目录

[问题： 3](#_Toc526367988)

[一、 成员变量的创建 3](#_Toc526367989)

[二、 Java基础 3](#_Toc526367990)

[1. Static 3](#_Toc526367991)

[2. Final 4](#_Toc526367992)

[3. Transient 4](#_Toc526367993)

[4. Volatile 4](#_Toc526367994)

[5. 并发编程的三大特性 5](#_Toc526367995)

[6. Interface与abstract类的区别 5](#_Toc526367996)

[7. 堆和栈 6](#_Toc526367997)

[8. 数据类型的默认值和长度 6](#_Toc526367998)

[A. 默认值 6](#_Toc526367999)

[B. 长度 6](#_Toc526368000)

[9. 补码 6](#_Toc526368001)

[三、 多线程 7](#_Toc526368002)

[1. 状态 7](#_Toc526368003)

[2. 创建线程的方法 9](#_Toc526368004)

[A. 通过实现 Runnable 接口 9](#_Toc526368005)

[B. 通过继承 Thread 类本身 10](#_Toc526368006)

[C. 通过 Callable 和 Future 创建线程。 13](#_Toc526368007)

[D. 创建线程的三种方式的对比 14](#_Toc526368008)

[E. 线程池 14](#_Toc526368009)

[a) 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。（是什么） 14](#_Toc526368010)

[b) 那么，我们为什么需要用到线程池呢？每次用的时候手动创建不行吗？ 14](#_Toc526368011)

[c) 线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。 15](#_Toc526368012)

[d) 这里介绍两种使用线程池创建线程的方法 15](#_Toc526368013)

[3. 线程的优先级 17](#_Toc526368014)

[4. JAVA多线程和并发基础面试问答 17](#_Toc526368015)

[1) 进程和线程之间有什么不同？ 17](#_Toc526368016)

[2) 多线程编程的好处是什么？ 18](#_Toc526368017)

[3) 用户线程和守护线程有什么区别？ 18](#_Toc526368018)

[4) 我们如何创建一个线程？ 18](#_Toc526368019)

[5) 有哪些不同的线程生命周期？ 18](#_Toc526368020)

[6) 可以直接调用Thread类的run()方法么？ 19](#_Toc526368021)

[7) 如何让正在运行的线程暂停一段时间？ 19](#_Toc526368022)

[8) 你对线程优先级的理解是什么？ 19](#_Toc526368023)

[9) 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？ 19](#_Toc526368024)

[10) 在多线程中，什么是上下文切换(context-switching)？ 20](#_Toc526368025)

[11) 你如何确保main()方法所在的线程是Java程序最后结束的线程？ 20](#_Toc526368026)

[12) 线程之间是如何通信的？ 20](#_Toc526368027)

[13) 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？ 20](#_Toc526368028)

[14) 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？ 21](#_Toc526368029)

[15) 为什么Thread类的sleep()和yield()方法是静态的？ 21](#_Toc526368030)

[16) 如何确保线程安全？ 21](#_Toc526368031)

[17) volatile关键字在Java中有什么作用？ 21](#_Toc526368032)

[18) 同步方法和同步块，哪个是更好的选择？ 22](#_Toc526368033)

[19) 如何创建守护线程？ 22](#_Toc526368034)

[20) 什么是ThreadLocal? 22](#_Toc526368035)

[21) 什么是Thread Group？为什么不建议使用它？ 22](#_Toc526368036)

[22) 什么是Java线程转储(Thread Dump)，如何得到它？ 23](#_Toc526368037)

[23) 什么是死锁(Deadlock)？如何分析和避免死锁？ 23](#_Toc526368038)

[24) 什么是Java Timer类？如何创建一个有特定时间间隔的任务？ 24](#_Toc526368039)

[25) 什么是线程池？如何创建一个Java线程池？ 24](#_Toc526368040)

[5. Java并发面试问题 24](#_Toc526368041)

[1) 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？ 24](#_Toc526368042)

[2) Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？ 25](#_Toc526368043)

[3) 什么是Executors框架？ 25](#_Toc526368044)

[4) 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？ 26](#_Toc526368045)

[5) 什么是Callable和Future? 26](#_Toc526368046)

[6) 什么是FutureTask? 27](#_Toc526368047)

[7) 什么是并发容器的实现？ 27](#_Toc526368048)

[8) Executors类是什么？ 27](#_Toc526368049)

[9) 为什么说ConcurrentHashMap是弱一致性的？以及为何多个线程并发修改ConcurrentHashMap时不会报ConcurrentModificationException？ 28](#_Toc526368050)

[四、 异常 28](#_Toc526368051)

[1. 异常类型 28](#_Toc526368052)

[2. finally关键字 28](#_Toc526368053)

[3. Java内置异常 29](#_Toc526368054)

[4. 异常方法 30](#_Toc526368055)

[5. 使用throws抛出异常 31](#_Toc526368056)

[6. 使用throw抛出异常 32](#_Toc526368057)

[Java7增强的throw语句： 32](#_Toc526368058)

[7. 自定义异常类 33](#_Toc526368059)

[8. catch和throw同时使用 33](#_Toc526368060)

[9. 异常链 34](#_Toc526368061)

[10. Java的异常跟踪栈 35](#_Toc526368062)

[11. 异常处理规则 35](#_Toc526368063)

[五、 设计模式 35](#_Toc526368064)

[1. 单例模式 Singleton 35](#_Toc526368065)

[2. 简单工厂模式 StaticFactory Method 36](#_Toc526368066)

[3. 工厂模式 Factory Method 36](#_Toc526368067)

[4. 抽象工厂模式 Abstract Factory Method 36](#_Toc526368068)

[5. 代理模式 Proxy Method 36](#_Toc526368069)

[6. 命令模式 Command Method 36](#_Toc526368070)

[7. 策略模式 Strategy Method 36](#_Toc526368071)

[8. 观察者模式 Observer Method 37](#_Toc526368072)

[9. 生产者消费者模式 ProducerConsumer Method 37](#_Toc526368073)

## 问题：

Lambds表达式是什么，为什么要有；

Jdk7和jdk8的字符串常量池分别在哪；

水杯

谈一下对Thread

Java面向对象的理解

Static,final,线程、进程间通信，

Interface、abstract，设计模式

## Java基础

### 成员变量的创建

静态成员变量在类装载的时候就进行了创建，在整个程序结束时按序销毁。

实例变量在类实例化对象时候创建，在对象销毁的时候销毁。

局部变量在局部范围使用时创建，跳出局部范围销毁。

### 代码块

#### 普通代码块

在方法或语句中出现的{}就称为普通代码块。普通代码块和一般语句的执行顺序由他们在代码中出现的次序决定，先出现先执行。

public class Test {  
 public static void main(String[] args) {  
 {  
 int x = 3;  
 System.*out*.println("普通代码块内的变量x=" + x);  
 }  
 int x = 1;  
 System.*out*.println("主方法内的变量x=" + x);  
 {  
 int y = 7;  
 System.*out*.println("普通代码块内的变量y=" + y);  
 }  
 }  
}  
/\*  
 \* 运行结果 普通代码块内的变量x=3  
 \* 主方法内的变量x=1  
 \* 普通代码块内的变量y=7  
 \*/

