作,性能比较低.

ConcurrentMap加锁是 将锁加载数据分段(桶)上 只锁正在操作的部分数据 效率高.

b. 引入了读写锁机制

在多线程并发操作的过程中,多个并发的读其实不需要隔离,但只要有任意一个写操作,就必须隔离.

HashTable没有考虑一点,无论什么类型的操作,直接在整个HashTable上加锁.

ConcurrentMap则区分了读写操作,读的时候加读锁,写的时候加写锁,读锁和读锁可以共存,写锁和任意锁都不能共存,从而实现了在多个读的过程中不会隔离 提高了效率.

3. ConcurrentMap的继承结构

java.util.concurrent

接口 ConcurrentMap<K,V>

|-

java.util.concurrent

类 ConcurrentHashMap<K,V>

4. 案例

```
1  ConcurrentMap<String,String>map = new ConcurrentHashMap<>();
2  map.put("name", "zs");
3  map.put("addr","bj");
4  System.out.println(map.get("name"));
5  System.out.println(map.get("addr"));
```

CountDownLatch-闭锁

2018年8月18日 星期六　　下午 2:31

1. 闭锁概述

是Concurrent包提供的一种新的并发构造,可以协调线程的执行过程,实现协调某个线程阻塞直到其他若干线程执行达到一定条件才放开阻塞继续执行的效果.

2. 重要API

构造方法,需要在构造的过程中直接传入一个数字作为闭锁的计数器的初始值,构造闭锁

构造方法摘要
CountDownLatch(int count) 构造一个用给定计数初始化的 CountDownLatch。

在闭锁上调用此方法,可以阻塞当前线程,阻塞到CountDownLatch中的计数器count值变为0,自动放开阻塞

void	**await**() 使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。

调用此方法可以将闭锁中的计数器数值-1,如果减到零,await的阻塞会自动放开

void	**countDown**() 递减锁存器的计数，如果计数到达零，则释放所有等待的线程。

案例

做饭线程 需要等到 买锅 买米 买菜的线程 执行完成后 才能执行

```java
package cn.tedu.concurrent;


import java.util.concurrent.CountDownLatch;


public class CountDownLatchDemo01 {
	public static void main(String[] args) {
		//1.创建闭锁 计数器初始值设置为3
		CountDownLatch cdl = new CountDownLatch(3);
		new Thread(new MaiGuo(cdl)).start();
		new Thread(new MaiMi(cdl)).start();
		new Thread(new MaiCai(cdl)).start();
		new Thread(new ZuoFan(cdl)).start();
	}
}


class ZuoFan implements Runnable{
	private CountDownLatch cdl = null;

	public ZuoFan(CountDownLatch cdl) {
		this.cdl = cdl;
	}
	@Override
	public void run() {
		try {
			//--调用await等待 达到执行条件
			cdl.await();
			System.out.println("开始做饭...");
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}


class MaiGuo implements Runnable{
	private CountDownLatch cdl = null;
	public MaiGuo(CountDownLatch cdl) {
		this.cdl = cdl;
	}
	@Override
	public void run() {
		try {
			Thread.sleep(2000);
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
		System.out.println("锅买回来了...");

		//--在闭锁上-1
		cdl.countDown();
	}
}


class MaiCai implements Runnable{
	private CountDownLatch cdl = null;
	public MaiCai(CountDownLatch cdl) {
		this.cdl = cdl;
	}
	@Override
	public void run() {
		try {
			Thread.sleep(5000);
		} catch (InterruptedException e) {
			e.printStackTrace();
```

```
66                }
67                System.out.println("菜买回来了...");
68
69                //--在闭锁上-1
70                cdl.countDown();
71          }
72    }
73
74
75    class MaiMi implements Runnable{
76          private CountDownLatch cdl = null;
77          public MaiMi(CountDownLatch cdl) {
78                this.cdl = cdl;
79          }
80          @Override
81          public void run() {
82                try {
83                      Thread.sleep(3000);
84                } catch (InterruptedException e) {
85                      e.printStackTrace();
86                }
87                System.out.println("米买回来了...");
                  //--在闭锁上-1
                  cdl.countDown();
          }
    }
```

CyclicBarrier-栅栏

2018年8月18日 星期六　　下午 3:01

1. CyclicBarrier栅栏概述

Concurrent包中提供的一种并发构造,可以实现多个并发的线程在执行过程中,在某一个节点进行阻塞等待,直到所有的线程都到达了指定位置后,一起放开阻塞继续运行的效果.

