网络服务器应用程序框架

代码重用与应用程序框架

在不同软件之间，或多或少都会存在一些共同点，都会需要一些共同的功能。如果每次编写软件都完全从零开始的话，这将会是一种非常没有必要的浪费。因此，在可能的情况下，我们提倡尽量对已有的代码进行重用。

代码重用可以带来很大益处。首先，重用已有代码，可以减少编写新代码的工作量，提高工作效率。其次，被重用的代码都是经过一定时间和实践检验的，错误、故障和漏洞都会比全新编写的代码少。再次，当修正了重用代码中的错误后，使用了该重用代码的其它代码中的相应漏洞也会得到同样地修正，可以极大地提高程序的可靠性。

重用代码，并不等于"复制/粘贴"已有代码后再进行修改以适应新的需要。固然，这种做法也能够"重复使用"以前写下的代码，也能在一定程度上提高一时的 "工作效率"，但从长远和全局上看来，这种做法是相当有害的。如果使用"复制/粘贴"代码的方式"重用"了大量代码，当在某个版本的代码中发现了错误之后，将不得不修改所有使用了这些代码的地方的相应程序，不但费时费力，而且非常容易出错。简而言之，就是使用"复制/粘贴"代码法来进行代码重用完全不具备可维护性。

因此，一段良好设计的可重用代码，应该具备一定的通用性，应该能做到不作任何修改就能在许多不同的环境下使用。因此，编写可重用代码需要考虑的问题比编写不可重用代码要多得多，编写起来也比不可重用的代码困难。总之，编写可重用的代码是要付出一定代价的。据专家统计，编写可重用代码的工作量大约是为一个单独的应用编写特定代码工作量的3 到 9 倍。但是，如果一段可重用代码能够应用到多个实际项目中的话，总工作量将会由于共享了以往的成果而降低。显而易见，代码重用的次数越多，节约工作量的效果就越明显。因此，为了重用代码而付出的额外代价是值得的。

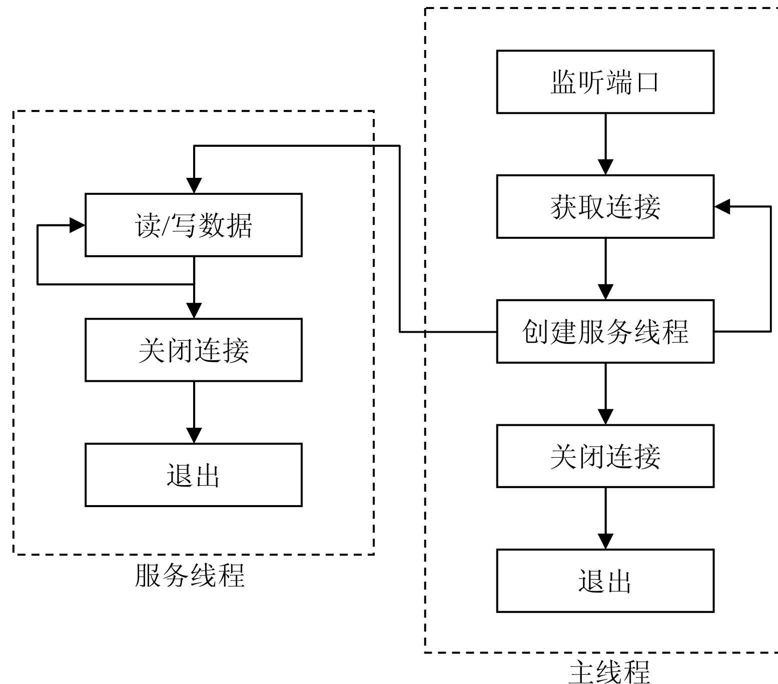
可重用的代码有两种类型，一种叫部件库（Off-shelf library），一种叫应用程序框架（Application Framework）。部件库的特点是，它提供了一些零散的、具有一定功能的部件。在编写程序的时候，由我们自己搭建整个应用程序的架构，并将这些部件"组装"到我们的程序之中。程序运行的时候，由我们自己编写的程序来调用现有的部件代码。而应用程序框架则与之相反，它自己已经具备了一个完整应用程序的结构。在编写程序的时候，改由我们提供各个部分的具体功能，由它来调用。

