如何构造一个安全的单例?

bd7xzz 于 2023-08-07 23:35:23 发布

为什么要问这个问题?

我们知道，单例是一种很常用的设计模式，主要作用就是节省 系统资源，让对象在服务器中只有一份。但是实际开发中可能有很多人压根没有写过单例这种模式，只是看过或者为了面试去写写 demo 熟悉一下。那为啥说是一种常用的模式?
其实我们用的 spring 管理对象生命周期，用到默认的 scope 就是单例。这样的场景几乎每天都在用，所以我们不需要自己手写单例了。
那么为了面试，进大厂，是不是就要刷刷文章学习学习呢？当我们刷完单例的整体结构时，会发现还是很简单的嘛，无非就是懒汉、饿汉。饿汉上来就创建，没什么难的，懒汉可能会在创建的时候线程不安全，还要防止 jvm 在 server 模式下进行指令重排，加双层锁判断就 ok 啦！面试很简单嘛，照着文章中的背下来就行了。
但是，你有没有遇到面试官问你，如何**构造一个安全的单例？注意，是安全的。**
如果你没遇到，恭喜你，面试官不想为难你，或者他没把单例玩明白。如果你遇到了，很不幸，这个面试官是个注重细节的人，而且在给你挖坑。
当然，我觉得在面试的时候这么问单例的人，可能只有我。
刚刚说了，线程不安全加锁就解决了。但是，这里安全，可不只是是线程安全，光知道线程安全没什么特殊的，无非你准备过面试。那这里还有什么猫腻?
答案：

1. 反射攻击，导致单例变成多例，不安全了
2. 序列化、反序列化变成多例，不安全了

这两点，下面再说具体为什么和怎么解决。先说说，作为面试官的我为什么这么问?
首先，我知道现在存在很多刷题网站，说用过的人，并不一定真的用过，只是刷了题，我要筛出真正会的人，而不是刷过题的人。
其次，一个单例，考察的不是一个模式这么简单，如果回答出这两个答案的人，我会认为，他的 java 基础非常好，而且考虑问题非常全面和谨慎。怎么看出来的?
基础好：我们最常用的序列化方式恐怕就是 json、xml、pb、hessian 等协议，很少有人用 java 自带的字节流序列化。用字节流序列化只有一种情况，redis 存储、消息报文投递、IO 编程时**考虑性能**，还有可能对字节进行压缩。这个时候，如果你只是对 Serializable 接口有所了解，知道 serialVersionUID 就有一些浅了。如果你知道 readResolve，那证明你对 java 序列化理解的很透彻。
考虑问题全面和谨慎：一般的人实现单例，只是满足功能就可以了，甚至不考虑懒汉的线程安全问题。如果你考虑反射攻击带来的危害，那你在做架构方案设计时，一定是很全面和谨慎的，你的方案也是可靠的。

反射攻击是什么?

如果不使用饿汉，不使用枚举做单例，那我们要么做静态内部类，要么做双重锁 + volatile 来保证线程安全。同时无论是饿汉还是懒汉，只要不用枚举，我们都需要做私有构造函数。如下：

```
//静态内部类实现方式
private Singleton(){} //不安全的点
public Singleton getInstance(){
    return Instance.INSTANCE;
}
private static class Instance{
    private static final Singleton INSTANCE = new Singleton();
}
```

```
//双重锁实现方式
private static volatile Singleton instance = null;
private Singleton(){} //不安全的点
public Singleton getInstance(){
    if(instance == null){
        synchronized(Singleton.class){
            if(instance == null){
                instance = new Singleton();
            }
        }
    }
}
```

```
//饿汉实现方式
private static final Singleton INSTANCE = new Singleton();
private Singleton(){} //不安全的点
public Singleton getInstance(){
    return INSTANCE;
}
```

上面的三种实现方式，注意私有构造函数（这里加了注释）是不安全的，本意是防止被调用者直接 new Singleton () 创建对象设置为私有，在一般情况下，这是没问题的。但是用心良苦的人，可能会这么调用你：

