

多线程并发编程在 Netty 中的应用分析

作者：李林锋 © 版权所有 email: neu_lilinfeng@sina.com QQ 群：143951915

多线程并发编程在 Netty 中的应用分析.....	1
1. JAVA 内存模型与多线程编程.....	2
1.1. 硬件的发展和多任务处理.....	2
1.2. JAVA 内存模型.....	3
1.2.1. 工作内存和主内存.....	3
1.2.2. JAVA 内存交互协议.....	3
1.2.3. JAVA 的线程.....	4
2. Netty 的并发编程分析.....	5
2.1. 对共享的可变数据进行正确的同步.....	5
2.2. 正确的使用锁.....	6
2.3. volatile 的正确使用.....	8
2.4. CAS 指令和原子类.....	11
2.5. 线程安全类的应用.....	13
2.6. 读写锁的应用.....	16
2.7. 线程安全性的文档说明.....	18
2.8. 不要使用线程优先级.....	19
3. 附录.....	20
3.1. 说明.....	20
3.2. 作者简介.....	20
3.3. 交流方式.....	20

1. JAVA 内存模型与多线程编程

1.1. 硬件的发展和多任务处理

随着硬件特别是多核处理器的发展和价格的下降，多任务处理已经是所有操作系统必备的一项基本功能。在同一个时刻让计算机做多件事情，不仅仅是因为处理器的并行计算能力得到了很大提升，还有一个重要的原因是计算机的存储系统、网络通信等 IO 性能与 CPU 的计算能力差距太大，导致程序的很大一部分执行时间被浪费在 IO wait 上面，CPU 的强大运算能力没有得到充分利用。

Java 提供了很多类库和工具用于降低并发编程的门槛，提升开发效率，一些开源的第三方软件也提供了额外的并发编程类库方便 JAVA 开发者，使开发者将重心放在业务逻辑的设计和实现上，而不是处处考虑线程的同步和锁。但是，无论并发类库设计的如何完美，它都无法完全满足使用者的需求，对于一个高级 JAVA 程序员来说，如果不懂得 JAVA 并发编程的内膜，只懂得使用一些简单的并发类库和工具，是无法完全驾驭 JAVA 多线程这匹野马的。

1.2. JAVA 内存模型

JVM 规范定义了 JAVA 内存模型（Java Memory Model）来屏蔽掉各种操作系统、虚拟机实现厂商和硬件的内存访问差异，以实现 JAVA 程序在所有操作系统和平台上能够实现一次编写、到处运行的效果。

Java 内存模型的制定既要严谨，保证语义无歧义，另外，也要制定的尽量宽松一些，允许各硬件和虚拟机实现厂商有足够的灵活性来充分利用硬件的特性提升 JAVA 的内存访问性能。随着 JDK 的发展，Java 的内存模型已经逐渐成熟起来。

1.2.1. 工作内存和主内存

Java 内存模型规定所有的变量都存储在主内存中（JVM 内存的一部分），每个线程有自己独立的工作内存，它保存了被该线程使用的变量的主内存拷贝，线程对这些变量的操作都在自己的工作内存中进行，不能直接操作主内存和其它工作内存中存储的变量或者变量副本，线程间的变量访问需通过主内存来完成，三者的关系如下图所示：

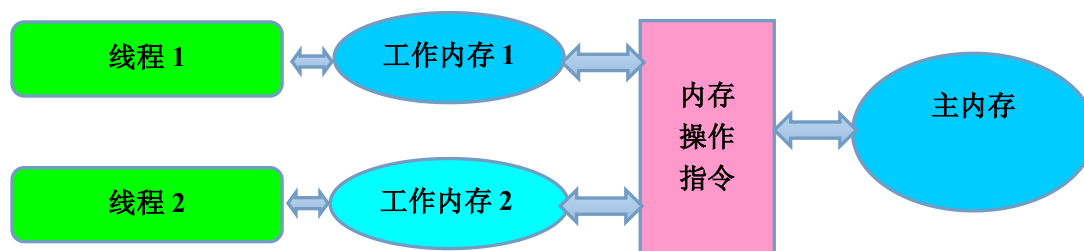


图 1.2.1 JAVA 内存访问模型

1.2.2. JAVA 内存交互协议

JAVA 内存模型定义了八种操作来完成主内存和工作内存的变量访问，具体如下：

1. lock: 主内存变量，把一个变量标识为某个线程独占的状态；
2. unlock: 主内存变量，把一个处于锁定状态变量释放出来，被释放后的变量才可以被其它线程锁定；
3. read: 主内存变量，把一个变量的值从主内存传输到线程的工作内存中，以便随后的 load 动作使用；
4. load: 工作内存变量，把 read 读取到的主内存中的变量值放入工作内存的变量拷贝中；
5. use: 工作内存变量，把工作内存中变量的值传递给 java 虚拟机执行引擎，每当虚拟机遇到一个需要使用到变量值的字节码指令时将会执行该操作；
6. assign: 工作内存变量，把从执行引擎接收到的变量的值赋值给工作变量，每当虚拟机遇到一个给变量赋值的字节码时将会执行该操作；
7. store: 工作内存变量，把工作内存中一个变量的值传送到主内存中，以便随后的 write 操作使用；
8. write: 主内存变量，把 store 操作从工作内存中得到的变量值放入主内存的变量中。

