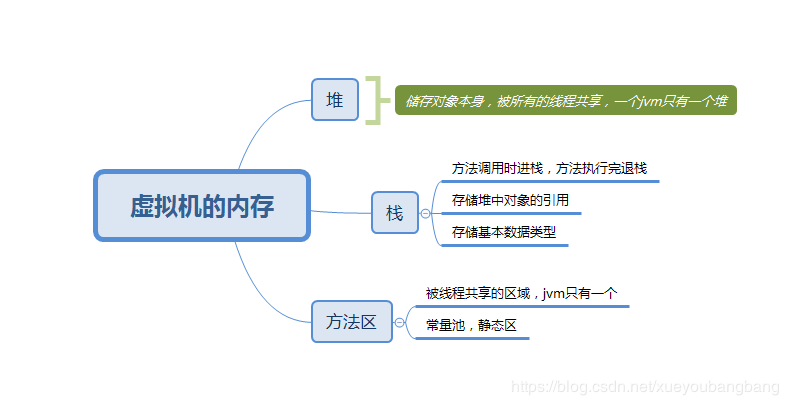
## 堆和栈

栈的效率高.由操作系统自动分配释放,在硬件层级对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行

按照先后定义的顺序依次压入栈中，也就是说相邻变量的地址之间不会存在其它变量。栈的内存地址生长方向与堆相反，由高到底，后定义的变量地址低于先定义的变量。栈中存储的数据的生命周期随着函数的执行完成而结束

栈是运行时单位，用来解决程序运行时的问题，堆是存储单位，解决数据存储的问题。



堆: (heap)

1.存储的全部是对象，每个对象都包含一个与之对应的class的信息。(class的目的是得到操作指令)

2.只有一个堆,伴随着JVM的启动而创建。

3.堆的申请和释放工作由程序员控制，容易产生内存泄漏--己动态分配的堆内存未释放或无法释放

栈区:

1.每个线程一个栈，栈中只保存基本数据类型和自定义对象的引用(不存放对象)

2.每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。

3.栈分为3个部分：基本类型变量区、执行环境上下文、操作指令区(存放操作指令)

方法区:



1.又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class文件信息和static变量

2.方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。

常量池

1.常量池中包含基本类型和对象型的常量值（如String及数组）。

　　1.1 这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另 外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值小于等于127时才可使用常量池，也即对象不负责创建和管理大于127的这些类的对象。大于127的对象存放在堆中。

　　1.2 String类也实现了常量池技术

2.常量池还包含一些以文本形式出现的符号引用，比如：

　　2.1 类和接口的全限定名；

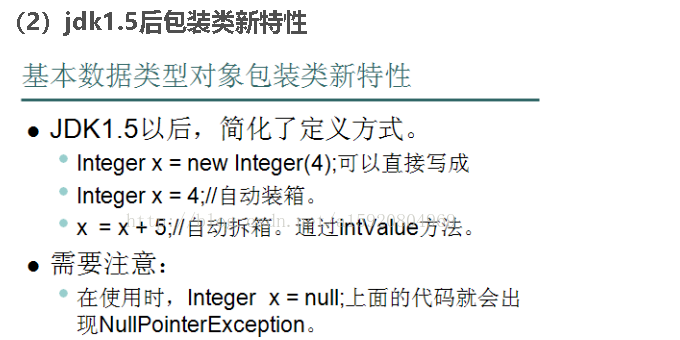
　　2.2 字段的名称和描述符；

　　2.3 方法和名称和描述符。

### 拆箱装箱JDK1.5

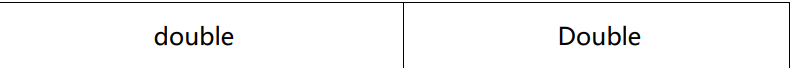
装箱：基本数据类型🡪包装器类型 valueOf方法

拆箱：包装器类型🡪基本数据类型 xxxValue方法



基本数据类型和包装类





基本数据类型不是面向对象（没有属性、方法），实际使用时存在很多的不便（比如集合的元素只能是 Object）。所以需要包装类。

## static非 static 变量区别

1. static变量只有1份, 被该类创建的所有对象共享, 也被称为类变量

非static变量有多个, 也被称为实例变量

2. static变量存在方法区中，实例变量存在堆内存中

3.访问的方式不同

static变量 类名.变量名

实例变量： 对象名.变量名

4.在内存中分配空间的时间不同

static变量 第一次使用类

实例变量 创建对象时候

static 内部类 只能访问外部类的静态成员

## final 和 abstract 关键字的作用

abstract不能修饰属性和构造方法；abstract 修饰的方法是抽象方法，需要子类被重写。

final不能修饰构造方法。final修饰的类不能被继承，方法不能被重写，使用 final 修饰的变 量的值不能被修改，所以就成了常量。

final 修饰引用类型变量，栈内存中的引用不能改变，但是所指向的堆内存中对象的属性值可以改变

final修饰类：表示该类不能被继承；

修饰方法：表示方法不能被重写但是允许重载；

修饰对象：对象的引用地址不能变，但是对象的初始化值可以变。

## final、finally、finalize 的区别

finally 在异常处理时提供 finally 块来执行任何清除操作。如果有 finally 的话，则不管是否发生异常，finally 语句都会被执行。一般情况下，都把关闭物理连接(IO流、数据库连接、Socket连接)等相关操作，放入到此代码块中

finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要清理工作。finalize() 方法是在垃圾收集器删除对象之前被调用的。它是在Object类中定义的，因此所有类都继承。

