31 领域模型实现，我又贫血了

更新时间：2020-08-21 10:09:24

> 从不浪费时间的人，没有工夫抱怨时间不够。——杰弗逊

前言

你好，我是彤哥。

上一节，我们一起把实战项目的协议给实现了，同时，通过 HelloRequest 和 HelloResponse 把客户端和服务端成功连接起来了。

本节，我们将根据系统设计中领域模型设计的内容，把这些领域模型给实现了。

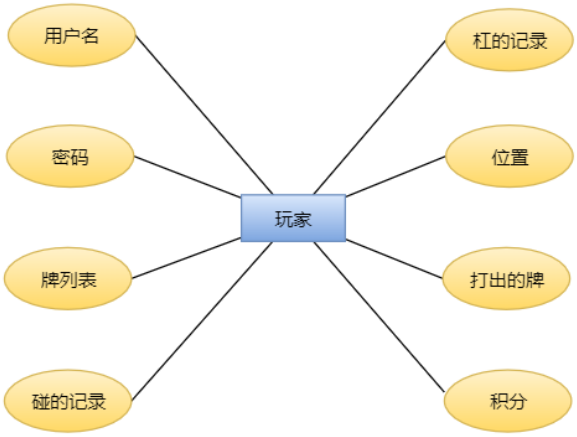
下面就进入今天的学习吧。

领域模型实现

根据系统设计一节的内容，我们归纳出来的领域模型有：玩家、房间、牌、各种消息。

> 我这里把消息也看作是领域模型了，更确切地讲，消息属于接口设计的内容，不过为了方便讲解，我是把他们合并在一起了。怎么区分两者呢？看主体本身是不是一个纯粹的名词，比如，玩家就是一个纯粹的名词，所以，它是领域模型，而出牌请求就不是一个纯粹的名词，它包含动词含义，所以，它是接口。不过，总体来说，它们都是一个主体加上一些属性构成，所以，看成是一样的东西也是没问题的。

玩家



所谓实现，就是把设计的内容转换成代码，所以，编码是最简单的工作。当然了，在实现的时候可能也会扩充一些字段，比如，玩家应当具有唯一的 ID，所以，上面这张图转换成代码如下：

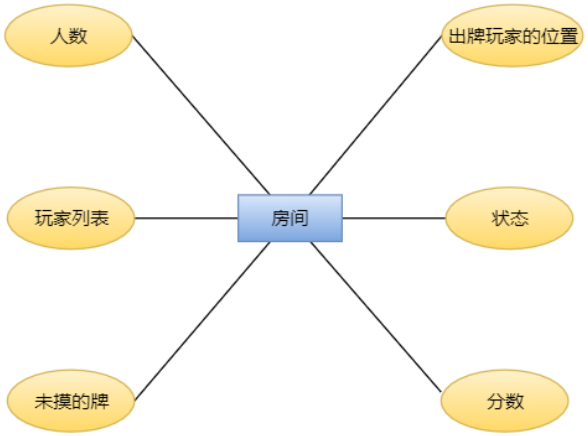
```
@Data
public class Player {
    /**
     * 唯一标识id
     */
    private long id;
    /**
     * 用户名
     */
    private String username;
    /**
     * 密码
     */
    private String password;
    /**
     * 玩家的积分
     */
    private int score;

    /**
     * 玩家在房间的位置
     */
    private int pos;
    /**
     * 手牌列表
     */
    private byte[] cards;
    /**
     * 打出的牌
     */
    private byte[] chuCards;
    /**
     * 碰的牌
     */
    private byte[] pengList;
    /**
     * 杠的牌
     */
    private byte[] gangList;

}
```

我这里做了一些调整，将玩家天然的属性移到了上面，将牌局相关的信息移到了下面。

为什么要这么做呢？

因为，牌局相关的信息在每一局都不一样，所以，需要在牌局结束时进行重置，把牌局相关的信息放在一起，到时候写重置方法的时候会比较简单清晰一些。

房间



同样地，房间也会扩充一些属性：

```
@Data
public class Room {
    /**
     * 房间id
     */
    private long id;
    /**
     * 房间最大的人数
     */
    private int maxPlayerNum;
    /**
     * 底分
     */
    private int baseScore;
    /**
     * 房间内的玩家列表
     */
    private Player[] players;
    /**
     * 未摸的牌
     */
    private byte[] remainCards;
    /**
     * 出牌玩家的位置
     */
    private int chuPos;
    /**
     * 状态
     */
    private int status;
}
```

