NIO概述

2018年8月17日 星期五　上午 9:08

1. NIO概述

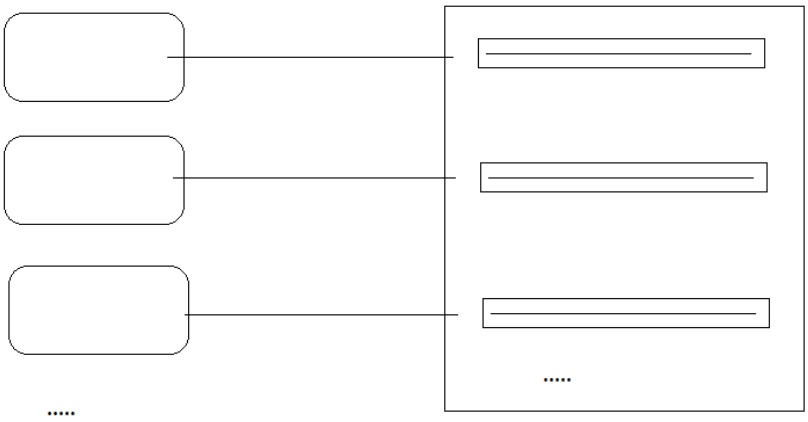
BIO - JDK1.0 - 同步阻塞式IO

在执行ACCEPT CONNECT READ WRITE 时都会产生阻塞

在平常开发当中并不是问题 甚至因为这样的模型直观而简单 用的很多

但是在高并发的场景下 这样的阻塞式IO可能会造成问题

在服务器开发中 需要在服务器端通过少量线程处理多个客户端请求 这就要求 在少量的线程应该可以灵活的切换处理不同客户端 但传统的BIO阻塞式的工作方式 一旦阻塞了线程 线程就被挂起 无法继续执行 无法实现这样的功能



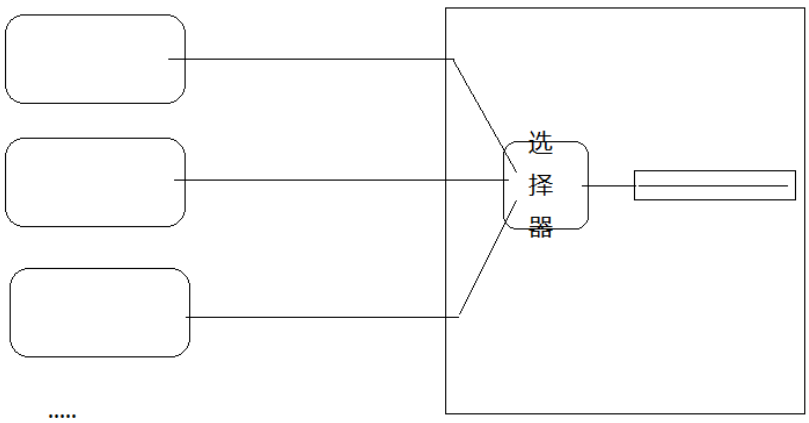
NIO - JDK4.0 - 同步非阻塞式IO

NewIO

NonBlockingIO

和传统的BIO比起来最主要的特点是 在执行ACCEPT CONNECT READ WRITE 操作时是非阻塞的

非常便于实现 在服务器开发中 用少量的线程来处理多个客户端请求 由于以上四种操作都是非阻塞的 可以随时让线程切换所处理的客户端 从而可以实现高并发服务器的开发



2. 特点

BIO:同步阻塞式IO 面向流 操作字节或字符 单向传输数据

NIO:同步非阻塞式IO 面向通道 操作缓冲区 双向传输数据

Buffer详解

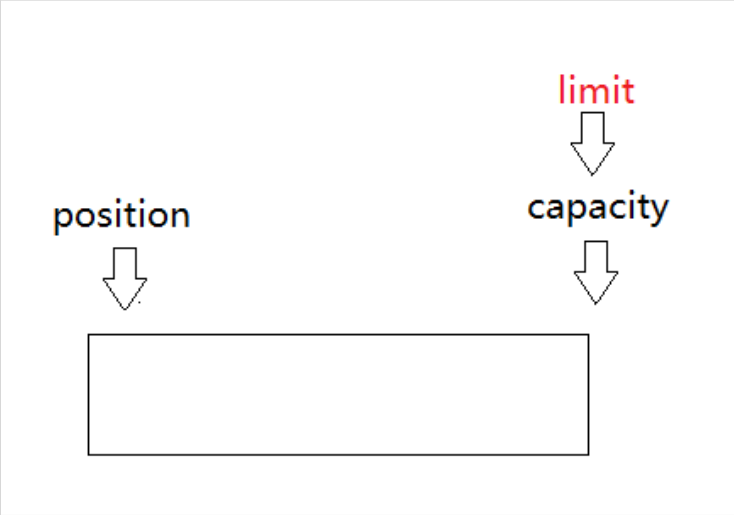
2018年8月17日 星期五　　上午 10:15

1. Buffer概述

缓冲区,本质上就是一段连续的内存空间,用来临时存放大量指定类型的数据

```
java.nio.Buffer
    |
    ByteBuffer, CharBuffer, DoubleBuffer,
     FloatBuffer, IntBuffer, LongBuffer, ShortBuffer
```

