龙果学院
roncoo.com

2015年 版教程

Java+ 您

全栈修养

技术与自我营销两手抓，做一个有追求的程序猿

Leader.us@Mycat

Java IO

本节目录

- Java IO概述
- 常见几种IO操作
- Java序列化机制
- Java内存映射文件
- Java IO设计模式分析

IO很重要！！！

龙果学院 roncoo.com

如果我告诉你，你要到内存中看这句话，而不是在屏幕上看，你会不会从此远离电脑，远离IT？

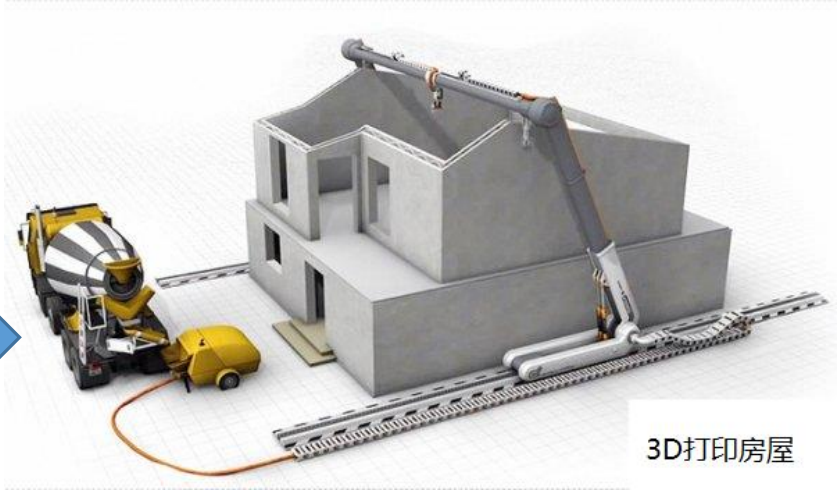
System.out.println(“Hello world”)

其实我们的终极目标是这样的.....

Print

但你们这样太俗了，所以，我的终极目标是这样的（**亲亲山庄**）

3D Print

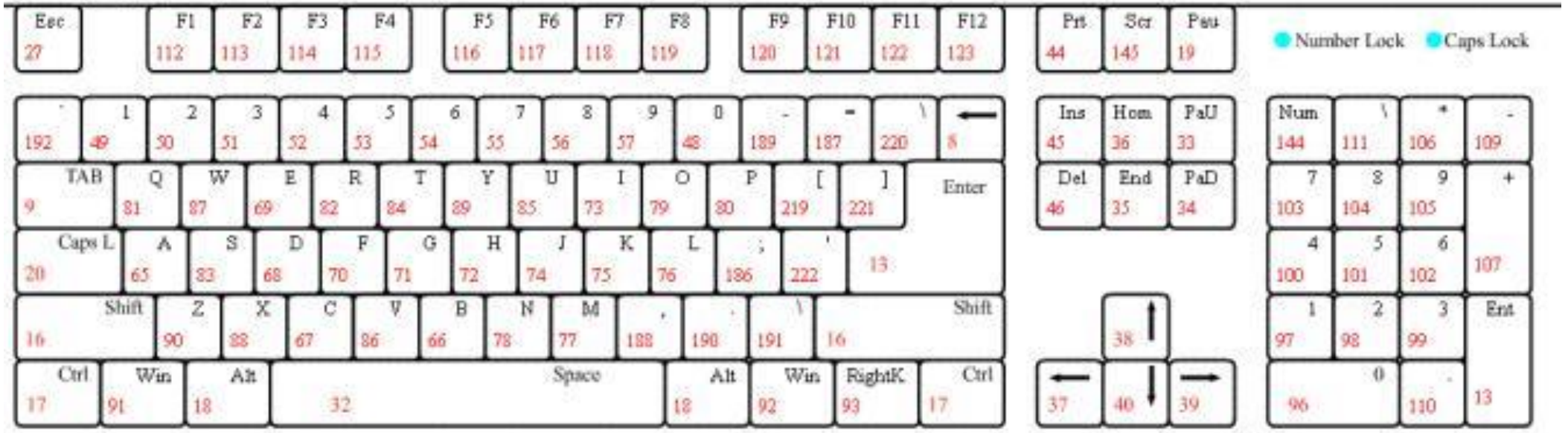


3D打印房屋

先理解ASCII字符集

ASCII数字0-9对应的byte值为48-57

键盘键值表



回车与换行是两个字符

符号	ASCII码	意义
\n	10	换行NL
\r	13	回车CR

```
/*=========================================*/
\n:  UNIX 系统行末结束符
\n\r: window 系统行末结束符
\r:  MAC OS 系统行末结束符
/*=========================================*/
```

一个直接后果是，Unix/Mac系统下的文件在Windows里打开的话，所有文字会变成一行；而Windows里的文件在Unix/Mac下打开的话，在每行的结尾可能会多出一个^M符号。

Unicode与Java

char 是字符数据类型 ，是无符号型的，占2字节(Unicode码 ）；大小范围 是0—65535 ； 是一个16位二进制的Unicode字符，JAVA用char来表示一个字符 。

UTF-16 具体定义了 Unicode 字符在计算机中存取方法。UTF-16 用两个字节来表示 Unicode 转化格式，这个是定长的表示方法，不论什么字符都可以用两个字节表示，两个字节是 16 个 bit，所以叫 UTF-16。UTF-16 表示字符非常方便，每两个字节表示一个字符，这个在字符串操作时就大大简化了操作，这也是 Java 以 UTF-16 作为内存的字符存储格式的一个很重要的原因。

UTF-8 采用了一种变长技术，每个编码区域有不同的字码长度。不同类型的字符可以是由 1~6 个字节组成。
UTF-8 有以下编码规则：

- 如果一个字节，最高位（第 8 位）为 0，表示这是一个 ASCII 字符（00 - 7F）。可见，所有 ASCII 编码已经是 UTF-8 了。
- 如果一个字节，以 11 开头，连续的 1 的个数暗示这个字符的字节数，例如：110xxxxx 代表它是双字节 UTF-8 字符的首字节。
- 如果一个字节，以 10 开始，表示它不是首字节，需要向前查找才能得到当前字符的首字节

中文大部分是3个字节的UTF-8方式编码

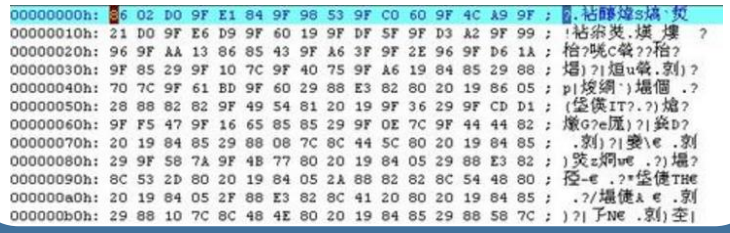
二进制PK文本编码

龙果学院 roncoo.com

假如我们要传输123456

对于二进制传输，这就是一个整数，编码为4个byte
文本方式传输，这就是6个ASCII字符，分别是’1’,’2’,’3’,’4’,’5’,’6’

二进制数据



文本数据

编程复杂、更节省IO带宽，速度更快

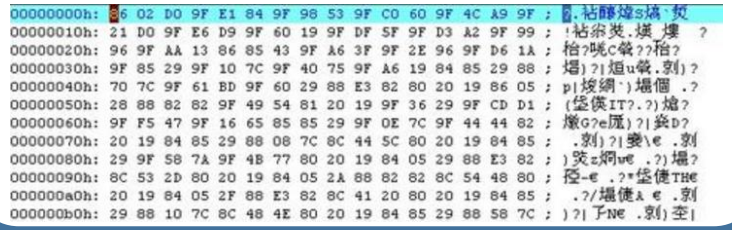
- FTP协议、P2P协议
- 流媒体协议(rtp/rtcp/rtsp/rtmp/mms/hls)
- Java RMI、Zeroc Ice
- snmp

