



32 | Balking模式：再谈线程安全的单例模式

2019-05-11 王宝令

Java并发编程实战

[进入课程 >](#)



讲述：王宝令

时长 07:06 大小 6.51M

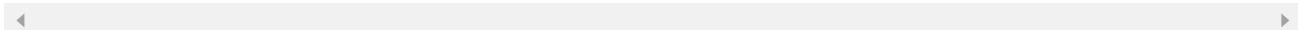


上一篇文章中，我们提到可以用“多线程版本的 if”来理解 Guarded Suspension 模式，不同于单线程中的 if，这个“多线程版本的 if”是需要等待的，而且还很执着，必须要等到条件为真。但很显然这个世界，不是所有场景都需要这么执着，有时候我们还需要快速放弃。

需要快速放弃的一个最常见的例子是各种编辑器提供的自动保存功能。自动保存功能的实现逻辑一般都是隔一定时间自动执行存盘操作，存盘操作的前提是文件做过修改，如果文件没有执行过修改操作，就需要快速放弃存盘操作。下面的示例代码将自动保存功能代码化了，很显然 AutoSaveEditor 这个类不是线程安全的，因为对共享变量 changed 的读写没有使用同步，那如何保证 AutoSaveEditor 的线程安全性呢？

复制代码

```
1 class AutoSaveEditor{  
2     // 文件是否被修改过  
3     boolean changed=false;  
4     // 定时任务线程池  
5     ScheduledExecutorService ses =  
6         Executors.newSingleThreadScheduledExecutor();  
7     // 定时执行自动保存  
8     void startAutoSave(){  
9         ses.scheduleWithFixedDelay(()->{  
10             autoSave();  
11         }, 5, 5, TimeUnit.SECONDS);  
12     }  
13     // 自动存盘操作  
14     void autoSave(){  
15         if (!changed) {  
16             return;  
17         }  
18         changed = false;  
19         // 执行存盘操作  
20         // 省略且实现  
21         this.execSave();  
22     }  
23     // 编辑操作  
24     void edit(){  
25         // 省略编辑逻辑  
26         .....  
27         changed = true;  
28     }  
29 }
```

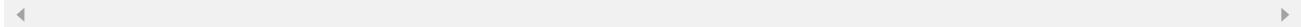


解决这个问题相信你一定手到擒来了：读写共享变量 `changed` 的方法 `autoSave()` 和 `edit()` 都加互斥锁就可以了。这样做虽然简单，但是性能很差，原因是锁的范围太大了。那我们可以将锁的范围缩小，只在读写共享变量 `changed` 的地方加锁，实现代码如下所示。

复制代码

```
1 // 自动存盘操作  
2 void autoSave(){  
3     synchronized(this){  
4         if (!changed) {  
5             return;  
6         }  
7         changed = false;  
8     }  
9     // 执行存盘操作  
10    // 省略且实现  
11    this.execSave();
```

```
12 }
13 // 编辑操作
14 void edit(){
15     // 省略编辑逻辑
16     .....
17     synchronized(this){
18         changed = true;
19     }
20 }
```



如果你深入地分析一下这个示例程序，你会发现，示例中的共享变量是一个状态变量，业务逻辑依赖于这个状态变量的状态：当状态满足某个条件时，执行某个业务逻辑，其本质其实不过就是一个 if 而已，放到多线程场景里，就是一种“多线程版本的 if”。这种“多线程版本的 if”的应用场景还是很多的，所以也有人把它总结成了一种设计模式，叫做**Balking 模式**。

Balking 模式的经典实现

Balking 模式本质上是一种规范化地解决“多线程版本的 if”的方案，对于上面自动保存的例子，使用 Balking 模式规范化之后的写法如下所示，你会发现仅仅是将 edit() 方法中对共享变量 changed 的赋值操作抽取到了 change() 中，这样的好处是将并发处理逻辑和业务逻辑分开。

复制代码

```
1 boolean changed=false;
2 // 自动存盘操作
3 void autoSave(){
4     synchronized(this){
5         if (!changed) {
6             return;
7         }
8         changed = false;
9     }
10    // 执行存盘操作
11    // 省略且实现
12    this.execSave();
13 }
14 // 编辑操作
15 void edit(){
16     // 省略编辑逻辑
17     .....
18     change();
19 }
```

```
20 // 改变状态
21 void change(){
22     synchronized(this){
23         changed = true;
24     }
25 }
```

