



下载APP

## 17 | 如何正确使用锁保护共享数据，协调异步线程？

2019-08-31 李玥

消息队列高手课

[进入课程 >](#)



讲述：李玥

时长 14:04 大小 12.89M



你好，我是李玥。

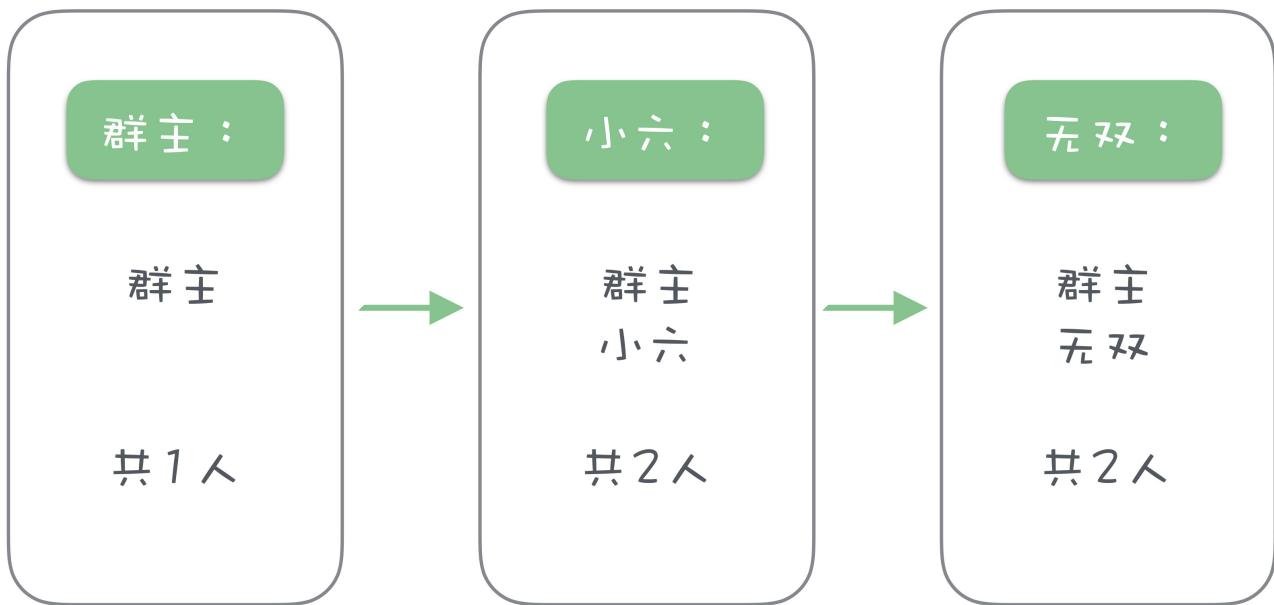
在前几天的加餐文章中我讲到，JMQ 为了提升整个流程的处理性能，使用了一个“近乎无锁”的设计，这里面其实隐含着两个信息点。第一个是，在消息队列中，“锁”是一个必须要使用的技术。第二个是，使用锁其实会降低系统的性能。

那么，如何正确使用锁，又需要注意哪些事项呢？今天我们就来聊一聊这个问题。

我们知道，使用异步和并发的设计可以大幅提升程序的性能，但我们为此付出的代价是，程序比原来更加复杂了，多线程在并行执行的时候，带来了很多不确定性。特别是对于一些需要多个线程并发读写的共享数据，如果处理不好，很可能会产出不可预期的结果，这肯定不是我们想要的。

我给你举个例子来说明一下，大家应该都参与过微信群投票吧？比如，群主说：“今晚儿咱们聚餐，能来的都回消息报一下名，顺便统计一下人数。都按我这个格式来报名。”然后，群主发了一条消息：“群主，1人”。

这时候小六和无双都要报名，过一会儿，他俩几乎同时各发了一条消息，“小六，2人”“无双，2人”，每个人发的消息都只统计了群主和他们自己，一共2人，而这时候，其实已经有3个人报名了，并且，在最后发消息的无双的名单中，小六的报名被覆盖了。



