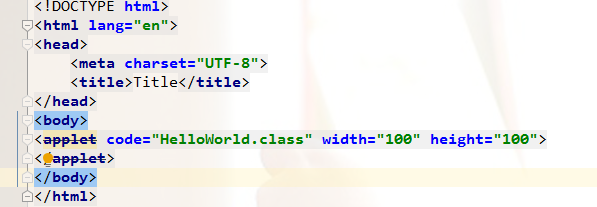
## applet

### package.html

Java [Applet](https://baike.baidu.com/item/Applet" \t "https://baike.baidu.com/item/java.applet/_blank)就是用Java语言编写的一些[小应用程序](https://baike.baidu.com/item/%E5%B0%8F%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F/3350978" \t "https://baike.baidu.com/item/java.applet/_blank)，它们可以直接嵌入到网页中，并能够产生特殊的效果。包含Applet的网页被称为Java-Powered页，可以称其为Java支持的网页。示例如下



如果浏览器支持applet那么打开html会显示Hello World！内容，这项技术由SUN公司设计，但是随着时代的发展目前被JavaScript/VBScript等脚本语言所取代

## awt

### package.html

AWT(Abstract Window Toolkit)，中文译为抽象窗口工具包，该包提供了一套与本地图形界面进行交互的[接口](https://baike.baidu.com/item/%E6%8E%A5%E5%8F%A3/15422203" \t "https://baike.baidu.com/item/AWT/_blank)，是Java提供的用来建立和设置Java的[图形用户界面](https://baike.baidu.com/item/%E5%9B%BE%E5%BD%A2%E7%94%A8%E6%88%B7%E7%95%8C%E9%9D%A2/3352324" \t "https://baike.baidu.com/item/AWT/_blank)的基本工具。AWT中的图形函数与操作系统所提供的图形函数之间有着一一对应的关系，称之为[peers](https://baike.baidu.com/item/peers/6664140" \t "https://baike.baidu.com/item/AWT/_blank)，当利用AWT编写[图形用户界面](https://baike.baidu.com/item/%E5%9B%BE%E5%BD%A2%E7%94%A8%E6%88%B7%E7%95%8C%E9%9D%A2/3352324" \t "https://baike.baidu.com/item/AWT/_blank)时，实际上是在利用本地操作系统所提供的图形库。由于不同 操作系统的图形库所提供的样式和功能是不一样的，在一个平台上存在的功能在另一个平台上则可能不存在。为了实现Java语言所宣称的“一次编写，到处运行(write once, run anywhere)”的概念，AWT不得不通过牺牲功能来实现平台无关性，也即AWT所提供的图形功能是各种操作系统所提供的图形功能的交集。

javax.swing

Swing ：为解决 AWT 存在的问题而新开发的图形界面包。Swing是对AWT的改良和扩展,，不仅提供了AWT 的所有功能，还用纯粹的Java代码对AWT的功能进行了大幅度的扩充。

java 开发的桌面应用如eclipse、idea、俄罗斯方块等，一般都用公司自己开发的awt，因为样式和速度问题一般桌面应用很少用java awt开发

## beans

### package.html

**java bean**

符合一定规范的编写的Java类，只是一种规范而不是某种类或者说某种技术，为了实现这个规范，要满足下面四个条件。

1、所有属性为private  
2、提供默认构造方法  
3、提供getter和setter  
4、实现serializable接口

Java语言欠缺属性、事件、多重继承功能。所以，如果要在Java程序中实现一些面向对象编程的常见需求，只能手写大量胶水代码。Java Bean正是编写这套胶水代码的惯用模式或约定。这些约定包括getXxx、setXxx、isXxx、addXxxListener、XxxEvent等。遵守上述约定的类可以用于若干工具或库。

举例说明

假如有人要用Java实现一个单向链表类，可能会这样写

|  |
| --- |
| // 编译成 java-int-list\_1.0.jar  public final class JavaIntList {  static class Node {  public Node next;  public int value;  }  public Node head;  public int size;  } |

上述实现为了能够快速获取链表的大小，把链表大小缓存在size变量中。用法如下：

|  |
| --- |
| JavaIntList myList = new JavaIntList();  System.out.println(myList.size); |