#### 构造代码块

直接在类中定义且没有加static关键字的代码块称为{}构造代码块。构造代码块在创建对象时被调用，每次创建对象都会被调用，并且构造代码块的执行次序优先于类构造函数。如果存在多个构造代码块，执行顺序由他们在代码中出现的次序决定，先出现先执行。

public class Test1 {  
 {  
 System.*out*.println("第一构造块");  
 }  
  
 public Test1(int i) {  
 System.*out*.println("第" + i + "次调用" + "构造方法");  
 }  
  
 {  
 System.*out*.println("第二构造块");  
 }  
  
 public static void main(String[] args) {  
 new Test1(0);  
 new Test1(1);  
 new Test1(2);  
  
 }  
}  
  
/\*  
 \* 执行结果 第一构造块  
 \* 第二构造块  
 \* 第0次调用构造方法  
 \* 第一构造块  
 \* 第二构造块  
 \* 第1次调用构造方法  
 \* 第一构造块  
 \* 第二构造块  
 \* 第2次调用构造方法  
 \*/

#### 静态代码块:

在java中使用static关键字声明的代码块。静态块用于初始化类，为类的属性初始化。每个静态代码块只会执行一次。由于JVM在加载类时会执行静态代码块，所以静态代码块先于主方法执行。  
如果类中包含多个静态代码块，那么将按照"先定义的代码先执行，后定义的代码后执行"。  
      注意：1 .静态代码块不能存在于任何方法体内。

2 .静态代码块不能直接访问实例变量和实例方法，需要通过类的实例对象来访问。

public class Test3 {  
 public static String *STATIC\_FIELD* = "静态属性";  
  
 // 静态块  
 static {  
 System.*out*.println(*STATIC\_FIELD*);  
 System.*out*.println("静态代码块1");  
 }  
  
 public String field = "非静态属性";  
  
 // 非静态块  
 {  
 System.*out*.println(field);  
 System.*out*.println("非静态代码块2");  
 }  
  
 public InitOderTest() {  
 System.*out*.println("无参构造函数");  
 }  
  
 public static void main(String[] args) {  
 InitOderTest test = new InitOderTest();  
 }  
  
 // 非静态块  
 {  
 System.*out*.println(field);  
 System.*out*.println("非静态代码块1");  
 }  
 // 静态块  
 static {  
 System.*out*.println(*STATIC\_FIELD*);  
 System.*out*.println("静态代码块2");  
 }  
  
}  
/\*  
 \* 运行结果 静态属性  
 \* 静态代码块1  
 \* 静态属性  
 \* 静态代码块2  
 \* 非静态属性  
 \* 非静态代码块2  
 \* 非静态属性  
 \* 非静态代码块1  
 \* 无参构造函数  
 \*/

--------------------- 本文来自 krisll 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/u012804721/article/details/52439311?utm\_source=copy

### Static

静态变量只能在类主体中定义，不能在方法中定义

成员方法又称为实例方法

静态方法又称为类方法

类方法是指类中被static修饰的方法，无this指针。

类方法是可以调用其他类的static方法的。

* 1. 方便在没有创建对象的情况下来进行调用（方法/变量）。
  2. 在静态方法中不能访问非静态成员方法和非静态成员变量，但是在非静态成员方法中是可以访问静态成员方法/变量的。举个简单的例子：
  3. 静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。
  4. static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。
  5. 类属性中被static所引用的变量，会被作为GC的root根节点。作为根节点就意味着，这一类变量是基本上不会被回收的。因此，static很容易引入内存泄漏的风险。

### Final

* 1. 当用final修饰一个类时，表明这个类不能被继承。
  2. 使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；
  3. 对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。
  4. 必须在定义时或者构造器中进行初始化赋值，而且final变量一旦被初始化赋值之后，就不能再被赋值了。

### Transient

1. 一旦变量被transient修饰，变量将不再是对象持久化的一部分，该变量内容在序列化后无法获得访问。
2. transient关键字只能修饰变量，而不能修饰方法和类。注意，本地变量是不能被transient关键字修饰的。变量如果是用户自定义类变量，则该类需要实现Serializable接口。
3. 被transient关键字修饰的变量不再能被序列化，一个静态变量不管是否被transient修饰，均不能被序列化。

### Volatile

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。
    2. 禁止进行指令重排序。
    3. 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；
    4. 在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

### 并发编程的三大特性

1. 原子性

在Java中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。

1. 可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

1. 有序性

　　在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”（具体原理在下一节讲述）。另外可以通过synchronized和Lock来保证有序性.

### Interface与abstract类的区别

含有abstract修饰符的class即为抽象类，abstract 类不能创建类的实例对象。含有abstract方法的类必须定义为abstract class，abstract class类中的方法不必是抽象的, 但是有抽象方法的类必定是抽象类。abstract class类中定义抽象方法必须在具体(Concrete)子类中实现，所以，不能有抽象构造方法或抽象静态方法。如果的子类没有实现抽象父类中的所有抽象方法，那么子类也必须定义为abstract类型。

接口（interface）可以说成是抽象类的一种特例，接口中的所有方法都必须是抽象的。接口中的方法定义默认为public abstract类型，接口中的成员变量类型默认为public static final。

下面比较一下两者的语法区别：

1. 抽象类可以有构造方法，接口中不能有构造方法。
2. 抽象类中可以有普通成员变量，接口中没有普通成员变量
3. 抽象类中可以包含非抽象的普通方法，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。
4. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。
5. 抽象类中可以包含静态方法，接口中不能包含静态方法
6. 抽象类和接口中都可以包含静态成员变量，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。
7. 一个类可以实现多个接口，但只能继承一个抽象类。

### 堆、栈、方法区

JAVA的JVM的内存可分为3个区：堆(heap)、堆栈(stack)和方法区(method)

