1、面向对象的特征

- **抽象**：将一类对象的共同特征总结出来构造类。数据抽象和行为抽象。只关注对象有哪些

属性和行为，不关注行为细节。

- **继承：**从已有类（父类（超类、基类））得到信息创建新类（子类（派生类））。封装程序中可变因素的重要手段。

- **封装**：把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。

|  |
| --- |
| 面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。  在类中编写的方法就是对实现细节的封装；  类就是对数据和数据操作的封装。  封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口 |

- **多态性**：允许不同子类型的对象对同一消息作出不同的响应。就是用同样的对象引用调用同样的方法但是做了不同的事情。

分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当A系统访问B系统提供的服务时，B系统有多种提供服务的方式，但一切对A系统来说都是透明的（就像电动剃须刀是A系统，它的供电系统是B系统，B系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A系统只会通过B类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。

方法重载（overload）实现的是编译时的多态性（也称为前绑定），

方法重写（override）实现的是运行时的多态性（也称为后绑定）。是面向对象最精髓的东西

要实现多态：

1).方法重写（子类继承父类并重写父类中已有的或抽象的方法）；

2).对象造型（用父类型引用引用子类型对象，同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

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

答：



类的成员不写访问修饰时默认default。

默认:对同一包中其他类相当于公开（public），对非同一包中其他类相当于私有（private）。

Protected：对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。

Java中，外部类的修饰符只能是public或默认，类的成员（包括内部类）的修饰符可是以上四种。

3、String 是最基本的数据类型吗？

答：不是。Java中基本数据类型8个：byte、short、int、long、float、double、char、boolean；其它都是引用类型（reference type），枚举类型也算一种较特殊的引用类型。

4、float f=3.4;是否正确？

答:不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

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

答：对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。

而short s1 = 1; s1 += 1;可以正确编译，因为s1+= 1;相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

7、int和Integer有什么区别？

答：Java是一个近乎纯洁的面向对象编程语言，但为了编程的方便还是引入了基本数据类型，但为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

- 原始类型: byte，short，int，long，float，double， boolean，char

- 包装类型:Byte，Short，Integer，Long，Float，Double ，Boolean，Character

|  |
| --- |
| class AutoUnboxingTest {  public static void main(String[] args) {  Integer a = new Integer(3);  Integer b = 3; // 将3自动装箱成Integer类型  int c = 3;  System.out.println(a == b); // false 两个引用没有引用同一对象  System.out.println(a == c); // true a自动拆箱成int类型再和c比较  }  } |

和自动装箱和拆箱有点关系，代码如下所示：

|  |
| --- |
| public class Test03 {  public static void main(String[] args) {  Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;  System.out.println(f1 == f2);// true -128-127之间，不会new新的对象  System.out.println(f3 == f4); //false  }  } |

如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是Integer对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。

|  |
| --- |
| public static Integer valueOf(int i) {  if (i >= IntegerCache.low && i <= IntegerCache.high)  return IntegerCache.cache[i + (-IntegerCache.low)];  return new Integer(i);  } |

IntegerCache是Integer的内部类，其代码如下所示：

|  |
| --- |
| /\*\*  \* Cache to support the object identity semantics of autoboxing for values between  \* -128 and 127 (inclusive) as required by JLS.  \*  \* The cache is initialized on first usage. The size of the cache  \* may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.  \* During VM initialization, java.lang.Integer.IntegerCache.high property  \* may be set and saved in the private system properties in the  \* sun.misc.VM class.  \*/  private static class IntegerCache {  static final int low = -128;  static final int high;  static final Integer cache[];  static {  // high value may be configured by property  int h = 127;  String integerCacheHighPropValue =  sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");  if (integerCacheHighPropValue != null) {  try {  int i = parseInt(integerCacheHighPropValue);  i = Math.max(i, 127);  // Maximum array size is Integer.MAX\_VALUE  h = Math.min(i, Integer.MAX\_VALUE - (-low) -1);  } catch( NumberFormatException nfe) {  // If the property cannot be parsed into an int, ignore it.  }  }  high = h;  cache = new Integer[(high - low) + 1];  int j = low;  for(int k = 0; k < cache.length; k++)  cache[k] = new Integer(j++);  // range [-128, 127] must be interned (JLS7 5.1.7)  assert IntegerCache.high >= 127;  }  private IntegerCache() {}  } |

简单的说，如果整型字面量的值在-128到127之间，那么不会new新的Integer对象，而是直接引用常量池中的Integer对象，所以上面题中f1==f2结果是true，而f3==f4结果是false。

8、&和&&的区别？

答：&运算符有两种用法：(1)按位与；(2)逻辑与。

&&:短路与运算。如&&左边的表达式值是false，右边表达式会被直接短路掉，不会运算。例如在验证用户登录时判定用户名不是null而且不是空字符串，应当写为：username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的equals比较，否则会产生NullPointerException异常。

9、解释内存中的栈(stack)、堆(heap)和方法区(method area)的用法。



答：基本数据类型的变量，对象的引用，函数调用的现场保存都使用JVM中的栈空间；

而通过new关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为Eden、Survivor（又可分为From Survivor和To Survivor）、Tenured；

方法区和堆都是各个线程共享的内存区域，用于存储已经被JVM加载的类信息、常量、静态变量、JIT编译器编译后的代码等数据；

程序中字面量如100、"hello"和常量都是放在常量池中，常量池是方法区的一部分。

栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过JVM的启动参数来进行调整，栈空间用光了会引发StackOverflowError，而堆和常量池空间不足则会引发OutOfMemoryError。

String str = new String("hello");

变量str放在栈上，用new创建出来的字符串对象放在堆上，"hello"字面量放在方法区。

方法区有个静态区，专门存放静态变量和静态块。

补充1：较新版本的Java（从Java 6的某个更新开始）中，由于JIT编译器的发展和"逃逸分析"技术的逐渐成熟，栈上分配、标量替换等优化技术使得对象一定分配在堆上这件事情已经变得不那么绝对了。

补充2：运行时常量池相当于Class文件常量池, 具有动态性，Java语言并不要求常量一定只有编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的。

看看下面代码的执行结果是什么并且比较一下Java 7以前和以后的运行结果是否一致。

String s1 = new StringBuilder("go").append("od").toString();

System.out.println(s1.intern() == s1);

String s2 = new StringBuilder("ja").append("va").toString();

System.out.println(s2.intern() == s2);

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

答：Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。

四舍五入的原理是在参数上加0.5然后进行下取整。

11、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：在Java 5以前，switch(expr)中，expr只能是byte、short、char、int。

从Java 5开始，Java中引入了枚举类型，expr也可以是enum类型，

Java 7开始，expr还可为字符串（String），但长整型（long）在目前所有的版本中都不可以

12、用最有效率的方法计算2乘以8？

答： 2 << 3（左移3位相当于乘以2的3次方，右移3位相当于除以2的3次方）。

补充：为编写的类重写hashCode方法时，可能会看到如下所示代码，其实不太理解为什么要使用这样的乘法运算来产生哈希码（散列码），而且为什么这个数是个素数，为什么通常选择31这个数？

选择31是因为可用移位和减法运算来代替乘法，从而得到更好的性能。31 \* num 等价于(num << 5) - num，左移5位相当于乘以2的5次方再减去自身就相当于乘以31，现在的VM都能自动完成这个优化。

|  |
| --- |
| public class PhoneNumber {  private int areaCode;  private String prefix;  private String lineNumber;  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result + areaCode;  result = prime \* result+ ((lineNumber == null) ? 0 : lineNumber.hashCode());  result = prime \* result + ((prefix == null) ? 0 : prefix.hashCode());  return result;  }  @Override  public boolean equals(Object obj) {  if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  PhoneNumber other = (PhoneNumber) obj;  if (areaCode != other.areaCode)  return false;  if (lineNumber == null) {  if (other.lineNumber != null)  return false;  } else if (!lineNumber.equals(other.lineNumber))  return false;  if (prefix == null) {  if (other.prefix != null)  return false;  } else if (!prefix.equals(other.prefix))  return false;  return true;  }  } |

13、数组有没有length()方法？String有没有length()方法？

答：数组没有length()方法，有length 的属性。

String 有length()方法。JavaScript中，获得字符串的长度是通过length属性得到的.

14、在Java中，如何跳出当前的多重嵌套循环？

答：在最外层循环前加一个标记如A，然后用break A;可以跳出多重循环。

（Java中支持带标签的break和continue语句，作用有点类似于C和C++中的goto语句，但是就像要避免使用goto一样，应该避免使用带标签的break和continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用，所以这种语法其实不知道更好）

15、构造器（constructor）是否可被重写（override）？

答：构造器不能被继承，因此不能被重写，但可被重载。

构造函数不是成员函数。是用来造唯一的东西的。不能用一个构造函数既造爸爸，又造儿子。这样身份就乱了。

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

答：不对，如果两个对象x和y满足x.equals(y) == true，哈希码（hash code）应当相同。

Java对于eqauls方法和hashCode方法是这样规定的：

(1)如果两个对象相同（equals方法返回true），那么它们的hashCode值一定要相同；

(2)如果两个对象的hashCode相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在Set集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。

补充：关于equals和hashCode方法，（《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍）：首先equals方法必须满足

自反性（x.equals(x)必须返回true）

对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）

传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）

一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。

实现高质量的equals方法的诀窍：

1. 使用==操作符检查"参数是否为这个对象的引用"；

2. 使用instanceof操作符检查"参数是否为正确的类型"；

3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；

4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；

5. 重写equals时总是要重写hashCode；

6. 不要将equals方法参数中Object对象替换为其他类型，重写时不要忘掉@Override注解。

17、是否可以继承String类？

答：String 类是final类，不可被继承。

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

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

答：是值传递。

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

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

19、String和StringBuilder、StringBuffer的区别？

答：可储存和操作字符串。

String是只读字符串，引用的字符串内容不能被改变。

StringBuffer/StringBuilder类表示的字符串对象可直接进行修改。

StringBuilder和StringBuffer方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。

题1 - 什么情况下用+运算符进行字符串连接比调用StringBuffer/StringBuilder对象的append方法连接字符串性能更好？

答案:如使用少量的字符串操作，使用 (+运算符)连接字符串；

如果频繁的对大量字符串进行操作，则使用

(1)全局变量或者需要多线程支持则使用StringBuffer；

(2)局部变量或者单线程不涉及线程安全则使有StringBuilder。

题2 - 请说出下面程序的输出。

|  |
| --- |
| class StringEqualTest {  public static void main(String[] args) {  String s1 = "Programming";  String s2 = new String("Programming");  String s3 = "Program";  String s4 = "ming";  String s5 = "Program" + "ming";  String s6 = s3 + s4;  System.out.println(s1 == s2);  System.out.println(s1 == s5);  System.out.println(s1 == s6);  System.out.println(s1 == s6.intern());  System.out.println(s2 == s2.intern());  }  } |

补充：

1. String对象的intern方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与String对象的equals结果是true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；

2. 字符串的+操作其本质是创建了StringBuilder对象进行append操作，然后将拼接后的StringBuilder对象用toString方法处理成String对象，这一点可以用javap -c StringEqualTest.class命令获得class文件对应的JVM字节码指令就可以看出来。

~~20、重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？~~

~~答：(1)都是实现多态的方式~~

~~(2)重载:编译时的多态性~~

~~重写:运行时的多态性~~

~~(3)重载发生在一个类中，同名方法有不同的参数列表（参数类型、参数个数或二者都不同）~~

~~重写发生在子类与父类之间，在子类中定义某方法与其父类有相同的名称和参数及返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。~~

~~(如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类)~~

~~重载对返回类型没有特殊的要求。~~

* ~~"为什么不能根据返回类型来区分重载"?~~

~~对于重载方法，不要试图通过给予不同的返回值类型（参数列表完全相同）加以区分。因为方法在调用时可以不将返回值赋值给一个对应的变量，这样就没有特征区分。也不要试图通过在调用这样的重载方法时赋值给对应的变量加以区分，因为问题是出现在重载方法定义上，而不是调用上。总而言之，Java不允许仅仅利用返回值的不同来区分重载方法。~~

21、描述一下JVM加载class文件的原理机制？

答：JVM中类的装载由类加载器（ClassLoader）和它的子类来实现.



类加载器是一个重要的Java运行时系统组件，负责在运行时查找和装入类文件中的类。

因Java的跨平台性，编译后的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。

类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。

类被加载后就进入连接阶段，包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。

最后JVM对类进行初始化：

1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；

2)如果类中存在初始化语句，就依次执行这些初始化语句。

类的加载由类加载器完成的，类加载器包括：

* 根加载器（BootStrap）
* 扩展加载器（Extension）
* 系统加载器（System）
* 用户自定义类加载器（java.lang.ClassLoader的子类）。

从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。更好的保证了Java平台的安全性。在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。

说明：

• Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；

• Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；

• System：又叫应用类加载器，其父类是Extension。是应用最广泛的类加载器。从环境变量classpath或系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。



22、char 型变量中能不能存贮一个中文汉字，为什么？

答：能. Java使用Unicode编码（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），所有文字都用2字节来表示。一个char类型占2字节（16比特），中文汉字(2字节)，所以放一个中文是没问题的。

补充：使用Unicode意味着字符在JVM内部和外部有不同的表现形式，在JVM内部都是Unicode，当这个字符被从JVM内部转移到外部时（如存入文件系统中），需进行编码转换。

所以Java中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如InputStreamReader和OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；。

~~23、抽象类（abstract class）和接口（interface）有什么异同？~~

~~答：~~

**~~相同点:~~**

* ~~抽象类和接口都不能够实例化，但可定义抽象类和接口类型的引用。~~
* ~~一个类如果继承了某抽象类或实现了某接口, 都需对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。~~

**~~不同点：~~**

* ~~接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全都是抽象方法。~~
* ~~抽象类中成员可是private、默认、protected、public，而接口中的成员全都是public的。~~
* ~~抽象类中可定义成员变量，而接口中定义的成员变量实际上都是常量。~~
* ~~有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。~~

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

答：Static Nested Class是被声明为静态(static)的内部类，可不依赖于外部类实例被实例化。

而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示。

|  |
| --- |
| /\*\*  \* 扑克类（一副扑克）  \*/  public class Poker {  private static String[] suites = {"黑桃", "红桃", "草花", "方块"};  private static int[] faces = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};  private Card[] cards;  /\*\*  \* 构造器  \*  \*/  public Poker() {  cards = new Card[52];  for(int i = 0; i < suites.length; i++) {  for(int j = 0; j < faces.length; j++) {  cards[i \* 13 + j] = new Card(suites[i], faces[j]);  }  }  }  /\*\*  \* 洗牌 （随机乱序）  \*/  public void shuffle() {  for(int i = 0, len = cards.length; i < len; i++) {  int index = (int) (Math.random() \* len);  Card temp = cards[index];  cards[index] = cards[i];  cards[i] = temp;  }  }  /\*\*  \* 发牌  \* @param index 发牌的位置  \*/  public Card deal(int index) {  return cards[index];  }  /\*\*  \* 卡片类（一张扑克）  \* [内部类]  \*  \*/  public class Card {  private String suite; // 花色  private int face; // 点数  public Card(String suite, int face) {  this.suite = suite;  this.face = face;  }  @Override  public String toString() {  String faceStr = "";  switch(face) {  case 1: faceStr = "A"; break;  case 11: faceStr = "J"; break;  case 12: faceStr = "Q"; break;  case 13: faceStr = "K"; break;  default: faceStr = String.valueOf(face);  }  return suite + faceStr;  }  }  }  测试代码：  class PokerTest {  public static void main(String[] args) {  Poker poker = new Poker();  poker.shuffle(); // 洗牌  Poker.Card c1 = poker.deal(0); // 发第一张牌  // 对于非静态内部类Card  // 只有通过其外部类Poker对象才能创建Card对象  Poker.Card c2 = poker.new Card("红心", 1); // 自己创建一张牌  System.out.println(c1); // 洗牌后的第一张  System.out.println(c2); // 打印: 红心A  }  } |

下面的代码哪些地方会产生编译错误？

|  |
| --- |
| public class Outer {  class Inner{}  //外部类的静态方法中不能new内部类的实例对象  public static void foo() {  //new Inner();//报错:没有可以访问外部类型的外部实例。  //new Outer().new Inner();//正确  }    public void bar() {  new Inner();  }    public static void main(String[] args) {  //new Inner();//报错:没有可以访问外部类型的外部实例。  new Outer().new Inner();//正确  }  } |

注意：Java中非静态内部类对象的创建要依赖其外部类对象，上面题中foo和main方法都是静态方法，静态方法中没有this，也就是说没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中创建内部类对象，可以这样做：

new Outer().new Inner();

25、Java 中会存在内存泄漏吗，请简单描述。

答：

理论上Java因有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；

实际开发中，可能会存在无用但可达的对象，不能被GC回收，也会导致内存泄露。

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

下面例子中的代码也会导致内存泄露。

|  |
| --- |
| import java.util.Arrays;  import java.util.EmptyStackException;  public class MyStack<T> {  private T[] elements;  private int size = 0;  private static final int INIT\_CAPACITY = 16;  public MyStack() {  elements = (T[]) new Object[INIT\_CAPACITY];  }  public void push(T elem) {  ensureCapacity();  elements[size++] = elem;  }  public T pop() {  if(size == 0)  throw new EmptyStackException();  return elements[--size];  }  private void ensureCapacity() {  if(elements.length == size) {  elements = Arrays.copyOf(elements, 2 \* size + 1);  }  }  } |

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

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

答：都不能。

抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。

（因静态的可用类名调用，而调用一个抽象方法没有意义）

本地方法是由本地代码（如C代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

|  |
| --- |
| 重写只能适用于可观察的实例方法.不能用于静态方法和final、private.  static、final、private方法本身都是编译期绑定的（也叫前期绑定），不存在多态，是在还没有运行的时候，程序在编译器里面就知道该调用哪个类的哪个方法了，而其他可观察的普通方法的绑定是在运行的时候根据具体的对象决定的，是可被继承的，有了多态而造成了不确定性 |

27、阐述静态变量和实例变量的区别。

答：

* 静态变量：static修饰的变量，也称类变量，属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；

静态变量可以实现让多个对象共享内存。

* 实例变量依存于某一实例，需要先创建对象然后通过对象才能访问到它。

补充：在Java开发中，上下文类和工具类中通常会有大量的静态成员。

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

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

29、如何实现对象克隆？

答：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，可实现真正的深度克隆

|  |
| --- |
| import java.io.ByteArrayInputStream;  import java.io.ByteArrayOutputStream;  import java.io.ObjectInputStream;  import java.io.ObjectOutputStream;  import java.io.Serializable;  public class MyUtil {  private MyUtil() {  throw new AssertionError();  }  @SuppressWarnings("unchecked")  public static <T extends Serializable> T clone(T obj) throws Exception {  ByteArrayOutputStream bout = new ByteArrayOutputStream();  ObjectOutputStream oos = new ObjectOutputStream(bout);  oos.writeObject(obj);  ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());  ObjectInputStream ois = new ObjectInputStream(bin);  return (T) ois.readObject();  // 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有意义  // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放  }  } |

|  |
| --- |
| 下面是测试代码：  import java.io.Serializable;  /\*\*  \* 人类  \*/  class Person implements Serializable {  private static final long serialVersionUID = -9102017020286042305L;  private String name; // 姓名  private int age; // 年龄  private Car car; // 座驾  public Person(String name, int age, Car car) {  this.name = name;  this.age = age;  this.car = car;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  public Car getCar() {  return car;  }  public void setCar(Car car) {  this.car = car;  }  @Override  public String toString() {  return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";  }  }  /\*\*  \* 小汽车类  \* @author 骆昊  \*  \*/  class Car implements Serializable {  private static final long serialVersionUID = -5713945027627603702L;  private String brand; // 品牌  private int maxSpeed; // 最高时速  public Car(String brand, int maxSpeed) {  this.brand = brand;  this.maxSpeed = maxSpeed;  }  public String getBrand() {  return brand;  }  public void setBrand(String brand) {  this.brand = brand;  }  public int getMaxSpeed() {  return maxSpeed;  }  public void setMaxSpeed(int maxSpeed) {  this.maxSpeed = maxSpeed;  }  @Override  public String toString() {  return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";  }  }  class CloneTest {  public static void main(String[] args) {  try {  Person p1 = new Person("Hao LUO", 33, new Car("Benz", 300));  Person p2 = MyUtil.clone(p1); // 深度克隆  p2.getCar().setBrand("BYD");  // 修改克隆的Person对象p2关联的汽车对象的品牌属性  // 原来的Person对象p1关联的汽车不会受到任何影响  // 因为在克隆Person对象时其关联的汽车对象也被克隆了  System.out.println(p1);  } catch (Exception e) {  e.printStackTrace();  }  }  } |

注意：基于序列化和反序列化实现的克隆不仅是深度克隆，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，这种方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

30、GC是什么？为什么要有GC？

答：GC（垃圾收集），忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃

GC功能可自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。

Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。

要请求垃圾收集，可调：System.gc() 或Runtime.getRuntime().gc() ，但JVM可屏蔽掉显示的垃圾回收调用。

垃圾回收可有效的防止内存泄露，有效的使用可使用的内存。

垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于android系统中垃圾回收的不可预知性。

补充：垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等

标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。

Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

- 伊甸园（Eden）：对象最初诞生的区域，对大多数对象来说，是它们唯一存在过的区域。

- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。

- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM参数：

• -Xms / -Xmx — 堆的初始大小 / 堆的最大大小

• -Xmn — 堆中年轻代的大小

• -XX:-DisableExplicitGC — 让System.gc()不产生任何作用

• -XX:+PrintGCDetails — 打印GC的细节

• -XX:+PrintGCDateStamps — 打印GC操作的时间戳

• -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小

• -XX:NewRatio — 可以设置老生代和新生代的比例

• -XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布

• -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值

• -XX:TargetSurvivorRatio：设置幸存区的目标使用率

|  |
| --- |
| jvm参数在哪里设置，分情况：  1、集成开发环境下启动并使用JVM，如eclipse需要修改根目录文件eclipse.ini；  2、Windows服务器下安装版Tomcat，可使用Tomcat7w.exe工具（tomcat目录下）和直接修改注册表两种方式修改Jvm参数；  3、Windows服务器解压版Tomcat注册Windows服务，方法同上；  4、解压版本的Tomcat, 通过startup.bat启动tomcat加载配置的，在tomcat 的bin 下catalina.bat 文件内添加；  5、Linux服务器Tomcat设置JVM，修改TOMCAT\_HOME/bin/catalina.sh； |

31、String s = new String("xyz");创建了几个字符串对象？

答：两个对象，一个是静态区(方法区内)的"xyz"，一个是用new创建在堆上的对象。

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

答：

* 接口可继承接口，而且支持多重继承。
* 抽象类可实现(implements)接口

|  |
| --- |
| 当只想实现接口中的个别方法（不是所有方法）时，可先写一个抽象类来实现该接口，并实现除了你想要的方法之外的所有方法（方法体为空）。  接着再用你的类继承这个抽象类，这个类中就只用实现你需要的方法了，就可达到需要。但是，如果你直接实现接口的话，你就需要实现接口的所有方法。 |

* 抽象类可继承具体类也可继承抽象类。

|  |  |  |  |
| --- | --- | --- | --- |
| 问： 抽象类是否可继承实体类 (concrete class)  答： 可以。前提是实体类必须有明确的构造函数  Object就是个实体类，java的API里，抽象类条目里都明确写着直接或间接继承自Object。   |  | | --- | | class A{}abstract class B extends A{} |   结果完全正常，编译通过。  涉及到两个基础知识：  1.所有的class必须有构造方法，如没有在代码里声明，系统会自动生成公有无参构造方法。而只要自己声明了，无论有参无参，私有公有，系统就不再帮你生成默认的无参构造器。  2.所有的子类构造器都要求在第一行代码中调用父类构造器，如果不写，系统默认去调用父类的无参构造器。  所以，如果把系统默认配给的方法也算进去，class A{}的代码实际上是   |  | | --- | | class A{  public A(){}  }  B继承 A 的时候，则是  abstract class B extends A{  public B(){  super();  }  } |   假如：   |  | | --- | | class A{  private A(){}  } |   系统不再给你默认无参构造器， B的构造器根据（2）中的规则去调用super()，却找不到A的无参构造器，导致abstract class B extends A{} 编译不能通过。   * “实体类必须有明确的构造函数”的含义：   1.没写构造器的，那是拥有默认无参公有构造函数的  2.写了子类可访问的无参构造器的，也是一样，子类里可以什么都不写，用默认机制调用。  3.写了有参构造器却没写无参构造器的，父类里没有子类可访问的无参构造器，子类必须在子类构造器里的第一句写明，调用父类有参构造器，并把参数传进去。  4.声明为final的以及所有构造器都不在子类访问权限之内的类无法继承  只要是在类的继承中，无论抽象还是实体，都需符合这个规则。。  “可继承，但和实体类的继承一样，也要求父类可继承，并且拥有子类可访问到的构造器。 |

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

答：可以。

但一个源文件中最多只能有一个公开类（public class）

且文件名必须同公开类名。

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

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

35、内部类可引用它的包含类（外部类）的成员吗？有没有什么限制？

答：一个内部类对象可访问创建它的外部类对象的成员，包括私有成员。

36、Java 中的final关键字有哪些用法？

答：

(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被重写；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

37、指出下面程序的运行结果。

|  |
| --- |
| class A {  static {  System.out.print("1");  }  public A() {  System.out.print("2");  }  }  class B extends A{  static {  System.out.print("a");  }  public B() {  System.out.print("b");  }  }  public class Hello {  public static void main(String[] args) {  A ab = new B();  ab = new B();  }  } |

答：执行结果：1a2b2b。

创建对象时构造器的调用顺序是：

先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。

38.1如何将字符串转换为基本数据类型？

* 基本数据类型 a=基本数据类型包装类.parse基本数据类型(String str);

|  |
| --- |
| str="123";  int parseInt = Integer.parseInt(str); |

38.2 如何将基本数据类型转换为字符串？

答： 基本数据类型 + ""

String.valueOf(Object)

39、如何实现字符串的反转及替换？

答：可自己写实现，也可使用String或StringBuffer/StringBuilder中方法。

用递归实现字符串反转：

|  |
| --- |
| public static String reverse(String originStr) {  if(originStr == null || originStr.length() <= 1) {  return originStr;  }  return reverse(originStr.substring(1)) + originStr.charAt(0);  } |

40、怎样将GB2312编码的字符串转换为ISO-8859-1编码的字符串？

答：代码如下所示：

|  |
| --- |
| String s1 = "你好";  String s2 = new String(s1.getBytes("GB2312"), "ISO-8859-1"); |

41、日期和时间：

答：

* 问题1：如何取得年月日、小时分钟秒？

创建java.util.Calendar 实例，调用其get()方法传入不同的参数即可获得参数所对应的值。Java 8中可以使用java.time.LocalDateTimel来获取，代码如下所示。

|  |
| --- |
| public class DateTimeTest {  public static void main(String[] args) {  Calendar cal = Calendar.getInstance();  System.out.println(cal.get(Calendar.YEAR));  System.out.println(cal.get(Calendar.MONTH)); // 0 - 11  System.out.println(cal.get(Calendar.DATE));  System.out.println(cal.get(Calendar.HOUR\_OF\_DAY));  System.out.println(cal.get(Calendar.MINUTE));  System.out.println(cal.get(Calendar.SECOND));  // Java 8  LocalDateTime dt = LocalDateTime.now();  System.out.println(dt.getYear());  System.out.println(dt.getMonthValue()); // 1 - 12  System.out.println(dt.getDayOfMonth());  System.out.println(dt.getHour());  System.out.println(dt.getMinute());  System.out.println(dt.getSecond());  }  } |

* 问题2：如何取得从1970年1月1日0时0分0秒到现在的毫秒数？

//当前时间，以从历元至现在所经过的毫秒数形式。

//历元（即格林威治标准时间 1970 年 1 月 1 日的 00:00:00.000，格里高利历）

Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis(); //自1970年1月1日0时起的毫秒数

|  |
| --- |
| 补充:  public Date() {  this(System.currentTimeMillis());  } |

Clock.systemDefaultZone().millis(); // Java 8

问题3：- 如何取得某月的最后一天？

Calendar time = Calendar.getInstance();

time.getActualMaximum(Calendar.DAY\_OF\_MONTH);

问题4：如何格式化日期？

SimpleDateFormat类中的format(Date)方法可将日期格式化。

Java 8中可以用java.time.format.DateTimeFormatter来格式化时间日期，代码如下所示。

|  |
| --- |
| import java.text.SimpleDateFormat;  import java.time.LocalDate;  import java.time.format.DateTimeFormatter;  import java.util.Date;  class DateFormatTest {  public static void main(String[] args) {  SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");  Date date1 = new Date();  System.out.println(oldFormatter.format(date1));  // Java 8  DateTimeFormatter newFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");  LocalDate date2 = LocalDate.now();  System.out.println(date2.format(newFormatter));  }  } |

•

补充：Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括LocalDate、LocalTime、LocalDateTime、Clock、Instant等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。

42、打印昨天的当前时刻。

答：

|  |
| --- |
| import java.util.Calendar;  class YesterdayCurrent {  public static void main(String[] args){  Calendar cal = Calendar.getInstance();  cal.add(Calendar.DATE, -1);  System.out.println(cal.getTime());  }  } |

在Java 8中，可以用下面的代码实现相同的功能。

|  |
| --- |
| import java.time.LocalDateTime;  class YesterdayCurrent {  public static void main(String[] args) {  LocalDateTime today = LocalDateTime.now();  LocalDateTime yesterday = today.minusDays(1);  System.out.println(yesterday);  }  } |

43、比较一下Java和JavaSciprt。

答：

* 两个公司开发的不同产品。Java 是原Sun Microsystems公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而JavaScript是Netscape公司的产品，为了扩展Netscape浏览器的功能而开发的一种可以嵌入Web页面中运行的基于对象和事件驱动的解释性语言。JavaScript的前身是LiveScript；而Java的前身是Oak语言。
* 基于对象和面向对象：Java是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象；JavaScript是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象（Object-Based）和事件驱动（Event-Driven）的编程语言，因而它本身提供了非常丰富的内部对象供设计人员使用。
* 解释和编译：

Java的源代码在执行前，必须经过编译。

JavaScript:解释性，源代码不需编译，由浏览器解释执行。

* 强类型变量和类型弱变量：

Java:强类型变量检查，即所有变量在编译之前必须作声明；

JavaScript:弱类型，在使用变量前可不作声明，JS的解释器在运行时检查推断其数据类型。

* 代码格式不一样。

补充：其实Java和JavaScript最重要的区别是一个是静态语言，一个是动态语言。

目前的编程语言的发展趋势是函数式语言和动态语言。

在Java中类（class）是一等公民，而JavaScript中函数（function）是一等公民，因此JavaScript支持函数式编程，可以使用Lambda函数和闭包（closure），

当然Java 8也开始支持函数式编程，提供了对Lambda表达式以及函数式接口的支持。

~~44、什么时候用断言（assert）？~~

~~答：用于保证程序最基本、关键的正确性。~~

~~断言检查通常在开发和测试时开启。~~

~~为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。~~

~~断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；~~

~~如果表达式的值为false，那么系统会报告一个AssertionError。~~

~~断言使用如下面的代码所示：~~

|  |
| --- |
| ~~assert(a > 0); // throws an AssertionError if a <= 0~~ |

~~断言可有两种形式：~~

|  |
| --- |
| ~~assert Expression1;~~  ~~assert Expression1 : Expression2 ;~~ |

~~Expression1 应该总是产生一个布尔值。~~

~~Expression2可是得出一个值的任意表达式;这个值用于生成显示更多调试信息的字符串消息。~~

~~要在运行时启用断言，可以在启动JVM时使用-enableassertions或者-ea标记。~~

~~要在运行时选择禁用断言，可以在启动JVM时使用-da或者-disableassertions标记。~~

~~要在系统类中启用或禁用断言，可使用-esa或-dsa标记。~~

~~还可以在包的基础上启用或者禁用断言。~~

~~注意：断言不应该以任何方式改变程序的状态。简单的说，如果希望在不满足某些条件时阻止代码的执行，就可以考虑用断言来阻止它。~~

|  |  |
| --- | --- |
| ~~（一）首先明确：~~  ~~Assert是jdk1.4引入的。~~  ~~默认关闭。~~  ~~只适用于复杂的调试过程。~~  ~~一般用于程序执行结果的判断，不要让断言处理业务流程。~~  ~~（二）判断eclipse是否开启了断言，代码如下：~~   |  | | --- | | ~~public class AssertTest {~~  ~~public static void main(String[] args) {~~  ~~boolean isOpen = false;~~  ~~assert isOpen=true; //如果开启了断言，会将isOpen的值改为true~~  ~~System.out.println(isOpen);//打印是否开启了断言~~  ~~}~~  ~~}~~ |   ~~如果打印 true说明已经启用了断言，如果为 false 则没有启用断言。~~  ~~（三）eclipse中开启断言~~  ~~选择菜单：Run ---> Run Configurations...---> 选择 Arguments 选项卡~~  ~~在 VM arguments 文本框中输入： -ea 注意：中间没有空格，如果输入 -da 表示禁止断言。~~  ~~然后关闭该窗口，然后保存就开启了断言。~~  ~~(四) 断言使用~~  ~~assert关键字语法很简单，有两种用法：~~  ~~1、assert <boolean表达式>~~  ~~如果<boolean表达式>为true，则程序继续执行。~~  ~~如果为false，则程序抛出AssertionError，并终止执行。~~  ~~2、assert <boolean表达式> : <错误信息表达式>~~  ~~如果<boolean表达式>为true，则程序继续执行。~~  ~~如果为false，则程序抛出java.lang.AssertionError，并输入<错误信息表达式>~~  ~~四、陷阱~~  ~~assert用法简单，但使用assert往往会让你陷入越来越深的陷阱中。应避免使用。~~  ~~原因：~~  ~~1、assert关键字需要在运行时候显式开启才能生效，否则断言就没有意义。~~  ~~主流的Java IDE工具默认都没有开启-ea断言检查功能。~~  ~~并且，对于Java Web应用，程序代码都是部署在容器里面，没法直接去控制程序的运行，如果一定要开启-ea的开关，则需要更改Web容器的运行配置参数。这对程序的移植和部署都带来很大的不便。~~  ~~2、用assert代替if是陷阱之二。assert的判断和if语句差不多，但两者的作用有着本质区别：~~  ~~assert本意上是为测试调试程序时使用的，但如果不小心用assert来控制了程序的业务流程，~~  ~~那在测试调试结束后去掉assert关键字就意味着修改了程序的正常的逻辑。~~  ~~3、assert断言失败将面临程序的退出。这在一个生产环境下的应用是绝不能容忍的。~~  ~~一般都是通过异常处理来解决程序中潜在错误。但使用断言就很危险，一旦失败系统就挂了。~~  ~~五、对assert的思考~~  ~~assert既然是为了调试测试程序用，不在正式生产环境下用，那应该考虑更好的测试JUint来代替其做用，~~  ~~JUint相对assert关键的所提供的功能是有过之而无不及。~~  ~~当然完全可以通过IDE debug来进行调试测试。~~  ~~应避免在Java中用assert关键字，除非哪一天Java默认支持开启-ea的开关，这时可考虑。~~ |

45、Error和Exception有什么区别？

答：Error表系统级的错误和程序不必处理的异常，是不是不可能恢复但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；

Exception表需要捕捉或需要程序进行处理的异常，是一种设计或实现问题；就是说，它表示如果程序运行正常，从不会发生的情况。

面试题：2005年摩托罗拉的面试问题

“If a process reports a stack overflow run-time error, what’s the most possible cause?”，

a. lack of memory;

b. write on an invalid memory space;

c. recursive function calling;

d. array index out of boundary.

Java程序在运行时也可能会遭遇StackOverflowError，是一个无法恢复的错误，只能重新修改代码了

答案是c。

如果写了不能迅速收敛的递归，则很有可能引发栈溢出的错误，如下所示：

|  |
| --- |
| class StackOverflowErrorTest {  public static void main(String[] args) {  main(null);  }  } |

结果: Exception in thread "main" java.lang.StackOverflowError

提示：用递归编写程序时一定要牢记两点：

1. 递归公式；

2. 收敛条件（什么时候就不再继续递归）。

46、try{}里有一个return语句，那么紧跟在这个try后的finally{}里的代码会不会被执行，什么时候被执行，在return前还是后?

答：会执行，在方法返回调用者前执行。

注意：在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是记录下返回值待finally代码块执行完毕之后, 再向调用者返回其值，然后如果在finally中修改了返回值，就会返回修改后的值。

显然，在finally中返回或者修改返回值会对程序造成很大的困扰

C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情

Java中也可通过提升编译器的语法检查级别来产生警告或错误，强烈建议将此项设置为编译错误。

47、Java语言如何进行异常处理，关键字：throws、throw、try、catch、finally分别如何使用？

答：Java通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供良好接口。

在Java中，每个异常都是一个对象，是Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。

Java的异常处理通过5个关键词来实现：try、catch、throw、throws和finally。

用try来执行一段程序，如系统会抛出（throw）一个异常对象，可通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；

try用来指定一块预防所有异常的程序；

catch子句紧跟在try块后面，用来指定想要捕获的异常的类型；

throw语句用来明确地抛出一个异常；

throws用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；

finally为确保一段代码不管发生什么异常状况都要被执行；

try语句可以嵌套，每当遇到一个try语句，异常的结构就会被放入异常栈中，直到所有的try语句都完成。如果下一级的try语句没有对某种异常进行处理，异常栈就会执行出栈操作，直到遇到有处理这种异常的try语句或者最终将异常抛给JVM。

48、运行时异常与受检异常有何异同？

答：异常表示程序运行过程中可能出现的非正常状态

运行时异常表虚拟机的通常操作中可能遇到的异常，是常见运行错误，只要程序设计没有问题通常就不会发生。

受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java编译器要求方法必须声明抛出可能发生的受检异常，但并不要求必须声明抛出未被捕获的运行时异常。

异常和继承一样，是面向对象程序设计中经常被滥用的东西

异常的使用指导原则：

- 不要将异常处理用于正常的控制流（设计良好的API不应该强迫它的调用者为了正常的控制流而使用异常）

- 对可恢复的情况使用受检异常，对编程错误使用运行时异常

- 避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）

- 优先使用标准的异常

- 每个方法抛出的异常都要有文档

- 保持异常的原子性

- 不要在catch中忽略掉捕获到的异常

49、列出一些你常见的运行时异常？

答：

- ArithmeticException（算术异常）

- ClassCastException （类转换异常）

- NullPointerException （空指针异常）

- IllegalArgumentException （非法参数异常）

- IndexOutOfBoundsException （下标越界异常）

- SecurityException （安全异常）

~~50、阐述final、finally、finalize的区别。~~

~~答：~~

* ~~final~~

~~三种用法：类被声明为final，不能被继承。~~

~~变量声明为final，在使用中不被改变，必须在声明时给定初值，引用中只读不可修改。~~

~~声明为final的方法不能在子类中被重写。~~

* ~~finally：~~

~~放在try…catch…后面构造总是执行代码块，意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可将释放外部资源的代码写在finally块中。~~

* ~~finalize~~

~~Object类中定义的方法. 在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。由垃圾收集器在销毁对象时调用，通过重写finalize()方法可整理系统资源者执行其他清理工作~~。

51、类ExampleA继承Exception，类ExampleB继承ExampleA。

有如下代码片断：

|  |
| --- |
| try {  throw new ExampleB("b")  } catch（ExampleA e）{  System.out.println("ExampleA");  } catch（Exception e）{  System.out.println("Exception");  } |

请问执行此段代码的输出是什么？

答：输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，

抓取ExampleA类型异常的catch块能够抓住try块中抛出的ExampleB类型的异常）

52、List、Set、Map是否继承自Collection接口？

答：List、Set 是，Map 不是。

Map是键值对映射容器，与List和Set有明显的区别，而Set存储的零散的元素且不允许有重复元素，List是线性结构的容器，适用于按数值索引访问元素的情形。

53、阐述ArrayList、Vector、LinkedList的存储性能和特性。

答：ArrayList 和Vector都使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接按序号索引元素，但插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢

Vector中方法由于添加了synchronized修饰，因此是线程安全，但性能上较ArrayList差，因此已经是Java中的遗留容器。

LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

Vector属遗留容器（Java早期版本中提供的，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已不推荐使用，但由于ArrayList和LinkedListed都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可通过工具类Collections中的synchronizedList方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现）。

补充：遗留容器中Properties和Stack类设计上有严重问题

Properties是一个键和值都是字符串的特殊的键值对映射，在设计上应是关联一个Hashtable并将其两个泛型参数设置为String类型，但是Java API中的Properties直接继承了Hashtable，明显是对继承的滥用。这里复用代码的方式应该是Has-A关系而不是Is-A关系，另一方面容器都属于工具类，继承工具类本身就是一个错误的做法，使用工具类最好的方式是Has-A关系（关联）或Use-A关系（依赖）。

同理，Stack类继承Vector也是不正确的。

54、Collection和Collections的区别？

答：Collection是一个接口，是Set、List等容器的父接口；

Collections是一个工具类，提供了一系列的静态方法来辅助容器操作，包括对容器的搜索、排序、线程安全化等等。

55、List、Map、Set三个接口存取元素时，各有什么特点？

答：

List以特定索引来存取元素，可有重复元素。

Set不能存放重复元素（用对象的equals()方法来区分元素是否重复）。

Map保存键值对（key-value pair）映射，映射关系可是一对一或多对一。

Set和Map容器都有基于哈希存储和排序树的两种实现版本

基于哈希存储的理论存取时间复杂度为O(1)

而基于排序树版本在插入或删除元素时会按元素或元素的键（key）构成排序树从而达到排序和去重的效果。

56、TreeMap和TreeSet在排序时如何比较元素？Collections.sort()方法如何比较元素？

答：

TreeSet要求存放对象所属的类必须实现Comparable接口,通过compareTo()比较元素大小。

TreeMap要求存放的键值对映射的键必须实现Comparable接口从而根据键对元素进行排序。

Collections工具类的sort方法有两种重载的形式:

* 第一种要求传入的待排序容器中存放的对象实现Comparable接口以实现元素比较；
* 第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator接口的子类型（需要重写compare方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java中对函数式编程的支持）。

|  |
| --- |
| 例子2：  public class Student {  private String name; // 姓名  private int age; // 年龄  public Student(String name, int age) {  this.name = name;  this.age = age;  }  /\*\*  \* 获取学生姓名  \*/  public String getName() {  return name;  }  /\*\*  \* 获取学生年龄  \*/  public int getAge() {  return age;  }  @Override  public String toString() {  return "Student [name=" + name + ", age=" + age + "]";  }  } |

|  |
| --- |
| import java.util.ArrayList;  import java.util.Collections;  import java.util.Comparator;  import java.util.List;  class Test02 {  public static void main(String[] args) {  List<Student> list = new ArrayList<>();  // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)  list.add(new Student("Hao LUO", 33));  list.add(new Student("XJ WANG", 32));  list.add(new Student("Bruce LEE", 60));  list.add(new Student("Bob YANG", 22));  // 通过sort方法的第二个参数传入一个Comparator接口对象  // 相当于是传入一个比较对象大小的算法到sort方法中  // 由于Java中没有函数指针、仿函数、委托这样的概念  // 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调  Collections.sort(list, new Comparator<Student> () {  @Override  public int compare(Student o1, Student o2) {  return o1.getName().compareTo(o2.getName()); // 比较学生姓名  }  });  for(Student stu : list) {  System.out.println(stu);  }  // 输出结果:  // Student [name=Bob YANG, age=22]  // Student [name=Bruce LEE, age=60]  // Student [name=Hao LUO, age=33]  // Student [name=XJ WANG, age=32]  }  } |

57、Thread类的sleep()方法和对象的wait()方法都可让线程暂停执行，有什么区别?

