西北工業大學
NORTHWESTERN POLYTECHNICAL UNIVERSITY

NCStrategy

技术说明文档

作者：耿瑞琦

目录

一、前言：

2016 年 7 月，笔者随 NACIT 团队于美国加利福尼亚州圣地亚哥参加了 IAUVC 比赛，负责图像与控制部分。

由于各人有各人的编程风格，自然笔者与控制部分源代码撰写者嘻嘻也有一些在风格上的分歧。另外由于第一次参加这种比赛，所以对于总控框架，笔者和嘻嘻对其工作效率也不尽满意。对于此，笔者一直深表遗憾，只待老师决定下来我们老一辈的经验可以传承的指令，之后便准备就比赛时遇到的一些困难做参考，只依赖于稳定的 C++，完成一套只属于自己的总控程序。这是笔者撰写 NCStrategy 系列代码的初衷。

当然，该系列代码基本在于 C++，其余也涉及了很多 socket 网络编程、Qt 界面开发。笔者才疏学浅，在调试过程中也在这两方面上出现不少的 bug，耗费了不少精力调试，但 socket 网络编程是得到广泛应用的技术，Qt 界面开发也是很成熟的商业软件，所以也可以认为是稳定的技术，出现不尽人意的 bug，应该也是我们这些开发者的水平没有达到吧……

笔者为 NACIT 团队所撰写的 NCStrategy 系列代码，主要可以分为三个部分：

1、 基于 socket 通信的服务器/客户端 NCServer, NCClient：

图像、下位机、水声、调试界面等都作为单独的客户端，各自处理自己的信号并将处理结果传输给服务器端；服务器端接收各个传感器的信号，结合自定义的任务阶段 NCStage 与收到的传感器信号，判断任务进行的情况，并将判断后需要进行变动的结果作为指令发送给各个客户端；

2、 自定义任务阶段 NCStage：

将任务执行的流程分成一个一个的小阶段，每个阶段都有自己跳出目标与对应的条件，也有自己所属的任务，而且设定了时间限制。这样就把所有阶段串联在了一起，形成了一个任务网，这个阶段就被称为 NCStage；NCStage 运行在服务器端 NCServer 中；

3、 界面客户端 NCStrategy：

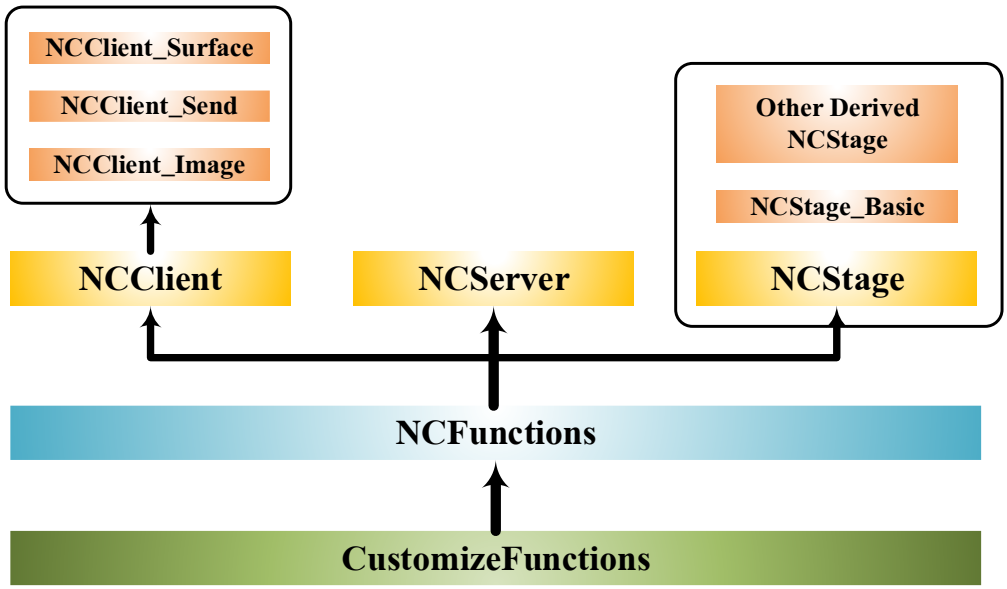
NCStrategy 本质上也是一个 NCClient，但主要作用是作为 NCServer 的可视化界面，将 NCServer 收到的信号数据、判断出的结果实时显示在界面上，并且和 NCServer 一样可以实现向各个客户端发送指令的功能。

下面对各个部分分别说明：

二、NCStrategy 系统结构：

1. 代码结构：

NCStrategy 的代码结构如下图所示：



基层是笔者为了使用方便而写的一些自定义函数，即 CustomizeFunctions。以这些自定义函数为基础，写出 NACIT 团队专属系列函数集 NCFunctions，其中包括客户端部分 NCClient，总控服务器部分 NCServer，策略运行过程基本单位 NCStage，意为策略的过程是由一个个阶段组合构成。NCClient 派生出各客户端程序 NCClient_Image, NCClient_Send, NCClient_Surface；NCStage 也派生出其他阶段单位。

目录具体如下：

（1） CustomizeFunctions：自定义函数
- A. CustomizeStructs：自定义结构体；
- B. GeneralImageProcess：通用图像处理函数；
- C. SupportFunctions：其余支持函数，如部分系统操作、字符串处理函数等；

（2） IPCClients：客户端程序
- A. IPCImageClient：ImageClient 的控制台实体程序；
- B. IPCSendClient：SendClient 的控制台实体程序；
- C. IPCSurfClient：SurfaceClient 的控制台实体程序；

（3） IPCServer：服务器 NCServer 的控制台实体程序

（4） NCFunctions：NACIT 专属代码
- A. NCClient：客户端及其派生客户端的类定义代码；
  - a. 基类 NCClient；
  - b. 派生类 NCClient_Image，图像客户端；
  - c. 派生类 NCClient_Send，下位机客户端；（该客户端只是用来测试，所以