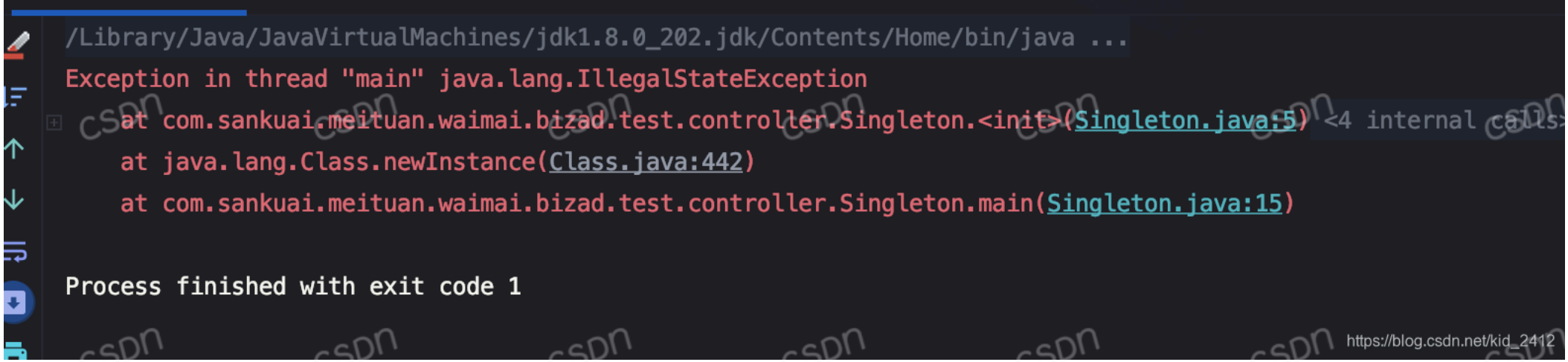
```
for(int i=0;i<10000000;i++){
    Singleton.class.newInstance();//用反射绕过私有构造函数，直接创建对象
}
```

这个时候，你的业务是不是会 **Denial of service**?
所以这很坑，但是你会说，我写的单例代码在 java 服务器内部，怎么会被人这么调用？这是不可能发生的！没错，这没问题。如果你的代码是开源的，你怎么知道那些内心黑暗的人会不会从某个 http 接口伪造什么东西来触发 newInstance ()？仔细想想，这两年有多少人被 FastJSON 坑的大晚上不能安心睡觉，要紧急升级代码？恐怕开发阿里爸爸 FastJSON 团队，也不想出现这样的状况。
所以我们要严谨！

```
private Singleton(){
    throw new IllegalStateException();
}
```

当然，这个时候是无法使用双重锁 + volatile 方式创建单例的，因为自身调用也会抛异常。所以直接用静态内部类方式解决问题。

```
//静态内部类实现方式
private Singleton(){
    if(Instance.INSTANCE!=null){
        throw new IllegalStateException();//这下安全了
    }
}
public Singleton getInstance(){
    return Instance.INSTANCE;
}
private static class Instance{
    private static final Singleton INSTANCE = new Singleton();
}
```



序列化反序列化会怎样?

直接上代码

```
1  public class Singleton implements Serializable{
2      private Singleton() {
3          if (Instance.INSTANCE != null) {//这里我可是防止了反射攻击哦!
4              throw new IllegalStateException();
5          }
6      }
7
8      public static Singleton getInstance() {
9          return Instance.INSTANCE;
10     }
11
12     private static class Instance {
13         private static final Singleton INSTANCE = new Singleton();
14     }
15
16     public static void main(String[] args) throws IOException, ClassNotFoundException {
17         File file = new File("~/Desktop/Singleton.bin");
18         Singleton singleton = Singleton.getInstance();
19         ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));
20         oos.writeObject(singleton);
21         ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));
```

```
        Singleton singleton1 = (Singleton) ois.readObject();
        System.out.println(singleton == singleton1);
    }
}
```

这时候，是不是又会 **Denial of service**?

