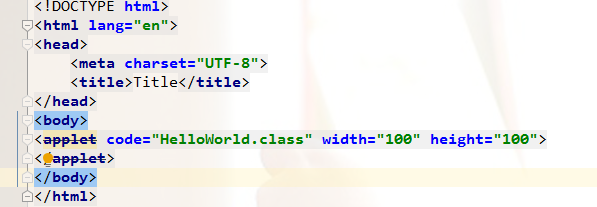
## applet

### 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);

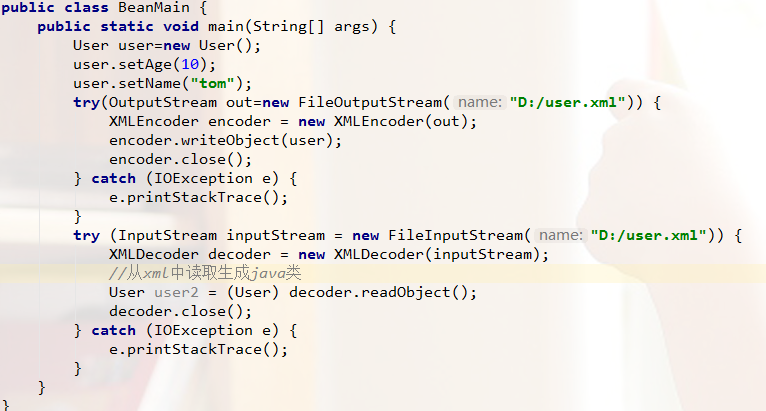
VO(value object):用于前端展示使用(例如放置到JSP中解析或者给前端传递数据)

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);

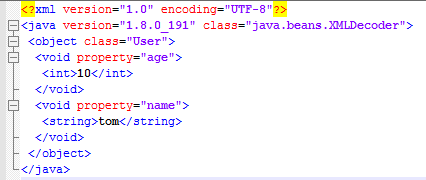
**java.beans**

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。

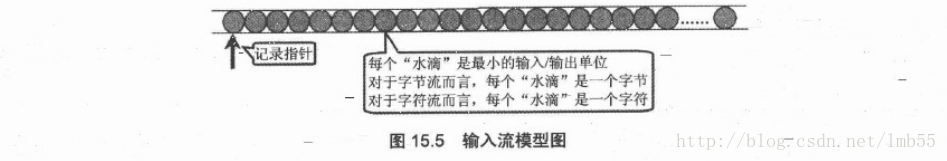
用来进行输入输出操作的流就称为IO流（Input Output Stream）。换句话说，IO流就是以流的方式进行输入输出。我们通常所说的IO流指的是Input和Output，可以将数据写入文件，也可以读取文件中的数据。

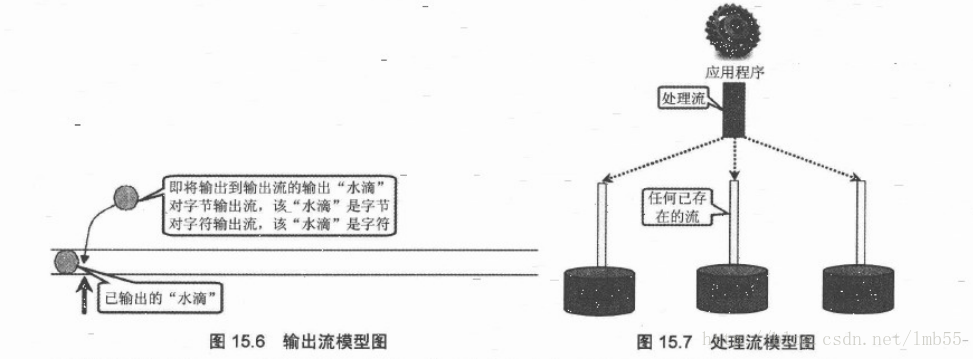
**目的**

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。

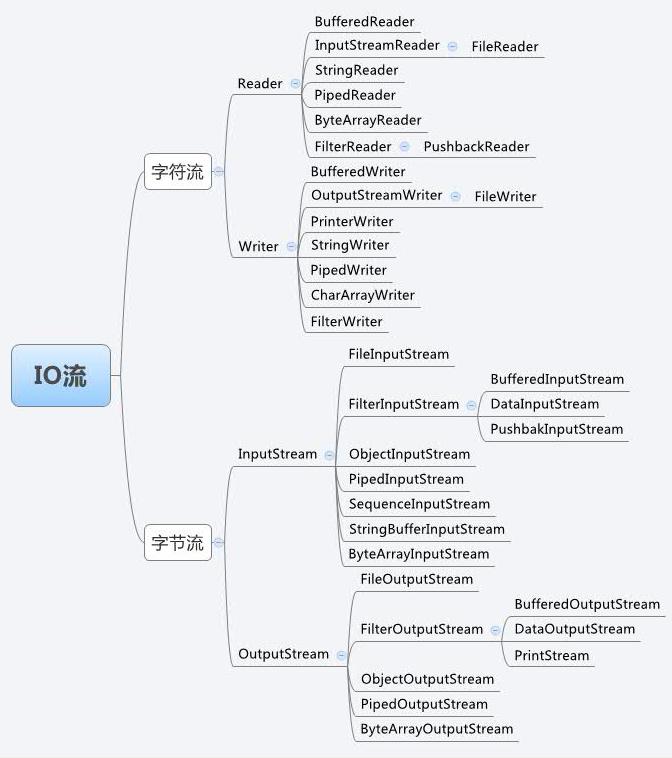
1. 格式化/解析：在内部数据表示（以字节为单位）与外部数据表示（以字符为单位）之间进行双向转换。例如一个2字节的整数10002，就需要5个字符来表示。
2. 缓冲：用于在格式/解析与传递之间缓存字符序列。对于输出，较短的字符序列格式化之后并不马上输出，而是保存在缓冲区里，待累积到一定规模之后再传递到外部设备。相反，从外部设备读入的大量数据也是先放在缓冲区，然后逐步取出完成输入。默认时，IO流的输入输出都是经过缓冲的，也可以让IO流工作在无缓冲模式下。
3. 编码转换： 将一种字符表达式转换成另一种字符表达式。如果格式化产生的字符表达式与外部字符表达式不同（输出时），或者外部表达式与IO流能解析的表达式不同（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**

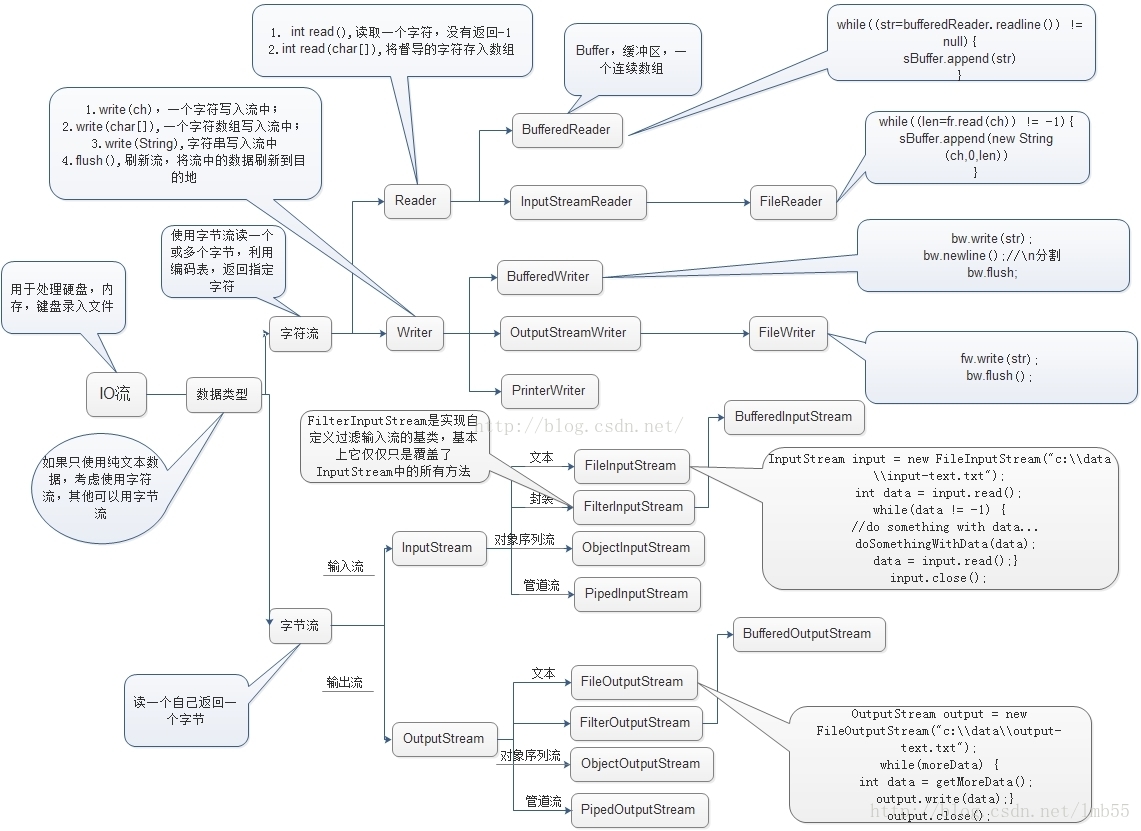
1. 确定源和目标文件
2. 构建流对象
3. 读取内容
4. 写出内容
5. 关闭流

**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  

### PrintStream

简介

PrintStream继承了FilterOutputStream.是"装饰类"的一种,所以属于字节流体系中(与PrintStream相似的流PrintWriter继承于Writer,属于字符流体系中),为其他的输出流添加功能.使它们能够方便打印各种数据值的表示形式.此外,值得注意的是:

* 与其他流不同的是,PrintStream流永远不会抛出异常.因为做了try{}catch(){}会将异常捕获,出现异常情况会在内部设置标识,通过checkError()获取此标识.
* PrintStream流有自动刷新机制,例如当向PrintStream流中写入一个字节数组后自动调用flush()方法.

PrintStream流打印的字符通过平台默认的编码方式转换成字节,在写入的是字符,而不是字节的情况下,应该使用PrintWriter.PrintStream流中基本所有的print(Object obj)重载方法和println(Object obj)重载方法都是通过将对应数据先转换成字符串,然后调用write()方法写到底层输出流中.常见用到PrintStream流:System.out就被包装成PrintStream流,System.err也是PrintStream流,注意System.in不是PrintStream,是没有包装过的OutputStream.所以System.in不能直接使用.

PrintStream流不是直接将数据写到文件的流,需要传入底层输出流out,而且要实现指定编码方式,需要中间流OutputStreamWriter,OutputStreamWriter流实现了字符流以指定编码方式转换成字节流.此外为了提高写入文件的效率,使用到了字符缓冲流BufferWriter.写入PrintStream流的数据怎么写到文件中.需要先了解一下数据读取和写入的流程.

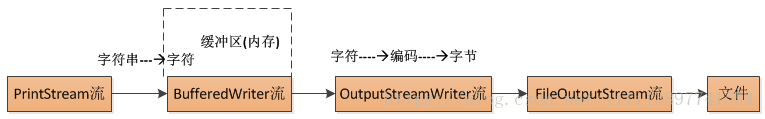
1.数据从流写到文件过程

   输出流----->缓冲流----->转化流----->文件流------>文件.