由于应用程序框架本身已经是一个完整程序的半成品，因此在它的基础上编写程序，会比全新创建程序或使用部件库搭建程序都简便很多，而且可以搭建出更加可靠的程序。当然，使用了框架代码后，程序的流程就必须按照框架的规定进行，不可能随心所欲地编写代码了。但是，应该明白这是一种福利而不是一种限制。

基于 TCP/IP 协议的服务器

随着网络技术的普及，网络程序的数量和种类都越来越多。在目前的应用模式下，服务器、客户端模式的应用十分广泛。目前现有的服务器程序据说有上千种，而绝大部分服务器都是基于 TCP/IP 协议的。

对比基于 TCP/IP 协议构建的各种应用层协议，不难发现，虽然各种协议之间差异很大，但服务模式都是类似的。无论何种服务器，它的工作流程都是，首先由主线程打开一个 TCP 端口并监听，当有用户连接服务器时，主线程获取该连接产生的 Socket 对，创建一个服务线程，由该线程为用户服务。主线程回到监听状态，继续等待下一个用户连接，如下图所示：[1]



从图中不难看出，不同类型的服务器之间主线程监听和获取连接这部分动作并没有差别，有差别的部分仅仅是服务线程。因此，我们可以把主线程封装起来以便复用。

作为一个应用程序框架，封装的东西越多，能够复用的东西就越多，以后编写程序也就会越简单。但是如果封装了太多东西，灵活性上又会受到影响，使框架的使用范围受到限制。因此，如何在保证灵活性的前提下封装更多的东西，就是我们需要研究的一个重要问题。

通过调查和归纳总结，我发现目前基于 TCP/IP 协议的服务程序大致可以分为以下三种模式：

第一、单向发送模式：当用户连接服务器时，服务器主动给用户发送用户需要的信息，然后切断连接。Daytime、Fortune、Chargen 等服务都属于此类。

第二、问答模式：当用户连接服务器时，显示或不显示欢迎信息后，即进入一种与服务器一问一答的状态——用户向服务器发送一个请求，然后服务器反馈一段信息。Echo、Http 等就属于此类。Ftp 服务需要创建额外的数据连接，和单纯的问答模式不完全相同，但它的控制连接上的动作仍然是一个典型的问答模式服务器。

第三、异步通讯模式：服务器和客户都在同时向对方发送数据，没有明显的一问一答过程，Terminal 服务就属于这种类型。

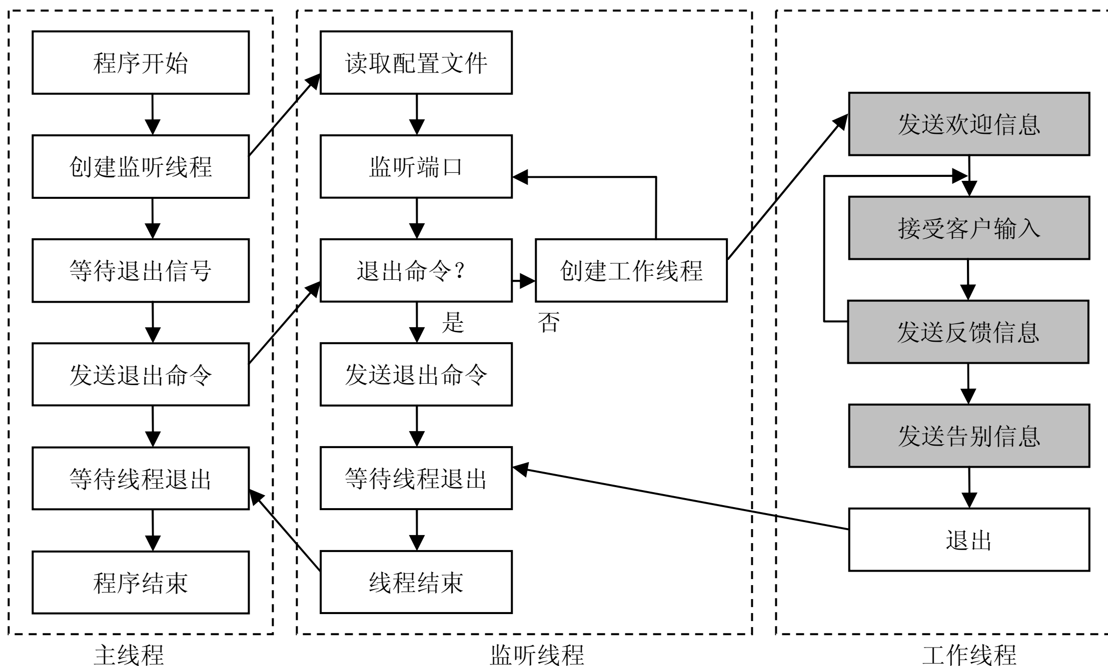
如果我们构建一种模式二的服务器，在发送欢迎信息后马上断开连接，那其实它就成为了模式一的服务器了。因此，我们可以把前两种模式统一为一种模式。

