1、super 的作用

在 Java 中 super 指代父类对象（直接父类），也就是说，super 相当于是一个直接 new 出来的父类对象，所以可以通过它来调用父类的那些非 private 修饰的变量、方法（对于我们普通 new 出来的对象来说，也就只能访问那些非 private 的成员变量、方法了，这里的访问是指通过“对象名.变量名或方法名”的形式）。所以，super 这个对象也就是一个普通对象，同样遵循访问控制修饰符的准则。

然而，对于子类来说，子类通过继承就直接拥有了父类的非 private 变量、方法，也就可以在子类中直接使用，再加一个 super 来修饰，岂不是显得有点多余了？正常情况下来说，是有点多余了（但是可以明确提示我们这是调用的父类变量或方法），但 super 关键字主要是用在以下两种情况中：

（1）发生了重写的情况

重写也分为两种情况，一个是重写了父类的方法；一个是重写了父类的成员变量；

重写父类方法的情况：

```
public class A {

    String name = "lly";

    protected void getName(){

        System.out.println("父类 getName->"+ name);

    }

}

public class B extends A {

    String nameB = "llyB";

    @Override

        protected void getName() {

        System.out.println("子类 getName->"+nameB);
```

```
        super.getName();

    }

    public static void main(String[] args) {

        B b = new B();

        b.getName();

    }

}
```

打印如下：

```
子类 getName->llyB

父类 getName->lly
```

在子类 B 中，我们重写了父类的 getName 方法，如果在重写的 getName 方法中我们去调用了父类的相同方法，必须要通过 super 关键字显示的指明出来。

如果不明确出来，按照子类优先的原则，相当于还是再调用重写的 getName() 方法，此时就形成了死循环，执行后会报 java.lang.StackOverflowError 异常。

重写父类变量的情况：

我们将 B 类简单改造一下：

```
public class B extends A {

    String name = "llyB";

    @Override

        protected void getName() {

        name = super.name;

        System.out.println("子类 getName->"+name);
```

```
    }

    public static void main(String[] args) {

        B b = new B();

        b.getName();

    }

}
```

此时子类 B 中有一个和父类一样的字段（也可以说成父类字段被隐藏了），为了获得父类的这个字段我们就必须加上 super，如果没有加，直接写成 name = name;不会报错，只是会警告，表示此条语句没有任何意义，因为此时都是访问的子类 B 里面的那么字段。

我们通过 super 是不能访问父类 private 修饰的变量和方法的，因为这个只属于父类的内部成员，一个对象是不能访问它的 private 成员的。

（2）在子类的构造方法中

编译器会自动在子类构造函数的第一句加上 super(); 来调用父类的无参构造器；此时可以省略不写。如果想写上的话必须在子类构造函数的第一句，可以通过 super 来调用父类其他重载的构造方法，只要相应的把参数传过去就好。

因此，super 的作用主要在下面三种情况下：

- 调用父类被子类重写的方法；
- 调用父类被子类重定义的字段（被隐藏的成员变量）；
- 调用父类的构造方法；

其他情况，由于子类自动继承了父类相应属性方法，关键字 super 可以不显示写出来。

2、关于构造方法

如果一个类中没有写任何的构造方法，JVM 会生成一个默认的无参构造方法。在继承关系中，由于在子类的构造方法中，第一条语句默认为调用父类的无参构造方法（即默认为 super()，一般这句话省略了）。所以当在父类中定义了有参构造函数，都是没有定义无参构造函数时，IDE 会强制要求我们定义一个相

同参数类型的构造器。这也是我们在 Android 中自定义组件去继承其他 View 是经常被要求定义几个构造函数的原因。

以下子类 B 的情形是错误不能通过编译的：

```
public class A {

    public A(String s){

    }

}

public class B extends A {

    //编译错误，JVM 默认给 B 加了一个无参构造方法，而在这个方法中默认调用了 super()，但是父类中并不存在该构造方法

    String name = "llyB";

}

public class B extends A {

    //同样编译错误，相同的道理，虽然我们在子类中自己定义了一个构造方法，但是在这个构造方法中还是默认调用了 super()，但是父类中并不存在该构造方法

    String name = "llyB";

    public B(String s){

    }

}
```

此时就需要显示的去调用父类构造方法了，如下：

```
public class B extends A {

    //正确编译

    String name = "llyB";
```

```
    public B(String s){

        super(s);

    }

}
```

所以，只要记住，在子类的构造方法中，只要里面没有显示的通过 super 去调用父类相应的构造方法，默认都是调用 super()，即无参构造方法，因此要确保父类有相应的构造方法。

3、transient 关键字用法

当用 transient 关键字修饰一个变量时，这个变量将不会参与序列化过程。也就是说它不会在网络操作时被传输，也不会再本地被存储下来，这对于保护一些敏感字段（如密码等...）非常有帮助。

