# New Thread

## New thread的缺点：

1. 每次new Thread新建对象性能差。  
   b. 线程缺乏统一管理，可能无限制新建线程，相互之间竞争，及可能占用过多系统资源导致死机或oom。  
   c. 缺乏更多功能，如定时执行、定期执行、线程中断。

# Java线程池

## 相关链接

<https://www.jianshu.com/p/135c89001b61>

<https://www.cnblogs.com/zhaoyan001/p/7049627.html>

## 线程池的创建和获取

### 创建

new ThreadPoolExecutor

### 获取

一般我们直接使用Executors.newCachedThreadPool();

或者Executors.newCachedThreadPool(threadFactory);获取线程池。

其实获取时，jdk也是调用了new ThreadPoolExecutor方法进行创建的。

### 注

线程工厂详情看<ThreadFactory.docx>

## 线程池的优势：

a. 重用存在的线程，减少对象创建、消亡的开销，性能佳。  
b. 可有效控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。  
c. 提供定时执行、定期执行、单线程、并发数控制等功能。

## 相关线程池

Java通过Executors提供五种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。  
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。  
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

newWorkStealingPool 获取当前可用的线程数量进行创建作为并行级别

## 相关介绍

### newCachedThreadPool

* 重用之前的线程
* 适合执行许多短期异步任务的程序。
* 调用 execute() 将重用以前构造的线程
* 如果没有可用的线程，则创建一个新线程并添加到池中
* 默认为60s未使用就被终止和移除
* 长期闲置的池将会不消耗任何资源

线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。示例代码如下：

|  |
| --- |
| ExecutorService cachedThreadPool = Executors.newCachedThreadPool();  for (int i = 0; i < 10; i++) {  final int index = i;  try {  Thread.sleep(index \* 1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  cachedThreadPool.execute(new Runnable() {  @Override  public void run() {  System.out.println(index);  }  });  } |

### newFixedThreadPool

* 创建重用固定数量线程的线程池，不能随时新建线程
* 当所有线程都处于活动状态时，如果提交了其他任务，  
  他们将在队列中等待一个线程可用
* 线程会一直存在，直到调用shutdown

因为线程池大小为3，每个任务输出index后sleep 2秒，所以每两秒打印3个数字。  
定长线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()。可参考PreloadDataCache。

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。示例代码如下：

|  |
| --- |
| ExecutorService fixedThreadPool = Executors.newFixedThreadPool(3);  for (int i = 0; i < 10; i++) {  final int index = i;  fixedThreadPool.execute(new Runnable() {  @Override  public void run() {  try {  System.out.println(index);  Thread.sleep(2000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  });  } |

这种方式是没有返回值的线程.

### newScheduledThreadPool

创建一个定长线程池，支持定时及周期性任务执行。延迟执行示例代码如下：

表示延迟3秒执行。

|  |
| --- |
| ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);  scheduledThreadPool.schedule(new Runnable() {  @Override  public void run() {  System.out.println("delay 3 seconds");  }  }, 3, TimeUnit.SECONDS); |

定期执行示例代码如下：

|  |
| --- |
| scheduledThreadPool.scheduleAtFixedRate(new Runnable() {  @Override  public void run() {  System.out.println("delay 1 seconds, and excute every 3 seconds");  }  }, 1, 3, TimeUnit.SECONDS); |

表示延迟1秒后每3秒执行一次。  
ScheduledExecutorService比Timer更安全，功能更强大，后面会有一篇单独进行对比。

### newSingleThreadExecutor

* 在任何情况下都不会有超过一个任务处于活动状态
* 与newFixedThreadPool(1)不同是不能重新配置加入线程，使用 FinalizableDelegatedExecutorService进行包装
* 能保证执行顺序，先提交的先执行。
* 当线程执行中出现异常，去创建一个新的线程替换之

结果依次输出，相当于顺序执行各个任务。  
现行大多数GUI程序都是单线程的。Android中单线程可用于数据库操作，文件操作，应用批量安装，应用批量删除等不适合并发但可能IO阻塞性及影响UI线程响应的操作。

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。示例代码如下：

|  |
| --- |
| ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();  for (int i = 0; i < 10; i++) {  final int index = i;  singleThreadExecutor.execute(new Runnable() {  @Override  public void run() {  try {  System.out.println(index);  Thread.sleep(2000);  } catch (InterruptedException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  }  });  } |

