NIO 基本应用
(IO 加强)

1.Nio 简介

1.1. NIO 概述

Java NIO（New IO）是从 Java 1.4 版本开始引入的一组新的 IO API（其核心构成有 Channels，Buffers，Selectors 三部分），目的主要是基于这组 API 改善 IO 操作性能。

1.2. NIO&IO 分析

1.2.1. IO 操作流程

对于一个 network IO （这里我们以 read 举例），它会涉及到两个系统对象，一个是调用这个 IO 的 process （or thread），另一个就是系统内核（kernel）。当一个 read 操作发生时，该操作会经历两个阶段：

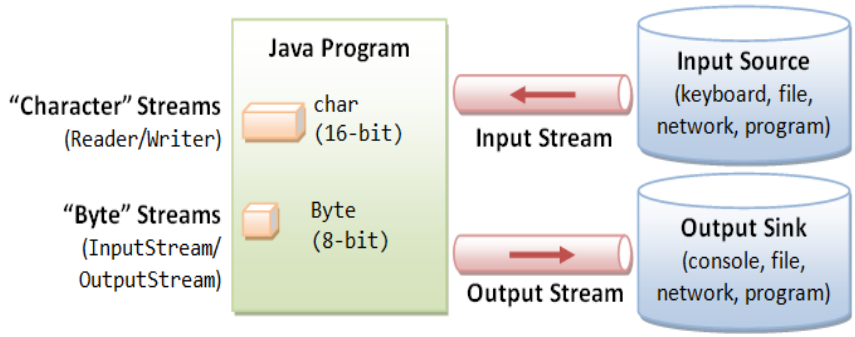
1）将数据拷贝到操作系统内核 Buffer。

2）将操作系统内核数据拷贝到用户进程。

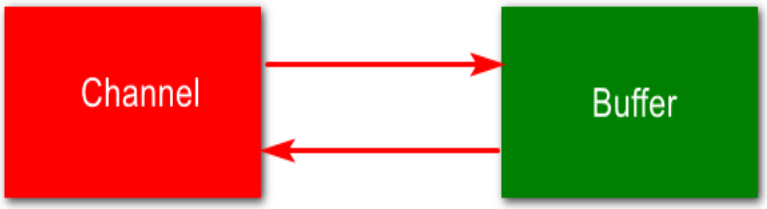
1.2.2. 面向流与面向缓冲区

Java NIO 和 IO 之间第一个最大的区别是：IO 是面向流的，NIO 是面向缓冲区的。例如：

1）面向流的操作（输入&输出流操作是单向的）



2）面向缓冲区的操作：（操作是双向且可移动指针）



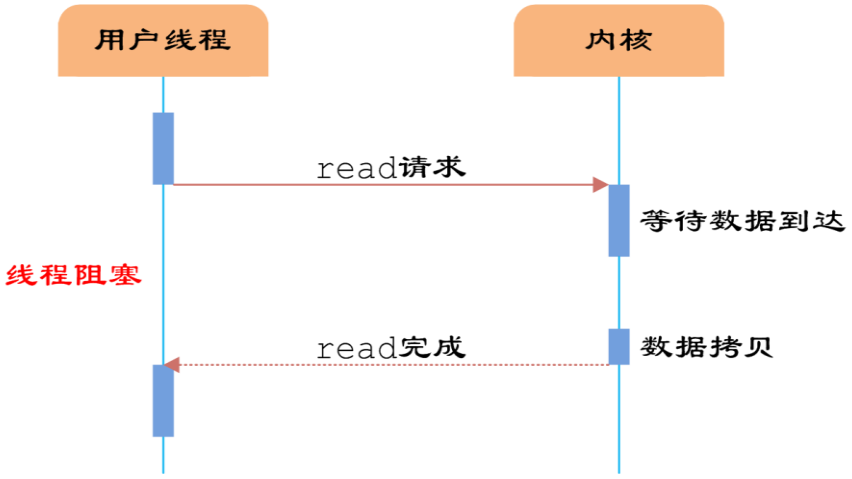
说明：

面向缓冲区的操作时，是缓冲区中的读写模式进行操作，写模式用于向缓冲区写数据，读模式用于从缓冲区读。

1.2.3. 阻塞与非阻塞

阻塞和非阻塞的概念描述的是用户线程调用内核 IO 操作的方式。

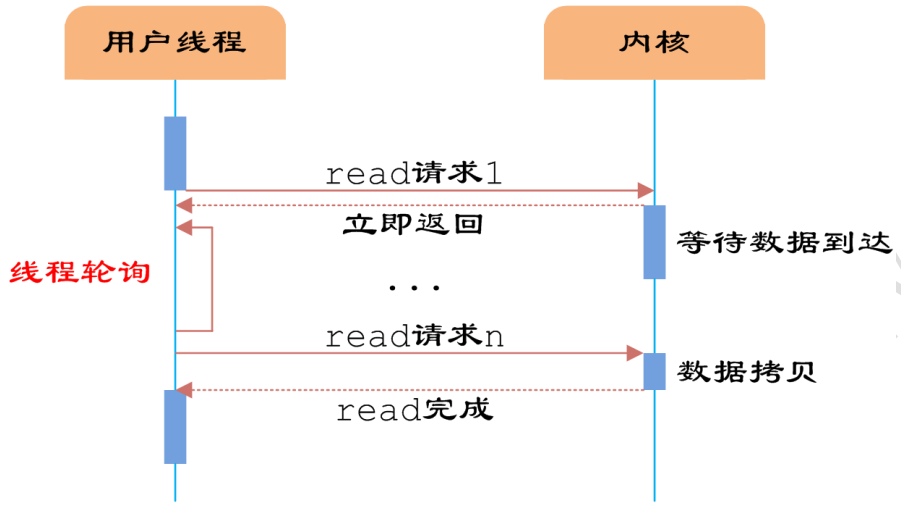
1）阻塞：是指调用操作需要等待结果的完成，同时会影响后操作的执行。例如阻塞式 IO 操作，用户线程会在内核等待数据以及拷贝数据期间都处于阻塞状态。



