1. 阅读指引

此文档详细描述了 spp 各个插件 example，以及如何运行这些 example，以加深对 spp 的理解。一共包括了如下示例：

- Simple: 直接在 worker 进程中的的 spp_handle_process 方法中，将收到数据 echo 回去，不使用异步或微线程框架
- 微线程框架的各 example
  - MT Tcp Send Recv: 消息处理微线程使用 mt_tcpsendrcv 方法，与后端进行 Tcp 通信，一发一收
  - MT Udp Send Recv: 消息处理微线程使用 mt_udpsendrcv 方法，与后端进行 Udp 通信，一发一收
  - MT Exec All Task：消息处理微线程里，建立多个子微线程与后端进行 Tcp/Udp 通信，父微线程等待子微线程完成工作后，将从后端收到的包返回给用户。
  - MT Start Thread：消息处理微线程里，建立一个微线程去完成旁路逻辑工作。消息

处理微线程不关心新建微线程工作结果，继续自己的逻辑处理

- 单独使用微线程库 Mt Alone: 微线程库是支撑 spp 微线程框架的底层库，它其实是一个独立于 spp 的库，可以脱离 spp 使用。示例中，使用微线程库创建了多个微线程，独立的去完成各类 Tcp/Udp 通信。
- 异步框架的各 example
  - Single State & Action: 只有一个 State，State 中只有一个 Action 的简单示例，Action 中将接收到的包转发到后端 Udp Server，然后将收到的 Udp Server 回包直接返回给用户
  - Multiple State & Action: 多个 State，State 中有多个 Action 的示例。每个 State 中，Action 与后端 Tcp/Udp Server 进行通信，并将收到的包返回给用户。

编译 so 的机器环境如下：

- 操作系统: Linux TENCENT64.site 3.10.104-1-tlinux2_kvm_guest-0021.tl1 #1 SMP Thu Oct 27 18:41:41 CST 2016 x86_64 x86_64 x86_64 GNU/Linux
- gcc/g++: gcc (GCC) 4.4.6 20110731 (Red Hat 4.4.6-4)

所有示例代码，相关配置，以及编译后的 so 都在发布包的 example 目录下。

运行时如何关闭

example 中各个目录和重要文件介绍如下：

```
.
|-- Makefile
|-- async_frame              #异步框架各 example 代码
|   |-- action               #多个 State/Action 用法
|   `-- echo                 #单个 State/Action 用法
|-- bin                      #编译后的 so 存放目录
|-- create_svr.sh
|-- etc                      #各 example 配置存放目录
|   |-- async_frame          #异步框架各 example 配置目录
|   |   |-- action
|   |   `-- echo
|   |-- mt_frame             #微线程框架各 example 配置目录
|   |   |-- mt_exec_all_task
|   |   |-- mt_start_thread
|   |   |-- mt_tcpsendrcv
|   |   `-- mt_udpsendrcv
|   `-- simple
|-- mt_alone                 #单独使用微线程库 example 代码
|-- mt_frame                 #微线程框架各 example 代码
|   |-- mt_exec_all_task
|   |-- mt_start_thread
|   |-- mt_tcpsendrcv
|   `-- mt_udpsendrcv
|-- simple                   #不使用任何框架的 simple example 代码
`-- tools                    #提供了开启一些本地 tcp/udp 服务的 python 脚本
```

```
|-- python_tcp_server.py          #tcp 服务，将所收到的字符逆序返回
|-- python_udp_echo_server.py     #udp 服务，echo 所有收到的字符
`-- python_udp_server.py          #udp 服务，将所收到的字符逆序返回
```

2. Simple

在这个示例中，Worker 简单的在 spp_handle_process 中将收到的消息 echo 回去，没有使用异步框架或微线程框架。

对应 so: simple.so

2.1. 编译

如果你要自己编译，编译命令如图所示，编译后 module/Example/bin 就多了一个 simple.so 文件。

```
[~/spp_release/spp/example/simple]$ make
Compiling echo_example.cpp ==> obj/echo_example.o...
g++ -I../../include/spp_incl/ -I../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe -fno-ident
-MMD -D_GNU_SOURCE -D_REENTRANT -c echo_example.cpp -o obj/echo_example.o

Building obj/echo_example.o ==> simple.so...
g++ -o simple.so obj/echo_example.o -g -fPIC -shared -Wall -O2 -pipe -fno-ident -MMD
-D_GNU_SOURCE -D_REENTRANT
install *.so ../bin
[~/spp_release/spp/example/simple]$ ll ../bin
total 36
-rwxr-xr-x 1 sibinluo sibinluo 36340 Jun 27 21:08 simple.so
```

