# Java面试知识点总结

[**www.xwder.com**](https://d.docs.live.net/4a9175e00281e9e4/Java开发/www.xwder.com)

# ****一、基础篇****

## ****1.1、Java基础****

### 1.1.1 九种基本数据类型的大小，以及他们的封装类

|  |  |  |  |
| --- | --- | --- | --- |
| 基本类型 | 大小(字节) | 默认值 | 封装类 |
| byte | 1 | (byte)0 | Byte |
| short | 2 | (short)0 | Short |
| int | 4 | 0 | Integer |
| long | 8 | 0L | Long |
| float | 4 | 0.0f | Float |
| double | 8 | 0.0d | Double |
| boolean | - | false | Boolean |
| char | 2 | \u0000(null) | Character |
| void | - | - | Void |

一个字节是8位，一个汉字编码两个字节是十六位。

为什么Java会这么做？

在java中使用基本类型来存储语言支持的基本数据类型，这里没有采用对象，而是使用了传统的面向过程语言所采用的基本类在型，主要是从性能方面来考虑的：因为即使最简单的数学计算，使用对象来处理也会引起一些开销，而这些开销对于数学计算本来是毫无必要的。但是在java中，泛型类包括预定义的集合，使用的参数都是对象类型，无法直接使用这些基本数据类型，所以java又提供了这些基本类型的包装器。

**基本数据类型与其对应的封装类由于本质的不同，具有一些区别：**

* 基本数据类型只能按值传递，而封装类按引用传递。
* 基本类型在堆栈中创建；而对于对象类型，对象在堆中创建，对象的引用在堆栈中创建。基本类型由于在堆栈中，效率会比较高，但是可能会存在内存泄漏的问题。
* 堆中分配空间所需的时间远大于从栈中分配存储空间,所以JAVA速度比C 慢

基本类型所占的存储空间是不变的：这种不变性也是java具有可移植性的原因之一。基本类型是放在栈中，直接存储值。所有数值类型都有正负号，没有无符号的数值类型。

注：

1.int是基本数据类型，Integer是int的封装类，是引用类型。int默认值是0，而Integer默认值是null，所以Integer能区分出0和null的情况。一旦java看到null，就知道这个引用还没有指向某个对象，再任何引用使用前，必须为其指定一个对象，否则会报错。

2.基本数据类型在声明时系统会自动给它分配空间，而引用类型声明时只是分配了引用空间，必须通过实例化开辟数据空间之后才可以赋值

**布尔类型包装器**：  Boolean 是封装布尔值的包装器。它定义了下面的的构造方法：

Boolean (boolean boolValue);//boolValue 的值必须是true或false；

Boolean (String boolString);//boolString 的值为“true”,那么创建的布尔对象为true，否则为false；

为了获得Booolean 对象中的布尔值，可以使用**booleanValue()**方法；

boolean占多少字节呢？

**虽然定义了boolean这种数据类型，但是只对它提供了非常有限的支持。在Java虚拟机中没有任何供boolean值专用的字节码指令，Java语言表达式所操作的boolean值，在编译之后都使用Java虚拟机中的int数据类型来代替，而boolean数组将会被编码成Java虚拟机的byte数组，每个元素boolean元素占8位”。这样我们可以得出boolean类型占了单独使用是4个字节，在数组中又是1个字节。使用int的原因是，对于当下32位的处理器（CPU）来说，一次处理数据是32位（这里不是指的是32/64位系统，而是指CPU硬件层面），具有高效存取的特点。**

**数字类型包装器：**

所有数字类型的包装器都是从抽象类Number 继承下来的。Number 声明了多种以不同的数字格式返回一个对象值的方法：

byte byteValue( );

int intValue();.........................

所有的数字类型包装器都提供了一个静态的方法，可以将字符串 转换为自己所封装的数据类型；例如：

static int parseInt(String s  );

<http://blog.csdn.net/oLaoHuBuChiRen1/article/details/51079738>

<http://blog.csdn.net/syc434432458/article/details/49964089>

### 1.1.2 Switch能否用string做参数？

参数可以是整数表达式，JDK1.5开始支持Enum类，JDK1.7开始支持String。在jdk 7 之前，switch 只能支持 byte、short、char、int 这几个基本数据类型和其对应的封装类型。switch后面的括号里面只能放int类型的值，**但由于byte，short，char类型，它们会 自动 转换为int类型（精精度小的向大的转化），所以它们也支持。**

**注意，对于精度比int大的类型，比如long、float，doulble，不会自动转换为int，如果想使用，就必须强转为int，如(int)float;**

当字符串不会频繁改变时可以用枚举来代替String：用Enum.parse。

|  |
| --- |
| public enum Weekend  {  Monday,  Tuesday,  Friday,  }  public void Test(string week)  {  Weekend weekend = (Weekend)Enum.Parse(typeof(Enum), week, false);//转换一下    switch (weekend)  {  case Weekend.Monday:  Console.WriteLine("Today is Monday");  break;  case Weekend.Tuesday:  Console.WriteLine("Today is Tuesday");  break;  case Weekend.Friday:  Console.WriteLine("Today is Friday");  break;  default:  break;  }  } |

注意：

1.整数表达式可以是int和Integer，而byte、short、char等可以转换成int，所以也可以用。但是long和String不能转换成int，所以不能用

2.String时不能传入null作为参数，同时case语句中使用的字符串也不能为null，因为底层是通过equals和hashCode来判断的

|  |
| --- |
| public void test(String str) {  switch(str) {  case "abc":  System.out.println("abc");  break;  case "def":  System.out.println("def");  break;  default:  System.out.println("default");  }  }  } |

反编译后的结果：

|  |
| --- |
| public class Test {  public void test(String str) {  int i = -1;  switch(str.hashCode()) {  case 96354: // "abc".hashCode()  if (str.equals("abc")) {  i = 0;  }  break;  case 99333: // "def".hashCode()  if (str.equals("def")) {  i = 1;  }  break;  default:  break;  }    switch(i) {  case 0:  System.out.println("abc");  break;  case 1:  System.out.println("def");  break;  default:  System.out.println("default");  }  }  } |

PS：hashcode返回的是int型  
可以发现：进行switch的实际是hash值，然后用equals进行安全检查（因为hash值可能会发生碰撞），所以性能不如枚举。switch支持String其实是一个语法糖，在编译后的字节码文件中都会被还原成原生类型，并在相应位置插入了强制转换代码，底层的JVM在switch上并没有修改。

当传入switch的是null时，在运行时对一个null调用hashCode()方法，会抛出NullPointerException；如果case写的是null，那么在编译时无法求出hashcode，就会编译报错。

### 1.1.3 equals与==的区别

<http://www.cnblogs.com/dolphin0520/p/3592500.html>

<http://blog.csdn.net/echosilly/article/details/51763543>

### 1.1.4 native

Native：本地的

#### 1.1.4.1 认识 native 即 JNI,Java Native Interface

凡是一种语言，都希望是纯。比如解决某一个方案都喜欢就单单这个语言来写即可。Java平台有个用户和本地C代码进行互操作的API，称为Java Native Interface (Java本地接口)。

#### 1.1.4.2 用 Java 调用 C 的“Hello，JNI”

我们需要按照下班方便的步骤进行：

1、创建一个Java类，里面包含着一个 native 的方法和加载库的方法 loadLibrary。HelloNative.java 代码如下：

|  |
| --- |
| **public** **class** HelloNative {  **static** {  System.*loadLibrary*("HelloNative");  }  **public** **static** **native** **void** sayHello();  @SuppressWarnings("static-access")  **public** **static** **void** main(String[] args) {  **new** HelloNative().*sayHello*();  }  } |

首先泥瓦匠让大家注意的是native方法，那个加载库的到后面也起作用。native 关键字告诉编译器（其实是JVM）调用的是该方法在外部定义，这里指的是C。如果大家直接运行这个代码， JVM会告之：“A Java Exception has occurred.”控制台输出如下：

Exception in thread "main" java.lang.UnsatisfiedLinkError: no HelloNative in java.library.path

    at java.lang.ClassLoader.loadLibrary(Unknown Source)

    at java.lang.Runtime.loadLibrary0(Unknown Source)

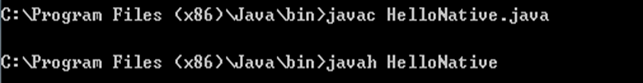
    at java.lang.System.loadLibrary(Unknown Source)

    at HelloNative.<clinit>(HelloNative.java:5)

这是程序使用它的时候，虚拟机说不知道如何找到sayHello。下面既可以手动写，自然泥瓦匠是用

2、运行javah，得到包含该方法的C声明头文件.h

泥瓦匠将HelloNative.java ，简单地 javac javah，如图



就得到了下面的 HelloNative.h文件 ：

/\* DO NOT EDIT THIS FILE - it is machine generated \*/

#include <jni.h>

/\* Header for class HelloNative \*/

#ifndef \_Included\_HelloNative

#define \_Included\_HelloNative

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*

 \* Class:     HelloNative

 \* Method:    sayHello

 \* Signature: ()V

 \*/

JNIEXPORT void JNICALL Java\_HelloNative\_sayHello

  (JNIEnv \*, jclass);

#ifdef \_\_cplusplus

}

#endif

#endif

jni.h 这个文件，在/%JAVA\_HOME%include

3、根据头文件，写C实现本地方法。

这里我们简单地实现这个sayHello方法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include "HelloNative.h"  #include <stdio.h>    JNIEXPORT void JNICALL Java\_HelloNative\_sayHello  {      printf("Hello，JNI");  } |

4、生成dll共享库，然后Java程序load库，调用即可。

在Windows上，MinGW GCC 运行如下

|  |  |
| --- | --- |
| 1 | gcc -m64  -Wl,--add-stdcall-alias -I"C:\Program Files\Java\jdk1.7.0\_71\include" -I"C:\Program Files\Java\jdk1.7.0\_71\include\include\win32" -shared -o HelloNative.dll HelloNative.c |

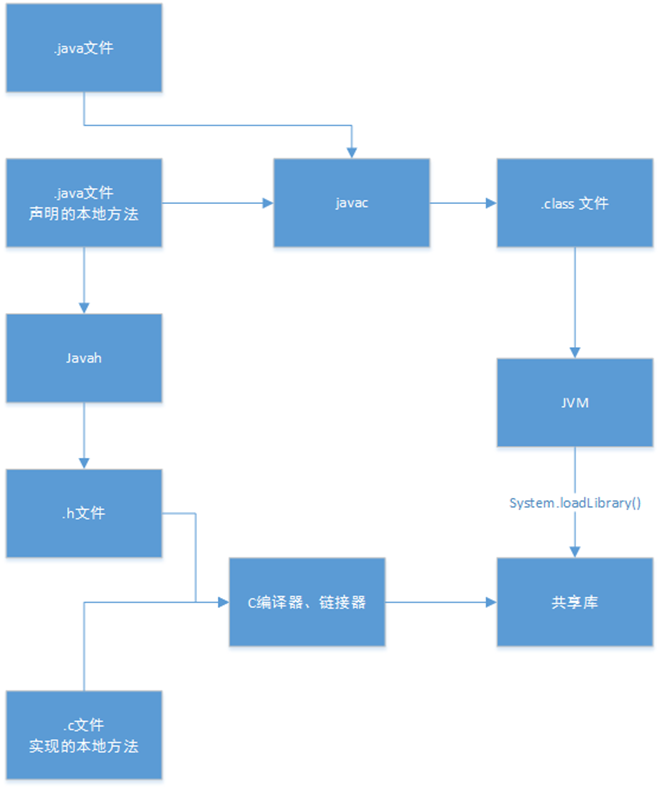
-m64表示生成dll库是64位的。然后运行 HelloNative：

|  |  |
| --- | --- |
| 1 | java HelloNative |

终于成功地可以看到控制台打印如下：

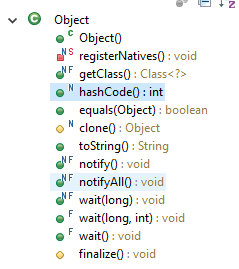
|  |  |
| --- | --- |
| 1 | Hello，JNI |

#### 1.1.4.3 JNI 调用 C 流程图



Writer：BYSocket（泥沙砖瓦浆木匠）  
<http://www.cnblogs.com/Alandre/p/4456719.html>

### 1.1.5 Object有哪些公用方法？



#### 1.1.5.1 clone方法

**protected** **native** Object clone() **throws** CloneNotSupportedException;

详细描述见： [java对象克隆以及深拷贝和浅拷贝](#_6._java对象克隆以及深拷贝和浅拷贝)

保护方法，实现对象的浅复制，只有实现了Cloneable接口才可以调用该方法，否则抛出CloneNotSupportedException异常。

主要是JAVA里除了8种基本类型传参数是值传递，其他的类对象传参数都是引用传递，我们有时候不希望在方法里将参数改变，这是就需要在类中复写clone方法。

#### 1.1.5.2 getClass方法

**public** **final** **native** Class<?> getClass();

final方法，获得运行时类型。

**一、**getClass方法：  
类型：public final Class<? extends Object> getClass()  
功能：返回**该对象的运行时类的java.lang.Class对象**（API上的解释）  
有方法类型可以知道，该方法只能由类的实例变量调用  
例子：

1. JButton b1 = **new** JButton("button1");
2. System.out.println(b1.getClass());

输出：  
 class javax.swing.JButton

class属性  
当你要获得一个**类的Class对象**时（作函数参数的时候），你不能调用getClass方法，那你只能用类名.class来达到效果  
例子： System.out.println(JButton.class);  、、

输出： class javax.swing.JButton

getName方法：  
类型：public String getName()  
功能：以String形式返回次Class对象所表示的实体名称  
例子：

1. JButton b1 = **new** JButton("button1");
2. System.out.println(b1.getName());

输出：  
javax.swing.JButton

可以发现用class属性和getClass返回的输出是一样的，用getName返回的比前面两种少了class和一个空格。

**.eclipse工具 可以按"."然后马上提示很多方法 供你选择那他如何知道"."了以后有哪些方法?他用的语法就是getClass().getMethods();**

二、

.class其实是在java运行时就加载进去的

getClass()是运行程序时动态加载的

**三、** 四种获取Class对象的方法 Java反射机制

<http://blog.csdn.net/Ghost_T/article/details/5811974>

#### 1.1.5.3 toString方法

**public** String toString() {

**return** getClass().getName() + "@" + Integer.*toHexString*(hashCode());

}

该方法用得比较多，一般子类都有覆盖。

#### 1.1.5.4 finalize方法

**protected** **void** finalize() **throws** Throwable { }

该方法用于释放资源。因为无法确定该方法什么时候被调用，很少使用。

#### 1.1.5.5 equals方法

java.lang.Object：

**public** **boolean** equals(Object obj) {

**return** (**this** == obj);

}

java.lang.String

|  |
| --- |
| **public** **boolean** equals(Object anObject) {  **if** (**this** == anObject) {  **return** **true**;  }  **if** (anObject **instanceof** String) {  String anotherString = (String)anObject;  **int** n = count;  **if** (n == anotherString.count) {  **char** v1[] = value;  **char** v2[] = anotherString.value;  **int** i = offset;  **int** j = anotherString.offset;  **while** (n-- != 0) {  **if** (v1[i++] != v2[j++])  **return** **false**;  }  **return** **true**;  }  }  **return** **false**;  } |

该方法是非常重要的一个方法。一般equals和==是不一样的，但是在Object中两者是一样的。子类一般都要重写这个方法。

#### 1.1.5.6 hashCode方法

**public** **native** **int** hashCode();

该方法用于哈希查找，可以减少在查找中使用equals的次数，重写了equals方法一般都要重写hashCode方法。这个方法在一些具有哈希功能的Collection中用到。

一般必须满足obj1.equals(obj2)==true。可以推出obj1.hash- Code()==obj2.hashCode()，但是hashCode相等不一定就满足equals。不过为了提高效率，应该尽量使上面两个条件接近等价。

如果不重写hashcode(),在HashSet中添加两个equals的对象，会将两个对象都加入进去。

#### 1.1.5.7 wait方法

wait方法就是使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait()方法一直等待，直到获得锁或者被中断。wait(long timeout)设定一个超时间隔，如果在规定时间内没有获得锁就返回。

调用该方法后当前线程进入睡眠状态，直到以下事件发生。

（1）其他线程调用了该对象的notify方法。

（2）其他线程调用了该对象的notifyAll方法。

（3）其他线程调用了interrupt中断该线程。

（4）时间间隔到了。

此时该线程就可以被调度了，如果是被中断的话就抛出一个InterruptedException异常。

#### 1.1.5.8 notify方法

该方法唤醒在该对象上等待的某个线程。

#### 1.1.5.9 notifyAll方法

该方法唤醒在该对象上等待的所有线程。

### 1.1.6 java对象克隆以及深拷贝和浅拷贝

#### 1.1.6.1 什么是"克隆"？

在实际编程过程中，我们常常要遇到这种情况：有一个对象A，在某一时刻A中已经包含了一些有效值，此时可能会需要一个和A完全相同新对象B，并且此后对B任何改动都不会影响到A中的值，也就是说，A与B是两个独立的对象，但B的初始值是由A对象确定的。在Java语言中，用简单的赋值语句是不能满足这种需求的。要满足这种需求虽然有很多途径，但实现clone（）方法是其中最简单，也是最高效的手段。   
 Java的所有类都默认继承java.lang.Object类，在java.lang.Object类中有一个方法clone()。JDK API的说明文档解释这个方法将返回Object对象的一个拷贝。要说明的有两点：