2. Buffer中的重要概念



a. capacity - 容量,在创建Buffer时就需要指定好,后续不可修改

int	**capacity**() 返回此缓冲区的容量。

b. position - 当前位置,初始值为0,指定Buffer进行读写操作时操作位置,每当操作过后position自动+1指向下一个位置

int	**position**() 返回此缓冲区的位置。
Buffer	**position**(int newPosition) 设置此缓冲区的位置。

c. limit - 限制位,初始值等于capacity,position永远小于等于limit

int	**limit**() 返回此缓冲区的限制。

Buffer	**limit**(int newLimit) 设置此缓冲区的限制。

3. 创建Buffer

创建出来的Buffer默认 capacity等于容量 position为0 limit等于capacity

a. 没有构造方法 不能直接new

b. 可以直接使用如下方法创建指定大小的空的缓冲区 其中的capacity参数就是缓冲区的容量的大小 缓冲区容量大小 只能再创建时指定 之后无法进行修改

static ByteBuffer	**allocate**(int capacity) 分配一个新的字节缓冲区。

c. 将一个已经存在的byte数组 包装为一个ByteBuffer 其中的数据会被保留 ByteBuffer的容量 等于 byte数据中获取到的数据的大小

static ByteBuffer	**wrap**(byte[] array) 将 byte 数组包装到缓冲区中。
static ByteBuffer	**wrap**(byte[] array, int offset, int length) 将 byte 数组包装到缓冲区中。

4. 写入数据到buffer

position指向写入数据数据的位置,每当写入一个数据,position自动+1指向下一个位置,position不可大于limit,如果一直写入,达到limit大小,再写入会抛出异常

通过putXxx()可以向缓冲区中写入不同类型的数据 要注意 无论用的是哪个方法 put的是什么类型的数据 存到缓冲区里都是字节数据

abstract ByteBuffer	**put**(byte b) 相对 *put* 方法 *（可选操作）*。
ByteBuffer	**put**(byte[] src) 相对批量 *put* 方法 *（可选操作）*。
ByteBuffer	**put**(byte[] src, int offset, int length) 相对批量 *put* 方法 *（可选操作）*。
ByteBuffer	**put**(ByteBuffer src) 相对批量 *put* 方法 *（可选操作）*。
abstract ByteBuffer	**put**(int index, byte b) 绝对 *put* 方法 *（可选操作）*。
abstract ByteBuffer	**putChar**(char value) 用来写入 char 值的相对 *put* 方法 *（可选操作）*。

abstract ByteBuffer	**putChar**(int index, char value) 用于写入 char 值的绝对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putDouble**(double value) 用于写入 double 值的相对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putDouble**(int index, double value) 用于写入 double 值的绝对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putFloat**(float value) 用于写入 float 值的相对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putFloat**(int index, float value) 用于写入 float 值的绝对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putInt**(int value) 用于写入 int 值的相对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putInt**(int index, int value) 用于写入 int 值的绝对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putLong**(int index, long value) 用于写入 long 值的绝对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putLong**(long value) 用于写入 long 值（*可先操作*）的相对 *put* 方法。
abstract ByteBuffer	**putShort**(int index, short value) 用于写入 short 值的绝对 *put* 方法（*可选操作*）。
abstract ByteBuffer	**putShort**(short value) 用于写入 short 值的相对 *put* 方法（*可选操作*）。

5. 从buffer中获取数据

position指向读取数据的位置,每当读到一个数据,position自动+1指向下一个位置,position不可大于limit,如果一直读取,达到limit大小,再读取会抛出异常

通过getXxx()可以向缓冲区中写入不同类型的数据

abstract byte	**get**() 相对 *get* 方法。
ByteBuffer	**get**(byte[] dst) 相对批量 *get* 方法。
ByteBuffer	**get**(byte[] dst, int offset, int length) 相对批量 *get* 方法。
abstract byte	**get**(int index) 绝对 *get* 方法。
abstract char	**getChar**() 用于读取 char 值的相对 *get* 方法。

abstract char	**getChar**(int index) 用于读取 char 值的绝对 *get* 方法。
abstract double	**getDouble**() 用于读取 double 值的相对 *get* 方法。
abstract double	**getDouble**(int index) 用于读取 double 值的绝对 *get* 方法。
abstract float	**getFloat**() 用于读取 float 值的相对 *get* 方法。
abstract float	**getFloat**(int index) 用于读取 float 值的绝对 *get* 方法。
abstract int	**getInt**() 用于读取 int 值的相对 *get* 方法。
abstract int	**getInt**(int index) 用于读取 int 值的绝对 *get* 方法。
abstract long	**getLong**() 用于读取 long 值的相对 *get* 方法。
abstract long	**getLong**(int index) 用于读取 long 值的绝对 *get* 方法。
abstract short	**getShort**() 用于读取 short 值的相对 *get* 方法。
abstract short	**getShort**(int index) 用于读取 short 值的绝对 *get* 方法。

