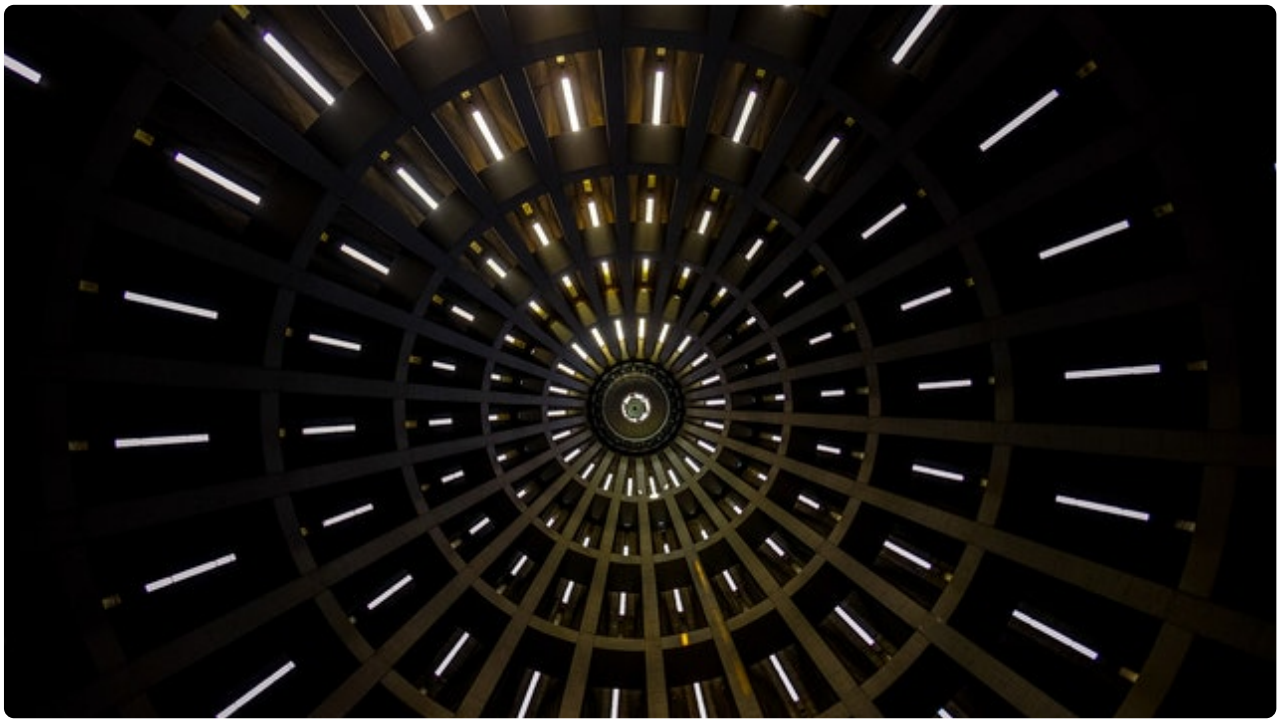
26 Netty的线程池有什么样的特性

更新时间：2020-08-14 10:10:00



> 立志是事业的大门，工作是登堂入室的旅程。——巴斯德

前言

你好，我是彤哥。

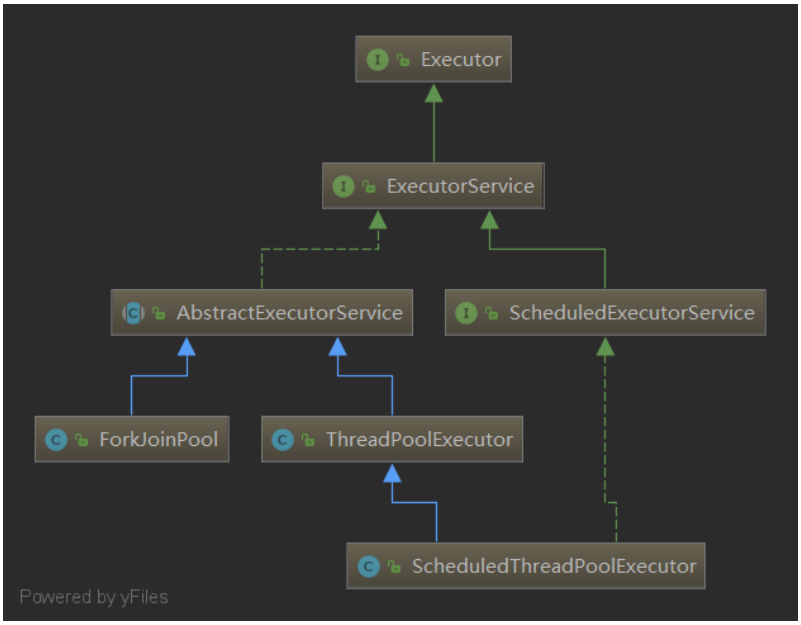
上一节，我们一起学习了 Netty 对 Java 原生 Future 的增强，并剖析了其中一个主要实现类 ——DefaultPromise，当时，我们说，Promise 主要是运用在线程池中，它必须绑定一个线程一起运行才有意义。