一是拷贝对象返回的是一个新对象，而不是一个引用。

二是拷贝对象与用 new操作符返回的新对象的区别就是这个拷贝已经包含了一些原来对象的信息，而不是对象的初始信息。

#### 1.1.6.2 怎样应用clone方法

一个很典型的调用clone()代码如下：

|  |
| --- |
| **class** CloneClass **implements** Cloneable {  **public** **int** aInt;  @Override  **protected** Object clone() {  CloneClass o = **null**;  **try** {  o = (CloneClass) **super**.clone();  } **catch** (CloneNotSupportedException e) {  e.printStackTrace();  }  **return** o;  }  } |

有三个值得注意的地方：

一是希望能实现clone功能的CloneClass类实现了Cloneable接口，这个接口属于java.lang 包，java.lang包已经被缺省的导入类中，所以不需要写成java.lang.Cloneable。

另一个值得请注意的是重载了clone()方 法。最后在clone()方法中调用了super.clone()，这也意味着无论clone类的继承结构是什么样的，**super.clone()直接或 间接调用了java.lang.Object类的clone()方法**。下面再详细的解释这几点。

第三点是最重要的，仔细 观察一下**Object类的clone()是一个native方法，native方法的效率一般来说都是远高于java中的非native方法**。这也解释了为什么要用Object中clone()方法而不是先new一个类，然后把原始对象中的信息赋到新对象中，虽然这也实现了clone功能。对于第二点，也要**观察Object类中的clone()还是一个protected属性的方法**。这也意味着如果要应用clone()方法，必须继承Object类，在 Java中所有的类是缺省继承Object类的，也就不用关心这点了。然后重写clone()方法。还有一点要考虑的是**为了让其它类能调用这个clone 类的clone()方法，重写之后要把clone()方法的属性设置为public。**

那么clone类**为什么还要实现 Cloneable接口呢**？稍微注意一下，Cloneable接口是不包含任何方法的！其实**这个接口仅仅是一个标志，而且这个标志也仅仅是针对 Object类中clone()方法的**，如果clone类没有实现Cloneable接口，并调用了Object的clone()方法（也就是调用了 super.Clone()方法），那么Object的clone()方法就会抛出CloneNotSupportedException异常。

#### 1.1.6.3 深拷贝与浅拷贝

浅拷贝是指拷贝对象时仅仅拷贝对象本身（包括对象中的基本变量），而不拷贝对象包含的引用指向的对象。深拷贝不仅拷贝对象本身，而且拷贝对象包含的引用指向的所有对象。举例来说更加清楚：对象A1中包含对B1的引用，B1中包含对C1的引用。浅拷贝A1得到A2，A2 中依然包含对B1的引用，B1中依然包含对C1的引用。深拷贝则是对浅拷贝的递归，深拷贝A1得到A2，A2中包含对B2（B1的copy）的引用，B2 中包含对C2（C1的copy）的引用。

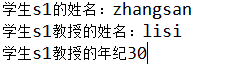
**若不对clone()方法进行改写，则调用此方法得到的对象即为浅拷贝**，下面我们着重谈一下深拷贝。运行下面的程序，看一看浅拷贝：

|  |
| --- |
| class Professor0 implements Cloneable {  String name;  int age;    Professor0(String name, int age) {  this.name = name;  this.age = age;  }    public Object clone() throws CloneNotSupportedException {  return super.clone();  }  } |

|  |
| --- |
| class Student0 implements Cloneable {  String name;// 常量对象。  int age;  Professor0 p;// 学生1和学生2的引用值都是一样的。    Student0(String name, int age, Professor0 p) {  this.name = name;  this.age = age;  this.p = p;  }    public Object clone() {  Student0 o = null;  try {  o = (Student0) super.clone();  } catch (CloneNotSupportedException e) {  System.out.println(e.toString());  }    return o;  }  } |

|  |
| --- |
| public class ShallowCopy {  public static void main(String[] args) {  Professor0 p = new Professor0("wangwu", 50);  Student0 s1 = new Student0("zhangsan", 18, p);  Student0 s2 = (Student0) s1.clone();  s2.p.name = "lisi";  s2.p.age = 30;  s2.name = "z";  s2.age = 45;  System.out.println("学生s1的姓名：" + s1.name + "\n学生s1教授的姓名：" + s1.p.name + "," + "\n学生s1教授的年纪" + s1.p.age);// 学生1的教授  }  } |

运行结果：



s2变了，但s1也变了，证明s1的p和s2的p指向的是同一个对象。这在我们有的实际需求中，却不是这样，因而我们需要深拷贝：

#### 1.1.6.4 深拷贝方法一： 使用clone

|  |
| --- |
| class Professor implements Cloneable {  String name;  int age;    Professor(String name, int age) {  this.name = name;  this.age = age;  }  public Object clone() {  Object o = null;  try {  o = super.clone();  } catch (CloneNotSupportedException e) {  System.out.println(e.toString());  }  return o;  }  } |

|  |
| --- |
| class Student implements Cloneable {  String name;  int age;  Professor p;    Student(String name, int age, Professor p) {  this.name = name;  this.age = age;  this.p = p;  }    public Object clone() {  Student o = null;  try {  o = (Student) super.clone();  } catch (CloneNotSupportedException e) {  System.out.println(e.toString());  }  o.p = (Professor) p.clone();  return o;  }  }  1 public class DeepCopy {  public static void main(String args[]) {  long t1 = System.currentTimeMillis();  Professor p = new Professor("wangwu", 50);  Student s1 = new Student("zhangsan", 18, p);  Student s2 = (Student) s1.clone();  s2.p.name = "lisi";  s2.p.age = 30;  System.out.println("name=" + s1.p.name + "," + "age=" + s1.p.age);// 学生1的教授不改变。  long t2 = System.currentTimeMillis();  System.out.println(t2-t1);  }  } |

运行结果：

http://images2015.cnblogs.com/blog/829125/201703/829125-20170316122941026-206223786.png

#### 1.1.6.5 深拷贝方法二： 对象串行化

当然我们还有一种深拷贝方法，就是将对象串行化：

|  |
| --- |
| class Professor2 implements Serializable {  /\*\*  \*  \*/  private static final long serialVersionUID = 1L;  String name;  int age;    Professor2(String name, int age) {  this.name = name;  this.age = age;  } |

|  |
| --- |
| class Student2 implements Serializable {  /\*\*  \*  \*/  private static final long serialVersionUID = 1L;  String name;// 常量对象。  int age;  Professor2 p;// 学生1和学生2的引用值都是一样的。    Student2(String name, int age, Professor2 p) {  this.name = name;  this.age = age;  this.p = p;  }    public Object deepClone() throws IOException, OptionalDataException,  ClassNotFoundException {  // 将对象写到流里  ByteArrayOutputStream bo = new ByteArrayOutputStream();  ObjectOutputStream oo = new ObjectOutputStream(bo);  oo.writeObject(this);  // 从流里读出来  ByteArrayInputStream bi = new ByteArrayInputStream(bo.toByteArray());  ObjectInputStream oi = new ObjectInputStream(bi);  return (oi.readObject());  }  } |

|  |
| --- |
| public class DeepCopy2 {    /\*\*  \* @param args  \*/  public static void main(String[] args) throws OptionalDataException,  IOException, ClassNotFoundException {  long t1 = System.currentTimeMillis();  Professor2 p = new Professor2("wangwu", 50);  Student2 s1 = new Student2("zhangsan", 18, p);  Student2 s2 = (Student2) s1.deepClone();  s2.p.name = "lisi";  s2.p.age = 30;  System.out.println("name=" + s1.p.name + "," + "age=" + s1.p.age); // 学生1的教授不改变。  long t2 = System.currentTimeMillis();  System.out.println(t2-t1);  }    } |

要想序列化对象，必须先创建一个OutputStream，然后把它嵌入ObjectOutputStream。这时就能用writeObject()方法把对象写入OutputStream。读的时候需要把InputStream嵌到ObjectInputStream中，然后再调用readObject()方法。不过这样读出来的只是一个Object的reference，因此，在用之前，还要下转型。

对象序列化不仅能保存对象的副本，而且会跟着对象中的reference把它所引用的对象也保存起来，然后再继续跟踪那些对象的reference，以此类推。这种情形常被称作”单个对象所联结的‘对象网’ “。

但是串行化却很耗时，在一些框架中，我们便可以感受到，它们往往将对象进行串行化后进行传递，耗时较多。

学习自：<http://www.cnblogs.com/xuanxufeng/p/6558330.html>

### 1.1.7 Java finalize方法使用

《**JAVA**编程思想》：

java提供[**finalize**](http://www.51testing.com/javascr%C4%ABpt:;)()方法，垃圾回收器准备释放内存的时候，会先调用finalize()。

       (1).对象不一定会被回收。

       (2).垃圾回收不是析构函数。

       (3).垃圾回收只与内存有关。

(4).垃圾回收和finalize()都是靠不住的，只要JVM还没有快到耗尽内存的地步，它是不会浪费时间进行垃圾回收的。

有时当撤消一个对象时，需要完成一些操作。例如，如果一个对象正在处理的是非Java 资源，如文件句柄或window 字符字体，这时你要确认在一个对象被撤消以前要保证这些资源被释放。为处理这样的状况，Java 提供了被称为收尾（finalization ）的机制。使用该机制你可以定义一些特殊的操作，这些操作在一个对象将要被垃圾回收程序释放时执行。

要给一个类增加收尾（finalizer ），你只要定义finalize ( ) 方法即可。Java 回收该类的一个对象时，就会调用这个方法。在finalize ( )方法中，你要指定在一个对象被撤消前必须执行的操作。垃圾回收周期性地运行，**检查对象不再被运行状态引用或间接地通过其他对象引用**。就在对象被释放之前，Java 运行系统调用该对象的finalize( ) 方法。

finalize()方法的通用格式如下：

protected void finalize( ){  
 // finalization code here  
}

其中，关键字protected是防止在该类之外定义的代码访问finalize()标识符。该标识符和其他标识符将在第7章中解释。

理解finalize( ) 正好在垃圾回收以前被调用非常重要。例如当一个对象超出了它的作用域时，finalize( ) 并不被调用。这意味着你不可能知道何时——甚至是否——finalize( ) 被调用。因此，你的程序应该提供其他的方法来释放由对象使用的系统资源，而不能依靠finalize( ) 来完成程序的正常操作。

注意：如果你熟悉C ，那你知道C 允许你为一个类定义一个撤消函数（destructor ），它在对象正好出作用域之前被调用。Java不支持这个想法也不提供撤消函数。finalize() 方法只和撤消函数的功能接近。当你对Java 有丰富经验时，你将看到因为Java使用垃圾回收子系统，几乎没有必要使用撤消函数。

#### 1.1.7.1 理解[****finalize****](http://www.51testing.com/javascr%C4%ABpt:;)()-析构函数的替代者

by Tim Gooch

在许多方面，Java 类似于 C++。Java 的语法非常类似于 C++，Java 有类、方法和数据成员；Java 的类有构造函数； Java 有异常处理。

但是，如果你使用过 C++ 会发现 Java 也丢掉一些可能是你熟悉的特性。这些特性之一就是析构函数。取代使用析构函数，Java 支持finalize() 方法。析构函数(destructor) 与构造函数相反，当对象结束其生命周期时（例如对象所在的函数已调用完毕），系统自动执行析构函数。析构函数往往用来做“清理善后” 的工作（例如在建立对象时用new开辟了一片内存空间，delete会自动调用析构函数后释放内存）。

在本文中，我们将描述 finalize() 与 C++ 析构函数的区别。另外，我们将创建一个简单的 Applet 来演示 finalize() 是如何**工作**的。

#### 1.1.7.2 ****最终的界限****

与 Java 不同，C++ 支持局部对象（基于栈）和全局对象（基于堆）。因为这一双重支持，C++ 也提供了自动构造和析构，这导致了对构造函数和析构函数的调用，（对于堆对象）就是内存的分配和释放。

**在 Java 中，所有对象都驻留在堆内存，因此局部对象就不存在**。结果，Java 的设计者觉得不需要析构函数（象 C++ 中所实现的）。

取而代之，Java 定义了一个特殊的方法叫做finalize() ，它提供了 C++ 析构函数的一些功能。但是，finalize() 并不完全与 C++ 的析构函数一样，并可以假设它会导致一系列的问题。**finalize() 方法作用的一个关键元素是 Java 的垃圾回收器。**

#### 1.1.7.3 垃圾回收器

在 C/C++、Pascal和[**其他**](http://www.51testing.com/javascr%C4%ABpt:;)几种多种用途的编程语言中，开发者有责任在内存管理上发挥积极的作用。例如，如果你为一个对象或数据结构分配了内存，那么当你不再使用它时必须释放掉该内存。

在 Java 中，当你创建一个对象时，Java 虚拟机（JVM）为该对象分配内存、调用构造函数并开始跟踪你使用的对象。当你停止使用一个对象（就是说，当没有对该对象有效的引用时），JVM 通过垃圾回收器将该对象标记为释放状态。

当垃圾回收器将要释放一个对象的内存时，它调用该对象的finalize() 方法（如果该对象定义了此方法）。**垃圾回收器以独立的低优先级的方式运行，只有当其他线程挂起等待该内存释放的情况出现时，它才开始运行释放对象的内存**。（事实上，你可以调用System.gc() 方法强制垃圾回收器来释放这些对象的内存。）

在以上的描述中，有一些重要的事情需要注意。首先，**只有当垃圾回收器释放该对象的内存时，才会执行finalize()**。如果在 Applet 或应用程序退出之前垃圾回收器没有释放内存，垃圾回收器将不会调用finalize()。

其次，除非垃圾回收器认为你的 Applet 或应用程序需要额外的内存，否则它不会试图释放不再使用的对象的内存。换句话说，这是完全可能的：一个 Applet 给少量的对象分配内存，没有造成严重的内存需求，于是垃圾回收器没有释放这些对象的内存就退出了。

显然，**如果你为某个对象定义了finalize() 方法，JVM 可能不会调用它，因为垃圾回收器不曾释放过那些对象的内存**。调用System.gc() 也不会起作用，因为它仅仅是给 JVM 一个建议而不是命令。

#### 1.1.7.4 finalize() 有什么优点呢？

如果finalize() 不是析构函数，JVM 不一定会调用它，你可能会疑惑它是否在任何情况下都有好处。事实上，在 Java 1.0 中它并没有太多的优点。

**根据 Java 文档，finalize() 是一个用于释放非 Java 资源的方法。但是，JVM 有很大的可能不调用对象的finalize() 方法，因此很难证明使用该方法释放资源是有效的。**

Java 1.1 通过提供一个System.runFinalizersOnExit() 方法部分地解决了这个问题。（不要将这个方法与 Java 1.0 中的System.runFinalizations() 方法相混淆。）不象System.gc() 方法那样，**System.runFinalizersOnExit() 方法并不立即试图启动垃圾回收器。而是当应用程序或 Applet 退出时，它调用每个对象的finalize() 方法。**

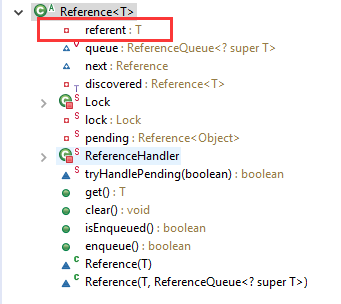
正如你可能猜测的那样，通过调用System.runFinalizersOnExit() 方法强制垃圾回收器清除所有独立对象的内存，当清除代码执行时可能会引起明显的延迟。

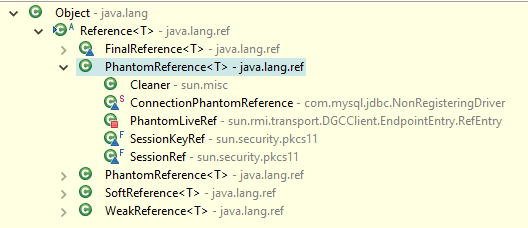
#### 1.1.7.5 结论

然而有益的是，Java 的自动垃圾回收器不会失去平衡。作为便利的代价，你不得不放弃对系统资源释放的控制。不象 C++ 中的析构函数，Java Applet 不会自动执行你的类中的finalize() 方法。事实上，如果你正在使用 Java 1.0，即使你试图强制它调用finalize() 方法，也不能确保将调用它。

因此，你不应当依靠finalize() 来执行你的 Applet 和应用程序的资源清除工作。取而代之，你应当明确的清除那些资源或创建一个try...finally 块（或类似的机制）来实现。

### 1.1.8 Java的四种引用，强弱软虚，用到的场景





众所周知，java中是JVM负责内存的分配和回收，这是它的优点（使用方便，程序不用再像使用c那样操心内存），但同时也是它的缺点（不够灵活）。为了解决内存操作不灵活这个问题，可以采用软引用等方法。

在JDK1.2以前的版本中，当一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及状态，程序才能使用它。这 就像在日常生活中，从商店购买了某样物品后，如果有用，就一直保留它，否则就把它扔到垃圾箱，由清洁工人收走。一般说来，如果物品已经被扔到垃圾箱，想再 把它捡回来使用就不可能了。