实现简单、方便人工理解，便于排错

- HTTP
- telnet
- smtp,pop3

IO操作总揽

二进制数据



文本数据

姓名|性别|手机号码|固定电话
张三|男|839293|8932893
李四|男|3298128|09090
王五|女|0929301|909090
赵六|女|0923029|930293

File Stream

file

二进制编码，目标是内存中的数据以原生byte方式传输
文本编码，目标是传输的内容是ASCII字符数据

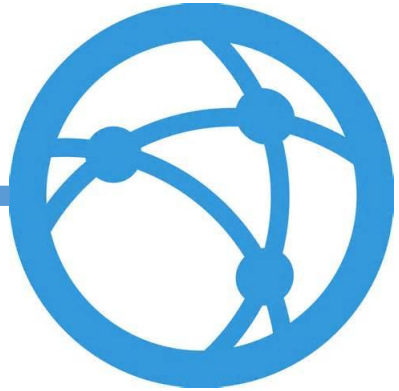
8bit Bytes

TCP/IP

TCP Stream

TCP报文 TCP报文 TCP报文

UDP报文



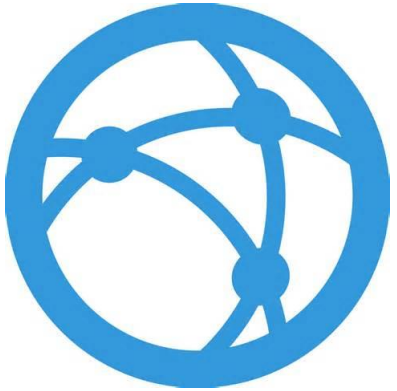
IO传输的2个关键问题

Unicode Chars/short/int/long/**Java Object**

Encode/Decode

8bit Bytes

Big Endian/Little Endian



IO里的大头小头问题

多字节数据涉及到内存中存储时候的字节顺序问题，称之为big endian & little endian

数字0x12345678在两种不同字节序CPU中的存储顺序如下所示：

Big Endian SUN、IBM的CPU

网络传输标准

```
低地址                                            高地址
---------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     12     |     34     |     56     |     78     |
```

Little Endian X86架构的CPU

```
低地址                                            高地址
---------------------------------------->
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     78     |     56     |     34     |     12     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Java默认Big-Endian


```
static short getShort(byte[] b, int off) {
        return (short) ((b[off + 1] & 0xFF) +
                        (b[off] << 8));
    }

    static int getInt(byte[] b, int off) {
        return ((b[off + 3] & 0xFF)       ) +
               ((b[off + 2] & 0xFF) <<  8) +
               ((b[off + 1] & 0xFF) << 16) +
               ((b[off    ]       ) << 24);
    }

    static long getLong(byte[] b, int off) {
        return ((b[off + 7] & 0xFFL)       ) +
               ((b[off + 6] & 0xFFL) <<  8) +
               ((b[off + 5] & 0xFFL) << 16) +
               ((b[off + 4] & 0xFFL) << 24) +
               ((b[off + 3] & 0xFFL) << 32) +
               ((b[off + 2] & 0xFFL) << 40) +
               ((b[off + 1] & 0xFFL) << 48) +
               (((long) b[off])      << 56);
    }
```

```
package java.io;

/**
 * Utility methods for packing/unpacking primitive values
in/out of byte arrays
 * using big-endian byte ordering.
 */
class Bits {
```


```
package java.nio;

import java.security.AccessController;
import sun.misc.Unsafe;
import sun.misc.VM;

/**
 * Access to bits, native and otherwise.
 */

class Bits {

    private Bits() { }
```

- ByteBufferAsCharBufferB.class
- ByteBufferAsCharBufferL.class
- ByteBufferAsCharBufferRB.class
- ByteBufferAsCharBufferRL.class
- ByteBufferAsDoubleBufferB.class
- ByteBufferAsDoubleBufferL.class

IO里的编码/解码问题

- 字符串的编码/解码问题

  charset

- 多字节数据(short/int/long/double/float)的编码/解码问题

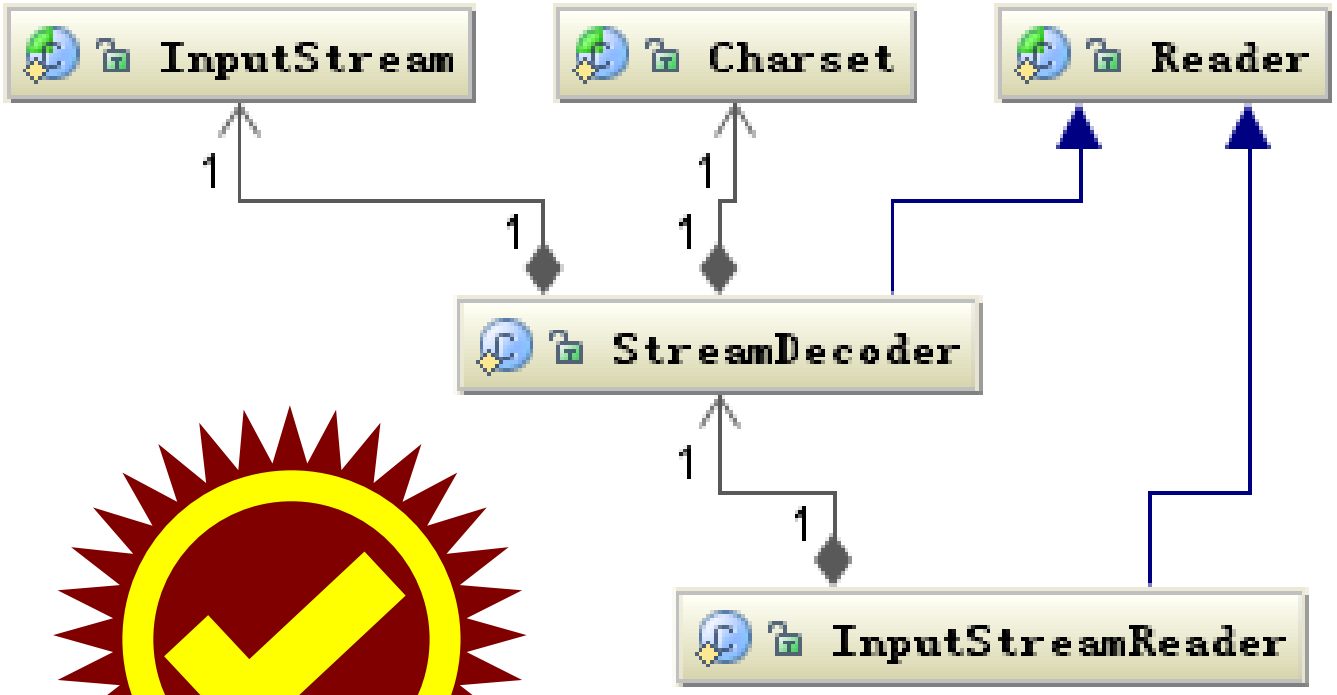
  ObjectOutputStream（默认Big-Endian顺序）

- Object对象编码/解码问题

  Java Serialize机制（二进制）以及 java.beans包里面的类XMLEncoder（文本编码）

一：普通文件IO操作

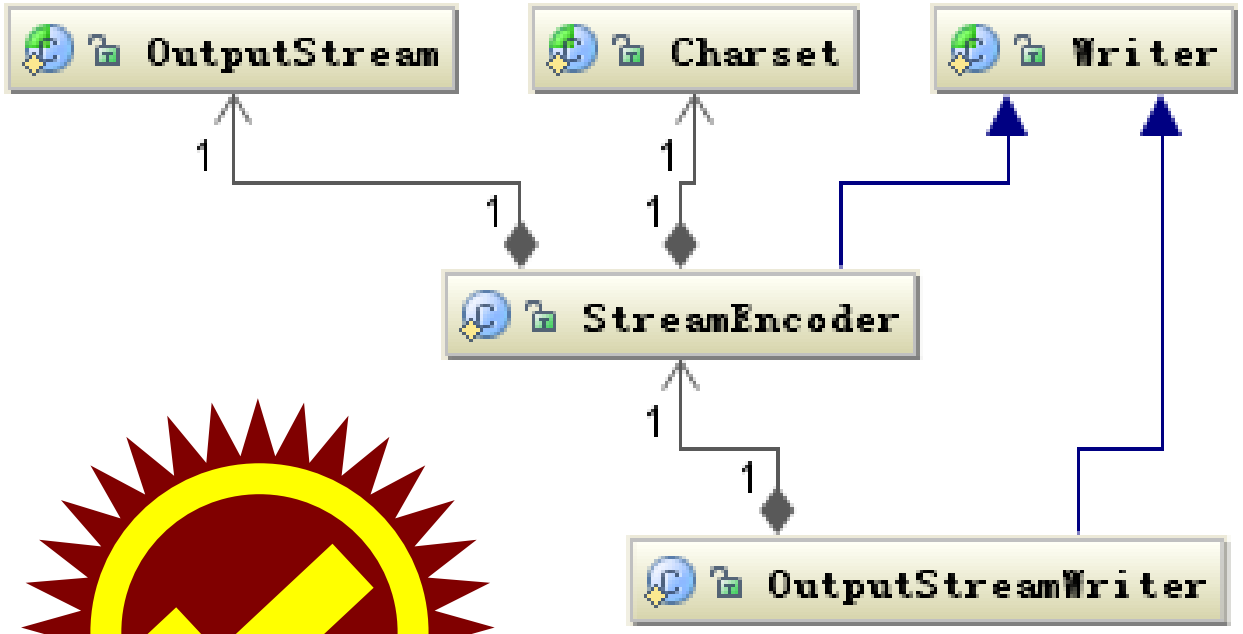
Java读文件的API