第三种模式比较复杂，但我们仍然可以套用模式二的框架：当用户连接后，服务器再创建一个新的线程用于处理服务端发向客户端的信息，在原先的“问答”模块中处理客户端发

[1] 在 Unix/Linux 平台下，许多服务器采用的是多进程工作模式，即每次创建新的进程来提供服务。进程的运行开销比线程大，而且由于不同进程不工作在同一个地址空间中，进程间通讯的难度也比线程间通讯大。这里我们只讨论多线程服务器的设计。

向服务端的信息。

这样一来，三种模式的服务程序都被统一到模式二了。因此，我的目标就是实现一个基于模式二的应用程序框架。整个程序的架构如下图：



很容易看出，一个服务程序，除了工作线程中灰色的三个方块所示步骤外，其它步骤都几乎是完全相同的。因此，为服务程序创建应用程序框架是可行的。

行问答模式的服务器

在目前的众多基于 TCP/IP 协议的服务器中，基于行问答模式的服务器又占多数，例如 POP3、SMTP、FTP 服务器就都是命令问答模式的，HTTP 协议在某种程度上也可以被看作命令问答模式。[1]命令问答模式的服务器特点是：服务器和客户端交替收发以行为单位的纯文本格式。下图是使用 POP3 协议从邮件服务器上收取邮件的一个典型过程：

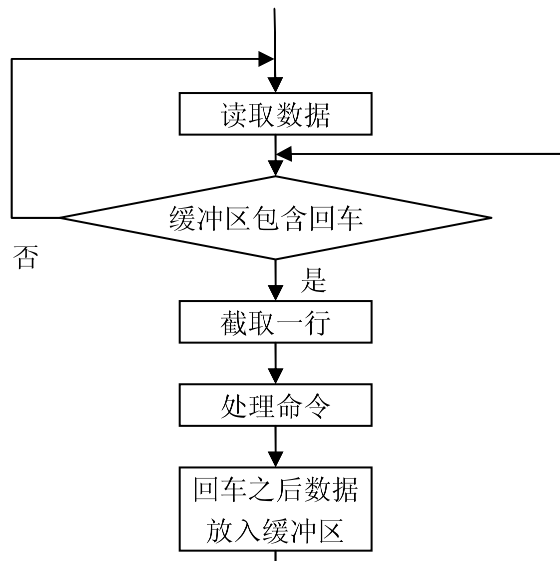
```
←  +OK SMTH BBS POP3/POP3S server at smth.org starting.
→  USER edyfox
←  +OK Password required for edyfox
→  PASS xxxxx
←  +OK edyfox has 279 message(s) (629904 octets).
→  STAT
←  +OK 279 629904
→  LIST 279
←  +OK 279 1724
→  TOP 279 0
←  +OK 1724 octets
←  Received: from insidesmtp (unknown [127.0.0.1]); Thu, 18 Mar 2004 23:19:15 +0800
←  Date: Thu, 18 Mar 2004 23:19:15 +0800
←  From: edyfox@smth.org
←  To: edyfox@smth.org
←  Subject: [Mar 18 22:51] 所有讯息备份
←
```