2.2. 运行

首先将 example/etc/simple 文件夹中对应 simple 的配置拷贝到 etc 目录下。然后使用 yaml_tool 进行配置转换，yaml_tool 的介绍和使用请参见 http://km.oa.com/group/657/articles/show/176655 。

使用 spp_ctrl 拉起来 worker，然后控制台打印 worker 启动成功的消息。

Proxy/Worker 拉起来之后， 由于默认配置的是 proxy 监听 9902 端口，因此使用 nc 命令与本机 IP 地址（示例中是 10.123.2.243）的 tcp/udp 9902 端口通信，发现任何输入都 echo 回来了。

```
[~/spp_release/spp]$ cp example/etc/simple/* etc/
[~/spp_release/spp]$ cd bin
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
```

```
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
[~/spp_release/spp/bin]$
Proxy[22493] init...
[22493] Proxy[22493] Bind On [udp][9902]...
[22493] Proxy[22493] Bind On [tcp][9902]...
[22493] Proxy[22493] [Shm]Proxy->WorkerGroup[1] [16MB]...
[22493] Proxy[22493] [Shm]WorkerGroup[1]->Proxy [16MB]...
[22493] Proxy[22493] Load module[../example/bin/simple.so] etc[]...
[WARRING]spp_handle_report not implemented.
[22493] Proxy[22493] OK!

Worker[22495] init...
Worker[22495] Groupid = 1  L5us = 0 shm_fifo = 0
[22495] Worker[22495] [Shm]Proxy->Worker [16MB]
[22495] Worker[22495] [Shm]Worker->Proxy [16MB]
[22495] Worker[22495] Load module[../example/bin/simple.so] etc[]
[22495] call spp_handle_init ...
[22495] Worker[22495] OK!
[~/spp_release/bin]$ nc 10.123.2.243 9902
hello
hello
world
world
^C
[~/spp_release/bin]$ nc -u 10.123.2.243 9902
hello
hello
world
world
^C
[~/spp_release/bin]$
```

3. 微线程框架

3.1. MT Tcp Send Recv

这个示例，展示了微线程与后端服务 TCP 通信的使用。

微线程框架中，对于每一个消息的处理都会开一个微线程去处理，在微线程中，我们需要调用框架提供的各类 IO 方法，诸如此例中的 mt_tcpsendrcv，这些方法不会阻塞掉整个进程，

而是在 IO 未就绪时将对应微线程切换出去，运行其他可运行的微线程。

示例中，Worker 将收到的消息，调用微线程框架提供的 mt_tcpsendrcv 方法发送到本地 127.0.0.1 的 tcp 5574 端口上，然后将 127.0.0.1:5574 的返回消息返回给用户。

代码中提供了一个 python tcp 服务器，可以将收到的字符逆序返回，运行时我们启动此服务器，工作在 127.0.0.1 的 5574 端口上，nc 观察 spp 的返回是否逆序。

对应 so：mt_tcp.so

3.1.1. 编译

也是在对应源代码文件夹下执行 make 即可，以下是我在自己编译机上执行的命令和结果。

```
[~/spp_release/spp/example/mt_frame/mt_tcpsendrcv]$ make
Compiling mt_tcpsendrcv.cpp ==> obj/mt_tcpsendrcv.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c mt_tcpsendrcv.cpp -o obj/mt_tcpsendrcv.o

Building obj/mt_tcpsendrcv.o ==> mt_tcpsendrcv.so...
g++ -o mt_tcpsendrcv.so obj/mt_tcpsendrcv.o -g -fPIC -shared -Wall -O2 -pipe -fno-ident -MMD
-D_GNU_SOURCE -D_REENTRANT
install *.so ../../bin
[~/spp_release/spp/example/mt_frame/tcpsendrcv]$ ll ../../bin
total 96
-rwxr-xr-x 1 sibinluo sibinluo 58590 Jun 27 21:16 mt_tcpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 36340 Jun 27 21:08 simple.so
[~/spp_release/spp/example/mt_frame/tcpsendrcv]$
```