对于房间的状态，我们也是要定义的，这一块在系统设计的时候给遗漏了，在这里补上。

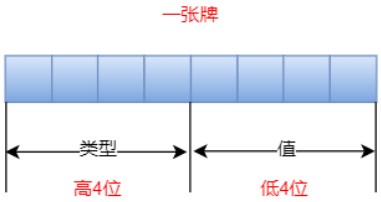
首先，一个玩家创建了房间之后，房间的状态进入等待其他玩家进入的状态，当玩家满了的情况下，游戏开始了，记录一个游戏开始的状态，发完牌了，轮到庄家出牌，记录一个等待出牌的状态，出牌后如果有其它玩家可以操作，还需要记录一个等待操作的状态，最后，游戏结束了，记录一个结束的状态。

所以，房间的状态一共有：等待玩家、游戏开始、等待出牌、等待操作、游戏结束，简单点，我们把这些常量也定义到房间这个类中：

```
public class Room {
    /**
     * 等待其他玩家进入房间
     */
    public static final int STATUS_WAITING_PLAYER = 1;
    /**
     * 游戏开始
     */
    public static final int STATUS_GAME_STARTING = 2;
    /**
     * 等待玩家出牌
     */
    public static final int STATUS_WAITING_CHU = 3;
    /**
     * 等待其他玩家操作（碰、杠、胡）
     */
    public static final int STATUS_WAITING_OPERATION = 4;
    /**
     * 游戏结束
     */
    public static final int STATUS_GAME_OVER = 5;
}
```

牌

按照系统设计的一节的内容，一张牌用一个 byte 表示就可以的，所以，牌不需要单独的类来表示，但是，需要一个工具类告诉我哪个 byte 表示哪张牌，因此，我们这里也定义一下牌的信息：



这样设计的话，类型占 4 位，牌值点 4 位，各能代表 16 种不同的数值，是完全足够的，所以，我们的工具类看起来是这样的：

```
public class CardUtils {
    /**
     * 万的掩码，一万到九万：0x11~0x19
     */
    public static final byte CARD_TYPE_WAN_MASK = 0x10;
    /**
     * 条的掩码，一条到九条：0x21~0x29
     */
    public static final byte CARD_TYPE_TIAO_MASK = 0x20;
    /**
     * 筒的掩码，一筒到九筒：0x31~0x39
     */
    public static final byte CARD_TYPE_TONG_MASK = 0x30;

    /**
     * 类型的掩码
     */
    public static final byte CARD_TYPE_MASK = 0x70;
    /**
     * 值的掩码
     */
    public static final byte CARD_VALUE_MASK = 0x0f;

    /**
     * 获取牌的类型
     * @param card
     * @return
     */
    public static final byte cardType(byte card) {
        return (byte) (CARD_TYPE_MASK & card);
    }

    /**
     * 获取牌的数值
     * @param card
     * @return
     */
    public static final byte cardValue(byte card) {
        return (byte) (CARD_VALUE_MASK & card);
    }

}
```

