数组：

静态初始化：确定数组长度和具体元素。

动态初始化：确定数组长度，不设置具体元素。

一维数组：数组中的每一个元素都是一个值（基本类型和应用类型）

二维数组：数组中的每一个元素又是一个一维数组

三维数组：数组中的每一个元素又是一个二维数组

----------严格上说 Java 中不存在多维数组，一般称之为数组中的数组

数组的搜索算法：从指定数组中搜索某一个元素的索引是多少。

方式一：线性搜索（从头搜索到尾）：index'Of/lastIndexOf

对于元素过多的数组，性能极低：有 N 个元素，循环次数=（N+1）/2；

方式二：二分搜索法/二分查找法/折半查找

前提：数组元素必须有顺序。

算法：当数据量很大时适宜采用该方法，并且数据须要是排序好的

排序 sort（int[] a,to..to..）to 表示闭包区间内[ )。

对象：

在 java 虚拟机 jvm 中 New 一个对象的数据储存在堆内存中，实例化在栈内存中，方法在方法去里。

对象的生命周期：

对象什么时候出生：每次使用 new 关键字的时候后，就会在内存开辟新的空间此时对象开始存在。

对象结束：当堆中的对象，没有被任何变量所引用。此时对象就成了垃圾，就等这垃圾回收器（GC）来回收该垃圾，当被回收的时候对象就被销毁了，回收垃圾的目的是为了释放更多的内存空间。

如果这个状态/行为属于整个事物（类），就直接用 static 修饰。

被所有对象共享。

在开发中工具方法往往用 static 修饰。

如果不使用 static 修饰，则这些方法属于该类的对象，我们得像创建对象在调用方法，在开发中工具对象只需要一份即可，可能创建 N 个对象，此时往往把该类设计成单例的，但还是有点麻烦。

所以一般开发工具的方法，为了简单用 static 修饰。

类的成员使用：

利处：对对象的共享数据进行单独的空间存储，节省空间没有必要每一个对象都存储一份，可以直接类名调用。

弊端：生命周期过长。

变量：

	存在位置	生命周期开始	生命周期结束	在内存位置
类变量	字段，使用 static 修饰	当所在字节码被加载进 jvm	当 jvm 停止	方法区
实例变量	字段，没有使用 static	当创建所在类的对象的时候	当该对象被 GC 回收	堆
局部变量	方法形参，代码块中方法内	当代码执行到初始化变量	所在方法/代码	

当前方法的栈帧中

什么时候使用成员变量和局部变量：
1、 考虑变量的生存时间，这会影响内存开销。
2、 扩大变量作用域，不利于提高程序的高内聚。
开发中应该尽量缩小变量的作用范围，如此在内存中停留时间越短，性能也就更高。
不要动不动就是用 static 修饰，一般定义工具方法的时候，static 方法需要访问的变量，该变量属于类，此时才使用 static 字段。
也不要动不动。就是用成员变量，因为存在着线程不安全问题，能使用局部变量尽量使用局部变量

Package 最佳实践：
1、 自定义包名，不能以 java.打头，因为 Java 的安全机制会检查。
2、 包名必须遵循标识符规范/全部小写。
3、 企业开发中，包名才有公司域名到写.。
Package 域名倒写.模块名.组件名.类名；
例：package com.xxx.pass.a;
类名称:
类的简单名称：定义类的名称
类的全限定名称：包含.类名；
4、 在开发中，都是先有 package 而后在 package 中定义类

静态导入（static import）：
Import static 类的全限定名*；此时表示当前类的任意使用的静态成员。

通过反编译工具，其实所谓的静态导入也是一个语法糖/编译器级别的新特性。
在实际开发中我们不使用静态导入，因为如此分不清某一个静态方法或字段来源于哪一个类。
Eclipse 工具，当格式化代码的时候，就字段取消了所有的静态导入，变成使用类名调用。

什么是封装：
1、 把对象的行为和状态看成一个统一的整体，将二者存放在一个独立的模块中（类!）；
2、 “信息隐藏“，把不需要让外界知道的信息隐藏起来，尽可能隐藏对象功能实现细节，向外暴露方法，保证外界安全访问功能；把所有的字段使用 ptivate 私有化，不准对外界访问，把方法使用 public 修饰，允许外界访问。
把所有数据信息隐藏起来，尽可能隐藏多的功能，只向外暴露便捷的方法，一共调用。

封装的好处
1、 使调用者正确方便地使用系统功能，防止调用者随意修改系统属性。
2、 提高组件的重用性。
3、 达到组件之间的低耦合性（当某一个模块实现发生变化时，只要对外暴露的接口不变，就不会影响到，其他的模块）通过什么来实现隐藏和暴露功能呢?

高内聚：把该模块的内部数据，功能细节隐藏在模块内部，不允许外界直接干预。
低耦合：该模块只需要给外界暴露少量的功能方法。

JavaBean 规范：
成员：
1、 方法：Method
2、 事件：event
3、 属性：property

属性：

1、 attribute：表示状态，Java 中没有该概念，很多人把字段（Field）称之为属性（attribute）
2、 property：表示状态，但是不是字段，是属性的操作方法（getter/setter）决定的，框架中使用的大多是是属性。

使用 this：
1、 解决成员变量和参数（局部变量）之间的二义性，必须使用；
2、 同类中实例方法互调（此时可以省略 this，但是不建议省略）。
3、 将 this 作为参数传递给另一个方法；
4、 将 this 作为方法的返回值（链式方法编程）；
5、 构造器重载的互调，this（参数）必须写在构造方法第一行；
6、 Static 不能和 this 一起使用；
当字节码被加载进 jvm，static 成员以及存在了
但是此时对象还没有创建，没有对象，就没有 this。

方法覆写原则（一同两小一大）
一同：
实例方法签名必须相同。（方法签名=方法名+方法的参数列表）
两小：
子类方法的返回值类型是和父类方法的返回类型相同或者是其子类。
子类可以返回一个更加具体的类
子类方法中声明抛出的异常类型和父类方法声明抛出的异常类型相同或者是其子类。
子类方法中声明抛出的异常小于或等于父类方法抛出异常类型；
子类方法可以同时声明抛出多个属于父类方法声明抛出异常类的子类（RuntimeException 类型除外）
一大：
子类方法的访问权限比父类方法访问权限更大或者相等。

Super 关键字的使用场景：
1、 可以使用 super 解决子类隐藏父类的字段情况，该情况，一般不讨论，因为会破坏封装。
2、 在子类方法中，调用父类被覆盖的方法，引出 super 的例子，必须使用 super
3、 在子类构造器中，调用父类构造器，此时必须使用 super 语句：super（实参）。
满足继承的访问权限下，隐藏父类静态方法：若子类定义的静态方法的签名和超类中的静态方法签名相同，此时就是隐藏父类方法。两个相同方法并且是静态方法。

Static 不能 super 和 this 共存

在外部访问内部类：外部类。内部类对象 = 外部类实例对象。new 内部类（）
Outer.in = (new Outer)out.new Inner();
Final 修饰符：

面试题：列举 5 个 Java 中内置的使用 final 修饰的类。
Java 里 final 修饰的类有很多，比如八大基本数据类型保证类和 String 等
包装类：Boolean，Character，Short，Integer，Long，Float，Double，Byte，Void
字符串类：String，StringBuilder，StringBuffer
系统类：Class，System，RuntimePermission，Compiler
数学类：Math，StrictMath
其他：Character.UnicodeBlock，ProcessBuilder，StackTraceElement
面试题：final 修饰的引用类型变量到底表示引用的地址不能变，还是引用空间中的数据不能改变。
Final 修饰基本类型变量：表示该变量的值不能改变，既不能用“=”号重新赋值。
Final 修饰引用类型变量：表示变量的引用的地址不能变，而不是引用地址的内容不能变。

设计模式：是一套被反复使用、多数人只晓的、经过分类编目的、代码设计经验的总结

单例设计模式（singleton）：最常用简单的设计模式单例的编写有 N 种写法。
目的：保证某一个在整个应用中某一个类且只有一个实例（一个类在堆内存只存在一个对象），即所有指向该类型实例的引用都指同一块内存空间。
写单例模式的步骤：单讲饿汉式

1、 必须在该类中，自己先创建出一个对象。
2、 私有化自身的构造器，防止外界通过构造器创建新的对象。
3、 想外暴露一个公共的静态方法用于获取自身的对象。

意识：

当看到 API 文档的时候，发现没有公共的构造器，但是方法又不是 static 修饰的，立马又意识：该类中至少有一个 static 方法用于返回当前类对象->体现单例模式
工具类如何设计：工具在开发中其实只需要一份即可。

1、 如果工具方法没有使用 static 修饰，说明工具方法的使用工具类的对象来调用，此时把工具类设计为单例的。
2、 如果工具方法全部使用 static 修饰，说明工具方法，只需要使用工具类名调用即可。此时必须把工具类的构造方法私有化（防止创建工具类对象调用静态方法）。

一般首选第二种。

模板模式：
一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

主要解决：一些方法通用，却在每一个子类都重新写了这一方法。
何时使用：有一些通用的方法。

如何解决：将这些通用算法抽象出来。

String 和基本类型转换：
String 和 int/Integer 之间的转换操作：转换方法必须在 String 类中或 Integer 类中。
把包装类转对象 String： String str = 任何对象.toString();
把基本数据类型转换 String：String str = 17+“ ”;
把 String 转换为基本数据类型：方法 parseXxx（String s）：Xxx 表示 8 大基本数据类型

包装类中的缓存设计（享元模式）：Character：（0，128） Byte、Short、Integer、Long：缓存[-128,127]区间。
Integer 与 int 的区别（包装类和基本数据类型的区别）：

1、 默认值：
   Int 的默认值 0
   Integer 的默认值既可以是 0 也可以是 null
2、 包装类提供该类相关的算法操作方法：
   例如进制转换。。。
3、 在集合框架中只能存储对象类型，不能存储基本数据类型。
4、 方法中的，基本类型变量存储在栈中，包装类型存放于堆中。

接口和抽象类的区别：
相同点：
1、 都位于继承的顶端，用于被其他实现或继承。
2、 都不能被实例化。
3、 都可以定义抽象方法，其子类/实现类都必须覆写这些抽象方法。
不同点：
1、 抽象类没有构造方法，抽象类有构造方法。
2、 抽象类可包含普通方法和抽象方法，接口只能包含抽象类方法（Java8 之前）。
3、 一个类只能继承一个直接父类（可能是抽象类），接口是多继承的并且只支持一个类实现多个接口（弥补了 Java 中的单继承）。
4、 成员变量：接口默认 public static final ，抽象类是默认包权限。
5、 方法：接口默认是 public abstract，抽象类默认是包访问权限。
6、 内部类：接口默认是 public static，抽象类默认是包访问权限。

如果接口和实现类可以完成相同的功能，尽量使用接口，面向接口编程，
设计模式：接口和抽象类集合使用的（适配器模式）

内部类根据使用不同的修饰符或者定位的位置不同，分成四张
四种内部类：
1、 实力内部类：内部类没有使用 static 修饰。
2、 静态内部类：使用 static 修饰
3、 局部内部类：在方法中定义的内部类
4、 匿名内部类适合于仅使用一次使用的类，属于局部内部类的特殊情况。
局部内部类（打死不会用）：
在方法中定义的内部类，其可见范围是当前方法和局部变量是同一个级别。
1、 不能使用 public，private，protect，static 修饰符
2、 局部内部类只能在当前方法中使用
3、 局部内部类和实例内部类一样，不能包括静态成员。
4、 局部内部类和实力内部类，可以访问外部类所有成员。
5、 局部内部类访问的局部变量必须使用 final 修饰（Java 中是自动隐式加上 final，但依然是常量不能改变值）
   原因：如果当前方法不是 main 方法，那么当前方法调用完毕之后，当前方法的栈帧被销毁，方法内部的局部变量的空间销毁。然后局部内部类是定义在方法中的，而且在方法中创建局部内部类对象，而局部内部类会去访问局部变量，当前方法被销毁的时候，

对象还在堆内存中，依然持有对局部变量的引用，但是方法被销毁的时候局部变量以及被销毁。

此时出现：在堆内存中，一个对象引用看一个不存在的数据，为了避免该问题，我们使用 final 修饰局部变量，从而变成常量永驻内存空间，即使方法被销毁之后，该局部变量也在内存中，对象可以继续持有。

枚举类的特点：

枚举特点：【这样写没有任何意义。每一个特点都是通过具体的原因引入的，不是在这里写总结】

1、 枚举的直接父类 java.lang.Enum，但不能显示继承 Enum。
2、 枚举就相当于一个类，可以定义构造方法、成员变量、普通方法和抽象方法。
3、 默认私有的构造方法，即使不写访问权限也是 private。（假构造器）
4、 每一个实例分别用一个全局常量表示，枚举的对象是固定的，实例个数有限，不能使用 new 关键字。
5、 枚举实例必须位于枚举体中最开始部分，枚举实例表的分号于其他成员相隔。
6、 枚举实例后有括号时，该实例是枚举类的匿名内部类对象（查看编译后的 clas 文件）

枚举的使用：

1、 枚举中都是全局公共的静态常量，可以直接使用枚举类名调用
   Weekday day=Weekday.SATURDAY;
2、 java.long.Enum 类是所有枚举类的父类，所以所有的枚举类可以调用 Enum 类中的方法。
3、 编译其生成枚举类的静态方法（从反编译代码中）：
   枚举类型[] values();
   Weekday[] ws=Weekday.values();//返回当前枚举所有常量，使用一个数组封装起来
   枚举类型 valueof(String name)
   Weekday day=Weekday.valueof(“xx”);//把一个指定名称字符串转换为当前枚举类中同名常量。
4、 从 java5 开始出现枚举，switch 也支持操作枚举类型
   Switch 只支持 int 类型，支持枚举的原因是因为底层使用的枚举对象的 ordinal，而 ordinal 方法的类型依然是 int 类型

