引言

合理利用线程池能带来的好处？

- 降低资源消耗。通过重复利用以创建的线程降低线程创建和销毁造成的消耗
- 提高相应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资

源，还会降低系统的稳定性，使用线程池可以进行统一的分配、调优和监控。但是要做到合理的利用线程池，必须对其了解。

线程池的使用

线程池的创建

可以通过`ThreadPoolExecutor`来创建线程池。

```
new ThreadPoolExecutor(corePoolSize,        //线程池基本大小
                       maximumPoolSize,
                       keepAliveTime,
                       milliseconds,
                       runnableTaskQueue,   //任务队列
                       threadFactory,
                       handler);
```

创建一个线程需要输入几个参数

corePoolSize

> corePoolSize(线程池的基本大小)：当提交一个任务到线程池时，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，等到需要执行的任务数大于线程池基本大小时就不会创建。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基本线程。

runnableTaskQueue

> 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： 是一个基于列表结构的阻塞队列，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: 是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，吞吐量通常高于LinkedBlockingQueue,静态工厂方法Executors.newCachedThreadPool使

用了这个队列。

- PriorityBlockingQueue: 是一个具有优先级的无限阻塞队列。

maximumPoolSize

> maximumPoolSize（线程池最大大小）：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，则线程池会在创建新的线程执行任务。值得注意的是`如果使用了无界的队列任务这个参数就没有什么效果`。

ThreadFactory

> ThreadFactory: 用于设置创建线程的工厂，可以通过线程工厂给每个创建出来的线程设置更具有意义的名字，Debug和定位问题时非常有帮助。

RejectedExecutionHandler

> 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。这种策略默认情况下是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，丢弃到。
> - 当然也可以更具应用场景需要来实现RejectdExcutionHandler接口来自定义策略。如记录日志和持久化不处理的任务。

keepAliveTime

> keepAliveTime（线程活动保持时间）： 线程池的工作线程空闲后，保持存货的时间。索引如果任务很多，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。

TimeUnit

> 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)和毫微秒(NANOSECONDS, 千分之一微秒)。

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方法输入的任务时一个Runnable类的实例。

```
threadsPool.execute(new Runnable() {
@Override
    public void run() {

    // TODO Auto-generated method stub

    }

});
```

- 也可以使用submit方法来提交任务，他会返回一个Future，那么可以通过这个Future来判断任务是否执行成功，通过future的get方法可以获取返回值，get方法会阻塞住知道任务完成，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。

```
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，但是他们的实现原理不同，

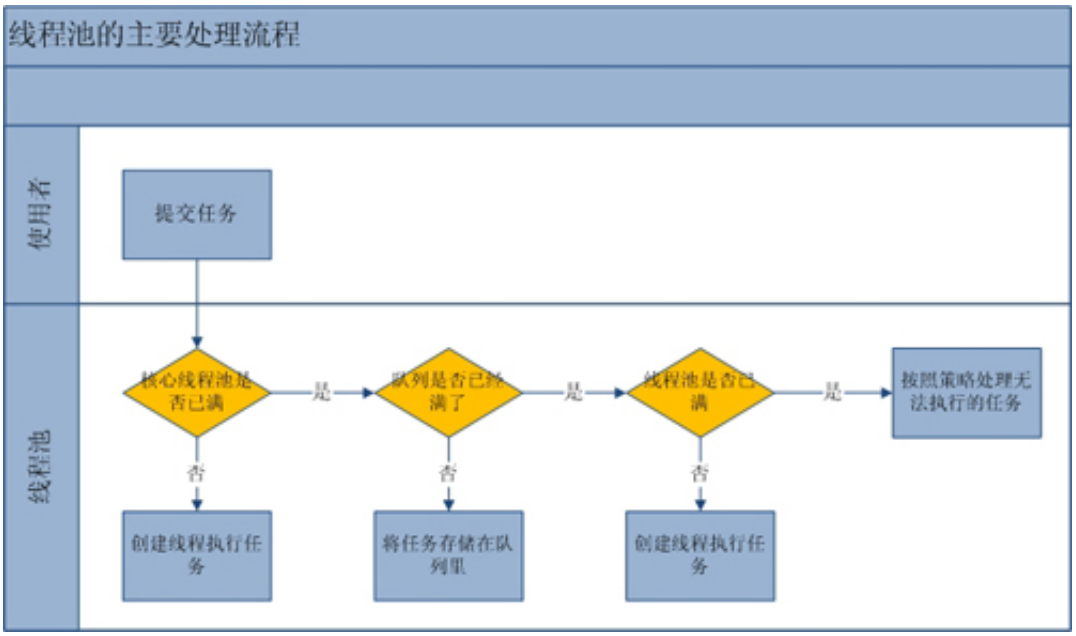
> - shutDown的原理只是将线程池的状态设置为 `SHUTDOWN` 状态，然后中断所有没有执行任务的线程。
> - shutdownNow的原理是遍历线程池中的工作线程，然后逐个调用线程的 `interrupt` 方法来中断线程，所以无法响应中断的任务可能永远无法终止。shutdownNow()会首先将线程池的状态设置 `STOP` ,然后尝试停止所有正在执行或暂停任务的线程，并返回等待执行任务的列表。

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，这是调用isTerminaed方法返回true。至于我们应该调用哪一种方法来关闭线程池，应该有提交到线程池的任务特性来决定，通常调用shutDown()来管理线程池，如果任务不一定要执行完毕，则可以调用shutdownNow()。

线程池的分析

流程分析

线程池的主要工作流程如下图：



从上图可以看出，当提交一个新任务到线程池，线程池的处理流程如下：

1. 首先线程池判断**基本线程池**是否已满？没满，创建一个工作线程来执行任务。满了，则进入下个流程。
2. 其次线程池判断**工作队列**是否已满？没满，则将新提交的任务存储在工作队列里。满了，则进入下个流程。
3. 最后线程池判断**整个线程池**是否已满？没满，则创建一个新的工作线程来执行任务，满了，则交给饱和策略来处理这个任务。

源码分析

上面的流程分析让我们很直观的了解的线程池的工作原理，让我们再通过源代码来看看是如何实现的。