3.1.2. 运行

首先我们需要启动 python tcp 服务器: python_tcp_server.py 5574。

然后使用 spp_ctrl 拉起 worker 和 proxy，最后我们使用 nc 命令，分别尝试 tcp/udp 连接 spp_proxy 监听的本机（10.123.2.243）9902 端口，发现无论输入什么，都逆序回来了。

```
[~/spp_release/spp/bin]$ cp ../example/etc/mt_frame/tcpsendrcv/* ../etc
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ../example/tools/python_tcp_server.py 5574 &
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
```

```
[~/spp_release/spp/bin]$
Proxy[25923] init...
[25923] Proxy[25923] Bind On [udp][9902]...
[25923] Proxy[25923] Bind On [tcp][9902]...
[25923] Proxy[25923] [Shm]Proxy->WorkerGroup[1] [16MB]...
[25923] Proxy[25923] [Shm]WorkerGroup[1]->Proxy [16MB]...
[25923] Proxy[25923] Load module[../example/bin/mt_tcpsendrcv.so] etc[]...
[WARRING]spp_handle_report not implemented.
[25923] Proxy[25923] OK!

Worker[25925] init...
Worker[25925] Groupid = 1  L5us = 0 shm_fifo = 0
[25925] Worker[25925] [Shm]Proxy->Worker [16MB]
[25925] Worker[25925] [Shm]Worker->Proxy [16MB]
[25925] Worker[25925] Load module[../example/bin/mt_tcpsendrcv.so] etc[]
[25925] call spp_handle_init ...
[25925] Worker[25925] OK!

[~/spp_release/spp/bin]$ nc 10.123.2.243 9902
hello

olleh^C
[~/spp_release/spp/bin]$ nc -u 10.123.2.243 9902
helo

oleh
^C
[~/spp_release/bin]$
```

3.2. MT Udp Send Recv

此示例和 SPP Tcp Relay case 几乎雷同，不同之处在于我们在消息处理微线程中，调用 mt_udpsendrcv 将消息中继到本地的 5574 端口上。

编译和运行也是几乎一样的，见下面的我在自己机器上的执行结果。

对应 so: mt_udp.so

3.2.1. 编译

```
[~/spp_release/spp/example/mt_frame/mt_udpsendrcv]$ make
Compiling mt_udpsendrcv.cpp ==> obj/mt_udpsendrcv.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c mt_udpsendrcv.cpp -o obj/mt_udpsendrcv.o
```

```
Building obj/mt_udpsendrcv.o ==> mt_udpsendrcv.so...
g++ -o mt_udpsendrcv.so obj/mt_udpsendrcv.o -g -fPIC -shared -Wall -O2 -pipe -fno-ident -MMD
-D_GNU_SOURCE -D_REENTRANT
install *.so ../../bin
[~/spp_release/spp/example/mt_frame/udpsendrcv]$ ll ../../bin
total 156
-rwxr-xr-x 1 sibinluo sibinluo 58590 Jun 27 21:16 mt_tcpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 58233 Jun 27 21:29 mt_udpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 36340 Jun 27 21:08 simple.so
[~/spp_release/spp/example/mt_frame/udpsendrcv]$
```

3.2.2. 运行

启动 spp 服务后，nc 与本机(此例中是 10.123.2.243)端口通信，观察返回。

```
[~/spp_release/spp/bin]$ cp ../example/etc/mt_frame/udpsendrcv/* ../etc/
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ../example/tools/python_udp_server.py 5574 &
[2] 26926
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
[~/spp_release/spp/bin]$
Proxy[26961] init...
[26961] Proxy[26961] Bind On [udp][9902]...
[26961] Proxy[26961] Bind On [tcp][9902]...
[26961] Proxy[26961] [Shm]Proxy->WorkerGroup[1] [16MB]...
[26961] Proxy[26961] [Shm]WorkerGroup[1]->Proxy [16MB]...
[26961] Proxy[26961] Load module[../example/bin/mt_udpsendrcv.so] etc[]...
[WARRING]spp_handle_report not implemented.
[26961] Proxy[26961] OK!

Worker[26963] init...
Worker[26963] Groupid = 1  L5us = 0 shm_fifo = 0
[26963] Worker[26963] [Shm]Proxy->Worker [16MB]
[26963] Worker[26963] [Shm]Worker->Proxy [16MB]
[26963] Worker[26963] Load module[../example/bin/mt_udpsendrcv.so] etc[]
[26963] call spp_handle_init ...
[26963] Worker[26963] OK!

[~/spp_release/spp/bin]$ nc 10.123.2.243 9902
hello
```

```
olleh^C
[~/spp_release/spp/bin]$ nc -u 10.123.2.243 9902
nihao


oahin^C
[~/spp_release/spp/bin]$
```

