**1 引言**

        相信如果能看到我这篇博客的小伙伴，肯定都看过Joshua Bloch大神说过的这句话：“单元素的枚举类型已经成为实现Singleton的最佳方法”。其实，第一次读到这句话，我连其中说的单元素指什么都不知道，尴尬。后来，网上看了搜索了好几篇文章，发现基本上都是转载自相同的一篇文章，而我的困惑是***“为什么要用枚举类型实现单例模式呢***”，文章中都说的很笼统，于是决定自己结合Joshua Bloch的《effective java》写一篇总结下，给后来的同学做个参考。

**2 什么是单例模式**

        关于什么是单例模式的定义，我之前的一篇文章（[最简单的设计模式--单例模式](https://www.cnblogs.com/chiclee/p/6475864.html)）中有写过，主要是讲恶汉懒汉、线程安全方面得问题，我就不再重复了，只是做下单例模式的总结。之前文章中实现单例模式三个主要特点：1、构造方法私有化；2、实例化的变量引用私有化；3、获取实例的方法共有。

        如果不使用枚举，大家采用的一般都是“双重检查加锁”这种方式，如下，对单例模式还不了解的同学希望先大致看下这种思路，接下来的3.1和3.2都是针对这种实现方式进行探讨，了解过单例模式的同学可以跳过直接看3.1的内容：

[复制代码](javascript:void(0);)

1 public class Singleton {

2 private volatile static Singleton uniqueInstance;

3 private Singleton() {}

4 public static Singleton getInstance() {

5 if (uniqueInstance == null) {

6 synchronized (Singleton.class){

7 if(uniqueInstance == null){//进入区域后，再检查一次，如果仍是null,才创建实例

8 uniqueInstance = new Singleton();

9 }

10 }

11 }

12 return uniqueInstance;

13 }

14 }

[复制代码](javascript:void(0);)

**3 为什么要用枚举单例**

**3.1 私有化构造器并不保险**

        《effective java》中只简单的提了几句话：“享有特权的客户端可以借助AccessibleObject.setAccessible方法，通过反射机制调用私有构造器。如果需要低于这种攻击，可以修改构造器，让它在被要求创建第二个实例的时候抛出异常。”下面我以代码来演示一下，大家就能明白：

[复制代码](javascript:void(0);)

1 public static void main(String[] args) throws NoSuchMethodException, IllegalAccessException, InvocationTargetException, InstantiationException {

2 Singleton s=Singleton.getInstance();

3 Singleton sUsual=Singleton.getInstance();

4 Constructor<Singleton> constructor=Singleton.class.getDeclaredConstructor();

5 constructor.setAccessible(true);

6 Singleton sReflection=constructor.newInstance();

7 System.out.println(s+"\n"+sUsual+"\n"+sReflection);

8 System.out.println("正常情况下，实例化两个实例是否相同："+(s==sUsual));

9 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："+(s==sReflection));

10 }

[复制代码](javascript:void(0);)

输出为：

com.lxp.pattern.singleton.Singleton@1540e19d

com.lxp.pattern.singleton.Singleton@1540e19d

com.lxp.pattern.singleton.Singleton@677327b6

正常情况下，实例化两个实例是否相同：true

通过反射攻击单例模式情况下，实例化两个实例是否相同：false

既然存在反射可以攻击的问题，就需要按照Joshua Bloch做说的，加个异常处理。这里我就不演示了，等会讲到枚举我再演示。

**3.2 序列化问题**

大家先看下面这个代码：

[复制代码](javascript:void(0);)

1 public class SerSingleton implements Serializable {

2 private volatile static SerSingleton uniqueInstance;

3 private String content;

4 public String getContent() {

5 return content;

6 }

7

8 public void setContent(String content) {

9 this.content = content;

10 }

11 private SerSingleton() {

12 }

13

14 public static SerSingleton getInstance() {

15 if (uniqueInstance == null) {

16 synchronized (SerSingleton.class) {

17 if (uniqueInstance == null) {

18 uniqueInstance = new SerSingleton();

19 }

20 }

21 }

22 return uniqueInstance;

23 }

24

25

26 public static void main(String[] args) throws IOException, ClassNotFoundException {

27 SerSingleton s = SerSingleton.getInstance();

28 s.setContent("单例序列化");

29 System.out.println("序列化前读取其中的内容："+s.getContent());

30 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("SerSingleton.obj"));

31 oos.writeObject(s);

32 oos.flush();

33 oos.close();

34

35 FileInputStream fis = new FileInputStream("SerSingleton.obj");

36 ObjectInputStream ois = new ObjectInputStream(fis);

37 SerSingleton s1 = (SerSingleton)ois.readObject();

38 ois.close();

