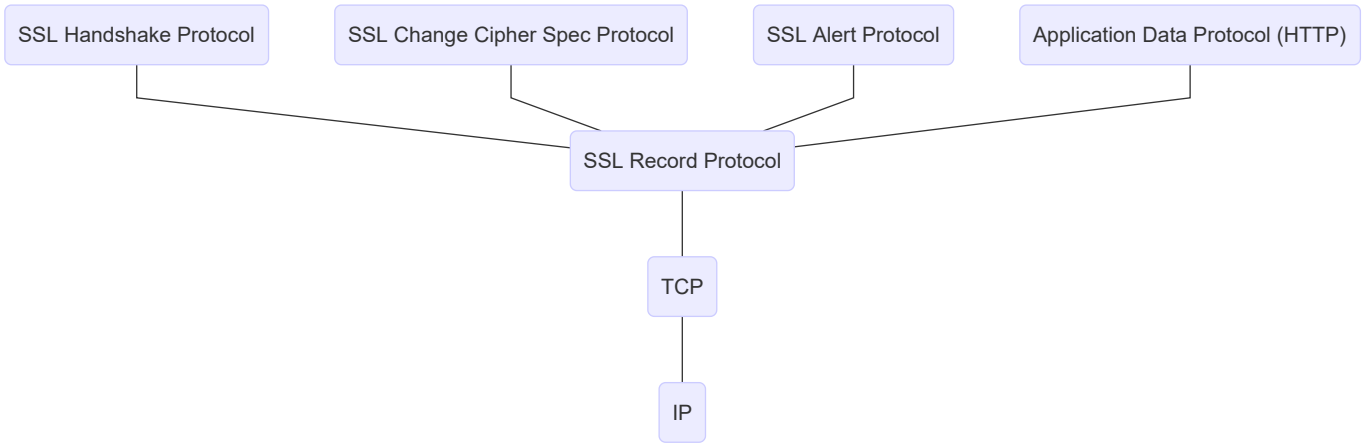
第7单元　安全Web服务器

7.1　知识讲解

7.1.1　SSL协议

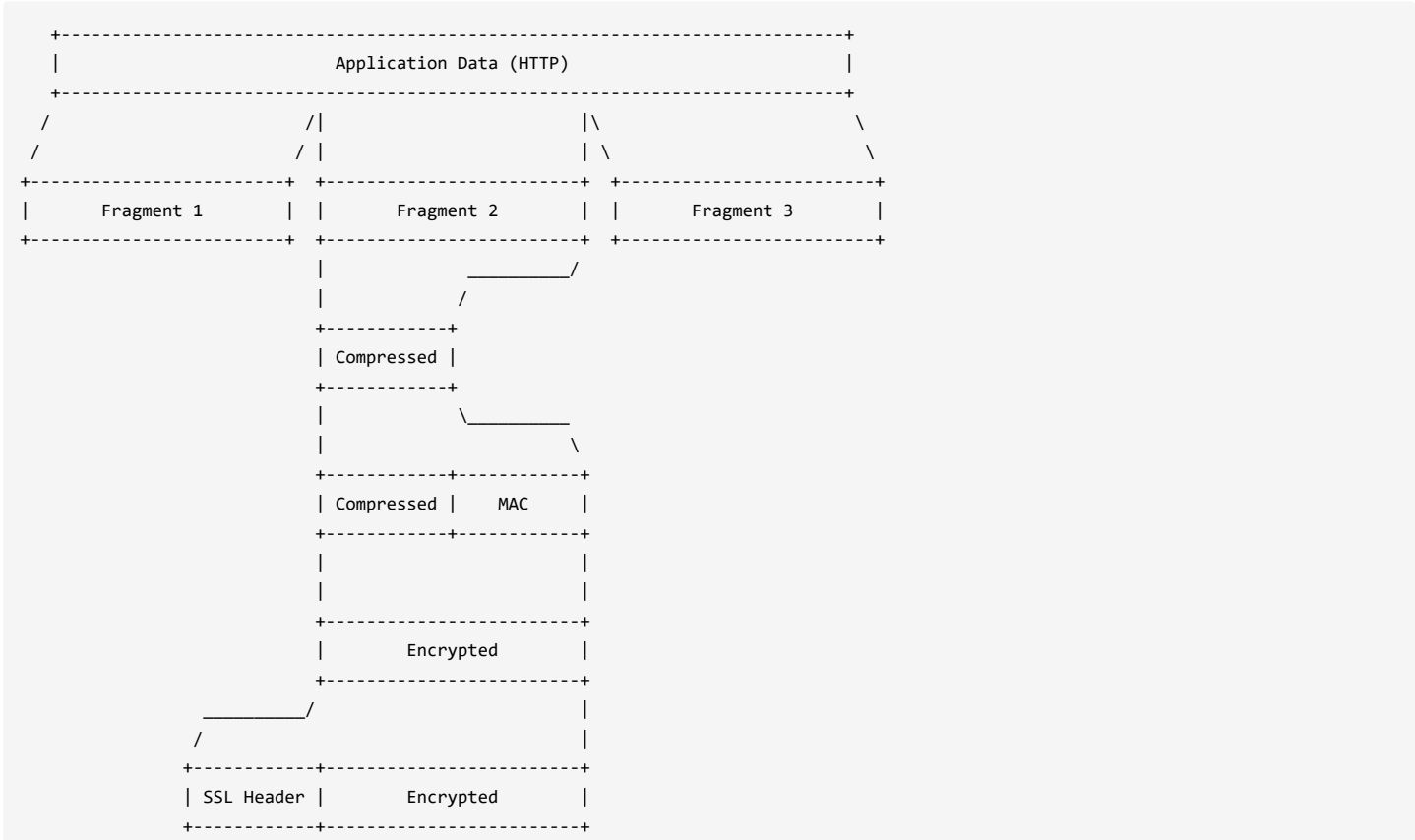
安全套接字层(Security Socket Layer, SSL)协议是网景(Netscape)公司于1996年提出的安全通信协议。它为网络应用层的通信提供了身份认证、数据保密和数据完整性校验等安全服务。设计该协议的主要目的是在网络环境中两个相互通信的进程之间，建立一个安全的数据通道。

SSL并非一个单独协议，而是包含两层结构的协议族。上层由SSL握手协议(SSL Handshake Protocol)、SSL修改密文规约协议(SSL Change Cipher Spec Protocol)和SSL警告协议(SSL Alert Protocol)组成，下层则是SSL记录协议(SSL Record Protocol)。SSL协议栈的结构如下图所示：



1. SSL记录协议

SSL记录协议为通信提供机密性和完整性保护，其工作流程如下图所示：



接收到应用层数据后，SSL记录协议首先对其进行分组，分组后每个数据块的大小不超过2^{14}=16384个字节。

然后，SSL记录协议对分组产生的每个数据块进行压缩，压缩过程中不能出现任何信息损失。

SSL记录协议为每个压缩后的数据块计算消息认证码(Message Authetication Code, MAC)，即一种带密钥的消息摘要，并将该消息认证码附加在压缩数据块之后。

SSL记录协议将压缩数据块和消息认证码作为一个整体进行加密。

最后，SSL记录协议在加密形成的密文之前添加SSL头部。

2. SSL握手协议

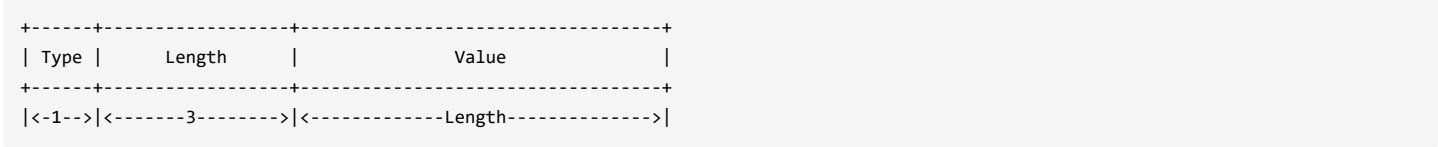
当SSL客户端和服务器首次通信时，双方通过SSL握手协议就一系列安全事宜达成共识，具体包括：

- 密钥交换算法
- 加密认证算法
- 数据压缩算法
- 双方数字证书
- 与算法相关的各种参数

SSL握手在应用层数据传输之前进行，包含一系列客户端与服务器之间的消息交换，其中每条消息均由三个字段组成：

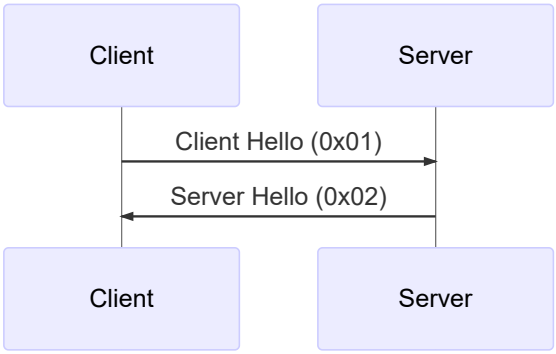
- 消息类型(T)，一个字节
- 消息长度(L)，三个字节
- 消息内容(V)，至少一个字节