这里使用 16 进制表示，正好一个数字表示 4 位，同学们好好体会一下。

对于万牌来说，它的值为 0x11~0x19，分别表示一万到九万。

消息

在系统设计一节中，我们归纳出来的消息一共有：登录、创建房间、加入房间、房间刷新通知、操作通知、操作请求、操作结果通知、结束通知、结算通知。

它们都比较简单，我们快速的定义一下：

```
@Data
public class LoginRequest implements MahjongMessage {
    private String username;
    private String password;
}
@Data
public class LoginResponse implements MahjongMessage {
    private boolean result;
    private Player player;
    private String message;
}
@Data
public class CreateRoomRequest implements MahjongMessage {
    private int playerNum;
    private int baseScore;
}
@Data
public class CreateRoomResponse implements MahjongMessage {
    private boolean result;
    private String message;
}
@Data
public class EnterRoomRequest implements MahjongMessage {
    private long tableId;
}
@Data
public class EnterRoomResponse implements MahjongMessage {
    private boolean result;
    private String message;
}
@Data
public class RoomRefreshNotification implements MahjongMessage {
    private int operation;
    private Room room;
}
@Data
public class OperationNotification implements MahjongMessage {
    private int operation;
    private int pos;
    private byte fireCard;
}
@Data
public class OperationRequest implements MahjongMessage {
    private int operation;
    private int pos;
    private byte card;
}
@Data
public class OperationResultNotification implements MahjongMessage {
    private int operation;
    private int pos;
    private byte card;
}
@Data
public class GameOverNotificaion implements MahjongMessage {
    private Room room;
}
@Data
public class SettlementNotification implements MahjongMessage {
    private int[] scores;
}
```

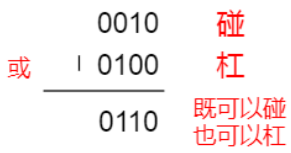
另外，要记得把它们都添加到 MessageManager 里面：

```
public enum MessageManager {
    HELLO_REQUEST(1, HelloRequest.class),
    HELLO_RESPONSE(2, HelloResponse.class),
    LOGIN_REQUEST(3, LoginRequest.class),
    LOGIN_RESPONSE(4, LoginResponse.class),
    CREATE_ROOM_REQUEST(5, CreateRoomRequest.class),
    CREATE_ROOM_RESPONSE(6, CreateRoomResponse.class),
    ENTER_ROOM_REQUEST(7, EnterRoomRequest.class),
    ENTER_ROOM_RESPONSE(8, EnterRoomResponse.class),
    ROOM_REFRESH_NOTIFICATION(9, RoomRefreshNotification.class),
    OPERATION_NOTIFICATION(10, OperationNotification.class),
    OPERATION_REQUEST(11, OperationRequest.class),
    OPERATION_RESULT_NOTIFICATION(12, OperationResultNotification.class),
    GAME_OVER_NOTIFICATION(13, GameOverNotification.class),
    SETTLEMENT_NOTIFICATION(14, SettlementNotification.class),
    ;
}
```

关于操作相关的消息，我们还应该定义有哪些操作类型，所以，我们再新加一个工具类用来定义操作类型：

```
public class OperationUtils {
    /**
     * 出
     */
    public static final int OPERATION_CHU = 1;
    /**
     * 碰
     */
    public static final int OPERATION_PENG = 2;
    /**
     * 杠
     */
    public static final int OPERATION_GANG = 4;
    /**
     * 胡
     */
    public static final int OPERATION_HU = 8;
    /**
     * 摸
     */
    public static final int OPERATION_GRAB = 16;
    /**
     * 过
     */
    public static final int OPERATION_GUO = 32;
}
```

细心的同学会发现，操作类型的这些数值转换成二进制正好是错开的，这也是为了能在一个 int 字段中表示多个操作类型，比如，玩家 A 出了一张牌，玩家 B 手中有三张，那么，他既可以选择碰，也可以选择杠，此时，我们只需要把 OPERATION_PENG 和 OPERATION_GANG 做一个 “或” 操作就可以了。



好了，所有的领域模型都实现了，细心的同学可能会发现，所有的领域模型中都没有行为，也就是方法，这也就是传说中的贫血模型，为什么要使用贫血模型呢？

原因主要有以下几点：

1. 我们定义的这些领域模型是要给到客户端的，里面定义了方法，这些方法可能并不适用于客户端，造成冗余；
2. 目前这些领域模型是手动编写的，试想如果后期扩展为 protobuf，它们是通过文件生成的，这些生成的类中是不应该添加自己的方法的；
3. 如果后期要做持久化到数据库，这个持久化的动作放在哪里比较合适，放在领域模型中会很奇怪；
4. 充血模型很难定义业务逻辑的边界，也就是说哪些方法应该放在领域模型中，哪些应该放在 service 中，很难把控，所以，倒不如把这些业务逻辑全部放在 service 中，更加清晰；
5. 开发人员的技术水平参差不齐，使用充血模型会导致上面所说的业务边界的问题更加混乱；