只有参考价值；这里随便取名为 Send，以后最好重写，改名成 NCClient_Serial 之类的名字）

d. 派生类 NCClient_Surface，Qt 界面客户端；

B. NCServer：服务器的类定义代码

a. 类函数文件 NCServer_dataproc.cpp：对获取的数据处理与分析的类函数；

b. 类函数文件 NCServer_Link.cpp：网络连接相关类函数；

c. 类函数文件 NCServer_Record.cpp：数据记录相关类函数；

d. 类函数文件 NCServer_Stage.cpp：策略阶段控制相关类函数；

e. 类函数文件 NCServer_Strategy.cpp：策略运行相关类函数；

C. NCStage：策略阶段的类定义代码

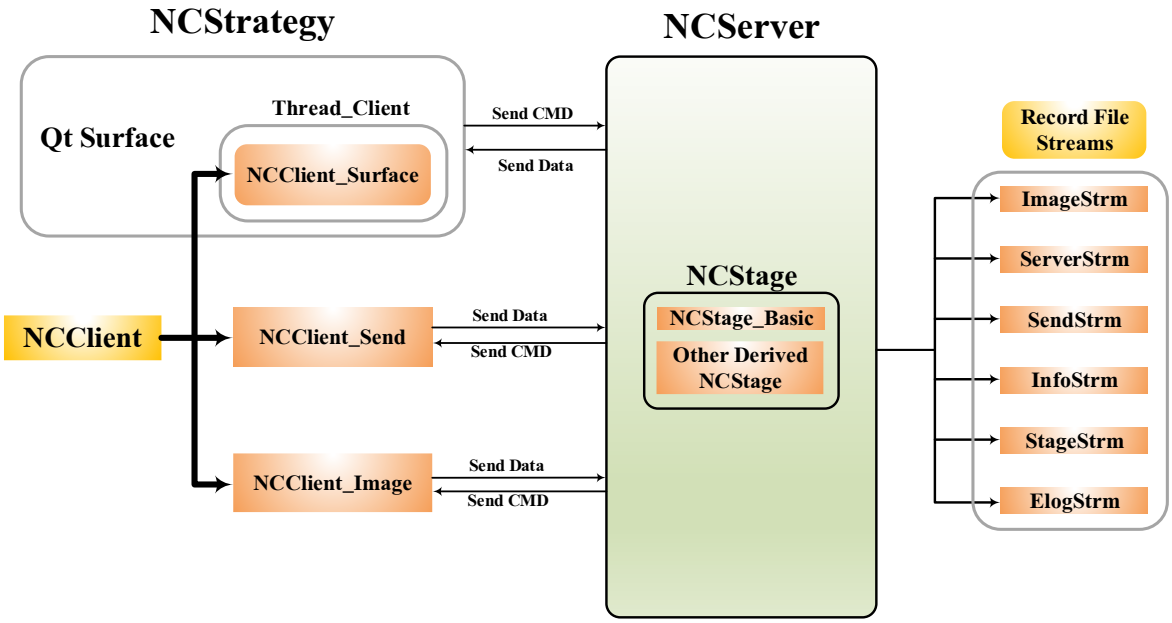
a. 基本 NCStage：NCStage 基类，以及由 NCStage 派生出的特殊阶段的类定义函数；

b. 测试用 NCStage：由 NCStage 派生出的测试用 NCStage 派生类的类定义函数；

（5） NCStrategy：NACIT 策略总控界面

2. 工作结构：

NCStrategy 的工作结构如下图所示：



NCStrategy 系统基本以 NCServer 服务器部分为中心进行整体系统的工作。

NCServer 运行后，等待其他客户端连接。在笔者编写系统代码的过程中，尝试写了三个客户端代码：模拟图像处理的客户端 NCClient_Image，模拟下位机的客户端 NCClient_Send（但没有写串口部分，有点滥竽充数的客户端……），NCServer 的界面客户端 NCClient_Surface。

NCServer 与各个 NCClient 连接成功后，开始运行 NCStrategy 系统，同时 NCServer 端开始进行数据记录，记录路径下保存了图像客户端、服务器端、下位机客户端的发送接收消息记录，以及各个传感器信息记录、策略阶段运行记录、连接情况与错误报告记录。

NCClient_Surface 相对于其他客户端是比较特殊的存在，它是一个显示服务器内部数据

的界面客户端，也可以向服务器传输命令并使服务器执行该命令，所以可以将其作为服务器的界面版本。

NCClient_Surface 工作在 NCStrategy 工程中，该工程由两部分组成：一部分是 Qt 显示界面，显示 NCServer 的实时信息，并可以向 NCServer 发送命令；另一部分是在另一线程中运行的 NCClient_Surface，它负责界面与服务器的连接部分。

三、基于 socket 通信的服务器/客户端 NCServer, NCClient：

1. 为什么要写 NCServer/NCClient 架构？

2016 年 NACIT 团队的总控的实现，主要是基于下面的三句代码实现的：

```
QTimer *time_clock = new QTimer(this);
connect(time_clock,SIGNAL(timeout()),this,SLOT(AuvMainControl()));
time_clock->start(CONTROLTIME);
```

这里用到了 Qt 的信号槽机制：定义一个定时器 time_clock，设定定时器每 CONTROLTIME 个时间便调用一次函数 AuvMainControl()函数。而 AuvMainControl()函数中便实现了单个周期对航行器的控制。

对于这种单线程的简单控制，用一两句话就能把其中原理说明白，其优点当然是控制简单，但其中也有缺点。缺点就是在于效率比较低。

AuvMainControl()中，包含了任务控制、串口数据读取与解算、图像处理与分析等功能。字符串相关的分析倒没什么，但图像处理需要大量的资源，消耗大量时间来计算，所以经常导致主控程序调用的 Qt 计时器不精准。当然这也与笔者在图像处理方面的功力不足有关。