3.3. MT Exec All Task

这个示例，展示了微线程框架中子微线程的 exec-wait 用法，也就是在一个微线程中，开一个或多个子微线程，去完成工作。父微线程等待子微线程工作完成后，继续逻辑处理。

微线程框架中，对于每一个消息的处理都会开一个微线程去处理，在微线程中，我们需要调用框架提供的各类 IO 方法，诸如此例中的 mt_tcpsendrcv，这些方法不会阻塞掉整个进程，而是在 IO 未就绪时将对应微线程切换出去，运行其他可运行的微线程。

我们继承微线程框架提供的线程接口 IMtTask，定义了一个子线程类 ExampleTask；实现了 Process 方法，用于将接收到的消息中继到本机的 udp 5574 端口。

然后在消息处理中，我们实例化了两个子线程类 ExampleTask 对象，消息处理父微线程等待这两个子线程运行完成后，将每个线程获取到的输出合并之后返回给用户。

对应 so：mt_exec_wait_sub_task

3.3.1. 编译

```
[~/spp_release/spp/example/mt_frame/mt_exec_all_task]$ make

Compiling mt_exec_all_task.cpp ==> obj/mt_exec_all_task.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c mt_exec_all_task.cpp -o obj/mt_exec_all_task.o

Building obj/mt_exec_all_task.o ==> mt_exec_all_task.so...
g++ -o mt_exec_all_task.so obj/mt_exec_all_task.o -g -fPIC -shared -Wall -O2 -pipe -fno-ident
-MMD -D_GNU_SOURCE -D_REENTRANT
install *.so ../../bin
[~/spp_release/spp/example/mt_frame/exec_all_task]$ ll ../../bin
total 248
-rwxr-xr-x 1 sibinluo sibinluo 91850 Jun 27 21:35 mt_exec_all_task.so
-rwxr-xr-x 1 sibinluo sibinluo 58590 Jun 27 21:16 mt_tcpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 58233 Jun 27 21:29 mt_udpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 36340 Jun 27 21:08 simple.so
[~/spp_release/spp/example/mt_frame/exec_all_task]$
```

3.3.2. 运行

使用 nc 命令连本机（此例中是 10.123.2.243）接 tcp/udp 9902 端口，我们发现无论输入什么，都逆序返回了两次。

```
[~/spp_release/spp/bin]$ cp  ../example/etc/mt_frame/exec_all_task/* ../etc/
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
[~/spp_release/spp/bin]$
Proxy[27681] init...
[27681] Proxy[27681] Bind On [udp][9902]...
[27681] Proxy[27681] Bind On [tcp][9902]...
[27681] Proxy[27681] [Shm]Proxy->WorkerGroup[1] [16MB]...
[27681] Proxy[27681] [Shm]WorkerGroup[1]->Proxy [16MB]...
[27681] Proxy[27681] Load module[../example/bin/mt_exec_all_task.so] etc[]...
[WARRING]spp_handle_report not implemented.
[27681] Proxy[27681] OK!

Worker[27683] init...
Worker[27683] Groupid = 1  L5us = 0 shm_fifo = 0
[27683] Worker[27683] [Shm]Proxy->Worker [16MB]
[27683] Worker[27683] [Shm]Worker->Proxy [16MB]
[27683] Worker[27683] Load module[../example/bin/mt_exec_all_task.so] etc[]
[27683] call spp_handle_init ...
[27683] Worker[27683] OK!

[~/spp_release/spp/bin]$ nc 10.123.2.243 9902
hello

olleh
olleh^C
[~/spp_release/spp/bin]$ nc -u 10.123.2.243 9902
nihao

oahin
oahin^C
[~/spp_release/spp/bin]$ $
```