1.2.3. JAVA 的线程

并发的实现可以通过多种方式来实现，例如：单进程-单线程模型，通过在

一台服务器上启动多个进程实现多任务的并行处理。但是在 JAVA 语言中，是通过单进程-多线程的模型进行多任务的并发处理。因此，我们有必要熟悉一下 JAVA 的线程。

大家都知道，线程是比进程更轻量级的调度执行单元，它可以把进程的资源分配和调度执行分开，各个线程可以共享内存、IO 等操作系统资源，但是又能够被操作系统发的内核线程或者进程执行。各线程可以独立的启动、运行和停止，实现任务的解耦。

主流的操作系统都提供了线程实现，目前实现线程的方式主要有三种，分别是：

1. 内核线程（KLT）实现，这种线程由内核来完成线程切换，内核通过线程调度器对线程进行调度，并负责将线程任务映射到不同的处理器上；
2. 用户线程实现（UT），通常情况下，用户线程指的是完全建立在用户空间线程库上的线程，用户线程的创建、启动、运行、销毁和切换完全在用户态中完成，不需要内核的帮助，因此执行性能更高；
3. 混合实现：将内核线程和用户线程混合在一起使用的方式。

由于虚拟机规范并没有强制规定 JAVA 的线程必须使用哪种方式实现，因此，不同的操作系统实现的方式也可能存在差异。对于 SUN 的 JDK，在 Windows 和 Linux 操作系统上采用了内核线程的实现方式，在 Solaris 版本的 JDK 中，提供了一些专有的虚拟机线程参数，用于设置使用哪种线程模型。

2. Netty 的并发编程分析

2.1. 对共享的可变数据进行正确的同步

关键字 `synchronized` 可以保证在同一时刻，只有一个线程可以执行某一个方法或者代码块。同步的作用不仅仅是互斥，它的另一个作用就是共享可变性，当某个线程修改了可变数据并释放锁后，其它的线程可以获取被修改变量的最新值。如果没有正确的同步，这种修改对其它线程是不可见的。

下面我们就通过对 Netty 的源码进行分析，看看 Netty 是如何对并发可变

数据进行正确同步的。

以 ServerBootstrap 为例进行分析，首先看它的 option 方法：

```
@SuppressWarnings("unchecked")
public <T> B option(ChannelOption<T> option, T value) {
    if (option == null) {
        throw new NullPointerException("option");
    }
    if (value == null) {
        synchronized (options) {
            options.remove(option);
        }
    } else {
        synchronized (options) {
            options.put(option, value);
        }
    }
    return (B) this;
}
```

这个方法的作用是设置 ServerBootstrap 的 ServerSocketChannel 的 Socket 属性，它的属性集定义如下：

```
private final Map<ChannelOption<?>, Object> options = new LinkedHashMap<ChannelOption<?>, Object>();
```

由于是非线程安全的 LinkedHashMap，所以如果多线程创建、访问和修改 LinkedHashMap 时，必须在外部进行必要的同步，LinkedHashMap 的 API DOC 对于线程安全的说明如下：

```
* <p><strong>Note that this implementation is not synchronized.</strong>
* If multiple threads access a linked hash map concurrently, and at least
* one of the threads modifies the map structurally, it <em>must</em> be
* synchronized externally. This is typically accomplished by
* synchronizing on some object that naturally encapsulates the map.
```

由于 ServerBootstrap 是被使用者创建和使用的，我们无法保证它的方法和成员变量不被并发访问，因此，作为成员变量的 options 必须进行正确的同步。由于考虑到锁的范围需要尽可能的小，我们对传参的 option 和 value 的合法性判断不需要加锁。因此，代码才对两个判断分支独立加锁，保证锁的范围尽可能的细粒度。

Netty 加锁的地方非常多，大家在阅读代码的时候可以仔细体会下，为什么有的地方要加锁，有的地方有不需要？如果不需要，为什么？当你对锁的真谛理解以后，对于这些锁的使用时机和技巧理解起来就非常容易了。

2.2. 正确的使用锁

对于很多刚接触多线程编程的开发者，意识到了并发访问可变变量需要加锁，但是对于锁的范围、加锁的时机和锁的协同缺乏认识，往往会导致一些问题，下面我就结合 Netty 的代码来讲解下这方面的知识。