整个 IO 请求的过程中,用户线程是被阻塞的,这导致用户在发起 IO 请求时,不能做任何事情，对 CPU 的资源利用率不够。

话外音：小李去火车站买票，排队两天买到一张票。

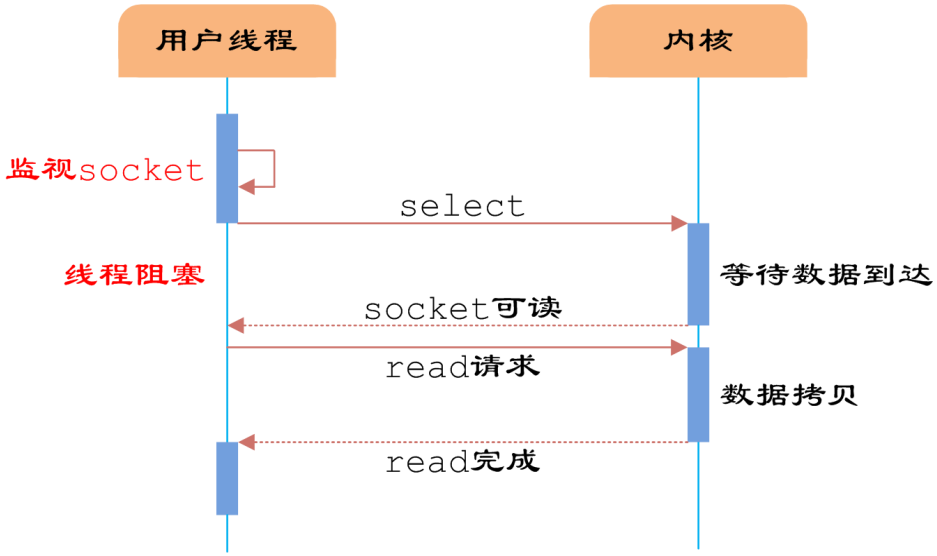
2）非阻塞：是指 IO 操作被调用后立即返回给用户一个状态值，无需等到 IO 操作彻底完成。例如：



虽然用户线程每次发起 IO 请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的 CPU 的资源。

话外音：小李去火车站买票，火车站没票，然后每隔 3 小时去火车站问有没有人退票，两天天后买到一张票.

为了避免同步非阻塞 IO 模型中轮询等待的问题，基于内核提供的多路分离函数 select(),可以实现 IO 的多路复用，例如



使用 select 以后最大的优势是用户可以在一个线程内同时处理多个 socket 的 IO 请求。用户可以注册多个 socket，然后不断地调用 select 读取被激活的 socket，即可达到在同一个线程内同时处理多个 IO 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

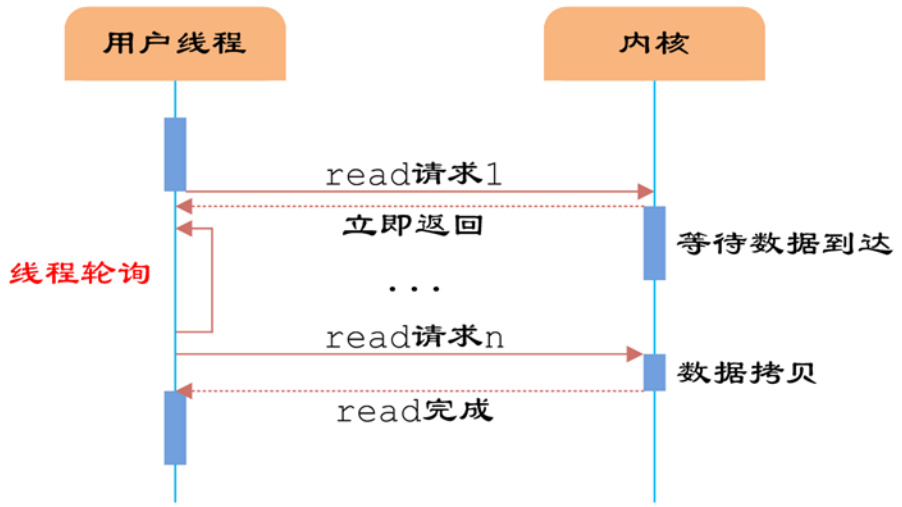
话外音：小李去火车站买票，委托黄牛，然后每隔 1 小时电话黄牛询问，黄牛两天内买到票，然后老李去火车站交钱领票。

这里的 select 函数是阻塞的，因此多路 IO 复用模型也被称为异步阻塞 IO 模型。注意，这里的所说的阻塞是指 select 函数执行时线程被阻塞，而不是指 socket。