JavaIntList的作者很满意，于是开源了java-int-list库的1.0版。文件名是java-int-list\_1.0.jar。发布后，吸引了许多用户来使用java-int-list\_1.0.jar。  
有一天，作者决定要节省内存，不要缓存size变量了，把代码改成这样：

|  |
| --- |
| // 编译成 java-int-list\_2.0.jar  public final class JavaIntList {  static final class Node {  public Node next;  public int value;  }  public Node head;  public int getSize() {  Node n = head;  int i = 0;  while (n != null) {  n = n.next;  i++;  }  return i;  }  } |

然后发布了2.0版：java-int-list\_2.0.jar。发布后，原有java-int-list\_1.0.jar的用户纷纷升级版本到2.0。这些用户一升级，就发现自己的程序全部坏掉了，说是找不到什么size变量。会导致再也不敢用java-int-list库了。

这个故事告诉我们，如果不想被暴打致死，你就必须保持向后兼容性。太阳公司在设计Java语言时，也懂得这个道理。所以Java标准库中，绝对不会出现public int size这样的代码，而一定会一开始就写成：

|  |
| --- |
| private int size;  public int getSize() { return size; } |

让用户一开始就使用getSize，以便有朝一日修改getSize实现时，不破坏向后兼容性。这种public int getSize() { return size; }的惯用手法，就是Java Bean。

**EJB**  
在企业开发中，需要可伸缩的性能和事务、安全机制，这样能保证企业系统平滑发展，而不是发展到一种规模重新更换一套软件系统。 然后有提高了协议要求，就出现了Enterprise JavaBean（企业级的JavaBean）。EJB在javabean基础上又提了一些要求，当然更复杂了

**POJO**

"Plain Ordinary Java Object"，简单普通的java对象。主要用来指代那些没有遵循特定的java对象模型，约定或者框架的对象。有一些private的参数作为对象的属性，然后针对每一个参数定义get和set方法访问的接口。没有从任何类继承、也没有实现任何接口，更没有被其它框架侵入的java对象。

**总结**

在java1996年发布,当年12月即发布了java bean1.00-A,有什么用呢?通过统一的规范可以设置对象的值(get,set方法),这是最初的java bean;在实际企业开发中,需要实现事务,安全,分布式,javabean就不好用了.sun公司就开始往上面堆功能,这里java bean就复杂为EJB;EJB功能强大,但是太重了.此时出现DI(依赖注入),AOP(面向切面)技术,通过简单的java bean也能完成EJB的事情,这里的java bean简化为POJO，Spring此时诞生了.

PO(persistence object):用于持久化时(例如保存到数据库或者缓存);

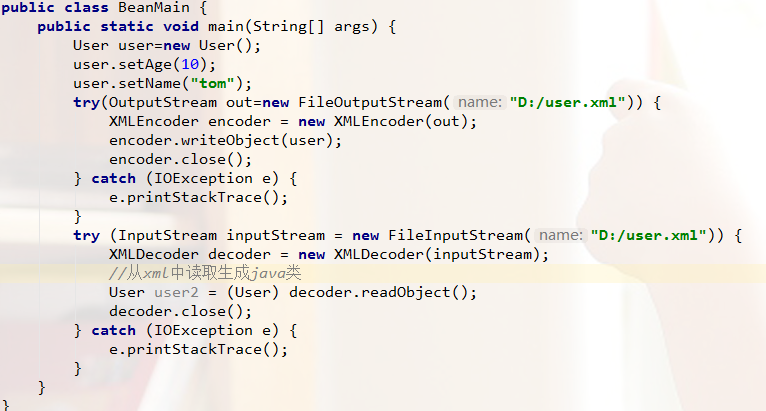
VO(value object):用于前端展示使用(例如放置到JSP中解析或者给前端传递数据)

DTO(data transfer object):用于接口互相调用返回,数据传输(例如很多接口调用返回值或消息队列内容);

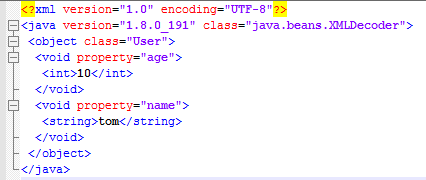
**java.beans**

java.beans包包含了开发bean相关的类,各种操作自定义bean的类,比如持久化类实例,内省实现。

如java.beans.XMLEncoder将bean类写入到xml中，XMLDecoder将xml信息生成bean类



生成的user.xml



## io

### package.html

**概述**

流是一种抽象概念，它代表了数据的无结构化传递。按照流的方式进行输入输出，数据被当成无结构的字节序或字符序列。从流中取得数据的操作称为提取操作，而向流中添加数据的操作称为插入操作。