3.4 MT Start Thread

这个示例，展示了微线程框架中新建微线程去完成任务的 start_thread 用法，也就是在一个微线程中,新建一个微线程去做某些工作，而完全不用等待该微线程的返回。原来的线程继续自己的逻辑。

在示例代码中，我们新建一个微线程将接收的消息转发到本地 udp 端口 5574，并接收回包。消息处理不关心该微线程的接收结果，直接将接收的消息 echo 回去。等到本地 udp 端口 5574 的回包到来，会打印到标准输出中。

对应 so: mt_start_thread.so

3.4.1. 编译

```
[~/spp_release/spp/example/mt_frame/mt_start_thread]$ make
Compiling start_thread.cpp ==> obj/mt_start_thread.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c mt_start_thread.cpp -o obj/mt_start_thread.o

Building obj/mt_start_thread.o ==> mt_start_thread.so...
g++ -o mt_start_thread.so obj/mt_start_thread.o -g -fPIC -shared -Wall -O2 -pipe -fno-ident
-MMD -D_GNU_SOURCE -D_REENTRANT
install *.so ../../bin
[~/spp_release/spp/example/mt_frame/start_thread]$ ll ../../bin
total 308
-rwxr-xr-x 1 sibinluo sibinluo 91850 Jun 27 21:35 mt_exec_all_task.so
-rwxr-xr-x 1 sibinluo sibinluo 60231 Jun 27 21:40 mt_start_thread.so
-rwxr-xr-x 1 sibinluo sibinluo 58590 Jun 27 21:16 mt_tcpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 58233 Jun 27 21:29 mt_udpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 36340 Jun 27 21:08 simple.so
```

3.4.2. 运行

启动 spp 服务后，使用 nc 与本机（此例中是 10.123.2.243）的 spp proxy 监听端口 9902 进行通信。

```
[~/spp_release/spp/bin]$ cp  ../example/etc/mt_frame/start_thread/* ../etc/
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
[~/spp_release/spp/bin]$
```

```
Proxy[29434] init...
[29434] Proxy[29434] Bind On [udp][9902]...
[29434] Proxy[29434] Bind On [tcp][9902]...
[29434] Proxy[29434] [Shm]Proxy->WorkerGroup[1] [16MB]...
[29434] Proxy[29434] [Shm]WorkerGroup[1]->Proxy [16MB]...
[29434] Proxy[29434] Load module[../example/bin/mt_start_thread.so] etc[]...
[WARRING]spp_handle_report not implemented.
[29434] Proxy[29434] OK!

Worker[29436] init...
Worker[29436] Groupid = 1  L5us = 0 shm_fifo = 0
[29436] Worker[29436] [Shm]Proxy->Worker [16MB]
[29436] Worker[29436] [Shm]Worker->Proxy [16MB]
[29436] Worker[29436] Load module[../example/bin/mt_start_thread.so] etc[]
[29436] call spp_handle_init ...
[29436] Worker[29436] OK!
[~/spp_release/spp/bin]$ nc 10.123.2.243 9902
hello
hello
new thread handle_msg rcvd:
olleh
sdfs
sdfs
new thread handle_msg rcvd:
sfds
^C
```

4. 单独使用微线程库 Mt Alone

本示例展示了微线程库单独使用的场景。微线程库是支撑微线程框架的底层库，hook 了网络 IO 各类方法。可以单独使用微线程库，创建各类微线程任务去处理网络 IO，库会自动调度这些微线程。

示例代码中，有三类微线程类：

- UdpSndRcvTask： udp 收发微线程，与本地 udp 端口 5574 进行通信，一收一发
- TcpSndRcvTask： tcp 收发微线程，与本地 tcp 端口 5574 进行通信，一收一发
- ApiVerifyTask: 验证微线程库是否可用，向标准输出打印信息

