**设计模式[单例模式]**

单例模式：饿汉式单例、懒汉式单例、枚举式单例、注册式单例

单利模式的优点：

* 在内存中只有一个实例，减少了内存开销
* 可以避免对资源的多重占用
* 设置全局访问点，严格控制访问

单利模式缺点：

* 没有接口，扩展难
* 如果要扩展单例对象，只有修改代码，没有其它方式。

线程同步关键字synchronized，使用双重检查锁的单例

饿汉式单例模式：

在类加载的时就立即初始化，并创建单利对象。

优点：没有加装任何锁，执行效率比较高，用户体验感比较好。

缺点：用与不用都会初始化，站着空间，浪费内存。

public class hungrySingleton{ //1.私有构造方法 private hungrySingleton(){} //2.创建单利对象 /\*\* private static final HungrySingleton hungrySingleton=new HungrySingleton(); \*/ //使用静态代码块机制 private static final HungrySingleton; static{ hungrySingleton=new HungrySingleton(); } //3.实现方法 public static HungrySingleton getInstance(){ return hungrySingleton; } }

懒汉式单例模式：

被外部类调用时侯内部类才会加载。

优点:不会在类加载的时候初始化，减少了内存消耗，提高了资源利用效率

缺点：多线程的时候容易造成线程不一，即线程拿到的对象都不一样。唯一解决办法是加上synchrinozed关键字，同步线程.

懒汉式双重锁检查模式：

  优点：  资源利用率高，不执行getInstance()就不被实例，可以执行该类其他静态方法

缺点

缺点    第一次加载时反应不快，由于java内存模型一些原因偶尔失败

/\*\*懒汉式单例（双重检查锁） public class LazyDoubleCheckSingleton{ //1.私有构造方法 private LazyDoubleCheckSingleton(){}; private static LazyDoubleCheckSingleton lazy=null; //2.实现方法 public static LazyDoubleCheckSingleton getInstance(){ if(lazy == null){ synchronized(LazyDoubleCheckSingleton.class){ if(lazy == null){ } } } return lazy; } }

线程类：ExectorThread public class ExectorThread implements Runnable{ @Override publc void run(){ LazyDoubleCheckSingleton singleton =LazyDoubleCheckSingleton.getInstance(); System.out.println(Thread.currentThread().getName() + ":" + singleton); } }

破坏单例的方式：

反射破坏单例

//通过发射机制，暴力访问私有构造方法 public class LazyInnerClassSingletonTest { public static void main(String[] args) { try { //调用者装逼，不走寻常路，破坏了单例 Class<?> clazz= LazyInnerClassSingleton.class; //获取默认的构造方法 Constructor c=clazz.getDeclaredConstructor(null); c.setAccessible(true); //暴力强吻 Object o1= c.newInstance(); Object o2=LazyInnerClassSingleton.getInstance();//使用单例再创建一个 System.out.println(o1==o2); } catch (Exception e) { e.printStackTrace(); } } }

针对线程反射破坏单例的解决方案：

//通过静态内部类去实现,全程没有用到synchronized //性能最优的写法 public class LazyInnerClassSingleton { //虽然构造方法私有了，但是逃不过反射的法眼 private LazyInnerClassSingleton(){ if (LazyHolder.lazy !=null){ throw new RuntimeException("不允许构建多个实例"); } }; //懒汉式单例 //LazyHolder里面的逻辑需要等到外部方法调用时才执行 //巧妙的利用了内部类的特性 //JVM底层执行逻辑，完美的避免了线程安全问题 public static final LazyInnerClassSingleton getInstance(){ return LazyHolder.lazy; } private static class LazyHolder{ private static final LazyInnerClassSingleton lazy=new LazyInnerClassSingleton(); } }

序列化破坏单例

当我们将一个单例对象创建好，有时候需要将对象序列化然后写入到磁盘，下次使用时

再从磁盘中读取到对象，反序列化转化为内存对象。反序列化后的对象会重新分配内存，

即重新创建。那如果序列化的目标的对象为单例对象，就违背了单例模式的初衷，相当

于破坏了单例。反序列化后的对象和手动创建的对象是不一致的。

序列化破坏单例的解决方案

增加readResolve()方法

注册式单例(RegisterSingleton)

注册式单例又称为登记式单例，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。注册式单例有两种写法：一种为容器缓存，一种为枚举登记。

//注册式单例 //使用枚举 public enum EnumSingleton { INSTANCE; private Object data; public Object getData() { return data; } public void setData(Object data) { this.data = data; } public static EnumSingleton getInstance(){ return INSTANCE; }; }

ThreadLocal 线程单例(ThreadLocalSingleton)

package com.vip.parrent.singleton.thread; /\*\* \* @Data:2019/5/27 21:27 \* @Author:Dream \*/ /\*\* \* 伪线程安全，在同一线程中是安全的,无论在同一线程中无论new多少个都是安全的 \*/ public class ThreadLocalSingleton { private ThreadLocalSingleton(){}; private static final ThreadLocal<ThreadLocalSingleton> threadLocalInstance= new ThreadLocal<ThreadLocalSingleton>(){ @Override protected ThreadLocalSingleton initialValue() { return new ThreadLocalSingleton(); } }; public static ThreadLocalSingleton getInstance(){ return threadLocalInstance.get(); } }

1、梳理内部类的执行逻辑，并画出时序图。