### **newWorkStealingPool**

* 获取当前可用的线程数量进行创建作为并行级别
* 使用ForkJoinPool

public static ExecutorService newWorkStealingPool() {

return new ForkJoinPool

(Runtime.getRuntime().availableProcessors(),

ForkJoinPool.defaultForkJoinWorkerThreadFactory,

null, true);

}

通过源码可以看出底层调用的是ForkJoinPool线程池

public ForkJoinPool(int parallelism,

ForkJoinWorkerThreadFactory factory,

UncaughtExceptionHandler handler,

boolean asyncMode) {

this(checkParallelism(parallelism),

checkFactory(factory),

handler,

asyncMode ? FIFO\_QUEUE : LIFO\_QUEUE,

"ForkJoinPool-" + nextPoolId() + "-worker-");

checkPermission();

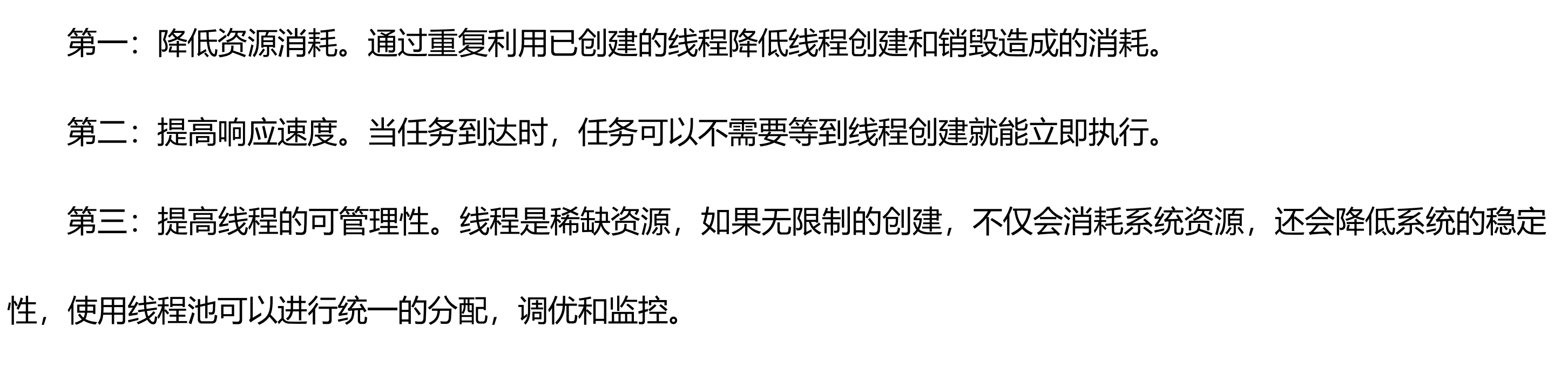
}

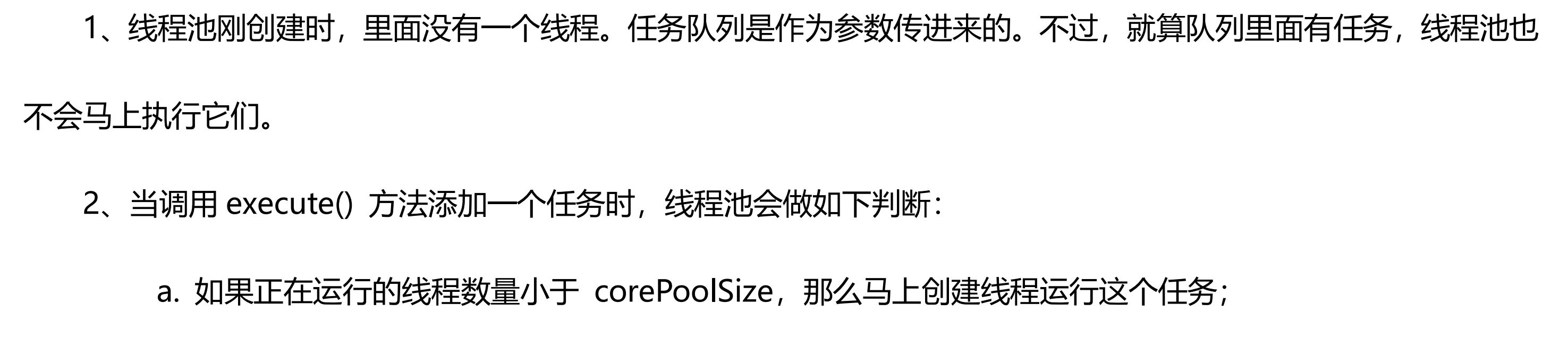
使用一个无限队列来保存需要执行的任务，可以传入线程的数量，不传入，则默认使用当前计算机中可用的cpu数量，使用分治法来解决问题，使用fork()和join()来进行调用

