## 基础篇

**四大编程思想**

1.POP-面向过程编程（Process Oriented Programming）

2.OOP-面向对象编程（Object Oriented Programming）

3.SOA-面向服务架构（service-oriented architecture）

4.AOP-面向方面(切面)编程（Aspect Oriented Programming）

详细：

1:面向过程编程是以功能为中心来进行思考和组织的一种编程方法，它强调的是系统的数据被加工和处理的过程，在程序设计中主要以函数或者过程为程序的基本组织方式，系统功能是由一组相关的过程和函数序列构成。面向过程强调的是功能（加工），数据仅仅是作为输入和输出存在。

2:面向对象编程的组织方式更加贴近现实世界。面向对象以对象为中心，将对象的内部组织与外部环境区分开来，将表征对象的内部属性数据与外部隔离开来，其行为与属性构成一个整体，而系统功能则表现为一系列对象之间的相互作用的序列，能更加形象的模拟或表达现实世界。在编程组织中，对象的属性与方法不再像面向过程那样分开存放，而是视为一个整体（程序的最终实现其实还是分离的，但这仅仅是物理实现上的，不影响将对象的这两个部分视为一个整体），因此具有更好的封装性和安全性（表征内部的属性数据需要通过对象的提供的方法来访问）。面向对象强调的是整体性，因此面向对象与面向过程在很多方面是可以互补的。同时由于对象继承和多态技术的引入，使得面向对象具有更强、更简洁的对现实世界的表达能力。从而增强了编程的组织性，重用性和灵活性。 面向对象依然保留着面向过程的特性，面向过程中的功能变成了对象的方法，加工处理功能变成了对象的服务性方法，而这部分方法依然需要外界的输入，同时也对外界进行输出，只是输入和输出也变成了对象。从方法论来讲，我们可以将面向过程与面向对象看做是事物的两个方面--局部与整体（注意：局部与整体是相对的），在实际应用中，两者方法都同样重要。面向过程和面向对象是编程方法中最基本的两种方法，处于编程方法体系的底层。

3:面向服务与面向过程、面向对象本质上没有什么不同，区别就在于考虑问题的层面不同。面向对象和面向过程多用于系统内部的组织和管理，而面向服务主要用于系统间的组织和管理。面向服务是更大的对象或者过程。面向服务设计的三大原则是无状态、单一实例和明确的服务接口。明确的服务接口是强制和必须的，但无状态和单一实例则不属于强制性原则，虽然说服务提供状态管理会增加服务的复杂性，多实例也一样会增加服务的复杂性（需要增加同步并发处理等，而且会导致访问不确定性），但很多情况下这又是无法避免的。现在的面向服务架构，主要用于系统间的交互和集成，有一系列的标准(XML,SOAP,WSDL,XSD,WS-policy,WS-BPEL等)。

4:面向方面应该属于面向对象的范畴，从对象组织角度来讲，我们一般采用的分类方法都是使用类似生物学分类的方法，以“继承”关系为主线，我们称之为纵向。但事实上，对象之间除了这种纵向分类之外，我们同样可以从横向的角度去观察这些对象，这就是面向方面（切面）编程的基本出发点。原来要解决这类问题，我们一般是采用接口来完成，但这有两个问题，一是对象设计的时候一般都是纵向思维，如果这个时候需要就需要考虑这些不同类的对象的这些共性，不仅会增加设计的难度和复杂性，还会造成类的接口过多而难以维护，二是需要对现有的对象动态增加这种行为或者责任的时候非常困难。现在很多程序的都是以中间语言存在，执行的时候是解释执行或者即时编译执行，这也为增加这种切面行为或者责任提供了比较好的切入口。面向方面跟Api hook很类似。

**Java基本数据类型**

JAVA中的数值类型不存在无符号的，它们的取值范围是固定的，不会随着机器硬件环境或者操作系统的改变而改变。

boolean（布尔类型）、byte（字节类型）、char（字符类型）、short（短整型）、int（整型）、long（长整型）、double（双精度数）、float（单精度数）八种。

