# 32 业务逻辑实现，游戏线程池如何设计？

更新时间：2020-08-24 10:37:40

> 世上无难事,只要肯登攀。——毛泽东

## 前言

你好，我是彤哥。

上一节，我们一起将所有的领域模型都实现了，但是，它们都是贫血模型，还缺少了非常重要的行为，本节，我们就一起来把它们的行为实现了，当然了，我们不会直接在这些模型内部实现行为，而是把它们抽离出来，具体的原因在上一节我们已经解释过了，此处，就不再赘述了。

好了，开始今天的学习吧。

## 游戏线程池的设计

在前面的章节中，我们一再提到，游戏的业务逻辑复杂，很多数据都是放在内存中进行处理的，但是，仅仅放在内存中还不能完全满足要求。

我举个例子，房间 A 里面的四个玩家打牌，如果使用普通的 Java 线程池处理这个房间的所有信息，就会带来新的问题：

1. 所有消息对于房间信息的修改，需要使用锁，对性能有一定的影响；
2. 使用锁之后，业务实现者要时时考虑锁的使用问题，增加了业务开发的难度；
3. 有序性，我们知道 Java 线程池中任务的流转过程，先看有没有达到核心线程数，再看队列有没有满，最后看有没有达到最大线程数，我举个极端的例子，如果队列满了，此时，发来一条消息，它会去尝试创建一个新的

线程（未达最大线程数）来处理当前消息，这导致该房间之前的消息可能还在队列中未处理，而这条消息却先处理了，这明显不符合要求。

基于以上三点原因，使用 Java 线程池肯定是不能满足我们的要求了，那么，使用 Netty 的线程池是否可以呢？

其实也是不行的，我们知道，Netty 线程池中每个 EventExecutor 都会绑定一个队列，不同客户端的消息是发往不同的队列，它依然会带来上面的三个问题。

所以，我们需要重新设计一种线程池：不加锁且消息有序，这要怎么实现呢？

我们前面提到过一个名词：房间制，那么，是不是把同一个房间的消息发到同一个线程进行处理就可以了呢？

答案确实是这样的，但是实现起来并没有那么简单。

首先，使用 Java 自带的线程池肯定是不行的了。

其次，使用 Netty 的线程池，通过前面关于 Netty 线程池的源码剖析，我们知道，在 Netty 中，有两种选择线程的方式，即 PowerOfTwoEventExecutorChooser 和 GenericEventExecutorChooser，然而，它们的内部实现并没有本质上的区别，都是通过轮询的方式调用线程池中的线程来处理任务，所以，乍看之下，也不满足我们的要求。

最后，只能自己来实现了？自己实现的话，也是每个线程维护自己的队列，且把同一个房间的消息都到同一个线程的队列中，这其实跟 Netty 还是有些像的，而且，自己实现的线程池，在稳定性、安全性等方面的考量，也是一个非常重要的问题，所以，我们能不能对 Netty 的线程池进行改造，拿过来即用呢？

通过前面的分析，可以发现，其实 Netty 的线程池本身是满足我们的要求的，关键在于选择线程的方式这里，所以，改造的要点就在这里，我们需要自己实现一种线程选择的方式，比如，按房间号去选择线程。

仔细观察 PowerOfTwoEventExecutorChooser 或者 GenericEventExecutorChooser 的代码：

```
private static final class PowerOfTwoEventExecutorChooser implements EventExecutorChooser {
    private final AtomicInteger idx = new AtomicInteger();
    private final EventExecutor[] executors;

    PowerOfTwoEventExecutorChooser(EventExecutor[] executors) {
        this.executors = executors;
    }

    @Override
    public EventExecutor next() {
        return executors[idx.getAndIncrement() & executors.length - 1];
    }
}
```

在调用 next () 方法选择线程的时候，似乎不能传递额外的参数了，但是，别忘了，我们还有终极杀器 —— 线程本地变量，在调用 next () 方法之前，可以把房间号存储在 FastThreadLocal 中，这样在 next () 中就能获取到这个房间号，然后，就可以使用房间号来做路由了，比如按房间号取模，完美 ^^

