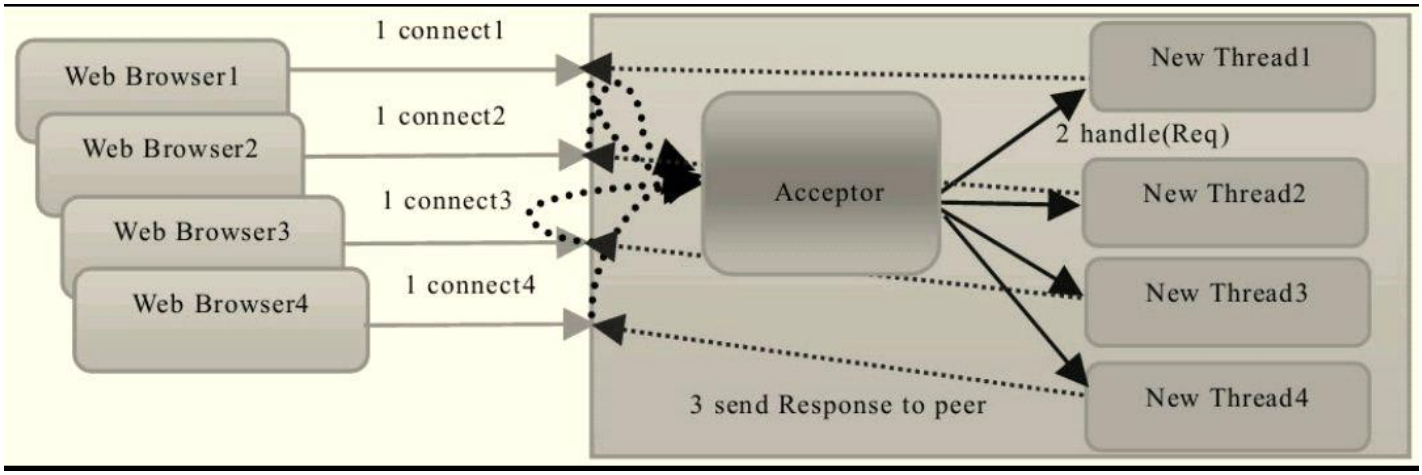
BIO网络通信

2018年9月8日 13:25

概述

网络编程的基本模型是Client/Server模型，也就是两个进程之间进行相互通信，其中服务端提供位置信息（绑定的IP地址和监听端口），客户端通过连接操作向服务端监听的地址发起连接请求，通过三次握手建立连接，如果连接建立成功，双方就可以通过网络套接字（Socket）进行通信。

在基于传统**同步阻塞模型**开发中，ServerSocket负责绑定IP地址，启动监听端口；Socket负责发起连接操作。连接成功之后，**双方通过输入和输出流进行同步阻塞式通信。**



上图采用BIO通信模型的服务端，通常由一个独立的Acceptor线程负责监听客户端的连接，它接收到客户端连接请求之后为每个客户端创建一个新的线程进行链路处理，处理完成之后，通过输出流返回应答给客户端，线程销毁。这就是典型的一请求一应答通信模型。

服务端代码示例：

```
public class Start {

    public static void main(String[] args) throws Exception {
        System.out.println("服务端启动");
        ServerSocket server=new ServerSocket(8888);

        while(true){
            Socket socket=server.accept();
            new Thread(new ClientRunner(socket)).start();
        }
    }
```

```
}
class ClientRunner implements Runnable{
	private Socket socket;

	public ClientRunner(Socket socket) {
		this.socket=socket;
	}

	@Override
	public void run() {
		try {
			InputStream in=socket.getInputStream();
			byte[] data=new byte[10];
			in.read(data);
			System.out.println("服务端收到数据:"+new String(data));

		} catch (Exception e) {
			// TODO: handle exception
		}

	}

}
```

客户端代码：

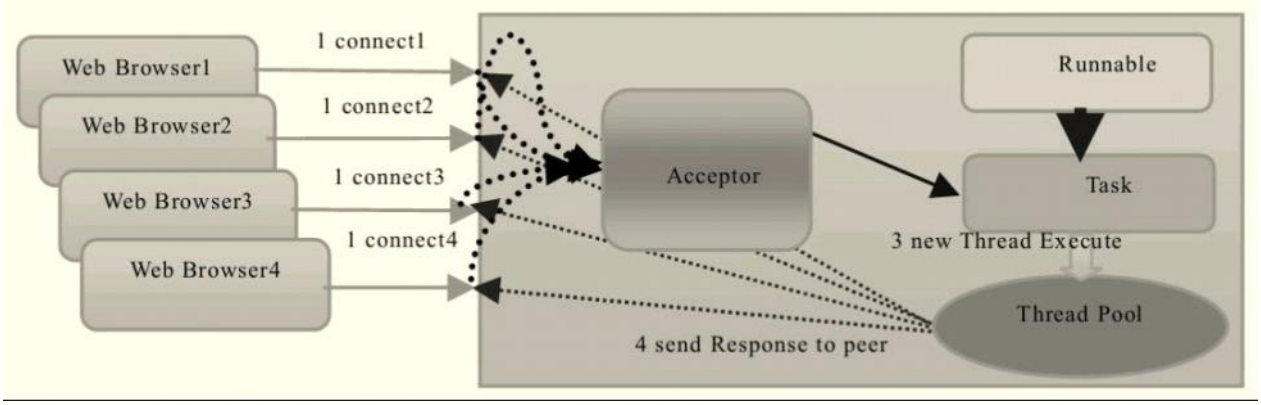
```
public class Start {

	public static void main(String[] args) throws Exception {
		System.out.println("客户端启动");
		Socket socket=new Socket("127.0.0.1",8888);
		OutputStream out=socket.getOutputStream();
		out.write("helloworld".getBytes());
		while(true);
	}
}
```

