

课程目标

内容定位

1. 单例模式

1.1. 单例模式的应用场景

1.2. 饿汉式单例模式

1.3. 懒汉式单例模式

1.4. 反射破坏单例

1.5. 序列化破坏单例

1.6. 注册式单例模式

1.6.1. 枚举式单例

1.6.2. 容器式单例

1.7. 线程单例实现ThreadLocal

1.8. 单例模式小结

课程目标

- 1、掌握单例模式的应用场景。
- 2、掌握 IDEA 环境下的多线程调试方式。
- 3、掌握保证线程安全的单例模式策略。
- 4、掌握反射暴力攻击单例解决方案及原理分析。
- 5、序列化破坏单例的原理及解决方案。
- 6、掌握常见的单例模式写法。

内容定位

- 1、听说过单例模式，但不知道如何应用的人群。
- 2、单例模式是非常经典的高频面试题，希望通过面试单例彰显技术深度，顺利拿到 Offer 的人群。

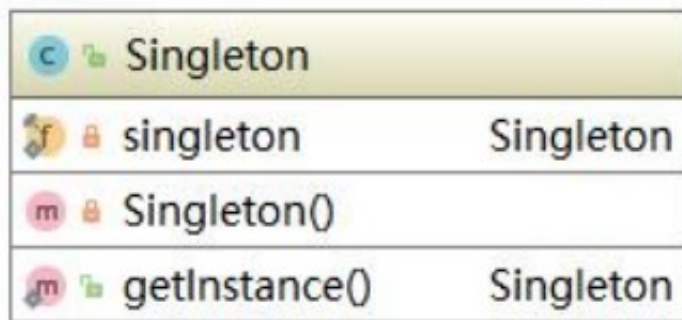
1. 单例模式

1.1. 单例模式的应用场景

单例模式（Singleton Pattern）是指确保一个类在任何情况下都绝对只有一个实例，并提供一个全局访问点。单例模式是创建型模式。单例模式在现实生活中应用也非常广泛，例如，公司 CEO、部门经理等。J2EE 标准中的 ServletContext、ServletContextConfig 等、Spring 框架应用中的 ApplicationContext、数据库的连接池等也都是单例形式。

1.2. 饿汉式单例模式

先来看单例模式的类结构图，如下图所示。



饿汉式单例模式在类加载的时候就立即初始化，并且创建单例对象。它绝对线程安全，在线程还没出现以前就实例化了，不可能存在访问安全问题。

优点：没有加任何锁、执行效率比较高，用户体验比懒汉式单例模式更好。

缺点：类加载的时候就初始化，不管用与不用都占着空间，浪费了内存，有可能“占着茅坑不拉屎”。Spring 中 IoC 容器 ApplicationContext 本身就是典型的饿汉式单例模式。接下来看一段代码：

```
1 package cn.sitedev.hungry;
2
3 public class HungrySingleton {
4     private static final HungrySingleton INSTANCE = new HungrySingleton();
5
6     private HungrySingleton() {
7     }
8
9     public static HungrySingleton getInstance() {
10         return INSTANCE;
11     }
12 }
```

```
13 }
```

还有另外一种写法，利用静态代码块的机制：

```
1 package cn.sitedev.hungry;
2
3 /**
4  * 饿汉式静态代码块单例
5  */
6 public class HungrySingletonWithStaticBlock {
7     private static final HungrySingletonWithStaticBlock INSTANCE;
8
9     static {
10         INSTANCE = new HungrySingletonWithStaticBlock();
11     }
12
13     private HungrySingletonWithStaticBlock() {
14     }
15
16     public static HungrySingletonWithStaticBlock getInstance() {
17         return INSTANCE;
18     }
19
20 }
```

这两种写法都非常简单，也非常好理解，饿汉式单例模式适用于单例对象较少的情况。下面我们来看性能更优的写法。

1.3. 懒汉式单例模式

懒汉式单例模式的特点是：被外部类调用的时候内部类才会加载。下面看懒汉式单例模式的简单实现 LazySimpleSingleton：

```
1 package cn.sitedev.lazy;
2
```

```

3  /**
4   * 懒汉式单例模式在外部需要使用的时候才进行实例化
5   */
6  public class LazySimpleSingleton {
7      private static LazySimpleSingleton instance;
8
9      private LazySimpleSingleton() {
10     }
11
12     public static LazySimpleSingleton getInstance() {
13         if (instance == null) {
14             instance = new LazySimpleSingleton();
15         }
16         return instance;
17     }
18 }

```

然后写一个线程类 ExecutorThread :

```

1  package cn.sitedev.lazy;
2
3  public class ExecutorThread implements Runnable {
4      @Override
5      public void run() {
6          LazySimpleSingleton singleton = LazySimpleSingleton.getInstance();
7          System.out.println(Thread.currentThread().getName() + ":" + singleton);
8      }
9  }

```

客户端测试代码如下 :

```

1  package cn.sitedev.lazy;
2
3  public class LazySimpleSingletonTest {
4      public static void main(String[] args) {
5          Thread t1 = new Thread(new ExecutorThread());

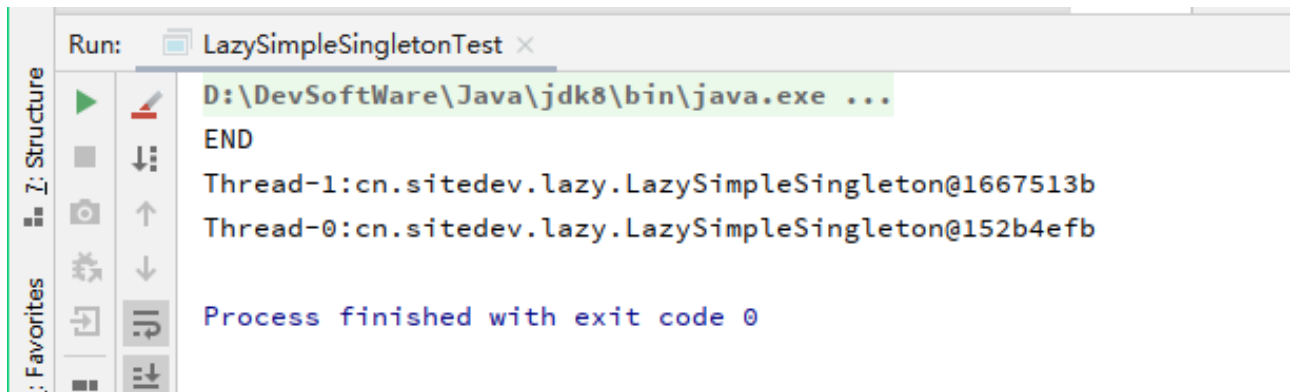
```

```

6      Thread t2 = new Thread(new ExecutorThread());
7      t1.start();
8      t2.start();
9      System.out.println("END");
10 }
11 }

