在所有的设计模式中，单例模式是我们在项目开发中最为常见的设计模式之一，而单例模式有很多种实现方式，你是否都了解呢？高并发下如何保证单例模式的线程安全性呢？如何保证序列化后的单例对象在反序列化后任然是单例的呢？这些问题在看了本文之后都会一一的告诉你答案，赶快来阅读吧！

什么是单例模式?

在文章开始之前我们还是有必要介绍一下什么是单例模式。单例模式是为确保一个类只有一个实例，并为整个系统提供一个全局访问点的一种模式方法。

从概念中体现出了单例的一些特点：

1. 在任何情况下，单例类永远只有一个实例存在
2. 单例需要有能力为整个系统提供这一唯一实例

为了便于读者更好的理解这些概念，下面给出这么一段内容叙述：

在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例。这些应用都或多或少具有资源管理器的功能。每台计算机可以有若干个打印机，但只能有一个Printer Spooler，以避免两个打印作业同时输出到打印机中。每台计算机可以有若干通信端口，系统应当集中管理这些通信端口，以避免一个通信端口同时被两个请求同时调用。总之，选择单例模式就是为了避免不一致状态，避免政出多头。

正是由于这个特点，单例对象通常作为程序中的存放配置信息的载体，因为它能保证其他对象读到一致的信息。例如在某个服务器程序中，该服务器的配置信息可能存放在数据库或文件中，这些配置数据由某个单例对象统一读取，服务进程中的其他对象如果要获取这些配置信息，只需访问该单例对象即可。这种方式极大地简化了在复杂环境 下，尤其是多线程环境下的配置管理，但是随着应用场景的不同，也可能带来一些同步问题。

各式各样的单例实现

1、饿汉式单例

饿汉式单例是指在方法调用前，实例就已经创建好了。下面是实现代码：

```
package org.mlinge.s01;
public class MySingleton {
    private static MySingleton instance = new MySingleton();
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        return instance;
    }
}
```

以上是单例的饿汉式实现，我们来看看饿汉式在多线程下的执行情况，给出一段多线程的执行代码：

```
package org.mlinge.s01;
public class MyThread extends Thread{
    @Override
        public void run() {
        System.out.println(MySingleton.getInstance().hashCode());
    }
    public static void main(String[] args) {
```

```
        MyThread[] mts = new MyThread[10];
        for (int i = 0 ; i < mts.length ; i++){
            mts[i] = new MyThread();
        }
        for (int j = 0; j < mts.length; j++) {
            mts[j].start();
        }
    }
}
```

以上代码运行结果：

```
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
```

从运行结果可以看出实例变量额hashCode值一致，这说明对象是同一个，饿汉式单例实现了。

2、懒汉式单例

懒汉式单例是指在方法调用获取实例时才创建实例，因为相对饿汉式显得“不急迫”，所以被叫做“懒汉模式”。下面是实现代码：

```
package org.mlinge.s02;
public class MySingleton {
    private static MySingleton instance = null;
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        if(instance == null){
            //懒汉式
            instance = new MySingleton();
        }
        return instance;
    }
}
```

这里实现了懒汉式的单例，但是熟悉多线程并发编程的朋友应该可以看出，在多线程并发下这样的实现是无法保证实例实例唯一的，甚至可以说这样的失效是完全错误的，下面我们就来看一下多线程并发下的执行情况，这里为了看到效果，我们对上面的代码做一小点修改：

```
package org.mlinge.s02;
public class MySingleton {
    private static MySingleton instance = null;
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        try {
```

```
            if(instance != null){
                //懒汉式
            } else{
                //创建实例之前可能会有一些准备性的耗时工作
                Thread.sleep(300);
                instance = new MySingleton();
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        return instance;
    }
}
```

这里假设在创建实例前有一些准备性的耗时工作要处理，多线程调用：

```
package org.mlinge.s02;
public class MyThread extends Thread{
    @Override
        public void run() {
        System.out.println(MySingleton.getInstance().hashCode());
    }
    public static void main(String[] args) {
        MyThread[] mts = new MyThread[10];
        for (int i = 0 ; i < mts.length ; i++){
            mts[i] = new MyThread();
        }
        for (int j = 0; j < mts.length; j++) {
            mts[j].start();
        }
    }
}
```

执行结果如下：

```
1210420568
1210420568
1935123450
1718900954
1481297610
1863264879
369539795
1210420568
1210420568
602269801
```

从这里执行结果可以看出，单例的线程安全性并没有得到保证，那要怎么解决呢？

3、线程安全的懒汉式单例

要保证线程安全，我们就得需要使用同步锁机制，下面就来看看我们如何一步步的解决 存在线程安全问题的懒汉式单例（错误的单例）。

（1）、 方法中声明synchronized关键字

出现非线程安全问题，是由于多个线程可以同时进入getInstance()方法，那么只需要对该方法进行synchronized的锁同步即可：

```
package org.mlinge.s03;
public class MySingleton {
    private static MySingleton instance = null;
    private MySingleton(){
    }
    public synchronized static MySingleton getInstance() {
        try {
            if(instance != null){
                //懒汉式
            } else{
                //创建实例之前可能会有一些准备性的耗时工作
                Thread.sleep(300);
                instance = new MySingleton();
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        return instance;
    }
}
```