答：

sleep()方法是线程类的静态方法，调用此方法会让当前线程暂停执行指定时间，将执行机会（CPU）让给其他线程，但对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。

wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池，如果线程重新获得对象的锁就可进入就绪状态。

补充：进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位；

线程是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。

线程的划分尺度小于进程，使得多线程程序的并发性高；

进程在执行时通常拥有独立的内存单元，而线程间可共享内存。

使用多线程的编程通常能够带来更好的性能和用户体验，但多线程程序对其他程序是不友好的，因为它可能占用了更多的CPU资源。

当然，也不是线程越多，程序的性能就越好，因为线程间的调度和切换也会浪费CPU时间。时下很时髦的Node.js就采用了单线程异步I/O的工作模式。

58、线程的sleep()和yield()方法有什么区别？

答：

① sleep()给其他线程运行机会时不考虑线程优先级，因此会给低优先级线程运行的机会；

yield()方法只会给相同或更高优先级的线程运行的机会；

② 线程执行sleep()方法后转入阻塞（blocked）状态，

执行yield()方法后转入就绪（ready）状态；

③ sleep()方法声明抛出InterruptedException，而yield()没有声明任何异常；

④ sleep()比yield()（跟操作系统CPU调度相关）有更好的可移植性。

59、当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？

答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。

因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

60、请说出与线程同步以及线程调度相关的方法。

答：

- wait()：使线程处于等待（阻塞）状态，且释放所持有的对象的锁；

- sleep()：使正在运行的线程处于睡眠状态，静态方法，调用时要处理InterruptedException异常；

- notify()：唤醒一个处于等待状态的线程，在调用此方法时，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，且与优先级无关；

- notityAll()：唤醒所有处于等待状态的线程，并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

提示：关于Java多线程和并发编程的问题，《关于Java并发编程的总结和思考》。

补充：Java 5通过Lock接口提供了显式的锁机制，增强了灵活性以及对线程的协调。

Lock接口中定义了加锁（lock()）和解锁（unlock()）的方法，同时还提供了newCondition()方法来产生用于线程间通信的Condition对象；

此外，Java 5还提供了信号量机制（semaphore），信号量可用来限制对某个共享资源进行访问的线程的数量。在对资源访问前，线程必须得到信号量的许可（调用Semaphore对象的acquire()方法）；在完成对资源的访问后，线程必须向信号量归还许可（调用Semaphore对象的release()方法）。

下面例子演示了100个线程同时向一个银行账户中存入1元钱，在没有使用同步机制和使用同步机制情况下的执行情况。

* 没有使用同步机制

•银行账户类：

|  |
| --- |
| /\*\*  \* 银行账户  \*/  public class Account {  private double balance; // 账户余额  /\*\*  \* 存款  \* @param money 存入金额  \*/  public void deposit(double money) {  double newBalance = balance + money;  try {  Thread.sleep(10); // 模拟此业务需要一段处理时间  }  catch(InterruptedException ex) {  ex.printStackTrace();  }  balance = newBalance;  } |

|  |
| --- |
| /\*\*  \* 获得账户余额  \*/  public double getBalance() {  return balance;  }  }  • 存钱线程类：  /\*\*  \* 存钱线程  \* @author 骆昊  \*  \*/  public class AddMoneyThread implements Runnable {  private Account account; // 存入账户  private double money; // 存入金额  public AddMoneyThread(Account account, double money) {  this.account = account;  this.money = money;  }  @Override  public void run() {  account.deposit(money);  }  } |

|  |
| --- |
| • 测试类：  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  public class Test01 {  public static void main(String[] args) {  Account account = new Account();  ExecutorService service = Executors.newFixedThreadPool(100);  for(int i = 1; i <= 100; i++) {  service.execute(new AddMoneyThread(account, 1));  }  service.shutdown();  while(!service.isTerminated()) {}  System.out.println("账户余额: " + account.getBalance());  }  } |

未同步时，执行结果通常是显示账户余额在10元以下，原因是，当一个线程A试图存入1元的时候，另外一个线程B也能够进入存款的方法中，线程B读取到的账户余额仍然是线程A存入1元钱之前的账户余额，因此也是在原来的余额0上面做了加1元的操作，同理线程C也会做类似的事情，所以最后100个线程执行结束时，本来期望账户余额为100元，但实际得到的通常在10元以下（很可能是1元哦）。解决这个问题的办法就是同步，当一个线程对银行账户存钱时，需要将此账户锁定，待其操作完成后才允许其他的线程进行操作，代码有如下几种调整方案：

* 在银行账户的存款（deposit）方法上同步（synchronized）关键字

|  |
| --- |
| /\*\*  \* 银行账户  \*/  public class Account {  private double balance; // 账户余额  /\*\*  \* 存款  \* @param money 存入金额  \*/  public synchronized void deposit(double money) {  double newBalance = balance + money;  try {  Thread.sleep(10); // 模拟此业务需要一段处理时间  }  catch(InterruptedException ex) {  ex.printStackTrace();  }  balance = newBalance;  }  /\*\*  \* 获得账户余额  \*/  public double getBalance() {  return balance;  }  } |

* 在线程调用存款方法时对银行账户进行同步

|  |
| --- |
| /\*\*  \* 存钱线程  \*/  public class AddMoneyThread implements Runnable {  private Account account; // 存入账户  private double money; // 存入金额  public AddMoneyThread(Account account, double money) {  this.account = account;  this.money = money;  }  @Override  public void run() {  synchronized (account) {  account.deposit(money);  }  }  } |

* 通过Java 5显示锁机制，为每个银行账户创建一个锁对象，在存款操作进行加锁和解锁的操作

|  |
| --- |
| import java.util.concurrent.locks.Lock;  import java.util.concurrent.locks.ReentrantLock;  /\*\*  \* 银行账户  \*/  public class Account {  private Lock accountLock = new ReentrantLock();  private double balance; // 账户余额  /\*\*  \* 存款  \*  \* @param money  \* 存入金额  \*/  public void deposit(double money) {  accountLock.lock();  try {  double newBalance = balance + money;  try {  Thread.sleep(10); // 模拟此业务需要一段处理时间  }  catch (InterruptedException ex) {  ex.printStackTrace();  }  balance = newBalance;  }  finally {  accountLock.unlock();  }  }  /\*\*  \* 获得账户余额  \*/  public double getBalance() {  return balance;  }  } |

按上述三种方式对代码修改后，重写执行测试代码Test01，将看到最终的账户余额为100元。当然也可以使用Semaphore或CountdownLatch来实现同步。

61、编写多线程程序有几种实现方式？

答：Java 5以前有两种实现方法：

* 继承Thread类；
* 实现Runnable接口。

都要重写run()方法来定义线程行为，推荐用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

补充：Java 5后创建线程还有第三种方式：

* 实现Callable接口，接口中call方法可在线程执行结束时产生一个返回值

|  |
| --- |
| import java.util.ArrayList;  import java.util.List;  import java.util.concurrent.Callable;  import java.util.concurrent.ExecutorService;  import java.util.concurrent.Executors;  import java.util.concurrent.Future;  class MyTask implements Callable<Integer> {  private int upperBounds;  public MyTask(int upperBounds) {  this.upperBounds = upperBounds;  }  @Override  public Integer call() throws Exception {  int sum = 0;  for(int i = 1; i <= upperBounds; i++) {  sum += i;  }  return sum;  }  } |

|  |
| --- |
| class Test {  public static void main(String[] args) throws Exception {  List<Future<Integer>> list = new ArrayList<>();  ExecutorService service = Executors.newFixedThreadPool(10);  for(int i = 0; i < 10; i++) {  list.add(service.submit(new MyTask((int) (Math.random() \* 100))));  }  int sum = 0;  for(Future<Integer> future : list) {  // while(!future.isDone()) ;  sum += future.get();  }  System.out.println(sum);  }  } |

62、synchronized关键字的用法？

答：synchronized关键字可将对象或方法标记为同步，以实现对对象和方法的互斥访问，

可用synchronized(对象) { … }定义同步代码块，

或在声明方法时将synchronized作为方法的修饰符。

63、举例说明同步和异步。

答：如系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。

当应用程序在对象上调用了一个需要花费很长时间来执行的方法，且不希望让程序等待方法的返回时，就应使用异步编程，很多情况下采用异步往往更有效率。

事实上，同步就是指阻塞式操作，而异步就是非阻塞式操作。

64、启动一个线程是调用run()还是start()方法？

答：启动线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，意味着它可由JVM 调度并执行，并不意味着线程就会立即运行。

run()方法是线程启动后要进行回调（callback）的方法。

65、线程池（thread pool）

答：在面向对象编程中，创建和销毁对象很费时间

因创建一个对象要获取内存资源或其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。

线程池就是事先创建若干个可执行的线程放入一个池（容器）中，需要时从池中获取线程不用自行创建，使用完毕不需销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

- newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

- newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需的线程，就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。

- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

66、线程的基本状态及状态间的关系

答：

* Running表运行状态，
* Runnable表就绪状态（万事俱备，只欠CPU），
* Blocked表阻塞状态:
* 调用wait()方法进入等待池
* 执行同步方法或同步代码块进入等锁池
* 或是调用了sleep()或join()方法等待休眠或其他线程结束
* 或是因发生了I/O中断



67、synchronized 和java.util.concurrent.locks.Lock的异同

答：

相同点：Lock 能完成synchronized所实现的所有功能；

不同点：Lock有比synchronized更精确的线程语义和更好的性能，且不强制性的要求一定要获得锁。

synchronized会自动释放锁，而Lock要求程序员手工释放，且最好在finally 块中释放。

~~68、序列化~~

* ~~Java中如何实现序列化，有什么意义？~~

~~答：对象流就是将对象的内容进行流化。~~

~~可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。~~

~~序列化是为了解决对象流读写操作时可能引发的问题（如不进行序列化可能会数据乱序）。~~

~~要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造对象输出流，并通过writeObject(Object)方法就可将实现对象写出（即保存其状态）；~~

~~如需反序列化，可用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。~~

~~序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆~~

69、Java中有几种类型的流？

答：字节流和字符流。

字节流继承于InputStream、OutputStream，字符流继承于Reader、Writer。

在java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。

关于Java的I/O需要注意的有两点：

一是两种对称性（输入和输出的对称性，字节和字符的对称性）；

二是两种设计模式（适配器模式和装潢模式）。

另外Java中的流不同于C#的是它只有一个维度一个方向。

编程实现文件拷贝。

|  |
| --- |
| import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  import java.io.InputStream;  import java.io.OutputStream;  import java.nio.ByteBuffer;  import java.nio.channels.FileChannel;  public final class MyUtil {  private MyUtil() {  throw new AssertionError();  }  public static void fileCopy(String source, String target) throws IOException {  try (InputStream in = new FileInputStream(source)) {  try (OutputStream out = new FileOutputStream(target)) {  byte[] buffer = new byte[4096];  int bytesToRead;  while((bytesToRead = in.read(buffer)) != -1) {  out.write(buffer, 0, bytesToRead);  }  }  }  }  public static void fileCopyNIO(String source, String target) throws IOException {  try (FileInputStream in = new FileInputStream(source)) {  try (FileOutputStream out = new FileOutputStream(target)) {  FileChannel inChannel = in.getChannel();  FileChannel outChannel = out.getChannel();  ByteBuffer buffer = ByteBuffer.allocate(4096);  while(inChannel.read(buffer) != -1) {  buffer.flip();  outChannel.write(buffer);  buffer.clear();  }  }  }  }  } |

注意：上面用到Java 7的TWR，使用TWR后可以不用在finally中释放外部资源 ，从而让代码更加优雅。

70、$$写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。

答：代码如下：

|  |
| --- |
| import java.io.BufferedReader;  import java.io.FileReader;  public final class MyUtil {  // 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)  private MyUtil() {  throw new AssertionError();  }  /\*\*  \* 统计给定文件中给定字符串的出现次数  \* @param filename 文件名  \* @param word 字符串  \* @return 字符串在文件中出现的次数  \*/  public static int countWordInFile(String filename, String word) {  int counter = 0;  try (FileReader fr = new FileReader(filename)) {  try (BufferedReader br = new BufferedReader(fr)) {  String line = null;  while ((line = br.readLine()) != null) {  int index = -1;  while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {  counter++;  line = line.substring(index + word.length());  }  }  }  } catch (Exception ex) {  ex.printStackTrace();  }  return counter;  }  } |

Java中字符串中子串的查找:

|  |
| --- |
| 1、int indexOf(String str) ：返回第一次出现的指定子字符串在此字符串中的索引。  2、int indexOf(String str, int startIndex)：从指定的索引处开始，返回第一次出现的指定子字符串在此字符串中的索引。  3、int lastIndexOf(String str) ：返回在此字符串中最右边出现的指定子字符串的索引。  4、int lastIndexOf(String str, int startIndex) ：从指定的索引处开始向后搜索，返回在此字符串中最后一次出现的指定子字符串的索引。 |

例子

|  |
| --- |
| public class Test {  public static void main(String args[]) {  String Str = new String("菜鸟教程:www.runoob.com");  String SubStr1 = new String("runoob");  String SubStr2 = new String("com");  System.out.print("查找字符 o 第一次出现的位置 :" );  System.out.println(Str.indexOf( 'o' ));  System.out.print("从第14个位置查找字符 o 第一次出现的位置 :" );  System.out.println(Str.indexOf( 'o', 14 ));  System.out.print("子字符串 SubStr1 第一次出现的位置:" );  System.out.println( Str.indexOf( SubStr1 ));  System.out.print("从第十五个位置开始搜索子字符串 SubStr1 第一次出现位置 :" );  System.out.println( Str.indexOf( SubStr1, 15 ));  System.out.print("子字符串 SubStr2 第一次出现的位置 :" );  System.out.println(Str.indexOf( SubStr2 ));  }  } |

结果:

|  |
| --- |
| 查找字符 o 第一次出现的位置 :12  从第14个位置查找字符 o 第一次出现的位置 :17  子字符串 SubStr1 第一次出现的位置:9  从第十五个位置开始搜索子字符串 SubStr1 第一次出现的位置 :-1  子字符串 SubStr2 第一次出现的位置 :16 |

71、用Java代码列出一个目录下所有的文件？

答：

如果只要求列出当前文件夹下的文件，代码如下所示：

|  |
| --- |
| import java.io.File;  class Test12 {  public static void main(String[] args) {  File f = new File("/Users/Hao/Downloads");  for(File temp : f.listFiles()) {  if(temp.isFile()) {  System.out.println(temp.getName());  }  }  }  } |

如果需要对文件夹继续展开，代码如下所示：

|  |
| --- |
| import java.io.File;  class Test12 {  public static void main(String[] args) {  showDirectory(new File("/Users/Hao/Downloads"));  }  public static void showDirectory(File f) {  \_walkDirectory(f, 0);  }  private static void \_walkDirectory(File f, int level) {  if(f.isDirectory()) {  for(File temp : f.listFiles()) {  \_walkDirectory(temp, level + 1);  }  } else {  for(int i = 0; i < level - 1; i++) {  System.out.print("\t");  }  System.out.println(f.getName());  }  }  } |

在Java 7中可以使用NIO.2的API来做同样的事情，代码如下所示：

|  |
| --- |
| class ShowFileTest {  public static void main(String[] args) throws IOException {  Path initPath = Paths.get("/Users/Hao/Downloads");  Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {  @Override  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)  throws IOException {  System.out.println(file.getFileName().toString());  return FileVisitResult.CONTINUE;  }  });  }  } |

72、$$ 用Java的套接字编程实现一个多线程的回显（echo）服务器。

答：

|  |
| --- |
| import java.io.BufferedReader;  import java.io.IOException;  import java.io.InputStreamReader;  import java.io.PrintWriter;  import java.net.ServerSocket;  import java.net.Socket;  public class EchoServer {  private static final int ECHO\_SERVER\_PORT = 6789;  public static void main(String[] args) {  try(ServerSocket server = new ServerSocket(ECHO\_SERVER\_PORT)) {  System.out.println("服务器已经启动...");  while(true) {  Socket client = server.accept();  new Thread(new ClientHandler(client)).start();  }  } catch (IOException e) {  e.printStackTrace();  }  }  private static class ClientHandler implements Runnable {  private Socket client;  public ClientHandler(Socket client) {  this.client = client;  }  @Override  public void run() {  try(BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));  PrintWriter pw = new PrintWriter(client.getOutputStream())) {  String msg = br.readLine();  System.out.println("收到" + client.getInetAddress() + "发送的: " + msg);  pw.println(msg);  pw.flush();  } catch(Exception ex) {  ex.printStackTrace();  } finally {  try {  client.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  }  } |

注意：上面的代码使用了Java 7的TWR语法，由于很多外部资源类都间接的实现了AutoCloseable接口（单方法回调接口），因此可以利用TWR语法在try结束的时候通过回调的方式自动调用外部资源类的close()方法，避免书写冗长的finally代码块。此外，上面的代码用一个静态内部类实现线程的功能，使用多线程可以避免一个用户I/O操作所产生的中断影响其他用户对服务器的访问，简单的说就是一个用户的输入操作不会造成其他用户的阻塞。当然，上面的代码使用线程池可以获得更好的性能，因为频繁的创建和销毁线程所造成的开销也是不可忽视的。

下面是一段回显客户端测试代码：

|  |
| --- |
| import java.io.BufferedReader;  import java.io.InputStreamReader;  import java.io.PrintWriter;  import java.net.Socket;  import java.util.Scanner;  public class EchoClient {  public static void main(String[] args) throws Exception {  Socket client = new Socket("localhost", 6789);  Scanner sc = new Scanner(System.in);  System.out.print("请输入内容: ");  String msg = sc.nextLine();  sc.close();  PrintWriter pw = new PrintWriter(client.getOutputStream());  pw.println(msg);  pw.flush();  BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));  System.out.println(br.readLine());  client.close();  }  } |

•

如果希望用NIO的多路复用套接字实现服务器，代码如下所示。NIO的操作虽然带来了更好的性能，但是有些操作是比较底层的，对于初学者来说还是有些难于理解。

|  |
| --- |
| import java.io.IOException;  import java.net.InetSocketAddress;  import java.nio.ByteBuffer;  import java.nio.CharBuffer;  import java.nio.channels.SelectionKey;  import java.nio.channels.Selector;  import java.nio.channels.ServerSocketChannel;  import java.nio.channels.SocketChannel;  import java.util.Iterator;  public class EchoServerNIO {  private static final int ECHO\_SERVER\_PORT = 6789;  private static final int ECHO\_SERVER\_TIMEOUT = 5000;  private static final int BUFFER\_SIZE = 1024;  private static ServerSocketChannel serverChannel = null;  private static Selector selector = null; // 多路复用选择器  private static ByteBuffer buffer = null; // 缓冲区  public static void main(String[] args) {  init();  listen();  }  private static void init() {  try {  serverChannel = ServerSocketChannel.open();  buffer = ByteBuffer.allocate(BUFFER\_SIZE);  serverChannel.socket().bind(new InetSocketAddress(ECHO\_SERVER\_PORT));  serverChannel.configureBlocking(false);  selector = Selector.open();  serverChannel.register(selector, SelectionKey.OP\_ACCEPT);  } catch (Exception e) {  throw new RuntimeException(e);  }  }  private static void listen() {  while (true) {  try {  if (selector.select(ECHO\_SERVER\_TIMEOUT) != 0) {  Iterator<SelectionKey> it = selector.selectedKeys().iterator();  while (it.hasNext()) {  SelectionKey key = it.next();  it.remove();  handleKey(key);  }  }  } catch (Exception e) {  e.printStackTrace();  }  }  }  private static void handleKey(SelectionKey key) throws IOException {  SocketChannel channel = null;  try {  if (key.isAcceptable()) {  ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();  channel = serverChannel.accept();  channel.configureBlocking(false);  channel.register(selector, SelectionKey.OP\_READ);  } else if (key.isReadable()) {  channel = (SocketChannel) key.channel();  buffer.clear();  if (channel.read(buffer) > 0) {  buffer.flip();  CharBuffer charBuffer = CharsetHelper.decode(buffer);  String msg = charBuffer.toString();  System.out.println("收到" + channel.getRemoteAddress() + "的消息：" + msg);  channel.write(CharsetHelper.encode(CharBuffer.wrap(msg)));  } else {  channel.close();  }  }  } catch (Exception e) {  e.printStackTrace();  if (channel != null) {  channel.close();  }  }  }  }  •  import java.nio.ByteBuffer;  import java.nio.CharBuffer;  import java.nio.charset.CharacterCodingException;  import java.nio.charset.Charset;  import java.nio.charset.CharsetDecoder;  import java.nio.charset.CharsetEncoder;  public final class CharsetHelper {  private static final String UTF\_8 = "UTF-8";  private static CharsetEncoder encoder = Charset.forName(UTF\_8).newEncoder();  private static CharsetDecoder decoder = Charset.forName(UTF\_8).newDecoder();  private CharsetHelper() {  }  public static ByteBuffer encode(CharBuffer in) throws CharacterCodingException{  return encoder.encode(in);  }  public static CharBuffer decode(ByteBuffer in) throws CharacterCodingException{  return decoder.decode(in);  }  } |

•

73【##】XML文档定义有几种形式？它们之间有何本质区别？解析XML文档有哪几种方式？

答：XML文档定义分为DTD和Schema两种形式，二者都是对XML语法的约束，其本质区别在于Schema本身也是一个XML文件，可以被XML解析器解析，而且可以为XML承载的数据定义类型，约束能力较之DTD更强大。对XML的解析主要有DOM（文档对象模型，Document Object Model）、SAX（Simple API for XML）和StAX（Java 6中引入的新的解析XML的方式，Streaming API for XML），其中DOM处理大型文件时其性能下降的非常厉害，这个问题是由DOM树结构占用的内存较多造成的，而且DOM解析方式必须在解析文件之前把整个文档装入内存，适合对XML的随机访问（典型的用空间换取时间的策略）；SAX是事件驱动型的XML解析方式，它顺序读取XML文件，不需要一次全部装载整个文件。当遇到像文件开头，文档结束，或者标签开头与标签结束时，它会触发一个事件，用户通过事件回调代码来处理XML文件，适合对XML的顺序访问；顾名思义，StAX把重点放在流上，实际上StAX与其他解析方式的本质区别就在于应用程序能够把XML作为一个事件流来处理。将XML作为一组事件来处理的想法并不新颖（SAX就是这样做的），但不同之处在于StAX允许应用程序代码把这些事件逐个拉出来，而不用提供在解析器方便时从解析器中接收事件的处理程序。

74 [##] 你在项目中哪些地方用到了XML？

答：XML的主要作用有两个方面：数据交换和信息配置。在做数据交换时，XML将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从XML文件中还原相关信息进行处理，XML曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被JSON（JavaScript Object Notation）取而代之。当然，目前很多软件仍然使用XML来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在XML文件中，Java的很多框架也是这么做的，而且这些框架都选择了dom4j作为处理XML的工具，因为Sun公司的官方API实在不怎么好用。

补充：现在有很多时髦的软件（如Sublime）已经开始将配置文件书写成JSON格式，我们已经强烈的感受到XML的另一项功能也将逐渐被业界抛弃。

75、阐述JDBC操作数据库的步骤。

答：下面的代码以连接本机的Oracle数据库为例，演示JDBC操作数据库的步骤。

• 加载驱动。

Class.forName("oracle.jdbc.driver.OracleDriver");

• 创建连接。

Connection con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "scott", "tiger");

• 创建语句。

PreparedStatement ps = con.prepareStatement("select \* from emp where sal between ? and ?");

ps.setInt(1, 1000);

ps.setInt(2, 3000);

• 执行语句。

ResultSet rs = ps.executeQuery();

• 处理结果。

while(rs.next()) {

System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));

• 关闭资源。

|  |
| --- |
| finally {  if(con != null) {  try {  con.close();  } catch (SQLException e) {  e.printStackTrace();  }  }  } |

提示：关闭外部资源的顺序应该和打开的顺序相反，也就是说先关闭ResultSet、再关闭Statement、在关闭Connection。

上面代码只关闭了Connection（连接），虽然通常情况下在关闭连接时，连接上创建的语句和打开的游标也会关闭，但不能保证总是如此，因此应该按照刚才说的顺序分别关闭。

此外，第一步加载驱动在JDBC 4.0中可省略（自动从类路径中加载驱动），但建议保留。

76、Statement和PreparedStatement区别？哪个性能更好？

答：

与Statement相比，

①PreparedStatement接口代表预编译语句，主要优势在于可减少SQL的编译错误并增加SQL的安全性（减少SQL注射攻击的可能性）；

②PreparedStatement中SQL语句可带参数，避免了用字符串连接拼接SQL语句的麻烦和不安全；

③当批量处理SQL或频繁执行相同的查询时，PreparedStatement有明显的性能上的优势，由于数据库可将编译优化后的SQL语句缓存起来，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

补充：为了提供对存储过程的调用，JDBC API中还提供了CallableStatement接口。

存储过程（Stored Procedure）是数据库中一组为了完成特定功能的SQL语句的集合，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。

虽然调用存储过程会在网络开销、安全性、性能上获得很多好处，但是存在如果底层数据库发生迁移时就会有很多麻烦，因为每种数据库的存储过程在书写上存在不少的差别。

77、使用JDBC操作数据库时，如何提升读取/更新数据的性能？

答：

要提升读取数据的性能，可指定通过结果集（ResultSet）对象的setFetchSize()方法指定每次抓取的记录数（典型的空间换时间策略）；

要提升更新数据的性能可以使用PreparedStatement语句构建批处理，将若干SQL语句置于一个批处理中执行。

78、数据库编程时，连接池作用？

答：由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，释放连接需要进行TCP四次握手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。

池化技术在Java开发中很常见，用线程时创建线程池的道理与此相同。基于Java的开源数据库连接池主要有：C3P0、Proxool、DBCP、BoneCP、Druid等。

补充：在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

79、什么是DAO模式？

答：DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。

实际开发中，应将所有对数据源的访问操作抽象化后封装在一个公共API中。

就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。

在这个应用程序中，当需要和数据源交互时则使用这个接口，且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。

DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。

80、【B】事务的ACID是指什么？

答：

- 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；

- 一致性(Consistent)：事务结束后系统状态是一致的；

- 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态；

- 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

补充：关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在5类问题，包括3类数据读取问题（脏读、不可重复读和幻读）和2类数据更新问题（第1类丢失更新和第2类丢失更新）。

脏读（Dirty Read）：A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。

时间 转账事务A 取款事务B

T1 开始事务

T2 开始事务

T3 查询账户余额为1000元

T4 取出500元余额修改为500元

T5 查询账户余额为500元（脏读）

T6 撤销事务余额恢复为1000元

T7 汇入100元把余额修改为600元

T8 提交事务

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

不可重复读（Unrepeatable Read）：事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务B修改过了。

时间 转账事务A 取款事务B

T1 开始事务

T2 开始事务

T3 查询账户余额为1000元

T4 查询账户余额为1000元

T5 取出100元修改余额为900元

T6 提交事务

T7 查询账户余额为900元（不可重复读）

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

幻读（Phantom Read）：事务A重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务B提交的行。

时间 统计金额事务A 转账事务B

T1 开始事务

T2 开始事务

T3 统计总存款为10000元

T4 新增一个存款账户存入100元

T5 提交事务

T6 再次统计总存款为10100元（幻读）

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

第1类丢失更新：事务A撤销时，把已经提交的事务B的更新数据覆盖了。

时间 取款事务A 转账事务B

T1 开始事务

T2 开始事务

T3 查询账户余额为1000元

T4 查询账户余额为1000元

T5 汇入100元修改余额为1100元

T6 提交事务

T7 取出100元将余额修改为900元

T8 撤销事务

T9 余额恢复为1000元（丢失更新）

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

第2类丢失更新：事务A覆盖事务B已经提交的数据，造成事务B所做的操作丢失。

时间 转账事务A 取款事务B

T1 开始事务

T2 开始事务

T3 查询账户余额为1000元

T4 查询账户余额为1000元

T5 取出100元将余额修改为900元

T6 提交事务

T7 汇入100元将余额修改为1100元

T8 提交事务

T9 查询账户余额为1100元（丢失更新）

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。

直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析SQL语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，如下表所示：

隔离级别 脏读 不可重复读 幻读 第一类丢失更新 第二类丢失更新

READ UNCOMMITED 允许 允许 允许 不允许 允许

READ COMMITTED 不允许 允许 允许 不允许 允许

REPEATABLE READ 不允许 不允许 允许 不允许 不允许

SERIALIZABLE 不允许 不允许 不允许 不允许 不允许

需要说明的是，事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

81、【B】JDBC中如何进行事务处理？

答：Connection提供了事务处理的方法，通过调用setAutoCommit(false)可以设置手动提交事务；当事务完成后用commit()显式提交事务；如果在事务处理过程中发生异常则通过rollback()进行事务回滚。除此之外，从JDBC 3.0中还引入了Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。

82、【F】JDBC能否处理Blob和Clob？

答： Blob是指二进制大对象（Binary Large Object），而Clob是指大字符对象（Character Large Objec），因此其中Blob是为存储大的二进制数据而设计的，而Clob是为存储大的文本数据而设计的。

JDBC的PreparedStatement和ResultSet都提供了相应的方法来支持Blob和Clob操作。

下面的代码展示了如何使用JDBC操作LOB：

下面以MySQL数据库为例，创建一个张有三个字段的用户表，包括编号（id）、姓名（name）和照片（photo），建表语句如下：

create table tb\_user

(

id int primary key auto\_increment,

name varchar(20) unique not null,

photo longblob

);

下面的Java代码向数据库中插入一条记录：

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

class JdbcLobTest {

public static void main(String[] args) {

Connection con = null;

try {

// 1. 加载驱动（Java6以上版本可以省略）

Class.forName("com.mysql.jdbc.Driver");

// 2. 建立连接

con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123456");

// 3. 创建语句对象

PreparedStatement ps = con.prepareStatement("insert into tb\_user values (default, ?, ?)");

ps.setString(1, "骆昊"); // 将SQL语句中第一个占位符换成字符串

try (InputStream in = new FileInputStream("test.jpg")) { // Java 7的TWR

ps.setBinaryStream(2, in); // 将SQL语句中第二个占位符换成二进制流

// 4. 发出SQL语句获得受影响行数

System.out.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");

} catch(IOException e) {

System.out.println("读取照片失败!");

}

} catch (ClassNotFoundException | SQLException e) { // Java 7的多异常捕获

e.printStackTrace();

} finally { // 释放外部资源的代码都应当放在finally中保证其能够得到执行

try {

if(con != null && !con.isClosed()) {

con.close(); // 5. 释放数据库连接

con = null; // 指示垃圾回收器可以回收该对象

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

83、【B】简述正则表达式及其用途。

答：在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

说明：计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，正则表达式就是在进行字符串匹配和处理的时候最为强大的工具，绝大多数语言都提供了对正则表达式的支持。

84、【B】Java中是如何支持正则表达式操作的？

答：Java中的String类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，Java中可以用Pattern类表示正则表达式对象，它提供了丰富的API进行各种正则表达式操作，请参考下面面试题的代码。

面试题： - 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

import java.util.regex.Matcher;

import java.util.regex.Pattern;

class RegExpTest {

public static void main(String[] args) {

String str = "北京市(朝阳区)(西城区)(海淀区)";

Pattern p = Pattern.compile(".\*?(?=\\()");

Matcher m = p.matcher(str);

if(m.find()) {

System.out.println(m.group());

}

}

}

说明：上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的《正则表达式30分钟入门教程》。

85、获得一个类的类对象有哪些方式？

答：

- 方法1：类型.class，例如：String.class

- 方法2：对象.getClass()，例如："hello".getClass()

- 方法3：Class.forName()，例如：Class.forName("java.lang.String")

86、如何通过反射创建对象？

答：

- 方法1：通过类对象调用newInstance()方法，例如：String.class.newInstance()

- 方法2：通过类对象的getConstructor()或getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance("Hello");

87、如何通过反射获取和设置对象私有字段的值？

答：可以通过类对象的getDeclaredField()方法字段（Field）对象，然后再通过字段对象的setAccessible(true)将其设置为可以访问，接下来就可以通过get/set方法来获取/设置字段的值了。下面的代码实现了一个反射的工具类，其中的两个静态方法分别用于获取和设置私有字段的值，字段可以是基本类型也可以是对象类型且支持多级对象操作，例如ReflectionUtil.get(dog, "owner.car.engine.id");可以获得dog对象的主人的汽车的引擎的ID号。

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Modifier;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 反射工具类

\*

\*/

public class ReflectionUtil {

private ReflectionUtil() {

throw new AssertionError();

}

/\*\*

\* 通过反射取对象指定字段(属性)的值

\* @param target 目标对象

\* @param fieldName 字段的名字

\* @throws 如果取不到对象指定字段的值则抛出异常

\* @return 字段的值

\*/

public static Object getValue(Object target, String fieldName) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

try {

for(int i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(true);

target = f.get(target);

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(true);

return f.get(target);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

/\*\*

\* 通过反射给对象的指定字段赋值

\* @param target 目标对象

\* @param fieldName 字段的名称

\* @param value 值

\*/

public static void setValue(Object target, String fieldName, Object value) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

try {

for(int i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(true);

Object val = f.get(target);

if(val == null) {

Constructor<?> c = f.getType().getDeclaredConstructor();

c.setAccessible(true);

val = c.newInstance();

f.set(target, val);

}

target = val;

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(true);

f.set(target, value);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

}

•

88、如何通过反射调用对象的方法？

答：请看下面的代码：

import java.lang.reflect.Method;

class MethodInvokeTest {

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

String str = "hello";

Method m = str.getClass().getMethod("toUpperCase");

System.out.println(m.invoke(str)); // HELLO

}

}

89、简述一下面向对象的"六原则一法则"。

答：

- 单一职责原则：一个类只做它该做的事情。（单一职责原则想表达的就是"高内聚"，写代码最终极的原则只有六个字"高内聚、低耦合"，就如同葵花宝典或辟邪剑谱的中心思想就八个字"欲练此功必先自宫"，所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫"因为专注，所以专业"，一个对象如果承担太多的职责，那么注定它什么都做不好。这个世界上任何好的东西都有两个特征，一个是功能单一，好的相机绝对不是电视购物里面卖的那种一个机器有一百多种功能的，它基本上只能照相；另一个是模块化，好的自行车是组装车，从减震叉、刹车到变速器，所有的部件都是可以拆卸和重新组装的，好的乒乓球拍也不是成品拍，一定是底板和胶皮可以拆分和自行组装的，一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这样才能实现软件复用的目标。）

- 开闭原则：软件实体应当对扩展开放，对修改关闭。（在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。）

- 依赖倒转原则：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。）

里氏替换原则：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）

- 接口隔离原则：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。）

- 合成聚合复用原则：优先使用聚合或合成关系复用代码。（通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，类与类之间简单的说有三种关系，Is-A关系、Has-A关系、Use-A关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是Has-A关系，合成聚合复用原则想表达的是优先考虑Has-A关系而不是Is-A关系复用代码，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在Java的API中也有不少滥用继承的例子，例如Properties类继承了Hashtable类，Stack类继承了Vector类，这些继承明显就是错误的，更好的做法是在Properties类中放置一个Hashtable类型的成员并且将其键和值都设置为字符串来存储数据，而Stack类的设计也应该是在Stack类中放一个Vector对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。）

- 迪米特法则：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到"低耦合"，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web开发中作为前端控制器的Servlet或Filter不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）

90、简述一下你了解的设计模式。

答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

在GoF的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共23种设计模式，包括：Abstract Factory（抽象工厂模式），Builder（建造者模式），Factory Method（工厂方法模式），Prototype（原始模型模式），Singleton（单例模式）；Facade（门面模式），Adapter（适配器模式），Bridge（桥梁模式），Composite（合成模式），Decorator（装饰模式），Flyweight（享元模式），Proxy（代理模式）；Command（命令模式），Interpreter（解释器模式），Visitor（访问者模式），Iterator（迭代子模式），Mediator（调停者模式），Memento（备忘录模式），Observer（观察者模式），State（状态模式），Strategy（策略模式），Template Method（模板方法模式）， Chain Of Responsibility（责任链模式）。

面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：

- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。

- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。

- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。

- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。

除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

91、用Java写一个单例类。

答：

- 饿汉式单例

public class Singleton {

private Singleton(){}

private static Singleton instance = new Singleton();

public static Singleton getInstance(){

return instance;

}

}

• 懒汉式单例

public class Singleton {

private static Singleton instance = null;

private Singleton() {}

public static synchronized Singleton getInstance(){

if (instance == null) instance ＝ new Singleton();

return instance;

}

}

注意：实现一个单例有两点注意事项，①将构造器私有，不允许外界通过构造器创建对象；②通过公开的静态方法向外界返回类的唯一实例。这里有一个问题可以思考：Spring的IoC容器可以为普通的类创建单例，它是怎么做到的呢？

92、什么是UML？

答：UML是统一建模语言（Unified Modeling Language）的缩写，它发表于1997年，综合了当时已经存在的面向对象的建模语言、方法和过程，是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用UML可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。

93、UML中有哪些常用的图？

答：UML定义了多种图形化的符号来描述软件系统部分或全部的静态结构和动态结构，包括：用例图（use case diagram）、类图（class diagram）、时序图（sequence diagram）、协作图（collaboration diagram）、状态图（statechart diagram）、活动图（activity diagram）、构件图（component diagram）、部署图（deployment diagram）等。在这些图形化符号中，有三种图最为重要，分别是：用例图（用来捕获需求，描述系统的功能，通过该图可以迅速的了解系统的功能模块及其关系）、类图（描述类以及类与类之间的关系，通过该图可以快速了解系统）、时序图（描述执行特定任务时对象之间的交互关系以及执行顺序，通过该图可以了解对象能接收的消息也就是说对象能够向外界提供的服务）。

用例图：

类图：

时序图：

94、用Java写一个冒泡排序。

答：冒泡排序几乎是个程序员都写得出来，但是面试的时候如何写一个逼格高的冒泡排序却不是每个人都能做到，下面提供一个参考代码：

import java.util.Comparator;

/\*\*

\* 排序器接口(策略模式: 将算法封装到具有共同接口的独立的类中使得它们可以相互替换)

\* @author骆昊

\*

\*/

public interface Sorter {

/\*\*

\* 排序

\* @param list 待排序的数组

\*/

public <T extends Comparable<T>> void sort(T[] list);

/\*\*

\* 排序

\* @param list 待排序的数组

\* @param comp 比较两个对象的比较器

\*/

public <T> void sort(T[] list, Comparator<T> comp);

}

import java.util.Comparator;

/\*\*

\* 冒泡排序

\*

\*/

public class BubbleSorter implements Sorter {

@Override

public <T extends Comparable<T>> void sort(T[] list) {

boolean swapped = true;

for (int i = 1, len = list.length; i < len && swapped; ++i) {

swapped = false;

for (int j = 0; j < len - i; ++j) {

if (list[j].compareTo(list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

}

}

@Override

public <T> void sort(T[] list, Comparator<T> comp) {

boolean swapped = true;

for (int i = 1, len = list.length; i < len && swapped; ++i) {

swapped = false;

for (int j = 0; j < len - i; ++j) {

if (comp.compare(list[j], list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

}

}

}

95、用Java写一个折半查找。

答：折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是O(logN)。

import java.util.Comparator;

public class MyUtil {

public static <T extends Comparable<T>> int binarySearch(T[] x, T key) {

return binarySearch(x, 0, x.length- 1, key);

}

// 使用循环实现的二分查找

public static <T> int binarySearch(T[] x, T key, Comparator<T> comp) {

int low = 0;

int high = x.length - 1;

while (low <= high) {

int mid = (low + high) >>> 1;

int cmp = comp.compare(x[mid], key);

if (cmp < 0) {

low= mid + 1;

}

else if (cmp > 0) {

high= mid - 1;

}

else {

return mid;

}

}

return -1;

}

// 使用递归实现的二分查找

private static<T extends Comparable<T>> int binarySearch(T[] x, int low, int high, T key) {

if(low <= high) {

int mid = low + ((high -low) >> 1);

if(key.compareTo(x[mid])== 0) {

return mid;

}

else if(key.compareTo(x[mid])< 0) {

return binarySearch(x,low, mid - 1, key);

}

else {

return binarySearch(x,mid + 1, high, key);

}

}

return -1;

}

}

说明：上面的代码中给出了折半查找的两个版本，一个用递归实现，一个用循环实现。需要注意的是计算中间位置时不应该使用(high+ low) / 2的方式，因为加法运算可能导致整数越界，这里应该使用以下三种方式之一：low + (high - low) / 2或low + (high – low) >> 1或(low + high) >>> 1（>>>是逻辑右移，是不带符号位的右移）

System.out.println(a == b); // false 两个引用没有引用同一对象

System.out.println(a == c); // true a自动拆箱成int类型再和c比较

}

}

上面你说a自动拆箱成int类型,那他的引用是池中的还是堆中的?

