zsxxsz的博客文章

作者: zsxxsz http://zsxxsz.iteye.com

我的博客文章精选

目 录

1.1 dmd.2.029 编译过程

发表时间: 2009-05-19 关键字: Linux, RedHat, DOS, Unix, Windows

D编译完全开源了，于是忍不住下载了最新的dmd编译源码(dmd.2.029),然后在Redhat AS3上进行编译，但是还是遇到了一些小问题。下面将详细过程介绍一下：

1) 先解压

unzip dmd.2.029.zip

便生成目录 dmd, 该目录下有：src/, linux/, windows/samples/ 等目录

2) 编译dmd编译器

cd dmd/src/dmd/

make

便会报错

backend/dwarf.c:54:26: ../mars/mars.h: No such file or directory

于是创建目录同时要拷贝两个文件至新建目录：

make mars

cp mars.h mars/

cp complex_t.h mars/

OK，编译器完毕，将以下几个可执行文件拷贝至 dmd/linux/bin/ 目录下：

cp dmd idgen optabgen impcnvgen ../../linux/bin/

3) 编译运行时库

cd dmd/src/druntime/src

chmod 755 build-dmd.sh

./build-dmd.sh

于是报错

: No such file or directory

为找到各种原因，本人花了好久，终于搞明白是因为 build-dmd.sh 的原始格式为DOS格式，

所以需要转换为 unix 格式

于是乎用工具将 build-dmd.sh 由DOS格式转换为UNIX格式, 再运行：

./build-dmd.sh

Ok

4) 编译 phobos 库，里面包含常用的函数库

cd dmd/src/phobos

make -f linux.mak

于是报错

make: *** No rule to make target `../druntime/lib/libdruntime.a', needed by `obj/posix/release/

libphobos2.a'. Stop.
解决方法如下：
cd dmd/src/druntime/lib/
ln -s release/libdruntime.a libdruntime.a
然后
cd dmd/src/phobos
make -f linux.mak
ok, 编译 phobos 成功，同时会自动将库 libphobos2.a 拷贝至 dmd/linux/lib/ 目录

5) 最后将 dmd/src/druntime/lib/release/ 目录下的所有库拷贝至 dmd/linux/lib/ 目录，
同时修改环境变量设置，将 xxx/dmd/linux/bin, xxx/dmd/linux/lib 放入个人环境变量中

不知即将发布的 dmd.2.030 会不会将这些小问题都给解决了:)
另外，本人在编译 dmd/samples/ 下的例子时，发现基本上是编译不过去，查看了源码，原来里面用的库基本都是 dmd.1.0 的库，希望 Walter Bright 等人将此类问题都解决了，呵呵

注：以上仅是将DMD.2.029 在Redhat AS4的编译过程，如果想要在AS3上编译通过，还需要将附件的 linux.mak 替换 src/phobos/ 目录下的 linux.mak;
补充：如果你用的DMD.2.030在 AS3上编译，则需要将 linux.2.030.rar 替换 src/phobos/ 下的 linux.mak;
附件下载:

- linux.rar (4.2 KB)
- dl.iteye.com/topics/download/facd2b24-c834-3ea7-844b-066399d5db27
- linux.2.030.rar (4.2 KB)
- dl.iteye.com/topics/download/aa9d8d5e-162e-3c18-991d-f8efadcbe661

2.1 acl介绍

发表时间: 2010-01-21 关键字: 网络应用, 网络协议, C, C++, C#

一、acl 是什么？

其实是一个很简单的问题，acl 的英文字母 advanced C library 的缩写（当然，您也可以认为是 a C library 的缩写）。也许有人会问："现在有这么多C的函数库，为何还费这么大劲再写一个？"。的确，现在开源的 C 函数库真是太多了，比如：glib（这是一个gnome工作组开发维护的库，开始也是为了gnome界面用，后来发展成通用的 C 函数库），libevent（这是UNIX平台下一个封装了 select/poll/epoll/kquue 等的库，主要是为了应对大并发网络程序的开发），pthread-win32（这是在win32平台下按Posix接口标准写的库，其中google的 chrome 浏览器就用到了它），当然还有更为著名的 C++ 函数库 ACE（还有自称比ACE还好的C++库ICE）。每个库都有各自的特点，都能满足不同的应用需求。但是，当我在开发一个希望能跨平台的、支持常见数据算 法、有很好的服务器框架、支持线程池/进程池、支持同步/异步通信、简单易用可扩展、支持HTTP协议、ICMP协议、DNS协议等的应用时，就不得不组 合各个函数库，有时还得为这些库的不兼容性做些努力。

二、acl 是怎么来的？

因此写一个通用简单的 C 库就成为了一个小小的目标，开始 acl 的名字叫 util，生成的库名叫 lib_util.a，并且该库的函数大部是从 Postfix 借鉴过来的，也就是说初期没有 Postfix 就不会有 acl。Postifx 虽然只是一个邮件的 mta（现在国内的很多大的网站的MTA就是由 Postfix 修改而成，象 sohu, 263, sina, qq 等的邮件系统），但其中的函数库却设计的如此巧夺天工， 虽然是用 C 语言写成，但其设计思想却可与 C++ 媲美，Wietse Venema 不愧为世界级架构师，否则IBM也就不会请他来写 Postfix 了，在读了 Postfix 的源码后，本人将其中一些具有通用性的函数库(象vstring.c, htable.c, ring.c, etc)抽出来，构成了 lib_util.a 的主体部分。随着时间的演变， lib_util 里加的函数库越来越多，并且在与其它函数库连用时很容易造成命名冲突，因为 C 语言没有象 C++ 命名空间的概念，一般的做法都是在函数名前加 xxx 前后缀来区分。所以，是应该重新给 lib_util 起个名字的时候了，象 tcl(turbo c library), 等名字都被用过，但总觉得有点别扭，有一天突然想到用 acl 似乎也不错，嗯，advanced C library - - - 高级 C 库，意思是应该比 ANSI C 的标准库提供更为高级的功能。acl 名称确定下来后，于是把函数的名称都加了小写的 acl_ 前缀，结构、宏、全局定义的名称前都加了 ACL_ 前缀，确实是个不错的命名。

三、acl 的目录划分

开始时 acl 的目录规划比较简单，只有 src/ 和 include/ 两个目录，所有的源代码及头文件都分别放在这两个目录下，当然这也主要是因为 Postifx 的 util/ 目录下是通用的函数库的原因（当初就是想把 Postfix 的 util/ 下的库单独抽出来），当发现函数库越来越多时，发现将所有源代码放在同一目录下维护起来是多么地麻烦，所以想应该规划一下目录结构了，于是乎，划分了几个 一级目录：stdlib/, event/, aio/, thread/ 等目录，其中在 stdlib/ 目录下还有 common/, memroy/, string/, configure/ 等二级目录。这样，acl 的体系结构算是基本形成了。现在 acl 的目录划分基本是以功能为标准的，各个目录的主要功能如下：

3.1 src 目录

3.1.1 init/ : 主要用于初始化 acl 基础库

3.1.2 stdlib/ : 是一些比较基础的功能函数库，在 stdlib/ 根目录下主要包括一些有关日志记录、网络/文件流处理、VSTRING缓冲操作等功能函数；在 stdlib/ 下还有二级目录，如下：

3.1.2.1 common/ : 该目录主要为一些常用的数据结构及算法的功能函数库，象哈希表、链表、队列、动态数组、堆栈、缓存、平衡二叉树、模式匹配树等；

3.1.2.2 memory/ : 该目录主要包含与内存操作相关的函数库，象内存基础分配与校验、内存池管理、内存切片管理等；

3.1.2.3 filedir/ : 该目录主要包含与目录遍历、目录创建等相关的库；

3.1.2.4 configure/ : 该目录主要包含配置文件的分析库；

3.1.2.5 iostuff/ : 该目录主要包含一些常用的IO操作的函数库，象读/写超时、设置IO句柄的阻塞模式等；

3.1.2.6 string/ : 该目录主要包含一些常用的字符串操作的库，提供了比标准C更灵活高效的字符串操作功能；

3.1.2.7 debug/ : 主要用于协助调试内存的泄露等功能；

3.1.2.8 sys/ : 主要是与不同操作系统平台相关的API的封装函数库；

3.1.3 net/: 是与网络操作相关的函数库，包含网络监听、网络连接、DNS查询、套接口参数设置等功能；

3.1.3.1 connect/ : 主要是与网络连接相关的函数库，包含网络连接、域套接口连接等；

3.1.3.2 listen/ : 主要是与网络监听相关的函数库，包含网络监听、域套接口监听等；

3.1.3.3 dns/ : 主要是与DNS域名查询相关的函数库，包含对 gethostbyname 等接口的封装、按RFC1035标准直接发送UDP包方式进行查询等功能；

3.1.4 event/ : 主要封装了 select/poll/epoll/kqueue/devpoll 等系统API接口，使处理网络事件更加灵活、高效、简单，另外还包含定时器接口，acl 中的很多网络应用都会用到这些接口，象 aio、master 等模块；

3.1.5 aio/ : 主要包含网络异步操作的功能函数，该套函数库在处理高并发时有非常高的效率，而且提供了比基础API更为高级的调用方式，比使用象 libevent 之类的函数库更为简单，而且是线程安全的；

3.1.6 msg/ : 主要包含了基于线程的消息事件及基于网络的消息事件功能；

3.1.7 thread/ : 主要是封装了各个OS平台下的基础线程API，使对外接口保持一致性，消除了平台的差异性，同时还提供了半驻留线程池的函数库，以及对于线程局部变量的扩展；

3.1.8 db/ : 主要是一些与数据库有关的功能库，定义了一个通用的数据库连接池的框架（并且实现了mysql的连接池实例）；一个简单的内存数据库（由哈希表、链表、平衡二叉树组合而成）；ZDB数据存储引擎，这是一个高效的基于数字键的存储引擎；

3.1.9 proctl/ : win32 平台下父子进程控制功能库；

3.1.10 code/ : 常见编码函数库，包括 base64编解码、URL编解码以及一些汉字字符集编码等；

3.1.11 unit_test/ : 包含有关进行 C 语言单元测试的功能库；

3.1.12 xml/：是一个流式的 xml 解析器及构造器，可以支持阻塞及阻塞式网络通信；

3.1.13 json/：是一个流式的 json 解析器及构造器，可以支持阻塞及阻塞式网络通信；

3.1.14 master/: 是在 UNIX 环境下支持多种服务器模式的服务器框架，目前主要支持多进程模式、多进程多线程模式、多进程非阻塞模式以及多进程触发器模式；

四、acl 是如何跨平台的？是如何支持WIN32平台的？

在开发 acl 库时是完全基于 linux 平台的，后来公司要做一个P2P的项目，需要在WIN32下开发客户端程序，主要需要 acl 中的 HTTP 协议通信及协议解析部分，所以为了项目需要， acl 中的HTTP协议库及HTTP协议库所依赖的一些函数库被移植至WIN32下。由此，觉得 acl 应该有一个 win32 版本，所以本人花费数周N个晚上，将 acl 库中的几乎所有函数库都移植至WIN32下（当然目前仅有master服务器框架还没移植），后来还移植至FreeBSD、Solaris(x86)上（从LINUX移植至这些UNIX平台相对容易些），主要还是在移植至WIN32平台时比较费劲，因为WIN32的API与UNIX及POSIX的真是相 差太远了，感觉一点就是在WIN32下直接用API编程还是挺辛苦的，有时本来一个非常简单的IO读写，在WIN32下却需要N个参数，并且微软的API 还有一个特点，就是很多API都有一个Ex（扩展）版，看来是它们的工程师在当初设计API时发现不够用又加进去的。

另外，微软的编译器版本比较多，有VC6，VC2002，VC2003，VC2005，VC2008，现在还有VC2010，各个编译器之间还 有一些差别，要想完全支持这些编译器工作量是巨大的，所以acl 目前可以用 VC6, VC2003, VC2008 编译，说起仅支持这些编译器还是有原因的，VC6做为一个老牌的编译器，存在的历史时期比较久，自 Borland 的BC被VC打败后，微软就在VC编译器无所作为了，N年未大改过VC6，说实在话，这个VC6也真是够难用的（当然现在还有许多人在用它），后来微软终 于出VC2002，算是VC6的一个大版本升级，终于变得“比较”好用了，不过VC2002似乎存在不少BUGS，所以微软又推出了VC2003，这个版 本本人觉得还是很不错的，可以说是VC6的终极版，因为VC2005后，微软在VC编译器的改动就比较大了，显著一点就是比较浪费系统资源了，对于象我这 样用惯 vi 编辑程序的程序员，机器配置并不需要太高，而用VC2005时明显机器的CPU及内存不够用的，本人感觉 VC6, VC2002,VC2003 属于一个系列的，标准C库变动不是特别大，但到了VC2005后就变动比较多了，比如在VC2003时一般用 snprintf 就可以了，但用VC2008时，你会发现微软“自作聪明”地提示你应该用 snprintf_s ，而且多了几个参数，为了支持VC2008，acl库里还是做了不少改动。但本人还是比较偏爱VC2003，它仅不会象VC2005/VC2008那样费 资源，又比VC6好用，而且没有VC2002那么多BUGS，所以 acl 库在WIN32的主推编译器是VC2003:)。

五、acl 从哪些开源软件中汲取了经验？

首先得感谢 Postfix(2002-2005本人在方标以此作为方标邮件系统的MTA模块), 里面良好的框架设计及编码风格是 acl 始终追求的；squid(2005-2007年本人在和讯改造SQUID以适应和讯的网络访问需求)/nginx，这两款HTTP代理加速软件在HTTP协议处理方面给本人了很多启示；ircd 是一个比较“古老”的聊天服务器(2000-2002年本人在263做的263 web聊天室的后台服务器就是基于 ircd)，也是本人最早接触的服务器程序，而且同 squid/nginx 一样是非阻塞通讯模式。

acl 下载

acl 的编译与使用

更多文章

3.1 acl 的编译与使用

发表时间: 2012-05-01

acl 库的功能参见文章 <acl介绍>，本文主要讲述如何编译和使用 acl 库。

acl 下其实有四个库：lib_acl （基础库）、lib_protocol（http 和 icmp 协议库）、lib_dict（封装了 bdb, cdb, Tokyo Cabinet 库的用于字典查询的库）以及 lib_tls（封装了openssl部分功能的库，主要用于 lib_acl 的 ssl 加密传输）。其中，笔者用的最多还是 lib_acl 和 lib_protocol 两个库，所以本文主要介绍这两个库的编译与使用。

开始时 acl 库是支持 Linux、Solaris、FreeBSD 和 Windows 平台的，但后来由于受限于机器环境，所以现在仅支持到 Linux 和 Windows 两个平台了，其它几个平台欢迎读者进行移植。

一、Linux 平台上编译

1、编译 lib_acl.a 库

进入 lib_acl/ 目录，直接运行 make，正常情况下便可在 lib/ 目录下生成 lib_acl.a 静态库，用户在使用 lib_acl.a 编写自己的程序时，需要在自己的 Makefile 文件中添加如下选项：

1.1、编译选项：-I 指定 lib_acl/include 所在目录，-DLINUX2 指定 Linux 平台

1.2、链接选项：-L 指定 lib_acl.a 所在目录，-l_acl 指定需要链接 lib_acl.a 库

同时，用户需要在自己的源程序中包含 lib_acl 的头文件，如下：

#include "lib_acl.h"

2、编译 lib_protocol.a 库

进入 lib_protocol/ 目录，直接运行 make，正常情况下便可以在 lib/ 目录下生成 lib_protocol.a 静态库，用户在使用 lib_protocol.a 编写自己的程序时，需要在自己的 Makefile 文件中添加如下选项：

2.1、编译选项：-I 指定 lib_protocol/include 所在目录，-DLINUX2 指定 Linux 平台

2.2、链接选项：-L 指定 lib_protocol.a 所在目录，-L 指定 lib_acl.a 所在目录，-l_protocol -l_acl

同时，用户需要在自己的应用程序中包含 lib_protocol 头文件，如下：

```
#include "lib_protocol.h"
```

二、Windows 平台

2.1 编译

进入 acl\win32_build\vc 目录，用 vc2003 打开工程文件：acl_project_vc2003.sln （原来还支持VC6及VC2008，但也好久没有更新这两个工程文件了）。用户可以选择编译 lib_acl、lib_protocol 的静态库调试版、静态库发布版、动态库调试版以及动态库发布版，编译完成后，会在 acl\dist\lib\win32 目录生成的静态库有：

lib_acl_vc2003d.lib、lib_acl_vc2003.lib、lib_protocol_vc2003d.lib 和 lib_protocol_vc2003.lib；
生成的与动态库相关的文件有：lib_acl_d.dll/lib_acl_d.lib，lib_acl.dll/lib_acl.lib，lib_protocol_d.dll/lib_protocol_d.lib 和 lib_protocol.dll/lib_protocol.lib。

2.2 使用

a）在 win32 平台下使用 lib_acl 和 lib_protocol 静态库时，只需要在包含目录中添加 lib_acl/include 和 lib_protocol/include 所在的路径，在链接时指定静态库路径及静态库名称。

b）在 win32 平台下使用 lib_acl 的动态库时，不仅要做与 a） 中所指定的操作，而且需要在预处理器定义中添加：ACL_DLL；在使用 lib_protocol 的动态库，需要在预处理器定义中添加：HTTP_DLL 和 ICMP_DLL。

acl 下载

原文地址

更多文章

4.1 C语言中迭代器的设计与使用

发表时间: 2009-09-23 关键字: C, C++, C#, D语言, 数据结构

经常使用C + +、JAVA等面向对象语言开发的程序员都会比较喜欢容器的迭代器功能，用起来方便简洁。象一些常用的数据结构，如：哈希表、动态数组、链表等，在这些面向对象语言中都可以非常方便地使用迭代器。当然，在C语言中也有对这些常用数据结构的函数封装，但要对容器中元素的遍历，则一般会通过注册回调函数的方式。如下：

```
/* 以C语言中非常流行的 glib 库中的哈希表操作为例 */

static void print_record(gpointer key, gpointer val, gpointer ctx)
{
    printf("%s: key(%s), value(%s)\n", (char*) ctx, (char*) key, (char*) val));
}

static void free_record(gpointer key, gpointer val, gpointer ctx)
{
    printf("%s: free(%s) now\n", (char*) ctx, (char*) key);
    free(val);
}

static void htable_test(void)
{
    char  *myname = "hash_test";
    char  key[32], *value;
    GHashTable *table;
    int   i;

    /* 创建哈希表 */
    table = g_hash_table_new(g_str_hash, g_str_equal);

    /* 依次向哈希表中添加数据 */
    for (i = 0; i < 10; i++) {
```

```
        snprintf(key, sizeof(key), "key:%d", i);
        value = malloc(64);
        snprintf(value, 64, "value:%d", i);
        g_hash_table_insert(table, key, value);
    }

    /* 遍历并输出哈希表中的数据 */
    g_hash_table_foreach(table, print_record, myname);

    /* 依次释放哈希表中的数据 */
    g_hash_table_foreach(table, free_record, myname);

    /* 销毁哈希表 */
    g_hash_table_destroy(table);
}
```

这是C函数库中比较常用的回调函数方式，它主要有两个缺点：多写了一些代码，使用不太直观。下面介绍一下ACL库中的设计与实现是如何克服这两个缺点的。首先先请看一个ACL库中使用哈希表的例子：

```
void htable_test(void)
{
        ACL_HTABLE *table = acl_htable_create(10, 0);  /* 创建哈希表 */
        ACL_ITER iter;  /* 通用迭代器对象 */
        char  key[32], *value;
        int   i;

        /* 依次向哈希表中添加数据 */
        for (i = 0; i < 20; i++) {
                snprintf(key, sizeof(key), "key: %d", i);
                value = acl_mymalloc(32);
                snprintf(value, 32, "value: %d", i);
                assert(acl_htable_enter(table, key, value));
        }

        printf("\n>>>acl_foreach for htable:\n");
```

```
        /* 正向遍历哈希表中数据 */
        acl_foreach(iter, table) {
                printf("hash i=%d, [%s]\n", iter.i, (char*) iter.data);
        }

        /* 释放哈希表中数据 */
        acl_foreach(iter, table) {
                acl_myfree(iter.data);
        }

        /* 销毁哈希表 */
        acl_htable_free(table, NULL);
}
```

由以上例子可以明显看出ACL库中的哈希表遍历更加简单直观，不需要回调函数方式便可以遍历哈希表中的所有元素。ACL库不仅哈希表可以用 "ACL_ITER iter; acl_foreach(iter, hash_table) {}" 的方式进行遍历，其它的通用数据结构容器都可以如此使用，如ACL库中的先进先出队列：ACL_FIFO 使用迭代器的例子：

```
static void fifo_iter(void)
{
        ACL_FIFO fifo;
        ACL_ITER iter;
        ACL_FIFO_INFO *info;
        int   i;
        char *data;

        /* 初始化堆栈队列 */
        acl_fifo_init(&fifo);

        /* 向队列中添加数据 */
        for (i = 0; i < 10; i++) {
                data = acl_mymalloc(32);
                snprintf(data, 32, "data: %d", i);
                acl_fifo_push(&fifo, data);
        }
```

```
        printf("\n>>> acl_foreach for fifo:\n");
        /* 正向遍历队列中数据 */
        acl_foreach(iter, &fifo) {
                printf("i: %d, value: %s\n", iter.i, (char*) iter.data);
        }

        printf("\n>>> acl_foreach_reverse for fifo:\n");
        /* 反向遍历队列中数据 */
        acl_foreach_reverse(iter, &fifo) {
                printf("i: %d, value: %s\n", iter.i, (char*) iter.data);
        }

        /* 弹出并释放队列中数据 */
        while (1) {
                data = acl_fifo_pop(&fifo);
                if (data == NULL)
                        break;
                acl_myfree(data);
        }
}
```

可以看出，ACL库中的迭代器都是同样的东东 ACL_ITER, 遍历方式也都一样，这是如何做到的？下面请先看一下ACL库中 ACL_ITER 结构的定义：

```
#ifndef      __ACL_ITERATOR_INCLUDE_H__
#define __ACL_ITERATOR_INCLUDE_H__

typedef struct ACL_ITER ACL_ITER;

/**
 * ACL 库中数据结构用的通用迭代器结构定义
 */
struct ACL_ITER {
        void *ptr;              /**< 迭代器指针，与容器相关 */
```

```
        void *data;             /**< 用户数据指针 */
        int   dlen;             /**< 用户数据长度，实现者可设置此值也可不设置 */
        const char *key;        /**< 若为哈希表的迭代器，则为哈希键值地址 */
        int   klen;             /**< 若为ACL_BINHASH迭代器，则为键长度 */
        int   i;                /**< 当前迭代器在容器中的位置索引 */
        int   size;             /**< 当前容器中元素总个数 */
};

/**
 * 正向遍历容器中元素
 * @param iter {ACL_ITER}
 * @param container {void*} 容器地址
 * @examples: samples/iterator/
 */
#define ACL_FOREACH(iter, container)  \
    if ((container))  \
        for ((container)->iter_head(&(iter), (container));  \
             (iter).ptr;  \
             (container)->iter_next(&(iter), (container)))

/**
 * 反向遍历容器中元素
 * @param iter {ACL_ITER}
 * @param container {void*} 容器地址
 * @examples: samples/iterator/
 */
#define ACL_FOREACH_REVERSE(iter, container)  \
    if ((container))  \
        for ((container)->iter_tail(&(iter), (container));  \
             (iter).ptr;  \
             (container)->iter_prev(&(iter), (container)))

/**
 * 获得当前迭代指针与某容器关联的成员结构类型对象
 * @param iter {ACL_ITER}
 * @param container {void*} 容器地址
 */
```

```
#define ACL_ITER_INFO(iter, container)  \
        ((container) ? (container)->iter_info(&(iter), (container)) : NULL)

#define acl_foreach_reverse     ACL_FOREACH_REVERSE
#define acl_foreach             ACL_FOREACH
#define acl_iter_info           ACL_ITER_INFO

#endif
```

其实，ACL_ITER 只是定义了一些规则，具体实现由各个容器自己来实现，如果容器要实现正向遍历，则需要遵守如下原则：

1）则容器的结构中必须要有成员变量：iter_head(ACL_ITER* iter, /* 容器本身的对象指针 */), iter_next(ACL_ITER* iter, /* 容器本身的对象指针 */); 如果没有这两个成员变量怎么办？那在编译时如果有函数使用该容器的 acl_foreach(){} 则编译器会报错，这样的好处是尽量让错误发生在编译阶段。

2）同时在容器内部需要实现两个注册函数: iter_head()/2, iter_next()/2, 此两函数内部需要将容器的数据赋值给 iter->data；同时改变容器中下一个对象的位置并赋值给 iter->ptr；如果容器本身是由整数值来标识元素索引位置的，则可以把索引位置赋值给 iter->i，但别忘记依然需要将 iter->ptr 赋值 - - 可以赋与iter->data 同样的值，这样可以避免acl_foreach() 提前退出。

至于反向遍历容器中元素，规则约束下正向遍历类似，在此不再详述。

下面，以一个大家常用的字符串分隔功能的函数例子来结束本文：

```
void argv_iter(void)
{
        const char *s = "hello world, you are welcome!";  /* 源串 */
        ACL_ARGV *argv = acl_argv_split(s, " ,!"); /* 对源串进行分隔 */
        ACL_ITER iter;  /* 通用的迭代器 */

        printf("\nacl_foreach for ACL_ARGV:\n");
        /* 正向遍历字符串数组 argv 中的所有元素 */
        acl_foreach(iter, argv) {
                printf(">> i: %d, value: %s\n", iter.i, (char*) iter.data);
```

```
        }

        printf("\nacl_foreach_reverse for ACL_ARGV:\n");
        /* 反向遍历字符串数组 argv 中的所有元素 */
        acl_foreach_reverse(iter, argv) {
                printf(">> i: %d, value: %s\n", iter.i, (char*) iter.data);
        }
        /* 释放字符串数组 argv */
        acl_argv_free(argv);
}
```

ACL中有哪些常见的容器实现了 ACL_ITER 所要求的功能，可以通过 samples/iterator/ 下的例子进行查看.

ACL 库下载位置：http://acl.sourceforge.net/

4.2 先进先出队列

发表时间: 2009-11-03 关键字: .net

ACL库中有个模块实现了先进先出队列的功能，其使用方法非常简单，下面是结构定义：

```
struct ACL_FIFO {
        ACL_FIFO_INFO *head;
        ACL_FIFO_INFO *tail;
        int   cnt;

        /* for acl_iterator */

        /* 取迭代器头函数 */
        const void *(*iter_head)(ACL_ITER*, struct ACL_FIFO*);
        /* 取迭代器下一个函数 */
        const void *(*iter_next)(ACL_ITER*, struct ACL_FIFO*);
        /* 取迭代器尾函数 */
        const void *(*iter_tail)(ACL_ITER*, struct ACL_FIFO*);
        /* 取迭代器上一个函数 */
        const void *(*iter_prev)(ACL_ITER*, struct ACL_FIFO*);
        /* 取迭代器关联的当前容器成员结构对象 */
        const ACL_FIFO_INFO *(*iter_info)(ACL_ITER*, struct ACL_FIFO*);
};
```

有两个初始化函数，一个是静态初始化，一个是动态初始化（需要释放），如下：

```
/**
 * 初始化一个给定队列，应用可以在栈上分配队列，而后调用该函数进行初始化
 * @param fifo {ACL_FIFO *}
 * @example:
 *   void test(void) {
        ACL_FIFO fifo;

        acl_fifo_init(&fifo);
 *   }
 */
ACL_API void acl_fifo_init(ACL_FIFO *fifo);
```

```
/**
 * 从内存堆中分配一个队列对象，需要用 acl_fifo_free 释放
 * @return {ACL_FIFO*}
 */
ACL_API ACL_FIFO *acl_fifo_new(void);

/**
 * 释放以堆分配的队列对象
 * @param fifo {ACL_FIFO*}
 * @param free_fn {void (*)(void*)}, 如果该函数指针不为空则
 *  用来释放队列中动态分配的对象
 */
ACL_API void acl_fifo_free(ACL_FIFO *fifo, void (*free_fn)(void *));
```

向队列中添加及获得对象的函数如下：

```
/**
 * 向队列中添加一个动态堆对象
 * @param fifo {ACL_FIFO*}
 * @param data {void*} 动态对象
 */
ACL_API void acl_fifo_push(ACL_FIFO *fifo, void *data);

/**
 * 从队列中以先进先出方式弹出一个动态对象，同时将该对象从队列中删除
 * @param fifo {ACL_FIFO*}
 * @return {void*}, 如果为空，则表示队列为空
 */
ACL_API void *acl_fifo_pop(ACL_FIFO *fifo);
```

其它的辅助函数：

```
/**
 * 返回队列中头部(即最新添加的)的动态对象
 * @param fifo {ACL_FIFO*}
```

```
 * @return {void*}, 如果为空，则表示队列为空
 */
ACL_API void *acl_fifo_head(ACL_FIFO *fifo);

/**
 * 返回队列中尾部(即最早添加的)的动态对象
 * @param fifo {ACL_FIFO*}
 * @return {void*}, 如果为空，则表示队列为空
 */
ACL_API void *acl_fifo_tail(ACL_FIFO *fifo);

/**
 * 返回队列中动态对象的总个数
 * @param fifo {ACL_FIFO*}
 * @return {int}, >= 0
 */
ACL_API int acl_fifo_size(ACL_FIFO *fifo);
```

下面是一个简单的例子：

```
#include "lib_acl.h"
#include <stdio.h>
#include <stdlib.h>

static void fifo_test(void)
{
        ACL_FIFO *fifo;
        int   i;
        char *ptr;
        ACL_ITER iter;

        fifo = acl_fifo_new();  // 创建队列

        for (i = 0; i < 20; i++) {
                ptr = (char*) acl_mymalloc(100);
```

```
                snprintf(ptr, 100, "test:%d", i);
                acl_fifo_push(fifo, ptr);  // 向队列中添加动态元素
        }

        // 遍历队列中的所有元素
        acl_foreach(iter, fifo) {
                const char *ptr = (const char*) iter.data;
                printf(">>>%s\n", ptr);
        }

        while (1) {
                ptr = (char*) acl_fifo_pop(fifo);  // 从队列中取得动态元素
                if (ptr == NULL)
                        break;
                printf("fifo pop: %s\n", ptr);
        }

        acl_fifo_free(fifo, acl_myfree_fn);  // 释放队列
}

int main(int argc acl_unused, char *argv[] acl_unused)
{
        fifo_test();
        getchar();
        return (0);
}
```

这个例子非常简单，只是演示了如何使用ACL库中的先进先出队列。头文件参看：lib_acl/include/stdlib/acl_fifo.h

ACL 下载：http://acl.sourceforge.net/

5.1 利用ACL开发并发网络服务器

发表时间: 2009-06-07 关键字: 网络应用, 应用服务器, FreeBSD, 编程, Socket

1、概述

本节结合 "利用ACL库开发高并发半驻留式线程池程序" 和 "利用ACL库快速创建你的网络程序" 两篇文章的内容，创建一个简单的线程池网络服务器程序。

2、并发式网络通信实例

```
#include "lib_acl.h"  /* 先包含ACL库头文件 */
#include <stdio.h>
#include <stdlib.h>

/**
 * 单独的线程处理来自于客户端的连接
 * @param arg {void*} 添加任务时的对象
 */
static void echo_client_thread(void *arg)
{
        ACL_VSTREAM *client = (ACL_VSTREAM*) arg;
        char  buf[1024];
        int   n;

        /* 设置客户端流的读超时时间为30秒 */
        ACL_VSTREAM_SET_RWTIMO(client, 30);

        /* 循环读客户端的数据，直到其关闭或出错或超时 */
        while (1) {
                /* 等待读客户端发来的数据 */
                n = acl_vstream_read(client, buf, sizeof(buf));
                if (n == ACL_VSTREAM_EOF)
                        break;
                /* 将读到的数据写回至客户端流 */
                if (acl_vstream_writen(client, buf, n) == ACL_VSTREAM_EOF)
                        break;
        }
```

```
        /* 关闭客户端流 */
        acl_vstream_close(client);
}

/**
 * 创建半驻留线程池的过程
 * @return {acl_pthread_pool_t*} 新创建的线程池句柄
 */
static acl_pthread_pool_t *create_thread_pool(void)
{
        acl_pthread_pool_t *thr_pool;  /* 线程池句柄 */
        int  max_threads = 100;  /* 最多并发100个线程 */
        int  idle_timeout = 10;  /* 每个工作线程空闲10秒后自动退出 */
        acl_pthread_pool_attr_t attr;  /* 线程池初始化时的属性 */

        /* 初始化线程池对象属性 */
        acl_pthread_pool_attr_init(&attr);
        acl_pthread_pool_attr_set_threads_limit(&attr, max_threads);
        acl_pthread_pool_attr_set_idle_timeout(&attr, idle_timeout);

        /* 创建半驻留线程句柄 */
        thr_pool = acl_pthread_pool_create(&attr);
        assert(thr_pool);
        return (thr_pool);
}

/**
 * 开始运行
 * @param addr {const char*} 服务器监听地址，如：127.0.0.1:8081
 */
static void run(const char *addr)
{
        const char *myname = "run";
        acl_pthread_pool_t *thr_pool;
        ACL_VSTREAM *sstream;
        char  ebuf[256];
```

```
        thr_pool = create_thread_pool();

        /* 监听一个本地地址 */
        sstream = acl_vstream_listen(addr, 128);
        if (sstream == NULL) {
                printf("%s(%d): listen on %s error(%s)\r\n",
                                myname, __LINE__, addr,
                                acl_last_strerror(ebuf, sizeof(ebuf)));
                return;
        }

        printf("%s: listen %s ok\r\n", myname, addr);
        while (1) {
                /* 等待接受客户端的连接 */
                client = acl_vstream_accept(sstream, NULL, 0);
                if (client == NULL) {
                        printf("%s(%d): accept error(%s)\r\n",
                                        myname, __LINE__,
                                        acl_last_strerror(ebuf, sizeof(ebuf)));
                        break;
                }
                printf("accept one\r\n");
                /* 获得一个客户端连接流 */
                /* 开始处理该客户端连接流 */

                /**
                 * 向线程池中添加一个任务
                 * @param thr_pool 线程池句柄
                 * @param echo_client_thread 工作线程的回调函数
                 * @param client 客户端数据流
                 */
                acl_pthread_pool_add(thr_pool, echo_client_thread, client);
        }

        /* 销毁线程池对象 */
        acl_pthread_pool_destroy(thr_pool);
}
```

```
/**
 * 初始化过程
 */
static void init(void)
{
        /* 初始化ACL库 */
        acl_init();
}

/**
 * 使用提示接口
 * @param procname {cosnt char*} 程序名
 */
static void usage(const char *procname)
{
        printf("usage: %s listen_addr\r\n", procname);
        printf("example: %s 127.0.0.1:8081\r\n", procname);
        getchr();
}

int main(int argc, char *argv[])
{
        if (argc != 2) {
                usage(argv[0]);
                return (0);
        }

        init();
        run(argv[1]);
        return (0);
}
```

由上可以看出，创建一个并发式服务器程序也是如此的简单。 该例子可以同时运行在WIN32平台及UNIX平

台（Linux, FreeBSD, Solaris-x86).

3、小结

由以上例子可以看出，ACL库屏蔽底层SOCKET的细节操作，使网络编程变得简单，使使用者可以专心于其应用，而不是拘泥于SOCKET操作上，另外结合半驻留线程池的ACL库就可以开发高效的并发网络应用来。

当然，以上例子也存在一个缺点，那就是当客户端并发连接比较高时，因为一个连接占用一个线程，所以高并发时就需要更多的线程（为了启动更多的线程，可以通过 acl_pthread_pool_set_stacksize 或 acl_pthread_pool_attr_set_stacksize 设置每个线程的堆栈为较小的值，如 500KB）；而采用ACL库里的另一种编程技术 - - 非阻塞式IO，可以使一个线程同时处理多个并发TCP连接，同时可以启动多个这样的非阻塞线程，从而可以更好地利用多核（一般是一个核可以启用一个非阻塞IO线程），将来，我们将会对此类问题进行讨论，并给出具体实例。

acl 库的下载地址：http://acl.sourceforge.net/

acl 库的在线帮助地址：http://acl.sourceforge.net/acl_help/index.html

4 参考

1) 利用ACL库快速创建你的网络程序--ACL_VSTREAM 流的使用

2) 利用ACL库开发高并发半驻留式线程池程序

5.2 利用ACL库快速创建你的网络程序--ACL_VSTREAM 流的使用

发表时间: 2009-06-07 关键字: 网络应用, Socket, 应用服务器, Unix, 编程

1、概述

操作系统在API层为我们提供了进行网络通讯的库(一组socket函数库)，但使用起来未免复杂，而且极易出错，虽然这些socket库最初起源于 BSD系统，各个操作系统厂商都提供了自身平台的接口实现，但这些接口在不同OS上又略有差别，所以当你想写一个跨平台的网络通信程序时，工作量还是有的，并且如不知晓各个平台下的差异也极易出错。

本节向你介绍了怎样使用ACL库中的数据流(ACL_VSTREAM)来快速搭建你的网络通信程序；另外，ACL_VSTREAM 不仅是跨平台的，而且既可用于网络通信流，又可用于文件流，本节仅介绍网络流的例子。

2、网络通信函数接口说明

2.1）服务端接口

```
/**
 * 监听某个地址（对于UNIX，还可以监听域套接字）
 * @param addr {const char*} 监听地址
 *  如：127.0.0.1:80，或域套接字，如：/tmp/test.sock
 * @param qlen {int} 监听队列的长度
 * @return {ACL_VSTREAM*} 监听流指针
 */
ACL_API ACL_VSTREAM *acl_vstream_listen(const char *addr, int qlen);

/**
 * 从监听流中接收一个客户端连接流
 * @param listen_stream {ACL_VSTREAM*} 监听流
 * @param ipbuf {char*} 如果不为空则用来存储客户端的IP地址
 * @param bsize {int} 如果 ipbuf 不为空，则表示 ipbuf 的空间大小
 * @return {ACL_VSTREAM*} 如果不为空则表示新接收的客户端流
 */
ACL_API ACL_VSTREAM *acl_vstream_accept(ACL_VSTREAM *listen_stream,
                    char *ipbuf, int bsize);
```

2.2、客户端接口

```
/**
 * 远程连接服务器
 * @param addr {const char*} 服务器地址，格式如：127.0.0.1，
 *  或 域套接地址：/tmp/test.sock
 * @param block_mode {int} 阻塞连接还是非阻塞连接，ACL_BLOCKING, ACL_NON_BLOCKING
 * @param connect_timeout {int} 连接超时时间(秒)
 * @param rw_timeout {int} 连接流成功后的读写超时时间，单位为秒
 * @param rw_bufsize {int} 连接流成功后的缓冲区大小
 * @return {ACL_VSTREAM*} 如果不为空，则表示连接成功后的数据流
 */
ACL_API ACL_VSTREAM *acl_vstream_connect(const char *addr, int block_mode,
                        int connect_timeout, int rw_timeout, int rw_bufsize);
```

2.3、读写过程接口

```
/**
 * 从数据流中一次性读取 n 个数据，该 n 有可能会小于用户所需要的 maxlen
 * @param stream {ACL_VSTREAM*} 数据流
 * @param vptr {void*} 用户的数据缓冲区指针地址
 * @param maxlen {size_t} vptr 数据缓冲区的空间大小
 * @return ret {int}, ret == ACL_VSTREAM_EOF: 表示出错，应该关闭本地数据流，
 *  ret > 0:  表示读到了 ret 个字节的数据
 *  注: 如果缓冲区内有数据，则直接把缓冲区内的数据复制到用户的缓冲区然后直接返回;
 *      如果缓冲区内无数据，则需要调用系统读操作(有可能会阻塞在系统读操作上)，该
 *      次调用返回后则把读到数据复制到用户缓冲区返回.
 *      在这两种情况下都不能保证读到的字节数等于所要求的字节数，若想读到所要求的
 *      字节后才返回则请调用 vstream_loop_readn() 函数.
 */
ACL_API int acl_vstream_read(ACL_VSTREAM *stream, void *vptr, size_t maxlen);

/**
 * 从数据流中读取一行数据，直到读到  "\n" 或读结束为止，正常情况下包括 "\n"
 * @param stream {ACL_VSTREAM*} 数据流
 * @param vptr {void*} 用户所给的内存缓冲区指针
```

```
 * @param maxlen {size_t} vptr 缓冲区的大小
 * @return  ret {int}, ret == ACL_VSTREAM_EOF:  读出错或对方关闭了连接,
 *  应该关闭本地数据流; n > 0:  读到 了 n 个字节的数据, 如果该 n 个数据
 *  的最后一个非 0 字符为 "\n" 表明读到了一个完整的行, 否则表明读到了 n
 *  个数据但对方未发送 "\n" 就关闭了连接; 还可以通过检查
 *  (stream->flag & ACL_VSTREAM_FLAG_TAGYES)
 *  不等于 0 来判断是否读到了 "\n", 如果非 0 则表示读到了 "\n".
 */
ACL_API int acl_vstream_gets(ACL_VSTREAM *stream, void *vptr, size_t maxlen);

/**
 * 循环向数据流中写 dlen 个字节的数据直至写完或出错为止
 * @param stream {ACL_VSTREAM*} 数据流
 * @param vptr {const char*} 数据区指针地址
 * @param dlen {size_t} 待写的数据区数据长度
 * @return ret {int}, ret == ACL_VSTREAM_EOF: 表示写出错, 应该关闭本地数据流,
 *  ret > 0:  表示成功写了 dlen 个字节的数据
 */
ACL_API int acl_vstream_writen(ACL_VSTREAM *stream, const void *vptr, size_t dlen);

/**
 * 带格式的流输出, 类似于 fprintf()
 * @param stream {ACL_VSTREAM*} 数据流
 * @param fmt {const char*} 数据格式
 * @return ret {int}, ret == ACL_VSTREAM_EOF: 表示写出错, 应该关闭本地数据流,
 *  ret > 0:  表示成功写了 dlen 个字节的数据
 */
ACL_API int acl_vstream_fprintf(ACL_VSTREAM *stream, const char *fmt, ...);
```

2.3、流关闭接口

```
/**
 * 释放一个数据流的内存空间并关闭其所携带的 socket 描述符
 * @param stream {ACL_VSTREAM*} 数据流
```

```
 */
ACL_API int acl_vstream_close(ACL_VSTREAM *stream);
```

3、网络通信实例

3.1 一个简单的服务器程序

```
#include "lib_acl.h"  /* 先包含ACL库头文件 */
#include <stdio.h>
#include <stdlib.h>

static void echo_client(ACL_VSTREAM *client)
{
    char  buf[1024];
    int   n;

    /* 设置客户端流的读超时时间为30秒 */
    ACL_VSTREAM_SET_RWTIMO(client, 30);

    /* 循环读客户端的数据，直到其关闭或出错或超时 */
    while (1) {
        /* 等待读客户端发来的数据 */
        n = acl_vstream_read(client, buf, sizeof(buf));
        if (n == ACL_VSTREAM_EOF)
            break;
        /* 将读到的数据写回至客户端流 */
        if (acl_vstream_writen(client, buf, n) == ACL_VSTREAM_EOF)
            break;
    }

    /* 关闭客户端流 */
    acl_vstream_close(client);
}

static void run(const char *addr)
```

```
{
    const char *myname = "run";
    ACL_VSTREAM *sstream;
    char  ebuf[256];

    /* 监听一个本地地址 */
    sstream = acl_vstream_listen(addr, 128);
    if (sstream == NULL) {
        printf("%s(%d): listen on %s error(%s)\r\n",
            myname, __LINE__, addr,
            acl_last_strerror(ebuf, sizeof(ebuf)));
        return;
    }

    printf("%s: listen %s ok\r\n", myname, addr);
    while (1) {
        /* 等待接受客户端的连接 */
        client = acl_vstream_accept(sstream, NULL, 0);
        if (client == NULL) {
            printf("%s(%d): accept error(%s)\r\n",
                myname, __LINE__,
                acl_last_strerror(ebuf, sizeof(ebuf)));
            return;
        }
        printf("accept one\r\n");
        /* 获得一个客户端连接流 */
        /* 开始处理该客户端连接流 */
        echo_client(client);
    }
}

static void init(void)
{
    acl_init();  /* 初始化ACL库 */
}

static void usage(const char *procname)
```

```
{
    printf("usage: %s listen_addr\r\n", procname);
    printf("example: %s 127.0.0.1:8081\r\n", procname);
}

int main(int argc, char *argv[])
{
     if (argc != 2) {
         usage(argv[0]);
         return (0);
     }

     init();
     run(argv[1]);
     return (0);
}
```

由上可以看出，创建一个服务器程序是多么的简单，当然，这是一个阻塞式线程的服务器程序，如果你要想提高并发度，可以在线程里处理来自客户端的连接。

3.2、一个简单的客户端程序

```
#include "lib_acl.h"
#include <stdio.h>
#include <stdlib.h>

static void run(const char *addr)
{
    const char *myname = "run";
    ACL_VSTREAM *client;
    char  ebuf[256], buf[1024];
    int   n, cnt = 0;

    /* 连接远程服务器，采用阻塞模式连接，连接超时为10秒，
```

```
     * 流的读超时时间为20秒，流的缓冲区大小为1024字节
     */
    client = acl_vstream_connect(addr, ACL_BLOCKING, 10, 20, 1024);
    if (client == NULL) {
        printf("%s(%d): connect addr %s error(%s)\r\n",
            myname, __LINE__, addr,
            acl_last_strerror(ebuf, sizeof(ebuf)));
        return;
    }

    printf("%s: connect %s ok\r\n", myname, addr);
    while (1) {
        /* 向服务器发送一行数据 */
        n = acl_vstream_fprintf(client, ">>hi, I'm coming in...(%d)\r\n", ++cnt);
        if (n == ACL_VSTREAM_EOF)
            break;

        /* 从服务器读取一行数据 */
        n = acl_vstream_gets(client, buf, sizeof(buf));
        if (n == ACL_VSTREAM_EOF)
            break;

        /* 最多循环5次 */
        if (cnt >= 5)
            break;

        /* 休息一下 */
        sleep(1);
    }
    /* 关闭流 */
    acl_vstream_close(client);
}

static void init(void)
{
    acl_init();  /* 初始化ACL库 */
}
```

```
static void usage(const char *procname)
{
    printf("usage: %s server_addr\r\n", procname);
    printf("example: %s 127.0.0.1:8081\r\n", procname);
}

int main(int argc, char *argv[])
{
    if (argc != 2) {
        usage(argv[0]);
        return (0);
    }

    init();
    run(argv[1]);
    return (0);
}
```

嗯，看来创建网络客户端程序原来也这么简单。

4、小结

由以上例子可以看出，ACL库屏蔽底层SOCKET的细节操作，使网络编程变得简单，使使用者可以专心于其应用，而不是拘泥于SOCKET操作上。

当然，若要完全编译通过，别忘了还需要包含ACL库的二进制版本 lib_acl.a (Unix) 或 lib_acl_vc2003.lib(Windows)。

acl 库的下载地址：http://acl.sourceforge.net/

5.3 使用ACL库编写DNS查询应用

发表时间: 2009-09-26 关键字: 网络应用, 应用服务器, 网络协议, 设计模式, 数据结构

在用C/C++写程序时，若要根据域名查找其对应的IP地址，大家一般会用到 gethostbyname 的标准函数，如若要查询 www.sina.com.cn 域名调用 gethostbyname 时，该函数首先会查找本机 hosts 文件里的条目，若该配置文件里没有对应域名，则该函数会向本机配置的DNS服务器发送查询请求，然后将DNS服务器的返回结果反给用户。该函数的使用较为简单，但却有一个限制，如果针对某个域名，在内外网有不同的IP地址时就不好办了，假设该域名由内网DNS（192.168.1.33）解析时的IP地址为 192.168.1.22, 由外网DNS(211.239.1.33)解析因为gethostbyname使用系统统一的 resolve.conf里的DNS服务器配置地址。要解决这个问题，最好的方式就是直接连接DNS服务器以遵循DNS协议格式发送查询指令，ACL库里便实现DNS协议的通信及协议解析功能（在 lib_acl/src/net/dns/ 目录下），下面介绍如何使用ACL的DNS库进行域名查询。

首先列出本例子用到的函数接口说明：

```
/**
 * 创建一个DNS查询对象
 * @param dns_ip {const char*} DNS的IP地址
 * @param dns_port {unsigned short} DNS的Port
 * @return {ACL_RES*} 新创建的查询对象
 */
ACL_API ACL_RES *acl_res_new(const char *dns_ip, unsigned short dns_port);

/**
 * 释放一个DNS查询对象
 * @param res {ACL_RES*} DNS查询对象
 */
ACL_API void acl_res_free(ACL_RES *res);

/**
 * 查询某个域名的IP地址
 * @param res {ACL_RES*} DNS查询对象
 * @param domain {const char*} 要查询的域名
 * @return {ACL_DNS_DB*} 查询的结果集
 */
ACL_API ACL_DNS_DB *acl_res_lookup(ACL_RES *res, const char *domain);

/**
```

```
 * 根据错误号获得查询失败的原因
 * @param errnum {int} 错误号
 * @return {const char*} 错误信息
 */
ACL_API const char *acl_res_strerror(int errnum);

/**
 * 获得当前查询的错误信息
 * @param res {ACL_RES*} DNS查询对象
 * @return {const char*} 错误信息
 */
ACL_API const char *acl_res_errmsg(const ACL_RES *res);
```

然后给出具体例子如下：

```
#include "lib_acl.h"
#include <stdio.h>

static int dns_lookup(const char *domain, const char *dns_ip,
        unsigned short dns_port)
{
        ACL_RES *res = NULL;  /* DNS 查询对象 */
        ACL_DNS_DB *dns_db = NULL;  /* 存储查询结果 */
        ACL_ITER iter;  /* 通用迭代对象 */

#define RETURN(_x_) do { \
        if (res) \
                acl_res_free(res); \
        if (dns_db) \
                acl_netdb_free(dns_db); \
        return (_x_); \
} while (0)

        /* 创建一个DNS查询对象 */
        res = acl_res_new(dns_ip, dns_port);
```

```
        /* 向DNS服务器发送查询指令并接收处理结果 */
        dns_db = acl_res_lookup(res, domain);
        if (dns_db == NULL) {
                printf("failed for domain %s, %s", domain, acl_res_errmsg(res));
                RETURN (-1);
        }

        /* 遍历查询结构并输出至标准输出 */
        printf("type\tttl\tip\t\tnet\t\tqid\t\n");
        acl_foreach(iter, dns_db) {
                ACL_HOST_INFO *info;
                struct in_addr in;
                char  buf[32];

                info = (ACL_HOST_INFO*) iter.data;
                in.s_addr = info->saddr.sin_addr.s_addr;

                /* 假设网络掩码为24位，获得网络地址 */
                acl_mask_addr((unsigned char*) &in.s_addr, sizeof(in.s_addr), 24);

                /* 将地址转换成字符串 */
                acl_inet_ntoa(in, buf, sizeof(buf));

                /* 输出查询结果 */
                printf("A\t%d\t%s\t%s\t%d\r\n",
                        info->ttl, info->ip, buf, res->cur_qid);
        }

        RETURN (0);
}

int main(int argc acl_unused, char *argv[] acl_unused)
{
        const char *dns_in_ip = "192.168.1.33", *dns_out_ip = "211.239.1.33";
        unsigned short dns_in_port = 53, dns_out_port = 53;
        const char *domain = "www.test.com.cn";
```

```
        /* 查询内网DNS */
        (void) dns_lookup(domain, dns_in_ip, dns_in_port);

        /* 查询外网DNS */
        (void) dns_lookup(domain, dns_out_ip, dns_out_port);

        return (0);
}
```

