

NIO 与 Netty 编程

课程大纲

- 多线程编程
 - 基本知识回顾
 - 线程安全
 - 线程间通信
- BIO 编程
- NIO 编程
 - 概述
 - 文件 IO
 - 网络 IO
 - AIO 和 IO 总结
- Netty 框架
 - 概述
 - 核心 API
 - 入门案例
 - 网络聊天案例
 - 编码和解码
- 自定义 RPC
 - 整体分析
 - 设计和实现

一. 多线程编程

1.1 基本知识回顾

线程是比进程更小的能独立运行的基本单位，它是进程的一部分，一个进程可以拥有多个线程，但至少要有个线程，即主执行线程(Java 的 `main` 方法)。我们既可以编写单线程应用，也可以编写多线程应用。

一个进程中的多个线程可以并发(同时)执行，在一些执行时间长、需要等待的任务上(例如：文件读写和网络传输等)，多线程就比较有用了。

怎么理解多线程呢？来两个例子：

1. 进程就是一个工厂，一个线程就是工厂中的一条生产线，一个工厂至少有一条生产线，只有一条生产线就是单线程应用，拥有多条生产线就是多线程应用。多条生产线可以同时运行。
2. 我们使用迅雷可以同时下载多个视频，迅雷就是进程，多个下载任务就是线程，这几个线程可以同时运行去下载视频。

多线程可以共享内存、充分利用 CPU，通过提高资源(内存和 CPU)使用率从而提高程序的执行效率。CPU 使用抢占式调度模式在多个线程间进行着随机的高速的切换。对于 CPU 的一个核而言，某个时刻，只能执行一个线程，而 CPU 在多个线程间的切换速度相对我们的感觉要快很多，看上去就像是多个线程或任务在同时运行。

Java 天生就支持多线程并提供了两种编程方式，一个是继承 `Thread` 类，一个是实现 `Runnable` 接口，接下来咱们通过两个案例快速复习回顾一下。

方式一：继承 `Thread` 类

```
public class ThreadFor1 extends Thread{
    public void run() {
        for (int i = 0; i < 50; i++) {
            System.out.println(this.getName()+":"+i);
        }
    }
}
```

上述代码自定义了一个类去继承 `Thread` 类，并重写了 `run` 方法，在该方法内实现具体业务功能，这里用一个 `for` 循环模拟一下。

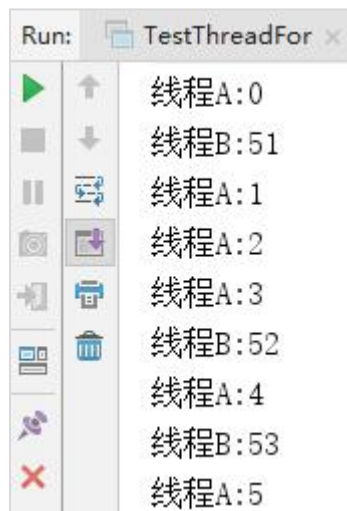
```
public class ThreadFor2 extends Thread{
    public void run() {
        for (int i = 51; i < 100; i++) {
            System.out.println(this.getName()+":"+i);
        }
    }
}
```

```
}
```

上述代码又自定义了一个类去继承 **Thread** 类，并重写了 **run** 方法，在该方法内实现另一个业务功能，这里仍用一个 **for** 循环模拟一下。

```
public class TestThreadFor {  
    public static void main(String[] args) {  
        ThreadFor1 tf1=new ThreadFor1();  
        tf1.setName("线程 A");  
        ThreadFor2 tf2=new ThreadFor2();  
        tf2.setName("线程 B");  
  
        tf1.start();  
        tf2.start();  
    }  
}
```

上述代码创建了两个线程对象并分别启动，运行效果如下图所示，我们能够清晰观察到，CPU 在两个线程之间快速随机切换，也就是我们平时说的在同时运行。



方式二：实现 **Runnable** 接口

```
public class RunnableFor1 implements Runnable{  
    public void run() {  
        for (int i = 0; i < 50; i++) {  
            System.out.println(Thread.currentThread().getName()+"-"+i);  
        }  
    }  
}
```

上述代码自定义一个类去实现 **Runnable** 接口，并实现了 **run** 方法，在该方法内实现具体业务功能，这里用一个 **for** 循环模拟一下。

```
public class RunnableFor2 implements Runnable{  
    public void run() {  
        for (int i = 51; i < 100; i++) {  
            System.out.println(Thread.currentThread().getName()+"-"+i);  
        }  
    }  
}
```

```

    }
}

```

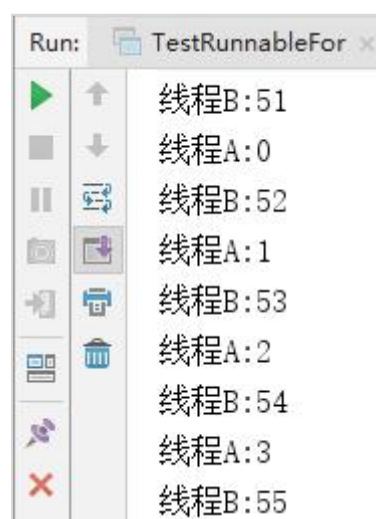
上述代码又自定义一个类去实现 `Runnable` 接口，并实现了 `run` 方法，在该方法内实现另一个业务功能，这里仍用一个 `for` 循环模拟一下。

```

public class TestRunnableFor {
    public static void main(String[] args) {
        Thread t1=new Thread(new RunnableFor1());
        t1.setName("线程 A");
        Thread t2=new Thread(new RunnableFor2());
        t2.setName("线程 B");
        t1.start();
        t2.start();
    }
}

```

上述代码创建两个线程对象并分别启动，运行效果如下图所示，我们能够清晰观察到，CPU 在两个线程之间快速随机切换，也就是我们平时说的在同时运行。



1.2 线程安全

1.2.1 产生线程安全问题的原因

在进行多线程编程时，要注意线程安全问题，我们先通过一个案例了解一下什么是线程安全问题。该案例模拟用两个售票窗口同时卖火车票，具体代码如下所示：

```

public class SaleWindow implements Runnable {
    private int id = 10;    //表示 10 张火车票    这是共享资源

    //卖 10 张火车票
    public void run() {
        for (int i = 0; i < 10; i++) {

```

```

        if (id > 0) {
            System.out.println(Thread.currentThread().getName()
                               + "卖了编号为" + id + "的火车票");
            id--;
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
            }
        }
    }
}

```

上述代码定义了一个类去实现 `Runnable` 接口，该类中有一个属性 `id`，表示 10 张火车票，并在 `run` 方法中通过一个 `for` 循环销售火车票，为了让效果明显一些，中间用 `sleep` 方法停顿半秒钟。

```

public class TestSaleWindow {
    public static void main(String[] args) {
        SaleWindow sw=new SaleWindow();

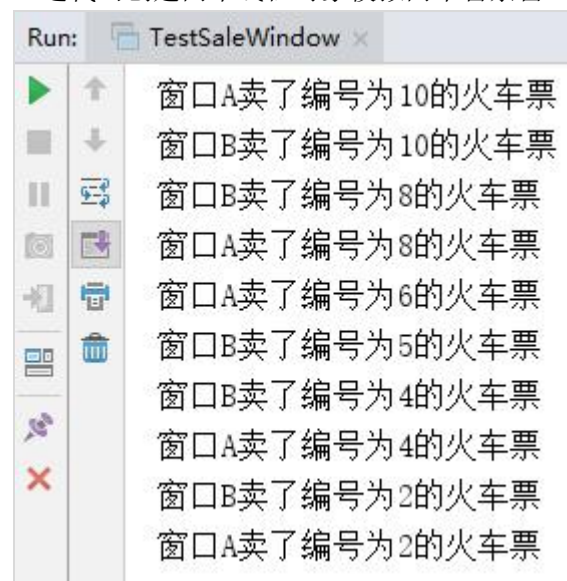
        Thread t1=new Thread(sw);
        Thread t2=new Thread(sw);

        t1.setName("窗口 A");
        t2.setName("窗口 B");

        t1.start();
        t2.start();
    }
}

```

上述代码创建两个线程对象模拟两个售票窗口同时卖票，运行结果如下图所示：



我们看到，10 张火车票都卖出去了，但是出现了重复售票，这就是线程安全问题造成的。这 10 张火车票是共享资源，也就是说任何窗口都可以进行操作和销售，问题在于窗口 A 把某一张火车票卖出去之后，窗口 B 并不知道，因为这是两个线程，所以窗口 B 也可能会再卖出去一张相同的火车票。

多个线程操作的是同一个共享资源，但是线程之间是彼此独立、互相隔绝的，因此就会出现数据（共享资源）不能同步更新的情况，这就是线程安全问题。

1.2.2 解决线程安全问题

Java 中提供了一个同步机制(锁)来解决线程安全问题，即让操作共享数据的代码在某一段时间，只被一个线程执行(锁住)，在执行过程中，其他线程不可以参与进来，这样共享数据就能同步了。简单来说，就是给某些代码加把锁。

锁是什么？又从哪儿来呢？锁的专业名称叫监视器 **monitor**，其实 Java 为每个对象都自动内置了一个锁（监视器 **monitor**），当某个线程执行到某代码块时就会自动得到这个对象的锁，那么其他线程就无法执行该代码块了，一直要等到之前那个线程停止(释放锁)。需要特别注意的是：多个线程必须使用同一把锁（对象）。

Java 的同步机制提供了两种实现方式：

- 同步代码块：即给代码块上锁，变成同步代码块
- 同步方法：即给方法上锁，变成同步方法

接下来我们分别用这两种方式解决卖火车票案例的线程安全问题，其实这两种方式本质上差不多，都是通过 **synchronized** 关键字来实现的。

方式一：同步代码块

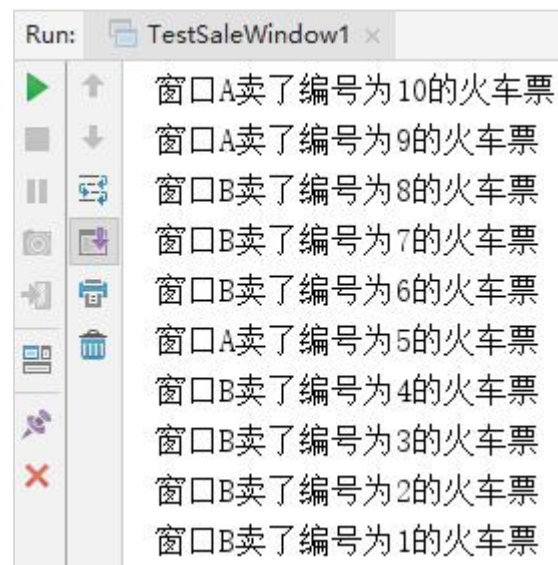
```
public class SaleWindow1 implements Runnable {  
    private int id = 10;    //表示 10 张火车票    共享资源  
  
    //卖 10 张火车票  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            synchronized (this) {  
                if (id > 0) {  
                    System.out.println(Thread.currentThread().getName()  
                        + "卖了编号为" + id + "的火车票");  
                    id--;  
                    try {  
                        Thread.sleep(500);  
                    } catch (InterruptedException e) {}  
                }  
            }  
        }  
    }  
}
```

```
}
```

其实大家也发现了，上述代码和之前的只有一点差别，就是多了红色部分的代码，同步代码块的语法是：`synchronized(锁){...业务代码...}`。

```
public class TestSaleWindow1 {  
    public static void main(String[] args) {  
        SaleWindow1 sw=new SaleWindow1();  
  
        Thread t1=new Thread(sw);  
        Thread t2=new Thread(sw);  
  
        t1.setName("窗口 A");  
        t2.setName("窗口 B");  
  
        t1.start();  
        t2.start();  
    }  
}
```

我们同样创建两个线程对象模拟两个售票窗口同时卖票，这次运行结果如下图所示：



我们看到 10 张火车票都卖出去了，这次没有问题，我们不关心这 10 张票都是哪个窗口卖出去的，我们关心的是没有重复卖票。

方式二：同步方法

```
public class SaleWindow2 implements Runnable {  
    private int id = 10; // 表示 10 张火车票 共享资源  
  
    public synchronized void saleOne(){ //该方法内是上面同步代码块中的代码  
        if (id > 0) {  
            System.out.println(Thread.currentThread().getName() + "卖了编号为"  
                + id + "的火车票");  
            id--;  
        }  
    }  
}
```



```

        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
    }
}

// 卖 10 张火车票
public void run() {
    for (int i = 0; i < 10; i++) {
        saleOne();
    }
}
}

```

第二种方式是把原来同步代码块中的代码抽取出来放到一个方法中，然后给这个方法加上 `synchronized` 关键字修饰，锁住的代码是一样的，因此本质上和第一种方式没什么区别。

```

public class TestSaleWindow2 {
    public static void main(String[] args) {
        SaleWindow2 sw=new SaleWindow2();

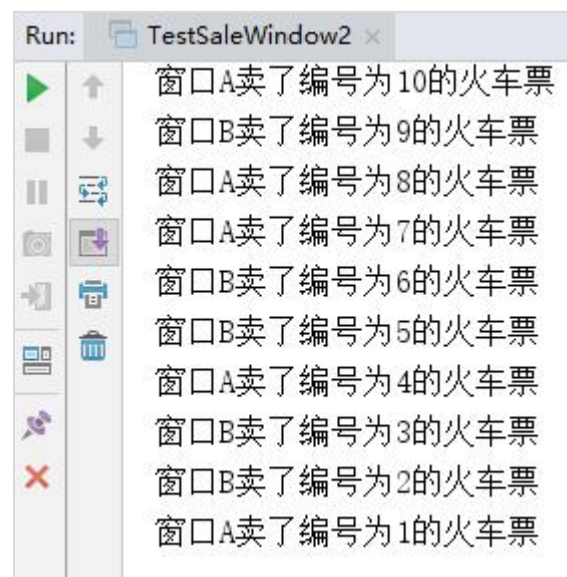
        Thread t1=new Thread(sw);
        Thread t2=new Thread(sw);

        t1.setName("窗口 A");
        t2.setName("窗口 B");

        t1.start();
        t2.start();
    }
}