用 volatile 实现 Balking 模式

前面我们用 synchronized 实现了 Balking 模式，这种实现方式最为稳妥，建议你实际工作中也使用这个方案。不过在某些特定场景下，也可以使用 volatile 来实现，但**使用 volatile 的前提是原子性没有要求**。

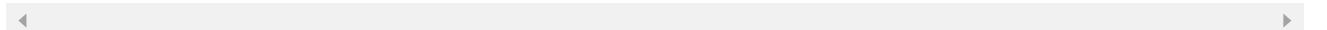
在 [《29 | Copy-on-Write 模式：不是延时策略的 COW》](#) 中，有一个 RPC 框架路由表的案例，在 RPC 框架中，本地路由表是要和注册中心进行信息同步的，应用启动的时候，会将应用依赖服务的路由表从注册中心同步到本地路由表中，如果应用重启的时候注册中心宕机，那么会导致该应用依赖的服务均不可用，因为找不到依赖服务的路由表。为了防止这种极端情况出现，RPC 框架可以将本地路由表自动保存到本地文件中，如果重启的时候注册中心宕机，那么就从本地文件中恢复重启前的路由表。这其实也是一种降级的方案。

自动保存路由表和前面介绍的编辑器自动保存原理是一样的，也可以用 Balking 模式实现，不过我们这里采用 volatile 来实现，实现的代码如下所示。之所以可以采用 volatile 来实现，是因为对共享变量 changed 和 rt 的写操作不存在原子性的要求，而且采用 scheduleWithFixedDelay() 这种调度方式能保证同一时刻只有一个线程执行 autoSave() 方法。

 复制代码

```
1 // 路由表信息
2 public class RouterTable {
3     //Key: 接口名
4     //Value: 路由集合
5     ConcurrentHashMap<String, CopyOnWriteArraySet<Router>>
6     rt = new ConcurrentHashMap<>();
7     // 路由表是否发生变化
8     volatile boolean changed;
9     // 将路由表写入本地文件的线程池
10    ScheduledExecutorService ses=
11        Executors.newSingleThreadScheduledExecutor();
12    // 启动定时任务
13    // 将变更后的路由表写入本地文件
```

```
14     public void startLocalSaver(){
15         ses.scheduleWithFixedDelay(()->{
16             autoSave();
17         }, 1, 1, MINUTES);
18     }
19     // 保存路由表到本地文件
20     void autoSave() {
21         if (!changed) {
22             return;
23         }
24         changed = false;
25         // 将路由表写入本地文件
26         // 省略其方法实现
27         this.save2Local();
28     }
29     // 删除路由
30     public void remove(Router router) {
31         Set<Router> set=rt.get(router iface);
32         if (set != null) {
33             set.remove(router);
34             // 路由表已发生变化
35             changed = true;
36         }
37     }
38     // 增加路由
39     public void add(Router router) {
40         Set<Router> set = rt.computeIfAbsent(
41             route iface, r ->
42                 new CopyOnWriteArraySet<>());
43         set.add(router);
44         // 路由表已发生变化
45         changed = true;
46     }
47 }
```



Balking 模式有一个非常典型的应用场景就是单次初始化，下面的示例代码是它的实现。这个实现方案中，我们将 init() 声明为一个同步方法，这样同一个时刻就只有一个线程能够执行 init() 方法；init() 方法在第一次执行完时会将 initited 设置为 true，这样后续执行 init() 方法的线程就不会再执行 doInit() 了。

复制代码

```
1 class InitTest{
2     boolean initited = false;
3     synchronized void init(){
4         if(initited){
5             return;
```

```
6     }
7     // 省略 doInit 的实现
8     doInit();
9     initied=true;
10    }
11 }
```

线程安全的单例模式本质上其实也是单次初始化，所以可以用 Balking 模式来实现线程安全的单例模式，下面的示例代码是其实现。这个实现虽然功能上没有问题，但是性能却很差，因为互斥锁 synchronized 将 getInstance() 方法串行化了，那有没有办法可以优化一下它的性能呢？