6. 反转缓冲区

在写入缓冲区完成,想要从中读取数据之前需要先进行反转缓冲区操作,本质上就是将 limit设置为当前position的值,再将position设置为0的过程

Buffer	**flip**() 反转此缓冲区。

等价于:

```
buf.limit(buf.position());
buf.position(0);
```

7. 判断边界

这个方法可以返回 limit - position的值,通常用来获取读写时是距离边界的距离

int	**remaining**() 返回当前位置与限制之间的元素数。

这个方法可以返回 limit-position>0 的值,通常用来判断度写时是否到达了边界

boolean	**hasRemaining**()

	告知在当前位置和限制之间是否有元素。

8. 重绕缓冲区

这个方法将将position置为0 ,可以重新进行读写操作

Buffer	**rewind**() 重绕此缓冲区。

9. 设置/重置标记

可以在读写缓冲区的过程中 通过mark()方法设置一个临时的指针mark指向当前position的值 之后 在任何时候 可以调用reset()方法 使position重新指向mark指定的值 从而 恢复到上一次mark位进行读写操作

Buffer	**mark**() 在此缓冲区的位置设置标记。
Buffer	**reset**() 将此缓冲区的位置重置为以前标记的位置。

10. 清空缓冲区

此方法用来清空缓冲区,但是它并不会真的取清除缓冲区中的数据,而只是修改响应buffer的参数.抛弃mark 将limit设置为capacity 将position设置为0,效果上等价于清空数据.

Buffer	**clear**() 清除此缓冲区。

Channel详解

2018年8月17日 星期五　下午 2:09

1. 通道概述

NIO中的基本概念,类似于BIO中的流,不同的是,操作的是缓冲区,且可以双向传输数据

```
AbstractInterruptibleChannel implements Channel
    |
    |-FileChannel
    |
    |-SelectableChannel
        |
        |-AbstractSelectableChannel
            |
            |-SocketChannel
            |-ServerSocketChannel
            |-DatagramChannel
```

2. ServerSocketChannel

代表tcp通信中的服务器端

构造方法被保护起来 无法直接使用

构造方法摘要	
protected	**ServerSocketChannel**(SelectorProvider provider) 初始化此类的一个新实例。

可以通过如下静态方法 得到ServerSocketChannel对象

static ServerSocketChannel	**open**() 打开服务器套接字通道。

获取到底层ServerSocketChannel底层对应的真正的Socket套接字对象

abstract ServerSocket	**socket**() 获取与此通道关联的服务器套接字。

绑定监听端口

abstract ServerSocket	bind(InetSocketAddress isa) 将通道绑定到某一地址端口

**jdk7才开始有这个方法 在这以前 需要先获取socket通过socket来绑定端口

配置通道的阻塞模式,默认是阻塞模式,通过传入false可以改为非阻塞模式

SelectableChannel	**configureBlocking**(boolean block) 调整此通道的阻塞模式。

等待客户端连接,在阻塞模式下会一直阻塞直到客户端连接,返回一个代表链接的SockentChannel对象.在非阻塞模式下,此方法不会阻塞,直接执行下去,如果没有得到一个新的连接,此方法返回null

abstract SocketChannel	**accept**() 接受到此通道套接字的连接。

**在非阻塞模式下,ACCEPT操作没有阻塞,无论是否收到一个连接,都直接执行下去,此时即使ACCEPT方法执行成功,也无法确认连接完成.此时应该自己通过代码来控制实现连接,或者,通过选择器来实线选择操作.

```
SocketChannel sc = null;
while(sc == null){
    sc = ssc.accept();
}
```

关闭通道

void	**close**() 关闭此通道。

3. SocketChannel

代表tcp通信中的客户端

构造方法被保护起来 无法直接使用

构造方法摘要	
protected	**SocketChannel**(SelectorProvider provider) 初始化此类的一个新实例。

可以通过如下静态方法 得到SocketChannel对象

static SocketChannel	**open**() 打开套接字通道。

配置通道的阻塞模式,默认是阻塞模式,通过传入false可以改为非阻塞模式

SelectableChannel	**configureBlocking**(boolean block) 调整此通道的阻塞模式。

命令客户端连接指定服务器地址端口 如果通道处于阻塞模式 则此方法会一直阻塞 直到 连接成功 而如果通道处于非阻塞模式 此方法 将仅仅尝试着去连接 如果连接成功则返回true 如果连接一时间没有结束 也不阻塞程序 此方法返回false 程序继续执行 此时需要在后续调用finishConnection方法来完成连接

abstract boolean	**connect**(SocketAddress remote) 连接此通道的套接字。

如果通道是非阻塞模式的 之前的connect又没有一次性的完成连接 则后续调用此方法 完成连接 此方法也是非阻塞的 调用结束并不意味着连接完成 所以如果此方法返回false 应该继续重复调用 直到返回true 才表明连接完成

abstract boolean	**finishConnect**() 完成套接字通道的连接过程。

**在非阻塞模式下,CONNECT操作没有阻塞,无论是否完成一个连接,都直接执行下去,此时即使CONNECT方法执行成功,也无法确认连接完成.此时应该自己通过代码来控制实现连接,或者,通过选择器来实线选择操作.

```
boolean isConn = sc.connect(new InetSocketAddress("127.0.0.1", 44444));
if(!isConn){
    while(!sc.finishConnect()){
    }
}
```

