# Java常见面试题及答案

作者 littleKang (/u/471df0ed9fe6) + 关注
2015.10.29 14:42* 字数 9523 阅读 30556 评论 36 喜欢 291 赞赏 1
(/u/471df0ed9fe6)

## java常见面试题及答案

### 1.什么是Java虚拟机？为什么Java被称作是“平台无关的编程语言”?

> Java 虚拟机是一个可以执行 Java 字节码的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。
> Java 被设计成允许应用程序可以运行在任意的平台，而不需要程序员为每一个平台单独重写或者是重新编译。
> Java 虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

### 2.JDK和JRE的区别是什么?

> *JDK:* java开发工具包,包含了JRE、编译器和其它工具（如：javaDOc、java调试器)
> *JRE:* java运行环境,包含java虚拟机和java程序所需的核心类库。
> 如果只是想跑java程序，那么只需安装JRE，如果要写java程序并且运行，那就需要JDK了。

### 3."static"关键字是什么意思？Java中是否可以覆盖一个private或者是static的方法?

> 如果一个类的变量或者方法前面有**static**修饰，那么表明这个方法或者变量属于这个类，也就是说可以在不创建对象的情况下直接使用
> 当父类的方法被**private**修饰时，表明该方法为父类私有，对其他任何类都是不可见的，因此如果子类定了一个与父类一样的方法，这对于子类来说相当于是一个新的私有方法，且如果要进行向上转型，然后去调用该“覆盖方法”，会产生编译错误

```
class Parent {
    private fun() {
        ...
    }
}
class Child extends Parent {
    private fun() {
        ...
    }
}
class Test {
    public static void main(String[] args) {
        Parent c = new Child();
        c.fun(); //编译出错
    }
}
```

> **static**方法时编译时静态绑定的，属于类，而覆盖是运行时动态绑定的(动态绑定的多态),因此不能覆盖.

## 4.Java支持的基本数据类型有哪些？什么是自动拆装箱?

> java支持的基本数据类型有以下9种:byte,shot,int,long,float,double,char,boolean,void.
> 自动拆装箱是java从jdk1.5引用，目的是将原始类型自动的装换为相对应的对象，也可以逆向进行，即拆箱。这也体现java中一切皆对象的宗旨。
> 所谓自动装箱就是将原始类型自动的转换为对应的对象，而拆箱就是将对象类型转换为基本类型。java中的自动拆装箱通常发生在变量赋值的过程中，如：

```
Integer object = 3; //自动装箱
int o = object; //拆箱
```

> 在java中，应该注意自动拆装箱，因为有时可能因为java自动装箱机制，而导致创建了许多对象，对于内存小的平台会造成压力。

## 5. 覆盖和重载是什么?

> **覆盖**也叫**重写**，发生在子类与父类之间，表示子类中的方法可以与父类中的某个方法的名称和参数完全相同，通过子类创建的实例对象调用这个方法时，将调用子类中的定义方法，这相当于把父类中定义的那个完全相同的方法给覆盖了，这也是面向对象编程的多态性的一种表现。
> **重载**是指在一个类中，可以有多个相同名称的方法，但是他们的参数列表的个数或类型不同，当调用该方法时，根据传递的参数类型调用对应参数列表的方法。
> 当参数列表相同但返回值不同时，将会出现编译错误，这并不是重载，因为jvm无法根据返回值类型来判断应该调用哪个方法。

## 6.Java支持多继承么？如果不支持，如何实现?

> 在java中是单继承的，也就是说一个类只能继承一个父类。
> java中实现多继承有两种方式,一是接口，而是内部类.

```
//实现多个接口 如果两个接口的变量相同 那么在调用该变量的时候 编译出错
interface interface1 {
    static String field = "dd";
    public void fun1();
    }
interface interface2 {
static String field = "dddd";
    public void fun2();
    }
class child implements interface1,interface2 {
    static String field = "dddd";
    @Override
    public void fun2() {
    }

    @Override
    public void fun1() {
    }
}

//内部类 间接多继承
class Child {
class Father {
    private void strong() {
        System.out.println("父类");
    }
}
class Mother {
    public void getCute() {
        System.out.println("母亲");
    }
}
public void getStrong() {
    Father f = new Father();
    f.strong();
    }
public void getCute() {
    Mother m = new Mother();
    m.getCute();
    }
}
```