该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈1：1的正比关系，由于线程是Java虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能将急剧下降，随着并发访问量的继续增大，系统会发生线程堆栈溢出、创建新线程失败等问题，并最终导致进程宕机或者僵死，不能对外提供服务。

由此我们发现，BIO主要的问题在于每当有一个新的客户端请求接入时，服务端必须创建一个新的线程处理新接入的客户端链路，一个线程只能处理一个客户端连接。在高性能服务器应用领域，往往需要面向成千上万个客户端的并发连接，这种模型显然无法满足高性能、高并发接入的场景。

引入线程池

为了解决同步阻塞I/O面临的一个链路需要一个线程处理的问题，后来有人对它的线程模型进行了优化——后端通过一个线程池来处理多个客户端的请求接入，形成客户端个数M：线程池最大线程数N的比例关系，其中M可以远远大于N。通过线程池可以灵活地调配线程资源，设置线程的最大值，防止由于海量并发接入导致线程耗尽。



服务端代码示例（线程池分配10个线程）：

```
public class ServerStart {

    public static void main(String[] args) throws Exception {
        System.out.println("服务端启动");
        ServerSocket server=new ServerSocket(8888);
        ExecutorService pool=Executors.newFixedThreadPool(10);
        while(true){
            Socket socket=server.accept();
            pool.execute(new ClientRunner(socket));
        }

    }
}

class ClientRunner implements Runnable{

    private Socket socket;
```

```
    public ClientRunner(Socket socket) {
        this.socket=socket;
    }

    @Override
    public void run() {
        try {
            InputStream in=socket.getInputStream();
            byte[] data=new byte[10];
            in.read(data);
            System.out.println("线程编号:"+Thread.currentThread().getId()+"服务端收到数据:"+new
            String(data));


        } catch (Exception e) {
            // TODO: handle exception
        }

    }

}
```

客户端代码示例（启动20个客户端）：

```
public class ClientStart {

    public static void main(String[] args) throws Exception {
        for(int i=0;i<20;i++){
            System.out.println("客户端"+i+"启动");
            Socket socket=new Socket("127.0.0.1",8888);
            OutputStream out=socket.getOutputStream();
            out.write("helloworld".getBytes());
            Thread.sleep(3000);
        }


    }
}
```

通过上面代码的演示我们可以看到，由于线程池和消息队列都是有界的，而且避免了为每个请求都创建一个独立线程造成的线程资源耗尽问题。因此，无论客户端并发连接数多大，它都不会导致线程个数过于膨胀或者内存溢出。相比于传统的一连接一线程模型，是一种改良。

BIO通信框架的弊病

虽然上述的示例引入线程池避免了创建大量线程，但是由于底层的通信依然采用同步阻塞模型，其实并未从根本上解决问题，原因是什么呢？

```
/**
   * Reads the next byte of data from the input stream. The value byte is
   * returned as an <code>int</code> in the range <code>0</code> to
   * <code>255</code>. If no byte is available because the end of the stream
   * has been reached, the value <code>-1</code> is returned. This method
   * blocks until input data is available, the end of the stream is detected,
   * or an exception is thrown.
```

请注意加粗斜体字部分的API说明，当对Socket的输入流进行读取操作的时候，它会一直阻塞下去，直到发生如下三种事件。

1）有数据可读；

2）可用数据已经读取完毕；

3）发生空指针或者I/O异常。

这意味着当对方发送请求或者应答消息比较缓慢，或者网络传输较慢时，读取输入流一方的通信线程将被长时间阻塞，如果对方要60s才能够将数据发送完成，读取一方的I/O线程也将会被同步阻塞60s，在此期间，其他接入消息只能在消息队列中排队。

下面我们接着对输出流进行分析

服务端代码示例（向客户端写出数据）：

```
public class ServerStart {

    public static void main(String[] args) throws Exception {
        System.out.println("服务端启动");
        ServerSocket server=new ServerSocket(8888);
        ExecutorService pool=Executors.newFixedThreadPool(10);
        while(true){
            Socket socket=server.accept();
            pool.execute(new ClientRunner(socket));
        }

    }
```

```
}

class ClientRunner implements Runnable{

	private Socket socket;

	public ClientRunner(Socket socket) {
		this.socket=socket;
	}

	@Override
	public void run() {
		try {

			OutputStream out=socket.getOutputStream();
			for(int i=0;i<100000;i++){
				System.out.println("服务端写出第"+i+"次成功");
				out.write("helloworld".getBytes());
			}



		} catch (Exception e) {
			// TODO: handle exception
		}

	}

}
```

客户端代码示例（未读数据）：

```
public class ClientStart {

	public static void main(String[] args) throws Exception {

		System.out.println("客户端启动");
		Socket socket=new Socket("127.0.0.1",8888);
		while(true);

	}
}
```

当调用OutputStream的write方法写输出流的时候，如果一端写，而另外一端未进行读取也将会被阻

塞，可以想象，当消息的接收方未进行读取处理或处理缓慢的时候，通信双方会被write操作无限期阻塞。

通过以上的分析，我们了解到BIO的网络通信，读和写操作都是同步阻塞的，阻塞的时间取决于对方I/O线程的处理速度和网络I/O的传输速度。本质上来讲，我们无法保证生产环境的网络状况和对端的应用程序能足够快，如果我们的应用程序依赖对方的处理速度，它的可靠性就非常差。也许在实验室进行的性能测试结果令人满意，但是一旦上线运行，面对恶劣的网络环境和良莠不齐的第三方系统，问题就会如火山一样喷发。

