Netty In Action中文版 - 第一章：Netty介绍

目录(?) [-]

本章介绍

- Netty介绍
- 为什么要使用non-blocking IO(NIO)
- 阻塞IO(blocking IO)和非阻塞IO(non-blocking IO)对比
- Java NIO的问题和在Netty中的解决方案

Netty是基于Java NIO的网络应用框架，如果你是Java网络方面的新手，那么本章将是你学习Java网络应用的开始；对于有经验的开发者来说，学习本章内容也是很好的复习。如果你熟悉NIO和NIO2，你可以随时跳过本章直接从第二章开始学习。在你的机器上运行第二章编写的Netty服务器和客户端。

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。Netty的内部实现时很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。Netty是完全基于NIO实现的，所以整个Netty都是异步的。

网络应用程序通常需要有较高的可扩展性，无论是Netty还是其他的基于Java NIO的框架，都会提供可扩展性的解决方案。Netty中一个关键组成部分是它的异步特性，本章将讨论同步(阻塞)和异步(非阻塞)的IO来说明为什么使用异步代码来解决扩展性问题以及如何使用异步。

对于那些初学网络变成的读者，本章将帮助您对网络应用的理解，以及Netty是如何实现他们的。它说明了如何使用基本的Java网络API，探讨Java网络API的优点和缺点并阐述Netty是如何解决Java中的问题的，比如Eploo错误或内存泄露问题。

在本章的结尾，你会明白什么是Netty以及Netty提供了什么，你会理解Java NIO和异步处理机制，并通过本书的其他章节加强理解。

1.1 为什么使用Netty？

David John Wheeler说过“在计算机科学中的所有问题都可以通过间接的方法解决。”作为一个NIO client-server框架，Netty提供了这样的一个间接的解决方法。Netty提供了高层次的抽象来简化TCP和UDP服务器的编程，但是你仍然可以使用底层地API。

(David John Wheeler有一句名言“计算机科学中的任何问题都可以通过加上一层逻辑层来解决”，这个原则在计算机各技术领域被广泛应用)

1.1.1 不是所有的网络框架都是一样的

Netty的“quick and easy(高性能和简单易用)”并不意味着编写的程序的性能和可维护性会受到影响。从Netty中实现的协议如FTP，SMTP，HTTP，WebSocket，SPDY以及各种二进制和基于文本的传统协议中获得的经验导致Netty的创始人要非常小心它的设计。Netty成功的提供了易于开发，高性能和高稳定性，以及较强的扩展性。

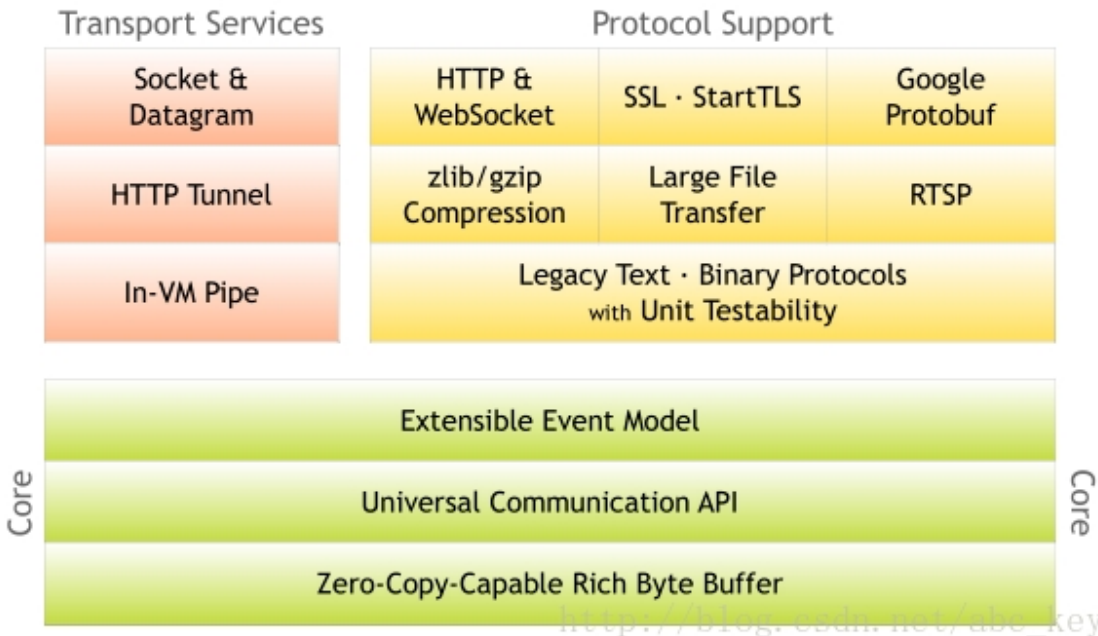
高调的公司和开源项目有RedHat, Twitter, Infinispan, and HornetQ, Vert.x, Finagle, Akka, Apache Cassandra, Elasticsearch，以及其他人的使用有助于Netty的发展，Netty的一些特性也是这些项目的需要所致。多年来，Netty变的更广为人

