16 | Semaphore：如何快速实现一个限流器？

2019-04-04 王宝令

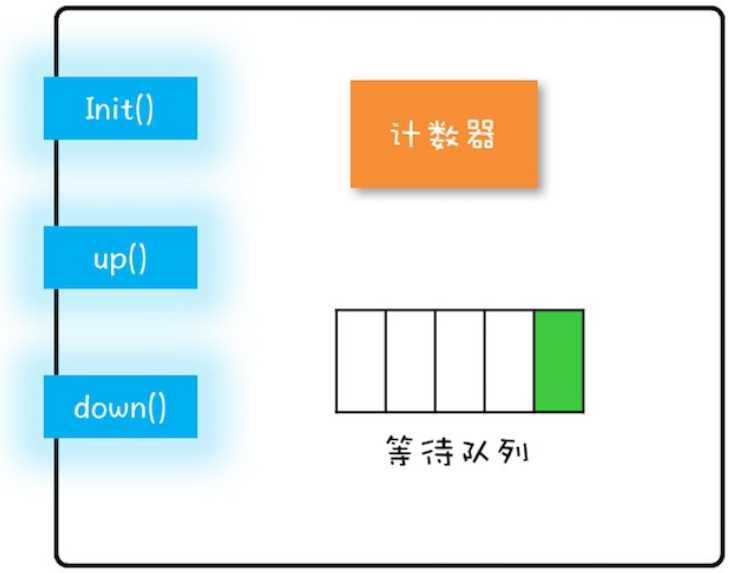
Semaphore，现在普遍翻译为“信号量”，以前也曾被翻译成“信号灯”，因为类似现实生活里的红绿灯，车辆能不能通行，要看是不是绿灯。同样，在编程世界里，线程能不能执行，也要看信号量是不是允许。

信号量是由大名鼎鼎的计算机科学家迪杰斯特拉（Dijkstra）于1965年提出，在这之后的15年，信号量一直都是并发编程领域的终结者，直到1980年管程被提出来，我们才有了第二选择。目前几乎所有支持并发编程的语言都支持信号量机制，所以学好信号量还是很有必要的。

下面我们首先介绍信号量模型，之后介绍如何使用信号量，最后我们再用信号量来实现一个限流器。

信号量模型

信号量模型还是很简单的，可以简单概括为：**一个计数器，一个等待队列，三个方法**。在信号量模型里，计数器和等待队列对外是透明的，所以只能通过信号量模型提供的三个方法来访问它们，这三个方法分别是：init()、down()和up()。你可以结合下图来形象化地理解。



信号量模型图

这三个方法详细的语义具体如下所示。

- init()：设置计数器的初始值。
- down()：计数器的值减1；如果此时计数器的值小于0，则当前线程将被阻塞，否则当前线程可以继续执行。
- up()：计数器的值加1；如果此时计数器的值小于或者等于0，则唤醒等待队列中的一个线程，并将其从等待队列中移除。

这里提到的init()、down()和up()三个方法都是原子性的，并且这个原子性是由信号量模型的实现方保证的。在Java SDK里面，信号量模型是由java.util.concurrent.Semaphore实现的，Semaphore这个类能够保证这三个方法都是原子操作。

如果你觉得上面的描述有点绕的话，可以参考下面这个代码化的信号量模型。

```
class Semaphore{
  // 计数器
  int count;
  // 等待队列
  Queue queue;
  // 初始化操作
  Semaphore(int c){
    this.count=c;
  }
  //
  void down(){
    this.count--;
    if(this.count<0){
      //将当前线程插入等待队列
      //阻塞当前线程
    }
  }
  void up(){
    this.count++;
    if(this.count<=0) {
      //移除等待队列中的某个线程T
      //唤醒线程T
    }
  }
}
```

这里再插一句，信号量模型里面，down()、up()这两个操作历史上最早称为P操作和V操作，所以信号量模型也被称为PV原语。另外，还有些人喜欢用semWait()和semSignal()来称呼它们，虽然叫法不同，但是语义都是相同的。在Java SDK并发包里，down()和up()对应的则是acquire()和release()。