* 堆区:
  1. 提供所有类实例和数组对象存储区域
  2. jvm只有一个堆区(heap)被所有线程共享，堆中不存放基本类型和对象引用，只存放对象本身
* 栈区:
  1. 每个线程包含一个栈区，栈中只保存基础数据类型的对象和自定义对象的引用(不是对象)，对象都存放在堆区中
  2. 每个栈中的数据(原始类型和对象引用)都是私有的，其他栈不能访问。
* 方法区:
  1. 又叫静态区，跟堆一样，被所有的线程共享。方法区包含所有的class和static变量。
  2. 方法区中包含的都是在整个程序中永远唯一的元素，如class，static变量。
  3. 运行时常量池都分配在 Java 虚拟机的方法区之中

--------------------- 本文来自 zhuliyuan丶 的CSDN 博客 ，全文地址请点击：<https://blog.csdn.net/zly921112/article/details/61192747?utm_source=copy>

对堆和栈的补充：

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。

堆内存用于存放由new创建的对象和数组。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组＆对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，数组和对象在没有引用变量指向它的时候（比如先前的引用变量x=null时），才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因。

以上段落来自于某一本Java程序设计的书中，实际上，栈中的变量指向堆内存中的变量，这就是Java中的指针。

总结起来就是对象存储在堆内存，引用变量存储在栈内存。栈内存指向堆内存。

### 数据类型的默认值和长度

#### 数据类型分类

基本数据类型（或叫做原生类、内置类型）8种：

整数：byte，short，int，long（默认是int类型）

浮点类型： float，double（默认是double类型， 在定义 float 类型时必须在数字后面跟上 F 或者 f，long同理）

字符类型：char

布尔类型：boolean

引用数据类型3种：数组，类，接口

Java中的byte，short，char进行计算时都会提升为int类型。

#### 默认值

1. 布尔型(boolean)变量默认值为false，byte、short、int、long为0，字符型为'\u0000'(空字符)，浮点型(float double)为0.0,引用类型(String)为null。
2. 注意：未初始化的局部变量是不可以使用的，在这里可以认为是因为局部变量没有默认值，所以不可以直接使用。空字符('\u0000')什么也不输出，不要认为输出是空格。

#### 长度

32位的系统，则一个字是4个字节，如果是64位，则是8个字节。

1字节（byte） = 8位（bit）

基本类型：byte 二进制位数：8

包装类：java.lang.Byte

最小值：Byte.MIN\_VALUE=-128

最大值：Byte.MAX\_VALUE=127

基本类型：short 二进制位数：16

包装类：java.lang.Short

最小值：Short.MIN\_VALUE=-32768

最大值：Short.MAX\_VALUE=32767

基本类型：int 二进制位数：32

包装类：java.lang.Integer

最小值：Integer.MIN\_VALUE=-2147483648

最大值：Integer.MAX\_VALUE=2147483647

基本类型：long 二进制位数：64

包装类：java.lang.Long

最小值：Long.MIN\_VALUE=-9223372036854775808

最大值：Long.MAX\_VALUE=9223372036854775807

基本类型：float 二进制位数：32

包装类：java.lang.Float

最小值：Float.MIN\_VALUE=1.4E-45

最大值：Float.MAX\_VALUE=3.4028235E38

基本类型：double 二进制位数：64

包装类：java.lang.Double

最小值：Double.MIN\_VALUE=4.9E-324

最大值：Double.MAX\_VALUE=1.7976931348623157E308

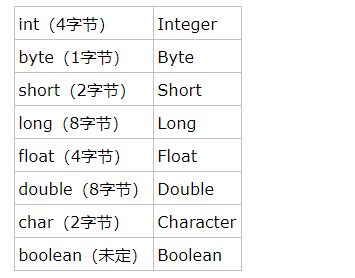
基本类型：char 二进制位数：16

包装类：java.lang.Character

最小值：Character.MIN\_VALUE=0

最大值：Character.MAX\_VALUE=65535

#### 包装器类型

装箱过程是通过调用包装器的valueOf方法实现的，而拆箱过程是通过调用包装器的 xxxValue方法实现的。（xxx代表对应的基本数据类型）。

主动：Integer i = new Integer(10); //装箱

int i = a.intValue();//拆箱

自动：Integer i = 10; //装箱

int n = i; //拆箱

### 补码

1. 求**负整数的补码**，将其对应正数二进制表示所有位取反（包括符号位，0变1，1变0）后加1，-5对应正数5（00000101）→所有位取反（11111010）→加1(11111011)。

如果补码的符号位为“0”，表示是一个正数，其原码就是补码。

如果补码的符号位为“1”，表示是一个负数，那么求给定的这个补码的补码就是要求的原码。

小数补码求法，符号位保持1不变，数值位从右边数第一个1及其右边的0保持不变，左边按位取反。-0.0101为负数，补码为1.1011。

1. **补码加法**

[X+Y]补 = [X]补 + [Y]补

【例】X=+0110011,Y=-0101001，求[X+Y]补

[X]补=00110011 [Y]补=11010111

[X+Y]补 = [X]补 + [Y]补 = 00110011+11010111=00001010

注：因为计算机中运算器的位长是固定的（定长运算），上述运算中产生的最高位进位将丢掉，所以结果不是100001010，而是00001010。

1. **补码减法**

[X-Y]补 = [X]补 - [Y]补 = [X]补 + [-Y]补。

【例】1-1 [十进制]

1的原码00000001 转换成补码：00000001

-1的原码10000001 转换成补码：11111111

1+（-1)=0

00000001+11111111=00000000

00000000转换成十进制为0

0=0所以运算正确。

【例】-7-（-10) [十进制]

改为加法形式：-7-（-10）=-7+（-（-10））

-7的补码：11111001

-（-10）的补码：-10的原码为10001010，-（-10）的原码为00001010，

-（-10）的补码就是其原码，为00001010

-7 - （-10)= -7 + 10 = 3

11111001+00001010 = 00000011

转换成十进制为3

## 多线程

### 状态

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

下图显示了一个线程完整的生命周期。



* **新建状态:**

使用 **new** 关键字和 **Thread** 类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序 **start()** 这个线程。

* **就绪状态:**

当线程对象调用了start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待JVM里线程调度器的调度。

* **运行状态:**

如果就绪状态的线程获取 CPU 资源，就可以执行 **run()**，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

* **阻塞状态:**

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以重新进入就绪状态。可以分为三种：

* + 等待阻塞：运行状态中的线程执行 wait() 方法，使线程进入到等待阻塞状态。
  + 同步阻塞：线程在获取 synchronized 同步锁失败(因为同步锁被其他线程占用)。
  + 其他阻塞：通过调用线程的 sleep() 或 join() 发出了 I/O 请求时，线程就会进入到阻塞状态。当sleep() 状态超时，join() 等待线程终止或超时，或者 I/O 处理完毕，线程重新转入就绪状态。
* **死亡状态:**