1.2.4. 同步与异步

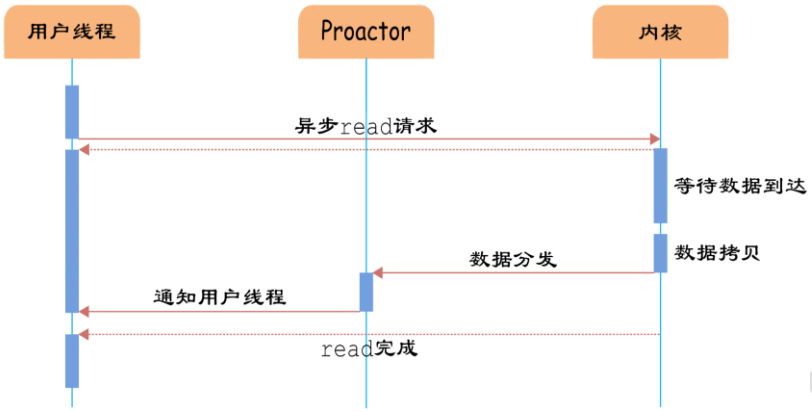
同步和异步的概念描述的是用户线程与内核的交互方式。

1）同步是指用户线程发起 IO 请求后需要等待或者轮询内核 IO 操作完成后才能继续执行。



说明：这种用户线程轮询的改进方式是 IO 多路复用的实现。

2）异步是指用户线程发起 IO 请求后仍继续执行，当内核 IO 操作完成后会通知用户线程，或者调用用户线程注册的回调函数，例如



说明：

异步 IO 的实现需要操作系统的支持，目前系统对异步 IO 的支持并非特别完善，更多的是采用 IO 多路复用模型模拟异步 IO 的方式。

话外音：小李去火车站买票，给售票员留下电话，有票后，售票员电话通知小李并快递送票上门

2.Buffer 基本应用

2.1. Buffer 概述

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。

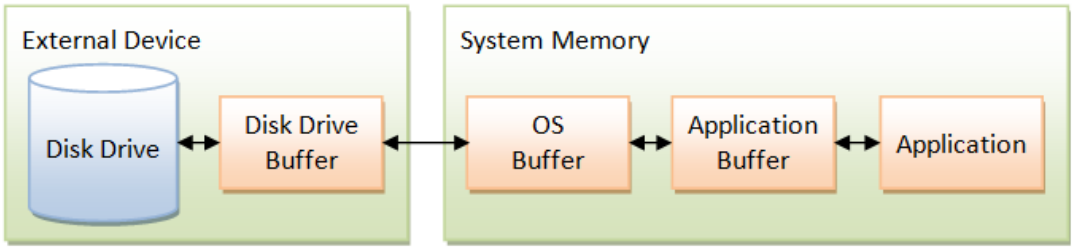
Java NIO 里关键的 Buffer 实现：

1) ByteBuffer
2) CharBuffer
3) DoubleBuffer
4) FloatBuffer
5) IntBuffer
6) LongBuffer
7) ShortBuffer

这些 Buffer 覆盖了你能通过 IO 发送的基本数据类型：byte, short, int, long, float, double 和 char。

说明：

实际的项目中的物理 I/O 操作相比与内存操作要慢数千倍，所以一般为了提高应用程序的性能通常会对数据进行缓存。我们的 JAVA 应用程序和物理磁盘间通常会有多级缓存，例如：



其中：

1) Disk Drive Buffer(磁盘缓存):位于磁盘驱动器中的 RAM，将磁盘数据移动到磁盘缓冲区是一件相当耗时的操作。
2) OS Buffer（系统缓存）：操作系统自己缓存，可以在应用程序间共享数据
3) Application Buffer(应用缓存)：应用程序的私有缓存。

2.2. Buffer 基本应用

使用 Buffer 读写数据一般遵循以下四个步骤:

1) 写入数据到 Buffer
2) 调用 flip()方法
3) 从 Buffer 中读取数据
4) 调用 clear()方法或者 compact()方法

当向 buffer 写入数据时，buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip()方法将 Buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 buffer 的所有数据。
一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear()或 compact()方法。clear()方法会清空整个缓冲区。compact()方法只会清除已经读过的数据。任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。

```
@Test
public void testBuffer01(){
    //构建一个缓冲区对象(在JVM内存中分配一块区域)
    ByteBuffer buf=ByteBuffer.allocate(1024);
    System.out.println("===数据写入前===");
    doPrint(buf.position(),buf.limit(),buf.capacity());
    //向缓冲区写入数据
    byte []data="hello".getBytes();
    buf.put(data);//放入缓冲区
    System.out.println("===数据写入后===");
    doPrint(buf.position(),buf.limit(),buf.capacity());
    //切换模式(底层同时会移动指针,position位置会发生变换)
    buf.flip();
    System.out.println("===读数据之前===");
    doPrint(buf.position(),buf.limit(),buf.capacity());
```

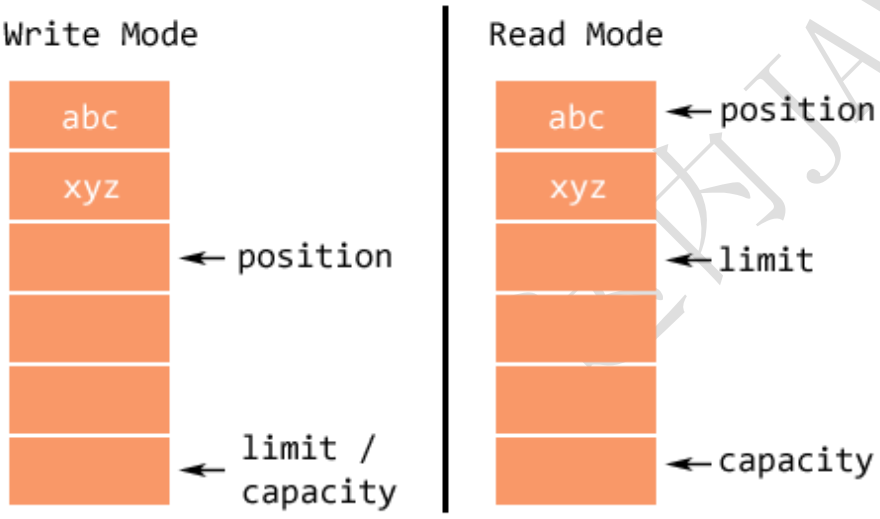
```
        byte c1=buf.get();
        System.out.println((char)c1);
        System.out.println("===读数据之后===");
        doPrint(buf.position(),buf.limit(),buf.capacity());
    }
```

```
private void doPrint(int pos,int limit,int cap){
        System.out.println("position:"+pos);
        System.out.println("limit:"+limit);
        System.out.println("capacity:"+cap);
}
```

缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。
为了理解 Buffer 的工作原理，需要熟悉它的三个属性：

1) capacity 容量
2) position 位置
3) limit 限制

position 和 limit 的含义取决于 Buffer 处在读模式还是写模式。不管 Buffer 处在什么模式，capacity 的含义总是一样的。



0 <= mark <= position <= limit <= capacity

Capacity

作为一个内存块，Buffer 有一个固定的大小值，也叫“capacity”.你只能往里写 capacity 个 byte、long，char 等类型数据。一旦 Buffer 满了，需要将其清空（通过读数据或者清除数据）才能继续往里写数据。