枚举适用单例设计模式

常用类笔记：

字符串的分类：

不可变的字符串 String：当前对象创建完毕之后，该对象的内容（字符序列）是不能改变的，一旦内容改变就是一个新的对象。

可变的 字符串 StringBuilder/StringBuffer：当对象创建完毕之后，该对象的内容可以发生改变，当内容发生改变的时候，对象保持不变。

String 对象的空值：

1、 引用为控：String s=null
2、 内容为空：String s=””

常量池：专门储存常量的地方，都指的方法区中。

编译常量池：把字节码加载进 JVM 的时候，存储的是字节码的相关信息

运行常量池：存储常量数据。

面试题 1：
下列代码分别创建了几个 String 对象
String str=”ASD”;
最少创建一个 String 对象，最少不能创建 String 对象，如果常量池中存在“ASD“，那么直接引用，此时不创建，否则，先在常量池创建”ASD“内存空间，再引用
String str=new String(“ASD”)
最多创建两个 String 对象，至少创建一个 String 对象。New 关键字：绝对会在对空间，创建内存区域，所以至少创建一个 String 对象。
String 对象比较：
1、 单独使用““引号创建的字符串都是直接量，编译其就已经确定存储到常量池中；
2、 使用 new String（““）创建的对象会存储到堆内存中，是运行期才创建；
3、 使用只包含直接量的字符串符和如“AA“+”BB“创建的也是直接量编译期就能确定，已经确定存储常量池。
4、 使用包含 String 直接量（无 final 修饰字符）的字符串表达式（“A“+s1）创建的对象运行期才创建的，存储在堆中。
通过变量/调用方法去连接字符串，都只能在运行时期确定变量的值和方法的返回值，不存在编译优化操作

String 做字符串拼接的时候，性能极低（耗时），原因是 String 是不可变的，每次内容改变都会在内存中创建新的对象。
拼接字符串统统使用 StringBuffer/StringBuilder，不使用 String

String 和 StringBuilder 以及 StringBuffer 的区别：
StringBuffer 和 StringBuilder 都表示可变字符串，功能方法都是相同的。
唯一区别：
StringBuffer：StringBuffer 中的方法都使用了 synchronized 同步锁，表示同步的，保证线程安全性能较低。
StringBuilder：StringBuilder 中的方法没有使用 synchronized 修饰符，线程不安全，效率高。

随机数：随机的生成的任意的一个书（理论上讲具有不可预知性）
Random 类：
ThreadLocalRandom 类：
UUID 类：

Random 类用于生产一个伪随机数（通过相同的种子，产生的随机数是相同的）。
public Random（）：使用默认的种子（以前系统时间作为种子）。
public Random（long seed）：根据指定的种子。

ThreadLocalRandom 是 java7 新增类，是 random 类的子类，在多线程并发情况下，ThreadLocalRandom 相对于 Random 可以见少多线程资源竞争，保证了线程的安全性。
public class ThreadLocalRandom extends Random{
因为构造器是默认的访问权限，只能在 java.util 包中创建对象，提供了一个方法 ThreadLocalRandom.current（）用于返回当前类对象。
ThreadLocalRandom random = ThreadLocalRandom.current();
System.out.println(random.nextInt(34,179);//生成之间随机数
}

UUID：通用唯一识别：Universally Unique Identifler；在一台机器上生成的数字，它保证对在

同一时空中的所有机器都是唯一的。
UUID 是一个 128 位长的数字，一般用 16 进制表示。算法的核心思想时结合机器的网卡、当地时间、一个随即数来生产 UUID。

```
String uuid = UUID.randomUUID().toString();
System.out.println(uuid);
```

Date 类
日期格式化操作：

DateFormat：可以完成日期的格式化操作
格式化（format）：Date 类型对象--→String 类型：String format（Date date）
解析（parse）：String 类型时间--→Date 类型：Date parse（String source）

DateFormat 转换的格式是固定的，我们想根据自己的风格来转换。
2020-03-03 10:10:59
2020/03/03 10:10:59
SimpleDateFormat 类：是 DateFormat 的子类支持自定义格式模式

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
Y	Week year	Year	2009; 09
M	Month in year (context sensitive)	Month	July; Jul; 07
L	Month in year (standalone form)	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day name in week	Text	Tuesday; Tue
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	1
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800
X	Time zone	ISO 8601 time zone	-08; -0800; -08:00

Calendar(日期)类：

YEAR	年	*MINUTE*	分	*DAY_OF_WEEK_IN_MONTH*	某月中第几周
MONTH	月	*SECOND/MILLISECOND*	秒/毫秒	*WEEK_OF_MONTH*	日历式的第几周
DATE	日	*DAY_OF_MONTH*	和DATE一样	*DAY_OF_YEAR*	一年的第多少天
HOUR_OF_DAY	时	*DAY_OF_WEEK*	周几	*WEEK_OF_YEAR*	一年的第多少周

Regex（正则表达式）包含类：
1、Pattern 类：
Pattern 对象是一个正则表达式的编译表示。Pattern 类没有公共构造方法。要创建

一个 Pattern 对象，必须首先调用其公共静态方法，它返回一个 Pattern 对象。该方法接受一个正则表达式作为它的第一个参数。

Pattern p=Pattern.compile(String regex)

将给定的正则表达式编译为模式。

2、Matcher 类：

Matcher 对象是对输入字符串进行解释和匹配操作的引擎。与 Pattern 类一样，Matcher 也没有公共构造方法。需要调用 Pattern 对象的 matcher 方法来获得一个 Matcher 对象。

Matcher m=p. matcher(String str)；创建一个匹配器，匹配给定的输入与此模式。

3、PatternSyntaxException：

PatternSyntaxException 是一个非强制的异常类，他表示一个正则表达式中的语法错误。

异常类：

1、 Error:表示错误，一般指 JVM 相关的不可修复的错误，如，系统奔溃，内存溢出，JVM 错误，所有的子类都是以 Error 结尾。

2、 Exception:表示异常，指程序中出现不正当的情况，该问题可以修复（处理异常）。

常见的 Error：

StackOverflowError：当应用程序递归太深发生内存溢出时，抛出该错误。

常见的 Exception：

NullPotinterException：空指针异常。

ArrayIndexOutOfBoundsException：数组索引越界异常等。。。。。。

出现异常处理异常用

try{

有异常程序

}catch(　程序异常类型 Exception){

打印异常信息}finally{}

注意：

一个 catch 语句，只能捕获一种类型的异常，如果需要捕获多种异常，就得使用多个 catch 语句。

代码在一瞬间只能出现一种类型的异常，只需要一个 catch 捕获，不可能同时出现多个异常

Finally 语句块表示最终都会执行的代码，无论有没有异常。

Finally 不能单独使用　当只有 try 或者 catch 中调用退出 JVM 的相关方法，此时 finally 才不会执行，否则 finally 永远会执行。

如何获取异常信息，Throwable 类的方法：

1、 String 个体 Message（）：获取异常的描述信息。

2、 String toString（）：获取异常的类型和异常描述信息。

3、 void printStackTrace（）：打印异常的跟踪信息并输出到控制台。

在开发测试阶段都得使用 printStackTrace

异常（Exception）的分类：根据在编译时期还是运行时期去检查异常？

1、 编译时期异常：checked 异常.在编译时期，就会检查，如果没有处理异常，则编译失败。

2、 运行时期异常：Runtime 异常，在运行时期，检查异常，在编译时期，运行异常不会被编译其检测（不报错） 运行异常：在编译时期，可处理，可不处理

Throw 语句：运用于方法内部，抛出一个具体的异常对象。

throw new 异常类（“异常信息”）；终止方法。

Return 是返回一个值，throw 是返回一个错误返回该方法的调用者

Throws：运用于方法声明之上，用于表示当前方法不处理异常，而是提醒该方法的调用者来处理异常（抛出异常）

自定义异常类：在开发中根据自己业务的异常情况来定义异常类。

自定义一个业务逻辑异常。

异常类如何定义：

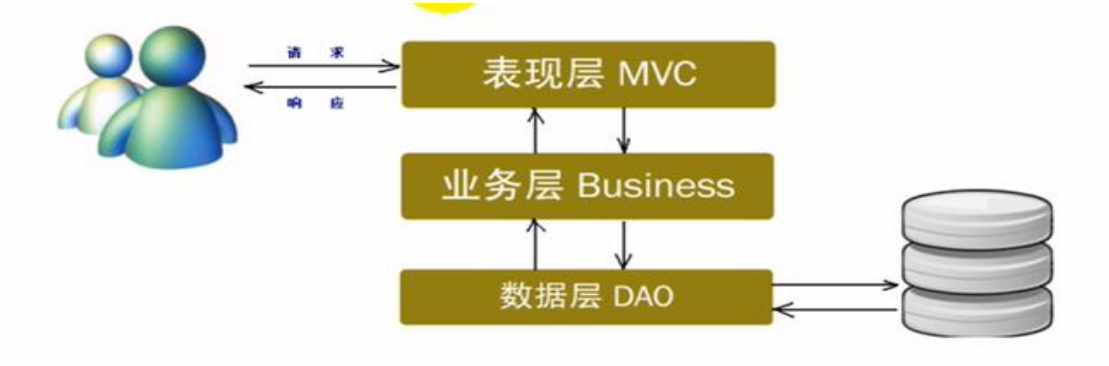
方式一：自定义一个受检查的异常类：自定义类并继承于 java.lang.Exception.

方式二：自定义一个运行时期的异常类：自定义类并继承于 java.lang.RuntimeException.

异常转译：当位于最上层的子系统不需要关心底层的异常细节时，常见的做法是捕获原始的异常，把它转换为一个新的不同类型的异常，再抛出异常，

例子：上级领导不关心下属出现的问题细节，关心下属是否出问题了，下属只能报告上级出现问题了，而不能将问题细节报告。

异常链：把原始的异常包装为新的异常类，从而形成多个异常的有序排列，有助于查找自身异常的根本原因。



Java7 开始自动关闭资源

Try（

资源对象 并且资源类口实现 java.lang.AutoCloseable 接口）

{可能出现异常代码}

catch（Exception e）{

}

处理异常的原则：

1、异常只能用于非正常情况，try-catch 的存在也会影响性能。

2、需要为异常提供说明文档 java doc ，如果自定义异常或某一个方法抛出了异常，我们应该记录在文档注释中。

3、尽可能避免异常。 如 NullPointerException

4、异常的力度很重要，应该为一个基本操作定义一个 try-catch 块，不要为了简便，将几百行代码放到 try-catch 块中。

5、不建议在循环中进行异常，应该在循环外对异常进行捕获处理（在循环之外使用 try-catch）。

6、自定义异常尽量使用 RuntimeException 类型的。

面试题：
1、 Error 和 Exception 的区别和关系。
2、 Checked 异常和 Runtime 异常区别。
受检查（编译）异常和运行异常
3、 如何保证一段代码必须执行到（finally）。
4、 Finally 中的代码一定会执行吗？
5、 Finally 和 return 的执行顺序？
如果程序是从 try 代码块或者 catch 代码块中返回时，finally 中的代码总会执行。而且 finally 语句在 return 语句执行之后 return 返回之前执行的。可以使用编译器的 Debug 功能查看详细过程。
6、 Throw 和 throws 的区别？
7、 列举 5 个常见的异常类。
8、 列举 5 个常见的 Runtime 异常类。
ArithmeticException：算术异常
SystemException：系统异常
ArrayStoreException：数组不兼容异常
NullPointerException：空指针异常
StringIndexOutOfboundsException：String 操作中索引越界异常
NumberFormatException：数字格式化异常
ClassCastException：类型强制转换异常

多线程：
并发和并行是即相似又有区别（微观概念）：
并行：指两个或多个事件同一时刻点发生；
并发：指两个或多个事件在同一时间段内发生。
进程包含线程，一个进程至少包含一个线程。
进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。
线程：堆空间共享的，栈空间是独立的，线程消耗资源也比进程小，相互之间，可以影响的，又称为轻型进程或进程元。
因为一个进程中的多个线程是并发运行的，那么从微观角度上考虑也是有先后顺序的，那么哪个多线程执行完全取决于 CPU 的调度器（JVM 来调度），程序员是控制不了的。‘
多线程并发可以看作多个线程在瞬间抢占 CPU 资源，谁抢到资源就运行，这也造就了多线程的随机性。
Java 程序的进程里至少包含主线程和垃圾回收线程（后台线程）。
线程调度：
计算机通常只有一个 CPU 时，在任意时刻只能执行一条计算机指令，每一个进程只有获得 CPU 的使用权才能执行指令。所谓进程多并发运行，从宏观上看，其实是各个进程轮流活得 CPU 的使用权，分别执行各自的任务。
那么在运行池中，会有多个线程处于就绪状态等到 CPU，JVM 就负责了线程的调度。
JVM 采用的是抢占式调度，没有采用分时调度，因此，可能造成多线程执行结果的随机性。
Java 操作进程：
开启一个进程：“
方法 1：Runtime 类中 exec（）方法

方式 2：ProcessBuilder 类中的 start()方法。

创建和启动线程，传统有两种方法；
方式 1：继承 Thread 类；
方式 2：实现 Runnable 接口；

方式 1、继承 Thread 类步骤：
1、 定义一个类 A 继承于 java.lang.Thread 类
2、 在 A 类中覆盖 Thread 类中的 run()方法
3、 在 run()方法中编写需要执行的操作-----→run 方法里的线程执行体。
4、 在 main()方法中创建线程对象，并启动线程。
创建线程类： A 类 a=new A 类；
调用线程的 start()方法： a.start();//启动一个线程。
注意千万不要调用 run()方法，如果调用 run()方法好比是对象调用方法，依然还是只有一个线程，并没有开启新的线程
方式 2、实现 Runnable 接口
步骤：
1、 定义一个类 A 实现于 java.lang.Runnable 接口，注意 A 类不是线程类。
2、 在 A 类中覆盖 Runnable 接口 run 方法。
3、 在 run()方法中编写需要执行的操作-----→run 方法里的线程执行体。
4、 在 main()方法（线程）中，创建线程对象，并开启线程。
创建线程对象： Thread t=new Thread(new A());
调用线程对象打的 start 方法： t.start();
匿名内部类线程创建：

```
New Thread(new Runnable{
Public void run(){
    }
}).start();
```