```
// 读取字节转换成字符
FileInputStream inputStream = new FileInputStream(file);
InputStreamReader reader = new InputStreamReader(
inputStream, charset);
StringBuffer buffer = new StringBuffer();
char[] buf = new char[64];
int count = 0;
try {
   while ((count = reader.read(buf)) != -1) {
      buffer.append(buffer, 0, count);
   }
} finally {
   reader.close();
}
```

Reader 类是 Java 的 I/O 中读字符的父类，而 InputStream 类是读字节的父类，InputStreamReader 类就是关联字节到字符的桥梁，它负责在 I/O 过程中处理读取字节到字符的转换，而具体字节到字符的解码实现它由 StreamDecoder 去实现，在 StreamDecoder 解码过程中必须由用户指定 Charset 编码格式。值得注意的是如果你没有指定 Charset，将使用本地环境中的默认字符集，例如在中文环境中将使用 GBK 编码。

Java写文件的API




```
String file = "c:/stream.txt";
String charset = "UTF-8";
// 写字符换转成字节流
FileOutputStream outputStream = new FileOutputStream(file);
OutputStreamWriter writer = new OutputStreamWriter(
outputStream, charset);
try {
  writer.write("这是要保存的中文字符");
} finally {
  writer.close();
}
```

字符的父类是 Writer，字节的父类是 OutputStream，通过 OutputStreamWriter 转换字符到字节。

如何转换字符编码


```
private FileInputStream fis;// 文件输入流:读取文件中内容
private InputStream is;
private InputStreamReader isr;
private OutputStream os;
private OutputStreamWriter osw;//写入
private char[] ch = new char[1024];
public  void convertionFile() throws IOException{
    is = new FileInputStream("C:/项目进度跟踪.txt");//文件读取
    isr = new InputStreamReader(is, "gbk");//解码
    os = new FileOutputStream("C:/项目进度跟踪_utf-8.txt");//文件输出
    osw = new OutputStreamWriter(os, "utf-8");//开始编码
    char[] c = new char[1024];//缓冲
    int length = 0;
    while(true){
        length = isr.read(c);
        if(length == -1){
            break;
        }
        System.out.println(new String(c, 0, length));
        osw.write(c, 0, length);
        osw.flush();
    }

}
```

二进制方式读文件

ByteArrayOutputStream:　可以捕获内存缓冲区的数据，转换成字节数组。
ByteArrayInputStream: 可以将字节数组转化为输入流

```
ByteArrayOutputStream bos = new ByteArrayOutputStream((int)f.length());
    BufferedInputStream in = null;
    try{
        in = new BufferedInputStream(new FileInputStream(f));
        int buf_size = 1024;
        byte[] buffer = new byte[buf_size];
        int len = 0;
        while(-1 != (len = in.read(buffer,0,buf_size))){
            bos.write(buffer,0,len);
        }
        return bos.toByteArray();
```

Line文本方式读文件


```
static void readAppointedLineNumber(File sourceFile, int lineNumber)
        throws IOException {
    FileReader in = new FileReader(sourceFile);
    LineNumberReader reader = new LineNumberReader(in);
    String s = "";
    if (lineNumber <= 0 || lineNumber > getTotalLines(sourceFile)) {
        System.out.println("不在文件的行数范围(1至总行数)之内。");
        System.exit(0);
    }
    int lines = 0;
    while (s != null) {
        lines++;
        s = reader.readLine();
        if((lines - lineNumber) == 0) {
            System.out.println(s);
            System.exit(0);
        }
    }
    reader.close();
    in.close();
}
```

二：Java对象序列化

Java序列化机制

Java 对象序列化是 JDK 1.1 中引入的一组开创性特性之一，用于作为一种将 Java 对象的状态转换为字节数组，以便存储或传输的机制，以后，仍可以将字节数组转换回 Java 对象原有的状态

ObjectInputStream/ObjectOutputStream

- Java序列化写入不仅是完整的类名，也包含整个类的定义，包含所有被引用的类。类定义可以是相当大的，也许构成了性能和效率的问题当一个父类实现序列化，子类自动实现序列化，不需要显式实现Serializable接口
- 当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化
- 并非所有的对象都可以序列化，,至于为什么不可以，有很多原因了,比如：
  1.安全方面的原因，某些对象不能传输到远端
  2.无法重新还原的对象，比如socket，thread、Stream等

传统序列化的编程实现


```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.Serializable;
public class Box implements Serializable{
   private int width;
   private int height;

   private void setWidth(int width){
      this.width=width;
   }

   private void setHeight(int height){
      this.height=height;
   }

   public static void main(String[] args) {
      Box myBox=new Box();
      myBox.setWidth(50);
      myBox.setHeight(30);

      try{
         FileOutputStream fos=new FileOutputStream("foo.ser");
         ObjectOutputStream os=new ObjectOutputStream(fos);
         os.writeObject(myBox);
         os.close();