```

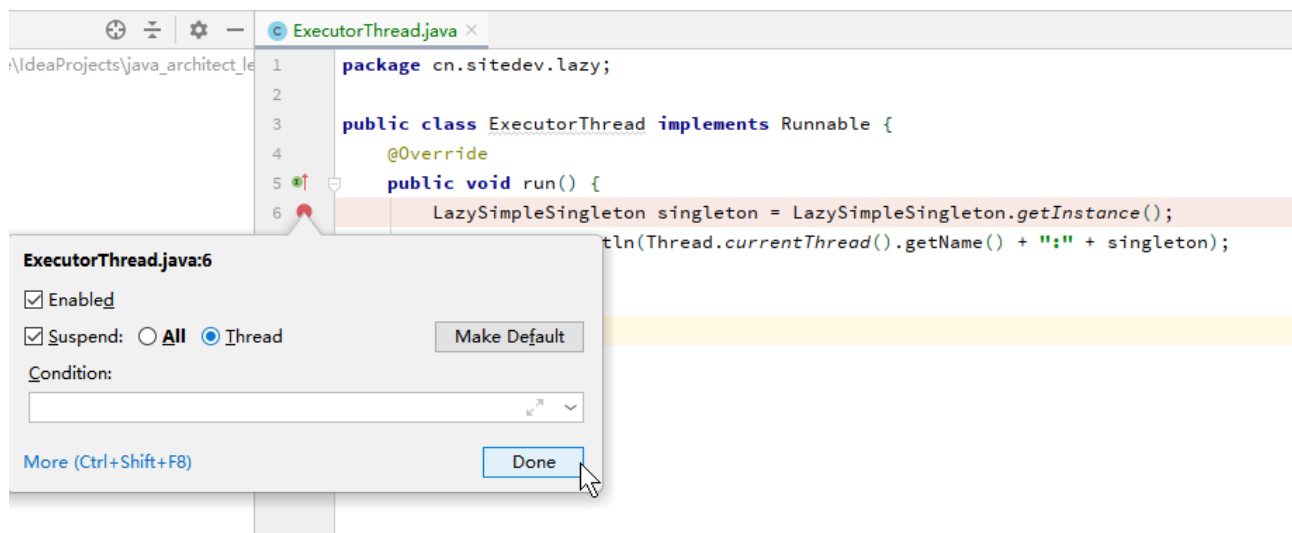
运行结果如下图所示。



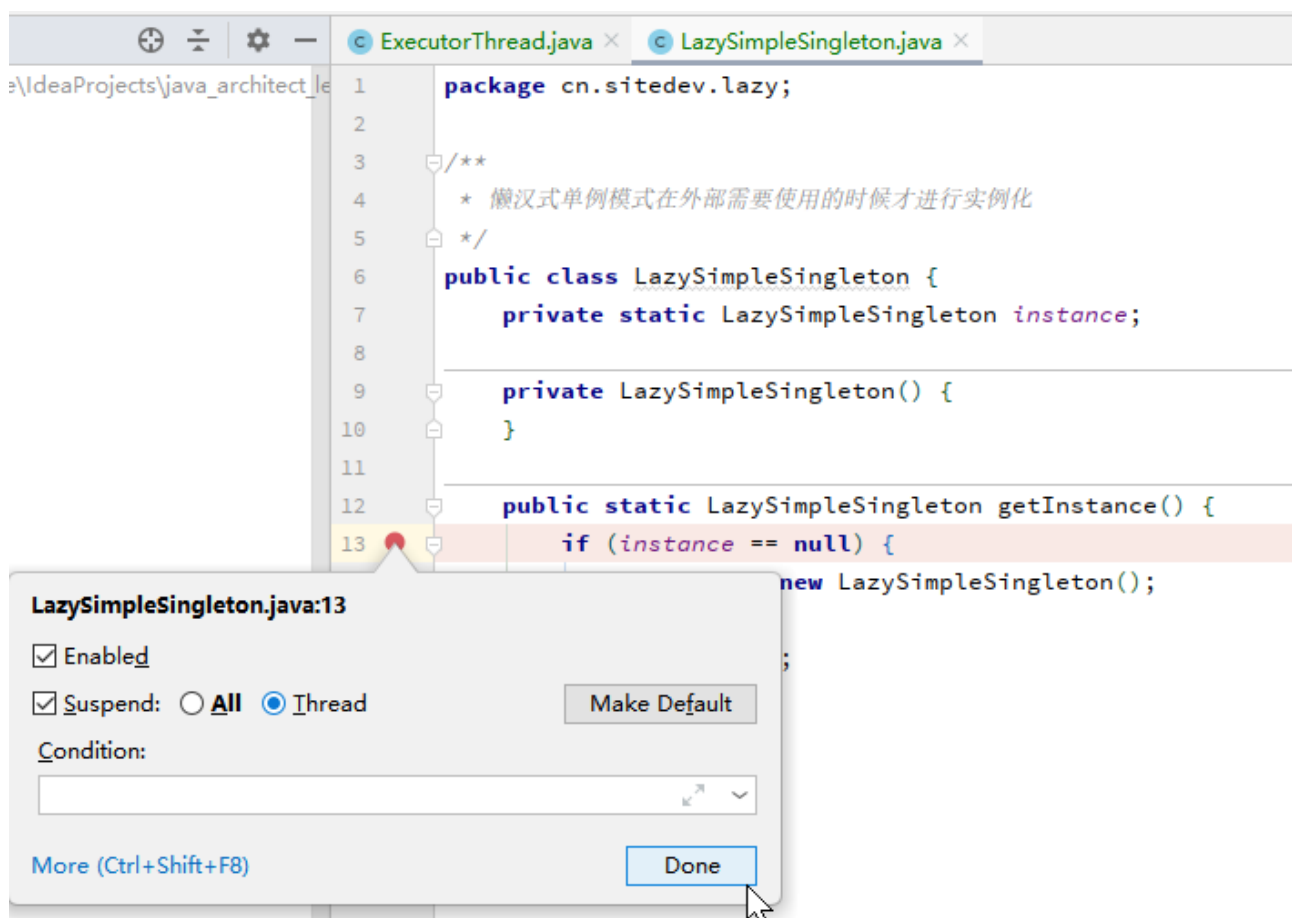
上面的代码有一定概率出现两种不同结果，这意味着上面的单例存在线程安全隐患。我们通过调试运行再具体看一下。这里教大家一种新技能，用线程模式调试，手动控制线程的执行顺序来跟踪内存的变化。先给 ExecutorThread 类打上断点，如下图所示。



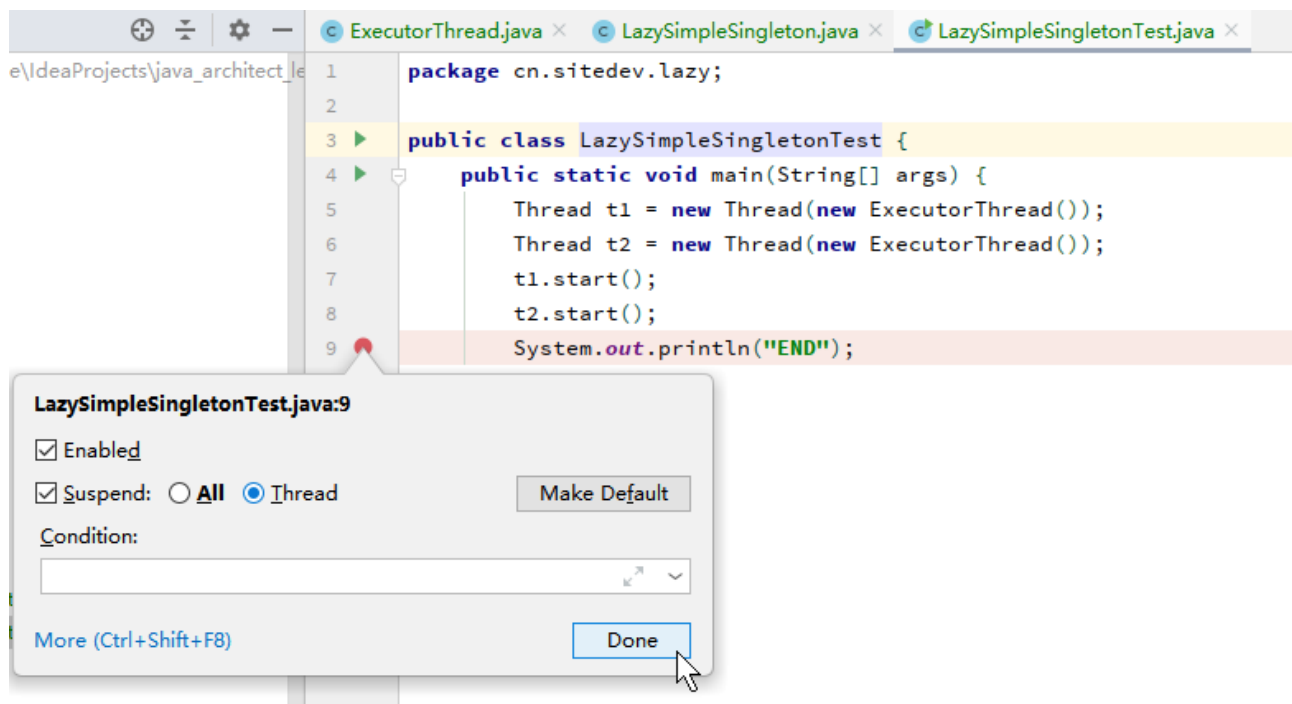
使用鼠标右键单击断点，切换为 Thread 模式，如下图所示。



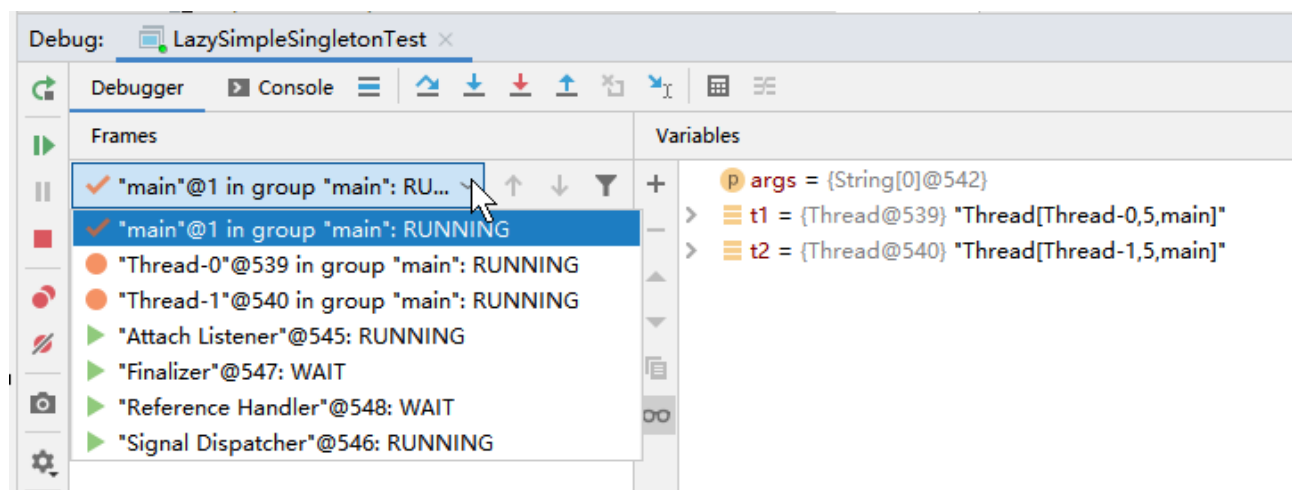
然后给 LazySimpleSingleton 类打上断点，同样标记为 Thread 模式，如下图所示。



切回客户端测试代码，同样也打上断点，同时改为 Thread 模式，如下图所示。



开始“Debug”之后，会看到 Debug 控制台可以自由切换 Thread 的运行状态，如下图所示。



通过不断切换线程，并观测其内存状态，我们发现在线程环境下

LazySimpleSingleton 被实例化了两次。有时我们得到的运行结果可能是相同的两个对象，实际上是被后面执行的线程覆盖了，我们看到了一个假象，线程安全隐患依旧存在。那么，我们如何来优化代码，使得懒汉式单例模式在线程环境下安全呢？来看下面的代码，给 getInstance() 加上 synchronized 关键字，使这个方法变成线程同步方法：

```
1 package cn.sitedev.lazy.threadsafe;
2
3 /**
```

