**概述**

**什么是线程池？**

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。

new Thread的弊端如下:

a: 每次new thread新建对象,性能很差.

b: 线程池缺乏统一管理,可能无限制新建线程,相互之间竞争,极可能占用过多资源导致死机或oom.

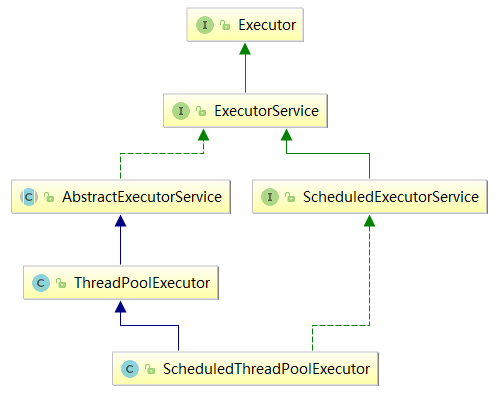
c: 匮乏更多功能,如定时执行,定期执行,线程中断.

**为什么要用线程池？**

* 降低资源消耗
  + 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
* 提高响应速度
  + 当任务到达时，任务可以不需要等到线程创建就能立即执行。
* 提高线程的可管理性
  + 使用线程池可以进行统一的分配，调优和监控。可有效控制组大线程并发数,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。
* Java通过Executors提供四种线程池，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。

**Executor 框架**

**简介**



* Executor：一个接口，其定义了一个接收 Runnable 对象的方法 executor，其方法签名为 executor(Runnable command),
* ExecutorService：是一个比 Executor 使用更广泛的子类接口，其提供了生命周期管理的方法，以及可跟踪一个或多个异步任务执行状况返回 Future 的方法。
* AbstractExecutorService：ExecutorService 执行方法的默认实现。
* ScheduledExecutorService：一个可定时调度任务的接口。
* ScheduledThreadPoolExecutor：ScheduledExecutorService 的实现，一个可定时调度任务的线程池。
* ThreadPoolExecutor：线程池，可以通过调用 Executors 以下静态工厂方法来创建线程池并返回一个 ExecutorService 对象。

**ThreadPoolExecutor**

java.uitl.concurrent.ThreadPoolExecutor 类是 Executor 框架中最核心的一个类。

ThreadPoolExecutor 有四个构造方法，前三个都是基于第四个实现。第四个构造方法定义如下：

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) {

**参数说明**

* corePoolSize：线程池的基本线程数。这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了 prestartAllCoreThreads()或者 prestartCoreThread()方法，从这 2 个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建 corePoolSize 个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为 0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到 corePoolSize 后，就会把到达的任务放到缓存队列当中。
* maximumPoolSize：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。
* 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，并且每个任务执行的时间比较短，可以调大这个时间，提高线程的利用率。
* unit：参数 keepAliveTime 的时间单位，有 7 种取值。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)和毫微秒(NANOSECONDS, 千分之一微秒)。
* workQueue：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 （先进先出） 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 LinkedBlockingQueue，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。
* threadFactory：创建线程的工厂。可以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。
* handler：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。以下是 JDK1.5 提供的四种策略。
  + 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。
  + 当然也可以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不能处理的任务。

**重要方法**

在 ThreadPoolExecutor 类中有几个非常重要的方法：

* execute() 方法实际上是 Executor 中声明的方法，在 ThreadPoolExecutor 进行了具体的实现，这个方法是 ThreadPoolExecutor 的核心方法，通过这个方法可以向线程池提交一个任务，交由线程池去执行。
* submit() 方法是在 ExecutorService 中声明的方法，在 AbstractExecutorService 就已经有了具体的实现，在 ThreadPoolExecutor 中并没有对其进行重写，这个方法也是用来向线程池提交任务的，但是它和 execute()方法不同，它能够返回任务执行的结果，去看 submit()方法的实现，会发现它实际上还是调用的 execute()方法，只不过它利用了 Future 来获取任务执行结果（Future 相关内容将在下一篇讲述）。
* shutdown() 和 shutdownNow() 是用来关闭线程池的。

**向线程池提交任务**

我们可以使用 execute 提交任务，但是 execute 方法没有返回值，所以无法判断任务是否被线程池执行成功。

通过以下代码可知 execute 方法输入的任务是一个 Runnable 实例。

threadsPool.execute(new Runnable() {

@Override

public void run() {

// TODO Auto-generated method stub

}

});