当我们一个对象实现了 Serializable 接口，这个对象的所有字段和方法就可以被自动序列化。当我们持久化对象时，可能有一个一些特殊字段我们不想让它随着网络传输过去，或者在本地序列化缓存起来，这时我们就可以在这些字段前加上 transient 关键字修饰，被 transient 修饰变量的值不包括在序列化的表示中，也就不会被保存下来。这个字段的生命周期仅存在调用者的内存中。

如下一个例子：

```
public class UserBean implements Serializable{

    private static final long serialVersionUID = 856780694939330811L;

    private String userName;

    private transient String password;

    //此字段不需要被序列化

    public String getUserName() {

        return userName;

    }
```

```
    public void setUserName(String userName) {

        this.userName = userName;

    }

    public String getPassword() {

        return password;

    }

    public void setPassword(String password) {

        this.password = password;

    }

}
```

测试类：

```
public class Test {

    public static void main(String[] args) {

        UserBean bean = new UserBean();

        bean.setUserName("lly");

        bean.setPassword("123");

        System.out.println("序列化前--->userName:"+bean.getUserName()+", password:"+
bean.getPassword());

        //下面序列化到本地

        ObjectOutputStream oos = null;

        try {

            oos = new ObjectOutputStream(new FileOutputStream("e:/userbean.txt"));
```

```
            oos.writeObject(bean);
            //将对象序列化缓存到本地
            oos.flush();
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally{
            if(oos != null){
                try {
                    oos.close();
                }
                catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        //下面从本地反序列化缓存出来
        try {
```

```
            ObjectInputStream ois = new ObjectInputStream(new FileInputStream("e:/u
serbean.txt"));

            bean = (UserBean) ois.readObject();

            ois.close();

            System.out.println("反序列化后获取出的数据--->userName:"+bean.getUserName
()+", password:"+bean.getPassword());

        }

        catch (FileNotFoundException e) {

            e.printStackTrace();

        }

        catch (IOException e) {

            e.printStackTrace();

        }

        catch (ClassNotFoundException e) {

            e.printStackTrace();

        }

    }

}
```

打印结果如下：

```
序列化前--->userName:lly, password:123

反序列化后获取出的数据--->userName:lly, password:null
```

看到，password 反序列化后的值为 null，说明它没有被保存到本地，因为我们给它加上了 transient 修饰。

注意：

- 从上面可以看到，被 transient 修饰后，反序列化后不能获取到值；
- transient 只能修饰变量，不能修饰方法，修饰我们自定义的对象变量时，这个对象一定要实现 Serializable 接口；
- static 变量不能被序列化，即使它被 transient 修饰。因为 static 修饰的变量是属于类的，而我们序列化是去序列化对象的。

上面的例子中如果给 userName 加上 static 修饰，反序列化后依然能够获取到值，但是这个时候的值是 JVM 内存中对应的 static 的值，因为 static 修饰后，它属于类不属于对象，存放在一块单独的区域，直接通过对象也是可以获取到这个值的。上面的第三点依然成立。

4、下面哪些类可以被继承

下面哪些类可以被继承？ Java.lang.Thread、java.lang.Number、java.lang.Double、java.lang.Math、 java.lang.ClassLoader

- A、Thread
- B、Number
- C、Double
- D、Math
- E、ClassLoader

正确答案：ABE

1、对于 Thread，我们可以继承它来创建线程；
2、Byte、Short、Integer、Long、Float、Double 几个数字类型都是继承自 Number；
3、Byte、Short、Integer、Long、Float、Double、Boolean、Character 几个基本类型的包装类，以及 String、Math 它们都是被定义为 public final class，因此这几个都不能被继承；
4、我们可以自定义类加载器来实现加载功能，因此 ClassLoader 是可以被继承的。
5、for 和 foreach 遍历的比较
针对 for 和 foreach 两种循环，我用 10 万、100 万、1000 万级别大小的 List 集合数据进行了测试一下，发现整体来说 foreach 的执行时间和普通 for 循环的时间差别不大。对于这两个的比较，我网上查了一下，有说 foreach 效率高一点的，有说 for 效率高一点的。

网上说 for 效率高的，主要是在针对遍历集合类的数据时，for 表现要稍微好一点，因为 foreach 内部它使用的是 Iterator 迭代器的执行方式。因此下面分两种结构来测试：
先来测试对数组遍历的时间快慢，数组里面保存 1000 万的随机数。如下：

```
List<Integer> list = new ArrayList<Integer>();

        Integer[] list = new Integer[10000000];

        for(int i = 0; i < 10000000; i++){//先生成1000万随机数

            list[i] = Math.round(100000);//随机数只是咋100000以内

        }

        long size = list.length;

        long start = System.currentTimeMillis();

        int a;

        for(int i = 0; i < size; i++){

            a =list[i];

        }

        long end = System.currentTimeMillis();

        System.out.println("耗时--->"+(end-start));
```

