第4章 底层通信

- 信息解析
- 动作封装和发送
- 球员智能体的通信协议

4.0 Connection类

- 消息解析主要是由SenseHandler类来进行的，
- 动作的发送主要是由ActHandler类来完成的
- 底层Class Connection
  - 基于socket的通讯
  - 信息的接受

4.0 Connection类

- typedef struct _socket{
  - int socketfd ;
  - struct sockaddr_in serv_addr ;
- } Socket ;

4.0 Connection类

- class Connection {
- Socket m_sock;
- int m_iMaxMsgSize;
- Connection ( );
- Connection (const char *hostname, int port, int iSize);
- ~Connection ( );
- bool connect (const char *host, int port);
- void disconnect (void);
- bool isConnected (void) const;
- int message_loop (FILE *in, FILE *out);
- int receiveMessage (char *msg, int maxsize);
- bool sendMessage (const char *msg);
- };

```
// Open UDP socket.
if( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0)  {
 cerr << "(Connection::connect) Cannot create socket " << host << endl;
 return false ;   }
// insert the information of the client
cli_addr.sin_family      = AF_INET ;
cli_addr.sin_addr.s_addr = htonl(INADDR_ANY) ;
cli_addr.sin_port        = htons(0) ;
 // bind the client to the server socket
if(bind(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr)) < 0)  {
cerr << "(Connection::connect) Cannot bind local address " << host << endl;
  return false ;   }
//  Fill in the structure with the address of the server.
m_sock.socketfd = sockfd ;
m_sock.serv_addr.sin_family      = AF_INET ;
m_sock.serv_addr.sin_addr.s_addr  = inet_addr(host);
m_sock.serv_addr.sin_port        = htons(port) ;
```

4.1消息解析

- 视觉消息，如：SeeGlobalMessage、SeeMessage、SenseMessage；
- 听觉消息，如：RefereeMessage、CoachMessage、PlayerMessage；
- 控制消息，如：ChangePlayerTypeMessage、CheckBallMessage.

4.1消息解析

- (see 0 ((g r) 64.1 13) ((f r t) 65.4 -16) ((f r b) 79 38) ((f p r t) 46.1 -6) ((f p r c) 48.4 18) ((f p r b) 58 37) ((f g r t) 62.8 7) ((f g r b) 66 19) ((f t r 20) 38.5 -38) ((f t r 30) 46.5 -30) ((f t r 40) 55.7 -25) ((f t r 50) 64.7 -21) ((f b r 50) 80.6 41) ((f r t 30) 69.4 -12) ((f r t 20) 67.4 -4) ((f r t 10) 67.4 4) ((f r 0) 69.4 12) ((f r b 10) 72.2 20) ((f r b 20) 75.9 27) ((f r b 30) 81.5 33) ((l r) 62.8 -89))

4.1消息解析

- (***server_param*** (audio_cut_dist 50)(auto_mode 0)(back_dash_rate 0.6)(back_passes 1)(ball_accel_max 2.7)(ball_decay 0.94)(ball_rand 0.05)(ball_size 0.085)(ball_speed_max 3)(ball_stuck_area 3)(ball_weight 0.2)(catch_ban_cycle 5)(catch_probability 1)(catchable_area_l 1.2)(catchable_area_w 1)(ckick_margin 1)(clang_advice_win 1)(clang_define_win 1)(clang_del_win 1)(clang_info_win 1)(clang_mess_delay 50)(clang_mess_per_cycle 1)(clang_meta_win 1)(clang_rule_win 1)(clang_win_size 300)(coach 0)(coach_msg_file “”)(coach_port 6001)(coach_w_referee 0)(connect_wait 300)(control_radius 2)(dash_angle_step 45）

4.1.1信息解析

- void* sense_callback(void *v) {
  SenseHandler* s = (SenseHandler*)v;
- s->handleMessagesFromServer( );
- return NULL; }

pthread_create(&sense, NULL, sense_callback , &s);

