Java并行(4):线程安全前传之Singleton

1.寂寞的Singleton

如果你是一名OO程序员，Singleton的名字对你来说就不会陌生，它是GoF设计模式的一种，江湖人称“单例”的便是；即便你不是OO程序员，中国人你总该是吧？那么下面一段你应该也会背：“世界上只有一个敏感词，敏感词是敏感词的一部分，敏感词是代表敏感词的唯一合法敏感词，任何企图制造两个敏感词的企图都是注定要失败的。”说的多么好！一语道破Singleton的真谛。但是，为了让帖子存活下去，为了更好地娱乐大众，下面我们从“敏感词系统”转到“世界杯系统”，我们来看：

```
public class WorldCup {
        private static WorldCup instance = new WorldCup();

        public static WorldCup getInstance(){
                return instance;
        }
}
```

这就是一个极为简易的Singleton范例，但是雷米特同学看到这个类估计哭得心思都有了：这里的instance是eager initialization，也就是说作为世界杯的发起人，雷米特小朋友必须在提出“世界杯”这个概念的时候，就自己掏钱铸一座金杯撂那，这赛事成不成还两说。搁谁谁也不乐意。那这事咋整？雷米特老婆温柔地说，“你个完蛋败家玩意，那就等破世界杯板上钉钉，第一届举办的时候再造呗！”真是一语惊醒梦中人，雷米特立刻打开IDE，敲出下面的代码：

```
public class WorldCup {
        private static WorldCup instance;

        public static WorldCup getInstance(){
                if(instance == null) //A
                        instance = new WorldCup(); //B
                return instance;
        }
}
```

雷米特长出一口气，这回lazy initialization，总高枕无忧了吧~

2. 当Singleton遇见多线程

这时，温柔贤惠的老婆又发话了：“你傻啊？倒霉玩意你想造多少个破杯啊？杯具啊，我~不~活~了~~我错了，我从一开始就错了，如果我不嫁过来……”雷米特表示理解不能，这么NB的代码错在哪了？

观众朋友们来分析一下，是什么让雷米特的老婆如此伤心欲绝呢？绝大部分朋友应该已经知道了，那就是多线程的问题。在A和B之间存在一个时间差，可能有t1，t2两个线程，t1检测instance为null，没有继续执行B而是被切走，t2又检测到instance为null，这下，两个世界杯就被造出来了。Singleton名不副实。那应该如何？你可能已经对我的傻逼描述烦不胜烦了，加锁呗：

```
public class WorldCup {
        private static WorldCup instance;

        public static WorldCup getInstance() {
                synchronized (WorldCup.class) {
                        if (instance == null) // A
                                instance = new WorldCup();// B
                        return instance;
                }
        }
}
```

问题解决，不是吗？对，但不那么完美。我们知道，加锁/放锁是很费的操作，这里完全没有必要每次调用getInstance都加锁，事实上我们只想保证一次初始化成功而已，其余的快速返回就好了。

3. 又见DCL

那也不难，用传说中的“双检锁”（Double-Checked Lock）即可：

```
public class WorldCup {
        private static WorldCup instance;

        public static WorldCup getInstance() {
                if (instance == null) { //C
                        synchronized (WorldCup.class) {
                                if (instance == null) // A
```

```
                                        instance = new WorldCup();// B
                        }
                }
                return instance;
        }
}
```

新加的C操作过滤掉了大量的“快速返回”，让程序只有在真正需要加锁时才去加锁，效率大涨。雷米特大喜过望，终于可以和老婆交差了。但是，结束了么?

4.安全发布

“结束了么？”一段时间之前的一次电话面试中，面对同样的问题，面试官不怀好意地问。我立刻深深地觉得我被这两个臭不要脸的家伙彻底调戏了，万分纠结地败下阵来。相信雷米特再次面对他老婆时，也会有相同的感受。那么，看似精巧的DCL，会有什么问题呢？我们要从“安全发布”谈起。

所谓对象的“发布”（publish），是指使他能够让当前范围以外的代码使用。为了方便理解，列举几种发布对象的常用方法:

- 把对象的引用存到公共静态域里。
- 把一个对象传递给一个“外部方法”，所谓外部方法是指其他类的方法或者自身可以被子类重写的方法（因为你不知道这些方法会有些什么动作）
- 发布一个Inner Class的实例。这是因为每个Inner Class实例都保存了外部对象的引用。

