链滴

单例模式（特点，实现方式）

作者：JellyfishMIX
原文链接：https://ld246.com/article/1598881595782
来源网站：链滴

优点

- 提供了对唯一实例的受控访问。
- 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象例模式无疑可以提高系统的性能。
- 允许可变数目的实例。

缺点

- 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。
- 单例类的职责过重，在一定程度上违背了“单一职责原则”。
- 滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾被回收，这将导致对象状态的丢失。

特点

- 构造方法私有。
- 内部对象私有。
- 提供返回对象的函数公有。

Java中单例模式的实现方式

利用私有的内部工厂类（线程安全，内部类也可以换成内部接口，不工厂类变量的作用于要改为public)

```
public class Singleton {

    private Singleton(){
        System.out.println("Singleton: " + System.nanoTime());
    }

    public static Singleton getInstance(){
        return SingletonFactory.singletonInstance;
    }

    private static class SingletonFactory{
        private static Singleton singletonInstance = new Singleton();
    }
}
```

为什么使用静态内部类实现单例模式，可以保证线程安全？

- 加载一个类时，其内部类不会同时被加载。一个类被加载，当且仅当其某个静态成员（静态域、构器、静态方法等）被调用时发生。
- 类的加载的过程是单线程执行的。它的并发安全是由JVM保证的。所以，这样写的好处是在instanc

初始化的过程中，由JVM的类加载机制保证了线程安全，而在初始化完成以后，不管后面多少次调用gtInstance方法都不会再遇到锁的问题了。

饿汉式和懒汉式

饿汉式和懒汉式的区别?

在程序启动或单件模式类被加载的时候，单件模式实例就已经被创建。

- 饿汉式：在程序启动或单件模式类被加载的时候，单件模式实例就已经被创建。
- 懒汉式：当程序第一次访问单件模式实例时才进行创建。

饿汉式（线程安全）

在程序启动或单件模式类被加载的时候，单件模式实例就已经被创建。

1. 不让外界调用构造方法创建对象，构造方法使私有化，使用 private修饰。

2. 怎么让外部获取本类的实例对象？通过本类提供一个方法，供外部调用获取实例。由于没有对象调，所以此方法为类方法，用static修饰。

3. 通过方法返回实例对象，由于类方法(静态方法)只能调用静态方法，所以存放该实例的变量改为类量，用static修饰。

4. 类变量，类方法是在类加载时初始化的，只加载一次。由于外部不能创建对象，并且实例只在类加时创建一次，饿汉式单例模式完成。

```
public class Single2 {

    private static Single2 instance = new Single2();

    private Single2(){
        System.out.println("Single2: " + System.nanoTime());
    }

    public static Single2 getInstance(){
        return instance;
    }
}
```

懒汉式（如果方法没有synchronized，则线程不安全）

```
public class Single3 {

    private static Single3 instance = null;

    private Single3(){
        System.out.println("Single3: " + System.nanoTime());
    }

    public static synchronized Single3 getInstance(){
        if(instance == null){
            instance = new Single3();
```

```
        }
        return instance;
    }
}
```

懒汉模式改良版（线程安全，使用了double-check，即check-加锁-check，的是为了减少同步的开销）

```
public class Single4 {
    // volatile关键字必须加，保证可见性
    private volatile static Single4 instance = null;

    private Single4(){
        System.out.println("Single4: " + System.nanoTime());
    }

    public static Single4 getInstance(){
        if(instance == null){
            synchronized (Single4.class) {
                if(instance == null){
                    instance = new Single4();
                }
            }
        }
        return instance;
    }
}
```

指令重排序是怎么回事?

在给instance对象初始化的过程中，jvm做了下面3件事：

1. 给instance对象分配内存
2. 调用构造函数
3. 将instance对象指向分配的内存空间

由于jvm的"优化",指令2和指令3的执行顺序是不一定的，当执行完指定3后，此时的instance对象就经不在是null的了,但此时指令2不一定已经被执行。

假设线程1和线程2同时调用getInstance()方法，此时线程1执行完指令1和指令3，线程2抢到了执行，此时instance对象是非空的。

所以线程2拿到了一个尚未初始化的instance对象，此时线程2调用这个instance就会抛出异常。

为什么volatile关键字可以保证双检锁不会出现指令重排序的问题?