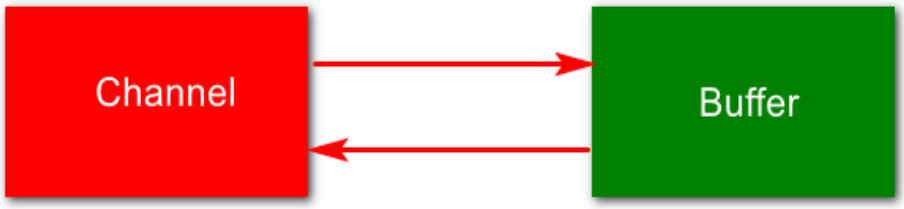
position
当你写数据到 Buffer 中时，position 表示当前的位置。初始的 position 值为 0。当一个 byte、long 等数据写到 Buffer 后， position 会向前移动到下一个可插入数据的 Buffer 单元。position 最大可为 capacity – 1.
当读取数据时，也是从某个特定位置读。当将 Buffer 从写模式切换到读模式，position 会被重置为 0. 当从 Buffer 的 position 处读取数据时，position 向前移动到下一个可读的位置。

limit
在写模式下，Buffer 的 limit 表示你最多能往 Buffer 里写多少数据。 写模式下，limit 等于 Buffer 的 capacity。
当切换 Buffer 到读模式时， limit 表示你最多能读到多少数据。因此，当切换 Buffer 到读模式时，limit 会被设置成写模式下的 position 值。换句话说，你能读到之前写入的所有数据（limit 被设置成已写数据的数量，这个值在写模式下就是 position）

3.Channel 基本应用

3.1. Channel 概述

NIO 是基于通道（Channel）和缓冲区（Buffer）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。如图所示：



NIO 中 Channel 的一些具体实现类有：

1) FileChannel ：从文件中读写数据。
2) DatagramChannel ：能通过 UDP 读写网络中的数据。
3) SocketChannel ：能通过 TCP 读写网络中的数据。
4) ServerSocketChannel 可以监听新进来的 TCP 连接，像 Web 服务器那样。

正如你所看到的，这些通道涵盖了 UDP 和 TCP 网络 IO，以及文件 IO。

3.2. FileChannel 基本应用

借助 Channel 对象（FileChannel 类型），从文件读取数据。代码示例：

案例 1：

```
@Test
    public void testFileChannel()throws Exception{
        //构建一个Buffer对象(缓冲区)：JVM内存
        ByteBuffer buf=ByteBuffer.allocate(1024);
        //构建一个文件通道对象(可以读写数据)
        FileChannel fChannel=
        FileChannel.open(Paths.get("data.txt"),
        StandardOpenOption.READ);//读模式
        //将文件内容读到缓冲区(ByteBuffer)
        fChannel.read(buf);
        System.out.println("切换buf模式，开始从buf读数据");
        System.out.println(buf.position());
        //从Buffer中取数据
        buf.flip();
        System.out.println(buf.position());
        System.out.println(new String(buf.array()));
        buf.clear();//不是是清除，而将数据标记为脏数据(无用数据)
        //释放资源
        fChannel.close();
```

```
    }
```

案例 2：

```
@Test
    public void testFileChannel()throws Exception{
        //构建一个Buffer对象(缓冲区)：JVM内存
        ByteBuffer buf=ByteBuffer.allocate(2);
        //构建一个文件通道对象(可以读写数据)
        FileChannel fChannel=
        FileChannel.open(Paths.get("data.txt"),
        StandardOpenOption.READ);//读模式
        //将文件内容读到缓冲区(ByteBuffer)
        int len=-1;
        do{
         len=fChannel.read(buf);
         System.out.println("切换buf模式，开始从buf读数据");
         buf.flip();
         //判定缓冲区中是否有剩余数据
         while(buf.hasRemaining()){
         System.out.print((char)buf.get());//每次都1个字节
         }
         System.out.println();
         buf.clear();//每次读数据应将原数据设置为无效。
        }while(len!=-1);
        //释放资源
        fChannel.close();
    }
```

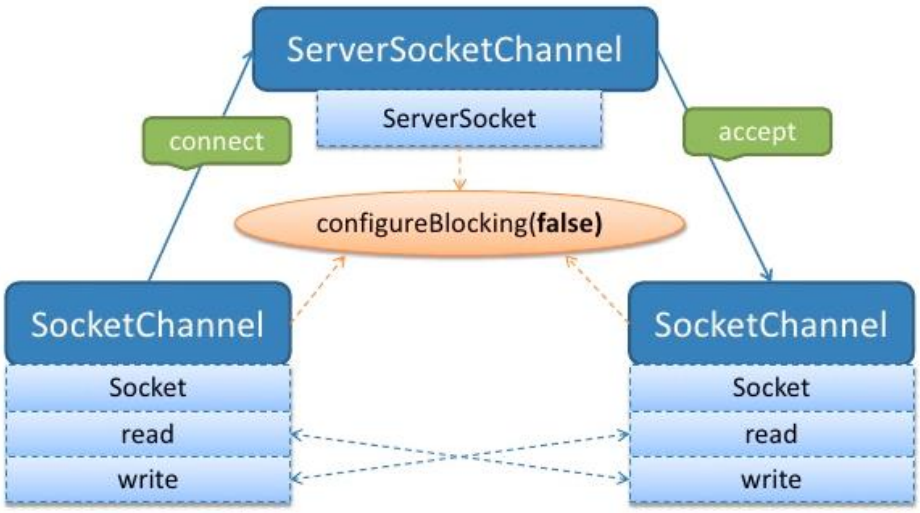
3.3. SocketChanel 基本应用

Java NIO 中的 SocketChannel 是一个连接到 TCP 网络另一端的通道。可以通过以下 2 种方式创建 SocketChannel：

1）客户端打开一个 SocketChannel 并连接到互联网上的某台服务器。

2）服务端一个新连接到达 ServerSocketChannel 时，会创建一个 SocketChannel。

基于 channel 实现通讯



代码示例:

Client 代码示例:

```
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.connect(new InetSocketAddress("127.0.0.1", 9999));
        String newData = "New String to write to file..." +
        System.currentTimeMillis();
        ByteBuffer buf = ByteBuffer.allocate(48);
        buf.clear();
        buf.put(newData.getBytes());
        buf.flip();
        while(buf.hasRemaining()) {
            socketChannel.write(buf);
        }
        socketChannel.close()
```

