**第1章 Java多线程技能(Thread API)**

1.1 进程和多线程的概念及线程的优点

* **进程:**



一个正在操作系统上执行的exe程序。

* **线程**

进程中独立运行的子任务。

**单任务:**排队执行，即同步。CPU利用率大幅降低。

**1.2 使用多线程**

JVM创建调用main()的线程

1.2.1 继承Thread类

* Public class Thread implements Runnable {}
* CPU以不确定的方式(随机时间)调用线程中run方法。

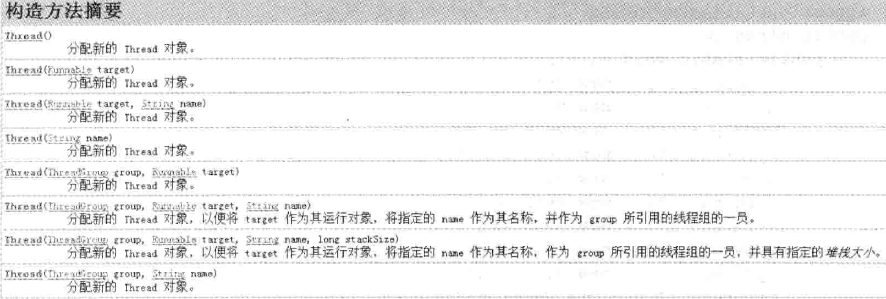
如多次调用start(),会出现Exception in thread “main” java.lang. IllegalThreadStateException

* start()通知”线程规划器”此线程已准备就绪，等待调用线程对象的run()方法，就是让系统安排时间来调用Thread中的run()；线程启动顺序与start()执行顺序无关。
* 如用thread.run()，是同步，此线程对象不交给”线程规划器”处理，而是由main线程调用run()

1.2.2 实现Runnable接口

如欲创建的线程类已有父类，就不能再继承Thread类。

* Thread构造函数：



* 例子:
* Thread(Runnable target, String name)
* Thread(Runnable target)：因Thread implements Runnable，因此也可传入Thread类对象

|  |
| --- |
| package com.thread.myrunable;  public class MyRunable implements Runnable {  @Override  public void run() {  System.out.println("MyRunable运行中!");  }  } |

|  |
| --- |
| package com.thread.myrunable;  public class Run {  public static void main(String[] args) {  MyRunable myRunable = new MyRunable();  Thread thread = new Thread(myRunable);//Thread(Runnable target)  thread.start();  System.out.println("main()运行结束！");  }  } |

结果：

|  |
| --- |
| main()运行结束！  MyRunable运行中! |

**1.2.3 实例变量与线程安全**

**1.2.3.1 不共享数据**

(1)

|  |
| --- |
| package com.thread.www;  public class MyThread3 extends Thread{  private int count = 5;  public MyThread3(String name) {  super();  this.setName(name);//设置线程名称  }  @Override  public void run() {  super.run();  while(count > 0) {  count--;  System.out.println("由" + this.currentThread().getName() + "计算，count=" +  count );  }  }  } |

(2)

|  |
| --- |
| package test;  import com.thread.www.MyThread3;  public class RunMyThread3 {  public static void main(String[] args) {  MyThread3 a = new MyThread3("A");  MyThread3 b = new MyThread3("B");  MyThread3 c = new MyThread3("C");  a.start();  b.start();  c.start();  }  } |

(3)结果:一共创建了3线程，每个线程都有各自的count变量，自己减少自己的count变量。变量不共享

|  |
| --- |
| 由A计算，count=3  由A计算，count=2  由A计算，count=1  由A计算，count=0  由C计算，count=4  由B计算，count=4  由B计算，count=3  由B计算，count=2  由B计算，count=1  由B计算，count=0  由C计算，count=3  由C计算，count=2  由C计算，count=1  由C计算，count=0 |

**1.2.3.2 共享数据**

1.2.3.2.1 例1:出现非线程安全问题

(1) MyThread4.java

|  |
| --- |
| package com.thread.www;  public class MyThread4 extends Thread {  private int count = 5;  @Override  public void run() {  super.run();  count--;  //此示例不要用for语句，因为使用同步后，其它线程就得不到线程运行的机会了，  //会一直由一个线程做减法运算  System.out.println("由" + this.currentThread().getName() + "计算count=" + count);  }  } |

(2) TestMyThread4.java

|  |
| --- |
| package test;  import com.thread.www.MyThread4;  public class TestMyThread4 {  public static void main(String[] args) {  MyThread4 myThread4 = new MyThread4();  Thread a = new Thread(myThread4, "a");  Thread b = new Thread(myThread4, "b");  Thread c = new Thread(myThread4, "c");  Thread d = new Thread(myThread4, "d");  Thread e = new Thread(myThread4, "e");  a.start();  b.start();  c.start();  d.start();  e.start();  }  } |

(3)结果: c和b线程同时对count进行处理，产生”非线程安全” 问题

|  |
| --- |
| 由a计算count=4  由c计算count=2  由b计算count=2  由e计算count=1  由d计算count=0 |

1.2.3.2.2 例2:

需要使多个线程间同步，即按顺序排队的方式进行减1操作

(1) 在例1基础上，更改 MyThread4.java代码如下：

|  |
| --- |
| package com.thread.www;  public class MyThread4 extends Thread {  private int count = 5;  @Override  synchronized public void run() {  super.run();  count--;  //此示例不要用for语句，因为使用同步后，其它线程就得不到线程运行的机会了，  //会一直由一个线程做减法运算  System.out.println("由" + this.currentThread().getName() + "计算count=" + count);  }  } |

(2)结果:

|  |
| --- |
| 由a计算count=4  由b计算count=3  由d计算count=2  由c计算count=1  由e计算count=0 |

synchronized使多个线程在执行run方法时，以排队方式处理。当一个线程调用run前，先判断run方法有没被上锁，若上锁，说明有其它线程在调用run方法，必须等其它线程对run方法调用结束后才可执行run方法。

Synchronized可在任意对象及方法前加锁，被加锁的代码称为”互斥区”或“临界区”。

如果不能拿到锁，该线程会不断的去尝试拿这把锁，直到能够拿到为止，且多个线程会同时争抢这把锁。

非线程安全:多个线程对同一个对象中的同一个实例变量进行操作时，会出现值更改、值不同步的情况，进而影响程序执行流程。

**1.2.4 留意i—与System.out.println()的异常**

1. MyThread5.java

|  |
| --- |
| package com.thread.www;  public class MyThread5 extends Thread {  private int i = 5;  @Override  public void run() {  System.out.println("i=" + (i--) + "threadname=" + Thread.currentThread().getName());  }  } |

2. TestMyThread5.java

|  |
| --- |
| package test;  import com.thread.www.MyThread5;  public class TestMyThread5 {  public static void main(String[] args) {  MyThread5 myThread5 = new MyThread5();  Thread a = new Thread(myThread5);  Thread b = new Thread(myThread5);  Thread c = new Thread(myThread5);  Thread d = new Thread(myThread5);  Thread e = new Thread(myThread5);  a.start();  b.start();  c.start();  d.start();  e.start();  }  } |

3.结果

|  |
| --- |
| i=5threadname=Thread-1  i=3threadname=Thread-4  i=4threadname=Thread-2  i=2threadname=Thread-3  i=1threadname=Thread-5 |

虽然println()方法在内部是同步的，但i--操作是在println之前发生的，所以有非线程同步的问题的概率。

|  |
| --- |
| public void println(String x) {  synchronized (this) {  print(x);  newLine();  }  } |

**1.3 currentThread()方法**

Thread.currentThread().getName()：代码正被哪个线程调用

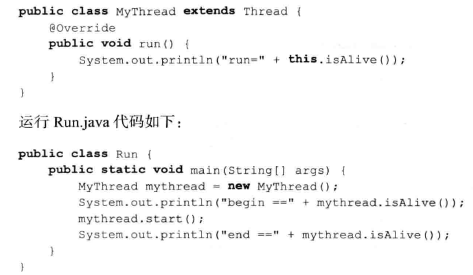


Mythread.java类的构造函数是被main线程调用，run()方法被Thread-0的线程调用。

**1.4 isAlive()方法**

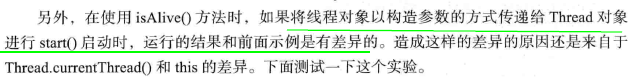
判断当前线程是否处于活动状态：已启动且尚未终止。

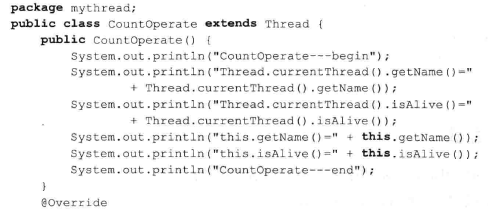
准备开始运行或正在运行的状态

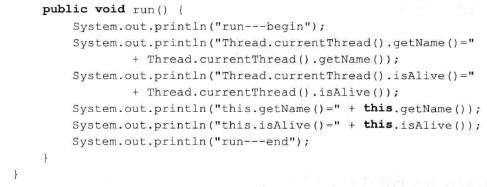


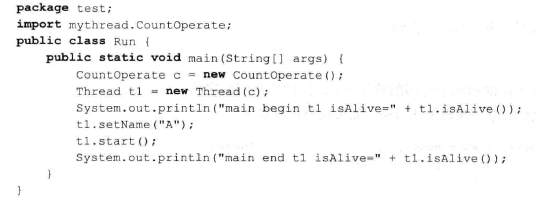


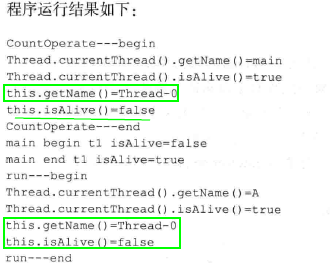
* 例2(不是太明白)







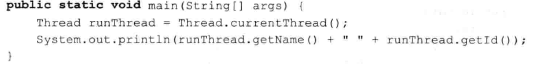


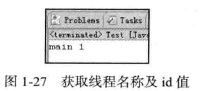


**1.5 sleep()方法**

**1.6 getId()方法**

取得线程的唯一标识





**1.7 停止线程**

* 终止正在运行的线程:

1. run方法完成后线程正常终止
2. Thread.stop()：不安全，且已被废弃，最好不用。
3. Thread.interrupted()

1.7.1 停止不了的线程：interrupt ()

Thread.interrupt ()仅是在当前线程中打了一个停止标记，并不是真正停止线程。

1.7.8 使用return(+interrupt)停止线程

* 例子:

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  while(true) {  if(this.isInterrupted()) {  System.out.println("停止了！");  return;  }  System.out.println("timer=" + System.currentTimeMillis());  }  }  } |

|  |
| --- |
| package com.thread.sleepinterupt;  public class Run2 {  public static void main(String[] args) {  MyThrrad myThrrad = new MyThrrad();  myThrrad.start();    try {  Thread.sleep(300);  } catch (InterruptedException e) {  e.printStackTrace();  }    myThrrad.interrupt();  }  } |

结果:

|  |
| --- |
| timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  timer=1503036078539  停止了！ |

* 总结:

但还是建议用”抛异常”的方法停止线程，因在catch中还可将异常往上抛，使线程停止事件得到传播。

1.7.2 判断线程是否是停止状态

* Thread.interrupted()：测试当前线程是否已中断

|  |
| --- |
| public static boolean interrupted() {  return currentThread().isInterrupted(true);  } |

调用后，线程的中断状态由该方法清除。

* thread.isInterrupted()：测试线程Thread对象是否已中断

|  |
| --- |
| public boolean isInterrupted() {  return isInterrupted(false);  } |

调用后，未清除线程的中断状态。

1.7.3 【补】能停止的线程——异常法

1.7.4 sleep()状态下停止线程

* 例子

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  super.run();  try {  System.out.println("run begin");  Thread.sleep(200000);  System.out.println("run end");  } catch (InterruptedException e) {  System.out.println("在沉睡中停止！进入catch:" + this.isInterrupted());  e.printStackTrace();  }  }  } |

|  |
| --- |
| package com.thread.sleepinterupt;  public class Run2 {  public static void main(String[] args) {  try {  MyThrrad myThrrad = new MyThrrad();  myThrrad.start();  Thread.sleep(200);  myThrrad.interrupt();  } catch (InterruptedException e) {  System.out.println("main catch");  e.printStackTrace();  }  System.out.println("main end!");  }  } |

结果：

|  |
| --- |
| run begin  main end!  在沉睡中停止！进入catch:false  java.lang.InterruptedException: sleep interrupted  at java.lang.Thread.sleep(Native Method)  at com.thread.sleepinterupt.MyThrrad.run(MyThrrad.java:11) |

* 总结:

在sleep状态下停止某一线程，会抛异常，进入catch语句，且清除停止状态，变为false

1.7.5 【补】能停止的线程——暴力停止(stop())

1.7.6 stop()与java.lang.ThreadDeath异常

* 例子:

|  |
| --- |
| package com.thread.sleepinterupt;  public class MyThrrad extends Thread{  @Override  public void run() {  try {  this.~~stop();~~  } catch (ThreadDeath e) {  System.out.println("进入了catch方法！");  e.printStackTrace();  }  }  } |

调用代码(略),结果：

|  |
| --- |
| 进入了catch方法！  java.lang.ThreadDeath  at java.lang.Thread.stop(Thread.java:836)  at com.thread.sleepinterupt.MyThrrad.run(MyThrrad.java:8) |

* 总结:
* 调用stop()会抛出java.lang.ThreadDeath异常，但通常不用显式捕捉。

//public class ThreadDeath extends Error

* stop()已被废弃，因强制停止：

可能使一些清理性工作得不到完成。

另一情况是对锁定对象”解锁”,导致数据得不到同步处理，导致数据不一样问题。

1.7.7【补】 释放锁的不良后果

**1.8 暂停线程（**suspend- resume**）**

1.8.1 【补】suspend与resume方法的使用

suspend(停止)[已废弃] –resume(恢复)

1.8.2 【补】suspend与resume方法的缺点—独占

使用不当（如获得锁对象后suspend），易造成公共同步对象的独占，其它线程无法访问公共同步对象。

1.8.3 suspend与resume方法的缺点—不同步

**1.9 yield方法**

作用:放弃当前CPU资源，让给其它任务区去占用CPU执行时间。

但放弃的时间不确定，可能刚放弃，马上又获得。

**1.10线程的优先级：**setPriority()

public final void setPriority(int newPriority)

优先级较高的线程得到CPU资源较多。

newPriority：1-10, 不在此范围抛出异常:throw new IllegalArgumentException()

|  |
| --- |
| myThrrad.setPriority(Thread.MIN\_PRIORITY);  3个常量:  public final static int MIN\_PRIORITY = 1;  public final static int NORM\_PRIORITY = 5;  public final static int MAX\_PRIORITY = 10; |

1.10.1 【代码补】线程优先级的继承特性

如A线程启动B线程，B的线程优先级同A

1.10.2 优先级具有规则性

高优先级的线程总是大部分先执行完。

当线程优先级差距很大，低优先级先调用，却不一定先执行完

即:线程优先级与代码执行顺序无关，CPU尽量将执行资源让给优先级较高的线程。

1.10.3 优先级具有随机性和不确定性

即优先级较高的线程不一定每次都先执行完。

1.10.4 看谁运行得快

**1.11 守护线程**

|  |
| --- |
| 用户线程  守护线程 |

守护线程：

仅当进程中不存在非守护线程，则守护线程自动销毁。

作用：为其它线程的运行提供便利服务

典型:垃圾回收线程

|  |
| --- |
| MyThrrad myThrrad = new MyThrrad();  myThrrad.setDaemon(true); |

**第2章 对象及变量的并发访问**

**2.1 synchronized同步方法**

“线程安全”:获得的实例变量的值经过了同步处理，不会出现脏读。

2.1.1 方法内的变量为线程安全

“非线程安全”存在于实例变量中，方法内部的私有变量，不存在该问题。

2.1.2 【可写代码】实例变量非线程安全

如多个线程共同访问1个对象中的实例变量，可能出现非线程安全问题

用线程访问的对象中如有多个实例变量，结果可能交叉。

如对象只有一个实例变量，可能覆盖

2.1.3 多个对象多个锁

Synchronized取得的是对象锁，不是把一段代码或方法(函数)当作锁。

如多个线程访问多个对象，JVM会创建多个锁。

2.1.4 synchronized方法与锁对象

1个对象一个锁

* 结论:

1. A线程先持有object对象的Lock锁，B线程可异步方式调用object对象中的非

Synchronized类型方法。

1. A线程先持有object对象的Lock锁，B线程若在此时调用object对象的Synchronized方法则需等待，即同步

2.1.5 脏读

2.1.4中已实现多个线程调用同一方法时，用Synchronized进行同步。虽在赋值时同步，取值时可能出意外，即脏读。

脏读：在读取实例变量时，此值已被其它线程更改了。

解决方法：在取值(getValue())方法前加上synchronized



2.1.6 synchronized锁重入(代码P65)

当一个线程得到一个对象锁，再次请求此对象锁是可得到的。

* 锁重入:自己可再次获得自己的内部锁。

如不可锁重入，会造成死锁。

* 锁重入也支持在父子类继承的环境中(代码P67)。

即子类(的同步方法)可通过“可重入锁”调用父类的同步方法。

2.1.7 出现异常，锁自动释放

当一个线程执行代码出现异常，其持有的锁会自动释放。

2.1.8 同步不可继承(看代码P69)

父类的方法A是synchronized的，子类重写该方法，不能继承synchronized，需在子类方法中添加synchronized

**2.2 synchronized同步语句块**

2.2.1 synchronized方法的弊端(看代码P74)

弊端:运行时间长

2.2.2 synchronized同步代码块的使用(synchronized(this))

2.2.3 用同步代码块解决同步方法的弊端(看代码P76)

缩短时间，加快运行效率。

2.2.4 一半异步，一半同步

不在synchronized块中的是异步执行，在synchronized块中就是同步执行。

2.2.5 synchronized代码块间的同步性

当一个线程访问object的一个synchronized(this)同步代码块时，其它线程对同一个object中所有其它synchronized(this)同步代码块的访问将被阻塞，说明synchronized使用的”对象监视器”是同一个。

2.2.6 synchronized（this）同步代码块锁定当前对象(看代码P82)

* 多个线程调用同一对象中不同名称的synchronized方法或synchronized同步代码块，调用效果是按顺序执行，是同步的，阻塞的。即锁定当前对象的。
* synchronized方法或synchronized(this):

1. 对其它同步方法或synchronized(this)同步代码块呈阻塞状态
2. 同一时间只有一个线程可执行同步方法/同步代码块

2.2.7将任意对象作为对象监视器

* 任意对象:实例变量及方法的参数

synchronized(非this对象x)同步代码块：

(1)在多个线程持有“对象监视器”为同一个对象的前提下，同一时间只有一个线程可执行synchronized(非this对象x)同步代码块中代码。

(2)当持有“对象监视器”为同一个对象的前提下，同一时间只有一个线程可执行synchronized(非this对象x)同步代码块中代码。

* 优点

如一个类中有很多synchronized方法，虽能实现同步，但会阻塞，影响效率。

synchronized(非this对象x)代码块中程序与同步方法是异步的，不与其他锁this方法争抢this锁，提高效率。

|  |
| --- |
| 自己的理解:  synchronized(this)与synchronized方法的对象监视器都是当前对象，相同。  synchronized(非this对象x)的对象监视器与它们不同。  持有不同对象监视器是异步效果。 |

* 解决脏读问题【可写代码】

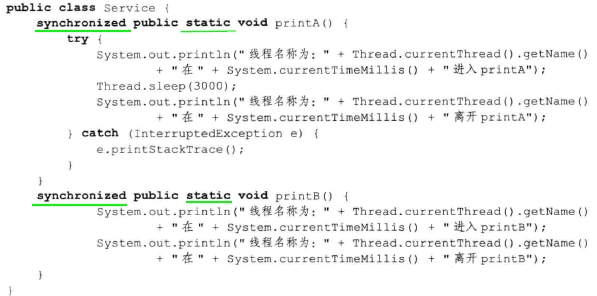
2.2.8 细化验证3个结论

synchronized(非this对象x)将x对象作为”对象监视器”：

1. 多个线程同时执行(同一x对象的)synchronized(x)同步代码块，同步效果
2. 【可写代码】当其它线程执行x对象中synchronized方法或synchronized(this)代码块，同步效果。(**自己理解**:调用synchronized方法或synchronized(this)代码块的那个当前对象是x对象)