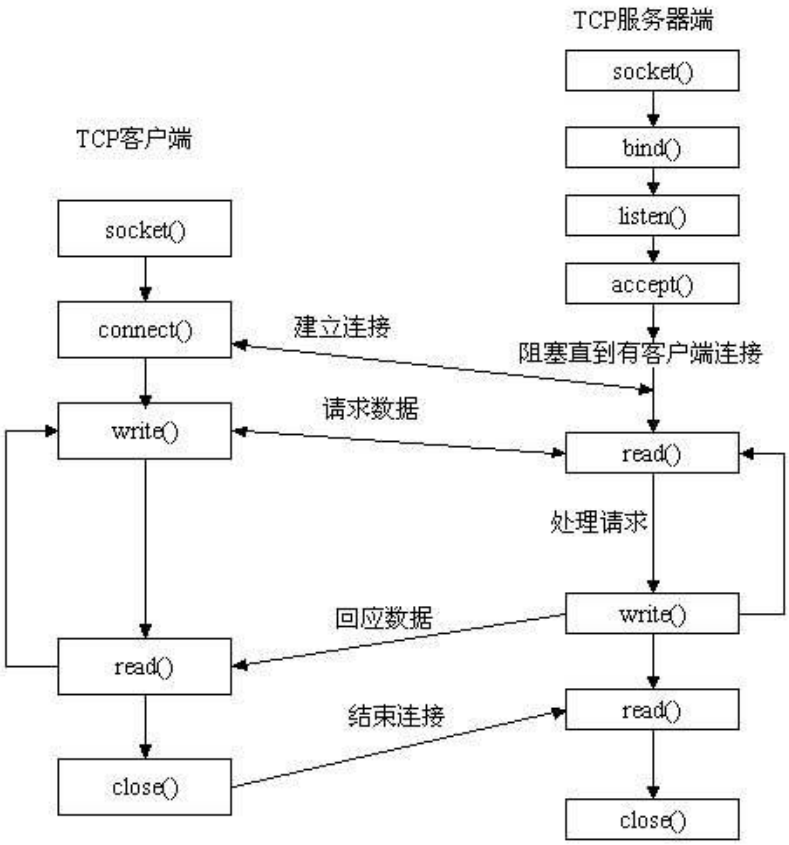
另外，单线程任务实现起来简单，但万一出现了中间某个环节出现错误崩溃的情况，在笔者没有写相关的异常处理功能的前提下，势必导致整个总控程序的崩溃。

综上，笔者认为应该把单线程的控制框架改为多进程框架：以总控程序作为服务器端，其余传感器相关的程序作为客户端，服务器端与客户端之间相互通信，由服务器端实现信息的汇总与决策，实现对航行器的控制。一方面，以图像传感器为首的客户端可以独自占用系统资源而不影响控制系统其他部分的工作，提高了整个控制系统的运行效率；另一方面，如果产生异常，则不会导致整个控制系统的崩溃，并在异常出现的同时作出适应的反应动作。

这里，笔者使用的是以 TCP 协议为基础的 socket 网络通讯，实现了进程间的通信。下面分为服务器端 NCServer 与客户端 NCClient 分别说明：

2. 简要介绍 TCP 协议的三次握手与 socket 套接字：

笔者使用的 socket 网络通讯实现了进程间收与发的通讯，是基于 TCP 协议的。TCP 的三次握手完成过程，基本如下图所示：



参考链接：http://goodcandle.cnblogs.com/archive/2005/12/10/294652.aspx

至于 socket 通讯，就是较为复杂的 TCP 协议的编程 API 接口。对编程用户来说，了解调用接口即可实现 TCP 通讯。

注：

Socket 通讯是可以实现网络间的进程间通讯的，但由于笔者功力不足，且现在只是想实现本地的进程间通讯，所以没有作更多的尝试。如果以后师弟们有时间有精力，完全可以扩展为局域网的进程间通讯。

如果以后把系统移植到了 Linux 下，并实现了这个功能，那么笔者认为完全可以更进一步的构建软件框架：以航行器的主控程序作为服务器，航行器工控机连接岸上的路由器并设置局域网，其它电脑连入局域网。在这样的基础上，工控机中将获取的信息以文件的形式保存在某个地址，并合理的设置这些文件的读写权限，这样岸上的其它电脑就可以通过网络读取工控机的信息，在自己的电脑上读取分析这些数据。

如果完成了上述技术，那么便可以针对图像、水声等模块写自己的监控程序，调试时便可以令图像、水声等负责人分别打开自己的监控程序，实时监控他们自己负责的部分。

3. 总控服务器 NCServer：

(1) 服务器端的工作流程：

A. socket()：创建 socket；

B. bind()：绑定 socket；
C. listen()：服务器监听；
D. while 循环：服务器工作；
   a. 设置超时时间；
   b. 初始化 FD（File Descriptor，文件描述符）集合；
   c. 连接服务器端，FD_SET()函数；
   d. 连接多个客户端，FD_SET()函数；
   e. select()：I/O 端口复用；
   f. send(), recv()：向客户端和服务器端发送数据；
   g. 客户端对应数据处理；
   h. 策略执行（具体见一. 3. (2) 策略工作流程）

(2) 策略工作流程：

策略工作流程基本是一个大循环，以自定义的 NCStage 为基本单元进行工作。设置当前阶段 CurStage，对 CurStage 进行判断与更新，实现策略的工作流程。

关于 NCStage 的说明，请见第二章。

工作流程如下：

A. 初始化 CurStage 为起始阶段 Stage_START，并设置初始任务；
B. while()循环，直到判断 CurStage 到达了最终阶段 Stage_END 为止：
   a. 激活 CurStage；
   b. 判断激活后的 CurStage 是否错误：根据 CurStage 的目标 Stage 名称，判断是否 CurStage 是否错误；
   c. 循环运行 CurStage，直到判断 CurStage 已经完成，或已经达到 CurStage 的最大时间为止：
      I. 设置超时时间；
      II. 初始化 FD（File Descriptor，文件描述符）集合；
      III. 连接服务器端，FD_SET()函数；
      IV. 连接多个客户端，FD_SET()函数；
      V. select()：I/O 端口复用；
      VI. send(), recv()：向客户端和服务器端发送数据；
      VII. 客户端对应数据处理；
      VIII. 服务器数据记录到文件；
      IX. 当前阶段、任务状态发送给界面客户端；
      X. 运行 CurStage；
   d. 若任务超时，则进入该任务的目标任务；
      跳出上面的循环后，CurStage 会返回几种不同的值。如果获取任务超时状态 STATE_DONE_TIMEUP_MISSION，说明在当前任务上已经耗费了太长时间，这时候就需要进入当前任务的默认目标任务。
   e. 目标 Stage 的同名检测与警告；
      如果目标 Stage 与当前 Stage 相同，则可能会出现循环在当前 Stage 的情况，这种情况可能会引起死循环，所以这里做出警告；
      注：这里完全可以将错误信息输出到 NCElog.dat 中，但笔者当时忘加了，师弟们