好了，说干就干，让我们来实现之～～

## 游戏线程池的实现

参考 DefaultEventExecutorChooserFactory 的代码，我们实现一个自己用的 MahjongEventExecutorChooserFactory，并将其运用到自定义的线程池 MahjongEventExecutorGroup 中：

```
public class MahjongEventExecutorGroup extends MultithreadEventExecutorGroup {

    private MahjongEventExecutorGroup(int nThreads) {
        // 使用自定义的选择器工厂
        super(nThreads, null, new MahjongEventExecutorChooserFactory(), null);
    }

    @Override
    protected EventExecutor newChild(Executor executor, Object... args) throws Exception {
        return new DefaultEventExecutor(this, executor);
    }
    // 工厂类
    private static class MahjongEventExecutorChooserFactory implements EventExecutorChooserFactory {

        @Override
        public EventExecutorChooser newChooser(EventExecutor[] executors) {
            return new MahjongEventExecutorChooser(executors);
        }
        // 真正使用的选择器
        private static class MahjongEventExecutorChooser implements EventExecutorChooserFactory.EventExecutorChooser {

            private final AtomicInteger idx = new AtomicInteger();
            private final EventExecutor[] executors;

            MahjongEventExecutorChooser(EventExecutor[] executors) {
                this.executors = executors;
            }

            @Override
            public EventExecutor next() {
                // 从上下文中取出当前的房间id
                Long roomId = MahjongContext.currentContext().getCurrentRoomId();
                long id;
                if (roomId != null) {
                    id = roomId;
                } else {
                    // 没获取到房间号的消息轮询扔到业务线程池中处理
                    // 他们往往跟房间信息没啥关系，比如登录请求
                    id = idx.getAndIncrement();
                }
                // 根据id取模选择线程执行
                return executors[(int) (id & executors.length - 1)];
            }
        }
    }
}
```

与默认的两种选择方式不同的是，在 next () 方法中，我们先尝试从上下文中获取当前的房间号，这里的上下文自然就是线程本地变量了，如果获取到了，就使用这个房间号取模来获取执行的线程，如果没获取到就使用原来的轮询方式进行处理，为什么会有没取到的情况呢？

比如，登录请求，它是没有房间信息的。

好了，现在的问题变成了：在什么时候把房间号放入到上下文中呢？

这就牵涉到要怎么使用这个线程池了，总体来说有两种方式：

1. 在 ChannelPipeline 中与 MahjongServerHandler 进行关联，即 `p.addLast(new MahjongEventExecutorGroup(8), new MahjongServerHandler());`；
2. 在 MahjongServerHandler 中手动调用，即 `mahjongEventExecutorGroup.execute()`；

使用第一种方式无疑是比较友好的，但是，整个过程都是在 ChannelPipeline 中自动完成的，没有地方可以设置房间号到上下文中，所以，只能使用第二种方式了。