SSL握手消息的结构如下图所示：



1) 发起阶段

SSL握手的发起阶段如下图所示：



客户端向服务器发送Client Hello类型的消息，其内容包括：

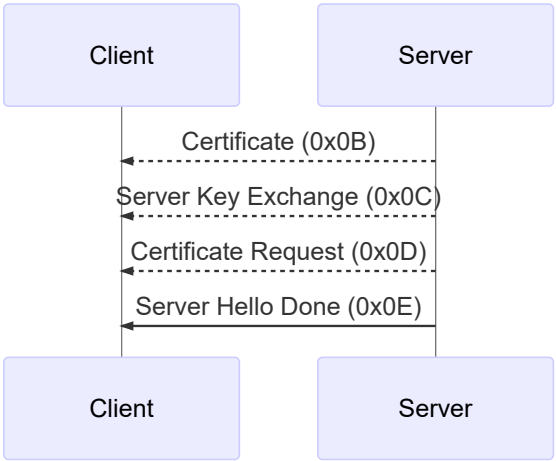
- 客户端所支持SSL协议的最高版本号
- 随机码
- 会话ID
- 密码套件(Cipher Suite)，具体包括：
  - 密钥交换算法
  - 加密认证算法
  - 数据压缩算法

服务器从客户端所提供的密码套件中确定后面使用的密钥交换算法、加密认证算法和数据压缩算法，并向客户端返回Server Hello类型的消息，其内容包括：

- 客户端和服务器都支持的SSL协议最高版本号
- 与客户端独立的随机码

2) 服务器认证和密钥交换

SSL握手的服务器认证和密钥交换如下图所示：



服务器向客户端发送Certificate类型的消息，向其出示自己的数字证书。

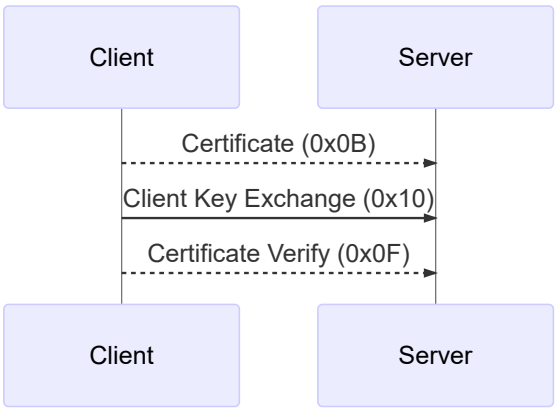
服务器向客户端发送Server Key Exchange类型的消息，告知其与密钥交换算法有关的参数。

服务器向客户端发送Certificate Request类型的消息，要求客户端出示其数字证书。

服务器向客户端发送Server Hello Done类型的消息，表明消息发送完毕，等待对方回应。

3) 客户端认证和秘钥交换

SSL握手的客户端认证和秘钥交换如下图所示：



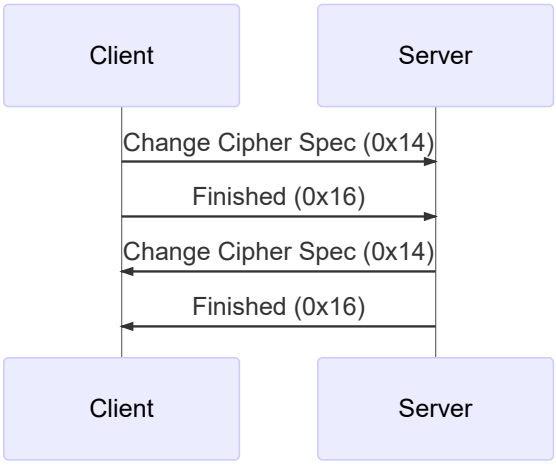
收到服务器Server Hello Done类型的消息后，客户端首先验证服务器的数字证书是否合法，与密钥交换算法有关的参数是否可行。如果服务器要求出示数字证书，客户端还要向服务器发送Certificate类型的消息，向其出示自己的数字证书。

客户端向服务器发送Client Key Exchange类型的消息，告知其与密钥交换算法有关的参数。

客户端向服务器发送Certificate Verify类型的消息，该消息使用与客户端数字证书中的公钥相对应的私钥做了数字签名。

4) 结束阶段

SSL握手的结束阶段如下图所示：



客户端和服务器分别向对方发送Change Cipher Spec和Finished类型的消息，告知对方SSL握手已经完成，双方的后续通信全部按照协商好的加密认证算法、数据压缩算法和密钥进行处理。

3. SSL修改密文规约协议

SSL修改密文规约协议只包含一种消息格式，消息类型为Change Cipher Spec (0x14)，消息内容只有一个字节——0x01。如下图所示：

```
+-------+-------+-------+
|  0x14 |   1   |  0x01 |
+-------+-------+-------+
               |<--1-->|
```

该消息用于通知对方后续通信全部按照协商好的加密认证算法、数据压缩算法和密钥进行处理。

4. SSL警告协议

SSL警告协议用于在通信过程中出现错误或异常情况时给出警告或关闭连接。根据错误的严重程度分为：

- Warning：一般错误
- Fatal：致命错误，立即关闭连接

7.1.2 OpenSSL

OpenSSL最早发布于1998年，其前身是Eric Young和Tim Hudson共同开发的SSLeay，目前已更新至1.0.2g版本。OpenSSL提供了完全的、免费的和开源的SSL协议实现，支持SSL2.0、SSL3.0以及TSL1.0等协议版本，并且能工作于大部分主流操作系统上，如UNIX、Linux和Windows等。OpenSSL支持最常用的对称和公钥加密算法、消息摘要算法等，在提供命令行工具的同时也提供了应用编程接口(API)支持二次开发。

1. 安装OpenSSL

在Ubuntu上安装OpenSSL非常简单：

```
$ sudo apt-get install libssl-dev
```

2. OpenSSL命令

1) genrsa子命令

genrsa子命令用于生成RSA私钥，其命令语法如下：

```
openssl genrsa [-out filename][-passout arg][-des][-des3]
               [-idea][-f4][-3][-rand file(s)][numbits]
```

选项	含义
-out filename	输出私钥到指定文件，默认为标准输出
-passout arg	输出文件口令
-des/-des3/-idea	针对私钥的加密算法，不指定则不加密
-f4/-3	选择公共组件
-rand file(s)	随机种子文件
numbits	模位数，缺省2048位

2) req子命令

req子命令用于创建和处理数字证书申请(Cerificate Signing Request, CSR)，其命令语法如下：

```
openssl req [-in filename][-inform DER|PEM][-out filename][-outform DER|PEM]
            [-text][-noout][-verify][-modulus][-new][-config filename]
            [-rand file(s)][-newkey rsa:bits][-newkey dsa:file][-keyout filename]
            [-key filename][-keyform DER|PEM][-x509][-days n]
```

选项	含义
-in filename	输入数字证书申请文件，仅当未使用-new和-newkey选项时有效
-inform DER|PEM	输入数字证书申请格式，DER采用ANSI的DER标准格式，PEM则为Base64编码格式
-out filename	输出数字证书申请文件
-outform DER|PEM	输出数字证书申请格式
-text	以文本方式打印数字证书申请
-noout	不打印数字证书申请编码版本
-verify	验证数字证书申请的数字签名
-modulus	打印数字证书申请的公钥模数
-new	生成数字证书申请

-config filename	数字证书申请模板文件
-rand file(s)	随机种子文件
-newkey rsa:bits	同时生成数字证书申请和RSA私钥，其参数指明私钥长度
-newkey dsa:file	同时生成数字证书申请和DSA私钥，其参数指明参数文件
-keyout filename	输出私钥文件
-key filename	输入私钥文件
-keyform DER|PEM	输入私钥格式
-x509	输出x509结构
-days n	如果使用了-x509选项，指定CA给第三方签证书的有效期天数，默认30天

3) x509子命令

x509子命令用于显示数字证书的内容、转换数字证书的格式、给CSR签名等，其命令语法如下：

```
openssl x509 [-in filename][-inform DER|PEM|NET][-out filename][-outform DER|PEM|NET]
             [-md2/-md5/-sha1/-mdc2][-text][-noout][-modulus][-serial][-issuer]
             [-subject][-hash][-nameopt option][-email][-startdate][-enddate][-dates]
             [-fingerprint][-C][-trustout][-setalias arg][-alias][-clrtrust][-purpose]
```

选项	含义
-in filename	输入数字证书文件
-inform DER|PEM|NET	输入数字证书格式
-out filename	输出数字证书文件
-outform DER|PEM|NET	输出数字证书格式
-md2/-md5/-sha1/-mdc2	哈希算法，默认MD5
-text	以文本方式打印数字证书
-noout	不打印数字证书编码版本
-modulus	打印数字证书的公钥模数
-serial	打印数字证书的序列号
-issuer	打印数字证书颁发者名
-subject	打印数字证书持有者名
-hash	打印数字证书持有者名的哈希值
-nameopt option	各种证书名称选项
-email	打印证书申请者的电子邮箱
-startdate	打印数字证书生效时间
-enddate	打印数字证书到期时间
-dates	打印数字证书生效和到期时间
-fingerprint	打印DER格式数字证书的DER版本
-C	以C语言代码的格式打印结果

-trustout	打印可信数字证书
-setalias arg	设置数字证书别名
-alias	打印数字证书别名
-clrtrust	清除数字证书附加项中所有关于用途允许的内容
-purpose	打印数字证书附加项中所有关于用途允许和禁止的内容