那么，Netty 中的线程池又是怎样的呢？为什么 Java 原生线程池无法适用于 Netty 的使用场景呢？Netty 又对线程池做了哪些改进呢？

让我们带着这些问题进入今天的学习吧。

Java 原生线程池

在正式讲解 Netty 线程池之前，同样地，我们先回顾一下 Java 原生的线程池。



Java 原生的线程池主要有三种：ThreadPoolExecutor、ScheduledThreadPoolExecutor、ForkJoinPool。

ThreadPoolExecutor 是最古老的类，我们通常说的线程池，也是指这个类。

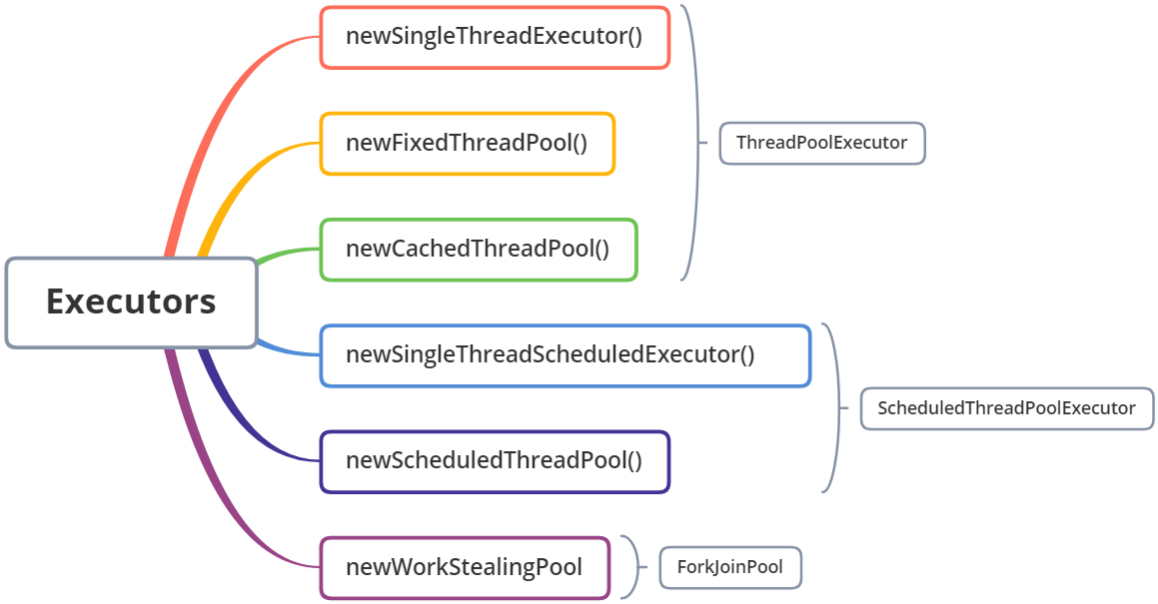
ScheduledThreadPoolExecutor 是用来执行定时任务的线程池。

ForkJoinPool 是 Java7 新增的类，它使用的是工作窃取的算法实现的一种高效的线程池，非常适合解决大任务不断地拆成小任务，小任务再最终合并成结果的场景，比如，归并排序，等等。

> 对于这三种线程池的源码解析，可以参考文末的链接自行学习。

Java 还提供了一个工具类 Executors 专门用于生成各种不同的线程池，不过阿里巴马开发者手册中，强制禁止使用此工具类创建线程池。

你知道 Executors 可以创建哪些线程池吗？



Executors 主要提供了这么六种方法用来创建线程池，当然，针对每个方法可能还有重载方法，但是，为什么阿里巴巴又禁止使用呢？看完下面的分析，你可以想想这个问题。

ThreadPoolExecutor

ThreadPoolExecutor，它是我们通常所说的线程池，也是我们拿来跟 Netty 进行对比的线程池，所以，我们今天主要讲一下这个线程池。

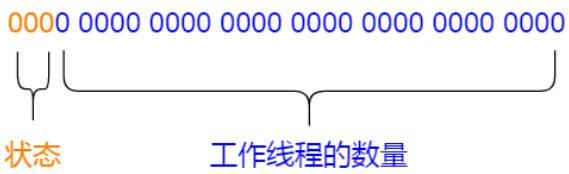
主要属性

我们先来看一下 ThreadPoolExecutor 的主要属性：

```
public class ThreadPoolExecutor extends AbstractExecutorService {
    // 控制变量
    private final AtomicInteger ctl = new AtomicInteger(ctlOf(RUNNING, 0));
    // 任务队列
    private final BlockingQueue<Runnable> workQueue;
}
```

ThreadPoolExecutor 的主要属性就是这么两个：

- ctl，控制变量，高 3 位存储的是线程池的状态，这些状态有 RUNNING、SHUTDOWN、STOP、TIDYING、TERMINATED，代表了线程池的生命周期，低 29 位存储的是工作线程的数量。



workQueue，任务队列，它是一个阻塞队列，即在多线程环境下是安全的，我认为叫作 `taskQueue` 可能更合适。

构造方法

我们再来看一看 ThreadPoolExecutor 的构造方法：

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

面试中也经常会问到：请你说一说线程池的参数？

针对 ThreadPoolExecutor，它的构造方法一共有七个参数：

- corePoolSize，核心线程数，默认情况下，这部分线程不会被销毁
- maximumPoolSize，最大线程数，最大可以创建多少个线程
- keepAliveTime，线程保持时间，线程等待多长时间还没有任务就销毁
- unit，线程保持时间的单位
- workQueue，任务队列，存放任务的队列
- threadFactory，创建线程的工厂
- handler，拒绝策略，当线程池无法再承载更多的任务时如何拒绝

任务流转

上面介绍了七个参数，那么，当我们提交一个任务到线程池的时候，这个任务又经历了怎样的历程呢？也就是任务的流转，这部分逻辑主要是在 execute () 方法中：

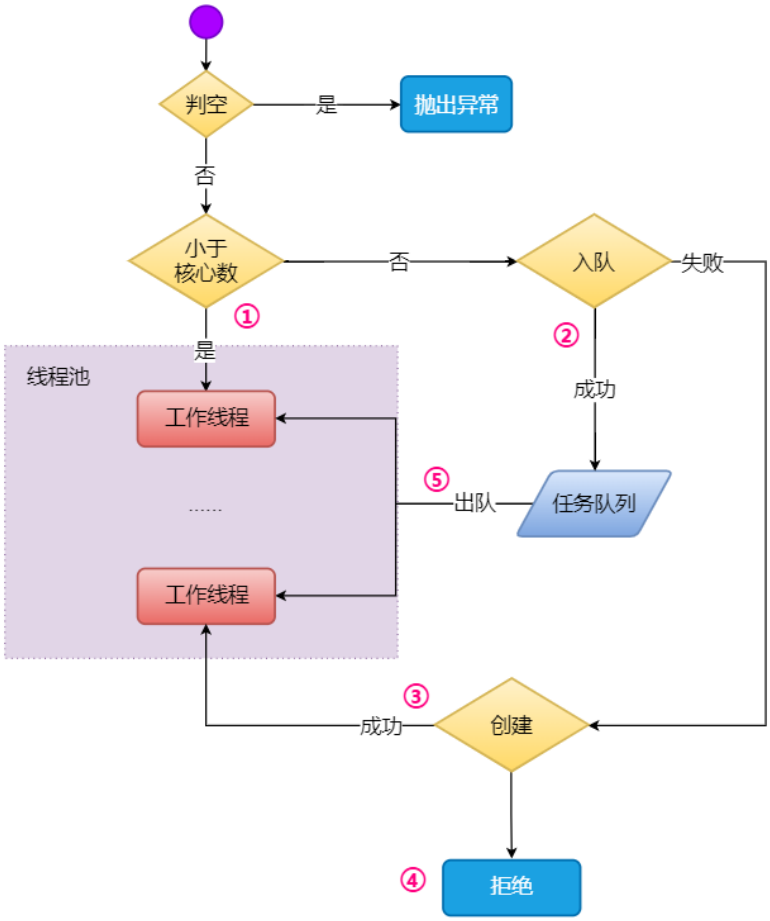
```
public void execute(Runnable command) {
    // 判空
    if (command == null)
        throw new NullPointerException();

    int c = ctl.get();
    // 工作线程的数量小于核心线程数时
    if (workerCountOf(c) < corePoolSize) {
        // 添加一个工作线程
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    // 工作线程的数量达到核心线程数时，任务入队
    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);
    }
    // 任务队列满了，添加一个工作线程
    else if (!addWorker(command, false))
        // 添加工作线程失败（达到了最大线程数），执行拒绝策略
        reject(command);
}
```