OK，这个例子很简单，完全满足刚才所说的需求。此外，该例子用到了几个结构类型，如下（也可以直接查询 lib_acl/include/net/ 目录下的头文件说明）：

```
/* DNS查询对象结构定义 */
typedef struct ACL_RES {
        char dns_ip[64];                /**< DNS的IP地址 */
        unsigned short dns_port;        /**< DNS的Port */
        unsigned short cur_qid;         /**< 内部变量，数据包的标识 */
        time_t tm_spent;                /**< 查询时间耗费(秒) */
        int   errnum;
#define ACL_RES_ERR_SEND        -100    /**< 写出错 */
#define ACL_RES_ERR_READ        -101    /**< 读出错 */
#define ACL_RES_ERR_RTMO        -102    /**< 读超时 */
#define ACL_RES_ERR_NULL        -103    /**< 空结果 */
#define ACL_RES_ERR_CONN        -104    /**< TCP方式时连接失败 */

        int transfer;                   /**< TCP/UDP 传输模式 */
#define ACL_RES_USE_UDP         0       /**< UDP 传输模式 */
#define ACL_RES_USE_TCP         1       /**< TCP 传输模式 */

        int   conn_timeout;             /**< TCP 传输时的连接超时时间，默认为10秒 */
        int   rw_timeout;               /**< TCP/UDP 传输的IO超时时间，默认为10秒 */
} ACL_RES;

/**
```

```
 * 主机地址结构
 */
typedef struct ACL_HOSTNAME ACL_HOST_INFO;
typedef struct ACL_HOSTNAME {
        char  ip[64];                   /**< the ip addr of the HOST */
        struct sockaddr_in saddr;       /**< ip addr in sockaddr_in */
        unsigned int ttl;               /**< the HOST's ip timeout(second) */
        int   hport;
        unsigned int nrefer;            /**< refer number to this HOST */
} ACL_HOSTNAME;

/**
 * DNS查询结果集
 */
typedef struct ACL_DNS_DB {
        ACL_ARRAY *h_db;
        int   size;
        char  name[256];

        /* for acl_iterator */

        /* 取迭代器头函数 */
        const ACL_HOST_INFO *(*iter_head)(ACL_ITER*, struct ACL_DNS_DB*);
        /* 取迭代器下一个函数 */
        const ACL_HOST_INFO *(*iter_next)(ACL_ITER*, struct ACL_DNS_DB*);
        /* 取迭代器尾函数 */
        const ACL_HOST_INFO *(*iter_tail)(ACL_ITER*, struct ACL_DNS_DB*);
        /* 取迭代器上一个函数 */
        const ACL_HOST_INFO *(*iter_prev)(ACL_ITER*, struct ACL_DNS_DB*);
        /* 取迭代器关联的当前容器成员结构对象 */
        const ACL_HOST_INFO *(*iter_info)(ACL_ITER*, struct ACL_DNS_DB*);
} ACL_DNS_DB;
```

至于文所提到的迭代器遍历过程，请参考文章：C语言中迭代器的设计与使用

6.1 基于POSTFIX的服务器框架的服务器程序设计

发表时间: 2009-06-07 关键字: 应用服务器, 框架, 设计模式, 工作, 网络应用

一、 概述

在当今网络应用中，各种开源服务器可谓遍地开花。Web服务器如 Apache、AOL - Server、Lighttpd等；数据库服务器如：Mysql、PostgreSQL；MTA服务如Postfix、 Sendmail、Qmail等；HTTP代理服务器有Squid、Oops等。每一类服务器的设计都比较复杂且相关性较强，它们所用的服务器框架通用性不够，很难提炼出来，形成相对独立的服务器框架，供程序员快速开发自己的项目。在我们习惯了JAVA、.NET、PHP等快速开发带给我们的快乐时，却逐渐地远离的程序设计的本质，结果是知其然面不知所以然。好的开发框架及开发工具固然大大提高了软件生产率，但却容易使程序员浮于表面，无法深入，当需要设计开发更加高效、安全的服务器程序时，令很多人一愁莫展。

本文讲述了以Postfix（http://www.postfix.org/）的服务器框架为基础的半驻留进程池服务器的设计模式，经过本人整理，将其纳入ACL库中，做为ACL基础库中进程池服务器框架部分的核心。如果你非常熟悉 Postfix的框架设计，那么本节的内容对于你来说非常容易接受，当然，如果你未曾接触过postfix，也没关系，通过本文，你可以非常方便快捷地创建基于高级服务器框架之上的应用服务器程序：简单、高效、安全、可扩展性好等。

二、ACL进程池服务器框架概述

开发CGI程序、用过 inetd/xinetd的技术人大概应知道，自己所写的程序少有网络通信过程，基本上所有的IO操作都是基于标准IO(stdin, stdout, stderr）进行的，一切来得是那样简单，因为后台的服务器框架程序（如开发CGI时用的Apache）帮我们处理了复杂的网络通信过程。ACL的进程池服务器框架进行开发也是如此，但是提供了更大的灵活性，同时更加高效。ACL进程池服务器框架有如下特点：

a、用户直接与网络连接进行网络通信操作，但操作的接口是经过ACL库封装的更加灵活、丰富的函数接口；

b、进程池是动态可调的，只需在配置文件中设置进程池中的最大进程数，则ACL进程框架便自动根据应用负载情况调整工作进程数量；

c、进程池中的工作进程是半驻留的（这也是Postfix进程池的一大特色），当网络负载较高时，进程池中的每个工作进程可能都在工作，负载逐渐下降时，进程池中的工作进程并不立即退出，而是要空闲一段时间至超时，在超时之前又有新的会话连接到达时，其中的一个空闲进程便被服务器框架唤醒处理新连接，如果某些工作进程在超时时依然未接到新的处理任务则会自动退出。这便是半驻留进程池的特点：并不一直驻留内存等待新任务。这种设计的好处是：任务负载高时，进程池中的每个工作进程在处理完上一个任务可以立即处理新的任务，这样大大降低了创建新进程的开销；当任务负载较低时，工作进程在空闲一定时间后便自动退出，从而降低了资源浪费，同时也尽量避免了开发人员的应用程序本身的一些内存泄漏隐患（当然，开发人员应尽量避免内存泄漏问题）；

d、半驻留的进程池有多种控制条件使工作进程处于半驻留状态：空闲时间、任务处理最大上限等；

ACL进程池服务器框架有一个后台守护进程（acl_master），该进程处于长驻留状态，监听着其所管理着的工作进程的对外服务的所有端口。当有新的客户端连接到达时，acl_master并不具体处理该连接，而是创建

一个新的工作进程或“让”处于空闲状态的工作进程接管该连接并进行处理。acl_master与半驻留进程池中的工作进程之间是管理与被管理的关系，同时又是一种协作关系，因为acl_master并不象真正的操作系统那样可以任意调度哪个工作任务（大多通过中断进行任务时间片的分配），acl_master没有那种权限功能；在进程池中没有工作进程或进程池中的所有进程都处于忙碌状态时，如果整个进程池中工作进程的数量未超过配置中规定的最大值，acl_master便会创建新的工作进程来处理新到任务；当进程池中有空闲工作进程而又有新连接到达时，则acl_master知道有空闲进程存在，所以不会插手任务处理过程，此时，进程池中的空间工作进程阻塞在某一个 accept() 调用（或某一个锁）上，这时由操作系统具体将该新到连接分配给某一个工作进程，该工作进程便通知acl_master自己目前已处于忙状态，于是 acl_master便在其所维护的工作进程池的表中标记该工作进程的状态为忙状态。

小结，ACL进程池服务器框架应由两部分组成：acl_master后台守护管理进程与半驻留式工作进程池。acl_master负责任务分配及工作进程池的管理与维护；工作进程池中的工作进程负责具体的客户端连接请示与任务处理。

三、ACL进程池服务器所支持的工作进程框架模板

因为ACL的进程池服务器框架模型是基于Postfix进行修改的，所以首先大家需要知道原Postfix的工作进程框架模板有哪些被ACL所采纳。

a、single模板：该工作进程框架为开发者提供了开发服务器程序的函数接口，它的特点是每个工作进程在同一时刻最多仅能处理一个客户端连接；

b、multi模板：它的特点是每个工作进程可以同时处理多个并发连接，当然开发者在用此接口时需要在网络IO操作中采取非阻塞的方式，以免造成某工作进程在阻塞地读一个连接的数据时，而延迟了对其它网络连接的处理过程，从而造成性能上的极大降低；

c、trigger模板：它的特点是定时触发一个工作任务，可以把它与UNIX下的CRON功能相类比。

除以工作进程框架模板外，另外还增加了其它两个与线程池相结合的工作进程框架模板，如下：

a、listener模板：该模板允许与线程池相结合，从而使一个工作进程可以同时处理多个连接，而每个连接则由该进程内线程池的某个线程进行处理(目前已不继续开发了)；

b、ioctl模板：该模板是listener模板的进一步改良，使工作进程与工作线程池结合的更加紧密，直接在工作进程的模板内嵌入了工作线程池句柄。

d、aio模板，将异步IO处理过程嵌入至工作进程模板中，从而更加高效地处理大并发网络连接的情况，其事件触发机制采用类型于squid, lighttpd, ircd 等类似的非阻塞服务器的模式。

四、使用ACL进程池服务器框架

有时为了让他人明白自己所讲的话，演讲人总会因唯恐别人不明白而特意讲得罗罗嗦嗦，岂不知此时观众已经被给绕晕至昏昏欲睡了，哈，但愿阁下看了以上长篇累牍的描述后依然保持头脑清醒，精神抖擞。OK，下面让我们从实例入手，看看怎样用ACL进程池框架编写自己的应用服务器程序。

a、编译并发布ACL进程池服务器框架

首先，需要编译acl_master守护进程，其实很简单，只需要如下步骤即可：

cd acl_project/; make all; make install

则在 acl_project/dist/master/libexec/linux32(以32位LINUX为例) 目录下生成 acl_master 可执行程序

cd dist/master; chmod 755 setup.sh; ./setup.sh /opt/acl

则在 /opt/acl/ 目录下存在如下目录：

libexec 目录：该目录下存放了 acl_master 程序及其它与具体应用相关的工作进程程序；

conf 目录：该目录下存放了 acl_master 的配置文件 main.cf 及工作进程程序的配置文件的存放目录 service/ ；

var 目录： 该目录下有两个了目录 log (存放日志的位置) 及 pid (存放运行进程进程号的位置)，同时该 var 目录又是ACL进程池服务框架的默认运行路径；

sbin 目录：该目录下仅有两个比较简单的启动与停止脚本，它们来控制整个ACL进程池服务的启动与停止过程。

b、编写你的工作进程代码并发布，参考 快速创建你的服务器程序 - - single进程池模型；

c、修改配置文件、启动与停止整个进程池框架。

五、ACL进程池服务器框架配置说明

ACL进程池服务器框架有自己的配置文件格式及命名规范，它是你正确使用ACL进程池框架不可或缺的一部分，ACL进程池构架的开发者及维护者应熟练掌握这些格式与规范。下面就这些文件格式及命名规范一一说明。

5.1）main.cf 配置文件

该文件为 acl_master 守护进程的主配置文件，目前常用的配置项的含义如下：

daemon_directory：指定用户所写的工作进程可执行程序的存入目录，acl_master 会在该路径下搜索工作进程程序；

log_file：acl_master 的日志文件全路径名；

config_directory：工作进程程序的配置文件存放目录，acl_master 会读取该目录下所有有效的配置文件中并分析所有以 master_ 开头的配置项，注：以 master_ 开头的配置项为保留配置项的命名规范；

queue_directory：acl_master 进程及所有的工作进程的运行时所在路径；

pid_file：acl_master 进程的进程号的存储文件。

5.2）工作进程的配置文件

在 config_directory 所指的目录下，存放着各个工作进程的配置文件，acl_master 会首先读取该配置进行分析，与 acl_master 相关的配置项的含义如下：

master_disable：acl_master 是否要启动该工作进程所提供的对外服务（即是否会监听该配置文件中指定的监听端口），变量值有两个，yes 或 no。yes: 禁止该服务工作进程提供的服务；no：允许启动该服务工作进程；

master_service：服务工作进程所监听的地址及端口，格式为：[ip:]port，如：127.0.0.1:30080, :30080, 30080；

master_type：网络服务类型，目前一般为 inet 类型；

master_maxproc：服务工作进程的最大进程数；

master_command：服务工作进程的程序名，如果 master_disable 的配置内容为 no，则 acl_master 会在 main.cf 中 daemon_directory配置项指定的目录下搜索 master_command 配置项指定的服务工作进程程序名；

master_log：该服务工作进程的日志文件名；

以下的配置与工作进程各个框架模板相关的配置项，以 xxx_开头，其中，xxx 可以为：single，multi，trigger, listener，ioctl, aio。下面以 single_server 工作进程框架模板为例进行说明：

single_use_limit：服务工作进程处理事务的次数，如果为0则表明该服务工作进程长驻留内存，一直提供服务；如果大于0，则当该服务工作进程处理事务的次数达到此值后会自动退出；

single_pid_dir：服务工作进程的进程号的存放路径；

single_queue_dir：服务工作进程的运行路径；

single_rw_timeout：服务工作进程进行IO操作的超时时间，单位为秒；

single_buf_size：服务工作进程进行IO操作时所用的 ACL_VSTREAM 库函数的缓存大小；

除了以上的保留的配置项外，用户可以定义自己的配置项于该配置文件中，然后通过参数传递由服务器模板获得自己所要的参数。

acl 库的下载地址：http://www.sourceforge.net/projects/acl/

6.2 快速创建你的服务器程序 - - single进程池模型

发表时间: 2009-06-07 关键字: 框架, 配置管理, Linux, Unix, 数据结构

1、概述

本节主要描述了以进程池模式创建服务器程序的过程，而该进程池框架是以 acl_master 模板为管理进程，以 acl_single_server 单一进程池模式为半驻留进程池模板创建的。该进程池模型有如下特点：

1.1）半驻留进程池特征；

1.2）一个网络连接对应一个工作进程。

2、创建过程(以 acl_project/samples/master/single_echo 为例，ACL库是跨平台的，但 acl_master 服务器框架仅能运行在UNIX平台下)

在acl_project/samples/master/single_echo/ 目录下存放着一个以 acl_single_server 为服务器模型的echo 服务程序。在该目录下应该有 main.c, app_log.c, app_log.h, Makefile, Makefile.elib 四个文件，所需要修改的只是 main.c 文件，其它几个文件无须修改。

2.1）编写源文件

a)包含 ACL 库的头文件: #include "lib_acl.h"

b)调用服务函数 acl_single_server_main() 并注册相关函数：

函数原型：void acl_single_server_main(int argc, char **argv, ACL_SINGLE_SERVER_FN service,...);

argc, argv： 是 main() 入口的两个参数；

service ：是用户自己的服务工作函数指针，该函数是以注册函数的方式注册进服务框架模板并由服务框架调用的；

...：是一些不定参数，这些参数都是可选的，这些不定参数是以“类型：指针”的方式传递给服务框架的，由服务框架根据类型自动进行分析，常用类型有：

ACL_MASTER_SERVER_INT_TABLE - - 为int类型的配置项集合的结构数组指针（该类型与由框架读取用户所需要的配置项相关）；

ACL_MASTER_SERVER_STR_TABLE - - 字符串类型的配置项集合的结构数组指针（该类型与由框架读取用户所需要的配置项相关）；

ACL_MASTER_SERVER_BOOL_TABLE - - 布尔类型的配置项集合的结构数组指针（该类型与由框架读取用户所需要的配置项相关）；

ACL_MASTER_SERVER_PRE_INIT - - 该类型表明后面的参数为一函数指针，服务进程启动后会自动切换成普通用户身份，该类型所代表的函数会在服务切换成普通用户身份前进行调用；

ACL_MASTER_SERVER_POST_INIT - - 该类型表明后面的参数为一函数指针，当服务框架将该服务工作进程切换成普通用户身份后所调用的函数；

ACL_MASTER_SERVER_PRE_ACCEPT - - 该类型表明后面的参数为一函数指针，当服务框架监听到监听套接口上有新的客户端连接到达，在用户 accept()接受该连接之前可以先回调用户的注册函数，此函数指针即为该回调函数；

ACL_MASTER_SERVER_EXIT - - 该类型表明后面的参数为一函数指针，当该服务进程退出所回调的用户的

注册函数。

2.2）源文件展示

```
/* main.c */
#include "lib_acl.h"
#include "app_log.h"
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

static char *var_cfg_single_banner;
static int   var_cfg_single_timeout;

static ACL_CONFIG_INT_TABLE __conf_int_tab[] = {
        { "single_timeout", 60, &var_cfg_single_timeout, 0, 0 },
        { 0, 0, 0, 0, 0 },
};

static ACL_CONFIG_STR_TABLE __conf_str_tab[] = {
        { "single_banner", "hello, welcome!", &var_cfg_single_banner },
        { 0, 0, 0 },
};

static void __service(ACL_VSTREAM *stream, char *service, char **argv acl_unused)
{
        const char *myname = "__service";
        char  buf[4096];
        int   n, ret;

        /*
         * Sanity check. This service takes no command-line arguments.
         */
        if (argv[0])
                acl_msg_fatal("%s(%d)->%s: unexpected command-line argument: %s",
                                __FILE__, __LINE__, myname, argv[0]);
```

```
        acl_msg_info("%s(%d)->%s: service name = %s, rw_timeout = %d",
                        __FILE__, __LINE__, myname, service, stream->rw_timeout);

        acl_msg_info("total alloc: %d", acl_mempool_total_allocated());
        do {
                acl_watchdog_pat();

                n = acl_vstream_readline(stream, buf, sizeof(buf) - 1);
                if (n == ACL_VSTREAM_EOF) {
                        acl_msg_info("%s(%d)->%s: read over",
                                        __FILE__, __LINE__, myname);
                        break;
                }

                ret = acl_vstream_writen(stream, buf, n);
                if (ret != n) {
                        acl_msg_info("%s(%d)->%s: write error = %s",
                                        __FILE__, __LINE__, myname, strerror(errno));
                        break;
                }
        } while (0);
}

static void __pre_accept(char *name acl_unused, char **argv acl_unused)
{
}

static void __pre_jail_init(char *name acl_unused, char **argv acl_unused)
{
        acl_mempool_open(512000000, 1);
        /* 是否采用 libcore 的日志记录 */
#ifdef  HAS_LIB_CORE
# ifdef USE_LIBCORE_LOG
        app_set_libcore_log();
# endif
#endif
}
```

```
static void __post_jail_init(char *name acl_unused, char **argv acl_unused)
{
}

static void service_exit(char *service acl_unused, char **argv acl_unused)
{
#ifdef  HAS_LIB_CORE
# ifdef USE_LIBCORE_LOG
        app_libcore_log_end();
# endif
#endif
}

int main(int argc, char *argv[])
{
        acl_single_server_main(argc, argv, __service,
                        ACL_MASTER_SERVER_INT_TABLE, __conf_int_tab,
                        ACL_MASTER_SERVER_STR_TABLE, __conf_str_tab,
                        ACL_MASTER_SERVER_PRE_INIT, __pre_jail_init,
                        ACL_MASTER_SERVER_PRE_ACCEPT, __pre_accept,
                        ACL_MASTER_SERVER_POST_INIT, __post_jail_init,
                        ACL_MASTER_SERVER_EXIT, service_exit,
                        0);
        exit (0);
}
```

2.3) 配置文件：single_echo.cf

```
service single
{
# 进程是否禁止运行
master_disable = no
# 服务地址及端口号
master_service = :5003
# 服务监听为域套接口
```

```
# master_service = single_echo.sock
# 服务类型
master_type = inet
# master_type = unix
# 是否只允许私有访问, 如果为 y, 则域套接口创建在 {install_path}/var/log/private/ 目录下,
# 如果为 n, 则域套接口创建在 {install_path}/var/log/public/ 目录下,
master_private = n
master_unpriv = n
# 是否需要 chroot: n -- no, y -- yes
master_chroot = n
# 每隔多长时间触发一次，单位为秒(仅对 trigger 模式有效)
master_wakeup = -
# 最大进程数
master_maxproc = 10
# 进程程序名
master_command = single_echo
# 进程启动参数，只能为: -u [是否允许以某普通用户的身份运行]
# master_args =
# 进程日志记录文件
master_log = {install_path}/var/log/single_echo.log
# 传递给服务子进程的环境变量, 可以通过 getenv("SERVICE_ENV") 获得此值
# master_env = logme:FALSE, priority:E_LOG_INFO, action:E_LOG_PER_DAY, flush:sync_flush,
imit_size:512,\
#   sync_action:E_LOG_SEM, sem_name:/tmp/single_echo.sem

# 每个进程实例处理连接数的最大次数，超过此值后进程实例主动退出
single_use_limit = 250
# 每个进程实例的空闲超时时间，超过此值后进程实例主动退出
# single_idle_limit = 180
# 记录进程PID的位置(对于多进程实例来说没有意义)
single_pid_dir = {install_path}/var/pid
# 进程运行时所在的路径
single_queue_dir = {install_path}/var
# 读写超时时间, 单位为秒
single_rw_timeout = 1800
# 读缓冲区的缓冲区大小
single_buf_size = 8192
```

```
# 进程运行时的用户身份
single_owner = root

# single_in_flow_delay = 1
# single_owner = owner
# 用 select 进行循环时的时间间隔
# 单位为秒
# single_delay_sec = 1
# 单位为微秒
# single_delay_usec = 5000
# single_daemon_timeout = 1800
}
```

2.4 ）编译源文件

生成 single_echo 可执行程序

make

2.5 ）拷贝文件

将 single_echo 拷贝至 acl_project/dist/master/libexec/linux32 (假设操作系统是LINUX 32位平台的) 目录，将 single_echo.cf 拷贝至acl_project/dist/master/conf/service/ 目录，从而将 single_echo 置于 acl_master 守护管理进程的控制范围内。

2.6) 安装

cd acl_project/dist/master; chmod 755 setup.sh; ./setup.sh /opt/acl

2.7 ）启动框架管理控制进程(acl_master)

/opt/acl/sh/start.sh

2.8 ）手工测试

telnet 127.0.0.1 5003

看是否正常连接服务器，如果连接成功，则随意输入一些字符然后按回车发送，看服务器是否将所发送的数据回显给发送者；如果连接不成功或服务器未正常回显，请查看日志文件：/opt/acl/var/log/single_echo，并找出出错原因。

acl 库的下载地址：http://acl.sourceforge.net/

6.3 协作半驻留式服务器程序开发框架 --- 基于 Postfix 服务器框架改造

发表时间: 2009-08-15 关键字: 应用服务器, 框架, 多线程, 网络应用, 编程

一、概述

现在大家在和Java, PHP, .net写应用程序时，都会用到一些成熟的服务框架，所以开发效率是比较高的。而在用C/C++写服务器程序时，用的就五花八门了，有些人用ACE, 有些人用ICE（号称比ACE强许多），等等，这类服务器框架及库比较丰富，但入门门槛比较高，所以更多的人是自己直接写服务器程序，初始写时觉得比较简单，可时间久了，便会觉得难以扩展，性能低，容易出错。其实，Postfix 作者为我们提供了一个高效、稳定、安全的服务器框架模型，虽然Postfix主要用作邮件系统的 mta，但其框架设计却非常具有通用性。ACL(http://acl.sourceforge.net/) 的作者将Postfix的服务器框架模型抽取出来，形成了更加通用的服务器程序开发框架，使程序员在编写服务器程序时可以达到事半功倍的效果。本文主要介绍了ACL中acl_master服务器程序（基于Postifx服务器程序框架）的设计及功能。

二、框架设计图

如下图所示：

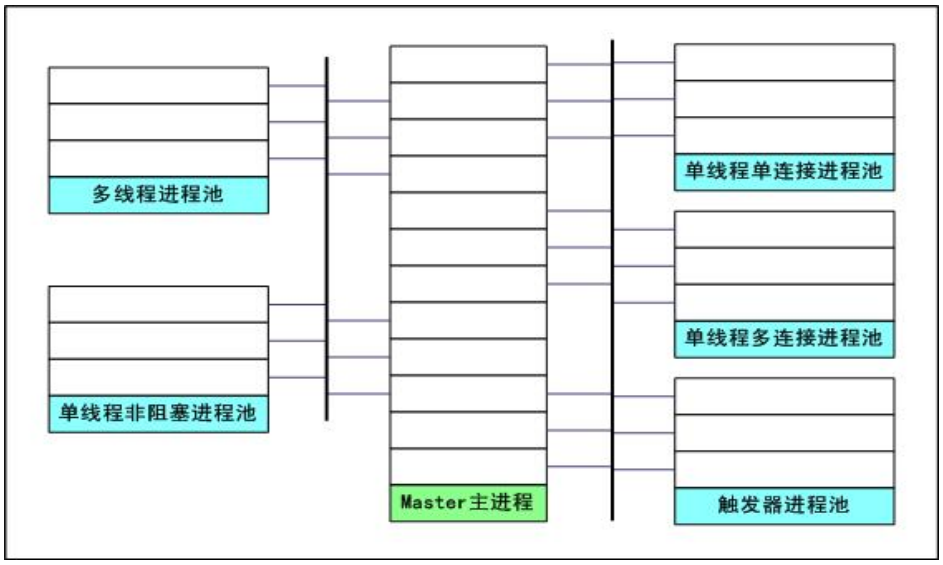


图1--框架图

master主进程为控制进程，刚启动时其负责监听所有端口服务，当有新的客户端连接到达时，master便会启动子进程进行服务，而自己依然监控服务端口，同时监控子进程的工作状态；而提供对外服务的子进程在master启动时，若没有请求任务则不会被启动，只有当有连接或任务到达时才会被master启动，当该服务子进程处理完某个连接服务后并不立即退出，而是驻留在系统一段时间，等待可能的新连接到达，这样当有新的连

接到达时master就不会启动新的子进程，因为已经有处于空闲的子进程在等待下一个连接请求；当服务子进程空闲时间达一定阀值后，就会选择退出，将资源全部归还操作系统（当然，也可以配置成服务子进程永不退出的模式）。因此，可以称这种服务器框架为协作式半驻留式服务器框架，下面将会对协作式和半驻留作进一步介绍。

三、协作方式

Postfix服务器框架设计的非常巧妙，因为master毕竟属于用户空间进程，不能象操作系统那样可以控制每个进程的运行时间片，所以master主进程必须与其服务子进程之间协作好，以处理好以下几个过程：

1）新连接到达时，master是该启动新的子进程接管该连接还是由空闲子进程直接接管

2）master何时应该启动新的子进程

3）新连接到达，空闲子进程池中的子进程如何竞争接管该连接

4）子进程异常退出时，master如何处理新连接

5）空闲子进程如何选择退出时间（空闲时间或服务次数应决定子进程的退出）

6）master如何知道各个子进程的工作状态（是死了还是活着？）

7）在不停止服务的前提下，服务子进程程序如果在线更新、如何添加新的服务、如何在线更新子进程配置

8）如何减少所有子进程与master之间的通讯次数从而降低master的负载

四、流程图

1) master主进程流程图

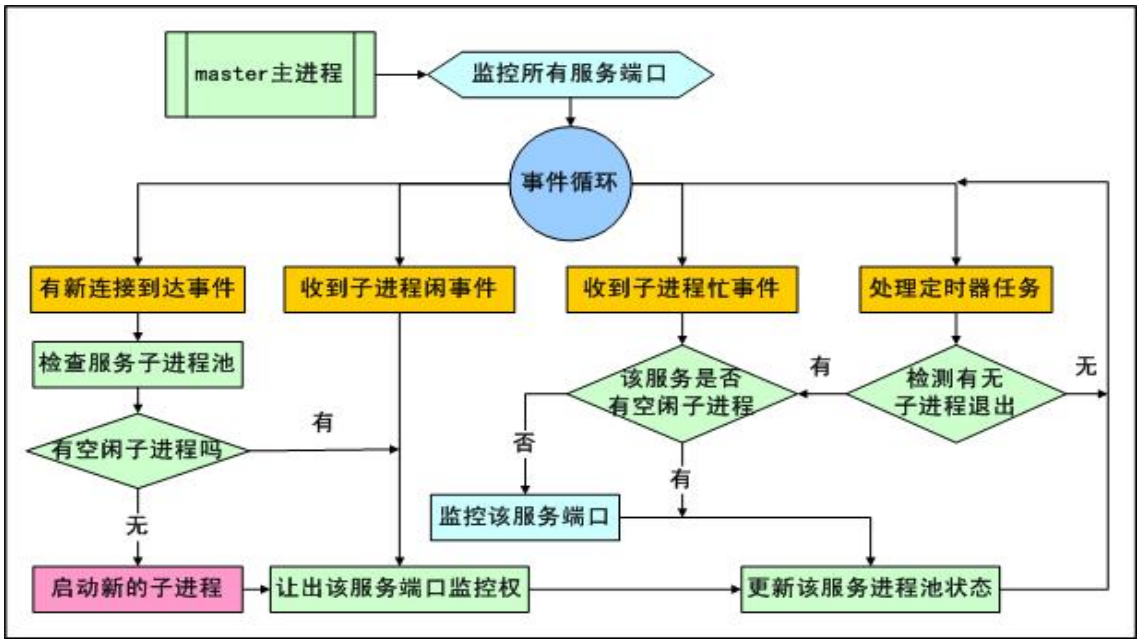


图2--master进程流程图

Postifx 中的 master 主进程与各个子进程之间的IPC通讯方式为管道，所以管道的个数与子进程数是成正比的。如果管道中断，则 master 认为该管道所对应的子进程已经退出，如果是异常退出，master还需要标记该服务类子进程池以防止该类子进程异常退出频繁而启动也异常频繁（如果子进程启动过于频繁则会给操作系统造成巨大负载）；另外，如果某类服务的子进程在服务第一个连接时就异常退出，则master认为该服务有可能是不可用的，所以当有新的连接再到达时就会延迟启动该服务子进程。

当服务子进程池中有空闲子进程时，master便会把该服务端口的监听权让出，从而该服务的空闲子进程在该服务端口上接收新的连接。当某个子进程获得新的连接后便会立即通知master其已经处于忙状态，master便立即查找该服务的子进程进程池还有无空闲子进程，如果有则master依然不会接管该服务端口的监听任务；如果没有了，则master立即接管该服务端口的监听任务，当有新的连接到达时，master先检查有没有该服务的空闲进程，若有便让出该服务端口的监听权，若没有便会启动新的子进程，然后让出监听权。

2）服务子进程流程图

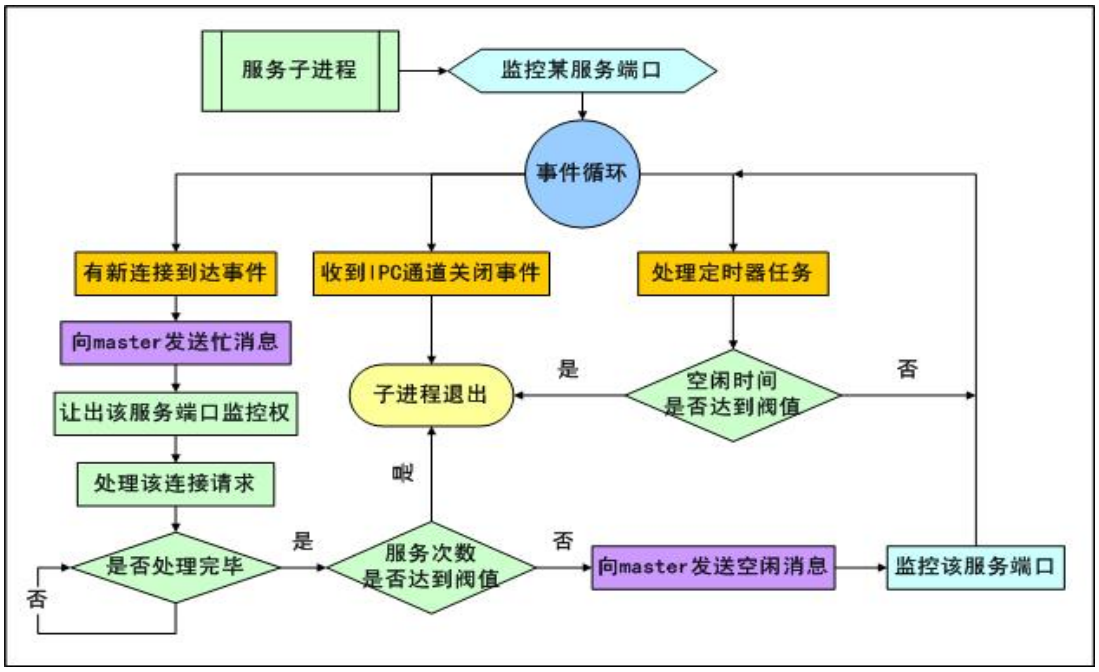


图3--子进程流程图

在master主进程刚启动时，因为没有任何服务请求，所以子进程是不随master一起启动的，此时所有服务端口的监控工作是由master统一负责，当有客户端连接到达时，服务子进程才由master启动，进而接收该新连接，在进一步处理客户端请求前，子进程必须让master进程知道它已经开始"忙"了，好由master来决定是否再次接管该服务端口的监控任务，所以子进程首先向master发送“忙”消息，然后才开始接收并处理该客户端请求，当子进程完成了对该客户端的请求任务后，需向master发送“空闲”消息，以表明自己又可以继续处理新的客户端连接任务了。这一“忙”一“闲”两个消息，便体现了服务子进程与master主进程的协作特点。

当然，服务子进程可以选择合适的退出时机：如果自己的服务次数达到配置的阀值，或自己空闲时间达到阀值，或与master主进程之间的IPC管道中断(一般是由master停止服务要求所有服务子进程退出时或master要求所有服务子进程重读配置时而引起的)，则服务子进程便应该结束运行了。这个停止过程，一方面体现了子进程与master主进程之间的协作特点，另一方面也体现了子进程半驻留的特点，从而体现子进程进程池的半驻留特性，这一特性的最大好处就是：按需分配，当请求连接比较多时，所启动运行的子进程就多，当请求连接任务较少时，只有少数的子进程在运行，其它的都退出了。

3) 小结

以上从整体上介绍了Postfix服务器框架模型中master主进程与服务子进程的逻辑流程图，当然，其内部实现要点与细节远比上面介绍的要复杂，只是这些复杂层面别人已经帮我们屏蔽了，我们所要做的是在此基础上写出自己的应用服务来，下面简要介绍了基于Postfix的服务器框架改造抽象的ACL库中服务器程序的开发方式，以使大家比较容易上手。

五、五种服务器框架模板简介

要想使用ACL服务器框架库开发服务器程序，首先介绍两个个小名词：acl_master服务器主进程与服务器模板。acl_master服务器主进程与Postfix中的master主进程功能相似，它的主要作用是启动并控制所有的服务子进程，这个程序不用我们写，是ACL服务器框架中已经写好的；所谓服务器模板，就是我们需要根据自己的需要从ACL服务器框架中选择一种服务器模型(即服务器模板)，然后在编写服务器时按该服务器模板的方式写即可。为什么还要选择合适的服务器框架模板？这是因为Postfix本身就提供了三类服务器应用形式(单进程单连接进程池、单进程多连接进程池、触发器进程池)，这三类应用形式便分别使用了Postfix中的三种服务器模板，此外，ACL中又增加了两种服务器应用形式(多线程进程池、单进程非阻塞进程池)。下面分别就这五种服务器框架模板一一做简介：

1) 单进程单连接进程池：由 acl_master 主进程启动多个进程组成进程池提供某类服务，但每个进程每次只能处理一个客户端连接请求

2) 单进程多连接进程池：由 acl_master 主进程启动多个进程组成进程池提供某类服务，而每个进程可以同时处理多个客户端连接请求

3) 触发器进程池：由 acl_master 主进程启动多个进程组成进程池提供定时器类服务(类似于UNIX中的cron)，当某个定时器时间到达时，便由一个进程开始运行处理任务

4) 多线程进程池：由 acl_master 主进程启动多个进程组成进程池提供某类服务，而每个进程是由多个线程组成，每个线程处理一个客户端连接请求

5) 单进程非阻塞进程池：由 acl_master 主进程启动多个进程组成进程池提供某类服务，每个进程可以并发处理多个连接(类似于Nginx, Lighttpd, Squid, Ircd)，由于采用非阻塞技术，该模型服务器的并发处理能力大大提高，同时系统资源消耗也最小；当然，该模型与单进程多连接进程池采用的技术都是非阻塞技术，但该模型进行更多的应用封装与高级处理，使编写非阻塞程序更加容易

以上五种服务器方式中，由于可以根据需要配置成多个进程实例，所以可以充分地利用多核的系统。其中，第5种的运行效率是最高的，当然其编程的复杂度要比其它的高，而第1种是执行效率最低的，其实它也是最安全的(在Postfix中的smtpd/smtp 等进程就属于这一类)，而相对来说， 第4种在运行效率与编写复杂度方面是一个比较好的折衷，所以在写一般性服务器时，该服务器模型是作者推荐的方案，此外，第4种方案还有一个好处，可以做到对于空连接不必占用线程，这样也大大提供了并发度(即线程数可以远小于连接数)。

六、使用简介

本节暂不介绍具体的编程过程，只是介绍一些配置使用过程。假设已经写好了服务器程序：echo_server, 该程序可以采用上面的 1), 2), 4), 5) 中的任一服务器模型来写，假设采用了第4)种；另外，还假设：acl_master 等所有文件安装的根目录为 /opt/acl/, 主进程程序 acl_master 及 echo_server 安装在 /opt/acl/libexec/, acl_master 主程序配置文件安装在 /opt/acl/conf/，echo_server 配置文件安装在 /opt/acl/conf/service/, 日志文件目录为 /opt/acl/var/log/, 进程号文件目录为 /opt/acl/var/pid/。比如，你让 echo_server 的服务端口为 6601，服务地址为 127.0.0.1, 它只是提供简单的 echo 服务。

你可以运行 /opt/acl/sh/start.sh 脚本来启动 acl_master 主进程(用 ps -ef|grep acl_master 会看到 acl_master 进程存在 ，但 ps -ef|grep echo_server 却没有发现它的存在)，然后你在一个Unix终端上 telnet 127.0.0.1 6601, 在另一个终端上 ps -ef|grep echo_server 就会发现有一个 echo_server子进程了，然后在 telnet 6601 的终端上随便输入一些字符串，便会立即得到回复，关闭该 telnet 连接，用 ps -ef|grep echo_server 会发现该进程依然存在，当再次 telnet 127.0.0.1 6601 时，该echo_server进程又继续为新连接提供服务了。可以试着多开几个终端同时 telnet 127.0.0.1 6601，看看运行效果如何？

注意，服务子进程的配置文件格式需要与其所采用的模板类型一致；进程池中最大进程个数、进程中线程池最大线程个数、进程最大服务次数、空闲时间等都是可以以配置文件中指定的。

参考文章：

1) 快速创建你的服务器程序 - - single进程池模型
2) 基于POSTFIX的服务器框架的服务器程序设计

3) 开发多线程进程池服务器程序---acl 服务器框架应用

4) 利用ACL库快速创建你的网络程序--ACL_VSTREAM 流的使用

5) 利用ACL库开发高并发半驻留式线程池程序

6.4 开发多线程进程池服务器程序---acl 服务器框架应用

发表时间: 2009-08-16 关键字: 应用服务器, 多线程, 框架, 网络应用, C

在“协作半驻留式服务器程序开发框架 --- 基于 Postfix 服务器框架改造 ”文章中，介绍了ACL库中协作式半驻留服务器程序框架，本文将以其中第4)种(多线程进程池)开发框架为基础编写一个简单的 demo 程序，使大家熟悉这类服务器程序的开发方式。

该 demo 一个简单的 echo 服务器程序，主要由 main.c, service_main.c, service_var.c, service_main.h, service_var.h, Makefile 六个文件组成。下面分别介绍一下各个文件的主要功能。

1) main.c 主程序

```
#include "lib_acl.h"  /* ACL库的头文件 */
#include <assert.h>
#include "service_main.h"
#include "service_var.h"

/* 测试函数入口 */
static void service_test(void)
{
        const char *addr = "127.0.0.1:8885";
        ACL_VSTREAM *sstream, *client;
        int   ret;

        sstream = acl_vstream_listen(addr, 32);  /* 创建服务端口的监听套接口并转化流 */
        assert(sstream != NULL);

        acl_xinetd_params_int_table(NULL, var_conf_int_tab);  /* 设置整数类型的配置项 */
        acl_xinetd_params_str_table(NULL, var_conf_str_tab);  /* 设置字符串类型的配置项 */
        acl_xinetd_params_bool_table(NULL, var_conf_bool_tab);  /* 设置 bool 类型的配置项 */

        printf("listen %s ok\n", addr);

        while (1) {
                client = acl_vstream_accept(sstream, NULL, 0);  /* 等待客户端连接 */
```

```
                if (client == NULL) {
                        printf("accept error: %s\n", acl_last_serror());
                        break;
                }

                /* 获得一个客户端连接流 */
                while (1) {
                        /* 处理该客户端的请求 */
                        ret = service_main(client, NULL);
                        if (ret < 0) {
                               /* 关闭客户端连接流 */
                                acl_vstream_close(client);
                                break;
                        }
                        if (ret > 0) {
                                /* service_main() 内部关闭了客户端流 */
                                break;
                        }
                }
        }

        /* 测试结束，关闭监听套接口 */
        acl_vstream_close(sstream);
}


/* 程序入口 */
int main(int argc, char *argv[])
{
        if (argc == 2 && strcasecmp(argv[1], "test") == 0) {
                /* 测试入口 */
                service_test();
        } else {
                /* 服务器框架入口 */
                acl_ioctl_app_main(argc, argv, service_main, NULL,
                                ACL_APP_CTL_INIT_FN, service_init,
                                ACL_APP_CTL_EXIT_FN, service_exit,
```

```
                                ACL_APP_CTL_CFG_BOOL, var_conf_bool_tab,
                                ACL_APP_CTL_CFG_INT, var_conf_int_tab,
                                ACL_APP_CTL_CFG_STR, var_conf_str_tab,
                                ACL_APP_CTL_END);
        }
        return (0);
}
```

该文件中，可以看到两个分支，一个是测试用入口 service_test()，另一个是服务器框架入口 acl_ioctl_app_main()。其中的 service_test() 主要是为了开发者调试自己的程序用，此时程序运行是独立运行的，不需要 acl_master 主进程参与；而 acl_ioctl_app_main() 则为服务器框架入口，需要 acl_master 主进程进行控制，这主要是用在生产环境中。

2) service_main.c 任务处理函数

```
#include "lib_acl.h"
#include "service_var.h"
#include "service_main.h"

/* 初始化函数 */
void service_init(void *init_ctx acl_unused)
{
        const char *myname = "service_init";

        acl_msg_info("%s: init ok ...", myname);
}

/* 进程退出前调用的函数 */
void service_exit(void *arg acl_unused)
{
        const char *myname = "service_exit";

        acl_msg_info("%s: exit now ...", myname);
}
```

```
/* 协议处理函数入口 */
int service_main(ACL_VSTREAM *client, void *run_ctx acl_unused)
{
	const char *myname = "service_main";
	char  buf[1024];
	int   ret;

	ret = acl_vstream_gets(client, buf, sizeof(buf));
	if (ret == ACL_VSTREAM_EOF) {
		if (var_cfg_debug_enable)
			acl_msg_info("%s: close client now, (%s)",
				myname, var_cfg_debug_msg);
		return (-1);  /* 返回负值以使框架内部关闭 client 数据流 */
	}

	if (acl_vstream_writen(client, buf, ret) == ACL_VSTREAM_EOF) {
		if (var_cfg_debug_enable)
			acl_msg_info("%s: write to client error, close now(%s)",
				myname, var_cfg_debug_msg);
		return (-1);  /* 返回负值以使框架内部关闭 client 数据流 */
	}

	if (var_cfg_keep_alive) {
		if (var_cfg_debug_enable)
			acl_msg_info("%s: keep alive, wait client...", myname);
		return (0);  /* 返回 0 以使框架内部自动监听该数据流从而保持长连接 */
	} else {
		/* 可以在此处返回 =1, 使框架内部自动关闭 client 数据流,
		 * 也可以在此处直接关闭 client 数据流，同时返回 1 告诉框架
		 * 该流已经被用户关闭了不必再关心该 client 数据流.
		 */
		acl_vstream_close(client);
		return (1);
	}
}
```

该文件中主要有三个函数(这三个函数都是在 main.c 中的 acl_ioctl_app_main() 设置的，这样服务器框架就以回调的方式分别调用它们): service_init(), 进程初始化回调函数，用户可以在些函数中做一些全局化的初始化，如数据库的连接建立 ；service_main(), 任务调用入口，每当有一个客户端连接建立时，服务器框架便会调用此函数由应用来处理与客户端的信息交流，其中的 client 参数为由服务器框架已经与客户端之间建立起的数据流，参数 run_ctx 可以在 acl_ioctl_app_main() 中进行设置；service_exit(), 当进程退出前便会回调此函数，用户可以在此函数里做一些清理工作。

3) service_var.c 配置参数源码

```
#include "lib_acl.h"
#include "service_var.h"

char *var_cfg_debug_msg;

ACL_CFG_STR_TABLE var_conf_str_tab[] = {
        { "debug_msg", "test_msg", &var_cfg_debug_msg },

        { 0, 0, 0 }
};

int  var_cfg_debug_enable;
int  var_cfg_keep_alive;

ACL_CFG_BOOL_TABLE var_conf_bool_tab[] = {
        { "debug_enable", 1, &var_cfg_debug_enable },
        { "keep_alive", 1, &var_cfg_keep_alive },

        { 0, 0, 0 }
};

int  var_cfg_io_timeout;

ACL_CFG_INT_TABLE var_conf_int_tab[] = {
        { "io_timeout", 120, &var_cfg_io_timeout, 0, 0 },
```

```
        { 0, 0 , 0 , 0, 0 }
};
```

该文件主要是一些全局化的配置项参数，主要有三类：int 类型的配置项，bool 类型的配置项，字符串类型的配置项，分别被设置在 var_conf_int_tab，var_conf_bool_tab，var_conf_str_tab 可，而这三个变量也是通过 acl_ioctl_app_main() 以参数方式传递给服务器框架，由框架读取配置文件后将配置内容分别设置在这三个变量中的具体配置变量中。

4) service_main.h 为 service_main.c 的头文件

```
#ifndef __SERVICE_MAIN_INCLUDE_H__
#define __SERVICE_MAIN_INCLUDE_H__

#include "lib_acl.h"

#ifdef  __cplusplus
extern "C" {
#endif

/**
 * 初始化函数，服务器模板框架启动后仅调用该函数一次
 * @param init_ctx {void*} 用户自定义类型指针
 */
extern void service_init(void *init_ctx);

/**
 * 进程退出时的回调函数
 * @param exist_ctx {void*} 用户自定义类型指针
 */
extern void service_exit(void *exit_ctx);

/**
 * 协议处理函数入口
 * @param stream {ACL_VSTREAM*} 客户端数据连接流
```

```
 * @param run_ctx {void*} 用户自定义类型指针
 */
extern int service_main(ACL_VSTREAM *stream, void *run_ctx);

#ifdef  __cplusplus
}
#endif

#endif
```

5) service_var.h 为 service_var.c 的头文件

```
#ifndef __SERVICE_VAR_INCLUDE_H__
#define __SERVICE_VAR_INCLUDE_H__

#include "lib_acl.h"

/*------------- 字符串配置项 ----------------*/

extern ACL_CFG_STR_TABLE var_conf_str_tab[];

/* 日志调试输出信息 */
extern char *var_cfg_debug_msg;

/*-------------- 布尔值配置项 ---------------*/

extern ACL_CFG_BOOL_TABLE var_conf_bool_tab[];

/* 是否输出日志调试信息 */
extern int var_cfg_debug_enable;

/* 是否与客户端保持长连接 */
extern int var_cfg_keep_alive;

/*-------------- 整数配置项 -----------------*/
```

```
extern ACL_CFG_INT_TABLE var_conf_int_tab[];

/* 每次与客户端通信时，读超时时间(秒) */
extern int  var_cfg_io_timeout;

#endif
```

6) Makefile 文件中记录着怎样与 ACL 库 lib_acl.a 及 ACL 的头文件进行编译连接，以及在不同UNIX平台下需要哪些系统库。

7) demo.cf 为配置文件

```
service server {
#       进程是否禁止运行
        master_disable = yes
#       服务地址及端口号
        master_service = 127.0.0.1:5001

#       服务监听为域套接口
#       master_service = aio_echo.sock
#       服务类型
        master_type = inet
#       master_type = unix

#       当子进程异常退出时，如果该值非空，则将子进程异常退出的消息通知该服务
#       master_notify_addr = 127.0.0.1:5801

#       是否允许延迟接受客户端连接，如果为0则表示关闭该功能，如果大于0则表示打开此功能
#       并且此值代表延迟接受连接的超时值，超过此值时如果客户端依然没有发来数据，则操作
#       系统会在系统层直接关闭该连接
#       master_defer_accept = 0
#       是否只允许私有访问, 如果为 y, 则域套接口创建在 {install_path}/var/log/private/ 目录下,
#       如果为 n, 则域套接口创建在 {install_path}/var/log/public/ 目录下,
        master_private = n
```

```
        master_unpriv = n
#       是否需要 chroot: n -- no, y -- yes
        master_chroot = n
#       每隔多长时间触发一次，单位为秒(仅对 trigger 模式有效)
        master_wakeup = -
#       最大进程数
        master_maxproc = 1
#       进程程序名
        master_command = ioctl_echo
#       进程日志记录文件
        master_log = {install_path}/var/log/ioctl_echo.log
#       进程启动参数，只能为: -u [是否允许以某普通用户的身份运行]
#       master_args =
#       传递给服务子进程的环境变量, 可以通过 getenv("SERVICE_ENV") 获得此值

#       master_env = logme:FALSE, priority:E_LOG_INFO, action:E_LOG_PER_DAY, flush:sync_flush,
imit_size:512,\
#                   sync_action:E_LOG_SEM, sem_name:/tmp/ioctl_echo.sem

#       每个进程实例处理连接数的最大次数，超过此值后进程实例主动退出
        ioctl_use_limit = 100
#       每个进程实例的空闲超时时间，超过此值后进程实例主动退出
        ioctl_idle_limit = 120
#       记录进程PID的位置(对于多进程实例来说没有意义)
        ioctl_pid_dir =  {install_path}/var/pid
#       进程运行时所在的路径
        ioctl_queue_dir = {install_path}/var
#       读写超时时间, 单位为秒
        ioctl_rw_timeout = 120
#       读缓冲区的缓冲区大小
        ioctl_buf_size = 8192
#       每次 accept 时的循环接收的最大次数
        ioctl_max_accept = 25
#       在并发访问量非常低的情况下，如访问量在 10 次/秒 以下时，可以找开此值(即赋为1)，以加速事件循环
过程,
#       从而防止服务进程阻塞在 select 上的时间过长而影响访问速度
#       ioctl_enable_dog = 0
```

```
#       进程运行时的用户身份
        ioctl_owner = root

#       用 select 进行循环时的时间间隔
#       单位为秒
        ioctl_delay_sec = 0
#       单位为微秒
        ioctl_delay_usec = 500

#       采用事件循环的方式: select(default), poll, kernel(epoll/devpoll/kqueue)
        ioctl_event_mode = select

#       线程池的最大线程数
        ioctl_max_threads = 250

#       线程的堆栈空间大小，单位为字节，0表示使用系统缺省值
        ioctl_stacksize = 0
#       允许访问 udserver 的客户端IP地址范围
        ioctl_access_allow = 127.0.0.1:255.255.255.255, 127.0.0.1:127.0.0.1

################################################################################
#       应用自己的配置选项
        debug_msg = test msg
        debug_enable = 1
        keep_alive = 1
}
```