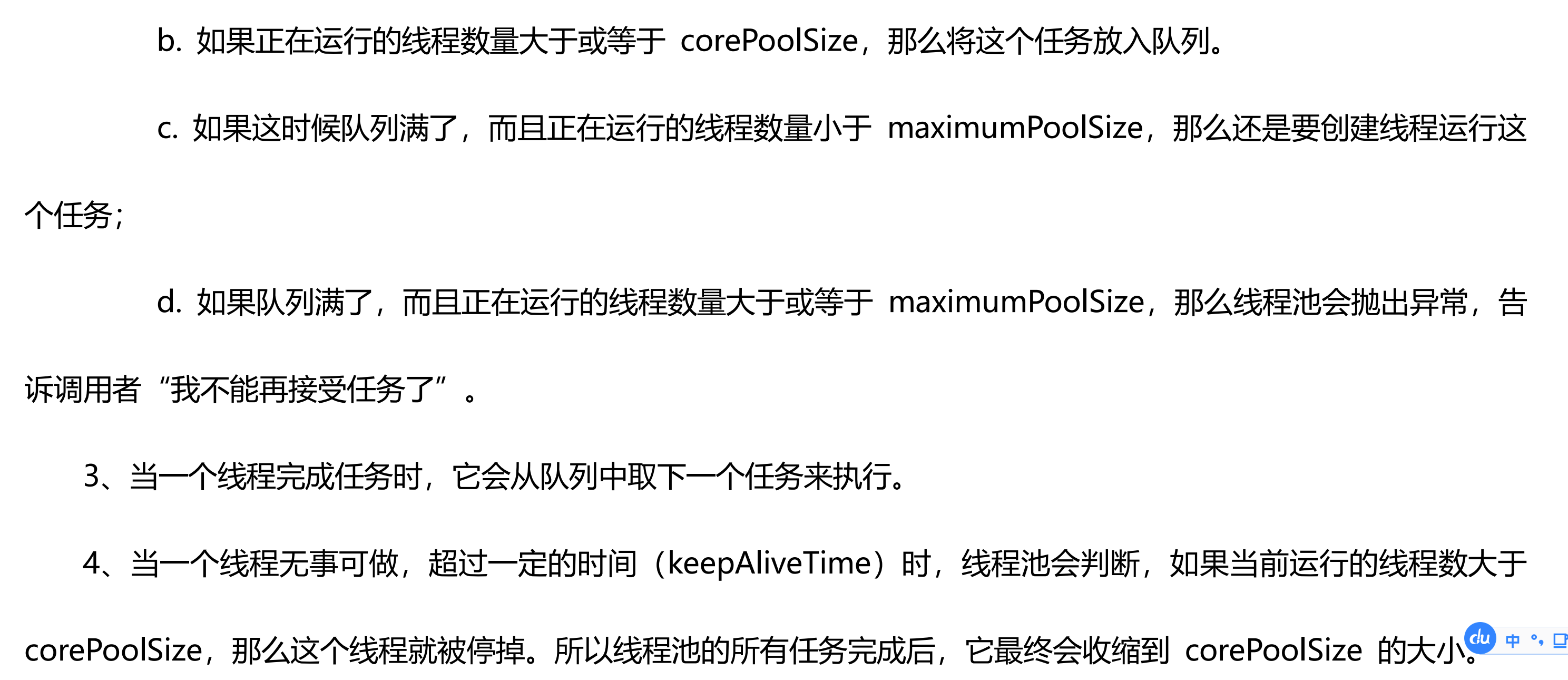
## 五种线程池的适用场景

1. newCachedThreadPool：用来创建一个可以无限扩大的线程池，适用于服务器负载较轻，执行很多短期异步任务。
2. newFixedThreadPool：创建一个固定大小的线程池，因为采用无界的阻塞队列，所以实际线程数量永远不会变化，适用于可以预测线程数量的业务中，或者服务器负载较重，对当前线程数量进行限制。
3. newSingleThreadExecutor：创建一个单线程的线程池，适用于需要保证顺序执行各个任务，并且在任意时间点，不会有多个线程是活动的场景。
4. newScheduledThreadPool：可以延时启动，定时启动的线程池，适用于需要多个后台线程执行周期任务的场景。
5. newWorkStealingPool：创建一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行，适用于大耗时的操作，可以并行来执行

# 使用线程池的好处、启动策略、







# Callable和Future [ˈfjuːtʃə(r)]

## Callable与Runnable

Runnable中的run方法返回值为void类型，所以在执行完任务之后无法返回任何结果。

Callable位于java.util.concurrent包下，它也是一个接口，在它里面也只声明了一个方法，只不过这个方法叫做call()：

public interface Callable<V> {

/\*\*

\* Computes a result, or throws an exception if unable to do so.

\*

\* @return computed result

\* @throws Exception if unable to compute a result

\*/

V call() throws Exception;

}

可以看到，这是一个泛型接口，call()函数返回的类型就是传递进来的V类型。

怎么使用callback？？？？

一般情况下是配合ExecutorService来使用的，在ExecutorService接口中声明了若干个submit方法的重载版本

<T> Future<T> submit(Callable<T> task);

<T> Future<T> submit(Runnable task, T result);

Future<?> submit(Runnable task);

第一个submit方法里面的参数类型就是Callable。

暂时只需要知道Callable一般是和ExecutorService配合来使用的，具体的使用方法讲在后面讲述。

一般情况下我们使用第一个submit方法和第三个submit方法，第二个submit方法很少使用。

## Future

　Future就是对于具体的Runnable或者Callable任务的执行结果进行取消、查询是否完成、获取结果。必要时可以通过get方法获取执行结果，该方法会阻塞直到任务返回结果。

　Future类位于java.util.concurrent包下，它是一个接口：