可以考虑加进去；

f. 此时 CurStage 已经运行结束，将 CurStage 的数据类型更新为 CurStage 的目标 Stage 的数据类型；

(3) 服务器状态信息 Server_State：

服务器状态信息记录了不同的工作状态，用以判断并记录服务器运行的错误状态；

A. SERVER_LINKING：服务器正常连接状态；
B. SERVER_ERROR_SOCKET：创建 socket 错误；
C. SERVER_ERROR_BIND：绑定 socket 的 bind 函数错误；
D. SERVER_ERROR_LISTEN：监听 listen 函数错误；
E. SERVER_ERROR_STAGENULL：当前阶段 CurStage 的目标阶段为空；
F. SERVER_ERROR_ACCEPT：接受连接请求 accept 函数错误；
G. SERVER_END：CurStage 运行到了最终阶段 Stage_END，策略运行结束；

(4) 数据记录：

服务器中有数据记录功能，开启了数据记录功能后即可运作。
主要记录的内容有：

A. NCServer.dat：记录服务器接收、发送的信息；
B. NCImage.dat：记录图像客户端接收、发送的信息（暂时为空）；
C. NCSend.dat：记录下位机客户端接收、发送的信息（暂时为空）；
D. NCInfo.dat：记录各传感器随时间变化的数值（暂时为空）；
E. NCStage.dat：记录总控策略运行随时间变化的 Stage, Mission 状态；
F. NCElog.dat：记录网络连接状况，以及 NCServer 运行过程中的错误信息；

4. 客户端 NCClient：

(1) 客户端工作流程：

A. 设置服务器地址；
B. 创建 socket；
C. 连接服务器 socket 文件描述符与服务器地址；
D. 客户端数据初始化；
E. FD（文件描述符）集合的初始化与设置；
F. 进入循环：
   a. FD 集合初始化与设置；
   b. 客户端选择 FD 集合；

c. 客户端数据处理；
d. 向服务器传输数据并接受数据；
e. 客户端数据更新；

注：笔者现在认为，也许 F.c, F.d 的顺序应该调换一下，读者们可自行斟酌；

(2) NCClient 基类：

客户端程序用来处理传感器相关的内容，比如笔者就大致写了两个用来参考的客户端程序：模拟图像客户端 Client_Image 与模拟下位机数据客户端 Client_Send。所有不同客户端的共同部分，便是 NCClient 基类。至于体现多态性的其余部分，可以用派生类的虚函数与新增变量实现。

这里主要说明几个比较重要的函数：

A. 通用函数部分：

void ClientFunction()：客户端实际工作部分，连接完成后，便开始进行客户端工作；该函数的工作流程即为 4.(1)的 C—F 部分内容；

B. 虚函数部分：

virtual void ClientInit()：客户端数据初始化；

virtual void ClientProcessing()：客户端数据处理部分；

virtual void SendToServer()：向服务器传输数据；

virtual void ReceiveFromServer：从服务器接收数据；

virtual void CmdProcess()：客户端匹配到来自于服务器传来的消息后，处理服务器传来的指令；

virtual void ClientDataUpdate()：客户端数据更新；

(3) NCClient 派生类：

笔者此处写了三个 NCClient 派生客户端为读者进行参考：模拟图像客户端 Client_Image 与模拟下位机数据客户端 Client_Send 与界面客户端 Client_Surface。下面对这三个客户端的功能进行简单的说明：

A. 模拟图像客户端 Client_Image：

笔者测试用的 Client_Image，实现了对 PC 机视频流文件的读取，并根据从服务器接收的指令，进行不同的图像处理策略（模拟针对不同任务的处理）。

按照前文所述的客户端工作流程进行说明，说明如下：

a. 图像客户端数据初始化 ClientInit()：设置了视频流在 PC 机的地址，并打开该视频；
b. 图像处理函数 ClientProcessing()：根据服务器传来的不同指令，作不同的处理操作：

IMG_NO：该指令下不做任何处理；

IMG_GRAY：该指令下做灰度处理；

IMG_BIN：该指令下做二值化处理；

c. 向服务器传输数据 SendToServer()：仅直接调用 NCClient 基类测试函数，发送持续增大的整型数；
d. 从服务器接收数据 ReceiveFromServer()：接收服务器指令，对指令进行分析，并更新客户端当前工作任务；
e. 分析指令 CmdProcess()：在 ReceiveFromServer()函数中调用，用于分析指令，更新客户端当前任务；
f. 客户端数据更新 ClientDataUpdate()：调用 OpenCV 函数 waitKey()对图像帧进行更新；

关于从服务器端接收到的指令，有一定的格式要求：

服务器指令格式为：CLIENT_NAME@CMD

CLIENT_NAME 是目标客户端的名称，此处需要写成”IMAGE”；

CMD 是针对该客户端的指令。

注：CLIENT_NAME 与 CMD 如果写成了小写也可以，笔者在 ReceiveFromServer()函数中，已经将 CLIENT_NAME 与 CMD 作了大写处理；

针对图像客户端，笔者定下了三个指令：

IMAGE@RAW：不作任何处理，直接显示原图；

IMAGE@BGR2GRAY：对原图作灰度处理；

IMAGE@BGR2BIN：对原图作二值化处理；

B. 模拟下位机数据客户端 Client_Send：

笔者测试用的 Client_Send，实现了从服务器接收的指令，并持续向服务器发送数据的功能。

按照前文所述的客户端工作流程进行说明，说明如下：

a. 图像客户端数据初始化 ClientInit()：该客户端中无处理；
b. 图像处理函数 ClientProcessing()：该客户端中无处理；
c. 向服务器传输数据 SendToServer()：仅直接调用 NCClient 基类测试函数，发送持续增大的整型数；
d. 从服务器接收数据 ReceiveFromServer()：接收服务器指令，对指令进行分析，并更新客户端当前工作任务；
e. 分析指令 CmdProcess()：在 ReceiveFromServer()函数中调用，用于分析指令，更新客户端当前任务；
f. 客户端数据更新 ClientDataUpdate()：调用系统函数 usleep()，每 0.3 秒发送一次整型数据到服务器端；