小结，可以看出开发一个多线程的进程池服务程序是如此之简单，我们无需写一些复杂的服务器控制代码，这个过程完全由ACL服务器框架内部自动处理。这个例子在 acl库的 samples/master/ioctl_echo3 中可以看到，要想使其运行在生产环境下，需要将编译后的可执行程序及配置文件按 "协作半驻留式服务器程序开发框架 --- 基于 Postfix 服务器框架改造 "文章所介绍的 acl_master 框架的运行位置及配置位置中即可，然后运行 acl_master 中的 reload.sh 即可以将这个新的服务加载了。

参考：

协作半驻留式服务器程序开发框架 --- 基于 Postfix 服务器框架改造

利用ACL库快速创建你的网络程序

7.1 利用ACL库开发高并发半驻留式线程池程序

发表时间: 2009-06-07 关键字: 多线程, thread, VC++, 工作, FreeBSD

一、概述

在当今强调多核开发的年代，要求程序员能够写出高并发的程序，而利用多个核一般有两种方式：采用多线程方式或多进程方式。每处理一个新任务时如果临时产生一个线程或进程且处理完任务后线程或进程便立即退出，显示这种方式是非常低效的，于是人们一般采用线程池的模型（这在JAVA 或 .NET 中非常普遍）或多进程进程池模型（这一般在UNIX平台应用较多）。此外，对于线程池或进程池模型又分为两种情形：常驻留内存或半驻留内存，常驻内存是指预先产生一批线程或进程，等待新任务到达，这些线程或进程即使在空闲状态也会常驻内存；半驻留内存是指当来新任务时如果线程池或进程池没有可利用线程或进程则启动新的线程或进程来处理新任务，处理完后，线程或进程并不立即退出，而是空闲指定时间，如果在空闲阀值时间到达前有新任务到达则立即处理新任务，如果到达空闲超时后依然没有新任务到达，则这些空闲的线程或进程便退出，以让出系统资源。所以，对比常驻内存方式和半驻留内存方式，不难看出半驻留方式更有按需分配的意味。

下面仅以ACL框架中的半驻留线程池模型为例介绍了如何写一个半驻留线程池的程序。

二、半驻留线程池函数接口说明

2.1）线程池的创建、销毁及任务添加等接口

```
/**
 * 创建一个线程池对象
 * @param attr {acl_pthread_pool_attr_t*} 线程池创建时的属性，如果该参数为空，
 *  则采用默认参数: ACL_PTHREAD_POOL_DEF_XXX
 * @return {acl_pthread_pool_t*}, 如果不为空则表示成功，否则失败
*/
ACL_API acl_pthread_pool_t *acl_pthread_pool_create(const acl_pthread_pool_attr_t *attr);

/**
 * 销毁一个线程池对象，成功销毁后该对象不能再用.
 * @param thr_pool {acl_pthread_pool_t*} 线程池对象，不能为空
 * @return {int} 0: 成功; != 0: 失败
 */
ACL_API int acl_pthread_pool_destroy(acl_pthread_pool_t *thr_pool);

/**
 * 向线程池添加一个任务
 * @param thr_pool {acl_pthread_pool_t*} 线程池对象，不能为空
 * @param run_fn {void (*)(*)} 当有可用工作线程时所调用的回调处理函数
```

```
 * @param run_arg {void*} 回调函数 run_fn 所需要的回调参数
 * @return {int} 0: 成功; != 0: 失败
 */
ACL_API int acl_pthread_pool_add(acl_pthread_pool_t *thr_pool,
        void (*run_fn)(void *), void *run_arg);

/**
 * 当前线程池中的线程数
 * @param thr_pool {acl_pthread_pool_t*} 线程池对象，不能为空
 * @return {int} 返回线程池中的总线程数
 */
ACL_API int acl_pthread_pool_size(acl_pthread_pool_t *thr_pool);
```

2.2）线程池属性设置接口

```
/**
 * 初始化线程池属性值
 * @param attr {acl_pthread_pool_attr_t*}
 */
ACL_API void acl_pthread_pool_attr_init(acl_pthread_pool_attr_t *attr);

/**
 * 设置线程池属性中的最大堆栈大小(字节)
 * @param attr {acl_pthread_pool_attr_t*}
 * @param size {size_t}
 */
ACL_API void acl_pthread_pool_attr_set_stacksize(acl_pthread_pool_attr_t *attr, size_t size);

/**
 * 设置线程池属性中的最大线程数限制值
 * @param attr {acl_pthread_pool_attr_t*}
 * @param threads_limit {int} 线程池中的最大线程数
 */
ACL_API void acl_pthread_pool_attr_set_threads_limit(acl_pthread_pool_attr_t *attr,
                                                                        int t
```

```
/**
 * 设置线程池属性中线程空闲超时值
 * @param attr {acl_pthread_pool_attr_t*}
 * @param idle_timeout {int} 线程空闲超时时间(秒)
 */
ACL_API void acl_pthread_pool_attr_set_idle_timeout(acl_pthread_pool_attr_t *attr,
                                                    int id
```

2.3 ）线程池中的工作线程创建、退出时设置回调函数接口

```
/**
 * 添加注册函数，在线程创建后立即执行此初始化函数
 * @param thr_pool {acl_pthread_pool_t*} 线程池对象，不能为空
 * @param init_fn {int (*)(void*)} 工作线程初始化函数，如果该函数返回 < 0,
 *  则该线程自动退出。
 * @param init_arg {void*} init_fn 所需要的参数
 * @return {int} 0: OK; != 0: Error.
 */
ACL_API int acl_pthread_pool_atinit(acl_pthread_pool_t *thr_pool,
        int (*init_fn)(void *), void *init_arg);

/**
 * 添加注册函数，在线程退出立即执行此初函数
 * @param thr_pool {acl_pthread_pool_t*} 线程池对象，不能为空
 * @param free_fn {void (*)(void*)} 工作线程退出前必须执行的函数
 * @param free_arg {void*} free_fn 所需要的参数
 * @return {int} 0: OK; != 0: Error.
 */
ACL_API int acl_pthread_pool_atfree(acl_pthread_pool_t *thr_pool,
        void (*free_fn)(void *), void *free_arg);
```

三、半驻留线程池例子

3.1 ）程序代码

```
#include "lib_acl.h"
#include <assert.h>

/**
 * 用户自定义数据结构
 */
typedef struct THREAD_CTX {
        acl_pthread_pool_t *thr_pool;
        int   i;
} THREAD_CTX;

/* 全局性静态变量 */
static acl_pthread_pool_t *__thr_pool = NULL;

/* 线程局部存储变量(C99支持此种方式声明，方便许多) */
static __thread unsigned int __local = 0;

static void work_thread_fn(void *arg)
{
        THREAD_CTX *ctx = (THREAD_CTX*) arg; /* 获得用户自定义对象 */
        int   i = 5;

        /* 仅是验证参数传递过程 */
        assert(ctx->thr_pool == __thr_pool);

        while (i-- > 0) {
                printf("thread start! tid=%d, i=%d, __local=%d\r\n",
                                acl_pthread_self(), ctx->i, __local);
                /* 在本线程中将线程局部变量加1 */
                __local++;
                sleep(1);
        }

        acl_myfree(ctx);

        /* 至此，该工作线程进入空闲状态，直到空闲超时退出 */
}
```

```
static int on_thread_init(void *arg)
{
        const char *myname = "on_thread_init";
        acl_pthread_pool_t *thr_pool = (acl_pthread_pool_t*) arg;

        /* 判断一下，仅是为了验证参数传递过程 */
        assert(thr_pool == __thr_pool);
        printf("%s: thread(%d) init now\r\n", myname, acl_pthread_self());

        /* 返回0表示继续执行该线程获得的新任务，返回-1表示停止执行该任务 */
        return (0);
}

static void on_thread_exit(void *arg)
{
        const char *myname = "on_thread_exit";
        acl_pthread_pool_t *thr_pool = (acl_pthread_pool_t*) arg;

        /* 判断一下，仅是为了验证参数传递过程 */
        assert(thr_pool == __thr_pool);
        printf("%s: thread(%d) exit now\r\n", myname, acl_pthread_self());
}

static void run_thread_pool(acl_pthread_pool_t *thr_pool)
{
        THREAD_CTX *ctx;  /* 用户自定义参数 */

        /* 设置全局静态变量 */
        __thr_pool = thr_pool;

        /* 设置线程开始时的回调函数 */
        (void) acl_pthread_pool_atinit(thr_pool, on_thread_init, thr_pool);

        /* 设置线程退出时的回调函数 */
        (void) acl_pthread_pool_atfree(thr_pool, on_thread_exit, thr_pool);
```

```
        ctx = (THREAD_CTX*) acl_mycalloc(1, sizeof(THREAD_CTX));
        assert(ctx);
        ctx->thr_pool = thr_pool;
        ctx->i = 0;

        /**
         * 向线程池中添加第一个任务，即启动第一个工作线程
         * @param thr_pool 线程池句柄
         * @param workq_thread_fn 工作线程的回调函数
         * @param ctx 用户定义参数
         */
        acl_pthread_pool_add(thr_pool, work_thread_fn, ctx);
        sleep(1);

        ctx = (THREAD_CTX*) acl_mycalloc(1, sizeof(THREAD_CTX));
        assert(ctx);
        ctx->thr_pool = thr_pool;
        ctx->i = 1;
        /* 向线程池中添加第二个任务，即启动第二个工作线程 */
        acl_pthread_pool_add(thr_pool, work_thread_fn, ctx);
}

int main(int argc acl_unused, char *argv[] acl_unused)
{
        acl_pthread_pool_t *thr_pool;
        int  max_threads = 20;  /* 最多并发20个线程 */
        int  idle_timeout = 10; /* 每个工作线程空闲10秒后自动退出 */
        acl_pthread_pool_attr_t attr;

        acl_pthread_pool_attr_init(&attr);
        acl_pthread_pool_attr_set_threads_limit(&attr, max_threads);
        acl_pthread_pool_attr_set_idle_timeout(&attr, idle_timeout);

        /* 创建半驻留线程句柄 */
        thr_pool = acl_pthread_pool_create(&attr);
        assert(thr_pool);
        run_thread_pool(thr_pool);
```

```
        if (0) {
                /* 如果立即运行 acl_pthread_pool_destroy，则由于调用了线程池销毁函数，
                 * 主线程便立刻通知空闲线程退出，所有空闲线程不必等待空闲超时时间便可退出，
                 */
                printf("> wait all threads to be idle and free thread pool\r\n");
                /* 立即销毁线程池 */
                acl_pthread_pool_destroy(thr_pool);
        } else {
                /* 因为不立即调用 acl_pthread_pool_destroy，所有所有空闲线程都是当空闲
                 * 超时时间到达后才退出
                 */
                while (1) {
                        int   ret;

                        ret = acl_pthread_pool_size(thr_pool);
                        if (ret == 0)
                                break;
                        printf("> current threads in thread pool is: %d\r\n", ret);
                        sleep(1);
                }
                /* 线程池中的工作线程数为0时销毁线程池 */
                printf("> all worker thread exit now\r\n");
                acl_pthread_pool_destroy(thr_pool);
        }

        /* 主线程等待用户在终端输入任意字符后退出 */
        printf("> enter any key to exit\r\n");
        getchar();

        return (0);
}
```

3.2) 编译链接

从 http://www.sourceforge.net/projects/acl/ 站点下载 acl_project 代码，在WIN32平台下请用

VC2003编译，打开 acl_project\win32_build\vc\acl_project_vc2003.sln 编译后在目录 acl_project\dist\lib_acl\lib\win32 下生成lib_acl_vc2003.lib, 然后在示例的控制台工程中包含该库，并包含acl_project\lib_acl\include 下的 lib_acl.h 头文件，编译上述源代码即可。

因为本例子代码在 ACL 的例子里有，所以可以直接编译 acl_project\win32_build\vc\samples\samples_vc2003.sln 中的 thread_pool 项目即可。

3.3) 运行

运行示例程序后，在我的机器的显示结果如下：

```
on_thread_init: thread(23012) init now
thread start! tid=23012, i=0, __local=0
thread start! tid=23012, i=0, __local=1
> current threads in thread pool is: 2
on_thread_init: thread(23516) init now
thread start! tid=23516, i=1, __local=0
thread start! tid=23516, i=1, __local=1
> current threads in thread pool is: 2
thread start! tid=23012, i=0, __local=2
thread start! tid=23516, i=1, __local=2
thread start! tid=23012, i=0, __local=3
> current threads in thread pool is: 2
thread start! tid=23516, i=1, __local=3
thread start! tid=23012, i=0, __local=4
> current threads in thread pool is: 2
thread start! tid=23516, i=1, __local=4
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
> current threads in thread pool is: 2
on_thread_exit: thread(23012) exit now
> current threads in thread pool is: 1
```

on_thread_exit: thread(23516) exit now
> all worker thread exit now
> enter any key to exit

四、小结

可以看出，使用ACL库创建半驻留式高并发多线程程序是比较简单的，ACL线程池库的接口定义及实现尽量与POSIX中规定的POSIX线程的实现接口相似，创建与使用ACL线程池库的步骤如下：

a) acl_pthread_pool_attr_init: 初始化创建线程池对象所需要属性信息(可以通过 acl_pthread_pool_attr_set_threads_limit 设置线程池最大并发数及用 acl_pthread_pool_attr_set_idle_timeout 设置线程池中工作线程的空闲退出时间间隔)

b) acl_pthread_pool_create: 创建线程池对象

c) acl_pthread_pool_add: 向线程池中添加新任务，新任务将由线程池中的某一工作线程执行

d) acl_pthread_pool_destroy: 通知并等待线程池中的工作线程执行完任务后退出，同时销毁线程池对象

还可以在选择在创建线程池对象后，调用 acl_pthread_pool_atinit 设置工作线程第一次被创建时回调用户自定义函数过程，或当线程空闲退出后调用 acl_pthread_pool_atfree 中设置的回调函数。

另外，可以将创建线程池的过程进行一抽象，写成如下形式：

```
/**
 * 创建半驻留线程池的过程
 * @return {acl_pthread_pool_t*} 新创建的线程池句柄
 */
static acl_pthread_pool_t *create_thread_pool(void)
{
        acl_pthread_pool_t *thr_pool;  /* 线程池句柄 */
        int  max_threads = 100;  /* 最多并发100个线程 */
        int  idle_timeout = 10;  /* 每个工作线程空闲10秒后自动退出 */
        acl_pthread_pool_attr_t attr;  /* 线程池初始化时的属性 */

        /* 初始化线程池对象属性 */
        acl_pthread_pool_attr_init(&attr);
        acl_pthread_pool_attr_set_threads_limit(&attr, max_threads);
        acl_pthread_pool_attr_set_idle_timeout(&attr, idle_timeout);

        /* 创建半驻留线程句柄 */
        thr_pool = acl_pthread_pool_create(&attr);
        assert(thr_pool);
```

```
        return (thr_pool);
}
```

其实，利用ACL创建线程池还有一个简化接口（只所以叫 acl_thread_xxx 没有加 p, 是因为这个接口不太遵守 Posix的一些规范），如下：

```
/**
 * 更简单地创建线程对象的方法
 * @param threads_limit {int}  线程池中最大并发线程数
 * @param idle_timeout {int} 工作线程空闲超时退出时间(秒)，如果为0则工作线程永不退出
 * @return {acl_pthread_pool_t*}, 如果不为空则表示成功，否则失败
 */
ACL_API acl_pthread_pool_t *acl_thread_pool_create(int threads_limit, int idle_timeout);
```

这样，用户就可以非常方便地创建自己的线程池了，而且别忘了，这个线程池还是可以是半驻留的（当然也是跨平台的，可以运行在 Linux/Solaris/FreeBSD/Win32 的环境下）。

为了方便使用ACL库，用户可以参看ACL的在线帮助文档：http://acl.sourceforge.net/acl_help/index.html。ACL下载位置：http://acl.sourceforge.net/

7.2 多线程开发时线程局部变量的使用

发表时间: 2009-09-22 关键字: 多线程, thread, Linux, Unix

一、概述

现在多核时代多线程开发越来越重要了，多线程相比于多进程有诸多优势（当然也有诸多劣势）。在早期C的库中，有许多函数是线程不安全的，因为内部用到了静态变量，比如：char *strtok(char *s, const char *delim); 该函数内部就有一个静态指针，如果多个线程同时调用此函数时，可能就会出现奇怪的结果，当然也不是我们所想要的，现在LINUX对此函数功能有一个线程安全版本的接口：char *strtok_r(char *s, const char *delim, char **ptrptr)，这就避免了多个线程同时访问的冲突问题。其实，如果保持 strtok()/2 接口不变，同时还要保证线程安全，还有一个解决办法，那就是采用线程局部变量。

使用线程局部变量有两种使用方式，一个稍微麻烦些，一个比较简单，下面一一做个介绍（以LINUX为例）

二、线程局部变量的使用

比较麻烦些的使用方法用到的函数主要有三个：pthread_once(pthread_once_t*, void (*init_routine)(void)), pthread_key_create()/2, pthread_setspecific()/2, pthread_getspecific()/1，其中 pthread_once 可以保证在整个进程空间init_routine函数仅被调用一次（它解决了多线程环境中使得互斥量和初始化代码都仅被初始化一次的问题）；pthread_key_create 的参数之一指一个析构函数指针，当某个线程终止时该析构函数将被调用，并用对于一个进程内的给定键，该函数只能被调用一次；pthread_sespecific 和 pthread_getspecific 用来存放和获取与一个键关联的值。例子如下：

```
pthread_key_t key;
pthread_once_t once = PTHREAD_ONCE_INIT;

static void destructor(void *ptr)
{
    free(ptr);
}

void init_once(void)
{
    pthread_key_create(&key, destructor);
}
```

```
static void *get_buf(void)
{
    pthread_once(&once, init_once);

    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(1024);
        pthread_setspecific(key, ptr);
    }
    return (ptr);
}

static void *thread_fn(void *arg)
{
    char *ptr = (char*) get_buf();

    sprintf(ptr, "hello world");
    printf(">>%s\n", ptr);
    return (NULL);
}

void test(void)
{
    int   i, n = 10;
    pthread_t tids[10];

    for (i = 0; i < n; i++) {
        pthread_create(&tids[i], NULL, thread_fn, NULL);
    }

    for (i = 0; i < n; i++) {
        pthread_join(&tids[i], NULL);
    }
}
```

另外，还有一个更加简单使用线程局部变量的方法：__thread 修饰符, (在WIN32平台下需要用: __declspec(thread) 修饰符，WIN32的东东总得要多写几笔，呵呵)，于是上述代码可以修改如下：

```
static void *get_buf(void)
{
    static __thread void *ptr = malloc(1024);
    return (ptr);
}

static void *thread_fn(void *arg)
{
    char *ptr = (char*) get_buf();

    sprintf(ptr, "hello world");
    printf(">>%s\n", ptr);
    return (NULL);
}

void test(void)
{
    int   i, n = 10;
    pthread_t tids[10];

    for (i = 0; i < n; i++) {
        pthread_create(&tids[i], NULL, thread_fn, NULL);
    }

    for (i = 0; i < n; i++) {
        pthread_join(&tids[i], NULL);
    }
}
```

看到没有，这段代码比前面一个简单许多，但却有一个问题，它存在内存泄露问题，因为当线程退出时各个线程分配的动态内存(ptr = malloc(1024)) 并没有被释放。

三、用ACL线程接口操作线程局部变量

为了解决上述问题，ACL库中实现了线程局部变量的简单释放功能：acl_pthread_atexit_add(void *arg, void (*free_callback)(void*))，修改上述代码如下：

```
static void free_fn(void *ptr)
{
    free(ptr);
}

static void *get_buf(void)
{
    static __thread void *ptr = malloc(1024);

    acl_pthread_atexit_add(ptr, free_fn);
    return (ptr);
}

static void *thread_fn(void *arg)
{
    char *ptr = (char*) get_buf();

    sprintf(ptr, "hello world");
    printf(">>%s\n", ptr);
    return (NULL);
}

void test(void)
{
    int   i, n = 10;
    pthread_t tids[10];

    for (i = 0; i < n; i++) {
        acl_pthread_create(&tids[i], NULL, thread_fn, NULL);
    }
```

```
    for (i = 0; i < n; i++) {
        acl_pthread_join(&tids[i], NULL);
    }
}
```

ok, 一切问题得到解决。细心的读者会发现 pthread_create, pthread_join 前面都加了前缀: acl_, 这是因为ACL库对线程库进行了封装，以适应不同平台下(UNIX、WIN32)下的使用，这个例子是跨平台的，WIN32下同样可用。

7.3 再谈线程局部变量

发表时间: 2009-12-15 关键字: 多线程, thread, Linux, Unix, 编程

在文章 多线程开发时线程局部变量的使用 中，曾详细提到如何使用 __thread (Unix 平台) 或 __declspec(thread) (win32 平台）这类修饰符来申明定义和使用线程局部变量（当然在ACL库里统一了使用方法，将 __declspec(thread) 重定义为 __thread)，另外，为了能够正确释放由 __thread 所修饰的线程局部变量动态分配的内存对象，ACL库里增加了个重要的函数：acl_pthread_atexit_add()/2，此函数主要作用是当线程退出时自动调用应用的释放函数来释放动态分配给线程局部变量的内存。以 __thread 结合 acl_pthread_atexit_add()/2 来使用线程局部变量非常简便，但该方式却存在以下主要的缺点(将 __thread/ __declspec(thread) 类线程局部变量方式称为 “静态 TLS 模型”）：

如果动态库（.so 或 .dll)内部有以 __thread/__declspec(thread) 申明使用的线程局部变量，而该动态库被应用程序动态加载（dlopen/LoadLibrary）时，如果使用这些局部变量会出现内存非法越界问题，原因是动态库被可执行程序动态加载时此动态库中的以“静态TLS模型”定义的线程局部变量无法被系统正确地初始化（参见：Sun 的C/C++ 编程接口 及 MSDN 中有关 “静态 TLS 模型” 的使用注意事项）。

为解决 “静态 TLS 模型” 不能动态装载的问题，可以使用 “动态 TLS 模型”来使用线程局部变量。下面简要介绍一下 Posix 标准和 win32 平台下 “动态 TLS 模型” 的使用：

1、Posix 标准下 “动态 TLS 模型” 使用举例：

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

static pthread_key_t key;

// 每个线程退出时回调此函数来释放线程局部变量的动态分配的内存
static void destructor(void *arg)
{
    free(arg);
}

static init(void)
{
    // 生成进程空间内所有线程的线程局部变量所使用的键值
```

```
    pthread_key_create(&key, destructor);
}

static void *thread_fn(void *arg)
{
    char *ptr;

    // 获得本线程对应 key 键值的线程局部变量
    ptr = pthread_getspecific(key);
    if (ptr == NULL) {
        // 如果为空，则生成一个
        ptr = malloc(256);
        // 设置对应 key 键值的线程局部变量
        pthread_setspecific(key, ptr);
    }

     /* do something */

     return (NULL);
}

static void run(void)
{
     int   i, n = 10;
     pthread_t tids[10];

    // 创建新的线程
    for (i = 0; i < n; i++) {
        pthread_create(&tids[i], NULL, thread_fn, NULL);
    }

    // 等待所有线程退出
    for (i = 0; i < n; i++) {
        pthread_join(&tids[i], NULL);
    }
}
```

```
int main(int argc, char *argv[])
{
    init();
    run();
    return (0);
}
```

可以看出，在同一进程内的各个线程使用同样的线程局部变量的键值来“取得/设置”线程局部变量，所以在主线程中先初始化以获得一个唯一的键值。如果不能在主线程初始化时获得这个唯一键值怎么办？ Posix 标准规定了另外一个函数：pthread_once(pthread_once_t *once_control, void (*init_routine)(void)), 这个函数可以保证 init_routine 函数在多线程内仅被调用一次，稍微修改以上例子如下：

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

static pthread_key_t key;

// 每个线程退出时回调此函数来释放线程局部变量的动态分配的内存
static void destructor(void *arg)
{
    free(arg);
}

static init(void)
{
    // 生成进程空间内所有线程的线程局部变量所使用的键值
    pthread_key_create(&key, destructor);
}

static pthread_once_t once_control = PTHREAD_ONCE_INIT;

static void *thread_fn(void *arg)
{
    char *ptr;

    // 多个线程调用 pthread_once 后仅能是第一个线程才会调用 init 初始化
```

```
    // 函数，其它线程虽然也调用 pthread_once 但并不会重复调用 init 函数，
    // 同时 pthread_once 保证 init 函数在完成前其它线程都阻塞在
    // pthread_once 调用上（这一点很重要，因为它保证了初始化过程）
    pthread_once(&once_control, init);

    // 获得本线程对应 key 键值的线程局部变量
    ptr = pthread_getspecific(key);
    if (ptr == NULL) {
        // 如果为空，则生成一个
        ptr = malloc(256);
        // 设置对应 key 键值的线程局部变量
        pthread_setspecific(key, ptr);
    }

     /* do something */

     return (NULL);
}

static void run(void)
{
     int   i, n = 10;
     pthread_t tids[10];

    // 创建新的线程
    for (i = 0; i < n; i++) {
        pthread_create(&tids[i], NULL, thread_fn, NULL);
    }

    // 等待所有线程退出
    for (i = 0; i < n; i++) {
        pthread_join(&tids[i], NULL);
    }
}

int main(int argc, char *argv[])
{
```

```
    run();
    return (0);
}
```

可见 Posix 标准当初做此类规定时是多么的周全与谨慎，因为最早期的 C 标准库有很多函数都是线程不安全的，后来通过这些规定，使 C 标准库的开发者可以“修补“这些函数为线程安全类的函数。

2、win32 平台下 “动态 TLS 模型” 使用举例：

```
static DWORD key;

static void init(void)
{
    // 生成线程局部变量的唯一键索引值
    key = TlsAlloc();
}

static DWORD WINAPI thread_fn(LPVOID data)
{
    char *ptr;

    ptr = (char*) TlsGetValue(key);  // 取得线程局部变量对象
    if (ptr == NULL) {
        ptr = (char*) malloc(256);
        TlsSetValue(key, ptr);  // 设置线程局部变量对象
    }

    /* do something */

    free(ptr);  // 应用自己需要记住释放由线程局部变量分配的动态内存
    return (0);
}

static void run(void)
{
    int   i, n = 10;
```

```
    unsigned int tid[10];
    HANDLE handles[10];

    // 创建线程
    for (i = 0; i < n; i++) {
       handles[i] =  _beginthreadex(NULL,
                                    0,
                                    thread_fn,
                                    NULL,
                                    0,
                                    &tid[i]);
    }

    // 等待所有线程退出
    for (i = 0; i < n; i++) {
        WaitForSingleObject(handles[i]);
    }
}

int main(int argc, char *argv[])
{
    init();
    run();
    return (0);
}
```

在 win32 下使用线程局部变量与 Posix 标准有些类似，但不幸的是线程局部变量所动态分配的内存需要自己记着去释放，否则会造成内存泄露。另外还有一点区别是，在 win32 下没有 pthread_once()/2 类似函数，所以我们无法直接在各个线程内部调用 TlsAlloc() 来获取唯一键值。在ACL库模拟实现了 pthread_once()/2 功能的函数，如下：

```
int acl_pthread_once(acl_pthread_once_t *once_control, void (*init_routine)(void))
{
        int   n = 0;
```

```
        if (once_control == NULL || init_routine == NULL) {
                acl_set_error(ACL_EINVAL);
                return (ACL_EINVAL);
        }

        /* 只有第一个调用 InterlockedCompareExchange 的线程才会执行 init_routine,
         * 后续线程永远在 InterlockedCompareExchange 外运行，并且一直进入空循环
         * 直至第一个线程执行 init_routine 完毕并且将 *once_control 重新赋值,
         * 只有在多核环境中多个线程同时运行至此时才有可能出现短暂的后续线程空循环
         * 现象，如果多个线程顺序至此，则因为 *once_control 已经被第一个线程重新
         * 赋值而不会进入循环体内
         * 只所以如此处理，是为了保证所有线程在调用 acl_pthread_once 返回前
         * init_routine 必须被调用且仅能被调用一次
         */
        while (*once_control != ACL_PTHREAD_ONCE_INIT + 2) {
                if (InterlockedCompareExchange(once_control,
                        1, ACL_PTHREAD_ONCE_INIT) == ACL_PTHREAD_ONCE_INIT)
                {
                        /* 只有第一个线程才会至此 */
                        init_routine();
                        /* 将 *conce_control 重新赋值以使后续线程不进入 while 循环或
                         * 从 while 循环中跳出
                         */
                        *once_control = ACL_PTHREAD_ONCE_INIT + 2;
                        break;
                }
                /* 防止空循环过多地浪费CPU */
                if (++n % 100000 == 0)
                        Sleep(10);
        }
        return (0);
}
```

3、使用ACL库编写跨平台的 “动态 TLS 模型” 使用举例：

```
#include "lib_acl.h"
#include <stdlib.h>
#include <stdio.h>

static acl_pthread_key_t key = -1;

// 每个线程退出时回调此函数来释放线程局部变量的动态分配的内存
static void destructor(void *arg)
{
    acl_myfree(arg);
}

static init(void)
{
    // 生成进程空间内所有线程的线程局部变量所使用的键值
    acl_pthread_key_create(&key, destructor);
}

static acl_pthread_once_t once_control = ACL_PTHREAD_ONCE_INIT;

static void *thread_fn(void *arg)
{
    char *ptr;

    // 多个线程调用 acl_pthread_once 后仅能是第一个线程才会调用 init 初始化
    // 函数，其它线程虽然也调用 acl_pthread_once 但并不会重复调用 init 函数，
    // 同时 acl_pthread_once 保证 init 函数在完成前其它线程都阻塞在
    // acl_pthread_once 调用上（这一点很重要，因为它保证了初始化过程）
    acl_pthread_once(&once_control, init);

    // 获得本线程对应 key 键值的线程局部变量
    ptr = acl_pthread_getspecific(key);
    if (ptr == NULL) {
        // 如果为空，则生成一个
        ptr = acl_mymalloc(256);
        // 设置对应 key 键值的线程局部变量
        acl_pthread_setspecific(key, ptr);
```

```
    }

     /* do something */

     return (NULL);
}

static void run(void)
{
     int   i, n = 10;
     acl_pthread_t tids[10];

    // 创建新的线程
    for (i = 0; i < n; i++) {
        acl_pthread_create(&tids[i], NULL, thread_fn, NULL);
    }

    // 等待所有线程退出
    for (i = 0; i < n; i++) {
        acl_pthread_join(&tids[i], NULL);
    }
}

int main(int argc, char *argv[])
{
    acl_init();  // 初始化 acl 库
    run();
    return (0);
}
```

这个例子是跨平台的，它消除了UNIX、WIN32平台之间的差异性，同时当我们在WIN32下开发多线程程序及使用线程局部变量时不必再那么烦锁了，但直接这么用依然存在一个问题：因为每创建一个线程局部变量就需要分配一个索引键，而每个进程内的索引键是有数量限制的（在LINUX下是1024，BSD下是256，在WIN32下也就是1000多），所以如果要以”TLS动态模型“创建线程局部变量还是要小心不可超过系统限制。ACL库对这一限制做了扩展，理论上讲用户可以设定任意多个线程局部变量（取决于你的可用内存大小），下面主要介绍一下如何用ACL库来打破索引键的系统限制来创建更多的线程局部变量。

4、使用ACL库创建线程局部变量

接口介绍如下：

```
/**
 * 设置每个进程内线程局部变量的最大数量
 * @param max {int} 线程局部变量限制数量
 */
ACL_API int acl_pthread_tls_set_max(int max);

/**
 * 获得当前进程内线程局部变量的最大数量限制
 * @return {int} 线程局部变量限制数量
 */
ACL_API int acl_pthread_tls_get_max(void);

/**
 * 获得对应某个索引键的线程局部变量，如果该索引键未被初始化则初始之
 * @param key_ptr {acl_pthread_key_t} 索引键地址指针，如果是由第一
 *    个线程调用且该索引键还未被初始化(其值应为 -1)，则自动初始化该索引键
 *    并将键值赋予该指针地址，同时会返回NULL; 如果 key_ptr 所指键值已经
 *    初始化，则返回调用线程对应此索引键值的线程局部变量
 * @return {void*} 对应索引键值的线程局部变量
 */
ACL_API void *acl_pthread_tls_get(acl_pthread_key_t *key_ptr);

/**
 * 设置某个线程对应某索引键值的线程局部变量及自动释放函数
 * @param key {acl_pthread_key_t} 索引键值，必须是 0 和
 *    acl_pthread_tls_get_max() 返回值之间的某个有效的数值，该值必须
 *    是由 acl_pthread_tls_get() 初始化获得的
 * @param ptr {void*} 对应索引键值 key 的线程局部变量对象
 * @param free_fn {void (*)(void*)} 线程退出时用此回调函数来自动释放
 *    该线程的线程局部变量 ptr 的内存对象
 * @return {int} 0: 成功; !0: 错误
 * @example:
 *    static void destructor(void *arg)
```

```
 *    {
 *        acl_myfree(arg};
 *    }
 *    static void test(void)
 *    {
 *        static acl_pthread_key_t key = -1;
 *        char *ptr;
 *
 *        ptr = acl_pthread_tls_get(&key);
 *        if (ptr == NULL) {
 *            ptr = (char*) acl_mymalloc(256);
 *            acl_pthread_tls_set(key, ptr, destructor);
 *        }
 *    }
 */
ACL_API int acl_pthread_tls_set(acl_pthread_key_t key, void *ptr, void (*free_fn)(void *));
```

现在使用ACL库中的这些新的接口函数来重写上面的例子如下：

```
#include "lib_acl.h"
#include <stdlib.h>
#include <stdio.h>

// 每个线程退出时回调此函数来释放线程局部变量的动态分配的内存
static void destructor(void *arg)
{
    acl_myfree(arg);
}

static void *thread_fn(void *arg)
{
    static acl_pthread_key_t key = -1;
    char *ptr;

    // 获得本线程对应 key 键值的线程局部变量
```

```
    ptr = acl_pthread_tls_get(&key);
    if (ptr == NULL) {
        // 如果为空，则生成一个
        ptr = acl_mymalloc(256);
        // 设置对应 key 键值的线程局部变量
        acl_pthread_tls_set(key, ptr, destructor);
    }

    /* do something */

    return (NULL);
}

static void run(void)
{
     int   i, n = 10;
     acl_pthread_t tids[10];

    // 创建新的线程
    for (i = 0; i < n; i++) {
        acl_pthread_create(&tids[i], NULL, thread_fn, NULL);
    }

    // 等待所有线程退出
    for (i = 0; i < n; i++) {
        acl_pthread_join(&tids[i], NULL);
    }
}

int main(int argc, char *argv[])
{
    acl_init();  // 初始化ACL库
    // 打印当前可用的线程局部变量索引键的个数
    printf(">>>current tls max: %d\n", acl_pthread_tls_get_max());
    // 设置可用的线程局部变量索引键的限制个数
    acl_pthread_tls_set_max(10240);
```

```
    run();
    return (0);
}
```

这个例子似乎又比前面的例子更加简单灵活，如果您比较关心ACL里的内部实现，请直接下载ACL库源码(http://acl.sourceforge.net/)，参考 acl_project/lib_acl/src/thread/, acl_project/lib_acl/include/thread/ 下的内容。

8.1 使用 acl 库开发一个 HTTP 下载客户端

发表时间: 2010-01-11 关键字: 网络协议, F#, Cache, HTML, Apache

在 acl 的协议库(lib_protocol) 中有专门针对 HTTP 协议和 ICMP 协议的，本文主要介绍如何使用 lib_protocol 协议库来开发一个简单的 http 客户端。下面首先介绍一下几个本文用到的函数接口。

```
/**
 * 创建一个 HTTP_UTIL 请求对象
 * @param url {const char*} 完整的请求 url
 * @param method {const char*} 请求方法，有效的请求方法有：GET, POST, HEAD, CONNECT
 * @return {HTTP_UTIL*}
 */
HTTP_API HTTP_UTIL *http_util_req_new(const char *url, const char *method);

/**
 * 设置 HTTP 代理服务器地址
 * @param http_util {HTTP_UTIL*}
 * @param proxy {const char*} 代理服务器地址，有效格式为: IP:PORT, DOMAIN:PORT,
 *  如: 192.168.0.1:80, 192.168.0.2:8088, www.g.cn:80
 */
HTTP_API void http_util_set_req_proxy(HTTP_UTIL *http_util, const char *proxy);

/**
 * 设置 HTTP 响应体的转储文件，设置后 HTTP 响应体数据便会转储于该文件
 * @param http_util {HTTP_UTIL*}
 * @param filename {const char*} 转储文件名
 * @return {int} 如果返回值 < 0 则表示无法打开该文件，否则表示打开文件成功
 */
HTTP_API int http_util_set_dump_file(HTTP_UTIL *http_util, const char *filename);

/**
 * 打开远程 HTTP 服务器或代理服务器连接，同时构建 HTTP 请求头数据并且将该数据
 * 发给新建立的网络连接
 * @param http_util {HTTP_UTIL*}
 * @return {int} 0: 成功; -1: 无法打开连接或发送请求头数据失败
 */
```

```
HTTP_API int http_util_req_open(HTTP_UTIL *http_util);

/**
 * 发送完请求数据后调用此函数从 HTTP 服务器读取完整的 HTTP 响应头
 * @param http_util {HTTP_UTIL*}
 * @return {int} 0: 成功; -1: 失败
 */
HTTP_API int http_util_get_res_hdr(HTTP_UTIL *http_util);

/**
 * 读完 HTTP 响应头后调用此函数从 HTTP 服务器读取 HTTP 数据体数据，需要连续调用
 * 此函数，直至返回值 <= 0，如果之前设置了转储文件或转储则在读取数据过程中同时会
 * 拷贝一份数据给转储文件或转储流
 * @param http_util {HTTP_UTIL*}
 * @param buf {char *} 存储 HTTP 响应体的缓冲区
 * @param size {size_t} buf 的空间大小
 * @return {int} <= 0: 表示读结束; > 0: 表示本次读到的数据长度
 */
HTTP_API int http_util_get_res_body(HTTP_UTIL *http_util, char *buf, size_t size);
```

以上仅是 lib_http_util.h 函数接口中的一部分，下面就写一个简单的例子：

```
#include "lib_acl.h"
#include "lib_protocol.h"

static void get_url(const char *method, const char *url,
        const char *proxy, const char *dump, int out)
{
        /* 创建 HTTP_UTIL 请求对象 */
        HTTP_UTIL *http = http_util_req_new(url, method);
        int   ret;

        /* 如果设定代理服务器，则连接代理服务器地址，
         * 否则使用 HTTP 请求头里指定的地址
         */
```

```
if (proxy && *proxy)
        http_util_set_req_proxy(http, proxy);

/* 设置转储文件 */
if (dump && *dump)
        http_util_set_dump_file(http, dump);

/* 输出 HTTP 请求头内容 */

http_hdr_print(&http->hdr_req->hdr, "---request hdr---");

/* 连接远程 http 服务器 */

if (http_util_req_open(http) < 0) {
        printf("open connection(%s) error\n", http->server_addr);
        http_util_free(http);
        return;
}

/* 读取 HTTP 服务器响应头*/

ret = http_util_get_res_hdr(http);
if (ret < 0) {
        printf("get reply http header error\n");
        http_util_free(http);
        return;
}

/* 输出 HTTP 响应头 */

http_hdr_print(&http->hdr_res->hdr, "--- reply http header ---");

/* 如果有数据体则开始读取 HTTP 响应数据体部分 */
while (1) {
        char  buf[4096];
```

```
                ret = http_util_get_res_body(http, buf, sizeof(buf) - 1);
                if (ret <= 0)
                        break;
                buf[ret] = 0;
                if (out)
                        printf("%s", buf);
        }
        http_util_free(http);
}

static void usage(const char *procname)
{
        printf("usage: %s -h[help] -t method -r url -f dump_file -o[output] -X proxy_addr\n"
                "example: %s -t GET -r http://www.sina.com.cn/ -f url_dump.txt\n",
                procname, procname);
}

int main(int argc, char *argv[])
{
        int   ch, out = 0;
        char  url[256], dump[256], proxy[256], method[32];

        acl_init();  /* 初始化 acl 库 */

        ACL_SAFE_STRNCPY(method, "GET", sizeof(method));
        url[0] = 0;
        dump[0] = 0;
        proxy[0] = 0;
        while ((ch = getopt(argc, argv, "hor:t:f:X:")) > 0) {
                switch (ch) {
                case 'h':
                        usage(argv[0]);
                        return (0);
                case 'o':
                        out = 1;
                        break;
                case 'r':
```

```
                        ACL_SAFE_STRNCPY(url, optarg, sizeof(url));
                        break;
                case 't':
                        ACL_SAFE_STRNCPY(method, optarg, sizeof(method));
                        break;
                case 'f':
                        ACL_SAFE_STRNCPY(dump, optarg, sizeof(dump));
                        break;
                case 'X':
                        ACL_SAFE_STRNCPY(proxy, optarg, sizeof(proxy));
                        break;
                default:
                        break;
                }
        }

        if (url[0] == 0) {
                usage(argv[0]);
                return (0);
        }

        get_url(method, url, proxy, dump, out);
        return (0);
}
```

编译成功后，运行 ./url_get -h 会给出如下提示：

usage: ./url_get -h[help] -t method -r url -f dump_file -o[output] -X proxy_addr
example: ./url_get -t GET -r http://www.sina.com.cn/ -f url_dump.txt

输入: ./url_get -t GET -r http://www.sina.com -o， 该命令是获取 www.sina.com 页面并输出至标准输出，得到的结果为：

```
HTTP/1.0 301 Moved Permanently
Date: Tue, 12 Jan 2010 01:54:39 GMT
Server: Apache
Location: http://www.sina.com.cn/
Cache-Control: max-age=3600
```

```
Expires: Tue, 12 Jan 2010 02:54:39 GMT
Vary: Accept-Encoding
Content-Length: 231
Content-Type: text/html; charset=iso-8859-1
Age: 265
X-Cache: HIT from tj175-135.sina.com.cn
Connection: close
--------------- end -----------------
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.sina.com.cn/">here</a>.</p>
</body></html>
```

如果想把页面转存至文件中，可以输入：./url_get -t GET -r http://www.sina.com -f dump.txt, 这样就会把新浪的首页下载并存储于 dump.txt 文件中。

这个例子非常简单，其实如果查看 http_util.c 源码，会看到这个文件是对 lib_http.h 里一些更为底层 API 的封装。

如果仅是下载一个页面至某个文件中，其实还有更为简单的方法，只需要调用接口：

```
/**
 * 将某个 url 的响应体数据转储至某个文件中
 * @param url {const char*} 完整请求 url, 如: http://www.g.cn
 * @param dump {const char*} 转储文件名
 * @param {int} 读到的响应体数据长度, >=0: 表示成功, -1: 表示失败
 */
HTTP_API int http_util_dump_url(const char *url, const char *dump);
```

只这一个函数便可以达到与上一个例子相同的效果。

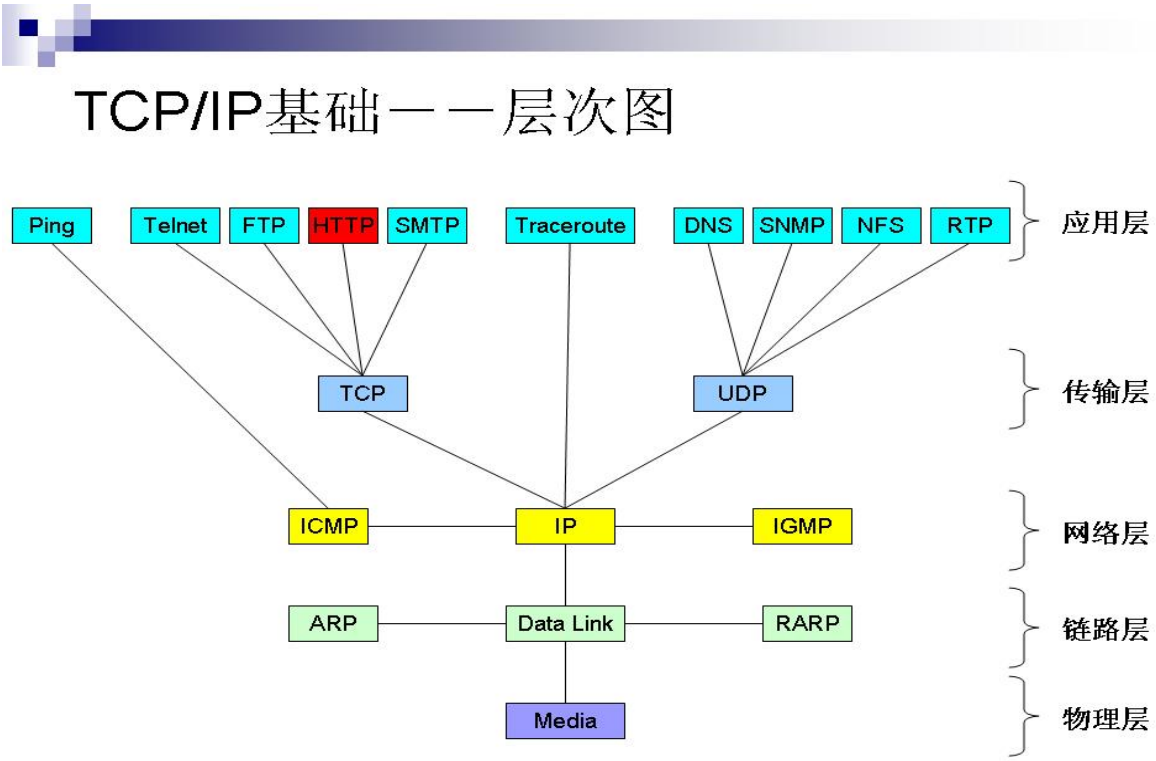
可以从 acl.sourceforge.net 上下载最新的 acl_project 库，查看 samples/http/ 下的三个例子，看一下下载WEB文件的不同方式。

8.2 HTTP 协议简介

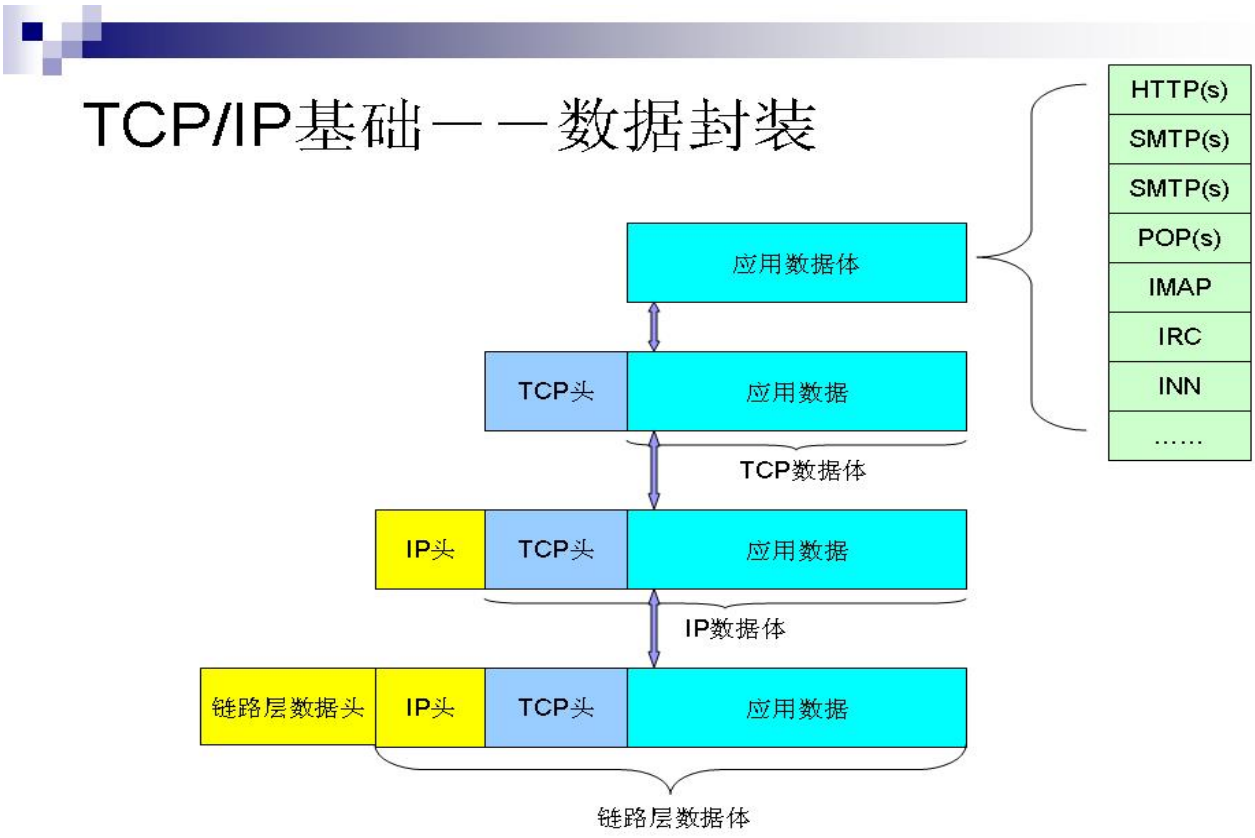
发表时间: 2010-01-12 关键字: 网络协议, 应用服务器, Windows, Cache, 网络应用

一、TCP/IP 协议介绍

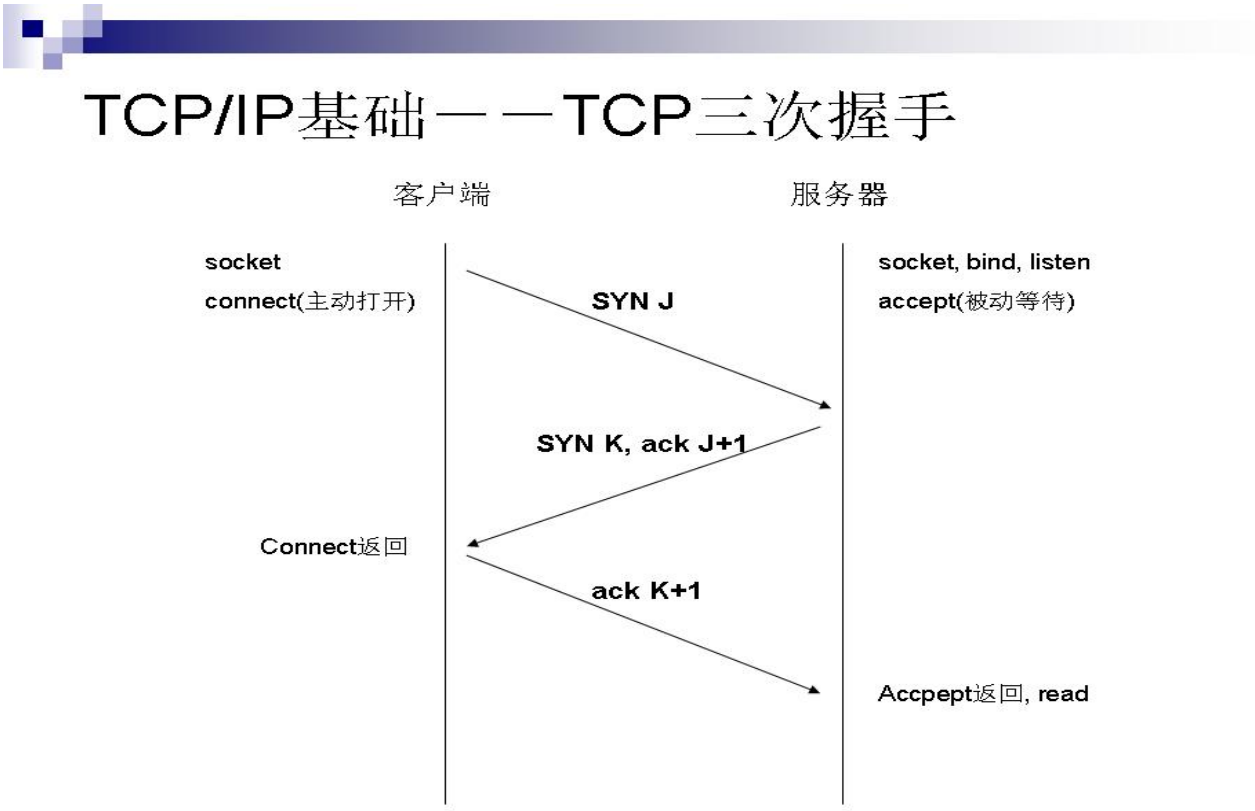
在介绍 HTTP 协议之前，先简单说一下TCP/IP协议的相关内容。TCP/IP协议是分层的，从底层至应用层分别为：物理层、链路层、网络层、传输层和应用层，如下图所示：



