

JavaOOP面试题

1、什么是B/S架构？什么是C/S架构

1. B/S(Browser/Server)，浏览器/服务器程序
 2. C/S(Client/Server)，客户端/服务端，桌面应用程序
- 2.C/S(Client/Server)，客户端/服务端，桌面应用程序

2、Java都有那些开发平台？

1. JAVA SE：主要用在客户端开发
2. JAVA EE：主要用在web应用程序开发
3. JAVA ME：主要用在嵌入式应用程序开发

3、什么是JDK？什么是JRE?

1. JDK：java development kit：java开发工具包，是开发人员所需要安装的环境
2. JRE：java runtime environment：java运行环境，java程序运行所需要安装的环境

4、Java语言有哪些特点

1. 简单易学、有丰富的类库
2. 面向对象（Java最重要的特性，让程序耦合度更低，内聚性更高）
3. 与平台无关性（JVM是Java跨平台使用的根本）
4. 可靠安全
5. 支持多线程

5、面向对象和面向过程的区别

1. 面向过程：

一种较早的编程思想，顾名思义就是该思想是站着过程的角度思考问题，强调的就是功能行为，功能的执行过程，即先后顺序，而每一个功能我们都使用函数（类似于方法）把这些步骤一步一步实现。使用的时候依次调用函数就可以了。

2. 面向对象：

一种基于面向过程的新编程思想，顾名思义就是该思想是站在对象的角度思考问题，我们把多个功能合理放到不同对象里，强调的是具备某些功能的对象。

具备某种功能的实体，称为对象。面向对象最小的程序单元是：类。面向对象更加符合常规的思维方式，稳定性好，可重用性强，易于开发大型软件产品，有良好的可维护性。

在软件工程上，面向对象可以使工程更加模块化，实现更低的耦合和更高的内聚。

6、什么是数据结构？

计算机保存，组织数据的方式

7、Java的数据结构有那些？

- 1.线性表 (ArrayList)
- 2.链表 (LinkedList)
- 3.栈 (Stack)
- 4.队列 (Queue)
- 5.图 (Map)
- 6.树 (Tree)

8、什么是OOP?

面向对象编程

9、类与对象的关系?

类是对象的抽象，对象是类的具体，类是对象的模板，对象是类的实例

10、Java中有几种数据类型

整形 : byte,short,int,long
浮点型 : float,double
字符型 : char
布尔型 : boolean

11、标识符的命名规则。

1. 标识符的含义：
是指在程序中，我们自己定义的内容，譬如，类的名字，方法名称以及变量名称等等，都是标识符。
2. 命名规则：（硬性要求）
标识符可以包含英文字母，0-9的数字，\$以及_
标识符不能以数字开头
标识符不是关键字
3. 命名规范：（非硬性要求）
类名规范：首字母大写，后面每个单词首字母大写（大驼峰式）。
变量名规范：首字母小写，后面每个单词首字母大写（小驼峰式）。
方法名规范：同变量名。

12、instanceof关键字的作用

instanceof 严格来说是Java中的一个双目运算符，用来测试一个对象是否为一个类的实例，用法为：

```
boolean result = obj instanceof Class
```

其中 obj 为一个对象，Class 表示一个类或者一个接口，当 obj 为 Class 的对象，或者是其直接或间接子类，或者是其接口的实现类，结果result 都返回 true，否则返回false。

注意：编译器会检查 obj 是否能转换成右边的class类型，如果不能转换则直接报错，如果不能确定类型，则通过编译，具体看运行时定。

```
int i=0;  
  
System.out.println(i instanceof Integer); //编译不通过 i必须是引用类型，不能是基本类型  
  
System.out.println(i instanceof Object); //编译不通过  
  
Integer integer=newInteger(1);  
  
System.out.println(integer instanceof Integer); //true  
//false，在JavaSE规范中对instanceof运算符的规定就是：如果obj为null，那么将返回false。  
System.out.println(null instanceof Object);
```

13、什么是隐式转换，什么是显式转换

显示转换就是类型强转，把一个大类型的数据强制赋值给小类型的数据；隐式转换就是大范围的变量能够接受小范围的数据；隐式转换和显式转换其实就是自动类型转换和强制类型转换。

14、Char类型能不能转成int类型？能不能转化成string类型，能不能转成double类型

Char在java中也是比较特殊的类型，它的int值从1开始，一共有2的16次方个数据；
Char<int<long<float<double；Char类型可以隐式转成int,double类型，但是不能隐式转换成string；如果char类型转成byte，short类型的时候，需要强转。

15、什么是拆装箱？

1. 装箱就是自动将基本数据类型转换为包装器类型（int-->Integer）；调用方法：Integer的
valueOf(int)方法

拆箱就是自动将包装器类型转换为基本数据类型（Integer-->int）。调用方法：Integer的intValue方法
在Java SE5之前，如果要生成一个数值为10的Integer对象，必须这样进行：

```
Integer i = new Integer(10);
```

而在从Java SE5开始就提供了自动装箱的特性，如果要生成一个数值为10的Integer对象，只需要这样就可以了：

```
Integer i = 10;
```

面试题1：以下代码会输出什么？

```
public class Main {  
    public static void main(String[] args) {  
        Integer i1 = 100;  
        Integer i2 = 100;  
        Integer i3 = 200;  
        Integer i4 = 200;  
        System.out.println(i1==i2);  
        System.out.println(i3==i4);  
    }  
}
```

结果：

```
true  
false
```

16、Java中的包装类都是那些？

byte : Byte , short : Short , int : Integer , long : Long , float : Float , double : Double , char : Character , boolean : Boolean

17、一个java类中包含那些内容？

属性、方法、内部类、构造方法、代码块。

18、那针对浮点型数据运算出现的误差的问题，你怎么解决？

使用BigDecimal类进行浮点型数据的运算

19、面向对象的特征有哪些方面？

抽象：

抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。

继承：

继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段（如果

不能理解请阅读阎宏博士的《Java 与模式》或《设计模式精解》中关于桥梁模式的部分)。

封装：

通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问

只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自

治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写

一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，

只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，

明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是

封装得足够好的，因为几个按键就搞定了所有的事情）。

多态性：

多态性是指允许不同子类型的对象对同一消息作出不同的响应。

简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分

为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的

服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B

系统有多种提供服务的方式，但一切对 A 系统来说都是透明的（就像电动剃须

刀是 A 系统，它的供电系统是 B 系统，B 系统可以使用电池供电或者用交流电，

甚至还有可能是太阳能，A 系统只会通过 B 类对象调用供电的方法，但并不知道

供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载

（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）

实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的

东西，要实现多态需要做两件事：1). 方法重写（子类继承父类并重写父类中已

有的或抽象的方法）；2). 对象造型（用父类型引用引用子类型对象，这样同样

的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

20、访问修饰符 public,private,protected,以及不写（默认）时的区别？

修饰符	当前类	同包	子类	其他包
public	能	能	能	能
protected	能	能	能	不能
default	能	能	不能	不能
private	能	不能	不能	不能

类的成员不写访问修饰时默认为 default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java 中，外部类的修饰符只能是 public 或默认，类的成员（包括内部类）的修饰符可以是以上四种。

21、String 是最基本的数据类型吗？

不是。Java 中的基本数据类型只有 8 个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type），剩下的都是引用类型（reference type），Java 5 以后引入的枚举类型也算是一种比较特殊的引用类型。

22、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;。

23、short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1; 有错吗？

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。

24、重载和重写的区别

重写**（Override）**

从字面上看，重写就是重新写一遍的意思。其实就是在子类中把父类本身有的方法重新写一遍。子类继承了父类原有的方法，但有时子类并不想原封不动的继承父类中的某个方法，所以在方法名，参数列表，返回类型（除过子

类中方法的返回值是父类中方法返回值的子类时)都相同的情况下，对方法体进行修改或重写，这就是重写。但要注意子类函数的访问修饰权限不能少于父类的。

```
public class Father {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Son s = new Son();  
        s.sayHello();  
    }  
  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
}  
  
class Son extends Father{  
    @Override  
    public void sayHello() {  
        // TODO Auto-generated method stub  
        System.out.println("hello by ");  
    }  
}
```

原因：在某个范围内的整型数值的个数是有限的，而浮点数却不是。

重写 总结：

- 1.发生在父类与子类之间
- 2.方法名，参数列表，返回类型（除过子类中方法的返回类型是父类中返回类型的子类）必须相同
- 3.访问修饰符的限制一定要大于被重写方法的访问修饰符（public>protected>default>private）
- 4.重写方法一定不能抛出新的检查异常或者比被重写方法声明更加宽泛的检查型异常

重载 (Overload)

在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同甚至是参数顺序不同）

则视为重载。同时，重载对返回类型没有要求，可以相同也可以不同，但**不能通过返回类型是否相同来判断重载**。

```
public class Father {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        Father s = new Father();  
        s.sayHello();  
        s.sayHello("wintershii");  
    }  
  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
  
    public void sayHello(String name) {  
        System.out.println("Hello" + " " + name);  
    }  
}
```

重载总结：

- 1.重载Overload是一个类中多态性的一种表现
- 2.重载要求同名方法的参数列表不同(参数类型，参数个数甚至是参数顺序)
- 3.重载的时候，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准

25、equals与==的区别

== :

== 比较的是变量(栈)内存中存放的对象的(堆)内存地址，用来判断两个对象的地址是否相同，即是否是指同一个对象。比较的是真正意义上的指针操作。

- 1、比较的是操作符两端的操作数是否是同一个对象。
- 2、两边的操作数必须是同一类型的（可以是父子类之间）才能编译通过。
- 3、比较的是地址，如果是具体的阿拉伯数字的比较，值相等则为true，如：

int a=10 与 long b=10L 与 double c=10.0都是相同的（为true），因为他们都指向地址为10的堆。

equals :

equals用来比较的是两个对象的内容是否相等，由于所有的类都是继承自java.lang.Object类的，所以适用于所有对象，如果没有对该方法进行覆盖的话，调用的仍然是Object类中的方法，而Object中的**equals**方法返回的却是**==**的判断。

总结：

所有比较是否相等时，都是用**equals** 并且在对常量相比较时，把常量写在前面，因为使用**object.equals** **object**可能为null 则空指针

在阿里的代码规范中只使用**equals**，阿里插件默认会识别，并可以快速修改，推荐安装阿里插件来排查老代码使用“**==**”，替换成**equals**

36、++i与i++的区别

i++：先赋值，后计算

++i：先计算，后赋值

37、程序的结构有那些？

顺序结构
选择结构
循环结构

38、数组实例化有几种方式？

静态实例化：创建数组的时候已经指定数组中的元素。

```
int [] a= new int[]{ 1 , 3 , 3}
```

动态实例化：实例化数组的时候，只指定了数组程度，数组中所有元素都是数组类型的默认值

39、Java中各种数据默认值

Byte,short,int,long默认是都是0

Boolean默认值是false

Char类型的默认值是“

Float与double类型的默认是0.0

对象类型的默认值是null

40、Java常用包有那些？

```
Java.lang  
Java.io  
Java.sql  
Java.util  
Java.awt  
Java.net  
Java.math
```

41、Object类常用方法有那些？

```
Equals  
HashCode  
ToString  
Wait  
Notify  
Clone  
GetClass
```

42、java中有没有指针？

有指针，但是隐藏了，开发人员无法直接操作指针，由jvm来操作指针

43、java中是值传递引用传递？

理论上说，java都是引用传递，对于基本数据类型，传递是值的副本，而不是值本身。对于对象类型，传递是对象的引用，当在一个方法操作操作参数的时候，其实操作的是引用所指向的对象。

44、实例化数组后，能不能改变数组长度呢？

不能，数组一旦实例化，它的长度就是固定的

45、假设数组内有5个元素，如果对数组进行反序，该如何做？

创建一个新数组，从后到前循环遍历每个元素，将取出的元素依次顺序放入新数组中

46、形参与实参区别

实参(argument)：

全称为“实际参数”是在调用时传递给函数的参数。实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形参。因此应预先用赋值，输入等办法使实参获得确定值。

形参(parameter)：

全称为“形式参数”由于它不是实际存在变量，所以又称虚拟变量。是在定义函数名和函数体的时候使用的参数，目的是用来接收调用该函数时传入的参数。在调用函数时，实参将赋值给形参。因而，必须注意实参的个数，类型应与形参一一对应，并且实参必须要有确定的值。

形参出现在函数定义中，在整个函数体内都可以使用，离开该函数则不能使用。

实参出现在主调函数中，进入被调函数后，实参变量也不能使用。

形参和实参的功能是作数据传送。发生函数调用时，主调函数把实参的值传送给被调函数的形参从而实现主调函数向被调函数的数据传送。

1.形参变量只有在被调用时才分配内存单元，在调用结束时，即刻释放所分配的内存单元。因此，形参只有在函数内部有效。函数调用结束后返回

主调函数后则不能再使用该形参变量。

2.实参可以是常量、变量、表达式、函数等，无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值，以便把这些值传送给形

参。因此应预先用赋值，输入等办法使实参获得确定值。

3.实参和形参在数量上，类型上，顺序上应严格一致，否则会发生“类型不匹配”的错误。

4.函数调用中发生的数据传送是单向的。即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。因此在函数调用过程中，形参的值

发生改变，而实参中的值不会变化。

5.当形参和实参不是指针类型时，在该函数运行时，形参和实参是不同的变量，他们在内存中位于不同的位置，形参将实参的内容复制一份，在该

函数运行结束的时候形参被释放，而实参内容不会改变。

而如果函数的参数是指针类型变量，在调用该函数的过程中，传给函数的是实参的地址，在函数体内部使用的也是实参的地址，即使用的就是实参

本身。所以在函数体内部可以改变实参的值。

47、构造方法能不能显式调用？

不能，构造方法当成普通方法调用，只有在创建对象的时候它才会被系统调用

48、什么是方法重载？

方法的重载就是在同一个类中允许同时存在一个以上的同名方法，只要它们的参数个数或者类型不同即可。在这种情况下，该方法就叫被重载了，这个过程称为方法的重载（override）

49、构造方法能不能重写？能不能重载？

可以重载，但不能重写。

50、内部类与静态内部类的区别？

静态内部类相对与外部类是独立存在的，在静态内部类中无法直接访问外部类中变量、方法。如果要访问的话，必须要new一个外部类的对象，使用new出来的对象来访问。但是可以直接访问静态的变量、调用静态的方法；

普通内部类作为外部类一个成员而存在，在普通内部类中可以直接访问外部类属性，调用外部类的方法。

如果外部类要访问内部类的属性或者调用内部类的方法，必须要创建一个内部类的对象，使用该对象访问属性或者调用方法。

如果其他的类要访问普通内部类的属性或者调用普通内部类的方法，必须要在外部类中创建一个普通内部类的对象作为一个属性，外同类可以通过该属性调用普通内部类的方法或者访问普通内部类的属性

如果其他的类要访问静态内部类的属性或者调用静态内部类的方法，直接创建一个静态内部类对象即可。

51、Static关键字有什么作用？

Static可以修饰内部类、方法、变量、代码块

Static修饰的类是静态内部类

Static修饰的方法是静态方法，表示该方法属于当前类的，而不属于某个对象的，静态方法也不能被重写，可以直接使用类名来调用。在static方法中不能使用this或者super关键字。

Static修饰变量是静态变量或者叫类变量，静态变量被所有实例所共享，不会依赖于对象。静态变量在内存中只有一份拷贝，在JVM加载类的时候，只为静态分配一次内存。

Static修饰的代码块叫静态代码块，通常用来做程序优化的。静态代码块中的代码在整个类加载的时候只会执行一次。静态代码块可以有多个，如果有多个，按照先后顺序依次执行。

52、final在java中的作用，有哪些用法？

final也是很多面试喜欢问的地方，但我觉得这个问题很无聊，通常能回答下以下5点就不错了：

1. 被final修饰的类不可以被继承
2. 被final修饰的方法不可以被重写
3. 被final修饰的变量不可以被改变，如果修饰引用，那么表示引用不可变，引用指向的内容可变。
4. 被final修饰的方法，JVM会尝试将其内联，以提高运行效率
5. 被final修饰的常量，在编译阶段会存入常量池中。

除此之外，编译器对final域要遵守的两个重排序规则更好：

在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序

初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序

53、String String StringBuffer 和 StringBuilder 的区别是什么？

String是只读字符串，它并不是基本数据类型，而是一个对象。从底层源码来看是一个final类型的字符串数组，所引用的字符串不能被改变，一经定义，无法再增删改。每次对String的操作都会生成新的String对象

```
private final char value[];
```

每次+操作：隐式在堆上new了一个跟原字符串相同的StringBuilder对象，再调用append方法拼接+后面的字符。

StringBuffer与StringBuilder都继承了AbstractStringBulder类，而AbstractStringBulder又实现了CharSequence接口，两个类都是用来进行字符串操作的。

在做字符串拼接修改删除替换时，效率比string更高。

StringBuffer是线程安全的，StringBuilder是非线程安全的。所以StringBuilder比StringBuffer效率更高，StringBuffer的方法大多都加了synchronized关键字

54、String str="aaa",与String str=new String("aaa")一样吗？

一共有两个引用，三个对象。因为"aa"与"bb"都是常量，常量的值不能改变，当执行字符串拼接时候，会创建一个新的常量是" aabbb",将其存到常量池中。

55、讲下java中的math类有那些常用方法？

Pow() : 幂运算

Sqrt() : 平方根

Round() : 四舍五入

Abs() : 求绝对值

Random() : 生成一个0-1的随机数，包括0不包括1

56、String类的常用方法有那些？

charAt : 返回指定索引处的字符

indexOf() : 返回指定字符的索引

replace() : 字符串替换

trim() : 去除字符串两端空白

split() : 分割字符串，返回一个分割后的字符串数组

getBytes() : 返回字符串的byte类型数组

length() : 返回字符串长度

toLowerCase() : 将字符串转成小写字母

toUpperCase() : 将字符串转成大写字符

substring() : 截取字符串

format() : 格式化字符串

equals() : 字符串比较

57、Java中的继承是单继承还是多继承

Java中既有单继承，又有多继承。对于java类来说只能有一个父类，对于接口来说可以同时继承多个接口

58、Super与this表示什么？

Super表示当前类的父类对象

This表示当前类的对象

59、普通类与抽象类有什么区别？

普通类不能包含抽象方法，抽象类可以包含抽象方法

抽象类不能直接实例化，普通类可以直接实例化

60、什么是接口？为什么需要接口？

接口就是某个事物对外提供的一些功能的声明，是一种特殊的java类，接口弥补了java单继承的缺点

61、接口有什么特点？

接口中声明全是public static final修饰的常量

接口中所有方法都是抽象方法

接口是没有构造方法的

接口也不能直接实例化

接口可以多继承

62、抽象类和接口的区别？

抽象类：

1. 抽象方法，只有行为的概念，没有具体的行为实现。使用abstract关键字修饰，没有方法体。子类必须重写这些抽象方法。

2. 包含抽象方法的类，一定是抽象类。

3. 抽象类只能被继承，一个类只能继承一个抽象类。

接口：

1. 全部的方法都是抽象方法，属性都是常量
2. 不能实例化，可以定义变量。
3. 接口变量可以引用具体实现类的实例
4. 接口只能被实现，一个具体类实现接口，必须实现全部的抽象方法
5. 接口之间可以多实现
6. 一个具体类可以实现多个接口，实现多继承现象

