# JAVA

## 1. 八种基本数据类型的大小，以及他们的封装类

**八种基本数据类型：**boolean, byte, short, int, float, double, long, char

**对应的封装类：**Boolean, Byte, Short, Float, Double, Long, Character

**封装类与基本数据类型的区别：**

**1.1**封装类可以有方法和属性，可以利用这些封装类来处理数据。

1.2 初始值不同，比如int型数据的默认值是0，而Integer型数据的基本数据类型是null.

double比较大小：在精度（小数点后面16位及其以内）可以比较大小(==,>=,<=),超过16为就不可以了。

## 2. 关于String

String不是基本数据类型，它是final修饰的，所以也不可以继承。

**2.1 String的两种赋值方式：**

String a = “abc”;

String b = new String(“abc”);

a：在字符串常量池中创建了一个对象然后把这个对象的引用传给a。

b:创建了两个对象，一个是在字符串常量池中的字符串对象，另一个是new的时候在堆内存中创建的String对象，而new String这个对象中保存了字符串对象。

**2.2 字符串常量池**

用到 “xxx”的地方都会在字符串常量池中创建字符串对象，如果常量池中有这个对象，就直接返回这个对象地址，如果不存在这个字符串对象，就要重新创建。

**2.3 字符串的比较**

String类型的比较函数是依次比两个字符串的第i位，出现不相同的字符就直接返回了。如果在Min(a.length,b.length)范围内都相同，就返回a.length-b.length。

## 3.  Switch能否用string做参数

java7之前是不允许的，java7之后可以使用，因为switch时，比较的其实是String的hashCode的值，hashCode返回的是int型。

## 4. equals与==的区别

equals是基类Object的方法，而==是操作符。

Object中的equals使用就是==，判断两个对象的地址是否相等。

java中的equals方法是留给开发者覆写的，让开发者自己定义什么情况下两个Object是equals的。

4.1 基本数据类型的==比较的都是值

4.2 引用类型的==比较的都是地址，基本数据类型的包装类覆写了Object的equals，比较的是值。String类型的equals比较的也是值。

## 5. Object有哪些公用方法

**clone()**

克隆，java中除了8种基本数据类型传参数是值传递，其它的类对象传参都是引用传递，如果不想在函数改变原参数的值，可以用参数的clone方法。clone实现的是对象的浅拷贝。

**浅拷贝：**在被拷贝对象的属性中，对于int这样的基本数据类型，直接复制就可以了，而对于类似String这样的引用数据类型，直接拷贝其引用值。

**深拷贝：**对于String这样的数据类型，会重新生成一个新的对象。如果要在一个类中实现深拷贝，除了这个类要实现Cloneable，覆写clone()方法之外，还需要拷贝它内部的引用对象，这就要求内部的应用类型也要实现Cloneable，覆写clone()方法。

如果在拷贝一个对象时，要想让这个拷贝的对象和源对象完全彼此独立，那么在引用链上的每一级对象都要被显式的拷贝。

Object中的clone实现的都是浅拷贝，如果需要深拷贝需要自己实现。

注：只有实现了Cloneable接口才可以调用这个方法。Cloneable接口中其实什么方法都没有，它只是起到标记的作用。

clone的重要作用，用于不可变对象的创建。

**getClass方法**

final方法，获取运行时的类型。

**toString方法**

一般由子类覆盖，输出引用数据类型的地址，对于基本数据类型的封装类，输出对于的基本数据类型的值，如果是数组，输出数组元素。

**finalize方法**

垃圾回收器在释放对象资源之前，会调用finalize方法，做一些收尾的工作(比如释放这个对象操作的资源)。垃圾回收和内存相关，不知道垃圾回收什么时候会执行，所以对象资源不一定会被释放。

