## &和&&的区别

&和&&都可以用作逻辑与的运算符，表示逻辑与（and），当运算符两边的表达式的结果都为true时，整个运算结果才为true，否则，只要有一方为false，则结果为false。

&&还具有短路的功能，即如果第一个表达式为false，则不再计算第二个表达式，例如，对于if(str != null && !str.equals(“”))表达式，当str为null时，后面的表达式不会执行，所以不会出现NullPointerException如果将&&改为&，则会抛出NullPointerException异常。If(x==33 & ++y>0) y会增长，If(x==33 && ++y>0)不会增长

&还可以用作位运算符，当&操作符两边的表达式不是boolean类型时，&表示按位与操作，我们通常使用0x0f来与一个整数进行&运算，来获取该整数的最低4个bit位，例如，0x31 & 0x0f的结果为0x01。

## Integer与int的区别

int是java提供的8种原始数据类型之一。Java为每个原始类型提供了封装类，Integer是java为int提供的封装类。int的默认值为0，而Integer的默认值为null，即Integer可以区分出未赋值和值为0的区别，int则无法表达出未赋值的情况，例如，要想表达出没有参加考试和考试成绩为0的区别，则只能使用Integer。在JSP开发中，Integer的默认为null，所以用el表达式在文本框中显示时，值为空白字符串，而int默认的默认值为0，所以用el表达式在文本框中显示时，结果为0，所以，int不适合作为web层的表单数据的类型。

## char 型变量中能放下一个汉字吗？

答：可以,Java中使用的编码是Unicode ,一个char类型占2个字节正好可以放下一个汉字;

## short a =1;a=a+1;和short a=1;a+=1;有什么区别？

答：前者编译会报错。因为1默认是int类型,需要强转才能和short类型的a相加,而+=实际上已经完成了强转的操作,所以后者不报错;

## 能将 int 强制转换为 byte 类型的变量吗？如果该值大于 byte 类型的范围，将会出现什么现象？

答：可以做强制转换，但是 Java 中 int 是 32 位的，而 byte 是 8 位的，所以，如果强制转化，int 类型的高 24 位将会被丢弃，因为byte 类型的范围是从 -128 到 127。

## switch是否能作用在byte上，是否能作用在long上，是否能作用在string上?

答：1.7版本以后除了long都可以，但是1.7版本之前不支持String,long始终不行

## 静态的多态和动态的多态的区别？

答：静态多态性指的是程序在编译时，系统就能决定调用哪个函数，如重载。动态多态性指在运行中才能动态确定操作指针所指的对象，主要通过虚函数和重写来实现。

## 抽象类的方法是否可同时是static,是否可同时是native，是否可同时是synchronized?

答：都不能。抽象类最重要的一点就是他的方法能够被子类实现，而被static和native修饰的方法都是没有子类的。而synchronized与abstract不可同时修饰一个方法，synchronized应该是作用在一个具体的方法上才有意义.

## 静态变量和实例变量的区别？

答：静态变量也称作类变量，由static修饰，只能通过类来访问。实例变量没有static修饰符，它只能通过实例对象来访问。同一类的不同的实例对象有自己的实例变量，但是它们共享同一个静态变量。java虚拟机在加载类的过程中为静态变量分配内存，静态变量位于方法区，被类的所有实例共享，其生命周期取决于类的生命周期。每创建一个实例，java虚拟机就会为实例变量分配一次内存，实例变量位于堆区中，其生命周期取决于实例的生命周期。

## 类加载的初始化顺序

答：父类静态变量>父类静态初始化块>子类静态变量>子类静态初始化块>父类变量>父类初始化块>父类构造器>子类变量>子类初始化块>子类构造器

## 是否可以从一个static方法内部发出对非static方法的调用？

答：不可以。当一个static方法被调用时，可能还没有创建任何实例对象，所以非static方法可能没有关联的对象。

## Object中有哪些公共方法?至少说出5个

答：equals、clone、hashcode、getClass、notify()、notifyAll()、wait()；

## 常见的runtime exception有哪些，至少说出5个

答：NullPointerException空指针、ClassCastExcetion强转错误、NumberFormatException字符串转数字异常、IOException流异常、NoSuchMethodException未找到方法、FileNotFoundException未找到文件。

## transient 有什么用，为什么 ArrayList 的 elementData 要加上 transient 修饰

答：transient 的作用是说不希望 elementData 数组被序列化 ，ArrayList是可序列化的，加上transient修饰elementData可以让ArrayList在每次序列化的时候只序列化已存入的元素，既加快了序列化的速度，又减小了序列化之后的文件大小；

## Collection接口的remove()方法和Iterator接口的remove()方法区别

答 ：Collection的remove中必须需要一个对象做为参数，且remove之后并不会改变collection的长度；Iterator的remove是直接进行指针操作，并且remove的同事会改变集合的长度。

## 为什么HashMap中String、Integer这样的包装类适合作为Key？

答：都是final类型，即不可变性，保证key的不可更改性，不会存在获取hash值不同的情况；内部已重写了equals()、hashCode()等方法，遵守了HashMap内部的规范，不容易出现Hash值计算错误的情况。

## hashCode()和equals方法的联系？

答：hashCode()方法和equals()方法的作用其实一样，在Java里都是用来对比两个对象是否相等一致 ,**equals()相等的两个对象他们的hashCode()肯定相等**,但是hashCode()方法速度效率比equals()更高更快，所以一般有大量的对比时我们应该先比较hashCode()，如果hashCode相等，再进行equals()。故只要重写equals，就必须重写hashCode来保证能够进行重复对象的判断。

## Serializable和Externalizable的区别

答：Externalizable接口继承自Serializable接口，使用Serializable的所有方式，在反序列化是不会调用任何的序列化对象构造器，而使用Externalizable是会调用一个无参构造方法，所以实现**Externalizable接口的类必须要提供一个public的无参的构造器。**

当一个父类序列化之后其子类自动实现序列化，序列化时最好指定一个序列ID，当某个字段被声明为transient后，默认序列化机制就会忽略该字段

## ArrayList 和 LinkedList 有什么区别

ArrayList和LinkedList都实现了List接口，有以下的不同点：

1.ArrayList是基于索引的数据接口。它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。以此对应，LinkedList是以元素列表的形式存储的数据，每一个元素都和它的前一个后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。

2.相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

3.LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

## 抽象类和接口的区别，类可以继承多个类么，接口可以继承多个接口么,类可以实现多个接口么

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该 类也只能为抽象类。

6、抽象方法只能申明，不能实现。abstract void abc();不能写成abstract void abc(){}。

7、抽象类里可以没有抽象方法 。

8、如果一个类里有抽象方法，那么这个类只能是抽象类 。

9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

10、接口可继承接口，并可多继承接口，但类只能单根继承

## JDK 和 JRE 有什么区别？

JRE是Java Runtime Environment的缩写，顾名思义是java运行时环境，包含了java虚拟机，java基础类库。是使用java语言编写的程序运行所需要的软件环境，是提供给想运行java程序的用户使用的，还有所有的Java类库的class文件，都在lib目录下，并且都打包成了jar。至于在Windows上的虚拟机是哪个文件呢？就是<JRE安装目录>/bin/client中的jvm.dll。

Jdk是Java Development Kit的缩写，顾名思义是java开发工具包，是程序员使用java语言编写java程序所需的开发工具包，是提供给程序员使用的。JDK包含了JRE，同时还包含了编译java源码的编译器javac，还包含了很多java程序调试和分析的工具：jconsole，jvisualvm等工具软件，还包含了java程序编写所需的文档和demo例子程序。

## == 和 equals 的区别是什么？

“==” 比较的是两个引用在内存中指向的是不是同一对象（即同一内存空间），也就是说在内存空间中的存储位置是否一致。如果两个对象的引用相同时（指向同一对象时），“==”操作符返回true，否则返回flase。

equals方法是由Object类提供的，可以由子类来进行重写

Object类默认的实现如下：

1 public boolean equals(Object obj) {

2

3 return (this == obj);

4

5 }

默认的实现只有当对象和自身进行比较时才会返回true， 这个时候和 “==”是等价的

两个对象的 hashCode()相同，则 equals()也一定为 true，对吗？

## final 在 java 中有什么作用？

final关键字可以用于三个地方。用于修饰类、类属性和类方法。

被final关键字修饰的类不能被继承，被final关键字修饰的类属性和类方法不能被覆盖（重写）；

对于被final关键字修饰的类属性而言，子类就不能给他重新赋值了，如果重新赋值，会报错

## java 中的 Math.round(-1.5) 等于多少？

-1

## String 属于基础的数据类型吗

不是

## java 中操作字符串都有哪些类？它们之间有什么区别？

StringBuffer、StringBuilder

## String str="i"与 String str=new String(“i”)一样吗？

String A = "ABC";内存会去查找永久代(常量池) ，如果没有的话，在永久代中中开辟一块儿内存空间，把地址付给栈指针，如果已经有了"ABC"的内存，直接把地址赋给栈指针；

因此

String str1="aa";

Srting str2="aa";

String Str3="aa";

....

这样下去，str1==Str2==str3;会一直相等下去，(a) ==的判断， (b) equals()的判断；都相等，因为他们的地址都相等，因此只在常量池中有一份内存空间，地址全部相同；

而String str = new String("a");是根据"a"这个String对象再次构造一个String对象;在堆中从新new一块儿内存，把指针赋给栈，

将新构造出来的String对象的引用赋给str。 因此 只要是new String()，则，栈中的地址都是指向最新的new出来的堆中的地址

## 如何将字符串反转？

很多方法吧。。。

## String 类的常用方法都有哪些？

很多方法吧。。。

## 抽象类必须要有抽象方法吗？

不一定。

## 普通类和抽象类有哪些区别？

抽象类不能实例化.

## 抽象类能使用 final 修饰吗？

Java抽象类是内部类时，可以被private修饰

## 接口和抽象类有什么区别？

很多区别。。。

## java 中 IO 流分为几种？

两种，字符流（Reader、Writer）和字节流(InputStream、OutputStream)。

<https://www.cnblogs.com/moonpool/p/5488463.html>

## BIO、NIO、AIO 有什么区别？

<https://blog.csdn.net/dreamer23/article/details/80903978>

## Files的常用方法都有哪些？

1.7的新类，用于替换老式的IO操作。

static Path copy(Path source, Path target, CopyOption... options)

将一个文件复制到目标文件。

static Path createDirectories(Path dir, FileAttribute<?>... attrs)

创建一个目录，通过创建所有不存在的父目录。

static Path createDirectory(Path dir, FileAttribute<?>... attrs)

创建一个新目录。

static Path createFile(Path path, FileAttribute<?>... attrs)

创建一个新的和空的文件，如果文件已经存在，失败了。

static Path createLink(Path link, Path existing)

创建一个新的链接（目录项）为现有的文件（可选操作）。

static Path createSymbolicLink(Path link, Path target, FileAttribute<?>... attrs)

创建一个符号链接到一个目标（可选操作）。

static Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs)

在指定的目录中创建一个新的目录，使用给定的前缀来生成它的名称。

## java 容器都有哪些？

List:存放有序,列表存储,元素可重复

Set:无序,元素不可重复

Map:无序,元素可重复

## Collection 和 Collections 有什么区别？

java.util.Collection 是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

Collections则是集合类的一个工具类/帮助类，其中提供了一系列静态方法，用于对集合中元素进行排序、搜索以及线程安全等各种操作。

## List、Set、Map 之间的区别是什么？

List：1.可以允许重复的对象。

　　 2.可以插入多个null元素。

3.是一个有序容器，保持了每个元素的插入顺序，输出的顺序就是插入的顺序。

4.常用的实现类有 ArrayList、LinkedList 和 Vector。ArrayList 最为流行，它提供了使用索引的随意访问，而 LinkedList 则对于经常需要从 List 中添加或删除元素的场合更为合适。

Set：1.不允许重复对象

　　 2. 无序容器，你无法保证每个元素的存储顺序，TreeSet通过 Comparator 或者 Comparable 维护了一个排序顺序。

3. 只允许一个 null 元素

4.Set 接口最流行的几个实现类是 HashSet、LinkedHashSet 以及 TreeSet。最流行的是基于 HashMap 实现的 HashSet；TreeSet 还实现了 SortedSet 接口，因此 TreeSet 是一个根据其 compare() 和 compareTo() 的定义进行排序的有序容器。

Map：1.Map不是collection的子接口或者实现类。Map是一个接口。

2.Map 的 每个 Entry 都持有两个对象，也就是一个键一个值，Map 可能会持有相同的值对象但键对象必须是唯一的。

3. TreeMap 也通过 Comparator 或者 Comparable 维护了一个排序顺序。

4. Map 里你可以拥有随意个 null 值但最多只能有一个 null 键。

5.Map 接口最流行的几个实现类是 HashMap、LinkedHashMap、Hashtable 和 TreeMap。（HashMap、TreeMap最常用）。

## HashMap 和 Hashtable 有什么区别？

HashMap和Hashtable都实现了Map接口，但决定用哪一个之前先要弄清楚它们之间的分别。主要的区别有：线程安全性，同步(synchronization)，以及速度。

1、HashMap几乎可以等价于Hashtable，除了HashMap是非synchronized的，并可以接受null(HashMap可以接受为null的键值(key)和值(value)，而Hashtable则不行)。

2、HashMap是非synchronized，而Hashtable是synchronized，这意味着Hashtable是线程安全的，多个线程可以共享一个Hashtable；而如果没有正确的同步的话，多个线程是不能共享HashMap的。Java 5提供了ConcurrentHashMap，它是HashTable的替代，比HashTable的扩展性更好。

3、另一个区别是HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。

4、由于Hashtable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢。如果你不需要同步，只需要单一线程，那么使用HashMap性能要好过Hashtable。

5、HashMap不能保证随着时间的推移Map中的元素次序是不变的。

## 如何决定使用 HashMap 还是 TreeMap？

HashMap：数组方式存储key/value，线程非安全，允许null作为key和value，key不可以重复，value允许重复，不保证元素迭代顺序是按照插入时的顺序，key的hash值是先计算key的hashcode值，然后再进行计算，每次容量扩容会重新计算所以key的hash值，会消耗资源，要求key必须重写equals和hashcode方法

默认初始容量16，加载因子0.75，扩容为旧容量乘2，查找元素快，如果key一样则比较value，如果value不一样，则按照链表结构存储value，就是一个key后面有多个value；

TreeMap：基于红黑二叉树的NavigableMap的实现，线程非安全，不允许null，key不可以重复，value允许重复，存入TreeMap的元素应当实现Comparable接口或者实现Comparator接口，会按照排序后的顺序迭代元素，两个相比较的key不得抛出classCastException。主要用于存入元素的时候对元素进行自动排序，迭代输出的时候就按排序顺序输出。

## 说一下 HashMap 的实现原理？

<https://www.cnblogs.com/chengxiao/p/6059914.html#t1>

<https://www.cnblogs.com/duodushuduokanbao/p/9492952.html>

jdk1.8以前：链表数组

jdk1.8：如果链表长度到达阀值(默认是8)，就会将链表转换成红黑二叉树。

红黑树：

<https://www.cnblogs.com/CarpenterLee/p/5503882.html>

<http://www.360doc.com/content/18/0904/19/25944647_783893127.shtml>

## 说一下 HashSet 的实现原理？

它是基于HashMap实现的，HashSet底层使用HashMap来保存所有元素，因此HashSet 的实现比较简单，相关HashSet的操作，基本上都是直接调用底层HashMap的相关方法来完成

## LinkedHashMap原理

LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。

LinkedHashMap实现与HashMap的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。

注意，此实现不是同步的。如果多个线程同时访问链接的哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

LinkedHashMap的实现：

对于LinkedHashMap而言，它继承与HashMap、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类HashMap相似，它通过重写父类相关的方法，来实现自己的链接列表特性。