一、软件思路(不需要花费太多经费)

1、lvs或者nginx做负载，后台tomcat集群实例做均衡。

2、需要优化tomcat中jre参数以及tomcat连接相关参数。

3、缓存技术，针对热数据或者更新少的数据，利用缓存技术

4、linux环境比windows稳定，最好内存大点，CPU多核。部署的实例数量最好不要超过cpu核数。

二、硬件思路（需要花费比较多经费）

1、还是负载均衡的思路。

2、F5等技术

《构建高性能Web站点》

* 1. **~~数据库优化的方面~~**

|  |
| --- |
| 1、没有创建索引，或者没有正确使用索引;  2、存在死锁的情况，从而导致select语句挂起;  3、返回不必要的列，如很多人喜欢在程序中使用select \* from 这样会查询表或视图中的所有字段，如果表或视图的数量比较大时候，就会浪费系统资源，应该只要返回必要的列 ;  4、在select语句中使用where子句，限制返回的记录数 ;  5、使用TOP关键字，限制返回的记录数 ;  6、在使用like关键字的时候需要注意通配符的使用方法 ;  7、慎用union关键字，因为它会影响到查询的效率 ;  8、慎用Distinct关键字，因为在结果集中的重复记录并不会影响到查询的效率，相反则需要消耗相应的资源去过滤重复记录，除非必须如此，不要使用此关键字 ;  9、如果需要经常对表中的数据进行统计，可以在表中增加一个统计字段，每次表中的数据发生变化的时候，动态更新统计字段。这样，在查询统计结果时，就不需要临时对表中的数据进行统计计算了 ;  10、如果需要多次对一个数据量非常大的表中的一部分数据进行查询操作，可以将这部分数据保存到临时表中，然后对临时表进行查询操作，并且可以根据需要在临时表中增加索引 ;  11、在where子句中使用BETWEEN关键字比IN要快，因为IN关键字对其后面的集合中的每一个元素都进行比较，如果必须使用IN关键字，则可以将频繁使用的值放在集合的前面，从而减少比较的次数 ;  12、 尽量少使用视图，特别是嵌套视图，可以直接从表中获取数据。在开发中，有人为了方便，设计一个包含很多字段的大视图，这样无论查询什么字段都从此视图中 取。实际上，最好从表中直接获取数据，一面查询大视图而造成的系统的开销。建议用存储过程取代视图，从而提高执行效率 ;  13、如果不要排序，尽量不要使用order by语句，这也会增加开销 ;  14、不要在where语句中=的左侧使用函数火表达式，英文系统将无法应用函数或表达式中的索引 ;  15、当判断表中是否包含记录时，使用EXISTS关键字，而不要使用count统计表中所有的记录数数量 。 |

1．补充

1.1【补】操作系统 tcp三次握手(为什么

1.2【F】临界区：

Synchronized可在任意对象及方法前加锁，被加锁的代码称为”互斥区”或“临界区”。

同一时刻只能有一个任务访问的代码区

1.3【F】乐观悲观锁

一：悲观锁

|  |
| --- |
| 一段执行逻辑加上悲观锁,不同线程同时执行时,只能有一个线程执行,其他的线程在入口处等待,直到锁被释放. |

不管是否发生多线程冲突，只要存在这种可能，就每次访问都加锁，加锁就会导致锁之间的争夺，有争夺就会有输赢，输者等待。

syncrhoized是一种独占锁，即：占用该锁的线程才可执行，申请该锁的线程就只能挂起等待，直到占用锁的线程释放锁才唤醒，拿到锁并执行。由于在进程挂起和恢复执行过程中存在着很大的开销，并且当一个线程正在等待锁时，它不能做任何事。所以syncrhoized是一种悲观锁，凡是用syncrhoized加了锁的多线程之间都会因锁的争夺结果导致挂起、唤醒等开销。

二：乐观锁

|  |
| --- |
| 一段执行逻辑加上乐观锁,不同线程同时执行时,可同时进入执行,在最后更新数据时，要检查这些数据是否被其他线程修改了(版本和执行初是否相同),没有修改则进行更新,否则放弃本次操作. |

获得锁后一直持有锁以防本线程再次申请该锁造成无谓的解锁再加锁开销

或假设没冲突而去完成同步代码块 ,如果冲突再循环重试

或采取申请锁失败后不立刻挂起而是稍微等待再次尝试获取 等待策略，以减少线程因挂起、阻塞、唤醒（发生CPU的调度切换） 而造成的开销。

偏向锁、轻量级锁（CAS轮询）、自旋锁 就是基于上述思路的乐观锁。

在多线程的加锁机制中，JVM会首先尝试乐观锁，失败后才调悲观锁。

悲观锁有很强的独占性,最安全.

乐观锁很开放,效率高,安全性比悲观锁低,因在乐观锁检查数据版本一致性时也可被其他线程修改数据.

**1.3** 【F】**死锁**

不同线程都在不等待可能被释放的锁，导致所有任务都无法继续完成，会造成线程”假死”,

**1.3.1 原因：**

（1） 因为系统资源不足。

（2） 进程运行推进的顺序不合适。

（3） 资源分配不当等。

如果系统资源充足，进程的资源请求都能够得到满足，死锁出现的可能性就很低，否则

就会因争夺有限的资源而陷入死锁。其次，进程运行推进顺序与速度不同，也可能产生死锁。

**1.3.2 产生死锁的四个必要条件：**

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

**1.3.3死锁的解除与预防**

在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。

**1.3.4 避免死锁的常见方法**

|  |
| --- |
| ·避免一个线程同时获取多个锁。通常并不现实  ·设计时考虑清楚锁的顺序，尽量减少嵌在的加锁交互数量  ·避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源。  ·尝试使用定时锁，使用lock.tryLock（timeout）来替代使用内部锁机制。可指定一个超时时限，在等待超过该时限之后变回返回一个失败信息  ·对于数据库锁，加锁和解锁必须在一个数据库连接里，否则会出现解锁失败的情况s |

**【示例及分析】**

如线程A持有锁L并且想获得锁M，线程B持有锁M且想获得锁L，这两个线程将永远等待下去，是最简单的死锁形式。

在数据库系统的设计中考虑了监测死锁及从死锁中恢复，如监测到一组事物发生了死锁，将选择一个牺牲者并放弃这个事物。

Java虚拟机解决死锁问题方面没有数据库这么强大，当一组Java线程发生死锁时，两个线程就永远不能再使用了，且由于两个线程分别持有了两个锁，这两段同步代码/代码块也无法再运行了----除非终止并重启应用。

死锁是设计的BUG，问题比较隐晦。不过死锁造成的影响很少会立即显现出来，一个类可能发生死锁，并不意味着每次都会发生死锁，这只是表示有可能。当死锁出现时，往往是在最糟糕的情况----高负载的情况下。

下面给出一个产生死锁的简单代码：

|  |
| --- |
| public class DeadLock  {  private final Object left = new Object();  private final Object right = new Object();    public void leftRight() throws Exception  {  synchronized (left)  {  Thread.sleep(2000);  synchronized (right)  {  System.out.println("leftRight end!");  }  }  }    public void rightLeft() throws Exception  {  synchronized (right)  {  Thread.sleep(2000);  synchronized (left)  {  System.out.println("rightLeft end!");  }  }  }  } |

一定要有"Thread.sleep(2000)"，不然一个线程运行了，另一个线程还没有运行，先运行的线程很有可能就已经连续获得两个锁了。写两个线程分别调用它们：

|  |
| --- |
| public class Thread0 extends Thread  {  private DeadLock dl;  public Thread0(DeadLock dl)  {  this.dl = dl;  }    public void run()  {  try  {  dl.leftRight();  }  catch (Exception e)  {  e.printStackTrace();  }  }  } |

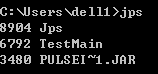
|  |
| --- |
| public class Thread1 extends Thread  {  private DeadLock dl;  public Thread1(DeadLock dl)  {  this.dl = dl;  }  public void run()  {  try  {  dl.rightLeft();  }  catch (Exception e)  {  e.printStackTrace();  }  }  } |

写个main函数调用一下：

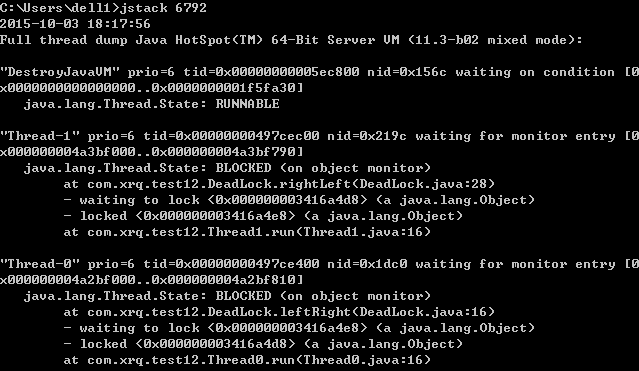
|  |
| --- |
| public static void main(String[] args)  {  DeadLock dl = new DeadLock();  Thread0 t0 = new Thread0(dl);  Thread1 t1 = new Thread1(dl);  t0.start();  t1.start();  while(true);  } |

没有结果，什么语句都不会打印，因为死锁了。下面演示一下如何定位死锁问题：

1、jps获得当前Java虚拟机进程的pid



2、jstack打印堆栈。jstack打印内容的最后其实已经报告发现了一个死锁



以"Thread-1"为例：

（1）"Thread-1"表示线程名称

（2）"prio=6"表示线程优先级

（3）"tid=00000000497cec00"表示线程Id

（4）nid=0x219c线程对应的本地线程Id，这个重点说明下。

因Java线程是依附于Java虚拟机中的本地线程运行的，实际上是本地线程在执行Java线程代码，只有本地线程才是真正的线程实体。Java代码中创建一个thread，虚拟机在运行期就会创建一个对应的本地线程，而这个本地线程才是真正的线程实体。

Linux环境下可使用"top -H -p JVM进程Id"来查看JVM进程下的本地线程（也被称作LWP）信息，注意这个本地线程是用十进制表示的，nid是用16进制表示的，转换一下就好了，0x219c对应的本地线程Id应该是8604。

（5）"[0x000000004a3bf000..0x000000004a3bf790]"表示线程占用的内存地址

（6）"java.lang.Thread.State：BLOCKED"表示线程的状态

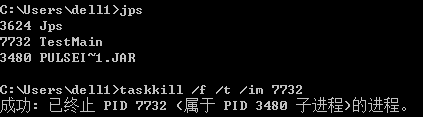
Thread-1处于BLOCKED状态，Thread-0处于BLOCKED状态。对这两个线程分析一下：

（1）Thread-1获得了锁0x000000003416a4e8，在等待锁0x000000003416a4d8

（2）Thread-0获得了锁0x000000003416a4d8，在等待锁0x000000003416a4e8

由于两个线程都在等待获取对方持有的锁，所以就这么永久等待下去了。

3、注意一下使用Eclipse/MyEclipse，这段程序如果不点击控制台上面的红色方框去Terminate掉它，而是右键->Run As->1 Java Application的话，这个进程会一直存在的，这时候可以利用taskkill命令去终止没有被Terminate的进程：



**1.4 【F】Hashmap@@@**

**1.4.1 HashMap底层实现原理-源码剖析**

**一、HashMap概述**

允许使用 null 值和 null 键。不保证映射顺序不变。

HashMap不是线程安全的,获得线程安全的HashMap:

Map map = Collections.synchronizedMap(new HashMap());

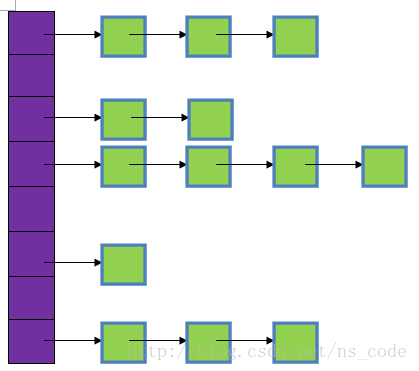
**二、数据结构**

基于数组和链表，查询速度快:通过计算散列码决定存储位置。

HashMap中主要通过key的hashCode计算hash值，只要hashCode相同，算出来的hash值就一样。

如存储的对象对多了，就有可能不同对象算出来的hash值相同，就出现了hash冲突。

紫色部分代表哈希表，数组的每个元素都是一个单链表的头节点，链表用来解决冲突，如不同的key映射到了数组同一位置处，就将其放入单链表中。



HashMap中Entry类代码：

|  |
| --- |
| /\*\*单向链表。  \* “HashMap链式存储法”对应的链表。  \* 实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode()这些函数  \*\*/  static class Entry<K,V> implements Map.Entry<K,V> {  final K key;  V value;  // 指向下一个节点  Entry<K,V> next;  final int hash;    // 构造函数。  // 输入参数包括"哈希值(h)", "键(k)", "值(v)", "下一节点(n)"  Entry(int h, K k, V v, Entry<K,V> n) {  value = v;  next = n;  key = k;  hash = h;  }    public final K getKey() {  return key;  }    public final V getValue() {  return value;  }  public final V setValue(V newValue) {  V oldValue = value;  value = newValue;  return oldValue;  }    // 判断两个Entry是否相等  // 若两个Entry的“key”和“value”都相等，则返回true。  // 否则，返回false  public final boolean equals(Object o) {  if (!(o instanceof Map.Entry))  return false;  Map.Entry e = (Map.Entry)o;  Object k1 = getKey();  Object k2 = e.getKey();  if (k1 == k2 || (k1 != null && k1.equals(k2))) {  Object v1 = getValue();  Object v2 = e.getValue();  if (v1 == v2 || (v1 != null && v1.equals(v2)))  return true;  }  return false;  }    // 实现hashCode()  public final int hashCode() {  return (key==null ? 0 : key.hashCode()) ^  (value==null ? 0 : value.hashCode());  }    public final String toString() {  return getKey() + "=" + getValue();  }    // 当向HashMap中添加元素时，绘调用recordAccess()。  // 这里不做任何处理  void recordAccess(HashMap<K,V> m) {  }    // 当从HashMap中删除元素时，绘调用recordRemoval()。  // 这里不做任何处理  void recordRemoval(HashMap<K,V> m) {  }  } |

HashMap其实就是个Entry数组，Entry对象中包含了键和值，其中next也是一个Entry对象，用来处理hash冲突的，形成一个链表。

**三、HashMap源码分析**

**1、关键属性**

|  |
| --- |
| transient Entry[] table; //存储元素的实体数组  transient int size; //存放元素的个数  int threshold; //临界值 当实际大小超过临界值时，会进行扩容threshold = 加载因子\*容量  final float loadFactor; //加载因子  transient int modCount;//被修改的次数 |

loadFactor加载因子表示Hsah表中元素的填满的程度.

|  |
| --- |
| 加载因子大,填满元素越多,空间利用率高,但冲突的机会加大.链表越来越长,查找效率降低。  反之,加载因子越小,填满的元素越少,冲突的机会减小,但空间浪费多了.表中数据将过于稀疏（很多空间还没用，就开始扩容了）  须在 "冲突的机会"与"空间利用率"间寻找平衡与折衷.本质上是"时-空"矛盾的平衡与折衷.  如机器内存足够，且想要提高查询速度,可将加载因子设置小一点；  如机器内存紧张，且对查询速度没什么要求,可将加载因子设置大一点。  不过一般不用去设置，取默认值0.75就好。 |

**2、构造方法**

|  |
| --- |
| public HashMap(int initialCapacity, float loadFactor) {  //确保数字合法  if (initialCapacity < 0)  throw new IllegalArgumentException("Illegal initial capacity: " +initialCapacity);  if (initialCapacity > MAXIMUM\_CAPACITY)  initialCapacity = MAXIMUM\_CAPACITY;  if (loadFactor <= 0 || Float.isNaN(loadFactor))  throw new IllegalArgumentException("Illegal load factor: " +loadFactor);  // Find a power of 2 >= initialCapacity  int capacity = 1; //初始容量  while (capacity < initialCapacity)  //确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次幂  capacity <<= 1;  this.loadFactor = loadFactor;  threshold = (int)(capacity \* loadFactor);  table = new Entry[capacity];  init();  }  public HashMap(int initialCapacity) {  this(initialCapacity, DEFAULT\_LOAD\_FACTOR);  }  public HashMap() {  this.loadFactor = DEFAULT\_LOAD\_FACTOR;  threshold = (int)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);  table = new Entry[DEFAULT\_INITIAL\_CAPACITY];  init();  } |

构造HashMap时如指定了加载因子和初始容量,就调用第一个构造方法，否则用默认的。

默认初始容量为16，默认加载因子为0.75。

可看到上面代码中13-15行，作用是确保容量为2的n次幂，使capacity为大于initialCapacity的最小的2的n次幂

**3、存储数据**

先看put方法：

|  |
| --- |
| public V put(K key, V value) {  //若“key为null”，则将该键值对添加到table[0]中。  if (key == null)  return putForNullKey(value);  //若“key不为null”，则计算该key的哈希值，然后将其添加到该哈希值对应的链表中。  int hash = hash(key.hashCode());  //搜索指定hash值在对应table中的索引  int i = indexFor(hash, table.length);  // 循环遍历Entry数组,若“该key”对应的键值对已存在，  // 则用新的value取代旧的value。然后退出！  for (Entry<K,V> e = table[i]; e != null; e = e.next) {  Object k;  if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  //如key相同则覆盖并返回旧值  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  //修改次数+1  modCount++;  //将key-value添加到table[i]处  addEntry(hash, key, value, i);  return null;  } |

每个Map.Entry 其实就是个 key-value 对。

当系统决定存储 HashMap 中的 key-value 对时，完全没考虑 Entry 中的 value，仅是根据 key 来计算并决定每个 Entry 的存储位置。

完全可把 Map 集合中的 value 当成 key 的附属，当系统决定了 key 的存储位置之后，value 随之保存在那里即可。

* 处理key值为null的情况， putForNullKey(value)方法：

|  |
| --- |
| private V putForNullKey(V value) {  for (Entry<K,V> e = table[0]; e != null; e = e.next) {  if (e.key == null) { //如有key为null的对象存在，则覆盖掉  V oldValue = e.value;  e.value = value;  e.recordAccess(this);  return oldValue;  }  }  modCount++;  addEntry(0, null, value, 0); //如果键为null的话，则hash值为0,存在table[0]处  return null;  } |

put方法中第4行，通过key的hashCode值计算hash码

|  |
| --- |
| //计算hash值的方法 通过键的hashCode来计算  static int hash(int h) {  // This function ensures that hashCodes that differ only by  // constant multiples at each bit position have a bounded  // number of collisions (approximately 8 at default load factor).  h ^= (h >>> 20) ^ (h >>> 12);  return h ^ (h >>> 7) ^ (h >>> 4);  } |

得到hash码后就会通过hash码计算出应存储在数组中的索引：

|  |
| --- |
| static int indexFor(int h, int length) { //根据hash值和数组长度算出索引值  return h & (length-1); //这里不能随便算取，用hash&(length-1)是有原因的，这样可确保算出来的索引是在数组大小范围内，不会超出} |

一般先想到用hash值对length取模（即除法散列法），如Hashtable，基本能保证元素在哈希表中散列的较均匀，但取模用到除法运算，效率很低

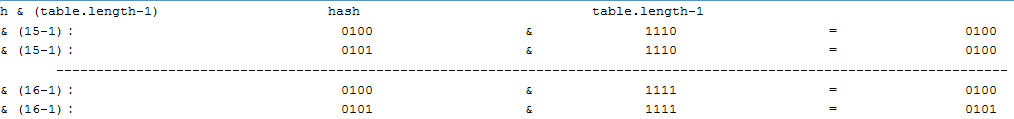
HashMap中通过h&(length-1)代替取模，同样实现了均匀散列，但效率要高很多，也是HashMap对Hashtable的一个改进。

接下来，分析下为什么哈希表的容量一定要是2的整数次幂。

|  |
| --- |
| 首先，length为2的整数次幂，h&(length-1)就相当于对length取模，保证了散列均匀，也提升了效率；  其次，length为2的整数次幂，为偶数，length-1为奇数，奇数的最后一位是1，便保证了h&(length-1)的最后一位可能为0，也可能为1（取决于h的值），即与后的结果可能为偶数，也可能为奇数，便可保证散列的均匀性  而如length为奇数，length-1为偶数，最后一位是0，h&(length-1)的最后一位肯定为0，即只能为偶数，任何hash值都只会被散列到数组的偶数下标位置上，浪费了近一半的空间，因此，length取2的整数次幂，是为了使不同hash值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。 |

这看上去很简单，其实有玄机，举个例子：

假设数组长度分别为15和16，优化后的hash码分别为8和9，那么&运算后的结果如下：



可看出：当8、9和(15-1)（1110）“与”时，产生了相同结果，就是说8、9会定位到数组中的同一位置上去，就产生了碰撞，8和9会被放到数组中的同一个位置上形成链表，查询时需遍历这个链表，得到8或者9，降低了查询效率。

同时，也可发现，当数组长度为15时，hash值会与15-1（1110）进行“与”，那么最后一位永远是0，而0001，0011，0101，1001，1011，0111，1101这几个位置永远都不能存放元素了，空间浪费相当大，更糟的是，数组可用的位置比数组长度小了很多，意味着进一步增加了碰撞的几率，减慢了查询效率！

而当数组长度为16时，即为2的n次方时，2^n-1得到的二进制数的每个位上的值都为1，使得在低位上&时，得到的和原hash的低位相同，加之hash(int h)方法对key的hashCode的进一步优化，加入了高位计算，使得只有相同的hash值的两个值才会被放到数组中的同一个位置上形成链表。

　　 所以说，当数组长度为2^n时，不同key算得的index相同的几率较小，数据在数组上分布就比较均匀，就是说碰撞的几率小，相对的，查询时就不用遍历某个位置上的链表，查询效率也就较高了。

【完整描述】根据上面 put 方法的源代码可看出，当程序试图将一个key-value对放入HashMap中时，程序首先根据该 key 的 hashCode() 返回值决定该 Entry 的存储位置：如果两个 Entry 的 key 的 hashCode() 返回值相同，那它们的存储位置相同。如果这两个 Entry 的 key 通过 equals 比较返回 true，新添加 Entry 的 value 将覆盖集合中原有 Entry 的 value，但key不会覆盖。如果这两个 Entry 的 key 通过 equals 比较返回 false，新添加的 Entry 将与集合中原有 Entry 形成 Entry 链，而且新添加的 Entry 位于 Entry 链的头部——具体说明继续看 addEntry() 方法的说明。

|  |
| --- |
| void addEntry(int hash, K key, V value, int bucketIndex) {  Entry<K,V> e = table[bucketIndex];  //如果要加入的位置有值，将该位置原先的值设置为新entry的next,  //也就是新entry链表的下一个节点  table[bucketIndex] = new Entry<>(hash, key, value, e);  if (size++ >= threshold) //如果大于临界值就扩容  resize(2 \* table.length); //以2的倍数扩容  } |

参数bucketIndex就是indexFor函数计算出来的索引值，第2行代码是取得数组中索引为bucketIndex的Entry对象，第3行就是用hash、key、value构建一个新的Entry对象放到索引为bucketIndex的位置，且将该位置原先的对象设置为新对象的next构成链表。

　　第4行和第5行就是判断put后size是否达到了临界值threshold，如达到了临界值就要进行扩容，HashMap扩容是扩为原来的两倍。

**4、调整大小**

**resize()**方法：

重新调整HashMap的大小，newCapacity是调整后的单位

|  |
| --- |
| void resize(int newCapacity) {  Entry[] oldTable = table;  int oldCapacity = oldTable.length;  if (oldCapacity == MAXIMUM\_CAPACITY) {  threshold = Integer.MAX\_VALUE;  return;  }    Entry[] newTable = new Entry[newCapacity];  transfer(newTable);//用来将原先table的元素全部移到newTable里面  table = newTable; //再将newTable赋值给table  threshold = (int)(newCapacity \* loadFactor);//重新算临界值  } |

新建了一个HashMap的底层数组，上面代码中第10行为调用transfer方法，将HashMap的全部元素添加到新的HashMap中,并重新计算元素在新的数组中的索引位置

|  |
| --- |
| void transfer(Entry[] newTable)  {  Entry[] src = table;  int newCapacity = newTable.length;  //下面这段代码的意思是：  // 从OldTable里摘一个元素出来，然后放到NewTable中  for (int j = 0; j < src.length; j++) {  Entry<K,V> e = src[j];  if (e != null) {  src[j] = null;  do {  //多个线程同时往HashMap添元素时，多次resize会有一定概率出现死循环  Entry<K,V> next = e.next;  int i = indexFor(e.hash, newCapacity);  e.next = newTable[i];  newTable[i] = e;  e = next;  } while (e != null); //  }  }  } |

当HashMap中的元素越来越多时，hash冲突的几率也就越来越高，因数组的长度是固定的。所以为了提高查询效率，就要对HashMap的数组进行扩容，数组扩容这个操作也会出现在ArrayList中，这是一个常用的操作，而在HashMap数组扩容之后，最消耗性能的点就出现了：原数组中的数据必须重新计算其在新数组中的位置，并放进去，这就是resize。

那么HashMap什么时候进行扩容呢？当HashMap中的元素个数超过数组大小\*loadFactor时，就会进行数组扩容，loadFactor的默认值为0.75，是个折中的取值。默认，数组大小为16，当HashMap中元素个数超过16\*0.75=12时，就把数组大小扩展为 2\*16=32，即扩大一倍，然后重新计算每个元素在数组中位置，扩容是需要进行数组复制的，复制数组非常消耗性能，所以如已经预知HashMap中元素个数，预设元素的个数能够有效的提高HashMap的性能。

**5、数据读取**

|  |
| --- |
| public V get(Object key) {  if (key == null)  return getForNullKey();  int hash = hash(key.hashCode());  for (Entry<K,V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {  Object k;  if (e.hash == hash && ((k = e.key) == key || key.equals(k)))  return e.value;  }  return null;  } |

有了上面存储时的hash算法作为基础，理解起来这段代码就很容易。

可看出：从HashMap中get元素时，首先计算key的hashCode，找到数组中对应位置的某一元素，然后通过key的equals方法在对应位置的链表中找到需要的元素。

**6、HashMap的性能参数：**

HashMap 包含如下几个构造器：

HashMap()：构建一个初始容量为 16，负载因子为 0.75 的 HashMap。

HashMap(int initialCapacity)：构建一初始容量initialCapacity，负载因子 0.75 的 HashMap。

HashMap(int initialCapacity, float loadFactor)：以指定初始容量和负载因子创建一个HashMap。

initialCapacity：HashMap的最大容量，即为底层数组的长度。

loadFactor：负载因子 散列表的实际元素数目(n)/ 散列表的容量(m)。

**负载因子**:衡量一个散列表的空间使用程度，越大表散列表的装填程度越高，反之愈小。对于用链表法的散列表，查找一个元素的平均时间是O(1+a)，因此如负载因子越大，对空间的利用更充分，后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。

threshold = (int)(capacity \* loadFactor); //HashMap的最大容量

threshold是在此loadFactor和capacity对应下允许的最大元素数目，超过这个数目就重新resize，以降低实际的负载因子。默认的的负载因子0.75是对空间和时间效率的一个平衡选择。当容量超出此最大容量时， resize后的HashMap容量是容量的两倍：

if (size++ >= threshold)

resize(2 \* table.length);

**7、Fail-Fast机制：**

java.util.HashMap不是线程安全的，如在使用迭代器过程中有其他线程修改了map，将抛出ConcurrentModificationException，就是所谓fail-fast策略。

源码中的实现是通过modCount域，就是修改次数，对HashMap内容的修改都将增加这个值，在迭代器初始化过程中会将这个值赋给迭代器的expectedModCount。

|  |
| --- |
| private abstract class HashIterator<E> implements Iterator<E> {  Entry<K,V> next; // next entry to return  int expectedModCount; // For fast-fail  int index; // current slot  Entry<K,V> current; // current entry  HashIterator() {  expectedModCount = modCount;  if (size > 0) { // advance to first entry  Entry[] t = table;  while (index < t.length && (next = t[index++]) == null)  ;  }  }  public final boolean hasNext() {  return next != null;  }  final Entry<K,V> nextEntry() {  if (modCount != expectedModCount)  throw new ConcurrentModificationException();  Entry<K,V> e = next;  if (e == null)  throw new NoSuchElementException();  if ((next = e.next) == null) {  Entry[] t = table;  while (index < t.length && (next = t[index++]) == null)  ;  }  current = e;  return e;  }  public void remove() {  if (current == null)  throw new IllegalStateException();  if (modCount != expectedModCount)  throw new ConcurrentModificationException();  Object k = current.key;  current = null;  HashMap.this.removeEntryForKey(k);  expectedModCount = modCount;    } |

迭代过程中，判断modCount跟expectedModCount是否相等，如不相等就表示已有其他线程修改了Map：

注意到modCount声明为volatile，保证线程间修改的可见性。

|  |
| --- |
| final Entry<K,V> nextEntry() {  if (modCount != expectedModCount)  throw new ConcurrentModificationException(); |

在HashMap的API中指出：

由所有HashMap类的“collection 视图方法”所返回的迭代器都是快速失败的：迭代器创建后，如果从结构上对映射进行修改，除非通过迭代器本身的 remove 方法，其他任何时间任何方式的修改，迭代器都将抛出 ConcurrentModificationException。因此，面对并发的修改，迭代器很快就会完全失败，而不冒在将来不确定时间发生任意不确定行为的风险。

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误

|  |
| --- |
| 为了保证线程安全，在迭代器迭代的过程中，线程是不能对集合本身进行操作（修改，删除，增加）的，否则会抛ConcurrentModificationException的异常。 |

另一总结:

HashMap基于hashing原理，通过put()和get()储存和获取对象。

将键值对传递给put()方法时，调用键对象的hashCode()计算hashcode，让后找到bucket位置来储存值对象。

获取对象时，通过键对象的equals()方法找到正确的键值对，返回值对象。

当两个不同的键对象的hashcode相同时，会储存在同一个bucket位置的链表中。 HashMap在每个链表节点中储存键值对对象。

**④题目**

HashMap的工作原理实现HashMap。ConcurrentHashMap和其它同步集合的引入

* “你用过HashMap吗？” “什么是HashMap？你为什么用到它？”

HashMap可接受null键值和值，Hashtable则不能；非synchronized;很快;储存的是键值对等。

* “你知道HashMap的工作原理吗？” “你知道HashMap的get()方法的工作原理吗？”

“HashMap基于hashing的原理，使用put(key, value)存储对象到HashMap中，使用get(key)从HashMap中获取对象。

给put()传递键和值时，先对键调用hashCode()，返回的hashCode用于找到bucket位置来储存Entry对象。”关键点，HashMap是在bucket中储存键对象和值对象，作为Map.Entry。。

* “当两对象的hashcode相同会发生什么？”

“因hashcode同，bucket位置同，‘碰撞’会发生。因HashMap用链表存储对象，这个Entry(含有键值对的Map.Entry对象)会存储在链表中。”

* “如果两个键的hashcode相同，你如何获取值对象？”

调用get()，HashMap会用键对象的hashcode找到bucket位置，之后会调用keys.equals()找到链表中正确的节点，最终找到要找的值对象。

* “如HashMap的大小超过了负载因子(load factor)定义的容量，怎么办？”

默认负载因子大小为0.75，即当一个map填满了75%的bucket时候，将会创建原来HashMap大小的两倍的bucket数组，来重新调整map的大小，并将原来的对象放入新的bucket数组中。这个过程叫作rehashing，因为它调用hash方法找到新的bucket位置。

* “重新调整HashMap大小存在什么问题吗？”

多线程的情况下，可能产生条件竞争(race condition)。

如两线程都发现HashMap需重新调整大小，会同时试着调整。

调整过程中，存储在链表中的元素的次序会反过来，因移动到新的bucket位置时，HashMap不将元素放在链表头部，为了避免尾部遍历(tail traversing)。如条件竞争发生，就死循环了。

|  |
| --- |
| 因是链表结构，就很容易形成闭合链路，在循环时只要有线程对这HashMap进行get操作就会产生死循环。  只有在多线程并发情况下，就是在put时，如size>initialCapacity\*loadFactor，进行rehash操，随之HashMap结构变化。有可能在两线程在同时触发rehash，产生了闭合的回路。 |

* 为什么String, Interger这样的wrapper类适合作为键？

使用不可变的、声明作final的对象，且采用合适的equals()和hashCode()方法的话，将会减少碰撞的发生，提高效率。不可变性使得能够缓存不同键的hashcode，这将提高整个获取对象的速度，使用String，Interger这样的wrapper类作为键是非常好的选择。

且String最为常用。因String不可变，也是final的，且已重写了equals()和hashCode()方法了。其他的wrapper类也有这个特点。

不可变性是必要的，因为为了要计算hashCode()，就要防止键值改变，如键值在放入时和获取时返回不同的hashcode的话，就不能从HashMap中找到想要的对象。

不可变性还有其他优点如线程安全。如可以仅通过将某个field声明成final就能保证hashCode不变，那么请这么做吧。因为获取对象的时候要用到equals()和hashCode()方法，那么键对象正确的重写这两个方法是非常重要的。

如两不相等对象返回不同hashcode，碰撞的几率就会小些，就能提高HashMap的性能。

* 可使用自定义对象作为键吗？

可用任何对象作为键，只要遵守了equals()和hashCode()方法的定义规则，且当对象插入到Map中后将不会再改变了。如这个自定义对象是不可变的，那么它已经满足了作为键的条件，因为当它创建之后就已经不能改变了。

<http://www.importnew.com/22011.html>

**五．多线程下hashmap问题(hash碰撞与扩容导致)**

|  |  |
| --- | --- |
| 1、多线程put操作后，get操作导致死循环。  多个线程put时造成个key值Entry key List的死循环. 当另一线程get 这个Entry List 死循环的key时，get也会一直执行。结果是越来越多的线程死循环，导致服务器dang掉  2、多线程put非NULL元素，get NULL值。   |  | | --- | | 在transfer方法里，将旧数组赋值给src，遍历src，当src的元素非null时，就将src中的该元素置null，即将旧数组中的元素置null了  void transfer(Entry[] newTable) {  Entry[] src = table;  int newCapacity = newTable.length;  for (int j = 0; j < src.length; j++) {  Entry e = src[j];  if (e != null) {  src[j] = null;  do {  Entry next = e.next;  int i = indexFor(e.hash, newCapacity);  e.next = newTable[i];  newTable[i] = e; e = next; }  while (e != null); } } } |   3、多线程put操作，导致元素丢失。  问题出在addEntry()的new Entry (hash, key, value, e)  http://img.blog.csdn.net/20150814144614564?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQv/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/Center   * 多个线程同时put，如图中a1、a2(hash值相同)，这时需要解决碰撞冲突，这时两线程如果恰好都取到了对应位置的头结点e1，则他们下一个元素都是e, 最终，a1、a2两数据中必会有一个丢失 |

**五．CocurrentHashMap**

* 可用CocurrentHashMap代替Hashtable吗？

ConcurrentHashMap同步性能更好，因为它仅根据同步级别对map的一部分进行上锁。可代替HashTable，但HashTable提供更强的线程安全性。

ConcurrentHashMap提供了和Hashtable及SynchronizedMap中不同的锁机制。

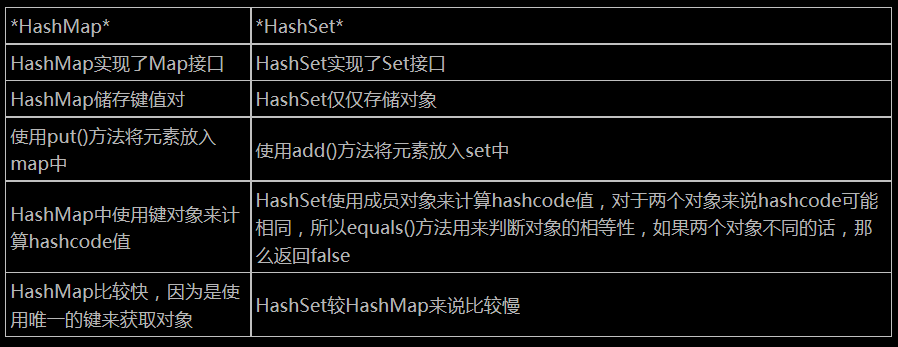
Hashtable的锁机制是一次锁住整个hash表，从而同一时刻只能由一个线程对其进行操作；

而ConcurrentHashMap中则是一次锁住一个桶。默认将hash表分为16个桶，诸如get,put,remove等常用操作只锁当前需要用到的桶。这样，原来只能一个线程进入，现在却能同时有16个写线程执行，并发性能的提升是显而易见的。

读操作大部分都不需用到锁。只有在size等操作时才需要锁住整个hash表。

**1.4.2 HashMap和HashSet的区别**

|  |
| --- |
| hashCode不同时，则必为不同对象。  hashCode相同时，根据equlas()方法判断是否为同一对象。 |



* HashSet

实现了Set接口，不允许集合中有重复值

将对象存储在HashSet前，要先确保对象重写equals()和hashCode()方法，才能比较对象的值是否相等，以确保set中没有储存相等的对象。

public boolean add(Object o)方法用来在Set中添加元素，当元素值重复时则会立即返回false，如果成功添加的话会返回true

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| * 没有重写equals   示例代码   |  | | --- | | //2. 验证唯一性，存入自定义元素  class Person{  private String name;  private int age;    public static void sop(Object o){  System.out.println(o);  }    public Person(String name, int age){  this.name =name;  this.age =age;  }    public void setName(String name){  this.name =name;  }    public String getName(){  return this.name;  }    public void setAge(int age){  this.age =age;  }    public int getAge(){  return this.age;  }    public String toString(){  return this.name+"::"+this.age;  }  } |   测试代码：   |  | | --- | | class HashSetDemoIII  {  public static void sop(Object o){  System.out.println(o);  }    public static void main(String[] args){  HashSeths =new HashSet();  hs.add(new Person("a1", 11));  hs.add(new Person("a2", 12));  hs.add(new Person("a3", 13));  hs.add(new Person("a2", 12));    Iteratorit =hs.iterator();  while(it.hasNext()){  sop(it.next());  }  }  } |   测试结果：  http://img.blog.csdn.net/20130725142618125?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMTQwNjEyNA==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast  【结论】结果：a2::12被打印了两次，说明重复元素new Person("a2", 12)被添加到集合中去。没有保证HashSet的唯一性。  {3}. 为Person类重写equals()方法判定两个对象的内容是否一致   |  | | --- | | public boolean equals(Object obj){  if(!(obj instanceof Person)){  return false;  }    Personp =(Person)obj;    //用来查看equals方法是否被调用  sop(this.name +".......equals......."+ p.name);  //认为名字相同并且年龄一样大的两个对象是一个  return this.name.equals(p.name) && this.age == p.age;  } |   测试结果：  http://img.blog.csdn.net/20130725142618125?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMTQwNjEyNA==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast  【结论】循环迭代结果：a2::12被打印了两次，说明重复元素new Person("a2", 12)被添加到集合中去。没有保证HashSet的唯一性。  同时equals没有被调用，说明仅重写equals方法没起到作用  【分析】HashSet在存储元素的时候，是先查看两个对象的哈希值是否一样的。  {3}【1】.  如果哈希值是一样，再调用元素对象的equals方法，查看两个对象的内容是否一样。  {1}. 如果equlas比较的值一样，说明两个对象相同，不能添加到HashSet对应的集合中。  {2}. 如果equlas比较的值不一样，说明两个对象不同，可以添加到HashSet对应的集合中  {3}【2】.  如哈希值不一样，直接认为两对象不一样，equals方法不起作用。  【本例分析】这个例子中，由于四个Person类的对象都是new出来的，所以没有重写的hashCode是调用本地方法产生的值，所以一定是四个不相同的hashCode值。hashCode值都不一样，equals方法不起作用。  (3). 重写HashCode方法  为避免内容重复的数据添加到HashSet集合中，就必须重写hashCode()方法。  [2]. 根据比较的条件重写HashCode方法   |  | | --- | | public int hashCode(){  sop(this.name +"......hashCode");  return this.name.hashCode() + 29\*age;  } |   这种做法，同样可避免HashSet集合中不会存入相同元素。  2. **HashSet总结**  1). HashCode如何保证元素唯一性？  (1). 通过重写自定义类的hashCode()和equals( )两个方法来达到元素的唯一性保证。  (2). 方法重写后，执行的顺序  [1]. 如果元素的hashCode值相同，才会使用equals()方法进行判断。  [2]. 如果元素的hashCode值不同，不会调用equals()方法。   |  | | --- | | equals()相等的两对象，hashcode()一定相等；  equals()不等的两对象，hashcode()可能相等。  hashcode()不等， equals()一定不等；  hashcode()相等，equals()可能相等，也可能不等 |   3). HashSet判断和删除元素的依据  示例代码：   |  | | --- | | public class HashSetDemoIV {  public static void main(String[] args) {  HashSeths =new HashSet();  hs.add(new Person("a1", 11));  hs.add(new Person("a2", 12));  hs.add(new Person("a3", 13));  System.out.println("hashSet中的元素："+ hs);  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("HashSet的 contains依据："+ hs.contains(new Person("a1", 11)));    System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("HashSet的 remove依据："+ hs.remove(new Person("a3", 13)));  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");  System.out.println("hashSet中的元素："+ hs);  }  } |   打印结果：  http://img.blog.csdn.net/20130725142939718?watermark/2/text/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdTAxMTQwNjEyNA==/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70/gravity/SouthEast  hashSet添加、删除和判断元素时，依据的仍是hashCode和equals这两方法。  ArrayList的equals方法只用到数据项的equals方法  而HashMap和HashSet用到了数据项的hashCode和equals方法 |

* HashMap

实现了Map接口，对键值对进行映射。键不可重复。

TreeMap保存了对象的排列次序，而HashMap则不能。

HashMap允许键和值为null。非synchronized的。

**1.4.3 HashMap和HashTable的区别，实现原理**

|  |
| --- |
| HashTable底层用HashMap实现，区别是，HashTable是按存入顺序排序的，而HashMap不是。  HashMap的原理是有一个大的table数组组成，每个数组元素是一个Entry。为了处理冲突，通常会将Entry用链表实现。  ②HashMap和Hashtable的区别  主要的区别：线程安全性，同步(synchronization)，速度。  HashMap是非synchronized的，可接受null的key和value，Hashtable则不行  HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。  当有其它线程改变了HashMap的结构（增加或移除元素），会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出该异常。不是一个一定发生的行为，要看JVM。同样也是Enumeration和Iterator的区别。  Hashtable是线程安全的也是synchronized，单线程环境下比HashMap要慢。  要注意的一些重要术语：  2) Fail-safe和iterator迭代器相关。如果某个集合对象创建了Iterator或者ListIterator，然后其它的线程试图“结构上”更改集合对象，将会抛出ConcurrentModificationException异常。但其它线程可通过set()方法更改集合对象，因这并没有从“结构上”更改集合。但假如已经从结构上进行了更改，再调用set()方法，将会抛出IllegalArgumentException异常。  3) 结构上的更改指的是删除或插入一个元素，会影响到map的结构。 |

**1.4.4 ArrayList,LinkedList和Vector的区别和实现原理。**

|  |
| --- |
| ArrayList基于数组的可变长数组，更适合实现get和set；  LinkedList基于双向链表，较适合实现插入和删除；  以上两都是非线程安全的  Vector的实现和ArrayList差不多，改进的地方是使用synchronized实现了线程安全。 |

**1.4.5 TreeMap和TreeSet区别和实现原理。**

TreeMap:Map 接口, TreeSet: Set 接口。

TreeSet 底层是通过 TreeMap 实现，因此二者的实现方式完全一样。

而 TreeMap 的实现就是红黑树算法。

* 相同点：

都是有序集合

都是非同步集合，不能在多线程间共享，可使用Collections.synchroinzedMap()实现同步

运行速度都比Hash集合慢，内部对元素的操作时间复杂度为O(logN)，而HashMap/HashSet则为O(1)。

* 不同点：

TreeSet和TreeMap分别实现Set和Map接口

TreeSet只存储一个对象，而TreeMap存储两个对象Key和Value（仅仅key对象有序）

TreeSet中不能有重复对象，而TreeMap中可存在

TreeMap的底层采用红黑树的实现，完成数据有序的插入，排序。

**1.5 【F】Queue接口**

**1.5.1 Queue接口**

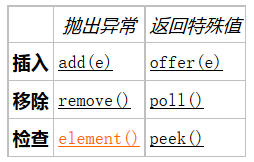
FCFS算法(First Come, First Serve)。

处理元素前保存元素。除基本Collection 操作，队列还提供插入、提取和检查操作。

* 每个方法都有两种形式：

|  |
| --- |
| 一种抛出异常（操作失败时），  另一种返回一特殊值（null 或 false，具体取决于操作）。 |

插入操作的后一种形式是为有容量限制的 Queue 设计；大多数实现中，插入操作不会失败。



**头**

队列常以FIFO（先进先出）方式排序各元素。

例外：

|  |
| --- |
| 优先级队列：根据提供的比较器或元素自然顺序对元素排序  LIFO （后进先出）队列（或堆栈） |

无论用哪种排序，队列头都是调用 remove() 或 poll() 所移除的元素。

|  |
| --- |
| * offer()可插入一元素，否则返回 false。   Collection.add只能通过抛出未经检查的异常使添加元素失败。  offer()方法设计用于正常的失败情况，而不是出现异常的情况，如在有界队列中。   * remove() 和 poll() 可移除和返回队列头。到底移除哪个元素是队列排序策略的功能，。队列为空时：remove() 抛异常，而poll() 返回 null。 * element() 和 peek() 返回，但不移除，队列的头。 |

