遍历器：

遍历器其实是由指针进行遍历

Iterator iterator = al.iterator()获取遍历器

iterator.next()通过指针的移动不断访问ArrayList中的元素

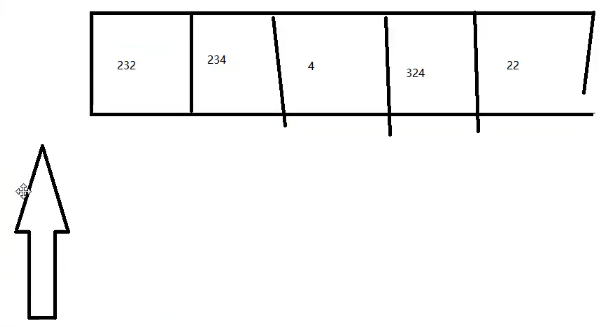
//判断指针是否移动到tail

while(iterator.hasNext()){

System.out.println(iterator.next())

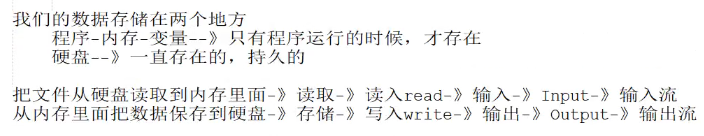
}

缺点：当指针到末尾时，访问元素为空，该遍历器不能重新将指针移动到head，只能重新获取遍历器。



Vector是线程安全的，ArrayList是非线程安全的，Vector用法类似于ArrayList

IO流



数据类型：

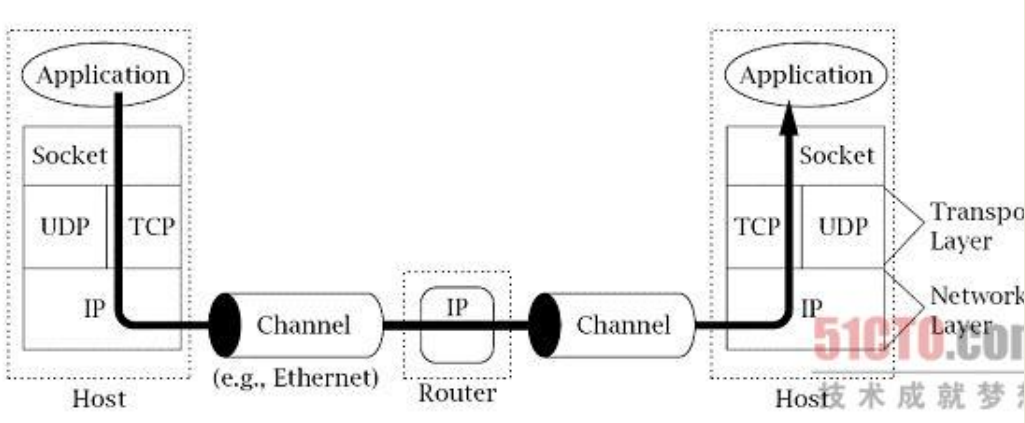
1. 文本数据
2. 二进制（字节）数据

其实所有文件都是用二进制数据存储的，不过文本数据属于我们可以打开查看浏览的数据。

数据流：

1. 字符流：只可以读取文本数据（抽象基类：Reader Writer）
2. 字节流：可以读取任意类型数据（抽象基类：InputStream OutputStream）

网络编程



一个TCP/IP网络

Socket(套接字)是一种抽象层，应用程序通过它来发送和接收数据，就像应用程序打开一个文件句柄，将数据读写到稳定的存储器上一样。

Java 为 TCP 协议提供了两个类：Socket 类和 ServerSocket 类。一个 Socket 实例代表了

TCP 连接的一端。一个 TCP 连接（TCP connection）是一条抽象的双向信道，两端分别由 IP 地址和端口号确定。在开始通信之前，要建立一个 TCP 连接，这需要先由客户端 TCP 向服 务器端 TCP 发送连接请求。ServerSocket 实例则监听 TCP 连接请求，并为每个请求创建新的 Socket 实例。也就是说，服务器端要同时处理 ServerSocket 实例和 Socket 实例，而客户端只需要使用 Socket 实例。