## TreeMap原理

<https://blog.csdn.net/chenssy/article/details/26668941>

红黑树没搞懂哎

## ConcurrentHashMap原理

<http://www.importnew.com/28263.html>

<https://www.cnblogs.com/banjinbaijiu/p/9147434.html>

实在是看不下去。。以后再补吧。。

## 什么是红黑树？他的性质？

答：红黑树是特殊的二叉查找树 ；每个结点或是红色的，或是黑色的；根节点是黑色的 ；每个叶结点（NIL）是黑色的 ；如果一个节点是红色的，则它的两个儿子都是黑色的；对于每个结点，从该结点到其叶子结点构成的所有路径上的黑结点个数相同。

## ConcurrentHashMap1.7和1.8的区别

<https://blog.csdn.net/bolang789/article/details/79855053>

总结下：

1.7的segment是用重入锁做线程安全的。

1.8的segment用CAS来保证线程安全。

JDK6,7中的ConcurrentHashmap主要使用Segment来实现减小锁粒度，把HashMap分割成若干个Segment，在put的时候需要锁住Segment，get时候不加锁，使用volatile来保证可见性，当要统计全局时（比如size），首先会尝试多次计算modcount来确定，这几次尝试中，是否有其他线程进行了修改操作，如果没有，则直接返回size。如果有，则需要依次锁住所有的Segment来计算。

jdk7中ConcurrentHashmap中，当长度过长碰撞会很频繁，链表的增改删查操作都会消耗很长的时间，影响性能,所以jdk8 中完全重写了concurrentHashmap,代码量从原来的1000多行变成了 6000多 行，实现上也和原来的分段式存储有很大的区别。

主要设计上的变化有以下几点:

不采用segment而采用node，锁住node来实现减小锁粒度。

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

使用3个CAS操作来确保node的一些操作的原子性，这种方式代替了锁。

sizeCtl的不同值来代表不同含义，起到了控制的作用。

至于为什么JDK8中使用synchronized而不是ReentrantLock，我猜是因为JDK8中对synchronized有了足够的优化吧。

## ArrayList 和 LinkedList 的区别是什么？

1.ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

2.对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

3.对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

## 如何实现数组和 List 之间的转换？

List转数组：toArray(arraylist.size()方法

数组转List：Arrays的asList(a)方法

## ArrayList 和 Vector 的区别是什么？

1） Vector的方法都是同步的(Synchronized),是线程安全的(thread-safe)，而ArrayList的方法不是，由于线程的同步必然要影响性能，因此,ArrayList的性能比Vector好。

2） 当Vector或ArrayList中的元素超过它的初始大小时,Vector会将它的容量翻倍,而ArrayList只增加50%的大小，这样,ArrayList就有利于节约内存空间。

## Array 和 ArrayList 有何区别？

Array可以包含基本类型和对象类型，ArrayList只能包含对象类型

　　Array大小固定，ArrayList的大小是动态变化的。

　　ArrayList提供了更多的方法和特性：比如 ：addAll(),removeAll(),iterator()等等。

　　对于基本数据类型，集合使用自动装箱来减少编码工作量。但是，当处理固定大小基本数据类型的时候，这种方式相对较慢。

## 在 Queue 中 poll()和 remove()有什么区别？

1. queue的增加元素方法add和offer的区别在于，add方法在队列满的情况下将选择抛异常的方法来表示队列已经满了，而offer方法通过返回false表示队列已经满了；在有限队列的情况，使用offer方法优于add方法；

2. remove方法和poll方法都是删除队列的头元素，remove方法在队列为空的情况下将抛异常，而poll方法将返回null；

3. element和peek方法都是返回队列的头元素，但是不删除头元素，区别在与element方法在队列为空的情况下，将抛异常，而peek方法将返回null.

## 哪些集合类是线程安全的？

vector：就比arraylist多了个同步化机制（线程安全），因为效率较低，现在已经不太建议使用。在web应用中，特别是前台页面，往往效率（页面响应速度）是优先考虑的。

stack：堆栈类，先进后出

hashtable：就比hashmap多了个线程安全

enumeration：枚举，相当于迭代器

## 迭代器 Iterator 是什么？

为了方便的处理集合中的元素,Java中出现了一个对象,该对象提供了一些方法专门处理集合中的元素.例如删除和获取集合中的元素.该对象就叫做迭代器(Iterator).

对 Collection 进行迭代的类，称其为迭代器。还是面向对象的思想，专业对象做专业的事情，迭代器就是专门取出集合元素的对象。但是该对象比较特殊，不能直接创建对象（通过new），该对象是以内部类的形式存在于每个集合类的内部。

## Iterator 怎么使用？有什么特点？

 Iterator遍历集合元素的过程中不允许线程对集合元素进行修改，否则会抛出ConcurrentModificationEception的异常。

Iterator遍历集合元素的过程中可以通过remove方法来移除集合中的元素。

Iterator必须依附某个Collection对象而存在，Iterator本身不具有装载数据对象的功能。

Iterator.remove方法删除的是上一次Iterator.next()方法返回的对象。

强调以下next（）方法，该方法通过游标指向的形式返回Iterator下一个元素。

## Iterator 和 ListIterator 有什么区别？

1. ListIterator有add()方法，可以向List中添加对象，而Iterator不能

2. ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator就不可以。

3. ListIterator可以定位当前的索引位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。

4. 都可实现删除对象，但是ListIterator可以实现对象的修改，set()方法可以实现。Iierator仅能遍历，不能修改。

## 怎么确保一个集合不能被修改？

Collections.unmodifiableMap(Map<? extends K,? extends V> m);

Collections.unmodifiableList(List<? extends T> list)

unmodifiableSet(Set<? extends T> s)

## 并行和并发有什么区别？

所有的并发处理都有排队等候，唤醒，执行至少三个这样的步骤.所以并发肯定是宏观概念，在微观上他们都是序列被处理的，只不过资源不会在某一个上被阻塞(一般是通过时间片轮转)，所以在宏观上看多个几乎同时到达的请求同时在被处理。如果是同一时刻到达的请求也会根据优先级的不同，而先后进入队列排队等候执行。

    并发与并行是两个既相似而又不相同的概念：并发性，又称共行性，是指能处理多个同时性活动的能力；并行是指同时发生的两个并发事件，具有并发的含义，而并发则不一定并行，也亦是说并发事件之间不一定要同一时刻发生。

并发的实质是一个物理CPU(也可以多个物理CPU) 在若干道程序之间多路复用，并发性是对有限物理资源强制行使多用户共享以提高效率。

并行性指两个或两个以上事件或活动在同一时刻发生。在多道程序环境下，并行性使多个程序同一时刻可在不同CPU上同时执行。

## 线程和进程的区别？

根本区别：进程是操作系统资源分配的基本单位，而线程是任务调度和执行的基本单位

在开销方面：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。

所处环境：在操作系统中能同时运行多个进程（程序）；而在同一个进程（程序）中有多个线程同时执行（通过CPU调度，在每个时间片中只有一个线程执行）

内存分配方面：系统在运行的时候会为每个进程分配不同的内存空间；而对线程而言，除了CPU外，系统不会为线程分配内存（线程所使用的资源来自其所属进程的资源），线程组之间只能共享资源。

包含关系：没有线程的进程可以看做是单线程的，如果一个进程内有多个线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。

## 守护线程是什么？

守护线程（即daemon thread），是个服务线程，准确地来说就是服务其他的线程，这是它的作用——而其他的线程只有一种，那就是用户线程。所以java里线程分2种，

1、守护线程，比如垃圾回收线程，就是最典型的守护线程。

2、用户线程，就是应用程序里的自定义线程。

守护线程

1. 守护线程，专门用于服务其他的线程，如果其他的线程（即用户自定义线程）都执行完毕，连main线程也执行完毕，那么jvm就会退出（即停止运行）——此时，连jvm都停止运行了，守护线程当然也就停止执行了。
2. 再换一种说法，如果有用户自定义线程存在的话，jvm就不会退出——此时，守护线程也不能退出，也就是它还要运行，干嘛呢，就是为了执行垃圾回收的任务啊。

## 创建线程有哪几种方式？

一、继承Thread类创建线程类

（1）定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。

（2）创建Thread子类的实例，即创建了线程对象。

（3）调用线程对象的start()方法来启动该线程。

二、通过Runnable接口创建线程类

（1）定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。

（3）调用线程对象的start()方法来启动该线程。

三、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

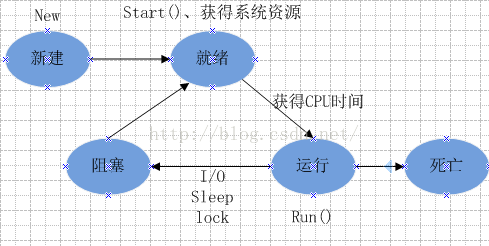
## 说一下 runnable 和 callable 有什么区别？

Runnable没有返回值；Callable可以返回执行结果，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果

Callable接口的call()方法允许抛出异常；Runnable的run()方法异常只能在内部消化，不能往上继续抛

注：Callalbe接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

## 线程有哪些状态？



### 1.新建状态

当用new操作符创建一个线程时。此时程序还没有开始运行线程中的代码。

### 2.就绪状态

一个新创建的线程并不自动开始运行，要执行线程，必须调用线程的start()方法。当线程对象调用start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行run()方法。当start()方法返回后，线程就处于就绪状态。

处于就绪状态的线程并不一定立即运行run()方法，线程还必须同其他线程竞争CPU时间，只有获得CPU时间才可以运行线程。因为在单CPU的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程是由Java运行时系统的线程调度程序来调度的。

### 3.运行状态（running）

当线程获得CPU时间后，它才进入运行状态，真正开始执行run()方法。

### 4.阻塞状态（blocked）

线程运行过程中，可能由于各种原因进入阻塞状态：

①线程通过调用sleep方法进入睡眠状态；

②线程调用一个在I/O上被阻塞的操作，即该操作在输入输出操作完成之前不会返回到它的调用者；

③线程试图得到一个锁，而该锁正被其他线程持有；

④线程在等待某个触发条件；

所谓阻塞状态是正在运行的线程没有运行结束，暂时让出CPU，这时其他处于就绪状态的线程就可以获得CPU时间，进入运行状态。

### 5.死亡状态（dead）

有两个原因会导致线程死亡：

①run方法正常退出而自然死亡；

②一个未捕获的异常终止了run方法而使线程猝死；

为了确定线程在当前是否存活着（就是要么是可运行的，要么是被阻塞了），需要使用isAlive方法，如果是可运行或被阻塞，这个方法返回true；如果线程仍旧是new状态且不是可运行的，或者线程死亡了，则返回false。

## sleep() 和 wait() 有什么区别？

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。

sleep()方法导致了程序暂停执行指定的时间，让出cpu给其他线程，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。

在调用sleep()方法的过程中，线程不会释放对象锁。

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。

## notify()和 notifyAll()有什么区别？

先说两个概念：锁池和等待池

锁池:假设线程A已经拥有了某个对象(注意:不是类)的锁，而其它的线程想要调用这个对象的某个synchronized方法(或者synchronized块)，由于这些线程在进入对象的synchronized方法之前必须先获得该对象的锁的拥有权，但是该对象的锁目前正被线程A拥有，所以这些线程就进入了该对象的锁池中。

等待池:假设一个线程A调用了某个对象的wait()方法，线程A就会释放该对象的锁后，进入到了该对象的等待池中。

如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。

当有线程调用了对象的 notifyAll()方法（唤醒所有 wait 线程）或 notify()方法（只随机唤醒一个 wait 线程），被唤醒的的线程便会进入该对象的锁池中，锁池中的线程会去竞争该对象锁。也就是说，调用了notify后只要一个线程会由等待池进入锁池，而notifyAll会将该对象等待池内的所有线程移动到锁池中，等待锁竞争

优先级高的线程竞争到对象锁的概率大，假若某线程没有竞争到该对象锁，它还会留在锁池中，唯有线程再次调用 wait()方法，它才会重新回到等待池中。而竞争到对象锁的线程则继续往下执行，直到执行完了 synchronized 代码块，它会释放掉该对象锁，这时锁池中的线程会继续竞争该对象锁。

## 线程的 run()和 start()有什么区别？

调用start（）后，线程会被放到等待队列，等待CPU调度，并不一定要马上开始执行，只是将这个线程置于可动行状态。然后通过JVM，线程Thread会调用run（）方法，执行本线程的线程体。先调用start后调用run，这么麻烦，为了不直接调用run？就是为了实现多线程的优点，没这个start不行。

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；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。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。  
   **2.run（）**方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。  
   记住：多线程就是分时利用CPU，宏观上让所有线程一起执行 ，也叫并发

## 如何停止一个线程？

当run() 或者 call() 方法执行完的时候线程会自动结束 ；

使用interrupt方法中断线程(这其实只是给县城一个中断的标记，并不是真正的停止)；

使用stop方法暴力停止线程(非常不建议)；

在run方法中 使用 this.interrupted();判断线程终止状态，如果为true则 throw new interruptedException()然后捕获该异常即可停止线程；

## interrupted 和 isInterruptedd方法的区别？

interrupted()：测试 当前线程 是否已经是中断状态，执行后具有清除状态功能。

isInterrupted()：测试线程 Thread 对象 是否已经是中断状态，但不清除状态标志

## Thread类中的yield方法有什么作用

答：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。

## 创建线程池有哪几种方式？

Executors的方法：

static ExecutorService newCachedThreadPool()

创建一个线程池，根据需要创建新的线程，但在可用时将重用先前构建的线程。

static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)

创建一个线程池线程创建新的需要，但会重用以前构建的线程可用时，使用所提供的threadfactory创建新线程时，需要。

static ExecutorService newFixedThreadPool(int nThreads)

创建一个线程池，使用固定数量的线程操作了共享无界队列。

static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)

创建一个线程池，使用固定数量的线程操作了共享无界队列，使用提供的threadfactory创建新线程时，需要。

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个线程池，可以调度命令在一个给定的延迟后运行，或周期性地执行。

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)

创建一个线程池，可以调度命令在一个给定的延迟后运行，或周期性地执行。

static ExecutorService newSingleThreadExecutor()

创建一个执行器，使用一个单一的工作线程操作关闭一个无限的队列。

static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)

创建一个执行器，采用单线程操作了无界队列，并使用所提供的threadfactory创建新线程需要时。

static ScheduledExecutorService newSingleThreadScheduledExecutor()

创建一个单线程的执行器，可以调度命令在一个给定的延迟后运行，或周期性地执行。

static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)

创建一个单线程的执行器，可以调度命令在一个给定的延迟后运行，或周期性地执行。

static ExecutorService newWorkStealingPool()

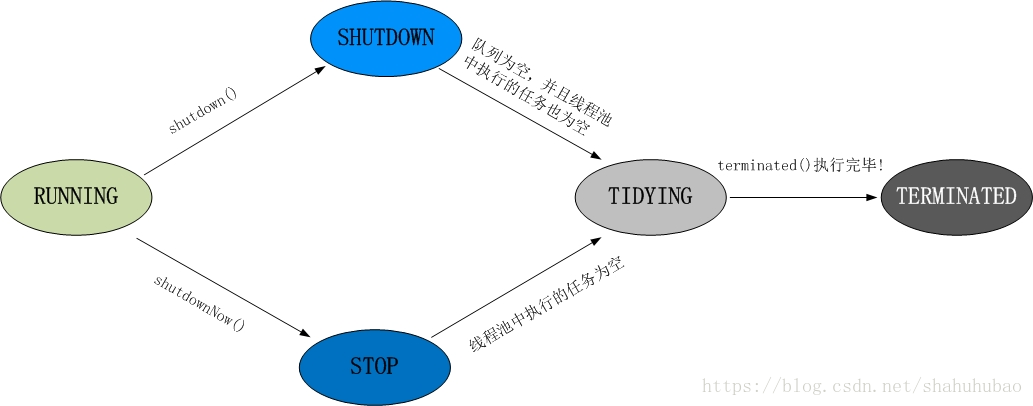
创建一个工作线程池使用偷 available processors为目标的并行度。

static ExecutorService newWorkStealingPool(int parallelism)