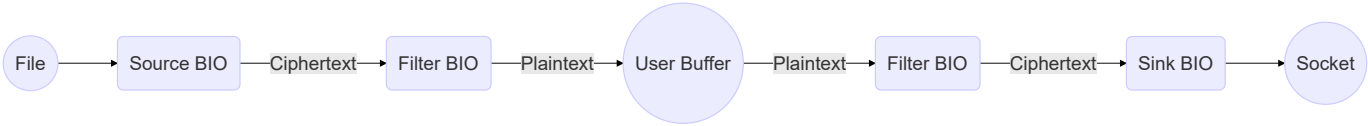
7.1.3 BIO结构

BIO是OpenSSL中重要的数据结构，其作用是封装并隐藏底层I/O操作的细节，无论是针对文件的读取和写入，还是基于套接字的接收和发送，都可以通过BIO类型的对象来完成。借助BIO结构，应用程序可以完全透明地实现SSL连接、加密通信以及文件读写等功能。

BIO分为两种：

- Source/Sink型BIO：代表数据源/目的，如文件BIO和套接字BIO
- Filter型BIO：把数据从一个BIO传递到另一个BIO，在传递的过程中完成对数据的加解密

如下图所示：



1. BIO结构

```
typedef struct bio_st {
    BIO_METHOD      * method;               // 方法结构
    long           (* callback)(            // 回调函数
                       struct bio_st * bio,
                       int             mode,
                       const char    * argp,
                       int             argi,
                       long            argl,
                       long            ret);
    char            * cb_arg;                // 回调参数
    int               init;                  // 初始化标志，1表示已初始化
    int               shutdown;              // 已关闭标志，1表示已被关闭
    int               flags;
    int               retry_reason;
    int               num;
    void            * ptr;
    struct bio_st   * next_bio;              // Filter型BIO的下一个节点
    struct bio_st   * prev_bio;              // Filter型BIO的上一个节点
    int               references;
    unsigned long     num_read;              // 读取字节数
    unsigned long     num_write;             // 写入字节数
    CRYPTO_EX_DATA    ex_data;
}   BIO;
```

2. 常用BIO相关函数

通过BIO_new函数可以创建BIO对象：

```
BIO* BIO_new(BIO_METHOD* type);
```

在BIO结构的所有字段中，method可以说是最关键的一个字段，它决定了BIO对象的功能，而该字段的值则来自创建BIO对象时所提供的type参数。一般情况下，传递给type参数的值是通过某个具体的生成函数获得的，例如执行下面的代码，即可创建一个用于读写内存的BIO对象：

```
BIO* membio = BIO_new(BIO_s_mem());
```

1) Source/Sink型BIO的生成函数

函数	功能
BIO_s_mem	封装内存操作，读写内存
BIO_s_fd	封装一个文件描述符，读写该文件
BIO_s_file	封装标准输入、标准输出和标准错误
BIO_s_accept	封装Socket API的accept函数，等待并接受来自远程主机的连接请求
BIO_s_connect	封装Socket API的connect函数，向远程主机发起连接请求
BIO_s_socket	封装Socket API，实现网络通信
BIO_s_bio	封装一个BIO对，向其中一个写入，从另外一个读出
BIO_s_null	封装空设备，写入数据被丢弃，读取得到EOF

2) Filter型BIO的生成函数

函数	功能
BIO_f_ssl	封装SSL协议，按照协议的要求对数据进行加工
BIO_f_base64	封装Base64编解码，写入时编码，读取时解码
BIO_f_cipher	封装加解密，写入时加密，读取时解密
BIO_f_md	封装摘要计算，通过的消息被计算摘要
BIO_f_buffer	封装缓冲区操作，写入的数据被传递给下一个BIO，读取的数据则来自前一个BIO
BIO_f_null	封装空操作，相当于不存在

3. 通过BIO进行I/O操作

1) BIO_read函数

```
int BIO_read(BIO* bio, void* buf, int len);
```

从bio读取len字节数据到buf中。成功返回实际读取的字节数，失败返回0或-1，若该bio没有实现此功能的方法，则返回-2。

2) BIO_write函数

```
int BIO_write(BIO* bio, const void* buf, int len);
```

将buf中的len字节数据写入bio。成功返回实际写入的字节数，失败返回0或-1，若该bio没有实现此功能的方法，则返回-2。

3) BIO_gets函数

```
int BIO_gets(BIO* bio, char* buf, int size);
```

从bio读取最多包含size-1个字符的字符串到buf中，该字符串以空字符结尾。成功返回实际读取的字符数(不含结尾空字符)，失败返回0或-1，若该bio没有实现此功能的方法，则返回-2。

注意，对消息摘要型BIO调用此函数，会返回整个摘要字符串，而不受size参数的限制。

4) BIO_puts函数

```
int BIO_puts(BIO* bio, const char* buf);
```

将buf中以空字符结尾的字符串写入bio。成功返回实际写入的字符数(不含结尾空字符)，失败返回0或-1，若该bio没有实现此功能的方法，则返回-2。

5) BIO_flush函数(宏)

```
#define BIO_flush(bio) (int)BIO_ctrl(bio, BIO_CTRL_FLUSH, 0, NULL)
```

将bio内部缓冲区中的数据一次性全部写出。有时也用于设置EOF标志，表示无数据可写。成功返回1，失败返回0或-1。

注意，所有针对非阻塞Source/Sink型BIO的读写操作返回失败(0或-1)，并不意味着一定发生了错误，也可能仅仅是目前暂时不可读取或写入，此时若BIO_should_retry(bio)的值为1，可于稍后重试。

7.2 实训案例

7.2.1 基于OpenSSL的安全Web服务器

在理解HTTPS和SSL工作原理的基础上，实现安全的Web服务器。

服务器能够并发处理多个请求，要求至少能支持GET命令。

进一步扩展Web服务器的功能，增加对HEAD、POST和DELETE命令的支持。

编写必要的客户端测试程序，用于发送HTTPS请求并显示服务器返回的响应，也可以使用一般的Web浏览器测试服务器。

7.2.2 程序清单

1. 声明Thread类

```
// thread.h
// 声明Thread类

#pragma once

#include <pthread.h>

// 线程
class Thread {
public:
    // 析构函数
    virtual ~Thread(void) {}

    // 启动线程
    int start(void);

private:
    // 线程处理
    virtual void* run(void) = 0;

    // 线程过程
    static void* run(void* arg);

    pthread_t tid; // 线程标识
};
```

2. 实现Thread类

```cpp
// thread.cpp
// 实现Thread类

#include <string.h>
#include <iostream>
using namespace std;

#include "thread.h"

// 启动线程
int Thread::start(void) {
    // 设置线程分离状态
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    // 创建线程
    int error = pthread_create(&tid, &attr, run, this);
    if (error) {
        cerr << "pthread_create: " << strerror(error) << endl;
        pthread_attr_destroy(&attr);
        return -1;
    }

    pthread_attr_destroy(&attr);
    return 0;
}

// 线程过程
void* Thread::run(void* arg) {
    return static_cast<Thread*>(arg)->run();
}
```

3. 声明ClientThread类

```
// clientthread.h
// 声明ClientThread类

#pragma once

#include <stdint.h>
#include <openssl/ssl.h>
#include <map>
using namespace std;

#include "thread.h"

// 客户线程
class ClientThread : public Thread {
public:
    // 构造函数
    ClientThread(int sockfd, SSL_CTX* ctx, const string& root);
    // 析构函数
    ~ClientThread(void);

private:
    // 方法
    typedef enum tag_Method {
        EMETHOD_HEAD,
        EMETHOD_GET
    }   EMETHOD;

    // 初始化I/O链
    int initChain(void);
    // 接收请求包
    int recvReq(void);
    // 分析请求包
    int analyzeReq(void);
    // 发送响应头
    int sendHeader(void) const;
    // 发送响应体
    int sendBody(void) const;

    // 线程处理
    void* run(void);

    static const size_t REQSIZE;    // 请求包缓冲区大小
    static const size_t HEADERSIZE; // 响应头缓冲区大小
    static const size_t BODYSIZE;   // 响应体缓冲区大小

    int                 sockfd; // 套接字文件描述符
    SSL_CTX*            ctx;    // 上下文
    SSL*                ssl;    // SSL对象
    BIO*                bufbio; // 缓冲区BIO
    char*               req;    // 请求包缓冲区
    EMETHOD             method; // 方法
    string              uri;    // 统一资源标识符
    bool                alive;  // 保持连接
    map<string, string> mime;   // 内容类型映射
    char*               header; // 响应头缓冲区
    uint8_t*            body;   // 响应体缓冲区
};
```