2.数据从文件到流的过程

  文件----->文件流----->转化流----->缓冲流----->输入流.

那么从PrintStream流写到文件的过程是:



System.out.println(..)

void print（...）打印，一般用户自定义的PrintStream会将数据打印到构造是传入的文件中,System.out则是由jvm通过PrintStream私有构造器创建出来的特定打印流，会将数据展示在jvm的控制台，idea控制台数据展示就是从jvm控制台copy来的，原理大概是jvm实例创建时创建特定的文件地址，再将文件内容输出到jvm的控制台，也可能打印目的地就是jvm的控制台，和公共构造方法略有差别主要体现在数据传输的目的地不一样。日志框架就获取PrintStream ps=System.out流打印到控制台，再更具用户配置写入到指定的日志文件中，比简单的System.out.println(..)多了文件记录。

### OutputStreamWriter

API说明：OutputStreamWriter是从字符流到字节流的桥接：使用指定的字符集将写入其中的字符编码为字节。它使用的字符集可以通过名称指定，也可以明确指定，或者可以接受平台的默认字符集。

1、字符的输出需要通过字符流来操作，但是本质最后还是通过字节流输出到计算机上进行存储的

2、因此OutputStreamWriter流的作用就是利用字节流作为底层输出流然后构建字符输出流，字符输出流输出字符到流中，然后通过指定的字符集把流中的字符编码成字节输出到字节流中，其作用就是一个桥梁，使得双方链接起来

OutputStreamWriter流的API和构造方法本质全部是调用StreamEncoder流的方法，因此需要了解OutputStreamWriter流的具体实现需要查看StreamEncoder流源码才可知.

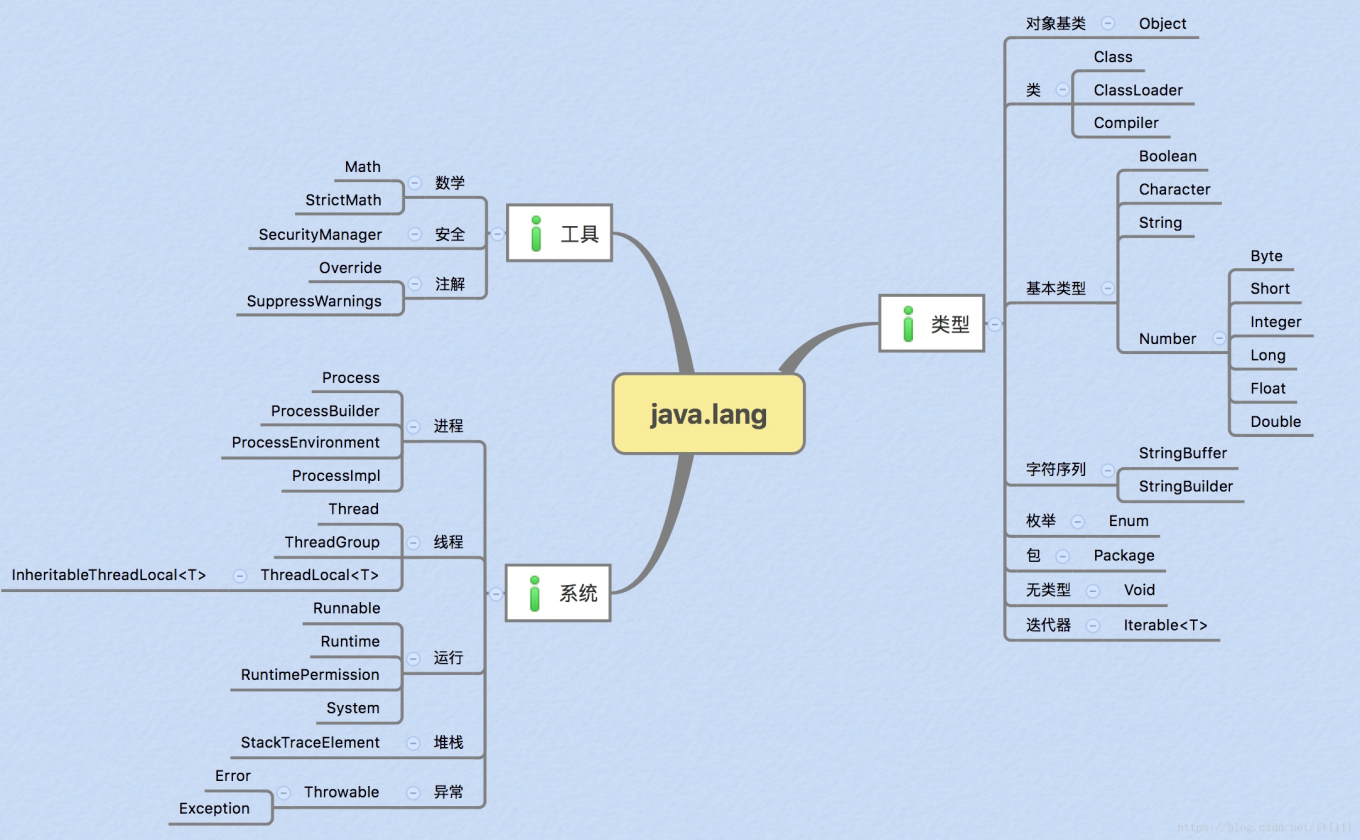
如log4j输出日志到控制台的原理，本质还是System.out.print()一样

|  |
| --- |
| OutputStreamWriter osw=new OutputStreamWriter(System.out); osw.write("sss"); osw.flush(); |

## 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

### Object

*/\*\*  
 \* 所有类的根，默认都是Object类的子类，包括数组。  
 \*/***public class** Object {  
 **private static native void** registerNatives();  
  
 **static** {  
 *registerNatives*();  
 }  
  
 */\*\*  
 \* 获取运行时对象的Class，对Class类的理解详见Class  
 \*/* **public final native** Class<?> getClass();  
  
 */\*\*  
 \* 返回对象的哈希码值,hash算法的一种实现，哈希表详情见数据结构之哈希表  
 \* 总契约是:  
 \* 1.在同一次的java程序应用过程中，对应同样的对象多次调用hashCode方法，hashCode方法必须一致性的返回同样的一个地址值，前提是这个对象不能改变  
 \* 2.两个对象相同是依据equals方法来的，那么其中的每一个对象调用hashCode方法都必须返回相同的一个integer值，也就是对象的地址。equals方法相等，那么hashCode方法也必须相等。  
 \* 3.如果两个对象依据equals方法返回的结果不相等，那么对于其中的每一个对象调用hashCode方法返回的结果也不是一定必须得相等（也就是说，equals方法的结果为false，那么hashCode方法返回的结果可以相同也可以不相同），但是，对于我们开发者来说，针对两个对象的不相等如果生成相同的hashCode则可以提高应用程序的性能即减少hash碰撞的几率。  
 \*/* **public native int** hashCode();  
  
 */\*\*  
 \* 表示其他对象是否“等于”这个对象。  
 \* 自反性 ： x.equals(x) 结果应该返回true。  
 \* 对称性 ： x.equals(y) 结果返回true当且仅当y.equals(x)也应该返回true。  
 \* 传递性 ： x.equals(y) 返回true，并且y.equals(z) 返回true，那么x.equals(z) 也应该返回true。  
 \* 一致性 ： x.equals(y)的第一次调用为true，那么x.equals(y)的第二次，第三次等多次调用也应该为true，但是前提条件是在进行比较之前，x和y都没有被修改。  
 \* x.equals(null) 应该返回false。  
 \* 这个方法返回true当且仅当x和y指向了同样的对象(x==y)，这句话也就是说明了在默认情况下，Object类中的equals方法默认比较的是对象的地址，  
 \* 因为只有是相同的地址才会相等(x == y)，如果没有重写equals方法，那么默认就是比较的是地址。  
 \* 注意：无论何时这个equals方法被重写那么都是有必要去重写hashCode方法，这个是因为为了维持hashCode的一种约定，相同的对象必须要有相同的hashCode值。  
 \*/* **public boolean** equals(Object1 obj) {  
 **return** (**this** == obj);  
 }  
  
 */\*\*  
 \* 返回对象的浅复制对象,调用此方法的类必须实现Cloneable接口重写clone方法,因为Object的clone是protected修饰的  
 \* 知识点：  
 \* 浅复制：复制一个新对象实例，但对象里的对象不是新的实例，所以如果改变了新实例里对象的属性，那么之前的对象里的对象也被改变，如spring框架中 BeanUtils.copyProperties(source,target)  
 \* 就是浅复制应用，实际开发中应注意修改source里的对象时对target的影响。  
 \* 深复制：复制一个完全的新实例，对象里的对象也是新实例，可用通过序列化、备份等等具体实现  
 \*/* **protected native** Object clone() **throws** CloneNotSupportedException;  
  
 */\*\*  
 \* 返回对象的字符串表示形式,System.out.println(obj)方法内部是print(String.valueOf(obj))，valueOf(Object obj) {return (obj == null) ? "null" :obj.toString();},所以打印对象即是打印对象的字符串形式，一般对象都重写了toString方法  
 \*/* **public** String toString() {  
 **return** getClass().getName() + **"@"** + Integer.*toHexString*(hashCode());  
 }  
  
 */\*\*  
 \* obj.notify()随即唤醒一个需要obj对象锁的线程,如果是多个线程则会根据线程的优先权等判断先后顺序  
 \*/* **public final native void** notify();  
  
 */\*\*  
 \* obj.notifyAll()随即唤醒所有需要obj对象锁的线程  
 \*/* **public final native void** notifyAll();  
  
 */\*\*  
 \* 导致当前线程等待，直到另一个线程调用该对象的notify()方法notifyAll()方法，或者经过指定的时间量。  
 \* 注意点：  
 \* 1.是当前的线程等待调用该方法的对象的锁  
 \* public class WaitTest {  
 \* public static void main(String[] args) {  
 \* ThreadA t1 = new ThreadA("t1");  
 \* synchronized(t1) {  
 \* try {  
 \* // 启动“线程t1”  
 \* System.out.println(Thread.currentThread().getName()+" start t1");  
 \* t1.start();  
 \* // 主线程等待t1通过notify()唤醒。  
 \* System.out.println(Thread.currentThread().getName()+" wait()");  
 \* t1.wait(); // 不是使t1线程等待，而是当前执行wait的线程等待即main主线程等待，t1就是一个对象，maim线程等待t1对象的锁  
 \* System.out.println(Thread.currentThread().getName()+" continue");  
 \* } catch (InterruptedException e) {  
 \* e.printStackTrace();  
 \* }  
 \* }  
 \* }  
 \* }  
 \* 2.wait等待释放锁，所以前提是当前线程获取了锁，所以要在synchronized里调用，不然会报错，如下  
 \* public class MainTest {  
 \* public static void main(String[] args) throws InterruptedException {  
 \* Object obj=new Object();  
 \* obj.wait();  
 \* }  
 \* }  
 \* 会报错java.lang.IllegalMonitorStateException指示线程试图在对象的监视器上等待，或通知其他线程在不拥有指定监视器的情况下等待对象的监视器  
 \* 正确的示例  
 \* public class MainTest {  
 \* public static void main(String[] args) throws InterruptedException {  
 \* Object obj=new Object();  
 \* //获取obj的锁，然后在里面才能释放，当然在此之前应该开启一个线程执行唤醒操作，不然这个线程一直在等待状态  
 \* synchronized (obj){  
 \* obj.wait();  
 \* }  
 \* }  
 \* }  
 \*/* **public final native void** wait(**long** timeout) **throws** InterruptedException;  
  