```

我们同样还是创建两个线程对象模拟两个售票窗口同时卖票，这次运行结果如下图所示：



1.2.3 Java API 中的线程安全问题

我们平时在使用 Java API 进行编程时，经常遇到说哪个类是线程安全的，哪个类是不保证线程安全的，例如：StringBuffer / StringBuilder 和 Vector / ArrayList，谁是线程安全的？谁不是线程安全的？我们查一下它们的源码便可知晓。

```
StringBuffer.java x
418      @Override
419      public synchronized StringBuffer append(double d) {...}
424
425      /**...*/
429      @Override
430      public synchronized StringBuffer delete(int start, int end) {...}
435
436      /**...*/
440      @Override
441      public synchronized StringBuffer deleteCharAt(int index) {...}
446
447      /**...*/
451      @Override
452      public synchronized StringBuffer replace(int start, int end, String str) {...}
457
458      /**...*/
462      @Override
463      public synchronized String substring(int start) { return substring(start, count); }
```

```
StringBuilder.java x
225      public StringBuilder append(double d) {...}
229
230      /**...*/
233      @Override
234      public StringBuilder appendCodePoint(int codePoint) {...}
238
239      /**...*/
242      @Override
243      public StringBuilder delete(int start, int end) {...}
247
248      /**...*/
251      @Override
252      public StringBuilder deleteCharAt(int index) {...}
256
257      /**...*/
260      @Override
261      public StringBuilder replace(int start, int end, String str) {...}
265
266      /**...*/
269      @Override
270      public StringBuilder insert(int index, char[] str, int offset,
```

```
C:\Vector.java x
736
737 /**...*/
746 @ public synchronized E get(int index) {...}
752
753 /**...*/
764 @ public synchronized E set(int index, E element) {...}
772
773 /**...*/
780 @ public synchronized boolean add(E e) {...}
786
787 /**...*/
798 @ public boolean remove(Object o) { return removeElement(o); }
801
802 /**...*/
813 @ public void add(int index, E element) { insertElementAt(element, index); }
816
817 /**...*/
828 @ public synchronized E remove(int index) {...}
842
843 /**...*/
849 @ public void clear() { removeAllElements(); }
```

```
C:\ArrayList.java x
420
421 /**...*/
428 @ public E get(int index) {...}
433
434 /**...*/
443 @ public E set(int index, E element) {...}
450
451 /**...*/
457 @ public boolean add(E e) {...}
462
463 /**...*/
472 @ public void add(int index, E element) {...}
481
482 /**...*/
491 @ public E remove(int index) {...}
505
506 /**...*/
519 @ public boolean remove(Object o) {...}
```

通过查看源码，我们发现 `StringBuffer` 和 `Vector` 类中的大部分方法都是同步方法，所以证明

这两个类在使用时是保证线程安全的;而 `StringBuilder` 和 `ArrayList` 类中的方法都是普通方法,没有使用 `synchronized` 关键字进行修饰,所以证明这两个类在使用时不保证线程安全。线程安全和性能之间不可兼得,保证线程安全就会损失性能,保证性能就不能满足线程安全。

1.3 线程间通信

多个线程并发执行时,在默认情况下 CPU 是随机性的在线程之间进行切换的,但是有时候我们希望它们能有规律的执行,那么,多线程之间就需要一些协调通信来改变或控制 CPU 的随机性。Java 提供了等待唤醒机制来解决这个问题,具体来说就是多个线程依靠一个同步锁,然后借助于 `wait()`和 `notify()`方法就可以实现线程间的协调通信。

同步锁相当于中间人的作用,多个线程必须用同一个同步锁(认识同一个中间人),只有同一个锁上的被等待的线程,才可以被持有该锁的另一个线程唤醒,使用不同锁的线程之间不能相互唤醒,也就无法协调通信。

Java 在 `Object` 类中提供了一些方法可以用来实现线程间的协调通信,我们一起来了解一下:

- `public final void wait();` 让当前线程释放锁
- `public final native void wait(long timeout);` 让当前线程释放锁,并等待 xx 毫秒
- `public final native void notify();` 唤醒持有同一锁的某个线程
- `public final native void notifyAll();` 唤醒持有同一锁的所有线程

需要注意的是:在调用 `wait` 和 `notify` 方法时,当前线程必须已经持有锁,然后才可以调用,否则将会抛出 `IllegalMonitorStateException` 异常。接下来咱们通过两个案例来演示一下具体如何编程实现线程间通信。

案例 1: 一个线程输出 10 次 1,一个线程输出 10 次 2,要求交替输出“1 2 1 2 1 2...”或“2 1 2 1 2 1...”

```
public class MyLock {  
    // 锁  
    public static Object o=new Object();  
}
```

为了保证两个线程使用的一定是同一个锁,我们创建一个对象作为静态属性放到一个类中,这个对象就用来充当锁。

```
public class ThreadForNum1 extends Thread{  
    public void run() {  
        for (int i = 0; i < 11; i++) {  
            synchronized (MyLock.o) {  
                System.out.println("1");  
                MyLock.o.notify(); //唤醒另一个线程  
                try {  
                    MyLock.o.wait(); //让自己休眠并释放锁  
                } catch (InterruptedException e) {  
                }  
            }  
        }  
    }  
}
```

```

    }
}

```

该线程输出十次 1，使用 **MyLock.o** 作为锁，每输出一个 1 就唤醒另一个线程，然后自己休眠并释放锁。

```

public class ThreadForNum2 extends Thread{
    public void run() {
        for (int i = 0; i < 10; i++) {
            synchronized (MyLock.o) {
                System.out.println("2");
                MyLock.o.notify(); //唤醒另一个线程
            } try {
                MyLock.o.wait(); //让自己休眠并释放锁
            } catch (InterruptedException e) {
            }
        }
    }
}

```

该线程输出十次 2，也使用 **MyLock.o** 作为锁，每输出一个 2 就唤醒另一个线程，然后自己休眠并释放锁。

```

public class TestNum {
    public static void main(String[] args) {
        new ThreadForNum1().start();
        new ThreadForNum2().start();
    }
}

```

我们创建两个线程对象分别运行，效果如下图所示：



案例 2：生产者消费者模式

该模式在现实生活中很常见，在项目开发中也广泛应用，它是线程间通信的经典应用。生产者是一堆线程，消费者是另一堆线程，内存缓冲区可以使用 `List` 集合存储数据。该模式的关键之处是如何处理多线程之间的协调通信，内存缓冲区为空的时候，消费者必须等待，而内存缓冲区满的时候，生产者必须等待，其他时候可以是个动态平衡。

下面的案例模拟实现农夫采摘水果放到筐里，小孩从筐里拿水果吃，农夫是一个线程，小孩是一个线程，水果筐放满了，农夫停；水果筐空了，小孩停。

```
public class Kuang {  
    //这个集合就是水果筐 假设最多存 10 个水果  
    public static ArrayList<String> kuang=new ArrayList<String>();  
}
```

上述代码定义一个静态集合作为内存缓冲区用来存储数据，同时这个集合也可以作为锁去被多个线程使用。

```
public class Farmer extends Thread {  
    public void run() {  
        while (true) {  
            synchronized (Kuang.kuang) {  
                //1.筐放满了就让农夫休息  
                if (Kuang.kuang.size() == 10) {  
                    try {  
                        Kuang.kuang.wait();  
                    } catch (InterruptedException e) {  
                    }  
                }  
                //2.往筐里放水果  
                Kuang.kuang.add("apple");  
                System.out.println("农夫放了一个水果,目前筐里有" + Kuang.kuang.size()  
                    + "个水果");  
                //3.唤醒小孩继续吃  
                Kuang.kuang.notify();  
            }  
            //4.模拟控制速度  
            try {  
                Thread.sleep(100);  
            } catch (InterruptedException e) {  
            }  
        }  
    }  
}
```

上述代码就是农夫线程，不断的往集合（筐）里放水果，当筐满了就停，同时释放锁。

```
public class Child extends Thread {  
    public void run() {  
        while (true) {
```

```

        synchronized (Kuang.kuang) {
            //1.筐里没水果了就让小孩休息
            if (Kuang.kuang.size() == 0) {
                try {
                    Kuang.kuang.wait();
                } catch (InterruptedException e) {
                }
            }

            //2.小孩吃水果
            Kuang.kuang.remove("apple");
            System.out.println("小孩吃了一个水果,目前筐里有" + Kuang.kuang.size() + "个水果");
            //3.唤醒农夫继续放水果
            Kuang.kuang.notify();
        }

        //4.模拟控制速度
        try {
            Thread.sleep(400);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

上述代码是小孩线程，不断的从集合（筐）里拿水果吃，当筐空了就停，同时释放锁。

```

public class TestFarmerChild {
    public static void main(String[] args) {
        new Farmer().start();
        new Child().start();
    }
}

```

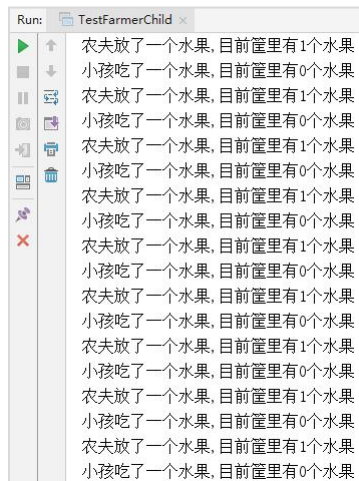
我们创建两个线程同时运行，可以通过双方线程里的 **sleep** 方法模拟控制速度，当农夫往框里放水果的速度快于小孩吃水果的速度时，运行效果如下图所示：

```

Run: TestFarmerChild
农夫放了一个水果, 目前筐里有1个水果
农夫放了一个水果, 目前筐里有2个水果
农夫放了一个水果, 目前筐里有3个水果
小孩吃了一个水果, 目前筐里有2个水果
农夫放了一个水果, 目前筐里有3个水果
农夫放了一个水果, 目前筐里有4个水果
农夫放了一个水果, 目前筐里有5个水果
农夫放了一个水果, 目前筐里有6个水果
小孩吃了一个水果, 目前筐里有5个水果
农夫放了一个水果, 目前筐里有6个水果
农夫放了一个水果, 目前筐里有7个水果
农夫放了一个水果, 目前筐里有8个水果
农夫放了一个水果, 目前筐里有9个水果
小孩吃了一个水果, 目前筐里有8个水果
农夫放了一个水果, 目前筐里有9个水果
农夫放了一个水果, 目前筐里有10个水果
小孩吃了一个水果, 目前筐里有9个水果
农夫放了一个水果, 目前筐里有10个水果
小孩吃了一个水果, 目前筐里有9个水果

```

当小孩吃水果的速度快于农夫往框里放水管的速度时，运行效果如下图所示：



二. BIO 编程

BIO 有的称之为 **basic**(基本) IO,有的称之为 **block**(阻塞) IO,主要应用于文件 IO 和网络 IO,这里不再说文件 IO, 大家对此都非常熟悉, 本次课程主要讲解网络 IO。

在 JDK1.4 之前, 我们建立网络连接的时候只能采用 BIO, 需要先在服务端启动一个 **ServerSocket**, 然后在客户端启动 **Socket** 来对服务端进行通信, 默认情况下服务端需要对每个请求建立一个线程等待请求, 而客户端发送请求后, 先咨询服务端是否有线程响应, 如果没有则会一直等待或者遭到拒绝, 如果有的话, 客户端线程会等待请求结束后才继续执行, 这就是阻塞式 IO。

接下来通过一个例子复习回顾一下 BIO 的基本用法（基于 TCP）。

```
//BIO 服务器端程序
public class TCPServer {
    public static void main(String[] args) throws Exception {
        //1.创建 ServerSocket 对象
        ServerSocket ss=new ServerSocket(9999);

        while (true) {
            //2.监听客户端
            Socket s = ss.accept(); //阻塞
            //3.从连接中取出输入流来接收消息
            InputStream is = s.getInputStream(); //阻塞
            byte[] b = new byte[10];
            is.read(b);
            String clientIP = s.getInetAddress().getHostAddress();
            System.out.println(clientIP + "说:" + new String(b).trim());
            //4.从连接中取出输出流并回话
            OutputStream os = s.getOutputStream();
            os.write("没钱".getBytes());
            //5.关闭
```



```

        s.close();
    }
}
}

```

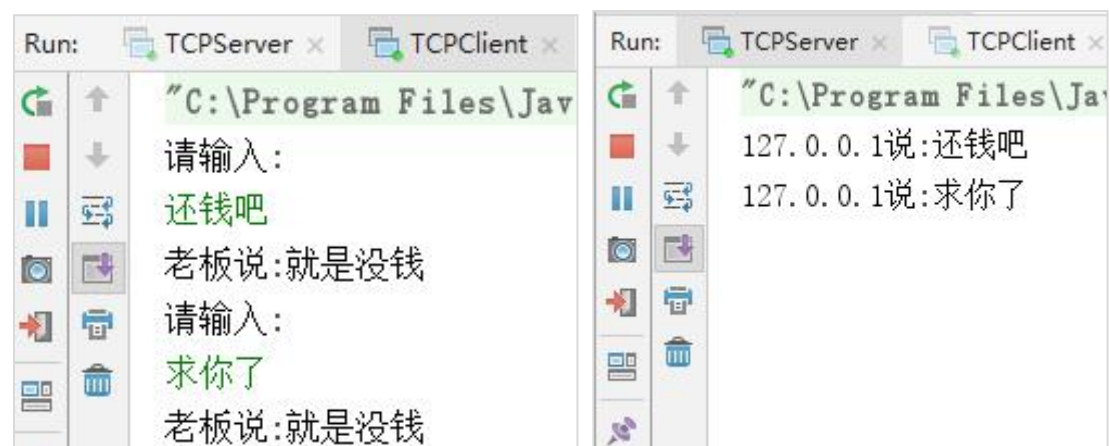
上述代码编写了一个服务器端程序，绑定端口号 9999，accept 方法用来监听客户端连接，如果没有客户端连接，就一直等待，程序会阻塞到这里。