4.1. 编译

```
[~/spp_release/spp/example/mt_alone]$ make
g++ -g -Wall -Wno-write-strings -Werror -L../../bin/lib/  -lmt -ldl -Wl,-rpath,../../bin/lib/
-o mt_alone mt_alone.o
```

```
install mt_alone ../bin
[~/spp_release/spp/example/mt_alone]$ ll ../bin
total 372
-rwxr-xr-x 1 sibinluo sibinluo 62112 Jun 27 21:55 mt_alone
-rwxr-xr-x 1 sibinluo sibinluo 91850 Jun 27 21:35 mt_exec_all_task.so
-rwxr-xr-x 1 sibinluo sibinluo 60231 Jun 27 21:45 mt_start_thread.so
-rwxr-xr-x 1 sibinluo sibinluo 58590 Jun 27 21:16 mt_tcpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 58233 Jun 27 21:29 mt_udpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo 36340 Jun 27 21:08 simple.so
[~/spp_release/spp/example/mt_alone]$
```

4.2. 运行

编译出来是一个可执行文件 mt_alone，直接执行。该文件会与后端通信，将接收到的消息直接打印到标准输出

```
[~/spp_release/spp/example/bin]$ ./mt_alone
This is the api verify task!!!
UdpSndRcvTask recvd: dlrow olleh
TcpSndRcvTask recvd: dlrow olleh

This is the api verify task!!!
UdpSndRcvTask recvd: dlrow olleh
TcpSndRcvTask recvd: dlrow olleh

^C
[~/spp_release/spp/example/bin]$
```

5. 异步框架

5.1. Single State & Action

此示例展示了 spp 异步编程框架的最简单用法：只有一个 State 和一个 Action。

我们继承了 IAction 类：CGetInfo。它简单的将中继消息对象类 CMsg 中用户输入的字节，打包成对通信后台要发送的消息，然后等待通信后台相同字节长度的回包。

我们定义了一个 State: CGetState

- CGetState：包含了一个 CActionInfo 对象。
  - pAction1：与本地的 tcp 5575 端口通信，一发一收。负责打包，接收，处理回包的 IAction 指针指向 CGetInfo 类对象。

我们设置了 SPP 框架执行 CGetState，并发向本地 udp 5575 端口一发一收，收包存储在 CMsg 消息对象中。然后将接收到的字节，发送给用户。

本地 udp 5575 我们用 python_udp_echo_server.py 起来一个 echo server，这样就形成了一个异步 echo 的闭环。

对应 so: async_echo.so

5.1.1. 编译

```
[~/spp_release/spp/example/async_frame/echo]$ make
Compiling GetInfo.cpp ==> obj/GetInfo.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c GetInfo.cpp -o obj/GetInfo.o
GetInfo.cpp:          In          member          function          'virtual          int
CGetInfo::HandleProcess(SPP_ASYNCFRAME::CAsyncFrame*,       const       char*,       int,
SPP_ASYNCFRAME::CMsgBase*)':
GetInfo.cpp:60: warning: deprecated conversion from string constant to 'char*'

Compiling GetState.cpp ==> obj/GetState.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c GetState.cpp -o obj/GetState.o

Compiling msg.cpp ==> obj/msg.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c msg.cpp -o obj/msg.o

Compiling service.cpp ==> obj/service.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c service.cpp -o obj/service.o
service.cpp:     In     function     'int     OverloadProcess(SPP_ASYNCFRAME::CAsyncFrame*,
SPP_ASYNCFRAME::CMsgBase*)':
service.cpp:45: warning: deprecated conversion from string constant to 'char*'

Building obj/GetInfo.o obj/GetState.o obj/msg.o obj/service.o ==> async_echo.so...
g++ -o async_echo.so obj/GetInfo.o obj/GetState.o obj/msg.o obj/service.o -g -fPIC -shared
-Wall -O2 -pipe -fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT
install *.so  ../../bin
[~/spp_release/spp/example/async_frame/echo]$ ll ../../bin
total 600
-rwxr-xr-x 1 sibinluo sibinluo 231821 Jun 27 22:14 async_echo.so
-rwxr-xr-x 1 sibinluo sibinluo  62112 Jun 27 21:55 mt_alone
-rwxr-xr-x 1 sibinluo sibinluo  91850 Jun 27 21:35 mt_exec_all_task.so
-rwxr-xr-x 1 sibinluo sibinluo  60231 Jun 27 21:45 mt_start_thread.so
-rwxr-xr-x 1 sibinluo sibinluo  58590 Jun 27 21:16 mt_tcpsendrcv.so
```

```
-rwxr-xr-x 1 sibinluo sibinluo  58233 Jun 27 21:29 mt_udpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo  36340 Jun 27 21:08 simple.so
[~/spp_release/spp/example/async_frame/echo]$
```

