37-设计模式模块热点问题答疑

多线程设计模式是前人解决并发问题的经验总结，当我们试图解决一个并发问题时，首选方案往往是使用匹配的设计模式，这样能避免走弯路。同时，由于大家都熟悉设计模式，所以使用设计模式还能提升方案和代码的可理解性。

在这个模块，我们总共介绍了9种常见的多线程设计模式。下面我们就对这9种设计模式做个分类和总结，同时也对前面各章的课后思考题做个答疑。

避免共享的设计模式

Immutability模式、**Copy-on-Write模式**和**线程本地存储模式**本质上都是**为了避免共享**，只是实现手段不同而已。这3种设计模式的实现都很简单，但是实现过程中有些细节还是需要格外注意的。例如，**使用Immutability模式需要注意对象属性的不可变性，使用Copy-on-Write模式需要注意性能问题，使用线程本地存储模式需要注意异步执行问题**。所以，每篇文章最后我设置的课后思考题的目的就是提醒你注意这些细节。

《28 | Immutability模式：如何利用不变性解决并发问题？》的课后思考题是讨论Account这个类是不是具备不可变性。这个类初看上去属于不可变对象的中规中矩实现，而实质上这个实现是有问题的，原因在于StringBuffer不同于String，StringBuffer不具备不可变性，通过getUser()方法获取user之后，是可以修改user的。一个简单的解决方案是让getUser()方法返回String对象。

```
public final class Account{
  private final
    StringBuffer user;
  public Account(String user){
    this.user =
      new StringBuffer(user);
  }
  //返回的StringBuffer并不具备不可变性
  public StringBuffer getUser(){
    return this.user;
  }
  public String toString(){
    return "user"+user;
  }
}
```

《29 | Copy-on-Write模式：不是延时策略的COW》的课后思考题是讨论Java SDK中为什么没有提供CopyOnWriteLinkedList。这是一个开放性的问题，没有标准答案，但是性能问题一定是其中一个很重要的原因，毕竟完整地复制LinkedList性能开销太大了。

《30 | 线程本地存储模式：没有共享，就没有伤害》的课后思考题是在异步场景中，是否可以使用 Spring 的事务管理器。答案显然是不能的，Spring 使用 ThreadLocal 来传递事务信息，因此这个事务信息是不能跨线程共享的。实际工作中有很多类库都是用 ThreadLocal 传递上下文信息的，这种场景下如果有异步操作，一定要注意上下文信息是不能跨线程共享的。

多线程版本IF的设计模式

Guarded Suspension模式和**Balking模式**都可以简单地理解为“多线程版本的if”，但它们的区别在于前者会等待if条件变为真，而后者则不需要等待。

Guarded Suspension模式的经典实现是使用**管程**，很多初学者会简单地用线程sleep的方式实现，比如《31 | Guarded Suspension模式：等待唤醒机制的规范实现》的思考题就是用线程sleep方式实现的。但不推荐你使用这种方式，最重要的原因是性能，如果sleep的时间太长，会影响响应时间；sleep的时间太短，会导致线程频繁地被唤醒，消耗系统资源。

同时，示例代码的实现也有问题：由于obj不是volatile变量，所以即便obj被设置了正确的值，执行`while(!p.test(obj))`的线程也有可能看不到，从而导致更长时间的sleep。

```
//获取受保护对象
T get(Predicate<T> p) {
  try {
    //obj的可见性无法保证
    while(!p.test(obj)){
      TimeUnit.SECONDS
        .sleep(timeout);
    }
  }catch(InterruptedException e){
    throw new RuntimeException(e);
  }
  //返回非空的受保护对象
  return obj;
}
//事件通知方法
void onChanged(T obj) {
  this.obj = obj;
}
```

实现Balking模式最容易忽视的就是**竞态条件问题**。比如，《32 | Balking模式：再谈线程安全的单例模式》的思考题就存在竞态条件问题。因此，在多线程场景中使用if语句时，一定要多问自己一遍：是否存在竞态条件。

```
class Test{
  volatile boolean inited = false;
  int count = 0;
  void init(){
    //存在竞态条件
    if(inited){
      return;
    }
    //有可能多个线程执行到这里
    inited = true;
    //计算count的值
    count = calc();
  }
}
```

三种最简单的分工模式

Thread-Per-Message模式、Worker Thread模式和**生产者-消费者模式**是三种**最简单实用的多线程分工方法**。虽说简单，但也还是有许多细节需要你多加小心和注意。

Thread-Per-Message模式在实现的时候需要注意是否存在线程的频繁创建、销毁以及是否可能导致OOM。在《33 | Thread-Per-Message模式：最简单实用的分工方法》文章中，最后的思考题就是关于如何快速解决OOM问题的。在高并发场景中，最简单的办法其实是**限流**。当然，限流方案也并不局限于解决Thread-Per-Message模式中的OOM问题。