[1] HTTP 请求也是基于行的，它的每个请求由若干行组成。因此，可以在实现了行问答模式的前提上再扩充为“行组”问答模式。

```
←
←   .
→   QUIT
←   +OK SMTH BBS POP3/POP3S server at smth.org signing off.
```

由于 TCP/IP 协议是基于“字节流”模式进行传输的，它仅仅保证不同数据报到达的顺序不被改变，并不保证同一个数据报不会在传输过程中被截断为多个，也不保证多个数据报在到达时不会被拼接为一个。因此，以行为单位的字符问答型服务程序需要在以字节流为单位的服务器基础上再解决两个问题：

1.一条命令被截断为若干截，分布在若干个数据报中陆续到达。例如，用户使用字符模式的 Telnet 程序访问服务器，就会出现每次 read 只能收到一个字符的情况，需要等到整行都接收完毕后再进行处理。另外，有的路由器也会将超过规定长度的数据报分割为几个再传出。

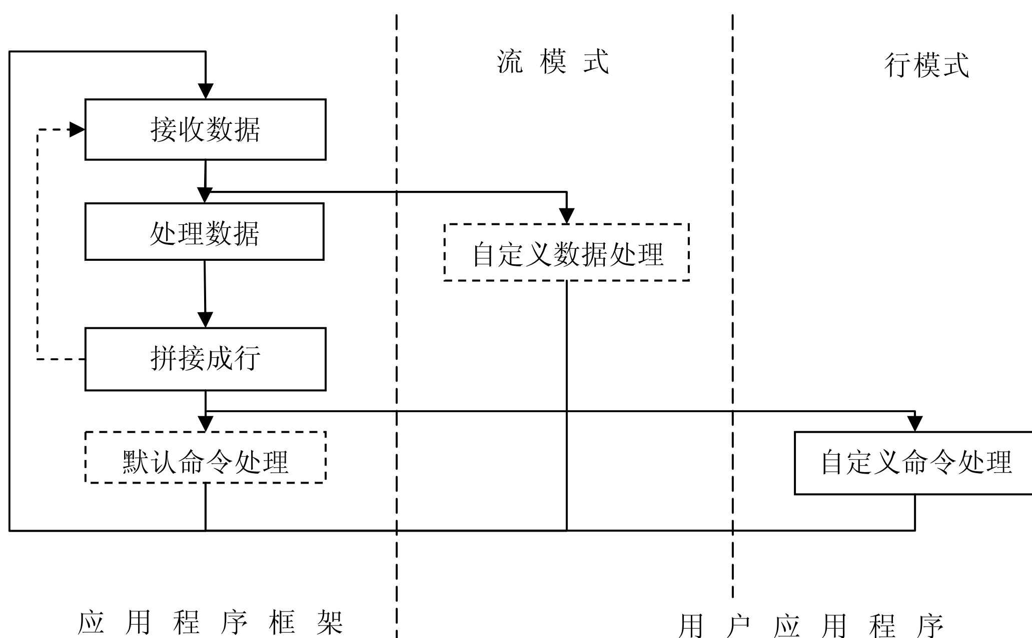


2.若干条命令在同一个数据包中一次到达。例如，有的路由器或交换机有时会将一部分到达的数据报拼接起来一次发出。当服务器收到这样的包的时候，就需要将数据按行截开之后，再进行处理。还有，有的客户端出于某种目的，也有可能将多行命令放在同一个数据报中一次传出。

总之，编写一个以行为单位的字符问答型服务器，不应该依赖客户端将每行单独封装在一个数据报中，必须将收到的数据按照一定的规则重新拼接成行后，再进行处理。

按照右图的流程进行处理，可以很好地解决上述问题。由于基于命令问答模式的服务器目前使用还相当普遍，因此我们有必要在网络服务器应用程序框架中针对行模式的服务程序进行专门的处理。