* **Queue 接口并未定义阻塞队列方法**

**BlockingQueue 接口**定义了那些等待元素出现或等待队列中有可用空间的方法，扩展了此接口。

* **Queue 实现通常不允许插入null 元素**

尽管LinkedList不禁止插入 null。但也不应将 null 插入到 Queue 中，因 null 也用作 poll()的返回值，表明队列不包含元素。

* Queue接口**方法列表**如下：

|  |
| --- |
| boolean add(E e) 将指定的元素插入此队列（如果立即可行且不会违反容量限制），在成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。  E element() 获取，但是不移除此队列的头。  boolean offer(E e) 将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，此方法通常要优于 add(E)，后者可能无法插入元素，而只是抛出一个异常。  E peek() 获取但不移除此队列的头；如果此队列为空，则返回 null。  E poll() 获取并移除此队列的头，如果此队列为空，则返回 null。  E remove() 获取并移除此队列的头。 |

在List接口中已经说过，LinkedList也实现了Queue接口，可作为普通队列的容器，在这里不再赘述

**1.5.2 阻塞队列**

**1.5.1.1 PriorityQueue**

**无界优先级**队列。

|  |
| --- |
| 元素  (1)按自然顺序排序：不允许插入不可比较对象 ClassCastException  (2)或根据构造队列时提供的 Comparator 排序，取决于用的构造方法。 |

* 不允许用 null 元素。
* 头是最小元素。如多个元素都是最小值，则头是其中一个元素—选择方法任意。
* 无界，但有一内部容量（默认11），控制着用于存储队列元素的数组大小。随元素增加自动扩容。
* 此类及其迭代器实现了 Collection 和 Iterator 接口所有方法。

iterator() 中迭代器不保证以任何顺序遍历优先级队列中的元素。

如需要按顺序遍历，用 Arrays.sort(pq.toArray())。

* 扩展的构造方法：

|  |
| --- |
| * PriorityQueue(int initialCapacity)   用指定初始容量创建一PriorityQueue，根据其自然顺序对元素进行排序。   * PriorityQueue(int initialCapacity, Comparator<? super E> comparator)   用指定初始容量创建一PriorityQueue，并根据指定比较器对元素进行排序。   * PriorityQueue(PriorityQueue<?extends E> c)   创建包含指定优先级队列元素的PriorityQueue。   * PriorityQueue(SortedSet<?extends E> c)   创建包含指定有序set元素的 PriorityQueue。 |

**1.5.1.2 ArrayBlockingQueue**

**数组**支持的**有界阻塞**队列。FIFO原则对元素排序。

队列的头部是在队列中存在时间最长的元素。

新元素插入到队尾，队列获取从队头

典型的“有界缓存区”，固定大小的数组在其中保持生产者插入的元素和使用者提取的元素。一旦创建，就不能再增加其容量。

试图向已满队列中放入元素会导致操作受阻塞；

试图从空队列中提取元素将导致类似阻塞。

此类支持对等待的生产者线程和使用者线程进行排序的可选公平策略。默认不保证。

通过将公平性 (fairness) 设为 true 构造的队列允许按 FIFO 顺序访问线程。

|  |
| --- |
| 如fairness = true，等待时间最长的线程会优先处理（其实就是通过将ReentrantLock设置为true来达到这种公平性：即等待时间最长的线程会先操作）。 |

公平性通常会降低吞吐量，但也减少了可变性和避免了“不平衡性”。

* 用阻塞队列可很方便的实现生产者-消费者模型：

|  |
| --- |
| //**生产者**  class Producer implements Runnable{  private final BlockingQueue<Integer> queue;    public Producer(BlockingQueue<Integer> q){  this.queue = q;  }    public void run() {  while(true){  try {  queue.put(1);  //生产者不断地往队列添加产品，当队列满时，该线程被阻塞  System.out.println("生产者线程：+1，队列长度："+queue.size());  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } |

|  |
| --- |
| //消费者  class Consumer implements Runnable{  private final BlockingQueue<Integer> queue;    public Consumer(BlockingQueue<Integer> q){  this.queue = q;  }    public void run() {  while(true){  try {  queue.take(); //消费者不断消耗产品，当队列已空时，该线程会被阻塞  System.out.println("消费者线程：-1，队列长度："+queue.size());  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  } |

|  |
| --- |
| import java.util.concurrent.ArrayBlockingQueue;  import java.util.concurrent.BlockingQueue;  public class Test {  public static void main(String[] args) {  //定义一个容量为5的阻塞队列  BlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);    Producer p = new Producer(queue);  Consumer c = new Consumer(queue);    new Thread(p).start();  new Thread(c).start();  }  } |

**1.5.1.3 ConcurrentLinkedQueue**

用于并发环境

**1.5.3 Deque接口（双端队列）**

* Queue的子接口。
* 支持在两端插入和移除元素。 “double ended queue（双端队列）”的缩写，读为“deck”。
* 大多Deque 实现对于它们能包含的元素数没固定限制，支持有容量限制的或没固定大小限制的双端队列。
* 提供插入、移除和检查元素的方法。每种方法存在两种形式：



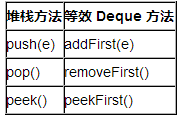
* 将双端队列用作队列时，将得到FIFO行为。元素添加到队末，从队头移除元素。
* 从 Queue 接口继承的方法完全等效于 Deque 方法，如下表所示：



* 也可用作 LIFO（后进先出）堆栈。应优先用此接口而不是Stack 类。

元素推入队头并从队头弹出。

堆栈方法完全等效于 Deque 方法，如下表所示：



|  |
| --- |
| 将双端队列用作队列或堆栈时，peek()同样正常工作；都从队头抽取元素。  此接口提供了两种移除内部元素的方法：removeFirstOccurrence和 removeLastOccurrence。  与 List 接口不同，此接口不支持通过索引访问元素。  虽 Deque 实现没严格要求禁止插入null 元素，但建议最好这样做。因会将 null 用作特殊的返回值来指示双端队列为空。  LinkedList实现了Deque接口，再次不再赘述 |

**1.5.3.2 ArrayDeque(数组双端队列)**

|  |
| --- |
| Deque 接口的可变大小数组的实现。  没容量限制；可动态扩容。  非线程安全；没外部同步时，不支持多个线程的并发访问。  禁止null 元素。  用作堆栈时快于 Stack，在用作队列时快于LinkedList。 |

* 大多 ArrayDeque 操作以摊销的固定时间运行。异常包括 remove、removeFirstOccurrence、removeLastOccurrence、contains、iterator.remove() 以及批量操作，均以线性时间运行。
* 此类及其迭代器实现 Collection 和 Iterator 接口所有可选方法。
* 扩展的构造方法：

ArrayDeque(int numElements) 构造一初始容量能够容纳指定数量元素的空数组双端队列。

**1.5.3.3 LinkedBlockingDeque**

基于已链接节点的、任选范围的阻塞双端队列。

可选的容量范围构造方法参数是一种防止过度膨胀的方式。

如未指定容量，容量= Integer.MAX\_VALUE。只要插入元素不会使双端队列超出容量，每次插入后都将动态地创建链接节点。

大多数操作以固定时间运行（不计阻塞消耗时间）。异常包括 remove、removeFirstOccurrence、removeLastOccurrence、contains、iterator.remove() 以及批量操作，均以线性时间运行。

**1.6 NIO**

NIO（Non-blocking I/O，，也称为New I/O），同步非阻塞的I/O模型，I/O多路复用的基础，已被越来越多地应用到大型应用服务器，成为解决高并发与大量连接、I/O处理问题的有效方式。

那么NIO的本质是什么样？是怎样与事件模型结合来解放线程、提高系统吞吐的呢？

本文会从传统阻塞I/O和线程池模型面临的问题讲起，对比常见I/O模型，分析NIO怎么利用事件模型处理I/O，解决线程池瓶颈处理海量连接，包括利用面向事件的方式编写服务端/客户端程序。最后延展到一些高级主题，如Reactor与Proactor模型的对比、Selector的唤醒、Buffer的选择等。

注：本文都是伪代码，为了示意，不可用于生产环境。

**1.6.1 传统BIO模型分析**

先回忆一下传统服务器端同步阻塞I/O处理（也就是BIO，Blocking I/O）的经典编程模型：

|  |
| --- |
| {  ExecutorService executor = Excutors.newFixedThreadPollExecutor(100);//线程池  ServerSocket serverSocket = new ServerSocket();  serverSocket.bind(8088);  while(!Thread.currentThread.isInturrupted()){ //主线程死循环等待新连接到来  Socket socket = serverSocket.accept();  executor.submit(new ConnectIOnHandler(socket));//为新连接创建新线程  }  class ConnectIOnHandler extends Thread{  private Socket socket;  public ConnectIOnHandler(Socket socket){  this.socket = socket;  }  public void run(){  while(!Thread.currentThread.isInturrupted()&&!socket.isClosed()){死循环处理读写事件  String someThing = socket.read()....//读取数据  if(someThing!=null){  ......//处理数据  socket.write()....//写数据  }  }  }  } |

这是一个经典的每连接每线程的模型，之所以用多线程，原因在于socket.accept()、socket.read()、socket.write()三个主要函数都是同步阻塞的，当一个连接处理I/O时，系统阻塞，如是单线程必然就挂死在那里；但CPU是被释放出来的，开启多线程，就可让CPU去处理更多事情。其实也是所有使用多线程的本质：

|  |
| --- |
| 1.利用多核。  2.当I/O阻塞系统，但CPU空闲时，可利用多线程使用CPU资源。 |

多线程一般都用线程池，可让线程的创建和回收成本相对较低。

活动连接数不是特别高（小于单机1000）情况下，这种模型较不错，可让每个连接专注于自己的I/O且编程模型简单，也不用过多考虑系统的过载、限流等问题。线程池本身就是一天然漏斗，可缓冲一些系统处理不了的连接或请求。

不过，这个模型最本质的问题在于，严重依赖于线程。但线程是很"贵"的资源，表现在：

|  |
| --- |
| * 线程的创建和销毁成本很高，在Linux中，线程本质上是一进程。创建和销毁都是重量级的系统函数。 * 线程本身占用较大内存，Java线程栈，一般至少分配512K～1M，如系统中的线程数过千，恐怕整个JVM的内存都会被吃掉一半。 * 线程的切换成本很高。操作系统发生线程切换时，需保留线程上下文，然后执行系统调用。如线程数过高，可能执行线程切换时间甚至会大于线程执行时间，表现是系统load偏高、CPU sy使用率特别高（超20%)，导致系统几乎陷入不可用状态。 * 容易造成锯齿状的系统负载。因系统负载是用活动线程数或CPU核心数，一旦线程数量高但外部网络环境不稳定，很容易造成大量请求的结果同时返回，激活大量阻塞线程从而使系统负载压力过大。 |

当面对十万/百万级连接时，传统BIO模型无能为力。随移动端应用的兴起和各种网络游戏的盛行，百万级长连接日趋普遍，必然需要一种更高效的I/O处理模型。

**1.6.2 NIO怎么工作**

Java相对晦涩的API，如：Channel，Selector，Socket；然后就是一坨上百行的代码来演示NIO的服务端Demo……瞬间头大有没有？

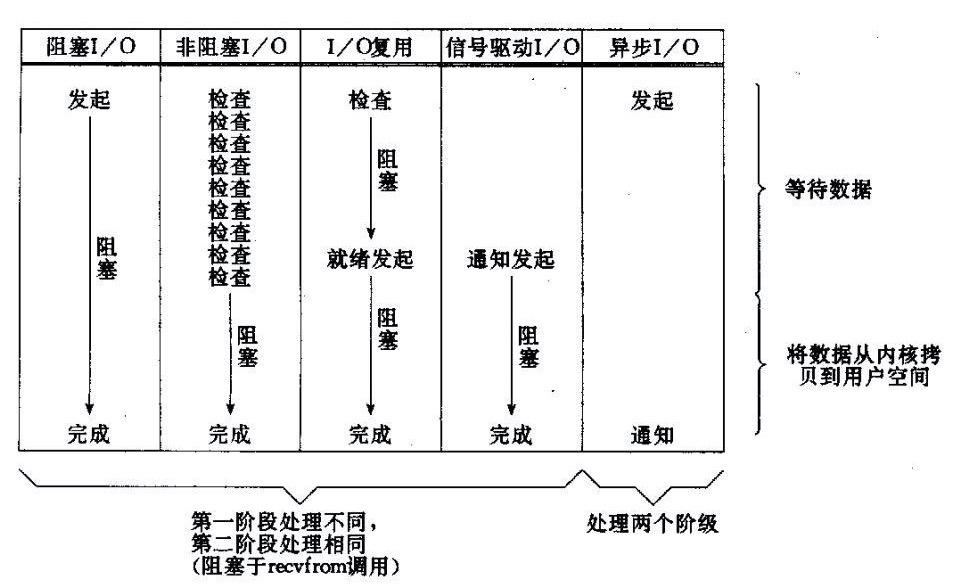
**1.6.2.1 常见I/O模型对比**

* 系统I/O分两阶段：等待就绪和操作。

举例，读函数，分为等待系统可读和真正的读；写函数分为等待网卡可写和真正的写。

注:等待就绪的阻塞不使用CPU，“空等”；而真正读写操作的阻塞使用CPU，真正在"干活"，且这个过程非常快，属于memory copy，带宽常在1GB/s以上，可理解为基本不耗时。

* 常见I/O模型对比：



* 以socket.read()为例：

|  |
| --- |
| * BIO里socket.read()，如TCP RecvBuffer里没数据，函数会一直阻塞，直到收到数据，返回读到的数据。 * NIO，如TCP RecvBuffer有数据，把数据从网卡读到内存，且返回给用户；反之则直接返回0，永远不会阻塞。 * AIO(Async I/O)更进一步：等待就绪非阻塞，数据从网卡到内存过程也异步。 |

即BIO里用户最关心“我要读”，NIO里用户最关心"我可以读了"，AIO里用户更需关注“读完了”。

* NIO重要特点：socket主要的读、写、注册和接收函数，在等待就绪阶段都是非阻塞的，真正的I/O操作是同步阻塞的（消耗CPU但性能非常高）。

**1.6.2.2 如何结合事件模型使用NIO同步非阻塞特性**

BIO之所以要多线程，因为进行I/O时，一是没办法知道到底能不能写、能不能读，只能"傻等"，即使通过各种估算，算出来操作系统没有能力进行读写，也没法在socket.read()和socket.write()函数中返回，这两个函数无法进行有效的中断。所以除了多开线程另起炉灶，没有好的办法利用CPU。

NIO的读写函数可立刻返回，就给了不开线程利用CPU的最好机会：如一连接不能读写（socket.read()返回0或者socket.write()返回0），可把这件事记下来，在Selector上注册标记位，然后切换到其它就绪的连接继续读写。

* 下面具体看如何用事件模型单线程处理所有I/O请求：

NIO主要事件：读就绪、写就绪、有新连接到来。

* 先需注册事件到来时对应的处理器。然后在合适时机告诉事件选择器：我对这个事件感兴趣。

|  |
| --- |
| 写操作，就是写不出去时对写事件感兴趣；  读操作，完成连接和系统没办法承载新读入数据时；  accept，服务器刚启动时；  connect，connect失败需重连或直接异步调用connect时。 |

* 其次，用个死循环选择就绪事件，会执行系统调用（Linux 2.6之前是select、poll，2.6之后是epoll，Windows是IOCP），还会阻塞的等待新事件到来。新事件到来时，会在selector上注册标记位，标示可读、可写或有连接到来。

注意，select是阻塞的，无论通过操作系统的通知（epoll）还是不停轮询(select，poll)，都是阻塞的。所以你可放心大胆地在一个while(true)里面调用这个函数而不用担心CPU空转。

程序大概模样是：

|  |
| --- |
| interface ChannelHandler{  void channelReadable(Channel channel);  void channelWritable(Channel channel);  }  class Channel{  Socket socket;  Event event;//读，写或连接  }  //IO线程主循环:  class IoThread extends Thread{  public void run(){  Channel channel;  while(channel=Selector.select()){//选择就绪事件和对应连接  if(channel.event==accept){  registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读写处理器  }  if(channel.event==write){  getChannelHandler(channel).channelWritable(channel);//如果可写，则执行写事件  }  if(channel.event==read){  getChannelHandler(channel).channelReadable(channel);//如果可读，则执行读事件  }  }  }  Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器  } |

最简单的Reactor模式：注册所有感兴趣的事件处理器，单线程轮询选择就绪事件，执行事件处理器。

**1.6.2.3 优化线程模型**

可总结出NIO是怎么解决掉线程的瓶颈并处理海量连接的：

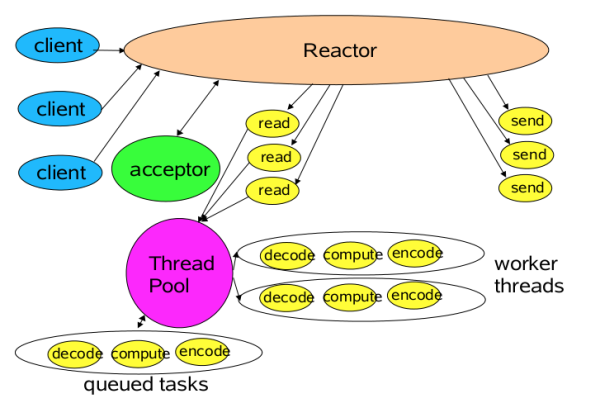
|  |
| --- |
| NIO由原来阻塞读写（占用线程）变成了单线程轮询事件，找到可进行读写的网络描述符进行读写。  除事件的轮询阻塞（没可干的事情必须要阻塞），剩余I/O操作都是纯CPU操作，没必要开启多线程。  且由于线程的节约，连接数大时因线程切换带来的问题也随之解决，进而为处理海量连接提供了可能。  单线程处理I/O效率非常高，没线程切换，只是拼命的读、写、选择事件。但现在服务器，一般都是多核处理器，如能够利用多核心进行I/O，无疑对效率会有更大的提高。 |

* 仔细分析一下我们需要的线程，其实主要包括以下几种：

|  |
| --- |
| * 事件分发器，单线程选择就绪的事件。 * I/O处理器,包括connect、read、write，这种纯CPU操作，开启CPU核心个线程就可 * 业务线程，处理完I/O后，一般还会有业务逻辑，有的还会有其他阻塞I/O，如DB操作，RPC等。只要有阻塞，就需单独线程。 |

Java的Selector对于Linux系统来说，有一致命限制：同一channel的select不能被并发的调用。因此，如有多个I/O线程，须保证：一个socket只能属于一个IoThread，而一个IoThread可管理多个socket。

另外连接的处理和读写的处理常可选择分开，对于海量连接的注册和读写就可分发。虽read()和write()是较高效无阻塞的函数，但毕竟会占用CPU，如面对更高的并发则无能为力。



**1.6.2.4NIO在客户端的魔力**

可看出NIO在服务端对于解放线程，优化I/O和处理海量连接方面，有用武之地。

常见的客户端BIO+连接池模型，可建立n个连接，当某一连接被I/O占用时，可用其他连接来提高性能。

但多线程的模型面临和服务端相同的问题：如指望增加连接数来提高性能，则连接数又受制于线程数、线程很贵、无法建立很多线程，则性能遇到瓶颈。

**1.6.2.5 每连接顺序请求的Redis**

Redis，由于服务端全局串行，能保证同一连接的请求与返回顺序一致。

可用单线程＋队列，把请求数据缓冲。

然后pipeline发送，返回future.

然后channel可读时，直接在队列中把future取回来，done()就可以了。

伪代码如下：

|  |
| --- |
| class RedisClient Implements ChannelHandler{  private BlockingQueue CmdQueue;  private EventLoop eventLoop;  private Channel channel;  class Cmd{  String cmd;  Future result;  }  public Future get(String key){  Cmd cmd= new Cmd(key);  queue.offer(cmd);  eventLoop.submit(new Runnable(){  List list = new ArrayList();  queue.drainTo(list);  if(channel.isWritable()){  channel.writeAndFlush(list);  }  });  }  public void ChannelReadFinish(Channel channel，Buffer Buffer){  List result = handleBuffer();//处理数据  //从cmdQueue取出future，并设值，future.done();  }  public void ChannelWritable(Channel channel){  channel.flush();  }  } |

这样做，能充分利用pipeline来提高I/O能力，同时获取异步处理能力。

**1.6.2.6 多连接短连接的HttpClient**

类似于竞对抓取的项目，需建立无数HTTP短连接，然后抓取，销毁

* 当需单机抓取上千网站线程数又受制的时候，怎么保证性能?

何不尝试NIO，单线程进行连接、写、读操作？如果连接、读、写操作系统没有能力处理，简单的注册一个事件，等待下次循环就好了。

* 如何存储不同的请求/响应呢？

http是无状态没有版本的协议，又没办法用队列，好像办法不多。较笨的办法是对不同的socket，直接存储socket的引用作为map的key。

**1.6.2.7 【不理解】常见的RPC框架，如Thrift，Dubbo**

内部一般维护了请求的协议和请求号，可维护一个以请求号为key，结果的result为future的map，结合NIO+长连接，获取非常不错的性能。

**1.6.3【没看】NIO高级主题**

**1.6.3.1 Proactor与Reactor**

一般情况下，I/O 复用机制需要事件分发器（event dispatcher）。 事件分发器的作用，即将那些读写事件源分发给各读写事件的处理者，就像送快递的在楼下喊: 谁谁谁的快递到了， 快来拿吧！开发人员在开始的时候需要在分发器那里注册感兴趣的事件，并提供相应的处理者（event handler)，或者是回调函数；事件分发器在适当的时候，会将请求的事件分发给这些handler或者回调函数。

涉及到事件分发器的两种模式称为：Reactor和Proactor。 Reactor模式是基于同步I/O的，而Proactor模式是和异步I/O相关的。在Reactor模式中，事件分发器等待某个事件或者可应用或个操作的状态发生（比如文件描述符可读写，或者是socket可读写），事件分发器就把这个事件传给事先注册的事件处理函数或者回调函数，由后者来做实际的读写操作。

而在Proactor模式中，事件处理者（或者代由事件分发器发起）直接发起一个异步读写操作（相当于请求），而实际的工作是由操作系统来完成的。发起时，需要提供的参数包括用于存放读到数据的缓存区、读的数据大小或用于存放外发数据的缓存区，以及这个请求完后的回调函数等信息。事件分发器得知了这个请求，它默默等待这个请求的完成，然后转发完成事件给相应的事件处理者或者回调。举例来说，在Windows上事件处理者投递了一个异步IO操作（称为overlapped技术），事件分发器等IO Complete事件完成。这种异步模式的典型实现是基于操作系统底层异步API的，所以我们可称之为“系统级别”的或者“真正意义上”的异步，因为具体的读写是由操作系统代劳的。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（写操作类似）。

**1.6.3.2 在Reactor中实现读**

|  |
| --- |
| 注册读就绪事件和相应的事件处理器。  事件分发器等待事件。  事件到来，激活分发器，分发器调用事件对应的处理器。  事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。 |

**1.6.3.3 在Proactor中实现读：**

|  |
| --- |
| 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。  事件分发器等待操作完成事件。  在分发器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分发器读操作完成。  事件分发器呼唤处理器。  事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分发器。 |

可看出，两个模式的相同点，都是对某个I/O事件的事件通知（即告诉某个模块，这个I/O操作可以进行或已经完成)。在结构上，两者也有相同点：事件分发器负责提交IO操作（异步)、查询设备是否可操作（同步)，然后当条件满足时，就回调handler；不同点在于，异步情况下（Proactor)，当回调handler时，表示I/O操作已经完成；同步情况下（Reactor)，回调handler时，表示I/O设备可以进行某个操作（can read 或 can write)。

下面，我们将尝试应对为Proactor和Reactor模式建立可移植框架的挑战。在改进方案中，我们将Reactor原来位于事件处理器内的Read/Write操作移至分发器（不妨将这个思路称为“模拟异步”），以此寻求将Reactor多路同步I/O转化为模拟异步I/O。以读操作为例子，改进过程如下：

|  |
| --- |
| 注册读就绪事件和相应的事件处理器。并为分发器提供数据缓冲区地址，需要读取数据量等信息。  分发器等待事件（如在select()上等待）。  事件到来，激活分发器。分发器执行一个非阻塞读操作（它有完成这个操作所需的全部信息），最后调用对应处理器。  事件处理器处理用户自定义缓冲区的数据，注册新的事件（当然同样要给出数据缓冲区地址，需要读取的数据量等信息），最后将控制权返还分发器。  如我们所见，通过对多路I/O模式功能结构的改造，可将Reactor转化为Proactor模式。改造前后，模型实际完成的工作量没有增加，只不过参与者间对工作职责稍加调换。没有工作量的改变，自然不会造成性能的削弱。对如下各步骤的比较，可以证明工作量的恒定： |

**1.6.3.4 标准/典型的Reactor：**

|  |
| --- |
| 步骤1：等待事件到来（Reactor负责）。  步骤2：将读就绪事件分发给用户定义的处理器（Reactor负责）。  步骤3：读数据（用户处理器负责）。  步骤4：处理数据（用户处理器负责）。 |

**1.6.3.5 改进实现的模拟Proactor：**

|  |
| --- |
| 步骤1：等待事件到来（Proactor负责）。  步骤2：得到读就绪事件，执行读数据（现在由Proactor负责）。  步骤3：将读完成事件分发给用户处理器（Proactor负责）。  步骤4：处理数据（用户处理器负责）。 |

对于不提供异步I/O API的操作系统来说，这种办法可以隐藏Socket API的交互细节，从而对外暴露一个完整的异步接口。借此，我们就可以进一步构建完全可移植的，平台无关的，有通用对外接口的解决方案。

代码示例如下：

|  |
| --- |
| interface ChannelHandler{  void channelReadComplate(Channel channel，byte[] data);  void channelWritable(Channel channel);  }  class Channel{  Socket socket;  Event event;//读，写或者连接  }  //IO线程主循环：  class IoThread extends Thread{  public void run(){  Channel channel;  while(channel=Selector.select()){//选择就绪的事件和对应的连接  if(channel.event==accept){  registerNewChannelHandler(channel);//如果是新连接，则注册一个新的读写处理器  Selector.interested(read);  }  if(channel.event==write){  getChannelHandler(channel).channelWritable(channel);//如果可以写，则执行写事件  }  if(channel.event==read){  byte[] data = channel.read();  if(channel.read()==0)//没有读到数据，表示本次数据读完了  {  getChannelHandler(channel).channelReadComplate(channel，data;//处理读完成事件  }  if(过载保护){  Selector.interested(read);  }  }  }  }  Map<Channel，ChannelHandler> handlerMap;//所有channel的对应事件处理器  } |

**1.6.3.6 Selector.wakeup()**

* 主要作用

解除阻塞在Selector.select()/select(long)上的线程，立即返回。

两次成功的select之间多次调用wakeup等价于一次调用。

如果当前没有阻塞在select上，则本次wakeup调用将作用于下一次select——“记忆”作用。

* 为什么要唤醒？

注册了新的channel或者事件。

channel关闭，取消注册。

优先级更高的事件触发（如定时器事件），希望及时处理。

* 原理

Linux上利用pipe调用创建一个管道，Windows上则是一个loopback的tcp连接。这是因为win32的管道无法加入select的fd set，将管道或者TCP连接加入select fd set。

wakeup往管道或者连接写入一个字节，阻塞的select因为有I/O事件就绪，立即返回。可见，wakeup的调用开销不可忽视。

Buffer的选择

通常情况下，操作系统的一次写操作分为两步：

将数据从用户空间拷贝到系统空间。

从系统空间往网卡写。同理，读操作也分为两步：

① 将数据从网卡拷贝到系统空间；

② 将数据从系统空间拷贝到用户空间。

对于NIO来说，缓存的使用可以使用DirectByteBuffer和HeapByteBuffer。如果使用了DirectByteBuffer，一般来说可以减少一次系统空间到用户空间的拷贝。但Buffer创建和销毁的成本更高，更不宜维护，通常会用内存池来提高性能。

如果数据量比较小的中小应用情况下，可以考虑使用heapBuffer；反之可以用directBuffer。

* NIO存在的问题

用NIO != 高性能，当连接数<1000，并发程度不高或局域网环境下NIO并没显著性能优势。

NIO并没完全屏蔽平台差异，仍是基于各操作系统的I/O系统实现的，差异仍存在。用NIO做网络编程构建事件驱动模型不容易，陷阱重重。

推荐用成熟的NIO框架，如Netty，MINA等。解决了很多NIO陷阱，并屏蔽了操作系统差异，有较好性能和编程模型。

**1.6.4 总结**

NIO给带来了些什么：

|  |
| --- |
| 事件驱动模型  避免多线程  单线程处理多任务  非阻塞I/O，I/O读写不再阻塞，而是返回0  基于block的传输，通常比基于流的传输更高效  更高级的IO函数，zero-copy  IO多路复用大大提高了Java网络应用的可伸缩性和实用性 |

* ~~ConcurrentHashMap实现原理（锁分离技术）。~~
* String和StringBuffer,StringBuilder区别和联系，String为啥不可变，在内存中的具体形态。

~~String：字符串常量，字符串长度不可变。~~

~~StringBuffer：字符串变量（Synchronized，即线程安全）。如果要频繁对字符串内容进行修改，出于效率考虑最好使用StringBuffer，如果想转成String类型，可以调用StringBuffer的toString()方法。~~

~~StringBuilder：字符串变量（非线程安全）。在内部，StringBuilder对象被当作是一个包含字符序列的变长数组。~~

~~StringBuilder与StringBuffer有公共父类AbstractStringBuilder(抽象类)。~~

~~2.java中多线程机制，实现多线程的两种方式（继承Thread类和实现Runnable接口）的区别和联系。~~

~~3.java线程阻塞调用wait函数和sleep区别和联系，还有函数yield，notify等的作用。~~

~~wait是Object的方法，sleep是Thread类的方法；~~

~~wait让出CPU资源的同时会放弃锁，sleep让出CPU资源的同时不会释放锁；~~

~~wait需要notify或者notifyall来唤醒，sleep在沉睡指定时间后，会自动进入就绪状态~~；

~~4.java中的同步机制，synchronized关键字，锁（重入锁）机制，其他解决同步的方volatile关键字ThreadLocal类的实现原理要懂。~~

~~5.java中异常机制~~

~~Throwable是Error和Exception的父类，Error一般是指JVM抛出的错误，不需要捕获，Exception是程序错误，需要捕获处理；~~

8.在栈上为什么不能用变量做数组的长度，堆上可以吗？

9.问了我的项目，QQ聊天系统，怎么实现的，客户端为什么要用TCP和UDP结合，用UDP协议有什么好处，消息是怎样定义的，怎样区分不同的 消息，怎么知道使用锁的。我给他说了epoll、线程池，Reactor模式，以及自己实现的哈希表，线程之间怎样同步等

---------------------------------------------tenxun-----------------------------------------------------

1.【补】TCP/UDP区别，TCP三次握手，SYN攻击

TCP是面向连接的可靠传输，需要三次握手，保证可靠通信；有重传机制；

UDP是无连接的不可靠传输，但是速度快，适用于视频和电话会议等实时应用场景；

TCP三次握手是：SYN=x（SYN\_SEND）、ACK=x+1,SYN=y（SYN\_RECV）、ACK=y+1（ESTABLISHED）；

SYN攻击是：SYN攻击属于DOS攻击的一种，它利用TCP协议缺陷，通过发送大量的半连接请求，耗费CPU和内存资源。

检测SYN攻击非常的方便，当你在服务器上看到大量的半连接状态时，特别是源IP地址是随机的，基本上可以断定这是一次SYN攻击。

一类是通过防火墙、路由器等过滤网关防护，另一类是通过加固TCP/IP协议栈防范。过滤网关防护主要包括超时设置，SYN网关和SYN代理三种。调整tcp/ip协议栈，修改tcp协议实现。主要方法有SynAttackProtect保护机制、SYN cookies技术、增加最大半连接和缩短超时时间等。

但一般服务器所能承受的连接数量比半连接数量大得多

【补】SHA，MD5

|  |
| --- |
| SHA（安全哈希算法）：该算法的思想是接收一段明文，然后以一种不可逆的方式将它转换成一段（通常更小）密文，也可以简单的理解为取一串输入码（称为预映射或信息），并把它们转化为长度较短、位数固定的输出序列即散列值（也称为信息摘要或信息认证代码）的过程。可以对任意长度的数据运算生成一个160位的数值。SHA将输入流按照每块512位（64个字节）进行分块，并产生20个字节的被称为信息认证代码或信息摘要的输出。SHA-1是不可逆的、防冲突，并具有良好的雪崩效应。  MD5（信息-摘要算法5）：MD5将任意长度的“字节串”映射为一个128bit的大整数。MD5以512位分组来处理输入的信息，且每一分组又被划分为16个32位子分组，经过了一系列的处理后，算法的输出由四个32位分组组成，将这四个32位分组级联后将生成一个128位散列值。  原理参见：http://baike.baidu.com/link?url=ZAyVdzKs-4Oj7e6VQFY4zbzSw-OhOfCyRGlneCm4f4J8fUuTCPomQyIX7kZwfbFA3Dwmry70s2RsSBiVvU\_m9\_  两者比较：  （1）对强行攻击的安全性：最显著和最重要的区别是SHA-1摘要比MD5摘要长32 位。使用强行技术，产生任何一个报文使其摘要等于给定报摘要的难度对MD5是2^128数量级的操作，而对SHA-1则是2^160数量级的操作。这样，SHA-1对强行攻击有更大的强度。  （2）对密码分析的安全性：由于MD5的设计，易受密码分析的攻击，SHA-1显得不易受这样的攻击。  （3）速度：在相同的硬件上，SHA-1的运行速度比MD5慢。 |

3.【补】网络攻击手段，怎么预防

拒绝服务攻击：

（1）SYN拒绝服务攻击：目标计算机如果接收到大量的TCP SYN报文，而没有收到发起者的第三次ACK回应，会一直等待，处于这样尴尬状态的半连接如果很多，则会把目标计算机的资源（TCB控制结构，TCB，一般情况下是有限的）耗尽，而不能响应正常的TCP连接请求。

（2）ICMP洪水：。这样如果攻击者向目标计算机发送大量的ICMP ECHO报文（产生ICMP洪水），则目标计算机会忙于处理这些ECHO报文，而无法继续处理其它的网络数据报文，这也是一种拒绝服务攻击（DOS）。

（3）UDP洪水；

（4）死亡之PING：TCP/IP规范要求IP报文的长度在一定范围内（比如，0－64K），但有的攻击计算机可能向目标计算机发出大于64K长度的PING报文，导致目标计算机IP协议栈崩溃。

（5）IP地址欺骗：如果一个攻击者向一台目标计算机发出一个报文，而把报文的源地址填写为第三方的一个IP地址，这样这个报文在到达目标计算机后，目标计算机便可能向毫无知觉的第三方计算机回应。这便是所谓的IP地址欺骗攻击。

**1.7 海量数据处理**

1.7.1 10亿条短信，找出前一万条重复率高的

|  |
| --- |
| (1)将文本导入数据库，再用select语句某些方法得出前10条短信。（索引）但时间效率很低；  (2)用hash计算并存储次数，遍历一次找出top10； |

1.7.2 一万条数据排序，最好方式?

申请长度为一千万位的位向量bit[10000000]，所有位设为0

顺序读取待排序文件，每读入一个数i，便将bit[i]置为1。

所有数据读入完成，对bit做顺序遍历，如bit[i]=1，则输出i到文件

当遍历完，文件则已排好序。本机运行耗时9秒49毫秒。

（要求数组中的数据没重复，且都不超过最大值）

1.7.3 10w行数据，每行一单词，统计出现次数出现最多的前100个。

|  |
| --- |
| (1)可用小根堆；  (2)linux中：cat words.txt | sort | uniq -c | sort -k1,1nr | head -10   * uniq -c: 显示唯一行，并在每行行首加上本行在文件中出现的次数 * sort -k1,1nr: 按照第一个字段，数值排序，且为逆序 |

~~1.7.4 一个文本文件，给你一个单词，判断单词是否出现。~~

~~grep -wq "fail" 123.txt && echo "no"||echo "yes"~~

~~新手推荐看~~

~~http://www.tldp.org/LDP/Bash-Beginners-Guide/Bash-Beginners-Guide.pdf~~

~~进阶推荐看~~

~~http://www.tldp.org/LDP/abs/abs-guide.pdf~~

------------------------------------微店------------------------------------

【补】servlet和filter的区别。filter你在哪些地方用到过。

servlet是一种运行服务器端的java应用程序，具有独立于平台和协议的特性，并且可以动态的生成web页面，它工作在客户端请求与服务器响应的中间层。

1) 客户端发送请求至服务器端；

2) 服务器将请求信息发送至 Servlet；

3) Servlet 生成响应内容并将其传给服务器。响应内容动态生成，通常取决于客户端的请求；

4) 服务器将响应返回给客户端。

在 Web 应用程序中，一个 Servlet 在一个时刻可能被多个用户同时访问。这时 Web 容器将为每个用户创建一个线程来执行 Servlet。

filter是一个可以复用的代码片段，可以用来转换HTTP请求、响应和头信息。Filter不像Servlet，它不能产生一个请求或者响应，它只是修改对某一资源的请求，或者修改从某一的响应。只要你在web.xml文件配置好要拦截的客户端请求，它都会帮你拦截到请求，此时你就可以对请求或响应(Request、Response)统一设置编码，简化操作；同时还可进行逻辑判断，如用户是否已经登陆、有没有权限访问该页面等等工作。

Filter对用户请求进行预处理，接着将请求交给Servlet进行处理并生成响应，最后Filter再对服务器响应进行后处理。

创建一个Filter只需两个步骤：

建Filter处理类；

web.xml文件中配置Filter。

一定要实现javax.servlet包的Filter接口的三个方法init()、doFilter()、destroy().

listener：监听器，从字面上可以看出listener主要用来监听只用。通过listener可以监听web服务器中某一个执行动作，并根据其要求作出相应的响应。

interceptor：是在面向切面编程的，就是在你的service或者一个方法，前调用一个方法，或者在方法后调用一个方法，是基于JAVA的反射机制。比如动态代理就是拦截器的简单实现

【补】堆与普通二叉树有什么区别。

堆是结点之间满足一定次序关系的完全二叉树（大多数情况下使用的是完全二叉树，但不一定必须是完全二叉树）。

具有n个结点的堆，其深度即为堆所对应的完全二叉树的深度log n 。

以小根堆为例，堆的特点是双亲结点的关键字必然小于等于孩子结点的关键字，而两个孩子结点的关键字没有次序规定。

堆存在的原因：

我们需要能得到一个集合的最小值，并且，在它被任意划分成为若干个子集的时候，这些子集的最小值我们也是知道的，这些子集不断的被划分，我们依然知道再次被划分出来的这些子集的最小值。

堆不止二叉堆，还有更复杂的二项堆，斐波那契堆，配对堆等等。

总之，堆是一种特殊的树。

【补】 linux如何查找文件

cat、vi、

find ：find <指定目录> <指定条件> <指定动作>

使用locate搜索linux系统中的文件，它比find命令快。因为它查询的是数据库(/var/lib/locatedb)，数据库包含本地所有的 文件信息。使用locate命令在根目录下搜索interfaces文件的命令为”locate interfaces“

使用”whereis“命令可以搜索linux系统中的所有可执行文件即二进制文件。使用whereis命令搜索grep二进制文件的命令为”whereis grep“。

--------------------------------------------------去哪儿--------------------------------------------

3.【F】类加载机制

* 启动类加载器( Bootstrap ClassLoader)启动类加载器无法被 java 程序员直接引用, 这个类加载器负责把存放在<JAVA\_HOME>\lib目录中的,或者被-Xbootclasspath参数指定路径中的, 并且是被虚拟机识别的类库加载到虚拟机内存中.
* 扩展类加载器(Extension ClassLoader)负责加载在<JAVA\_HOME>\lib\ext目录中的, 或者被java.ext.dirs系统变量所指定的路径中的所有类库。
* 应用程序类加载器( Application ClassLoader )这个类加载器是ClassLoader 中的 getSystemClassLoader()方法的返回值,一般称其为系统类加载器, 它负责加载用户类路径( ClassPath )上所指定的类库

从 java 虚拟机的角度而降, 只存在两种不同的类加载器：

一个是启动类加载器( Bootstrap ClassLoader ), 这个类加载使用 C++ 语言实现, 是虚拟机自身的一部分;

另一种是其他所有的类加载器, 他们由 java 语言实现, 独立于虚拟机之外, 并且全部继承自java.lang.ClassLoader

加载类的寻找范围就是 JVM 默认路径加上Classpath,

类具体是使用哪个类加载器不确定。

4. 【F】类加载主要步骤

加载 把 class 文件的二进制字节流加载到 jvm 里面

验证 确保 class 文件的字节流包含的信息符合当前 jvm 的要求 有文件格式验证, 元数据验证, 字节码验证, 符号引用验证等

准备 正式为类变量分配内存并设置类变量初始值的阶段, 初始化为各数据类型的零值

解析 把常量值内的符号引用替换为直接引用的过程

初始化 执行类构造器<clinit>()方法

使用 根据相应的业务逻辑代码使用该类

卸载 类从方法区移除

5. 【F】双亲委派模型：

除了顶层的启动类加载器之外, 其余的类加载器都应当有自己的父类加载器, 父子关系这儿一般都是以组合来实现。

工作过程: 如果一个类加载器收到了类加载的请求, 它首先不会自己去尝试加载这个类, 而是把这个请求委派给父类加载器去完成, 最终所有的加载请求都会传送到顶层的启动类加载器中, 只有当父类加载器反馈自己无法完成这个请求时候, 才由子加载器来加载。

例如类Object，它放在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。

对于任何一个类, 都需要由加载它的类加载器和这个类本身一同确定其在 java 虚拟机中的唯一性。

ClassLoader.loadClass()的代码如下，先检查是否已经被加载过，如果没有则parent.loadClass()调用父加载器的loadClass()方法，如果父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载器加载失败，抛出ClassNotFoundException，再调用自己的findClass()方法进行加载。

另外，如自己实现类加载器，一般是Override复写 findClass方法，而不是loadClass方法。

1.双亲委派模型

Java虚拟机类加载过程是把Class类文件加载到内存，并对Class文件中的数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的java类型的过程。