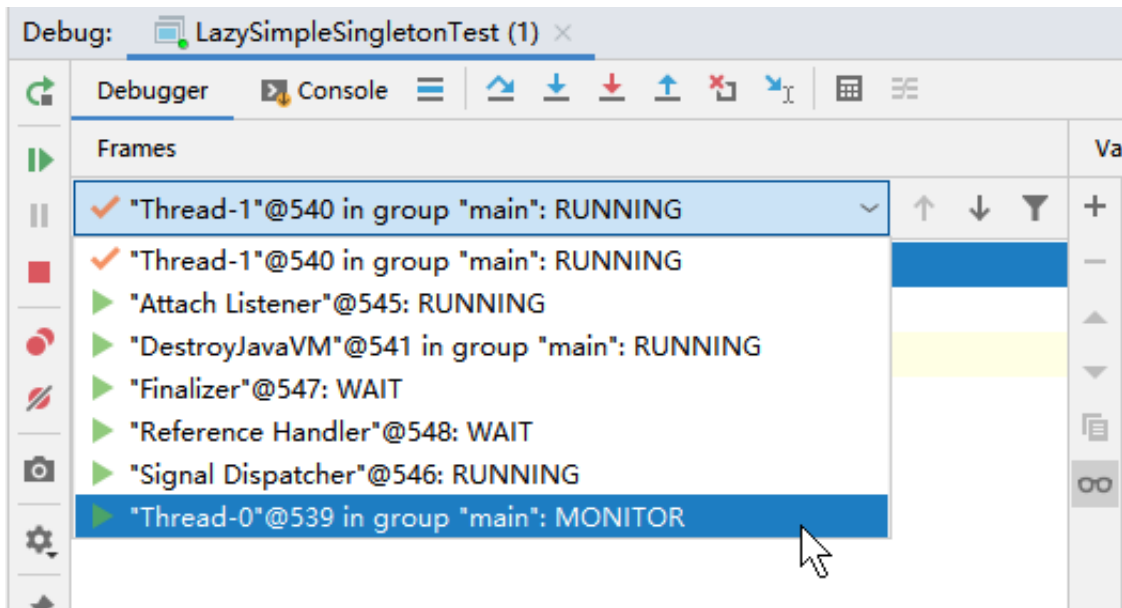
```

4  * 懒汉式单例模式(线程安全)
5  */
6  public class LazySimpleSingleton {
7      private static LazySimpleSingleton instance;
8
9      private LazySimpleSingleton() {
10     }
11
12     public static synchronized LazySimpleSingleton getInstance() {
13         if (instance == null) {
14             instance = new LazySimpleSingleton();
15         }
16         return instance;
17     }
18 }
19 //////////////////////////////////////
20 package cn.sitedev.lazy.threadsafe;
21
22 public class ExecutorThread implements Runnable {
23     @Override
24     public void run() {
25         LazySimpleSingleton singleton = LazySimpleSingleton.getInstance();
26         System.out.println(Thread.currentThread().getName() + ":" + singleton);
27     }
28 }
29 //////////////////////////////////////
30 package cn.sitedev.lazy.threadsafe;
31
32 public class LazySimpleSingletonTest {
33     public static void main(String[] args) {
34         Thread t1 = new Thread(new ExecutorThread());
35         Thread t2 = new Thread(new ExecutorThread());
36         t1.start();
37         t2.start();
38         System.out.println("END");
39     }
40 }

```

我们再来调试。当执行其中一个线程并调用 `getInstance()` 方法时，另一个线程在调用

getInstance()方法，线程的状态由 RUNNING 变成了 MONITOR，出现阻塞。直到第一个线程执行完，第二个线程才恢复到 RUNNING 状态继续调用 getInstance()方法，如下图所示。



上图完美地展现了 synchronized 监视锁的运行状态，线程安全的问题解决了。但是，用synchronized 加锁时，在线程数量比较多的情况下，如果 CPU 分配压力上升，则会导致大批线程阻塞，从而导致程序性能大幅下降。那么，有没有一种更好的方式，既能兼顾线程安全又能提升程序性能呢？答案是肯定的。我们来看双重检查锁的单例模式：

```
1 package cn.sitedev.lazy.doublecheck;
2
3 /**
4  * 双重检测锁单例
5  */
6 public class LazyDoubleCheckSingleton {
7     private volatile static LazyDoubleCheckSingleton instance;
8
9     private LazyDoubleCheckSingleton() {
10    }
11
12     public static LazyDoubleCheckSingleton getInstance() {
13         if (instance == null) {
14             synchronized (LazyDoubleCheckSingleton.class) {
15                 if (instance == null) {
16                     instance = new LazyDoubleCheckSingleton();
17                 }
18             }
19         }
20     }
21 }
```

```

17         // 1. 分配内存给这个对象
18         // 2. 初始化对象
19         // 3. 设置instance指向刚分配的内存地址
20         // 4. 初次访问对象
21     }
22 }
23 }
24     return instance;
25 }
26 }
27 ///////////////////////////////////////////////////
28 package cn.sitedev.lazy.doublecheck;
29
30 public class ExecutorThread implements Runnable {
31     @Override
32     public void run() {
33         LazyDoubleCheckSingleton singleton = LazyDoubleCheckSingleton.getI
34         System.out.println(Thread.currentThread().getName() + ":" + single
35     }
36 }
37 ///////////////////////////////////////////////////
38 package cn.sitedev.lazy.doublecheck;
39
40 public class LazyDoubleCheckSingletonTest {
41     public static void main(String[] args) {
42         Thread t1 = new Thread(new ExecutorThread());
43         Thread t2 = new Thread(new ExecutorThread());
44         t1.start();
45         t2.start();
46         System.out.println("END");
47     }
48 }

```

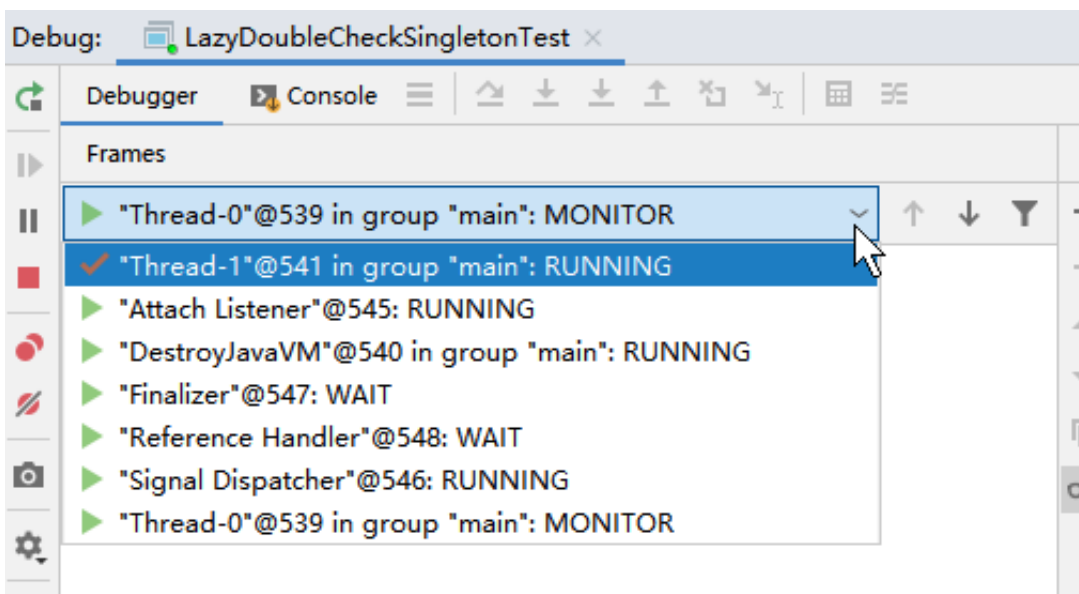
现在，我们来进行断点调试，如下图所示。

```

6 public class LazyDoubleCheckSingleton {
7     private volatile static LazyDoubleCheckSingleton instance;
8
9     private LazyDoubleCheckSingleton() {
10    }
11
12    public static LazyDoubleCheckSingleton getInstance() {
13        if (instance == null) {
14            synchronized (LazyDoubleCheckSingleton.class) {
15                if (instance == null) {
16                    instance = new LazyDoubleCheckSingleton();
17                    // 1. 分配内存给这个对象
18                    // 2. 初始化对象
19                    // 3. 设置instance指向刚分配的内存地址
20                    // 4. 初次访问对象
21                }
22            }
23        }
24        return instance;
25    }
26 }

```

当第一个线程调用 `getInstance()` 方法时，第二个线程也可以调用。当第一个线程执行到 `synchronized` 时会会上锁，第二个线程就会变成 `MONITOR` 状态，出现阻塞。此时，阻塞并不是基于整个 `LazyDoubleCheckSingleton` 类的阻塞，而是在 `getInstance()` 方法内部的阻塞，只要逻辑不太复杂，对于调用者而言感知不到。



但是，用到 `synchronized` 关键字总归要上锁，对程序性能还是存在一定影响的。难道就真的没有更好的方案吗？当然有。我们可以从类初始化的角度来考虑，看下面的代码，采用静态内部类的方式：

```

1 package cn.sitedev.lazy.staticinnerclass;
2
3 /**
4  * 这种形式兼顾饿汉式单例模式的内存浪费问题和 synchronized 的性能问题
5  * 完美地屏蔽了这两个缺点
6  */
7 public class LazyInnerClassSingleton {
8     /**
9      * 使用LazyInnerClassSingleton的时候，会先初始化内部类
10     * 如果没有使用，则内部类是不加载的
11     */
12     private LazyInnerClassSingleton() {
13     }
14
15     /**
16     * 每一个关键字都不是多余的，static是为了使单例的空间共享，final保证这个方法
17     *
18     * @return
19     */
20     public static final LazyInnerClassSingleton getInstance() {
21         // 在返回结果以前，一定会先加载内部类
22         return LazyHolder.INSTANCE;
23     }
24
25     /**
26     * 内部类，默认不加载
27     */
28     private static class LazyHolder {
29         private static final LazyInnerClassSingleton INSTANCE = new LazyIn
30     }
31 }

```

这种方式兼顾了饿汉式单例模式的内存浪费问题和 `synchronized` 的性能问题。内部类一定是要在方法调用之前初始化，巧妙地避免了线程安全问题。由于这种方式比较简单，我们就不带大家一步一步调试了