如何使用信号量

通过上文，你应该会发现信号量的模型还是很简单的，那具体该如何使用呢？其实你想想红绿灯就可以了。十字路口的红绿灯可以控制交通，得益于它的一个关键规则：车辆在通过路口前必须先检查是否是绿灯，只有绿灯才能通行。这个规则和我们前面提到的锁规则是不是很类似？

其实，信号量的使用也是类似的。这里我们还是用累加器的例子来说明信号量的使用吧。在累加

器的例子里面，count+=1操作是个临界区，只允许一个线程执行，也就是说要保证互斥。那这种情况用信号量怎么控制呢？

其实很简单，就像我们用互斥锁一样，只需要在进入临界区之前执行一下down()操作，退出临界区之前执行一下up()操作就可以了。下面是Java代码的示例，acquire()就是信号量里的down()操作，release()就是信号量里的up()操作。

```
static int count;
//初始化信号量
static final Semaphore s
    = new Semaphore(1);
//用信号量保证互斥
static void addOne() {
  s.acquire();
  try {
    count+=1;
  } finally {
    s.release();
  }
}
```

下面我们再来分析一下，信号量是如何保证互斥的。假设两个线程T1和T2同时访问addOne()方法，当它们同时调用acquire()的时候，由于acquire()是一个原子操作，所以只能有一个线程（假设T1）把信号量里的计数器减为0，另外一个线程（T2）则是将计数器减为-1。对于线程T1，信号量里面的计数器的值是0，大于等于0，所以线程T1会继续执行；对于线程T2，信号量里面的计数器的值是-1，小于0，按照信号量模型里对down()操作的描述，线程T2将被阻塞。所以此时只有线程T1会进入临界区执行count+=1；。

当线程T1执行release()操作，也就是up()操作的时候，信号量里计数器的值是-1，加1之后的值是0，小于等于0，按照信号量模型里对up()操作的描述，此时等待队列中的T2将会被唤醒。于是T2在T1执行完临界区代码之后才获得了进入临界区执行的机会，从而保证了互斥性。

快速实现一个限流器

上面的例子，我们用信号量实现了一个最简单的互斥锁功能。估计你会觉得奇怪，既然有Java SDK里面提供了Lock，为啥还要提供一个Semaphore？其实实现一个互斥锁，仅仅是Semaphore的部分功能，Semaphore还有一个功能是Lock不容易实现的，那就是：**Semaphore可以允许多个线程访问一个临界区**。

现实中还有这种需求？有的。比较常见的需求就是我们工作中遇到的各种池化资源，例如连接池、对象池、线程池等等。其中，你可能最熟悉数据库连接池，在同一时刻，一定是允许多个线程同时使用连接池的，当然，每个连接在被释放前，是不允许其他线程使用的。

其实前不久，我在工作中也遇到了一个对象池的需求。所谓对象池呢，指的是一次性创建出N个对象，之后所有的线程重复利用这N个对象，当然对象在被释放前，也是不允许其他线程使用的。对象池，可以用List保存实例对象，这个很简单。但关键是限流器的设计，这里的限流，指的是不允许多于N个线程同时进入临界区。那如何快速实现一个这样的限流器呢？这种场景，我立刻就想到了信号量的解决方案。

信号量的计数器，在上面的例子中，我们设置成了1，这个1表示只允许一个线程进入临界区，但如果我们把计数器的值设置成对象池里对象的个数N，就能完美解决对象池的限流问题了。下面就是对象池的示例代码。

```
class ObjPool<T, R> {
  final List<T> pool;
  // 用信号量实现限流器
  final Semaphore sem;
  // 构造函数
  ObjPool(int size, T t){
    pool = new Vector<T>(){};
    for(int i=0; i<size; i++){
      pool.add(t);
    }
    sem = new Semaphore(size);
  }
  // 利用对象池的对象，调用func
  R exec(Function<T,R> func) {
    T t = null;
    sem.acquire();
    try {
      t = pool.remove(0);
      return func.apply(t);
    } finally {
      pool.add(t);
      sem.release();
    }
  }
}
// 创建对象池
ObjPool<Long, String> pool =
  new ObjPool<Long, String>(10, 2);
// 通过对象池获取t，之后执行
pool.exec(t -> {
    System.out.println(t);
    return t.toString();
});
```