2. 重要API

构造方法,接收一个初始值,指定了栅栏要等待的线程的数量

构造方法摘要
CyclicBarrier(int parties) 创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。

当线程到达了栅栏时,可以调用此方法,进入阻塞等待的状态,线程被挂起,直到在栅栏上等待的线程的数量达到了栅栏上设定的要等待的线程的数量,所有线程的阻塞同时被放开,一起继续执行

int	**await**() 在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。

3. 案例

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.BrokenBarrierException;
5  import java.util.concurrent.CyclicBarrier;
6
7
8  public class CyclicBarrierDemo01 {
9        public static void main(String[] args) {
10             //构造栅栏,指定要等待的线程的数量
11             CyclicBarrier cb = new CyclicBarrier(5);
12
13             new Thread(new Horse(cb)).start();
14             new Thread(new Horse(cb)).start();
15             new Thread(new Horse(cb)).start();
16             new Thread(new Horse(cb)).start();
17             try {
18                   Thread.sleep(3000);
19             } catch (InterruptedException e) {
20                   e.printStackTrace();
               }
```

```
21                }
22                new Thread(new Horse(cb)).start();
23          }
24    }
25
26    class Horse implements Runnable{
27          private CyclicBarrier cb = null;
28          public Horse(CyclicBarrier cb) {
29                this.cb  = cb;
30          }
31          @Override
32          public void run() {
33                System.out.println("马到达了栅栏,开始等待...");
34                try {
35                      cb.await();
36                } catch (Exception e) {
37                      e.printStackTrace();
38                }
39                System.out.println("马跑了出去...");
          }
      }
```

Exchanger-交换机

2018年8月18日 星期六　　下午 3:15

1. Exchanger交换机概述

可以实现两个线程交换对象的效果 先到达的线程会产生阻塞 等待后续到来的线程,直到两个线程都到达交换机后 互换对象 各自继续执行

2. 重要API

构造方法

构造方法摘要
Exchanger() 创建一个新的 Exchanger。

在交换机中交换对象的方法 先到的线程调用此方法时 会进入阻塞状态 直到另一个线程也调用这个方法 互换对象 阻塞放开 各自继续执行

方法摘要	
V	**exchange**(V x) 等待另一个线程到达此交换点（除非当前线程被中断），然后将给定的对象传送给该线程，并接收该线程的对象。

3. 案例

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.Exchanger;
5
6
7  public class ExchangerDemo01 {
8        public static void main(String[] args) {
9              //1.创建交换机
10             Exchanger<String> exchanger = new Exchanger<>();
11
12             //2.利用交换机互换对象
13             new Thread(new A(exchanger)).start();
14             new Thread(new B(exchanger)).start();
15       }
16 }
17
```

```
class B implements Runnable{
	private Exchanger<String> exchanger = null;
	public B(Exchanger<String> exchanger) {
		this.exchanger = exchanger;
	}
	@Override
	public void run() {
		try {
			Thread.sleep(5000);
			System.out.println("B到达了交换机...");
			String msg = exchanger.exchange("化而为鸟 其名为鹏 鹏之大 需要两个烧烤架");
			System.out.println("B收到了A发送的消息:["+msg+"]");
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
class A implements Runnable{
	private Exchanger<String> exchanger = null;
	public A(Exchanger<String> exchanger) {
		this.exchanger = exchanger;
	}
	@Override
	public void run() {
		try {
			Thread.sleep(2000);
			System.out.println("A到达了交换机...");
			String msg = exchanger.exchange("北冥有鱼 其名为鲲 鲲之大一锅炖不下");
			System.out.println("A收到了B发送的消息:["+msg+"]");
		} catch (Exception e) {
			e.printStackTrace();
		}
	}
}
```

Semaphore-信号量

2018年8月18日 星期六　　下午 3:26

1. Semaphore信号量概述

Concurrent包中提供的一个并发构造,可以在创建信号量时指定信号量的初始数量,后续可以调用acquire()来获取信号量 通过release()释放信号量,如果某一个时刻,信号量被取完,再调用acquire()方法时,该方法将会产生阻塞,直到有其他线程release()信号量回来.

2. 信号量的主要用途

a. 保护一个重要(代码)部分防止一次超过 N 个线程进入。
b. 在两个线程之间发送信号。

3. 重要API

构造方法,在构造的时候需要指定信号量的初始值,可选的还可以传入一个布尔类型的值,来指定当前信号量是否采用公平策略,默认不公平的

构造方法摘要
Semaphore(int permits) 创建具有给定的许可数和非公平的公平设置的 Semaphore。
Semaphore(int permits, boolean fair) 创建具有给定的许可数和给定的公平设置的 Semaphore。