知，它是Java网络的首选框架，在一些开源或非开源的项目中可以体现。并且，Netty在2011年获得Duke's Choice Award(Duke's Choice奖)。

此外，在2011年，Netty的创始人Trustion Lee离开RedHat后加入Twitter，在这一点上，Netty项目奖会成为一个独立的项目组织。RedHat和Twitter都使用Netty，所以它毫不奇怪。在撰写本书时RedHat和Twitter这两家公司是最大的贡献者。使用Netty的项目越来越多，Netty的用户群体和项目以及Netty社区都是非常活跃的。

1.1.2 Netty的功能非常丰富

通过本书可以学习Netty丰富的功能。下图是Netty框架的组成



Netty除了提供传输和协议，在其他各领域都有发展。Netty为开发者提供了一套完整的工具，看下面表格：

Development Area	Netty Features
Design(设计)	• 各种传输类型，阻塞和非阻塞套接字统一的API • 使用灵活 • 简单但功能强大的线程模型 • 无连接的DatagramSocket支持 • 链逻辑，易于重用
Ease of Use(易于使用)	• 提供大量的文档和例子 • 除了依赖jdk1.6+，没有额外的依赖关系。某些功能依赖jdk1.7+，其他特性可能有相关依赖，但都是可选的。
Performance(性能)	• 比Java APIS更好的吞吐量和更低的延迟 • 因为线程池和重用所有消耗较少的资源 • 尽量减少不必要的内存拷贝
Robustness(鲁棒性)	鲁棒性，可以理解为健壮性 • 链接快或慢或超载不会导致更多的OutOfMemoryError • 在高速的网络程序中不会有不公平的read/write
Security(安全性)	• 完整的SSL/TLS和StartTLS支持 • 可以在如Applet或OSGI这些受限制的环境中运行
Community(社区)	• 版本发布频繁 • 社区活跃

除了列出的功能外，Netty为Java NIO中的bug和限制也提供了解决方案。我们需要深刻理解Netty的功能以及它的异步处理

机制和它的架构。NIO和Netty都大量使用了异步代码，并且封装的很好，我们无需了解底层的事件选择机制。下面我们来看看为什么需要异步APIS。

1.2 异步设计

整个Netty的API都是异步的，异步处理不是一个新的机制，这个机制出来已经有一些时间了。对网络应用来说，IO一般是性能的瓶颈，使用异步IO可以较大程度上提高程序性能，因为异步变的越来越重要。但是它是如何工作的呢？以及有哪些不同的模式可用呢？

异步处理提倡更有效的使用资源，它允许你创建一个任务，当有事件发生时将获得通知并等待事件完成。这样就不会阻塞，不管事件完成与否都会及时返回，资源利用率更高，程序可以利用剩余的资源做一些其他的事情。

本节将说明一起工作或实现异步API的两个最常用的方法，并讨论这些技术之间的差异。

1.2.1 Callbacks(回调)

回调一般是异步处理的一种技术。一个回调是被传递到并且执行完该方法。你可能认为这种模式来自JavaScript，在Javascript中，回调是它的核心。下面的代码显示了如何使用这种技术来获取数据。下面代码是一个简单的回调