```

//BIO 客户端程序
public class TCPClient {
    public static void main(String[] args) throws Exception {
        while (true) {
            //1.创建 Socket 对象
            Socket s = new Socket("127.0.0.1", 9999);
            //2.从连接中取出输出流并发消息
            OutputStream os = s.getOutputStream();
            System.out.println("请输入:");
            Scanner sc = new Scanner(System.in);
            String msg = sc.nextLine();
            os.write(msg.getBytes());
            //3.从连接中取出输入流并接收回话
            InputStream is = s.getInputStream(); //阻塞
            byte[] b = new byte[20];
            is.read(b);
            System.out.println("老板说:" + new String(b).trim());
            //4.关闭
            s.close();
        }
    }
}
}

```









上述代码编写了一个客户端程序，通过 9999 端口连接服务器端，getInputStream 方法用来等待服务器端返回数据，如果没有返回，就一直等待，程序会阻塞到这里。运行效果如下图所示：



三. NIO 编程

3.1 概述

java.nio 全称 java non-blocking IO，是指 JDK 提供的新 API。从 JDK1.4 开始，Java 提供了一系列改进的输入/输出的新特性，被统称为 NIO(即 New IO)。新增了许多用于处理输入输出的类，这些类都被放在 java.nio 包及子包下，并且对原 java.io 包中的很多类进行改写，新增了满足 NIO 的功能。

- ▷  java.nio
- ▷  java.nio.channels
- ▷  java.nio.channels.spi
- ▷  java.nio.charset
- ▷  java.nio.charset.spi
- ▷  java.nio.file
- ▷  java.nio.file.attribute
- ▷  java.nio.file.spi

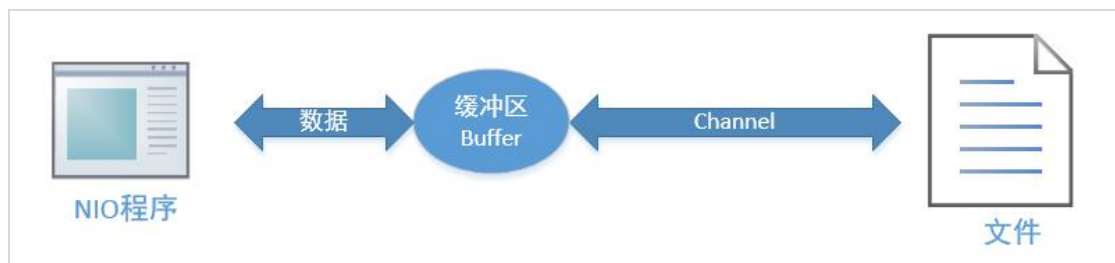
NIO 和 BIO 有着相同的目的和作用，但是它们的实现方式完全不同，BIO 以流的方式处理数据，而 NIO 以块的方式处理数据，块 I/O 的效率比流 I/O 高很多。另外，NIO 是非阻塞式的，这一点跟 BIO 也很不相同，使用它可以提供非阻塞式的高伸缩性网络。

NIO 主要有三大核心部分：Channel(通道), Buffer(缓冲区), Selector(选择器)。传统的 BIO 基于字节流和字符流进行操作，而 NIO 基于 Channel(通道)和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择器)用于监听多个通道的事件（比如：连接请求，数据到达等），因此使用单个线程就可以监听多个客户端通道。

3.2 文件 IO

3.2.1 概述和核心 API

缓冲区（Buffer）：实际上是一个容器，是一个特殊的数组，缓冲区对象内置了一些机制，能够跟踪和记录缓冲区的状态变化情况。Channel 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由 Buffer，如下图所示：



在 NIO 中，Buffer 是一个顶层父类，它是一个抽象类，常用的 Buffer 子类有：

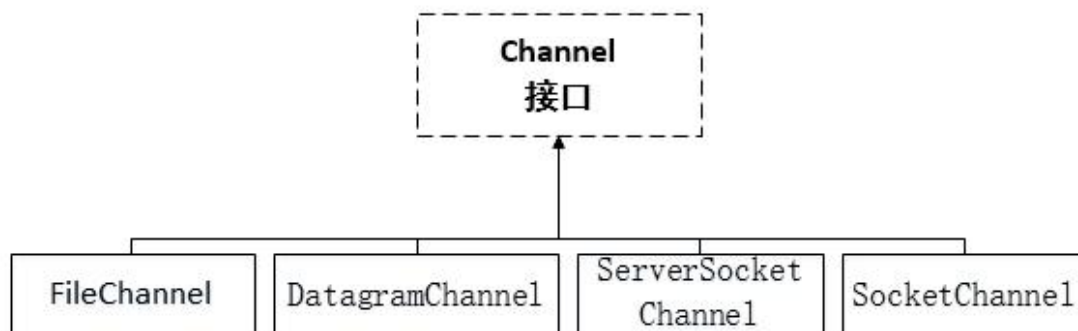
- ByteBuffer，存储字节数据到缓冲区
- ShortBuffer，存储字符串数据到缓冲区

- CharBuffer, 存储字符数据到缓冲区
- IntBuffer, 存储整数数据到缓冲区
- LongBuffer, 存储长整型数据到缓冲区
- DoubleBuffer, 存储小数到缓冲区
- FloatBuffer, 存储小数到缓冲区

对于 Java 中的基本数据类型, 都有一个 Buffer 类型与之相对应, 最常用的自然是 ByteBuffer 类(二进制数据), 该类的主要方法如下所示:

- public abstract ByteBuffer put(byte[] b); 存储字节数据到缓冲区
- public abstract byte[] get(); 从缓冲区获得字节数据
- public final byte[] array(); 把缓冲区数据转换成字节数组
- public static ByteBuffer allocate(int capacity); 设置缓冲区的初始容量
- public static ByteBuffer wrap(byte[] array); 把一个现成的数组放到缓冲区中使用
- public final Buffer flip(); 翻转缓冲区, 重置位置到初始位置

通道(Channel): 类似于 BIO 中的 stream, 例如 FileInputStream 对象, 用来建立到目标(文件, 网络套接字, 硬件设备等)的一个连接, 但是需要注意: BIO 中的 stream 是单向的, 例如 FileInputStream 对象只能进行读取数据的操作, 而 NIO 中的通道(Channel)是双向的, 既可以用来进行读操作, 也可以用来进行写操作。常用的 Channel 类有: FileChannel、DatagramChannel、ServerSocketChannel 和 SocketChannel。FileChannel 用于文件的数据读写, DatagramChannel 用于 UDP 的数据读写, ServerSocketChannel 和 SocketChannel 用于 TCP 的数据读写。



这里我们先讲解 FileChannel 类, 该类主要用来对本地文件进行 IO 操作, 主要方法如下所示:

- public int read(ByteBuffer dst) , 从通道读取数据并放到缓冲区中
- public int write(ByteBuffer src) , 把缓冲区的数据写到通道中
- public long transferFrom(ReadableByteChannel src, long position, long count), 从目标通道中复制数据到当前通道
- public long transferTo(long position, long count, WritableByteChannel target), 把数据从当前通道复制给目标通道

3.2.2 案例

接下来我们通过 NIO 实现几个案例, 分别演示一下本地文件的读、写和复制操作, 并和 BIO 做个对比。

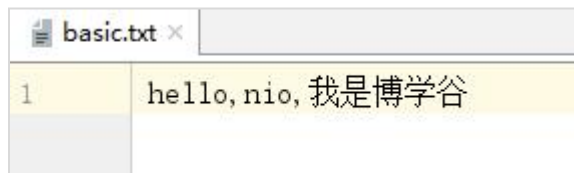
1. 往本地文件中写数据

```

@Test
public void test1() throws Exception{
    String str="hello,nio,我是博学谷";
    FileOutputStream fos=new FileOutputStream("basic.txt");
    FileChannel fc=fos.getChannel();
    ByteBuffer buffer=ByteBuffer.allocate(1024);
    buffer.put(str.getBytes());
    buffer.flip();
    fc.write(buffer);
    fos.close();
}

```

NIO 中的通道是从输出流对象里通过 `getChannel` 方法获取到的，该通道是双向的，既可以读，又可以写。在往通道里写数据之前，必须通过 `put` 方法把数据存到 `ByteBuffer` 中，然后通过通道的 `write` 方法写数据。在 `write` 之前，需要调用 `flip` 方法翻转缓冲区，把内部重置到初始位置，这样在接下来写数据时才能把所有数据写到通道里。运行效果如下图所示：



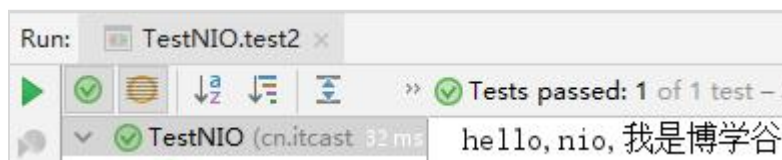
2. 从本地文件中读数据

```

@Test
public void test2() throws Exception{
    File file=new File("basic.txt");
    FileInputStream fis=new FileInputStream(file);
    FileChannel fc=fis.getChannel();
    ByteBuffer buffer=ByteBuffer.allocate((int)file.length());
    fc.read(buffer);
    System.out.print(new String(buffer.array()));
    fis.close();
}

```

上述代码从输入流中获得一个通道，然后提供 `ByteBuffer` 缓冲区，该缓冲区的初始容量和文件的大小一样，最后通过通道的 `read` 方法把数据读取出来并存储到了 `ByteBuffer` 中。运行效果如下图所示：



3. 复制文件

- 通过 BIO 复制一个视频文件，代码如下所示：

```

@Test
public void test3() throws Exception{

```

```

FileInputStream fis=new FileInputStream("C:\\Users\\zdx\\Desktop\\oracle.mov");
FileOutputStream fos=new FileOutputStream("d:\\oracle.mov");
byte[] b=new byte[1024];
while (true) {
    int res=fis.read(b);
    if(res==-1){
        break;
    }
    fos.write(b,0,res);
}
fis.close();
fos.close();
}

```

上述代码分别通过输入流和输出流实现了文件的复制，这是通过传统的 BIO 实现的，大家都比较熟悉，不再多说。

- 通过 NIO 复制相同的视频文件，代码如下所示：

```

@Test
public void test4() throws Exception{
    FileInputStream fis=new FileInputStream("C:\\Users\\zdx\\Desktop\\oracle.mov");
    FileOutputStream fos=new FileOutputStream("d:\\oracle.mov");
    FileChannel sourceCh = fis.getChannel();
    FileChannel destCh = fos.getChannel();
    destCh.transferFrom(sourceCh, 0, sourceCh.size());
    sourceCh.close();
    destCh.close();
}

```

上述代码分别从两个流中得到两个通道，sourceCh 负责读数据，destCh 负责写数据，然后直接调用 transferFrom 方法一步到位实现了文件复制。

3.3 网络 IO

3.3.1 概述和核心 API

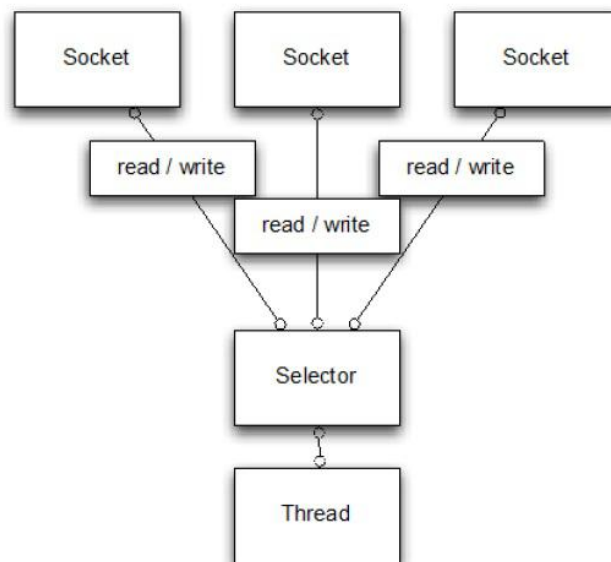
前面在进行文件 IO 时用到的 FileChannel 并不支持非阻塞操作，学习 NIO 主要就是进行网络 IO，Java NIO 中的网络通道是非阻塞 IO 的实现，基于事件驱动，非常适用于服务器需要维持大量连接，但是数据交换量不大的情况，例如一些即时通信的服务等等....

在 Java 中编写 Socket 服务器，通常有以下几种模式：

- 一个客户端连接用一个线程，优点：程序编写简单；缺点：如果连接非常多，分配的线程也会非常多，服务器可能会因为资源耗尽而崩溃。
- 把每一个客户端连接交给一个拥有固定数量线程的连接池，优点：程序编写相对简单，可以处理大量的连接。确定：线程的开销非常大，连接如果非常多，排队现象会比较严重。

- 使用 Java 的 NIO，用非阻塞的 IO 方式处理。这种模式可以用一个线程，处理大量的客户端连接。

1. Selector(选择器)，能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的处理。这样就可以只用一个单线程去管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。



该类的常用方法如下所示：

- `public static Selector open()`，得到一个选择器对象
- `public int select(long timeout)`，监控所有注册的通道，当其中有 IO 操作可以进行时，将对应的 `SelectionKey` 加入到内部集合中并返回，参数用来设置超时时间
- `public Set<SelectionKey> selectedKeys()`，从内部集合中得到所有的 `SelectionKey`

2. `SelectionKey`，代表了 `Selector` 和网络通道的注册关系，一共四种：

- `int OP_ACCEPT`：有新的网络连接可以 `accept`，值为 16
- `int OP_CONNECT`：代表连接已经建立，值为 8
- `int OP_READ` 和 `int OP_WRITE`：代表了读、写操作，值为 1 和 4

该类的常用方法如下所示：

- `public abstract Selector selector()`，得到与之关联的 `Selector` 对象
- `public abstract SelectableChannel channel()`，得到与之关联的通道
- `public final Object attachment()`，得到与之关联的共享数据
- `public abstract SelectionKey interestOps(int ops)`，设置或改变监听事件
- `public final boolean isAcceptable()`，是否可以 `accept`
- `public final boolean isReadable()`，是否可以读
- `public final boolean isWritable()`，是否可以写

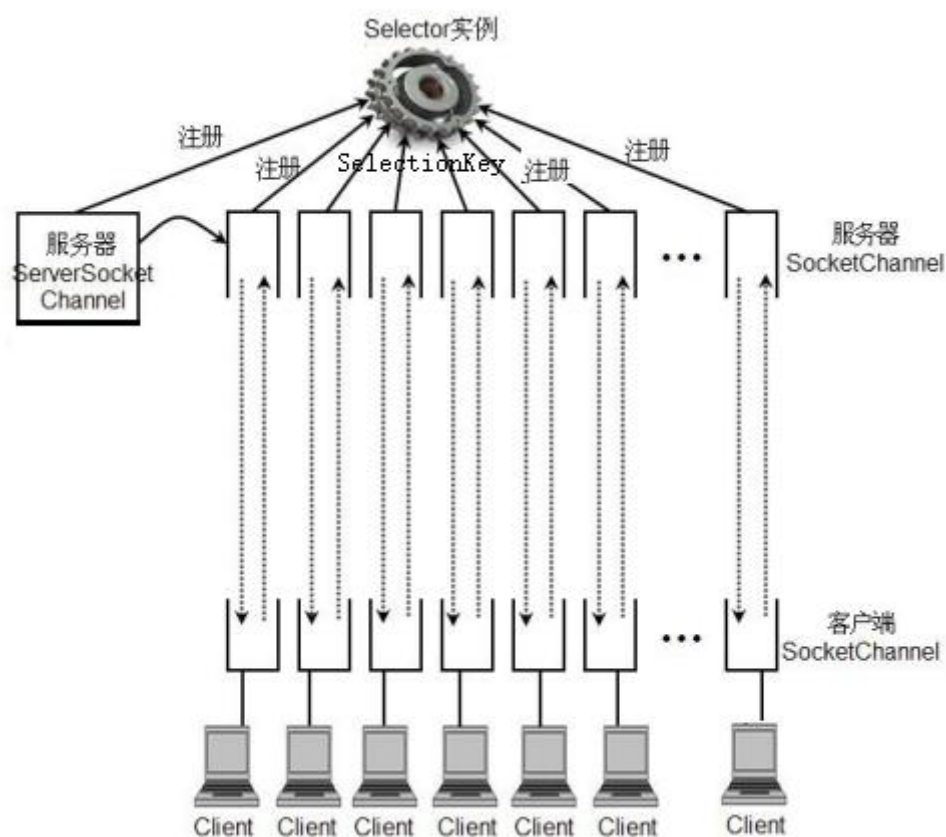
3. `ServerSocketChannel`，用来在服务器端监听新的客户端 `Socket` 连接，常用方法如下所示：

- `public static ServerSocketChannel open()`，得到一个 `ServerSocketChannel` 通道
- `public final ServerSocketChannel bind(SocketAddress local)`，设置服务器端口号

- `public final SelectableChannel configureBlocking(boolean block)`, 设置阻塞或非阻塞模式, 取值 `false` 表示采用非阻塞模式
- `public SocketChannel accept()`, 接受一个连接, 返回代表这个连接的通道对象
- `public final SelectionKey register(Selector sel, int ops)`, 注册一个选择器并设置监听事件

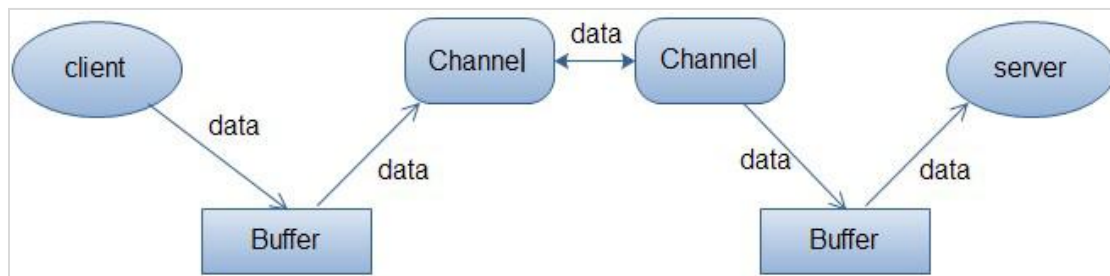
4. `SocketChannel`, 网络 IO 通道, 具体负责进行读写操作。NIO 总是把缓冲区的数据写入通道, 或者把通道里的数据读到缓冲区。常用方法如下所示:

- `public static SocketChannel open()`, 得到一个 `SocketChannel` 通道
- `public final SelectableChannel configureBlocking(boolean block)`, 设置阻塞或非阻塞模式, 取值 `false` 表示采用非阻塞模式
- `public boolean connect(SocketAddress remote)`, 连接服务器
- `public boolean finishConnect()`, 如果上面的方法连接失败, 接下来就要通过该方法完成连接操作
- `public int write(ByteBuffer src)`, 往通道里写数据
- `public int read(ByteBuffer dst)`, 从通道里读数据
- `public final SelectionKey register(Selector sel, int ops, Object att)`, 注册一个选择器并设置监听事件, 最后一个参数可以设置共享数据
- `public final void close()`, 关闭通道



3.3.2 入门案例

API 学习完毕后, 接下来我们使用 NIO 开发一个入门案例, 实现服务器端和客户端之间的数据通信 (非阻塞)。



//网络服务器端程序