## 7.什么是值传递和引用传递？java中是值传递还是引用传递，还是都有？

> **值传递** 就是在方法调用的时候，实参是将自己的一份拷贝赋给形参，在方法内，对该参数值的修改不影响原来实参，常见的例子就是刚开始学习c语言的时候那个交换方法的例子了。
> **引用传递** 是在方法调用的时候，实参将自己的地址传递给形参，此时方法内对该参数值的改变，就是对该实参的实际操作。
> 在java中只有一种传递方式，那就是**值传递**.可能比较让人迷惑的就是java中的对象传递时，对形参的改变依然会影响到该对象的内容。
> 下面这个例子来说明java中是值传递.

```
    public class Test {
    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("hello ");
        getString(sb);
        System.out.println(sb);
    }
    public static void getString(StringBuffer s) {
        //s = new StringBuffer("ha");
        s.append("world");
    }
}
```

> 在上面这个例子中,当前输出结果为:hello world。这并没有什么问题，可能就是大家平常所理解的引用传递，那么当然会改变StringBuffer的内容。但是如果把上面的注释去掉，那么就会输出:hello.此时sb的值并没有变成ha hello. 假如说是引用传递的话，那么形参的s也就是sb的地址，此时在方法里new StringBuffer（），并将该对象赋给s，也就是说s现在指向了这个新创建的对象.按照引用传递的说法，此时对s的改变就是对sb的操作，也就是说sb应该也指向新创建的对象，那么输出的结果应该为ha world.但实际上输出的仅是hello.这说明sb指向的还是原来的对象，而形参s指向的才是创建的对象,这也就验证了java中的对象传递也是值传递。

## 8.接口和抽象类的区别是什么?

> 不同点在于：
>
> 1. 接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。
> 2. 类可以实现很多个接口，但是只能继承一个抽象类
> 3. 类如果要实现一个接口，它必须要实现接口声明的所有方法。但是，类可以不实现抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。
> 4. 抽象类可以在不提供接口方法实现的情况下实现接口。
> 5. Java 接口中声明的变量默认都是 final 的。抽象类可以包含非 final 的变量。
> 6. Java 接口中的成员函数默认是 public 的。抽象类的成员函数可以是 private，protected 或者是 public 。
> 7. 接口是绝对抽象的，不可以被实例化(java 8已支持在接口中实现默认的方法)。抽象类也不可以被实例化，但是，如果它包含 main 方法的话是可以被调用的。

## 9.构造器（constructor）是否可被重写（override）？

> 构造方法是不能被子类重写的，但是构造方法可以重载，也就是说一个类可以有多个构造方法。

## 10.Math.round(11.5) 等于多少? Math.round(-11.5)等于多少?

> Math.round(11.5)==12 Math.round(-11.5)==-11 round 方法返回与参数 最接近的长整数，参数加 1/2 后求其 floor.

## 11. String, StringBuffer StringBuilder的区别。

> tring 的长度是不可变的；
>
> StringBuffer的长度是可变的，如果你对字符串中的内容经常进行操作，特别是内容要修改时，那么使用 StringBuffer，如果最后需要 >String，那么使用 StringBuffer 的 toString() 方法；线程安全；
>
> StringBuilder 是从 JDK 5 开始，为StringBuffer该类补充了一个单个线程使用的等价类；通常应该优先使用 StringBuilder 类，因>为它支持所有相同的操作，但由于它不执行同步，所以速度更快。
> 使用字符串的时候要特别小心，如果对一个字符串要经常改变的话，就一定不要用String,否则会创建许多无用的对象出来.
> 来看一下比较

```
String s = "hello"+"world"+"i love you";
StringBuffer Sb = new StringBuilder("hello").append("world").append("i love you");
```

> 这个时候s有多个字符串进行拼接，按理来说会有多个对象产生，但是jvm会对此进行一个优化，也就是说只创建了一个对象，此时它的执行速度要比StringBuffer拼接快.再看下面这个:

```
String s2 = "hello";
String s3 = "world";
String s4 = "i love you";
String s1 = s2 + s3 + s4;
```

> 上面这种情况，就会多创建出来三个对象，造成了内存空间的浪费.