创建一个线程池，保持足够的线程来支持给定的并行级别，并可以使用多个队列来减少争用。

## 线程池都有哪些状态？

线程池的5种状态：Running、ShutDown、Stop、Tidying、Terminated。



1、RUNNING

(1) 状态说明：线程池处在RUNNING状态时，能够接收新任务，以及对已添加的任务进行处理。

(02) 状态切换：线程池的初始化状态是RUNNING。换句话说，线程池被一旦被创建，就处于RUNNING状态，并且线程池中的任务数为0！

2、 SHUTDOWN

(1) 状态说明：线程池处在SHUTDOWN状态时，不接收新任务，但能处理已添加的任务。

(2) 状态切换：调用线程池的shutdown()接口时，线程池由RUNNING -> SHUTDOWN。

3、STOP

(1) 状态说明：线程池处在STOP状态时，不接收新任务，不处理已添加的任务，并且会中断正在处理的任务。

(2) 状态切换：调用线程池的shutdownNow()接口时，线程池由(RUNNING or SHUTDOWN ) -> STOP。

4、TIDYING

(1) 状态说明：当所有的任务已终止，ctl记录的”任务数量”为0，线程池会变为TIDYING状态。当线程池变为TIDYING状态时，会执行钩子函数terminated()。terminated()在ThreadPoolExecutor类中是空的，若用户想在线程池变为TIDYING时，进行相应的处理；可以通过重载terminated()函数来实现。

(2) 状态切换：当线程池在SHUTDOWN状态下，阻塞队列为空并且线程池中执行的任务也为空时，就会由 SHUTDOWN -> TIDYING。

当线程池在STOP状态下，线程池中执行的任务为空时，就会由STOP -> TIDYING。

5、 TERMINATED

(1) 状态说明：线程池彻底终止，就变成TERMINATED状态。

(2) 状态切换：线程池处在TIDYING状态时，执行完terminated()之后，就会由 TIDYING -> TERMINATED。

## 线程池中 submit()和 execute()方法有什么区别？

线程池中的execute方法大家都不陌生，即开启线程执行池中的任务。还有一个方法submit也可以做到，它的功能是提交指定的任务去执行并且返回Future对象，即执行的结果。下面简要介绍一下两者的三个区别：

1、接收的参数不一样。Execute方法用Runnable对象，submit用callable对象；

2、submit有返回值，而execute没有；

用到返回值的例子，比如说我有很多个做validation的task，我希望所有的task执行完，然后每个task告诉我它的执行结果，是成功还是失败，如果是失败，原因是什么。

然后我就可以把所有失败的原因综合起来发给调用者。

而最大的用处应该是第二点。

3、submit方便Exception处理

意思就是如果你在你的task里会抛出checked或者unchecked exception，

而你又希望外面的调用者能够感知这些exception并做出及时的处理，那么就需要用到submit，通过捕获Future.get抛出的异常。

## 在 java 程序中怎么保证多线程的运行安全？

当多个线程要共享一个实例对象的值得时候，那么在考虑安全的多线程并发编程时就要保证下面3个要素：

原子性（Synchronized, Lock）

有序性(Volatile，Synchronized, Lock)

可见性(Volatile，Synchronized,Lock)

当然由于synchronized和Lock保证每个时刻只有一个线程执行同步代码，所以是线程安全的，也可以实现这一功能，但是由于线程是同步执行的，所以会影响效率。

**原子性**：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。在Java中，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。

**有序性**：即程序执行的顺序按照代码的先后顺序执行。

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。可以通过volatile关键字来保证一定的“有序性”。

**可见性**：指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

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共享变量时，它会去内存中读取新值。**volatile变量可以保证下一个读取操作会在前一个写操作之后发生。**

普通的共享变量不能保证可见性，因为普通共享变量被修改后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

更新主存的步骤：当前线程将其他线程的工作内存中的缓存变量的缓存行设置为无效，然后当前线程将变量的值跟新到主存，更新成功后将其他线程的缓存行更新为新的主存地址

其他线程读取变量时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

## § 线程池的原理，为什么要创建线程池？创建线程池的方式；

### 线程池的实现原理

提交一个任务到线程池中，线程池的处理流程如下：

1、判断**线程池里的核心线程**是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、判断**线程池里的线程**是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

### 为什么要创建线程池

为每个请求创建一个新线程的开销很大；为每个请求创建新线程的服务器在创建和销毁线程上花费的时间和消耗的系统资源要比花在处理实际的用户请求的时间和资源更多。

除了创建和销毁线程的开销之外，活动的线程也消耗系统资源。在一个 JVM 里创建太多的线程可能会导致系统由于过度消耗内存而用完内存或“切换过度”。为了防止资源不足，服务器应用程序需要一些办法来限制任何给定时刻处理的请求数目。

### 创建线程池的方式

#### 1 newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

这种类型的线程池特点是：

* 工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。
* 如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。
* 在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

#### 2 newFixedThreadPool

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

#### 3 newSingleThreadExecutor

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO,优先级)执行。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

#### 4 newScheduleThreadPool

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。

## §线程的生命周期，什么时候会出现僵死进程；

<https://blog.csdn.net/qingzhuyuxian/article/details/80312517>

僵尸进程的产生：

当一个进程创建了一个子进程时，他们的运行时异步的。即父进程无法预知子进程会在什么时候结束，那么如果父进程很繁忙来不及wait 子进程时，那么当子进程结束时，会不会丢失子进程的结束时的状态信息呢？处于这种考虑unix提供了一种机制可以保证只要父进程想知道子进程结束时的信息，它就可以得到。

这种机制是：在每个进程退出的时候，内核释放该进程所有的资源，包括打开的文件，占用的内存，但是仍然保留了一些信息（如进程号pid 退出状态 运行时间等）。这些保留的信息直到进程通过调用wait/waitpid时才会释放。这样就导致了一个问题，如果没有调用wait/waitpid的话，那么保留的信息就不会释放。比如进程号就会被一直占用了。但系统所能使用的进程号的有限的，如果产生大量的僵尸进程，将导致系统没有可用的进程号而导致系统不能创建进程。

## § 说说线程安全问题，什么是线程安全，如何实现线程安全；

**当多个线程访问某个方法时，不管你通过怎样的调用方式或者说这些线程如何交替的执行，我们在主程序中不需要去做任何的同步，这个类的结果行为都是我们设想的正确行为，那么我们就可以说这个类时线程安全的。**

## § 创建线程池有哪几个核心参数？ 如何合理配置线程池的大小？

### corePoolSize（核心线程数）

（1）核心线程会一直存在，即使没有任务执行；

（2）当线程数小于核心线程数的时候，即使有空闲线程，也会一直创建线程直到达到核心线程数；

（3）设置allowCoreThreadTimeout=true（默认false）时，核心线程会超时关闭。

### queueCapacity（任务队列容量）

也叫阻塞队列，当核心线程都在运行，此时再有任务进来，会进入任务队列，排队等待线程执行。

### maxPoolSize（最大线程数）

（1）线程池里允许存在的最大线程数量；

（2）当任务队列已满，且线程数量大于等于核心线程数时，会创建新的线程执行任务；

（3）线程池里允许存在的最大线程数量。当任务队列已满，且线程数量大于等于核心线程数时，会创建新的线程执行任务。

### keepAliveTime（线程空闲时间）

（1）当线程空闲时间达到keepAliveTime时，线程会退出（关闭），直到线程数等于核心线程数；

（2）如果设置了allowCoreThreadTimeout=true，则线程会退出直到线程数等于零。

allowCoreThreadTimeout（允许核心线程超时）

### rejectedExecutionHandler（任务拒绝处理器）

（1）当线程数量达到最大线程数，且任务队列已满时，会拒绝任务；

（2）调用线程池shutdown()方法后，会等待执行完线程池的任务之后，再shutdown()。如果在调用了shutdown()方法和线程池真正shutdown()之间提交任务，会拒绝新任务。

对于不同性质的任务来说，CPU密集型任务应配置尽可能小的线程，如配置CPU个数+1的线程数，IO密集型任务应配置尽可能多的线程，因为IO操作不占用CPU，不要让CPU闲下来，应加大线程数量，如配置两倍CPU个数+1，而对于混合型的任务，如果可以拆分，拆分成IO密集型和CPU密集型分别处理，前提是两者运行的时间是差不多的，如果处理时间相差很大，则没必要拆分了。

若任务对其他系统资源有依赖，如某个任务依赖数据库的连接返回的结果，这时候等待的时间越长，则CPU空闲的时间越长，那么线程数量应设置得越大，才能更好的利用CPU。

## § volatile、ThreadLocal的使用场景和原理；

<https://blog.csdn.net/vking_wang/article/details/9982709>

<https://www.cnblogs.com/chenssy/p/6379280.html>

Volatile的使用场景：

要使 volatile 变量提供理想的线程安全，必须同时满足下面两个条件：

* 对变量的写操作不依赖于当前值。
* 该变量没有包含在具有其他变量的不变式中。

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值**独立于任何程序的状态**，包括变量的当前状态。

**volatile经常用于两个两个场景：状态标记两、double check**

ThreadLocal的主要用途是为了保持线程自身对象和避免参数传递，主要适用场景是按线程多实例（每个线程对应一个实例）的对象的访问，并且这个对象很多地方都要用到。

<https://blog.csdn.net/qq_36632687/article/details/79551828>

<https://mp.weixin.qq.com/s/aM03vvSpDpvwOdaJ8u3Zgw>

## § ThreadLocal什么时候会出现OOM的情况？为什么？

<https://blog.csdn.net/bntx2jsqfehy7/article/details/78315161>

## synchronized锁粒度、模拟死锁场景、原子性与可见性；

<http://www.360doc.com/content/16/0311/16/18924983_541356132.shtml>

## 多线程锁的升级原理是什么？

先说说为什么会有锁升级

因为Sycronized是重量级锁（也是悲观锁），每次在要进行锁的请求的时候，如果当前资源被其他线程占有要将当前的线程阻塞加入到阻塞队列，然后清空当前线程的缓存，等到锁释放的时候再通过notify或者notifyAll唤醒当前的线程，并让其处于就绪状态。这样线程的来回切换是非常消耗系统资源的，而且有的时候，线程刚挂起资源就释放了。而Java的线程是映射到操作系统的原生线程之上的。每次线程的阻塞或者唤醒都要经过用户态到核心态或者核心态到用户态的转化，这样是十分浪费资源的，这样就会造成性能上的降低，因此JVM对Sychronized进行了优化，将Sycronized分为三种锁的级别：偏向锁，轻量级锁，重量级锁。

很多的时候，对于一个可能发生并发访问的对象而言，其实很少会被竞争，就算有些资源存在竞争也是在很少的一段时间资源就会被释放，而这样的情况下将线程挂起是十分浪费性能的。

偏向锁（乐观锁）：

当锁对象第一次被线程获取的时候，虚拟机会将锁对象的对象头中的锁标志位设置成为01，并将偏向锁标志设置为1，线程通过CAS的方式将自己的ID值放置到对象头中（因为在这个过程中有可能会有其他线程来竞争锁，所以要通过CAS的方式，一旦有竞争就会升级为轻量级锁了），如果成功线程就获得了该轻量级锁。这样每次再进入该锁对象的时候不用进行任何的同步操作，直接比较当前锁对象的对象头是不是该线程的ID，如果是就可以直接进入。

偏向锁升级为轻量级锁

偏向锁是一种无竞争锁，一旦出现了竞争大多数情况下就会升级为轻量级锁。现在我们假设有线程1持有偏向锁，线程2来竞争偏向锁会经历以下几个过程：

1. 首先线程2会先检查偏向锁标记， 如果是1，说明当前是偏向锁，那么JVM会找到线程1，看线程1是否还存活着；

2. 如果线程1已经执行完毕，就是说线程1不存在了（线程1自己是不会去释放偏向锁的），那么先将偏向锁置为0，对象头设置成为无锁的状态，用CAS的方式尝试将线程2的ID放入对象头中，不进行锁升级，还是偏向锁 ；

3. 如果线程1还活着，先暂停线程1，将锁标志位变成00（轻量级锁）然后在线程1的栈帧中开辟出一块空间（Display Mark Word）将对象头的Mark Word置换到线程一的栈帧当中，而对象头中此时存储的是指向当前线程栈帧的指针。此时就变成了轻量级锁。继续执行线程1，然后线程2采用CAS的方式尝试获取锁。

轻量级锁与偏向锁最大的不同之处

轻量级锁和偏向锁的不同之处就在于轻量级锁对于获取锁对象采用CAS的同步方式而偏向锁直接是把整个同步过程给取消。

轻量级锁(乐观锁)

轻量级锁如何创建在上面已经讲过了，接下来说说轻量级锁如何获取锁对象。轻量级锁是通过CAS也就是自旋的方式尝试获取锁对象，一旦失败会先检查，对象头中存储的是否是指向当前线程栈帧的指针，如果是，就可以获取对象，如果不是说明存在竞争那么就要膨胀为重量级锁。

轻量级锁的解锁也是通过CAS的方式尝试将对象头的Mark Word和线程中的Display Mark Word替换回来。如果成功，就释放锁，如果失败说明还有许多其他等待锁的线程（说明此时已经不是轻量级锁而是重量级锁了），会将这些线程唤醒，然后释放锁。

轻量级锁膨胀为重量级锁

**一旦有两条以上的线程竞争锁**，轻量级锁膨胀为重量级锁，锁的状态变成10，此时对象头中存储的就是指向重量级锁的栈帧的指针。而且其他等待锁的线程要进入阻塞状态，等待重量级锁释放再来被唤醒然后去竞争。

线程的生命周期存在5个状态，start、running、waiting、blocking和dead

对于一个synchronized修饰的方法(代码块)来说：

当多个线程同时访问该方法，那么这些线程会先被放进\_EntryList队列，此时线程处于blocking状态；

当一个线程获取到了实例对象的监视器（monitor）锁，那么就可以进入running状态，执行方法，此时，ObjectMonitor对象的\_owner指向当前线程，\_count加1表示当前对象锁被一个线程获取

当running状态的线程调用wait()方法，那么当前线程释放monitor对象，进入waiting状态，ObjectMonitor对象的\_owner变为null，\_count减1，同时线程进入\_WaitSet队列，直到有线程调用notify()方法唤醒该线程，则该线程重新获取monitor对象进入\_Owner区

如果当前线程执行完毕，那么也释放monitor对象，进入waiting状态，ObjectMonitor对象的\_owner变为null，\_count减1

**锁升级**

锁的4中状态：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态（级别从低到高）

1）偏向锁：

为什么要引入偏向锁？

因为经过HotSpot的作者大量的研究发现，大多数时候是不存在锁竞争的，常常是一个线程多次获得同一个锁，因此如果每次都要竞争锁会增大很多没有必要付出的代价，为了降低获取锁的代价，才引入的偏向锁。

偏向锁的升级

当线程1访问代码块并获取锁对象时，会在java对象头和栈帧中记录偏向的锁的threadID，因为偏向锁不会主动释放锁，因此以后线程1再次获取锁的时候，需要比较当前线程的threadID和Java对象头中的threadID是否一致，如果一致（还是线程1获取锁对象），则无需使用CAS来加锁、解锁；如果不一致（其他线程，如线程2要竞争锁对象，而偏向锁不会主动释放因此还是存储的线程1的threadID），那么需要查看Java对象头中记录的线程1是否存活，如果没有存活，那么锁对象被重置为无锁状态，其它线程（线程2）可以竞争将其设置为偏向锁；如果存活，那么立刻查找该线程（线程1）的栈帧信息，如果还是需要继续持有这个锁对象，那么暂停当前线程1，撤销偏向锁，升级为轻量级锁，如果线程1 不再使用该锁对象，那么将锁对象状态设为无锁状态，重新偏向新的线程。

偏向锁的取消：

偏向锁是默认开启的，而且开始时间一般是比应用程序启动慢几秒，如果不想有这个延迟，那么可以使用-XX:BiasedLockingStartUpDelay=0；

如果不想要偏向锁，那么可以通过-XX:-UseBiasedLocking = false来设置；

2）轻量级锁

为什么要引入轻量级锁？

轻量级锁考虑的是竞争锁对象的线程不多，而且线程持有锁的时间也不长的情景。因为阻塞线程需要CPU从用户态转到内核态，代价较大，如果刚刚阻塞不久这个锁就被释放了，那这个代价就有点得不偿失了，因此这个时候就干脆不阻塞这个线程，让它自旋这等待锁释放。

