线程池

题目难度：★★★

知识点标签：Java多线程

学习时长：15分钟

题目描述

Java提供了哪几种线程池?

解题思路

> 需要从 什么是线程池？为什么需要线程池？线程池有哪几种已经创建方式等作答

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。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件），则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

为什么需要线程池

我们有两种常见的创建线程的方法，一种是继承Thread类，一种是实现Runnable的接口，Thread类其实也是实现了Runnable接口。但是我们创建这两种线程在运行结束后都会被虚拟机销毁，如果线程数量多的话，频繁的创建和销毁线程会大大浪费时间和效率，更重要的是浪费内存。那么有没有一种方法能让线程运行完后不立即销毁，而是让线程重复使用，继续执行其他的任务哪?

这就是线程池的由来，很好的解决线程的重复利用，避免重复开销

线程池的优点?

1）重用存在的线程，减少对象创建销毁的开销。

2）可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。

3）提供定时执行、定期执行、单线程、并发数控制等功能。

Java 提供了哪几种线程池?

Java 主要提供了下面4种线程池

- FixedThreadPool： 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。
- SingleThreadExecutor： 方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。
- CachedThreadPool： 该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新

的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

- ScheduledThreadPoolExecutor： 主要用来在给定的延迟后运行任务，或者定期执行任务。ScheduledThreadPoolExecutor又分为：ScheduledThreadPoolExecutor（包含多个线程）和SingleThreadScheduledExecutor （只包含一个线程）两种。

4种线程池各自的使用场景是什么?

- FixedThreadPool： 适用于为了满足资源管理需求，而需要限制当前线程数量的应用场景。它适用于负载比较重的服务器；
- SingleThreadExecutor： 适用于需要保证顺序地执行各个任务并且在任意时间点，不会有多个线程是活动的应用场景；
- CachedThreadPool： 适用于执行很多的短期异步任务的小程序，或者是负载较轻的服务器；
- ScheduledThreadPoolExecutor： 适用于需要多个后台执行周期任务，同时为了满足资源管理需求而需要限制后台线程的数量的应用场景；
- SingleThreadScheduledExecutor： 适用于需要单个后台线程执行周期任务，同时保证顺序地执行各个任务的应用场景。

创建线程池的方式

（1） 使用 Executors 创建

我们上面刚刚提到了 Java 提供的几种线程池，通过 Executors 工具类我们可以很轻松的创建我们上面说的几种线程池。但是实际上我们一般都不是直接使用Java提供好的线程池，另外在《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 构造函数 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险。

```
public abstract class Reader implements Readable, Closeable {
    protected Object lock;
    protected Reader() {
        this.lock = this;
    }
    protected Reader(Object lock) {
        if (lock == null) {
            throw new NullPointerException();
        }
        this.lock = lock;
    }
   //试图将字符读入指定的字符缓冲区。缓冲区可照原样用作字符的存储库：所做的唯一改变是 put 操作的结果。不对缓冲区执行翻转或重绕操作。
    public int read(java.nio.CharBuffer target) throws IOException {}
   //读取单个字符。在字符可用、发生 I/O 错误或者已到达流的末尾前，此方法一直阻塞。 用于支持高效的单字符输入的子类应重写此方法。
    public int read() throws IOException { }
    //将字符读入数组。在某个输入可用、发生 I/O 错误或者已到达流的末尾前，此方法一直阻塞。
    public int read(char cbuf[]) throws IOException {}
    // 将字符读入数组的某一部分。在某个输入可用、发生 I/O 错误或者到达流的末尾前，此方法一直阻塞。
    abstract public int read(char cbuf[], int off, int len) throws IOException;
   //跳过字符。在某个字符可用、发生 I/O 错误或者已到达流的末尾前，此方法一直阻塞。
    public long skip(long n) throws IOException {}
   //判断是否准备读取此流。
    public boolean ready() throws IOException { }
   //判断此流是否支持 mark() 操作。默认实现始终返回 false。子类应重写此方法。
    public boolean markSupported() {}
```

```
   //标记流中的当前位置。对 reset() 的后续调用将尝试将该流重新定位到此点。并不是所有的字符输入流都支持 mark() 操作。
    public void mark(int readAheadLimit) throws IOException { }
    //重置该流。如果已标记该流，则尝试在该标记处重新定位该流。如果已标记该流，则以适用于特定流的某种方式尝试重置该流，
    //例如，通过将该流重新定位到其起始点。并不是所有的字符输入流都支持 reset() 操作，有些支持 reset() 而不支持 mark()。
    public void reset() throws IOException { }
     //关闭该流并释放与之关联的所有资源。在关闭该流后，再调用 read()、ready()、mark()、reset() 或 skip() 将抛出 IOException。关闭以前关闭的流无效。
     abstract public void close() throws IOException;
}
```

（2） ThreadPoolExecutor的构造函数创建

我们可以自己直接调用 ThreadPoolExecutor 的构造函数来自己创建线程池。在创建的同时，给 BlockQueue 指定容量就可以了。示例如下：

```
private static ExecutorService executor = new ThreadPoolExecutor(13, 13,
        60L, TimeUnit.SECONDS,
        new ArrayBlockingQueue(13));
```

这种情况下，一旦提交的线程数超过当前可用线程数时，就会抛出 java.util.concurrent.RejectedExecutionException，这是因为当前线程池使用的队列是有边界队列，队列已经满了便无法继续处理新的请求。但是异常（Exception）总比发生错误（Error）要好。

（3） 使用开源类库

Hollis 大佬之前在他的文章中也提到了：“除了自己定义ThreadPoolExecutor外。还有其他方法。这个时候第一时间就应该想到开源类库，如apache和guava等。”他推荐使用guava提供的 ThreadFactoryBuilder来创建线程池。下面是参考他的代码示例：

```
public class ExecutorsDemo {

    private static ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
        .setNameFormat("demo-pool-%d").build();

    private static ExecutorService pool = new ThreadPoolExecutor(5, 200,
        0L, TimeUnit.MILLISECONDS,
        new LinkedBlockingQueue<Runnable>(1024), namedThreadFactory, new
ThreadPoolExecutor.AbortPolicy());

    public static void main(String[] args) {

        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            pool.execute(new SubThread());
        }
    }
}
```

通过上述方式创建线程时，不仅可以避免OOM的问题，还可以自定义线程名称，更加方便的出错的时候溯源。