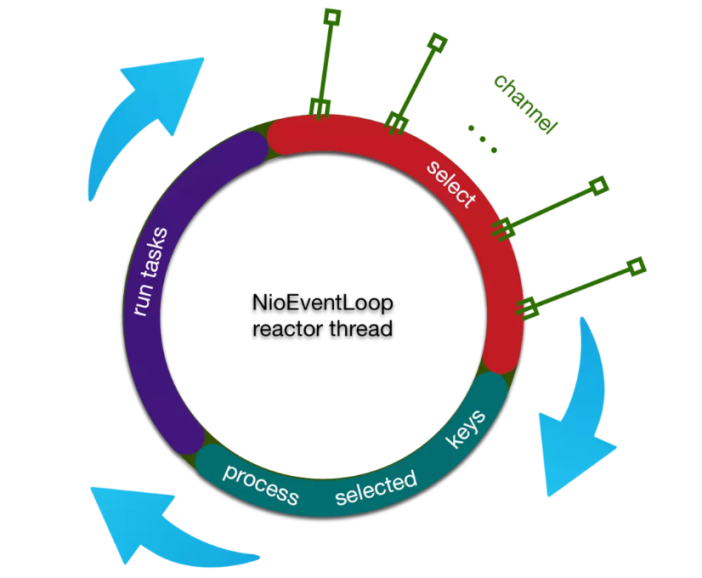
Netty 异步Task机制

Netty是一个高性能、异步事件驱动的NIO框架，其基于Reactor模型，提供对TCP、UDP和文件传输的支持，Netty所有的IO操作都是异步非阻塞的，通过FutureListener机制，用户可以方便的主动获取或者通过通知机制获得IO操作结果。在Netty中，Reactor线程NioEventLoop的执行过程如下所示：



*final long ioStartTime = System.nanoTime();*

*try {*

*processSelectedKeys();*

*} finally {*

*final long ioTime = System.nanoTime() - ioStartTime;*

*runAllTasks(ioTime \* (100 - ioRatio) / ioRatio);*

*}*

在Netty线程中，其任务的执行分为以下三步：

* 轮询出IO事件，通过NIO Selector
* 处理IO事件，根据第一步获取到的Nio连接，接收并处理数据
* 处理任务队列，其包括eventLoop#execute及schedule执行的任务，其余IO事件没有必然的联系

其中的ioRatio的含义为控制控制IO运行比例，如果ioRatio默认是50，则表示IO操作和执行Task所占用的线程比例是1:1。Task的执行通过SingleThreadEventExecutor#runAllTask来执行，如下所示：

*protected boolean runAllTasks() {*

*do {*

*fetchedAll = fetchFromScheduledTaskQueue();*

*if (runAllTasksFrom(taskQueue)) { //从队列获取Task并 执行*

*ranAtLeastOne = true;}*

*} while (!fetchedAll); ....}*

在NioEventLoop中，使用成员变量

*private final Queue<Runnable> taskQueue;  
private volatile Thread thread;  
private final Executor executor;*

来实现Task的执行，任务存放在TaskQueue中等待NioEventLoop线程调用。有三种task使用场景

1. 用户自定义普通任务，通过NioEventLoop.execute来实现，示例如下：

*ctx.channel().eventLoop().execute(new Runnable() {*

*@Override*

*public void run() {*

*System.out.println("Execute a new Runnable");*

*}*

*});*

跟进execute方法，其执行如下：

*public void execute(Runnable task) {*

*boolean inEventLoop = inEventLoop();*

*addTask(task);*

*=> offerTask(Task)*

*=> taskQueue.offer(task)*

*}*

1. 在非reactor线程中调用channel的方法，例如

AbstractChannelHandlerContext#write其执行核心代码如下：

*private void write(Object msg, boolean flush, ChannelPromise promise) {*

*AbstractChannelHandlerContext next = findContextOutbound();*

*final Object m = pipeline.touch(msg, next);*

*{*

*final AbstractWriteTask task;*

*if (flush) {*

*task = WriteAndFlushTask.newInstance(next, m, promise);*

*} else {*

*task = WriteTask.newInstance(next, m, promise);*

*}*

*if (!safeExecute(executor, task, promise, m)) {*

*task.cancel();*

*}*

*}*

*}*

将message封装成Task后，通过safeExecute直接执行

*protected static void safeExecute(Runnable task) {*

*task.run();*

*}*

1. 用户自定义定时任务

示例如下：

*ctx.channel().eventLoop().schedule(new Runnable() {*

*@Override*

*public void run() {}*

*}, 60, TimeUnit.SECONDS);*

这种场景是定时任务逻辑，在一定时间后执行，其执行核心逻辑如下：

*ScheduledFuture<V> schedule(final ScheduledFutureTask<V> task) {*

*if (inEventLoop()) {*

*scheduledTaskQueue().add(task);*

*} ...*

*}*

将Task添加到队列中，这里使用PriorityQueue，是根据时间进行排序

*PriorityQueue<ScheduledFutureTask<?>> scheduledTaskQueue*

ScheduledFureTask的比较方法如下：

*@Override*

*public int compareTo(Delayed o) {*

*ScheduledFutureTask<?> that = (ScheduledFutureTask<?>) o;*

*long d = deadlineNanos() - that.deadlineNanos();*

*if (d < 0) {*

*return -1;*

*} else if (d > 0) {*

*return 1;*

*} else if (id < that.id) {*

*return -1;*

*} else if (id == that.id) {*

*throw new Error();*

*} else {*

*return 1;*

*}*

*}*

执行时先添加到TaskQueue中

*private boolean fetchFromScheduledTaskQueue() {*

*long nanoTime = AbstractScheduledEventExecutor.nanoTime();*

*Runnable scheduledTask = pollScheduledTask(nanoTime);*

*while (scheduledTask != null) {*

*if (!taskQueue.offer(scheduledTask)) {*

*scheduledTaskQueue().add((ScheduledFutureTask<?>) scheduledTask);*

*return false;*

*}*

*scheduledTask = pollScheduledTask(nanoTime);*

*}*

*return true;*

*}*

TaskQueue中任务的执行不再介绍

https://www.jianshu.com/p/58fad8e42379

https://www.cnblogs.com/myJavaEE/p/6793332.html

http://www.blogjava.net/jb2011/archive/2018/11/05/433468.html