轻量级锁什么时候升级为重量级锁？

线程1获取轻量级锁时会先把锁对象的对象头MarkWord复制一份到线程1的栈帧中创建的用于存储锁记录的空间（称为DisplacedMarkWord），然后使用CAS把对象头中的内容替换为线程1存储的锁记录（DisplacedMarkWord）的地址；

如果在线程1复制对象头的同时（在线程1CAS之前），线程2也准备获取锁，复制了对象头到线程2的锁记录空间中，但是在线程2CAS的时候，发现线程1已经把对象头换了，线程2的CAS失败，那么线程2就尝试使用自旋锁来等待线程1释放锁。

但是如果自旋的时间太长也不行，因为自旋是要消耗CPU的，因此自旋的次数是有限制的，比如10次或者100次，如果自旋次数到了线程1还没有释放锁，或者线程1还在执行，线程2还在自旋等待，这时又有一个线程3过来竞争这个锁对象，那么这个时候轻量级锁就会膨胀为重量级锁。重量级锁把除了拥有锁的线程都阻塞，防止CPU空转。

**\*注意：**为了避免无用的自旋，轻量级锁一旦膨胀为重量级锁就不会再降级为轻量级锁了；偏向锁升级为轻量级锁也不能再降级为偏向锁。一句话就是锁可以升级不可以降级，但是偏向锁状态可以被重置为无锁状态。



2、锁粗化

按理来说，同步块的作用范围应该尽可能小，仅在共享数据的实际作用域中才进行同步，这样做的目的是为了使需要同步的操作数量尽可能缩小，缩短阻塞时间，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

但是加锁解锁也需要消耗资源，如果存在一系列的连续加锁解锁操作，可能会导致不必要的性能损耗。

锁粗化就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁，避免频繁的加锁解锁操作。

3、锁消除

Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，经过逃逸分析，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间

## 什么是死锁？

线程死锁是指由于两个或者多个线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。当线程进入对象的synchronized代码块时，便占有了资源，直到它退出该代码块或者调用wait方法，才释放资源，在此期间，其他线程将不能进入该代码块。当线程互相持有对方所需要的资源时，会互相等待对方释放资源，如果线程都不主动释放所占有的资源，将产生死锁。

当然死锁的产生是必须要满足一些特定条件的：

1.互斥条件：进程对于所分配到的资源具有排它性，即一个资源只能被一个进程占用，直到被该进程释放

2.请求和保持条件：一个进程因请求被占用资源而发生阻塞时，对已获得的资源保持不放。

3.不剥夺条件：任何一个资源在没被该进程释放之前，任何其他进程都无法对他剥夺占用

4.循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路（类似于死循环），造成永久阻塞。

## 怎么防止死锁？

我们可以通过破坏死锁产生的4个必要条件来 预防死锁，由于资源互斥是资源使用的固有特性是无法改变的。

1. 破坏“不可剥夺”条件：一个进程不能获得所需要的全部资源时便处于等待状态，等待期间他占有的资源将被隐式的释放重新加入到 系统的资源列表中，可以被其他的进程使用，而等待的进程只有重新获得自己原有的资源以及新申请的资源才可以重新启动，执行。
2. 破坏”请求与保持条件“：第一种方法静态分配即每个进程在开始执行时就申请他所需要的全部资源。第二种是动态分配即每个进程在申请所需要的资源时他本身不占用系统资源。
3. 破坏“循环等待”条件：采用资源有序分配其基本思想是将系统中的所有资源顺序编号，将紧缺的，稀少的采用较大的编号，在申请资源时必须按照编号的顺序进行，一个进程只有获得较小编号的进程才能申请较大编号的进程。

## 怎么检测一个线程是否拥有锁？

答：在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁；

## ThreadLocal 是什么？有哪些使用场景？

ThreadLocal是如何为每个线程创建变量的副本的：

首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

ThreadLocal 用一种存储变量与线程绑定的方式，在每个线程中用自己的 ThreadLocalMap 安全隔离变量，为解决多线程程序的并发问题提供了一种新的思路，如为每个线程创建一个独立的数据库连接。因为是线程绑定的，所以在很多场景也被用来实现线程参数传递，如 Spring 的 RequestContextHolder。也因为每个线程拥有自己唯一的 ThreadLocalMap ，所以 ThreadLocalMap 是天然线程安全的。

## 说一下 synchronized 底层实现原理？

<https://blog.csdn.net/javazejian/article/details/72828483>

<https://baijiahao.baidu.com/s?id=1612142459503895416&wfr=spider&for=pc>

synchronized的俩种加锁方式：

1、某个对象实例内

此作用域内的synchronized锁 ，可以防止多个线程同时访问这个对象的synchronized方法

并且一个对象有多个synchronized方法，只要一个线程访问了其中的一个synchronized方法，其它线程不能同时访问这个对象中任何一个synchronized方法。

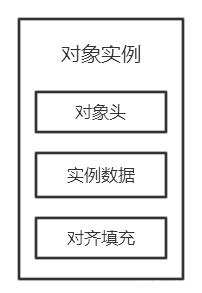
此外，不同对象实例的synchronized方法是不相干预的。也就是说，其它线程可以同时访问此类下的另一个对象实例中的synchronized方法；

2、某个类

此作用域下，可以防止多个线程同时访问这个类中的synchronized方法。也就是说此种修饰，可以对此类的所有对象实例起作用。

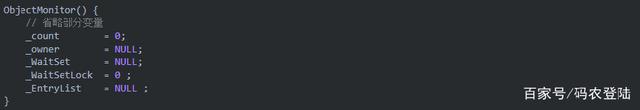
注意一点，synchronized关键字是不能继承的，也就是说，基类的方法synchronized fun(){} 在继承类中并不自动是synchronized fun(){}，而是变成了fun(){}。继承时，需要显式的指定它的某个方法为synchronized方法。有机会你可以自己写个demo试一下。

对象在堆中的结构：



synchronized的对象锁，其指针指向的是一个monitor对象（由C++实现）的起始地址。每个对象实例都会有一个 monitor。其中monitor可以与对象一起创建、销毁；亦或者当线程试图获取对象锁时自动生成。

monitor是由ObjectMonitor实现（ObjectMonitor.hpp文件，C++实现的），对于我们来说主要关注的是如下代码：



我们可以看到这里定义了\_WaitSet 和 \_EntryList俩个队列，其中\_WaitSet 用来保存每个等待锁的线程对象。

让我们先看一下\_owner，它指向持有ObjectMonitor对象的线程。当多个线程同时访问一段同步代码时，会先存放到 \_EntryList 集合中，接下来当线程获取到对象的monitor时，就会把\_owner变量设置为当前线程。同时count变量+1。如果线程调用wait() 方法，就会释放当前持有的monitor，那么\_owner变量就会被置为null，同时\_count减1，并且该线程进入 WaitSet集合中，等待下一次被唤醒。

当然，若当前线程顺利执行完方法，也将释放monitor，重走一遍刚才的内容，也就是\_owner变量就会被置为null，同时\_count减1，并且该线程进入 WaitSet集合中，等待下一次被唤醒。

因为这个锁对象存放在对象本身，也就是为什么Java中任意对象可以作为锁的原因。

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现， 无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法 并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的。

## synchronized 和 volatile 的区别是什么？

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。  
   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。

1.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

2.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

3.volatile不会造成线程的阻塞；

synchronized可能会造成线程的阻塞。

4.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

volatile作为java中的关键词之一，用以声明变量的值可能随时会别的线程修改，使用volatile修饰的变量会强制将修改的值立即写入主存，主存中值的更新会使缓存中的值失效(非volatile变量不具备这样的特性，非volatile变量的值会被缓存，线程A更新了这个值，线程B读取这个变量的值时可能读到的并不是是线程A更新后的值)。volatile会禁止指令重排。

volatile具有可见性、有序性，不具备原子性。

注意，volatile不具备原子性，这是volatile与java中的synchronized、java.util.concurrent.locks.Lock最大的功能差异，这一点在面试中也是非常容易问到的点。

## synchronized 和 Lock 有什么区别？

<https://www.cnblogs.com/baizhanshi/p/6419268.html>

采用synchronized关键字来实现同步的话，就会导致一个问题：

　　如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

　　因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。

　　另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

　　总结一下，也就是说Lock提供了比synchronized更多的功能。但是要注意以下几点：

　　1）Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

　　2）Lock和synchronized有一点非常大的不同，采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。

## synchronized 和 ReentrantLock 区别是什么？

相同点：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（操作系统需要在用户态与内核态之间来回切换，代价很高，不过可以通过对锁优化进行改善）。

**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JDK1.5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是托管给JVM执行的，而lock是java写的控制锁的代码。在Java1.5中，synchronize是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用Java提供的Lock对象，性能更高一些。但是到了Java1.6，发生了变化。synchronize在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在Java1.6上synchronize的性能并不比Lock差。官方也表示，他们也更支持synchronize，在未来的版本中还有优化余地。

synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。

而Lock用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就是CAS操作（CompareandSwap）。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。

1、ReentrantLock 拥有Synchronized相同的并发性和内存语义，此外还多了 锁投票，定时锁等候和中断锁等候

线程A和B都要获取对象O的锁定，假设A获取了对象O锁，B将等待A释放对O的锁定，

如果使用 synchronized ，如果A不释放，B将一直等下去，不能被中断

如果 使用ReentrantLock，如果A不释放，可以使B在等待了足够长的时间以后，中断等待，而干别的事情

**ReentrantLock获取锁定的三种方式：**

a) lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁

b) tryLock(), 如果获取了锁立即返回true，如果别的线程正持有锁，立即返回false；

c) tryLock(long timeout,TimeUnit unit)， 如果获取了锁定立即返回true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回true，如果等待超时，返回false；

d) lockInterruptibly:如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到或者锁定，或者当前线程被别的线程中断

2、synchronized是在JVM层面上实现的，不但可以通过一些监控工具监控synchronized的锁定，而且在代码执行时出现异常，JVM会自动释放锁定，但是使用Lock则不行，lock是通过代码实现的，要保证锁定一定会被释放，就必须将unLock()放到finally{}中

3、在资源竞争不是很激烈的情况下，Synchronized的性能要优于ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized的性能会下降几十倍，但是ReetrantLock的性能能维持常态；

## 说一下 atomic 的原理？

CAS+自旋

<https://blog.csdn.net/wuzhiwei549/article/details/82621947>

## 什么是反射？

Java反射就是在运行状态中，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方法和属性；并且能改变它的属性。而这也是Java被视为动态（或准动态，为啥要说是准动态，因为一般而言的动态语言定义是程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言。从这个观点看，Perl，Python，Ruby是动态语言，C++，Java，C#不是动态语言。）语言的一个关键性质。

**使用反射基本上一种解释操作，告诉JVM我们要做什么并且满足我们的要求，这类操作总是慢于直接执行java代码。**

## JDK 为每种类型的流提供了一些抽象类以供继承，请说出他们分别是哪些类？

字节流继承于InputStream、OutputStream；字符流继承于Reader、Writer。其它与IO 操作相关的类都是派生至上述4 个抽象类，如字节相关的：FileInputStream、FileOutputStream 类；字符相关的：BufferedReader、Buffered Writer类。

## BufferedReader的readLine()方法有什么需要注意的地方

在网络上，readLine()方法是阻塞式的 , 如果readLine()读取不到数据的话，会一直阻塞，而不是返回null；

在while()里面判断readLine()！= null的时候要赋值给一个String，因为如果不为null，那么这时候已经读了一行。如果用while (br.readLine()!=null)，那么下面没法再获取到这一行，所以应该用while ((line = br.readLine())!=null){} ；

## 怎么样把我们控制台的输出改成输出到一个文件里面,这个技术叫什么

答：SetOut（printWriter,printStream）重定向

## 流一般需要不需要关闭,如果关闭的话在用什么方法,一般要在那个代码块里面关闭比较好，处理流是怎么关闭的，如果有多个流互相调用传入是怎么关闭的？

流一般需要不需要关闭,如果关闭的话在用什么方法,一般要在那个代码块里面关闭比较好，处理流是怎么关闭的，如果有多个流互相调用传入是怎么关闭的？

答：流一旦打开就必须关闭，使用close方法；放入finally语句块中（finally 语句一定会执行）；调用的处理流就关闭处理流；多个流互相调用只关闭最外层的流；

## 什么是 java 序列化？什么情况下需要序列化？

序列化：把Java对象转换为字节序列的过程。

序列化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。

对象的默认序列化机制写入的内容是：对象的类，类签名，以及非瞬态和非静态字段的值。

## 动态代理是什么？有哪些应用？

动态代理（以下称代理），利用Java的反射技术(Java Reflection)，在运行时创建一个实现某些给定接口的新类（也称“动态代理类”）及其实例（对象）

(Using Java Reflection to create dynamic implementations of interfaces at runtime)。

代理的是接口(Interfaces)，不是类(Class)，更不是抽象类。

动态代理有什么用

解决特定问题：一个接口的实现在编译时无法知道，需要在运行时才能实现

实现某些设计模式：适配器(Adapter)或修饰器(Decorator)

面向切面编程：如AOP in Spring

代理对象和被代理对象一般实现相同的接口，调用者与代理对象进行交互。代理的存在对于调用者来说是透明的，调用者看到的只是接口。代理对象则可以封装一些内部的处理逻辑，如访问控制、远程通信、日志、缓存等。比如一个对象访问代理就可以在普通的访问机制之上添加缓存的支持。这种模式在RMI和EJB中都得到了广泛的使用。传统的代理模式的实现，需要在源代码中添加一些附加的类。这些类一般是手写或是通过工具来自动生成。  
JDK 5引入的动态代理机制，允许开发人员在运行时刻动态的创建出代理类及其对象。在运行时刻，可以动态创建出一个实现了多个接口的代理类。每个代理类的对象都会关联一个表示内部处理逻辑的InvocationHandler接 口的实现。当使用者调用了代理对象所代理的接口中的方法的时候，这个调用的信息会被传递给InvocationHandler的invoke方法。在 invoke方法的参数中可以获取到代理对象、方法对应的Method对象和调用的实际参数。invoke方法的返回值被返回给使用者。这种做法实际上相 当于对方法调用进行了拦截。熟悉AOP的人对这种使用模式应该不陌生。但是这种方式不需要依赖AspectJ等AOP框架。

## 怎么实现动态代理？

<https://www.cnblogs.com/lxcmyf/p/6433018.html>

## 为什么要使用克隆？

克隆是为了防止，修改复制的对象时改变了原对象。

clone方法是在复制一个对象，赋值的对象是单独独立的，有独自的内存空间。

clone方法是java中顶层父类Object中的一个方法，此方法在java中为实现，是一个native方法，也就是本地方法（可以调用底层操作系统的方法），在调用本地方法创建对象，比直接new创建对象效率高。

## 如何实现对象克隆？

两种不同的克隆方法，浅克隆(ShallowClone)和深克隆(DeepClone)。

在Java语言中，数据类型分为值类型（基本数据类型）和引用类型，值类型包括int、double、byte、boolean、char等简单数据类型，引用类型包括类、接口、数组等复杂类型。浅克隆和深克隆的主要区别在于是否支持引用类型的成员变量的复制。

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；

**Cloneable接口**

（1）clone()方法是定义在java.lang.Object类中，该方法是一个protected的方法，所以重载时要把clone()方法的属性设置为public，这样其它类才能调用这个clone类的clone()方法

（2）实现Cloneable接口：Cloneable接口是不包含任何方法的！其实这个接口仅仅是一个标志，而且这个标志也仅仅是针对Object类中clone()方法的，如果clone类没有实现Cloneable接口，并调用了Object的clone()方法（也就是调用了super.Clone()方法），那么Object的clone()方法就会抛出 CloneNotSupportedException异常。

**Serializale接口**

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();

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定，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，这种是方案明显优于使用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优于把问题留到运行时。

## 深拷贝和浅拷贝区别是什么？

在浅克隆中，如果原型对象的成员变量是值类型，将复制一份给克隆对象；**如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。**

在深克隆中，无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。

简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制。

如果引用类型里面还包含很多引用类型，或者内层引用类型的类里面又包含引用类型，使用clone方法就会很麻烦。这时我们可以用序列化的方式来实现对象的深克隆。

序列化就是将对象写到流的过程，写到流中的对象是原有对象的一个拷贝，而原对象仍然存在于内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且可以复制其引用的成员对象，因此通过序列化将对象写到一个流中，再从流里将其读出来，可以实现深克隆。需要注意的是能够实现序列化的对象其类必须实现Serializable接口，否则无法实现序列化操作。

## jsp 和 servlet 有什么区别？

jsp和servlet的区别和联系：

1.jsp经编译后就变成了Servlet.(JSP的本质就是Servlet，JVM只能识别java的类，不能识别JSP的代码,Web容器将JSP的代码编译成JVM能够识别的java类)

2.jsp更擅长表现于页面显示,servlet更擅长于逻辑控制.