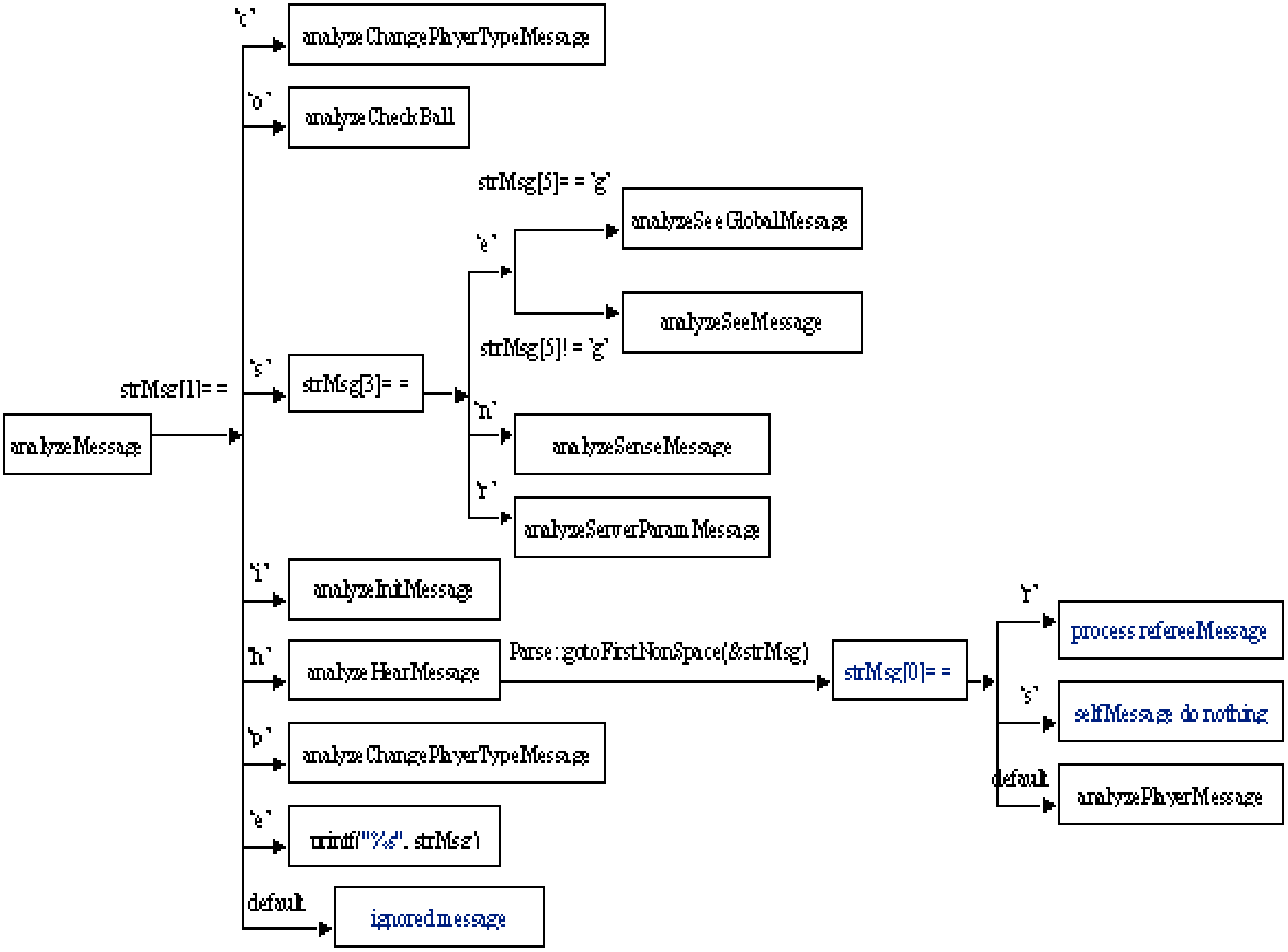
4.1.1信息解析

- **handleMessagesFromServer**()是整个信息解析的入口;
- 监听来自Server的消息，并把字符串信息交给**analyzeMessage**(char *strMsg)进行分类解析处理。

```
void SenseHandler::handleMessagesFromServer( ){
 char strBuf[MAX_MSG];
int i=0;
  while( 1 )  {
    strBuf[0]='\0';
    if( i != -1 )            // if no error
    i = connection->receiveMessage( strBuf, MAX_MSG
   );  // get message
   if( strBuf[0] != '\0' )                    // if not empty
   analyzeMessage( strBuf );          // parse message
  }
}
```