另外需要记住的规则是“当发布一个对象时，实际上隐式地发布了他的所有非私有域对象”。

发布对象并不可怕，可怕的是错误地发布对象。当一个对象还没有做好准备时就将他发布，我们称作“逃逸”（escape）。举一个简单的例子:

```
public class Argentina {
        private boolean isMessiReady;

        public Argentina() {
                new Thread(new Runnable() {
```

```
                        @Override
                        public void run() {
                                tell();  //Argentina.this
                        }
                }).start();
                isMessiReady = true;
        }

        void tell() {
                System.out.println("Is Messi Here?:" + isMessiReady);
        }
}
```

阿根廷队队伍还没组建好就开新闻发布会，这时Messi到底在不在呢？老马可能在放烟雾弹。这里的对象发布属于我们上面提到的第三种，即发布内部类，因为这里的Thread其实是用一个匿名内部类Runnable实现的，新线程可以访问到外部类，在我们加注释的那一行其实隐含的访问了外部类Argentina.this。这属于臭名昭著的“this逃逸”，常见情景包括了“在构造函数中添加listener，启动新线程或者调用可重写方法”。

其实说白了，所谓逃逸，无非是对象“发布”和另一个线程访问该对象之间没有正确的Happens-before关系。

回过头来看我们上面的DCL，他虽然不是“this逃逸”，但也属于肇事逃逸的一种。一个线程t1的B操作和另一线程t2的C操作之间没有HB关系，也就是对instance的读写没有同步，可能会造成的现象是t1-B的new WorldCup()还没有完全构造成功，但t2-C已经看到instance非空，这样t2就直接返回了未完全构造的instance的引用，t2想当然地对instance进行操作，结果是微妙的。

看到这里，结合上一次的讨论，可能你已经明白了，“说了这么久，原来还不是劳什子的可见性问题，翠花，上volatile～”

```
public class WorldCup {
        private volatile static WorldCup instance;

        public static WorldCup getInstance() {
                if (instance == null) { //C
                        synchronized (WorldCup.class) {
                                if (instance == null) // A
                                        instance = new WorldCup();// B
                        }
```

```
                }
                return instance;
        }
}
```

“这下，你，们，该，满，足，了，吧？”

5. Yet another 解决方法

恭喜你，成功了。但是，其实我还想说，恭喜你，你out了。随着时代的发展JVM的进步，DCL这样的技巧已经逐步被淘汰了，而lazy initialization holder这样的新秀在效率上和DCL已经没什么差别：

```
public class WorldCup {
        private static class WorldCupHolder{
                public static WorldCup instance = new WorldCup();
        }

        public static WorldCup getInstance() {
                return WorldCupHolder.instance;
        }
}
```

同样是“惰性初始化”，这个是不是更好看？

在这里我们回过头来看看我们最初的eager initialization，你这时可能会反过来思考，他不是volatile，会不会有escape问题？不会。因为Java保证了域初始化段落对其余操作的HB关系。好了，这下，雷米特家的河东狮估计可以休矣。

6. 讨论的延续

关于上面阿根廷的例子，写的时候我发现一点疑问，把自己的理解拿出来和大家讨论。那就是如果我把isMessiReady = true（记做A）放在新线程start（记做B）的前面，在这里新线程不就可以保证HB关系了么？因为有IntraThread原则，A hb B，而我们又有start操作hb于任何该线程的动作，比如tell（记做C），那么不就有A hb B hb C了么？可以保证新闻发布会上梅西肯定在场。那么，为什么几乎所有看到的资料里都警告说“即使是构造函数的最后一个操作，也不要启新线程、加listener、调可重写函数”云云？我的

理解是，是为了防止这种情况：

```
class SuperArgentina extends Argentina{
        private boolean isDiegoReady;
        public SuperArgentina() {
                super();
                isDiegoReady =true;
        }

        @Override
        protected void tell() {
                super.tell();
                System.out.println("Is Diego Here?:"+isDiegoReady);
        }
}
```

我们拓展阿根廷为“超级阿根廷”，加入老马的状态，这样一来，老马自己在不在场就成问题了。关于上面的这段分析，我的把握并不是特别大，希望大牛们能够提点一下。

主要参考资料：

1. JavaWorld文章： Double-checked locking: Clever, but broken

2. Java Concurrency in Practice

3. GoF设计模式