3.Servlet中没有内置对象，Jsp中的内置对象都是必须通过HttpServletRequest对象，HttpServletResponse对象以及HttpServlet对象得到.

Jsp是Servlet的一种简化，使用Jsp只需要完成程序员需要输出到客户端的内容，Jsp中的Java脚本如何镶嵌到一个类中，由Jsp容器完成。而Servlet则是个完整的Java类，这个类的Service方法用于生成对客户端的响应。

联系： JSP是Servlet技术的扩展，本质上就是Servlet的简易方式。JSP编译后是“类servlet”。Servlet和JSP最主要的不同点在于，Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML里分离开来。而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。JSP侧重于视图，Servlet主要用于控制逻辑。

Servlet是[**Java**](http://lib.csdn.net/base/javase)提供的用于开发Web服务器应用程序的一个组件，运行在服务器端，由Servlet容器所管理，用于生成动态的内容。Servlet是平台独立的Java类，编写一个Servlet，实际上就是按照Servlet规范编写一个Java类。

Servlet的生命周期可以被归纳为以下几步：  
　　（1） 装载Servlet，这一项操作一般是动态执行的。然而，Servlet通常会提供一个管理的选项，用于在Servlet启动时强制装载和初始化特定的Servlet  
　　（2） Server创建一个Servlet实例  
　　（3） Server调用Servlet的init方法  
　　（4） 一个客户端请求到达Server  
　　（5） Server创建一个请求对象  
　　（6） Server创建一个响应对象  
　　（7） Server激活Servlet的service方法，传递请求和响应对象作为参数  
　　（8） service方法获得关于请求对象的信息，处理请求，访问其他资源，获得需要的信息  
　　（9） service方法使用响应对象的方法。将响应传回Server，最终到达客户端。Service方法可能激活其他方法以处理请求。如doGet，doPost或其他程序员自己开发的方法  
　　（10） 对于更多的客户端请求，Server创建新的请求和响应对象，仍然激活此servlet的service方法，将这两个对象作为参数传递给它，如此重复以上的循环，但无需再次调用init方法，Servlet一般只初始化一次  
　　（11） 当Server不再需要Servlet时，比如当Server要关闭时，Server调用Servlet的destroy

JSP一般的运行方式为：当服务器启动后，当Web浏览器端发送过来一个页面请求时，Web服务器先判断是否是JSP页面请求。如果该页面只是一般的HTML/XML页面请求，则直接将HTML/XML页面代码传给Web浏览器端。如果请求的页面是JSP页面，则由JSP引擎检查该JSP页面，如果该页面是第一次被请求、或不是第一次被请求但已被修改，则JSP引擎将此JSP页面代码转换成Servlet代码，然后JSP引擎调用服务器端的Java编译器javac.exe对Servlet代码进行编译，把它变成字节码(.class)文件，然后再调用JAVA虚拟机执行该字节码文件，然后将执行结果传给Web浏览器端。如果该JSP页面不是第一次被请求，且没有被修改过，则直接由JSP引擎调用JAVA虚拟机执行已编译过的字节码.class文件，然后将结果传送Web浏览器端。

## jsp 有哪些内置对象？作用分别是什么？

Page，pageContext，request，response，session，application，out，config，exception

Page指的是JSP被翻译成Servlet的对象的引用.

pageContext对象可以用来获得其他8个内置对象,还可以作为JSP的域范围对象使用.pageContext中存的值是当前的页面的作用范围》.

request代表的是请求对象,可以用于获得客户机的信息,也可以作为域对象来使用，使用request保存的数据在一次请求范围内有效。

Response是响应对象,代表的是从服务器向浏览器响应数据.

Session代表的是一次会话，可以用于保存用户的私有的信息,也可以作为域对象使用，使用session保存的数据在一次会话范围有效.

Application：代表整个应用范围,使用这个对象保存的数据在整个web应用中都有效。

Out:JSPWriter是用于向页面输出内容的对象.

Config：指的是ServletConfig用于JSP翻译成Servlet后 获得Servlet的配置的对象. 该对象用于存取servlet实例的初始化参数。

Exception:在页面中设置isErrorPage=”true”，即可使用，是Throwable的引用.用来获得页面的错误信息。

## 说一下 jsp 的 4 种作用域？

page、request、session、application

这四大作用域，其实就是其九大内置对象中的四个，为什么说他们也是JSP的四大作用域呢？因为这四个对象都能存储数据，比如request.setAttribute()注意和request.setParameter()区分开来，一个是存储在域中的、一个是请求参数，session.setAttribute()、application其实就是SerlvetContext，自然也有setAttribute()方法。而page作用域的操作就需要依靠pageContext对象来进行了。在上面我们也有提到JSP的四大作用域，

　　　page作用域：代表变量只能在当前页面上生效

　　　request：代表变量能在一次请求中生效，一次请求可能包含一个页面，也可能包含多个页面，比如页面A请求转发到页面B

session：代表变量能在一次会话中生效，基本上就是能在web项目下都有效，session的使用也跟cookie有很大的关系。一般来说，只要浏览器不关闭，cookie就会一直生效，cookie生效，session的使用就不会受到影响。

　　　application：代表变量能一个应用下(多个会话)，在服务器下的多个项目之间都能够使用。比如baidu、wenku等共享帐号。

## JSP共有以下6种基本动作

jsp:include：在页面被请求的时候引入一个文件。

jsp:useBean：寻找或者实例化一个JavaBean。

jsp:setProperty：设置JavaBean的属性。

jsp:getProperty：输出某个JavaBean的属性。

jsp:forward：把请求转到一个新的页面。

jsp:plugin：根据浏览器类型为Java插件生成OBJECT或EMBED标记

## session 和 cookie 有什么区别？

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗

考虑到安全应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能

考虑到减轻服务器性能方面，应当使用COOKIE。

4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

## 说一下 session 的工作原理？

本质上：session技术就是一种基于后端有别于数据库的临时存储数据的技术.

为什么要有session?

主要的一个原因就是HTTP的无状态性.因为HTTP的无状态性，所以我们没有办法在HTTP发送请求的时候知道当前用户的状态，也就是比如说，当前是哪个用户的之类的这种信息，所以这个时候我们需要session来标识当前的状态.

要了解Session的底层工作原理。我们还是先看在一个会话过程中，同一个浏览器在访问多个web资源的情况好了，大致分为以下几个步骤：

1，浏览器访问某个Servlet，这时如果服务器要从请求对象中获取Session对象（第一次获取也是创建），那么服务器会为这个Session对象创建一个id：JSESSIONID

2，同时在对浏览器的响应过程中，这个Session会将JSESSIONID这个id以Cookie形式回送给客户端浏览器，记住，这时候Cookie服务器没有设置有效时间，因此是存在浏览器的缓存中，而不是在硬盘文件。

3，当用户继续在这个会话过程中访问其他Servlet，这时候这个Servlet再从请求对象中获取Session对象，注意这时候获取Session对象是从浏览器发来的请求中查询是否有名为JSESSIONID的这个Cookie，如果有，那么这个Session就不用再创建，而是直接根据查询服务器中这个相同JSESSIONID值的Session，换句话说就可以取得之前存在这个Session中的数据。

总结来说，Session是基于Cookie的。

（注：cookie并不是万能的，Session首先是依据cookie，但是有时候cookie不能用，这时候Session会查询发来请求的URL地址是否有JSESSIONID。）

如果我们将浏览器关闭，由于这个cookie没有设置“setMaxAge”，因此这个cookie只存在于浏览器的缓冲，浏览器关闭即被销毁。如果想使关闭浏览器之后，Session还能存在，我们就要人为的覆盖这个Session的cookie，并设置覆盖cookie的有效时间和有效路径。而这个cookie的值，也就是JSESSIONID的值，可以通过Session的getId()方法得到。

## 如果客户端禁止 cookie 能实现 session 还能用吗？

<https://blog.csdn.net/qq_25281057/article/details/52424811>

可以，有两种方法：

1. 主动将JSESSIONID写入cookie中；
2. 将JSESSIONID写入url中。

## 什么是 CSRF 攻击，如何避免？

CSRF（Cross-site request forgery）跨站请求伪造，这个词还是不了解，不用担心，我们先讲一个故事，一个真实的故事。那是在2009年，一名黑客利用CSRF通过Gmail的成功攻击成功窃取了好莱坞明星Vanessa Hudgens的邮箱。攻击很简单，她给她发了一封电子邮件，内容是图片，但这张照片有点特别。它是这样写的：当用户点击邮件加载图片时，他实际上访问了这样的链接：https：//mail.google.com/mail？ui = 2＆fw = true＆fwe = my @ email.com，这个链接的功能是Gmail设置邮件转发，因此目标邮箱中的所有邮件都被转发到黑客指定的邮箱。有很多这样的例子。只需构造正确的Get链接，让浏览器发送Get请求。由于用户已经登录，因此它将携带正确的cookie并直接访问链接以实现相关功能。

在了解了原理之后，如何破解黑客的攻击？解决方案是向表单添加字段标记。黑客无法猜出已经改变的令牌，所以即使你添加一个UUID令牌，将它保存到服务器会话，黑客也不可能猜出随机令牌，就无法构造出能实现功能的链接了。

## 什么是 XSS 攻击，如何避免？

<https://baijiahao.baidu.com/s?id=1618267672561552800&wfr=spider&for=pc>

首先，这个词实际上是CSS（Cross Site Scripting），但它与CSS同名。所以名字是XSS。**跨站点脚本攻击**类似于上面提到的CSRF。实际上，原则是将一段JavaScript代码注入网页。然后当其他用户访问该页面时，他们将运行黑客编写的JS代码来实现一些帐户控制。经典故事是：2011年6月28日，新浪微博被XSS攻击，大量用户自动转发微博、私信。自动关注用户，大量用户被莫名其妙地控制。因为可以使用JS代码代替用户单击按钮发送请求，所以损坏非常大。反射XSS攻击：搜索模块越常见，例如：https：//www.lucklnk.com/Search？wd =攻击，当搜索链接正常时，应将wd参数传递给搜索词，并且然后在页面上显示当前搜索词，您可以使用此门户传递一些不同的数据，例如，我将其更改为：https：//www.lucklnk.com/Search？wd =，页面将是使用JS代码注入类似地，也可以在没有用户感知的情况下在站点中插入其他站点的js文件，从而实现诸如控制用户和窃取cookie之类的操作，但是这样的链接必须由用户点击，并且将不要让所有用户都被招募。

存储XSS攻击：最后一次反射XSS攻击只能攻击点击该链接的某些用户。存储类型XSS可能会对所有用户带来攻击。经典的是用户签名栏。这个地方允许用户自定义。内容将存储在数据库中。当其他用户浏览他时，它将从数据库加载。如果恶意用户在其签名栏中注入了js参考，那么浏览他的所有人都将被Js控制，存储类型XSS的危险可能危及整个站的用户，非常危险。

在了解了原理之后，如何破解黑客的攻击？有很多方法，例如：关键字判断当有一个关键字如脚本、src来代替损坏；返回内容时进行转码，转码尖括号采用Unicode编码格式。

## throw 和 throws 的区别？

throws是用来声明一个方法可能抛出的所有异常信息，throws是将异常声明但是不处理，而是将异常往上传，谁调用我就交给谁处理。而throw则是指抛出的一个具体的异常类型。

## final、finally、finalize 有什么区别？

final可以用来修饰类，方法和变量（成员变量或局部变量）。

当用final修饰类的时，表明该类不能被其他类所继承。当我们需要让一个类永远不被继承，此时就可以用final修饰，但要注意：

**final类中所有的成员方法都会隐式的定义为final方法。**

使用final方法的原因主要有两个：

　　 (1) 把方法锁定，以防止继承类对其进行更改。

　　 (2) 效率，在早期的java版本中，会将final方法转为内嵌调用。但若方法过于庞大，可能在性能上不会有多大提升。因此在最近版本中，不需要final方法进行这些优化了。

final方法意味着“最后的、最终的”含义，即此方法不能被重写。

**注意：若父类中final方法的访问权限为private，将导致子类中不能直接继承该方法，因此，此时可以在子类中定义相同方法名的函数，此时不会与重写final的矛盾，而是在子类中重新地定义了新方法。**

final成员变量表示常量，只能被赋值一次，赋值后其值不再改变。类似于C++中的const。

当final修饰一个基本数据类型时，表示该基本数据类型的值一旦在初始化后便不能发生变化；如果final修饰一个引用类型时，则在对其初始化之后便不能再让其指向其他对象了，但该引用所指向的对象的内容是可以发生变化的。本质上是一回事，因为引用的值是一个地址，final要求值，即地址的值不发生变化。

**final修饰一个成员变量（属性），必须要显示初始化。这里有两种初始化方式**，一种是在变量声明的时候初始化；第二种方法是在声明变量的时候不赋初值，但是要在这个变量所在的类的所有的构造函数中对这个变量赋初值。

　　当函数的参数类型声明为final时，说明该参数是只读型的。即你可以读取使用该参数，但是无法改变该参数的值。

在java中，String被设计成final类，那为什么平时使用时，String的值可以被改变呢？

　　字符串常量池是java堆内存中一个特殊的存储区域，当我们建立一个String对象时，假设常量池不存在该字符串，则创建一个，若存在则直接引用已经存在的字符串。当我们对String对象值改变的时候，例如 String a="A"; a="B" 。a是String对象的一个引用（我们这里所说的String对象其实是指字符串常量），当a=“B”执行时，并不是原本String对象("A")发生改变，而是创建一个新的对象("B")，令a引用它。

**finally**作为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。（×）（这句话其实存在一定的问题）。

只有与finally对应的try语句块得到执行的情况下，finally语句块才可能会执行。以上两种情况在执行try语句块之前已经返回或抛出异常，所以try对应的finally语句并没有执行。但是，在某些情况下，即使try语句执行了，finally语句也不一定执行。

当一个线程在执行 try 语句块或者 catch 语句块时被打断（interrupted）或者被终止（killed），与其相对应的 finally 语句块可能不会执行。还有更极端的情况，就是在线程运行 try 语句块或者 catch 语句块时，突然死机或者断电，finally 语句块肯定不会执行了。可能有人认为死机、断电这些理由有些强词夺理，没有关系，我们只是为了说明这个问题。

　finalize()是在java.lang.Object里定义的，也就是说每一个对象都有这么个方法。这个方法在gc启动，该对象被回收的时候被调用。其实gc可以回收大部分的对象（凡是new出来的对象，gc都能搞定，一般情况下我们又不会用new以外的方式去创建对象），所以一般是不需要程序员去实现finalize的。

特殊情况下，需要程序员实现finalize，当对象被回收的时候释放一些资源，比如：一个socket链接，在对象初始化时创建，整个生命周期内有效，那么就需要实现finalize，关闭这个链接。

　　使用finalize还需要注意一个事，调用super.finalize();

一个对象的finalize()方法只会被调用一次，而且finalize()被调用不意味着gc会立即回收该对象，所以有可能调用finalize()后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会调用finalize()，产生问题。 所以，推荐不要使用finalize()方法，它跟析构函数不一样。

## try-catch-finally 中哪个部分可以省略？

try语句后面是可以省略catch语句的，但是必须有finally语句；

也可以省略finally语句，但是必须要有catch语句；

也就是说try语句后面必须要有一个别的语句跟在后面。

## try-catch-finally 中，如果 catch 中 return 了，finally 还会执行吗？

会执行。

1、不管有没有异常，finally中的代码都会执行

2、当try、catch中有return时，finally中的代码依然会继续执行

3、finally是在return后面的表达式运算之后执行的，此时并没有返回运算之后的值，而是把值保存起来，不管finally对该值做任何的改变，返回的值都不会改变，依然返回保存起来的值。也就是说方法的返回值是在finally运算之前就确定了的。

4、如果return的数据是引用数据类型，而在finally中对该引用数据类型的属性值的改变起作用，try中的return语句返回的就是在finally中改变后的该属性的值。

5、finally代码中最好不要包含return，程序会提前退出，也就是说返回的值不是try或catch中的值

先执行try中的语句，包括return后面的表达式，

有异常时,先执行catch中的语句，包括return后面的表达式,

然后执行finally中的语句,如果finally里面有return语句，会提前退出，

最后执行try中的return，有异常时执行catch中的return。

在执行try、catch中的return之前一定会执行finally中的代码（如果finally存在），如果finally中有return语句，就会直接执行finally中的return方法，所以finally中的return语句一定会被执行的。编译器把finally中的return语句标识为一个warning.