打开 ForkJoinTask，我们学习一些多线程同步和协作方面的技巧，先看下当条件不满足时阻塞某个任务，直到条件满足后再继续执行，代码如下：

```
private int externalAwaitDone() {
    int s;
    ForkJoinPool cp = ForkJoinPool.common;
    if ((s = status) >= 0) {
        if (cp != null) {
            if (this instanceof CountedCompleter)
                s = cp.externalHelpComplete((CountedCompleter<?>)this);
            else if (cp.tryExternalUnpush(this))
                s = doExec();
        }
        if (s >= 0 && (s = status) >= 0) {
            boolean interrupted = false;
            do {
                if (U.compareAndSwapInt(this, STATUS, s, s | SIGNAL)) {
                    synchronized (this) {
                        if (status >= 0) {
                            try {
                                wait();
                            } catch (InterruptedException ie) {
                                interrupted = true;
                            }
                        } else
                            notifyAll();
                    }
                }
            } while ((s = status) >= 0);
        }
    }
}
```

重点看下红框中的代码，首先通过循环检测的方式对状态变量 status 进行判断，当它的状态大于等于 0 时，执行 wait()，阻塞当前的调度线程，直到 status 小于 0，唤醒所有被阻塞的线程，继续执行。这个方法有三个多线程的编程技巧需要说明：

1. wait 方法别用来使线程等待某个条件，它必须在同步块内部被调用，这个同步块通常会锁定当前对象实例。下面是这个模式的标准使用方式：

```
synchronized (this)
{
    While(condition)
        Object.wait;
    .....
}
```

2. 始终使用 wait 循环来调用 wait 方法，永远不要在循环之外调用 wait 方法。原因是尽管条件并不满足被唤醒条件，但是由于其它线程意外调用

notifyAll() 方法会导致被阻塞线程意外唤醒，此时执行条件并不满足，它将破坏被锁保护的约定关系，导致约束失效，引起意想不到的结果；

3. 唤醒线程，应该使用 notify 还是 notifyAll，当你不知道究竟该调用哪个方法时，保守的做法是调用 notifyAll 唤醒所有等待的线程。从优化的角度看，如果处于等待的所有线程都在等待同一个条件，而每次只有一个线程可以从这个条件中被唤醒，那么就应该选择调用 notify。

当多个线程共享同一个变量的时候，每个读或者写数据的操作方法都必须加锁进行同步，如果没有正确的同步，就无法保证一个线程所做的修改被其它线程可见。未能同步共享变量会造成程序的活性失败和安全性失败，这样的失败通常是难以调试和重现的，它们可能间歇性的出问题，也可能随着并发的线程个数而失败，也可能在不同的虚拟机或者操作系统上存在不同的失败概率。因此，我们务必要保证锁的正确使用。下面这个案例，就是个典型的错误应用：

```
int size = 0;

public synchronized void increase()

{

    size++;

}

public int current()

{

    Return size;

}
```

2.3. volatile 的正确使用

在制定《华为技术有限公司 JAVA 编程规范 V2.0》版本中，大家对于 volatile 如何正确使用有很多的争议。我发现即便是一些经验丰富的 JAVA 设计师，对于 volatile 和多线程编程的认识仍然存在误区。其实，volatile 的使用非常简单，只要理解了 JAVA 的内存模型和多线程编程基础知识，正确使用 volatile 是不存

在任何问题的，下面我们结合 Netty 的源码，对 volatile 的正确使用进行说明。

打开 NioEventLoop 的代码，我们来看下控制 IO 操作和其它任务运行比例的 ioRatio，它是 int 类型的变量，定义如下：

```
private volatile int ioRatio = 50;
```

我们发现，它被定义为 volatile，为什么呢？我们首先对 volatile 关键字进行说明，然后再结合 Netty 的代码进行分析。

关键字 volatile 是 JAVA 提供的最轻量级的同步机制，JAVA 内存模型对 volatile 专门定义了一些特殊的访问规则，下面我们就看下它的规则：

当一个变量被 volatile 修饰后，它将具备两种特性：

1. 线程可见性：当一个线程修改了被 volatile 修饰的变量后，无论是否加锁，其它线程都可以立即看到最新的修改，而普通变量却做不到这点；
2. 禁止指令重排序优化，普通的变量仅仅保证在该方法的执行过程中所有依赖赋值结果的地方都能获取正确的结果，而不能保证变量赋值操作的顺序与程序代码的执行顺序一致。举个简单的例子说明下指令重排序优化问题：

```
public class ThreadStopExample {  
    private static boolean stop;  
  
    /**  
     * @param args  
     * @throws InterruptedException  
     */  
    public static void main(String[] args) throws InterruptedException {  
        java.lang.Thread workThread = new java.lang.Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while (!stop) {  
                    i++;  
                    try {  
                        TimeUnit.SECONDS.sleep(1);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        });  
        workThread.start();  
        TimeUnit.SECONDS.sleep(3);  
        stop = true;  
    }  
}
```

我们预期程序会在 3S 后停止，但是实际上它会一直执行下去，原因就是虚拟机对代码进行了指令重排序和优化，优化后的指令如下：

```
if (!stop)
```


While(true)

.....