经过多次运行，发现平均耗时 7ms 左右。将上面的 for 循环改成 foreach 形式，如下：

```
for (Integer i : list){

    a = i;

}
```

经过多次运行后，发现平均耗时 5ms 左右。
再来测试遍历集合 List 的情况：

```
List<Integer> list = new ArrayList<Integer>();

        for(int i = 0; i < 10000000; i++){

            list.add(Math.round(100000));

        }

        long size = list.size();

        long start = System.currentTimeMillis();

        int a;

        for(int i = 0; i < size; i++){

            a =list.get(i);

        }

        long end = System.currentTimeMillis();

        System.out.println("耗时--->"+(end-start));
```

多次运行后，平均运行时间为 21ms 左右，将上面的 for 循环改成 foreach 形式，如下：

```
for (Integer i : list){

    a = i;

}
```

多次运行后，平均运行时间为 24ms 左右。

经过上面一些简单的实验，我们发现，两种方式在遍历数组的时候时间普遍都较快，在遍历集合的时候耗时会更久一些。在遍历数组时，foreach 的表现要稍微好一点，在遍历集合的时候，for 的表现要好一点。但是不管哪种情况，for 和 foreach 这两种遍历方式时间都相差不大。因此对于这两者的比较在时间效率来说应该相差不会很大（上面没有测试复杂数据的情况，以及其他集合结果的情况，可能不准确）。主要是在对于两者的应用场景上的选择：

- 普通 for 循环可以给定下标，因此当我们需要这个信息时，我们可以选用普通 for 循环来操作遍历；
- foreach 在代码结构上更加清晰、简单；
- foreach 在遍历的时候会锁定集合中的对象，期间不能修改，而 for 中可以修改集合中的元素。

6、Java IO 与 NIO

Java NIO（Java non-blocking IO）是 JDK1.4 以后推出的，相对于原来的 IO 来说，Java NIO 是一种非阻塞的 IO 方式，它为所有的基本类型提供了缓存支持。

一般来说，I/O 操作包括：对硬盘的读写、对 socket 的读写以及外设的读写。

阻塞和非阻塞：

- 阻塞：当某个事件或者任务在执行过程中，它发出一个请求操作，但是由于该请求操作需要的条件不满足，那么就会一直在那等待，直至条件满足；
- 非阻塞：当某个事件或者任务在执行过程中，它发出一个请求操作，如果该请求操作需要的条件不满足，会立即返回一个标志信息告知条件不满足，不会一直在那等待。

这就是阻塞和非阻塞的区别。也就是说阻塞和非阻塞的区别关键在于当发出请求一个操作时，如果条件不满足，是会一直等待还是返回一个标志信息。

阻塞 IO 和非阻塞 IO：

当用户线程发起一个 IO 请求操作（本文以读请求操作为例），内核会去查看要读取的数据是否就绪，对于阻塞 IO 来说，如果数据没有就绪，则会一直在那等待，直到数据就绪；

对于非阻塞 IO 来说，如果数据没有就绪，则会返回一个标志信息告知用户线程当前要读的数据没有就绪。当数据就绪之后，便将数据拷贝到用户线程，这样才完成了一个完整的 IO 读请求操作。

也就是说一个完整的 IO 读请求操作包括两个阶段：

- 查看数据是否就绪；

- 进行数据拷贝（内核将数据拷贝到用户线程）。

那么阻塞（blocking IO）和非阻塞（non-blocking IO）的区别就在于第一个阶段，如果数据没有就绪，在查看数据是否就绪的过程中是一直等待，还是直接返回一个标志信息。

Java 中传统的 IO 都是阻塞 IO，比如通过 socket 来读数据，调用 read()方法之后，如果数据没有就绪，当前线程就会一直阻塞在 read 方法调用那里，直到有数据才返回；而如果是非阻塞 IO 的话，当数据没有就绪，read()方法应该返回一个标志信息，告知当前线程数据没有就绪，而不是一直在那里等待。

同步 IO 和异步 IO：

从字面的意思可以看出：同步 IO 即 如果一个线程请求进行 IO 操作，在 IO 操作完成之前，该线程会被阻塞；

而异步 IO 为 如果一个线程请求进行 IO 操作，IO 操作不会导致请求线程被阻塞。

事实上，同步 IO 和异步 IO 模型是针对用户线程和内核的交互来说的：

对于同步 IO：当用户发出 IO 请求操作之后，如果数据没有就绪，需要通过用户线程或者内核不断地去轮询数据是否就绪，当数据就绪时，再将数据从内核拷贝到用户线程；

而异步 IO：只有 IO 请求操作的发出是由用户线程来进行的，IO 操作的两个阶段都是由内核自动完成，然后发送通知告知用户线程 IO 操作已经完成。也就是说在异步 IO 中，不会对用户线程产生任何阻塞。