类型转换：char-->    自动转换：byte-->short-->int-->long-->float-->double

强制转换： ①会损失精度，产生误差，小数点以后的数字全部舍弃。

②容易超过取值范围。

boolean：只有true和false两个取值。

byte：8位，最大存储数据量是255，存放的数据范围是-128~127之间。

short：16位，最大数据存储量是65536，数据范围是-32768~32767之间。

int：32位，最大数据存储容量是2的32次方减1，数据范围是负的2的31次方到正的2的31次方减1。

long：64位，最大数据存储容量是2的64次方减1，数据范围为负的2的63次方到正的2的63次方减1。

float：32位，数据范围在3.4e-45~1.4e38，直接赋值时必须在数字后加上f或F。

double：64位，数据范围在4.9e-324~1.8e308，赋值时可以加d或D也可以不加。

记忆：

char：16位，字符占两个字节，存储Unicode码，用单引号赋值。

8位：Byte（字节型）

16位：short（短整型）、char（字符型）

32位：int（整型）、float（单精度型/浮点型）

64位：long（长整型）、double（双精度型）

最后一个：boolean(布尔类型)

1个字节（Byte）等于8位（bit）可以储存1个英文字母或者半个汉字。

字："字"由若干个字节构成，字的位数叫做字长，不同档次的机器有不同的字长。例如一台8位机，它的1个字就等于1个字节，字长为8位。如果是一台16位机，那么，它的1个字就由2个字节构成，字长为16位。字是计算机进行数据处理和运算的单位。

使用包装类

Character: **public final class Character implements java.io.Serializable, Comparable<Character>**

public static final int *MIN\_RADIX* = 2;

public static final int *MAX\_RADIX* = 36;

public static final char *MIN\_VALUE* = '\u0000';

public static final char *MAX\_VALUE* = '\uFFFF';

Boolean : **public final class Boolean implements java.io.Serializable, Comparable<Boolean>**

public static final Boolean *TRUE* = new Boolean(true)

public static final Boolean *FALSE* = new Boolean(false);

Byte: **public final class Byte extends Number implements Comparable<Byte>**

public static final byte *MIN\_VALUE* = -128;

public static final byte *MAX\_VALUE* = 127;

Short: **public final class Short extends Number implements Comparable<Short>**

public static final short *MIN\_VALUE* = -32768;

public static final short *MAX\_VALUE* = 32767;

Integer:  **public final class Integer extends Number implements Comparable<Integer>**

@Native public static final int *MIN\_VALUE* = 0x80000000;

@Native public static final int *MAX\_VALUE* = 0x7fffffff;

Long: **public final class Long extends Number implements Comparable<Long>**

@Native public static final long *MIN\_VALUE* = 0x8000000000000000L;

@Native public static final long *MAX\_VALUE* = 0x7fffffffffffffffL;

Double: **public final class Double extends Number implements Comparable<Double>**

public static final double *MAX\_VALUE* = 0x1.fffffffffffffP+1023;

public static final double *MIN\_VALUE* = 0x0.0000000000001P-1022; // 4.9e-324

Float: **public final class Float extends Number implements Comparable<Float>**

public static final float *MAX\_VALUE* = 0x1.fffffeP+127f; // 3.4028235e+38f

public static final float *MIN\_VALUE* = 0x0.000002P-126f; // 1.4e-45f

**（1）**

short s1 = 1; s1 = s1 + 1;会报错 等价于short s1=(int) s1+1

short s1 = 1; s1 +=1;没错 等价于s1=(Short)s1+1

**（2）**

byte a = 127; a+=5; System.out.println(a); 结果为-124

**访问修饰符**

public：当前类、同包、子类、其他包

protected：当前类、同包、子类

default：当前类、同包 不写访问修饰符时默认的

friendly：当前类、同包

private：当前类

**&与&&的区别**

两者都是要求运算符的两端的布尔值都是true，整个表达式的值才是true，但是&&是短路运算，如果左边表达式的值为false，则右边表达式会被直接短路掉

**抽象类和接口有什么异同**

相同点：

抽象类和接口都不能实例化；

一个类如果继承了某个抽象类或者实现了某个接口，都需要对其中的抽象方法全部实现，否则该类仍然需要被声明为抽象类。

不同点：

抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器，并且方法必须是抽象的。

抽象类中的成员不限制访问修饰符，而接口中的成员必须全部是public。

抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量

**接口是否可以继承接口？抽象类是否可以实现接口？抽象类是否可以继承具体类**

1）接口可以继承接口，并且支持多重继承

2）抽象类可以实现接口，抽象类除了不能实例化对象以外，其他和普通类一样

3）抽象类可以继承具体类，也可以继承抽象类

**try{}里有一个return语句，那么紧跟在这个try后的finally{}里的代码会不会被执行，什么时候被执行，在return前还是后?**

**会执行，在方法返回调用之前执行**

**Java中如何实现序列化，有什么意义？**

**实现序列化需要让一个类实现Serializable接口（该接口可以标注该类对象是可被序列化的），然后用一个输出流建立对象输出流并通过writeObject方法就可以写出对象；反序列化可以用一个输入流建立对象输入流，并通过readObject方法从流中读取对象。**

