线程池

什么是线程池

线程池（ThreadPool）是一种基于池化思想管理和使用线程的机制。它是将多个线程预先存储在一个“池子”内，当有任务出现时可以避免重新创建和销毁线程所带来性能开销，只需要从“池子”内取出相应的线程执行对应的任务即可。

使用new Thread()创建线程的弊端：

- 每次通过new Thread()创建对象性能不佳。
- 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。
- 缺乏更多功能，如定时执行、定期执行、线程中断。

使用Java线程池的好处：

- 重用存在的线程，减少对象创建、消亡的开销，提升性能。
- 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。
- 提供定时执行、定期执行、单线程、并发数控制等功能。

同时**阿里巴巴在其《Java开发手册》中也强制规定：线程资源必须通过线程池提供，不允许在应用中自行显式创建线程。**

> 说明：线程池的好处是减少在创建和销毁线程上所消耗的时间以及系统资源的开销，解决资源不足的问题。如果不使用线程池，有可能造成系统创建大量同类线程而导致消耗完内存或者“过度切换”的问题。

线程池使用

newCachedThreadPool

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

```
1 public static void main(String[] args) {
2     ExecutorService cacheThreadPool =
  Executors.newCachedThreadPool();
3     for(int i =1;i<=5;i++){
4         final int index=i ;
5         try{
```

```
6                 Thread.sleep(50);
7             }catch(InterruptedException  e )
  {
8                 e.printStackTrace();
9             }
10            cacheThreadPool.execute(new
  Runnable(){
11                @Override
12                public void run() {
13                    System.out.println("第"
  +index +"个线程"
  +Thread.currentThread().getName());
14                }
15            });
16        }
17    }
```

newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

```
1  public class ThreadPoolExecutorTest {
2      public static void main(String[]
  args) {
```

```
3         ExecutorService  fixedThreadPool
  =Executors. newFixedThreadPool(3);
4         for (int i =1; i<=5;i++){
5             final int index=i ;
6             fixedThreadPool.execute(new
  Runnable(){
7                 @Override
8                 public void run() {
9                     try {
10
   System.out.println("第" +index + "个线
  程" +Thread.currentThread().getName());
11
   Thread.sleep(1000);
12                    }
   catch(InterruptedException  e ) {
13                        e
  .printStackTrace();
14                    }
15                }});
16        }
17    }
```

newScheduledThreadPool

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
corePoolSize - 池中所保存的线程数，即使线程是空闲的也包括在内。

延迟执行示例代码：

```
1  public class ThreadPoolExecutorTest {
2      public static void main(String[] args) {
3          ScheduledExecutorService scheduledThreadPool= Executors.newScheduledThreadPool(3);
4
   scheduledThreadPool.schedule(newRunnable(){
5              @Override
6              public void run() {
7               System.out.println("延迟三秒");
8               }
9          }, 3, TimeUnit.SECONDS);
10     }
11 }
12
```

表示延迟3秒执行。 定期执行示例代码：

```
1 public class ThreadPoolExecutorTest {
2     public static void main(String[]
  args) {
3         ScheduledExecutorService
  scheduledThreadPool=
  Executors.newScheduledThreadPool(3);

4
   scheduledThreadPool.scheduleAtFixedRate
  (newRunnable(){
5             @Override
6             public void run() {
7                 System.out.println("延迟
  1秒后每三秒执行一次");
8             }
9         },1,3,TimeUnit.SECONDS);
10     }
11 }
```

表示延迟1秒后每3秒执行一次。

newSingleThreadExecutor

创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。（注意，如果因为在关闭前的执行期间出现失败而终止了此单个线程，那么如果需要，一个新线程将代替它执行后续的任务）。可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。与其他等效的newFixedThreadPool(1)不同，可保证无需重新配置此方法所返回的执行程序即可使用其他的线程。

public static ExecutorService newSingleThreadExecutor()

```
1 public class ThreadPoolExecutorTest {
2     public static void main(String[] args) {
3         ExecutorService singleThreadPool= Executors.newSingleThreadExecutor();

4         for(int i=1;i<=5;i++){
5             int index=i;
6             singleThreadPool.execute(new Runnable(){
7                 @Override
8                 public void run() {

9                     try{
```

```
10
    System.out.println("第"+index+"个线程");
11
    Thread.sleep(2000);
12
   }catch(InterruptedException e) {

13
    e.printStackTrace();
14                      }
15                  }  });
16          }
17      }
18 }
```