此时任然使用前面验证多线程下执行情况的MyThread类来进行验证，将其放入到org.mlinge.s03包下运行，执行结果如下：

```
1689058373
1689058373
1689058373
1689058373
1689058373
1689058373
1689058373
1689058373
1689058373
1689058373
```

从执行结果上来看，问题已经解决了，但是这种实现方式的运行效率会很低。同步方法效率低，那我们考虑使用同步代码块来实现：

（2）、 同步代码块实现

```
package org.mlinge.s03;
public class MySingleton {
    private static MySingleton instance = null;
    private MySingleton(){
    }
    //public synchronized static MySingleton getInstance() {
    public static MySingleton getInstance() {
        try {
            synchronized (MySingleton.class) {
                if(instance != null){
```

```
                    //懒汉式
                } else{
                    //创建实例之前可能会有一些准备性的耗时工作
                    Thread.sleep(300);
                    instance = new MySingleton();
                }
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        return instance;
    }
}
```

这里的实现能够保证多线程并发下的线程安全性，但是这样的实现将全部的代码都被锁上了，同样的效率很低下。

（3）、 针对某些重要的代码来进行单独的同步（可能非线程安全）

针对某些重要的代码进行单独的同步，而不是全部进行同步，可以极大的提高执行效率，我们来看一下：

```
package org.mlinge.s04;
public class MySingleton {
    private static MySingleton instance = null;
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        try {
            if(instance != null){
                //懒汉式
            } else{
                //创建实例之前可能会有一些准备性的耗时工作
                Thread.sleep(300);
                synchronized (MySingleton.class) {
                    instance = new MySingleton();
                }
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        return instance;
    }
}
```

此时同样使用前面验证多线程下执行情况的MyThread类来进行验证，将其放入到org.mlinge.s04包下运行，执行结果如下：

```
1481297610
397630378
1863264879
1210420568
1935123450
369539795
590202901
1718900954
1689058373
602269801
```

从运行结果来看，这样的方法进行代码块同步，代码的运行效率是能够得到提升，但是却 **没能保住线程的安全性。**看来还得进一步考虑如何解决此问题。

（4）、 Double Check Locking 双检查锁机制（推荐）

为了达到线程安全，又能提高代码执行效率，我们这里可以采用DCL的双检查锁机制来完成，代码实现如下：

```
package org.mlinge.s05;
public class MySingleton {
    //使用volatile关键字保其可见性
    volatile private static MySingleton instance = null;
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        try {
            if(instance != null){
                //懒汉式
            } else{
                //创建实例之前可能会有一些准备性的耗时工作
                Thread.sleep(300);
                synchronized (MySingleton.class) {
                    if(instance == null){
                        //二次检查
                        instance = new MySingleton();
                    }
                }
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        return instance;
    }
}
```

将前面验证多线程下执行情况的MyThread类放入到org.mlinge.s05包下运行，执行结果如下：

```
369539795
369539795
369539795
369539795
369539795
369539795
369539795
369539795
369539795
369539795
```

从运行结果来看，该中方法保证了多线程并发下的线程安全性。

这里在声明变量时使用了volatile关键字来保证其线程间的可见性；在同步代码块中使用二次检查，以保证其不被重复实例化。集合其二者，这种实现方式既保证了其高效性，也保证了其线程安全性。

4、使用静态内置类实现单例模式

DCL解决了多线程并发下的线程安全问题，其实使用其他方式也可以达到同样的效果，代码实现如下：

```
package org.mlinge.s06;
public class MySingleton {
    //内部类
    private static class MySingletonHandler{
        private static MySingleton instance = new MySingleton();
    }
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        return MySingletonHandler.instance;
    }
}
```

以上代码就是使用静态内置类实现了单例模式，这里将前面验证多线程下执行情况的MyThread类放入到org.mlinge.s06包下运行，执行结果如下：

```
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
```

从运行结果来看，静态内部类实现的单例在多线程并发下单个实例得到了保证。

5、序列化与反序列化的单例模式实现

静态内部类虽然保证了单例在多线程并发下的线程安全性，但是在遇到序列化对象时，默认的方式运行得到的结果就是多例的。

代码实现如下：

```
package org.mlinge.s07;
import java.io.Serializable;
public class MySingleton implements Serializable {
    private static final long serialVersionUID = 1L;
    //内部类
    private static class MySingletonHandler{
        private static MySingleton instance = new MySingleton();
    }
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        return MySingletonHandler.instance;
    }
}
```

序列化与反序列化测试代码：

```
package org.mlinge.s07;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
public class SaveAndReadForSingleton {
    public static void main(String[] args) {
        MySingleton singleton = MySingleton.getInstance();
        File file = new File("MySingleton.txt");
        try {
            FileOutputStream fos = new FileOutputStream(file);
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(singleton);
            fos.close();
            oos.close();
            System.out.println(singleton.hashCode());
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        try {
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            MySingleton rSingleton = (MySingleton) ois.readObject();
            fis.close();
            ois.close();
            System.out.println(rSingleton.hashCode());
        }
        catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        catch (IOException e) {
            e.printStackTrace();
        }
```

```
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

运行以上代码，得到的结果如下：

