转载请注明链接:https://blog.csdn.net/feather_wch/article/details/78517568

单例模式作为设计模式中非常重要的部分。本文进行详细的介绍，争取做到一篇文章吃透单例模式。

> 鸣谢: 《Head First 设计模式》

单例模式详解

版本: 2019/3/26-1(14:23)

- 单例模式详解
  - 定义
  - 有什么用?
  - 特点
  - 多线程
    - 同步方法
    - 立即实例化
    - 双重检查加锁
      - volatile
    - JVM特性实现单例
  - LEARN MORE
    - 扩展
  - 参考资料

1、什么是单例模式(Singleton Pattern)?

> 1. 只有一个类的模式

定义

2、单例模式的定义

> 确保一个类只有一个实例，并提供一个全局访问点。

有什么用?

3、单例模式有什么用?

1. 适用于对象只要一个的情况

4、单例模式的应用场景

1. 线程池(thread pool): 要方便对线程池中的线程进行控制。
2. 充当打印机、显卡等设备的驱动程序的对象: 避免多个打印任务，同时输入到打印机中。
3. 缓存(cache)
4. 对话框
5. 处理偏好设置
6. register(注册表)对象
7. 日志对象

5、单例模式在Android中的应用场景

1. EventBus: 需要确保各个组件向同一个EventBus对象进行注册。EventBus采用双重检查加锁实现的单例模式。
2. Glide: 双重检查加锁
3. 线程池

特点

6、单例模式的优点

1. 单例模式提供一个全局访问点，和全局变量一样方便，却没有全局变量的缺点。

7、全局变量的缺点

1. 需要一开始创建，假如其本身很消耗资源，却一直用不到，就会造成性能损耗。

多线程

8、单例模式在多线程中的问题

1. 单纯的单例模式在多线程的时候，可能会出现多次实例化的问题。
2. 因此有三种传统方法解决多线程问题。

同步方法

9、同步方法实现单例

1. 用synchronized修饰getInstance()方法
2. 适合性能要求不高的地方。

```
/**
 * 单纯同步：不考虑性能
 * */
public class SingletonSyn {
        private static SingletonSyn uniqueInstance;
        public static synchronized SingletonSyn getInstance() {
                if(uniqueInstance == null) {
                        uniqueInstance = new SingletonSyn();
                }
                return uniqueInstance;
        }
}
```

立即实例化

10、立即实例化实现单例

> 1. 让字段instance成为静态变量。
> 2. 这种立即创建的方法：能保证多线程同步问题。(JVM保证其多线程安全问题)
> 3. 和全局变量一个缺点，在使用不到该对象时，白白浪费性能。

```
/**
 * 立即创建：非实例化延迟，能保证多线程同步问题。
 *        JVM保证任何线程访问uniqueInstance静态变量前，一定会创建此实例。
 * */
public class SingletonImmediately {
        private static SingletonImmediately uniqueInstance
                = new SingletonImmediately();
        public static SingletonImmediately getInstance() {
                return uniqueInstance;
        }
}
```

双重检查加锁

11、双重检查加锁实现单例模式

> 1. 采用延迟实例化的方法。
> 2. 采用“双重检查”和“volatile关键字”解决多线程同步问题。
> 3. 性能更高，因为synchronized只有第一次会进行同步。

```
/**
 * 双重上锁同步(double-checked locking): 只有第一次会同步。
 *     不适用于1.4以及更老版本java
 * */
public class SingletonLockDouble {
        //volatile确保多线程正确处理uniqueInstance变量
        private static volatile SingletonLockDouble uniqueInstance;
        public static SingletonLockDouble getInstance() {
                //检查实例，不存在就进入同步块
                if(uniqueInstance == null) {
                        //上锁
                        synchronized(SingletonLockDouble.class) {
                                //不存在则创建
                                if(uniqueInstance == null) {
                                        uniqueInstance = new SingletonLockDouble();
                                }
                        }
                }
                return uniqueInstance;
        }
}
```

12、双重检查的第一重检查的作用?

> 1. 如果把第一层 `if(uniqueInstance == null)` 去掉会发现，不会影响实现多线程中的单例。
> 2. 作用在于提高性能：在一次执行时才会去执行 `synchronized` 进行同步，后续的调用都不会再同步。

13、双重检查的第二重检查的作用?

> 用于防止第二个线程获得锁后去执行 `new SingletonLockDouble()` 导致有两个对象：
>
> 1. 线程A、线程B同时去执行 `getInstance()`
> 2. 线程A获得锁，去执行 `new SingletonLockDouble()` ，此时线程B阻塞。
> 3. 线程A执行完后，释放锁。
> 4. 然后线程B获得锁，执行 `new SingletonLockDouble()` 创建了第二个实例。

14、双重检查加锁和采用同步方法实现的单例模式的性能差别?

> 1. 同步方法：每次调用都会进行synchronized同步，效率低。
> 2. 双重检查加锁：第一次执行才会利用synchronized的同步，后续都不需要同步，性能整体更高。

15、哪些场景使用了双重检查加锁?