**意义：序列化为了解决对象流读写操作时可能引发的数据乱序的问题，能够实现对象的持久化，还能够用于对象的深度克隆**

**Statement和PreparedStatement有什么区别？哪个性能更好？**

1）Statement会引起注入，PreparedStatement防止注入，增强SQL的安全性

2）PreparedStatement中的SQL语句是可以带参数的

3）PreparedStatement是预编译语句，批量处理SQL语句时有明显性能上的优势

---------------------

**Java集合**

Java集合类存放于 java.util 包中，是一个用来存放对象的容器。

注意：①、集合只能存放对象。比如你存一个 int 型数据 1放入集合中，其实它是自动转换成 Integer 类后存入的，Java中每一种基本类型都有对应的引用类型。

　　　②、集合存放的是多个对象的引用，对象本身还是放在堆内存中。

　　　③、集合可以存放不同类型，不限数量的数据类型。

**Iterator:迭代器，它是Java集合的顶层接口（不包括 map 系列的集合，Map接口 是 map 系列集合的顶层接口）**

public interface Iterator<E>

public interface Iterable<T>//存在于java.lang包中;

Iterator ：存在于 java.util 包中。核心的方法next(),hasnext(),remove()。

**Collection:List 接口和 Set 接口的父接口**

public interface Collection<E> extends Iterable<E>

public interface List<E> extends Collection<E>

**List :有序，可以重复的集合。**

List 接口的三个典型实现：

　　①、List list1 = new ArrayList();

　　　　底层数据结构是数组，查询快，增删慢;线程不安全，效率高

　 ②、List list2 = new Vector();

　　　　底层数据结构是数组，查询快，增删慢;线程安全，效率低,几乎已经淘汰了这个集合

　 ③、List list3 = new LinkedList();

　　　　底层数据结构是链表，查询慢，增删快;线程不安全，效率高

**Set：典型实现 HashSet()是一个无序，不可重复的集合**

**1.Set hashSet = new HashSet();**

①、HashSet:不能保证元素的顺序；不可重复；不是线程安全的；集合元素可以为 NULL;

②、其底层其实是一个数组，存在的意义是加快查询速度。我们知道在一般的数组中，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，此时的查询效率依赖于查找过程中比较的次数。而 HashSet 集合底层数组的索引和值有一个确定的关系：index=hash(value),那么只需要调用这个公式，就能快速的找到元素或者索引。

③、对于 HashSet: 如果两个对象通过 equals() 方法返回 true，这两个对象的 hashCode 值也应该相同。

　 　　1、当向HashSet集合中存入一个元素时，HashSet会先调用该对象的hashCode（）方法来得到该对象的hashCode值，然后根据hashCode值决定该对象在HashSet中的存储位置

　　　　　　1.1、如果 hashCode 值不同，直接把该元素存储到 hashCode() 指定的位置

　　　　　　1.2、如果 hashCode 值相同，那么会继续判断该元素和集合对象的 equals() 作比较

　　　　　　　　　　1.2.1、hashCode 相同，equals 为 true，则视为同一个对象，不保存在 hashSet（）中

　　　　　　　　　　1.2.2、hashCode 相同，equals 为 false，则存储在之前对象同槽位的链表上，这非常麻烦，我们应该约束这种情况，即保证：如果两个对象通过 equals() 方法返回 true，这两个对象的 hashCode 值也应该相同。

注意：每一个存储到 哈希 表中的对象，都得提供 hashCode() 和 equals() 方法的实现，用来判断是否是同一个对象

　　　对于 HashSet 集合，我们要保证如果两个对象通过 equals() 方法返回 true，这两个对象的 hashCode 值也应该相同。

**2、Set linkedHashSet = new LinkedHashSet();**

　　①、不可以重复，有序

　　因为底层采用 链表 和 哈希表的算法。链表保证元素的添加顺序，哈希表保证元素的唯一性

**3、Set treeSet = new TreeSet();**

　　TreeSet:有序；不可重复，底层使用 红黑树算法，擅长于范围查询。

　　\* 如果使用 TreeSet() 无参数的构造器创建一个 TreeSet 对象, 则要求放入其中的元素的类必须实现 Comparable 接口所以, 在其中不能放入 null 元素