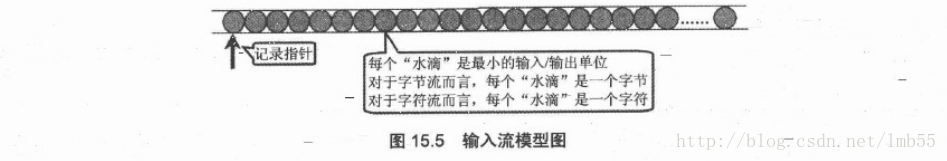
用来进行输入输出操作的流就称为IO流（Input Output Stream）。换句话说，IO流就是以流的方式进行输入输出。我们通常所说的IO流指的是Input和Output，可以将数据写入文件，也可以读取文件中的数据。

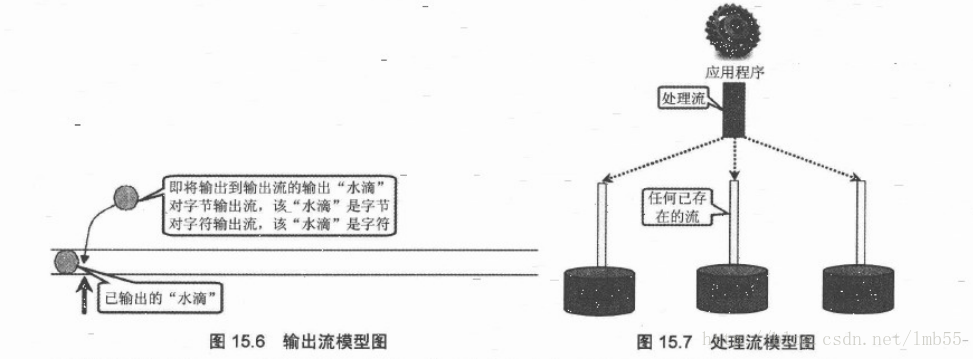
**目的**

集合或数组保存数据的时候将数据保存在内存中，一旦断电，就没有了。因为内存中存储的数据是临时的，所以想要将数据永久保存，则可以存储在文件中，我们可以通过IO流我们可以将数据写入到文件中，也可以将文件中的数据读取到程序中。比如我们经常用到的读取文件拷贝文件等，都是IO流的具体实现。

**输出流和输出流（以方向分类）**

IO流（Input Output Stream），顾名思义，即Input输入流和Output输出流，其中输入输出流均有字符流和字节流的用法，比如字节流：输入流为InputStream，输出流为OutputStream；而字符流：输入流为Reader，输出流为Writer。





**字节流和字符流（以传输单位分类）**

字节流：传输过程中，传输数据的最基本单位是字节的流。read返回的是一个字节

字符流：传输过程中，传输数据的最基本单位是字符的流。read返回的是一个或多个字符

| **流** | **字节流** | **字符流** |
| --- | --- | --- |
| 读取方式 | 字节（byte） | 字符（char） |
| 速度 | 较慢 | 较快 |
| 效率 | 低 | 高 |
| 适用场景 | 计算机中都以字节存储，所以字节流可以处理所有文件 | 以字符为单位读取，所以只能处理纯文本文件，若处理图片则会乱码破坏文件 |

