# 再讲单例设计模式

## 今日内容

- 单例设计模式精讲

## 01_饿汉式单例设计模式(推荐)

- A,步骤
  - 私有化构造方法
  - 声明本类对象,并使用private static修饰
  - 提供get方法访问
- B,实现

```
public class SingleClass01 {

    private SingleClass01(){}

    private static SingleClass01 instance = new SingleClass01();

    public static SingleClass01 getInstance() {
        return instance;
    }
}
```

- C,特点
  - 效率高
  - 线程安全
  - 不支持懒加载

## 02_静态内部类单例设计模式(推荐)

- A,实现

```
public class SingleClass02 {

    private SingleClass02() {
    }

    static class SingleClass02Holder {

        private static SingleClass02 instance = new SingleClass02();

    }
```

```
    public static SingleClass02 getInstance() {
        return SingleClass02Holder.instance;
    }

}
```

- B,特点
  - 效率高
  - 线程安全
  - 支持懒加载

# 03_懒汉式单例设计模式(不推荐)

- A,步骤
  - 私有化构造方法
  - 声明本类引用,并使用private static修饰
  - 提供get方法访问
- B,实现

```
public class SingleClass03 {

    private SingleClass03(){}

    private static SingleClass03 instance = null;

    public static SingleClass03 getInstance() {
        if(null == instance){
            instance = new SingleClass03();
        }
        return instance;
    }
}
```

- C,特点
  - 效率高
  - 线程不安全
  - 支持懒加载

# 04_同步懒汉式单例设计模式(不推荐)

- A,实现

```
public class SingleClass04 {

    private SingleClass04() {
    }

    private static SingleClass04 instance = null;
```

```
    public static SingleClass04 getInstance() {
        synchronized (SingleClass04.class) {
            if (null == instance) {
                instance = new SingleClass04();
            }
            return instance;
        }
    }
}
```

- B,特点
  - 效率低
  - 线程安全
  - 支持懒加载
- C,分析
  在懒汉式基础上,只是第一次创建对象的时候才会出现线程安全问题,这样加上同步代码块后,只要调用getInstance方法都会是线程同步的,这会造成其他线程无故等待.

# 05_双重锁校验单例设计模式(推荐)

- A,实现

```
public class SingleClass05 {

    private SingleClass05() {
    }

    private static SingleClass05 instance = null;

    public static SingleClass05 getInstance() {
        if (null == instance) {
            synchronized (SingleClass05.class) {
                if (null == instance) {
                    instance = new SingleClass05();
                }
            }
        }
        return instance;
    }
}
```

- B,特点
  - 效率高
  - 线程安全
  - 支持懒加载

# 06_枚举单例设计模式(推荐)

- A,实现

```
public enum SingleClass06 {
    INSTANCE;
}
```

- B,特点
  - 效率高
  - 线程安全
  - 不支持懒加载

# 07_反射/序列化攻击

- A,概念
  单例设计模式的作用,就是为了保证类在内存中只有一个对象,但是使用反射机制和序列化技术可以获取到单例类的多个实例对象.
- B,反射机制攻击(静态内部类单例设计模式)

```
try {
    Constructor<SingleClass02> c1 = SingleClass02.class.getDeclaredConstructor();
    c1.setAccessible(true);
    SingleClass02 instance1 = c1.newInstance();
    SingleClass02 instance2 = c1.newInstance();
    System.out.println(instance1 == instance2);
} catch (Exception e) {
    e.printStackTrace();
}
```

通过结果发现,反射可以创建多个实例对象,这破坏了单例设计模式的设计初衷.

- C,解决方法

```
public class SingleClass02 {

    private SingleClass02() {
        if(null != SingleClass02Holder.instance){
            throw new RuntimeException("已经创建了对象");
        }
    }

    static class SingleClass02Holder {

        private static SingleClass02 instance = new SingleClass02();

    }

    public static SingleClass02 getInstance() {
```

```
        return SingleClass02Holder.instance;
    }

}
```

- 序列化攻击(静态内部类单例设计模式)

```
try {
    SingleClass02 instance1 = SingleClass02.getInstance();
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj.obj"));
    oos.writeObject(instance1);
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.obj"));
    Object instance2 = ois.readObject();
    System.out.println(instance1 == instance2);
} catch (Exception e) {
    e.printStackTrace();
}
```

通过结果发现,给构造方法加判断无法阻止序列化攻击.

# 08_如何避免反射/序列化攻击

在六种单例设计模式中,枚举单例设计模式,可以有效地避免反射/序列化攻击

- A,反射攻击

```
try {
    Constructor<SingleClass06> c1 = SingleClass06.class.getDeclaredConstructor();
    c1.setAccessible(true);
    SingleClass06 instance1 = c1.newInstance();
    SingleClass06 instance2 = c1.newInstance();
    System.out.println(instance1 == instance2);
} catch (Exception e) {
    e.printStackTrace();
}
```

```
java.lang.NoSuchMethodException: com.qzw.single.SingleClass06.<init>()
        at java.lang.Class.getConstructor0(Class.java:3082)
        at java.lang.Class.getDeclaredConstructor(Class.java:2178)
        at com.qzw.demo.Demo03.main(Demo03.java:14)
```

结果发现,程序执行报错java.lang.NoSuchMethodException: com.qzw.single.SingleClass06.<init>(),攻击失败.通过枚举类Enum分析,得知枚举类没有无参构造方法,所以无法攻击.

- B,反射再攻击

```
try {
```

```
    Constructor<SingleClass06> c1 =
SingleClass06.class.getDeclaredConstructor(String.class,int.class);
    c1.setAccessible(true);
    SingleClass06 instance1 = c1.newInstance("hello",11);
    SingleClass06 instance2 = c1.newInstance("hello",11);
    System.out.println(instance1 == instance2);
} catch (Exception e) {
    e.printStackTrace();
}
```

```
java.lang.IllegalArgumentException: Cannot reflectively create enum objects
        at java.lang.reflect.Constructor.newInstance(Constructor.java:416)
        at com.qzw.demo.Demo04.main(Demo04.java:14)
```

答案在Constructor类的newInstance方法中

- C,序列化攻击

```
try {
    SingleClass06 instance1 = SingleClass06.INSTANCE;
    ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("obj.obj"));
    oos.writeObject(instance1);
    ObjectInputStream ois = new ObjectInputStream(new FileInputStream("obj.obj"));
    Object instance2 = ois.readObject();
    System.out.println(instance1 == instance2);
} catch (Exception e) {
    e.printStackTrace();
}
```

通过结果发现,枚举单例模式可以有效地避免序列化攻击.