22-Executor与线程池：如何创建正确的线程池？

虽然在Java语言中创建线程看上去就像创建一个对象一样简单，只需要new Thread()就可以了，但实际上创建线程远不是创建一个对象那么简单。创建对象，仅仅是在JVM的堆里分配一块内存而已；而创建一个线程，却需要调用操作系统内核的API，然后操作系统要为线程分配一系列的资源，这个成本就很高了，所以**线程是一个重量级的对象，应该避免频繁创建和销毁**。

那如何避免呢？应对方案估计你已经知道了，那就是线程池。

线程池的需求是如此普遍，所以Java SDK并发包自然也少不了它。但是很多人在初次接触并发包里线程池相关的工具类时，多少会都有点蒙，不知道该从哪里入手，我觉得根本原因在于线程池和一般意义上的池化资源是不同的。一般意义上的池化资源，都是下面这样，当你需要资源的时候就调用acquire()方法来申请资源，用完之后就调用release()释放资源。若你带着这个固有模型来看并发包里线程池相关的工具类时，会很遗憾地发现它们完全匹配不上，Java提供的线程池里面压根就没有申请线程和释放线程的方法。

```
class XXXPool{
  // 获取池化资源
  XXX acquire() {
  }
  // 释放池化资源
  void release(XXX x){
  }
}
```

线程池是一种生产者-消费者模式

为什么线程池没有采用一般意义上池化资源的设计方法呢？如果线程池采用一般意义上池化资源的设计方法，应该是下面示例代码这样。你可以来思考一下，假设我们获取到一个空闲线程T1，然后该如何使用T1呢？你期望的可能是这样：通过调用T1的execute()方法，传入一个Runnable对象来执行具体业务逻辑，就像通过构造函数Thread(Runnable target)创建线程一样。可惜的是，你翻遍Thread对象的所有方法，都不存在类似execute(Runnable target)这样的公共方法。

```
//采用一般意义上池化资源的设计方法
class ThreadPool{
  // 获取空闲线程
  Thread acquire() {
  }
  // 释放线程
  void release(Thread t){
  }
}
//期望的使用
ThreadPool pool;
Thread T1=pool.acquire();
//传入Runnable对象
T1.execute(()->{
  //具体业务逻辑
  ......
});
```

所以，线程池的设计，没有办法直接采用一般意义上池化资源的设计方法。那线程池该如何设计呢？目前业界线程池的设计，普遍采用的都是生产者-消费者模式。线程池的使用方是生产者，线程池本身是消费者。在下面的示例代码中，我们创建了一个非常简单的线程池MyThreadPool，你可以通过它来理解线程池的工作原理。

```
//简化的线程池，仅用来说明工作原理
class MyThreadPool{
  //利用阻塞队列实现生产者-消费者模式
  BlockingQueue<Runnable> workQueue;
  //保存内部工作线程
  List<WorkerThread> threads
    = new ArrayList<>();
  // 构造方法
  MyThreadPool(int poolSize,
    BlockingQueue<Runnable> workQueue){
    this.workQueue = workQueue;
    // 创建工作线程
    for(int idx=0; idx<poolSize; idx++){
      WorkerThread work = new WorkerThread();
      work.start();
      threads.add(work);
    }
  }
  // 提交任务
  void execute(Runnable command){
    workQueue.put(command);
  }
  // 工作线程负责消费任务，并执行任务
  class WorkerThread extends Thread{
    public void run() {
      //循环取任务并执行
      while(true){ ①
        Runnable task = workQueue.take();
        task.run();
      }
    }
  }
}

/** 下面是使用示例 **/
// 创建有界阻塞队列
BlockingQueue<Runnable> workQueue =
  new LinkedBlockingQueue<>(2);
// 创建线程池
MyThreadPool pool = new MyThreadPool(
  10, workQueue);
// 提交任务
pool.execute(()->{
    System.out.println("hello");
});
```

在MyThreadPool的内部，我们维护了一个阻塞队列workQueue和一组工作线程，工作线程的个数由构造函数中的poolSize来指定。用户通过调用execute()方法来提交Runnable任务，execute()方法的内部实现仅仅是将任务加入到workQueue中。MyThreadPool内部维护的工作线程会消费workQueue中的任务并执行任务，相关的代码就是代码①处的while循环。线程池主要的工作原理就这些，是不是还挺简单的？

如何使用Java中的线程池

Java并发包里提供的线程池，远比我们上面的示例代码强大得多，当然也复杂得多。Java提供的线程池相关的工具类中，最核心的是**ThreadPoolExecutor**，通过名字你也能看出来，它强调的是Executor，而不是一般意义上的池化资源。

ThreadPoolExecutor的构造函数非常复杂，如下面代码所示，这个最完备的构造函数有7个参数。

```
ThreadPoolExecutor(
  int corePoolSize,
  int maximumPoolSize,
  long keepAliveTime,
  TimeUnit unit,
  BlockingQueue<Runnable> workQueue,
  ThreadFactory threadFactory,
  RejectedExecutionHandler handler)
```