4. 实现ClientThread类

```cpp
// clientthread.cpp
// 实现ClientThread类

#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <string.h>
#include <openssl/ssl.h>
#include <openssl/err.h>
#include <iostream>
using namespace std;

#include "clientthread.h"

const size_t ClientThread::REQSIZE    = 4096; // 请求包缓冲区大小
const size_t ClientThread::HEADERSIZE = 2048; // 响应头缓冲区大小
const size_t ClientThread::BODYSIZE   = 2048; // 响应体缓冲区大小

// 构造函数
ClientThread::ClientThread(int sockfd, SSL_CTX* ctx,
    const string& root) : sockfd(sockfd), ctx(ctx), ssl(NULL),
    req(new char[REQSIZE]), uri(root), alive(false),
    header(new char[HEADERSIZE]), body(new uint8_t[BODYSIZE]) {
    // 初始化内容类型映射
    mime["."]        = "application/x-";
    mime[".*"]       = "application/octet-stream";
    mime[".001"]     = "application/x-001";
    mime[".301"]     = "application/x-301";
    mime[".323"]     = "text/h323";
    mime[".906"]     = "application/x-906";
    mime[".907"]     = "drawing/907";
    mime[".acp"]     = "audio/x-mei-aac";
    mime[".ai"]      = "application/postscript";
    mime[".aif"]     = "audio/aiff";
    mime[".aifc"]    = "audio/aiff";
    mime[".aiff"]    = "audio/aiff";
    mime[".a11"]     = "application/x-a11";
    mime[".anv"]     = "application/x-anv";
    mime[".apk"]     = "application/vnd.android.package-archive";
    mime[".asa"]     = "text/asa";
    mime[".asf"]     = "video/x-ms-asf";
    mime[".asp"]     = "text/asp";
    mime[".asx"]     = "video/x-ms-asf";
    mime[".au"]      = "audio/basic";
    mime[".avi"]     = "video/avi";
    mime[".awf"]     = "application/vnd.adobe.workflow";
    mime[".biz"]     = "text/xml";
    mime[".bmp"]     = "application/x-bmp";
    mime[".bot"]     = "application/x-bot";
    mime[".c4t"]     = "application/x-c4t";
    mime[".c90"]     = "application/x-c90";
    mime[".cal"]     = "application/x-cals";
    mime[".cat"]     = "application/vnd.ms-pki.seccat";
    mime[".cdf"]     = "application/x-netcdf";
    mime[".cdr"]     = "application/x-cdr";
    mime[".cel"]     = "application/x-cel";
    mime[".cer"]     = "application/x-x509-ca-cert";
    mime[".cg4"]     = "application/x-g4";
    mime[".cgm"]     = "application/x-cgm";
    mime[".cit"]     = "application/x-cit";
    mime[".class"]   = "java/*";
    mime[".cml"]     = "text/xml";
    mime[".cmp"]     = "application/x-cmp";
    mime[".cmx"]     = "application/x-cmx";
    mime[".cot"]     = "application/x-cot";
    mime[".crl"]     = "application/pkix-crl";
    mime[".crt"]     = "application/x-x509-ca-cert";
    mime[".csi"]     = "application/x-csi";
```

```
    mime[".css"]     = "text/css";
    mime[".cut"]     = "application/x-cut";
    mime[".dbf"]     = "application/x-dbf";
    mime[".dbm"]     = "application/x-dbm";
    mime[".dbx"]     = "application/x-dbx";
    mime[".dcd"]     = "text/xml";
    mime[".dcx"]     = "application/x-dcx";
    mime[".der"]     = "application/x-x509-ca-cert";
    mime[".dgn"]     = "application/x-dgn";
    mime[".dib"]     = "application/x-dib";
    mime[".dll"]     = "application/x-msdownload";
    mime[".doc"]     = "application/msword";
    mime[".dot"]     = "application/msword";
    mime[".drw"]     = "application/x-drw";
    mime[".dtd"]     = "text/xml";
    mime[".dwf"]     = "application/x-dwf";
    mime[".dwg"]     = "application/x-dwg";
    mime[".dxb"]     = "application/x-dxb";
    mime[".dxf"]     = "application/x-dxf";
    mime[".edn"]     = "application/vnd.adobe.edn";
    mime[".emf"]     = "application/x-emf";
    mime[".eml"]     = "message/rfc822";
    mime[".ent"]     = "text/xml";
    mime[".epi"]     = "application/x-epi";
    mime[".eps"]     = "application/x-ps";
    mime[".eps"]     = "application/postscript";
    mime[".etd"]     = "application/x-ebx";
    mime[".exe"]     = "application/x-msdownload";
    mime[".fax"]     = "image/fax";
    mime[".fdf"]     = "application/vnd.fdf";
    mime[".fif"]     = "application/fractals";
    mime[".fo"]      = "text/xml";
    mime[".frm"]     = "application/x-frm";
    mime[".g4"]      = "application/x-g4";
    mime[".gbr"]     = "application/x-gbr";
    mime[".gif"]     = "image/gif";
    mime[".gl2"]     = "application/x-gl2";
    mime[".gp4"]     = "application/x-gp4";
    mime[".hgl"]     = "application/x-hgl";
    mime[".hmr"]     = "application/x-hmr";
    mime[".hpg"]     = "application/x-hpgl";
    mime[".hpl"]     = "application/x-hpl";
    mime[".hqx"]     = "application/mac-binhex40";
    mime[".hrf"]     = "application/x-hrf";
    mime[".hta"]     = "application/hta";
    mime[".htc"]     = "text/x-component";
    mime[".htm"]     = "text/html";
    mime[".html"]    = "text/html";
    mime[".htt"]     = "text/webviewhtml";
    mime[".htx"]     = "text/html";
    mime[".icb"]     = "application/x-icb";
    mime[".ico"]     = "image/x-icon";
    mime[".iff"]     = "application/x-iff";
    mime[".ig4"]     = "application/x-g4";
    mime[".igs"]     = "application/x-igs";
    mime[".iii"]     = "application/x-iphone";
    mime[".img"]     = "application/x-img";
    mime[".ins"]     = "application/x-internet-signup";
    mime[".ipa"]     = "application/vnd.iphone";
    mime[".isp"]     = "application/x-internet-signup";
    mime[".IVF"]     = "video/x-ivf";
    mime[".java"]    = "java/*";
    mime[".jfif"]    = "image/jpeg";
    mime[".jpe"]     = "image/jpeg";
    mime[".jpeg"]    = "image/jpeg";
    mime[".jpg"]     = "image/jpeg";
    mime[".js"]      = "application/x-javascript";
    mime[".jsp"]     = "text/html";
    mime[".la1"]     = "audio/x-liquid-file";
    mime[".lar"]     = "application/x-laplayer-reg";
```

```
mime[".latex"]   = "application/x-latex";
mime[".lavs"]    = "audio/x-liquid-secure";
mime[".lbm"]     = "application/x-lbm";
mime[".lmsff"]   = "audio/x-la-lms";
mime[".ls"]      = "application/x-javascript";
mime[".ltr"]     = "application/x-ltr";
mime[".m1v"]     = "video/x-mpeg";
mime[".m2v"]     = "video/x-mpeg";
mime[".m3u"]     = "audio/mpegurl";
mime[".m4e"]     = "video/mpeg4";
mime[".mac"]     = "application/x-mac";
mime[".man"]     = "application/x-troff-man";
mime[".math"]    = "text/xml";
mime[".mdb"]     = "application/msaccess";
mime[".mfp"]     = "application/x-shockwave-flash";
mime[".mht"]     = "message/rfc822";
mime[".mhtml"]   = "message/rfc822";
mime[".mi"]      = "application/x-mi";
mime[".mid"]     = "audio/mid";
mime[".midi"]    = "audio/mid";
mime[".mil"]     = "application/x-mil";
mime[".mml"]     = "text/xml";
mime[".mnd"]     = "audio/x-musicnet-download";
mime[".mns"]     = "audio/x-musicnet-stream";
mime[".mocha"]   = "application/x-javascript";
mime[".movie"]   = "video/x-sgi-movie";
mime[".mp1"]     = "audio/mp1";
mime[".mp2"]     = "audio/mp2";
mime[".mp2v"]    = "video/mpeg";
mime[".mp3"]     = "audio/mp3";
mime[".mp4"]     = "video/mpeg4";
mime[".mpa"]     = "video/x-mpg";
mime[".mpd"]     = "application/vnd.ms-project";
mime[".mpe"]     = "video/x-mpeg";
mime[".mpeg"]    = "video/mpg";
mime[".mpg"]     = "video/mpg";
mime[".mpga"]    = "audio/rn-mpeg";
mime[".mpp"]     = "application/vnd.ms-project";
mime[".mps"]     = "video/x-mpeg";
mime[".mpt"]     = "application/vnd.ms-project";
mime[".mpv"]     = "video/mpg";
mime[".mpv2"]    = "video/mpeg";
mime[".mpw"]     = "application/vnd.ms-project";
mime[".mpx"]     = "application/vnd.ms-project";
mime[".mtx"]     = "text/xml";
mime[".mxp"]     = "application/x-mmxp";
mime[".net"]     = "image/pnetvue";
mime[".nrf"]     = "application/x-nrf";
mime[".nws"]     = "message/rfc822";
mime[".odc"]     = "text/x-ms-odc";
mime[".out"]     = "application/x-out";
mime[".p10"]     = "application/pkcs10";
mime[".p12"]     = "application/x-pkcs12";
mime[".p7b"]     = "application/x-pkcs7-certificates";
mime[".p7c"]     = "application/pkcs7-mime";
mime[".p7m"]     = "application/pkcs7-mime";
mime[".p7r"]     = "application/x-pkcs7-certreqresp";
mime[".p7s"]     = "application/pkcs7-signature";
mime[".pc5"]     = "application/x-pc5";
mime[".pci"]     = "application/x-pci";
mime[".pcl"]     = "application/x-pcl";
mime[".pcx"]     = "application/x-pcx";
mime[".pdf"]     = "application/pdf";
mime[".pdx"]     = "application/vnd.adobe.pdx";
mime[".pfx"]     = "application/x-pkcs12";
mime[".pgl"]     = "application/x-pgl";
mime[".pic"]     = "application/x-pic";
mime[".pko"]     = "application/vnd.ms-pki.pko";
mime[".pl"]      = "application/x-perl";
mime[".plg"]     = "text/html";
```

```
    mime[".pls"]     = "audio/scpls";
    mime[".plt"]     = "application/x-plt";
    mime[".png"]     = "image/png";
    mime[".pot"]     = "application/vnd.ms-powerpoint";
    mime[".ppa"]     = "application/vnd.ms-powerpoint";
    mime[".ppm"]     = "application/x-ppm";
    mime[".pps"]     = "application/vnd.ms-powerpoint";
    mime[".ppt"]     = "application/vnd.ms-powerpoint";
    mime[".ppt"]     = "application/x-ppt";
    mime[".pr"]      = "application/x-pr";
    mime[".prf"]     = "application/pics-rules";
    mime[".prn"]     = "application/x-prn";
    mime[".prt"]     = "application/x-prt";
    mime[".ps"]      = "application/x-ps";
    mime[".ps"]      = "application/postscript";
    mime[".ptn"]     = "application/x-ptn";
    mime[".pwz"]     = "application/vnd.ms-powerpoint";
    mime[".r3t"]     = "text/vnd.rn-realtext3d";
    mime[".ra"]      = "audio/vnd.rn-realaudio";
    mime[".ram"]     = "audio/x-pn-realaudio";
    mime[".ras"]     = "application/x-ras";
    mime[".rat"]     = "application/rat-file";
    mime[".rdf"]     = "text/xml";
    mime[".rec"]     = "application/vnd.rn-recording";
    mime[".red"]     = "application/x-red";
    mime[".rgb"]     = "application/x-rgb";
    mime[".rjs"]     = "application/vnd.rn-realsystem-rjs";
    mime[".rjt"]     = "application/vnd.rn-realsystem-rjt";
    mime[".rlc"]     = "application/x-rlc";
    mime[".rle"]     = "application/x-rle";
    mime[".rm"]      = "application/vnd.rn-realmedia";
    mime[".rmf"]     = "application/vnd.adobe.rmf";
    mime[".rmi"]     = "audio/mid";
    mime[".rmj"]     = "application/vnd.rn-realsystem-rmj";
    mime[".rmm"]     = "audio/x-pn-realaudio";
    mime[".rmp"]     = "application/vnd.rn-rn_music_package";
    mime[".rms"]     = "application/vnd.rn-realmedia-secure";
    mime[".rmvb"]    = "application/vnd.rn-realmedia-vbr";
    mime[".rmx"]     = "application/vnd.rn-realsystem-rmx";
    mime[".rnx"]     = "application/vnd.rn-realplayer";
    mime[".rp"]      = "image/vnd.rn-realpix";
    mime[".rpm"]     = "audio/x-pn-realaudio-plugin";
    mime[".rsml"]    = "application/vnd.rn-rsml";
    mime[".rt"]      = "text/vnd.rn-realtext";
    mime[".rtf"]     = "application/msword";
    mime[".rtf"]     = "application/x-rtf";
    mime[".rv"]      = "video/vnd.rn-realvideo";
    mime[".sam"]     = "application/x-sam";
    mime[".sat"]     = "application/x-sat";
    mime[".sdp"]     = "application/sdp";