# 继承

## 父类成员在子类的访问权限

Public继承方式 不改变父类的访问权限

protected private不变 ,其余都变为protected

private 都改成private

## 子类成员在外部的访问权限

父类的private 只有父类能访问

private方式继承的非private成员 只有子类的成员函数能访问 ,子类的子类/外部不能访问

protected方式继承的非private成员 只有子类及子类的子类(非private继承) 能访问

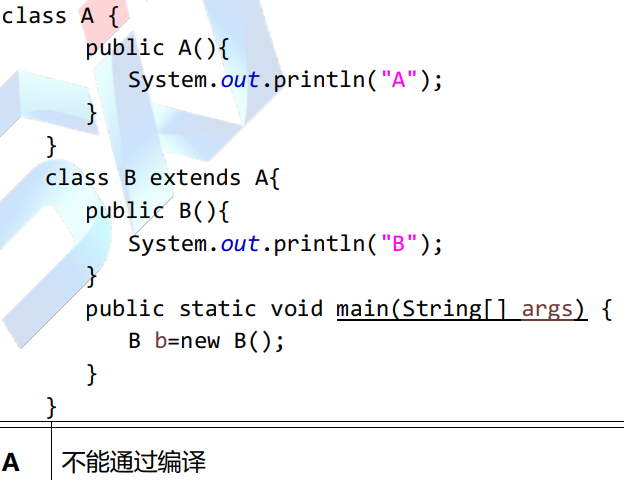
## 继承中构造方法的执行过程

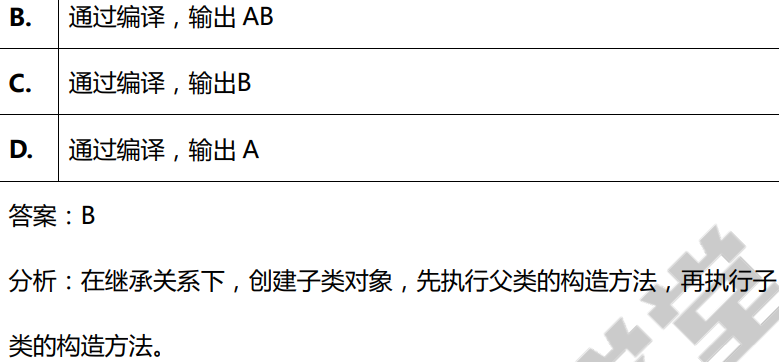
情况 1：没有super显式调用父类有参构造，没有this显式调用自身的构造，则系统会默认先调用父类无参,再调用子类构造方法。在这种情况下，写不写“super();”语句，效果是一样 的。

情况 2：通过super显式调用父类的有参构造方法，那将执行父类相应构造方法，而不执行父类无参构造方法。

情况 3：如果子类的构造方法中通过this显式调用自身的其他构造方法，在相应构造方法中应用以上两条规则。

特别注意的是，如果存在多级继承关系，在创建一个子类对象时，以上规则会多次向更高一级父类应用，一直到执行顶级父类 Object 类的无参构造方法为止。





## ==和 equals 的区别和联系

“==”是关系运算符，equals()是方法，同时他们的结果都返回布尔值

== a) 基本类型，比较的是值 b) 引用类型，比较地址 c) 不能比较没有父子关系的两个对象

equals() a) 系统类一般已经覆盖了 equals()，比较的是内容。 b) 用户自定义类如果没有覆盖 equals()，将调用父类的 equals（Object的equals比较地址）c) 用户自定义类需要覆盖父类的 equals()

Object 的==和 equals 比较的都是地址

## 面向对象的三大基本特征，五大基本原则

　1、封装

　　封装就是隐藏对象的属性和实现细节，仅对外公开接口，控制在程序中属性的读和修改的访问级别，将抽象得到的数据和行为（或功能）相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成“类”，其中数据和函数都是类的成员。

　　2、继承

　　继承是面向对象的基本特征之一，继承机制允许创建分等级层次的类。继承就是子类继承父类的特征和行为，使得子类对象（实例）具有父类的实例域和方法，或子类从父类继承方法，使得子类具有父类相同的行为。类似下面这个图：

