ndnetengine 2.1 使用□ 明

1 概述

1.1 新增内容

在 2.0 版本的基础上进行优化,主要修改:

网络层优化,提高了 udt 协议的效率和稳定性;

增加用户自定义协议的支持;

明确划分网络层,协议层和运用层;

增加了对线程池的支持;

支持程序运行时对函数调用栈的跟踪,方便调试和查错.

1.2 目的

在开发了多个网络游戏服务器程序之后，逐步的把一些公共模块提出来经过优化形成了一个比较通用的网络游戏服务器开发底层套件。作为一个通用的网络底层套件，并不局限与 MMO，同样适应棋牌游戏和休闲游戏，当然也适应其他的网络运用。

Ndengine 的实现基本原则：

高效：

单个服务器提供尽可能多的连接；

尽可能快的网络响应。数据从客户端发出到服务器处理并返回的时间尽可能短。

简洁：

不对功能模块做过多的封装；

减少数据流的余冗，尽力实现零拷贝；

程序接口简洁；

结构简单，便于理解。

稳定：

减少 BUG；

防止内存泄漏，和内存碎片；

最终目的达到 7×24 小时的运行。

可移植：

支持 windows、linux 平台

为了方便移植，请使用 NDcommon 模块提供的函数，避免使用特定平台函数。

1.3 功能

当前版本位 2.1 版本，主要实现的功能:

平台无关（linux、windows）

网络层封装（支持 TCP 和 UDT－UDP 数据传输协议）

网络 IO（支持 IOCP 、epoll、非阻塞模式轮询模式）

内存池

数据加密、简易数字信封

线程池，多线程通讯，管理

多连接绘话管理

基本上实现了一个与服务器逻辑无关的底层网络中间件。

1。3

开发环境

Windows XP + VC9, VC10

linux Ubuntu 9.10 + GCC

1.4 版权

作者： neil duan

2010

email：luckduan@126.com

2 规范

2.1 文件目录

根目录

bin	**生产可执行文件**
build_clean.bat	**\\vc6 自动编译脚本**
build_debug.bat	
build_release.bat	
_build_all.bat	
cfg	**配置文件**
dbg_bin	**\\debug 可执行文件目录**
dbg_lib	**\\debug lib 库文件**
doc	**\\文档**
include	**\\头文件目录**
lib	**\\release lib 库**
Makefile	**\\linux makefile**

src	**源代码目录,包含以下目录**
nd_app	**\\C 语言版运用程序框架**
nd_appcpp	**\\C++语言版运用程序框架**
nd_cliapp	**\\客户段运用程序框架**
nd_common	**\\公共模块文件**
nd_crypt	**\\密码模块**
nd_net	**\\网络模块**
nd_srvcore	**\\服务核心模块**

demo **测试目录包含以下目录**

ndclient **\\客户端测试程序**

test **\\测试**

test_srv **\\服务器测试**

2.2 编码规范

nd engine 核心模块采用 ANSI C 开发,具有良好的移植性。编码规范类似于 linux 风格。主要遵循以下简单规则：

a 函数变量和结构名字以'nd_' 或者 'nd' 开的,名字之间用 '_' 连接

b 宏定义一般用是大写

c 函数命名一般规则 nd_modulename_functionname()

d 一般函数返回 0 是成功,返回-1 失败

e 不是 nd 开头为内部使用,建议不要直接使用

f 全局变量 以“_ _” 开头.

g C++类以' ND'开头名字之间大写.

3 模块说明

3.1 编译

在 linux 下使用以下命令即可

$ cd nd_engine

$ make

windows 下直接打开 vs 工程文件即可编译,不需要额外的依赖库.

目前版本使用的是 vc6,将来我的电脑升级会考虑 vc9 或者更高版本.

3.2 使用

命令执行成功会生成

nd_app.lib

nd_common.lib

nd_crypt.lib

nd_net.lib

nd_srvcore.lib

在工程文件中包含以上文件,并设置头文件包含路径 include 即可

根据配置不同会生成 release 和 debug 版本.

具体使用可以参考 nd_app 和 test_srv

4 结构流程

当前版本并实现服务器群组的,或者是限制用户使用什么样的策略,只是对一个纯网络服务器器程序提供支持,也就是你可以在此基础上构建你喜欢的任何类型.

这里将详细阐述 nd 网络套件的各个组成部分和数据流图.

