

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

- ❖ (**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)...))
- ❖ (**player_param** (allow_mult_default_type 0)(catchable_area_l_stretch_max 1.3)))
- ❖ (**player_type (id 0)**(player_speed_max 1.05)(stamina_inc_max 45))
- ❖ (**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)))
- ❖ (**hear** 1020 -30 ****))
- ❖ (**sense_body** 1020)

4.1.1 信息解析

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

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


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);
 - ✎ if(strBuf[0] != '\0') // if not empty
 analyzeMessage(strBuf); // parse message
 - ❖ }
- ❖ }

4.1.1 信息解析

❖ **analyzeMessage** (char *strMsg) 判断消息字符串的首字符来分别进行处理

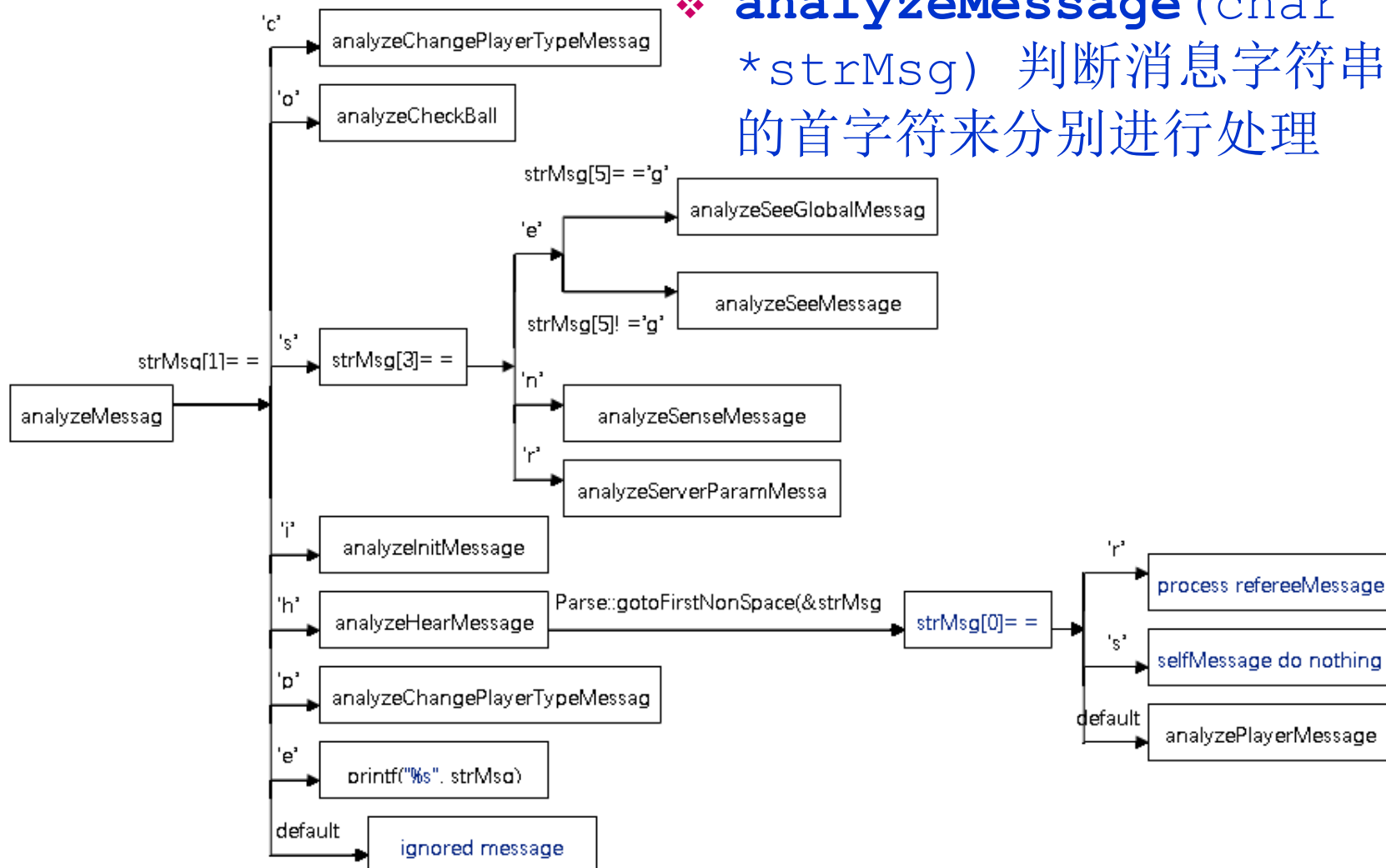


图 4.2 消息解析的基本流程

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，不同的消息均已传递给其解析函数进行处理。
- ❖ 视觉信息更新函数：
 - ✎ `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）