5.1.2. 运行

首先需要确保 python_udp_echo_server.py 监听了 5575 端口，然后拉起 spp。最后使用 nc 命令与本机（此例中 IP 为 10.123.2.243）的 spp proxy 监听端口 9902 进行通信。

```
[~/spp_release/spp/bin]$ cp ../example/etc/async_frame/echo/* ../etc
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ../example/tools/python_udp_echo_server.py 5575 &
[3] 869
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
[~/spp_release/spp/bin]$
Proxy[  902] init...
[902] Proxy[  902] Bind On [udp][9902]...
[902] Proxy[  902] Bind On [tcp][9902]...
[902] Proxy[  902] [Shm]Proxy->WorkerGroup[1] [16MB]...
[902] Proxy[  902] [Shm]WorkerGroup[1]->Proxy [16MB]...
[902] Proxy[902] Load module[../example/bin/async_echo.so] etc[]...
[WARRING]spp_handle_report not implemented.
[902] Proxy[902] OK!

Worker[  904] init...
Worker[  904] Groupid = 1  L5us = 0 shm_fifo = 0
[904] Worker[  904] [Shm]Proxy->Worker [16MB]
[904] Worker[  904] [Shm]Worker->Proxy [16MB]
[904] Worker[  904] Load module[../example/bin/async_echo.so] etc[]
[904] call spp_handle_init ...
[904] Worker[  904] OK!

[~/spp_release/spp/bin]$ nc 10.123.2.243 9902
hello

GetInfo Recv: hello
^C
[~/spp_release/spp/bin]$ nc -u 10.123.2.243 9902
world

GetInfo Recv: world
```

```
^C
[~/spp_release/spp/bin]$
```

5.2. Multiple State & Action

上面的 Async Echo 用例，展示了 spp 异步编程框架最简单的情况，只有一个 State 和一个 Action，在此用例中，我们更近一步，创建多个 State 和 Action。

我们继承了两个 IAction 类：CGetInfo， CGetExtInfo。这两个 Action 类都是简单的将中继消息对象类 CMsg 中用户输入的字节，打包成对通信后台要发送的消息，然后等待通信后台相同字节长度的回包。

我们定义了两个 State: CGetState，CGetState2。

- CGetState：包含了两个 CActionInfo 对象。
  - pAction1：与本地的 tcp 5574 端口通信，一发一收。负责打包，接收，处理回包的 IAction 指针指向 CGetInfo 类对象。
  - pAction2：与本地的 udp 5574 端口通信，一发一收。负责打包，接收，处理回包的 IAction 指针指向 CGetExtInfo 类对象。
- CGetState2: 包含了一个 CActionInfo 对象。
  - pAction3: 与本地的 udp 5574 端口通信，一发一收。负责打包，接收，处理回包的 IAction 指针指向 CGetExtInfo 类对象。

我们设置了 SPP 框架先执行 CGetState，并发向本地 tcp/udp 5574 端口一发一收，收包存储在 CMsg 消息对象中。然后在执行 CGetState2,回包也存储在 CMsg 消息对象中。最后将消息对象中存储的接收到的字节，发送给用户。