从应用层至物理层，数据是一层层封装，封装的方式一般都是在原有数据的前面加一个数据控制头，数据封装格式如下：



其中，对于TCP传输协议，客户端在于服务器建立连接前需要经过TCP三层握手，过程如下：



二、HTTP协议

2.1 简介

超文本传输协议（Hypertext Transfer Protocol，简称HTTP）是应用层协议，自 1990 年起，HTTP 就已经被应用于 WWW 全球信息服务系统。

HTTP 是一种请求/响应式的协议。一个客户机与服务器建立连接后，发送一个请求给服务器；服务器接到请求后，给予相应的响应信息。

HTTP 的第一版本 HTTP/0.9是一种简单的用于网络间原始数据传输的协议；

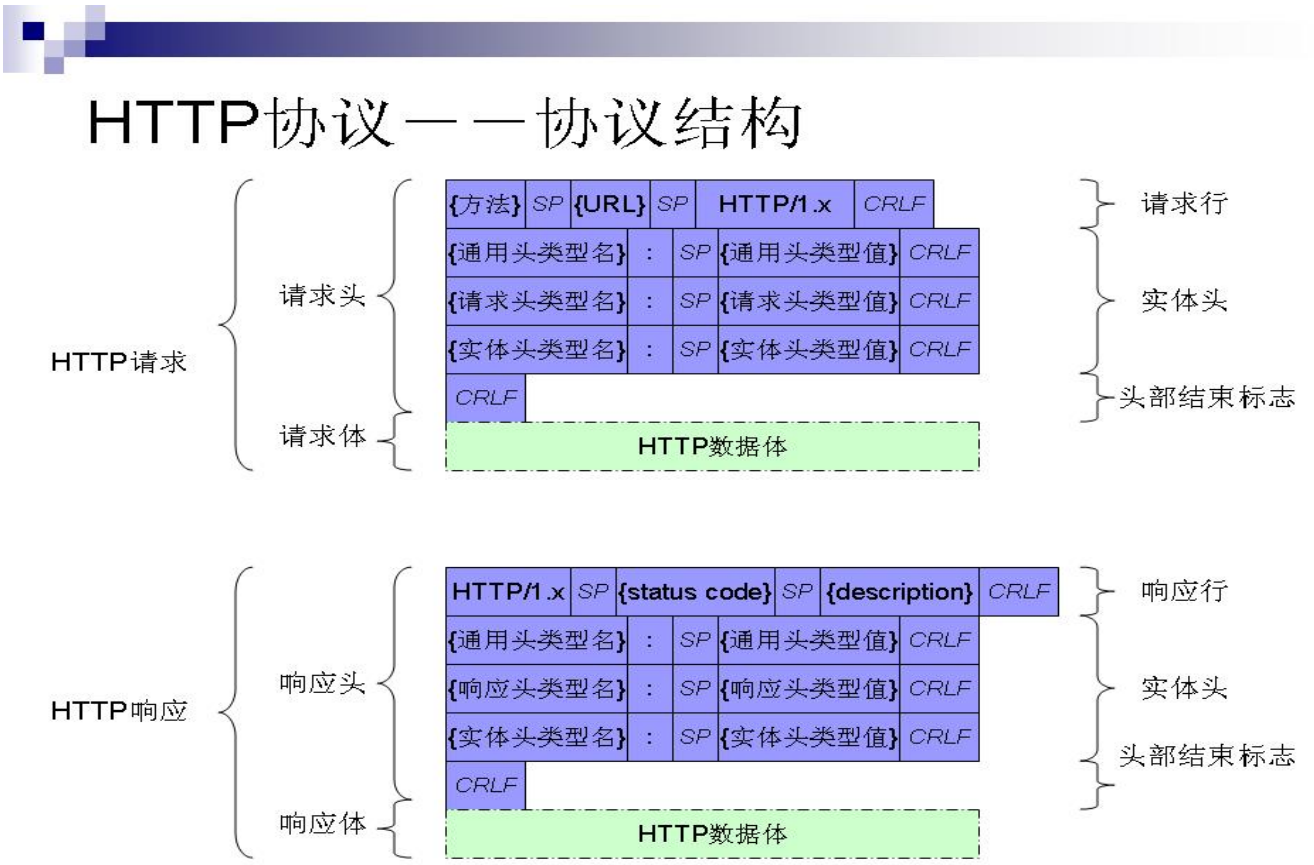
HTTP/1.0由 RFC 1945 定义 ，在原 HTTP/0.9 的基础上，有了进一步的改进，允许消息以类 MIME 信息格式存 在，包括请求/响应范式中的已传输数据和修饰符等方面的信息；

HTTP/1.1(RFC2616) 的要求更加严格以确保服务的可靠性，增强了在HTTP/1.0 没有充分考虑到分层代理服务器、高速缓冲存储器、持久连接需求或虚拟主机等方面的效能；

安全增强版的 HTTP （即S-HTTP或HTTPS），则是HTTP协议与安全套接口层(SSL)的结合，使HTTP的协议数据在传输过程中更加安全。

2.2 协议结构

HTTP协议格式也比较简单，格式如下：



2.3 HTTP 协议举例

下面是一个HTTP请求及响应的例子：

2.4 请求头格式

a) 通用头(general-header)：
Cache-Control：客户端希望服务端如何缓存自己的请求数据，如"Cache-Control: no-cache"，"Cache-Control: max-age=0"；
Connection：客户端是否希望与服务端之间保持长连接，如"Connection: close", "Connection: keep-alive"；
Date：只有当请求方法为POST或PUT方法时客户端才可能会有些字段；
Pragma：包含了客户端一些特殊请求信息，如 "Pragma: no-cache" 客户端希望代理或应用服务器不应缓存与该请求相关的结果数据；
Via：一般用在代理网关向应用服务器发送的请求头中，表明该来自客户端的请求经过了网关代理，
格式为："Via: 请求协议版本 网关标识 [其它信息] "，
如 ：" Via: 1.1 webcache_250_199.hexun.com:80 (squid)"

b) 请求头(request-header)：
Accept： 表明客户同端可接受的请求回应的媒体类型范围列表。星号“*”用于按范围将类型分组，用“*/*”指示可接受全部类型；用“type/*”指示可接受 type类型的所有子类型，如“ Accept: image/gif, image/jpeg, */*” ；
Accept-Charset：客户端所能识别的字符集编码格式，格式：“Accept-Charset: 字符集1[:权重]，字符集2[:权重]” ，如：“ Accept-Charset: iso-8859-5, unicode-1-1;q=0.8” ；
Accept-Language：客户端所能识别的语言，格式：“Accept-Language: 语言1[:权重]，语言2[:权重]” ，如：” Accept-Language: zh, en;q=0.7” ；
Host：客户请求的主机域名或主机IP，格式：“Host: 域名或IP[:端口号]” ，如：“Host: www.hexun.com:80 “，请求行中若有HTTP/1.1则必须有该请求头；
User-Agent：表明用户所使用的浏览器标识，主要用于统计的目的；
Referer：指明该请求是从哪个关联连接而来；

Accept-Encoding：客户端所能识别的编码压缩格式，如：“Accept-Encoding: gzip, deflate” ；
If- Modified-Since：该字段与客户端缓存相关，客户端所访问的URL自该指定日期以来在服务端是否被修改过，如果修改过则服务端返回新的修改后 的信息，如果未修改过则服务器返回304表明此请求所指URL未曾修改过，如：“If-Modified-Since: Fri, 2 Sep 2006 19:37:36 GMT” ；
If-None-Match：该字段与客户端缓存相关，客户端发送URL请求的同时发送该字段及标识，如 果服务端的标识与客户端的标识一致，则返回304表明此URL未修改过，如果不一致则服务端返回完整的数据信息，如：“If-None-Match: 0f0a893aad8c61:253, 0f0a893aad8c61:252, 0f0a893aad8c61:251” ；

Cookie：为扩展字段，存储于客户端，向同一域名的服务端发送属于该域的cookie，如：“Cookie: MailUserName=whouse” ；

c) 实体头(entity-header): (此类头存在时要求有数据体)
Content-Encoding：客户端所能识别的编码压缩格式，如：“Content-Encoding: gzip, deflate” ；
Content-Length：客户端以POST方法上传数据时数据体部分的内容长度，如：“ Content-Length: 24” ；
Content- Type：客户端发送的数据体的内容类型，如：“Content-Type: application/x-www-form-urlencoded”为以普通的POST方法发送的数据；“Content-Type: multipart/form-data; boundary=----------5169208281820” ，则表明数据体由多部分组成，分隔符为 “----------5169208281820” ；

2.5 ）响应格式

a) 通用头(general-header)：
Cache- Control：服务端要求中间代理及客户端如何缓存自己响应的数据，如“Cache-Control: no-cache” ，如：“Cache-Control: private” 不希望被缓存，“Cache-Control: public” 可以被缓存；
Connection：服务端是否希望与客户端之间保持长连接，如“Connection: close”，“Connection: keep-alive” ；
Date：只有当请求方法为POST或PUT方法时客户端才可能会有些字段；
Pragma：包含了服务端一些特殊响应信息，如 “Pragma: no-cache” 服务端希望代理或客户端不应缓存结果数据；
Transfer-Encoding：服务端向客户端传输数据所采用的传输模式(仅在HTTP1.1中出现)，如：“Transfer-Encoding: chunked” ，注：该字段的优先级要高于“Content-Length” 字段的优先级；

b ）响应头(response-header)：
Accept-Ranges：表明服务端接收的数据单位，如：“Accept-Ranges: bytes”，；
Location：服务端向客户端返回此信息以使客户端进行重定向，如：“Location: http://www.hexun.com” ；
Server：服务端返回的用于标识自己的一些信息，如：“ Server: Microsoft-IIS/6.0” ；
ETag：服务端返回的响应数据的标识字段，客户端可根据此字段的值向服务器发送某URL是否更新的信息；

c ）实体头(entity-header): (此类头存在时要求有数据体)
Content-Encoding：服务端所响应数据的编码格式，如：“Content-Encoding: gzip” ；

Content-Length：服务端所返回数据的数据体部分的内容长度，如：“ Content-Length: 24” ；
Content-Type：服务端所返回的数据体的内容类型，如：“Content-Type: text/html; charset=gb2312” ；
Set-Cookie：服务端返回给客户端的cookie数据，如：“ Set-Cookie: ASP.NET_SessionId=icnh2ku2dqlmkciyobgvzl55; path=/”

2.6）服务器返回状态码

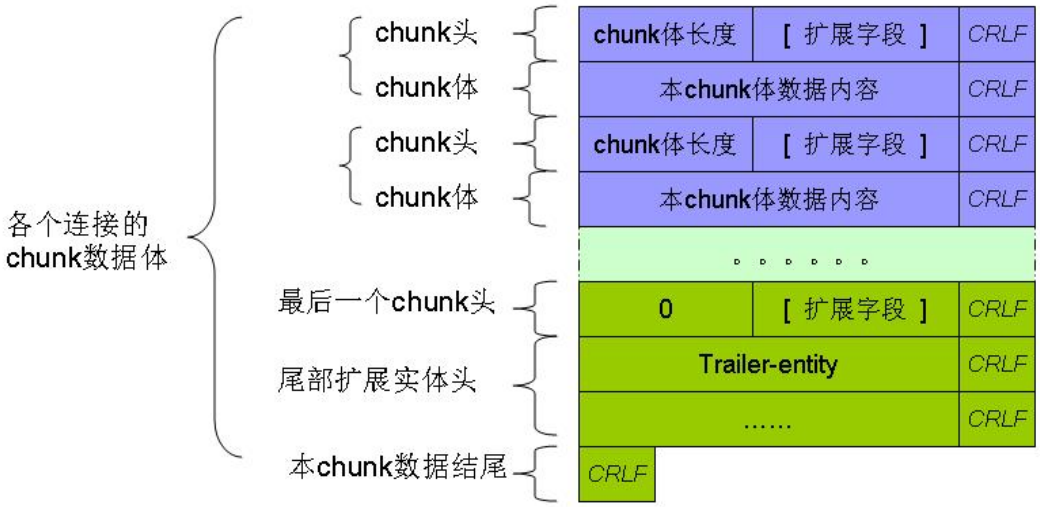
1xx：表明服务端接收了客户端请求，客户端继续发送请求；
2xx：客户端发送的请求被服务端成功接收并成功进行了处理；
3xx：服务端给客户端返回用于重定向的信息；
4xx：客户端的请求有非法内容；
5xx：服务端未能正常处理客户端的请求而出现意外错误。

举例：

“100” ；服务端希望客户端继续；
“200” ；服务端成功接收并处理了客户端的请求；
“301” ；客户端所请求的URL已经移走，需要客户端重定向到其它的URL；
“304” ；客户端所请求的URL未发生变化；
“400” ；客户端请求错误；
“403” ；客户端请求被服务端所禁止；
“404” ；客户端所请求的URL在服务端不存在；
“500” ；服务端在处理客户端请求时出现异常；
“501” ；服务端未实现客户端请求的方法或内容；
“502” ；此为中间代理返回给客户端的出错信息，表明服务端返回给代理时出错；
“503” ；服务端由于负载过高或其它错误而无法正常响应客户端请求；
“504” ；此为中间代理返回给客户端的出错信息，表明代理连接服务端出现超时。

2.7）chunked 传输

编码使用若干个Chunk组成，由一个标明长度为0的chunk结束，每个Chunk有两部分组成，第一部分是该Chunk的长度(以十六进制表示)和 长度单位（一般不写），第二部分就是指定长度的内容，每个部分用CRLF隔开。在最后一个长度为0的Chunk中的内容是称为footer的内容，是一些 没有写的头部内容。另外，在HTTP头里必须含有：” Transfer-Encoding: chunked” 通用头字段。格式如下：



2.8）HTTP 请求方法

GET、POST、HEAD、CONNECT、PUT、DELETE、TRACE

2.9）举例

a）GET请求

```
GET http://photo.test.com/inc/global.js HTTP/1.1
Host: photo.test.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; zh-CN; rv:1.8.1) Gecko/20061010 Firefox/2.
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/p
Accept-Language: en-us,zh-cn;q=0.7,zh;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
```

```
Proxy-Connection: keep-alive
Cookie: ASP.NET_SessionId=ey5drq45lsomio55hoydzc45
Cache-Control: max-age=0
```

b) POST请求

```
POST / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, application/vnd.ms-powerpoint, application/mswo
Accept-Language: zh-cn
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: www.test.com
Content-Length: 24
Connection: Keep-Alive
Cache-Control: no-cache

name=value&submit=submit
```

c) 通过HTTP代理发送GET请求

```
GET http://mail.test.com/ HTTP/1.1
Host: mail.test.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; zh-CN; rv:1.8.1) Gecko/20061010 Firefox/2.
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/p
Accept-Language: en-us,zh-cn;q=0.7,zh;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
```

d) POST方式上传文件

```
POST http://www.test.comt/upload_attach?uidl=%3C HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; zh-CN; rv:1.8.1) Gecko/20061010 Firefox/2.
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/p
Accept-Language: en-us,zh-cn;q=0.7,zh;q=0.3
Accept-Charset: gb2312,utf-8;q=0.7,*;q=0.7
Content-Type: multipart/form-data; boundary=---------------------------5169208281820
Content-Length: 449

-----------------------------5169208281820
Content-Disposition: form-data; name="file_1"; filename=""
Content-Type: application/octet-stream



-----------------------------5169208281820
Content-Disposition: form-data; name="file_0"; filename="test.txt"
Content-Type: text/plain

hello world!

-----------------------------5169208281820
Content-Disposition: form-data; name="oper"

upload
-----------------------------5169208281820--
```

e ）CONNECT举例

```
CONNECT mail.test.com:80 HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; zh-CN; rv:1.8.1) Gecko/20061010 Firefox/2.
Proxy-Connection: keep-alive
Host: mail.test.com:80
```

3.0 ）在终端以 telnet 方式测试

a) 打开回显功能 (针对windows)

Windows 2000：进入DOS模式->输入 telnet->set LOCAL_ECHO->退出：quit->telnet ip 80

Windows xp：进入DOS模式->输入telnet->set local echo->open ip 80

b) 按HTTP协议格式输入GET请求、HEAD请求、POST请求。

参考：

文章 《用C++实现类似于JAVA HttpServlet 的编程接口》给出了 acl_cpp 库中提供的 WEB 编程示例。

文章 《使用 acl 库开发一个 HTTP 下载客户端》给出了利用 acl 库写的 HTTP 下载客户端。

8.3 使用 acl 较为底层的 HTTP 协议库写 HTTP 下载客户端举例

发表时间: 2010-01-13 关键字: 网络协议, FP, F#

在" 使用 acl 库开发一个 HTTP 下载客户端" 文章中介绍利用ACL库中的 HTTP 高级API函数编写HTTP下载客户端的简单的例子，本文介绍一下如何使用稍微底层的API来编写同样功能的例子。在这个例子中，可以看到那些高级API是如何封装底层API的。

请先看一个例子如下：

```
#include "lib_acl.h"
#include "lib_protocol.h"

static void get_url(const char *method, const char *url,
        const char *proxy, const char *dump)
{
        /* 创建 HTTP 请求头 */
        HTTP_HDR_REQ *hdr_req = http_hdr_req_create(url, method, "HTTP/1.1");
        ACL_VSTREAM *stream;  /* 网络连接流 */
        ACL_VSTRING *buf = acl_vstring_alloc(256);  /* 分配内存缓冲区 */
        HTTP_HDR_RES *hdr_res;  /* HTTP 响应头 */
        HTTP_RES *res;  /* HTTP响应体 */
        ACL_FILE *fp = NULL;  /* 转储文件句柄 */
        const char *ptr;
        int   ret;

        /* 输出 HTTP 请求头内容 */

        http_hdr_print(&hdr_req->hdr, "---request hdr---");

        /* 如果设定代理服务器，则连接代理服务器地址，
         * 否则使用 HTTP 请求头里指定的地址
         */

        if (*proxy)
                acl_vstring_strcpy(buf, proxy);
        else
                acl_vstring_strcpy(buf, http_hdr_req_host(hdr_req));
```

```
/* 获得远程 HTTP 服务器的连接地址 */

ptr = acl_vstring_memchr(buf, ':');
if (ptr == NULL)
        acl_vstring_strcat(buf, ":80");
else {
        int   port;
        ptr++;
        port = atoi(ptr);
        if (port <= 0 || port >= 65535) {
                printf("http server's addr(%s) invalid\n", acl_vstring_str(buf));
                acl_vstring_free(buf);
                http_hdr_req_free(hdr_req);
                return;
        }
}

/* 连接远程 http 服务器 */

stream = acl_vstream_connect(acl_vstring_str(buf) /* 服务器地址 */,
                ACL_BLOCKING /* 采用阻塞方式 */,
                10 /* 连接超时时间为 10 秒 */,
                10 /* 网络 IO 操作超时时间为 10 秒 */,
                4096 /* stream 流缓冲区大小为 4096 字节 */);
if (stream == NULL) {
        /* 连接服务器失败 */

        printf("connect addr(%s) error(%s)\n",
                acl_vstring_str(buf), acl_last_serror());
        acl_vstring_free(buf);
        http_hdr_req_free(hdr_req);
        return;
}

/* 构建 HTTP 请求头数据 */
```

```
	http_hdr_build_request(hdr_req, buf);

	/* 向 HTTP 服务器发送请求 */

	ret = acl_vstream_writen(stream, acl_vstring_str(buf), ACL_VSTRING_LEN(buf));
	if (ret == ACL_VSTREAM_EOF) {
		printf("write to server error(%s)\n", acl_last_serror());
		acl_vstream_close(stream);
		acl_vstring_free(buf);
		http_hdr_req_free(hdr_req);
		return;
	}

	/* 创建一个 HTTP 响应头对象 */

	hdr_res = http_hdr_res_new();

	/* 读取 HTTP 服务器响应头*/

	ret = http_hdr_res_get_sync(hdr_res, stream, 10 /* IO 超时时间为 10 秒 */);
	if (ret < 0) {
		printf("get http reply header error(%s)\n", acl_last_serror());
		http_hdr_res_free(hdr_res);
		acl_vstream_close(stream);
		acl_vstring_free(buf);
		http_hdr_req_free(hdr_req);
		return;
	}

	/* 分析HTTP服务器响应头 */

	if (http_hdr_res_parse(hdr_res) < 0) {
		printf("parse http reply header error\n");
		http_hdr_print(&hdr_res->hdr, "--- reply http header ---");
		http_hdr_res_free(hdr_res);
		acl_vstream_close(stream);
		acl_vstring_free(buf);
```

```
        http_hdr_req_free(hdr_req);
        return;
}

/* 如果需要转储至磁盘则需要先打开文件 */

if (dump != NULL) {
        fp = acl_fopen(dump, "w+");
        if (fp == NULL)
                printf("open file(%s) error(%s)\n",
                        dump, acl_last_serror());
}

/* 如果 HTTP 响应没有数据体则仅输出 HTTP 响应头即可 */

if (hdr_res->hdr.content_length == 0
        || (hdr_res->hdr.content_length == -1
                && !hdr_res->hdr.chunked
                && hdr_res->reply_status > 300
                && hdr_res->reply_status < 400))
{
        if (fp)
                http_hdr_fprint(ACL_FSTREAM(fp), &hdr_res->hdr,
                        "--- reply http header ---");
        else
                http_hdr_fprint(ACL_VSTREAM_OUT, &hdr_res->hdr,
                        "--- reply http header ---");
        http_hdr_res_free(hdr_res);
        acl_vstream_close(stream);
        acl_vstring_free(buf);
        http_hdr_req_free(hdr_req);
        return;
}

/* 输出 HTTP 响应头 */

http_hdr_print(&hdr_res->hdr, "--- reply http header ---");
```

```
/* 创建 HTTP 响应体对象 */

res = http_res_new(hdr_res);

/* 如果有数据体则开始读取 HTTP 响应数据体部分 */

while (1) {
        http_off_t  n;
        char  buf2[4096];

        /* 以同步方式读取HTTP响应数据 */

        n = http_res_body_get_sync(res, stream, buf2, sizeof(buf2) - 1);
        if (n <= 0)
                break;

        if (fp) {
                /* 转储至文件中 */

                if (acl_fwrite(buf2, (size_t) n, 1, fp) == (size_t) EOF) {
                        printf("write to dump file(%s) error(%s)\n",
                                dump, acl_last_serror());
                        break;
                }
        } else {
                buf2[n] = 0;
                printf("%s", buf2);
        }
}

if (fp)
        acl_fclose(fp);  /* 关闭转储文件句柄 */
http_res_free(res);  /* 释放 HTTP 响应对象，hdr_res 会在此函数内部自动被释放 */
acl_vstream_close(stream);  /* 关闭网络流 */
acl_vstring_free(buf);  /* 释放内存区 */
http_hdr_req_free(hdr_req);  /* 释放 HTTP 请求头对象 */
```

```
}

static void usage(const char *procname)
{
        printf("usage: %s -h[help] -t method -r url -f dump_file -X proxy_addr\n"
                "example: %s -t GET -r http://www.sina.com.cn/ -f url_dump.txt\n",
                procname, procname);
}

int main(int argc, char *argv[])
{
        int   ch;
        char  url[256], dump[256], proxy[256], method[32];

        acl_init();  /* 初始化 acl 库 */

        ACL_SAFE_STRNCPY(method, "GET", sizeof(method));
        url[0] = 0;
        dump[0] = 0;
        proxy[0] = 0;
        while ((ch = getopt(argc, argv, "hr:t:f:X:")) > 0) {
                switch (ch) {
                case 'h':
                        usage(argv[0]);
                        return (0);
                case 'r':
                        ACL_SAFE_STRNCPY(url, optarg, sizeof(url));
                        break;
                case 't':
                        ACL_SAFE_STRNCPY(method, optarg, sizeof(method));
                        break;
                case 'f':
                        ACL_SAFE_STRNCPY(dump, optarg, sizeof(dump));
                        break;
                case 'X':
                        ACL_SAFE_STRNCPY(proxy, optarg, sizeof(proxy));
                        break;
```

```
                default:
                        break;
                }
        }

        if (url[0] == 0) {
                usage(argv[0]);
                return (0);
        }

        get_url(method, url, proxy, dump);
        return (0);
}
```

可以明显地看出，该例子的实现代码量要比 使用 acl 库开发一个 HTTP 下载客户端 麻烦许多，但它却比较清晰地展示了 HTTP 协议的请求与响应过程。该例子可以在 acl_project/samples/http/get_url1/ 目录下找到。

acl 库下载： http://acl.sourceforge.net/

8.4 使用 acl_cpp 的 HttpServlet 类及服务器框架编写WEB服务器程序

发表时间: 2012-05-21 关键字: web 应用, HTTP应用, 服务器编程, acl_cpp 库

在 《用C++实现类似于JAVA HttpServlet 的编程接口 》 文章中讲了如何用 HttpServlet 等相关类编写 CGI 程序，于是有网友提出了 CGI 程序低效性，不错，确实 CGI 程序的进程开销是比较大的，本文就将说明依然是这些 HTTP 相关的类，如果在使用 acl_cpp/src/master 下的服务器框架类的条件下，可以非常方便地转为服务器程序。现在依然是使用 《用C++实现类似于JAVA HttpServlet 的编程接口 》示例中的 http_servlet 类，只是稍微修改一下 main 函数，就变成下面的情形：

```
////////////////////////////////////////////////////////////////////////

class master_service : public acl::master_proc
{
public:
        master_service() {}
        ~master_service() {}
protected:
        // 基类虚函数，当接收到一个 HTTP 客户端请求时，服务器
        // 框架回调此函数将连接流传入
        virtual void on_accept(acl::socket_stream* stream)
        {
                // HttpServlet 的子类实例
                http_servlet servlet("127.0.0.1:11211");
                servlet.setLocalCharset("gb2312");  // 设置本地字符集
                servlet.doRun(stream);  // 开始处理浏览器请求过程
        }
};

////////////////////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{
        acl::acl_cpp_init();  // 初始化 acl_cpp 库
```

```
        master_service service;  // 半驻留进程池服务类对象

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                // 当在手工调试时一般采用此方式
                printf("listen: 127.0.0.1:8888 ...\r\n");
                service.run_alone("127.0.0.1:8888", NULL, 1);  // 单独运行方式
        }
        else  // 生产环境中以半驻留进程池模式运行
                service.run_daemon(argc, argv);  // acl_master 控制模式运行

        return 0;
}
```

上面的例子是一个结合 HttpServlet 类及 master_service 进程池服务类的 HTTP 服务器程序，关于进程池的例子，可以先结合本人原来写过的基于C语言库 acl 的一篇文章《快速创建你的服务器程序 - - single进程池模型 》。

不仅可以非常容易地将 HttpServlet 写成进程池方式，同时还可以结合 acl_cpp 的线程池框架模板，将 HttpServlet 类实现为半驻留线程池实例，下面就显示了这一过程：

```
class master_threads_test : public acl::master_threads
{
public:
        master_threads_test() {}

        ~master_threads_test() {}

protected:
        // 基类纯虚函数：当客户端连接有数据可读或关闭时回调此函数，返回 true 表示
        // 继续与客户端保持长连接，否则表示需要关闭客户端连接
```

```
        virtual bool thread_on_read(acl::socket_stream* stream)
        {
                // HttpServlet 的子类实例
                http_servlet servlet("127.0.0.1:11211");
                servlet.setLocalCharset("gb2312");  // 设置本地字符集
                servlet.doRun(stream);  // 开始处理浏览器请求过程
        }

        // 基类虚函数：当接收到一个客户端请求时，调用此函数，允许
        // 子类事先对客户端连接进行处理，返回 true 表示继续，否则
        // 要求关闭该客户端连接
        virtual bool thread_on_accept(acl::socket_stream*)
        {
                return true;  // 返回 true 以允许服务器框架继续调用 thread_on_read 过程
        }
};

// 字符串类配置参数项

static char *var_cfg_debug_msg;

static acl::master_str_tbl var_conf_str_tab[] = {
        { "debug_msg", "test_msg", &var_cfg_debug_msg },

        { 0, 0, 0 }
};

// 布尔配置参数项
static int  var_cfg_debug_enable;
static int  var_cfg_keep_alive;
static int  var_cfg_loop;

static acl::master_bool_tbl var_conf_bool_tab[] = {
        { "debug_enable", 1, &var_cfg_debug_enable },
        { "keep_alive", 1, &var_cfg_keep_alive },
        { "loop_read", 1, &var_cfg_loop },
```

```
        { 0, 0, 0 }
};

// 整数配置参数项
static int  var_cfg_io_timeout;

static acl::master_int_tbl var_conf_int_tab[] = {
        { "io_timeout", 120, &var_cfg_io_timeout, 0, 0 },

        { 0, 0 , 0 , 0, 0 }
};

int main(int argc, char* argv[])
{
        master_threads_test mt;  // 半驻留线程池服务器实例

        // 设置配置参数表
        mt.set_cfg_int(var_conf_int_tab);
        mt.set_cfg_int64(NULL);
        mt.set_cfg_str(var_conf_str_tab);
        mt.set_cfg_bool(var_conf_bool_tab);

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                // 当在手工调试时一般采用此方式
                printf("listen: 127.0.0.1:8888\r\n");
                mt.run_alone("127.0.0.1:8888", NULL, 2, 10);  // 单独运行方式
        }
        else  // 生产环境中以半驻留线程池模式运行
                mt.run_daemon(argc, argv);  // acl_master 控制模式运行
        return 0;
}
```

该例子显示了一个基于线程池服务器模型的WEB实例，可以依然使用了文章 《用C++实现类似于JAVA HttpServlet 的编程接口 》示例中的 http_servlet 类，但采用的是由文章 《 开发多线程进程池服务器程序---acl 服务器框架应用 》所介绍的多进程多线程服务器框架模板。

acl_cpp 下载

原文地址

更多文章

9.1 使用 acl 异步库及ICMP协议库编写了一个同时PING多个目标IP的程序

发表时间: 2012-04-29

在 acl 的软件包中，lib_acl 是一个基础的库，另外，还有另外一个库 lib_protocol，这个库中不仅包含了 HTTP 协议的实现，而且还有一个 ICMP PING 协议的实现。前些日子，看到开源中国的微博中提到 fping 的新版本发布了，这是一个可以在一个线程里同时 PING 多个 IP 的程序，心中不尽暗痒，想到自己曾经还专门实现了一个类似的库，并且通用性可能更强，何不写出来给大家分享一下？因此，本篇主要是先以一个简单的小例子入手，演示如何使用 acl 中的这个 ICMP 协议包实现一个可以同时支持 WIN32 和 LINUX 的多目标 PING 程序。

部分 API 接口说明：

```
/**
 * 创建ICMP会话对象
 * @param aio {ACL_AIO*} 如果该项不为空，则内部在通信过程中采用非阻塞模式，
 *  否则采用阻塞模式
 * @param check_tid {int} 是否在校验响应包时检查数据中的线程号字段
 * @return {ICMP_CHAT*} ICMP会话对象句柄
 */
ICMP_API ICMP_CHAT *icmp_chat_create(ACL_AIO *aio, int check_tid);

/**
 * 释放ICMP会话对象
 * @param chat {ICMP_CHAT*} ICMP会话对象句柄
 */
ICMP_API void icmp_chat_free(ICMP_CHAT *chat);

/**
 * 判断当前的ICMP会话对象中所有探测任务是否已经完成
 * @param chat {ICMP_CHAT*} 会话对象句柄
 * @return {int} != 0: 表示完成; 0: 表示未完成
 */
ICMP_API int icmp_chat_finish(ICMP_CHAT *chat);

/**
 * ping 一台主机(内部默认每个探测包长度为64个字节)
 * @param chat {ICMP_CHAT*} 会话对象句柄
```

```
 * @param domain {const char*} 域名标识字符串，可以为空
 * @param ip {const char*} 主机IP地址，不能为空
 * @param npkt {size_t} 对该主机发送的数据包个数
 * @param delay {int} 发送探测数据包的时间间隔(秒)
 * @param timeout {int} 被探测主机的响应包超时时间(秒)
 */
ICMP_API void icmp_ping_one(ICMP_CHAT *chat, const char *domain,
        const char *ip, size_t npkt, int delay, int timeout);

/**
 * 输出当前ICMP的会话状态
 * @param chat {ICMP_CHAT*} 会话对象句柄
 */
ICMP_API void icmp_stat(ICMP_CHAT *chat);

/**
 * 取得当前ICMP会话对象中的当前会话序列号值
 * @param chat {ICMP_CHAT*} 会话对象句柄
 * @return {unsigned short} 会话序列号值
 */
ICMP_API unsigned short icmp_chat_seqno(ICMP_CHAT *chat);
```

下面一个简单的小例子：

```
#include "lib_acl.h"
#include "lib_protocol.h"

static void ping_main_async(void)
{
        int   delay = 1;  /* 发送 PING 数据包的时间间隔(秒) */
        int   npkt = 10;  /* 发送的 PING 数据包个数 */
        ACL_AIO *aio;
        ICMP_CHAT *icmp;
```

```
        /* 创建非阻塞异步通信句柄 */
        aio = acl_aio_create(ACL_EVENT_SELECT);
        acl_aio_set_keep_read(aio, 0);

        /* 创建 ICMP 对象 */
        icmp = icmp_chat_create(aio, 1);

        /* PING www.baidu.com 的一个 IP 地址*/
        icmp_ping_one(icmp, NULL, 61.135.169.115, npkt, delay, 1);
        /* PING www.sina.com.cn 的一个 IP 地址 */
        icmp_ping_one(icmp, NULL, 202.108.33.60, npkt, delay, 1);
        /* PING www.hexun.com 的一个 IP 地址 */
        icmp_ping_one(icmp, NULL, 202.99.16.169, npkt, delay, 1);
        /* PING www.qq.com 的一个 IP 地址 */
        icmp_ping_one(icmp, NULL, 61.135.167.36, npkt, delay, 1);

        while (1) {
                /* 如果 PING 结束，则退出循环 */
                if (icmp_chat_finish(icmp)) {
                        printf("over now!, hosts' size=%d, count=%d\r\n",
                                icmp_chat_size(icmp), icmp_chat_count(icmp));
                        break;
                }

                /* 异步事件循环过程 */
                acl_aio_loop(aio);
        }

        /* 显示 PING 结果 */
        icmp_stat(icmp);;
        /* 释放 ICMP 对象 */
        icmp_chat_free(icmp);

        /* 销毁非阻塞句柄 */
        acl_aio_free(aio);
}
```

可以看出，该例子还是非常简单的，在 acl/samples/ping 下有该例子的完整实现，编译后，运行下面命令：

./ping -n 10 www.baidu.com www.sina.com.cn www.hexun.com www.qq.com

得到如下的输出结果

```
Reply from 202.108.33.60: bytes=56 time=5.427ms TTL=202 icmp_seq=2 status=0
Reply from 61.135.169.105: bytes=64 time=5.975ms TTL=61 icmp_seq=1 status=0
Reply from 61.135.169.125: bytes=64 time=6.394ms TTL=61 icmp_seq=0 status=0
Reply from 61.135.167.36: bytes=64 time=8.147ms TTL=61 icmp_seq=4 status=0
Reply from 202.99.16.169: bytes=64 time=8.532ms TTL=202 icmp_seq=3 status=0
Reply from 61.135.169.105: bytes=64 time=4.879ms TTL=61 icmp_seq=6 status=0
Reply from 202.108.33.60: bytes=56 time=5.313ms TTL=202 icmp_seq=5 status=0
Reply from 61.135.169.125: bytes=64 time=6.695ms TTL=61 icmp_seq=7 status=0
Reply from 61.135.167.36: bytes=64 time=5.963ms TTL=61 icmp_seq=8 status=0

。。。

Ping statistics for 61.135.169.125: www.baidu.com
        Packets: Sent = 10, Received = 10, Lost = 0 (0.00% loss),
Approximate round trip times in milli-seconds:
        Minimum = 5.187 ms, Maximum = 7.143 ms, Average = 6.307 ms
Ping statistics for 61.135.169.105: www.baidu.com
        Packets: Sent = 10, Received = 10, Lost = 0 (0.00% loss),
Approximate round trip times in milli-seconds:
        Minimum = 4.123 ms, Maximum = 8.807 ms, Average = 5.556 ms
Ping statistics for 202.108.33.60: www.sina.com.cn
        Packets: Sent = 10, Received = 10, Lost = 0 (0.00% loss),
Approximate round trip times in milli-seconds:
        Minimum = 4.319 ms, Maximum = 8.073 ms, Average = 5.855 ms
Ping statistics for 202.99.16.169: www.hexun.com
        Packets: Sent = 10, Received = 10, Lost = 0 (0.00% loss),
Approximate round trip times in milli-seconds:
        Minimum = 5.376 ms, Maximum = 8.532 ms, Average = 6.453 ms
Ping statistics for 61.135.167.36: www.qq.com
```

```
        Packets: Sent = 10, Received = 10, Lost = 0 (0.00% loss),
Approximate round trip times in milli-seconds:
        Minimum = 4.292 ms, Maximum = 8.147 ms, Average = 5.948 ms
>>>max pkts: 50
```

acl 库的下载地址：http://http://sourceforge.net/projects/acl/?source=directory

10.1 ACL编程之父子进程机制，父进程守护子进程以防止子进程异常退出

发表时间: 2009-06-07 关键字: 编程, 应用服务器, VC++, Socket, 框架

在WIN32平台进行编程时，经常会遇到工作进程因为程序内部BUG而异常退出的现象，当然为了解决此类问题最好还是找到问题所在并解决它，但如果这类导致程序崩溃的BUG并不是经常出现，只有当某种条件发生时才会有，在我们解决BUG的时间里，为了尽最大可能地为用户提供服务可以采用一种父进程守护机制：当子进程异常退出时，守护父进程可以截获这一消息，并立即重启子进程，这样用户就可以继续使用我们的程序了，当然如果子进程的问题比较严重频繁地 DOWN掉，而父进程却不停地重启子进程的话，势必造成用户机系统资源的大量耗费，那我们的程序就如病毒一样，很快耗尽了用户机资源，所以需要父进程能够智能地控制重启子进程的时间间隔。

本文将给出一个具体的例子（利用ACL库），介绍父、子进程的编程方法。

一、接口介绍

1.1 以守护进程方式运行的接口

创建守护进程的方式非常简单，只需要调用 acl_proctl_deamon_init, acl_proctl_daemon_loop 两个函数即可

接口说明如下：

```
/**
 * 初始化进程控制框架（仅 acl_proctl_start 需要）
 * @param progname {const char*} 控制进程进程名
 */
ACL_API void acl_proctl_deamon_init(const char *progname);

/**
 * 控制进程作为后台服务进程运行，监视所有子进程的运行状态，
 * 如果子进程异常退出则会重启该子进程
 */
ACL_API void acl_proctl_daemon_loop(void);
```

1.2 以命令方式来控制守护进程（守护进程即控制进程的意思）

守护进程启动后，可以以命令方式控制守护进程来启动、停止子进程，或查询显示当前正在运行的子进程。

启动子进程：acl_proctl_start_one

停止子进程：acl_proctl_stop_one

停止所有子进程：acl_proctl_stop_all

查询子进程是否在运行：acl_proctl_probe

查询当前所有在运行的子进程：acl_proctl_list

通过守护进程停止所有子进程且守护进程自身退出：acl_proctl_quit

接口说明如下：

```
/**
 * 以命令方式启动某个子进程
 * @param progname {const char*} 控制进程进程名
 * @param progchild {const char*} 子进程进程名
 * @param argc {int} argv 数组的长度
 * @param argv {char* []} 传递给子进程的参数
 */
ACL_API void acl_proctl_start_one(const char *progname,
    const char *progchild, int argc, char *argv[]);

/**
 * 以命令方式停止某个子进程
 * @param progname {const char*} 控制进程进程名
 * @param progchild {const char*} 子进程进程名
 * @param argc {int} argv 数组的长度
 * @param argv {char* []} 传递给子进程的参数
 */
ACL_API void acl_proctl_stop_one(const char *progname,
    const char *progchild, int argc, char *argv[]);

/**
 * 以命令方式停止所有的子进程
 * @param progname {const char*} 控制进程进程名
 */
ACL_API void acl_proctl_stop_all(const char *progname);

/**
 * 探测某个服务进程是否在运行
 * @param progname {const char*} 控制进程进程名
 * @param progchild {const char*} 子进程进程名
 */
ACL_API void acl_proctl_probe(const char *progname, const char *progchild);

/**
```

```
 * 列出当前所有正在运行的服务进程
 * @param progname {const char*} 控制进程进程名
 */
ACL_API void acl_proctl_list(const char *progname);

/**
 * 以命令方式通知控制进程停止所有的子进程，并在子进程退出后控制进程也自动退出
 * @param progname {const char*} 控制进程进程名
 */
ACL_API void acl_proctl_quit(const char *progname);
```

1.3、子进程编写

子进程编程也比较容易，只需在程序初始化时调用 acl_proctl_child 即可，这样子进程就会在硬盘创建自己的信息并与父进程（即守护进程）建立联系。

接口说明：

```
/**
 * 子进程调用接口，通过此接口与父进程之间建立控制/被控制关系
 * @param progname {const char*} 子进程进程名
 * @param onexit_fn {void (*)(void*)} 如果非空则当子进程退出时调用的回调函数
 * @param arg {void*} onexit_fn 参数之一
 */
ACL_API void acl_proctl_child(const char *progname, void (*onexit_fn)(void *), void *arg);
```

二、例子

2.1、父进程

程序名：acl_project\samples\proctl\proctld.cpp

```
// proctld.cpp : 定义控制台应用程序的入口点。
//
#pragma comment(lib,"ws2_32")
#include "lib_acl.h"
#include <assert.h>

static void init(void)
```

```
{
        acl_init();  // 初始化ACL库
}

static void usage(const char *progname)
{
        printf("usage: %s -h [help] -d [START|STOP|QUIT|LIST|PROBE] -f filepath -a args\r\n",
                        progname);
        getchar();
}

int main(int argc, char *argv[])
{
        char  ch, filepath[256], cmd[256];
        char **child_argv = NULL;
        int   child_argc = 0, i;
        ACL_ARGV *argv_tmp;

        filepath[0] = 0;
        cmd[0] = 0;

        init();

        while ((ch = getopt(argc, argv, "d:f:a:h")) > 0) {
                switch(ch) {
                        case 'd':
                                ACL_SAFE_STRNCPY(cmd, optarg, sizeof(cmd));
                                break;
                        case 'f':
                                ACL_SAFE_STRNCPY(filepath, optarg, sizeof(filepath));
                                break;
                        case 'a':
                                argv_tmp = acl_argv_split(optarg, "|");
                                assert(argv_tmp);
                                child_argc = argv_tmp->argc;
                                child_argv = (char**) acl_mycalloc(child_argc + 1, sizeof(char*
                                for (i = 0; i < child_argc; i++) {
```

```
                                child_argv[i] = acl_mystrdup(argv_tmp->argv[i]);
                        }
                        child_argv[i] = NULL;

                        acl_argv_free(argv_tmp);
                        break;
                case 'h':
                        usage(argv[0]);
                        return (0);
                default:
                        usage(argv[0]);
                        return (0);
        }
}

if (strcasecmp(cmd, "STOP") == 0) {
        // 向守护进程发送消息命令，停止某个子进程或所有的子进程
        if (filepath[0])
                acl_proctl_stop_one(argv[0], filepath, child_argc, child_argv);
        else
                acl_proctl_stop_all(argv[0]);
} else if (strcasecmp(cmd, "START") == 0) {
        if (filepath[0] == 0) {
                usage(argv[0]);
                return (0);
        }
        // 向守护进程发送消息命令，启动某个子进程
        acl_proctl_start_one(argv[0], filepath, child_argc, child_argv);
} else if (strcasecmp(cmd, "QUIT") == 0) {
        // 向守护进程发送消息命令，停止所有的子进程同时守护父进程也退出
        acl_proctl_quit(argv[0]);
} else if (strcasecmp(cmd, "LIST") == 0) {
        // 向守护进程发送消息命令，列出由守护进程管理的正在运行的所有子进程
        acl_proctl_list(argv[0]);
} else if (strcasecmp(cmd, "PROBE") == 0) {
        if (filepath[0] == 0) {
                usage(argv[0]);
```

```
                return (0);
        }
        // 向守护进程发送消息命令，探测某个子进程是否在运行
        acl_proctl_probe(argv[0], filepath);
} else {
        // 父进程以守护进程方式启动
        char  buf[MAX_PATH], logfile[MAX_PATH], *ptr;

        // 获得父进程执行程序所在的磁盘路径
        acl_proctl_daemon_path(buf, sizeof(buf));
        ptr = strrchr(argv[0], '\\');
        if (ptr == NULL)
                ptr = strrchr(argv[0], '/');

        if (ptr == NULL)
                ptr = argv[0];
        else
                ptr++;

        snprintf(logfile, sizeof(logfile), "%s/%s.log", buf, ptr);
        // 打开日志文件
        acl_msg_open(logfile, "daemon");
        // 打开调试信息
        acl_debug_init("all:2");

        // 以服务器模式启动监控进程
        acl_proctl_deamon_init(argv[0]);
        // 父进程作为守护进程启动
        acl_proctl_daemon_loop();
}

if (child_argv) {
        for (i = 0; child_argv[i] != NULL; i++) {
                acl_myfree(child_argv[i]);
        }
        acl_myfree(child_argv);
}
```

```
        return (0);
}
```

2.2、子进程

acl_project\samples\proctl\proctlc.cpp

```
// proctlc.cpp : 定义控制台应用程序的入口点。
//
#pragma comment(lib,"ws2_32")
#include "lib_acl.h"

static void onexit_fn(void *arg acl_unused)
{
        printf("child exit now\r\n");
}

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

        acl_socket_init();
        acl_msg_open("debug.txt", "proctlc");
        acl_msg_info(">>> in child progname(%s), argc=%d\r\n", argv[0], argc);
        if (argc > 1)
                acl_msg_info(">>> in child progname, argv[1]=(%s)\r\n", argv[1]);

        // 子进程启动，同时注册自身信息
        acl_proctl_child(argv[0], onexit_fn, NULL);

        for (i = 0; i < argc; i++) {
                acl_msg_info(">>>argv[%d]:%s\r\n", i, argv[i]);
        }

        i = 0;
        while (1) {
                acl_msg_info("i = %d\r\n", i++);
```

```
                if (i == 5)
                        break;
                else
                        sleep(1);
        }
        return (-1);  // 返回 -1 是为了让父进程继续启动
}
```

2.3、编译、运行

可以打开 acl_project\win32_build\vc\samples\samples_vc2003.sln，编译其中的 proctlc, proctld 两个工程，便会生成两个可执行文件：proctlc.exe(子进程程序），proctld.exe(父进程程序）。

先让父进程以守护进程模式启动 proctld.exe，然后运行 proctld.exe -d START {path}/proctlc.exe 通知父进程启动子进程；可以运行 proctld.exe -d LIST 列出当前正在运行的子进程，运行 proctld.exe -d PROBE {path}/proctld.exe 判断子进程是否在运行，运行 proctld.exe -d STOP {path}/proctld.exe 让守护父进程停止子进程，运行 proctld.exe -d QUID 使守护进程停止所有子进程并自动退出。

另外，从子进程的程序可以看出，每隔5秒子进程就会异常退出，则守护进程便会立即重启该子进程，如果子进程死的过于频繁，则守护进程会延迟重启子进程，以防止太过耗费系统资源。

三、小结

因为有守护进程保护，就不必担心子进程（即你的工作进程）异常崩溃了，这种父子进程模型可以应用于大多数工作子进程偶尔异常崩溃的情形，如果你的程序 BUG太多，每一会儿就崩溃好多次，建议你还是先把主要问题解决后再使用父子进程，毕竟如果你的程序太过脆弱，虽然父进程能不断地重启你的程序，但你还是不能为用户提供正常服务。这种模型适用于在WIN32平台下，你的程序可能写得比较复杂，程序基本上是比较健壮的，只是会因偶尔某些原因而异常退出的情况。

关于ACL库的在线帮助可以参照：http://acl.sourceforge.net/

11.1 ACL_VSTRING - - 字符串操作的法宝

发表时间: 2009-10-19 关键字: 数据结构, C++, C#, C, D语言

一、概述

当我们在使用C + +、JAVA、.NET等面向对象语言甚至象PHP等脚本程序编写字符串处理程序，觉得字符串处理是如此的简单，但当用C编写字符串处理程序时，用得最多可能是诸如：snprintf 、strchr、strcat等标准C函数，但若遇到字符串缓冲区需要动态变化时，就不得不多做一些工作来做内存重新分配的事情，嗯，好繁琐的工作。而POSTFIX代码里有一个非常好用的字符串处理函数库：基于VSTRING结构的字符串处理函数集合，ACL作者将该函数集移植至ACL工程中（重新命名为ACL_VSTRING，没办法，C语言没有象C + +那样的命名空间）并使该库可以运行在多个平台上（UNIX、WIN32等），同时增加了一些常用的功能。本文就介绍一下ACL_VSTRING函数库的使用。

二、常用函数接口

1、创建与释放

```
/**
 * 动态分配一个 ACL_VSTRING 对象并指定内部缓冲区的初始化大小
 * @param len {size_t} 初始时缓冲区大小
 * @return {ACL_VSTRING*} 新分配的 ACL_VSTRING 对象
 */
ACL_API ACL_VSTRING *acl_vstring_alloc(size_t len);

/**
 * 释放由 acl_vstring_alloc 动态分配的 ACL_VSTRING 对象
 * @param vp {ACL_VSTRING*}
 * @return {ACL_VSTRING*} 永远为 NULL，所以不必关心返回值
 */
ACL_API ACL_VSTRING *acl_vstring_free(ACL_VSTRING *vp);
```

2、字符串拷贝及添加

```
/**
 * 拷贝字符串
 * @param vp {ACL_VSTRING*}
```

```
 * @param src {const char*} 源字符串
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_strcpy(ACL_VSTRING *vp, const char *src);

/**
 * 附加拷贝字符串
 * @param vp {ACL_VSTRING*}
 * @param src {const char*} 源字符串
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_strcat(ACL_VSTRING *vp, const char *src);

/**
 * 向缓冲区按格式方式添加数据
 * @param vp {ACL_VSTRING*}
 * @param format {const char*} 格式化字符串
 * @param ... 变参序列
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_sprintf(ACL_VSTRING *vp, const char *format,...);

/**
 * 以附加方式向缓冲区按格式方式添加数据
 * @param vp {ACL_VSTRING*}
 * @param format {const char*} 格式化字符串
 * @param ... 变参序列
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_sprintf_append(ACL_VSTRING *vp, const char *format,...);

/**
 * 按规定格式添加数据
 * @param vp {ACL_VSTRING*}
 * @param format {const char*}
 * @param ap {va_list}
 * @return {ACL_VSTRING*} 与 vp 相同
```

```
 * @see acl_vstring_sprintf
 */
ACL_API ACL_VSTRING *acl_vstring_vsprintf(ACL_VSTRING *vp, const char *format, va_list ap);

/**
 * 按规定格式向尾部添加数据
 * @param vp {ACL_VSTRING*}
 * @param format {const char*}
 * @param ap {va_list}
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_vsprintf_append(ACL_VSTRING *vp, const char *format, va_list a
```