在加载阶段，java虚拟机需要完成以下3件事：

a.通过一个类的全限定名来获取定义此类的二进制字节流。

b.将定义类的二进制字节流所代表的静态存储结构转换为方法区的运行时数据结构。

c.在java堆中生成一个代表该类的java.lang.Class对象，作为方法区数据的访问入口。

(1).BootStrap ClassLoader：启动类加载器，负责加载存放在%JAVA\_HOME%\lib目录中的，或者通被-Xbootclasspath参数所指定的路 径中的，并且被java虚拟机识别的(仅按照文件名识别，如rt.jar，名字不符合的类库，即使放在指定路径中也不会被加载)类库到虚拟机的内存中，启 动类加载器无法被java程序直接引用。

(2).Extension ClassLoader：扩展类加载器，由sun.misc.Launcher$ExtClassLoader实现，负责加 载%JAVA\_HOME%\lib\ext目录中的，或者被java.ext.dirs系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载 器。

(3).Application ClassLoader：应用程序类加载器，由sun.misc.Launcher$AppClassLoader实现，负责加载用户类路径 classpath上所指定的类库，是类加载器ClassLoader中的getSystemClassLoader()方法的返回值，开发者可以直接使 用应用程序类加载器，如果程序中没有自定义过类加载器，该加载器就是程序中默认的类加载器。

java虚拟机规范推荐开发者使用双亲委派模式(ParentsDelegation Model)进行类加载，其加载过程如下：

(1).如果一个类加载器收到了类加载请求，它首先不会自己去尝试加载这个类，而是把类加载请求委派给父类加载器去完成。

(2).每一层的类加载器都把类加载请求委派给父类加载器，直到所有的类加载请求都应该传递给顶层的启动类加载器。

(3).如果顶层的启动类加载器无法完成加载请求，子类加载器尝试去加载，如果连最初发起类加载请求的类加载器也无法完成加载请求时，将会抛出ClassNotFoundException，而不再调用其子类加载器去进行类加载。

具体参看《深入理解Java虚拟机》第七章 虚拟机类加载机制。

2.单源最短路算法简述

Dijkstra算法是解单源最短路径问题的贪心算法。

算法流程：

(a) 初始化：用起点v到该顶点w的直接边(弧)初始化最短路径，否则设为∞；

(b) 从未求得最短路径的终点中选择路径长度最小的终点u：即求得v到u的最短路径；

(c) 修改最短路径：计算u的邻接点的最短路径，若(v,…,u)+(u,w)<(v,…,w)，则以(v,…,u,w)代替。

(d) 重复(b)-(c)，直到求得v到其余所有顶点的最短路径。

算法基于：如果存在一条从i到j的最短路径(Vi.....Vk,Vj)，Vk是Vj前面的一顶点。那么(Vi...Vk)也必定是从i到k的最短路径。

dist[j]=min{dist[j],dist[i]+matrix[i][j]}

单源最短路除了dijkstra算法之外，还有一种常用的算法叫做SPFA（shortest path faster algorithm）算法，不同于dijkstra的复杂度为o(n^2)，SPFA算法的平均复杂度为o(kE)，E为边数，且k通常不超过2。用bfs实现。

SPFA的核心思想如下：

最开始起点入队，然后考虑和起点相邻的点，更新dis数组，并将这些点入队；

当队列不为空时，每次取队首一个点，对这个点相邻的点进行松弛操作，即比较原先的dis和经过新加入的点的优化后的dis，如果松弛成功，且被松弛的点不在队列中，则将其加入队列，重复上述动作。

SPFA可以用来判断负环，我们开一个cnt数组记录每个点入队的次数，如果次数超过n说明出现负环。

3..有若干个点,给出哪些点是相连的,求第一个点到最后一个点是否联通,并求出经过几个点(实际还是最短路问题)

4.Mybatis和hibernate有什么区别

Hibernate和Mybatis都是orm对象关系映射框架，都是用于将数据持久化的框架技术。

Hiberante较深度的封装了jdbc，对开发者写sql的能力要求的不是那么的高，我们只要通过hql语句操作对象即可完成对数据持久化的操作了。

另外hibernate可移植性好，如一个项目开始使用的是mysql数据库，但是随着业务的发展，现mysql数据库已经无法满足当前的绣球了，现在决定使用Oracle数据库，虽然sql标准定义的数据库间的sql语句差距不大，但是不同的数据库sql标准还是有差距的，那么我们手动修改起来会存在很大的困难，使用hibernate只需改变一下数据库方言即可搞定。用hibernate框架，数据库的移植变的非常方便。

但是hibernate也存在着诸多的不足，比如在实际开发过程中会生成很多不必要的sql语句耗费程序资源，优化起来也不是很方便，且对存储过程支持的也不够强大。但是针对于hibernate它也提供了一些优化策略，比如说懒加载、缓存、策略模式等都是针对于它的优化方案。

Mybatis 也是对jdbc的封装，但是封装的没有hibernate那么深，我们可以再配置文件中写sql语句，可以根据需求定制sql语句，数据优化起来较hibernate容易很多。

Mybatis要求程序员写sql的能力要相对使用hibernate的开发人员要高的多，且可移植性也不是很好。

涉及到大数据的系统使用Mybatis比较好，因为优化较方便。涉及的数据量不是很大且对优化没有那么高，可以使用hibernate。

两者相同

Hibernate与MyBatis都可以是通过SessionFactoryBuider由XML配置文件生成SessionFactory，然后由SessionFactory 生成Session，最后由Session来开启执行事务和SQL语句。其中SessionFactoryBuider，SessionFactory，Session的生命周期都是差不多的。

Hibernate和MyBatis都支持JDBC和JTA事务处理。

Mybatis优势

MyBatis可以进行更为细致的SQL优化，可以减少查询字段。

MyBatis容易掌握，而Hibernate门槛较高。

Hibernate优势

Hibernate的DAO层开发比MyBatis简单，Mybatis需要维护SQL和结果映射。

Hibernate对对象的维护和缓存要比MyBatis好，对增删改查的对象的维护要方便。

Hibernate数据库移植性很好，MyBatis的数据库移植性不好，不同的数据库需要写不同SQL。

Hibernate有更好的二级缓存机制，可以使用第三方缓存。MyBatis本身提供的缓存机制不佳。

Hibernate功能强大，数据库无关性好，O/R映射能力强，如果你对Hibernate相当精通，而且对Hibernate进行了适当的封装，那么你的项目整个持久层代码会相当简单，需要写的代码很少，开发速度很快，非常爽。

Hibernate的缺点就是学习门槛不低，要精通门槛更高，而且怎么设计O/R映射，在性能和对象模型之间如何权衡取得平衡，以及怎样用好Hibernate方面需要你的经验和能力都很强才行。

iBATIS入门简单，即学即用，提供了数据库查询的自动对象绑定功能，而且延续了很好的SQL使用经验，对于没有那么高的对象模型要求的项目来说，相当完美。

iBATIS的缺点就是框架还是比较简陋，功能尚有缺失，虽然简化了数据绑定代码，但是整个底层数据库查询实际还是要自己写的，工作量也比较大，而且不太容易适应快速数据库修改。

针对高级查询，Mybatis需要手动编写SQL语句，以及ResultMap。而Hibernate有良好的映射机制，开发者无需关心SQL的生成与结果映射，可以更专注于业务流程。

5.数据库索引优点和缺点？sql优化方案（见数据库总结）

6.滑动窗口算法

7.简单介绍下线程池的参数？你了解哪些线程池

（参看7.线程池的配置）

8.简述JVM,从类加载一直到运行时的内存区域

<http://blog.csdn.net/d12345678a/article/details/53956334>

10.说下CMS收集器,说下G1收集器（之前有总结了）

----------------------------------------------------mogujie---------------------------------------

1.Mysql和mongodb索引原理（说一下B+tree和B-tree区别，为什么一个用b+一个用b-，为什么索引用btree不用平衡二叉树）mysql两种存储引擎

B-树和B+树最重要的一个区别就是B+树只有叶节点存放数据，其余节点用来索引，而B-树是每个索引节点都会有Data域。

这就决定了B+树更适合用来存储外部数据，也就是所谓的磁盘数据。

从Mysql（Inoodb）的角度来看，B+树是用来充当索引的，一般来说索引非常大，尤其是关系性数据库这种数据量大的索引能达到亿级别，所以为了减少内存的占用，索引也会被存储在磁盘上。

那么Mysql如何衡量查询效率呢？磁盘IO次数，B-树（B类树）的特定就是每层节点数目非常多，层数很少，目的就是为了就少磁盘IO次数，当查询数据的时候，最好的情况就是很快找到目标索引，然后读取数据，使用B+树就能很好的完成这个目的，但是B-树的每个节点都有data域（指针），这无疑增大了节点大小，说白了增加了磁盘IO次数（磁盘IO一次读出的数据量大小是固定的，单个数据变大，每次读出的就少，IO次数增多，一次IO多耗时啊！），而B+树除了叶子节点其它节点并不存储数据，节点小，磁盘IO次数就少。这是优点之一。

另一个优点是什么，B+树所有的Data域在叶子节点，一般来说都会进行一个优化，就是将所有的叶子节点用指针串起来。这样遍历叶子节点就能获得全部数据，这样就能进行区间访问啦。

至于MongoDB为什么使用B-树而不是B+树，可以从它的设计角度来考虑，它并不是传统的关系性数据库，而是以Json格式作为存储的nosql，目的就是高性能，高可用，易扩展。首先它摆脱了关系模型，上面所述的优点2需求就没那么强烈了，其次Mysql由于使用B+树，数据都在叶节点上，每次查询都需要访问到叶节点，而MongoDB使用B-树，所有节点都有Data域，只要找到指定索引就可以进行访问，无疑单次查询平均快于Mysql（但侧面来看Mysql至少平均查询耗时差不多）。

总体来说，Mysql选用B+树和MongoDB选用B-树还是以自己的需求来选择的。

2.浮点型为什么不是精确值

十进制小数部分化二进制，常常化不尽。

如同无限循环小数，最后有截断误差。

二进制无法精确表示十进制的十分之一。

使用BigDecimal来做精确运算

1. 线程池原理，单核操作系统是否应该使用多线程，为什么

4.有1万个左右的脏词，每次发帖要判断帖子里边是否包括有脏词，数据库应该如何设计

可以建立一个脏词字典表，写好函数判断帖子正文里面是否有脏词。然后再这个帖子正文字段上面建立check 约束。

-招银---------------------------------------------

1.pubilc A｛ public void test(){} }

public B extends A{ protected void test(){} }

这样有问题吗？为什么？

错，

2.public A｛ public long test(){} }

public B extends A{ public int test(){} }

这样有问题吗？为什么？

错。子类重写父类方法时不能更改返回值类型。

3.多态

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）。

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

Java中多态的实现方式：接口实现，继承父类进行方法重写，同一个类中进行方法重载。

多态存在的三个必要条件

一、要有继承；

二、要有重写；

三、父类引用指向子类对象。

父类的引用指向了自己的子类对象。

父类的引用也可以接收自己的子类对象。

4.int i=0； Integer i1=0； Integer i2=new Integer（0）； 输出i==i1；i==i2；i1==i2

分别是true，true，false

5.输出用户表里重名的用户，说sql语句

select myname, count(\*) from tab group by myname having count(\*) > 1

7.说一下ajax

是指一种创建交互式网页应用的网页开发技术。通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

优点：因为是异步的，所以不会阻塞用户操作；

局部刷新，这样就可使网页从服务器请求少量的信息，而不是整个页；

8.就你项目中使用的 MyBatis，有哪些标签？

<configuration>，<typeAliases>，<plugins>，<mappers>，<resultMap>，

9.查询语句使用 MyBatis 框架中的 where 标签和不使用 MyBatis 框架的sql语句有啥不同？

where

<foreach item="item" index="index" collection="list"

separator="or">

( classify\_id = #{item.classify\_id}

and program\_id =

#{item.program\_id})

</foreach>

<where>

　　　　　　<if test="id != null ">id=#{id}</if>

10.new 实例化过程

寻找类定义 ；

加载类定义，jvm启动时，会用启动类加载器加载，对于用户的class，则会用应用程序类加载器实时加载，所谓实时加载，指的是遇到的时候再加载，而不是预先一次 性加载。关于类加载器，有三级，jvm严格的限制了每一级的加载权限，加载模式为“双亲委托模式”，加载任何类，都先由父加载器加载。

给对象分配内存空间；jvm在内存“堆”中，开辟一个空间，该空间按照MyObject类定义开辟，并将该空间中的各个内存段设置默认值，对应的就是对象的属性初始化默认值。

对象的初始化顺序 ；给父类静态变量默认值，对父类静态变量赋值，执行父类静态块；给当前类静态变量默认值 ，对当前类静态变量赋值，执行当前类静态块；给父类变量默认值，对父类变量赋值， 执行父类构造函数；给当前类变量默认值，对当前类变量赋值，执行当前类构造函数。

对象构造完成 ；

11.socket实现过程，具体用的方法；怎么实现异步socket.（不会）

Socket编程基本就是create，listen，accept，connect，read和write等等。在建立连接前，必须知道对方的IP地址和端口号。一个指定的端口号不能被多个程序共用

1：加载套接字库，创建套接字(WSAStartup()/socket())；

2：绑定套接字到一个IP地址和一个端口上(bind())；

3：将套接字设置为监听模式等待连接请求(listen())；

4：请求到来后，接受连接请求，返回一个新的对应于此次连接的套接字(accept())；

5：用返回的套接字和客户端进行通信(send()/recv())；

6：返回，等待另一连接请求；

7：关闭套接字，关闭加载的套接字库(closesocket()/WSACleanup())。

客户端编程的步骤：

1：加载套接字库，创建套接字(WSAStartup()/socket())；

2：向服务器发出连接请求(connect())；

3：和服务器端进行通信(send()/recv())；

4：关闭套接字，关闭加载的套接字库(closesocket()/WSACleanup())。

12.反射的所有包，怎实现反射

java中反射的三种方法：

1. 通过Object类的getClass方法来获取

2.使用.class的方式，使用类名加“.class”的方式即会返回与该类对应的Class对象。

3. 使用Class.forName方法，Class有一个著名的static方法forName

13.网络重定向，说下流程

重定向(Redirect)就是通过各种方法将各种网络请求重新定个方向转到其它位置。

流程：客户端浏览器访问一个URL，得到的响应状态是3xx的时候，会在响应信息中包含重定向的URL，客户端浏览器需要再向重定向的URL发送HTTP请求，并获得响应数据。

15. servlet是单例么？

是

16.多线程共享变量的问题？

17.Tomcat的结构

就按server.xml中的层次结构来答

Tomcat 体系结构中的六个主要概念：

Server：Server代表整个容器(Container)。它可以包含一个或多个Service，还可以包含一个GlobalNamingResources。Server支持className，port和shutdown三个公共属性。

Service：Service中可以含有一个或多个Connector，但只能含有一个Engine。这使得不同的Connector可以共享同一个Engine。同一个Server中的多个Service之间没有相关性。Service支持className和name两个公共属性。

Engine：Engine负责接收和处理来自它所属的Service中的所有Connector的请求。Engine支持backgroundProcessorDelay、className、defaultHost、jvmRoute和name五个公共属性。

Host：Host表示一个虚拟主机，并和一个服务器的网络名关联。注意Engine中必须有一个Host的名字和Engine的defaultHost属性匹配。Host支持appBase、autoDeploy、backgroundProcessorDelay、className、deployOnStartup和name六个公共属性。

Connector：Connector负责接收来自客户端(Client)的请求。比较常见的两个是HTTP Connector和AJP

Connector。

Context：Context表示在虚拟主机中运行的web应用程序。一个虚拟主机中能够运行多个Context，它们通过各自的Context Path进行相互区分。如果Context Path为""，那么该web应用为该虚拟主机的默认的web应用。Context支持backgroundProcessorDelay、className、cookies、crossContext、docBase、override、privileged、path、reloadable和wrapperClass十个公共属性。

19.继承的好处和坏处

优点：1.可以使用父类的所有非私有方法；而且单继承可由接口来弥补。

2.可以继承父类中定义的成员方法以及成员变量，使得子类可以减少代码的书写。还可以重写父类的方法以增加子类的功能。

缺点：1.耦合性太大

2.就是破坏了类的封装性，其实继承一般多用于抽象方法的继承和接口的实现

21.静态类与单例模式的区别

单例模式比静态类有很多优势：

（1）单例可以继承类，实现接口，而静态类不能（可以集成类，但不能集成实例成员）；

（2）单例可以被延迟初始化，静态类一般在第一次加载是初始化；

（3）单例类可以被集成，他的方法可以被覆写；

（4）或许最重要的是，单例类可以被用于多态而无需强迫用户只假定唯一的实例。举个例子，你可能在开始时只写一个配置，但是以后你可能需要支持超过一个配置集，或者可能需要允许用户从外部文件中加载一个配置对象，或者编写自己的。你的代码不需要关注全局的状态，因此你的代码会更加灵活。

观点二：（静态方法）静态方法中产生的对象，会随着静态方法执行完毕而释放掉，而且执行类中的静态方法时，不会实例化静态方法所在的类。如果是用singleton, 产生的那一个唯一的实例，会一直在内存中，不会被GC清除的(原因是静态的属性变量不会被GC清除)，除非整个JVM退出了。这个问题我之前也想几天，并 且自己写代码来做了个实验。

观点三：（Good！）

由于DAO的初始化，会比较占系统资源的，如果用静态方法来取，会不断地初始化和释放，所以我个人认为如果不存在比较复杂的事务管理，用singleton会比较好。

22.面向对象的设计规范

是五大基本原则吗？

—————————————————乐视--------------------------------------------

1.新生老年代gc算法的实现

新生代：复制；

老年代：标记整理

2.maven冲突如何解决；

build-helper-maven-plugin

4.DNS均衡

智能DNS可以通过多种负载均衡策略来将客户端需要访问的域名解析到不同的数据中心不同的线路上，比如通过IP地理信息数据库解析到最近的线路，或者权衡不同线路的繁忙度解析到空闲的线路等等。

将客户端的访问引导到不同的机器上去，使得不同的客户端访问不同的服务器，从而达到负载均衡的目的。

1.全局的运营商+区域层面的负载均衡，主要是就近调度原则；

2.机房或集群内部的负载均衡，主要实现流量均摊、合理利用资源等；

5.Linux(查看指定进程)

(1) 先用pgrep [str] 命令进行模糊匹配，找到匹配该特征串的进程ID；

(2) 其次根据进程ID显示指定的进程信息，ps --pid [pid]；

(3) 因为查找出来的进程ID需要被作为参数传递给ps命令，故使用xargs命令，通过管道符号连接；

(4) 最后显示进程详细信息，需要加上-u参数。

pgrep Java

| xargs ps -u –pid

6.IP地址分为几类，每类都代表什么，私网是哪些

A：1.0.0.0到127.255.255.255之间

B：128.0.0.0—191.255.255.255（局域网）

C：192.0.0.0至223.255.255.255（校园网）

D：224.0.0.0到239.255.255.255，也被称作多播地址或组播地址

私网地址：

A类：10.0.0.1--10.255.255.254

B类：172.16.0.0.1--172.31.255.254

C类：192.168.0.1--192.168.255.254

内网的计算机以NAT（网络地址转换）协议，通过一个公共的网关访问Internet。内网的计算机可向Internet上的其他计算机发送连接请求，但Internet上其他的计算机无法向内网的计算机发送连接请求。

7.Linux 查看有几个多大的内存条组成

#dmidecode | grep -A16 "Memory Device$"

# free -m （查看当前内存使用情况）

1、查看内存槽数、那个槽位插了内存，大小是多少

dmidecode|grep -P -A5 "Memory\s+Device"|grep Size|grep -v Range

2、查看最大支持内存数

dmidecode|grep -P 'Maximum\s+Capacity'

3、查看槽位上内存的速率，没插就是unknown。

dmidecode|grep -A16 "Memory Device"|grep 'Speed'

8.多线程的实现方式，以及意义

start()方法是一个native方法，它将启动一个新线程，并执行run()方法。

Java多线程实现方式主要有三种：继承Thread类、实现Runnable接口、使用ExecutorService、Callable、Future实现有返回结果的多线程。

1.Thread是Runnable接口的子类，实现Runnable接口的方式解决了Java单继承的局限

　　2.Runnable接口实现多线程比继承Thread类更加能描述数据共享的概念

继承Thread类

实现Runnable接口

继承接口更好，因为java的类是单继承的，如果继承了Thread类就不能再继承别的类了。

9.数据库事务，什么是事务，什么情况下会用到事务，举例说明

数据库事务(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

当需要保证一系列的操作都执行的时候会用到事务。比如在图书馆借书。

-----------------------------------------------滴滴------------------------------------------------

2.两个长度为n的排好序数组，计算中位数（总是让我证明我说的二分法为啥对，这就尴尬了，怎么证明）。一个排好序的 1,2,....n的数组，其中一个数变为-1，如何找到这个数，至少用三中方法。

求和公式可以求，位数组记录哪个数不存在可以求，排序也可以求

3.问了锁机制，问了浏览器如何实现的免登陆之类的功能，进而问了cookie和session相关的知识。

（1） 很多人说利用cookie保存加密后的密码和用户ID，当再次访问网站时服务器提取cookie的用户ID和密码，如果通过验证，则不需要重新输入账号和密码。

（2）1、在用户登录成功时，创建session对象，保存用户信息

2、将此session的sessionid保存到cookie中

3、同时将sessionid于session对应关系存储到应用域中，以便后面可以根据sessionid来获取到session

4、在用户关闭浏览器，重新打开浏览器访问网站时，读取用户的cookie，得到sessionid

5、根据sessionid获取到第3步存储到应用域中的session对象

6、从session中读取用户信息

------------------------------------------华为------------------------------------------------------

1.拓扑排序思想

拓扑排序是对有向无环图的顶点的一种排序，它使得如果存在一条从顶点A到顶点B的路径，那么在排序中B出现在A的后面。

（1）从有向图中选取一个没有前驱的顶点，并输出之；

（2）从有向图中删去此顶点以及所有以它为尾的弧；

重复上述两步，直至图空，或者图不空但找不到无前驱的顶点为止。没有前驱 -- 入度为零，删除顶点及以它为尾的弧-- 弧头顶点的入度减1。

2.操作系统IO的几种类型和原理

1）阻塞IO

2）非阻塞IO：轮询

3）IO复用（select和poll）

4）信号驱动IO（sigio）：当我们所监控的套接口有IO操作准备就绪时，由内核通知触发前面注册的信号处理程序执行

5）异步IO（aio\_）：异步IO直接是在第二个阶段完成后内核直接通知可以进程后续操作了

3.nginx事件处理模型

异步、事件循环；

nginx事件驱动：通信机制采用epoll模型，支持更大的并发连接。

通过异步非阻塞的事件处理机制，Nginx实现由进程循环处理多个准备好的事件，从而实现高并发和轻量级。

Nginx特点：

1. 跨平台：Nginx 可以在大多数 Unix like OS编译运行，而且也有Windows的移植版本。

2. 配置异常简单，非常容易上手。配置风格跟程序开发一样，神一般的配置

3. 非阻塞、高并发连接：数据复制时，磁盘I/O的第一阶段是非阻塞的。官方测试能够支撑5万并发连接，在实际生产环境中跑到2～3万并发连接数.(这得益于Nginx使用了最新的epoll模型)

4. 事件驱动：通信机制采用epoll模型，支持更大的并发连接。

5. nginx代理和后端web服务器间无需长连接；

6. 接收用户请求是异步的，即先将用户请求全部接收下来，再一次性发送后后端web服务器，极大的减轻后端web服务器的压力

7. 发送响应报文时，是边接收来自后端web服务器的数据，边发送给客户端的

8. 网络依赖型低。NGINX对网络的依赖程度非常低，理论上讲，只要能够ping通就可以实施负载均衡，而且可以有效区分内网和外网流量

9. 支持服务器检测。NGINX能够根据应用服务器处理页面返回的状态码、超时信息等检测服务器是否出现故障，并及时返回错误的请求重新提交到其它节点上

4.程序开发的流程，什么样的代码是比较好的代码？

需求分析->概要设计->详细设计（框架、模块）->编码（环境搭建、编码、调试、联调、集成）->测试->软件交付->验收->维护

功能实现良好、效率高、易读、易扩展和维护、简洁和明确；模块化，封装好；

7.http与https的区别

（1）HTTPS是HTTP的安全版，是HTTP+SSL，HTTP协议以明文方式发送内容，不提供任何方式的数据加密；

（2）https协议需要到ca申请证书，一般免费证书很少，需要交费。http是免费的；

（3）http和https使用的是完全不同的连接方式用的端口也不一样,前者是80,后者是443；

（4）http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

8.设计一个xml格式，将一个类序列化为xml

public static <T> T parserXML(String xml) {

ByteArrayInputStream in = new ByteArrayInputStream(xml.getBytes());

XMLDecoder decoder = new XMLDecoder(new BufferedInputStream(in));

decoder.close();

return (T) decoder.readObject();

}

public static <T> String formatXML(T entity) {

ByteArrayOutputStream out = new ByteArrayOutputStream();

XMLEncoder encoder = new XMLEncoder(new BufferedOutputStream(out));

encoder.writeObject(entity);

encoder.close();

return out.toString();

}

9.介绍非对称加密

非对称加密算法需要两个密钥：公开密钥（publickey）和私有密钥（privatekey）。公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；

1、乙方生成一对密钥（公钥和私钥）并将公钥向其它方公开。

2、得到该公钥的甲方使用该密钥对机密信息进行加密后再发送给乙方。

3、乙方再用自己保存的另一把专用密钥（私钥）对加密后的信息进行解密。乙方只能用其专用密钥（私钥）解密由对应的公钥加密后的信息。

在非对称加密中使用的主要算法有：RSA、Elgamal、背包算法、Rabin、D-H、ECC（椭圆曲线加密算法）等。

10.介绍三级缓存原理

内存 > 硬盘 > 网络

11.怎么保持长连接

一个连接是通过服务器地址和端口还有客户端地址和端口来识别唯一连接的。

Connection:keep-alive

（1）在应用层使用heartbeat来主动检测，在应用层制定协议，发心跳包，客户端和服务端制定一个通讯协议，每隔一定时间（一般15秒左右），由一方发起，向对方发送协议包；对方收到这个包后，按指定好的通讯协议回一个。若没收到回复，则判断网络出现问题，服务器可及时的断开连接，客户端也可以及时重连。

（2）通过TCP协议层发送KeepAlive包。这个方法只需设置好你使用的TCP的KeepAlive项就好，其他的操作系统会帮你完成。操作系统会按时发送KeepAlive包，一发现网络异常，马上断开。在应用层不需自己定协议，通信的两端，只要有一端设好这个值。还有一个好处就是节省网络资源。

ServerSocket serverSocket=new ServerSocket(1111);

Socket socket=serverSocket.accept();

while(true){//保持长连接

try {

Thread.sleep(100);//等待时间

} catch (InterruptedException e1) {

e1.printStackTrace();

}

if (socket !=null){

try {

String ip = socket.getInetAddress().toString().replace("/", "");

System.out.println("====socket.getInetAddress()====="+ip);

socket.setKeepAlive(true);

InputStream is = socket.getInputStream();

OutputStream os = socket.getOutputStream();

System.out.println("服务器端接受请求");

String tempdata = StreamEazyUse.getContent(is);

system.out.printl(“接收到的数据为：”+tempdata);

if(tempdata.contains(“stop”)){

is.close();

os.close();

}

os.flush();

}catch(Exception e){

system.out.println(“出现了错误”);

}

}

2.HashMap和HashSet的实现原理，hashset 和hashMap区别

HashSet底层就是HashMap实现的，

\*HashMap\* \*HashSet\*

HashMap实现了Map接口 HashSet实现了Set接口

HashMap储存键值对 HashSet仅仅存储对象

使用put()方法将元素放入map中 使用add()方法将元素放入set中

HashMap中使用键对象来计算hashcode值 HashSet使用成员对象来计算hashcode值，对于两个对象来说hashcode可能相同，所以equals()方法用来判断对象的相等性，如果两个对象不同的话，那么返回false

HashMap比较快，因为是使用唯一的键来获取对象 HashSet较HashMap来说比较慢

3.ConcurrentHashMap

HashTable是线程安全的，是给每个方法加synchronized；

4.同步方法。

（1）synchronized关键字修饰的语句块，synchronized只能标记非抽象的方法，不能标识成员变量。synchronized关键字也可以修饰静态方法，此时如果调用该静态方法，将会锁住整个类。

（2）使用特殊域变量(Volatile)实现线程同步

a.volatile关键字为域变量的访问提供了一种免锁机制

b.使用volatile修饰域相当于告诉虚拟机该域可能会被其他线程更新

c.因此每次使用该域就要重新计算，而不是使用寄存器中的值

d.volatile不会提供任何原子操作，它也不能用来修饰final类型的变量

（3）ReentrantLock类是可重入、互斥、实现了Lock接口的锁；

（4）使用ThreadLocal管理变量，则每一个使用该变量的线程都获得该变量的副本，副本之间相互独立，这样每一个线程都可以随意修改自己的变 量副本，而不会对其他线程产生影；

5.很常见的 Nullpointerexception ，你是怎么排查的，怎么解决的；

赋初值。

6.Out of MemoryError 产生的原因是什么，具体怎么去调优，以及理解那几个参数的含义 -Xms， -Xmx ，-Xmn， -XX:PermSize

内存泄露是指程序在运行过程中动态申请的内存空间不再使用后没有及时释放，从而很可能导致应用程序内存无线增长。

内存溢出即用户在对其数据缓冲区操作时，超过了其缓冲区的边界；尤其是对缓冲区写操作时，缓冲区的溢出很可能导致程序的异常。

原因：

1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；

2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；

3.代码中存在死循环或循环产生过多重复的对象实体；

4.启动参数内存值设定的过小；

调优：

1)检查代码中是否有死循环或递归调用。

2)检查是否有大循环重复产生新对象实体。

3)检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出

4 )检查List、MAP等集合对象是否有使用完后，未清除的问题

5）应用服务器提示错误的解决： 把启动参数内存值设置足够大

-Xms， -Xmx 最小，最大堆大小；-Xmn新生代大小；-XX:PermSize设置永久代大小；

7.线程池的配置

Excutor 以及Connector的配置

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler)

corePoolSize，线程池里最小线程数

maximumPoolSize，线程池里最大线程数量，超过最大线程时候会使用RejectedExecutionHandler

keepAliveTime，unit，线程最大的存活时间

workerQueue，缓存异步任务的队列

threadFactory，用来构造线程池里的worker线程

8.负载均衡如何实现Seesion共享

（1）我们可以把用户访问页面产生的session放到cookie里面，就是以cookie为中转站。

（2）会话保持：Session保持是我们见到最多的名词之一，通过会话保持，负载均衡进行请求分发的时候保证每个客户端固定的访问到后端的同一台应用服务器。会话保持方案在所有的负载均衡都有对应的实现。而且这是在负载均衡这一层就可以解决Session问题。

（3）会话复制：将每个应用服务器中的Session信息复制到其它服务器节点上。（不可取）

（4）会话共享：对于Session来说，肯定是频繁使用的，虽然你可以把它存放在数据库中，但是真正生产环境中我更推荐存放在性能更快的分布式KV数据中，例如：Memcached和Redis。

10.HashMap如果有很多相同key，后面的链很长的话，你会怎么优化？或者你会用什么数据结构来存储？我说了一个SkipList

如果是hash冲突，可以通过（1）设置大一些的table值；（2）改进hashCode算法；

Skip List是一种随机化的数据结构（必须有序），基于并联的链表，其效率可比拟于二叉查找树（对于大多数操作需要O(log n)平均时间）。基本上，跳跃列表是对有序的链表增加上附加的前进链接，增加是以随机化的方式进行的，所以在列表中的查找可以快速的跳过部分列表(因此得 名)。所有操作都以对数随机化的时间进行。Skip List可以很好解决有序链表查找特定值的困难。

再向跳跃表中插入新的结点时候，我们需要生成该结点的层数，使用的就是随机数生成器，随机的生成一个层数。

11.谈一谈对volatile理解，这个问题很常见，答出要点：

可见性、防止指令重排即可。

在并发编程中，我们通常会遇到以下三个问题：原子性问题，可见性问题，有序性问题。

只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。

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）volatile也无法保证对变量的任何操作都是原子性的。

原理：

“观察加入volatile关键字和没有加入volatile关键字时所生成的汇编代码发现，加入volatile关键字时，会多出一个lock前缀指令”

　　lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供3个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他CPU中对应的缓存行无效。

12.讲一下Spring的AOP和IOC理解与实际使用

spring 的优点？

1.降低了组件之间的耦合性 ，实现了软件各层之间的解耦

2.可以使用容易提供的众多服务，如事务管理，消息服务等

3.容器提供单例模式支持

4.容器提供了AOP技术，利用它很容易实现如权限拦截，运行期监控等功能

5.容器提供了众多的辅助类，能加快应用的开发

6.spring对于主流的应用框架提供了集成支持，如hibernate，JPA，Struts等

7.spring属于低侵入式设计，代码的污染极低

8.独立于各种应用服务器

9.spring的DI机制降低了业务对象替换的复杂性

10.Spring的高度开放性，并不强制应用完全依赖于Spring，开发者可以自由选择spring的部分或全部

* 什么是DI机制？

依赖注入（Dependecy Injection）和控制反转（Inversion of Control）是同一个概念，具体的讲：当某个角色需要另外一个角色协助的时候，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但在spring中创建被调用者的工作不再由调用者来完成，因此称为控制反转。创建被调用者的工作由spring来完成，然后注入调用者因此也称为依赖注入。

spring以动态灵活的方式来管理对象 ， 注入的两种方式，设置注入和构造注入。

设置注入的优点：直观，自然

构造注入的优点：可以在构造器中决定依赖关系的顺序。

* 什么是AOP？

面向切面编程（AOP）完善spring的依赖注入（DI），面向切面编程在spring中主要表现为两个方面

1.面向切面编程提供声明式事务管理

2.spring支持用户自定义的切面

面向切面编程（aop）是对面向对象编程（oop）的补充，

面向对象编程将程序分解成各个层次的对象，面向切面编程将程序运行过程分解成各个切面。

AOP从程序运行角度考虑程序的结构，提取业务处理过程的切面，oop是静态的抽象，aop是动态的抽象，是对应用执行过程中的步骤进行抽象，，从而获得步骤之间的逻辑划分。

aop框架具有的两个特征：

1.各个步骤之间的良好隔离性

2.源代码无关性

15.什么是递归，递归的几个条件？写递归要注意些什么？

程序调用自身的编程技巧称为递归。

两个条件：可以通过递归调用来缩小问题规模，且新问题与原问题有着相同的形式。

存在一种简单情境，可以使递归在简单情境下退出。

19.对后台的优化有了解吗？比如负载均衡。我给面试官说了 Ngix+Tomcat负载均衡，异步处理（消息缓冲服务器），缓存(Redis,Memcache)，NoSQL，数据库优化，存储索引优化。

epoll之会把哪个流发生了怎样的I/O事件通知我们。

20.对面向对象的认识，面向对象的特征

面向对象编程强调抽象、封装、继承、多态

抽象：我们在定义一个抽象类的时候，实际上就是把一类事物共有的属性和行为提取出来，形成一个物理模型（模版），这种研究问题的方法称为抽象。

封装：就是将类的属性包装起来，用户无需知道对象内部方法的实现细节,但可以根据对象提 供的外部接口。好处可以增强模块的独立性。

继承：子类拥有父类的所有属性，和方法. 继承的好处：抽取出了重复的代码，减少代码冗余; 继承的缺点：耦合性太强

多态：一 个对象变量可以指向多种实际类型的现象。

顺便说一下重载和重写（覆盖）的区别：

重载：相同的方法名，不同的实现机制（通过传入不同个数或类型的参数）。当程序运行过程中自己去判断到底该调用谁。

重写：从父类继承而来的方法不能满足需要的情况下，可以将此方法的逻辑在子类中重新实现。我们最常用的就是重写toString()方法了。

22.java中弱引用

强引用、软引用、弱引用和虚引用

弱引用就是不保证不被垃圾回收器回收的对象，它拥有比较短暂的生命周期，在垃圾回收器扫描它所管辖的内存区域过程中，一旦发现了只具有弱引用的对象，就会回收它的内存，不过一般情况下，垃圾回收器的线程优先级很低，也就不会很快发现那些只有弱引用的对象。

弱引用可以和一个引用队列(ReferenceQueue)联合使用，如果弱引用的对象被垃圾回收的话，Java虚拟机就会把这个弱引用加入相关的引用队列中。引用队列可以很容易地实现跟踪不需要的引用。当你在构造WeakReference时传入一个ReferenceQueue对象，当该引用指向的对象被标记为垃圾的时候，这个引用对象会自动地加入到引用队列里面。

弱引用简单来说就是将对象留在内存的能力不是那么强的引用。弱引用类型可以通过get获取。

WeakHashMap和HashMap几乎一样，唯一的区别就是它的键（不是值!!!）使用WeakReference引用。

软引用：但是如果是软引用可以到达，那么这个对象会停留在内存更时间上长一些。当内存不足时垃圾回收器才会回收这些软引用可到达的对象。

虚引用：虚引用指向的对象十分脆弱，我们不可以通过get方法来得到其指向的对象。它的唯一作用就是当其指向的对象被回收之后，自己被加入到引用队列，用作记录该引用指向的对象已被销毁。虚引用只有在其指向的对象从内存中移除掉之后才会加入到引用队列中。

虚引用使用场景主要由两个。它允许你知道具体何时其引用的对象从内存中移除。第二点，虚引用可以避免很多析构时的问题。

23.@数据表中有一万条数据，查询时需要几次IO

1.**[W]**@了解面向服务框架SOA，微服务吗？

|  |
| --- |
| 1. 什么是SOA的服务？  例：去亚马逊订购了一本书，有不同的服务，如支付网关，库存系统，货运系统等  不需了解业务服务的内部工作细节。对外部世界，它只是个能使用消息交互的黑盒子。如在“支付网关”业务服务获得消息“检查信贷”后会给出输出：这个客户的信贷有或没有。对于“订单系统”，“支付网关”的服务是一个黑盒子  2. SOA服务的主要特点是什么？  A） SOA组件是松耦合的。每个服务是自包含单独逻辑。如采取了“支付网关”的服务，并将它附加到不同系统。  B） SOA服务是黑匣子。在SOA中，服务隐藏有内在的复杂性。只使用交互消息，服务接受和发送消息。通过虚拟化一个服务为黑盒子，服务变得更松散的耦合。  C） SOA服务应该能自定义。  D） SOA服务保持在一个中央存储库。应用程序可在中央存储库中搜索服务，并调用  E） SOA服务可编排和链接实现一特定功能： SOA服务可使用即插即用的方式。如，“业务流程”中有两个服务“安全服务”和“订单处理服务”。从它的业务流程可实现两种类型：一，您可以先检查用户，然后处理订单，或反之亦然。是的，你猜对了，使用SOA可以松散耦合的方式管理服务之间的工作流。  3.什么是SOA？  SOA代表了面向服务的架构。 使用松耦合的黑盒子服务构建业务应用的体系架构，这些服务可通过编排连接在一起以实现特定的功能。  4.什么是合同，地址和绑定？  　每个服务都必须公开一或多个端点，以便让该服务提供给客户端调用。  合同是两或多方间的协议。定义了客户端与服务通信的协议。有描述参数和返回值的方法。  地址是一URL，指向服务的位置。  绑定是决定这个端点如何访问。决定了如何完成通信。如你暴露你的服务，可用SOAP over HTTP或通过TCP的BINARY进行访问。因此，对于这些通信介质将被创建两个绑定。  5. 什么Web服务的SOA？  SOA是一种思想，架构理念。  Web服务是完成它的技术方法。 是实现SOA首选的标准。  SOA中需要的服务是松耦合的。 Web服务进行通信使用的SOAP协议是基于XML的，是非常松散的耦合。回答了什么是服务的一部分。 SOA服务应能够来形容自己。  WSDL描述了如何可访问该服务。SOA服务都位于一个目录： UDDI的描述，从那里可得到Web服务。其实是SOA注册表的实现。  6. SOA的主要好处？  　　有助于建立IT和业务线间的无缝对接，通过产生更多的 IT灵活性，以支持更大的业务灵活性。您的业务流程变化越来越快以及全球剧烈竞争需要SOA能够提供高的灵活性。 SOA能够帮助您更好地重用现有的IT投资和服务。 SOA通过定义服务之间良好接口更易于集成。 SOA还提供了业务伙伴，客户和供应商的服务集成到一个企业的业务流程体系结构模型中。这降低了成本，提高了客户满意度。  7. 什么是可重用的服务？  　　服务是自主的，可重复用的，可发现的，无状态的，有一定粒度的功能，且是一个复合应用程序或一个组合服务的一部分。  可重复用的服务通过业务活动标识，这个业务活动是使用服务规范（设计时合同）描述的。  一个服务约束是，包括安全性，QoS，SLA，使用策略，可由多个运行时的合同多个接口（WSDL中的Web服务）及多个实现（代码）定义的。  可重用服务应在被管制在其从设计到运行整个企业级生命周期。其重用应通过规范流程来推动，重用应该是可测量的。  8。谈谈关于服务的标识，自顶向下 top-down和自底向上bottom-up 的两方法间如何鼓励重用和维护？  自顶向下方法是业务驱动，可在不同计划上分离业务和IT的不同关注，提供两者间共同的基础。因此，在大多数情况下，如想提高重用和中期/长期的投资回报率，可采取  9.在一个SOA中如何实现松耦合？  用服务接口（WSDL中为SOAP Web服务）来限制服务间的依赖性，对消费者隐藏服务实现。松耦合可通过实施服务的功能封装及限制服务接口的实现变化影响来解决。  然而，某些时候，需改变接口，也不会影响服务的消费者，除了管理多个安全约束，多种传输，及其他方面的考虑。  10. 你还记得可以用来实现松散耦合的任何模式？  调解Mediation 模式，用企业服务总线（ESB）也有助于实现这一目标。将建立所有层面上消费者和供应商间的独立性，包括消息格式，消息类型（包括SOAP，REST，XML，二进制）和传输协议（包括HTTP，HTTPS，JMS）。  从体系结构上讲，意味着分离关注消费者和提供者间的传输，消息类型和消息格式的水平。  11.SOA的服务应该被设计为无状态或有状态？  服务应该无状态。有一个无状态的执行上下文，但不会有中间状态来等待一个事件或一个回调。状态有关的数据的保留一定不能超出的服务的请求/响应。这是因为状态管理消耗了大量的资源，这可能会影响服务的可重用 可伸缩性和可用性。  12.什么是服务的组合？  　　组合是一个该服务被组合以产生复合应用或合成服务的流程。一个组合应用程序是由产生企业门户或企业流程的服务聚集。组合服务也是产生另一种可重复使用的服务的集合。这就像相结合的电子组件来创建一个电脑主板，然后使用该主板组装一台电脑一样。想想主板作为一个可重用的组合服务，它是计算机的一个组成部分，计算机是作为一个复合应用程序。  13. 如何使用SOA集成遗留系统？  如遗留老的旧的程序是IT环境的核心。通过正确的技能和工具，将遗留应用程序分立组装，然后基于标准的接口包装成SOA的服务。  14. ESB企业总线是什么？  企业服务总线是SOA的核心要素。 ESB产品提供内部服务之间“任何点到任何点”的连接，并超越您的业务边界连接到您的贸易伙伴。但是，SOA并没有止步于仅仅实现了ESB。您可能需要使用ESB来实现SOA，如消息服务，互动服务和业务流程管理服务中的连接等服务。此外，您将还需要考虑发展服务和IT服务管理服务。 SOA参考架构可以帮助你制定出能够满足您的需求和优先事项的SOA环境。 ESB是这个参考架构的一部分，并提供SOA的骨干，但它不应该在SOA中被单独考虑。  15. 在SOA中我们是否需要从头开始构建系统？  否。如果您需要集成现有系统为业务服务，你只需要创建松耦合的包装，包装您的现有系统，并以一种通用的方式暴露功能给外部世界。  16. 什么是服务和组件之间的区别？  服务组件来实现业务功能的逻辑组件分组。组件是为实现服务这个目标的途径。组件可以使用Java，C＃，C，但服务将以通用格式如像Web服务方式被暴露。  17. SOA概念其实不是什么新鲜事，但是为什么每个人在最近几年都开始谈论SOA？  一些技术在发展变化，过去10年面向服务的体系结构取得了详细实用，比以前应用在更多的组织。其中：  普遍接受的行业标准，如XML及它的许多变种，和Web服务标准一起促成了SOA新的热点。  数据治理框架，这对于成功的SOA实施是很重要的，多年来已经有完善的测试。  各种有利的技术和工具（例如，建模，开发，基础设施/中间件，管理和测试）已经成熟。  注意到企业和企业战略的不断壮大，注意力从技术转移到人，文化的变化，流程是关键业务成功的因素。  18. 什么是你采用SOA最重要的技能？技术或文化？  当然是文化, SOA确实需要人们去思考业务和技术的不同。但不是首先考虑技术（例如，如果我们实现这个系统，我们可以用它做什么样的事情呢？），从业人员必须首先想到的业务功能或服务的条款（例如，我公司要做这些业务功能，所以我如何设置我的IT系统，以最有效地实现这些事情？）预计采用SOA将改变企业的IT部门，建立以服务为导向，而不是技术导向型）IT组织。  19. 什么是实现SOA的主要障碍？  1。短缺的技能。 2. 过分考虑SOA项目的投资回报率。  20. 我可以买一个SOA或者必须自己建立呢？  将你的组织迁走向更大的服务方向，你需要采取平衡的方式来建立与购买SOA。首先需要创建一个SOA的基础架构，你需要补充而不是替代您现有的IT基础设施的和现成的软件。这是“购买”。在“建立”方面，你也可以选择进入相应的专业并动手参与有效地使用这些软件产品，以获得他们的最大效益。这一基础设施和相关的工具可以帮助您创建您的SOA并运行您的业务服务。  21. 我是否需要开始SOA治理吗？  成功SOA实现的一个关键方面是从一开始就努力参与经营。通过SOA你能够得到的价值是改进业务/ IT集成。 SOA治理提供决策权，流程和业务策略和IT一起工作。当服务被部署，就必须管理到位，以控制和监控服务管理方面的问题。开始您不需要很多SOA治理，但足以与正在实施的SOA水平协调就可以。  22 . 什么是SOA切入点？  若要SOA快速入门，您需要选择重点，选择一个初始项目可以在相当短的时间内完成特定的业务。SOA入口点是已被证明能及时提供商业价值的项目区域。每个入口点提供了一个关键的SOA相关的解决方案：  人们 - 通过协作提高生产力，让员工和合作伙伴创造一个与他人互动的个性化的，综合的方式的能力。  流程 - 优化和动态部署流程以及监视改变流程的有效性。  资讯 - 提高业务洞察力和通过使用行和上下文交付降低信息服务风险。  重用 - 可重用的服务是SOA的基石。重用为用户减少周期时间，消除重复流程的灵活性。  连接性 - 虽然在过去，连接性已经成为需求， 但是SOA带来了更高水平的灵活性。  23. 什么是SOA常见的陷阱？  最常见的错误是把SOA作为结束，而不是一种手段。专注于构建SOA解决方案，而不是解决特定业务问题，如创造IT资源之间的复杂，难以管理和不必要的互连。另一个常见的错误是一次试图解决多个问题，而不是解决小部分的问题。  24. SOA是真正符合你的要求？  SOA并不是适合每一个人。虽然SOA带来显著效益和成本节约，SOA要求集中治理以及执行纪律是成功的。对于一些组织，制定和执行这些原则的成本可能比实现带来更高成本而不是收益。 |