[java]

```java
package netty.in.action;

public class Worker {

    public void doWork() {
        Fetcher fetcher = new MyFetcher(new Data(1, 0));
        fetcher.fetchData(new FetcherCallback() {
            @Override
            public void onError(Throwable cause) {
                System.out.println("An error accour: " + cause.getMessage());
            }

            @Override
            public void onData(Data data) {
                System.out.println("Data received: " + data);
            }
        });
    }

    public static void main(String[] args) {
        Worker w = new Worker();
        w.doWork();
    }

}
```

[java]

```java
package netty.in.action;

public interface Fetcher {
    void fetchData(FetcherCallback callback);
}
```

[java]

```java
package netty.in.action;

public class MyFetcher implements Fetcher {

    final Data data;

    public MyFetcher(Data data){
        this.data = data;
    }

    @Override
    public void fetchData(FetcherCallback callback) {
        try {
            callback.onData(data);
        } catch (Exception e) {
            callback.onError(e);
        }
```

```
    }

}
```

```java
package netty.in.action;

public interface FetcherCallback {
    void onData(Data data) throws Exception;
    void onError(Throwable cause);
}
```

```java
package netty.in.action;

public class Data {

    private int n;
    private int m;

    public Data(int n,int m){
        this.n = n;
        this.m = m;
    }

    @Override
    public String toString() {
        int r = n/m;
        return n + "/" + m +" = " + r;
    }
}
```

上面的例子只是一个简单的模拟回调，要明白其所表达的含义。Fetcher.fetchData()方法需传递一个FetcherCallback类型的参数，当获得数据或发生错误时被回调。对于每种情况都提供了同意的方法：

- FetcherCallback.onData()，将接收数据时被调用
- FetcherCallback.onError()，发生错误时被调用

因为可以将这些方法的执行从"caller"线程移动到其他的线程执行；但也不会保证FetcherCallback的每个方法都会被执行。回调过程有个问题就是当你使用链式调用
很多不同的方法会导致线性代码；有些人认为这种链式调用方法会导致代码难以阅读，但是我认为这是一种风格和习惯问题。例如，基于Javascript的Node.js越来越受欢迎，它使用了大量的回调，许多人都认为它的这种方式利于阅读和编写。

1.2.2 Futures

第二种技术是使用Futures。Futures是一个抽象的概念，它表示一个值，该值可能在某一点变得可用。一个Future要么获得计算完的结果，要么获得计算失败后的异常。Java在java.util.concurrent包中附带了Future接口，它使用Executor异步执行。例如下面的代码，每传递一个Runnable对象到ExecutorService.submit()方法就会得到一个回调的Future，你能使用它检测是否执行完成。

```java
package netty.in.action;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class FutureExample {

    public static void main(String[] args) throws Exception {
        ExecutorService executor = Executors.newCachedThreadPool();
        Runnable task1 = new Runnable() {
            @Override
            public void run() {
                //do something
                System.out.println("i am task1.....");
```

```
            }
        };
        Callable<Integer> task2 = new Callable<Integer>() {
            @Override
            public Integer call() throws Exception {
                //do something
                return new Integer(100);
            }
        };
        Future<?> f1 = executor.submit(task1);
        Future<Integer> f2 = executor.submit(task2);
        System.out.println("task1 is completed? " + f1.isDone());
        System.out.println("task2 is completed? " + f2.isDone());
        //waiting task1 completed
        while(f1.isDone()){
            System.out.println("task1 completed.");
            break;
        }
        //waiting task2 completed
        while(f2.isDone()){
            System.out.println("return value by task2: " + f2.get());
            break;
        }
    }

}
```

有时候使用Future感觉很丑陋，因为你需要间隔检查Future是否已完成，而使用回调会直接收到返回通知。看完这两个常用的异步执行技术后，你可能想知道使用哪个最好？这里没有明确的答案。事实上，Netty两者都使用，提供两全其美的方案。下一节将在JVM上首先使用阻塞，然后再使用NIO和NIO2写一个网络程序。这些是本书后续章节必不可少的基础知识，如果你熟悉Java网络AIPs，你可以快速翻阅即可。

1.3 Java中的Blocking和non-blocking IO对比

本节主要讲解Java的IO和NIO的差异，这里不过多赘述，网络已有很多相关文章。

1.4 NIO的问题和Netty中是如何解决这些问题的

本节中将介绍Netty是如何解决NIO中的一些问题和限制。Java的NIO相对老的IO APIs有着非常大的进步，但是使用NIO是受限制的。这些问题往往是设计的问题，有些是缺陷知道的。

