JAVA并发编程实战

——企业级案例

讲师：尹洪亮

资深系统架构师

目录

CONTENTS

CHAPTER 01

第一章 线程池

线程池

HEADLINE HEADLINE

1.1 线程池核心原理

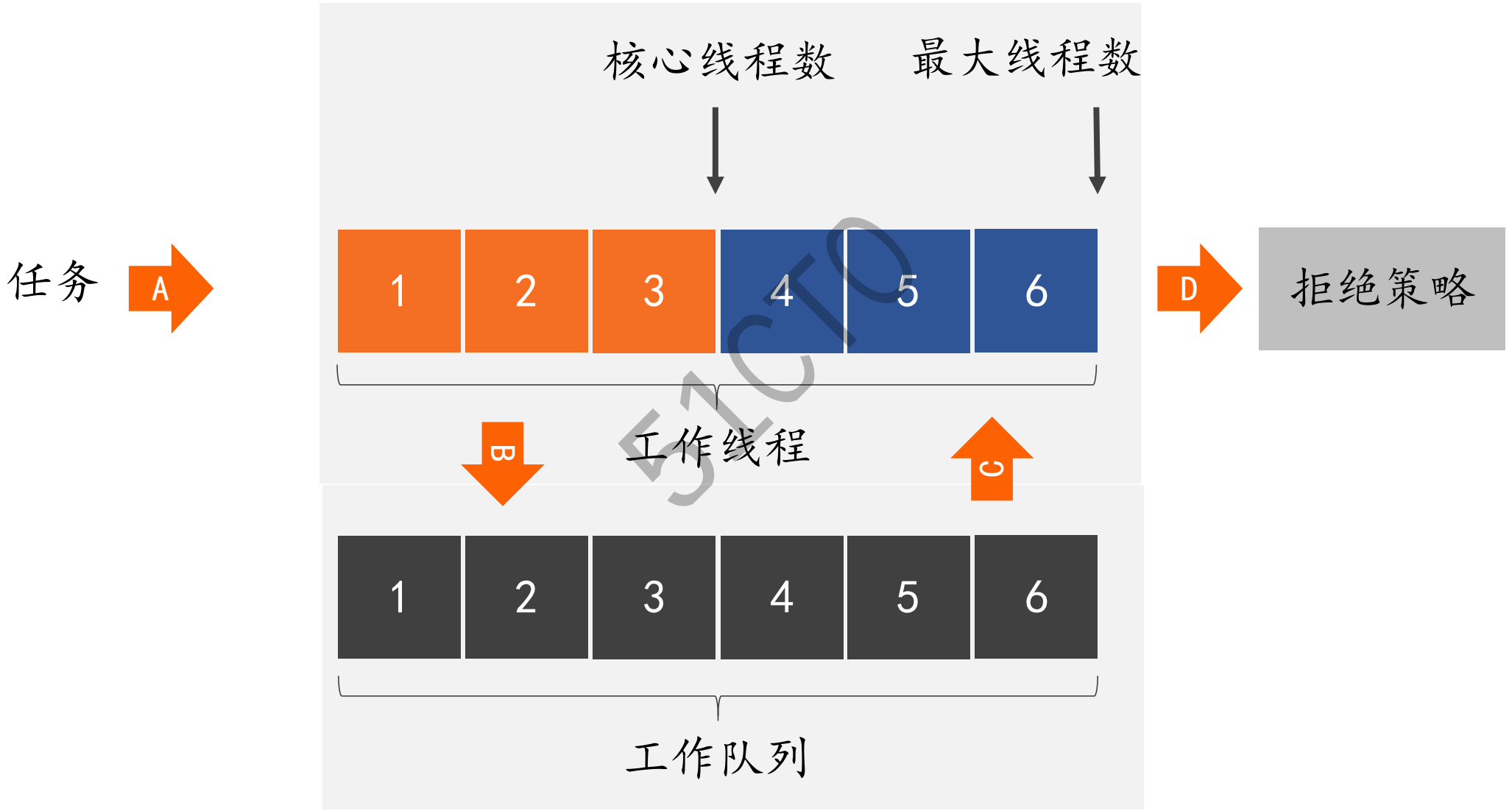
线程池核心原理

1. 线程池的核心思想
2. 线程池的原理动画
3. 线程池7大核心参数
4. 线程池7大核心参数
5. 线程池4种拒绝策略

线程池核心思想

1. 利用池化思想，管理线程的工具
2. 避免线程创建、销毁、调度的开销
3. 避免线程数量膨胀，保证对内核的充分利用

线程池核心原理



线程池7大核心参数

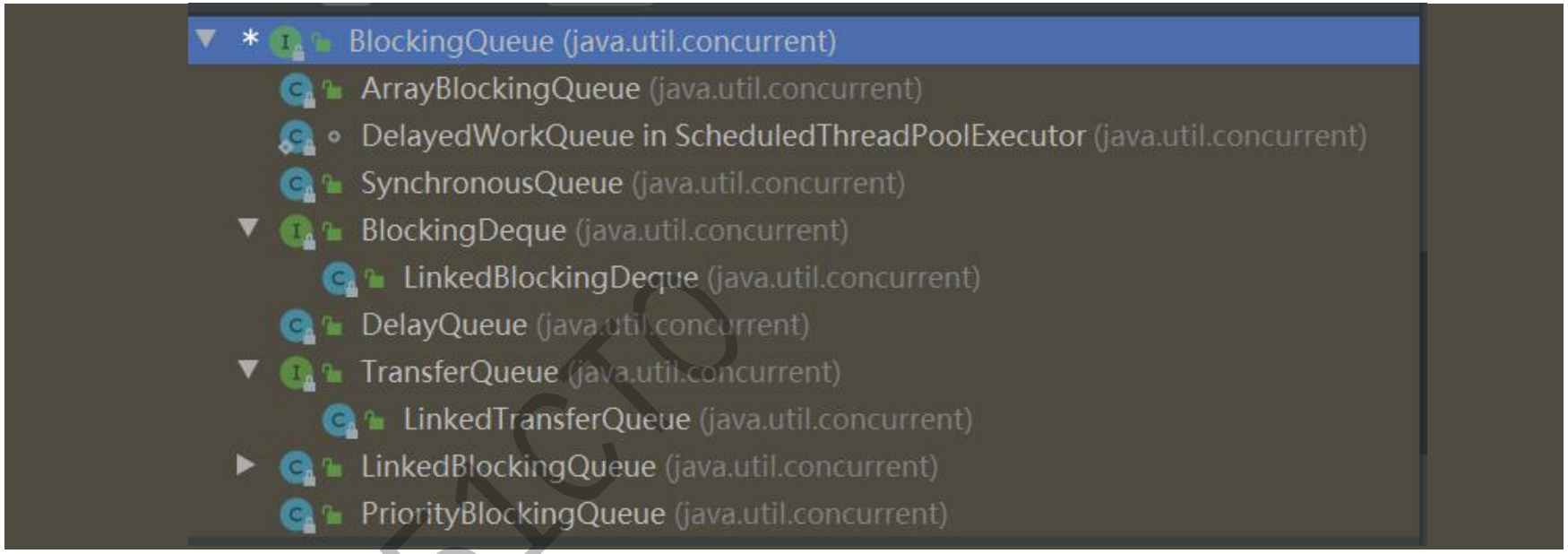
核心类 java.util.concurrent.ThreadPoolExecutor，构造函数参数如下：

```
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>)
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, ThreadFactory)
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, RejectedExecutionHandler)
ThreadPoolExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, ThreadFactory, RejectedExecutionHandler)
```

线程池的7种阻塞队列

7种

阻塞队列实现



7种

阻塞队列说明

1. ArrayBlockingQueue ： 一个由数组结构组成的有界阻塞队列
2. LinkedBlockingQueue ： 一个由链表结构组成的有界阻塞队列（常用）
3. PriorityBlockingQueue ： 一个支持优先级排序的无界阻塞队列
4. DelayQueue: 一个使用优先级队列实现的无界阻塞队列
5. SynchronousQueue: 一个不存储元素的阻塞队列（常用）
6. LinkedTransferQueue: 一个由链表结构组成的无界阻塞队列
7. LinkedBlockingDeque: 一个由链表结构组成的双向阻塞队列