public interface Future<V> {

boolean cancel(boolean mayInterruptIfRunning);

boolean isCancelled();

boolean isDone();

V get() throws InterruptedException, ExecutionException;

V get(long timeout, TimeUnit unit)

throws InterruptedException, ExecutionException, TimeoutException;

}

### 在Future接口中声明了5个方法，下面依次解释每个方法的作用：

cancel方法用来取消任务，如果取消任务成功则返回true，如果取消任务失败则返回false。参数mayInterruptIfRunning表示是否允许取消正在执行却没有执行完毕的任务，如果设置true，则表示可以取消正在执行过程中的任务。如果任务已经完成，则无论mayInterruptIfRunning为true还是false，此方法肯定返回false，即如果取消已经完成的任务会返回false；如果任务正在执行，若mayInterruptIfRunning设置为true，则返回true，若mayInterruptIfRunning设置为false，则返回false；如果任务还没有执行，则无论mayInterruptIfRunning为true还是false，肯定返回true。

isCancelled方法表示任务是否被取消成功，如果在任务正常完成前被取消成功，则返回 true。

isDone方法表示任务是否已经完成，若任务完成，则返回true；

get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；

get(long timeout, TimeUnit unit)用来获取执行结果，如果在指定时间内，还没获取到结果，就直接返回null。

### 也就是说Future提供了三种功能

1. 判断任务是否完成
2. 能够中断任务
3. 能够获取任务的执行结果

因为Future只是一个接口，所以是无法直接用来创建对象使用的，因此就有了下面的FutureTask。

## FutureTask

### FutureTask实现了RunnableFuture接口，这个接口的定义如下：

public interface RunnableFuture<V> extends Runnable, Future<V> {

void run();

}

可以看到这个接口实现了Runnable和Future接口，接口中的具体实现由FutureTask来实现。这个类的两个构造方法如下 ：

public FutureTask(Callable<V> callable) {

if (callable == null)

throw new NullPointerException();

sync = new Sync(callable);

}

public FutureTask(Runnable runnable, V result) {

sync = new Sync(Executors.callable(runnable, result));