一个运行状态的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

### 创建线程的方法

**Java 提供了三种创建线程的方法：**

#### 通过实现 Runnable 接口

创建一个线程，最简单的方法是创建一个实现 Runnable 接口的类。

为了实现 Runnable，一个类只需要执行一个方法调用 run()，声明如下：

public void run()

你可以重写该方法，重要的是理解的 run() 可以调用其他方法，使用其他类，并声明变量，就像主线程一样。

在创建一个实现 Runnable 接口的类之后，你可以在类中实例化一个线程对象。

Thread 定义了几个构造方法，下面的这个是我们经常使用的：

Thread(Runnable threadOb,String threadName);

新线程创建之后，你调用它的 start() 方法它才会运行。

void start();

class TestForRunnable implements Runnable {  
 private int k;  
 //多线程,implements Runnable  
  
 */\*\*  
 \*   
 \*/* public void run() {  
 for (; k < 100; k++) {  
 System.*out*.println(Thread.*currentThread*().getName()  
 + " " + k);  
 }  
 }  
 */\*\*  
 \*   
 \*/* public void test() {  
 //Runnable  
 for (int i = 0; i < 100; i++) {  
 System.*out*.println(Thread.*currentThread*().getName() + " " + i);  
 if (20 == i) {  
 try {  
 //共享k  
 TestForRunnable a = new TestForRunnable();  
// new Thread(a, "新线程1").start();  
// new Thread(a, "新线程2").start();  
 Thread t1 = new Thread(a, "新线程1");  
 Thread t2 = new Thread(a, "新线程2");  
 t2.start();  
 t1.start();  
 t1.join();  
 t2.*sleep*(1);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

#### 通过继承 Thread 类本身

创建一个线程的第二种方法是创建一个新的类，该类继承 Thread 类，然后创建一个该类的实例。

继承类必须重写 run() 方法，该方法是新线程的入口点。它也必须调用 start() 方法才能执行。

该方法尽管被列为一种多线程实现方式，但是本质上也是实现了 Runnable 接口的一个实例。

class TestForThread extends Thread {  
 private int k;  
 */\*\*  
 \*  
 \** ***@param*** *name  
 \*/* public TestForThread(String name) {  
 super(name);  
 }  
  
 */\*\*  
 \*  
 \*/* public void run() {  
 try {  
 for (; k < 100; k++) {  
 System.*out*.println(getName() + " " + k);  
 //有sleep()抛出InterruptedException  
 *sleep*(10);  
 if (20 == k) {  
 //转入就绪状态，给同级或者更高等级线程机会  
 Thread.*yield*();  
 }  
 }  
  
 }catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }

**下表列出了Thread类的一些重要方法：**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public void start()** 使该线程开始执行；**Java** 虚拟机调用该线程的 run 方法。 |
| 2 | **public void run()** 如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。 |
| 3 | **public final void setName(String name)** 改变线程名称，使之与参数 name 相同。 |
| 4 | **public final void setPriority(int priority)**  更改线程的优先级。 |
| 5 | **public final void setDaemon(boolean on)** 将该线程标记为守护线程或用户线程。 |
| 6 | **public final void join(long millisec)** 等待该线程终止的时间最长为 millis 毫秒。 |
| 7 | **public void interrupt()** 中断线程。 |
| 8 | **public final boolean isAlive()** 测试线程是否处于活动状态。 |

**测试线程是否处于活动状态。 上述方法是被Thread对象调用的。下面的方法是Thread类的静态方法。**

|  |  |
| --- | --- |
| **序号** | **方法描述** |
| 1 | **public static void yield()** 暂停当前正在执行的线程对象，并执行其他线程。 |
| 2 | **public static void sleep(long millisec)** 在指定的毫秒数内让当前正在执行的线程休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。 |
| 3 | **public static boolean holdsLock(Object x)** 当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。 |
| 4 | **public static Thread currentThread()** 返回对当前正在执行的线程对象的引用。 |
| 5 | **public static void dumpStack()** 将当前线程的堆栈跟踪打印至标准错误流。 |

#### 通过 Callable 和 Future 创建线程。

* 1. 创建 Callable 接口的实现类，并实现 call() 方法，该 call() 方法将作为线程执行体，并且有返回值。
* 2. 创建 Callable 实现类的实例，使用 FutureTask 类来包装 Callable 对象，该 FutureTask 对象封装了该 Callable 对象的 call() 方法的返回值。
* 3. 使用 FutureTask 对象作为 Thread 对象的 target 创建并启动新线程。
* 4. 调用 FutureTask 对象的 get() 方法来获得子线程执行结束后的返回值。
* class CallableAndFutureTask {  
   */\*\*  
   \*   
   \*/* public void test() {  
   //多线程Callable，FutureTask  
   FutureTask<Integer> task = new FutureTask<Integer>((Callable<Integer>)()-> {  
   //call()  
   int i = 0;  
   for (; i < 100; i++) {  
   System.*out*.println(Thread.*currentThread*().getName() + "的循环变量i的值：" + i);  
   }  
   return i;  
   });  
   for (int i = 0; i<100; i++) {  
   System.*out*.println(Thread.*currentThread*().getName() + "的循环变量i的值：" + i);  
   if (20 == i) {  
   new Thread(task,"有返回值的线程").start();  
   }  
   }  
   try {  
   System.*out*.println("子线程的返回值：" + task.get());  
   } catch (Exception e) {  
   e.printStackTrace();  
   }  
   }  
  }

#### 创建线程的三种方式的对比

* 1. 采用实现 Runnable、Callable 接口的方式创建多线程时，线程类只是实现了 Runnable 接口或 Callable 接口，还可以继承其他类。
* 2. 使用继承 Thread 类的方式创建多线程时，编写简单，如果需要访问当前线程，则无需使用 Thread.currentThread() 方法，直接使用 this 即可获得当前线程。

#### 线程池

##### 线程池，其实就是一个容纳多个线程的容器，其中的线程可以反复使用，省去了频繁创建线程对象的操作，无需反复创建线程而消耗过多资源。（是什么）

##### 那么，我们为什么需要用到线程池呢？每次用的时候手动创建不行吗？

在java中，如果每个请求到达就创建一个新线程，开销是相当大的。在实际使用中，创建和销毁线程花费的时间和消耗的系统资源都相当大，甚至可能要比在处理实际的用户请求的时间和资源要多的多。除了创建和销毁线程的开销之外，活动的线程也需要消耗系统资源。如果在一个jvm里创建太多的线程，可能会使系统由于过度消耗内存或“切换过度”而导致系统资源不足。为了防止资源不足，需要采取一些办法来限制任何给定时刻处理的请求数目，尽可能减少创建和销毁线程的次数，特别是一些资源耗费比较大的线程的创建和销毁，尽量利用已有对象来进行服务。（为什么）

