个人资料

Mr_xiaoM

关注 发私信

访问： 12739次

积分： 306

等级： BLOG 2

排名： 千里之外

原创： 12篇 转载： 40篇

译文： 0篇 评论： 0条

文章搜索

文章分类

java (32)

php (8)

算法 (7)

前端 (3)

web service (4)

工具 (6)

其他 (8)

阅读排行

使用Jersey框架创建RESTful ... (614)

使用 jersey构建RESTful的We... (540)

存储到数据库的文章如何保留... (478)

Netty In Action中文版 - 第八... (470)

Netty In Action中文版 - 第一... (461)

Netty In Action中文版 - 第十... (427)

Netty In Action中文版 - 第十... (397)

Netty In Action中文版 - 第五... (396)

Netty In Action中文版 - 第七... (361)

Netty In Action中文版 - 第十... (350)

推荐文章

* Android 反编译初探 应用是如何被注入广告的
* 凭兴趣求职80%会失败，为什么
* 安卓微信自动抢红包插件优化和实现

Netty In Action中文版 - 第十五章：选择正确的线程模型

标签： java netty nio

2014-09-15 21:27 241人阅读 评论

分类： java (31)

本章介绍

- 线程模型(thread-model)
- 事件循环(EventLoop)
- 并发(Concurrency)
- 任务执行(task execution)
- 任务调度(task scheduling)

线程模型定义了应用程序或框架如何执行你的代码，选择应用程序/框架的正确的线程模型是很重要的。Netty提供模型来帮助我们简化代码，Netty对所有的核心代码都进行了同步。所有ChannelHandler，包括业务逻辑，都保证由一通道。这并不意味着Netty不能使用多线程，只是Netty限制每个连接都由一个线程处理，这种设计适用于非阻塞程序。程中的任何问题，也不用担心会抛ConcurrentModificationException或其他一些问题，如数据冗余、加锁等，这些问发时是经常会发生的。

读完本章就会深刻理解Netty的线程模型以及Netty团队为什么会选择这样的线程模型，这些信息可以让我们在使用性能。此外，Netty提供的线程模型还可以让我们编写整洁简单的代码，以保持代码的整洁性；我们还会学习Netty团队线程模型，现在我们将使用Netty提供的更容易更强大的线程模型来开发。

尽管本章讲述的是Netty的线程模型，但是我们仍然可以使用其他的线程模型；至于如何选择一个完美的线程模型需求来判断。

本章假设如下：

- 你明白线程是什么以及如何使用，并有使用线程的工作经验；若不是这样，就请花些时间来了解清楚这些知识。推程实战。
- 你了解多线程应用程序及其设计，也包括如何保证线程安全和获取最佳性能。
- 你了解java.util.concurrent以及ExecutorService和ScheduledExecutorService。

15.1 线程模型概述

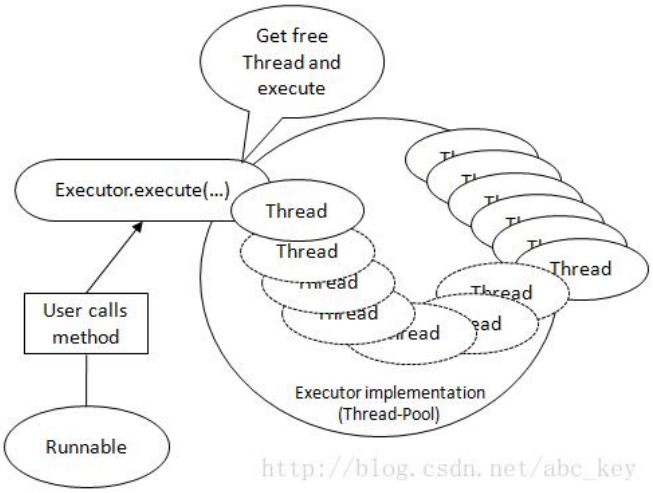
本节将简单介绍一般的线程模型，Netty中如何使用指定的线程模型，以及Netty不同的版本中使用的线程模型。你程模型的所有利弊。

如果思考一下，在我们的生活中会发现很多情况都会使用线程模型。例如，你有一个餐厅，向你的客户提供食品，才能给客户；某个客户下了订单后，你需要将煮熟事物这个任务发送到厨房，而厨房可以以不同的方式来处理，这就像何执行任务。

