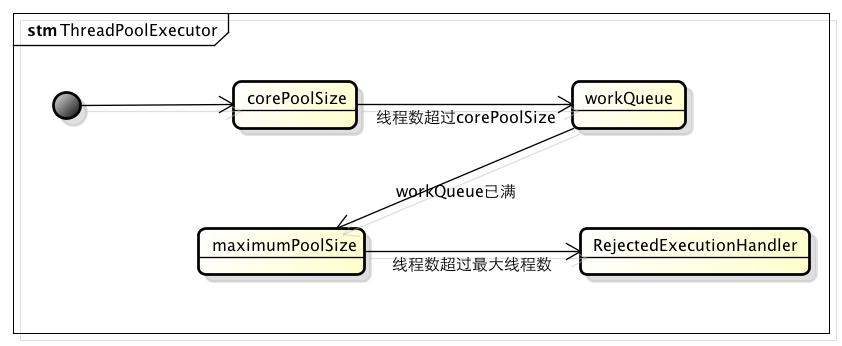
**ThreadPoolExecutor机制**  
**一、概述**   
**1、ThreadPoolExecutor作为java.util.concurrent包对外提供基础实现，以内部线程池的形式对外提供管理任务执行，线程调度，线程池管理等等服务；   
2、Executors方法提供的线程服务，都是通过参数设置来实现不同的线程池机制。   
3、先来了解其线程池管理的机制，有助于正确使用，避免错误使用导致严重故障。同时可以根据自己的需求实现自己的线程池**   
  
**二、核心构造方法讲解**   
**下面是ThreadPoolExecutor最核心的构造方法**

**Java代码  [收藏代码](javascript:void())**

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**.corePoolSize = corePoolSize;
16. **this**.maximumPoolSize = maximumPoolSize;
17. **this**.workQueue = workQueue;
18. **this**.keepAliveTime = unit.toNanos(keepAliveTime);
19. **this**.threadFactory = threadFactory;
20. **this**.handler = handler;
21. }

**构造方法参数讲解** 

|  |  |
| --- | --- |
| 参数名 | 作用 |
| corePoolSize | 核心线程池大小 |
| maximumPoolSize | 最大线程池大小 |
| keepAliveTime | 线程池中超过corePoolSize数目的空闲线程最大存活时间；可以allowCoreThreadTimeOut(true)使得核心线程有效时间 |
| TimeUnit | keepAliveTime时间单位 |
| workQueue | 阻塞任务队列 |
| threadFactory | 新建线程工厂 |
| RejectedExecutionHandler | 当提交任务数超过maxmumPoolSize+workQueue之和时，任务会交给RejectedExecutionHandler来处理 |

**重点讲解：**   
其中比较容易让人误解的是：corePoolSize，maximumPoolSize，workQueue之间关系。   
  
1.当线程池小于corePoolSize时，新提交任务将创建一个新线程执行任务，即使此时线程池中存在空闲线程。   
2.当线程池达到corePoolSize时，新提交任务将被放入workQueue中，等待线程池中任务调度执行   
3.当workQueue已满，且maximumPoolSize>corePoolSize时，新提交任务会创建新线程执行任务   
4.当提交任务数超过maximumPoolSize时，新提交任务由RejectedExecutionHandler处理   
5.当线程池中超过corePoolSize线程，空闲时间达到keepAliveTime时，关闭空闲线程   
6.当设置allowCoreThreadTimeOut(true)时，线程池中corePoolSize线程空闲时间达到keepAliveTime也将关闭   
  
**线程管理机制图示：**   
  
  
**三、Executors提供的线程池配置方案**   
  
**1、构造一个固定线程数目的线程池，配置的corePoolSize与maximumPoolSize大小相同，同时使用了一个无界LinkedBlockingQueue存放阻塞任务，因此多余的任务将存在再阻塞队列，不会由RejectedExecutionHandler处理**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** ExecutorService newFixedThreadPool(**int** nThreads) {
2. **return** **new** ThreadPoolExecutor(nThreads, nThreads,
3. 0L, TimeUnit.MILLISECONDS,
4. **new** LinkedBlockingQueue<Runnable>());
5. }

**2、构造一个缓冲功能的线程池，配置corePoolSize=0，maximumPoolSize=Integer.MAX\_VALUE，keepAliveTime=60s,以及一个无容量的阻塞队列 SynchronousQueue，因此任务提交之后，将会创建新的线程执行；线程空闲超过60s将会销毁**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** ExecutorService newCachedThreadPool() {
2. **return** **new** ThreadPoolExecutor(0, Integer.MAX\_VALUE,
3. 60L, TimeUnit.SECONDS,
4. **new** SynchronousQueue<Runnable>());
5. }