线程池的4种拒绝策略

4种 拒绝策略实现

4种 拒绝策略说明

1. ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常
2. ThreadPoolExecutor.DiscardPolicy: 丢弃任务，但是不抛出异常
3. ThreadPoolExecutor.DiscardOldestPolicy: 丢弃队列最前面的任务，然后重新提交被拒绝的任务
4. ThreadPoolExecutor.CallerRunsPolicy: 由调用线程（提交任务的线程）处理该任务

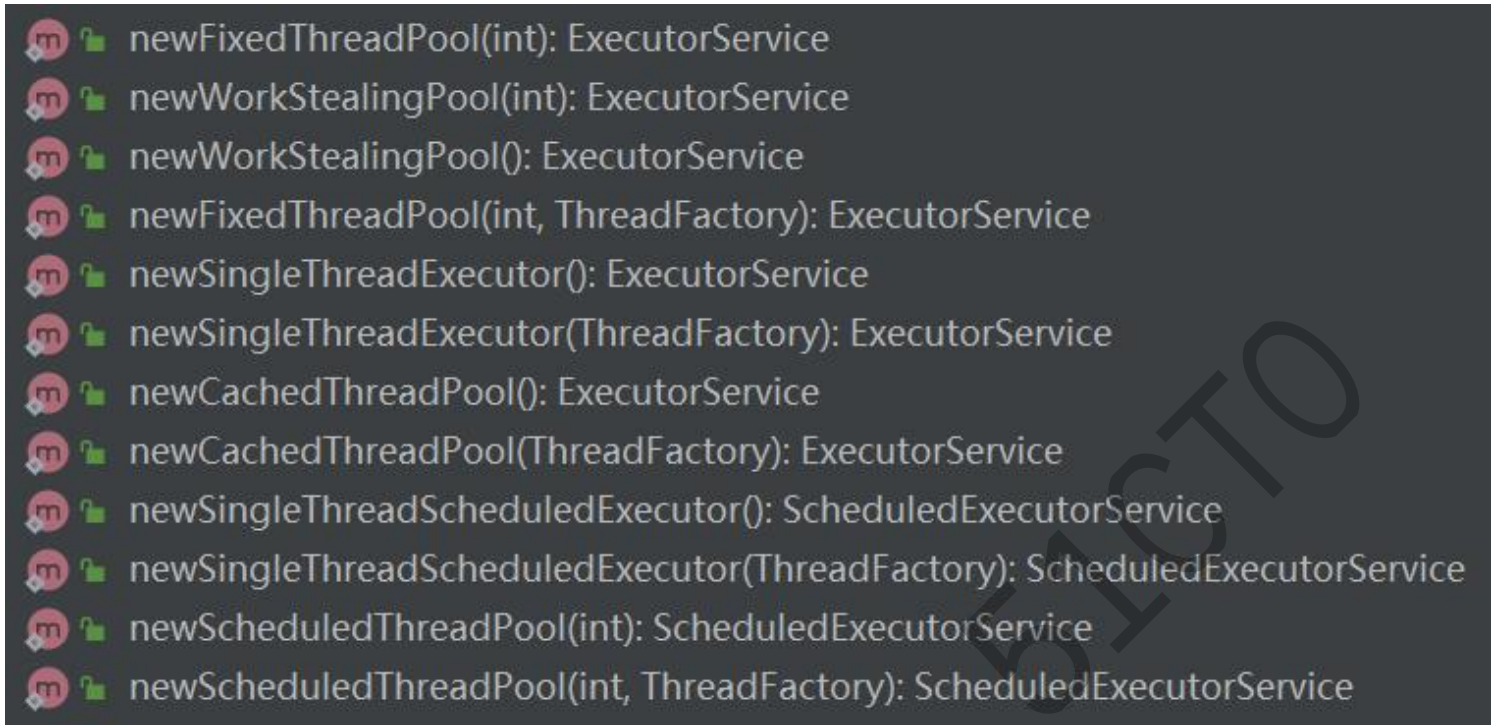