**步骤**

在IO流里，输入输出分为4步：格式化/解析，缓冲，编码转换和传递。

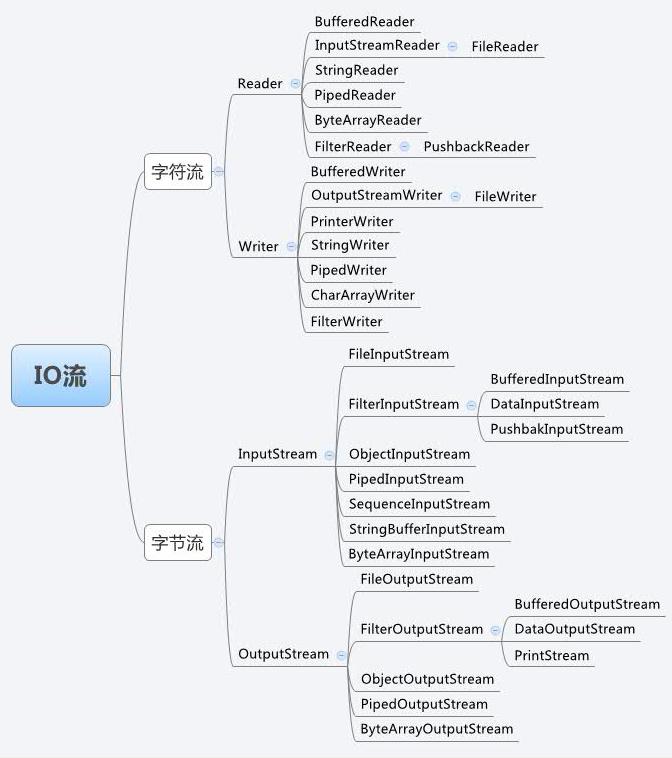
1. 格式化/解析：在内部数据表示（以字节为单位）与外部数据表示（以字符为单位）之间进行双向转换。例如一个2字节的整数10002，就需要5个字符来表示。
2. 缓冲：用于在格式/解析与传递之间缓存字符序列。对于输出，较短的字符序列格式化之后并不马上输出，而是保存在缓冲区里，待累积到一定规模之后再传递到外部设备。相反，从外部设备读入的大量数据也是先放在缓冲区，然后逐步取出完成输入。默认时，IO流的输入输出都是经过缓冲的，也可以让IO流工作在无缓冲模式下。
3. 编码转换： 将一种字符表达式转换成另一种字符表达式。如果格式化产生的字符表达式与外部字符表达式不同（输出时），或者外部表达式与IO流能解析的表达式不同（输入时），就必须进行编码转换。如多字节编码与宽字符编码之间的转换等。多数情况下并不需要进行编码转换。
4. 传递：主要是与外部设备进行通信。输出时，传递负责将经过格式化、缓冲即编码转换后的字符序列发送到外部设备；输入时，则负责将外部设备抽取数据，为其后进行的编码转换、缓冲及解析提供字符序列。

**具体实现过程**

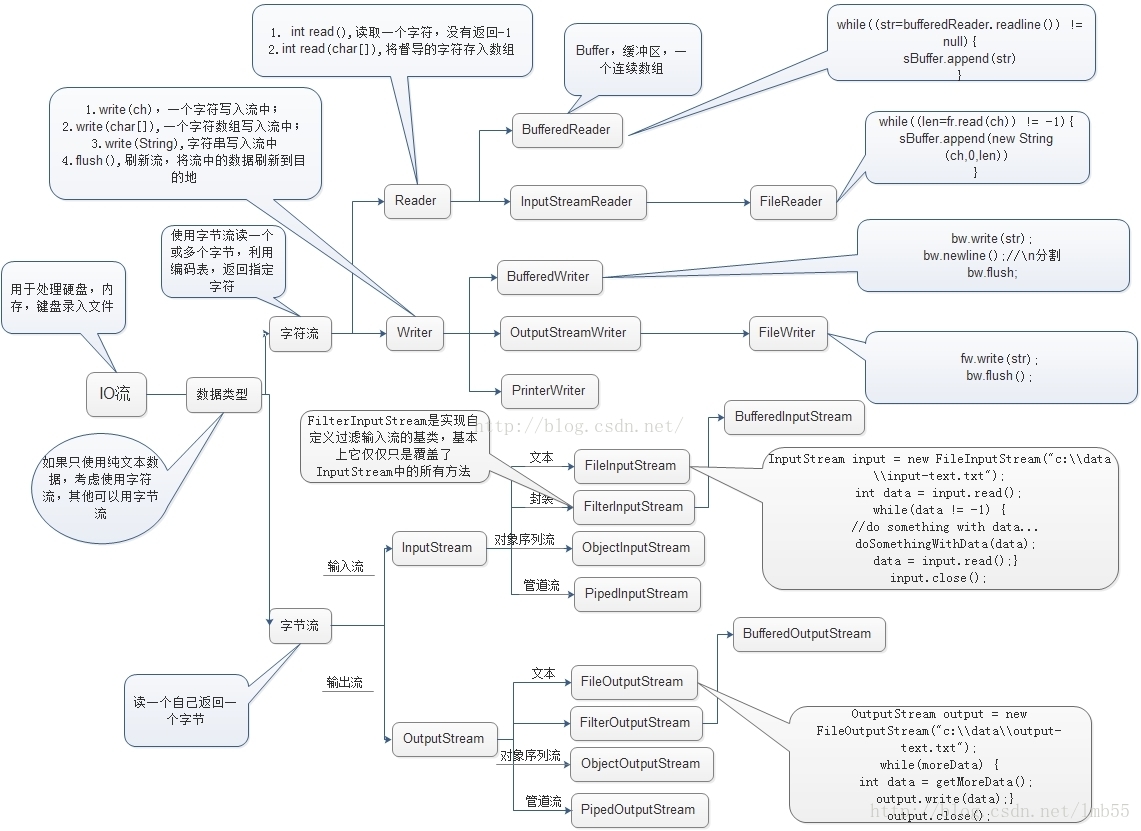
1. 确定源和目标文件
2. 构建流对象
3. 读取内容
4. 写出内容
5. 关闭流