线程池主要用来解决线程生命周期开销问题和资源不足问题。通过对多个任务重复使用线程，线程创建的开销就被分摊到了多个任务上了，而且由于在请求到达时线程已经存在，所以消除了线程创建所带来的延迟。这样，就可以立即为请求服务，使用应用程序响应更快；另外，通过适当的调整线程中的线程数目可以防止出现资源不足的情况。（什么用）

##### 线程池都是通过线程池工厂创建，再调用线程池中的方法获取线程，再通过线程去执行任务方法。

* **Executors**：线程池创建工厂类
* **public static ExecutorServicenewFixedThreadPool(int nThreads)**：返回线程池对象
* **ExecutorService**：线程池类
* **Future<?> submit(Runnable task)**：获取线程池中的某一个线程对象，并执行
* **Future 接口**：用来记录线程任务执行完毕后产生的结果。线程池创建与使用

##### 这里介绍两种使用线程池创建线程的方法

###### **使用Runnable接口创建线程池**

使用线程池中线程对象的步骤：

* 1、创建线程池对象
* 2、创建 Runnable 接口子类对象
* 3、提交 Runnable 接口子类对象
* 4、关闭线程池

**Test.java** 代码如下：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test {

public static void main(String[] args) {

//创建线程池对象 参数5，代表有5个线程的线程池

ExecutorService service = Executors.newFixedThreadPool(5);

//创建Runnable线程任务对象

TaskRunnable task = new TaskRunnable();

//从线程池中获取线程对象

service.submit(task);

System.out.println("----------------------");

//再获取一个线程对象

service.submit(task);

//关闭线程池

service.shutdown();

}

}

TaskRunnable.java 接口文件如下：

public class TaskRunnable implements Runnable{

@Override

public void run() {

for (int i = 0; i < 1000; i++) {

System.out.println("自定义线程任务在执行"+i);

}

}

}

###### 使用Callable接口创建线程池

Callable接口：与Runnable接口功能相似，用来指定线程的任务。其中的call()方法，用来返回线程任务执行完毕后的结果，call方法可抛出异常。

**ExecutorService**：线程池类

**<T> Future<T> submit(Callable<T> task)**：获取线程池中的某一个线程对象，并执行线程中的 call() 方法

**Future 接口**：用来记录线程任务执行完毕后产生的结果。线程池创建与使用

使用线程池中线程对象的步骤：

* 1、创建线程池对象
* 2、创建 Callable 接口子类对象
* 3、提交 Callable 接口子类对象
* 4、关闭线程池

Test.java 代码如下：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test{

public static void main(String[] args) {

ExecutorService service = Executors.newFixedThreadPool(3);

TaskCallable c = new TaskCallable();

//线程池中获取线程对象，调用run方法

service.submit(c);

//再获取一个

service.submit(c);

//关闭线程池

service.shutdown();

}

}

TaskCallable.java 接口文件如下：

import java.util.concurrent.Callable;

public class TaskCallable implements Callable<Object>{

@Override

public Object call() throws Exception {

for (int i = 0; i < 1000; i++) {

System.out.println("自定义线程任务在执行"+i);

}

return null;

}

}

### 线程的优先级

每一个 Java 线程都有一个优先级，这样有助于操作系统确定线程的调度顺序。

Java 线程的优先级是一个整数，其取值范围是 1 （Thread.MIN\_PRIORITY ） - 10 （Thread.MAX\_PRIORITY ）。

默认情况下，每一个线程都会分配一个优先级 NORM\_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

### JAVA多线程和并发基础面试问答

https://www.cnblogs.com/dolphin0520/p/3932934.html

#### 进程和线程之间有什么不同？

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。Java运行环境是一个包含了不同的类和程序的单一进程。线程可以被称为轻量级进程。线程需要较少的资源来创建和驻留在进程中，并且可以共享进程中的资源。

#### 多线程编程的好处是什么？

在多线程程序中，多个线程被并发的执行以提高程序的效率，CPU不会因为某个线程需要等待资源而进入空闲状态。多个线程共享堆内存(heap memory)，因此创建多个线程去执行一些任务会比创建多个进程更好。举个例子，Servlets比CGI更好，是因为Servlets支持多线程而CGI不支持。

#### 用户线程和守护线程有什么区别？

当我们在Java程序中创建一个线程，它就被称为用户线程。一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

#### 我们如何创建一个线程？

有两种创建线程的方法：一是实现Runnable接口，然后将它传递给Thread的构造函数，创建一个Thread对象；二是直接继承Thread类。若想了解更多可以阅读这篇关于如何在[Java中创建线程](http://www.journaldev.com/1016/java-thread-example-extending-thread-class-and-implementing-runnable-interface)的文章。

#### 有哪些不同的线程生命周期？

当我们在Java程序中新建一个线程时，它的状态是New。当我们调用线程的start()方法时，状态被改变为Runnable。线程调度器会为Runnable线程池中的线程分配CPU时间并且讲它们的状态改变为Running。其他的线程状态还有Waiting，Blocked 和Dead。读这篇文章可以了解更多关于[线程生命周期](http://www.journaldev.com/1044/life-cycle-of-thread-understanding-thread-states-in-java)的知识。

#### 可以直接调用Thread类的run()方法么？

当然可以，但是如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

#### 如何让正在运行的线程暂停一段时间？

我们可以使用Thread类的Sleep()方法让线程暂停一段时间。需要注意的是，这并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为Runnable，并且根据线程调度，它将得到执行。

#### 你对线程优先级的理解是什么？

每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

#### 什么是线程调度器(Thread Scheduler)和时间分片(Time Slicing)？

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。线程调度并不受到Java虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

#### 在多线程中，什么是上下文切换(context-switching)？

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

#### 你如何确保main()方法所在的线程是Java程序最后结束的线程？

我们可以使用Thread类的joint()方法来确保所有程序创建的线程在main()方法退出前结束。这里有一篇文章关于[Thread类的joint()方法](http://www.journaldev.com/1024/java-thread-join-example-with-explanation)。

#### 线程之间是如何通信的？

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。点击[这里](http://www.journaldev.com/1037/java-thread-wait-notify-and-notifyall-example)有更多关于线程wait, notify和notifyAll.

