**Java创建对象有哪几种方式**

**1.使用new关键字**

使用new关键字，需要构造器支持；

Date date = new Date();

**2.使用反射方式**

1.利用Class或Constructor对象的newInstance()方法来创建类的实例，会调用对应的函数；

**2.1获取Class实例的三种方式**

I.通过类名.class；

Class<?> clazz =Date.class;

II.通过Class.forName；

Class<?> clazz = Class.forName("com.stu.obj.People");

III.通过已知对象实例.getClass()；

Date date = new Date();

Class<? extends Date> clazz = date.getClass();

**2.2反射方式创建对象**

使用Class对象的newInstance()

Object newInstance = clazz.newInstance();

使用Constructor对象的newInstance()

Constructor<?> constructor = clazz.getConstructor();

Object newInstance4 = constructor.newInstance();

**3.使用Object类的clone();**

1.类必须实现Cloneable接口且覆盖clone()方法，调用一个对象的clone方法，JVM就会创建一个新的对象，将前面对象的内容全部拷贝进去。用clone方法创建对象并不会调用任何构造函数。

**4.使用反序列化方式**

利用ObjectInputStream对象中的readObject()方法反序列化类对象，反序列化时，JVM会创建一个单独的对象。为了反序列化一个对象，我们需要让类实现Serializable接口。在反序列化时，JVM创建对象并不会调用任何构造函数

**5.面试**

**I.Java中创建对象有哪几种方式**  
1）使用new关键字；  
2）使用反射,利用Class或Constructor对象的newInstance()方法实例化对象；  
3）使用Object对象中的clone方法；  
4）使用反序列化  
**II.相关知识储备**  
1.掌握反射相关操作及Spring在哪些地方使用了反射机制；  
2.掌握浅克隆和深克隆实现和区别  
3.掌握序列化和反序列化相关操作；

**Java如何实现序列化和反序列化**

[Java基础](http://localhost:9093/itour/work/detail?id=107)

阅读  最后发布于 2022-07-24 14:25

**1.什么是序列化和反序列化？**

1.序列化是把对象转换为字节序列的过程；

2.反序列化是把字节序列恢复为Java对象的过程；

**2.序列化的作用？**

1）在传递和保存对象的时候，保证对象的完整性和可传递性。

2）将对象转换成有序字节流，以便在网络上传输或者保存在本地文件中。

**3.反序列化的作用？**

客户端从文件中或网络上得到序列化后的对象字节流，根据字节流中所保存的对象状态及描述信息，经过反序列化重建对象；

**4.什么场景下需要使用序列化？**

需要在网络,如socket中传输Java对象;由于数据只可以以二进制的形式在网络中进行传输，所以当把对象经过网络发送出去以前须要先序列化成二进制数据，在接收 端读到二进制数据以后反序列化成Java对象；

**5.如何实现序列化？**

**5.1具体步骤**

1)序列化类必须实现Serializable接口；

2)利用ObjectOutputStream中的writeObject()方法，将对象写入指定位置；

**5.2具体实现**

FileOutputStream fos = null;

ObjectOutputStream oos = null;

try {

Emp emp = new Emp();

emp.setEmpno(10000L);

emp.setEname("MIKE");

fos = new FileOutputStream(new File("D:\\temp\\emp.txt"));

oos = new ObjectOutputStream(fos);

oos.writeObject(emp);

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}finally {

try {

oos.close();

fos.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

**6.如何实现反序列化？**

**6.1具体步骤**

1)反序列化类必须实现Serializable接口；

2)利用ObjectInputStream对象中的readObject()方法反序列化类对象；

**6.2具体步骤**

FileInputStream fis = null;

ObjectInputStream ois = null;

try {

fis = new FileInputStream(new File("D:\\temp\\emp.txt"));

ois = new ObjectInputStream(fis);

Emp emp = (Emp)ois.readObject();

System.out.println(emp);

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

} catch (ClassNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}finally {

try {

ois.close();

fis.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

**7.序列化中注意事项**

1）Serializable是个空接口，并没有包含任何方法,只是一个做标记用，告诉程序只要是实现了Serializable接口的类都是可以被序列化的；

2)serialVersionUID是序列化前后的唯一标识符,默认如果没有人为显式定义过serialVersionUID，那编译器会为它自动声明一个。在反序列化时，JVM会把字节流中的序列号ID和被序列化类中的序列号ID做比对，只有两者一致，才能重新反序列化，否则就会报异常来终止反序列化的过程。建议凡是implements Serializable的类，都最好人为显式地为它声明一个serialVersionUID明确值！

3)实现了Serializable接口，被static或transient修饰符修饰的字段不会被序列化；

**面试**

1）了解什么是序列化和反序列化？  
2）什么场景下需要使用序列化？  
3）如何实现序列化及序列化中的注意事项？

**浅拷贝和深拷贝的区别？**

**1.什么是浅拷贝？**

Java中数据类型分为基本数据类型和引用数据类型，对于浅拷贝来说，如果对象中的属性的数据类型为基本数据类型，则拷贝值；如果是引用数据类型，则会拷贝内存地址，如果任何一个对象改变都会导致其他对象发生对应改变；

由于String和包装类本身的final性(不可变性),每次赋值都是一个新的引用地址，原对象的引用和副本的引用互不影响。 因此String就和基本数据类型一样,表现出了"深拷贝"特性.