其中，acl_vstring_strcpy 功能与 strcpy 相似，但 acl_vstring_strcpy 不会产生内存溢出问题，因为ACL_VSTRING内存是自动调整的；acl_vstring_strcat 与 strcat 功能相似，但效率却要比 strcat 高，因为 strcat 需要先从字符串头移到字符串尾后才开始添加，并且还可能产生内存越界问题，而 acl_vstring_strcat 内部直接在 ACL_VSTRING 的字符串尾部添加（得益于ACL_VSTRING的内部结构的指针成员），所以效率更高，同时也不存在内存非法越界的问题；acl_vstring_sprintf 与 snprintf 功能类似；acl_vstring_sprintf_append 是对 acl_vstring_sprintf 函数功能的一种扩充；acl_vstring_vsprintf 与 vsnprintf 类似，acl_vstring_vsprintf_append 是对 acl_vstring_vsprintf 的功能扩展。

除了以上专门针对字符串的一些拷贝、添加功能外，ACL_VSTRING还支持非字符串数据的拷贝、添加，接口如下：

```
/**
 * 拷贝内存区
 * @param vp {ACL_VSTRING*}
 * @param src {const char*} 源数据地址
 * @param len {size_t} 源数据长度
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_memcpy(ACL_VSTRING *vp, const char *src, size_t len);

/**
 * 拷贝内存区
 * @param vp {ACL_VSTRING*}
 * @param src {const char*} 源数据地址
```

```
 * @param len {size_t} 源数据长度
 * @return {ACL_VSTRING*} 与 vp 相同
 */
ACL_API ACL_VSTRING *acl_vstring_memcat(ACL_VSTRING *vp, const char *src, size_t len);
```

3、字符串查找

```
/**
 * 查找某个字符
 * @param vp {ACL_VSTRING*}
 * @param ch {int} 要查找的字符
 * @return {char*} 目标字符所在位置的地址，如果未查到则返回 NULL，注：该返回地址是不能
 *  被单独释放的，因为其由 ACL_VSTRING 对象统一进行管理
 */
ACL_API char *acl_vstring_memchr(ACL_VSTRING *vp, int ch);

/**
 * 查找某个字符串，字符串大小写敏感
 * @param vp {ACL_VSTRING*}
 * @param needle {const char*} 要查找的字符
 * @return {char*} 目标字符所在位置的地址，如果未查到则返回 NULL，注：该返回地址是不能
 *  被单独释放的，因为其由 ACL_VSTRING 对象统一进行管理
 */
ACL_API char *acl_vstring_strstr(ACL_VSTRING *vp, const char *needle);

/**
 * 查找某个字符串，忽略字符串大小写
 * @param vp {ACL_VSTRING*}
 * @param needle {const char*} 要查找的字符
 * @return {char*} 目标字符所在位置的地址，如果未查到则返回 NULL，注：该返回地址是不能
 *  被单独释放的，因为其由 ACL_VSTRING 对象统一进行管理
 */
ACL_API char *acl_vstring_strcasestr(ACL_VSTRING *vp, const char *needle);

/**
```

```
 * 从后向前查找字符串，字符串大小写敏感
 * @param vp {ACL_VSTRING*}
 * @param needle {const char*} 要查找的字符
 * @return {char*} 目标字符所在位置的地址，如果未查到则返回 NULL，注：该返回地址是不能
 *  被单独释放的，因为其由 ACL_VSTRING 对象统一进行管理
 */
ACL_API char *acl_vstring_rstrstr(ACL_VSTRING *vp, const char *needle);

/**
 * 从后向前查找字符串，字符串大小写不敏感
 * @param vp {ACL_VSTRING*}
 * @param needle {const char*} 要查找的字符
 * @return {char*} 目标字符所在位置的地址，如果未查到则返回 NULL，注：该返回地址是不能
 *  被单独释放的，因为其由 ACL_VSTRING 对象统一进行管理
 */
ACL_API char *acl_vstring_rstrcasestr(ACL_VSTRING *vp, const char *needle);
```

这些字符串查找函数要比标准C的查找函数功能更为强大。

此外，ACL_VSTRING还提供了一些方便使用的宏，如下：

4、常用宏

```
/**
 * 取得当前 ACL_VSTRING 数据存储地址
 * @param vp {ACL_VSTRING*}
 * @return {char*}
 */
#define acl_vstring_str(vp)             ((char *) (vp)->vbuf.data)

/**
 * 取得当前 ACL_VSTRING 的数据偏移指针位置
 * @param vp {ACL_VSTRING*}
 * @return {char*}
 */
#define acl_vstring_end(vp)             ((char *) (vp)->vbuf.ptr)

/**
```

```
 * 取得当前 ACL_VSTRING 所存储的数据的长度
 * @param vp {ACL_VSTRING*}
 * @return {int}
 */
#define ACL_VSTRING_LEN(vp)             (size_t) ((vp)->vbuf.ptr - (vp)->vbuf.data)

/**
 * 将 ACL_VSTRING 的数据偏移指针位置置 0
 * @param vp {ACL_VSTRING*}
 */
#define ACL_VSTRING_TERMINATE(vp)       { if ((vp)->vbuf.cnt <= 0) \
                                        ACL_VSTRING_SPACE((vp),1); \
                                         *(vp)->vbuf.ptr = 0; }

/**
 * 重置 ACL_VSTRING 内部缓冲区
 * @param vp {ACL_VSTRING*}
 */
#define ACL_VSTRING_RESET(vp)           { (vp)->vbuf.ptr = (vp)->vbuf.data; \
                                          (vp)->vbuf.cnt = (vp)->vbuf.len; }

/**
 * 添加一个字符至 ACL_VSTRING 缓冲区
 * @param vp {ACL_VSTRING*}
 * @param ch {int} 字符
 */
#define ACL_VSTRING_ADDCH(vp, ch)       ACL_VBUF_PUT(&(vp)->vbuf, ch)
```

当然，除了以上常用函数及宏外，还有一些其它的函数及宏，请参考 lib_acl/include/stdlib/acl_vstring.h。

三、举例

```
#include "lib_acl.h"

static void end(void)
{
```

```
#ifdef ACL_MS_WINDOWS
        getchar();
#endif
}

#define STR     acl_vstring_str

// 字符串查找，大小写敏感

static void string_find(ACL_VSTRING *vp, const char *needle)
{
        char *ptr;

        ptr = acl_vstring_strstr(vp, needle);
        if (ptr)
                printf(">>>acl_vstring_strstr: find %s from %s, ptr: %s\r\n", needle, STR(vp),
        else
                printf(">>>acl_vstring_strstr: not find %s from %s\r\n", needle, STR(vp));
}

// 字符串查找，大小写不敏感

static void string_case_find(ACL_VSTRING *vp, const char *needle)
{
        char *ptr;

        ptr = acl_vstring_strcasestr(vp, needle);
        if (ptr)
                printf(">>>acl_vstring_strcasestr: find %s from %s, ptr: %s\r\n", needle, STR(v
        else
                printf(">>>acl_vstring_strcasestr: not find %s from %s\r\n", needle, STR(vp));
}

// 从尾部向前查找字符串，大小写敏感

static void string_rfind(ACL_VSTRING *vp, const char *needle)
{
```

```
        char *ptr;

        ptr = acl_vstring_rstrstr(vp, needle);
        if (ptr)
                printf(">>>acl_vstring_rstrstr: find %s from %s, ptr: %s\r\n", needle, STR(vp),
        else
                printf(">>>acl_vstring_rstrstr: not find %s from %s\r\n", needle, STR(vp));
}

// 从尾部向前查找字符串，大小写不敏感

static void string_case_rfind(ACL_VSTRING *vp, const char *needle)
{
        char *ptr;

        ptr = acl_vstring_rstrcasestr(vp, needle);
        if (ptr)
                printf(">>>acl_vstring_rstrcasestr: find %s from %s, ptr: %s\r\n", needle, STR(
        else
                printf(">>>acl_vstring_rstrcasestr: not find %s from %s\r\n", needle, STR(vp));
}

// 需要查找的字符串表

static char *needle_tab[] = {
        "h",
        "el",
        "o",
        "e",
        "l",
        "lo",
        "he",
        "He",
        "hello",
        "hel",
        "Hel",
        "helo",
```

```
        NULL
};

int main(int argc acl_unused, char *argv[] acl_unused)
{
        ACL_VSTRING *vp = acl_vstring_alloc(256);  // 分配 ACL_VSTRING对象
        char *ptr;
        int   i;

        // 字符串拷贝
        acl_vstring_strcpy(vp, "大家好，中国人民，Hi大家好, Hello World! 中国人民银行!");

        printf(">>>%s\r\n", acl_vstring_str(vp));
        ptr = acl_vstring_strstr(vp, "Hello");  // 查询字符串，区分大小写
        if (ptr)
                printf(">>ok, find it, ptr = %s\r\n", ptr);
        else
                printf(">>error, no find it\r\n");

        ptr = acl_vstring_strcasestr(vp, "WORLD");  // 查询字符串，不区分大小写
        if (ptr)
                printf(">>ok, find it, ptr = %s\r\n", ptr);
        else
                printf(">>error, no find it\r\n");

        ptr = acl_vstring_strstr(vp, "中国");
        if (ptr)
                printf(">>ok, find it, ptr = %s\r\n", ptr);
        else
                printf(">>error, no find it\r\n");

        ptr = acl_vstring_strcasestr(vp, "Hi大家好");
        if (ptr)
                printf(">>ok, find it, ptr = %s\r\n", ptr);
        else
                printf(">>error, no find it\r\n");
```

```
ptr = acl_vstring_memchr(vp, 'W');  // 查询某个字符
if (ptr)
        printf(">>ok, find it, ptr = %s\r\n", ptr);
else
        printf(">>error, no find it\r\n");

acl_vstring_strcpy(vp, "hello");  // 字符串拷贝
ptr = acl_vstring_strstr(vp, "h");  // 字符串查找
if (ptr)
        printf(">>>find it, ptr: %s\r\n", ptr);
else
        printf(">>>not find it\r\n");

printf("\r\n--------------------------------------------------------------\r\n");
for (i = 0; needle_tab[i]; i++) {
        string_rfind(vp, needle_tab[i]);
}

printf("\r\n--------------------------------------------------------------\r\n");
for (i = 0; needle_tab[i]; i++) {
        string_case_rfind(vp, needle_tab[i]);
}

printf("\r\n--------------------------------------------------------------\r\n");
for (i = 0; needle_tab[i]; i++) {
        string_find(vp, needle_tab[i]);
}

printf("\r\n--------------------------------------------------------------\r\n");
for (i = 0; needle_tab[i]; i++) {
        string_case_find(vp, needle_tab[i]);
}

printf("\r\n--------------------------------------------------------------\r\n");
const char *s1 = "hello world", *s2 = "WOrld", *s3 = "world";

printf("strrncasecmp: %s %s %s, n: %d\n", s1,
```

```
                strrncasecmp(s1, s2, strlen(s2)) == 0 ? "==" : "!=", s2, (int) strlen(s2));
        printf("strrncasecmp: %s %s %s, n: %d\n", s1,
                strrncasecmp(s1, s2, strlen(s2) + 1) == 0 ? "==" : "!=", s2, (int) strlen(s2) +

        s1 = "www.hexun.com";
        s2 = ".hexun.com";
        printf("strrncasecmp: %s %s %s, n: %d\n", s1,
                strrncasecmp(s1, s2, strlen(s2)) == 0 ? "==" : "!=", s2, (int) strlen(s2));

        printf("\r\n--------------------------------------------------------\r\n");
        printf("strrncmp: %s %s %s, n: %d\n", s1, strrncmp(s1, s2, strlen(s2)) == 0 ? "==" : "!
        printf("strrncmp: %s %s %s, n: 3\n", s1, strrncmp(s1, s2, 3) == 0 ? "==" : "!=", s2);
        printf("strrncmp: %s %s %s, n: %d\n", s1, strrncmp(s1, s3, strlen(s3)) == 0 ? "==" : "!
        printf("strrncmp: %s %s %s, n: %d\n", s1, strrncmp(s1, s3, strlen(s3) + 1) == 0 ? "=="
                s3, (int) strlen(s3) + 1);

        // 带格式写字符串
        acl_vstring_sprintf(vp, "max long long int: %llu", (unsigned long long int) -1);
        printf("%s\n", acl_vstring_str(vp));

        // 释放 ACL_VSTRING对象

        acl_vstring_free(vp);

        end();
        return (0);
}
```

四、小结

ACL_VSTRING强大易用的功能足可以使你编写常见的字符串应用例子，使你省去了诸如内存重新分配等琐事，一方面提高了开发效率，另一方面减少了出错概率。

ACL库下载地址： http://acl.sourceforge.net

11.2 ACL_ARGV --- 字符串分割动态数组

发表时间: 2009-10-20 关键字: 数据结构, .net

字符串分割是在程序编写过程中经常需要做的事情，如，将字符串：hello world, you are welcome!，进行单词分割，结果希望得到5个单词：hello, world, you, are, welcome。使用 ACL_ARGV 函数便可以非常轻松地实现此功能，如下：

```
// 分割字符串，分割符为 ' ', '\t', ',', '!'

ACL_ARGV *argv = acl_argv_split("hello world, you are welcome!", " \t,!");
ACL_ITER iter;  // 遍历指针

// 遍历分割后的结果

acl_foreach(iter, argv) {
    // 从遍历指针中取出字符串型数据
    const char *ptr =  (const char*) iter.data;

    // 打印单词
    printf(">>>%s\n", ptr);
 }

// 释放内存
acl_argv_free(argv);
```

由此可见使用ACL_ARGV函数分割字符串是如此简单。此外，因为 ACL_ARGV 结构定义符合 ACL_ITER 规范(参见C语言中迭代器的设计与使用)，所以可以直接以 acl_foreach() {} 方式进行遍历。

该例子用到了 lib_acl/include/acl_argv.h 中的两个函数接口，如下：

```
/**
 * 根据源字符串及分隔字符串生成一个字符串动态数组
 * @param str {const char*} 源字符串
 * @param delim {const char*} 分隔字符串
 * @return {ACL_ARGV*}
 */
```

```
ACL_API ACL_ARGV *acl_argv_split(const char *str, const char *delim);

/**
 * 释放字符串动态数组
 * @param argvp {ACL_ARGV*} 字符串动态数组指针
 */
ACL_API ACL_ARGV *acl_argv_free(ACL_ARGV *argvp);
```

此外，acl_argv.h 中还提供了其它方便使用的函数接口，如：

```
/**
 * 向字符串动态数组中添加一至多个字符串，最后一个NULL字符串表示结束
 * @param argvp {ACL_ARGV*} 字符串动态数组指针
 * @param ... 字符串列表，最后一个为NULL，格式如：{s1}, {s2}, ..., NULL
 */
ACL_API void acl_argv_add(ACL_ARGV *argvp,...);

/**
 * 向字符串动态数组中添加字段长度有限制的字符串列表
 * @param argvp {ACL_ARGV*} 字符串动态数组指针
 * @param ... 一组有长度限制的字符串列表，如: {s1}, {len1}, {s2}, {len2}, ... NULL
 */
ACL_API void acl_argv_addn(ACL_ARGV *argvp,...);

/**
 * 根据源字符串及分隔字符串生成一个字符串动态数组，但限定最大分隔次数
 * @param str {const char*} 源字符串
 * @param delim {const char*} 分隔字符串
 * @param n {size_t} 最大分隔次数
 * @return {ACL_ARGV*}
 */
ACL_API ACL_ARGV *acl_argv_splitn(const char *str, const char *delim, size_t n);

/**
 * 源字符串经分隔符分解后，其结果被附加至一个字符串动态数组
```

```
 * @param argvp {ACL_ARGV*} 字符串动态数组指针
 * @param str {const char*} 源字符串
 * @param delim {const char*} 分隔字符串
 * @return {ACL_ARGV*}
 */
ACL_API ACL_ARGV *acl_argv_split_append(ACL_ARGV *argvp, const char *str, const char *delim);
```

这些函数提供了在对字符串进行分割组合时的操作功能。

ACL 库下载位置：http://acl.sourceforge.net/

12.1 配置文件的读取

发表时间: 2009-11-03 关键字: XML

配置文件的读取是程序中必要部分，虽然不算复杂，但如果每次都写配置文件的分析提取代码也是件烦人的事。现在流行的配置文件格式有：ini，xml ，简单name-value对等格式，ACL库中实现了最简单的 name-value对格式的配置文件，该文件格式有点类似于 xinetd.conf 的格式，文件格式如下：

test.cf:

```
service myapp {
    my_addr = 127.0.0.1
    my_port = 80
    my_list = www.test1.com, www.test2.com, www.test3.com, \
              www.test4.com, www.test5.com, www.test6.com
    ...
}
```

其中的 "\" 是连接符，可以把折行的数据连接起来。

下面的例子读取该配置文件并进行解析：

```
static int var_cfg_my_port;

static ACL_CFG_INT_TABLE __conf_int_tab[] = {
  /* 配置项名称，配置项缺省值，存储配置项值的地址，保留字，保留字 */
  { "my_port", 8080, &var_cfg_my_port, 0, 0 },
  { 0, 0 , 0, 0, 0 }
};
```

```
static char *var_cfg_my_addr;
static char *var_cfg_my_list;

static ACL_CFG_STR_TABLE __conf_str_tab[] = {
  /* 配置项名称，配置项缺省值，存储配置项值的地址 */
  { "my_addr", "192.168.0.1", &var_cfg_my_addr },
  { "my_list", "www.test.com", &var_cfg_my_list },
  { 0, 0, 0 }
};

static int var_cfg_my_check;

static ACL_CFG_BOOL_TABLE __conf_bool_tab[] = {
  /* 配置项名称，配置项缺省值，存储配置项值的地址 */
  { "my_check", 0, &var_cfg_my_check },
  { 0, 0, 0 }
};

void test(void)
{
  ACL_XINETD_CFG_PARSER *cfg;  // 配置解析对象

  cfg = acl_xinetd_cfg_load("test.cf");  // 读取并解析配置文件
  acl_xinetd_params_int_table(cfg, __conf_int_tab);  // 读取所有 int 类型的配置项
  acl_xinetd_params_str_table(cfg, __conf_str_tab);  // 读取所有字符串类型的配置项
  acl_xinetd_params_bool_table(cfg, __conf_bool_tab);  // 读取所有 bool 型的配置项

  acl_xinetd_cfg_free(cfg);  // 释放内存
}
```

通过调用 acl_xinetd_params_xxx_table() 函数，直接将配置项的值赋给变量，这样省去了很多麻烦。

13.1 doxygen 帮助手册生成使用心得

发表时间: 2009-06-07 关键字: C#, C++, C, HTML, Windows

程序员在写程序时经常需要在源程序里写一些注释，以备自己或他人以后再看时方便理解，程序的注释格式有多种，而 JavaDoc 的注释格式现在似乎更为流行，不仅 Java 语言在使用它，其它语言如：Javascript, C/C++, Php 等流程语言也越来多地采用 JavaDoc 注释方式。有了注释格式，当然需要有一个注释格式分析工具能够对注释内容进行清晰。Doxygen 便是这个方便的注释分析工具，它可以非常方便地将源程序里的按 JavaDoc 格式注释的内容提取出来，形成各种需要的帮助手册（如：Html格式，Man 格式，RTF格式等），因为doxygen是如此好用，以至于很多开源的软件都在用它，象比较著名的ACE项目的接口帮助手册就是由doxygen生成的，当然还有更多的使用doxygen工具的项目（你可以在 http://www.stack.nl/~dimitri/doxygen/projects.html 上发现如此多的软件项目在使用它）。

因为本人开发了一套基于C语言的支持 Unix/Windows 的网络通信及服务器框架的函数库（名叫：ACL，http://acl.sourceforge.net/），有一些朋友和同事在用它，大家在使用ACL 库时既感到了它的强大、高效，同时又在不断地抱怨接口文档的缺乏及查找的不方便，本人对此也深感痛觉，于是痛下决心，经过相当一段时间的努力，终于在 ACL的所有对外头文件中添加了 JavaDoc 格式注释，于是非常高兴地从 doxygen 的主站(www.doxygen.org)下载了win32平台的安装程序（版本为：1.5.8),然后按使用说明（其实非常简单）将ACL的头文件生成了HTML格式帮助文档，但可惜的是，打开这些HTML页面时却发现是一大堆乱码，原因是doxygen生成的文档默认采用 utf-8编码，而我的文档注释采用的是GB2312的编码，于是修改 doxyen配置，将 INPUT_ENCODING 修改成 GB2312, 将 DOXYFILE_ENCODING 设置成 UTF-8，然后再生成时 doxygen 在分析头文件时报错说无法进行编码转换，既然 doxygen 无法进行编码转换，那只好自己写个转换器了，于是亲手写了个WINDOWS下的编码转换器，将GB2312直接转换成UTF-8, 再由 doxygen 从 UTF-8 的源文件生成UTF-8格式的HTML文档。

至于将ACL的头文件由GB2312转换成UTF-8的过程也是曲折的，开始本人用的转换器的库是 iconv.lib, 转换完发现文件的内容总是不全，不知是不是该库本身的问题还是本人使用不当造成的，于是一不做，二不休，编码转换库也用源代码进行编译运行（好在本人曾经做过邮件系统，在字符集编码方面不乏源代码，随手粘来一段代码贴在程序里），OK，转换成功，所有的头文件的中文注释均由原来的GB2312转换成 UTF-8了。当然在用 doxygen 生成HTML文档时，别忘了将 INPUT_ENCODING 设置成空，意思是说无需 doxygen进行字符集编码转换（通过跟踪 doxygen 的源代码，发现只要将 INPUT_ENCODING 设置为空，则它不会对注释文档进行字符集编码转换 - - - 这样也好，免得它总是转错，其实它的源程序里是用 iconv.lib 进行转换的，不知转换不成功的原因是否也与 iconv.lib 有关）。

其实， doxygen 在对中文的字符集转换时所出现的问题在以前的旧的版本并未出现过（那是因为原来还比较土，根本就不进行转换，哈，看来少做有时也有好处，至少还能用），只是在一些比较新的版本（包括当前的1.5.8）都存在这类问题（将中文由GB2312转换成UTF-8时会失败），本人本想采用旧的 doxygen 生成HTML帮助文档，但最终经不住新版的诱惑：更好的CSS控制，更优美的外观，更方便的索引方式，等等。于是自写了个工具软件，进行了字符集转换。（如哪位朋友需要，可以发邮件至 zhengshuxin@hexun.com 索取这个小软件，等本人将其做完善后再放在网上）。

字符集转码工作做完了，但另一个问题又出现了，本人的函数库都是由C语言完成的，因为没有象C + +式

的命名空间，所以为了防止与他人的代码冲突，所有的函数名前都加了前缀 acl_, 所有的结构前都加了前缀 ACL_, 这样问题就来了，本来用 doxygen 生成的HTML文档是有按字母索引、排序功能的，但因为我的函数前都有 acl_ 字样，这样所有的函数都堆在一个 a 开头的排序里，于是更加郁闷。怎么办？得，还得自己写工具进行转换，第一步：先用工具将所有中文注释的头文件转换成UTF-8格式的；第二步：将函数名前的前缀 acl_ 转换成后缀 _acl(如：原来的一个函数 acl_vstream_gets 前缀变后缀后应为 vstream_gets_acl, 结构：ACL_VSTREAM 则变为 VSTREAM_ACL)；第三步：用 doxygen 将第二步生成的头文件生成HTML文档；第四步：用自写工具将函数名、结构类型中的字符串由后缀转前缀（如：vstream_gets_acl->acl_vstream_gets, VSTREAM_ACL->ACL_VSTREAM)，这样就一切OK了，最终生成的HTML帮助文档都是UTF-8格式的，可以按字母序进行排列的函数接口帮助文档了。（题外话：象顶顶大名的 glib 库，虽然文档注释格式是 JavaDoc 格式的，但它并没有使用 doxygen 工具生成帮助手册，估计原因可能和我一样，它是一个C库，为了避免与别人冲突，都有前缀g-, 如果用 doxygen就无法按首字母进行排序查看，所以 glib 的开发小组自己开发文档生成工具）。

您可以参看 http://acl.sourceforge.net/ 看一下ACL库的在线帮助文档。

此外，此文档还有一个小小的缺陷，那就是在搜索栏里，如果你要查acl_vstream_xxx 类的函数时，需要输入 vstream_xxx, 而不是 acl_vstream_xxx, 并且查得的结果也是 vstream_xxx_acl 格式（当你点其链接时的结果是正确的），主要是因为 doxgen 生成ACL帮助文档的全文索引时的源文档都是 vstream_xxx_acl - - - 即以 _acl/_ACL 为后缀的，目前主要是对其索引文档还不太熟悉，得有时间再完善这点吧。当然，希望 doxygen 能将我所遇到的问题都解决，这样也就不再需要那个转换工具了。

14.1 C语言中也可以方便地进行遍历

发表时间: 2009-08-16 关键字: C#, C, C++, D语言, Xcode

请先看一个例子，如下：

```
void test()
{
    ACL_HTABLE *table = acl_htable_create(10, 0);  /* 创建哈希表 */
    ACL_HTABLE_ITER iter;  /* 哈希表的遍历变量 */
    char *value, key[32];
    int   i;

    for (i = 0; i < 100; i++) {
        value = (char*) acl_mystrdup("value");
        snprintf(key, sizeof(key), "key:%d", i);
        (void) acl_htable_enter(table, key, value);  /* 向哈希表中添加元素 */
    }

    /* 遍历哈希表中的所有元素 */
    acl_htable_foreach(iter, table) {
        printf("%s=%s\n", acl_htable_iter_key(iter), acl_htable_iter_value(iter));
    }

    /* 释放哈希表表 */
    acl_htable_free(table, acl_myfree_fn);
}
```

哈，用C语言也可以实现其它编程语言里的迭代器，而且用法也异常简单，虽然它没有C++中的功能强大，但却比较实用，而且操作手法有点象D、JAVA的遍历方式。下面再请看一个利用ACL里的先进先出队列的例子：

```
void test()
{
    ACL_FIFO fifo;
```

```
    ACL_FIFO_ITER iter;
    char *data;
    int   i;

    acl_fifo_init(&fifo);  /* 初始化队列对象 */

    for (i = 0; i < 10; i++) {
        data = acl_mymalloc(32);
        snprintf(data, 32, "data: %d", i);
        acl_fifo_push(&fifo, data);  /* 向队列中添加元素 */
    }

    /* 反向遍历队列中的所有元素 */
    acl_fifo_foreach_reverse(iter, &fifo) {
        printf("%s\n", (char*) iter.ptr->data);  /* 打印元素字符串 */
    }

    while (1) {
        /* 弹出队列中的所有元素 */
        data = acl_fifo_pop(&fifo);
        if (data == NULL)
            break;
    }
}
```

上面这个是ACL里反向遍历先进先出队列的例子。

我们在使用 C++ 里的迭代器里，其实基本上都是在用C++标准模板库的算法而已，这些常用算法无非也就是动态数组、哈希表、队列、堆栈等数据结构而已，而现在C++的使用替代器的过程未免过于烦琐（模板是由C++发扬光大，但现在搞的也太罗嗦了，不知C++标准委员会里的那些老头整天都在忙些什么，呵呵），我还是比较喜欢Java和D语言里的使用方式。

上面的两个例子的遍历过程其实是由宏来实现的，效率不会有问题，但写法也未免有些拙劣，呵呵，不过实用即可。具体实现方式请参考ACL里的头文件：lib_acl/include/stdlib/ 下的 acl_htable.h, acl_fifo.h. ACL库下载位置：https://acl.sourceforge.net

14.2 小谈C语言中常见数据类型在32及64位机上的使用

发表时间: 2009-09-17 关键字: C, C++, C#, D语言, Linux

1、概述

C语言有一些非常基本的数据类型，正是这些基本类型让我们可以延伸了无限的用户自定义类型，本文主要介绍了 int, size_t, time_t, long, long long int 等基本数据类型在Linux32 及 Linux64 的使用情况。表面看上去，这些类型确实太基础太简单，似乎没啥可讲的，实事似乎也是如此，用过C的对这些都已经非常熟悉了，这还用讲？在PC　64位机器出来之前，我们确实不用太关注这些，因为在32位机上编程，似乎很少出现过什么问题，但64位机出来了，象Linux 也支持64位机器，问题就来了，为什么？因为它们的长度发生了变化，而我们的程序也就有可能需要改变一下。

2、举例

先举个例子，如下：

```
#include <stdio.h>
#include <stdlib.h>

static void get_length(size_t *size)
{
    if (size)
        *size = 100;
}

static void test(void)
{
    char *buf = strdup("hello world");
    int  n;

    printf("buf: %s\n", buf);
    get_length((size_t*) &n);

    printf("buf: %s, n: %d\n",  buf, n);
    free(buf);
}
```

```
int main(int argc, char *argv[])
{
    test();
    return (0);
}
```

首先将此程序在32位机的 Linux 上运行一下，如下：

buf: hello world

buf: hello world, n: 100

OK，如我们所料，一切正常。

然后再将些程序在64位机的 Linux 上运行一下，如下：

buf: hello world

buf: (null), n: 100

奇怪的现象出来了，怎么printf出的结果为空呢？晕菜，为啥经过 get_length()/1 后世界改变了，buf 的内容没有了，被指向一个空指针，而 buf 明明是还没有被释放呀。赶快用 valgrind 检查一下，

valgrind --tool=memcheck --leak-check=yes --show-reachable=yes ./a.out

“2 bytes in 1 blocks are definitely lost in loss record 1 of 1”，说有块内存未被释放，而在 test() 后面确实释放过 buf 呀，谁偷偷地给释放了而没有告诉俺？更晕菜，难道是 libc 的问题？再用 valgrind 在32位机检查一下，一切OK，没有出现64位机上的错误提示，说明内存确实由 test() 中的 free(buf) 释放了。

正当对此问题百思不解时，忽然想到一个问题 int * 至 size_t* 类型转换会不会有问题？因为 size_t 在32位机上是4字节，而在64位机上是8字节，int在32位及64位机上都是4字节，嗯，问题就在于此，再回头仔细看看上述代码，在 test() 中将 &n 由 int * 强制转换成 size_t *, 这样可以避免编译警告，但在 get_length()/1 中呢？它是不会知道 size_t *size 中 size 所指空间是4字节的，而依然当8字节对待，这样在对 *size = 100 进行赋值时就发生了改变，size 所指的8字节空间发生改变，而实际应该只改变4字节才是，这便是问题的关键所在，所以在遇到此类问题时，一定得要注意基本类型在不同机器上的空间大小了。

3、小结

以上的例子只是一个简单的例子，也许还容易看得出来，当我们的项目比较大时，这种错误可能会偶尔发生一下，那可能就是致命的了，因为有时它并不会导致程序 异常退出产生core文件，但却会改变我们的运行结果，本人就因此问题调试了两天多的时间才找到原因，另外，即使因此问题产生了 core 文件，你会发现用 gdb 调试该 core 时根本找不到原因所在。

下面列出一些基本类型在32位及64位机上的大小差异

	int	long	size_t	time_t	long long int
32位机器	4字节	4字节	4字节	4字节	8字节
64位机器	4字节	8字节	8字节	8字节	8字节

在写跨平台的程序时，一定要注意这些基本类型的长度大小。

15.1 ACL缓存开发

发表时间: 2009-10-18 关键字: Cache, 数据结构, 多线程, D语言

在编写高效的程序时，内存缓存有时是非常有用的，提到缓存，大家可能会很容易想到可以使用哈希表这种最常用的方式来缓存内存对象，但哈希的实现代码一般不具备两项功能：缓存过期时间、缓存数量限制，如果要增加对此二项功能的支持，一般需要增加辅助的链表结构。如果使用ACL里的 ACL_CACHE，则在高效缓存的前提下支持这两项功能。

下面是ACL_CACHE的数据结构及常用的接口调用：

一、数据结构及常用接口说明

1、数据结构定义

```
/**
 * 缓冲池对象结构定义
 */
typedef struct ACL_CACHE {
        ACL_HTABLE *table;      /**< 哈希表 */
        ACL_RING ring;          /**< 将被删除的对象的数据链表 */
        int   max_size;                 /**< 缓存池容量大小限制值 */
        int   size;                             /**< 当前缓存池中的缓存对象个数 */
        int   timeout;                  /**< 每个缓存对象的生存时长(秒) */

        /**< 释放用户动态对象的释放回调函数 */
        void  (*free_fn)(const ACL_CACHE_INFO*, void *);
        acl_pthread_mutex_t lock;       /**< 缓存池锁 */
        ACL_SLICE *slice;               /**< 内存切片对象 */

        /* for acl_iterator */

        /* 取迭代器头函数 */
        const void *(*iter_head)(ACL_ITER*, struct ACL_CACHE*);
        /* 取迭代器下一个函数 */
        const void *(*iter_next)(ACL_ITER*, struct ACL_CACHE*);
        /* 取迭代器尾函数 */
```

```
        const void *(*iter_tail)(ACL_ITER*, struct ACL_CACHE*);
        /* 取迭代器上一个函数 */
        const void *(*iter_prev)(ACL_ITER*, struct ACL_CACHE*);
        /* 取迭代器关联的当前容器成员结构对象 */
        const ACL_CACHE_INFO *(*iter_info)(ACL_ITER*, struct ACL_CACHE*);
} ACL_CACHE;

/**
 * 缓存池中存储的缓存对象
 */
typedef struct ACL_CACHE_INFO {
        char *key;              /**< 健值 */
        void *value;            /**< 用户动态对象 */
        int   nrefer;           /**< 引用计数 */
        time_t when_timeout;    /**< 过期时间截 */
        ACL_RING entry;         /**< 内部数据链成员 */
} ACL_CACHE_INFO;
```

2、创建与释放接口

```
/**
 * 创建一个缓存池，并设置每个缓存对象的最大缓存时长及该缓存池的空间容量限制
 * @param max_size {int} 该缓存池的容量限制
 * @param timeout {int} 每个缓存对象的缓存时长
 * @param free_fn {void (*)(void*)} 用户级的释放缓存对象的函数
 * @return {ACL_CACHE*} 缓存池对象句柄
 */
ACL_API ACL_CACHE *acl_cache_create(int max_size, int timeout,
        void (*free_fn)(const ACL_CACHE_INFO*, void*));

/**
 * 释放一个缓存池，并自动调用 acl_cache_create()/3 中的释放函数释放缓存对象
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 */
ACL_API void acl_cache_free(ACL_CACHE *cache);
```

3、增、删、查接口

```
/**
 * 向缓存池中添加被缓存的对象
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 * @param key {const char*} 缓存对象的健值
 * @param value {void*} 动态缓存对象
 * @return {ACL_CACHE_INFO*} 缓存对象所依附的结构对象，其中的 value 与用户的对象相同，
 *   如果返回 NULL 则表示添加失败，失败原因为：缓存池太大溢出或相同健值的对象存在
 *   且引用计数非0；如果返回非 NULL 则表示添加成功，如果对同一健值的重复添加，会用
 *   新的数据替换旧的数据，且旧数据调用释放函数进行释放
 */
ACL_API ACL_CACHE_INFO *acl_cache_enter(ACL_CACHE *cache, const char *key, void *value);

/**
 * 从缓存池中查找某个被缓存的对象
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 * @param key {const char*} 查询健
 * @return {void*} 被缓存的用户对象的地址，为NULL时表示未找到
 */
ACL_API void *acl_cache_find(ACL_CACHE *cache, const char *key);

/**
 * 从缓存池中查找某个被缓存的对象所依附的缓存信息对象
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 * @param key {const char*} 查询健
 * @return {ACL_CACHE_INFO*} 缓存信息对象地址，为NULL时表示未找到
 */

/**
 * 从缓存池中删除某个缓存对象
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 * @param key {const char*} 健值
 * @return {int} 0: 表示删除成功; -1: 表示该对象的引用计数非0或该对象不存在
```

```
 */
ACL_API int acl_cache_delete2(ACL_CACHE *cache, const char *key);
```

4、同步互斥接口

```
/**
 * 加锁缓存池对象，在多线程时用
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 */
ACL_API void acl_cache_lock(ACL_CACHE *cache);

/**
 * 解锁缓存池对象，在多线程时用
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 */
ACL_API void acl_cache_unlock(ACL_CACHE *cache);
```

5、遍历接口

```
/**
 * 遍历缓存中的所有对象
 * @param cache {ACL_CACHE*} 缓存池对象句柄
 * @param walk_fn {void (*)(ACL_CACHE_INFO*, void*)} 遍历回调函数
 * @param arg {void *} walk_fn()/2 中的第二个参数
 */
ACL_API void acl_cache_walk(ACL_CACHE *cache, void (*walk_fn)(ACL_CACHE_INFO *, void *), void *
```

当然，因为 ACL_CACHE 符合 ACL_ITER 通用迭代器(C语言中迭代器的设计与使用)的规则要求，所以也可以采用ACL通用迭代方式遍历ACL_CACHE内部缓存对象，如下：

```
 typedef struct {
  char  name[32];
  char  dummy[32];
} MY_DAT;
```

```
static free_mydat_fn(const ACL_CACHE_INFO *info, void *arg)
{
  MY_DAT *mydat = (MY_DAT*) mydat;

  acl_myfree(mydat);
}

void test(void)
{
  ACL_CACHE *cache;
  ACL_ITER iter;
  MY_DAT *mydat;
  char  key[32];

  /* 创建缓存对象句柄 */
  cache = acl_cache_create(100, 60, free_mydat_fn);

  /* 向缓存中添加缓存数据 */
  for (i = 0; i < 10; i++) {
    mydat = (MY_DAT*) acl_mymalloc(sizeof(MY_DAT));
    snprintf(key, sizeof(key), "key:%d", i);
    snprintf(mydat->name, sizeof(mydat->name), "name: %d", i);
    (void) acl_cache_enter(cache, key, mydat);
  }

  /* 遍历所有缓存数据 */
  acl_foreach(iter, cache) {
    const MY_DAT *mydat = (const MY_DAT*) iter.data;
    printf(">>>name: %s\n", mydat->name);
  }

  /* 释放缓存句柄并清除缓存数据 */
   acl_cache_free(cache);
}
```

除了以上几个常用接口外，ACL_CACHE 模块还提供了其它方便使用的接口调用方式，参见: lib_acl/include/stdlib/acl_cache.h 头文件说明。

二、举例

下面是一个完整使用 ACL_CACHE 的例子：

```
#include "lib_acl.h"
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* 用户自定义数据结构 */
typedef struct {
        char *buf;
        int   len;
} MYOBJ;

/**
 * 释放用户数据回调数据
 * @param info {const ACL_CACHE_INFO*} 用户缓存数据所依附的ACL_CACHE 的某个数据对象
 * @param arg {void*} 用户数据对象以 void * 表示
 */
static void free_fn(const ACL_CACHE_INFO *info, void *arg)
{
        MYOBJ *o = (MYOBJ*) arg;

        printf("%s: when_timeout: %ld, now: %ld, len: %d, deleted\n",
                info->key, (long) info->when_timeout, (long) time(NULL), o->len);

        acl_myfree(o->buf);
        acl_myfree(o);
}

/**
 * 创建一个新的用户数据对象
```

```
 * @param len {int} MYOBJ.buf 的长度
 * @return {MYOBJ*}
static MYOBJ *myobj_new(int len)
{
        MYOBJ *o = (MYOBJ*) acl_mymalloc(sizeof(MYOBJ));

        o->buf = (char*) acl_mymalloc(len <= 0 ? 100 : len);
        o->len = len;
        return (o);
}

/**
 * 遍历数据数据缓存中每一个数据对象的回调函数
 * @param info {ACL_CACHE_INFO*}
 * @arg {void*} 用户数据对象
 */
static void walk_fn(ACL_CACHE_INFO *info, void *arg)
{
        MYOBJ *o = (MYOBJ*) info->value;

       assert(info->value == arg);
        printf("%s: size: %d; when_timeout: %ld\n", info->key, o->len, (long) info->when_timeou
}

static void usage(const char *procname)
{
        printf("usage: %s -h [help] -n max_size -t timeout\n", procname);
}

int main(int argc, char *argv[])
{
        int   i, n = 100, ch, timeout = 1;
        ACL_CACHE_INFO *info;
        ACL_CACHE *cache;
        char  key[32];
        MYOBJ *o;
```

```
while ((ch = getopt(argc, argv, "hn:t:")) > 0) {
        switch (ch) {
                case 'h':
                        usage(argv[0]);
                        exit (0);
                case 'n':
                        n = atoi(optarg);
                        break;
                case 't':
                        timeout = atoi(optarg);
                        break;
                default:
                        break;
        }
}

/* 创建缓存句柄 */
cache = acl_cache_create(n, timeout, free_fn);

/* 向缓存中添加用户缓存数据 */
for (i = 0; i < n + 5; i++) {
        o = myobj_new(i + 1);
        snprintf(key, sizeof(key), "key(%d)", i);
        assert(acl_cache_enter(cache, key, o));
        printf("add one: %s\n", key);
        sleep(1);
}

printf("\nfirst walk cache, cache size: %d\n", acl_cache_size(cache));

/* 遍历所有缓存数据 */
acl_cache_walk(cache, walk_fn, NULL);
printf("\nfirst call acl_cache_timeout, size: %d\n", acl_cache_size(cache));

/* 过期的缓存数据被自动删除 */
acl_cache_timeout(cache);
printf(">>>after first acl_cache_timeout, second walk cache, cache's size: %d\n", acl_c
```

```
/* 遍历所有缓存数据 */
acl_cache_walk(cache, walk_fn, NULL);

printf("\n");
i = 0;

/* 休眠以使有些缓存数据过期 */
while (i++ < 5) {
        printf("slee one second, i=%d\n", i);
        sleep(1);
}

printf("\nsecond call acl_cache_timeout, size: %d\n", acl_cache_size(cache));

/* 过期的缓存数据被自动删除 */
acl_cache_timeout(cache);
printf(">>>after second acl_cache_timeout, third walk_cache, cache's size: %d\n", acl_c

/* 遍历所有缓存数据 */
acl_cache_walk(cache, walk_fn, NULL);

/* 查询缓存对象 */
o = (MYOBJ*) acl_cache_find(cache, "key(5)");
if (o == NULL)
        printf("\n>>>key(5) not exist\n");
else
        printf("\n>>>key(5): len: %d\n", o->len);

/* 定位用户缓存数据所依附的缓存对象 */
info = acl_cache_locate(cache, "key(11)");
if (info == NULL)
        printf("\n>>>key(11) not exist\n");
else {
        o = (MYOBJ*) info->value;
        printf("\n>>>key(11): len: %d, when_timeout: %ld\n", o->len, (long) info->when_
}
```

```
        printf("\nfree cache, size: %d\n", acl_cache_size(cache));

        /* 释放缓存并清除所有用户缓存数据 */
        acl_cache_free(cache);
        return (0);
}
```

ACL 库下载位置：http://acl.sourceforge.net/

16.1 创建多级目录

发表时间: 2009-10-21 关键字: Unix, C, C++, C#, .net

在UNIX平台下有一个创建目录的API接口 mkdir()，该函数使用比较简单，但如果需要创建多级目录，则操作起来就稍微麻烦些，也有一些公开的代码实现了创建多级目录的功能，而很多都是采用递归调用 mkdir() 创建目录的方式。在 Postfix 代码也有一段代码用于创建多级目录，它的优点是迭代方式创建多级目录，而不是递归调用 mkdir()，于是乎ACL作者将其改造了一下（主要增加了针对WIN32的支持），加入ACL库中。接口比较简单，如下：

```
/**
 * 功能: 创建多级目录结构
 *  如创建 "/tmp/dir1/dir2" (for unix) 或 "C:\\test\\test1\\test2" (for win32)
 * @param path: 一级或多级目录路径
 * @param perms: 创建权限(如: 0755, 0777, 0644 ...)
 * @return == 0: OK;  == -1, Err
 */
ACL_API int acl_make_dirs(const char *path, int perms);
```

该函数接口在UNIX、WIN32平台下完全一致，只是在WIN32平台下，其中的参数 perms 是无效的，此外，还有一个差别就是在WIN32平台下各级目录分隔符可以为 "/" 或 "\"，而在UNIX平台下分隔符则只能为 "/"。

以下是一个简单的例子：

```
#include "lib_acl.h"
#include <stdio.h>

static void test_mkdirs(void)
{
    const char *path = "path1\\path2\\path3\\path4";
    int   perms = 0700, ret;

    ret = acl_make_dirs(path, perms);
    if (ret < 0) {
        printf("create path(%s) error(%s)\n", path, acl_last_serror());
    } else {
```

```
        printf("create path(%s) ok\n", path);
    }
}

int main(int argc acl_unused, char *argv[] acl_unused)
{
    test_mkdirs();
    return (0);
}
```

acl 库下载： http://acl.sourceforge.net/

16.2 ACL_VSTREAM 进行文件读写

发表时间: 2009-11-04 关键字: acl_vstream, 文件读写

一、概述

ACL_VSTREAM 函数集不仅可以用在网络IO中，同时也可以用在文件IO中，而且读写所调用的函数接口一致，这样就给使用者带了方便。当然在UNIX类系统中，文件IO与网络IO的读写函数是一致的，因为UNIX比较好的将设备等都抽象为文件描述符，但如果您使用WINDOWS的API进行网络IO及文件IO编程时，不幸的是，微软提供了不同的调用方式，而且差别较大，这也可以看出似乎微软在统一规划上面比较“欠缺”，经常性地推出新的规范与标准，而新的总是不顾旧的，呵呵。

好了，现在具体地介绍一下如何使用 ACL_VSTREAM 进行跨平台式文件IO操作。ACL_VSTREAM 针对不同操作系统平台的API进行了封装，从而给使用者提供了统一的使用方式。

二、接口说明

1、文件打开与关闭

```
/**
 * 打开一个文件的数据流
 * @param path {const char*} 文件名
 * @param oflags {unsigned int} 标志位, We're assuming that O_RDONLY: 0x0000,
 *  O_WRONLY: 0x0001, O_RDWR: 0x0002, O_APPEND: 0x0008, O_CREAT: 0x0100,
 *  O_TRUNC: 0x0200, O_EXCL: 0x0400; just for win32, O_TEXT: 0x4000,
 *  O_BINARY: 0x8000, O_RAW: O_BINARY, O_SEQUENTIAL: 0x0020, O_RANDOM: 0x0010.
 * @param mode {int} 打开文件句柄时的模式(如: 0600)
 * @param buflen {size_t} 内置缓冲区的大小
 * @return ret {ACL_VSTREAM*}, ret== NULL: 出错, ret != NULL: OK
 */
ACL_API ACL_VSTREAM *acl_vstream_fopen(const char *path,
                                unsigned int oflags,
                                int mode,
                                size_t buflen);

/**
 * 释放一个数据流的内存空间并关闭其所携带的 socket 描述符
```

```
 * @param stream {ACL_VSTREAM*} 数据流
 */
ACL_API int acl_vstream_close(ACL_VSTREAM *stream);
#define acl_vstream_fclose      acl_vstream_close
```

17.1 acl 之 xml 流解析器

发表时间: 2010-07-24 关键字: XML, FP, XSL, JavaScript, 编程

现在 XML 解析器比较多，其实本没有必要在ACL中添加新的XML解析库，象JAVA、PHP、C#的开发者大都习惯于使用XML数据，因为他们有比较好用的XML解析库，而C/C++的程序员可能使用非XML数据的情形比较多，数据格式也各式各样。当然，如果C/C++程序员使用XML数据也有一些成熟的XML解析库，最丰富的解析库之一如libxml2，比较小型的如tinyxml等，这些库功能都比较丰富，但对流的支持可能有些局限性，象libxml2，接口使用起来还比较复杂，也不太容易掌握。经过再三考虑，决定在ACL中添加XML解析库，希望能满足如下功能：

1、接口丰富而简单（看似是矛盾的，呵呵）

2、很好地支持流的处理（可以支持同步网络流及异步网络流）

3、可以比较容易进行查询、添加、删除、遍历等操作，最好能象JS一样进行操作。

经过几个周末努力，终于算是完成了XML解析库并添加进ACL中。为了很好支持异步流，该XML库采用了有限状态机的方式（效率虽可能不是最好，但也不会差），下面对主要的编程接口进行介绍。

一、API 介绍

1、XML容器对象的创建、XML解析树的生成以及XML容器对象的释放

```
/**
* 创建一个 xml 容器对象
* @return {ACL_XML*} 新创建的 xml 对象
*/
ACL_API ACL_XML *acl_xml_alloc(void);
```

当进行XML解析前，首先必须调用 acl_xml_alloc 创建一个XML容易对象。

```
/**
 * 解析 xml 数据，并持续地自动生成 xml 结点树
 * @param xml {ACL_XML*} xml 对象
```

```
 * @param data {const char*} 以 '\0' 结尾的数据字符串，可以是完整的 xml 数据;
 *  也可以是不完整的 xml 数据，允许循环调用此函数，将不完整数据持续地输入
 */
ACL_API void acl_xml_parse(ACL_XML *xml, const char *data);
```

将 xml 源数据输入，通过 acl_xml_parser进行解析，因为该函数支持数据状态缓冲（采用有限状态机的好处），所以可以非常容易地支持流数据。

```
/**
 * 释放一个 xml 对象，同时释放该对象里容纳的所有 xml 结点
 * @param xml {ACL_XML*} xml 对象
 */
ACL_API int acl_xml_free(ACL_XML *xml);
```

最后需要调用 acl_xml_free 来释放 XML 容易对象。

2、XML数据结点的查询

XML数据结点的查询非常方便，有点类似于JAVASCRIPT中的方式（这也是作者的本意，如果象libxml2那样估计别人用起来就会比较麻烦）。在ACL的XML库里提供了非常丰富的查询方式，如下：

```
/**
 * 从 xml 对象中获得所有的与所给标签名相同的 xml 结点的集合
 * @param xml {ACL_XML*} xml 对象
 * @param tag {const char*} 标签名称
 * @return {ACL_ARRAY*} 符合条件的 xml 结点集合，存于 动态数组中，若返回 NULL 则
 *  表示没有符合条件的 xml 结点
 */
ACL_API ACL_ARRAY *acl_xml_getElementsByTagName(ACL_XML *xml, const char *tag);
```

```
/**
 * 从 xml 对象中获得所有给定属性名及属性值的 xml 结点元素集合
 * @param xml {ACL_XML*} xml 对象
```

```
 * @param name {const char*} 属性名
 * @param value {const char*} 属性值
 * @return {ACL_ARRAY*} 符合条件的 xml 结点集合，存于 动态数组中，若返回 NULL 则
 *  表示没有符合条件的 xml 结点
 */
ACL_API ACL_ARRAY *acl_xml_getElementsByAttr(ACL_XML *xml,
        const char *name, const char *value);
```

```
/**
 * 从 xml 对象中获得所有的与给定属性名 name 的属性值相同的 xml 结点元素集合
 * @param xml {ACL_XML*} xml 对象
 * @param value {const char*} 属性名为 name 的属性值
 * @return {ACL_ARRAY*} 符合条件的 xml 结点集合，存于 动态数组中，若返回 NULL 则
 *  表示没有符合条件的 xml 结点
 */
ACL_API ACL_ARRAY *acl_xml_getElementsByName(ACL_XML *xml, const char *value);
```