3.多态三个条件

1、继承

2、子类重写父类的方法（向上转型）

Student person = new Student() 将一个父类的引用指向一个子类对象，自动进行类型转换。 此时通过父类引用变量调用的方法是子类覆盖或继承父类的方法

3、父类引用变量指向子类对象（向下转型）

优点：

1. 消除类型之间的耦合关系2. 可替换性3. 可扩充性4. 接口性5. 灵活性6. 简化性

## sql.Date和util.Date

1） java.sql.Date 是 java.util.Date 的子类，是一个包装了毫秒值的瘦包装器，允许JDBC将毫秒值标识为 SQL DATE 值

2）java.sql.Date是针对 SQL 语句使用的，只包含日期而没有时间部分。

以下操作中容易出现不易被发现的 BUG：获得一个 JAVA 里的日期对象。 从数据库里读取日期 试图比较两个日期对象是否相等。如果毫秒部分丢失，本来认为相等的两个日期对象用 Equals 方法可能返回 false。sql.Timestamp比util.Date类精确度要高

# Java 创建对象几种方式

1、new 语句

2、反射,调用 java.lang.Class 或者 java.lang.reflect.Constructor类的 newInstance()实例方法。

3、调用对象的clone()方法。

4、运用反序列化手段，调用 java.io.ObjectInputStream 对象的readObject()方法。

(1)和(2)都会显式地调用构造函数

(3)是在内存上已有对象的影印，不会调用构造函数

(4)是从文件中还原类的对象，也不会调用构造函数。

## 匿名内部类可不可以继承或实现接口？

匿名内部类是没有名字的内部类,不能继承其它类,但内部类可以作为接口,由另一个内部类实现.

1、由于匿名内部类没有名字，所以它没有构造函数,所以它必须完全借用父类的构造函数来实例化，换言之：匿名内部类完全把创建对象的任务交给了父类去完成。

2、在匿名内部类里创建新的方法没有太大意义，但它可以通过覆盖父类的方法达到神奇效果

3、匿名内部类没有名字，所以无法向下强转，持有对一个匿名内部类对象引用的变量类型一定是它的直接或间接父类类型。

## 基本类型不能做为HashMap的键值，而只能是引用类型

(1) Java使用泛型来约束HashMap<K, V>的键值对；而泛型必须是Object类型

map.put(1, “Java”)，实际上是将1进行了自动装箱操作,变为了 Integer类型

(2) 引用数据类型分为两类：系统提供的引用数据类型（如包装类、String

等）以及自定义引用数据类型。系统提供的引用数据类型中已经重写了

HashCode()和 equals()两个方法，所以能够保证key的唯一性；

但是自定义的引用数据类型需要重写HashCode()和equals()，保证key值的唯一性

# 接口和抽象类的区别

## 相同点

Ø 抽象类和接口均包含抽象方法，类必须实现所有的抽象方法，否则是抽象类

Ø 抽象类和接口都不能实例化，他们位于继承树的顶端，用来被其他类继承和实现

## 区别

接口中只能定义全局静态常量，不能定义变量。抽象类中可以定义常量和变量

接口中所有的方法都是抽象方法。抽象类中可以不全为抽象方法

接口中不能定义构造方法。抽象类中可以有构造方法，但不能用来实例化，而在子类实例化是执行，完成属于抽象类的初始化操作。

单继承多接口

接口可以继承接口

抽象类可以实现接口，抽象类可以继承实体类,可以有main方法

抽象类与普通类的唯一区别就是不能创建实例对象和允许有abstract方法。

最主要区别还是设计理念

Ø 接口 实现类仅仅是实现了接口定义的约定。接口定义了“做什么”，而实现类负责“怎么做”，体现了功能和实现分离的原则。接口和实现之间可以认为是一种“has-a 的关系”

Ø抽象类体现了一种继承关系，目的是复用代码，抽象类中定义了各个子类的相同代码，可以认为父类是一个实现了部分功能的“中间产品”，而子类是“最终产品”。父类和子类之间必须存在“is-a”的关系，即父类和子类在概念本质上应该是相同的。

## 静态内部类和内部类有什么区别

静态内部类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。

静态内部类可以有静态成员(方法，属性)，而非静态内部类则不能有静态成员(方法，属性)。

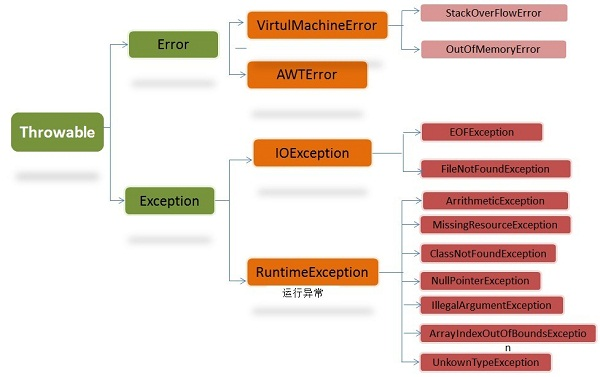
静态内部类只能访问外部类的静态成员。非静态内部类能够访问外部类的静态和非静态成员。