收到的服务器指令格式依旧为 CLIENT_NAME@CMD

CLIENT_NAME 为”SEND”，CMD 可以为任意字符串，该客户端全部接受并显示在客户端的控制台上；

C. 界面客户端 Client_Surface：

该客户端用于 Qt 界面部分，此处暂不介绍，具体请见后文的 Qt 界面介绍部分；

四、策略阶段 NCStage：

1. 为什么要写 NCStage 架构？

在 2016 版本的 NACIT 总控程序架构中，控制策略的代码形式是这样的：

```
// 风格门没有稳定在WHOLEDOOR的时候
if(!OffCenterOK)
{
    // 如果门状态为WHOLEDOOR，开始计时
    if(VInfo.StyleDoorSeenState == STYLE_WHOLEDOOR)
    {
        hold_time++;
        // 侧推停止
        if(SF)
            on_ExecuteSend(S_S);
        // WHOLEDOOR状态保持3秒以上，标志位置为真
        if(hold_time >= 3)
        {
            OffCenterOK = true;
        }
    }
    // 门状态不为WHOLEDOOR
    else
    {
        // 保持时间清零
        hold_time=0;
        if(MF)
            on_ExecuteSend(M_S);
        else
        {
            // 左状态，右移
            if(VInfo.StyleDoorSeenState==STYLE_HALFLEFT)
            {
                on_ExecuteSend(S_R2);
            }
            else
            {
                // 右状态，左移
                if(VInfo.StyleDoorSeenState==STYLE_HALFRIGHT)
                {
                    on_ExecuteSend(S_L2);
                }
                else
                {
                    // 底部状态，向上升20cm
                    if(VInfo.StyleDoorSeenState==STYLE_BOTTOMONLY)
                    {
                        if(SF)
                            on_ExecuteSend(S_S);
                        float temp=depth.toFloat() - 0.2;
                        Send_Depth(temp);
                    }
                }
            }
        }
    }
}
```

满屏的换行符让笔者看的简直生活不能自理……

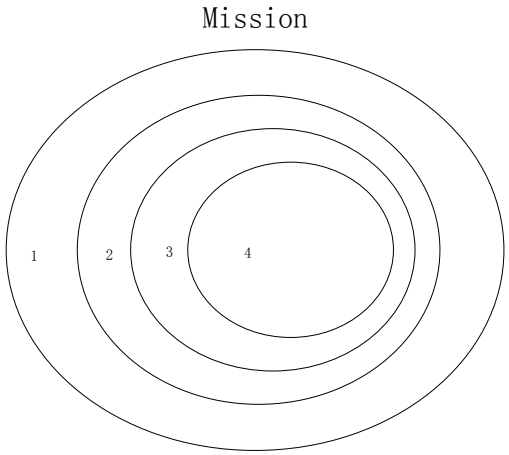
另外若按照这种形式来做总控的话，容易形成一种思路：我先简单思考应该怎么控制，先写一个简单的控制策略，看调试过程中遇到什么问题，然后根据这些问题一点一点加上去。这样抱着一点一点加控制内容的思路，代码会一步步慢慢变得混乱，比较严重的后果有三个：一是程序员会自己慢慢不明白自己写的是什么，失去对代码的维护能力；二是调试时暂时性添加的变量过多过杂且难以记忆它的具体作用，在一层层的 if…else…嵌套

中，很容易导致程序运行过程中意外跳转到不同层的 if…else…；三是很容易遗漏边界条件，在没有大量调试时间的情况下，很容易进入边界条件的情况。

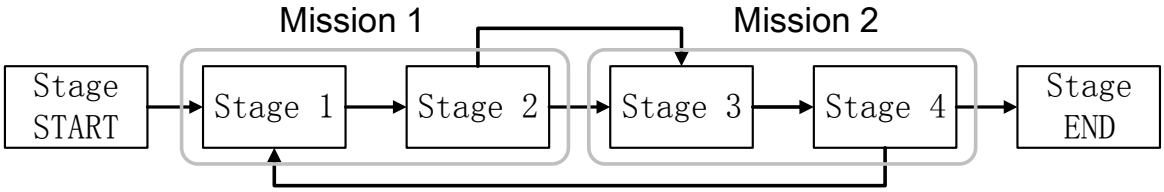
笔者认为如果用这种通篇 if…else 的控制方法，决不能用这种一点点加入控制策略的思路，应该从写控制策略之前就尽可能考虑到所有情况，然后才能较为系统性的写出控制策略。但是这种情况也极其考验策略制定者的思维缜密程度，也很容易有很多考虑不到的情况出现。

综上，笔者认为单纯的 if…else 结构层层嵌套，在维护上有很大的难度，所以便写了模块化进行的 NCStage 类。关于两者的工作模式：

可以说，if…else 的结构更像洋葱式的层式结构，由外到内的实现判断；



而笔者撰写 NCStage 的目的，是以 NCStage 派生类为基本模块的，实现模块化的进行任务流程。进入到某个 NCStage 中，达到特定条件之后便会跳转至下一个特定的阶段，否则就在该 NCStage 内部运行。笔者希望这样的思路，可以获得各 Stage 内部的程序运行局部封闭性，也可以获得各 Stage 外部连接的灵活性。



2. 基类 NCStage：

上图中，Stage 1, Stage 2, Stage 3 等都是独一无二的。所有独一无二的 NCStage 的共同部分，便是基类 NCStage。至于其他体现多态性的其余不同部分，可以用派生类的虚函数与新增变量实现。

(1) NCStage 类成员变量：

timeval StartTime：该阶段开始时间，格式为系统时间格式；
double RunTime：该阶段运行时间，单位秒；
double MaxTime：该阶段最大运行时间，单位秒；