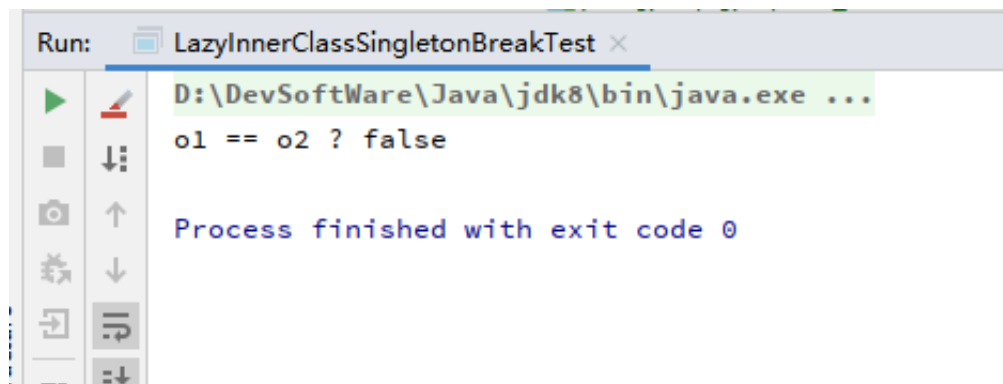
1.4. 反射破坏单例

大家有没有发现，上面介绍的单例模式的构造方法除了加上 `private` 关键字，没有做任何处理。如果我们使用反射来调用其构造方法，再调用 `getInstance()` 方法，应该有两个不同的实例。现在来看一段测试代码，以 `LazyInnerClassSingleton` 为例：

```
1 package cn.sitedev.breaksingleton.reflect;
2
3 import cn.sitedev.lazy.staticinnerclass.LazyInnerClassSingleton;
4
5 import java.lang.reflect.Constructor;
6 import java.lang.reflect.InvocationTargetException;
7
8 public class LazyInnerClassSingletonBreakTest {
9     public static void main(String[] args) {
10         try {
11             Class<?> clazz = LazyInnerClassSingleton.class;
12             testReflect(clazz);
13
14         } catch (NoSuchMethodException e) {
15             e.printStackTrace();
16         } catch (IllegalAccessException e) {
17             e.printStackTrace();
18         } catch (InstantiationException e) {
19             e.printStackTrace();
20         } catch (InvocationTargetException e) {
21             e.printStackTrace();
22         }
23     }
24
25     private static void testReflect(Class<?> clazz) throws NoSuchMethodExc
26         IllegalAccessException, InvocationTargetException {
27         // 通过反射获取私有的无参构造方法
28         Constructor constructor = clazz.getDeclaredConstructor(null);
29         // 强制访问
30         constructor.setAccessible(true);
31         // 创建实例
32         Object o1 = constructor.newInstance();
33         // 继续创建实例，相当于第二次new
34         Object o2 = constructor.newInstance();
35         System.out.println("o1 == o2 ? " + (o1 == o2));
```

```
36     }
37 }
```

运行结果如下图所示。



显然，创建了两个不同的实例。现在，我们在其构造方法中做一些限制，一旦出现多次重复创建，则直接抛出异常。来看优化后的代码：

```
1 package cn.sitedev.lazy.staticinnerclass.improved;
2
3 /**
4  * 这种形式兼顾饿汉式单例模式的内存浪费问题和 synchronized 的性能问题
5  * 完美地屏蔽了这两个缺点
6  */
7 public class LazyInnerClassSingleton {
8     /**
9      * 使用LazyInnerClassSingleton的时候，会先初始化内部类
10     * 如果没有使用，则内部类是不加载的
11     */
12     private LazyInnerClassSingleton() {
13         if (LazyHolder.INSTANCE != null) {
14             throw new RuntimeException("不允许创建多个实例");
15         }
16     }
17
18     /**
19     * 每一个关键字都不是多余的，static是为了使单例的空间共享，final保证这个方法
20     *
21     * @return
22     */
23 }
```

```

23     public static final LazyInnerClassSingleton getInstance() {
24         // 在返回结果以前，一定会先加载内部类
25         return LazyHolder.INSTANCE;
26     }
27
28     /**
29      * 内部类，默认不加载
30      */
31     private static class LazyHolder {
32         private static final LazyInnerClassSingleton INSTANCE = new LazyIn
33     }
34 }

```

再运行测试代码

```

1 package cn.sitedev.breaksingleton.reflect;
2
3 import cn.sitedev.lazy.staticinnerclass.LazyInnerClassSingleton;
4
5 import java.lang.reflect.Constructor;
6 import java.lang.reflect.InvocationTargetException;
7
8 public class LazyInnerClassSingletonBreakTest {
9     public static void main(String[] args) {
10         try {
11             Class<?> clazz = LazyInnerClassSingleton.class;
12             testReflect(clazz);
13
14             System.out.println("=====");
15             // 静态内部类式单例优化后
16             clazz = cn.sitedev.lazy.staticinnerclass.improved.LazyInnerCla
17             testReflect(clazz);
18
19
20         } catch (NoSuchMethodException e) {
21             e.printStackTrace();
22         } catch (IllegalAccessException e) {
23             e.printStackTrace();

```

```

24     } catch (InstantiationException e) {
25         e.printStackTrace();
26     } catch (InvocationTargetException e) {
27         e.printStackTrace();
28     }
29 }
30
31 private static void testReflect(Class<?> clazz) throws NoSuchMethodExc
32     IllegalAccessException, InvocationTargetException {
33     // 通过反射获取私有的无参构造方法
34     Constructor constructor = clazz.getDeclaredConstructor(null);
35     // 强制访问
36     constructor.setAccessible(true);
37     // 创建实例
38     Object o1 = constructor.newInstance();
39     // 继续创建实例，相当于第二次new
40     Object o2 = constructor.newInstance();
41     System.out.println("o1 == o2 ? " + (o1 == o2));
42 }
43 }

```

会得到如下图所示结果。

```

Run: LazyInnerClassSingletonBreakTest
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
o1 == o2 ? false
=====
java.lang.reflect.InvocationTargetException <4 internal calls>
    at cn.sitedev.breaksingleton.reflect.LazyInnerClassSingletonBreakTest.testReflect(LazyInnerClassSingletonBreakTest.java:38)
    at cn.sitedev.breaksingleton.reflect.LazyInnerClassSingletonBreakTest.main(LazyInnerClassSingletonBreakTest.java:17)
Caused by: java.lang.RuntimeException: 不允许创建多个实例
    at cn.sitedev.lazy.staticinnerclass.improved.LazyInnerClassSingleton.<init>(LazyInnerClassSingleton.java:14)
    ... 6 more
Process finished with exit code 0

```

至此，自认为史上最牛的单例模式的实现方式便大功告成。

1.5. 序列化破坏单例

一个单例对象创建好后，有时候需要将对象序列化然后写入磁盘，下次使用时再从磁盘中读取对象并进行反序列化，将其转化为内存对象。反序列化后的对象会重新分配内存，即重新创建。如果序列化的目标对象为单例对象，就违背了单例模式的初衷，相当于破坏了单例，来看一段代码：


```

1 package cn.sitedev.breaksingleton.serialize;
2
3 import java.io.Serializable;
4
5 /**
6  * 反射导致破坏单例
7  */
8 public class SerializableSingleton implements Serializable {
9     // 序列化就是把内存中的状态通过转换成字节码的形式
10    // 从而转换成一个IO流，写入其他地方(可以是磁盘，网络IO)
11    // 内存中的状态会被永久保存下来
12
13    // 反序列化就是将已经持久化的字节码内容转换为IO流
14    // 通过IO流的读取，进而将读取的内容转换成Java对象
15    // 在转换过程中会重新创建对象
16
17    private static final SerializableSingleton INSTANCE = new SerializableSingleton();
18
19    private SerializableSingleton() {
20    }
21
22    public static SerializableSingleton getInstance() {
23        return INSTANCE;
24    }
25 }

```

编写测试代码：

```

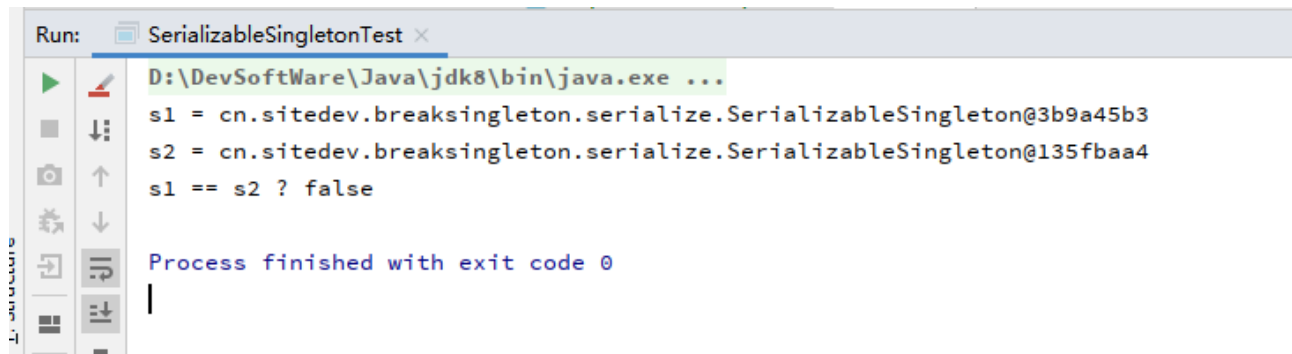
1 package cn.sitedev.breaksingleton.serialize;
2
3 import java.io.*;
4
5 public class SerializableSingletonTest {
6     public static void main(String[] args) {
7         Object s2 = SerializableSingleton.getInstance();
8         testSerialize(s2);
9     }

