你真的了解单例模式吗?

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：单例模式

题目描述

手把手教你手写写单例模式

题目解决

概念

系统中只需要一个全局的实例，比如一些工具类，Converter，SqlSession等。

为什么要用单例模式?

只有一个全局的实例，减少了内存开支，特别是某个对象需要频繁的创建和销毁的时候，而创建和销毁的过程由jvm执行，我们无法对其进行优化，所以单例模式的优势就显现出来啦。
单例模式可以避免对资源的多重占用，避免出现多线程的复杂问题。

单例模式的写法重点

构造方法私有化
我们需要将构造方法私有化，而默认不写的话，是公有的构造方法，外部可以显式的调用来创建对象，我们的目的是让外部不能创建对象。

提供获取实例的公有方法
对外只提供一个公有的的方法，用来获取实例，而这个实例是否是唯一的，单例的，由方法决定，外部无需关心。

单例模式的常见写法（如下，重点）

饿汉式和懒汉式的区别

饿汉式

饿汉式，从名字上也很好理解，就是“比较饿”，迫不及待的想吃饭，实例在初始化的时候就已经建好了，不管你有没有用到，都先建好了再说。

懒汉式

饿汉式，从名字上也很好理解，就是“比较懒”，不想吃饭，等饿的时候再吃。在初始化的时候先不建好对象，如果之后用到了，再创建对象。

1.饿汉式（静态变量）-可以使用

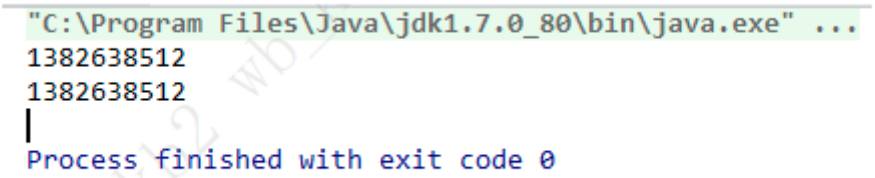
A类

```
public class A {
    //私有的构造方法
    private A(){}
    //私有的静态变量
    private final static A a=new A();
    //对外的公有方法
    public static A getInstance(){
        return a;
    }
}
```

测试类

```
public class test {
    public static void main(String[] args){
        A a1=A.getInstance();
        System.out.println(a1.hashCode());

        A a2=A.getInstance();
        System.out.println(a2.hashCode());
    }
}
```

运行结果



说明
该方法采用的静态常量的方法来生成对应的实例，其只在类加载的时候就生成了，后续并不会再生成，所以其为单例的。

优点
在类加载的时候，就完成实例化，避免线程同步问题。

缺点
没有达到懒加载的效果，如果从始到终都没有用到这个实例，可能会导致内存的浪费。

2.饿汉式（静态代码块）-可以使用

A类

```
public class A {
    //私有的构造方法
     private A(){}
    //私有的静态变量
     private final static A a;
    //静态代码块
    static{ a=new A(); }
    //对外的公有方法
    public static A getInstance(){
     return a;
    }
}
```

测试类

```
public class test {
    public static void main(String[] args){
        A a1=A.getInstance();
        System.out.println(a1.hashCode());

        A a2=A.getInstance();
        System.out.println(a2.hashCode());
    }
}
```

运行结果

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...
1438896971
1438896971

Process finished with exit code 0
```

说明
该静态代码块的饿汉式单例模式与静态变量的饿汉式模式大同小异，只是将初始化过程移到了静态代码块中。

优点缺点
与静态变量饿汉式的优缺点类似。

3.懒汉式

A类

```
public class A {
    //私有的构造方法
    private A(){}
    //私有的静态变量
    private  static A a;
    //对外的公有方法
    public static A getInstance(){
        if(a==null){
            a=new A();
        }
        return a;
    }
}
```

测试类和运行结果
同上。

优点
该方法的确做到了用到即加载，也就是当调用getInstance的时候，才判断是否有该对象，如果不为空，则直接放回，如果为空，则新建一个对象并返回，达到了懒加载的效果。

缺点

当多线程的时候，可能会产生多个实例。比如我有两个线程，同时调用getInstance方法，并都到了if语句，他们都新建了对象，那这里就不是单例的啦。

4.懒汉式（线程安全，同步方法）-可以使用

A类

```
public class A {
    //私有的构造方法
    private A(){}
    //私有的静态变量
    private  static A a;
    //对外的公有方法
    public synchronized static A getInstance(){
        if(a==null){
            a=new A();
        }
        return a;
    }
}
```

测试类和运行结果
同上。

优点
通过synchronize关键字，解决了线程不安全的问题。如果两个线程同时调用getInstance方法时，那就先执行一个线程，另一个等待，等第一个线程运行结束了，另一个等待的开始执行。

缺点
这种方法是解决了线程不安全的问题，却给性能带来了很大的问题，效率太低了，getInstance经常发生，每一次都要同步这个方法。

我们想着既然是方法同步导致了性能的问题，我们核心的代码就是新建对象的过程，也就是new A（）；的过程，我们能不能只对部分代码进行同步呢?

那就是方法5啦。

5.懒汉式（线程不安全）

A类