**3、构造一个只支持一个线程的线程池，配置corePoolSize=maximumPoolSize=1，无界阻塞队列LinkedBlockingQueue；保证任务由一个线程串行执行**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** ExecutorService newSingleThreadExecutor() {
2. **return** **new** FinalizableDelegatedExecutorService
3. (**new** ThreadPoolExecutor(1, 1,
4. 0L, TimeUnit.MILLISECONDS,
5. **new** LinkedBlockingQueue<Runnable>()));
6. }

**4、构造有定时功能的线程池，配置corePoolSize，无界延迟阻塞队列DelayedWorkQueue；有意思的是：maximumPoolSize=Integer.MAX\_VALUE，由于DelayedWorkQueue是无界队列，所以这个值是没有意义的**

**Java代码  [收藏代码](javascript:void())**

1. **public** **static** ScheduledExecutorService newScheduledThreadPool(**int** corePoolSize) {
2. **return** **new** ScheduledThreadPoolExecutor(corePoolSize);
3. }
5. **public** **static** ScheduledExecutorService newScheduledThreadPool(
6. **int** corePoolSize, ThreadFactory threadFactory) {
7. **return** **new** ScheduledThreadPoolExecutor(corePoolSize, threadFactory);
8. }
10. **public** ScheduledThreadPoolExecutor(**int** corePoolSize,
11. ThreadFactory threadFactory) {
12. **super**(corePoolSize, Integer.MAX\_VALUE, 0, TimeUnit.NANOSECONDS,
13. **new** DelayedWorkQueue(), threadFactory);
14. }

**四、定制属于自己的非阻塞线程池**

**Java代码  [收藏代码](javascript:void())**

1. **import** java.util.concurrent.ArrayBlockingQueue;
2. **import** java.util.concurrent.ExecutorService;
3. **import** java.util.concurrent.RejectedExecutionHandler;
4. **import** java.util.concurrent.ThreadFactory;
5. **import** java.util.concurrent.ThreadPoolExecutor;
6. **import** java.util.concurrent.TimeUnit;
7. **import** java.util.concurrent.atomic.AtomicInteger;

10. **public** **class** CustomThreadPoolExecutor {

13. **private** ThreadPoolExecutor pool = **null**;

16. /\*\*
17. \* 线程池初始化方法
18. \*
19. \* corePoolSize 核心线程池大小----10
20. \* maximumPoolSize 最大线程池大小----30
21. \* keepAliveTime 线程池中超过corePoolSize数目的空闲线程最大存活时间----30+单位TimeUnit
22. \* TimeUnit keepAliveTime时间单位----TimeUnit.MINUTES
23. \* workQueue 阻塞队列----new ArrayBlockingQueue<Runnable>(10)====10容量的阻塞队列
24. \* threadFactory 新建线程工厂----new CustomThreadFactory()====定制的线程工厂
25. \* rejectedExecutionHandler 当提交任务数超过maxmumPoolSize+workQueue之和时,
26. \*                          即当提交第41个任务时(前面线程都没有执行完,此测试方法中用sleep(100)),
27. \*                                任务会交给RejectedExecutionHandler来处理
28. \*/
29. **public** **void** init() {
30. pool = **new** ThreadPoolExecutor(
31. 10,
32. 30,
33. 30,
34. TimeUnit.MINUTES,
35. **new** ArrayBlockingQueue<Runnable>(10),
36. **new** CustomThreadFactory(),
37. **new** CustomRejectedExecutionHandler());
38. }

41. **public** **void** destory() {
42. **if**(pool != **null**) {
43. pool.shutdownNow();
44. }
45. }

48. **public** ExecutorService getCustomThreadPoolExecutor() {
49. **return** **this**.pool;
50. }
52. **private** **class** CustomThreadFactory **implements** ThreadFactory {
54. **private** AtomicInteger count = **new** AtomicInteger(0);
56. @Override
57. **public** Thread newThread(Runnable r) {
58. Thread t = **new** Thread(r);
59. String threadName = CustomThreadPoolExecutor.**class**.getSimpleName() + count.addAndGet(1);
60. System.out.println(threadName);
61. t.setName(threadName);
62. **return** t;
63. }
64. }

67. **private** **class** CustomRejectedExecutionHandler **implements** RejectedExecutionHandler {
69. @Override
70. **public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
71. // 记录异常
72. // 报警处理等
73. System.out.println("error.............");
74. }
75. }