- 只有一个厨师：
  - 这种方法是单线程的，一次只执行一个任务，完成当前订单后再处理下一个。
- 你有多个厨师，每个厨师都可以做，空闲的厨师准备着接单做饭：
  - 这种方式是多线程的，任务由多个线程(厨师)执行，可以并行同时执行。
- 你有多个厨师并分成组，一组做晚餐，一个做其他：
  - 这种情况也是多线程，但是带有额外的限制；同时执行多个任务是由实际执行的任务类型(晚餐或其他)决定

从上面的例子看出，日常活动适合在一个线程模型。但是Netty在这里适用吗？不幸的是，它没有那么简单，Nett藏了来自用户的大部分。Netty使用多个线程来完成所有的工作，只有一个线程模型线型暴露给用户。大多数现代应用程作，让应用程序充分使用系统的资源来有效工作。在早期的Java中，这样做是通过按需创建新线程并行工作。但很快发为创建和回收线程需要较大的开销。在Java5中加入了线程池，创建线程和重用线程交给一个任务执行，这样使创建和回低。

下图显示使用一个线程池执行一个任务，提交一个任务后会使用线程池中空闲的线程来执行，完成任务后释放线程池：



上图每个任务线程的创建和回收不需要新线程去创建和销毁，但这只是一半的问题，我们稍后学习。你可能会问为用一个ExecutorService可以有助于防止线程创建和回收的成本？

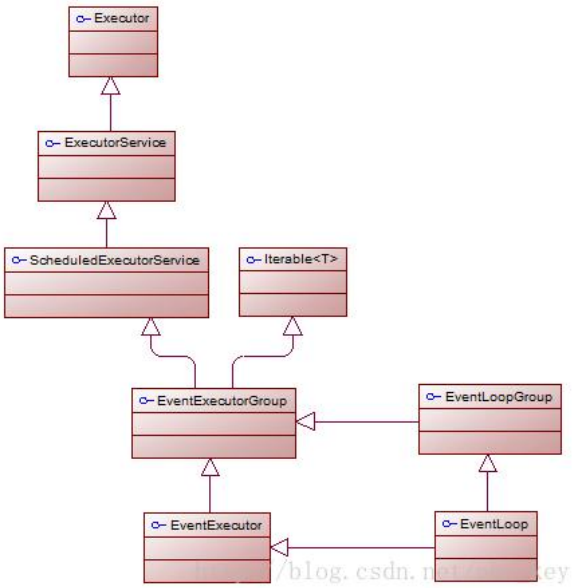
使用多线程会有太多的上下文切换，提高了资源和管理成本，这种副作用会随着运行线程的数量和执行的任务数量用多线程在刚开始可能没有什么问题，但随着系统的负载增加，可能在某个点就会让系统崩溃。

除了这些技术上的限制和问题，在项目生命周期内维护应用程序/框架可能还会发生其他问题。它有效的说明了增加于它是平行的，简单的陈述：编写多线程应用程序时一个辛苦的工作！我们怎么来解决这个问题呢？在实际的场景中需来看看Netty是如何解决这个问题的。

15.2 事件循环

事件循环所做的正如它的名字，它运行的事件在一个循环中，直到循环终止。这非常适合网络框架的设计，因为它接运行一个事件循环。这不是Netty的新发明，其他的框架和实现已经很早就这样做了。

在Netty中使用EventLoop接口代表事件循环，EventLoop是从EventExecutor和ScheduledExecutorService扩展直接交给EventLoop执行。类关系图如下：



15.2.1 使用事件循环

下面代码显示如何访问已分配给通道的EventLoop并在EventLoop中执行任务：

[java] view plain copy

```
Channel ch = ...;
ch.eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        System.out.println("run in the eventloop");
    }
});
```

使用事件循环的好处是不需要担心同步问题，在同一线程中执行所有其他关联通道的其他事件。这完全符合Netty否已执行，使用返回的Future，使用Future可以访问很多不同的操作。下面的代码是检查任务是否执行：

[java] view plain copy

```
Channel ch = ...;
Future<?> future = ch.eventLoop().submit(new Runnable() {
    @Override
    public void run() {

    }
});
if(future.isDone()){
    System.out.println("task complete");
}else {
    System.out.println("task not complete");
}
```

检查执行任务是否在事件循环中:

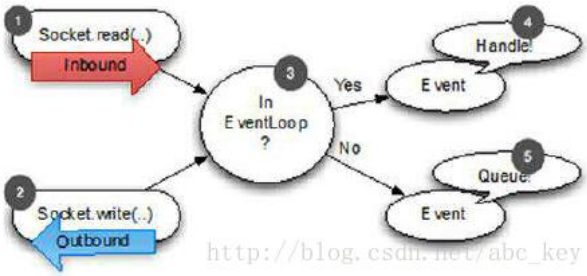
[java] view plain copy

```
Channel ch = ...;
if(ch.eventLoop().inEventLoop()){
    System.out.println("in the EventLoop");
}else {
    System.out.println("outside the EventLoop");
}
```

只有确认没有其他EventLoop使用线程池了才能关闭线程池，否则可能会产生未定义的副作用。

15.2.2 Netty4中的I/O操作

这个实现很强大，甚至Netty使用它来处理底层I/O事件，在socket上触发读和写操作。这些读和写操作是网络API层操作系统提供。下图显示在EventLoop上下文中执行入站和出站操作，如果执行线程绑定到EventLoop，操作会直接将排队执行：

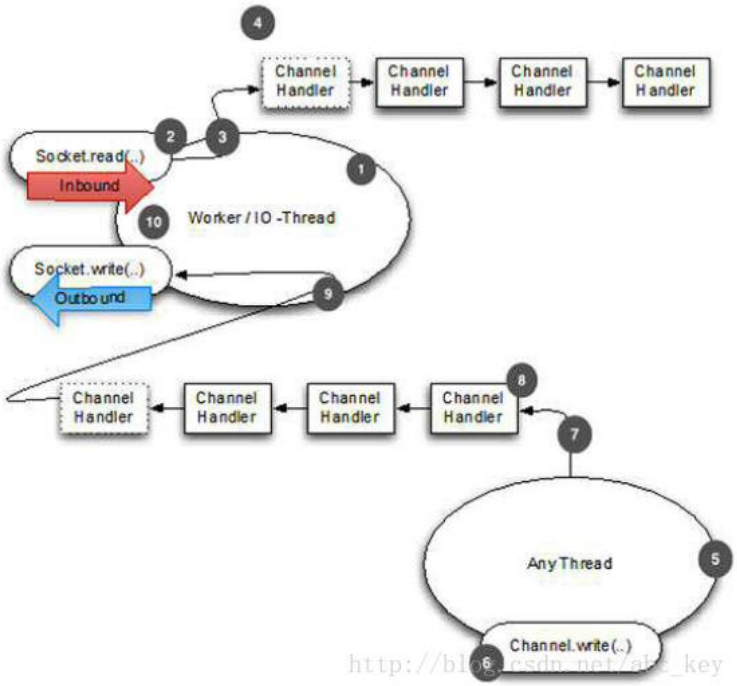


需要一次处理一个事件取决于事件的性质，通常从网络堆栈读取或传输数据到你的应用程序，有时在另外的方向做的应用程序传输数据到网络堆栈再发送到远程对等通道，但不限于这种类型的事物；更重要的是使用的逻辑是通用的，例。

应该指出的是，线程模型(事件循环的顶部)描述并不总是由Netty使用。我们在了解Netty3后会更容易理解为什么

15.2.3 Netty3中的I/O操作

在以前的版本有点不同，Netty保证在I/O线程中只有入站事件才被执行，所有的出站时间被调用线程处理。这看起易出错。它还将负责同步ChannelHandler来处理这些事件，因为它不保证只有一个线程同时操作；这可能发生在你去执例如，在不同的线程调用Channel.write(...)。下图显示Netty3的执行流程：



除了需要负担同步ChannelHandler，这个线程模型的另一个问题是你可能需要去掉一个入站事件作为一个出站事Channel.write(...)操作导致异常。在这种情况下，捕获的异常必须生成并抛出去。乍看之下这不像是一个问题，但我们件涉及，会让你知道问题出在哪里。问题是，事实上，你现在的情况是在调用线程上执行，但捕获到异常事件必须交给行的，但如果你忘了传递过去，它会导致线程模型失效；假设入站事件只有一个线程不是真，这可能会给你各种各样的