注：阶段最大运行时间在 NCStage 派生类的构造函数中确定；

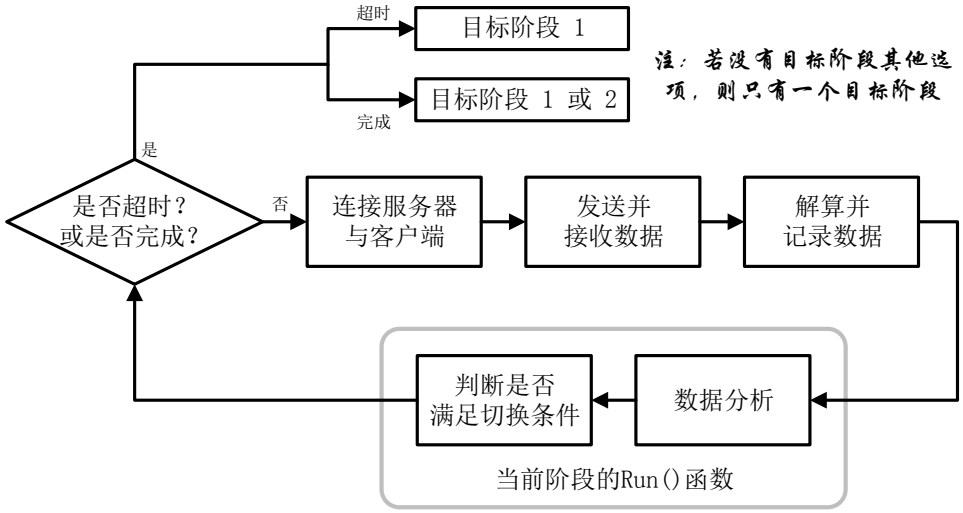
string Name：该阶段名称；
string NextName：该阶段的目标阶段名称；
Stage_State State：该阶段的运行状态；
bool isDone：该阶段是否已经完成；
注：isDone 值一般为 false；当 isDone 为 true 时，也会在相应的哈数中被重新置为 false；
bool isActived：该阶段被激活；
注：isActived 值一般为 false；当 isActived 为 true 时，也会在相应的哈数中被重新置为 false；
bool isNoSwitch：该阶段是否存在目标阶段的分支；
注：关于阶段之间的切换，请见后文说明；
Mission CurMission：当前任务；
timeval StartTime_Mission：当前任务开始时间，格式为系统时间格式；
double RunTime_Mission：当前任务运行时间，单位秒；
double MaxTime_Mission：当前任务最大运行时间，单位秒；
注：任务最大运行时间在 Active()函数中的 setMaxTimeMission()函数中设置，且 Active()函数在构造函数中被调用；
string Name_Mission：当前任务名称；
vector<bool> ChangeCondition：判断是否达到某种更换目标 Stage 的条件；
vector<string> ChangeName：与 ChangeCondition 向量配合，存放不同条件下更换目标 Stage 对应的名称；
注：关于两个 vector，将在后面的阶段切换部分进行说明；

(2) NCStage 类成员函数：

A. 虚函数：
每一个由 NCStage 派生出的 NCStage 都是独一无二的，其多态性是从虚函数与继承类中自定义数值体现出来的。其中，虚函数是众多 NCStage 的多态性最大体现。
virtual void *InitStageData*()：初始化该阶段的数据；
virtual void *StageFunction*()：该阶段的数据分析阶段；
virtual void *SetDefaultStagesName*()：设置默认的当前阶段与目标阶段名称；
virtual void *InitChangeNextStage*()：对更换目标阶段的条件、名称进行初始化；
virtual void *UpdateChangeNextStage*()：对更换目标的条件、名称进行更新；
上述五个函数都是在 NCStage 的通用 Active()，Run()函数中运行，虽然 Active()，Run()函数是固定顺序执行的，但其中每个虚函数都是不一样的，所以决定了 Active()，Run()函数的多态性，也就决定了每个 NCStage 派生类的多态性。
B. 通用函数：
每一个 NCStage 派生类都共同使用的函数，封装在基类 NCStage 中。
这里主要介绍几个主要的通用函数：
void UpdateTime(bool output = true)：更新当前时间，包括当前阶段 Stage 运行时间与当前任务 Mission 运行时间；

void Run()：NCStage 实际运行函数；

NCStage 的 Run()函数主要配合 NCServer 获取数据部分共同运作。工作流程如下图所示：



void Active()：激活当前阶段；

激活部分有下列任务：

a. 阶段完成标志位 isDone 置为 false；
b. 阶段运行状态 State 置为 STATE_RUNNING；
c. 获取当前时间作为阶段起始时间 StartTime；
d. 设置默认的当前阶段与目标阶段名称，通过虚函数 *SetDefaultStageName*()实现；
   注：默认的目标阶段，可以理解为超过了阶段最大运行时间而进入的目标阶段；
e. 对目标阶段的更换条件、名称进行初始化，通过虚函数 *InitChangeNextStage*()实现；
f. 对该阶段的数据进行初始化，通过虚函数 *InitStageData*()实现；
g. 设置该任务最大运行时间；

激活函数 Active()一般应用于构造函数之中；

bool isTimeUp(bool output = true)：判断是否超时函数；

判断超时分为两部分：是否超过阶段最大时间，是否超过任务最大时间；

如果超过阶段最大时间，则阶段运行状态 State 返回 STATE_DONE_TIMEUP_STAGE；

如果超过任务最大时间，则阶段运行状态 State 返回 STATE_DONE_TIMEUP_MISSION；

3. NCStage 派生类：

(1) NCStage_START, NCStage_END：

NCStage_START 是每一个任务的第一个阶段，也是服务器端程序运行时进入的第一个阶段，通过 SetStartStage()函数，可以将当前阶段设置为 NCStage_START，并为其设定对应的任务；

NCStage_END 是最后一个阶段，当前阶段运行到 NCStage_END，便意味着所有任务已经完成，跳出服务器运行函数的循环，服务器端程序退出并断开连接，客户端也随即退出。

(2) 其他自定义派生类：

对于其他自定义派生类，一般只需要更改其虚函数内容，并添加一些新变量，即可得到一个新的 NCStage 派生类。

关于命名原则：

笔者建议，NCStage 的派生类可以按照 NCStage_MISSION_NUM 的格式进行命名。

其中，MISSION 标识该 NCStage 所属的任务 Mission；NUM 标识该 NCStage 在该任务中的完成顺序，用 1, 2, 3, 4 标识；

例：笔者提供了三个 NCStage 派生类作参考，分别是 NCStage_TEST_1, NCStage_TEST_2, NCStage_TEST_3；三个命名中，TEST 是笔者定义的一个任务名称，后面的 1, 2, 3 便是在该任务中的标识；

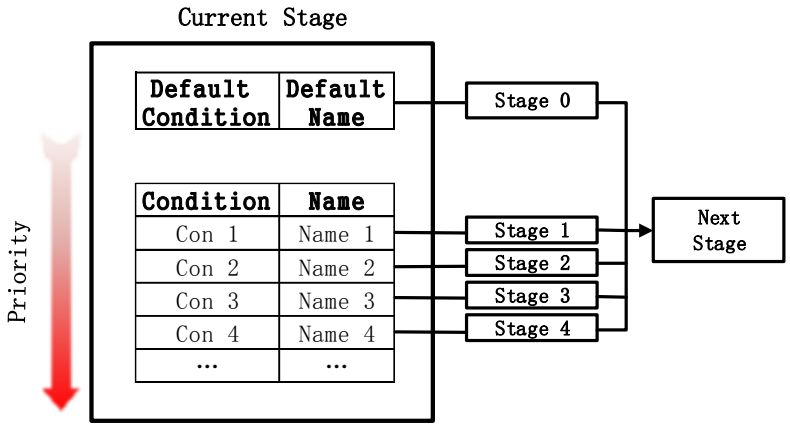
4. NCStage 阶段跳转机制：

理论上，笔者写的 NCStage 可以实现当前 NCStage 完成后，可以根据多种选择跳转到任一 NCStage 中。所以这里需要说明一下笔者所写的阶段跳转机制：