重排序后的代码是无法发现 stop 被主线程修改的，因此无法停止运行。如果要解决这个问题，只要将 stop 前增加 volatile 修饰符即可，代码修改如下：

```
public class ThreadStopExample {  
  
    private volatile static boolean stop;  
  
    /**  
     * @param args  
     * @throws InterruptedException  
     */  
    public static void main(String[] args) throws InterruptedException {  
        java.lang.Thread workThread = new java.lang.Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while (!stop) {  
                    i++;  
                    try {  
                        TimeUnit.SECONDS.sleep(1);  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
        });  
        workThread.start();  
        TimeUnit.SECONDS.sleep(3);  
        stop = true;  
    }  
}
```

再次运行，我们发现 3S 后程序退出，达到了预期效果，使用 volatile 解决了如下两个问题：

1. main 线程对 stop 的修改在 workThread 线程中可见，也就是说 workThread 线程立即看到了其它线程对于 stop 变量的修改；
2. 禁止指令重排序，防止因为重排序导致的并发访问逻辑混乱。

一些人错误的认为使用 volatile 可以代替传统锁，提升并发性能，这个认识是错误的，volatile 仅仅解决了可见性的问题，但是它并不能保证互斥性，也就是说多个线程并发修改某个变量时，依旧会产生多线程问题。因此，不能靠 volatile 来完全替代传的锁。

根据经验总结，volatile 最适合使用的地方是一个线程写、其它线程读的场合，如果有多个线程并发写操作，仍然需要使用锁或者线程安全的容器或者原子变量来代替。

讲了 volatile 的原理之后，我们继续对 Netty 的源码做分析，上面我们说到了 ioRatio 被定义成 volatile，下面看看代码为啥这样定义：

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

final int ioRatio = this.ioRatio;
runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
```

通过代码分析我们发现，在 NioEventLoop 线程中，ioRatio 并没有被修改，它是只读操作。那既然没有修改，为啥要定义成 volatile 呢？我们继续看代码，我们发现 NioEventLoop 提供了重新设置 IO 执行时间比例的公共方法，接口如下：

```
/**
 * Sets the percentage of the desired amount of time spent for I/O in the event loop. The default value is
 * (8code 50), which means the event loop will try to spend the same amount of time for I/O as for non-I/O tasks.
 */
public void setIoRatio(int ioRatio) {
    if (ioRatio <= 0 || ioRatio >= 100) {
        throw new IllegalArgumentException("ioRatio: " + ioRatio + " (expected: 0 < ioRatio < 100)");
    }
    this.ioRatio = ioRatio;
}
```

首先，NioEventLoop 线程没有调用该方法，说明调整 IO 执行时间比例是外部发起的操作，通常是由业务的线程调用该方法，重新设置该参数。这样就形成了一个线程写、一个线程读，根据前面针对 volatile 的应用总结，此时可以使用 volatile 来代替传统的 synchronized 关键字提升并发访问的性能。

Netty 中大量使用了 volatile 来修改成员变量，如果理解了 volatile 的应用场景，读懂 Netty volatile 的相关代码还是比较容易的。

2.4. CAS 指令和原子类

互斥同步最主要的问题就是进行线程阻塞和唤醒所带来的性能问题，因此这种同步被称为阻塞同步，它属于一种悲观的并发策略，我们称之为悲观锁。随着硬件和操作系统指令集的发展和优化，产生了非阻塞同步，被称为乐观锁。简单的说就是先进行操作，操作完成之后再判断下看看操作是否成功，是否有并发问题，如果有进行失败补偿，如果没有就算操作成功，这样就从根本上避免了同步锁的弊端。

目前，在 JAVA 中应用最广泛的非阻塞同步就是 CAS，在 IA64、X86 指令集中通过 cmpxchg 指令完成 CAS 功能，在 sparc-TS0 中由 case 指令完成，在 ARM 和 PowerPC 架构下，需要使用一对 Idrex/strex 指令完成。

从 JDK1.5 以后，可以使用 CAS 操作，该操作由 `sun.misc.Unsafe` 类里面的 `compareAndSwapInt()` 和 `compareAndSwapLong()` 等方法包装提供。通常情况下 `sun.misc.Unsafe` 类对于开发者是不可见的，因此，JDK 提供了很多 CAS 包装类简化开发者的使用，例如 `AtomicInteger` 等。

下面，结合 Netty 的源码，我们对于原子类的正确使用进行详细说明：

我们打开 `ChannelOutboundBuffer` 的代码，看看如何对发送的总字节数进行计数和更新操作，先看定义：

```
private static final AtomicLongFieldUpdater<ChannelOutboundBuffer> TOTAL_PENDING_SIZE_UPDATER =
    AtomicLongFieldUpdater.newUpdater(ChannelOutboundBuffer.class, "totalPendingSize");

private volatile long totalPendingSize;
```

首先定义了一个 `volatile` 的变量，它可以保证某个线程对于 `totalPendingSize` 的修改可以被其它线程立即访问到，但是，它无法保证多线程并发修改的安全性。紧接着又定义了一个 `AtomicIntegerFieldUpdater` 类型的变量 `WTOTAL_PENDING_SIZE_UPDATER`，实现 `totalPendingSize` 的原子更新，也就是保证 `totalPendingSize` 的多线程修改并发安全性，我们重点看下 `AtomicIntegerFieldUpdater` 的 API 说明：