63、Hashcode的作用

java的集合有两类，一类是List，还有一类是Set。前者有序可重复，后者无序不重复。当我们在set中插入的时候怎么判断是否已经存在该元素呢，可以通过equals方法。但是如果元素太多，用这样的方法就会比较慢。

于是有人发明了哈希算法来提高集合中查找元素的效率。这种方式将集合分成若干个存储区域，每个对象可以计算出一个哈希码，可以将哈希码分组，每组分别对应某个存储区域，根据一个对象的哈希码就可以确定该对象应该存储的那个区域。

hashCode方法可以这样理解：它返回的就是根据对象的内存地址换算出的一个值。这样一来，当集合要添加新的元素时，先调用这个元素的hashCode方法，就一下子能定位到它应该放置的物理位置上。如果这个位置上没有元素，它就可以直接存储在这个位置上，不用再进行任何比较了；如果这个位置上已经有元素了，就调用它的equals方法与新元素进行比较，相同的话就不存了，不相同就散列其它的地址。这样一来实际调用equals方法的次数就大大降低了，几乎只需要一两次。

64、Java的四种引用，强弱软虚

强引用

强引用是平常中使用最多的引用，强引用在程序内存不足（OOM）的时候也不会被回收，使用方式：

```
String str = new String("str");
```

软引用

软引用在程序内存不足时，会被回收，使用方式：

```
// 注意：wrf这个引用也是强引用，它是指向SoftReference这个对象的。  
// 这里的软引用指的是指向new String("str")的引用，也就是SoftReference类中T  
SoftReference<String> wrf = new SoftReference<String>(new String("str"));
```

可用场景：创建缓存的时候，创建的对象放进缓存中，当内存不足时，JVM就会回收早先创建的对象。

弱引用

弱引用就是只要JVM垃圾回收器发现了它，就会将之回收，使用方式：

```
WeakReference<String> wrf=newWeakReference<String>(str);
```

可用场景：Java源码中的java.util.WeakHashMap中的key就是使用弱引用，我的理解就是，

一旦我不需要某个引用，JVM会自动帮我处理它，这样我就不需要做其它操作。

虚引用

虚引用的回收机制跟弱引用差不多，但是它被回收之前，会被放入ReferenceQueue中。注意哦，其它引用是被JVM回收后才被传入ReferenceQueue中的。由于这个机制，所以虚引用大多被用于引用销毁前的处理工作。还有就是，虚引用创建的时候，必须带有ReferenceQueue，使用

例子：

```
PhantomReference<String> prf=newPhantomReference<String>(new  
String("str"),newReferenceQueue<()>());
```

可用场景：对象销毁前的一些操作，比如说资源释放等。** Object.finalize() 虽然也可以做这类动作，但是这种方式即不安全又低效

上诉所说的几类引用，都是指对象本身的引用，而不是指 Reference 的四个子类的引用

(SoftReference 等)。

65、Java创建对象有几种方式？

java中提供了以下四种创建对象的方式：

1. new创建新对象
2. 通过反射机制
3. 采用clone机制
4. 通过序列化机制

66、有没有可能两个不相等的对象有相同的hashcode

有可能,在产生hash冲突时,两个不相等的对象就会有相同的 hashcode 值.当hash冲突产生时,一般有以下几种方式来处理:

1. **拉链法**:每个哈希表节点都有一个next指针,多个哈希表节点可以用next指针构成一个单向链表,被分配到同一个索引上的多个节点可以用这个单向链表进行存储.
2. **开放定址法**:一旦发生了冲突,就去寻找下一个空的散列地址,只要散列表足够大,空的散列地址总能找到,并将记录存入.
3. **再哈希**:又叫双哈希法,有多个不同的Hash函数.当发生冲突时,使用第二个,第三个....等哈希函数计算地址,直到无冲突.

67、拷贝和浅拷贝的区别是什么?

浅拷贝:

被复制对象的所有变量都含有与原来的对象相同的值,而所有的对其他对象的引用仍然指向原来对象.换言之,浅拷贝仅仅复制所考虑的对象,而不复制它所引用的对象.

深拷贝:

被复制对象的所有变量都含有与原来的对象相同的值,而那些引用其他对象的变量将指向被复制过的新对象,而不再是原有的那些被引用的对象.换言之,深拷贝把要复制的对象所引用的对象都复制了一遍.

68、static都有哪些用法?

所有的人都知道static关键字这两个基本的用法:静态变量和静态方法.也就是被static所修饰的变量/方法都属于类的静态资源,类实例所共享.

除了静态变量和静态方法之外,static也用于静态块,多用于初始化操作:

```
public class PreCache{  
    static{  
        //执行相关操作  
    }  
}
```

此外static也多用于修饰内部类,此时称之为静态内部类.

最后一种用法就是静态导包,即 import static .import static是在JDK 1.5之后引入的新特性,可以用来指定导入某个类中的静态资源,并且不需要使用类名,可以直接使用资源名,比如:

```
import static java.lang.Math.*;  
  
public class Test{  
    public static void main(String[] args){  
        //System.out.println(Math.sin(20));传统做法  
        System.out.println(sin(20));  
    }  
}
```

69、a=a+b与a+=b有什么区别吗?

+= 操作符会进行隐式自动类型转换,此处a+=b隐式的将加操作的结果类型强制转换为持有结果的类型,而a=a+b则不会自动进行类型转换.如:

```
byte a = 127;  
  
byte b = 127;  
  
b = a + b; // 报编译错误:cannot convert from int to byte  
  
b += a;
```

以下代码是否有错,有的话怎么改?

```
short s1= 1;  
  
s1 = s1 + 1;
```

有错误.short类型在进行运算时会自动提升为int类型,也就是说 s1+1 的运算结果是int类型,而s1是short类型,此时编译器会报错.

正确写法 :

```
short s1= 1;  
s1 += 1;
```

+=操作符会对右边的表达式结果强转匹配左边的数据类型,所以没错.

70、final、finalize()、finally

性质不同

1. final为关键字 ;
2. finalize()为方法 ;
3. finally为区块标志 , 用于try语句中 ;

作用

1. final为用于标识常量的关键字 , final标识的关键字存储在常量池中 (在这里final常量的具体用法将在下面进行介绍) ;
2. finalize()方法在Object中进行了定义 , 用于在对象“消失”时 , 由JVM进行调用用于对对象进行垃圾回收 , 类似于C++中的析构函数 ; 用户自定义时 , 用于释放对象占用的资源 (比如进行I/O操作) ;
3. finally{}用于标识代码块 , 与try{}进行配合 , 不论try中的代码执行完或没有执行完 (这里指有异常) , 该代码块之中的程序必定会进行 ;

71、JDBC操作的步骤

加载数据库驱动类
打开数据库连接
执行sql语句
处理返回结果
关闭资源

72、在使用jdbc的时候 , 如何防止出现sql注入的问题。

使用PreparedStatement类 , 而不是使用Statement类

73、怎么在JDBC内调用一个存储过程

使用CallableStatement

74、是否了解连接池 , 使用连接池有什么好处 ?

数据库连接是非常消耗资源的 , 影响到程序的性能指标。连接池是用来分配、管理、释放数据库连接的 , 可以使应用程序重复使用同一个数据库连接 , 而不是每次都创建一个新的数据库连接。通过释放空闲时间较长的数据库连接避免数据库因为创建太多的连接而造成的连接遗漏问题 , 提高了程序性能。

75、你所了解的数据源技术有那些 ? 使用数据源有什么好处 ?

Dbcp,c3p0等 , 用的最多还是c3p0 , 因为c3p0比dbcp更加稳定 , 安全 ; 通过配置文件的形式来维护数据库信息 , 而不是通过硬编码。当连接的数据库信息发生改变时 , 不需要再更改程序代码就实现了数据库信息的更新。

76、&和&&的区别

&是位运算符。&&是布尔逻辑运算符 , 在进行逻辑判断时用&处理的前面为false后面的内容仍需处理 , 用&&处理的前面为false不再处理后面的内容。

77、静态内部类如何定义

定义在类内部的静态类 , 就是静态内部类。

```
public class Out {  
  
    private static int a;  
  
    private int b;
```

```

public static class Inner {
    public void print() {
        System.out.println(a);
    }
}

```

1. 静态内部类可以访问外部类所有的静态变量和方法，即使是 private 的也一样。
2. 静态内部类和一般类一致，可以定义静态变量、方法，构造方法等。
3. 其它类使用静态内部类需要使用“外部类.静态内部类”方式，如下所示：Out.Inner inner = new Out.Inner();inner.print();
4. Java集合类HashMap内部就有一个静态内部类Entry。Entry是HashMap存放元素的抽象，HashMap 内部维护 Entry 数组用了存放元素，但是 Entry 对使用者是透明的。像这种和外部类关系密切的，且不依赖外部类实例的，都可以使用静态内部类。

78、什么是成员内部类

定义在类内部的非静态类，就是成员内部类。成员内部类不能定义静态方法和变量（final修饰的除外）。这是因为成员内部类是非静态的，类初始化的时候先初始化静态成员，如果允许成员内部类定义静态变量，那么成员内部类的静态变量初始化顺序是有歧义的。实例：

```

public class Out {
    private static int a;
    private int b;
    public class Inner {
        public void print() {
            System.out.println(a);
            System.out.println(b);
        }
    }
}

```

79、Static Nested Class 和 Inner Class的不同

Nested Class（一般是C++的说法），Inner Class（一般是JAVA的说法）。Java内部类与C++嵌套类最大的不同就在于是否有指向外部的引用上。注：静态内部类（Inner Class）意味着1创建一个static内部类的对象，不需要一个外部类对象，2不能从一个static内部类的一个对象访问一个外部类对象

80、什么时候用assert

assertion(断言)在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。在实现中，assertion就是在程序中的一条语句，它对一个boolean表达式进行检查，一个正确程序必须保证这个boolean表达式的值为true；如果该值为false，说明程序已经处于不正确的状态下，系统将给出警告或退出。一般来说，assertion用于保证程序最基本、关键的正确性。assertion检查通常在开发和测试时开启。为了提高性能，在软件发布后，assertion检查通常是关闭的

81、Java有没有goto

java中的保留字，现在没有在java中使用

82、数组有没有length()这个方法? String有没有length()这个方法

数组没有length()这个方法，有length的属性。String有length()这个方法

83、用最有效率的方法算出2乘以8等于几

2 << 3

84、float型float f=3.4是否正确？

不正确。精度不准确,应该用强制类型转换,如下所示:float f=(float)3.4

85、排序都有哪几种方法?请列举

排序的方法有:插入排序(直接插入排序、希尔排序),交换排序(冒泡排序、快速排序),选择排序(直接选择排序、堆排序),归并排序,分配排序(箱排序、基数排序)快速排序的伪代码。
//使用快速排序方法对a[0:n-1]排序从a[0:n-1]中选择一个元素作为middle,该元素为支点把余下的元素分割为两段left和right,使得left中的元素都小于等于支点,而right中的元素都大于等于支点递归地使用快速排序方法对left进行排序递归地使用快速排序方法对right进行排序所得结果为left+middle+right

86、静态变量和实例变量的区别?

```
static i = 10; //常量 class A a; a.i =10;//可变
```

87、说出一些常用的类,包,接口,请各举5个

常用的类:BufferedReader BufferedWriter FileReader FileWirter String Integer
常用的包:java.lang java.awt java.io java.util java.sql
常用的接口:Remote List Map Document NodeList

88、a.hashCode()有什么用?与a.equals(b)有什么关系?

hashCode()方法是相应用对象整型的hash值。它常用于基于hash的集合类,如Hashtable、HashMap、LinkedHashMap等等。它与equals()方法关系特别紧密。根据Java规范,两个使用equal()方法来判断相等的对象,必须具有相同的hash code。

89、Java中的编译期常量是什么?使用它又什么风险?

公共静态不可变(public static final)变量也就是我们所说的编译期常量,这里的public可选的。实际上这些变量在编译时会被替换掉,因为编译器知道这些变量的值,并且知道这些变量在运行时不能改变。这种方式存在的一个问题是你使用了一个内部的或第三方库中的公有编译时常量,但是这个值后面被其他人改变了,但是你的客户端仍然在使用老的值,甚至你已经部署了一个新的jar。为了避免这种情况,当你在更新依赖JAR文件时,确保重新编译你的程序。

90、在Java中,如何跳出当前的多重嵌套循环?

在最外层循环前加一个标记如A,然后用break A;可以跳出多重循环。(Java中支持带标签的break和continue语句,作用有点类似于C和C++中的goto语句,但是就像要避免使用goto一样,应该避免使用带标签的break和continue,因为它不会让你的程序变得更优雅,很多时候甚至有相反的作用,所以这种语法其实不知道更好)

91、构造器(constructor)是否可被重写(override)?

构造器不能被继承,因此不能被重写,但可以被重载。

92、两个对象值相同(x.equals(y) == true),但却可有不同的hash code,这句话对不对?

不对,如果两个对象x和y满足x.equals(y) == true,它们的哈希码(hash code)应当相同。Java对于equals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。当然,你未必要按照要求去做,但是如果你违背了上述原则就会发现在使用容器时,相同的对象可以出现在Set集合中,同时增加新元素的效率会大大下降(对于使用哈希存储的系统,如果哈希码频繁的冲突将会造成存取性能急剧下降)。

93、是否可以继承String类?

String类是final类,不可以被继承,继承String本身就是一个错误的行为,对String类型最好的重用方式是关联关系(Has-A)和依赖关系(Use-A)而不是继承关系(Is-A)。

94、当一个对象被当作参数传递到一个方法后,此方法可改变这个对象的属性,并可返回变化后的结果,那么这里到底是值传递还是引用传递?

是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时,参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变,但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码,但是在Java中却做不到。

```
using System;
namespace CS01 {
    class Program {
        public static void swap(ref int x, ref int y) {
```

```

        int temp = x;
        x = y;
        y = temp;
    } p
    public static void Main (string[] args) {
        int a = 5, b = 10;
        swap (ref a, ref b);
// a = 10, b = 5;
        Console.WriteLine ("a = {0}, b = {1}", a, b);
    }
}
}

```

说明：Java 中没有传引用实在是非常的不方便，这一点在 Java 8 中仍然没有得到改进，正是如此在 Java 编写的代码中才会出现大量的 Wrapper 类（将需要通过方法调用修改的引用置于一个 Wrapper 类中，再将 Wrapper 对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从 C 和 C++ 转型为 Java 程序员的开发者无法容忍。

95、String 和 StringBuilde、StringBuffer 的区别？

Java 平台提供了两种类型的字符串：String 和 StringBuffer/StringBuilder，它们可以储存和操作字符串。其中 String 是只读字符串，也就意味着 String 引用的字符串内容是不能被改变的。而 StringBuffer/StringBuilder 类表示的字符串对象可以直接进行修改。StringBuilder 是 Java 5 中引入的，它和 StringBuffer 的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被 synchronized 修饰，因此它的效率也比 StringBuffer 要高。

96、重载 (Overload) 和重写 (Override) 的区别。重载的方法能否根据返回类型进行区分？

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

97、char 型变量中能不能存储一个中文汉字，为什么？

char 类型可以存储一个中文汉字，因为 Java 中使用的编码是 Unicode（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个 char 类型占 2 个字节（16 比特），所以放一个中文是没问题的。

补充：使用 Unicode 意味着字符在 JVM 内部和外部有不同的表现形式，在 JVM 内部都是 Unicode，当这个字符被从 JVM 内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以 Java 中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如 InputStreamReader 和 OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于 C 程序员来说，要完成这样的编码转换恐怕要依赖于 union（联合体/共用体）共享内存的特征来实现了。

98、抽象类 (abstract class) 和接口 (interface) 有什么异同？

抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。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进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为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且其中的方法全部都是抽象方法。抽象类中的成员可以是 private、默认、protected、public 的，而接口中的成员全都是 public 的。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。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

99、静态嵌套类(Static Nested Class)和内部类 (Inner Class) 的不同？

Static Nested Class 是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示

```

/**
 * 扑克类（一副扑克）
 * @author 骆昊
 */
public class Poker {
    private static String[] suites = {"黑桃", "红桃", "草花", "方块"};
    private static int[] faces = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
    private Card[] cards;
    /**
     * 构造器
     */
    public Poker() {
        cards = new Card[52];
        for(int i = 0; i < suites.length; i++) {
            for(int j = 0; j < faces.length; j++) {
                cards[i * 13 + j] = new Card(suites[i], faces[j]);
            }
        }
    }
}