任务的流转主要分为五个阶段：

- 当工作线程数小于核心线程数时，直接创建一个工作线程来执行任务；
- 当工作线程数达到核心线程数时，尝试入队，入队成功则进入任务队列中，等待被执行；
- 如果入队失败，表示队列满了，则尝试创建一个工作线程来执行任务；
- 如果创建工作线程失败，则执行拒绝策略；
- 对于在任务队列中等待的任务，待有空闲线程时，它们会从队列中被提取出来执行。

让我们用一张图来描述这整个过程：



通过代码，可以看到，在 execute () 的方法中，只涉及到了 3 个参数，另外四个是在哪里使用的呢？

- maximuPoolSize，最大线程数，这个是在 addWorker () 方法中使用到的，如果达到了最大线程数，则会创建失败；
- threadFactory，线程工厂，这个也是在 addWorker () 方法中，具体点是在 new Worker () 的构造方法中，用来创建一个线程与 Worker 对象进行绑定；
- keepAliveTime 和 unit，线程保持时间，这一对参数是在 getTask () 方法中，表示从任务队列中取任务时，阻塞多长时间没有取到任务则结束阻塞，此时返回的任务为 null，工作线程会自然消亡。

从任务的流转过程来看，似乎很完美，但是，这个线程池有什么缺陷呢？

缺点

其实，对于我们日常的工作来说，Java 原生的线程池对于我们来说已经足够完美，但是，对于追求高性能的 Netty 看来，性能这块还是有点欠缺啊。

这里说的性能主要体现在任务队列的设计上，我们想像一下，当线程数量特别多的时候，多个线程都去竞争这一个队列，势必会导致性能的下降。

那么，有没有更好的设计呢？有，ForkJoinPool，它使用工作窃取的算法，将队列分成了全局队列和线程私有队列，总体性能有了很大的提高。

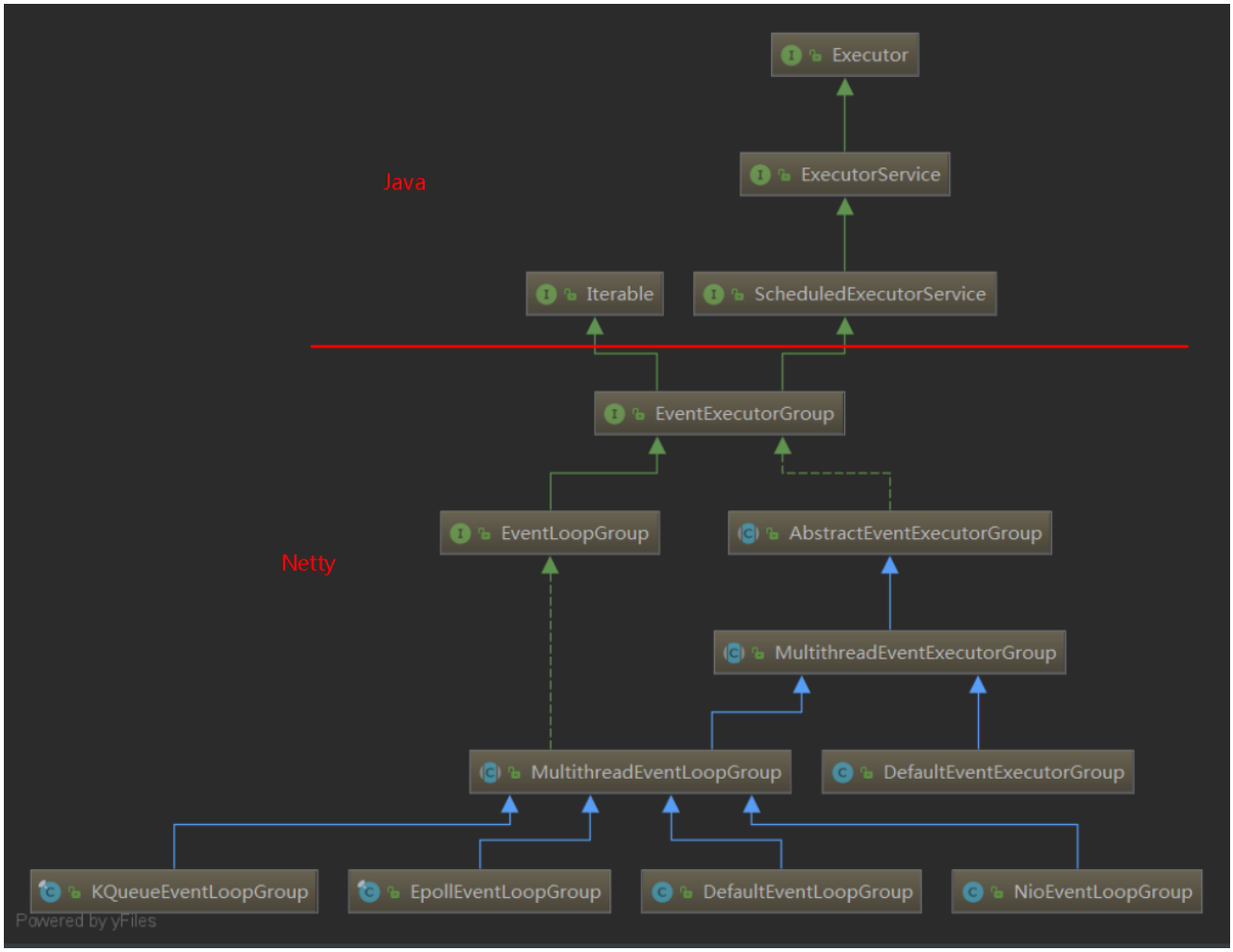
但是，Netty 的场景似乎又不太一样，它特殊在哪里呢？让我们一起进入 Netty 的世界探寻它的线程池吧。