但有时候情况并不这么简单，你可能会遇到类似鸡肋一样的物品，食之无味，弃之可惜。这种物品现在已经无用了，保留它会占空间，但是立刻扔掉它也不划算，因 为也许将来还会派用场。对于这样的可有可无的物品，一种折衷的处理办法是：如果家里空间足够，就先把它保留在家里，如果家里空间不够，即使把家里所有的垃 圾清除，还是无法容纳那些必不可少的生活用品，那么再扔掉这些可有可无的物品。

从JDK1.2版本开始，把对象的引用分为四种级别，从而使程序能更加灵活的控制对象的生命周期。这四种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

#### 1.1.8.1 强引用

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

#### 1.1.8.2 软引用（SoftReference）

如果一个对象只具有软引用，那就类似于可有可无的生活用品。如果内存空间足够，垃圾回收器就不会回收它，如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA虚拟机就会把这个软引用加入到与之关联的引用队列中。

#### 1.1.8.3 弱引用（WeakReference）

如果一个对象只具有弱引用，那就类似于可有可无的生活用品。弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程， 因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。

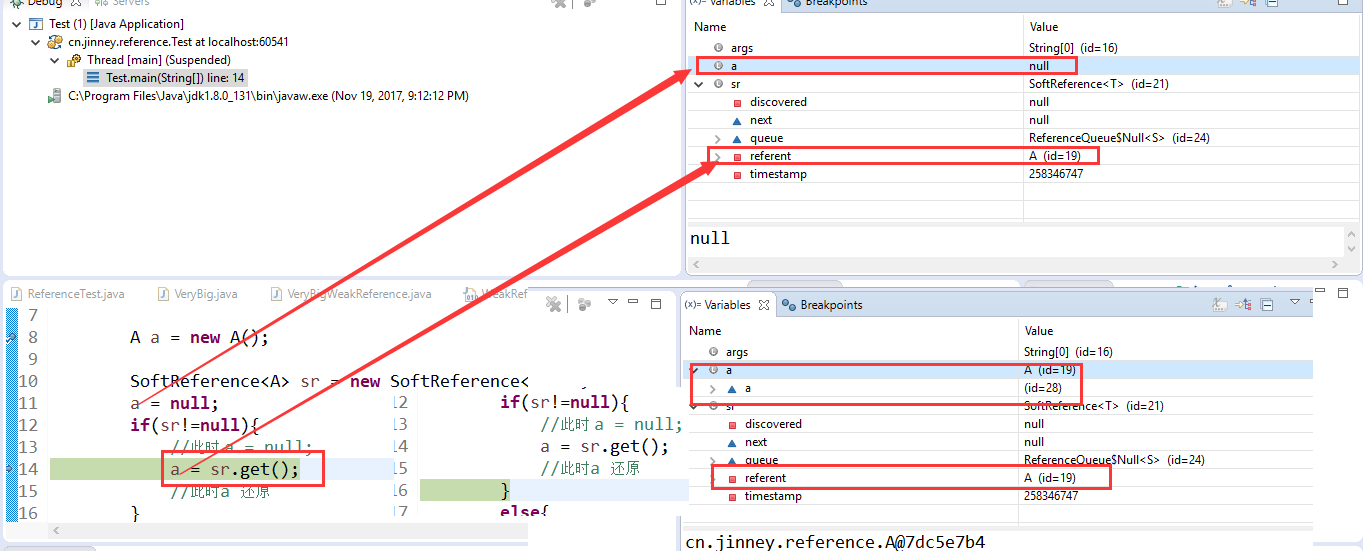
#### 1.1.8.4 虚引用（PhantomReference）

"虚引用"顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收。

虚引用主要用来跟踪对象被垃圾回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列（ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。程序如果发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

特别注意，在世纪程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，这是因为软引用可以加速JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出（OutOfMemory）等问题的产生。

|  |
| --- |
| **package** cn.jinney.reference;  **import** java.lang.ref.SoftReference;  **public** **class** Test {    **public** **static** **void** main(String[] args){  System.***out***.println("开始");    A a = **new** A();    SoftReference<A> sr = **new** SoftReference<A>(a);  a = **null**;  **if**(sr!=**null**){  a = sr.get();  }  **else**{  a = **new** A();  sr = **new** SoftReference<A>(a);  }    System.***out***.println("结束");  }  }    **class** A{  **int**[] a ;  **public** A(){  a = **new** **int**[100000000];  }  } |



#### 1.1.8.5 弱引用与虚引用的用处

　　软引用很明显可以用来制作caching和pooling，而弱引用与虚引用呢？其实用处也很大，首先我们来看看弱引用，举个例子：

|  |
| --- |
| **class** Registry {  @SuppressWarnings("rawtypes")  **private** Set registeredObjects = **new** HashSet();  /\*\*  \* registeredObjects 中的object永远不会被GC回收，因为这里有个强引用保存在registeredObjects  \*  \* **@param** object  \*/  @SuppressWarnings("unchecked")  **public** **void** register(Object object) {  registeredObjects.add(object);  }  } |

**所有我添加进 registeredObjects 中的object永远不会被GC回收，因为这里有个强引用保存在registeredObjects里**，另一方面如果我把代码改为如下：

|  |
| --- |
| **class** Registry2 {  @SuppressWarnings("rawtypes")  **private** Set registeredObjects = **new** HashSet();  /\*\*  \* 如果GC想要回收registeredObjects中的object，便能够实现了  \*  \* **@param** object  \*/  @SuppressWarnings({ "unchecked", "rawtypes" })  **public** **void** register(Object object) {  registeredObjects.add(**new** WeakReference(object));  }  } |

现在如果GC想要回收registeredObjects中的object，便能够实现了，同样在使用HashMap如果想实现如上的效果，一种更好的实现是使用WeakHashMap。

而虚引用呢？我们先来看看javadoc的部分说明：

Phantom references are useful for implementing cleanup operations that are necessary before an object gets garbage-collected. They are sometimes more flexible than the [finalize()](http://www.cnblogs.cc2/reference/java/lang/Object.html#finalize()) method.

翻译一下：

虚引用在实现一个对象被回收之前必须做清理操作是很有用的。有时候，他们比finalize()方法更灵活。

很明显的，虚引用可以用来做对象被回收之前的清理工作。

学习自：

<https://www.cnblogs.com/remember-forget/p/5971963.html>

<https://www.cnblogs.com/newcj/archive/2011/05/15/2046882.html>

### 1.1.9  Hashcode的作用

以下是关于HashCode的官方文档定义：

1. hashcode方法返回该对象的哈希码值。支持该方法是为哈希表提供一些优点，例如，java.util.Hashtable 提供的哈希表。
3. hashCode 的常规协定是：
4. 在 Java 应用程序执行期间，在同一对象上多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是对象上 equals 比较中所用的信息没有被修改。从某一应用程序的一次执行到同一应用程序的另一次执行，该整数无需保持一致。
5. 如果根据 equals(Object) 方法，两个对象是相等的，那么在两个对象中的每个对象上调用 hashCode 方法都必须生成相同的整数结果。
6. 以下情况不 是必需的：如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么在两个对象中的任一对象上调用 hashCode 方法必定会生成不同的整数结果。但是，程序员应该知道，为不相等的对象生成不同整数结果可以提高哈希表的性能。
7. 实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。（这一般是通过将该对象的内部地址转换成一个整数来实现的，但是 JavaTM 编程语言不需要这种实现技巧。）
8. 当equals方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定，该协定声明相等对象必须具有相等的哈希码。

以上这段官方文档的定义，我们可以抽出成以下几个关键点：

1、hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的；

2、如果两个对象相同，就是适用于equals(java.lang.Object) 方法，那么这两个对象的hashCode一定要相同；

3、如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；

4、两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们**“存放在同一个篮子里”**。

再归纳一下就是hashCode是用于查找使用的，而equals是用于比较两个对象的是否相等的。

1. 1.hashcode是用来查找的，如果你学过数据结构就应该知道，在查找和排序这一章有
2. 例如内存中有这样的位置
3. 0  1  2  3  4  5  6  7
4. 而我有个类，这个类有个字段叫ID,我要把这个类存放在以上8个位置之一，如果不用hashcode而任意存放，那么当查找时就需要到这八个位置里挨个去找，或者用二分法一类的算法。
5. 但如果用hashcode那就会使效率提高很多。
6. 我们这个类中有个字段叫ID,那么我们就定义我们的hashcode为ID％8，然后把我们的类存放在取得得余数那个位置。比如我们的ID为9，9除8的余数为1，那么我们就把该类存在1这个位置，如果ID是13，求得的余数是5，那么我们就把该类放在5这个位置。这样，以后在查找该类时就可以通过ID除 8求余数直接找到存放的位置了。
8. 2.但是如果两个类有相同的hashcode怎么办那（我们假设上面的类的ID不是唯一的），例如9除以8和17除以8的余数都是1，那么这是不是合法的，回答是：可以这样。那么如何判断呢？在这个时候就需要定义 equals了。
9. 也就是说，我们先通过 hashcode来判断两个类是否存放某个桶里，但这个桶里可能有很多类，那么我们就需要再通过 equals 来在这个桶里找到我们要的类。
10. 那么。重写了equals()，为什么还要重写hashCode()呢？
11. 想想，你要在一个桶里找东西，你必须先要找到这个桶啊，你不通过重写hashcode()来找到桶，光重写equals()有什么用啊

|  |
| --- |
| **package** cn.jinney.reference;  **import** java.util.HashSet;  **import** java.util.Set;  **public** **class** HashTest {  **private** **int** i;    **public** **int** getI() {  **return** i;  }  **public** **void** setI(**int** i) {  **this**.i = i;  }    **public** **int** hashCode() {  **return** i % 10;  }    **public** **final** **static** **void** main(String[] args) {  HashTest a = **new** HashTest();  HashTest b = **new** HashTest();  a.setI(1);  b.setI(1);  Set<HashTest> set = **new** HashSet<HashTest>();  set.add(a);  set.add(b);  System.***out***.println(a.hashCode() == b.hashCode());  System.***out***.println(a.equals(b));  System.***out***.println(set);  }  } |

这个输出的结果：

1. true
2. false
3. [com.ubs.sae.test.HashTest@1, com.ubs.sae.test.HashTest@1]

以上这个示例，我们只是重写了hashCode方法，从上面的结果可以看出，虽然两个对象的hashCode相等，但是实际上两个对象并不是相等；，我们没有重写equals方法，那么就会调用object默认的equals方法，是比较两个对象的引用是不是相同，显示这是两个不同的对象，两个对象的引用肯定是不定的。这里我们将生成的对象放到了HashSet中，而HashSet中只能够存放唯一的对象，也就是相同的（适用于equals方法）的对象只会存放一个，但是这里实际上是两个对象a,b都被放到了HashSet中，这样HashSet就失去了他本身的意义了。

此时我们把equals方法给加上：

true

true

[com.ubs.sae.test.HashTest@1]

从结果我们可以看出，现在两个对象就完全相等了，HashSet中也只存放了一份对象。

### 1.1.10 面向对象的特征：继承、封装和多态

**封装：**

* 把对象的全部属性和服务结合在一起，形成不可分割的独立单位；
* 尽可能隐藏内部细节。（数据隐藏、隐藏具体实现的细节）

    public、private、protected、default

* public：既可以跨类访问，也可以跨包（package）访问
* private：只能被该类对象访问，子类不能访问，更不能跨包访问
* protected：只能被该类本身和子类访问，即使子类不在包中也可以访问
* default：即不添加任何修饰符，只允许同一个包进行访

**继承：**

 1. is a 关系（eg：经理 is a 职员）

* 初始化子类必须先初始化父类（虚拟机自动完成）因为子类的构造方法中会调用父类的构造方法
* 单继承性：一个子类只能继承一个父亲
* 关键字super：用来引用自己的父类，比如父类的方法和属性都是可以访问的

  2.方法覆盖（也叫方法重写）

* 返回类型、方法名称、参数列表必须与它所覆盖的方法相同
* 访问权限不能缩小
* 不能抛出更多异常

  3.方法重载

* 同一个方法（包括父类），方法名相同，参数列表不同，返回类型可相同可不同

  4.组合与继承

* 组合是”has-a“的关系。组合指在新类中创建原有类的对象，重复利用已有类的功能。而继承是根据其他类的实现来定义一个类的实现。
* 是面向对象中两种代码复用的方式。
* 能使用组合就尽量不要使用继承。除非是”is-a“的关系否，则不要轻易使用继承。

**多态：**

* 简单说就是子父类（接口或实现类）中的类型转换

  1. 属性的特点：属性不是多态的，当子类中出现相同属性时，以左边的引用为准。

class A{

　　int num = 3;

}

class B extends A{

　　int num = 5;

}

main{

　　A a = new B();

　　sop(a.num);　　//输出3

　　B b = new B();

　　sop(b.num);　　//输出5

}

 2.非静态私有函数特点：

* 编译时期：引用变量所属类（父类）中是否有调用方法，如果有编译通过，若没有编译失败。
* 运行时期：如果子类复写了父类方法，运行子类中的方法，若没有运行父类方法

  3.静态函数特点：

* 和属性一样，static方法不是多态的，无论编译还是运行都参考左边。

  4.instanceof运算符：

* 前面是一个引用类型变量后面是一个类，用于判断前面的对象是否是后面的类的实例，如果是返回true，若不是返回false，避免了ClassCastException。
* public interface Animal {  
  　　public abstract void eat();  
  }
* public class Cat implements Animal{  
  　　public void eat(){  
  　　　　System.out.println("chi yu");  
  　　}  
  　　public void catchMouse(){  
  　　　　System.out.println("zhu lao shu");  
  　　}  
  }
* public class Dog implements Animal{  
  　　public void eat(){  
  　　　　System.out.println("chi gu tou");  
  　　}  
  　　public void kanHouse(){  
  　　　　System.out.println("kan men");  
  　　}  
  }
* public class Test {  
  　　public static void main(String[] args) {  
  　　Animal a = new Cat();  
  　　function(a);  
  　　}  
    
  　　public static void function(Animal a){  
  　　a.eat();  
  　　if(a instanceof Cat){  
  　　　　Cat c = (Cat)a;  
  　　　　c.catchMouse();  
  　　}else if(a instanceof Dog){  
  　　　　Dog d = (Dog)a;  
  　　　　d.kanHouse();  
  　　　　}  
  　　}  
  }

<https://www.cnblogs.com/wangqi327/p/4442603.html>

### 1.1.11 final, finally, finalize 的区别

#### 1.1.11.1 final

　　在java中，final可以用来修饰类，方法和变量（成员变量或局部变量）。下面将对其详细介绍。

**修饰类**

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

final类中所有的成员方法都会隐式的定义为final方法。

**修饰方法**

使用final方法的原因主要有两个：

　　(1) 把方法锁定，以防止继承类对其进行更改。

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

final方法意味着“最后的、最终的”含义，即此方法不能被重写。

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

* class A{
* private final void getName(){
* }
* }
* public class B extends A{
* public void getName(){
* }
* public static void main(String[]args){
* System.out.println("OK");
* }
* }

**修饰变量**

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

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

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

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



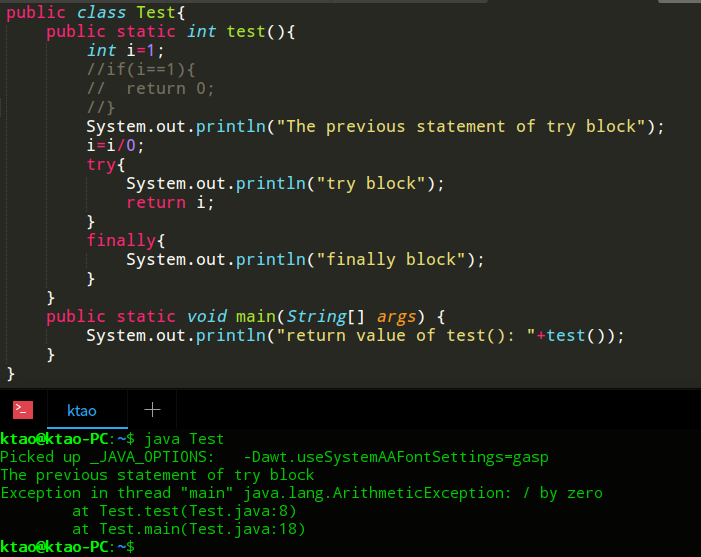
 　　在java中，String被设计成final类，那为什么平时使用时，String的值可以被改变呢？

