参考

Java中线程池，你真的会用吗？

深入理解线程池及相关面试题

线程池创建之后，会立即创建核心线程吗

1、线程池引入

所谓线程池，通俗来讲，就是一个管理线程的池子。它可以容纳多个线程，其中的线程可以反复利用，省去了频繁创建线程对象的操作。

在 Java 并发编程框架中的线程池是运用场景最多的技术，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来至少以下4个好处：

- **降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗；
- **提高响应速度**。当任务到达时，任务可以不需要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，如果无限制地创建，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分配、调优和监控。
- **提供更强大的功能**，比如延时定时线程池；

2、Executors

2.1、概述

Executors 是一个Java中的工具类。提供工厂方法来创建不同类型的线程池。

核心概念：这四个线程池的本质都是ThreadPoolExecutor对象：

- `newFiexedThreadPool(int Threads)`：创建固定数目线程的线程池。

- `newCachedThreadPool()`：创建一个可缓存的线程池，调用execute 将重用以前构造的线程（如果线程可用）。如果没有可用的线程，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。
- `newSingleThreadExecutor()`：创建一个单线程化的Executor。
- `newScheduledThreadPool(int corePoolSize)`：创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代Timer类。

2.2、Executors缺陷

在阿里巴巴Java开发手册中明确指出，**不允许使用Executors创建线程池**，这是因为使用Executors创建线程池可能会导致**OOM**(OutOfMemory ，内存溢出)。

> 4.【强制】线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 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。
>
> 说明：Executors 返回的线程池对象的弊端如下：
>
> 1）FixedThreadPool 和 SingleThreadPool:
>
> 允许的请求队列长度为 Integer.MAX_VALUE，可能会堆积大量的请求，从而导致 OOM。
>
> 2）CachedThreadPool 和 ScheduledThreadPool:
>
> 允许的创建线程数量为 Integer.MAX_VALUE，可能会创建大量的线程，从而导致 OOM。

3、优雅的创建线程池

3.1、正确挑选方法

避免使用Executors创建线程池，主要是避免使用其中的默认实现，那么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。

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

上面放出来的是**ThreadPoolExecutor**的全参构造函数，其中的参数分别为：

- `corePoolSize`：线程池中的核心线程数。指定线程数回一直存在与线程池中，除非设置了 **allowCoreThreadTimeOut**参数。当创建完成之后就会准备好等待接收异步任务去

执行；

- `maximumPoolSize`：最大线程数。当请求的线程超过最大线程数时，将会扩充线程数量到最大线程数，但不会无限扩充，达到控制资源的效果；
- `keepAliveTime`：非核心线程的存活时间。如果当前存活的线程数量大于核心线程数**corePoolSize**，则会释放空闲的线程直到线程数回到最大线程数**corePoolSize**；
- `unit`：keepAliveTime 参数的时间单位，如**TimeUnit.SECONDS**；
- `workQueue`：阻塞队列。用于保存多余的任务，如果任务很多，就会将多的任务存放进队列中，只要有空闲的线程就会去队列中取出新的任务执行直到队列为空；
- `threadFactory`：线程池工厂，标识线程，即为线程起一个具有意义的名称，可自定义；
- `handler`：拒绝策略。如果阻塞队列满了，就会按照我们指定的拒绝策略拒绝后续任务，默认为丢弃任务。

> 🌈
>
> 在线程创建过程中有一个细节，即创建阻塞队列时，队列默认的最大值为Integer的最大值，这很显然是不合理的，容易内存不够造成oom，因此一般都需要在创建时设定容量，如 `new LinkedBlockingDeque<>(1000)`

3.2、线程池配置类

在开发过程中一般会将线程池的创建抽取成一个配置类，其中的各类参数则会配置在配置文件中去。

这里有个细节，就是创建线程池的时候并不会立马准备好 `corePoolSize` 数量的线程来准备接收任务，而是要等到有任务提交时才会启动。

这一部分在下面的**4、线程池执行流程/线程池创建**中也有提及，这里使用了 `prestartCoreThread` 方法在初始化线程池的时候开启一个核心线程，避免在执行异步操作的时候初始化核心线程耗时巨大（可自行尝试，在后面的例子中因为加上了这一方法，接口的**耗时减少了50倍**）