Worker Thread模式的实现，需要注意潜在的线程**死锁问题**。《34 | Worker Thread模式：如何避免重复创建线程？》思考题中的示例代码就存在线程死锁。有名叫vector的同学关于这道思考题的留言，我觉得描述得很贴切和形象：“工厂里只有一个工人，他的工作就是同步地等待工厂里其他人给他提供东西，然而并没有其他人，他将等到天荒地老，海枯石烂！”因此，共享线程池虽然能够提供线程池的使用效率，但一定要保证一个前提，那就是：**任务之间没有依赖关系**。

```
ExecutorService pool = Executors
  .newSingleThreadExecutor();
//提交主任务
pool.submit(() -> {
  try {
    //提交子任务并等待其完成，
    //会导致线程死锁
    String qq=pool.submit(()->"QQ").get();
    System.out.println(qq);
  } catch (Exception e) {
  }
});
```

Java线程池本身就是一种生产者-消费者模式的实现，所以大部分场景你都不需要自己实现，直接使用Java的线程池就可以了。但若能自己灵活地实现生产者-消费者模式会更好，比如可以实现批量执行和分阶段提交，不过这过程中还需要注意如何优雅地终止线程，《36 | 生产者-消费者模式：用流水线思想提高效率》的思考题就是关于此的。

如何优雅地终止线程？我们在《35 | 两阶段终止模式：如何优雅地终止线程？》有过详细介绍，两阶段终止模式是一种通用的解决方案。但其实终止生产者-消费者服务还有一种更简单的方案，叫做**“毒丸”对象**。《Java并发编程实战》第7章的7.2.3节对“毒丸”对象有过详细的介绍。简单来讲，“毒丸”对象是生产者生产的一条特殊任务，然后当消费者线程读到“毒丸”对象时，会立即终止自身的执行。

下面是用“毒丸”对象终止写日志线程的具体实现，整体的实现过程还是很简单的：类Logger中声明了一个“毒丸”对象poisonPill，当消费者线程从阻塞队列bq中取出一条LogMsg后，先判断是否是“毒丸”对象，如果是，则break while循环，从而终止自己的执行。

```
class Logger {
  //用于终止日志执行的“毒丸”
  final LogMsg poisonPill =
    new LogMsg(LEVEL.ERROR, "");
  //任务队列
  final BlockingQueue<LogMsg> bq
    = new BlockingQueue<>();
```

```
  //只需要一个线程写日志
  ExecutorService es =
    Executors.newFixedThreadPool(1);
  //启动写日志线程
  void start(){
    File file=File.createTempFile(
      "foo", ".log");
    final FileWriter writer=
      new FileWriter(file);
    this.es.execute(()->{
      try {
        while (true) {
          LogMsg log = bq.poll(
            5, TimeUnit.SECONDS);
          //如果是“毒丸”，终止执行
          if(poisonPill.equals(logMsg)){
            break;
          }
          //省略执行逻辑
        }
      } catch(Exception e){
      } finally {
        try {
          writer.flush();
          writer.close();
        }catch(IOException e){}
      }
    });
  }
  //终止写日志线程
  public void stop() {
    //将“毒丸”对象加入阻塞队列
    bq.add(poisonPill);
    es.shutdown();
  }
}
```

总结

到今天为止，“并发设计模式”模块就告一段落了，多线程的设计模式当然不止我们提到的这9种，不过这里提到的这9种设计模式一定是最简单实用的。如果感兴趣，你也可以结合《图解Java多线程设计模式》这本书来深入学习这个模块，这是一本不错的并发编程入门书籍，虽然重点是讲解设计模式，但是也详细讲解了设计模式中涉及到的方方面面的基础知识，而且深入浅出，非常推荐入门的同学认真学习一下。

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：

- coolrandy 2019-05-23 01:18:11
  老师好 能不能后面讲一讲分布式锁相关的东西，比如实现方案，原理和场景之类的 [4赞]

- 缪文@有赞 2019-05-23 22:06:01
  毒丸对象，我也用过，就是一个可以通过外部接口或消息通知还写的bean，需要终止时设置为终止状态，不终止时是正常状态，消费线程在读到终止状态时直接跳过任务执行，线程也就完成终止了

- PJ ◕‿◕ 2019-05-23 10:34:25
  老师好 能不能后面讲一讲分布式锁相关的东西，比如实现方案，原理和场景之类的

  作者回复2019-05-23 20:59:53
  方案就是利用zk，redis，db，也可以用atomix这样的工具类自己做集群管理，网上有很多资料，最近实在太忙了😂😂😂

- 强哥 2019-05-23 09:33:10
  很期待接下来两个模块的深入讲解！

- 张三 2019-05-23 08:48:47
  打卡！