# 单例模式

## 解决的问题：

对象只实例化一次 ，且 是线程安全、懒加载、高性能 （枚举单例）

## 单例模式的分类

线程的安全性、性能、懒加载（lazy ）

#### 1、饿汉式

public class HungerySingleton {

//加载的时候就产生的实例对象

private static HungerySingleton instance=new HungerySingleton();

private HungerySingleton(){

}

//返回实例对象

public static HungerySingleton getInstance(){

return instance;

}

public static void main(String[] args) {

for (int i = 0; i < 20; i++) {

new Thread(()->{

System.out.println(HungerySingleton.getInstance());

}).start();

}

}

}

#### 线程安全性： 由于在静态块，只在类的初始化时加载一次 ，不会产生多个对象 因此是线程安全的。JVM ClassLoader

#### 缺点：没有延迟加载，一直保存在内存中，长时间不使用，影响整体性能 ，但是 稍优于懒汉模式

#### 2、懒汉式

public class HoonSingleton {

private static HoonSingleton instance=null;

private HoonSingleton(){

}

public static HoonSingleton getInstance(){

if(null==instance)

instance=new HoonSingleton();

return instance;

}

public static void main(String[] args) {

for (int i = 0; i < 20; i++) {

new Thread(()->{

System.out.println(HoonSingleton.getInstance());

}).start();

}

}

}

#### 线程安全：不能保证实例对象的唯一性 （线程A正在实例化对象的时候， 线程B 也进来实例化了 另一个对象 ，解决办法 ：使用同步块）

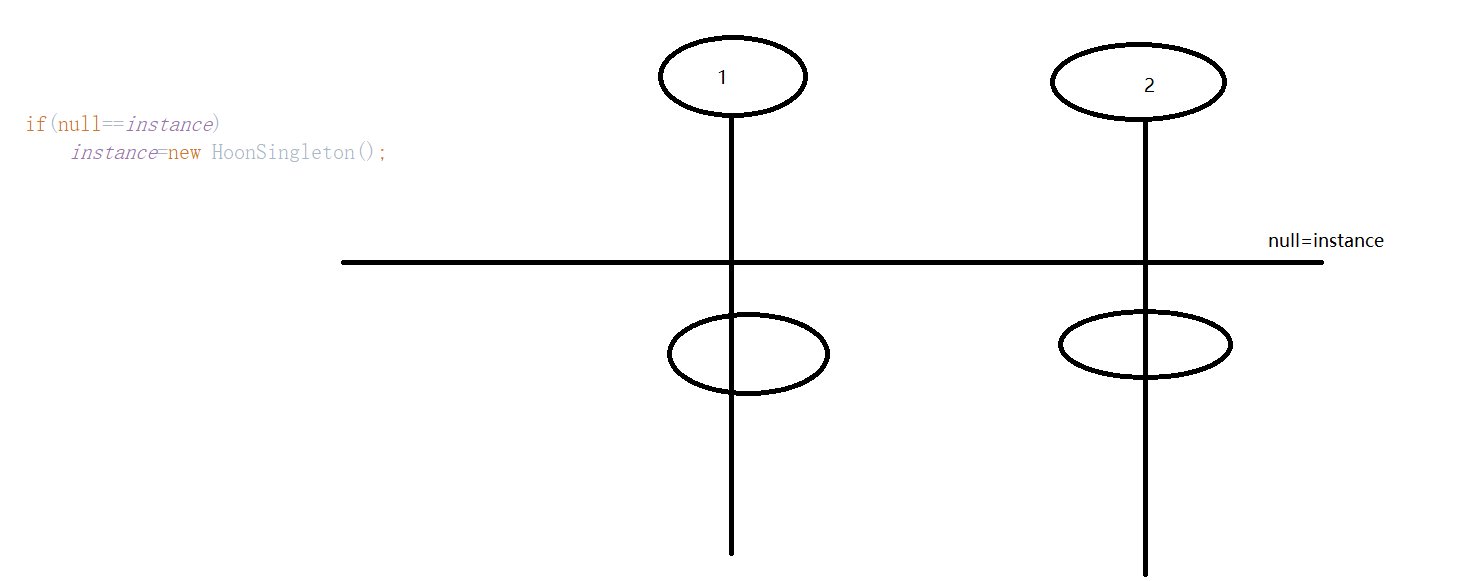
public static synchronized HoonSingleton getInstance(){

if(null==instance)

instance=new HoonSingleton();

return instance;