由于基于流模式的服务器同样存在着，而且有逐渐增加的趋势。因此我们不能为了实现行模式而将流模式完全抛弃。为了同时兼容流模式和行模式服务器，我设计了以下架构：



首先，服务器工作在流模式下，将收到的数据送给用户自定义的数据处理模块进行处理。如果用户没有提供数据处理模块的话，服务器改将数据发送到内置的数据处理模块。内置的数据处理模块按照上面介绍的流程将数据流拼接为行，再调用用户自定义的行处理模块进行处理。由于数据的分发机制是直接通过 C++提供的虚函数机制实现的，因此在效率上可以得到保障。

干净而优雅地退出

当服务器退出的时候，可能还有许多在线用户连接在该服务器上。在退出前，进行一些必要的清理工作是必要的。尤其是像 ftp 这样的服务，会为每个用户再开启一个数据连接。在清理掉用户之前，必须清理掉所有尚未关闭的数据连接才行。

该框架在用户切断连接的时候，会触发一个退出消息，让用户有机会进行资源清理。另外，当服务器退出的时候，会对所有的在线服务线程触发退出消息，让线程清理资源并切断与用户的连接，从而得以“干净而优雅地退出”。

在程序结束的时候，操作系统会释放该程序分配过的所有内存和系统资源。有的程序出于偷懒的目的，往往不释放那些只被使用一次的资源，这样做是不对的。虽然在这种情况下不会出现“内存泄露”、“资源泄露”的问题，但如果以后出于某种目的，需要将该程序用在另外一个程序中作为一个模块的话，那就很有可能导致泄露了。为了使该框架的应用范围尽可能地广，我在编写代码的时候很小心地避免了所有的内存泄露情况。

在服务器退出的时候，应该释放它监听的套接字。在 Linux 系统下，如果监听某个端口的程序被强行关闭而未释放套接字，其它程序在一定时间内将无法再使用该端口。因此，“干净而优雅地退出”是相当有必要的。

配置文件

一个服务器往往希望有部分参数可以配置，例如服务器端口，用户 IP 封禁列表等等，因此需要读写配置文件。在本框架中，我提供了一个用于读写类 ini 格式配置文件的类，完善地处理了文件读写的许多细节，可以很方便地读取、生成和修改配置文件。关于 ini 文件格式的具体情况，这里不再赘述。

然而，配置文件格式千千万万，存储方式各不相同，如何在框架中提供一个简单而统一的接口读取配置文件也是一个问题。许多时候，当 ini 不能提供用户所需的功能时，用户往往需要使用自己的配置文件管理器替换掉内置的 ini 管理器。经过一再归纳和抽象，我得出，不管什么格式的配置文件，读取的时候都需要提供文件名。而绝大部分的配置文件管理器都会提供从字符串构造的构造方法。因此，我的框架对于配置文件管理器的唯一要求就是：提供一个自 std::string 或 char *的构造方法。

如果一个程序不需要配置文件，例如 Windows 程序更希望从注册表中读取配置，只需要使用 std::string 类取代配置文件管理器模版参数，编译得到的目标代码中便不会再有任何与配置文件相关的代码，干净彻底。使用泛型和模版技术后，程序各个类之间保持了非常松散的耦合，为将来的维护和升级带来了极大的方便。

跨平台

跨平台是目前一个非常热门的话题。由于各方面条件和各种因素影响，许多不同的地方与场合都会用到不同的操作系统。即使在同一个操作系统下，也存在着各种各样不同的编译器。因此，对于一段代码，它的可移植性越好，重用性就越好。

作为一个服务程序，套接字是必不可少的。Windows 下的套接字相关函数和*nix[1]下的函数比较类似，常用函数函数名基本相同，然而，具体实作时，差异还是不少的。

首先，Windows 要求调用 Socket 系列函数之前先调用 WSAStarup，所有 Socket 系列函数调用结束后要调用 WSACleanup，*nix 下则没有这些要求。这个问题解决起来比较容易，只需简单地放置一个#ifdef 块，添加一段专门针对 Windows 的代码就行了。

其次，*nix 把 Socket 处理为一个文件描述字（file descriptor），在操作套接字的时候可以使用任何文件操作函数，如 read、write 等。而我在 Windows 下调试的时候，发现 read、write 函数对套接字并没有效果。据 MSDN 说，SOCKET 是内核对象句柄，并非文件描述字。虽然 Windows 下的套接字类型是 SOCKET，可阅读相关文档后就会发现其实它就是 int，和文件描述字的类型并没有区别。因此，无法用 C++的函数重载机制为 SOCKET 重写一套 read、write 函数。