 复制代码

```
1 class Singleton{
2     private static
3         Singleton singleton;
4     // 构造方法私有化
5     private Singleton(){}
6     // 获取实例（单例）
7     public synchronized static
8     Singleton getInstance(){
9         if(singleton == null){
10             singleton=new Singleton();
11         }
12         return singleton;
13     }
14 }
```

办法当然是有的，那就是经典的**双重检查**（Double Check）方案，下面的示例代码是其详细实现。在双重检查方案中，一旦 Singleton 对象被成功创建之后，就不会执行 synchronized(Singleton.class){} 相关的代码，也就是说，此时 getInstance() 方法的执行路径是无锁的，从而解决了性能问题。不过需要你注意的是，这个方案中使用了 volatile 来禁止编译优化，其原因你可以参考 [《01 | 可见性、原子性和有序性问题：并发编程 Bug 的源头》](#) 中相关的内容。至于获取锁后的二次检查，则是出于对安全性负责。

 复制代码

```
1 class Singleton{
2     private static volatile
3         Singleton singleton;
```

```
4 // 构造方法私有化
5 private Singleton() {}
6 // 获取实例（单例）
7 public static Singleton
8 getInstance() {
9     // 第一次检查
10    ifsingleton==null){
11        synchronize{Singleton.class}{
12            // 获取锁后二次检查
13            ifsingleton==null{
14                singleton=new Singleton();
15            }
16        }
17    }
18    return singleton;
19 }
20 }
```

总结

Balking 模式和 Guarded Suspension 模式从实现上看似乎没有多大的关系，Balking 模式只需要用互斥锁就能解决，而 Guarded Suspension 模式则要用到管程这种高级的并发原语；但是从应用的角度来看，它们解决的都是“线程安全的 if”语义，不同之处在于，Guarded Suspension 模式会等待 if 条件为真，而 Balkng 模式不会等待。

Balking 模式的经典实现是使用互斥锁，你可以使用 Java 语言内置 synchronized，也可以使用 SDK 提供 Lock；如果你对互斥锁的性能不满意，可以尝试采用 volatile 方案，不过使用 volatile 方案需要你更加谨慎。

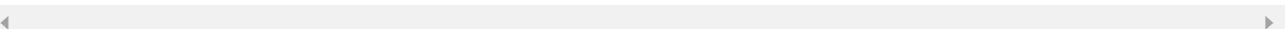
当然你也可以尝试使用双重检查方案来优化性能，双重检查中的第一次检查，完全是出于对性能的考量：避免执行加锁操作，因为加锁操作很耗时。而加锁之后的二次检查，则是出于对安全性负责。双重检查方案在优化加锁性能方面经常用到，例如 [《17 | ReadWriteLock：如何快速实现一个完备的缓存？》](#) 中实现缓存按需加载功能时，也用到了双重检查方案。

课后思考

下面的示例代码中，init() 方法的本意是：仅需计算一次 count 的值，采用了 Balkng 模式的 volatile 实现方式，你觉得这个实现是否有问题呢？

 复制代码

```
1 class Test{  
2     volatile boolean init = false;  
3     int count = 0;  
4     void init(){  
5         if(init){  
6             return;  
7         }  
8         init = true;  
9         // 计算 count 的值  
10        count = calc();  
11    }  
12 }
```



欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

 极客时间

Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 [31 | Guarded Suspension模式：等待唤醒机制的规范实现](#)

下一篇 [33 | Thread-Per-Message模式：最简单实用的分工方法](#)

精选留言 (18)

 写留言



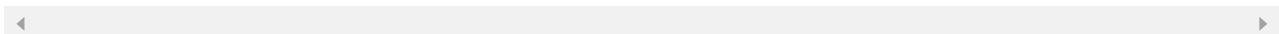
zero

2019-05-11

↑ 7

是有问题的，`volatile`关键字只能保证可见性，无法保证原子性和互斥性。所以`calc`方法有可能被重复执行。

作者回复: 





韩琪

2019-05-14

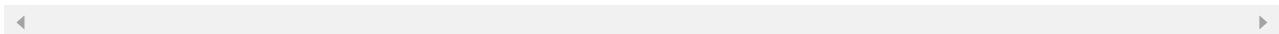
↑ 4

思考题代码相当于：

```
if (intied == false) { // 1  
    initied = true; //2  
    count = calc()  
}
```