```
public class NIOServer {
    public static void main(String[] args) throws Exception{
        //1. 得到一个 ServerSocketChannel 对象 老大
        ServerSocketChannel serverSocketChannel=ServerSocketChannel.open();
        //2. 得到一个 Selector 对象 间谍
        Selector selector=Selector.open();
        //3. 绑定一个端口号
        serverSocketChannel.bind(new InetSocketAddress(9999));
        //4. 设置非阻塞方式
        serverSocketChannel.configureBlocking(false);
        //5. 把 ServerSocketChannel 对象注册给 Selector 对象
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        //6. 干活
        while(true){
            //6.1 监控客户端
            if(selector.select(2000)==0){ //nio 非阻塞式的优势
                System.out.println("Server:没有客户端搭理我，我就干点别的事");
                continue;
            }
            //6.2 得到 SelectionKey,判断通道里的事件
            Iterator<SelectionKey> keyIterator=selector.selectedKeys().iterator();
            while(keyIterator.hasNext()){
                SelectionKey key=keyIterator.next();
                if(key.isAcceptable()){ //客户端连接请求事件
                    System.out.println("OP_ACCEPT");
                    SocketChannel socketChannel=serverSocketChannel.accept();
                    socketChannel.configureBlocking(false);
                    socketChannel.register(selector,SelectionKey.OP_READ, ByteBuffer.allocate(1024));
                }
                if(key.isReadable()){ //读取客户端数据事件
                    SocketChannel channel=(SocketChannel) key.channel();
                    ByteBuffer buffer=(ByteBuffer) key.attachment();
                    channel.read(buffer);
                    System.out.println("客户端发来数据: "+new String(buffer.array()));
                }
            }
            // 6.3 手动从集合中移除当前 key,防止重复处理
        }
    }
}
```



```

        keyIterator.remove();
    }
}
}
}

```

上面代码用 NIO 实现了一个服务器端程序，能不断接受客户端连接并读取客户端发过来的数据。

```

//网络客户端程序
public class NIOClient {
    public static void main(String[] args) throws Exception{
        //1. 得到一个网络通道
        SocketChannel channel=SocketChannel.open();
        //2. 设置非阻塞方式
        channel.configureBlocking(false);
        //3. 提供服务器端的 IP 地址和端口号
        InetSocketAddress address=new InetSocketAddress("127.0.0.1",9999);
        //4. 连接服务器端
        if(!channel.connect(address)){
            while(!channel.finishConnect()){ //nio 作为非阻塞式的优势
                System.out.println("Client:连接服务器端的同时，我还可以干别的一些事情");
            }
        }
        //5. 得到一个缓冲区并存入数据
        String msg="hello,Server";
        ByteBuffer writeBuf = ByteBuffer.wrap(msg.getBytes());
        //6. 发送数据
        channel.write(writeBuf);
        System.in.read();
    }
}

```

上面代码通过 NIO 实现了一个客户端程序，连接上服务器端后发送了一条数据，运行效果如下图所示：



3.3.3 网络聊天案例

刚才我们通过 NIO 实现了一个入门案例，基本了解了 NIO 的工作方式和运行流程，接下来我们用 NIO 实现一个多人聊天案例，具体代码如下所示：

```
public class ChatServer {  
    private Selector selector;  
    private ServerSocketChannel listenerChannel;  
    private static final int PORT = 9999; //服务器端口  
  
    public ChatServer() {  
        try {  
            // 得到选择器  
            selector = Selector.open();  
            // 打开监听通道  
            listenerChannel = ServerSocketChannel.open();  
            // 绑定端口  
            listenerChannel.bind(new InetSocketAddress(PORT));  
            // 设置为非阻塞模式  
            listenerChannel.configureBlocking(false);  
            // 将选择器绑定到监听通道并监听 accept 事件  
            listenerChannel.register(selector, SelectionKey.OP_ACCEPT);  
            println("Chat Server is ready.....");  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public void start() {  
        try {  
            while (true) { //不停轮询  
                int count = selector.select(); //获取就绪 channel  
                if (count > 0) {  
                    Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();  
                    while (iterator.hasNext()) {  
                        SelectionKey key = iterator.next();  
                        // 监听到 accept  
                        if (key.isAcceptable()) {  
                            SocketChannel sc = listenerChannel.accept();  
                            //非阻塞模式  
                            sc.configureBlocking(false);  
                            //注册到选择器上并监听 read  
                            sc.register(selector, SelectionKey.OP_READ);  
                            System.out.println(sc.getRemoteAddress().toString().substring(1)+"上线了...");  
                            //将此对应的 channel 设置为 accept,接着准备接受其他客户端请求
```

```

        key.interestOps(SelectionKey.OP_ACCEPT);
    }
    //监听到 read
    if (key.isReadable()) {
        readMsg(key); //读取客户端发来的数据
    }
    //一定要把当前 key 删掉，防止重复处理
    iterator.remove();
}
} else {
    System.out.println("独自在寒风中等待...");
}
}
} catch (IOException e) {
    e.printStackTrace();
}
}

private void readMsg(SelectionKey key) {
    SocketChannel channel = null;
    try {
        // 得到关联的通道
        channel = (SocketChannel) key.channel();
        //设置 buffer 缓冲区
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        //从通道中读取数据并存储到缓冲区中
        int count = channel.read(buffer);
        //如果读取到了数据
        if (count > 0) {
            //把缓冲区数据转换为字符串
            String msg = new String(buffer.array());
            printInfo(msg);
            //将关联的 channel 设置为 read，继续准备接受数据
            key.interestOps(SelectionKey.OP_READ);
            Broadcast(channel, msg); //向所有客户端广播数据
        }
        buffer.clear();
    } catch (IOException e) {
        try {
            //当客户端关闭 channel 时，进行异常如理
            printInfo(channel.getRemoteAddress().toString().substring(1) + "下线了...");
            key.cancel(); //取消注册
            channel.close(); //关闭通道
        } catch (IOException e1) {

```

```

        e1.printStackTrace();
    }
}

public void BroadCast(SocketChannel except, String msg) throws IOException {
    System.out.println("发送广播...");
    //广播数据到所有的 SocketChannel 中
    for (SelectionKey key : selector.keys()) {
        Channel targetchannel = key.channel();
        //排除自身
        if (targetchannel instanceof SocketChannel && targetchannel != except) {
            SocketChannel dest = (SocketChannel) targetchannel;
            //把数据存储到缓冲区中
            ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
            //往通道中写数据
            dest.write(buffer);
        }
    }
}

private void printInfo(String str) { //往控制台打印消息
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    System.out.println "[" + sdf.format(new Date()) + " ] -> " + str;
}

public static void main(String[] args) {
    ChatServer server = new ChatServer();
    server.start();
}
}

```

上述代码使用 NIO 编写了一个聊天程序的服务器端，可以接受客户端发来的数据，并能把数据广播给所有客户端。

```

public class ChatClient {
    private final String HOST = "127.0.0.1"; //服务器地址
    private int PORT = 9999; //服务器端口
    private Selector selector;
    private SocketChannel socketChannel;
    private String userName;

    public ChatClient() throws IOException {
        //得到选择器
        selector = Selector.open();
        //连接远程服务器
    }
}

```

```

        socketChannel = SocketChannel.open(new InetSocketAddress("127.0.0.1", PORT));
        //设置非阻塞
        socketChannel.configureBlocking(false);
        //注册选择器并设置为 read
        socketChannel.register(selector, SelectionKey.OP_READ);
        //得到客户端 IP 地址和端口信息，作为聊天用户名使用
        userName = socketChannel.getLocalAddress().toString().substring(1);
        System.out.println("-----Client(" + userName + ") is ready-----");
    }

```

//向服务器端发送数据

```

public void sendMsg(String msg) throws Exception {
    //如果控制台输入 bye 就关闭通道，结束聊天
    if (msg.equalsIgnoreCase("bye")) {
        socketChannel.close();
        socketChannel = null;
        return;
    }
    msg = userName + "说: " + msg;
    try {
        //往通道中写数据
        socketChannel.write(ByteBuffer.wrap(msg.getBytes()));
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

//从服务器端接收数据

```

public void receiveMsg() {
    try {
        int readyChannels = selector.select();
        if (readyChannels > 0) { //有可用通道
            Set selectedKeys = selector.selectedKeys();
            Iterator keyIterator = selectedKeys.iterator();
            while (keyIterator.hasNext()) {
                SelectionKey sk = (SelectionKey) keyIterator.next();
                if (sk.isReadable()) {
                    //得到关联的通道
                    SocketChannel sc = (SocketChannel) sk.channel();
                    //得到一个缓冲区
                    ByteBuffer buff = ByteBuffer.allocate(1024);
                    //读取数据并存储到缓冲区
                    sc.read(buff);
                    //把缓冲区数据转换成字符串

```

```

        String msg = new String(buff.array());
        System.out.println(msg.trim());
    }
    keyIterator.remove(); //删除当前 SelectionKey，防止重复处理
}
} else {
    System.out.println("人呢？都去哪儿了？没人聊天啊...");
}
} catch (IOException e) {
    e.printStackTrace();
}
}
}

```

上述代码通过 NIO 编写了一个聊天程序的客户端，可以向服务器端发送数据，并能接收服务器广播的数据。

```

public class TestChat {
    public static void main(String[] args) throws Exception {
        //创建一个聊天客户端对象
        ChatClient chatClient = new ChatClient();

        new Thread() { //单独开一个线程不断的接收服务器端广播的数据
            public void run() {
                while (true) {
                    chatClient.receiveMsg();
                    try { //间隔 3 秒
                        Thread.currentThread().sleep(3000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }.start();

        Scanner scanner = new Scanner(System.in);
        //在控制台输入数据并发送到服务器端
        while (scanner.hasNextLine()) {
            String msg = scanner.nextLine();
            chatClient.sendMsg(msg);
        }
    }
}