}

如上提供了两个构造函数，一个以Callable为参数，另外一个以Runnable为参数。这些类之间的关联对于任务建模的办法非常灵活，允许你基于FutureTask的Runnable特性（因为它实现了Runnable接口），把任务写成Callable，然后封装进一个由执行者调度并在必要时可以取消的FutureTask。

FutureTask可以由执行者调度，这一点很关键。它对外提供的方法基本上就是Future和Runnable接口的组合：get()、cancel、isDone()、isCancelled()和run()，而run()方法通常都是由执行者调用，我们基本上不需要直接调用它。

### 一个FutureTask的例子

public class MyCallable implements Callable<String> {

private long waitTime;

public MyCallable(int timeInMillis){

this.waitTime=timeInMillis;

}

@Override

public String call() throws Exception {

Thread.sleep(waitTime);

//return the thread name executing this callable task

return Thread.currentThread().getName();

}

}

public class FutureTaskExample {

public static void main(String[] args) {

MyCallable callable1 = new MyCallable(1000); // 要执行的任务

MyCallable callable2 = new MyCallable(2000);

FutureTask<String> futureTask1 = new FutureTask<String>(callable1);// 将Callable写的任务封装到一个由执行者调度的FutureTask对象

FutureTask<String> futureTask2 = new FutureTask<String>(callable2);

ExecutorService executor = Executors.newFixedThreadPool(2); // 创建线程池并返回ExecutorService实例

executor.execute(futureTask1); // 执行任务

executor.execute(futureTask2);

while (true) {

try {

if(futureTask1.isDone() && futureTask2.isDone()){// 两个任务都完成

System.out.println("Done");

executor.shutdown(); // 关闭线程池和服务

return;

}

if(!futureTask1.isDone()){ // 任务1没有完成，会等待，直到任务完成

System.out.println("FutureTask1 output="+futureTask1.get());

}

System.out.println("Waiting for FutureTask2 to complete");

String s = futureTask2.get(200L, TimeUnit.MILLISECONDS);

if(s !=null){

System.out.println("FutureTask2 output="+s);

}

} catch (InterruptedException | ExecutionException e) {

e.printStackTrace();

}catch(TimeoutException e){

//do nothing

}

}

}

}

运行如上程序后，可以看到一段时间内没有输出，因为get()方法等待任务执行完成然后才输出内容.

输出结果如下：

FutureTask1 output=pool-1-thread-1

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

Waiting for FutureTask2 to complete

FutureTask2 output=pool-1-thread-2

Done

### 一个FutureTask的例子

Task类

|  |
| --- |
| class Task implements Callable<Integer>{  @Override  public Integer call() throws Exception {  System.out.println("子线程在进行计算");  Thread.sleep(3000);  int sum = 0;  for(int i=0;i<100;i++)  sum += i;  return sum;  }  } |

|  |
| --- |
| public class CallableAndFutureTask {  public static void main(String[] args) {  //第一种方式  ExecutorService executor = Executors.newCachedThreadPool();  Task task = new Task();  FutureTask<Integer> futureTask = new FutureTask<Integer>(task);  executor.submit(futureTask);  executor.shutdown();    //第二种方式，注意这种方式和第一种方式效果是类似的，只不过一个使用的是ExecutorService，一个使用的是Thread  /\*Task task = new Task();  FutureTask<Integer> futureTask = new FutureTask<Integer>(task);  Thread thread = new Thread(futureTask);  thread.start();\*/    try {  Thread.sleep(1000);  } catch (InterruptedException e1) {  e1.printStackTrace();  }    System.out.println("主线程在执行任务");    try {  System.out.println("task运行结果"+futureTask.get());  } catch (InterruptedException e) {  e.printStackTrace();  } catch (ExecutionException e) {  e.printStackTrace();  }    System.out.println("所有任务执行完毕");  }    } |

注：

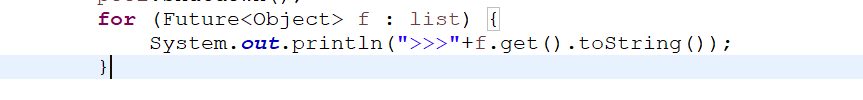
1. 这里也是用的submit方法进行提交任务，但是 不返回Future了，而是直接将任务结果 放到了futureTask中，在后面只需要从futureTask中调用get方法获取返回结果就行了。
2. 如果调用submit方法还返回值Future，这个Future是空的。获取不到返回结果的。
3. 注释的地方，可以用Thread去执行这个任务，因为futureTask实现了RunnableFuture， 而RunnableFuture又继承了Runnable，Future。就是因为这个Runnable所以可以用线程 方式启动