展开 

作者回复: 





Corner

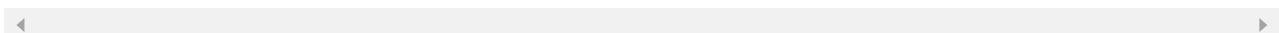
2019-05-11

↑ 1

最好就不要单独使用`volatile`防止产生线程安全问题。因为变量的读写是两个操作，和我们的直觉不一样，很容易出问题。老师的那个`volatile`就没有问题吗？如果一个线程修改了路由表，此时定时器任务判断共享变量为`true`，在将其修改为`false`之前，此时另一个线程又修改了路由表，然后定时任务继续执行会将其修改为`false`，这就出现问题了。最后还是要在`autoSave`方法上做同步的。

展开 

作者回复: 定时器任务只有一个线程，`autosave`加不加同步就无所谓了，多保存一次也没关系，这种概率毕竟很小





锦
2019-05-11

1

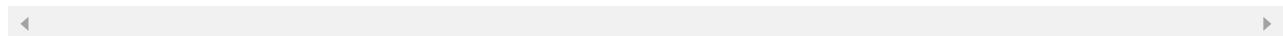
回答问题：

有问题，`volatile`不能保证原子性，题目要求只需计算一次Count，所以需要对共享变量`inited`加锁保护。

疑问：...

展开▼

作者回复：你没有办法控制调用方的线程数，`autosave`你是能控制的。不过加锁以后就串行了



points
2019-05-31

1

class Test{

```
AtomicBoolean inited = new AtomicBoolean(false);
```

```
void inited( ){...}
```

展开▼



Rancood
2019-05-22

1

这个Balking模式的好处就是将并发处理逻辑与业务逻辑分离吗

展开▼



贺宇
2019-05-21

1

这个问题好像和信号量那章的问题很相似

展开▼



ZOU志伟
2019-05-18

1

竞态条件问题

展开▼



Zach_
2019-05-16



没有锁 有共享变量 多个线程 可能同时读到false哇， 就可能有多个线程init而让count值超过1哇。

尽管读到了init=false, 真正的cal()也应该在同步里面，并且init此时任然是false哇~

展开 ▼

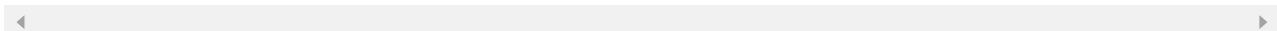


孙志强
2019-05-14



initied变量需要使用CAS的方式进行赋值，赋值失败就return，保证只有一个线程可以修改initied变量。

作者回复: ↗



张三
2019-05-12



有问题，在执行calc()方法之前，如果有别的线程进来，则直接返回count=0了，但第一个线程还是会执行calc()方法更新count值，安全性问题。



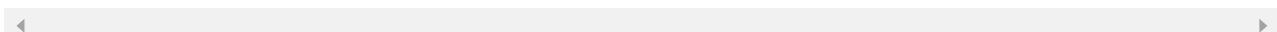
晓杰
2019-05-12



在微服务的场景下，synchornize应该不适用了吧

展开 ▼

作者回复: 不适用分布式情况的单例



JackJin
2019-05-11



老师， volatile只能保证变量的可见性，在多线程下，发生线程切换会都读取到变量为false，则计算count方法被调用多次，对吗？

作者回复: 凸对的



热台

2019-05-11



回答问题

- 1, cal () 可能被执行多次
2. 也可能cal () 执行结束前, count就被使用

解决方法...

展开 ▼

作者回复: 凸



刘晓林

2019-05-11



有问题, 存在竞态条件

展开 ▼

作者回复: 凸



ack

2019-05-11



有问题, initied共享变量和cal () 写操作需要保证原子性执行, 上面的初始化操作可能会执行多次



张三

2019-05-11



打卡, 文中关于使用volatile不需要考虑原子性的情况是什么意思呢?

展开 ▼



郑晨Cc

2019-05-11



第8行 `inited = true;` 改成cas操作
失败直接return。成功继续执行cal方法