```

```

}
/**
 * 洗牌 (随机乱序)
 */
public void shuffle() {
    for(int i = 0, len = cards.length; i < len; i++) {
        int index = (int) (Math.random() * len);
        Card temp = cards[index];
        cards[index] = cards[i];
        cards[i] = temp;
    }
}
/**
 * 发牌
 * @param index 发牌的位置
 */
public Card deal(int index) {
    return cards[index];
}
/**
 * 卡片类 (一张扑克)
 * [内部类]
 * @author 骆昊
 */
public class Card {
    private String suite; // 花色
    private int face; // 点数
    public Card(String suite, int face) {
        this.suite = suite;
        this.face = face;
    }
    @Override
    public String toString() {
        String faceStr = "";
        switch(face) {
            case 1: faceStr = "A"; break;
            case 11: faceStr = "J"; break;
            case 12: faceStr = "Q"; break;
            case 13: faceStr = "K"; break;
            default: faceStr = String.valueOf(face);
        } return suite + faceStr;
    }
}
}

//测试类
class PokerTest {
    public static void main(String[] args) {
        Poker poker = new Poker();
        poker.shuffle(); // 洗牌
        Poker.Card c1 = poker.deal(0); // 发第一张牌
        // 对于非静态内部类 Card
        // 只有通过其外部类 Poker 对象才能创建 Card 对象
        Poker.Card c2 = poker.new Card("红心", 1); // 自己创建一张牌
        System.out.println(c1); // 洗牌后的第一张
        System.out.println(c2); // 打印: 红心 A
    }
}

```

100、Java 中会存在内存泄漏吗，请简单描述。

理论上 Java 因为有垃圾回收机制 (GC) 不会存在内存泄露问题 (这也是 Java 被广泛使用于服务器端编程的一个重要原因) ; 然而在实际开发中 , 可能会存在无用但可达的对象 , 这些对象不能被 GC 回收 , 因此也会导致内存泄露的发生。

例如 Hibernate 的 Session (一级缓存) 中的对象属于持久态 , 垃圾回收器是不会回收这些对象的 , 然而这些对象中可能存在无用的垃圾对象 , 如果不及时关闭 (close) 或清空 (flush) 一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露

```

import java.util.Arrays;
import java.util.EmptyStackException;
public class MyStack<T> {
    private T[] elements;
    private int size = 0;
    private static final int INIT_CAPACITY = 16;
    public MyStack() {
        elements = (T[]) new Object[INIT_CAPACITY];
    }
    public void push(T elem) {
        ensureCapacity();
        elements[size++] = elem;
    }
    public T pop() {

```

```

        if(size == 0)
            throw new EmptyStackException();
        return elements[--size];
    }
    private void ensureCapacity() {
        if(elements.length == size) {
            elements = Arrays.copyOf(elements, 2 * size + 1);
        }
    }
}

```

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的 pop 方法却存在内存泄露的问题，当我们用 pop 方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发 Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成 OutOfMemoryError。

101、抽象的（abstract）方法是否可同时是静态的（static），是否可同时是本地方法（native），是否可同时被 synchronized 修饰？

都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如 C 代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized 和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

102、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？

不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

103、如何实现对象克隆？

有两种方式：

- 1). 实现 Cloneable 接口并重写 Object 类中的 clone()方法；
- 2). 实现 Serializable 接口，通过对对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

```

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
public class MyUtil {
    private MyUtil() {
        throw new AssertionError();
    }
    @SuppressWarnings("unchecked")
    public static <T extends Serializable> T clone(T obj) throws
Exception {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bout);
        oos.writeObject(obj);
        ByteArrayInputStream bin = new
ByteArrayInputStream(bout.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bin);
        return (T) ois.readObject();
    }
}

```

测试代码：

```

import java.io.Serializable;
class Person implements Serializable {
    private static final long serialVersionUID = -9102017020286042305L;
    private String name; // 姓名
    private int age; // 年龄
    private Car car; // 座驾
    public Person(String name, int age, Car car) {
        this.name = name;
        this.age = age;
        this.car = car;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

```

```

    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public Car getCar() {
        return car;
    }
    public void setCar(Car car) {
        this.car = car;
    }
    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", car=" +
               car + "]";
    }
}

```

```

class Car implements Serializable {
    private static final long serialVersionUID = -5713945027627603702L;
    private String brand; // 品牌
    private int maxSpeed; // 最高时速
    public Car(String brand, int maxSpeed) {
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }
    public String getBrand() {
        return brand;
    }
    public void setBrand(String brand) {
        this.brand = brand;
    }
    public int getMaxSpeed() {
        return maxSpeed;
    }
    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }
    @Override
    public String toString() {
        return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed +
               "]";
    }
}
class CloneTest {
    public static void main(String[] args) {
        try {
            Person p1 = new Person("Hao LUO", 33, new Car("Benz",
                300));
            Person p2 = MyUtil.clone(p1); // 深度克隆
            p2.getCar().setBrand("BYD");
            // 修改克隆的 Person 对象 p2 关联的汽车对象的品牌属性
            // 原来的 Person 对象 p1 关联的汽车不会受到任何影响
            // 因为在克隆 Person 对象时其关联的汽车对象也被克隆了
            System.out.println(p1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限制，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种方案明显优于使用 Object 类的 clone 方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

104、接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concreteclass）？

接口可以继承接口，而且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类。

105、一个".java"源文件中是否可以包含多个类（不是内部类）？有什么限制？

可以，但一个源文件中最多只能有一个公开类（public class）而且文件名必须和公开类的类名完全保持一致。

106、Anonymous Inner Class(匿名内部类)是否可以继承其它类？是否可以实现接口？

可以继承其他类或实现其他接口，在 Swing 编程和 Android 开发中常用此方式来实现事件监听和回调。

107、内部类可以引用它的包含类（外部类）的成员吗？有没有什么限制？

一个内部类对象可以访问创建它的外部类对象的成员，包括私有成员。

108、Java 中的 final 关键字有哪些用法？

- (1)修饰类：表示该类不能被继承；
- (2)修饰方法：表示方法不能被重写；
- (3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

Java集合/泛型面试题

1、ArrayList和LinkedList的区别

ArrayList(数组)是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。

ArrayList获取数据的时间复杂度是O(1),但是要删除数据却是开销很大，因为这需要重排数组中的所有数据，
(因为删除数据以后，需要把后面所有的数据前移)

缺点：数组初始化必须指定初始化的长度，否则报错

例如：

```
int[] a = new int[4];//推荐使用int[] 这种方式初始化  
int c[] = {23,43,56,78};//长度: 4, 索引范围: [0,3]
```

List是一个有序的集合，可以包含重复的元素，提供了按索引访问的方式，它继承Collection。

List有两个重要的实现类：ArrayList和LinkedList

ArrayList：可以看作是能够自动增长容量的数组

ArrayList的toArray方法返回一个数组

ArrayList的asList方法返回一个列表

ArrayList底层的实现是Array, 数组扩容实现

LinkedList是一个双链表，在添加和删除元素时具有比ArrayList更好的性能。但在get与set方面弱于

ArrayList.当然，这些对比都是指数据量很大或者操作很频繁。

2、HashMap和HashTable的区别

1、两者父类不同

HashMap是继承自AbstractMap类，而Hashtable是继承自Dictionary类。不过它们都实现了同时实现了map、Cloneable（可复制）、Serializable（可序列化）这三个接口。

2、对外提供的接口不同

Hashtable比HashMap多提供了elements() 和contains() 两个方法。

elements() 方法继承自Hashtable的父类Dictionary。elements() 方法用于返回此Hashtable中的value的枚举。

contains()方法判断该Hashtable是否包含传入的value。它的作用与containsValue()一致。事实上，containsValue()就只是调用了一下contains()方法。

3、对null的支持不同

Hashtable : key和value都不能为null。

HashMap : key可以为null，但是这样的key只能有一个，因为必须保证key的唯一性；可以有多个key值对应的value为null。

4、安全性不同

HashMap是线程不安全的，在多线程并发的环境下，可能会产生死锁等问题，因此需要开发人员自己处理多线程的安全问题。

Hashtable是线程安全的，它的每个方法上都有synchronized关键字，因此可直接用于多线程中。

虽然HashMap是线程不安全的，但是它的效率远远高于Hashtable，这样设计是合理的，因为大部分的使用场景都是单线程。当需要多线程操作的时候可以使用线程安全的ConcurrentHashMap。

ConcurrentHashMap虽然也是线程安全的，但是它的效率比Hashtable要高好多倍。因为

ConcurrentHashMap使用了分段锁，并不对整个数据进行锁定。

5、初始容量大小和每次扩充容量大小不同

6、计算hash值的方法不同

3、Collection包结构，与Collections的区别

Collection是集合类的上级接口，子接口有Set、List、LinkedList、ArrayList、Vector、Stack、Set；

Collections是集合类的一个帮助类，它包含有各种有关集合操作的静态多态方法，用于实现对各种集合的搜索、排序、线程安全化等操作。此类不能实例化，就像一个工具类，服务于Java的Collection框架。

4、泛型常用特点（待补充）

泛型是Java SE 1.5之后的特性，《Java核心技术》中对泛型的定义是：

“泛型”意味着编写的代码可以被不同类型的对象所重用。

“泛型”，顾名思义，“泛指的类型”。我们提供了泛指的概念，但具体执行的时候却可以有具体的规则来约束，比如我们用的非常多的ArrayList就是个泛型类，ArrayList作为集合可以存放各种元素，如Integer、String，自定义的各种类型等，但在我们使用的时候通过具体的规则来约束，如我们可以约束集合中只存放Integer类型的元素，如