## 12.JVM内存分哪几个区，每个区的作用是什么？

java虚拟机主要分为以下一个区:

**方法区：**

1. 有时候也成为**永久代**，在该区内很少发生垃圾回收，但是并不代表不发生GC，在这里进行的GC主要是对方法区里的常量池和对类型的卸载
2. 方法区主要用来存储已被虚拟机加载的类的信息、常量、静态变量和即时编译器编译后的代码等数据。
3. 该区域是被线程共享的。
4. 方法区里有一个运行时常量池，用于存放静态编译产生的字面量和符号引用。该常量池具有动态性，也就是说常量并不一定是编译时确定，运行时生成的常量也会存在这个常量池中。

**虚拟机栈:**

1. 虚拟机栈也就是我们平常所称的**栈内存**,它为java方法服务，每个方法在执行的时候都会创建一个栈帧，用于存储局部变量表、操作数栈、动态链接和方法出口等信息。
2. 虚拟机栈是线程私有的，它的生命周期与线程相同。
3. 局部变量表里存储的是基本数据类型、returnAddress类型（指向一条字节码指令的地址）和对象引用，这个对象引用有可能是指向对象起始地址的一个指针，也有可能是代表对象的句柄或者与对象相关联的位置。局部变量所需的内存空间在编译器间确定
   4.操作数栈的作用主要用来存储运算结果以及运算的操作数，它不同于局部变量表通过索引来访问，而是压栈和出栈的方式
   5.每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接.动态链接就是将常量池中的符号引用在运行期转化为直接引用。

**本地方法栈**
本地方法栈和虚拟机栈类似，只不过本地方法栈为Native方法服务。

**堆**
java堆是所有线程所共享的一块内存，在虚拟机启动时创建，几乎所有的对象实例都在这里创建，因此该区域经常发生垃圾回收操作。

**程序计数器**
内存空间小，字节码解释器工作时通过改变这个计数值可以选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理和线程恢复等功能都需要依赖这个计数器完成。该内存区域是唯一一个java虚拟机规范没有规定任何OOM情况的区域。

## 13.如和判断一个对象是否存活?(或者GC对象的判定方法)

判断一个对象是否存活有两种方法:

1. **引用计数法**
   所谓引用计数法就是给每一个对象设置一个引用计数器，每当有一个地方引用这个对象时，就将计数器加一，引用失效时，计数器就减一。当一个对象的引用计数器为零时，说明此对象没有被引用，也就是“死对象”,将会被垃圾回收.
   引用计数法有一个缺陷就是无法解决循环引用问题，也就是说当对象A引用对象B，对象B又引用者对象A，那么此时A,B对象的引用计数器都不为零，也就造成无法完成垃圾回收，所以主流的虚拟机都没有采用这种算法。

2.**可达性算法**(引用链法)
该算法的思想是：从一个被称为**GC Roots**的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。
在java中可以作为GC Roots的对象有以下几种:

- 虚拟机栈中引用的对象
- 方法区类静态属性引用的对象
- 方法区常量池引用的对象
- 本地方法栈JNI引用的对象

虽然这些算法可以判定一个对象是否能被回收，但是当满足上述条件时，一个对象比**不一定会被回收**。当一个对象不可达GC Root时，这个对象并**不会立马被回收**，而是出于一个死缓的阶段，若要被真正的回收需要经历两次标记
如果对象在可达性分析中没有与GC Root的引用链，那么此时就会被第一次标记并且进行一次筛选，筛选的条件是是否有必要执行finalize()方法。当对象没有覆盖finalize()方法或者已被虚拟机调用过，那么就认为是没必要的。
如果该对象有必要执行finalize()方法，那么这个对象将会放在一个称为F-Queue的对队列中，虚拟机会触发一个Finalize()线程去执行，此线程是低优先级的，并且虚拟机不会承诺一直等待它运行完，这是因为如果finalize()执行缓慢或者发生了死锁，那么就会造成F-Queue队列一直等待，造成了内存回收系统的崩溃。GC对处于F-Queue中的对象进行第二次被标记，这时，该对象将被移除"即将回收"集合，等待回收。