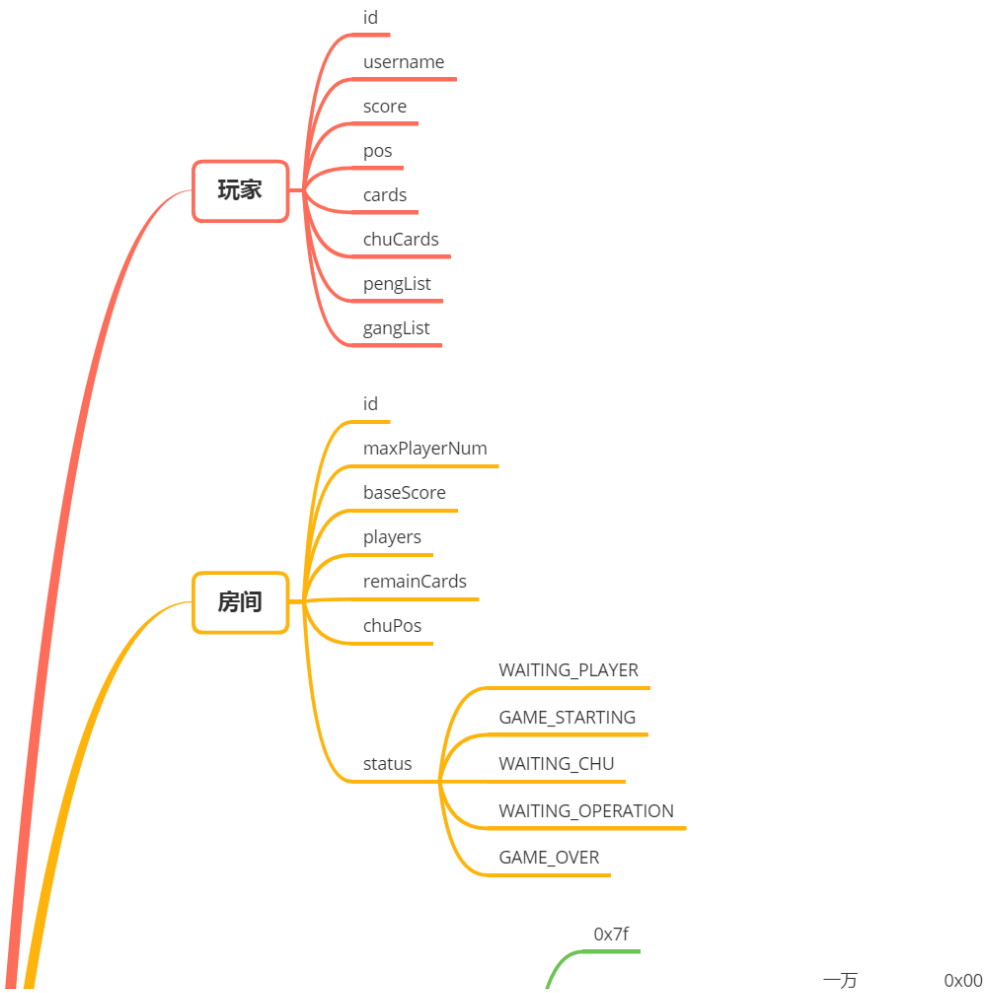
综上所述，虽然使用贫血模型不那么的面向对象（当然，一般程序员可能没有对象），但是，业务逻辑和领域模型的边界都足够清晰，更适合于团队开发，因此，我们这里也同样采用贫血模型来进行开发，这可能也是 Spring MVC 比较流行的主要原因。