继承和实现方式区别：

继承方式：
1、 java 中类是单继承的，如果继承了 Thread，该类就不能再有其他的直接父类了
2、 从多线程共享一个资源上分析，继承方式做不到。（不能共享同一个资源）
实现方式：
1、 java 中多实现接口，此时该类还可以继承其他类，并且还可以实现其他接口（设计上更优雅）。
2、 从多线程共享同一个资源。（可以共享同一个资源）

在开发中尽量使用 Runnable 接口
同步锁：
同步监听对象/同步锁/同步监听器/互斥锁
一般把当前并发访问得共同资源作为同步监听对象
注意：在任何时候，最多允许一个线程有同步锁，谁拿到锁就进入代码块，其他的线程只能在外等着。
为解决多线程并发访问多一个资源安全性问题：
解决方案：当 A 线程进入操作时，B 和 C 只能等待状态，A 操作结束时，A 和 B 和 C 才有机

会去执行代码。

方式 1：同步代码块
Synchronized（同步锁）{
代码块
}
方式 2：同步方法（把需要同步的代码块单独用一个同步方法提取出来）
使用 synchronized 修饰的方法，就叫做同步方法，保证 A 线程执行该方法的时候，其他线程在方法外等着。
Synchronized public void doWork(){
//TODO
}
同步锁是谁：
　　对于非 static 方法，同步锁就是 this
对于 static 方法，我们使用当前方法所在类的字节码对象(XX.class)
Synchronized 的优缺点：
有点：保证了多并发访问时的同步操作，避免多线程的安全问题。
缺点：使用 synchronized 的方法/代码块的性能比不用要低一些。
建议：尽量减小 synchronized 作用域。

---

synchronized 关键字主要有以下几种用法：
- 非静态方法的同步；
- 静态方法的同步；
- 代码块。

线程通信：不同线程知性不同的任务，如果这些任务有某种关系，线程之间必须能够通信，协调完成工作。
线程通信体现面向对象的低耦合。
高耦合：两线程之间直接进行交流
低耦合：使用一个中间对象，屏蔽直接进行的数据交互。
中间对象（共享数据）收到 A 线程的数据发送给 B，当中间对象未收到 A 发送的数据，中间对象则指令 B 处于等待状态，当中间对象收到 A 发送的数据，中间对象唤醒 B 去处理数据。
当 B 线程收到数据处理完数据就立刻进入等待状态，等待接受到 A 线程发送的数据当中间对象收到数据并唤醒 B 线程处理数据。
（发送一个，接收一个，发送完未接收，发送等待接收到后，再发送）
同步锁池：
　　同步锁对象必须是多个线程的共同资源对象
当线程在生成数据的时候（先拥有同步锁），其他线程就在锁池中等待获取锁。
当线程执行完同步代码块的时候，就会释放同步锁，其他线程就开始抢锁的使用权。
Wait 和 notify 方法结介绍：
Java.lang.Object 类提供两类用于操作线程通信的方法。
Wait（）：执行该方法的线程对象释放同步锁，JVM 把该线程存到等待池中，等待其他线程唤醒该线程。
Notify（）：执行该方法的线程唤醒在等待池中等待的任意一个线程，把线程转到锁池中等待。
注意：上述方法只能被同步监听锁对象调用，否则报错 IllegalMonitorStateException。

多个线程只有使用相同的一个对象的时候，多线程之间才有互斥效果。(同步监听对象/同步锁)

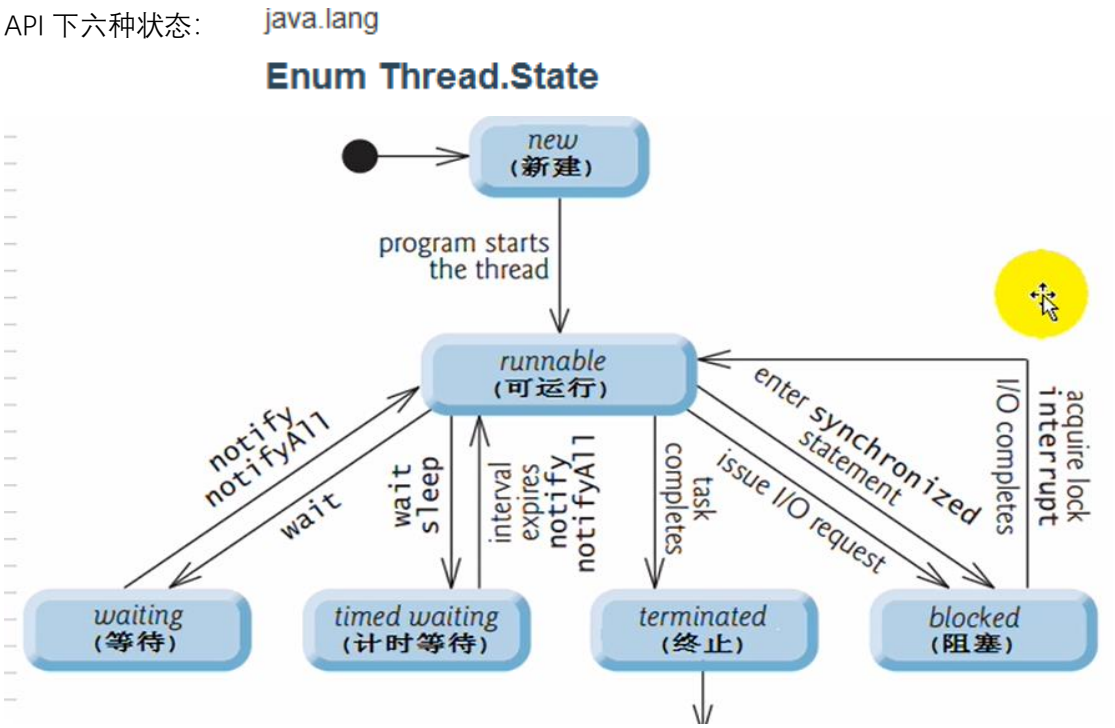
同步锁对象可以选择任意类型的对象即可，只需保证多个线程使用的时相同锁对象即可。
因为，只有同步监听锁对象才能调用 wait 和 notify 方法，所以，wait 和 notify 方法应该存在于 Object 类中
而不是 Thread 类中。
线程通信-使用 lock 和 Condition 接口：
Wait()和 notify()方法，只能被同步监听锁对象来调用，否则报错 IllegalMonitorStateException.
Lock 机制没有同步锁，也就没有自动获取锁和自动释放锁的概念，所以 Lock 机制不能调用 wait 和 notify 方法。
Java5 中提供了 Lock 机制的同时提供了处理 Lock 机制的通信控制的 Condition 接口

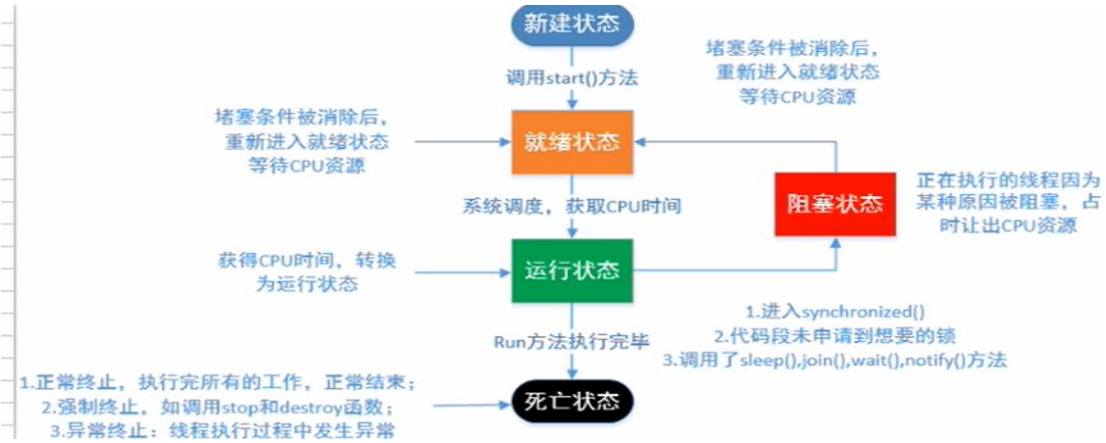
从 java5 开始：

1、 使用 Lock 机制取待 synchronized 代码块和 synchronized 方法
2、 使用 Condition 接口对象的 await、signal、signalall 方法取待 Object 类中的 wait、notify、notifyall 方法。

多线程通信的时候很容易造成死锁，死锁无法解决，只能避免。
避免思索地法则：多个线程都要访问共享的资源 A、B、C 时，保证每一个线程都按照相同的顺序访问他们。-----哲学家就餐问题---死锁例子
Thread 类中过时的方法：
Suspend（）：是正在运行的线程放弃 CPU 暂停运行。
Resume（）：使暂停的线程恢复运行。

生命周期：一个事物从出生的那一刻开始到死亡的整个过程。线程的生命周期中，不同的状态可以相互之间转换。
API 下六种状态：





线程对象的状态存放在 Thread 类的内部类（state）中：

注意：Thread.State 类其实是一个枚举类。

因为线程对象的状态是固定的，只有 6 中使用枚举类最恰当的。

1、 新建状态（new）：使用 new 创建一个线程对象，仅仅在堆中分配内存空间，在调用 start 方法之前。

新建状态下，线程压根就没有启动，仅仅只是存在于线程对象而已。

Thread thread=new Thread();//此时 thread 就属于新建状态

当新建状态下的线程对象调用了 start 方法，此时从新建状态进入可运行状态。

线程对象的 start()方法只能调用一次，否则报错 IllegalThreadStateException

2、 可运行状态（runnable）：分成两种状态，ready 和 running。分别表示就绪状态和可运行状态。

a) 就绪状态：线程对象调用 start 方法之后，等待 JVM 的调度。

b) 运行状态：线程对象获得 JVM 调度，如果存在多个 CPU，那么允许多个线程并行运行。

3、 阻塞状态（blocked）：正在运行的线程遇因为某些原因放弃 CPU，暂时停止运行，就会进入阻塞状态。

此时 JVM 不会给线程分配 CPU，直到线程重新进入就绪状态，才有机会转到运行状态

阻塞状态只能先进入就绪状态，再进入运行状态

阻塞情况两种：

1) 当 A 线程处于运行过程时，试图获取同步锁时，却被 B 线程获取，此时 JVM 把当前 A 线程存到对象的锁池中，A 线程处于阻塞状态。

2) 当线程处于运行过程时，发出 IO 请求时，此时进入阻塞状态。

4、 等待状态（waiting）（等待状态只能被其他线程唤醒）： 方法，此时 JVM 把当前线程存在对象等待池中。

5、 计时等待状态(time wait):

a) 当线程处于运行过程时，调用了 wait()

b) 当前线程执行了 sleep()方法

c) 当线程处于运行过程时，调用了 wait(long time)方法时，此时 JVM 把当前线程存在对象等待池中。

d) 当前执行力 sleep(long time)方法。

6、 终止状态(terminated)：通常称为死亡状态，表示线程终止

a) 正常执行完 run 方法而退出（正常死亡）。

b) 遇到异常而退出意外死亡。

线程一旦终止，就不能再重新启动，否则会报错。

线程休眠：让执行的线程暂停有一段时间，进入计时等待状态。

方法 sleep(long mills)

调用 sleep 后，当前线程放弃 CPU，在指定的时间段之内，sleep 所在的线程不会活得执行的机会。此刻状态下的线程不会释放同步锁/同步监听器

改方法更多用于模拟网络延迟让多线程并发访问同一个资源的错误效果更明显。

联合线程：

线程的 join()方法表示一个线程等待另一个线程完成后才执行。Join 方法被调用之后，线程处于阻塞状态。

后台线程：

在后台运行的线程，其目的是为了其他线程提供服务，也称为“守护线程”。JVM 的垃圾回收线程就是典型的后台线程。

特点：所有的前台线程都死亡，后台线程自动死亡，前台线程没有结束，后台线程就不会结束

测试线程对象，是否为后台线程：使用 Thread.isDaemon()方法

前台线程创建的线程默认是前台线程，可以通过 setDaemon()方法设置为后台线程，并且当且仅当后台线程创建的新线程时，新线程是后台线程

设置后台线程：Thread.setDaemon(true)方法，该方法必须在 start 方法前调用。否则出现 IllegalThreadStateException 异常。

线程优先级： setPriority(int newPriority) 更改此线程的优先级。

每个线程都有优先级，优先级的高低只是给线程执行机会的次数多少有关，并非线程优先级越高就先执行，线程的先运行取决于 CPU 的调度。

MAX_PRIORITY=10,最高

MIN_PRIORITY=1，最低

NORM_PRIORITY=5 默认优先级

每个线程都有默认的优先级，主线程默认优先级为 5，如果 A 线程创建了 B 线程，那么 B 线程和 A 线程具有相同优先级。（不同操作系统支持线程优先级不同，建议使用上述三个优先级，不要自定义）