## 14.简述java垃圾回收机制?

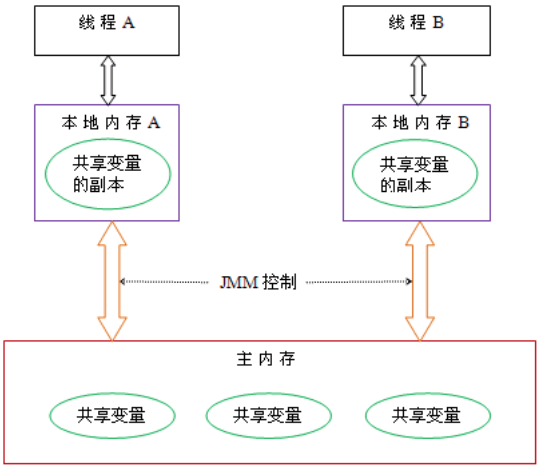
在java中，程序员是不需要显示的去释放一个对象的内存的，而是由虚拟机自行执行。在JVM中，有一个垃圾回收线程，它是低优先级的，在正常情况下是不会执行的，只有在虚拟机空闲或者当前堆内存不足时，才会触发执行，扫面那些没有被任何引用的对象，并将它们添加到要回收的集合中，进行回收。

## 15.java中垃圾收集的方法有哪些?

1. **标记-清除:**
   这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记哪些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：1.效率不高，标记和清除的效率都很低；2.会产生大量不连续的内存碎片，导致以后程序在分配较大的对象时，由于没有充足的连续内存而提前触发一次GC动作。
2. **复制算法:**
   为了解决效率问题，复制算法将可用内存按容量划分为相等的两部分，然后每次只使用其中的一块，当一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清楚完第一块内存，再将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一般的内存。
   于是将该算法进行了改进，内存区域不再是按照1：1去划分，而是将内存划分为8:1:1三部分，较大那份内存交Eden区，其余是两块较小的内存区叫Survior区。每次都会优先使用Eden区，若Eden区满，就将对象复制到第二块内存区上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制复制到老年代中。(java堆又分为新生代和老年代)
3. **标记-整理**
   该算法主要是为了解决标记-清除，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收对象移动到一端，然后清除掉端边界以外的对象，这样就不会产生内存碎片了。
   **分代收集**
   现在的虚拟机垃圾收集大多采用这种方式，它根据对象的生存周期，将堆分为新生代和老年代。在新生代中，由于对象生存期短，每次回收都会有大量对象死去，那么这时就采用**复制**算法。老年代里的对象存活率较高，没有额外的空间进行分配担保，所以可以使用**标记-整理** 或者 **标记-清除**。

## 16.java内存模型

java内存模型(JMM)是线程间通信的控制机制.JMM定义了主内存和线程之间抽象关系。线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。Java内存模型的抽象示意图如下：




从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。
2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。
   写的很好:http://www.infoq.com/cn/articles/java-memory-model-1
   (http://www.infoq.com/cn/articles/java-memory-model-1)

## 17.java类加载过程?

java类加载需要经历一下7个过程：

**加载**

加载时类加载的第一个过程，在这个阶段，将完成一下三件事情：

1. 通过一个类的全限定名获取该类的二进制流。
2. 将该二进制流中的静态存储结构转化为方法去运行时数据结构。
3. 在内存中生成该类的Class对象，作为该类的数据访问入口。