```

```
10
11 private static void testSerialize(Object s2) {
12     Object s1 = null;
13     ByteArrayInputStream byteArrayInputStream = null;
14     ByteArrayOutputStream byteArrayOutputStream = null;
15     ObjectOutputStream objectOutputStream = null;
16     ObjectInputStream objectInputStream = null;
17     try {
18         // 将对象s2进行序列化操作
19         byteArrayOutputStream = new ByteArrayOutputStream();
20         objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
21         objectOutputStream.writeObject(s2);
22
23         // 将对象s2进行反序列化操作，读取出来的对象赋值给s1
24         byteArrayInputStream = new ByteArrayInputStream(byteArrayOutputStream.toByteArray());
25         objectInputStream = new ObjectInputStream(byteArrayInputStream);
26         s1 = objectInputStream.readObject();
27
28         System.out.println("s1 = " + s1);
29         System.out.println("s2 = " + s2);
30         System.out.println("s1 == s2 ? " + (s1 == s2));
31
32     } catch (IOException | ClassNotFoundException e) {
33         e.printStackTrace();
34     } finally {
35         if (objectInputStream != null) {
36             try {
37                 objectInputStream.close();
38             } catch (IOException e) {
39                 e.printStackTrace();
40             }
41         }
42         if (objectOutputStream != null) {
43             try {
44                 objectOutputStream.close();
45             } catch (IOException e) {
46                 e.printStackTrace();
47             }
48         }
49     }
```

```
50     }
51 }
```

运行结果如下图所示。



```
Run: SerializableSingletonTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
s1 = cn.sitedev.breaksingleton.serialize.SerializableSingleton@3b9a45b3
s2 = cn.sitedev.breaksingleton.serialize.SerializableSingleton@135fbaa4
s1 == s2 ? false

Process finished with exit code 0
```

从运行结果可以看出，反序列化后的对象和手动创建的对象是不一致的，实例化了两次，违背了单例模式的设计初衷。那么，我们如何保证在序列化的情况下也能够实现单例模式呢？其实很简单，只需要增加 readResolve()方法即可。来看优化后的代码：

```
1 package cn.sitedev.breaksingleton.serialize.improved;
2
3 import java.io.Serializable;
4
5 public class SerializableSingleton implements Serializable {
6     // 序列化就是把内存中的状态通过转换成字节码的形式
7     // 从而转换成一个IO流，写入其他地方(可以是磁盘，网络IO)
8     // 内存中的状态会被永久保存下来
9
10    // 反序列化就是将已经持久化的字节码内容转换为IO流
11    // 通过IO流的读取，进而将读取的内容转换成Java对象
12    // 在转换过程中会重新创建对象
13
14    private static final SerializableSingleton INSTANCE = new SerializableSingleton();
15
16    private SerializableSingleton() {
17    }
18
19    public static SerializableSingleton getInstance() {
20        return INSTANCE;
21    }
22 }
```

```
22
23     private Object readResolve() {
24         return INSTANCE;
25     }
26 }
```

再运行测试类

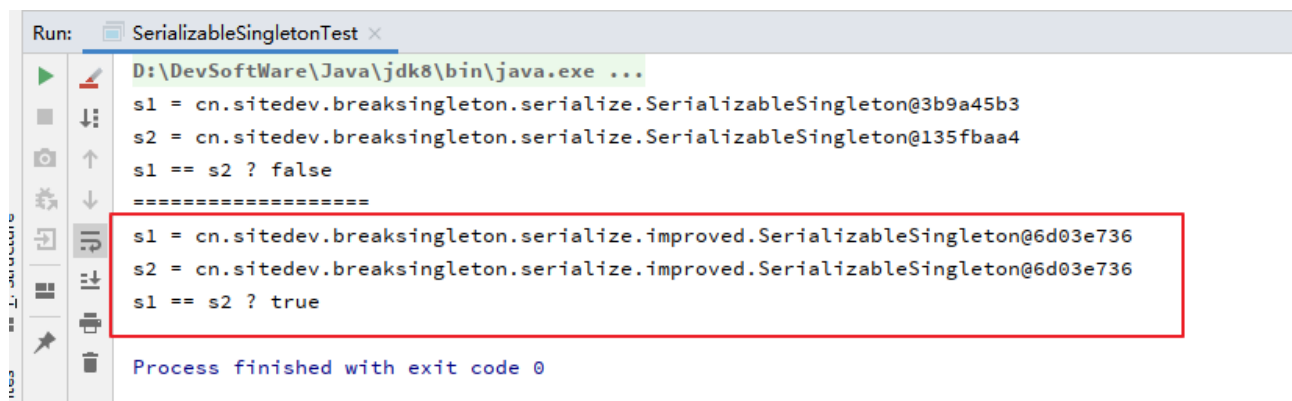
```
1 package cn.sitedev.breaksingleton.serialize;
2
3 import java.io.*;
4
5 public class SerializableSingletonTest {
6     public static void main(String[] args) {
7         Object s2 = SerializableSingleton.getInstance();
8         testSerialize(s2);
9
10        System.out.println("=====");
11        // 测试优化后的单例是否能够被反序列化
12        s2 = cn.sitedev.breaksingleton.serialize.improved.SerializableSing
13        testSerialize(s2);
14    }
15
16    private static void testSerialize(Object s2) {
17        Object s1 = null;
18        ByteArrayInputStream byteArrayInputStream = null;
19        ByteArrayOutputStream byteArrayOutputStream = null;
20        ObjectOutputStream objectOutputStream = null;
21        ObjectInputStream objectInputStream = null;
22        try {
23            // 将对象s2进行序列化操作
24            byteArrayOutputStream = new ByteArrayOutputStream();
25            objectOutputStream = new ObjectOutputStream(byteArrayOutputStr
26            objectOutputStream.writeObject(s2);
27
28            // 将对象s2进行反序列化操作，读取出来的对象赋值给s1
29            byteArrayInputStream = new ByteArrayInputStream(byteArrayOutput
30            objectInputStream = new ObjectInputStream(byteArrayInputStream
```

```

31         s1 = objectInputStream.readObject();
32
33         System.out.println("s1 = " + s1);
34         System.out.println("s2 = " + s2);
35         System.out.println("s1 == s2 ? " + (s1 == s2));
36
37     } catch (IOException | ClassNotFoundException e) {
38         e.printStackTrace();
39     } finally {
40         if (objectInputStream != null) {
41             try {
42                 objectInputStream.close();
43             } catch (IOException e) {
44                 e.printStackTrace();
45             }
46         }
47         if (objectOutputStream != null) {
48             try {
49                 objectOutputStream.close();
50             } catch (IOException e) {
51                 e.printStackTrace();
52             }
53         }
54     }
55 }
56 }

```

然后查看结果，如下图所示。



```

Run: SerializableSingletonTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
s1 = cn.sitedev.breaksingleton.serialize.SerializableSingleton@3b9a45b3
s2 = cn.sitedev.breaksingleton.serialize.SerializableSingleton@135fbaa4
s1 == s2 ? false
=====
s1 = cn.sitedev.breaksingleton.serialize.improved.SerializableSingleton@6d03e736
s2 = cn.sitedev.breaksingleton.serialize.improved.SerializableSingleton@6d03e736
s1 == s2 ? true

Process finished with exit code 0

```

大家一定会想：这是什么原因呢？为什么要这样写？看上去很神奇的样子，也让人有些费解。不如我们一起来看看 JDK 的源码实现以了解清楚。我们进入

ObjectInputStream 类的 readObject()方法，代码如下：

```
1      public final Object readObject()
2          throws IOException, ClassNotFoundException
3      {
4          if (enableOverride) {
5              return readObjectOverride();
6          }
7
8          // if nested read, passHandle contains handle of enclosing object
9          int outerHandle = passHandle;
10         try {
11             Object obj = readObject0(false);
12             handles.markDependency(outerHandle, passHandle);
13             ClassNotFoundException ex = handles.lookupException(passHandle);
14             if (ex != null) {
15                 throw ex;
16             }
17             if (depth == 0) {
18                 vlist.doCallbacks();
19             }
20             return obj;
21         } finally {
22             passHandle = outerHandle;
23             if (closed && depth == 0) {
24                 clear();
25             }
26         }
27     }
```

我们发现，在 readObject()方法中又调用了重写的 readObject0()方法。进入 readObject0()方法，代码如下：

```
1      private Object readObject0(boolean unshared) throws IOException {
2          ...
3          case TC_OBJECT:
4              return checkResolve(readOrdinaryObject(unshared));
```

我们看到 TC_OBJECT 中调用了 ObjectInputStream 的 readOrdinaryObject()方法，看源码：

```
1     private Object readOrdinaryObject(boolean unshared)
2         throws IOException
3     {
4         if (bin.readByte() != TC_OBJECT) {
5             throw new InternalError();
6         }
7
8         ObjectStreamClass desc = readClassDesc(false);
9         desc.checkDeserialize();
10
11         Class<?> cl = desc.forClass();
12         if (cl == String.class || cl == Class.class
13             || cl == ObjectStreamClass.class) {
14             throw new InvalidClassException("invalid class descriptor");
15         }
16
17         Object obj;
18         try {
19             obj = desc.isInstantiable() ? desc.newInstance() : null;
20         } catch (Exception ex) {
21             throw (IOException) new InvalidClassException(
22                 desc.forClass().getName(),
23                 "unable to create instance").initCause(ex);
24         }
25         ...
```

我们发现调用了 ObjectStreamClass 的 isInstantiable()方法，而 isInstantiable()方法的代码如下：

```
1     boolean isInstantiable() {
2         requireInitialized();
```

```
3         return (cons != null);
4     }
```