## 常见的异常类有哪些？Error和Exception的区别？

Java中的异常分为两大类：

1. Checked Exception（非Runtime Exception）

这种异常的特点是Java编译器会检查它，也就是说，当程序中可能出现这类异常，要么用try-catch语句捕获它，要么用throws子句声明抛出它，否则编译不会通过。

IOException、SQLException等以及用户自定义的Exception异常。

1. Unchecked Exception（Runtime Exception和Error）

这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的，程序应该从逻辑角度尽可能避免这类异常的发生。

Error（错误）:是程序无法处理的错误，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

NullPointerException(空指针异常)、

IndexOutOfBoundsException(下标越界异常)等。

Error和Exception的区别：

首先，Error类和Exception类都是继承Throwable类

Error（错误）是系统中的错误，程序员是不能改变的和处理的，是在程序编译时出现的错误，只能通过修改程序才能修正。一般是指与虚拟机相关的问题，如系统崩溃，虚拟机错误，内存空间不足，方法调用栈溢等。对于这类错误的导致的应用程序中断，仅靠程序本身无法恢复和和预防，遇到这样的错误，建议让程序终止。

Exception（异常）表示程序可以处理的异常，可以捕获且可能恢复。遇到这类异常，应该尽可能处理异常，使程序恢复运行，而不应该随意终止异常。

## http 响应码 301 和 302 代表的是什么？有什么区别？

301，302 都是HTTP状态的编码，都代表着某个URL发生了转移，不同之处在于：   
301 redirect: 301 代表永久性转移(Permanently Moved)。  
302 redirect: 302 代表暂时性转移(Temporarily Moved )。

**http状态返回代码 1xx（临时响应）**  
表示临时响应并需要请求者继续执行操作的状态代码。

**http状态返回代码 2xx （成功）**  
表示成功处理了请求的状态代码。

**http状态返回代码 3xx （重定向）**  
表示要完成请求，需要进一步操作。通常，这些状态代码用来重定向。

**http状态返回代码 4xx（请求错误）**  
这些状态代码表示请求可能出错，妨碍了服务器的处理。

**http状态返回代码 5xx（服务器错误）**  
这些状态代码表示服务器在尝试处理请求时发生内部错误。这些错误可能是服务器本身的错误，而不是请求出错。

## forward 和 redirect 的区别？

1.从地址栏显示来说

　　forward是服务器请求资源,服务器直接访问目标地址的URL,把那个URL的响应内容读取过来,然后把这些内容再发给浏览器.浏览器根本不知道服务器发送的内容从哪里来的,所以它的地址栏还是原来的地址.

　　redirect是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求那个地址.所以地址栏显示的是新的URL.所以redirect等于客户端向服务器端发出两次request，同时也接受两次response。

2.从数据共享来说

　　forward:转发页面和转发到的页面可以共享request里面的数据.

　　redirect:不能共享数据.

　　redirect不仅可以重定向到当前应用程序的其他资源,还可以重定向到同一个站点上的其他应用程序中的资源,甚至是使用绝对URL重定向到其他站点的资源.forward,方法只能在同一个Web应用程序内的资源之间转发请求.forward 是服务器内部的一种操作.redirect 是服务器通知客户端,让客户端重新发起请求.所以,你可以说 redirect是一种间接的请求, 但是你不能说"一个请求是属于forward还是redirect "

3.从运用地方来说

　　forward:一般用于用户登陆的时候,根据角色转发到相应的模块.

redirect:一般用于用户注销登陆时返回主页面和跳转到其它的网站等.

## 简述 tcp 和 udp的区别？

TCP 是面向连接的，UDP 是面向无连接的

UDP程序结构较简单

TCP 是面向字节流的，UDP 是基于数据报的

TCP 保证数据正确性，UDP 可能丢包

TCP 保证数据顺序，UDP 不保证

## tcp/ip协议的五层模型？

应用层：HTTP、FTP、SMTP、Telnet等

传输层：TCP/IP

网络层：

数据链路层：

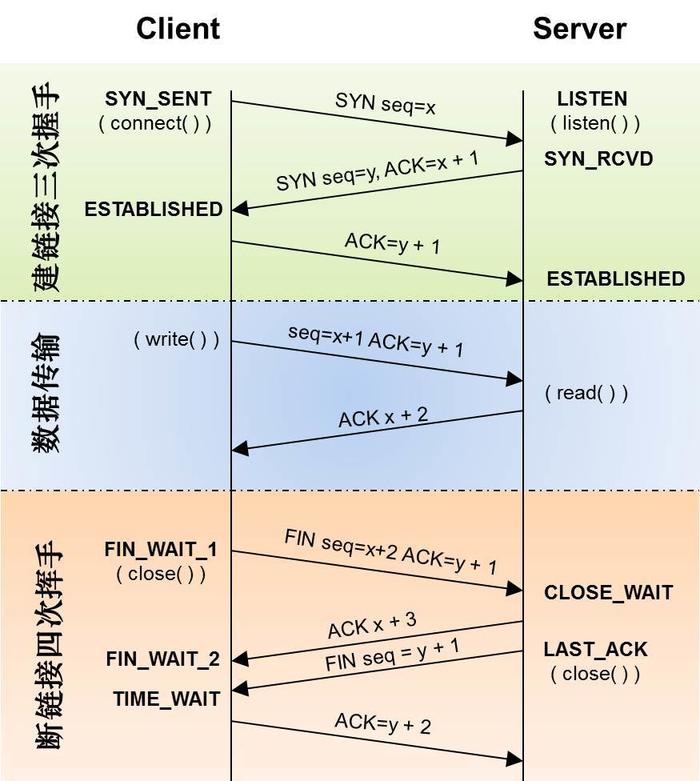
物理层：网线、双绞线、网卡等

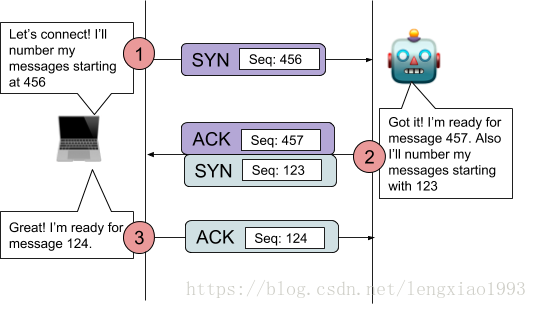
## tcp 为什么要三次握手，两次不行吗？为什么？

<https://blog.csdn.net/lengxiao1993/article/details/82771768>

为了实现可靠数据传输， TCP 协议的通信双方， 都必须维护一个序列号， 以标识发送出去的数据包中， 哪些是已经被对方收到的。 三次握手的过程即是通信双方相互告知序列号起始值， 并确认对方已经收到了序列号起始值的必经步骤。

如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认。





《计算机网络》的内容：防止已失效的连接请求又传送到服务器端，因而产生错误。

## 说一下 tcp 粘包是怎么产生的？

<https://www.cnblogs.com/qiaoconglovelife/p/5733247.html>

1 什么是粘包现象

　　TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。

2 为什么出现粘包现象

　　（1）发送方原因

　　我们知道，TCP默认会使用Nagle算法。而Nagle算法主要做两件事：1）只有上一个分组得到确认，才会发送下一个分组；2）收集多个小分组，在一个确认到来时一起发送。

　　所以，正是Nagle算法造成了发送方有可能造成粘包现象。

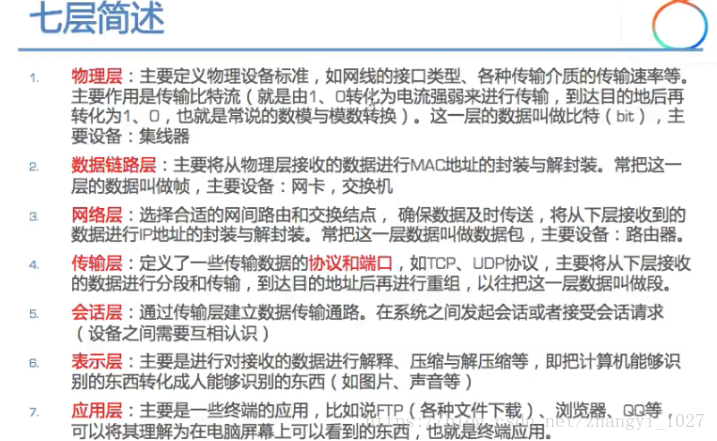
　　（2）接收方原因

　　TCP接收到分组时，并不会立刻送至应用层处理，或者说，应用层并不一定会立即处理；实际上，TCP将收到的分组保存至接收缓存里，然后应用程序主动从缓存里读收到的分组。这样一来，如果TCP接收分组的速度大于应用程序读分组的速度，多个包就会被存至缓存，应用程序读时，就会读到多个首尾相接粘到一起的包。

## OSI 的七层模型都有哪些？

OSI模型，即开放式通信系统互联参考模型，是国际标准化组织提出的一个试图是各种计算机或者通信系统在世界范围内互联为网络的标准框架。整个模型分为七层，

物理层，数据链路层，网络层，传输层，会话层，表示层，应用层。



## get 和 post 请求有哪些区别？

<https://www.cnblogs.com/logsharing/p/8448446.html>

GET和POST还有一个重大区别，简单的说：

**GET产生一个TCP数据包；POST产生两个TCP数据包。**

长的说：

对于GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200（返回数据）；

而对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

也就是说，GET只需要汽车跑一趟就把货送到了，而POST得跑两趟，第一趟，先去和服务器打个招呼“嗨，我等下要送一批货来，你们打开门迎接我”，然后再回头把货送过去。

因为POST需要两步，时间上消耗的要多一点，看起来GET比POST更有效。因此Yahoo团队有推荐用GET替换POST来优化网站性能。但这是一个坑！跳入需谨慎。为什么？

1. GET与POST都有自己的语义，不能随便混用。

2. 据研究，在网络环境好的情况下，发一次包的时间和发两次包的时间差别基本可以无视。而在网络环境差的情况下，两次包的TCP在验证数据包完整性上，有非常大的优点。

3. 并不是所有浏览器都会在POST中发送两次包，Firefox就只发送一次。

## 如何实现跨域？

<https://www.cnblogs.com/chenshishuo/p/4919224.html>

<https://www.cnblogs.com/yongshaoye/p/7423881.html>

跨域是指从一个域名的网页去请求另一个域名的资源。比如从www.baidu.com 页面去请求 www.google.com 的资源。但是一般情况下不能这么做，它是由浏览器的同源策略造成的，是浏览器对[JavaScript](http://lib.csdn.net/base/javascript)施加的安全限制。跨域的严格一点的定义是：只要 协议，域名，端口有任何一个的不同，就被当作是跨域

浏览器执行[javascript](http://lib.csdn.net/base/javascript)脚本时，会检查这个脚本属于哪个页面，如果不是同源页面，就不会被执行。

对于端口和协议的不同，只能通过后台来解决。

## 说一下 JSONP 实现原理？

JSONP是JSON with Padding（填充式json）的简写，是应用JSON的一种新方法，只不过是被包含在函数调用中的JSON，例如：

callback({"name","trigkit4"});

JSONP由两部分组成：回调函数和数据。回调函数是当响应到来时应该在页面中调用的函数，而数据就是传入回调函数中的JSON数据。

JSONP的原理：通过script标签引入一个js文件，这个js文件载入成功后会执行我们在url参数中指定的函数，并且会把我们需要的json数据作为参数传入。所以jsonp是需要服务器端的页面进行相应的配合的。（即用javascript动态加载一个script文件，同时定义一个callback函数给script执行而已。）

在js中，我们直接用XMLHttpRequest请求不同域上的数据时，是不可以的。但是，在页面上引入不同域上的js脚本文件却是可以的，jsonp正是利用这个特性来实现的。

## 说一下你熟悉的设计模式？

责任链模式

模板方法模式

装饰者模式

。。。

## 简单工厂和抽象工厂有什么区别？

<https://www.cnblogs.com/toutou/p/4899388.html>

<https://www.cnblogs.com/zailushang1996/p/8601808.html>

<https://blog.csdn.net/llussize/article/details/80276627>

<https://blog.csdn.net/auuea/article/details/84673570>这篇讲的比较清楚了。

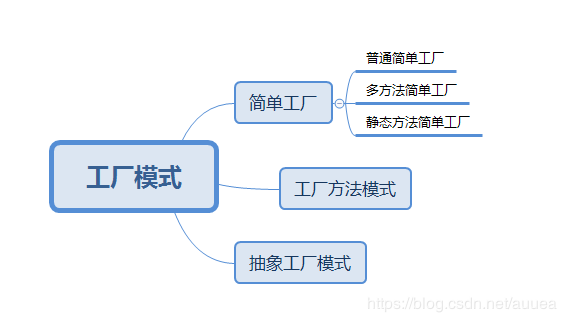
妈的完全看不出啥区别。。

工厂模式主要是为创建对象提供了接口。工厂模式按照《Java与模式》中的提法分为三类：

1. 简单工厂模式(Simple Factory)

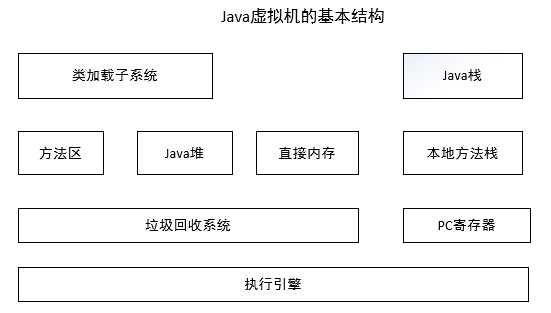
2. 工厂方法模式(Factory Method)

3. 抽象工厂模式(Abstract Factory)



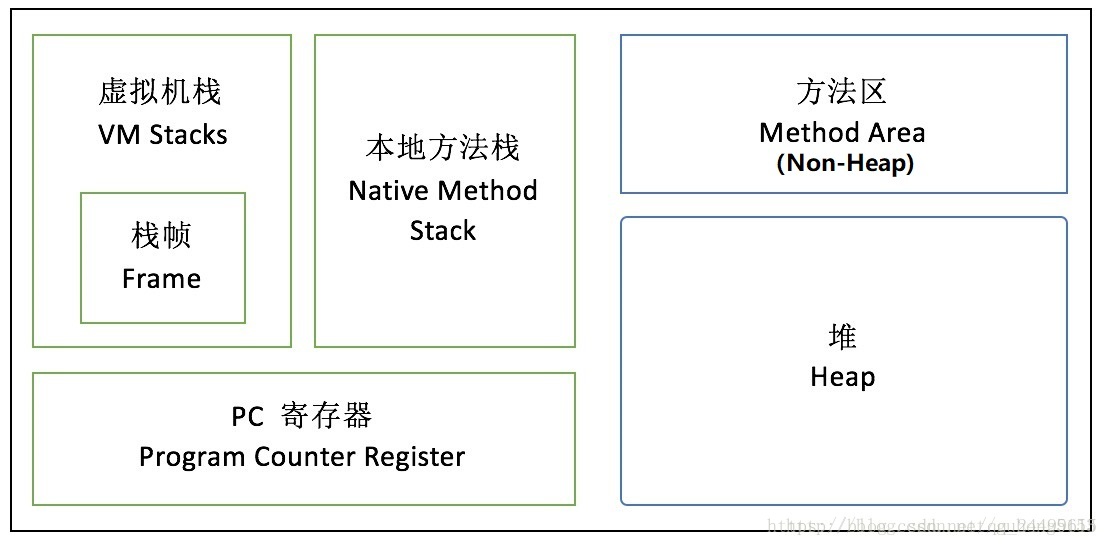
## 说一下 jvm 的主要组成部分？及其作用？

<https://www.cnblogs.com/zwbg/p/6194470.html>



## 说一下 jvm 运行时数据区？

JVM内存区域分为五个部分，分别是堆，方法区，虚拟机栈，本地方法栈，程序计数器。

1.堆。 **堆是Java对象的存储区域**，任何用new字段分配的Java对象实例和数组，都被分配在堆上，Java堆可使用-Xms -Xmx进行内存控制，**值得一提的是从JDK1.7版本之后，运行时常量池从方法区移到了堆上。**