    mime[".sdw"]     = "application/x-sdw";
    mime[".sis"]     = "application/vnd.symbian.install";
    mime[".sisx"]    = "application/vnd.symbian.install";
    mime[".sit"]     = "application/x-stuffit";
    mime[".slb"]     = "application/x-slb";
    mime[".sld"]     = "application/x-sld";
    mime[".slk"]     = "drawing/x-slk";
    mime[".smi"]     = "application/smil";
    mime[".smil"]    = "application/smil";
    mime[".smk"]     = "application/x-smk";
    mime[".snd"]     = "audio/basic";
    mime[".sol"]     = "text/plain";
    mime[".sor"]     = "text/plain";
    mime[".spc"]     = "application/x-pkcs7-certificates";
    mime[".spl"]     = "application/futuresplash";
    mime[".spp"]     = "text/xml";
    mime[".ssm"]     = "application/streamingmedia";
    mime[".sst"]     = "application/vnd.ms-pki.certstore";
    mime[".stl"]     = "application/vnd.ms-pki.stl";
    mime[".stm"]     = "text/html";
```

```
    mime[".sty"]     = "application/x-sty";
    mime[".svg"]     = "text/xml";
    mime[".swf"]     = "application/x-shockwave-flash";
    mime[".tdf"]     = "application/x-tdf";
    mime[".tg4"]     = "application/x-tg4";
    mime[".tga"]     = "application/x-tga";
    mime[".tif"]     = "image/tiff";
    mime[".tiff"]    = "image/tiff";
    mime[".tld"]     = "text/xml";
    mime[".top"]     = "drawing/x-top";
    mime[".torrent"] = "application/x-bittorrent";
    mime[".tsd"]     = "text/xml";
    mime[".ttf"]     = "application/font-woff";
    mime[".txt"]     = "text/plain";
    mime[".uin"]     = "application/x-icq";
    mime[".uls"]     = "text/iuls";
    mime[".vcf"]     = "text/x-vcard";
    mime[".vda"]     = "application/x-vda";
    mime[".vdx"]     = "application/vnd.visio";
    mime[".vml"]     = "text/xml";
    mime[".vpg"]     = "application/x-vpeg005";
    mime[".vsd"]     = "application/vnd.visio";
    mime[".vsd"]     = "application/x-vsd";
    mime[".vss"]     = "application/vnd.visio";
    mime[".vst"]     = "application/vnd.visio";
    mime[".vst"]     = "application/x-vst";
    mime[".vsw"]     = "application/vnd.visio";
    mime[".vsx"]     = "application/vnd.visio";
    mime[".vtx"]     = "application/vnd.visio";
    mime[".vxml"]    = "text/xml";
    mime[".wav"]     = "audio/wav";
    mime[".wax"]     = "audio/x-ms-wax";
    mime[".wb1"]     = "application/x-wb1";
    mime[".wb2"]     = "application/x-wb2";
    mime[".wb3"]     = "application/x-wb3";
    mime[".wbmp"]    = "image/vnd.wap.wbmp";
    mime[".wiz"]     = "application/msword";
    mime[".wk3"]     = "application/x-wk3";
    mime[".wk4"]     = "application/x-wk4";
    mime[".wkq"]     = "application/x-wkq";
    mime[".wks"]     = "application/x-wks";
    mime[".wm"]      = "video/x-ms-wm";
    mime[".wma"]     = "audio/x-ms-wma";
    mime[".wmd"]     = "application/x-ms-wmd";
    mime[".wmf"]     = "application/x-wmf";
    mime[".wml"]     = "text/vnd.wap.wml";
    mime[".wmv"]     = "video/x-ms-wmv";
    mime[".wmx"]     = "video/x-ms-wmx";
    mime[".wmz"]     = "application/x-ms-wmz";
    mime[".wp6"]     = "application/x-wp6";
    mime[".wpd"]     = "application/x-wpd";
    mime[".wpg"]     = "application/x-wpg";
    mime[".wpl"]     = "application/vnd.ms-wpl";
    mime[".wq1"]     = "application/x-wq1";
    mime[".wri"]     = "application/x-wri";
    mime[".wrk"]     = "application/x-wrk";
    mime[".wr1"]     = "application/x-wr1";
    mime[".ws"]      = "application/x-ws";
    mime[".ws2"]     = "application/x-ws";
    mime[".wsc"]     = "text/scriptlet";
    mime[".wsdl"]    = "text/xml";
    mime[".wvx"]     = "video/x-ms-wvx";
    mime[".xap"]     = "application/x-silverlight-app";
    mime[".xdp"]     = "application/vnd.adobe.xdp";
    mime[".xdr"]     = "text/xml";
    mime[".xfd"]     = "application/vnd.adobe.xfd";
    mime[".xfdf"]    = "application/vnd.adobe.xfdf";
    mime[".xhtml"]   = "text/html";
    mime[".xls"]     = "application/vnd.ms-excel";
    mime[".xls"]     = "application/x-xls";
```

```
    mime[".xlw"]     = "application/x-xlw";
    mime[".xml"]     = "text/xml";
    mime[".xpl"]     = "audio/scpls";
    mime[".xq"]      = "text/xml";
    mime[".xql"]     = "text/xml";
    mime[".xquery"]  = "text/xml";
    mime[".xsd"]     = "text/xml";
    mime[".xsl"]     = "text/xml";
    mime[".xslt"]    = "text/xml";
    mime[".xwd"]     = "application/x-xwd";
    mime[".x_b"]     = "application/x-x_b";
    mime[".x_t"]     = "application/x-x_t";
}

// 析构函数
ClientThread::~ClientThread(void) {
    delete[] body;
    delete[] header;
    delete[] req;

    if (ssl) {
        SSL_shutdown(ssl);
        SSL_free(ssl);
    }

    close(sockfd);
}

// 初始化I/O链
int ClientThread::initChain(void) {
    cout << "Initializing I/O chain ..." << endl;
    //
    // +--------+ +------------+ +-----+
    // | Socket |-| Socket BIO |-| SSL |
    // +--------+ +------------+ +-----+
    //
    // 创建套接字BIO
    BIO* sockbio = BIO_new_socket(sockfd, BIO_NOCLOSE);
    if (! sockbio) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    // 创建SSL对象
    if (! (ssl = SSL_new(ctx))) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    // 将SSL对象连到套接字BIO上
    SSL_set_bio(ssl, sockbio, sockbio);

    // 连接客户端
    if (SSL_accept(ssl) != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }
    //
    // +--------+ +------------+ +-----+ +---------+ +------------+
    // | Socket |-| Socket BIO |-| SSL |-| SSL BIO |-| BUFFER BIO |
    // +--------+ +------------+ +-----+ +---------+ +------------+
    //
    // 创建SSL协议BIO
    BIO* sslbio = BIO_new(BIO_f_ssl());
    if (! sslbio) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    // 将SSL协议BIO连到SSL对象上
```

```
    if (BIO_set_ssl(sslbio, ssl, BIO_CLOSE) != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    // 创建缓冲区BIO
    if (! (bufbio = BIO_new(BIO_f_buffer()))) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    // 将缓冲区BIO连到SSL协议BIO上
    BIO_push(bufbio, sslbio);

    cout << "Initializing I/O chain OK!" << endl;
    return 0;
}

// 接收请求包
int ClientThread::recvReq(void) {
    cout << "Receiving request ..." << endl;

    size_t reqlen = 0;

    for (;;) {
        char buf[1024];
        int len = BIO_gets(bufbio, buf, sizeof(buf) - 1);

        if (! len) {
            cerr << "Connection break" << endl;
            return -1;
        }

        if (SSL_get_error(ssl, len) != SSL_ERROR_NONE) {
            ERR_print_errors_fp(stderr);
            return -1;
        }

        memcpy(req + reqlen, buf, len);
        reqlen += len;

        if (! strcmp(buf, "\r\n") || ! strcmp(buf, "\n"))
            break;
    }

    req[reqlen] = '\0';
    cout << "----------------------------------------" << endl;
    cout << req;
    cout << "----------------------------------------" << endl;

    cout << "Receiving request OK!" << endl;
    return 0;
}

// 分析请求包
int ClientThread::analyzeReq(void) {
    // 若非法请求...
    if (strstr(req, "..")) {
        cerr << "Bad request" << endl;
        return -1;
    }

    // 获取方法
    const char* delimiters = " \n";
    char* token = strtok(req, delimiters);
    if (! strcmp(token, "HEAD")) // HEAD方法
        method = EMETHOD_HEAD;
    else if (! strcmp(token, "GET")) // GET方法
        method = EMETHOD_GET;
    else {
```

```
        cerr << "Not implemented" << endl;
        return -1;
    }
    cout << "Method: " << method << endl;

    // 获取统一资源标识符
    if ((token = strtok(NULL, delimiters)) != NULL)
        if (strlen(token) > 1)
            uri += token;
        else
            uri += "/index.html";
    else {
        cerr << "Bad request" << endl;
        return -1;
    }
    cout << "URI: " << uri << endl;

    // 获取保持连接属性
    token += strlen(token) + 1;
    if ((token = strcasestr(token, "connection:")) != NULL) {
        char connection[32];
        sscanf(token, "%*s%s", connection);
        if (! strcasecmp(connection, "keep-alive"))
            alive = true;
    }
    cout << "Keep Alive: " << alive << endl;

    return 0;
}

// 发送响应头
int ClientThread::sendHeader(void) const {
    cout << "Sending response header ..." << endl;

    // 日期时间
    char dt[32];
    time_t t = time(NULL);
    strftime(dt, sizeof(dt), "%a, %d %b %Y %T GMT", gmtime(&t));

    // 内容类型
    string postfix;
    string::size_type loc = uri.rfind('.');
    if (loc != string::npos)
        postfix = uri.substr(loc);
    map<string, string>::const_iterator it = mime.find(postfix);
    if (it == mime.end()) {
        cerr << "Invalid filename postfix" << endl;
        return -1;
    }

    // 内容长度
    struct stat st;
    if (stat(uri.c_str(), &st) == -1) {
        perror("stat");
        return -1;
    }

    // 格式化响应头
    sprintf(header,
        "HTTP/1.1 200 OK\r\n"
        "Date: %s\r\n"
        "Server: SecWebServer 1.0\r\n"
        "Content-Type: %s\r\n"
        "Content-Length: %ld\r\n\r\n",
        dt, it->second.c_str(), st.st_size);
    cout << "----------------------------------------" << endl;
    cout << header;
    cout << "----------------------------------------" << endl;

    int len = BIO_write(bufbio, header, strlen(header));
```

```
    if (SSL_get_error(ssl, len) != SSL_ERROR_NONE) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    if (BIO_flush(bufbio) != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    cout << "Sending response header OK!" << endl;
    return 0;
}

// 发送响应体
int ClientThread::sendBody(void) const {
    cout << "Sending response body ..." << endl;

    // 打开文件
    int fd = open(uri.c_str(), O_RDONLY);
    if (fd == -1) {
        perror("open");
        return -1;
    }

    for (;;) {
        ssize_t len = read(fd, body, BODYSIZE);
        if (len == -1) {
            perror("read");
            close(fd);
            return -1;
        }

        if (! len)
            break;

        len = BIO_write(bufbio, body, len);
        if (SSL_get_error(ssl, len) != SSL_ERROR_NONE) {
            ERR_print_errors_fp(stderr);
            close(fd);
            return -1;
        }

        if (BIO_flush(bufbio) != 1) {
            ERR_print_errors_fp(stderr);
            close(fd);
            return -1;
        }
    }

    close(fd);

    cout << "Sending response body OK!" << endl;
    return 0;
}

// 线程处理
void* ClientThread::run(void) {
    cout << "Client thread #" << syscall(SYS_gettid) << " ..." << endl;

    // 初始化I/O链
    if (initChain() == -1)
        goto escape;

    do {
        // 接收请求包
        if (recvReq() == -1)
            goto escape;

        // 分析请求包
```

```
        if (analyzeReq() == -1)
            goto escape;

        // 发送响应头
        if (sendHeader() == -1)
            goto escape;

        if (method == EMETHOD_GET)
            // 发送响应体
            if (sendBody() == -1)
                goto escape;
    }   while (alive);

    cout << "Client thread #" << syscall(SYS_gettid) << " OK!" << endl;

escape:
    delete this;
    return NULL;
}
```