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

#### 1.1.11.2 finally

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

　　很多人都认为finally语句块一定会执行，但真的是这样么？答案是否定的，例如下面这个例子：



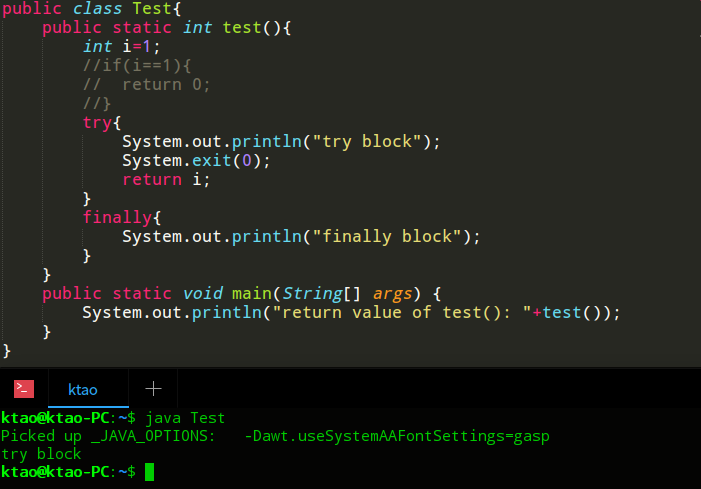
　　当我们去掉注释的三行语句，执行结果为：

https://images2018.cnblogs.com/blog/876623/201803/876623-20180317092814423-841848766.png

 　　为什么在以上两种情况下都没有执行finally语句呢，说明什么问题？

　　只有与finally对应的try语句块得到执行的情况下，finally语句块才会执行。以上两种情况在执行try语句块之前已经返回或抛出异常，所以try对应的finally语句并没有执行。

　　但是，在某些情况下，即使try语句执行了，finally语句也不一定执行。例如以下情况：



　　finally 语句块还是没有执行，为什么呢？因为我们在 try 语句块中执行了 System.exit (0) 语句，终止了 Java 虚拟机的运行。那有人说了，在一般的 Java 应用中基本上是不会调用这个 System.exit(0) 方法的。OK ！没有问题，我们不调用 System.exit(0) 这个方法，那么 finally 语句块就一定会执行吗？

　　再一次让大家失望了，答案还是否定的。当一个线程在执行 try 语句块或者 catch 语句块时被打断（interrupted）或者被终止（killed），与其相对应的 finally 语句块可能不会执行。还有更极端的情况，就是在线程运行 try 语句块或者 catch 语句块时，突然死机或者断电，finally 语句块肯定不会执行了。可能有人认为死机、断电这些理由有些强词夺理，没有关系，我们只是为了说明这个问题。

**易错点**

在try-catch-finally语句中执行return语句。我们看如下代码：



　　答案：4,4,4  。    为什么呢？

　　首先finally语句在改代码中一定会执行，从运行结果来看，每次return的结果都是4（即finally语句），仿佛其他return语句被屏蔽掉了。

　　事实也确实如此，因为finally用法特殊，所以会撤销之前的return语句，继续执行最后的finally块中的代码。

#### 1.1.11.3 finalize

　　finalize()是在java.lang.Object里定义的，也就是说每一个对象都有这么个方法。这个方法在gc启动，该对象被回收的时候被调用。其实gc可以回收大部分的对象（凡是new出来的对象，gc都能搞定，一般情况下我们又不会用new以外的方式去创建对象），所以一般是不需要程序员去实现finalize的。   
特殊情况下，需要程序员实现finalize，当对象被回收的时候释放一些资源，比如：一个socket链接，在对象初始化时创建，整个生命周期内有效，那么就需要实现finalize，关闭这个链接。   
　　使用finalize还需要注意一个事，调用super.finalize();

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

<https://www.cnblogs.com/ktao/p/8586966.html>

### 1.1.12 Exception、Error、运行时异常与一般异常有何异同

Throwable 是所有 [Java](http://lib.csdn.net/base/java) 程序中错误处理的父类 ，有两种资类： Error 和 Exception 。

Error ：表示由 JVM 所侦测到的无法预期的错误，由于这是属于 JVM 层次的严重错误 ，导致 JVM 无法继续执行，因此，这是不可捕捉到的，无法采取任何恢复的操作，顶多只能显示错误信息。

Exception ：表示可恢复的例外，这是可捕捉到的。

Java 提供了两类主要的异常 :runtime exception 和 checked exception 。 checked 异常也就是我们经常遇到的 IO 异常，以及 SQL 异常都是这种异常。 对于这种异常， JAVA 编译器强制要求我们必需对出现的这些异常进行 catch 。所以，面对这种异常不管我们是否愿意，只能自己去写一大堆 catch 块去处理可能的异常。

    但是另外一种异常： runtime exception ，也称运行时异常，我们可以不处理。当出现这样的异常时，总是由虚拟机 接管。比如：我们从来没有人去处理过 NullPointerException 异常，它就是运行时异常，并且这种异常还是最常见的异常之一。

    出现运行时异常后，系统会把异常一直往上层抛，一直遇到处理代码。如果没有处理块，到最上层，如果是多线程就由 Thread.run()抛出 ，如果是单线程就被 main() 抛出 。抛出之后，如果是线程，这个线程也就退出了。如果是主程序抛出的异常，那么这整个程序也就退出了。运行时异常是 Exception 的子类，也有一般异常的特点，是可以被 Catch 块处理的。只不过往往我们不对他处理罢了。也就是说，你如果不对运行时异常进行处理，那么出现运行时异常之后，要么是线程中止，要么是主程序终止。

    如果不想终止，则必须扑捉所有的运行时异常，决不让这个处理线程退出。队列里面出现异常数据了，正常的处理应该是把异常数据舍弃，然后记录日志。不应该由于异常数据而影响下面对正常数据的处理。 在这个场景这样处理可能是一个比较好的应用，但并不代表在所有的场景你都应该如此。如果在其它场景，遇到了一些错误，如果退出程序比较好，这时你就可以不太理会运行时异常 ，或者是通过对异常的处理显式的控制程序退出。

异常处理的目标之一就是为了把程序从异常中恢复出来 。

<http://blog.sina.com.cn/s/blog_6aefe425010187xq.html>

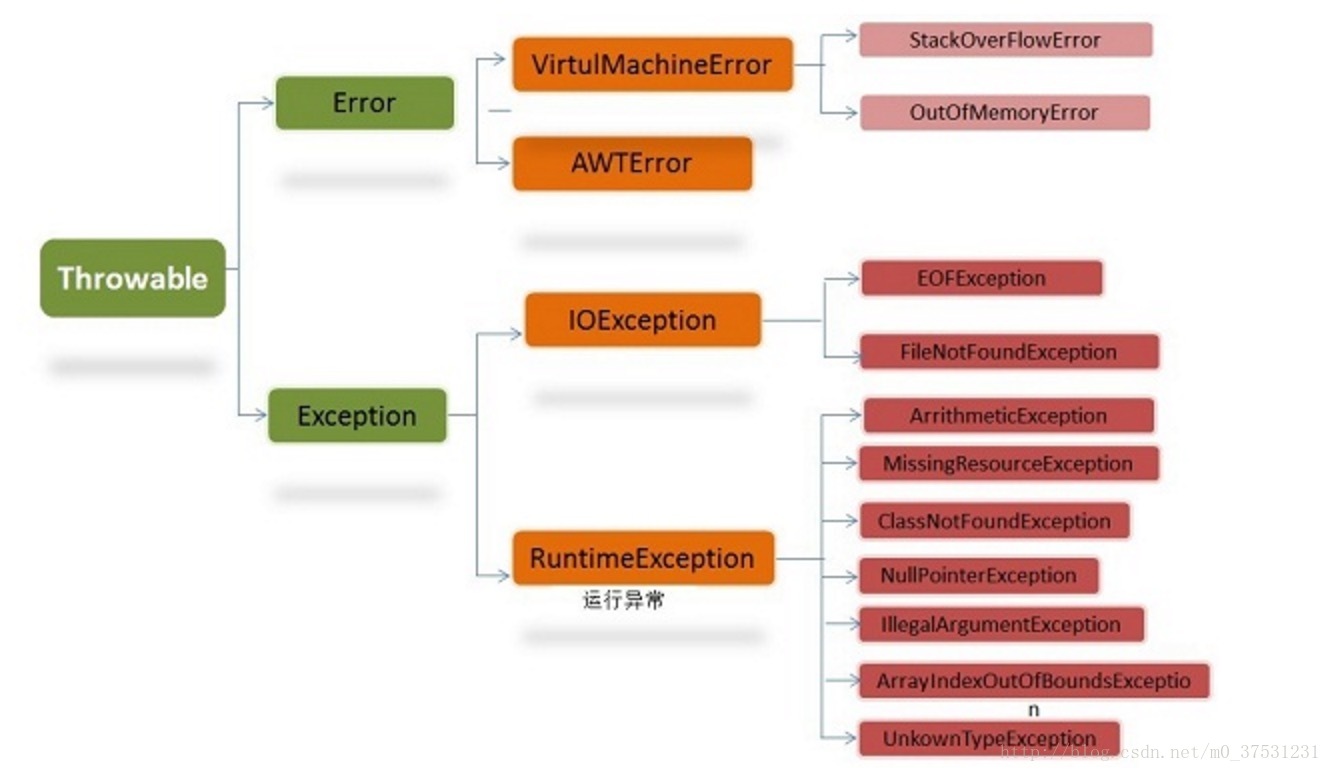
#### 1.1.12.1 开场白

对于程序运行过程中的可能出现异常情况，java语言使用一种称为异常处理的错误捕捉机制进行处理。相信大家对 try { }catch( ){} finally{} 这种结构非常熟悉，使用频率极高。既然经常使用它，而且也是面试常问知识点，我们就有必要去深入地了解一下。也谈不上深入，只是java语言的基本功。下面，开始吧！

#### 1.1.12.2 异常分类

在java中，异常对象都是派生于Throwable类的一个实例。如果java内置的异常类不能够满足需求，用户还可以创建自己的异常类。

下图是java异常类层次结构图



可以看出，所有的异常都是由Throwable类，下一层分解为两个分支：Error和Exceprion。Error层次结构描述了java运行时系统的内部错误和资源耗尽错误。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。应用程序不应该抛出这种类型的对象。Exceprion这个层次结构又分解为连个分支：一个分支派生于RuntimeException；另一个分支包含其他异常。划分两个分支的规则是：由程序错误导致的异常属于RuntimeException；而程序本身没有没有问题，但由于像I/O错误这类异常导致的异常属于其他异常。   
**常见的RuntimeException（运行时异常）：**   
IndexOutOfBoundsException(下标越界异常)   
NullPointerException(空指针异常)   
NumberFormatException （String转换为指定的数字类型异常）   
ArithmeticException -（算术运算异常 如除数为0）   
ArrayStoreException - （向数组中存放与声明类型不兼容对象异常）   
SecurityException -（安全异常）   
**IOException（其他异常）**   
FileNotFoundException（文件未找到异常。）   
IOException（操作输入流和输出流时可能出现的异常。）   
EOFException （文件已结束异常）

#### 1.1.12.3 概念理解

首先明白下面的两个概念   
**unchecked exception（非检查异常）**：包括运行时异常（RuntimeException）和派生于Error类的异常。对于运行时异常，java编译器不要求必须进行异常捕获处理或者抛出声明，由程序员自行决定。   
**checked exception（检查异常，编译异常，必须要处理的异常）**   
也：称非运行时异常（运行时异常以外的异常就是非运行时异常），java编译器强制程序员必须进行捕获处理，比如常见的IOExeption和SQLException。对于非运行时异常如果不进行捕获或者抛出声明处理，编译都不会通过。

#### 1.1.12.4 异常的处理

**（1）、抛出异常**

1、调用一个抛出受查异常的方法必须用throws 子句声明 调用method2()方法。   
 2、程序运行过程中发现错误，并且利用throw抛出一个受查异常 下面method2()方法。

|  |
| --- |
| @Test  public void test() throws FileNotFoundException {  method();  }  public void method() throws FileNotFoundException {  //一个会抛出异常的方法  method2();  }  //这里 方法后是throws  public void method2() throws FileNotFoundException {  //这里是throw  throw new FileNotFoundException();  } |

**（2）、捕获异常**

try { }catch( ){} finally{} 语句块这就比较常见了。不在赘述。 不过下面有一道有意思的题，实际使用中不太会遇见，面试题常见。

来，看题！

①

|  |
| --- |
| @Test  public void test() {  System.out.println(test11());  }  public String test11() {  try {  System.out.println("try block");  return test12();  } finally {  System.out.println("finally block");  }  }  public static String test12() {  System.out.println("return statement");  return "after return";  } |

答案：   
try block   
return statement   
finally block   
after return

②

|  |
| --- |
| @Test  public void test() {  System.out.println(test2());  }  public int test() {  int b = 20;  try {  System.out.println("try block");  return b += 80;  } catch (Exception e) {  System.out.println("catch block");  } finally {  System.out.println("finally block");  if (b > 25) {  System.out.println("b>25, b = " + b);  }  return 200;  }  } |

答案：   
try block   
finally block   
b>25, b = 100   
200

总结：finally块的语句在try或catch中的return语句执行之后返回之前执行且finally里的修改语句可能影响也可能不影响try或catch中 return已经确定的返回值，若finally里也有return语句则覆盖try或catch中的return语句直接返回。

<https://blog.csdn.net/m0_37531231/article/details/79502778>

### 1.1.13 请写出5种常见到的runtime exception

RuntimeException是java中所有运行时异常的父类，实际运行时出现的都是它的子类，看看RuntimeException的Java doc就可以随便列出几个：

**ClassCastException：**

Object x = new Integer(0);

System.out.println((String)x);

当试图将对象强制转换为不是实例的子类时，抛出该异常（ClassCastException)。

**ArithmeticException：**

int a=5/0;

一个整数“除以零”时，抛出ArithmeticException异常。

**NullPointerException：**

String s=null;

int size=s.size();

当应用程序试图在需要对象的地方使用 null 时，抛出NullPointerException异常

**StringIndexOutOfBoundsException：**

"hello".indexOf(-1);

指示索引或者为负，或者超出字符串的大小，抛出StringIndexOutOfBoundsException异常

**NegativeArraySizeException：**

String[] ss=new String[-1];

如果应用程序试图创建大小为负的数组，则抛出NegativeArraySizeException异常。

### 1.1.14 int 和 Integer 有什么区别，Integer的值缓存范围

#### 1.1.14.1 int与Integer的基本使用对比

（1）Integer是int的包装类；int是基本数据类型；   
（2）Integer变量必须实例化后才能使用；int变量不需要；   
（3）Integer实际是对象的引用，指向此new的Integer对象；int是直接存储数据值 ；   
（4）Integer的默认值是null；int的默认值是0。

#### 1.1.14.2 int与Integer的深入对比

（1）由于Integer变量实际上是对一个Integer对象的引用，所以两个通过new生成的Integer变量永远是不相等的（因为new生成的是两个对象，其内存地址不同）。

Integer i = new Integer(100);

Integer j = new Integer(100);

System.out.print(i == j); //false

（2）Integer变量和int变量比较时，只要两个变量的值是向等的，则结果为true（因为包装类Integer和基本数据类型int比较时，java会自动拆包装为int，然后进行比较，实际上就变为两个int变量的比较）

Integer i = new Integer(100);

int j = 100；

System.out.print(i == j); //true

（3）非new生成的Integer变量和new Integer()生成的变量比较时，结果为false。（因为非new生成的Integer变量指向的是java常量池中的对象，而new Integer()生成的变量指向堆中新建的对象，两者在内存中的地址不同）

Integer i = new Integer(100);

Integer j = 100;

System.out.print(i == j); //false

（4）对于两个非new生成的Integer对象，进行比较时，如果两个变量的值在区间

-128到127之间，则比较结果为true，如果两个变量的值不在此区间，则比较结果为false

Integer i = 100;

Integer j = 100;

System.out.print(i == j); //true

Integer i = 128;

Integer j = 128;

System.out.print(i == j); //false

对于第4条的原因： java在编译Integer i = 100 ;时，会翻译成为Integer i = Integer.valueOf(100)。而java API中对Integer类型的valueOf的定义如下，对于-128到127之间的数，会进行缓存，Integer i = 127时，会将127进行缓存，下次再写Integer j = 127时，就会直接从缓存中取，就不会new了。

public static Integer valueOf(int i){

assert IntegerCache.high >= 127;

if (i >= IntegerCache.low && i <= IntegerCache.high){

return IntegerCache.cache[i + (-IntegerCache.low)];

}

return new Integer(i);

}

#### 1.1.14.3 Java两种数据类型

**Java两种数据类型分类**

（1）基本数据类型，分为boolean、byte、int、char、long、short、double、float；

（2）引用数据类型 ，分为数组、类、接口。

**Java为每个原始类型提供了封装类**

　　为了编程的方便还是引入了基本数据类型，但是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。

基本数据类型: boolean，char，byte，short，int，long，float，double

封装类类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

#### 1.1.14.4 基本解析

**自动装箱：将基本数据类型重新转化为对象**

public class Test {

public static void main(String[] args) {

//声明一个Integer对象

Integer num = 9;

//以上的声明就是用到了自动的装箱：解析为:Integer num = new Integer(9);

}

}

9是属于基本数据类型的，原则上它是不能直接赋值给一个对象Integer的，但jdk1.5后你就可以进行这样的声明。自动将基本数据类型转化为对应的封装类型，成为一个对象以后就可以调用对象所声明的所有的方法。

**自动拆箱：将对象重新转化为基本数据类型**

public class Test {

public static void main(String[] args) {

//声明一个Integer对象

Integer num = 9;

//进行计算时隐含的有自动拆箱

System.out.print(num--);

}

}

　　因为对象时不能直接进行运算的，而是要转化为基本数据类型后才能进行加减乘除。对比：

/装箱

Integer num = 10;

