一、 **BIO**

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）。

TCPServer.java

```
package com.open.nio.bio;


import java.io.InputStream;

import java.net.ServerSocket;

import java.net.Socket;


//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); //端口号


        while (true) {

            //2.监听客户端

            System.out.println("来呀");

            Socket s = ss.accept(); //阻塞

            System.out.println("来呀");
```

```
            //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。

producttype/pom.xml

```
package com.open.nio.bio;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;
```

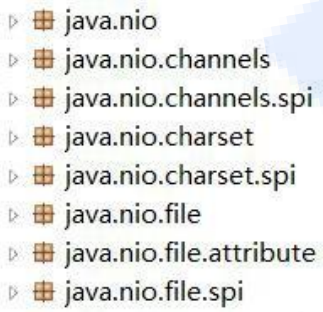
```java
//BIO 客户端程序

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。造成乱码的问题是因为读取的信息是按照特定编码读取的字节流信息，读取的时候受到读取限定，就有可能出现读取的不是一个完整的字节数组信息。

二、 NIO

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。



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；

1)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。

2)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(选择区)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

2. 网络 IO

1) 概述

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。

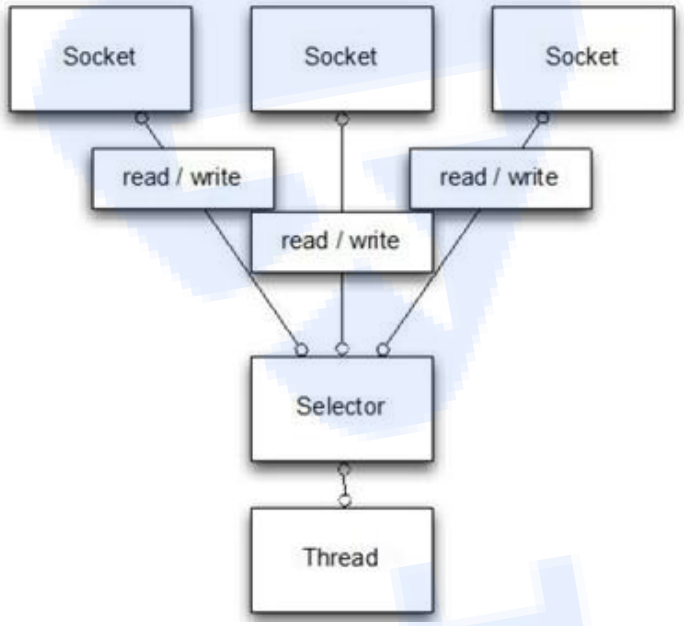
在 Java 中编写 Socket 服务器，通常有以下几种模式：

- 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。

2) Selector(选择器)

能够检测多个注册的通道上是否有事件发生（读、写、连接），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

3) SelectionKey

代表了 Selector 和网络通道的注册关系,一共四种（就是连接事件）

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

4) 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：

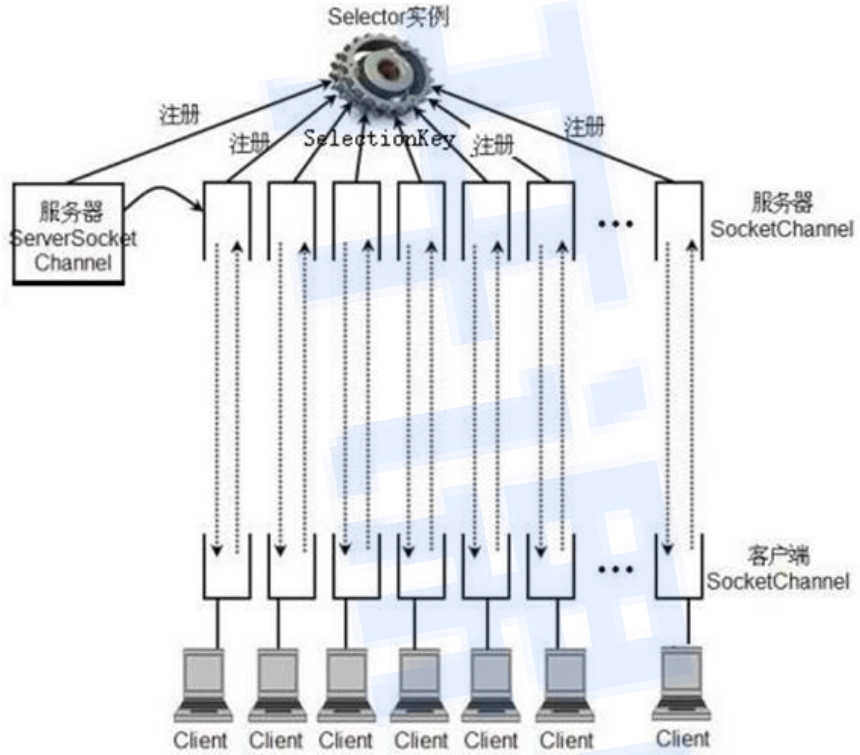
- public static ServerSocketChannel open()，得到一个 ServerSocketChannel 通道
- public final ServerSocketChannel bind(SocketAddress local)，设置服务器端端口号
- public final SelectableChannel configureBlocking(boolean block)，设置阻塞或非阻塞模式， 取值 false 表示采用非阻塞模式
- public SocketChannel accept()，接受一个连接，返回代表这个连接的通道对象