```
List<Integer> initData = new ArrayList<>()
```

使用泛型的好处？

以集合来举例，使用泛型的好处是我们不必因为添加元素类型的不同而定义不同类型的集合，如整型集合类，浮点型集合类，字符串集合类，我们可以定义一个集合来存放整型、浮点型，字符串型数据，而这并不是最重要的，因为我们只要把底层存储设置了Object即可，添加的数据全部都可向上转型为Object。更重要的是我们可以通过规则按照自己的想法控制存储的数据类型。

5、说说List,Set,Map三者的区别

List(对付顺序的好帮手) : List接口存储一组不唯一（可以有多个元素引用相同的对象），有序的对象

Set(注重独一无二的性质) : 不允许重复的集合。不会有多个元素引用相同的对象。

Map(用Key来搜索的专) : 使用键值对存储。Map会维护与Key有关联的值。两个Key可以引用相同的对象，但Key不能重复，典型的Key是String类型，但也可以是任何对象。

6、Array与ArrayList有什么不一样？

Array与ArrayList都是用来存储数据的集合。ArrayList底层是使用数组实现的，但是ArrayList对数组进行了封装和功能扩展，拥有许多原生数组没有的一些功能。我们可以理解成ArrayList是Array的一个升级版。

7、Map有什么特点

以键值对存储数据

元素存储循序是无序的

不允许出现重复键

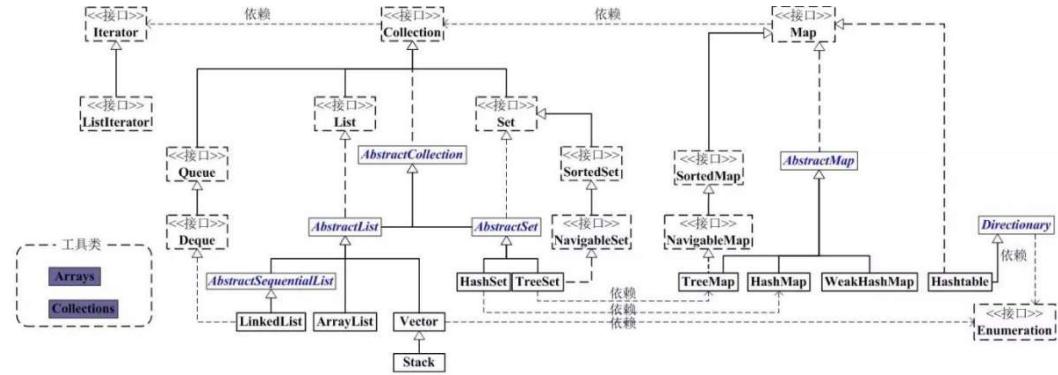
8、集合类存放于 Java.util 包中，主要有几种接口

主要包含set(集)、list(列表)包含 Queue) 和 map(映射)。

1. Collection : Collection 是集合 List、Set、Queue 的最基本的接口。

2. Iterator : 迭代器，可以通过迭代器遍历集合中的数据

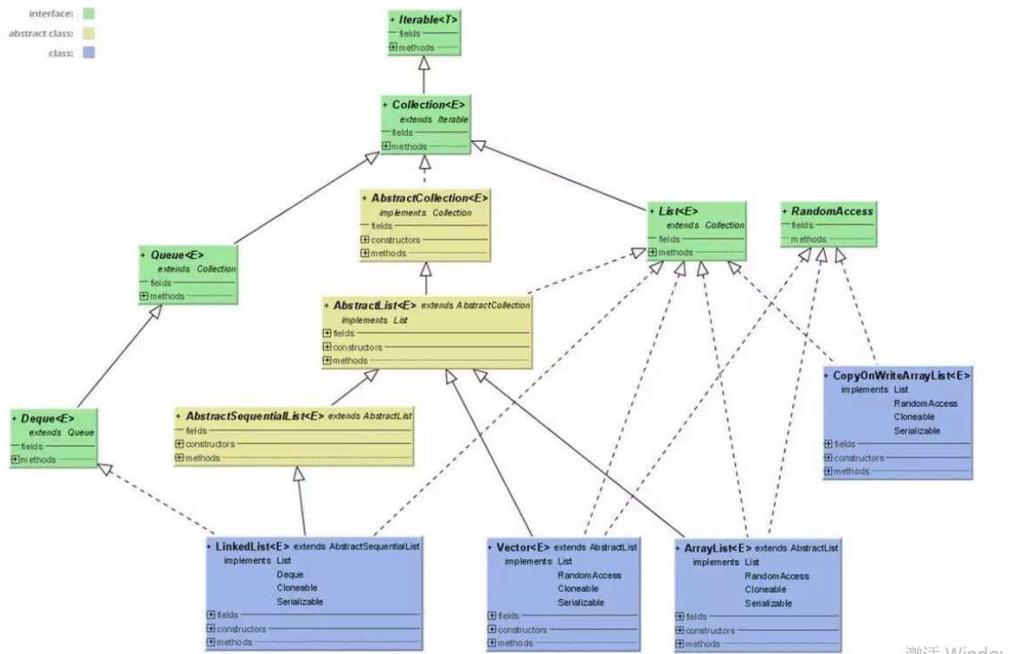
3. Map : 是映射表的基础接口



9、什么是list接口

Java 的 List 是非常常用的数据类型。List 是有序的 Collection。Java List 一共三个实现类：分别是 ArrayList、Vector 和 LinkedList。

list接口结构图



浙江.Window

10、说说ArrayList (数组)

ArrayList 是最常用的 List 实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要将已经有数组的数据复制到新的存储空间中。当从 ArrayList 的中间位置插入或者删除元素时，需要对数组进行复制、移动，代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。

11、Vector (数组实现、线程同步)

Vector 与 ArrayList 一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写 Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问 ArrayList 慢。

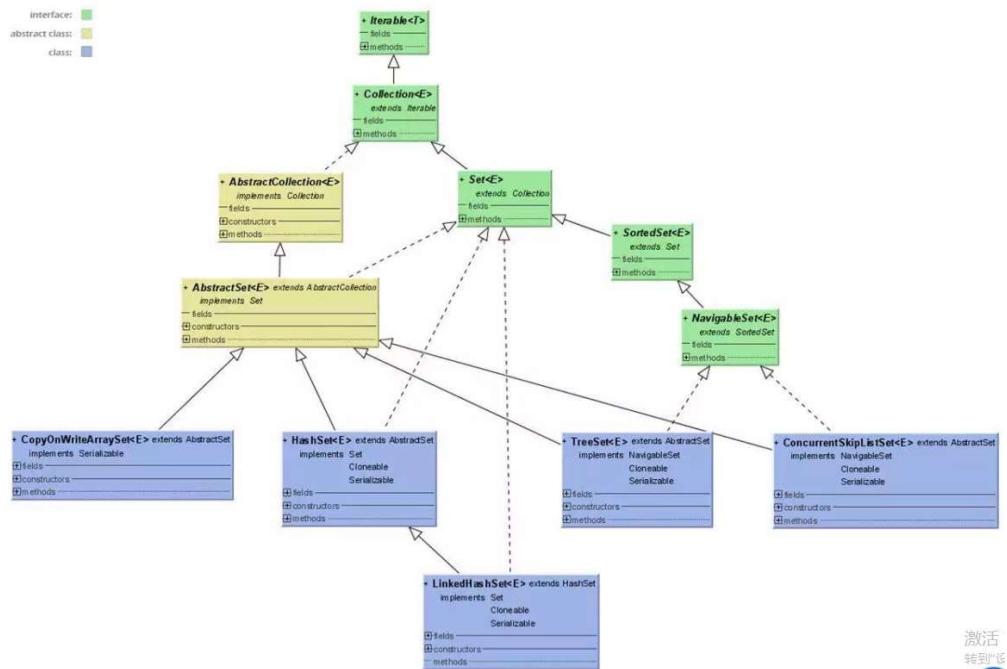
12、说说LinkedList (链表)

LinkedList 是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。另外，他还提供了 List 接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。

13、什么Set集合

Set 注重独一无二的性质，该体系集合用于存储无序(存入和取出的顺序不一定相同)元素，值不能重复。对象的相等性本质是对象 hashCode 值 (java 是依据对象的内存地址计算出的此序号) 判断的，如果想要让两个不同的对象视为相等的，就必须覆盖 Object 的 hashCode 方法和 equals 方法。

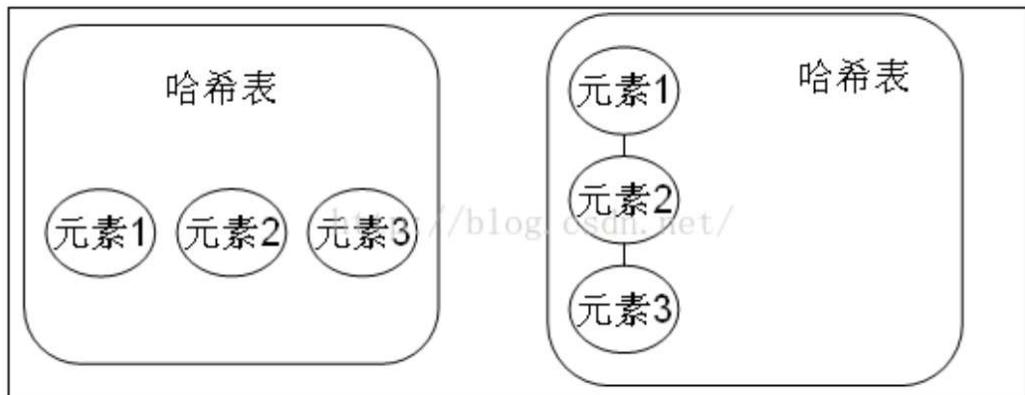
set结构结构图



14、HashSet (Hash 表)

哈希表存放的是哈希值。 HashSet 存储元素的顺序并不是按照存入时的顺序 (和 List 显然不同) 而是按照哈希值来存的所以取数据也是按照哈希值取得。元素的哈希值是通过元素的 hashCode 方法来获取的， HashSet 首先判断两个元素的哈希值，如果哈希值一样，接着会比较 equals 方法如果 equals 结果为 true ， HashSet 就视为同一个元素。如果 equals 为 false 就不是同一个元素。

哈希值相同 equals 为 false 的元素是怎么存储呢，就是在同样的哈希值下顺延 (可以认为哈希值相同的元素放在一个哈希桶中) 。也就是哈希一样的存一列。如图 1 表示 hashCode 值不相同的情况；图 2 表示 hashCode 值相同，但 equals 不相同的情况。



HashSet 通过 hashCode 值来确定元素在内存中的位置。一个 hashCode 位置上可以存放多个元素。

15、什么是TreeSet (二叉树)

1. TreeSet() 是使用二叉树的原理对新 add() 的对象按照指定的顺序排序 (升序、降序) ，每增加一个对象都会进行排序，将对象插入的二叉树指定的位置。
2. Integer 和 String 对象都可以进行默认的 TreeSet 排序，而自定义类的对象是不可以的，自己定义的类必须实现 Comparable 接口，并且覆写相应的 compareTo() 函数，才可以正常使用。
3. 在覆写 compare() 函数时，要返回相应的值才能使 TreeSet 按照一定的规则来排序
4. 比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数

16、说说LinkHashSet (HashSet+LinkedHashMap)

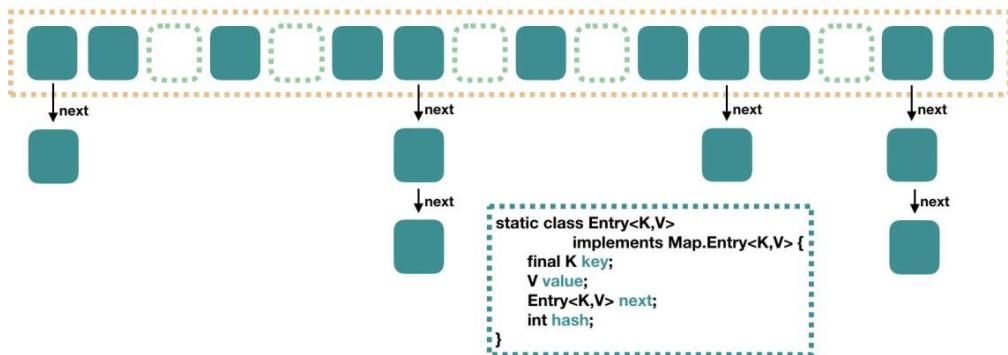
对于 LinkHashSet 而言，它继承与 HashSet、又基于 LinkedHashMap 来实现的。

LinkHashSet 底层使用 LinkedHashMap 来保存所有元素，它继承与 HashSet，其所有的方法操作上又与 HashSet 相同，因此 LinkHashSet 的实现上非常简单，只提供了四个构造方法，并通过传递一个标识参数，调用父类的构造器，底层构造一个 LinkedHashMap 来实现，在相关操作上与父类 HashSet 的操作相同，直接调用父类 HashSet 的方法即可。

17、HashMap (数组+链表+红黑树)

HashMap 根据键的 hashCode 值存储数据，大多数情况下可以直接定位到它的值，因而具有很快的访问速度，但遍历顺序却是不确定的。HashMap 最多只允许一条记录的键为 null，允许多条记录的值为 null。HashMap 非线程安全，即任一时刻可以有多个线程同时写 HashMap，可能会导致数据的不一致。如果需要满足线程安全，可以用 Collections 的 synchronizedMap 方法使 HashMap 具有线程安全的能力，或者使用 ConcurrentHashMap。我们用下面这张图来介绍 HashMap 的结构。

Java7 HashMap 结构



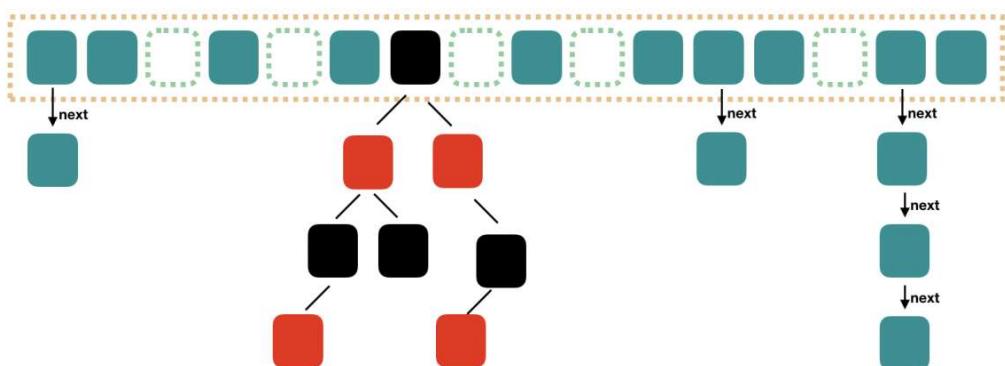
大方向上，HashMap 里面是一个数组，然后数组中每个元素是一个单向链表。上图中，每个绿色的实体是嵌套类 Entry 的实例，Entry 包含四个属性：key, value, hash 值和用于单向链表的 next。

1. capacity : 当前数组容量，始终保持 2^n ，可以扩容，扩容后数组大小为当前的 2 倍。
2. loadFactor : 负载因子，默认为 0.75。
3. threshold : 扩容的阈值，等于 capacity * loadFactor

Java8 对 HashMap 进行了一些修改，最大的不同就是利用了红黑树，所以其由 数组+链表+红黑树 组成。

根据 Java7 HashMap 的介绍，我们知道，查找的时候，根据 hash 值我们能够快速定位到数组的具体下标，但是之后的话，需要顺着链表一个个比较下去才能找到我们需要的，时间复杂度取决于链表的长度，为 $O(n)$ 。为了降低这部分的开销，在 Java8 中，当链表中的元素超过了 8 个以后，会将链表转换为红黑树，在这些位置进行查找的时候可以降低时间复杂度为 $O(\log N)$ 。

Java8 HashMap 结构



18、说说ConcurrentHashMap

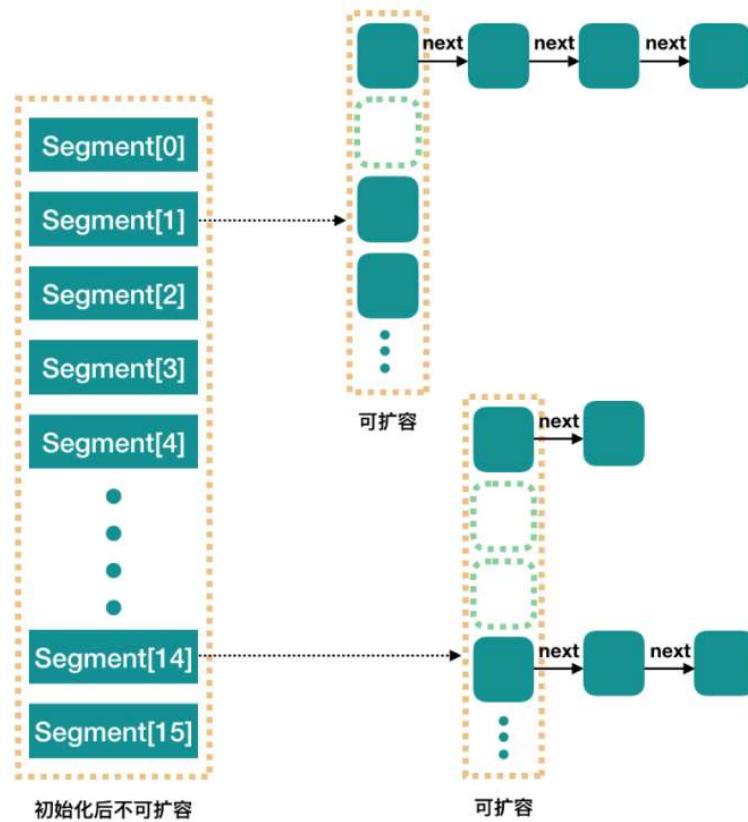
Segment 段

ConcurrentHashMap 和 HashMap 思路是差不多的，但是因为它支持并发操作，所以要复杂一些。整个 ConcurrentHashMap 由一个个 Segment 组成，Segment 代表“部分”或“一段”的意思，所以很多地方都会将其描述为分段锁。注意，行文中，我很多地方用了“槽”来代表一个 segment。

线程安全 (Segment 继承 ReentrantLock 加锁)

简单理解就是，ConcurrentHashMap 是一个 Segment 数组，Segment 通过继承 ReentrantLock 来进行加锁，所以每次需要加锁的操作锁定的是一个 segment，这样只要保证每个 Segment 是线程安全的，也就实现了全局的线程安全

Java7 ConcurrentHashMap 结构



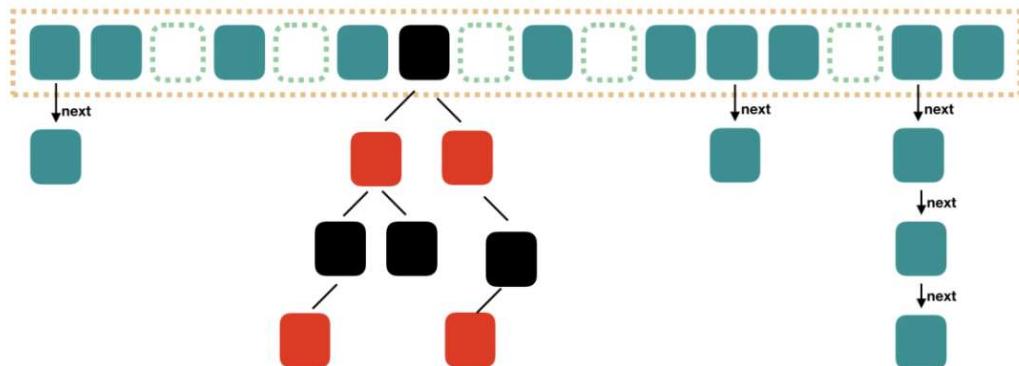
并行度（默认 16）

concurrencyLevel：并行级别、并发数、Segment 数，怎么翻译不重要，理解它。默认是 16，也就是说 ConcurrentHashMap 有 16 个 Segments，所以理论上，这个时候，最多可以同时支持 16 个线程并发写，只要它们的操作分别分布在不同的 Segment 上。这个值可以在初始化的时候设置为其他值，但是一旦初始化以后，它是不可以扩容的。再具体到每个 Segment 内部，其实每个 Segment 很像之前介绍的 HashMap，不过它要保证线程安全，所以处理起来要麻烦些。

Java8 实现（引入了红黑树）

Java8 对 ConcurrentHashMap 进行了比较大的改动。Java8 也引入了红黑树。

Java8 ConcurrentHashMap 结构



19、HashTable (线程安全)

Hashtable 是遗留类，很多映射的常用功能与 HashMap 类似，不同的是它承自 Dictionary 类，并且是线程安全的，任一时间只有一个线程能写 Hashtable，并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁。Hashtable 不建议在新代码中使用，不需要线程安全的场合可以用 HashMap 替换，需要线程安全的场合可以用 ConcurrentHashMap 替换

20、TreeMap (可排序)

TreeMap 实现 SortedMap 接口，能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。
如果使用排序的映射，建议使用 TreeMap。
在使用 TreeMap 时，key 必须实现 Comparable 接口或者在构造 TreeMap 传入自定义的 Comparator，否则会在运行时抛出 java.lang.ClassCastException 类型的异常。
参考：<https://www.ibm.com/developerworks/cn/java/j-lo-tree/index.html>

21、LinkHashMap (记录插入顺序)

LinkedHashMap 是 HashMap 的一个子类，保存了记录的插入顺序，在用 Iterator 遍历 LinkedHashMap 时，先得到的记录肯定是最先插入的，也可以在构造时带参数，按照访问次序排序。
参考 1：<http://www.importnew.com/28263.html>
参考 2：<http://www.importnew.com/20386.html#comment-648123>

22、泛型类

泛型类的声明和非泛型类的声明类似，除了在类名后面添加了类型参数声明部分。和泛型方法一样，泛型类的类型参数声明部分也包含一个或多个类型参数，参数间用逗号隔开。一个泛型参数，也被称为一个类型变量，是用于指定一个泛型类型名称的标识符。因为他们接受一个或多个参数，这些类被称为参数化的类或参数化的类型。

```
public class Box<T> {  
    private T t;  
    public void add(T t) {  
        this.t = t;  
    }  
    public T get() {  
        return t;  
    }  
}
```

23、类型通配符？

类型通配符一般是使用?代替具体的类型参数。例如 List<?> 在逻辑上是List,List 等所有 List<具体类型实参>的父类。

24、类型擦除

Java 中的泛型基本上都是在编译器这个层次来实现的。在生成的 Java 字节代码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会被编译器在编译的时候去掉。这个过程就称为类型擦除。

如在代码中定义的 List 和 List 等类型，在编译之后都会变成 List。JVM 看到的只是 List，而由泛型附加的类型信息对 JVM 来说是不可见的。

类型擦除的基本过程也比较简单，首先是找到用来替换类型参数的具体类。这个具体类一般是 Object。如果指定了类型参数的上界的话，则使用这个上界。把代码中的类型参数都替换成具体的类。

Java异常面试题

1、Java中异常分为哪两种？

编译时异常
运行时异常

2、异常的处理机制有几种？

异常捕捉：try...catch...finally，异常抛出：throws。

3、如何自定义一个异常

继承一个异常类，通常是RumtimeException或者Exception

4、try catch finally , try里有return , finally还执行么？

执行，并且finally的执行早于try里面的return

结论：

- 1、不管有没有出现异常，finally块中代码都会执行；
- 2、当try和catch中有return时，finally仍然会执行；
- 3、finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，管finally中的代码怎么样，返回的值都不会改变，任然是之前保存的值），所以函数返回值是在finally执行前确定的；
- 4、finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

5、Exception与Error包结构

Java可抛出(Throwable)的结构分为三种类型：被检查的异常(CheckedException)，运行时异常(RuntimeException)，错误(Error)。

1、运行时异常

定义：RuntimeException及其子类都被称为运行时异常。

特点：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时，倘若既“没有通过throws声明抛出它”，也“没有用try-catch语句捕获它”，还是会编译通过。例如，除数为零时产生的ArithmaticException异常，数组越界时产生的IndexOutOfBoundsException异常，fail-fast机制产生的ConcurrentModificationException异常（java.util包下面的所有集合类都是快速失败的，“快速失败”也就是fail-fast，它是Java集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生fail-fast机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程1、线程2），线程1通过Iterator在遍历集合A中的元素，在某个时候线程2修改了集合A的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出ConcurrentModificationException异常，从而产生fail-fast机制，这个错叫并发修改异常。Fail-safe，java.util.concurrent包下面的所有类都是安全失败的，在遍历过程中，如果已经遍历的数组上的内容变化了，迭代器不会抛出ConcurrentModificationException异常。如果未遍历的数组上的内容发生了变化，则有可能反映到迭代过程中。这就是ConcurrentHashMap迭代器弱一致的表现。ConcurrentHashMap的弱一致性主要是为了提升效率，是一致性与效率之间的一种权衡。要成为强一致性，就得到处使用锁，甚至是全局锁，这就与Hashtable和同步的HashMap一样了。）等，都属于运行时异常。

常见的五种运行时异常：

ClassCastException（类转换异常）

IndexOutOfBoundsException（数组越界）

NullPointerException（空指针异常）

ArrayStoreException（数据存储异常，操作数组是类型不一致）

Bu?erOver?owException

2、被检查异常

定义：Exception类本身，以及Exception的子类中除了“运行时异常”之外的其它子类都属于被检查异常。特点：Java编译器会检查它。此类异常，要么通过throws进行声明抛出，要么通过try-catch进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException就属于被检查异常。当通过clone()接口去克隆一个对象，而该对象对应的类没有实现Cloneable接口，就会抛出CloneNotSupportedException异常。被检查异常通常都是可以恢复的。

如：

IOException

FileNotFoundException

SQLException

被检查的异常适用于那些不是因程序引起的错误情况，比如：读取文件时文件不存在引发的FileNotFoundException。然而，不被检查的异常通常都是由于糟糕的编程引起的，比如：在对象引用时没有确保对象非空而引起的NullPointerException。

3、错误

定义：Error类及其子类。

特点：和运行时异常一样，编译器也不会对错误进行检查。当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError就属于错误。出现这种错误会导致程序终止运行。OutOfMemoryError、ThreadDeath。

Java虚拟机规范规定JVM的内存分为了好几块，比如堆，栈，程序计数器，方法区等

6、Throw与throws区别

位置不同：

1. throws 用在函数上，后面跟的是异常类，可以跟多个；而 throw 用在函数内，后面跟的是异常对象。

功能不同：

2. throws 用来声明异常，让调用者只知道该功能可能出现的问题，可以给出预先的处理方式；throw 抛出具体的问题对象，执行到 throw，功能就已经结束了，跳转到调用者，并将具体的问题对象抛给调用者。也就是说 throw 语句独立存在时，下面不要定义其他语句，因为执行不到。

3. throws 表示出现异常的一种可能性，并不一定会发生这些异常；throw 则是抛出了异常，执行 throw 则一定抛出了某种异常对象。

4. 两者都是消极处理异常的方式，只是抛出或者可能抛出异常，但是不会由函数去处理异常，真正的处理异常由函数的上层调用处理。

7、Error与Exception区别？

Error和Exception都是java错误处理机制的一部分，都继承了Throwable类。

Exception表示的异常，异常可以通过程序来捕捉，或者优化程序来避免。

Error表示的是系统错误，不能通过程序来进行错误处理。

8、error和exception有什么区别

error 表示恢复不是不可能但很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况 exception 表示一种设计或实现问题。也就是说，它表示如果程序运行正常，从不会发生的情况

Java中的IO与NIO面试题

1、Java 中 IO 流？

Java 中 IO 流分为几种？

1. 按照流的流向分，可以分为输入流和输出流；
2. 按照操作单元划分，可以划分为字节流和字符流；
3. 按照流的角色划分为节点流和处理流。

Java Io 流共涉及 40 多个类，这些类看上去很杂乱，但实际上很有规则，而且彼此之间存在非常紧密的联系，Java Io 流的 40 多个类都是从如下 4 个抽象类基类中派生出来的。

1. InputStream/Reader: 所有的输入流的基类，前者是字节输入流，后者是字符输入流。
2. OutputStream/Writer: 所有输出流的基类，前者是字节输出流，后者是字符输出流。

2、Java IO与 NIO的区别

NIO即New IO，这个库是在JDK1.4中才引入的。NIO和IO有相同的作用和目的，但实现方式不同，NIO 主要用到的是块，所以NIO的效率要比IO高很多。在Java API中提供了两套NIO，一套是针对标准输入输出NIO，另一套就是网络编程NIO。

3、常用io类有那些