Netty 线程池

Netty 线程池，是对 Java 原生线程池的一种增强，它的实现方式与 Java 原生线程池完全不一样，它的实现方式更适用于 Netty 的场景 —— 事件循环机制，所以，它的线程池又叫作事件循环线程池，即 EventLoopGroup。

同样地，对于 Netty 线程池的分析，我们也遵循从宏观到微观的分析方法。

继承体系



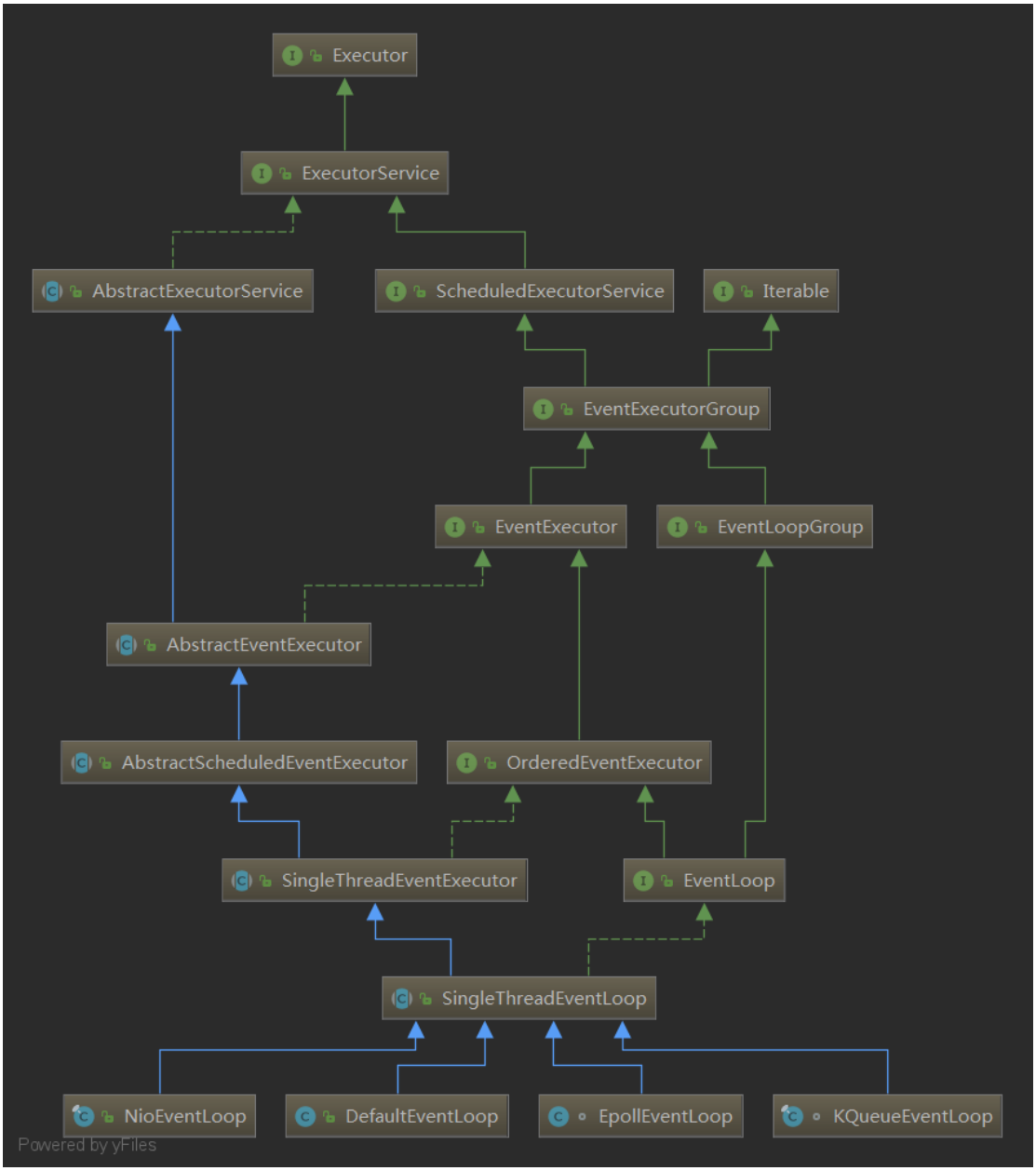
我们可以把这个图分为上下两部分，上半部分是 Java 原生的接口，包括线程池的接口以及迭代器的接口，下半部分是 Netty 扩展的接口。

Netty 主要扩展了两层接口：

- 第一层是 EventExecutorGroup，它扩展了 Java 原生的 ScheduledExecutorService 接口和 Iterable 接口，说明它同时具有定时执行任务的能力，以及迭代的能力，是不是很奇怪，它迭代的是什么？对于 EventExecutorGroup，目前，它只有一个纯的实现类 DefaultEventExecutorGroup，而且还没有被使用到。
- 第二层是 EventLoopGroup，它扩展了 EventExecutorGroup，同时，它添加了一些跟 Channel 绑定的方法，说明它是一个跟网络请求息息相关的接口，目前，Netty 中使用的都是基于 EventLoopGroup 的线程池。

既然，目前，Netty 都没有使用到只跟 EventExecutorGroup 相关的实现类，那么，把 EventExecutorGroup 和 EventLoopGroup 合并行不行呢？其实，也是可以的，不过，这样就破坏了接口隔离的原则，而且，对于以后的扩展也是不友好的，比如，在后面的版本中，就是需要某种线程池，它不是处理网络请求的，这时候我们只要实现 EventExecutorGroup 就可以了，而不再需要实现 EventLoopGroup。

同样地，针对这两层线程池的接口，Netty 也扩展出了两个工作线程的接口：



从图中可以看到，这两个工作线程的接口就是 EventExecutor 和 EventLoop，它们类似于 Java 原生线程池中的 Worker，它们本身不是线程，但是，它们维护了一个线程用来执行任务。

不过，你可能也发现了，EventExecutor 竟然继承自 EventExecutorGroup，且 EventLoop 继承自 EventLoopGroup，为什么设计得如此复杂呢？