- public final SelectionKey register(Selector sel, int ops)，注册一个选择器并设置监听事件

5) 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：

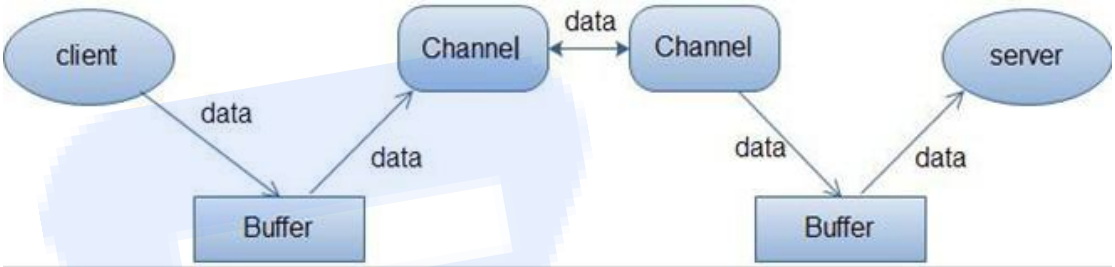
- public static SocketChannel open()，得到一个 SocketChannel 通道
- public final SelectableChannel configureBlocking(boolean block)，设置阻塞或非阻塞模式， 取值 false 表示采用非阻塞模式
- public boolean connect(SocketAddress remote)，连接服务器
- public boolean finishConnect()，如果上面的方法连接失败，接下来就要通过该方法完成连接操作
- public int write(ByteBuffer 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



服务器端有一个选择器对象 selector，服务器的 ServerSocketChannel 对象也要注册给 selector,它的 accept 方法负责接收客户端的连接请求。有一个客户端连接过来，服务端就会建立一个通道。Selector 会监控所有注册的通道，检查这些通道中是否有事件发生。（连接、断开、读、写等事件），如果某个通道有事件发生则做相应的处理。

6) 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）。



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

socket/NIOClient.java

```
package com.open.nio.socket;


import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.SocketChannel;


//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，如果用 connect()方法连接服务器不成功，则用

finishConnect()方法进行连接
```

```
            if(!channel.connect(address)){
        //因为连接需要花时间，所以用 while 一直去尝试连接。在连接服务端时还可以做别的事，体现非阻塞。
         while(!channel.finishConnect()){
                        //nio 作为非阻塞式的优势，如果服务器没有响应（不启动服务端），客户端不会阻塞，最后会报错，客户端尝试链接服务器连不上。
                        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 服务器";
            ByteBuffer writeBuf = ByteBuffer.wrap(msg.getBytes());
            //6. 发送数据
            channel.write(writeBuf);
            //阻止客户端停止，否则服务端也会停止。
            System.in.read();
        }
}
```