所以，无论是否引入线程池技术，如果底层用的是BIO通信框架，都是无法从根本上解决通信线程阻塞问题。

BIO实现的Socket——4种产生阻塞的方法

BIO 4种产生阻塞的方法：

ServerDemo1代码（用于测试accpet,connect,read)：

```
/**
 * 这个类是作为socket的服务端
 * @author ysq
 *
 */
public class ServerDemo1 {

    public static void main(String[] args) throws Exception {
        ServerSocket ss=new ServerSocket();
        ss.bind(new InetSocketAddress(9999));
         //accept方法会产生阻塞，直到有客户端连接
         //传统的BIO会产生阻塞：
         //1.服务端accept()方法会产生阻塞
        Socket s=ss.accept();

        //接下来做读入流的测试
        //当有客户端接入时，accpet()方法不阻塞
        //但是客户端没有任何的流输入，所以产生了阻塞
        //2.即read()方法也会产生阻塞
        InputStream in=s.getInputStream();
        System.out.println("1");
        in.read();
        System.out.println("2");

    }

}
```

ClientDemo1代码（用于测试accpet,connect,read)：

```
/**
 * 这个类是客户端的Socket
```

```
 * @author ysq
 *
 */
public class ClientDemo1 {

    public static void main(String[] args) throws Exception {
        //如果服务端未启动，客户端就连接的话会报错，Connection refused
        //但是，需要留意的是，这个异常在程序启动后，并不是马上抛出的
        //而是卡顿了一秒钟才出现的
        //这个现象的原因：
        //客户端程序启动=》尝试连接服务端=》等待服务端的链接响应=》服务端没有启动=》
        //客户端收到服务端的响应，报出错误提示
        //实际上，对应客户端，socket.connect()这个方法也会产生阻塞
        Socket socket=new Socket();
        socket.connect(new InetSocketAddress("127.0.0.1", 9999));

        //while(true){}的意思是让客户端一直保持连接。
        while(true){}




    }
}
```

ServerDemo2代码（用于测试write方法）：

```
public class ServerDemo2 {

    public static void main(String[] args) throws Exception {
        ServerSocket ss =new ServerSocket();
        ss.bind(new InetSocketAddress(9998));

        Socket s=ss.accept();

        while(true){

        }
```

```
    }
}
```

ClientDemo2代码（用于测试write方法）：

```
/**
 *这个类是用来测试客户端socket write()方法是否阻塞
 */
public class ClientDemo2 {

    public static void main(String[] args) throws IOException {
        Socket s=new Socket();
        s.connect(new InetSocketAddress("127.0.0.1",9998));

        OutputStream out=s.getOutputStream();
        for(int i=0;i<1000000;i++){
        System.out.println(i);
        //结果证明，当不断向outputStream里写数据时，写到一定大小后，会产生阻塞
        //即Write方法也会产生阻塞
        out.write("a".getBytes());
        }
        out.flush();
        out.close();
    }
}
```

NIO

NIO概述

Non-Blocking I/O，是一种非阻塞通信模型。不同的语言或操作系统都有其不同的实现。

我们主要学习基于Java语言的NIO，也称为java.nio。

java.nio是jdk1.4版本引入的一套API，我们可以利用这套API实现非阻塞的网络编程模型。

为什么要学习NIO

目前无论是何种应用，都是分布式架构，因为分布式架构能够抗高并发，实现高可用，负载均衡以及存储和处理海量数据，而**分布式架构的基础是网络通信**。因此， 因此，网络编程始终是分布式软件工程师和架构师的必备高端基础技能之一。

随着当前大数据和实时计算技术的兴起，高性能 RPC 框架与网络编程技术再次成为焦点。 比如Fackebook的Thrift框架，scala的 Akka框架，实时流领域的 Storm、Spark框架，又或者开源分布式数据库中的 Mycat、VoltDB，这些框架的底层通信技术都采用了 NIO（非阻塞通信）通信技术。而 Java 领域里大名鼎鼎的 NIO 框架——Netty，则被众多的开源项目或商业软件所采用。

BIO和NIO的对比

BIO	NIO
1.阻塞通信模型，典型代表是ServerSocket 和Socket accept connect read write 会产生阻塞。所以 BIO通信模型的弊端在于：如果有大量请求，会创建大量线程，一是可能造成内存溢出，此外，线程多了之后，会造成cpu的负载过高，因为要做线程管理和上下文切换。	1.非阻塞通信模型 2.面向缓冲区（Buffer) 3.NIO传输数据的方式：把数据放到缓冲区，然后通过Channel进行传输。

虽然引入线程池，也未能解决根本问题，因为底层还是同步阻塞模型。 同步阻塞模型一是性能低，二是不可靠，取决于对端环境。 2.面向流处理，即阻塞最根本的原子在于流的read和wirte方法是阻塞方法	

BIO和NIO的适用场景

NIO的适用场景：高并发，高访问量，短请求

BIO的使用场景：访问量少，长请求（比如下载一个大文件等场景）

Buffer—ByteBuffer

Buffer：缓冲区,内存里一段连续的空间

Buffer的子类，对应了8种基本数据类型里的七种（没有boolean类型），分别是：

1.ByteBuffer

2.CharBuffer

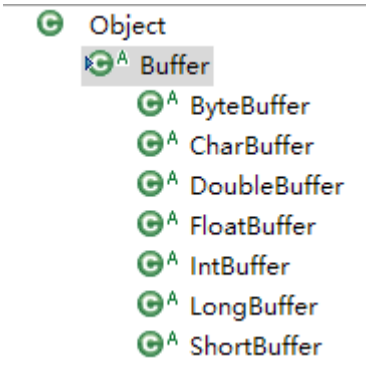
3.DoubleBuffer

4.FolatBuffer

5.IntBuffer

6.LongBuffer

7.ShortBuffer



ByteBuffer

1.创建缓冲区

static ByteBuffer allocate(int capacity)