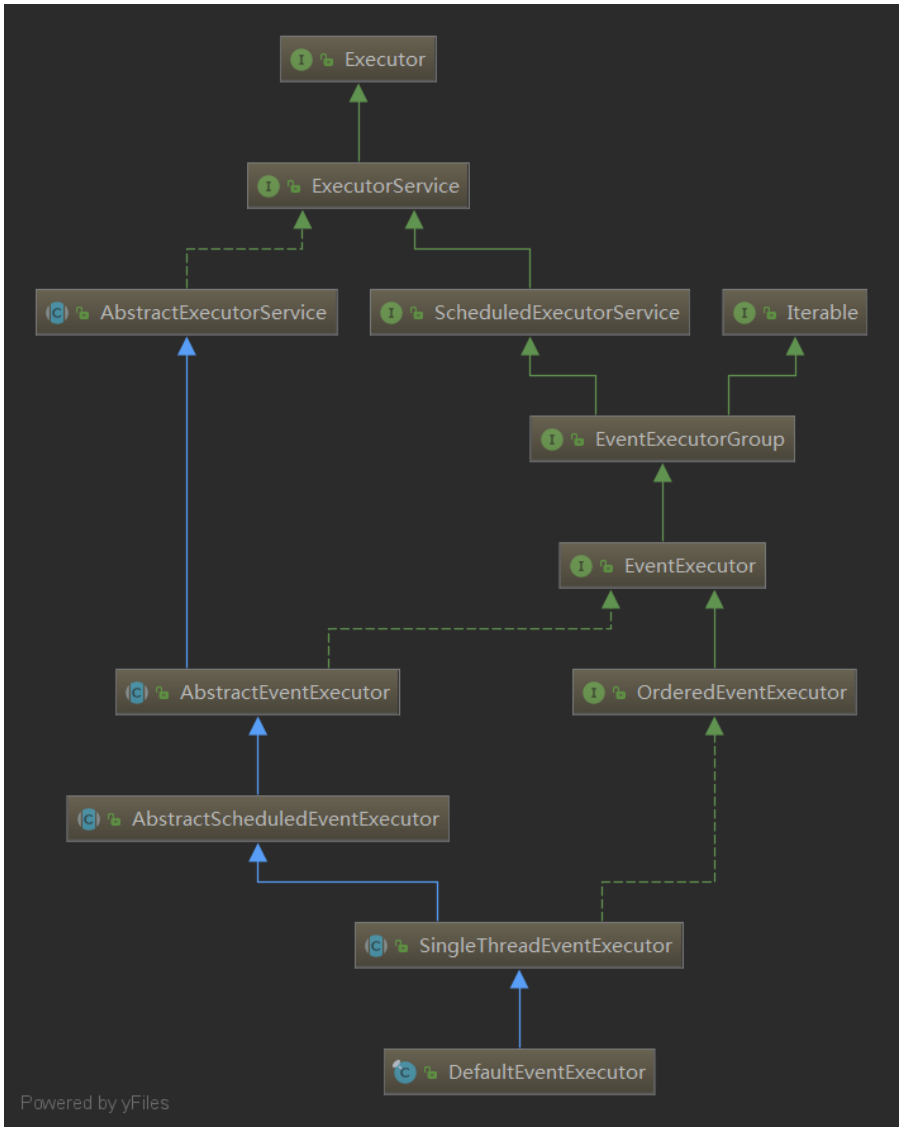
第二种方式就要简单得多了，只需要在 mahjongEventExecutorGroup.execute() 前一刻把房间号设置到上下文中即可。

为此，我专门在 MahjongEventExecutorGroup 中定义了一个静态方法来处理房间号相关的逻辑，给 MahjongServerHandler 调用：

```
public class MahjongEventExecutorGroup extends MultithreadEventExecutorGroup {
    // 实例，单例模式
    private static final MahjongEventExecutorGroup INSTANCE = new MahjongEventExecutorGroup(NettyRuntime.availableProcessors());

    // 静态方法，MahjongServerHandler中直接调用该方法
    public static void execute(Channel channel, MahjongProtocol mahjongProtocol) {
        // 协议头
        MahjongProtocolHeader header = mahjongProtocol.getHeader();
        // 协议体
        MahjongMessage message = (MahjongMessage) mahjongProtocol.getBody();
        // 创建房间和加入房间需要做一些处理
        Long roomId;
        if (message instanceof CreateRoomRequest) {
            // 如果是创建房间消息，生成一个roomId，并将当前channel与之绑定
            roomId = IdUtils.generateId();
            // 一个channel一旦建立，始终与一个eventLoop绑定
            // 所以，可以把channel与房间号的绑定放在线程本地缓存中
            MahjongContext.currentContext().setChannelRoomId(channel, roomId);
        } else if (message instanceof EnterRoomRequest) {
            // 如果是加入房间消息，将当前channel与传入的tableId绑定
            EnterRoomRequest enterRoomRequest = (EnterRoomRequest) message;
            roomId = enterRoomRequest.getRoomId();
            MahjongContext.currentContext().setChannelRoomId(channel, roomId);
        } else {
            // 其它消息则从context中获取当前channel对应的房间号
            // 当然，也可能没有，比如登录请求
            roomId = MahjongContext.currentContext().getChannelRoomId(channel);
        }

        // 设置房间id到context中，以便next()方法可以取到
        MahjongContext.currentContext().setCurrentRoomId(roomId);
        // 将消息扔到业务线程池中处理
        INSTANCE.execute(() -> {
            // todo 此处实现在下面
        });

    }
}
```