获得信号量,在信号量上数量-1,如果已经没有剩余的信号量,则此方法将会阻塞,直到有其他线程释放信号量

void	**acquire**() 从此信号量获取一个许可，在提供一个许可前一直将线程阻塞，否则线程被中断。

释放信号量,在信号量上数量+1

void	release() 释放一个许可，将其返回给信号量。

4. 案例

a. 保护人民大会堂

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.Semaphore;
5
6
7  public class SemaphoreDemo01 {
8          public static void main(String[] args) {
9                  //1.创建一个信号量
10                 Semaphore semaphore = new Semaphore(5);
11
12                 new Thread(new Cust(semaphore)).start();
13                 new Thread(new Cust(semaphore)).start();
14                 new Thread(new Cust(semaphore)).start();
15                 new Thread(new Cust(semaphore)).start();
16                 new Thread(new Cust(semaphore)).start();
17
18                 new Thread(new Cust(semaphore)).start();
19         }
20 }
21
22 class Cust implements Runnable{
23         private Semaphore semaphore = null;
24         public Cust(Semaphore semaphore) {
25                 this.semaphore  = semaphore;
26         }
27         @Override
28         public void run() {
29                 try {
30                         semaphore.acquire();
31                         System.out.println("得到信号量,开始参观人民大会堂..");
32                         Thread.sleep(5000);
33                         System.out.println("参观人民大会堂结束,释放信号量..");
34                         semaphore.release();
35                 } catch (InterruptedException e) {
36                         e.printStackTrace();
37                 }
           }
   }
```

b. 信号量实现线程间的通信

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.Semaphore;
5
6
7  public class SemaphoreDemo02 {
8        public static void main(String[] args) throws Exception {
9              //1.创建信号量
10             Semaphore semaphore = new Semaphore(1);
11             new Thread(new Master(semaphore)).start();
               Thread.sleep(1000);
```

```
12                  Thread.sleep(1000);
13                  new Thread(new Slave(semaphore)).start();
14            }
15      }
16
17      class Master implements Runnable{
18            private Semaphore semaphore = null;
19            public Master(Semaphore semaphore) {
20                  this.semaphore  = semaphore;
21            }
22            @Override
23            public void run() {
24                  try {
25                        semaphore.acquire();
26                        System.out.println("Master得到信号量..");
27                        Thread.sleep(3000);
28                        System.out.println("Master释放信号量..");
29                        semaphore.release();
30                  } catch (Exception e) {
31                        e.printStackTrace();
32                  }
33
34            }
35      }
36      class Slave implements Runnable{
37            private Semaphore semaphore = null;
38            public Slave(Semaphore semaphore) {
39                  this.semaphore  = semaphore;
40            }
41            @Override
42            public void run() {
43                  try {
44                        semaphore.acquire();
45                        System.out.println("Slave得到信号量..");
46                        Thread.sleep(3000);
47                        System.out.println("Slave释放信号量..");
48                        semaphore.release();
49                  } catch (Exception e) {
50                        e.printStackTrace();
51                  }
52
            }
      }
```

ExecutorService-执行器服务

2018年8月18日 星期六　　下午 3:59

1. ExecutorService概述

   ExecutorService是Concurrent包下提供的一个接口,主要用来实现线程池
   所谓的池就是用来重用对象的一个集合,可以减少对象的创建和销毁,提高效率.
   而线程本身就是一个重量级的对象,线程的创建和销毁都是非常耗费资源和时间,所以如果需要频繁使用大量线程,不建议每次都创建线程销毁线程,而是应该利用线程池的机制,实现线程对象的共享,提升程序的效率

2. ExecutorService用法 - 通过ThreadPoolExecutor实现类创建线程池

   a. ThreadPoolExecutor继承结构

      java.util.concurrent
      接口 ExecutorService
          |-
              java.util.concurrent
              类 ThreadPoolExecutor

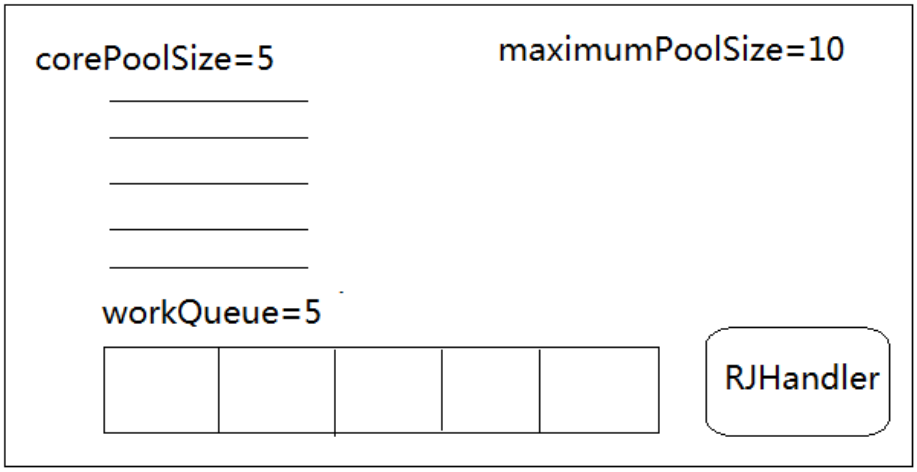
   b. ThreadPoolExecutor的重要API

      构造方法

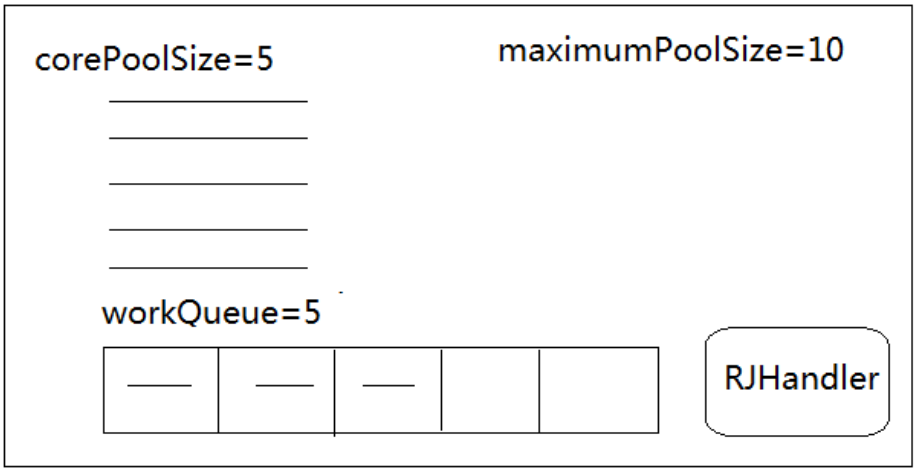
      ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
      long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue,
      RejectedExecutionHandler handler)
          用给定的初始参数和默认的线程工厂创建新的 ThreadPoolExecutor。