      }catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

```
public void serializeSingleObject(OutputStream os, Object obj)       // 序列化单个java对象
{
    // XMLEncoder xe = new XMLEncoder(os);
    XMLEncoder xe = new XMLEncoder(os, "GBK", true, 0);     // 仅用于Java SE 7
    xe.writeObject(obj);    // 序列化成XML字符串
    xe.close();
}

public Object deserializeSingleObject(InputStream is)       // 反序列化单个Java对象
{
    XMLDecoder xd = new XMLDecoder(is);
    Object obj = xd.readObject();       // 从XML序列中解码为Java对象
    xd.close();
    return obj;
}
```

自定义控制序列化细节


```
public class Order implements Externalizable {
    private long id;
    private String description;
    private BigDecimal totalCost = BigDecimal.valueOf(0);
    private List orderLines = new ArrayList();
    private Customer customer;

    public Order() {
    }

    public void readExternal(ObjectInput stream) throws IOException,
ClassNotFoundException {
        this.id = stream.readLong();
        this.description = (String)stream.readObject();
        this.totalCost = (BigDecimal)stream.readObject();
        this.customer = (Customer)stream.readObject();
        this.orderLines = (List)stream.readObject();
    }

    public void writeExternal(ObjectOutput stream) throws IOException {
        stream.writeLong(this.id);
        stream.writeObject(this.description);
        stream.writeObject(this.totalCost);
        stream.writeObject(this.customer);
        stream.writeObject(this.orderLines);
    }
}
```

其他几种方式

任性DIY序列化：ObjectInputStream/ObjectOutputStream
直接写类的属性数据

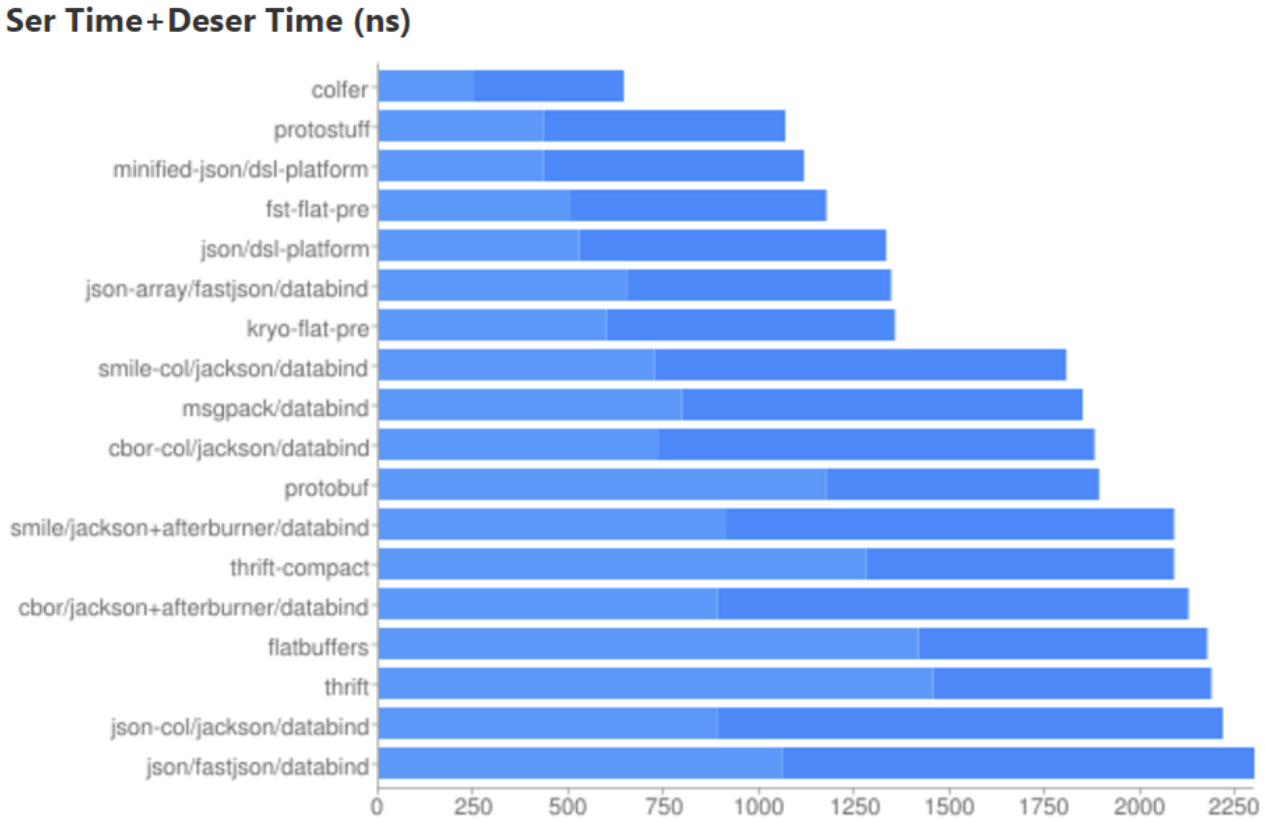
```
Person per=…
Out.writeInt(per.age)
Out.writeboolean(per.male)
…….
per=new Person()
per.age=in.readInt()
Per.male=in.readboolean()
```

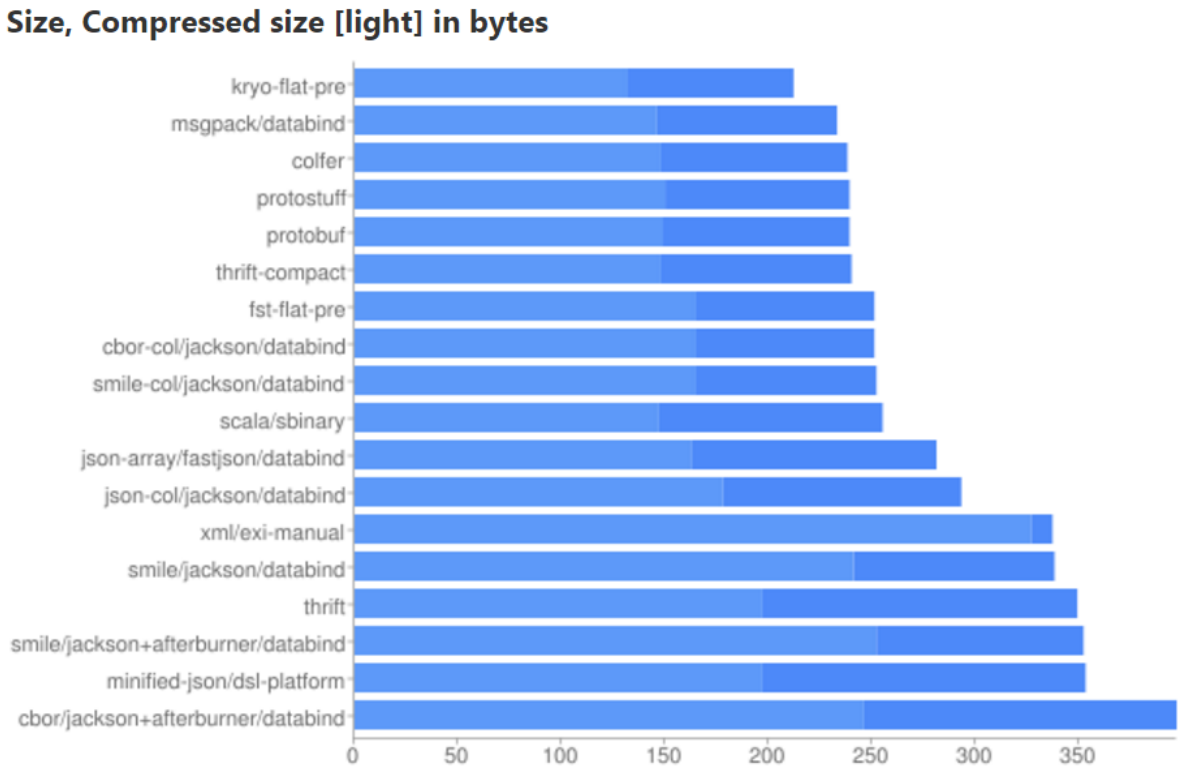
JSON框架：Gson/FastJson/…..