但是，幸运的是，无论是 Windows 还是*nix，都为套接字提供了一套专用读写函数：send 和 recv 在两个平台下都可正常运行。因此，为了保证跨平台，使用*nix 时就不可以任意使用任何文件操作函数来操作套接字了。有不少文件操作函数使用起来都比 send、recv 方便，然而，要编写跨平台代码，必将在编码的方便性上付出代价。

作为一个服务器程序，使用多线程是必然的。然而不幸的是，Windows 和*nix 下的线程相关函数差异实在是太大了。如果不将线程相关的操作完整地封装起来，对外提供一个统一的接口，将会给以后的编程工作造成相当大的不便。

Windows 下创建函数使用_beginthreadex 函数调用。_beginthreadex 函数间接调用了 CreateThread 系统调用。由于 C Runtime Library 中大量使用了全局变量，为了实现多线程，必须对这些全局变量都封装起来。因此，在创建线程的时候，函数库需要做一些初始化工作。为了保证这些工作能够正常进行，Windows 要求程序员不要直接调用 CreateThread，而要通过_beginthreadex 创建线程。

*nix 下也有许多线程库可以使用，创建线程的方式也各式各样。目前，POSIX 线程库已成为*nix 下的标准，因此，使用 POSIX 线程库可以得到最佳的兼容性。POSIX 下创建线程使用的函数是 pthread_create。

无论是 Windows 还是*nix，创建线程时都有大量的安全属性可以选择。然而，由于这些属性都涉及到了太多各自操作系统的细节，因此要想在保证可移植的前提下使用这些特性，几乎是不可能的。由于在大部分情况下，我们创建一个线程的时候也并不使用这些属性，因此，在这个函数库中，我将抛弃所有安全属性描述。

另外，在 Windows 下，创建线程的时候，程序员可以显式指定线程堆栈的大小。而 POSIX 的线程创建函数却没有这个选项。这样一来，Windows 必须为了兼容性而放弃这个特性，不让程序员自己指定堆栈大小，统一使用默认大小——每个线程 4MB。对于大部分情况，这个规定还不至于造成什么不方便的地方。

现代操作系统对线程都提供了非常完善的支持，创建一个线程是非常容易的，但线程同步就困难得多。由于同一进程的各个线程之间可以共享全局变量，所以线程间的同步不必动用代价高昂的核心对象，只需要使用一些轻量级的同步机制就足以解决问题了。Windows

[1] 在不出现二义性的情况下，本文之后的部分将“Unix 与 Linux”简写为“*nix”。

下实现线程间同步的机制是 CRITICAL_SECTION，POSIX 下的是 pthread_mutex_t。Posix 下的 mutex 允许程序员测试该对象是否被锁住而不必阻塞等待，Windows 则没有这个机制。需要测试而不锁住一个 critical section 的时候并不多，这里我保留了 POSIX 下的这个功能，而未在 Windows 下模拟类似功能。Windows 下一个线程锁住一个 critical section 后，可以再次锁住它，并不会阻塞，但 POSIX 下不同，需要自己处理重复上锁的问题。为了提供一个统一的对外接口，就必须在*nix 系统下使用代码仿真 Windows 下的这项功能。

当一个线程结束时，需要取得该线程的返回值。而且，有的时候我们需要等待某个线程的结束，以便处理别的事情。在 Windows 下，一个线程结束后，它的句柄仍然有效，程序员可以多次调用 GetExitCodeThread 来获取它的返回值。在*nix 下，一个线程结束后将阻塞，等待程序员调用 pthread_join 获取它的返回值，返回值获取后线程句柄失效。在 Windows 下，一个线程结束之前获取返回值，函数将返回错误信息。而*nix 下，线程结束前获取返回值，调用者将被阻塞，直到线程结束。对于这个问题，我引入了一个变量保存线程函数的返回值。这样可以防止*nix 下提前探测导致的阻塞现象出现。但是，当一个线程返回之前去取得它的返回值，这时得到的返回值是未定义的。