参数：capacity 缓冲区的容量，以字节为单位

相关代码：

```
@Test
    public void testCreateBuffer(){
        //ByteBuffer是一个抽象类，我们得到的是他的实现子类对象，HeapByteBuffer
        ByteBuffer buffer=ByteBuffer.allocate(1024);



    }
```

相关代码：

```
@Test
    public void testCreateBufferByWrap(){
        //wrap方法也可以创建一个Buffer，接收的是一个字节数组，
        //并且利用wrap方法创建完buffer之后，buffer里就有了字节数组里的数据
        byte[] b={1,2,3,4};
        ByteBuffer buffer=ByteBuffer.wrap(b);
        //也可以通过指定数组下标的界限来创建Buffer并填充数据，需要注意的是：
        //利用此方法创建buffer，buffer的容量和数组的容量一致，buffer里也包含了数组的全部数据，
        //不同的是limit的位置变化了
        ByteBuffer buffer2=ByteBuffer.wrap(b,0,2);
    }
```

2.向缓冲区里写数据

put(byte b)

put(byte[] src)

putXxx()

代码：

```
public void testPutData(){

    ByteBuffer buffer=ByteBuffer.allocate(1024);
    //利用put方法，存入的是字节数据，占一个字节
    buffer.put((byte) 1);
    //putInf，存入的是整数数据，占四个字节，在开发里，我们操作都是字节数据，所以掌握put方法即可
    buffer.putInt(1);
    //put方法也可以传入字节数组
    buffer.put("123".getBytes());

    }
```

3.从缓冲区里读数据

get()

但是，当向缓冲区里写入字节时，比如1，当调用get()方法时，得到的却是0，这是什么原因呢？

代码：

```
@Test
public void testGet(){
    ByteBuffer buffer=ByteBuffer.allocate(1024);
    buffer.put((byte) 1);
    buffer.put((byte) 2);
    buffer.put((byte) 3);
    //当调用get方法是，之所以是0,是因为，在buffer缓冲区里，有一个position指针，每次put后，position位置+1
    //所以，但put完之后，直接调用get()方法，get()是根据最新指针位置来取值的，最新位置肯定是没有数据的，所以是0
    System.out.println(buffer.get());
    //也可以通过get(position)来读取指定位置的数据
    System.out.println(buffer.get(0));
}
```

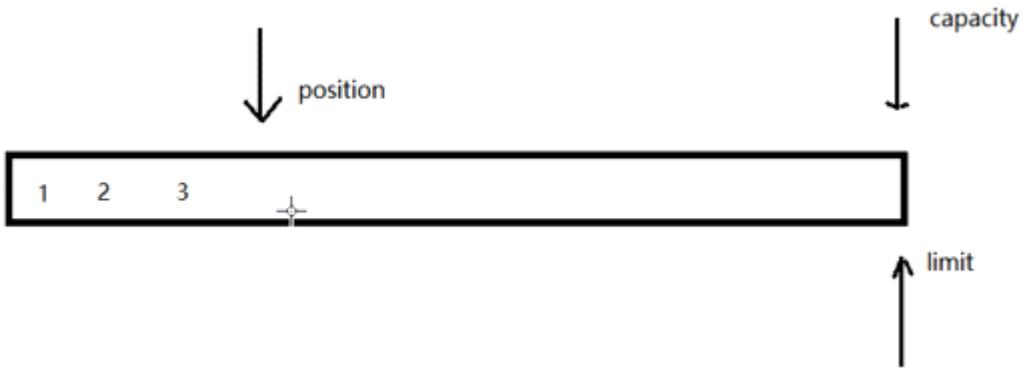
4.Buffter缓冲区的4个关键元素

①capacity,缓存区总容量

②limit的大小<=capacity的大小，创建缓冲区时，默认=capacity

③position,初始位置在缓存区的0位，当写入数据时，数据的写入位置就是position的位置写完后，position的位置+1。

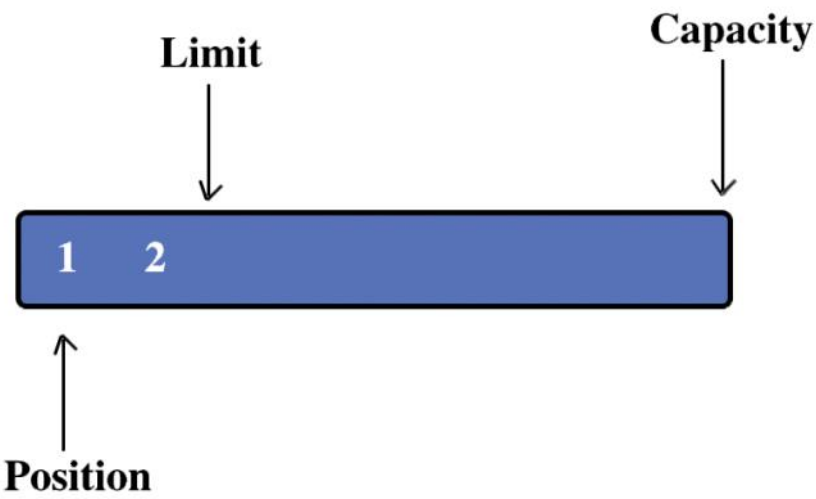
④mark，标记



测试代码：

```
@Test
    public void testPositionAndLimit(){
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        buffer.put((byte) 1);
        buffer.put((byte) 2);
```

```
        buffer.put((byte) 3);
        //当一个缓冲区创建出来之后，position的初始位置是0,每put()一次或每get()一次，都会使得positon的位置+1
        System.out.println("当前的position位置："+buffer.position());

        //在写完数据后，将limit设置为当前position位置，然后将position位置重置为0
        //这样做的目的是为了从头开始读数据，并且，用limit限制了读取下标，不会造成读出空数据的情况
        buffer.limit(buffer.position());
        //position位置重置为0
        buffer.position(0);



    }
```