注意: 其它线程调用不加synchronized的方法，异步。

2.2.9 静态同步synchronized方法与synchronized(class)代码块



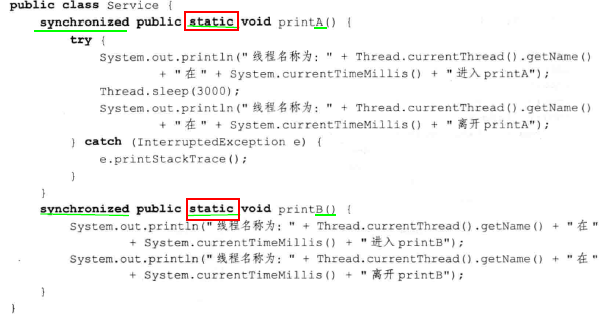
* Synchronized用在静态方法上，是对当前\*.java对应的Class类持锁。

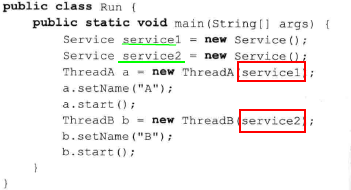
|  |
| --- |
| 比较: Synchronized用在非静态方法上，是给调用的对象上锁。 |



A与B同步，与C异步。因持有不同锁，A和B是Class锁，C是对象锁。

* Class锁可对类的所有实例起作用





结果:同步

* synchronized(class)代码块效果与synchronized static相同

如:synchronized(Service.class)

2.2.10 【可看代码】String的常量池特性

|  |
| --- |
| String a = "a";  String b = "a";  System.out.println(a==b); //true |

String常量池缓存

因此一般synchronized代码块不用String作为锁对象，而用例如new Object()

2.2.11 同步synchronized方法无限等待与解决【可看代码】

解决:使非静态方法的锁对象不一样。

2.2.12 多线程的死锁

不同的线程都在等待不可能被释放的锁，导致所有任务都无法继续完成，会造成线程”假死”,必须避免。

例如:



可用JDK自带工具检测是否有死锁。

2.2.13 内置类与静态内置类【可看代码】

2.2.14 内置类与同步：实验1(异步)

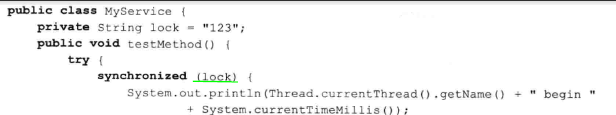
内置类中有两同步方法，但用的不同的锁(不通对象监视器)，结果异步。

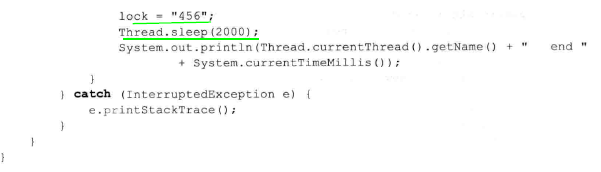
2.2.15 内置类与同步：实验2【代码没太明白】

2.2.16 锁对象的改变【代码没太明白】

只要对象不变，即使对象的属性被改变，依旧同步。

如:



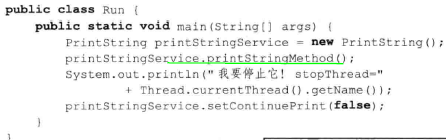


**2.3 【W】volatile关键字**

作用:使变量在多个线程间可见。

**2.3.1**【F】**死循环**



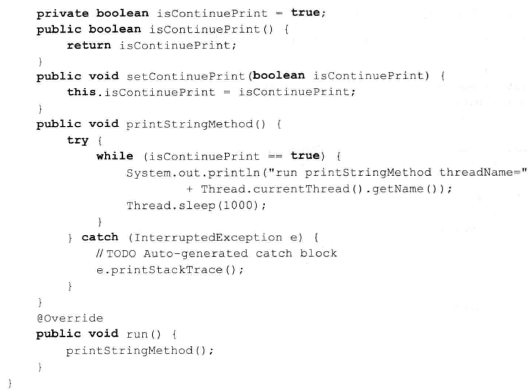


程序陷入死循环，原因:main线程一直在处理while()循环,程序不能执行后续代码。

**2.3.2**【F】**解决同步死循环(但会出现异步死循环)**

解决思路:用多线程技术



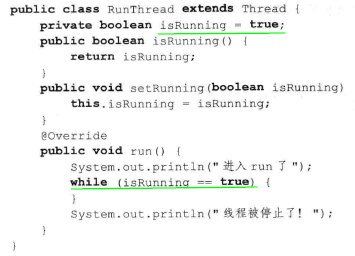




运行在-server服务器模式中64bit的JVM上时，会出现死循环.

**2.3.3** 【F】**解决异步死循环**

* 引出问题：



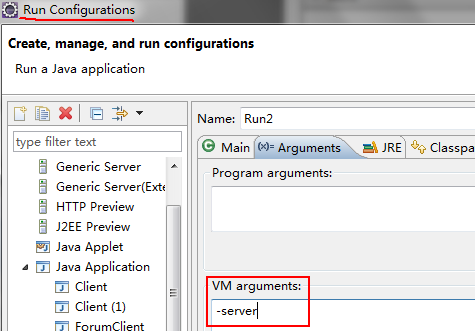


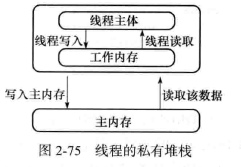
* 在win7结合JDK64bit环境中，用Eclipse运行，无死循环。
* 但当JVM设为server服务时，出现死循环。
* **原因:**

启动RunThread.java线程时，private boolean isRunning=true;存在于公共堆栈和线程私有堆栈中。

JVM设为-server时，为了线程运行效率，线程一直从私有堆栈中取isRunning值，为true.

thread.setRunning(false)更新的是公共堆栈中的isRunning

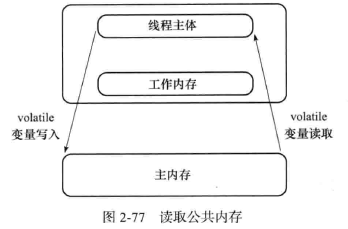




* **实质**:私有堆栈和公有堆栈中的值不同步。

**解决方法**:volatile关键字。当线程访问isRunning,强制从公有堆栈中取值。

修改: 



Volatile增加了实例变量在多个线程间的可见性，但不支持原子性

2.3.3.1【W】 synchronized与volatile比较

1.volatile是线程同步的轻量级实现，性能更好。

2. volatile只能修饰变量，synchronized可修饰方法及代码块。

3.多线程访问volatile不会阻塞，synchronized会阻塞。

4. volatile可保证数据可见性，不能保证原子性。

Synchronized可间接保证可见性，可保证原子性。因它会将私有内存和公有内存中数据同步。(理解:可免于再用volatile)

5. volatile解决变量在多个线程间的可见性

Synchronized解决多个线程访问资源的同步性(原子性)。

注:线程安全—原子性(同步性)+可见性

2.3.4 【F】Volatile非原子的特性

* volatile不具备同步性，也就不具备原子性。

volatile提示线程从共享内存中读取变量

* 例:**i++(非原子操作)，非线程安全**

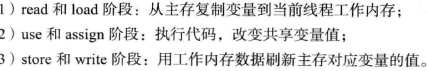
|  |
| --- |
| 1. 从内存中取出i的值 2. 计算i的值 3. 将i的值写到内存 |

假如在(2)步时，另一个线程也修改i的值，就会出现脏数据。需加Synchronized解决。

因此，Volatile本身不处理数据的原子性，而是强制对数据的读写及时影响到主内存。



结论:



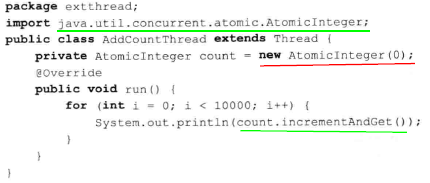
多线程时，use和assign多次出现，但并不是原子性。

在read和load之后，如主内存count变量发生修改，线程工作内存中的值由于已加载，不会感知变化，此时私有和公有内存中变量不同步了。

即用volatile修饰的变量，JVM虚拟机只保证从主内存加载到线程工作内存的值最新。

**2.3.5 使用原子类(AtomicInteger)进行i++操作**

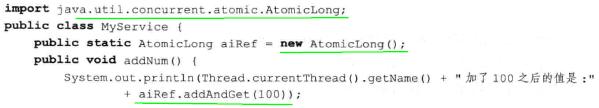
原子操作不可分割，没有锁时依旧线程安全。



**2.3.6 原子类也不完全安全**

在具有逻辑性的情况下也不是线程安全的，因多个原子方法组合后调用为非原子性。

例子:





解决方案: 

**2.3.7** 【F】Synchronized代码块有volatile的功能

Synchronized(互斥性+可见性)可使多个线程访问同一资源有同步性。

还会将私有内存和公有内存中变量同步。(理解:可免于再用volatile)

**2.4 本章总结**

**第3章 线程间通信**

线程间通信使线程在大大提高CPU利用率的同时，还使程序员对各线程任务处理过程进行把控和监督。

|  |
| --- |
| Wait/notify  生产者/消费者模式  Join  ThreadLocal |

**3.1 等待/通知(**Wait/notify**)机制**

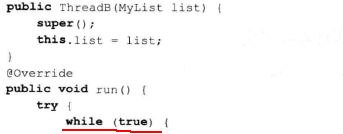
3.1.1 不使用等待/通知机制实现线程间通信:

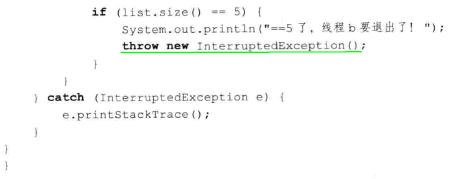
sleep() + [while(true)死循环 + 某条件下抛异常终止线程]

(1)ThreadA



(2) ThreadB





弊端: ThreadB不停通过while死循环检测某一条件，浪费CPU资源。轮询间隔越小，越浪费资源。

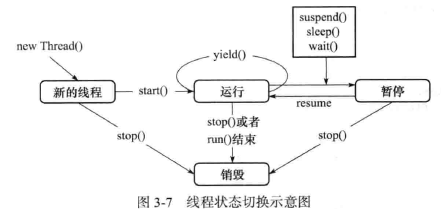
3.1.2 什么是等待/通知机制

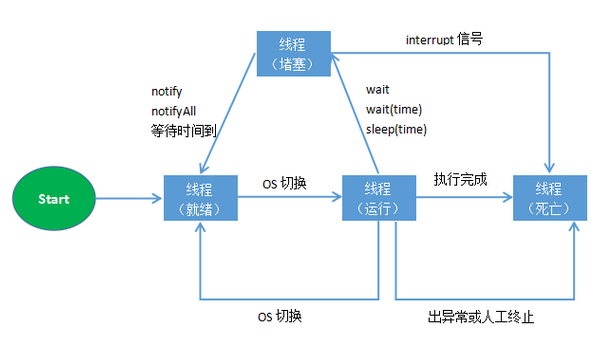
3.1.3 等待/通知机制的实现

* **wait()**
* wait()将当前线程置入”预执行队列”中，且在wait()所在的代码处停止执行。知道接到通知或被中断为止。
* 只能在同步方法或同步块中调用wait()。
* 执行wait()后，当前线程释放锁。
* 如调用wait()时，没持有锁，抛出IllegalMonitorStateException的运行时异常。
* **notify()**
* 也要在同步方法或同步块中调用
* 如调用notify ()时，没持有锁，抛出IllegalMonitorStateException的运行时异常。
* 如有多个线程等待，由线程规划器随机挑选其中一个wait状态的线程，对其发出notify，并使它等待获取该对象锁。
* 某线程执行完notify()后，不会马上释放锁，将程序执行完(即执行完synchronized代码块后)，该线程才释放锁。

因此，呈wait状态的线程也不会马上获取该对象锁。

* 当第一个获取该对象锁的wait线程运行完毕后，会释放掉该对象锁。此时，如该对象没有再次使用notify语句，则即使该对象已空闲，其它wait状态的线程由于收不到通知，还会继续阻塞在wait状态【理解:wait状态只能被唤醒】，直到该对象发出notify或notifyAll。
* 线程状态图:







1. 新建一个线程对象后，调用其start()，系统会为该线程分配CPU资源，使其处于Runable(可运行)状态。如线程抢到CPU资源，就处于Running(运行)状态
2. Runable与Running状态可相互切换。如运行一段时间被高优先级线程抢占CPU资源。
3. 线程进入Runable状态

|  |
| --- |
| 调用sleep()后时间到了  线程调用的阻塞IO已返回，阻塞方法执行完毕  线程获得了试图同步的监视器  线程等待通知，其它线程发出了通知。  处于挂起状态的线程调用了resume() |

1. Blocked（阻塞-暂停）状态

|  |
| --- |
| * 调用sleep(),主动放弃占用的CPU资源 * 调用了阻塞式IO方法，方法返回前，线程被阻塞 * 线程试图获取锁对象，但该锁对象正被其它线程持有。 * 线程等待某个通知 * 程序调用了suspend()将该线程挂起，易导致死锁，尽量避免。 |

每个锁对象都有两个队列:

|  |
| --- |
| 就绪队列: 存储将要获得锁的线程  阻塞队列：被阻塞的线程 |

一个线程被唤醒后，才会进入就绪队列，等待CPU调度

一个线程被wait后，就会进入阻塞队列，等待被唤醒

3.1.4 方法wait()锁释放与notify()锁不释放

3.1.5 当interrupt方法遇到wait方法

wait状态下调用interrupt(),会出现InterruptedException异常。

|  |
| --- |
| 执行完同步代码块会释放对象的锁；  执行同步代码块过程中，遇到异常导致线程终止，锁会被释放  执行同步代码块过程中，执行wait(),线程会释放对象锁，此线程对象进入线程等待池，等待被唤醒。 |

3.1.6 只通知一个线程

notify()一次只随机通知一个线程进行唤醒。

3.1.7 唤醒所有线程(notifyAll)

3.1.8 wait(long)的使用

等待某一时间内是否有线程对锁进行唤醒，如超时，则自动唤醒。

3.1.9 通知过早

通知过早，会打乱程序正常的运行逻辑。可能仅执行notify方法。

3.1.10 等待wait的条件发生变化

此时也会造成程序逻辑混乱

3.1.11 生产者/消费者模式实现

3.1.1.11.1 一生产一消费者:操作值[代码]

|  |
| --- |
| 消费者:空，等待；不空，唤醒  生产者:空，唤醒；不空，等待 |

3.1.1.11.2 多生产与多消费者:假死[代码]

假死:全部线程进入waitting状态，程序不再执行任何业务功能

比如:有可能生产者唤醒生产者，消费者唤醒消费者，导致大家都在等待。

原因：有可能连续唤醒同类。

解决方法:同类异类一起唤醒(用notifyAll)

3.1.1.11.3多生产与多消费:操作值(用notifyAll)

3.1.1.11.4一生产与一消费:操作栈【代码没仔细看】

private List list = new ArrayList();

生产者向堆栈List对象中放入数据，消费者从中取出数据。List的最大容量是1.

3.1.1.11.5一生产与多消费—操作栈：解决wait条件改变与假死【代码没仔细看】

3.1.1.11.5多生产与一消费—操作栈【代码没仔细看】

3.1.1.11.6多生产与多消费—操作栈【good】【代码没仔细看】

3.1.12 通过管道进行线程间通信：字节流【代码没仔细看】

管道流:在不同线程间直接传送数据。一个线程发送数据到输出管道，另一个线程从输入管道中读取数据。

使两个stream间产生通信链路。

|  |
| --- |
| PipedInputStream PipedOutputStream |

3.1.13 通过管道进行线程间通信：字符流

|  |
| --- |
| PipedReader PipedWriter |

3.1.14 实战：等待/通知之交叉备份【代码没仔细看】

**3.2 方法join的使用**

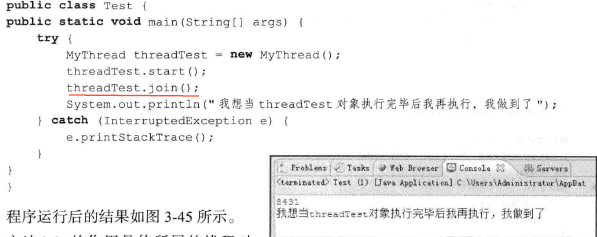
主线程创建并启动子线程，如子线程中要进行大量耗时运算，主线程将早于子线程结束。

如主线程想等子线程执行完后再结束(比如主线程想取得子线程处理好的某个数据)，用join()

Join:等待线程对象销毁

3.2.1 学习join前的铺垫

3.2.2 用join（）方法来解决



* 使所属线程对象x正常执行run()方法中的任务，而使当前线程z进行无限期的阻塞，等线程x销毁后再继续执行线程z后面的代码。
* Join具有使线程排队运行的作用。
* join与sychronized区别：

|  |
| --- |
| Join内部用wait()方法进行等待  Sychronized用对象监视器原理做同步 |

3.2.3 方法join与异常

join过程中，如当前线程对象被中断(interrupt())，则当前线程出现异常(InterruptedException)。

即join()遇到interrupt()，异常，但进程按钮还是“红色”，因被调用的线程还在继续运行。

|  |
| --- |
| 理解: Join内部用wait()方法进行等待,wait()与interrupt()一起用时，异常 |

3.2.4 方法join(long)的使用

如:join(2000) //只等2秒

3.2.5 方法join(long)与sleep(long)的区别

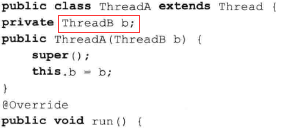
join(long)内部用wait(long)实现，所以会释放锁。

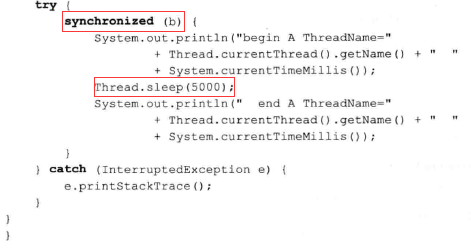
而Thread.sleep(long)不释放锁。

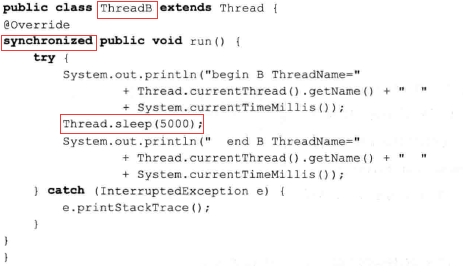
3.2.6 join()后的代码提前运行：出现意外[见代码]

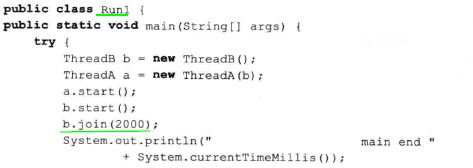
3.2.7 join()后的代码提前运行：解释意外

例子:

1.代码：  


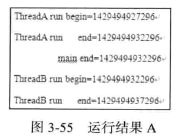


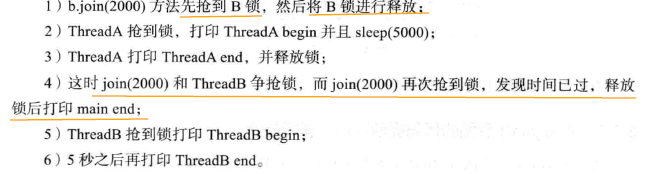




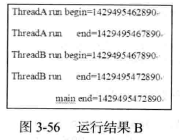
2.结果：

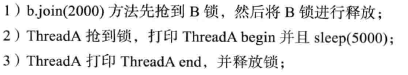
(1)结果A



解释：  


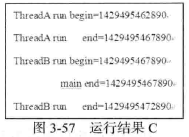
(2)结果B

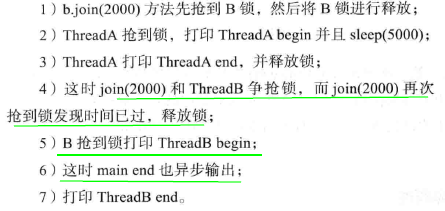


解释：  




(3)结果C



解释:  


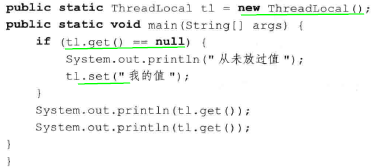
3.3 类ThreadLocal的使用

<http://blog.csdn.net/lufeng20/article/details/24314381>

实现每个线程都有自己的私有数据。

ThreadLocal解决的是每个线程绑定自己的值。

3.3.1 方法get()与null

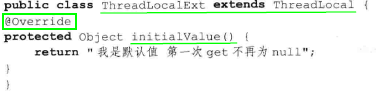




ThreadLocal解决的是变量在不同线程间的隔离性，就是不同线程有自己的值，可放入ThreadLocal中保存。

3.3.2 验证线程变量的隔离性

3.3.3 解决get()返回null问题



覆盖initialValue()方法使get()具有初始值。

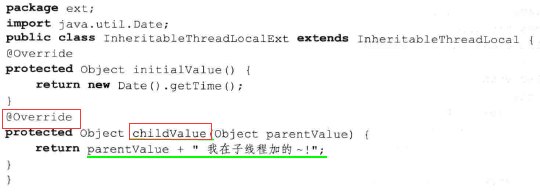
3.3.4 再次验证线程变量的隔离性[代码没仔细看]