**java.io**

java对io流处理的api



java输入/输出流体系中常用的流的分类表  

### File

### InputStream

### OutputStream

### FileInputStream

### FileOutputStream

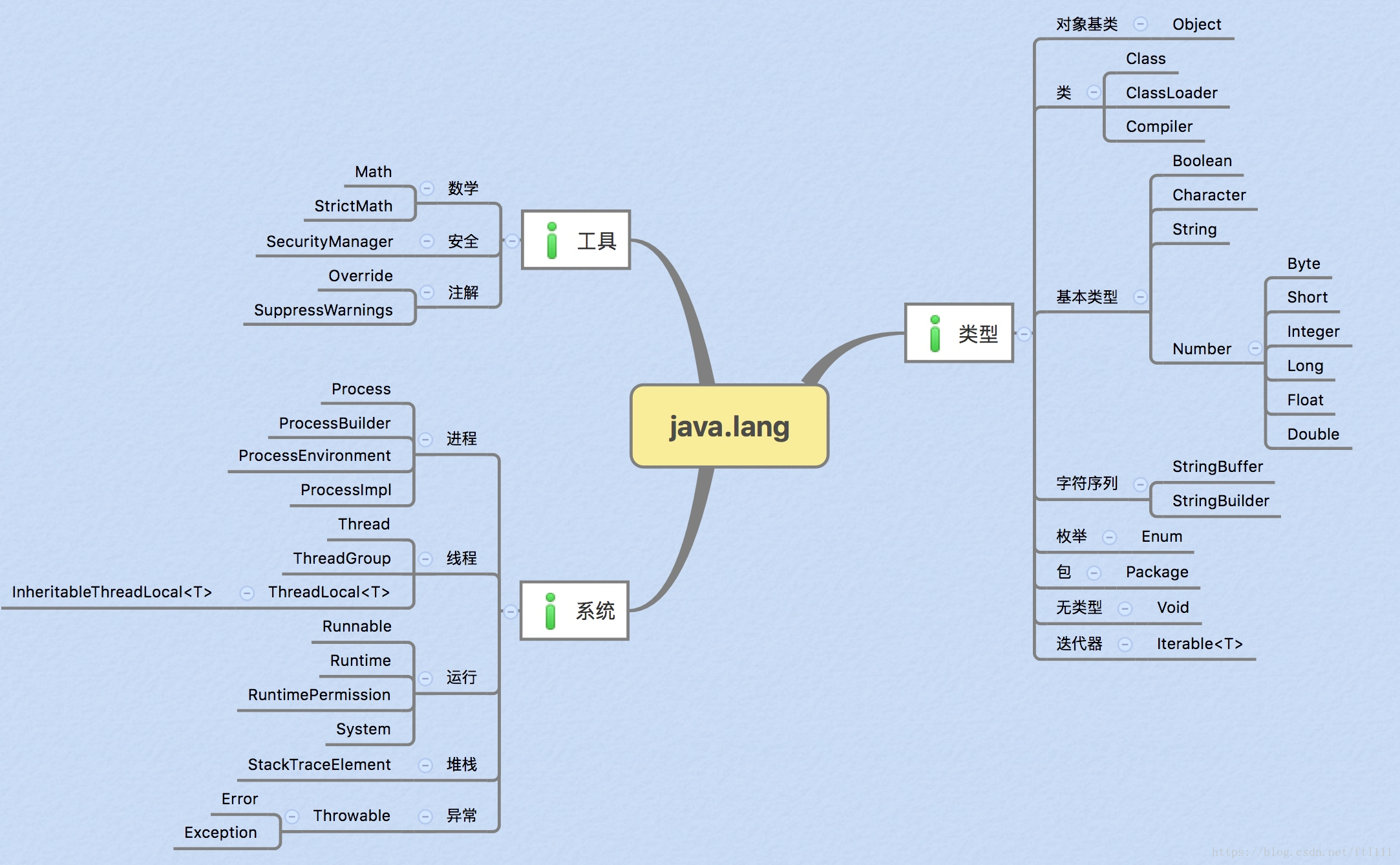
### Reader

### Writer

## lang

### package.html

java.lang包是Java语言的核心类库（lang是language的缩写），包括了运行Java程序必不可少的系统类，如基本数据类型、基本数学函数、字符串处理、线程、异常处理类等。每个Java程序运行时，系统都会自动地引入java.lang包，所以这个包的加载是缺省的。



### annotation

#### package.html

### instrument

#### package.html

### invoke

#### package.html

### management

#### package.html

### ref

#### package.html

### reflect

#### package.html

## math

### package.html

## net

### package.html

## nio

### package.html

**同步、异步、阻塞、非阻塞的理解**

**同步**

所谓同步，就是发出一个功能调用时，在没有得到结果之前，该调用就不返回或继续执行后续操作。简单来说，同步就是必须一件一件事做，等前一件做完了才能做下一件事。

例如：B/S模式中的表单提交，具体过程是：客户端提交请求->等待服务器处理->处理完毕返回，在这个过程中客户端（浏览器）不能做其他事。

**异步**

异步与同步相对，当一个异步过程调用发出后，调用者在没有得到结果之前，就可以继续执行后续操作。当这个调用完成后，一般通过状态、通知和回调来通知调用者。对于异步调用，调用的返回并不受调用者控制。

