> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

《手册》第 14 页有关于线程池的论述：

> 【强制】创建线程或线程池时请指定有意义的线程名称，方便出错时回溯。
>
> 【强制】线程资源必须通过线程池提供，不允许在应用中自行显式创建线程
>
> 【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。

看到这些规定我们可以思考下面几个问题：

- 那么为何会有这样的规定呢？
- 线程池那么重要，我们该如何学习线程池？

这些都是本节所要解决的问题。

那么第一个问题：**为什么要指定有意义的线程名称呢？**

《手册》给出的解释是 “方便出错时回溯”。

如果大家还没啥体会的话， 可以对比一下下面通过 `jstack` 看到的线程片段：

默认命名：

```
"pool-1-thread-1" #11 prio=5 os_prio=31 tid=0x00007fa0964c7000 nid=0x4403 waiting
on condition [0x000070000db67000]

...

at java.lang.Thread.run(Thread.java:748)
```

自定义命名：

```
"定时短息任务线程 thread-2" #11 prio=5 os_prio=31 tid=0x00007fa0964c7000 nid=0x4403
waiting on condition [0x000070000db67000]
...
at java.lang.Thread.run(Thread.java:748)
```

反差是不是很明显呢？

通过自定义名称，我们可以快速理解所关注的线程所属的线程池，对一些问题可以快速作出预判。

如何实现呢？

很多人总是先直接百度，直接查资料，虽然便捷，但是容易浅尝辄止，学啥都不深入，离开了资料就束手无策。

显然这不是我们想要的，那么怎么办呢？

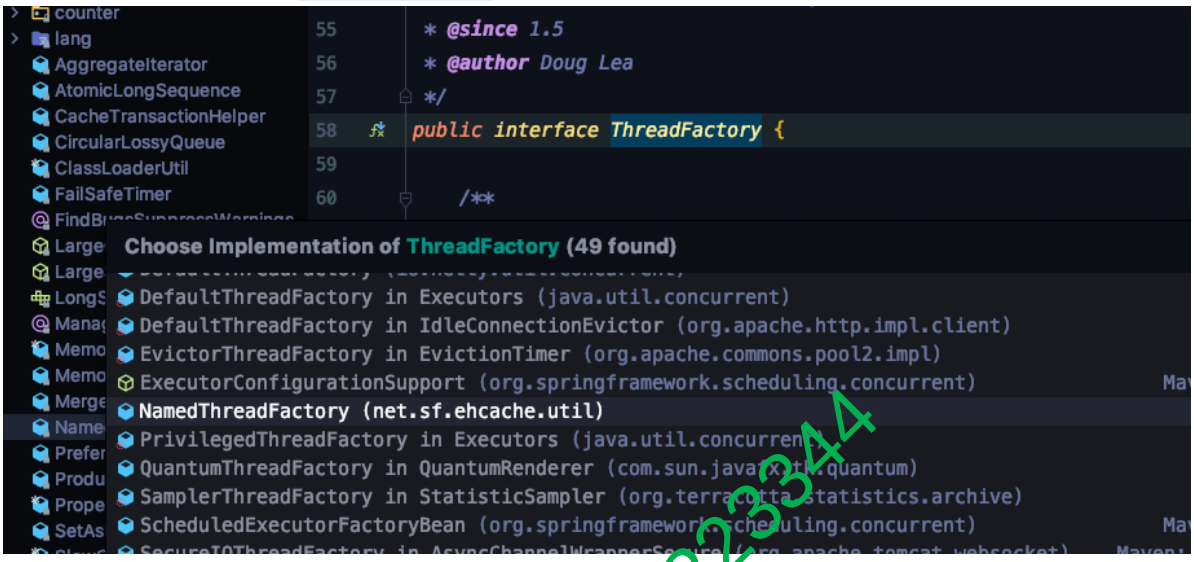
我们可以去 `ThreadPoolExecutor` 的构造方法中寻找答案，构造函数中有一个 `threadFactory` 参数，通过常识或者其注释我们可以知道该参数是为线程池构造线程。

它的类型为：`java.util.concurrent.ThreadFactory`，按照惯例，我们查看源码：

```
Thread newThread(Runnable r);
```

通过注释我们可以知道，重写此函数可以指定线程的优先级，设置是否是守护线程、设置线程的线程组等。

那么我们如何找到自定义 `ThreadFactory` 的参考范例呢?