为啥会这样？其实很简单，只要实现 Serializable 接口的类对象，ObjectOutputStream 会毫不犹豫的吐成字节，或者读回来，它才不管你是不是单例，当然它也不知道内存中有这么一个单例，直接在内存中创建对象了。

如何解决这个问题?

```
public class Singleton implements Serializable{
    private Singleton() {
        if (Instance.INSTANCE != null) {
            throw new IllegalStateException();
        }
    }

```

```
    public static Singleton getInstance() {
        return Instance.INSTANCE;
    }
    //实现readResolve接口就ok了
    private Object readResolve() throws ObjectStreamException {
        return Instance.INSTANCE;
    }

    private static class Instance {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        File file = new File("/Users/baodi/Desktop/Singleton.bin");
        Singleton singleton = Singleton.getInstance();
        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file));
        oos.writeObject(singleton);
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file));
        Singleton singleton1 = (Singleton) ois.readObject();
        System.out.println("源对象singleton == 反序列化对象 singleton1吗? "+(singleton == singleton1));
    }
```

实现 readResolve，返回静态内部类的对象就可以了。

看看源码（我用的 jdk1.8，1.6、1.7 也一样）ObjectOutputStream.readObject () 中会调用内部私有方法 readObject0 ()，其中 byte tc 是把对象头读出来

```
byte tc;
while ((tc = bin.peekByte()) == TC_RESET) {
    bin.readByte();
    handleReset();
}
```