#### 为什么线程通信的方法wait(), notify()和notifyAll()被定义在Object类里？

Java的每个对象中都有一个锁(monitor，也可以成为监视器) 并且wait()，notify()等方法用于等待对象的锁或者通知其他线程对象的监视器可用。在Java的线程中并没有可供任何对象使用的锁和同步器。这就是为什么这些方法是Object类的一部分，这样Java的每一个类都有用于线程间通信的基本方法

#### 为什么wait(), notify()和notifyAll()必须在同步方法或者同步块中被调用？

当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

#### 为什么Thread类的sleep()和yield()方法是静态的？

Thread类的sleep()和yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

#### 如何确保线程安全？

在Java中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁，使用volatile关键字，使用不变类和线程安全类。在[线程安全教程](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples)中，你可以学到更多。

#### volatile关键字在Java中有什么作用？

当我们使用volatile关键字去修饰变量的时候，所以线程都会直接读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

#### 同步方法和同步块，哪个是更好的选择？

同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

#### 如何创建守护线程？

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

#### 什么是ThreadLocal?

ThreadLocal用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。

每个线程都会拥有他们自己的Thread变量，它们可以使用get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望它们同线程状态关联起来是private static属性。在[ThreadLocal例子](http://www.journaldev.com/1076/java-threadlocal-example-to-create-thread-local-variables)这篇文章中你可以看到一个关于ThreadLocal的小程序。

#### 什么是Thread Group？为什么不建议使用它？

ThreadGroup是一个类，它的目的是提供关于线程组的信息。

ThreadGroup API比较薄弱，它并没有比Thread提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaught exception handler)。但在Java 1.5中Thread类也添加了setUncaughtExceptionHandler(UncaughtExceptionHandler eh) 方法，所以ThreadGroup是已经过时的，不建议继续使用。

#### 什么是Java线程转储(Thread Dump)，如何得到它？

线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获取线程转储——使用Profiler，Kill -3命令，jstack工具等等。我更喜欢jstack工具，因为它容易使用并且是JDK自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。读这篇文档可以了解更多关于[产生线程转储](http://www.journaldev.com/1053/how-to-generate-thread-dump-in-java)的知识。

#### 什么是死锁(Deadlock)？如何分析和避免死锁？

死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

分析死锁，我们需要查看Java应用程序的线程转储。我们需要找出那些状态为BLOCKED的线程和他们等待的资源。每个资源都有一个唯一的id，用这个id我们可以找出哪些线程已经拥有了它的对象锁。

避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法，阅读这篇文章去学习[如何分析死锁](http://www.journaldev.com/1058/java-deadlock-example-and-how-to-analyze-deadlock-situation)。

#### 什么是Java Timer类？如何创建一个有特定时间间隔的任务？

java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

这里有关于[java Timer的例子](http://www.journaldev.com/1050/java-timer-and-timertask-example-tutorial)。

#### 什么是线程池？如何创建一个Java线程池？

一个线程池管理了一组工作线程，同时它还包括了一个用于放置等待执行的任务的队列。

java.util.concurrent.Executors提供了一个 java.util.concurrent.Executor接口的实现用于创建线程池。[线程池例子](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)展现了如何创建和使用线程池，或者阅读[ScheduledThreadPoolExecutor](http://www.journaldev.com/2340/java-scheduledthreadpoolexecutor-example-to-schedule-tasks-after-delay-and-execute-periodically)例子，了解如何创建一个周期任务。

### Java并发面试问题

#### 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用同步技术来做到这一点。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。可以阅读这篇文章来了解[Java的atomic类](http://www.journaldev.com/1095/java-atomic-operations-atomicinteger-example)。

#### Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

Lock接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

* 可以使锁更公平
* 可以使线程在等待锁的时候响应中断
* 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间
* 可以在不同的范围，以不同的顺序获取和释放锁

阅读更多[关于锁的例子](http://www.journaldev.com/2377/java-lock-example-and-concurrency-lock-vs-synchronized)

#### 什么是Executors框架？

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池，阅读这篇文章可以了解[如何使用Executor框架创建一个线程池](http://www.journaldev.com/1069/java-thread-pool-example-using-executors-and-threadpoolexecutor)。

#### 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

阅读这篇文章了解[如何使用阻塞队列实现生产者-消费者问题。](http://www.journaldev.com/1034/java-blockingqueue-example-implementing-producer-consumer-problem)

#### 什么是Callable和Future?

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

阅读这篇文章了解更多[关于Callable，Future的例子](http://www.journaldev.com/1090/java-callable-future-example)。

#### 什么是FutureTask?

FutureTask是Future的一个基础实现，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，单当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。阅读[Java FutureTask例子](http://www.journaldev.com/1650/java-futuretask-example-program)，学习如何使用它。

#### 什么是并发容器的实现？

Java集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的next()方法将抛出ConcurrentModificationException异常。

并发容器支持并发的遍历和并发的更新。

主要的类有ConcurrentHashMap, CopyOnWriteArrayList 和CopyOnWriteArraySet，阅读这篇文章了解[如何避免ConcurrentModificationException](http://www.journaldev.com/378/how-to-avoid-concurrentmodificationexception-when-using-an-iterator)。

#### Executors类是什么？

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法。

Executors可以用于方便的创建线程池

#### 为什么说ConcurrentHashMap是弱一致性的？以及为何多个线程并发修改ConcurrentHashMap时不会报ConcurrentModificationException？

　　参考：<http://ifeve.com/java-concurrent-hashmap-2/>

<http://ifeve.com/concurrenthashmap-weakly-consistent/>

<http://blog.csdn.net/liuzhengkang/article/details/2916620>

## 异常

### 异常类型

1. **unchecked exceptions（运行时异常）：**都是RuntimeException类及其子类异常，就是我们在开发中测试功能时程序终止，控制台出现的异常，比如：   
   NullPointerException(空指针异常)、   
   IndexOutOfBoundsException(下标越界异常)   
   ClassCastException(类转换异常)   
   ArrayStoreException(数据存储异常，操作数组时类型不一致)   
   IO操作的BufferOverflowException异常
2. **checked exceptions，非运行时异常 （编译异常）**：是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果不处理，程序就不能编译通过。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。   
   通俗的话说就是在写代码时出现红线，需要try catch或者throws时出现的异常。