HEADLINE HEADLINE

1.2 线程池最佳使用原则

线程池最佳使用原则

1. 禁止使用Executors线程工具类
2. 线程池调优原则
3. 避免线程池死锁/饿死

禁止使用Executors线程工具类



支持快速创建7种线程池

底层都是采用ThreadPoolExecutor进行创建，只是7大参数不同而已。

线程池使用原则

禁止最大线程数使用MAX_INTEGER

禁止使用无界队列

禁用Executors各个方法的原因：

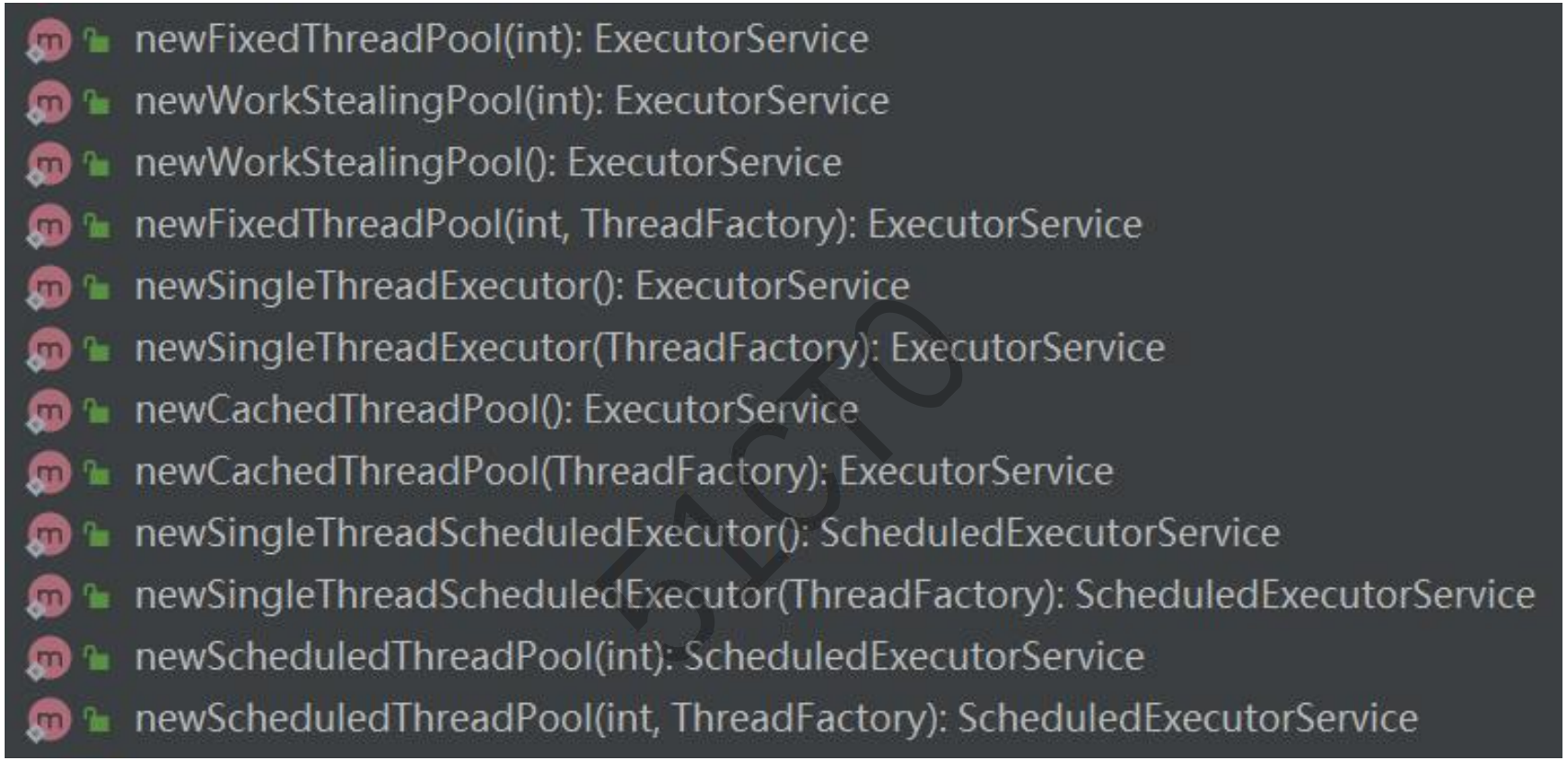
1）newFixedThreadPool和 newSingleThreadExecutor：