线程池原理

```
1 public ThreadPoolExecutor(int
  corePoolSize,
2                           int
  maximumPoolSize,
3                           long
  keepAliveTime,
4                           TimeUnit unit,
5
    BlockingQueue<Runnable> workQueue,
```

```
6                                 ThreadFactory
  threadFactory,
7
   RejectedExecutionHandler handler) {
8      if (corePoolSize < 0 ||
9          maximumPoolSize <= 0 ||
10         maximumPoolSize < corePoolSize
  ||
11         keepAliveTime < 0)
12         throw new
  IllegalArgumentException();
13     if (workQueue == null ||
  threadFactory == null || handler ==
  null)
14         throw new
  NullPointerException();
15     this.acc =
  System.getSecurityManager() == null ?
16             null :
17
   AccessController.getContext();
18     this.corePoolSize = corePoolSize;
19     this.maximumPoolSize =
  maximumPoolSize;
20     this.workQueue = workQueue;
21     this.keepAliveTime =
  unit.toNanos(keepAliveTime);
```

```
22      this.threadFactory = threadFactory;
23      this.handler = handler;
24  }
```

corePoolSize

线程池核心线程大小

线程池中会维护一个最小的线程数量，即使这些线程处理空闲状态，他们也不会被销毁，除非设置了allowCoreThreadTimeOut。这里的最小线程数量即是corePoolSize。

maximumPoolSize

线程池最大线程数量

一个任务被提交到线程池以后，首先会找有没有空闲存活线程，如果有则直接将任务交给这个空闲线程来执行，如果没有则会缓存到工作队列（后面会介绍）中，如果工作队列满了，才会创建一个新线程，然后从工作队列的头部取出一个任务交由新线程来处理，而将刚提交的任务放入工作队列尾部。线程池不会无限制的去创建新线程，它会有一个最大线程数量的限制，这个数量即由maximunPoolSize指定。

keepAliveTime

空闲线程存活时间

一个线程如果处于空闲状态，并且当前的线程数量大于corePoolSize，那么在指定时间后，这个空闲线程会被销毁，这里的指定时间由keepAliveTime来设定

unit

空闲线程存活时间单位

keepAliveTime的计量单位

workQueue

工作队列

新任务被提交后，会先进入到此工作队列中，任务调度时再从队列中取出任务。jdk中提供的工作队列：

ArrayBlockingQueue

基于数组的有界阻塞队列，按FIFO排序。新任务进来后，会放到该队列的队尾，有界的数组可以防止资源耗尽问题。当线程池中线程数量达到corePoolSize后，再有新任务进来，则会将任务放入该队列的队尾，等待被调度。如果队列已经是满的，则创建一个新线程，如果线程数量已经达到maxPoolSize，则会执行拒绝策略。

LinkedBlockingQuene

基于链表的无界阻塞队列（其实最大容量为Interger.MAX），按照FIFO排序。由于该队列的近似无界性，当线程池中线程数量达到corePoolSize后，再有新任务进来，会一直存入该队列，而不会去创建新线程直到maxPoolSize，因此使用该工作队列时，参数maxPoolSize其实是不起作用的。

SynchronousQuene

一个不缓存任务的阻塞队列，生产者放入一个任务必须等到消费者取出这个任务。也就是说新任务进来时，不会缓存，而是直接被调度执行该任务，如果没有可用线程，则创建新线程，如果线程数量达到maxPoolSize，则执行拒绝策略。

PriorityBlockingQueue

具有优先级的无界阻塞队列，优先级通过参数Comparator实现。

DelayQueue

DelayQueue中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞。 使用场景： DelayQueue使用场景较少，但都相当巧妙，常见的例子比如使用一个DelayQueue来管理一个超时未响应的连接队列。

threadFactory

线程工厂

创建一个新线程时使用的工厂，可以用来设定线程名、是否为daemon线程等等

handler

拒绝策略

当工作队列中的任务已到达最大限制，并且线程池中的线程数量也达到最大限制，这时如果有新任务提交进来，该如何处理呢。这里的拒绝策略，就是解决这个问题的，jdk中提供了4中拒绝策略：

CallerRunsPolicy

该策略下，在调用者线程中直接执行被拒绝任务的run方法，除非线程池已经shutdown，则直接抛弃任务。

AbortPolicy

该策略下，直接丢弃任务，并抛出RejectedExecutionException异常。

DiscardPolicy

该策略下，直接丢弃任务，什么都不做。

DiscardOldestPolicy

该策略下，抛弃进入队列最早的那个任务，然后尝试把这次拒绝的任务放入队列