5.filp()方法:

flip()反转缓冲区，作用相当于buffer.limit(buffer.position());+

```
buffer.position(0);
```

代码：

```
@Test
    public void testFlip(){
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);
```

```
        //flip()反转缓冲区，作用相当于buffer.limit(buffer.position());
        +buffer.position(0);
        buffer.flip();
    }
```

HasRemaining方法：

告知当前位置（position）和限制位（limit）之间是否还有元素

代码：

```
public void testHasRemaining(){
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);
        buffer.flip();
        while(buffer.hasRemaining()){
            System.out.println(buffer.get());
        }
    }
```

6.rewind（ ）重绕缓冲区

将position置为0

7.clear()清空缓冲区

```
@Test
    public void testClear(){
        ByteBuffer buffer=ByteBuffer.allocate(1024);
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);
        //clear的作用是清空缓冲区，但并不真正的清除数据，而是把postion置为0，
        limit置为容量上限
        //所以，当get(0)时，是可以得到原缓冲区数据的。但是我们一般都是在clear()
        方法之后，写数据，然后filp()
        //所以，并不影响缓冲区的重用。
        buffer.clear();
        System.out.println(buffer.get(0));
    }
```

Channel

Channel：通道，面向缓冲区，进行双向传输

总接口：Channel

其中重点关注他的4个子类：

- 操作TCP的

  SocketChannel

  ServerSocketChannel

- 操作UDP的

  DatagramChannel

- 操作文件的

  FileChannel

- **1.ServerSocketChannel**

- **代码：**

```
//这个方法用来测试ServerSocketChannel


    @Test
    public void testServerSocketChannel() throws IOException{
        //ServerSocketChannel是一个抽象类，不能直接new,所以调用其静态方法open()
        //创建出来的对象是ServerSocketChannelImpl的实例
        ServerSocketChannel ssc=ServerSocketChannel.open();
        ssc.socket().bind(new InetSocketAddress(9999));
        //ServerSocketChannel 创建出来之后，默认是阻塞的，如果要设置成非阻塞模
        式，需要设置：
```

```
        //ssc.configureBlocking(false); 属性为false表示非阻塞
        ssc.configureBlocking(false);
        ssc.accept();
        System.out.println("NIO服务端收到客户端请求");

    }
```

2.SocketChannel

代码：

```
@Test
    public void testSocketChannel() throws Exception{
        SocketChannel sc=SocketChannel.open();
        //SocketChannel默认也是非阻塞的，需要个更改下configureBlocking(false);
        sc.configureBlocking(false);
        sc.connect(new InetSocketAddress("127.0.0.1",9999));

    }
```

3.read和write方法

服务端代码（read)代码：

```
public class ServerDemo1 {

    public static void main(String[] args) throws Exception {

        ServerSocketChannel ssc=ServerSocketChannel.open();
        ssc.configureBlocking(false);
        ssc.socket().bind(new InetSocketAddress(9999));
```

```
        //ServerSocketChannel不做任何数据上的处理，只是提供通道，负责连
        接。SocketChannel职责和net包下的socket类似
        SocketChannel sc=null;
        while(sc==null){
            sc=ssc.accept();
        }
        ByteBuffer buffer=ByteBuffer.allocate(12);
        //这里需要注意的是：当read读的时候，需要分配好容量。
        //此外，read方法也是非阻塞的，需要用while(buffer.hasRemaining())来确保数
        据读取完整
        while(buffer.hasRemaining()){
            sc.read(buffer);

        }
        System.out.println("服务端接收数据："+new String(buffer.array()));


    }
}
```

客户端代码（write）代码：

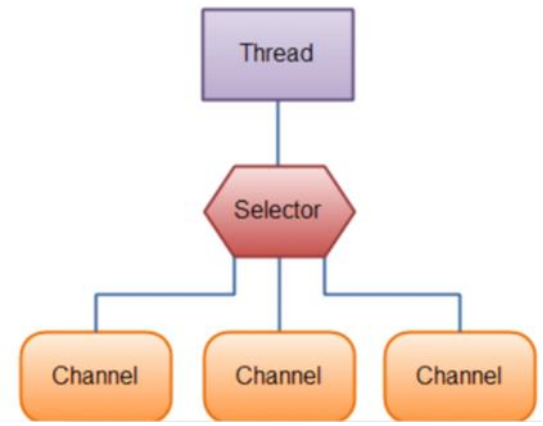
```
public class ClientDemo1 {

    public static void main(String[] args) throws IOException {
        SocketChannel sc=SocketChannel.open();
```

```
            sc.configureBlocking(false);
            sc.connect(new InetSocketAddress("127.0.0.1", 9999));
            while(!sc.isConnected()){
                sc.finishConnect();
            }
            ByteBuffer buffer=ByteBuffer.wrap("hello1604NIO".getBytes());
            while(buffer.hasRemaining()){
                //因为write方法是非阻塞的，那就是意味着write方法是否已经将buffer里
                的数据全部写完都会执行后面的代码
                //所以要用到while(buffer.hasRemaining())这种形式来确保buffer数据全
                部写出
                sc.write(buffer);
            }
            sc.close();
        }
}
```