}



上面加synchronized 后，由于互斥性 和 原子性 就可以实现多线程下懒汉 模式

但是 性能会降低 ，因此将锁加到里面

public static HoonSingleton getInstance(){

if(null==instance)

synchronized （类.Class） {

instance=new HoonSingleton();

}

return instance;

}

如果锁后加到 上面的位置 ， 也不是线程安全的 。在多线程下 如果线程A和 线程B 同时到达 ，if null 的条件内 ， 加入A 线程执行获取到锁后 ，实例化完对象 释放锁，B再获取到锁后 也会再 实例化一个对象 ，因此出现下面的DCL（Double-Check-Locking） 双重检测 是否等于null

public static HoonSingleton getInstance(){

if(null==instance)

synchronized （类.Class） {

if(null==instance){

instance=new HoonSingleton();

}

}

return instance;

}

那么当前面的 线程释放锁 ，下面的线程获取到锁后 ， 也不会进入 if ==nul 的语句中 ，始终返回一个对象 ，另外这样些 也只有最先 到达的线程会执行synchronized 语句块， 后面再来的线程不用执行 synchronized ，直接return

这样做的目的 ，减小锁的粒度 调高性能，如果写成下面的

public static HoonSingleton getInstance(){

synchronized （类.Class） {

if(null==instance)

instance=new HoonSingleton();

}

return instance;

}

虽然是线程 安全的 ，但是每一个线程 都要执行synchronized 语句块，性能大大降低

#### 3、Double-Check-Locking

#### //Double-check-locking

#### public class DCL {

#### private static DCL instance=null;

#### private DCL(){

#### }

#### public static DCL getInstance(){

#### if(null==instance)

#### synchronized (DCL.class){

#### if(null==instance)

#### instance=new DCL();

#### }

#### return instance;

#### }

#### public static void main(String[] args) {

#### for (int i = 0; i < 20; i++) {

#### new Thread(()->{

#### System.out.println(DCL.getInstance());

#### }).start();

#### }

#### }

#### }

性能比较好

懒加载

线程的安全性

问题：因为指令重排一起空指针异常 ，原因 加入新的属性

public class DCL {

private static DCL instance=null;

Public A a; //新加入属性

Public B b; //新加入属性

Private DCL(A a,B b) {

This.a = a;

This.b = b;

Instance = new DCL();

}

}

指令重排： 由于 解释器 最终执行的代码是 是重排后的代码， 也就说 可能会出现这样的情况

Private DCL(A a,B b) {

Instance = new DCL();

This.a = a;

This.b = b;

}

从排序后的代码 在构造器中 先实例化对象再 对a、b 属性进行赋值 。假如在多线程下 ，当第一个A线程在 已经调用了Instance = new DCL() ; 但是还没完成 a、b 属性的赋值 时 ，线程 B也进行了 对象的实例化 此时发现 A 线程已经创建了对象于是就直接返回 ，那么 B线程获取到的对象属性 也是空的

#### DCL解决方案 ：Volatile+Double-check ：打破编译时代码重排 运行时指令重排，严格按照编码顺序执行

private volatile static DCL *instance*=null;

#### 4、Holder 模式（使用较多：利用静态内部类 在出次使用只实例化一次的特点，实现懒汉模式 ，并且确保线程安全）

声明类的时候，成员变量中不声明实例变量，而放到静态内部类中

public class HolderDemo {

private HolderDemo(){

}

private static class Holder{

private static HolderDemo instance=new HolderDemo();

}

public static HolderDemo getInstance(){

return Holder.instance;

}

}

当调用 HolderDemo .getInstance() 方法时 记性HolderDemod的初始化，而静态内部类的加载不需要依附外部类，在使用时才会进行类的初始化，（首次用） 所以实现了懒汉懒汉 ，由于类只初始化一次 也是线程安全的

#### 5、枚举 （最优）

#### Effectice Java

public class EnumSingletonDemo {

private EnumSingletonDemo(){

}

//延迟加载

private enum EnumHolder{

INSTANCE;

private static EnumSingletonDemo instance=null;

private EnumSingletonDemo getInstance(){

instance=new EnumSingletonDemo();

return instance;

}

}//懒加载

public static EnumSingletonDemo getInstance(){

return EnumHolder.INSTANCE.instance;

}

}