到此，我们真正实现了根据房间号来选择指定线程的目标，那么，还有一个问题：这个线程池能否保证有序性呢？

答案是肯定的，请看下面这张图：




这里有个非常重要的接口 ——OrderedEventExecutor，从名字就可以知道，它就是用来保证有序性的。

另外，根据前面 Netty 线程池源码的剖析，我们也知道，所有的任务提交的时候是第一时间放到队列中的：

```
private void execute(Runnable task, boolean immediate) {
    boolean inEventLoop = inEventLoop();
    addTask(task);
    if (!inEventLoop) {
        startThread();
        // ...省略其它代码
    }
}
```

而不是像 Java 那样，把提交的任务与新创建的线程绑定，优先执行这个任务：

```
Worker(Runnable firstTask) {
    setState(-1); // inhibit interrupts until runWorker
    this.firstTask = firstTask;
    this.thread = getThreadFactory().newThread(this);
}
final void runWorker(Worker w) {
    // 优先执行firstTask
    Runnable task = w.firstTask;
    try {
        while (task != null || (task = getTask()) != null) {
            try {
                beforeExecute(wt, task);
                try {
                    task.run();
                } finally {
                    afterExecute(task, thrown);
                }
            } finally {
            }
        }
    } finally {
        processWorkerExit(w, completedAbruptly);
    }
}
```

所以，使用 Netty 线程池是完全可以保证任务有序执行的，而使用 Java 线程池则不行。

好了，通过这样的线程池实现，在后续的业务处理中，完全不用考虑锁的问题了，因为同一个房间的消息都在同一个线程中有序的执行，完全不需要使用锁，极大地减轻了业务开发的难度，下面我们就来愉快地实现业务逻辑吧。

## 业务逻辑实现

为了把代码足够地解耦，我定义了一个 MahjongProcessor 接口，每个消息都可以实现自己的处理器：

```
public interface MahjongProcessor<T extends MahjongMessage> {
    void process(T message);
}
```

比如，以 HelloRequest 为例：

```
@Slf4j
public class HelloRequestProcessor implements MahjongProcessor<HelloRequest> {

    @Override
    public void process(HelloRequest message) {
        log.info("receive hello request: {}", message);
        HelloResponse response = new HelloResponse();
        response.setMessage("你好，" + message.getName());
        MessageUtils.sendResponse(response);
    }
}
```

有了前面的铺垫，每个消息处理器里面只需要处理自己的消息即可，非常简单。

同样地，我们需要维护消息与处理器之间的映射，与前面消息与 cmd 的映射一样，我这里同样使用枚举的形式来实现：

```
public enum MahjongProcessorManager {
    // 映射关系
    HELLO_REQUEST_PROCESSOR(HelloRequest.class, new HelloRequestProcessor()),
    LOGIN_REQUEST_PROCESSOR(LoginRequest.class, new LoginRequestProcessor()),
    CREATE_ROOM_REQUEST_PROCESSOR(CreateRoomRequest.class, new CreateRoomRequestProcessor()),
    ENTER_ROOM_REQUEST_PROCESSOR(EnterRoomRequest.class, new EnterRoomRequestProcessor()),
    OPERATION_REQUEST_REQUEST_PROCESSOR(OperationRequest.class, new OperationRequestProcessor()),
    START_GAME_MESSAGE_PROCESSOR(StartGameMessage.class, new StartGameMessageProcessor()),
    ;

    // 属性
    private Class<? extends MahjongMessage> msgType;
    private MahjongProcessor mahjongProcessor;

    MahjongProcessorManager(Class<? extends MahjongMessage> msgType, MahjongProcessor mahjongProcessor) {
        this.msgType = msgType;
        this.mahjongProcessor = mahjongProcessor;
    }

    // 根据消息选择处理器
    public static MahjongProcessor choose(MahjongMessage message) {
        for (MahjongProcessorManager value : MahjongProcessorManager.values()) {
            if (value.msgType == message.getClass()) {
                return value.mahjongProcessor;
            }
        }
        return null;
    }

}
```

可以看到，对于 MahjongProcessor 我们维护的是一个一个的实例，而不再是 Class 类，因为 MahjongProcessor 是设计成无状态的，整个系统只需要一个实例，即单例。

