

- 1、简介
- 2、构建思想
- 3、创建方式
 - 3.0、固定测试类
 - 3.1、饿汉式
 - 3.2、懒汉式-线程不安全
 - 3.3、懒汉式-Synchronize
 - 3.4、懒汉式-双重检查锁
 - 3.5、懒汉式-volatile
 - 3.6、懒汉式-静态内部类
- 4、单例破坏
 - 4.1、序列化和反序列化
 - 4.1.1、破坏演示
 - 4.1.2、原因分析
 - 4.2、反射
 - 4.2.1、破坏演示
 - 4.2.2、问题解决
 - 4.2.3、多说一句

1、简介

所谓类的**单例设计模式**（`singleton`），就是采取了一定的方法保证在整个的软件系统中，对**某个类只能存在一个对象实例**，并且该类只提供一个**取得其对象实例的静态方法**。由于单例模式只生成了一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

2、构建思想

如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将**类的构造器的访问权限**设置成 `private`，这样就不能通过 `new` 操作符在累的外部产生类的对象了，但在类内部仍可以产生该类的对象。又因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。我将其总结成以下几个步骤：

1. 将类的构造器的访问权限设置成 `private`
2. 将指向类内部的该类对象的变量定义成 `static` 静态的
3. 将获取该对象实例方法定义成 `static` 静态的

3、创建方式



单例设计模式分类两种：

饿汉式：类加载就会导致该单实例对象被创建

懒汉式：类加载不会导致该单实例对象被创建，而是首次使用该对象时才会创建

3.0、固定测试类



在后续创建方式中，如果没有单独编写测试类，则统一使用这一测试类

```
1 public class SingletonText1 {
2     public static void main(String[] args) {
3         Singleton singleton1 = Singleton.getInstance();
4         Singleton singleton2 = Singleton.getInstance();
5         System.out.println(singleton1 == singleton2);    //控制台打
        印true，则证明两个是同一个对象
6     }
7 }
```

3.1、饿汉式

该方式在成员位置声明Singleton类型的静态变量，并创建Singleton类的对象instance。instance对象是随着类的加载而创建的。如果该对象足够大的话，而一直没有使用就会造成内存的浪费。

```
1 class Singleton {
2
3     //1. 私有化类的构造器
4     private Singleton() {
5
6     }
7     //2. 内部创建类的对象，并且需要设置成 static，因为静态方法中只能调用
    static 属性，并且需要控制只有一个对象
8     private static Singleton instance = new Singleton();
9
10    //3. 由于构造器进行了私有化不能直接 new， 因此需要提供公共的静态的方
    法，返回类的对象
11    public static Singleton getInstance() {
12        return instance;
13    }
14 }
```

3.2、懒汉式-线程不安全

该方式在成员位置声明Singleton类型的静态变量，并没有进行对象的赋值操作，当调用 `getInstance()` 方法获取 `Singleton类` 的对象的时候才创建Singleton类的对象，这样就实现了懒加载的效果。但是，如果是多线程环境，会出现线程安全问题。

```
1 public class Singleton {
2     // 私有构造方法
3     private Singleton() {}
4
5     // 在成员位置创建该类的对象
6     private static Singleton instance;
7
8     //对外提供静态方法获取该对象
9     public static Singleton getInstance() {
10
11         if(instance == null) {
12             instance = new Singleton();
13         }
14         return instance;
15     }
16 }
```

3.3、懒汉式-Synchronize

该方式也实现了懒加载效果，同时又解决了线程安全问题。但是在 `getInstance()` 方法上添加了 `synchronized` 关键字，导致该方法的执行效果特别低。与此同时，这一方法并不是一定确保了线程安全，在初始化instance的时候依旧可能会出现线程安全问题，一旦初始化完成就不存在了。

```
1 public class Singleton {
2     // 私有构造方法
3     private Singleton() {}
4
5     // 在成员位置创建该类的对象
6     private static Singleton instance;
7
8     // 对外提供静态方法获取该对象
9     public static synchronized Singleton getInstance() {
10         // 判断是否已经实例化
11         if(instance == null) {
12             instance = new Singleton();
13         }
14         return instance;
15     }
16 }
```