3.4 类InheritableThreadLocal的使用

可在子线程中取得父线程继承下来的值。

3.4.1 值继承[代码没仔细看]

3.4.2 值继承再修改



如子线程取值时，主线程将InheritableThreadLocal中的值更改了，子线程取的还是旧值。

**第4章 Lock的使用**

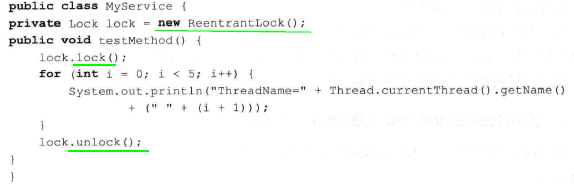
4.1 使用ReentrantLock类

JDK1.5中增加ReentrantLock类，可达到与synchronized同样的效果。

且在扩展功能上更强大，如嗅探锁定、多路分支通知等

使用上也比synchronized更灵活

4.1.1 使用ReentrantLock实现同步：测试1



4.1.2 使用ReentrantLock实现同步：测试2

调用lock.lock()代码的线程就持有了”对象监视器”，其它线程只能等待锁释放时再次争抢。

4.1.3 使用Condition实现等待/通知:错误用法与解决

1. Synchronized与wait()和notify()/notifyAll()结合，可实现等待/通知模式。

此时，被通知的线程是由JVM随机选择的。

相当于整个Lock对象中只有单一的condition对象。

2. ReentrantLock的condition+

Condition可实现多路通知，即在一个Lock对象中可创建多个condition(即对象监视器)实例，线程对象可注册在指定的condition中，从而可选择性的进行线程通知。

3．condition.await()之前必须调用lock.lock()获得同步监视器。

否则出现IllegalMonitorStateException异常。（与wait方法一样的情况）

4.1.4 正确使用Condition实现等待/通知

|  |
| --- |
| package com.thread.lock.reentrantlock;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  public class MyService {  private Lock lock = new ReentrantLock();//非公平锁  private Condition condition = lock.newCondition();    public void await() {  try {  lock.lock();  System.out.println("await时间为:" + System.currentTimeMillis());  condition.await();  } catch (InterruptedException e) {  e.printStackTrace();  } finally{  lock.unlock();  System.out.println("await锁释放了");  }  }    public void signal() {  try {  lock.lock();  System.out.println("signal时间为:" + System.currentTimeMillis());  condition.signal();  } catch (Exception e) {  } finally {  lock.unlock();  System.out.println("signal锁释放了");  }  }  } |

ThreadA.java

|  |
| --- |
| package com.thread.lock.reentrantlock;  public class ThreadA extends Thread{  private MyService service;  public ThreadA(MyService service) {  super();  this.service = service;  }  @Override  public void run() {  service.await();  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock;  public class Run {  public static void main(String[] args) {  MyService myService = new MyService();  ThreadA threadA = new ThreadA(myService);  threadA.start();  try {  Thread.sleep(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  myService.signal();  }  } |

结果：

|  |
| --- |
| await时间为:1503317900324  signal时间为:1503317903324  signal锁释放了  await锁释放了 |

* Object类中的wait()方法相当于condition类中的await()
* Object类中的wait(long timeout)方法相当于condition类中的await(long time, TimeUnit unit)
* Object类中的notify)方法相当于condition类中的signal()
* Object类中的notifyAll()方法相当于condition类中的signalAll()

4.1.5 使用多个Condition实现通知部分线程：错误用法

错误:唤醒了所有线程

4.1.6 使用多个Condition实现通知部分线程：正确用法

先对线程进行分组，再唤醒指定组中线程。

|  |
| --- |
| package com.thread.lock.reentrantlock.condition;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  public class Myservice {  private Lock lock = new ReentrantLock();  private Condition conditionA = lock.newCondition();  private Condition conditionB = lock.newCondition();    public void awaitA() {  try {  lock.lock();  System.out.println("begin awaitA时间为:" + System.currentTimeMillis()  + "TheadName=" + Thread.currentThread().getName());  conditionA.await();  System.out.println("end awaitA时间为:" + System.currentTimeMillis()  + "TheadName=" + Thread.currentThread().getName());  } catch (Exception e) {  e.printStackTrace();  } finally {  lock.unlock();  }  }    public void awaitB() {  try {  lock.lock();  System.out.println("begin awaitB时间为:" + System.currentTimeMillis()  + "TheadName=" + Thread.currentThread().getName());  conditionB.await();  System.out.println("end awaitB时间为:" + System.currentTimeMillis()  + "TheadName=" + Thread.currentThread().getName());  } catch (Exception e) {  e.printStackTrace();  } finally {  lock.unlock();  }  }    public void siganlAll\_A() {  try {  lock.lock();  System.out.println("siganlAll\_A时间为:" + System.currentTimeMillis()  + "TheadName=" + Thread.currentThread().getName());  conditionA.signalAll();  } catch (Exception e) {  e.printStackTrace();  } finally {  lock.unlock();  }  }    public void siganlAll\_B() {  try {  lock.lock();  System.out.println("siganlAll\_B时间为:" + System.currentTimeMillis()  + "TheadName=" + Thread.currentThread().getName());  conditionB.signalAll();  } catch (Exception e) {  e.printStackTrace();  } finally {  lock.unlock();  }  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock.condition;  public class ThreadA extends Thread{  private Myservice myservice;  public ThreadA(Myservice myservice) {  super();  this.myservice = myservice;  }  @Override  public void run() {  myservice.awaitA();  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock.condition;  public class ThreadB extends Thread{  private Myservice myservice;  public ThreadB(Myservice myservice) {  super();  this.myservice = myservice;  }  @Override  public void run() {  myservice.awaitB();  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock.condition;  public class Run {  public static void main(String[] args) throws InterruptedException {  Myservice myservice = new Myservice();  ThreadA threadA = new ThreadA(myservice);  threadA.setName("A");  threadA.start();    ThreadB threadB = new ThreadB(myservice);  threadB.setName("B");  threadB.start();    Thread.sleep(3000);  myservice.siganlAll\_A();  }  } |

结果：只唤醒了线程A

|  |
| --- |
| begin awaitA时间为:1503371801637TheadName=A  begin awaitB时间为:1503371801638TheadName=B  siganlAll\_A时间为:1503371804638TheadName=main  end awaitA时间为:1503371804638TheadName=A |

4.1.7 实现生产者/消费者模式：一对一交替打印

|  |
| --- |
| package com.thread.lock.reentrantlock.produceconsume;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  public class Myservice {  private Lock lock = new ReentrantLock();//非公平锁  private Condition condition = lock.newCondition();  public boolean hasValue = false;  public void set() {  try {  lock.lock();  while(hasValue==true) {  condition.await();  }    System.out.println("打印\*");  hasValue =true;  condition.signal();  } catch (Exception e) {    } finally {  lock.unlock();  }  }    public void get() {  try {  lock.lock();  while(hasValue==false) {  condition.await();  }    System.out.println("打印&");  hasValue =false;  condition.signal();  } catch (Exception e) {  } finally {  lock.unlock();  }  }  } |

4.1.8 实现生产者/消费者模式：多对多交替打印

|  |
| --- |
| package com.thread.lock.reentrantlock.produceconsume;  import java.util.concurrent.locks.Condition;  import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  public class Myservice {  private Lock lock = new ReentrantLock();//非公平锁  private Condition condition = lock.newCondition();  public boolean hasValue = false;    public void set() {  try {  lock.lock();  while(hasValue==true) {  //System.out.println("有可能\*\*连续");  condition.await();  }    System.out.println("打印\*");  hasValue =true;  //condition.signal();//假死:改成condition.signalAll()  condition.signalAll();  } catch (Exception e) {    } finally {  lock.unlock();  }  }    public void get() {  try {  lock.lock();  while(hasValue==false) {  //System.out.println("有可能&&连续");  condition.await();  }    System.out.println("打印&");  hasValue =false;  //condition.signal();//假死:改成condition.signalAll()  condition.signalAll();  } catch (Exception e) {  } finally {  lock.unlock();  }  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock.produceconsume;  public class MyThreadA extends Thread{  private Myservice myservice;  public MyThreadA(Myservice myservice) {  super();  this.myservice = myservice;  }  @Override  public void run() {  for(int i=0;i<100;i++) {  myservice.set();  }  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock.produceconsume;  public class MyThreadB extends Thread{  private Myservice myservice;  public MyThreadB(Myservice myservice) {  super();  this.myservice = myservice;  }  @Override  public void run() {  for(int i=0;i<100;i++) {  myservice.get();  }  }  } |

|  |
| --- |
| package com.thread.lock.reentrantlock.produceconsume;  public class Run {  public static void main(String[] args) {  Myservice myservice = new Myservice();  MyThreadA[] threadAs= new MyThreadA[10];  MyThreadB[] threadBs= new MyThreadB[10];  for(int i=0;i<10;i++) {  threadAs[i] = new MyThreadA(myservice);  threadBs[i] = new MyThreadB(myservice);  threadAs[i].start();  threadBs[i].start();  }  }  } |

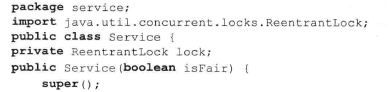
结果：

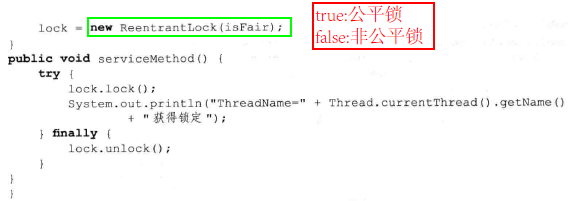
|  |
| --- |
| 打印\*  有可能\*\*连续  打印&  有可能&&连续  有可能&&连续  有可能&&连续  打印\*  有可能\*\*连续  打印& |

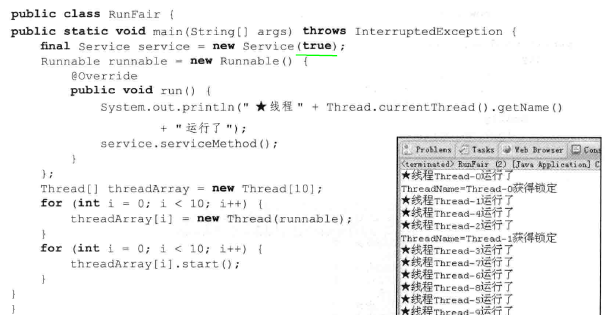
“打印&”与“打印\*”交替输出，但有可能\*\*连续与有可能&&连续不是交替输出，有时连续打印。原因是用了signalAll(),唤醒的线程可能是同类。

4.1.9 公平锁与非公平锁

* 公平锁：线程获取锁的顺序按照线程加锁的顺序分配(FIFO)。







结果:基本有序

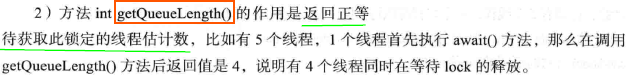
* 非公平锁：获取锁的抢占机制，随机获得锁的.

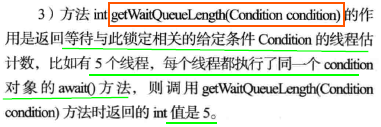


结果:基本乱序

4.1.10 方法getHoldCount()、getQueueLength()和getWaitQueueLength()的测试

getHoldCount():查询当前线程保持此锁定，即调用lock()的次数。





4.1.11 方法hasQueuedThread()、hasQueuedThreads()和hasWaiters()的测试

1.[ 代码没仔细看]



2.[代码没仔细看]



4.1.12 方法isFair()、isHeldByCurrentThread()和isLocked()的测试

* isFair()：判断是不是公平锁

ReentrantLock类默认用的非公平锁

* 
* isLocked()：查询此锁定是否由任意线程保持

4.1.13 方法lockInterruptibly()、tryLock()和tryLock(long timeout，TimeUnit unit)的测试

* lockInterruptibly() [代码没仔细看]

如当前线程未被中断，则获取锁定；

如已经中断，则出现异常。

* tryLock()

调用时，锁定未被另外一个线程保持的情况下，才能获取该锁定。

* 

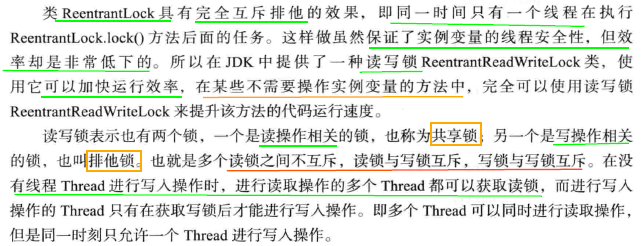
4.1.14 方法awaitUninterruptibly()的使用[代码补]

4.1.15 方法awaitUntil()的使用[代码补写]

线程在等待时间到达前，可被其它线程提前唤醒。

|  |
| --- |
| 理解: A唤醒B,B唤醒C,C唤醒A |

4.2 使用ReentrantReadWriteLock类



理解:多个线程可同时进行读操作，但同一时刻只允许一个线程进入写操作。

只要有写操作就互斥。

4.2.1 类ReentrantReadWriteLock的使用：读读共享

4.2.2 类ReentrantReadWriteLock的使用：写写互斥

4.2.3 类ReentrantReadWriteLock的使用：读写互斥

4.2.4 类ReentrantReadWriteLock的使用：写读互斥

**第5章 [#没看]定时器Timer(Android用的较多,暂不看)**

5.1 定时器Timer的使用

5.1.1 方法schedule（TimerTask task， Date time）的测试

5.1.2 方法schedule（TimerTask task， Date firstTime， long period）的测试

5.1.3 方法schedule（TimerTask task， long delay）的测试

5.1.4 方法schedule（TimerTask task， long delay， long period）的测试

5.1.5 方法scheduleAtFixedRate（TimerTask task， Date firstTime， long period）的测试

5.2 本章总结

**第6章 [见设计模式]单例模式与多线程**

6.1 立即加载/"饿汉模式"

6.2 延迟加载/"懒汉模式"

6.3 使用静态内置类实现单例模式

6.4 序列化与反序列化的单例模式实现

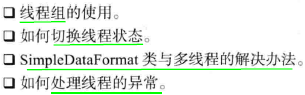
6.5 使用static代码块实现单例模式

6.6 使用enum枚举数据类型实现单例模式

6.7 完善使用enum枚举实现单例模式

6.8 本章总结

**第7章 拾遗增补**

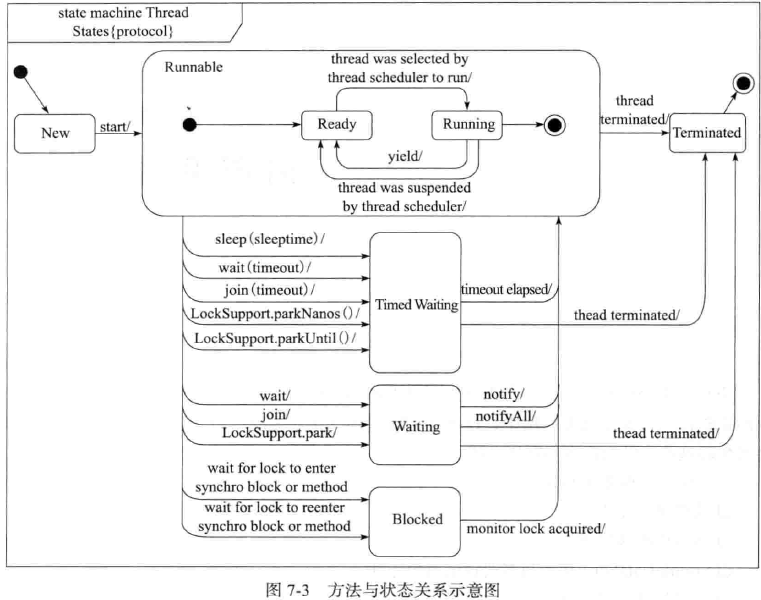


**7.1 线程的状态**





|  |
| --- |
| **Thread.State**：  1：new:线程实例化后还未执行  2：runnable可运行态。  ~~3：running运行态。~~  4：blocked堵塞状态：某个线程正在等待锁  线程在运行态会等待某些操作wait、sleep、synchronized等方法进入阻塞状态。阻塞状态的线程会在sleep方法结束后，锁释放，notify、notifyall被唤醒，suspend被调用resume方法以及其他方式被激活成runnable可运行状态。  5：waiting: 执行wait()后  6：timed-waiting：等待另一个线程来执行。取决于指定等待时间的线程。如sleep(3000)  7：terminated:死亡状态。 |



7.1.1 验证new、runnable和terminated

7.1.2 验证timed-waiting

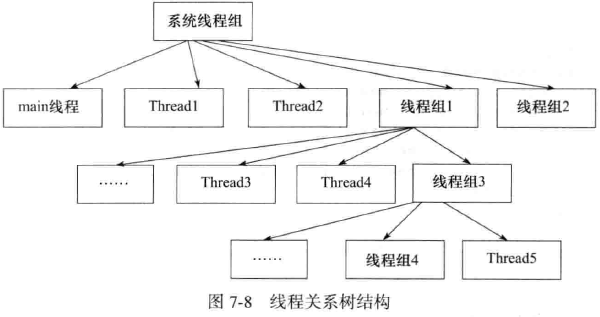
7.1.3 验证blocked

7.1.4 验证waiting

**7.2 [#没看]线程组**

线程组中可有线程对象，也可由线程组，组中还可有线程。

可批量管理线程或线程组对象.



7.2.1 线程对象关联线程组：1级关联

即父对象中有子对象,但不创建子孙对象.

7.2.2 线程对象关联线程组：多级关联

7.2.3 线程组自动归属特性

7.2.4 获取根线程组

7.2.5 线程组里加线程组

7.2.6 组内的线程批量停止

7.2.7 递归与非递归取得组内对象

7.3 **[#没看]**使线程具有有序性

7.4 **[#补看]**SimpleDateFormat非线程安全

7.5 线程中出现异常的处理

* setUncaughtExceptionHandler：对指定线程对象设置默认的异常处理器

|  |
| --- |
| package com.thread.\_07threadexception;  public class Mythread extends Thread{  @Override  public void run() {  String username = null;  System.out.println(username.hashCode());  }  } |

|  |
| --- |
| package com.thread.\_07threadexception;  import java.lang.Thread.UncaughtExceptionHandler;  public class Main1 {  public static void main(String[] args) {  Mythread mythread1 = new Mythread();  mythread1.setName("线程t1");  mythread1.setUncaughtExceptionHandler(new UncaughtExceptionHandler() {  public void uncaughtException(Thread t, Throwable e) {  System.out.println("线程:" + t.getName() + "出现了异常：");  e.printStackTrace();  }  });  mythread1.start();    Mythread mythread2 = new Mythread();  mythread2.setName("线程t2");  mythread2.start();  }  } |

结果: 线程t1的异常被捕捉了

|  |
| --- |
| 线程:线程t1出现了异常：  Exception in thread "线程t2" java.lang.NullPointerException  at com.thread.\_07threadexception.Mythread.run(Mythread.java:8)  java.lang.NullPointerException  at com.thread.\_07threadexception.Mythread.run(Mythread.java:8) |

* setDefaultUncaughtExceptionHandler:对所有线程对象设置异常处理器

|  |
| --- |
| Mythread.setDefaultUncaughtExceptionHandler(new UncaughtExceptionHandler() {  @Override  public void uncaughtException(Thread t, Throwable e) {    }  }); |

7.6 **[#没看]**线程组内处理异常

7.7 **[#没看]**线程异常处理的传递

**《java并发编程的艺术》**

**1.2　死锁**

死锁，就会造成系统功能不可用。

先来看一段代码，会引起死锁，使线程t1和t2互相等待对方释放锁。

|  |
| --- |
| public class DeadLockDemo {  privat static String A = "A";  private static String B = "B";  public static void main(String[] args) {  new DeadLockDemo().deadLock();  }  private void deadLock() {  Thread t1 = new Thread(new Runnable() {  @Override  Public void run() {  synchronized (A) {  try { Thread.currentThread().sleep(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  synchronized (B) {  System.out.println("1");  }  }  }  });  Thread t2 = new Thread(new Runnable() {  @Override  publicvoid run() {  synchronized (B) {  synchronized (A) {  System.out.println("2");  }  }  }  });  t1.start();  t2.start();  }  } |