socket/NIOServer.java

```
package com.open.nio.socket;
```

```
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;


//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，服务端不能停
        while (true) {
            //6.1 监控客户端 ，selecto.select()方法返回的是客户端的通道数,如果为 0,
则说明没有客户端连接。
            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();
                //客户端先连接上，处理连接事件，然后客户端会向服务端发信息，再处
理读取客户端数据事件。
                if (key.isAcceptable()) {   //客户端连接请求事件
                    System.out.println("OP_ACCEPT");
                    SocketChannel socketChannel =
serverSocketChannel.accept();
                    socketChannel.configureBlocking(false);
        //注册通道 ,将通道交给 selector 选择器进行监控.第一个参数是选择器,
                    //第二个参数服务器要监控读事件，因为客户端发数据，服务端要读
取，第三个参数，客户端传过来的数据要放在缓冲区
```

```
                    socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));
                }
                if (key.isReadable()) {  //读取客户端数据事件
                    //数据在通道中，先得到通道
                    SocketChannel channel = (SocketChannel) key.channel();
                    //取到一个缓冲区，nio 读写数据都是基于缓冲区。
                    ByteBuffer buffer = (ByteBuffer) key.attachment();
                    //从通道中将客户端发来的数据读到缓冲区
                    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

三、 AIO 编程

1. 概念

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。

2. 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

BIO:上厕所，坑位满了，只能等待，不可以干别的事。

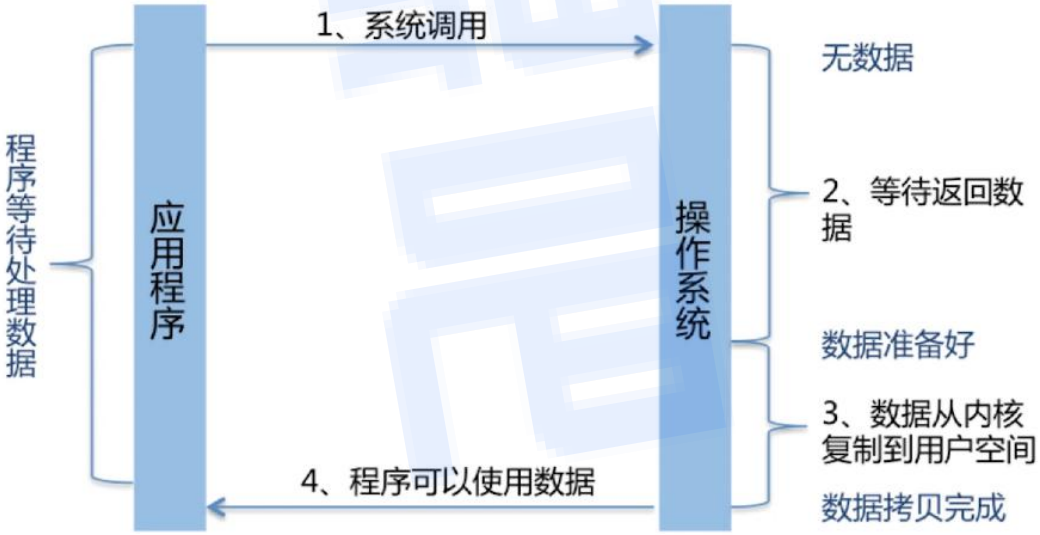
NIO:上厕所，坑位满了，可以在等待时干别的事，抽烟，打游戏。

AIO:上厕所，坑位满了，可以在等待时干别的事，抽烟，打游戏。当有位子了会发一条信息通知你。

四、 常见面试题

1. BIO

InputStream 和 OutputStream，Reader 和 Writer

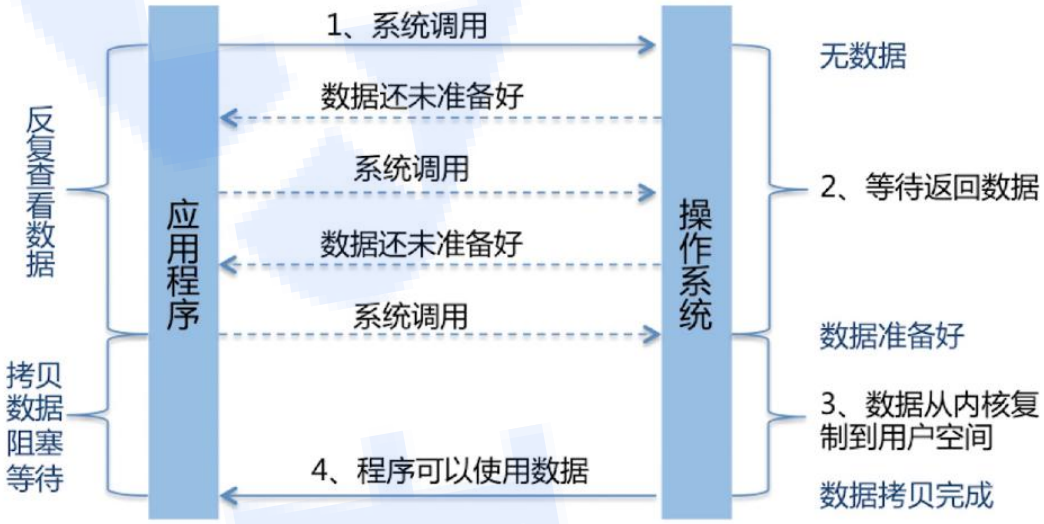


特点：在 IO 执行的两个阶段都被阻塞，优点代码简单，易于实现，缺点是 IO 效率和扩展性存在瓶颈。

2. **NIO**

构建多路复用的、同步非阻塞的 IO 操作。

IO 多路复用：调用系统级别的 select\poll\epoll，优点在于单线程可以处理多个网络 IO



特点：

在发起第一次请求后，线程不会被阻塞，会反复检查数据是否准备好。客户端发出的一次请求就是一个线程，都会被注册到多路复用器上，多路复用器轮询到有 IO 请求时才启动一个线程进行处理，需要不断的主动询问操作系统，数据是否已经准备好。

核心：

Channels:数据可以从 Channel 读取到 Buffer 中，也可以从 Buffer 写到 Channel 中。

Buffers

Selectors：Java NIO 的选择器允许一个单独的线程来监视多个输入通道，可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入或者选择已准备写入的通道。

select、poll、epoll 区别：支持一个进程可以打开的最大连接数。

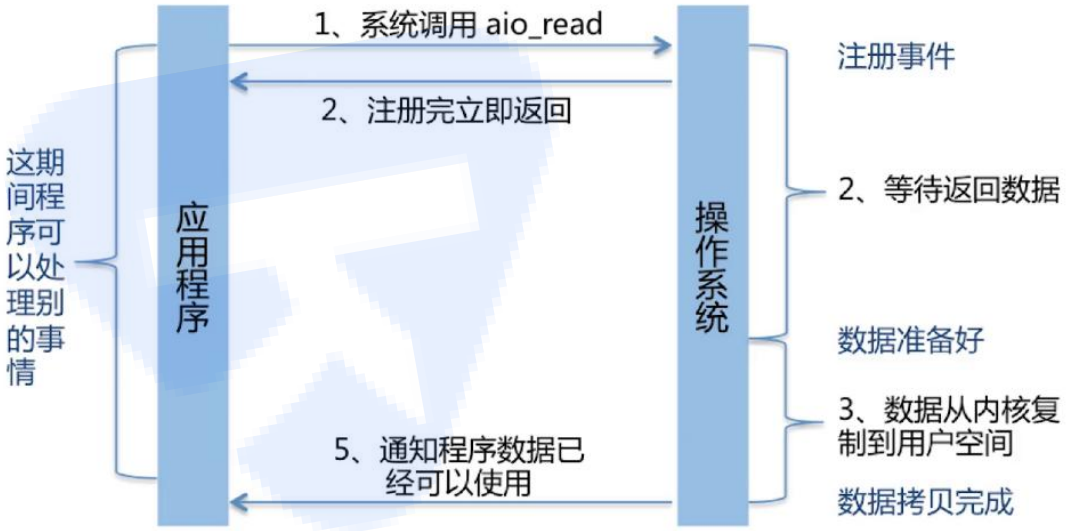
Select:单个进程所能打开的最大连接数由 FD_SETSIZE 宏定义，其大小是 32 个整数的大小（在 32 位机器上，大小是 32*32，在 64 位机器上 FD_SETSIZE 为 32*64）。消息传递时，内核需要将消息传递到用户空间，需要内核的拷贝动作。