线程礼让：

Yield()方法：表示当前对象提示调度器自己愿意让出 CPU 资源，但是调度器可以自由的忽略该提示。

调用该方法之后，线程对象进入就绪状态，所以完全可能：某个线程调用了 yield()方法之后，线程调度器又把它调度出来重新执行。

从 java7 提供的文档上可以清楚的看出，开发中很少使用该方法，该方法主要用于调试或者测试，它可能有助于因多线程竞争条件下的错误重现现象。

Sleep 和 yield 方法的区别：

1) 都能使当前处于运行状态的线程放弃 cpu，把运行的机会给其他线程。
2) Sleep()方法会给其他线程运行机会，当不会考虑其他线程优先级，yield 只会给相同优先级或者更高优先级的线程运行机会
3) 调用 sleep()方法后，线程进入计时等待状态，调用 yield()方法后，线程进入就绪状态。

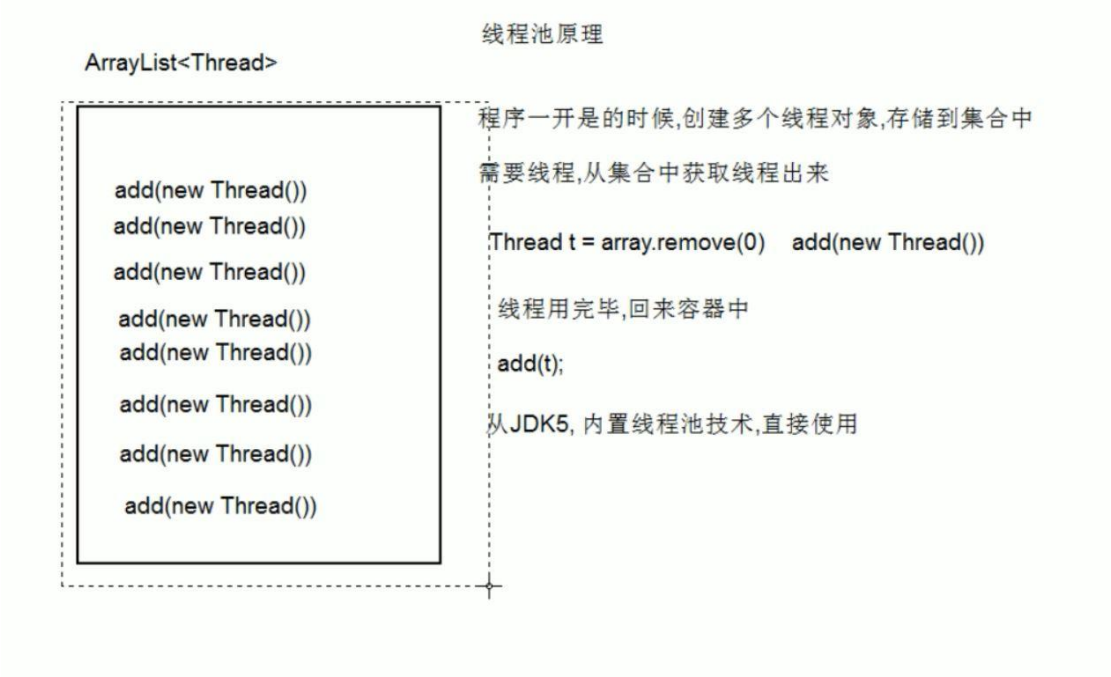
线程定时器：

使用 java.util.Timer;类,可以执行特定的任务

TimerTask 类表示定时器执行的某一项任务

ThreadGroup 类表示线程组，可以对一组线程进行集中管理
用户在创建线程对象时，可以通过构造器指定器所属的线程组。
Thread(ThreadGroup grouo,String name);
如果 A 线程创建了 B 线程，如果没有设置 B 线程的分组，那么 B 线程加入到 A 线程的线程组。一旦线程加入某个线程组，该线程就一直存在于该线程组中直到死亡，不能在中途修改线程的分组

当 java 程序运行时，JVM 会创建名为 main 的线程组，在默认情况下，所有的线程都在该线程组下。



Java 集合框架==数据结构的封装：

1. 数组(Array)
2. 栈(Stack)
3. 链表(Linked List)
4. 哈希表(Hash)
5. 队列(Queue)
6. 堆(Heap)
7. 图（Graoh）
8. 树(Tree)

队列：是一种特殊线性表，只允许再表的前端进行删除作者，在表后端进行插入操作。队头删除，队尾插入。
单向队列（Queue）：先进先出，队头删除，队尾插入。
双向队列（Deque）：两条反向队列。
（最擅长操作头和尾）
栈（stack）：堆栈，先进先出。（压栈，弹栈）
栈底索引为 0;
哈希表：散列表，在一般的数组中，元素在数组中的索引位置是随机的，元素的取值和元素的位置之间不存在确定的关系，因此在数组中查找特定的值时，需要把查找值和一系列的元

素进行比较。
此时查询效率依赖于查找过程中所进行的比较次数。
如果元素的值（value）和在数组中的索引位置（index）有一个确定的对应关系（hash）
Hash 算法：

公式为：index = hash(value);
那么对于给定的值,只要调用上述的hash(value)方,就能找到数组中取值为value的元素的位置.

元 素 值	11	22	33	44	55	66	77	88	99
元素的位置	0	1	2	3	4	5	6	7	8

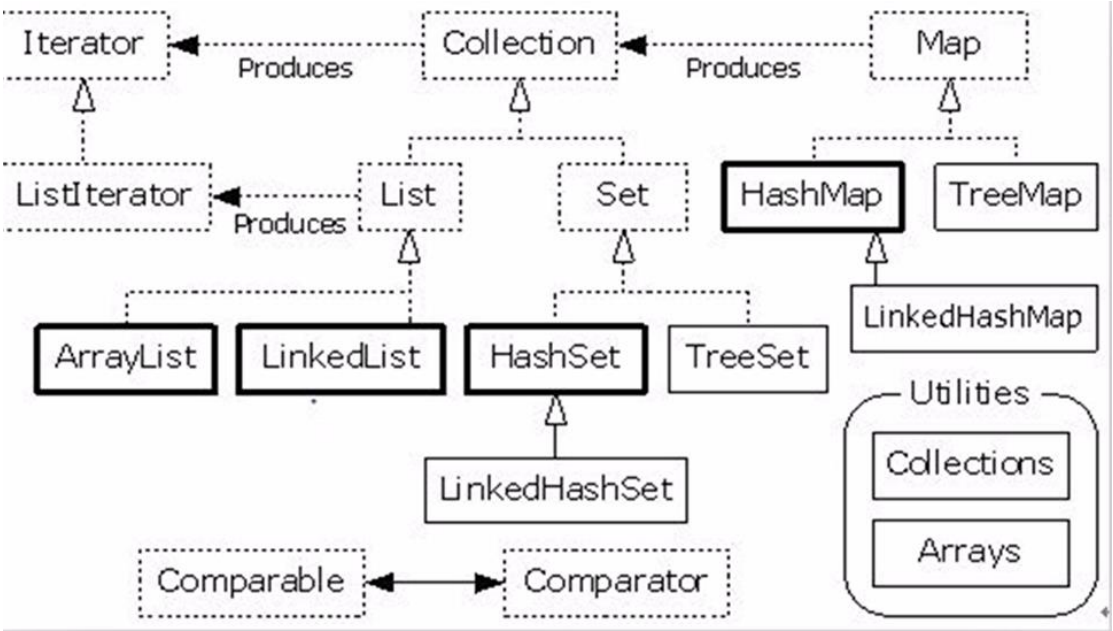
index = 元素值/10 -1;

如果数组中元素的值和索引位置存在对应的关系，这样的数组称之为哈希表，最大的优点时提供查找数据的效率。
一般情况下，我们不会把哈希码（hashCode）作为元素在数组中的索引位置的，因为哈希码很大，数组长度有限会造成索引越界问题。
元素值—hash(value)-→哈希码---某一种映射关系--→元素存储索引
注意：每个哈希码是不同的
哈希表的插入查询是很效率的。
可是当哈希表接近装满时，因为数组的扩容性问题，性能较低（转移到更大的哈希表中）。

数组是会记录添加顺序，按照索引位置来存储，允许元素重复。
哈希表中：元素是不能重复的，对象如果相同则 hashCode 相同→index 相同，不会添加记录元素的先后顺序。

基于数组的列表和基于链表的列表的性能对比：
ArrayList：查询，更改比较快，添加和删除比较慢。
LinkedList：查询，更改比较慢，添加和删除比较快。

常用的集合框架



No	接口	描述
1	Collection	是存放一组单值的最大接口，所谓的单值是指集合中的每个元素都是一个对象。一般很少会直接使用此接口直接操作。
2	List	是Collection接口的子接口，也是最常用的接口，此接口对Collection接口进行了大量的扩充，里面的内容是允许重复的。
3	Set	是Collection接口的子类，没有对Collection接口进行扩充，里面不允许存放重复内容。
4	Map	Map是存放一对值的最大接口，即，接口中的每个元素都是一对，以key→value的形式保存。
5	Iterator	集合的输出接口，用于输出集合中的内容，只能进行从前到后的单向输出。
6	ListIterator	是Iterator的子接口，可以进行双向输出。
7	Enumeration	是最早的输出接口，用于输出指定集合中的内容。
8	SortedSet	单值的排序接口，实现此接口的集合类，里面的内容是可以排序的，使用比较器排序。
9	SortedMap	存放一对值的排序接口，实现此接口的集合类，里面的内容按照key排序，使用比较器排序。
10	Queue	队列接口，此接口的子类可以实现队列操作。
11	Map.Entry	Map.Entry的内部接口，每个Map.Entry对象都保存着一对key→value的内容，每个Map接口中都保存多个Map.Entry接口实例。

Set（集）：集合中的对象不能按照特定方式排序，不允许元素重复。（无序，元素不重复）
List（列表）：集合中的对象按照索引位置排序，允许元素重复。（有序，元素重复）
Map（映射）：集合中每一个元素都包含一对 key 和 value 对象，不允许 key 对象重复，值对象可以重复。

Vector 类存储原理:

源码分析，vector 类中有一个 object[]类型数组

1、表面上把数据存储到 vector 对象中，其实底层依然是把数据存储到 object 数组中的

2、发现数组的元素类型是 object 类型，意味着集合中只能存储任意类型的对象。

集合中只能存储对象，不能存储基本数据类型的值。

3、集合类中存储的对象，都是存储对象的引用

栈（stack）：数据结构的一种存储特点：last in first out。

java.util.Vector<E>
└ java.util.Stack<E>

ArrayList 类是 java 集合框架之后取代 vector 类的:
二者底层原理一样
区别:

Vector：所有方法使用了 synchronized 修饰符　　线程安全性能较低，适用于多线程环境

ArrayList：没有使用同步方法　　线程不安全性能较高

即使以后多线程环境下，我们也不适用 Vector 类。
常用方法参照 Vector 类。
从源代码中分析，Vector 和 ArrayList 的差异大

有点时候某个方法需要返回一个 ArrayList 对象，但在该方法中，如果没有查到，我们不会返回 null，而是返回一个空集对象（没有元素集合）

在 java7 之前即使使用 new ArrayList 创建对象，一个元素都不存储，但是在堆空间依然初始化长度 10 的 Object 数组。从 java7 开始优化了这个设计，new ArrayList 其实底层创建的使用一个空数组。Object[] element=new Object[]{};

第一次调用 add 方法的时候才会重新初始化数组。

LinkedList 类是双向链表，单向队列，双向队列，栈的实现类：

LiinkedList 类实现单向队列和双向队列的接口，自身提高了栈操作的方法，链表的方法。

在 LinkedList 类中存在很多方法，但是功能都是相同的，LinkedList 表示了多种数据结构的实现，每一种数据结构的操作名字不同

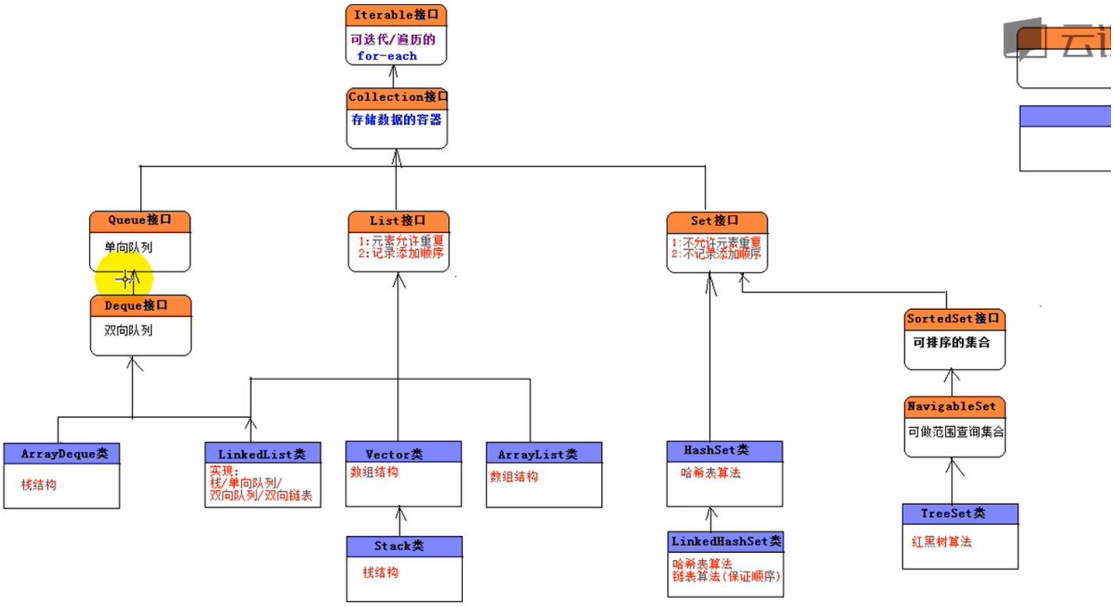
面试题：编写一个双向链表。

LinkedList 类是线程不安全的类，多线程环境下所有保证线程安全。

LinkedList linked=Collections.synchronizedList(new LinkedList(..));