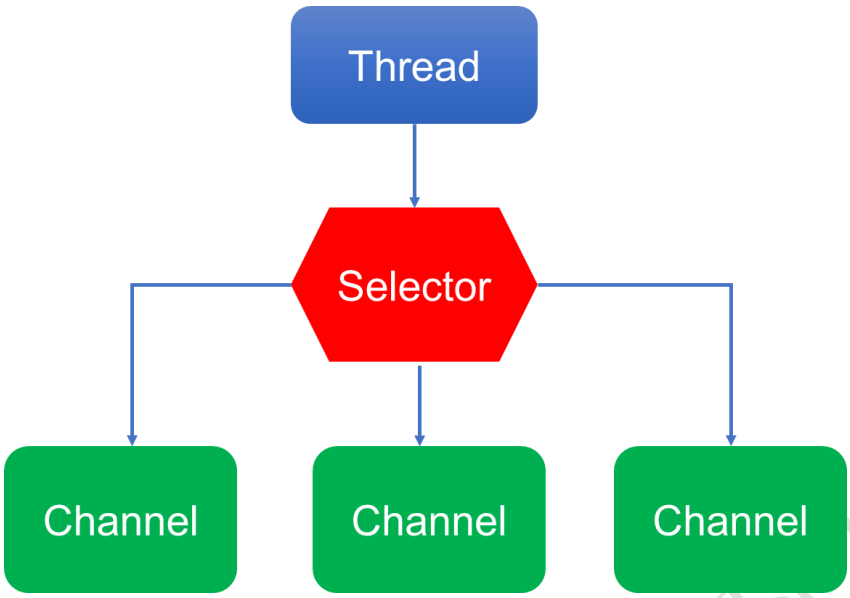
Server 端代码实现:

```
      ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
      serverSocketChannel.socket().bind(new InetSocketAddress(9999));
      ByteBuffer byteBuffer=ByteBuffer.allocate(48);
      while(true){
            SocketChannel socketChannel =serverSocketChannel.accept();
            int byteReader = socketChannel.read(byteBuffer);
            System.out.println(new String(byteBuffer));
           socketChannel.close();
        }
}
```

4.Selector 基本应用

4.1. Selector 概述

Selector 是 Java NIO 中实现多路复用技术的关键，多路复用技术又是提高通讯性能的一个重要因素。项目中可以基于 selector 对象实现了一个线程管理多个 channel 对象，多个网络链接的目的。例如：在一个单线程中使用一个 Selector 处理 3 个 Channel，如图所示



为什么使用 Selector?

仅用单个线程来处理多个 Channel 的好处是：只用一个线程处理所有的通道，可以有效避免线程之间上下文切换带来的开销，而且每个线程都要占用系统的一些资源（如内存）。因此，使用的线程越少越好。

4.2. Selector 基本应用

Selector 的创建

通过调用 Selector.open()方法创建一个 Selector，如下：

```
Selector selector = Selector.open();
```

向 Selector 注册通道

为了将 Channel 和 Selector 配合使用,必须将 channel 注册到 selector 上。通过 SelectableChannel.register()方法来实现，如下：

```
channel.configureBlocking(false);
SelectionKey key = channel.register(selector,Selectionkey.OP_READ);
```

channel 与 Selector 一起使用时，Channel 必须处于非阻塞模式下。这意味着不能将 FileChannel 与 Selector 一起使用，因为 FileChannel 不能切换到非阻塞模式。而套接字通道都可以。

注意 register()方法的第二个参数。这是一个“interest 集合”，意思是在通过 Selector 监听 Channel 时对什么事件感兴趣。可以监听四种不同类型的事件：

1) Connect
2) Accept
3) Read
4) Write

通道触发了一个事件意思是该事件已经就绪。所以，某个 channel 成功连接到另一个服务器称为“连接就绪”。一个 server socket channel 准备好接收新进入的连接称为“接收就绪”。一个有数据可读的通道可以说是“读就绪”。等待写数据的通道可以说是“写就绪”。

这四种事件用 SelectionKey 的四个常量来表示：

1) SelectionKey.OP_CONNECT
2) SelectionKey.OP_ACCEPT
3) SelectionKey.OP_READ

4) SelectionKey.OP_WRITE

代理示例如下：

服务端实现：

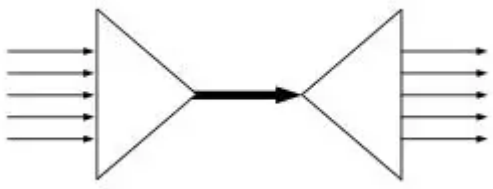
```
//1. 获取通道
ServerSocketChannel ssChannel = ServerSocketChannel.open();
//2. 切换非阻塞模式
ssChannel.configureBlocking(false);
//3. 绑定连接
ssChannel.bind(new InetSocketAddress(9898));
//4. 获取选择器
Selector selector = Selector.open();
//5. 将通道注册到选择器上, 并且指定“监听接收事件”
ssChannel.register(selector, SelectionKey.OP_ACCEPT);
//6. 轮询式的获取选择器上已经“准备就绪”的事件
while(selector.select() > 0){
//7. 获取当前选择器中所有注册的“选择键(已就绪的监听事件)”
Iterator<SelectionKey> it = selector.selectedKeys().iterator();
while(it.hasNext()){
//8. 获取准备“就绪”的是事件
SelectionKey sk = it.next();
//9. 判断具体是什么事件准备就绪
if(sk.isAcceptable()){
//10. 若“接收就绪”，获取客户端连接
SocketChannel sChannel = ssChannel.accept();
//11. 切换非阻塞模式
sChannel.configureBlocking(false);
//12. 将该通道注册到选择器上
sChannel.register(selector, SelectionKey.OP_READ);
}else if(sk.isReadable()){
//13. 获取当前选择器上“读就绪”状态的通道
SocketChannel sChannel = (SocketChannel) sk.channel();
//14. 读取数据
ByteBuffer buf = ByteBuffer.allocate(1024);
int len = 0;
while((len = sChannel.read(buf)) > 0 ){
      buf.flip();
      System.out.println(new String(buf.array(), 0, len));
      buf.clear();
```

```
}
}
//15. 取消选择键 SelectionKey
  it.remove();
}
}
```