1.4.1 跨平台和兼容性问题

NIO是一个比较底层的APIs，它依赖于操作系统的IO APIs。Java实现了统一的接口来操作IO，其在所有操作系统中的工作行为是一样的，这是很伟大的。使用NIO会经常发现代码在Linux上正常运行，但在Windows上就会出现问题。我建议你如果使用NIO编写程序，就应该在所有的操作系统上进行测试来支持，使程序可以在任何操作系统上正常运行；即使在所有的Linux系统上都测试通过了，也要在其他的操作系统上进行测试；你若不验证，以后就可能会出问题。

NIO2看起来很理想，但是NIO2只支持Jdk1.7+，若你的程序在Java1.6上运行，则无法使用NIO2。另外，Java7的NIO2中没有提供DatagramSocket的支持，所以NIO2只支持TCP程序，不支持UDP程序。

Netty提供一个统一的接口，同一语义无论在Java6还是Java7的环境下都是可以运行的，开发者无需关心底层APIs就可以轻松实现相关功能。

1.4.2 扩展ByteBuffer

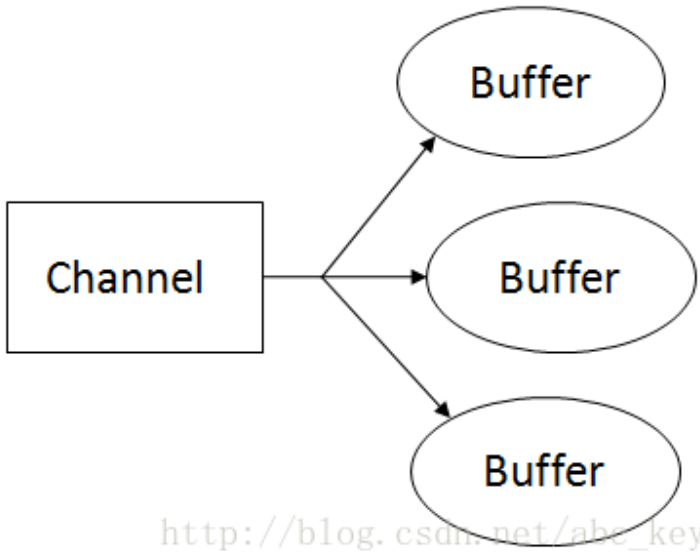
ByteBuffer是一个数据容器，但是可惜的是JDK没有开发ByteBuffer实现的源码；ByteBuffer允许包装一个byte[]来获得一个实例，如果你希望尽量减少内存拷贝，那么这种方式是非常有用的。若果你想将ByteBuffer重新实现，那么不要浪费你的时间了，ByteBuffer的构造函数是私有的，所以它不能被扩展。Netty提供了自己的ByteBuffer实现，Netty通过一些简单的APIs对ByteBuffer进行构造、使用和操作，以此来解决NIO中的一些限制。

1.4.3 NIO对缓冲区的聚合和分散操作可能会操作内存泄露

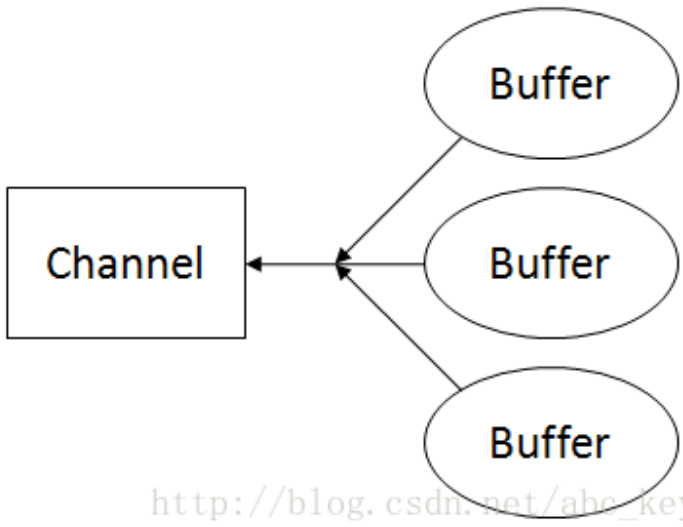
很多Channel的实现支持Gather和Scatter。这个功能允许从从多个ByteBuffer中读入或写入到过个ByteBuffer，这样做可以提供性能。操作系统底层知道如何处理这些被写入/读出，并且能以最有效的方式处理。如果要分割的数据再多个不同的ByteBuffer中，使用Gather/Scatter是比较好的方式。