3.4、懒汉式-双重检查锁

使用双重检查锁机制后运行顺序就成了：

1. 检查变量是否被初始化(不去争夺锁)，如果已被初始化则立即返回。
2. 获取锁。
3. 再次检查变量是否已经被初始化，如果还没被初始化就初始化一个对象。

执行双重检查是因为，如果多个线程同时通过了第一次检查，并且其中一个线程首先通过了第二次检查并实例化了对象，那么剩余通过了第一次检查的线程就不会再去实例化对象。

这样，除了初始化的时候会出现加锁的情况，后续的所有调用都会避免加锁而直接返回，解决了性能消耗的问题。

```
1 public class Singleton {
2
3     // 私有构造方法
4     private Singleton() {}
5
6     private static Singleton instance;
7
8     // 对外提供静态方法获取该对象
9     public static Singleton getInstance() {
10         // 第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实例
11         if(instance == null) {
12             synchronized (Singleton.class) {
13                 // 抢到锁之后再次判断是否为null
14                 if(instance == null) {
15                     instance = new Singleton();
16                 }
17             }
18         }
19         return instance;
20     }
21 }
```

3.5、懒汉式-volatile

双重检查锁模式是一种非常好的单例实现模式，解决了单例、性能、线程安全问题，上面的双重检测锁模式看上去完美无缺，其实是存在问题，在多线程的情况下，可能会出现NPE问题，出现问题的原因是JVM在实例化对象的时候会进行优化和指令重排序操作。

要解决双重检查锁模式带来NPE的问题，只需要使用 `volatile` 关键字，`volatile` 关键字可以保证可见性和有序性。

```
1 public class Singleton {
2
3     //私有构造方法
```

```

4     private Singleton() {}
5
6     private static volatile Singleton instance;
7
8     //对外提供静态方法获取该对象
9     public static Singleton getInstance() {
10         //第一次判断，如果instance不为null，不进入抢锁阶段，直接返回实际
11         if(instance == null) {
12             synchronized (Singleton.class) {
13                 //抢到锁之后再次判断是否为空
14                 if(instance == null) {
15                     instance = new Singleton();
16                 }
17             }
18         }
19         return instance;
20     }
21 }

```

3.6、懒汉式-静态内部类

静态内部类单例模式中实例由内部类创建，由于 JVM 在加载外部类的过程中，是不会加载静态内部类的，只有内部类的属性/方法被调用时才会被加载，并初始化其静态属性。静态属性由于被 `static` 修饰，保证只被实例化一次，并且严格保证实例化顺序。

第一次加载Singleton类时不会去初始化INSTANCE，只有第一次调用 `getInstance`，虚拟机加载 `SingletonHolder` 并 初始化 `INSTANCE`。在没有加任何锁的情况下，保证了多线程下的安全，并且没有任何性能影响和空间的浪费。

```

1 public class Singleton {
2
3     //私有构造方法
4     private Singleton() {}
5
6     private static class SingletonHolder {
7         private static final Singleton INSTANCE = new Singleton();
8     }
9
10    //对外提供静态方法获取该对象
11    public static Singleton getInstance() {
12        return SingletonHolder.INSTANCE;
13    }
14 }