那是因为，在 Netty 看来，EventExecutor 它就是一种特殊的 EventExecutorGroup，可以理解成，它是只包含一个线程的线程池，所以，在 Netty 中，你可以把任何 EventExecutor 的实现当作单线程的线程池使用，类似于 Executors 工具类提供 newSingleThreadExecutor () 方法，同样地，EventLoop 也是一样的道理。

调试用例

从继承体系中，我们想找到蛛丝马迹来编写调试用例是不太容易的，此时，我们打开 EventLoop 和 EventLoopGroup 的实现类，会发现，除了 DefaultEventLoop 和 DefaultEventLoopGroup，其它的实现类都是跟 Channel 强相关的，里面都是为了处理 Channel 的 IO 事件，其实，它们也正是为了不同平台而编写的多路利用的事件处理器。如果一上来就看这些类，我们势必要迷失在 Netty 的线程池中而不能自拔，所以，我们选择足够简单的 DefaultEventLoop 和 DefaultEventLoopGroup 来作为我们的研究对象，编写调试用例。

既然，它就是线程池，那我们就按线程池的写法来写调试用例就好了，下面是我写的调试用例：

```
public class DefaultEventLoopGroupTest {
    public static void main(String[] args) {
        DefaultEventLoopGroup eventLoopGroup = new DefaultEventLoopGroup(5);

        for (int i = 0; i < 10; i++) {
            eventLoopGroup.execute(() -> {
                System.out.println("thread: " + Thread.currentThread().getName());
            });
        }
    }
}
```

是不是很简单？我们定义了固定数量为 5 的线程池，然后，用它执行 10 条任务。

源码剖析

创建 DefaultEventLoopGroup

让我们来看这段代码的运行逻辑，在创建 DefaultEventLoopGroup 的位置打一个断点，跟踪进去：

```
public DefaultEventLoopGroup(int nThreads) {
    this(nThreads, (ThreadFactory) null);
}
// 省略中间的构造方法
protected MultithreadEventExecutorGroup(int nThreads, Executor executor,
                                        EventExecutorChooserFactory chooserFactory, Object... args) {
    if (nThreads <= 0) {
        throw new IllegalArgumentException(String.format("nThreads: %d (expected: > 0)", nThreads));
    }

    if (executor == null) {
        // key1，使用了一个叫作ThreadPerTaskExecutor
        executor = new ThreadPerTaskExecutor(newDefaultThreadFactory());
    }
    // 初始化工作线程数组
    children = new EventExecutor[nThreads];

    for (int i = 0; i < nThreads; i ++) {
        boolean success = false;
        try {
            // key2，创建工作线程
            children[i] = newChild(executor, args);
            success = true;
        } catch (Exception e) {
            throw new IllegalStateException("failed to create a child event loop", e);
        } finally {
            if (!success) {
                // 创建失败的处理，相当于回滚
                // 省略这部分代码
            }
        }
    }

    // key3，创建选择器
    chooser = chooserFactory.newChooser(children);

    // 省略其它代码
}
```

在构造方法中，主要是初始化一些属性，这里有三个比较重要的点：

1. 在 key1 处创建一个 ThreadPerTaskExecutor，并在创建 key2 处创建工作线程的时候当作参数传进去了，这个 ThreadPerTaskExecutor 是什么，它的作用是什么？
2. key2 处的 EventExecutor 是如何创建的？
3. key3 处的选择器的作用是什么？

我们先说第三点吧，这个选择器是什么呢？其实，它是 DefaultEventLoopGroup 用来选择哪一个 DefaultEventLoop 来执行任务时使用的，在 Netty 内部，有两种选择器，分别为 `PowerOfTwoEventExecutorChooser` 和 `GenericEventExecutorChooser`，它们本质上来说没有什么区别，主要的区别在于如果数量为 2 的 N 次方，会使用 PowerOfTwoEventExecutorChooser 按 `&` 操作来计算下一个 EventExecutor，而 GenericEventExecutorChooser 则按 `%` 运算来计算下一个 EventExecutor，本质上都是取模运算，显然直接使用 `&` 操作效率更高一些，这是 Netty 优化到极致的一个表现：

```
private static final class PowerOfTwoEventExecutorChooser implements EventExecutorChooser {
    private final AtomicInteger idx = new AtomicInteger();
    private final EventExecutor[] executors;
    @Override
    public EventExecutor next() {
        // &操作，减法操作优先级高于&操作
        // 轮询地获取EventExecutor（DefaultEventLoop）
        return executors[idx.getAndIncrement() & executors.length - 1];
    }
}

private static final class GenericEventExecutorChooser implements EventExecutorChooser {
    private final AtomicInteger idx = new AtomicInteger();
    private final EventExecutor[] executors;
    // &操作
    @Override
    public EventExecutor next() {
        return executors[Math.abs(idx.getAndIncrement() % executors.length)];
    }
}
```

好了，我们再回头看第一点，ThreadPerTaskExecutor，从名字看表示每个任务一个线程的执行器，请看它的真面目：

```
public final class ThreadPerTaskExecutor implements Executor {
    private final ThreadFactory threadFactory;

    public ThreadPerTaskExecutor(ThreadFactory threadFactory) {
        this.threadFactory = ObjectUtil.checkNotNull(threadFactory, "threadFactory");
    }

    @Override
    public void execute(Runnable command) {
        // 使用线程工厂创建一个线程并启动这个线程
        threadFactory.newThread(command).start();
    }
}
```

这个类非常简单，只有一个 execute () 方法，在被调用的时候使用线程工厂创建一个线程并启动这个线程，所以，它有一个问题，就是 execute () 方法每被调用一次就创建一个线程，这也是它的名字的由来，来一个任务创建一个线程。

好了，我们接着看第二点，EventExecutor 是如何被创建的，这里是调用了 newChild () 的方法，这个方法实际上是位于 DefaultEventLoopGroup 中：

```
// io.netty.channel.DefaultEventLoopGroup#newChild
@Override
protected EventLoop newChild(Executor executor, Object... args) throws Exception {
    return new DefaultEventLoop(this, executor);
}
// 省略中间的构造方法
protected SingleThreadEventExecutor(EventExecutorGroup parent, Executor executor,
                                    boolean addTaskWakesUp, int maxPendingTasks,
                                    RejectedExecutionHandler rejectedHandler) {
    super(parent);
    this.addTaskWakesUp = addTaskWakesUp;
    this.maxPendingTasks = Math.max(16, maxPendingTasks);
    // key1, 包装了一下传进来的ThreadPerTaskExecutor
    // 注意第二个参数
    this.executor = ThreadExecutorMap.apply(executor, this);
    // key2, 任务队列，默认使用的是LinkedBlockingQueue
    taskQueue = newTaskQueue(this.maxPendingTasks);
    // 拒绝策略
    rejectedExecutionHandler = ObjectUtil.checkNotNull(rejectedHandler, "rejectedHandler");
}
// 创建任务队列（默认的），子类可重写此方法
protected Queue<Runnable> newTaskQueue(int maxPendingTasks) {
    return new LinkedBlockingQueue<Runnable>(maxPendingTasks);
}
```