 */\*\*  
 \*导致当前线程等待，直到另一个线程调用该对象的notify()、notifyAll()方法。这个方法的行为与它简单地执行调用wait(0)完全一样。  
 \*/* **public final void** wait() **throws** InterruptedException {  
 wait(0);  
 }  
  
 */\*\*  
 \* 当垃圾收集确定没有对对象的引用时，由对象上的垃圾收集器调用。子类重写 finalize方法配置系统资源或执行其他清理。  
 \*/* **protected void** finalize() **throws** Throwable {  
 }  
}

### Thread

*/\*\*  
 \* 在java中一个Thread实例就是一个程序执行的线程，java虚拟机允许一个应用程序拥有多个并发运行的执行线程。  
 \* 每一个线程都有一个优先权。具有更高优先级的线程在优先于较低优先级的线程中执行。每一个线程可能会或可能不会被标记为一个守护进程。  
 \* 在一些线程中运行的代码创建了一个新的Thread对象时，新线程的优先级被设置为等于创建线程的优先级，并且只有在创建线程是守护进程时才为守护进程。  
 \* 当一个java虚拟机启动时，通常有一个单一的非守护线程（通常调用方法命名为main某指定的类）。java虚拟机继续执行线程，直到发生以下情况：  
 \* 1.Runtime的exit方法被调用和安全经理允许退出操作发生。  
 \* 2.非守护线程的所有线程都已经死了，要么通过从调用返回run方法，要么通过抛出一个传播到run方法之外的异常。  
 \* 有两种创建线程的方法，不同创建方式只是重写run方法的不同方式，也就是创建Thread类中target方式不同。  
 \* 1.继承Thread类重写run方法，创建该实例(也实现了Runnable接口)。  
 \* 2.实现Runable接口重写run方法，创建该实例作为创建Thread构造参数。  
 \* <p>*

*设计模式——生产消费者模式  
 \* <p>  
 \* 以下只是部分源码  
 \*/***public class** Threada **implements** Runnable {  
  
 */\* 该线程是否为守护线程.必须在开启前设置好 \*/* **private boolean daemon** = **false**;  
  
 */\*将运行的，真正的处理部分 \*/* **private** Runnable **target**;  
  
 */\* 这个线程的组\*/* **private** ThreadGroup **group**;  
  
 */\*与此线程相关的ThreadLocal值。这map通过ThreadLocal类维护.详见ThreadLocal \*/* ThreadLocal.ThreadLocalMap **threadLocals** = **null**;  
  
 */\*与这个线程相关的InheritableThreadLocal值。这map是由InheritableThreadLocal类维护。  
 \*/* ThreadLocal.ThreadLocalMap **inheritableThreadLocals** = **null**;  
  
 */\*工具的Java线程状态，初始化，表示线程“尚未启动”\*/* **private volatile int threadStatus** = 0;  
  
 */\*对象锁\*/* **private final** Object **blockerLock** = **new** Object();  
  
 */\*\*  
 \* 线程可以拥有的最小优先级。  
 \*/* **public final static int *MIN\_PRIORITY*** = 1;  
  
 */\*\*  
 \* 分配给线程的默认优先级。  
 \*/* **public final static int *NORM\_PRIORITY*** = 5;  
  
 */\*\*  
 \* 线程可以拥有的最大优先级。  
 \*/* **public final static int *MAX\_PRIORITY*** = 10;  
  
  
 */\*\*  
 \* 返回当前执行的线程对象的引用。  
 \*/* **public static native** Thread currentThread();  
  
 */\*\*  
 \*  
 \*/* **public static native void** yield();  
  
 */\*\*  
 \* 导致当前正在执行的线程处于休眠状态(临时停止) 执行)，以指定的毫秒数为准 系统定时器和调度器的精度和准确性。线程不会失去任何锁的所有权。  
 \*/* **public static native void** sleep(**long** millis) **throws** InterruptedException;  
  
 */\*\*  
 \* 初始化一个线程.  
 \*/* **private void** init(ThreadGroup g, Runnable target, String name,  
 **long** stackSize, AccessControlContext acc,  
 **boolean** inheritThreadLocals) {  
 **if** (name == **null**) {  
 **throw new** NullPointerException(**"name cannot be null"**);  
 }  
  
 **this**.name = name;  
  
 Thread parent = *currentThread*();  
 SecurityManager security = System.*getSecurityManager*();  
 **if** (g == **null**) {  
 */\*确定它是否是applet \*/  
  
 /\* If there is a security manager, ask the security manager what to do. \*/* **if** (security != **null**) {  
 g = security.getThreadGroup();  
 }  
  
 */\* 如果安全性对这个问题没有很强的概念，那么使用父线程组. \*/* **if** (g == **null**) {  
 g = parent.getThreadGroup();  
 }  
 }  
  
 */\* checkAccess，不管是否显式地传入threadgroup. \*/* g.checkAccess();  
  
 */\*  
 \* Do we have the required permissions?  
 \*/* **if** (security != **null**) {  
 **if** (isCCLOverridden(getClass())) {  
 security.checkPermission(SUBCLASS\_IMPLEMENTATION\_PERMISSION);  
 }  
 }  
  
 g.addUnstarted();  
  
 **this**.**group** = g;  
 **this**.**daemon** = parent.isDaemon();  
 **this**.priority = parent.getPriority();  
 **if** (security == **null** || isCCLOverridden(parent.getClass()))  
 **this**.contextClassLoader = parent.getContextClassLoader();  
 **else  
 this**.contextClassLoader = parent.**contextClassLoader**;  
 **this**.inheritedAccessControlContext =  
 acc != **null** ? acc : AccessController.getContext();  
 **this**.**target** = target;  
 setPriority(priority);  
 **if** (inheritThreadLocals && parent.**inheritableThreadLocals** != **null**)  
 **this**.**inheritableThreadLocals** =  
 ThreadLocal.*createInheritedMap*(parent.**inheritableThreadLocals**);  
 */\* Stash the specified stack size in case the VM cares \*/* **this**.stackSize = stackSize;  
  
 */\* Set thread ID \*/* tid = nextThreadID();  
 }  
  
 */\*\*  
 \* 重写clone方法，抛出CloneNotSupportedException作为一个线程不能有意义克隆。相反，构造一个新线程。  
 \*/* @Override  
 **protected** Object clone() **throws** CloneNotSupportedException {  
 **throw new** CloneNotSupportedException();  
 }  
  
 */\* 最常用的的2中构造器之一 \*/* **public** Thread() {  
 init(**null**, **null**, **"Thread-"** + nextThreadNum(), 0);  
 }  
  
 */\* 最常用的的2中构造器之一 \*/* **public** Thread(Runnable target) {  
 init(**null**, target, **"Thread-"** + nextThreadNum(), 0);  
 }  
  
 */\*\*  
 \* 使该线程开始执行;Java虚拟机\*调用该线程的run方法。结果是两个线程并发运行:当前线程(从调用\* <code>start</code>方法返回)和  
 \* 另一个线程(执行其\* <code>run</code>方法)。\* <p> \*启动一个线程超过一次是不合法的。\*特别地，一个线程在完成\*执行后可能不会重新启动。  
 \*/* **public synchronized void** start() {  
 */\*\*  
 \*零状态值对应于状态“NEW”.  
 \*/* **if** (**threadStatus** != 0)  
 **throw new** IllegalThreadStateException();  
  
 */\* 通知组此线程即将启动以便可以将其添加到组的线程列表中该组的未开始计数可以减少. \*/* **group**.add(**this**);  
  
 **boolean** started = **false**;  
 **try** {  
 start0();  
 started = **true**;  
 } **finally** {  
 **try** {  
 **if** (!started) {  
 **group**.threadStartFailed(**this**);  
 }  
 } **catch** (Throwable ignore) {  
 */\* 什么也不做。如果start0抛出一个投掷物它将被传递到调用堆栈 \*/* }  
 }  
 }  
  
 */\*\*  
 \* 调用strat方法后最终调用本地方法start0，最后jvm会调用run方法  
 \*/* **private native void** start0();  
  
 */\*\*  
 \* 真正的业务处理都放在target的run方法中  
 \*/* @Override  
 **public void** run() {  
 **if** (**target** != **null**) {  
 **target**.run();  
 }  
 }  
  
 */\*\*  
 \* 系统调用这个方法来给一个线程 在它真正离开之前清理的机会。  
 \*/* **private void** exit() {  
 **if** (**group** != **null**) {  
 **group**.threadTerminated(**this**);  
 **group** = **null**;  
 }  
 */\* Aggressively null out all reference fields: see bug 4006245 \*/* **target** = **null**;  
 */\* Speed the release of some of these resources \*/* **threadLocals** = **null**;  
 **inheritableThreadLocals** = **null**;  
 inheritedAccessControlContext = **null**;  
 blocker = **null**;  
 uncaughtExceptionHandler = **null**;  
 }  
  
  
 */\*\*  
 \* 假如在main线程中，调用thread.join方法，则main方法会等待thread线程执行完毕或者等待一定的时间。如果调用的是无参join方法，  
 \* 则等待thread执行完毕，如果调用的是指定了时间参数的join方法，则等待一定的时间，内部由wait方法实现，  
 \* 由于wait方法会让线程释放对象锁，所以join方法同样会让线程释放对一个对象持有的锁  
 \*/* **public final synchronized void** join(**long** millis)  
 **throws** InterruptedException {  
 **long** base = System.*currentTimeMillis*();  
 **long** now = 0;  
  
 **if** (millis < 0) {  
 **throw new** IllegalArgumentException(**"timeout value is negative"**);  
 }  
  
 **if** (millis == 0) {  
 **while** (isAlive()) {  
 wait(0);  
 }  
 } **else** {  
 **while** (isAlive()) {  
 **long** delay = millis - now;  
 **if** (delay <= 0) {  
 **break**;  
 }  
 wait(delay);  
 now = System.*currentTimeMillis*() - base;  
 }  
 }  
 }  
  
 **public enum** State {  
 */\*\*  
 \* 尚未启动的线程的线程状态.  
 \*/* ***NEW***,  
  
 */\*\*  
 \* 可运行线程的线程状态。一个线程在运行 状态是执行在Java虚拟机，但它可以等待操作系统的其他资源 例如处理器。  
 \*/* ***RUNNABLE***,  
  
 */\*\*  
 \* 等待监视器锁的阻塞线程的线程状态。 处于阻塞状态的线程正在等待监视器锁输入一个同步块/方法或 调用后重新输入同步块/方法\* Object.wait.  
 \*/* ***BLOCKED***,  
  