4.1 名词解释

句柄(nd_handle)： 对象指针,主要是为了隐藏内部实现细节。

内存池： 管理内存的对象，可以分配释放内存。

连接器：能够使用特定协议连接指定的 IP 和端口的对象(客户端）。

连接绘话： 每个客户连接到服务器后，accept 之后会在服务器生成一个对象，称为一个绘话。

绘话 ID： 每个绘话有一个唯一的 16bit 的编号

监听器： 在服务器端打开一个端口，并在此端口上提供相应服务器，管理每个到来的客户端连接。

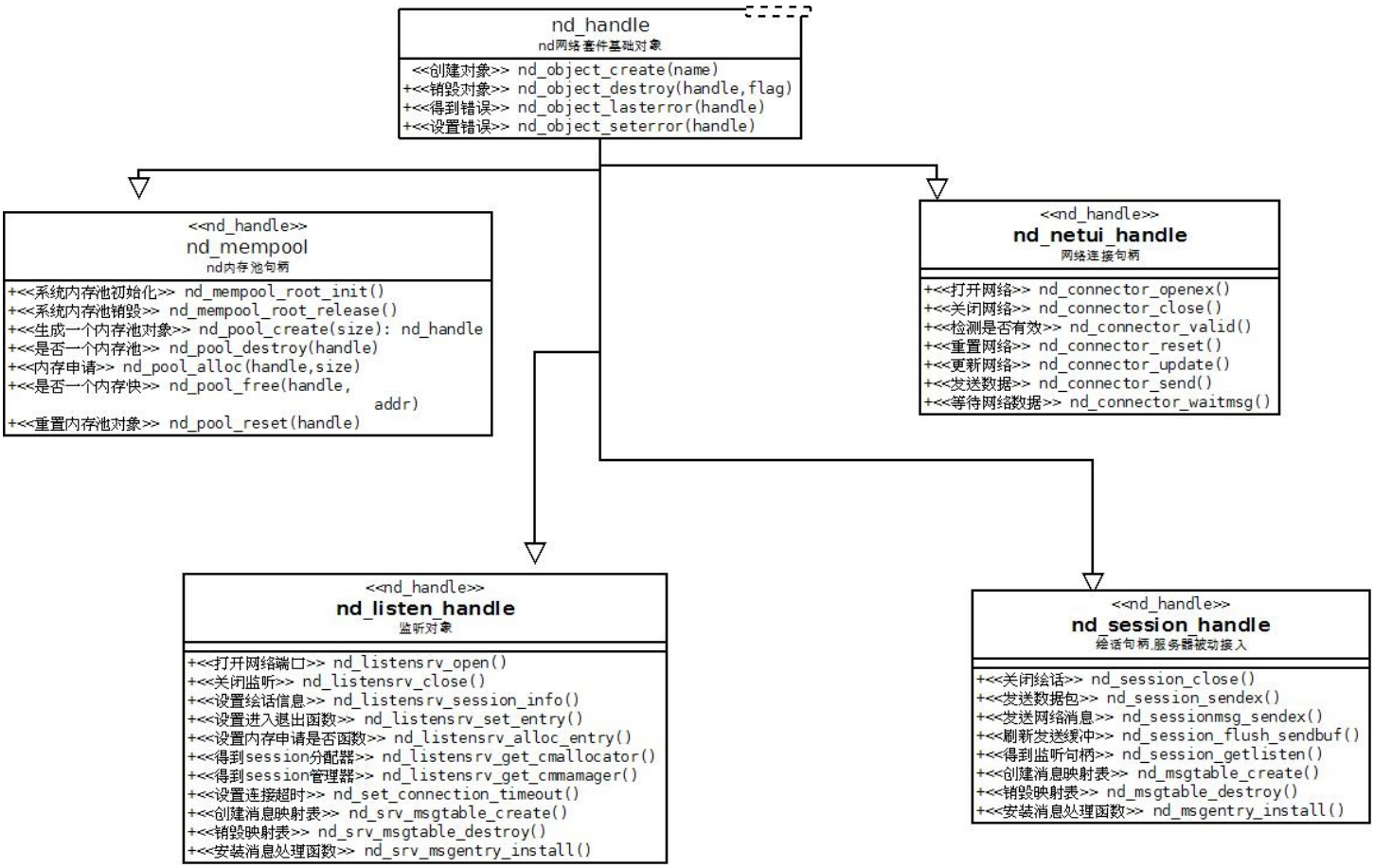
线程服务器： 对操作系统的线程封装，提供了线程间的通讯和其他管理功能。

绘话管理器： 管理每个监听器所属的绘话，提供了查找，遍历，迭代，互斥等功能。

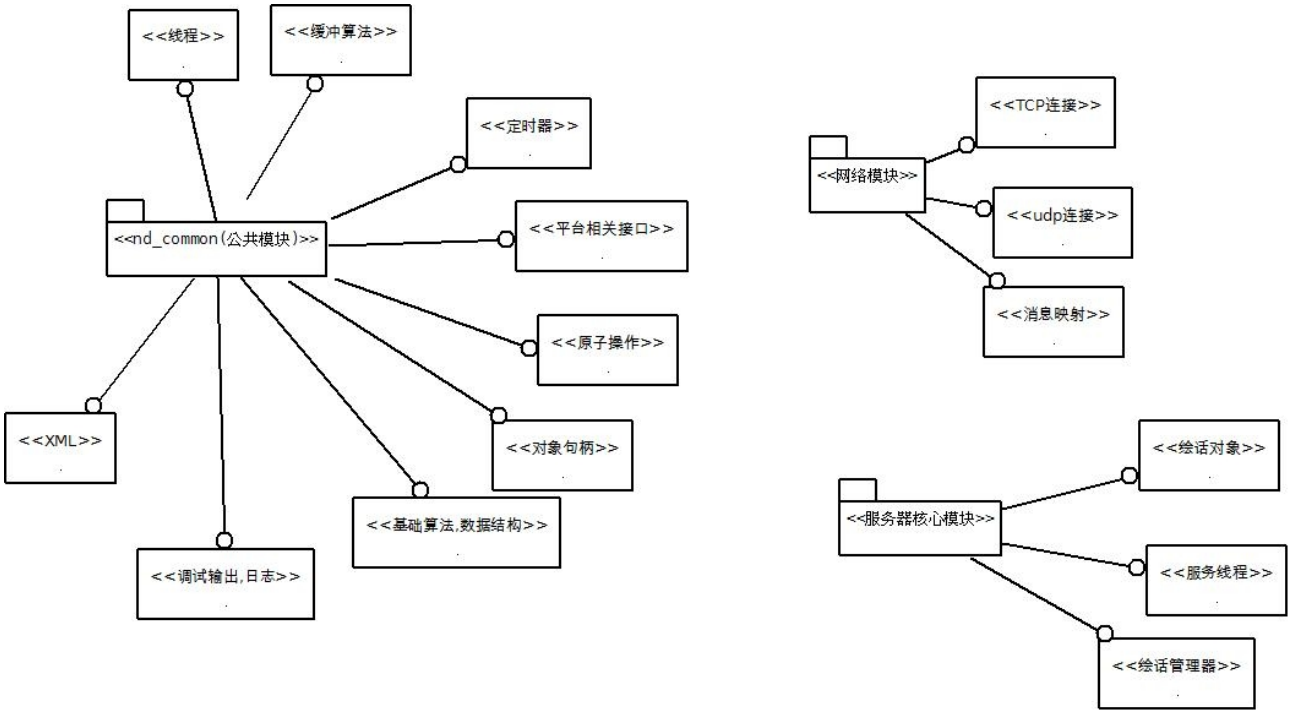
绘话分配器： 分配绘话内存。

消息映射表： 保存消息号对应的处理函数入口地址

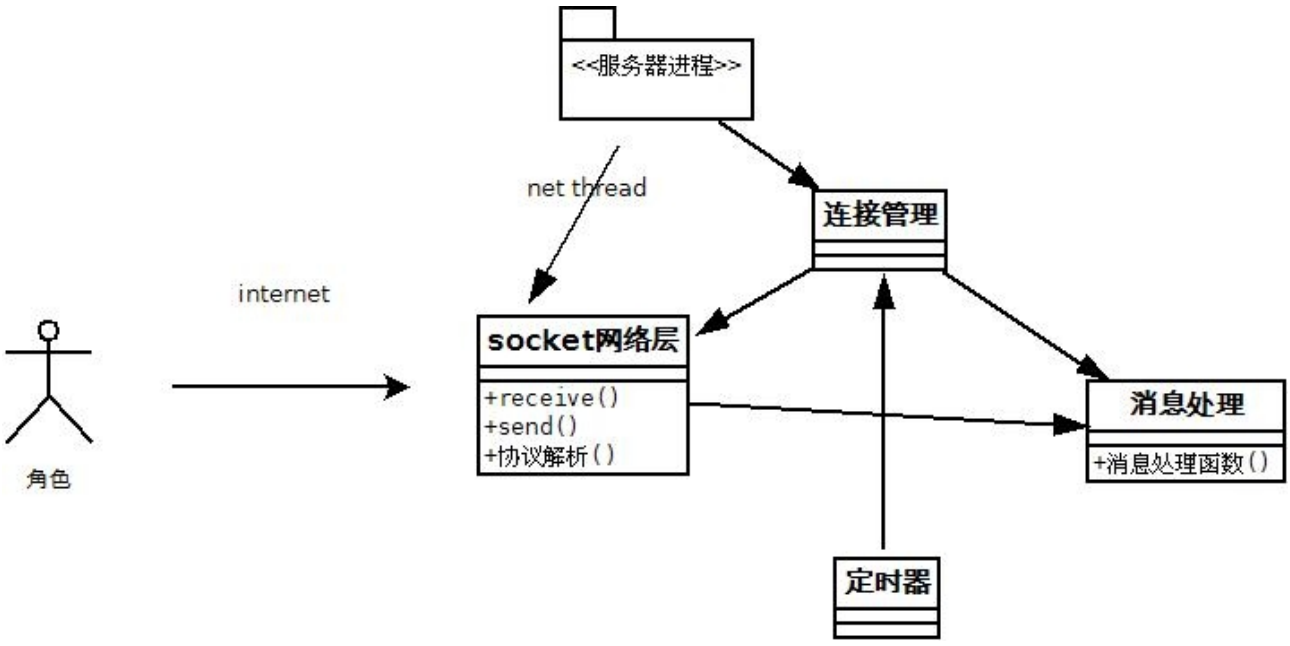
4.2 主要对象结构


nd_handle
nd网络套件基础对象
<<创建对象>> nd_object_create(name)
+<<销毁对象>> nd_object_destroy(handle,flag)
+<<得到错误>> nd_object_lasterror(handle)
+<<设置错误>> nd_object_seterror(handle)
<<nd_handle>>
nd_mempool
nd内存池句柄
+<<系统内存池初始化>> nd_mempool_root_init()
+<<系统内存池销毁>> nd_mempool_root_release()
+<<生成一个内存池对象>> nd_pool_create(size): nd_handle
+<<是否一个内存池>> nd_pool_destroy(handle)
+<<内存申请>> nd_pool_alloc(handle,size)
+<<是否一个内存快>> nd_pool_free(handle,
addr)
+<<重置内存池对象>> nd_pool_reset(handle)
<<nd_handle>>
nd_netui_handle
网络连接句柄
+<<打开网络>> nd_connector_openex()
+<<关闭网络>> nd_connector_close()
+<<检测是否有效>> nd_connector_valid()
+<<重置网络>> nd_connector_reset()
+<<更新网络>> nd_connector_update()
+<<发送数据>> nd_connector_send()
+<<等待网络数据>> nd_connector_waitmsg()
<<nd_handle>>
nd_listen_handle
监听对象
+<<打开网络端口>> nd_listensrv_open()
+<<关闭监听>> nd_listensrv_close()
+<<设置会话信息>> nd_listensrv_session_info()