### finally关键字

finally 关键字用来创建在 try 代码块后面执行的代码块。

无论是否发生异常，finally 代码块中的代码总会被执行。

在 finally 代码块中，可以运行清理类型等收尾善后性质的语句

### Java内置异常

Java 根据各个类库也定义了一些其他的异常，下面的表中列出了 Java 的非检查性异常。

|  |  |
| --- | --- |
| **异常** | **描述** |
| ArithmeticException | 当出现异常的运算条件时，抛出此异常。例如，一个整数"除以零"时，抛出此类的一个实例。 |
| ArrayIndexOutOfBoundsException | 用非法索引访问数组时抛出的异常。如果索引为负或大于等于数组大小，则该索引为非法索引。 |
| ArrayStoreException | 试图将错误类型的对象存储到一个对象数组时抛出的异常。 |
| ClassCastException | 当试图将对象强制转换为不是实例的子类时，抛出该异常。 |
| IllegalArgumentException | 抛出的异常表明向方法传递了一个不合法或不正确的参数。 |
| IllegalMonitorStateException | 抛出的异常表明某一线程已经试图等待对象的监视器，或者试图通知其他正在等待对象的监视器而本身没有指定监视器的线程。 |
| IllegalStateException | 在非法或不适当的时间调用方法时产生的信号。换句话说，即 Java 环境或 Java 应用程序没有处于请求操作所要求的适当状态下。 |
| IllegalThreadStateException | 线程没有处于请求操作所要求的适当状态时抛出的异常。 |
| IndexOutOfBoundsException | 指示某排序索引（例如对数组、字符串或向量的排序）超出范围时抛出。 |
| NegativeArraySizeException | 如果应用程序试图创建大小为负的数组，则抛出该异常。 |
| NullPointerException | 当应用程序试图在需要对象的地方使用 null 时，抛出该异常 |
| NumberFormatException | 当应用程序试图将字符串转换成一种数值类型，但该字符串不能转换为适当格式时，抛出该异常。 |
| SecurityException | 由安全管理器抛出的异常，指示存在安全侵犯。 |
| StringIndexOutOfBoundsException | 此异常由 String 方法抛出，指示索引或者为负，或者超出字符串的大小。 |
| UnsupportedOperationException | 当不支持请求的操作时，抛出该异常。 |

下面的表中列出了 Java 定义在 java.lang 包中的检查性异常类。

|  |  |
| --- | --- |
| **异常** | **描述** |
| ClassNotFoundException | 应用程序试图加载类时，找不到相应的类，抛出该异常。 |
| CloneNotSupportedException | 当调用 Object 类中的 clone 方法克隆对象，但该对象的类无法实现 Cloneable 接口时，抛出该异常。 |
| IllegalAccessException | 拒绝访问一个类的时候，抛出该异常。 |
| InstantiationException | 当试图使用 Class 类中的 newInstance 方法创建一个类的实例，而指定的类对象因为是一个接口或是一个抽象类而无法实例化时，抛出该异常。 |
| InterruptedException | 一个线程被另一个线程中断，抛出该异常。 |
| NoSuchFieldException | 请求的变量不存在 |
| NoSuchMethodException | 请求的方法不存在 |

### 异常方法

下面的列表是 Throwable 类的主要方法:

|  |  |
| --- | --- |
| **序号** | **方法及说明** |
| 1 | **public String getMessage()** 返回关于发生的异常的详细信息。这个消息在Throwable 类的构造函数中初始化了。 |
| 2 | **public Throwable getCause()** 返回一个Throwable 对象代表异常原因。 |
| 3 | **public String toString()** 使用getMessage()的结果返回类的串级名字。 |
| 4 | **public void printStackTrace()** 打印toString()结果和栈层次到System.err，即错误输出流。 |
| 5 | **public StackTraceElement [] getStackTrace()** 返回一个包含堆栈层次的数组。下标为0的元素代表栈顶，最后一个元素代表方法调用堆栈的栈底。 |
| 6 | **public Throwable fillInStackTrace()** 用当前的调用栈层次填充Throwable 对象栈层次，添加到栈层次任何先前信息中。 |

### 使用throws抛出异常

如果在当前方法不知道该如何处理该异常时，则可以使用throws对异常进行抛出给调用者处理或者交给JVM。JVM对异常的处理方式是：打印异常的跟踪栈信息并终止程序运行。   
throws在使用时应处于方法签名之后使用，可以抛出多种异常并用英文字符逗号’,’隔开   
e.g.1   
public void throwsTest() throws ExceptionClass1, ExceptionClass2 {...}   
如果抛出给调用者的异常是Checked异常，这种异常是我们需要处理以来提高程序健壮性的，一般抛出则要调用者做相应处理，要么调用者对该异常进行try…catch处理，要么再次throws交给上一层。这其中需要注意一点：子类方法声明抛出的异常类型应是父类方法声明抛出的异常类型的子类或相同，子类方法声明跑出的异常不允许比父类方法声明抛出的异常多。   
e.g.2

public class ThrowsTest2{

//因为test();会抛出IOException，main方法是调用者

//则需要使用try...catch进行捕获或者抛给JVM

//抛出时要遵循父类声明抛出“大于”子类声明抛出的原则

public static void main(String[] args) throws Exception {

test();

}

public static void test() throws IOException() {

//因为FileInputStream构造器会抛出IOException

//所以需要使用try...catch块进行处理或使用throws抛给调用者

FileInputStream fis = new FileInputStream("a.txt");

}

}

### 使用throw抛出异常

如果需要程序在程序中自行抛出异常，应该使用throw语句抛出，抛出的不是一个类而是一个对象且只能抛出一个对象。它可以单独使用，也可以结合catch块捕获使用。如果抛出的异常对象时Checked异常则处于try块里被catch捕获或者放在一个带throws的方法里；如果抛出的是RuntimeException则既可以显示使用try…catch捕获也可以不理会它   
e.g.3

public class ThrowTest {

public static void main(String[] args) {

try{

throwChecked(3);

}catch(Exception e) {

System.out.println(e.getMessage());

}

throwRuntime(-3);

}

//该方法内抛出一个Exception异常对象，必须捕获或抛给调用者

public static void throwChecked(int a) throws Exception {

if(a < 0) {

throw new Exception("a的值应大于0，不符合要求")

}

}

//该方法内抛出一个RuntimeException对象，可以不理会直接交给JVM处理

public static void throwRuntime(int a) {

if(a < 0) {

throw new RuntimeException("a的值应大于0，不符合要求")

}

}

}

Java7增强的throw语句：   
在Java7之前，父类和子类在声明抛出异常时应符合父类包含的异常“大于等于”子类包含的异常的规则；从Java7开始，Java编译器会检查throw语句抛出的异常的实际类型，如下代码中编译器知道throw e只能抛出FileNotFoundException 所以在方法签名上可以直接写该异常：   
e.g.4

public class ThrowTest2 {

public static void main(String[] args) throws FileNotFoundException {

try {

new FileOutputStream("a.txt");

}catch(Exception e) {

e.printStackTrace();

throw e;