```
/**
 * A reflection-based utility that enables atomic updates to
 * designated {@code volatile int} fields of designated classes.
 * This class is designed for use in atomic data structures in which
 * several fields of the same node are independently subject to atomic
 * updates.
 *
 * <p>Note that the guarantees of the {@code compareAndSet}
 * method in this class are weaker than in other atomic classes.
 * Because this class cannot ensure that all uses of the field
 * are appropriate for purposes of atomic access, it can
 * guarantee atomicity only with respect to other invocations of
 * {@code compareAndSet} and {@code set} on the same updater.
 *
 * @since 1.5
 * @author Doug Lea
 * @param <T> The type of the object holding the updatable field
 */
public abstract class AtomicIntegerFieldUpdater<T> {
```

从 API 的说明我们可以看出来，它主要用于实现 `volatile` 修饰的 `int` 变量的原子更新操作，对于使用者，必须通过类似 `compareAndSet` 或者 `set` 或者与这些操作等价的原子操作来保证更新的原子性，否则会导致问题。

我们继续看代码，当执行 `write` 操作外发消息的时候，需要对外发的消息字节数

进行统计汇总，由于调用 write 操作的既可以是 IO 线程，也可以是业务的线程，也可能由业务线程池多个工作线程同时执行发送任务，因此，统计操作是多线程并发的，这也就是为什么要将计数器定义成 volatile 并使用原子更新类进行原子操作，下面，我们看下计数的代码：

```
long oldValue = totalPendingSize;
long newWriteBufferSize = oldValue + size;
while (!TOTAL_PENDING_SIZE_UPDATER.compareAndSet(this, oldValue, newWriteBufferSize)) {
    oldValue = totalPendingSize;
    newWriteBufferSize = oldValue + size;
}
```

首先，我们发现计数操作并没有实现锁，而是使用了 CAS 自旋操作，通过 TOTAL_PENDING_SIZE_UPDATER.compareAndSet(this, oldValue, newWriteBufferSize) 来判断本次原子操作是否成功，如果成功则退出循环，代码继续执行；如果失败，说明在本次操作的过程中计数器已经被其它线程更新成功，我们需要进入循环，首先，对 oldValue 进行更新，代码如下：

```
oldValue = totalPendingSize;
```

然后重新对更新值进行计算：

```
newWriteBufferSize = oldValue + size;
```

继续循环进行 CAS 操作，直到成功。它跟 AtomicInteger 的 compareAndSet 操作类似。

使用 JAVA 自带的 Atomic 原子类，可以避免同步锁带来的并发访问性能降低的问题，减少犯错的机会，因此，Netty 中对于 int、long、boolean 等大量使用其原子类，减少了锁的应用，降低了频繁使用同步锁带来的性能下降。

2.5. 线程安全类的应用

在 JDK1.5 的发行版本中，Java 平台新增了 java.util.concurrent，这个包中提供了一系列的线程安全集合、容器和线程池，利用这些新的线程安全类可以极大的降低 Java 多线程编程的难度，提升开发效率。

新的并发编程包中的工具可以分为如下四类：

1. 线程池 Executor Framework 以及定时任务相关的类库，包括 Timer 等；
2. 并发集合，包括 List、Queue、Map 和 Set 等；
3. 新的同步器，例如读写锁 ReadWriteLock 等；
4. 新的原子包装类，例如 AtomicInteger 等。

在实际编码过程中，我们建议通过使用线程池、Task(Runnable/Callable)、原子类和线程安全容器来代替传统的同步锁、wait 和 notify，提升并发访问的性能、降低多线程编程的难度。

下面，我们针对新的线程并发包在 Netty 中的应用进行分析和说明，以期为大家的应用提供指导。

首先，我们看下线程安全容器在 Netty 中的应用，NioEventLoop 是 IO 线程，负责网络读写操作，它同时也执行一些非 IO 的任务，例如事件通知、定时任务执行等，因此，它需要一个任务队列来缓存这些 Task，它的任务队列定义如下：

```
@Override
protected Queue<Runnable> newTaskQueue() {
    // This event loop never calls takeTask()
    return new ConcurrentLinkedQueue<Runnable>();
}
```

它是一个 ConcurrentLinkedQueue，我们看下它的 API 说明：

```
/**
 * An unbounded thread-safe {@link PlainQueue} based on linked nodes.
 * This queue orders elements FIFO (first-in-first-out).
 * The <em>head</em> of the queue is that element that has been on the
 * queue the longest time.
 * The <em>tail</em> of the queue is that element that has been on the
 * queue the shortest time. New elements
 * are inserted at the tail of the queue, and the queue retrieval
 * operations obtain elements at the head of the queue.
 * A {@code ConcurrentLinkedQueue} is an appropriate choice when
 * many threads will share access to a common collection.
 * Like most other concurrent collection implementations, this class
 * does not permit the use of {@code null} elements.
 */
```

DOC 文档明确说明这个类是线程安全的，因此，对它进行读写操作不需要加锁，下面我们继续看下队列中增加一个任务：