**验证**

验证的目的是为了确保Class文件的字节流中的信息不回危害到虚拟机.在该阶段主要完成以下四钟验证:

1. 文件格式验证：验证字节流是否符合Class文件的规范，如主次版本号是否在当前虚拟机范围内，常量池中的常量是否有不被支持的类型.
2. 元数据验证:对字节码描述的信息进行语义分析，如这个类是否有父类，是否集成了不被继承的类等。
3. 字节码验证：是整个验证过程中最复杂的一个阶段，通过验证数据流和控制流的分析，确定程序语义是否正确，主要针对方法体的验证。如：方法中的类型转换是否正确，跳转指令是否正确等。
4. 符号引用验证：这个动作在后面的解析过程中发生，主要是为了确保解析动作能正确执行。

**准备**

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

```
public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。
```

**解析**

该阶段主要完成符号引用到直接引用的转换动作。解析动作并不一定在初始化动作完成之前，也有可能在初始化之后。

**初始化**

初始化时类加载的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

## 18. 简述java类加载机制?

虚拟机把描述类的数据从Class文件加载到内存，并对数据进行校验，解析和初始化，最终形成可以被虚拟机直接使用的java类型。

## 19. 类加载器双亲委派模型机制?

当一个类收到了类加载请求时，不会自己先去加载这个类，而是将其委派给父类，由父类去加载，如果此时父类不能加载，反馈给子类，由子类去完成类的加载。

## 20.什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。
主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载java核心类库，无法被java程序直接引用。
2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。
4. 用户自定义类加载器，通过继承 java.lang.ClassLoader类的方式实现。

## 21.简述java内存分配与回收策率以及Minor GC和Major GC

1. 对象优先在堆的Eden区分配。
2. 大对象直接进入老年代.
3. 长期存活的对象将直接进入老年代.
   当Eden区没有足够的空间进行分配时，虚拟机会执行一次Minor GC.Minor Gc通常发生在新生代的Eden区，在这个区的对象生存期短，往往发生Gc的频率较高，回收速度比较快;Full Gc/Major GC 发生在老年代，一般情况下，触发老年代GC的时候不会触发Minor GC,但是通过配置，可以在Full GC之前进行一次Minor GC这样可以加快老年代的回收速度。

## 22.HashMap的工作原理是什么?

HashMap内部是通过一个数组实现的，只是这个数组比较特殊，数组里存储的元素是一个Entry实体(jdk 8为Node)，这个Entry实体主要包含key、value以及一个指向自身的next指针。HashMap是基于hashing实现的，当我们进行put操作时，根据传递的key值得到它的hashcode，然后再用这个hashcode与数组的长度进行模运算，得到一个int值，就是Entry要存储在数组的位置（下标）；当通过get方法获取指定key的值时，会根据这个key算出它的hash值（数组下标），根据这个hash值获取数组下标对应的Entry，然后判断Entry里的key，hash值或者通过equals()比较是否与要查找的相同，如果相同，返回value，否则的话，遍历该链表（有可能就只有一个Entry，此时直接返回null），直到找到为止，否则返回null。
HashMap之所以在每个数组元素存储的是一个链表，是为了解决hash冲突问题，当两个对象的hash值相等时，那么一个位置肯定是放不下两个值的，于是hashmap采用链表来解决这种冲突，hash值相等的两个元素会形成一个链表。

## 23.HashMap与HashTable的区别是什么?

1.HashTable基于Dictionary类，而HashMap是基于AbstractMap。Dictionary是任何可将键映射到相应值的类的抽象父类，而AbstractMap是基于Map接口的实现，它以最大限度地减少实现此接口所需的工作。（在java 8中我查看源码发现Hashtable并没有继承Dictionary,而且里面也没有同步方法，是不是java 8中Hashtable不在同步的了？有没有人解释一下？）

1. HashMap的key和value都允许为null，而Hashtable的key和value都不允许为null。HashMap遇到key为null的时候，调用putForNullKey方法进行处理，而对value没有处理；Hashtable遇到null，直接返回NullPointerException。
2. Hashtable是同步的，而HashMap是非同步的，但是我们也可以通过Collections.synchronizedMap(hashMap),使其实现同步。