```
File  
FileInputStream, FileOutputStream  
BufferInputStream, BufferedOutputStream  
PrintWriter  
FileReader, FileWriter  
BufferedReader, BufferedWriter  
ObjectInputStream, ObjectOutputStream
```

4、字节流与字符流的区别

以字节为单位输入输出数据，字节流按照8位传输
以字符为单位输入输出数据，字符流按照16位传输

5、阻塞 IO 模型

最传统的一种 IO 模型，即在读写数据过程中会发生阻塞现象。当用户线程发出 IO 请求之后，内核会去查看数据是否就绪，如果没有就绪就会等待数据就绪，而用户线程就会处于阻塞状态，用户线程交出 CPU。当数据就绪之后，内核会将数据拷贝到用户线程，并返回结果给用户线程，用户线程才解除 block 状态。典型的阻塞 IO 模型的例子为：data = socket.read();如果数据没有就绪，就会一直阻塞在 read 方法

6、非阻塞 IO 模型

当用户线程发起一个 read 操作后，并不需要等待，而是马上就得到了一个结果。如果结果是一个error 时，它就知道数据还没有准备好，于是它可以再次发送 read 操作。一旦内核中的数据准备好了，并且又再次收到了用户线程的请求，那么它马上就将数据拷贝到了用户线程，然后返回。所以事实上，在非阻塞 IO 模型中，用户线程需要不断地询问内核数据是否就绪，也就是说非阻塞 IO 不会交出 CPU，而会一直占用 CPU。典型的非阻塞 IO 模型一般如下：

```
while(true){  
    data = socket.read();  
    if(data != error){  
        //处理数据  
        break;  
    }  
}
```

但是对于非阻塞 IO 就有一个非常严重的问题，在 while 循环中需要不断地去询问内核数据是否就绪，这样会导致 CPU 占用率非常高，因此一般情况下很少使用 while 循环这种方式来读取数据。

7、多路复用 IO 模型

多路复用 IO 模型是目前使用得比较多的模型。Java NIO 实际上就是多路复用 IO。在多路复用 IO 模型中，会有一个线程不断去轮询多个 socket 的状态，只有当 socket 真正有读写事件时，才真正调用实际的 IO 读写操作。因为在多路复用 IO 模型中，只需要使用一个线程就可以管理多个 socket，系统不需要建立新的进程或者线程，也不必维护这些线程和进程，并且只有在真正有 socket 读写事件进行时，才会使用 IO 资源，所以它大大减少了资源占用。在 Java NIO 中，是通过 selector.select() 去查询每个通道是否有到达事件，如果没有事件，则一直阻塞在那里，因此这种方式会导致用户线程的阻塞。多路复用 IO 模式，通过一个线程就可以管理多个 socket，只有当 socket 真正有读写事件发生才会占用资源来进行实际的读写操作。因此，多路复用 IO 比较适合连接数比较多的情况。

另外多路复用 IO 为何比非阻塞 IO 模型的效率高是因为在非阻塞 IO 中，不断地询问 socket 状态时通过用户线程去进行的，而在多路复用 IO 中，轮询每个 socket 状态是内核在进行的，这个效率要比用户线程要高的多。

不过要注意的是，多路复用 IO 模型是通过轮询的方式来检测是否有事件到达，并且对到达的事件逐一进行响应。因此对于多路复用 IO 模型来说，一旦事件响应体很大，那么就会导致后续的事件迟迟得不到处理，并且会影响新的事件轮询。

8、信号驱动 IO 模型

在信号驱动 IO 模型中，当用户线程发起一个 IO 请求操作，会给对应的 socket 注册一个信号函数，然后用户线程会继续执行，当内核数据就绪时会发送一个信号给用户线程，用户线程接收到信号之后，便在信号函数中调用 IO 读写操作来进行实际的 IO 请求操作。

9、异步 IO 模型

异步 IO 模型才是最理想的 IO 模型，在异步 IO 模型中，当用户线程发起 read 操作之后，立刻就可以开始去做其它的事。而另一方面，从内核的角度，当它受到一个 asynchronous read 之后，它会立刻返回，说明 read 请求已经成功发起了，因此不会对用户线程产生任何 block。然后，内核会等待数据准备完成，然后将数据拷贝到用户线程，当这一切都完成之后，内核会给用户线程发送一个信号，告诉它 read 操作完成了。也就是说用户线程完全不需要实际的整个 IO 操作是如何进行的，只需要先发起一个请求，当接收内核返回的成功信号时表示 IO 操作已经完成，可以直接去使用数据了。

也就说在异步 IO 模型中，IO 操作的两个阶段都不会阻塞用户线程，这两个阶段都是由内核自动完成，然后发送一个信号告知用户线程操作已完成。用户线程中不需要再次调用 IO 函数进行具体的读写。这点是和信号驱动模型有所不同的，在信号驱动模型中，当用户线程接收到信号表示数据已经就绪，然后需要用户线程调用 IO 函数进行实际的读写操作；而在异步 IO 模型中，收到信号表示 IO 操作已经完成，不需要再在用户线程中调用 IO 函数进行实际的读写操作。

注意，异步 IO 是需要操作系统的底层支持，在 Java 7 中，提供了 Asynchronous IO。

更多参考：<http://www.importnew.com/19816.html>

10、JAVA NIO

NIO 主要有三大核心部分：Channel(通道)，Buffer(缓冲区)，Selector。传统 IO 基于字节流和字符流进行操作，而 NIO 基于 Channel 和 Buffer(缓冲区)进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入到通道中。Selector(选择区)用于监听多个通道的事件（比如：连接打开，数据到达）。因此，单个线程可以监听多个数据通道。NIO 和传统 IO 之间第一个最大的区别是，IO 是面向流的，NIO 是面向缓冲区的。

11、NIO 的缓冲区

Java IO 面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。NIO 的缓冲导向方法不同。数据读取到一个稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

12、NIO 的非阻塞

IO 的各种流是阻塞的。这意味着，当一个线程调用 read() 或 write() 时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。NIO 的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程现在可以管理多个输入和输出通道（channel）。

13、Channel

首先说一下 Channel，国内大多翻译成“通道”。Channel 和 IO 中的 Stream(流)是差不多一个等级的。只不过 Stream 是单向的，譬如：InputStream, OutputStream，而 Channel 是双向的，既可以用来进行读操作，又可以用来进行写操作。NIO 中的 Channel 的主要实现有：

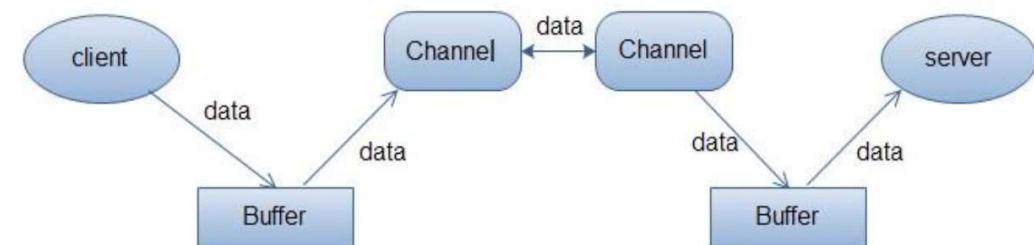
1. FileChannel
2. DatagramChannel
3. SocketChannel
4. ServerSocketChannel

这里看名字就可以猜出个所以然来：分别可以对应文件 IO、UDP 和 TCP（Server 和 Client）。

下面演示的案例基本上就是围绕这 4 个类型的 Channel 进行陈述的。

14、Buffer

Buffer，故名思意，缓冲区，实际上是一个容器，是一个连续数组。Channel 提供从文件、网络读取数据的渠道，但是读取或写入的数据都必须经由 Buffer。



上面的图描述了从一个客户端向服务端发送数据，然后服务端接收数据的过程。客户端发送数据时，必须先将数据存入 Buffer 中，然后将 Buffer 中的内容写入通道。服务端这边接收数据必须通过 Channel 将数据读入到 Buffer 中，然后再从 Buffer 中取出数据来处理。

在 NIO 中，Buffer 是一个顶层父类，它是一个抽象类，常用的 Buffer 的子类有：ByteBuffer、IntBuffer、CharBuffer、LongBuffer、DoubleBuffer、FloatBuffer、ShortBuffer

15、Selector

Selector 类是 NIO 的核心类，Selector 能够检测多个注册的通道上是否有事件发生，如果有事件发生，便获取事件然后针对每个事件进行相应的响应处理。这样一来，只是用一个单线程就可以管理多个通道，也就是管理多个连接。这样使得只有在连接真正有读写事件发生时，才会调用函数来进行读写，就大大地减少了系统开销，并且不必为每个连接都创建一个线程，不用去维护多个线程，并且避免了多线程之间的上下文切换导致的开销。

Java反射面试题

1、除了使用new创建对象之外，还可以用什么方法创建对象？

使用Java反射可以创建对象！

2、Java反射创建对象效率高还是通过new创建对象的效率高？

通过new创建对象的效率比较高。通过反射时，先查找类资源，使用类加载器创建，过程比较繁琐，所以效率较低。

3、java反射的作用

反射机制是在运行时，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法。在java中，只要给定类的名字，就可以通过反射机制来获得类的所有信息。

这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

4、哪里会用到反射机制？

jdbc就是典型的反射

```
Class.forName('com.mysql.jdbc.Driver.class');//加载MySQL的驱动类
```

这就是反射。如hibernate，struts等框架使用反射实现的。

5、反射的实现方式：

第一步：获取Class对象，有4种方法：

- 1) Class.forName("类的路径")；
- 2) 类名.class
- 3) 对象名.getClass()
- 4) 基本类型的包装类，可以调用包装类的Type属性来获得该包装类的Class对象

6、实现Java反射的类：

- 1) Class：表示正在运行的Java应用程序中的类和接口

注意：所有获取对象的信息都需要Class类来实现。

- 2) Field：提供有关类和接口的属性信息，以及对它的动态访问权限。
- 3) Constructor：提供关于类的单个构造方法的信息以及它的访问权限
- 4) Method：提供类或接口中某个方法的信息

7、反射机制的优缺点：

优点：

- 1) 能够运行时动态获取类的实例，提高灵活性；
- 2) 与动态编译结合

缺点：

1) 使用反射性能较低，需要解析字节码，将内存中的对象进行解析。

解决方案：

1、通过setAccessible(true)关闭JDK的安全检查来提升反射速度；

2、多次创建一个类的实例时，有缓存会快很多

3、RefilectASM工具类，通过字节码生成的方式加快反射速度

2) 相对不安全，破坏了封装性（因为通过反射可以获得私有方法和属性）

8、Java 反射 API

反射 API 用来生成 JVM 中的类、接口或对象的信息。

1. Class 类：反射的核心类，可以获取类的属性，方法等信息。

2. Field 类：java.lang.reflec 包中的类，表示类的成员变量，可以用来获取和设置类之中的属性

值。

3. Method 类：java.lang.reflec 包中的类，表示类的方法，它可以用来获取类中的方法信息或者执行方法。

4. Constructor 类：java.lang.reflec 包中的类，表示类的构造方法。

9、反射使用步骤（获取 Class 对象、调用对象方法）

1. 获取想要操作的类的 Class 对象，他是反射的核心，通过 Class 对象我们可以任意调用类的方法。

2. 调用 Class 类中的方法，既然是反射的使用阶段。

3. 使用反射 API 来操作这些信息。

10、获取 Class 对象有几种方法

调用某个对象的 getClass() 方法

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

调用某个类的 class 属性来获取该类对应的 Class 对象

```
Class clazz=Person.class;
```

使用 Class 类中的 forName() 静态方法(最安全/性能最好)

```
Class clazz=Class.forName("类的全路径"); (最常用)
```

当我们获得了想要操作的类的 Class 对象后，可以通过 Class 类中的方法获取并查看该类中的方法和属性。

```
//获取 Person 类的 Class 对象
Class clazz=Class.forName("reflection.Person");
//获取 Person 类的所有方法信息
Method[] method=clazz.getDeclaredMethods();
for(Method m:method){
    System.out.println(m.toString());
}
//获取 Person 类的所有成员属性信息
Field[] field=clazz.getDeclaredFields();
for(Field f:field{
    System.out.println(f.toString());
}
//获取 Person 类的所有构造方法信息
Constructor[] constructor=clazz.getDeclaredConstructors();
for(Constructor c:constructor{
    System.out.println(c.toString());
}
```

11、利用反射动态创建对象实例

Class 对象的 newInstance()

1. 使用 Class 对象的 newInstance() 方法来创建该 Class 对象对应类的实例，但是这种方法要求该 Class 对象对应的类有默认的空构造器。

调用 Constructor 对象的 newInstance()

2. 先使用 Class 对象获取指定的 Constructor 对象，再调用 Constructor 对象的 newInstance()

方法来创建 Class 对象对应类的实例。通过这种方法可以选定构造方法创建实例。

```
//获取 Person 类的 Class 对象  
Class clazz=Class.forName("reflection.Person");  
  
//使用newInstane 方法创建对象  
Person p=(Person) clazz.newInstance();  
  
//获取构造方法并创建对象  
Constructor c=clazz.getDeclaredConstructor(String.class,String.class,int.class);  
  
//创建对象并设置属性13/04/2018  
Person p1=(Person) c.newInstance("李四","男",20);
```

Java序列化面试题

1、什么是java序列化，如何实现java序列化？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。序列化的实现：将需要被序列化的类实现Serializable接口，该接口没有需要实现的方法，implements Serializable只是为了标注该对象是可被序列化的，然后使用一个输出流(如：FileOutputStream)来构造一个ObjectOutputStream(对象流)对象，接着，使用ObjectOutputStream对象的writeObject(Object obj)方法就可以将参数为obj的对象写出(即保存其状态)，要恢复的话则用输入流。

2、保存(持久化)对象及其状态到内存或者磁盘

Java 平台允许我们在内存中创建可复用的 Java 对象，但一般情况下，只有当 JVM 处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比 JVM 的生命周期更长。但在现实应用中，就可能要求在 JVM 停止运行之后能够保存(持久化)指定的对象，并在将来重新读取被保存的对象。Java 对象序列化就能够帮助我们实现该功能。

3、序列化对象以字节数组保持-静态成员不保存

使用 Java 对象序列化，在保存对象时，会把其状态保存为一组字节，在未来，再将这些字节组装成对象。必须注意地是，对象序列化保存的是对象的“状态”，即它的成员变量。由此可知，对象序列化不会关注类中的静态变量。

4、序列化用户远程对象传输

除了在持久化对象时会用到对象序列化之外，当使用 RMI(远程方法调用)，或在网络中传递对象时，都会用到对象序列化。Java 序列化 API 为处理对象序列化提供了一个标准机制，该 API 简单易用。

5、Serializable 实现序列化

在 Java 中，只要一个类实现了 java.io.Serializable 接口，那么它就可以被序列化。 ObjectOutputStream 和 ObjectInputStream 对对象进行序列化及反序列化通过 ObjectOutputStream 和 ObjectInputStream 对对象进行序列化及反序列化。

6、writeObject 和 readObject 自定义序列化策略

在类中增加 writeObject 和 readObject 方法可以实现自定义序列化策略。

7、序列化 ID

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（就是 private static final long serialVersionUID

8、序列化并不保存静态变量

序列化子父类说明

要想将父类对象也序列化，就需要让父类也实现 Serializable 接口。

9、Transient 关键字阻止该变量被序列化到文件中

1. 在变量声明前加上 Transient 关键字，可以阻止该变量被序列化到文件中，在被反序列化后， transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null。
2. 服务器端给客户端发送序列化对象数据，对象中有一些数据是敏感的，比如密码字符串等，希望对该密码字段在序列化时，进行加密，而客户端如果拥有解密的密钥，只有在客户端进行反序列化时，才可以对密码进行读取，这样可以一定程度保证序列化对象的数据安全。

10、序列化（深 clone 一中实现）

在 Java 语言里深复制一个对象，常常可以先使对象实现 Serializable 接口，然后把对象（实际上只是对象的一个拷贝）写到一个流里，再从流里读出来，便可以重建对象。

Java注解面试题

1、4种标准元注解是哪四种？

元注解的作用是负责注解其他注解。Java5.0 定义了 4 个标准的 meta-annotation 类型，它们被用来提供对其它 annotation 类型作说明。

@Target 修饰的对象范围

@Target说明了Annotation所修饰的对象范围：Annotation可被用于 packages、types（类、接口、枚举、Annotation 类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数和本地变量（如循环变量、catch 参数）。在 Annotation 类型的声明中使用了 target 可更加明晰其修饰的目标

@Retention 定义 被保留的时间长短

Retention 定义了该 Annotation 被保留的时间长短：表示需要在什么级别保存注解信息，用于描述注解的生命周期（即：被描述的注解在什么范围内有效），取值（RetentionPoicy）由：

1. SOURCE:在源文件中有效（即源文件保留）
2. CLASS:在 class 文件中有效（即 class 保留）
3. RUNTIME:在运行时有效（即运行时保留）
- 4.

@Documented 描述-javadoc

@ Documented 用于描述其它类型的 annotation 应该被作为被标注的程序成员的公共 API，因此可以被例如 javadoc 此类的工具文档化。

@Inherited 阐述了某个被标注的类型是被继承的

@Inherited 元注解是一个标记注解，@Inherited 阐述了某个被标注的类型是被继承的。如果一个使用了@Inherited 修饰的 annotation 类型被用于一个 class，则这个 annotation 将被用于该 class 的子类。

2、注解是什么？

Annotation (注解) 是 Java 提供的一种对元程序中元素关联信息和元数据 (metadata) 的途径和方法。 Annotation(注解)是一个接口，程序可以通过反射来获取指定程序中元素的 Annotation 对象，然后通过该 Annotation 对象来获取注解中的元数据信息。

多线程&并发面试题

JAVA 并发知识库



1、Java中实现多线程有几种方法

继承Thread类；
实现Runnable接口；
实现Callable接口通过FutureTask包装器来创建Thread线程；
使用ExecutorService、Callable、Future实现有返回结果的多线程（也就是使用了ExecutorService来管理前面的三种方式）。

2、继承 Thread 类

Thread 类本质上是实现了 Runnable 接口的一个实例，代表一个线程的实例。启动线程的唯一方法就是通过 Thread 类的 start() 实例方法。 start() 方法是一个 native 方法，它将启动一个新的线程，并执行 run() 方法。