**equals方法**

比较两个对象是否相等，在Object中实现和==是一样的，子类一般会覆写这个方法。

**hashCode方法**

获取到对象的hash值。

**equals和hashCode的比较：**

1.如果两个对象的equals比较相等，那么hashCode一定相等；

2.但是hashCode相等，equals比较不一定相等。

hashCode相当于是对象的编码，equals相等于比较对象的物理内容。

hashCode一般用在hash查找中，这样可以减少使用equals方法，在具有哈希功能的Collection中会用到。HashSet/HashMap在添加元素时会先判断hashCode的值，如果一样再用equals比较内容；如果hashCode不一样就不用比较eqauls了。

一般情况下，hashCode()和eqauls()会在子类中重写，协助HashMap,HashSet,HashTable对收集的对象进行定位，搜索和比较。

**wait方法**

调用该方法，当前线程释放对象锁，进入等待状态。可以传入最长等待时间，但是一般不进行设置。

wait之后线程被唤醒的方法：

1. 时间间隔到了；

2. 其他线程调用了该对象的notify()/notifyAll方法。

3. 其他线程调用了interrupt中断该线程。

**notify方法**

随机唤醒一个在该对象锁上等待的线程。

**notifyAll方法**

唤醒在该对象锁上等待的所有线程。

## 6. ArrayList、LinkedList、Vector的区别

首先，ArrayList，LinkedList，Vector都是List的子类。

1. Vector动态数组，同步的，增删，查询都很慢。

2. ArrayList动态数组，不同步，在中间增删慢，可能导致数组重新分配，时间复杂度为O(n)，因为增删需要移动后面的元素。查询快，基于数组的查询，时间复杂度为O(1)，增删最后一个元素也快.

3. LinkedList链表，也是动态增长，内部有一个Entry的静态内部类，每个Entry对象都对应链表中的一个元素，还有它在LinkedList中的上一个元素和下一个元素，所以LinkedList的控件开销很大。增删快（O(1)），不支持随机查询，查询较慢。

**ArrayList和Vector的区别：**

ArrayList和Vector都存在一定程度的空间浪费。

数据增长：

ArrayList的初始大小是10，空间增长的计算公式：新容量=(旧容量\*3)/2+1；所以容量不够时，每次大概会增长50%。

Vector的默认增容量是是原来容量的2倍，但是Vector多了一个构造函数（ArrayList有3个构造函数），可以设置增长因子，每次容量超不足时会在原来容量的基础上加上这个增量因子。所以Vector的空间浪费可能会小一些。

## 7. Java的四种引用，强弱软虚，用到的场景

**强引用**

强引用没有对应的具体的数据类型，如果new的对象赋值给了一个变量，那么这个变量都持有一个强引用。gc就算在内存溢出的情况的都不会回收强引用对象。如果**函数内部**new的强引用对象，对象引用保存在栈中，对象的内容保存在堆中，函数结束之后，这个强引用就不存在了，这个对象就会被回收。如果是全局变量，需要注意在不用的时候将其赋值为null，这样这个不用的对象就可以被回收了。如果是一个数组对象，直接将数组persons = null，其实数组中的Person对象的强引用还是存在的，只有person[i]=null，存储的Person对象才有可能被gc回收。

软引用

SoftReference，如果一个对象具有软引用，内存空间足够的时候，gc不会回收它，当内存不足时，就会被gc回收。只要gc没有回收，程序就可以继续使用。软引用可以用来实现内存敏感的高速缓存。

**弱引用**

WeakReference，与软引用相比，所以用有更短暂的生命周期，当垃圾回收线程扫描它锁管辖的内存区域时，一旦发现只具有弱引用的对象，不管当前内存充足与否，都会立刻被回收。但是垃圾回收的优先级很低，因此不会立刻发现只具有弱引用的对象。

由于这个引用不会在对象的垃圾回收中造成任何影响，所以，如果你想使用一个对象，但是不想介入它的生命周期时，就可以使用弱引用。

**虚引用**

形同虚设，和没有引用一样，虚引用不会决定对象的生命周期，如果一个对象只具有一个虚引用，那么它随时可能被垃圾回收器回收。

和软/弱引用不同的是，虚引用必须和一个ReferenceQueue联合使用，当垃圾回收器准备回收一个对象时，发现它还有一个虚引用，就会把这个虚引用加入到与之相关的引用队列中。程序中可以通过判断引用队列中是否存在这个对象的虚引用，来判断该对象是否将要被回收，用来跟踪对象被垃圾回收的活动。