      **线程池的工作方式

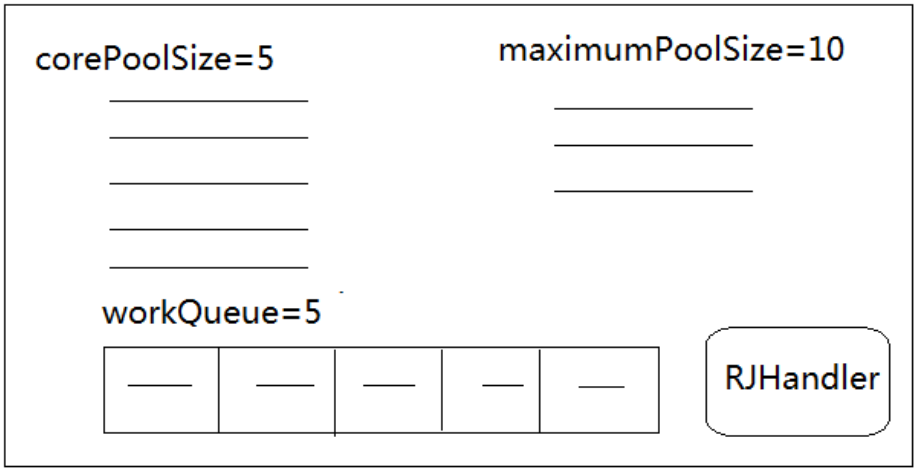
      1) 在线程池刚创建出来时,线程池中没有任何线程,当有任务提交过来时,如果线程池中管理的线程的数量小于corePoolSize,则无论是否有闲置的线程都会创建新的线程来使用.
      而当线程池中管理的线程的数量达到了corePoolSize,再有新任务过来时,会复用闲置的线程.



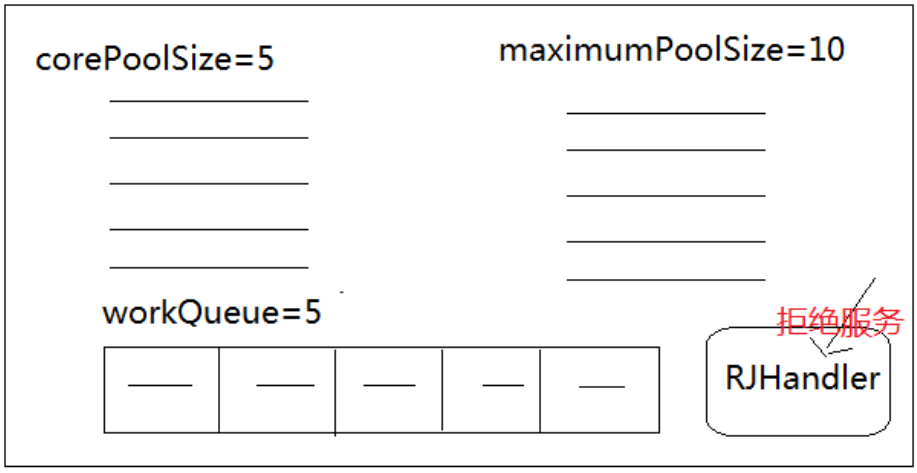
2) 当所有的核心池大小中的线程都在忙碌,则再有任务提交,会存入workQueue中,进行排队,当核心池大小中的线程闲置后,会自动从workQueue获取任务执行



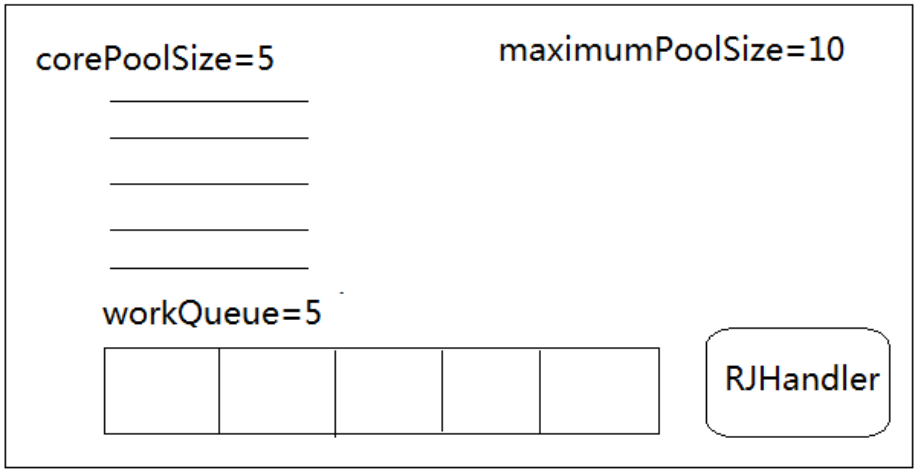
3) 而当所有的核心池大小中的线程都在忙碌,workQueue也满了,则会再去创建新的临时线程来处理提交的任务,但是,无论如何,总的线程数量,不允许超过maximumPoolSize