```

public class MyThread extends Thread {
    public void run() {
        System.out.println("MyThread.run()");
    }
}
MyThread myThread1 = new MyThread();
myThread1.start();

```

3、实现 Runnable 接口。

如果自己的类已经 extends 另一个类，就无法直接 extends Thread，此时，可以实现一个 Runnable 接口。

```

public class MyThread extends OtherClass implements Runnable {
    public void run() {
        System.out.println("MyThread.run()");
    }
}
//启动 MyThread，需要首先实例化一个 Thread，并传入自己的 MyThread 实例：
MyThread myThread = new MyThread();
Thread thread = new Thread(myThread);
thread.start();
//事实上，当传入一个 Runnable target 参数给 Thread 后， Thread 的 run()方法就会调用
target.run()
public void run() {
    if (target != null) {
        target.run();
    }
}

```

4、ExecutorService、 Callable、 Future 有返回值线程

有返回值的任务必须实现 Callable 接口，类似的，无返回值的任务必须 Runnable 接口。执行 Callable 任务后，可以获取一个 Future 的对象，在该对象上调用 get 就可以获取到 Callable 任务返回的 Object 了，再结合线程池接口 ExecutorService 就可以实现传说中有返回结果的多线程了。

```

//创建一个线程池
ExecutorService pool = Executors.newFixedThreadPool(taskSize);
// 创建多个有返回值的任务
List<Future> list = new ArrayList<Future>();
for (int i = 0; i < tasksize; i++) {
    Callable c = new MyCallable(i + " ");
    // 执行任务并获取 Future 对象
    Future f = pool.submit(c);
    list.add(f);
}
// 关闭线程池
pool.shutdown();
// 获取所有并发任务的运行结果
for (Future f : list) {
    // 从 Future 对象上获取任务的返回值，并输出到控制台
    System.out.println("res: " + f.get().toString());
}

```

5、基于线程池的方式

线程和数据库连接这些资源都是非常宝贵的资源。那么每次需要的时候创建，不需要的时候销毁，是非常浪费资源的。那么我们就可以使用缓存的策略，也就是使用线程池。

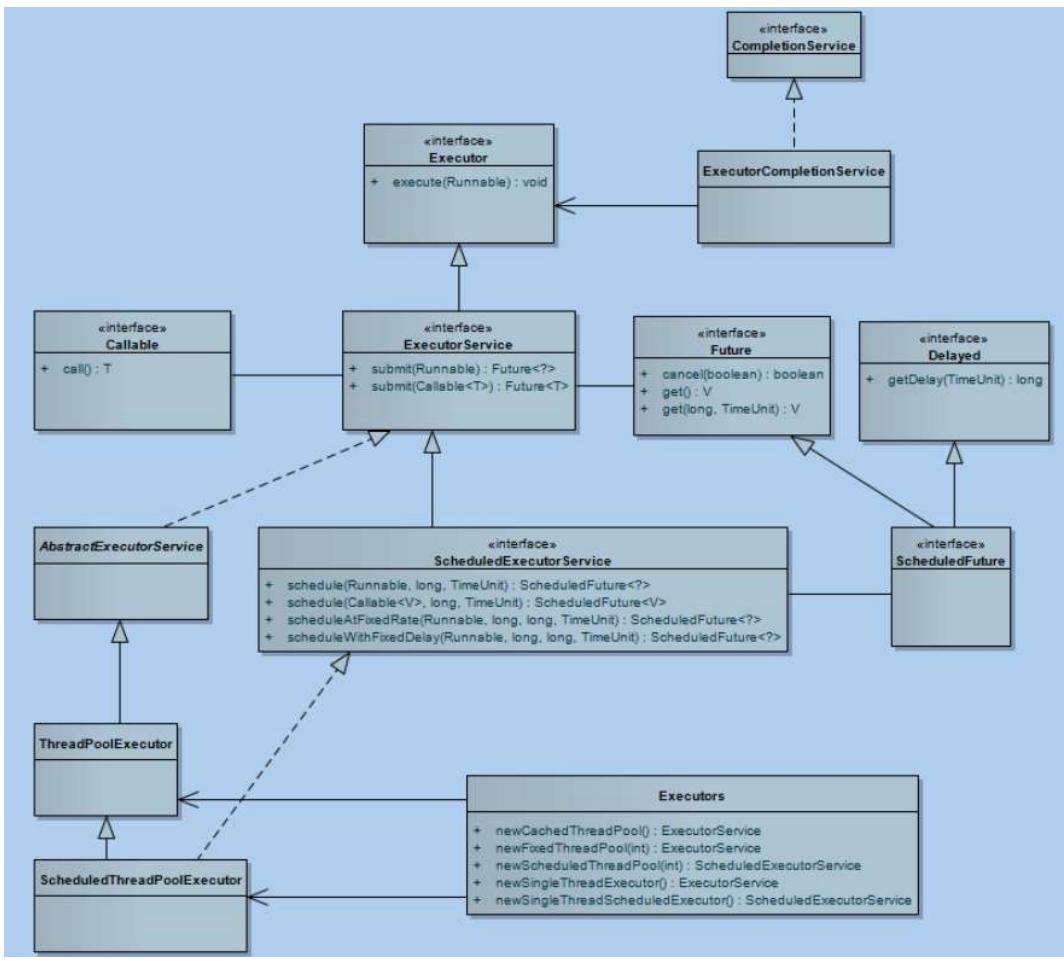
```

// 创建线程池
ExecutorService threadPool = Executors.newFixedThreadPool(10);
while(true) {
    threadPool.execute(new Runnable() { // 提交多个线程任务，并执行
        @Override
        public void run() {
            System.out.println(Thread.currentThread().getName() + " is running ..");
            try {
                Thread.sleep(3000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
}

```

6、4 种线程池

Java 里面线程池的顶级接口是 Executor，但是严格意义上讲 Executor 并不是一个线程池，而只是一个执行线程的工具。真正的线程池接口是 ExecutorService。



newCachedThreadPool

创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。

newFixedThreadPool

创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。在任意点，在大多数 nThreads 线程会处于处理任务的活动状态。如果在所有线程处于活动状态时提交附加任务，则在有可用线程之前，附加任务将在队列中等待。如果在关闭前的执行期间由于失败而导致任何线程终止，那么一个新线程将代替它执行后续的任务（如果需要）。在某个线程被显式地关闭之前，池中的线程将一直存在。

newScheduledThreadPool

创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

```

ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(3);
scheduledThreadPool.schedule(newRunnable() {
    @Override
    public void run() {
        System.out.println("延迟三秒");
    }
}, 3, TimeUnit.SECONDS);
scheduledThreadPool.scheduleAtFixedRate(newRunnable() {
    @Override
    public void run() {
        System.out.println("延迟 1 秒后每三秒执行一次");
    }
}, 1, 3, TimeUnit.SECONDS);

```

newSingleThreadExecutor

Executors.newSingleThreadExecutor()返回一个线程池（这个线程池只有一个线程），这个线程池可以在线程死后（或发生异常时）重新启动一个线程来替代原来的线程继续执行下去！

7、如何停止一个正在运行的线程

- 1、使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
- 2、使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
- 3、使用interrupt方法中断线程。

```

class MyThread extends Thread {
    volatile boolean stop = false;
    public void run() {
        while (!stop) {
            System.out.println(getName() + " is running");
            try {
                sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("wake up from block..."); 
                stop = true; // 在异常处理代码中修改共享变量的状态
            }
        }
        System.out.println(getName() + " is exiting...");
    }
}

class InterruptThreadDemo3 {
    public static void main(String[] args) throws InterruptedException {
        MyThread m1 = new MyThread();
        System.out.println("Starting thread...");
        m1.start();
        Thread.sleep(3000);
        System.out.println("Interrupt thread...: " + m1.getName());
        m1.stop = true; // 设置共享变量为true
        m1.interrupt(); // 阻塞时退出阻塞状态
        Thread.sleep(3000); // 主线程休眠3秒以便观察线程m1的中断情况
        System.out.println("Stopping application...");
    }
}

```

8、notify()和notifyAll()有什么区别？

notify可能会导致死锁，而notifyAll则不会

任何时候只有一个线程可以获得锁，也就是说只有一个线程可以运行synchronized 中的代码
使用notifyAll,可以唤醒

所有处于wait状态的线程，使其重新进入锁的竞争队列中，而notify只能唤醒一个。

wait() 应配合while循环使用，不应使用if，务必在wait()调用前后都检查条件，如果不满足，必须调用notify()唤醒另外的线程来处理，自己继续wait()直至条件满足再往下执行。

notify() 是对notifyAll()的一个优化，但它有很精确的应用场景，并且要求正确使用。不然可能导致死锁。正确的场景应该是 WaitSet中等待的是相同的条件，唤醒任一个都能正确处理接下来的事情，如果唤醒的线程无法正确处理，务必确保继续notify()下一个线程，并且自身需要重新回到WaitSet中。

9、sleep()和wait() 有什么区别？

1. 对于 sleep()方法，我们首先要知道该方法是属于 Thread 类中的。而 wait()方法，则是属于 Object 类中的。
2. sleep()方法导致了程序暂停执行指定的时间，让出 cpu 该其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态
3. 在调用 sleep()方法的过程中，线程不会释放对象锁。
4. 而当调用 wait()方法的时候，线程会放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象调用 notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

10、volatile 是什么?可以保证有序性吗?

一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：

1) 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的,volatile关键字会强制将修改的值立即写入主存。

2) 禁止进行指令重排序。

volatile 不是原子性操作

什么叫保证部分有序性？

当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

```

x = 2;      //语句1
y = 0;      //语句2
flag = true; //语句3
x = 4;      //语句4
y = -1;     //语句5

```

由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

使用Volatile一般用于状态标记量和单例模式的双检锁

11、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。

12、为什么wait, notify 和 notifyAll这些方法不在thread类里面？

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。

13、为什么wait和notify方法要在同步块中调用？

1. 只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的wait(),notify()和notifyAll()方法。
2. 如果你不这么做，你的代码会抛出IllegalMonitorStateException异常。
3. 还有一个原因是为了避免wait和notify之间产生竞态条件。

wait()方法强制当前线程释放对象锁。这意味着在调用某对象的wait()方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的wait()方法。

在调用对象的notify()和notifyAll()方法之前，调用线程必须已经得到该对象的锁。因此，必须在某个对象的同步方法或同步代码块中才能调用该对象的notify()或notifyAll()方法。

调用wait()方法的原因通常是，调用线程希望某个特殊的状态(或变量)被设置之后再继续执行。调用notify()或notifyAll()方法的原因通常是，调用线程希望告诉其他等待中的线程：“特殊状态已经被设置”。这个状态作为线程间通信的通道，它必须是一个可变的共享状态(或变量)。

14、Java中interrupted 和 isInterrupted方法的区别？

interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。

当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。

而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有可能被其它线程调用中断来改变。

15、Java中synchronized 和 ReentrantLock 有什么不同？

相似点：

这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的。

区别：

这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成。

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：

1. 等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。

2. 公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

3. 锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象。

16、有三个线程T1,T2,T3,如何保证顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

实际上先启动三个线程中哪一个都行，

因为在每个线程的run方法中用join方法限定了三个线程的执行顺序

```
public class JoinTest2 {  
    // 1.现在有T1、T2、T3三个线程，你怎样保证T2在T1执行完后执行， T3在T2执行完后执行
```

```

public static void main(String[] args) {
    final Thread t1 = new Thread(new Runnable() {
        @Override
        public void run() {
            System.out.println("t1");
        }
    });
    final Thread t2 = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                // 引用t1线程，等待t1线程执行完
                t1.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("t2");
        }
    });
    Thread t3 = new Thread(new Runnable() {
        @Override
        public void run() {
            try {
                // 引用t2线程，等待t2线程执行完
                t2.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("t3");
        }
    });
    t3.start(); //这里三个线程的启动顺序可以任意，大家可以试下！
    t2.start();
    t1.start();
}
}

```

17、SynchronizedMap和ConcurrentHashMap有什么区别？

SynchronizedMap()和Hashtable一样，实现上在调用map所有方法时，都对整个map进行同步。而 ConcurrentHashMap的实现却更加精细，它对map中的所有桶加了锁。所以，只要有一个线程访问map，其他线程就无法进入map，而如果一个线程在访问ConcurrentHashMap某个桶时，其他线程，仍然可以对map执行某些操作。

所以，ConcurrentHashMap在性能以及安全性方面，明显比Collections.synchronizedMap()更加有优势。同时，同步操作精确控制到桶，这样，即使在遍历map时，如果其他线程试图对map进行数据修改，也不会抛出ConcurrentModificationException。

18、什么是线程安全

线程安全就是说多线程访问同一代码，不会产生不确定的结果。

在多线程环境中，当各线程不共享数据的时候，即都是私有（private）成员，那么一定是线程安全的。但这种情况并不多见，在多数情况下需要共享数据，这时就需要进行适当的同步控制了。

线程安全一般都涉及到synchronized，就是一段代码同时只能有一个线程来操作不然中间过程可能会产生不可预知的结果。

如果你的代码所在的进程中多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的一样的，就是线程安全的。

19、Thread类中的yield方法有什么作用？

Yield方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃CPU占用而不能保证使其它线程一定能占用CPU，执行yield()的线程有可能在进入到暂停状态后马上又被执行。

20、Java线程池中submit() 和 execute()方法有什么区别？

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中，而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。

21、说一说自己对于synchronized关键字的了解

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。

另外，在 Java 早期版本中，`synchronized` 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。

如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的`synchronized` 效率低的原因。

庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对`synchronized` 较大优化，所以现在的`synchronized` 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

22、说说自己是怎么使用 `synchronized` 关键字，在项目中用到了吗`synchronized`关键字最主要三种使用方式

修饰实例方法：作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

修饰静态方法：也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（`static` 表明这是该类的一个静态资源，不管`new`了多少个对象，只有一份）。所以如果一个线程 A 调用一个实例对象的非静态`synchronized` 方法，而线程 B 需要调用这个实例对象所属类的静态`synchronized` 方法，是允许的，不会发生互斥现象，**因为访问静态`synchronized` 方法占用的锁是当前类的锁，而访问非静态`synchronized` 方法占用的锁是当前实例对象锁。**

修饰代码块：指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

总结：`synchronized` 关键字加到 static 静态方法和`synchronized(class)` 代码块上都是给 Class 类上锁。`synchronized` 关键字加到实例方法上是给对象实例上锁。尽量不要使用`synchronized(String a)` 因为 JVM 中，字符串常量池具有缓存功能

23、什么是线程安全？`Vector` 是一个线程安全类吗？

如果你的代码所在的进程中中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的一样的，就是线程安全的。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。`Vector` 是用同步方法来实现线程安全的，而和它相似的`ArrayList` 不是线程安全的。

24、`volatile`关键字的作用？

一旦一个共享变量（类的成员变量、类的静态成员变量）被`volatile` 修饰之后，那么就具备了两层语义：

保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

禁止进行指令重排序。

1. `volatile` 本质是在告诉 JVM 当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取；`synchronized` 则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。
2. `volatile` 仅能使用在变量级别；`synchronized` 则可以使用在变量、方法、和类级别的。
3. `volatile` 仅能实现变量的修改可见性，并不能保证原子性；`synchronized` 则可以保证变量的修改可见性和原子性。
4. `volatile` 不会造成线程的阻塞；`synchronized` 可能会造成线程的阻塞。

`volatile` 标记的变量不会被编译器优化；`synchronized` 标记的变量可以被编译器优化

25、简述一下你对线程池的理解

如果问到了这样的问题，可以展开的说一下线程池如何用、线程池的好处、线程池的启动策略）合理利用线程池能够带来三个好处。

第一：降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调度和监控。

26、线程生命周期(状态)

当线程被创建并启动以后，它既不是一启动就进入了执行状态，也不是一直处于执行状态。在线程的生命周期中，它要经过新建(New)、就绪(Runnable)、运行(Running)、阻塞(Blocked)和死亡(Dead)5 种状态。尤其是当线程启动以后，它不可能一直“霸占”着 CPU 独自运行，所以 CPU 需要在多条线程之间切换，于是线程状态也会多次在运行、阻塞之间切换。

27、新建状态 (NEW)

当程序使用`new` 关键字创建了一个线程之后，该线程就处于新建状态，此时仅由 JVM 为其分配内存，并初始化其成员变量的值。

28、就绪状态 (RUNNABLE)

当线程对象调用了`start()` 方法之后，该线程处于就绪状态。Java 虚拟机会为其创建方法调用栈和程序计数器，等待调度运行。

29、运行状态 (RUNNING)

如果处于就绪状态的线程获得了 CPU，开始执行`run()` 方法的线程执行体，则该线程处于运行状态。

30、阻塞状态 (BLOCKED)

阻塞状态是指线程因为某种原因放弃了 cpu 使用权，也即让出了 cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得 cpu timeslice 转到运行(running)状态。阻塞的情况分三种：

等待阻塞 (o.wait->等待对列) :

运行(running)的线程执行 o.wait()方法，JVM 会把该线程放入等待队列(waitting queue)中。

同步阻塞(lock->锁池)

运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则 JVM 会把该线程放入锁池(lock pool)中。

其他阻塞(sleep/join)

运行(running)的线程执行 Thread.sleep(long ms)或 t.join()方法，或者发出了 I/O 请求时，JVM 会把该线程置为阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入可运行(runnable)状态。

31、线程死亡 (DEAD)

线程会以下面三种方式结束，结束后就是死亡状态。

正常结束

1. run()或 call()方法执行完成，线程正常结束。异常结束
2. 线程抛出一个未捕获的 Exception 或 Error。调用 stop
3. 直接调用该线程的 stop()方法来结束该线程—该方法通常容易导致死锁，不推荐使用。

32、终止线程 4 种方式

正常运行结束

程序运行结束，线程自动结束。

使用退出标志退出线程

一般 run()方法执行完，线程就会正常结束，然而，常常有些线程是伺服线程。它们需要长时间的运行，只有在外部某些条件满足的情况下，才能关闭这些线程。使用一个变量来控制循环，例如：最直接的方法就是设一个 boolean 类型的标志，并通过设置这个标志为 true 或 false 来控制 while 循环是否退出，代码示例：