首先，NCStage 需要在构造函数中对成员变量 isNoSwitch 赋值：

若 isNoSwitch == false，则没有其他可供选择的目标阶段，这种情况下无论是超时跳出当前阶段，还是正常完成该阶段，都只能进入默认目标阶段；默认目标阶段的设置是在 *SetDefaultStagesName*()函数中实现的。

若 isNoSwitch == true，则除了默认目标阶段之外，还有其他可选择的阶段。示意图如下：



（1） 所有条件按照由上至下的优先级排列，且以超时而进入的阶段作为默认目标阶段；

（2） 若出现两个条件同时满足的情况，也会根据优先级高低而选择目标阶段；比如若同时满足途中 Condition 1, 2 则会因为 Condition 1 的优先级高于 2，所以会进入 Stage 1；

5. 阶段运行状态 Stage_State：

Stage_State 作为枚举变量，记录了当前阶段的工作状态。有如下状态：

（1） STATE_RUNNING：正在工作状态；
（2） STATE_CHANGE_NEXT：准备切换为目标阶段的状态；
（3） STATE_DONE_TIMEUP_STAGE：阶段超时而跳出当前阶段的状态；
（4） STATE_DONE_TIMEUP_MISSION：任务超时而跳出当前阶段的状态；
（5） STATE_DONE_NORMAL：正常满足当前状态而跳出当前阶段的状态。

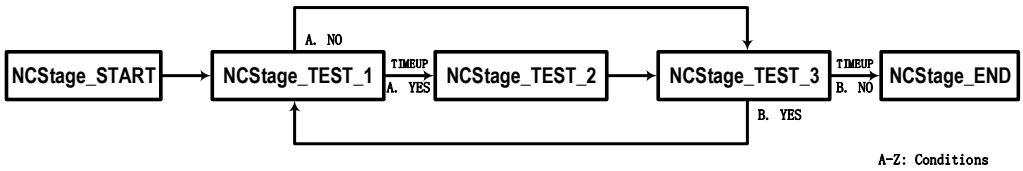
6. 笔者提供的 NCStage_TEST 例程：

笔者在测试阶段，已经撰写了三个 NCStage_TEST 系列的 NCStage 派生类以及两个任务 Mission 进行测试。现对其进行说明：

（1） NCStage 的派生类工作流程：

关于命名规则，笔者建议按照 NCStage_MISSION_NUM 的格式进行命名。笔者在这里设置的任务名为 TEST，所以定义的三个派生类名字分别命名为 NCStage_TEST_1, NCStage_TEST_2, NCStage_TEST_3。

三个派生类中，NCStage_TEST_1, NCStage_TEST_3 存在分支目标阶段，NCStage_TEST_2 只有一个目标阶段。任务流程如下图所示：



图中 A, B 是目标阶段跳转条件，解释如下：

A. NCStage_TEST_1 中设置了两个计数器：计数器 1 为固定计数器，从 1-10 计数；计数器 2 为随机计数器，每次增加不固定数值；若计数器 1 比计数器 2 快，则条件 A 为真，否则为假；
B. NCStage_TEST_3 中设置了两个计数器：计数器 1 为固定计数器，对 char 字符'A'--'Z' 计数；计数器 2 为随机计数器，每次增加不固定数值；若计数器 1 比计数器 2 快，则条件 B 为真，否则为假；

另外，NCStage_TEST_2 仅从 0.01—0.20 计数，无论计数结束或超时结束，都正常进入 NCStage_TEST_3；

7. 任务 Mission 测试说明：

关于任务 Mission，笔者为了完成任务计时功能，所以只是强行设置了两个任务：Mission_TEST 和 Mission_END 任务，令两个任务在满足固定条件时切换，直到运行到 Stage_END 为止。

设置的两个任务，Mission_TEST 和 Mission_END 任务中，Mission_TEST 的初始 Stage 为 NCStage_TEST_1，Mission_END 的初始 Stage 为 NCStage_TEST_2。

Mission 切换的条件：

A. 每个 Mission 执行了 3 个 Stage，便会切换 Mission；
B. 到达了 Mission 的最大执行时间，切换 Mission；

五、Qt 调试界面 NCStrategy

1. 为什么要写 Qt 调试界面：

为了调试的方便，肯定是要写一个界面的。2016 年版本的界面和控制完全放在了一起，笔者认为这有功能耦合的嫌疑。就如下图所示：

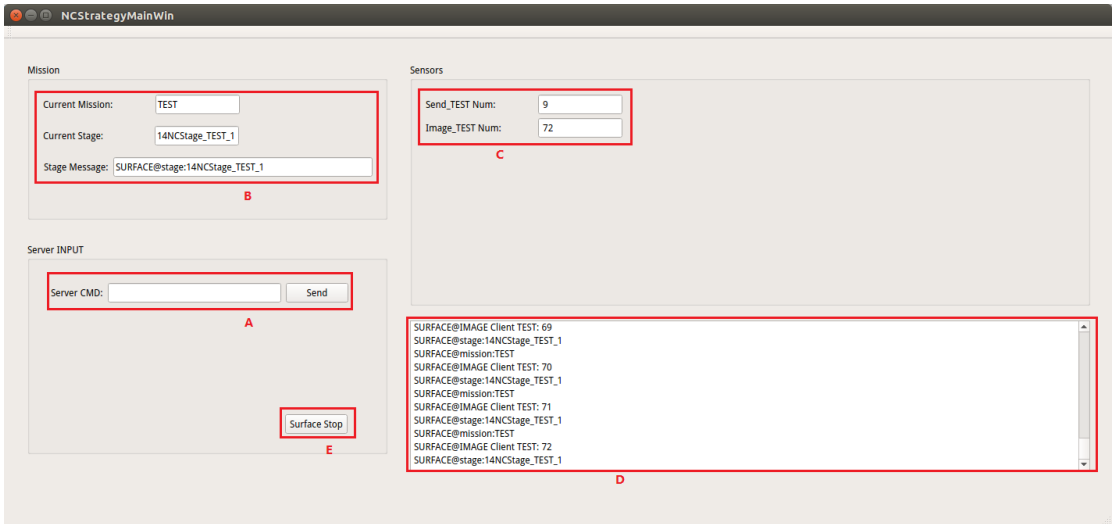
```
            // 22代表右边
            AcDir = 22;
            ui->lineEdit_acd->setText("RIGHT");
        }

        // 计算在解算方向的角度平均值
        GoAngle = GoAngle / u1;
        ui->lineEdit_acd2->setText(QString::number(GoAngle));

        // 角度解算完毕标志位置为真
        GoAngleOk = true;
        // 重新计数
        count_time = 0;
    }
}
```