+<<设置进入退出函数>> nd_listensrv_set_entry()
+<<设置内存申请是否函数>> nd_listensrv_alloc_entry()
+<<得到session分配器>> nd_listensrv_get_cmallocator()
+<<得到session管理器>> nd_listensrv_get_cmmamager()
+<<设置连接超时>> nd_set_connection_timeout()
+<<创建消息映射表>> nd_srv_msgtable_create()
+<<销毁映射表>> nd_srv_msgtable_destroy()
+<<安装消息处理函数>> nd_srv_msgentry_install()
<<nd_handle>>
nd_session_handle
会话句柄,服务器被动接入
+<<关闭会话>> nd_session_close()
+<<发送数据包>> nd_session_sendex()
+<<发送网络消息>> nd_sessionmsg_sendex()
+<<刷新发送缓冲>> nd_session_flush_sendbuf()
+<<得到监听句柄>> nd_session_getlisten()
+<<创建消息映射表>> nd_msgtable_create()
+<<销毁映射表>> nd_msgtable_destroy()
+<<安装消息处理函数>> nd_msgentry_install()

4.3 功能模块


<<线程>>
<<缓冲算法>>
<<定时器>>
<<nd_common(公共模块)>>
<<平台相关接口>>
<<原子操作>>
<<对象句柄>>
<<基础算法,数据结构>>
<<XML>>
<<调试输出,日志>>
<<网络模块>>
<<TCP连接>>
<<udp连接>>
<<消息映射>>
<<服务器核心模块>>
<<会话对象>>
<<服务线程>>
<<会话管理器>>

4.4 程序结构



Ndengine 主要的功能:

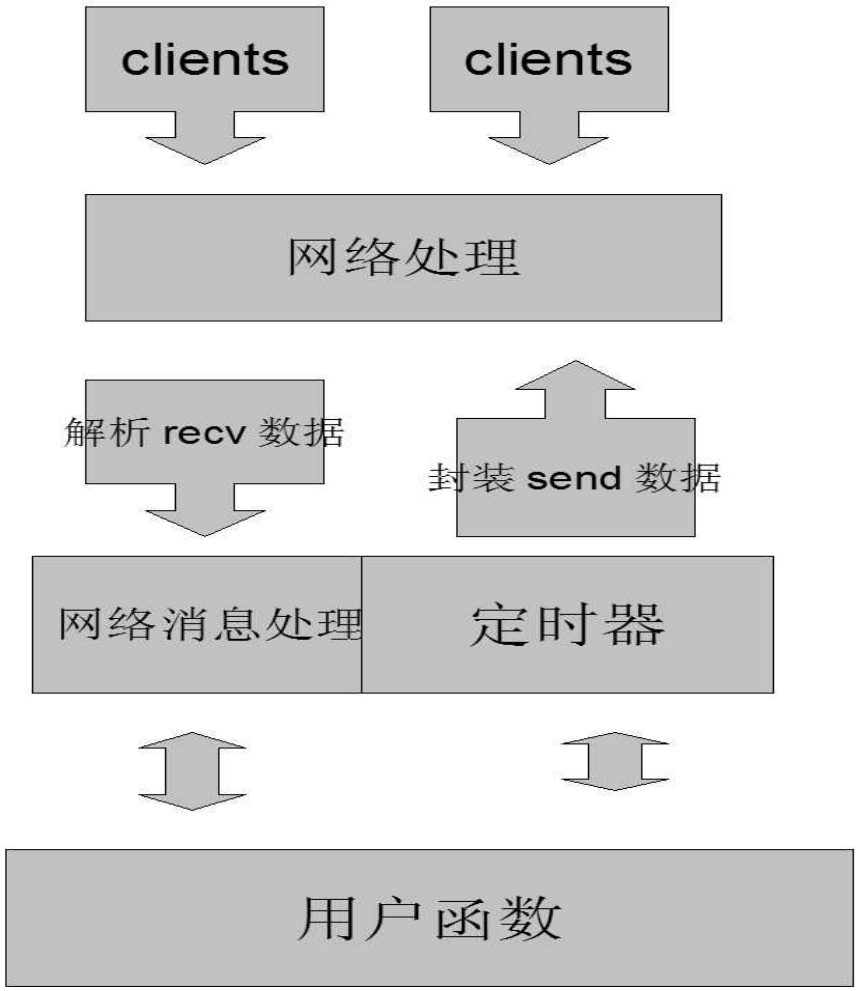
接收网络数据，分解为固定消息格式，并且根据网络消息的编号找到特定的执行函数。用户不需要关心网络协议和数据收发线程。同时提供了定时器和线程池。

对网络数据的处理支持立即模式和单线程模式，在立即模式（默认）中网络线程接收到数据会直接调用数据处理函数处理，这个过程中传递连接当前连接绘话给消息处理函数并锁住，如果需要访问其他绘话，必须通过绘话管理器锁定。

在单线程模式中所有的消息处理函数通过特定线程执行，不会导致线程的竞争，传递绘话 ID 给消息处理函数，整个过程与绘话句柄无关，用户程序无需关心绘话句柄。

要使用单线程模式，需要在监听器创建好以后，设置为单线程模式，使用 nd_listensrv_set_single_thread(h_listen) 函数。

4.5 数据流图



程序数据流图

除了上面介绍的功能 ndengine 还提供了很多公共的基础功能，在 ndcommon 模块中.