客户端向服务器发起连接请求后，就被动地等待服务器的响应。典型的 TCP 客户端要

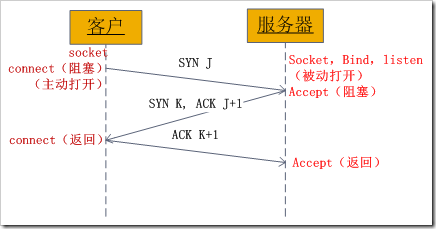
经过下面三步：

1.创建一个 Socket 实例：构造器向指定的远程主机和端口建立一个 TCP 连接。

2. 通过套接字的输入输出流（I/O streams）进行通信：一个 Socket 连接实例包括

一个 InputStream 和一个 OutputStream，它们的用法同于其他 Java 输入输出流。

3. 使用 Socket 类的 close()方法关闭连接





典型的 TCP 服务器有如下两步工作：

1.创建一个 ServerSocket 实例并指定本地端口。此套接字的功能是侦听该指定端口收

到的连接。

2.重复执行：

a. 调用 ServerSocket 的 accept()方法以获取下一个客户端连接。基于新建立的客户端连

接，创建一个 Socket 实例，并由 accept()方法返回。

b. 使用所返回的 Socket 实例的 InputStream 和 OutputStream 与客户端进行通信。

c. 通信完成后，使用 Socket 类的 close()方法关闭该客户端套接字连接。

ServerSocket 实例的唯一目的，是为新的 TCP 连接请求提供一个新的已连接的 Socket实例。当服务器端已经准备好处理客户端请求时，就调用 accept()方法。该方法将阻塞等待，

直到有向 ServerSocket 实例指定端口的新的连接请求到来。（如果新的连接请求到来时，在 服务器端套接字刚创建，而尚未调用 accept()方法，那么新的连接将排在一个队列中，在这 种情况下调用 accept()方法，将立即得到响应，即立即返回客户端套接字。）ServerSocket 类的 accept()方法将返回一个 Socket 实例，该实例已经连接到 了远程客户端的套接字，并已准备好读写数据。

bind()方法为套接字关联一个本地端口。每个 ServerSocket 实例只能与唯一一个端口相 关联。如果该实例已经关联了一个端口，或所指定的端口已经被占用，则将抛出 IOException

异常。

accept()方法为下一个传入的连接请求创建 Socket 实例，并将已成功连接的 Socket 实例

返回给服务器端套接字。如果没有连接请求等待，accept()方法将阻塞等待，直到有新的连

接请求到来或超时。

close()方法关闭套接字。调用该方法后，服务器将拒绝接受传入该套接字的客户端连接

请求。

UDP 协议提供了一种不同于 TCP 协议的端到端服务。实际上 UDP 协议只实现两个功

能：1）在 IP 协议的基础上添加了另一层地址（端口），2）对数据传输过程中可能产生的

数据错误进行了检测，并抛弃已经损坏的数据。

与 TCP 协议发送和接收字节流不同，UDP 终端交换的是一种称为数据报文的自包含（self-contained）信息。

UDP 客户端首先向被动等待联系的服务器端发送一个数据报文。一个典型的 UDP 客户

端主要执行以下三步：

1. 创建一个 DatagramSocket 实例，可以选择对本地地址和端口号进行设置。

2. 使用 DatagramSocket 类的 send() 和 receive()方法来发送和接收 DatagramPacket 实

例，进行通信。

3. 通信完成后，使用 DatagramSocket 类的 close()方法来销毁该套接字。

使用 UDP 协议的一个后果是数据报文可能丢失。在我们的回馈协议中，客户端的回馈

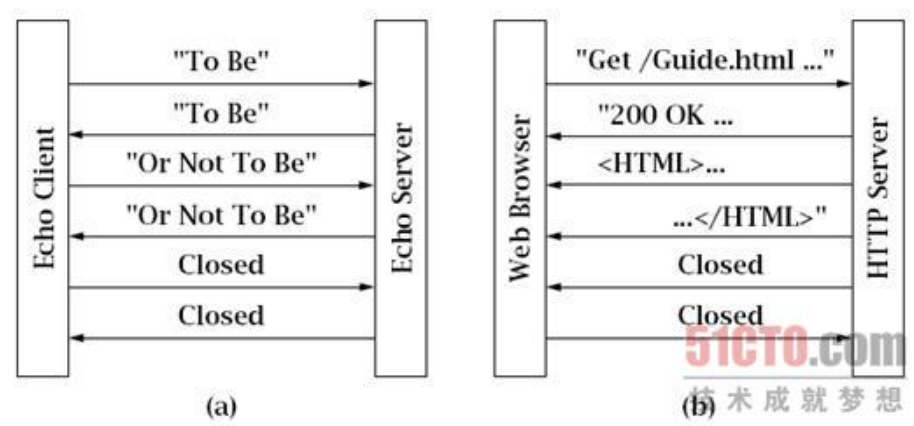
请求信息和服务器端的响应信息都有可能在网络中丢失。

在一客户一线程（thread-per-client）的服务器中，为每个连接都创建了一个新的线程来处理。服务器循环执行一些任务，在指定端口上侦听连接，反复接收客户端传入的连接请求，并为每个连接创建一个新的线程来对其进行处理。