```

上述代码运行了聊天程序的客户端，并在主线程中发送数据，在另一个线程中不断接收服务器端的广播数据，该代码运行一次就是一个聊天客户端，可以同时运行多个聊天客户端，聊天效果如下图所示：

```
Run: ChatServer x TestChat x TestChat x
[2018-11-10 22:20:57] -> Chat Server is ready.....
127.0.0.1:59560上线了...
127.0.0.1:59569上线了...
[2018-11-10 22:21:18] -> 127.0.0.1:59560说: 大家好
发送广播...
[2018-11-10 22:21:23] -> 127.0.0.1:59569说: 你好
发送广播...
[2018-11-10 22:21:50] -> 127.0.0.1:59560说: 我来自黑马程序员
发送广播...
[2018-11-10 22:21:58] -> 127.0.0.1:59569说: 我来自博学谷
发送广播...
[2018-11-10 22:22:06] -> 127.0.0.1:59560下线了...
[2018-11-10 22:22:08] -> 127.0.0.1:59569下线了...
```

```
Run: ChatServer x TestChat x TestChat x
-----Client(127.0.0.1:59560) is ready-----
大家好
127.0.0.1:59569说: 你好
我来自黑马程序员
127.0.0.1:59569说: 我来自博学谷
```

```
Run: ChatServer x TestChat x TestChat x
-----Client(127.0.0.1:59569) is ready-----
127.0.0.1:59560说: 大家好
你好
127.0.0.1:59560说: 我来自黑马程序员
我来自博学谷
```

3.4 AIO 编程

JDK 7 引入了 Asynchronous I/O, 即 AIO。在进行 I/O 编程中, 常用到两种模式: Reactor 和 Proactor。Java 的 NIO 就是 Reactor, 当有事件触发时, 服务器端得到通知, 进行相应的处理。

AIO 即 NIO2.0, 叫做异步不阻塞的 IO。AIO 引入异步通道的概念, 采用了 Proactor 模式, 简化了程序编写, 一个有效的请求才启动一个线程, 它的特点是先由操作系统完成后才通知服务端程序启动线程去处理, 一般适用于连接数较多且连接时间较长的应用。

目前 AIO 还没有广泛应用，并且也不是本课程的重点内容，这里暂不做讲解。

3.5 IO 对比总结

IO 的方式通常分为几种：同步阻塞的 BIO、同步非阻塞的 NIO、异步非阻塞的 AIO。

- BIO 方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于应用中，JDK1.4 以前的唯一选择，但程序直观简单易理解。
- NIO 方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4 开始支持。
- AIO 方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用 OS 参与并发操作，编程比较复杂，JDK7 开始支持。

举个例子再理解一下：

- 同步阻塞：你到饭馆点餐，然后在那等着，啥都干不了，饭馆没做好，你就必须等着！
- 同步非阻塞：你在饭馆点完餐，就去玩儿了。不过玩一会儿，就回饭馆问一声：好了没啊！
- 异步非阻塞：饭馆打电话说，我们知道您的位置，一会给你送过来，安心玩儿就可以了，类似于现在的外卖。

对比总结	BIO	NIO	AIO
IO 方式	同步阻塞	同步非阻塞（多路复用）	异步非阻塞
API 使用难度	简单	复杂	复杂
可靠性	差	好	好
吞吐量	低	高	高

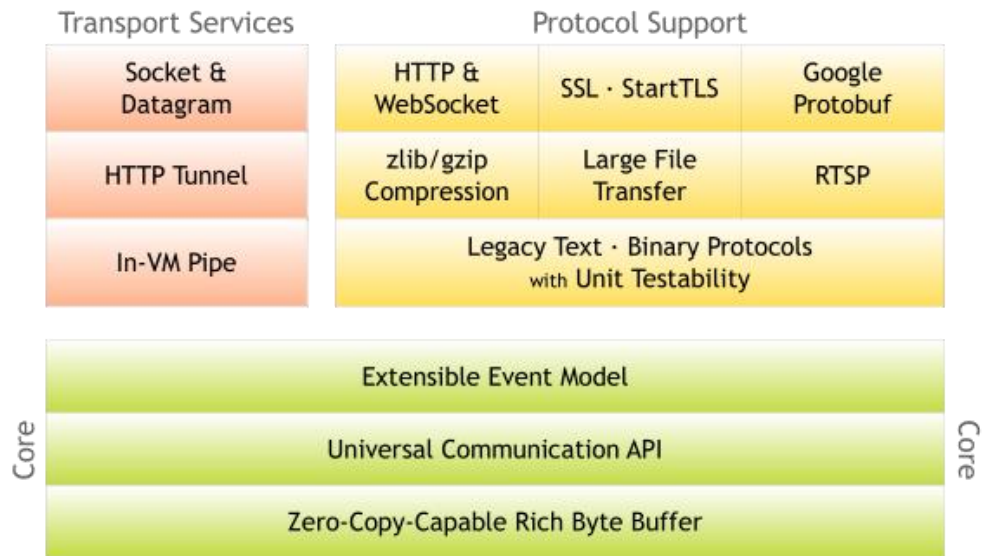
四. Netty

4.1 概述

Netty 是由 JBOSS 提供的一个 Java 开源框架。Netty 提供异步的、基于事件驱动的网络应用程序框架，用以快速开发高性能、高可靠性的网络 IO 程序。

Netty 是一个基于 NIO 的网络编程框架，使用 Netty 可以帮助你快速、简单的开发出一个网络应用，相当于简化和流程化了 NIO 的开发过程。

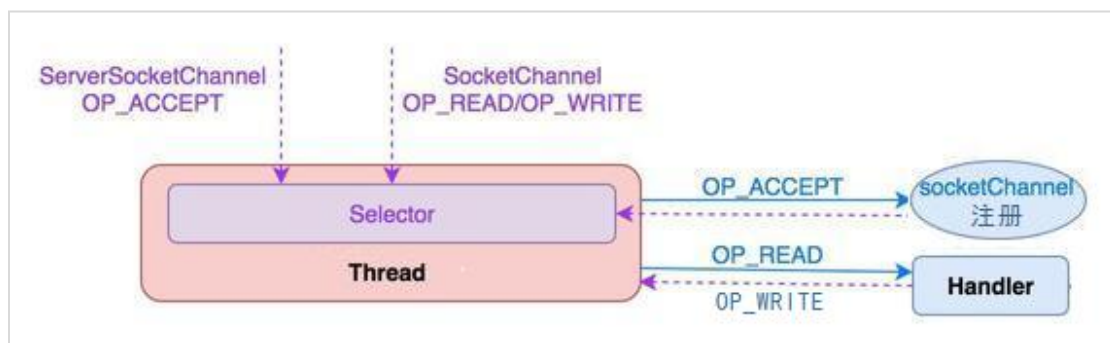
作为当前最流行的 NIO 框架，Netty 在互联网领域、大数据分布式计算领域、游戏行业、通信行业等获得了广泛的应用，知名的 Elasticsearch 、Dubbo 框架内部都采用了 Netty。



4.2 Netty 整体设计

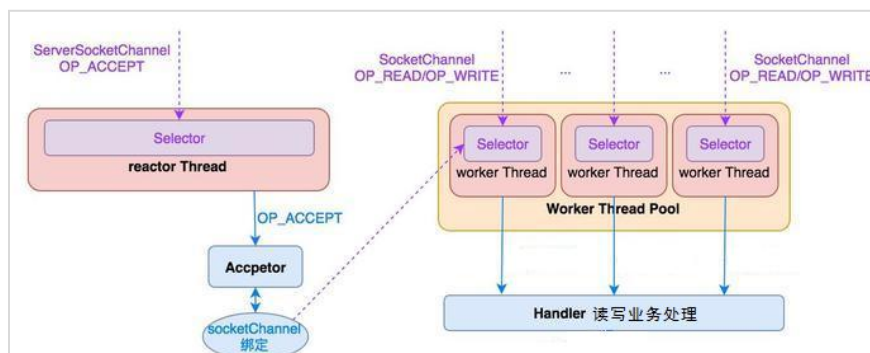
4.2.1 线程模型

1. 单线程模型



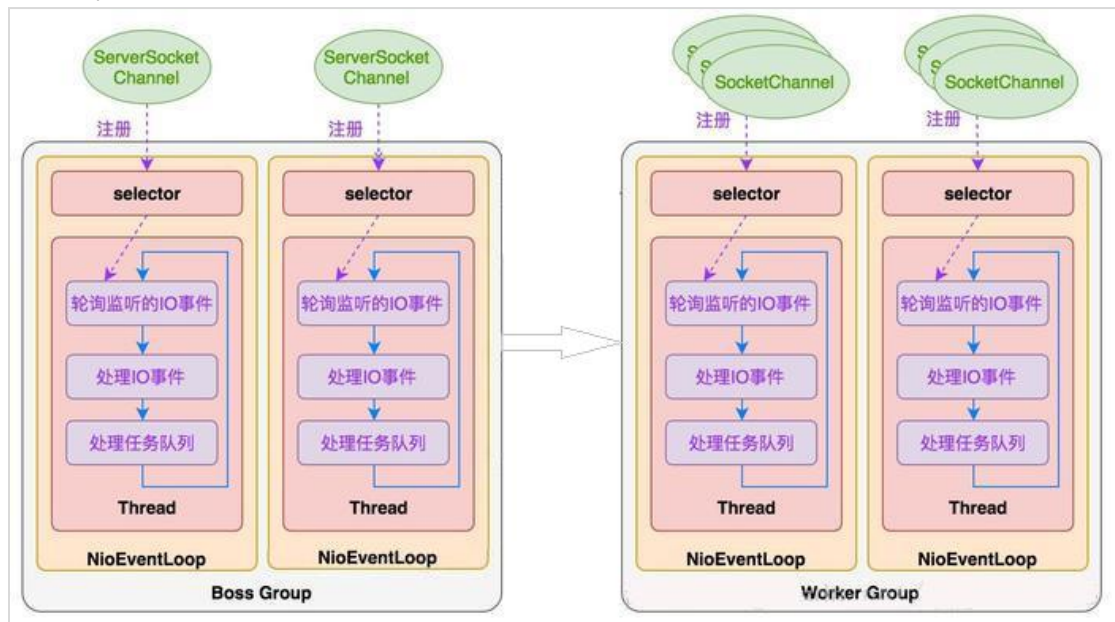
服务器端用一个线程通过多路复用搞定所有的 IO 操作（包括连接，读、写等），编码简单，清晰明了，但是如果客户端连接数量较多，将无法支撑，咱们前面的 NIO 案例就属于这种模型。

2. 线程池模型



服务器端采用一个线程专门处理客户端连接请求，采用一个线程池负责 IO 操作。在绝大多数场景下，该模型都能满足使用。

3. Netty 模型



比较类似于上面的线程池模型，Netty 抽象出两组线程池，BossGroup 专门负责接收客户端连接，WorkerGroup 专门负责网络读写操作。NioEventLoop 表示一个不断循环执行处理任务的线程，每个 NioEventLoop 都有一个 selector，用于监听绑定在其上的 socket 网络通道。NioEventLoop 内部采用串行化设计，从消息的读取->解码->处理->编码->发送，始终由 IO 线程 NioEventLoop 负责。

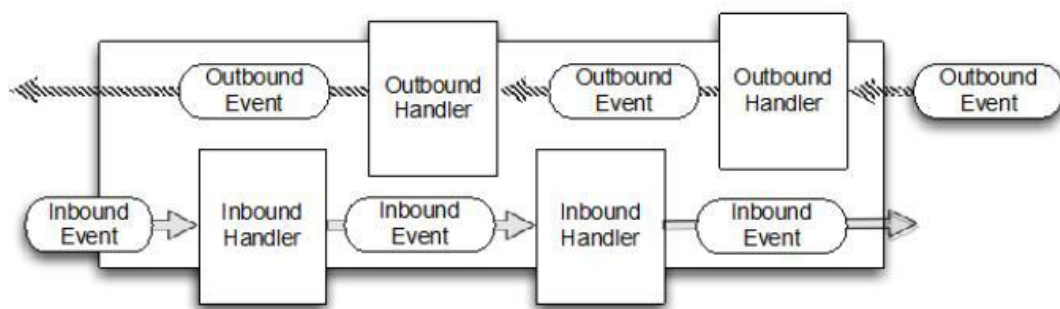
- 一个 NioEventLoopGroup 下包含多个 NioEventLoop
- 每个 NioEventLoop 中包含有一个 Selector，一个 taskQueue
- 每个 NioEventLoop 的 Selector 上可以注册监听多个 NioChannel
- 每个 NioChannel 只会绑定在唯一的 NioEventLoop 上
- 每个 NioChannel 都绑定有一个自己的 ChannelPipeline

4.2.2 异步模型

● FUTURE, CALLBACK 和 HANDLER

Netty 的异步模型是建立在 future 和 callback 的之上的。callback 大家都比较熟悉了，这里重点说说 Future，它的核心思想是：假设一个方法 fun，计算过程可能非常耗时，等待 fun 返回显然不合适。那么可以在调用 fun 的时候，立马返回一个 Future，后续可以通过 Future 去监控方法 fun 的处理过程。

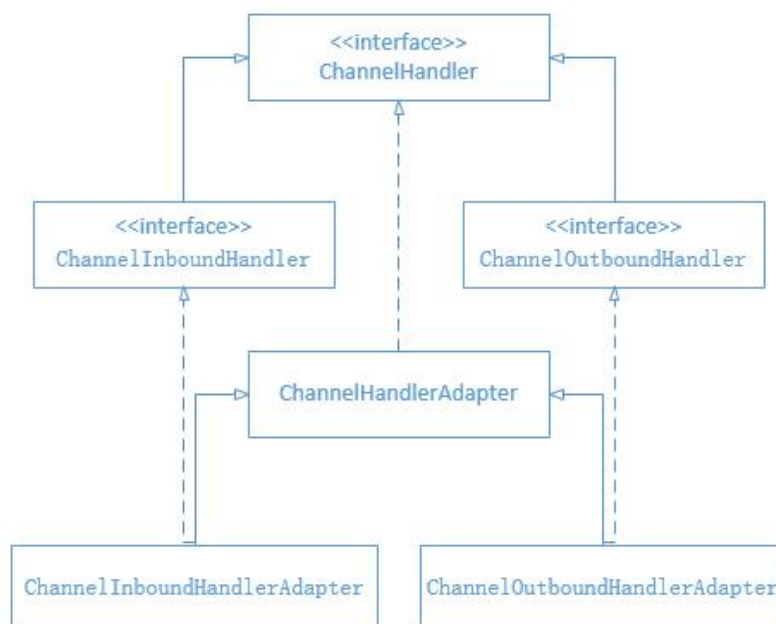
在使用 Netty 进行编程时，拦截操作和转换出入站数据只需要您提供 callback 或利用 future 即可。这使得链式操作简单、高效，并有利于编写可重用的、通用的代码。Netty 框架的目标就是让你的业务逻辑从网络基础应用编码中分离出来、解脱出来。