```

4、单例破坏

虽然在上面的介绍中都有对单例模式的创建方式及改进做了一个介绍，但是按照目前的版本任然存在问题，一些特殊的场景下可能会对单例进行破坏。以下两种情况可导致单例破坏：

- 序列化/反序列化；
- 反射。

4.1、序列化和反序列化

序列化意义是将实现序列化的Java对象转换成字节序列，这些字节序列可以被保存在磁盘上，或者通过网络传输。以备以后重新恢复成原来的对象。

对于单例类使用序列化、反序列化操作时，会破坏单例（**序列化前的对象和反序列化后得到的对象内存地址不同**）

4.1.1、破坏演示

```
1 package top.xbaoziplus.singleton;
2
3 import java.io.*;
4
5 public class Singleton implements Serializable {
6
7     //私有构造方法
8     private Singleton() {}
9
10    private static class SingletonHolder {
11        private static final Singleton INSTANCE = new Singleton();
12    }
13
14    //对外提供静态方法获取该对象
15    public static Singleton getInstance() {
16        return SingletonHolder.INSTANCE;
17    }
18
19    /**
20     * 测试
21     */
22    public static void main(String[] args) throws Exception {
23        // 往文件中写对象
24        writeObject2File();
25        // 从文件中读取对象
26        Singleton s1 = readObjectFromFile();
27        Singleton s2 = readObjectFromFile();
28
29        //判断两个反序列化后的对象是否是同一个对象
30        System.out.println(s1);
31        System.out.println(s2);
32        System.out.println(s1 == s2);
33    }
```

```

34
35     /**
36     * 从文件中反序列化对象
37     */
38     private static Singleton readObjectFromFile() throws Exception
39     {
40         // 创建对象输入流对象
41         ObjectInputStream ois = new ObjectInputStream(new
42         FileInputStream("D:\\a.txt"));
43         // 第一个读取Singleton对象
44         Singleton instance = (Singleton) ois.readObject();
45
46         return instance;
47     }
48
49     /**
50     * 序列化到文件
51     */
52     public static void writeObject2File() throws Exception {
53         // 获取Singleton类的对象
54         Singleton instance = Singleton.getInstance();
55         // 创建对象输出流
56         ObjectOutputStream oos = new ObjectOutputStream(new
57         FileOutputStream("D:\\a.txt"));
58         // 将instance对象写出到文件中
59         oos.writeObject(instance);
60     }
61 }

```

运行结果显示通过序列化前后对象不同，表明单例已经被破坏了

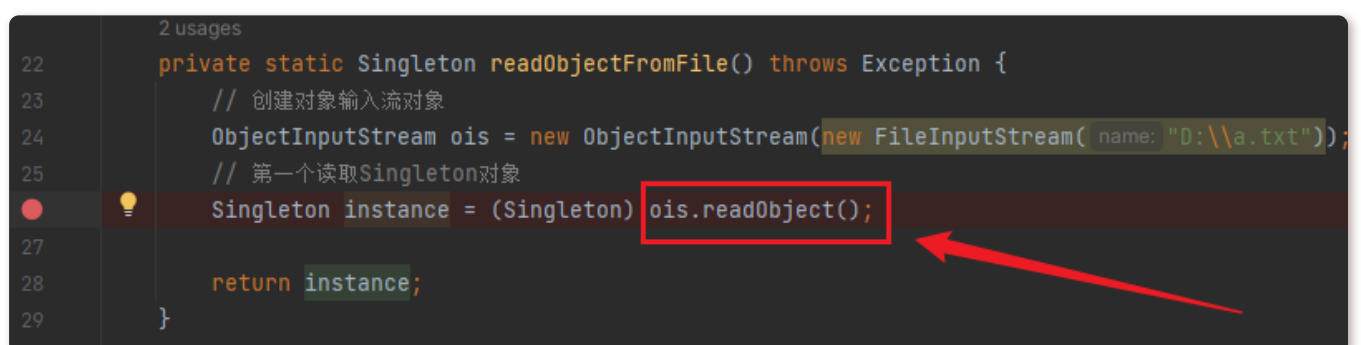
```

1 top.xbaoziplus.singleton.Singleton@5c7fa833
2 top.xbaoziplus.singleton.Singleton@39aeed2f
3 false