比如t1拿到锁后，因一些异常情况没释放锁（死循环）。或者t1拿到一个数据库锁，释放锁时抛出了异常，没释放掉。

一旦出现死锁，业务可感知，因不能继续提供服务了，只能通过dump线程查看是哪个线程出现了问题，以下线程信息告诉我们是DeadLockDemo类的第42行和第31行引起的死锁。

|  |
| --- |
| "Thread-2" prio=5 tid=7fc0458d1000 nid=0x116c1c000 waiting for monitor entry [116c1b000]  java.lang.Thread.State: BLOCKED (on object monitor)  at com.ifeve.book.forkjoin.DeadLockDemo$2.run(DeadLockDemo.java:42)  - waiting to lock <7fb2f3ec0> (a java.lang.String)  - locked <7fb2f3ef8> (a java.lang.String)  at java.lang.Thread.run(Thread.java:695)  "Thread-1" prio=5 tid=7fc0430f6800 nid=0x116b19000 waiting for monitor entry [116b18000]  java.lang.Thread.State: BLOCKED (on object monitor)  at com.ifeve.book.forkjoin.DeadLockDemo$1.run(DeadLockDemo.java:31)  - waiting to lock <7fb2f3ef8> (a java.lang.String)  - locked <7fb2f3ec0> (a java.lang.String)  at java.lang.Thread.run(Thread.java:695) |

* 避免死锁的常见方法

|  |
| --- |
| ·避免一个线程同时获取多个锁。  ·避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。  ·尝试使用定时锁，使用lock.tryLock（timeout）来替代使用内部锁机制。  ·对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况。 |

**第6章　Java并发容器和框架**

**6.1　ConcurrentHashMap的实现原理与使用**

是线程安全且高效的HashMap。

**6.1.1　为什么要使用ConcurrentHashMap**

原因:并发编程中用HashMap可能导致程序死循环。而使用线程安全的HashTable效率又非常低。

(1s)线程不安全的HashMap

多线程环境下，用HashMap进行put操作会引起死循环，导致CPU利用率接近100%

所以并发情况下不能用HashMap。例如，执行以下代码会引起死循环。

|  |
| --- |
| final HashMap<String, String> map = new HashMap<String, String>(2);  Thread t = new Thread(new Runnable() {  @Override  public void run() {  for (int i = 0; i < 10000; i++) {  new Thread(new Runnable() {  @Override  public void run() {  map.put(UUID.randomUUID().toString(), "");  }  }, "ftf" + i).start();  }  }  }, "ftf");  t.start();  t.join() |

HashMap在并发执行put操作时会引起死循环，是因为多线程会导致HashMap的Entry链表

形成环形数据结构，一旦形成环形数据结构，Entry的next节点永不为空，就会产生死循环获取Entry。

（2）效率低下的HashTable

HashTable用synchronized来保证线程安全，但在线程竞争激烈的情况下HashTable

的效率非常低下。因为当一个线程访问HashTable的同步方法，其他线程也访问HashTable的同步方法时，会进入阻塞或轮询状态。如线程1使用put进行元素添加，线程2不但不能使用put方法添加元素，也不能使用get方法来获取元素，所以竞争越激烈效率越低。

（3）ConcurrentHashMap的锁分段技术可有效提升并发访问率

HashTable容器在竞争激烈的并发环境下表现出效率低下的原因是所有访问HashTable的

线程都必须竞争同一把锁，假如容器里有多把锁，每一把锁用于锁容器中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，从而可有效提高并发访问效率，这就是ConcurrentHashMap所使用的锁分段技术。

首先将数据分段存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

**6.1.2　ConcurrentHashMap的结构**

由Segment数组结构和HashEntry数组结构组成。

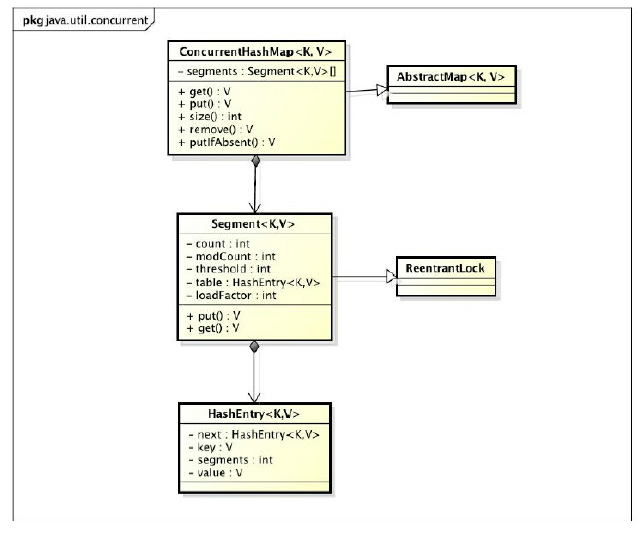
Segment是一种可重入锁（ReentrantLock），在ConcurrentHashMap里扮演锁的角色；

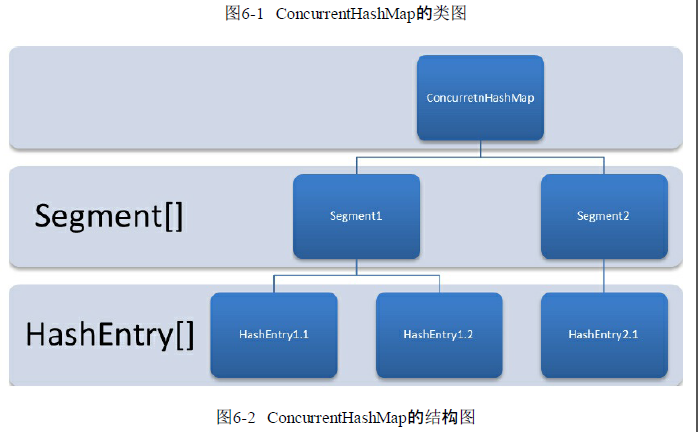
HashEntry则用于存储键值对数据。

一个ConcurrentHashMap里包含一个Segment数组。

Segment的结构和HashMap类似，是一种数组和链表结构。

一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素，每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得与它对应的Segment锁，如图6-2所示。





**6.1.3　ConcurrentHashMap的初始化**

通过initialCapacity、loadFactor和concurrencyLevel等参数来初始化segment数组、段偏移量segmentShift、段掩码segmentMask和每个segment里的HashEntry数组来实现的。

1.初始化segments数组

源代码

|  |
| --- |
| if (concurrencyLevel > MAX\_SEGMENTS)  concurrencyLevel = MAX\_SEGMENTS;  int sshift = 0;  int ssize = 1;  while (ssize < concurrencyLevel) {  ++sshift;  ssize <<= 1;  }  segmentShift = 32 - sshift;  segmentMask = ssize - 1;  this.segments = Segment.newArray(ssize); |

segments数组的长度ssize通过concurrencyLevel计算得出。为了能通过按位与的散列算法来定位segments数组的索引，必须保证segments数组的长度是2的N次方（power-of-two size），所以必须计算出一个大于或等于concurrencyLevel的最小的2的N次方值来作为segments数组的长度。假如concurrencyLevel等于14、15或16，ssize都会等于16，即容器里锁的个数也是16。

注意　concurrencyLevel的最大值是65535，这意味着segments数组的长度最大为65536，

对应的二进制是16位。

2.初始化segmentShift和segmentMask

这两个全局变量需在定位segment时的散列算法里使用，sshift等于ssize从1向左移位的次数，默认情况下concurrencyLevel等于16，1需要向左移位移动4次，所以sshift等于4。segmentShift用于定位参与散列运算的位数，segmentShift等于32减sshift，所以等于28，这里之所以用32是因为ConcurrentHashMap里的hash()方法输出的最大数是32位的。segmentMask是散列运算的掩码，等于ssize减1，即15，掩码的二进制各个位的值都是1。因为ssize的最大长度是65536，所以segmentShift最大值是16，segmentMask最大值是65535，对应的二进制是16位，每个位都是1。

3.初始化每个segment

输入参数initialCapacity是ConcurrentHashMap的初始化容量，loadfactor是每个segment的负载因子，在构造方法里需通过这两个参数来初始化数组中的每个segment。

上面代码中的变量cap就是segment里HashEntry数组的长度，它等于initialCapacity除以ssize

的倍数c，如果c大于1，就会取大于等于c的2的N次方值，所以cap不是1，就是2的N次方。

segment的容量threshold＝（int）cap\*loadFactor，默认情况下initialCapacity等于16，loadfactor等于0.75，通过运算cap等于1，threshold等于零。

**6.1.4　定位Segment**

既然ConcurrentHashMap使用分段锁Segment来保护不同段的数据，那么在插入和获取元素

时，须先通过散列算法定位到Segment。可以看到ConcurrentHashMap会首先使用

Wang/Jenkins hash的变种算法对元素的hashCode进行一次再散列。

|  |
| --- |
| private static int hash(int h) {  h += (h << 15) ^ 0xffffcd7d;  h ^= (h >>> 10);  h += (h << 3);  h ^= (h >>> 6);  h += (h << 2) + (h << 14);  return h ^ (h >>> 16);  } |

之所以进行再散列，目的是减少散列冲突，使元素能够均匀地分布在不同的Segment上，

从而提高容器的存取效率。假如散列的质量差到极点，那么所有的元素都在一个Segment中，不仅存取元素缓慢，分段锁也会失去意义。笔者做了一个测试，不通过再散列而直接执行散列计算。

|  |
| --- |
| System.out.println(Integer.parseInt("0001111", 2) & 15);  System.out.println(Integer.parseInt("0011111", 2) & 15);  System.out.println(Integer.parseInt("0111111", 2) & 15);  System.out.println(Integer.parseInt("1111111", 2) & 15); |

计算后输出的散列值全是15，通过这个例子可以发现，如果不进行再散列，散列冲突会非

常严重，因为只要低位一样，无论高位是什么数，其散列值总是一样。我们再把上面的二进制数据进行再散列后结果如下（为了方便阅读，不足32位的高位补了0，每隔4位用竖线分割下）。

|  |
| --- |
| 0100｜0111｜0110｜0111｜1101｜1010｜0100｜1110  1111｜0111｜0100｜0011｜0000｜0001｜1011｜1000  0111｜0111｜0110｜1001｜0100｜0110｜0011｜1110  1000｜0011｜0000｜0000｜1100｜1000｜0001｜1010 |

可以发现，每一位的数据都散列开了，通过这种再散列能让数字的每一位都参加到散列

运算当中，从而减少散列冲突。ConcurrentHashMap通过以下散列算法定位segment。

|  |
| --- |
| final Segment<K,V> segmentFor(int hash) {  return segments[(hash >>> segmentShift) & segmentMask];  } |

默认情况下segmentShift为28，segmentMask为15，再散列后的数最大是32位二进制数据，

向右无符号移动28位，意思是让高4位参与到散列运算中，（hash>>>segmentShift）

&segmentMask的运算结果分别是4、15、7和8，可以看到散列值没有发生冲突。

**6.1.5　ConcurrentHashMap的操作**

1.get操作

先经过一次再散列，然后使用这个散列值通过散列运算定位到Segment，再通过散列算法定位到元素，代码如下。

|  |
| --- |
| public V get(Object key) {  int hash = hash(key.hashCode());  return segmentFor(hash).get(key, hash);  } |

高效之处在于整个get过程不需加锁，除非读到的值是空才会加锁重读。

get方法里将要使用的共享变量都定义成volatile类型，如用于统计当前Segement大小的count字段和用于存储值的HashEntry的value。

volatile的变量，能在线程间保持可见性，能被多线程同时读，且保证不会读到过期的值，但只能被单线程写（有一种情况可被多线程写，就是写入的值不依赖于原值），在get操作里只需要读不需要写共享变量count和value，所以可以不用加锁。

之所以不会读到过期的值，是因为根据Java内存模型的happen before原则，对volatile字段的写入先于读，即使两个线程同时修改和获取volatile变量，get操作也能拿到最新的值，这是用volatile替换锁的经典应用场景。

|  |
| --- |
| transient volatile int count;  volatile V value; |

在定位元素的代码里可发现，定位HashEntry和定位Segment的散列算法虽一样，都与数组的长度减去1再相“与”，但是相“与”的值不一样，定位Segment使用的是元素的hashcode通过再散列后得到的值的高位，而定位HashEntry直接使用的是再散列后的值。目的是避免两次散列后的值一样，虽然元素在Segment里散列开了，但是却没有在HashEntry里散列开。

|  |
| --- |
| hash >>> segmentShift) & segmentMask　　 // 定位Segment所使用的hash算法  int index = hash & (tab.length - 1);　　 // 定位HashEntry所使用的hash算法 |

2.put操作

需对共享变量进行写入，所以为了线程安全，在操作共享变量时必须加锁。put方法首先定位到Segment，然后在Segment里进行插入操作。

两个步骤:

第一步判断是否需要对Segment里的HashEntry数组进行扩容，

第二步定位添加元素的位置，然后将其放在HashEntry数组里。

（1）是否需要扩容

在插入元素前会先判断Segment里的HashEntry数组是否超过容量（threshold），如果超过阈

值，则对数组进行扩容。值得一提的是，Segment的扩容判断比HashMap更恰当，因为HashMap是在插入元素后判断元素是否已经到达容量的，如果到达了就进行扩容，但是很有可能扩容之后没有新元素插入，这时HashMap就进行了一次无效的扩容。

（2）如何扩容

扩容时，首先会创建一个容量是原来容量两倍的数组，然后将原数组里的元素进行再散列后插入到新的数组里。为了高效，ConcurrentHashMap不会对整个容器进行扩容，而只对某segment扩容。

3.size操作

如要统计整个ConcurrentHashMap里元素大小，就必须统计所有Segment里元素的大小

后求和。

Segment里的全局变量count是一个volatile变量，那么在多线程场景下，是不是直接把所有Segment的count相加就可以得到整个ConcurrentHashMap大小了呢？不是的，虽然相加时可以获取每个Segment的count的最新值，但是可能累加前使用的count发生了变化，那么统计结果就不准了。所以，最安全的做法是在统计size时把所有Segment的put、remove和clean方法全部锁住，但是这种做法显然非常低效。

因在累加count操作过程中，之前累加过的count发生变化的几率非常小，所以ConcurrentHashMap的做法是先尝试2次通过不锁住Segment的方式来统计各个Segment大小，如果统计的过程中，容器的count发生了变化，则再采用加锁方式来统计所有Segment大小。

那么ConcurrentHashMap是如何判断在统计的时候容器是否发生了变化呢？使用modCount

变量，在put、remove和clean方法里操作元素前都会将变量modCount加1，那么在统计size前后比较modCount是否发生变化，从而得知容器的大小是否发生变化。

**6.2　ConcurrentLinkedQueue(非阻塞)**

并发编程中，有时需用线程安全的队列。

实现方式：

* 阻塞算法

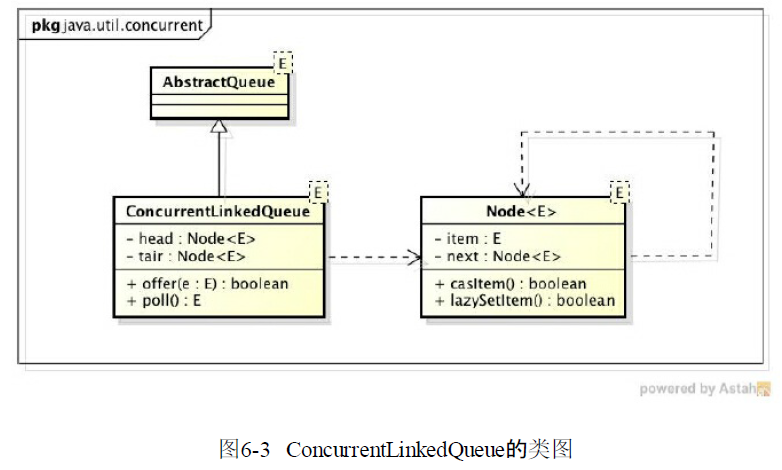
使用阻塞算法的队列可用一个锁（入队和出队用同一把锁）或两个锁（入队和出队用不同的锁）等方式来实现。

* 非阻塞算法。

ConcurrentLinkedQueue:基于链接节点的无界线程安全队列，采用先进先出规则对节点排序，添加元素时，会添加到队尾；获取元素时，会返回队头元素。

采用了“wait-free”算法（即CAS算法）实现

**6.2.1　ConcurrentLinkedQueue的结构**



由head和tail节点组成，每个节点（Node）由节点元素（item）和指向下一节点（next）的引用组成，节点与节点间就是通过这个next关联起来，从而组成一张链表结构的队列。

默认情况下head节点存储的元素为空，tail节点等于head节点。

private transient volatile Node<E> tail = head;

**6.2.2　入队列**

1.过程

将入队节点添加到队列的尾部。

示例。假设想在一个队列中依次插入4个节点，为了帮助大家理解，每添加一个节点就做了一个队列的快照图，如图6-4所示。



* 过程:

·添元素1。队列更新head节点的next节点为元素1节点。又因为tail节点默认情况下等于head节点，所以它们的next节点都指向元素1节点。

·添加元素2。队列首先设置元素1节点的next节点为元素2节点，然后更新tail节点指向元素2节点。

·添加元素3，设置tail节点的next节点为元素3节点。

·添加元素4，设置元素3的next节点为元素4节点，然后将tail节点指向元素4节点。

* 发现入队主要做两件事：

|  |
| --- |
| 第一是将入队节点设置成当前队列尾节点的下一个节点；  第二是更新tail节点  如tail节点的next节点不为空，则将入队节点设置成tail节点，  如tail节点的next节点为空，则将入队节点设置成tail的next节点，所以tail节点不总是尾节点（理解这一点对于我们研究源码会非常有帮助）。 |

上面从单线程入队角度理解了入队过程，但多个线程同时进行入队的情况就变得更复杂了，因为可能会出现其他线程插队的情况。

如有一个线程正入队，它必须先获取尾节点，然后设置尾节点的下一个节点为入队节点，但这时可能有另一个线程插队了，那么队列的尾节点就会发生变化，这时当前线程要暂停入队操作，然后重获尾节点。

通过源码来详细分析一下它是如何使用CAS算法来入队的。

|  |
| --- |
| public boolean offer(E e) {  if (e == null) throw new NullPointerException();  // 入队前，创建一入队节点  Node<E> n = new Node<E>(e);  retry:  // 死循环，入队不成功反复入队。  for (;;) {  // 创建一个指向tail节点的引用  Node<E> t = tail;  // p用来表示队列的尾节点，默认情况下等于tail节点。  Node<E> p = t;  for (int hops = 0; ; hops++) {  // 获得p节点的下一个节点。  Node<E> next = succ(p);  // next节点不为空，说明p不是尾节点，需要更新p后在将它指向next节点  if (next != null) {  // 循环了两次及其以上，并且当前节点还是不等于尾节点  if (hops > HOPS && t != tail)  continue retry;  p = next;  }  // 如果p是尾节点，则设置p节点的next节点为入队节点。  else if (p.casNext(null, n)) {  /\*如果tail节点有大于等于1个next节点，则将入队节点设置成tail节点，  更新失败了也没关系，因为失败了表示有其他线程成功更新了tail节点\*/  if (hops >= HOPS)  casTail(t, n); // 更新tail节点，允许失败  return true;  }  // p有next节点,表示p的next节点是尾节点，则重新设置p节点  else {  p = succ(p);  }  }  }  } |

从源代码角度看，整个入队过程主要做两件事：

第一是定位出尾节点；

第二是使用CAS算法将入队节点设置成尾节点的next节点，如不成功则重试。

2.定位尾节点

尾节点可能是tail节点或tail节点的next节点。

代码循环体中的第一个if就是判断tail是否有next节点，有则表示next节点可能是尾节点。获取tail节点的next节点需注意的是p节点等于p的next节点的情况，只有一种可能就是p节点和p的next节点都等于空，表示这个队列刚初始化，正准备添加节点，所以需要返回head节点。获取p节点的next节点代码如下。

|  |
| --- |
| final Node<E> succ(Node<E> p) {  Node<E> next = p.getNext();  return (p == next) head : next;  } |

3.设置入队节点为尾节点

p.casNext（null，n）方法用于将入队节点设置为当前队尾节点的next节点，

如果p是null，表p是当前队列的尾节点

如不为null，表示有其他线程更新了尾节点，则需要重新获取当前队列的尾节点。

4.HOPS的设计意图

上面分析过对于先进先出的队列入队所要做的事情是将入队节点设置成尾节点，doug lea

写的代码和逻辑还是稍微有点复杂。那么，我用以下方式来实现是否可行？

public boolean offer(E e) {

if (e == null)

throw new NullPointerException();

Node<E> n = new Node<E>(e);

for (;;) {

Node<E> t = tail;

if (t.casNext(null, n) && casTail(t, n)) {

return true;

}

}

}

让tail节点永远作为队列的尾节点，这样实现代码量非常少，且逻辑清晰和易懂。

但有缺点，每次都需使用循环CAS更新tail节点。

如能减少CAS更新tail节点的次数，就能提高入队效率，所以doug lea使用hops变量来控制并减少tail节点的更新频率，并不是每次节点入队后都将tail节点更新成尾节点，而是当tail节点和尾节点的距离大于等于常量HOPS的值（默认等于1）时才更新tail节点，tail和尾节点的距离越长，使用CAS更新tail节点的次数就会越少，但是距离越长带来的负面效果就是每次入队时定位尾节点的时间就越长，因为循环体需要多循环一次来定位出尾节点，但是这样仍然能提高入队的效率，因为从本质上来看它通过增加对volatile变量的读操作来减少对volatile变量的写操作，而对volatile变量的写操作开销要远远大于读操作，所以入队效率会有所提升。