无论是链表还是队列，都特别擅长操作头和为节点。

在 LinkedList 中存在 Object get(int index)，表示根据索引位置获取对应的元素。链表没有索引的概念，本不应该有索引，但是从 java2 开始存在了集合框架，让 LinkedList 类作为了 List 接口的实现类，List 提供了该根据索引查询元素的方法，LinkedList 内部类提供了一个变量来当作索引。该方法要少用，因为 LinkedList 不擅长做查询操作，擅长最多保存和删除操作。



List 实现类特点和性能分析：

三者相同点：

1、 允许元素重复

2、 记录元素的先后添加顺序

Vector：底层才有数组结构算法，方法使用 synchronized 修饰，线程安全，性能较低。

ArrayList：底层才有数组结构算法，方法没有使用 synchronized 修饰，线程不安全，性能较高。（ArrayList 现在已经取待 Vector）

为了保证 ArrayList 的线程安全， List list=Collections.synchronizedList(new ArrayList(…));(面向接口编程)

LinkedList：底层才有双向链表结构算法，方法没有使用同步方法，线程不安全。

数组结构算法和双向链表结构算法的性能问题：

数组结构算法：插入和删除操作速度低，查询和更改较快

链表结构算法：插入和删除操作速度快，查询和更改较慢

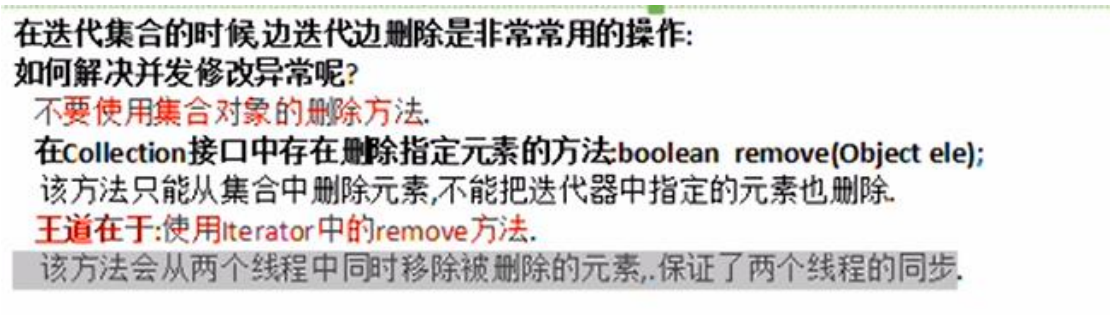
使用选择：

Vector 类不适用要使用选用 ArrayList 类

如果删除和插入操作频繁则使用 LinkedList 类
反正使用 ArrayList 类。

迭代器对象：
Iterator：迭代器对象，只能从上往下迭代。
ListIterator：是 Iterator 接口的子接口，支持双向迭代从上往下，或从下往上。
Enumeration：古老的迭代器对象，现已被 Iterator 取待，适用于 Vector 类。

深入 for-each 和迭代器：
1、 foreach 可以操作数组：底层依然采用 for 循环+索引来获取元素。
2、 foreach 可以操作 Iterator 的实例：底层其实采用的 Iterator（迭代器）
所以直接使用 foreach 迭代数组和集合元素即可。
当需要边迭代边删除指定集合中元素时：此时只能使用迭代器，并且只能迭代器 remove()方法。（不然会报多线程并发异常）



泛型：
< >:T type 类型，E Elements 元素，K key，V value 值
泛型类：直接在类/接口上定义的泛型。
Java7 开始推出泛型的菱形语法<>.
List<String> list=new ArrayList<>();
泛型不存在继承关系，使用泛型来约束集合中的元素类型（泛型是语法糖）底层也是类型强转。

泛型方法：在方法上声明泛型
情况 1：泛型类中的泛型只能适用于非静态方法，如果需要给静态方法设置泛型此时使用泛型方法
情况 2：泛型类中的泛型应该适用于整个类中多个方法，只对某一个方法设置泛型即可。
一般的，把自定义的泛型作为方法的返回类型才有意义，而且此时的泛型必须是由参数设置进来的。如果没有参数来设置泛型的具体类型，此时的方法一般返回设计为 Object 即可。

泛型的通配符、上限和下限：泛型使用通配符 ？ 时，只能接受数据，不能往集合中 储数据。

上限下限：用来限定元素的类型必须是 X 类的子类或相同，X 的父类或相同。上限 extends，下限 super。

泛型的擦除和转换：
擦除：
1、 泛型编译之后就会消失（泛型自动擦除）；
2、 当把带有泛型的集合给不带泛型的集合，此时泛型被擦除（手动擦除）；

堆污染：

单一个方法即使用泛型的时候也使用可变参数，此时容易导致堆污染问题。

如：在 Arrays 类中的 asList 方法。

Set 是 Collection 子接口，模拟了数学上的集的概念。

Set 只包含从 Collection 继承的方法，不过 set 无法记住添加顺序，不允许包含重复的元素。

当试图添加两个相同的元素进 set 集合，添加操作失败，add()方法返回 false

Set 判断两个对象是否相等用 equals，而不是==。也就是说两个对象 equals 比较返回 true，Set 集合是不会接受这两个对象的。

HashSet 是 set 接口最常用的实现类，顾名思义，底层采用了哈希表（散列表/hash）算法。其底层其实也是一个数组，存在的意义是提供查询速度。适用于少量数据的插入操作。

在 HashSet 如何判断两个对象是否相等：

1、 两个对象的 equals 比较相等，返回 true，则说明相同对象

2、 两对象的 hashCode 方法返回相等。

对象的 hashCode 值决定了在哈希表中的存储位置

二者缺一不可。

当往 HashSet 集合中添加新的对象的时候，先会判读该对象和集合对象中的 hashCode 值：

1)：不等：直接把该新对象存储到 hashCode 指定位置。

2)：相等：再继续判断新对象和集合对象中的 equals 作比较

1>:hashCode 相同，equals 为 true：则视为同一个对象，则不保存在哈希表中。

1>:hashCode 相同，equals 为 false：非常麻烦，存储在之前对象同槽为的链表上（拒绝，操作比较麻烦）

不同的数据类型如何来计算hashCode值:

Eclipse可以自动生成hashCode和equals方法.

Boolean	hashCode=(f?0:1)
整数类型(byte short int char)	hashCode=(int)f
long	hashCode=(int)(f^(f>>>32))
float	hashCode=Float.floatToIntBits(f)
double	long l = Double.doubleToLongBits(f); hashCode=(int)(l^(l>>>32))
普通引用类型	hashCode=f.hashCode()

自定义对象集合，为不添加重复对象进集合在自定义该类中重写 hashCode 方法和 equals 方法。

LinkedHashSet：底层是哈希算法和链表算法。

哈希表：维护元素的唯一性。 链表：添加元素添加的先后顺序。

TreeSet 集合底层才有红黑树算法会对存储的元素默认使用自然排序（从小到大）。

注意：必须保证 TreeSet 集合中的元素对象是相同的数据类型，否则会报错。

TreeSet 排序:

1、 自然排序: 调用集合元素的 compareTo 方法来比较元素的大小关系，然后讲集合元素按照升序排列（从小到大）

注意：要求 TreeSet 集合中元素的实现 java.util.Comparable 接口

Java.util.Comparable 接口可比较的

覆盖 compareTo 方法，该方法中的比较规则。(在该方法中，比较当前对象（this）和参数对象 o 作比较，严格说比较的是对象的数据)

在 TreeSet 的自然排序中，认为如果两个对象作比较的 compareTo 方法返回的是 0，则认为同一个对象。

2、 比较器排序:

在 Tree 构造器中传递，java.lang.Comparator 对象，并覆盖 compare()再编写比较规则。

对于 TreeSet 集合来说，要么使用自然排序，要么使用定制排序。

判断两对象是否相等的规则:

自然排序：compareTo 方法返回 0;

定制排序：comparae 方法返回 0;

Set 接口的实现类:

共同特点：不允许元素重复，线程不安全类。

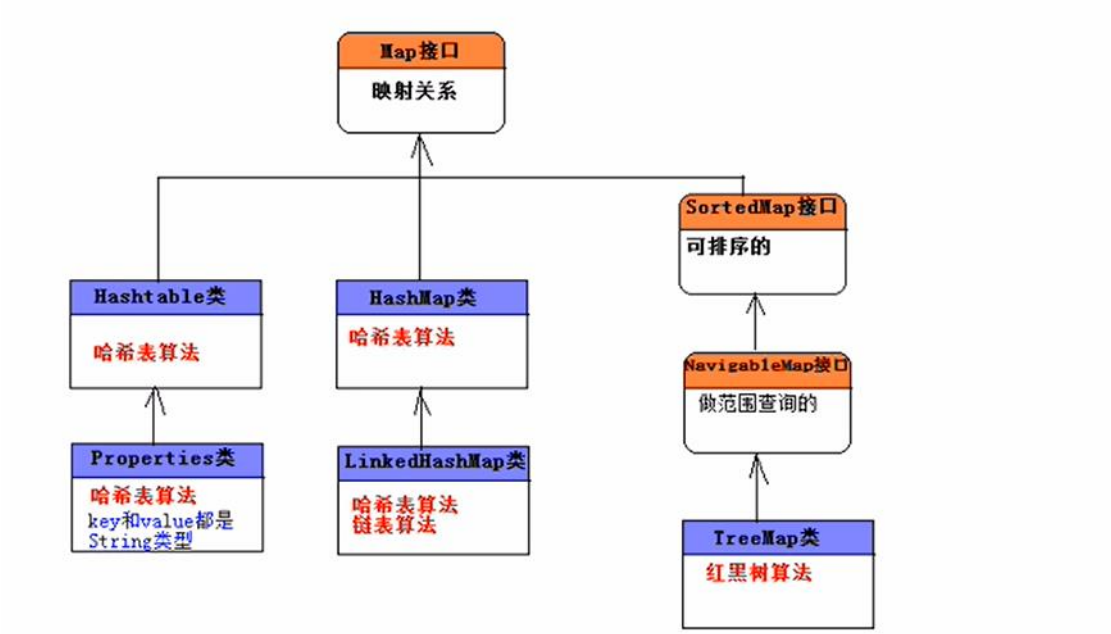
解决方法方案：Set set=Collections.sychrocizedSet(set 对象);

HashSet：不保证元素的先后顺序，底层采用哈希表算法，查询效率极高。

LinkedHashSet：HashSet 的子类，底层也采用的是哈希表算法，但是也使用了链表算法来维持元素的先后添加顺序。判断两个对象是否相等的规则和 HashSet 相同。因为需要多使用一个链表来记录元素的顺序，所以性能相对于 HashSet 较低。(一般少用)

TreeSet：不保证元素的先后添加顺序，但是会对集合中对的元素做排序操作。底层采用红黑树算法（树结构，比较擅长做范围查询)。TreeSet 要么采用自然排序，要么用定制排序。

Map 不是继承于 Collection 接口。



Map 的实现类 HashMap 和 Set 集合的实现类，源代码的算法相同，把 Set 的集合对象作为 Map 的 key，再使用一个 Object 常量为 value。

因此，可得 Map 中，所有的 key 就是 Set 集合。

Map 的常用实现类：

HashMap：采用哈希表算法，此时 Map 中的 key 不会保证添加的先后顺序，key 也不允许重复。Key 判断重复的标准是：key1 和 key2 是否 equals 相等，并且 hashCode 相等。

TreeMap：采用红黑树算法，此时 Map 中的 key 会按照自然顺序或定制排序进行排序，key 也不允许重复。Key 判断重复的标准：compareTo/compare 方法的返回值是否为 0.

LinkedHashMap：采用链表和哈希表算法，此时 Map 中的 key 会保证添加的顺序，key 不允许重复。Key 判断重复标准和 HashMap 的判断标准一样。

HashTable：采用哈希表算法，是 HashMap 的前身（类似于 Vector 是 ArrayList 的前身）。

Properties：是 Hashtable 的子类，此时要求 key 和 value 都是 String 类型。（用来加载资源文件）

HashMap 和 TreeMap 已经 LinkedHashMap 都是线程不安全的，但是性能较高；

解决方案：Map map=Collections.synchronizedMap(Map 对象);

Hashtable 类是线程安全的，但性能较低。

哈希表算法：做等值查询最快　　树结构算法：做范围查询最快→应用到索引上。

选用哪一种容器取决于每一种容器的存储特点以及当前业务的需求：

List：单一元素集合。允许元素重复/记录元素的添加顺序。

Set：单一元素集合。不允许元素重复/不记录元素的添加顺序。

既要不重复有，又要保证先后顺序：LinkedHashSet。

Map：双元素集合，如果存储数据的时候，还得给数据其为一个的一个名称，次数考虑使用 Map。

List 和 Set 以及 Map 之间相互转换问题：

List<String> list=new ArrayList<>();

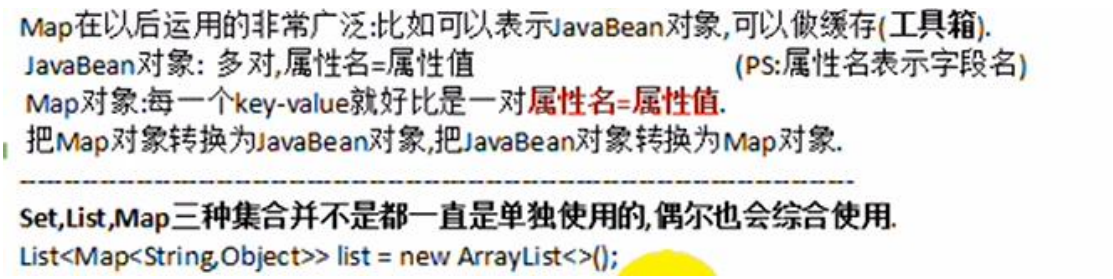
把 List 转换为 Set：

Set<String> set=new HashSet<>(list);//此时会消除重复的元素。

把 Set 转换为 List：

List<String> list2=new ArrayList<>(set);

Map 不能直接转换为 List 或 Set（但是 Map 中的方法可以间接转换 keySet()方法）



1):Arrays类:
2):Collections类.