- volatile关键字可以保证jvm执行的一定的“有序性”，在指令1和指令2执行完之前，指定3一定不被执行。为什么说是一定的"有序性"呢，因为对于非易失的读写，jvm仍然允许对volatile变量进行乱读写
- 保证了volatile变量被修改后立刻刷新到CPU的缓存中。

枚举类型实现单例模式

在Java引入了enum关键字以后，可以使用枚举来实现单例类：

```
public class Single5 {

    private Single5(){

    }

    /**
     * 枚举类型是线程安全的，并且只会装载一次
     */
    private enum Singleton{
        INSTANCE;

        private final Single5 instance;

        Singleton(){
            instance = new Single5();
        }

        private Single5 getInstance(){
            return instance;
        }
    }

    public static Single5 getInstance(){

        return Singleton.INSTANCE.getInstance();
    }
}
```

枚举类实现单例模式是 effective java 作者极力推荐的单例实现模式，因为枚举类型是线程安全的，且只会装载一次，设计者充分的利用了枚举的这个特性来实现单例模式，枚举的写法非常简单，而且举类型是所用单例实现中唯一一种不会被破坏的单例实现模式。

反射如何破坏单例模式

演示

一个单例类：

```
public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton() {}

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

通过反射破坏单例模式：

```
public class Test {
    public static void main(String[] args) throws Exception{
        Singleton s1 = Singleton.getInstance();

        Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        Singleton s2 = constructor.newInstance();

        System.out.println(s1.hashCode());
        System.out.println(s2.hashCode());
    }
}
```

输出结果：

```
671631440
935563443
```

结果表明s1和s2是两个不同的实例了。

分析

通过反射获得单例类的构造函数，由于该构造函数是private的，通过setAccessible(true)指示反射的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，这样使得单例模式失效。

注释

```
publicConstructor<T> getDeclaredConstructor(Class<?>... parameterTypes)
```

获取单个构造方法（能获取私有的，但要用Constructor类的 setAccessible(true) 方法设置访问权限，参数表示的是：你要获取的构造方法的构造参数个数及数据类型的class字节码文件对象。

破坏单例模式的方法及解决办法

除枚举方式外, 其他方法都会通过反射的方式破坏单例,反射是通过调用构造方法生成新的对象，所以果我们想要阻止单例破坏。

- 可以在构造方法中进行判断，若已有实例, 则阻止生成新的实例：

```
private SingletonObject1(){
    if (instance !=null){
        throw new RuntimeException("实例已经存在，请通过 getInstance()方法获取");
    }
}
```

- 如果单例类实现了序列化接口Serializable, 就可以通过反序列化破坏单例，所以我们可以不实现序化接口,如果非得实现序列化接口，可以重写反序列化方法readResolve(), 反序列化时直接返回相关单对象：

```
  public Object readResolve() throws ObjectStreamException {
      return instance;
```

```
}
```

- 防止构造函数被成功调用两次，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。

```
public class Singleton {
    private static int count = 0;

    private static Singleton instance = null;

    private Singleton(){
        synchronized (Singleton.class) {
            if(count > 0){
                throw new RuntimeException("创建了两个实例");
            }
            count++;
        }

    }

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

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

        Constructor<Singleton> constructor = Singleton.class.getDeclaredConstructor();
        constructor.setAccessible(true);
        Singleton s1 = constructor.newInstance();
        Singleton s2 = constructor.newInstance();
    }

}
```

执行结果

```
Exception in thread "main" java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
    at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
    at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
    at java.lang.reflect.Constructor.newInstance(Unknown Source)
    at com.yzz.reflect.Singleton.main(Singleton.java:33)
Caused by: java.lang.RuntimeException: 创建了两个实例
    at com.yzz.reflect.Singleton.<init>(Singleton.java:14)
    ... 5 more
```

分析

在通过反射创建第二个实例时抛出异常，防止实例化多个对象。构造函数中的synchronized是为了防多线程情况下实例化多个对象。

引用/参考

设计模式：懒汉式和饿汉式 - 北京小辉 - CSDN

"泡泡201908061058789"的回答 - 牛客

内部类加载顺序及静态内部类单例模式 - CSDN

java中双检锁为什么要加上volatile关键字 - CSDN

反射如何破坏单例模式 - Everglow的博客