OK，此时，我们就可以在 MahjongEventExecutorGroup 中补全 execute () 的逻辑了：

```
public class MahjongEventExecutorGroup extends MultithreadEventExecutorGroup {
    public static void execute(Channel channel, MahjongProtocol mahjongProtocol) {
        // 协议头
        MahjongProtocolHeader header = mahjongProtocol.getHeader();
        // 协议体
        MahjongMessage message = (MahjongMessage) mahjongProtocol.getBody();
        // 创建房间和加入房间需要做一些处理
        Long roomId;
        // ...省略roomId的处理

        // 设置房间id到context中，以便next()方法可以取到
        MahjongContext.currentContext().setCurrentRoomId(roomId);
        // 将消息扔到业务线程池中处理
        INSTANCE.execute(() -> {
            // 已经切换线程，重新设置房间号到context中
            MahjongContext.currentContext().setCurrentRoomId(roomId);
            // 设置channel等其它线程本地变量
            MahjongContext.currentContext().setCurrentChannel(channel);
            MahjongContext.currentContext().setChannelRoomId(channel, roomId);
            MahjongContext.currentContext().setRequestHeader(header);
            MahjongContext.currentContext().setCurrentRoom(MahjongContext.currentContext().getRoomById(roomId));

            Player currentPlayer = DataManager.getChannelPlayer(channel);
            if (currentPlayer != null) {
                MahjongContext.currentContext().setCurrentPlayer(currentPlayer);
                MahjongContext.currentContext().setPlayerChannel(currentPlayer, channel);
            }
            // 寻找处理器
            MahjongProcessor processor = MahjongProcessorManager.choose(message);
            if (processor != null) {
                // 交给处理器处理
                processor.process(message);
            } else {
                throw new RuntimeException("not found processor, msgType=" + message.getClass().getName());
            }
        });

    }
}
```

可以看到，我们这里大量使用了 MahjongContext，它其实就是一个线程本地变量池，怎么实现的呢？我提供两种思路：

1. 把每一个本地变量，比如 currentChannel，都声明为一个 ThreadLocal 或者 FastThreadLocal；
2. 把 MahjongContext 声明为一个 ThreadLocal 或者 FastThreadLocal，其它的变量都维护在这个 MahjongContext 中；

两种方式都是可以的，第一种方式实现起来有点繁琐，第二种方式实现起来相对简单一点，我参考 zuul 网关等框架采用了第二种实现方式：

```
public class MahjongContext {
    // 线程安全，只需要用HashMap就可以了
    private final Map<String, Object> map = new HashMap<>();
    // 把MahjongContext本身放到FastThreadLocal中
    private static final FastThreadLocal<MahjongContext> MAHJONG_CONTEXT = new FastThreadLocal<MahjongContext>() {
        @Override
        protected MahjongContext initialValue() throws Exception {
            return new MahjongContext();
        }
    };

    private MahjongContext() {
    }
    // 当前线程的MahjongContext
    public static MahjongContext currentContext() {
        return MAHJONG_CONTEXT.get();
    }
    // 包装map的put()方法
    private void set(String key, Object value) {
        map.put(key, value);
    }
    // 包装map的get()方法
    private <T> T get(String key) {
        return (T) map.get(key);
    }
    // 包装map的remove()方法
    private void remove(String key) {
        map.remove(key);
    }
    // 设置当前玩家的channel
    public void setCurrentChannel(Channel channel) {
        set("currentChannel", channel);
    }
    // 获取当前玩家的channel
    public Channel getCurrentChannel() {
        return get("currentChannel");
    }
}
```

好了，最后，我再介绍一个稍微复杂一点的消息处理逻辑 —— 出牌。

出牌是通过 OperationRequest 这个消息来承载的，我们先分析一下它的主要逻辑：

1. 服务端收到出牌消息，先检查是不是轮到该玩家出牌了，出的这张牌玩家手里有没有，等等；
2. 从玩家手中移除这张牌，并添加到添加到出牌列表中；
3. 通知所有人谁出了什么牌，并刷新房间的信息；
4. 检查其他玩家是否可以操作；
5. 如果有玩家可以操作，则提醒其操作，并提醒其他玩家等待；
6. 如果没有玩家可以操作，则光标移动到下一个玩家；
7. 下一个玩家摸牌；
8. 检查下一个玩家摸完牌后是否可胡、可杠；
9. 如果其可胡、可杠，则提醒其操作；
10. 如果不可胡、可杠，则提醒其出牌；
11. 不管其他玩家是否可胡、可杠，对于其他玩家一律提示等待该玩家出牌；