我们用一个List来保存对象实例，用Semaphore实现限流器。关键的代码是ObjPool里面的exec()方法，这个方法里面实现了限流的功能。在这个方法里面，我们首先调用acquire()方法（与之匹

配的是在finally里面调用release()方法），假设对象池的大小是10，信号量的计数器初始化为10，那么前10个线程调用acquire()方法，都能继续执行，相当于通过了信号灯，而其他线程则会阻塞在acquire()方法上。对于通过信号灯的线程，我们为每个线程分配了一个对象 t（这个分配工作是通过pool.remove(0)实现的），分配完之后会执行一个回调函数func，而函数的参数正是前面分配的对象 t；执行完回调函数之后，它们就会释放对象（这个释放工作是通过pool.add(t)实现的），同时调用release()方法来更新信号量的计数器。如果此时信号量里计数器的值小于等于0，那么说明有线程在等待，此时会自动唤醒等待的线程。

简言之，使用信号量，我们可以轻松地实现一个限流器，使用起来还是非常简单的。

总结

信号量在Java语言里面名气并不算大，但是在其他语言里却是很有知名度的。Java在并发编程领域走的很快，重点支持的还是管程模型。 管程模型理论上解决了信号量模型的一些不足，主要体现在易用性和工程化方面，例如用信号量解决我们曾经提到过的阻塞队列问题，就比管程模型麻烦很多，你如果感兴趣，可以课下了解和尝试一下。

课后思考

在上面对象池的例子中，对象保存在了Vector中，Vector是Java提供的线程安全的容器，如果我们把Vector换成ArrayList，是否可以呢？

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

老杨同志 35

需要用线程安全的vector，因为信号量支持多个线程进入临界区，执行list的add和remove方法时可能是多线程并发执行

2019-04-04

作者回复

[]

2019-04-04

CCC 28

我理解的和管程相比，信号量可以实现的独特功能就是同时允许多个线程进入临界区，但是信号量不能做的就是同时唤醒多个线程去争抢锁，只能唤醒一个阻塞中的线程，而且信号量模型是没有Condition的概念的，即阻塞线程被醒了直接就运行了而不会去检查此时临界条件是否已经不满足了，基于此考虑信号量模型才会设计出只能让一个线程被唤醒，否则就会出现因为缺少Condition检查而带来的线程安全问题。正因为缺失了Condition，所以用信号量来实现阻塞队列就很麻烦，因为要自己实现类似Condition的逻辑。

2019-04-04

作者回复

[[[]]]

2019-04-04

任大鹏 9

有同学认为up()中的判断条件应该>=0，我觉得有可能理解为生产者-消费者模式中的生产者了。可以这么想，>0就意味着没有阻塞的线程了，所以只有<=0的情况才需要唤醒一个等待的线程。其实down()和up()是成对出现的，并且是先调用down()获得锁，处理完成再调用up()释放锁，如果信号量初始值为1，应该是不会出现>0的情况的，除非故意调先用up()，这也失去了信号量本身的意义了。不知道我理解的对不对。

2019-04-04

作者回复

对[[[]]]

2019-04-05

crazypokerk 8

文中，up()：计数器的值加 1；如果此时计数器的值小于或者等于0，这句话应该是大于等于0吧

2019-04-04

ken 4

public class Food {

public String name;

```
private long warmTime;

public Food(String name, long warmTime) {
this.name = name;
this.warmTime = warmTime;
}

public String getName() {
return name;
}

public long getWarmTime() {
return warmTime;
}
}



public class MicrowaveOven {

public String name;

public MicrowaveOven(String name) {
this.name = name;
}

public Food warm(Food food) {
long second = food.getWarmTime() * 1000;
try {
Thread.sleep(second);
} catch (InterruptedException e) {
e.printStackTrace();
}

System.out.println(String.format("%s warm %s %d seconds food.", name,food.getName() ,foo
d.getWarmTime()));
return food;
}

public String getName() {
return name;
}
```

```
}
public class MicrowaveOvenPool {

private List<MicrowaveOven> microwaveOvens;

private Semaphore semaphore;

public MicrowaveOvenPool(int size,@NotNull List<MicrowaveOven> microwaveOvens) {
this.microwaveOvens = new Vector<>(microwaveOvens);
this.semaphore = new Semaphore(size);
}
public Food exec(Function<MicrowaveOven, Food> func) {
MicrowaveOven microwaveOven = null;
try{
semaphore.acquire();
microwaveOven = microwaveOvens.remove(0);
return func.apply(microwaveOven);
}catch (InterruptedException e) {
e.printStackTrace();
} finally {
microwaveOvens.add(microwaveOven);
semaphore.release();
}
return null;
}

}
```