这是同步 IO 和异步 IO 关键区别所在，同步 IO 和异步 IO 的关键区别反映在数据拷贝阶段是由用户线程完成还是内核完成。所以说异步 IO 必须要有操作系统的底层支持。（即同步 IO 是用户线程不断的轮询、有数据之后进行拷贝，而异步 IO 是内核完成这两个步骤，与用户线程无关。）

阻塞 IO 和非阻塞 IO 是反映在当用户请求 IO 操作时，如果数据没有就绪，是用户线程一直等待数据就绪，还是会收到一个标志信息这一点上面的。也就是说，阻塞 IO 和非阻塞 IO 是反映在 IO 操作的第一个阶段，在查看数据是否就绪时是如何处理的。

注意同步 IO 和异步 IO 与阻塞 IO 和非阻塞 IO 是不同的两组概念，同步 IO 和异步 IO 考虑的是由哪个线程（用户线程 or 内核线程）来完成 IO 的处理，而阻塞 IO 和非阻塞 IO，针对的是 IO 操作中的第一个阶段的处理方式，是一直等待还是直接返回状态信息。

Java NIO：

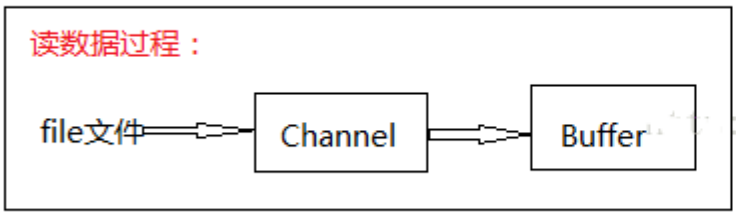
Java NIO 中比较核心的三个概念是：

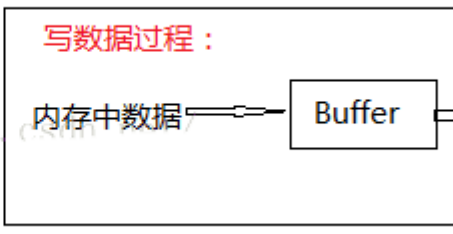
- 通道（Channel）
- 缓冲区（Buffer）
- 选择器（Selector）

（1）通道（Channel）：
通道和传统 IO 中的 Stream 类似，但传统 IO 中的 Stream 是单向的，比如 InputStream 只能进行读（read）操作，OutputStream 只能进行写（write）操作，而 NIO 中的通道是双向的，既可以进行读也可以进行写操作。常用的有以下几种通道：

- FileChannel --从文件读或者向文件写入数据
- SocketChanel --以 TCP 来向网络连接的两端读写数据
- ServerSocketChannel --服务器端通过 ServerSocketChanel 能够监听客户端发起的 TCP 连接，并为每个 TCP 连接创建一个新的 SocketChannel 来进行数据读写
- DatagramChannel --以 UDP 协议来向网络连接的两端读写数据

（2）缓冲区（Buffer）：
Buffer 实际上是一个容器，是一个连续数组。Channel 提供从文件、网络读取数据的渠道，使用 Java NIO 时，所有数据处理（读取或写入）必须经由 Buffer。即读入数据时，先通过通道 Channel 获取数据渠道，然后将数据写入到缓冲区 Buffer 中，往外写数据时，先把内存中的数据放到 Buffer 中，然后再从 Buffer 送往通道中。如下图所示：





NIO 为 Java 的所有基本类型都提供了缓存对象，如 ByteBuffer、IntBuffer。。。

一个传统 IO 读写数据的例子：

```
public static String readFromStream(InputStream is) throws IOException{

        ByteArrayOutputStream baos = new ByteArrayOutputStream();

        byte[] buffer = new byte[1024];

        int len = 0;

        while((len = is.read(buffer))!=-1){

            baos.write(buffer, 0, len);

        }

        is.close();

        String result = baos.toString();

        baos.close();

        return result;

    }
```

传统 IO 在读取数据时，直接从 InputStream 中读入到 byte[]数组中，如 is.read(buffer)，在写数据时，直接将内存数据写到输出流中,如 baos.write(buffer,0,len);

使用 NIO 读取数据：按照上图，读取数据时，从通道中将数据读入到 Buffer 中，写数据时，将内存数据先写到 Buffer，再送入通道。

NIO 读取数据：

```
FileInputStream in = new FileInputStream("e:\lly.txt");

FileChannel fileChannel = in.getChannel();

//创建一个ByteBuffer 缓冲区

ByteBuffer buffer = ByteBuffer.allocate(10);

//从通道读入数据到缓冲区
```

```
fileChannel.read(buffer);

//重设此缓冲区，将限制设置为当前位置，然后将当前位置设置为0

buffer.flip();

while(buffer.hasRemaining()){

    byte b = buffer.get();

    //从缓冲区中取数据到内存中

    System.out.print(((char)b));

}

in.close();
```