```
/**
 * Add a task to the task queue, or throws a {@link RejectedExecutionException} if this instance was shutdown
 * before.
 */
protected void addTask(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }
    if (isShutdown()) {
        reject();
    }
    taskQueue.add(task);
}
```

读取任务，也不需要加锁：

```
BlockingQueue<Runnable> taskQueue = (BlockingQueue<Runnable>) this.taskQueue;
for (;;) {
    ScheduledFutureTask<?> delayedTask = delayedTaskQueue.peek();
    if (delayedTask == null) {
        Runnable task = null;
        try {
            task = taskQueue.take();
            if (task == WAKEUP_TASK) {
                task = null;
            }
        } catch (InterruptedException e) {
            // Ignore
        }
        return task;
    }
}
```

JDK 的线程安全容器底层采用了 CAS、volatile 和 ReadWriteLock 实现，相比于传统重量级的同步锁，采用了更轻量、细粒度的锁，因此，性能会更高。采用这些线程安全容器，不仅仅能提升多线程并发访问的性能，还能降低开发难度。

下面我们看看线程池在 Netty 中的应用，打开 SingleThreadEventExecutor 看下它是如何定义和使用线程池的：

首先定义了一个标准的线程池用于执行任务：

```
private final Executor executor;
```

接着对它赋值并且进行初始化操作：

```
this.addTaskWakesUp = addTaskWakesUp;
this.executor = executor;

taskQueue = newTaskQueue();
```

执行任务：

```
public void execute(Runnable task) {
    if (task == null) {
        throw new NullPointerException("task");
    }

    boolean inEventLoop = inEventLoop();
    if (inEventLoop) {
        addTask(task);
    } else {
        startThread();
        addTask(task);
        if (isShutdown() && removeTask(task)) {
            reject();
        }
    }

    if (!addTaskWakesUp) {
        wakeup(inEventLoop);
    }
}
```

我们发现，实际上是执行任务就是先把任务加入到任务队列中，然后判断线程是

否已经启动循环执行，如果不是需要启动线程，启动线程代码如下：

```
private void startThread() {
    synchronized (stateLock) {
        if (state == ST_NOT_STARTED) {
            state = ST_STARTED;
            delayedTaskQueue.add(new ScheduledFutureTask<Void> {
                this, delayedTaskQueue, Executors.<Void>callable(new PurgeTask(), null),
                ScheduledFutureTask.deadlineNanos(SCHEDULE_PURGE_INTERVAL, -SCHEDULE_PURGE_INTERVAL));
            doStartThread();
        }
    }
}

private void doStartThread() {
    assert thread == null;
    executor.execute(new Runnable() {
        @Override
        public void run() {
            thread = Thread.currentThread();
            if (interrupted) {
                thread.interrupt();
            }
        }
    });
}
```

实际上就是执行当前线程的 run 方法，循环从任务队列中获取 Task 并执行，我们看下它的子类 NioEventLoop 的 run 方法就能一目了然：

```
cancelledKeys = 0;

final long ioStartTime = System.nanoTime();
needsToSelectAgain = false;
if (selectedKeys != null) {
    processSelectedKeysOptimized(selectedKeys.flip());
} else {
    processSelectedKeysPlain(selector.selectedKeys());
}

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

final int ioRatio = this.ioRatio;
runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
```

如红框中所示，循环从任务队列中获取任务并执行：

```
/**
 * Poll all tasks from the task queue and run them via {@link Runnable#run()} method. This method stops running
 * the tasks in the task queue and returns if it ran longer than {@code timeoutNanos}.
 */
protected boolean runAllTasks(long timeoutNanos) {
    fetchFromDelayedQueue();
    Runnable task = pollTask();
    if (task == null) {
        return false;
    }

    final long deadline = ScheduledFutureTask.nanoTime() + timeoutNanos;
    long runTasks = 0;
    long lastExecutionTime;
    for (;;) {
        try {
            task.run();
        } catch (Throwable t) {
            logger.warn("A task raised an exception.", t);
        }
        runTasks++;
    }
}
```

Netty 对 JDK 的线程池进行了封装和改造，但是，本质上仍然是利用了线程池和线程安全队列简化了多线程编程。

2. 6. 读写锁的应用

JDK1.5 新的并发编程工具包中新增了读写锁，它是个轻量级、细粒度的锁，合理的使用读写锁，相比于传统的同步锁，可以提升并发访问的性能和吞吐量，在读多写少的场景下，使用同步锁比同步块性能高一大截。

尽管 JDK1.6 之后，随着 JVM 团队对 JIT 即使编译器的不断优化，同步块和读写锁的性能差距缩小了很多；但是，读写锁的应用依然非常广泛，例如，JDK 的线程安全 List CopyOnWriteArrayList 就是基于读写锁实现的，代码如下：

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    private static final long serialVersionUID = 8673264195747942595L;

    /** The lock protecting all mutators */
    transient final ReentrantLock lock = new ReentrantLock();

    /** The array, accessed only via getArray/setArray. */
    private volatile transient Object[] array;
```