More（只是一小部分）….

	优点	缺点
Kryo	速度快，序列化后体积小	跨语言支持较复杂
Hessian	默认支持跨语言	较慢
Protostuff	速度快，基于protobuf	需静态编译
Protostuff-Runtime	无需静态编译，但序列化前需预先传入schema	不支持无默认构造函数的类，反序列化时需用户自己初始化序列化后的对象，其只负责将该对象进行赋值
Java	使用方便，可序列化所有类	速度慢，占空间

序列化框架的性能对比





Colfer序列化框架

Colfer build passing

Colfer is a schema-based binary data format optimized for speed and size.

The project's compiler `colf(1)` generates source code from schema definitions to marshal and unmarshall data structures.

This is free and unencumbered software released into the public domain. The format is inspired by Protocol Buffer.

Features

- Simple and straightforward in use
- Support for: Go, Java and ECMAScript/JavaScript
- No dependencies other than the core library
- Both faster and smaller than: Protocol Buffers, FlatBuffers and MessagePack
- The generated code is human-readable
- Configurable data limits with sane defaults (memory protection)
- Maximum of 127 fields per data structure
- No support for enumerations
- Framed; suitable for concatenation/streaming

TODO's

- RMI
- Lists for numbers, timestamps and binaries

序列化框架的发展方向

多语言兼容/更快/数据量更小

三：文件映射内存

把文件看做“扩展内存”

直接寻址读写数据

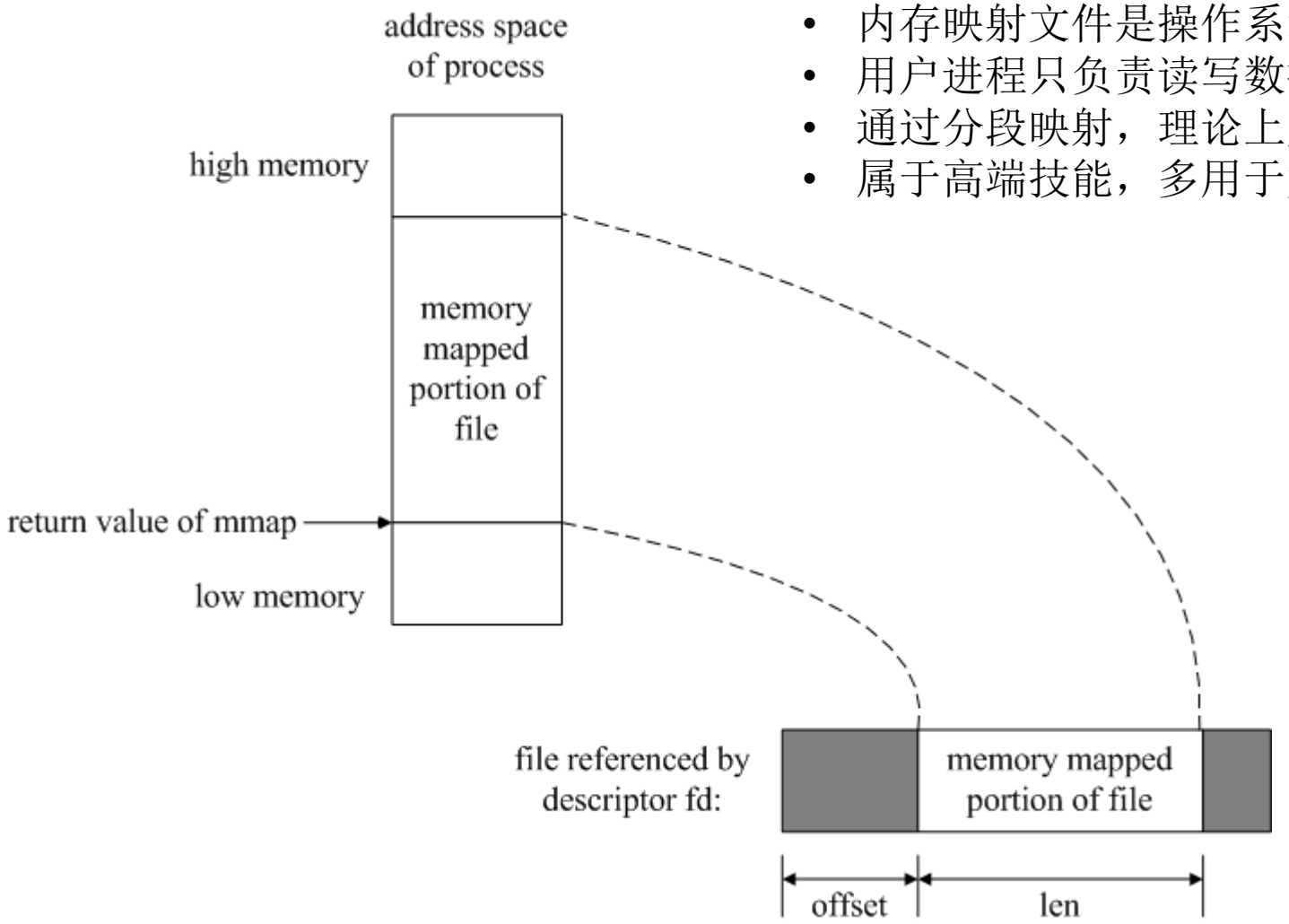
直接操作short,int,long,double,float等数据

MappedByteBuffer

高端技术，大数据和中间件领域常用

RandomAccessFile

内存映射文件



- 内存映射文件是操作系统虚拟内存管理的一部分
- 用户进程只负责读写数据，而文件与内存页的映射等复杂过程被屏蔽
- 通过分段映射，理论上用户程序可以操作很大的内存块
- 属于高端技能，多用于大数据和中间件系统

随机读文件


```
public class TestRandomAccessFile {
   public static void main(String[] args) throws IOException {
      RandomAccessFile rf = new RandomAccessFile("rtest.dat", "rw");
      for (int i = 0; i < 10; i++) {
         //写入基本类型double数据
         rf.writeDouble(i * 1.414);
      }
      rf.close();
      rf = new RandomAccessFile("rtest.dat", "rw");
      //直接将文件指针移到第5个double数据后面
      rf.seek(5 * 8);
      //覆盖第6个double数据
      rf.writeDouble(47.0001);
      rf.close();
      rf = new RandomAccessFile("rtest.dat", "r");
      for (int i = 0; i < 10; i++) {
         System.out.println("Value " + i + ": " + rf.readDouble());
      }
      rf.close();
   }
}
```

```
public class ObjectOutputStream
    extends OutputStream implements ObjectOutput,
ObjectStreamConstants
{
```

