单例模式

1.1、饿汉式

1.2、懒汉式

1.3、DCL懒汉式

2、静态内部类实现单例

3、反射机制破坏普通单例模式

3.1、攻击1

3.2、抵御

3.3、攻击2

3.4、反射破坏单例总结

4、枚举反射

1.1、饿汉式

public class HungryMan {

private HungryMan() {

}

private final static HungryMan hungryMan = new HungryMan();

public static HungryMan getInstance() {

return hungryMan;

}

}

1

2

3

4

5

6

7

8

9

10

缺点：浪费内存。

1.2、懒汉式

public class LazyMan {

private LazyMan() {

}

private static LazyMan lazyMan;

public static LazyMan getInstance(){

if (lazyMan == null){

lazyMan = new LazyMan();

}

return lazyMan;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

缺点：无法多线程并发。

1.3、DCL懒汉式

public class LazyMan {

private LazyMan() {

}

private static LazyMan lazyMan;

//双重检测锁的 懒汉模式

public static LazyMan getInstance() {

if (lazyMan == null) {

synchronized (LazyMan.class) {

if (lazyMan == null) {

lazyMan = new LazyMan(); }

}

}

return lazyMan;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

缺点：线程不安全。

具体原因：jvm执行代码第12行时分为三步：

分配内存空间。

执行构造方法，初始化对象。

把这个对象指向这个空间。

jvm为了优化程序性能会对程序执行序列进行重排序。

因此，上述执行过程可能变成1 3 2。这就会导致第12行lazyMan对象在未完成初始化时，不为null。

正确写法：增加volatile关键字，禁止jvm重排序。

public class LazyMan {

private LazyMan() {

}

private volatile static LazyMan lazyMan;

public static LazyMan getInstance() {

if (lazyMan == null) {

synchronized (LazyMan.class) {

if (lazyMan == null) {

lazyMan = new LazyMan();

}

}

}

return lazyMan;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

2、静态内部类实现单例

public class Holder {

private Holder() {

}

public static Holder getInstance() {

return InnerClass.holder;

}

public static class InnerClass {

private static final Holder holder = new Holder();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

3、反射机制破坏普通单例模式

3.1、攻击1

import java.lang.reflect.Constructor;

public class Singleton {

private Singleton() {

}

private static Singleton singleton;

public static Singleton getInstance() {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

return singleton;

}

public static void main(String[] args) throws Exception {

Singleton s1 = Singleton.getInstance();

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

//指示反射的对象在使用时应该取消 Java 语言访问检查,使得私有的构造函数能够被访问

constructor.setAccessible(true);

Singleton s2 = constructor.newInstance();

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

输出结果：

1163157884

1956725890

结果表明s1和s2是两个不同的实例了。

3.2、抵御

如果要抵御这种攻击，要防止构造函数被成功调用两次。需要在构造函数中对实例化次数进行统计，大于一次就抛出异常。也可以设置密钥，在构造方法中判断密钥是否一致。

public class Singleton {

private static Singleton singleton;

private static int count = 0;

private Singleton() {

synchronized (Singleton.class) {

if (count > 0) {

throw new RuntimeException("单例禁止反射");

}

count++;

}

}

public static Singleton getInstance() {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

return singleton;

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

输出结果：

Caused by: java.lang.RuntimeException: 单例禁止反射

3.3、攻击2

public static void main(String[] args) throws Exception {

Field count = Singleton.class.getDeclaredField("count");

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

constructor.setAccessible(true);

Singleton s1 = constructor.newInstance();

count.set(singleton,0);

Singleton s2 = constructor.newInstance();

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

1

2

3

4

5

6

7

8

9

10

11

12

输出结果：

1956725890

356573597

结果表明s1和s2是两个不同的实例了。

3.4、反射破坏单例总结

道高一尺，魔高一尺

​

私有化构造器并不保险，无法避免反射的恶意攻击。

因此推荐使用枚举单例模式。枚举单例有序列化和线程安全的保证，而且只要几行代码就能实现是单例最好的的实现方式

1

2

3

4、枚举反射

代码：

public enum SomeThing {  
 INSTANCE;  
 private Resource instance;  
  
 SomeThing() {  
 instance = new Resource();  
 }  
  
 public Resource getInstance() {  
 return instance;  
 }  
}

public enum Singleton {

INSTANCE;

public static Singleton getInstance(){

return INSTANCE;

}

}

1

2

3

4

5

6

7

idea反编译：

public enum Singleton {

INSTANCE;

private Singleton() {

}

public static Singleton getInstance() {

return INSTANCE;

}

}

1

2

3

4

5

6

7

8

9

10

cmd反编译：

D:\IdeaProjects\Test\out\production\Test>javap -p Singleton.class

Compiled from "Singleton.java"

public final class Singleton extends java.lang.Enum<Singleton> {

public static final Singleton INSTANCE;

private static final Singleton[] $VALUES;

public static Singleton[] values();

public static Singleton valueOf(java.lang.String);

private Singleton();

public static Singleton getInstance();

static {};

}

1

2

3

4

5

6

7

8

9

10

11

实际上，反编译的结果都错了。构造函数并不是无参数的。

最终，使用jad工具进行反编译

D:\IdeaProjects\Test\out\production\Test>jad -sjava Singleton.class

Parsing Singleton.class... Generating Singleton.java

1

2

public final class Singleton extends Enum

{

public static Singleton[] values()

{

return (Singleton[])$VALUES.clone();

}

public static Singleton valueOf(String name)

{

return (Singleton)Enum.valueOf(Singleton, name);

}

private Singleton(String s, int i)

{

super(s, i);

}

public static Singleton getInstance()

{

return INSTANCE;

}

public static final Singleton INSTANCE;

private static final Singleton $VALUES[];

static

{

INSTANCE = new Singleton("INSTANCE", 0);

$VALUES = (new Singleton[] {

INSTANCE

});

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

由第14行代码可知，实际上构造方法的参数(String s, int i)

测试

public static void main(String[] args) throws Exception {

Singleton s1 = Singleton.getInstance();

//源码构造方法参数 (String,int)

Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor(String.class,int.class);

constructor.setAccessible(true);

Singleton s2 = constructor.newInstance();

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

1

2

3

4

5

6

7

8

9

10

输出结果

java.lang.IllegalArgumentException: Cannot reflectively create enum objects

结果证明了枚举可解决线程安全问题，避免了反射，序列化问题。

————————————————

版权声明：本文为CSDN博主「G\_gBoy」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/qq\_44972847/article/details/108057553