# CompletionService

## 将任务执行结果Future放到List中的缺点

Future.get()方法用来获取执行结果，这个方法会产生阻塞，会一直等到任务执行完毕才返回；

前面讲的事例中是将任务返回的Future对象放到list中自己去维护。



上图就是存放执行结果的list。遍历该list方法获取Future对象，输出返回结果。

分析一下缺点：

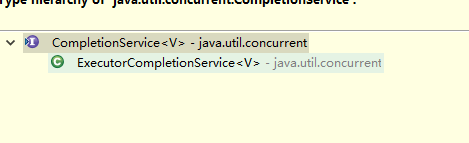
上面已经说过Future.get()方法会导致阻塞，假如第一个任务1秒就能完成，第二个任务100秒完成，剩下的任务各需2秒完成。

在上面的List的遍历时，第一个future在1秒后输出任务返回的结果，到第二个的时候，由于get方法导致阻塞，所以只能等到第二个任务执行完毕后get方法才能获取到结果。其实在任务二执行的过程中，后面的任务已经执行完成。但是由于任务二的阻塞，只能等待任务二完成。

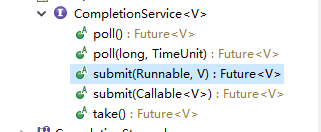
这样的效果不是我们想要的。

## 解决办法CompletionService

### 实现类



### 方法



Future<V> submit(Callable<V> task)

提交Callable类型的task；

Future<V> submit(Runnable task, V result)

提交Runnable类型的task；

Future<V> take() throws InterruptedException

获取并移除已完成状态的task，如果目前不存在这样的task，则等待；

只要有运行完成的就会返回这个Future对象。

Future<V> poll()

获取并移除已完成状态的task，如果目前不存在这样的task，返回null；

Future<V> poll(long timeout, TimeUnit unit) throws InterruptedException

获取并移除已完成状态的task，如果在指定等待时间内不存在这样的task，返回null

### 例子

|  |
| --- |
| public class CompletionServiceTest {  public static void main(String[] args) throws InterruptedException, ExecutionException {  testExecutorCompletionService();  }    private static void testExecutorCompletionService()throws InterruptedException, ExecutionException{  int numThread = 5;  ExecutorService executor = Executors.newFixedThreadPool(numThread);    CompletionService<String> completionService = new ExecutorCompletionService<String>(executor);  for(int i = 0; i < numThread; i++ ){  completionService.submit(new Task(i));  }    for(int i = 0; i < numThread; i++ ){  Future<String> f = completionService.take();  System.out.println(f.get());  }  executor.shutdown();    }  } |

Task类

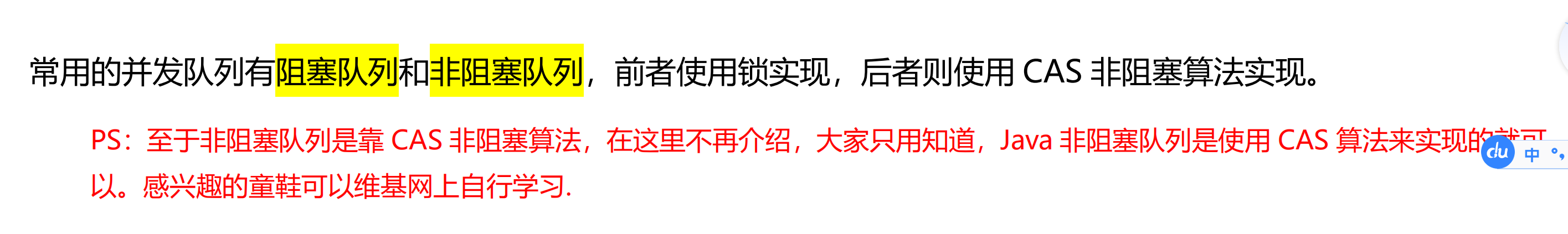
|  |
| --- |
| class Task implements Callable<String>{  private int i ;    public Task(int i){  this.i= i;  }    @Override  public String call() throws Exception {  // TODO Auto-generated method stub  if(i==1){  Thread.sleep(10000);  }  return Thread.currentThread().getName()+"执行任务"+i;  }    } |

