链滴

单例模式的几种用法比较

作者：huihui

原文链接：https://ld246.com/article/1505746100090

来源网站：链滴

转自## 旭日的芬芳

1.定义

确保某个类只有一个实例，能自行实例化并向整个系统提供这个实例。

2.应用场景

1. 当产生多个对象会消耗过多资源，比如IO和数据操作
2. 某种类型的对象只应该有且只有一个，比如Android中的Application。

3.考虑情况

1. 多线程造成实例不唯一。
2. 反序列化过程生成了新的实例。

4.实现方式

4.1普通单例模式

```
/**
 * 普通模式
 * @author josan_tang
 */
public class SimpleSingleton {
    //1.static单例变量
    private static SimpleSingleton instance;

    //2.私有的构造方法
    private SimpleSingleton() {

    }

    //3.静态方法为调用者提供单例对象
    public static SimpleSingleton getInstance() {
        if (instance == null) {
            instance = new SimpleSingleton();
        }
        return instance;
    }
}
```

在多线程高并发的情况下，这样写会有明显的问题，当线程A调用getInstance方法，执行到16行时检测到instance为null，于是执行17行去实例化instance，当17行没有执行完时，线程B又调用了getInstance方法，这时候检测到instance依然为空，所以线程B也会执行17行去创建一个新的实例。这时，线程A和线程B得到的instance就不是一个了，这违反了单例的定义。

4.2 饿汉单例模式

```
/**
 * 饿汉单例模式
 * @author josan_tang
 */
public class EHanSingleton {
    //static final单例对象，类加载的时候就初始化
    private static final EHanSingleton instance = new EHanSingleton();

    //私有构造方法，使得外界不能直接new
    private EHanSingleton() {
    }

    //公有静态方法，对外提供获取单例接口
    public static EHanSingleton getInstance() {
        return instance;
    }
}
```

饿汉单例模式解决了多线程并发的问题，因为在加载这个类的时候，就实例化了instance。当getInstance方法被调用时，得到的永远是类加载时初始化的对象（反序列化的情况除外）。但这也带来了另个问题，如果有大量的类都采用了饿汉单例模式，那么在类加载的阶段，会初始化很多暂时还没有用的对象，这样肯定会浪费内存，影响性能，我们还是要倾向于4.1的做法，在首次调用getInstance方时才初始化instance。请继续看4.3用法。

4.3懒汉单例模式

```
import java.io.Serializable;

/**
 * 懒汉模式
 * @author josan_tang
 */
public class LanHanSingleton {
    private static LanHanSingleton instance;

    private LanHanSingleton() {

    }

    /**
     * 增加synchronized关键字，该方法为同步方法，保证多线程单例对象唯一
     */
    public static synchronized LanHanSingleton getInstance() {
        if (instance == null) {
            instance = new LanHanSingleton();
        }
        return instance;
    }
}
```

与4.1的唯一区别在于getInstance方法前加了synchronized 关键字，让getInstance方法成为同步方，这样就保证了当getInstance第一次被调用，即instance被实例化时，别的调用不会进入到该方法

保证了多线程中单例对象的唯一性。

优点：单例对象在第一次调用才被实例化，有效节省内存，并保证了线程安全。

缺点：同步是针对方法的，以后每次调用getInstance时（就算intance已经被实例化了），也会进行步，造成了不必要的同步开销。不推荐使用这种方式。

4.4 Double CheckLock(DCL)单例模式

```
/**
 * Double CheckLock(DCL)模式
 * @author josan_tang
 *
 */
public class DCLSingleton {
    //增加volatile关键字，确保实例化instance时，编译成汇编指令的执行顺序
    private volatile static DCLSingleton instance;

    private DCLSingleton() {

    }

    public static DCLSingleton getInstance() {
        if (instance == null) {
            synchronized (DCLSingleton.class) {
                //当第一次调用getInstance方法时，即instance为空时，同步操作，保证多线程实例唯一
                //当以后调用getInstance方法时，即instance不为空时，不进入同步代码块，减少了不必
的同步开销
                if (instance == null) {
                    instance = new DCLSingleton();
                }
            }
        }
        return instance;
    }
}
```

DCL失效：

在JDK1.5之前，可能会有DCL实现的问题，上述代码中的20行，在Java里虽然是一句代码，但它并不一个真正的原子操作。

```
instance = new DCLSingleton();
```

它编译成最终的汇编指令，会有下面3个阶段：

1. 给DCLSingleton实例分配内存
2. 调用DCLSingleton的构造函数，初始化成员变量。
3. 将instance指向分配的内存空间（这个操作以后，instance才不为null）

在jdk1.5之前，上述的2、3步骤不能保证顺序，也就是说有可能是1-2-3，也有可能是1-3-2。如果是1

3-2，当线程A执行完步骤3（instance已经不为null），但是还没执行完2，线程B又调用了getInstance方法，这时候线程B所得到的就是线程A没有执行步骤2（没有执行完构造函数）的instance，线程B使用这样的instance时，就有可能会出错。这就是DCL失效。