下面，我们对 Netty 中的读写锁应用进行分析，让大家掌握读写锁的用法，打开 HashedWheelTimer 代码，读写锁定义如下：

```
final int mask;
final ReadWriteLock lock = new ReentrantReadWriteLock();
```

当新增一个定时任务的时候使用了读锁，用于感知 wheel 的变化，由于读锁是

```
public Timeout newTimeout(TimerTask task, long delay, TimeUnit unit) {
    start();

    if (task == null) {
        throw new NullPointerException("task");
    }
    if (unit == null) {
        throw new NullPointerException("unit");
    }

    long deadline = System.nanoTime() + unit.toNanos(delay) - startTime;

    // Add the timeout to the wheel.
    HashedWheelTimeout timeout;
    lock.readLock().lock();
    try {
        timeout = new HashedWheelTimeout(task, deadline);
        if (workerState.get() == WORKER_STATE_SHUTDOWN) {
            throw new IllegalStateException("Cannot enqueue after shutdown");
        }
        wheel[timeout.stopIndex].add(timeout);
    } finally {
        lock.readLock().unlock();
    }
}
```

共享锁，所以当有多个线程同时调用 newTimeout 的时候，并不会互斥，这样，就提升了并发读的性能。

获取并删除所有过期的任务时，由于要从迭代器中删除任务，所以使用了写锁：


```

private void fetchExpiredTimeouts(
    List<HashedWheelTimeout> expiredTimeouts, long deadline) {

    // Find the expired timeouts and decrease the round counter
    // if necessary. Note that we don't send the notification
    // immediately to make sure the listeners are called without
    // an exclusive lock.
    lock.writeLock().lock();
    try {
        fetchExpiredTimeouts(expiredTimeouts, wheel[(int) (tick & mask)].iterator(), deadline);
    } finally {
        // Note that the tick is updated only while the writer lock is held,
        // so that newTimeout() and consequently new HashedWheelTimeout() never see an old value
        // while the reader lock is held.
        tick++;
        lock.writeLock().unlock();
    }
}

```

读写锁的使用总结：

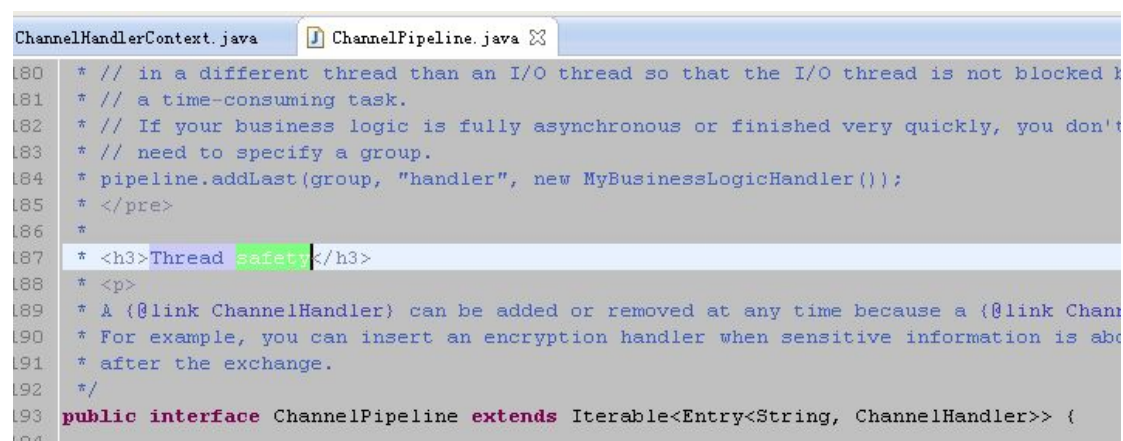
1. 主要用于读多写少的场景，用来替代传统的同步锁，以提升并发访问性能；
2. 读写锁是可重入、可降级的，一个线程获取读写锁后，可以继续递归获取；从写锁可以降级为读锁，以便快速释放锁资源；
3. ReentrantReadWriteLock 支持获取锁的公平策略，在某些特殊的应用场景下，可以提升并发访问的性能，同时兼顾线程等待公平性；
4. 读写锁支持非阻塞的尝试获取锁，如果获取失败，直接返回 false, 而不是同步阻塞，这个功能在一些场景下非常有用。例如多个线程同步读写某个资源，当发生异常或者需要释放资源的时候，由哪个线程释放是个挑战，因为某些资源不能重复释放或者重复执行，这样，可以通过 tryLock 方法尝试获取锁，如果拿不到，说明已经被其它线程占用，直接退出即可；
5. 获取锁之后一定要释放锁，否则会发生锁溢出异常。通常的做法是通过 finally 块释放锁。如果是 tryLock，获取锁成功才需要释放锁。

