**spserver 是一个实现了半同步/半异步(Half-Sync/Half-Async)和领导者/追随者(Leader/Follower) 模式的服务器框架，能够简化 TCP server 的开发工作。   
spserver 使用 c++ 实现，目前实现了以下功能：   
1.封装了 TCP server 中接受连接的功能；   
2.使用非阻塞型I/O和事件驱动模型，由主线程负责处理所有 TCP 连接上的数据读取和发送，因此连接数不受线程数的限制；   
3.主线程读取到的数据放入队列，由一个线程池处理实际的业务。   
4.一个 http 服务器框架，即嵌入式 web 服务器（请参考：**[**SPWebServer：一个基于 SPServer 的 web 服务器框架**](http://iunknown.javaeye.com/blog/78561)**）**   
  
0.6 版本之前只包含 Half-Sync/Half-Async 模式的实现，0.6 版本开始包含 Leader/Follower 模式的实现   
0.7 版本开始支持 ssl 。把 socket 相关的操作抽象为一个 IOChannel 层，关于 openssl 的部分单独实现为一个 plugin 的形式，对于不使用 ssl 功能的用户，不需要引入 ssl 相关的头文件和库。   
0.7.5 增加了一个 sptunnel 程序，是一个通用的 ssl proxy 。类似 stunnel 。   
0.9.0 移植 spserver 到 windows 平台，需要在 windows 下编译 libevent 和 pthread 。   
0.9.1 在 windows 平台，去掉了对 libevent 和 pthread 依赖，完全使用 iocp 和 windows 的线程机制实现了半同步半异步的框架。   
0.9.2 移植了所有的功能到 windows 平台，同时新增加了 xyssl 的插件。   
  
主页：   
<http://code.google.com/p/spserver/>   
  
源代码下载：   
[http://spserver.googlecode.com/files/spserver-0.6.src.tar.gz](http://freshmeat.net/redir/spserver/68862/url_tgz/spserver-0.3.src.tar.gz)   
<http://code.google.com/p/spserver/downloads/list>   
  
  
在实现并发处理多事件的应用程序方面，有如下两种常见的编程模型：   
ThreadPerConnection的多线程模型和事件驱动的单线程模型。   
  
**ThreadPerConnection的多线程模型**   
**优点：**简单易用，效率也不错。在这种模型中，开发者使用同步操作来编写程序，比如使用阻塞型I/O。使用同步操作的程序能够隐式地在线程的运行堆栈中维护应用程序的状态信息和执行历史，方便程序的开发。   
**缺点：**没有足够的扩展性。如果应用程序只需处理少量的并发连接，那么对应地创建相应数量的线程，一般的机器都还能胜任；但如果应用程序需要处理成千上万个连接，那么为每个连接创建一个线程也许是不可行的。   
  
**事件驱动的单线程模型**   
**优点：**扩展性高，通常性能也比较好。在这种模型中，把会导致阻塞的操作转化为一个异步操作，主线程负责发起这个异步操作，并处理这个异步操作的结果。由于所有阻塞的操作都转化为异步操作，理论上主线程的大部分时间都是在处理实际的计算任务，少了多线程的调度时间，所以这种模型的性能通常会比较好。   
**缺点：**要把所有会导致阻塞的操作转化为异步操作。一个是带来编程上的复杂度，异步操作需要由开发者来显示地管理应用程序的状态信息和执行历史。第二个是目前很多广泛使用的函数库都很难转为用异步操作来实现，即是可以用异步操作来实现，也将进一步增加编程的复杂度。   
  
**并发系统通常既包含异步处理服务，又包含同步处理服务。系统程序员有充分的理由使用异步特性改善性能。相反，应用程序员也有充分的理由使用同步处理简化他们的编程强度。**   
  
**针对这种情况，ACE 的作者提出了 半同步/半异步 （Half-Sync/Half-Async） 模式。**

**引用**

[《POSA2》](http://www.douban.com/subject/1137259/)上对这个模式的描述如下：   
半同步/半异步 体系结构模式将并发系统中的异步和同步服务处理分离，简化了编程，同时又没有降低性能。该模式介绍了两个通信层，一个用于异步服务处理，另一个用于同步服务处理。   
  