大家可以通过点击左侧的 f 标志或使用快捷键查看实现类，进行学习。

具体写法我们可以参考：`net.sf.ehcache.util.NamedThreadFactory`

```
public class NamedThreadFactory implements ThreadFactory {

    private static AtomicInteger threadNumber = new AtomicInteger(1);
    private final String namePrefix;
    private final boolean daemon;


    public NamedThreadFactory(String namePrefix, boolean daemon) {
        this.namePrefix = namePrefix;
        this.daemon = daemon;

    }

    public NamedThreadFactory(String namePrefix) {
        this(namePrefix, false);
    }


    public Thread newThread(Runnable runnable) {
        final Thread thread = new Thread(runnable, namePrefix + " thread-" +
threadNumber.getAndIncrement());
        thread.setDaemon(daemon);
        return thread;
    }

}
```

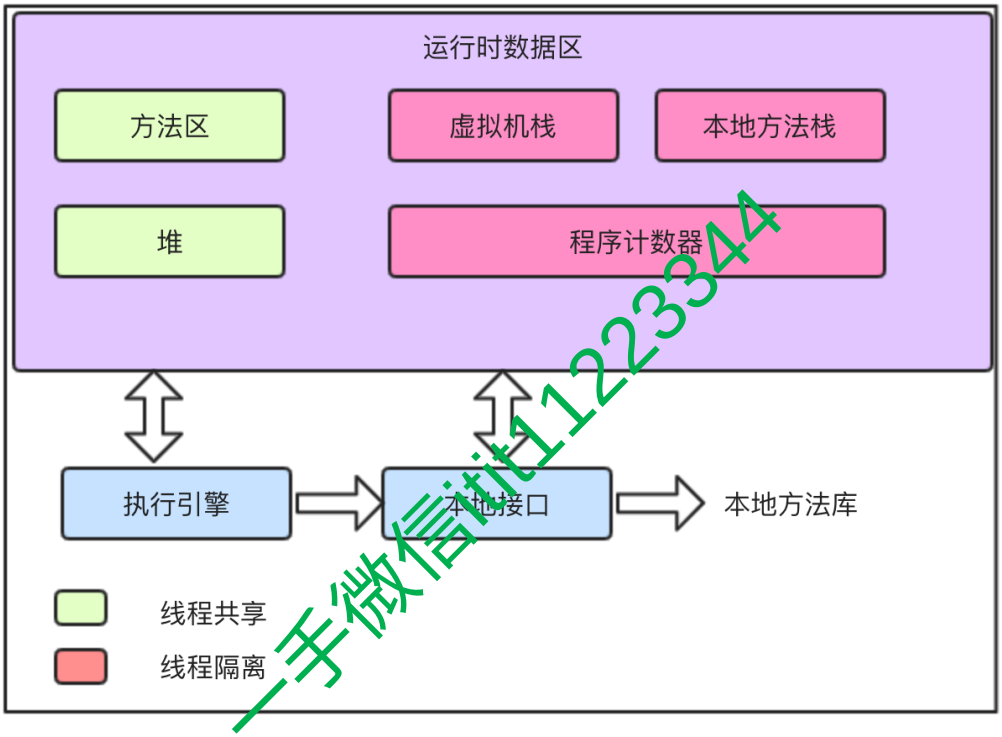
大家可以参考这个类进行改编。

另外，建议大家工作中如果不忙的时候要主动地去源码中看一看，看一些 JDK 源码中的接口有哪些实现类，它们的代码都是如何编写的，这对我们学习进阶有很大帮助。

这里要先讲一个设计模式："对象池模式"， 参见 《Java 设计模式及实践》 34 页：

> 对象的实例化是最耗费性能的操作之一，这在过去是一个大问题，现在已经不需要再那么关注。
>
> 但当我们处理封装外部资源的对象（例如数据库连接）时，对象的创建会耗费很多资源。
>
> 解决方案就是重用和共享这些创建成本昂贵的对象，这被称为对象池模式。

而根据《 Java 虚拟机规范 (Java SE 8 版)》第 9 - 15 页 描述，以及《深入理解 Java 虚拟机：JVM 高级特性与最佳实践》第 39 页 相关描述可知：



线程的创建需要开辟虚拟机栈、本地方法栈、程序计数器等线程私有的内存空间。

线程销毁时也会回收这些系统资源，因此如果频繁创建和销毁线程将大量消耗系统资源。

从上述特点我们可以看出，该场景非常符合对象池设计模式，其**核心目的是复用资源消耗加大的对象**。

> 建议大家学习设计模式时，着重了解设计模式的常见使用场景，优势和劣势，而不是着急跟着书上敲代码。
>
> 这样才能在看到对应的源码时能够 "恍然大悟"，遇到使用的场景时才能够想到要用这种设计模式。

另外，既然不提倡某种用法而提倡另外一种用法 / 技术，我们要着重思考另外一种用法 / 技术的优势。

不提倡手动创建线程的另外一个原因是线程池自身的优点，使用线程池有利于控制最大并发数，可以实现任务队列的缓存和拒绝策略，实现定时和周期执行任务，可以更好地隔离不同的场景等。

推荐通过源码和写 DEMO 来学习线程池。

那么为什么推荐这种学习方式呢?