//拆箱

int num1 = num;

#### 1.1.14.5 深入解析

**情况描述**

public class Test {

public static void main(String[] args) {

//在-128~127 之外的数

Integer num1 = 128; Integer num2 = 128;

System.out.println(num1==num2); //false

// 在-128~127 之内的数

Integer num3 = 9; Integer num4 = 9;

System.out.println(num3==num4); //true

}

}

解析原因：归结于java对于Integer与int的自动装箱与拆箱的设计，是一种模式：叫享元模式（flyweight）。   
　　加大对简单数字的重利用，Java定义在自动装箱时对于值从–128到127之间的值，它们被装箱为Integer对象后，会存在内存中被重用，始终只存在一个对象。   
 而如果超过了从–128到127之间的值，被装箱后的Integer对象并不会被重用，即相当于每次装箱时都新建一个 Integer对象。

**Integer源码解析**

　　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，源码如下：

public static Integer valueOf(String s, int radix) throws NumberFormatException {

return Integer.valueOf(parseInt(s,radix));

}

public static Integer valueOf(int i) {

assert IntegerCache.high >= 127;

if (i >= IntegerCache.low && i <= IntegerCache.high)

return IntegerCache.cache[i + (-IntegerCache.low)];

return new Integer(i);

}

IntegerCache是Integer的内部类，源码如下：

/\*\*

\* 缓存支持自动装箱的对象标识语义

\* -128和127（含）。

\*

\* 缓存在第一次使用时初始化。 缓存的大小

\* 可以由-XX：AutoBoxCacheMax = <size>选项控制。

\* 在VM初始化期间，java.lang.Integer.IntegerCache.high属性

\* 可以设置并保存在私有系统属性中

\*/

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) {

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);

}

high = h;

cache = new Integer[(high - low) + 1];

int j = low;

for(int k = 0; k < cache.length; k++)

cache[k] = new Integer(j++);

}

private IntegerCache() {}

}

<https://blog.csdn.net/chenliguan/article/details/53888018>

### 1.1.15 包装类，装箱和拆箱

#### 1.1.15.1 包装类（Wrapper Class）

　　包装类是针对于原生数据类型的包装。

　　因为有8个原生数据类型，所以对应有8个包装类。

　　所有的包装类（8个）都位于java.lang下。

　　Java中的8个包装类分别是：**Byte, Short, Integer, Long, Float, Double, Character, Boolean**，它们的使用方式都是一样的，可以实现原生数据类型与包装类型的双向转换。

　　下面以主要Integer类为例说明。

#### 1.1.15.2 Integer

　　Integer类将int类型的值包装到一个对象中。

　　Integer通过下面这个构造方法构造相应的整型数的对象：

　　public Integer(int value)；

　　public int intValue()方法则返回这个包装类所包装的整型值。

#### 1.1.15.3 自动装箱/拆箱（Autoboxing/unboxing）

　　JDK5.0的一个新特性是自动装箱和自动拆箱。

　　自动装箱/拆箱大大方便了基本类型数据和它们包装类的使用。

**自动装箱**：基本类型自动转为包装类（int >> Integer）

**自动拆箱**：包装类自动转为基本类型（Integer >> int）

　　比如下面的例子：

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Collection;  **public** **class** BoxTest  {  **public** **static** **void** main(String[] args)  {  Collection<Integer> c = **new** ArrayList<Integer>();    c.add(3);//将int类型的3转换为Integer类型并放到集合当中  }  } |

#### 缓存

　　下面这个程序很奇怪：

|  |
| --- |
| **public** **class** BoxTest2 {  **public** **static** **void** main(String[] args) {  Integer i1 = 200;  Integer i2 = 200;  **if** (i1 == i2) {  System.***out***.println("i1 == i2");  } **else** {  System.***out***.println("i1 != i2");  }  }  } |

<https://www.cnblogs.com/mengdd/archive/2013/01/21/2869921.html>

### 1.1.16 String、StringBuilder、StringBuffer

<https://www.cnblogs.com/dolphin0520/p/3778589.html>

### 1.1.16 重载和重写的区别

Java中的方法重载发生在同一个类里面两个或者是多个方法的方法名相同但是参数不同的情况。

与此相对，方法覆盖是说子类重新定义了父类的方法。方法覆盖必须有相同的方法名，参数列表和返回类型。覆盖者可能不会限制它所覆盖的方法的访问。

### 1.1.18 抽象类和接口有什么区别

Java提供和支持创建抽象类和接口。它们的实现有共同点，不同点在于：

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。

类可以实现很多个接口，但是只能继承一个抽象类

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。

抽象类可以在不提供接口方法实现的情况下实现接口。

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。

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。

接口是绝对抽象的，不可以被实例化。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。

也可以参考JDK8中抽象类和接口的区别

### 1.1.19 说说反射的用途及实现

反射的核心是JVM在运行时才动态加载类或调用方法/访问属性，它不需要事先（写代码的时候或编译期）知道运行对象是谁。

#### 1.1.19.1 Java反射框架主要提供以下功能：

1.在运行时判断任意一个对象所属的类；

2.在运行时构造任意一个类的对象；

3.在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用private方法）；

4.在运行时调用任意一个对象的方法

#### 1.1.19.2 主要用途 ：

1、反射最重要的用途就是开发各种通用框架。

#### 1.1.19.3 基本反射功能的实现(反射相关的类一般都在java.lang.relfect包里)：

1、获得Class对象

     使用Class类的forName静态方法

    直接获取某一个对象的class

  调用某个对象的getClass()方法

2、判断是否为某个类的实例

    用instanceof关键字来判断是否为某个类的实例

3、创建实例

     使用Class对象的newInstance()方法创建Class对象对应类的实例。

      先通过Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建实例。

4、获取方法

    getDeclaredMethods()

5、获取构造器信息

     getDeclaredMethods()

     getMethods()

     getMethod()

6、获取类的成员变量（字段）信息

     getFiled: 访问公有的成员变量

     getDeclaredField：所有已声明的成员变量。但不能得到其父类的成员变量

     getFileds和getDeclaredFields用法

7、调用方法

       invoke()

8、利用反射创建数组

    Array.newInstance()

#### 1.1.19.4 注意：

      由于反射会额外消耗一定的系统资源，因此如果不需要动态地创建一个对象，那么就不需要用反射。另外，反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。

### 1.1.20 说说自定义注解的场景及实现

### 1.1.21 HTTP请求的GET与POST方式的区别

GET在浏览器回退时是无害的，而POST会再次提交请求。

GET产生的URL地址可以被Bookmark，而POST不可以。

GET请求会被浏览器主动cache，而POST不会，除非手动设置。

GET请求只能进行url编码，而POST支持多种编码方式。

GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

GET请求在URL中传送的参数是有长度限制的，而POST么有。

对参数的数据类型，GET只接受ASCII字符，而POST没有限制。

GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。

GET参数通过URL传递，POST放在Request body中。

详见：

<https://mp.weixin.qq.com/s?__biz=MzI3NzIzMzg3Mw==&mid=100000054&idx=1&sn=71f6c214f3833d9ca20b9f7dcd9d33e4#rd>

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

### 1.1.22 Session与Cookie区别

#### 1.1.22.1 前言

HTTP是一种无状态的协议，为了分辨链接是谁发起的，需自己去解决这个问题。不然有些情况下即使是同一个网站每打开一个页面也都要登录一下。而Session和Cookie就是为解决这个问题而提出来的两个机制。

#### 1.1.22.2 应用场景

登录网站，今输入用户名密码登录了，第二天再打开很多情况下就直接打开了。这个时候用到的一个机制就是cookie。

session一个场景是购物车，添加了商品之后客户端处可以知道添加了哪些商品，而服务器端如何判别呢，所以也需要存储一些信息就用到了session。

**Cookie**

通俗讲，是访问某些网站后在本地存储的一些网站相关信息，下次访问时减少一些步骤。更准确的说法是：Cookies是服务器在本地机器上存储的小段文本并随每一个请求发送至同一服务器，是在客户端保持状态的方案。

Cookie的主要内容包括：名字，值，过期时间，路径和域。使用Fiddler抓包就可以看见，比方说我们打开百度的某个网站可以看到Headers包括Cookie，如下：



key, value形式。过期时间可设置的，如不设，则浏览器关掉就消失了，存储在内存当中，否则就按设置的时间来存储在硬盘上的，过期后自动清除，比方说开关机关闭再打开浏览器后他都会还存在，前者称之为Session cookie 又叫 transient cookie，后者称之为Persistent cookie 又叫 permenent cookie。路径和域就是对应的域名，a网站的cookie自然不能给b用。

**Session**

存在服务器的一种用来存放用户数据的类HashTable结构。

浏览器第一次发送请求时，服务器自动生成了一HashTable和一Session ID来唯一标识这个HashTable，并将其通过响应发送到浏览器。浏览器第二次发送请求会将前一次服务器响应中的Session ID放在请求中一并发送到服务器上，服务器从请求中提取出Session ID，并和保存的所有Session ID进行对比，找到这个用户对应的HashTable。

一般这个值会有个时间限制，超时后毁掉这个值，默认30分钟。

当用户在应用程序的 Web页间跳转时，存储在 Session 对象中的变量不会丢失而是在整个用户会话中一直存在下去。

Session的实现方式和Cookie有一定关系。建立一个连接就生成一个session id，打开几个页面就好几个了，这里就用到了Cookie，把session id存在Cookie中，每次访问的时候将Session id带过去就可以识别了.

### 1.1.23 列出自己常用的JDK包

JDK所提供的所有标准Java类都存放在Java包中，如java.lang包中包含了运行Java必不可少的系统类。由于系统会自动将java.lang引入，所以不需要在源文件中用import语句来显示地引入这个包。另外，Java java.util和java.io是必须提供的标准包，在JDK中常用的包有以下几种：

1.java.lang：语言包

2.java.util：实用包

3.java.awt：抽象窗口工具包

4.javax.swing：轻量级的窗口工具包，这是目前使用最广泛的GUI程序设计包

5.java.io：输入输出包

6.java.net：网络函数包

7.java.applet：编制applet用到的包（目前编制applet程序时，更多的是使用swing中的JApplet类）。

下面分别对这几个包做下介绍：

**①.java.lang：语言包**

这是Java语言的核心包，系统自动将这个包引入到用户程序，该包中主要类有：

1.**object类**：它是所有类的父类，其中定义的方法其它类都可以使用。

2.数据类型包装类：简单的数据类型的类包装，包括Integer、Float、Boolean等。

3.**数学类Math**：提供常量和数学函数，包括E和PI常数及abs()、sin()、cos()、min()、max()、random()等方法，这些常量和方法都是静态的。

4.字符串类**String和StringBuffer**类。

5.系统和运行时类：System类提供一个独立于具体计算机系统资源的编程界面；Runtime类可以直接访问运行时资源。

6.操作类 ：**Class和ClassLoader类**。类Class提供了对象运行时的若干信息，ClassLoader是一个抽象类，它提供了将类名转换成文件名并在文件系统中查找并装载该文件的方法。

7.线程类：**Thread**类。Java是一个多线程环境，主要有Thread（线程建立）、ThreadDeath（线程结束后的清理操作）、ThreadGroup（组织一组线程）和Runnable（建立线程的交互工具）等类。

8.错误和异常处理类：**Throwable**（所有错误和异常处理的父类），Exception（处理异常，需要用户捕获处理）和Error（处理硬件错误，不要求用户捕获处理）。

9.过程类**Process**：它支持系统过程，当实用类Runtime执行系统命令时，会建立处理系统过程的Process类。

**②.实用包**

实用包提供了各种实用功能的类，主要包括日期类、数据结构类和随机数类等。

1.日期类：包括Date（获取日期和时间）、Calendar（抽象类，日历类）和GregorianCalendar（Calendar类的子类）类。

2.数据结构类：包括链表类LinkedList、向量类Vector、栈类Stack和散列表类Hashtable等。

3.随机数类Random：它封装了Math类中的random方法，并提供了更多的辅助功能。

**③抽象窗口工具包**

Java的java.awt提供了绘图和图像类，主要用于编写GUI程序，包括按钮、标签等常用组件以及相应的事件类。

1.组件类：包括Button，Panel，Label，Choice等类，用于设计图形界面。

2事件包：在java.awt.event中包括各种事件处理的接口和类

3.颜色包：在java.awt.color中提供用于颜色的类。

4.字体包：在java.awt.font中提供用于字体相关的接口和类。

**④输入输出包**

java.io提供了系统输入输出类和接口，只要包括输入流类InputStream和输出流OutputStream就可以实现文件的输入输出、管道的数据传输以及网络数据传输的功能

**⑤网络函数包**

java.net提供了实现网络应用程序的类，主要包括用于实现Socket通信的Socket类，此外还提供了便于处理URL的类

**⑥applet包**

java.applet是专为创建Applet程序提供的包，它包含了基本的applet类和通信类，目前基本上被JApplet类所代替。

### 1.1.24 MVC设计思想

mvc(model view controller)

#### 1.1.24.1 什么是mvc?

是一种软件架构的思想，将一个软件按照模型、视图、控制器进行划分。其中，模型用来封装业务逻辑，视图用来实现表示逻辑，控制器用来协调模型与视图(视图要通过控制器来调用模型，模型返回的处理结果也要先交给控制器，由控制器来选择合适的视图来显示 处理结果)。

1)模型: 业务逻辑包含了业务数据的加工与处理以及相应的基础服务(为了保证业务逻辑能够正常进行的事务、安全、权限、日志等等的功能模块)

2)视图:展现模型处理的结果；另外，还要提供相应的操作界面，方便用户使用。

3)控制器:视图发请求给控制器，由控制器来选择相应的模型来处理；模型返回的结果给控制器，由控制器选择合适的视图。

#### 1.1.24.2 为什么要使用mvc?

1)使用mvc的思想来设计一个软件，最根本的原因是为了实现模型的复用:

a,模型不用关心处理结果如何展现。

            比如，模型返回一些数据，然后交给不同的视图来展现(表格的方式、图形的方式等等)。

b,可以使用不同的视图来访问同一个模型。

2)代码的维护性更好

     修改模型不会影响到视图，反过来，修改视图，也不会影响到模型。

3)方便测试

       比如，将业务逻辑代码写在servlet里面，需要部署到容器上，然后才能测试。而将业务逻辑代码写在java类里面，可以直接用main()测试（不依赖容器）。

#### 1.1.24.3 如何使用mvc来开发一个web应用(javaee)。

1)使用java类来实现模型，使用jsp来实现视图，使用servlet或者filter来实现控制器。

注意：模型也可以使用一些企业级的Java组件，比如，ejb，spring容器管理的JavaBean来实现。视图也可以使用jsf，模板技术（freemarker等）。

2)请求都发送给控制器(servlet),控制器依据请求的

不同调用相应的模型(java类)来处理；模型返回的处理结果给控制器(servlet)，然后控制器选择合适的视图(jsp)来展现处理结果。

#### 1.1.24.4 mvc的缺点

使用mvc，会增加代码量、相应地也会增加软件开发的成文，设计的难度也会增加。

原文：https://blog.csdn.net/daijin888888/article/details/51169156

### 1.1.25 hashCode和equals方法的区别与联系

**hashCode()和equals()方法的重要性体现：**

**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。

区别和详见：

<https://blog.csdn.net/haobaworenle/article/details/53819838>

### 1.1.26 Java序列化和反序列化，如何实现，请解释Serializable 接口的作用

<https://www.cnblogs.com/xdp-gacl/p/3777987.html>

### 1.1.30 Java 8有哪些新特性

### 1.1.31 Java的三种代理模式

<https://www.cnblogs.com/cenyu/p/6289209.html>

LinkedBlockingQueue

PriorityQueue

ConcurrentLinkedQueue

## ****1.2、Java常见集合****

<https://www.cnblogs.com/ysocean/p/6555373.html>

### 1.2.1 HashMap实现原理及源码分析

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

### 1.2.2 List 和 Set 区别

### 1.2.3 Set和hashCode以及equals方法的联系

### 1.2.4 List 和 Map 区别

### 1.2.5 Arraylist 与 LinkedList 区别

### 1.2.6 ArrayList 与 Vector 区别

### 1.2.7 HashMap 和 Hashtable 的区别

### 1.2.8 HashSet 和 HashMap 区别

### 1.2.9 HashMap 和 ConcurrentHashMap 的区别

### 1.2.10 HashMap 的工作原理及代码实现，什么时候用到红黑树

### 1.2.11 多线程情况下HashMap死循环的问题

### 1.2.12 HashMap出现Hash DOS攻击的问题

### 1.2.13 ConcurrentHashMap 的工作原理及代码实现，统计所有的元素个数

### 1.2.14 手写简单的HashMap

### 1.2.15 看过那些Java集合类的源码

## ****1.3、进程和线程****

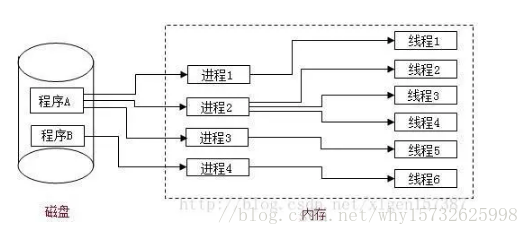
### 1.3.1 线程和进程的概念、并行和并发的概念

#### 1.3.1.1 进程与线程

1、 进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，是操作系统结构的基础。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