客户端实现：

```
//1. 获取通道
SocketChannel sChannel = SocketChannel.open(new
InetSocketAddress("127.0.0.1", 9898));
//2. 切换非阻塞模式
sChannel.configureBlocking(false);
//3. 分配指定大小的缓冲区
ByteBuffer buf = ByteBuffer.allocate(1024);
//4. 发送数据给服务端
Scanner scan = new Scanner(System.in);
while(scan.hasNext()){
String str = scan.next();
            buf.put((new Date().toString() + "\n" + str).getBytes());
            buf.flip();
            sChannel.write(buf);
            buf.clear();
}
//5. 关闭通道
sChannel.close();
```

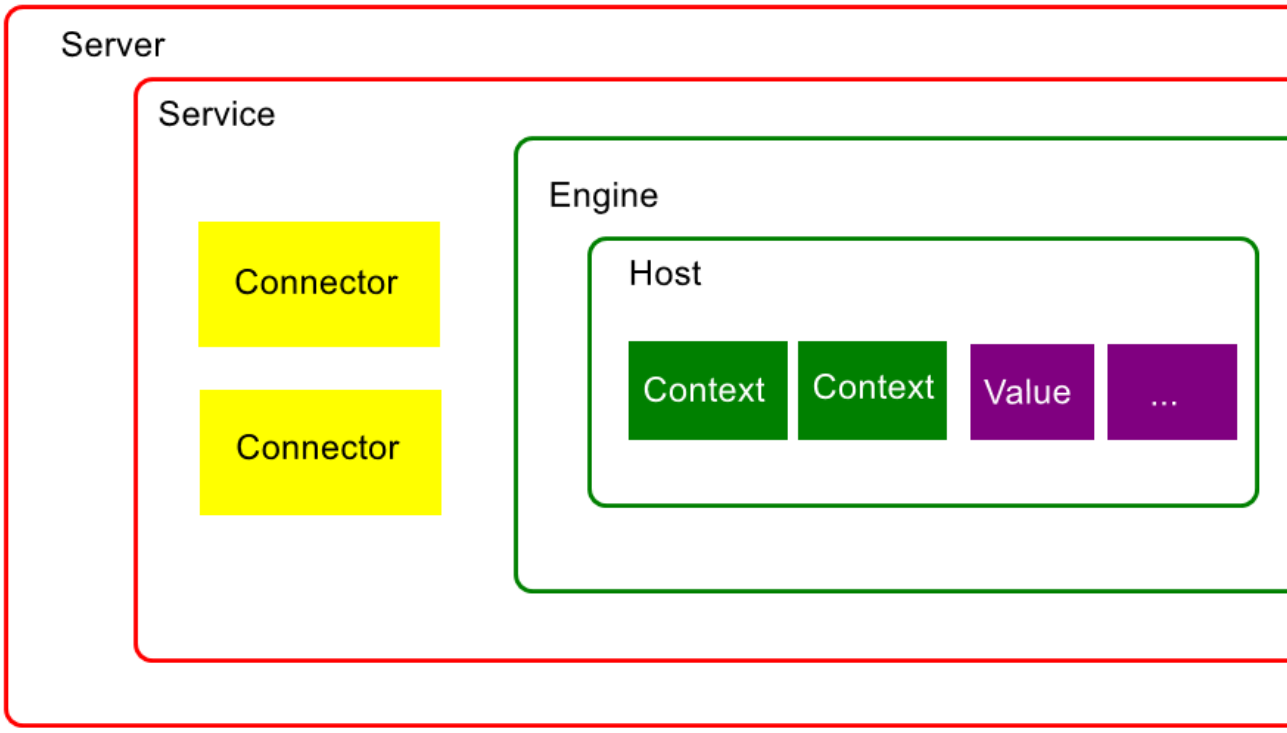
程序运行分析如下：



5.Tomcat 中的 NIO 应用

5.1. Tomcat 核心架构

Tomcat 是一个 apache 推出的一个 web 应用服务器，核心功能就是解析 Http 协议，处理网络 IO 操作，执行 Servlet 对象，其简易架构如下：



其中：

1) Server：代表整个容器，它可能包含一个或多个 Service 和全局的对象资源；
2) Service：包含一个或多个 Connector 和一个 Engine，这些 Connector 和 Engine 相关联；
3) Connector：处理与客户端的通信，网络 I/O 操作；
4) Engine：表示请求处理流水线（pipeline），它接收所有连接器的请求，并将响应交给适当的连接器返回给客户端；
5) Host：网络名称(域名)与 Tomcat 服务器的关联，默认主机名 localhost，一个 Engine 可包含多个 Host；

6) Context：表示一个 Web 应用程序，一个 Host 包含多个上下文。
7) …

5.2. Tomcat 中的 NIO 应用配置

Tomcat 中的 NIO 应用要从 Connector 说起，Connector 是请求接收环节与请求处理环节的连接器。具体点说，就是 Connector 将接收到的请求传递给 Tomcat 引擎(Engine)进行处理，引擎处理完成以后会交给 Connector 将其响应到客户端。但是 Connector 本身并不会读写网络中的数据，读写网络中的数据还是要基于网络 IO 进行实现。但使用哪种 IO 模型需要由 Connector 对象进行指定。

例如：可在 tomcat 的 server.xml 进行配置，其默认配置如下：

```
<Connector connectionTimeout="20000"
           port="8080"
           protocol="HTTP/1.1"
           redirectPort="8443"/>
```