实例化方式不同：

1) 静态内部类：不依赖于外部类的实例，直接实例化内部类对象

2) 非静态内部类：通过外部类的对象实例生成内部类对象

## 编译时异常与运行时异常的区别



两个子类:异常,错误

异常和错误的区别：异常能被程序本身可以处理，错误是无法处理

运行时异常：都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

运行时异常的特点是Java编译器不会检查它，也就是说，当程序中可能出现这类异常，即使没有用try-catch/throws也会编译通过。

非运行时异常 （编译异常）：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

Error（错误）:是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

。这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错误，本质上也不应该试图去处理它所引起的异常状况。在 Java中，错误通过Error的子类描述。

Exception（异常）:是程序本身可以处理的异常。Exception 类有一个重要的子类 RuntimeException。RuntimeException 类及其子类表示“JVM 常用操作”引发的错误。例如，若试图使用空值对象引用、除数为零或数组越界，则分别引发运行时异常（NullPointerException、ArithmeticException）和 ArrayIndexOutOfBoundException。

注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。

通常，Java的异常(包括Exception和Error)分为可查的异常（checked exceptions）和不可查的异常（unchecked exceptions）。

可查异常（编译器要求必须处置的异常）：正确的程序在运行中，很容易出现的、情理可容的异常状况。可查异常虽然是异常状况，但在一定程度上它的发生是可以预计的，而且一旦发生这种异常状况，就必须采取某种方式进行处理。

除了RuntimeException及其子类以外，其他的Exception类及其子类都属于可查异常。这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过。

不可查异常(编译器不要求强制处置的异常):包括运行时异常（RuntimeException与其子类）和错误（Error）。

Exception 这种异常分两大类运行时异常和非运行时异常(编译异常)。程序中应当尽可能去处理这些异常。

## 下列说法错误的有（）(多选)

A. Java 面向对象语言容许单独的过栈与函数存在

B. Java 面向对象语言容许单独的方法存在

C. Java 语言中的方法属于类中的成员（member）

D. Java 语言中的方法必定隶属于某一类（对象），调用方法与过程或函

数相同

答案：ABC

B. Java 不允许单独的方法，过程或函数存在，需要隶属于某一类中；

C. 静态方法属于类的成员，非静态方法属于对象的成员。

### Math.round(11.5)和Math.round(-11.5)

round()是+0.5再向下取整

所以是12 -11

## final修饰变量，是引用还是引用的对象不能变？

final 修饰基本类型变量，其值不能改变

但是final修饰引用类型变量，栈内存中的引用不能改变，堆内存中的对象的属性值可以改变

例如class Test {

public static void main(String[] args) {

final Dog dog = new Dog("欧欧");

dog.name = "美美";//正确

dog = new Dog("亚亚");//错误

### static修饰变量,其值能改变吗?

对于静态变量在内存中只有一个拷贝（节省内存），JVM只为静态分配一次内存，在加载类的过程中完成静态变量的内存分配，可用类名直接访问（方便），当然也可以通过对象来访问（但是这是不推荐的）。

　对于实例变量，每创建一个实例，就会为实例变量分配一次内存，实例变量可以在内存中有多个拷贝，互不影响（灵活）。

所以一般在需要实现以下两个功能时使用静态变量：

1.在对象之间共享值时

2.方便访问变量时

static final修饰的成员变量可理解为“全局常量”，常量的值才是不能更改的，才是不可修改的！

静态变量并不是说其就不能改变值，不能改变值的量叫常量。 其拥有的值是可变的 ，而且它会保持最新的值。

说其静态，是因为它不会随着函数的调用和退出而发生变化。即上次调用的时候，如果我们给静态变量赋予某个值的话，下次函数调用时，这个值保持不变。

## try/catch中的finally

在异常处理时提供 finally 块来执行任何清除操作。不管是否发生异常，finally语句都会被执行，包括 遇到return。

finally 中语句不执行的唯一情况中执行了 System.exit(0)语句。

## static变量修饰符

### 1、生命周期不同。

成员变量随着对象的创建而存在，随着对象的被回收而释放。

静态变量随着类的加载而存在，随着类的消失而消失。

### 2、调用方式不同。

成员变量只能被对象调用。

静态变量可以被对象调用，还可以被类名调用。

对象调用：p.country

类名调用 ：Person.country

### 3、别名不同。

成员变量也称为实例变量。

静态变量称为类变量。

### 4、数据存储位置不同。

成员变量数据存储在堆内存的对象中，所以也叫对象的特有数据.