analyzeMessage函数

- **analyzeMessage**(char *strMsg)在进行消息解析时并没有按上述分类进行解析，而是判断消息字符串的首字符来分别进行处理，这样简化了程序的结构，提高了效率。解析处理的基本流程如下图。


analyzeMessage
strMsg[1]==
'c'
analyzeChangePlayerTypeMessage
'o'
analyzeCheckBall
's'
strMsg[3]==
strMsg[5]=='g'
'g'
analyzeSeeGlobalMessage
'e'
analyzeSeeMessage
strMsg[5]!='g'
'h'
analyzeSenseMessage
'r'
analyzeServerParamMessage
'i'
analyzeInitMessage
'h'
analyzeHearMessage
Parse::gotoFirstNonSpace(&strMsg)
strMsg[0]==
'r'
processRefereeMessage
'c'
selfMessage do nothing
default
analyzePlayerMessage
'p'
analyzeChangePlayerTypeMessage
'e'
printf("%s", strMsg)
default
ignored message

4.1.2 世界模型的更新

- 程序已经完成了实时消息的分类处理，不同的消息均已交递给其解析函数进行处理。
- 视觉信息更新函数：
  - `analyzeSeeMessage (char* strMsg)`

4.1.2 世界模型的更新

- analyzeSeeMessage (char *strMsg):
  - 1、从消息中提取出周期数，并更新世界模型中的周期，使之同步。
  - 2、将视觉信息保存在世界模型中，用于世界模型的更新。

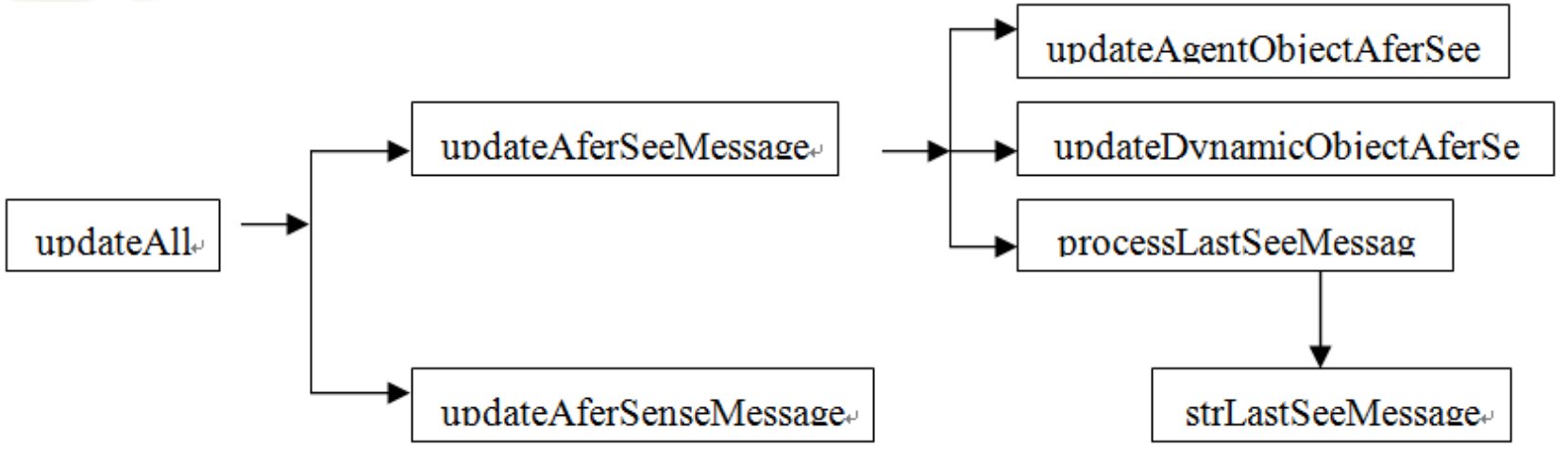
analyzeSeeMessage函数

- bool SenseHandler::analyzeSeeMessage(char *strMsg)
- {
- //将视觉信息保存在世界模型中，并由它处理
  - strcpy(WM->strLastSeeMessage, strMsg);
- Log.logWithTime(2, " incoming see message");
- Log.logWithTime(2, " %s",strMsg);
  - .
  - .//周期同步处理，不做介绍
  - .
  - return true;
- }