从通道中读取数据到指定的缓冲区内,在阻塞模式下,如果没有读取到数据或者读取到的数据不够填满缓冲区,此方法将会阻塞,直到读取到的数据填满了缓冲区阻塞才会被放开.在非阻塞模式下,此方法只是尝试读取数据到缓冲区,无论是否读到或者是否读满缓冲区,都不会产生阻塞

abstract int	**read**(ByteBuffer dst) 将字节序列从此通道中读入给定的缓冲区。

**在非阻塞模式下,read方法执行过后,并不能保证真的读到了数据,或读全了数据,此时只能自己写代码来控制读取的过程.

```
ByteBuffer buf = ByteBuffer.allocate(5);
while(buf.hasRemaining()){
    sc.read(buf);
}
```

将缓冲区中的数据写入通道,在阻塞模式下,这个方法将会尝试将通道中的数据写入缓冲区,如果无法完全写入此方法将会产生阻塞,直到所有数据都写入成功,阻塞才会被放开.在非阻塞模式下,这个方法会尝试写入数据,无论是否

写入成功,或无论是否将整个缓冲区都写入完成,此方法都不会产生阻塞.

abstract int	**write**(ByteBuffer src) 将字节序列从给定的缓冲区中写入此通道。

**在非阻塞模式下,write方法执行过后,并不能保证写出了所有的数据,此时只能自己写代码来控制写出的过程.

```
while(buf.hasRemaining()){
    sc.write(buf);
}
```

关闭通道

void	**close**() 关闭此通道。

案例 - NIO实现TCP通信

2018年8月17日 星期五　　下午 3:20

服务端:

```
package cn.tedu.nio.channel;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;

public class ServerSocketChannelDemo01 {
	public static void main(String[] args) throws Exception {
		//1.创建ServerSockentChannel对象
		ServerSocketChannel ssc = ServerSocketChannel.open();
		//2.绑定指定端口
		ssc.bind(new InetSocketAddress(44444));
		//3.设置非阻塞模式
		ssc.configureBlocking(false);
		//4.接收客户端连接
		SocketChannel sc = null;
		while(sc == null){
			sc = ssc.accept();
		}
		sc.configureBlocking(false);
		//5.读取数据
		ByteBuffer buf = ByteBuffer.allocate(5);
		while(buf.hasRemaining()){
			sc.read(buf);
		}
		//6.获取数据打印
		byte[] arr = buf.array();
		String str = new String(arr);
		System.out.println(str);

		//5.关闭通道
		sc.close();
		ssc.close();
	}
}
```

客户端:

```
package cn.tedu.nio.channel;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class SocketChannelDemo01 {
	public static void main(String[] args) throws Exception {
		//1.创建客户端SocketChannel
```

```
        SocketChannel sc = SocketChannel.open();
        //2.配置启用非阻塞模式
        sc.configureBlocking(false);
        //3.连接服务器
        boolean isConn = sc.connect(new InetSocketAddress("127.0.0.1", 44444));
        if(!isConn){
            while(!sc.finishConnect()){
            }
        }

        //4.发送数据到服务器
        ByteBuffer buf = ByteBuffer.wrap("abcde".getBytes());
        while(buf.hasRemaining()){
            sc.write(buf);
        }

        //5.关闭通道
        sc.close();
    }
}
```

Selector详解

2018年8月17日 星期五　　下午 3:22

1. Selector概述

Selector选择器是NIO中实现少量线程服务多个客户端连接的最关键的组件.一方面允许多个客户端连接注册到选择器中关注对应的事件.另一方面提供了"选择"操作 来在之前注册的操作中选择已经就绪的操作 交给线程来执行. 一头连接了多个客户端连接 另一头连接了少量的线程,进行协调 从而实现了 少量线程 来处理多个客户端连接的功能.

从功能上来说:

功能1:注册功能 允许客户端连接 注册到选择器上 关注对应事件

功能2:选择功能 可以在线程中 选择已经就绪的注册事件 来进行处理

2. 继承结构

java.nio.channels.Selector

3. 相关方法

构造方法被保护起来,无法直接new.