我们也可以使用 submit 方法来提交任务，它会返回一个 Future ，那么我们可以通过这个 Future 来判断任务是否执行成功。

通过 Future 的 get 方法来获取返回值，get 方法会阻塞住直到任务完成。而使用 get(long timeout, TimeUnit unit) 方法则会阻塞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(harReturnValuetask);

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();

}

**线程池的关闭**

我们可以通过调用线程池的 shutdown 或 shutdownNow 方法来关闭线程池，它们的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。但是它们存在一定的区别，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，而 shutdown 只是将线程池的状态设置成 SHUTDOWN 状态，然后中断所有没有正在执行任务的线程。

只要调用了这两个关闭方法的其中一个，isShutdown 方法就会返回 true。当所有的任务都已关闭后,才表示线程池关闭成功，这时调用 isTerminaed 方法会返回 true。至于我们应该调用哪一种方法来关闭线程池，应该由提交到线程池的任务特性决定，通常调用 shutdown 来关闭线程池，如果任务不一定要执行完，则可以调用 shutdownNow。

**Executors**

JDK 中提供了几种具有代表性的线程池，这些线程池是基于 ThreadPoolExecutor 的定制化实现。

在实际使用线程池的场景中，我们往往不是直接使用 ThreadPoolExecutor ，而是使用 JDK 中提供的具有代表性的线程池实例。

**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。

这种类型的线程池特点是：

* 工作线程的创建数量几乎没有限制（其实也有限制的,数目为 Interger.MAX\_VALUE）, 这样可灵活的往线程池中添加线程。
* 如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间（默认为 1 分钟），则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。
* 在使用 CachedThreadPool 时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

示例：

public class CachedThreadPoolDemo {

public static void main(String[] args) {

ExecutorService executorService = Executors.newCachedThreadPool();

for (int i = 0; i < 10; i++) {

final int index = i;

try {

Thread.sleep(index \* 1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

executorService.execute(() -> System.out.println(Thread.currentThread().getName() + " 执行，i = " + index));

}

}

}

**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。

FixedThreadPool 是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

示例：

public class FixedThreadPoolDemo {

public static void main(String[] args) {

ExecutorService executorService = Executors.newFixedThreadPool(3);

for (int i = 0; i < 10; i++) {

final int index = i;

executorService.execute(() -> {

try {

System.out.println(Thread.currentThread().getName() + " 执行，i = " + index);

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

}

}

}

**newSingleThreadExecutor**

创建一个单线程化的 Executor，即只创建唯一的工作者线程来执行任务，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。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是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。

示例：

public class SingleThreadExecutorDemo {

public static void main(String[] args) {

ExecutorService executorService = Executors.newSingleThreadExecutor();

for (int i = 0; i < 10; i++) {

final int index = i;

executorService.execute(() -> {

try {

System.out.println(Thread.currentThread().getName() + " 执行，i = " + index);

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

});

}

}

}

**newScheduleThreadPool**

创建一个线程池，可以安排任务在给定延迟后运行，或定期执行。

public class ScheduledThreadPoolDemo {

private static void delay() {

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

scheduledThreadPool.schedule(() -> System.out.println(Thread.currentThread().getName() + " 延迟 3 秒"), 3,

TimeUnit.SECONDS);

}

private static void cycle() {

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(5);

scheduledThreadPool.scheduleAtFixedRate(

() -> System.out.println(Thread.currentThread().getName() + " 延迟 1 秒，每 3 秒执行一次"), 1, 3,

TimeUnit.SECONDS);

}

public static void main(String[] args) {

delay();

cycle();

}

}

**源码**

线程池的具体实现原理，大致从以下几个方面讲解：

1. 线程池状态
2. 任务的执行
3. 线程池中的线程初始化
4. 任务缓存队列及排队策略
5. 任务拒绝策略
6. 线程池的关闭
7. 线程池容量的动态调整

**线程池状态**

// runState is stored in the high-order bits

private static final int RUNNING = -1 << COUNT\_BITS;

private static final int SHUTDOWN = 0 << COUNT\_BITS;

private static final int STOP = 1 << COUNT\_BITS;

private static final int TIDYING = 2 << COUNT\_BITS;

private static final int TERMINATED = 3 << COUNT\_BITS;