```
public class A {
    //私有的构造方法
    private A(){}
    //私有的静态变量
    private  static A a;
    public  static A getInstance(){
        if(a==null){
            synchronized (A.class){
                a=new A();
            }
        }
        return a;
    }
}
```

测试类和运行结果

如上。

优点

懒汉式的通用优点，用到才创建，达到懒加载的效果。

缺点

这个没有意义，并没有解决多线程的问题。我们可以看到如果两个线程同时调用getInstance方法，并且都已经进入了if语句，即synchronized的位置，即便同步了，第一个线程先执行，进入synchronized同步的代码块，创建了对象，另一个进入等待状态，等第一个线程执行结束，第二个线程还是会进入synchronized同步的代码块，创建对象。这个时候我们可以发现，对这代码块加了synchronized没有任何意义，还是创建了多个对象，并不符合单例。

6.双重检查 --强烈推荐使用

A类

```
public class A {
    //私有的构造方法
    private A() {
    }

    //私有的静态变量
    private volatile static A a;

    //对外的公有方法
    public static A getInstance() {
        if (a == null) {
            synchronized (A.class) {
                if (a == null) {
                    a = new A();
```

```
                }
            }
        }
        return a;
    }
}
```

测试类和运行结果

同上。

优点

强烈推荐使用，这种写法既避免了在多线程中出现线程不安全的情况，也能提高性能。

咱具体来说，如果两个线程同时调用了getInstance方法，并且都已到达了if语句之后，synchronized语句之前，此时第一个线程进入synchronized之中，先判断是否为空，很显然第一次肯定为空，那么则新建了对象。等到第二个线程进入synchronized之中，先判断是否为空，显然第一个已经创建了，所以即不新建对象。下次，不管是一个线程或者多个线程，在第一个if语句那就判断出有对象了，便直接返回啦，根本进不了里面的代码。

缺点

就是这么完美，没有缺点，哈哈哈。

volatile（插曲）

咱先来看一个概念,`重排序`，也就是语句的执行顺序会被重新安排。其主要分为三种：

1.编译器优化的重排序：可以重新安排语句的执行顺序。

2.指令级并行的重排序：现代处理器采用指令级并行技术，将多条指令重叠执行。

3.内存系统的重排序：由于处理器使用缓存和读写缓冲区，所以看上去可能是乱序的。

上面代码中的a = new A();可能被被JVM分解成如下代码：

```
// 可以分解为以下三个步骤
1 memory=allocate();// 分配内存 相当于c的malloc
2 ctorInstanc(memory) //初始化对象
3 s=memory //设置s指向刚分配的地址

// 上述三个步骤可能会被重排序为 1-3-2，也就是：
1 memory=allocate();// 分配内存 相当于c的malloc
3 s=memory //设置s指向刚分配的地址
2 ctorInstanc(memory) //初始化对象
```

一旦假设发生了这样的重排序，比如线程A在执行了步骤1和步骤3，但是步骤2还没有执行完。这个时候线程B有进入了第一个if语句，它会判断a不为空，即直接返回了a。其实这是一个未初始化完成的a，即会出现问题。

所以我们会将入volatile关键字，来禁止这样的重排序，即可正常运行。

7.静态内部类 --强烈推荐使用

A类

```
public class A {
    //私有构造函数
    private A() {
    }

    //私有的静态内部类
    private static class B {
        //私有的静态变量
        private static A a = new A();
    }

    //对外的公有方法
    public static A getInstance() {
        return B.a;
    }
}
```

优点

B在A装载的时候并不会装载，而是会在调用getInstance的时候装载，这利用了JVM的装载机制。这样一来，优点有两点，其一就是没有A加载的时候，就装载了a对象，而是在调用的时候才装载，避免了资源的浪费。其二是多线程状态下，没有线程安全性的问题。

缺点

没有缺点，太完美啦。

8.枚举 --Java粑粑强烈推荐使用

问题1：私有构造器并不安全

如果我们的对象是通过反射方法invoke出来，这样新建的对象与通过调用getInstance新建的对象是不一样的，具体咱来看代码。

```
public class test {
    public static void main(String[] args) throws Exception {
        A a=A.getInstance();
        A b=A.getInstance();
        System.out.println("a的hash：+a.hashCode()+",b的hash：+b.hashCode());

        Constructor<A> constructor=A.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        A c=constructor.newInstance();
        System.out.println("a的hash：+a.hashCode()+",c的hash：+c.hashCode());