这是因为：

1. 源码最权威，通过读源码印象更深刻，面试时或者使用时更有底气。

2. 写 DEMO 能够构造更多场景，我们可以通过运行看结果，通过各种调试技巧等方式验证自己的想法。

另外大家如果细心，可以看到很多人用过线程池，但是面试时或者和别人交流时迷迷糊糊。

为什么呢?

其实，这是因为很多人都是通过读书来记住线程池的一些参数和用法，而不是通过读源码和练习来学习的，导致印象不深刻，回答问题时没底气，没把握。

接下来我们介绍一下这两种不错的方式在线程池学习中的运用。

我们先从 `java.util.concurrent.ThreadPoolExecutor` 的构造函数说起。

前面注释章节讲过 JDK 的注释是我们学习的典范。**我们看源码时注释是帮助我们理解的一大突破口。**

如果不看书，我们如何更准确地理解参数含义呢？如何避免被一些博客误导呢?

我们应该先从核心参数的名称对参数的含义有一个大概的了解，然后看再看线程池的核心函数的逻辑。

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
    if (corePoolSize < 0 ||
        maximumPoolSize <= 0 ||
        maximumPoolSize < corePoolSize ||
        keepAliveTime < 0)
        throw new IllegalArgumentException();
    if (workQueue == null || threadFactory == null || handler == null)
        throw new NullPointerException();
    this.acc = System.getSecurityManager() == null ?
            null :
            AccessController.getContext();
    this.corePoolSize = corePoolSize;
    this.maximumPoolSize = maximumPoolSize;
    this.workQueue = workQueue;
    this.keepAliveTime = unit.toNanos(keepAliveTime);
    this.threadFactory = threadFactory;
    this.handler = handler;
}
```


通过注释我们可以清晰地知道每个参数的含义。

- `corePoolSize` 表示核心常驻线程池。即使空闲也会在线程池中保活，除非设置了允许核心线程池超时;
- `maximumPoolSize` 表示线程池同时执行的最大线程数量;
- `keepAliveTime` 表示线程池中的线程空闲时间，线程在销毁前等待新任务的最大时限;
- `unit` 表示 `keepAliveTime` 的单位;
- `workQueue` 存放执行前的任务。只会存放通过 `execute` 函数提交的 `Runnable` 任务;
- `threadFactory` 创建新线程的工厂;
- `handler` 线程超限且队列容量也达到最大值时执行受阻的处理策略。

注释中还给出了抛出异常的条件，大家可以自行学习。

接下来我们查看其核心函数之一的 `execute` 源码：

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();

    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    else if (!addWorker(command, false))
        reject(command);
}
```

通过注释我们可以知道，该函数的作用：

> 在未来的某个时刻执行给定的任务。该任务可能会被新创建的线程执行，也可能会被线程池中已经存在的线程执行。
>
> 如果任务因为 executor 被关闭 (shutdown) 或者容量达到上限而不能再提交执行时，会调用当前设置的 `RejectedExecutionHandler`。

另外源码中关于执行步骤的注释是我们理解线程池的关键：

> `execute` 分为 3 个处理步骤：
>
> 1、如果线程池中小于 `corePoolSize` 个执行的线程，则新建线程将当前任务作为第一个任务来执行；
>
> 2、如果任务成功入队，我们仍然需要 double-check 判断是否需要往线程池中新增线程（因为上次检查后可能有一个已经存在的线程挂了）或者进入这段函数时线程池关闭了；
>
> 3、如果不能入队，则创建一个新线程。如果失败，我们就知道线程池已经被关闭或已经饱和就需要调用拒绝策略来拒绝当前任务。

读完注释，哪怕我们不读代码或者读不懂源码，我们对线程池的理解也会较为深入的理解，读完注释后再读代码就会发现容易了很多。

我们再学习 `java.util.concurrent.ThreadPoolExecutor#shutdown` 函数：

```
    public void shutdown() {
        final ReentrantLock mainLock = this.mainLock;
        mainLock.lock();
        try {
            checkShutdownAccess();
            advanceRunState(SHUTDOWN);
            interruptIdleWorkers();
```

```
            onShutdown();
        } finally {
            mainLock.unlock();
        }
        tryTerminate();
    }
```

根据注释我们可知：

> 已经提交的任务执行完后关闭，此时不会不再接收新的任务。
>
> 如果已经关闭，那么调用此函数没啥副作用。
>
> 此函数不会等待已经提交的任务执行完成（才返回）。如果需要可以使用 `java.util.concurrent.ThreadPoolExecutor#awaitTermination`。