上述代码非常简单，就是判断一下构造方法是否为空，构造方法不为空就返回 true。这意味着只要有无参构造方法就会实例化。

这时候其实还没有找到加上 readResolve()方法就避免了单例模式被破坏的真正原因。再回到ObjectInputStream 的 readOrdinaryObject()方法，继续往下看：

```
1     private Object readOrdinaryObject(boolean unshared)
2         throws IOException
3     {
4         if (bin.readByte() != TC_OBJECT) {
5             throw new InternalError();
6         }
7
8         ObjectStreamClass desc = readClassDesc(false);
9         desc.checkDeserialize();
10
11         Class<?> cl = desc.forClass();
12         if (cl == String.class || cl == Class.class
13             || cl == ObjectStreamClass.class) {
14             throw new InvalidClassException("invalid class descriptor");
15         }
16
17         Object obj;
18         try {
19             obj = desc.isInstantiable() ? desc.newInstance() : null;
20         } catch (Exception ex) {
21             throw (IOException) new InvalidClassException(
22                 desc.forClass().getName(),
23                 "unable to create instance").initCause(ex);
24         }
25
26         ...
27
28         if (obj != null &&
29             handles.lookupException(passHandle) == null &&
```



```

30         desc.hasReadResolveMethod())
31     {
32         Object rep = desc.invokeReadResolve(obj);
33         if (unshared && rep.getClass().isArray()) {
34             rep = cloneArray(rep);
35         }
36         if (rep != obj) {
37             handles.setObject(passHandle, obj = rep);
38         }
39     }
40
41     return obj;
42 }

```

判断无参构造方法是否存在之后，又调用了 `hasReadResolveMethod()` 方法，来看代码：

```

1     boolean hasReadResolveMethod() {
2         requireInitialized();
3         return (readResolveMethod != null);
4     }

```

上述代码逻辑非常简单，就是判断 `readResolveMethod` 是否为空，不为空就返回 `true`。那么 `readResolveMethod` 是在哪里赋值的呢？通过全局查找知道，在私有方法 `ObjectStreamClass()` 中给 `readResolveMethod` 进行了赋值，来看代码：

```

1         readResolveMethod = getInheritableMethod(
2             cl, "readResolve", null, Object.class);

```

上面的逻辑其实就是通过反射找到一个无参的 `readResolve()` 方法，并且保存下来。现在回到 `ObjectInputStream` 的 `readOrdinaryObject()` 方法继续往下看，如果 `readResolve()` 方法存在则调用 `invokeReadResolve()` 方法，来看代码：

```

1      Object invokeReadResolve(Object obj)
2          throws IOException, UnsupportedOperationException
3      {
4          requireInitialized();
5          if (readResolveMethod != null) {
6              try {
7                  return readResolveMethod.invoke(obj, (Object[]) null);
8              } catch (InvocationTargetException ex) {
9                  Throwable th = ex.getTargetException();
10                 if (th instanceof ObjectStreamException) {
11                     throw (ObjectStreamException) th;
12                 } else {
13                     throwMiscException(th);
14                     throw new InternalError(th); // never reached
15                 }
16             } catch (IllegalAccessException ex) {
17                 // should not occur, as access checks have been suppressed
18                 throw new InternalError(ex);
19             }
20         } else {
21             throw new UnsupportedOperationException();
22         }
23     }

```

我们可以看到，在 `invokeReadResolve()` 方法中用反射调用了 `readResolveMethod` 方法。

通过 JDK 源码分析我们可以看出，虽然增加 `readResolve()` 方法返回实例解决了单例模式被破坏的问题，但是实际上实例化了两次，只不过新创建的对象没有被返回而已。如果创建对象的动作发生频率加快，就意味着内存分配开销也会随之增大，难道真的就没办法从根本上解决问题吗？下面讲的注册式单例也许能帮助你。

1.6. 注册式单例模式

注册式单例模式又称为登记式单例模式，就是将每一个实例都登记到某一个地方，使用唯一的标识获取实例。注册式单例模式有两种：一种为枚举式单例模式，另一种为容器式单例模式。

1.6.1. 枚举式单例

先来看枚举式单例模式的写法，来看代码，创建 EnumSingleton 类：

```
1 package cn.sitedev.registry.enumeration;
2
3 public enum EnumSingleton {
4     INSTANCE;
5
6     public static EnumSingleton getInstance() {
7         return INSTANCE;
8     }
9 }
```

来看测试代码：

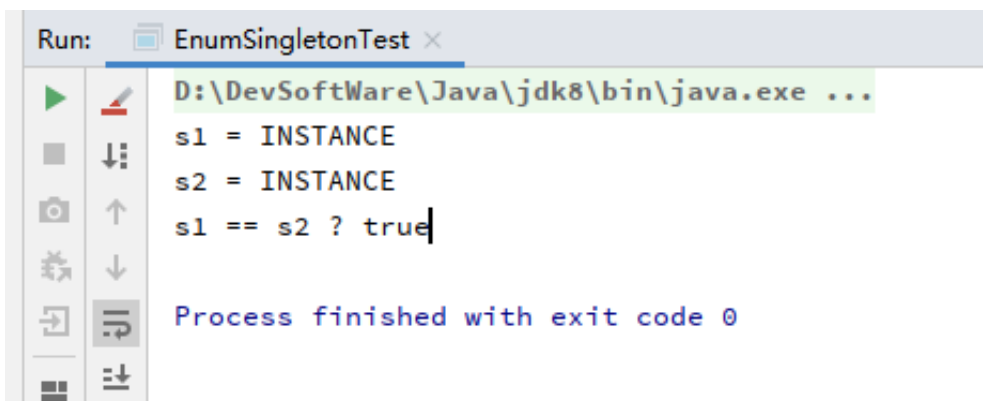
```
1 package cn.sitedev.registry.enumeration;
2
3 import java.io.*;
4
5 public class EnumSingletonTest {
6     public static void main(String[] args) {
7         Object s1 = null;
8         Object s2 = EnumSingleton.getInstance();
9
10        ByteArrayOutputStream byteArrayOutputStream = null;
11        ByteArrayInputStream byteArrayInputStream = null;
12        ObjectOutputStream objectOutputStream = null;
13        ObjectInputStream objectInputStream = null;
14        try {
15            // 将对象s2进行序列化，即将对象转换为byte[]
16            byteArrayOutputStream = new ByteArrayOutputStream();
17            objectOutputStream = new ObjectOutputStream(byteArrayOutputStream);
18            objectOutputStream.writeObject(s2);
19
20            // 将对象s2进行反序列化，即将byte[] 转换成对象，并将该对象的引用赋
21            byteArrayInputStream = new ByteArrayInputStream(byteArrayOutput
```

```

22         objectInputStream = new ObjectInputStream(byteArrayInputStrearn
23         s1 = objectInputStream.readObject();
24
25         System.out.println("s1 = " + s1);
26         System.out.println("s2 = " + s2);
27         System.out.println("s1 == s2 ? " + (s1 == s2));
28     } catch (IOException | ClassNotFoundException e) {
29         e.printStackTrace();
30     } finally {
31         if (objectInputStream != null) {
32             try {
33                 objectInputStream.close();
34             } catch (IOException e) {
35                 e.printStackTrace();
36             }
37         }
38         if (objectOutputStream != null) {
39             try {
40                 objectOutputStream.close();
41             } catch (IOException e) {
42                 e.printStackTrace();
43             }
44         }
45     }
46 }
47 }

```

运行结果如下图所示。



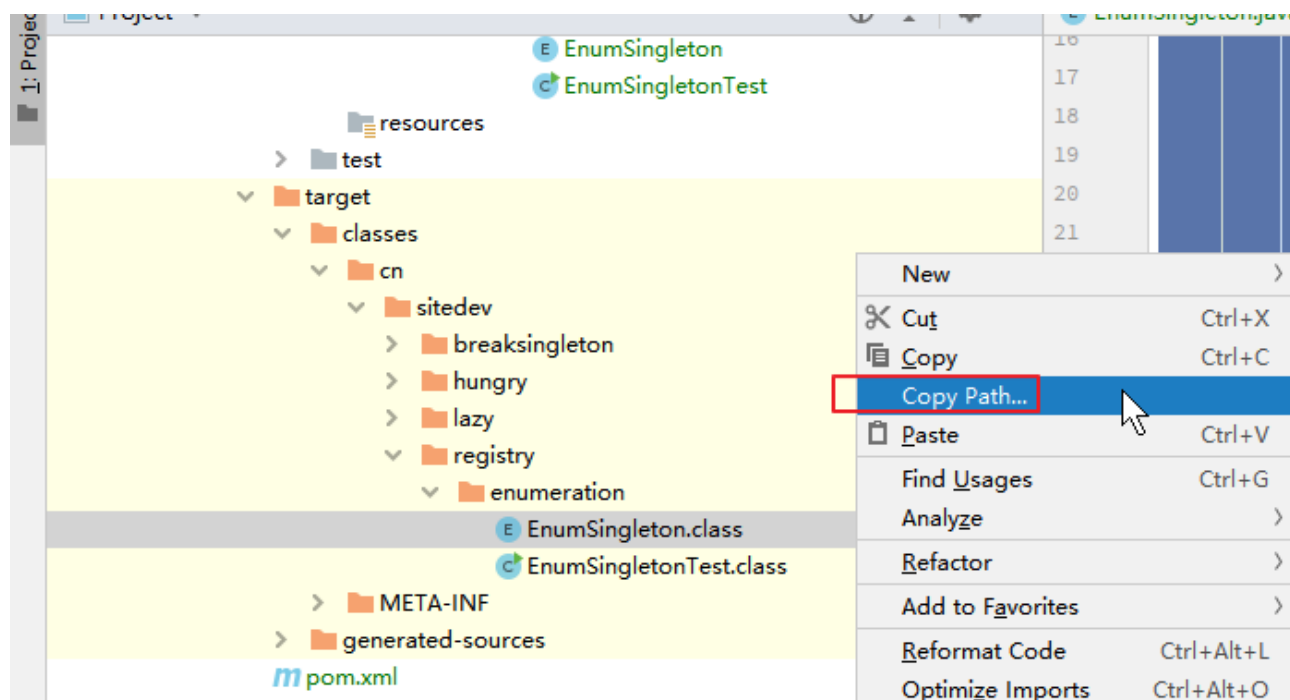
```

Run: EnumSingletonTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
s1 = INSTANCE
s2 = INSTANCE
s1 == s2 ? true
Process finished with exit code 0