这段代码，最终，创建了一个 DefaultEventLoop，且这个 DefaultEventLoop 绑定了一个 executor 和一个任务队列，请注意这里的包含的信息：

1. executor，它是被包装之后的 ThreadPerTaskExecutor，如果被多次执行，那就会创建多个线程，所以，这个 executor 是不是只能执行一次 execute () 方法呢？
2. taskQueue，这个 DefaultEventLoop 包含一个任务队列，如果上面的 1 成立，也就是说一个 DefaultEventLoop 只有一个线程，那这个任务队列就是这个线程独享的，所以，它的出队操作不存在竞争，还记得我们前面介绍的多生产者单消费者队列 ——MpscArrayQueue 吗？

我们先卖个关子，直接看任务的执行流程。

任务的执行流程

待上面的 DefaultEventLoopGroup 创建完毕后，程序又回到的 main () 方法中，我们在任务执行的地方跟踪进去，请注意每个方法的类名：

```
// io.netty.util.concurrent.AbstractEventExecutorGroup#execute
@Override
public void execute(Runnable command) {
    // 调用选择器选择一个DefaultEventLoop
    // 根据上面选择器的源码，可知，使用的是轮询方式
    next().execute(command);
}
// io.netty.util.concurrent.SingleThreadEventExecutor#execute
@Override
public void execute(Runnable task) {
    ObjectUtil.checkNotNull(task, "task");
    // 调用下面的私有方法
    execute(task, !(task instanceof LazyRunnable) && wakesUpForTask(task));
}
// io.netty.util.concurrent.SingleThreadEventExecutor#execute
private void execute(Runnable task, boolean immediate) {
    // 当前线程是main，所以不在eventLoop中
    boolean inEventLoop = inEventLoop();
    // 添加任务到当前这个DefaultEventLoop的任务队列中
    // 如果添加失败会执行拒绝策略
    addTask(task);
    // 非不在，所以进入条件
    if (!inEventLoop) {
        // 启动线程
        startThread();
        // 省略其它代码
    }

}
```

到这里还很好理解，先把任务扔到一个队列中，再启动一个线程来运行它，关键是这个线程是如何启动的，继续跟踪到 startThread () 方法中：

```
// io.netty.util.concurrent.SingleThreadEventExecutor#startThread
private void startThread() {
    // 如果当前DefaultEventLoop的状态是未启动，才执行下面的内容
    // 也就是说对于一个DefaultEventLoop来说，这个判断下方的内容只会执行一次
    // 也就是说一个DefaultEventLoop只会创建一个线程！
    if (state == ST_NOT_STARTED) {
        // 原子更新状态变量，又使用到了前面介绍过的AtomicIntegerFieldUpdater这种方式
        if (STATE_UPDATER.compareAndSet(this, ST_NOT_STARTED, ST_STARTED)) {
            boolean success = false;
            try {
                // 又一层调用
                doStartThread();
                success = true;
            } finally {
                if (!success) {
                    STATE_UPDATER.compareAndSet(this, ST_STARTED, ST_NOT_STARTED);
                }
            }
        }
    }
}
```

好了，到这里大致的逻辑已经很清晰了，通过上面的注释，一个 DefaultEventLoop 不管执行多少次任务，只会启动一个线程，我们再接着看 doStartThread () 的内部逻辑，这个方法有七八十行，我把干扰代码都删除了：

```
// io.netty.util.concurrent.SingleThreadEventExecutor#doStartThread
private void doStartThread() {
    assert thread == null;
    // 这个executor是什么？
    // 它就是上面我们没介绍的被包装之后的ThreadPerTaskExecutor
    executor.execute(new Runnable() {
        @Override
        public void run() {
            try {
                // SingleThreadEventExecutor.this表示的是DefaultEventLoop对象
                // 所以，会调用到DefaultEventLoop的run()方法
                SingleThreadEventExecutor.this.run();
                success = true;
            } catch (Throwable t) {
            } finally {
            }
        }
    });
}
```

好烦啊，这里又把任务包装了一层，然后，调用了被包装之后的 ThreadPerTaskExecutor 的 execute () 方法，好了，下面就是揭开这个包装类真面目的时候了，上面 execute () 方法指向的是下面我加了标记的那行：

```
public final class ThreadExecutorMap {
    // 一个FastThreadLocal，存储着一个EventExecutor
    private static final FastThreadLocal<EventExecutor> mappings = new FastThreadLocal<EventExecutor>();
    // 第一个参数是ThreadPerTaskExecutor
    // 第二个参数是DefaultEventLoop
    public static Executor apply(final Executor executor, final EventExecutor eventExecutor) {
        ObjectUtil.checkNotNull(executor, "executor");
        ObjectUtil.checkNotNull(eventExecutor, "eventExecutor");
        // 返回一个Executor的匿名对象
        return new Executor() {
            @Override
            public void execute(final Runnable command) {
                // ***************这行*****************
                // 调用下面的apply
                // 这个executor就是真正的ThreadPerTaskExecutor了
                executor.execute(apply(command, eventExecutor));
            }
        };
    }
    // 第一个参数是任务
    // 第二个参数是DefaultEventLoop
    public static Runnable apply(final Runnable command, final EventExecutor eventExecutor) {
        ObjectUtil.checkNotNull(command, "command");
        ObjectUtil.checkNotNull(eventExecutor, "eventExecutor");
        return new Runnable() {
            @Override
            public void run() {
                // 设置DefaultEventLoop到FastThreadLocal中
                // 这样任务执行的过程中，都可以随时获取到这个DefaultEventLoop
                setCurrentEventExecutor(eventExecutor);
                try {
                    command.run();
                } finally {
                    // 执行完了移除
                    setCurrentEventExecutor(null);
                }
            }
        };
    }
    private static void setCurrentEventExecutor(EventExecutor executor) {
        mappings.set(executor);
    }
}
```