都使用LinkedBlockingQueue无界队列，堆积的任务处理队列会耗费非常大的内存，甚至内存溢出。

2）newCachedThreadPool和 newScheduledThreadPool：

线程数最大数都是Integer.MAX_VALUE，可能会创建数量非常多的线程，打爆CPU、甚至OOM。

禁止使用Executors线程工具类

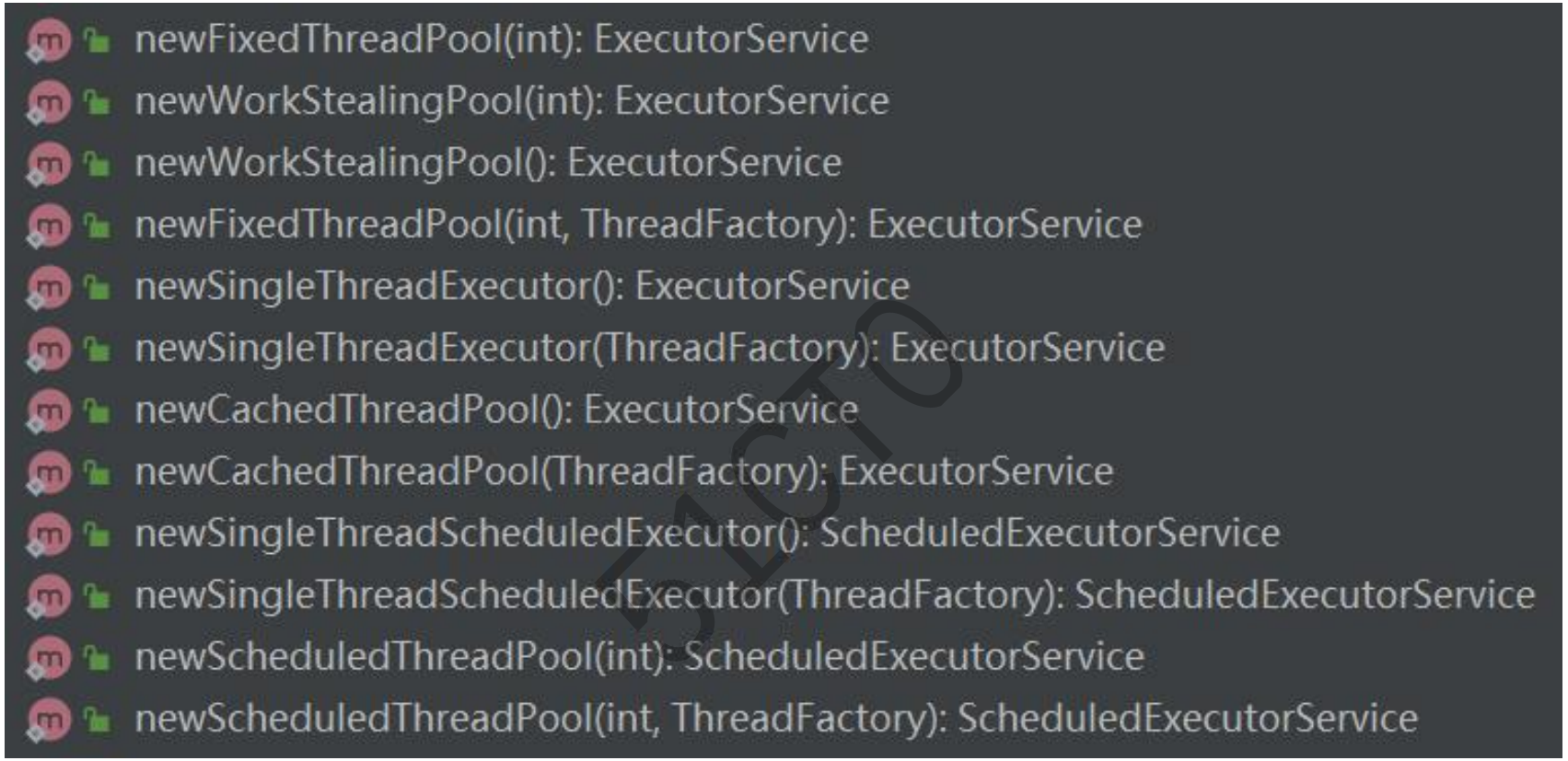


- 支持快速创建7种线程池
- 底层都是采用ThreadPoolExecutor进行创建，只是7大参数不同而已。

线程池使用原则

- 禁止最大线程数使用MAX_INTEGER
- 禁止使用无界队列

禁止使用Executors线程工具类



禁用Executors各个方法的原因：

1）newFixedThreadPool和 newSingleThreadExecutor：

都使用LinkedBlockingQueue无界队列，堆积的任务处理队列会耗费非常大的内存，甚至内存溢出。

2）newCachedThreadPool和 newScheduledThreadPool：

线程数最大数都是Integer.MAX_VALUE，可能会创建数量非常多的线程，打爆CPU、甚至OOM。

线程池调优原则

线程池的大小不能写死，而是使用Runtime.getRuntime().availableProcessors() 动态计算

快捷算法

CPU密集型应用，执行复杂算法、大量循环等 → 线程池大小=N+1(或者是N)