## 24.CorrentHashMap的工作原理?

**jdk 1.6版:** ConcurrenHashMap可以说是HashMap的升级版，ConcurrentHashMap是线程安全的，但是与Hashtablea相比，实现线程安全的方式不同。Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。ConcurrentHashMap是采用分离锁的方式，它并没有对整个hash表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对hash表其他地方的访问。
具体实现:ConcurrentHashMap内部有一个Segment<K,V>数组,该Segment对象可以充当锁。Segment对象内部有一个HashEntry<K,V>数组，于是每个Segment可以守护若干个桶(HashEntry),每个桶又有可能是一个HashEntry连接起来的链表，存储发生碰撞的元素。
每个ConcurrentHashMap在默认并发级下会创建包含16个Segment对象的数组，每个数组有若干个桶，当我们进行put方法时，通过hash方法对key进行计算，得到hash值，找到对应的segment，然后对该segment进行加锁，然后调用segment的put方法进行存储操作，此时其他线程就不能访问当前的segment，但可以访问其他的segment对象，不会发生阻塞等待。
**jdk 1.8版** 在jdk 8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

## 25.遍历一个List有哪些不同的方式?

```
List<String> strList = new ArrayList<>();
//for-each
for(String str:strList) {
    System.out.print(str);
}

//use iterator 尽量使用这种 更安全(fail-fast)
Iterator<String> it = strList.iterator();
while(it.hasNext) {
    System.out.printf(it.next());
}
```

## 26.fail-fast与fail-safe有什么区别?

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util包中的所有集合类都被设计为fail->fast的，而java.util.concurrent中的集合类都为fail-safe的。当检测到正在遍历的集合的结构被改变时，Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

## 27.Array和ArrayList有何区别？什么时候更适合用Array?

1. Array可以容纳基本类型和对象，而ArrayList只能容纳对象。
2. Array是指定大小的，而ArrayList大小是固定的

## 28.哪些集合类提供对元素的随机访问?

ArrayList、HashMap、TreeMap和HashTable类提供对元素的随机访问。

## 29.HashSet的底层实现是什么?

通过看源码知道HashSet的实现是依赖于HashMap的，HashSet的值都是存储在HashMap中的。在HashSet的构造法中会初始化一个HashMap对象，HashSet不允许值重复，因此，HashSet的值是作为HashMap的key存储在HashMap中的，当存储的值已经存在时返回false。

## 30.LinkedHashMap的实现原理?

LinkedHashMap也是基于HashMap实现的，不同的是它定义了一个Entry header，这个header不是放在Table里，它是额外独立出来的。LinkedHashMap通过继承hashMap中的Entry,并添加两个属性Entry before,after,和header结合起来组成一个双向链表，来实现按插入顺序或访问顺序排序。LinkedHashMap定义了排序模式accessOrder，该属性为boolean型变量，对于访问顺序，为true；对于插入顺序，则为false。一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。

## 31.LinkedList和ArrayList的区别是什么?

1. ArrayList是基于数组实现，LinkedList是基于链表实现
2. ArrayList在查找时速度快，LinkedList在插入与删除时更具优势

## 32.什么是线程？进程和线程的关系是什么?

线程可定义为进程内的一个执行单位，或者定义为进程内的一个可调度实体。 在具有多线程机制的操作系统中，处理机调度的基本单位不是进程而是线程。一个进程可以有多个线程，而且至少有一个可执行线程。
打个比喻:进程好比工厂(计算机)里的车间，一个工厂里有多个车间(进程)在运转，每个车间里有多个工人（线程）在协同工作，这些工人就可以理解为线程。
线程和进程的关系:

1. 线程是进程的一个组成部分.
2. 进程的多个线程都在进程地址空间活动.
3. 系统资源是分配给进程的，线程需要资源时，系统从进程的资源里分配给线程.
4. 处理机调度的基本单位是线程.