\* 必须放入同样类的对象.(默认会进行排序) 否则可能会发生类型转换异常.我们可以使用泛型来进行限制。

**以上三个 Set 接口的实现类比较：**

　　共同点：1、都不允许元素重复

　　　　　　2、都不是线程安全的类，解决办法：Set set = Collections.synchronizedSet(set 对象)

　　不同点：

　　　　HashSet:不保证元素的添加顺序，底层采用 哈希表算法，查询效率高。判断两个元素是否相等，equals() 方法返回 true,hashCode() 值相等。即要求存入 HashSet 中的元素要覆盖 equals() 方法和 hashCode()方法

　　　　LinkedHashSet:HashSet 的子类，底层采用了 哈希表算法以及 链表算法，既保证了元素的添加顺序，也保证了查询效率。但是整体性能要低于 HashSet

　　　　TreeSet:不保证元素的添加顺序，但是会对集合中的元素进行排序。底层采用 红-黑 树算法（树结构比较适合范围查询）

**Map：key-value 的键值对，key 不允许重复，value 可以**

1、严格来说 Map 并不是一个集合，而是两个集合之间 的映射关系。

2、这两个集合没每一条数据通过映射关系，我们可以看成是一条数据。即 Entry(key,value）。Map 可以看成是由多个 Entry 组成。

3、因为 Map 集合即没有实现于 Collection 接口，也没有实现 Iterable 接口，所以不能对 Map 集合进行 for-each 遍历。

HashMap：基于哈希表的Map接口实现（哈希表对键进行散列，Map结构即映射表存放键值对）

LinkedHashMap:HashMap 的基础上加上了链表数据结构

HashTable:哈希表

TreeMap:红黑树（自平衡的排序二叉树）

hashCode（）与equals（）的相关规定：

HashSet如何检查重复

当你把对象加入HashSet时，HashSet会先计算对象的hashcode值来判断对象加入的位置，同时也会与其他加入的对象的hashcode值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同hashcode值的对象，这时会调用equals（）方法来检查hashcode相等的对象是否真的相同。如果两者相同，HashSet就不会让加入操作成功。

如果两个对象相等，则hashcode一定也是相同的

两个对象相等,对两个equals方法返回true

两个对象有相同的hashcode值，它们也不一定是相等的

综上，equals方法被覆盖过，则hashCode方法也必须被覆盖

hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）。

==与equals的区别

==是判断两个变量或实例是不是指向同一个内存空间 equals是判断两个变量或实例所指向的内存空间的值是不是相同

==是指对内存地址进行比较 equals()是对字符串的内容进行比较3.==指引用是否相同 equals()指的是值是否相同

comparable 和 comparator的区别？

comparable接口实际上是出自java.lang包 它有一个 compareTo(Object obj)方法用来排序

comparator接口实际上是出自 java.util 包它有一个compare(Object obj1, Object obj2)方法用来排序

一般我们需要对一个集合使用自定义排序时，我们就要重写compareTo方法或compare方法，当我们需要对某一个集合实现两种排序方式，比如一个song对象中的歌名和歌手名分别采用一种排序方法的话，我们可以重写compareTo方法和使用自制的Comparator方法或者以两个Comparator来实现歌名排序和歌星名排序，第二种代表我们只能使用两个参数版的Collections.sort().

对objects数组进行排序，我们可以用Arrays.sort()方法

对objects的集合进行排序，需要使用Collections.sort()方法

集合总结：

List 元素是有序的、可重复

ArrayList：线程不安全，但速度快，底层数据结构是数组结构。

初始默认大小为10 每次扩容为之前的1.5倍，

扩容增量：原容量的 0.5倍+1

　　　　　如 ArrayList的容量为10，一次扩容后是容量为16

Vector：线程安全，但速度慢，底层数据结构是数组结构。

加载因子为1：即当元素个数超过容量长度时，进行扩容，

扩容增量：原容量的 2倍，

如 Vector的容量为10，一次扩容后是容量为20。

Set(集) 元素无序的、不可重复。

HashSet：线程不安全，存取速度快

　　　　　底层实现是一个HashMap（保存数据），实现Set接口

　　　　　默认初始容量为16（为何是16，见下方对HashMap的描述）

　　　　　加载因子为0.75：即当 元素个数 超过 容量长度的0.75倍 时，进行扩容

　　　　　扩容增量：原容量的 2倍

　　　　　如 HashSet的容量为16，一次扩容后是容量为32

Map是一个双列集合

如果HashMap的键(key)是自定义的对象，那么需要按规则定义它的equals()和hashCode()方法。

HashMap：默认初始容量为16（为何是16—>16是2^4，可以提高查询效率，另外，32=16<<1-->index = HashCode（Key） & （Length - 1）计算当前key值放入哪个位置。相对平均）

　　　　　加载因子为0.75：即当 元素个数 超过 容量长度的0.75倍 时，进行扩容

　　　　　扩容增量：原容量的 2倍

　　　　　如 HashSet的容量为16，一次扩容后是容量为32

HashTable：默认初始容量为1.