2、线程，有时被称为轻量级进程（Lightweight Process,LWP），是程序执行流的最小单元。线程是程序中一个单一的顺序控制流程。进程内一个相对独立、可调度的执行单元，是系统独立调度和分派CPU的基本单位，也指运行中的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为多线程。

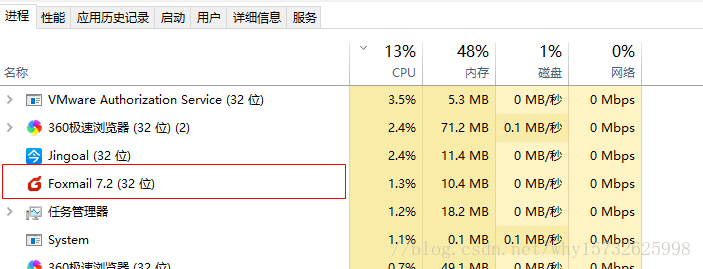
● 进程和线程的关系

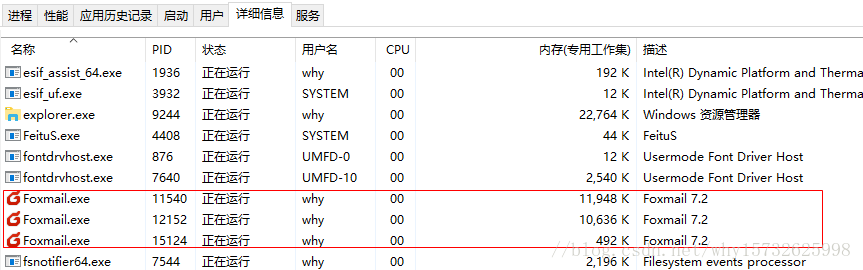


● 理解

当我在电脑上打开一个foxmail，开启了一个进程，打开一个浏览器，开启了一个进程，当我使用foxmail的时候，收邮件是一个线程，发邮件是一个线程，写邮件又是一个线程，这几个线程同时为进程Foxmail工作，完成Foxmail的全部功能。

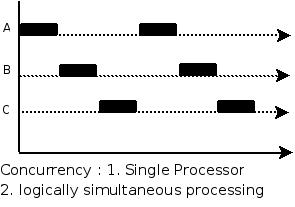
注意：图片中的下面三个Foxmail线程不一定是我所说的功能，但是就是这个效果。





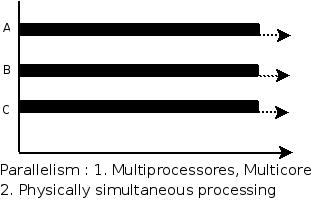
#### 1.3.1.2 并行和并发

1、并发（concurrency）：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干端，使多个进程快速交替的执行。



如上图所示，并发就是只有一个CPU资源，程序（或线程）之间要竞争得到执行机会。图中的第一个阶段，在A执行的过程中，B、C不会执行，因为这段时间内这个CPU资源被A竞争到了，同理，第二阶段只有B在执行，第三阶段只有C在执行。其实，并发过程中，A、B、C并不是同事进行的（微观角度），但又是同时进行的（宏观角度）

2、并行（parallellism）:指在同一时刻，有多条指令在多个处理器上同时执行



如图所示，在同一时刻，ABC都是同时执行（微观、宏观）

通过多线程实现并发，并行：

➤ java中的Thread类定义了多线程，通过多线程可以实现并发或并行。

➤ 在CPU比较繁忙，资源不足的时候（开启了很多进程），操作系统只为一个含有多线程的进程分配仅有的CPU资源，这些线程就会为自己尽量多抢时间片，这就是通过多线程实现并发，线程之间会竞争CPU资源争取执行机会。

➤ 在CPU资源比较充足的时候，一个进程内的多线程，可以被分配到不同的CPU资源，这就是通过多线程实现并行。

➤ 至于多线程实现的是并发还是并行？上面所说，所写多线程可能被分配到一个CPU内核中执行，也可能被分配到不同CPU执行，分配过程是操作系统所为，不可人为控制。所以，如果有人问我我所写的多线程是并发还是并行的？我会说，都有可能。

➤ 不管并发还是并行，都提高了程序对CPU资源的利用率，最大限度地利用CPU资源

<https://blog.csdn.net/why15732625998/article/details/79625556>

参考链接：

<http://blog.csdn.net/xlgen157387/article/details/77920497>

[http://blog.csdn.net/qq\_33290787/article/details/51790605](http://blog.csdn.net/qq_33290787/article/details/51790605%20)

<http://blog.csdn.net/java_zero2one/article/details/51477791>

### 1.3.2 线程如何同步（七种同步方式）

同学回答说 synchronized 方法或代码块！xxx官似乎不太满意！

只有多个 synchronized 代码块使用的是同一个监视器对象，这些 synchronized 代码块之间才具有线程互斥的效果，假如 a 代码块用 obj1 作为监视器对象，假如 b 代码块用 obj2 作为监视器对象，那么，两个并发的线程可以同时分别进入这两个代码块中。 …这里还可以分析一下同步的原理。

对于同步方法的分析，所用的同步监视器对象是 this

接着对于静态同步方法的分析，所用的同步监视器对象是该类的 Class 对象接着对如何实现代码块与方法的同步进行分析。

为何要使用同步？

java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），

将会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，

从而保证了该变量的唯一性和准确性。

**为何要使用同步？**

java允许多线程并发控制，当多个线程同时操作一个可共享的资源变量时（如数据的增删改查），将会导致数据不准确，相互之间产生冲突，因此加入同步锁以避免在该线程没有完成操作之前，被其他线程的调用，从而保证了该变量的唯一性和准确性。

#### 1.3.2.1 同步方法

即有synchronized关键字修饰的方法。

由于java的每个对象都有一个内置锁，当用此关键字修饰方法时， 内置锁会保护整个方法。在调用该方法前，需要获得内置锁，否则就处于阻塞状态。

代码如：

public synchronized void save(){}

**注： 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**

#### 1.3.2.2 同步代码块

  即有synchronized关键字修饰的语句块。   
    被该关键字修饰的语句块会自动被加上内置锁，从而实现同步

    代码如：   
    synchronized(object){     }

 注：同步是一种高开销的操作，因此应该尽量减少同步的内容。 通常没有必要同步整个方法，使用synchronized代码块同步关键代码即可。

|  |
| --- |
| /\*\*  \* 线程同步的运用  \*  \* **@author** XIEHEJUN  \* https://www.cnblogs.com/XHJT/p/3897440.html  \*/  **public** **class** SynchronizedThread {  **class** Bank {  **private** **int** account = 100;  **public** **int** getAccount() {  **return** account;  }  /\*\*  \* 用同步方法实现  \* **@param** money  \*/  **public** **synchronized** **void** save(**int** money) {  account += money;  }  /\*\*  \* 用同步代码块实现  \* **@param** money  \*/  **public** **void** save1(**int** money) {  **synchronized** (**this**) {  account += money;  }  }  }  **class** NewThread **implements** Runnable {  **private** Bank bank;  **public** NewThread(Bank bank) {  **this**.bank = bank;  }  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 10; i++) {  // bank.save1(10);  bank.save(10);  System.***out***.println(i + "账户余额为：" + bank.getAccount());  }  }  }  /\*\*  \* 建立线程，调用内部类  \*/  **public** **void** useThread() {  Bank bank = **new** Bank();  NewThread new\_thread = **new** NewThread(bank);  System.***out***.println("线程1");  Thread thread1 = **new** Thread(new\_thread);  thread1.start();  System.***out***.println("线程2");  Thread thread2 = **new** Thread(new\_thread);  thread2.start();  }  **public** **static** **void** main(String[] args) {  SynchronizedThread st = **new** SynchronizedThread();  st.useThread();  }  } |

#### 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   
        例如：   
        在上面的例子当中，只需在account前面加上volatile修饰，即可实现线程同步。   
        代码实例：

|  |
| --- |
| //只给出要修改的代码，其余代码与上同  class Bank {  //需要同步的变量加上volatile  private volatile int account = 100;  public int getAccount() {  return account;  }  //这里不再需要synchronized  public void save(int money) {  account += money;  }  ｝ |

**注：多线程中的非同步问题主要出现在对域的读写上，如果让域自身避免这个问题，则就不需要修改操作该域的方法。**

**用final域，有锁保护的域和volatile域可以避免非同步的问题。**

#### 1.3.2.4 使用重入锁实现线程同步

在JavaSE5.0中新增了一个java.util.concurrent包来支持同步。ReentrantLock类是可重入、互斥、实现了Lock接口的锁，它与使用synchronized方法和快具有相同的基本行为和语义，并且扩展了其能力，ReenreantLock类的常用方法有：

ReentrantLock() : 创建一个ReentrantLock实例

lock() : 获得锁

unlock() : 释放锁

注：ReentrantLock()还有一个可以创建公平锁的构造方法，但由于能大幅度降低程序运行效率，不推荐使用

#### 1.3.2.5 使用局部变量实现线程同步

如果使用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响。

ThreadLocal 类的常用方法

ThreadLocal() : 创建一个线程本地变量

get() : 返回此线程局部变量的当前线程副本中的值

initialValue() : 返回此线程局部变量的当前线程的"初始值"

set(T value) : 将此线程局部变量的当前线程副本中的值设置为value

#### 1.3.2.6 使用阻塞队列实现线程同步

使用LinkedBlockingQueue<E>来实现线程的同步

#### 1.3.2.7 使用原子变量实现线程同步

util.concurrent.atomic

<https://www.cnblogs.com/XHJT/p/3897440.html>

### 1.3.3 创建线程的方式及实现（三种方式）

Java使用Thread类代表线程，所有的线程对象都必须是Thread类或其子类的实例。Java可以用三种方式来创建线程，如下所示：

1）继承Thread类创建线程

2）实现Runnable接口创建线程

3）使用Callable和Future创建线程

#### 1.3.3.1 继承Thread类创建线程

通过继承Thread类来创建并启动多线程的一般步骤如下

1】定义Thread类的子类，并重写该类的run()方法，该方法的方法体就是线程需要完成的任务，run()方法也称为线程执行体。

2】创建Thread子类的实例，也就是创建了线程对象

3】启动线程，即调用线程的start()方法

代码实例

|  |
| --- |
| **public** **class** MyThread **extends** Thread {// 继承Thread类  @Override  **public** **void** run() {  **super**.run();  // 重写run方法  }  }  **public** **class** Main {  **public** **static** **void** main(String[] args) {  **new** MyThread().start();// 创建并启动线程  }  } |

#### 1.3.3.2 实现Runnable接口创建线程

通过实现Runnable接口创建并启动线程一般步骤如下：

1】定义Runnable接口的实现类，一样要重写run()方法，这个run（）方法和Thread中的run()方法一样是线程的执行体

2】创建Runnable实现类的实例，并用这个实例作为Thread的target来创建Thread对象，这个Thread对象才是真正的线程对象

3】第三部依然是通过调用线程对象的start()方法来启动线程

代码实例：

|  |
| --- |
| **public** **class** MyThread2 **implements** Runnable {  @Override  **public** **void** run() {  // 实现Runnable接口  }  }  **public** **class** Main {  **public** **static** **void** main(String[] args) {  // 创建并启动线程  MyThread2 myThread = **new** MyThread2();  Thread thread = **new** Thread(myThread);  thread.start();  // 或者 new Thread(new MyThread2()).start();  }  } |

#### 1.3.3.3 使用Callable和Future创建线程

和Runnable接口不一样，Callable接口提供了一个call（）方法作为线程执行体，call()方法比run()方法功能要强大。

》call()方法可以有返回值

》call()方法可以声明抛出异常

Java5提供了Future接口来代表Callable接口里call()方法的返回值，并且为Future接口提供了一个实现类FutureTask，这个实现类既实现了Future接口，还实现了Runnable接口，因此可以作为Thread类的target。在Future接口里定义了几个公共方法来控制它关联的Callable任务。

>boolean cancel(boolean mayInterruptIfRunning)：视图取消该Future里面关联的Callable任务

>V get()：返回Callable里call（）方法的返回值，调用这个方法会导致程序阻塞，必须等到子线程结束后才会得到返回值

>V get(long timeout,TimeUnit unit)：返回Callable里call（）方法的返回值，最多阻塞timeout时间，经过指定时间没有返回抛出TimeoutException

>boolean isDone()：若Callable任务完成，返回True

>boolean isCancelled()：如果在Callable任务正常完成前被取消，返回True

介绍了相关的概念之后，创建并启动有返回值的线程的步骤如下：

1】创建Callable接口的实现类，并实现call()方法，然后创建该实现类的实例（从java8开始可以直接使用Lambda表达式创建Callable对象）。

2】使用FutureTask类来包装Callable对象，该FutureTask对象封装了Callable对象的call()方法的返回值

3】使用FutureTask对象作为Thread对象的target创建并启动线程（因为FutureTask实现了Runnable接口）

4】调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

代码实例：cao

#### 三种创建线程方法对比

实现Runnable和实现Callable接口的方式基本相同，不过是后者执行call()方法有返回值，后者线程执行体run()方法无返回值，因此可以把这两种方式归为一种这种方式与继承Thread类的方法之间的差别如下：

1、线程只是实现Runnable或实现Callable接口，还可以继承其他类。

2、这种方式下，多个线程可以共享一个target对象，非常适合多线程处理同一份资源的情形。

3、但是编程稍微复杂，如果需要访问当前线程，必须调用Thread.currentThread()方法。

4、继承Thread类的线程类不能再继承其他父类（Java单继承决定）。

**注：一般推荐采用实现接口的方式来创建多线程**

### 1.3.4 进程间通信的方式

<https://blog.csdn.net/ls5718/article/details/51878770>

### 1.3.5 scanvenge gc 和 full GC

<https://blog.csdn.net/ShyTan/article/details/81036950>

### 1.3.6 说说 CountDownLatch、CyclicBarrier 原理和区别

### 1.3.7 说说 Semaphore 原理

### 1.3.8 说说 Exchanger 原理

### 1.3.9 ThreadLocal原理 ThreadLocal 出现OOM 出现的深层次原理

### 1.3.10 讲讲线程池的实现原理

### 1.3.11 线程池的几种实现方式

### 1.3.12 线程的生命周期，状态是如何转移的

## ****1.4、锁机制****

### 1.4.1 说说线程安全问题，什么是线程安全，如何保证线程安全

### 1.4.2 重入锁的概念，重入锁为什么可以防止死锁

### 1.4.3 产生死锁的四个条件（互斥、请求与保持、不剥夺、循环等待）

### 1.4.4 如何检查死锁（通过jConsole检查死锁）

### 1.4.5 volatile 实现原理（禁止指令重排、刷新内存）

### 1.4.6 synchronized 实现原理（对象监视器）

### 1.4.7 synchronized 与 lock 的区别

### 1.4.8 AQS同步队列

### 1.4.9 CAS 无锁的概念、乐观锁和悲观锁

### 1.4.10 常见的原子操作类

### 1.4.11 什么是ABA问题，出现ABA问题JDK是如何解决的

### 1.4.12 乐观锁的业务场景及实现方式

### 1.4.13 Java 8并法包下常见的并发类

### 1.4.14 偏向锁、轻量级锁、重量级锁、自旋锁的概念

## ****1.5、JVM****

### 1.5.1 JVM内存管理机制和垃圾回收机制

<https://blog.csdn.net/sinat_35821285/article/details/80236377>

<https://blog.csdn.net/aijiudu/article/details/72991993>

### 1.5.2 Java什么时候进行垃圾回收,垃圾回收的执行流程

<https://blog.csdn.net/jidong2622/article/details/78147364>

1.5.3 JVM运行时内存区域划分

### 1.5.4 内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

### 1.5.5 如何判断对象是否可以回收或存活

### 1.5.6 常见的GC回收算法及其含义

### 1.5.7 常见JVM性能监控和故障处理工具类

jps、jstat、jmap、jinfo、jconsole等

### 1.5.8 JVM如何设置参数

### 1.5.9 JVM性能调优

### 1.5.10 双亲委派模型、一个类的生命周期、类是如何加载到JVM中的

### 1.5.11 类加载的过程：加载、验证、准备、解析、初始化

### 1.5.12 Java内存模型JMM

## ****1.6、设计模式****

### 1.6.1 常见的设计模式

### 1.6.2 设计模式的的六大原则及其含义

### 1.6.3 常见的单例模式 各种实现方式 优缺点

### 1.6.4 设计模式在实际场景中的应用

### 1.6.5 Spring中用到了哪些设计模式

### 1.6.6 MyBatis中用到了哪些设计模式

### 1.6.7 你项目中有使用哪些设计模式

### 1.6.8 说说常用开源框架中设计模式使用分析

## ****1.7、数据结构****

### 1.7.1 树（二叉查找树、平衡二叉树、红黑树、B树、B+树）

### 1.7.2 深度有限算法、广度优先算法

### 1.7.3 克鲁斯卡尔算法、普林母算法、迪克拉斯算法

### 1.7.4 什么是一致性Hash及其原理、Hash环问题

### 1.7.5 常见的排序算法和查找算法：快排、折半查找、堆排序等

## ****1.8、网络/IO基础****

### 1.8.1 BIO、NIO、AIO的概念

### 1.8.2 计算机网络

<https://mp.weixin.qq.com/s?__biz=MzU4NDQ4MzU5OA==&mid=2247484289&idx=1&sn=6b556843c60aac9a17b0e7c2e3cd6bca&scene=19#wechat_redirect>