例如，你可能希望header在一个ByteBuffer中，而body在另外的ByteBuffer中；

下图显示的是Scatter(分散)，将ScatteringByteBuffer中的数据分散读取到多个ByteBuffer中：



下图显示的是Gather(聚合)，将多个ByteBuffer的数据写入到GatheringByteChannel：



可惜Gather/Scatter功能会导致内存泄露，知道Java7才解决内存泄露问题。使用这个功能必须小心编码和Java版本。

1.4.4 Squashing the famous epoll bug

压碎著名的epoll缺陷。

On Linux-like OSs the selector makes use of the epoll- IO event notification facility. This is a high-performance technique in which the OS works asynchronously with the networking stack.Unfortunately, even today the "famous" epoll- bug can lead to an "invalid" state in the selector, resulting in 100% CPU-usage and spinning. The only way to recover is to recycle the old selector and transfer the previously registered Channel instances to the newly created Selector.

Linux-like OSs的选择器使用的是epoll-IO事件通知工具。这是一个在操作系统以异步方式工作的网络stack.Unfortunately，即使是现在，著名的epoll-bug也可能会导致无效的状态的选择和100%的CPU利用率。要解决epoll-bug的唯一方法是回收旧的选择器，将先前注册的通道实例转移到新创建的选择器上。

What happens here is that the Selector.select() method stops to block and returns immediately-even if there are no selected SelectionKeys present. This is against the contract, which is in the Javadocs of the Selector.select() method:Selector.select() must not unblock if nothing is selected.

这里发生的是，不管有没有已选择的SelectionKey，Selector.select()方法总是不会阻塞并且会立刻返回。这违反了Javadoc中对Selector.select()方法的描述，Javadoc中的描述：Selector.select() must not unblock if nothing is selected.
(Selector.select()方法若未选中任何事件将会阻塞。)

The range of solutions to this epoll- problem is limited, but Netty attempts to automatically detect and prevent it. The following listing is an example of the epoll- bug.

NIO中对epoll问题的解决方案是有限制的，Netty提供了更好的解决方案。下面是epoll-bug的一个例子：

```
...
while (true) {
int selected = selector.select();
Set<SelectedKeys> readyKeys = selector.selectedKeys();
Iterator iterator = readyKeys.iterator();
while (iterator.hasNext()) {
...
...
```

```
}
}
...
```

The effect of this code is that the while loop eats CPU:

这段代码的作用是while循环消耗CPU：

```
...
while (true) {
}
...
```

The value will never be false, and the code keeps your CPU spinning and eats resources. This can have some undesirable side effects as it can consume all of your CPU, preventing any other CPU-bound work.

该值将永远是假的，代码将持续消耗你的CPU资源。这会有一些副作用，因为CPU消耗完了就无法再去做其他任何的工作。

These are only a few of the possible problems you may see while using non-blocking IO. Unfortunately, even after years of development in this area, issues still need to be resolved; thankfully, Netty addresses them for you.

这些仅仅是在使用NIO时可能会出现的一些问题。不幸的是，虽然在这个领域发展了多年，问题依然存在；幸运的是，Netty给了你解决方案。

1.5 Summary

This chapter provided an overview of Netty's features, design and benefits. I discussed the difference between blocking and non-blocking processing to give you a fundamental understanding of the reasons to use a non-blocking framework. You learned how to use the JDK API to write network code in both blocking and non-blocking modes. This included the new non-blocking API, which comes with JDK 7. After seeing the NIO APIs in action, it was also important to understand some of the known issues that you may run into. In fact, this is why so many people use Netty: to take care of workarounds and other JVM quirks. In the next chapter, you'll learn the basics of the Netty API and programming model, and, finally, use Netty to write some useful code.