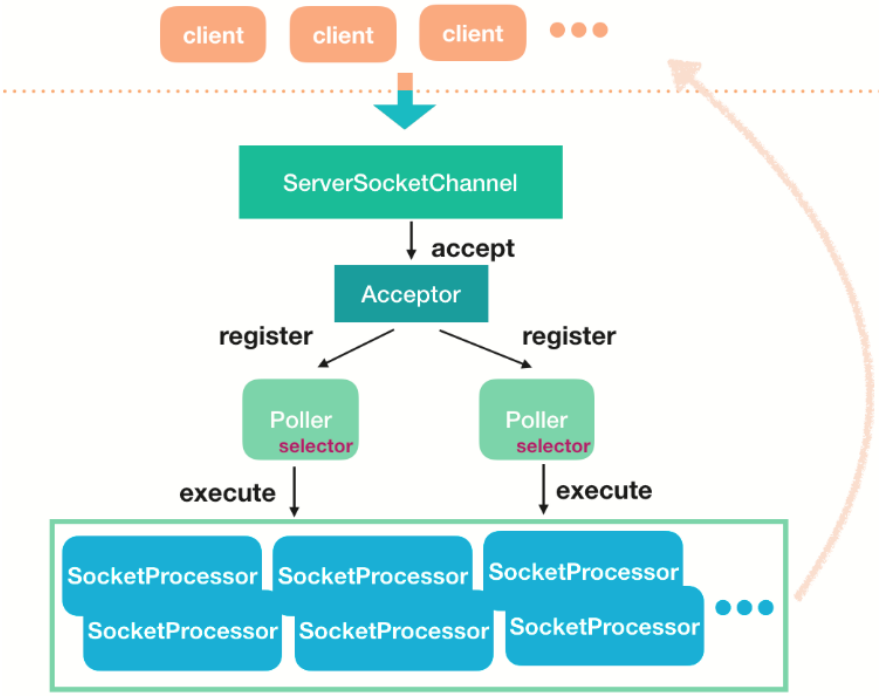
一个 Tomcat 中可以配置多个 Connector，分别用于监听不同端口，或处理不同协议，在如上配置中 Connector 使用的协议默认为”HTTP/1.1”，系统底层会基于此配置，通过反射创建 Http11NioProtocol 对象，而此对象底层就是基于 NIO 中的 IO 多路复用技术实现网路数据读写的。假如希望使用其它协议或 NIO 方式可以修改其默认配置。例如：我们配置 NIO 中的异步应用模型。

```
<Connector connectionTimeout="20000"
           port="8080"
           protocol="org.apache.coyote.http11.Http11Nio2Protocol"
           redirectPort="8443"/>
```

5.3. Tomcat 中的 NIO 应用设计

在 tomcat 中，目前 IO 模型的最佳应用模式还是 IO 多路复用，因为 BIO 的缺点在于不管当前连接有没有数据传输，它始终阻塞占用线程池内的一个线程，而 NIO 的处理方式是若通道无数据可读取，此时线程不阻塞直接返回，可用于处理

其他连接，提高了线程利用率。其工作模型大致如下：



1) Acceptor 以阻塞模式接收 TCP 连接，然后对连接信息进行封装并以事件方式注册（register）到 Poller 上；
2) Poller 进行事件迭代，循环执行 selector.select(xxx)，如果有通道 readable，那么在 processKey 方法中交给 worker 线程池中的线程处理。

说明：假如想在 maven 项目中想对 tomcat 的源码进行快速分析，可先添加如下依赖，基于此依赖可借助 maven 对 tomcat 源码进行组织。

```
<dependency>
<groupId>org.apache.tomcat.embed</groupId>
<artifactId>tomcat-embed-core</artifactId>
<version>9.0.6</version>
</dependency>
```

基于 tomcat 依赖编写如下代码启动 tomcat，进行 debug 分析。

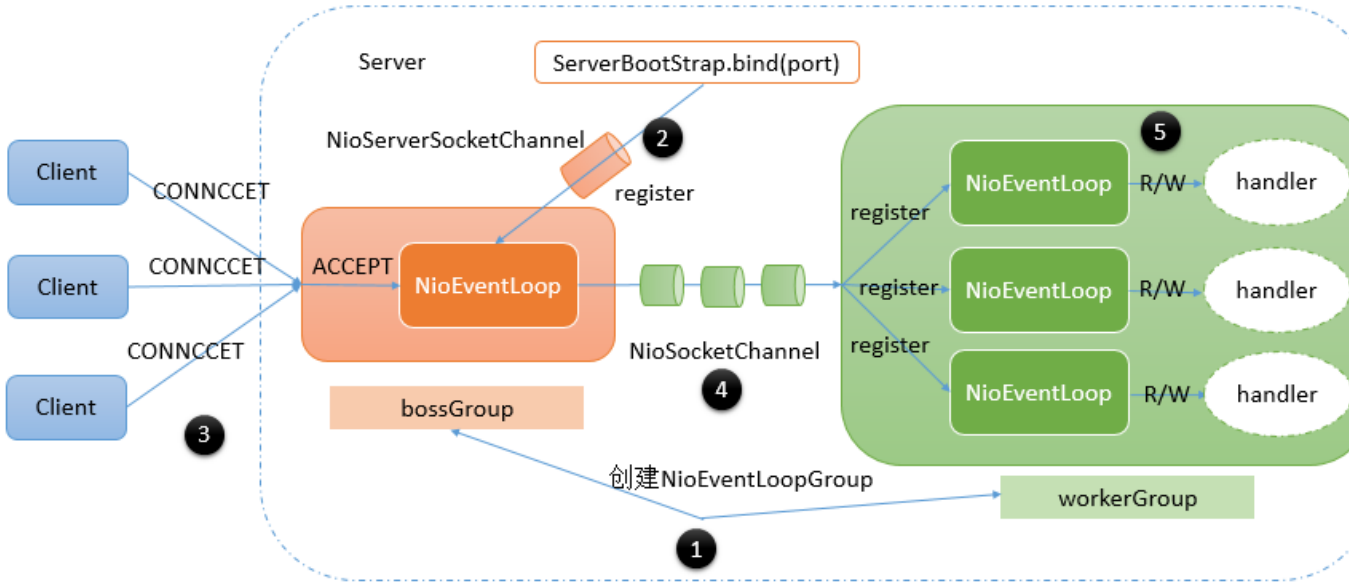
```
public static void main(String[] args)throws Exception {
    //1.构建tomcat对象
    Tomcat tomcat = new Tomcat();
    //2.构建connector对象，并指定协议
    //tomcat实用connector处理链接，一个tomcat可以配置多个connector
    Connector connector = new Connector("HTTP/1.1");
```

```
        //3.设置tomcat舰艇端口
        connector.setPort(8080);
        tomcat.setConnector(connector);
        //启动tomcat
        tomcat.start();
        tomcat.getServer().await();
    }
```