Socket的I/O调用可能会因为多种原因而阻塞。**数据输入方法read()和receive()在没有数据可读时会阻塞。TCP套接字的write()方法在没有足够的空间缓存传输的数据时可能阻塞。ServerSocket的accept()方法和Socket的构造函数都会阻塞等待，直到连接建立。**同时，长的信息往返时间，高错误率的连接和慢速的（或已发生故障的）服务器，都可能导致需要很长的时间来建立连接。所有这些情况，只有在连接请求得到满足后这些方法才会返回。当然，调用一个已经阻塞的方法将使应用程序停止（并使运行它的线程无效）。

网络协议通常明确指定了由谁来发起"关闭"连接。在回显协议中，见图4.1（a），服务器原原本本地将客户端发送的一切数据回显回去。当客户端完成数据发送后，则调用close()方法。在服务器接收并回显了客户端调用close()方法前的所有数据后，它的read操作将返回-1，以表示客户端已经完成了数据发送。然后，服务器端套接字将调用close()方法。关闭连接是协议中的关键部分，因为如果没有关闭，服务器将不知道客户端什么时候发送完了要回显的字符。对于HTTP协议，见图4.1（b），是由服务器端发起的关闭连接。客户端先向服务器发送一个请求（"get"），然后服务器发送回一个响应头信息（通常由"200 OK"开始），后面跟的是所请求的文件。由于客户端不知道文件的大小，因此服务器必须通过关闭套接字来指示文件的结束。

调用Socket的close()方法将同时终止两个方向（输入和输出）的数据流。



**NIO**

在使用线程的扩展性方面还涉及一些更加难以把握的挑战。其中一个挑战就是程序员几乎不能对什么时候哪个线程将获得服务进行控制。你可以设置一个线程实例的优先级（priority）（高优先级的线程相对于低优先级的线程有优先权），但是这个优先级只是一种"建议"--下一个选择执行的线程完全 取决于具体实现。因此，如果程序员想要保证某些连接优先获得服务，或想要指定一定的服务顺序，线程可能就很难做到。

NIO 中将介绍的另一个主要特性是 Buffer 类。就像 selector 和 channel 为一次处理多个 客户端的系统开销提供了更高级的控制和可预测性，Buffer 则提供了比 Stream 抽象更高效和可预测的 I/O。 Stream 抽象好的方面是隐藏了底层缓冲区的有限性，提供了一

个能够容纳任意长度数据的容器的假象。坏的方面是要实现这样一个假象，要么会产生大量的内存开销，要么会引入大量的上下文切换，甚至可能两者都有。在使用线程时，这些开销都隐藏在了具体实现中，因此也失去了对其的可控性和可预测性。这种方法使编写程序变得容易，但要调整它们的性能则变得更困难。不幸的是，如果要使用 Java 的 Socket 抽象，流就是唯一的选择。

Buffer 抽象代表了一个**有限容量（finite-capacity）的数据容器--其本质是一个数组，由指针指示了在哪存放数据和从哪读取数据。**使用 Buffer 有两个主要好处。第一，与读写缓冲区数据相关联的系统开销暴露给了程序员。例如，如果想要向缓冲区存入数据，但又没有足够的空间时，就必须采取一些措施来获得空间（即，移出一些数据，或移开已经在那个位置的数据来获得空间，或者创建一个新的实例）。

**Channel** 使用的不是流，而是缓冲区来发送或读取数据。Buffer 类或其任何子类的实例都可以看作是一个定长的 Java 基本数据类型元素序列。与流不同，缓冲区有固定的、有限的容量，并由内部（但可以被访问）状态记录了有多少数据放入或取出，就像是有限容量的队列一样。NIO 的强大功能部分来自于 channel 的非阻塞特性。