这就是一个非常典型的由于并发读写导致的数据错误。使用锁可以非常有效地解决这个问题。锁的原理是这样的：**任何时间都只能有一个线程持有锁，只有持有锁的线程才能访问被锁保护的资源。**

在上面微信群报名的例子中，如果说我们的微信群中有一把锁，想要报名的人必须先拿到锁，然后才能更新报名名单。这样，就避免了多个人同时更新消息，报名名单也就不会出错了。

## 避免滥用锁

那是不是遇到这种情况都要用锁解决呢？我分享一下我个人使用锁的第一条原则：**如果能不用锁，就不用锁；如果你不确定是不是应该用锁，那也不要用锁。**为什么这么说呢？因为，虽然说使用锁可以保护共享资源，但是代价还是不小的。

第一，加锁和解锁过程都是需要 CPU 时间的，这是一个性能的损失。另外，使用锁就有可能导致线程等待锁，等待锁过程中线程是阻塞的状态，过多的锁等待会显著降低程序的性能。

第二，如果对锁使用不当，很容易造成死锁，导致整个程序“卡死”，这是非常严重的问题。本来多线程的程序就非常难于调试，如果再加上锁，出现并发问题或者死锁问题，你的程序将更加难调试。

所以，你在使用锁以前，一定要非常清楚明确地知道，这个问题必须要用一把锁来解决。切忌看到一个共享数据，也搞不清它在并发环境中会不会出现争用问题，就“为了保险，给它加个锁吧。”**千万不能有这种不负责任的想法，否则你将会付出惨痛的代价！**我曾经遇到过的严重线上事故，其中有几次就是由于不当地使用锁导致的。

**只有在并发环境中，共享资源不支持并发访问，或者说并发访问共享资源会导致系统错误的情况下，才需要使用锁。**

## 锁的用法

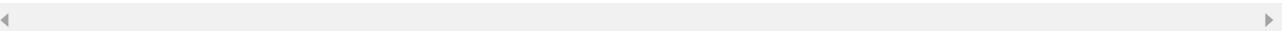
锁的用法一般是这样的：

1. 在访问共享资源之前，先获取锁。
2. 如果获取锁成功，就可以访问共享资源了。
3. 最后，需要释放锁，以便其他线程继续访问共享资源。

在 Java 语言中，使用锁的例子：

 复制代码

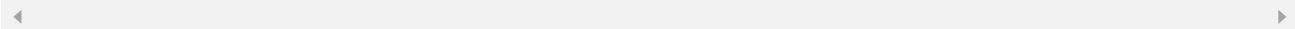
```
1 private Lock lock = new ReentrantLock();
2
3 public void visitShareResWithLock() {
4     lock.lock();
5     try {
6         // 在这里安全的访问共享资源
7     } finally {
8         lock.unlock();
9     }
10 }
```



也可以使用 `synchronized` 关键字，它的效果和锁是一样的：

 复制代码

```
1 private Object lock = new Object();
2
3 public void visitShareResWithLock() {
4     synchronized (lock) {
5         // 在这里安全的访问共享资源
6     }
7 }
```



使用锁的时候，你需要注意几个问题：

第一个，也是最重要的问题就是，**使用完锁，一定要释放它**。比较容易出现状况的地方是，很多语言都有异常机制，当抛出异常的时候，不再执行后面的代码。如果在访问共享资源时抛出异常，那后面释放锁的代码就不会被执行，这样，锁就一直无法释放，形成死锁。所以，你要考虑到代码可能走到的所有正常和异常的分支，确保所有情况下，锁都能被释放。

有些语言提供了 `try-with` 的机制，不需要显式地获取和释放锁，可以简化编程，有效避免这种问题，推荐你使用。

比如在 Python 中：

 复制代码

```
1 lock = threading.RLock()
2
3 def visitShareResWithLock():
4     with lock:
5         # 注意缩进
6         # 在这里安全的访问共享资源
7
8     # 锁会在 with 代码段执行完成后自动释放
```