5 函数和数据结构

5.1 公共模块

头文件 ： include\nd_common

库文件： nd_common.lib

5.2 类型定义：

整型

NDINT8
NDUINT8
NDINT16
NDUINT16
NDINT32
NDUINT32

NDINT64

NDUINT64

在网络发送，或者需要使用指定长度时，请使用带指定长度的类型。

其他情况可以使用 c 内置类型。

NDBYTE： 字节，如果定义 ND_UNICODE 是 16bits 宽度否则 8bits 宽度。

ndchar_t ： 同 NDBYTE

nd_handle ：句柄，指向 nd_object 的指针。

ndth_handle：线程句柄 。
ndthread_t： 线程 id。

ndatomic_t： 原子类型。

ndsem_t ： 信号变量。

nd_mutex ： 互斥锁

nd_cond： 条件变量

ndtime_t： 时间类型 ms

5.3 函数：

int nd_common_init()

公共模块初始化，使用公共模块的功能需要先初始化。

void nd_common_release();

释放公共模块。

int nd_arg(int argc, char *argv[]);

设置运行参数，一般情况只需把 main 函数参数直接传递进去即可。

内存管理函数：

malloc()

free()

重定义标准库函数，只需包含 nd_alloc.h 文件即可使用。

对象操作：

nd_handle nd_object_create(char *name) ;

/* 创建一个对象实例,在 nd_create_object() 函数之前注册一个名字叫 name 的对象类型,就可以用 create 函数创建

* 返回对象的句柄

* 系统内置对象名字：

”udt-connector” ： udt 连接器。

,”tcp-connector”： tcp 连接器。

“lieten-tcp”： tcp 监听器，使用 ndengine 自定义网络 IO 模型。

“ listen-ext”, tcp 监听器，使用操作系统提供的 IO 模型，win32 使用 IOCP，linux 使用 EPOLL。

“listen-udt”： udt 监听器，使用 udt 监听器的服务器，客户端必须使用 udt 连接器。

*/

int nd_object_destroy(nd_handle handle, int force)

/* 销毁一个句柄, 如果是用 nd_object_create 函数创建的对象,则不必自己释放内存

* 否则需要手动释放内存

* force = 0 正常释放,等等所占用的资源释放后返回

* force = 1 强制释放,不等待

*/

int nd_object_register(struct nd_handle_reginfo *reginfo)

/* 注册一个对象类型,类似于 windows 窗口类型的注册

注册完成之后就可以使用创建函数创建一个对于的实例

@ reginfo 类型定义

```
struct nd_handle_reginfo
{
        unsigned int object_size ;              //句柄对象的大小
        nd_close_callback close_entry ;         //关闭函数
        nd_init_func   init_entry ;     //初始化函数
        char name[OBJECT_NAME_SIZE] ;           //对象名字
}
 */
```

NDUINT32 nd_object_lasterror(nd_handle h)

返回对句柄操作出错后最后一个错误号。

void nd_object_seterror(nd_handle h, NDUINT32 errcode)

设置一个错误号。

char *nd_object_errordesc(nd_handle h)

得到错误描述。

系统相关函数：

ndtime_t nd_time(void)

得到程序启动以来的时间滴嗒值，单位 MS

int kbhit (void)

检测是否敲击键盘，有键盘输入返回大于 0 值，用 getch()获得键盘输入。支持 linux 多种终端模式。

int getch()

等待键盘输入返回 ASCII 码

ndthread_t nd_thread_self()

得到现成自己的 id

ndthread_t nd_processid()

得到进程 ID

nd_sleep(ndtime_t tval)

让调用线程睡眠 tval MS，unix/linux 系列的 sleep 会导致整个进程睡眠。

nd_assert(a)

断言

内存池：

所有内存池函数在 ND 模块初始化以后可以使用。

nd_handle nd_pool_create(size_t size) ;

创建一个内存池,返回内存池句柄，内存池句柄不能使用通用对象创建函数 nd_object_create()创建。

@size 指定了内存池的上限，系统事先预定义了几种常用大小（emem_pool_type）。

Reuturn value： 成功返回内存池句柄，是否返回 NULL。

```
//内存池类型大小
enum emem_pool_type{
    EMEMPOOL_TINY =  32*1024,          //微型内存池
    EMEMPOOL_NORMAL = 256*1024,        //普通大小内存池
    EMEMPOOL_HUGE      = 1024*1024,    //巨型内存池
    EMEMPOOL_UNLIMIT = -1              //无限制内存池
};
```

int nd_pool_destroy(nd_handle pool, int flag);

销毁一个内存缓冲池，

◎flag 指定强制销毁 flag＝1 还是，正常销毁 flag＝0，一般情况都使用 flag＝0。

void *nd_pool_alloc(nd_handle pool , size_t size);

从缓冲池 pool 中申请一个长度为 size 的内存

Reuturn value：成功返回内存地址，失败返回 NULL。

void nd_pool_free(nd_handle pool ,void *addr) ;

释放从内存池中申请的内存，如果不是从内存池中申请的将会产生不可预知的错误。

void nd_pool_reset(nd_handle pool) ;

重新初始化一个已经申请好的内存池。

等效于：

nd_pool_destroy(pool, 0);

nd_pool_create(size) ;

nd_handle nd_global_mmpool() ;

得到全局默认的内存池。

系统初始化时会创建一个全局的内存池。

void *nd_global_alloc(size_t size) ;

void nd_global_free(void *p) ;

全局分配释放函数，在 global_mmpool 上实现。主要是底层实现 malloc 和 free 函数。

线程函数：

ndth_handle nd_createthread(NDTH_FUNC func, void* param,ndthread_t *thid,int priority);

创建线程函数

@func 线程函数，类型 void*NDTH_FUNC(void* param)

@param 线程函数参数

@thid 线程 ID

@ priority 线程优先级，可以有一下几个值

NDT_PRIORITY_NORMAL, 正常优先级

NDT_PRIORITY_HIGHT, 高优先级

NDT_PRIORITY_LOW 低优先级

int nd_threadsched(void) ;

强迫当前运行线程让出执行时间

void nd_threadexit(int exitcode);

退出当前运行的线程

int nd_waitthread(ndth_handle handle) ;

等待一个线程的结束

int nd_terminal_thread(ndth_handle handle,int exit_code);

结束一个线程

互斥与信号：

int nd_sem_init(ndsem_t *s)

初始化一个信号

void nd_sem_destroy(ndsem_t s)

销毁一个信号

int nd_sem_wait(ndsem_t s, ndtime_ttimeout)

等待信号

int nd_sem_post(ndsem_t s)

唤醒信号

互斥函数（用法同 posix)

int nd_mutex_init(nd_mutex *m)

void nd_mutex_lock(nd_mutex *m)

int nd_mutex_trylock(nd_mutex *m)　　//返回 0 表示成功

void nd_mutex_unlock(nd_mutex *m)

void nd_mutex_destroy(nd_mutex *m)

/条件变量（用法同 posix)

int nd_cond_init(nd_cond *c)

void nd_cond_destroy(nd_cond *c)

int nd_cond_wait(nd_cond *c, nd_mutex *m)

int nd_cond_timewait(nd_cond *c, nd_mutex *m, ndtime_t ms)

int nd_cond_signal(nd_cond *c)

int nd_cond_broadcast(nd_cond *c)

日志和错误输出

nd_logmsg(fmt,arg...)