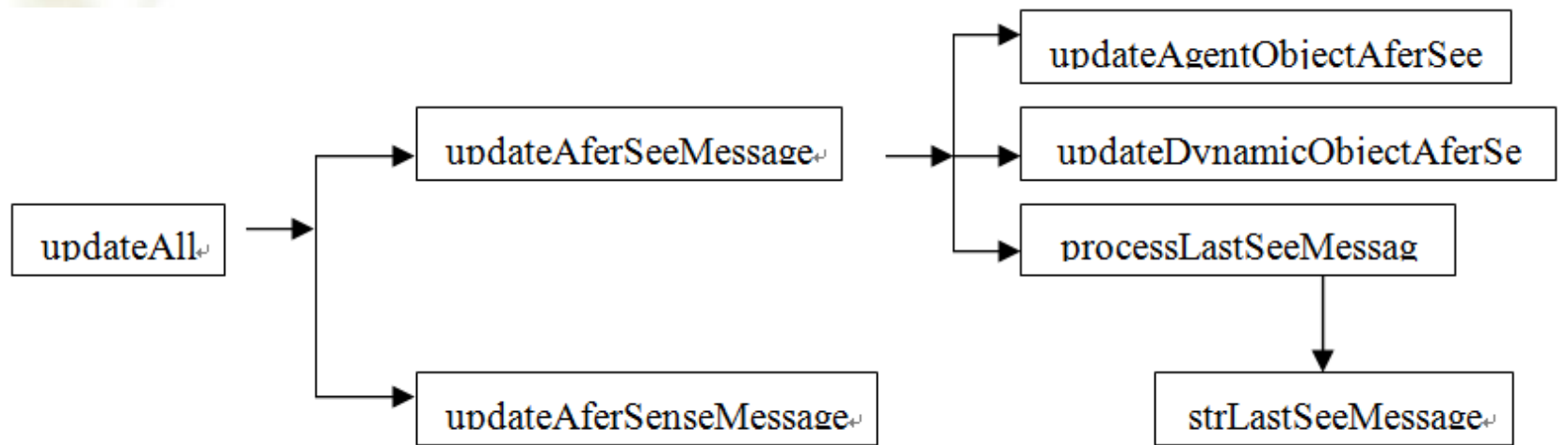
以strcpy(WM->strLastSeeMessage, strMsg)为入口，追踪世界模型的更新过程。

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 updateDynamicObjectAfterSee(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

analyzeSeeMessage函数（续）



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。
- ❖ ActHandler控制着整个动作发送的流程，很重要的是CommandQueue，动作进入队列后，会在“合适”的时间发出。
- ❖ 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。

Class ActHandler

- ❖ class ActHandler {
- ❖ Connection *connection; /*!< Connection with the server */
- ❖ ServerSettings *SS; /*!< ServerSettings with server parameters */
- ❖ WorldModel *WM; /*!< needed to set performed actions */
- ❖ SoccerCommand m_queueOneCycleCommand; /*!< primary command is saved here */
- ❖ SoccerCommand m_queueMultipleCommands[CMD_MAX_COMMANDS];
- ❖ int m_iMultipleCommands; /*!< number of non-primary commands */
- ❖ //成员函数
- ❖ };

Class ActHandler

- ❖ ActHandler(Connection* c, WorldModel *wm, ServerSettings *ss);
- ❖ bool putCommandInQueue(SoccerCommand command);
- ❖ void emptyQueue ();
- ❖ bool isQueueEmpty ();
- ❖ bool sendCommands ();
- ❖ SoccerCommand getPrimaryCommand();
- ❖ // methods to send commands directly to the server
- ❖ bool sendCommand (SoccerCommand soc);
- ❖ bool sendMessage (const char *str);
- ❖ bool sendCommandDirect(SoccerCommand soc);
- ❖ bool sendMessageDirect(const char *str);

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)
- ❖ 需要指出的所有的动作函数最终都要给出如下的参数：
- ❖ commandType, 对应的值，如：COMMAND_DASH
对应的值：power, angle

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：
遍历动作、解释动作、构造相应的字符串
发送字符串

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);  
}
```

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;
}
```


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：A如何把自己的传球意图告诉B:

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-2

(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-3

- ❖ (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-4

(3) 从11号球员那里得到上述chSend字符串。存储在strLastHearMessage

```
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-5

- ❖ 我们可以看到，每个球员都接收到了：
(hear 85 -74 “1a*+*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。