analyzeSeeMessage函数（续）

- 以strcpy(WM->strLastSeeMessage, strMsg)为入口，追踪世界模型的更新过程。
- 在WorldModel.h中定义了strLastSeeMessage[MAX_MSG]，用于存储最新看到的视觉信息。与之相关的方法有：
- bool processLastSeeMessage()　　[private]
- bool updateAfterSeeMessage()　　[private]
- bool updateAgentObjectAfterSee()　[private]
- bool updateDynamicObjectAferSee(ObjectT o)　[private]
- bool updateAll()
- 在player::mainLoop()中调用了updateAll()，使其不断的更新世界模型，包括视觉信息和感知信息，视觉信息更新的流程下图：

analyzeSeeMessage函数（续）



4.1.2 世界模型的更新

- 绝大多数消息的世界模型更新过程类似于上述过程。
- 在众多的消息中，我们要特别注意一下RefereeMessage，比赛中比赛状态的更新是由其决定的。消息解析函数通过对该消息的解析，从而更新世界模型中的比赛状态。

4.2.1动作解释发送ActHandler

- 当球员获得了足够真实的信息后，高层决策根据已有的信息作出决策，驱动球员进行相应的动作。动作经过解释成为标准的Server可接受的消息，再发送给Server，直观的由Monitor表现出来。
- 动作的解释主要是由SoccerCommand来进行的
- 而整体的控制是由ActHandler来完成的。

4.2.1动作解释和发送（续）

- Server规定了球员的一些基本动作，如dash、kick、turn等，诸如跑位、传球、射门此类的高级动作是由基本动作配以不同的参数来实现的。

4.2.1动作解释和发送（续）

- ActHandler控制着整个动作发送的流 程，值得注意的是其中的CommandQueue，动作进入队列后，会在“合适”的时间发出。
- 整个动作发送的流程。

4.2.2动作解释发送流程

- SoccerCommand和ActHandler的类结构和成员有了一定的了解.
- ActHandler 中 的 **putCommandInQueue**(SoccerCommand command)是面向球员动作执行的入口，也是整个动作解释发送的入口。

4.2.3 实例

- 我们以球员跑向定点（0,0）为例，底层动作函数为：
- SoccerCommand BasicPlayer::dashToPoint(VecPosition pos)

4.2.3 实例(续)

- 在决策部分Player中的调用的方式为：
- ACT->putCommandInQueue(dashToPoint (VecPosition(0, 0)))
- 这样这个跑位的动作进入队列，在球员的外层主循环中，每次均调用：
- ACT->sendCommands()
- sendCommands函数会触发命令的解释和发送，其基本过程简化如下：

5.2.3 实例（续）

1、遍历动作队列，解释动作。

- for(int i = 0； i < m_iMultipleCommands; i++)
- {
- m_queueMultipleCommands[i].getCommandString(
  - &strCommand[strlen(strCommand)], SS);
- }

4.2.3 实例（续）

- 上段代码的作用就是解释每个动作为相应的动作消息字符串，其中m_queueMultipleCommands[]存储的是soccerCommand类型的动作，即为动作队列；strCommand[]是存储解释后动作消息的字符串，所有的动作消息顺次存放。我们进入soccerCommand::getCommandString(char *str, ServerSettings *ss)内部，看看动作的解释过程：

1）进行基本动作选择，不同的基本动作对应不同的解释函数；

```
bool b = false;
        switch(commandType)
        {
        case CMD_DASH: b = makeDashCommand(str); break;
        . . ..
        default: . . . .
        }
        if(b == false){
        commandType = CMD_ILLEGAL;
        str[0] = ‘\0’
        }
        dashToPoint 返 回 的 soccerCommand.commandType == CMD_DASH
```

2）执行相应的解释函数makeDashCommand(str)，生成动作消息字符串。

4.2.3 实例（续）

2. 将动作序列字符串发送给Server。

```
if(strCommand[0] != '\0'
{
timeLastSent = WM->getCurrentTime();
connection->sendMessage(strCommand);
    Log.logWithTime( 2, " send queued action to
      server: %s", strCommand);
}
else
{
Log.logWithTime( 2, " no action in queue??" );
return false;
}
```