4.3 核心 API

● ChannelHandler 及其实现类

ChannelHandler 接口定义了许多事件处理的方法，我们可以通过重写这些方法去实现具体的业务逻辑。API 关系如下图所示：

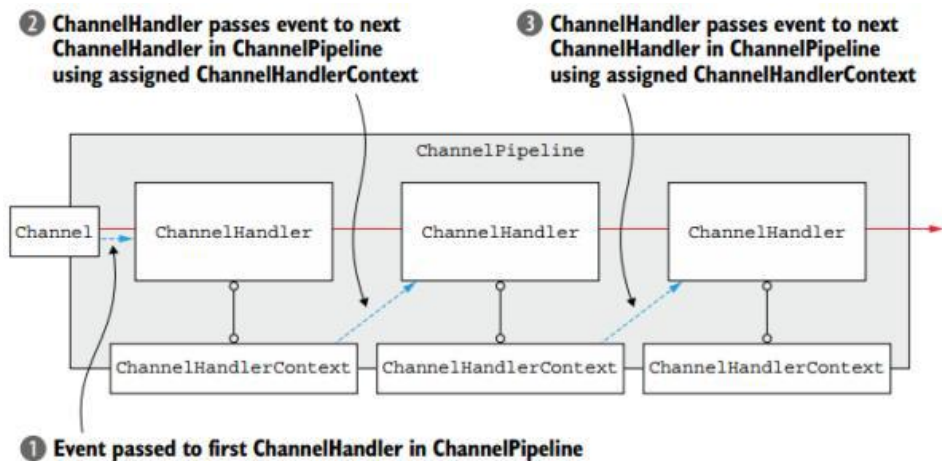


我们经常需要自定义一个 Handler 类去继承 `ChannelInboundHandlerAdapter`，然后通过重写相应方法实现业务逻辑，我们接下来看看一般都需要重写哪些方法：

- `public void channelActive(ChannelHandlerContext ctx)`，通道就绪事件
- `public void channelRead(ChannelHandlerContext ctx, Object msg)`，通道读取数据事件
- `public void channelReadComplete(ChannelHandlerContext ctx)`，数据读取完毕事件
- `public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)`，通道发生异常事件

● Pipeline 和 ChannelPipeline

ChannelPipeline 是一个 Handler 的集合，它负责处理和拦截 inbound 或者 outbound 的事件和操作，相当于一个贯穿 Netty 的链。



- ChannelPipeline addFirst(ChannelHandler... handlers), 把一个业务处理类（handler）添加到链中的第一个位置
- ChannelPipeline addLast(ChannelHandler... handlers), 把一个业务处理类（handler）添加到链中的最后一个位置

● ChannelHandlerContext

这是事件处理器上下文对象，Pipeline 链中的实际处理节点。每个处理节点 ChannelHandlerContext 中包含一个具体的事件处理器 ChannelHandler，同时 ChannelHandlerContext 中也绑定了对应的 pipeline 和 Channel 的信息，方便对 ChannelHandler 进行调用。常用方法如下所示：

- ChannelFuture close(), 关闭通道
- ChannelOutboundInvoker flush(), 刷新
- ChannelFuture writeAndFlush(Object msg), 将数据写到 ChannelPipeline 中当前 ChannelHandler 的下一个 ChannelHandler 开始处理（出站）

● ChannelOption

Netty 在创建 Channel 实例后,一般都需要设置 ChannelOption 参数。ChannelOption 是 Socket 的标准参数，而非 Netty 独创的。常用的参数配置有：

1. ChannelOption.SO_BACKLOG

对应 TCP/IP 协议 listen 函数中的 backlog 参数，用来初始化服务器可连接队列大小。服务端处理客户端连接请求是顺序处理的，所以同一时间只能处理一个客户端连接。多个客户端来的时候，服务端将不能处理的客户端连接请求放在队列中等待处理，backlog 参数指定了队列的大小。

2. ChannelOption.SO_KEEPALIVE，一直保持连接活动状态。

● ChannelFuture

表示 Channel 中异步 I/O 操作的结果，在 Netty 中所有的 I/O 操作都是异步的，I/O 的调用会直接返回，调用者并不能立刻获得结果，但是可以通过 ChannelFuture 来获取 I/O 操作的处理状态。

常用方法如下所示：

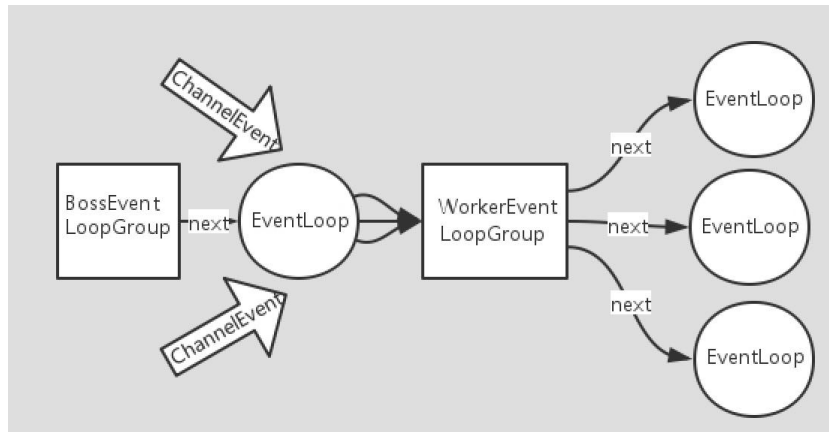
- Channel channel(), 返回当前正在进行 IO 操作的通道
- ChannelFuture sync(), 等待异步操作执行完毕

● EventLoopGroup 和其实现类 NioEventLoopGroup

EventLoopGroup 是一组 EventLoop 的抽象，Netty 为了更好的利用多核 CPU 资源，一般会有多个 EventLoop 同时工作，每个 EventLoop 维护着一个 Selector 实例。

EventLoopGroup 提供 next 接口，可以从组里面按照一定规则获取其中一个 EventLoop 来处理任务。在 Netty 服务器端编程中，我们一般都需要提供两个 EventLoopGroup，例如：BossEventLoopGroup 和 WorkerEventLoopGroup。

通常一个服务端口即一个 ServerSocketChannel 对应一个 Selector 和一个 EventLoop 线程。BossEventLoop 负责接收客户端的连接并将 SocketChannel 交给 WorkerEventLoopGroup 来进行 IO 处理，如下图所示：



BossEventLoopGroup 通常是一个单线程的 EventLoop，EventLoop 维护着一个注册了 ServerSocketChannel 的 Selector 实例，BossEventLoop 不断轮询 Selector 将连接事件分离出来，通常是 OP_ACCEPT 事件，然后将接收到的 SocketChannel 交给 WorkerEventLoopGroup，WorkerEventLoopGroup 会由 next 选择其中一个 EventLoopGroup 来将这个 SocketChannel 注册到其维护的 Selector 并对其后续的 IO 事件进行处理。

常用方法如下所示：

- public NioEventLoopGroup(), 构造方法
- public Future<?> shutdownGracefully(), 断开连接，关闭线程

● ServerBootstrap 和 Bootstrap

ServerBootstrap 是 Netty 中的服务器端启动助手，通过它可以完成服务器端的各种配置；Bootstrap 是 Netty 中的客户端启动助手，通过它可以完成客户端的各种配置。常用方法如下所示：

- public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup), 该方法用于服务器端，用来设置两个 EventLoop
- public B group(EventLoopGroup group) , 该方法用于客户端，用来设置一个 EventLoop
- public B channel(Class<? extends C> channelClass), 该方法用来设置一个服务器端的通道实现
- public <T> B option(ChannelOption<T> option, T value), 用来给 ServerChannel 添加配置
- public <T> ServerBootstrap childOption(ChannelOption<T> childOption, T value), 用来给接收到的通道添加配置
- public ServerBootstrap childHandler(ChannelHandler childHandler), 该方法用来设置业务处理类（自定义的 handler）

- `public ChannelFuture bind(int inetPort)` ，该方法用于服务器端，用来设置占用的端口号
- `public ChannelFuture connect(String inetHost, int inetPort)` ，该方法用于客户端，用来连接服务器端

- **Unpooled 类**

这是 Netty 提供的一个专门用来操作缓冲区的工具类，常用方法如下所示：

- `public static ByteBuf copiedBuffer(CharSequence string, Charset charset)`，通过给定的数据和字符编码返回一个 `ByteBuf` 对象（类似于 NIO 中的 `ByteBuffer` 对象）

4.4 入门案例

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.itcast.netty</groupId>
    <artifactId>NettyDemo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty</artifactId>
            <version>4.1.8.Final</version>
        </dependency>
    </dependencies>

    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.2</version>
                    <configuration>
                        <source>1.8</source>
                        <target>1.8</target>
                        <encoding>UTF-8</encoding>
                        <showWarnings>true</showWarnings>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

```

        </plugin>
    </plugins>
</pluginManagement>

</build>
</project>

```

上述代码在 pom 文件中引入了 netty 的坐标，采用 4.1.8 版本。

//自定义服务器端业务处理类

```

public class NettyServerHandler extends ChannelInboundHandlerAdapter {
    @Override //读取数据事件
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        System.out.println("Server: " + ctx);
        ByteBuf buf = (ByteBuf) msg;
        System.out.println("客户端发来的消息 : " + buf.toString(CharsetUtil.UTF_8));
    }
    @Override //数据读取完毕事件
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.writeAndFlush(Unpooled.copiedBuffer("就是没钱", CharsetUtil.UTF_8));
    }
    @Override //异常发生事件
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        ctx.close();
    }
}

```

上述代码定义了一个服务器端业务处理类，继承 ChannelInboundHandlerAdapter，并分别重写了三个方法。

```

public class NettyServer {
    public static void main(String[] args) throws Exception{
        //1.创建一个线程组：用来处理网络事件（接受客户端连接）
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        //2.创建一个线程组：用来处理网络事件（处理通道 IO 操作）
        EventLoopGroup workerGroup = new NioEventLoopGroup();

        //3.创建服务器端启动助手来配置参数
        ServerBootstrap b = new ServerBootstrap();
        b.group(bossGroup, workerGroup) //4.设置两个线程组 EventLoopGroup
            .channel(NioServerSocketChannel.class) //5.使用 NioServerSocketChannel 作为服务器
            端通道实现
            .option(ChannelOption.SO_BACKLOG, 128) //6.设置线程队列中等待连接的个数
            .childOption(ChannelOption.SO_KEEPALIVE, true) //7.保持活动连接状态
            .childHandler(new ChannelInitializer<SocketChannel>() { //8.创建一个通道初始化对象
                public void initChannel(SocketChannel sc) { //9.往 Pipeline 链中添加自定义的业务
                    处理 handler
                    sc.pipeline().addLast(new NettyServerHandler()); //服务器端业务处理类
                }
            })
    }
}

```

```

        System.out.println(".....Server is ready.....");
    }

    });

    //10.启动服务器端并绑定端口，等待接受客户端连接(非阻塞)
    ChannelFuture cf = b.bind(9999).sync();
    System.out.println(".....Server is Starting.....");

    //11.关闭通道，关闭线程池
    cf.channel().closeFuture().sync();
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}

```

上述代码编写了一个服务器端程序，配置了线程组，配置了自定义业务处理类，并绑定端口号进行了启动

```

//自定义客户端业务处理类
public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    @Override    //通道就绪事件
    public void channelActive(ChannelHandlerContext ctx) {
        System.out.println("Client: " + ctx);
        ctx.writeAndFlush(Unpooled.copiedBuffer("老板,还钱吧", CharsetUtil.UTF_8));
    }
    @Override    //通道读取数据事件
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        ByteBuf in = (ByteBuf) msg;
        System.out.println("服务器端发来的消息 : " + in.toString(CharsetUtil.UTF_8));
    }
    @Override    //数据读取完毕事件
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.flush();
    }
    @Override    //异常发生事件
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        ctx.close();
    }
}

```

上述代码自定义了一个客户端业务处理类，继承 ChannelInboundHandlerAdapter ，并分别重写了四个方法。

```

public class NettyClient {
    public static void main(String[] args) throws Exception {
        //1.创建一个 EventLoopGroup 线程组
        EventLoopGroup group = new NioEventLoopGroup();
        //2.创建客户端启动助手
        Bootstrap b = new Bootstrap();
    }
}

```



```

        b.group(group) //3.设置 EventLoopGroup 线程组
        .channel(NioSocketChannel.class) //4.使用 NioSocketChannel 作为客户端通道实现
        .handler(new ChannelInitializer<SocketChannel>() { //5.创建一个通道初始化对象
            @Override
            protected void initChannel(SocketChannel sc) { //6.往 Pipeline 链中添加自定义的业务
                处理 handler
                sc.pipeline().addLast(new NettyClientHandler()); //客户端业务处理类
                System.out.println(".....Client is ready.....");
            }
        });

    //7.启动客户端,等待连接上服务器端(非阻塞)
    ChannelFuture cf = b.connect("127.0.0.1", 9999).sync();

    //8.等待连接关闭(非阻塞)
    cf.channel().closeFuture().sync();
}
}

```

上述代码编写了一个客户端程序，配置了线程组，配置了自定义的业务处理类，并启动连接了服务器端。最终运行效果如下图所示：

The first screenshot shows the server's execution:

```

Run: NettyServer x NettyClient x
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
.....Server is Starting.....
.....Server is ready.....
Server: ChannelHandlerContext(NettyServerHandler#0, [id: 0x94d5200d, L:/127.0.0.1:9999 - R:/127.0.0.1:59889])
客户端发来的消息 : 老板, 还钱吧

```

The second screenshot shows the client's execution:

```

Run: NettyServer x NettyClient x
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe" ...
.....Client is ready.....
Client: ChannelHandlerContext(NettyClientHandler#0, [id: 0x29cec338, L:/127.0.0.1:59889 - R:/127.0.0.1:9999])
服务器端发来的消息 : 就是没钱