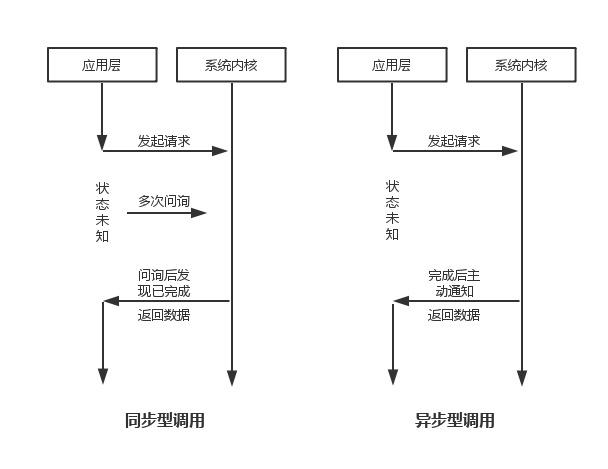
对于通知调用者的三种方式

状态：即监听被调用者的状态（轮询），调用者需要每隔一定时间检查一次，效率会很低。

通知：当被调用者执行完成后，发出通知告知调用者，无需消耗太多性能。

回调：与通知类似，当被调用者执行完成后，会调用调用者提供的回调函数。

例如：B/S模式中的ajax请求，具体过程是：客户端发出ajax请求->服务端处理->处理完毕执行客户端回调，在客户端（浏览器）发出请求后，仍然可以做其他的事。

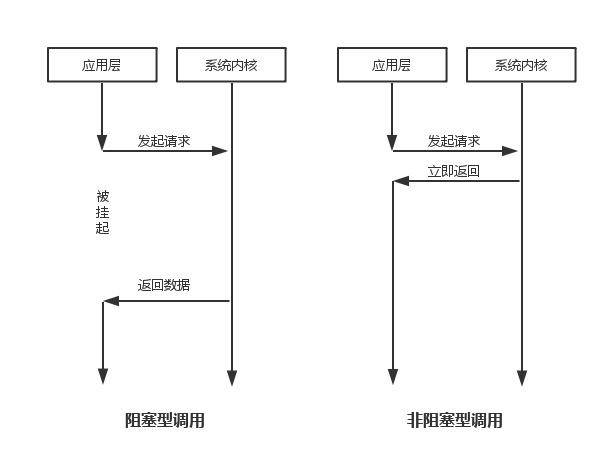


**阻塞**

指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回

**非阻塞**

指在不能立刻得到结果之前，该调用不会阻塞当前线程。



**重点注意**

**同步/异步关注的是消息通知的机制，而阻塞/非阻塞关注的是程序（线程）等待消息通知时的状态。**

同步阻塞：小明一直盯着下载进度条，到 100% 的时候就完成。

同步体现在：等待下载完成通知；

阻塞体现在：等待下载完成通知过程中，不能做其他任务处理；

同步非阻塞：小明提交下载任务后就去干别的，每过一段时间就去瞄一眼进度条，看到 100% 就完成。

同步体现在：等待下载完成通知，但是要在；

非阻塞体现在：等待下载完成通知过程中，去干别的任务了，只是时不时会瞄一眼进度条；【小明必须要在两个任务间切换，关注下载进度】

异步阻塞：小明换了个有下载完成通知功能的软件，下载完成就“叮”一声。不过小明仍然一直等待“叮”的声音（看起来很傻，不是吗）。

异步体现在：下载完成“叮”一声通知；

阻塞体现在：等待下载完成“叮”一声通知过程中，不能做其他任务处理；

异步非阻塞：仍然是那个会“叮”一声的下载软件，小明提交下载任务后就去干别的，听到“叮”的一声就知道完成了。

异步体现在：下载完成“叮”一声通知；

非阻塞体现在：等待下载完成“叮”一声通知过程中，去干别的任务了，只需要接收“叮”声通知即可；软件处理下载任务，小明处理其他任务，不需关注进度，只需接收软件“叮”声通知即可

**nio**

new IO/non-block IO非阻塞io的java api，传统io是阻塞的，如执行read方式时程序一直停留在方法里面等待数据，没有数据就不继续往下执行，至到得到数据。传统的IO流是阻塞式的，会一直监听一个ServerSocket，在调用read等方法时，他会一直等到数据到来或者缓冲区已满时才返回(网络编程socket通信属于io操作)。调用accept也是一直阻塞到有客户端连接才会返回。每个客户端连接过来后，服务端都会启动一个线程去处理该客户端的请求。并且多线程处理多个连接。每个线程拥有自己的栈空间并且占用一些 CPU 时间。每个线程遇到外部未准备好的时候，都会阻塞掉。阻塞的结果就是会带来大量的进程上下文切换。