NIO 写数据：

```
     //模拟数据

byte[] data = { 83, 111, 109, 101, 32,

                98, 121, 116, 101, 115, 46};

FileOutputStream out = new FileOutputStream("e:\lly.txt");

FileChannel fileChannel = out.getChannel();

//创建一个ByteBuffer 缓冲区

ByteBuffer buffer = ByteBuffer.allocate(20);

//先将数据写入到Buffer 中

for (int i = 0; i < data.length; i++){

    buffer.put(data[i]);

}

//重设此缓冲区，将限制设置为当前位置，然后将当前位置设置为0

buffer.flip();
```

```
//把buffer 中的数据送入到通道中

fileChannel.write(buffer);

out.close();
```

可以看到，Java NIO 在数据处理方面需要通过 Buffer 来缓存，与传统 IO 处理方式相比，NIO 相当于是数据块处理，传统 IO 是一个一个字节处理。其实，在传统 IO 后面出现了 BufferInputStream、BufferOutputStream 这种也是缓存数据块的处理方式。

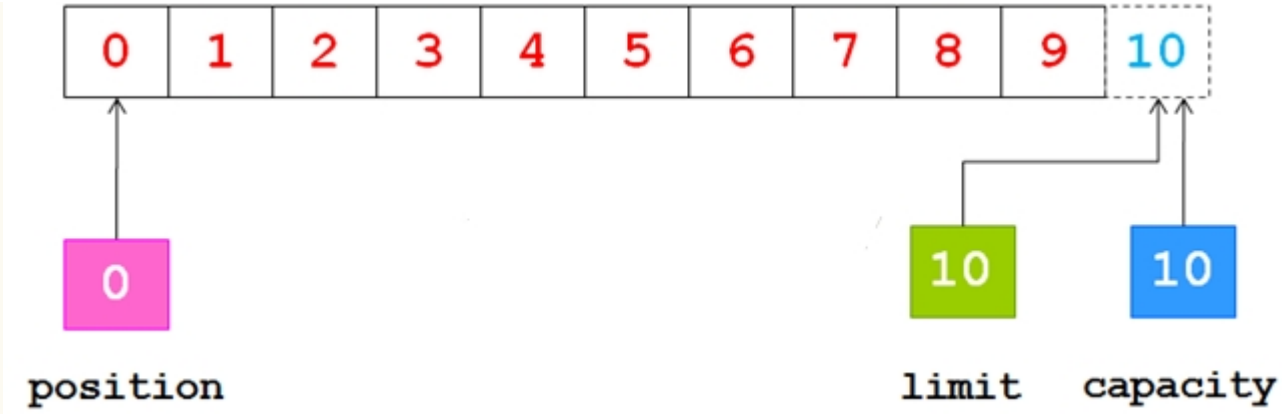
同时，我们看到在读或写时都调用了 Buffer 的 buffer.flip();这个方法非常重要，它是用来设置缓冲对象的一些状态信息，在读 get()和写 put()之前，都需要调用 buffer.flip();来设置状态。

在第一篇中，我们介绍了 NIO 中的两个核心对象：缓冲区和通道，在谈到缓冲区时，我们说缓冲区对象本质上是一个数组，但它其实是一个特殊的数组，缓冲区对象内置了一些机制，能够跟踪和记录缓冲区的状态变化情况，如果我们使用 get()方法从缓冲区获取数据或者使用 put()方法把数据写入缓冲区，都会引起缓冲区状态的变化。本文为 NIO 使用及原理分析的第二篇，将会分析 NIO 中的 Buffer 对象。

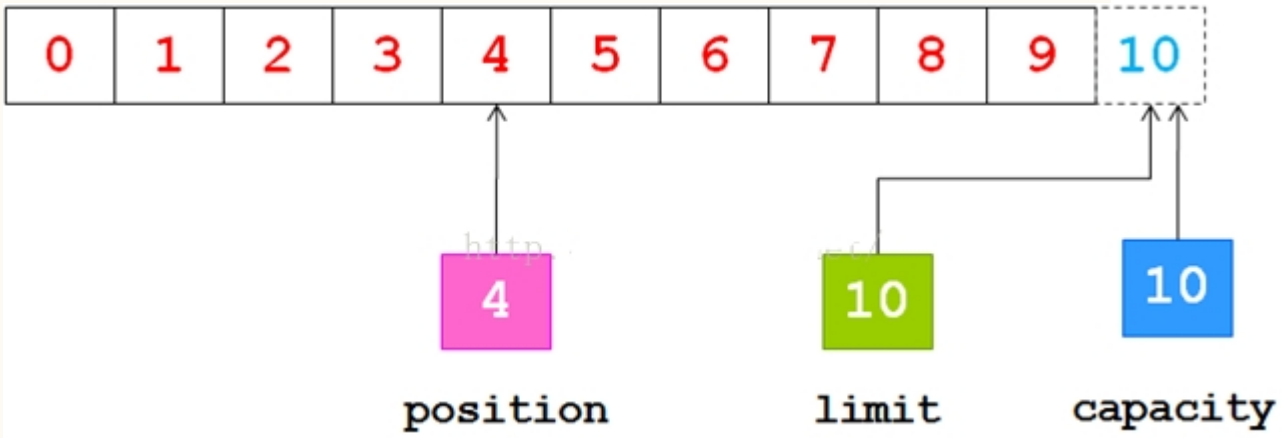
在缓冲区中，最重要的属性有下面三个，它们一起合作完成对缓冲区内部状态的变化跟踪：