在jdk1.5之后，可以使用volatile关键字，保证汇编指令的执行顺序，虽然会影响性能，但是和程序的确性比起来，可以忽悠不计。

Java内存模型

优点：第一次执行getInstance时instance才被实例化，节省内存；多线程情况下，基本安全；并且在nstance实例化以后，再次调用getInstance时，不会有同步消耗。

缺点：jdk1.5以下，有可能DCL失效；Java内存模型影响导致失效；jdk1.5以后，使用volatile关键字虽然能解决DCL失效问题，但是会影响部分性能。

4.5 静态内部类单例模式

```
/**
 * 静态内部类实现单例模式
 * @author josan_tang
 *
 */
public class StaticClassSingleton {
    //私有的构造方法，防止new
    private StaticClassSingleton() {

    }

    private static StaticClassSingleton getInstance() {
        return StaticClassSingletonHolder.instance;
    }

    /**
     * 静态内部类
     */
    private static class StaticClassSingletonHolder {
        //第一次加载内部类的时候，实例化单例对象
        private static final StaticClassSingleton instance = new StaticClassSingleton();
    }
}
```

第一次加载StaticClassSingleton类时，并不会实例化instance，只有第一次调用getInstance方法时Java虚拟机才会去加载StaticClassSingletonHolder类，继而实例化instance，这样延时实例化instance，节省了内存，并且也是线程安全的。这是推荐使用的一种单例模式。

4.6 枚举单例模式

```
/**
 * 枚举单例模式
 * @author josan_tang
 *
```

```
 */
public enum EnumSingleton {
    //枚举实例的创建是线程安全的，任何情况下都是单例（包括反序列化）
    INSTANCE;

    public void doSomething(){

    }
}
```

枚举不仅有字段还能有自己的方法，并且枚举实例创建是线程安全的，就算反序列化时，也不会创建的实例。除了枚举模式以外，其他实现方式，在反序列化时都会创建新的对象。

为了防止对象在反序列化时创建新的对象，需要加上如下方法：

```
    private Object readResole() throws ObjectStreamException {
        return instance;
    }
```

这是一个钩子函数，在反序列化创建对象时会调用它，我们直接返回instance就是说，不要按照默认样去创建新的对象，而是直接将instance返回。

4.7 容器单例模式

```
import java.util.HashMap;
import java.util.Map;

/**
 * 容器单例模式
 * @author josan_tang
 */
public class ContainerSingleton {
    private static Map singletonMap = new HashMap();

    //单例对象加入到集合，key要保证唯一性
    public static void putSingleton(String key, Object singleton){
        if (key != null && !"".equals(key) && singleton != null) {
            if (!singletonMap.containsKey(key)) {   //这样防止集合里有一个类的两个不同对象
                singletonMap.put(key, singleton);
            }
        }
    }

    //根据key从集合中得到单例对象
    public static Object getSingleton(String key) {
        return singletonMap.get(key);
    }
}
```

在程序初始化的时候，讲多种单例类型对象加入到一个单例集合里，统一管理。在使用时，通过key集合中获取单例对象。这种方式多见于系统中的单例，像安卓中的系统级别服务就是采用集合形式的例模式，比如常用的LayoutInfalter，我们一般在Fragment中的getView方法中使用如下代码:

```
View view = LayoutInflater.from(context).inflate(R.layout.xxx, null);
```

其实LayoutInflater.from(context)就是得到LayoutInflater实例，看下面的Android源码：

```
/**
 * Obtains the LayoutInflater from the given context.
 */
public static LayoutInflater from(Context context) {
    //通过key，得到LayoutInflater实例
    LayoutInflater LayoutInflater =
            (LayoutInflater) context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
    if (LayoutInflater == null) {
        throw new AssertionError("LayoutInflater not found.");
    }
    return LayoutInflater;
}
```

总结

不管是那种形式实现单例模式，其核心思想都是将构造方法私有化，并且通过静态方法获取一个唯一实例。在获取过程中需要保证线程安全、防止反序列化导致重新生成实例对象。选择哪种实现方式看况而定，比如是否高并发、JDK版本、单例对象的资源消耗等。

名称	优点	缺点	备注
简单模式	实现简单	线程不安全	
饿汉模式	线程安全	内存消耗太大	
懒汉模式	线程安全	同步方法消耗比较大	
DCL模式	线程安全，节省内存	jdk版本受限、高发会导致DCL失效	**推荐使用**
静态内部类模式	线程安全、节省内存	实现比较麻烦	**推荐使用**
枚举模式	线程安全、支持反序列化	个人感觉比较异	
集合模式	统一管理、节省资源	线程不安全	