Selector

Selector 多路复用器：
可以理解为路由器和交换机

在一个Selector上可以同时注册多个非阻塞的通道，从而只需要很少的线程数既可以管理许多通道。特别适用于开启许多通道但是每个通道中数据流量都很低的情况



NIO实现之服务端

服务端代码：

```
public class ServerSocketBySelector {

    public static void main(String[] args) {
        new Thread(new Server()).start();
    }

static class Server implements Runnable{


        @Override
        public void run() {
            try {
                selector=Selector.open();
                ServerSocketChannel ssc=ServerSocketChannel.open();
                ssc.configureBlocking(false);
                ssc.socket().bind(new InetSocketAddress(8888));
                Selector selector=Selector.open();
                ssc.register(selector, SelectionKey.OP_ACCEPT);

                while(true){
                    //select()是选择器selector查询是否有事件触发的方法，比如accpet事件是否触发，如果accpet事件触发：
                    //就意味着有客户端接入了。注意，select()是一个阻塞方法。当有事件被触发时，阻塞放开。
                    //引入selector的好处是：线程不必每时每刻都去工作、去查询客户端是否有新事件，没有事件的时候，线程就睡觉，休息
                    //有事件发生，selector会知道，线程再醒来工作。这样一来，可以避免了线程无意义的空转，节省cpu资源，同时也不影响工作
                    selector.select();
                    //能走到下面的代码，说明有事件需要处理了，我们需要根据具体是什么事件，来做相应的处理，对于服务端来说，事件分为：
                    //1.SelectionKey.OP_ACCPET 新客户端接入事件
                    //2.SelectionKey.OP_WRITE  客户端接入后，客户端给服务端传
```

```
数据事件
//3.SelectionKey.OP_READ   客户端接入后，服务端给客户端传
数据事件
Set<SelectionKey> set=selector.selectedKeys();
Iterator<SelectionKey> it=set.iterator();
while(it.hasNext()){
    SelectionKey key=it.next();

    if(key.isAcceptable()){
        System.out.println("有客户端接入");
        ServerSocketChannel ss=(ServerSocketChannel)
        key.channel();
        SocketChannel sc=ss.accept();
        sc.configureBlocking(false);
        sc.register(selector, SelectionKey.OP_READ);



    }else if(key.isReadable()){
        System.out.println("read");
        //处理如何向客户端写出数据
        //处理完后，将OP_READ事件移除
        SocketChannel s=(SocketChannel) key.channel();
        ByteBuffer buffer=ByteBuffer.allocate(3);
        while(buffer.hasRemaining()){
            s.read(buffer);
        }
        //处理完read事件后，需要把read事件从当前的
        SelectionKey键集里删除，避免重复处理
        //如果想取消SelectionKey里某一个事件，先对这个
        事件的二进制取反，在&当前键集的状态

        System.out.println("服务端接收到信息："+new
        String(buffer.array()));
        s.register(selector,
        key.interestOps()&~SelectionKey.OP_READ);

    }
```

```
                        //防止已处理完毕的SelectionKey再次被处理
                        it.remove();
                    }

                }
            } catch (Exception e) {

                e.printStackTrace();
            }

        }

    }
}003356
```

NIO实现之客户端

客户端代码：

```
public class SocketChannelBySelector {

    public static void main(String[] args) {
        new Thread(new Client()).start();
    }

    static class Client implements Runnable{


        @Override
        public void run() {

            try {
                Selector selector=Selector.open();
                SocketChannel sc=SocketChannel.open();
                sc.configureBlocking(false);
                sc.connect(new InetSocketAddress("127.0.0.1",8888));
                sc.register(selector,SelectionKey.OP_CONNECT);

                while(true){
                    selector.select();

                    Set<SelectionKey> set=selector.selectedKeys();
                    Iterator<SelectionKey> it=set.iterator();
                    while(it.hasNext()){
                        SelectionKey key=it.next();

                        if(key.isConnectable()){
                            SocketChannel s=(SocketChannel) key.channel();
                            if(!s.isConnected()){
                                s.finishConnect();
                            }
                            s.register(selector, SelectionKey.OP_WRITE);
```

```
                    }else if(key.isWritable()){
                        SocketChannel s=(SocketChannel) key.channel();
                        ByteBuffer
                        buffer=ByteBuffer.wrap("aaa".getBytes());
                        while(buffer.hasRemaining()){
                            s.write(buffer);
                        }
                        s.register(selector,
                        key.interestOps()&~SelectionKey.OP_WRITE);


                    }
                    it.remove();
                }
            }
        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

    }

}
}
```

FileChannel

代码：

```
/*
 * 这个方法用来测试FileChannel，FileChannel只能通过FileInputStream，
FileOutputStream和
 * RandomAccessFile的getChannel()方法得到。
 * FileChannel在文件操作上，性能上没什么差别。读或写都是通过缓冲区来操作。此
外还提供了一些额外方法，比如可以指定从文件的某个位置开始读或写
 * 如果FileChannel是通过FileInputStream得到，那他只能读文件，不能写文件。
 */
@Test
public void test02() throws Exception{
    FileChannel fc=new FileInputStream(new File("test02.txt")).getChannel();
    ByteBuffer buffer=ByteBuffer.allocate(1);
    fc.position(4);
    fc.read(buffer);
    System.out.println(new String(buffer.array()));
    fc.close();

}

@Test
public void test03() throws Exception{
    FileChannel fc=new FileOutputStream(new
```

```
        File("test03.txt")).getChannel();
        fc.write(ByteBuffer.wrap("test03".getBytes()));
        fc.close();
    }
    /*
     * 通过RandomAccessFile得到的FileChannel，既可以对指定文件读也可以写。并且都可以指定开始读或写的位置
     */
    @Test
    public void test04() throws Exception{
        FileChannel fc=new RandomAccessFile(new File("test03.txt"),
        "rw").getChannel();
        ByteBuffer readBuf=ByteBuffer.allocate(1);
        fc.position(2);
        fc.read(readBuf);
        System.out.println("读到的是："+new String(readBuf.array()));
        fc.write(ByteBuffer.wrap("new data".getBytes()));
        fc.close();

    }
```