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

40 System.out.println("序列化后读取其中的内容："+s1.getContent());

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

42 }

43

44 }

[复制代码](javascript:void(0);)

先猜猜看输出结果：

序列化前读取其中的内容：单例序列化

com.lxp.pattern.singleton.SerSingleton@135fbaa4

com.lxp.pattern.singleton.SerSingleton@58372a00

序列化后读取其中的内容：单例序列化

序列化前后两个是否同一个：false

        可以看出，序列化前后两个对象并不想等。为什么会出现这种问题呢？这个讲起来，又可以写一篇博客了，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。”当然，这个问题也是可以解决的，想详细了解的同学可以翻看《effective java》第77条：对于实例控制，枚举类型优于readResolve。

**3.3 枚举类详解**

**3.3.1 枚举单例定义**

咱们先来看一下枚举类型单例：

[复制代码](javascript:void(0);)

public enum EnumSingleton {

INSTANCE;

public EnumSingleton getInstance(){

return INSTANCE;

}

}

[复制代码](javascript:void(0);)

怎么样，是不是觉得好简单，只有这么点代码，其实也没这么简单啦，编译后相当于：

[复制代码](javascript:void(0);)

1 public final class EnumSingleton extends Enum< EnumSingleton> {

2 public static final EnumSingleton ENUMSINGLETON;

3 public static EnumSingleton[] values();

4 public static EnumSingleton valueOf(String s);

5 static {};

6 }

[复制代码](javascript:void(0);)

咱们先来验证下会不会避免上述的两个问题，先看下枚举单例的优点，然后再来讲原理。

**3.3.2 避免反射攻击**

[复制代码](javascript:void(0);)

1 public enum EnumSingleton {

2 INSTANCE;

3 public EnumSingleton getInstance(){

4 return INSTANCE;

5 }

6

7 public static void main(String[] args) throws IllegalAccessException, InvocationTargetException, InstantiationException, NoSuchMethodException {

8 EnumSingleton singleton1=EnumSingleton.INSTANCE;

9 EnumSingleton singleton2=EnumSingleton.INSTANCE;

10 System.out.println("正常情况下，实例化两个实例是否相同："+(singleton1==singleton2));

11 Constructor<EnumSingleton> constructor= null;

12 constructor = EnumSingleton.class.getDeclaredConstructor();

13 constructor.setAccessible(true);

14 EnumSingleton singleton3= null;

15 singleton3 = constructor.newInstance();

16 System.out.println(singleton1+"\n"+singleton2+"\n"+singleton3);

17 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："+(singleton1==singleton3));

18 }

19 }

[复制代码](javascript:void(0);)

结果就报异常了：

[复制代码](javascript:void(0);)

1 Exception in thread "main" java.lang.NoSuchMethodException: com.lxp.pattern.singleton.EnumSingleton.<init>()

2 at java.lang.Class.getConstructor0(Class.java:3082)

3 at java.lang.Class.getDeclaredConstructor(Class.java:2178)

4 at com.lxp.pattern.singleton.EnumSingleton.main(EnumSingleton.java:20)

5 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

6 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

7 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

8 at java.lang.reflect.Method.invoke(Method.java:498)

9 at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)

10 正常情况下，实例化两个实例是否相同：true

[复制代码](javascript:void(0);)

然后debug模式，可以发现是因为EnumSingleton.class.getDeclaredConstructors()获取所有构造器，会发现并没有我们所设置的无参构造器，只有一个参数为（String.class,int.class）构造器，然后看下Enum源码就明白，这两个参数是name和ordial两个属性：

[复制代码](javascript:void(0);)

1 public abstract class Enum<E extends Enum<E>>

2 implements Comparable<E>, Serializable {

3 private final String name;

4 public final String name() {

5 return name;

6 }

7 private final int ordinal;

8 public final int ordinal() {

9 return ordinal;

10 }

11 protected Enum(String name, int ordinal) {

12 this.name = name;

13 this.ordinal = ordinal;

14 }

15 //余下省略

[复制代码](javascript:void(0);)

        枚举Enum是个抽象类，其实一旦一个类声明为枚举，实际上就是继承了Enum，所以会有（String.class,int.class）的构造器。既然是可以获取到父类Enum的构造器，那你也许会说刚才我的反射是因为自身的类没有无参构造方法才导致的异常，并不能说单例枚举避免了反射攻击。好的，那我们就使用父类Enum的构造器，看看是什么情况:

[复制代码](javascript:void(0);)

1 public enum EnumSingleton {

2 INSTANCE;

3 public EnumSingleton getInstance(){

4 return INSTANCE;