2.方法区。**它用于存储已被虚拟机加载的类信息，常量，静态变量，即时编译器编译后的代码等数据**，方法区在JDK1.7版本及以前被称为永久代，从JDK1.8永久代被移除。

3.虚拟机栈。虚拟机栈中执行每个方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

4.本地方法栈。与虚拟机栈发挥的作用相似，相比于虚拟机栈为Java方法服务，本地方法栈为虚拟机使用的Native方法服务，执行每个本地方法的时候，都会创建一个栈帧用于存储局部变量表，操作数栈，动态链接，方法出口等信息。

5.程序计数器。指示Java虚拟机下一条需要执行的字节码指令。 **这是JVM规范中唯一一个没有规定会导致OutOfMemory（内存泄露，下文简称OOM）的区域**。

以上五个区域是Java虚拟机内存划分情况，其中方法区和堆被JVM中多个线程共享，比如类的静态常量就被存放在方法区，供类对象之间共享，虚拟机栈，本地方法栈，pc寄存器是每个线程独立拥有的，不会与其他线程共享。

所以Java在通过new创建一个类对象实例的时候，一方面会在虚拟机栈中创建一个该对象的引用，另一方面会在堆上创建类对象的实例，然后将对象引用指向该对象的实例。对象引用存放在每一个方法对应的栈帧中。

## 说一下堆栈的区别？

栈内存:栈内存首先是一片内存区域，存储的都是局部变量，凡是定义在方法中的都是局部变量（方法外的是全局变量），for循环内部定义的也是局部变量，是先加载函数才能进行局部变量的定义，所以方法先进栈，然后再定义变量，变量有自己的作用域，一旦离开作用域，变量就会被释放。栈内存的更新速度很快，因为局部变量的生命周期都很短。

       堆内存:存储的是数组和对象（其实数组就是对象），凡是new建立的都是在堆中，堆中存放的都是实体（对象），实体用于封装数据，而且是封装多个（实体的多个属性），如果一个数据消失，这个实体也没有消失，还可以用，所以堆是不会随时释放的，但是栈不一样，栈里存放的都是单个变量，变量被释放了，那就没有了。堆里的实体虽然不会被释放，但是会被当成垃圾，Java有垃圾回收机制不定时的收取。

所以堆与栈的区别很明显：

            1.栈内存存储的是局部变量而堆内存存储的是实体；

            2.栈内存的更新速度要快于堆内存，因为局部变量的生命周期很短；

            3.栈内存存放的变量生命周期一旦结束就会被释放，而堆内存存放的实体会被垃圾回收机制不定时的回收。

## 队列和栈是什么？有什么区别？

队列和栈是两种不同的数据结构。它们有以下区别：

（1）操作的名称不同。队列的插入称为入队，队列的删除称为出队。栈的插入称为进栈，栈的删除称为出栈。

（2）可操作的方式不同。队列是在队尾入队，队头出队，即两边都可操作。而栈的进栈和出栈都是在栈顶进行的，无法对栈底直接进行操作。

（3）操作的方法不同。队列是先进先出（FIFO），即队列的修改是依先进先出的原则进行的。新来的成员总是加入队尾（不能从中间插入），每次离开的成员总是队列头上（不允许中途离队）。而栈为后进先出（LIFO）,即每次删除（出栈）的总是当前栈中最新的元素，即最后插入（进栈）的元素，而最先插入的被放在栈的底部，要到最后才能删除。

## 什么是双亲委派模型？

**双亲委派模型**:如果一个类加载器收到了加载某个类的请求,则该类加载器并不会去加载该类,而是把这个请求委派给父类加载器,每一个层次的类加载器都是如此,因此所有的类加载请求最终都会传送到顶端的启动类加载器;只有当父类加载器在其搜索范围内无法找到所需的类,并将该结果反馈给子类加载器,子类加载器会尝试去自己加载.

**使用双亲委派模型原因:**java类随着其类加载器一起具备了一种带有优先级的层次关系.例如 java.lang.Object,无论哪一个类加载器要加载该类,最终都是委托给处于顶端的启动类加载器,因此object在程序的各种类加载器环境中都是同一个类.相反如果没有使用双亲委派模型,那么假如用户自定义了一个称为java.lang.Object的类,并放在classPath中,那么系统将会出现多个不同的Object类,则java类型体系中最基础的行为都无法保证.

**为什么需要自定义类加载器:**

1)加密:java代码可以轻易的被反编译,如果你需要对你的代码进行加密以防止反编译,可以先将编译后的代码用加密算法加密,类加密后就不能再使用java自带的类加载器了,这时候就需要自定义类加载器.

2)从非标准的来源加载代码:字节码是放在数据库,甚至是云端,就可以自定义类加载器,从指定来源加载类.

使用双亲委托机制的好处是：能够有效确保一个类的全局唯一性，当程序中出现多个限定名相同的类时，类加载器在执行加载时，始终只会加载其中的某一个类。

## 说一下类加载的执行过程？

<https://www.cnblogs.com/xiaoxian1369/p/5498817.html>

类从被加载到JVM中开始，到卸载为止，整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载七个阶段。

其中类加载过程包括加载、验证、准备、解析和初始化五个阶段。

1、加载

简单的说，类加载阶段就是由类加载器负责根据一个类的全限定名来读取此类的二进制字节流到JVM内部，并存储在运行时内存区的方法区，然后将其转换为一个与目标类型对应的java.lang.Class对象实例（Java虚拟机规范并没有明确要求一定要存储在堆区中，只是hotspot选择将Class对戏那个存储在方法区中），这个Class对象在日后就会作为方法区中该类的各种数据的访问入口。

2、链接

链接阶段要做的是将加载到JVM中的二进制字节流的类数据信息合并到JVM的运行时状态中，经由验证、准备和解析三个阶段。

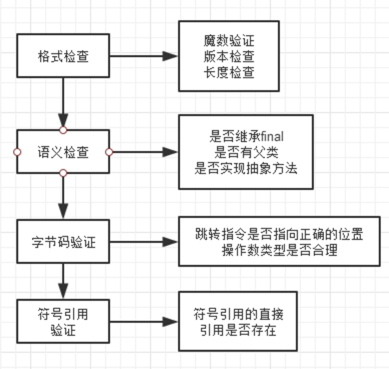
1）、验证

验证类数据信息是否符合JVM规范，是否是一个有效的字节码文件，验证内容涵盖了类数据信息的格式验证、语义分析、操作验证等。

格式验证：验证是否符合class文件规范

语义验证：检查一个被标记为final的类型是否包含子类；检查一个类中的final方法视频被子类进行重写；确保父类和子类之间没有不兼容的一些方法声明（比如方法签名相同，但方法的返回值不同）

操作验证：在操作数栈中的数据必须进行正确的操作，对常量池中的各种符号引用执行验证（通常在解析阶段执行，检查是否通过富豪引用中描述的全限定名定位到指定类型上，以及类成员信息的访问修饰符是否允许访问等）



2）、准备

为类中的所有静态变量分配内存空间，并为其设置一个初始值（由于还没有产生对象，实例变量不在此操作范围内）  
被final修饰的静态变量，会直接赋予原值；类字段的字段属性表中存在ConstantValue属性，则在准备阶段，其值就是ConstantValue的值

3）、解析

将常量池中的符号引用转为直接引用（得到类或者字段、方法在内存中的指针或者偏移量，以便直接调用该方法），这个可以在初始化之后再执行。  
可以认为是一些静态绑定的会被解析，动态绑定则只会在运行是进行解析；静态绑定包括一些final方法(不可以重写),static方法(只会属于当前类)，构造器(不会被重写)

3、初始化

将一个类中所有被static关键字标识的代码统一执行一遍，如果执行的是静态变量，那么就会使用用户指定的值覆盖之前在准备阶段设置的初始值；如果执行的是static代码块，那么在初始化阶段，JVM就会执行static代码块中定义的所有操作。

所有类变量初始化语句和静态代码块都会在编译时被前端编译器放在收集器里头，存放到一个特殊的方法中，这个方法就是<clinit>方法，即类/接口初始化方法。该方法的作用就是初始化一个中的变量，使用用户指定的值覆盖之前在准备阶段里设定的初始值。任何invoke之类的字节码都无法调用<clinit>方法，因为该方法只能在类加载的过程中由JVM调用。

如果父类还没有被初始化，那么优先对父类初始化，但在<clinit>方法内部不会显示调用父类的<clinit>方法，由JVM负责保证一个类的<clinit>方法执行之前，它的父类<clinit>方法已经被执行。

JVM必须确保一个类在初始化的过程中，如果是多线程需要同时初始化它，仅仅只能允许其中一个线程对其执行初始化操作，其余线程必须等待，只有在活动线程执行完对类的初始化操作之后，才会通知正在等待的其他线程。

## 怎么判断对象是否可以被回收？

可以用引用计数法和**根搜索算法**。

这个算法的关键就在于GC Roots的定义，教科书中给出了四种作为GC Roots的对象：

首先第一种是虚拟机栈中的引用的对象，我们在程序中正常创建一个对象，对象会在堆上开辟一块空间，同时会将这块空间的地址作为引用保存到虚拟机栈中，如果对象生命周期结束了，那么引用就会从虚拟机栈中出栈，因此如果在虚拟机栈中有引用，就说明这个对象还是有用的，这种情况是最常见的。

第二种是我们在类中定义了全局的静态的对象，也就是使用了static关键字，由于虚拟机栈是线程私有的，所以这种对象的引用会保存在共有的方法区中，显然将方法区中的静态引用作为GC Roots是必须的。

第三种便是常量引用，就是使用了static final关键字，由于这种引用初始化之后不会修改，所以方法区常量池里的引用的对象也应该作为GC Roots。

最后一种是在使用JNI技术时，有时候单纯的Java代码并不能满足我们的需求，我们可能需要在Java中调用C或C++的代码，因此会使用native方法，JVM内存中专门有一块本地方法栈，用来保存这些对象的引用，所以本地方法栈中引用的对象也会被作为GC Roots。

## java 中都有哪些引用类型？

在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为

1. 强引用 Strong Reference

2. 软引用 Soft Reference

3. 弱引用 Weak Reference

4. 虚引用 Phantom Reference

**强引用**

Object obj = new Object();

代码中普遍存在的，像上述的引用。只要强引用还在，垃圾收集器永远不会回收掉被引用的对象。

**软引用**

用来描述一些还有用，但并非必须的对象。软引用所关联的对象，有在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围，并进行第二次回收。如果这次回收还是没有足够的内存，才会抛出内存异常。提供了 SoftReference 类实现软引用。

**弱引用**

描述非必须的对象，强度比软引用更弱一些，被弱引用关联的对象，只能生存到下一次垃圾收集发生前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。提供了 WeakReference 类来实现弱引用。

**虚引用**

一个对象是否有虚引用，完全不会对其生存时间够成影响，也无法通过虚引用来取得一个对象实例。为一个对象关联虚引用的唯一目的，就是希望在这个对象被收集器回收时，收到一个系统通知。java提供了 PhantomReference 类来实现虚引用。

## 说一下 jvm 有哪些垃圾回收算法？

<https://www.cnblogs.com/yulinfeng/p/7163052.html>

### 1.****标记-清除算法****

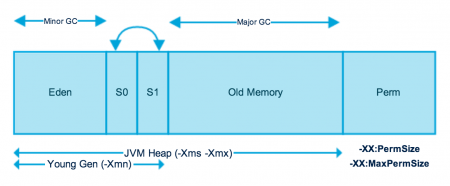
　　等待被回收对象的“标记”过程在上文已经提到过，如果在被标记后直接对对象进行清除，会带来另一个新的问题——**内存碎片化**。如果下次有比较大的对象实例需要在堆上分配较大的内存空间时，可能会出现无法找到足够的连续内存而不得不再次触发垃圾回收。

复制算法（Java堆中新生代的垃圾回收算法）

### 2.复制算法

此GC算法实际上解决了标记-清除算法带来的“内存碎片化”问题。首先还是先标记处待回收内存和不用回收的内存，下一步将不用回收的内存复制到新的内存区域，这样旧的内存区域就可以全部回收，而新的内存区域则是连续的。它的缺点就是会损失掉部分系统内存，因为你总要腾出一部分内存用于复制。

Java堆中的新生代就使用了GC复制算法。在新生代中又分为了三个区域：Eden 空间、To Survivor空间、From Survivor空间。不妨将注意力回到这张图的左边新生代部分：



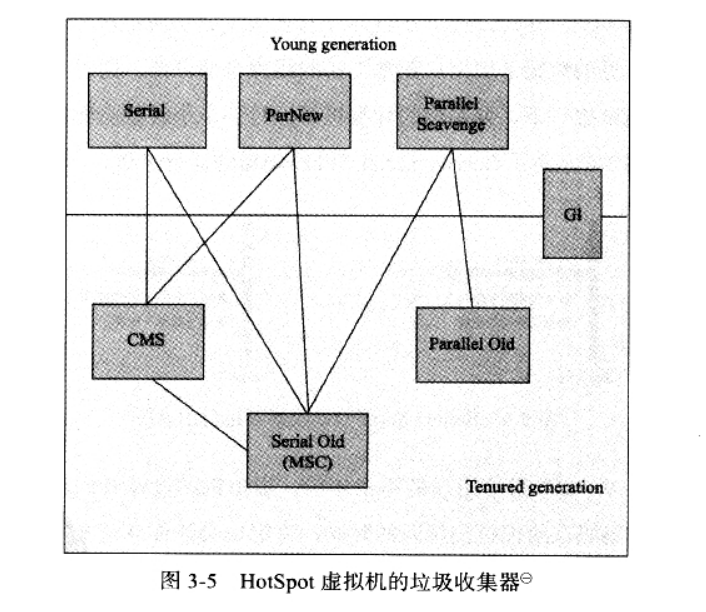
新的对象实例被创建的时候通常在Eden空间，发生在Eden空间上的GC称为Minor GC，当在新生代发生一次GC后，会将Eden和其中一个Survivor空间的内存复制到另外一个Survivor中，如果反复几次有对象一直存活，此时内存对象将会被移至老年代。可以看到新生代中Eden占了大部分，而两个Survivor实际上占了很小一部分。这是因为大部分的对象被创建过后很快就会被GC（这里也许运用了是二八原则）。

### 3.标记-压缩算法（或称为标记-整理算法，Java堆中老年代的垃圾回收算法）

对于新生代，大部分对象都不会存活，所以在新生代中使用复制算法较为高效，而对于老年代来讲，大部分对象可能会继续存活下去，如果此时还是利用复制算法，效率则会降低。标记-压缩算法首先还是“标记”，标记过后，将不用回收的内存对象压缩到内存一端，此时即可直接清除边界处的内存，这样就能避免复制算法带来的效率问题，同时也能避免内存碎片化的问题。老年代的垃圾回收称为“Major GC”。

## 说一下 jvm 有哪些垃圾回收器？

<https://www.cnblogs.com/chengxuyuanzhilu/p/7088316.html>



### **1.Serial收集器**

Serial收集器是最基本、发展历史最悠久的收集器。是单线程的收集器。它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集完成。

Serial收集器依然是虚拟机运行在Client模式下默认新生代收集器，对于运行在Client模式下的虚拟机来说是一个很好的选择。

### **2.ParNew收集器**

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The Worl、对象分配规则、回收策略等都与Serial收集器完全一样。

### **3.Parallel Scavenge（并行回收）收集器**

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器

该收集器的目标是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即 吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）

停顿时间越短就越适合需要与用户交互的程序，良好

的响应速度能提升用户体验，而高吞吐量则可用高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

Parallel Scavenge收集器提供两个参数用于精确控制吞吐量，分别是控制最大垃圾收起停顿时间的

-XX:MaxGCPauseMillis参数以及直接设置吞吐量大小的-XX:GCTimeRatio参数

Parallel Scavenge收集器还有一个参数：-XX:+UseAdaptiveSizePolicy。这是一个开关参数，当这个参数打开后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）等细节参数，只需要把基本的内存数据设置好（如-Xmx设置最大堆），然后使用MaxGVPauseMillis参数或GCTimeRation参数给虚拟机设立一个优化目标。

自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别

### **4.Serial Old 收集器**

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记整理算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。

如果在Server模式下，主要两大用途：

（1）在JDK1.5以及之前的版本中与Parallel Scavenge收集器搭配使用

（2）作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用

### **5.Parallel Old 收集器**

Parallel Old 是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器在1.6中才开始提供。

### **6.CMS收集器**

CMS(Concurrent Mark Sweep)收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务器的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求

CMS收集器是基于“标记-清除”算法实现的。它的运作过程相对前面几种收集器来说更复杂一些，整个过程分为4个步骤：

（1）初始标记

（2）并发标记

（3）重新标记

（4）并发清除

其中，初始标记、重新标记这两个步骤仍然需要“Stop The World”.