静态变量数据存储在方法区(共享数据区)的静态区，所以也叫对象的共享数据

### float f=3.4是否正确?

答:不正确。3.4 是双精度数，将双精度型（double）赋值给浮点型（float）

属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强

制类型转换 float f =(float)3.4; 或者写成 float f =3.4F;。

### short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗?

答：对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结

果也是 int 型，需要强制转换类型才能赋值给 short 型。而 short s1 = 1; s1

+= 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

### 两个值相同对象(x.equals(y) == true)， hash code可能不同?

答：不对，如果两个对象 x 和 y 满足 x.equals(y) == true，它们的哈希码应当相同。Java 对于 eqauls 方法和 hashCode 方法是这样规定的：

(1)如果两个对象相同（equals 方法返回 true），那么它们的hashCode 值一定要相同；

(2)如果两个对象的 hashCode 相同，它们并不一定相同

### 数据类型之间的转换:

字符串->基本数据类型

调用基本数据类型对应的包装类中的方法 parseXXX(String)或valueOf(String)即可返回相应基本类型；

基本数据类型->字符串

一种方法是将基本数据类型与空字符串””连接（+）即可获得其所对应的字符串；另一种方法是调用 String 类中的 valueOf(…)方法返回相应字符串

# 反射

## Class 类的作用？生成 Class 对象的方法有哪些？

Class 类是 Java 反射机制的起源和入口，用于获取与类相关的各种信息，提供了获取类信息的相关方法。Class 类继承自 Object 类Class 类是所有类的共同的图纸。每个类有自己的对象，好比图纸和实物的关系；每个类也可看做是一个对象，有共同的图纸 Class，存放类的 结构信息，能够通过相应方法取出相应信息：类的名字、属性、方法、构造方法、父类和接口

## 反射的使用场合和作用、及其优缺点

在编译时根本无法知道该对象或类可能属于哪些类，程序只依靠运行时信息来发现该对象和类的真实信息。通过反射可以使程序代码访问装载到 JVM 中的类的内部信息，获取已装载类的属性信息，获取已装载类的方法，获取已装载类的构造方法信息

反射提高了 Java 程序的灵活性和扩展性，降低耦合性，提高自适应能力。它允许程序创建和控制任何类的对象，无需提前硬编码目标类；

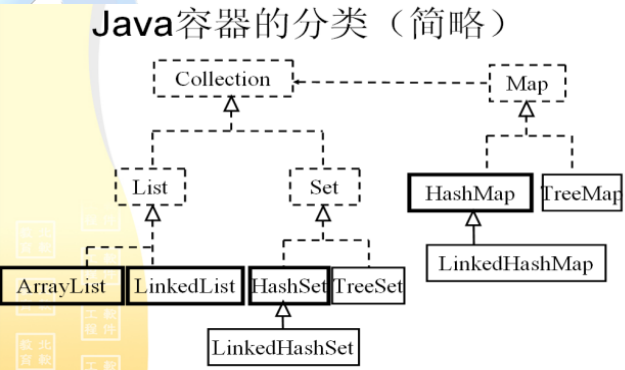
Java 反射技术应用领域很广，如软件测试等；许多流行的开源框架例如Struts、Hibernate、Spring 在实现过程中都采用了该技术

反射基本上是一种解释操作，用于字段和方法接入时要远慢于直接代码。因此 Java 反射机制主要应用在对灵活性和扩展性要求很高的系统框架上,普通程序不建议使用。使用反射会模糊程序内部逻辑：程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术，因而会带来维护问题。反射代码比相应的直接代码更复杂。

# 集合

## Collection（List、Set、Collection、Map的区别联系）

Java 中集合主要分为两种：Collection 和 Map



1、Collection不唯一，无序

2、List不唯一，有序

3、Set唯一，无序

4、Map Key无序，唯一 value不要求有序，允许重复

## ArrayList 和 LinkedList 的区别和联系

相同点：

都实现List接口，有序、不唯一

不同点：

ArrayList数组实现,长度可变，查询效率高 虽然是数组,初始容量10,但是当满了时,就会新建一个2倍容量的数组,并把原数组复制过去,实现了扩容

LinkedList 采用双向链表存储方式。插入、删除元素时效率高

#### List list = new ArrayList() 与 ArrayList alist = new ArrayList()

List接口有多个实现类，现在你用的是ArrayList，也许哪一天需要换成LinkedList或者Vector等等，这时你只要改变一行就行 这就是面向接口编程

LinkedList和ArrayList都实现了add等方法,都实现了List接口

在List list时,并不知道实例化了Linked还是Array,但是这个list都是要去add()的

这也是多态的体现,父类引用指向子类对象

## HashSet 的使用和原理（hashCode()和 equals()）

1）哈希表的查询速度特别快，时间复杂度为 O（1）。

2）HashMap、Hashtable、HashSet 这些集合采用的是哈希表结构，需要用hashCode

3）系统类已经覆盖了hashCode 方法 自定义类放入hash类集合，必须重写 hashcode。如果不重写，调用的是 Object 的 hashcode，而Object 的 hashCode 实际上是地址。