OK，这里只列了主要逻辑，还有一些细节的点需要注意的，我就不一一列举了，这里只贴出一小部分的代码实现：

```
@Slf4j
```

```
public class OperationRequestProcessor implements MahjongProcessor<OperationRequest> {
    @Override
    public void process(OperationRequest message) {
        // 获取当前房间信息
        Room room = MahjongContext.currentContext().getCurrentRoom();
        // 当前操作
        int operation = message.getOperation();

        // 检查
        if (room == null) {
            log.error("room not exist");
            return;
        }

        if (room.getStatus() != Room.STATUS_WAITING_CHU && room.getStatus() != Room.STATUS_WAITING_OPERATION) {
            log.error("room status error, roomStatus={}", room.getStatus());
            return;
        }

        if (room.getStatus() == Room.STATUS_WAITING_CHU && operation != OperationUtils.OPERATION_CHU) {
            log.error("operation request error, roomStatus={}, operation={}", room.getStatus(), operation);
            return;
        }

        if (room.getStatus() == Room.STATUS_WAITING_OPERATION && operation == OperationUtils.OPERATION_CHU) {
            log.error("operation request error, roomStatus={}, operation={}", room.getStatus(), operation);
            return;
        }

        if (operation == OperationUtils.OPERATION_CHU) {
            // 如果是出牌操作
            chu(room, message);
        } else {
            // 如果是其他操作，需等待所有可操作之人操作完成之后才能判断谁的操作是有效的
            operate(room, message);
        }
    }

    private void chu(Room room, OperationRequest message) {
        // 当前玩家
        Player player = MahjongContext.currentContext().getCurrentPlayer();
        // 检查是不是当前玩家出牌
        if (room.getChuPos() != player.getPos()) {
            log.error("not current player chu, roomChuPos={}, currentPlayerPos={}", room.getChuPos(), player.getPos());
            return;
        }

        byte chuCard = message.getCard();

        // 从玩家手中移除该牌
        if (!removeCard(player, chuCard)) {
            log.error("player has not that card, playerPos={}, chuCard={}", player.getPos(), chuCard);
            return;
        }

        // 添加到出牌列表中
        addToChuCards(player, chuCard);

        // 通知有人出了一张牌
        OperationResultNotification operationResultNotification = new OperationResultNotification();
        operationResultNotification.setOperation(OperationUtils.OPERATION_CHU);
        operationResultNotification.setPos(player.getPos());
        operationResultNotification.setCard(chuCard);
        MessageUtils.sendNotification(room, operationResultNotification);

        // 刷新房间信息牌
        RoomRefreshNotification refreshNotification = new RoomRefreshNotification();
        refreshNotification.setOperation(OperationUtils.OPERATION_CHU);
        refreshNotification.setRoom(room);
```

```
        MessageUtils.sendRoomRefreshNotification(refreshNotification);

        // 检查其他玩家是否可以操作
        if (!checkOtherCanOperation(room, player, chuCard)) {
            // 如果其他玩家不可以操作，移动到下一个玩家摸牌
            moveToNextPlayer(room);
        }
    }
}
```

抛却业务本身逻辑的复杂性，代码写起来是不是非常简单，完全不用关心锁的问题，完美 ^^

OK，到这里，游戏线程池的设计、业务逻辑实现的方法我就介绍完了。

## 后记

本节，我们一起学习了“房间制”游戏的线程池设计，以及业务逻辑的实现方法，你觉得这种实现方式怎么样？还有没有其它更优的解法呢？欢迎留言，共同讨论。

到此，整个服务端算是全部实现完毕了，可是，没有客户端的服务端是不完美的，所以，下一节，我们将模拟实现一个客户端，并通过这个客户端来真正的打一局，敬请期待。

## 思维导图


}

31 领域模型实现，我又贫血了

33 Mock客户端实现，四个人来一局