    }
}
```

我们来看下运行结果：

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...
a的hash: 811560660,b的hash: 811560660
a的hash: 811560660,c的hash: 1231370523

Process finished with exit code 0
```

我们可以看到c的hashcode是和a,b不一样，因为c是通过构造器反射出来的，由此可以证明私有构造器所组成的单例模式并不是十分安全的。

问题2：序列化问题

我们先将A类实现一个Serializable接口，具体代码如下，跟之前的双重if检查一样，只是多了个接口。

```
public class A implements Serializable {
    //私有的构造方法
    private A() {
    }

    //私有的静态变量
    private volatile static A a;

    //对外的公有方法
    public static A getInstance() {
        if (a == null) {
            synchronized (A.class) {
                if (a == null) {
                    a = new A();
                }
            }
        }
        return a;
    }
}
```

测试类:

```
public class test {
    public static void main(String[] args) throws Exception {
        A s = A.getInstance();

        //写
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("学习Java的小姐姐"));
        oos.writeObject(s);
        oos.flush();
        oos.close();
        //读
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("学习Java的小姐姐"));
        A s1 = (A)ois.readObject();
        ois.close();

        System.out.println(s+"\n"+s1);
```

```
        System.out.println("序列化前后两个是否同一个："+(s==s1));
    }
}
```

我们来看下运行结果，很显然序列化前后两个对象并不相等。为什么会出现这种问题呢？这个讲起来，又可以写一篇文章了。简单来说，任何一个readObject方法，不管是显式的还是默认的，它都会返回一个新建的实例，这个新建的实例不同于该类初始化时创建的实例。

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...
com.eastrobot.single.A@5f72cbae
com.eastrobot.single.A@22666c7a
序列化前后两个是否同一个：false

Process finished with exit code 0
```

A类

```
public enum A {
  a;
  public A getInstance(){
      return a;
  }
}
```

看着代码量很少，我们将其编译下，代码如下：

```
public final class  A extends Enum< A> {
   public static final  A a;
   public static  A[] values();
  public static  AvalueOf(String s);
   static {};
}
```

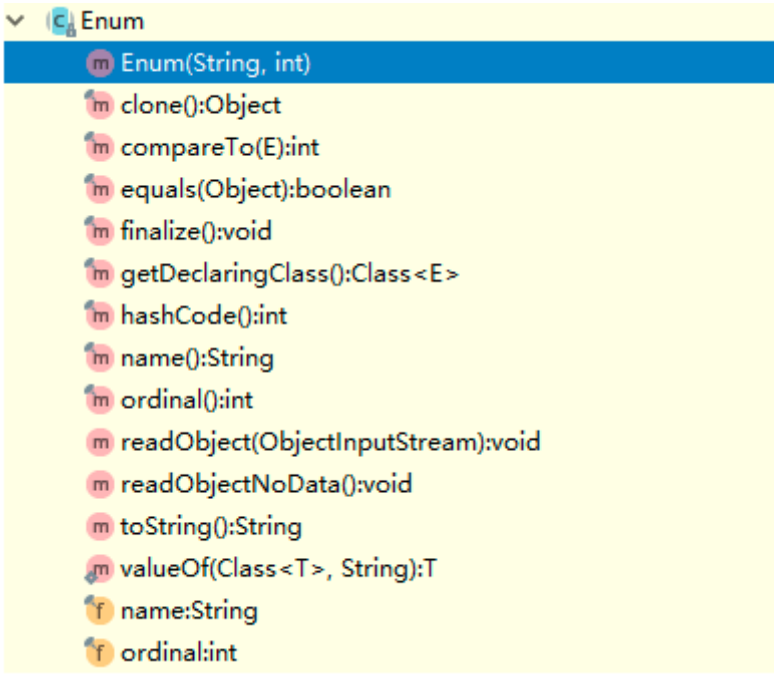
如何解决问题1？

```
public class test {
    public static void main(String[] args) throws Exception {
        A a1 = A.a;
        A a2 = A.a;
        System.out.println("正常情况下，实例化两个实例是否相同：" + (a1 == a2));

        Constructor<A> constructor = null;
        constructor = A.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        A a3 = null;
        a3 = constructor.newInstance();
        System.out.println("a1的hash:" + a1.hashCode() + ",a2的hash:" +
a2.hashCode() + ",a3的hash:" + a3.hashCode());
        System.out.println("通过反射攻击单例模式情况下，实例化两个实例是否相同：" + (a1
== a3));
    }
}
```

运行结果：

```
正常情况下，实例化两个实例是否相同：true
Exception in thread "main" java.lang.NoSuchMethodException: com.eastrobot.single.A.<init>()
    at java.lang.Class.getConstructor0(Class.java:2902)
    at java.lang.Class.getDeclaredConstructor(Class.java:2066)
    at com.eastrobot.single.test.main(test.java:12)
Disconnected from the target VM, address: '127.0.0.1:56631', transport: 'socket'

