Java并发笔记——单例与双重检测

单例模式可以使得一个类只有一个对象实例，能够减少频繁创建对象的时间和空间开销。单线程模式下一个典型的单例模式代码如下：

①

```
class Singleton{
    private static Singleton singleton;
    private Singleton(){}

    public static Singleton getInstance(){
        if(singleton == null){
            singleton = new Singleton();    //1
        }
        return singleton;
    }
}
```

构造器私有使得外界无法通过构造器实例化Singleton类，要取得实例只能通过getInstance()方法。这是一个延迟加载的版本，即在需要对象的时候才进行实例化操作。该方法在单线程下能够正常运行，但是在多线程环境下会出现由于没有同步措施而导致产生多个单例对象的情况。原因在于可能同时有两个线程A和B同时执行到 if 条件判断语句，A判断singleton为空准备执行//1时让出了CPU时间片，B也判断singleton为空，接着执行//1，此时创建了一个实例对象；A获取了CPU时间片后接着执行//1，也创建了实例对象，这就导致多个单例对象的情况。

解决问题的方法也很简单，使用synchronized关键字：

②

```
class Singleton{
    private static Singleton singleton;
    private Singleton(){}

    public static synchronized Singleton getInstance(){
        if(singleton == null){
            singleton = new Singleton();    //1
        }
        return singleton;
    }
}
```

这样解决了多线程并发的问题，但是却带来了效率问题：我们的目的是只创建一个实例，即//1处代码只会执行一次，也正是这个地方才需要同步，后面创建了实例之后，singleton非空就会直接返回对象引用，而不用每次都在同步代码块中进行非空验证。那么可以考虑只对//1处进行同步：

③

```
class Singleton{
    private static Singleton singleton;
    private Singleton(){}

    public static Singleton getInstance(){
        if(singleton == null){
            synchronized(Singleton.class){
                singleton = new Singleton();   //1
            }
        }
        return singleton;
    }
}
```

这样会带来与第一种一样的问题，即多个线程同时执行到条件判断语句时，会创建多个实例。问题在于当一个线程创建一个实例之后，singleton就不再为空了，但是后续的线程并没有做第二次非空检查。那么很明显，在同步代码块中应该再次做检查，也就是所谓的双重检测：

④双重检测：

```
class Singleton{
    private static Singleton singleton;
    private Singleton(){}

    public static Singleton getInstance(){
        if(singleton == null){
            synchronized(Singleton.class){
                if(singleton == null)
                    singleton = new Singleton();   //1
            }
        }
        return singleton;
    }
}
```

到这里已经很完美了，看起来没有问题。但是这种双重检测机制在JDK1.5之前是有问题的，问题还是出在//1，由所谓的无序写入造成的。一般来讲，当初始化一个对象的时候，会经历内存分配、初始化、返回对象在堆上的引用等一系列操作，这种方式产生的对象是一个完整的对象，可以正常使用。但是JAVA的无序写入可能会造成顺序的颠倒，即内存分配、返回对象引用、初始化的顺序，这种情况下对应到//1就是singleton已经不是null，而是指向了堆上的一个对象，但是该对象却还没有完成初始化动作。当后续的线程发现singleton不是null而直接使用的时候，就会出现意料之外的问题。

JDK1.5之后，可以使用volatile关键字修饰变量来解决无序写入产生的问题，因为volatile关键字的一个重要作用是禁止指令重排序，即保证不会出现内存分配、返回对象引用、初始化这样的顺序，从而使得双重检测真正发挥作用。

当然，也可以选择不使用双重检测，而采用非延迟加载的方式来达到相同的效果：

```
class Singleton{
    private static Singleton singleton = new Singleton();
    private Singleton(){}

    public static Singleton getInstance(){
        return singleton;
    }
}
```

【参考】

Java单例模式中双重检查锁的问题

单例模式与双重检测

Java 中的双重检查（Double-Check）

Java并发编程：volatile关键字解析

标签：JAVA并发编程

好文要顶 关注我 收藏该文

纳兰小依
关注 - 12
粉丝 - 2
+加关注

0 推荐 0 反对

posted @ 2017-09-07 23:59 纳兰小依 阅读(3500) 评论(0) 编辑 收藏

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

相关博文：
· 关于Singleton
· 单例模式三境界
· Java常见设计模式之单例模式
· 单例模式
· 设计模式之单例模式

最新新闻：
· Adobe再出漏洞，一个word文档就能控制电脑
· 银行客户登进入星际空间，下一步是什么？
· 打车先驱Sidecar起诉Uber：手段卑劣带来今天的垄断
· 美的集团40亿元回购已完成70% 累计回购金额达29.4亿元
· 小米设立中国区 联合创始人王川担任总裁
» 更多新闻...

< 2018年12月 >

日	一	二	三	四	五	六
25	26	27	28	29	30	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

搜索

找找看
谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

最新随笔

1. 我为什么要写博客
2. Java深拷贝与序列化
3. Java并发笔记——单例与双重检测
4. Mysql索引
5. 小解红黑树
6. Spring与Mybatis配置问题
7. Springmvc加载静态文件和开启EL表达式的支持
8. php调用java类文件
9. 在Nginx上配置ThinkPHP项目

我的标签

JAVA并发编程(1)
Java基础(1)
Mysql(1)
SpringMVC(1)
spring-mybatis(1)
SSH(1)
一些思考(1)

随笔分类

SpringMVC(1)

随笔档案

2018年6月 (2)
2017年9月 (1)
2017年8月 (1)
2017年4月 (1)
2016年8月 (2)
2015年4月 (1)
2015年2月 (1)

积分与排名

积分 - 12334
排名 - 37874

阅读排行榜

1. Springmvc加载静态文件和开启EL表达式的支持(3956)
2. Java并发笔记——单例与双重检测(3499)
3. 在Nginx上配置ThinkPHP项目(3171)
4. php调用java类文件(3025)
5. Java深拷贝与序列化(123)

推荐排行榜

1. php调用java类文件(1)
2. Springmvc加载静态文件和开启EL表达式的支持(1)