5. 声明SecWebServer类

```
// secwebserver.h
// 声明SecWebServer类

#pragma once

#include <openssl/ssl.h>
#include <string>
using namespace std;

// 安全Web服务器
class SecWebServer {
public:
    // 构造函数
    SecWebServer(const char* port, const char* root);
    // 析构函数
    ~SecWebServer(void);

    // 运行
    int run(void);

private:
    // 密码回调
    static int passwordCallback(char *buf, int size, int rwflag,
        void* userdata);

    // 初始化上下文
    int initCtx(void);
    // 加载密钥交换参数
    int loadDhParams(void) const;

    // 初始化套接字
    int initSocket(void);
    // 等待并接受客户端的连接请求
    int acceptClient(void) const;

    const string port;   // 端口号
    const string root;   // 根目录
    SSL_CTX*     ctx;    // 上下文
    int          sockls; // 套接字文件描述符
};
```

6. 实现SecWebServer类

```cpp
// secwebserver.cpp
// 实现SecWebServer类

#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <string.h>
#include <openssl/err.h>
#include <iostream>
using namespace std;

#include "secwebserver.h"
#include "clientthread.h"

// 构造函数
SecWebServer::SecWebServer(const char* port, const char* root) :
    port(port), root(root), ctx(NULL), sockls(-1) {}

// 析构函数
SecWebServer::~SecWebServer(void) {
    // 关闭套接字
    if (sockls != -1)
        close(sockls);

    // 释放上下文
    if (ctx)
        SSL_CTX_free(ctx);
}

// 运行
int SecWebServer::run(void) {
    // 初始化上下文
    if (initCtx() == -1)
        return -1;

    // 加载密钥交换参数
    if (loadDhParams() == -1)
        return -1;

    // 初始化套接字
    if (initSocket() == -1)
        return -1;

    // 等待并接受客户端连接
    if (acceptClient() == -1)
        return -1;

    return 0;
}

// 密码回调
int SecWebServer::passwordCallback(char *buf, int size, int rwflag,
    void* userdata) {
    const char* password = "qgahome";
    int len = strlen(password);

    if(size < len + 1)
        return(0);

    strcpy(buf, password);
    return len;
}

// 初始化上下文
int SecWebServer::initCtx(void) {
    cout << "Initializing SSL context ..." << endl;

    // 初始化OpenSSL
```

```
    SSL_library_init();

    // 创建上下文
    ctx = SSL_CTX_new(SSLv23_method());

    // 设置密码回调函数
    SSL_CTX_set_default_passwd_cb(ctx, passwordCallback);
    //
    //    root_cer.pem
    // +----------------+
    // | CA CERTIFICATE |     root_key.pem
    // | -------------- |  +----------------+
    // | CA PUBLIC KEY  |  | CA PRIVATE KEY |
    // | CA INFORMATION |  +----------------+
    // | CA SIGNATURE   |
    // +----------------+
    //
    // root_key.pem -+
    //      v        +-> root_cer.pem
    // root_req.csr -+
    //
    // openssl genrsa -out root_key.pem 1024
    // openssl req -new -out root_req.csr -key root_key.pem
    // openssl x509 -req -in root_req.csr -out root_cer.pem
    //              -signkey root_key.pem -days 3650
    //
    // 加载受信任的CA证书
    if (SSL_CTX_load_verify_locations(
        ctx, "../pems/root_cer.pem", NULL) != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }
    //
    // openssl genrsa -out server_key.pem 1024
    //
    // 加载私钥文件
    if (SSL_CTX_use_PrivateKey_file(
        ctx, "../pems/server_key.pem", SSL_FILETYPE_PEM) != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }
    //
    //   server_cer.pem
    // +----------------+
    // | MY CERTIFICATE |    server_key.pem
    // | -------------- |  +----------------+
    // | MY PUBLIC KEY  |  | MY PRIVATE KEY |
    // | MY INFORMATION |  +----------------+
    // | MY SIGNATURE   |
    // | CA INFORMATION |
    // | CA SIGNATURE   |
    // +----------------+
    //
    // server_key.pem -+                    +- root_key.pem
    //       v         +-> server_cer.pem <-+
    // server_req.csr -+                    +- root_cer.pem
    //
    // openssl req -new -out server_req.csr -key server_key.pem
    // openssl x509 -req -in server_req.csr -out server_cer.pem
    //              -signkey server_key.pem -days 3650
    //              -CA root_cer.pem -CAkey root_key.pem -CAcreateserial
    //
    // 指定所使用的证书文件
    if(SSL_CTX_use_certificate_chain_file(
        ctx, "../pems/server_cer.pem") != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    cout << "Initializing SSL context OK!" << endl;
```

```
    return 0;
}

// 加载密钥交换参数
int SecWebServer::loadDhParams(void) const {
    cout << "Loading key exchange parameters ..." << endl;
    //
    // openssl dhparam -out dh1024.pem 1024
    //
    BIO* bio = BIO_new_file("../pems/dh1024.pem", "r");
    if (! bio) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    DH* dh = PEM_read_bio_DHparams(bio, NULL, NULL, NULL);
    BIO_free(bio);
    if (! dh) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    if (SSL_CTX_set_tmp_dh(ctx, dh) != 1) {
        ERR_print_errors_fp(stderr);
        return -1;
    }

    cout << "Loading key exchange parameters OK!" << endl;
    return 0;
}

// 初始化套接字
int SecWebServer::initSocket(void) {
    cout << "Initializing socket ..." << endl;

    // 创建套接字
    if ((sockls = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        return -1;
    }

    // 为套接字设置重用地址选项
    int on = 1;
    if (setsockopt(sockls, SOL_SOCKET, SO_REUSEADDR, &on,
        sizeof(on)) == -1) {
        perror("setsockopt");
        return -1;
    }

    // 绑定本地主机地址和端口
    struct sockaddr_in addr;
    socklen_t addrlen = sizeof(addr);
    bzero(&addr, addrlen);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(port.c_str()));
    addr.sin_addr.s_addr = INADDR_ANY;
    if (bind(sockls, (struct sockaddr*)&addr, addrlen) == -1) {
        perror ("bind");
        return -1;
    }

    // 设置套接字为侦听状态
    if (listen(sockls, 1024) == -1) {
        perror("listen");
        return -1;
    }

    cout << "Initializing socket OK!" << endl;
    return 0;
}
```

```
// 等待并接受客户端连接
int SecWebServer::acceptClient(void) const {
    for (;;) {
        cout << "Waiting for connection ..." << endl;

        // 等待并接受客户端的连接请求
        struct sockaddr_in addr;
        socklen_t addrlen = sizeof(addr);
        int sockfd = accept(sockls, (struct sockaddr*)&addr, &addrlen);
        if (sockfd == -1) {
            perror("accept");
            return -1;
        }

        cout << "Connect with " << inet_ntoa(addr.sin_addr) << ':' <<
            ntohs(addr.sin_port) << endl;

        // 创建客户线程
        ClientThread* clientThread = new ClientThread(sockfd, ctx, root);
        if (clientThread->start() == -1)
            delete clientThread;
    }

    return 0;
}
```