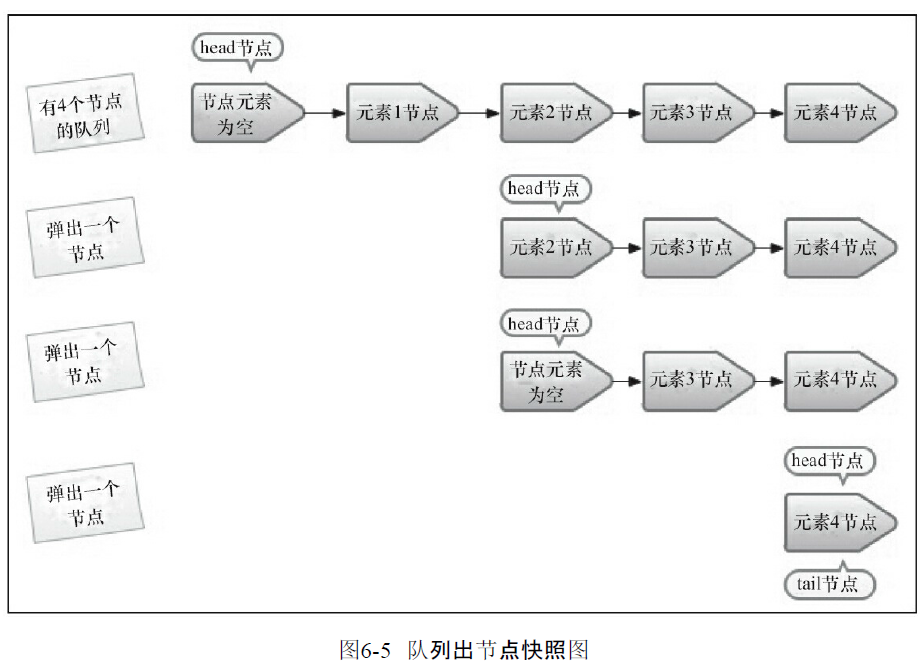
private static final int HOPS = 1;

注意　入队方法永远返回true，所以不要通过返回值判断入队是否成功。

**6.2.3　出队列**

从队列里返回一个节点元素，并清空该节点对元素的引用。

让我们通过每个节点出队的快照来观察一下head节点的变化，如图6-5所示。



从图中可知，并不是每次出队时都更新head节点

当head节点里有元素时，直接弹出head节点里的元素，而不会更新head节点。

只有当head节点里没元素时，出队操作才会更新head节点。

也是通过hops变量来减少使用CAS更新head节点的消耗，从而提高出队效率。

源码分析

public E poll() {

Node<E> h = head;

// p表示头节点，需要出队的节点

Node<E> p = h;

for (int hops = 0;; hops++) {

// 获取p节点的元素

E item = p.getItem();

// 如果p节点的元素不为空，使用CAS设置p节点引用的元素为null,

// 如果成功则返回p节点的元素。

if (item != null && p.casItem(item, null)) {

if (hops >= HOPS) {

// 将p节点下一个节点设置成head节点

Node<E> q = p.getNext();

updateHead(h, (q != null) q : p);

}

return item;

}

// 如果头节点的元素为空或头节点发生了变化，这说明头节点已经被另外

// 一个线程修改了。那么获取p节点的下一个节点

Node<E> next = succ(p);

// 如果p的下一个节点也为空，说明这个队列已经空了

if (next == null) {

// 更新头节点。

updateHead(h, p);

break;

}

// 如果下一个元素不为空，则将头节点的下一个节点设置成头节点

p = next;

}

return null;

}

首先获取头节点元素，然后判断头节点元素是否为空，如为空，表另外一个线程已

经进行了一次出队操作将该节点的元素取走，如不为空，则使用CAS的方式将头节点的引用设置成null，如果CAS成功，则直接返回头节点的元素，如果不成功，表示另外一个线程已经进行了一次出队操作更新了head节点，导致元素发生了变化，需要重新获取头节点。

**6.3　Java中的阻塞队列**

Java中阻塞队列的4种处理方式，Java 7中提供的7种阻塞队列

**6.3.1　什么是阻塞队列**

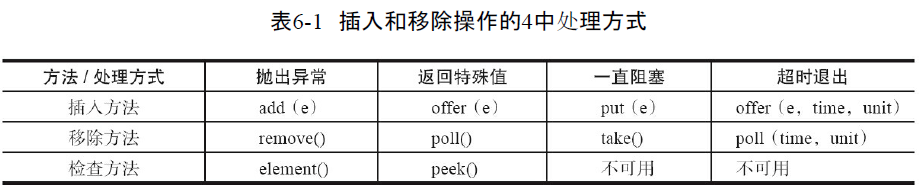
* 阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。

1）支持阻塞的插入方法：当队列满时，队列会阻塞插入元素的线程，直到队列不满。

2）支持阻塞的移除方法：队列为空时，获取元素的线程会等待队列变为非空。

常用于生产者和消费者场景，生产者是向队列里添元素的线程，消费者是从队列里取元素的线程。阻塞队列就是生产者用来存放元素、消费者用来获取元素的容器。

* 阻塞队列不可用时，两附加操作提供了4种处理方式，如表6-1所示。



·抛出异常：当队列满时，如果再往队列里插入元素，会抛出IllegalStateException（"Queue

full"）异常。当队列空时，从队列里获取元素会抛出NoSuchElementException异常。

·返回特殊值：当往队列插入元素时，会返回元素是否插入成功，成功返回true。

如是移除方法，则是从队列里取出一个元素，如果没有则返回null。

·一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者

线程，直到队列可用或响应中断退出。当队列空时，如果消费者线程从队列里take元素，队列会阻塞住消费者线程，直到队列不为空。

·超时退出：当阻塞队列满时，如生产者线程往队列里插入元素，队列会阻塞生产者线程

一段时间，如果超过了指定的时间，生产者线程就会退出。

put和take分别尾首含有字母t，offer和poll都含有字母o。

注意　如是无界阻塞队列，队列不可能会出现满的情况，所以使用put或offer方法永

远不会被阻塞，而且使用offer方法时，该方法永远返回true。

**6.3.2　Java里的阻塞队列**

JDK 7提供了7个阻塞队列

·ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。

·LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。

·PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。

·DelayQueue：一个使用优先级队列实现的无界阻塞队列。

·SynchronousQueue：一个不存储元素的阻塞队列。

·LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。

·LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

**1.ArrayBlockingQueue**

数组实现的有界阻塞队列。先进先出（FIFO）的原则对元素进行排序。

默认不保证线程公平的访问队列，所谓公平访问队列是指阻塞的线程，可以按照阻塞的先后顺序访问队列，即先阻塞线程先访问队列。非公平性是对先等待的线程是非公平的，当队列可用时，阻塞的线程都可争夺访问队列的资格，有可能先阻塞的线程最后才访问队列。为了保证公平性，通常会降低吞吐量。我们可以使用以下代码创建一个公平的阻塞队列。

|  |
| --- |
| ArrayBlockingQueue fairQueue = new ArrayBlockingQueue(1000,true);  访问者的公平性是使用可重入锁实现的，代码如下。  public ArrayBlockingQueue(int capacity, boolean fair) {  if (capacity <= 0)  throw new IllegalArgumentException();  this.items = new Object[capacity];  lock = new ReentrantLock(fair);  notEmpty = lock.newCondition();  notFull = lock.newCondition();  } |

**2.LinkedBlockingQueue**

链表实现的有界阻塞队列。默认和最大长度为Integer.MAX\_VALUE。

先进先出的原则对元素进行排序。

**3.PriorityBlockingQueue**

支持优先级的无界阻塞队列。默认元素采取自然顺序升序排列。也可自定义类实现compareTo()方法来指定元素排序规则，或者初始化PriorityBlockingQueue时，指定构造参数Comparator来对元素进行排序。需要注意的是不能保证同优先级元素的顺序。

**4.DelayQueue**

支持延时获取元素的无界阻塞队列。用PriorityQueue实现。

队列中的元素必须实现Delayed接口，创建元素时可指定多久才能从队列中获取当前元素。

只有在延迟期满时才能从队列中提取元素。

* 应用场景。

·缓存系统的设计：保存缓存元素的有效期，使用一个线程循环查询DelayQueue，一旦能从DelayQueue中获取元素时，表缓存有效期到了。

·定时任务调度：使用DelayQueue保存当天将会执行的任务和执行时间，一旦从

DelayQueue中获取到任务就开始执行，比如TimerQueue就是使用DelayQueue实现的。

(1)如何实现Delayed接口

DelayQueue队列的元素必须实现Delayed接口。可参考ScheduledThreadPoolExecutor

里ScheduledFutureTask类的实现，一共有三步。

第一步：对象创建时，初始化基本数据。用time记录当前对象延迟到什么时候可使用，使用sequenceNumber来标识元素在队列中的先后顺序。代码如下。

|  |
| --- |
| private static final AtomicLong sequencer = new AtomicLong(0);  ScheduledFutureTask(Runnable r, V result, long ns, long period) {  ScheduledFutureTask(Runnable r, V result, long ns, long period) {  super(r, result);  this.time = ns;  this.period = period;  this.sequenceNumber = sequencer.getAndIncrement();  } |

第二步：实现getDelay方法，返回当前元素还需延时多长时间，单位是纳秒，代码如下。

|  |
| --- |
| public long getDelay(TimeUnit unit) {  return unit.convert(time - now(), TimeUnit.NANOSECONDS);// 纳秒  } |

因实现getDelay()方法时可指定任意单位，一旦以秒或分作为单位，而延时时间又精确不到

纳秒就麻烦了。使用时请注意当time小于当前时间时，getDelay会返回负数。

第三步：实现compareTo方法来指定元素的顺序。例如，让延时时间最长的放在队列的末

尾。实现代码如下。

|  |
| --- |
| public int compareTo(Delayed other) {  if (other == this)　　// compare zero ONLY if same object  return 0;  if (other instanceof ScheduledFutureTask) {  ScheduledFutureTask<> x = (ScheduledFutureTask<>)other;  long diff = time - x.time;  if (diff < 0)  return -1;  else if (diff > 0)  return 1;  else if (sequenceNumber < x.sequenceNumber)  return -1;  else  return 1;  }  long d = (getDelay(TimeUnit.NANOSECONDS) -  other.getDelay(TimeUnit.NANOSECONDS));  return (d == 0) 0 : ((d < 0) -1 : 1);  } |

(2)如何实现延时阻塞队列

当消费者从队列里获取元素时，如果元素没有达到延时时间，就阻塞当前线程。

|  |
| --- |
| long delay = first.getDelay(TimeUnit.NANOSECONDS);  if (delay <= 0)  return q.poll();  else if (leader != null)  available.await();  else {  Thread thisThread = Thread.currentThread();  leader = thisThread;  try {  available.awaitNanos(delay);  } finally {  if (leader == thisThread)  leader = null;  }  } |

leader:等待获取队列头部元素的线程。如leader不等于空，表示已有线程在等待获取队列的头元素。所以，使用await()方法让当前线程等待信号。

如果leader等于空，则把当前线程设置成leader，并使用awaitNanos()方法让当前线程等待接收信号或等待delay时间。

**5.SynchronousQueue**

不存储元素的阻塞队列。每一个put操作必须等待一个take操作，否则不能继续添加元素。

支持公平访问队列。默认情况下线程采用非公平性策略访问队列。使用以下构造方法

可创建公平性访问的SynchronousQueue，如果设置为true，则等待的线程会采用先进先出的

顺序访问队列。

|  |
| --- |
| public SynchronousQueue(boolean fair) {  transferer = fair new TransferQueue() : new TransferStack();  } |

SynchronousQueue可看成是一个传球手，负责把生产者线程处理的数据直接传递给消费

者线程。队列本身并不存储任何元素，非常适合传递性场景。SynchronousQueue的吞吐量高于LinkedBlockingQueue和ArrayBlockingQueue。

**6.LinkedTransferQueue**

由链表结构组成的无界阻塞TransferQueue队列。相对于其他阻塞队列，多了tryTransfer和transfer方法。

(1)transfer方法

如当前有消费者正在等待接收元素（消费者使用take()方法或带时间限制的poll()方法时），transfer方法可把生产者传入的元素立刻transfer（传输）给消费者。

如没有消费者在等待接收元素，transfer方法会将元素存放在队列的tail节点，并等到该元素被消费者消费了才返回。

transfer方法的关键代码如下。

|  |
| --- |
| Node pred = tryAppend(s, haveData);  return awaitMatch(s, pred, e, (how == TIMED), nanos); |

第一行代码是试图把存放当前元素的s节点作为tail节点。

第二行代码是让CPU自旋等待消费者消费元素。因自旋会消耗CPU，所以自旋一定次数后使用Thread.yield()方法来暂停当前正在执行的线程，并执行其他线程。

(2)tryTransfer方法

用来试探生产者传入的元素是否能直接传给消费者。

如没有消费者等待接收元素，则返回false。

和transfer方法的区别是tryTransfer方法无论消费者是否接收，方法立即返回，而transfer方法是必须等到消费者消费了才返回。

对于带有时间限制的tryTransfer（E e，long timeout，TimeUnit unit）方法，试图把生产者传入的元素直接传给消费者，但是如果没有消费者消费该元素则等待指定的时间再返回，如果超时还没消费元素，则返回false，如果在超时时间内消费了元素，则返回true。

**7.LinkedBlockingDeque**

由链表结构组成的双向阻塞队列。

双向队列:可从队列两端插入和移出元素。

因多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。

多了addFirst、addLast、offerFirst、offerLast、peekFirst和peekLast等方法

以First单词结尾的方法，表示插入、获取（peek）或移除双端队列的第一个元素。

以Last单词结尾的方法，表示插入、获取或移除双端队列的最后一个元素。

另外，插入方法add等同于addLast，移除方法remove等效于removeFirst。但是take方法却等同于takeFirst，不知道是不是JDK的bug，使用时还是用带有First和Last后缀的方法更清楚。在初始化LinkedBlockingDeque时可以设置容量防止其过度膨胀。另外，双向阻塞队列可以运用在“工作窃取”模式中。

**6.3.3　阻塞队列的实现原理**

如果队列是空的，消费者会一直等待，当生产者添加元素时，消费者是如何知道当前队列

有元素的呢？如何让生产者和消费者进行高效率的通信呢？

* 用通知模式:当生产者往满的队列里添加元素时会阻塞住生产者，当消费者消费了一个队列中的元素后，会通知生产者当前队列可用。

通过查看JDK源码发现ArrayBlockingQueue使用了Condition来实现，代码如下。

|  |
| --- |
| private final Condition notFull;  private final Condition notEmpty;  public ArrayBlockingQueue(int capacity, boolean fair) {  // 省略其他代码  notEmpty = lock.newCondition();  notFull = lock.newCondition();  }  public void put(E e) throws InterruptedException {  checkNotNull(e);  final ReentrantLock lock = this.lock;  lock.lockInterruptibly();  try {  while (count == items.length)  notFull.await();  insert(e);  } finally {  lock.unlock();  }  }  public E take() throws InterruptedException {  final ReentrantLock lock = this.lock;  lock.lockInterruptibly();  try {  while (count == 0)  notEmpty.await();  return extract();  } finally {  lock.unlock();  }  }  private void insert(E x) {  items[putIndex] = x;  putIndex = inc(putIndex);  ++count;  notEmpty.signal();  } |

往队列里插入一元素时，如队列不可用，阻塞生产者主要通过LockSupport.park（this）来实现。

|  |
| --- |
| public final void await() throws InterruptedException {  if (Thread.interrupted())  throw new InterruptedException();  Node node = addConditionWaiter();  int savedState = fullyRelease(node);  int interruptMode = 0;  while (!isOnSyncQueue(node)) {  LockSupport.park(this);  if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)  break;  }  if (acquireQueued(node, savedState) && interruptMode != THROW\_IE)  interruptMode = REINTERRUPT;  if (node.nextWaiter != null) // clean up if cancelled  unlinkCancelledWaiters();  if (interruptMode != 0)  reportInterruptAfterWait(interruptMode);  } |

继续进入源码，发现调用setBlocker先保存一下将要阻塞的线程，然后调用unsafe.park阻塞

当前线程。

|  |
| --- |
| public static void park(Object blocker) {  Thread t = Thread.currentThread();  setBlocker(t, blocker);  unsafe.park(false, 0L);  setBlocker(t, null);  } |

unsafe.park是个native方法，代码如下。

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

park方法会阻塞当前线程，只有以下4种情况中的一种发生时，该方法才会返回。

·与park对应的unpark执行或已经执行时。“已经执行”是指unpark先执行，然后再执行park的情况。

·线程被中断时。

·等待完time参数指定的毫秒数时。

·异常现象发生时，这个异常现象没有任何原因。

继续看一下JVM是如何实现park方法：park在不同的操作系统中使用不同的方式实现，在Linux下使用的是系统方法pthread\_cond\_wait实现。实现代码在JVM源码路径

src/os/linux/vm/os\_linux.cpp里的os::PlatformEvent::park方法，代码如下。

|  |
| --- |
| void os::PlatformEvent::park() {  int v ;  for (;;) {  v = \_Event ;  if (Atomic::cmpxchg (v-1, &\_Event, v) == v) break ;  }  guarantee (v >= 0, "invariant") ;  if (v == 0) {  // Do this the hard way by blocking ...  int status = pthread\_mutex\_lock(\_mutex);  assert\_status(status == 0, status, "mutex\_lock");  guarantee (\_nParked == 0, "invariant") ;  ++ \_nParked ;  while (\_Event < 0) {  status = pthread\_cond\_wait(\_cond, \_mutex);  // for some reason, under 2.7 lwp\_cond\_wait() may return ETIME ...  // Treat this the same as if the wait was interrupted  if (status == ETIME) { status = EINTR; }  assert\_status(status == 0 || status == EINTR, status, "cond\_wait");  }  -- \_nParked ;  // In theory we could move the ST of 0 into \_Event past the unlock(),  // but then we'd need a MEMBAR after the ST.  \_Event = 0 ;  status = pthread\_mutex\_unlock(\_mutex);  assert\_status(status == 0, status, "mutex\_unlock");  }  guarantee (\_Event >= 0, "invariant") ;  }  } |

pthread\_cond\_wait是一个多线程的条件变量函数，cond是condition的缩写，字面意思可以

理解为线程在等待一个条件发生，这个条件是一个全局变量。

这个方法接收两个参数：一个共享变量\_cond，一个互斥量\_mutex。而unpark方法在Linux下是使用pthread\_cond\_signal实现的。park方法在Windows下则是使用WaitForSingleObject实现的。想知道pthread\_cond\_wait是如何实现的，可以参考glibc-2.5的nptl/sysdeps/pthread/pthread\_cond\_wait.c。

当线程被阻塞队列阻塞时，线程会进入WAITING（parking）状态。可使用jstack dump

阻塞的生产者线程看到这点，如下。

|  |
| --- |
| "main" prio=5 tid=0x00007fc83c000000 nid=0x10164e000 waiting on condition [0x000000010164d000]  java.lang.Thread.State: WAITING (parking)  at sun.misc.Unsafe.park(Native Method)  - parking to wait for <0x0000000140559fe8> (a java.util.concurrent.locks.  AbstractQueuedSynchronizer$ConditionObject)  at java.util.concurrent.locks.LockSupport.park(LockSupport.java:186)  at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.  await(AbstractQueuedSynchronizer.java:2043)  at java.util.concurrent.ArrayBlockingQueue.put(ArrayBlockingQueue.java:324)  at blockingqueue.ArrayBlockingQueueTest.main(ArrayBlockingQueueTest.java: |