4）向哈希表中添加数据的原理：当向集合Set中增加对象时，首先集合计算要增加对象的 hashCode 码，得到一个位置用来存放当前对象，如在该位置没有一个对象存在的话，那么集合 Set 认为该对象在集合中不存在，直接增加进去。如果在该位置有一个对象存在的话，接着将准备增加到集合中的对象与该位置上的对象进行equals方法比较，如果该 equals 方法返回 false,那么集合认为集合中不存在该对象，在进行一次散列，将该对象放到散列后计算出的新地址里。如果equals方法返回 true，那么集合认为集合中已经存在该对象了，不会再将该对象增加到集合中了。

5）在哈希表中判断两个元素是否重复要使用到hashCode()和equals()。hashCode决定数据在表中的存储位置，而equals判断是否存在相同数据。

## TreeSet 的原理和使用（Comparable和comparator）

1）TreeSet 集合，元素不允许重复且有序(自然顺序)

2）TreeSet 采用树结构存储数据，存入元素时需要和树中元素进行对比，需

要指定比较策略。

3）可以通过 Comparable(外部比较器)和 Comparator(内部比较器)来指定

比较策略，实现了 Comparable 的系统类可以顺利存入 TreeSet。自定

义类可以实现 Comparable 接口来指定比较策略。

4）可创建 Comparator 接口实现类来指定比较策略，并通过 TreeSet 构造

方法参数传入。这种方式尤其对系统类非常适用。

## 集合和数组的比较（为什么引入集合）

数组不是面向对象的，存在明显的缺陷，集合完全弥补了数组的一些缺点，比数组更灵活实用，可大大提高软件的开发效率.不同的集合框架类可适用于不同场合

1）数组的效率高于集合类.

2）数组能存放基本数据类型和对象，而集合类中只能放对象

3）数组容量固定且无法动态改变，集合类容量动态改变

4）数组无法判断实际有多少元素，length只告诉了array的容量

5）集合有多种实现方式和不同的适用场合，而不像数组仅采用顺序表方式。

6）集合以类的形式存在，具有封装、继承、多态等类的特性，通过简单的方法和属性调用即可实现各种复杂操作，提高效率

## Collection 和 Collections 的区别

1）Collection集合接口，存储一组不唯一，无序的对象。两个子接口List和Set

2）Collections类专门用来操作集合类 ，它提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

Map 的实现类中，哪些是有序的，哪些是无序的，有序的是

如何保证其有序性，你觉得哪个有序性性能更高，你有没有更好

或者更高效的实现方式？

答：

1. Map 的实现类有 HashMap,LinkedHashMap,TreeMap

2. HashMap 是有无序的，LinkedHashMap 和 TreeMap 都是有序的（LinkedHashMap记录了添加数据的顺序；TreeMap默认是自然升序）

3. LinkedHashMap 底层存储结构是哈希表+链表，链表记录了添加数据的顺序

4. TreeMap 底层存储结构是二叉树，二叉树的中序遍历保证了数据的有序性

5. LinkedHashMap 性能比较高，因为底层数据存储结构采用的哈希表

## TreeMap 和 TreeSet 在排序时如何比较元素？Collections工具类中的 sort（）方法如何比较元素？

TreeSet 要求存放的对象所属的类必须实现 Comparable 接口，该接口提供了比较元素的 compareTo()方法，当插入元素时会 回调该方法比较元素的大小。

TreeMap 要求存放的键值对映射的键必须实现 Comparable接口从而根据键对元素进行排序。Collections 工具类的 sort 方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现

Comparable 接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是 Comparator 接口的子类型（需要重写 compare 方法实现元素的比较），相当于一个临时定义的排序规则，其实就是是通过接口注入比较元素大小的算法，也是对回调模式的应用。

## List、Map、Set 三个接口，存取元素时，各有什么特点？

List 以特定索引来存取元素，可有重复元素。

Set 不能存放重复元素（用对象的 equals()方法来区分元素是否重复）

Map 保存键值对映射，映射关系可以是一对一或多对一。

Set 和 Map 容器都有基于哈希存储和排序树（红黑树）的两种实现版本，基于哈希存储的版本理论存取时间复杂度为O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

# Spring事务传播机制

## 事务的特性

* 原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。
* 一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。
* 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。
* 持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

## Spring事务的配置方式

Spring支持编程式事务管理以及声明式事务管理两种方式。

### 1. 编程式事务管理

编程式事务管理是侵入性事务管理，使用TransactionTemplate或者直接使用PlatformTransactionManager，对于编程式事务管理，Spring推荐使用TransactionTemplate。

