Server：

server是vs工程，需要另编译一个boost1.52

文件结构

base：底层、socket、收发消息的队列，略

DiceGameServer：游戏逻辑

protocol：不用看，略

tinyxml：读xml配置文件，略

DiceGameServer：

DiceGameServer：起各种服务

GameManager：房间线程管理

UserSessionManager: 通信线程管理

Game：游戏基类

GameGrail： 星杯游戏类

GameGrailCommon: 星杯各种共用的函数、错误代号

GrailState： 星杯状态类

PlayerEntity： 角色基类

CardEntity： 牌类

UserTask： 通信线程类

代码结构：

一局游戏涉及：一个游戏线程GameGrail，n个通信线程（给的代码，n=4）

GameGrail通过sendMessage可直接要UserTask发包

UserTask收到的消息在cmdMsgParse里，需要通过tryNotify才能跟GameGrail交互，tryNotify会检测用户ID、状态号（state）、step是否正确，正确才能干涉游戏（详见后）

推倒重做后，用了状态机把游戏流程断成一个个状态。因为星杯的插入结算，会维护一个状态栈std::stack。 栈内放的是GrailState的指针。栈操作就不细说了pushGameState, popGameState, popGameState\_if。

class GrailState

{

public:

int state;

int step;

int iterator;

GrailState(int s): state(s), step(0), iterator(0) {}

void moveIterator(int ret);

virtual int handle(GameGrail\* engine) { return GE\_EMPTY\_HANDLE; }

};

GrailState是所有状态类的基类

三个成员变量：

state是状态编号，step是子状态编号，iterator是遍历角色类时的游标（详见后）

两个成员函数：

handle，是virtual函数，设计上所有子类都要实现这个函数，否则会返回错误代码GE\_EMPTY\_HANDLE（详见后）

moveIterator, 顾名思义就是移动游标的（详见后）

游戏的主函数是GameGrail：：GameRun（），GameRun很简单就是从状态栈里取出栈顶元素，并执行他的handle函数，此时handle函数指向哪个函数取决于currentState的实际类型

while(true)

{

ret = GE\_NO\_STATE;

currentState = topGameState();

if(currentState){

ret = currentState->handle(this);

}

}

有的GrailState会刷一遍所有角色类的对应函数，这些GrailState一般对应星杯中的一个响应点

如回合开始前

int StateBeforeTurnBegin::handle(GameGrail\* engine)

{

for(int i = iterator; i< engine->getGameMaxPlayers(); i++){

while(iterator < engine->getGameMaxPlayers()){

ret = engine->getPlayerEntity(iterator)->p\_before\_turn\_begin(step, m\_currentPlayerID);

moveIterator(ret);

if(GE\_SUCCESS != ret){

return ret;

}

}

engine->popGameState()

engine->pushGameState(new StateTurnBegin);

return ret;

}

可以看到handle里有个循环，循环里调用了playerEntity的p\_before\_turn\_begin。p\_before\_turn\_begin也是virtual函数，但不同的是，这里并不要求每个子类都实现他，实际上只有天使需要覆写它，其余的默认返回成功代号

virtual int p\_before\_turn\_begin(int currentPlayerID) { return GE\_SUCCESS; }

一般情况iterator没有意义，但当p\_before\_turn\_begin内部插入了新的状态时，就会有问题。因为接下来的代码就把你插入的状态给pop出来了（当然我这里还加了层保险pop\_if）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| StateBeforeTurnBegin | push(StateNew) | StateBeforeTurnBegin | pop() | StateBeforeTurnBegin |
|  | StateNew |  |

一定要保持一个意识，你**在handle里push的状态**在handle返回前是不会执行的。那如果一定要立刻执行呢，看代码就知道ret不等于GE\_SUCCESS就好了，**统一返回GE\_URGENT**。

不过，这又分两种情况，当前playerEntity的p\_before\_turn\_begin搞定没有（还有子状态？），如果搞定了请返回GE\_ URGENT，并把step设为STEP\_DONE（iterator加一，下次跳过你这个角色），否则返回GE\_URGENT（iterator不加一，下次还是你）