* SOA是一组件模型，将应用程序的不同功能单元（称为服务）通过这些服务间定义良好的接口和契约联系起来。接口采用中立的方式 进行定义的，应独立于实现服务的硬件平台、操作系统和编程语言。使得构建在各种各样的系统中的服务可以一种统一和通用的方式进行交互。
* 微服务：彻底的组件化和服务化，Docker容器，进程隔离；微服务的目的是有效的拆分应用，实现敏捷开发和部署 。

**2.【W】分布式系统如何负载均衡(Load Balance)？如何确定访问的资源在哪个服务器上？**

|  |
| --- |
| 先写一个类来模拟Ip列表：  public class IpMap  {  // 待路由的Ip列表，Key代表Ip，Value代表该Ip的权重  public static HashMap<String, Integer> serverWeightMap = new HashMap<String, Integer>();  static  {  serverWeightMap.put("192.168.1.100", 1);  serverWeightMap.put("192.168.1.101", 1);  // 权重为4  serverWeightMap.put("192.168.1.102", 4);  serverWeightMap.put("192.168.1.103", 1);  serverWeightMap.put("192.168.1.104", 1);  // 权重为3  serverWeightMap.put("192.168.1.105", 3);  serverWeightMap.put("192.168.1.106", 1);  // 权重为2  serverWeightMap.put("192.168.1.107", 2);  serverWeightMap.put("192.168.1.108", 1);  serverWeightMap.put("192.168.1.109", 1);  serverWeightMap.put("192.168.1.110", 1);  }  } |

将外部发送来的请求均匀分配到对称结构中的某一台服务器上"的各种算法

一.**轮询**（**Round Robbin**）

最古老、最可信，缺点是有状态，必须在并发之下记住上一次到谁了。

|  |
| --- |
| public class RoundRobin  {  private static Integer pos = 0;    public static String getServer()  {  // 重建一个Map，避免服务器的上下线导致的并发问题  Map<String, Integer> serverMap = new HashMap<String, Integer>();  serverMap.putAll(IpMap.serverWeightMap);    // 取得Ip地址List  Set<String> keySet = serverMap.keySet();  ArrayList<String> keyList = new ArrayList<String>();  keyList.addAll(keySet);    String server = null;  synchronized (pos)  {  if (pos > keySet.size()) {  pos = 0;  }  server = keyList.get(pos);//pos为list中下标  pos ++;  }    return server;  }  } |

serverWeightMap中地址列表动态，随时可能有机器上线、下线或宕机，为避免可能出现的并发问题，方法内部要新建局部变量serverMap，现将serverMap中的内容复制到线程本地，以避免被多个线程修改。可能会引入新的问题，复制后serverWeightMap的修改无法反映给serverMap，也就是说这一轮选择服务器的过程中， 新增或下线服务器，负载均衡算法将无法获知。新增无所谓，如有服务器下线或者宕机，那么可能会访问到不存在的地址。因此，服务调用端需有相应的容错处理，如重新发起一次server选择并调用。

对于当前轮询的位置变量pos，为保证服务器选择的顺序性，需要在操作时对其加锁，使得同一时刻只能有一个线程可以修改pos的值，否则当pos变量被并发修改，则无法保证服务器选择的顺序性，甚至有可能导致keyList数组越界。

优点：试图做到请求转移的绝对均衡。

缺点：为做到请求转移的绝对均衡，必须付出相当大的代价，为保证pos变量修改的互斥性，需要引入重量级的悲观锁synchronized，将会导致该段轮询代码的并发吞吐量明显下降。

二.**随机**。

最简单的，无状态的算法。

通过系统随机函数，根据后端服务器列表的大小值来随机选择其中一台进行访问。由概率统计理论可以得知，随着调用量的增大，其实际效果越来越接近于平均分配流量到每一台后端服务器，就是轮询的效果。

随机法的代码实现大致如下：

|  |
| --- |
| public class Random  {  public static String getServer()  {  // 重建一个Map，避免服务器的上下线导致的并发问题  Map<String, Integer> serverMap = new HashMap<String, Integer>();  serverMap.putAll(IpMap.serverWeightMap);    // 取得Ip地址List  Set<String> keySet = serverMap.keySet();  ArrayList<String> keyList = new ArrayList<String>();  keyList.addAll(keySet);    java.util.Random random = new java.util.Random();  int randomPos = random.nextInt(keyList.size());    return keyList.get(randomPos);  }  } |

整体代码思路和轮询法一致，先重建serverMap，再获取到server列表。在选取server的时候，通过Random的nextInt方法取0~keyList.size()区间的一个随机值，从而从服务器列表中随机获取到一台服务器地址进行返回。基于概率统计的理论，吞吐量越大，随机算法的效果越接近于轮询算法的效果。

* **一致性哈希(源地址哈希（Hash）法)**

思想是获取客户端访问的IP地址值，通过哈希函数计算得到一个数值，用该数值对服务器列表大小进行取模运算，得到的结果便是要访问的服务器的序号。

源地址哈希算法的代码实现大致如下：

|  |
| --- |
| public class Hash  {  public static String getServer()  {  // 重建一个Map，避免服务器的上下线导致的并发问题  Map<String, Integer> serverMap = new HashMap<String, Integer>();  serverMap.putAll(IpMap.serverWeightMap);    // 取得Ip地址List  Set<String> keySet = serverMap.keySet();  ArrayList<String> keyList = new ArrayList<String>();  keyList.addAll(keySet);    // 在Web应用中可通过HttpServlet的getRemoteIp方法获取  String remoteIp = "127.0.0.1";  int hashCode = remoteIp.hashCode();  int serverListSize = keyList.size();  int serverPos = hashCode % serverListSize;    return keyList.get(serverPos);  }  } |

前两部分和轮询法、随机法一样，差别在于路由选择部分。通过客户端的ip(remoteIp)，取得Hash值，对服务器列表的大小取模，结果便是选用的服务器在服务器列表中的索引值。

优点：保证了相同客户端IP地址将会被哈希到同一台后端服务器，直到后端服务器列表变更。可在服务消费者与提供者间建立有状态的session会话。

缺点：除非集群中服务器的非常稳定，基本不会上下线，否则一旦有服务器上线、下线，那么通过源地址哈希算法路由到的服务器是服务器上线、下线前路由到的服务器的概率非常低，如果是session则取不到session，如果是缓存则可能引发"雪崩"。如果这么解释不适合明白，可以看我之前的一篇文章MemCache超详细解读，一致性Hash算法部分。

* **加权轮询（Weight Round Robin）法**

不同服务器可能机器配置和当前系统负载不同，因此抗压能力也不同，给配置高、负载低的机器配置更高的权重，让其处理更多的请求，而低配置、高负载的机器，则给其分配较低的权重，降低其系统负载。加权轮询法可很好地处理这一问题，并将请求顺序按照权重分配到后端。加权轮询法的代码实现大致如下：

|  |
| --- |
| public class WeightRoundRobin  {  private static Integer pos;    public static String getServer()  {  // 重建一个Map，避免服务器的上下线导致的并发问题  Map<String, Integer> serverMap = new HashMap<String, Integer>();  serverMap.putAll(IpMap.serverWeightMap);    // 取得Ip地址List  Set<String> keySet = serverMap.keySet();  Iterator<String> iterator = keySet.iterator();    List<String> serverList = new ArrayList<String>();  while (iterator.hasNext())  {  String server = iterator.next();  int weight = serverMap.get(server);  for (int i = 0; i < weight; i++)  serverList.add(server);  }    String server = null;  synchronized (pos)  {  if (pos > keySet.size())  pos = 0;  server = serverList.get(pos);  pos ++;  }    return server;  }  } |

与轮询法类似，只是在获取服务器地址前增加了一段权重计算的代码，根据权重的大小，将地址重复地增加到服务器地址列表中，权重越大，该服务器每轮所获得的请求数量越多。

* **加权随机（Weight Random）法**

与加权轮询法类似，加权随机法也是根据后端服务器不同的配置和负载情况来配置不同的权重。不同的是，是按照权重来随机选择服务器的，而不是顺序。加权随机法的代码实现如下：

|  |
| --- |
| public class WeightRandom  {  public static String getServer()  {  // 重建一个Map，避免服务器的上下线导致的并发问题  Map<String, Integer> serverMap =  new HashMap<String, Integer>();  serverMap.putAll(IpMap.serverWeightMap);    // 取得Ip地址List  Set<String> keySet = serverMap.keySet();  Iterator<String> iterator = keySet.iterator();    List<String> serverList = new ArrayList<String>();  while (iterator.hasNext())  {  String server = iterator.next();  int weight = serverMap.get(server);  for (int i = 0; i < weight; i++)  serverList.add(server);  }    java.util.Random random = new java.util.Random();  int randomPos = random.nextInt(serverList.size());    return serverList.get(randomPos);  }  } |

这段代码相当于是随机法和加权轮询法的结合，比较好理解，就不解释了。

* **最小连接数**

前几种方法费尽心思来实现服务消费者请求次数分配的均衡，没错，可为后端的多台服务器平均分配工作量，最大程度地提高服务器的利用率。

上面的问题，再换一个角度来说就是：以后端服务器的视角来观察系统的负载，而非请求发起方来观察。最小连接数法便属于此类。

最小连接数算法比较灵活和智能，由于后端服务器的配置不尽相同，对于请求的处理有快有慢，它正是根据后端服务器当前的连接情况，动态地选取其中当前积压连接数最少的一台服务器来处理当前请求，尽可能地提高后端服务器的利用效率，将负载合理地分流到每一台机器。由于最小连接数设计服务器连接数的汇总和感知，设计与实现较为繁琐，此处就不说它的实现了。

|  |  |  |
| --- | --- | --- |
| 集群、负载均衡及分布式系统架构  1.**集群**  1.1 定义：  一组独立的计算机系统构成一个松耦合的多处理器系统，通过网络实现进程间的通信。  应用程序可通过网络共享内存进行消息传送，实现分布式计算机。  是一组连在一起的计算机，从外部看是一个系统，各节点可是不同的操作系统或硬件构成的计算机。  如一个提供Web服务的集群，对外界来看是一个大Web服务器。不过集群的节点也可以单独提供服务。   |  | | --- | | Linux集群可分成三大类：高可用集群， 负载均衡集群，科学计算集群  集群是一个统称，他分为好几种，如高性能科学群集、负载均衡群集、高可用性群集等。   * 科学群集 、高性能集群（High performance cluster，HPC）   利用一个集群中的多台机器共同完成同一任务，使得完成任务的速度和可靠性都远高于单机运行的效果。弥补了单机性能上的不足。在天气预报、环境监控等数据量大，计算复杂的环境中应用比较多；  涉及为群集开发并行编程应用程序，以解决复杂的科学问题。不用专门的超级并行计算机，而是用商业系统（如通过高速连接来链接的一组单处理器或双处理器PC），且在公共消息传递层上进行通信以运行并行应用程序。  常听说一种便宜的 Linux 超级计算机问世了，大多数情况就是指这种集群系统，其处理能力与真的超级计算机相当，而价格与上百万美元的专用超级计算机相比是相当的便宜。   * 负载均衡集群（Load balance cluster， LBC）   是利用一个集群中的多台单机，完成许多并行的小的工作。一般情况下，如果一个应用使用的人多了，那么用户请求的相应时间就会增大，机器的性能也会受到影响，如果使用负载均衡集群，那么集群中任意一台机器都能相应用户的请求，这样集群就会在用户发出服务请求之后，选择当时负载最小，能提供最好服务的机器来接受请求并响应，这样就可用集群来增加系统的可用性和稳定性。这类集群在网站中使用较多；  这种系统使负载可在多台计算机中尽可能平均地分摊处理。负载可是需均衡的应用程序处理负载或网络流量负载。在系统中，每个节点都可处理一部分负载，并且可以在节点之间动态分配负载，以实现平衡。对于网络流量也是如此。   * **高可用性**集群（High availability cluster，HAC）   是利用集群中系统的冗余，当系统中某台机器发生损坏时，其他后备机器可迅速接替它来启动服务，等待故障机的维修和返回。最大限度的保证集群中服务的可用性。这类系统一般在银行，电信服务这类对系统可靠性有高要求的领域有着广泛应用。  是为了使群集的整体服务尽可能可用。如主节点发生故障，这段时间内将由次节点代替它。次节点通常是主节点的镜像，所以当它代替主节点时，它可以完全接管其身份，对用户没有任何影响。  这三种基本类型之间，经常会发生交叉、混合。比如：在高可用性的群集系统中也可以在其节点之间实现负载均衡，同时仍然维持着其高可用性。 |   1.2 **负载均衡系统**：  集群中所有节点都处于活动状态，分摊系统的工作负载。Web服务器集群、数据库集群和应用服务器集群。  负载均衡集群一般用于相应网络请求的网页服务器，数据库服务器。  可在接到请求时，检查接受请求较少，不繁忙的服务器，把请求转到这些服务器上。  从检查其他服务器状态这一点上看，负载均衡和容错集群很接近，不同之处是数量上更多。   |  | | --- | | 负载均衡:指由多台服务器以对称方式组成一个服务器集合，每台服务器都具有等价的地位，都可单独对外提供服务而无须其他服务器的辅助。通过某种负载分担技术，将外部发来的请求均匀分配到对称结构中的某台服务器上，而接到请求的服务器独立地回应客户的请求。负载均衡能够平均分配客户请求到服务器阵列，借此提供快速获取重要数据，解决大量并发访问服务问题，这种集群技术可用最少的投资获得接近于大型主机的性能。  软件负载均衡:代表是阿里章文嵩博士研发的LVS，  硬件负载均衡:均衡服务器比如F5 |   1.3 集群系统主要解决的问题：   * 高可靠性（HA）： 用集群管理软件，当主服务器故障时，备份服务器能自动接管工作，并及时切换过去，以实现对用户的不间断服务。 * 负载均衡：即把负载压力根据某种算法合理分配到集群中的每一台计算机上，以减轻主服务器的压力，降低对主服务器的硬件和软件要求 * 高性能计算（HP）：即充分利用集群中的每一台计算机的资源，实现复杂运算的并行处理，通常用于科学计算领域，比如基因分析，化学分析等。   **2.负载均衡系统**  负载均衡又有DNS负载均衡（较常用）、IP负载均衡、反向代理负载均衡等，也就是在集群中有服务器A、B、C，互不影响，互不相干，任何一台的机器宕了，都不会影响其他机器的运行，当用户来一个请求，由负载均衡器的算法决定由哪台机器来处理，假如你的算法是采用round算法，有用户a、b、c，那么分别由服务器A、B、C来处理；  **2.1基于DNS的负载均衡**  通过DNS服务中的随机名字解析来实现负载均衡，在DNS服务器中，可为多个不同地址配置同一名字，而最终查询这个名字的客户机将在解析这个名字时得到其中一个地址。因此，对同一名字，不同客户机会得到不同地址，也就访问不同地址上的Web服务器，从而达到负载均衡的目的。  **2.2反向代理负载均衡 （如Apache+JK2+Tomcat这种组合）**  用代理服务器可将请求转发给内部的Web服务器，让代理服务器将请求均匀地转发给多台内部Web服务器之一上，从而达到负载均衡的目的。  这种代理方式与普通的代理方式有所不同，标准代理方式是客户使用代理访问多个外部Web服务器，而这种代理方式是多个客户使用它访问内部Web服务器，因此也被称为反向代理模式。  **2.3基于NAT（Network Address Translation）的负载均衡技术 （如Linux Virtual Server，简称LVS）**  网络地址转换为在内、外部地址间转换，以便具备内部地址的计算机能访问外部网络，而当外部网络中的计算机访问地址转换网关拥有的某一外部地址时，地址转换网关能将其转发到一个映射的内部地址上。因此如果地址转换网关能将每个连接均匀转换为不同的内部服务器地址，此后外部网络中的计算机就各自与自己转换得到的地址上服务器进行通信，从而达到负载分担的目的。  **3.分布式是指将不同业务分布在不同地方。**  而集群指的是将几台服务器集中在一起，实现同一业务。  分布式中每一个节点，都可做集群。而集群并不一定就是分布式的。  举例：就比如新浪网，访问的人多了，他可以做一个群集，前面放一个响应服务器，后面几台服务器完成同一业务，如果有业务访问的时候，响应服务器看哪台服务器的负载不是很重，就将给哪一台去完成。  分布式，从窄意上理解，也跟集群差不多，但它的组织比较松散，不像集群，有一个组织性，一台服务器垮了，其它服务器可顶上来。 分布式的每一个节点，都完成不同的业务，一个节点垮了，这个业务就不可访问了。  在群集的这三种基本类型间，经常会发生交叉、混合。比如：在高可用性的群集系统中也可在其节点间实现负载均衡，同时仍然维持着其高可用性。还有一种概括性说法：cluster是手段，load banlance是目标之一。 |

1. 设计一个分布式负载均衡缓冲系统，如何快速定位到是那个服务器（使用key分段、一致性hash）

4.如何保证缓冲区和数据库间的强一致性（用加锁）

事务，锁，

* 写流程：(1)写数据时更新存储 （2）失效缓存数据
* 读流程：

（1）先读cache，如有数据则返回

（2）如数据未命中，则读db

（3）将db数据写入缓存

* 数据库层面的读写并发，引发数据库与缓存数据不一致（本质是后发生的读请求先返回了, 导致旧数据入缓存）。可通过两个小的改动解决：

（1）修改服务连接池，id取模选取服务连接，能保证同一个数据的读写都落在同一后端服务上

（2）修改数据库连接池，id取模选取DB连接，能保证同一数据的读写在数据库层面是串行的

|  |
| --- |
| 不需让全局的请求串行化，只需“让同一个数据的访问能串行化”，只需“让同一个数据的访问通过同一条DB连接执行”就行，只需“在DB连接池层面稍微修改,按数据取连接即可”  获取DB连接的CPool.GetDBConnection()【返回任何一个可用DB连接】改为  CPool.GetDBConnection(longid)【返回id取模相关联的DB连接】 |

~~5.@HashMap高并发情况下会出现什么问题，（扩容问题）~~

6.@说一说在浏览器中输入一个url后，直到浏览器显示页面的过程中发生了什么

（我主要说了DNS，然后他有接着问了DNS的细节，然后就是ARP路由，然后服务器处理，返回，浏览器呈现，获取html中的依赖资源）

**7. 【W】字符串中句子的反转（比如ABC DEF，输出DEF ABC）**

（很简单，可以先反转整个字符串，然后反转单词，或者先将句子切分为单词，然后反转）；

8.@给任意二叉树的所有结点加next指针

**8.1【W】 从上往下打印出二叉树的每个节点，同层节点从左至右打印。(按层打印二叉树)**

**http://blog.csdn.net/qq\_27703417/article/details/55212664**



|  |
| --- |
| class TreeNode {  int val = 0;  TreeNode left = null;  TreeNode right = null;  public TreeNode(int val) {  this.val = val;  }  }  public class Solution {  public static void main(String[] args) {  TreeNode t1=new TreeNode(1);  TreeNode t2=new TreeNode(2);  TreeNode t3=new TreeNode(3);  TreeNode t4=new TreeNode(4);  TreeNode t5=new TreeNode(5);  TreeNode t6=new TreeNode(6);  TreeNode t7=new TreeNode(7);  TreeNode t8=new TreeNode(8);    t1.left=t2; t1.right=t3;  t2.left=t4; t3.left=t5; t3.right=t6;  t5.left=t7; t5.right=t8;  new Solution().printFromTopToBottom(t1);  }  public ArrayList<Integer> printFromTopToBottom(TreeNode root) {  ArrayList<Integer> list=new ArrayList<Integer>();  Queue <TreeNode> queue=new ArrayBlockingQueue<>(100);  TreeNode last=root; //当前行的最后节点  TreeNode nLast=root; //下一行的最右节点  queue.add(root);    while (!queue.isEmpty()) {  TreeNode out=queue.poll();  System.out.print(out.val+" ");  list.add(out.val);  if (out.left!=null) {  queue.add(out.left);  nLast=out.left;  }  if (out.right!=null) {  queue.add(out.right);  nLast=out.right;  }  if (out==last) {  System.out.println("");  last=nLast;  }    }  return list;  }  } |

**9.@用过反向代理吗？**

~~1.@@进程间共享内存的方式有哪些？（8种）~~

~~无名管道，有名管道，高级管道，信号量，消息队列，共享内存，套接字。~~

2.linux下如何查看网络端口状态（netstat），如何查看内存使用情况（top）；

3. 【W】@ @ConcurrentHashMap如何扩容，内部结构？

4.知道java的异常吗？

5.运行时异常如果不处理会怎么样？应该怎么处理运行时异常？

不会怎么样，会在运行的时候出错。（RuntimeError是uncheck异常，所以不需要捕获）

6.@@写代码：给你5000万个int，求出前1000个最大的数，有2G内存。

（我刚开始以为5000万个int很多，还把G和byte的换算忘了，后来面试官指导才想起来。我的方法是维护一个1000的小根堆，然后遍历数组，总体下来时间复杂度是O（nlg1000)）

见排序的堆排序。

7.@@给你n个不重复的整数，随机找出m个不重复的整数，要求时间和空间复杂度都是O(m)。

（方法很简单，就是每次把取出来的数放到后面，只在前面的数组随机访问就可以了，时间复杂度是O(m)，空间复杂度是O(1)，不过我刚开始没有想到把选出来的放后面去。。。）。

8.@@对于SQL慢查询的优化？

（主要是从查询语句和数据库表设计两个方面来考虑，查询语句方面可以增加索引，增加查询筛选的限制条件；数据库表设计的时候可以拆分表，设计得更细粒度。但是后来才发现面试官想要的就是查询大量数据的慢查询问题的优化。。。）

先使用explain等找到耗时的地方，然后针对性的优化。SQL语句的优化。应该是需要针对业务数据进行一些特殊的优化。比如数据分类等。如果无法改变数据库设计，那就只能优化SQL语句，改变数据库参数设置了。

9.@@用过哪些容器？

（tomcat）对比过Tomcat与其他服务器的区别吗？比如nginx？

nginx常用做静态内容服务和代理服务器，Tomcat能够动态的生成资源并返回到客户端。

Nginx 应该叫做「HTTP Server」；而 Tomcat 则是一个「Application Server」，或者更准确的来说，是一个「Servlet/JSP」应用的容器；

虽然Tomcat也可以认为是HTTP服务器，但通常它仍然会和Nginx配合在一起使用：

动静态资源分离——运用Nginx的反向代理功能分发请求：所有动态资源的请求交给Tomcat，而静态资源的请求（例如图片、视频、CSS、JavaScript文件等）则直接由Nginx返回到浏览器，这样能大大减轻Tomcat的压力。

负载均衡，当业务压力增大时，可能一个Tomcat的实例不足以处理，那么这时可以启动多个Tomcat实例进行水平扩展，而Nginx的负载均衡功能可以把请求通过算法分发到各个不同的实例进行处理。

10.用过动态代理吗？

AOP。

11.@@有遇到过OOM吗？

三@@@：[美]

1.项目中的数据流向（画结构框图）

2.看过什么书？

3.说说深入理解JVM中印象最深刻的章节（我说了JVM内存模型，垃圾回收和类加载）；

补充：还问了堆和栈中存的是什么？static修饰的变量存在哪里？（方法区）

4.说说《Effective Java》中你印象最深的三条和你的理解

能不用string的就不使用；

用静态工厂方法代替构造器；

不要过度使用异常；

重写equals方法；

重写equals方法的时候记得重写hashcode；

使类和成员的可访问性最小化等等。

5.你觉得你哪一块只是最熟悉（我说了Java的基本数据结构）

6.那你说说HashMap的内部实现；

jdk1.8是用的红黑树，之前用的Entry数组，而Entry是一个链表类型的数据结构。

7.HashMap是线程安全的吗？（不是，ConcurrentHashMap是）

8.那ConcurrentHashMap内部是如何实现的？每个segment是个什么数据结构？（HashTable）

由多个segment组成，每个segment是一个HashEntry数组，HashEntry是链表。

实现同步的方法是segment继承了ReentrantLock类；Hashtable实现线程安全的方法是使用synchronized。

9.你的项目中用到哪些技术？（Spring）

10.说说你用了它的什么？（Spring IOC用的最多）

11.Spring的优点？Spring AOP的原理？Spring如何实现解耦合？

|  |
| --- |
| 1.使用Spring的IOC容器，将对象之间的依赖关系交给Spring，降低组件之间的耦合性，让我们更专注于应用逻辑。  2.可以提供众多服务，事务管理，WS等。  3.引入AOP技术，将公共服务抽象出来，简化了代码编写；  4.对主流的框架提供了很好的集成支持，方便继承各种优秀的框架；  5.Spring属于低侵入，代码污染极低。  6.降低Java EE API的使用难度； |

12.对链表了解吗？（我说是List吗）是，（了解ArrayList和LinkedList），那你说说他们的区别？

13.会做链表两个结点的交换吗？（链表反转吗？）是的，你写代码实现把。

14.再写一个，给你一个链表和一个整数k（k大于等于0，小于等于链表长度，链表长度未知），按k步长反转链表（比如1->2->3->4->5->6->7，当k=3的时候结果是3->2->1->6->5->4->7)

15.@@mybatis配置了xml过后是如何完成数据库操作的？

16.static变量内存分配在哪里？

在方法区，方法区用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译后的代码等。

1.@@判断一个数是不是丑数

2.打印树的每一层最右侧节点

public void printRight(TreeNode root){

if(root == null)

return;

LinkedList<Integer> queue = new LinkedList<Integer>();

queue.add(root);

int cur = 1;

int nextLine = 0;

int curSum = 1;

while(!queue.isEmpty()){

TreeNode tmp = queue.poll();

if(curSum == cur)

System.out.print(tmp.val+",");

cur--;

if(tmp.left != null){

queue.add(root.left);

nextLine++;

}

if(tmp.right!= null){

queue.add(root.right);

nextLine++;

}

if(cur == 0){

cur = nextLine;

curSum = nextLine;

nextLine = 0;

}

}

}

3.HashMap 的原理，实现简单的 get/put 操作；

4.map 源码，怎样解决冲突，和扩容

5.servlet流程

init->service->destory;

6.@@序列化，以及json传输

7.@@tomcat均衡方式 ，netty

8.java的反射机制

|  |
| --- |
| 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；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。 |

-------------------------------------网上的美团---------------------------------------------------------

1. java中hashMap结构，处理冲突方法，还有啥方法，各个方法优缺点。

|  |
| --- |
| 1.开放定址法，这种方法也称再散列法，其基本思想是：当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。  2.再哈希法，这种方法是同时构造多个不同的哈希函数：  Hi=RH1（key） i=1，2，…，k  当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。  3. 链地址法，这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。  4.建立公共溢出区，这种方法的基本思想是：将哈希表分为基本表和溢出表两部分，凡是和基本表发生冲突的元素，一律填入溢出表。 |

3.@@红黑树。

4.@@进程调度；

|  |
| --- |
| 无论是在批处理系统还是分时系统中，用户进程数一般都多于处理机数、这将导致它们互相争夺处理机。另外，系统进程也同样需要使用处理机。这就要求进程调度程序按一定的策略，动态地把处理机分配给处于就绪队列中的某一个进程，以使之执行。  调度算法：  先来先服务和短作业(进程)优先调度算法，高优先权优先调度算法，基于时间片的轮转调度算法。 |

5.@@页面置换；

在地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

先入先出（FIFO）法。这种算法的实质是，总是选择在主存中停留时间最长（即最老）的一页置换，即先进入内存的页，先退出内存。

当需要置换一页时，选择在之前一段时间里最久没有使用过的页面予以置换。这种算法就称为最久未使用算法

8.spring，quartz,spark,lucene，

Quartz是一个完全由Java编写的开源作业调度框架,为在Java应用程序中进行作业调度提供了简单却强大的机制。

Spark 是一种与 Hadoop 相似的开源集群计算环境。

lucene：是一个开放源代码的全文检索引擎工具包，但它不是一个完整的全文检索引擎，而是一个全文检索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎。

9.@@自己实现IOC

<http://blog.csdn.net/d12345678a/article/details/53909646>

10.const/reactor

Reactor 是一个基于 JVM 之上的异步应用基础库。

12. 【W】@@volatile实现原理《书》

共享变量的“可见性”。当一线程修改一共享变量时，另一线程能读到这个修改的值。

这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效。

处理器为提高处理速度，不直接和内存进行通讯，而是先将系统内存数据读到内部缓存后再操作，但操作完后不知何时会写到内存.

如对声明了Volatile的变量进行写，JVM会向处理器发送Lock前缀指令，将变量所在缓存行数据写回到系统内存。

但其他处理器缓存值还是旧的，为保证各处理器的缓存一致，实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存值是不是过期，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

一个处理器的缓存回写到内存会导致其他处理器的缓存无效。

13.spring的一些知识，如：有哪些注入方式

三种：通过set方式注入，通过构造器注入，通过工厂方法（接口注入，具有侵入性，不推荐）

XML配置，Java注解，构造器。

14.boolean占几个字节，protected权限能否被包外访问

因为jvm没有boolean类型，所以如果jvm实现规范的话，那么如果是boolean类型的数组的话，每个boolean类型（按照byte数组存储）占用1bytes,否者按照int类型存储，占用4bytes.得出boolean类型占了单独使用是4个字节，在数组中又是1个字节。

protected权限不能被包外访问。

16.@@说怎么判断是一个人注册大量小号。

从判断注册页面过来的请求是否是同一IP之类的说了

问题1：美团面试官来到一个城市面试应聘者，面试有三天，每天面试官上午可以面试三场，下午可以面试四场，怎么设计面试系统，面试者可以选择面试日期，面试时间和面试官。

标记，同步，

问题2：有一些爬虫IP不断的访问美团网站，现在美团设定一个IP5分钟之内访问美团网站超过100次，就判定为爬虫IP，怎么设计这个程序？如果100改成10000，怎么设计？

读取log中的IP记录，并统计每个IP的访问次数，以5分钟为分隔，超过100的IP就是爬虫IP，没有超过的在5分钟的时候清零，继续统计。

1.关键字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。

【F】@@MyISAM引擎 MySQL 5.1及之前版本的默认引擎：

|  |
| --- |
| 不支持行锁，读-表锁，写-表锁  不支持事务、外键、崩溃后的安全恢复  读时可写  支持BLOB和TEXT的前500个字符索引，全文索引  支持延迟更新索引，提升写性能  对不改的表，支持压缩表，减少磁盘空间占用 |

InnoDB MySQL 5.5后默认：

|  |
| --- |
| 支持行锁，采用MVCC来支持高并发  支持事务  支持外键  支持崩溃后的安全恢复  不支持全文索引 |

1.怎样设计实现一个高效的线程安全的hashmap 。

方法一：通过Collections.synchronizedMap()返回一个新的Map，这个新的map就是线程安全的。 这个要求大家习惯基于接口编程，因为返回的并不是HashMap，而是一个Map的实现。

方法二：重新改写了HashMap，具体的可以查看java.util.concurrent.ConcurrentHashMap. 这个方法比方法一有了很大的改进。（锁分离）

方法一使用的是的synchronized方法,是一种悲观锁.在进入之前需要获得锁,确保独享当前对象,然后做相应的修改/读取.

方法二使用的是乐观锁,只有在需要修改对象时,比较和之前的值是否被人修改了,如果被其他线程修改了,那么就会返回失败.锁的实现,使用的是 NonfairSync. 这个特性要确保修改的原子性,互斥性,无法在JDK这个级别得到解决,JDK在此次需要调用JNI方法,而JNI则调用CAS指令来确保原子性与互斥性.

重写和重载

（1）重写：需要继承；方法吗和参数、返回值必须相同，权限修饰符必须大于等于父类；不可以重写父类的private方法；不可以抛出父类没有抛出的异常；

（2）重载：方法名必须相同；参数个数、类型有一个不相同；返回值可以相同也可以不同；可以有不同的访问修饰符；可以抛出不同异常。

2.用过的设计模式。写单例模式代码，各种情况下比如去掉某一句会怎样。

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

单例模式：

工厂模式：程序在接口和子类之间加入了一个过渡端，通过此过渡端可以动态取得实现了共同接口的子类实例化对象。

代理模式：指由一个代理主题来操作真实主题，真实主题执行具体的业务操作，而代理主题负责其他相关业务的处理。

观察者模式：

适配器模式：如果一个类要实现一个具有很多抽象方法的接口，但是本身只需要实现接口中的部分方法便可以达成目的，所以此时就需要一个中间的过渡类，但此过渡类又不希望 直接使用，所以将此类定义为抽象类最为合适，再让以后的子类直接继承该抽象类便可选择性的覆写所需要的方法，而此抽象类便是适配器类。

3、介绍同步锁，使用情景；

synchronized

4、假如有两个线程，一个线程A，一个线程B都会访问一个加锁方法，可能存在并发情况，但是线程B访问频繁，线程A访问次数很少，问如何优化。(然后面试官说有了解过重度锁和轻度锁吗)

5、【F】@@@进程间通信方式；

|  |
| --- |
| 1.无名管道( pipe )：管道半双工，数据只能单向流,只能在有亲缘关系(父子进程)的进程间用。  2.高级管道(popen)：将另一个程序当做当前程序的子进程,启动  3.有名管道 (named pipe):半双工，允许无亲缘关系进程间通信。  4.消息队列( message queue )：消息链表存在内核中,由消息队列标识符标识。克服了信号传递信息少、管道只能承载无格式字节流及缓冲区大小受限等缺点。  5.信号量( semophore ):计数器,控制多进程对共享资源的访问。锁机制。进程间及同一进程内多线程间的同步手段。  6.信号 ( sinal ) :较复杂，通知接收进程某事件已发生。  7.共享内存( shared memory ) :映射一段能被其他进程访问的内存，一个进程创建，多个进程可访问。共享内存是最快的 IPC 方式，是针对其他进程间通信方式运行效率低而专门设计的。往往与信号量配合用，实现进程间的同步和通信。  8.套接字( socket ) :可用于不同机器间的进程通信。 |

6、有一个整数，转换为二进制，请找出所有的1的数量；

只计算1的个数使用：

while(n>0)

count++;

n = n & (n-1);

7、你觉得Java是完全面相对象的语言吗？

8、讲解下你对抽象，多态的理解；

抽象就是将实际事物特征抽象出来，用一个类表示，类中的属性就表示事物的不同特征。

抽象类和接口就是抽象的一种表象。

多态

①使用父类类型的引用指向子类的对象；

②该引用只能调用父类中定义的方法，不能调用子类中独有的方法；

③如果子类中重写了父类中的一个方法，那么在调用该方法的时候，将会调用子类中的这个方法；

④在多态中，子类可以调用父类中的所有方法；

⑤多态主要体现为方法重载和方法重写，相同的行为，不同的实现方式；

⑥多态的主要应用是多态参数和多态集合

⑦如果一个方法中有父类作为参数类型，就意味着可以给该方法传递任何一个子类对象。

9、@@讲解下http请求里的header有什么作用；

10.Get和post区别。

（1）get用于获取数据，而且应该是安全的和幂等的。post用于提交数据。（多用于新建）。

（2）GET请求的数据会附在URL之后，最多只能是1024字节；POST把提交的数据则放置在是HTTP包的包体中，理论上数据大小没有限制。

（3）POST的安全性要比GET的安全性高。

11. 【W】@@Http协议相关。

基于 TCP/IP 协议

应用层协议。

不涉及数据包（packet）传输，主要规定客户端和服务器间通信格式，默认用80端口。

HTTP/1.0不支持连接保持

HTTP/1.1增加了Connection:keep-alive及管道(允许浏览器同时发出A/ B请求)，但服务器还是按顺序。

HTTP/2.0:二进制协议;双向、实时通信（多工），避免了队头阻塞；

用ID区分数据流，客户端发出的数据流，ID为奇数，服务器发出的，ID为偶数；

可取消数据流；

客户端还可指定数据流优先级。越高，服务器越早回应；

引入了头信息压缩机制和信息表；

允许服务器未经请求，主动向客户端发送资源，叫服务器推送；

12.Java有没有多继承，接口和抽象类区别。jvm 理解

接口可以多继承，类不可以。

（1）抽象类是由abstract关键字修饰，允许包含未实现的方法。

接口是方法声明和常量值定义的集合，不允许包含变量。

（2）接口是绝对抽象的，不可以被实例化。抽象类不能被实例化。

（3）定义接口时可以使用继承，而且可以使用多继承。抽象类只能单继承。类可以实现很多个接口，但是只能继承一个抽象类；

（4）接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。Java接口中声明的变量默认都是final的。抽象类可以包含非final的变量。

（5）Java接口中的成员函数默认是public的。抽象类的成员函数可以是private，protected或者是public。

类可以不实现抽象类和接口声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

抽象类也不可以被实例化，但是，如果它包含main方法的话是可以被调用的。

public是在不同包中也可以访问；protected是必须在同一个包中或者子类才能访问；default是不允许子类访问，可以在类内部或者同一个包中访问；private是只能在类内部访问。

13. 【F】@@JDK和JRE的区别是什么？

Java运行时环境(JRE): JVM。为用户编写的JAVA代码提供运行环境

Java开发工具包(JDK):完整Java软件开发包，含JRE(为JDK自带开发工具提供运行环境)，编译器和其他工具(如JavaDoc，Java调试器)，可让开发者开发、编译、执行Java应用程序。

Classpath: JDK的lib目录

Path:JDK的bin目录。

14. 【F】@@数据库四个范式的区别。

1NF 属性不可拆分或无重复列（保证列唯一）

2NF 存在一个列被定义为唯一主键的表就是第二范式。（保证行唯一）

3NF 要求一个数据库表中不包含已在其它表中已包含的非主键信息。（外键关系）

4NF 禁止主键列和非主键列一对多关系不受约束用复合列做主键的表。

15. 【W】@@64个马8个跑道，没计时工具，最少需多少次跑，如何找到跑最快的1个和4个马？

先分8组，找出每组第一再跑一次，一共9次就可以找到第一名；

第九次的第一名的那组的二（1）、三（2），四（3），

第二名（4）那组的第二名（5），第三名（6），

第三名（7）的第二名（8）

再跑一次就可找到最快的第4个马。

16.正则表达式

\s表示所有空白字符；\S表示非空白字符；\*表示匹配零次或多次；+表示匹配一次或多次；. 表示除换行符外的任意字符；？匹配零次或一次。

1.如何停止运行一个线程

interrupt

2.如何实现一个hashmap，需要考虑哪些问题

hashcode

3.equal和==的区别

值类型是存储在内存中的堆栈（以后简称栈），而引用类型的变量在栈中仅仅是存储引用类型变量的地址，而其本身则存储在堆中。

==操作比较的是两个变量的值是否相等，对于引用型变量表示的是两个变量在堆中存储的地址是否相同，即栈中的内容是否相同。

equals操作表示的两个变量是否是对同一个对象的引用，即堆中的内容是否相同。

String s2 = new String("Monday").intern();会放入缓冲池。

4.如何判断一个对象已经被回收

5.@@为什么四次挥手比三次握手多一次

因为三次握手是要建立和确认客户端和服务器之间的通信线路；而四次挥手是为了关闭两方之间的数据传递，因为是双工的，所以被动方和主动方两个流向都需要关闭和确认，所以是四次。