输出日志信息

nd_logerror(fmt,arg...)

输出错误信息

nd_logfatal(fmt,arg...)

输出严重错误信息

nd_logwarn(fmt,arg...)

输出警告信息

void set_log_file(char *file_name) ;

指定输出日志文件名，默认日志名为可执行程序名字。

void NDTRAC(msg)

输出调试信息

NDUINT32 nd_last_errno() ;

得到最后出错的系统信息

const char *nd_last_error() ;

得到系统的最后一个错误描述(不是 nd_common 模块的)

void nd_showerror()

弹出错误描述的对话框，或者显示错误信息

函数栈追踪:

如果需要跟踪函数调用,必须手动在程序开始时初始化启动函数堆栈追踪器,然后在每个需要被追踪的函数手动记录,然后在其他进程中监控函数堆栈.

int nd_callstack_init(ndchar_t *filename) ;

初始化函数堆栈

int nd_callstack_end() ;

销毁堆栈

int nd_callstack_monitor_init(ndchar_t *filename) ;

初始化函数监控器

```
int nd_callstack_monitor_end() ;
```

结束函数监控器

```
int push_func(const char *funcname);
```

记录函数

```
void pop_func();
```

清除上一次记录的函数

```
int nd_callstack_monitor_dump(out_func func,FILE *outfile);
```

输出函数堆栈信息

XML 操作

int ndxml_load(const char *file,ndxml_root *xmlroot) ;

从文件中加载一个 xml 列表

ndxml *ndxml_getnode(ndxml_root *xmlroot, char *name) ;

ndxml *ndxml_getnodei(ndxml_root *xmlroot, int index) ;

查找到一个 xml 接点

ndxml *ndxml_addnode(ndxml_root *xmlroot, char *name,char *value) ;

添加一个 xml 节点

int ndxml_delnode(ndxml_root *xmlroot, char *name) ;

int ndxml_delnodei(ndxml_root *xmlroot, int index) ;

删除一个 XML 节点

void ndxml_destroy(ndxml_root *xmlroot) ;

销毁这个 xml 集合

int ndxml_save(ndxml_root *xmlroot, const char *file) ;

//把 xml 保存到文件中

ndxml *ndxml_refsub(ndxml *root, const char *name) ;

引用一个子节点

ndxml *ndxml_refsub(ndxml *root, const char *name) ;

引用一个子节点

ndxml *ndxml_refsubi(ndxml *root, int index) ;

通过下标引用一个子节点

char *ndxml_getval(ndxml *node);

得到 xml 的值

struct ndxml_attr *ndxml_getattrib(ndxml *node , char *name);
struct ndxml_attr *ndxml_getattribi(ndxml *node, int index);

得到属性节点

struct ndxml_attr *ndxml_addattrib(ndxml *parent, char *name, char *value) ;

给 xml 增加一个属性,

ndxml *ndxml_addsubnode(ndxml *parent, char *name, char *value) ;

给 xml 增加一个子节点需要输入新节点的名字和值,返回新节点地址

int ndxml_setval(ndxml *node , char *val) ;

设置 XML 的值

int ndxml_delattrib(ndxml *parent, char *name) ;
int ndxml_delattribi(ndxml *parent, int index) ;

//删除一个属性节点

int ndxml_delsubnode(ndxml *parent, char *name) ;
int ndxml_delsubnodei(ndxml *parent, int index) ;

删除一个子节点

char *ndxml_getname(ndxml *node)

得到节点名字

int ndxml_getattr_num(ndxml *node)

得到属性节点数目

int ndxml_num(ndxml_root *root)

得到 XML 节点数目

int ndxml_getsub_num(ndxml *node)

得到子节点数目

char *ndxml_getattr_name(ndxml *node, int index)

得到属性名字

char *ndxml_getattr_val(ndxml *node, char *name)

char *ndxml_getattr_vali(ndxml *node, int index)

得到属性值

xml_errlog nd_setxml_log(xml_errlog logfunc) ;

设置 xml 解析出错时的 log 函数,返回原先的函数地址

原子操作函数：

封装了原子操作功能，算法入下：

```
long nd_compare_swap(ndatomic_t *desc,ndatomic_t cmp, ndatomic_t exch)
{
	ndatomic_t tmp = *desc ;
	if(*desc==cmp) {
		*desc = exch ;
		return 1 ;
	}
	return 0 ;
}

ndatomic_t nd_atomic_inc(ndatomic_t *p)
{
```

```
        ++(*p ) ;
        return *p ;
}
ndatomic_t nd_atomic_dec(ndatomic_t *p)
{
        --(*p ) ;
        return *p ;
}
ndatomic_t nd_atomic_add(ndatomic_t *p,int step)
{
        ndatomic_t tmp = *desc ;

        *desc += step ;

        return tmp ;
}
ndatomic_t nd_atomic_sub(ndatomic_t *p,int step)
{
        ndatomic_t tmp = *desc ;

        *desc -= step ;

        return tmp ;
}
int  nd_atomic_swap(ndatomic_t *p, ndatomic_t val)
{
        ndatomic_t tmp = *desc ;
        *desc = val ;
        return tmp ;
}

int nd_testandset(ndatomic_t *p)
{
        int old = (int)*p ;
```

```
        if(p==0)
                p =1 ;
        return old;
}

void nd_atomic_set(ndatomic_t* p, ndatomic_t val)
{
        *p = val ;
}
void nd_atomic_read(ndatomic_t *p)
{
        return *p;
}
```

数据结构：

略

5.2 网络模块

数据结构：

nd_netui_handle　　网络连接句柄

struct netui_info　　网络句柄数据结构

```
typedef struct packet_hdr
{
        NDUINT16    length ;                    /*length of packet*/
} nd_packhdr_t;             网络封包头结构
```

网络封包

```
typedef struct nd_net_packet_buf
{
        nd_packhdr_t  hdr ;
        char                data[ND_PACKET_DATA_SIZE] ;
}nd_packetbuf_t;
```

函数：

int nd_net_init(void);

网络模块初始化

void nd_net_destroy(void);