**目标：**   
需要同步处理的简易性的应用程序开发者无需考虑异步的复杂性。同时，必须将性能最大化的系统开发者不需要考虑同步处理的低效性。让同步和异步处理服务能够相互通信，而不会使它们的编程模型复杂化或者过度地降低它们的性能。   
  
**解决方案：**   
将系统中的服务分解成两层，同步和异步，并且在它们之间增加一个排队层协调异步和同步层中的服务之间的通信。在独立线程或进程中同步地处理高层服务（如耗时长的数据库查询或文件传输），从而简化并发编程。相反，异步地处理底层服务（如从网络连接上读取数据），以增强性能。如果驻留在相互独立的同步和异步层中的服务必须相互通信或同步它们的处理，则应允许它们通过一个排队层向对方传递消息。   
  
模式原文：Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-structured Concurrent I/O   
<http://www.cs.wustl.edu/~schmidt/PDF/HS-HA.pdf>   
中文翻译：<http://blog.chinaunix.net/u/31756/showart_245841.html>   
  
如果上面关于 半同步/半异步 的说明过于抽象，那么可以看一个《POSA2》中提到的例子：   
**许多餐厅使用 半同步/半异步 模式的变体。**例如，餐厅常常雇佣一个领班负责迎接顾客，并在餐厅繁忙时留意给顾客安排桌位，为等待就餐的顾客按序排队是必要的。领班由所有顾客“共享”，不能被任何特定顾客占用太多时间。当顾客在一张桌子入坐后，有一个侍应生专门为这张桌子服务。

**下面来看一个使用 spserver 实现的简单的 line echo server 。**

**Java代码  [收藏代码](javascript:void())**

1. **class** SP\_EchoHandler : **public** SP\_Handler {
2. **public**:
3. SP\_EchoHandler(){}
4. virtual ~SP\_EchoHandler(){}
6. // return -1 : terminate session, 0 : continue
7. virtual **int** start( SP\_Request \* request, SP\_Response \* response ) {
8. request->setMsgDecoder( **new** SP\_LineMsgDecoder() );
9. response->getReply()->getMsg()->append(
10. "Welcome to line echo server, enter 'quit' to quit.\r\n" );
12. **return** 0;
13. }
15. // return -1 : terminate session, 0 : continue
16. virtual **int** handle( SP\_Request \* request, SP\_Response \* response ) {
17. SP\_LineMsgDecoder \* decoder = (SP\_LineMsgDecoder\*)request->getMsgDecoder();
19. **if**( 0 != strcasecmp( (**char**\*)decoder->getMsg(), "quit" ) ) {
20. response->getReply()->getMsg()->append( (**char**\*)decoder->getMsg() );
21. response->getReply()->getMsg()->append( "\r\n" );
22. **return** 0;
23. } **else** {
24. response->getReply()->getMsg()->append( "Byebye\r\n" );
25. **return** -1;
26. }
27. }
28. virtual **void** error( SP\_Response \* response ) {}
30. virtual **void** timeout( SP\_Response \* response ) {}
32. virtual **void** close() {}
33. };
35. **class** SP\_EchoHandlerFactory : **public** SP\_HandlerFactory {
36. **public**:
37. SP\_EchoHandlerFactory() {}
38. virtual ~SP\_EchoHandlerFactory() {}
40. virtual SP\_Handler \* create() **const** {
41. **return** **new** SP\_EchoHandler();
42. }
43. };
45. **int** main( **int** argc, **char** \* argv[] )
46. {
47. **int** port = 3333;
49. SP\_Server server( "", port, **new** SP\_EchoHandlerFactory() );
50. server.runForever();
52. **return** 0;
53. }

在最简单的情况下，使用 spserver 实现一个 TCP server 需要实现两个类：SP\_Handler 的子类 和 SP\_HandlerFactory 的子类。   
SP\_Handler 的子类负责处理具体业务。   
SP\_HandlerFactory 的子类协助 spserver 为每一个连接创建一个 SP\_Handler 子类实例。   
  
**1.SP\_Handler 生命周期**   
SP\_Handler 和 TCP 连接一对一，SP\_Handler 的生存周期和 TCP 连接一样。   
当 TCP 连接被接受之后，SP\_Handler 被创建，当 TCP 连接断开之后，SP\_Handler将被 destroy。   
  