```
865113938
1442407170
```

从结果中我们发现，序列号对象的hashCode和反序列化后得到的对象的hashCode值不一样，说明反序列化后返回的对象是重新实例化的，单例被破坏了。那怎么来解决这一问题呢？

解决办法就是在反序列化的过程中使用readResolve()方法，单例实现的代码如下：

```
package org.mlinge.s07;
import java.io.ObjectStreamException;
import java.io.Serializable;
public class MySingleton implements Serializable {
    private static final long serialVersionUID = 1L;
    //内部类
    private static class MySingletonHandler{
        private static MySingleton instance = new MySingleton();
    }
    private MySingleton(){
    }
    public static MySingleton getInstance() {
        return MySingletonHandler.instance;
    }
    //该方法在反序列化时会被调用，该方法不是接口定义的方法，有点儿约定俗成的感觉
    protected Object readResolve() throws ObjectStreamException {
        System.out.println("调用了readResolve方法！");
        return MySingletonHandler.instance;
    }
}
```

再次运行上面的测试代码，得到的结果如下：

```
865113938
调用了readResolve方法!
865113938
```

从运行结果可知，添加readResolve方法后反序列化后得到的实例和序列化前的是同一个实例，单个实例得到了保证。

6、使用static代码块实现单例

静态代码块中的代码在使用类的时候就已经执行了，所以可以应用静态代码块的这个特性的实现单例设计模式。

```
package org.mlinge.s08;
public class MySingleton{
    private static MySingleton instance = null;
    private MySingleton(){
    }
    static{
        instance = new MySingleton();
    }
    public static MySingleton getInstance() {
        return instance;
    }
}
```

测试代码如下：

```
package org.mlinge.s08;
public class MyThread extends Thread{
    @Override
        public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(MySingleton.getInstance().hashCode());
        }
    }
    public static void main(String[] args) {
        MyThread[] mts = new MyThread[3];
        for (int i = 0 ; i < mts.length ; i++){
            mts[i] = new MyThread();
        }
        for (int j = 0; j < mts.length; j++) {
            mts[j].start();
        }
    }
}
```

运行结果如下：

```
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
1718900954
```

从运行结果看，单例的线程安全性得到了保证。

7、使用枚举数据类型实现单例模式

枚举enum和静态代码块的特性相似，在使用枚举时，构造方法会被自动调用，利用这一特性也可以实现单例：

```
package org.mlinge.s09;
public enum EnumFactory{
    singletonFactory;
    private MySingleton instance;
    private EnumFactory(){
        //枚举类的构造方法在类加载是被实例化
        instance = new MySingleton();
    }
    public MySingleton getInstance(){
        return instance;
    }
}
class MySingleton{
    //需要获实现单例的类，比如数据库连接Connection
    public MySingleton(){
    }
}
```

测试代码如下：

```
package org.mlinge.s09;
public class MyThread extends Thread{
    @Override
        public void run() {

System.out.println(EnumFactory.singletonFactory.getInstance().hashCode());
    }
    public static void main(String[] args) {
        MyThread[] mts = new MyThread[10];
        for (int i = 0 ; i < mts.length ; i++){
            mts[i] = new MyThread();
        }
        for (int j = 0; j < mts.length; j++) {
            mts[j].start();
        }
    }
}
```

执行后得到的结果：

```
1481297610
1481297610
1481297610
1481297610
1481297610
1481297610
1481297610
1481297610
1481297610
1481297610
```

运行结果表明单例得到了保证，但是这样写枚举类被完全暴露了，据说违反了“职责单一原则”，那我们来看看怎么进行改造呢。

8、完善使用enum枚举实现单例模式

不暴露枚举类实现细节的封装代码如下：

```
package org.mlinge.s10;
public class ClassFactory{
    private enum MyEnumSingleton{
        singletonFactory;
        private MySingleton instance;
        private MyEnumSingleton(){
            //枚举类的构造方法在类加载是被实例化
            instance = new MySingleton();
        }
        public MySingleton getInstance(){
            return instance;
        }
    }
    public static MySingleton getInstance(){
        return MyEnumSingleton.singletonFactory.getInstance();
    }
}
class MySingleton{
    //需要获实现单例的类，比如数据库连接Connection
    public MySingleton(){
    }
}
```

验证单例实现的代码如下：

```
package org.mlinge.s10;
public class MyThread extends Thread{
    @Override
        public void run() {
        System.out.println(ClassFactory.getInstance().hashCode());
    }
    public static void main(String[] args) {
        MyThread[] mts = new MyThread[10];
        for (int i = 0 ; i < mts.length ; i++){
            mts[i] = new MyThread();
        }
        for (int j = 0; j < mts.length; j++) {
            mts[j].start();
        }
    }
}
```

验证结果：

```
1935123450
1935123450
1935123450
1935123450
1935123450
1935123450
1935123450
1935123450
1935123450
1935123450
```

验证结果表明，完善后的单例实现更为合理。