- position：指定了下一个将要被写入或者读取的元素索引，它的值由 get()/put()方法自动更新，在新创建一个 Buffer 对象时，position 被初始化为 0。
- limit：指定还有多少数据需要取出(在从缓冲区写入通道时)，或者还有多少空间可以放入数据(在从通道读入缓冲区时)。
- capacity：指定了可以存储在缓冲区中的最大数据容量，实际上，它指定了底层数组的大小，或者至少是指定了准许我们使用的底层数组的容量。

以上四个属性值之间有一些相对大小的关系：0 <= position <= limit <= capacity。如果我们创建一个新的容量大小为 10 的 ByteBuffer 对象，在初始化的时候，position 设置为 0，limit 和 capacity 被设置为 10，在以后使用 ByteBuffer 对象过程中，capacity 的值不会再发生变化，而其它两个个将会随着使用而变化。四个属性值分别如图所示：



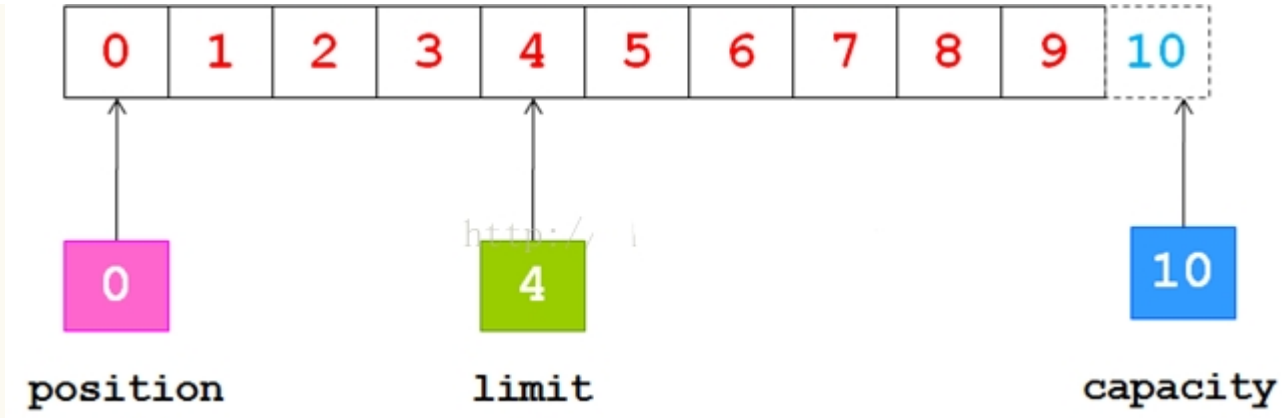
现在我们可以从通道中读取一些数据到缓冲区中，注意从通道读取数据，相当于往缓冲区中写入数据。如果读取 4 个自己的数据，则此时 position 的值为 4，即下一个将要被写入的字节索引为 4，而 limit 仍然是 10，如下图所示：



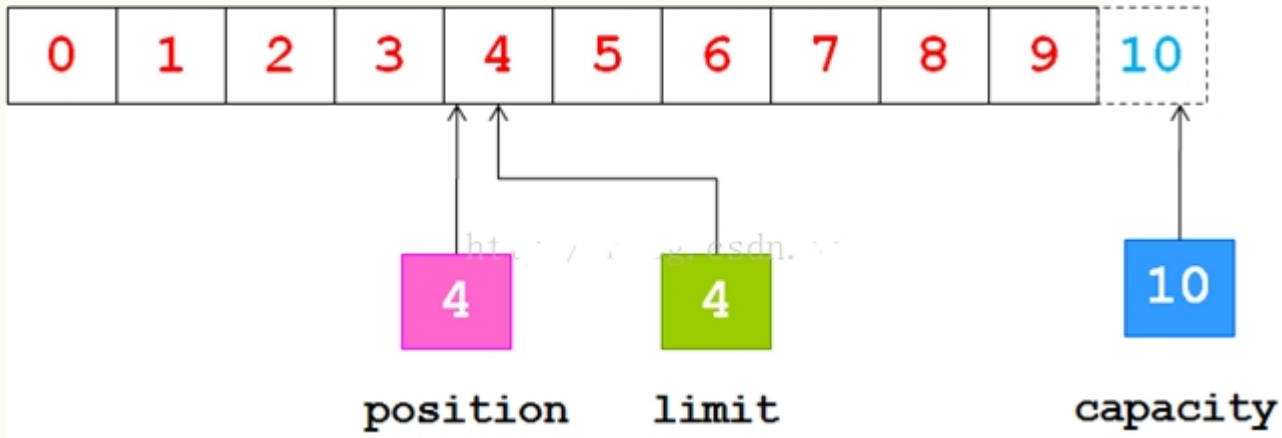
下一步把读取的数据写入到输出通道中，相当于从缓冲区中读取数据，在此之前，必须调用 flip()方法，该方法将会完成两件事情：