接下来我们说一下，使用锁的时候，遇到的最常见的问题：死锁。

## 如何避免死锁？

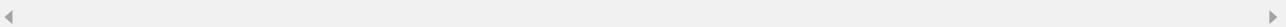
死锁是指，由于某种原因，锁一直没有释放，后续需要获取锁的线程都将处于等待锁的状态，这样程序就卡死了。

导致死锁的原因并不多，第一种原因就是我在刚刚讲的，获取了锁之后没有释放，有经验的程序员很少会犯这种错误，即使出现这种错误，也很容易通过查看代码找到 Bug。

还有一种是锁的重入问题，我们来看下面这段代码：

 复制代码

```
1 public void visitShareResWithLock() {  
2     lock.lock(); // 获取锁  
3     try {  
4         lock.lock(); // 再次获取锁，会导致死锁吗?  
5     } finally {  
6         lock.unlock();  
7     }  
}
```



在这段代码中，当前的线程获取到了锁 lock，然后在持有这把锁的情况下，再次去尝试获取这把锁，这样会导致死锁吗？

答案是，不一定。**会不会死锁取决于，你获取的这把锁它是不是可重入锁。**如果是可重入锁，那就没有问题，否则就会死锁。

大部分编程语言都提供了可重入锁，如果没有特别的要求，你要尽量使用可重入锁。有的同学可能会问，“既然已经获取到锁了，我干嘛还要再次获取同一把锁呢？”

其实，如果你的程序足够复杂，调用栈很深，很多情况下，当你需要获取一把锁的时候，你是不太好判断在 n 层调用之外的某个地方，是不是已经获取过这把锁了，这个时候，获取可重入锁就有意义了。

最后一种死锁的情况是最复杂的，也是最难解决的。如果你的程序中存在多把锁，就有可能出现这些锁互相锁住的情况。我们一起来看下面这段 Python 代码：

 复制代码

```
1 import threading  
2
```

```
3 def func1(lockA, lockB):
4     while True:
5         print("Thread1: Try to accquire lockA...")
6         with lockA:
7             print("Thread1: lockA accquired. Try to accquire lockB...")
8             with lockB:
9                 print("Thread1: Both lockA and LockB accrquired.")
10
11
12 def func2(lockA, lockB):
13     while True:
14         print("Thread2: Try to accquire lockB...")
15         with lockB:
16             print("Thread2: lockB accquired. Try to accquire lockA...")
17             with lockA:
18                 print("Thread2: Both lockA and LockB accrquired.")
19
20
21 if __name__ == '__main__':
22     lockA = threading.RLock();
23     lockB = threading.RLock()
24     t1 = threading.Thread(target=func1, args=(lockA, lockB,))
25     t2 = threading.Thread(target=func2, args=(lockA, lockB,))
26     t1.start()
27     t2.start()
```

这个代码模拟了一个最简单最典型的死锁情况。在这个程序里面，我们有两把锁：lockA 和 lockB，然后我们定义了两个线程，这两个线程反复地去获取这两把锁，然后释放。我们执行以下这段代码，看看会出现什么情况：

 复制代码

```
1 $ python3 DeadLock.py
2 Thread1: Try to accquire lockA...
3 Thread1: lockA accquired. Try to accquire lockB...
4 Thread1: Both lockA and LockB accrquired.
5 Thread1: Try to accquire lockA...
6 ...
7 Thread1: Try to accquire lockA...
8 Thread2: Try to accquire lockB...
9 Thread1: lockA accquired. Try to accquire lockB...
10 Thread2: lockB accquired. Try to accquire lockA...
```

可以看到，程序执行一会儿就卡住了，发生了死锁。那死锁的原因是什么呢？请注意看代码，这两个线程，他们获取锁的顺序是不一样的。第一个线程，先获取 lockA，再获取 lockB，而第二个线程正好相反，先获取 lockB，再获取 lockA。