加载因子为0.75：即当 元素个数 超过 容量长度的0.75倍 时，进行扩容

计算扩容边界值的方式是：

threshold = (int) Math.min(initialCapacity \* loadFactor, MAXIMUM\_CAPACITY + 1);取最小值

ConcurrentHashMap: 线程安全默认16个Segment ，每个Segmenty有2个数组。

JDK6,7中的ConcurrentHashmap主要使用Segment来实现减小锁粒度，把HashMap分割成若干个Segment，在put的时候需要锁住（Lock）Segment。

get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

jdk8 中完全重写了concurrentHashmap，

1.不采用segment而采用node，锁住node来实现减小锁粒度。

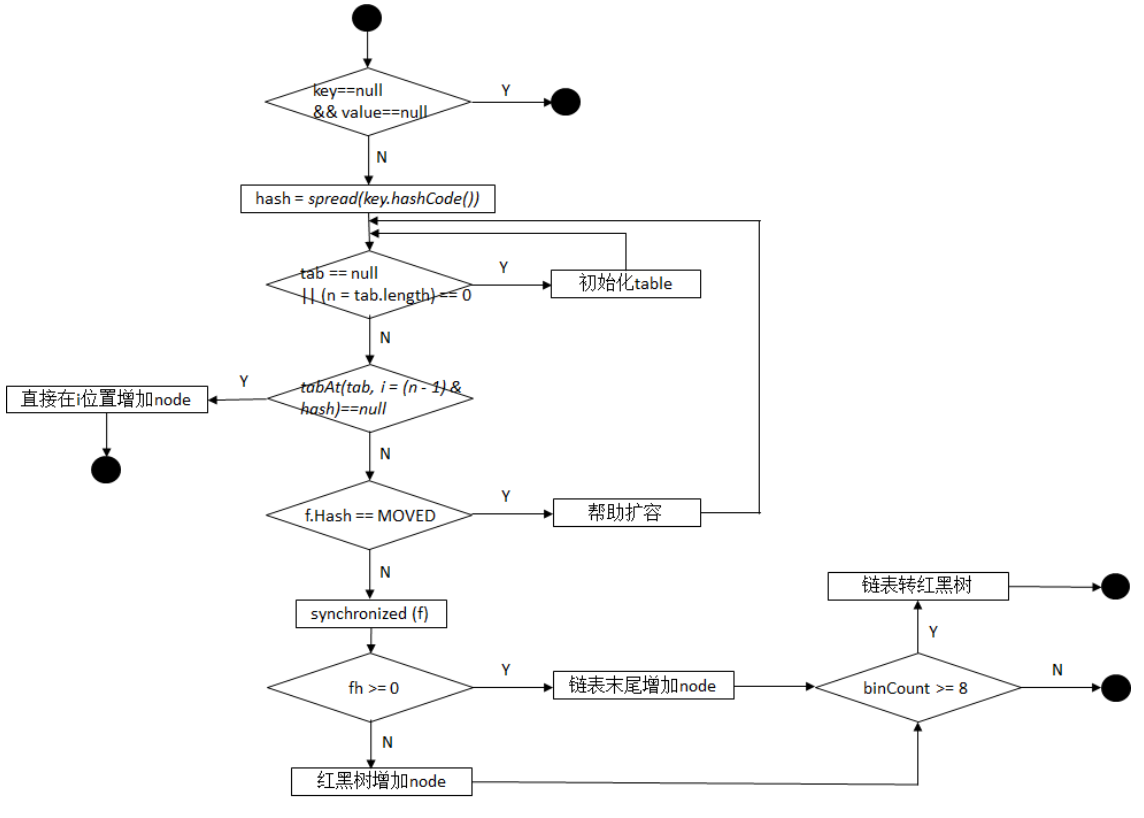
2.设计了MOVED状态 当resize的中过程中 线程2还在put数据，线程2会帮助resize。