2019-04-08

作者回复

[]

2019-04-09

缪文@有赞 3

这个限流器实际上限的是并发量，也就是同时允许多少个请求通过，如果限制每秒请求数，不是这个实现的吧

2019-04-06

作者回复

后面会介绍guava的限流器

2019-04-06

master 3

老师，void up()方法中的this.count判断条件是否应该为>=0

2019-04-04

小和尚笨南北 3

semaphore底层通过AQS实现，AQS内部通过一个volatile变量间接实现同步。
根据happen-before原则的volatile规则和传递性规则。使用arraylist也不会发生线程安全问题。

2019-04-04

作者回复

不可以，有多个线程进入临界区

2019-04-05

陈华应 2

不可以，临界区会有多个线程并发执行

2019-04-06

shawn 2

老师能否把课程所有的完整代码放到github上，这样我们学起来更方便。包括全面几章的也发下，因为有时候根据您的代码，我没法运行

2019-04-04

Mr Q. 1

创建对象池的时候都是添加的同一个对象。

2019-05-17

长眉_张永 1

对于进入的多个线程资源之间，如果有公用的信息的话，是否还需要加锁操作呢？

2019-04-09

作者回复

需要

2019-04-09

木偶人King 1

```
ObjPool(int size, T t){
pool = new Vector<T>(){};
for(int i=0; i<size; i++){
pool.add(t);
}
sem = new Semaphore(size);
}
//------------------------------
```

老师这里pool.add(t) 一直循环添加的是同一个引用对象。没太明白。 为什么不是添加不同的t

2019-04-09

作者回复

实际项目中一定是不同的

2019-04-09

QQ怪 1

用初始化为1的Semaphore和管程来单单控制线程安全，哪个更有优势？为啥java不直接用信号量来实现互斥?

2019-04-05

作者回复

如果仅仅是为了互斥，都可以。

2019-04-05

Presley 1

进入临界区的N个线程不安全。add/remove都是不安全的。拿remove举例, ArrayList remove()源码：

```
public E remove(int index) {
rangeCheck(index);

modCount++;

// 假设连个线程 t1,t2都执行到这一步，t1 让出cpu,t2执行
E oldValue = elementData(index);
// 到这步,t1继续执行，这时t1,t2拿到的oldValue是一样的，两个线程能拿到同一个对象，明显线程不安全啊

int numMoved = size - index - 1;
if (numMoved > 0)
System.arraycopy(elementData, index+1, elementData, index,
numMoved);
elementData[--size] = null; // clear to let GC do its work

return oldValue;
}
```

2019-04-04

作者回复

[]

2019-04-05

摇山樵客™ 1

换ArrayList是不行的，临界区内可能存在多个线程来执行remove操作，出现不可预知的后果。

对于chaos同学说return之前释放的问题，我觉得可以这么理解：return的是执行后的结果，而不是“执行”。所以顺序应该是这样的：1acquire；2apply；3finally release；4return2的结果

2019-04-04

作者回复

是的，感谢回复的这么详细！！！

2019-04-04

crazypokerk

1

老师，那个计数器中得s.acquire()是需要捕获异常的。

```
static int count;
static final Semaphore s = new Semaphore(1);

static void addOne() throws InterruptedException {
s.acquire();
try {
count += 1;
}finally {
s.release();
}
}
```

2019-04-04

作者回复

异常都被我省略了，这样代码更能专注的表达问题，如果你本地实验，加上就可以了。手机屏幕太小，折行后行数太多，看到后面忘了前面，所以我尽讲精炼代码

2019-04-05

超

0

结合线程池的场景进行使用比较多。

2019-05-28

疯狂埃里克

0

针对任何一个key的写入都有全局的写锁，这不合适吧？如果想给每个key建立自己的锁怎么做呢？

2019-05-15

不瘦二十斤不换头像

谢特

0

为什么都是小于0

2019-05-09