销毁网络模块

连接器：

int nd_connector_send(nd_handle net_handle, nd_packhdr_t *msg_hdr, int flag)

```
/* 发送网络消息
* @net_handle 网络连接的句柄,指向 struct nd_tcp_node(TCP 连接)
*                  或者 ndudt_socket(UDT)节点
* @nd_msgui_buf 发送消息内容
* @flag ref send_flag
* return value:
                        on error return -1 ,else return send data len ,
```

发送标识定义

```
enum send_flag {
    ESF_NORMAL = 0 ,        //正常发送
    ESF_WRITEBUF =1,            //写入发送缓冲
    ESF_URGENCY = 2,        //紧急发送
    ESF_POST    = 4,        //不可靠的发送(可能回丢失)
    ESF_ENCRYPT = 8             //加密(可以和其他位连用|)
};
```

```
int nd_connector_open(nd_handle net_handle,char *host, int port, struct nd_proxy_info *proxy);
```

连接到远程主机端口上。

把已经创建的句柄连接到主机 host 的端口 port 上

@port remote port

@host host name

@ proxy 代理服务器信息

用法:

```
nd_handle connector = nd_object_create("tcp-connector")
        if(!connector) {
            //error ;
        }
        if(-1==nd_connector_open(connector, host, port,NULL) )
            // error
        nd_connector_close(connector, 0 ) ; // or nd_object_destroy(connector) ;
```

```
int nd_connector_close(nd_handle net_handle, int force) ;
```

关闭连接

```
ND_NET_API int nd_connector_valid(nd_netui_handle net_handle) ;
```

检测连接是否有效

```
int nd_connector_reset(nd_handle net_handle) ;
```

重置网络连接;

关闭网络连接并重新初始化连接状态,但保留用户设置信息(消息处理函数,加密密钥)

```
int nd_connector_update(nd_handle net_handle, ndtime_t timeout) ;
```

/*更新,驱动网络模块, 处理连接消息

* 主要是用在处理 connect 端,server 端不在此定义

* 出错放回-1,网络需要被关闭

* 返回 0 等待超时

* on error return -1,check errorcode ,

* return nothing to be done

* else success

```
 * if return -1 connect need to be closed
 */

size_t nd_connector_sendlen(nd_handle net_handle);
        /* 得到发送缓冲的空闲长度*/

void nd_connector_set_crypt(nd_handle net_handle, void *key, int size);
        设置加密密钥

int nd_connector_check_crypt(nd_handle net_handle) ;
        检测加密密钥是否有效

int nd_connector_waitmsg(nd_handle net_handle, nd_packetbuf_t *msg_hdr, ndtime_t tmout);
        /*等待一个网络消息消息
 *如果有网络消息到了则返回消息的长度(整条消息的长度,到来的数据长度)
 *超时,出错返回-1.网络被关闭返回 0
 */

int nd_packet_encrypt(nd_handle net_handle, nd_packetbuf_t *msgbuf);
        加密网络数据包


 int nd_packet_decrypt(nd_handle net_handle, nd_packetbuf_t *msgbuf);
        解密网络包

nd_connector_starttime(nd_handle net_handle)
        得到连接启动时间

void nd_set_close_reason(nd_handle handle, int reason)
        设置网络关闭原因

char *nd_connect_close_reasondesc(nd_netui_handle net_handle) ;
```

得到网络关闭的原因

消息处理函数：

用户消息类型 头

```
typedef struct nd_usermsghdr_t
{
        nd_packhdr_t packet_hdr ;          //消息包头
        ndmsgid_t           maxid ;              //主消息号 16bits
        ndmsgid_t           minid ;        //次消息号 16bits
        ndmsgparam_t        param;         //消息参数
}nd_usermsghdr_t ;

//定义用户消息包
typedef struct nd_usermsgbuf_t
{
        nd_usermsghdr_t msg_hdr ;
        char                data[ND_USERMSG_DATA_CAPACITY] ;
}nd_usermsgbuf_t;
```

消息处理函数：

```
/* 用户自定义消息处理函数
 * 在网络数据到达时会根据消息号执行相应的处理函数.
 * 函数返回值: 出错时返回-1, 系统会自动关闭对应的连接.
 * 需要关闭连接时,只需要在处理函数中返回-1 即可
 */
typedef int (*nd_usermsg_func)(nd_handle  handle, nd_usermsgbuf_t *msg , nd_handle listener);

//使用 session id 的消息处理函数，功能同上
```

typedef int (*nd_usermsg_func1)(NDUINT16 session_id, nd_usermsgbuf_t *msg , nd_handle listener);

/* 为连接句柄创建消息映射表

* @mainmsg_num 主消息的个数(有多数类消息

* @base_msgid 主消息开始号

* return value : 0 success on error return -1

*/

int nd_msgtable_create(nd_handle handle, int mainmsg_num, int base_msgid) ;

void nd_msgtable_destroy(nd_handle handle) ;

//销毁映射表，

@base_msgid 主消息开始的编号，为了避免没个监听器消息号都从 0 开始。

int nd_msgentry_install(nd_handle handle, nd_usermsg_func, ndmsgid_t maxid, ndmsgid_t minid,int level) ;

在 handle 连接句柄上安装消息处理函数

消息处理函数被安装在以 maxid 和 minid 为下标的数组中，这样可以很快找到处理函数。

@level 是函数执行的权限等级和 handle 的权限登记对应，只有在 handle 的权限登记大于消息处理函数要求的权限等级函数才会被执行，有效防止非授权访问。

int nd_connectmsg_send(nd_handle connector_handle, nd_usermsgbuf_t *msg)

正常发送网络消息，数据格式为消息格式，而不是封包格式。

int nd_connectmsg_post(nd_handle connector_handle, nd_usermsgbuf_t *msg) ;

不可靠的发送,消息可能会丢失

int nd_connectmsg_send_urgen(nd_handle connector_handle, nd_usermsgbuf_t *msg)

发送紧急数据,立即发送，不缓冲。

int nd_connectmsg_sendex(nd_handle connector_handle, nd_usermsgbuf_t *msg , int flag)

扩展发送函数，根据 flag 选择发送模式

```
 /* 把网络接口过来的消息传递给用户定义的消息,
 * 一般是网络解析层调用
 * @connect_handle 进入的消息句柄
 * @msg 消息内容
 * @callback_param 用户熟人参数
 */
int nd_translate_message(nd_handle  connector_handle, nd_packhdr_t *msg ) ;


/*
 * 服务器端消息传递函数
 * 把网络接口过来的消息传递给用户定义的消息,
 * 一般是网络解析层调用
 */
int nd_srv_translate_message(nd_handle listen_handle,nd_handle  connector_handle, nd_packhdr_t
*msg ) ;

//使用 sessionid 模式
int nd_srv_translate_message1(nd_handle listen_handle,NDUINT16 session_id, nd_packhdr_t
*msg ) ;

NDUINT32 nd_connect_level_get(nd_handle  connector_handle) ;
        得到权限等级

void nd_connect_level_set(nd_handle  connector_handle,NDUINT32 level);
        设置权限等级
int nd_connector_raw_waitdata(nd_handle net_handle, void *buf, size_t size, ndtime_t timeout) ;
        读取网络socket数据

data_in_entry nd_hook_data(nd_handle h, data_in_entry data_entry) ;
        拦截网络数据

 net_msg_entry nd_hook_packet(nd_handle h, net_msg_entry msg_entry) ;
        拦截消息处理函数
```