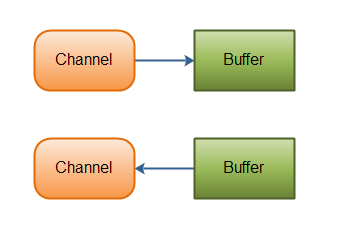
**IO（BIO）和NIO核心区别**

其本质就是阻塞和非阻塞的区别，IO为同步阻塞形式,NIO为同步非阻塞形式,NIO并没有实现异步,在JDK1.7后升级NIO库包，支持异步非阻塞

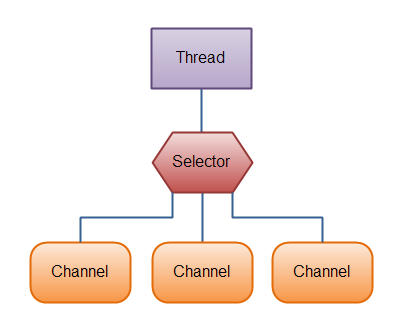
**java.nio**

nio的java api，Java NIO核心由Channels、buffers、Selectors组成。

基本上，所有的 IO 在NIO 中都从一个Channel 开始。Channel 有点象流。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中。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，是双向的。



Selector允许单线程处理多个 Channel。如果应用打开了多个连接（通道），但每个连接的流量都很低，使用Selector就会很方便。这是在一个单线程中使用一个Selector处理3个Channel的图示



示例：文件复制

|  |
| --- |
| private static void copyFileUsingFileChannels(File source, File dest) throws IOException {  FileChannel inputChannel = null;  FileChannel outputChannel = null;  try {  inputChannel = new FileInputStream(source).getChannel();  outputChannel = new FileOutputStream(dest).getChannel();  outputChannel.transferFrom(inputChannel, 0, inputChannel.size());  } finally {  inputChannel.close();  outputChannel.close();  }  } |

## rmi

### package.html

## security

### package.html

## sql

### package.html

## text

### package.html

## time

### package.html

## util

### package.html