由于上述种种不同，导致线程的跨平台封装较套接字复杂了许多。

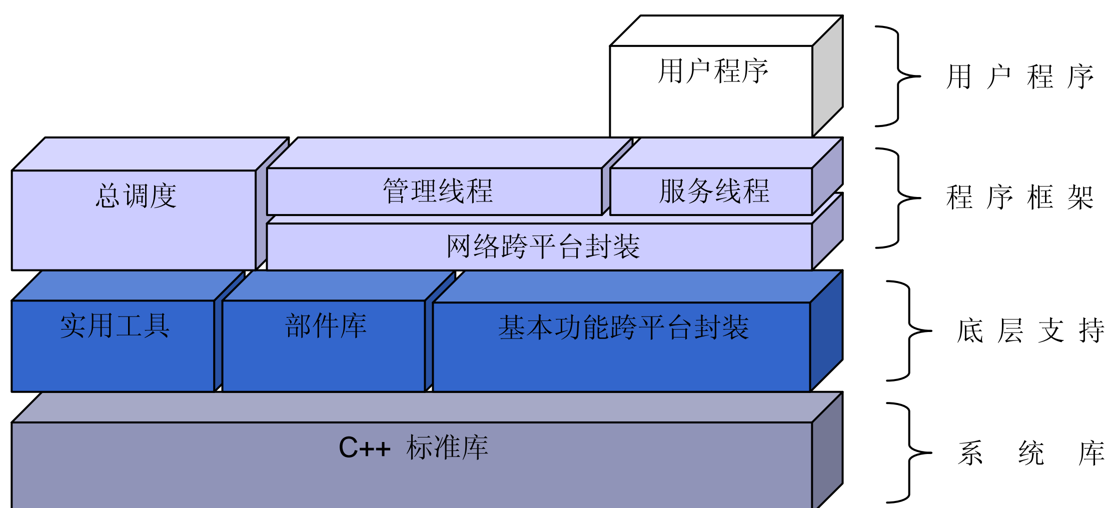
栈对象的自动释放是封装线程类过程中遇到的非常尴尬的一件事。如果在一个函数中创建了一个线程局部对象，当函数调用结束时，线程函数很可能还在运行，而这时线程对象却会被弹出堆栈并被析构掉。当线程函数结束时访问类对象时，类对象自身早已不复存在。这给线程类的封装带来了极大的障碍。将对象创建在堆中就不会出现这个问题，但堆对象必须靠程序员自己显式删除，删除时机较难掌握。为了解决这个问题，我引入了“线程监听器”的概念——当一个线程被创建时，你可以为它指定一个监听器，当线程结束后，它会在恰当的时候调用监听器，将对象指针传递给监听器，这时候监听器就可以安全地删除这个堆对象了。虽然这种处理方式很古怪而欠优雅，但的确很有效。