构造方法摘要	
protected	**Selector**() 初始化此类的一个新实例。

可以调用此静态方法 创建一个选择器

static Selector	**open**() 打开一个选择器。

注册通道到选择器

**注册通道到选择器的方法 不在Selector上 而在Channel身上

SelectionKey	**register**(Selector sel, int ops) 向给定的选择器注册此通道，返回一个选择键。

**sel : 要将当前通道注册到哪个选择器上

**ops : 将当前通道注册到指定选择器上时 声明的要关注的事件

可供选择的操作类型:

java.nio.channels.SelectionKey

字段摘要	
static int	**OP_ACCEPT** 用于套接字接受操作的操作集位。
static int	**OP_CONNECT** 用于套接字连接操作的操作集位。
static int	**OP_READ**

	用于读取操作的操作集位。
static int	**OP_WRITE** 用于写入操作的操作集位。

**此方法返回了一个SelectionKey对象 此对象是一个代表本次注册事件的对象 从这个对象上 可以得到 对应的是哪个通道 注册在哪个选择器上 关注的是哪个事件

选择已经就绪的事件

选择器会在已经注册的事件中检查 有多少个事件已经就绪 返回就绪的事件的数量

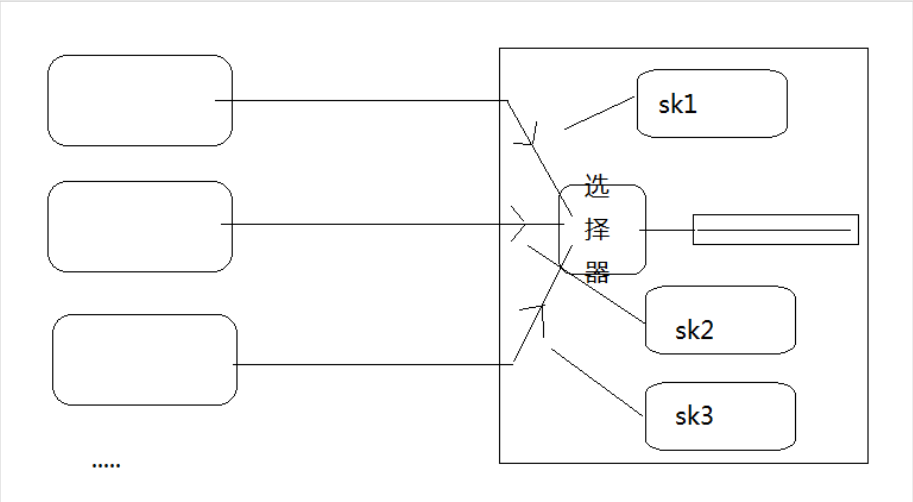
这个方法调用时如果发现没有任何一个事件就绪 则进入阻塞状态 直到有事件就绪阻塞才会被放开

abstract int	**select**() 选择一组键，其相应的通道已为 I/O 操作准备就绪。

在调用select()之后 确定选择到了已经就绪的键 此时可以调用此方法 来获取这些就绪的键

abstract Set<SelectionKey>	**selectedKeys**() 返回此选择器的已选择键集。

**此方法返回了一个SelectionKey组成的集合 其中包含的就是已经就绪 可以处理的SelectionKey们的集合 只要遍历这个集合 拿到SelectionKey 就可以知道 是哪个通道 哪类事件就绪了



案例 - 实现少量线程 处理多个客户端请求

2018年8月17日 星期五　　下午 4:16

1. 目标

   利用Selector+channel+Buffer实现 少量线程处理多个客户端请求

2. 客户端

```
package cn.tedu.nio.selector;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class SocketChannelDemo01 {
	public static void main(String[] args) throws Exception {
		//0.创建选择器
		Selector selc = Selector.open();
		//1.创建SocketChannel
		SocketChannel sc = SocketChannel.open();
		//2.设定非阻塞模式
		sc.configureBlocking(false);
		//3.连接服务端
		sc.connect(new InetSocketAddress("127.0.0.1", 44444));
		sc.register(selc, SelectionKey.OP_CONNECT);

		//4.通过选择器实行选择操作
		while(true){
			selc.select();//选择器尝试选择就绪的键 选不到就阻塞 选择到就返回就绪的键的数量

			//5.得到并遍历就绪的键们
			Set<SelectionKey> keys = selc.selectedKeys();
			Iterator<SelectionKey> it = keys.iterator();
			while(it.hasNext()){
```

```
                    //6.得到每一个就绪的键
                    SelectionKey key = it.next();
                    //7.获取就绪的键 对应的 操作 和 通道
                    if(key.isAcceptable()){

                    }else if(key.isConnectable()){
                        //--是通道的Connect操作
                        //--获取通道
                        SocketChannel scx = (SocketChannel)
                        //--完成连接
                        if(!scx.isConnected()){
                            while(!scx.finishConnect()){};
                        }
                        //--将通道再次注册到selc中 关注WRITE操作
                        scx.register(selc, SelectionKey.OP_WRITE);
                    }else if(key.isReadable()){

                    }else if(key.isWritable()){
                        //--发现是Write操作就绪
                        //--获取通道
                        SocketChannel scx = (SocketChannel)
                        //--写出数据
                        ByteBuffer buf = ByteBuffer.wrap("hello nio~
                        hello java~".getBytes());
                        while(buf.hasRemaining()){
                            scx.write(buf);
                        }
                        //--取消掉当前通道 在选择器中的注册 放置重复
                        key.cancel();
                    }else{
                        throw new RuntimeException("未知的键,见了
                    }
                    //8.移除就绪键
                    it.remove();
                }
            }
        }
}
```