CMS收集器主要优点：并发收集，低停顿。

CMS三个明显的缺点：

（1）CMS收集器对CPU资源非常敏感。CPU个数少于4个时，CMS对于用户程序的影响就可能变得很大，为了应付这种情况，虚拟机提供了一种称为“增量式并发收集器”的CMS收集器变种。所做的事情和单CPU年代PC机操作系统使用抢占式来模拟多任务机制的思想

（2）CMS收集器无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。在JDK1.5的默认设置下，CMS收集器当老年代使用了68%的空间后就会被激活，这是一个偏保守的设置，如果在应用中蓝年代增长不是太快，可以适当调高参数-XX:CMSInitiatingOccupancyFraction的值来提高触发百分比，以便降低内存回收次数从而获取更好的性能，在JDK1.6中，CMS收集器的启动阀值已经提升至92%。

（3）CMS是基于“标记-清除”算法实现的收集器，手机结束时会有大量空间碎片产生。空间碎片过多，可能会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前出发FullGC。为了解决这个问题，CMS收集器提供了一个-XX:+UseCMSCompactAtFullCollection开关参数（默认就是开启的），用于在CMS收集器顶不住要进行FullGC时开启内存碎片合并整理过程，内存整理的过程是无法并发的，空间碎片问题没有了，但停顿时间变长了。虚拟机设计者还提供了另外一个参数-XX:CMSFullGCsBeforeCompaction,这个参数是用于设置执行多少次不压缩的Full GC后，跟着来一次带压缩的（默认值为0，标识每次进入Full GC时都进行碎片整理）

### **7. G1收集器**

G1收集器的优势：

（1）并行与并发

（2）分代收集

（3）空间整理 （标记整理算法，复制算法）

（4）可预测的停顿（G1处处理追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒，这几乎已经实现Java（RTSJ）的来及收集器的特征）

使用G1收集器时，Java堆的内存布局是整个规划为多个大小相等的独立区域（Region）,虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在真个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获取的空间大小以及回收所需要的时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大的Region（这也就是Garbage-First名称的又来）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽量可能高的灰机效率

G1 内存“化整为零”的思路

在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会遗漏。

如果不计算维护Remembered Set的操作，G1收集器的运作大致可划分为一下步骤：

（1）初始标记

（2）并发标记

（3）最终标记

（4）筛选回收

## 详细介绍一下 CMS 垃圾回收器？

<https://www.jianshu.com/p/2329d1c43ceb>

CMS全称 Concurrent Mark Sweep，是一款并发的、使用标记-清除算法的垃圾回收器，

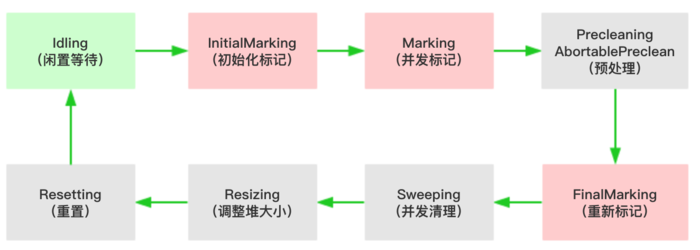
如果老年代使用CMS垃圾回收器，需要添加虚拟机参数 **-XX:+UseConcMarkSweepGC**。

使用场景：

**GC过程短暂停，适合对时延要求较高的服务**，用户线程不允许长时间的停顿。

缺点：

服务长时间运行，造成严重的内存碎片化。



### 1 初始标记(Initial Mark)：

**CMS算法中两个会触发Stop the World事件中的一个**，这个阶段会标记所有与GC Roots直接相关联的对象，以及被存活的青年代对象所直接引用的对象。

### 2并发标记(Concurrent Mark)：

并发标记，顾名思义，它是并发的执行标记任务的，这也就意味着GC在运行的过程中用户的应用线程并不会停止工作。该阶段GC收集器会从第一步“初始标记”中所标记出来的对象开始逐步遍历这些对象（与GCRoot直接相连或与存活的青年代对象直接相关联的对象）的所引用的对象，并将这些被引用的对象加上标记。

需要注意的是，这一步中，会漏掉一下老年代的存活对象，这是因为在并发的过程中，用户应用线程可能会对老年代的对象产生引用上的改变。某一些被改变的标记可能会被遗漏。

### 3 并发预清理(Concurrent Preclean)：

并发预清理是Java1.5被加入进来的。主要目的是减少重标记（Remark）步骤Stop-the-World的时间。这一步同样也是并发的，不会停止用户应用线程。在前面的并发标记中，一些引用被改变了。当某一块块（Card）中的对象引用发生改变时，JVM会标记这个空间为“脏块”（Dirty Card）。

在预清理阶段，JVM根据之前记录的这些“脏对象”重新标记了他们新的可达对象。这一步结束后空间重新进入clean状态。另外，一些必要的最终重标记之前的准备步骤也会在这一步做好。

预清理步骤将会不断重复一直到Eden区的占用量达到某个指定的阈值。设定这个阈值作为结束条件的原因主要是为了防止YoungGC产生的Stop-the-World和下一阶段的Remark同时产生，导致系统产生一个更长的停滞。设定了这个阈值之后基本可以保证Remark阶段可以在两次YoungGC之间进行。

### 4 重标记(Remark):

**这是CMS算法中第二个会触发Stop-the-World事件的步骤**，由于前一步是一个并发的步骤，预清理的速度可能会赶不上用户应用对对象改变的速度，所以需要一个Stop-the-World的暂停来完整的标记所有对象结束整个标记阶段。

通常CMS会在年轻代为空时来运行重标记阶段，以此避免一个接一个的Stop-the-World阶段。

### 5.并发清理(Concurrent Sweep):

这一阶段程序并发地工作，目的是移除所有不用的对象，并且重新声明内存空间的归属等候将来使用。

### 6 并发重置(Concurrent Reset):

并发地重置所有算法需要的内部数据结构，为下一次GC做准备。

### CMS相关JVM参数

-XX:+UseConcMarkSweepGC：激活CMS收集器，默认情况下使用ParNew + CMS + Serial Old的收集器组合进行内存回收，Serial Old作为CMS出现“Concurrent Mode Failure”失败后的后备收集器使用。

-XX:CMSInitiatingOccupancyFraction={x}：在老年代的空间被占用{x}%时，调用CMS算法对老年代进行垃圾回收。

-XX:CMSFullGCsBeforeCompaction={x}：在进行了{x}次CMS算法之后，对老年代进行一次compaction

-XX:+CMSPermGenSweepingEnabled & -XX:+CMSClassUnloadingEnabled：让CMS默认遍历永久代（Perm区）

-XX:ParallelCMSThreads={x}：设置CMS算法中并行线程的数量为{x}。（默认启动(CPU数量+3) / 4个线程。）

-XX:+ExplicitGCInvokesConcurrent：用户程序中可能出现利用System.gc()触发系统Full GC（将会stop-the-world），利用这个参数可以指定System.gc()直接调用CMS算法做GC。

-XX:+DisableExplicitGC：该参数直接让JVM忽略用户程序中的System.gc()

### Concurrent Mode Failures

当CMS算法在并发的过程中堆空间无法满足用户程序对新空间的需求时，Stop-the-World的Full GC就会被触发，这就是Concurrent Mode Failures，这通常会造成一个长时间停顿。这种情况通常是因为老年代没有足够的空间供青年代对象promote。（包括没有足够的连续空间）

## 新生代垃圾回收器和老生代垃圾回收器都有哪些？有什么区别？

### 新生代垃圾回收器：

1. 新生代串行回收器

采用复制算法，-XX:+UseSerialGC开启，这是client模式下默认的回收器；

2、新生代ParNew回收器

-XX：+useParNewGC，-XX:+useConcMarkSweepGC开启。使用-XX:ParallelGCTheads指定线程数量，一般cpu数量小于8则等于cpu数量，cpu数量大于8则等于3+（（5 \* cpu数量）/8）

3、新生代ParalleGC回收器

关注系统吞吐量。

使用-XX:+UseParalleGC，-XX:UseParallelOldGC开启

-XX:MaxGCPauseMillis 设置最大垃圾收集停顿时间；

-XX:GCTimeRatio 设置吞吐量大小，它的值是介于0-100之间的整数。若该值为n，则系统将花费不超过1/（n + 1）的时间用于垃圾回收。默认值为99.

### 老年代垃圾回收器：

1、老年代串行回收器

标记压缩算法，-XX：+UseSerialGC，-XX：+useParNewGC,-XX:+useParallelGC

2、老年代ParallelOldGC

关注吞吐量。-XX:UseParallelOldGC开启

1. CMS收集器

关注系统停顿时间。-XX:+UseConcMarkSweepGC开启

CMS默认启动的并发线程数量是（ParallelGCThread + 3）/4

-XX:ConcGCThreads设置线程数量

-XX:CMSInitiatingOcuupancyFraction 老年代空间使用率达到多少时才进行回收，默认值是68.

## 简述分代垃圾回收器是怎么工作的？

<http://www.importnew.com/19255.html>

### 为什么要分代？

分代的垃圾回收策略，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。

### 分代清理过程

1.        创建新对象，大多数放在Eden区

2.        Eden满了（或达到一定比例），触发Minor GC,   把有用的复制到Survivor1, 同时清空Eden区。

3.        Eden区再次满了，出发Minor GC, 把Eden和Survivor1中有用的，复制到Survivor2, 同时清空Eden，Survivor1。

4.        Eden区第三次满了，出发Minor GC, 把Eden和Survivor2中有用的，复制到Survivor1, 同时清空Eden，Survivor2。

        形成循环，Survoivor1和Survivor中来回清空、复制，过程中有一个Survivor处于空的状态用于下次复制的。

5.        重复多次（默认15），没有被Survivor清理的对象，复制到Old（Tenuerd）区.

6.        当Old达到一定比例，触发Major GC，清理老年代。

7.        当Old满了，触发Full GC。注意，Full GC清理代价大，系统资源消耗高。

注：

1.        程序员可以像JVM提出垃圾回收请求，但是实际是否回收由JVM决定，程序员只提出清理请求建议，不能直接清理。

2.        Full GC清理代价大，系统资源消耗高，很多性能优化是针对Full GC做优化。

## 说一下 jvm 调优的工具？

<https://www.cnblogs.com/anxiao/p/6796644.html?utm_source=itdadao&utm_medium=referral>

## 常用的 jvm 调优的参数都有哪些？

<http://www.cnblogs.com/panxuejun/p/6058022.html>

## § Minor GC 和Full GC有什么区别？什么时候会触发Full GC？分别采用什么算法？

<https://www.jianshu.com/p/74b1f76834a1>

### Minor GC

从年轻代空间（包括 Eden 和 Survivor 区域）回收内存被称为 Minor GC。这一定义既清晰又易于理解。但是，当发生Minor GC事件的时候，有一些有趣的地方需要注意到：

当 JVM 无法为一个新的对象分配空间时会触发 Minor GC，比如当 Eden 区满了。所以分配率越高，越频繁执行 Minor GC。

内存池被填满的时候，其中的内容全部会被复制，指针会从0开始跟踪空闲内存。Eden 和 Survivor 区进行了标记和复制操作，取代了经典的标记、扫描、压缩、清理操作。所以 Eden 和 Survivor 区不存在内存碎片。写指针总是停留在所使用内存池的顶部。

执行 Minor GC 操作时，不会影响到永久代。从永久代到年轻代的引用被当成 GC roots，从年轻代到永久代的引用在标记阶段被直接忽略掉。

Major GC:清理永久代，但是由于很多MojorGC 是由MinorGC 触发的，所以有时候很难将MajorGC 和MinorGC区分开。

FullGC：是清理整个堆空间—包括年轻代和永久代。FullGC 一般消耗的时间比较长，远远大于MinorGC，因此，有时候我们必须降低FullGC 发生的频率。

新生代GC采用复制算法

老年代GC采用标记-整理算法

## § JVM里的有几种classloader，为什么会有多种？

1. BootstrapClassLoader:只能用于加载JDK核心类库，系统变量为sun.boot.class.path下面的类。该目录下的%JAVA\_HOME%/jre/lib/下的resources.jar;rt.jar等核心类库，该loader底层采用C++编写，自然你也就不能调用啦。
2. ExtClassLoader :用于加载一些扩展类，系统变量为java.ext.dirs中的类。作用：加载开发者自己扩展类。
3. AppClassLoader:用于加载用户类，这个就是java.class.path下的类，也就是我们自己编写出来的类。

其中这三个加载器顺序为BootstrapClassLoader>ExtClassLoader>AppClassLoader,为啥要这样设计？主要是为了扩展与安全。首先你将BootstrapClassLoader作为一个核心类加载器，只加载核心类，不与其他耦合在一起。并且为何要设计这三个加载器，就应该和双亲委派机制放在一起了。

## § 什么是双亲委派机制？介绍一些运作过程，双亲委派模型的好处；

当你需要加载类的时候，必须从顶级父加载器先加载，如果父加载不了，则交给子加载器。就相当于小孩子要做决定的时候，要先问问父亲怎么做。

为什么要有这种双亲委派机制：

1. 保证唯一性：试想，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，多个类加载器都去加载这个类到内存中，系统中将会出现多个不同的Object类，那么类之间的比较结果及类的唯一性将无法保证.
2. 保证安全：由于所有的用户类都会先通过bootstrapclassloader 查看里面有没有该类资源，有则直接安徽或者加载，从而保证了底层的类一定是预先加载的，这样可以对虚拟机的安全得到了很好的保证。

## § 什么情况下我们需要破坏双亲委派模型；

<https://blog.csdn.net/m0_37556444/article/details/81908685>

感觉就是，当父加载器需要加载**父加载器加载范围之外的类**的时候，就需要破坏了。

这篇文章说了jdbc违背：<https://www.oschina.net/question/2275855_2278405>

## § 常见的JVM调优方法有哪些？可以具体到调整哪个参数，调成什么值？

## § JVM虚拟机内存划分、类加载器、垃圾收集算法、垃圾收集器、class文件结构是如何解析的；

<https://www.cnblogs.com/wade-luffy/p/5929325.html>

NIO是什么？适用于何种场景？

<https://blog.csdn.net/weixin_33696106/article/details/87118322>

## 如何自定义spring boot starter？

<https://blog.csdn.net/boling_cavalry/article/details/82956512>

<https://blog.csdn.net/aa1358075776/article/details/81022285>

1. 创建自动配置参数承接类；