```

4.5 网络聊天案例

刚才我们通过 Netty 实现了一个入门案例，基本了解了 Netty 的 API 和运行流程，接下来我们在入门案例的基础上再实现一个多人聊天案例，具体代码如下所示：

```

//自定义一个服务器端业务处理类
public class ChatServerHandler extends SimpleChannelInboundHandler<String> {

    public static List<Channel> channels = new ArrayList<>();

    @Override //通道就绪
    public void channelActive(ChannelHandlerContext ctx) {
        Channel incoming = ctx.channel();
        channels.add(incoming);
        System.out.println("[Server:]" + incoming.remoteAddress().toString().substring(1) + "在线");
    }
}

```

```

@Override //通道未就绪
public void channelInactive(ChannelHandlerContext ctx) {
    Channel incoming = ctx.channel();
    channels.remove(incoming);
    System.out.println("[Server:]" + incoming.remoteAddress().toString().substring(1) + "掉线");
}

@Override //读取数据
protected void channelRead0(ChannelHandlerContext ctx, String s) {
    Channel incoming = ctx.channel();
    for (Channel channel : channels) {
        if (channel != incoming) { //排除当前通道
            channel.writeAndFlush("[ " + incoming.remoteAddress().toString().substring(1) + "] 说: " + s
                + "\n");
        }
    }
}

@Override //发生异常
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    Channel incoming = ctx.channel();
    System.out.println("[Server:]" + incoming.remoteAddress().toString().substring(1) + "异常");
    ctx.close();
}
}

```

上述代码通过继承 SimpleChannelInboundHandler 类自定义了一个服务器端业务处理类，并在该类中重写了四个方法，当通道就绪时，输出在线；当通道未就绪时，输出下线；当通道发来数据时，读取数据；当通道出现异常时，关闭通道。

```

//聊天程序服务器端
public class ChatServer {

    private int port; //服务器端端口号
    public ChatServer(int port) {
        this.port = port;
    }

    public void run() throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    public void initChannel(SocketChannel ch) {

```

```

        ChannelPipeline pipeline = ch.pipeline(); //得到 Pipeline 链
        //往 Pipeline 链中添加一个解码器
        pipeline.addLast("decoder", new StringDecoder());
        //往 Pipeline 链中添加一个编码器
        pipeline.addLast("encoder", new StringEncoder());
        //往 Pipeline 链中添加一个自定义的业务处理对象
        pipeline.addLast("handler", new ChatServerHandler());
    }
}

.option(ChannelOption.SO_BACKLOG, 128)
.childOption(ChannelOption.SO_KEEPALIVE, true);

System.out.println("Netty Chat Server 启动.....");
ChannelFuture f = b.bind(port).sync();
f.channel().closeFuture().sync();
} finally {
    workerGroup.shutdownGracefully();
    bossGroup.shutdownGracefully();
    System.out.println("Netty Chat Server 关闭.....");
}
}

public static void main(String[] args) throws Exception {
    new ChatServer(9999).run();
}
}

```

上述代码通过 Netty 编写了一个服务器端程序，里面要特别注意的是：我们往 Pipeline 链中添加了处理字符串的编码器和解码器，它们加入到 Pipeline 链中后会自动工作，使得我们在服务器端读写字符串数据时更加方便（不用人工处理 ByteBuf）。

```

//自定义一个客户端业务处理类
public class ChatClientHandler extends SimpleChannelInboundHandler<String> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, String s) throws Exception {
        System.out.println(s.trim());
    }
}

```

上述代码通过继承 SimpleChannelInboundHandler 自定义了一个客户端业务处理类，重写了方法 channelRead0 用来读取服务器端发过来的数据。

```

//聊天程序客户端
public class ChatClient {
    private final String host; //服务器端 IP 地址
    private final int port; //服务器端口号

    public ChatClient(String host, int port) {
        this.host = host;
    }
}

```

```

        this.port = port;
    }

    public void run(){
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap()
                .group(group)
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    public void initChannel(SocketChannel ch){
                        ChannelPipeline pipeline = ch.pipeline(); //得到 Pipeline 链
                        //往 Pipeline 链中添加一个解码器
                        pipeline.addLast("decoder", new StringDecoder());
                        //往 Pipeline 链中添加一个编码器
                        pipeline.addLast("encoder", new StringEncoder());
                        //往 Pipeline 链中添加一个自定义的业务处理对象
                        pipeline.addLast("handler", new ChatClientHandler());
                    }
                });

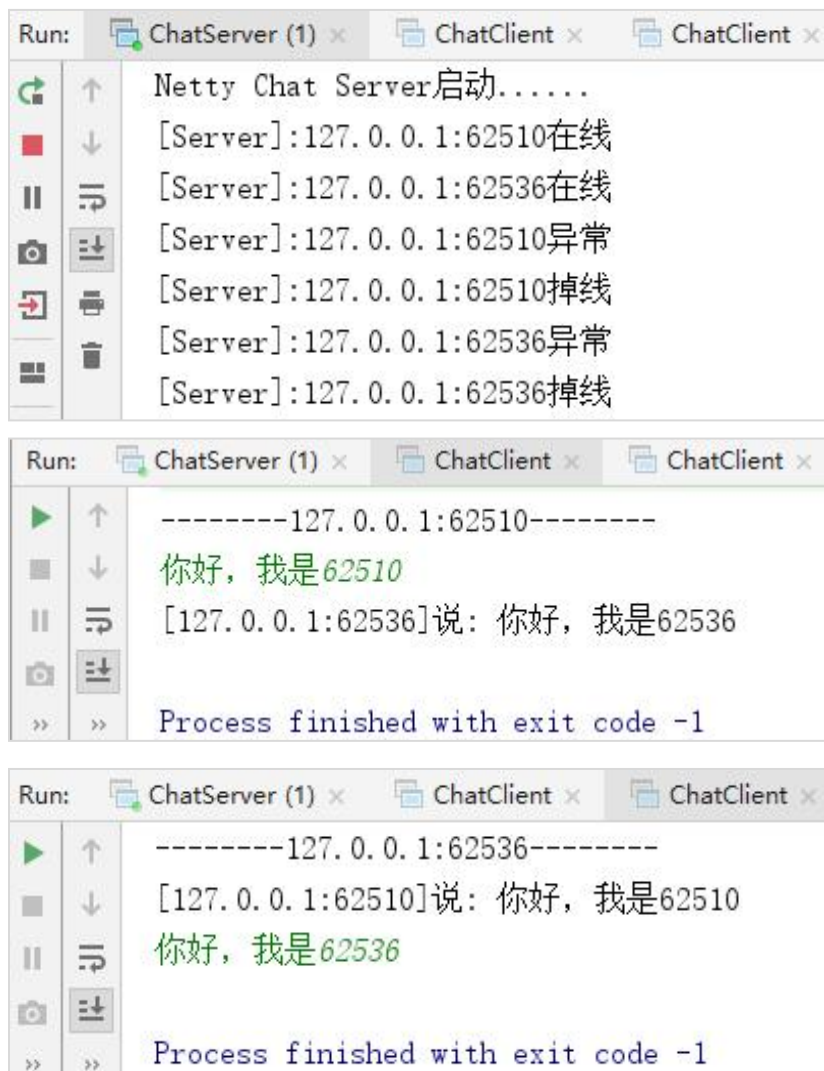
            Channel channel = bootstrap.connect(host, port).sync().channel();
            System.out.println("-----"+channel.localAddress().toString().substring(1)+"-----");
            Scanner scanner=new Scanner(System.in);
            while (scanner.hasNextLine()) {
                String msg=scanner.nextLine();
                channel.writeAndFlush(msg + "\r\n");
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            group.shutdownGracefully();
        }
    }

    public static void main(String[] args) throws Exception {
        new ChatClient("127.0.0.1", 9999).run();
    }
}

```

上述代码通过 Netty 编写了一个客户端程序，里面要特别注意的是：我们往 Pipeline 链中添加了处理字符串的编码器和解码器，他们加入到 Pipeline 链中后会自动工作，使得我们在客户端读写字符串数据时更加方便（不用人工处理 ByteBuf）。

我们可以同时运行多个聊天客户端，运行效果如下图所示：



4.6 编码和解码

4.6.1 概述

我们在编写网络应用程序的时候需要注意 **codec** (编解码器), 因为数据在网络中传输的都是二进制字节码数据, 而我们拿到的目标数据往往不是字节码数据。因此在发送数据时就需要编码, 接收数据时就需要解码。

codec 的组成部分有两个: **decoder**(解码器)和 **encoder**(编码器)。 **encoder** 负责把业务数据转换成字节码数据, **decoder** 负责把字节码数据转换成业务数据。

其实 **Java** 的序列化技术就可以作为 **codec** 去使用, 但是它的硬伤太多:

1. 无法跨语言, 这应该是 **Java** 序列化最致命的问题了。
2. 序列化后的体积太大, 是二进制编码的 5 倍多。
3. 序列化性能太低。

由于 **Java** 序列化技术硬伤太多, 因此 **Netty** 自身提供了一些 **codec**, 如下所示:
Netty 提供的解码器:

1. **StringDecoder**, 对字符串数据进行解码
2. **ObjectDecoder**, 对 **Java** 对象进行解码

3.

Netty 提供的编码器:

1. StringEncoder, 对字符串数据进行编码
2. ObjectEncoder, 对 Java 对象进行编码
3.

Netty 本身自带的 ObjectDecoder 和 ObjectEncoder 可以用来实现 POJO 对象或各种业务对象的编码和解码, 但其内部使用的仍是 Java 序列化技术, 所以我们不建议使用。因此对于 POJO 对象或各种业务对象要实现编码和解码, 我们需要更高效更强的技术。

4.6.2 Google 的 Protobuf

Protobuf 是 Google 发布的开源项目, 全称 Google Protocol Buffers, 特点如下:

- 支持跨平台、多语言 (支持目前绝大多数语言, 例如 C++、C#、Java、python 等)
- 高性能, 高可靠性
- 使用 protobuf 编译器能自动生成代码, Protobuf 是将类的定义使用 .proto 文件进行描述, 然后通过 protoc.exe 编译器根据 .proto 自动生成 .java 文件

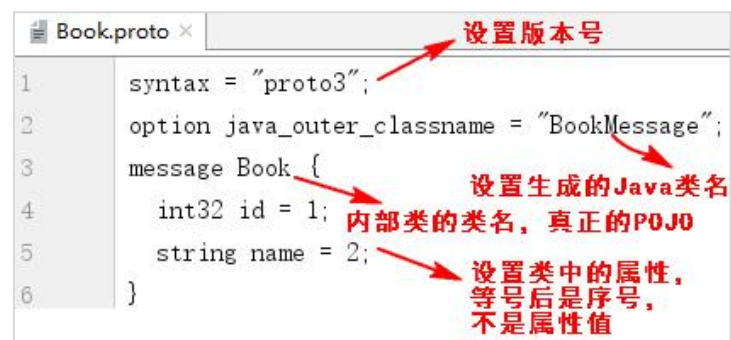
目前在使用 Netty 开发时, 经常会结合 Protobuf 作为 codec (编解码器) 去使用, 具体用法如下所示。

第 1 步:

```
<dependencies>
  <dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty</artifactId>
    <version>4.1.8.Final</version>
  </dependency>
  <dependency>
    <groupId>com.google.protobuf</groupId>
    <artifactId>protobuf-java</artifactId>
    <version>3.6.1</version>
  </dependency>
</dependencies>
```

上述代码在 pom 文件中分别引入 netty 和 protobuf 的坐标。

第 2 步: 假设我们要处理的数据是图书信息, 那就需要为此编写 proto 文件



```
Book.proto
1  syntax = "proto3";
2  option java_outer_classname = "BookMessage";
3  message Book {
4    int32 id = 1;
5    string name = 2;
6  }
```

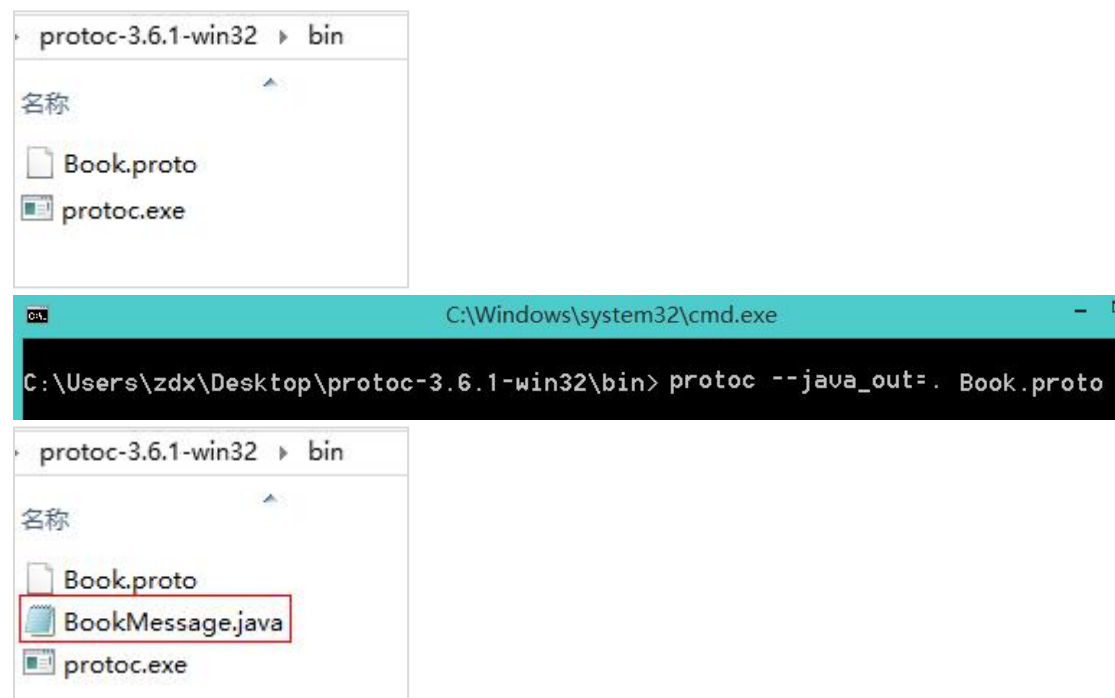
设置版本号

设置生成的Java类名

内部类的类名, 真正的POJO

设置类中的属性, 等号后是序号, 不是属性值

第 3 步：通过 protoc.exe 根据描述文件生成 Java 类，具体操作如下所示：



第四步：把生成的 BookMessage.java 拷贝到自己的项目中打开，如下图所示：



这个类我们不要编辑它，直接拿着用即可，该类内部有一个内部类，这个内部类才是真正的 POJO，一定要注意。

第 5 步：在 Netty 中去使用

```

.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel sc) {
        sc.pipeline().addLast(name: "encoder", new ProtobufEncoder());
        sc.pipeline().addLast(new NettyClientHandler()); //客户端业务处理类
        System.out.println(".....Client is ready.....");
    }
});