IO密集型应用，数据库、文件、网络读写传输等 → 线程池大小=2N+1(或者是2N)

固定任务应用、例如每次计算30个机构的收益数据。 → 线程池大小=M（固定任务数）

最佳公式

最佳线程数 = ((线程等待时间+线程CPU时间)/线程CPU时间+1) * CPU数目

额外还需要考虑：内存、文件打开数、端口数等其他资源限制

压测算法

模拟线程数被打满状态下的CPU利用率、磁盘IO、网络IO、内存等指标。

任务专用服务器，CPU、内存、IO等指标应该较高，充分利用的状态。

任务混用服务器，CPU、内存、IO等指标应适当，不能占用过多资源。

避免线程池死锁/饿死

避免线程池死锁or饿死

1. 线程池中执行单一任务，不要在任务中再提交新任务到原线程池，造成循环阻塞
2. 设置任务超时机制，任务中不要使用永久阻塞API
3. 使用ForkJoin线程池，替代普通线程池

1.3 ForkJoin 线程池

ForkJoin线程池

1. ForkJoinPool线程池是什么？
2. ForkJoin核心思想 分治算法
3. ForkJoin核心思想 窃取算法
4. Parallel Strem 并行流

ForkJoinPool线程池是什么？

1

ForkJoinPool是自java7开始，JVM提供的一个用于并行执行的任务框架。其主旨是将大任务分成若干小任务，之后再并行对这些小任务进行计算，最终汇总这些任务的结果。得到最终的结果。其广泛用在java8的Stream中。