> 1. EventBus
> 2. Glide

volatile

15、JVM在对象创建的过程中做了哪些事情？(3步)

> 1. 分配内存空间
> 2. 调用构造方法
> 3. 将对象指向分配的内存空间

16、JVM在对象创建过程中的优化

> 1. JVM会对 调用构造方法 和 将对象指向分配的内存空间 这两个指令进行重排。
> 2. 会导致两个指令执行顺序不一定，但是能保证最终结果的正确性。
> 3. 但是JVM的优化在多线程中，无法保证结果的正确性。

17、双重检查加锁会因为JVM优化导致错误吗？

> 会：
>
> 1. 线程A和线程B中同时调用 `getInstance()`
> 2. 线程A执行了“第1步-分配内存空间 第3步-将对象指向分配的内存空间”，但是还没有执行"第二步-调用构造方法"
> 3. 此时线程B开始执行，发现 对象 不为 `null` ，去用这个还未初始化的对象去进行操作，就会出错。

18、 `volatile` 关键字如何解决双重检查加锁的问题？

> 1. 保证了对象创建过程中的有序性，不会被JVM进行指令重排。
> 2. 保证了可见性，对volatile变量的修改，会立即刷新到主存中。

JVM特性实现单例

19、JVM特性实现最高效的单例模式

> 1. 一次Synchronized同步都不需要
> 2. 利用JVM在类的初始化阶段能保证多线程安全，这个特点来实现单例。
> 3. 相比于双重检查锁来说，效率进一步提升。

```
public class SingleTon {
    private SingleTon(){}
    private static class LazyHolder{
        public static SingleTon INSTANCE = new SingleTon();
    }
    public static SingleTon getInstance(){
        return LazyHolder.INSTANCE;
    }
}
```

LEARN MORE

扩展

1、单例模式产生的背景?作者是谁?

> 著名 Gang of Four-四人帮 的 23个设计模式之一

2、单例模式有哪些优点?

> 1. 适合 只能有一个实例对象的场景
> 2. 对于频繁使用的 重量级对象 ， 单例能节省可观的系统开销， 提高性能
> 3. 减少 new的操作频率 ， 减少GC压力 ， 缩短GC停顿时间

3、四人帮(GOF)是谁?

> 1-四个作者Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides
>
> 1. 1994 合著 Design Patterns: Elements of Reusable Object-Oriented Software ， 开创设计模式新纪元。

4、单例模式横向扩展：有哪些类似的事物?

> 1. 通知栏提示接收到10个手机短信，点击多个短信，最终都只有一个短信页面，点击上一步都是回到短信列表。
> 2. Activity启动模式中的 singleInstance 是一种单例模式
> 3. 点外卖，同一住处在同一时间，下多个订单。该店家只有派出一个外卖小哥来送所有的外卖，而不是一个订单一个外卖小哥。

5、 立即实例化实现单例 就像店家将所有食物都制作了出来，有一个外卖订单就直接发出去，如果没有订单，那么多货物需要 很大很大的地方 来暂存。

6、 同步方法实现单例 ，就是两个外卖小哥送一份订单，进门要过安检，只允许一个进入。保证了只会送出一份。安全无误，性能差。

7、 双重检查加锁

> 1. 线程A、线程B都想听音乐，就先后在音乐家门口排队【synchronized】。
> 2. 线程A先点, 点了一首 义勇军进行曲 ， 音乐家开始演奏。
> 3. 线程A点好后，轮到了线程B，线程B检查后发现需要的音乐已经在演奏了，就直接啥都不做。【第二重检查】
> 4. 线程C姗姗来迟，也要听 义勇军进行曲 ， 为了减少无意义的排队，直接检查到当前播放的音乐是想听的音乐，就志得意满的走了。【第一重检查】

8、JVM特性实现的单例就如同校园广播

> 线程A、B、C都要听 花香 ， 由学校广播室处理多线程问题。只会放一首歌，而不会同时放三首“花香”--交给JVM特性处理。

9、JVM创造对象就如同建造房子

> 1. 拿到政府划分的一块地皮(划分内存空间)
> 2. 根据图纸进行建造(调用构造方法)
> 3. 将门牌号指向这个地皮(对象指向这个空间)

10、JVM的指令重排序

> 1. JVM在保证单线程中结果正确的前提下，对指令进行重排序，达到优化的效果
> 2. 依旧拿到地皮
> 3. 将门牌号指向这个地皮
> 4. 根据图纸进行建造(调用构造方法)

11、JVM的指令重排序在多线程中就会导致问题

> 如同游客B需要住到旅馆，根据门牌号找到了该旅馆，却发现只有地皮还没建造，直接居住会出问题

12、Volatile如同监工

> 1. 严格保证房屋建造的顺序- 有序性
> 2. 严格保证房屋信息的改动会立即更新到政府部门系统中- 可见性

参考资料

1. 《Head First 设计模式》-单例模式
2. 单例模式的双层锁原理