## 8. Hashcode的作用

用来加快查找，减少使用equals的频率。如果查找一个元素时，如果使用hashcode就会快速定位到这个元素，避免从头开始一个一个比较equals，但是我们不能保证由hashCode找到的这个元素是不是我们要找的，只能说它在hashCode标记这个篮子里，这个时候就需要使用equals了。详见5的hashCode方法。

## 9. String、StringBuffer与StringBuilder的区别

StringBuffer，StringBuilder和String一样，都是用来表示字符串的。String类是一个不可变类，任何对String的改变都会引发新的String对象的生成。

**StringBuffer和StringBuilder**

StringBuffer和StringBuilder是可变的，增加、删除、修改数组都不会产生新的对象。

此外StringBuffer是线程安全的，里面很多方法都是同步，StringBuilder是线程不安全的，只能在单线程之下保证不出错，单线程下可以保证StringBuilder比StringBuffer快，String的连接操作是最慢的。

StringBuffer和StringBuilder的初始大小都是16个字节，也可以指定初始大小，以后每次添加字符串出现容器大小不足时，都会在原来的基础增加2倍（newCapacity = value.length \* 2 + 2）

## 10. Map、Set、List、Queue、Stack的特点与用法

**Collection接口**

对象集合，其中的对象属于同一个类别，每个元素就是一个独立的对象，服从某种规则规则。

集合的特点：存储对象的容器，长度可变，不可存储基本数据类型。

**子类接口：List，Set，Queue**

List：集合中的对象和插入元素的顺序一致。允许重复。

Set：集合中的元素是无序的，不允许重复。检索效率低下，插入和删除效率高，因为不会引起其他元素改变其位置。

Queue：用于模拟队列这种“先进先出”的数据结构。除了Collection的方法外，还实现了另一套插入，移除，检查的函数offer,poll,peek.比较如下：



Deque（Queue的子接口）：实现的是双端队列

**Map接口**

Map中的对象是一组键值对。

Map和Collection比较：

1. Map中一次添加两个元素（key,value），Collection中一次添加一个元素。

2. Map也称为双列集合，Collection称为单列集合。

3. Map集合存储键值对，保证键的唯一性。

## 11. HashMap和Hashtable的区别

**1. synchronized**

HashTable是线程安全的，说明多个线程可以共享一个HashTable，HashMap不是线程安全的，所以在单线程环境下使用HashMap要比Hashtable快，而在多线程环境下，HashMap需要做额外的同步，Hashtable则 不需要。现在Hashtable已经被废弃了，在多线程环境下推荐使用CurrentHashMap代替Hashtable。

**2.父类**

Hashtable的父类是Dictionary，这个父类给Hashtable提供了elements对外方法，获取到Enumeration对象，用于迭代哈希表；Hashtable中保留了contains()，这个方法和containsValue()效果是一样的。HashMap的父类是AbstractMap中去掉了contains()，使用containsKey()和containsValue()代替了contains()。此外，两个类都提供了键值对映射的服务，可以增删改查键值对。

**3.null Key & null Value**

Hashtable不允许null Key和null Value，在遇到null的时候会抛出NullPointerException异常；而HashMap是支持null Key和null Value，这是因为HashMap在实现层做了一些处理，在遇到null Key的时候，直接将hashCode设置为0，从而放在哈希表的第一个bucket中。