MappedByteBuffer

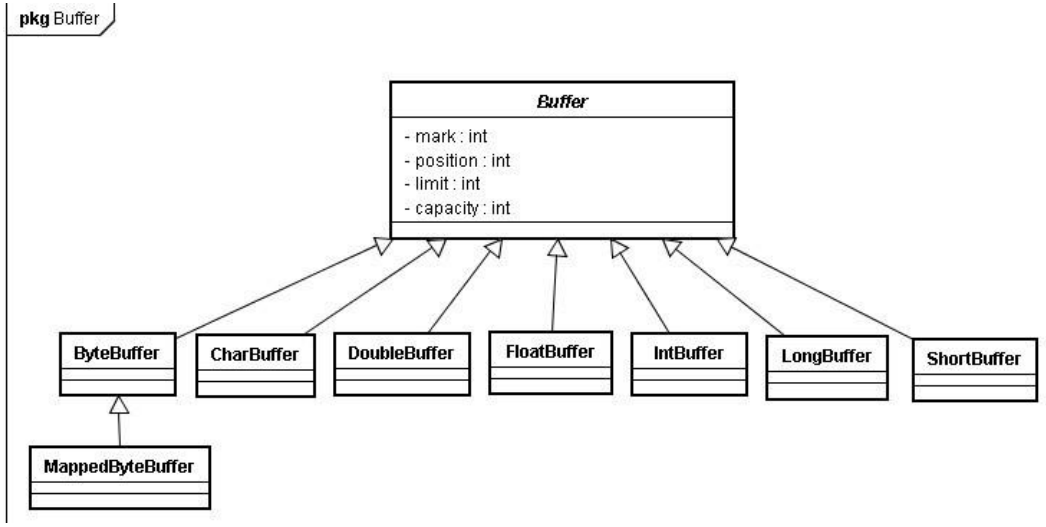

```
RandomAccessFile inRandomAccessFile = new RandomAccessFile(srcfilePath, "r");
    FileChannel inFileChannel = inRandomAccessFile.getChannel();
    MappedByteBuffer inMappedByteBuffer = inFileChannel.map(MapMode.READ_ONLY, 0, inFileChannel.size());
    inRandomAccessFile.close();
    inFileChannel.close();
```

MapMode 映射模式

1.READ_ONLY 只读映射模式

2.READ_WRITE 读/写映射模式

3.PRIVATE 通过put方法对MappedByteBuffer的修改不会修改到磁盘文件 只是虚拟内存的修改



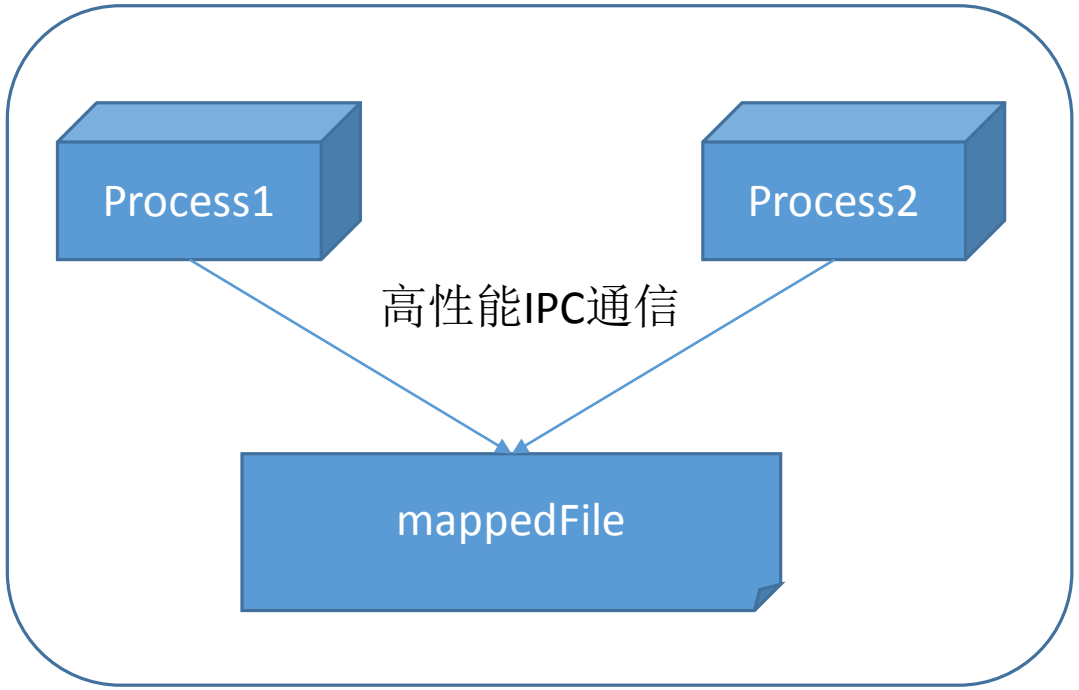
MappedByteBuffer在父类ByteBuffer的基础上 新增的几个方法