```

没有做任何处理，我们发现运行结果和预期的一样。那么枚举式单例模式如此神奇，它的神秘之处在哪里体现呢？下面通过分析源码来揭开它的神秘面纱。

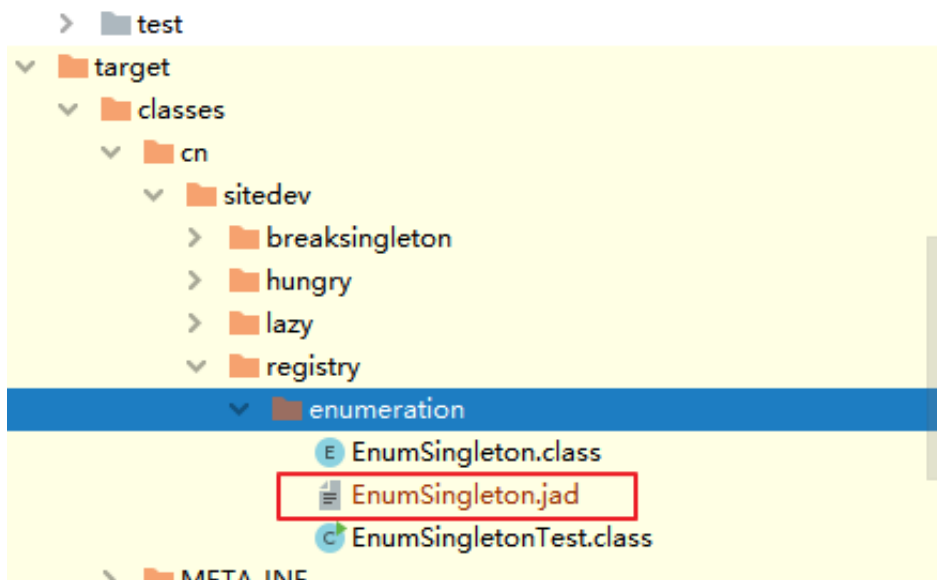
下载一个非常好用的 Java 反编译工具 Jad（下载地址：<https://varaneckas.com/jad/>，解压后配置好环境变量（这里不做详细介绍），就可以使用命令行调用了。找到工程所在的 Class 目录，复制EnumSingleton.class 所在的路径，如下图所示。



然后切换到命令行，切换到工程所在的 Class 目录，输入命令 jad 并在后面输入复制好的路径，在Class 目录下会多出一个 EnumSingleton.jad 文件。

```
C:\Windows\system32\cmd.exe

C:\Users\qchen>cd D:\DevSoftWare\workspace\IdeaProjects\java_architect_lesson_2020\chap01_Architect_Inner_Power_Skills\chap01_02_Design_Patterns\chap01_02_02_Singleton_Pattern\singleton-pattern-lesson\target\classes\cn\sitedev\registry\enumeration\
C:\Users\qchen>D:
D:\DevSoftWare\workspace\IdeaProjects\java_architect_lesson_2020\chap01_Architect_Inner_Power_Skills\chap01_02_Design_Patterns\chap01_02_02_Singleton_Pattern\singleton-pattern-lesson\target\classes\cn\sitedev\registry\enumeration>jad D:\DevSoftWare\workspace\IdeaProjects\java_architect_lesson_2020\chap01_Architect_Inner_Power_Skills\chap01_02_Design_Patterns\chap01_02_02_Singleton_Pattern\singleton-pattern-lesson\target\classes\cn\sitedev\registry\enumeration\EnumSingleton.class... Generating EnumSingleton.jad
D:\DevSoftWare\workspace\IdeaProjects\java_architect_lesson_2020\chap01_Architect_Inner_Power_Skills\chap01_02_Design_Patterns\chap01_02_02_Singleton_Pattern\singleton-pattern-lesson\target\classes\cn\sitedev\registry\enumeration>
```



打开 EnumSingleton.jad 文件我们惊奇地发现如下代码：

```
1 // Decompiled by Jad v1.5.8g. Copyright 2001 Pavel Kouznetsov.
2 // Jad home page: http://www.kpdus.com/jad.html
3 // Decompiler options: packimports(3)
4 // Source File Name:   EnumSingleton.java
5
6 package cn.sitedev.registry.enumeration;
7
8
9 public final class EnumSingleton extends Enum
10 {
11
12     public static EnumSingleton[] values()
13     {
14         return (EnumSingleton[])$VALUES.clone();
15     }
16
17     public static EnumSingleton valueOf(String name)
18     {
19         return (EnumSingleton)Enum.valueOf(cn/sitedev/registry/enumeration
20     }
21
22     private EnumSingleton(String s, int i)
23     {
24         super(s, i);
25     }
```

```

26
27     public static EnumSingleton getInstance()
28     {
29         return INSTANCE;
30     }
31
32     public static final EnumSingleton INSTANCE;
33     private static final EnumSingleton $VALUES[];
34
35     static
36     {
37         INSTANCE = new EnumSingleton("INSTANCE", 0);
38         $VALUES = (new EnumSingleton[] {
39             INSTANCE
40         });
41     }
42 }

```

原来，枚举式单例模式在静态代码块中就给 INSTANCE 进行了赋值，是饿汉式单例模式的实现。至此，我们还可以试想，序列化能否破坏枚举式单例模式呢？不妨再来看一下 JDK 源码，还是回到 ObjectInputStream 的 readObject0() 方法：

```

1     private Object readObject0(boolean unshared) throws IOException {
2         ...
3         case TC_ENUM:
4             return checkResolve(readEnum(unshared));
5         ...

```

我们看到，在 readObject0() 中调用了 readEnum() 方法，来看 readEnum() 方法的代码实现：

```

1     private Enum<?> readEnum(boolean unshared) throws IOException {
2         if (bin.readByte() != TC_ENUM) {
3             throw new InternalError();
4         }

```

```

5
6     ObjectStreamClass desc = readClassDesc(false);
7     if (!desc.isEnum()) {
8         throw new InvalidClassException("non-enum class: " + desc);
9     }
10
11     int enumHandle = handles.assign(unshared ? unsharedMarker : null);
12     ClassNotFoundException resolveEx = desc.getResolveException();
13     if (resolveEx != null) {
14         handles.markException(enumHandle, resolveEx);
15     }
16
17     String name = readString(false);
18     Enum<?> result = null;
19     Class<?> cl = desc.forClass();
20     if (cl != null) {
21         try {
22             @SuppressWarnings("unchecked")
23             Enum<?> en = Enum.valueOf((Class)cl, name);
24             result = en;
25         } catch (IllegalArgumentException ex) {
26             throw (IOException) new InvalidObjectException(
27                 "enum constant " + name + " does not exist in " +
28                 cl).initCause(ex);
29         }
30         if (!unshared) {
31             handles.setObject(enumHandle, result);
32         }
33     }
34
35     handles.finish(enumHandle);
36     passHandle = enumHandle;
37     return result;
38 }

```

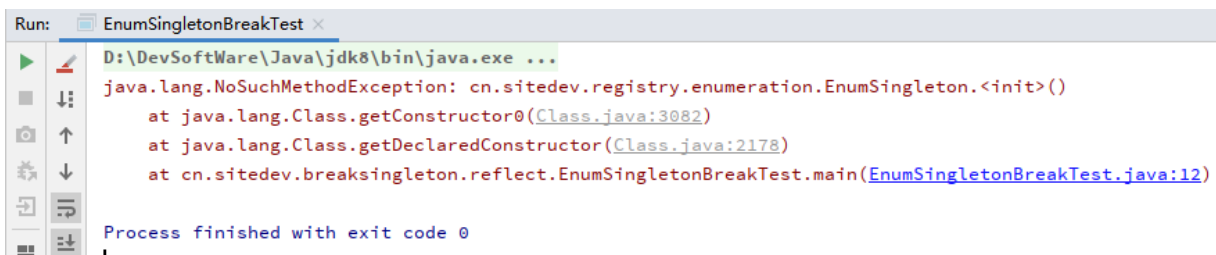
我们发现，枚举类型其实通过类名和类对象类找到一个唯一的枚举对象。因此，枚举对象不可能被类加载器加载多次。那么反射是否能破坏枚举式单例模式呢？来看一段测试代码：


```

1 package cn.sitedev.breaksingleton.reflect;
2
3 import cn.sitedev.registry.enumeration.EnumSingleton;
4
5 import java.lang.reflect.Constructor;
6 import java.lang.reflect.InvocationTargetException;
7
8 public class EnumSingletonBreakTest {
9     public static void main(String[] args) {
10         try {
11             Class clazz = EnumSingleton.class;
12             Constructor constructor = clazz.getDeclaredConstructor(null);
13             constructor.newInstance();
14         } catch (InstantiationException e) {
15             e.printStackTrace();
16         } catch (InvocationTargetException e) {
17             e.printStackTrace();
18         } catch (NoSuchMethodException e) {
19             e.printStackTrace();
20         } catch (IllegalAccessException e) {
21             e.printStackTrace();
22         }
23     }
24 }

```

运行结果如下图所示。



```

Run: EnumSingletonBreakTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
java.lang.NoSuchMethodException: cn.sitedev.registry.enumeration.EnumSingleton.<init>()
    at java.lang.Class.getConstructor0(Class.java:3082)
    at java.lang.Class.getDeclaredConstructor(Class.java:2178)
    at cn.sitedev.breaksingleton.reflect.EnumSingletonBreakTest.main(EnumSingletonBreakTest.java:12)
Process finished with exit code 0

```

结果中报的是 `java.lang.NoSuchMethodException` 异常，意思是没找到无参的构造方法。这时候，我们打开 `java.lang.Enum` 的源码，查看它的构造方法，只有一个 `protected` 类型的构造方法，代码如下：

```

1     protected Enum(String name, int ordinal) {

```

```
2         this.name = name;
3         this.ordinal = ordinal;
4     }
```