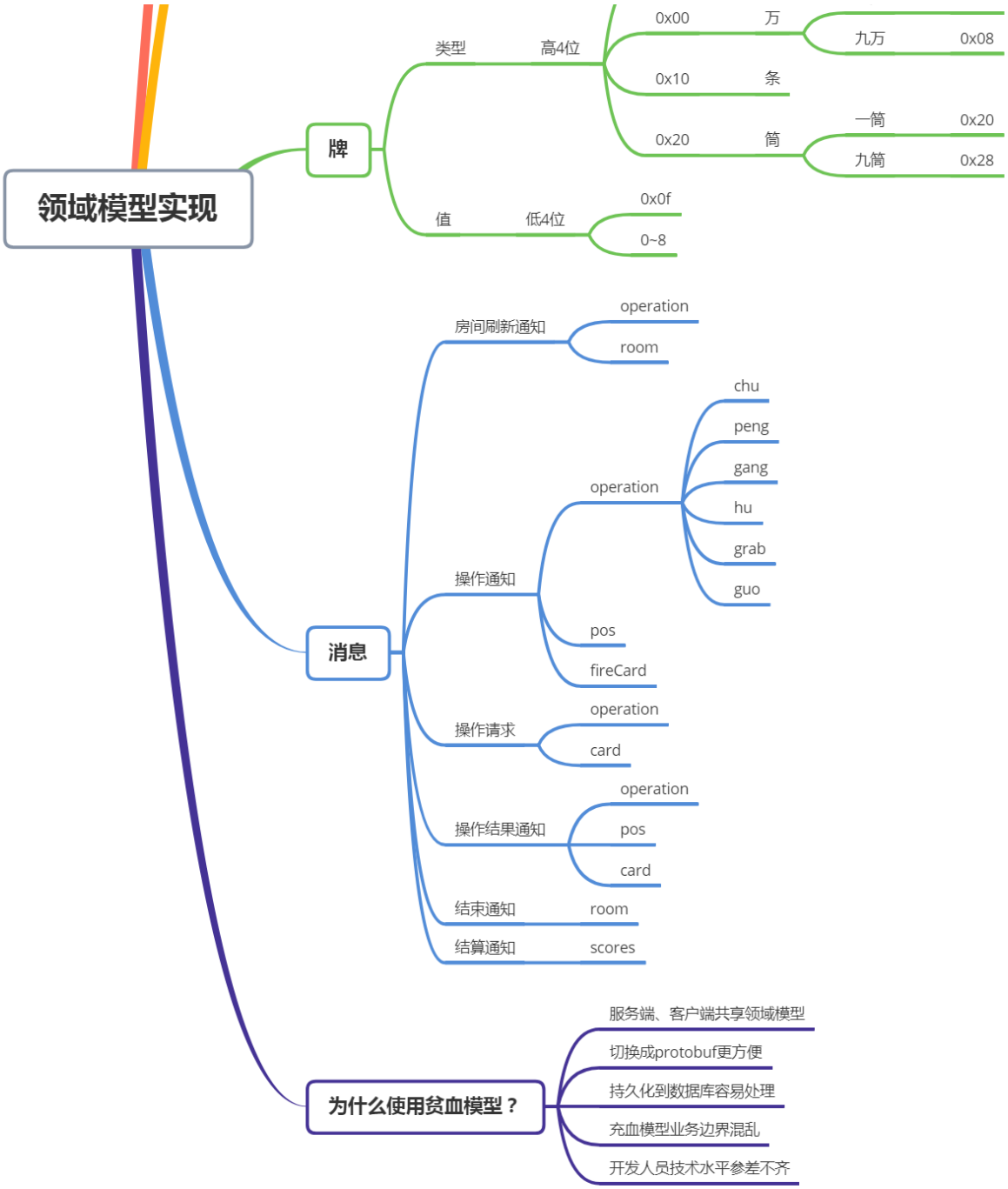
后记

本节，我们根据系统设计中领域模型设计以及接口设计的内容把所有的领域模型给实现了，并在最后，分析了我们使用贫血模型的主要原因。

下一节，我们将给这些贫血的模型加上业务处理的逻辑，当然了，这些逻辑是剥离出去的，同时，会考虑使用什么样的线程池来承载我们的业务处理逻辑，敬请期待。

思维导图





}

30 协议实现，双端打通

32 业务逻辑实现，游戏线程池如何设计？