扩展：Zero Copy

2017年12月5日 22:09

概述

首先来看一下维基百科对Zero Copy的定义：

（来自 <https://en.wikipedia.org/wiki/Zero-copy>）

"**Zero-copy**" describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. This is frequently used to **save CPU cycles and memory bandwidth when transmitting a file over a network.**

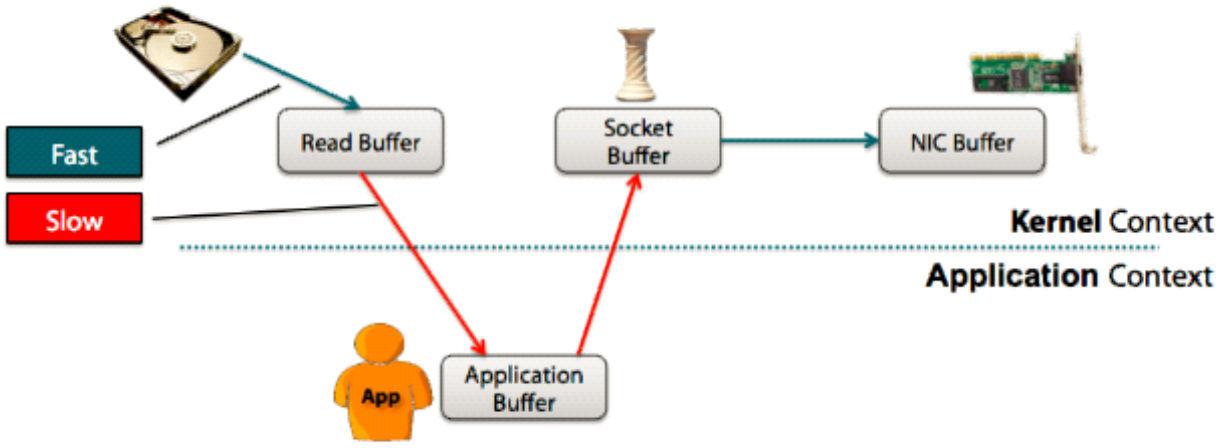
Zero-copy versions of operating system elements, such as device drivers, file systems, and network protocol stacks, greatly increase the performance of certain application programs and more efficiently utilize system resources. Also, **zero-copy operations reduce the number of time-consuming mode switches between user space and kernel space**.

原理概述

zero copy（零复制）是一种特殊形式的内存映射，它允许你将Kernel内存直接映射到设备内存空间上。其实就是设备可以通过直接内存访问（direct memory access，DMA）方式来访问Kernal Space。

我们拿Kafka举例，Kafka会对流数据做持久化存储以实现容错，比如说一个topic会对应多个Partition，每个Partition实际上最后会落地为一个文件存储在磁盘上，
这意味着当Consumer从Kafka消费数据时，会有很多data从硬盘读出之后，会原封不动的通过socket传输给用户。

这种操作看起来可能不会怎么消耗CPU，但是实际上它是低效的：kernal把数据从disk读出来，然后把它传输给user级的application，然后application再次把同样的内容再传回给处于kernal级的socket。这种场景下，application实际上只是作为一种低效的中间介质，用来把disk file的data传给socket。
如下图所示：



Non-Zero Copy

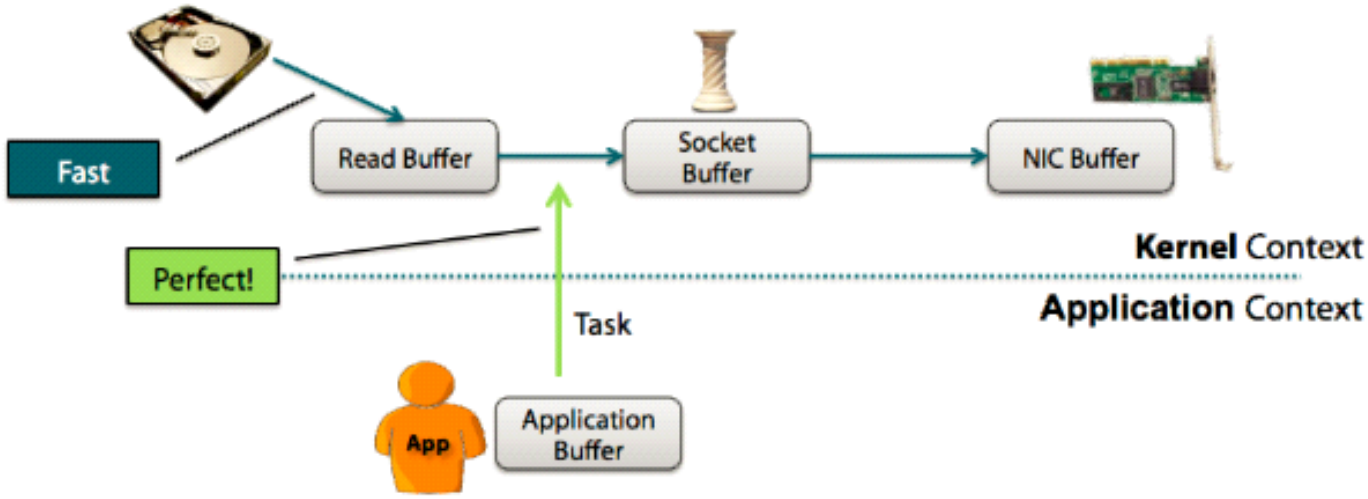
所以，当我们通过网络传输数据时，尽管看起来很简单，但是在OS的内部，这个copy操作要经历四次user mode和kernel mode之间的上下文切换，而**数据都被拷贝了四次**！