4. linux相关指令，把能记得的说一下，出了个小功能，用shell实现；

5. @@sql中replace和update的区别；

update是SQL中的数据更新命令，replace是VF中的数据更新命令；

update不需要打开表，repl命令需要先打开表；

update不加条件是更新表中所有记录，repl不加条件是更新当前记录，要加上all短语才是更新所有记录。

9. 接触过哪些软件测试的方法。

黑盒，白盒测试；

性能测试，压力测试，冒烟测试；

3. @@如何拦截5分钟前恶意登录（用户名密码不正确）超多一定次数的ip

6.介绍一下信号量和互斥锁

信号量又称为信号灯，它是用来协调不同进程间的数据对象的，而最主要的应用是共享内存方式的进程间通信。本质上，信号量是一个计数器，它用来记录对某个资源（如共享内存）的存取状况。

互斥体禁止多个线程同时进入受保护的代码“临界区”（critical section）。因此，在任意时刻，只有一个线程被允许进入这样的代码保护区。

对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，"自旋"一词就是因此而得名。

7.@@画项目的系统架构图，功能的流程图。

——————————————————————————-——————————

二面：

输入字符流，找第一个重复的字符串。

1、Linux下的一些指令，$$（进程id），$?（上一条命令退出时状态），怎么查看进程，按照内存大小，CPU占用排序等等。

（大写M和大写P）

2、项目中遇到的问题，自己咋解决的等等。

多数据库，使用Mybatis的拦截器。

3、介绍一下hash，怎么解决冲突。（链地址法）

4、进程间的通信，共享内存方式的优缺点。

环境变量/文件描述符：子进程接受父进程环境数据的拷贝以及所有文件描述符。

管道：用于相关和无关进程间的通信，而且形成两个进程间的一个通信通道，通常使用文件读写程序访问。

共享内存：使用客户机/服务器模型（C/S），服务器对客户的数据或动作请求作出反应。

动态数据交换：动态数据交换使用消息传递、共享内存、事务协议、客户/服务器范畴、同步规则以及会话协议来让数据和控制信息在进程间流动。动态数据交换对话( dynamic data exchange session, DDE )的基本模型是客户、服务器。服务器对来自客户的数据或动作作出反应。客户和服务器可以以多种关系来通信。

共享内存针对消息缓冲的缺点改而利用内存缓冲区直接交换信息，无须复制，快捷、信息量大是其优点。但是共享内存的通信方式是通过将共享的内存缓冲区直接附加到进程的虚拟地址空间中来实现的．因此，这些进程之间的读写操作的同步问题操作系统无法实现。必须由各进程利用其他同步工具解决。另外，由于内存实体存在于计算机系统中．所以只能由处于同一个计算机系统中的诸进程共享,不方便网络通信。

共享内存块提供了在任意数量的进程之间进行高效双向通信的机制。每个使用者都可以读取写入数据，但是所有程序之间必须达成并遵守一定的协议，以防止诸如在 读取信息之前覆写内存空间等竞争状态的出现。不幸的是，Linux无法严格保证提供对共享内存块的独占访问，甚至是在您通过使用IPC\_PRIVATE创 建新的共享内存块的时候也不能保证访问的独占性。 同时，多个使用共享内存块的进程之间必须协调使用同一个键值。

5.讲下堆的作用。

6.堆的垃圾回收对象选择原则。

所谓根集就量正在执行的Java程序可以访问的引用变量的集合(包括局部变量、参数、类变量)，程序可以使用引用变量访问对象的属性和调用对象的方法。垃 圾收集首选需要确定从根开始哪些是可达的和哪些是不可达的，从根集可达的对象都是活动对象，它们不能作为垃圾被回收，这也包括从根集间接可达的对象。而根 集通过任意路径不可达的对象符合垃圾收集的条件，应该被回收。下面介绍几个常用的算法。

7 进程间通信详细叙述

8 tcp udp 区别 UDP应用场景

UDP一般用于即时通信（QQ聊天 对数据准确性和丢包要求比较低，但速度必须快），在线视频（RTSP 速度一定要快，保证视频连续，但是偶尔花了一个图像帧，人们还是能接受的），网络语音电话（VoIP 语音数据包一般比较小，需要高速发送，偶尔断音或串音也没有问题）等等。

9 linux中文件名存储在哪里？ 改变文件名之后 MD5会不会变

只有两项数据存放在目录项中：文件名和i节点编号。不会的,只有文件内容发生改变MD5才会变化。一般对文件求MD5值只取文件内容部分，像文件名、保存目录、创建时间、修改时间、权限等等的元信息都是不进行计算的。

7 linux命令相关 问用过哪些 然后展开

9 crontab 怎么设置一个每小时、每3小时的定时任务

6. 求一个struct的大小

7.Linux命令相关，问有一个文件A.txt，里面有许多行，找出其中带关键字'B'的行，并统计重复度。我问了下重复度是指啥，他说，这样吧，假设每一行都是由空格分隔开的若干字符，若整个文件中，有2行的最后一个字符都是'10'，你就输出 “10” : 2。

8.假如在服务器上执行一个进程时，你发现服务器很卡顿，你会怎么查找原因。

9.time\_wait 是TCP四次握手哪个阶段的状态，为什么要有这个状态。这个状态带来的好处和坏处是什么。在网络中，什么与这个状态相似

~~1. 【F】@@Spring的事务有哪些？~~

~~什么是数据库事务：访问并可能改变数据库中个数据项的一个程序执行单元。~~

~~实现方式有两种：编码方式即采用注解的方式（类头的@Transactional为默认事务配置）；声明式事务管理方式（bean）。~~

~~基于AOP技术实现的声明式事务管理，实质就是：在方法执行前后进行拦截，然后在目标方法开始之前创建并加入事务，执行完目标方法后根据执行情况提交或回滚事务。~~

~~声明式事务管理又有两种方式：基于XML配置文件的方式；另一个是在业务方法上进行@Transactional注解，将事务规则应用到业务逻辑中。~~

2. 【F】@@Java的锁有哪几种？

1、自旋锁:让当前线程不停在循环体内执行，当循环条件被其他线程改变时才能进入临界区。

3、阻塞锁：synized

4、可重入锁：ReentrantLock

5、读写锁：写锁是排他锁，读锁是共享锁。

6、互斥锁

7、悲观锁：实现依靠数据库提供的锁机制。

8、乐观锁：假设数据一般不会冲突，数据提交更新时，才会正式对数据的冲突与否进行检测，如发现冲突，返回用户错误的信息，让用户决定如何去做。

公平锁（Fair）：加锁前检查是否有排队等待的线程，优先排队等待的线程，先来先得

非公平锁（Nonfair）：加锁时不考虑排队等待问题，直接尝试获取锁，获取不到自动到队尾等待

18、信号量

3.join的优势是什么？

mysql不支持Full join。

select \* from A inner join B on A.name = B.name;

在ON匹配阶段 WHERE 子句的条件都不会被使用。仅在匹配阶段完成以后，WHERE 子句条件才会被使用。它将从匹配阶段产生的数据中检索过滤。

cross join：交叉连接，得到的结果是两个表的乘积，即笛卡尔积。

还有inner join，left join，right join。

4.用线程实现输出1A2B3C....

5.抽象类和接口的区别

6.JDK1.8和之前版本的区别

JDK7新特性：

1，switch中可以使用字串了

2，"<>"这个玩意儿的运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断

3. 自定义自动关闭类

4. 新增一些取环境信息的工具方法

5. Boolean类型反转，空指针安全,参与位运算

6. 两个char间的equals

7，安全的加减乘除

8.数值可加下划线

例如：int one\_million = 1\_000\_000;

9.支持二进制文字

例如：int binary = 0b1001\_1001;

10、在try catch异常扑捉中，一个catch可以写多个异常类型，用"|"隔开

11、jdk7之前，你必须用try{}finally{}在try内使用资源，在finally中关闭资源，不管try中的代码是否正常退出或者异常退出。jdk7之后，你可以不必要写finally语句来关闭资源，只要你在try()的括号内部定义要使用的资源。

JDK8：

一、接口的默认方法

Java 8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法

二、Lambda 表达式

Collections.sort(names, (a, b) -> b.compareTo(a));

三、函数式接口？

四、方法与构造函数引用 ::

Java 8 在包java.time下包含了一组全新的时间日期API。

7.常用集合类

我的说的HashMap，他问为什么不是List。

1。JSP的内置对象及方法。

在JSP中无需创建就可以使用的9个对象，它们是：request、response、session、application、out、pagecontext、config、page、exception

l out（JspWriter）：等同与response.getWriter()，用来向客户端发送文本数据；

1. void clear() ;

清除输出缓冲区的内容，但是不输出到客户端。

2. void clearBuffer() ;

清除输出缓冲区的内容，并输出到客户端。

3. void close() ;

关闭输出流，清除所有内容。

4. void flush() ;

输出缓冲区里面的数据。

5. int getBufferSize() ;

获取以kb为单位的目前缓冲区大小。

6. int getRemaining() ;

获取以kb为单位的缓冲区中未被占用的空间大小。

7. boolean isAutoFlush() ;是否自动刷新缓冲区。

8. void newLine() ;

输出一个换行字符。

l config（ServletConfig）：对应“真身”中的ServletConfig，config对象用来存放Servlet初始的数据结构；config

对象的主要作用是取得服务器的配置信息。通过 pageConext对象的 getServletConfig() 方法可以获取一个config对象。当一个Servlet 初始化时，容器把某些信息通过 config对象传递给这个 Servlet。

l page（当前JSP的真身类型）：当前JSP页面的“this”，即当前对象，page对象代表JSP对象本身，或者说代表编译后的servlet对象，可以用(

(javax.servlet.jsp.HttpJspPage)page )来取用它的方法和属性；

l pageContext（PageContext）：页面上下文对象，它是最后一个没讲的域对象，pageContext对象存储本JSP页面相关信息，如属性、内建对象等；pageContext

对象的作用是取得任何范围的参数，通过它可以获取 JSP页面的out、request、reponse、session、application 等对象。

l exception（Throwable）：只有在错误页面中可以使用这个对象，错误对象，只有在JSP页面的page指令中指定isErrorPage="true"后，才可以在本页面使用exception对象；

l request（HttpServletRequest）：即HttpServletRequest类的对象，request对象包含所有请求的信息，如请求的来源、标头、cookies和请求相关的参数值等，request对象代表了客户端的请求信息，主要用于接受通过HTTP协议传送到服务器的数据。（包括头信息、系统信息、请求方式以及请求参数等）。

request对象的作用域为一次请求。；

getParameterNames() 获取客户端提交的所有参数的名字。

l response（HttpServletResponse）：即HttpServletResponse类的对象，response对象主要将JSP容器处理后的结果传回到客户端，response

代表的是对客户端的响应，主要是将JSP容器处理过的对象传回到客户端。response对象也具有作用域，它只在JSP页面内有效；

response.sendRedirect(index.jsp)

l application（ServletContext）：即ServletContext类的对象，主要功用在于取得或更改Servlet的设定；

setAttribute(String key,Object obj)：将参数Object指定的对象obj添加到Application对象中，并为添加的对象指定一个索引关键字。

getAttribute(String key)：获取Application对象中含有关键字的对象。

l session（HttpSession）：即HttpSession类的对象，不是每个JSP页面中都可以使用，如果在某个JSP页面中设置<%@page session=”false”%>，说明这个页面不能使用session，session对象表示目前个别用户的会话状态，用来识别每个用户。

public String getId()：获取Session对象编号。

public void setAttribute(String key,Object obj)：将参数Object指定的对象obj添加到Session对象中，并为添加的对象指定一个索引关键字。

public Object getAttribute(String key)：获取Session对象中含有关键字的对象。

public Boolean isNew()：判断是否是一个新的客户。

2.说说ArrayList（对比着LinkedList来说）；

1.ArrayList是实现了基于动态数组的数据结构，ArrayList不具有线程安全性，LinkedList基于链表的数据结构，LinkedList可以看做为一个双向链表，LinkedList也是线程不安全的。

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。

Vector是线程安全的

26.数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用Array而不是ArrayList？

下面列出了Array和ArrayList的不同点：

Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。

Array大小是固定的，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。

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

27.ArrayList和LinkedList有什么区别？

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

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。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和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)。

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。

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。

28. 【W】@@Comparable和Comparator接口干什么的？区别。

* Comparable接口只含一个compareTo()。

可给两对象排序。

返回负数，0，正数来表明输入对象小于，等于，大于已存在对象。

* Comparator接口包含compare()和equals()。

compare()给两个输入参数排序，返回负数，0，正 数表明第一个参数是小于，等于，大于第二个参数。

equals()需一对象作参数，只有当输入参数也是一comparator, 且输入参数和当前comparator排序结果相同时, 才返回true。

3.写一个单例模式。我写的是静态内部类的单例，然后他问我这个地方为什么用private，这儿为啥用static,这就考察你的基本功啦

类的构造函数定义为private的，保证其他类不能实例化此类，static保证每个类访问到的都是同一个。

4. 【F】@@写SQL语句。

SQL语句语法、聚集函数和Group By的用法

姓名 分数 课程

name score course

统计出每个学生有多少门课分数大于80分

select name,count(\*) as num from student where score > 80 group by name;

5.说一说GC算法吧。

<http://www.cnblogs.com/hnrainll/archive/2013/11/06/3410042.html>

（讲到了minorGC)

当两个存活区切换了几次（HotSpot虚拟机默认15次，用-XX:MaxTenuringThreshold控制，大于该值进入老年代）之后，仍然存活的对象（其实只有一小部分，比如，我们自己定义的对象），将被复制到老年代。Eden区与Survivor的比例较大，HotSpot默认是

8:1。在发生Minor GC时，虚拟机会检查每次晋升进入老年代的大小是否大于老年代的剩余空间大小，如果大于，则直接触发一次Full GC。

1) 从JVM内存模型开始说起，在纸上画出大概的组成部分，然后说出每个组成部分的特点

线程隔离的有：虚拟机栈，本地方法栈和程序计数器；

线程共享的有：方法区（被虚拟机加载的类的元数据信息：如常量、静态变量、即时编译器编译后的代码。也成为永久代）和堆区（存放所有对象实例和数组）；

程序计数器（Program Counter Register）：

每一个Java线程都有一个程序计数器来用于保存程序执行到当前方法的哪一个指令，对于非Native方法，这个区域记录的是正在执行的VM原语的地址，如果正在执行的是Natvie方法，这个区域则为空（undefined）。此内存区域是唯一一个在VM Spec中没有规定任何OutOfMemoryError情况的区域。

Java虚拟机栈：

本地变量表存放了编译期可知的各种标量类型（boolean、byte、char、short、int、float、long、double）、对象引用（不是对象本身，仅仅是一个引用指针）、方法返回地址等。其中long和double会占用2个本地变量空间（32bit），其余占用1个。

本地方法栈：

本地方法栈是为虚拟机使用到的Native方法服务。

堆：

绝大部分的对象实例和数组都在这里分配。

2) 开始说说分代GC，这时就把GC算法引入进来，再结合每个区域的特点 把Minor GC 和Full GC 引入进来

标记-清除算法：从根节点开始标记所有可达对象，其余没标记的即为垃圾对象，执行清除。但回收后的空间是不连续的。

复制算法：将内存分成两块，每次只使用其中一块，垃圾回收时，将标记的对象拷贝到另外一块中，然后完全清除原来使用的那块内存。复制后的空间是连续的。（适用于新生代）

标记-压缩算法：适合用于老年代的算法（存活对象多于垃圾对象）。标记后不复制，而是将存活对象压缩到内存的一端，然后清理边界外的所有对象。

2) 可以跟他说说垃圾回收器，Serial 、 ParNew 、CMS 等等

Serial 是单线程的，新生代使用的是复制算法，老年代使用的是标记整理算法；

ParNew是Serial的多线程版本，Parallel Scavenge是ParNew的增强版，提供可控制的吞吐量设置；

Serial Old和ParNew Old对应的是老年代的GC；

CMS是一种以获取最短回收停顿时间为目标的收集器；使用的标记清楚算法；

初始标记，并发标记，重新标记，并发清除；

21.垃圾回收器G1,我只说了CMS

CMS包含初始标记、并发标记、重新标记、并发清除4步。其中初始标记和重新标记是需要停止其他操作的，并发标记和并发清除是与应用线程并发执行。

但是CMS是CPU资源敏感的；无法处理浮动垃圾；CMS是基于“标记-清除”算法的，所以有碎片。

G1是面向服务端的，特点：并行与并发、分代收集、空间整合（使用标记-整理算法）、可预测停顿。它不再区分新老生代，而是划分region，并通过Remembered Set来避免全堆扫描。

G1收集过程是：初始标记、并发标记、最终标记和筛选回收。除了并发标记是并发的，其他都是阻塞的。

6.说说死锁吧，怎么检测死锁？

死锁：是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。

四个条件同时具备：互斥条件、不可抢占条件、占有且申请条件、循环等待条件。

避免方法：

（1）严格按照一定顺序访问资源；

（2）在取得进程能够执行的所有资源后才给进程分配资源，否则不分配；

（3）银行家算法：

7.说一说ThreadLocal 关键字吧，答出核心。还有volatile，synchronized，Lock

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

安全：使用元子类，实现并发锁，使用线程安全的类，使用volatile关键字，

8.@@聚簇索引和非聚簇索引的区别 。

建立聚族索引： CREATE CLUSTER INDEX index\_name ON table\_name(column\_name1,column\_name2,...);

存储特点：

聚集索引：表数据按照索引的顺序来存储的，也就是说索引项的顺序与表中记录的物理顺序一致。对于聚集索引，叶子结点即存储了真实的数据行，不再有另外单独的数据页。 在一张表上最多只能创建一个聚集索引，因为真实数据的物理顺序只能有一种。

非聚集索引 ： 表数据存储顺序与索引顺序无关。对于非聚集索引，叶结点包含索引字段值及指向数据页数据行的逻辑指针，其行数量与数据表行数据量一致。

9.说一说项目中Spring的IOC和AOP具体怎么使用的。

几乎都用的IOC(通过Spring容器进行对象的管理，以及对象之间组合关系的映射)

日志管理是AOP

10.谈一谈对数据库事务的理解。四个特性和隔离级别必须要牢记，了解

事务并发控制带来的问题：脏读、不可重复读、更新丢失、幻读等等

原子性，一致性，隔离性，持久性；

① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

③ Read committed (读已提交)：可避免脏读的发生。

④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

MySql在哪一个级别？

4种都有啊？

11.问我Hibernate的生成策略，主要说了native 、uuid

12.说说struts，我就拿出struts 1 和 struts 2 对比着说了

13.mysql 间隙锁？InnodB，MyISAM区别？

http://blog.csdn.net/d12345678a/article/details/53858117

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据记录的索引项加锁；对于键值在条件范围内但并不存在 的记录，叫做“间隙（GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制就是所谓的间隙锁（Next-Key锁）。InnoDB除了通过范围条件加锁时使用间隙锁外，如果使用相等条件请求给一个不存在的记录加锁，InnoDB也会使用间隙锁！

Select \* from emp where empid > 100 for update;

间隙锁的主要作用是为了防止出现幻读，但是它会把锁定范围扩大，有时候也会给我们带来麻烦。

间隙锁，然后数据库会向左扫描扫到第一个比给定参数小的值， 向右扫描扫描到第一个比给定参数大的值， 然后以此为界，构建一个区间， 锁住整个区间内的数据。

InnoDB存储引擎既支持行级锁（row-level locking），也支持表级锁，但默认情况下是采用行级锁。MyISAM存储引擎只支持表锁。

MyISAM表锁是deadlock free的，这是因为MyISAM总是一次获得所需的全部锁，要么全部满足，要么等待，因此不会出现死锁。

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

MySQL的表级锁有两种模式：表共享读锁（Table Read Lock）和表独占写锁（Table Write Lock）。MyISAM在执行查询语句（SELECT）前，会自动给涉及的所有表加读锁，在执行更新操作（UPDATE、DELETE、INSERT等）前，会自动 给涉及的表加写锁，这个过程并不需要用户干预，因此，用户一般不需要直接用LOCK TABLE命令给MyISAM表显式加锁。

例子：

Lock tables orders read local, order\_detail read local;

Select sum(total) from orders;

Select sum(subtotal) from order\_detail;

Unlock tables;

上面的例子在LOCK TABLES时加了“local”选项，其作用就是在满足MyISAM表并发插入条件的情况下，允许其他用户在表尾并发插入记录。MyISAM是写优先的。

通过定期在系统空闲时段执行 OPTIMIZE TABLE语句来整理空间碎片，收回因删除记录而产生的中间空洞。

InnoDB与MyISAM的最大不同有两点：一是支持事务（TRANSACTION）；二是采用了行级锁。

l 共享锁（S）：允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。

l 排他锁（X)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。

另外，为了允许行锁和表锁共存，实现多粒度锁机制，InnoDB还有两种内部使用的意向锁（Intention Locks），这两种意向锁都是表锁。

l 意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。

l 意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

共享锁（S）：SELECT \* FROM table\_name WHERE ... LOCK IN SHARE MODE。

排他锁（X)：SELECT \* FROM table\_name WHERE ... FOR UPDATE。

InnoDB行锁是通过给索引上的索引项加锁来实现的.由于MySQL的行锁是针对索引加的锁，不是针对记录加的锁，所以虽然是访问不同行的记录，但是如果是使用相同的索引键，是会出现锁冲突的。如果不是相同的索引就不会等待。

create table tab\_no\_index(id int,name varchar(10)) engine=innodb;

一是MySQL的恢复是SQL语句级的，也就是重新执行BINLOG中的SQL语句。

二是MySQL的Binlog是按照事务提交的先后顺序记录的，恢复也是按这个顺序进行的。

14.java的优势

简单（GC）、面向对象、平台无关（JVM）、分布式、多线程、可靠和安全等特性

15.有没有用过哪些集合?

Collection 接口的接口 对象的集合

├ List 子接口 按进入先后有序保存 可重复

│├ LinkedList 接口实现类 链表 插入删除 没有同步 线程不安全

│├ ArrayList 接口实现类 数组 随机访问 没有同步 线程不安全

│└ Vector 接口实现类 数组 同步 线程安全

│ 　 └ Stack

├ Queue 子接口 队列集合

└ Set 子接口 仅接收一次，并做内部排序

├ HashSet

│ 　 └ LinkedHashSet 插入的次序

└ TreeSet

Map 接口 键值对的集合

├ Hashtable 接口实现类 同步 线程安全 键值非空

├ HashMap 接口实现类 没有同步 线程不安全

│├ LinkedHashMap 插入次序

│└ WeakHashMap

├ TreeMap 基于红黑树，可以返回子树 排序的

└ IdentifyHashMap

Collections 是针对集合类的一个帮助类。提供了一系列静态方法实现对各种集合的搜索、排序、线程完全化等操作。相当于对 Array 进行类似操作的类—— Arrays 。

16. 【F】@@Arraylist如何实现排序

Collections.sort()传入ArrayList，用默认方式排序（字典序）

Collections.sort()传入ArrayList和自己实现Commparator接口的类的对象，实现自定义排序

List.sort()传入自己实现Commparator接口的类的对象，实现自定义排序

17. 【W】@@Spring的annotation（注解）如何实现

Annotation是代码里特殊标记，可在编译、类加载、运行时被读取，并执行相应处理。

用Annotation，可在不改变原有逻辑情况下，在源文件中嵌入补充信息。

代码分析工具、开发工具和部署工具可通过这些补充信息进行验证、处理或部署。

* 4个元注解：

1.@Target, 注解使用范围，@Target(ElementType.FIELD)

2.@Retention, 注解生命周期，@Retention(RetentionPolicy.RUNTIME)

3.@Documented,可被javadoc类工具文档化

4.@Inherited，被标注的class的子类所继承。类并不从它实现的接口继承annotation，方法并不从它所重载的方法继承annotation。

* 三种标准annontation类型：

（1）Override，重载父类方法；

（2）Deprecated

（3）SuppressWarnings，告诉Java编译器关闭对类、方法及成员变量的警告。

@Autowired 自动装配，消除代码Java代码里面的getter/setter与bean属性中的property。

如容器有一个以上匹配的Bean，可通过@Qualifier注解限定Bean名称；

@Service业务层Bean；

@Repository数据访问层Bean；

@Configuration把一个类作为IoC容器，它的某个方法头上如注册了@Bean，就会作为这个Spring容器中的Bean。

@Scope 作用域

@Lazy(true) 延迟初始化

@Service业务层组件、

@Controller控制层组件（如struts中的action）

@Component泛指组件，当组件不好归类时用

@PostConstruct指定初始化方法（用在方法上）

@PreDestory指定销毁方法（用在方法上）

@DependsOn：定义Bean初始化及销毁时顺序 默认 单例 启动就加载

@Primary：自动装配时当出现多个Bean候选者，注解为@Primary的Bean将作为首选者，否则将抛出异常

@Autowired 默认按类型装配，如想按名称，可结合@Qualifier。如下：

@Autowired @Qualifier("personDaoBean") 存在多个实例配合使用

@Resource默认按名称装配，当找不到才会按类型。

@Async 异步方法调用

【W】@@Restful的优势：

透明性，暴露资源存在。

充分利用 HTTP 协议本身语义。

无状态，这点非常重要。在调用一接口（访问、操作资源）时，可不用考虑上下文，当前状态，降低了复杂度

HTTP 本身提供了丰富的内容协商手段，无论是缓存，还是资源修改的乐观并发控制，都可以业务无关的中间件来实现

低耦合；

18.@@Redis如何解决key冲突

19.HashMap，ConcurrentHashMap以及在什么情况下性能会不好

HashMap不是线程安全的

23.Java中的HashMap的工作原理是什么？

Java中的HashMap是以键值对(key-value)的形式存储元素的。HashMap需要一个hash函数，它使用hashCode()和 equals()方法来向集合/从集合添加和检索元素。当调用put()方法的时候，HashMap会计算key的hash值，然后把键值对存储在集合中 合适的索引上。如果key已经存在了，value会被更新成新值。HashMap的一些重要的特性是它的容量(capacity)，负载因子(load factor)和扩容极限(threshold

resizing)。

24.hashCode()和equals()方法的重要性体现在什么地方？

equals一般是用来实现两个对象内容的比较，如果不重写就会和==一样。一般重写equals方法的时候都会相应的重写hashCode方法。（重写hashCode方法是个难点）

Java中的HashMap使用hashCode()和equals()方法来确定键值对的索引，当根据键获取值的时候也会用到这两个方法。如果没有正确 的实现这两个方法，两个不同的键可能会有相同的hash值，因此，可能会被集合认为是相等的。而且，这两个方法也用来发现重复元素。所以这两个方法的实现 对HashMap的精确性和正确性是至关重要的。

如何保证key的唯一性？即hashCode和equals的实现原理

相等的对象必须具有相同的散列码，但散列码相同则不一定是相等的对象，而且不相等的对象也不一定需要有不同的散列码。

25.HashMap和Hashtable有什么区别？

HashMap和Hashtable都实现了Map接口，因此很多特性非常相似。但是，他们有以下不同点：

HashMap允许键和值是null，而Hashtable不允许键或者值是null。

Hashtable是同步的，而HashMap不是。因此，HashMap更适合于单线程环境，而Hashtable适合于多线程环境。

HashMap提供了可供应用迭代的键的集合，因此，HashMap是快速失败的。另一方面Hashtable提供了对键的列举(Enumeration)。一般认为Hashtable是一个遗留的类。

20.Synchronized和lock区别

wait()和notify()必须在synchronized代码块中调用。

在并发量比较小的情况下，使用synchronized是个不错的选择，但是在并发量比较高的情况下，其性能下降很严重，此时ReentrantLock是个不错的方案。

区别：

（1）Synchronized是java关键字，是内置特性；而Lock是一个类；

（2）Lock可以让等待锁的线程响应中断，而synchronized却不行；

（3）Lock可以提高多个线程进行读操作的效率；

（4）Synchronized的锁是自动释放的，Lock需要手动释放；

（5）Lock可以知道线程有没有成功获得锁。

java.util.concurrent.locks有Lock是一个接口：

void lock();

void lockInterruptibly() throws InterruptedException;

boolean tryLock();

boolean tryLock(long time,

TimeUnit unit) throws InterruptedException;

void unlock();

ReentrantLock（可重入锁）是唯一实现了Lock接口的类。

单独调用interrupt()方法不能中断正在运行过程中的线程，只能中断阻塞过程中的线程。

ReentrantReadWriteLock是唯一实现ReadWriteLock接口的类，有ReadLock和WriteLock方法

synchronized和ReentrantLock都是可重入锁。

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

ReentrantLock lock = new ReentrantLock(true);

，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

　　而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。

public void test() {

...

synchronized(this) {

// todo your code

}

...

}

此时，其效果等同于

public synchronized void test() {

// todo your code

}

22.一般线程和守护线程的区别

Daemon的作用是为其他线程的运行提供服务，比如说GC线程。thread.setDaemon(true)必须在thread.start()之前设置。 在Daemon线程中产生的新线程也是Daemon的。

当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程

24.JDK中哪些体现了命令模式

命令模式：将一个请求封装为一个对象，从而让我们可用不同的请求对客户进行参数化，用于“行为请求者”与“行为实现者”解耦，可实现二者之间的松耦合，以便适应变化。命令模式是一种对象行为型模式，其别名为动作(Action)模式或事务(Transaction)模式。

java.lang.Runnable 和 javax.swing.Action是使用命令模式的经典场景。

25.简单介绍下线程池的参数？你了解哪些线程池？线程池使用了什么设计模式？线程池使用时一般要考虑哪些问题?

java.util.concurrent.ThreadPoolExecutor;

corePoolSize：核心池的大小

maximumPoolSize：线程池最大线程数

keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。

threadFactory：线程工厂，主要用来创建线程；

handler：表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

线程池状态：

static

final int

RUNNING = 0;

static

final int

SHUTDOWN = 1;

static

final int

STOP = 2;

static

final int

TERMINATED = 3;

execute()

submit()

shutdown()

shutdownNow()立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务.

线程池的种类：

Executors.newCachedThreadPool(); //创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

Executors.newSingleThreadExecutor(); //创建容量为1的缓冲池

Executors.newFixedThreadPool(int); //创建固定容量大小的缓冲池

Executors.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，类似于Timer。

一般需要根据任务的类型来配置线程池大小：

如果是CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 NCPU+1

如果是IO密集型任务，参考值可以设置为2\*NCPU

单个任务处理的时间很短而请求的数目却是巨大的。join和close都可以用来关闭线程池。不同的是，join会把队列中的任务执行完，而close则立刻清空队列，并且中断所有的工作线程。

线程池大小，死锁，系统资源不足，并发错误，线程泄漏（比如都在等待用户输入），任务过载（太多等待任务）。

26.@@一致性Hash原理

27. 【W】@@Java虚拟机的一些参数配置

* Trace跟踪参数：

|  |
| --- |
| -XX:+PrintGCDetails 打印GC详细信息；  -XX:+PrintGCTimeStamps 打印GC时间戳；  -Xloggc:log/gc.log 指定GC log的位置；  -XX:+PrintHeapAtGC 每一次GC前和GC后，都打印堆信息；  -XX:+TraceClassLoading 监控类的加载；  -XX:+PrintClassHistogram 按下Ctrl+Break后，打印类的信息； |

* 堆分配参数：

|  |
| --- |
| -Xmx20m 最大堆  -Xms5m 最小堆； |

问题1： -Xmx（最大堆空间）和 –Xms（最小堆空间）应该保持一个什么关系，可以让系统的性能尽可能的好呢？

问题2：如果你要做一个Java的桌面产品，需要绑定JRE，但是JRE又很大，如何做一下JRE的瘦身呢？

-Xmn 官方推荐新生代占堆的3/8，幸存代占新生代的1/10

　　　　设置新生代大小

-XX:NewRatio=4 //新生代（eden+2\*s）/老年代（不含永久区）

//4，新生代:老年代=1:4，即新生代=1/5堆

-XX:SurvivorRatio=2//（幸存代）Survivor区/eden

　　　 //例：8，表两Survivor:eden=2:8，即一个Survivor=1/10年轻代

-XX:+HeapDumpOnOutOfMemoryError

-XX:HeapDumpPath=d:/a.dump//如OOM异常，就把dump信息导出到d:/a.dump文件中。

-XX:OnOutOfMemoryError=D:/tools/jdk1.7\_40/bin/printstack.bat %p //p代表当前进程的pid

执行printstack.bat脚本：D:/tools/jdk1.7\_40/bin/jstack -F %1 > D:/a.txt，即当程序OOM时，在D:/a.txt中将会生成线程的dump。

-XX:PermSize=16m -XX:MaxPermSize：设置永久区的初始和最大空间；

栈的分配参数：

|  |
| --- |
| Xss 栈大小。只有几百K；-Xss128K  -XX:+UseParallelGC -XX:ParallelGCThreads=20仅对年轻代有效。  -XX:MaxGCPauseMillis=100 : 每次年轻代垃圾回收的最长时间  -XX:+UseCMSCompactAtFullCollection ：打开对年老代的压缩。 |

**常见配置**

* 堆设置

-Xms :初始堆大小

-Xmx :最大堆大小

-XX:NewSize=n :设置年轻代大小

-XX:NewRatio=n: 如:为3，表年轻代:年老代=为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n :如3，表示Eden：Survivor=3：2，一个Survivor区=1/5年轻代

-XX:MaxPermSize=n :设置持久代大小

* 收集器设置

-XX:+UseSerialGC :串行收集器

-XX:+UseParallelGC :并行收集器

-XX:+UseParalledlOldGC :并行年老代

-XX:+UseConcMarkSweepGC :并发收集器

* 垃圾回收统计信息

-XX:+PrintGC

-XX:+PrintGCDetails

-XX:+PrintGCTimeStamps

-Xloggc:filename

* 并行收集器设置

-XX:ParallelGCThreads=n :并行收集器收集时用的CPU数。并行收集线程数。

-XX:MaxGCPauseMillis=n :并行收集最大暂停时间

-XX:GCTimeRatio=n :设置垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)

* 并发收集器设置

-XX:+CMSIncrementalMode :设置为增量模式。适用于单CPU情况。

-XX:ParallelGCThreads=n :设置并发收集器年轻代收集方式为并行收集时，使用的CPU数。并行收集线程数。

28.Raft协议的leader选举，正常情况下，网络抖动造成follower发起leader选举，且该follower的Term比现有leader高。集群中所有结点的日志信息当前一致，这种情况下会选举成功吗？

只要term加1之后，之前的leader就作废，这轮选举一定能选出来。我觉得还得看下这个term什么时候会加1，意思就是题目中follower在什么情况下term比之前leader高。因为follower在一个时间段内因为网络波动收不到leader的心跳，他就自动转化为candidate状态且term+1。

29.Tcp连接4次挥手的原因。Time\_wait等待超时了会怎样？

time\_wait 状态本来就是等待2个msl就直接进入closed状态

TIME\_WAIT状态维持时间是两个MSL时间长度，也就是在1-4分钟。Windows操作系统就是4分钟。

MSL就是maximum segment lifetime(最大分节生命期），这是一个IP数据包能在互联网上生存的最长时间.

time\_wait存在的原因：

1.可靠的终止TCP连接。

2.保证让迟来的TCP报文段有足够的时间被识别并丢弃。

为什么要三次握手：

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误。

“已失效的连接请求报文段”的产生在这样一种情况下：client发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达server。本来这是一个早已失效的报文段。但server收到此失效的连接请求报文段后，就误认为是client再次发出的一个新的连接请求。于是就向client发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要server发出确认，新的连接就建立了。由于现在client并没有发出建立连接的请求，因此不会理睬server的确认，也不会向server发送数据。但server却以为新的运输连接已经建立，并一直等待client发来数据。这样，server的很多资源就白白浪费掉了。

为什么需要四次挥手：

原因是因为tcp是全双工模式，接收到FIN时意味将没有数据再发来，但是还是可以继续发送数据。

30.不同进程打开了同一个文件，那么这两个进程得到的文件描述符（fd）相同吗？

31.给定一个2叉树，打印每一层最右边的结点

32. 【W】@@spring bean初始化过程，Spring的DI工作原理，spring bean生命周期

|  |
| --- |
| 1. 容器找Bean的定义,将其实例化。  2. 用依赖注入，Spring按Bean定义信息配置Bean的属性。  3. 如Bean实现BeanNameAware接口，工厂调用Bean的setBeanName()方法传递Bean的ID  4.如Bean实现BeanFactoryAware接口，工厂调用setBeanFactory()方法传入工厂自身。  5.果BeanPostProcessor和Bean关联，那么它们的postProcessBeforeInitialzation()将被调用。  6.如Bean指定了init-method方法，它将被调用。  7.如有BeanPsotProcessor和Bean关联，那么它们的postProcessAfterInitialization()方法将被调用。 |

生命周期：执行完上面的步骤就可以使用了，使用完毕后关闭容器：

调用DisposableBean的destory方法；

Bean在实例化的过程中：Constructor > @PostConstruct >InitializingBean > init-method

Bean在销毁的过程中：@PreDestroy > DisposableBean > destroy-method

33.说一说对java io的理解

在整个Java.io包中最重要的就是5个类和一个接口。5个类指的是File、OutputStream、InputStream、Writer、Reader；一个接口指的是Serializable.

适配器模式，改变接口来重复使用

把一个类的接口变成客户端能接受的另一种接口。

使两个接口不一样的类能一起工作。

InputStreamReader适配器实现了Reader接口，持有InputStream引用。这样Reader字节最终调用InputStream

InputStreamReader：对象+接口

装饰器模式，接口不变，增强原来对象的功能

赋予类更多的功能，

FileInputStream,

BufferedInputStream是具体实现者，把InputStream读取内容保存在内存中，增强了功能。

34.juc包（Java.util.concurrent）内有哪些类，CAS（比较并交换（compare

and swap））如何实现，说一下对线程池的理解

它的基础就是AbstractQueuedSynchronizer抽象类，Lock，CountDownLatch等的基础就是该类，而该类又用到了 CAS操作。CAS之前已经介绍过。JUC在同步器的设计上有获取操作和释放操作。AbstractQueuedSynchronizer（AQS）就是 一种同步器的实现。

原子对象；

locks；

Semaphore一个计数信号量，主要用于控制多线程对共同资源库访问的限制。

CountDownLatch一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。CyclicBarrier用于一组线程执行时，每个线程执行有多个节点，每个节点的处理需要相互等待。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。但是有ABA的问题，可以通过版本号来解决。

public synchronized int compareAndSwap(int expectedValue, int newValue) {

int oldValue = value;

if (oldValue == expectedValue) {

value = newValue;

}

return oldValue;

}

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

36.树的中序遍历，除了递归和栈还有什么实现方式

Morris遍历，实际上是用叶子节点的空指针来记录当前节点的位置，然后一旦遍历到了叶子节点，发现叶子节点的右指针指向的是当前节点，那么就认为以当前节点的左子树已经遍历完成。

它的遍历规则如下：

1. 如果当前节点的左子节点为空时，输出当前节点，并将当前节点置为该节点的右子节点；

2. 如果当前节点的左子节点不为空，找到当前节点左子树的最右节点（该节点为当前节点中序遍历的前驱节点）；

2.1. 如果最右节点的右指针为空(right=null)，将最右节点的右指针指向当前节点，当前节点置为其左子节点；

2.2. 如果最右节点的右指针不为空，将最右节点右指针重新置为空(恢复树的原状)，输出当前节点，并将当前节点置为其右节点；

3. 重复1~2，直到当前节点为空。

37.操作系统如何实现输出

38.说一下对B+树的了解

39.linux查询java进程

ps -ef | grep java

kill -9 [PID] -9 表示强迫进程立即停止

40.linux查看内存占用情况

cat

/proc/meminfo

sudo atop atop命令是一个终端环境的监控命令。它显示的是各种系统资源的综合，并且在高负载的情况下进行了彩色标注。

top top命令提供了实时的运行中的程序的资源使用统计。

free -h free命令是一个快速查看内存使用情况的方法，它是对 /proc/meminfo 收集到的信息的一个概述。

ps命令可以实时的显示各个进程的内存使用情况。

vmstat命令显示实时的和平均的统计，覆盖CPU、内存、I/O等内容。

41.String，是否可以继承，“+”怎样实现，与StringBuffer，StringBuilder区别

因为有final关键字，所以不能被继承。

“+”是通过StringBuffer或者是StringBuilder的append()和toString()方法来实现的。

String类对象为不可变对象，StringBuffer类对象为可修改对象，可以通过append()方法来修改值；都是线程安全的。

StringBuilder线程不安全，但是效率比StringBuffer高。

42.map 源码，怎样解决冲突，和扩容

开放地址法是对那些发生冲突的记录，用hi=(h(key)+di)mod n方法再次确定Hash地址。

n：为哈希表长;

di：为增量序列，其取法有以下三种：

1）线性探测再散列 di= c \* i

2）二次探测再散列 di = 12, -12, 22, -22, …,

3) 随机探测再散列 di是一组伪随机数列 或者 di=i×H2(key) (又称双散列函数探测)

链地址法：将所有哈希地址相同的记录都链接在同一链表中

43.servlet流程

Web Client 向Servlet容器（Tomcat）发出Http请求

Servlet容器接收Web Client的请求

Servlet容器创建一个HttpRequest对象，将Web Client请求的信息封装到这个对象中。

Servlet容器创建一个HttpResponse对象

Servlet容器调用HttpServlet对象的service方法，把HttpRequest对象与HttpResponse对象作为参数传给 HttpServlet 对象。

HttpServlet调用HttpRequest对象的有关方法，获取Http请求信息。

HttpServlet调用HttpResponse对象的有关方法，生成响应数据。

Servlet容器把HttpServlet的响应结果传给Web Client。

44.异常：Java中的两种异常类型是什么？他们有什么区别？

Java异常分为Exception和Error两大类型，Exception又分为运行时异常（RuntimeException)和非运行时异常（如IOException），运行时异常代表运行时由Java虚拟机生成的异常，Java编译器要求Java程序必须捕获或生命所有的非运行时异常，但对运行时异常可以不做处理。

Java中有两种异常：受检查的(checked)异常和不受检查的(unchecked)异常。不受检查的异常不需要在方法或者是构造函数上声明，就算方法或者是构造函数的执行可能会抛出这样的异常，并且不受检查的异常可以传播到方法或者是构造函数的外面。相反，受检查的异常必须要用throws语句在方法或者是构造函数上声明。这里有Java异常处理的一些小建议。

Java中Exception和Error有什么区别？

方法运行过程中如果发生了意外，方法将生成一个相应类型的异常，并把它交给运行时系统，这一过程称为抛出。运行时系统受到异常后，会寻找相应的代码来处理，这一过程称为捕获。

Exception和Error都是Throwable的子类。Exception用于用户程序可以捕获的异常情况。Error是指Java虚拟机内部发生错误，由虚拟机生产并抛出，定义了不期望被用户程序捕获的异常。

Java的异常处理方式有两种：（1）try-catch-finally；（2）throws。

为什么使用自定义异常？

因为项目中有时会有符合Java语法，但是不符合业务逻辑的情况，这时就需要自定义异常了。

45.@@序列化，以及json传输

47.tomcat均衡方式 ，netty

48.session/cookie

什么是cookie？session和cookie有什么区别？

cookie是Web服务器发送给浏览器的一块信息。浏览器会在本地文件中给每一个Web服务器存储cookie。以后浏览器在给特定的Web服务器发请求的时候，同时会发送所有为该服务器存储的cookie。下面列出了session和cookie的区别：

无论客户端浏览器做怎么样的设置，session都应该能正常工作。客户端可以选择禁用cookie，但是，session仍然是能够工作的，因为客户端无法禁用服务端的session。

在存储的数据量方面session和cookies也是不一样的。session能够存储任意的Java对象，cookie只能存储String类型的对象。

设置session有效期的方法有：

（1）在Tomcat的server.xml中定义；

（2）在工程的web.xml中定义；

（3）通过session.setMaxInactiveInterval(1000)；

判断session是否失效：

request.getSession(false) == null ?