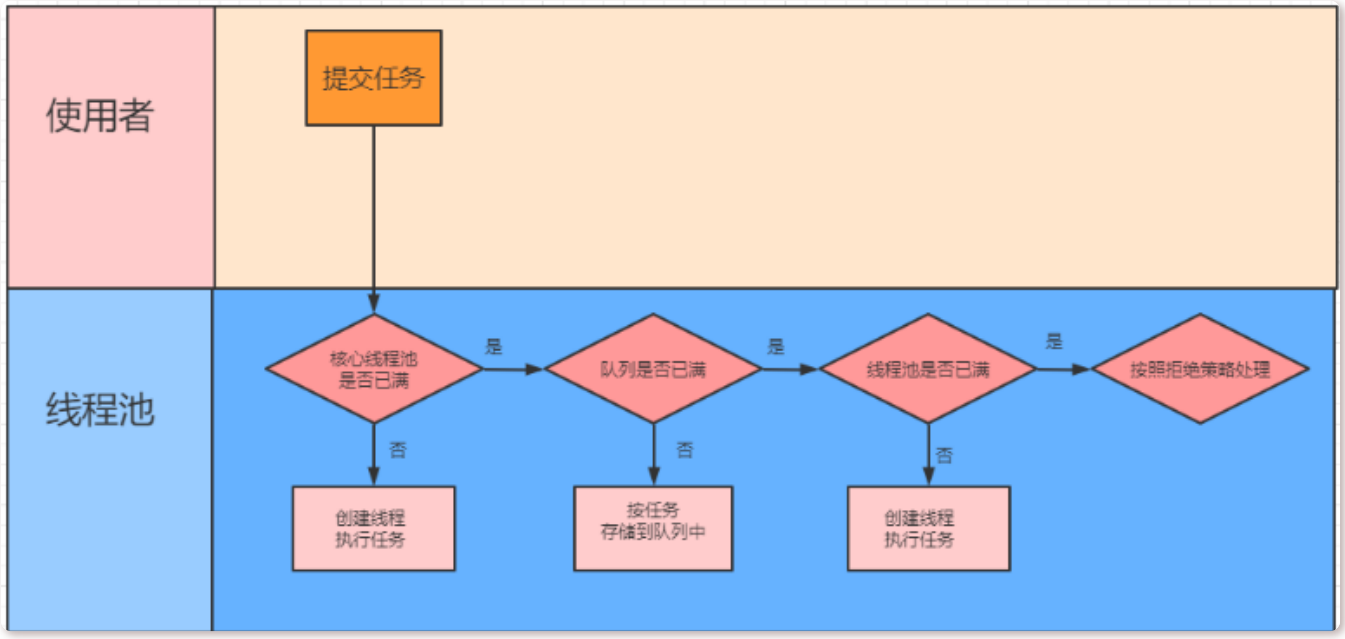
```
@Configuration
public class ThreadPoolConfig {

    @Bean
    public ThreadPoolExecutor threadPoolExecutor(
            @Value("${thread.pool.coreSize}") Integer coreSize,
            @Value("${thread.pool.maxSize}") Integer maxSize,
            @Value("${thread.pool.keepalive}") Integer keepalive,
            @Value("${thread.pool.blockQueueSize}") Integer
blockQueueSize
    ) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                coreSize,
                maxSize,
```

```
                keepalive,
                TimeUnit.SECONDS,
                new ArrayBlockingQueue<>(blockQueueSize),
                Executors.defaultThreadFactory(),
                new ThreadPoolExecutor.AbortPolicy()
        );
        executor.prestartCoreThread();
        return executor;
    }
}
```

4、线程池执行流程

当向线程池提交一个任务之后，线程池是如何处理这个任务的呢？下面就先来看一下它的主要处理流程。



下面详细介绍线程池的详细运行流程：

1. **线程池创建**，但是并不会立马准备好 `corePoolSize` 数量的线程来准备接收任务，线程并不会立即启动，而是要等到有任务提交时才会启动。除非调用了 `prestartCoreThread/prestartAllCoreThreads` 事先启动核心线程：

   1.1. `prestartCoreThread`: Starts a core thread, causing it to idly wait for work. This overrides the default policy of starting core threads only when new t asks are executed;

   1.2. `prestartAllCoreThreads`: Starts all core threads。

2. **任务到来**，用准备好的 `corePoolSize` 个空闲线程执行：

   2.1. **核心线程数已满**，就将再进来的任务放入阻塞队列中，期间如果运行中的线程小于核心线程数时，就会去阻塞队列中获取任务执行；

2.2. **阻塞队列已满**，就会创建新线程去执行阻塞队列中的任务，但最大只能创建到最大线程数 `maximumPoolSize`；

2.3. **存活且运行的线程数达到最大线程数 `maximumPoolSize`**，即线程已满时，根据设定的拒绝策略 `handler` 来对后来任务进行相应处理；

2.4. **当所有线程都执行完**，在指定时间 `keepAliveTime` 之后，将会自动销毁线程，最终保持在 `corePoolSize` 大小。

3. **在线程创建过程中**，所有的线程都由指定的工厂 `threadFactory` 进行创建，并为线程设置标识，即起名。

> **线程池场景模拟：**
>
> 一个线程池**corePoolSize**为7，**maximumPoolSize**为20，阻塞队列最大50，100并发进来怎么分配的？
>
> 答案：先有7个线程能够直接处理7个任务，接下来50个进入队列排队，再多开13个继续执行。此时所有线程池和阻塞队列都已满，但只有70个被安排上，剩下的30个走设定好的拒绝策略进行相对应操作。

最终以一张图来总结和概括下线程池的执行示意图：