2.7. 线程安全性的文档说明

当一个类的方法或者成员变量被并发使用的时候，这个类的行为如何，是该类与其客户端程序建立约定的重要组成部分。如果没有在这个类的文档中描述其行为的并发情况，使用这个类的程序员不得不做出某种假设。如果这些假设是错误的，这个程序就缺少必要的同步保护，会导致意想不到的并发问题，这些问题通常都是隐蔽和调试困难的。如果同步过度，会导致意外的性能下降，无论是发生何种情况，缺少线程安全性的说明文档，都会令开发人员非常沮丧，他们会

对这些类库的使用小心翼翼，提心吊胆。

在 Netty 中，对于一些关键的类库，给出了线程安全习惯的 API DOC，尽管 Netty 的线程安全性并不是非常完善，但是，相比于一些做的更糟糕的产品，它还是迈出了总要一步。



```
ChannelHandlerContext.java ChannelPipeline.java ✕
180 * // in a different thread than an I/O thread so that the I/O thread is not blocked
181 * // a time-consuming task.
182 * // If your business logic is fully asynchronous or finished very quickly, you don't
183 * // need to specify a group.
184 * pipeline.addLast(group, "handler", new MyBusinessLogicHandler());
185 * </pre>
186 *
187 * <h3>Thread safety</h3>
188 * <p>
189 * A {@link ChannelHandler} can be added or removed at any time because a {@link Chan
190 * For example, you can insert an encryption handler when sensitive information is ab
191 * after the exchange.
192 */
193 public interface ChannelPipeline extends Iterable<Entry<String, ChannelHandler>> {
194
```

由于 ChannelPipeline 的应用非常广泛，因此，在 API 中对它的线程安全性进行了详细的说明，这样，开发者在调用 ChannelPipeline 的 API 时，就不要再额外的考虑线程同步和并发问题。

2.8. 不要使用线程优先级

当有多个线程同时运行的时候，由线程调度器来决定哪些线程运行、哪些等待以及线程切换的时间点，由于各个操作系统的线程调度器实现大相径庭，因此，依赖 JDK 自带的线程优先级来设置线程优先级策略的方法是错误和非平台可移植的。所以，在任何情况下，你的程序都不能依赖 JDK 自带的线程优先级来保证执行顺序、比例和策略。

Netty 中默认的线程工厂实现类，开放了包含设置线程优先级字段的构造函数，这个是个错误的决定，对于使用者来说，既然提供了优先级字段，它就本能的认为它可以正确并且到处运行，但实际上由于它基于 JDK 默认的线程优先级实现，因此无法满足使用者的需求。

```
public DefaultThreadFactory(String poolName, boolean daemon, int priority) {  
    if (poolName == null) {  
        throw new NullPointerException("poolName");  
    }  
    if (priority < Thread.MIN_PRIORITY || priority > Thread.MAX_PRIORITY) {  
        throw new IllegalArgumentException(  
            "priority: " + priority + " (expected: Thread.MIN_PRIORITY <= priority <= Thread.MAX_PRIORITY)"  
        );  
    }  
  
    prefix = poolName + '-' + poolId.incrementAndGet() + '-';  
    this.daemon = daemon;  
    this.priority = priority;  
}
```

3. 附录

3.1. 说明

本文并不试图面面俱到的涵盖所有 JAVA 多线程编程的知识点，如果读者希望了解更多的多线程编程知识，建议阅读 Joshua Bloch 的《Java Concurrency in Practive》。

事实上，多线程编程作为一个难点，也是阻拦读者深入理解 Netty 架构和源码的一个拦路虎，如果你的基本功不够扎实，对形形色色的多线程编程技术无法透彻理解，就很难说能够真正理解 Netty。当你带着这些疑问去定位 Netty 深层次问题的时候，也是困难重重。

实际上，如果你不能够精通 JAVA 的多线程编程，对于很多开源框架的深入理解都会非常困难。

本文通过对 Netty 中主要使用的多线程编程技术进行归纳、汇总和分析，以期让读者更快的熟悉多线程编程的常用知识点和技巧，加深大家对 Netty 的理解。

3.2. 作者简介

李林锋，华为技术有限公司平台中间件架构与设计部架构设计师，曾参与和主持多个重要平台产品的设计和开发工作。《华为技术有限公司 JAVA 编程规范 V2.0》版本的主要起草人和评审专家，负责 JAVA 多线程编程、JAVA 安全等重要章节的编写。2011 年，参与设计和开发的某平台获得软件公司总裁技术创新奖二等奖。

3.3. 交流方式

大家可以通过 QQ 和微博与交流和互动，联系方式如下：

QQ: Netty 中国社区，群号：143951915

微博：李林锋 hw

Email: neu_lilinfeng@sina.com