假如我们对这里关键的一句话：“This method does not wait for previously submitted tasks to complete execution.” 很困惑，可以通过 StackOverFlow 搜索相关关键词来寻找解答。

> The point is that the `shutDown` method *returns* without waiting for the previously submitted tasks to complete, but it still *lets* them complete. You might want to think of it as a “start shutting down” method.
>
> `shutDown` 不会等待直接提交的任务执行完成（但是会让它们执行完毕）就会返回。你可以将该方法理解为 “开始关闭” 函数。

线程池还有其它核心函数，需要大家自己去学习，这里就不作展开。

上面讲述了线程池的核心参数和核心函数。

那么我们来看另外一个问题，**为啥《手册》不建议用 `Executors` 来创建线程池?**

我们以 `FixedThreadPool` 为例，来分析具体原因：

```
public static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory
threadFactory) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>(),
                                  threadFactory);
}
```

通过工作上面的学习我们知道，工作队列是用来存放线程执行前的任务。

通过上面源码我们可以看出 `FixedThreadPool` 的核心线程数和最大线程数相等，而工作队列为 `java.util.concurrent.LinkedBlockingQueue`。

通过其默认构造方法，我们可以看出其容量为整数的最大值。

```
public LinkedBlockingQueue() {
    this(Integer.MAX_VALUE);
}
```

根据前面学到的知识，我们试想一下这样的场景：

> 如果对该线程池的请求不断增多，达到核心线程数后，任务将暂存到该工作队列。但是这个阻塞队列是 “无界” 的，如果大量任务过来，工作队列可能还没达到整数最大值可能就已经 OOM 了。

如果我们自定义线程池对象，可以设置相对可控的最大线程数和可控的工作队列长度以及拒绝策略。那么即使任务大量堆积，在 OOM 之前就进入了拒绝策略。

总之通过自定义线程池参数，线程池的可控性更强。

前面讲到 `java.util.concurrent.ThreadPoolExecutor#shutdown` 的功能，那么如何验证该函数的效果呢?

我们可以通过下面的例子来学习：

```
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

@Slf4j
public class ThreadPoolShutDownDemo {

    public static void main(String[] args) throws InterruptedException {
        ThreadPoolExecutor executorService = new ThreadPoolExecutor(10, 10,
                0L, TimeUnit.MILLISECONDS,
                new LinkedBlockingQueue<>(50000), new
NamedThreadFactory("shutdown-demo"));

        int total = 20000;
        for (int i = 0; i < total; i++) {
            executorService.submit(() -> {
                try {
                    TimeUnit.MILLISECONDS.sleep(5L);
                } catch (InterruptedException ignore) {
                }

            });
        }



        printExecutorInfo(total, executorService);


        executorService.shutdown();
```

```
        while (!executorService.isTerminated()) {
            TimeUnit.SECONDS.sleep(1);
            printExecutorInfo(total, executorService);
        }
    }


    private static void printExecutorInfo(int total, ThreadPoolExecutor
executorService) {
        String dateString =
DateUtil.toDateString(LocalDateTime.now(ZoneId.systemDefault()));
        log.debug("时间:{},总任务数: {}, 工作队列中有:{}个任务，已完成:{}个任务，正在执行:
{}个任务", dateString, total, executorService.getQueue().size(),
executorService.getCompletedTaskCount(), executorService.getActiveCount());
    }
}
```

执行效果如下：

> 时间：2019-08-24 20:58:50，总任务数：20000，工作队列中有：19900 个任务，已完成：90 个任务，正在执行：10 个任务
>
> ...
>
> 时间：2019-08-24 20:59:02，总任务数：20000，工作队列中有：0 个任务，已完成：20000 个任务，正在执行：0 个任务

线程池没结束，每隔一秒打印一次线程池的任务信息。

从此示例中可以清楚地观察到调用 `executorService.shutdown()` 后，已经提交的任务仍然会被执行。

大家可以打开第 1 处代码，观察执行 `ThreadPoolExecutor#executorService.shutdownNow` 后如果继续提交任务将抛出 `RejectedExecutionException`。

如果需要学习其他特性，大家都可以写一些简单的 DEMO，也可以断点调试观察更多细节。

本节，我们再次使用源码法、StackOverFlow 大法、写 DEMO 法来学习线程池的一些知识点，包括线程池的核心参数，线程池的核心函数的源码和用法。

当然，大家还可以尝试断点调试法来进入核心函数来学习执行流程等。

下一节我们将带着大家深入研究：为何 JUnit 单元测试 “不支持多线程” 的问题。

1. 请大家根据本节学的内容，分析为什么不推荐使用 CachedThreadPool 的原因；
2. 通过本节课的学习，通过读源码和写 DEMO 的方式研究 `ThreadPoolExecutor` 的 `shutdownNow` 和 `shutdown` 函数的区别。