对应 so: async_echo.so

5.2.1. 编译

```
[~/spp_release/spp/example/async_frame/action]$ make

Compiling GetExtInfo.cpp ==> obj/GetExtInfo.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c GetExtInfo.cpp -o obj/GetExtInfo.o
GetExtInfo.cpp:          In          member          function          'virtual          int
CGetExtInfo::HandleProcess(SPP_ASYNCFRAME::CAsyncFrame*,      const      char*,      int,
SPP_ASYNCFRAME::CMsgBase*)':
GetExtInfo.cpp:62: warning: deprecated conversion from string constant to 'char*'

Compiling GetInfo.cpp ==> obj/GetInfo.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c GetInfo.cpp -o obj/GetInfo.o
GetInfo.cpp:          In          member          function          'virtual          int
```

```
CGetInfo::HandleProcess(SPP_ASYNCFRAME::CAsyncFrame*, const char*, int,
SPP_ASYNCFRAME::CMsgBase*)':
GetInfo.cpp:60: warning: deprecated conversion from string constant to 'char*'

Compiling GetState.cpp ==> obj/GetState.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c GetState.cpp -o obj/GetState.o

Compiling GetState2.cpp ==> obj/GetState2.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c GetState2.cpp -o obj/GetState2.o

Compiling msg.cpp ==> obj/msg.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c msg.cpp -o obj/msg.o

Compiling service.cpp ==> obj/service.o...
g++ -I../../../include/spp_incl/ -I../../../include  -I./ -g -fPIC -shared -Wall -O2 -pipe
-fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT -c service.cpp -o obj/service.o
service.cpp:    In    function    'int    OverloadProcess(SPP_ASYNCFRAME::CAsyncFrame*,
SPP_ASYNCFRAME::CMsgBase*)':
service.cpp:46: warning: deprecated conversion from string constant to 'char*'

Building   obj/GetExtInfo.o   obj/GetInfo.o   obj/GetState.o   obj/GetState2.o   obj/msg.o
obj/service.o ==> async_action.so...
g++ -o async_action.so obj/GetExtInfo.o obj/GetInfo.o obj/GetState.o obj/GetState2.o obj/msg.o
obj/service.o -g -fPIC -shared -Wall -O2 -pipe -fno-ident -MMD -D_GNU_SOURCE -D_REENTRANT
install *.so ../../bin
[~/spp_release/spp/example/async_frame/action]$ ll ../../bin
total 912
-rwxr-xr-x 1 sibinluo sibinluo 317479 Jun 27 22:19 async_action.so
-rwxr-xr-x 1 sibinluo sibinluo 231821 Jun 27 22:14 async_echo.so
-rwxr-xr-x 1 sibinluo sibinluo  62112 Jun 27 21:55 mt_alone
-rwxr-xr-x 1 sibinluo sibinluo  91850 Jun 27 21:35 mt_exec_all_task.so
-rwxr-xr-x 1 sibinluo sibinluo  60231 Jun 27 21:45 mt_start_thread.so
-rwxr-xr-x 1 sibinluo sibinluo  58590 Jun 27 21:16 mt_tcpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo  58233 Jun 27 21:29 mt_udpsendrcv.so
-rwxr-xr-x 1 sibinluo sibinluo  36340 Jun 27 21:08 simple.so
[~/spp_release/spp/example/async_frame/action]$
```

5.2.2. 运行

需要 python tcp, udp 服务器都开启，监听在 5574 端口上面，然后 nc 到本机（此例中是

10.123.2.243）的 tcp/udp 9902 端口，看返回是否正确。

```
[~/spp_release/spp/bin]$ cp ../example/etc/async_frame/action/* ../etc
[~/spp_release/spp/bin]$ ./yaml_tool x
Instance name: spp
extract cpu num: 12!
Transform OK!
[~/spp_release/spp/bin]$ ./spp_ctrl ../etc/spp_ctrl.xml
[~/spp_release/spp/bin]$
Proxy[ 1743] init...
[1743] Proxy[ 1743] Bind On [udp][9902]...
[1743] Proxy[ 1743] Bind On [tcp][9902]...
[1743] Proxy[ 1743] [Shm]Proxy->WorkerGroup[1] [16MB]...
[1743] Proxy[ 1743] [Shm]WorkerGroup[1]->Proxy [16MB]...
[1743] Proxy[1743] Load module[../example/bin/async_action.so] etc[]...
[WARRING]spp_handle_report not implemented.
[1743] Proxy[1743] OK!

Worker[ 1745] init...
Worker[ 1745] Groupid = 1  L5us = 0 shm_fifo = 0
[1745] Worker[ 1745] [Shm]Proxy->Worker [16MB]
[1745] Worker[ 1745] [Shm]Worker->Proxy [16MB]
[1745] Worker[ 1745] Load module[../example/bin/async_action.so] etc[]
[1745] call spp_handle_init ...
[1745] Worker[ 1745] OK!

[~/spp_release/spp/bin]$ nc 10.123.2.243 9902
hello

GetExtInfo Recv:
olleh
GetInfo Recv:
olleh
GetExtInfo Recv:
olleh^C
[~/spp_release/spp/bin]$ nc -u 10.123.2.243 9902
world

GetExtInfo Recv:
dlrow
GetInfo Recv:
dlrow
GetExtInfo Recv:
dlrow^C
[~/spp_release/spp/bin]$
```