}

}

}

### 自定义异常类

在抛出异常时，异常类名往往包含有用的信息，所以在选择抛出异常时需要选择适合的类，从而可以明确的描述该异常情况。这时候就需要我们自己定义异常，自定义异常一定是Throwable的子类，若是检查异常就要继承自Exception，若是运行时异常就要继承自RuntimeException   
e.g.5

public class AuctionException extends Exception {

//无参构造

public AuctionException() {}

//含参构造

//通过调用父类的构造器将字符串msg传给异常对象的massage属性，

//massage属性就是对异常的描述

public AuctionException(String msg) {

super(msg);

}

}

关于自定义异常深入学习可以参考以下链接：   
<http://www.cnblogs.com/AlanLee/p/6104492.html>

### catch和throw同时使用

在实际应用中，在异常出现的当前方法中，程序只能对异常做部分处理，还有些处理需要在该方法的调用者才能够完成，使用需要再次抛出异常。这时，就需要将catch和throw结合使用。   
e.g.6

public class AuctionTest {

private double initPrice = 30.0;

public void bid(String bidPrice) throws AuctionException {

double d = 0.0;

try{

d = Double.parseDouble(bidPrice);

}catch(Exception e) {

e.printStackTrace();

throw new AuctionException("竞拍价必须为数值，不能包含其它数值");

}

if(initPrice > d) {

throw new AuctionException("竞拍价应比起拍价高");

}

initPrice = d;

}

public static void main(String[] args) {

AuctionTest at = new AuctionTest();

try{

at.bid("..");

}catch(AuctionException ae) {

System.out.println(ae.getMessage());

}

}

}

以上代码的运行结果为：

竞拍价必须为数值，不能包含其它数值

java.lang.NumberFormatException: multiple points

at sun.misc.FloatingDecimal.readJavaFormatString(Unknown Source)

at sun.misc.FloatingDecimal.parseDouble(Unknown Source)

at java.lang.Double.parseDouble(Unknown Source)

at ExceptionTest.AuctionTest.bid(AuctionTest.java:15)

at ExceptionTest.AuctionTest.main(AuctionTest.java:28)

### 异常链

把捕获一个异常然后接着抛出另一个异常，并把原始异常信息保存下来是一种典型的链式处理，也被称作异常链。

### Java的异常跟踪栈

如e.g.6的运行结果   
除去第一行打印的自定义的提示信息之外的第一行是异常的类型和详细消息，之后的提示信息中含有确切代码行数的信息记录了所有的异常发生点，各行显示被调用方法中执行的停止位置。跟踪栈总是最内部的被调用方法逐渐上传知道外部业务操作的起点，通常就是main方法或者Thread的run方法。

### 异常处理规则

1. 不要过度使用异常：对于完全已知的错误应编写处理这种错误代码从而提高代码的健壮性，只有外部的、不能确定的和不可预知的运行时错误使是用异常，并且异常机制的效率低于正常的流程控制。
2. 不要使用过于庞大的try块：过于庞大的try块业务也相对更复杂，会导致try块中异常的可能性大大增加，在分析发生异常的原因时难度增加。
3. 避免使用Catch All语句：Catch All是catch(Throwable t)，也会在发生异常是分析原因的复杂度增加。
4. 不要忽略已捕获到的异常：对于捕获到的异常应该对其进行处理从而提高代码健壮性，而不是什么都不做或者只是打印跟踪栈信息。

--------------------- 本文来自 sdr\_zd 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/sdr\_zd/article/details/75675004?utm\_source=copy

## 设计模式

**【单例模式 工厂模式   策略模式  适配器模式  生产者消费者模式 观察者模式】**

### ****单例模式 Singleton****

**如果一个类始终只能创建一个实例，则这个类被称为单例类，这种模式就被称为单例模式。**

**一般建议单例模式的方法命名为：getInstance()，可以有参数，一般无参数。**

**优点：**

1)减少创建[Java](http://lib.csdn.net/base/javase)实例所带来的系统开销

2)便于系统跟踪单个Java实例的生命周期、实例状态等。

### ****简单工厂模式 StaticFactory Method****

**由一个工厂对象决定创建出哪一种产品类的实例。**

**优点：**

**让对象的调用者和对象创建过程分离，当对象调用者需要对象时，直接向工厂请求即可。**

### ****工厂模式 Factory Method****

**程序可以为不同产品类提供不同的工厂，不同的工厂类生产不同的产品（也就是一个工厂生产一个产品，通过选择不同的工厂获得不同的产品。）**

### ****抽象工厂模式 Abstract Factory Method****

**为了解决客户端代码与不同工厂类耦合的问题。在工厂类的基础上再增加一个工厂类，该工厂类不制造具体的被调用对象，而是制造不同工厂对象。（建立一个生产工厂的工厂，把原来工厂模式的工厂作为方法）**

### ****代理模式 Proxy Method****

**当客户端代码需要调用某个对象时，客户端实际上不关心是否准确得到该对象，它只要一个能提供该功能的对象即可，此时我们就可返回该对象的代理（Proxy）。**

### ****命令模式 Command Method****

**某个方法需要完成某一个功能，完成这个功能的大部分步骤已经确定了，但可能有少量具体步骤无法确定，必须等到执行该方法时才可以确定。在Java中，传入该方法的是一个对象，该对象通常是某个接口的匿名实现类的实例，该接口通常被称为命令接口，这种设计方式也被称为命令模式。**

### ****策略模式 Strategy Method****

**策略模式用于封装系列的**[算法](http://lib.csdn.net/base/datastructure)**，这些算法通常被封装在一个被称为Context的类中，客户端程序可以自由选择其中一种算法，或让Context为客户端选择一种最佳算法——使用策略模式的优势是为了支持算法的自由切换。**

**可以考虑使用配置文件来指定用户使用哪种具体算法——这就彻底分离客户端代码和具体算法策略。**

### ****观察者模式 Observer Method****

**观察者模式定义了对象间的一对多依赖关系，让一个或多个观察者对象观察一个主题对象。当主题对象的状态发生变化时，系统能通知所有的依赖于此对象的观察者对象，从而使得观察者对象能够自动更新。**

**在观察者模式中，被观察的对象常常也被称为目标或主题（Subject），依赖的对象被称为观察者（Observer）。**

### ****生产者消费者模式 ProducerConsumer Method****

**生产者和消费者在同一时间段内共用同一存储空间，生产者向空间（仓库/缓冲区）里生产数据，而消费者取走数据。**

**优点：**

1)解耦

2)支持并发

生产者与消费者是两个独立的并发体，他们之间是用缓冲区作为桥梁连接。

3)支持忙闲不均

当数据制造快的时候，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。 等生产者的制造速度慢下来，消费者再慢慢处理掉。

--------------------- 本文来自 cyhaolihai 的CSDN 博客 ，全文地址请点击：https://blog.csdn.net/cyhaolihai/article/details/82902368?utm\_source=copy