这里不管是对 ThreadPerTaskExecutor 的包装还是对任务的包装，都是为了找个地方把 DefaultEventLoop 存储到线程本地变量中去，以便任务在执行的过程中随时可以使用 DefaultEventLoop。

好了，程序接着走就到 ThreadPerTaskExecutor 的 execute () 方法中了：

```
// io.netty.util.concurrent.ThreadPerTaskExecutor#execute
@Override
public void execute(Runnable command) {
    threadFactory.newThread(command).start();
}
```

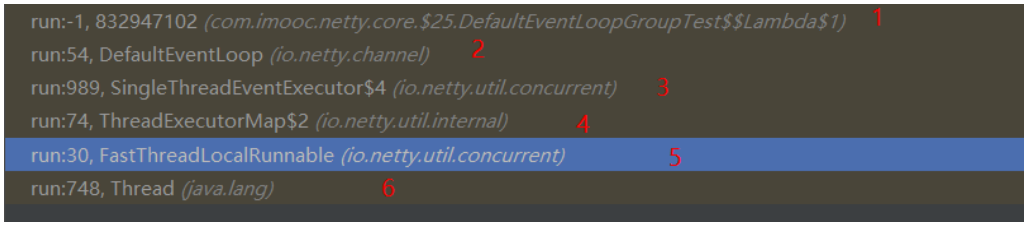
这里就调用线程工厂创建一个线程了，当然，这个线程自然是 FastThreadLocalThread，然后，启动这个线程。

此时，这个任务已经被包装了 N 层了，所以，在跳过这行之前，先在 main () 方法中任务内部打一个断点，即下面的 System.out.println () 处：

```
public class DefaultEventLoopGroupTest {
    public static void main(String[] args) {
        DefaultEventLoopGroup eventLoopGroup = new DefaultEventLoopGroup(5);

        for (int i = 0; i < 10; i++) {
            eventLoopGroup.execute(() -> {
                System.out.println("thread: " + Thread.currentThread().getName());
            });
        }
    }
}
```

然后，按 F9 就行了，因为到目前为止我们还在 main 线程中，而任务已经扔到队列中了，线程马上也要启动了，按完 F9，断点自然就停在了上面这个断点处，我们看看调用栈：



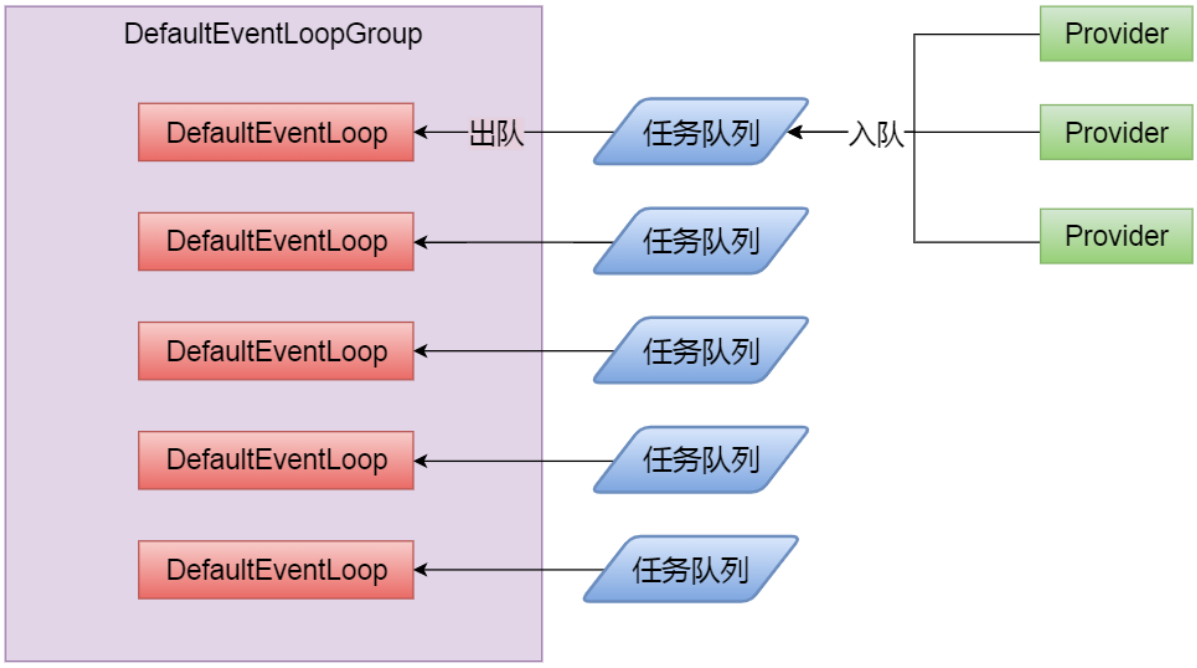
图中，1 的位置即上面断点处，2 是没有分析到的地方，3、4 是已经分析过的地方，5 是在 FastTheadLocalThread 那章分析过的，6 是 Thread 的 run () 方法。好了，我们来看看 2 处的代码：

```
// io.netty.channel.DefaultEventLoop#run
@Override
protected void run() {
    for (;;) {
        // 从队列中取任务
        Runnable task = takeTask();
        if (task != null) {
            // 执行任务
            task.run();
            updateLastExecutionTime();
        }

        if (confirmShutdown()) {
            break;
        }
    }
}
```

这个 run () 方法位于 DefaultEventLoop 中，可以看到，这是一个死循环，不断地从任务队列中取任务，然后执行，一直重复着这个动作。

结合我们前面的分析，一个 DefaultEventLoop 只会启动一个线程，而这个 DefaultEventLoop 又有自己专属的队列，所以，我们很容易就可以得出下面的线程模型：



在这个图中，我加入了 Provider 的概念，它就是任务的生产者，生产者可以有多个，所以，这就衍生出了一种多生产单消费者的任务队列，根据我们前面的学习，把这里的任务队列直接换成 MPSC 家族的队列是不是就能极大地提高效率呢？

没错，你可以看看 NioEventLoop 中重写的 newTaskQueue () 方法：

```
// io.netty.channel.nio.NioEventLoop#newTaskQueue(int)
@Override
protected Queue<Runnable> newTaskQueue(int maxPendingTasks) {
    return newTaskQueue0(maxPendingTasks);
}
private static Queue<Runnable> newTaskQueue0(int maxPendingTasks) {
    return maxPendingTasks == Integer.MAX_VALUE ? PlatformDependent.<Runnable>newMpscQueue()
        : PlatformDependent.<Runnable>newMpscQueue(maxPendingTasks);
}
```