下面我们一一介绍这些参数的意义，你可以**把线程池类比为一个项目组，而线程就是项目组的成员。**

- **corePoolSize**：表示线程池保有的最小线程数。有些项目很闲，但是也不能把人都撤了，至少要留corePoolSize个人坚守阵地。
- **maximumPoolSize**：表示线程池创建的最大线程数。当项目很忙时，就需要加人，但是也不能无限制地加，最多就加到maximumPoolSize个人。当项目闲下来时，就要撤人了，最多能撤到corePoolSize个人。
- **keepAliveTime & unit**：上面提到项目根据忙闲来增减人员，那在编程世界里，如何定义忙和闲呢？很简单，一个线程如果在一段时间内，都没有执行任务，说明很闲，keepAliveTime 和 unit 就是用来定义这个“一段时间”的参数。也就是说，如果一个线程空闲了`keepAliveTime & unit`这么久，而且线程池的线程数大于 corePoolSize ，那么这个空闲的线程就要被回收了。
- **workQueue**：工作队列，和上面示例代码的工作队列同义。
- **threadFactory**：通过这个参数你可以自定义如何创建线程，例如你可以给线程指定一个有意义的名字。
- **handler**：通过这个参数你可以自定义任务的拒绝策略。如果线程池中所有的线程都在忙碌，并且工作队列也满了（前提是工作队列是有界队列），那么此时提交任务，线程池就会拒绝接收。至于拒绝的策略，你可以通过handler这个参数来指定。ThreadPoolExecutor已经提供了以下4种策略。
  - CallerRunsPolicy：提交任务的线程自己去执行该任务。
  - AbortPolicy：默认的拒绝策略，会throws RejectedExecutionException。
  - DiscardPolicy：直接丢弃任务，没有任何异常抛出。
  - DiscardOldestPolicy：丢弃最老的任务，其实就是把最早进入工作队列的任务丢弃，然后把新任务加入到工作队列。

Java在1.6版本还增加了 allowCoreThreadTimeOut(boolean value) 方法，它可以让所有线程都支持超时，这意味着如果项目很闲，就会将项目组的成员都撤走。

使用线程池要注意些什么

考虑到ThreadPoolExecutor的构造函数实在是有些复杂，所以Java并发包里提供了一个线程池的静态工厂类

Executors，利用Executors你可以快速创建线程池。不过目前大厂的编码规范中基本上都不建议使用Executors了，所以这里我就不再花篇幅介绍了。

不建议使用Executors的最重要的原因是：Executors提供的很多方法默认使用的都是无界的LinkedBlockingQueue，高负载情境下，无界队列很容易导致OOM，而OOM会导致所有请求都无法处理，这是致命问题。所以**强烈建议使用有界队列**。

使用有界队列，当任务过多时，线程池会触发执行拒绝策略，线程池默认的拒绝策略会throw RejectedExecutionException 这是个运行时异常，对于运行时异常编译器并不强制catch它，所以开发人员很容易忽略。因此**默认拒绝策略要慎重使用**。如果线程池处理的任务非常重要，建议自定义自己的拒绝策略；并且在实际工作中，自定义的拒绝策略往往和降级策略配合使用。

使用线程池，还要注意异常处理的问题，例如通过ThreadPoolExecutor对象的execute()方法提交任务时，如果任务在执行的过程中出现运行时异常，会导致执行任务的线程终止；不过，最致命的是任务虽然异常了，但是你却获取不到任何通知，这会让你误以为任务都执行得很正常。虽然线程池提供了很多用于异常处理的方法，但是最稳妥和简单的方案还是捕获所有异常并按需处理，你可以参考下面的示例代码。

```
try {
  //业务逻辑
} catch (RuntimeException x) {
  //按需处理
} catch (Throwable x) {
  //按需处理
}
```

总结

线程池在Java并发编程领域非常重要，很多大厂的编码规范都要求必须通过线程池来管理线程。线程池和普通的池化资源有很大不同，线程池实际上是生产者-消费者模式的一种实现，理解生产者-消费者模式是理解线程池的关键所在。

创建线程池设置合适的线程数非常重要，这部分内容，你可以参考《10 | Java线程（中）：创建多少线程才是合适的？》的内容。另外《Java并发编程实战》的第7章《取消与关闭》的7.3节“处理非正常的线程终止”详细介绍了异常处理的方案，第8章《线程池的使用》对线程池的使用也有更深入的介绍，如果你感兴趣或有需要的话，建议你仔细阅读。

课后思考

使用线程池，默认情况下创建的线程名字都类似pool-1-thread-2这样，没有业务含义。而很多情况下为了便于诊断问题，都需要给线程赋予一个有意义的名字，那你知道有哪些办法可以给线程池里的线程指定名字吗？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言：

- 西西弗与卡夫卡 2019-04-18 00:16:33
  线程命名常用方法是：线程的构造函数传入名字，或者调用setName设置