```

4.1.2、原因分析

在学习过程中其实并没有接触到这个原因的分析，这一块空白自己好像又不太心甘的样子，因此，配合上资料加查看源码，大致弄清了其原因，以下便针对源码进行分析。从代码中的读取输入流方法 `readObject()` 入手，逐个查看其底层源码。



```

22 private static Singleton readObjectFromFile() throws Exception {
23     // 创建对象输入流对象
24     ObjectInputStream ois = new ObjectInputStream(new FileInputStream( name: "D:\\a.txt"));
25     // 第一个读取Singleton对象
26     Singleton instance = (Singleton) ois.readObject();
27
28     return instance;
29 }

```

一路点击 `readObject()` 方法直到 `readObject0()` 方法，点击进去

```
496 // if nested read, passHandle contains handle of enclosing object
497 int outerHandle = passHandle;
498 try {
499     Object obj = readObject0(type, unshared: false);
500     handles.markDependency(outerHandle, passHandle);
501     ClassNotFoundException ex = handles.lookupException(passHandle);
502     if (ex != null) {
503         throw ex;
504     }
}
```

找到switch通道中的对象入口，返回值会调用 `readOrdinaryObject` 方法，点进去 `readOrdinaryObject()`

```
1687
1688 case TC_OBJECT:
1689     if (type == String.class) {
1690         throw new ClassCastException("Cannot cast an object to java.lang.String");
1691     }
1692     return checkResolve(readOrdinaryObject(unshared));
1693
1694 case TC_EXCEPTION:
```

方法中通过三目允许算符判断了对象是否可实例化，如果是可实例化的会通过 `newInstance()` 方法反射实例化一个新的对象，所以序列化前的对象和反序列化后得到的对象不同！

```
2205
2206 ObjectStreamClass desc = readClassDesc(unshared: false);
2207 desc.checkDeserialize();
2208
2209 Class<?> cl = desc.forClass();
2210 if (cl == String.class || cl == Class.class
2211     || cl == ObjectStreamClass.class) {
2212     throw new InvalidClassException("invalid class descriptor");
2213 }
2214
2215 Object obj;
2216 try {
2217     obj = desc.isInstantiable() ? desc.newInstance() : null;
2218 } catch (Exception ex) {
2219     throw (IOException) new InvalidClassException(
```



这时问题出现的原因就找到了，序列化/反序列化导致单例破坏的原因竟是因为其底层使用到了反射，而反射则是另外一个造成单例破坏的原因之一。接下来将针对反射问题作出对应的解决方案。

4.2、反射

4.2.1、破坏演示

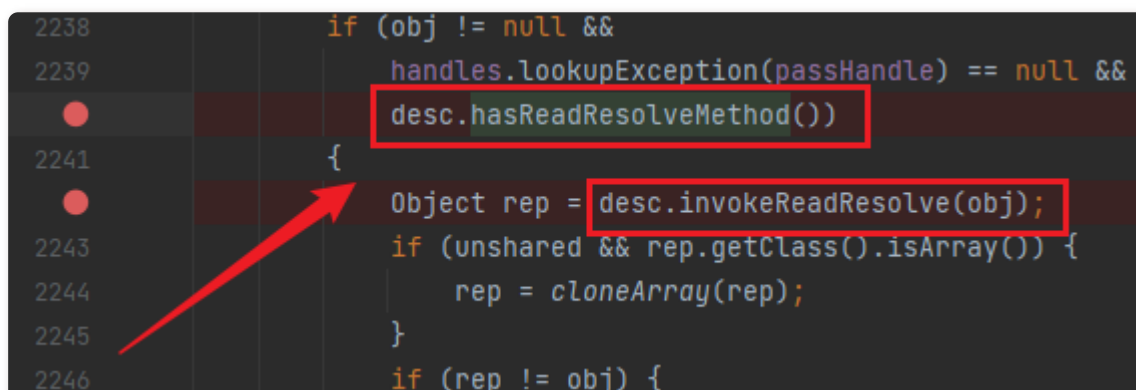
```
1 public class SingletonTest {
2     public static void main(String[] args) throws Exception {
3         //获取Singleton类的字节码对象
4         Class clazz = Singleton.class;
5         //获取Singleton类的私有无参构造方法对象
6         Constructor constructor = clazz.getDeclaredConstructor();
7         //取消访问检查
8         constructor.setAccessible(true);
9
10        //创建Singleton类的对象s1
11        Singleton s1 = (Singleton) constructor.newInstance();
12        //创建Singleton类的对象s2
13        Singleton s2 = (Singleton) constructor.newInstance();
14
15        //判断通过反射创建的两个Singleton对象是否是同一个对象
16        System.out.println(s1);
17        System.out.println(s2);
18        System.out.println(s1 == s2);
19    }
20 }
```