Arrays类:

在Collection接口中有一个方法叫toArray把集合转换为Object数组.

把集合转换为数组: **Object[] arr = 集合对象.toArray();**

数组也可以转换为集合(List集合):

public static <T> List<T> asList(T... a) 等价于public static <T> List<T> asList(T[] a).

```
//把数组转换为List对象
List<String> list = Arrays.asList("A","B","C","D");
List<Date> list2 = Arrays.asList(new Date(),new Date());
```

通过Arrays.asList方法得到的List对象的长度是固定的,不能增,也不能减.

为什么: asList方法返回的ArrayList对象,不是java.util.ArrayList而是Arrays类中的内部类对象.

```
//可以自动装箱,把1看成是Integer对象
List<Integer> list1 = Arrays.asList(1,2,3,4,5);
//定义了int类型的数组,
int[] arr2={1,2,3,4,5};
//直接把数组当做是对象
List<int[]> list2 = Arrays.asList(arr2);
System.out.println(list2);
```

面试题:

Collections 和 Collection 区别:

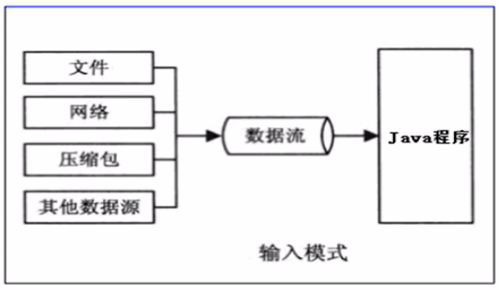
Collection 类：封装了 Set，List，Map 的操作的工具方法。

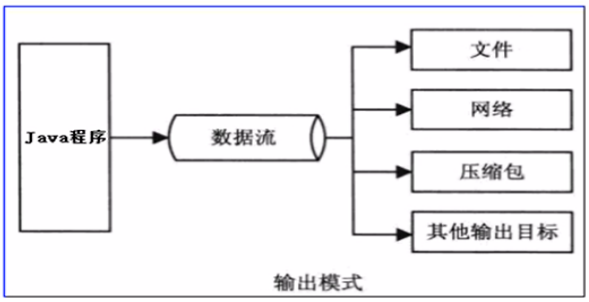
获取空集对象（不为 null，没有元素）

常用集合：HashSet/ArrayList/HashMap 都是线程不安全的，在多线程环境下不安全。

在 Collections 类中有获取线程安全的集合方法（同步方法）

使用迭代的时候还要使用 synchronized。





	字节流	字符流
输 出 流	OutputStream 字节输出流	Writer 字符输出流
输 入 流	InputStream 字节输入流	Reader 字符输入流

IO 流分类和操作模板:

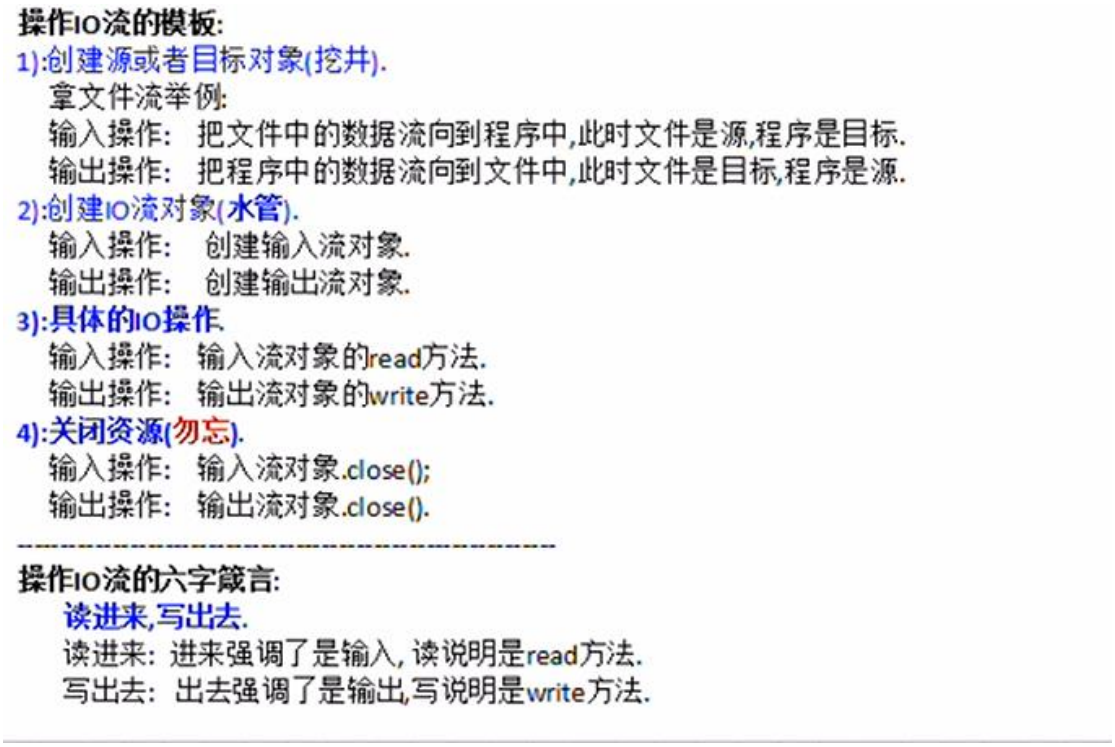
1)：根据流向分为：输出流和输入流。

2)：根据数据单位划分：字节流和字符流。

3)：根据功能划分：节点流和包装流。

四大基本流（字节输出流、输入流，字符输出流、输入流）

四大基本流都是抽象类：其他流都是继承于这四大基本流的。



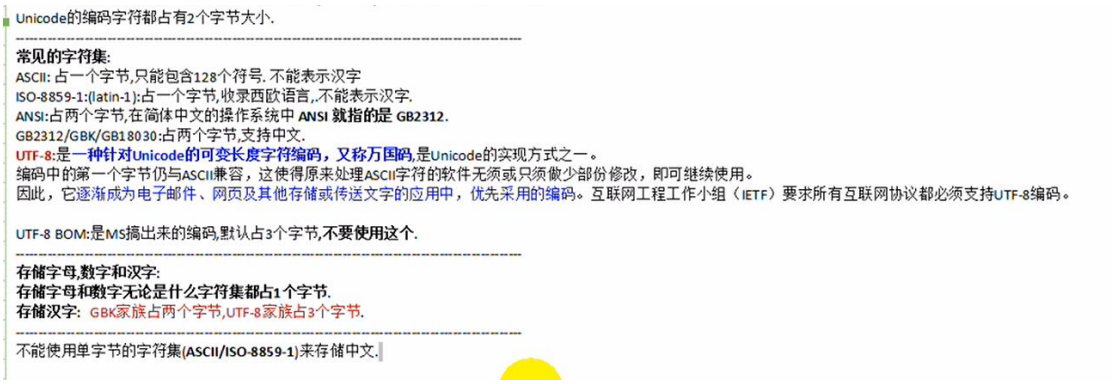
文件流：

FileInputStream：文件字节输入流

FileOutputStream：文件字节输出流

FileReader：文件的字符输入流

FileWriter：文件的字符输出流



字符集的编码和解码操作：

编码：把字符串转换为 byte 数组。

解码：把 byte 数组转换为字符串。

一定要保证编码和解码的字符串相同，否则会乱码。

处理流/包装流相对于节点流更高级（装饰设计模式/包装设计模式）：

1、 隐藏了底层的节点流的差异，并对外提供了更方便的输入/输出功能，让我们只关心高级流的操作

2、 使用处理流包装了节点流，程序直接操作处理流，让节点流与底层流的设备做 IO 操作。

3、 只需要关闭处理流即可。

包装流如何区分：写代码的时候，发现创建对象的时候，需要传递另一个流对象。

New 包装流(流对象)

缓冲流：是一个包装流，目的起缓存作用。

BufferedInputStream ： A BufferedInputStream 为另一个输入流添加了功能，即缓冲输入和支持 mark 和 reset 方法的功能。

BufferedOutputStream：该类实现缓冲输出流。

BufferedReader ：从字符输入流读取文本，缓冲字符，以提供字符，数组和行的高效读取。

BufferedWriter ：将文本写入字符输出流，缓冲字符，以提供单个字符，数组和字符串的高效写入。

转换流：把字节流转换成字符流

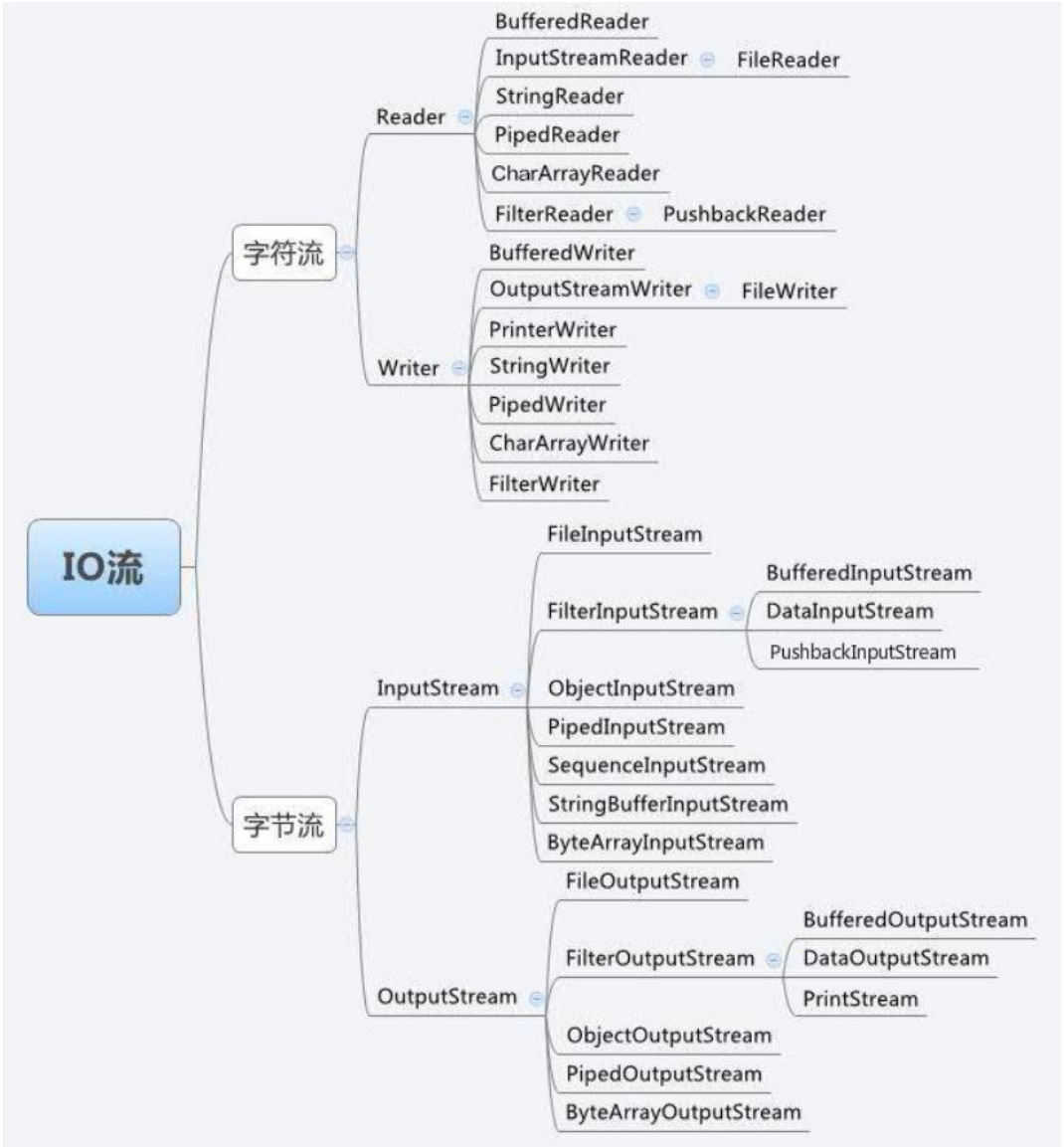
InputStreamReader：把字节输入流转换成字符输出流

OutputStreamWriter：把字节输出流转换成字符输出流

为什么有字节转字符，没有字符转字节流。

字节流可以操作一切文件（纯文本/二进制文件）。

字符流是用来操作中文纯文本使用的，本身是对字节流得增强。



内存流（数组流）适配器模式：
把数据先临时存储到数组中，待会再从数组中获取出来。
1)：字节内存流：ByteArrayInputStream/ByteOutputStream
2)：字符内存流：CharArrayReader/CharArrayWriter
3)：字符串流：StringReader/StringWriter （把数据临时存储到字符串中）
StringBufferedInputStream