 */\*\*  
 \* 等待线程的线程状态。 一个线程由于调用其中一个而处于等待状态 以下方法:  
 \* <ul>  
 \* <li>{****@link*** *Object#wait() Object.wait} with no timeout</li>  
 \* <li>{****@link*** *#join() Thread.join} with no timeout</li>  
 \* <li>{****@link*** *LockSupport#park() LockSupport.park}</li>  
 \* </ul>  
 \*/* ***WAITING***,  
  
 */\*\*  
 \* 具有指定等待时间的等待线程的线程状态。 一个线程由于调用其中一个而处于定时等待状态 下列指定正等待时间的方法:  
 \* <ul>  
 \* <li>{****@link*** *#sleep Thread.sleep}</li>  
 \* <li>{****@link*** *Object#wait(long) Object.wait} with timeout</li>  
 \* <li>{****@link*** *#join(long) Thread.join} with timeout</li>  
 \* <li>{****@link*** *LockSupport#parkNanos LockSupport.parkNanos}</li>  
 \* <li>{****@link*** *LockSupport#parkUntil LockSupport.parkUntil}</li>  
 \* </ul>  
 \*/* ***TIMED\_WAITING***,  
  
 */\*\*  
 \* 终止线程的线程状态。 线程已经完成执行。  
 \*/* ***TERMINATED***;  
 }  
  
 */\*\*  
 \* 返回此线程的状态。\*本方法用于监控系统状态，\*不用于同步控制。  
 \*/* **public** State getState() {  
 *// get current thread state* **return** sun.misc.VM.*toThreadState*(**threadStatus**);  
 }  
  
 */\*\*  
 \* 当一个线程终止由于未捕获的异常\*Java虚拟机将查询线程的UncaughtExceptionHandler使用getUncaughtExceptionHandler,  
 \* 将调用处理程序的uncaughtException方法,通过线程和异常作为参数。\*如果一个线程没有显式地设置它的UncaughtExceptionHandler，  
 \* 那么它的ThreadGroup对象充当它的UncaughtExceptionHandler如果<tt>ThreadGroup</tt>对象没\*处理异常的特殊要求，  
 \* 则可以将\*调用转发给getDefaultUncaughtExceptionHandler 默认未捕获异常处理程序。  
 \*  
 \* 可以自定义运行时异常处理器如下  
 \* public class MainTest {  
 \* public static void main(String[] args) {  
 \* Thread.setDefaultUncaughtExceptionHandler(new MyUncaughtExceptionHandler());  
 \* int a = 10 / 0;  
 \* }  
 \* }  
 \* class MyUncaughtExceptionHandler implements Thread.UncaughtExceptionHandler {  
 \** ***@Override*** *\* public void uncaughtException(Thread t, Throwable e) {  
 \* System.out.println("发生了运行时异常");  
 \* }  
 \* }  
 \* 运行结果为：  
 \* "C:\Program Files\Java\jdk1.8.0\_191\bin\java.exe" "-javaagent:C:\Program Files\idea2018.3\lib\idea\_rt.jar=5990......  
 \* \* 发生了运行时异常  
 \*  
 \* Process finished with exit code 1  
 \*  
 \*这就是当发生异常时控制台打印信息的原理  
 \*  
 \*/* @FunctionalInterface  
 **public interface** UncaughtExceptionHandler {  
 */\*\*  
 \* 1.当程序发生运行时异常时调用此方法打印异常信息并终止，如int a=10/0发生异常，jvm会调用此方法。  
 \* 2.此方法里如再发生异常那么jvm会忽略。  
 \*/* **void** uncaughtException(Thread t, Throwable e);  
 }  
}

### ThreadLocal

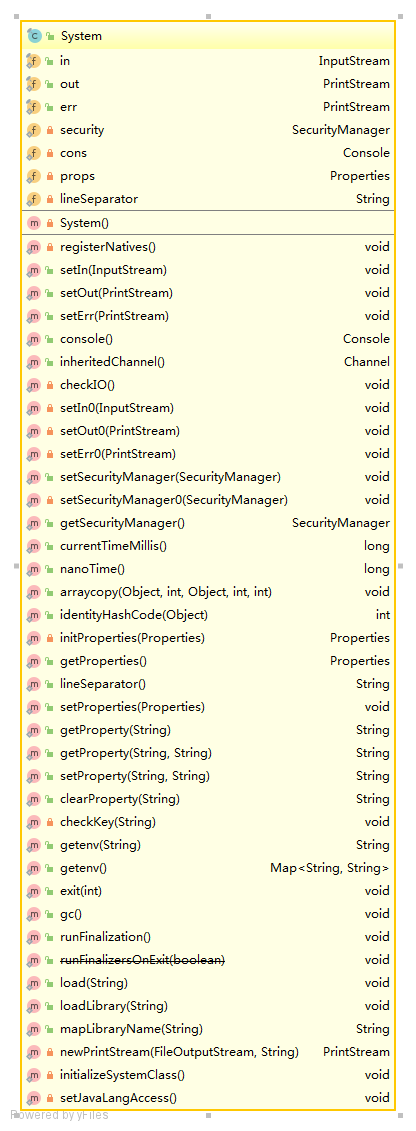
*/\*\*  
 \* 该类提供线程局部变量。这些变量与普通的变量不同，因为每个访问一个变量的线程get或set方法都有自己独立初始化的变量副本。  
 \* ThreadLocal实例通常是希望将状态与线程例如用户ID或事务ID关联的类中的私有静态字段。目的就是让各个线程有自己的变量。  
 \* 如下示例  
 \* public class MainTest {  
 \* public static void main(String[] args) {  
 \* Bank bank = new Bank();  
 \* MyThread myThread = new MyThread(bank);  
 \* new Thread(myThread).start();  
 \* new Thread(myThread).start();  
 \* <p>  
 \* }  
 \* }  
 \* <p>  
 \* class MyThread implements Runnable {  
 \* private Bank bank;  
 \* <p>  
 \* public MyThread(Bank bank) {  
 \* this.bank = bank;  
 \* }  
 \*  
 \** ***@Override*** *public void run() {  
 \* for (int i = 0; i < 10; i++) {  
 \* bank.set(bank.get() + 1);  
 \* System.out.println(Thread.currentThread().getName() + ":" + bank.get());  
 \* }  
 \* }  
 \* }  
 \* <p>  
 \* class Bank {  
 \* ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 10);  
 \* <p>  
 \* public void set(Integer t) {  
 \* threadLocal.set(t);  
 \* }  
 \* <p>  
 \* public Integer get() {  
 \* return threadLocal.get();  
 \* }  
 \* }  
 \* 虽然不同线程共享一个bank变量，但是其中bank的get和set方法调用的是ThreadLocal的get和set方法，只与线程有关，所以每个线程的得到的值  
 \* 不会互相影响，原理是ThreadLocal中有个map，map.put("当前线程"，值)详情查看源码。*

*\**

*\* 项目应用如spring security中保存用户信息，每个请求即一个线程进入应用，都会根据session等相关值封装成authentication保存到SecurityContext，再将  
 \* SecurityContext保存到SecurityContextHolder中的，而SecurityContextHolder的实现类有有一个Threadlocal保存SecurityContext。  
 \*/***public class** ThreadLocal<T> {  
  
 */\*\*  
 \* thread局部变量依赖于每个线程附加的线性探测散列映射。和\*inheritablethreadlocality。ThreadLocal对象充当键，通过threadLocalHashCode  
 \* 搜索.这是一个自定义哈希码\*(仅在ThreadLocalMaps中有用)，它消除了冲突，通常情况下，相同的线程使用连续构造的threadlocality ，  
 \* 而在不太常见的情况下仍然保持良好的行为。  
 \*/* **private final int threadLocalHashCode** = nextHashCode();  
  
 */\*\*  
 \* 为这个\*线程局部变量返回当前线程的“初始值”  
 \*/* **protected** T initialValue() {  
 **return null**;  
 }  
  
 */\*\*  
 \* 返回当前线程的\*线程局部变量副本中的值。如果变量没有\*当前线程的值，则首先通过调用{****@link*** *#initialValue}方法将其初始化为返回的值\*。  
 \*/* **public** T get() {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**) {  
 ThreadLocalMap.Entry e = map.getEntry(**this**);  
 **if** (e != **null**) {  
 @SuppressWarnings(**"unchecked"**)  
 T result = (T) e.value;  
 **return** result;  
 }  
 }  
 **return** setInitialValue();  
 }  
  
 */\*\*  
 \* set()的变体以建立initialValue,如果用户覆盖了set()方法，则使用\*代替set()  
 \*/* **private** T setInitialValue() {  
 T value = initialValue();  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**)  
 map.set(**this**, value);  
 **else** createMap(t, value);  
 **return** value;  
 }  
  
 */\*\*  
 \* 将当前线程的该线程局部变量\*的副本设置为指定的值。大多数子类将不需要\*覆盖这个方法，仅仅依靠initialValue方法来设置线程局部变量的值。  
 \*/* **public void** set(T value) {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 **if** (map != **null**)  
 map.set(**this**, value);  
 **else** createMap(t, value);  
 }  
  
 */\*\*  
 \* 删除此线程本地\*变量的当前线程值,一般来说一次使用完之后，要把相应的值去除掉，防止内存泄漏  
 \*/* **public void** remove() {  
 ThreadLocalMap m = getMap(Thread.*currentThread*());  
 **if** (m != **null**)  
 m.remove(**this**);  
 }  
  
 */\*\*  
 \* 定制的hash map实现，Threadlocal实现线程封闭的关键，每个线程都有一个ThreadLocalMap 存放自己的变量，详情查看源码  
 \*/* **static class** ThreadLocalMap {  
 **private** ThreadLocalMap(ThreadLocalMap parentMap) {  
 Entry[] parentTable = parentMap.table;  
 **int** len = parentTable.**length**;  
 setThreshold(len);  
 table = **new** Entry[len];  
  
 **for** (**int** j = 0; j < len; j++) {  
 Entry e = parentTable[j];  
 **if** (e != **null**) {  
 @SuppressWarnings(**"unchecked"**)  
 ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();  
 **if** (key != **null**) {  
 Object value = key.childValue(e.value);  
 Entry c = **new** Entry(key, value);  
 **int** h = key.**threadLocalHashCode** & (len - 1);  
 **while** (table[h] != **null**)  
 h = nextIndex(h, len);  
 table[h] = c;  
 size++;  
 }  
 }  
 }  
 }  
 }  
}

### System

包含几个有用的类字段和方法。它不能被实例化。提供的功能有标准输入、标准输出和错误输出流;访问外部定义的属性和环境变量;加载文件和库的一种方法;以及一个实用程序方法，用于快速复制数组的一部分。