poll:本质上 select 没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的。消息传递时，内核需要将消息传递到用户空间，需要内核的拷贝动作。

Epoll:有连接数上限，但是很大，1G 内存的机器可以打开 10 万连接。消息传递时，通过内核和用户空间共享一块内存来实现，性能较高。

3. **AIO**

Asynchronous IO:基于事件和回调机制



AIO 如何处理结果？

1.基于回调：实现 CompletionHandler 接口，调用时触发回调函数

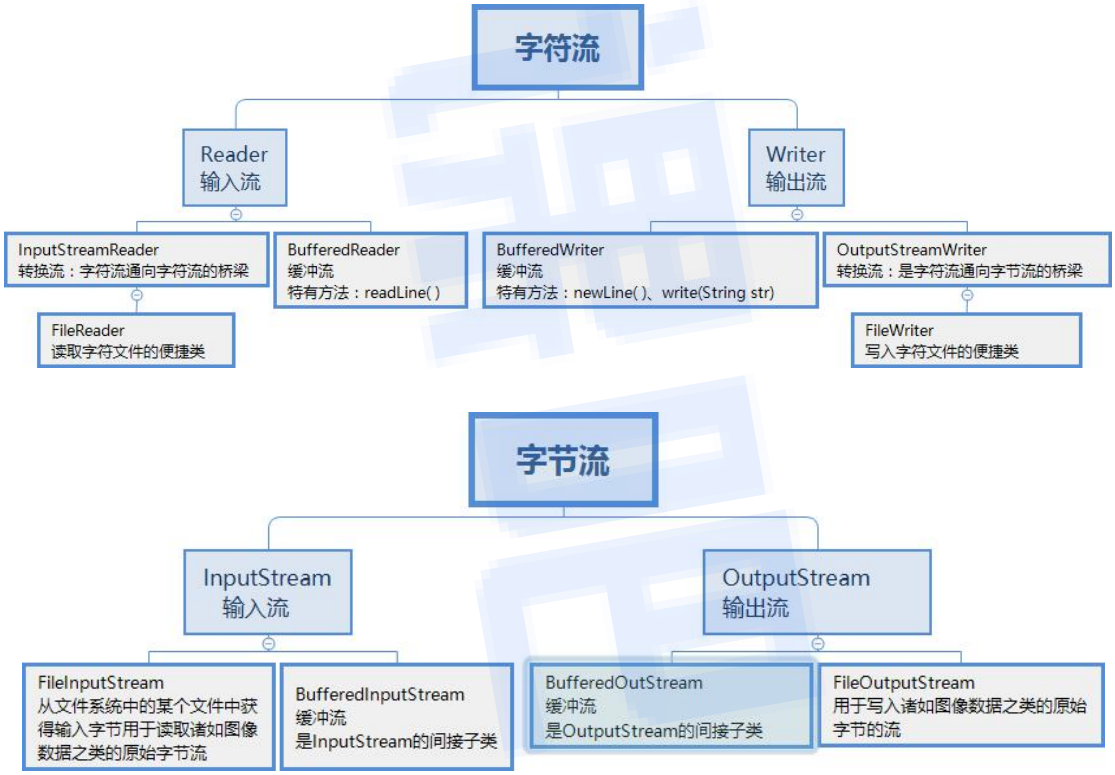
2.返回 Future:通过 isDone()查看是否准备好，通过 get()等待返回数据。

4. Java 中有几种类型的流

按照流的方向：输入流（inputStream）和输出流（outputStream）。

按照实现功能分：节点流（可以从一个特定的地方（节点）读写数据。如 FileReader）和处理流（对一个已存在的流的连接和封装，通过所封装的流的功能调用实现数据读写。如 BufferedReader。处理流的构造方法总是要带一个其他的流对象做参数。一个流对象经过其他流的多次包装，称为流的链接。）

按照处理数据的单位： 字节流和字符流。字节流继承于 InputStream 和 OutputStream ， 字符流继承于 InputStreamReader 和 OutputStreamWriter 。



5. Java 的序列化，如何实现 Java 的序列化？列举在哪些程序中见过 Java 序列化？

可序列化 Serializalbe 接口存在于 java.io 包中，构成了 Java 序列化机制的核心，它没有任何方法，它的用途是标记某对象为可序列化对象，指示编译器使用 Java 序列化机制序列化此对象。

对象的序列化主要用途：

1） 把对象的字节序列永久地保存到硬盘上，通常存放在一个文件中。

2） 在网络上传送对象的字节序列。(RPC)

6. 如何实现对象克隆

有两种方式：

1)实现 Cloneable 接口并重写 Object 类中的 clone()方法（浅克隆）

```
Person p = new Person(23, "zhang");
Person p1 = (Person) p.clone();
```

2)实现 Serializable 接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆(对象中的对象也会被克隆)