此外，data每次穿过user-kernel boundary，都会被copy，这会消耗cpu，并且占用RAM的带宽。

注：随机存取存储器(random access memory，RAM)又称作"随机存储器"，是与CPU直接交换数据的内部存储器，也叫主存(内存)。它可以随时读写，而且速度很快，通常作为操作系统或其他正在运行中的程序的临时数据存储媒介。

Zero Copy的具体实现

实际上第二次和第三次copy是毫无意义的。应用程序仅仅缓存了一下data就原封不动的把它发回给socket buffer。实际上，data应该直接在read buffer和socket buffer之间传输，如下图：

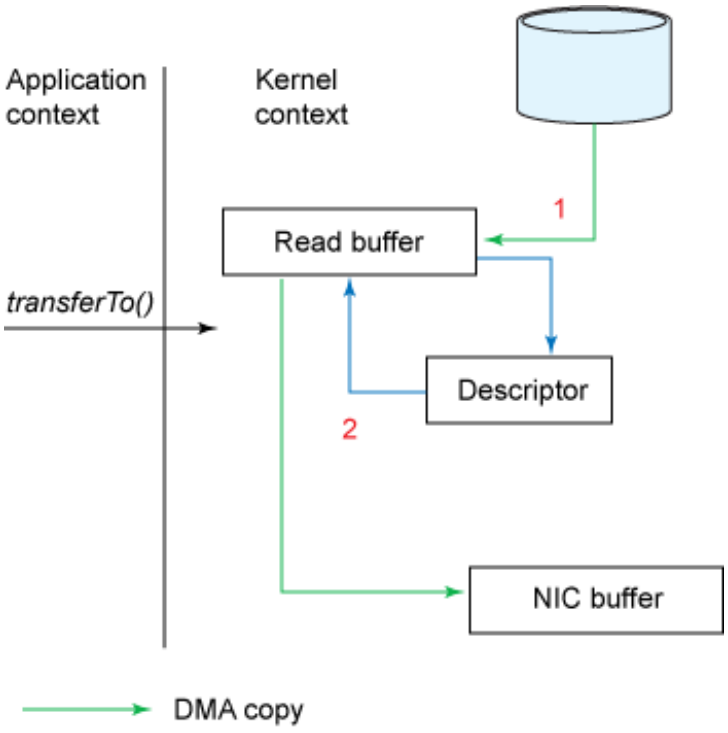


从上图中可以清楚的看到，Zero Copy的模式中，避免了数据在用户空间和内存空间之间的拷贝，从而提高了系统的整体性能。Linux中的sendfile()以及Java NIO中的FileChannel.transferTo()方法都实现了零拷贝的功能，而在Netty中也通过在FileRegion中包装了NIO的FileChannel.transferTo()方法实现了零拷贝。

这是一个很明显的进步：我们把context switch的次数从4次减少到了2次，同时也把data copy的次数从4次降低到了3次(而且其中只有一次占用了CPU，另外两次由DMA完成)。但是，要做到真正的zero copy，这还差一些。

如果网卡支持 gather operation，我们可以通过kernel进一步减少数据的拷贝操作。在2.4及以上版本的linux内核中，开发者修改了socket buffer descriptor来适应这一需求。这个方法不仅减少了context switch，还消除了和CPU有关的数据拷贝。

如下图：



最新的Zero Copy的机制是追加了一些descriptor的信息，包括data的位置和长度。然后DMA 直接把data从kernel buffer传输到protocol engine，这样就消除了唯一的一次需要占用CPU的拷贝操作。

为什么要使用kernel buffer做中介

使用kernel buffer做中介(而不是直接把data传到user buffer中)看起来比较低效(多了一次copy)。然而实际上kernel buffer是用来提高性能的。在进行读操作的时候，kernel buffer起到了预读cache的作用。当写请求的data size比kernel buffer的size小的时候，这能够显著的提升性能。在进行写操作时，kernel buffer的存在可以使得写请求完全异步。

但悲剧的是，当读请求的data size远大于kernel buffer size的时候，这个方法本身变成了性能的瓶颈。

Java NIO的总结

总结

最后总结一下到底NIO给我们带来了些什么：

1）事件驱动模型

2）避免多线程

3）单线程处理多任务

4）非阻塞I/O，I/O读写不再阻塞

5）基于block(缓冲区buffer)的传输，通常比基于流的传输更高效，缓冲区是可以被复用的

6）IO多路复用大大提高了Java网络应用的可伸缩性和实用性

相对于它的老前辈 BIO（阻塞通信）来说，NIO 模型非常复杂，以至于苦学了很久以后也很少有人能够精通它，难以编写出一个没有缺陷、高效且适应各种意外情况的稳定的 NIO 通信模块。所以我们一般使用已有的NIO框架，比如java的Netty框架

之所以会出现这样的问题，是因为 NIO 编程不是单纯的一个技术点，而是涵盖了一系列相关的技术、专业知识、编程经验和编程技巧的复杂工程。

今天内容的重点

2018年9月11日 16:50

1.理解好BIO网络通信的弊病

2.理解同步阻塞的概念及引起的问题

3.NIO网络通信模型的设计思想(基于事件驱动+单线程或少量线程处理多任务+非阻塞)

4.掌握FileChannel的使用