## 33.Thread 类中的start() 和 run() 方法有什么区别?

start()方法被用来启动新创建的线程，而且start()内部调用了run()方法，这和直接调用run()方法的效果不一样。当你调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动，start()方法才会启动新线程。

## 34.什么是线程安全?

当多个线程访问某个类时，不管运行时环境采用何种调度方式或者线程将如何交替执行，并且在主调代码中不需要任何额外的同步或协同，这个类都能表现出正确的行为。
线程安全的核心是“**正确性**”，也就是说当多个线程访问某个类时，能够得到预期的结果，那么就是线程安全的。

## 35.Java中有哪几种锁?

**自旋锁:** 自旋锁在JDK1.6之后就默认开启了。基于之前的观察，共享数据的锁定状态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行一个忙循环也就是自旋操作。

在jdk6之后，引入了自适应的自旋锁，也就是等待的时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定

**偏向锁:** 在JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原语。进一步提升程序的运行性能。 偏向锁就是偏心的偏，意思是这个锁会偏向第一个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考虑是否使用偏向锁。

**轻量级锁:** 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁。这种锁升级却不能降级的策略，目的是为了提高获得锁和释放锁的效率，下文会详细分析

## 36.synchronized内置锁

java中以synchronize的形式,为防止资源冲突提供了内置支持。当任务要执行被synchronize关键字保护的代码段时,它将检查锁是否可用,然后获取锁——执行代码——释放锁。

所有对象都自动含有单一的锁。当一个线程正在访问一个对象的synchronized方法,那么其他线程不能访问该对象的其他synchronized方法,但可以访问非synchronized方法。因为一个对象只有一把锁,当一个线程获取了该对象的锁之后,其他线程无法获取该对象的锁,所以无法访问该对象的其他synchronized方法。

synchronized代码块

```
synchronized(synObject) {

}
```

当在某个线程中执行这段代码块，该线程会获取对象synObject的锁，从而使得其他线程无法同时访问该代码块。synObject可以是this,代表获取当前对象的锁,也可以是类中的一个属性,代表获取该属性的锁。

针对每一个类,也有一个锁,所以static synchronize 方法可以在类的范围内防止对static数据的并发访问。如果一个线程执行一个对象的非static synchronized方法，另外一个线程需要执行这个对象所属类的static synchronized方法，此时不会发生互斥现象，因为访问static synchronized方法占用的是类锁，而访问非static synchronized方法占用的是对象锁，所以不存在互斥现象。

对于synchronized方法或者synchronized代码块，当出现异常时，JVM会自动释放当前线程占用的锁，因此不会由于异常导致出现死锁现象。

## 37.ThreadLocal理解

ThreadLocal是一个创建线程局部变量的类。通常情况下我们创建的变量,可以被多个线程访问并修改,通过ThreadLocal创建的变量只能被当前线程访问。

**ThreadLocal内部实现**

ThreadLocal提供了set和get方法.
set方法会先获取当前线程,然后用当前线程作为句柄,获取ThreadLocaMap对象,并判断该对象是否为空,如果为空则创建一个,并设置值,不为空则直接设置值。

```
public void set(T value) {
Thread t = Thread.currentThread();
ThreadLocalMap map = getMap(t);
if (map != null)
    map.set(this, value);
else
    createMap(t, value);
}
```

ThreadLocal的值是放入了当前线程的一个ThreadLocalMap实例中，所以只能在本线程中访问，其他线程无法访问。

ThreadLocal并不会导致内存泄露,因为ThreadLocalMap中的key存储的是ThreadLocal实例的弱引用,因此如果应用使用了线程池,即便之前的线程实例处理完之后出于复用的目的依然存活,也不会产生内存泄露。

## 38.为什么wait, notify 和 notifyAll这些方法不在thread类里面?

这是个设计相关的问题，它考察的是面试者对现有系统和一些普遍存在但看起来不合理的事物的看法。回答这些问题的时候，你要说明为什么把这些方法放在Object类里是有意义的，还有不把它放在Thread类里的原因。一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。