```

上述代码在编写客户端程序时，要向 Pipeline 链中添加 ProtobufEncoder 编码器对象。

```

public class NettyClientHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        System.out.println("Client: " + ctx);
        BookMessage.Book message = BookMessage.Book.newBuilder().setId(1).setName("Java从入门到精通").build();
        ctx.writeAndFlush(message);
    }
}

```

上述代码在往服务器端发送图书（POJO）时就可以使用生成的 BookMessage 类搞定，非常方便。

```

.childHandler(new ChannelInitializer<SocketChannel>() {
    public void initChannel(SocketChannel sc) throws Exception {
        sc.pipeline().addLast(name: "decoder", new ProtobufDecoder(BookMessage.Book.getDefaultInstance()));
        sc.pipeline().addLast(new NettyServerHandler()); //服务端业务处理类
        System.out.println(".....Server is ready.....");
    }
});

```

上述代码在编写服务器端程序时，要向 Pipeline 链中添加 ProtobufDecoder 解码器对象。

```

public class NettyServerHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        System.out.println("Server: " + ctx);
        BookMessage.Book message = (BookMessage.Book) msg;
        System.out.println("客户端发来的数据 : " + message.getName());
    }
}

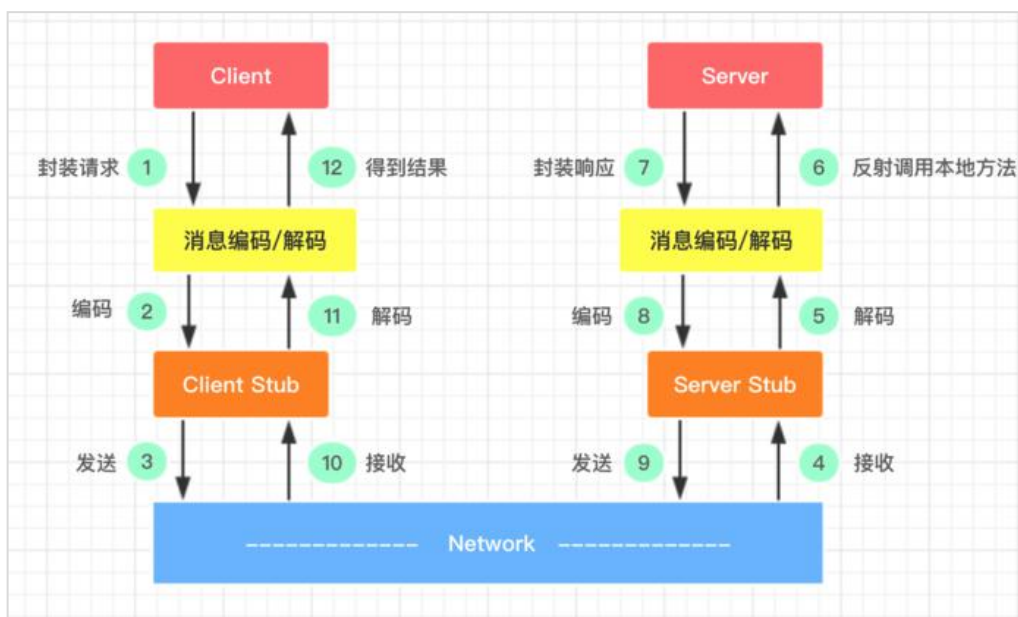
```

上述代码在服务器端接收数据时，直接就可以把数据转换成 POJO 使用，非常方便。

五. 自定义 RPC

5.1 概述

RPC（Remote Procedure Call），即远程过程调用，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络实现的技术。常见的 RPC 框架有：源自阿里的 Dubbo，Spring 旗下的 Spring Cloud，Google 出品的 grpc 等等。

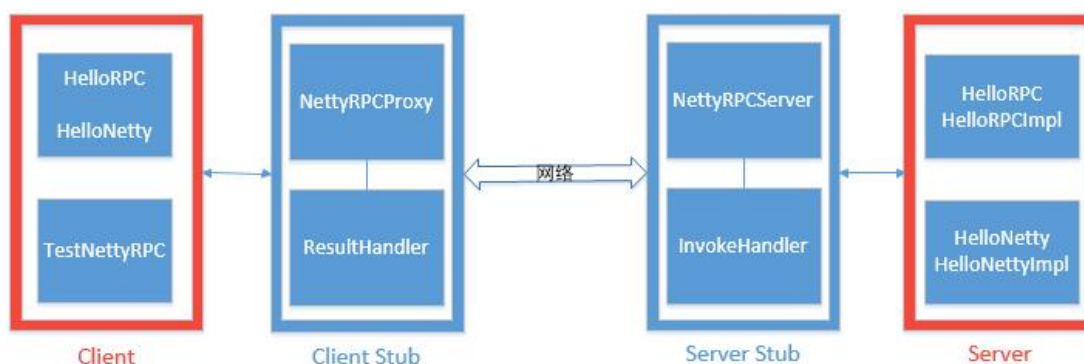


1. 服务消费方(client)以本地调用方式调用服务
2. client stub 接收到调用后负责将方法、参数等封装成能够进行网络传输的消息体
3. client stub 将消息进行编码并发送到服务端
4. server stub 收到消息后进行解码
5. server stub 根据解码结果调用本地的服务
6. 本地服务执行并将结果返回给 server stub
7. server stub 将返回导入结果进行编码并发送至消费方
8. client stub 接收到消息并进行解码
9. 服务消费方(client)得到结果

RPC 的目标就是将 2-8 这些步骤都封装起来，用户无需关心这些细节，可以像调用本地方法一样即可完成远程服务调用。接下来我们基于 Netty 自己动手搞定一个 RPC。

5.2 设计和实现

5.2.1 结构设计



- Client(服务的调用方): 两个接口 + 一个包含 main 方法的测试类
- Client Stub: 一个客户端代理类 + 一个客户端业务处理类
- Server(服务的提供方): 两个接口 + 两个实现类

- **Server Stub**: 一个网络处理服务器 + 一个服务器业务处理类

注意：服务调用方的接口必须跟服务提供方的接口保持一致（包路径可以不一致）

最终要实现的目标是：在 `TestNettyRPC` 中远程调用 `HelloRPCImpl` 或 `HelloNettyImpl` 中的方法

5.2.2 代码实现

- **Server(服务的提供方)**

```
public interface HelloNetty {
    String hello();
}

public class HelloNettyImpl implements HelloNetty {
    @Override
    public String hello() {
        return "hello,netty";
    }
}

public interface HelloRPC {
    String hello(String name);
}

public class HelloRPCImpl implements HelloRPC {
    @Override
    public String hello(String name) {
        return "hello," + name;
    }
}
```

上述代码作为服务的提供方，我们分别编写了两个接口和两个实现类，供消费方远程调用。

- **Server Stub 部分**

```
//封装类信息
public class ClassInfo implements Serializable {

    private static final long serialVersionUID = 1L;

    private String className;    //类名
    private String methodName;  //方法名
    private Class<?>[] types;    //参数类型
    private Object[] objects;    //参数列表

    //此处省略 getter 和 setter 方法
}
```

上述代码作为实体类用来封装消费方发起远程调用时传给服务方的数据。

```
//服务器端业务处理类
public class InvokeHandler extends ChannelInboundHandlerAdapter {
    //得到某接口下某个实现类的名字
    private String getImplClassName(ClassInfo classInfo) throws Exception{
```

```

//服务方接口和实现类所在的包路径
String interfacePath="cn.itcast.rpc.server";
int lastDot = classInfo.getClassName().lastIndexOf(".");
String interfaceName=classInfo.getClassName().substring(lastDot);
Class superClass=Class.forName(interfacePath+interfaceName);
Reflections reflections = new Reflections(interfacePath);
//得到某接口下的所有实现类
Set<Class> ImplClassSet=reflections.getSubTypesOf(superClass);
if(ImplClassSet.size()==0){
    System.out.println("未找到实现类");
    return null;
}else if(ImplClassSet.size(>1){
    System.out.println("找到多个实现类，未明确使用哪一个");
    return null;
}else {
    //把集合转换为数组
    Class[] classes=ImplClassSet.toArray(new Class[0]);
    return classes[0].getName(); //得到实现类的名字
}
}

@Override //读取客户端发来的数据并通过反射调用实现类的方法
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    ClassInfo classInfo = (ClassInfo) msg;
    Object clazz = Class.forName(getImplClassName(classInfo)).newInstance();
    Method method = clazz.getClass().getMethod(classInfo.getMethodName(), classInfo.getTypes());
    //通过反射调用实现类的方法
    Object result = method.invoke(clazz, classInfo.getObjects());
    ctx.writeAndFlush(result);
}
}

```

上述代码作为业务处理类，读取消费方发来的数据，并根据得到的数据进行本地调用，然后把结果返回给消费方。

```

//网络处理服务器
public class NettyRPCServer {
    private int port;
    public NettyRPCServer(int port) {
        this.port = port;
    }

    public void start() {
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {

```

```

ServerBootstrap serverBootstrap = new ServerBootstrap();
serverBootstrap.group(bossGroup, workerGroup)
    .channel(NioServerSocketChannel.class)
    .option(ChannelOption.SO_BACKLOG, 128)
    .childOption(ChannelOption.SO_KEEPALIVE, true)
    .localAddress(port).childHandler(
        new ChannelInitializer<SocketChannel>() {
            @Override
            protected void initChannel(SocketChannel ch) throws Exception {
                ChannelPipeline pipeline = ch.pipeline();
                //编码器
                pipeline.addLast("encoder", new ObjectEncoder());
                //解码器
                pipeline.addLast("decoder", new ObjectDecoder(Integer.MAX_VALUE,
                    ClassResolvers.cacheDisabled(null)));
                //服务器端业务处理类
                pipeline.addLast(new InvokeHandler());
            }
        });

ChannelFuture future = serverBootstrap.bind(port).sync();
System.out.println(".....Server is ready.....");
future.channel().closeFuture().sync();
} catch (Exception e) {
    bossGroup.shutdownGracefully();
    workerGroup.shutdownGracefully();
}
}

public static void main(String[] args) throws Exception {
    new NettyRPCServer(9999).start();
}
}

```

上述代码是用 Netty 实现的网络服务器，采用 Netty 自带的 ObjectEncoder 和 ObjectDecoder 作为编解码器（为了降低复杂度，这里并没有使用第三方的编解码器），当然实际开发时也可以采用 JSON 或 XML。

● Client Stub 部分

```

//客户端业务处理类
public class ResultHandler extends ChannelInboundHandlerAdapter {

    private Object response;

    public Object getResponse() {
        return response;
    }

    @Override //读取服务器端返回的数据(远程调用的结果)

```

```

        public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
            response = msg;
            ctx.close();
        }
    }
}

```

上述代码作为客户端的业务处理类读取远程调用返回的数据

```

//客户端代理类
public class NettyRPCProxy {
    //根据接口创建代理对象
    public static Object create(Class target) {
        return Proxy.newProxyInstance(target.getClassLoader(), new Class[]{target}, new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
                //封装 ClassInfo
                ClassInfo classInfo = new ClassInfo();
                classInfo.setClassName(target.getName());
                classInfo.setMethodName(method.getName());
                classInfo.setObjects(args);
                classInfo.setTypes(method.getParameterTypes());

                //开始用 Netty 发送数据
                EventLoopGroup group = new NioEventLoopGroup();
                ResultHandler resultHandler = new ResultHandler();
                try {
                    Bootstrap b = new Bootstrap();
                    b.group(group)
                        .channel(NioSocketChannel.class)
                        .handler(new ChannelInitializer<SocketChannel>() {
                            @Override
                            public void initChannel(SocketChannel ch) throws Exception {
                                ChannelPipeline pipeline = ch.pipeline();
                                //编码器
                                pipeline.addLast("encoder", new ObjectEncoder());
                                //解码器
                                pipeline.addLast("decoder", new ObjectDecoder(Integer.MAX_VALUE,
                                    ClassResolvers.cacheDisabled(null)));
                                //客户端业务处理类
                                pipeline.addLast("handler", resultHandler);
                            }
                        });
                    ChannelFuture future = b.connect("127.0.0.1", 9999).sync();
                    future.channel().writeAndFlush(classInfo).sync();
                    future.channel().closeFuture().sync();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }
}

```

```

        } finally {
            group.shutdownGracefully();
        }
        return resultHandler.getResponse();
    }
}
});
}
}
}

```

上述代码是用 Netty 实现的客户端代理类，采用 Netty 自带的 `ObjectEncoder` 和 `ObjectDecoder` 作为编解码器（为了降低复杂度，这里并没有使用第三方的编解码器），当然实际开发时也可以采用 JSON 或 XML。

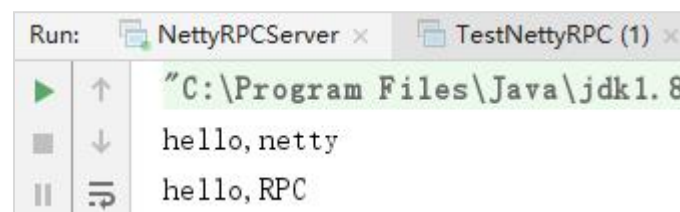
● Client (服务的调用方-消费方)

```

//服务调用方
public class TestNettyRPC {
    public static void main(String [] args){
        //第 1 次远程调用
        HelloNetty helloNetty=(HelloNetty) NettyRPCProxy.create(HelloNetty.class);
        System.out.println(helloNetty.hello());
        //第 2 次远程调用
        HelloRPC helloRPC = (HelloRPC) NettyRPCProxy.create(HelloRPC.class);
        System.out.println(helloRPC.hello("RPC"));
    }
}
}

```

消费方不需要知道底层的网络实现细节，就像调用本地方法一样成功发起了两次远程调用。运行效果如下图所示：



该案例的目的不是为了实现一个多么成熟的 RPC 框架，因此会有很多地方不够完善，我们的目的是想通过该案例让大家更熟悉 Netty 的使用。