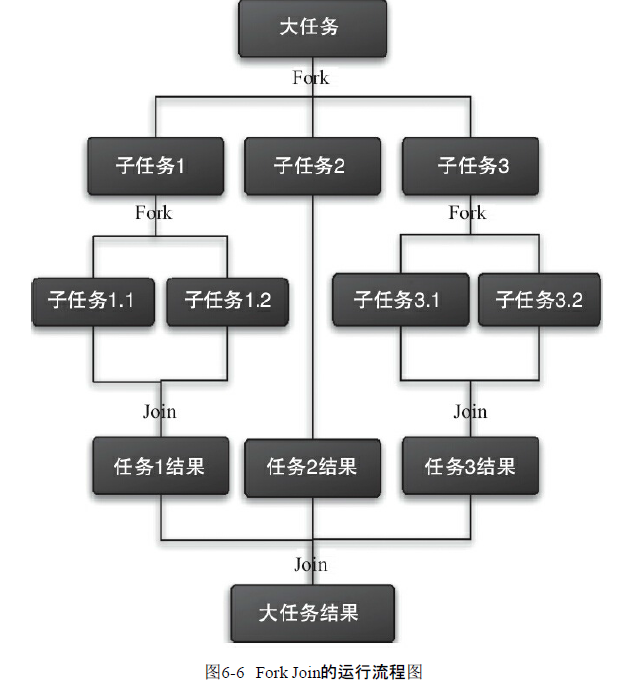
**6.4　Fork/Join框架**

基本原理、算法、设计方式、应用与实现等。

**6.4.1　什么是Fork/Join框架**

Java 7 并行执行任务的框架，把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

Fork就是把一个大任务切分为若干子任务并行的执行，Join就是合并这些子任务的执行结果，最后得到这个大任务的结果。比如计算1+2+…+10000，可以分割成10个子任务，每个子任务分别对1000个数进行求和，最终汇总这10个子任务的结果。Fork/Join的运行流程如图6-6所示。



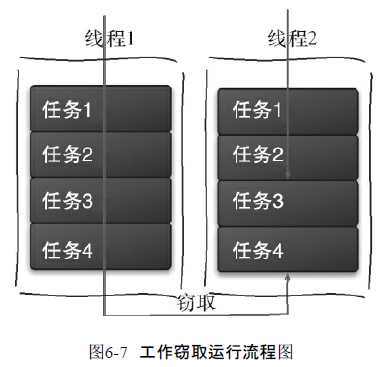
**6.4.2　工作窃取算法**

指某个线程从其他队列里窃取任务来执行。

假如需做一个较大任务，可把这个任务分割为若干互不依赖的子任务，为了减少线程间的竞争，把这些子任务分别放到不同的队列里，并为每个队列创建一个单独的线程来执行队列里的任务，线程和队列一一对应。

比如A线程负责处理A队列里的任务。但是，有的线程会先把自己队列里的任务干完，而其他线程对应的队列里还有任务等待处理。干完活的线程与其等着，不如去帮其他线程干活，于是它就去其他线程的队列里窃取一个任务来执行。而在这时它们会访问同一个队列，所以为了减少窃取任务线程和被窃取任务线程间的竞争，通常会使用双端队列，被窃取任务线程永远从双端队列的头部拿任务执行，而窃取任务的线程永远从双端队列的尾部拿任务执行。

工作窃取的运行流程如图6-7所示。



优点：充分利用线程进行并行计算，减少了线程间的竞争。

缺点：在某些情况下还是存在竞争，比如双端队列里只有一个任务时。并

且该算法会消耗了更多的系统资源，比如创建多个线程和多个双端队列。

**6.4.3　Fork/Join框架的设计**

步骤1　分割任务。首先需有一个fork类来把大任务分割成子任务，有可能子任务还

是很大，所以还需要不停地分割，直到分割出的子任务足够小。

步骤2　执行任务并合并结果。分割的子任务分别放在双端队列里，然后几个启动线程分

别从双端队列里获取任务执行。子任务执行完的结果都统一放在一个队列里，启动一个线程

从队列里拿数据，然后合并这些数据。

Fork/Join使用两个类来完成以上两件事情。

①ForkJoinTask：必须首先创建一个ForkJoin任务。提供在任务中执行fork()和join()操作的机制。通常不需要直接继承ForkJoinTask类，只需要继承它的子类，Fork/Join框架提供了以下两个子类。

·RecursiveAction：用于没有返回结果的任务。

·RecursiveTask：用于有返回结果的任务。

②ForkJoinPool：ForkJoinTask需要通过ForkJoinPool来执行。

任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列头部。当一个工作线程的队列里暂时没任务时，会随机从其他工作线程的队列的尾部获取一个任务。

**6.4.4　使用Fork/Join框架**

需求是：计算1+2+3+4的结果。

使用Fork/Join框架首先要考虑到的是如何分割任务，如果希望每个子任务最多执行两个

数的相加，那么我们设置分割的阈值是2，由于是4个数字相加，所以Fork/Join框架会把这个任务fork成两个子任务，子任务一负责计算1+2，子任务二负责计算3+4，然后再join两个子任务的结果。因为是有结果的任务，所以必须继承RecursiveTask，实现代码如下。

|  |
| --- |
| package fj;  import java.util.concurrent.ExecutionException;  import java.util.concurrent.ForkJoinPool;  import java.util.concurrent.Future;  import java.util.concurrent.RecursiveTask;  public class CountTask extends RecursiveTask<Integer> {  private static final int THRESHOLD = 2;　　// 阈值  private int start;  private int end;  public CountTask(int start, int end) {  this.start = start;  this.end = end;  }  @Override  protected Integer compute() {  int sum = 0;  // 如果任务足够小就计算任务  boolean canCompute = (end - start) <= THRESHOLD;  if (canCompute) {  for (int i = start; i <= end; i++) {  sum += i;  }  } else {  // 如果任务大于阈值，就分裂成两个子任务计算  int middle = (start + end) / 2;  CountTask leftTask = new CountTask(start, middle);  CountTask rightTask = new CountTask(middle + 1, end);  // 执行子任务  leftTask.fork();  rightTask.fork();  // 等待子任务执行完，并得到其结果  int leftResult=leftTask.join();  int rightResult=rightTask.join();  // 合并子任务  sum = leftResult + rightResult;  }  return sum;  }  public static void main(String[] args) {  ForkJoinPool forkJoinPool = new ForkJoinPool();  // 生成一个计算任务，负责计算1+2+3+4  CountTask task = new CountTask(1, 4);  // 执行一个任务  Future<Integer> result = forkJoinPool.submit(task);  try {  System.out.println(result.get());  } catch (InterruptedException e) {  } catch (ExecutionException e) {  }  }  } |

ForkJoinTask与一般任务的主要区别在于需实现compute方法

首先需判断任务是否足够小，如果足够小就直接执行任务。

如不足够小，必须分割成两个子任务，每个子任务在调用fork方法时，又会进入compute方法，看看当前子任务是否需要继续分割成子任务，如果不需要继续分割，则执行当前子任务并返回结果。

使用join方法会等待子任务执行完并得到其结果。

**6.4.5　Fork/Join框架的异常处理**

ForkJoinTask执行时可能抛出异常，但没办法在主线程里直接捕获异常，

所以ForkJoinTask提供了isCompletedAbnormally()方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过ForkJoinTask的getException方法获取异常。使用如下代码。

|  |
| --- |
| if(task.isCompletedAbnormally())  {  System.out.println(task.getException());  } |

getException方法返回Throwable对象

如果任务被取消了则返回CancellationException。

如任务没完成或没有抛出异常则返回null。

**6.4.6　Fork/Join框架的实现原理**

ForkJoinPool由ForkJoinTask数组和ForkJoinWorkerThread数组组成，ForkJoinTask数组负责将存放程序提交给ForkJoinPool的任务，而ForkJoinWorkerThread数组负责执行这些任务。

（1）ForkJoinTask的fork方法实现原理

当调用ForkJoinTask的fork方法时，程序会调用ForkJoinWorkerThread的pushTask方法异步地执行这个任务，然后立即返回结果。代码如下。

|  |
| --- |
| public final ForkJoinTask<V> fork() {  ((ForkJoinWorkerThread) Thread.currentThread())  .pushTask(this);  return this;  } |

pushTask:把当前任务存放在ForkJoinTask数组队列里。然后再调用ForkJoinPool的

signalWork()方法唤醒或创建一个工作线程来执行任务。代码如下。

|  |
| --- |
| final void pushTask(ForkJoinTask<> t) {  ForkJoinTask<>[] q; int s, m;  if ((q = queue) != null) {　　　　// ignore if queue removed  long u = (((s = queueTop) & (m = q.length - 1)) << ASHIFT) + ABASE;  UNSAFE.putOrderedObject(q, u, t);  queueTop = s + 1;　　　　　　// or use putOrderedInt  if ((s -= queueBase) <= 2)  pool.signalWork();  else if (s == m)  growQueue();  }  } |

（2）ForkJoinTask的join方法实现原理

Join()的主要作用是阻塞当前线程并等待获取结果。

代码如下。

|  |
| --- |
| public final V join() {  if (doJoin() != NORMAL)  return reportResult();  else  return getRawResult();  }  private V reportResult() {  int s; Throwable ex;  if ((s = status) == CANCELLED)  throw new CancellationException();  if (s == EXCEPTIONAL && (ex = getThrowableException()) != null)  UNSAFE.throwException(ex);  return getRawResult();  } |

首先调用了doJoin()，得到当前任务的状态来判断返回什么结果，

任务状态：已完成（NORMAL）、被取消（CANCELLED）、信号（SIGNAL）和出现异常

（EXCEPTIONAL）。

·如果任务状态是已完成，则直接返回任务结果。

·如果任务状态是被取消，则直接抛出CancellationException。

·如果任务状态是抛出异常，则直接抛出对应的异常。

让我们再来分析一下doJoin()方法的实现代码。

|  |
| --- |
| private int doJoin() {  Thread t; ForkJoinWorkerThread w; int s; boolean completed;  if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread) {  if ((s = status) < 0)  return s;  if ((w = (ForkJoinWorkerThread)t).unpushTask(this)) {  try {  completed = exec();  } catch (Throwable rex) {  return setExceptionalCompletion(rex);  }  if (completed)  return setCompletion(NORMAL);  }  return w.joinTask(this);  }else  return externalAwaitDone();  } |

doJoin()里，查看任务的状态，看任务是否已经执行完，如执行完，则直接返回任务状态；

如没执行完，则从任务数组里取出任务并执行。

如果任务顺利执行完成，则设置任务状态为NORMAL，如果出现异常，则记录异常，并将任务状态设置为EXCEPTIONAL。

6.5　本章小结

本章介绍了Java中提供的各种并发容器和框架，并分析了该容器和框架的实现原理，从中

能够领略到大师级的设计思路，希望读者能够充分理解这种设计思想，并在以后开发的

并发程序时，运用上这些并发编程的技巧。

**第9章 java中线程池**

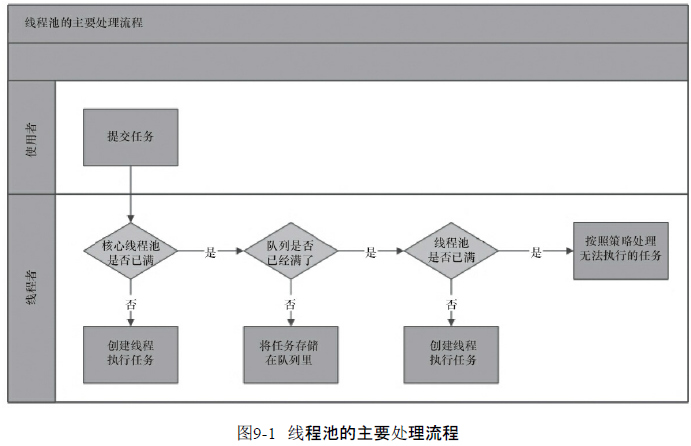
Java中的线程池是运用场景最多的并发框架，几乎所有需异步或并发执行任务的程序

都可使用线程池。好处：

|  |
| --- |
| 第一：降低资源消耗。通过重复用已创建的线程降低线程创建和销毁造成的消耗。  第二：提高响应速度。当任务到达时，任务可不需要等到线程创建就能立即执行。  第三：提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，  还会降低系统的稳定性，使用线程池可进行统一分配、调优和监控。 |

**9.1　线程池的实现原理**

9.1.1 线程池的处理流程



当提交一个新任务到线程池，线程池的处理流程如下:

1. 核心线程池里的线程是否都在执行任务。

如不是，则创建新的线程执行任务。

如是，则进入下个流程。

1. 工作队列是否已满。

如没满，则将任务存储在工作队列里。

如满了，则进入下个流程。

1. 线程池的线程是否都处于工作状态。

如没有，则创建新线程来执行任务。

如已满，则交给饱和策略来处理。

**9.1.2 ThreadPoolExecutor执行execute()方法示意图**

* 4种情况:

|  |
| --- |
| 1）如当前运行的线程少于corePoolSize，则创建新线程来执行任务（需要获取全局锁）。  2）如运行的线程等于或多于corePoolSize，则将任务加入BlockingQueue。  3）如无法将任务加入BlockingQueue（队列已满），则创建新线程处理任务（需获取全局锁）。  4）如创建新线程将使当前运行的线程超出maximumPoolSize，任务将被拒绝，并调用  RejectedExecutionHandler.rejectedExecution()方法。 |

是为了在执行execute()时，尽可能地避免获取全局锁（那将会是一个严重的可伸缩瓶颈）。在ThreadPoolExecutor完成预热后（当前运行的线程数大于等于corePoolSize），几乎所有的execute()方法调用都是执行步骤2，而步骤2不需要获取全局锁。

* 源码分析：上面的流程分析让我们很直观地了解了线程池的工作原理，再通过源代码来看看如何实现的，线程池执行任务的方法如下。

|  |
| --- |
| public void execute(Runnable command) {  if (command == null)  throw new NullPointerException();  // 如果线程数小于基本线程数，则创建线程并执行当前任务  if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {  // 如线程数大于等于基本线程数或线程创建失败，则将当前任务放到工作队列中。  if (runState == RUNNING && workQueue.offer(command)) {  if (runState != RUNNING || poolSize == 0)  ensureQueuedTaskHandled(command);  } // 如果线程池不处于运行中或任务无法放入队列，  //并且当前线程数量小于最大允许的线程数量，则创建一个线程执行任务。  else if (!addIfUnderMaximumPoolSize(command))  // 抛出RejectedExecutionException异常  reject(command); // is shutdown or saturated  }  } |

* **工作线程**：线程池创建线程时，会将线程封装成工作线程Worker，Worker在执行完任务后，还会循环获取工作队列里的任务来执行。可从Worker类的run()方法里看到这点。

|  |
| --- |
| public void run() {  try {  Runnable task = firstTask;  firstTask = null;  while (task != null || (task = getTask()) != null) {  runTask(task);  task = null;  }  } finally {  workerDone(this);  }  } |

ThreadPoolExecutor中线程执行任务的示意图如图9-3所示



线程池中的线程执行任务分两种情况:

|  |
| --- |
| 1）在execute()方法中创建一个线程时，会让这个线程执行当前任务。  2）这个线程执行完上图中1的任务后，会反复从BlockingQueue获取任务来执行。 |

**9.2　线程池的使用**

**9.2.1　线程池的创建**

|  |
| --- |
| new ThreadPoolExecutor(corePoolSize, maximumPoolSize, keepAliveTime,  milliseconds, runnableTaskQueue, handler); |

参数:

|  |  |  |
| --- | --- | --- |
| 1. corePoolSize（线程池的基本大小）：   当提交一个任务到线程池，线程池会创建一个线程来执行任务，即使其他空闲的基本线程能够执行新任务也会创建线程，等到需执行的任务数大于线程池基本大小时就不再创建。如果调用了线程池的prestartAllCoreThreads()方法，线程池会提前创建并启动所有基本线程。   1. maximumPoolSize（线程池最大数量）：   线程池允许创建的最大线程数。如果队列满了，并且已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。值得注意的是，如果使用了无界的任务队列这个参数就没什么效果。  3) keepAliveTime（线程活动保持时间）：  线程池的工作线程空闲后，保持存活的时间。所以，如果任务很多，且每个任务执行时间较短，可调大时间，提高线程的利用率。  4)TimeUnit（线程活动保持时间的单位）：  可选的单位有天（DAYS）、小时（HOURS）、分钟（MINUTES）、毫秒（MILLISECONDS）、微秒（MICROSECONDS，千分之一毫秒）和纳秒（NANOSECONDS，千分之一微秒）。  3）ThreadFactory：用于设置创建线程的工厂，可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。使用开源框架guava提供的ThreadFactoryBuilder可以快速给线程池里的线程设置有意义的名字，代码如下。  new ThreadFactoryBuilder().setNameFormat("XX-task-%d").build();   1. runnableTaskQueue（任务队列）：   用于保存等待执行的任务的阻塞队列。可以选择以下几个阻塞队列。   |  | | --- | | ·ArrayBlockingQueue：是一个基于数组结构的有界阻塞队列，此队列按FIFO（先进先出）原则对元素进行排序。  ·LinkedBlockingQueue：一个基于链表结构的阻塞队列，此队列按FIFO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一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于Linked-BlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。  ·PriorityBlockingQueue：一个具有优先级的无限阻塞队列。 |   5）RejectedExecutionHandler（饱和策略）：当队列和线程池都满了，说明线程池处于饱和状  态，那么必须采取一种策略处理提交的新任务。这个策略默认情况下是AbortPolicy，表示无法处理新任务时抛出异常。   |  | | --- | | 在JDK 1.5中Java线程池框架提供了以下4种策略。  ·AbortPolicy：直接抛出异常。  ·CallerRunsPolicy：只用调用者所在线程来运行任务。  ·DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。  ·DiscardPolicy：不处理，丢弃掉。  当然，也可根据应用场景需要来实现RejectedExecutionHandler接口自定义策略。如记录日志或持久化存储不能处理的任务。 | |

**9.2.2　向线程池提交任务**

* execute()用于提交不需返回值的任务，所以无法判断任务是否被线程池执行成功。

输入的任务是Runnable类的实例。

|  |
| --- |
| threadsPool.execute(new Runnable() {  @Override  public void run() {  }  }); |

* submit()用于提交需返回值的任务。

线程池会返回一个future类型的对象，可判断任务是否执行成功

* 且可通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，
* 而使用get(long timeout，TimeUnit unit)方法则会阻塞当前线程一段时间后立即返回，这时有可能任务没有执行完。

|  |
| --- |
| Future<Object> future = executor.submit(harReturn Valuetask);  try {  Object s = future.get();  } catch (InterruptedException e) {  // 处理中断异常  } catch (ExecutionException e) {  // 处理无法执行任务异常  } finally {  // 关闭线程池  executor.shutdown();  } |

**9.2.3　关闭线程池**

可通过调用线程池的shutdown或shutdownNow方法来关闭线程池。

原理是遍历线程池中的工作线程，然后逐个调用线程的interrupt方法来中断线程，所以无法响应中断的任务可能永远无法终止。

区别

* shutdownNow首先将线程池的状态设置成STOP，然后尝试停止所有的正在执行或暂停任务的线程，并返回等待执行任务的列表
* shutdown只是将线程池的状态设置成SHUTDOWN状态，然后中断所有没在执行任务的线程。

只要调用了这两个关闭方法中的任意一个，isShutdown方法就会返回true。

当所有任务都已关闭后，才表示线程池关闭成功，这时调用isTerminaed方法会返回true。通常调用shutdown来关闭线程池，如任务不一定要执行完，则可以调用shutdownNow方法。

**9.2.4　合理地配置线程池**

必须首先分析任务特性，可从以下几个角度来分析。

·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、中和低。

·任务的执行时间：长、中和短。

·任务的依赖性：是否依赖其他系统资源，如数据库连接。

性质不同的任务可用不同规模的线程池分开处理。

CPU密集型任务应配置尽可能小的线程，如配置Ncpu+1个线程的线程池。

IO密集型任务线程并不是一直在执行任务，则应配置尽可能多的线程，如2\*Ncpu。

混合型任务，如可拆分，将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务执行的时间相差不是太大，分解后执行的吞吐量将高于串行执行的吞吐量。如这两个任务执行时间相差太大，则没必要进行分解。可通过Runtime.getRuntime().availableProcessors()方法获得当前设备的CPU个数。

优先级不同的任务可使用优先级队列PriorityBlockingQueue来处理。可让优先级高的任务先执行。注意如一直有优先级高的任务提交到队列里，那么优先级低的任务可能永远不能执行。

执行时间不同的任务可交给不同规模的线程池来处理，或可以使用优先级队列，让

执行时间短的任务先执行。

依赖数据库连接池的任务，因线程提交SQL后需要等待数据库返回结果，等待的时间越长，则CPU空闲时间就越长，那么线程数应该设置得越大，这样才能更好地利用CPU。

建议使用有界队列。有界队列能增加系统的稳定性和预警能力，可以根据需要设大一点

儿，比如几千。有一次，我们系统里后台任务线程池的队列和线程池全满了，不断抛出抛弃任务的异常，通过排查发现是数据库出现了问题，导致执行SQL变得非常缓慢，因为后台任务线程池里的任务全是需要向数据库查询和插入数据的，所以导致线程池里的工作线程全部阻塞，任务积压在线程池里。如果当时我们设置成无界队列，那么线程池的队列就会越来越多，有可能会撑满内存，导致整个系统不可用，而不只是后台任务出现问题。当然，我们的系统所有的任务是用单独的服务器部署的，我们使用不同规模的线程池完成不同类型的任务，但是出现这样问题时也会影响到其他任务。