我们再来做一个下面这样的测试：

```
1 package cn.sitedev.breaksingleton.reflect;
2
3 import cn.sitedev.registry.enumeration.EnumSingleton;
4
5 import java.lang.reflect.Constructor;
6 import java.lang.reflect.InvocationTargetException;
7
8 public class EnumSingletonBreakTest2 {
9     public static void main(String[] args) {
10         try {
11             Class clazz = EnumSingleton.class;
12             Constructor constructor = clazz.getDeclaredConstructor(String.class);
13             constructor.setAccessible(true);
14             constructor.newInstance("Sitedev", 10);
15         } catch (InstantiationException e) {
16             e.printStackTrace();
17         } catch (InvocationTargetException e) {
18             e.printStackTrace();
19         } catch (NoSuchMethodException e) {
20             e.printStackTrace();
21         } catch (IllegalAccessException e) {
22             e.printStackTrace();
23         }
24     }
25 }
```

运行结果如下图所示。

```
Run: EnumSingletonBreakTest2
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
Exception in thread "main" java.lang.IllegalArgumentException: Cannot reflectively create enum objects <1 internal call>
    at cn.sitedev.breaksingleton.reflect.EnumSingletonBreakTest2.main(EnumSingletonBreakTest2.java:14)
Process finished with exit code 1
```

这时错误已经非常明显了，“Cannot reflectively create enum objects”，即不能用反射来创建枚举类型。还是习惯性地想来看看 JDK 源码，进入 Constructor 的 `newInstance()`方法：

```
1 public T newInstance(Object ... initargs)
2     throws InstantiationException, IllegalAccessException,
3           IllegalArgumentException, InvocationTargetException
4 {
5     if (!override) {
6         if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {
7             Class<?> caller = Reflection.getCallerClass();
8             checkAccess(caller, clazz, null, modifiers);
9         }
10    }
11    if ((clazz.getModifiers() & Modifier.ENUM) != 0)
12        throw new IllegalArgumentException("Cannot reflectively create
13    ConstructorAccessor ca = constructorAccessor; // read volatile
14    if (ca == null) {
15        ca = acquireConstructorAccessor();
16    }
17    @SuppressWarnings("unchecked")
18    T inst = (T) ca.newInstance(initargs);
19    return inst;
20 }
```

从上述代码可以看到，在 `newInstance()`方法中做了强制性的判断，如果修饰符是 `Modifier.ENUM`枚举类型，则直接抛出异常。

到此为止，我们是不是已经非常清晰明了呢？枚举式单例模式也是《Effective Java》书中推荐的一种单例模式实现写法。JDK 枚举的语法特殊性及反射也为枚举保驾护航，让枚举式单例模式成为一种比较优雅的实现。

1.6.2. 容器式单例

接下来看注册式单例模式的另一种写法，即容器式单例模式，创建 ContainerSingleton 类：

```
1 package cn.sitedev.registry.container;
2
3 import java.util.Map;
4 import java.util.concurrent.ConcurrentHashMap;
5
6 public class ContainerSingleton {
7
8     private static Map<String, Object> ioc = new ConcurrentHashMap<>();
9
10    private ContainerSingleton() {
11    }
12
13    public static Object getBean(String className) {
14        synchronized (ioc) {
15            if (!ioc.containsKey(className)) {
16                Object object = null;
17                try {
18                    object = Class.forName(className).newInstance();
19                    ioc.put(className, object);
20                    return object;
21                } catch (IllegalAccessException e) {
22                    e.printStackTrace();
23                } catch (InstantiationException e) {
24                    e.printStackTrace();
25                } catch (ClassNotFoundException e) {
26                    e.printStackTrace();
27                }
28            } else {
29                return ioc.get(className);
30            }
31        }
32        return null;
33    }
34 }
```

容器式单例模式适用于实例非常多的情况，便于管理。但它是非线程安全的。到此，注册式单例模式介绍完毕。我们再来看看 Spring 中的容器式单例模式的实现代码：

```
1 public abstract class AbstractAutowireCapableBeanFactory extends AbstractB
2     implements AutowireCapableBeanFactory {
3     /** Cache of unfinished FactoryBean instances: FactoryBean name --> Be
4     private final Map<String, BeanWrapper> factoryBeanInstanceCache = new
5     ...
6 }
```

1.7. 线程单例实现ThreadLocal

最后赠送给大家一个彩蛋，讲讲线程单例实现 ThreadLocal。ThreadLocal 不能保证其创建的对象是全局唯一的，但是能保证在单个线程中是唯一的，天生是线程安全的。下面来看代码：

```
1 package cn.sitedev.threadlocal;
2
3 public class ThreadLocalSingleton {
4     private static final ThreadLocal<ThreadLocalSingleton> INSTANCE = new
5         @Override
6         protected ThreadLocalSingleton initialValue() {
7             return new ThreadLocalSingleton();
8         }
9     };
10
11     private ThreadLocalSingleton() {
12     }
13
14     public static ThreadLocalSingleton getInstance() {
15         return INSTANCE.get();
16     }
17 }
```

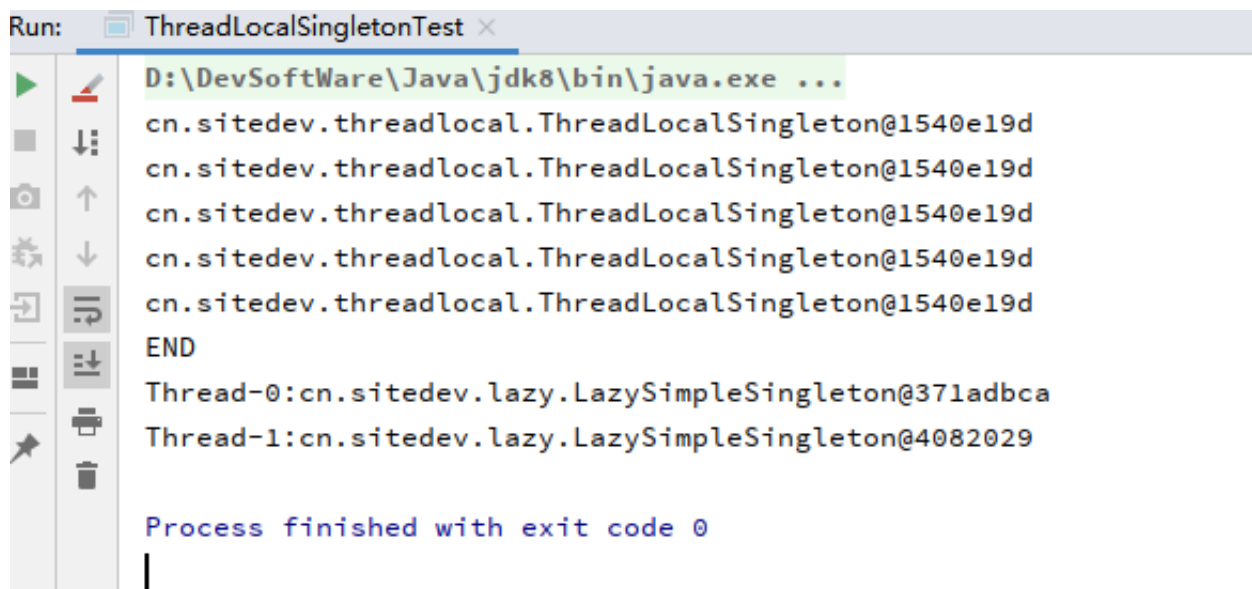
写一下测试代码：

```

1 package cn.sitedev.threadlocal;
2
3 public class ExecutorThread implements Runnable {
4     @Override
5     public void run() {
6         System.out.println(ThreadLocalSingleton.getInstance());
7     }
8 }
9 //////////////////////////////////////////////////
10 package cn.sitedev.threadlocal;
11
12 import cn.sitedev.lazy.ExecutorThread;
13
14 public class ThreadLocalSingletonTest {
15     public static void main(String[] args) {
16         System.out.println(ThreadLocalSingleton.getInstance());
17         System.out.println(ThreadLocalSingleton.getInstance());
18         System.out.println(ThreadLocalSingleton.getInstance());
19         System.out.println(ThreadLocalSingleton.getInstance());
20         System.out.println(ThreadLocalSingleton.getInstance());
21
22         Thread t1 = new Thread(new ExecutorThread());
23         Thread t2 = new Thread(new ExecutorThread());
24         t1.start();
25         t2.start();
26
27         System.out.println("END");
28     }
29 }

```

运行结果如下图所示。



```
Run: ThreadLocalSingletonTest x
D:\DevSoftWare\Java\jdk8\bin\java.exe ...
cn.sitedev.threadlocal.ThreadLocalSingleton@1540e19d
cn.sitedev.threadlocal.ThreadLocalSingleton@1540e19d
cn.sitedev.threadlocal.ThreadLocalSingleton@1540e19d
cn.sitedev.threadlocal.ThreadLocalSingleton@1540e19d
cn.sitedev.threadlocal.ThreadLocalSingleton@1540e19d
END
Thread-0:cn.sitedev.lazy.LazySimpleSingleton@371adbca
Thread-1:cn.sitedev.lazy.LazySimpleSingleton@4082029

Process finished with exit code 0
|
```

我们发现，在主线程中无论调用多少次，获取到的实例都是同一个，都在两个子线程中分别获取到了不同的实例。那么 ThreadLocal 是如何实现这样的效果的呢？我们知道，单例模式为了达到线程安全的目的，会给方法上锁，以时间换空间。ThreadLocal 将所有的对象全部放在 ThreadLocalMap 中，为每个线程都提供一个对象，实际上是以空间换时间来实现线程隔离的。

1.8. 单例模式小结

单例模式可以保证内存里只有一个实例，减少了内存的开销，还可以避免对资源的多重占用。单例模式看起来非常简单，实现起来其实也非常简单，但是在面试中却是一个高频面试题。希望“小伙伴们”通过本章的学习，对单例模式有了非常深刻的认识，在面试中彰显技术深度，提升核心竞争力，给面试加分，顺利拿到录取通知（Offer）。