### 1.8.3 什么是长连接和短连接

### 1.8.4 Http1.0和2.0相比有什么区别，可参考《[Http 2.0](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484611&idx=1&sn=66c875392eedff8150633ddcd5d83e7a&chksm=e9c5fd72deb274648a607b9bc39bac34adadd768577b77354f6dc85422691605e210b69eeb7b&scene=21#wechat_redirect)》

### 1.8.5 Https的基本概念

### 1.8.6 三次握手和四次挥手、为什么挥手需要四次

### 1.8.7 从游览器中输入URL到页面加载的发生了什么？可参考《[从输入URL到页面加载发生了什么](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483724&idx=1&sn=e58dd30d124971c795584e8673d6cc71&chksm=e9c5f8fddeb271ebebbb6c350ed1abc252f1f26b4f35c4ce36e10bde9659a37520feabed2290&scene=21#wechat_redirect)》

# **二、数据存储和消息队列**

## 2.1**、**MySQL

### 2.1.1 面试中有哪些经典的数据库问题？:

<https://mp.weixin.qq.com/s/ns9T02-zsO9I3tdG3XJbcw>

### 2.1.2 为什么用自增列作为主键

<https://blog.csdn.net/wukong_666/article/details/54982712>

### 2.1.3 为什么使用数据索引能提高效率

<https://blog.csdn.net/u011534095/article/details/48552877>

### 2.1.4 B+树索引和哈希索引的区别

<http://www.cnblogs.com/heiming/p/5865101.html>

### 2.1.5 哈希索引的优势

<https://blog.csdn.net/injurooioo/article/details/73506062>

### 2.1.6 哈希索引不适用的场景

<https://blog.csdn.net/sunjin9418/article/details/80334142>

### 2.1.7 B树和B+树的区别

### 2.1.8 为什么说B+比B树更适合实际应用中操作系统的文件索引和数据库索引

### 2.1.9 MySQL联合索引

### 2.1.10 表分区与分表的区别 表分区好处 分区表的限制因素

### 2.1.11 MySQL支持的分区类型有哪些

### 2.1.12 四种隔离级别

### 2.1.13 关于MVVC 在MVCC并发控制中，读操作可以分成两类

### 2.1.14 行级锁定的优缺点

### 2.1.14 MySQL优化

### 2.1.15 key和index的区别

### 2.1.16 Mysql 中 MyISAM 和 InnoDB 的区别有哪些？

### 2.1.17 数据库表创建注意事项

### 2.1.18 MySQL 索引使用的注意事项

### 2.1.19 DDL、DML、DCL分别指什么

### 2.1.20 explain命令

### 2.1.21 left join，right join，inner join

### 2.1.22 数据库事物ACID（原子性、一致性、隔离性、持久性）

### 2.1.23 事物的隔离级别（读未提交、读以提交、可重复读、可序列化读）

### 2.1.24 脏读、幻读、不可重复读

### 2.1.25 数据库的几大范式

### 2.1.26 数据库常见的命令

### 2.1.27 说说分库与分表设计

### 2.1.28 分库与分表带来的分布式困境与应对之策（如何解决分布式下的分库分表，全局表？）

### 2.1.29 说说 SQL 优化之道

### 2.1.30 MySQL遇到的死锁问题、如何排查与解决

### 2.1.31 存储引擎的 InnoDB与MyISAM区别，优缺点，使用场景

### 2.1.32 索引类别（B+树索引、全文索引、哈希索引）、索引的原理

### 2.1.33 什么是自适应哈希索引（AHI）

### 2.1.34 为什么要用 B+tree作为MySQL索引的数据结构

### 2.1.35 聚集索引与非聚集索引的区别

### 2.1.36 遇到过索引失效的情况没，什么时候可能会出现，如何解决

### 2.1.37 limit 20000 加载很慢怎么解决

### 2.1.38 如何选择合适的分布式主键方案

### 2.1.39 选择合适的数据存储方案

### 2.1.40 常见的几种分布式ID的设计方案

### 2.1.41 常见的数据库优化方案，在你的项目中数据库如何进行优化的

## ****2.2、Redis****

### ****2.2.1**** Redis 有哪些数据类型

可参考《[Redis常见的5种不同的数据类型详解](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483987&idx=1&sn=5c5e4cd5bc73a7e6f84e5d6adfab0935&chksm=e9c5fbe2deb272f4b5b75bd2ac92bb27950452623ec83c0e1add7e30c773160421fab1571680&scene=21#wechat_redirect)》

### ****2.2.1**** Redis 内部结构

### ****2.2.1**** Redis 使用场景

### ****2.2.1**** Redis 持久化机制

可参考《[使用快照和AOF将Redis数据持久化到硬盘中](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483992&idx=1&sn=8f554bc490c4db1a78a30144f873e911&chksm=e9c5fbe9deb272fff47483c241e6d2a7aae99dc8f6fe9fee31f2dd214d0cf81b33d51f7a7dbe&scene=21#wechat_redirect)》

### ****2.2.1**** Redis 集群方案与实现

### ****2.2.1**** Redis 为什么是单线程的？

### ****2.2.1**** 缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

### ****2.2.1**** 使用缓存的合理性问题

### ****2.2.1**** Redis常见的回收策略

## ****2.3、消息队列****

### 2.3.1 消息队列的使用场景

### 2.3.2 消息的重发补偿解决思路

### 2.3.3 消息的幂等性解决思路

### 2.3.4 消息的堆积解决思路

### 2.3.5 自己如何实现消息队列

### 2.3.6 如何保证消息的有序性

# ****三、开源框架和容器****

## ****3.1、Spring****

<http://blog.51cto.com/luecsc/1964056>

### 3.1.1 Servlet的生命周期

### 3.1.2 转发与重定向的区别

### 3.1.3 BeanFactory 和 ApplicationContext 有什么区别

### 3.1.4 Spring Bean 的生命周期

### 3.1.5 Spring IOC 如何实现

### 3.1.6 Spring中Bean的作用域，默认的是哪一个

### 3.1.7 说说 Spring AOP、Spring AOP 实现原理

### 3.1.8 动态代理（CGLib 与 JDK）、优缺点、性能对比、如何选择

### 3.1.9 Spring 事务实现方式、事务的传播机制、默认的事务类别

### 3.1.10 Spring 事务底层原理

### 3.1.11 Spring事务失效（事务嵌套），

JDK动态代理给Spring事务埋下的坑，可参考《[JDK动态代理给Spring事务埋下的坑！](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484940&idx=1&sn=0a0a7198e96f57d610d3421b19573002&chksm=e9c5ffbddeb276ab64ff3b3efde003193902c69acda797fdc04124f6c2a786255d58817b5a5c&scene=21#wechat_redirect)》

### 3.1.12 如何自定义注解实现功能

### 3.1.13 Spring MVC 运行流程

### 3.1.14 Spring MVC 启动流程

### 3.1.15 Spring 的单例实现原理

### 3.1.16 Spring 框架中用到了哪些设计模式

### 3.1.17 Spring 其他产品（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）

### 3.1.18 有没有用到Spring Boot，Spring Boot的认识、原理

* 可参考《[为什么会有Spring](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484822&idx=1&sn=6fbee2a12b31b6102a18d3725671d41b&chksm=e9c5fc27deb275319641c3f30d168b85c7c196fd276d47efa35046b5dc54f5b77174c5bf8808&scene=21#wechat_redirect)》
* 可参考《[为什么会有Spring AOP](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484827&idx=1&sn=b9d82f3fced6a875f8dfc22e5849b28e&chksm=e9c5fc2adeb2753c516ef8fc959c0c9dd84ccacaa40473b64bc58b5137c30562a0b45803ba8e&scene=21#wechat_redirect)》

## 3.2、MyBatis

### 3.2.1 #{}和${}的区别是什么？

注：这道题是面试官面试我同事的。

${}是Properties文件中的变量占位符，它可以用于标签属性值和sql内部，属于静态文本替换，比如${driver}会被静态替换为com.mysql.jdbc.Driver。#{}是sql的参数占位符，Mybatis会将sql中的#{}替换为?号，在sql执行前会使用PreparedStatement的参数设置方法，按序给sql的?号占位符设置参数值，比如ps.setInt(0, parameterValue)，#{item.name}的取值方式为使用反射从参数对象中获取item对象的name属性值，相当于param.getItem().getName()。

### 3.2.2 Xml映射文件中，除了常见的select|insert|updae|delete标签之外，还有哪些标签？

注：这道题是京东面试官面试我时问的。

答：还有很多其他的标签，<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态sql的9个标签，trim|where|set|foreach|if|choose|when|otherwise|bind等，其中<sql>为sql片段标签，通过<include>标签引入sql片段，<selectKey>为不支持自增的主键生成策略标签。

### 3.2.3 最佳实践中，通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

注：这道题也是京东面试官面试我时问的。

Dao接口，就是人们常说的Mapper接口，接口的全限名，就是映射文件中的namespace的值，接口的方法名，就是映射文件中MappedStatement的id值，接口方法内的参数，就是传递给sql的参数。Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MappedStatement，举例：com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到namespace为com.mybatis3.mappers.StudentDao下面id = findStudentById的MappedStatement。在Mybatis中，每一个<select>、<insert>、<update>、<delete>标签，都会被解析为一个MappedStatement对象。

Dao接口里的方法，是不能重载的，因为是全限名+方法名的保存和寻找策略。

Dao接口的工作原理是JDK动态代理，Mybatis运行时会使用JDK动态代理为Dao接口生成代理proxy对象，代理对象proxy会拦截接口方法，转而执行MappedStatement所代表的sql，然后将sql执行结果返回。

### 3.2.4 Mybatis是如何进行分页的？分页插件的原理是什么？

Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页，可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

举例：select \* from student，拦截sql后重写为：select t.\* from （select \* from student）t limit 0，10

### 3.2.5 简述Mybatis的插件运行原理，以及如何编写一个插件。

Mybatis仅可以编写针对ParameterHandler、ResultSetHandler、StatementHandler、Executor这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这4种接口对象的方法时，就会进入拦截方法，具体就是InvocationHandler的invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

实现Mybatis的Interceptor接口并复写intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

### 3.2.6 Mybatis执行批量插入，能返回数据库主键列表吗？

能，JDBC都能，Mybatis当然也能。

### 3.2.7 Mybatis动态sql是做什么的？都有哪些动态sql？能简述一下动态sql的执行原理不？

Mybatis动态sql可以让我们在Xml映射文件内，以标签的形式编写动态sql，完成逻辑判断和动态拼接sql的功能，Mybatis提供了9种动态sql标签trim|where|set|foreach|if|choose|when|otherwise|bind。

其执行原理为，使用OGNL从sql参数对象中计算表达式的值，根据表达式的值动态拼接sql，以此来完成动态sql的功能。

### 3.2.8 Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式？

第一种是使用<resultMap>标签，逐一定义列名和对象属性名之间的映射关系。第二种是使用sql列的别名功能，将列别名书写为对象属性名，比如T\_NAME AS NAME，对象属性名一般是name，小写，但是列名不区分大小写，Mybatis会忽略列名大小写，智能找到与之对应对象属性名，你甚至可以写成T\_NAME AS NaMe，Mybatis一样可以正常工作。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

### 3.2.9 Mybatis能执行一对一、一对多的关联查询吗？都有哪些实现方式，以及它们之间的区别。

能，Mybatis不仅可以执行一对一、一对多的关联查询，还可以执行多对一，多对多的关联查询，多对一查询，其实就是一对一查询，只需要把selectOne()修改为selectList()即可；多对多查询，其实就是一对多查询，只需要把selectOne()修改为selectList()即可。

关联对象查询，有两种实现方式，一种是单独发送一个sql去查询关联对象，赋给主对象，然后返回主对象。另一种是使用嵌套查询，嵌套查询的含义为使用join查询，一部分列是A对象的属性值，另外一部分列是关联对象B的属性值，好处是只发一个sql查询，就可以把主对象和其关联对象查出来。

那么问题来了，join查询出来100条记录，如何确定主对象是5个，而不是100个？其去重复的原理是<resultMap>标签内的<id>子标签，指定了唯一确定一条记录的id列，Mybatis根据<id>列值来完成100条记录的去重复功能，<id>可以有多个，代表了联合主键的语意。

同样主对象的关联对象，也是根据这个原理去重复的，尽管一般情况下，只有主对象会有重复记录，关联对象一般不会重复。

举例：下面join查询出来6条记录，一、二列是Teacher对象列，第三列为Student对象列，Mybatis去重复处理后，结果为1个老师6个学生，而不是6个老师6个学生。

 t\_id     t\_name           s\_id

|          1 | teacher      |      38 |  
|          1 | teacher      |      39 |  
|          1 | teacher      |      40 |  
|          1 | teacher      |      41 |  
|          1 | teacher      |      42 |  
|          1 | teacher      |      43 |

### 3.2.10 Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？

Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

### 3.2.11 Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？

不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复；毕竟namespace不是必须的，只是最佳实践而已。

原因就是namespace+id是作为Map<String, MappedStatement>的key使用的，如果没有namespace，就剩下id，那么，id重复会导致数据互相覆盖。有了namespace，自然id就可以重复，namespace不同，namespace+id自然也就不同。

### 3.2.12 Mybatis中如何执行批处理？

使用BatchExecutor完成批处理。

### 3.2.13 Mybatis都有哪些Executor执行器？它们之间的区别是什么？

Mybatis有三种基本的Executor执行器，**SimpleExecutor、ReuseExecutor、BatchExecutor。**

**SimpleExecutor：**每执行一次update或select，就开启一个Statement对象，用完立刻关闭Statement对象。

**ReuseExecutor：**执行update或select，以sql作为key查找Statement对象，存在就使用，不存在就创建，用完后，不关闭Statement对象，而是放置于Map<String, Statement>内，供下一次使用。简言之，就是重复使用Statement对象。

**BatchExecutor：**执行update（没有select，JDBC批处理不支持select），将所有sql都添加到批处理中（addBatch()），等待统一执行（executeBatch()），它缓存了多个Statement对象，每个Statement对象都是addBatch()完毕后，等待逐一执行executeBatch()批处理。与JDBC批处理相同。

作用范围：Executor的这些特点，都严格限制在SqlSession生命周期范围内。

### 3.2.14 Mybatis中如何指定使用哪一种Executor执行器？

在Mybatis配置文件中，可以指定默认的ExecutorType执行器类型，也可以手动给DefaultSqlSessionFactory的创建SqlSession的方法传递ExecutorType类型参数。

### 3.2.15 Mybatis是否可以映射Enum枚举类？

Mybatis可以映射枚举类，不单可以映射枚举类，Mybatis可以映射任何对象到表的一列上。映射方式为自定义一个TypeHandler，实现TypeHandler的setParameter()和getResult()接口方法。TypeHandler有两个作用，一是完成从javaType至jdbcType的转换，二是完成jdbcType至javaType的转换，体现为setParameter()和getResult()两个方法，分别代表设置sql问号占位符参数和获取列查询结果。

### 3.2.16 Mybatis映射文件中，如果A标签通过include引用了B标签的内容，请问，B标签能否定义在A标签的后面，还是说必须定义在A标签的前面？

虽然Mybatis解析Xml映射文件是按照顺序解析的，但是，被引用的B标签依然可以定义在任何地方，Mybatis都可以正确识别。

原理是，Mybatis解析A标签，发现A标签引用了B标签，但是B标签尚未解析到，尚不存在，此时，Mybatis会将A标签标记为未解析状态，然后继续解析余下的标签，包含B标签，待所有标签解析完毕，Mybatis会重新解析那些被标记为未解析的标签，此时再解析A标签时，B标签已经存在，A标签也就可以正常解析完成了。

### 3.2.17 简述Mybatis的Xml映射文件和Mybatis内部数据结构之间的映射关系？

Mybatis将所有Xml配置信息都封装到All-In-One重量级对象Configuration内部。在Xml映射文件中，<parameterMap>标签会被解析为ParameterMap对象，其每个子元素会被解析为ParameterMapping对象。<resultMap>标签会被解析为ResultMap对象，其每个子元素会被解析为ResultMapping对象。每一个<select>、<insert>、<update>、<delete>标签均会被解析为MappedStatement对象，标签内的sql会被解析为BoundSql对象。

### 3.2.18 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

面试题看似都很简单，但是想要能正确回答上来，必定是研究过源码且深入的人，而不是仅会使用的人或者用的很熟的人，以上所有面试题及其答案所涉及的内容，在我的Mybatis系列博客中都有详细讲解和原理分析。