以前的实现有一个唯一的积极影响，在某些情况下它可以提供更好的延迟；成本是值得的，因为它消除了复杂性。程序中，你不会遵守任何差异延迟，还取决于其他因数，如：

- 字节写入到远程对等通道有多快
- I/O线程是否繁忙
- 上下文切换
- 锁定

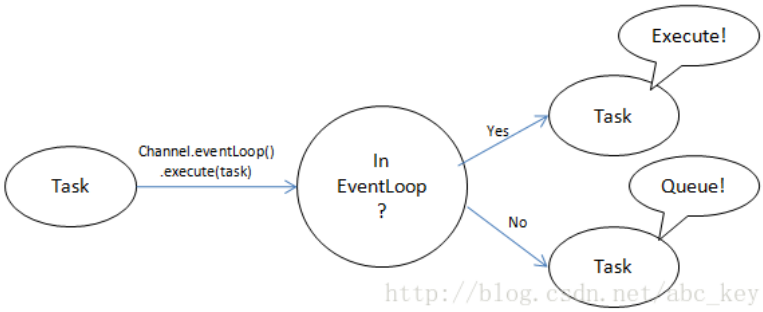
你可以看到很多细节影响整体延迟。

15.2.4 Netty线程模型内部

Netty的内部实现使其线程模型表现优异，它会检查正在执行的线程是否是已分配给实际通道(和EventLoop)，在EventLoop负责处理所有的事件。如果线程是相同的EventLoop中的一个，讨论的代码块被执行；如果线程不同，它安

队列后执行。通常是通过EventLoop的Channel只执行一次下一个事件，这允许直接从任何线程与通道交互，同时还确
ChannelHandler是线程安全，不需要担心并发访问问题。

下图显示在EventLoop中调度任务执行逻辑，这适合Netty的线程模型：



设计是非常重要的，以确保不要把任何长时间运行的任务放在执行队列中，因为长时间运行的任务会阻止其他在
这多少会影响整个系统依赖于EventLoop实现用于特殊传输的实现。传输之间的切换在你的代码库中可能没有任何改变
I/O线程。如果你必须做阻塞调用(或执行需要长时间才能完成的任务)，使用EventExecutor。

下一节将讲解一个在应用程序中经常使用的功能，就是调度执行任务(定期执行)。Java对这个需求提供了解决方案
好的方案。

15.3 调度任务执行

每隔一段时间需要调度任务执行，也许你想注册一个任务在客户端完成连接5分钟后执行，一个常见的用例是发送
着？”到远程对等通道，如果远程对等通道没有反应，则可以关闭通道(连接)和释放资源。就像你和朋友打电话，沉默
说“你还在吗？”，如果朋友没有回复，就可能是断线或朋友睡着了；不管是什么问题，你都可以挂断电话，没有什么
后，收起电话可以做其他的事。

本节介绍使用强大的EventLoop实现任务调度，还会简单介绍Java API的任务调度，以方便和Netty比较加深理解

15.3.1 使用普通的Java API调度任务

在Java中使用JDK提供的ScheduledExecutorService实现任务调度。使用Executors提供的静态方法创建Schedul
如下方法：

- newScheduledThreadPool(int)
- newScheduledThreadPool(int, ThreadFactory)
- newSingleThreadScheduledExecutor()
- newSingleThreadScheduledExecutor(ThreadFactory)

看下面代码：

[java] view plain copy

```
01. ScheduledExecutorService executor = Executors.newScheduledThreadPool(10);
02. ScheduledFuture<?> future = executor.schedule(new Runnable() {
03.     @Override
04.     public void run() {
05.         System.out.println("now it is 60 seconds later");
06.     }
07. }, 60, TimeUnit.SECONDS);
08. if(future.isDone()){
09.     System.out.println("scheduled completed");
10. }
11. //.....
12. executor.shutdown();
```

15.3.2 使用EventLoop调度任务

使用ScheduledExecutorService工作的很好，但是有局限性，比如在一个额外的线程中执行任务。如果需要执行
很严重；对于像Netty这样的高性能的网络框架来说，严重的资源使用是不能接受的。Netty对这个问题提供了很好的方

Netty允许使用EventLoop调度任务分配到通道，如下面代码：

[java] view plain copy

```
01. Channel ch = ...;
02. ch.eventLoop().schedule(new Runnable() {
03.     @Override
```

```
    public void run() {
        System.out.println("now it is 60 seconds later");
    }
}, 60, TimeUnit.SECONDS);
```