```
public class Threadsafe extends Thread {  
    public volatile boolean exit = false;  
    public void run() {  
        while (!exit){  
            //do something  
        }  
    }  
}
```

定义了一个退出标志 exit，当 exit 为 true 时，while 循环退出，exit 的默认值为 false。在定义 exit 时，使用了一个 Java 关键字 volatile，这个关键字的目的是使 exit 同步，也就是说在同一时刻只能由一个线程来修改 exit 的值。

Interrupt 方法结束线程

使用 interrupt()方法来中断线程有两种情况：

1. 线程处于阻塞状态：如使用了 sleep、同步锁的 wait、socket 中的 receiver、accept 等方法时，会使线程处于阻塞状态。当调用线程的 interrupt()方法时，会抛出 InterruptedException 异常。阻塞中的那个方法抛出这个异常，通过代码捕获该异常，然后 break 跳出循环状态，从而让我们有机会结束这个线程的执行。通常很多人认为只要调用 interrupt 方法线程就会结束，实际上是错的，一定要先捕获 InterruptedException 异常之后通过 break 来跳出循环，才能正常结束 run 方法。

2. 线程未处于阻塞状态：使用 isInterrupted() 判断线程的中断标志来退出循环。当使用 interrupt() 方法时，中断标志就会置 true，和使用自定义的标志来控制循环是一样的道理。

```
public class Threadsafe extends Thread {  
    public void run() {  
        while (!isInterrupted()){//非阻塞过程中通过判断中断标志来退出  
            try{  
                Thread.sleep(5*1000); //阻塞过程捕获中断异常来退出  
            }catch(InterruptedException e){  
                e.printStackTrace();  
                break;//捕获到异常之后，执行 break 跳出循环  
            }  
        }  
    }  
}
```

stop 方法终止线程 (线程不安全)

程序中可以直接使用 thread.stop() 来强行终止线程，但是 stop 方法是很危险的，就象突然关闭计算机电源，而不是按正常程序关机一样，可能会产生不可预料的结果，不安全主要是：thread.stop() 调用之后，创建子线程的线程就会抛出 ThreadDeatherror 的错误，并且会释放子线程所持有的所有锁。一般任何进行加锁的代码块，都是为了保护数据的一致性，如果在调用 thread.stop() 后导致了该线程所持有的所有锁的突然释放(不可控制)，那么被保护数据就有可能呈现不一致性，其他线程在使用这些被破坏的数据时，有可能导致一些很奇怪的应用程序错误。因

此，并不推荐使用 stop 方法来终止线程。

33、start 与 run 区别

1. start() 方法来启动线程，真正实现了多线程运行。这时无需等待 run 方法体代码执行完毕，可以直接继续执行下面的代码。
2. 通过调用 Thread 类的 start()方法来启动一个线程，这时此线程是处于就绪状态，并没有运行。
3. 方法 run()称为线程体，它包含了要执行的这个线程的内容，线程就进入了运行状态，开始运行 run 函数当中的代码。Run 方法运行结束，此线程终止。然后 CPU 再调度其它线程。

34、JAVA 后台线程

1. 定义：守护线程-也称“服务线程”，他是后台线程，它有一个特性，即为用户线程提供公共服务，在没有用户线程可服务时会自动离开。
2. 优先级：守护线程的优先级比较低，用于为系统中的其它对象和线程提供服务。
3. 设置：通过 setDaemon(true) 来设置线程为“守护线程”；将一个用户线程设置为守护线程的方式是在线程对象创建之前用线程对象的 setDaemon 方法。
4. 在 Daemon 线程中产生的新线程也是 Daemon 的。
5. 线程则是 JVM 级别的，以 Tomcat 为例，如果你在 Web 应用中启动一个线程，这个线程的生命周期并不会和 Web 应用程序保持同步。也就是说，即使你停止了 Web 应用，这个线程依旧是活跃的。
6. example: 垃圾回收线程就是一个经典的守护线程，当我们的程序中不再有任何运行的 Thread，程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是 JVM 上仅剩的线程时，垃圾回收线程会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源。
7. 生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统“同生共死”。当 JVM 中所有的线程都是守护线程的时候，JVM 就可以退出了；如果还有一个或以上的非守护线程则 JVM 不会退出

35、什么是乐观锁

乐观锁是一种乐观思想，即认为读多写少，遇到并发写的可能性低，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，采取在写时先读出当前版本号，然后加锁操作（比较跟上一次的版本号，如果一样则更新），如果失败则要重复读-比较-写的操作。

java 中的乐观锁基本都是通过 CAS 操作实现的，CAS 是一种更新的原子操作，比较当前值跟传入值是否一样，一样则更新，否则失败。

36、什么是悲观锁

悲观锁是就是悲观思想，即认为写多，遇到并发写的可能性高，每次去拿数据的时候都认为别人会修改，所以每次在读写数据的时候都会上锁，这样别人想读写这个数据就会 block 直到拿到锁。java 中的悲观锁就是 Synchronized, AQS 框架下的锁则是先尝试 cas 乐观锁去获取锁，获取不到，才会转换为悲观锁，如 ReentrantLock。

37、什么是自旋锁

自旋锁原理非常简单，如果持有锁的线程能在很短时间内释放锁资源，那么那些等待竞争锁的线程就不需要做内核态和用户态之间的切换进入阻塞挂起状态，它们只需要等一等（自旋），等持有锁的线程释放锁后即可立即获取锁，这样就避免用户线程和内核的切换的消耗。

线程自旋是需要消耗 CPU 的，说白了就是让 CPU 在做无用功，如果一直获取不到锁，那线程也不能一直占用 CPU 自旋做无用功，所以需要设定一个自旋等待的最大时间。

如果持有锁的线程执行的时间超过自旋等待的最大时间仍没有释放锁，就会导致其它争用锁的线程在最大等待时间内还是获取不到锁，这时争用线程会停止自旋进入阻塞状态。

自旋锁的优缺点

自旋锁尽可能的减少线程的阻塞，这对于锁的竞争不激烈，且占用锁时间非常短的代码块来说性能能大幅度的提升，因为自旋的消耗会小于线程阻塞挂起再唤醒的操作的消耗，这些操作会导致线程发生两次上下文切换！

但是如果锁的竞争激烈，或者持有锁的线程需要长时间占用锁执行同步块，这时候就不适合使用自旋锁了，因为自旋锁在获取锁前一直都是占用 CPU 做无用功，占着 XX 不 XX，同时有大量线程在竞争一个锁，会导致获取锁的时间很长，线程自旋的消耗大于线程阻塞挂起操作的消耗，其它需要 CPU 的线程又不能获取到 CPU，造成 CPU 的浪费。所以这种情况下我们要关闭自旋锁；

自旋锁时间阈值（1.6 引入了适应性自旋锁）

自旋锁的目的是为了占着 CPU 的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用 CPU 资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM 对于自旋周期的选择，jdk1.5 这个限度是一定的写死的，在 1.6 引入了适应性自旋锁，适应性自旋锁意味着自旋的时间不再是固定的了，而是由前一次在同一个锁上的自旋时间以及锁的拥有者的状态来决定，基本认为一个线程上下文切换的时间是最佳的一个时间，同时 JVM 还针对当前 CPU 的负载情况做了较多的优化，如果平均负载小于 CPUs 则一直自旋，如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞，如果正在自旋的线程发现 Owner 发生了变化则延迟自旋时间（自旋计数）或进入阻塞，如果 CPU 处于节电模式则停止自旋，自旋时间的最坏情况是 CPU 的存储延迟（CPU A 存储了一个数据，到 CPU B 得知这个数据直接的时间差），自旋时会适当放弃线程优先级之间的差异。

自旋锁的开启

JDK1.6 中-XX:+UseSpinning 开启；
-XX:PreBlockSpin=10 为自旋次数；
JDK1.7 后，去掉此参数，由 jvm 控制；

38、Synchronized 同步锁

synchronized 它可以把任意一个非 NULL 的对象当作锁。他属于独占式的悲观锁，同时属于可重入锁。 Synchronized 作用范围**

1. 作用于方法时，锁住的是对象的实例(this)；
2. 当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带PermGen (jdk1.8 则是 metaspace)，永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；
3. synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

Synchronized 核心组件

- 1) Wait Set：哪些调用 wait 方法被阻塞的线程被放置在这里；
- 2) Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中；
- 3) Entry List：Contention List 中那些有资格成为候选资源的线程被移动到 Entry List 中；
- 4) OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为 OnDeck；
- 5) Owner：当前已经获取到所资源的线程被称为 Owner；
- 6) !Owner：当前释放锁的线程。

Synchronized 实现

1. JVM 每次从队列的尾部取出一个数据用于锁竞争候选者 (OnDeck)，但是并发情况下，ContentionList 会被大量的并发线程进行 CAS 访问，为了降低对尾部元素的竞争，JVM 会将一部分线程移动到 EntryList 中作为候选竞争线程。
 2. Owner 线程会在 unlock 时，将 ContentionList 中的部分线程迁移到 EntryList 中，并指定 EntryList 中的某个线程为 OnDeck 线程 (一般是最先进去的那个线程)。
 3. Owner 线程并不直接把锁传递给 OnDeck 线程，而是把锁竞争的权利交给 OnDeck，OnDeck 需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在 JVM 中，也把这种选择行为称之为“竞争切换”。
 4. OnDeck 线程获取到锁资源后会变为 Owner 线程，而没有得到锁资源的仍然停留在 EntryList 中。如果 Owner 线程被 wait 方法阻塞，则转移到 WaitSet 队列中，直到某个时刻通过 notify 或者 notifyAll 唤醒，会重新进入 EntryList 中。
 5. 处于 ContentionList、EntryList、WaitSet 中的线程都处于阻塞状态，该阻塞是由操作系统来完成的 (Linux 内核下采用 pthread_mutex_lock 内核函数实现的)。
 6. Synchronized 是非公平锁。Synchronized 在线程进入 ContentionList 时，等待的线程会先尝试自旋获取锁，如果获取不到就进入 ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占 OnDeck 线程的锁资源。
- 参考：https://blog.csdn.net/zqz_zqz/article/details/70233767
7. 每个对象都有个 monitor 对象，加锁就是在竞争 monitor 对象，代码块加锁是在前后分别加上 monitorenter 和 monitorexit 指令来实现的，方法加锁是通过一个标记位来判断的
 8. synchronized 是一个重量级操作，需要调用操作系统相关接口，性能是低效的，有可能给线程加锁消耗的时间比有用操作消耗的时间更多。
 9. Java1.6，synchronized 进行了很多优化，有适应自旋、锁消除、锁粗化、轻量级锁及偏向锁等，效率有了本质上的提高。在之后推出的 Java1.7 与 1.8 中，均对该关键字的实现机理做了优化。引入了偏向锁和轻量级锁。都是在对象头中有标记位，不需要经过操作系统加锁。
 10. 锁可以从偏向锁升级到轻量级锁，再升级到重量级锁。这种升级过程叫做锁膨胀；
 11. JDK 1.6 中默认是开启偏向锁和轻量级锁，可以通过-XX:+UseBiasedLocking 来禁用偏向锁。

39、ReentrantLock

ReentrantLock 继承接口 Lock 并实现了接口中定义的方法，他是一种可重入锁，除了能完成 synchronized 所能完成的所有工作外，还提供了诸如可响应中断锁、可轮询锁请求、定时锁等避免多线程死锁的方法。

Lock 接口的主要方法

void lock(): 执行此方法时，如果锁处于空闲状态，当前线程将获取到锁。相反，如果锁已经被其他线程持有，将禁用当前线程，直到当前线程获取到锁。

boolean tryLock(): 如果锁可用，则获取锁，并立即返回 true，否则返回 false. 该方法和 lock() 的区别在于，tryLock() 只是“试图”获取锁，如果锁不可用，不会导致当前线程被禁用，当前线程仍然继续往下执行代码。而 lock() 方法则一定要获取到锁，如果锁不可用，就一直等待，在未获得锁之前，当前线程并不继续向下执行。

void unlock(): 执行此方法时，当前线程将释放持有的锁。锁只能由持有者释放，如果线程并不持有锁，却执行该方法，可能导致异常的发生。

Condition newCondition(): 条件对象，获取等待通知组件。该组件和当前的锁绑定，当前线程只有获取了锁，才能调用该组件的 await() 方法，而调用后，当前线程将释放锁。

getHoldCount(): 查询当前线程保持此锁的次数，也就是执行此线程执行 lock 方法的次数。

getQueueLength(): 返回正等待获取此锁的线程估计数，比如启动 10 个线程，1 个线程获得锁，此时返回的是 9

getWaitQueueLength : (Condition condition) 返回等待与此锁相关的给定条件的线程估计数。比如 10 个线程，用同一个 condition 对象，并且此时这 10 个线程都执行了 condition 对象的 await 方法，那么此时执行此方法返回 10

hasWaiters(Condition condition) : 查询是否有线程等待与此锁有关的给定条件 (condition)，对于指定 condition 对象，有多少线程执行了 condition.await 方法

hasQueuedThread(Thread thread) : 查询给定线程是否等待获取此锁

hasQueuedThreads() : 是否有线程等待此锁

isFair() : 该锁是否公平锁

isHeldByCurrentThread() : 当前线程是否保持锁锁定，线程的执行 lock 方法的前后分别是 false 和 true

isLock() : 此锁是否有任意线程占用

lockInterruptibly () : 如果当前线程未被中断，获取锁

tryLock () : 尝试获得锁，仅在调用时锁未被线程占用，获得锁

tryLock(long timeout TimeUnit unit) : 如果锁在给定等待时间内没有被另一个线程保持，则获取该锁。

不公平锁

JVM 按随机、就近原则分配锁的机制则称为不公平锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式，默认为不公平锁。不公平锁实际执行的效率要远远超出公平锁，除非程序有特殊需要，否则最常用不公平锁的分配机制。

公平锁

公平锁指的是锁的分配机制是公平的，通常先对锁提出获取请求的线程会先被分配到锁，ReentrantLock 在构造函数中提供了是否公平锁的初始化方式来定义公平锁。

40、Condition 类和 Object 类锁方法区别区别

1. Condition 类的 await 方法和 Object 类的 wait 方法等效
2. Condition 类的 signal 方法和 Object 类的 notify 方法等效
3. Condition 类的 signalAll 方法和 Object 类的 notifyAll 方法等效
4. ReentrantLock 类可以唤醒指定条件的线程，而 object 的唤醒是随机的

41、tryLock 和 lock 和 lockInterruptibly 的区别

1. tryLock 能获得锁就返回 true，不能就立即返回 false，tryLock(long timeout,TimeUnit unit)，可以增加时间限制，如果超过该时间段还没获得锁，返回 false
2. lock 能获得锁就返回 true，不能的话一直等待获得锁
3. lock 和 lockInterruptibly，如果两个线程分别执行这两个方法，但此时中断这两个线程，lock 不会抛出异常，而 lockInterruptibly 会抛出异常。

42、Semaphore 信号量

Semaphore 是一种基于计数的信号量。它可以设定一个阈值，基于此，多个线程竞争获取许可信号，做完自己的申请后归还，超过阈值后，线程申请许可信号将会被阻塞。Semaphore 可以用来构建一些对象池，资源池之类的，比如数据库连接池

实现互斥锁（计数器为 1）

我们也可以创建计数为 1 的 Semaphore，将其作为一种类似互斥锁的机制，这也叫二元信号量，表示两种互斥状态。

代码实现