**4.内部数据结构**

Hashtable和HashMap内部都是哈希表，Hashtable内部是使用Entry数组来模拟hash表；而HashMap内部是使用Node数组来模拟Hash表，Node其实就是Entry的子类，两个只是在计算hash的值的时候有所区别，Entry和Node内部都是结构是一样，有hash表示哈希值，Key和Value，还有next指向同一个bucket里的下一个Entry对象。所以有多少个键值对就有多少个Entry对象。哈希表table的一个位置代表一个一个桶，对应一个唯一的hash值。在如数据的时候，Hashtable和HashMap都是先判断这个hash值对应的位置是否有元素，如果没有元素就直接插入这个位置，如果有元素就判断这个元素和当前插入的元素是否是同一个，如果不是同一个，说明这个元素当前的hash值对应的桶里面已经有元素了，这个时候就需要用借助链表，把当前插入的元素接到桶的末尾。借助链表解决了hash冲突的问题，但是查找的时候非常不方便，所以在jdk1.8之后，映射到同一个桶中的Entry对象使用了红黑树来存储，加快查找效率。

**5.初始大小**

Hahstable的初始大小默认是11，当哈希表大小不足时，增加方式是old\*2+1；而HashMap的默认大小是16，增加方式是old\*2，所以HashMap的大小永远是2的n次幂。可以看出Hashtable在针对哈希表的大小上会尽量使用奇数和素数，这使得取模运算得到的分布更加均匀；而另一方面，HashMap尽量使用2的幂次，这使用取模运算可以使用位运算来代替，在hash计算效率上更胜一筹。由于HashMap将哈希表的大小固定为2的幂，自然引入了分布不均匀的问题，HashMap为了解决这个问题，在计算key的hash值的之后，在hashCode的基础上做了一些改动，利用位运算来打散数据分布。

## 12. HashMap和ConcurrentHashMap的区别，HashMap的底层源码

### HashMap

HashMap根据键的hashCode值来存储数据，能够快速定位到元素，所以查询效率比较高。HashMap虽然允许null Key和null Value，但是它只能接收一对null Key，如果多次添加key为null的键值对，它只会保留最后添加的一次，因为null key对应的hashCode都是0，值被覆盖了。HashMap是线程不安全的。

#### HashMap底层源码

在java1.8以前，HashMap是基于Node数据+链表来实现哈希表的，根据key的hash值来定位到桶，使用链表来处理冲突，但是当hash冲突的频率增高，导致一个桶中存储太多元素时，基于链表的查找使得查找效率变低，所以在jdk1.8是做出了改进，基于Node数据+链表+红黑树，当链表长度超过阈值(8)时，将链表转为红黑树，加快查找效率。

**1. 几个重要参数**

Node[] table：哈希表，初始大小为16，以后每次resize(扩容)，都是old\*2。

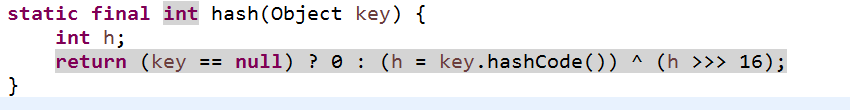
size：哈希表中实际元素的个数。

loadFactor：哈希表的负载因子，默认情况下是0.75.

threshold：当前哈希表中所能容纳的最大数据量。threshold =capacity \* load factor，也就是说负载因子越大，所能容纳的键值对越多。当哈希表中元素超过threshold的值时，就会调用resize()重新调整哈希表的大小。

**2. 几个重要方法**

**2.1 hash()**



hash()用于计算key对应的hash值，然后根据这个hash和table.length进行取模运算，确定这个键值对应该放在哈希表的哪个桶里面。在计算哈希值的时候，是先得到key的hashCode，然后对hashCode进行位运算，h>>>16是取其高16为位，而h^(h>>>16)其实取的是低16位。