清楚session中的信息：session.invalidate()；

Cookie可以设置有效期：cookie.setMaxAge(120)；

在禁用Cookie的情况下使用Session：response.encodeurl(url);

49.利用数组，实现一个循环队列类

队为空的判断：front==rear；队为满的判断：(rear+1)%MAXSIZE==front。

public class CircleQueue {

private int[] q = null;

//首索引

private int head = 0;

//尾索引

private int tail = 0;

private int len = 0;

public CircleQueue(int len){

this.len = len;

q = new int[len];

}

//insert into circle queue

private int insert(int x){

//判断循环队列是否已满

if (isFull()){

System.out.println("circle queue is Full");

return -1;

}else {

q[tail] = x;

//逻辑上实现首尾相连，循环队列

tail = (tail + 1) % len;

//返回最近插入的元素

return x;

}

}

private int pop(){

if (isNull()){

System.out.println("circle queue is Null");

return -1;

}else {

int n = q[head];

//头结点不断追赶尾结点

head = (head + 1)%len;

//返回最近弹出的元素

return n;

}

}

private int head(){

return q[head];

}

private int tail(){

return q[(tail + len - 1)%len];

}

private boolean isFull(){

if ((tail + 1) % len == head){

return true;

}else {

return false;

}

}

//check is null

private boolean isNull(){

//首尾相等的话则为空

if (head == tail){

return true;

}else {

return false;

}

}

}

50.http头，TCP/IP 有几层，每层有何含义

GET http://download.microtool.de:80/somedata.exe

Host: download.microtool.de

Accept:\*/\* 告诉WEB服务器自己接受什么介质类型，\*/\* 表示任何类型

Pragma: no-cache

Cache-Control: no-cache 指定请求和响应遵循的缓存机制。

Connection：Keep-Alive 表示连接状态

Referer: http://download.microtool.de/

User-Agent:Mozilla/4.04[en](Win95;I;Nav)

Range:bytes=554554-

响应：

HTTP/1.0 200 OK

Date:Mon,31Dec200104:25:57GMT

Server:Apache/1.3.14(Unix)

Content-type:text/html

Last-modified:Tue,17Apr200106:46:28GMT

Etag:"a030f020ac7c01:1e9f"

Content-length:39725426

Content-range:bytes554554-40279979/40279980

TCP/IP的4层模型：

（1）应用层：FTP、DNS、Telnet、SMTP、HTTP

（2）传输层：TCP（传输控制协议）、UDP（用户数据报协议）

（3）网络层：IP

（4）网络接口层：接受和发送IP数据包。

51.DNS的解析流程

1、在浏览器中输入www.qq.com域名，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

2、如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

3、如果hosts与本地DNS解析 器缓存都没有相应的网址映射关系，首先会找TCP/ip参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的 域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。

4、如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。

5、如果本地DNS服务器本地区域文 件与缓存解析都失效，则根据本地DNS服务器的设置（是否设置转发器）进行查询，如果未用转发模式，本地DNS就把请求发至13台根DNS，根DNS服务 器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负 责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址

(qq.com)给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找qq.com域服务器，重复上面的动作，进行查询，直至找到 www.qq.com主机。

6、如果用的是转发模式，此DNS服 务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地 DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。

当应用过程需要将一个主机域名映射为IP地址时，就调用域名解析函数，解析函数将待转换的域名放在DNS请求中，以UDP报文方式发给本地域名服务器。 本地的域名服务器查到域名后，将对应的IP地址放在应答报文中返回。同时域名服务器还必须具有连向其他服务器的信息以支持不能解析时的转发。若域名服务器

不能回答该请求，则此域名服务器就暂成为DNS中的另一个客户，向根域名服务器发出请求解析，根域名服务器一定能找到下面的所有二级域名的域名服务器，这样以此类推，一直向下解析，直到查询到所请求的域名。

52.MapReduce

52.Object中的方法

protected Object clone()

创建并返回此对象的一个副本。

boolean equals(Object obj)

指示其他某个对象是否与此对象“相等”。

protected void finalize()

当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。

Class<?> getClass()

返回此 Object 的运行时类。

int hashCode()

返回该对象的哈希码值。

void notify()

唤醒在此对象监视器上等待的单个线程。

void notifyAll()

唤醒在此对象监视器上等待的所有线程。

String toString()

返回该对象的字符串表示。

void wait()

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法前，导致当前线程等待。

void wait(long timeout)

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量前，导致当前线程等待。

void wait(long timeout, int nanos)

在其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。

54.进程和线程的区别是什么？

进程是执行着的应用程序，而线程是进程内部的一个执行序列。一个进程可以有多个线程。线程又叫做轻量级进程。

进程是CPU分配资源的最小单位，线程是CPU调度和程序执行的最小单位；

一个进程有几个（至少一个）线程组成，他们共享进程资源，同一个人进程可以有多个线程并发，一个线程可以创建和撤销另一个线程；

进程有自己的独立地址空间，线程只有自己的堆栈和局部变量；

进程切换需要很大的开销，线程开销相对较小。

协程避免了无意义的调度，由此可以提高性能，但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多CPU的能力。

55.java的反射机制

反射的作用于原理。

反射是指程序可以访问、检测和修改其自身状态或行为的一种能力。

56.数据库索引的实现，MySQL主从结构

数据库的DDL、DML（数据操纵语言）

DDL（数据库定义语句）是SQL语言的四大功能之一。

用于定义数据库的三级结构，包括外模式、概念模式、内模式及其相互之间的映像，定义数据的完整性、安全控制等约束

DDL不需要commit.

CREATE

ALTER

DROP

TRUNCATE

COMMENT

RENAME

DML分成交互型DML和嵌入型DML两类。

依据语言的级别，DML又可分成过程性DML和非过程性DML两种。

需要commit.

SELECT

INSERT

UPDATE

DELETE

MERGE

CALL

EXPLAIN PLAN

LOCK TABLE

DCL（数据库控制语言）

GRANT 授权

REVOKE 取消授权

TCL（Transaction Control Language）事务控制语言

SAVEPOINT 设置保存点

ROLLBACK 回滚

SET TRANSACTION

57.StringBuilder内部实现

58.linux相关：查看线程命令，用户态和内核态

内核态: CPU可以访问内存所有数据, 包括外围设备, 例如硬盘, 网卡. CPU也可以将自己从一个程序切换到另一个程序

用户态: 只能受限的访问内存, 且不允许访问外围设备. 占用CPU的能力被剥夺, CPU资源可以被其他程序获取

用户态程序将一些数据值放在寄存器中, 或者使用参数创建一个堆栈(stack frame), 以此表明需要操作系统提供的服务.

用户态程序执行陷阱指令

CPU切换到内核态, 并跳到位于内存指定位置的指令, 这些指令是操作系统的一部分, 他们具有内存保护, 不可被用户态程序访问

这些指令称之为陷阱(trap)或者系统调用处理器(system call handler). 他们会读取程序放入内存的数据参数, 并执行程序请求的服务

系统调用完成后, 操作系统会重置CPU为用户态并返回系统调用的结

用户态切换到内核态的3种方式：a. 系统调用；b. 异常；c. 外围设备的中断。

59.Runnable传入Thread构造方法，算是什么设计模式。

代理设计模式。

Runnable target = new MyTarget();// 目标角色

Thread proxy = new Thread(target);// 代理角色

proxy.start();

61.系统降级方案。

在开发高并发系统时有三把利器用来保护系统：缓存、降级和限流。

62.socket客户端和服务端

http://blog.csdn.net/zhangyiacm/article/details/49488721

private void handleSocket() throws Exception {

Reader reader = new InputStreamReader(socket.getInputStream());

char chars[] = new char[64];

int len;

StringBuilder sb = new StringBuilder();

String temp;

int index;

while ((len=reader.read(chars)) != -1) {

temp = new String(chars, 0, len);

if ((index = temp.indexOf("eof")) != -1) {//遇到eof时就结束接收

sb.append(temp.substring(0, index));

break;

}

sb.append(temp);

}

System.out.println("from client: " + sb);

//读完后写一句

Writer writer = new OutputStreamWriter(socket.getOutputStream());

writer.write("Hello Client.");

writer.flush();

writer.close();

reader.close();

socket.close();

}

}

----------------------------------------------------------------更新---------------------------------------------

1.数据库索引：

数据库索引；（高频）

CREATE CLUSTER INDEX id

索引的实现通常使用B树及其变种B+树。

优点：加快查询速度；

缺点：占用空间，维护起来比较耗时；

建议使用索引的地方：主键，外键；需要排序或者根据范围来检索的列；经常需要搜索的列；WHERE子句中的列。

不创建索引的列：查询很少用到的列；只有很少数据值的列；修改性远远大于检索性的列。

索引类型有：主键索引，唯一索引，聚集索引。

将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往 往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要

读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返 回，程序继续运行。

索引类型（全文索引），底层实现（B+树），什么情况下索引会失效

索引是对数据库表中一列或多列的值进行排序的一种结构。

唯一索引、主键索引和聚集索引。

MySQL目前主要有以下几种索引类型：FULLTEXT，HASH，BTREE，RTREE。

FULLTEXT，即为全文索引，目前只有MyISAM引擎支持。其可以在CREATE TABLE ，ALTER TABLE ，CREATE INDEX 使用，不过目前只有 CHAR、VARCHAR ，TEXT 列上可以创建全文索引。

HASH，hash很适合做索引，为某一列或几列建立hash索引，就会利用这一列或几列的值通过一定的算法计算出一个hash值，对应一行或几行数据。

以下几种情况，将导致索引失效：

如果条件中有or，即使其中有条件带索引也不会使用(这也是为什么尽量少用or的原因)；

对于多列索引，不是使用的第一部分，则不会使用索引

3.like查询是以%开头

4.如果列类型是字符串，那一定要在条件中将数据使用引号引用起来,否则不使用索引

5.如果mysql估计使用全表扫描要比使用索引快,则不使用索引

数据库优化（索引、存储引擎、sql优化、视图）

1.适当建立索引：对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引，但索引并不是越多越好：

（1）应尽量避免在 where 子句中对字段进行 null 值判断，应尽量避免在 where 子句中使用 != 或 <> 操作符，以及like "%aa"，使用参数， or （如果一个字段有索引，一个字段没有索引），in 和 not in 也要慎用，应尽量避免在where子句中对字段进行函数操作，不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

2.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。

3.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

4.select count(\*) from table；这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

5.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。

6.选择适当的字段类型，特别是主键：尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型。

索引的实现通常使用B树及其变种B+树。

----------------------------------------------------------------------------------------------------------------------------------------

<http://blog.csdn.net/d12345678a/article/details/53955348>。

java语言高级特性及框架

1，重要的几个集合的原理，如HashMap，HashTable,ConcurrentHashMap,arrayList,LinkedList,TreeMap

1）hashmap和hashtable，concurrentHashMap实现原理，区别和扩容规则

1:线程安全性和存取速度，HasMap线程不安全，存取速度快；HashTable线程安全，但存取速度很慢，每次存取都加锁；ConcurrentHashMap是锁分段技术，既达到了线程安全，又兼顾存取速度。

2：初始化和扩容，HashMap初始化是16，负载因子为0.75，以原容量2倍的方式扩容；HashTable初始化容量为11，负载因子为0.75，以原容量\*2+1的方式扩容；ConcurrentHashMap初始化容量是16，负载因子0.75，

2）vector,arrayList和linkedList实现原理和区别，以及扩容规则

1:arrayList和Vector都是基于数组实现的，LinkedList是基于链表实现的。数组方便于查找，链表方便于插入和删除。

2：Vector线程安全，但操作速度慢。LinkedList和ArrayList线程不安全，操作速度快。

3：ArrayList可以插入重复值且有序，HashSet不能插入重复值且无序，是使用HashMap的keySet来实现的，value用了一个new Object()这种类型。

4：扩容ArrayList以1.5倍的方式进行扩容，Vector以2倍的形式扩容。LinkedList无需扩容，HashSet根据HashMap的扩容规则。

5：

3）TreeMap用什么实现的，用来作为有序的Map和LinkedList孰优孰劣

TreeMap是用红黑树来实现的，用来排序比LinkedList要好，无论是查找还是排序，时间复杂度较低。

2，JVM原理，GC，内存结构，类加载器，多态的实现，调优和监控，反射

1）什么对象会被回收，垃圾回收的算法，Serial，ParNew，Parallel Scavenge，CMS，G1等收集器的特点和用途

1:那些内存需要被回收？

答：通过GC Roots进行可达性分析，如果不可达则证明对象需要被回收。GC Roots包括静态变量（引用）、常量、虚拟机栈、本地方法栈

2：有哪些常用的垃圾回收算法？

标记-清除算法、标记-整理算法、分代收集算法

3：引用的四种类型

强引用（不会被收集）、软引用（内存不够被收集）、弱引用（第一次就被收集）、幽灵引用（仅仅发送通知）

4：安全点和安全区域

安全点：当到达安全点时线程才暂停（进行GC），安全点一般是方法调用、循环跳转、异常跳转等。HotSpot生成一个轮询指令（如test %eax 0x160100)，当需要暂停线程时，将地址0x160100设为不可读，这样线程就会陷入异常，在异常处理中等待。

安全区域：是一系列代码，这段代码不会出现引用关系发生变化。线程执行到这段代码时，会标识自己进入安全区域。离开安全区域时会判断GC roots 是否完成。为了解决不在执行状态（非Running）的线程怎么样暂停问题。

4：并发和并行的区别

并发是指多条线程在CPU时间片轮转下交替运行，从外观看起来类似于同时执行。

并行是指多个线程在多个CPU上同时执行，需要多个处理器支持。

5：各垃圾收集器的特点和用途

Serial收集器：单线程回收，对于单CPU不会带来线程上下文切换的开销，只做新生代回收，简单而高效。对于几十乃至一两百兆的新生代，可以选用。

ParNew收集器：Serial的多线程版本，新生代收集器，CPU越多效率越高。

Parallel Scavenge收集器：新生代收集器，多线程并行执行，吞吐量优先，可以控制GC停顿时间和吞吐量大小，可以开启自适应调解策略。

CMS收集器：老年代收集器，并发收集低停顿。初始标记、并发标记、重新标记、并发清除。1,3花费时间段，2,4花费时间长但可以与用户

线程一起运行。缺点是：1,对CPU资源敏感；2,并发清理产生很多浮动垃圾；3,基于标记清除算法会产生大量空间碎片。

G1收集器：

2）堆内存区域，Eden区，Survivor1，Survivor2，Old，Permanent(方法区和常量池），本地方法栈，虚拟机栈，程序计数器

1：JVM内存结构可分为堆区、java虚拟机栈、本地方法栈、方法区、程序计数器（PC）。

2：堆区可分为年轻代（Young）和老年代（Tenured)，年轻代可分为Eden区、Survivor1、Survivor2三个区。

3：永久代包含方法区和运行时常量池，用来存放类加载的信息，可以被GC进行回收，当常量池内容过多，或者类加载过多时，可能OOM。

4：本地内存，基于通道（channel）和缓冲区（Buffer）的I/O方式，可以使用Native函数库直接分配堆外内存，然后通过一个directByteBuffer对象作为这块内存的引用进行操作，可以避免Java堆和Native堆来回复制数据。

5：程序计数器，执行java方法时当前线程所执行字节码的行号，线程独有。执行Native方法是该区域为空（undefined）。

6: 虚拟机栈，运行java字节码。本地方法栈运行本地方法。

3）虚拟机类加载父委托机制，如何实现，这么实现的好处是什么。类加载过程

1：父委托机制是指，

4）多态的实现，重载和重写的区别，静态分派，动态分派，静态解析，动态解析，方法的调用过程

5）jstatk，jstat,jps,jmap,jconsole，visualVM，

3，多线程相关，线程的实现，volatile，final，threadLocal(原理和举例)，AQS，CAS，线程池，CountDownLatch，CyclicBarrier，Semaphore实现原理，阻塞队列

1）线程的两种实现方式，那种更好，为什么？

2）volatile两种作用，内存语义，final的作用。

3）threadLocal实现原理，举例说明（spring中用到过）

4）AQS实现原理（同步队列，获取和释放同步状态），ReentrantLock(公平锁和非公平锁)，读写锁（实现原理），与sychronized有什么异同

5）@@CAS如何实现同步（配合volatile），如何解决ABA问题

6）@@线程池和阻塞队列的实现原理，为什么使用线程池更好些

7）CountDownLatch，CyclicBarrier，Semaphore实现原理，还有没有别的实现方式

4，@@IO，BIO，NIO,AIO实现原理,会使用mina

1）BIO作为阻塞式IO如何进行数据传输的（结合OS读写缓冲区的阻塞说明）

2）NIO的实现原理，为什么不阻塞？（结合轮训和上下文切换说明）

3）Mina和Netty的原理

5，字符串相关的问题，存储，使用，要兼顾JDK6和JDK7以后的版本有什么不同之处

1）字符串存储方式，string，StringBuffer和Stringbuilder的区别，频繁使用“+”作为连接符的坏处

2）==，equals,intern()判断相等的时候问题

3）字符串的常量池

6,三大框架，SSI

1）@@SpringMVC架构和实现原理

2）Spring的IOC和AOP原理（动态代理）

3）@@mybatis实现原理。

4）@@spring和mybatis的缓存实现

Java 编程

**1.唯一索引与主键索引的比较**

唯一索引

唯一索引不允许两行具有相同的索引值。

如果现有数据中存在重复的键值，则大多数数据库都不允许将新创建的唯一索引与表一起保存。当新数据将使表中的键值重复时，数据库也拒绝接受此数据。例如，如果在 employee 表中的职员姓氏(lname) 列上创建了唯一索引，则所有职员不能同姓。

主键索引

主键索引是唯一索引的特殊类型。

数据库表通常有一列或列组合，其值用来唯一标识表中的每一行。该列称为表的主键。

在数据库关系图中为表定义一个主键将自动创建主键索引，主键索引是唯一索引的特殊类型。主键索引要求主键中的每个值是唯一的。当在查询中使用主键索引时，它还允许快速访问数据。

它们的一些比较：

(1)对于主健/unique constraint ， oracle/sql server/mysql等都会自动建立唯一索引；

(2)主键不一定只包含一个字段，所以如果你在主键的其中一个字段建唯一索引还是必要的；

(3)主健可作外健，唯一索引不可；

(4)主健不可为空，唯一索引可；

(5)主健也可是多个字段的组合；

(6)主键与唯一索引不同的是：

a.有not null属性；

b.每个表只能有一个。

**2.java**

1)HashMap HashTable的区别

2)map的遍历

3）HashMap如何实现

4）垃圾回收机制

5）Throwable上还有基类么 Object

6）List容器中有100个元素，一个个删除

循环删除List数组容器里面元素

定义一个List<T>数组，然后需要循环一下，删除其中一个元素，但会遇到类似问题：

java.util.ConcurrentModificationException

怎么办，实用主义者会把List数组转换成Iterator进行迭代删除，一点问题都没有：

public static void main(String[] args) {

List<Student> students = pareStudents();

System.out.println("size : " + students.size());

System.out.println(students);

System.out.println("delete Student with name xiaoi");

Iterator<Student>stuIter = students.iterator();

while(stuIter.hasNext()){

Student stu = stuIter.next();

if(stu.getName().equals("xiaoi")){

stuIter.remove();

}

}

System.out.println("size : " + students.size());

System.out.println(students);

}

Iterator进行循环操作，然后删除，是很安全的。可以解决一些问题。但不要再使用Iterator的时候使用students.remove(stu)，一样会给出java.util.ConcurrentModificationException 错误。

另一个方法就是重新构建一个List,把原先的students 数组作为参数传递进去即可，虽然生成了一个全新的数组，但都是Student对象的引用，多个引用的地址数组而已。

|  |
| --- |
| public static void main(String[] args) {  List<Student> students = pareStudents();    System.out.println("size : " + students.size());  System.out.println(students);    System.out.println("delete Student with name xiaoi");  for(Student s : new ArrayList<Student>(students)){  if(s.getName().equals("xiaoi")){  students.remove(s);  }  }    System.out.println("size : " + students.size());  System.out.println(students);  } |

7）jvm

8）jvm性能调优

3.多线程

1）线程间共享数据的方式有哪几种，各自的优缺点

2）ThreadLocal

3) 线程池具体是通过哪个类和哪些方法实现的，怎么实现的

4）多线程之间是如何通信的

4.设计模式

1）单例模式

2）动态代理

5.Spring

1)Ioc和AOP的理解

2）源码

3）@@Spring MVC页面渲染的几种方式

4）AOP基于什么设计模式实现的？具体说下cglib代理和jdk代理的区别，他们是怎么实现动态代理的，核心类和核心方法是什么

5）spring bean的几种状态

在Spring的Bean配置中，存在这样两种情况：

<bean id="testManager" class="com.sw.TestManagerImpl" scope="singleton" />

<bean id="testManager" class="com.sw.TestManagerImpl" scope="prototype" />

当然，scope的值不止这两种，还包括了request,session 等。但用的最多的还是singleton单态，prototype多态。

6）spring的缓存优化是怎么做的？如何清缓存，缓存哪里用到了，用他做什么？

~~7) hibernate的二级缓存~~

~~8）spring如何接手Hibernate的事务~~

9）spring的缓存优化，tomcat在哪设置缓存

10）使用spring初始化需要加载的东西，bean.dispatcherServlet,加载Html,spring的配置文件

6.NIO

7.redis

1)持久化

2）如果redis缓存宕掉了怎么办

8.项目

1）用哪些技术解决哪些难题？

9.

1）数据库优化方面的经验

2）搜索引擎的区别

3)oracle,mysql的区别

10.linux

1)过滤日志中的ip，并统计其个数

2）shell前100ip

3）linux性能优化

4）两个服务器之间拷贝文件的命令 scp

5)查出张三所有日志文件，并将其归档，求总数和

11、如果数据存的过程中突然出错怎么办？

12.数据量多大，你们做的系统如何保证数据安全

13.sql:求学生的总分数

14.如果系统已经部署上了，此时若出现数据出错怎么办？

15.如果死锁怎么查看？有这么多对象，先过滤哪些对象

16.多台不同的机器（每台机器上部署相同的Java执行程序），同时读写同一个数据库中的同一张表，如何避免冲突？

1个MySQL数据库，其中有一张资源表（假设有1000条记录，每条记录代表1个资源，在这张表中，有一个Status字段，Status=0表示“待处理”，Status=1表示“正在处理”，Status=2表示“处理完成”），现在有5台机器，每台机器上部署相同的执行程序，每台机器做的事情就是 ：

① 到这个数据库中，获取1条Status=0的记录，然后将Status置为1，

② 然后根据这条记录中提供的资源信息进行处理，

③ 处理完了之后，修改数据库中的这条记录的Status为2，表示“处理完毕”；

④ 重复①到③步，直到在这张表中找不到Status=0的记录为止。

请问如何实现？给设计方案也行。

（另外，弱弱地问一句：我的这个情况可以用Java多线程么，在多个不同机器之间进行互斥访问数据库？）

解决：

可以把java应用拆分出两种角色，一种是处理者（N个），一种是资源管理者（1个）。

处理者：专门做“ 根据这条记录中提供的资源信息进行处理”的工作。从资源管理者处取status为0的记录，处理之，处理完通知资源管理者。

资源管理者：管理一个消息队列，队列里是各个处理者发来的请求。一般分两种请求，取资源和完成某资源的处理。根据请求对数据表做相应操作，管理资源的状态。

这种情况下，因为只有资源管理者唯一接触数据库，不需要再考虑神马互斥访问的问题，数据库资源表也不需要加锁。处理者和资源管理者之间以消息通信。高效也易实现。（隔离可变性设计）

17.分布式缓存

18注解写个table

19.什么是平衡二叉树，如何保持平衡的

1. junit用法，before,beforeClass,after, afterClass的执行顺序

2. 分布式锁

3. nginx的请求转发算法，如何配置根据权重转发

4. 用hashmap实现redis有什么问题（死锁，死循环，可用ConcurrentHashmap）

5. 线程的状态

5. 线程的阻塞的方式

6. sleep和wait的区别

7. hashmap的底层实现

8. 一万个人抢100个红包，如何实现（不用队列），如何保证2个人不能抢到同一个红包，可用分布式锁

9. java内存模型，垃圾回收机制，不可达算法

10. 两个Integer的引用对象传给一个swap方法在方法内部交换引用，返回后，两个引用的值是否会发现变化

11. aop的底层实现，动态代理是如何动态，假如有100个对象，如何动态的为这100个对象代理

12. 是否用过maven install。 maven test。git（make install是安装本地jar包）

13. tomcat的各种配置，如何配置docBase

14. spring的bean配置的几种方式

15. web.xml的配置

16. spring的监听器。

~~17. zookeeper的实现机制，有缓存，如何存储注册服务的~~

18. IO会阻塞吗？readLine是不是阻塞的

19. 用过spring的线程池还是java的线程池？

20. 字符串的格式化方法 （20，21这两个问题问的太低级了）

21. 时间的格式化方法

22. 定时器用什么做的

23. 线程如何退出结束

24. java有哪些锁？乐观锁 悲观锁 synchronized 可重入锁 读写锁,用过reentrantlock吗？reentrantlock与synmchronized的区别

25. ThreadLocal的使用场景

26. java的内存模型，垃圾回收机制

27. 为什么线程执行要调用start而不是直接run（直接run，跟普通方法没什么区别，先调start，run才会作为一个线程方法运行）

28. qmq消息的实现机制(qmq是去哪儿网自己封装的消息队列)

29. 遍历hashmap的三种方式

30. jvm的一些命令

@@Rabbitmq应用:

第一，你的业务需要用到消息队列

第二，消息需要持久化，重启服务能继续执行未完成的任务

第三，可能有大量的消息需要处理， rabbitmq 的性能能够满足

有的任务占用本机资源，而且不关心返回结果，这些扔进消息队列就好了

比如说一个任务需要跑很久 不能当时返回 然后就可以使用这种消息队列

因为消息服务的的通信协议和规则很随意而且是异步的，所以其实可以做很多很多事。

例如：

注册用户时候，发送激活邮件。

监控应用中抛出的异常，邮件通知管理员。也就是增加监控功能，且一个消息服务可以监控无数个应用。

异构型架构系统间的各服务通信，因为消息服务于平台或者语言无关，典型的应用在 SOA 分布式体系的各个服务之间，可以做到，我依赖你，但是我可以不在乎你是否在运行，的关系。

例如，微信的抢红包，淘宝的订单，铁道部的购票等，如果不丢给队列排队处理，突然性的高并发会有让应用或者数据库瘫痪的风险。

但是上述并不是只有消息服务能做，也没有消息服务能做，但是别的手段做不到的事情。

其实很简单，一种应用是：因为很多任务是需要花费大量时间的，比如你设计一套系统，要分别发送 100 封邮件给不同的人。在用户点击发送按钮之后，如果在完成这 100 封邮件后才反馈给用户发送完成，那用户体验简直崩溃。因此此时你将这个任务添加到 rabbitMQ 队列里，前台立即返回给用户说成功，后台用 python 慢慢发送就好

**RabbitMQ的深入理解和最简单的用途说明**

RabbitMQ的设计理念是.只要有接收消息的队列. 邮件就会存放到队列里. 直到订阅人取走. . 如果没有可以接收这个消息的消息队列. 默认是抛弃这个消息的..

* 如何在新系统中用RabbitMQ.

系统设计:.

* 要满足未来的业务需求的不断变化和增加. 也就是可扩展性.
* 要满足性能的可伸缩性. 也就是可集群性…通过增加机器能处理更多请求
* 解耦合.

如不解耦合, 未来业务增加或变更时你还在修改3年前写的代码.试问你有多大的把握保证升级好系统不出问题? 如何可写新的代码而不用修改老代码所带来的好处谁都知道…

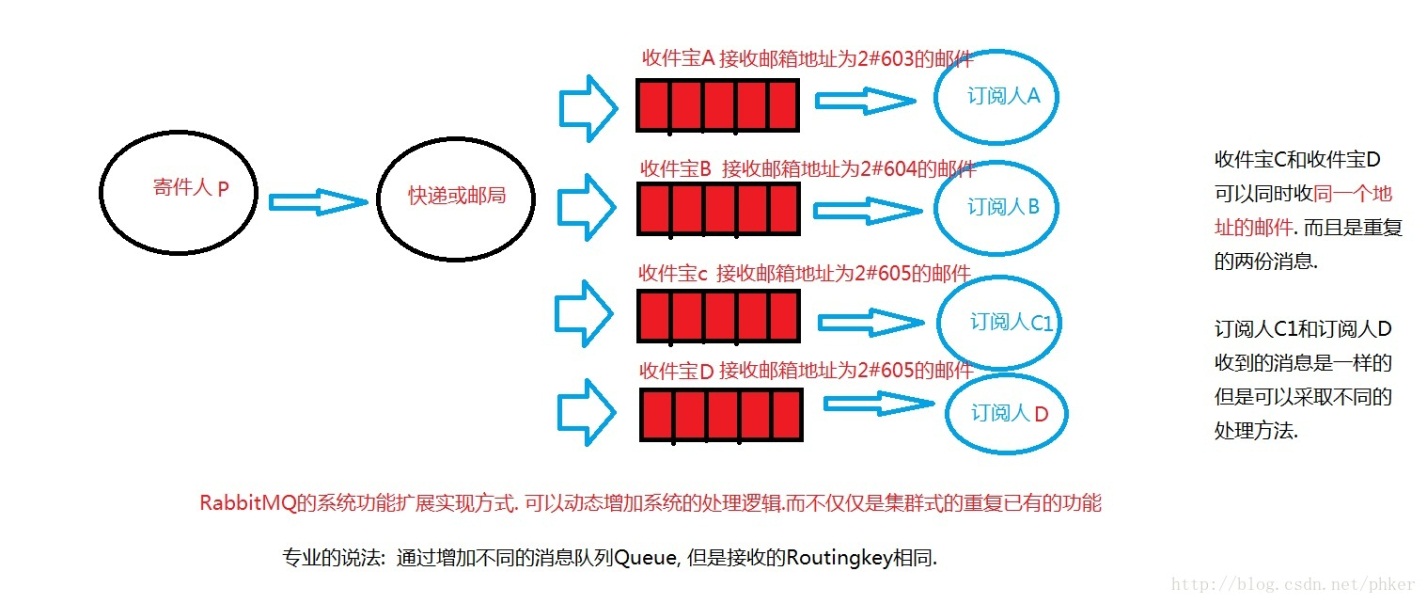
以前无法做到. 自从有了MQ. 这些都可同时做到了.

以前的设计理念是把系统看作一个人,按照工作的指令从上到下的执行.

现在要建立的概念是, 把系统的各个功能看作不同的人. 人与人之间的沟通通过消息进行交流传递信息…

有了MQ后把一个人的事情分给了不同的人, 分工合作所带来的好处是专业化, 并行化. 当然也引入了一些麻烦,性能开销多一些, 工作任务的完整性不能立即得到反馈.幸好可通过最终一致性.来解决这个麻烦的问题…

* RabbitMQ如何支持可扩展性.



如上图, 寄件人P是系统的一个功能模块. 用来发送消息. 一般是在某些重要的业务状态变更时发送消息. 例如: 新订单产生时, 订单已打包时, 订单已出库时, 订单已发出时.

那么当事件新订单产生时, 需把这个信息告诉谁呢? 给财务? 还是给仓库发货?

重点是. 当事件产生时. 根本不关心. 该投递给谁.

只要把重要信息投到这个MQ系统即可. 其它人你该干嘛干嘛. 反正我的任务完成了. (有没有甩手掌柜的感觉..)

只要告诉系统,我的事件属于那一类.

例如: “某某省.某某市.某某公司.产生新订单”

那么这个地址就属于 投递地址.. 至于这个地址具体投到哪个邮箱那是邮局的事情.

当然还有一些具体的订单内容也属于要告诉系统的内容.

下一个问题来了, 邮局怎么知道 这个消息应该投递给谁?

参考我们现实世界中的邮寄系统.是默认的省市县这么投递的. 这是固定思维.

但是我们的MQ系统中不是这样的. 是先有收件人的邮箱. (队列Queue). MQ才能投递. 否则就丢弃这个信息…

所以MQ系统应该先有收件人的邮箱 Queue 也就是队列. 才能接收到信息.

再有邮局

再有发信息的人.

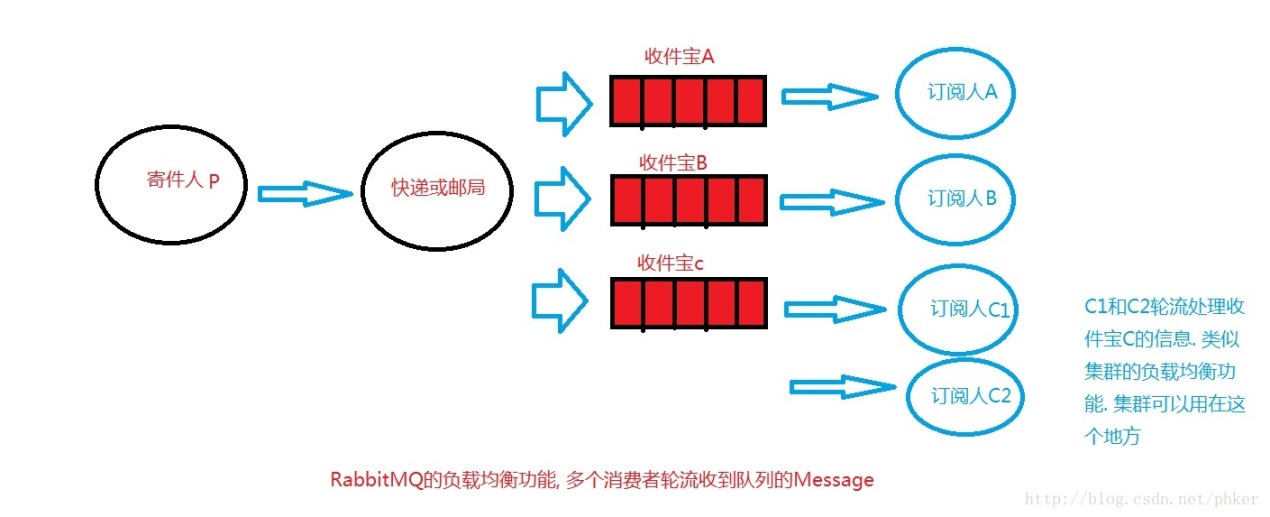
RabbitMQ能实现系统扩展的一个重要功能在于, 可两个邮箱收同一个地址的信.

RabbitMQ 可 两个队列Queue订阅同个RoutingKey的信息..

RabbitMQ投递时,会把一份信息,投递到多个队列邮箱中Queue…

这是系统可扩展性的基础.

* RabbitMQ如何满足性能的可伸缩性. 也就是可集群性



性能扩展的关键点就在于 订阅人C1, C2 轮流收到邮箱队列里的信息, 订阅人C1和订阅人C2收到的信息内容不同, 但都属于同一类….

所以. 订阅人C1和C2是干同一种工作的客户端.用来提高处理能力.

* 如果订阅人的down机了. 信息会丢失吗?

不会. 只要有邮箱(队列Queue)存在.信息就一直存在, 除非订阅人去取走.

* 如果订阅人一直down机, 邮箱队列能存多少信息?会不会爆掉?

理论上和实际上都是有上限的不可能无限多. 具体多少看硬盘吧.

这里只打算使用topic这种模式. 更灵活

不要在业务程序中用代码定义创建 邮局 ExChange. 和邮箱Queue队列 这属于系统设计者要构架的事情. 要有专门独立的程序和规则去创建. 这样可统一管理事件类型.避免过多的乱七八糟的RoutingKey混乱.

Exchange: 邮局的概念 等同于中国邮政和顺丰快递、

routingkey: 就是邮件地址的概念.

queue: 就是邮箱

producer： 消息生产者，就是投递消息的程序。

consumer：消息消费者，就是接受消息的程序。

channel：消息通道，在客户端的每个连接里，可建立多个channel，每个channel代表一个会话任务。

**2.@@@redis mysql性能**

* 测试报告

通过对redis的set、get命令测试观察，redis的读写性能在单线程下可以达到每秒2W左右

通过对mysql的select和insert、delete语句测试，mysql的读性能可达到6000每秒，写性能可到达3000每秒，读性能基本是写性能的2倍。

* redis性能分析

redis的读写快，主要体现在redis的设计上，redis是单进程单线程模式，采用队列模式将并发访问转为串行访问,不但没影响它的性能，反而因这样设计而变得更快速，要结合redis的使用场景。

在这种简单的设计模式下，redis本身并没有竞争没有锁的概念。

估计作者也是结合了redis的使用场景才这么干的，因为毕竟redis主要操作内存数据，而且定位数据时采用hash定位，它在写入和查询时所以基本不消耗什么性能，另外单个命令基本瞬间完成，应该不会出现因采用串行化执行而出现阻塞的情况，即应该不会出现某条命令执行时间过长而阻塞后面的命令执行的情况，这也是为什么不建议在Redis的一个key上存取太长数据的原因。这种简单的设计模式，不仅提高了redis的性能，而且在实现上代码也会少很多，实现起来也会变得简单起来。

**3.@@Java自定义注解和运行时靠反射获取注解**

java自定义注解

Java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。注解不会也不能影响代码的实际逻辑，仅起到辅助性作用。在 java.lang.annotation 包中。

**1、元注解**

注解的注解。包括 @Retention @Target @Document @Inherited四种。

1.1、@Retention: 定义注解的保留策略

@Retention(RetentionPolicy.SOURCE) //注解仅存在于源码中，在class字节码文件中不包含

@Retention(RetentionPolicy.CLASS) // 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得。编译器将把注释记录在类文件中，但在运行时 VM 不需要保留注释。

@Retention(RetentionPolicy.RUNTIME) // 注解会在class字节码文件中存在，在运行时可以通过反射获取到

@Target({ElementType.FIELD,ElementType.METHOD})//定义注解的作用目标\*\*作用范围字段、枚举的常量/方法

@Documented//该注解将被包含在javadoc中

**4.上**

**4.1数据库锁:**

面向对象基础:

多线程处理:

select count(\*) as linesize from SubType where SUPERTYPEID= ?

linesize就是行数

violate集合:Hashmap实现原理(2) ，是线程安全的吗。不是怎么做？

Linklist和arraylist区别

Sql

单例

Jvm：垃圾收集器类型、算法；参数作用 ；调优心得

jvm结构图

数据库:索引选取 实现原理

@@@Springmvc处理json的方法

@@Spring核心架构，画图

@@Maven命令:profile build怎么用

3轮:1轮基础 2轮项目 遇到的问题 算法 3轮群面

4轮:1轮基本技术 2.2轮小组leader 4轮hr

项目:表的结构设计

索引 索引优化

Yingjiesheng

Equals、hashcode的实现方法

Spring的类的初始化过程

hashmap的实现原理（链表的实现方式）

快速排序【3】

二分查找

二叉树的前序中序遍历-递归和非递归法

Student和course表，找出os课程分数排名前三的同学的名字

数组，构建平衡二叉树

数组，找到重复数字O(1)空间 O(N)时间

八匹马算法

Xiecheng

笔试

Java的优化 sql的优化 jvm的原理和机制

|  |
| --- |
| ~~hashing的概念~~  ~~HashMap中解决碰撞的方法~~  equals()和hashCode()的应用，以及在HashMap中的重要性  不可变对象的好处  HashMap多线程的条件竞争  重新调整HashMap的大小 |

6.comparable接口和comparator接口实现比较的区别和用法，Arrays静态类如下实现排序的。

7.问快排的优化，怎么选基准，我就说随机化，防止退化。谈谈快排，于是3种快排4种优化方式，以及partition函数的应用。

http://blog.csdn.net/hacker00011000/article/details/52176100

3种快排，是指3中基数的选择方法：固定位置、随机、三数取中；

4种优化：（1）当待排序序列的长度分割到一定大小后，使用插入排序；

（2）在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割；

（3）优化递归操作；

（4）使用并行或多线程处理子序列；

随机化可以解决当数组有序或者部分有序时的退化，但是当数组元素全部重复的时候，时间复杂度依然很高；

三数取中：对待排序序列中low、mid、high三个位置上数据进行排序，取他们中间的那个数据作为枢轴，并用0下标元素存储枢轴。这个方法还是无法解决重复数组的问题。

在一次分割结束后，可以把与Key相等的元素聚在一起，继续下次分割时，不用再对与key相等元素分割，可以明显提高重复数组的效率。

STL中的Sort函数：当数据量较大时采用快速排序，分段递归。一旦分段后的数据量小于某个阀值，为避免递归调用带来过大的额外负荷，便会改用插入排序。而如果递归层次过深，有出现最坏情况的倾向，还会改用堆排序。STL采用的做法称为median-of-three，即取整个序列的首、尾、中央三个地方的元素，以其中值作为枢轴。

3、两个线程如何同时监听一个端口。

多个线程可以监听同一个端口，但我们通常不这样做。

多个进程也可以同时监听一个端口，比如nginx

{

"className":"com.kedacom.tz.birdie.core.vo.RhzhResourceVo",

"valueString":{

"resourceStatue":1,

"deptName":"",

"alias":"单兵137109","netPing":1,"resourceE164":"0512110000451","domainid":"51564db1320a444eba952d6744c79ee4","organizationId":"","resourceType":4,"resourceIp":"172.16.137.109","id":0,"resourceId":"16859cb0-6771-11e7-8888-002564c57c64","subType":"","opertType":"","deptId":"85f7ece0-4289-11e7-a644-002564c57c64","resourceName":"单兵109","puid":"8aba60992f7b418dbcb7dc77cf35c8ba@kedacom

"}

}