79. // 测试构造的线程池
80. **public** **static** **void** main(String[] args) {
81. CustomThreadPoolExecutor exec = **new** CustomThreadPoolExecutor();
82. // 1.初始化
83. exec.init();
85. ExecutorService pool = exec.getCustomThreadPoolExecutor();
86. **for**(**int** i=1; i<100; i++) {
87. System.out.println("提交第" + i + "个任务!");
88. pool.execute(**new** Runnable() {
89. @Override
90. **public** **void** run() {
91. **try** {
92. Thread.sleep(3000);
93. } **catch** (InterruptedException e) {
94. e.printStackTrace();
95. }
96. System.out.println("running=====");
97. }
98. });
99. }


103. // 2.销毁----此处不能销毁,因为任务没有提交执行完,如果销毁线程池,任务也就无法执行了
104. // exec.destory();
106. **try** {
107. Thread.sleep(10000);
108. } **catch** (InterruptedException e) {
109. e.printStackTrace();
110. }
111. }
112. }

**方法中建立一个核心线程数为30个，缓冲队列有10个的线程池。每个线程任务，执行时会先睡眠3秒，保证提交10任务时，线程数目被占用完，再提交30任务时，阻塞队列被占用完，，这样提交第41个任务是，会交给CustomRejectedExecutionHandler 异常处理类来处理。**   
  
**提交任务的代码如下：**

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** execute(Runnable command) {
2. **if** (command == **null**)
3. **throw** **new** NullPointerException();
4. /\*
5. \* Proceed in 3 steps:
6. \*
7. \* 1. If fewer than corePoolSize threads are running, try to
8. \* start a new thread with the given command as its first
9. \* task.  The call to addWorker atomically checks runState and
10. \* workerCount, and so prevents false alarms that would add
11. \* threads when it shouldn't, by returning false.
12. \*
13. \* 2. If a task can be successfully queued, then we still need
14. \* to double-check whether we should have added a thread
15. \* (because existing ones died since last checking) or that
16. \* the pool shut down since entry into this method. So we
17. \* recheck state and if necessary roll back the enqueuing if
18. \* stopped, or start a new thread if there are none.
19. \*
20. \* 3. If we cannot queue task, then we try to add a new
21. \* thread.  If it fails, we know we are shut down or saturated
22. \* and so reject the task.
23. \*/
24. **int** c = ctl.get();
25. **if** (workerCountOf(c) < corePoolSize) {
26. **if** (addWorker(command, **true**))
27. **return**;
28. c = ctl.get();
29. }
30. **if** (isRunning(c) && workQueue.offer(command)) {
31. **int** recheck = ctl.get();
32. **if** (! isRunning(recheck) && remove(command))
33. reject(command);
34. **else** **if** (workerCountOf(recheck) == 0)
35. addWorker(**null**, **false**);
36. }
37. **else** **if** (!addWorker(command, **false**))
38. reject(command);
39. }

**注意：41以后提交的任务就不能正常处理了，因为，execute中提交到任务队列是用的offer方法，如上面代码，这个方法是非阻塞的，所以就会交给CustomRejectedExecutionHandler 来处理，所以对于大数据量的任务来说，这种线程池，如果不设置队列长度会OOM，设置队列长度，会有任务得不到处理，接下来我们构建一个阻塞的自定义线程池**   
  
**五、定制属于自己的阻塞线程池**

**Java代码  [收藏代码](javascript:void())**

1. **package** com.tongbanjie.trade.test.commons;
3. **import** java.util.concurrent.ArrayBlockingQueue;
4. **import** java.util.concurrent.ExecutorService;
5. **import** java.util.concurrent.RejectedExecutionHandler;
6. **import** java.util.concurrent.ThreadFactory;
7. **import** java.util.concurrent.ThreadPoolExecutor;
8. **import** java.util.concurrent.TimeUnit;
9. **import** java.util.concurrent.atomic.AtomicInteger;
11. **public** **class** CustomThreadPoolExecutor {

14. **private** ThreadPoolExecutor pool = **null**;

17. /\*\*
18. \* 线程池初始化方法
19. \*
20. \* corePoolSize 核心线程池大小----1
21. \* maximumPoolSize 最大线程池大小----3
22. \* keepAliveTime 线程池中超过corePoolSize数目的空闲线程最大存活时间----30+单位TimeUnit
23. \* TimeUnit keepAliveTime时间单位----TimeUnit.MINUTES
24. \* workQueue 阻塞队列----new ArrayBlockingQueue<Runnable>(5)====5容量的阻塞队列
25. \* threadFactory 新建线程工厂----new CustomThreadFactory()====定制的线程工厂
26. \* rejectedExecutionHandler 当提交任务数超过maxmumPoolSize+workQueue之和时,
27. \*                          即当提交第41个任务时(前面线程都没有执行完,此测试方法中用sleep(100)),
28. \*                                任务会交给RejectedExecutionHandler来处理
29. \*/
30. **public** **void** init() {
31. pool = **new** ThreadPoolExecutor(
32. 1,
33. 3,
34. 30,
35. TimeUnit.MINUTES,
36. **new** ArrayBlockingQueue<Runnable>(5),
37. **new** CustomThreadFactory(),
38. **new** CustomRejectedExecutionHandler());
39. }