5 }

6

7 public static void main(String[] args) throws IllegalAccessException, InvocationTargetException, InstantiationException, NoSuchMethodException {

8 EnumSingleton singleton1=EnumSingleton.INSTANCE;

9 EnumSingleton singleton2=EnumSingleton.INSTANCE;

10 System.out.println("正常情况下，实例化两个实例是否相同："+(singleton1==singleton2));

11 Constructor<EnumSingleton> constructor= null;

12 // constructor = EnumSingleton.class.getDeclaredConstructor();

13 constructor = EnumSingleton.class.getDeclaredConstructor(String.class,int.class);//其父类的构造器

14 constructor.setAccessible(true);

15 EnumSingleton singleton3= null;

16 //singleton3 = constructor.newInstance();

17 singleton3 = constructor.newInstance("testInstance",66);

18 System.out.println(singleton1+"\n"+singleton2+"\n"+singleton3);

19 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："+(singleton1==singleton3));

20 }

21 }

[复制代码](javascript:void(0);)

然后咱们看运行结果：

[复制代码](javascript:void(0);)

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:417)

at com.lxp.pattern.singleton.EnumSingleton.main(EnumSingleton.java:25)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)

[复制代码](javascript:void(0);)

        继续报异常。之前是因为没有无参构造器，这次拿到了父类的构造器了，只是在执行第17行(我没有复制import等包，所以行号少于我自己运行的代码)时候抛出异常，说是不能够反射，我们看下Constructor类的newInstance方法源码：

[复制代码](javascript:void(0);)

1 @CallerSensitive

2 public T newInstance(Object ... initargs)

3 throws InstantiationException, IllegalAccessException,

4 IllegalArgumentException, InvocationTargetException

5 {

6 if (!override) {

7 if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {

8 Class<?> caller = Reflection.getCallerClass();

9 checkAccess(caller, clazz, null, modifiers);

10 }

11 }

12 if ((clazz.getModifiers() & Modifier.ENUM) != 0)

13 throw new IllegalArgumentException("Cannot reflectively create enum objects");

14 ConstructorAccessor ca = constructorAccessor; // read volatile

15 if (ca == null) {

16 ca = acquireConstructorAccessor();

17 }

18 @SuppressWarnings("unchecked")

19 T inst = (T) ca.newInstance(initargs);

20 return inst;

21 }

[复制代码](javascript:void(0);)

请看黄颜色标注的第12行源码，说明反射在通过newInstance创建对象时，会检查该类是否ENUM修饰，如果是则抛出异常，反射失败。

**3.3.3 避免序列化问题**

 我按照3.2中方式来写，作为对比，方面大家看的更清晰些：

[复制代码](javascript:void(0);)

1 public enum SerEnumSingleton implements Serializable {

2 INSTANCE;

3 private String content;

4 public String getContent() {

5 return content;

6 }

7 public void setContent(String content) {

8 this.content = content;

9 }

10 private SerEnumSingleton() {

11 }

12

13 public static void main(String[] args) throws IOException, ClassNotFoundException {

14 SerEnumSingleton s = SerEnumSingleton.INSTANCE;

15 s.setContent("枚举单例序列化");

16 System.out.println("枚举序列化前读取其中的内容："+s.getContent());

17 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("SerEnumSingleton.obj"));

18 oos.writeObject(s);

19 oos.flush();

20 oos.close();

21

22 FileInputStream fis = new FileInputStream("SerEnumSingleton.obj");

23 ObjectInputStream ois = new ObjectInputStream(fis);

24 SerEnumSingleton s1 = (SerEnumSingleton)ois.readObject();

25 ois.close();

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

27 System.out.println("枚举序列化后读取其中的内容："+s1.getContent());

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

29 }

30 }

[复制代码](javascript:void(0);)

运行结果如下：

1 枚举序列化前读取其中的内容：枚举单例序列化

2 INSTANCE

3 INSTANCE

4 枚举序列化后读取其中的内容：枚举单例序列化

5 枚举序列化前后两个是否同一个：true

        枚举类是JDK1.5才出现的，那之前的程序员面对反射攻击和序列化问题是怎么解决的呢？其实就是像Enum源码那样解决的，只是现在可以用enum可以使我们代码量变的极其简洁了。至此，相信同学们应该能明白了为什么Joshua Bloch说的“单元素的枚举类型已经成为实现Singleton的最佳方法”了吧，也算解决了我自己的困惑。既然能解决这些问题，还能使代码量变的极其简洁，那我们就有理由选枚举单例模式了。对了，解决序列化问题，要先懂transient和readObject，鉴于我的主要目的不在于此，就不在此写这两个原理了