如果想任务每隔多少秒执行一次，看下面代码：

```
[java] view plain copy
Channel ch = ...;
ScheduledFuture<?> future = ch.eventLoop().scheduleAtFixedRate(new Runnable() {
    @Override
    public void run() {
        System.out.println("after run 60 seconds,and run every 60 seconds");
    }
}, 60, 60, TimeUnit.SECONDS);
// cancel the task
future.cancel(false);
```

15.3.3 调度的内部实现

Netty内部实现其实是基于George Varghese提出的“Hashed and hierarchical timing wheels: Data structu implement timer facility(散列和分层定时轮：数据结构有效实现定时器)”。这种实现只保证一个近似执行，也就是说100%准确；在实践中，这已经被证明是一个可容忍的限制，不影响多数应用程序。所以，定时执行任务不可能100%准

为了更好的理解它是如何工作，我们可以这样认为：

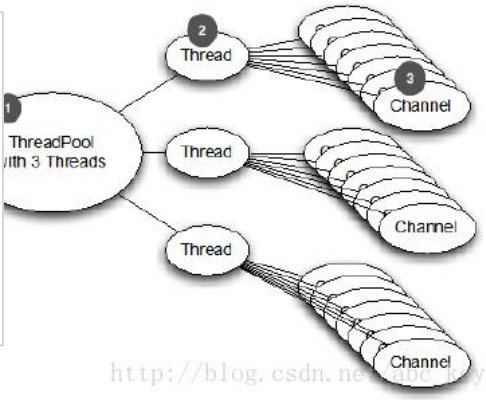
1. 在指定的延迟时间后调度任务；
2. 任务被插入到EventLoop的Schedule-Task-Queue(调度任务队列)；
3. 如果任务需要马上执行，EventLoop检查每个运行；
4. 如果有一个任务要执行，EventLoop将立刻执行它，并从队列中删除；
5. EventLoop等待下一次运行，从第4步开始一遍又一遍的重复。

因为这样的实现计划执行不可能100%正确，对于多数用例不可能100%准备的执行计划任务；在Netty中，这样的但是如果需要更准确的执行呢？很容易，你需要使用ScheduledExecutorService的另一个实现，这不是Netty的内容。的线程模型协议，你将需要自己同步并发访问。

15.4 I/O线程分配细节

Netty使用线程池来为Channel的I/O和事件服务，不同的传输实现使用不同的线程分配方式；异步实现是只有几个这样可以使用最小的线程数为很多的平道服务，不需要为每个通道都分配一个专门的线程。

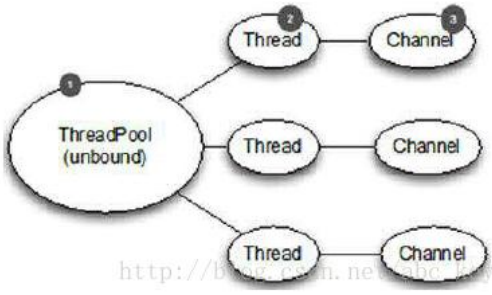
下图显示如何分配线程池：



如上图所示，使用一个固定大小的线程池管理三个线程，创建线程池后就把线程分配给线程池，确保在需要的时候程。这三个线程会分配给每个新创建的已连接通道，这是通过EventLoopGroup实现的，使用线程池来管理资源；实际线程上，这种分布以循环的方式完成，因此它可能不会100%准确，但大部分时间是准确的。

一个通道分配到一个线程后，在这个通道的生命周期内都会一直使用这个线程。这一点在以后 会被改赖这种方式；不会被改变的是一个线程在同一时间只会处理一个通道的I/O操作，我们可以依赖这种 种方式步。

下图显示OIO(Old Blocking I/O)传输：



从上图可以看出，每个通道都有一个单独的线程。我们可以使用java.io.*包里的类来开发基于阻塞I/O的应用程序一件事仍然保持不变，每个通道的I/O在同时只能被一个线程处理；这个线程是由Channel的EventLoop提供，我们可以这也是Netty框架比其他网络框架更容易编写的原因。

15.5 Summary

本章主要讲解Netty的线程模型，其核心接口是EventLoop；并和OIO中的线程模型做了比较，以突显Netty的优异性。

查看评论

暂无评论

发表评论

用 户 名： gavin2lee7

评论内容：

提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VP
IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apa
HTML SDK IIS Fedora XML LBS Unity Splashtop UML components Windows Mobile Ra
Cassandra CloudStack FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App S
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr An
Cloud Foundry Redis Scala Django Bootstrap