4) 而当所有的核心池大小中的线程都在忙碌,workQueue也满了,也创建了达到了maximumPoolSize的临时线程,再有任务提交,此时会交予RJHandler来拒绝该任务



5) 当任务高峰过去,workQueue中的任务也都执行完成,线程也依次闲置了下来,则在此时,会将闲置时间超过keepAliveTime(单位为unit)时长的线程关闭掉,但是关闭时会至少保证线程池中管理的线程的数量 不少于corePoolSize个



3. ExecutorService用法 - 通过Executors工具类的静态方法创建线程池

a. 关键API

创建一个擅长处理 大量短任务的线程池

corePoolSize=0

maximumPoolSize = Integer.MaxValue

keepAliveTime = 60

TimeUnit = Seconds

static ExecutorService	**newCachedThreadPool**() 创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

可以实现使用单一线程处理任务,多个任务在无界的阻塞式队列中排队等待处理

corePoolSize=1

maximumPoolSize = 1

workQueue = new LinkedBlockingQueue<Runnable>())

static ExecutorService	**newSingleThreadExecutor**() 创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

可以实现使用指定数量的线程处理任务,多个任务在无界的阻塞式队列中排队等待处理

corePoolSize=nTHreads

maximumPoolSize = nTHreads

workQueue = new LinkedBlockingQueue<Runnable>())

static ExecutorService	**newFixedThreadPool**(int nThreads) 创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。

b. 案例

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.ArrayBlockingQueue;
5  import java.util.concurrent.ExecutorService;
6  import java.util.concurrent.Executors;
7  import java.util.concurrent.RejectedExecutionHandler;
8  import java.util.concurrent.ThreadPoolExecutor;
9  import java.util.concurrent.TimeUnit;
10
11 public class ExecutorServiceDemo01 {
12         public static void main(String[] args) {
13                 //1.手动创建线程池
14                 ExecutorService s1 = new ThreadPoolExecutor(5, 10, 5, TimeUnit.SECONDS, new
15 ArrayBlockingQueue<>(5)
16                                 , new RejectedExecutionHandler() {
17                         @Override
18                         public void rejectedExecution(Runnable r, ThreadPoolExecutor executor)
19 {
20                                 System.err.println("不好意思,线程池实在是太忙了..您这个任务只能给你拒绝
21 了..["+r+"]");
22                         }
23                 });
24
25                 //2.利用工具类的静态方法快速创建线程池
26                 ExecutorService s2 = Executors.newCachedThreadPool();
27                 ExecutorService s3 = Executors.newSingleThreadExecutor();
                   ExecutorService s4 = Executors.newFixedThreadPool(5);

         }
   }
```

4. 向线程池中提交任务

a. execute(Runnable)

最普通的提交任务的方法,直接传入一个Runnable接口的实现类对象,即可要求线程池取执行这个任务,这种方式提交的任务无法监控线程的执行 也无法在线程内向调用者返回返回值.

void	**execute**(Runnable command) 在未来某个时间执行给定的命令。

案例:

```
1  s.execute(new Runnable() {
2          @Override
3          public void run() {
4                  for(int i=0;i<10;i++){
```

```
5                           System.out.println("run.."+i);
6                   }
7           }
8   });
```

b. submit(Runnable)

此方法也可以通过传入一个Runnable接口实现类对象的方式来向线程池提交任务,不同之处在于此方法带有一个Future类型的返回值,可以通过Future对象的get()方法来检测线程是否执行结束,如果线程未执行结束get()方法将会阻塞.

Future<?>	**submit**(Runnable task) 提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

案例:

```
1   Future<?> future = s.submit(new Runnable() {
2           @Override
3           public void run() {
4                   try {
5                           for(int i=0;i<10;i++){
6                                   System.out.println("run.."+i);
7                           }
8                           Thread.sleep(5000);
9                           System.out.println("Runnable结束了..");
10                  } catch (Exception e) {
11                          e.printStackTrace();
12                  }
13          }
14  });
15  future.get();
16  System.out.println("main线程走下来了...");
```

c. submit(Callable)

和上面submit(Runnable)方法非常类似,只不过这个方法传入的是Callable接口的实现类,Callable接口功能和Runnable接口基本一致,唯一的不同在于,内部的方法叫call,且可以返回返回值,这个返回值可以通过Future对象通过get()方法得到

<T> Future<T>	**submit**(Callable<T> task) 提交一个返回值的任务用于执行，返回一个表示任务的未决结果的 Future

案例:

```
1   Future<String> future = s.submit(new Callable<String>() {
2           @Override
3           public String call() throws Exception {
4                   System.out.println("子线程开始执行了..");
5                   Thread.sleep(2000);
6                   System.out.println("子线程执行结束了..");
7                   return "aaabbbccc";
8           }
9   });
10  String str = future.get();
11  System.out.println(str);
```

d. invokeAny(...)

可以接收若干Callable组成的集合,此方法将会自动从中选择任意一个执行

<T> T	**invokeAny**(Collection<? extends Callable<T>> tasks)

	执行给定的任务，如果某个任务已成功完成（也就是未抛出异常），则返回其结果。

案例:

```
List<Callable> list = new ArrayList<>();
list.add(new Callable<String>() {
        @Override
        public String call() throws Exception {
                Thread.sleep(2000);
                return "aaa";
        }
});
list.add(new Callable<String>() {
        @Override
        public String call() throws Exception {
                Thread.sleep(2000);
                return "bbb";
        }
});
list.add(new Callable<String>() {
        @Override
        public String call() throws Exception {
                Thread.sleep(2000);
                return "ccc";
        }
});
String str = s.invokeAny((Collection<? extends Callable<String>>) list);
System.out.println(str);
```

e. invokeAll(...)

可以接收若干Callable组成的集合,此方法将会执行所有的Callable将结果组成集合返回

<T> List<Future<T>>	**invokeAll**(Collection<? extends Callable<T>> tasks) 执行给定的任务，当所有任务完成时，返回保持任务状态和结果的 Future 列表。

案例:

```
List<Callable> list = new ArrayList<>();
list.add(new Callable<String>() {
      @Override
      public String call() throws Exception {
            Thread.sleep(2000);
            return "aaa";
      }
});
list.add(new Callable<String>() {
      @Override
      public String call() throws Exception {
            Thread.sleep(2000);
            return "bbb";
      }
});
list.add(new Callable<String>() {
      @Override
      public String call() throws Exception {
            Thread.sleep(2000);
            return "ccc";
      }
});
List<Future<String>> rs = s.invokeAll((Collection<? extends Callable<String>>) list);
for(Future<String> f : rs){
      System.out.println(f.get());
}
```

5. 关闭线程池

线程池中维护了大量线程,很耗费资源,所以当使用线程池结束时,应该手动关闭线程池,释放资源

关闭线程池的方法,即使调用也不会立即关闭所有线程,而是不再接收新的任务,之前已经提交但尚未完成执行的线程仍然会继续执行,直到所有的任务都执行完,线程池关闭所有线程,退出.

void	**shutdown**() 启动一次顺序关闭，执行以前提交的任务，但不接受新任务。

关闭线程池的方法,调用此方法时,会立即关闭所有线程,退出线程池,这种方式虽然可以立即推出线程池,但是正在执行的线程有可能被意外的中断,造成意想不到的问题

List<Runnable>	**shutdownNow**() 试图停止所有正在执行的活动任务，暂停处理正在等待的任务，并返回等待执行的任务列表。

Lock-锁

2018年8月20日 星期一　　上午 9:04

1. Lock锁概述

java.util.concurrent.locks.Lock 是一个类似于 synchronized 块的线程同步机制.但是Lock比 synchronized 块更加灵活、精细.

继承结构

java.util.concurrent.locks

接口 Lock

|-

java.util.concurrent.locks

类 ReentrantLock

重要方法

构造方法创建一把锁

构造方法摘要
ReentrantLock() 创建一个 ReentrantLock 的实例。
ReentrantLock(boolean fair) 创建一个具有给定公平策略的 ReentrantLock。

锁住锁

void	**lock**() 获取锁。

打开锁

void	**unlock**() 试图释放此锁。

2. 案例:

```
1  package cn.tedu.concurrent;
2
```

```
3
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6
7  /**
8          有共享资源
9          多个线程操作
10         涉及到修改操作
11
12  */
13
14 public class LockDemo2 {
15         public static String name = "夏洛";
16         public static String gender = "男";
17
18         //--创建锁对象
19
20         public static void main(String[] args) {
21                 Lock lock = new ReentrantLock(true);
22                 new Thread(new ChangeThread2(lock)).start();
23                 new Thread(new PrintThread2(lock)).start();
24         }
25
26 }
27
28 class ChangeThread2 implements Runnable{
29         private Lock lock = null;;
30         public ChangeThread2(Lock lock) {
31                 this.lock = lock;
32         }
33
34
35         @Override
36         public void run() {
37                 while(true){
38                         lock.lock();
39                         if("夏洛".equals(LockDemo2.name)){
40                                 LockDemo2.name = "马冬梅";
41                                 LockDemo2.gender = "女";
42                         }else{
43                                 LockDemo2.name = "夏洛";
44                                 LockDemo2.gender = "男";
45                         }
46                         lock.unlock();
47                 }
48         }
49 }
50
51
52 class PrintThread2 implements Runnable{
53         private Lock lock = null;
54         public PrintThread2(Lock lock) {
55                 this.lock = lock;
56         }
57
58         @Override
59         public void run() {
60                 while(true){
61                         lock.lock();
62                         System.out.println("姓名:"+LockDemo2.name);
63                         System.out.println("性别:"+LockDemo2.gender);
                        lock.unlock();
                }
        }
}
```

3. lock和syncronized对别

a. lock可以配置公平策略,实现线程按照先后顺序获取锁
b. 提供了trylock方法 可以试图获取锁 获取到 或 获取不到时 返回不同的返回值 让程序可以灵活处理
c. lock()和unlock()可以在不同的方法中执行,可以实现同一个线程在上一个方法中lock()在后续的其他方法中unlock(),比syncronized灵活的多

4. 读写锁

对于多线程并发安全问题,其实只在设计到并发写的时候才会发生,多个并发的读并不会有线程安全问题,所以在Concurrent包中提供了读写锁的机制,可以实现,分读锁和写锁来进行并发控制.多个读锁可以共存,而写锁和任意锁都不可共存,从而实现多个并发读并行执行提升效率,而任意时刻写都进行隔离,保证安全.这是一种非常高效而精细的锁机制.

继承结构

java.util.concurrent.locks
接口 ReadWriteLock
|-
java.util.concurrent.locks
类 ReentrantReadWriteLock

重要API

构造方法

构造方法摘要
ReentrantReadWriteLock() 使用默认（非公平）的排序属性创建一个新的 ReentrantReadWriteLock。
ReentrantReadWriteLock(boolean fair) 使用给定的公平策略创建一个新的 ReentrantReadWriteLock。

得到读写锁内部的读锁

ReentrantReadWriteLock.ReadLock	**readLock**()

	返回用于读取操作的锁。

得到读写所内部的写锁

ReentrantReadWriteLock.WriteLock	**writeLock**() 返回用于写入操作的锁。

5. 读写锁案例:

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.locks.ReadWriteLock;
5  import java.util.concurrent.locks.ReentrantReadWriteLock;
6
7  public class ReadWriteLockDemo01 {
8        public static String name = "夏洛";
9        public static String gender = "男";
10
11       public static void main(String[] args) {
12             ReadWriteLock rwLock = new ReentrantReadWriteLock();
13
14             new Thread(new Thread_A(rwLock)).start();
15             new Thread(new Thread_B(rwLock)).start();
16             new Thread(new Thread_C(rwLock)).start();
17       }
18 }
19
20
21 class Thread_C implements Runnable{
22       private ReadWriteLock rwLock = null;
23       public Thread_C(ReadWriteLock rwLock) {
24             this.rwLock = rwLock;
25       }
26       @Override
27       public void run() {
28             rwLock.writeLock().lock();
29             if("夏洛".equals(ReadWriteLockDemo01.name)){
30                   LockDemo01.name = "马冬梅";
31                   LockDemo01.gender = "女";
32             }else{
33                   LockDemo01.name = "夏洛";
34                   LockDemo01.gender = "男";
35             }
36             rwLock.writeLock().unlock();
37       }
38 }
39
40
41 class Thread_B implements Runnable{
42       private ReadWriteLock rwLock = null;
43       public Thread_B(ReadWriteLock rwLock) {
44             this.rwLock = rwLock;
45       }
46       @Override
47       public void run() {
48             while(true){
49                   rwLock.readLock().lock();
50                   System.out.println("B:"+ReadWriteLockDemo01.name);
51                   System.out.println("B:"+ReadWriteLockDemo01.gender);
52                   rwLock.readLock().unlock();
53             }
```

```
54          }
55    }
56    class Thread_A implements Runnable{
57          private ReadWriteLock rwLock = null;
58          public Thread_A(ReadWriteLock rwLock) {
59                this.rwLock = rwLock;
60          }
61          @Override
62          public void run() {
63                while(true){
64                      rwLock.readLock().lock();
65                      System.out.println("A:"+ReadWriteLockDemo01.name);
66                      System.out.println("A:"+ReadWriteLockDemo01.gender);
67                      rwLock.readLock().unlock();
                  }
            }
      }
```

Atomic-原子性操作

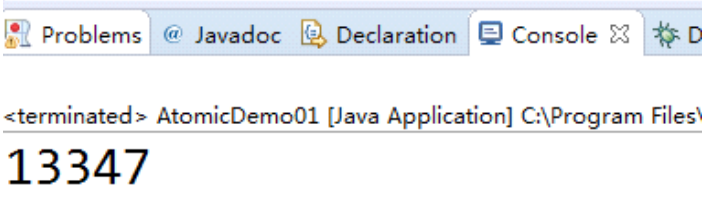
2018年8月20日 星期一　　上午 9:59

1. 原型操作问题

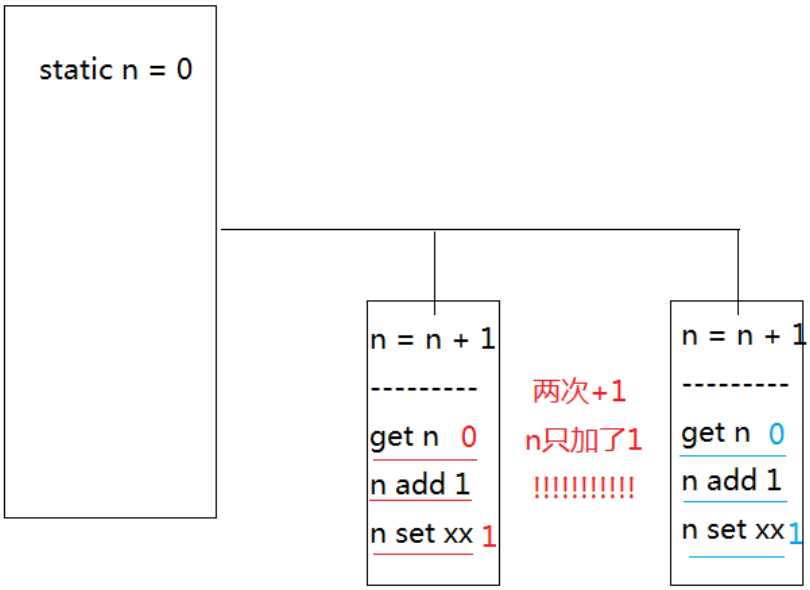
案例:

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.CountDownLatch;
5
6
7  public class AtomicDemo01 {
8          public static int n = 0;
9
10         public static void main(String[] args) throws Exception {
11                 CountDownLatch cdl = new CountDownLatch(2);
12                 new Thread(new T_A(cdl)).start();
13                 new Thread(new T_A(cdl)).start();
14                 cdl.await();
15
16                 System.out.println(n);
17         }
18 }
19
20 class T_A implements Runnable{
21         private CountDownLatch cdl = null;
22         public T_A(CountDownLatch cdl) {
23                 this.cdl = cdl;
24         }
25         @Override
26         public void run() {
27                 for(int i=0;i<10000;i++){
28                         AtomicDemo01.n = AtomicDemo01.n + 1;
29                 }
30                 cdl.countDown();
           }
   }
```

实验结果:



分析原因:



结论:

在java中最基本的运算都是非原子性的,在底层存在多个步骤,所以在多线程并发操作的过程中有可能有多线程并发安全问题

解决方案1:

使用同步代码块进行同步,可以解决问题,但是如果在多线程并发场景下,最基本的运算都要同步的话,代码会被大量的同步代码块包裹,代码混乱,效率低下

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.CountDownLatch;
5
6
7  public class AtomicDemo01 {
8          public static int n = 0;
9
10         public static void main(String[] args) throws Exception {
11                 CountDownLatch cdl = new CountDownLatch(2);
12                 new Thread(new T_A(cdl)).start();
13                 new Thread(new T_A(cdl)).start();
14                 cdl.await();
15
16                 System.out.println(n);
17         }
18 }
19
20 class T_A implements Runnable{
21         private CountDownLatch cdl = null;
22         public T_A(CountDownLatch cdl) {
23                 this.cdl = cdl;
24         }
25         @Override
26         public void run() {
27                 for(int i=0;i<10000;i++){
28                         synchronized (T_A.class) {
29                                 AtomicDemo01.n = AtomicDemo01.n + 1;
30                         }
31                 }
```

```
32                      cdl.countDown();
                }
        }
```

解决方案2:使用Concurrent包中提供的Atomic原子型操作

2. Atomic原子型操作

为了解决以上问题,在Concurrent包中,提供了大量可以实现原子型操作的包装类型

java.util.concurrent.atomic

类 AtomicBoolean

java.util.concurrent.atomic

类 AtomicInteger

java.util.concurrent.atomic

类 AtomicLong

java.util.concurrent.atomic

类 AtomicReference<V>

以AtomicInteger为例:

构造方法

构造方法摘要
AtomicInteger() 创建具有初始值 0 的新 AtomicInteger。
AtomicInteger(int initialValue) 创建具有给定初始值的新 AtomicInteger。

在原有值上加上给定值后返回该值

int	**addAndGet**(int delta) 以原子方式将给定值与当前值相加。

返回原有值后在原值上加给定值

int	**getAndAdd**(int delta) 以原子方式将给定值与当前值相加。

获取原值后在原值上加一

int	**getAndIncrement**() 以原子方式将当前值加 1。

获取原值后在原值上减一

int	**getAndDecrement**() 以原子方式将当前值减 1。

将值设置为给定值

void	**set**(int newValue) 设置为给定值。

案例:

```
1  package cn.tedu.concurrent;
2
3
4  import java.util.concurrent.CountDownLatch;
5  import java.util.concurrent.atomic.AtomicInteger;
6
7
8  public class AtomicDemo2 {
9        public static AtomicInteger n = new AtomicInteger();
10
11       public static void main(String[] args) throws Exception {
12             CountDownLatch cdl = new CountDownLatch(2);
13             new Thread(new T_Ax(cdl)).start();
14             new Thread(new T_Ax(cdl)).start();
15             cdl.await();
16
17             System.out.println(n.get());
18       }
19 }
20
21 class T_Ax implements Runnable{
22       private CountDownLatch cdl = null;
23       public T_Ax(CountDownLatch cdl) {
24             this.cdl = cdl;
25       }
26       @Override
27       public void run() {
28             for(int i=0;i<10000;i++){
29                   AtomicDemo2.n.incrementAndGet();
30             }
31             cdl.countDown();
       }
   }
```

总结&作业

2018年8月18日 星期六　　下午 5:15

1. java的Concurrent包,其中包含了大量多线程开发相关的工具 大大提供多线程程序开发的效率
2. 掌握 BlockingQueue ConcurrentMap CountDownLatch ExecutorServcie Lock
3. 了解 CyclicBarrier Exchanger Semaphore Atmoic