运行结果显示通过序列化前后对象不同，表明单例已经被破坏了

```
1 top.xbaoziplus.singleton.Singleton@6a5fc7f7
2 top.xbaoziplus.singleton.Singleton@3b6eb2ec
3 false
```

4.2.2、问题解决

重新根据4.1.2中的源码分析，往下翻的时候我们可以看到下面这么一段代码，在通过三目运算创建了对象之后，还会去找这个对象里是否有 `readResolve()` 方法，如果有，则通过这方法返回对象。



```
2238         if (obj != null &&
2239             handles.lookupException(passHandle) == null &&
2240             desc.hasReadResolveMethod())
2241         {
2242             Object rep = desc.invokeReadResolve(obj);
2243             if (unshared && rep.getClass().isArray()) {
2244                 rep = cloneArray(rep);
2245             }
2246             if (rep != obj) {
```

这样问题的解决方案就很明确了，既然会判断是否有 `readResolve()` 方法，那么我们主动在需要单例的类中创建这一个方法，并在该方法中获取对象实例不就好了。这里需要注意的是这一方法需要返回的数据类型为 `Object` 类型；

```
1 public class Singleton implements Serializable {
2
3     //私有构造方法
4     private Singleton() {}
5
6     private static class SingletonHolder {
7         private static final Singleton INSTANCE = new Singleton();
8     }
9
10    //对外提供静态方法获取该对象
11    public static Singleton getInstance() {
12        return SingletonHolder.INSTANCE;
13    }
14
15    /**
16     * 解决反射破坏单例问题
17     */
18    private Object readResolve() {
19        return getInstance();
20    }
21 }
```

4.2.3、多说一句

这里需要注意的是，在4.2.1中的破坏演示中是直接获取其无参构造函数进行生成对象的，这种情况是无法自动判断是否存在 `readResolve()` 方法的，当然，我们可以模拟源码中 `ObjectStreamClass desc = readClassDesc(false);` 的方式一步步判断从而执行 `readResolve()` 方法。

但这种方法毕竟治标不治本，因此我们可以通过抛异常的方式来提示用户不允许破坏规则生成多个对象实例。



注意的是在构造器中不需要加锁甚至双重检查锁，因为这是懒汉单例，在 `getInstance()` 方法中我们已经写好的双重检查锁已经帮我们把关了，避免了线程安全问题，无需重复加锁，做无用功的同时还影响了性能和资源消耗。

```
1 public class Singleton {
2
3     //私有构造方法
4     private Singleton() {
5         /*
6         反射破解单例模式需要添加的代码
```

```
7         */
8         if(instance != null) {
9             throw new RuntimeException();
10        }
11    }
12
13    private static volatile Singleton instance;
14
15    //对外提供静态方法获取该对象
16    public static Singleton getInstance() {
17
18        if(instance != null) {
19            return instance;
20        }
21
22        synchronized (Singleton.class) {
23            if(instance != null) {
24                return instance;
25            }
26            instance = new Singleton();
27            return instance;
28        }
29    }
30 }
```