3. 服务端

```
package cn.tedu.nio.selector;
```

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class ServerSocketDemo01 {
    public static void main(String[] args) throws Exception {
        //0.创建选择器
        Selector selc = Selector.open();
        //1.创建代表服务器的ServerSocketChannel对象
        ServerSocketChannel ssc = ServerSocketChannel.open();
        //2.设置为非阻塞模式
        ssc.configureBlocking(false);
        //3.设置监听的端口
        ssc.bind(new InetSocketAddress(44444));
        //4.将ssc注册到选择器中关注ACCEPT操作
        ssc.register(selc, SelectionKey.OP_ACCEPT);

        //5.通过选择器选择就绪的键
        while(true){
            selc.select();//尝试到注册的键集中来寻找就绪的键 如果一
            个就绪的键都找不到 就进入阻塞 直到找到就绪的键 返回就

            //6.获取就绪的键的集合
            Set<SelectionKey> keys = selc.selectedKeys();

            //7.遍历处理就绪的键 代表的操作
            Iterator<SelectionKey> it = keys.iterator();
            while(it.hasNext()){
                //--获取到就绪的键 根据键代表的操作的不同 来进行
                SelectionKey key = it.next();

                if(key.isAcceptable()){
                    //--发现了Accept操作
                    //--获取通道
                    ServerSocketChannel sscx =
```

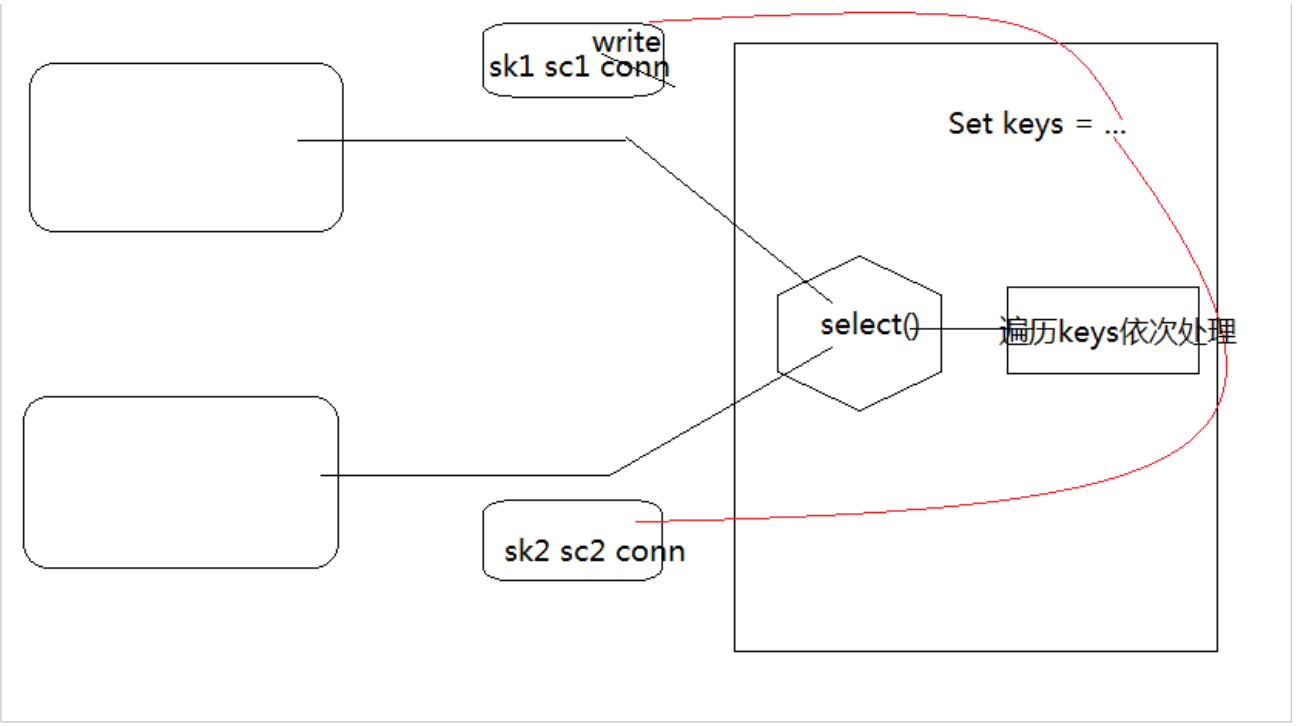
```
                    (ServerSocketChannel) key.channel();
                    //--完成Accept操作
                    SocketChannel sc = sscx.accept();
                    //--在sc上注册读数据的操作
                    sc.configureBlocking(false);
                    sc.register(selc, SelectionKey.OP_READ);
                }else if(key.isConnectable()){