Process finished with exit code 1
```

我们看到报错了，是在寻找构造函数的时候报错的，即没有无参的构造方法，那我们看下他继承的父类ENUM有没有构造函数，看下源码，发现有个两个参数String和int类型的构造方法，我们再看下是不是构造方法的问题。



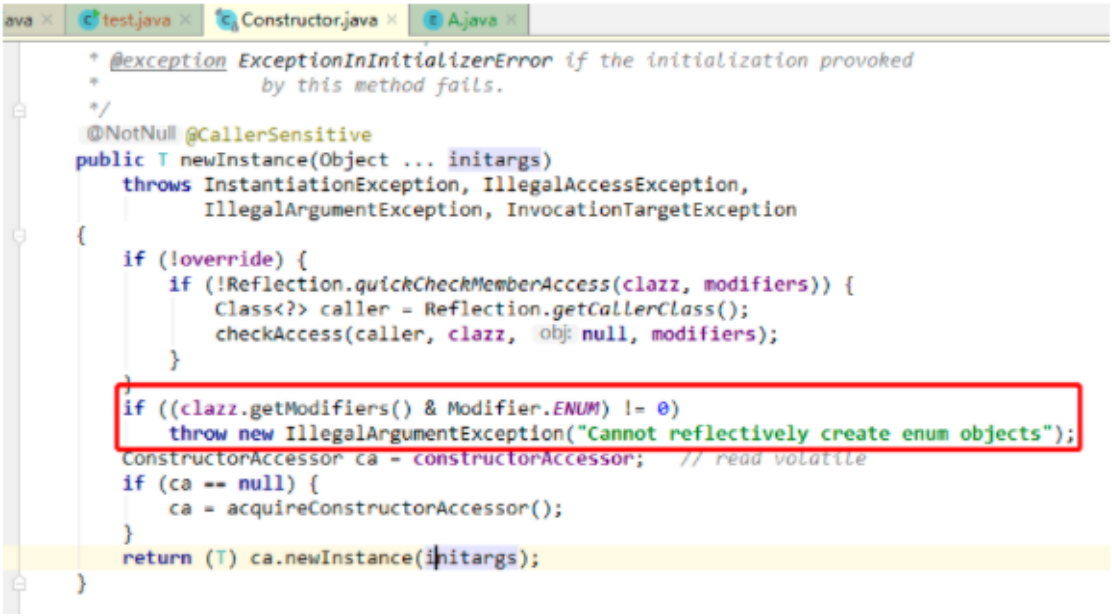
我们再用父类的有参构造方法试下，代码如下：

```
public class test {
    public static void main(String[] args) throws Exception {
        A a1 = A.a;
        A a2 = A.a;
        System.out.println("正常情况下，实例化两个实例是否相同：" + (a1 == a2));
        Constructor<A> constructor = null;
        constructor = A.class.getDeclaredConstructor(String.class,int.class);//
其父类的构造器
        constructor.setAccessible(true);
        A a3 = null;
        a3 = constructor.newInstance("学习Java的小姐姐",1);
        System.out.println("a1的hash:" + a1.hashCode() + ",a2的hash:" +
a2.hashCode() + ",a3的hash:" + a3.hashCode());
        System.out.println("通过反射攻击单例模式情况下，实例化两个实例是否相同：" + (a1
== a3));
    }
}
```

运行结果如下：

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...
正常情况下，实例化两个实例是否相同: true
Exception in thread "main" java.lang.IllegalArgumentException: Cannot reflectively create enum objects
    at java.lang.reflect.Constructor.newInstance(Constructor.java:521)
    at com.eastrobot.single.test.main(test.java:18)

Process finished with exit code 1
```

我们发现报错信息的位置已经换了，现在是已经有构造方法，而是在newInstance方法的时候报错了，我们跟下源码发现，人家已经明确写明了如果是枚举类型，直接抛出异常，代码如下，所以是无法使用反射来操作枚举类型的数据的。



如何解决问题2?

```
public class test {
    public static void main(String[] args) throws Exception {
        A s = A.a;

        //写
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("学习Java的小姐姐"));
        oos.writeObject(s);
        oos.flush();
        oos.close();
        //读
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("学习Java的小姐姐"));
        A s1 = (A)ois.readObject();
        ois.close();

        System.out.println(s+"\n"+s1);
        System.out.println("序列化前后两个是否同一个："+(s==s1));
    }
}
```

运行结果：

```
"C:\Program Files\Java\jdk1.7.0_80\bin\java.exe" ...
a
a
序列化前后两个是否同一个: true

Process finished with exit code 0
```

优点

避免了反射带来的对象不一致问题和反序列问题，简单来说，就是简单高效没问题。