**2.2 put()**

在加入新的键值对的时候会调用put()，在put新元素的时候，会经过一下几个过程：

a. 判断table是否为空，如果为空就调用resize()初始化table。

b. 根据key的hash值和table.length进行取模运算，计算key对应的桶下标i，判断table[i]如果为null，就直接插入。

c. 如果table[i]不为空，就判断当前位置的元素key和插入的key是否相同，如果相同，就覆盖原来的元素；如果不相同，说明hash冲突。

d. 处理hash冲突有两种方法，可以在链表末尾追加元素处理冲突，如果桶中元素超过8,就要改为红黑树存储。所以这里会先判断使用红黑树存储还是使用链表存储。

e. 插入成功后，size++，然后判断size是否超过threshold，如果超过就调用resize扩容。

**2.3 get()**

先通过key的hash值计算键值计算i，判断table[i]是不是当前要找的元素，如果不是，就判断桶中的元素使用链表还是红黑树存储，如果使用红黑树存储，就遍历红黑树找到元素，否则就遍历链表。

### ConcurrentHashMap

ConcurrentHashMap是Hashtable的替代，它能保证线程安全，而且效率要比Hashtable高很多。ConcurrentHashMap 允许多个修改读取操作并发进行，其关键在于它使用了锁分离技术，使用多个锁来控制对哈希表不同的部分的修改。ConcurrentHashMap针对写（put）和修改（remove）操作进行加锁，效率也会高很多。

**在jdk1.8以前，**ConcurrentHashMap中有一个Segment[]数组，每个segment包含很多个Node/Entry，在put的时候先获取到key的hash值，然后计算应该放在哪个segment段中，然后对这个段加锁，再计算元素应该放在哪个Node上，这使得其他线程可以对其他段的数据进行读写操作，真正的实现了并发操作。

**而在jdk1.8中做出了较大改进**：

1. 取消了segments字段，直接使用**transient** **volatile** Node<K,V>[] table保存数据，采用table数组元素作为锁，针对单个结点加锁，降低了并发冲突的概率。

2.将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构，使用了和HashMap一样的底层存储结构。是的查询性能提高。

## 13. TreeMap、HashMap、LindedHashMap的区别

## 14. Collection包结构，与Collections的区别。

**15. 实现同步的几种方式**

synchronized：同步函数或同步代码块，保证可见性和原子性

volatile：关键字，修饰变量，保证可见性和部分原子性。

Lock

**volatile变量自身具有的特性：**

**可见性：**任何线程对volatile变量的读，总是能看到任何线程对变量最后的写入的结果。

**关于volatile关于原子性的说明：**

jvm运行时刻的内存分配，有一块区域时虚拟机栈，每个线程都会一个线程栈，保存线程运行时的变量信息。当线程访问某一个变量的值时，会根据对象的引用，在堆内存中找到对象的值，然后把具体值load到线程本地内存中，保存一个副本，之后对这个副本进行修改操作，在修改完之后的某个时刻，写入到主存之中。

根据上面的阐述可以知道，对任意单个volatile变量的读写具有原子性，而类似于volatile++这样的复合操作不具有原子性。

# 计算机网络

## 1. TCP和UDP协议的区别，它们位于哪一层？

TCP协议和UDP协议都属于传输层协议。

OSI参考模型自顶向下分别是：应用层，表示层，会话层，传输层，网络层，数据链路层，物理层。

TCP和UDP的区别：

TCP：面向连接的，可靠，传输大量数据，传输速度慢。

UDP：无连接的，不可靠，传输少量数据，传输速度慢。

## 2.网络互连的中继系统

将网络互相连接起来要使用一些中间设备（或中间系统），ISO的术语称之为中继（relay）系统。根据中继系统所在的层次，可以有以下**五种中继系统**：

1.物理层（即常说的第一层、层L1）中继系统，即转发器（repeater）。

2.数据链路层（即第二层，层L2），即网桥或桥接器（bridge）。

3.网络层（第三层，层L3）中继系统，即路由器（router）。

4.网桥和路由器的混合物桥路器（brouter）兼有网桥和路由器的功能。

5.在网络层以上的中继系统，即网关（gateway）.

3. 路由器和交换机的区别。

路由器是网络层的分组交换设备，负责把数据（IP数据包）转发到正确的网络，

包括：

1.IP数据报的转发，包括数据包的寻址和传送。

2.子网隔离

3.维护路由表，并与其他路由器交换路由信息。

4.IP数据报的差错处理以及简单的拥塞控制。