// Packing and unpacking ctl

private static int runStateOf(int c) { return c & ~CAPACITY; }

runState 表示当前线程池的状态，它是一个 volatile 变量用来保证线程之间的可见性；

下面的几个 static final 变量表示 runState 可能的几个取值。

当创建线程池后，初始时，线程池处于 RUNNING 状态；

RUNNING -> SHUTDOWN

如果调用了 shutdown()方法，则线程池处于 SHUTDOWN 状态，此时线程池不能够接受新的任务，它会等待所有任务执行完毕。

(RUNNING or SHUTDOWN) -> STOP

如果调用了 shutdownNow()方法，则线程池处于 STOP 状态，此时线程池不能接受新的任务，并且会去尝试终止正在执行的任务。

SHUTDOWN -> TIDYING

当线程池和队列都为空时，则线程池处于 TIDYING 状态。

STOP -> TIDYING

当线程池为空时，则线程池处于 TIDYING 状态。

TIDYING -> TERMINATED

当 terminated() 回调方法完成时，线程池处于 TERMINATED 状态。

**任务的执行**

任务执行的核心方法是 execute() 方法。执行步骤如下：

1. 如果少于 corePoolSize 个线程正在运行，尝试使用给定命令作为第一个任务启动一个新线程。对 addWorker 的调用会自动检查 runState 和 workerCount，从而防止在不应该的情况下添加线程。
2. 如果任务排队成功，仍然需要仔细检查是否应该添加一个线程（因为现有的线程自上次检查以来已经死亡）或者自从进入方法后，线程池就关闭了。所以我们重新检查状态，如果有必要的话，在线程池停止状态时回滚队列，如果没有线程的话，就开始一个新的线程。
3. 如果任务排队失败，那么我们尝试添加一个新的线程。如果失败了，说明线程池已经关闭了，或者已经饱和了，所以拒绝这个任务。

public void execute(Runnable command) {

if (command == null)

throw new NullPointerException();

int c = ctl.get();

if (workerCountOf(c) < corePoolSize) {

if (addWorker(command, true))

return;

c = ctl.get();

}

if (isRunning(c) && workQueue.offer(command)) {

int recheck = ctl.get();

if (! isRunning(recheck) && remove(command))

reject(command);

else if (workerCountOf(recheck) == 0)

addWorker(null, false);

}

else if (!addWorker(command, false))

reject(command);

}

**线程池中的线程初始化**

默认情况下，创建线程池之后，线程池中是没有线程的，需要提交任务之后才会创建线程。

在实际中如果需要线程池创建之后立即创建线程，可以通过以下两个方法办到：

prestartCoreThread()：初始化一个核心线程； prestartAllCoreThreads()：初始化所有核心线程

public boolean prestartCoreThread() {

return addIfUnderCorePoolSize(null); //注意传进去的参数是null

}

public int prestartAllCoreThreads() {

int n = 0;

while (addIfUnderCorePoolSize(null))//注意传进去的参数是null

++n;

return n;

}

**任务缓存队列及排队策略**

在前面我们多次提到了任务缓存队列，即 workQueue，它用来存放等待执行的任务。

workQueue 的类型为 BlockingQueue，通常可以取下面三种类型：

1. ArrayBlockingQueue：基于数组的先进先出队列，此队列创建时必须指定大小；
2. LinkedBlockingQueue：基于链表的先进先出队列，如果创建时没有指定此队列大小，则默认为 Integer.MAX\_VALUE；
3. SynchronousQueue：这个队列比较特殊，它不会保存提交的任务，而是将直接新建一个线程来执行新来的任务。

**任务拒绝策略**

当线程池的任务缓存队列已满并且线程池中的线程数目达到 maximumPoolSize，如果还有任务到来就会采取任务拒绝策略，通常有以下四种策略

* ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出 RejectedExecutionException 异常。
* ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
* ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
* ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

**线程池的关闭**

ThreadPoolExecutor 提供了两个方法，用于线程池的关闭，分别是 shutdown()和 shutdownNow()，其中：

* shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务
* shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

**线程池容量的动态调整**

ThreadPoolExecutor 提供了动态调整线程池容量大小的方法：setCorePoolSize()和 setMaximumPoolSize()，

* setCorePoolSize：设置核心池大小
* setMaximumPoolSize：设置线程池最大能创建的线程数目大小

当上述参数从小变大时，ThreadPoolExecutor 进行线程赋值，还可能立即创建新的线程来执行任务。

免费Java资料需要自己领取，涵盖了Java、Redis、MongoDB、MySQL、Zookeeper、Spring Cloud、Dubbo高并发分布式等教程，一共30G。  
传送门：[https://mp.weixin.qq.com/s/JzddfH-7yNudmkjT0IRL8Q](https://link.zhihu.com/?target=https%3A//mp.weixin.qq.com/s/JzddfH-7yNudmkjT0IRL8Q)