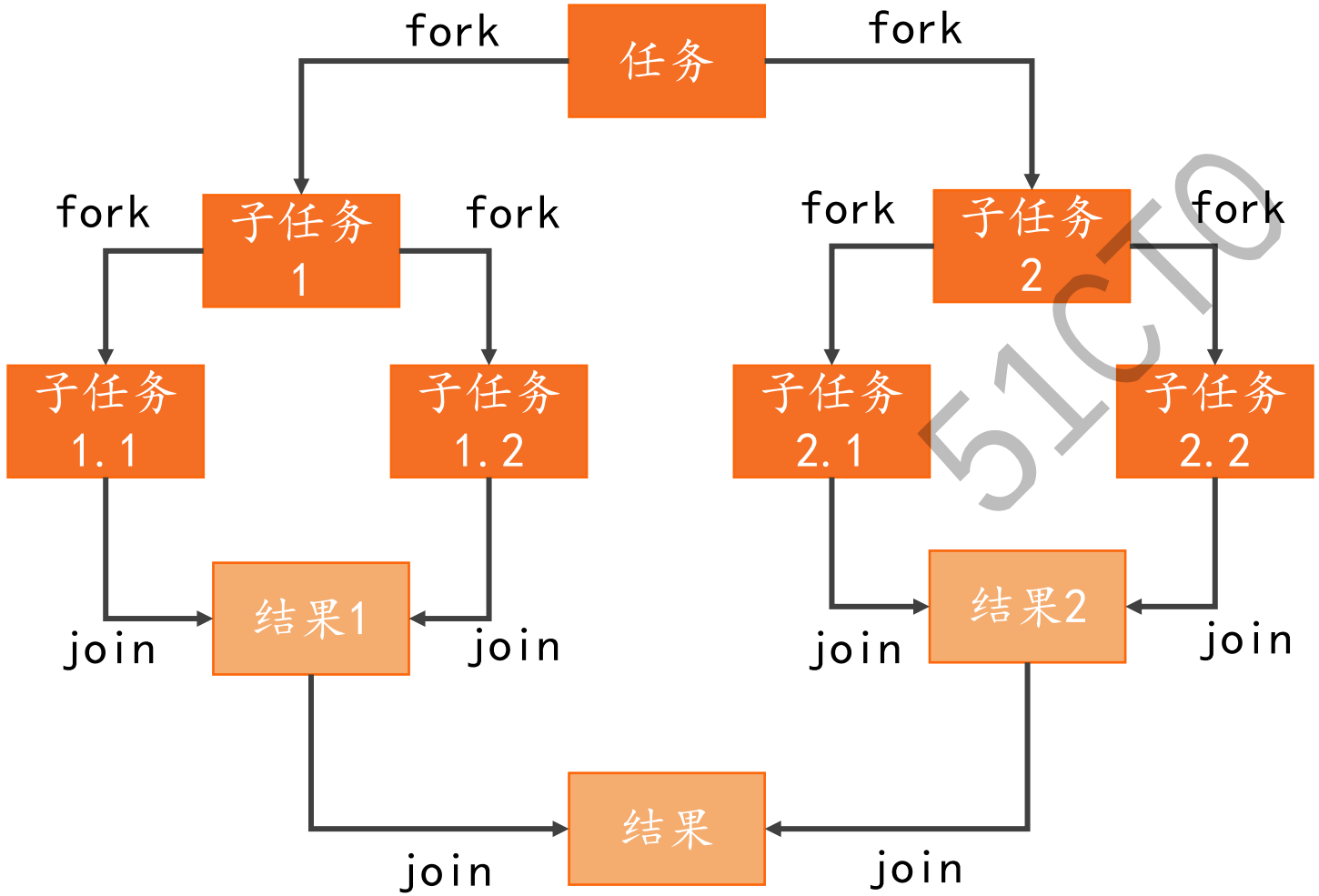
2

这个描述实际上比较接近于单机版的map-reduce。都是采用了分治算法，将大的任务拆分到可执行的任务，之后并行执行，最终合并结果集。区别就在于ForkJoin机制可能只能在单个jvm上运行，而map-reduce则是在集群上执行。

3

此外，ForkJoinPool采取工作窃取算法，以避免工作线程由于拆分了任务之后的join等待过程。这样处于空闲的工作线程将从其他工作线程的队列中主动去窃取任务来执行。这里涉及到的两个基本知识点是分治法和工作窃取。

