二

16 如何根据实际需要，定制自己的线程池?

在本课时我们主要学习如何根据自己的实际需求设置线程池的各个参数来定制自己的线程池。

核心线程数

第一个需要设置的参数往往是 corePoolSize 核心线程数，在上一课时我们讲过，合理的线程数量和任务类型，以及 CPU 核心数都有关系，基本结论是线程的平均工作时间所占比例越高，就需要越少的线程；线程的平均等待时间所占比例越高，就需要越多的线程。而对于最大线程数而言，如果我们执行的任务类型不是固定的，比如可能一段时间是 CPU 密集型，另一段时间是 IO 密集型，或是同时有两种任务相互混搭。那么在这种情况下，我们可以把最大线程数设置成核心线程数的几倍，以便应对任务突发情况。当然更好的办法是用不同的线程池执行不同类型的任务，让任务按照类型区分开，而不是混杂在一起，这样就可以按照上一课时估算的线程数或经过压测得到的结果来设置合理的线程数了，达到更好的性能。

阻塞队列

对于阻塞队列这个参数而言，我们可以选择之前介绍过的 LinkedBlockingQueue 或者 SynchronousQueue 或者 DelayedWorkQueue，不过还有一种常用的阻塞队列叫 ArrayBlockingQueue，它也经常被用于线程池中，这种阻塞队列内部是用数组实现的，在新建对象的时候要求传入容量值，且后期不能扩容，所以 ArrayBlockingQueue 的最大的特点就是容量是有限的。这样一来，如果任务队列放满了任务，而且线程数也已经达到了最大值，线程池根据规则就会拒绝新提交的任务，这样一来就可能会产生一定的数据丢失。

但相比于无限增加任务或者线程数导致内存不足，进而导致程序崩溃，数据丢失还是要更好一些的，如果我们使用了 ArrayBlockingQueue 这种阻塞队列，再加上我们限制了最大线程数量，就可以非常有效地防止资源耗尽的情况发生。此时的队列容量大小和 maxPoolSize 是一个 trade-off，如果我们使用容量更大的队列和更小的最大线程数，就可以减少上下文切换带来的开销，但也可能因此降低整体的吞吐量；如果我们的任务是 IO 密集型，则可以选择稍小容量的队列和更大的最大线程数，这样整体的效率就会更高，不过也会带来更多的上下文切换。

线程工厂

对于线程工厂 threadFactory 这个参数，我们可以使用默认的 defaultThreadFactory，也可以传入自定义的有额外能力的线程工厂，因为我们可能有多个线程池，而不同的线程池之间有必要通过不同的名字来进行区分，所以可以传入能根据业务信息进行命名的线程工厂，以便后续可以根据线程名区分不同的业务进而快速定位问题代码。比如可以通过 com.google.common.util.concurrent.ThreadFactory

Builder 来实现，如代码所示。

```
ThreadFactoryBuilder builder = new ThreadFactoryBuilder();
ThreadFactory rpcFactory = builder.setNameFormat("rpc-pool-%d").build();
```

我们生成了名字为 rpcFactory 的 ThreadFactory，它的 nameFormat 为 "rpc-pool-%d" ，那么它生成的线程的名字是有固定格式的，它生成的线程的名字分别为"rpc-pool-1"，"rpc-pool-2" ，以此类推。

拒绝策略

最后一个参数是拒绝策略，我们可以根据业务需要，选择第 11 讲里的四种拒绝策略之一来使用：AbortPolicy，DiscardPolicy，DiscardOldestPolicy 或者 CallerRunsPolicy。除此之外，我们还可以通过实现 RejectedExecutionHandler 接口来实现自己的拒绝策略，在接口中我们需要实现 rejectedExecution 方法，在 rejectedExecution 方法中，执行例如打印日志、暂存任务、重新执行等自定义的拒绝策略，以便满足业务需求。如代码所示。

```
private static class CustomRejectionHandler implements RejectedExecutionHandler {
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        //打印日志、暂存任务、重新执行等拒绝策略
    }
}
```

总结

所以定制自己的线程池和我们的业务是强相关的，首先我们需要掌握每个参数的含义，以及常见的选项，然后根据实际需要，比如说并发量、内存大小、是否接受任务被拒绝等一系列

因素去定制一个非常适合自己业务的线程池，这样既不会导致内存不足，同时又可以用合适数量的线程来保障任务执行的效率，并在拒绝任务时有所记录方便日后进行追溯。

上一页 下一页