以上三个函数的返回结果都是一个数组对象（ACL中数组对象库的使用请参考ACL中的 acl_array.h)，这些数组元素的类型为ACL_XML_NODE，提取这些数组元素的方式如下：

```
void test(const char *xml_data)
{
  ACL_XML *xml = acl_xml_create();
  ACL_ARRAY *a;
  ACL_ITER iter;

  acl_xml_parse(xml, xml_data);

  a = acl_xml_getElementsByTagName(xml, "user");
  if (a) {
    acl_foreach(iter, a) {
      ACL_XML_NODE *node = (ACL_XML_NODE*) iter.data;
```

```
      printf("tagname: %s\n", acl_vstring_str(node->ltag));
    }
    /* 释放数组对象 */
    acl_xml_free_array(a);
  }

  acl_xml_free(xml);
}
```

最后，得注意使用 acl_xml_free_array 来释放数组结果集。

另外，还有一些函数用来获得单个结果，如下：

```
/**
 * 从 xml 对象中获得指定 id 值的 xml 结点元素的某个属性对象
 * @param xml {ACL_XML*} xml 对象
 * @param id {const char*} id 值
 * @return {ACL_XML_ATTR*} 某 xml 结点的某个属性对象，若返回 NULL 则表示
 *  没有符合条件的属性
 */
ACL_API ACL_XML_ATTR *acl_xml_getAttrById(ACL_XML *xml, const char *id);

/**
 * 从 xml 对象中获得指定 id 值的 xml 结点元素的某个属性值
 * @param xml {ACL_XML*} xml 对象
 * @param id {const char*} id 值
 * @return {const char*} 某 xml 结点的某个属性值，若返回 NULL 则表示没有符合
 *  条件的属性
 */
ACL_API const char *acl_xml_getAttrValueById(ACL_XML *xml, const char *id);

/**
 * 从 xml 对象中获得指定 id 值的 xml 结点元素
 * @param xml {ACL_XML*} xml 对象
 * @param id {const char*} id 值
 * @return {ACL_XML_NODE*} xml 结点元素，若返回 NULL 则表示没有符合
```

```
 *  条件的 xml 结点
 */
ACL_API ACL_XML_NODE *acl_xml_getElementById(ACL_XML *xml, const char *id);

/**
 * 从 xml 结点中获得指定属性名的属性对象
 * @param node {ACL_XML_NODE*} xml 结点
 * @param name {const char*} 属性名称
 * @return {ACL_XML_ATTR*} 属性对象，为空表示不存在
 */
ACL_API ACL_XML_ATTR *acl_xml_getElementAttr(ACL_XML_NODE *node, const char *name);

/**
 * 从 xml 结点中获得指定属性名的属性值
 * @param node {ACL_XML_NODE*} xml 结点
 * @param name {const char*} 属性名称
 * @return {const char*} 属性值，为空表示不存在
 */
ACL_API const char *acl_xml_getElementAttrVal(ACL_XML_NODE *node, const char *name);
```

这些查询接口也非常类似于JAVASCRIPT的查询方式。因为查询结果具有唯一性，所以仅返回一个结果。

3、XML树结点的遍历

XML树结点的遍历遵循ACL框架库中定义的统一遍历方式，所以遍历XML树也非常容易，示例如下：

```
ACL_XML *xml;
ACL_XML_NODE *node;
ACL_ITER iter;

/*......*/
/* 假设  XML  解析树已经创建完毕 */

/* 遍历整个XML容器对象的所有数据结点 */
acl_foreach(iter, xml) {
  node = (ACL_XML_NODE*) iter.data;
```

```
  printf("tagname: %s, text: %s\n", acl_vstring_str(node->ltag),
    acl_vstring_str(node->text));
}

/* 遍历某个XML结点的下一级子结点 */
node = acl_xml_getElementById(xml, "id_test");
if (node) {
  acl_foreach(iter, node) {
    ACL_XML_NODE *node2 = (ACL_XML_NODE*) iter.data;
    printf("tagname: %s, text: %s\n", acl_vstring_str(node->ltag),
        acl_vstring_str(node->text));
  }
}
```

4、其它函数

```
/**
 * 从 xml 结点删除某个属性对象，如果该属性为 id 属性，则同时会从 xml->id_table 中删除
 * @param node {ACL_XML_NODE*} xml 结点
 * @param name {const char*} 属性名称
 * @return {int} 0 表示删除成功，-1: 表示删除失败(有可能是该属性不存在)
 */
ACL_API int acl_xml_removeElementAttr(ACL_XML_NODE *node, const char *name);

/**
 * 给 xml 结点添加属性，如果该属性名已存在，则用新的属性值替换其属性值，否则
 * 创建并添加新的属性对象
 * @param node {ACL_XML_NODE*} xml 结点
 * @param name {const char*} 属性名称
 * @param value {const char*} 属性值
 * @return {ACL_XML_ATTR*} 返回该属性对象(有可能是原来的，也有可能是新的)
 */
ACL_API ACL_XML_ATTR *acl_xml_addElementAttr(ACL_XML_NODE *node,
        const char *name, const char *value);
```

```
/**
 * 将 xml 对象转储于指定流中
 * @param xml {ACL_XML*} xml 对象
 * @param fp {ACL_VSTREAM*} 流对象
 */
ACL_API void acl_xml_dump(ACL_XML *xml, ACL_VSTREAM *fp);
```

当然，还有更多的函数未列出，以上仅可能是用户在使用XML库过程中常用的函数接口。

二、举例

下面举一个完整的例子以结束本文。

```
#include "lib_acl.h"

#define STR     acl_vstring_str

static void parse_xml(int once)
{
        ACL_XML *xml = acl_xml_alloc();
        const char *data =
                "<?xml version=\"1.0\"?>\r\n"
                "<?xml-stylesheet type=\"text/xsl\"\r\n"
                "\thref=\"http://docbook.sourceforge.net/release/xsl/current/manpages/docbook.x
                "\t<!DOCTYPE refentry PUBLIC \"-//OASIS//DTD DocBook XML V4.1.2//EN\"\r\n"
                "\t\"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd\" [\r\n"
                "       <!ENTITY xmllint \"<command>xmllint</command>\">\r\n"
                "]>\r\n"
                "<root name='root' id='root_id_1'>\r\n"
                "       <user name='user1' value='zsx1' id='id1'> user zsx1 </user>\r\n"
                "       <user name='user2' value='zsx2' id='id2'> user zsx2 \r\n"
                "               <age year='1972'>my age</age>\r\n"
                "       </user>\r\n"
                "       <user name='user3' value='zsx3' id='id3'> user zsx3 </user>\r\n"
                "</root>\r\n"
```

```
        "<root name='root' id='root_id_2'>\r\n"
        "       <user name='user1' value='zsx1' id='id1'> user zsx1 </user>\r\n"
        "       <user name='user2' value='zsx2' id='id2'> user zsx2 \r\n"
        "               <!-- date should be the date of the latest change or the releas
        "               <age year='1972'>my age</age>\r\n"
        "       </user>\r\n"
        "       <user name='user3' value='zsx3' id='id3'> user zsx3 </user>\r\n"
        "</root>\r\n"
        "<root name = 'root2' id = 'root_id_3'>\r\n"
        "       <user name = 'user2_1' value = 'zsx2_1' id = 'id2_1'> user zsx2_1 </use
        "       <user name = 'user2_2' value = 'zsx2_2' id = 'id2_2'> user zsx2_2 </use
        "       <user name = 'user2_3' value = 'zsx2_3' id = 'id2_3'> user zsx2_3 \r\n'
        "               <age year = '1978' month = '12' day = '11'> bao bao </age>\r\n'
        "       </user>\r\n"
        "       <!-- still a bit buggy output, will talk to docbook-xsl upstream to fi
        "       <!-- <releaseinfo>This is release 0.5 of the xmllint Manual.</releasein
        "       <!-- <edition>0.5</edition> -->\r\n"
        "       <user name = 'user2_2' value = 'zsx2_2' id = 'id2_4'> user zsx2_2 </use
        "</root>\r\n";
const char *ptr;
ACL_ITER iter1;
int   i, total, left;
ACL_ARRAY *a;
ACL_XML_NODE *pnode;

ptr = data;
if (once) {
        /* 一次性地分析完整 xml 数据 */
        acl_xml_parse(xml, ptr);
} else {
        /* 每次仅输入一个字节来分析 xml 数据 */
        while (*ptr != 0) {
                char  ch2[2];

                ch2[0] = *ptr;
                ch2[1] = 0;
                acl_xml_parse(xml, ch2);
```

```
                ptr++;
        }
}

if (acl_xml_is_complete(xml, "root")) {
        printf(">> Yes, the xml complete\n");
} else {
        printf(">> No, the xml not complete\n");
}

total = xml->node_cnt;

/* 遍历根结点的一级子结点 */
acl_foreach(iter1, xml->root) {
        ACL_ITER iter2;

        ACL_XML_NODE *node = (ACL_XML_NODE*) iter1.data;
        printf("tag> %s, text: %s\n", STR(node->ltag), STR(node->text));

        /* 遍历一级子结点的二级子结点 */
        acl_foreach(iter2, node) {
                ACL_ITER iter3;
                ACL_XML_NODE *node2 = (ACL_XML_NODE*) iter2.data;

                printf("\ttag> %s, text: %s\n", STR(node2->ltag), STR(node2->text));

                /* 遍历二级子结点的属性 */
                acl_foreach(iter3, node2->attr_list) {
                        ACL_XML_ATTR *attr = (ACL_XML_ATTR*) iter3.data;
                        printf("\t\tattr> %s: %s\n", STR(attr->name), STR(attr->value));
                }
        }
}

printf("-----------------------------------------------------\n");

/* 从根结点开始遍历 xml 对象的所有结点 */
```

```
acl_foreach(iter1, xml) {
        ACL_ITER iter2;
        ACL_XML_NODE *node = (ACL_XML_NODE*) iter1.data;

        for (i = 1; i < node->depth; i++) {
                printf("\t");
        }

        printf("tag> %s, text: %s\n", STR(node->ltag), STR(node->text));

        /* 遍历 xml 结点的属性 */
        acl_foreach(iter2, node->attr_list) {
                ACL_XML_ATTR *attr = (ACL_XML_ATTR*) iter2.data;

                for (i = 1; i < node->depth; i++) {
                        printf("\t");
                }

                printf("\tattr> %s: %s\n", STR(attr->name), STR(attr->value));
        }
}

/* 根据标签名获得 xml 结点集合 */

printf("--------- acl_xml_getElementsByTagName ----------\n");
a = acl_xml_getElementsByTagName(xml, "user");
if (a) {
        /* 遍历结果集 */
        acl_foreach(iter1, a) {
                ACL_XML_NODE *node = (ACL_XML_NODE*) iter1.data;
                printf("tag> %s, text: %s\n", STR(node->ltag), STR(node->text));
        }
        /* 释放数组对象 */
        acl_xml_free_array(a);
}
```

```
/* 查询属性名为 name, 属性值为 user2_1 的所有 xml 结点的集合 */

printf("--------- acl_xml_getElementsByName ------------\n");
a = acl_xml_getElementsByName(xml, "user2_1");
if (a) {
        /* 遍历结果集 */
        acl_foreach(iter1, a) {
                ACL_XML_NODE *node = (ACL_XML_NODE*) iter1.data;
                printf("tag> %s, text: %s\n", STR(node->ltag), STR(node->text));
        }
        /* 释放数组对象 */
        acl_xml_free_array(a);
}

/* 查询属性名为 id, 属性值为 id2_2 的所有 xml 结点集合 */
printf("----------- acl_xml_getElementById ------------\n");
pnode = acl_xml_getElementById(xml, "id2_2");
if (pnode) {
        printf("tag> %s, text: %s\n", STR(pnode->ltag), STR(pnode->text));
        /* 遍历该 xml 结点的属性 */
        acl_foreach(iter1, pnode->attr_list) {
                ACL_XML_ATTR *attr = (ACL_XML_ATTR*) iter1.data;
                printf("\tattr_name: %s, attr_value: %s\n",
                        STR(attr->name), STR(attr->value));
        }

        pnode = acl_xml_node_next(pnode);
        printf("----------------- the id2_2's next node is --------------------\n");
        if (pnode) {
                printf("-------------- walk node ------------------\n");
                /* 遍历该 xml 结点的属性 */
                acl_foreach(iter1, pnode->attr_list) {
                        ACL_XML_ATTR *attr = (ACL_XML_ATTR*) iter1.data;
                        printf("\tattr_name: %s, attr_value: %s\n",
                                STR(attr->name), STR(attr->value));
                }
```

```
        } else {
                printf("-------------- null node ------------------\n");
        }
}

pnode = acl_xml_getElementById(xml, "id2_3");
if (pnode) {
        int   ndel = 0, node_cnt;

        /* 删除该结点及其子结点 */
        printf(">>>before delete %s, total: %d\n", STR(pnode->ltag), xml->node_cnt);
        ndel = acl_xml_node_delete(pnode);
        node_cnt = xml->node_cnt;
        printf(">>>after delete id2_3(%d deleted), total: %d\n", ndel, node_cnt);
}

acl_foreach(iter1, xml) {
        ACL_XML_NODE *node = (ACL_XML_NODE*) iter1.data;
        printf(">>tag: %s\n", STR(node->ltag));
}

pnode = acl_xml_getElementById(xml, "id2_3");
if (pnode) {
        printf("-------------- walk %s node ------------------\n", STR(pnode->ltag));
        /* 遍历该 xml 结点的属性 */
        acl_foreach(iter1, pnode->attr_list) {
                ACL_XML_ATTR *attr = (ACL_XML_ATTR*) iter1.data;
                printf("\tattr_name: %s, attr_value: %s\n",
                                STR(attr->name), STR(attr->value));
        }
} else {
        printf("---- the id2_3 be deleted----\n");
}

/* 释放 xml 对象 */
left = acl_xml_free(xml);
```

```
        printf("free all node ok, total(%d), left is: %d\n", total, left);
}

static void parse_xml_file(const char *filepath, int once)
{
        char *data = acl_vstream_loadfile(filepath);
        ACL_VSTREAM *fp;
        char *ptr;
        ACL_XML *xml;
        struct timeval  begin, end;

        if (data == NULL)
                return;

        gettimeofday(&begin, NULL);

        /* 创建 xml 对象 */
        xml = acl_xml_alloc();

        ptr = data;

        if (once) {
                /* 一次性地分析完整 xml 数据 */
                acl_xml_parse(xml, ptr);
        } else {
                /* 每次仅输入一个字节来分析 xml 数据 */
                while (*ptr) {
                        char  ch2[2];

                        ch2[0] = *ptr;
                        ch2[1] = 0;
                        acl_xml_parse(xml, ch2);
                        ptr++;
                }
        }

        gettimeofday(&end, NULL);
```

```
        printf("------ok, time: %ld seconds, %ld microseconds -------\r\n",
                end.tv_sec - begin.tv_sec, end.tv_usec - begin.tv_usec);


        fp = acl_vstream_fopen("dump.txt", O_RDWR | O_CREAT | O_TRUNC, 0600, 4096);

        /* 将 xml 对象转储至指定流中 */
        acl_xml_dump(xml, fp);

        acl_vstream_fclose(fp);
        acl_xml_free(xml);
        acl_myfree(data);
}

static void usage(const char *procname)
{
        printf("usage: %s -h[help] -f {xml_file} -s[parse once]\n", procname);
}

int main(int argc, char *argv[])
{
        int   ch, once = 0;
        char  filepath[256];

        acl_init();
        snprintf(filepath, sizeof(filepath), "xmlcatalog_man.xml");

        while ((ch = getopt(argc, argv, "hf:s")) > 0) {
                switch (ch) {
                case 'h':
                        usage(argv[0]);
                        return (0);
                case 'f':
                        snprintf(filepath, sizeof(filepath), "%s", optarg);
                        break;
                case 's':
```

```
                        once = 1;
                        break;
                default:
                        break;
                }
        }

        parse_xml(once);
        parse_xml_file(filepath, once);

#ifdef  ACL_MS_WINDOWS
        printf("ok, enter any key to exit ...\n");
        getchar();
#endif
        return 0;
}
```

18.1 斑马线免费企业邮件系统到底动了谁的奶酪？

发表时间: 2012-05-25 关键字: 免费邮件系统, 免费邮件反垃圾, 免费企业IM, 免费企业日历, 邮件厂商

昨天和几位朋友聊天，其中有位朋友说他们公司针对国内中小企业，推出了免费自建企业邮局的安装包（叫斑马邮），说是500用户以下永久免费使用，不仅提供功能丰富的邮件系统功能，而且还提供企业即时通信、企业日历、企业考勤、邮件反垃圾网关等。我的第一感受就是：你们到底想要干嘛？以前只是听说过好多面向个人服务的软件免费的情形，象周鸿祎，举着一杆免费的大旗，终于使 360 的客户端软件占据了中国的软件安全市场，象曾经的瑞星杀毒、金山杀毒都基本不见了踪迹，甚至于一些国外的品牌都失去了大部分的个人市场。而面向中小企业，向中小企业提供办公软件的厂商都是要向企业收取软件授权费的，象用友、金蝶财务软件提供者，象易邮、快客、安宁、方标提供独立邮件系统的厂商，他们的客户都会要向他们支付价值不菲的软件费用。在国内市场，几乎很少见免费提供企业软件的厂商，毕竟企业是有付费习惯的。

于是我问这位朋友，你们为什么要这么做？不怕与众多提供邮件软件的厂商为敌吗？难道不怕大家群起而攻之？

这位朋友说，他认为单纯靠软件收费的模式正逐渐丧失其市场依据，尤其是企业市场更是如此，原来大家都喜欢盗版微软的 Exchange 做企业的邮件系统，不仅承担着法律风险，而且还面临数据丢失的风险（毕竟微软是不给盗版用户提供服务的，再加上盗版软件还存在一定安全隐患）。但这些企业级的办公软件的价格又实在是比较高，动辄就是几万、几十万甚至上百万，这哪是一般中小企业所能承受的起的；然后他说，既然已经做好了给中小企业提供免费邮件系统的准备，就不怕与邮件厂商为敌，不怕他们说免费邮件系统不好用、有问题，因为他们的技术团队都是业内顶级的邮件专家（很多都是来自于国内的专业厂商，如 263, TOM 等），他们相信他们做的产品是经得起用户考验的，并且他们的团队还一直在不断完善他们的全套产品线。

这位朋友似乎很激动，接着说，他们不仅提供与商业邮件系统相同功能的免费邮件系统，同时还提供了邮件反垃圾网关，企业即时通信，企业日历，以及手机邮件，下一步还有电子考勤系统免费提供，这是一个企业公办产品系列。用过盗版 Exchange 的用户一定不会忘记被大量垃圾邮件骚扰的烦恼，但是要买商用的反垃圾网关，又是一笔不小的开销，更甭说再买腾讯的企业 RTX（那就更贵了）。所以他们就是为了给广大中小企业免费提供全套企业在线办公解决方案。

于是我又反问：既然你们想免费了，为啥还要定个 500 用户以下才免费呢？难道有什么不轨企图？他说：为什么非要定 500 用户以下免费使用邮件系统？一是这个数值已经完全覆盖了中国的绝大部分中小企业，500 用户以上的企业，要么是大型企业（他们不太可能会用免费的），要么就是一些邮件运营商，所以限制到 500 用户，就是为了避免这些所谓邮件运营商使用他们的软件提供在线运营服务，向最终中小企业用户收费。因为他们也有一块收费业务，是专门为中小企业提供在线网络办公服务的，而且也是收费的；而且他们老板的理念是“软件免费，服务收费”。

这我就不明白了：你们一会儿免费，一会儿收费，这有啥区别？听他解释了半天，我总算大体明白，原来是所谓免费，就是指用户可以从他们的网站（www.banma.cn）上下载邮件服务端安装包（说是同时集成了邮件

反垃圾，企业日历，企业IM等功能），将这些软件包安装在用户自己的计算机服务器上，因为这些软件使用的是用户自己的硬件资源，所以是免费的；所谓收费，是因为他们自己也提供邮件系统等办公软件的在线运营服务，使用的硬件资源（如：服务器，带宽等）是他们的，所以是收费的。

所谓无利不起早，我问他：你们都免费提供邮件系统等办公软件了，那你们的获得点是什么？这不是在赔本赚吆喝吗？他到是很干脆，我们就是为了积累更多的企业客户，让众多中小企业客户知道他们，为将来他们可以为中小企业提供更多付费服务造成一定知名度，并且如果只是提供软件，象微软的 office、Exchange、操作系统、数据库等的商业模式已经过时，下面就是看谁愿意先去打破这个瓶瓶罐罐了。所以，他们愿意做为中国中小企业市场提供免费软件的第一个敢吃螃蟹的人，而且如果免费软件做的好，则有可能他们目前收费的企业邮局服务也有可能会免费。

听这位朋友海阔天空地谈了这么多后，我在想：他们向企业免费提供邮件系统，到底是动了谁的奶酪?目前来看，首先肯定不是 263 这类的邮局运营厂商，那一定是提供邮件系统的软件厂商。这块蛋糕有多大？有多少中小企业愿意使用他们的免费软件系统？等等问题，现在都很难回答。但他们既然下决心这样做了，还是打心里佩服他们老板的决心，他们老板想的肯定不是只想做个现代活雷锋那么简单，呵呵。

18.2 乱炖现在流行应用之产品设计

发表时间: 2012-05-27 关键字: 产品设计, 用户体验

写这么个题目，并非是为了哗众取宠，实在是受不了现在网络很多流行产品的设计，所以就想乱炖它们一通，也许会得罪一些人，但请这些产品的设计人员也认真想一下：有则改之，无则加勉嘛。

一、邮箱产品

自 Ajax 的JS 技术流行，各大提供邮箱服务的厂商都打破了以往的界面分为三帧的设计，采用一些复杂的JS脚本将界面做成了一个整体，当然，这是一种很大的进步，但进步的背后确实还存在很多不足：

1、忽略了导航栏的价值

如网易的邮箱，当右边的内容区域比较长时，自然就会出现滚动条，但当用户拉着滚动条使页面向下滚动时，左侧的导航栏竟然跟着滚动，这样很快左侧的导航栏就会消失了，这岂不是失去了左侧导航的价值？QQ邮箱这点做的到是比较好，无论右边的滚动条如何下向滚动，左侧的导航栏依然存在；但QQ邮箱也一个问题，就是右侧的顶部的导航栏居然也会消失！当然这点网易邮箱又做的好些，右侧顶部的导航栏不会消失。所以不仅要问，二位能不能互补一下呀？

回过头来再看看 gmail 的导航栏，您会发现 gmail 就存在上面所谓两家邮箱的问题：各部分的导航部分不会随滚动条的滚动而消失。

2、没有文件夹功能

这主要是针对 gmail 的，gmail 确实有很多创新：标签功能、邮件会话功能、快速回复功能等。但我就不明白，您为啥非要去掉文件夹功能？虽然标签功能很好用，可以帮助用户很好地进行分类，但当用户的邮件比较多时，在收件箱里有这么多的邮件，看起来实是累眼睛，因为所有折邮件都在收件箱里，以至于我经常会漏掉一些重要邮件。

3、笨重的 JS 库

现在 Ajax 是如此之流行，使很多 WEBMAIL 都成了客户端软件似的模仿体，这必然导致了浏览器端的 JS 比较笨重，尤其是以 263 的邮箱为代表（hotmail也不咋地），为了进入 webmail 收看哪怕只是一封邮件，用户也不得不等待较长的 JS 加载时间和初始化时间，按说象 263 这样号称专门做邮箱的厂商不应如此吧（其它的象QQ邮箱，网易邮箱，YAHOO邮箱都没有象263的这么慢）。可能原因还是因为 263 用了 EXTJS 框架库的原因，当然 263 的技术和产品人员也许会说：只是第一次加载时慢，以后每次再进入时就不会慢了，因为本地给缓存了。那我不禁要问：您为啥不把第一次加载也弄的快些呢？

当然，还有一个 hotmail 不得不说你一下，您可是最早提供 webmail 功能的免费邮箱了，现在竟然成了 msn 的一个附属物了，为何现在还这么慢呢？

4、顶部区域的空间浪费

现在多数邮件厂商的 WEBMAIL 在顶部都有一个很大的空闲区域，岂不知现在大部分笔记本的屏幕都是宽屏的了，虽然宽度空间富裕了，但高度空间更加宝贵了，这些邮箱厂商，您们能不能给我们省点空间呀？（当然 263 的 webmail在这方面做的是比较好的，其它的都是浪费严重）

5、撰写邮件时的 suggest 功能

当用户在写一封邮件时，如果在地址栏输入收件人的前几个字母时，多数邮箱都会有自动提示功能，这确实是webmail的一个进步，但263邮箱，您为啥仅提示通讯录里的邮件地址，那常用联系人但并未加在通讯录里就没有提示了？

还有，现在所有的邮件系统都存在一个问题，当用户使用 outlook 等客户端发邮件时，收件人就不能自动加在常用联系人里了，难道是因为大部分邮件厂商的 MTA/SMTP 模块都是采用 POSTFIX 无法修改的原因？

二、SNS 产品

1、Facebook

现在 facebook 是上市了，而且市值很高，有事没事您就给俺的 hotmail 里发一些垃圾信息，也真够累的，象什么某某和某某成为好友之类的信息经常要发到用户的邮箱里，当用户取消订阅这类消息后，没过多久不知为何又开始发了，所以笔者想问，facebook您烦不烦呀，您不知道俺在国内上不了啊？要不您把所有的功能链接都做成 https 的，这样俺用代理软件还能上。

2、Google+

很喜欢 google+ 在很多方面的创新，干净清新的界面风格，丰富强大的权限管理，尤其是全程 HTTPS，这些都吸引了不少用户，但我就不明白，不什么不提供收藏功能呢？难道 Google+ 的产品经理认为上面的所有信息流都将成为垃圾信息不值得用户去收藏吗？还有，取消了翻页的功能，这确实不错，但为什么您就不提供返回顶部的功能呢？您可知道，当我滚动鼠标了看了上百条信息时，再想返回顶部是多么地困难，原来总是比较笨重地再往回滚动鼠标，后来终于可以按左上角的“首页”可以直接返回，但您就不知道当我需要返回第一条信息时的鼠标还在右下角吗？为了实现这一功能，鼠标可谓是在屏幕上走了最长的距离（从右下角划到左上角）；当用户浏览信息时如果突然想发布一条信息则就更加曲折了。

还有，google+ 界面上的 JS 确实很炫，但为啥在俺的机器运行起来总感觉这么慢呢（比第一版效率似乎高点了）？难道我需要把现在手上的电脑扔掉不成？

3、新浪微博

新浪微博的新版界面提供了上--左中右的设计 ，但您的左栏感觉形同虚设，因为这栏也会随用户往下看信息而消失，这个毛病和前端某些邮箱产品一样，失去了导航栏的作用（再回头看看 google+ 人家的左侧导航栏

咋就不会消失呢？）。版面设计上依然存在着顶部空间过度浪费的问题，似乎新浪的产品经理们的显示器要么不是宽屏的，要么就是比较大，他们很难体会我们这些用宽屏笔记本的“痛苦”。

安全功能也是新浪微博的一个问题，几月前本人的帐号密码被盗了，费了半天劲才找回来（又上传身份证，又找好友，好一翻折腾）。因为我的微博账号用的就是新浪的邮箱账号，所以通过邮箱找回微博密码根本就行不通，本人在新浪微博的密码找回那块可算是吃尽了苦头。既然微博和邮箱都用新浪的存在此类风险，新浪微博您为啥在当初俺注册时非要说为了安全需要，建议使用新浪邮箱呢？难道光为了积累用户而不管用户信息安全了？

4、腾讯社区

腾讯的很多产品都做的不错，而且做得很细，但很多功能一用就提示需要收费，实在不敢用，腾讯现在都完全钻到钱眼里去了，难怪它现在的年收入这么高。

5、人人网

曾经的猫扑、校内、山寨版开心网的结合体，除了山寨，弄些擦边的新闻，咱还能做些啥正经的东西？

三、客户端产品

1、QQ 聊天软件

腾讯赖以做大的最基础产品线，没有QQ聊天软件，可以说就没有今天腾讯年近300亿的收入规模。现在QQ可以说是目前国内最强大的聊天软件，用户体验做得非常细，经历了曾经的界面丑陋、功能臃肿，到今天的产品形态实在是投入了巨大的精力与财力，产品好有人用，但现在腾讯似乎已经分不清哪些该收费、哪些该免费了，因为从界面上集成了各种形式的收费陷阱，自从本人无奈将手机与QQ号绑定后，就不敢乱QQ上的功能了，免得哪天被扣了费自己还不知道呢。

2012版本的QQ客户端推出了多人聊天时的多标签功能，但怎么用都感觉不好用，建议QQ的产品经理还是多看看现在浏览器的多标签浏览是怎么设计的吧；2012版本QQ在好友的数据挖掘方面实在是“牛”，确实帮用户挖掘出不少“好友”来，但感觉有点挖掘过度了，据说已经停止了此类过度的数据挖掘。

2、360 安全软件

打着一杆“安全”的大旗，360 现在俨然已经成了 PC终端、移动终端的“守护神”，至少周鸿祎自己是这样标榜自己的。以“安全”的名义打击同行或非同行的软件产品实在使人怀疑其“安全”的动机。本人曾经安装过 360 的安全卫士，但偶然感染了病毒，这个小卫士不仅杀不了，而且自己的病毒库也无法升级了（因为360的升级界面是内嵌的IE控件，而本人的IE确实是被木马给感染了），后来只得重装了XP系统，既然无用，索性就不再安装假卫士了。

3、腾讯电脑管家

腾讯在这个知识产权薄弱的国家里，肆无忌惮地山寨别人的产品，当360安全产品比较火时，腾讯也推出了自己的安全产品，为了吸引用户使用电脑管家，竟然靠QQ社区积分方式来刺激，我就不明白了，电脑管家到底是安全产品呢还是娱乐产品？难怪360说你是靠垄断手段的不正当竞争。

四、搜索

1、百度搜索

就国内搜索市场份额来讲，就只讲百度搜索吧，这个号称最懂中文的国内第一大搜索引擎，除了过度的商业化开发，本人包括周围的朋友都说，真要在上面搜索信息时，很难搜到真正有价值的东西。比如：当输入一词语，但中间用空格分开搜时，百度的搜索结果中只有这个词的单个字的搜索结果，它就不会再多一个将这些空格去掉后的搜索条件？同样查询条件，谷歌就比较智能，最后给出了事个词的搜索结果，而不是空格分隔的单个字的搜索结果。

还有那个地图功能，有回本人开车去北海玩，百度给出的路线似乎比谷歌更短，但当本人真正按百度的搜索结果行驶时，结果在最后的路口却遇到了禁止掉头的标志，当时真是后悔死了。

好了，姑且说这么多吧，再说肯定会招人烦了，呵呵。不过还是真心希望我们的产品经理们和程序员们再努努力，把自己的产品再做得完美些，不要让人挑出这么多毛病来。

19.1 acl_cpp 概述

发表时间: 2012-04-29

acl (http://http://sourceforge.net/projects/acl/?source=directory) 是本人早些年开始的一个通用的跨平台的网络通信及服务器程序的框架库（当然，除此之外，还包含其它功能丰富的库），全部是用C语言开发的，后来在公司做一些具体的项目时，觉得直接用C语言写东西开发效率还是有点低，于是逐渐将 acl 库中一些好用的功能库用C++语言进行了封装和整理，逐渐地变演变成了现在 acl_cpp 库，里面的内容更加丰富，不仅有很多 acl 的C语言版本的一些功能，同时还增加了一些其它功能模块（如 mime 邮件解析库，handler socket 库，memcached协议库等）。既然 acl_cpp 库是在 acl 库的基础上发展起来的，acl_cpp 库的命名空间自然就定为 acl 了，所以用户在使用 acl_cpp 的类时需要添加 acl:: 前缀，或者在源程序开始处添加一行：using namespace acl。在此，不得不多说一句，C++语言确实非常强大，当然，做为一名应用开发者，我并不需要完全理解C++的深层含意，而应从实用角度出发将自己经常用到的一些语言特性掌握熟练即可，这也正如 C++ 之父所说，使用一门语言，你并不需要完全熟悉它。

20.1 acl_cpp 的编译与使用

发表时间: 2012-05-01

本文主要描述了如何编译与使用 acl_cpp，因为目前 acl_cpp 仅支持 WIN32 平台及 LINUX 平台，所以建议大家在此二平台上编译和使用。下面分别就此两个平台进行介绍。

一、Windows 平台的编译

在 acl_cpp/ 目录下有两个 vc 工程文件：acl_cpp_vc2003.sln 及 acl_cpp_vc2010.sln，即用户可以在 Windows 下使用 vc2003 及 vc2010 进行编译，另外，除了一些功能性的原因外，笔者还是建议用户使用 vc2003 进行编译，原因很简单，vc2003 在笔者的机器上要比 vc2010 更节省资源。

在 Windows 下编译 acl_cpp 时，可以编译成四个库：动态库调试版、动态库发布版、静态库调试版以及静态库发布版，分别在 DebugDll、ReleaseDll、Debug、Release 目录下，用户在使用时，只要根据需要将不同的库进行编译链接即可，另外，头文件在 include/ 目录下，用户只需要将 include/ 下的所有 .hpp 文件拷贝到一个独立的目录下即可。

另外，因为 acl_cpp 是在 acl 项目的基础上开发的，所以用户还需要将 acl/lib_acl 及 acl/lib_protocol 下的两个库拷贝到用户自己的编译环境中来，想要编译 lib_acl 及 lib_protocol ，用户只需要用 vc2003 打开 acl\win32_build\vc\acl_project_vc2003.sln 工程文件，便可分别编译 lib_acl 及 lib_protocol 库，这两个库分别在 acl\win32_build\vc\lib_acl 及 acl\win32_build\vc\lib_prrotocol 目录下。

20.2 acl_cpp 的编译与使用

发表时间: 2012-05-20

acl_cpp 是基于 acl 为基础开发的，目前 acl_cpp 象 acl 一样支持 Linux 和 Windows 平台。有关 acl 的编译，请参考 《acl 的编译与使用 》，本文主要描述 acl_cpp 的编译与使用。

一、Linux 平台

acl_cpp 库编译后的静态库名为：lib_acl_cpp.a

1、编译 lib_acl_cpp.a 库

进入 acl_cpp 目录，直接运行命令：make 便可以在 lib/ 目录下生成 lib_acl_cpp.a 库，头文件在 include/ 目录下。

2、使用 lib_acl_cpp.a 库

用户在使用 lib_acl_cpp.a 库时，需要修改自己的 Makefile 文件，增加编译选项如下：

-I 指定 acl_cpp/include 头文件目录 ；

-L 指定 lib_acl_cpp.a 所在目录

-l_acl_cpp

因为 lib_acl_cpp.a 基于 lib_acl.a 和 lib_protocol.a，所以在链接您的程序时还需要添加这两个依赖库的位置，如：

-L {path_to_acl} -l_acl -L {path_to_protocol} -l_protocol

另外，用户需要在自己的源程序或头文件中包含头文件：#include "lib_acl.hpp"

二、Windows 平台

Win32 平台下，静态库名为：lib_acl_cpp.lib

1、编译 lib_acl_cpp.lib/lib_acl_cpp.dll 库

目前可以用 vc2003 或 vc2010 分别打开工程文件：acl_cpp_vc2003.sln 或 acl_cpp_vc2010.sln 来编译 win32 下的静态库或动态库。

2、使用 lib_acl_cpp.lib/lib_acl_cpp.dll 库

在您的工程中需要指定 acl_cpp/include 的头文件路径，同时在链接时需要指定库的位置，此外，还需要在您的源程序或头文件中包括头文件 "lib_acl.hpp"；

另外，还得把 lib_acl_vc2003.lib 和 lib_protocol_vc2003.lib 拷贝到您的工程目录中；如果是连接动态库，则需要将 lib_acl.dll, lib_protocol.dll, lib_acl_cpp_vc2003_dll.dll 动态库拷贝至您的可执行程序折运行目录。

如果您对编译和使用 acl_cpp 的库有疑问，请参考 acl_cpp/samples/ 下的示例，有完整的 Makefile 文件或 win32 下的工程文件。

acl_cpp 下载： http://sourceforge.net/projects/aclcpp/

原文件地址：http://zsxxsz.iteye.com/blog/1535688

acl_cpp 库所依赖的 acl 库资源：

acl 介绍

acl 下载

acl 的编译与使用

更多文章： http://zsxxsz.iteye.com/

21.1 使用 acl_cpp 的 HttpServlet 类及服务器框架编写WEB服务器程序

发表时间: 2012-05-21 关键字: web 应用, HTTP应用, 服务器编程, acl_cpp 库

在 《用C++实现类似于JAVA HttpServlet 的编程接口 》 文章中讲了如何用 HttpServlet 等相关类编写 CGI 程序，于是有网友提出了 CGI 程序低效性，不错，确实 CGI 程序的进程开销是比较大的，本文就将说明依然是这些 HTTP 相关的类，如果在使用 acl_cpp/src/master 下的服务器框架类的条件下，可以非常方便地转为服务器程序。现在依然是使用 《用C++实现类似于JAVA HttpServlet 的编程接口 》示例中的 http_servlet 类，只是稍微修改一下 main 函数，就变成下面的情形：

```
////////////////////////////////////////////////////////////////////////

class master_service : public acl::master_proc
{
public:
        master_service() {}
        ~master_service() {}
protected:
        // 基类虚函数，当接收到一个 HTTP 客户端请求时，服务器
        // 框架回调此函数将连接流传入
        virtual void on_accept(acl::socket_stream* stream)
        {
                // HttpServlet 的子类实例
                http_servlet servlet("127.0.0.1:11211");
                servlet.setLocalCharset("gb2312");  // 设置本地字符集
                servlet.doRun(stream);  // 开始处理浏览器请求过程
        }
};

////////////////////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{
        acl::acl_cpp_init();  // 初始化 acl_cpp 库
```

```
        master_service service;  // 半驻留进程池服务类对象

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                // 当在手工调试时一般采用此方式
                printf("listen: 127.0.0.1:8888 ...\r\n");
                service.run_alone("127.0.0.1:8888", NULL, 1);  // 单独运行方式
        }
        else  // 生产环境中以半驻留进程池模式运行
                service.run_daemon(argc, argv);  // acl_master 控制模式运行

        return 0;
}
```

上面的例子是一个结合 HttpServlet 类及 master_service 进程池服务类的 HTTP 服务器程序，关于进程池的例子，可以先结合本人原来写过的基于C语言库 acl 的一篇文章《快速创建你的服务器程序 - - single进程池模型 》。

不仅可以非常容易地将 HttpServlet 写成进程池方式，同时还可以结合 acl_cpp 的线程池框架模板，将 HttpServlet 类实现为半驻留线程池实例，下面就显示了这一过程：

```
class master_threads_test : public acl::master_threads
{
public:
        master_threads_test() {}

        ~master_threads_test() {}

protected:
        // 基类纯虚函数：当客户端连接有数据可读或关闭时回调此函数，返回 true 表示
        // 继续与客户端保持长连接，否则表示需要关闭客户端连接
```

```
        virtual bool thread_on_read(acl::socket_stream* stream)
        {
                // HttpServlet 的子类实例
                http_servlet servlet("127.0.0.1:11211");
                servlet.setLocalCharset("gb2312");  // 设置本地字符集
                servlet.doRun(stream);  // 开始处理浏览器请求过程
        }

        // 基类虚函数：当接收到一个客户端请求时，调用此函数，允许
        // 子类事先对客户端连接进行处理，返回 true 表示继续，否则
        // 要求关闭该客户端连接
        virtual bool thread_on_accept(acl::socket_stream*)
        {
                return true;  // 返回 true 以允许服务器框架继续调用 thread_on_read 过程
        }
};

// 字符串类配置参数项

static char *var_cfg_debug_msg;

static acl::master_str_tbl var_conf_str_tab[] = {
        { "debug_msg", "test_msg", &var_cfg_debug_msg },

        { 0, 0, 0 }
};

// 布尔配置参数项
static int  var_cfg_debug_enable;
static int  var_cfg_keep_alive;
static int  var_cfg_loop;

static acl::master_bool_tbl var_conf_bool_tab[] = {
        { "debug_enable", 1, &var_cfg_debug_enable },
        { "keep_alive", 1, &var_cfg_keep_alive },
        { "loop_read", 1, &var_cfg_loop },
```

```
        { 0, 0, 0 }
};

// 整数配置参数项
static int  var_cfg_io_timeout;

static acl::master_int_tbl var_conf_int_tab[] = {
        { "io_timeout", 120, &var_cfg_io_timeout, 0, 0 },

        { 0, 0 , 0 , 0, 0 }
};

int main(int argc, char* argv[])
{
        master_threads_test mt;  // 半驻留线程池服务器实例

        // 设置配置参数表
        mt.set_cfg_int(var_conf_int_tab);
        mt.set_cfg_int64(NULL);
        mt.set_cfg_str(var_conf_str_tab);
        mt.set_cfg_bool(var_conf_bool_tab);

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                // 当在手工调试时一般采用此方式
                printf("listen: 127.0.0.1:8888\r\n");
                mt.run_alone("127.0.0.1:8888", NULL, 2, 10);  // 单独运行方式
        }
        else  // 生产环境中以半驻留线程池模式运行
                mt.run_daemon(argc, argv);  // acl_master 控制模式运行
        return 0;
}
```

该例子显示了一个基于线程池服务器模型的WEB实例，可以依然使用了文章 《用C++实现类似于JAVA HttpServlet 的编程接口 》示例中的 http_servlet 类，但采用的是由文章 《 开发多线程进程池服务器程序---acl 服务器框架应用 》所介绍的多进程多线程服务器框架模板。

acl_cpp 下载

原文地址

更多文章

21.2 使用 acl::master_threads 类编写多进程多线程服务器程序

发表时间: 2012-05-26 关键字: 服务器应用程序, 多进程, 多线程

文章《开发多线程进程池服务器程序》讲述了如何使用 acl 库中的服务器模板编写多进程多线程服务器程序，那个例子是用 C 语言实现的，acl_cpp 对 acl 库用 c++ 语言进行了封装，其中也包含服务器编程模块，本文主要讲述如何使用 acl_cpp 中的 master_threads 类编写可以由 acl_master 服务器父进程控制的服务器应用程序。关于基于acl_master 的服务器程序设计原理，请参考 《协作半驻留式服务器程序开发框架》。

一、类接口说明

master_threads 是一个纯虚类，其中定义的接口需要子类实现，如下：

```
        /**
              * 纯虚函数：当某个客户端连接有数据可读或关闭或出错时调用此函数
              * @param stream {socket_stream*}
              * @return {bool} 返回 false 则表示当函数返回后需要关闭连接，
              *  否则表示需要保持长连接，如果该流出错，则应用应该返回 false
              */
             virtual bool thread_on_read(socket_stream*) = 0;

             /**
              * 当线程池中的某个线程获得一个连接时的回调函数，
              * 子类可以做一些初始化工作
              * @param stream {socket_stream*}
              * @return {bool} 如果返回 false 则表示子类要求关闭连接，而不
              *  必将该连接再传递至 thread_main 过程
              */
             virtual bool thread_on_accept(socket_stream*) { return true; }

             /**
              * 当与某个线程绑定的连接关闭时的回调函数
              * @param stream {socket_stream*}
              */
             virtual void thread_on_close(socket_stream* ) {}

             /**
              * 当线程池中一个新线程被创建时的回调函数
```

```
     */
    virtual void thread_on_init() {}

    /**
     * 当线程池中一个线程退出时的回调函数
     */
    virtual void thread_on_exit() {}
```

master_threads 类提供了两个函数：

```
/**
     * 开始运行，调用该函数是指该服务进程是在 acl_master 服务框架
     * 控制之下运行，一般用于生产机状态
     * @param argc {int} 从 main 中传递的第一个参数，表示参数个数
     * @param argv {char**} 从 main 中传递的第二个参数
     */
    void run_daemon(int argc, char** argv);

    /**
     * 在单独运行时的处理函数，用户可以调用此函数进行一些必要的调试工作
     * @param addr {const char*} 服务监听地址
     * @param conf {const char*} 配置文件全路径
     * @param count {unsigned int} 循环服务的次数，达到此值后函数自动返回；
     *  若该值为 0 则表示程序一直循环处理外来请求而不返回
     * @param threads_count {int} 当该值大于 1 时表示自动采用线程池方式，
     *  该值只有当 count != 1 时才有效，即若 count == 1 则仅运行一次就返回
     *  且不会启动线程处理客户端请求
     * @return {bool} 监听是否成功
     */
    bool run_alone(const char* addr, const char* conf = NULL,
            unsigned int count = 1, int threads_count = 1);
```

从上面两个函数，可以看出 master_threads 类当在生产环境下（由 acl_master 进程统一控制调度），用户需要调用 run_daemon 函数；如果用户在开发过程中需要手工进行调试，则可以调用 run_alone 函数。

master_threads 的基类 master_base 的几个虚接口如下：

```
	/**
	 * 当进程切换用户身份前调用的回调函数，可以在此函数中做一些
	 * 用户身份为 root 的权限操作
	 */
	virtual void proc_pre_jail() {}

	/**
	 * 当进程切换用户身份后调用的回调函数，此函数被调用时，进程
	 * 的权限为普通受限级别
	 */
	virtual void proc_on_init() {}

	/**
	 * 当进程退出前调用的回调函数
	 */
	virtual void proc_on_exit() {}

	// 在 run_alone 状态下运行前，调用此函数初始化一些配置
```

基类的这几个虚函数用户可以根据需要调用。

另外，基类 master_base 还提供了几个用来读配置选项的函数：

```
	/**
	 * 设置 bool 类型的配置项
	 * @param table {master_bool_tbl*}
	 */
	void set_cfg_bool(master_bool_tbl* table);

	/**
	 * 设置 int 类型的配置项
	 * @param table {master_int_tbl*}
	 */
	void set_cfg_int(master_int_tbl* table);

	/**
```

```
     * 设置 int64 类型的配置项
     * @param table {master_int64_tbl*}
     */
    void set_cfg_int64(master_int64_tbl* table);

    /**
     * 设置 字符串 类型的配置项
     * @param table {master_str_tbl*}
     */
    void set_cfg_str(master_str_tbl* table);
```

二、示例源程序

```
// master_threads.cpp : 定义控制台应用程序的入口点。
//

#include "log.hpp"
#include "util.hpp"
#include "master_threads.hpp"
#include "socket_stream.hpp"

// 字符串类型的配置项
static char *var_cfg_debug_msg;

static acl::master_str_tbl var_conf_str_tab[] = {
        { "debug_msg", "test_msg", &var_cfg_debug_msg },

        { 0, 0, 0 }
};

// 布尔类型的配置项
static int  var_cfg_debug_enable;
static int  var_cfg_keep_alive;
static int  var_cfg_loop;
```

```
static acl::master_bool_tbl var_conf_bool_tab[] = {
        { "debug_enable", 1, &var_cfg_debug_enable },
        { "keep_alive", 1, &var_cfg_keep_alive },
        { "loop_read", 1, &var_cfg_loop },

        { 0, 0, 0 }
};

// 整数类型的配置项
static int  var_cfg_io_timeout;

static acl::master_int_tbl var_conf_int_tab[] = {
        { "io_timeout", 120, &var_cfg_io_timeout, 0, 0 },

        { 0, 0 , 0 , 0, 0 }
};

static void (*format)(const char*, ...) = acl::log::msg1;

//////////////////////////////////////////////////////////////////////////

class master_threads_test : public acl::master_threads
{
public:
        master_threads_test()
        {

        }

        ~master_threads_test()
        {

        }

protected:
        // 基类纯虚函数：当客户端连接有数据可读或关闭时回调此函数，返回 true 表示
        // 继续与客户端保持长连接，否则表示需要关闭客户端连接
```

```
	virtual bool thread_on_read(acl::socket_stream* stream)
	{
		while (true)
		{
			if (on_read(stream) == false)
				return false;
			if (var_cfg_loop == 0)
				break;
		}
		return true;
	}

	bool on_read(acl::socket_stream* stream)
	{
		format("%s(%d)", __FILE__, __LINE__);
		acl::string buf;
		if (stream->gets(buf) == false) // 读一行数据
		{
			format("gets error: %s", acl::last_serror());
			format("%s(%d)", __FILE__, __LINE__);
			return false;
		}
		if (buf == "quit")  // 如果客户端要求关闭连接，则返回 false 通知服务器框架关闭连接
		{
			stream->puts("bye!");
			return false;
		}

		if (buf.empty())
		{
			if (stream->write("\r\n") == -1)
			{
				format("write 1 error: %s", acl::last_serror());
				return false;
			}
		}
		else if (stream->write(buf) == -1)
```

```
                {
                        format("write 2 error: %s, buf(%s), len: %d",
                                acl::last_serror(), buf.c_str(), (int) buf.length());
                        return false;
                }
                else if (stream->write("\r\n") == -1)
                {
                        format("write 3 client error: %s", acl::last_serror());
                        return false;
                }
                return true;
        }

        // 基类虚函数：当接收到一个客户端请求时，调用此函数，允许
        // 子类事先对客户端连接进行处理，返回 true 表示继续，否则
        // 要求关闭该客户端连接
        virtual bool thread_on_accept(acl::socket_stream* stream)
        {
                format("accept one client, peer: %s, local: %s, var_cfg_io_timeout: %d\r\n",
                        stream->get_peer(), stream->get_local(), var_cfg_io_timeout);
                if (stream->format("hello, you're welcome!\r\n") == -1)
                        return false;
                return true;
        }

        // 基类虚函数：当客户端连接关闭时调用此函数
        virtual void thread_on_close(acl::socket_stream*)
        {
                format("client closed now\r\n");
        }

        // 基类虚函数：当线程池创建一个新线程时调用此函数
        virtual void thread_on_init()
        {
#ifdef WIN32
                format("thread init: tid: %lu\r\n", GetCurrentThreadId());
#else
```

```
                format("thread init: tid: %lu\r\n", pthread_self());
#endif
        }

        // 基类虚函数：当线程池中的一个线程退出时调用此函数
        virtual void thread_on_exit()
        {
#ifdef WIN32
                format("thread exit: tid: %lu\r\n", GetCurrentThreadId());
#else
                format("thread exit: tid: %lu\r\n", pthread_self());
#endif
        }

        // 基类虚函数：服务进程切换用户身份前调用此函数
        virtual void proc_pre_jail()
        {
                format("proc_pre_jail\r\n");
        }

        // 基类虚函数：服务进程切换用户身份后调用此函数
        virtual void proc_on_init()
        {
                format("proc init\r\n");
        }

        // 基类虚函数：服务进程退出前调用此函数
        virtual void proc_on_exit()
        {
                format("proc exit\r\n");
        }
private:
};

int main(int argc, char* argv[])
{
        master_threads_test mt;
```

```
        // 设置配置参数表
        mt.set_cfg_int(var_conf_int_tab);
        mt.set_cfg_int64(NULL);
        mt.set_cfg_str(var_conf_str_tab);
        mt.set_cfg_bool(var_conf_bool_tab);

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                format = (void (*)(const char*, ...)) printf;
                format("listen: 127.0.0.1:8888\r\n");
                mt.run_alone("127.0.0.1:8888", NULL, 2, 10);  // 单独运行方式
        }
        else
                mt.run_daemon(argc, argv);  // acl_master 控制模式运行
        return 0;
}
```