**2.什么是深拷贝？**

对于深拷贝，无论对象中的数据是基本数据类型还是引用数据类型对象都是独立的，相当于重新创建了一个对象；

**3.实现浅拷贝的方式**

**1.实现Cloneable接口，覆盖Object中的clone()方法**；

public class Emp implements Cloneable{

private Long empno;

private String ename;

private Dept dept;

public Long getEmpno() {

return empno;

}

public void setEmpno(Long empno) {

this.empno = empno;

}

public String getEname() {

return ename;

}

public void setEname(String ename) {

this.ename = ename;

}

public Dept getDept() {

return dept;

}

public void setDept(Dept dept) {

this.dept = dept;

}

@Override

protected Object clone() throws CloneNotSupportedException {

// TODO Auto-generated method stub

return super.clone();

}

@Override

public String toString() {

return "Emp [empno=" + empno + ", ename=" + ename + ", dept=" + dept + "]";

}

}

public class TestClone {

@Test

public void testShallowCopy() throws CloneNotSupportedException {

//1.构建数据

Emp emp = new Emp();

emp.setEmpno(1000L);

emp.setEname("Jack");

Dept dept = new Dept();

dept.setId(1L);

dept.setDname("技术部");

emp.setDept(dept);

//2.拷贝

Emp employee = (Emp)emp.clone();

/\*\*3. 源对象和拷贝对象的Hashcode不同,内存地址不同,

\* 可以知道,拷贝对象和源对象是不同的对象;

\*/

System.out.println("Emp和employee对象的HashCode: "+

(emp.hashCode() == employee.hashCode())+

" Emp和employee对象的内存地址："+(emp == employee) );//false false

/\*\*4.源对象和拷贝对象中Dept对象的Hashcode相同且内存地址相同,

\* 可以知道，源对象和拷贝对象持有相同的Dept对象的引用;

\*

\*/

System.out.println("Emp和employee中Dept对象的HashCode："

+(emp.getDept().hashCode() == employee.getDept().hashCode())+

" Emp和employee中Dept对象的内对象地址"+

(emp.getDept()==employee.getDept())); //true true

//5.修改对象操作

emp.setEmpno(2008L);

emp.setEname("google");

Dept d = emp.getDept();

d.setDname("销售部");

d.setId(2L);

/\*\*6.源对象和拷贝对象中的Dept都发生了改变,但是克隆的对象中的empno,ename没有改变，

\* 这是由于String和包装类本身的final性(不可变性),每次赋值都是一个新的引用地址，

\* 原对象的引用和副本的引用互不影响。 因此String就和基本数据类型一样,表现出了

\* "深拷贝"特性.\*\*/

System.out.println("原始对象"+emp);

System.out.println("拷贝对象"+employee);

}

}

**2.使用BeanUtils中copyProperties方法**

Emp p = new Emp(1L,"amy");

p.setDept(new Dept(1L,"SALE"));

Emp p1 = new Emp();

BeanUtils.copyProperties(p, p1);

System.out.println("p1 "+p1);

**4.深拷贝实现**

**1.实现Cloneable接口，覆盖Object中的clone()方法；**

public class Emp implements Cloneable{

private Long empno;

private String ename;

private Dept dept;

public Long getEmpno() {

return empno;

}

public void setEmpno(Long empno) {

this.empno = empno;

}

public String getEname() {

return ename;

}

public void setEname(String ename) {

this.ename = ename;

}

public Dept getDept() {

return dept;

}

public void setDept(Dept dept) {

this.dept = dept;

}

@Override

protected Object clone() throws CloneNotSupportedException {

// TODO Auto-generated method stub

Emp emp = (Emp)super.clone();

emp.dept = (Dept)dept.clone();

return emp;

}

@Override

public String toString() {

return "Emp [empno=" + empno + ", ename=" + ename + ", dept=" + dept + "]";

}

}

public class Dept implements Cloneable {

private Long id;

private String dname;

public Long getId() {

return id;

}

public void setId(Long id) {

this.id = id;

}

public String getDname() {

return dname;

}

public void setDname(String dname) {

this.dname = dname;

}

@Override

protected Object clone() throws CloneNotSupportedException {

// TODO Auto-generated method stub

return super.clone();

}

@Override

public String toString() {

return "Dept [id=" + id + ", dname=" + dname + "]";

}

}

public class TestObj {

@Test

public void testClone() throws CloneNotSupportedException {

Emp p = new Emp(1L,"amy");

p.setDept(new Dept(2L,"SALEMAN"));

Emp emp1 = (Emp)p.clone();

/\*\*源对象和拷贝对象中Dept对象的Hashcode不相同且内存地址不相同,

\* 可以知道，源对象和拷贝对象是两个对象;

\*

\*/

System.out.println("Emp和employee中Dept对象的HashCode："

+(emp.getDept().hashCode() == employee.getDept().hashCode())+

" Emp和employee中Dept对象的内对象地址"+

(emp.getDept()==employee.getDept())); //false false

}

**2.对象序列化后，然后反序列化生成对象**

Emp p = new Emp(1L,"amy");

//将对象写入流中

ByteArrayOutputStream bos = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(bos);

oos.writeObject(p);

//将流取出转换为对象

ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());

ObjectInputStream ois = new ObjectInputStream(bis);

Emp cEmp = (Emp)ois.readObject();

System.out.println("cemp "+cEmp);

**3.JSON方式**

Emp p = new Emp(1L,"amy");

String jsonString = JSON.toJSONString(p);

Emp parseObject = JSON.parseObject(jsonString, Emp.class);

**面试**

1）浅拷贝和深拷贝的区别  
2）浅拷贝和深拷贝的实现方式