通过 switch case (tc) 判断对象的类型，这里我们用的是 OBJECT 类型，魔数为 0x73

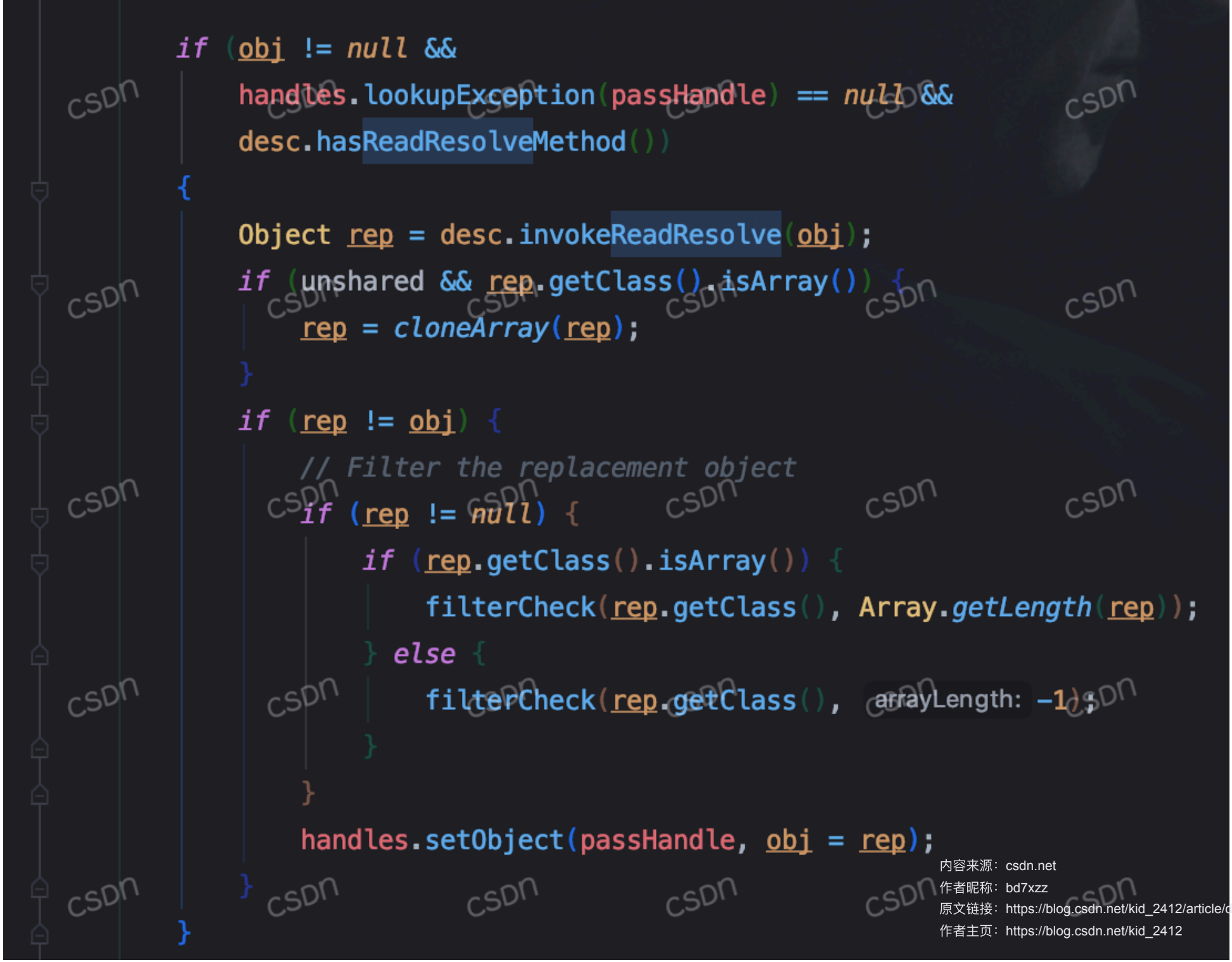
```
 * new Object.
 */
final static byte TC_OBJECT =      (byte)0x73;
```

```
00000000: aced 0005 7372 003a 636f 6d2e 7361 6e6b  ....sr.:com.sank
00000010: 7561 692e 6d65 6974 7561 6e2e 7761 696d  uai.meituan.waim
00000020: 6169 2e62 697a 6164 2e74 6573 742e 636f  ai.bizad.test.co
00000030: 6e74 726f 6c6c 6572 2e53 696e 676c 6574  ntroller.Singlet
00000040: 6f6e 4854 48b4 4334 46a9 0200 0078 70    onHTH.C4F....xp
```

这时候，会调用内部私有的 readOrdinaryObject () 方法

```
case TC_OBJECT:
    return checkResolve(readOrdinaryObject(unshared));
```

这里就是调用我们重写的 readResolve () 方法啦！



```
if (obj != null &&
    handles.lookupException(passHandle) == null &&
    desc.hasReadResolveMethod())
{
    Object rep = desc.invokeReadResolve(obj);
    if (unshared && rep.getClass().isArray()) {
        rep = cloneArray(rep);
    }
    if (rep != obj) {
        // Filter the replacement object
        if (rep != null) {
            if (rep.getClass().isArray()) {
                filterCheck(rep.getClass(), Array.getLength(rep));
            } else {
                filterCheck(rep.getClass(), arrayLength: -1);
            }
        }
        handles.setObject(passHandle, obj = rep);
    }
}
```

这就是 jdk 的大佬们为提供的一个 hook 方法，我们可以用它保证序列化和反序列化的安全。
有人一定会说，你有病，序列化一个单例！不，我没病，只是你没用到过而已
有人也会说，用枚举不就屏蔽这些问题了吗？不，如果 JDK1.6 的情况下是不能把枚举当做单例对象玩的。

好了，到这里就结束了吗?
不，记得给你的单例类加上 final，防止被继承后重写！

结论

1. 面试不只是刷刷题就 ok 了
2. 请认真的对待你写过的每一个代码，因为你很可能把别人坑了，比如 FastJSON
3. 做技术要刨根问底，网上的资料都是说如何序列化不安全的问题，并没有给出 ObjectOutputStream.readObject () 执行原理分析，不信你搜
4. 严谨、夯实基础