部分方法说明：

* public final static PrintStream out = null;默认的打印流输出流是打印到控制台，可以通过public static void setOut(PrintStream out) {}方法输出到指定的文件
* public static Properties getProperties() {}获取系统相关属性
* public static void exit(int status)0为关闭jvm实例，一般程序结束后会自动关闭Process finished with exit code 0
* public static void gc() {Runtime.getRuntime().gc();}运行垃圾收集器
* public static Console console(){}获取控制台实例如果Java程序要与windows下的cmd或者Linux下的Terminal交互，就可以使用这个Java Console类代劳。Java要与Console进行交互，不总是能得到可用的Java Console类的。一个JVM是否有可用的Console，依赖于底层平台和JVM如何被调用。如果JVM是在交互式命令行（比如Windows的cmd）中启动的，并且输入输出没有重定向到另外的地方，那么就可以得到一个可用的Console实例。但当使用Eclipse等IDE运行以上代码时Console中将会为null表示Java程序无法获得Console实例，是因为JVM不是在命令行中被调用的，或者输入输出被重定向了,如下代码在cmd中执行和在idea中执行结果是不同的。



## math

### package.html

java.math包提供了java中的数学类,包括基本的浮点库、复杂运算以及任意精度的数据运算

## net

### package.html

java网络编程的api

## 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。

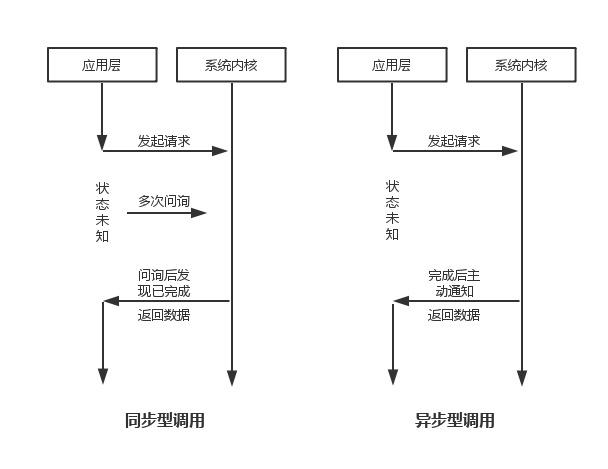
对于通知调用者的三种方式

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。

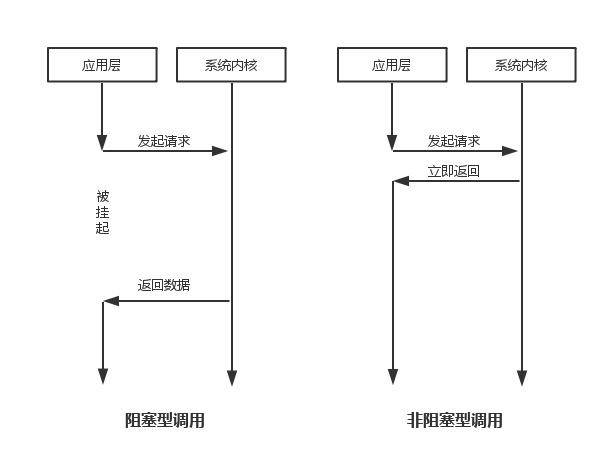


**阻塞**

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。

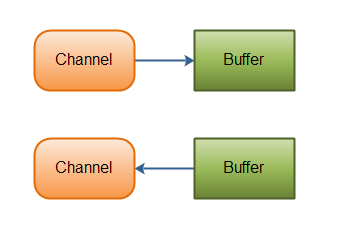
**IO（BIO）和NIO核心区别**

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

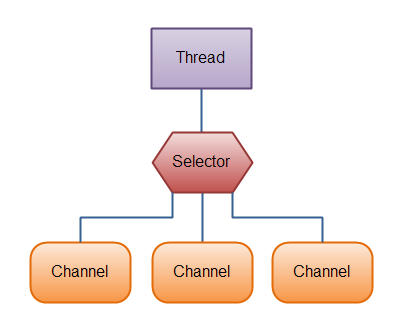
**java.nio**

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

**简介**

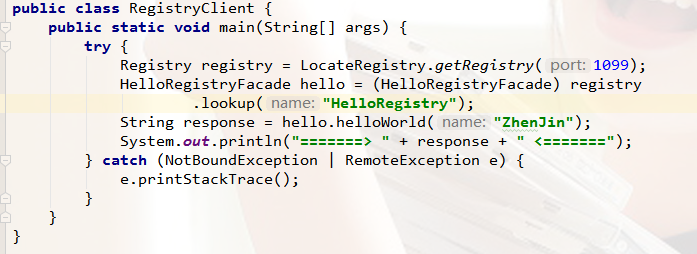
Java RMI，即 远程方法调用([Remote Method Invocation](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Distributed_object_communication" \t "https://www.jianshu.com/p/_blank))，一种用于实现远程过程调用(RPC)[(Remote procedure call)](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Remote_procedure_call" \t "https://www.jianshu.com/p/_blank)的Java API， 能直接传输序列化后的Java对象和[分布式垃圾收集](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Distributed_Garbage_Collection" \t "https://www.jianshu.com/p/_blank)。它的实现依赖于[Java虚拟机](https://links.jianshu.com/go?to=https://en.wikipedia.org/wiki/Java_Virtual_Machine" \t "https://www.jianshu.com/p/_blank)(JVM)，因此它仅支持从一个JVM到另一个JVM的调用。

**api简单示例**

服务端



客户端



**核心原理**

1. Registry registry = LocateRegistry.createRegistry(1099)底层源码知会开启一个socket并在程序中创建一个map
2. registry.rebind("HelloRegistry", hello)将对象放进map
3. HelloRegistryFacade hello = (HelloRegistryFacade) registry.lookup("HelloRegistry")客户端代码连接socket获取map中指定对象

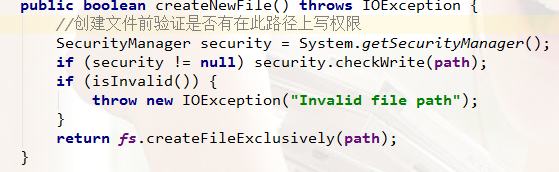
## security

### package.html

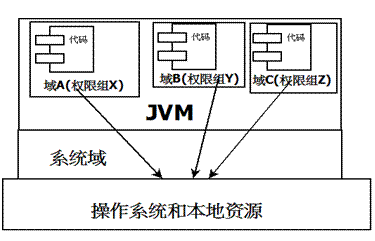
Java SDK 中与安全相关的类和接口都放在 java.security 包中，其中既包括访问控制配置及细粒度访问控制框架的实现，还包括签名和加解密的实现

**一、访问控制**

对一些敏感操作加上访问控制，判断是否有操作（read、write..）权限,通过Policy策略文件定义控制，如在磁盘上创建文件操作api



**安全模型**



**二、[Java加密框架](https://www.cnblogs.com/f1194361820/p/4262507.html)（JCA）**

**公钥和死钥**

相对而言，公钥加密则私钥解密，私钥加密公钥解密，私钥不可公开，公钥可公开。

**可逆加密**

1.对称加密:对称加密是最快速、最简单的一种加密方式，加密（encryption）与解密（decryption）用的是同样的密钥（secret key）,这种方法在密码学中叫做对称加密算法如AES、Base64。

2.非对称加密:非对称加密为数据的加密与解密提供了一个非常安全的方法，它使用了一对密钥，公钥（public key）和私钥（private key）。私钥只能由一方安全保管，不能外泄，而公钥则可以发给任何请求它的人。非对称加密使用这对密钥中的一个进行加密，而解密则需要另一个密钥，算法如RSA一般支付系统都用此算法加密

**不可逆的加密**

加密的密文无法再解密称原文，如SHA、MD5都属于hash算法，一般用户密码都用不可逆加密



**摘要**

用来被加密的部分内容，加密后生成签名

**签名**

解密后与发送的原文对比（验签），查看是否和原文一样，防止接口参数被篡改。

**数字证书**

私钥、公钥和其他信息通过证书中心加密生成的文件

**业务项目证书应用示例**

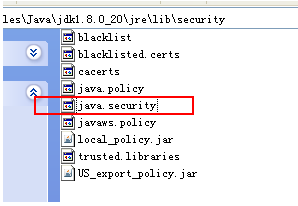
****

**JCA（Java加密框架）核心类介绍**

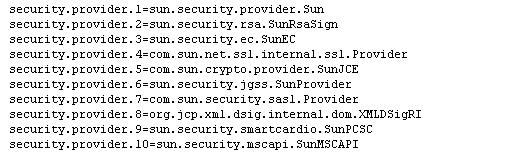
加密服务总是关联到一个特定的算法或类型,它既提供了密码操作(如Digital Signature或MessageDigest),生成或供应所需的加密材料(Key或Parameters)加密操作,也会以一个安全的方式生成数据对象(KeyStore或Certificate),封装(压缩)密钥(可以用于加密操作)。

一个engine class就是定义了一种加密服务，不同的engine class提供不同的服务。下面就来看看有哪些engine class

1. MessageDigest：对消息进行hash算法生成消息摘要（digest）。
2. Signature：对数据进行签名、验证数字签名。
3. KeyPairGenerator：根据指定的算法生成配对的公钥、私钥。
4. KeyFactory：根据Key说明（KeySpec）生成公钥或者私钥。
5. CertificateFactory：创建公钥证书和证书吊销列表（CRLs）。
6. KeyStore：keystore是一个keys的数据库。Keystore中的私钥会有一个相关联的证书链，证书用于鉴定对应的公钥。数字证书转为java对象keystore.load(in)
7. AlgorithmParameters：管理算法参数。KeyPairGenerator就是使用算法参数，进行算法相关的运算，生成KeyPair的。生成Signature时也会用到。
8. AlgorithmParametersGenerator：用于生成AlgorithmParameters。
9. SecureRandom：用于生成随机数或者伪随机数。
10. CertPathBuilder：用于构建证书链。
11. CertPathValidator：用于校验证书链。
12. CertStore：存储、获取证书链、CRLs到（从）CertStore中。
13. Provider**：**engine的实现都离不开Provider，算法的第三方实现，jdk也有自带的实现。每个engine class都有getInstance()方法，它们都是从provider中获取相关实例的。所以说Provider是JCA engine class的提供商，如银联支付用到的证书签名算法用的就是银联提供的指定算法实现包。
14. Security：管理Provider的工具，其实就是一个存放Provider的集合。如果自定义了一个XYZProvider，可以Java Security属性文件配置provider，也可以使用Security采用编程的方式来添加Provider。



已经默认配置了下列Provider：



配置方式：security.provider.11=com.fjn.security.XYZProvider， 编码方式：Security.addProvider(new XYZProvider());

**jdk自带各种加密算法实现示例**



## sql

### package.html

jdbc规范api

## text

### package.html

提供以与自然语言无关的方式来处理文本、日期、数字和消息的类和接口。这意味着所编写的主程序或 applet 是与语言无关的，并且它可以依靠独立的、动态链接的本地化资源。这实现了随时为新本地化添加本地化的灵活性。这些类能够格式化日期、数字和消息，分析，搜索和排序字符串，以及迭代字符、单词、语句和换行符。

注：自然语言即国家语言如汉语、英语，与自然语言无关，如文本有汉字、英文字母，日期date不同国家时区都不一样。

String strDateFormat = "yy年M月d日 晚上h时m分s秒 E yyyyy";