1.fore缓冲区在READ_WRITE模式下，此方法对缓冲区所做的内容更改强制写入文件

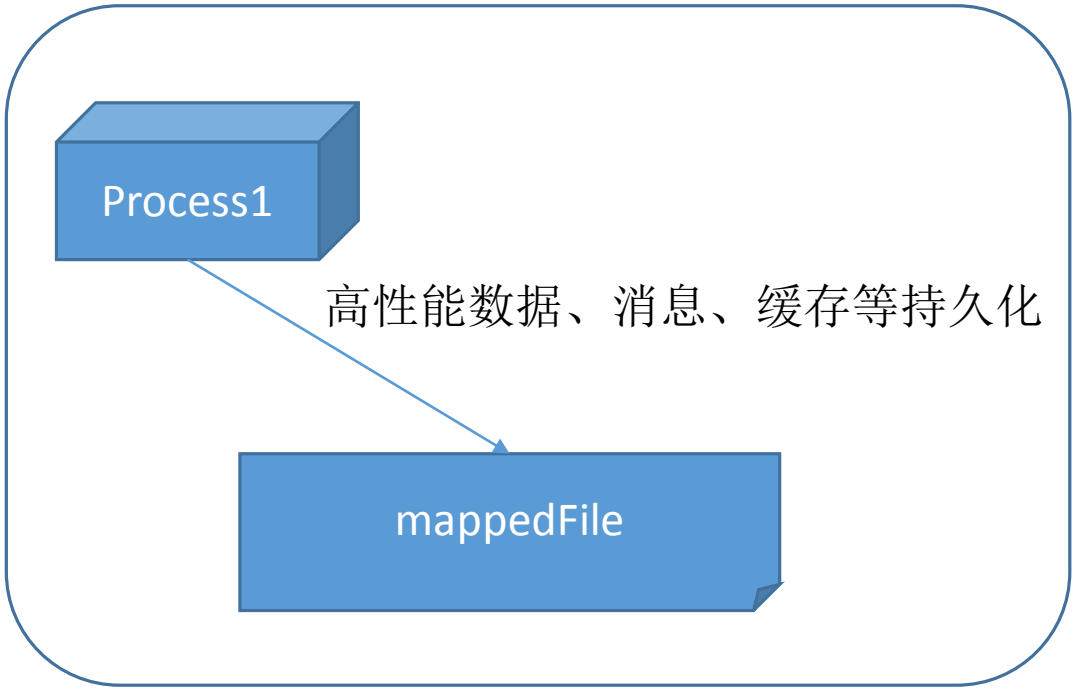
2.load：将缓冲区的内容载入物理内存，并返回该缓冲区的引用

3.isLoaded：判断缓冲区的内容是否在物理内存，如果在则返回true，否则返回false

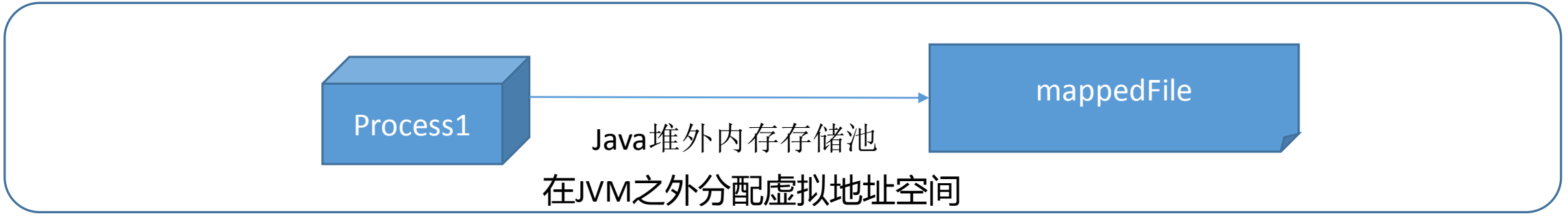
MappedByteBuffer用途



https://github.com/MyCATApache/Mycat-IPC



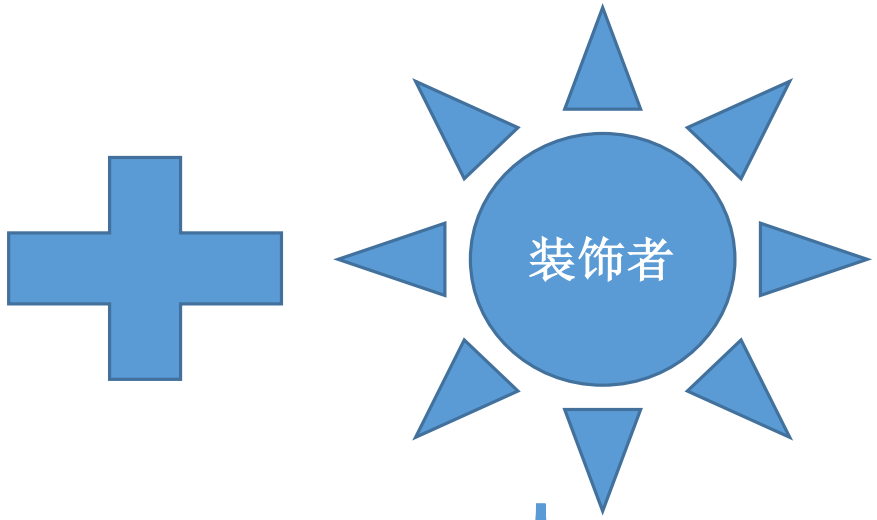
ActiveMQ、Kafka



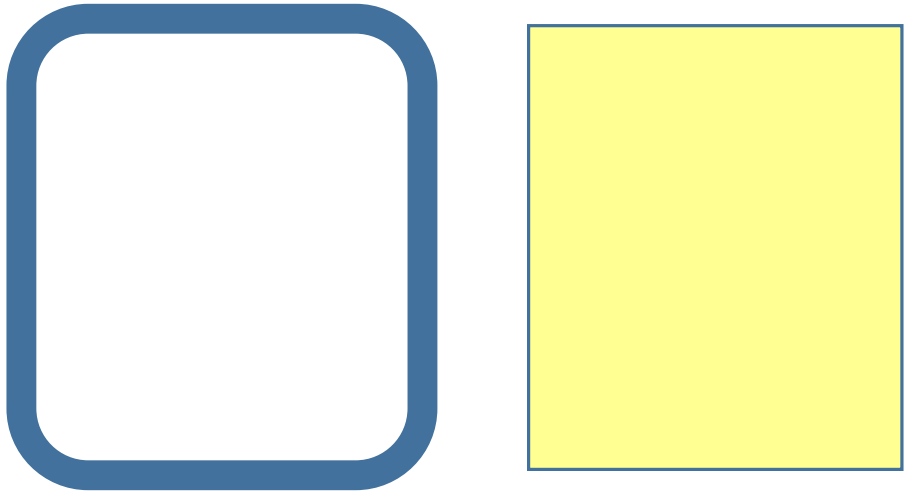
四：Java　IO类库设计模式

Java IO 的装饰者模式

被装饰主体

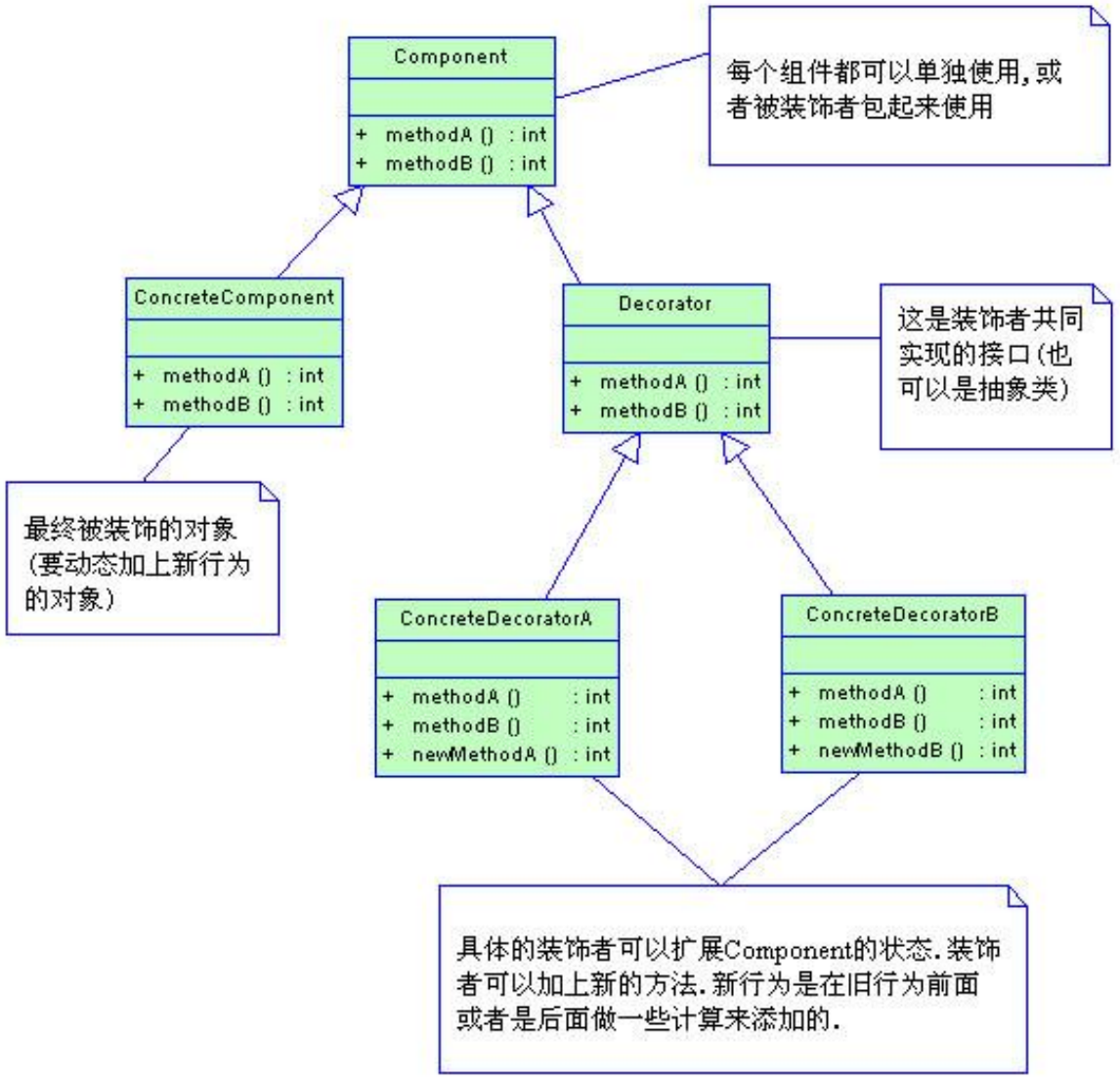


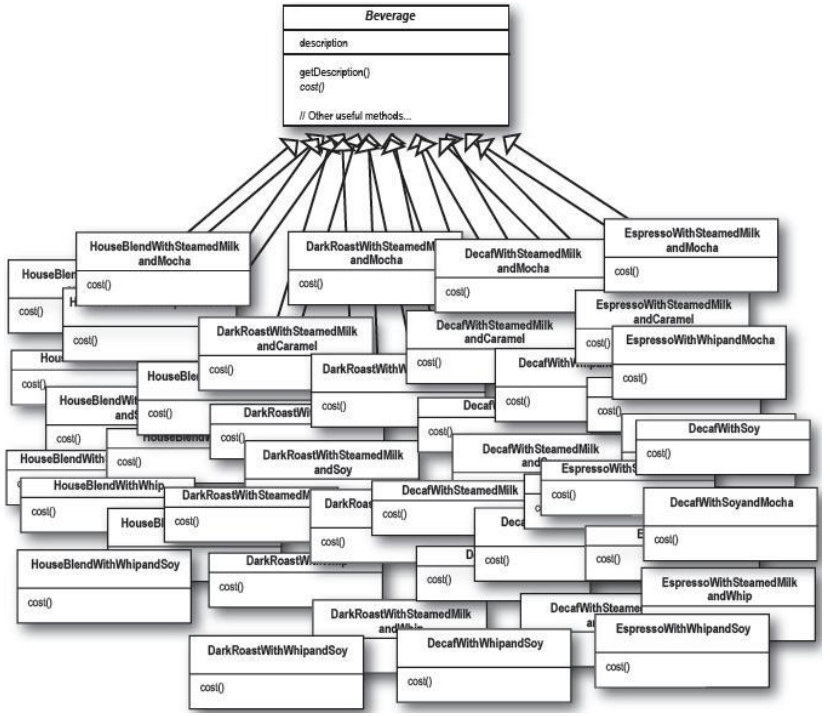
装饰的效果



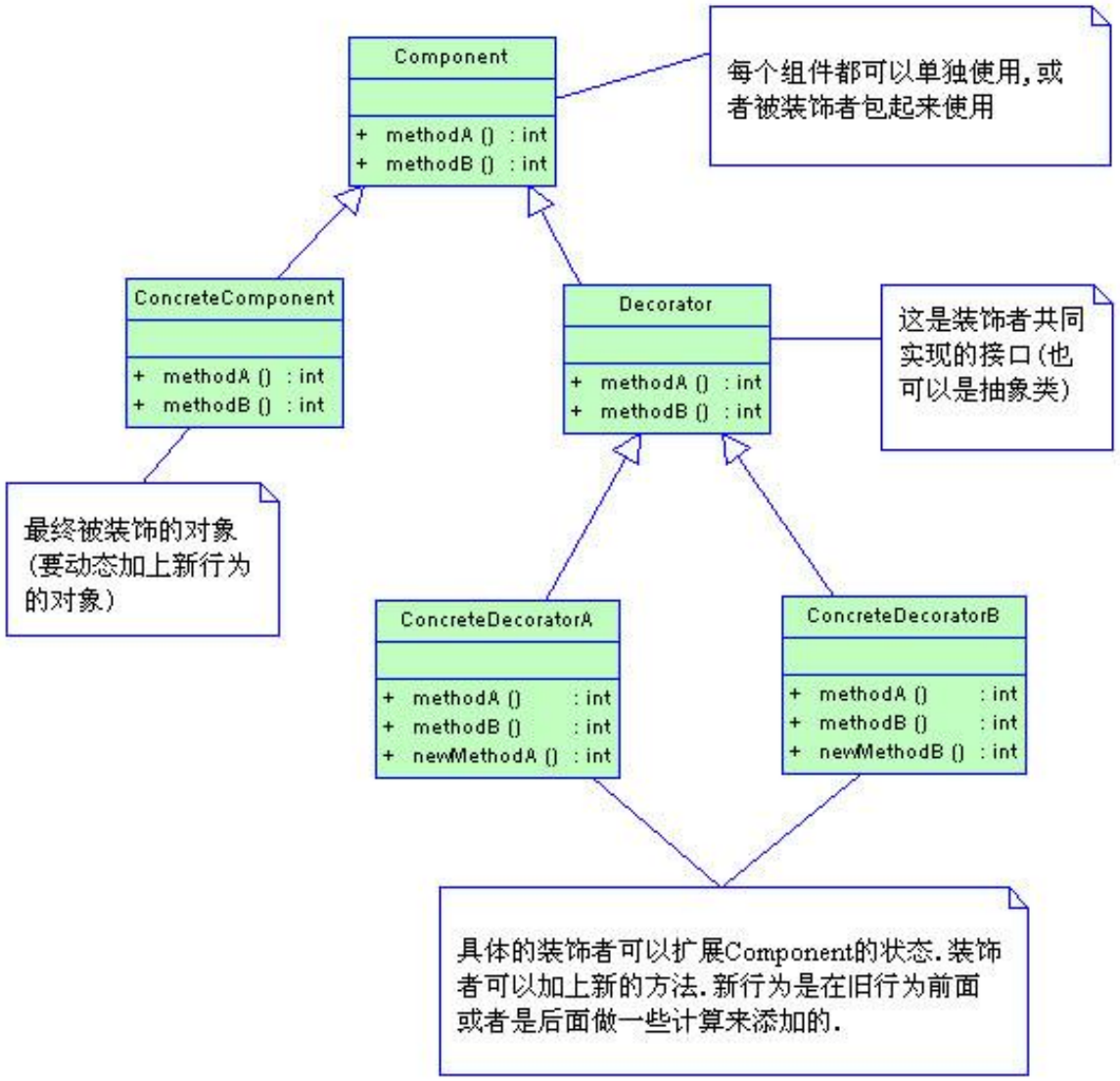
装饰者模式不能改变主体，与PS还是有很大区别

装饰者模式





装饰者模式



OutputStream是一个抽象类，它是所有输出流的公共父类，相当于Component接口：

```
public abstract class OutputStream implements Closeable，Flushable {
    public abstract void write（ int b ） throws IOException；
    ……