1. 把 limit 设置为当前的 position 值
2. 把 position 设置为 0

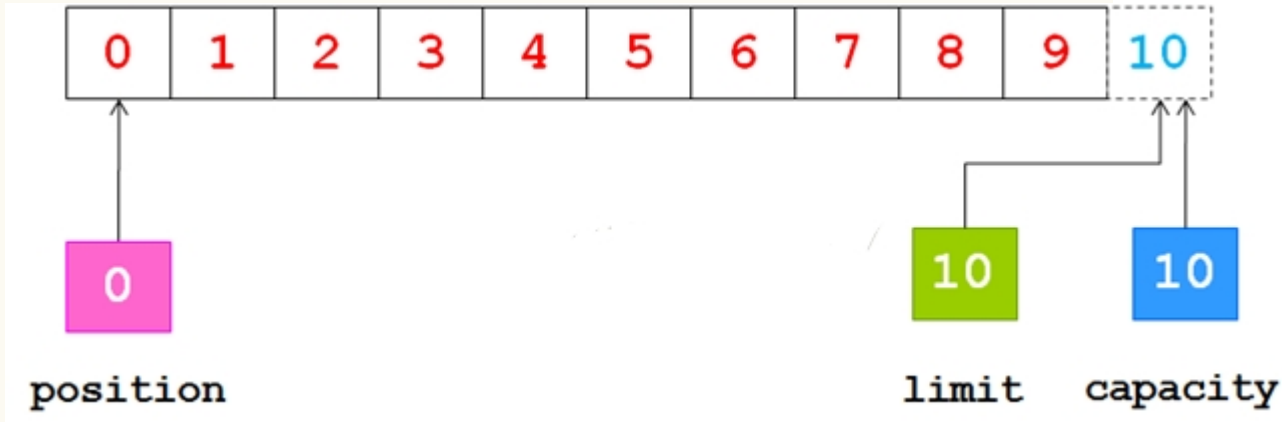
由于 position 被设置为 0，所以可以保证在下一步输出时读取到的是缓冲区中的第一个字节，而 limit 被设置为当前的 position，可以保证读取的数据正好是之前写入到缓冲区中的数据，如下图所示：



现在调用 get()方法从缓冲区中读取数据写入到输出通道，这会导致 position 的增加而 limit 保持不变，但 position 不会超过 limit 的值，所以在读取我们之前写入到缓冲区中的 4 个自己之后，position 和 limit 的值都为 4，如下图所示：



在从缓冲区中读取数据完毕后，limit 的值仍然保持在我们调用 flip()方法时的值，调用 clear()方法能够把所有的状态变化设置为初始化时的值，如下图所示：



（3）选择器（Selector）

Selector 类是 NIO 的核心类，Selector 能够检测多个注册的通道上是否有事件发生，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响应处理。

这样一来，只是用一个单线程就可以管理多个通道，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。如下如所示：