//会解析pattern字符串，转为为char数组，遍历每个字符，判断是是否为对应的y、s、M等，并更具传入的date获取对应的值

## time

### package.html

java.time是jdk8添加的全新的关于时间操作的api，基于ISO\_8601日历系统实现的日期时间库

**产生背景**

在Java SE 8前，日期时间工具库在java.util包中，包括：

* java.util.Date：表示日期和时间
* java.util.Calendar以及其实现子类：表示各种日历系统，常用的是格林威治日历java.util.GregorianCalendar
* java.util.TimeZone以及其实现子类：表示时区偏移量和夏令时

以及辅助其进行格式化和解析的工具库在java.text包中，包括：

* java.text.DateFormat：格式化日期时间和解析日期时间的工具抽象类
* java.text.SimpleDateFormat：DateDateFormat的实现

从以上的简述中，对java 8之前的日期时间库，有所宏观视觉。下面简要总结下其设计上的瑕疵和被开发者无限吐槽的诟病：

1. 从以上的api上看，java 8之前的日期时间工具库缺乏年、月、日、时间、星期的单独抽象；
2. Dater日期时间类既描述日期又描述时间，耦合，且Date不仅在java.util包中存在，在java.sql中也存在，重复名称，容易导致bug发生；
3. api的设计上晦涩，难用，不够生动，难以以自然人类的思维理解日期时间。年月日需要从Calendar中获取。q；
4. 最被开发者抱怨的是类型不安全，Calendar类中全局属性是可变的，在多线程访问时，会存在线程安全问题。SimpleDateFormat格式化和解析日期，需要使用年月日时分秒，所以持有了Calendar属性，导致其也是非线程安全；

不安全的原因



所以在多线程环境中使用Calendar是非线程安全，多个线程修改其属性域会发生数据一致性和可见性问题。在DateFormat中持有了Calendar属性，用于解析和格式化日期

format方法中设置了全局成员Calendar的time，多线程访问时每次都会改变Calendar类，导致format格式化时会出现线程安全问题。所以DateFormat和其子类SimpleDateFormat都是非类型安全。

所以每次在使用日期格式工具时大多数都会重新new或者使用ThreadLocal。基于此诸多问题，java设计者终于在Java SE 8中引入了新的日期时间库

**java.time优点**

1.设计

java.time中使用了大量的设计模式

* 工厂模式：now(）工厂方法直接生成当前日期时间或者瞬时；of()工厂方法根据年月日时分秒生成日期或者日期时间；
* 装饰模式：时区时间ZoneDateTime/便宜时间OffsetDateTime，都在LocalDateTime的基础上加上时区/偏移量的修饰成为时区时间，然后可以进行时区转换；
* 建造者模式：Calendar中加入建造者类，用于生成新的Calendar对象；

2.命名

java 8中的日期时间库类名、方法名命名上都是极其形象生动，易于理解，让开发者极易于使用——语义清晰精确！如：LocalDate中提供的now表示现在的日期，of用于年月日组成的日期（这里和英文中的of意义非常贴切），plus/minus加减等等；

3.合理的接口设计

* LocalDate表示日期，由年月日组成，提供了获取所在年，所在月，所在日的api，提供所在一年的第几天api，用于比较日期前后api，替换年份、月份、日的api，这些api使得日期或者日期时间的处理上得到的功能上的极大提升；
* 抽象出年、月、日、星期、日期、日期时间、瞬时、周期诸多接口，对事物本质有了细腻的抽象，并提供了相互转换的能力——提供极强的处理能力和语言表达能力；
* 对于遗留的日期时间库Calendar/Date/Timezone和新的日期时间库的互通性；
* 将全球的非标准日历系统单独抽象并支持扩展，从标准日历系统中隔离（符合设计原则：对修改关闭，对扩展开放）

Instant：时间戳

Duration：持续时间，时间差

LocalDate：只包含日期，比如：2016-10-20

LocalTime：只包含时间，比如：23:12:10

LocalDateTime：包含日期和时间，比如：2016-10-20 23:14:21

Period：时间段

ZoneOffset：时区偏移量，比如：+8:00

ZonedDateTime：带时区的时间

Clock：时钟，比如获取目前美国纽约的时间

## util

### package.html

包含集合框架、遗留的 collection 类、事件模型、日期和时间设施、国际化和各种实用工具类（字符串标记生成器、随机数生成器和位[数组](https://baike.baidu.com/item/%E6%95%B0%E7%BB%84/3794097" \t "https://baike.baidu.com/item/java.util/_blank)、日期Date类、堆栈Stack类、向量Vector类等）。集合类、时间处理模式、日期时间工具等各类常用工具包

### concurrent

#### package.html

多线程并发编程

基本概念

硬件：[计算机系统](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%B3%BB%E7%BB%9F/7210959" \t "https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6/_blank)中由电子，机械和光电[元件](https://baike.baidu.com/item/%E5%85%83%E4%BB%B6/7390839" \t "https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6/_blank)等组成的各种物理装置的总称，如CPU、内存等，功能是输入并存储[程序](https://baike.baidu.com/item/%E7%A8%8B%E5%BA%8F" \t "https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6/_blank)和数据，以及[执行程序](https://baike.baidu.com/item/%E6%89%A7%E8%A1%8C%E7%A8%8B%E5%BA%8F/4032665" \t "https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6/_blank)把[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE" \t "https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6/_blank)加工成可以利用的形式

软件：即程序是一系列按照特定顺序组织的计算机数据和[指令](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4/3225201" \t "https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6/_blank)的集合，如tomcat、qq音乐。

操作系统：管理[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA" \t "https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/_blank)[硬件](https://baike.baidu.com/item/%E7%A1%AC%E4%BB%B6" \t "https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/_blank)与[软件](https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6" \t "https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/_blank)资源的[计算机程序](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A8%8B%E5%BA%8F" \t "https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/_blank)，如windows系统、linux系统都属于软件中系统软件。

进程：计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位。

线程：是[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)能够进行运算[调度](https://baike.baidu.com/item/%E8%B0%83%E5%BA%A6" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)的最小单位，它被包含在[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)之中，是[进程](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B" \t "https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B/_blank)中的实际运作单位。

多线程：指从软件或者硬件上实现多个线程并发执行的技术。具有多线程能力的计算机因有硬件支持而能够在同一时间执行多于一个线程，进而提升整体处理性能

并发：在[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F/192" \t "https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91/_blank)中，指一个时间段中有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个[处理机](https://baike.baidu.com/item/%E5%A4%84%E7%90%86%E6%9C%BA/128842" \t "https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91/_blank)上运行，但任一个时刻点上只有一个程序在处理机上运行，不停的切换，只是处理速度非常快感觉是同时进行的。

简单认识理解

计算机的核心是CPU，它承担了所有的计算任务，而操作系统是计算机的管理者，它负责任务的调度，资源的分配和管理，统领整个计算机硬件；应用程序是具有某种功能的程序，程序是运行于操作系统之上的。进程是一个具有一定独立功能的程序在一个数据集上的一次动态执行的过程，是操作系统进行资源分配和调度的一个独立单位，是应用

程序运行的载体。进程是一种抽象的概念，从来没有统一的标准定义。进程一般由程序，数据集合和进程控制块三部分组成。程序用于描述进程要完成的功能，是控制进程执行的指令集；数据集合是程序在执行时所需要的数据和工作区；程序控制块包含进程的描述信息和控制信息是进程存在的唯一标志。

程序是指令和数据的有序集合，其本身没有任何运行的含义，是一种静态的概率，而进程是程序在处理机上的一次执行过程，是一种动态的概率，程序可以作为一种软件资料长期存在，而进程是有一定生命周期的，程序是永久的，进程是短暂的，如tomcat软件是程序，当运行时则产生了一个tomcat的进程，当kill后则进程消失，狭义定义进程是正在运行程序的示例。通常在一个进程中可以包含若干个[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)，它们可以利用他们所在进程所拥有的资源，在引入[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)的[操作系统](https://baike.baidu.com/item/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位，由于[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)比进程更小，基本上不拥有[系统资源](https://baike.baidu.com/item/%E7%B3%BB%E7%BB%9F%E8%B5%84%E6%BA%90" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度。当下推出的[通用操作系统](https://baike.baidu.com/item/%E9%80%9A%E7%94%A8%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)都引入了[线程](https://baike.baidu.com/item/%E7%BA%BF%E7%A8%8B" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)，以便进一步提高系统的并发性，并把它视为[现代操作系统](https://baike.baidu.com/item/%E7%8E%B0%E4%BB%A3%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F" \t "https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B/_blank)的一个重要指标。

进程与线程的区别

1. 线程是程序执行的最小单位，而进程是操作系统分配资源的最小单位；

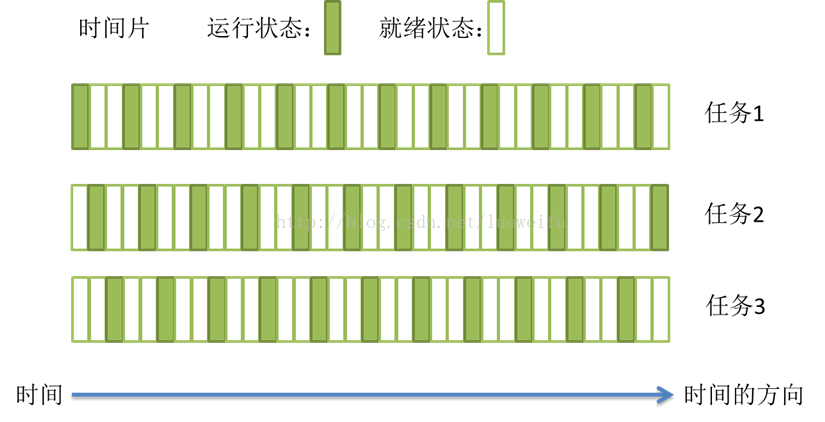
2. 一个进程由一个或多个线程组成，线程是一个进程中代码的不同执行路线

3. 进程之间相互独立，但同一进程下的各个线程之间共享程序的内存空间(包括代码段，数据集，堆等)及一些进程级的资源(如打开文件等)，某进程内的线程在其他进程不可见；

4. 调度和切换：线程上下文切换比进程上下文切换要快得多

任务调度理解

大部分操作系统的任务调度是采用时间片轮转的抢占式调度方式，也就是说一个任务执行一小段时间后强制暂停去执行下一个任务，每个任务轮流执行。任务执行的一小段时间叫做时间片，任务正在执行时的状态叫运行状态，任务执行一段时间后强制暂停去执行下一个任务，被暂停的任务就处于就绪状态，等待下一个属于它的时间片的到来。样每个任务都能得到执行，由于CPU的执行效率非常高，时间片非常短，在各个任务之间快速地切换，给人的感觉就是多个任务在“同时进行”，这也就是我们所说的并发



不使用多进程而是使用多线程的原因

线程廉价，线程启动比较快，退出比较快，对系统资源的冲击也比较小。而且线程彼此分享了大部分核心对象(File Handle)的拥有权，如果使用多重进程，但是不可预期，且测试困难。多线程应用场景主要能体现到多线程提高程序效率，如迅雷多线程下载、分批发送短信等、定时任务大量数据处理