- 3、 connection->sendMessage(char*)将字符串发送给Server。

4.3 球员智能体的通信协议

- 通讯机制及简单应用
- 高效的利用字符串

4.3.1 通讯机制及简单应用-1

- 假设如下场景：当前我方的一名队员A拿球，他想将球传给我方的B球员，这个时候如果我们可以提前的告诉B球员先跑位或者过来接球，这就可以大大的提高我们决策的成功率。A如何将传球意图告诉B，B又怎么“听”得到呢？

4.3.1 通讯机制及简单应用-2

- 在10字节的信息里，告诉我想要传球的队员的号码，告诉他我的出球方向。
  - 用前两个字符来代表我想要传球目标的号码
  - 用接下来的三个字符来代表要传球的方向整数位，另加一个字符来表示小数点后的一位信息。
  - 还需要一个字符来表示正负号。
  - 2+4+1=7个字符

4.3.1 通讯机制及简单应用-3

- （1）构造好这样一个字符串以后，我们就可以打包发送了。下面是示例程序：
  - char chSend[10] = “10*+*1055”;//想传给10号，向+105.5绝对角度传球
  - //当然，我们可能还要对这段代码做一个简单的加密，当然只是很简单的。
  - chSend[1]+=chSend[5];
  - chSend[6]+=chSend[2];
  - soc = SoccerCommand(CMD_SAY, chSend);//说话
  - ACT->putCommandInQueue(soc);//这样，当前Agent就广播了上述只有我们自己人能看懂的一串字符串。在特定区域内的球员，只要将注意力集中在她身上，就可以获得此字符串。

4.3.1 通讯机制及简单应用-4

- （2）其他球员需要做这样的决策。假如刚才那段字符串是我方的11号队员说的，那我其他队员想接收到的话就需要调用：
  - ACT->putCommandInQueue(listenTo(OBJECT_TEAMMATE_11));

4.3.1 通讯机制及简单应用-5

（3）这样，我们就可以从11号球员那里得到上述chSend字符串。那这个字符串被世界模型更新存储在什么位置呢？

```
bool SenseHandler::analyzeHearMessage ( char *strMsg )
{
    RefereeMessageT rm;
    PlayModeT       pm;
    strcpy ( WM->strLastHearMessage, strMsg );
if ( WM->getPlayerNumber() == 11 )
{
        char chSend[10] = "10*+*1055";//想传给10号，向+105.5绝对角度传球
        chSend[1]+=chSend[5]; chSend[6]+=chSend[2];
        soc = SoccerCommand(CMD_SAY, chSend);//说
        ACT->putCommandInQueue(soc);
     return soc;
  }
   else
{
        ACT->putCommandInQueue(listenTo(OBJECT_TEAMMATE_11));
        cout <<"我是"<<WM->getPlayerNumber()<<"号，我听到"
                                        << WM->strLastHearMessage << endl;
        return soc;
}
```

4.3.1 通讯机制及简单应用-6

- 我们可以看到，每个球员都接收到了：（hear 85 -74 our 11 “la*+*1Z55”)这样的字符串，

4.3.2高效的利用字符串

- 取最前面的3位来表示所需要传递的信息的类型，这样可以最多表示8种类型的信息，如：球信息类型、传球信息类型、截球信息类型、守门员信息类型、越位线信息类型、反越位线信息类型、三个球员对象信息类型等。剩下的57位信息，每一种信息格式都有自己的编码方式，通过对前三位的解析，将剩下的57位信息交给各自信息的解码器来进行解码。

4.3.2高效的利用字符串

信息类型	消息内容
占3位	占57位

4.3.2高效的利用字符串

- 以传递三个球员对象信息类型的为例来说明该数据编码方法。消息内容的57位中用19位表示一个对象信息，其中5位表示球场上的对象名称。14为表示坐标信息，这样可以传达57/19=3个对象的信息。为更加清楚的表示，给出三个球员对象信息类型的60位组成，如表4.2。

信息类型	自己号码	自己坐标的X轴	自己坐标的Y轴	离自己最近的对方位置	次近的对方位置
占3位	占5位	占7位	占7位	占19位，具体同前	占19位，具体同前

4.3.2高效的利用字符串

- 在三个球员对象信息类型中，充分利用这60位的信息，总共传递3个场上球员的位置信息，包括传球球员自己的位置以及两个离他最近的对方球员的位置。