**2.SP\_Handler 函数说明**   
SP\_Handler 有 5 个纯虚方法需要由子类来重载。这 5 个方法分别是：   
start：当一个连接成功接受之后，将首先被调用。返回 0 表示继续，-1 表示结束连接。   
handle：当一个请求包接收完之后，将被调用。返回 0 表示继续，-1 表示结束连接。   
error：当在一个连接上读取或者发送数据出错时，将被调用。error 之后，连接将被关闭。   
timeout：当一个连接在约定的时间内没有发生可读或者可写事件，将被调用。timeout 之后，连接将被关闭。   
close：当一个 TCP 连接被关闭时，无论是正常关闭，还是因为 error/timeout 而关闭。   
  
**3.SP\_Handler 函数调用时机**   
当需要调用 SP\_Handler 的 start/handle/error/timeout 方法时，相关的参数将被放入队列，然后由线程池来负责执行 SP\_Handler 对应的方法。因此在 start/handle/error/timeout 中可以使用同步操作来编程，可以直接使用阻塞型 I/O 。   
在发生 error 和 timeout 事件之后，close 紧跟着这两个方法之后被调用。   
如果是程序正常指示结束连接，那么在主线程中直接调用 close 方法。   
  
**4.高级功能－－MsgDecoder**  
这个 line echo server 比起常见的 echo server 有一点不同：只有在读到一行时才进行 echo。   
这个功能是通过一个称为 MsgDecoder 的接口来实现的。不同的 TCP server 在应用层的传输格式上各不相同。   
比如在 SMTP/POP 这一类的协议中，大部分命令是使用 CRLF 作为分隔符的。而在 HTTP 中是使用 Header + Body 的形式。   
为了适应不同的 TCP server，在 spserver 中有一个 MsgDecoder 接口，用来处理这些应用层的协议。   
比如在这个 line echo server 中，把传输协议定义为：只有读到一行时将进行 echo 。   
那么相应地就要实现一个 SP\_LineMsgDecoder ，这个 LineMsgDecoder 负责判断目前的输入缓冲区中是否已经有完整的一行。   
  
MsgDecoder 的接口如下：

**Java代码  [收藏代码](javascript:void())**

1. **class** SP\_MsgDecoder {
2. **public**:
3. virtual ~SP\_MsgDecoder();
5. **enum** { eOK, eMoreData };
6. virtual **int** decode( SP\_Buffer \* inBuffer ) = 0;
7. };

decode 方法对 inBuffer 里面的数据进行检查，看是否符合特定的要求。如果已经符合要求，那么返回 eOK ；如果还不满足要求，那么返回 eMoreData。比如 LineMsgDecoder 的 decode 方法的实现为：

**Java代码  [收藏代码](javascript:void())**

1. **int** SP\_LineMsgDecoder :: decode( SP\_Buffer \* inBuffer )
2. {
3. **if**( NULL != mLine ) free( mLine );
4. mLine = inBuffer->getLine();
6. **return** NULL == mLine ? eMoreData : eOK;
7. }

spserver 默认提供了几个 MsgDecoder 的实现：   
SP\_DefaultMsgDecoder ：它的 decode 总是返回 eOK ，即只要有输入就当作是符合要求了。   
    如果应用不设置 SP\_Request->setMsgDecoder 的话，默认使用这个。   
SP\_LineMsgDecoder ： 检查到有一行的时候，返回 eOK ，按行读取输入。   
SP\_DotTermMsgDecoder ：检查到输入中包含了特定的 <CRLF>.<CRLF> 时，返回 eOK。   
  
具体的使用例子可以参考示例：testsmtp 。   
  
**5.高级功能－－实现聊天室**  
spserver 还提供了一个广播消息的功能。使用消息广播功能可以方便地实现类似聊天室的功能。具体的实现可以参考示例：testchat 。   
  
**6.libevent**   
spserver 使用 c++ 实现，使用了一个第三方库－－libevent，以便在不同的平台上都能够使用最有效的事件驱动机制（Currently, libevent supports /dev/poll, kqueue(2), select(2), poll(2) and epoll(4). ）。