这是一个简单的提供 echo 行服务的服务器程序，可以支持多个并发连接，而且可以通过配置文件控制所启动的最大进程数、每个进程的最大线程数、进程空闲时间、线程空闲时间等控制参数，因为 acl 中的服务器框架都是半驻留的，所以既可以保证运行效率，又能够在空闲释放系统资源。该例子所在目录：acl_cpp/samples/master_threads。

需要指出的一点是，master_threads 内部是单例的，即要求该类的对象只能有一个，否则内部自动产生断言。只所以没有采用单例模板来设计单例，主要是为了不对外暴露 acl 库中的接口，使使用 acl_cpp 库的用户不必关心 acl 库的头文件在哪儿。

三、配置文件及程序安装

打开 acl_cpp/samples/master_threads/ioctl_echo.cf 配置文件，就其中几个配置参数说明一下：

```
## 由 acl_master 用来控制服务进程池的配置项
# 为 no 表示启用该进程服务，为 yes 表示禁止该服务进程
master_disable = no

# 表示本服务器进程监听 127.0.0.1 的 5001 端口
master_service = 127.0.0.1:5001

# 表示是 TCP 套接口服务类型
master_type = inet

# 表示该服务进程池的最大进程数为 2
master_maxproc = 2

# 进程日志记录文件，其中 {install_path} 需要用实际的安装路径代替
master_log = {install_path}/var/log/ioctl_echo.log
```

```
## 与该服务器框架模板相关的配置参数项
# 每个服务进程中最大的线程数为 250
ioctl_max_threads = 250

# 线程的堆栈空间大小，单位为字节，0表示使用系统缺省值
ioctl_stacksize = 0

# 每个进程实例处理连接数的最大次数，超过此值后进程实例主动退出
ioctl_use_limit = 100

# 每个进程实例的空闲超时时间，超过此值后进程实例主动退出
ioctl_idle_limit = 120

# 进程运行时的用户身份
ioctl_owner = root

# 采用事件循环的方式: select(default)/poll/kernel(epoll/devpoll/kqueue)
ioctl_event_mode = select

# 允许访问 udserver 的客户端IP地址范围
ioctl_access_allow = 10.0.0.1:10.0.0.255, 127.0.0.1:127.0.0.1
```

例如当 acl_master 服务器框架程序的安装目录为：/opt/acl，则：

/opt/acl/libexec： 该目录存储服务器程序（acl_master 程序也存放在该目录下）；

/opt/acl/conf：该目录存放 acl_master 程序配置文件 main.cf；

/opt/acl/conf/service：该目录存放服务子进程的程序配置文件，该路径由 main.cf 文件指定；

/opt/acl/var/log：该目录存放日志文件；

/opt/acl/var/pid：该目录存放进程号文件。

该程序编译通过后，需要把可执行程序放在 /opt/acl/libexec 目录下，把配置文件放在 /opt/acl/conf/service 目录下。

在 /opt/acl/sh 下有启动/停止 acl_master 服务进程的控制脚本；运行脚本：./start.sh，然后请用下面方法检查服务是否已经启动：

ps -ef|grep acl_master # 查看服务器控制进程是否已经启动

netstat -nap|grep LISTEN|grep 5001 # 查看服务端口号是否已经被监听

当然，您也可以查看 /opt/acl/var/log/acl_master 日志文件，查看服务进程的启动过程及监听服务是否正常监听。

可以命令行如下测试：telnet 127.0.0.1 5001

原文地址

acl_cpp 下载

acl_cpp 的编译与使用

更多文章

21.3 使用 acl::master_proc 类编写多进程服务器程序

发表时间: 2012-05-26 关键字: 服务器应用编程, 多进程服务器

文章《快速创建你的服务器程序》讲述了基于 C 语言版本的 acl 服务器框架下如何开发多进程服务器应用程序。本文则讲述了基于 C++ 语言的 acl_cpp 服务器框架下如何开发多进程服务器应用程序，当然 acl_cpp 下的服务器框架内部也是基于 acl 的服务器框架的。关于基于acl_master 的服务器程序设计原理，请参考 《协作半驻留式服务器程序开发框架》。

一、类成员函数说明

master_proc 是一个纯虚类，其中定义的接口需要子类实现，如下：

```
/**
 * 纯虚函数：当接收到一个客户端连接时调用此函数
 * @param stream {aio_socket_stream*} 新接收到的客户端异步流对象
 * 注：该函数返回后，流连接将会被关闭，用户不应主动关闭该流
 */
virtual void on_accept(socket_stream* stream) = 0;
```

master_proc 类提供了两个函数：

```
/**
 * 开始运行，调用该函数是指该服务进程是在 acl_master 服务框架
 * 控制之下运行，一般用于生产机状态
 * @param argc {int} 从 main 中传递的第一个参数，表示参数个数
 * @param argv {char**} 从 main 中传递的第二个参数
 */
void run_daemon(int argc, char** argv);

/**
 * 在单独运行时的处理函数，用户可以调用此函数进行一些必要的调试工作
 * @param addr {const char*} 服务监听地址
 * @param conf {const char*} 配置文件全路径
 * @param count {int} 当该值 > 0 时，则接收的连接次数达到此值且完成
```

```
     *  后，该函数将返回，否则一直循环接收远程连接
     * @return {bool} 监听是否成功
     */
    bool run_alone(const char* addr, const char* conf = NULL, int count = 1);
```

master_proc 类实例当在生产环境下（由 acl_master 进程统一控制调度），用户需要调用 run_daemon 函数；如果用户在开发过程中需要手工进行调试，则可以调用 run_alone 函数。

master_proc 的基类 master_base 的几个虚接口如下：

```
    /**
     * 当进程切换用户身份前调用的回调函数，可以在此函数中做一些
     * 用户身份为 root 的权限操作
     */
    virtual void proc_pre_jail() {}

    /**
     * 当进程切换用户身份后调用的回调函数，此函数被调用时，进程
     * 的权限为普通受限级别
     */
    virtual void proc_on_init() {}

    /**
     * 当进程退出前调用的回调函数
     */
    virtual void proc_on_exit() {}

    // 在 run_alone 状态下运行前，调用此函数初始化一些配置
```

二、示例

```
// master_proc.cpp : 定义控制台应用程序的入口点。
//
```

```
#include "stdafx.h"
#include "lib_acl.hpp"

// 字符串类型的配置项
static char *var_cfg_debug_msg;

static acl::master_str_tbl var_conf_str_tab[] = {
	{ "debug_msg", "test_msg", &var_cfg_debug_msg },

	{ 0, 0, 0 }
};

// 布尔类型的配置项
static int  var_cfg_debug_enable;

static acl::master_bool_tbl var_conf_bool_tab[] = {
	{ "debug_enable", 1, &var_cfg_debug_enable },

	{ 0, 0, 0 }
};

// 整数类型的配置项
static int  var_cfg_io_timeout;

static acl::master_int_tbl var_conf_int_tab[] = {
	{ "io_timeout", 120, &var_cfg_io_timeout, 0, 0 },

	{ 0, 0 , 0 , 0, 0 }
};

static void (*format)(const char*, ...) = acl::log::msg1;

//////////////////////////////////////////////////////////////////////////
using namespace acl;

class master_proc_test : public master_proc
```

```
{
public:
	master_proc_test() {}
	~master_proc_test() {}
protected:
	/**
	 * 基类纯虚函数：当接收到一个客户端连接时调用此函数
	 * @param stream {aio_socket_stream*} 新接收到的客户端异步流对象
	 * 注：该函数返回后，流连接将会被关闭，用户不应主动关闭该流
	 */
	virtual void on_accept(socket_stream* stream)
	{
		if (stream->format("hello, you're welcome!\r\n") == -1)
			return;
		while (true)
		{
			if (on_read(stream) == false)
				break;
		}
	}

	bool on_read(socket_stream* stream)
	{
		string buf;
		if (stream->gets(buf) == false)  // 读一行数据
		{
			format("gets error: %s", acl::last_serror());
			return false;
		}
		if (buf == "quit")
		{
			stream->puts("bye!");
			return false;
		}

		if (buf.empty())
		{
```

```
                if (stream->write("\r\n") == -1)
                {
                        format("write 1 error: %s", acl::last_serror());
                        return false;
                }
        }
        else if (stream->write(buf) == -1)
        {
                format("write 2 error: %s, buf(%s), len: %d",
                        acl::last_serror(), buf.c_str(), (int) buf.length());
                return false;
        }
        else if (stream->write("\r\n") == -1)
        {
                format("write 3 client error: %s", acl::last_serror());
                return false;
        }
        return true;
}

// 基类虚函数：服务进程切换用户身份前调用此函数
virtual void proc_pre_jail()
{
        format("proc_pre_jail\r\n");
}

// 基类虚函数：服务进程切换用户身份后调用此函数
virtual void proc_on_init()
{
        format("proc init\r\n");
}

// 基类虚函数：服务进程退出前调用此函数
virtual void proc_on_exit()
{
        format("proc exit\r\n");
}
```

```
private:
};
//////////////////////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{
	master_proc_test mp;

	// 设置配置参数表
	mp.set_cfg_int(var_conf_int_tab);
	mp.set_cfg_int64(NULL);
	mp.set_cfg_str(var_conf_str_tab);
	mp.set_cfg_bool(var_conf_bool_tab);

	// 开始运行

	if (argc >= 2 && strcmp(argv[1], "alone") == 0)
	{
		format = (void (*)(const char*, ...)) printf;
		mp.run_alone("127.0.0.1:8888", NULL, 5);  // 单独运行方式
	}
	else
		mp.run_daemon(argc, argv);  // acl_master 控制模式运行

	return 0;
}
```

这是一个简单的提供 echo 行服务的服务器程序，可以支持多个并发连接，而且可以通过配置文件控制所启动的最大进程数、进程空闲时间等控制参数，因为 acl 中的服务器框架都是半驻留的，所以既可以保证运行效率，又能够在空闲释放系统资源。该例子所在目录：acl_cpp/samples/master_proc。

需要注意的是，master_proc 内部是单例的，即要求该类的对象只能有一个，否则内部自动产生断言。只所以没有采用单例模板来设计单例，主要是为了不对外暴露 acl 库中的接口，使使用 acl_cpp 库的用户不必关心 acl 库的头文件在哪儿。

三、配置文件及程序安装

打开 acl_cpp/samples/master_proc/single_echo.cf 配置文件，就其中几个配置参数说明一下：

```
## 由 acl_master 用来控制服务进程池的配置项
# 为 no 表示启用该进程服务，为 yes 表示禁止该服务进程
master_disable = no

# 表示本服务器进程监听 127.0.0.1 的 5002 端口
master_service = 127.0.0.1:5002

# 表示是 TCP 套接口服务类型
master_type = inet

# 进程程序名
master_command = master_proc

# 表示该服务进程池的最大进程数为 2
master_maxproc = 2

# 进程日志记录文件，其中 {install_path} 需要用实际的安装路径代替
master_log = {install_path}/var/log/single_echo.log
```

```
# 每个进程实例处理连接数的最大次数，超过此值后进程实例主动退出
single_use_limit = 250
# 每个进程实例的空闲超时时间，超过此值后进程实例主动退出
single_idle_limit = 180
```

例如当 acl_master 服务器框架程序的安装目录为：/opt/acl，则：

/opt/acl/libexec： 该目录存储服务器程序（acl_master 程序也存放在该目录下）；

/opt/acl/conf：该目录存放 acl_master 程序配置文件 main.cf；

/opt/acl/conf/service：该目录存放服务子进程的程序配置文件，该路径由 main.cf 文件指定；

/opt/acl/var/log：该目录存放日志文件；

/opt/acl/var/pid：该目录存放进程号文件。

该程序编译通过后，需要把可执行程序放在 /opt/acl/libexec 目录下，把配置文件放在 /opt/acl/conf/service 目录下。

在 /opt/acl/sh 下有启动/停止 acl_master 服务进程的控制脚本；运行脚本：./start.sh，然后请用下面方法检查服务是否已经启动：

ps -ef|grep acl_master # 查看服务器控制进程是否已经启动

netstat -nap|grep LISTEN|grep 5002 # 查看服务端口号是否已经被监听

当然，您也可以查看 /opt/acl/var/log/acl_master 日志文件，查看服务进程的启动过程及监听服务是否正常监听。

可以命令行如下测试：telnet 127.0.0.1 5002

原文地址

acl_cpp 下载

acl_cpp 的编译与使用

更多文章

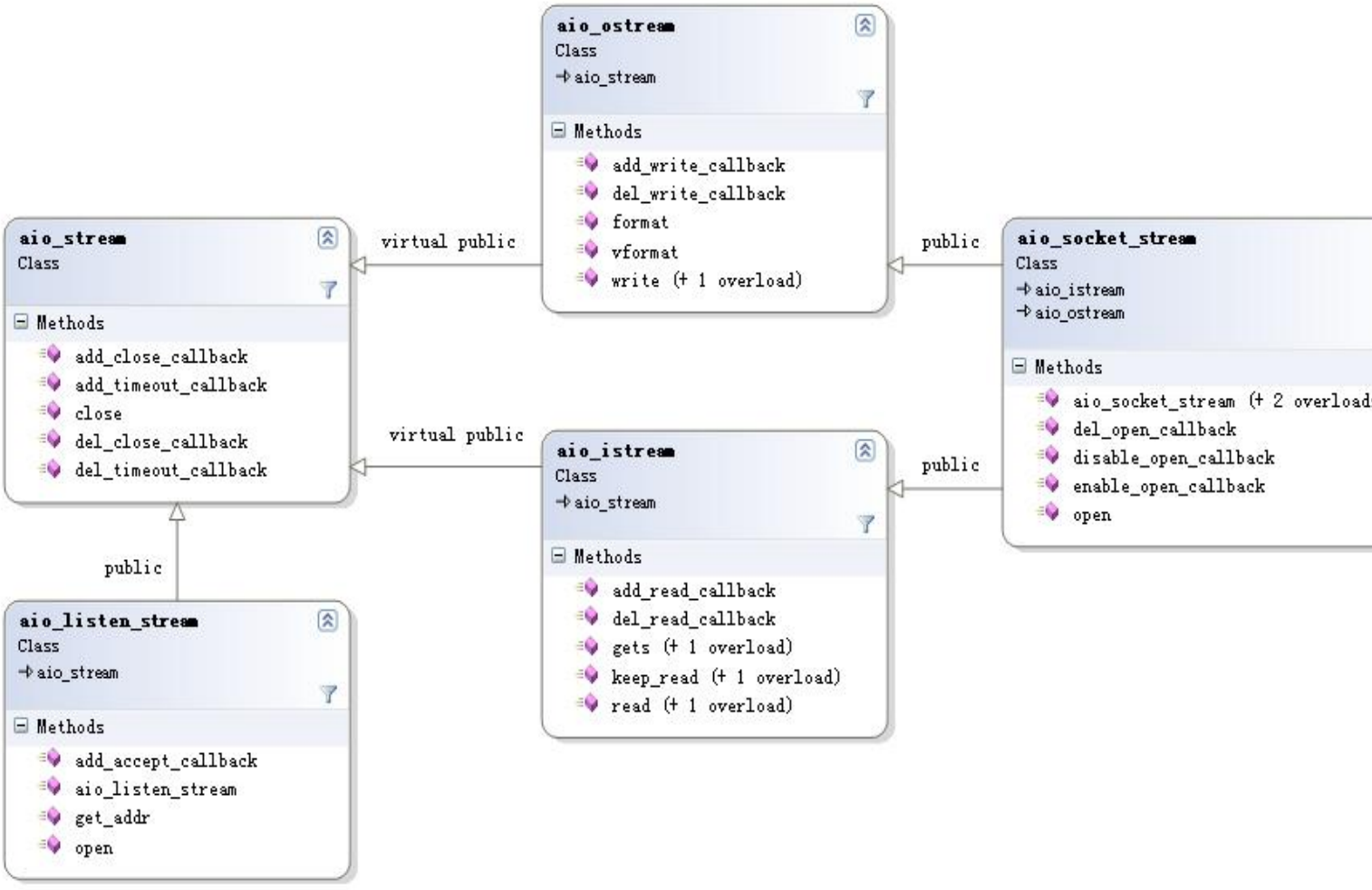
22.1 非阻塞网络编程实例讲解

发表时间: 2012-04-04 关键字: 服务器编程, 网络编程, 非阻塞编程, 异步流编程

一、概述

acl_cpp 异步流的设计思路为：异步流类与异步流接口类，其中异步流类对象完成网络套接口监听、连接、读写的操作，异步流接口类对象定义了网络读写成功/超时回调、连接成功回调、接收客户端连接回调等接口；用户在进行异步编程时，首先必须实现接口类中定义的纯方法，然后将接口类对象在异步流对象中进行注册，这样当满足接口类对象的回调条件时 acl_cpp 的异步框架便自动调用用户定义的接口方法。

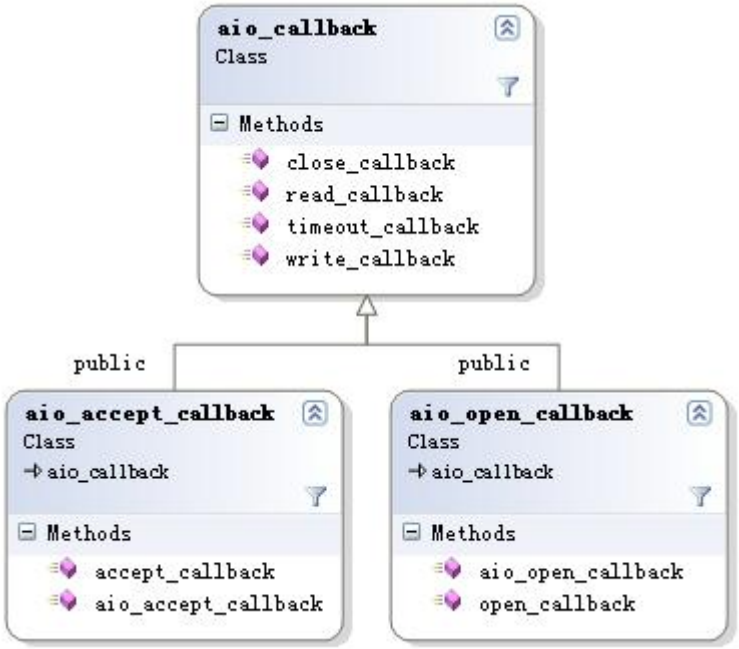
在 acl_cpp 中异步流的类继续关系如下图所示：



由上图可以看出，基类 aio_stream 中定义了流关闭，注册/取消流关闭回调和流超时回调等基础方法；aio_istream 和 aio_ostream 分别定义了异步流读及写的基本方法，aio_istream 中包含添加/删除流读成功回调接口类对象的方法，aio_ostream 中包含添加/删除流写成功回调接口类对象的方法；aio_socket_stream 类对象为连接服务器成功后的客户端流，或服务器接收到客户端连接创建的客户端连接流，其中定义了做为连接

流时远程连接的方法及添加连接成功回调接口的方法；aio_listen_stream 类为监听流类，其中定义了监听某个网络地址（或UNIX下的域套接口地址）方法，以及注册接收成功接口的方法。

acl_cpp 异步流接口类继承关系图如下图：



异步流接口类的设计中：aio_accept_callback 为监听流的回调接口类，用户应继承该类以获得外来客户端连接流，同时还需要定义继承于 aio_callback 的类，用于获得网络读写操作等结果信息；aio_open_callback 只有当客户端连接远程服务器时，用户需要实现其子类获得连接成功的结果。

二、实例

1、异步服务器

```
#include <iostream>
#include <assert.h>
#include "aio_handle.hpp"
#include "aio_istream.hpp"
#include "aio_listen_stream.hpp"
#include "aio_socket_stream.hpp"
```

```
using namespace acl;

/**
 * 异步客户端流的回调类的子类
 */
class io_callback : public aio_callback
{
public:
        io_callback(aio_socket_stream* client)
                : client_(client)
                , i_(0)
        {
        }

        ~io_callback()
        {
                std::cout << "delete io_callback now ..." << std::endl;
        }

        /**
         * 实现父类中的虚函数，客户端流的读成功回调过程
         * @param data {char*} 读到的数据地址
         * @param len {int} 读到的数据长度
         * @return {bool} 返回 true 表示继续，否则希望关闭该异步流
         */
        bool read_callback(char* data, int len)
        {
                i_++;
                if (i_ < 10)
                        std::cout << ">>gets(i:" << i_ << "): " << data;

                // 如果远程客户端希望退出，则关闭之
                if (strncasecmp(data, "quit", 4) == 0)
                {
                        client_->format("Bye!\r\n");
                        client_->close();
```

```
        }

        // 如果远程客户端希望服务端也关闭，则中止异步事件过程
        else if (strncasecmp(data, "stop", 4) == 0)
        {
                client_->format("Stop now!\r\n");
                client_->close();  // 关闭远程异步流

                // 通知异步引擎关闭循环过程
                client_->get_handle().stop();
        }

        // 向远程客户端回写收到的数据

        client_->write(data, len);

        return (true);
}

/**
 * 实现父类中的虚函数，客户端流的写成功回调过程
 * @return {bool} 返回 true 表示继续，否则希望关闭该异步流
 */
bool write_callback()
{
        return (true);
}

/**
 * 实现父类中的虚函数，客户端流的超时回调过程
 */
void close_callback()
{
        // 必须在此处删除该动态分配的回调类对象以防止内存泄露
        delete this;
}
```

```
        /**
         * 实现父类中的虚函数，客户端流的超时回调过程
         * @return {bool} 返回 true 表示继续，否则希望关闭该异步流
         */
        bool timeout_callback()
        {
                std::cout << "Timeout ..." << std::endl;
                return (true);
        }

private:
        aio_socket_stream* client_;
        int   i_;
};

/**
 * 异步监听流的回调类的子类
 */
class io_accept_callback : public aio_accept_callback
{
public:
        io_accept_callback() {}
        ~io_accept_callback()
        {
                printf(">>io_accept_callback over!\n");
        }

        /**
         * 基类虚函数，当有新连接到达后调用此回调过程
         * @param client {aio_socket_stream*} 异步客户端流
         * @return {bool} 返回 true 以通知监听流继续监听
         */
        bool accept_callback(aio_socket_stream* client)
        {
                // 创建异步客户端流的回调对象并与该异步流进行绑定
                io_callback* callback = new io_callback(client);
```

```
                // 注册异步流的读回调过程
                client->add_read_callback(callback);

                // 注册异步流的写回调过程
                client->add_write_callback(callback);

                // 注册异步流的关闭回调过程
                client->add_close_callback(callback);

                // 注册异步流的超时回调过程
                client->add_timeout_callback(callback);

                // 从异步流读一行数据
                client->gets(10, false);
                return (true);
        }
};

int main(int argc, char* argv[])
{
        // 初始化ACL库(尤其是在WIN32下一定要调用此函数，在UNIX平台下可不调用)
        acl_cpp_init();

        // 构建异步引擎类对象
        aio_handle handle(ENGINE_KERNEL);

        // 创建监听异步流
        aio_listen_stream* sstream = new aio_listen_stream(&handle);
        const char* addr = "127.0.0.1:9001";

        // 监听指定的地址
        if (sstream->open(addr) == false)
        {
                std::cout << "open " << addr << " error!" << std::endl;
                sstream->close();
                // XXX: 为了保证能关闭监听流，应在此处再 check 一下
                handle.check();
```

```
                getchar();
                return (1);
        }

        // 创建回调类对象，当有新连接到达时自动调用此类对象的回调过程
        io_accept_callback callback;
        sstream->add_accept_callback(&callback);
        std::cout << "Listen: " << addr << " ok!" << std::endl;

        while (true)
        {
                // 如果返回 false 则表示不再继续，需要退出
                if (handle.check() == false)
                {
                        std::cout << "aio_server stop now ..." << std::endl;
                        break;
                }
        }

        // 关闭监听流并释放流对象
        sstream->close();

        // XXX: 为了保证能关闭监听流，应在此处再 check 一下
        handle.check();

        return (0);
}
```

简要说明一下，上面代码的基本思路是：

a) 创建异步通信框架对象 aio_handle --> 创建异步监听流 aio_listen_stream 并注册回调类对象 io_accept_callback-->进入异步通信框架的事件循环中；

b) 当接收到客户端连接后，异步框架回调 io_accept_callback 类对象的 accept_callback 接口并将客户端异步流输入-->创建异步流接口类对象，并将该对象注册至客户端异步流对象中;

c) 当客户端异步流收到数据时回调异步流接口中的 read_callback 方法 --> 回写收到数据至客户端；当客户端流连接关闭时回调异步流接口中的close_callback --> 如果该接口类对象是动态创建的则需要手工 delete 掉；当接收客户端数据超时时会回调异步流接口中的 time_callback，该函数如果返回 true 则表示希望异步框架不关闭该客户端异步流，否则则关闭。

异步监听流的接口类的纯虚函数：virtual bool accept_callback(aio_socket_stream* client) 需要子类实现，子类在该函数中获得客户端连接异步流对象。

客户端异步流接口类 aio_callback 有四个虚函数：

a) virtual bool read_callback(char* data, int len) 当客户端异步流读到数据时的回调虚函数；

b) virtual bool write_callback() 当客户端异步流写数据成功后的回调虚函数；

c) virtual void close_callback() 当异步流(客户端流或监听流)关闭时的回调虚函数；

d) virtual bool timeout_callback() 当异步流（客户端流在读写超时或监听流在监听超时）超时时的回调函数虚函数。

2、异步客户端

```
#include <iostream>
#include <assert.h>
#include "string.hpp"
#include "util.hpp"
#include "aio_handle.hpp"
#include "acl_cpp_init.hpp"
#include "aio_socket_stream.hpp"

#ifdef WIN32
# ifndef snprintf
#  define snprintf _snprintf
```

```
# endif
#endif

using namespace acl;

typedef struct
{
        char  addr[64];
        aio_handle* handle;
        int   connect_timeout;
        int   read_timeout;
        int   nopen_limit;
        int   nopen_total;
        int   nwrite_limit;
        int   nwrite_total;
        int   nread_total;
        int   id_begin;
        bool  debug;
} IO_CTX;

static bool connect_server(IO_CTX* ctx, int id);

/**
 * 客户端异步连接流回调函数类
 */
class client_io_callback : public aio_open_callback
{
public:
        /**
         * 构造函数
         * @param ctx {IO_CTX*}
         * @param client {aio_socket_stream*} 异步连接流
         * @param id {int} 本流的ID号
         */
        client_io_callback(IO_CTX* ctx, aio_socket_stream* client, int id)
                : client_(client)
                , ctx_(ctx)
```

```
        , nwrite_(0)
        , id_(id)
{
}

~client_io_callback()
{
        std::cout << ">>>ID: " << id_ << ", io_callback deleted now!" << std::endl;
}

/**
 * 基类虚函数，当异步流读到所要求的数据时调用此回调函数
 * @param data {char*} 读到的数据地址
 * @param len {int } 读到的数据长度
 * @return {bool} 返回给调用者 true 表示继续，否则表示需要关闭异步流
 */
bool read_callback(char* data, int len)
{
        (void) data;
        (void) len;

        ctx_->nread_total++;

        if (ctx_->debug)
        {
                if (nwrite_ < 10)
                        std::cout << "gets(" << nwrite_ << "): " << data;
                else if (nwrite_ % 2000 == 0)
                        std::cout << ">>ID: " << id_ << ", I: "
                                << nwrite_ << "; "<<  data;
        }

        // 如果收到服务器的退出消息，则也应退出
        if (acl::strncasecmp_(data, "quit", 4) == 0)
        {
                // 向服务器发送数据
                client_->format("Bye!\r\n");
```

```
                // 关闭异步流连接
                client_->close();
                return (true);
        }

        if (nwrite_ >= ctx_->nwrite_limit)
        {
                if (ctx_->debug)
                        std::cout << "ID: " << id_
                                << ", nwrite: " << nwrite_
                                << ", nwrite_limit: " << ctx_->nwrite_limit
                                << ", quiting ..." << std::endl;

                // 向服务器发送退出消息
                client_->format("quit\r\n");
                client_->close();
        }
        else
        {
                char  buf[256];
                snprintf(buf, sizeof(buf), "hello world: %d\n", nwrite_);
                client_->write(buf, (int) strlen(buf));

                // 向服务器发送数据
                //client_->format("hello world: %d\n", nwrite_);
        }

        return (true);
}

/**
 * 基类虚函数，当异步流写成功时调用此回调函数
 * @return {bool} 返回给调用者 true 表示继续，否则表示需要关闭异步流
 */
bool write_callback()
{
        ctx_->nwrite_total++;
```

```
        nwrite_++;

        // 从服务器读一行数据
        client_->gets(ctx_->read_timeout, false);
        return (true);
}

/**
 * 基类虚函数，当该异步流关闭时调用此回调函数
 */
void close_callback()
{
        if (client_->is_opened() == false)
        {
                std::cout << "Id: " << id_ << " connect "
                        << ctx_->addr << " error: "
                        << acl::last_serror();

                // 如果是第一次连接就失败，则退出
                if (ctx_->nopen_total == 0)
                {
                        std::cout << ", first connect error, quit";
                        /* 获得异步引擎句柄，并设置为退出状态 */
                        client_->get_handle().stop();
                }
                std::cout << std::endl;
                delete this;
                return;
        }

        /* 获得异步引擎中受监控的异步流个数 */
        int nleft = client_->get_handle().length();
        if (ctx_->nopen_total == ctx_->nopen_limit && nleft == 1)
        {
                std::cout << "Id: " << id_ << " stop now! nstream: "
                        << nleft << std::endl;
                /* 获得异步引擎句柄，并设置为退出状态 */
```

```
                client_->get_handle().stop();
        }

        // 必须在此处删除该动态分配的回调类对象以防止内存泄露
        delete this;
}

/**
 * 基类虚函数，当异步流超时时调用此函数
 * @return {bool} 返回给调用者 true 表示继续，否则表示需要关闭异步流
 */
bool timeout_callback()
{
        std::cout << "Connect " << ctx_->addr << " Timeout ..." << std::endl;
        client_->close();
        return (false);
}

/**
 * 基类虚函数，当异步连接成功后调用此函数
 * @return {bool} 返回给调用者 true 表示继续，否则表示需要关闭异步流
 */
bool open_callback()
{
        // 连接成功，设置IO读写回调函数
        client_->add_read_callback(this);
        client_->add_write_callback(this);
        ctx_->nopen_total++;

        acl::assert_(id_ > 0);
        if (ctx_->nopen_total < ctx_->nopen_limit)
        {
                // 开始进行下一个连接过程
                if (connect_server(ctx_, id_ + 1) == false)
                        std::cout << "connect error!" << std::endl;
        }
```

```
                // 异步向服务器发送数据
                //client_->format("hello world: %d\n", nwrite_);
                char  buf[256];
                snprintf(buf, sizeof(buf), "hello world: %d\n", nwrite_);
                client_->write(buf, (int) strlen(buf));

                // 异步从服务器读取一行数据
                client_->gets(ctx_->read_timeout, false);

                // 表示继续异步过程
                return (true);
        }

protected:
private:
        aio_socket_stream* client_;
        IO_CTX* ctx_;
        int   nwrite_;
        int   id_;
};

static bool connect_server(IO_CTX* ctx, int id)
{
        // 开始异步连接远程服务器
        aio_socket_stream* stream = aio_socket_stream::open(ctx->handle,
                        ctx->addr, ctx->connect_timeout);
        if (stream == NULL)
        {
                std::cout << "connect " << ctx->addr << " error!" << std::endl;
                std::cout << "stoping ..." << std::endl;
                if (id == 0)
                        ctx->handle->stop();
                return (false);
        }

        // 创建连接后的回调函数类
        client_io_callback* callback = new client_io_callback(ctx, stream, id);
```

```
        // 添加连接成功的回调函数类
        stream->add_open_callback(callback);

        // 添加连接失败后回调函数类
        stream->add_close_callback(callback);

        // 添加连接超时的回调函数类
        stream->add_timeout_callback(callback);
        return (true);
}

int main(int argc, char* argv[])
{
        bool use_kernel = false;
        int   ch;
        IO_CTX ctx;

        memset(&ctx, 0, sizeof(ctx));
        ctx.connect_timeout = 5;
        ctx.nopen_limit = 10;
        ctx.id_begin = 1;
        ctx.nwrite_limit = 10;
        ctx.debug = false;
        snprintf(ctx.addr, sizeof(ctx.addr), "127.0.0.1:9001");

        acl_cpp_init();

        aio_handle handle(ENGINE_KERNEL);
        ctx.handle = &handle;

        if (connect_server(&ctx, ctx.id_begin) == false)
        {
                std::cout << "enter any key to exit." << std::endl;
                getchar();
                return (1);
        }
```

```
        std::cout << "Connect " << ctx.addr << " ..." << std::endl;

        while (true)
        {
                // 如果返回 false 则表示不再继续，需要退出
                if (handle.check() == false)
                        break;
        }

        return (0);
}
```

异步客户端的基本流程为：

a) 创建异步框架对象 aio_handle --> 异步连接远程服务器，创建连接成功/失败/超时的异步接口类对象并注册至异步连接流中 --> 异步框架进行事件循环中；

b) 连接成功后，异步接口类对象中的 open_callback 被调用，启动下一个异步连接过程（未达限制连接数前）--> 添加异步读及异步写的回调接口类 --> 异步写入数据，同时开始异步读数据过程；

c) 当客户端异步流收到数据时回调异步流接口中的 read_callback 方法 --> 回写收到数据至客户端；当客户端流连接关闭时回调异步流接口中的 close_callback --> 如果该接口类对象是动态创建的则需要手工 delete 掉；当接收客户端数据超时时会回调异步流接口中的 time_callback，该函数如果返回 true 则表示希望异步框架不关闭该客户端异步流，否则则关闭。

客户端异步连接流的接口类 aio_open_callback 的纯虚函数 virtual bool open_callback() 需要子类实现，在连接服务器成功后调用此函数，允许子类在该函数中做进一步操作，如：注册客户端流的异步读回调接口类对象及异步写回调类对象；如果连接超时或连接失败而导致的关闭，则基础接口类中的 timeout_callback() 或 close_callback() 将会被调用，以通知用户应用程序。

三、小结

以上的示例演示了基本的非阻塞异步流的监听、读写、连接等过程，类的设计中也提供了基本的操作方法，为了应对实践中的多样性及复杂性，acl_cpp 的异步流还设计了更多的接口和方法，如：延迟读写操作（这对于限流的服务器比较有用处）、定时器操作等。用户可以从如下地址下载完整的 acl_cpp 框架库的拷贝：https://sourceforge.net/projects/aclcpp/

非阻塞客户端实例：acl_cpp\samples\aio_client

非阻塞服务器实例：acl_cpp\samples\aio_server

2012.4.4

acl_cpp 下载：http://sourceforge.net/projects/aclcpp/

原文地址：http://zsxxsz.iteye.com/blog/1474009

更多文章：http://zsxxsz.iteye.com/

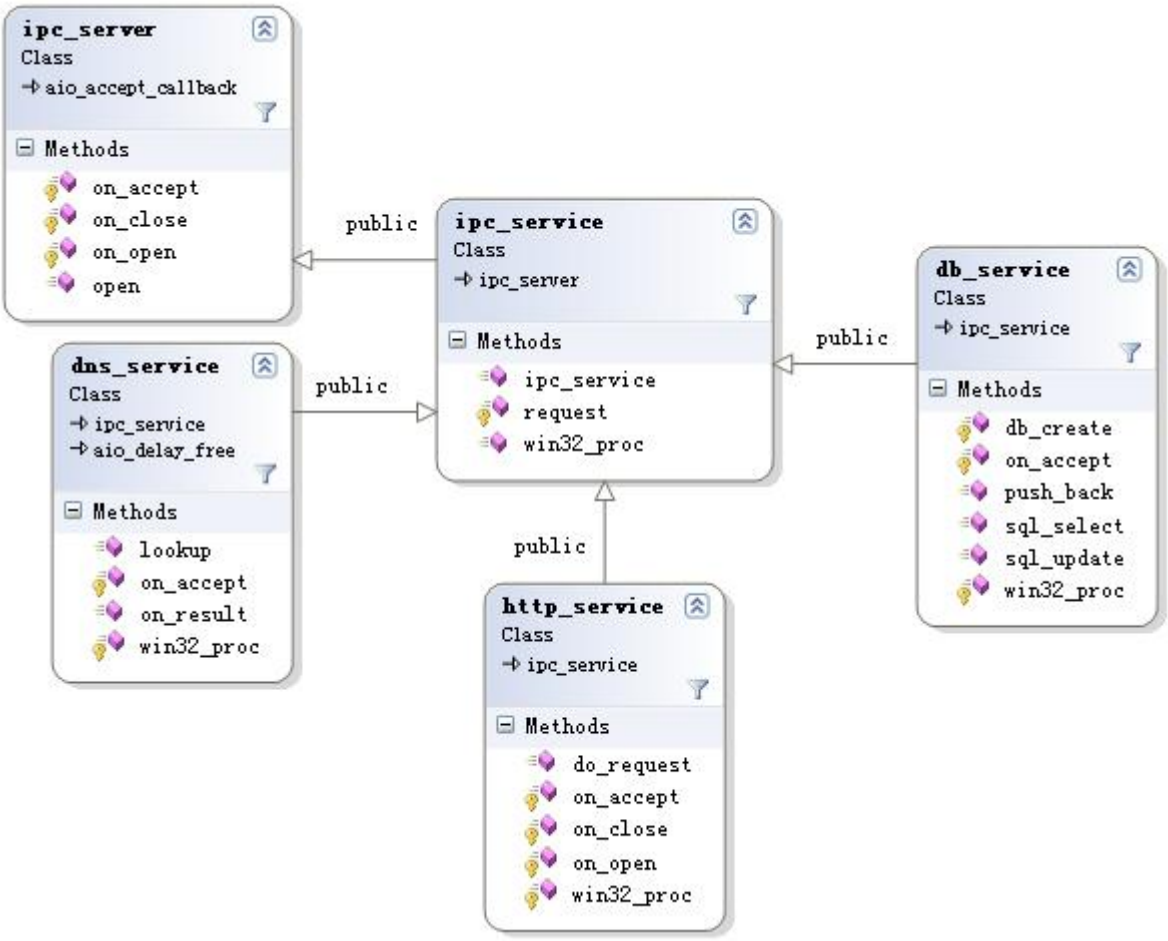
22.2 ipc_service 类：阻塞与非阻塞混合编程

发表时间: 2012-04-25

非阻塞编程主要解决了网络通讯中高并发的问题，采用非阻塞方式，服务器不必为每个连接启动单独的进程或线程，从而大大地减少了系统资源的浪费；但是现实网络应用中，阻塞应用又是不可避免的，如我们对数据库编程时使用的数据操作的客户端库本身就是阻塞的。因此，单纯的非阻塞模式或阻塞模式均不能很好地胜任互联应用，如果能够将一些必要的阻塞过程融合进非阻塞过程中将会是一个现实的需求。本文主要介绍了如何保证在使用 acl_cpp 的非阻塞框架的同时，可以把阻塞的过程与非阻塞过程进行整合。关于非阻塞编程，可以参考 acl_cpp开发--非阻塞网络编程 中的章节。

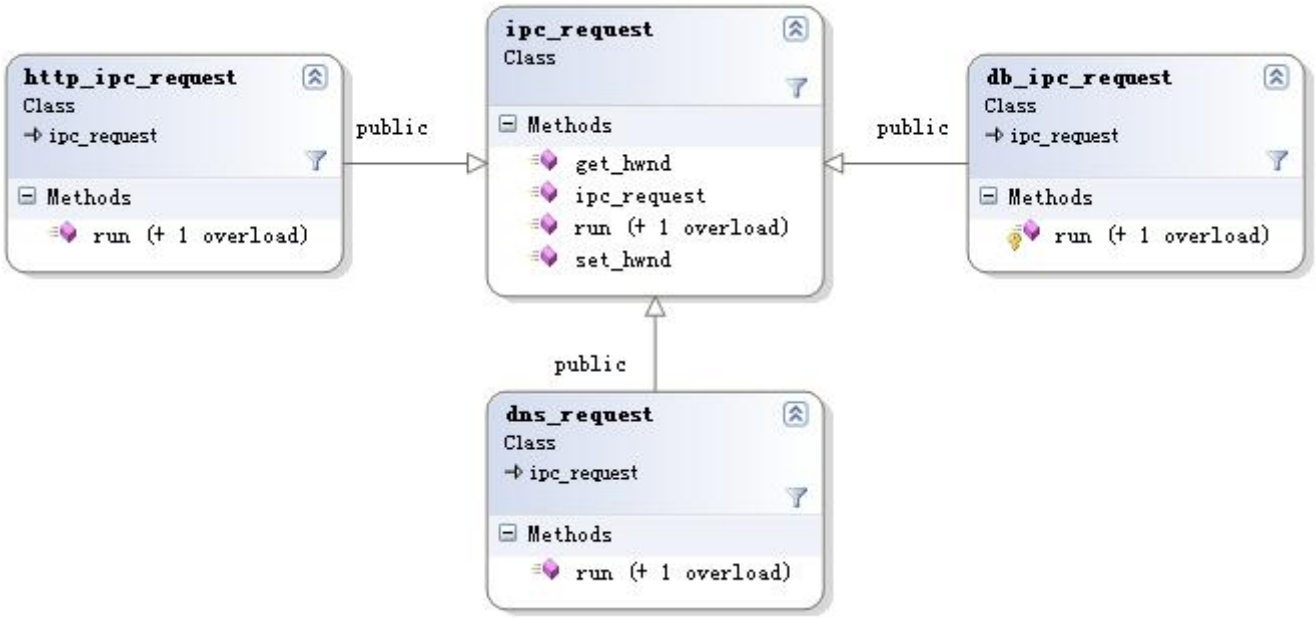
本文讨论的内容是建立在 acl_cpp 非阻塞模块的IPC通信机制 内容的基础之上，有两个基础类：ipc_service 类用来粘合阻塞与非阻塞过程，ipc_request 做为阻塞调用过程的基础类，提供了需要子线程中进行阻塞式过程的虚方法。

ipc_service 类的继承关系图如下：



可以看出，ipc_service 是从 ipc_server 类继承而来，在 ipc_service 中定义了受保护的成员方法: request(ipc_request*) ，它将被子类对象调用向子线程或线程池发送任务请求；同时在 ipc_service 中还专门针对基于 win32 消息窗口定义了虚方法：win32_proc，该虚方法被子类继承后用于主窗口过程接收来自于子线程的结果处理消息；另外，在该图中还专门封装了三个实际应用的子类 http_service、db_service、dns_service。在 ipc_service 的构造函数可以指定任务线程池中最大线程数，同时还可以指定是否采用基于 win32 窗口消息的 IPC 通信机制。

ipc_request 类的继承关系图如下：



ipc_request 的类对象在独立的子线程中运行，运行结果通过 ipc_client 通道给非阻塞的主线程发送消息（对于win2的窗口过程，可以发送窗口消息，此时 ipc_service 的类对象将会通过 win32_proc 虚方法接收消息）。另外，ipc_request 的两个重载方法: run(ipc_client*) 和 run(HWND) 都是在子线程中运行的。

程序执行的流程为：

1）在主线程中创建 ipc_service 对象实例，并监听某一端口：ipc_service::open(aio_handle*, const char* addr="127.0.0.1:0") ---> 主线程进行异步事件循环过程：while (true) { aio_handle::check(); }

2）在主线程中创建 ipc_request 对象实例，并创建需要在子线程中处理的阻塞式任务 ---> 在主线程 ipc_service 的对象中调用 ipc_service 的异步请求方法 request(ipc_request*) 将阻塞请求任务 ipc_request 传递给子线程

3）在子线程中调用 ipc_request::run 阻塞过程处理用户的请求

4）在子线程中调用 ipc_client::send_message 方法将结果数据以 ipc 消息的方式通知主线程

5）在主线程中接收到来自于子线程的结果处理消息后，将结果传递给用户过程

以下以 dns_service 为例，详细讲解 ipc_service 与 ipc_request 在主线程和子线程中的行为过程：

1）主线程中 dns_service 的主要代码如下：

```
dns_service::dns_service(int nthread /* = 1 */, bool win32_gui /* = false */)
        : ipc_service(nthread, win32_gui)
{

}

dns_service::~dns_service()
{

}

// 主线程的 dns_service 对象接收到子线程消息的 ipc 连接请求时的回调函数
void dns_service::on_accept(aio_socket_stream* client)
{
        // 创建 ipc_client 消息通道，同时用异步连接流作为通讯对象
        ipc_client* ipc = NEW dns_ipc(this);
        ipc->open(client);

        // 添加消息回调对象
        ipc->append_message(IPC_RES);
        // 异步等待来自于子线程的消息
        ipc->wait();
}

#ifdef WIN32

        // 当接收 WIN32 窗口消息时的回调过程
```

```
	void dns_service::win32_proc(HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam)
	{
		if (nMsg != IPC_RES + WM_USER)
		{
			logger_error("invalid nMsg(%d)", nMsg);
			return;
		}
		else if (lParam == 0)
		{
			logger_error("lParam invalid");
			return;
		}

		DNS_IPC_DATA* dat = (DNS_IPC_DATA*) lParam;
		dns_res* res = dat->res;

		on_result(*res);
		delete res;

		// 在采用 WIN32 消息时该对象空间是动态分配的，所以需要释放
		acl_myfree(dat);
	}

#endif

	// 用户调用此函数查询某个域名，需要查询的域名及域名查询结果
	// 均在 dns_result_callback 对象中记录
	void dns_service::lookup(dns_result_callback* callback)
	{
		// 如果发现有针对相同域名的查询过程，只需要将本查询过程
		// 下相同域名的查询过程合并，加入原来的查询过程的列表中，
		// 减少针对同一域名的查询次数
		std::list<dns_result_callback*>::iterator it;
		const char* domain = callback->get_domain().c_str();

		for (it= callbacks_.begin(); it != callbacks_.end(); it++)
		{
```

```
                if ((*it)->get_domain() == domain)
                {
                        callbacks_.push_back(callback);
                        return;
                }
        }

        callbacks_.push_back(callback);

        ipc_request* req = NEW dns_request(domain);

        // 调用基类 ipc_service 请求过程
        request(req);
}

// 查询结果的回调函数
void dns_service::on_result(const dns_res& res)
{
        std::list<dns_result_callback*>::iterator it, next;

        it= callbacks_.begin();
        for (; it != callbacks_.end();)
        {
                next = it;
                next++;
                if ((*it)->get_domain() == res.domain_.c_str())
                {
                        // 通知请求对象的解析结果
                        (*it)->on_result((*it)->get_domain(), res);
                        (*it)->destroy(); // 调用请求对象的销毁过程
                        callbacks_.erase(it);
                        it = next;
                }
                else
                        it++;
        }
}
```

2）在 dns_service::on_accept 函数中创建 ipc_client 通道的子类定义如下：

```
class dns_ipc : public ipc_client
	{
	public:
		dns_ipc(dns_service* server)
			: server_(server)
		{

		}

		~dns_ipc()
		{

		}

		// 在主线程接收到来自于子线程的消息时的回调函数
		// 因为在 dns_service::on_accept 中将该类对象
		// 与相应的消息号进行了绑定
		virtual void on_message(int nMsg acl_unused,
			void* data, int dlen acl_unused)
		{
			if (nMsg != IPC_RES)
			{
				logger_error("invalid nMsg(%d)", nMsg);
				this->close();
				return;
			}

			// 转换子线程传来的数据内容
			DNS_IPC_DATA* dat = (DNS_IPC_DATA*) data;
			dns_res* res = dat->res;
```

```
                // 调用主线程中的结果接收过程
                server_->on_result(*res);
                delete res;
        }
protected:
        virtual void on_close()
        {
                // 因为该类对象在 dns_service::on_accept 中是动态创建的，
                // 所需要当 ipc_client 通道关闭时需要自行释放所占内存
                delete this;
        }
private:
        dns_service* server_;
};
```

3）dns_request 类定义如下（该类对象是在主线程中创建的，但其中的阻塞方法：run 却是在子线程中运行的）：

```
// 由子线程传递给主线程的数据类型
struct DNS_IPC_DATA
{
        dns_res* res;
};

// 由子线程传递给主线程的消息号
typedef enum
{
        IPC_RES
};

//////////////////////////////////////////////////////////////////////
```

```
class dns_request : public ipc_request
{
public:
	dns_request(const char* domain)
		: domain_(domain)
	{

	}

	~dns_request()
	{

	}

	// 当 dns_request 类对象在 dns_service::lookup 方法中通过
	// request(ipc_request) 过程传递给子线程后，子线程便会连接
	// 主线程中 dns_service 监听的消息服务器地址，连接成功后调用
	// dns_request::run 虚方法，同时将 ipc_client 通道传入

	virtual void run(ipc_client* ipc)
	{
		// 阻塞式查询域名
		ACL_DNS_DB* db = acl_gethostbyname(domain_.c_str(), NULL);

		// 将查询结果放在自定义的结构中
		data_.res = NEW dns_res(domain_.c_str());

		if (db != NULL)
		{
			ACL_ITER iter;
			acl_foreach(iter, db)
			{
				ACL_HOSTNAME* hn = (ACL_HOSTNAME*) iter.data;
				data_.res->ips_.push_back(hn->ip);
			}
```

```
                        acl_netdb_free(db);
                }

                // 向主线程发送结果
                ipc->send_message(IPC_RES, &data_, sizeof(data_));

                // 销毁本类对象，因为其是动态分配的
                delete this;
        }

#ifdef WIN32

        // 基类虚接口，使子线程可以在执行完任务后向主线程发送 WIN32 窗口消息

        virtual void run(HWND hWnd)
        {
                ACL_DNS_DB* db = acl_gethostbyname(domain_.c_str(), NULL);
                DNS_IPC_DATA* data = (DNS_IPC_DATA*)
                        acl_mymalloc(sizeof(DNS_IPC_DATA));
                data->res = NEW dns_res(domain_.c_str());

                if (db != NULL)
                {
                        ACL_ITER iter;
                        acl_foreach(iter, db)
                        {
                                ACL_HOSTNAME* hn = (ACL_HOSTNAME*) iter.data;
                                data->res->ips_.push_back(hn->ip);
                        }

                        acl_netdb_free(db);
                }

                // 向主线程发送结果
                ::PostMessage(hWnd, IPC_RES + WM_USER, 0, (LPARAM) data);

                // 销毁本类对象，因为其是动态分配的
```

```
                        delete this;
                }
#endif
        private:
                string  domain_;
                DNS_IPC_DATA data_;
        };
```