                }else if(key.isWritable()){

                }else if(key.isReadable()){
                    //--发现了Read操作
                    //--获取就绪的通道
                    SocketChannel scx = (SocketChannel)
                    //--完成读取数据的操作
                    ByteBuffer buf = ByteBuffer.allocate(10);
                    while(buf.hasRemaining()){
                        scx.read(buf);
                    }
                    String msg = new String(buf.array());
                    System.out.println("[收到来自客户端的消
                }else{
                    throw new RuntimeException("未知的键,见了
                }

                //8.移除处理完的键
                it.remove();
            }
        }
    }
}
```

4. 原理图


write
sk1 sc1 conn
Set keys = ...
select()
遍历keys依次处理
sk2 sc2 conn

粘包问题

2018年8月18日 星期六　　上午 9:17

1. 粘包问题概述

socket编程其实是基于传输层的协议在开发网络程序的过程.传输层本身就没有针对会话进行管理的功能.所以在进行socket编程时会遇到和会话相关的各种问题.最典型的就是粘包问题.

所谓的粘包问题 就是当 通过socket发送多段数据时 底层的tcp协议 会自动根据需要 将数据 拆分或合并 组成数据包后发送给接受者 ,接受者收到数据后 无法直接通过tcp协议本身判断数据的边界,这个问题就称之为粘包问题.

粘包问题本质上是因为tcp协议是传输层的协议 本身没有对会话控制提供相应的能力 我们基于socket开发网络程序时 相当于在自己实现 会话层 表示层 和应用层的功能 所以 需要自己来相办法解决粘包问题.

2. 粘包问题的解决方案

a. 只发送固定长度的数据

通信的双发约定每次发送数据的长度,每次只发送固定长度的数据,接收数据方 每次都按照固定长度获取数据

缺点:

不够灵活,只适合每次传输的数据都有固定长度的场景

b. 约定分隔符

通信双方约定一个特殊的分隔符用来表示数据的边界,接收方收到数据时,不停读取,以分隔符为标志,区分数据的边界

缺点:

如果数据本身就包含分隔符字符,则需要对数据进行预处理将数据本身包含的分隔符进行转义,相对来说比较麻烦

c. 使用协议

所谓的协议 其实就是通信双方遵循的通信方式的约定

协议分为 公有协议 和 私有协议

所谓的公有协议 通常是由 国际标准化组织定义 全世界的计算机都遵循的协议 例如 HTTP协议 FTP协议 POP3协议 SMTP协议...

所谓的私有协议 通常是由 公司 组织 个人自定义的协议 只在小范围内使用 解决小范围内通信时特定的需求

案例-通过自定义协议完成任意长度数据通信

2018年8月18日 星期六　　上午 10:49

1. 服务端:

```
package cn.tedu.nio.nb;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class ServerSocketChannelDemo01 {
	public static void main(String[] args) throws Exception {
		System.err.println("服务端启动...");
		//0.创建选择器
		Selector selc = Selector.open();
		//1.创建ssc
		ServerSocketChannel ssc = ServerSocketChannel.open();
		ssc.configureBlocking(false);
		//2.绑定端口
		ssc.bind(new InetSocketAddress(44444));
		//3.注册ACCEPT
		ssc.register(selc, SelectionKey.OP_ACCEPT);

		//4.选择器执行选择操作 对就绪的键进行处理
		while(true){
			//--进行选择操作
			selc.select();

			//--获取选择到的键
			Set<SelectionKey> keys = selc.selectedKeys();
```

```
//--遍历选择到的键 进行处理
Iterator<SelectionKey> it = keys.iterator();
while(it.hasNext()){
    SelectionKey key = it.next();

    if(key.isAcceptable()){
        //--发现了Accept操作
        //--获取通道
        ServerSocketChannel sscx = (ServerSocketChannel)
        key.channel();
        //--完成Accept操作
        SocketChannel sc = sscx.accept();
        sc.configureBlocking(false);
        //--注册sc 到选择器 关注 read操作
        sc.register(selc,SelectionKey.OP_READ);
    }else if(key.isConnectable()){

    }else if(key.isReadable()){
        //--发现了Read操作
        //--获取通道
        SocketChannel scx = (SocketChannel) key.channel();
        //--完成Read
        //----根据协议 数据的结构 为 [长度\r\n内容]
        //----先读取长度
        ByteBuffer tmp = ByteBuffer.allocate(1);
        String line = "";
        while(!line.endsWith("\r\n")){
            scx.read(tmp);
            line += new String(tmp.array());
            tmp.clear();
        }
        int len = Integer.parseInt(line.substring(0, line.length()-2));
        //----读取后续len个字节 就是当前这段数据
        ByteBuffer buf = ByteBuffer.allocate(len);
        while(buf.hasRemaining()){
            scx.read(buf);
```