7. 测试SecWebServer类

```
// secwebserver_test.cpp
// 测试SecWebServer类

#include <stdlib.h>
#include <iostream>
using namespace std;

#include "secwebserver.h"

int main(int argc, char* argv[]) {
    const char *port = "8000", *root = "../root";
    if (argc > 1)
        port = argv[1];
    if (argc > 2)
        root = argv[2];

    // 安全Web服务器
    SecWebServer server(port, root);
    // 运行
    if (server.run() == -1)
        return EXIT_FAILURE;

    return EXIT_SUCCESS;
}
```

8. 测试SecWebServer类构建脚本

```
# secwebserver_test.mak
# 测试SecWebServer类构建脚本

PROJ   = secwebserver_test
OBJS   = secwebserver_test.o secwebserver.o thread.o clientthread.o
CXX    = g++
LINK   = g++
RM     = rm -rf
CFLAGS = -c -g -Wall -I.
LIBS   = -lssl -lcrypto -lpthread

$(PROJ): $(OBJS)
    $(LINK) $^ $(LIBS) -o $@

.cpp.o:
    $(CXX) $(CFLAGS) $^

clean:
    $(RM) $(PROJ) $(OBJS)
```

7.3 扩展提高

7.3.1 认证客户端

建立SSL连接时，客户端通常会要求服务器提供认证证书，认证通过后才能继续建立连接。相反服务器一般不会要求客户端提供认证证书。

所谓双向认证，即在客户端认证服务器的同时，服务器也认证客户端，双方都要向对方提供自己的认证证书，任何一方的认证没有通过，都不能建立SSL连接。

在OpenSSL中实现双向认证，只需在单向认证的基础上，设置客户端的可信任CA证书和要求客户端提供认证证书的属性即可。

1. 设置对客户端的可信任CA证书

```
int SSL_CTX_load_verify_locations(SSL_CTX* ctx, const char* cafile, const char* cadir);
```

例如：

```
SSL_CTX_load_verify_locations(ctx, "../pems/client_root_cer.pem", NULL);
```

2. 设置对客户端的可信任证书链深度

```
int SSL_CTX_set_verify_depth(SSL_CTX* ctx, int depth);
```

例如：

```
SSL_CTX_set_verify_depth(ctx, 1);
```

3. 设置要求客户端提供认证证书的属性

```
int SSL_CTX_set_verify(SSL_CTX* ctx, int mode, int (*verify_callback)(int, X509_STORE_CTX*));
```