42. **public** **void** destory() {
43. **if**(pool != **null**) {
44. pool.shutdownNow();
45. }
46. }

49. **public** ExecutorService getCustomThreadPoolExecutor() {
50. **return** **this**.pool;
51. }
53. **private** **class** CustomThreadFactory **implements** ThreadFactory {
55. **private** AtomicInteger count = **new** AtomicInteger(0);
57. @Override
58. **public** Thread newThread(Runnable r) {
59. Thread t = **new** Thread(r);
60. String threadName = CustomThreadPoolExecutor.**class**.getSimpleName() + count.addAndGet(1);
61. System.out.println(threadName);
62. t.setName(threadName);
63. **return** t;
64. }
65. }

68. **private** **class** CustomRejectedExecutionHandler **implements** RejectedExecutionHandler {
70. @Override
71. **public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
72. **try** {
73. // 核心改造点，由blockingqueue的offer改成put阻塞方法
74. executor.getQueue().put(r);
75. } **catch** (InterruptedException e) {
76. e.printStackTrace();
77. }
78. }
79. }


83. // 测试构造的线程池
84. **public** **static** **void** main(String[] args) {
86. CustomThreadPoolExecutor exec = **new** CustomThreadPoolExecutor();
87. // 1.初始化
88. exec.init();
90. ExecutorService pool = exec.getCustomThreadPoolExecutor();
91. **for**(**int** i=1; i<100; i++) {
92. System.out.println("提交第" + i + "个任务!");
93. pool.execute(**new** Runnable() {
94. @Override
95. **public** **void** run() {
96. **try** {
97. System.out.println(">>>task is running=====");
98. TimeUnit.SECONDS.sleep(10);
99. } **catch** (InterruptedException e) {
100. e.printStackTrace();
101. }
102. }
103. });
104. }

107. // 2.销毁----此处不能销毁,因为任务没有提交执行完,如果销毁线程池,任务也就无法执行了
108. // exec.destory();
110. **try** {
111. Thread.sleep(10000);
112. } **catch** (InterruptedException e) {
113. e.printStackTrace();
114. }
115. }
116. }

**解释：当提交任务被拒绝时，进入拒绝机制，我们实现拒绝方法，把任务重新用阻塞提交方法put提交，实现阻塞提交任务功能，防止队列过大，OOM，提交被拒绝方法在下面** 

**Java代码  [收藏代码](javascript:void())**

1. **public** **void** execute(Runnable command) {
2. **if** (command == **null**)
3. **throw** **new** NullPointerException();
5. **int** c = ctl.get();
6. **if** (workerCountOf(c) < corePoolSize) {
7. **if** (addWorker(command, **true**))
8. **return**;
9. c = ctl.get();
10. }
11. **if** (isRunning(c) && workQueue.offer(command)) {
12. **int** recheck = ctl.get();
13. **if** (! isRunning(recheck) && remove(command))
14. reject(command);
15. **else** **if** (workerCountOf(recheck) == 0)
16. addWorker(**null**, **false**);
17. }
18. **else** **if** (!addWorker(command, **false**))
19. // 进入拒绝机制， 我们把runnable任务拿出来，重新用阻塞操作put，来实现提交阻塞功能
20. reject(command);
21. }

**总结：**   
**1、用ThreadPoolExecutor自定义线程池，看线程是的用途，如果任务量不大，可以用无界队列，如果任务量非常大，要用有界队列，防止OOM   
2、如果任务量很大，还要求每个任务都处理成功，要对提交的任务进行阻塞提交，重写拒绝机制，改为阻塞提交。保证不抛弃一个任务   
3、最大线程数一般设为2N+1最好，N是CPU核数   
4、核心线程数，看应用，如果是任务，一天跑一次，设置为0，合适，因为跑完就停掉了，如果是常用线程池，看任务量，是保留一个核心还是几个核心线程数   
5、如果要获取任务执行结果，用CompletionService，但是注意，获取任务的结果的要重新开一个线程获取，如果在主线程获取，就要等任务都提交后才获取，就会阻塞大量任务结果，队列过大OOM，所以最好异步开个线程获取结果**