5.3 服务器核心模块

ND_SRV_API int nd_srvcore_init(void);　　//模块初始化函数
ND_SRV_API void nd_srvcore_destroy(void);

线程服务器：

nd_thsrvid_t nd_thsrv_createex(struct nd_thsrv_createinfo* create_info,int priority, int suspend) ;
nd_thsrvid_t nd_thsrv_create(struct nd_thsrv_createinfo* create_info)

创建线程服务器，

@ create_info　　线程服务器创建信息：

```
struct nd_thsrv_createinfo
{
    int run_module ;        //线程运行模式  (ref e_subsrv_runmod)
    nd_threadsrv_entry srv_entry ;        //线程函数
    void *srv_param ;                     //param of srv_entry
    nd_thsrvmsg_func msg_entry ;          //handle received message from other thread!
    nd_threadsrv_clean cleanup_entry ;  //clean up when server is terminal
    ndchar_t srv_name[ND_SRV_NAME] ;      //service name
    void *data ;                          //用户数据，保存用户需要的数据
};
```

@ priority

@ suspend

线程函数创建的参数类型定义

```
typedef int (*nd_threadsrv_entry)(void *param) ;            //service entry
typedef void (*nd_threadsrv_clean)(void) ;          //service terminal clean up fucnton
typedef int (*nd_thsrvmsg_func)(nd_thsrv_msg *msg) ;    //message handle function
enum e_thsrv_runmod{
    SUBSRV_RUNMOD_LOOP = 0,             //for(;;) srv_entry() ;
    SUBSRV_RUNMOD_STARTUP,              //return srv_entry() ;
    SUBSRV_MESSAGE          //创建一个单独的消息处理线程(主要是可以实现一个线程池,只处理消息的线程)
```

};

nd_handle nd_thsrv_gethandle(nd_thsrvid_t srvid);

得到线程服务器的句柄 srvid = 0 得到当前运行的线程句柄。

nd_thsrvid_t nd_thsrv_getid(nd_handle);

得到线程 ID

int nd_thsrv_destroy(nd_thsrvid_t srvid,int force);

销毁线程服务器 force=0 正常销毁 1 强制销毁

void nd_thsrv_release_all() ;

释放所有线程服务器

void nd_host_eixt() ;

/* 让所以线程的宿主退出,所以线程也退出*/

int nd_host_check_exit() ;

检测主机是否退出

int nd_thsrv_isexit(nd_handle) ;

int nd_thsrv_check_exit(nd_thsrvid_t srv_id) ;

检测线程是否退出

int nd_thsrv_end(nd_thsrvid_t srv_id) ; terminal a service

int nd_thsrv_suspend(nd_thsrvid_t srv_id) ; //wait a service exit

int nd_thsrv_resume(nd_thsrvid_t srv_id) ; //wait a service exit

int nd_thsrv_wait(nd_thsrvid_t srv_id) ; //wait a service exit

/* 线程服务器之间发送消息

- @data 消息数据

- @msgid 消息编号
- @ data_len 数据长度

*/

int nd_thsrv_send(nd_thsrvid_t recvid,NDUINT32 msgid,void *data, NDUINT32 data_len) ;

/* 处理用户消息

* 如果是用 SUBSRV_RUNMOD_STARTUP 模式来创建服务,则需要在服务程序中自行处理消息

* 一般用法是在服务循环中调用 nd_message_handler() ,参数中指定 context 是为了提高效率

* return 0 service(thread) exit

* return 1 message has been handled

*/

int nd_thsrv_msghandler(nd_handle thsrv_handle) ;

nd_handle nd_thsrv_local_mempool(nd_handle thsrv_handle) ;

得到线程使用的内存池。

监听器：

数据结构

nd_listen_handle 监听器句柄

nd_session_handle 绘话句柄

函数：

int nd_listensrv_open(int port, nd_listen_handle handle) ;

在指定端口打开网络,并且启动监听线程

使用这个函数之前，需要创建一个监听器对象

listen_handle = nd_object_create(“listen-ext”) ;

/*关闭网络关闭监听线程*/

int nd_listensrv_close(nd_listen_handle handle, int force) ;

/*设置对应连接的相关属性并分配内存*/

int nd_listensrv_session_info(nd_listen_handle handle, int max_client,size_t session_size) ;

/*设置连接进入和退出的回调函数*/

void nd_listensrv_set_entry(nd_listen_handle handle, accept_callback income,pre_deaccept_callback pre_close, deaccept_callback outcome) ;

/*设置 session 的 alloc 和 dealloc 还要 initilizer 函数

* 一般情况不要单独设置,除非你不想使用默认的分配和初始化函数.

*/

void nd_listensrv_alloc_entry(nd_listen_handle handle, cm_alloc al, cm_dealloc dealloc, cm_init init) ;

struct cm_manager *nd_listensrv_get_cmmamager(nd_listen_handle handle) ;

得到连接管理器

连接管理器是对但前监听器下的所有连接管理的对象。

管理器定义如下：

typedef void *(*cm_alloc)(nd_handle alloctor) ;

typedef void (*cm_init)(void *socket_node, nd_handle h_listen) ;

typedef void (*cm_dealloc)(void *socket_node,nd_handle alloctor) ;

typedef void(*cm_walk_callback)(void *socket_node, void *param) ;

typedef NDUINT16 (*cm_accept)(struct cm_manager *root, void *socket_node);

typedef int (*cm_deaccept)(struct cm_manager *root, NDUINT16 session_id);

typedef int (*cm_inc_ref)(struct cm_manager *root, NDUINT16 session_id);

typedef void (*cm_dec_ref)(struct cm_manager *root, NDUINT16 session_id);

typedef void *(*cm_lock)(struct cm_manager *root, NDUINT16 session_id);

typedef void *(*cm_trylock)(struct cm_manager *root, NDUINT16 session_id);

typedef void (*cm_unlock)(struct cm_manager *root, NDUINT16 session_id);

typedef void (*cm_walk_node)(struct cm_manager *root,cm_walk_callback cb_entry, void *param);

typedef void *(*cm_search)(struct cm_manager *root, NDUINT16 session_id);

迭代器

```
typedef struct cmlist_iterator
{
        NDUINT16 session_id ;
        NDUINT16 numbers ;
}cmlist_iterator_t ;

typedef void* (*cm_lock_first)(struct cm_manager *root,cmlist_iterator_t *it);
```

//所住下一个,同时释放但前输入 session_id 但前被锁住的 ID,输出 session_id 已经被所住的下一个 ID,并且释放当前被锁住的对象

```
typedef void* (*cm_lock_next)(struct cm_manager *root, cmlist_iterator_t *it);
typedef void (*cm_unlock_iterator)(struct cm_manager *root, cmlist_iterator_t *it) ;
```