State里还有子状态step，每个iterator开始时，step会被初始化为STEP\_INIT。当角色在一个时间点里有多于1个的技能可触发时，step就是用来区分当前进行到哪一步，**务必在走完后，把step设为STEP\_DONE。**

int JianSheng::p\_after\_attack(int &step, int playerID)

{

int ret = GE\_INVALID\_STEP;

if(playerID != id){

return GE\_SUCCESS;

}

while(STEP\_DONE != step)

{

switch(step)

{

case STEP\_INIT:

step = LIAN\_XU\_JI;

break;

case LIAN\_XU\_JI:

ret = LianXuJi(playerID);

if(toNextStep(ret)){

step = JIAN\_YING;

}

break;

case JIAN\_YING:

ret = JianYing(playerID);

if(toNextStep(ret)){

step = STEP\_DONE;

}

break;

default:

return GE\_INVALID\_STEP;

}

}

return ret;

}

还有的GrailState需要玩家的响应

如应战

int StateAttacked::handle(GameGrail\* engine)

{

if(engine->waitForOne(context->attack.dstID, Coder::askForReBat(context->hitRate, context->attack.cardID, context->attack.dstID, context->attack.srcID)))

{// 处理用户的回应

}

else{

//超时

return GE\_TIMEOUT;

}

}

这时会调用waitForOne或者waitForAll，游戏线程被挂起，等待notify唤醒。wait内部自带超时多少次统计，并且每次超时都会重发消息包（以防丢包），当达到超时次数上限时wait返回false；期间被唤醒的话，返回true

大多数state超时会执行默认操作，响应类的当不响应，应战的当不应战；而有些state不会处理超时，如行动阶段，次数会多次进入该状态，重新统计超时

UserTask通过tryNotify尝试唤醒GameGrail，调用时要附带玩家的id，跟声明的当前状态。GameGrail会检测id是否等于token（它在等待的人），声明的状态是否是当前状态，step是不是当前的step。只有都通过才真正被notify。这时可通过getReply获取玩家的回应。

**最后所有涉及push新状态的API都会以SetState开头，调用这些API时，请确保知道你在干什么，不然很可能GameGrail就不知道跑到哪里去了**

召唤宝典：

写角色以实现callback为主。

1. 先分析技能，确定触发点，触发点对应StateXXX
2. 然后你要实现的是相应的p\_xxx，一个p\_xxx里可能不只一个技能，这时候你要利用好step（详见上）
3. 一旦你在p\_xxx里**使用了waitForOne或者waitForAll（应该没有后面这种情况的）。你需要自己唤醒游戏线程**。但是不要直接改UserTask，UserTask会调用角色的cmdMsgParse。所以你应该在那里，调用tryNotify，并填上正确的三元组（id，state，step）
4. 再者有些角色**不遵循一般的游戏规则，这时你要覆写v\_xxx**。v\_xxx跟p\_xxx不同，它是用来做验证的。基本通用的验证已经实现，但对于有些角色并不适用。如连续技不适用一般的v\_attack，它应该只接受风系攻击；魔剑他回合内不能用法术，于是v\_weaken之类的都不应该返回true。
5. protobuf可以看action\_response.proto，找对应的变量，尽量不要用args，维护起来很麻烦，常用的会有Coder帮你初始化
6. 当涉及移动牌的时候，分两种情况：移到手里，跟移到其他地方。移到手里的要**填好相应的HARM**，以后可能会用做统计，**不要直接调用setStateMoveCards**
7. 把回合变量的初始化放在p\_before\_turn\_begin
8. 启动是boot
9. **技能代号**在**在GrailCommon.h的cause**里
10. 关于debug，一般出了不能恢复的错误的话，服务器会狂刷错误信息，很难发现不了。错误信息包括**state、errorCode**，可以**在GrailCommon.h里**找到对应枚举值，帮助定位出错的地方。