ForkJoin核心思想 分治算法

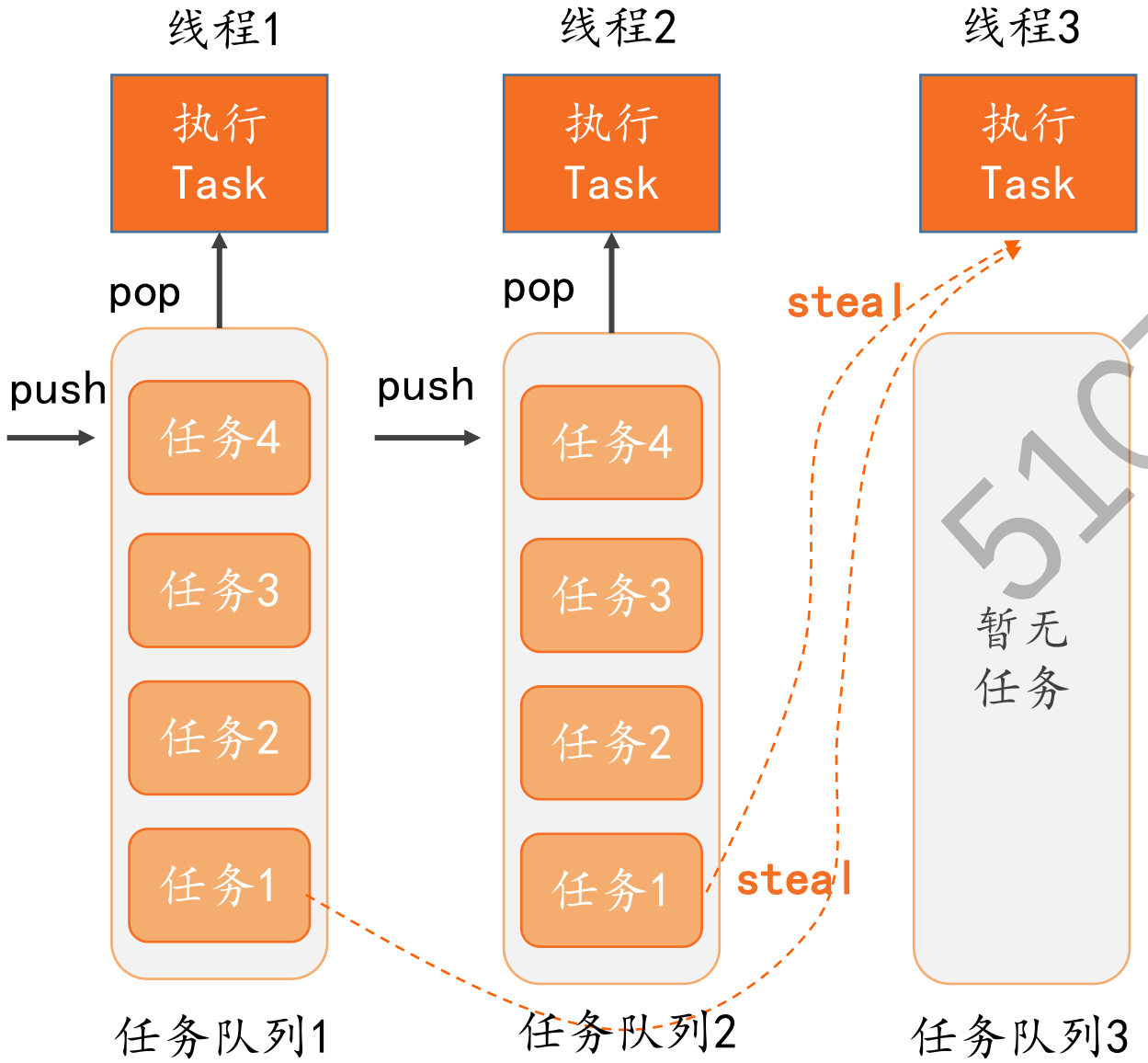


分治算法

分治法的基本思想是将一个规模为N的问题分解为K个规模较小的子问题，这些子问题的相互独立且与原问题的性质相同，求出子问题的解之后，将这些解合并，就可以得到原有问题的解。是一种分目标完成的程序算法。

将一个大的任务，通过fork方法不断拆解，直到能够计算为止，之后，再将这些结果用join合并。这样逐次递归，就得到了我们想要的结果。这就是再ForkJoinPool中的分治法。

ForkJoin核心思想 窃取算法



窃取算法

当某个线程的任务队列中没有可执行任务的时候，从其他线程的任务队列中窃取任务来执行，以充分利用工作线程的计算能力，减少线程由于获取不到任务而造成的空闲浪费。

在ForkJoinpool中，工作任务的队列都采用双端队列Deque容器。通常使用队列的过程中，我们都在队尾插入，而在队头消费以实现FIFO。而为了实现工作窃取。一般我们会改成工作线程在工作队列上LIFO,而窃取其他线程的任务的时候，从队列头部取获取。

企业应用 Parallel Stream

要点

Stream不支持并行，而Parallel Stream支持并行，编写简洁，底层使用ForkJoinPool线程池，但线程不安全，因此任务逻辑中一定要使用Atomic类、ConcurrentHashMap等并非类容器。

用法

用法1：集合.stream().parallel()

用法2：集合.parallelStream()

企业实战

企业中，对于信息查询类接口应用较多，例如某个页面要展示产品信息列表。包括产品名称、图片、库存、价格等信息。但是产品的图片来源于产品服务，库存来源于库存服务，价格来源于价格服务，还有部分信息来源于自己的数据库。这个时候就需要并行的去调用远程服务，组装为最终的结果。并行流由于其简单方便的特点，在企业中较为广泛的使用。