守护线程

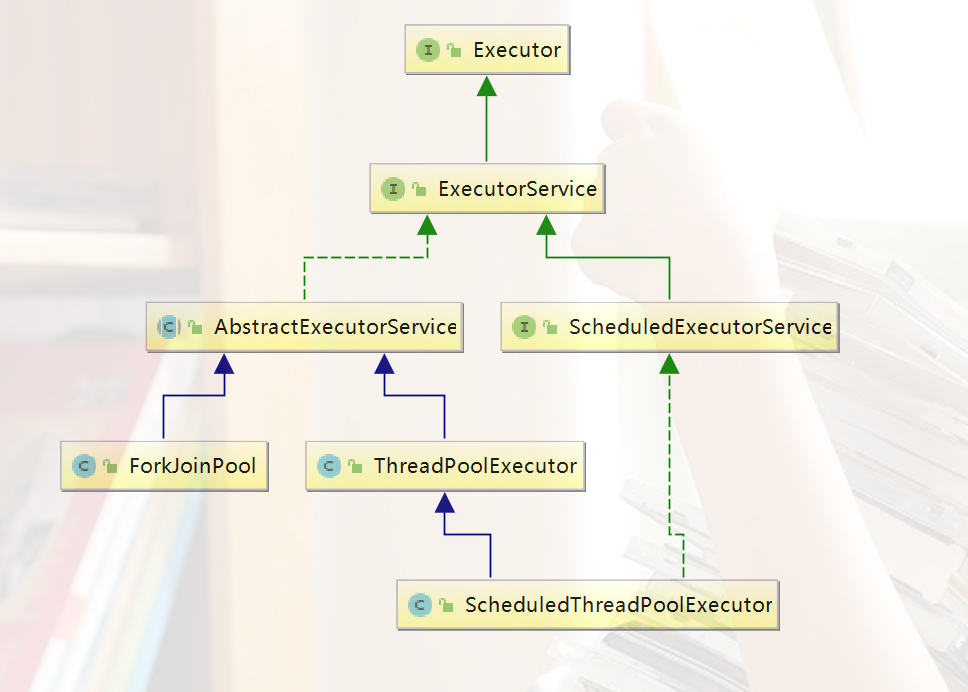
指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程。反过来说，只要任何非守护线程还在运行，程序就不会终止，唯一的不同之处就在于虚拟机的离开：如果用户线程已经全部退出运行了，只剩下守护线程存在了，虚拟机也就退出了。 因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了

多线程编程要保证的三大特性：原则性、可见性、有序性。利用syncronized、volatile等实现

java api

java.util.concurrent包是java关于多线程编程相关的类与接口

线程池相关核心类



Executor:一个表示执行提供的任务的对象的接口，将任务执行流与实际任务执行机制分离

ExecutorService:异步处理的完整解决方案,它管理内存中队列并根据线程可用性计划提交的任务

ThreadPoolExecutor:线程池，ExecutorService具体实现之一，由Executors线程池工厂获取

ScheduledThreadPoolExecutor:定时及周期执行的线程池

简单示例

public class MainTest {  
 public static void main(String[] args) throws ExecutionException, InterruptedException {  
 ExecutorService executorService = Executors.newFixedThreadPool(5);  
 for (int i = 0; i < 10; i++) {  
 executorService.execute(() -> {  
 System.out.println(Thread.currentThread().getName());  
 });  
 }  
 executorService.shutdown();  
 }  
}

Callable与Future接口

在Java中，创建线程一般有两种方式，一种是继承Thread类，一种是实现Runnable接口。然而，这两种方式的缺点是在线程任务执行结束后，无法获取执行结果。我们一般只能采用共享变量或共享存储区以及线程通信的方式实现获得任务结果的目的。不过，Java中，也提供了使用Callable和Future来实现获取任务结果的操作。Callable用来执行任务，产生结果，而Future用来获得结果。

示例

public class MainTest {  
 public static void main(String[] args) throws InterruptedException {  
 ExecutorService executorService = Executors.newFixedThreadPool(5);  
 List<Future> futures=new ArrayList<>();  
 for (int i = 0; i < 10; i++) {  
 Callable<Integer> task = new Callable<Integer>() {  
 @Override  
 public Integer call() throws Exception {  
 return 1;  
 }  
 };

//内部实现本质上也是创建一个Thread实例，将call里面的逻辑放入run中，没有异常将结果封装到future对象。  
 Future<Integer> future = executorService.submit(task);  
 futures.add(future);  
 }  
 for (Future future : futures) {  
 try {  
 System.out.println(future.get());  
 } catch (ExecutionException e) {  
 System.out.println("线程发生异常");  
 }  
 }  
 executorService.shutdown();  
 }  
}

并发队列

在并发队列上JDK提供了两套实现，一个是以ConcurrentLinkedQueue为代表的高性能队列，一个是以BlockingQueue接口为代表的阻塞队列，无论哪种都继承自Queue。

BlockingQueue：阻塞队列接口

阻塞队列与普通队列的区别在于，当队列是空的时，从队列中获取元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。同样，试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来，如从队列中移除一个或者多个元素，或者完全清空队列.

如BlockingQueue的一个具体实现ArrayBlockingQueue中的take方法

public E take() throws InterruptedException {

//加锁保证线程安全，而普通队列没有  
 final ReentrantLock lock = this.lock;  
 lock.lockInterruptibly();  
 try {

//判断为空时，等待其他线程向里插入数据，而普通线程则直接放回null或报错  
 while (count == 0)  
 notEmpty.await();  
 return dequeue();  
 } finally {  
 lock.unlock();  
 }  
}

注：java中的阻塞式方法是指在程序调用方法时，必须等待输入数据可用或者检测到输入结束或者[抛出异常](https://www.baidu.com/s?wd=%E6%8A%9B%E5%87%BA%E5%BC%82%E5%B8%B8&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)，否则程序会一直停留在该语句上，不会执行下面的语句如io中的read方法。

自定义一个阻塞队列示例

**public class** MainTest {  
 **public static void** main(String[] args) {  
 MyBlockingQueue<String> queue = **new** MyBlockingQueue<String>(5);  
 **new** Thread(() -> {  
 **for** (**int** i = 0; i < 10; i++) {  
 queue.put(**""**+i);  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
 **new** Thread(() -> {  
 **for** (**int** i = 0; i < 5; i++) {  
 queue.take();  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
  
 }  
}  
**class** MyBlockingQueue<T> {  
 **private** LinkedList<T> **linkedList** = **new** LinkedList<>();  
 **private** AtomicInteger **count** = **new** AtomicInteger(0);  
 **private int minSize** = 0;  
 **private int maxSize**;  
 **private** Object **lock** = **new** Object();  
 **public** MyBlockingQueue(**int** maxSize) {  
 **this**.**maxSize** = maxSize;  
 }  
 **public void** put(T t) {  
 **synchronized** (**lock**) {

//被唤醒后会再走循环判断是否符合条件，如果用if被唤醒后则直接往下走不会再判断，显示不能用if  
 **while** (**count**.get() == **maxSize**) {  
 **try** {  
 **lock**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 **linkedList**.add(t);  
 **count**.incrementAndGet();  
 System.***out***.println(**" 元素 "** + t + **" 被添加 "**);  
 **lock**.notify();  
 }  
 }  
 **public** T take() {  
 T temp;  
 **synchronized** (**lock**) {  
 **while** (**count**.get() == **minSize**) {  
 **try** {  
 **lock**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 temp = **linkedList**.removeFirst();  
 **count**.decrementAndGet();  
 System.***out***.println(**" 元素 "** + temp + **" 被消费 "**);  
 **lock**.notify();  
 }  
 **return** temp;  
 }  
 **public int** getSize() {  
 **return count**.get();  
 }  
}

阻塞队列实现生产者消费者模式

**public class** MainTest {  
 **public static void** main(String[] args) {  
 BlockingQueue publicBoxQueue= **new** LinkedBlockingQueue(5); *//定义了一个大小为5的盒子* WareHouse wareHouse=**new** WareHouse(publicBoxQueue);  
 Thread pro= **new** Consumer(wareHouse);  
 Thread con= **new** Producer(wareHouse);  
 pro.start();  
 con.start();  
 }  
}  
*/\*\*  
 \* 仓库  
 \*/***class** WareHouse {  
 **private** BlockingQueue **proQueue**;  
 **public** WareHouse(BlockingQueue proQueue) {  
 **this**.**proQueue** = proQueue;  
 }  
 **public void** produce() {  
 **for** (**int** i=0;i<10;i++)  
 {  
 **try** {  
 System. ***out*** .println(**"生产者生产的苹果编号为 : "** +i); *//放入十个苹果编号 为1到10* **proQueue** .put(i);  
 *//Thread.sleep(3000);* } **catch** (InterruptedException e) {  
 *//* ***TODO: handle exception*** e.printStackTrace();  
 }  
 }  
 }  
 **public void** consume() {  
 **for** (**int** i=0;i<10;i++)  
 {  
 **try** {  
 System. ***out*** .println(**"消费者消费的苹果编号为 ："** +**proQueue** .take());  
 Thread. *sleep*(3000); *//在这里sleep是为了看的更加清楚些* } **catch** (InterruptedException e) {  
 *//* ***TODO: handle exception*** e.printStackTrace();  
 }  
 }  
 }  
}  
**class** Consumer **extends** Thread {  
 **private** WareHouse **wareHouse**;  
 **public** Consumer( WareHouse wareHouse) {  
 **this**.**wareHouse** = wareHouse;  
 }  
  
 @Override  
 **public void** run() {  
 **wareHouse**.consume();  
 }  
}  
**class** Producer **extends** Thread {  
 **private** WareHouse **wareHouse**;  
  
 **public** Producer(WareHouse wareHouse) {  
 **this**.**wareHouse** = wareHouse;  
 }  
 @Override  
 **public void** run() {  
 **wareHouse**.produce();  
 }  
}

ConcurrentLinkedQueue：

一个适用于高并发场景下的队列，通过无锁的方式，实现了高并发状态下的高性能，通常ConcurrentLinkedQueue性能好于BlockingQueue.它是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。头是最先加入的，尾是最近加入的，该队列不允许null元素。

CyclicBarrier、CountDownLatch、Semaphore示例

CyclicBarrier:屏障，初始化时规定一个数目，然后计算调用了CyclicBarrier.await()进入等待的线程数，当线程数达到了这个数目时，所有进入等待状态的线程被唤醒并继续。

**public class** MainTest {  
 **public static void** main(String[] args) **throws** InterruptedException {  
 **int** threadCount=3;  
 CyclicBarrier cyclicBarrier = **new** CyclicBarrier(threadCount);  
 **for**(**int** i=0;i<10;i++){  
 **new** Worker(cyclicBarrier).start();  
 }  
 }  
}  
**class** Worker **extends** Thread{  
 **private** CyclicBarrier **cyclicBarrier**;  
  
 **public** Worker(CyclicBarrier cyclicBarrier) {  
 **this**.**cyclicBarrier** = cyclicBarrier;  
 }  
 @Override  
 **public void** run() {  
 **try** {  
 System.***out***.println(Thread.*currentThread*().getName() + **"开始等待其他线程"**);  
 **cyclicBarrier**.await();  
 System.***out***.println(Thread.*currentThread*().getName() + **"开始执行"**);  
 *// 工作线程开始处理，这里用Thread.sleep()来模拟业务处理* Thread.*sleep*(1000);  
 System.***out***.println(Thread.*currentThread*().getName() + **"执行完毕"**);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