然后，你再看一下死锁前的最后两行日志，线程 1 持有了 lockA，现在尝试获取 lockB，而线程 2 持有了 lockB，尝试获取 lockA。你可以想一下这个场景，两个线程，各持有一把锁，都等着对方手里的另外一把锁，这样就僵持住了。

这是最简单的两把锁两个线程死锁的情况，我们还可以分析清楚，你想想如果你的程序中有十几把锁，几十处加锁解锁，几百的线程，如果出现死锁你还能分析清楚是什么情况吗？

关于避免死锁，我在这里给你几点建议。

1. 再次强调一下，避免滥用锁，程序里用的锁少，写出死锁 Bug 的几率自然就低。
2. 对于同一把锁，加锁和解锁必须要放在同一个方法中，这样一次加锁对应一次解锁，代码清晰简单，便于分析问题。
3. 尽量避免在持有一把锁的情况下，去获取另外一把锁，就是要尽量避免同时持有多把锁。
4. 如果需要持有多把锁，一定要注意加解锁的顺序，解锁的顺序要和加锁顺序相反。比如，获取三把锁的顺序是 A、B、C，释放锁的顺序必须是 C、B、A。
5. 给你程序中所有的锁排一个顺序，在所有需要加锁的地方，按照同样的顺序加解锁。比如我刚刚举的那个例子，如果两个线程都按照先获取 lockA 再获取 lockB 的顺序加锁，就不会产生死锁。

最后，你需要知道，即使你完全遵从我这些建议，我也无法完全保证你写出的程序就没有死锁，只能说，会降低一些犯错误的概率。

## 使用读写锁要兼顾性能和安全性

对于共享数据来说，如果说某个方法在访问它的时候，只是去读取，并不更新数据，那是不是就不需要加锁呢？还是需要的，因为如果一个线程读数据的同时，另外一个线程同时在更新数据，那么你读到的数据有可能是更新到一半的数据，这肯定是不符合预期的。所以，无论是只读访问，还是读写访问，都是需要加锁的。

如果给数据简单地加一把锁，虽然解决了安全性的的问题，但是牺牲了性能，因为，那无论读还是写，都无法并发了，跟单线程的程序性能是一样。

实际上，如果没有线程在更新数据，那即使多个线程都在并发读，也是没有问题的。我在上节课跟你讲过，大部分情况下，数据的读写比是不均衡的，读要远远多于写，所以，我们希望的是：

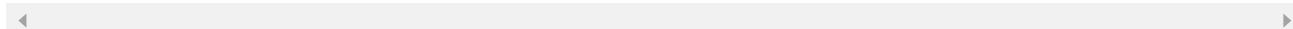
读访问可以并发执行。

写的同时不能并发读，也不能并发写。

这样就兼顾了性能和安全性。读写锁就是为这一需求设计的。我们来看一下 Java 中提供的读写锁：

 复制代码

```
1 ReadWriteLock rwlock = new ReentrantReadWriteLock();
2
3 public void read() {
4     rwlock.readLock().lock();
5     try {
6         // 在这儿读取共享数据
7     } finally {
8         rwlock.readLock().unlock();
9     }
10 }
11 public void write() {
12     rwlock.writeLock().lock();
13     try {
14         // 在这儿更新共享数据
15     } finally {
16         rwlock.writeLock().unlock();
17     }
18 }
```



在这段代码中，需要读数据的时候，我们获取读锁，获取到的读锁不是一个互斥锁，也就是说 `read()` 方法是可以多个线程并行执行的，这样使得读数据的性能依然很好。写数据的时候，我们获取写锁，当一个线程持有写锁的时候，其他线程既无法获取读锁，也不能获取写锁，达到保护共享数据的目的。