要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select()方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

```
package cn.nio;

import java.io.IOException;

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;

/**

* NIO 服务端

* @author 小路

*/

public class NIOServer {

    //通道管理器

    private Selector selector;

    /**

* 获得一个ServerSocket 通道，并对该通道做一些初始化的工作
```

```
* @param port 绑定的端口号
* @throws IOException
*/
    public void initServer(int port) throws IOException {
        // 获得一个ServerSocket 通道
        ServerSocketChannel serverChannel = ServerSocketChannel.open();
        // 设置通道为非阻塞
        serverChannel.configureBlocking(false);
        // 将该通道对应的ServerSocket 绑定到port 端口
        serverChannel.socket().bind(new InetSocketAddress(port));
        // 获得一个通道管理器
        this.selector = Selector.open();
        //将通道管理器和该通道绑定，并为该通道注册SelectionKey.OP_ACCEPT 事件,注册该事件后，
        //当该事件到达时，selector.select()会返回，如果该事件没到达selector.select()会一直阻塞。
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    }
    /**
* 采用轮询的方式监听selector 上是否有需要处理的事件，如果有，则进行处理
* @throws IOException
*/
    @SuppressWarnings("unchecked")
    public void listen() throws IOException {
```

```
        System.out.println("服务端启动成功！");
        // 轮询访问selector
        while (true) {
            //当注册的事件到达时，方法返回；否则,该方法会一直阻塞
            selector.select();
            // 获得selector中选中的项的迭代器，选中的项为注册的事件
            Iterator ite = this.selector.selectedKeys().iterator();
            while (ite.hasNext()) {
                SelectionKey key = (SelectionKey) ite.next();
                // 删除已选的key,以防重复处理
                ite.remove();
                // 客户端请求连接事件
                if (key.isAcceptable()) {
                    ServerSocketChannel server = (ServerSocketChannel) key
                    .channel();
                    // 获得和客户端连接的通道
                    SocketChannel channel = server.accept();
                    // 设置成非阻塞
                    channel.configureBlocking(false);
                    //在这里可以给客户端发送信息哦
                    channel.write(ByteBuffer.wrap(new String("向客户端发送了一条信息"
).getBytes()));
                    //在和客户端连接成功之后，为了可以接收到客户端的信息，需要给通道设置读
的权限。
```

```
                    channel.register(this.selector, SelectionKey.OP_READ);
                    // 获得了可读的事件
                } else if (key.isReadable()) {
                    read(key);
                }
            }
        }
    }
    /**
* 处理读取客户端发来的信息 的事件
* @param key
* @throws IOException
*/
    public void read(SelectionKey key) throws IOException{
        // 服务器可读取消息:得到事件发生的Socket 通道
        SocketChannel channel = (SocketChannel) key.channel();
        // 创建读取的缓冲区
        ByteBuffer buffer = ByteBuffer.allocate(10);
        channel.read(buffer);
        byte[] data = buffer.array();
        String msg = new String(data).trim();
        System.out.println("服务端收到信息: "+msg);
        ByteBuffer outBuffer = ByteBuffer.wrap(msg.getBytes());
```

```
            channel.write(outBuffer);
            // 将消息回送给客户端
        }
    /**
* 启动服务端测试
* @throws IOException
*/
    public static void main(String[] args) throws IOException {
        NIOServer server = new NIOServer();
        server.initServer(8000);
        server.listen();
    }
}
package cn.nio;
import java.io.IOException;
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;
/**
* NIO 客户端
```

```
* @author 小路
*/
public class NIOClient {
    //通道管理器
    private Selector selector;
    /**
* 获得一个Socket 通道，并对该通道做一些初始化的工作
* @param ip 连接的服务器的ip
* @param port 连接的服务器的端口号
* @throws IOException
*/
    public void initClient(String ip,int port) throws IOException {
        // 获得一个Socket 通道
        SocketChannel channel = SocketChannel.open();
        // 设置通道为非阻塞
        channel.configureBlocking(false);
        // 获得一个通道管理器
        this.selector = Selector.open();
        // 客户端连接服务器,其实方法执行并没有实现连接，需要在listen（）方法中调
        //用channel.finishConnect();才能完成连接
        channel.connect(new InetSocketAddress(ip,port));
        //将通道管理器和该通道绑定，并为该通道注册SelectionKey.OP_CONNECT 事件。
        channel.register(selector, SelectionKey.OP_CONNECT);
```

```
    }

    /**
* 采用轮询的方式监听selector上是否有需要处理的事件，如果有，则进行处理
* @throws IOException
*/

    @SuppressWarnings("unchecked")
    public void listen() throws IOException {
        // 轮询访问selector
        while (true) {
            selector.select();
            // 获得selector中选中的项的迭代器
            Iterator ite = this.selector.selectedKeys().iterator();
            while (ite.hasNext()) {
                SelectionKey key = (SelectionKey) ite.next();
                // 删除已选的key,以防重复处理
                ite.remove();
                // 连接事件发生
                if (key.isConnectable()) {
                    SocketChannel channel = (SocketChannel) key
                    .channel();
                    // 如果正在连接，则完成连接
                    if(channel.isConnectionPending()){
                        channel.finishConnect();
```

```
                    }
                    // 设置成非阻塞
                    channel.configureBlocking(false);
                    //在这里可以给服务端发送信息哦
                    channel.write(ByteBuffer.wrap(new String("向服务端发送了一条信息"
).getBytes()));
                    //在和服务端连接成功之后，为了可以接收到服务端的信息，需要给通道设置读
的权限。
                    channel.register(this.selector, SelectionKey.OP_READ);
                    // 获得了可读的事件
                } else if (key.isReadable()) {
                    read(key);
                }
            }
        }
    }
    /**
* 处理读取服务端发来的信息 的事件
* @param key
* @throws IOException
*/
    public void read(SelectionKey key) throws IOException{
        //和服务端的read 方法一样
    }
```

```
    /**
* 启动客户端测试
* @throws IOException
*/
    public static void main(String[] args) throws IOException {
        NIOClient client = new NIOClient();
        client.initClient("localhost",8000);
        client.listen();
    }
}
```