6.Netty 中的 NIO 应用

6.1. Netty 中的 NIO 模型分析

Netty 是一个基于 NIO 技术的网络编程框架，底层实现了对 java 中 NIO API 的封装。它基于异步事件驱动，可以快速开发高性能网络应用程序，并在可维护性方面有很好的表现。



Netty 的健壮性、功能、性能、可定制性和可扩展性在同类框架中都首屈一指，它已经得到成百上千的商用项目验证，当然这些都离不开它背后的 NIO 技术，线程技术的合理应用。

6.2. Netty 中的 NIO 案例分析

本小节基于一个时间服务器对象与时间客户端对象的通讯为案例，分析一下 netty 中 NIO 的应用.
本项目采用的 netty 为 4.x 的版本，例如

```
    <dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-all</artifactId>
    <version>4.1.16.Final</version>
    </dependency>
```

6.2.1. Netty 服务端入门实现

1．服务端创建关键步骤

1）创建服务端启动类对象（ServerBootstrap）
2）设置线程组（Boss 线程组和 Worker 线程组）
3）设置服务端 channel 对象(NioServerSocketChannel)
4）设置 ChanelHandler 对象
5）绑定并启动端口监听（等待客户端链接）

2．服务端代码实现

创建事件服务器

```
public class TimeServer {
    public void start() 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>() {
                public void initChannel(SocketChannel ch) throws Exception {
                    ch.pipeline().addLast(new TimeServerHandler());
```

```
                };
            })
            .childOption(ChannelOption.SO_KEEPALIVE, true);
            ChannelFuture f=b.bind(9999).sync();
            f.channel().closeFuture().sync();
        }finally{
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
    public static void main(String[] args)throws Exception {
        new TimeServer().start();
    }
}
```

创建服务端时间处理器

```
public class TimeServerHandler extends ChannelInboundHandlerAdapter  {
    @Override
    public void channelActive(final ChannelHandlerContext ctx)
            throws Exception {
        final ByteBuf time=ctx.alloc().buffer(8);
        time.writeLong(System.currentTimeMillis());
        final ChannelFuture f=ctx.writeAndFlush(time);
        f.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future)
                    throws Exception {
                assert f == future;
                ctx.close();
            }
        });
    }
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
{
        cause.printStackTrace();
        ctx.close();
    }
}
```

6.2.2. Netty 客户端入门实现

1．客户端创建关键步骤

1）创建服务端启动类对象（Bootstrap）
2）设置线程组（Worker 线程组）
3）设置客户端 channel 对象(NioSocketChannel)
4）设置 ChanelHandler 对象
5）连接服务端

2．客户端代码实现

创建时间客户端

```
public class TimeClient {
    public void connect(String ip,Integer port)throws Exception {
        EventLoopGroup workerGroup=new NioEventLoopGroup();
        try {
            //Bootstrap客户端用于简单建立Channel
            Bootstrap b=new Bootstrap();
            b.group(workerGroup);
            //NioSocketChannel用于客户端创建Channel
            b.channel(NioSocketChannel.class);
            b.option(ChannelOption.SO_KEEPALIVE,true);
            b.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel ch) throws Exception {
                    //指定使用的数据处理方式
                    ch.pipeline().addLast(new TimeClientHandler());
                };
            });
            //客户端开始连接
            ChannelFuture f=b.connect(ip,port).sync();
            //等待直到这个连接被关闭
            f.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
        }
    }
    public static void main(String[] args)throws Exception {
        new TimeClient().connect("127.0.0.1", 9999);
```

```
    }
}
```

创建客户端时间处理器

```
public class TimeClientHandler extends ChannelInboundHandlerAdapter {
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg)
            throws Exception {
        ByteBuf m=(ByteBuf)msg;
        try {
            long currentTimeMillis = m.readLong();
            System.out.println("from server:"+new Date(currentTimeMillis));
            ctx.close();
        } finally {
            m.release();
        }
    }
}
```

7.NIO 应用总结分析

7.1. 重难点分析

1. 常见的 IO 操作应用模型

1) 同步阻塞 IO（Blocking IO）：系统内核拷贝数据期间用户线程被阻塞。
2) 同步非阻塞 IO（Non-blocking IO）：系统内核拷贝数据期间用户线程

轮询，造成 CPU 资源浪费。

3）IO 多路复用 (IO Multiplexing)：减少 CPU 在用户线程间的切换时间。

4）异步 IO（Asynchronous IO）：需要操作系统支持，目前还不够完善。

2．JDK NIO 中核心 API 对象有哪些?

1）缓冲区对象:Buffer

2）通道对象:Channel

3）选择器对象:Selector

3．NIO 框架的基本应用?

1)Tomcat 中 NIO 的应用

2)Netty 中 NIO 的应用

3)…

7.2. 相关 FAQ

1．NIO 给我们带来了哪些特性应用?

1）事件驱动，单线程多任务

2）非阻塞 I/O（I/O 读写不再阻塞，而是返回 0）

3）基于 block 的传输，通常比基于流的传输更高效

4）IO 多路复用大大提高了 Java 网络应用的可伸缩性和实用性

2．NIO 还存在哪些问题呢?

1）使用 NIO != 高性能，当连接数<1000，并发程度不高或者局域网环境下 NIO 并没有显著的性能优势。

2）NIO 并没有完全屏蔽平台差异，它仍然是基于各个操作系统的 I/O 系统实现的，差异仍然存在。使用 NIO 做网络编程构建事件驱动模型并不容易，陷阱重重。

3）推荐大家使用成熟的 NIO 框架，如 Netty，MINA 等。解决了很多 NIO 的陷阱，并屏蔽了操作系统的差异，有较好的性能和编程模型。