管理器主要结构，主要是函数指针

```
struct cm_manager
{       cm_alloc            alloc;  //从特定分配器上分配一个绘话内存
        cm_init                     init ;   //初始化一个绘话节点
        cm_dealloc          dealloc ; //从分配器上释放绘话块内存
        //define connect manager function
        cm_accept           accept ;        //把绘话添加到管理器
        cm_deaccept         deaccept ;      //从管理器注销绘话
        cm_inc_ref          inc_ref ;       //增加绘话的使用计数
        cm_dec_ref          dec_ref ;       //减少绘话的使用计数
        cm_lock                     lock;           //锁定绘话
        cm_trylock          trylock ;       //尝试锁定绘话
        cm_unlock           unlock ;        //绘话解锁
        cm_walk_node        walk_node ;     //遍历所有绘话
        cm_search           search;         //查找绘话

        cm_lock_first lock_first ;      //锁住第一个绘话(开始迭代)
        cm_lock_next lock_next ;            //锁住下一个绘话(迭代下一个)
        cm_unlock_iterator  unlock_iterator;        //解锁迭代器(退出迭代)
```

};
//----------------------end----------------------

void nd_set_connection_timeout(nd_listen_handle h, int second)

设置绘话超时时间

ndtime_t nd_get_connection_timeout(nd_listen_handle h)

得到绘话超时时间。

绘话：

监听器会给每个进入的连接建立一个绘话，绘话用法类似连接，但不需要手动创建。

int nd_session_close(nd_handle session_handle, int force);

关闭绘话

nd_session_sendex(nd_handle session_handle,nd_packhdr_t *msg_buf, int flag) ;

发送一个封包格式的数据，根据 flag 选择不同的发送方式

flag 参考 enum send_flag 。

int nd_sessionmsg_sendex(nd_handle session_handle,nd_usermsghdr_t *msg, int flag)

发送消息格式数据。

nd_sessionmsg_writebuf(session,msg)	//写消息到缓冲
nd_sessionmsg_send(session,msg)	//正常发送
nd_sessionmsg_post(session,msg)	//不可靠发送
nd_sessionmsg_send_urgen(session,msg)	//立即发送
nd_session_flush(session)	//把缓冲区数据发送出去
nd_session_tryto_flush(session)	//尝试发送缓冲区数据
nd_session_flush_force(session)	//强制发送缓冲区数据

int nd_session_flush_sendbuf(nd_handle session_handle, int flag) ;

发送缓冲操作函数

int nd_session_broadcast(nd_handle listen_handle, nd_usermsghdr_t *msg, NDUINT16 send_sid) ;

广播数据给所有绘话

如果是在消息处理函数中使用,需要指定但前消息处理函数是对应那个绘话,并把绘话 ID 传给 send_sid,否会造成死锁.

nd_handle nd_session_getlisten(nd_handle session_handle)

得到绘话所属的监听器

NDUINT16 nd_sessionid_get(nd_handle session_handle)

得到绘话 ID

单线程模式:

为了简化用户程序编写，可以把所有的用户消息和定时器处理函数指定到一个特定的线程，这样可以认为这个服务器是以单线程模式运行，简化逻辑复杂度，提供开发效率。但消息处理函数将有少许不同。

int nd_listensrv_set_single_thread(nd_handle h_listen) ;

设置为单线程模式

在创建监听对象后，设置其他属性之前先设置这个属性，否则会提示出错！

int nd_listensrv_set_entry_st(nd_handle h_listen, st_accept_entry income, st_deaccept_entry outcome) ;

设置绘话进入和关闭的回调函数。

//单线程模式的接入和关闭函数

typedef int(* st_accept_entry) (NDUINT16 sessionid, SOCKADDR_IN *addr, nd_handle listener) ;

typedef void(* st_deaccept_entry) (NDUINT16 sessionid, nd_handle listener) ;

int nd_srv_msgentry_install_st(nd_handle listen_handle, nd_usermsg_func1 fn, ndmsgid_t maxid, ndmsgid_t minid,int level)

安装消息处理函数，入口函数定义入下：

typedef int (*nd_usermsg_func1)(NDUINT16 session_id, nd_usermsgbuf_t *msg , nd_handle listener);

int nd_st_send(NDUINT16 sessionid , nd_usermsghdr_t *msg, int flag, nd_handle h_listen) ;

单线程模式下的发送函数

int nd_st_close(NDUINT16 sessionid , int flag, nd_handle h_listen) ;

单线程模式下的关闭函数

void nd_st_set_crypt(NDUINT16 sessionid, void *key, int size,nd_handle h_listen);

设置加密密钥

5.4 运用程序框架

为了方便代码编写，把所有初始化和是否函数，封装在一个运用框架之中，用户程序只需要关心网络消息处理即可。

通过配置文件可以对运用框架提供相应的设定。

需要包含 nd_app 头文件和 nd_app.lib

int start_server(int argc, char *argv[]);

启动服务器

通过命令行启动 xx.exe -f config.xml

需要指定配置文件

int end_server(int force);

结束服务

int wait_services() ;

等待服务退出。

nd_handle get_listen_handle() ;

得到监听器句柄

5.5 处理用户自定义数据类型

处理用户自定义数据需要拦截网络数据,具体用法参考 proxy

如果希望 C++风格的话需要包含 nd_appcpp 头文件和库文件。

6 范例

```
#pragma comment(lib,"nd_app.lib")

#include "nd_app/nd_app.h"

int main(int argc, char *argv[])
{
	if(0!=start_server(argc, argv)) {
		exit(1) ;
	}
	wait_services() ;
	printf_dbg("leave wait server\n") ;
	end_server(0) ;
	printf_dbg("program exit from main\n Press ANY KEY to continue\n") ;
	getch();
	return 0;
}
```

启动 server.exe -f ../cfg/runconfig.xml

xml 配置文件内容

```
<server_config name="config">
	<run_config>
		<logfile>srvlog.log</logfile>
```

```
        <port>7828</port>
        <listen_mod desc="可以使用  lieten-tcp, listen-ext, listen-udt">listen-ext</listen_mod>
        <max_connect name="最大连接数">200</max_connect>
    </run_config>
</server_config>
```

使用 C++风格

在 nd_appcpp 力提供了 C++的支持

```
#include "nd_appcpp/nd_instance.h"
#pragma comment(lib,"nd_appcpp.lib")
#pragma comment(lib,"nd_app.lib")

int main(int argc, char *argv[])
{
    NDInstanceSrv nd_instance ;

    NDInstanceSrv::InitApp(argc, argv) ;

    nd_instance.InitServer(sizeof(NDSession));

    nd_instance.StartServer() ;

    nd_instance.WaitServer() ;

    nd_instance.EndServer(0) ;

    NDInstanceSrv::DestroyApp() ;

    printf_dbg("program exit from main\n Press ANY KEY to continue\n") ;
    getch();
    return 0;
}
```

7 测试

我尽量保证每个函数的每个分支都通过测试，包括黑盒白盒测试，也包含了压力测试。单元测试代码在 test\目录 和 ndclient\目录。

8 未来

进一步提供效率和稳定，也包括对其他硬件平台的支持。