但是，这种线程模型有个致命的缺陷 —— 千万不要在任务中执行耗时的操作，否则这个线程对应的任务队列中的任务将全部都会处于排队状态，即使整个线程池中有其它空闲的线程，它们也不会从不是自己的任务队列中挪任务过来执行。关于这一点，你可以使用下面的例子证明：

```
public class DefaultEventLoopGroupTest {
    public static void main(String[] args) {
        DefaultEventLoopGroup eventLoopGroup = new DefaultEventLoopGroup(2);

        for (int i = 0; i < 10; i++) {
            if (i%2 == 0) {
                eventLoopGroup.execute(() -> {
                    System.out.println("thread: " + Thread.currentThread().getName());
                });
            } else {
                eventLoopGroup.execute(() -> {
                    LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(1));
                    System.out.println("thread: " + Thread.currentThread().getName());
                });
            }
        }
    }
}
```

运行此程序，你会发现，一号线程早早地就执行完毕了所有 5 个任务，而二号线程则是 1 秒执行一个任务，一号线程是不会借二号线程的任务执行的，这也是 Netty 线程池与 ForkJoinPool 线程池的最大区别。

不过，这都不是个问题，Netty 线程池这样设计的目的也不是给我们的耗时业务使用的，如果有耗时的业务逻辑处理，请使用自定义的线程池进行处理，千万不要使用 Netty 的线程池。

好了，到这里，关于 Netty 的线程基本上就分析完毕了，有了这节的基础，相信你去看 NioEventLoop 的代码一定也会非常轻松的 ^^

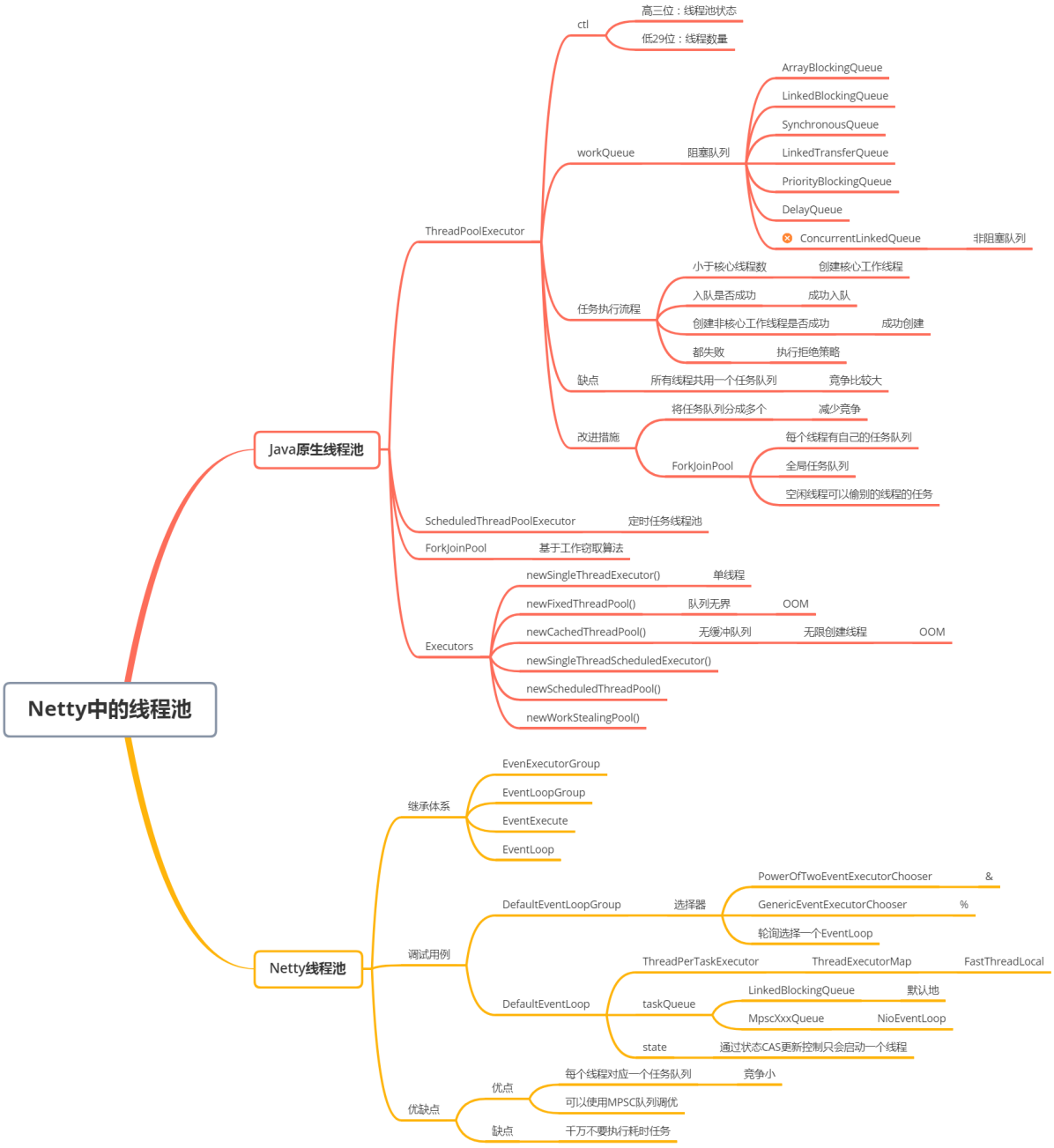
后记

本节，我们从宏观和微观两个层面深入剖析了 Netty 的线程池，我们一定要记住一点：Netty 的线程池中坚决不允许执行耗时操作。

随着本节内容的结束，所有源码的分析就到这里了，但是，这只是一个起点，有了关于 ByteBuf、内存池、对象池、FastThreadLocal、MpscArrayQueue、Future、线程池的这些源码分析，我希望你可以回过头再把服务启动过程、数据接收写出过程的源码再仔细分析一遍，这样才能真正地达到从源码级别理解 Netty。

从下一节开始，我们将进入实战课程，在实战课程中，我将通过一个游戏项目，带你手把手地写一个服务端 Netty 应用程序，敬请期待。

思维导图



参考链接

死磕 java 线程系列之终结篇

}

25 Netty的Future是如何做到简捷易用的

27 软件开发的基本步骤，无套路不欢