32 | Balking模式：再谈线程安全的单例模式

2019-05-11 王宝令

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？

```
class AutoSaveEditor{
  //文件是否被修改过
  boolean changed=false;
  //定时任务线程池
  ScheduledExecutorService ses =
    Executors.newSingleThreadScheduledExecutor();
  //定时执行自动保存
  void startAutoSave(){
    ses.scheduleWithFixedDelay(()->{
      autoSave();
    }, 5, 5, TimeUnit.SECONDS);
  }
  //自动存盘操作
  void autoSave(){
    if (!changed) {
      return;
    }
    changed = false;
    //执行存盘操作
    //省略且实现
    this.execSave();
  }
  //编辑操作
  void edit(){
    //省略编辑逻辑
    ......
    changed = true;
  }
}
```

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。

```
//自动存盘操作
void autoSave(){
  synchronized(this){
    if (!changed) {
      return;
    }
    changed = false;
  }
  //执行存盘操作
  //省略且实现
  this.execSave();
}
//编辑操作
void edit(){
  //省略编辑逻辑
  ......
  synchronized(this){
    changed = true;
  }
}
```

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。

```
boolean changed=false;
//自动存盘操作
void autoSave(){
  synchronized(this){
    if (!changed) {
      return;
    }
    changed = false;
  }
  //执行存盘操作
  //省略且实现
  this.execSave();
}
//编辑操作
void edit(){
  //省略编辑逻辑
  ......
  change();
}
//改变状态
void change(){
  synchronized(this){
    changed = true;
  }
}
```

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的前提是对原子性没有要求**。

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。

```
//路由表信息
public class RouterTable {
  //Key:接口名
  //Value:路由集合
  ConcurrentHashMap<String, CopyOnWriteArraySet<Router>>
    rt = new ConcurrentHashMap<>();
  //路由表是否发生变化
  volatile boolean changed;
  //将路由表写入本地文件的线程池
  ScheduledExecutorService ses=
    Executors.newSingleThreadScheduledExecutor();
  //启动定时任务
  //将变更后的路由表写入本地文件
  public void startLocalSaver(){
    ses.scheduleWithFixedDelay(()->{
      autoSave();
    }, 1, 1, MINUTES);
  }
  //保存路由表到本地文件
  void autoSave() {
    if (!changed) {
      return;
    }
    changed = false;
    //将路由表写入本地文件
    //省略其方法实现
    this.save2Local();
  }
  //删除路由
  public void remove(Router router) {
    Set<Router> set=rt.get(router.iface);
```

```
    if (set != null) {
      set.remove(router);
      //路由表已发生变化
      changed = true;
    }
  }
  //增加路由
  public void add(Router router) {
    Set<Router> set = rt.computeIfAbsent(
      route.iface, r ->
        new CopyOnWriteArraySet<>());
    set.add(router);
    //路由表已发生变化
    changed = true;
  }
}
```

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inited设置为true，这样后续执行init()方法的线程就不会再执行doInit()了。

```
class InitTest{
  boolean inited = false;
  synchronized void init(){
    if(inited){
      return;
    }
    //省略doInit的实现
    doInit();
    inited=true;
  }
}
```

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？

```
class Singleton{
  private static
    Singleton singleton;
  //构造方法私有化
  private Singleton(){}
  //获取实例（单例）
  public synchronized static
  Singleton getInstance(){
    if(singleton == null){
      singleton=new Singleton();
    }
    return singleton;
  }
}
```

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。

```
class Singleton{
  private static volatile
    Singleton singleton;
  //构造方法私有化
  private Singleton() {}
  //获取实例（单例）
  public static Singleton
  getInstance() {
    //第一次检查
    if(singleton==null){
      synchronize{Singleton.class){
        //获取锁后二次检查
        if(singleton==null){
          singleton=new Singleton();
        }
      }
    }
    return singleton;
  }
}
```

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Balking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Balking模式的volatile实现方式，你觉得这个实现是否有问题呢？

```
class Test{
  volatile boolean inited = false;
  int count = 0;
  void init(){
    if(inited){
      return;
    }
    inited = true;
    //计算count的值
    count = calc();
  }
}
```

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。

精选留言

zero

8

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。

2019-05-11

作者回复

[]

2019-05-14

韩琪

5

思考题代码相当于：

if（intied == false） { // 1

inited = true; //2

count = calc()

}

可能有多条线程同时到1的位置，判断到inited为false，都进入2执行。

解决方案：

（1）加锁保护临界区

（2） AtomicBoolean.compareAndSet(false, true)

2019-05-14

作者回复

[]

2019-05-14

Corner

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。

2019-05-11

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

2019-05-15

锦

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：

public class RouterTable 类中AutoSave方法同一时刻只有一个线程调用，而Remove和Add方法也是要求使用方单线程访问吗？在实际开发中一般采用什么方式达成这种约定呢？

2019-05-11

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

2019-05-20

Jxin 0

volative修饰的属性。我见过在方法中。用局部变量接收该属性值，方法后续的操作都基于该局部变量。这样是不是就不再有volative的特性了？性能虽然提高了，毕竟能走缓存和编译优化了。但是就像上例双重检查的场景。这么个操作就依旧会有空指针异常的可能。请问老师我理解对吗。

2019-06-16

奇奇 0

课后思考题应该是!inited 代码是错的

2019-06-05

points 0

```
class Test{

AtomicBoolean inited = new AtomicBoolean(false);

void inited( ){
if( inited.getAndSet(true) ){
return ;
}

}
}
```

2019-05-31

Rancood 0

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

2019-05-22

贺宇 0

这个问题好像和信号量那章的问题很相似

2019-05-21

ZOU志伟 0

竞态条件问题

2019-05-18

Zach_ 0

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~

2019-05-16

孙志强 0

inited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inited变量。

2019-05-14

作者回复

2019-05-15

张三 0

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。

2019-05-12

晓杰 0

在微服务的场景下，synchornize应该不适用了吧

2019-05-12

作者回复

不适用分布式情况的单例

2019-05-13

JackJin 0

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？

2019-05-11

作者回复

对的

2019-05-13

热台 0

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

inited 赋值和cal（）执行放在一个同步块中，并增加双重check

2019-05-11

作者回复

2019-05-13

刘晓林

0

有问题，存在竞态条件

2019-05-11

作者回复

2019-05-13

ack

0

有问题，inited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

2019-05-11

张三

0

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？

2019-05-11

郑晨Cc

0

第8行 inited = true；改成cas操作

失败直接return。成功继续执行cal方法

2019-05-11