这样，使用读写锁就兼顾了性能和安全。

## 小结

锁可以保护共享资源，避免并发更新造成的数据错误。只有持有锁的线程才能访问被保护资源。线程在访问资源之前必须获取锁，访问完成后一定要记得释放锁。

一定不要滥用锁，否则容易导致死锁。死锁的原因，主要由于多个线程中多把锁相互争用导致的。一般来说，如果程序中使用的锁比较多，很难分析死锁的原因，所以需要尽量少的使用锁，并且保持程序的结构尽量简单、清晰。

最后，我们介绍了读写锁，在某些场景下，使用读写锁可以兼顾性能和安全性，是非常好的选择。

## 思考题

我刚刚讲到，Python 中提供了 `try-with-lock`，不需要显式地获取和释放锁，非常方便。遗憾的是，在 Java 中并没有这样的机制，那能不能自己在 Java 中实现一个 `try-with-lock` 呢？

欢迎你把代码上传到 GitHub 上，然后在评论区给出访问链接。如果你有任何问题，也可以在评论区留言与我交流。

感谢阅读，如果你觉得这篇文章对你有一些启发，也欢迎把它分享给你的朋友。

极客时间

# 消息队列高手课

从源码角度全面解析 MQ 的设计与实现

李玥

京东零售技术架构部资深架构师



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。

上一篇 16 | 缓存策略：如何使用缓存来减少磁盘IO？

## 精选留言 (12)

 写留言



L!en6o

2019-08-31

加一个锁回调 封装起来 实现 try-with-lock

展开 ▾



2



张三

2019-08-31

幸亏学过极客时间的并发编程专栏，看懂了。我觉得并发容器的选择比较复杂。



2



糖醋

2019-09-01

java7开始io就有try-with-resource。

可以利用这一个特性，来说实现，自动释放。

代码如下：

```
public class AutoUnlockProxy implements Closeable {...
```

展开 ▾

作者回复: 凸凸凸

2

1



刘天鹏

2019-09-02

对于golang应该就是这样吧

```
func foo(){  
    lock.Lock()  
    defer lock.Unlock()
```

//do something.....

展开 ▼



你说的灰

2019-09-02

```
public void visitShareResWithLock() {  
    lock.lock();  
    try {  
        // 在这里安全的访问共享资源...  
    } finally {  
        lock.unlock();  
    }  
}
```

展开 ▼



Cast

2019-09-01

老师，请问为什么要按逆序去释放锁呢？按照获取的顺序去释放好像也没什么毛病吧？



游弋云端

2019-09-01

ABBA锁最容易出问题，老师的经验很重要，尽可能避免锁中锁。



humor

2019-09-01

```
/**  
 *业务调用接口  
 **/  
public interface Invoker{  
    void invoke();...  
}
```

展开 ▼



monalisali

2019-08-31

老师，请教一个问题：假设有一个方法在计算报表，但这个计算的线程在执行过程中被意外释放了（并不是抛异常），此时try catch捕获是捕获不到这种情况的。而从客户端看来，这个计算过程就永远停在那里了，后台又没能力告诉客户端：“你别等了”。这种情况应该如果处理呢？

展开 ▼

1



树梢的果实

2019-08-31

C语言下，通过宏很容易实现try-with-lock。

如果两个线程中获取mutex的顺序不一致，可以通过增加第三个mutex来避免死锁。

既然我们做异步、并行，磁盘读写也可以这么做啊，加一个queue，所有读写操作请求都放到queue中，在单独的线程中完成IO操作并通过callback或另一个queue返回结果。不知服务器上这么做有什么不妥？

展开 ▼

作者回复: 这样做是可以的，其实你用的这个阻塞队列它就是用锁来实现的。



1



许童童

2019-08-31

老师这篇文章的分享对我这样的非后端程序员很友好，感谢老师的分享。

1



⑩

2019-08-31

锁是为了并发时的共享而创建，如果没有共享的真正需求不应该使用锁。锁带来的最大问题是复杂度和心智负担上升，所以很多框架把最复杂的实现隐藏在内部，留给使用者使用准则

1