3.使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。

4.sizeCtl的不同值来代表不同含义，起到了控制的作用。

5.JDK8中使用synchronized而不是ReentrantLock.

put方法的主要流程如下:



ConcurrentHashMap的实现使用的是锁分离思想，只是锁住的是一个node，而锁住Node之前的操作是基于在volatile和CAS之上无锁并且线程安全的

**类加载机制**

类加载器的作用就是把class文件加载进JVM虚拟机中，但是JVM启动是并不会把所有的class文件一次性都加载，而是根据需要去动态加载。

class文件是字节码文件，是jvm虚拟机可以识别的文件。

java自带三个类加载器：

1.Bootstrp loader

Bootstrp加载器是用C++语言写的，它是在Java虚拟机启动后初始化的，它主要负责加载%JAVA\_HOME%/jre/lib,-Xbootclasspath参数指定的路径以及%JAVA\_HOME%/jre/classes中的类。

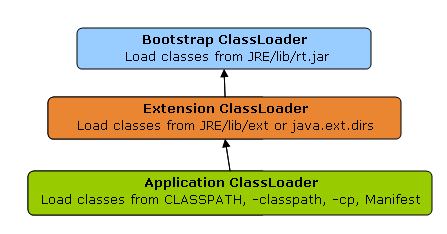
2.ExtClassLoader

Bootstrp loader加载ExtClassLoader,并且将ExtClassLoader的父加载器设置为Bootstrp loader.ExtClassLoader是用Java写的，具体来说就是 sun.misc.Launcher$ExtClassLoader，ExtClassLoader主要加载%JAVA\_HOME%/jre/lib/ext，此路径下的所有classes目录以及java.ext.dirs系统变量指定的路径中类库。

3.AppClassLoader

Bootstrp loader加载完ExtClassLoader后，就会加载AppClassLoader,并且将AppClassLoader的父加载器指定为 ExtClassLoader。AppClassLoader也是用Java写成的，它的实现类是 sun.misc.Launcher$AppClassLoader，另外我们知道ClassLoader中有个getSystemClassLoader方法,此方法返回的正是AppclassLoader.AppClassLoader主要负责加载classpath所指定的位置的类或者是jar文档，它也是Java程序默认的类加载器。

综上所述，它们之间的关系可以通过下图形象的描述：



为什么要有三个类加载器，一方面是分工，各自负责各自的区块，另一方面为了实现**委托模型**。

java采用了委托模型机制，这个机制简单来讲，就是“**类装载器有载入类的需求时，会先请示其Parent使用其搜索路径帮忙载入，如果Parent 找不到,那么才由自己依照自己的搜索路径搜索类**”

**类装载器ClassLoader（一个抽象类）描述一下JVM加载class文件的原理机制**

类装载器就是寻找类或接口字节码文件进行解析并构造JVM内部对象表示的组件，在java中类装载器把一个类装入JVM，经过以下步骤：

1. 装载：查找和导入Class文件
2. 链接：其中解析步骤是可以选择的 （a）检查：检查载入的class文件数据的正确性 （b）准备：给类的静态变量分配存储空间 （c）解析：将符号引用转成直接引
3. 初始化：对静态变量，静态代码块执行初始化工作

**定义自已的ClassLoader,既然JVM已经提供了默认的类加载器，为什么还要定义自已的类加载器呢？**

因为Java中提供的默认ClassLoader，只加载指定目录下的jar和class，如果我们想加载其它位置的类或jar时，比如：我要加载网络上的一个class文件，通过动态加载到内存之后，要调用这个类中的方法实现我的业务逻辑。在这样的情况下，默认的ClassLoader就不能满足我们的需求了，所以需要定义自己的ClassLoader。

**定义自已的类加载器分为两步：**

1、继承java.lang.ClassLoader

2、重写父类的findClass方法

**为什么偏偏只重写findClass方法？**

因为JDK已经在loadClass方法中帮我们实现了ClassLoader搜索类的算法，当在loadClass方法中搜索不到类时，loadClass方法就会调用findClass方法来搜索类，所以我们只需重写该方法即可。如没有特殊的要求，一般不建议重写loadClass搜索类的算法。

**java的设计模式**

**大体上分为三大类：**

创建型模式（5种）：工厂方法模式，抽象工厂模式，单例模式，建造者模式，原型模式。

结构型模式（7种）：适配器模式，装饰器模式，代理模式，外观模式，桥接模式，组合模式，享元模式。

行为型模式（11种）：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。