服务器框架实现

首先，在 C++标准库（STL）的基础上，我实现了一系列实用小工具和部件库，并对线程、同步机制等常用基本功能做了封装，为下一步开发完整框架奠定了基础。

然后，我在第一步工作的基础上实现了服务器总调度程序，并在进行了网络相关操作的跨平台封装之后，又实现了管理线程的调度和服务线程框架。

这样，以后需要编写网络服务程序时，只需要在服务线程框架的基础上实现用户自己的服务线程就可以了。整个架构之间的关系如下图：



支持平台

Windows 9x：

Windows 98，Visual C++ 6.0，P.J. STL 下调试通过。

Windows NT：

Windows 2000 Server Sp3，Visual C++ 6.0 Sp5，P.J. STL 或 STLPort 下均调试通过。

Windows 2003 Server，Visual C++ .NET 7.1，P. J. STL 或 STLPort 下均调试通过。

Windows 2003 Server，MinGW 下调试通过。

Linux：

Redhat 7.3，gcc 3.2，GNU STL，POSIX thread library 下调试通过。

Redhat 9.0，gcc 3.3，GNU STL，POSIX thread library 下调试通过。

Unix：

SunOS 5.8, gcc 3.2, GNU STL, POSIX thread library 下调试通过。

FreeBSD 5.0, gcc 3.2, GNU STL, POSIX thread library 下调试通过。

牛刀小试

经过这一层封装后，实现一个问答模式的服务程序可以将精力集中在处理命令本身上，不必过多关心网络和多线程的种种琐碎细节。以下用不足 30 行的代码就实现了一个 echo 服务器。

```
#include "etseq_app.h"

class echosrv : public etseq::etseq_srv
{
public:
    echosrv(SOCKET sock,
            void (*monitor)(euc::thread *addr, void *param) = NULL,
            sockaddr_in addr,
            void *param = NULL)
            : etseq::etseq_srv(sock, addr, monitor, param) {}
    static unsigned get_default_port();
protected:
    std::string parse_line(std::string question, bool *result = NULL);
};
```

```
unsigned echosrv::get_default_port()
{
    return 7;
}

//原样发回用户传入的信息
std::string echosrv::parse_line(std::string question, bool *result)
{
    if (result)
        *result = true;
    return std::string(question + "\r\n");
}

etseq::etseq_app<echosrv> app;
```

应用实例

实践是检验真理的唯一标准，这个网络服务器应用程序框架如果不能用来快速生成在实际中真正能用的服务器的话，再精致的设计也没有任何价值。下面我简单地介绍三个用该应用程序框架生成的服务器，以此探讨该网络服务器应用程序框架的表现能力。

回声服务器：

回声服务是一个非常简单的服务，它的作用仅仅是将用户输入的数据原样发回给用户。

该服务器在标准回声服务器的基础上作了扩展，当用户向服务器发送数据的时候，服务器使用十六进制与 ASCII 码对照的格式将客户端传来的数据 dump 出来，而当用户输入了回车之后，再将整行字符返回给客户端。该程序在编写网络程序的调试工作有一定的使用价值。

该服务器证明了该框架能够支持二进制流格式，并且可以用于编写网络服务器。

文件传输服务器(ftp)：

Ftp 协议是一种可靠而高效的数据传输协议，在扩大文件共享方面有着重要意义。使用前面所述的网络服务器应用程序框架，我开发了一个 ftp 服务器，它有以下几个特点：

小巧完善：整个服务器不足 200KB，可放在软盘中随身携带。支持简单的权限判断，支持断点续传。

快速可靠：在测试过程中，由中央主楼向紫荆公寓传播文件速度可达每秒 8~9MB。连续运行半个多月后未出现任何故障。

简单方便：不必安装，不必卸载，完全“绿色环保”；跨多个平台，处处可用。

该服务器证明了该框架可以用于编写能用于编写能用于实际应用的网络服务器。

五子棋网络对战：

该五子棋战网支持多用户多棋局，用户登录服务器后可以创建棋局或加入现有棋局进行游戏，或进入正在游戏的棋局充当观众。

本服务器支持三种模式的棋局——简易型、标准型和高级型。简易型就是某方连成五子为胜，没有任何限制；标准型规定 26 种开局定势，对黑方规定了双活三禁手、双四禁手和长连禁手，禁手不许落子；高级型在标准型的基础上增添了白棋三手可交换和黑棋五手两打的规则，并规定黑子走禁手直接判负，高级型规则是目前国际比赛中通行的规则。

该服务器证明了该框架还可以用于编写棋牌类的网络对战服务器。

SOCKS4 代理服务器：

在目前的网络环境下，代理服务器有着相当重要的意义。SOCKS4 代理是工作在二进制流异步通讯模式下的，服务端与客户端没有明显的问答过程。另外，由于浏览器在使用代理服务器时会并发大量连接，因此编写代理服务器对线程调度与同步的稳定性和可靠性的要求

很高。Etseq 对这一切提供了良好的支持，使编写服务器的工作大大简化。

该服务器证明了该框架可以很好地支持异步通讯模式的二进制流服务。

总结

该网络服务器应用程序框架是一个小巧而完善的框架，运用该框架可快速而高效地开发出基于 TCP/IP 协议的跨平台可移植网络服务程序，不但可以用于开发常规服务器，还可以用作其它软件中作为网络传输模块使用，拥有着广泛的应用前景。