CountDownLatch：计数器，实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。

**public class** MainTest {  
 **public static void** main(String[] args) **throws** InterruptedException {  
 CountDownLatch countDownLatch = **new** CountDownLatch(2);  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName() + **",子线程开始执行..."**);  
 countDownLatch.countDown();  
 System.***out***.println(Thread.*currentThread*().getName() + **",子线程结束执行..."**);  
 }  
 }).start();  
  
 **new** Thread(**new** Runnable() {  
 @Override  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName() + **",子线程开始执行..."**);  
 countDownLatch.countDown();*//计数器值每次减去1* System.***out***.println(Thread.*currentThread*().getName() + **",子线程结束执行..."**);  
 }  
 }).start();  
 countDownLatch.await();*// 減去为0,恢复任务继续执行* System.***out***.println(**"两个子线程执行完毕...."**);  
 System.***out***.println(**"主线程继续执行....."**);  
 }  
}

Semaphore：是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore可以用来构建一些对象池，资源池之类的，比如数据库连接池，我们也可以创建计数为1的Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。

wc.acquire(); //申请资源

wc.release();// 释放资源

**public class** MainTest {  
 **public static void** main(String[] args) **throws** InterruptedException {  
 Semaphore semaphore = **new** Semaphore(3);  
 **for** (**int** i = 1; i <= 10; i++) {  
 Worker worker = **new** Worker(**"第"** + i + **"个人"**, semaphore);  
 worker.start();  
 }  
 }  
}  
**class** Worker **extends** Thread {  
 **private** String **name**;  
 **private** Semaphore **wc**;  
  
 **public** Worker(String name, Semaphore wc) {  
 **this**.**name** = name;  
 **this**.**wc** = wc;  
 }  
 @Override  
 **public void** run() {  
 *// 剩下的资源* **int** availablePermits = **wc**.availablePermits();  
 **if** (availablePermits > 0) {  
 System.***out***.println(**name** + **"天助我也，终于有茅坑了....."**);  
 } **else** {  
 System.***out***.println(**name** + **"怎么没有茅坑了..."**);  
 }  
 *// 申请资源* **try** {  
 **wc**.acquire();  
 System.***out***.println(**name** + **"排到厕所，正在拉丝"** + **",剩下厕所:"** + **wc**.availablePermits());  
 Thread.*sleep*(**new** Random().nextInt(1000));  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**name** + **"厕所上完啦!"**);  
 *// 释放资源* **wc**.release();  
 }  
}

CAS算法理解（Compare and Swap即比较再交换）

（1）与锁相比，使用比较交换会使程序看起来更加复杂一些。但由于其非阻塞性，它对死锁问题天生免疫，并且，线程间的相互影响也远远比基于锁的方式要小。更为重要的是，使用无锁的方式完全没有锁竞争带来的系统开销，也没有线程间频繁调度带来的开销，因此，它要比基于锁的方式拥有更优越的性能。

（2）无锁的好处：

第一，在高并发的情况下，它比有锁的程序拥有更好的性能；

第二，它天生就是死锁免疫的。

就凭借这两个优势，就值得我们冒险尝试使用无锁的并发，atomic原子包下的类、ConcurrentLinkedQueue等没有加锁也是线程安全的类一般都用cas算法实现的。

（3）CAS算法的过程是这样：它包含三个参数CAS(V,E,N): V表示要更新的变量，E表示预期值，N表示新值。仅当V值等于E值时，才会将V的值设为N，如果V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。最后，CAS返回当前V的真实值。

（4）CAS操作是抱着乐观的态度进行的，它总是认为自己可以成功完成操作。当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败。失败的线程不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作。基于这样的原理，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰，并进行恰当的处理。

（5）简单地说，CAS需要你额外给出一个期望值，也就是你认为这个变量现在应该是什么样子的。如果变量不是你想象的那样，那说明它已经被别人修改过了。你就重新读取，再次尝试修改就好了。

（6）在硬件层面，大部分的现代处理器都已经支持原子化的CAS指令。在JDK 5.0以后，虚拟机便可以使用这个指令来实现并发操作和并发数据结构，并且，这种操作在虚拟机中可以说是无处不在。

CAS无锁机制的缺点

CAS存在一个很明显的问题，即ABA问题。问题：如果变量V初次读取的时候是A，并且在准备赋值的时候检查到它仍然是A，那能说明它的值没有被其他线程修改过了吗？

如果在这段期间曾经被改成B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。针对这种情况，java并发包中提供了一个带有标记的原子引用类AtomicStampedReference，它可以通过控制变量值的版本来保证CAS的正确性。

#### atomic

##### package.html

原子类的小工具包，支持在单个变量上解除锁的线程安全编程，原子变量类相当于一种泛化的 volatile 变量，能够支持原子的和有条件的读-改-写操作。AtomicInteger 表示一个int类型的值，并提供了 get 和 set 方法，这些 Volatile 类型的int变量在读取和写入上有着相同的内存语义。它还提供了一个原子的 compareAndSet 方法（如果该方法成功执行，那么将实现与读取/写入一个 volatile 变量相同的内存效果），以及原子的添加、递增和递减等方法。AtomicInteger 表面上非常像一个扩展的 Counter 类，但在发生竞争的情况下能提供更高的可伸缩性，因为它直接利用了硬件对并发的支持。

为什么会有原子类

jdk5增加了并发包java.util.concurrent.atomic其下面的类使用CAS算法实现了区别于synchronouse同步锁的一种乐观锁。JDK 5之前Java语言是靠synchronized关键字保证同步的，这是一种独占锁，也是是悲观锁。

如int i=0;i++在多线程下i++是不安全的，因为java在操作i++的时候，是分步骤做的，可以理解为：  
tp = i;  
tp2 = i+1;  
i=tp2;  
如果线程1在执行第一条代码的时候，线程2访问i变量，这个时候，i的值还没有变化，还是原来的值，所以是不安全的。在使用的时候，不可避免的会用到synchronized关键字。而java.util.concurrent.AtomicInteger是一个提供原子操作的Integer类，其提供了线程安全且高效的原子操作，是线程安全的。

#### locks

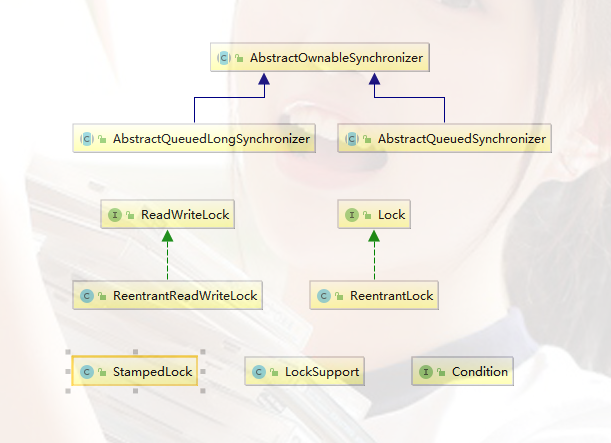
##### package.html

ava.util.concurrent.lock 中的Lock 框架是锁的一个抽象，它允许把锁定的实现作为 Java 类，而不是作为语言的特性来实现。这就为Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。简单来说，相对于synchronized ，锁是一种更灵活和精巧的线程同步机制。

灵活体现在：

* synchronized block 只能在一个方法内– Lock API的 lock() 和 unlock() 操作可以在不同的方法中
* synchronized 不支持公平锁，锁释放后处于等待的线程都有可能获得锁，也就是不能指定优先级。我们可以通过Lock API指定参数来实现公平锁，确保等待时间最长的线程最先获得锁
* 如果线程无法访问synchronized块，就会发生阻塞。Lock API提供了 \*\*tryLock()
* 方法，只有在锁可用而且没有被其他线程持有时去获得锁\*\* ，这减小了线程等待锁的阻塞时间
* 处于“waiting”状态的线程获得synchronized块时，不能被中断。而Lock API提供了lockInterruptibly() 方法，可以中断正在等待锁的线程

ava.util.concurrent.lock 包结构图



简单用法示例

ReentrantLock：实现了 Lock 接口，不仅提供了跟synchronized 方法和语句使用的隐式monitor锁相同的并发和内存语义 ，而且扩展了其功能。

class Outputter {

private Lock lock = new ReentrantLock();// 锁对象

public void output(String name) {

lock.lock(); // 得到锁

try {

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

System.out.print(name.charAt(i));

}

} finally {

lock.unlock();// 释放锁

}

}

}

ReadWriteLock：实现了ReadWriteLock 接口.以下是线程获取 ReadLock 和 WriteLock 的一些规则:

Read Lock – 没有线程获得写锁且没有获取写锁的请求，多个线程可以获得读锁

Write Lock – 如果没有线程读或者写，只有一个线程可以获取写锁

class Data {

private int data;// 共享数据

private ReadWriteLock rwl = new ReentrantReadWriteLock();

public void set(int data) {

rwl.writeLock().lock();// 取到写锁

try {

System.out.println(Thread.currentThread().getName() + "准备写入数据");

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

this.data = data;

System.out.println(Thread.currentThread().getName() + "写入" + this.data);

} finally {

rwl.writeLock().unlock();// 释放写锁

}

}

public void get() {

rwl.readLock().lock();// 取到读锁

try {

System.out.println(Thread.currentThread().getName() + "准备读取数据");

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName() + "读取" + this.data);

} finally {

rwl.readLock().unlock();// 释放读锁

}

}

}

Conditions

Condition可以替代传统的线程间通信，用await()替换wait()，用signal()替换notify()，用signalAll()替换notifyAll()。——为什么方法名不直接叫wait()/notify()/nofityAll()？因为Object的这几个方法是final的，不可重写！传统线程的通信方式，Condition都可以实现。

注意，Condition是被绑定到Lock上的，要创建一个Lock的Condition必须用newCondition()方法

class BoundedBuffer {

final Lock lock = new ReentrantLock(); //锁对象

final Condition notFull = lock.newCondition(); //写线程锁

final Condition notEmpty = lock.newCondition(); //读线程锁

final Object[] items = new Object[100];//缓存队列

int putptr; //写索引

int takeptr; //读索引

int count; //队列中数据数目

//写

public void put(Object x) throws InterruptedException {

lock.lock(); //锁定

try {

// 如果队列满，则阻塞<写线程>

while (count == items.length) {

notFull.await();

}

// 写入队列，并更新写索引

items[putptr] = x;

if (++putptr == items.length) putptr = 0;

++count;

// 唤醒<读线程>

notEmpty.signal();

} finally {

lock.unlock();//解除锁定

}

}

//读

public Object take() throws InterruptedException {

lock.lock(); //锁定

try {

// 如果队列空，则阻塞<读线程>

while (count == 0) {

notEmpty.await();

}

//读取队列，并更新读索引

Object x = items[takeptr];

if (++takeptr == items.length) takeptr = 0;

--count;

// 唤醒<写线程>

notFull.signal();

return x;

} finally {

lock.unlock();//解除锁定

}

}