序列化和反序列化：
序列化：指把堆内存得 java 对象数据，通过某种方式把对象存储到磁盘文件中或者传递给其他网络的节点（在网络上传输），我们把这个过程称之为序列化
反序列或：把磁盘文件中的对象数据或者把网络节点的数据对象，恢复成 Java 对象的过程。
为什么需要序列化：
1、在分布式系统中，需要共享的数据的 JavaBean 对象，都得作序列化，此时需要把对象再网络上传输，此时就得把对象数据转换为二进制形式。
储存在 HttpSession 中的对象，都应该实现序列化接口（只有实现序列化接口的类才能作序列化操作）
2、服务钝化：如果服务发现某些对象好久都没有活动了，此时服务器就会把这些内存中的对象，持久化在本地磁盘文件中（java 对象-→二进制）。
如果某些对象需要活动的时侯，现在内存中去寻找，找到就是用，找不到再去磁盘文件中，反序列化我们得对对象数据，恢复成 java 对象。
需要作序列化的对象的类必须实现序列化接口：java.io.Serializalble 接口（标志接口[没有抽象方法]）底层会判断，如果对象是 Serializable 的实例，才允许做序列化。 Booleam ret=java 对象 instanceof Serializable;
序列化和反序列化类：
ObjectOutputStream：序列化
ObjectInputStream：反序列化
序列化的细节序列化的版本：
1、如果某些数据不需要做序列化，比如密码，此时怎么办？
理论上说，静态的字段和瞬态的字段是不能做序列化操作的
Transient private String password;
（它可以调用 in.defaultReadObject 来调用恢复对象的非静态和非瞬态字段的默认机制）
2、序列化版本问题：
反序列化 java 对象的必须提供该对象的 class 文件，现在问题是，随着项目的升级，系统的 class 文件也会升级（增加一个字段/删除一个字段），如何保证两个 class 文件的兼容性？java 通过 serialVersionUID（版本序列号）来判断字节码是否发生改变。Java.io.InvalidClassException，如果不显示定义 serialVersionUID 类变量，该类通过变量的值由 JVM 根据类相关信息计算，而修改后的类的计算方式和之前往往不同。从而造成了对象反序列化因为版本不兼容而失败的问题。
解决方案：在操作对象的类中提供一个固定的 serialVersionUID。

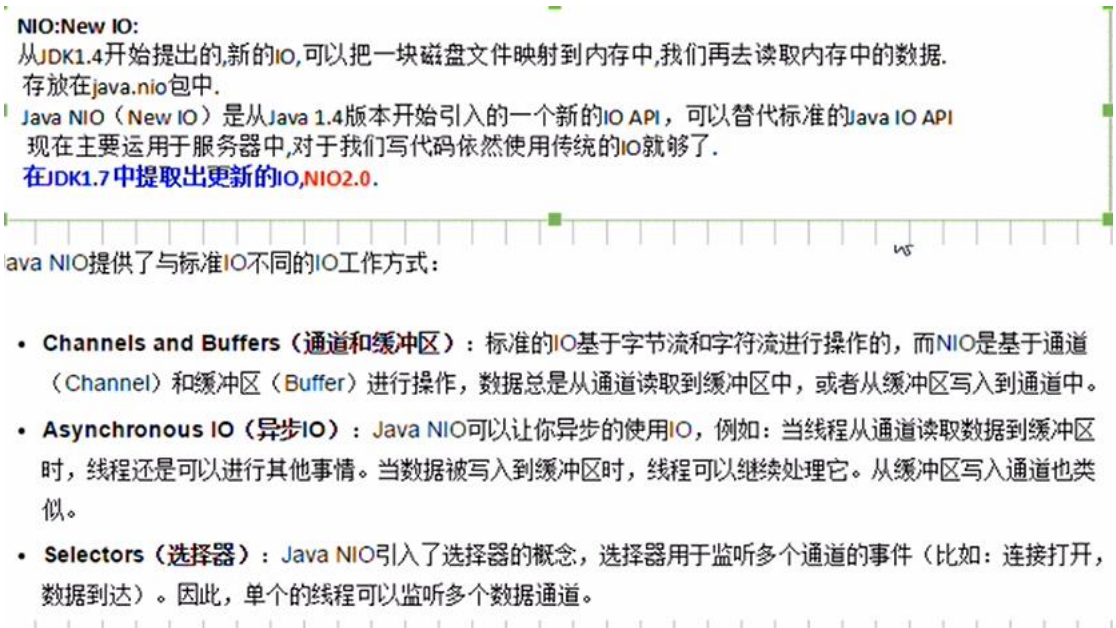
打印流，打印数据的，打印流只能是输出流：
PrintStream：字节打印流
PrintWriter：字符打印流
-对于 PrintWriter 来说，当启用字段刷新之后，

PrintWriter ps=new PrintWriter(new FileOutputStream(new File("XXX")),true);调用 println 或者 format 方法，便会立马刷新操作。
如果没有开启自动刷新，则需要手动刷新或者当缓冲去满，再自动刷新。
使用打印流作为输出流，此时的输出操作会特别简单，因为再打印流缓冲：
　　提供了 print 方法：打印不换行
　　提供了 println 方法：打印在换行
配置文件：资源文件（以.properties 作为拓展名的文件）/属性文件：

现在数据库的连接信息，再 db.properties 文件中，而 Java 代码需要获取该文件中的信息
重心转移：java 代码如何加载 properties 文件，必须使用 Properties 类（Hashtable 子类，Map 接口实现类）

数据流，提供了可以读/写任意数据类型的方法：
DataOutputStream：数据输出流
DataInputStream：数据输入流
注意：数据类型的读写的方法需要同一数据类型。

随机访问文件（RandomAccessFile）：表示在文件任意出读写。有一种游标，或索引到隐含的数组，称为文件指针 ；输入操作读取从文件指针开始的字节，并使文件指针超过读取的字节。
RandomAccessFile 经常用来做多线程断点下载：
1、 多线程
2、 断点下载
管道流：实现两个线程之间的数据交互。
PipedInputStream
PipedOutoutStream
PipedReader
PipedWriter
Java.nio 包里的



常用的 IO 流：

四大基流:

InputStream---- OutputStream

Reader --- Writer

IO流的总结和梳理:

文件流:

FileInputStream
FileOutputStream
FileReader
FileWriter

缓冲流:

BufferedInputStream
BufferedOutputStream
BufferedReader
BufferedWriter

转换流(把字节转换为字符):

InputStreamReader
OutputStreamWriter

内存流(临时存储数据):

ByteArrayInputStream
ByteArrayOutputStream
CharArrayReader
CharArrayWriter
StringReader
StringWriter

顺序流(合并流):

SequenceInputStream

对象流(序列化和反序列化):

ObjectInputStream
ObjectOutputStream

打印流:

PrintStream
PrintWriter

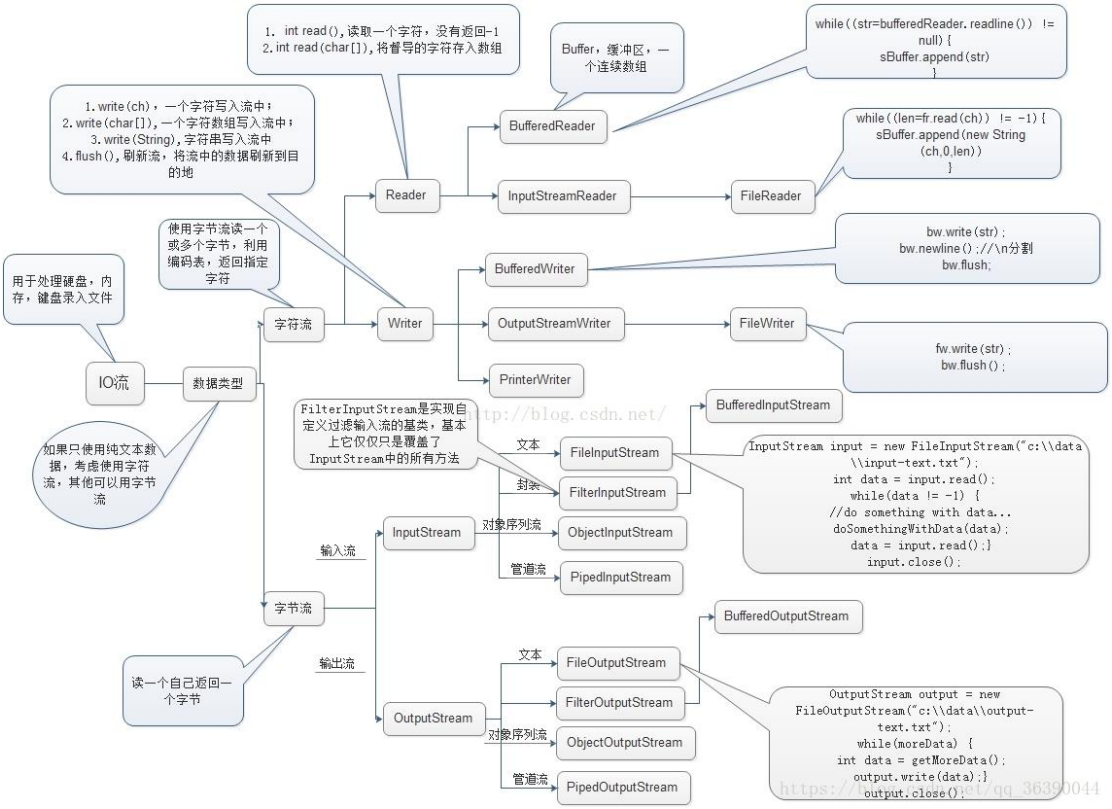
数据流:

DataInputStream
DataOutputStream

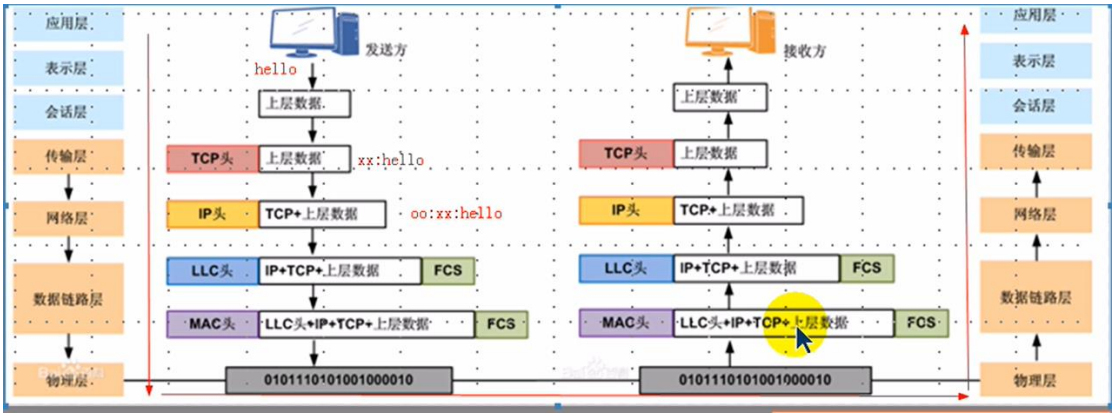
管道流:

PipedInputStream
PipedOutputStream
PipedReader
PipedWriter

File
FilenameFilter
RandomAccessFile
Files



网络分层：



应用层协议：FTP/HTTP/SMTP/DNS/WWW/NFS
表示层协议：JPEF/MPEG/ASII
会话层协议：NFS/SQL/RPC
传输层协议：TCP/UDP/SPX
网络层协议：IP/ICMP/ARP/RARP/RIP
数据链路层协议：PPP/VLAN/MAC
物理层协议：CLOCK/RJ45

网络编程：
什么是套接字：
源 IP 地址和目的 IP 地址以及源端口号和目的的端口号的组合称为套接字。启用于标识符客户端的请求的服务器和服务。
什么是网络编程：
使用套接字来达到进程间通信的目的编程就是网络编程。
Java.net 包。
网络编程三要素：
1、 IP 地址
2、 端口号
3、 协议，数据传递/交互规则

UDP 协议：

UDP 是一种面向无连接的协议，因此，在通信时发送端和接收端不用建立连接。UDP 通信的过程就像是货运公司在两个码头间发送货物一样。在码头发送和接收货物时都需要使用集装箱来装载货物，UDP 通信也是一样，发送和接收的数据也需要使用“集装箱”进行打包，为此 JDK 中提供了一个 DatagramPacket 类，该类的实例对象就相当于一个集装箱，用于封装 UDP 通信中发送或者接收的数据。

UDP 是无连接通信协议，即在数据传输时，数据的发送端和接收端不建立逻辑连接。简单来说，当一台计算机向另外一台计算机发送数据时，发送端不会确认接收端是否存在，就会发出数据，同样接收端在收到数据时，也不会向发送端反馈是否收到数据。

由于使用 UDP 协议消耗资源小，通信效率高，所以通常都会用于音频、视频和普通数据的传输例如视频会议都使用 UDP 协议，因为这种情况即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。
但是在使用 UDP 协议传送数据时，由于 UDP 的面向无连接性，不能保证数据的完整性，因

此在传输重要数据时不建议使用 UDP 协议。

TCP 协议：

TCP 协议是面向连接的通信协议，即在传输数据前先在发送端和接收端建立逻辑连接，然后再传输数据，它提供了两台计算机之间可靠无差错的数据传输。在 TCP 连接中必须要明确客户端与服务器端，由客户端向服务端发出连接请求，每次连接的创建都需要经过“三次握手”。第一次握手，客户端向服务器端发出连接请求，等待服务器确认，第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求，第三次握手，客户端再次向服务器端发送确认信息，确认连接。

由于 TCP 协议的面向连接特性，它可以保证传输数据的安全性，所以是一个被广泛采用的协议，例如在下载文件时，如果数据接收不完整，将会导致文件数据丢失而不能被打开，因此，下载文件时必须采用 TCP 协议。

TCP 通信同 UDP 通信一样，都能实现两台计算机之间的通信，通信的两端都需要创建 socket 对象。

区别在于，UDP 中只有发送端和接收端，不区分客户端与服务器端，计算机之间可以任意地发送数据。

而 TCP 通信是严格区分客户端与服务器端的，在通信时，必须先由客户端去连接服务器端才能实现通信，服务器端不可以主动连接客户端，并且服务器端程序需要事先启动，等待客户端的连接。

在 JDK 中提供了两个类用于实现 TCP 程序，一个是 ServerSocket 类，用于表示服务器端，一个是 Socket 类，用于表示客户端。

通信时，首先创建代表服务器端的 ServerSocket 对象，该对象相当于开启一个服务，并等待客户端的连接，然后创建代表客户端的 Socket 对象向服务器端发出连接请求，服务器端响应请求，两者建立连接开始通信。