# 任务队列--还需要深入学习，2018年最新版java程序员面试宝典

任务队列分为非阻塞队列和阻塞队列。他们都是线程安全的，这里的阻塞和非阻塞在于有界与否。

1. 对于阻塞有界队列，如果队列满了，则任何线程都会阻塞不能进行入队操作，反之， 如果队列为空的话，则任何线程都不能进行出队操作。
2. 对于非阻塞无界队列，不会出现队满或者队空的情况，就是不能有一个以上的线程 同时对队列进行入队或者出队操作。

## 非阻塞队列：ConcurrentLinkedQueue



CAS：

Compare and Swap，即比较再交换

例如主内存中的值为50，线程1与线程2同时对这个数进行修改，那么线程1和线程2 同时读取到该数50，然后线程1进行了+1并且刷新到了主内存中，此时线程2也进行 了修改+1，也要刷新到主内存中，如果操作成功，两次操作相当于成功了一次。结果还 是51.但是CAS算法保证线程2在刷新到主内存之前进行一次比较，就是比较当时取出 的数50和现在主内存中的数据是否一致，如果一直就刷新到内存中，如果不一致则不 能刷新到主内存中，重新取数进行操作。线程1的操作流程也是如此。所有的线程操作 都是这样，先比较再交换。

## 阻塞队列：BlockingQueue

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。

　　DealyQueue：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。 只能容纳一个元素

　　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。

## 阻塞无界队列DelayQueue [dɪˈleɪ]

延时队列，队列中的每个元素都有一个过期时间，并且队列是个优先级队列，当从队列获取元素时，只有过期的元素才会出队。

过期时间的设置：

task实现Delayed接口（这里用的是Message）

|  |
| --- |
| public class Message implements Delayed{  private int id;  private String body;  private long excuteTime;    public int getId() {  return id;  }  public String getBody() {  return body;  }  public long getExcuteTime() {  return excuteTime;  }  public void setId(int id) {  this.id = id;  }  public void setBody(String body) {  this.body = body;  }  public void setExcuteTime(long excuteTime) {  this.excuteTime = excuteTime;  }  public Message(int id, String body, long delayTime) {  this.id = id;  this.body = body;  this.excuteTime = TimeUnit.NANOSECONDS.convert(delayTime, TimeUnit.MILLISECONDS) + System.nanoTime();  }  @Override  public int compareTo(Delayed delayed) {  Message msg = (Message) delayed;  return Integer.valueOf(this.id) > Integer.valueOf(msg.id) ? 1  : (Integer.valueOf(this.id) < Integer.valueOf(msg.id) ? -1 : 0);  }  @Override  public long getDelay(TimeUnit unit) {  return unit.convert(this.excuteTime - System.nanoTime(), TimeUnit.NANOSECONDS);  }  } |
|  |

重写compareTo和getDelay方法，在这里计算过期时间。excuteTime 就是过期时间。

例如设置的过期时间是1000，意思就是1秒后这个任务才能从队列中出队。

Consumer类：

|  |
| --- |
| public class Consumer implements Runnable {  // 延时队列 ,消费者从其中获取消息进行消费  private DelayQueue<Message> queue;    public Consumer(DelayQueue<Message> queue) {  this.queue = queue;  }    @Override  public void run() {  while (true) {  try {  Message take = queue.take();  System.out.println("消费消息id：" + take.getId() + " 消息体：" + take.getBody());  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } |

实现Runnable接口，重写run方法，run方法中通过queue.take()方法获取任务。

从延迟队列中获取任务。

主方法：

|  |
| --- |
| public class QueueTest {  public static void main(String[] args) {  DelayQueue<Message> queue = new DelayQueue<Message>();  Message m1 = new Message(1, "执行任务一", 1000);  Message m2 = new Message(2, "执行任务二", 2000);    queue.offer(m1);  queue.offer(m2);  ExecutorService exec = Executors.newFixedThreadPool(1);  exec.execute(new Consumer(queue));    Message m3 = new Message(3, "3", 3000);  queue.offer(m3);  exec.execute(new Consumer(queue));  exec.shutdown();  }  } |