### 2. 声明式事务管理

声明式事务管理建立在AOP之上，其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，执行完目标方法之后根据执行的情况提交或者回滚。  
编程式事务每次实现都要单独实现，但业务量大功能复杂时，使用编程式事务无疑是痛苦的，而声明式事务不同，声明式事务属于无侵入式，不会影响业务逻辑的实现，只需要在配置文件中做相关的事务规则声明或者通过注解的方式，便可以将事务规则应用到业务逻辑中。  
显然声明式优于编程式事务管理，这正是Spring倡导的非侵入式的编程方式。唯一不足的地方就是声明式事务管理的粒度是方法级别，而编程式事务管理是可以到代码块的，但是可以通过提取方法的方式完成声明式事务管理的配置。

## 事务的传播机制

事务的传播性一般用在事务嵌套的场景，事务方法里调用了另外一个事务方法，需要事务传播机制的配置确定两个方法是各自作为独立的方法提交还是内层的事务合并到外层的事务一起提交

* PROPAGATION\_REQUIRED默认

外层有事务，则加入外层事务。没有则新建一个事务执行

* PROPAGATION\_REQUES\_NEW

外层事务挂起，开启新事务，当前事务执行完毕，恢复上层事务的执行。如果外层没有事务，执行当前新开的事务

* PROPAGATION\_SUPPORT  
  外层有事务，则加入外层事务，外层没有事务，以非事务方式执行。完全依赖外层的事务
* PROPAGATION\_NOT\_SUPPORT

不支持事务，如果外层有事务则挂起，执行完当前代码，则恢复外层事务，无论是否异常都不会回滚当前的代码

* PROPAGATION\_NEVER

不支持外层事务，即如果外层有事务就抛出异常

* PROPAGATION\_MANDATORY  
  与NEVER相反，如果外层没有事务则抛出异常
* PROPAGATION\_NESTED可以保存状态保存点，当前事务回滚到某一个点，从而避免所有的嵌套事务都回滚，如果子事务没有把异常吃掉，基本还是会引起全部回滚的。

## 只读

如果一个事务只对数据库执行读操作，那么该数据库就可能利用那个事务的只读特性，采取某些优化措施。通过把一个事务声明为只读，可以给后端数据库一个机会来应用那些它认为合适的优化措施。由于只读的优化措施是在一个事务启动时由后端数据库实施的， 因此，只有对于那些具有可能启动一个新事务的传播行为（REQUIRES\_NEW、EQUIRED、NESTED）的方法来说，将事务声明为只读才有意义。

## 事务超时

事务可能涉及对数据库的锁定，长时间运行事务会不必要地占用数据库资源。这时就可以声明一个事务在特定秒数后自动回滚，不必等它自己结束。

由于超时时钟在一个事务启动的时候开始的，因此，只有对于那些具有可能启动一个新事务的传播行为（REQUIRES\_NEW、REQUIRED、NESTED）的方法来说，声明事务超时才有意义。

## 回滚规则

在默认设置下，事务只在出现运行时异常（runtime exception）时回滚，而在出现受检查异常（checked exception）时不回滚。

不过，可以声明在出现特定受检查异常时像运行时异常一样回滚。同样，也可以声明一个事务在出现特定的异常时不回滚，即使特定的异常是运行时异常。

# Spring声明式事务配置参考

1. 事务的传播性：  
   @Transactional(propagation=Propagation.REQUIRED)
2. 事务的隔离级别：  
   @Transactional(isolation = Isolation.READ\_UNCOMMITTED)

读取未提交数据(会出现脏读, 不可重复读) 基本不使用

1. 只读：  
   @Transactional(readOnly=true)  
   该属性用于设置当前事务是否为只读事务，设置为true表示只读，false则表示可读写，默认值为false。
2. 事务的超时性：  
   @Transactional(timeout=30)
3. 回滚：  
   指定单一异常类：@Transactional(rollbackFor=RuntimeException.class)  
   指定多个异常类：@Transactional(rollbackFor={RuntimeException.class, Exception.class})

# 数据结构

## 堆

堆就是用数组实现的二叉树，所有它没有使用父指针或者子指针。堆根据“堆属性”来排序，“堆属性”决定了树中节点的位置。

堆分为两种：最大堆和最小堆，两者的差别在于节点的排序方式。

在最大堆中，父节点的值比每一个子节点的值都要大。在最小堆中，父节点的值比每一个子节点的值都要小。这就是所谓的“堆属性”，并且这个属性对堆中的每一个节点都成立。

### 堆和普通树的区别

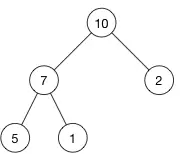
节点顺序。在二叉搜索树中，左子节点必须比父节点小，右子节点必须必比父节点大。但是在堆中并非如此。在最大堆中两个子节点都必须比父节点小，而在最小堆中，它们都必须比父节点大。