**public class** TCPEchoClientNonblocking {  
 **public static void** main(String[] args) **throws** Exception{  
 **byte**[] data = **"hello world"**.getBytes();  
 SocketChannel clientChanel = SocketChannel.*open*();  
 *//创建一个信道，设置为非阻塞模式* clientChanel.configureBlocking(**false**);  
 *//向服务器发起连接* **if** (!clientChanel.connect(**new** InetSocketAddress(**"127.0.0.1"**,33))){  
 *//不断地轮询连接状态，直到完成连接* **while** (!clientChanel.finishConnect()){  
 *//在等待连接的时间里，可以执行其他任务，以充分发挥非阻塞IO的异步特性，这里是不断打印* System.***out***.println(**"."**);  
 }  
 }  
  
 ByteBuffer writeBuf = ByteBuffer.*wrap*(data);  
 ByteBuffer readBuf = ByteBuffer.*allocate*(data.**length**);  
  
 **int** totalBytesRcvd = 0;  
 **int** bytesRcvd;  
 *//循环执行，直到接受到的字节数与发送的字符串的字节数相等* **while** (totalBytesRcvd < data.**length**){  
 **if** (writeBuf.hasRemaining()){  
 clientChanel.write(writeBuf);  
 }  
  
 **if** ((bytesRcvd = clientChanel.read(readBuf)) == -1){  
 **throw new** SocketException(**"Connection closed prematurely"**);  
 }  
  
 totalBytesRcvd += bytesRcvd;  
 *//最容易出现阻塞的是read(),在当没有数据可读时则会阻塞，因为使用chennel的非阻塞，则可以执行其他任务，  
 //体现非阻塞IO的异步特性* System.***out***.println(**"."**);  
 }  
 clientChanel.close();  
 }  
}

**Selector**

每个选择器都有一组与之关联的信道，选择器对这些信道上"感兴趣的"I/O 操作进行监听。Selector 与 Channel 之间的关联由一个 SelectionKey 实例表示。（注意，一个信道可以注册多个Selector实例，因此可以有多个关联的SelectionKey实例）SelectionKey维护了一个信道上感兴趣的操作类型信息，并将这些信息存放在一个 int 型的位图（bitmap）中，该 int 型数据的每一位都有相应的含义。

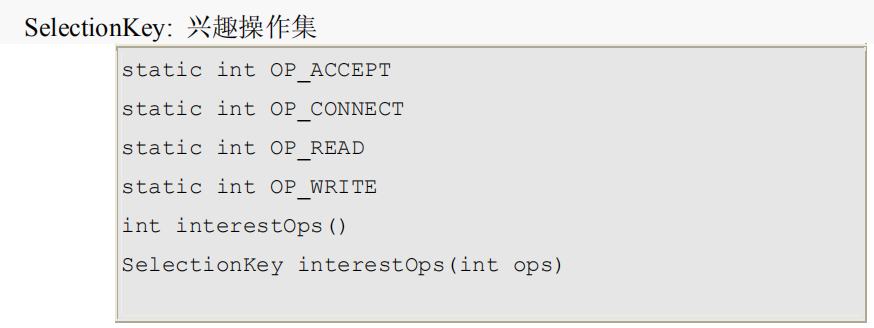
通过对 OP\_ ACCEPT，OP\_CONNECT，OP\_READ 以及 OP\_WRITE 中适当的常量进行

按位 OR，我们可以构造一个位向量来指定一组操作。例如，一个包含读和写的操作集可由

表达式（OP\_READ | OP\_WRITE）来指定。不带参数的 interestOps()方法将返回一个 int 型

位图，该位图中设置为 1 的每一位都指示了信道上需要监听的一种操作。另一个方法以一

个位图为参数，指示了应该监听信道上的哪些操作。重点提示：任何对 key（信道）所关联的兴趣操作集的改变，都只在下次调用了 select()方法后才会生效。



调用信道的 register()方法可以将一个选择器注册到该信道。在注册过程中，通过存储在int 型数据中的位图来指定该信道上的初始兴趣操作集（见上文的"SelectionKey：兴趣操作集"）。register()方法将返回一个代表了信道和给定选择器之间的关联的 SelectionKey 实例。

选择之后，我们需要知道哪些信道准备好了特定的 I/O 操作。每个选择器都维护了一个已选键集（selected-key set），与这些键关联的信道都有即将发生的特定 I/O 操作。通过调用 selectedKeys()方法可以访问已选键集，该方法返回一组 SelectionKey。我们可以在这组键上进行迭代，分别处理等待在每个键关联的信道上的 I/O 操作。