```
// 创建一个计数阈值为 5 的信号量对象
// 只能 5 个线程同时访问
Semaphore semp = new Semaphore(5);
try { // 申请许可
    semp.acquire();
    try {
        // 业务逻辑
    } catch (Exception e) {
    } finally {
        // 释放许可
        semp.release();
    }
} catch (InterruptedException e) {
}
```

43、Semaphore 与 ReentrantLock 区别

Semaphore 基本能完成 ReentrantLock 的所有工作，使用方法也与之类似，通过 acquire() 与 release() 方法来获得和释放临界资源。经实测，Semaphore.acquire() 方法默认为可响应中断锁，与 ReentrantLock.lockInterruptibly() 作用效果一致，也就是说在等待临界资源的过程中可以被 Thread.interrupt() 方法中断。

此外，Semaphore 也实现了可轮询的锁请求与定时锁的功能，除了方法名 tryAcquire 与 tryLock 不同，其使用方法与 ReentrantLock 几乎一致。Semaphore 也提供了公平与非公平锁的机制，也可在构造函数中进行设定。

Semaphore 的锁释放操作也由手动进行，因此与 ReentrantLock 一样，为避免线程因抛出异常而无法正常释放锁的情况发生，释放锁的操作也必须在 finally 代码块中完成。

44、可重入锁（递归锁）

本文里面讲的是广义上的可重入锁，而不是单指 JAVA 下的 ReentrantLock。可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后，内层递归函数仍然有获取该锁的代码，但不受影响。在 JAVA 环境下 ReentrantLock 和 synchronized 都是可重入锁。

45、公平锁与非公平锁

公平锁（Fair）

加锁前检查是否有排队等待的线程，优先排队等待的线程，先来先得

非公平锁（Nonfair）

加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待

1. 非公平锁性能比公平锁高 5~10 倍，因为公平锁需要在多核的情况下维护一个队列
2. Java 中的 synchronized 是非公平锁，ReentrantLock 默认的 lock() 方法采用的是非公平锁。

46、ReadWriteLock 读写锁

为了提高性能，Java 提供了读写锁，在读的地方使用读锁，在写的地方使用写锁，灵活控制，如果没有写锁的情况下，读是无阻塞的，在一定程度上提高了程序的执行效率。读写锁分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由 jvm 自己控制的，你只要上好相应的锁即可。

读锁

如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁

写锁

如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！

Java 中读写锁有个接口 java.util.concurrent.locks.ReadWriteLock，也有具体的实现 ReentrantReadWriteLock。

47、共享锁和独占锁

java 并发包提供的加锁模式分为独占锁和共享锁。

独占锁

独占锁模式下，每次只能有一个线程能持有锁，ReentrantLock 就是以独占方式实现的互斥锁。

独占锁是一种悲观保守的加锁策略，它避免了读/读冲突，如果某个只读线程获取锁，则其他读线程都只能等待，这种情况下就限制了不必要的并发性，因为读操作并不会影响数据的一致性。

共享锁

共享锁则允许多个线程同时获取锁，并发访问 共享资源，如：ReadWriteLock。共享锁则是一种乐观锁，它放宽了加锁策略，允许多个执行读操作的线程同时访问共享资源。

1. AQS 的内部类 Node 定义了两个常量 SHARED 和 EXCLUSIVE，他们分别标识 AQS 队列中等待线程的锁获取模式。
2. Java 的并发包中提供了 ReadWriteLock，读-写锁。它允许一个资源可以被多个读操作访问，或者被一个写操作访问，但两者不能同时进行。

48、重量级锁（Mutex Lock）

Synchronized 是通过对对象内部的一个叫做监视器锁（monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的 Mutex Lock 来实现的。

而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么 Synchronized 效率低的原因。

因此，这种依赖于操作系统 Mutex Lock 所实现的锁我们称之为“重量级锁”。JDK 中对 Synchronized 做的种种优化，其核心都是为了减少这种重量级锁的使用。

JDK1.6 以后，为了减少获得锁和释放锁所带来的性能消耗，提高性能，引入了“轻量级锁”和“偏向锁”。

49、轻量级锁

锁的状态总共有四种：无锁状态、偏向锁、轻量级锁和重量级锁。

锁升级

随着锁的竞争，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁（但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级）。

“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。

在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

50、偏向锁

Hotspot 的作者经过以往的研究发现大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得。偏向锁的目的是在某个线程获得锁之后，消除这个线程锁重入（CAS）的开销，看起来让这个线程得到了偏袒。

引入偏向锁是为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为轻量级锁的获取及释放依赖多次 CAS 原子指令，而偏向锁只需要在置换 ThreadID 的时候依赖一次 CAS 原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的 CAS 原子指令的性能消耗）。

上面说过，轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

51、分段锁

分段锁也并非一种实际的锁，而是一种思想 ConcurrentHashMap 是学习分段锁的最好实践。

52、锁优化

减少锁持有时间

只用在有线程安全要求的程序上加锁

减小锁粒度

将大对象（这个对象可能会被很多线程访问），拆成小对象，大大增加并行度，降低锁竞争。

降低了锁的竞争，偏向锁，轻量级锁成功率才会提高。最典型的减小锁粒度的案例就是 ConcurrentHashMap。

锁分离

最常见的锁分离就是读写锁 ReadWriteLock，根据功能进行分离成读锁和写锁，这样读读不互斥，读写互斥，写写互斥，即保证了线程安全，又提高了性能，具体也请查看[高并发 Java 五]JDK 并发包 1。读写分离思想可以延伸，只要操作互不影响，锁就可以分离。比如 LinkedBlockingQueue 从头部取出，从尾部放数据。

锁粗化

通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽量短，即在使用完公共资源后，应该立即释放锁。但是，凡事都有一个度，如果对同一个锁不停的进行请求、同步和释放，其本身也会消耗系统宝贵的资源，反而不利于性能的优化。

锁消除

锁消除是在编译器级别的事情。在即时编译器时，如果发现不可能被共享的对象，则可以消除这些对象的锁操作，多数是因为程序员编码不规范引起。

参考：<https://www.jianshu.com/p/39628e1180a9>

53、线程基本方法

线程相关的基本方法有 wait，notify，notifyAll，sleep，join，yield 等。

54、线程等待 (wait)

调用该方法的线程进入 WAITING 状态，只有等待另外线程的通知或被中断才会返回，需要注意的是调用 wait()方法后，会释放对象的锁。因此，wait 方法一般用在同步方法或同步代码块中。

55、线程睡眠 (sleep)

sleep 导致当前线程休眠，与 wait 方法不同的是 sleep 不会释放当前占有的锁。sleep(long)会导致线程进入 TIMED-WATING 状态，而 wait()方法会导致当前线程进入 WAITING 状态。

56、线程让步 (yield)

yield 会使当前线程让出 CPU 执行时间片，与其他线程一起重新竞争 CPU 时间片。一般情况下，优先级高的线程有更大的可能性成功竞争得到 CPU 时间片，但这又不是绝对的，有的操作系统对线程优先级并不敏感。

57、线程中断 (interrupt)

中断一个线程，其本意是给这个线程一个通知信号，会影响这个线程内部的一个中断标识位。这个线程本身并不会因此而改变状态(如阻塞，终止等)。

1. 调用 interrupt()方法并不会中断一个正在运行的线程。也就是说处于 Running 状态的线程并不会因为被中断而被终止，仅仅改变了内部维护的中断标识位而已。
2. 若调用 sleep()而使线程处于 TIMED-WATING 状态，这时调用 interrupt()方法，会抛出 InterruptedException，从而使线程提前结束 TIMED-WATING 状态。
3. 许多声明抛出 InterruptedException 的方法(如 Thread.sleep(long mills 方法))，抛出异常前，都会清除中断标识位，所以抛出异常后，调用 isInterrupted()方法将会返回 false。
4. 中断状态是线程固有的一个标识位，可以通过此标识位安全的终止线程。比如，你想终止一个线程 thread 的时候，可以调用 thread.interrupt()方法，在线程的 run 方法内部可以根据 thread.isInterrupted()的值来优雅的终止线程。

58、Join 等待其他线程终止

join() 方法，等待其他线程终止，在当前线程中调用一个线程的 join() 方法，则当前线程转为阻塞状态，回到另一个线程结束，当前线程再由阻塞状态变为就绪状态，等待 cpu 的宠幸。

59、为什么要用 join()方法？

很多情况下，主线程生成并启动了子线程，需要用到子线程返回的结果，也就是需要主线程需要在子线程结束后再结束，这时候就要用到 join() 方法。

```
System.out.println(Thread.currentThread().getName() + "线程运行开始!");
Thread6 thread1 = new Thread6();
thread1.setName("线程 B");
thread1.join();
System.out.println("这时 thread1 执行完毕之后才能执行主线程");
```

60、线程唤醒 (notify)

Object 类中的 notify() 方法，唤醒在此对象监视器上等待的单个线程，如果所有线程都在此对象上等待，则会选择唤醒其中一个线程，选择是任意的，并在对实现做出决定时发生，线程通过调用其中一个 wait() 方法，在对象的监视器上等待，直到当前的线程放弃此对象上的锁定，才能继续执行被唤醒的线程，被唤醒的线程将以常规方式与在该对象上主动同步的其他所有线程进行竞争。类似的方法还有 notifyAll()，唤醒再次监视器上等待的所有线程。

61、线程其他方法

1. sleep()：强迫一个线程睡眠N毫秒。
2. isAlive()：判断一个线程是否存活。
3. join()：等待线程终止。
4. activeCount()：程序中活跃的线程数。
5. enumerate()：枚举程序中的线程。
6. currentThread()：得到当前线程。
7. isDaemon()：一个线程是否为守护线程。
8. setDaemon()：设置一个线程为守护线程。(用户线程和守护线程的区别在于，是否等待主线程依赖于主线程结束而结束)
9. setName()：为线程设置一个名称。
10. wait()：强迫一个线程等待。
 11.notify()：通知一个线程继续运行。
11. setPriority()：设置一个线程的优先级。
12. getPriority(): 获得一个线程的优先级。

62、进程

(有时候也称做任务)是指一个程序运行的实例。在 Linux 系统中，线程就是能并行运行并且与他们的父进程(创建他们的进程)共享同一地址空间(一段内存区域)和其他资源的轻量级的进程。

63、上下文

是指某一时间点 CPU 寄存器和程序计数器的内容。

64、寄存器

是 CPU 内部的数量较少但是速度很快的内存（与之对应的是 CPU 外部相对较慢的 RAM 主内存）。寄存器通过对常用值（通常是运算的中间值）的快速访问来提高计算机程序运行的速度。

65、程序计数器

是一个专用的寄存器，用于表明指令序列中 CPU 正在执行的位置，存的值为正在执行的指令的位置或者下一个将要被执行的指令的位置，具体依赖于特定的系统。

66、PCB-“切换帧”

上下文切换可以认为是内核（操作系统的内核）在 CPU 上对于进程（包括线程）进行切换，上下文切换过程中的信息是保存在进程控制块（PCB, process control block）中的。PCB 还经常被称作“切换帧”（switchframe）。信息会一直保存到 CPU 的内存中，直到他们被再次使用。

67、上下文切换的活动

1. 挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处。
2. 在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复。
3. 跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程在程序中。

68、引起线程上下文切换的原因

1. 当前执行任务的时间片用完之后，系统 CPU 正常调度下一个任务；
2. 当前执行任务碰到 IO 阻塞，调度器将此任务挂起，继续下一任务；
3. 多个任务抢占锁资源，当前任务没有抢到锁资源，被调度器挂起，继续下一任务；
4. 用户代码挂起当前任务，让出 CPU 时间；
5. 硬件中断；

69、同步锁

当多个线程同时访问同一个数据时，很容易出现问题。为了避免这种情况出现，我们要保证线程同步互斥，就是指并发执行的多个线程，在同一时间内只允许一个线程访问共享数据。Java 中可以使用 synchronized 关键字来取得一个对象的同步锁。

70、死锁

何为死锁，就是多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。

71、线程池原理

线程池做的工作主要是控制运行的线程的数量，处理过程中将任务放入队列，然后在线程创建后启动这些任务，如果线程数量超过了最大数量超出数量的线程排队等候，等其它线程执行完毕，再从队列中取出任务来执行。他的主要特点为：线程复用；控制最大并发数；管理线程。

72、线程复

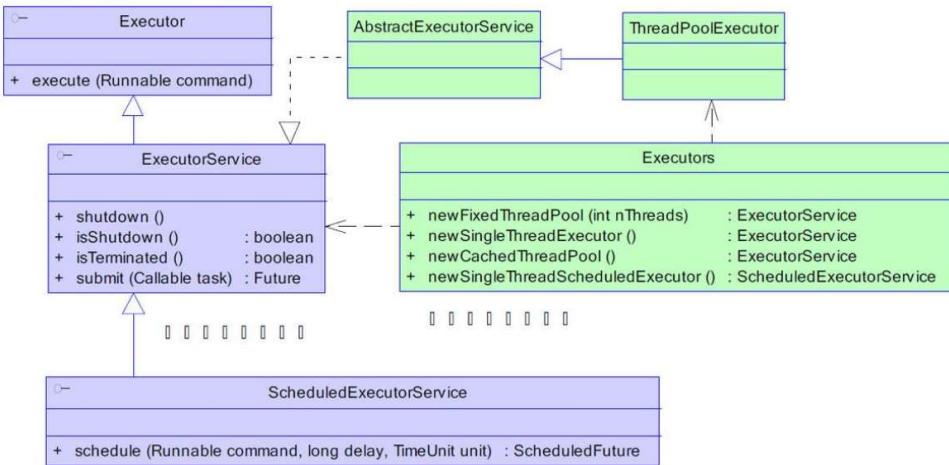
每一个 Thread 的类都有一个 start 方法。当调用 start 启动线程时 Java 虚拟机会调用该类的 run 方法。那么该类的 run() 方法中就是调用了 Runnable 对象的 run() 方法。我们可以继承重写 Thread 类，在其 start 方法中添加不断循环调用传递过来的 Runnable 对象。这就是线程池的实现原理。循环方法中不断获取 Runnable 是用 Queue 实现的，在获取下一个 Runnable 之前可以是阻塞的。

73、线程池的组成

一般的线程池主要分为以下 4 个组成部分：

1. 线程池管理器：用于创建并管理线程池
2. 工作线程：线程池中的线程
3. 任务接口：每个任务必须实现的接口，用于工作线程调度其运行
4. 任务队列：用于存放待处理的任务，提供一种缓冲机制

Java 中的线程池是通过 Executor 框架实现的，该框架中用到了 Executor , Executors , ExecutorService , ThreadPoolExecutor , Callable 和 Future、FutureTask 这几个类。



ThreadPoolExecutor 的构造方法如下：

```

public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime,
    TimeUnit unit, BlockingQueue<Runnable> workQueue) {
    this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,
        Executors.defaultThreadFactory(), defaultHandler);
}
  
```

1. corePoolSize : 指定了线程池中的线程数量。
2. maximumPoolSize : 指定了线程池中的最大线程数量。
3. keepAliveTime : 当前线程池数量超过 corePoolSize 时，多余的空闲线程的存活时间，即多次时间内会被销毁。
4. unit : keepAliveTime 的单位。
5. workQueue : 任务队列，被提交但尚未被执行的任务。
6. threadFactory : 线程工厂，用于创建线程，一般用默认的即可。
7. handler : 拒绝策略，当任务太多来不及处理，如何拒绝任务。

74、拒绝策略

线程池中的线程已经用完了，无法继续为新任务服务，同时，等待队列也已经排满了，再也塞不下新任务了。这时候我们就需要拒绝策略机制合理的处理这个问题。
JDK 内置的拒绝策略如下：

1. AbortPolicy : 直接抛出异常，阻止系统正常运行。
2. CallerRunsPolicy : 只要线程池未关闭，该策略直接在调用者线程中，运行当前被丢弃的任务。显然这样做不会真的丢弃任务，但是，任务提交线程的性能极有可能会急剧下降。
3. DiscardOldestPolicy : 丢弃最老的一个请求，也就是即将被执行的一个任务，并尝试再次提交当前任务。
4. DiscardPolicy : 该策略默默地丢弃无法处理的任务，不予任何处理。如果允许任务丢失，这是最好的一种方案。
以上内置拒绝策略均实现了 RejectedExecutionHandler 接口，若以上策略仍无法满足实际需要，完全可以自己扩展 RejectedExecutionHandler 接口。

75、Java 线程池工作过程

1. 线程池刚创建时，里面没有一个线程。任务队列是作为参数传进来的。不过，就算队列里面有任务，线程池也不会马上执行它们。
2. 当调用 execute() 方法添加一个任务时，线程池会做如下判断：
 - a) 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；
 - b) 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；
 - c) 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；
 - d) 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常 RejectExecutionException。
3. 当一个线程完成任务时，它会从队列中取下一个任务来执行。
4. 当一个线程无事可做，超过一定的时间 (keepAliveTime) 时，线程池会判断，如果当前运行的线程数大于 corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到 corePoolSize 的大小。

76、JAVA 阻塞队列原理

阻塞队列，关键字是阻塞，先理解阻塞的含义，在阻塞队列中，线程阻塞有这样两种情况：

1. 当队列中没有数据的情况下，消费者端的所有线程都会被自动阻塞（挂起），直到有数据放入队列。
2. 当队列中填满数据的情况下，生产者端的所有线程都会被自动阻塞（挂起），直到队列中有空的位置，线程被自动唤醒。

阻塞队列的主要方法：

方法类型	抛出异常	特殊值	阻塞	超时
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除	remove()	poll()	take()	poll(time,unit)
检查	element()	peek()	不可用	不可用

抛出异常：抛出一个异常；

特殊值：返回一个特殊值（null 或 false，视情况而定）

阻塞：在成功操作之前，一直阻塞线程

超时：放弃前只在最大的时间内阻塞

插入操作

1：public abstract boolean add(E paramE)：将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。如果该元素是 NULL，则会抛出 NullPointerException 异常。

2：public abstract boolean offer(E paramE)：将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则返回 false。

3：public abstract void put(E paramE) throws InterruptedException：将指定元素插入此队列中，将等待可用的空间（如果有必要）

```
public void put(E paramE) throws InterruptedException {  
    checkNotNull(paramE);  
    ReentrantLock localReentrantLock = this.lock;  
    localReentrantLock.lockInterruptibly();  
    try {  
        while (this.count == this.items.length)  
            this.notFull.await(); //如果队列满了，则线程阻塞等待  
        enqueue(paramE);  
        localReentrantLock.unlock();  
    } finally {  
        localReentrantLock.unlock();  
    }  
}
```

4：offer(E o, long timeout, TimeUnit unit)：可以设定等待的时间，如果在指定的时间内，还不能往队列中加入 BlockingQueue，则返回失败。

获取数据操作：

1：poll(time)：取走 BlockingQueue 里排在首位的对象，若不能立即取出，则可以等 time 参数规定的时间，取不到时返回 null；

2：poll(long timeout, TimeUnit unit)：从 BlockingQueue 取出一个队首的对象，如果在指定时间内，队列一旦有数据可取，则立即返回队列中的数据。否则直到时间超时还没有数据可取，返回失败。

3：take()：取走 BlockingQueue 里排在首位的对象，若 BlockingQueue 为空，阻塞进入等待状态直到 BlockingQueue 有新的数据被加入。

4.drainTo():一次性从 BlockingQueue 获取所有可用的数据对象（还可以指定获取数据的个数），通过该方法，可以提升获取数据效率；不需要多次分批加锁或释放锁。

77、Java 中的阻塞队列

1. ArrayBlockingQueue：由数组结构组成的有界阻塞队列。
2. LinkedBlockingQueue：由链表结构组成的有界阻塞队列。
3. PriorityBlockingQueue：支持优先级排序的无界阻塞队列。
4. DelayQueue：使用优先级队列实现的无界阻塞队列。
5. SynchronousQueue：不存储元素的阻塞队列。
6. LinkedTransferQueue：由链表结构组成的无界阻塞队列。
7. LinkedBlockingDeque：由链表结构组成的双向阻塞队列