```
                }
                String msg = new String(buf.array());
                System.out.println("收到了来自客户端的消息:[" +msg+ "]");

                //--将当前通道注册 关注Write事件
                scx.register(selc, SelectionKey.OP_WRITE);

            }else if(key.isWritable()){
                //--发现了Write操作
                //--获取通道
                SocketChannel scx = (SocketChannel) key.channel();
                //--完成Write
                String str = "来自服务器的相应消息:[你好,客户端]";
                String data = str.getBytes().length+"\r\n"+str;
                ByteBuffer buf = ByteBuffer.wrap(data.getBytes());
                while(buf.hasRemaining()){
                    scx.write(buf);
                }
                //--取消Write注册 防止重复写出
                key.cancel();
            }else{
                throw new RuntimeException("未知的键~!");
            }

            //--移除处理完成的键
            it.remove();
        }
    }

  }
}
```

2. 客户端:

```
package cn.tedu.nio.nb;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
```

```
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
import java.util.Set;

public class SocketChannelDemo01 {
    public static void main(String[] args) throws Exception {
        System.err.println("客户端启动...");
        //0.创建选择器
        Selector selc = Selector.open();
        //1.创建sc
        SocketChannel sc = SocketChannel.open();
        sc.configureBlocking(false);
        //2.注册Connect操作
        sc.connect(new InetSocketAddress("127.0.0.1", 44444));
        sc.register(selc, SelectionKey.OP_CONNECT);

        //3.选择器执行选择操作 对就绪的键进行处理
        while(true){
            //--进行选择操作
            selc.select();

            //--获取到就绪的键
            Set<SelectionKey> keys = selc.selectedKeys();

            //--遍历就绪的键 依次处理
            Iterator<SelectionKey> it = keys.iterator();
            while(it.hasNext()){
                SelectionKey key = it.next();

                if(key.isAcceptable()){

    }else if(key.isConnectable()){
        //--发现Connect操作
        //--获取通道
        SocketChannel scx = (SocketChannel) key.channel();
        //--完成Connect操作
```

```
        if(!scx.isConnected()){
            while(!scx.finishConnect()){}
        }
        //--注册 sc 到选择器 关注Write操作
        scx.register(selc, SelectionKey.OP_WRITE);
    }else if(key.isReadable()){
        //--发现了Read操作
        //--获取通道
        SocketChannel scx = (SocketChannel) key.channel();
        //--完成Read
        //----根据协议 数据的结构 为 [长度\r\n内容]
        //----先读取长度
        ByteBuffer tmp = ByteBuffer.allocate(1);
        String line = "";
        while(!line.endsWith("\r\n")){
            scx.read(tmp);
            line += new String(tmp.array());
            tmp.clear();
        }
        int len = Integer.parseInt(line.substring(0, line.length()-2));
        //----读取后续len个字节 就是当前这段数据
        ByteBuffer buf = ByteBuffer.allocate(len);
        while(buf.hasRemaining()){
            scx.read(buf);
        }
        String msg = new String(buf.array());
        System.out.println("收到了来自服务器的响应:["+msg+"]");

    }else if(key.isWritable()){
        //--发现了Write操作
        //--获取通道
        SocketChannel scx = (SocketChannel) key.channel();
        //--完成Write操作
        String str = "hello java hello nio hello China~";
        String data = str.getBytes().length+"\r\n"+str;
        ByteBuffer buf = ByteBuffer.wrap(data.getBytes());
        while(buf.hasRemaining()){
```

```
                scx.write(buf);
            }

            //--注册Read操作 接收服务器返回的数据
            scx.register(selc, SelectionKey.OP_READ);
        }else{
            throw new RuntimeException("未知的键~!");
        }
        //--清除处理完成的键
        it.remove();
        }
        }

        }
}
```

总结

2018年8月18日 星期六　　上午 10:50

1. 开源的NIO结构的服务器框架

MINA Netty

2. IO方式的总结

阻塞/非阻塞:

讨论的是线程的角度,当执行某些操作不能立即完成时,线程是否被挂起,失去cpu争夺权 无法继续执行 直到阻塞结束 或被 唤醒

同步/异步:

讨论的是参与通信双方的工作机制,是否需要互相等待对方的执行.

同步指的是 通信过程中 一方在处理通信 另一方 要等待对方执行 不能去做其他无关的事.

异步指的是 通信过程中 一方在处理通信 另一方 可以不用等待对方 而可以去做其他无关的事 直到对方处理通信完成 再在适合的时候继续处理通信过程

BIO	jdk1.0	同步阻塞式IO	面向流 操作字节或字符 单向传输数据
NIO	jdk4.0	同步非阻塞式IO	面向通道 操作缓冲区 双向传输数据
AIO	jdk7.0	异步非阻塞式IO	大量使用回调函数 异步处理通信过程

作业

2018年8月17日 星期五　　下午 5:10

掌握NIO相关概念和原理

理解NIO相关代码 不需要掌握

了解常见的NIO框架

了解三种IO机制的区别

预习Concurrent