强迫症如笔者，便把界面部分单独移出了控制程序，写出了这个相当于界面版本的服务器。

注：本来笔者想要直接把 NCServer 直接放在 Qt 调试界面中，但由于一开始并不想开启多线程，所以失败了。后来尝试多写了一个界面客户端用来做服务器与界面之间的数据传递，也是加了多线程才实现了功能。这样想来，也许把 NCServer 加入到多线程的方法应该也能实现。如果读者们有兴趣，可以自行实现。

2. NCStrategy 界面说明：

界面图如下所示：



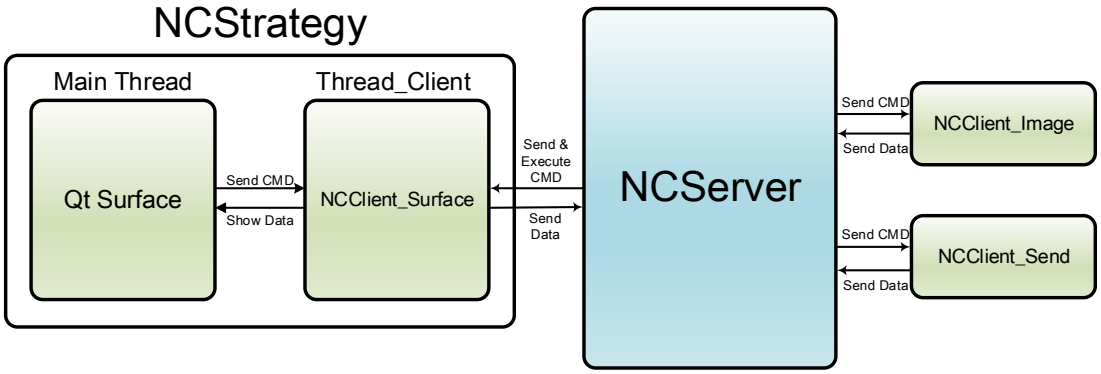
A. 向服务器端发送指令，服务器再向其他客户端发送该指令；

B. 当前任务、阶段的显示；

C. 其他客户端传来数据的分析结果；
D. 服务器端收到消息的实时显示；
E. 运行界面客户端，连接到服务器；

3. NCStrategy 工作说明：

NCStrategy 工作流程如下图所示：



NCStrategy 程序中，主线程用于界面的显示，新开的线程 Thread_Client 作为副线程，其中运行界面客户端 Client_Surface。

线程 Thread_Client 作为桥梁，用作连接 Qt 界面与主控服务器程序，服务器程序将程序内部的数据发送给 Thread_Client，Thread_Client 把数据发给 Qt Surface，将数据实时显示在界面上；反过来，Qt Surface 可以将命令发送给 Thread_Client，然后 Thread_Client 发送给服务器程序，服务器程序再将指令发送给其他客户端执行。

六、当前需要改进部分与已知 Bug：

1. 当前所有代码只在 Ubuntu 14.04 LTS 系统中运行成功，如果使用 Windows 系统，需要修改部分与时间、系统操作的头文件，请读者自行移植；
2. 每一个客户端在控制台程序运行时，都需要输入一下回车才开始工作；
3. 服务器主控程序运行到 NCStage_END 时会自动结束程序，此时其他客户端也会因为服务器程序的退出而退出，但再次连接会出现连接正在占用的情况，需要等待一会儿才能重新连接；
4. 每次都需要先将服务器程序打开，然后再打开所有客户端程序，在 Windows 系统下，可以考虑用 cmd 操作同时打开几个程序；Linux 下可以使用 bash 操作实现同样的功能；
5. Qt 界面可以再添加一种仿真界面，通过读记录文件进行仿真。中间可以再加上如 2/4/8 倍速的放慢播放功能；

七、寄语：

会议室开大会时，笔者与其余上一辈七人众看到上一代航行器，心中满是感慨：“这东西到底是怎么做出来的？”虽然很多问题师弟们会在我们之后再次遇到，但开山之作的困难与痛苦，也许师弟们再也难以理解。

笔者在美国之时，单人负担了图像与总控两个最重要的部分，同兄弟团队基地相比，笔者做了相当于他们四个人的工作量。当时为了抢占调试场地，笔者需要在早上六点乘坐第一班车赶往赛场。而每天夜晚，大家平均是两点睡觉，笔者一般要比他们更晚两个小时睡觉，更有甚者到了凌晨五点睡觉，六点起床的情况。除了负责所有软件部分内容外，在赛场上笔者也经常担任翻译官的角色。那段时间的状态，说是把自己的精力完全榨干也是一点都不夸张。这段时间为了写这套代码，好像回到了那段艰苦又有价值的岁月中，看来笔者还是怀念那时的自己啊。

然而时代已然不是我的了。

至此，笔者已经完成了之前所有关于主控程序的设想，完成了 NACIT 队伍的主控框架与视觉系统框架，并给师弟们写了文档若干，相信已经将交接工作做到了笔者的极致，笔者对于 IAUVC 比赛再无遗憾。

也许再无遗憾这种话说的太早。如果可以的话，希望师弟们可以将这套代码完善并好好使用，用这套代码获取我们之前没有完成的成绩，也许才算了了笔者最大的遗憾。

也许你们见不到凌晨四点的洛杉矶，但你们可以见到凌晨五点的圣地亚哥。

By SoftWare Leader of NACIT2016

GRQ