```
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在执行任务后，还会无循环获取工作队列里的任务来执行。可以从worker的run方法看到这点:

```
public void run() {
     try {
           Runnable task = firstTask;
```

```
            firstTask = null;
            while (task != null || (task = getTask()) != null) {
                    runTask(task);
                    task = null;
            }
        } finally {
                workerDone(this);
        }
}
```

合理的配置线程池

要想合理的配置线程池，就必须首先分析任务特性，可以从以下几个角度来进行分析

1. 任务的性质：CPU密集型任务，IO密集型任务和混合想任务。
2. 任务的优先级：高、中和低。
3. 任务的执行时间：长、中和短。
4. 任务的依赖性：是否依赖其他系统资源，如数据连接。

任务的性质

> 任务性质不同任务可以用不同规模的线程分开处理。`CPU密集型任务`配置尽可能少的线程数量，比如配置`Ncpu+1`个线程的线程池。
> `IO密集型`任务则由于需要等待IO操作，线程不是一直执行任务，则配置可能多的线程如`2*Ncpu`。
> `混合型的任务`,如果可以拆分，则将其拆分成一个CPU密集型任务和一个IO密集型任务，只要这两个任务的时间相差不是太大，那么分解后执行的吞吐率要高于串行的吞吐率，如果这两个任务执行时间相差太大，则没必要进行分解。我们可以通过`Runtime.getRuntime().availableProcessors()`方法获得当前设备的CPU个数。

优先级不同的任务

> 优先级不同的任务可以用于优先级队列PriorityBlockingQueue来处理。它可以让优先级高的任务先得到执行，需要注意的是如果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

> 执行时间不同的队列可以交给不同规模的线程池来处理，或者也可以使用优先级队列，让执行时间短的任务先执行。

依赖

> 依赖数据连接池的任务，因为线程提交SQL后需要等待数据库返回结果，如果等待的时间越长CPU空闲时间就越长，那么线程数应该设置越大，这样才能更好的利用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，可以根据需要设置大一点，比如几千。有一次我们组使用的后台任务线程池的队列和线程池全满了，不断的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住，任务积压在线程池里。如果当时我们设置成无界队列，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的单独的服务器部署的，而我们使用不同规模的线程池跑不同类型的任务，但是出现这样问题时也会影响到其他任务。

线程池的监控

通过线程池提供的参数进行监控。

线程池里有一些属性在监控线程池的时候可以使用。

> - taskCount:线程池需要执行的任务数量。
> - 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:线程池曾经创建过最大线程数量。通过这个数据可以知道线程池是否满过。如果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，池里线程不会自动销毁，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。

> 通过集成线程池并重写线程池的 `beforeExecute` ， `afterExecute` 和 `terminated` 方法，我们可以在任务执行前，执行后和线程关闭前干一些事情。如监控任务的平均时间，最大执行时间和最小执行时间等。这借个方法在线程池里是空方法。如

```
```

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

转载自并发编程网 – ifeve.com**本文链接地址:** 聊聊并发（三）Java线程池的分析和使用