内存占用。普通树占用的内存空间比它们存储的数据要多。你必须为节点对象以及左/右子节点指针分配额为是我内存。堆仅仅使用一个数据来存储数组，且不使用指针。

平衡。二叉搜索树必须是“平衡”的情况下，其大部分操作的复杂度才能达到O(log n)。你可以按任意顺序位置插入/删除数据，或者使用 AVL 树或者红黑树，但是在堆中实际上不需要整棵树都是有序的。我们只需要满足对属性即可，所以在堆中平衡不是问题。因为堆中数据的组织方式可以保证O(log n) 的性能。

搜索。在二叉树中搜索会很快，但是在堆中搜索会很慢。在堆中搜索不是第一优先级，因为使用堆的目的是将最大（或者最小）的节点放在最前面，从而快速的进行相关插入、删除操作。

### 堆的存储方式

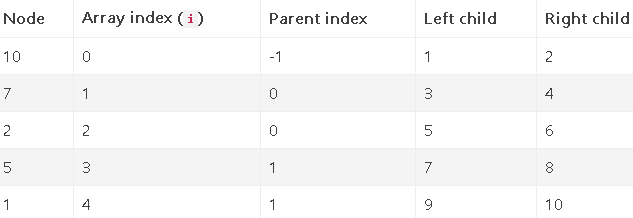
[ 10, 7, 2, 5, 1 ]

如果 i 是节点的索引，那么下面的公式就给出了它的父节点和子节点在数组中的位置

parent(i) = floor((i - 1)/2)

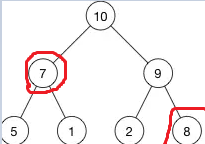
left(i) = 2i + 1

right(i) = 2i + 2



根节点(10)没有父节点，因为 -1 不是一个有效的数组索引。同样，节点 (2)，(5)和(1) 没有子节点，因为这些索引已经超过了数组的大小，所以我们在使用这些索引值的时候需要保证是有效的索引值。

堆并不一定是有序数组！要将堆转换成有序数组，需要使用堆排序。

堆中低层节点不一定小于高层节点！7<8

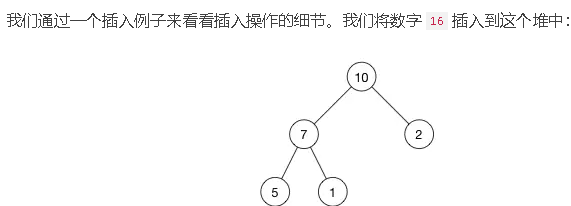
### 两个保证插入或删除节点后依然是最大/小堆的方法

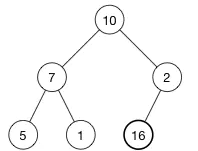
shiftUp(): 如果一个节点比它的父节点大（最大堆）或者小（最小堆），那么需要将它同父节点交换位置。这样是这个节点在数组的位置上升。

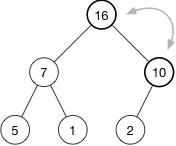
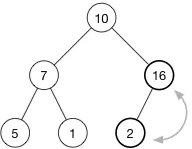
shiftDown(): 如果一个节点比它的子节点小（最大堆）或者大（最小堆），那么需要将它向下移动。这个操作也称作“堆化（heapify）”。

shiftUp 或者 shiftDown 是一个递归的过程，所以它的时间复杂度是 O(log n)

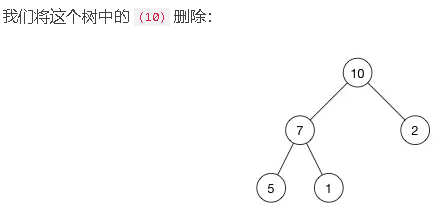
### 插入

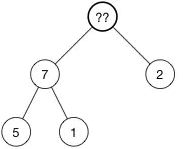


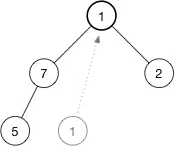
变成了[ 10, 7, 2, 5, 1, 16 ] 

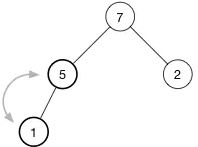
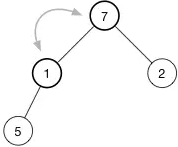
为了恢复堆属性,交换两次

### 删除根节点



此时顶部为空节点

取出最后一个节点,代替空节点

之后开始交换

### 删除子节点

remove() 的通用版本，它可能会使用到 shiftDown 和 shiftUp。

[ 10, 7, 2, 5, 1 ] 删除7

-> [ 10, 1, 2, 5, 7 ] 将最后的1与7交换,7就是需要返回的元素,用removeLast()删除,而1的位置不对,用shiftDown()调整位置

## 时间复杂度

二分查找 对数级别 logN

循环 线性级别 N

归并(分治思想) 线性对数级别 NlogN

穷举 指数级别 2^N