**9.2.5　线程池的监控**

如在系统中大量使用线程池，则有必要对线程池进行监控，方便在出现问题时，可根据线程池的使用状况快速定位问题。可通过线程池提供的参数进行监控，监控线程池时可使用以下属性。

·taskCount：线程池需执行的任务数量。

·completedTaskCount：线程池在运行过程中已完成的任务数量，小于或等于taskCount。

·largestPoolSize：线程池里曾创建过的最大线程数量。通过这个数据可知道线程池是

否曾满过。如该数值等于线程池的最大大小，则表示线程池曾经满过。

·getPoolSize：线程池的线程数量。如果线程池不销毁的话，线程池里的线程不会自动销

毁，所以这个大小只增不减。

·getActiveCount：获取活动的线程数。

通过扩展线程池进行监控。可通过继承线程池来自定义线程池，重写线程池的

beforeExecute、afterExecute和terminated方法，也可在任务执行前、执行后和线程池关闭前执行一些代码来进行监控。例如，监控任务的平均执行时间、最大执行时间和最小执行时间等。这几个方法在线程池里是空方法。

protected void beforeExecute(Thread t, Runnable r) { }

**9.3　本章小结**

本章介绍了为什么要使用线程池、如何使用线程池和线程池的使用原理

**第10章　Executor框架**

线程的创建与销毁需一定开销，如为每一个任务创建一个新线程来执行，这些线程的创建与销毁将消耗大量的计算资源。同时可能会使处于高负荷状态的应用最终崩溃。

Java的线程既是工作单元，也是执行机制。

从JDK 5开始，把工作单元(Runnable和Callable)与执行机制(Executor框架提供)分离开来。

**10.1　Executor框架简介**

**10.1.1　Executor框架的两级调度模型**

HotSpot VM线程模型中，Java线程（java.lang.Thread）被一对一映射为本地操作系统线程。

Java线程启动时会创建一个本地操作系统线程；当该Java线程终止时，这个操作系统线程也会被回收。操作系统会调度所有线程并将它们分配给可用的CPU。

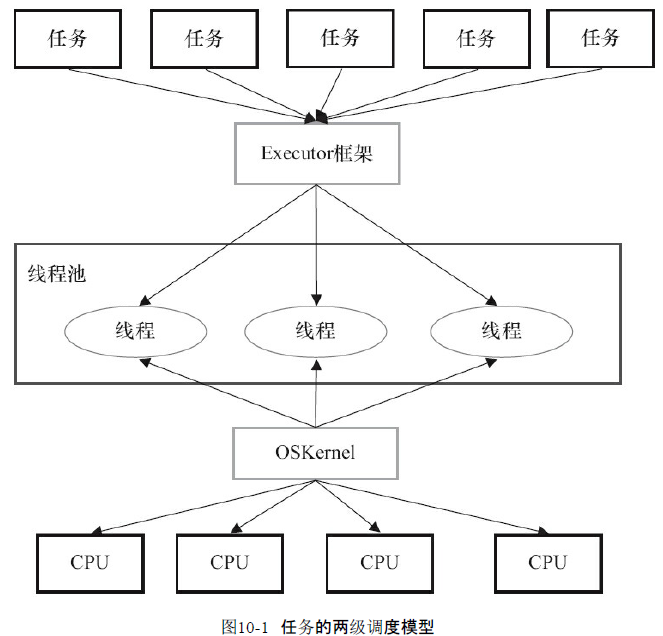
在上层，Java多线程程序常把应用分解为若干个任务，使用用户级的调度器（Executor框架）将这些任务映射为固定数量的线程；

在底层，操作系统内核将这些线程映射到硬件处理器上。

两级调度模型的示意图如图10-1所示:应用程序通过Executor框架控制上层的调度；下层调度由操作系统内核控制，不受应用程序的控制。

**10.1.2　Executor框架的结构与成员**

本文将分两部分来介绍Executor：Executor的结构和Executor框架包含的成员组件。

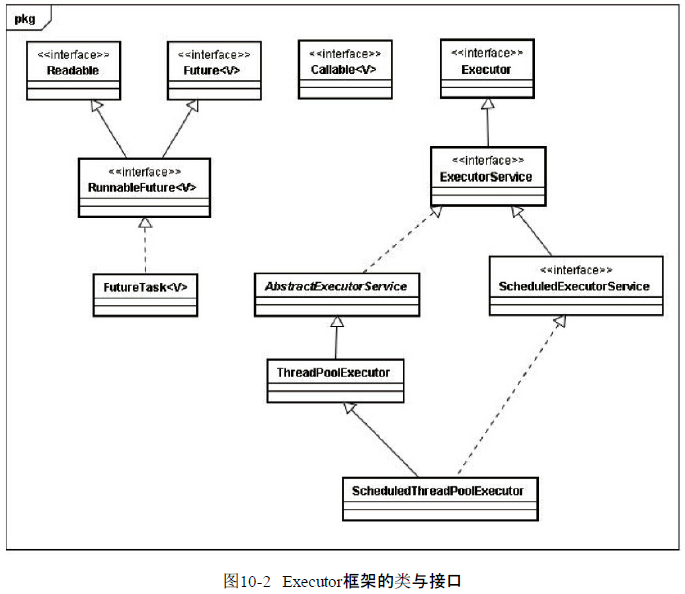


1.Executor框架的结构

* 由3大部分组成:

|  |
| --- |
| ·**任务**。包括被执行任务需实现的接口：Runnable接口或Callable接口。  ·任务的执行:任务执行机制的核心接口Executor,继承自Executor的ExecutorService接口。实现了ExecutorService接口的两个关键类  (ThreadPoolExecutor和ScheduledThreadPoolExecutor）。  ·异步计算的结果。包括接口Future和实现Future接口的FutureTask类。 |

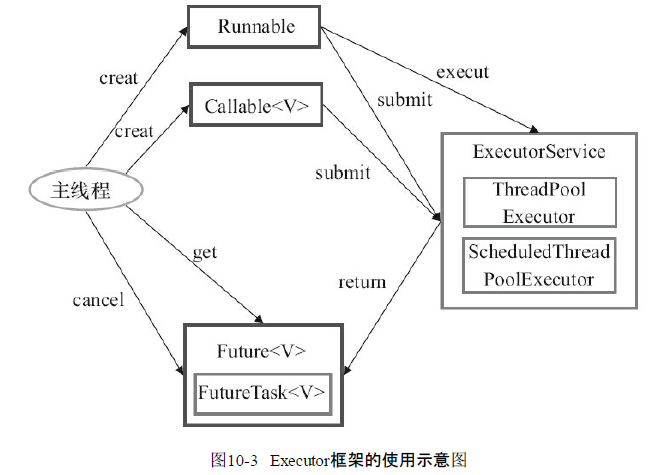
* Executor框架包含的主要的类与接口如图10-2所示。



* 这些类和接口的简介。

|  |
| --- |
| ·Executor是一个接口，是Executor框架的基础，将任务的提交与执行分离开。  ·ThreadPoolExecutor是线程池的核心实现类，用来执行被提交的任务。  ·ScheduledThreadPoolExecutor是一个实现类，可在给定的延迟后运行命令，或定期执  行命令。比Timer更灵活，功能更强大。  ·Future接口和实现Future接口的FutureTask类，代表异步计算的结果。  ·Runnable接口和Callable接口的实现类，都可被ThreadPoolExecutor或Scheduled-  ThreadPoolExecutor执行。 |

* Executor框架的使用示意图如图10-3所示。



* 主线程先要创建实现Runnable或Callable接口的任务对象。
* 工具类Executors可把Runnable对象封装为Callable对象

|  |
| --- |
| (Executors.callable（Runnable task）或Executors.callable（Runnable task，Object resule））。 |

* 然后可把Runnable对象直接交给ExecutorService执行

|  |
| --- |
| (ExecutorService.execute(Runnable command))； |

* 或也可把Runnable对象或Callable对象提交给ExecutorService执行

|  |
| --- |
| (Executor-Service.submit(Runnable task）或ExecutorService.submit(Callable<T>task))。 |

如执行ExecutorService.submit（…），ExecutorService将返回一个实现Future接口的对象

|  |
| --- |
| Future<?> submit(Runnable task); |

(到目前为止的JDK中，返回的是FutureTask对象）。

由于FutureTask实现了Runnable，也可创建FutureTask，然后直接交给ExecutorService执行。

|  |
| --- |
| public class FutureTask<V> implements RunnableFuture<V>  public interface RunnableFuture<V> extends Runnable, Future<V> |

* 最后，主线程可执行FutureTask.get()方法来等待任务执行完成。

主线程也可执行FutureTask.cancel（boolean mayInterruptIfRunning）来取消此任务的执行。

2.Executor框架的成员

ThreadPoolExecutor、ScheduledThreadPoolExecutor、Future接口、Runnable接口、Callable接口和Executors。

(1)ThreadPoolExecutor

常用工厂类Executors(java.util.concurrent. Executors)创建。

可创建3种类型的ThreadPoolExecutor：

|  |
| --- |
| SingleThreadExecutor、FixedThreadPool和CachedThreadPool。 |

1. FixedThreadPool。

下面是Executors提供的，创建使用固定线程数的FixedThreadPool的API。

|  |
| --- |
| public static ExecutorService newFixedThreadPool(int nThreads)  public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactorythreadFactory) |

适用于为满足资源管理的需求，而需限制当前线程数量的场景，适用于负载较重的服务器。

2）SingleThreadExecutor。

|  |
| --- |
| public static ExecutorService newSingleThreadExecutor()  public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory) |

适用于需保证顺序执行各个任务；且在任意时间点，不会有多个线程是活动的应用场景。

3）CachedThreadPool。

|  |
| --- |
| public static ExecutorService newCachedThreadPool()  public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) |

是大小无界的线程池，适用于执行很多的短期异步任务的小程序，或是负载较轻的服务器。

(2)ScheduledThreadPoolExecutor

通常用工厂类Executors创建。2种类型,如下:

|  |
| --- |
| ·ScheduledThreadPoolExecutor。包含若干个线程的ScheduledThreadPoolExecutor。  ·SingleThreadScheduledExecutor。只包含一个线程的ScheduledThreadPoolExecutor。 |

* ScheduledThreadPoolExecutor的API。

|  |
| --- |
| public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)  public static ScheduledExecutorService newScheduledThreadPool(intcorePoolSize,ThreadFactory |

适用于需多个后台线程执行周期任务，同时为满足资源管理的需求而需要限制后台线程数量的场景。

* SingleThreadScheduledExecutor

|  |
| --- |
| public static ScheduledExecutorService newSingleThreadScheduledExecutor()  public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory) |

适用于需单个后台线程执行周期任务，同时需要保证顺序地执行各个任务的应用场景。

(3)Future接口

Future接口和实现Future接口的FutureTask类用来表异步计算的结果。

当把Runnable接口或Callable接口的实现类提交（submit）给ThreadPoolExecutor或

ScheduledThreadPoolExecutor时，ThreadPoolExecutor或ScheduledThreadPoolExecutor会向我们返回一个FutureTask对象。下面是对应的API。

|  |
| --- |
| <T> Future<T> submit(Callable<T> task)  <T> Future<T> submit(Runnable task, T result) Future<> submit(Runnable task) |

到JDK 8为止，Java通过上述API返回的是一个FutureTask对象。

将来的JDK实现中，返回的可能不一定是FutureTask。

(4)Runnable接口和Callable接口

两者的实现类，都可被ThreadPoolExecutor或Scheduled-ThreadPoolExecutor执行。

区别是Runnable不会返回结果，而Callable可返回结果。

除了可自己创建实现Callable接口的对象外，还可用工厂类Executors来把一个

* Runnable包装成Callable。

|  |
| --- |
| public static Callable<Object> callable(Runnable task) // 假设返回对象Callable1 |

下面是Executors提供的，把一个Runnable和一个待返回的结果包装成一个Callable的API。

|  |
| --- |
| public static <T> Callable<T> callable(Runnable task, T result) // 假设返回对象Callable2 |

前面讲过，把Callable对象提交给ThreadPoolExecutor或ScheduledThreadPoolExecutor执行时，submit（…）会返回FutureTask对象。

可执行FutureTask.get()来等待任务执行完。当任务完成后FutureTask.get()将返回该任务结果。

|  |
| --- |
| 例如，如提交的是对象Callable1，FutureTask.get()方法将返回null；  如果提交的是对象Callable2，FutureTask.get()方法将返回result对象。 |

**10.2　ThreadPoolExecutor详解**

* Executor框架最核心的类是ThreadPoolExecutor，是线程池的实现类，4个组件构成。

|  |
| --- |
| ·corePool：核心线程池的大小。  ·maximumPool：最大线程池的大小。  ·BlockingQueue：用来暂时保存任务的工作队列。  ·RejectedExecutionHandler：当ThreadPoolExecutor已经关闭或已饱和时（达到了最大线程池大小且工作队列已满），execute()方法将要调用的Handler。 |

* 通过Executor框架的工具类Executors，可创建3种类型的ThreadPoolExecutor。

|  |
| --- |
| ·FixedThreadPool。  ·SingleThreadExecutor。  ·CachedThreadPool。 |

**10.2.1　FixedThreadPool详解**

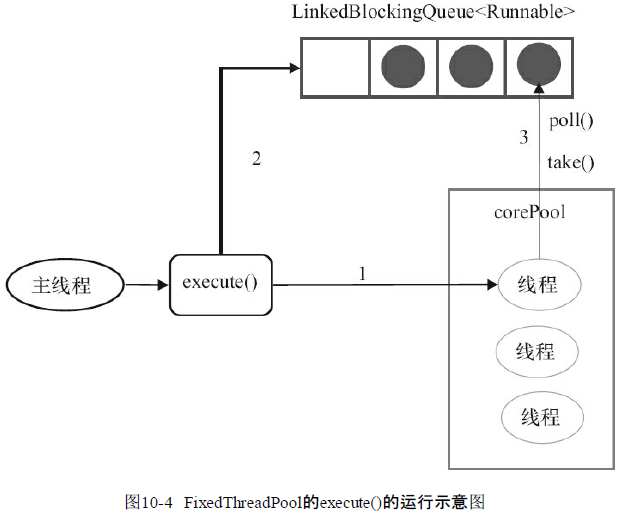
* 可重用固定线程数的线程池。源代码实现。

|  |
| --- |
| public static ExecutorService newFixedThreadPool(int nThreads) {  return new ThreadPoolExecutor(nThreads, nThreads,0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>());  } |

当线程池中的线程数大于corePoolSize时，keepAliveTime为多余空闲线程等待新任务的最长时间，超过这时间后多余线程将被终止。这里把keepAliveTime设置为0L，意味着多余

的空闲线程会被立即终止。

* 其execute()方法的运行示意图：



说明：

1.如当前运行的线程数少于corePoolSize，则创建新线程来执行任务。

2.在线程池完成预热之后（当前运行的线程数等于corePoolSize），将任务加入

LinkedBlockingQueue。

3.线程执行完1中的任务后，会在循环中反复从LinkedBlockingQueue获取任务来执行。

FixedThreadPool用无界队列LinkedBlockingQueue作为线程池的工作队列（队列的容量为

Integer.MAX\_VALUE）。使用无界队列作为工作队列会对线程池带来如下影响：

|  |
| --- |
| 1）当线程池中的线程数达到corePoolSize后，新任务将在无界队列中等待，因此线程池中  的线程数不会超过corePoolSize。  2）由于1，使用无界队列时maximumPoolSize将是一个无效参数。  3）由于1和2，使用无界队列时keepAliveTime将是一个无效参数。  4）由于使用无界队列，运行中的FixedThreadPool（未执行方法shutdown()或  shutdownNow()）不会拒绝任务（不会调用RejectedExecutionHandler.rejectedExecution方法）。 |

**10.2.2　SingleThreadExecutor详解**

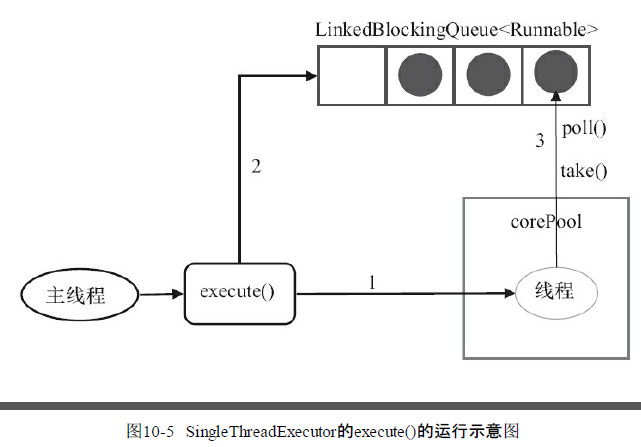
使用单个worker线程的Executor。源代码实现。

|  |
| --- |
| public static ExecutorService newSingleThreadExecutor() {  return new FinalizableDelegatedExecutorService  (new ThreadPoolExecutor(1, 1,0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>()));  } |

corePoolSize和maximumPoolSize被设置为1。其他参数与FixedThreadPool相同。

使用无界队列LinkedBlockingQueue作为线程池的工作队列（容量为Integer.MAX\_VALUE）。对线程池带来的影响与FixedThreadPool相同，不赘述。

SingleThreadExecutor的运行示意图如图10-5所示。



对图10-5的说明如下:

|  |
| --- |
| 1）如当前运行的线程数少于corePoolSize（即线程池中无运行的线程），则创建一新线  程来执行任务。  2）在线程池完成预热后（当前线程池中有一个运行的线程），将任务加入Linked-  BlockingQueue。  3）线程执行完1中的任务后，会在一个无限循环中反复从LinkedBlockingQueue获取任务来执行。 |

**10.2.3　CachedThreadPool详解**

会根据需要创建新线程的线程池。源代码。

|  |
| --- |
| public static ExecutorService newCachedThreadPool() {  return new ThreadPoolExecutor(0, Integer.MAX\_VALUE, 60L, TimeUnit.SECONDS,  new SynchronousQueue<Runnable>());  } |

corePoolSize被设置为0，即corePool为空；

maximumPoolSize被设置为Integer.MAX\_VALUE(0x7fffffff)，即maximumPool无界。keepAliveTime设置为60L，意味着CachedThreadPool中的空闲线程等待新任务的最长时间为60秒，空闲线程超过60秒后将会被终止。

使用没有容量的SynchronousQueue作为线程池的工作队列，但CachedThreadPool的maximumPool是无界的。这意味着，如果主线程提交任务的速度高于maximumPool中线程处理任务的速度时，CachedThreadPool会不断创建新线程。极端情况下，CachedThreadPool会因为创建过多线程而耗尽CPU和内存资源。

CachedThreadPool的execute()方法的执行示意图如图10-6所示。



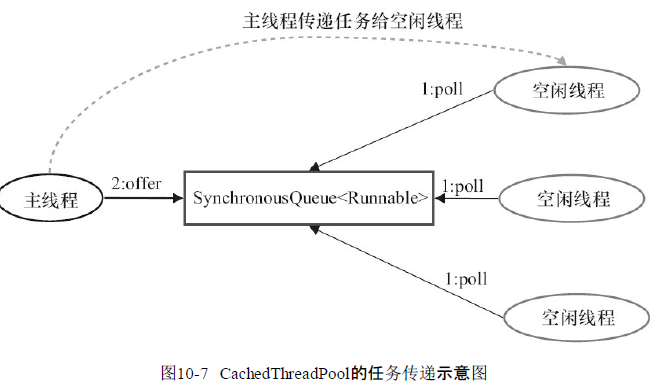
说明:

1. 首先执行SynchronousQueue.offer(Runnable task)。

如当前maximumPool中有空闲线程正在执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS），主线程执行offer操作与空闲线程执行的poll操作配对成功，主线程把任务交给空闲线程执行，execute()方法执行完成；否则执行下面的步骤2）。

2）当初始maximumPool为空，或maximumPool中当前没空闲线程时，将没有线程执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）。这种情况下，步骤1）将失败。此时CachedThreadPool会创建一个新线程执行任务，execute()方法执行完成。

3）在步骤2）中新创建的线程将任务执行完后，会执行SynchronousQueue.poll（keepAliveTime，TimeUnit.NANOSECONDS）。这个poll操作会让空闲线程最多在SynchronousQueue中等待60秒钟。如果60秒钟内主线程提交了一个新任务（主线程执行步骤1）），那么这个空闲线程将执行主线程提交的新任务；否则，这个空闲线程将终止。由于空闲60秒的空闲线程会被终止，因此长时间保持空闲的CachedThreadPool不会使用任何资源。前面提到过，SynchronousQueue是一个没有容量的阻塞队列。每个插入操作必须等待另一个线程的对应移除操作，反之亦然。CachedThreadPool使用SynchronousQueue，把主线程提交的任务传递给空闲线程执行。CachedThreadPool中任务传递的示意图如图10-7所示。