例如：

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
```

7.3.2 基于IPSec的安全通信

SSL可以保证Web浏览器和Web服务器之间的安全通信，PGP和S/MIME可以实现安全的邮件传递，但所有这些安全技术都只能用于局部业务，并不能保证TCP/IP整体上的安全通信。为此，互联网工程任务组(The Internet Engineering Task Force, IETF)于1998年11月发布了IP安全标准IPSec，作为一种工作在开放互联网上的通用安全协议。

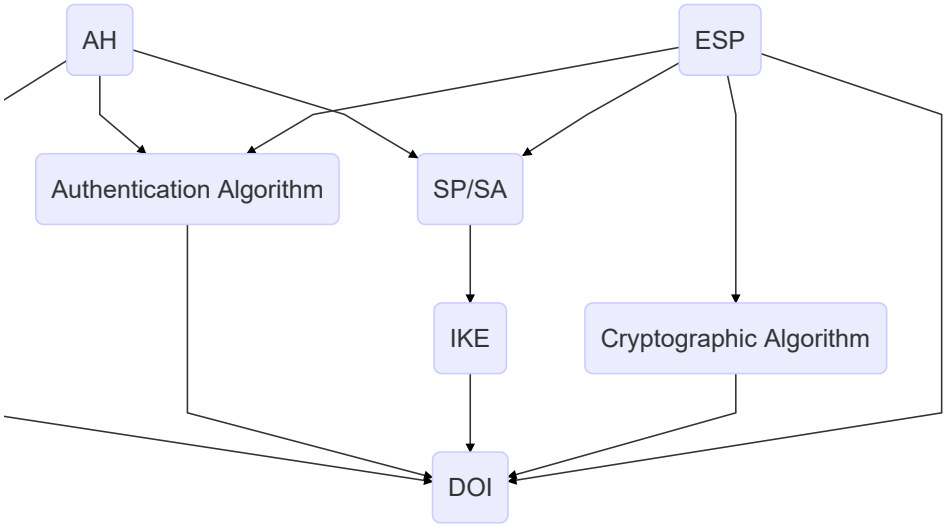
IPSec对IPv4是可选的，对IPv6则是强制的。IPSec是截至目前唯一一种可为任何形式的互联网通信提供安全保障的安全协议。同时，IPSec也是一套完整且易于扩展的基础网络安全解决方案。

1. IPSec的体系结构

IPSec协议是一个协议族，具体包括：

- 认证头(Authentication Header, AH)协议
- 封装安全载荷(Encapsulation Security Payload, ESP)协议
- 互联网密钥交换(Internet Key Exchange, IKE)协议
- 安全策略(Security Policy, SP)和安全联盟(Security Association, SA)
- 解释域(Domain of Interpretation, DOI)

IPSec的体系结构如下图所示：



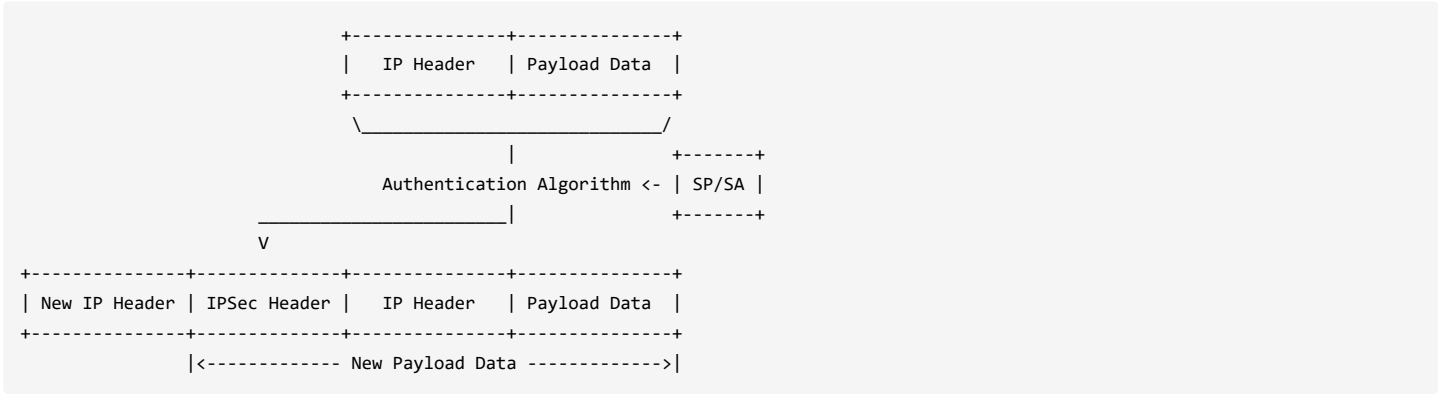
认证头(AH)和封装安全载荷(ESP)都是插入IP包数据载荷部分的协议头，为IP通信提供数据源认证、抗重播、数据完整性校验等安全服务。此外，封装安全载荷(ESP)还负责提供机密性服务。

安全策略(SP)决定两个实体之间能否通信以及如何通信。所有的安全策略(SP)都存放在安全策略库(Security Policy Database, SPD)中。针对库中的每条安全策略均可定义丢弃、绕过或应用IPSec三种行为中的一种。

那些被定义为应用IPSec的安全策略条目，均会指向一个或一串安全联盟(SA)。安全联盟(SA)包含针对IP包的各种安全参数，如安全协议、加密和认证算法、密钥及其生存周期、抗重播窗口大小等。

安全联盟(SA)既可以手动地静态创建也可以自动地动态创建，互联网密钥交换(IKE)会参与到动态创建安全联盟(SA)的过程中，以提供有关密钥协商的细节。

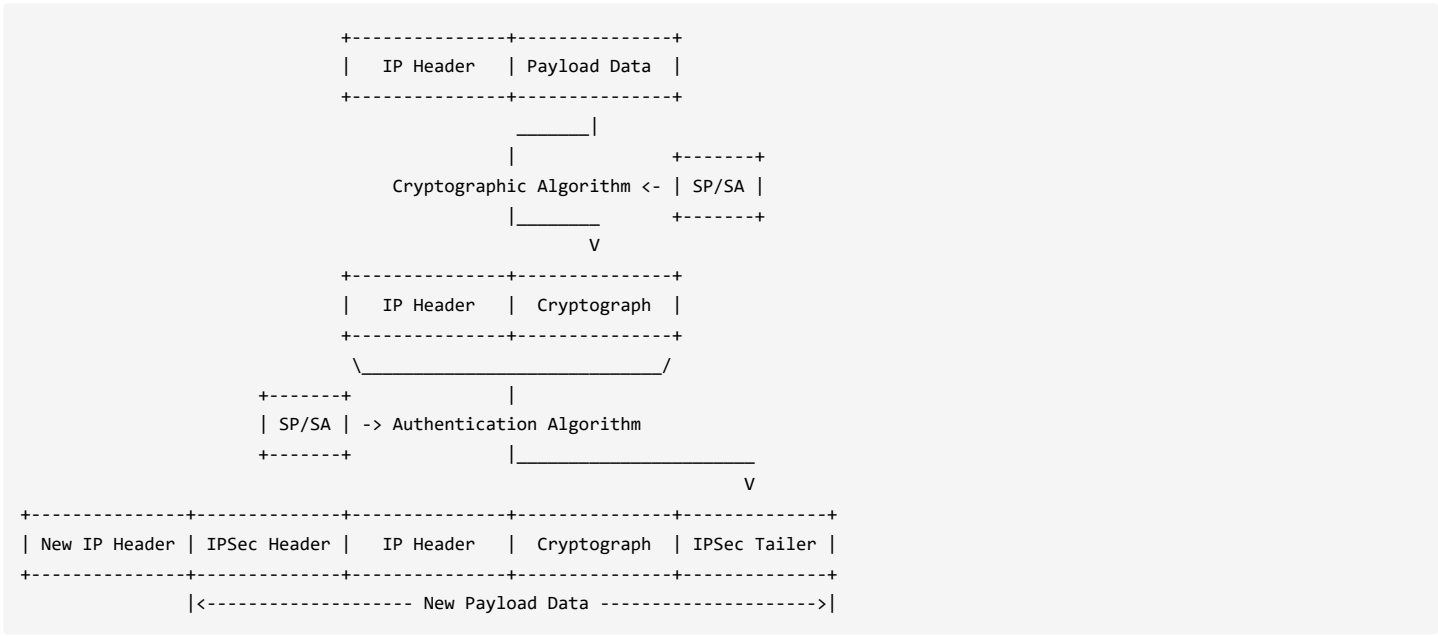
1) AH



AH的工作机制就是根据SP/SA中的认证算法，对整个包括上层协议在内的IP包计算消息摘要并签名，以形成包含认证数据在内的IPSec头，作为数据载荷的一部分放入IP包。

接收时，重新计算原始IP包的消息摘要，并验证IPSec头中的数字签名，验证失败者一律丢弃，以此实现针对数据源的身份鉴别和数据本身的完整性保护。

2) ESP



ESP的工作机制是先根据SP/SA中的加密算法，对包括上层协议在内的IP包数据载荷进行加密，然后根据SP/SA中的认证算法，对加密得到的密文连同IP包头计算消息摘要并签名，以形成包含认证数据在内的IPSec尾，连同包含加密信息在内的IPSec头一起作为数据载荷的一部分放入IP包。

接收时，重新计算密文连同IP包头的消息摘要，并验证IPSec尾中的数字签名，验签通过后再根据IPSec头中的加密信息对密文部分做解密，以此实现针对数据源的身份鉴别和数据本身的完整性与机密性保护。

3) IKE

IKE的主要任务就是在动态创建SA的过程中为其提供被称为“保护套件”的安全参数，其中包括：

- 加密算法
- 摘要算法
- 验证算法
- Diffie-Hellman组
  - Diffie-Hellman (DH)算法是一种公共密钥算法，通信双方在不传送密钥的情况下通过交换一些数据，计算出共享的密钥

IKE是一个用户态进程，系统启动后即以守护进程的方式运行于后台，可通过以下两种方式请求IKE服务：

- SP要求动态创建SA
- 远程IKE需要协商SA

IKE协商分为两个阶段：

- 第一阶段：通过协商创建一个经过认证的安全信道，为双方的后续通信提供机密性、完整性和源认证服务
- 第二阶段：在第一阶段所建SA的保护下完成IPSec的具体协商

IKE协商包括三对消息：

- SA交换消息：协商确认有关SP的细节
- 密钥交换消息：交换Diffie-Hellman公共值和辅助数据(随机数)
- 身份ID和认证数据交换消息：对身份和整个SA交换过程的认证

4) SP和SA

每个SP由选择符和策略项两部分组成：

- 选择符来自网络层和传输层，如源地址、源端口、目的地址、目的端口、传输层协议等
- 策略项为丢弃、绕过或应用IPSec三种行为之一

每个以应用IPSec作为策略项的SP均有一个SA与之对应，其中包含一系列经协商产生的安全约定：

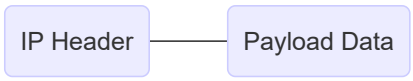
- 用于保护数据安全的IPSec协议，AH或者ESP
- 转码方式
- 密钥及其有效期

SA是单向的：

- 如果主机A和B通过ESP进行安全通信，那么主机A的SA (out)和主机B的SA (in)必须共享完全相同的安全参数
- 同理主机A的SA (in)和主机B的SA (out)也必须共享完全相同的安全参数

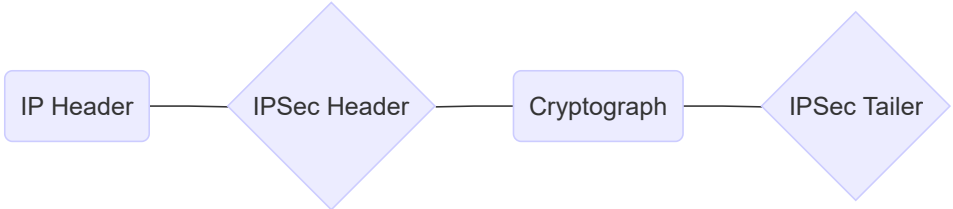
2. IPSec的工作模式

针对原始形态的IP包：

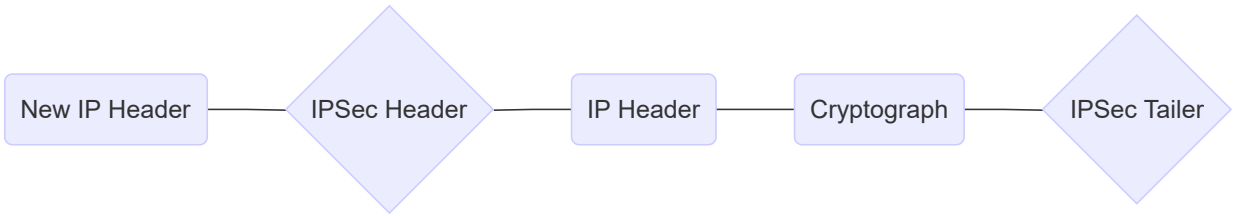


IPSec有两种工作模式：

- 传输模式：只保护传输层协议数据



- 隧道模式：保护整个IP层协议数据


New IP Header
IPSec Header
IP Header
Cryptograph
IPSec Tailer

达内集团◇C++教研部◇闵卫