## ****3.3、Netty****

3.3.为什么选择 Netty

### 3.3.1 说说业务中，Netty 的使用场景

### 3.3.2 原生的 NIO 在 JDK 1.7 版本存在 epoll bug

### 3.3.3 什么是TCP 粘包/拆包

### 3.3.4 TCP粘包/拆包的解决办法

### 3.3.5 Netty 线程模型

### 3.3.6 说说 Netty 的零拷贝

### 3.3.7 Netty 内部执行流程

### 3.3.8 Netty 重连实现

## ****3.4、Tomcat****

### 3.4.1 Tomcat的基础架构（Server、Service、Connector、Container）

### 3.4.2 Tomcat如何加载Servlet的

### 3.4.3 Pipeline-Valve机制

可参考：《[四张图带你了解Tomcat系统架构](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484905&idx=1&sn=6c8acd89476fadbc4cb9ccfda9c9c2e4&chksm=e9c5fc58deb2754e7519511bb0ed8dcbfa3fe29179663b53f3626643f8b9c82068d9b0464ee6&scene=21#wechat_redirect)！》

## ****3.5、Dubbo****

### 3.5.1 什么是Dubbo

可参考《[Dubbo入门](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483791&idx=1&sn=49345f1a022734e81e9257f2b8d38a52&chksm=e9c5f83edeb2712805a77c1e1589e8f1d04bd17e55eeb2a45cabddb46d03615636908f058628&scene=21#wechat_redirect)》

### 3.5.2 什么是RPC、如何实现RPC、RPC 的实现原理

可参考《[基于HTTP的RPC实现](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483900&idx=1&sn=c5ca198a66a701f81c2ab118fe7a734a&chksm=e9c5f84ddeb2715bc574e467cd6537ef81f223453e0989ffd136976b48dcc2d961a75be596de&scene=21#wechat_redirect)》

### 3.5.3 Dubbo中的SPI是什么概念

### 3.5.4 Dubbo的基本原理、执行流程

## 3.6、Zookeeper

### 3.6.1 zookeeper是什么框架？

### 3.6.2 有哪些应用场景？

### 3.6.3 使用什么协议？

### 3.6.4 说说分布式一致性算法Paxos

### 3.6.5 说一说选举算法及流程

### 3.6.6 zookeeper有哪几种节点类型？

### 3.6.7 zookeeper对节点的watch监听通知是永久的吗？

### 3.6.8 有哪几种部署模式？

### 3.6.9 集群中的机器角色都有哪些？

### 3.6.10 集群最少要几台机器，集群规则是怎样的

### 3.6.11 集群如果有3台机器，挂掉一台集群还能工作吗？挂掉两台呢？

### 3.6.12 集群支持动态添加机器吗？

### 3.6.13 zookeeper的java客户端都有哪些？

### 3.6.14 chubby是什么，和zookeeper比你怎么看？

### 3.6.15 说几个zookeeper常用的命令。

## 3.7、SpringCloud

### 3.7.1 Eureka

### 3.7.2 Ribbon

### 3.7.3 Feign

### 3.7.4 Hystrix

### 3.7.5 zuul

### 3.7.6 SpringCloud Config

# ****四、分布式****

## ****4.1、Nginx****

### 4.1.1 请解释什么是C10K问题或者知道什么是C10K问题吗？

### 4.1.2 Nginx简介，可参考《[Nginx简介](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483994&idx=1&sn=b6591f62c7ea6b4adc5a5bf1bf4eac40&chksm=e9c5fbebdeb272fdd865a9c61a380f6b909fc988f99d00ce0aa8c3efca501644db46c40bd4f2&scene=21#wechat_redirect)》

### 4.1.3 正向代理和反向代理.

### 4.1.4 Nginx几种常见的负载均衡策略

### 4.1.5 Nginx服务器上的Master和Worker进程分别是什么

### 4.1.6 使用“反向代理服务器”的优点是什么?

## ****4.2、分布式其他****

### 4.2.1 谈谈业务中使用分布式的场景

### 4.2.2 Session 分布式方案

<https://blog.csdn.net/sxiaobei/article/details/5708648>

### 4.2.3 Session 分布式处理

### 4.2.4 分布式锁的应用场景、分布式锁的产生原因、基本概念

### 4.2.5 分布是锁的常见解决方案

### 4.2.6 分布式事务的常见解决方案

### 4.2.7 集群与负载均衡的算法与实现

### 4.2.8 说说分库与分表设计，可参考《[数据库分库分表策略的具体实现方案](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483931&idx=1&sn=6eda41aa81c1243422a603205d2fad22&chksm=e9c5fbaadeb272bc92537803c14a6f55e1170b1a3b8f60160f66417800c0ace960dfe192717a&scene=21#wechat_redirect)》

### 4.2.9 分库与分表带来的分布式困境与应对之策

# **五、微服务**

## ****5.1、微服务****

### 5.1.1 什么是微服务

微服务化的核心就是将传统的一站式应用，根据业务拆分成一个一个的服务，彻底地去耦合,每一个微服务提供单个业务功能的服务，一个服务做一件事， 从技术角度看就是一种小而独立的处理过程，类似进程概念，能够自行单独启动或销毁，拥有自己独立的数据库。

### 5.1.2 微服务与微服务架构

微服务架构是⼀种架构模式，它提倡将单⼀应⽤程序划分成⼀组⼩的服务，服务之间互相协调、互相配合，为⽤户提供最终价值。每个服务运⾏在其独⽴的进程中，服务与服务间采⽤轻量级的通信机制互相协作（通常是基于HTTP协议的RESTful API）。每个服务都围绕着具体业务进⾏构建，并且能够被独⽴的部署到⽣产环境、类⽣产环境等。另外，应当尽量避免统⼀的、集中式的服务管理机制，对具体的⼀个服务⽽⾔，应根据业务上下⽂，选择合适的语⾔、⼯具对其进⾏构建。

### 5.1.3 微服务优缺点

**优点：**

* 每个服务足够内聚，足够小，代码容易理解这样能聚焦一个指定的业务功能或业务需求 开发简单、开发效率提高，一个服务可能就是专一的只干一件事。 微服务能够被小团队单独开发，这个小团队是2到5人的开发人员组成。
* 微服务是松耦合的，是有功能意义的服务，无论是在开发阶段或部署阶段都是独立的。 微服务能使用不同的语言开发。
* 易于和第三方集成，微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如Jenkins, Hudson, bamboo 。
* 微服务易于被一个开发人员理解，修改和维护，这样小团队能够更关注自己的工作成果。无需通过合作才能体现价值。 微服务允许你利用融合最新技术。 微服务只是业务逻辑的代码，不会和HTML,CSS 或其他界面组件混合。
* 每个微服务都有自己的存储能力，可以有自己的数据库。也可以有统一数据库。

**缺点**

* 开发人员要处理分布式系统的复杂性
* 多服务运维难度，随着服务的增加，运维的压力也在增大
* 系统部署依赖 服务间通信成本
* 数据一致性
* 系统集成测试
* 性能监控……

### 5.1.4 微服务技术栈

|  |  |
| --- | --- |
| **微服务条目** | **落地技术** |
| 服务开发 | Springboot、Spring、SpringMVC |
| 服务配置与管理 | Netflix公司的Archaius、阿里的Diamond等 |
| 服务注册与发现 | Eureka、Consul、Zookeeper等 |
| 服务调用 | Rest、RPC、gRPC |
| 服务熔断器 | Hystrix、Envoy等 |
| 负载均衡 | Ribbon、Nginx等 |
| 服务接口调用(客户端调用服务的简化工具) | Feign等 |
| 消息队列 | Kafka、RabbitMQ、ActiveMQ等 |
| 服务配置中心管理 | SpringCloudConfig、Chef等 |
| 服务路由 | Zuul等 |
| 服务监控 | Zabbix、Nagios、Metrics、Spectator等 |
| 全链路追踪 | Zipkin，Brave、Dapper等 |
| 服务部署 | Docker、OpenStack、Kubernetes等 |
| 数据流操作开发包 | SpringCloud Stream（封装与Redis,Rabbit、Kafka等发送接收消息） |
| 事件消息总线 | Spring Cloud Bus |
| 。。。。。。。。 |  |

### 5.1.5 springcloud 是什么

SpringCloud，基于SpringBoot提供了一套微服务解决方案，包括服务注册与发现，配置中心，全链路监控，服务网关，负载均衡，熔断器等组件，除了基于NetFlix的开源组件做高度抽象封装之外，还有一些选型中立的开源组件。

SpringCloud利用SpringBoot的开发便利性巧妙地简化了分布式系统基础设施的开发，SpringCloud为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等,它们都可以用SpringBoot的开发风格做到一键启动和部署。

SpringBoot并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过SpringBoot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

SpringCloud=分布式微服务架构下的一站式解决方案，是各个微服务架构落地技术的集合体，俗称微服务全家桶。

### 5.1.6 SpringCloud和SpringBoot是什么关系

SpringBoot专注于快速方便的开发单个个体微服务。

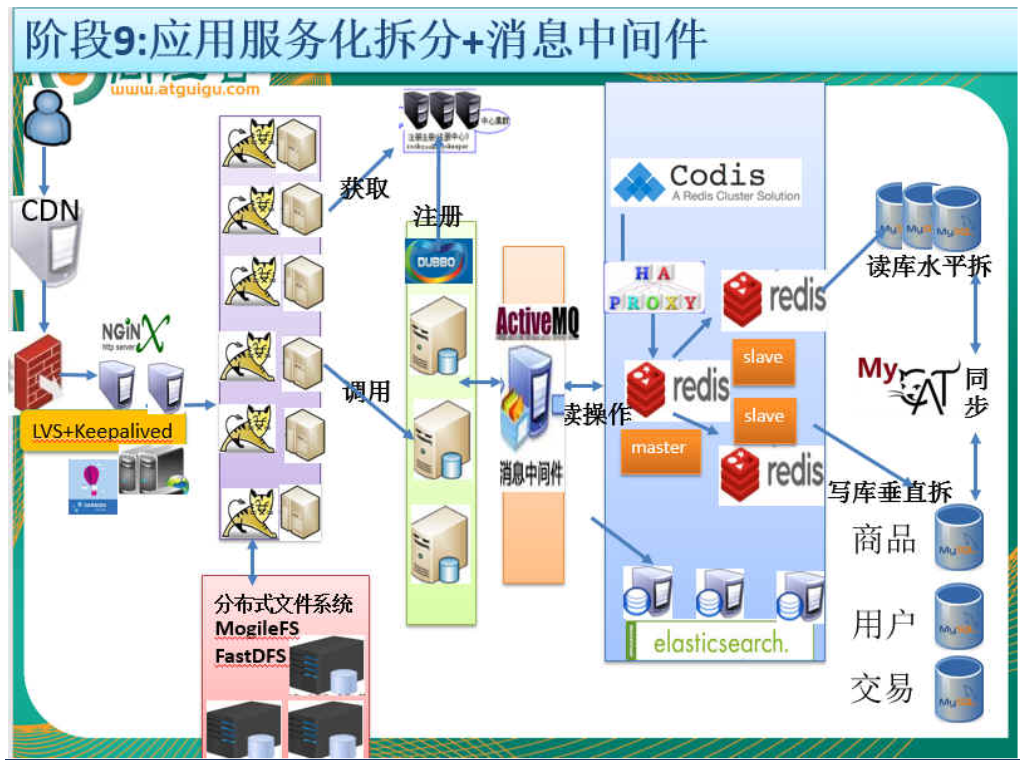
SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，

为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服务

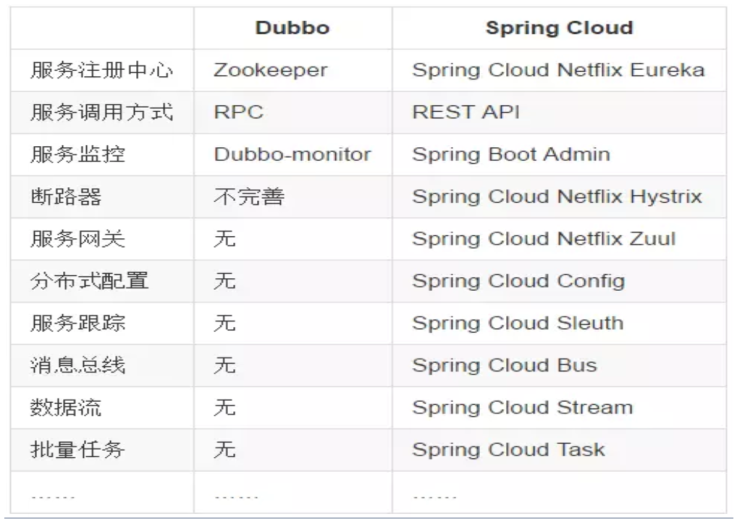
SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于依赖的关系.

SpringBoot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

### 5.1.7 目前成熟的互联网架构（分布式+服务治理Dubbo）



### 5.1.8 SpringCloud对比DUBBO



最大区别：**SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式**。严格来说，这两种方式各有优劣。虽然从一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更加合适。

品牌机与组装机的区别，很明显，Spring Cloud的功能比DUBBO更加强大，涵盖面更广，而且作为Spring的拳头项目，它也能够与Spring Framework、Spring Boot、Spring Data、Spring Batch等其他Spring项目完美融合，这些对于微服务而言是至关重要的。使用Dubbo构建的微服务架构就像组装电脑，各环节我们的选择自由度很高，但是最终结果很有可能因为一条内存质量不行就点不亮了，总是让人不怎么放心，但是如果你是一名高手，那这些都不是问题；而Spring Cloud就像品牌机，在Spring Source的整合下，做了大量的兼容性测试，保证了机器拥有更高的稳定性，但是如果要在使用非原装组件外的东西，就需要对其基础有足够的了解。

社区支持与更新力度，最为重要的是，DUBBO停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX），这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。

================

前后端分离是如何做的？

微服务哪些框架

Spring Could的常见组件有哪些？可参考《[Spring Cloud概述](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484125&idx=1&sn=ddba9fba6ae900f5ef71a68f70afebe5&chksm=e9c5fb6cdeb2727a1e8d16a05adb7d9df0170e392ae579d1e40075fb488d680f0c061aa45327&scene=21#wechat_redirect)》

领域驱动有了解吗？什么是领域驱动模型？充血模型、贫血模型

5.1.5 JWT有了解吗，什么是JWT，可参考《[前后端分离利器之JWT](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247485183&idx=1&sn=05dac824dbb534710dd99d6c895fbaf5&chksm=e9c5ff4edeb27658173c8b06ad6d1241d3b7822c734ddf6ac064d40e63cb0cb0a0c90804b9c7&scene=21#wechat_redirect)》

5.1.6 你怎么理解 RESTful

5.1.7 说说如何设计一个良好的 API

5.1.8 如何理解 RESTful API 的幂等性

5.1.9 如何保证接口的幂等性

5.1.10 说说 CAP 定理、BASE 理论

5.1.11 怎么考虑数据一致性问题

5.1.12 说说最终一致性的实现方案

5.1.13 微服务的优缺点，可参考《[微服务批判](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247485005&idx=1&sn=78a1d286c6a15a81ea5dcf6634a70b54&chksm=e9c5fffcdeb276ea3c766a6e5954685db0e89bea8ff0f47c9a3ff2c0c02991f791a5160287c4&scene=21#wechat_redirect)》

5.1.14 微服务与 SOA 的区别

5.1.15 如何拆分服务、水平分割、垂直分割

5.1.16 如何应对微服务的链式调用异常

5.1.17 如何快速追踪与定位问题

5.1.18 如何保证微服务的安全、认证

## ****5.2、安全问题****

### 5.2.1 如何防范常见的Web攻击、如何方式SQL注入

### 5.2.2 服务端通信安全攻防

### 5.2.3 HTTPS原理剖析、降级攻击、HTTP与HTTPS的对比

## ****5.3、性能优化****

### 5.3.1 性能指标有哪些

### 5.3.2 如何发现性能瓶颈

### 5.3.3 性能调优的常见手段

### 5.3.4 说说你在项目中如何进行性能调优

# ****六、其他****

## ****6.1、设计能力****

### 6.1.1 说说你在项目中使用过的UML图

### 6.1.2 你如何考虑组件化、服务化、系统拆分

### 6.1.3 秒杀场景如何设计

可参考：《[秒杀系统的技术挑战、应对策略以及架构设计总结一二！](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247485294&idx=1&sn=681b3fc8833bc978344f66c8dd33ff32&chksm=e9c5fedfdeb277c96e03e8943bff709ad5354cbbe0c0d894cdb9d1e8cc8da92bf51f1e832e37&scene=21#wechat_redirect)》

## ****6.2、业务工程****

### 6.2.1 说说你的开发流程、如何进行自动化部署的

### 6.2.2 你和团队是如何沟通的

### 6.2.3 你如何进行代码评审

### 6.2.4 说说你对技术与业务的理解

### 6.2.5 说说你在项目中遇到感觉最难Bug，是如何解决的

### 6.2.6 介绍一下工作中的一个你认为最有价值的项目，以及在这个过程中的角色、解决的问题、你觉得你们项目还有哪些不足的地方

## ****6.3、软实力****

### 6.3.1 说说你的优缺点、亮点

### 6.3.2 说说你最近在看什么书、什么博客、在研究什么新技术、再看那些开源项目的源代码

### 6.3.3 说说你觉得最有意义的技术书籍

### 6.3.4 工作之余做什么事情、平时是如何学习的，怎样提升自己的能力

### 6.3.5 说说个人发展方向方面的思考

### 6.3.6 说说你认为的服务端开发工程师应该具备哪些能力

### 6.3.7 说说你认为的架构师是什么样的，架构师主要做什么

### 6.3.8 如何看待加班的问题