URI:

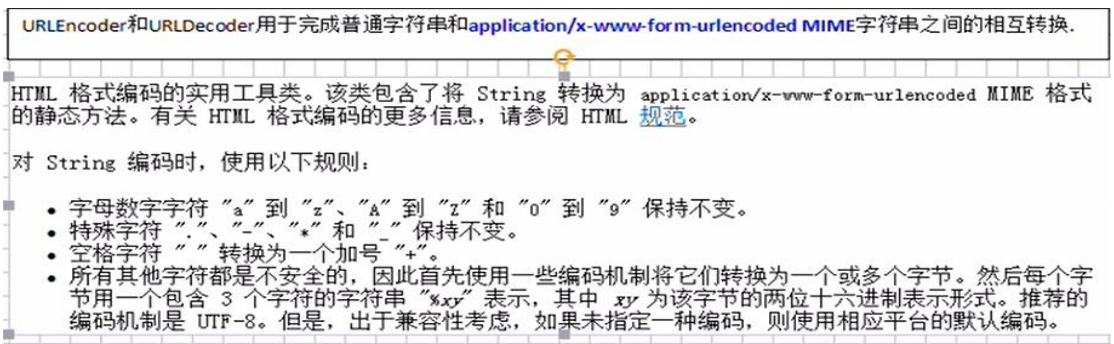
统一资源标识符（UniformResource Identifler，或 URI）是一个用于标识符某一互联网资源名称的字符串。包换：主机名，标识符，相对于 URI。

URL:

统一资源定位符石对可以从互联网上得到的资源位置和访问方法的一种简洁的表示，石互联网上标准资源的地址。互联网上的每个文件都有一个唯一的 URL，它包含的信息指出文件的位置以及浏览器应该怎么处理它。

在 java 中，URI 表示一个统一资源的标识符，不能用于定位任何资源，唯一的作用就是解析。而 URL 则包含有一个可以打开达到该资源的输入流，可以简单理解 URL 是 URI 的特例

编码和解码：


URLEncoder和URLDecoder用于完成普通字符串和application/x-www-form-urlencoded MIME字符串之间的相互转换.

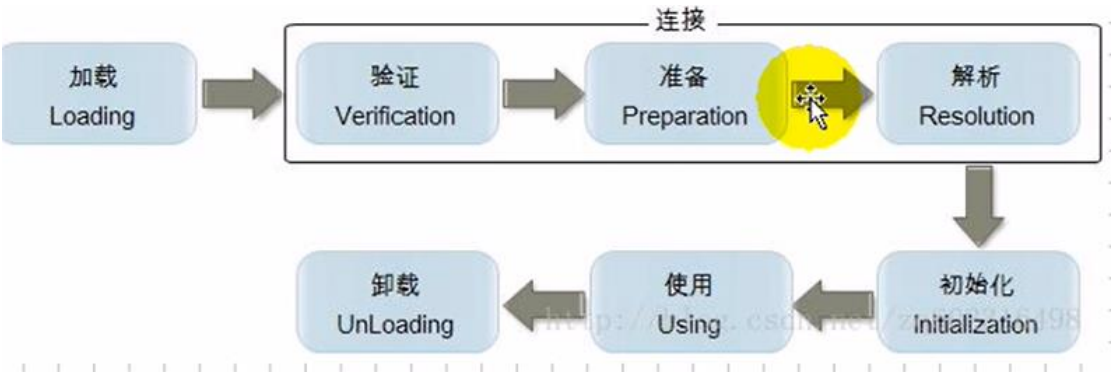
HTML 格式编码的实用工具类。该类包含了将 String 转换为 application/x-www-form-urlencoded MIME 格式的静态方法。有关 HTML 格式编码的更多信息，请参阅 HTML 规范。

对 String 编码时，使用以下规则：

- 字母数字字符 "a" 到 "z"、"A" 到 "Z" 和 "0" 到 "9" 保持不变。
- 特殊字符 "."、"-"、"*" 和 "_" 保持不变。
- 空格字符 " " 转换为一个加号 "+"。
- 所有其他字符都是不安全的，因此首先使用一些编码机制将它们转换为一个或多个字节。然后每个字节用一个包含 3 个字符的字符串 "%xy" 表示，其中 xy 为该字节的两位十六进制表示形式。推荐的编码机制是 UTF-8。但是，出于兼容性考虑，如果未指定一种编码，则使用相应平台的默认编码。

在 web 的浏览器中，不同的浏览器编码和解码规则是不一样的。
对于 w3c 浏览器：遵循 w3c 组织规范的浏览器。（非 IE）
传输层协议：
TCP 和 UDP 的区别：
TCP：面向连接（经历三次握手）传输可靠（保证数据正确性，保证数据顺序）、用于传输大量数据（流模式）、速度慢，建立连接需要多开销（时间，系统资源）。
UDP：面向非连接，传输不可靠、用于传递少量数据（数据包模式）、速度快。

反射机制：



类加载器：
当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过加载，连接，初始化三步来实现对这个类进行初始化。

- 加载

就是指将 class 文件读入内存，并为之创建一个 Class 对象。
任何类被使用时系统都会建立一个 Class 对象

- 连接

验证 是否有正确的内部结构，并和其他类协调一致
准备 负责为类的静态成员分配内存，并设置默认初始化值
解析 将类的二进制数据中的符号引用替换为直接引用

- 初始化

就是我们以前讲过的初始化步骤

初始化：
1. 创建类的实例
2. 类的静态变量，或者为静态变量赋值
3. 类的静态方法
4. 使用反射方式来强制创建某个类或接口对应的 java.lang.Class 对象
5. 初始化某个类的子类
6. 直接使用 java.exe 命令来运行某个主类

符号引用：
符号引用时一个字符串，它给出了被引用的内容的名字并且可能会包换一些其他关于这个被引用的项的信息—这些信息必须足够以唯一的识别一个类、字段、方法。
这样，对于其他类的符号引出必须给出类的全名。

Class 类的类表示正在运行的 Java 应用程序中的类和接口。 枚举是一种类，一个注释是一种界面。 每个数组也属于一个反映为类对象的类，该对象由具有相同元素类型和维数的所有数组共享。

- 获取 Class 对象的三种方式

方式一：通过 Object 类中的 getObject()方法

```
Person p = new Person();
Class c = p.getClass();
```

方式二：通过 类名.class 获取到字节码文件对象（任意数据类型都具备一个 class 静态属性,看上去要比第一种方式简单）。

```
Class c2 = Person.class;
```

方式三：通过 Class 类中的方法（将类名作为字符串传递给 Class 类中的静态方法 forName 即可）。

```
Class c3 = Class.forName("Person");
```

问题:在上述讲了三种获取Class对象的方式,基本数据类型不能表示为对象,也就不能使用getClass的方式,基本类型没有类名的概念,也不能使用Class.forName的方式,**如何表示基本类型的字节码对象呢?**

所有的数据类型都有class属性.

Class clz = 数据类型.class;

九大内置Class实例 JVM中预先提供好的Class实例,分别:byte,short,int,long,float,double,boolea,char,void.

表示:byte.class,short.class,int.class,....void.class.

在8大基本数据类型的包装类中,都有一个常量:TYPE,用于返回该包装类对应基本类的字节码对象.

System.out.println(Integer.TYPE == int.class);//true

注意:Integer和int是不同的数据类型

System.out.println(Integer.class == int.class);//false

数组的Class实例:数组是引用数据类型,数组其实是对象.

如何来表示数组的Class实例

方式1: 数组类型.class;

方式2: 数组对象.getClass();

注意:所有的具有相同的维数和相同元素类型的数组共享同一份字节码对象,和元素没有关系.

Class: 描述所有的类型,所以Class类中应该具有所有类型的相同的方法.

Object: 描述所有的对象,所以在Object类中应该具有所有对象的共同的方法.

Class类获取构造器方法:

Constructor类:表示类中构造器的类型,Constructor的实例就是某一个类中的某一个构造器

public Constructor<?>[] getConstructors():该方法只能获取当前Class所表示类**的public修饰的构造器**

public Constructor<?>[] getDeclaredConstructors():获取当前Class所表示类的所有的构造器,**和访问权限无关**

public Constructor<T> getConstructor(Class<?>... parameterTypes) :获取当前Class所表示类中指定的一个public的构造器

参数:parameterTypes表示:构造器参数的Class类型

如:public User(String name)

Constructor c = clz.getConstructor(String.class);

public Constructor<T> getDeclaredConstructor(Class<?>... parameterTypes) :获取当前Class所表示类中指定的一个的构造

构造器最大的作用:创建对象.
为什么使用反射创建对象,为什么不直接来new呢?
在框架中,提供给我们的都是字符串.

使用反射创建对象:
步骤:
1);找到构造器所在类的字节码对象.
2):获取构造器对象.
3):使用反射,创建对象

Constructor<T>类:表示类中构造器的类型,Constructor的实例就是某一个类中的某一个构造器
常用方法:
public T newInstance(Object... initargs):如调用带参数的构造器,只能使用该方式.
参数:initargs:表示调用构造器的实际参数
返回:**返回创建的实例**,T表示Class所表示类的类型

如果:一个类中的构造器是外界可以直接访问,同时没有参数.,那么可以直接使用Class类中的newInstance方法创建对象.
public Object newInstance():相当于new 类名();
调用私有的构造器:

使用反射获取类中的方法:
1):获取方法所在类的字节码对象.
2):获取方法.

Class类中常用方法:
public Method[] getMethods():获取包括自身和继承过来的所有的**public方法**
public Method[] getDeclaredMethods():获取自身类中所有的方法(不包括继承的,和访问权限无关)

public Method getMethod(**String methodName**,
Class<?>... parameterTypes):表示调用指定的一个公共的方法(包括继承的)
参数:
methodName: 表示被调用方法的名字
parameterTypes:表示被调用方法的参数的Class类型如String.class

public Method getDeclaredMethod(String name,
Class<?>... parameterTypes):表示调用指定的一个本类中的方法(不包括继承的)
参数:
methodName: 表示被调用方法的名字
parameterTypes:表示被调用方法的参数的Class类型如String.class

加载资源文件路径:
db.properties
注意:加载properties文件,只能使用Properties类的load方法.

方式1:使用绝对路径的方式加载.该方式不可行.

```
//方式1:使用绝对路径的方式加载.
private static void test1() throws Exception {
    Properties p = new Properties();
    InputStream inStream = new FileInputStream("H:/JavaApps/反射机制/resources/db.properties");
    p.load(inStream);//加载
    System.out.println(p);
}
```

方式2:使用相对路径-相对于classpath的根路径(字节码输出目录).
此时得使用ClassLoader(类加载器),类加载器默认就是从classpath根路径去寻找文件的.

```
//方式2:使用相对路径-相对于classpath的根路径(字节码输出目录).
private static void test2() throws Exception {
    Properties p = new Properties();
    //ClassLoader loader = LoadResourceDemo.class.getClassLoader();
    ClassLoader loader = Thread.currentThread().getContextClassLoader();
    InputStream inStream = loader.getResourceAsStream("db.properties");
    p.load(inStream);//加载
    System.out.println(p);
}
```

方式3:使用相对路径-相对于当前加载资源文件的字节码路径

```
//方式3:使用相对路径-相对于当前加载资源文件的字节码路径
private static void test3() throws Exception {
    Properties p = new Properties();
    InputStream inStream = LoadResourceDemo.class.getResourceAsStream("db.properties");
    p.load(inStream);//加载
    System.out.println(p);
}
```

在这里使用的是LoadResourceDemo的字节码路径去寻找db.properties文件.

也就是从bin/com/_520i/_06_load_resource路径去寻找.

Java8 新特性:

Lambda 表达式:

1，lambda的历史：λ（波长单位），带有参数变量的表达式称为lambda表达式；
2，lambda表达式的基本语法：参数列表 -> 表达式；
  1，参数列表：
    1，如果没有参数；直接用()来表示；()不能省略；
    2，如果只有一个参数，并且参数写了类型，参数外面一定要加()；
    3，如果只有一个参数，并且参数不写类型，那么这个参数外面可以不用加()；
    4，如果有两个或多个参数，不管是否写参数类型，都要加()；
    5，如果参数要加修饰符或者标签，参数一定要加上完整的类型；
  2，表达式：
    1，如果表达式只有一行，那么可以直接写（不需要{}）；
    2，如果表达式有多行，需要用{}变成代码块；
    3，如果表达式是代码块，并且方法需要返回值，那么，在代码块中就必须返回一个返回值；
    4，如果只有单行的情况，并且方法需要返回值，不能有return，编译器会自动帮我们推导return；

lambda表达式中的变量：
1，参数；
2，局部变量；
3，自由变量（不是参数也不是局部变量）；
结论：lambda表达式中的自由变量会被保存，无论什么时候执行lambda表达式，都可以直接使用；
1，自由变量在lambda表达式中是不能修改（final）；（操作自由变量的代码块，称为闭包；）
2，参数和局部变量的使用方式和普通的变量使用方式相同；
3，lambda表达式中的this指向，创建lambda表达式的方法中的this;【重要】

函数式接口：
1，我们能够写lambda表达式的地方？一个接口，且接口里面只有一个抽象方法；
2，在Java中，把只有一个抽象方法的接口称为函数式接口，如果一个接口是函数式接口，我们可以在接口上添加@FunctionalInterface标签标明这是一个函数式接口；（得到：1，检查；2，文档）
3，无论是否标示@FunctionalInterface，只要满足函数式接口的接口，Java都会直接识别为函数式接口；
4，简化函数式接口的使用是JAVA中提供lambda表达式唯一的作用；
5，可以用接口直接来引用一个lambda表达式；
6，函数式接口里面可以写Object对象中的方法；
7，lambda表达式中的异常处理：lambda表达式中产生的异常要么直接在代码块中处理，要么接口的方法声明抛出；