**10.3　ScheduledThreadPoolExecutor详解**

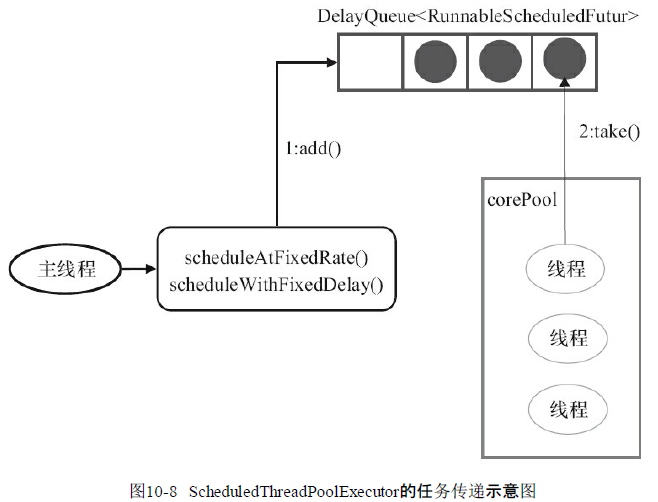
继承自ThreadPoolExecutor。主要用来在给定延迟后运行任务，或定期执行任务。

功能与Timer类似，但更强大、更灵活。

Timer对应的是单个后台线程，而ScheduledThreadPoolExecutor可在构造函数中指定多个对应的后台线程数。

**10.3.1　ScheduledThreadPoolExecutor的运行机制**

执行示意图（本文基于JDK 6）如图10-8所示。



DelayQueue是一个无界队列，所以ThreadPoolExecutor的maximumPoolSize在Scheduled-

ThreadPoolExecutor中没意义（设置maximumPoolSize的大小没效果）。

执行主要分两大部分:

|  |
| --- |
| 1)当调用ScheduledThreadPoolExecutor的scheduleAtFixedRate()方法或scheduleWith-  FixedDelay()方法时，会向ScheduledThreadPoolExecutor的DelayQueue添加一个实现了  RunnableScheduledFutur接口的ScheduledFutureTask。  2)线程池中的线程从DelayQueue中获取ScheduledFutureTask，然后执行任务。 |

ScheduledThreadPoolExecutor为了实现周期性的执行任务,对ThreadPoolExecutor做了如下

的修改。

|  |
| --- |
| ·使用DelayQueue作为任务队列。  ·获取任务的方式不同（后文会说明）。  ·执行周期任务后，增加了额外的处理（后文会说明）。 |

**10.3.2【补】　ScheduledThreadPoolExecutor的实现**

前面提到过，ScheduledThreadPoolExecutor会把待调度的任务（ScheduledFutureTask）

放到一个DelayQueue中。

ScheduledFutureTask主要包含3个成员变量，如下。

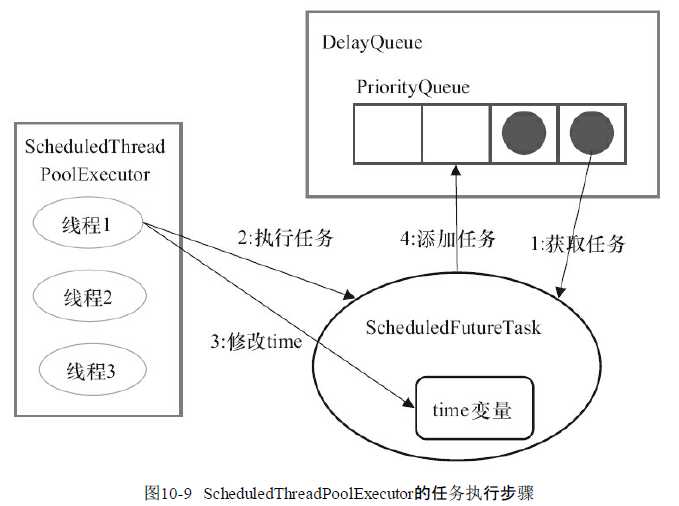
|  |
| --- |
| ·long型成员变量time，表这个任务将要被执行的具体时间。  ·long型成员变量sequenceNumber，表这个任务被添加到ScheduledThreadPoolExecutor中  的序号。  ·long型成员变量period，表任务执行的间隔周期。 |

DelayQueue封装了一个PriorityQueue，会对队列中的ScheduledFutureTask排序。

time小的排在前面（时间早的任务将被先执行）。

如两个ScheduledFutureTask的time相同，就比较sequenceNumber，sequenceNumber小的排在前面（也就是说，如果两个任务的执行时间相同，那么先提交的任务将被先执行）。

首先，看看ScheduledThreadPoolExecutor中的线程执行周期任务的过程。



说明:

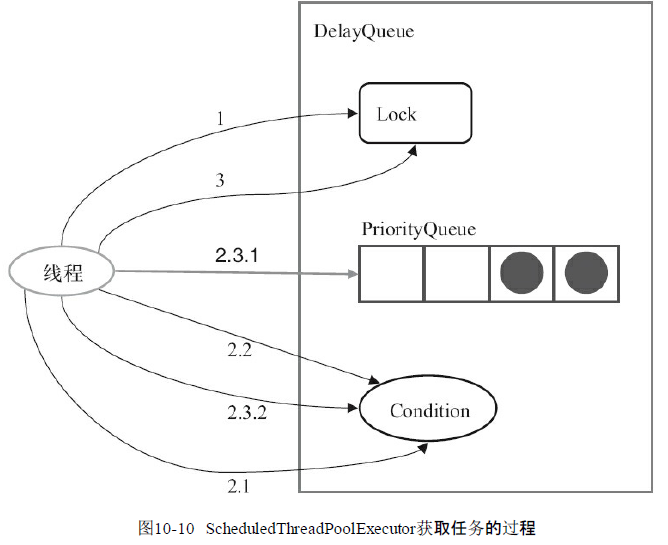
|  |
| --- |
| 1）线程1从DelayQueue中获取已到期的ScheduledFutureTask(DelayQueue.take())。到期任务是指ScheduledFutureTask的time大于等于当前时间。  2）线程1执行这个ScheduledFutureTask。  3）线程1修改ScheduledFutureTask的time变量为下次将要被执行的时间。  4）线程1把这个修改time之后的ScheduledFutureTask放回DelayQueue中（DelayQueue.add()）。 |

看看上面的步骤1）获取任务的过程。

下面是DelayQueue.take()方法的源码。

|  |
| --- |
| public E take() throws InterruptedException {  final ReentrantLock lock = this.lock;  lock.lockInterruptibly();　　　　　　　// 1  try {  for (;;) {  E first = q.peek();  if (first == null) {  available.await();　　　　　　　　　　// 2.1  } else {  long delay = first.getDelay(TimeUnit.NANOSECONDS);  if (delay > 0) {  long tl = available.awaitNanos(delay);　　// 2.2  } else {  E x = q.poll();　　　　　　　　　　 // 2.3.1  assert x != null;  if (q.size() != 0)  available.signalAll();　　　　　　　　 // 2.3.2  return x;  }  }  }  } finally {  lock.unlock();　　　　　　　　　　　　　　 // 3  }  } |

图10-10是DelayQueue.take()的执行示意图。



获取任务分为3大步骤。

|  |
| --- |
| 1）获取Lock。  2）获取周期任务。  ·如PriorityQueue为空，当前线程到Condition中等待；否则执行下面的2.2。  ·如PriorityQueue的头元素的time时间比当前时间大，到Condition中等待到time时间；否则执行下面的2.3。  ·获取PriorityQueue的头元素（2.3.1）；如果PriorityQueue不为空，则唤醒在Condition中等待的所有线程（2.3.2）。  3）释放Lock。  ScheduledThreadPoolExecutor在一个循环中执行步骤2，直到线程从PriorityQueue获取到一  个元素之后（执行2.3.1之后），才会退出无限循环（结束步骤2）。 |

* ScheduledThreadPoolExecutor中线程执行任务的步骤4，把ScheduledFutureTask放入DelayQueue中的过程。下面是DelayQueue.add()的源代码实现。

|  |
| --- |
| public boolean offer(E e) {  final ReentrantLock lock = this.lock;  lock.lock();　　　　　　　　　　 // 1  try {  E first = q.peek();  q.offer(e);　　　　　　　　 // 2.1  if (first == null || e.compareTo(first) < 0)  available.signalAll();　　　 // 2.2  return true;  } finally {  lock.unlock();　　　　　　　　 // 3  }  } |

图10-11是DelayQueue.add()的执行示意图。



添加任务分为3大步骤。

|  |
| --- |
| 1）获取Lock。  2）添加任务。  ·向PriorityQueue添加任务。  ·如在上面2.1中添加的任务是PriorityQueue的头元素,唤醒在Condition中等待的所有线程。  3）释放Lock。 |

**10.4　FutureTask详解**

Future接口和实现Future接口的FutureTask类，代表异步计算的结果。

**10.4.1　FutureTask简介**

FutureTask除了实现Future接口外，还实现了Runnable接口。

因此，可交给Executor执行，也可由调用线程直接执行（FutureTask.run()）。

根据FutureTask.run()方法被执行的时机，FutureTask可处于下面3种状态:

|  |
| --- |
| 1）未启动。FutureTask.run()方法还没有被执行之前，FutureTask处于未启动状态。当创建一  个FutureTask，且没有执行FutureTask.run()方法之前，这个FutureTask处于未启动状态。  2）已启动。FutureTask.run()方法被执行的过程中，FutureTask处于已启动状态。  3）已完成。FutureTask.run()方法执行完后正常结束，或被取消（FutureTask.cancel（…）），或执行FutureTask.run()方法时抛出异常而异常结束，FutureTask处于已完成状态。 |

图10-12是FutureTask的状态迁移的示意图。

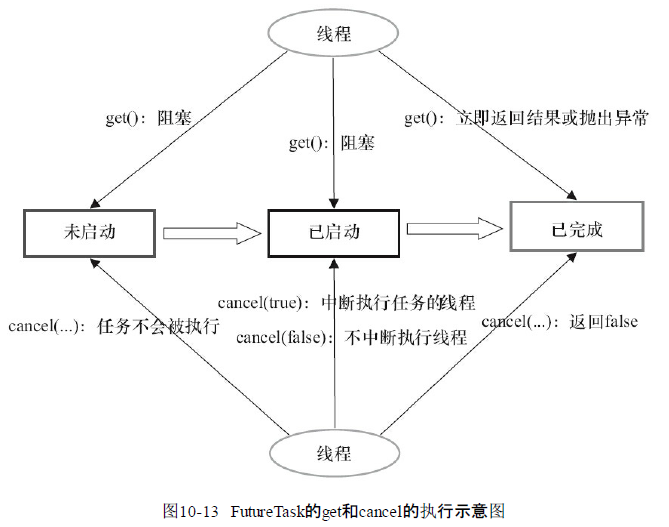


|  |
| --- |
| 当FutureTask处于未启动或已启动状态时，执行FutureTask.get()将导致调用线程阻塞；  当FutureTask处于已完成状态时，执行FutureTask.get()方法将导致调用线程立即返回结果或抛出异常。  当FutureTask处于未启动状态时，执行FutureTask.cancel()将导致此任务永远不会被执行；  当FutureTask处于已启动状态时:执行FutureTask.cancel（true）方法将以中断执行此任务线程的方式来试图停止任务；执行FutureTask.cancel（false）方法将不会对正在执行此任务的线程产生影响 |

当FutureTask处于已启动状态时，执行FutureTask.cancel（false）方法将不会对正在执行此任务的线程产生影响（让正在执行的任务运行完成）；

当FutureTask处于已完成状态时，执行FutureTask.cancel（…）方法将返回false。

图10-13是get方法和cancel方法的执行示意图。



**10.4.2　FutureTask的使用**

* 可把FutureTask交给Executor执行；
* 也可通过ExecutorService.submit（…）方法返回一个

FutureTask，然后执行FutureTask.get()方法或FutureTask.cancel（…）方法。

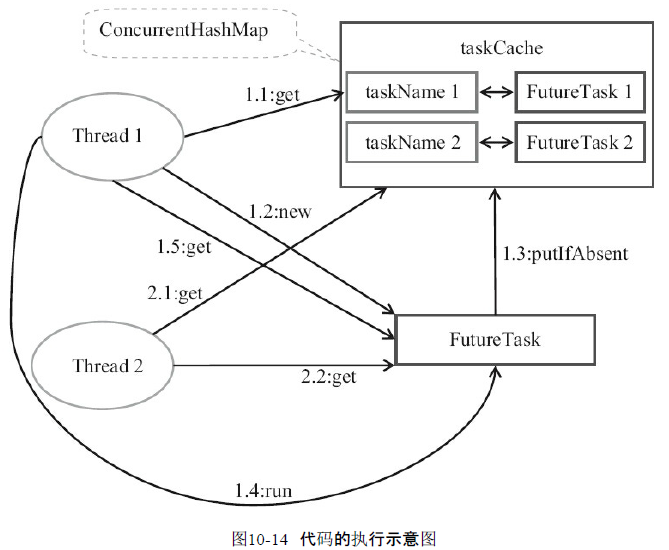
* 还可单独用FutureTask。

当一个线程需要等待另一个线程把某个任务执行完后它才能继续执行，可用FutureTask。

假设有多个线程执行若干任务，每个任务最多只能被执行一次。当多个线程试图同时执行同一个任务时，只允许一个线程执行任务，其他线程需要等待这个任务执行完后才能继续执行。下面是对应的示例代码。

|  |
| --- |
| private final ConcurrentMap<Object, Future<String>> taskCache =  new ConcurrentHashMap<Object, Future<String>>();  private String executionTask(final String taskName)throws ExecutionException, InterruptedException {  while (true) {  Future<String> future = taskCache.get(taskName);　　 // 1.1,2.1  if (future == null) {  Callable<String> task = new Callable<String>() {  public String call() throws InterruptedException {  return taskName;  }  };  FutureTask<String> futureTask = new FutureTask<String>(task);  future = taskCache.putIfAbsent(taskName, futureTask);　 // 1.3  if (future == null) {  future = futureTask;  futureTask.run();　　　　　　　　 // 1.4执行任务  }  }  try {  return future.get();　　　　　　 // 1.5,2.2  } catch (CancellationException e) {  taskCache.remove(taskName, future);  }  }  } |

上述代码的执行示意图如图10-14所示。



当两线程试图同时执行同一任务，如Thread 1执行1.3后Thread 2执行2.1，接下来Thread 2将在2.2等待，直到Thread 1执行完1.4后Thread 2才能从2.2（FutureTask.get()）返回。

**10.4.3　[没仔细看懂]FutureTask的实现**

* 基于AbstractQueuedSynchronizer（AQS）。

java.util.concurrent中的很多可阻塞类（比如ReentrantLock）都是基于AQS来实现的。

AQS是一个同步框架，提供通机制来原子性管理同步状态、阻塞和唤醒线程，以及维护被阻塞线程的队列。JDK 6中AQS被广泛使用，基于AQS实现的同步器包括：ReentrantLock、Semaphore、ReentrantReadWriteLock、CountDownLatch和FutureTask。

* 每个基于AQS实现的同步器都会包含两种类型的操作:

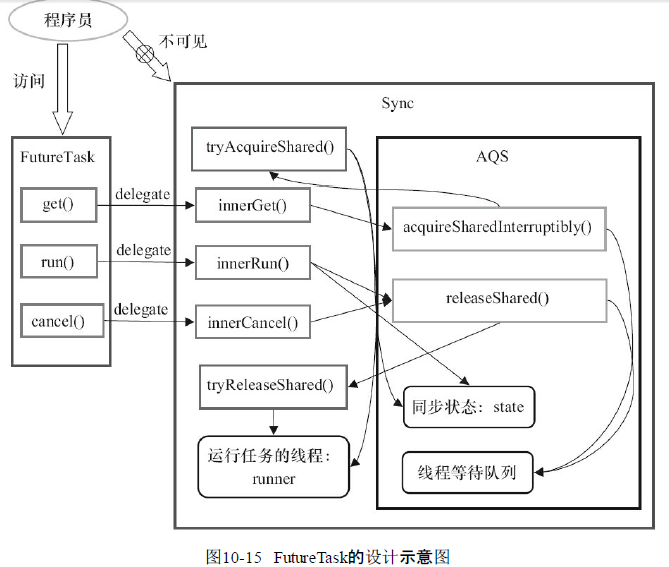
|  |
| --- |
| ·至少一个acquire(get()/get（long timeout，TimeUnit uni0)操作。这个操作阻塞调用线程，除非/直到AQS的状态允许该线程继续执行。  ·至少一个release操作。该操作改变AQS的状态，改变后的状态可允许一个或多个阻塞  线程被解除阻塞。FutureTask的release操作包括run()方法和cancel（…）方法。 |

基于“复合优先于继承”的原则，FutureTask声明了一个内部私有的继承于AQS的子类

Sync，对FutureTask所有公有方法的调用都会委托给这个内部子类。

AQS被作为“模板方法模式”的基础类提供给FutureTask的内部子类Sync，这个内部子类只需要实现状态检查和状态更新的方法即可，这些方法将控制FutureTask的获取和释放操作。具体来说，Sync实现了AQS的tryAcquireShared（int）方法和tryReleaseShared（int）方法，Sync通过这两个方法来检查和更新同步状态。

* FutureTask的设计示意图如图10-15所示。



如图所示，Sync是FutureTask的内部私有类，继承自AQS。

创建FutureTask时会创建内部私有的成员对象Sync，FutureTask所有的的公有方法都直接委托给了内部私有的Sync。

FutureTask.get()方法会调用AQS.acquireSharedInterruptibly（int arg）方法，该方法的执行

过程如下：

1）调用AQS.acquireSharedInterruptibly(int arg)方法，该方法首先会回调在子类Sync中实

现的tryAcquireShared()方法来判断acquire操作是否可以成功。acquire操作可成功的条件为：state为执行完成状态RAN或已取消状态CANCELLED，且runner不为null。

2）如果成功则get()方法立即返回。如果失败则到线程等待队列中去等待其他线程执行

release操作。

3）当其他线程执行release操作（比如FutureTask.run()或FutureTask.cancel（…））唤醒当前线程后，当前线程再次执行tryAcquireShared()将返回正值1，当前线程将离开线程等待队列并唤醒它的后继线程（这里会产生级联唤醒的效果，后面会介绍）。

4）最后返回计算的结果或抛出异常。

* **FutureTask.run()的执行过程**如下。

1）执行在构造函数中指定的任务（Callable.call()）。

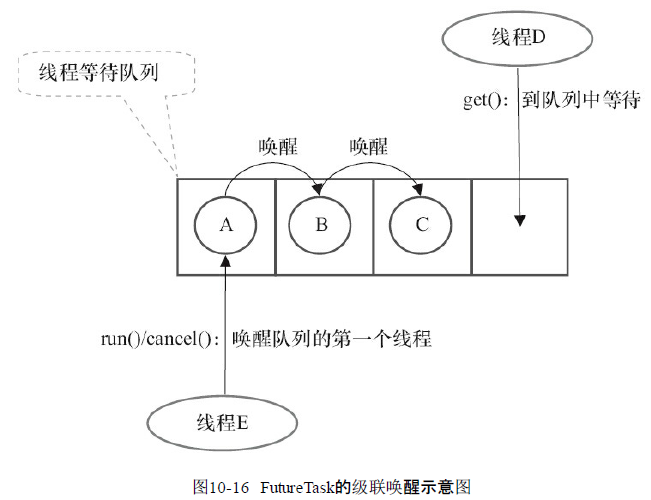
2）以原子方式来更新同步状态（调用AQS.compareAndSetState（int expect，int update），设置state为执行完成状态RAN）。如果这个原子操作成功，就设置代表计算结果的变量result的值为Callable.call()的返回值，然后调用AQS.releaseShared（int arg）。

3）AQS.releaseShared（int arg）首先会回调在子类Sync中实现的tryReleaseShared（arg）来执行release操作（设置运行任务的线程runner为null，然会返回true）；AQS.releaseShared（int arg），然后唤醒线程等待队列中的第一个线程。

4）调用FutureTask.done()。

当执行FutureTask.get()方法时，如果FutureTask不是处于执行完成状态RAN或已取消状态

CANCELLED，当前执行线程将到AQS的线程等待队列中等待（见下图的线程A、B、C和D）。当某个线程执行FutureTask.run()方法或FutureTask.cancel（...）方法时，会唤醒线程等待队列的第一个线程（见图10-16所示的线程E唤醒线程A）。



假设开始时FutureTask处于未启动状态或已启动状态，等待队列中已经有3个线程（A、B和C）在等待。此时，线程D执行get()方法将导致线程D也到等待队列中去等待。

当线程E执行run()方法时，会唤醒队列中的第一个线程A。线程A被唤醒后，首先把自己从队列中删除，然后唤醒它的后继线程B，最后线程A从get()方法返回。线程B、C和D重复A线程的处理流程。最终，在队列中等待的所有线程都被级联唤醒并从get()方法返回。