    }
```

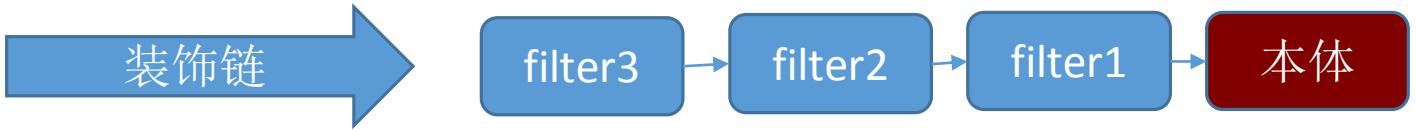
FileOutputStream相当于ConcreteComponent

```
public class FileOutputStream extends OutputStream
{
 ....
}
```

装饰者模式

FilterOutputStream相当于一个具体的Decorator（接口）

```
public class FilterOutputStream extends OutputStream {
protected OutputStream out;
public FilterOutputStream（OutputStream out） {
this.out = out;
}
public void write（int b） throws IOException {
out.write（b）;
}
……
}
```



最终用法如下：

```
try {
    OutputStream out = new DataOutputStream ( new
FileOutputStream ( "test.txt" ) ) ;
    } catch ( FileNotFoundException e ) {
    e.printStackTrace ( ) ;
    }
```

相当于一个具体的ConcreteDecorator

```
public class DataOutputStream extends FilterOutputStream
implements DataOutput {
```

龙果学院
roncoo.com

谢谢观看

Leader全栈Java报名群QQ 332702697