通过以上三个类的实现便完成了主线程的非阻塞过程与子线程的阻塞过程的结合。下面是一个具体的使用如上 dns_service 类的例子：

```
#ifdef WIN32
#include "acl/lib_acl.h"
#else
#include "lib_acl.h"
#include <getopt.h>
#endif
#include <iostream>
#include "aio_handle.hpp"
#include "dns_service.hpp"

using namespace acl;

static void usage(const char* procname)
{
        printf("usage: %s -h[help] -t[use thread]\n", procname);
}

static int __ncount = 0;

class dns_result : public dns_result_callback
```

```
{
public:
        dns_result(const char* domain)
                : dns_result_callback(domain)
        {

        }

        ~dns_result()
        {

        }

        virtual void destroy()
        {
                delete this;
                __ncount--;
        }

        virtual void on_result(const char* domain,  const dns_res& res)
        {
                std::cout << "result: domain: " << domain;
                if (res.ips_.size() == 0)
                {
                        std::cout << ": null" << std::endl;
                        return;
                }

                std::cout << std::endl;

                std::list<string>::const_iterator cit = res.ips_.begin();
                for (; cit != res.ips_.end(); cit++)
                        std::cout << "\t" << (*cit).c_str();
                std::cout << std::endl;
        }
};
```

```
int main(int argc, char* argv[])
{
	int   ch, nthreads = 2;

	while ((ch = getopt(argc, argv, "ht:")) > 0)
	{
		switch (ch)
		{
		case 'h':
			usage(argv[0]);
			return (0);
		case 't':
			nthreads = atoi(optarg);
			break;
		default:
			break;
		}
	}

	acl_init();

	aio_handle handle;

	const char* domain = "www.baidu.com";
	dns_service* server = new dns_service(nthreads);

	// 使消息服务器监听 127.0.0.1 的地址
	if (server->open(&handle) == false)
	{
		delete server;
		std::cout << "open server error!" << std::endl;
		getchar();
		return (1);
	}

	// 创建查询结果接收对象，并进行查询
```

```
        dns_result* result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        domain = "www.sina.com";
        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        domain = "www.51iker.com";
        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        domain = "www.hexun.com";
        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        domain = "www.com.dummy";
        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        result = new dns_result(domain);
        server->lookup(result);
        __ncount++;

        result = new dns_result(domain);
```

```
        server->lookup(result);
        __ncount++;

        // 异步消息循环过程
        while (true)
        {
                if (handle.check() == false)
                {
                        std::cout << "stop now!" << std::endl;
                        break;
                }
                if (__ncount == 0)
                        break;
        }

        // 销毁 dns_service 动态对象
        delete server;
        // 做最后的清理工作，以释放延迟释放的连接对象
        handle.check();

        std::cout << "server stopped!" << std::endl;
        getchar();
        return (0);
}
```

示例代码：samples/aio_dns

acl_cpp 下载：http://sourceforge.net/projects/aclcpp/

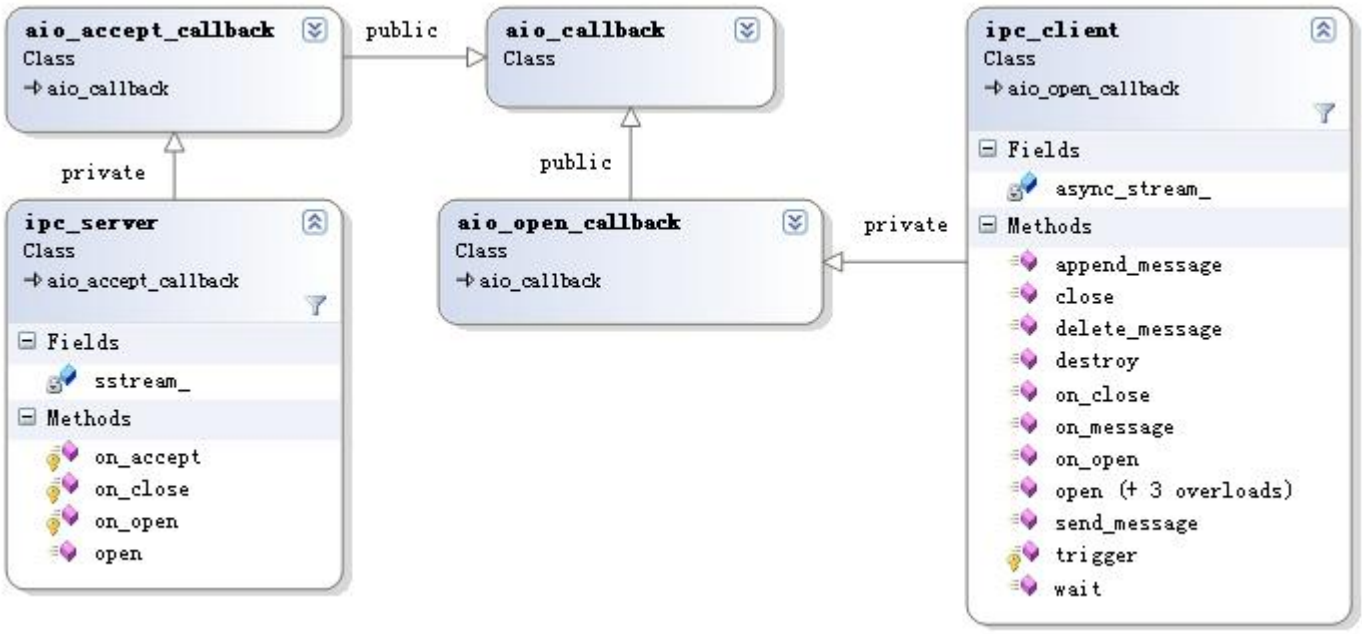
原文地址：http://zsxxsz.iteye.com/blog/1482208

更多文章：http://zsxxsz.iteye.com/

22.3 acl_cpp 非阻塞模块的IPC通信机制

发表时间: 2012-04-23

在 acl_cpp 的非阻塞框架的设计中，充分利用了操作系统平台的高并发机制，同时简化了异步编程的过程。但是，并不是所有的操作都是非阻塞的，现实的程序应用中存在着大量的阻塞式行为，acl_cpp 的非阻塞框架中设计了一种通过 ipc 模式使阻塞式函数与 acl_cpp 的非阻塞过程相结合的机制。即是说，在 acl_cpp 的主线程是非阻塞的，而把阻塞过程放在单独的一个线程中运行，当阻塞线程运算完毕，会以 ipc 方式通知主线程运行结果，这样就达到了阻塞与非阻塞相结合的模式。下图展示了 ipc 类的继承关系：



以上图中有两个类与 ipc 相关：ipc_server 及 ipc_client，其中 ipc_server 其实用于与监听异步流相关，而 ipc_client 与客户端连接流相关。在 ipc_server 类中有一个 open 函数用来打开本地监听地址，另有三个虚函数便于子类进行相关操作：

1）on_accept：当监听流获得一个客户端连接时回调此函数，将获得的客户端流传递给子类对象；

2）on_open：当用户调用 ipc_server::open 函数，且监听某一服务地址成功时通过该函数将实际的监听地址传递给子类对象（因为在调用 open(addr) 时，addr 一般仅指定 IP 地址，同时将 端口赋 0 以便于操作系统自动分配本地端口号，所以通过 on_open 便可以将实际的 端口传递给子类对象）；

3）on_close：当监听流关闭时调用此函数通知子类对象。

相对于 ipc_server 类，则 ipc_client 的接口就显得比较多，主要的函数如下：

1）open：共有四个 open 重载函数用连接监听流服务地址，其中两个是同步流方式，两个是异步流方式，一般同步建立的流用在阻塞式线程中，异步建立的流用在非阻塞线程中；

2）send_message：一般用来向非阻塞主线程发送消息；

3）on_message：异步接收到阻塞线程发来的消息的回调函数；

4）append_message：添加异步流想要接收的消息号。

ipc_client 类比较特殊，其充当着双重身份：1）作为客户端连接流连接监听流的服务地址，一般用在阻塞式线程中；2）作为监听流接收到来自于客户端流的连接请求而创建的与之对应的服务端异步流，一般用在非阻塞线程中。

下面以一个具体的实例来说明如果使用 ipc_server 及 ipc_client 两个类：

```
#include "lib_acl.h"
#include <iostream>
#include "aio_handle.hpp"
#include "ipc_server.hpp"
#include "ipc_client.hpp"

using namespace acl;

#define MSG_REQ         1
#define MSG_RES         2
#define MSG_STOP                3

// 消息客户端连接类定义
class test_client1 : public ipc_client
{
public:
        test_client1()
        {
```

```
}

~test_client1()
{

}

// 连接消息服务端地址成功后的回调函数
virtual void on_open()
{
        // 添加消息回调对象，接收消息服务器的
        // 此类消息
        this->append_message(MSG_RES);

        // 向消息服务器发送请求消息
        this->send_message(MSG_REQ, NULL, 0);

        // 异步等待来自于消息服务器的消息
        wait();
}

// 流关闭时的回调函数
virtual void on_close()
{
        delete this;
}

// 接收到消息服务器的消息时的回调函数，其中的消息号由
// append_message 进行注册
virtual void on_message(int nMsg, void*, int)
{
        std::cout << "test_client1 on message:" << nMsg << std::endl;

        // 向消息服务器发送消息，通知消息服务器停止
        this->send_message(MSG_STOP, NULL, 0);

        // 删除在 on_open 中注册的消息号
```

```
                this->delete_message(MSG_RES);

                // 本异步消息过程停止运行
                this->get_handle().stop();

                // 关闭本异步流对象
                this->close();
        }
protected:
private:
};

// 消息客户端处理过程

static bool client_main(aio_handle* handle, const char* addr)
{
        // 创建消息连接
        ipc_client* ipc = new test_client1();

        // 连接消息服务器
        if (ipc->open(handle, addr, 0) == false)
        {
                std::cout << "open " << addr << " error!" << std::endl;
                delete ipc;
                return (false);
        }

        return (true);
}

// 子线程的入口函数
static void* thread_callback(void *ctx)
{
        const char* addr = (const char*) ctx;
        aio_handle handle;

        if (client_main(&handle, addr) == false)
```

```
        {
                handle.check();
                return (NULL);
        }

        // 子线程的异步消息循环
        while (true)
        {
                if (handle.check() == false)
                        break;
        }

        // 最后清理一些可能未关闭的流连接
        handle.check();

        return (NULL);
}
```

```
// 消息服务器接收到的客户端连接流类定义
class test_client2 : public ipc_client
{
public:
        test_client2()
        {

        }

        ~test_client2()
        {

        }

        virtual void on_close()
        {
                delete this;
        }
```

```
        // 接收到消息客户端发来消息的回调函数
        virtual void on_message(int nMsg, void*, int)
        {
                std::cout << "test_client2 on message:" << nMsg << std::endl;

                // 如果收到消息客户端要求退出的消息，则主线程了退出
                if (nMsg == MSG_STOP)
                {
                        this->close();

                        // 通知主线程的非阻塞引擎关闭
                        this->get_handle().stop();
                }
                else
                        // 回应客户端消息
                        this->send_message(MSG_RES, NULL, 0);
        }
protected:
private:
};

// 主线程的消息服务器 ipc 服务监听类定义
class test_server : public ipc_server
{
public:
        test_server()
        {

        }

        ~test_server()
        {

        }

        // 消息服务器接收到消息客户端连接时的回调函数
```

```
        void on_accept(aio_socket_stream* client)
        {
                // 创建 ipc 连接对象
                ipc_client* ipc = new test_client2();

                // 打开异步IPC过程
                ipc->open(client);

                // 添加消息回调对象
                ipc->append_message(MSG_REQ);
                ipc->append_message(MSG_STOP);
                ipc->wait();
        }
protected:
private:
};

static void usage(const char* procname)
{
        printf("usage: %s -h[help] -t[use thread]\n", procname);
}

int main(int argc, char* argv[])
{
        int   ch;
        bool  use_thread = false;

        while ((ch = getopt(argc, argv, "ht")) > 0)
        {
                switch (ch)
                {
                case 'h':
                        usage(argv[0]);
                        return (0);
                case 't':
                        use_thread = true;
                        break;
```

```
                default:
                        break;
                }
        }

        acl_init();

        aio_handle handle;

        ipc_server* server = new test_server();

        // 使消息服务器监听 127.0.0.1 的地址
        if (server->open(&handle, "127.0.0.1:0") == false)
        {
                delete server;
                std::cout << "open server error!" << std::endl;
                getchar();
                return (1);
        }

        char  addr[256];
#ifdef WIN32
        _snprintf(addr, sizeof(addr), "%s", server->get_addr());
#else
        snprintf(addr, sizeof(addr), "%s", server->get_addr());
#endif

        if (use_thread)
        {
                // 使消息客户端在子线程中单独运行
                acl_pthread_t tid;
                acl_pthread_create(&tid, NULL, thread_callback, addr);
        }

        // 因为消息客户端也是非阻塞过程，所以也可以与消息服务器
        // 在同一线程中运行
        else
```

```
                client_main(&handle, addr);

        // 主线程的消息循环过程
        while (true)
        {
                if (handle.check() == false)
                {
                        std::cout << "stop now!" << std::endl;
                        break;
                }
        }

        delete server;
        handle.check();  // 清理一些可能未关闭的异步流对象

        std::cout << "server stopped!" << std::endl;
        getchar();
        return (0);
}
```

以上例子相对简单，其展示的消息服务器与消息客户端均是非阻塞过程，其实将上面的异步消息客户端稍微一改便可以改成同步消息客户端了，修改部分如下：

```
class test_client3 : public ipc_client
{
public:
        test_client3()
        {

        }
```

```
        ~test_client3()
        {

        }

        virtual void on_open()
        {
                // 添加消息回调对象
                this->append_message(MSG_RES);

                // 向消息服务器发送请求消息
                this->send_message(MSG_REQ, NULL, 0);

                // 同步等待消息
                wait();
        }

        virtual void on_close()
        {
                delete this;
        }

        virtual void on_message(int nMsg, void*, int)
        {
                std::cout << "test_client3 on message:" << nMsg << std::endl;
                this->send_message(MSG_STOP, NULL, 0);
                this->delete_message(MSG_RES);
                this->close();
        }
protected:
private:
};

// 子线程处理过程

static bool client_main(const char* addr)
{
```

```
        // 创建消息客户端对象
        ipc_client* ipc = new test_client3();

        // 同步方式连接消息服务器
        if (ipc->open(addr, 0) == false)
        {
                std::cout << "open " << addr << " error!" << std::endl;
                delete ipc;  // 当消息客户端未成功创建时需要在此处删除对象
                return (false);
        }

        return (true);
}

static void* thread_callback(void *ctx)
{
        const char* addr = (const char*) ctx;

        if (client_main(addr) == false)
                return (NULL);
        return (NULL);
}
```

对比 test_client1 与 test_client_3 两个消息客户端，可以发现二者区别并不太大，关键在于调用 open 时是采用了异步还是同步连接消息服务器，其决定了消息客户端是异步的还是同步的。

示例代码：samples/aio_ipc

acl_cpp 下载：http://sourceforge.net/projects/aclcpp/

原文地址：http://zsxxsz.iteye.com/blog/1495832

更多文章：http://zsxxsz.iteye.com/

23.1 acl_cpp 编程之 xml 流式解析与创建

发表时间: 2012-04-30 关键字: xml 解析, xml 流式解析

xml 数据格式做为当今WEB开发的重要数据格式之一，应用非常普及，在文章 <acl 之 xml 流解析器> 中，专门了 acl 库中是如何实现了流 xml 数据解析的，在 acl_cpp 库中利用 c++ 语言的特点对 acl 中的 xml 流式解析进行了进一步封装，从而更加方便用户使用，其中主要涉及到两个类：xml 类和 xml_node 类，现在分别就这两个类的函数功能做一简单介绍。

一、解析过程中的用法

1、 xml 类中的主要方法如下：

```
/**
 * 以流式方式循环调用本函数添加 XML 数据，也可以一次性添加
 * 完整的 XML 数据，如果是重复使用该 XML 解析器解析多个 XML
 * 对象，则应该在解析下一个 XML 对象前调用 reset() 方法来清
 * 除上一次的解析结果
 * @param data {const char*} xml 数据
 */
void update(const char* data);
```

```
/**
 * 从 XML 对象中取得某个标签名的所有结点集合
 * @param tag {const char*} 标签名(不区分大小写)
 * @return {const std::vector<xml_node*>&} 返回结果集的对象引用，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 *  注：返回的数组中的 xml_node 结点数据可以修改，但不能删除该结点，
 *  因为该库内部有自动删除的机制
 */
const std::vector<xml_node*>& getElementsByTagName(const char* tag) const;



/**
 * 从 xml 对象中获得所有的与给定多级标签名相同的 xml 结点的集合
```

```
 * @param tags {const char*} 多级标签名，由 '/' 分隔各级标签名，如针对 xml 数据：
 *  <root> <first> <second> <third name="test1"> text1 </third> </second> </first> ...
 *  <root> <first> <second> <third name="test2"> text2 </third> </second> </first> ...
 *  <root> <first> <second> <third name="test3"> text3 </third> </second> </first> ...
 *  可以通过多级标签名：root/first/second/third 一次性查出所有符合条件的结点
 * @return {const std::vector<xml_node*>&} 符合条件的 xml 结点集合，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 *  注：返回的数组中的 xml_node 结点数据可以修改，但不能删除该结点，
 *  因为该库内部有自动删除的机制
 */
const std::vector<xml_node*>& getElementsByTags(const char* tags) const;

/**
 * 从 xml 对象中获得所有的与给定属性名 name 的属性值相同的 xml 结点元素集合
 * @param name {const char*} 属性名为 name 的属性值
 * @return {const std::vector<xml_node*>&} 返回结果集的对象引用，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 *  注：返回的数组中的 xml_node 结点数据可以修改，但不能删除该结点，
 *  因为该库内部有自动删除的机制
 */
const std::vector<xml_node*>& getElementsByName(const char* value) const;

/**
 * 从 xml 对象中获得所有给定属性名及属性值的 xml 结点元素集合
 * @param name {const char*} 属性名
 * @param value {const char*} 属性值
 * @return {const std::vector<xml_node*>&} 返回结果集的对象引用，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 */
const std::vector<xml_node*>& getElementsByAttr(const char* name, const char* value) const;

/**
 * 从 xml 对象中获得指定 id 值的 xml 结点元素
 * @param id {const char*} id 值
 * @return {const xml_node*} xml 结点元素，若返回 NULL 则表示没有符合
 *  条件的 xml 结点，返回值不需要释放
```

```
 */
const xml_node* getElementById(const char* id) const;
```

```
/**
 * 开始遍历该 xml 对象并获得第一个结点
 * @return {xml_node*} 返回空表示该 xml 对象为空结点
 *  注：返回的结点对象用户不能手工释放，因为该对象被
 *  内部库自动释放
 */
xml_node* first_node(void);

/**
 * 遍历该 xml 对象的下一个 xml 结点
 * @return {xml_node*} 返回空表示遍历完毕
 *  注：返回的结点对象用户不能手工释放，因为该对象被
 *  内部库自动释放
 */
xml_node* next_node(void);
```

2、xml_node 类中的主要方法

```
/**
 * 取得本 XML 结点的标签名
 * @return {const char*} 返回 XML 结点标签名，如果返回空，则说明
 *  不存在标签？xxxx，以防万一，调用者需要判断返回值
 */
const char* tag_name(void) const;

/**
 * 如果该 XML 结点的 ID 号属性不存在，则返回空指针
 * @return {const char*} 当 ID 属性存在时返回对应的值，否则返回空
 */
const char* id(void) const;
```

```
/**
 * 返回该 XML 结点的正文内容
 * @return {const char*} 返回空说明没有正文内容
 */
const char* text(void) const;

/**
 * 返回该 XML 结点的某个属性值
 * @param name {const char*} 属性名
 * @return {const char*} 属性值，如果返回空则说明该属性不存在
 */
const char* attr_value(const char* name) const;

/**
 * 遍历结点的所有属性时，需要调用此函数来获得第一个属性对象
 * @return {const xml_attr*} 返回第一个属性对象，若为空，则表示
 *  该结点没有属性
 */
const xml_attr* first_attr(void) const;

/**
 * 遍历结点的所有属性时，调用本函数获得下一个属性对象
 * @return {const xml_attr*} 返回下一下属性对象，若为空，则表示
 *  遍历完毕
 */
const xml_attr* next_attr(void) const;
```

```
/**
 * 获得本结点的父级结点对象的引用
 * @return {xml_node&}
 */
xml_node& get_parent(void) const;

/**
 * 获得本结点的第一个子结点，需要遍历子结点时必须首先调用此函数
 * @return {xml_node*} 返回空表示没有子结点
 */
```

```
xml_node* first_child(void);

/**
 * 获得本结点的下一个子结点
 * @return {xml_node*} 返回空表示遍历过程结束
 */
xml_node* next_child(void);

/**
 * 返回该 xml 结点的下一级子结点的个数
 * @return {int} 永远 >= 0
 */
int   children_count(void) const;
```

上面列出的函数接口比较多，虽然还并未列出全部，用户在用时不免被这么多接口搞晕，下面就写一个简单的例子也说明如何使用这两个类。

```
#include <vector>
#include "xml.hpp"

static void test1(void)
{
        const char *data =
                "<?xml version=\"1.0\"?>\r\n"
                "<?xml-stylesheet type=\"text/xsl\"\r\n"
                "       href=\"http://docbook.sourceforge.net/release/xsl/current/manpages/docb
                "<!DOCTYPE refentry PUBLIC \"-//OASIS//DTD DocBook XML V4.1.2//EN\"\r\n"
                "       \"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd\" [\r\n"
                "       <!ENTITY xmllint \"<command>xmllint</command>\">\r\n"
                "]>\r\n"
                "<root>test\r\n"
                "       <!-- <edition> - <!--0.5--> - </edition> -->\r\n"
                "       <user name = user_name>zsx\r\n"
                "               <age>38</age>\r\n"
```

```
                "       </user>\r\n"
                "</root>\r\n"
                "<!-- <edition><!-- 0.5 --></edition> -->\r\n"
                "<!-- <edition>0.5</edition> -->\r\n"
                "<!-- <edition> -- 0.5 -- </edition> -->\r\n"
                "<root name='root' id='root_id'>test</root>\r\n";

        acl::xml xml;  // xml 解析器对象定义

        xml.update(data);  // 将 xml 数据输入并进行解析

        // 根据 xml 标签名获得所有相应的 xml 结点对象
        const std::vector<acl::xml_node*>& elements = xml.getElementsByTagName("user");

        if (!elements.empty()) {
                // 遍历查询结果集
                std::vector<acl::xml_node*>::const_iterator cit = elements.begin();
                for (; cit != elements.end(); cit++) {
                        acl::xml_node *node = *cit;
                        printf("tagname: %s, text: %s\n", node->tag_name() ? node->tag_name() :
                                node->text() ? node->text() : "");

                        // 遍历一个结点的所有属性
                        const acl::xml_attr* attr = (*cit)->first_attr();  // 取得结点的第一个属
                        while (attr)
                        {
                                printf("test1: %s=%s\r\n", attr->get_name(), attr->get_value()
                                attr = (*cit)->next_attr();  // 取得结点的下一个属性
                        }
                }
        }
}
```

上面的例子中是一次性将 xml 数据传入给 acl::xml 解析器进行解析的，当然也可以采用如下的方法：

```
const char* ptr;
	char  buf[2];
	ptr = data;

	while (*ptr) {
		buf[0] = *ptr++;
		buf[1] = 0;
		xml.update(buf);
	}
```

当然每次传给xml解析器一个字节效率比较低，这只是展示 acl_cpp 中的 xml 的流式解析器的特点，这对于网络通信中尤其是 HTTP 数据流中针对 xml 数据流的解析比较有帮助。

另外，xml 解析器还给出一个用于遍历所有 xml 结点对象的函数：first_node 和 next_node，通过这两个函数可以获得一个完整的 xml 树的所有结点，示例如下：

```
acl::xml xml;
	...
	acl::xml_node* node = xml.first_node(); // 取得第一个 xml 结点
	while (node) {
		printf("tag: %s\r\n", node->tag_name());
		node = xml.next_node(); // 取得下一个 xml 结点
	}
```

不仅 xml 树对象有遍历的功能函数，xml_node 结点对象也有遍历其下一级子结点的功能函数，示例如下：

```
acl::xml xml;
	...
	acl::xml_node* node = xml.first_node();  //取得 xml 对象的第一个xml_node 结点
	if (node) {
```

```
                acl::xml_node* child = node->first_child();  // 取得该 xml_node 结点的第一个第一
                while (child) {
                        printf("child tag: %s\r\n", child->tag_name());
                        child = node->next_child();  // 取得该 xml_node 结点的下一下第一级子结点
                }
        }
```

二、生成 xml 字符串的用法

为了便于生成 xml 对象，acl_cpp 的 xml 模块增加了相应的函数接口用于生成 xml 数据流，下面介绍如何生成 xml 数据流。

1、在 xml 类中相关函数接口：

```
/**
 * 创建一个 xml_node 结点对象
 * @param tag {const char*} 标签名
 * @param text {const char*} 文本字符串
 * @return {xml_node*} 新产生的 xml_node 对象不需要用户手工释放，因为在
 *  xml 对象被释放时这些结点会自动被释放，当然用户也可以在不用时调用
 *  reset 来释放这些 xml_node 结点对象
 */
xml_node& create_node(const char* tag, const char* text = NULL);

/**
 * 获得根结点对象
 * @return {xml_node&}
 */
xml_node& get_root();
```

在 xml 解析器中，有一个虚拟的 xml 根结点，这个结点本身不存任何 xml 数据，只是所有的 xml_node 结点都属于这个根结点的子孙结点而已。

2、在 xml_node 类中相关函数接口：

```
/**
 * 添加 XML 结点属性
 * @param name {const char*} 属性名
 * @param value {const char*} 属性值
 * @return {xml_node&}
 */
xml_node& add_attr(const char* name, const char* value);

/**
 * 设置 xml 结点的文本内容
 * @param str {const char*} 字符串内容
 * @return {xml_node&}
 */
xml_node& set_text(const char* str);
```

```
/**
 * 给本 xml 结点添加 xml_node 子结点对象
 * @param child {xml_node*} 子结点对象
 * @return {xml_node&} return_child 为 true 返回子结点的引用，
 *  否则返回本 xml 结点引用
 */
xml_node& add_child(xml_node* child, bool return_child = false);

/**
 * 给本 xml 结点添加 xml_node 子结点对象
 * @param child {xml_node&} 子结点对象
 * @return {xml_node&} return_child 为 true 返回子结点的引用，
 *  否则返回本 xml 结点引用
 */
xml_node& add_child(xml_node& child, bool return_child = false);

/**
```

```
 * 给本 xml 结点添加 xml_node 子结点对象
 * @param tag {const char* tag} 子结点对象的标签名
 * @return {xml_node&} return_child 为 true 返回子结点的引用，
 * @param str {const char*} 文本字符串
 *  否则返回本 xml 结点引用
 */
xml_node& add_child(const char* tag, bool return_child = false,
        const char* str = NULL);
```

下面举几个简单的例子来说明如何生成 xml 数据流：

```
acl::xml xml;
```

```
acl::xml_node& root = xml.get_root();  // 获得 xml 的根结点
     acl::xml_node* node1, *node2, *node11;

     // 创建一个 xml_node 结点
     node1 = &xml.create_node("test1");
     // 给 node1 结点添加属性值
     (*node1).add_attr("name1_1", "value1_1")
             .add_attr("name1_2", "value1_2")
             .add_attr("name1_3", "value1_3");
     // 将 node1 做为 xml 根结点的第一个子结点
     root.add_child(node1);

     // 创建一个 xml_node 结点
     node11 = &xml.create_node("test11");
     // 给 node11 结点添加属性值
     (*node11).add_attr("name11_1", "value11_1")
             .add_attr("name11_2", "value11_2")
             .add_attr("name11_3", "value11_3");
     // 将 node11 做为 node1 根结点的第一个子结点
     node1.add_child(node11);
```

```
	// 创建一个 xml_node 结点
	node2 = &xml.create_node("test2");
	// 给 node2 结点添加属性值
	(*node2).add_attr("name2_1", "value2_1")
		.add_attr("name2_2", "value2_2")
		.add_attr("name2_3", "value2_3");
	// 将 node2 做为 xml 根结点的第二个子结点
	root.add_child(node2);

	acl::string buf("<?xml version=\"1.0\"?>");
	xml.build_xml(buf);  // 生成 xml 数据流，注：在 函数 build_xml 内部对于缓冲区 buf 的处理方式

	printf("xml: %s\r\n", buf.c_str()); // 打印生成的 xml 数据
```

其实，上面的示例还有一个更加简洁的写法，如下：

```
acl::xml_node& root = xml.get_root();  // 获得 xml 的根结点
	acl::xml_node* node1, *node2, *node11;

	// 创建一个 xml_node 结点
	xml.get_root()
		.add_child("test1", true)  // 因第二个参数为 true，所以 add_child 函数返回新创建子
			.add_attr("name1_1", "value1_1")  // 给 test1 结点添加属性
			.add_attr("name1_2", "value1_2")
			.add_attr("name1_3", "value1_3");
			.add_child("test11", true)  // 给 test1 结点添加一个标签值为 test11 的子结
				.add_attr("name11_1", "value11_1")  // 给 test11 子结点添加属性
				.add_attr("name11_2", "value11_2")
				.add_attr("name11_3", "value11_3");
				.get_parent()  // 返回 test11 结点的父结点的引用，即返回 test1 结点
			.get_parent()  // 返回 test1 结点的引用即返回 xml 的 root 结点
		.add_child("test2", true)  // 给 xml 根结点添加 test2 子结点
```

```
                .add_attr("name2_1", "value2_1")  // 给 test2 子结点添加属性
                .add_attr("name2_2", "value2_2")
                .add_attr("name2_3", "value2_3");
```

可以看出，第二种写法更加简洁有效，同时逻辑关系更为清晰，有种一气呵成的感觉，呵呵。当然，读者可以根据自己的习惯使用其中任意一种写法。另外，大家仔细查看 xml_node 类的声明可能会看出，该类的构造函数和析构函数是私有的，这意味着用户不能使用 new 或delete 来手工创建和销毁 xml_node 类对象，同时不能如 acl::xml_node node 这样定义对象，这就说，xml_node 对象只能是由 acl::xml 类对象或 acl::xml_node 类对象来创建，同时对所有 xml_node 类对象的销毁都是在 acl::xml 类对象内部自动完成的，如果用户想在 acl::xml 类对象销毁之前提前销毁所有的 acl::xml_node 类对象，则用户可以手工调用 acl::xml 类中的 reset() 方法来达到此目的。

使用 xml 的例子在：samples/xml 目录下

acl_cpp 下载：http://sourceforge.net/projects/aclcpp/

原文地址：http://zsxxsz.iteye.com/blog/1505882

更多文章：http://zsxxsz.iteye.com/

23.2 acl_cpp 编程之 xml 流式解析与创建

发表时间: 2012-05-02 关键字: acl_cpp

xml 数据格式做为当今WEB开发的重要数据格式之一，应用非常普及，在文章 <acl 之 xml 流解析器> 中， 专门讲述了 acl 库中是如何实现流式 xml 数据解析的，在 acl_cpp 库中利用 c++ 语言特点对 acl 中的 xml 流式解析进行了进一步封装，从而更加方便用户使用，其中主要涉及到两个类：xml 类和 xml_node 类，现在分别就这两个类的函数功能做一简单介绍。

一、解析过程中的用法

1、 xml 类中的主要方法如下：

```
/**
 * 以流式方式循环调用本函数添加 XML 数据，也可以一次性添加
 * 完整的 XML 数据，如果是重复使用该 XML 解析器解析多个 XML
 * 对象，则应该在解析下一个 XML 对象前调用 reset() 方法来清
 * 除上一次的解析结果
 * @param data {const char*} xml 数据
 */
void update(const char* data);
```

```
/**
 * 从 XML 对象中取得某个标签名的所有结点集合
 * @param tag {const char*} 标签名(不区分大小写)
 * @return {const std::vector<xml_node*>&} 返回结果集的对象引用，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 *  注：返回的数组中的 xml_node 结点数据可以修改，但不能删除该结点，
 *  因为该库内部有自动删除的机制
 */
const std::vector<xml_node*>& getElementsByTagName(const char* tag) const;



/**
 * 从 xml 对象中获得所有的与给定多级标签名相同的 xml 结点的集合
```

```
 * @param tags {const char*} 多级标签名，由 '/' 分隔各级标签名，如针对 xml 数据：
 *  <root> <first> <second> <third name="test1"> text1 </third> </second> </first> ...
 *  <root> <first> <second> <third name="test2"> text2 </third> </second> </first> ...
 *  <root> <first> <second> <third name="test3"> text3 </third> </second> </first> ...
 *  可以通过多级标签名：root/first/second/third 一次性查出所有符合条件的结点
 * @return {const std::vector<xml_node*>&} 符合条件的 xml 结点集合，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 *  注：返回的数组中的 xml_node 结点数据可以修改，但不能删除该结点，
 *  因为该库内部有自动删除的机制
 */
const std::vector<xml_node*>& getElementsByTags(const char* tags) const;

/**
 * 从 xml 对象中获得所有的与给定属性名 name 的属性值相同的 xml 结点元素集合
 * @param name {const char*} 属性名为 name 的属性值
 * @return {const std::vector<xml_node*>&} 返回结果集的对象引用，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 *  注：返回的数组中的 xml_node 结点数据可以修改，但不能删除该结点，
 *  因为该库内部有自动删除的机制
 */
const std::vector<xml_node*>& getElementsByName(const char* value) const;

/**
 * 从 xml 对象中获得所有给定属性名及属性值的 xml 结点元素集合
 * @param name {const char*} 属性名
 * @param value {const char*} 属性值
 * @return {const std::vector<xml_node*>&} 返回结果集的对象引用，
 *  如果查询结果为空，则该集合为空，即：empty() == true
 */
const std::vector<xml_node*>& getElementsByAttr(const char* name, const char* value) const;

/**
 * 从 xml 对象中获得指定 id 值的 xml 结点元素
 * @param id {const char*} id 值
 * @return {const xml_node*} xml 结点元素，若返回 NULL 则表示没有符合
 *  条件的 xml 结点，返回值不需要释放
```

```
 */
const xml_node* getElementById(const char* id) const;
```

```
/**
 * 开始遍历该 xml 对象并获得第一个结点
 * @return {xml_node*} 返回空表示该 xml 对象为空结点
 *  注：返回的结点对象用户不能手工释放，因为该对象被
 *  内部库自动释放
 */
xml_node* first_node(void);

/**
 * 遍历该 xml 对象的下一个 xml 结点
 * @return {xml_node*} 返回空表示遍历完毕
 *  注：返回的结点对象用户不能手工释放，因为该对象被
 *  内部库自动释放
 */
xml_node* next_node(void);
```

2、xml_node 类中的主要方法

```
/**
 * 取得本 XML 结点的标签名
 * @return {const char*} 返回 XML 结点标签名，如果返回空，则说明
 *  不存在标签？xxxx，以防万一，调用者需要判断返回值
 */
const char* tag_name(void) const;

/**
 * 如果该 XML 结点的 ID 号属性不存在，则返回空指针
 * @return {const char*} 当 ID 属性存在时返回对应的值，否则返回空
 */
const char* id(void) const;
```

```
/**
 * 返回该 XML 结点的正文内容
 * @return {const char*} 返回空说明没有正文内容
 */
const char* text(void) const;

/**
 * 返回该 XML 结点的某个属性值
 * @param name {const char*} 属性名
 * @return {const char*} 属性值，如果返回空则说明该属性不存在
 */
const char* attr_value(const char* name) const;

/**
 * 遍历结点的所有属性时，需要调用此函数来获得第一个属性对象
 * @return {const xml_attr*} 返回第一个属性对象，若为空，则表示
 *  该结点没有属性
 */
const xml_attr* first_attr(void) const;

/**
 * 遍历结点的所有属性时，调用本函数获得下一个属性对象
 * @return {const xml_attr*} 返回下一下属性对象，若为空，则表示
 *  遍历完毕
 */
const xml_attr* next_attr(void) const;
```

```
/**
 * 获得本结点的父级结点对象的引用
 * @return {xml_node&}
 */
xml_node& get_parent(void) const;

/**
 * 获得本结点的第一个子结点，需要遍历子结点时必须首先调用此函数
 * @return {xml_node*} 返回空表示没有子结点
 */
```

```
xml_node* first_child(void);

/**
 * 获得本结点的下一个子结点
 * @return {xml_node*} 返回空表示遍历过程结束
 */
xml_node* next_child(void);

/**
 * 返回该 xml 结点的下一级子结点的个数
 * @return {int} 永远 >= 0
 */
int   children_count(void) const;
```

上面列出的函数接口比较多，还有一些未列出，用户在用时不免会被这么多接口搞晕，下面就写一个简单的例子说明如何使用这两个类。

```
#include <vector>
#include "xml.hpp"

static void test1(void)
{
        const char *data =
                "<?xml version=\"1.0\"?>\r\n"
                "<?xml-stylesheet type=\"text/xsl\"\r\n"
                "       href=\"http://docbook.sourceforge.net/release/xsl/current/manpages/docb
                "<!DOCTYPE refentry PUBLIC \"-//OASIS//DTD DocBook XML V4.1.2//EN\"\r\n"
                "       \"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd\" [\r\n"
                "       <!ENTITY xmllint \"<command>xmllint</command>\">\r\n"
                "]>\r\n"
                "<root>test\r\n"
                "       <!-- <edition> - <!--0.5--> - </edition> -->\r\n"
                "       <user name = user_name>zsx\r\n"
                "               <age>38</age>\r\n"
```

```
                "       </user>\r\n"
                "</root>\r\n"
                "<!-- <edition><!-- 0.5 --></edition> -->\r\n"
                "<!-- <edition>0.5</edition> -->\r\n"
                "<!-- <edition> -- 0.5 -- </edition> -->\r\n"
                "<root name='root' id='root_id'>test</root>\r\n";

        acl::xml xml;  // xml 解析器对象定义

        xml.update(data);  // 将 xml 数据输入并进行解析

        // 根据 xml 标签名获得所有相应的 xml 结点对象
        const std::vector<acl::xml_node*>& elements = xml.getElementsByTagName("user");

        if (!elements.empty()) {
                // 遍历查询结果集
                std::vector<acl::xml_node*>::const_iterator cit = elements.begin();
                for (; cit != elements.end(); cit++) {
                        acl::xml_node *node = *cit;
                        printf("tagname: %s, text: %s\n", node->tag_name() ? node->tag_name() :
                                node->text() ? node->text() : "");

                        // 遍历一个结点的所有属性
                        const acl::xml_attr* attr = (*cit)->first_attr();  // 取得结点的第一个属
                        while (attr)
                        {
                                printf("test1: %s=%s\r\n", attr->get_name(), attr->get_value()
                                attr = (*cit)->next_attr();  // 取得结点的下一个属性
                        }
                }
        }
}
```

上面的例子中是一次性将 xml 数据传给 acl::xml 解析器进行解析的，当然也可以采用如下的方法：

```
   const char* ptr;
        char  buf[2];
```

```
        ptr = data;

        while (*ptr) {
                buf[0] = *ptr++;
                buf[1] = 0;
                xml.update(buf);
        }
```

每次传给xml解析器一个字节的解析效率比较低，这只是展示 acl_cpp 中的 xml 的流式解析器的特点，这对于网络通信中尤其是 HTTP 数据流中针对 xml 数据流的解析比较有帮助。

另外，xml 解析器还给出一个用于遍历所有 xml 结点对象的函数：first_node 和 next_node，通过这两个函数可以获得一个完整的 xml 树的所有结点，示例如下：

```
acl::xml xml;
        ...
        acl::xml_node* node = xml.first_node(); // 取得第一个 xml 结点
        while (node) {
                printf("tag: %s\r\n", node->tag_name());
                node = xml.next_node(); // 取得下一个 xml 结点
        }
```

不仅 xml 树对象有遍历的功能函数，xml_node 结点对象也有遍历其下一级子结点的功能函数，示例如下：

```
acl::xml xml;
        ...
        acl::xml_node* node = xml.first_node();  //取得 xml 对象的第一个xml_node 结点
        if (node) {
                acl::xml_node* child = node->first_child();  // 取得该 xml_node 结点的第一个第一
                while (child) {
```

```
                        printf("child tag: %s\r\n", child->tag_name());
                        child = node->next_child();  // 取得该 xml_node 结点的下一下第一级子结点
                }
        }
```

二、生成 xml 字符串的用法

为了便于生成 xml 对象，acl_cpp 的 xml 模块增加了相应的函数接口用于生成 xml 数据流，下面介绍如何生成 xml 数据流。

1、在 xml 类中相关函数接口：

```
/**
 * 创建一个 xml_node 结点对象
 * @param tag {const char*} 标签名
 * @param text {const char*} 文本字符串
 * @return {xml_node*} 新产生的 xml_node 对象不需要用户手工释放，因为在
 *  xml 对象被释放时这些结点会自动被释放，当然用户也可以在不用时调用
 *  reset 来释放这些 xml_node 结点对象
 */
xml_node& create_node(const char* tag, const char* text = NULL);

/**
 * 获得根结点对象
 * @return {xml_node&}
 */
xml_node& get_root();
```

在 xml 解析器中，有一个虚拟的 xml 根结点，这个结点本身不存任何 xml 数据，但所有的 xml_node 结点都属于这个根结点的子结点。

2、在 xml_node 类中相关函数接口：

```
/**
 * 添加 XML 结点属性
 * @param name {const char*} 属性名
 * @param value {const char*} 属性值
 * @return {xml_node&}
 */
xml_node& add_attr(const char* name, const char* value);

/**
 * 设置 xml 结点的文本内容
 * @param str {const char*} 字符串内容
 * @return {xml_node&}
 */
xml_node& set_text(const char* str);
```

```
/**
 * 给本 xml 结点添加 xml_node 子结点对象
 * @param child {xml_node*} 子结点对象
 * @return {xml_node&} return_child 为 true 返回子结点的引用，
 *  否则返回本 xml 结点引用
 */
xml_node& add_child(xml_node* child, bool return_child = false);

/**
 * 给本 xml 结点添加 xml_node 子结点对象
 * @param child {xml_node&} 子结点对象
 * @return {xml_node&} return_child 为 true 返回子结点的引用，
 *  否则返回本 xml 结点引用
 */
xml_node& add_child(xml_node& child, bool return_child = false);

/**
 * 给本 xml 结点添加 xml_node 子结点对象
 * @param tag {const char* tag} 子结点对象的标签名
```

```
 * @return {xml_node&} return_child 为 true 返回子结点的引用，
 * @param str {const char*} 文本字符串
 *  否则返回本 xml 结点引用
 */
xml_node& add_child(const char* tag, bool return_child = false,
        const char* str = NULL);
```

下面举几个简单的例子来说明如何生成 xml 数据流：

```
acl::xml xml;
```

```
acl::xml_node& root = xml.get_root();  // 获得 xml 的根结点
      acl::xml_node* node1, *node2, *node11;

      // 创建一个 xml_node 结点
      node1 = &xml.create_node("test1");
      // 给 node1 结点添加属性值
      (*node1).add_attr("name1_1", "value1_1")
              .add_attr("name1_2", "value1_2")
              .add_attr("name1_3", "value1_3");
      // 将 node1 做为 xml 根结点的第一个子结点
      root.add_child(node1);

      // 创建一个 xml_node 结点
      node11 = &xml.create_node("test11");
      // 给 node11 结点添加属性值
      (*node11).add_attr("name11_1", "value11_1")
              .add_attr("name11_2", "value11_2")
              .add_attr("name11_3", "value11_3");
      // 将 node11 做为 node1 根结点的第一个子结点
      node1.add_child(node11);

      // 创建一个 xml_node 结点
      node2 = &xml.create_node("test2");
```

```
	// 给 node2 结点添加属性值
	(*node2).add_attr("name2_1", "value2_1")
		.add_attr("name2_2", "value2_2")
		.add_attr("name2_3", "value2_3");
	// 将 node2 做为 xml 根结点的第二个子结点
	root.add_child(node2);

	acl::string buf("<?xml version=\"1.0\"?>");
	xml.build_xml(buf);  // 生成 xml 数据流，注：在 函数 build_xml 内部对于缓冲区 buf 的处理方

	printf("xml: %s\r\n", buf.c_str()); // 打印生成的 xml 数据
```

其实，上面的示例还有一个更加简洁的写法，如下：

```
acl::xml_node& root = xml.get_root();  // 获得 xml 的根结点
	acl::xml_node* node1, *node2, *node11;

	// 创建一个 xml_node 结点
	xml.get_root()
		.add_child("test1", true)  // 因第二个参数为 true，所以 add_child 函数返回新创建子
			.add_attr("name1_1", "value1_1")  // 给 test1 结点添加属性
			.add_attr("name1_2", "value1_2")
			.add_attr("name1_3", "value1_3");
			.add_child("test11", true)  // 给 test1 结点添加一个标签值为 test11 的子结
				.add_attr("name11_1", "value11_1")  // 给 test11 子结点添加属性
				.add_attr("name11_2", "value11_2")
				.add_attr("name11_3", "value11_3");
				.get_parent()  // 返回 test11 结点的父结点的引用，即返回 test1 结
			.get_parent()  // 返回 test1 结点的引用即返回 xml 的 root 结点
		.add_child("test2", true)  // 给 xml 根结点添加 test2 子结点
			.add_attr("name2_1", "value2_1")  // 给 test2 子结点添加属性
			.add_attr("name2_2", "value2_2")
			.add_attr("name2_3", "value2_3");
```

可以看出，第二种写法更加简洁有效，同时逻辑关系更为清晰，有种一气呵成的感觉，呵呵。当然，读者可以根据自己的习惯使用其中任意一种写法。另外，大家仔 细查看 xml_node 类的声明可能会看出，该类的构造函数和析构函数是私有的，这意味着用户不能使用 new 或delete 来手工创建和销毁 xml_node 类对象，同时不能如 acl::xml_node node 这样定义对象，这就说，xml_node 对象只能是由 acl::xml 类对象或 acl::xml_node 类对象来创建，同时对所有 xml_node 类对象的销毁都是在 acl::xml 类对象内部自动完成的，即当 xml 对象销毁时，这些内部动态创建的 xml_node 结点会被自动销毁；如果用户想在 acl::xml 类对象销毁之前提前销毁所有的 acl::xml_node 类对象，则用户可以手工调用 acl::xml类中的 reset() 方法来达到此目的。

使用 xml 的例子在：samples/xml 目录下

acl_cpp 下载：http://sourceforge.net/projects/aclcpp/

原文地址：http://zsxxsz.iteye.com/blog/1506643

更多文章：http://zsxxsz.iteye.com/

24.1 编程杂谈

发表时间: 2012-05-08

一、好的编码习惯：代码整齐度（行长，函数长度，缩进数），恰当的注释（函数名，变量名，类名，复杂算法），命名规范（名字易懂，长短适当）

二、多用 const 及写时拷贝技术

三、清晰的类继承关系，尽量避免多继承，可以通过添加成员方式来解决

四、有效的前置式类声明，可以减少头文件引用

五、尽量少地使用向下硬转换

六、缺省参数出现顺序需要考虑使用频率

七、多考虑接口的可扩展性，同时需要在易用性做折中

八、变参接口设计时尽量不同名

九、成员类对象的构造的快速构造

十、更多的测试用例，尤其是边界测试用例

十一、多使用成熟的工具（如 valgrind）检查运行实例

十二、通用化设计与特殊化设计的结合

十三、要有代码迭代习惯

十四、合理使用单例模式，避免单例陷阱

十五、不要在头文件中使用 using namespace 语句，以免所使用的不同的库发生命名冲突

25.1 用C++实现类似于JAVA HttpServlet 的编程接口

发表时间: 2012-05-18

本人自99年加入263开始接触网络编程，当初最早做的项目都是用 C 语言写一大堆 CGI，可谓是偿尽了编程的苦，因为那时国内的技术水平普遍比较低，如果你会 CGI 编程，就已经算是行业中人了，如果你对 CGI 编程比较熟练，则就可以称得是“专家”了，后来技术不断进步，各种国外的新技术都进入中国并不断得到普及，CGI 就逐渐沦为一种落后的技术，后来的 PHP, JSP/Servlet, ASP 逐渐占领了 WEB 编程的技术市场，这个时候如果你说再用 C 写 CGI，别人会感觉是在和古人对话。现在主流的 WEB 开发语言一个很大的优势就是有各种相对成熟的基础库和框架，开发效率很高，而 CGI 则就逊色很多。当然，这些语言也得有执行效率相对较低的问题，毕竟它们都是脚本语言或半编译语言，需要虚拟机解释执行，象 facebook 的 WEB 前端基本都是用 PHP 写的，他们为了解决执行效率问题，在一位华人的领导下开发了可以将 PHP 代码转成 C++ 代码的工具（hiphop)，从而使执行效率大大提高，这也从另一个侧面反映出技术人员还是希望他们的程序能够运行的更快些。

本文主要描述了 acl_cpp 库中有关 WEB 编程的方法类，为了使大家容易上手，其中的接口设计及命名尽量模仿 JAVA HttpServlet 等相关的类（希望 Oracle 不会告我侵权，呵呵）。如果您会用C/C++编程，同时又有使用 Java Servlet 进行 WEB 编程的经验，则该文您读起来一点不会费力，当然如果您多年从事 WEB 开发，我想理解这些类的设计及用法也不应该有什么难度。好了，下面就开始讲如何使用 acl_cpp 库中的 http/ 模块下的类进行 web 编程。

在 acl_cpp/src/http 模块下，有几个类与 WEB 编程相关：HttpServlet，HttpServletRequest, HttpServletResponse, HttpSession, http_header, http_mime, http_client。如果您掌握了这几个类的用法，则进行 WEB 编程就不会有什么问题了，下面一一介绍这几个类：

一、HttpServlet 类

构造函数及析构函数：

```
/**
 * 构造函数
 * @param cache_addr {const char*} memcached 的服务地址，
 *  不能为空，因为需要用 memcached 来存储与 session 相关的信息
 * @param session_name {const char*} 发给浏览器的与 session
 *  相关的 cookie 名称
 */
HttpServlet(const char* cache_addr = "127.0.0.1:11211",
```

```
                const char* session_name = "acl_session");

        /**
         * 纯虚析构函数，即该类必须由子类进行实例化
         */
        virtual ~HttpServlet(void) =0;
```

在构建函数中，为了支持 HttpSession 数据的存储，需要用户给出 memcached 的服务器地址（目前仅支持采用 memcached 来存储 session 数据，将来应该会扩展至可以支持 redis 等），同时用户还需要给出 session 的 cookie ID 标识符以发给浏览器。

四个虚接口，需要子类实现以应对不同的浏览器的 HTTP 请求：

```
    /**
     * 当 HTTP 请求为 GET 方式时的虚函数
     */
    virtual bool doGet(HttpServletRequest&, HttpServletResponse&);

    /**
     * 当 HTTP 请求为 POST 方式时的虚函数
     */
    virtual bool doPost(HttpServletRequest&, HttpServletResponse&);

    /**
     * 当 HTTP 请求为 PUT 方式时的虚函数
     */
    virtual bool doPut(HttpServletRequest&, HttpServletResponse&);

    /**
     * 当 HTTP 请求为 CONNECT 方式时的虚函数
     */
    virtual bool doConnect(HttpServletRequest&, HttpServletResponse&);
```

```
        /**
         * 当 HTTP 请求为 PURGE 方式时的虚函数，该方法在清除 SQUID 的缓存
         * 时会用到
         */
        virtual bool doPurge(HttpServletRequest&, HttpServletResponse&);
```

用户实现的 HttpServlet 子类中可以实现以上几个虚接口的一个或者几个，以满足不同的 HTTP 请求。

下面的函数为 HttpServlet 类开始运行的函数：

```
    /**
         * HttpServlet 对象开始运行，接收 HTTP 请求，并回调以下 doXXX 虚函数
         * @param stream {socket_stream*} 当在 acl_master 服务器框架控制下
         *  运行时，该参数必须非空；当在 apache 下以 CGI 方式运行时，该参数
         *  设为 NULL；另外，该函数内部不会关闭流连接，应用应自行处理流对象
         *  的关闭情况，这样可以方便与 acl_master 架构结合
         * @param session {HttpSession*} 将来准备再扩展
         * @return {bool} 返回处理结果
         */
        bool doRun(socket_stream* stream = NULL, HttpSession* session = NULL);
```

从上面五个虚方法中，可以看到两个重要的类：HttpServletRequest 和 HttpServletResponse。这两个类分别表示 http 请求类及 http 响应类，这两个类都是由 HttpServlet 类对象创建并释放的，所以用户不必创建和销毁这两个类对象实例。下面分别介绍这两个类：

二、HttpServletRequest 类

该类主要是与浏览器的请求过程相关，您可以通过该类的方法获得浏览器的请求数据。该类的方法比较多（基本上是参照了 java HttpServlet 的功能方法及名称），所以下面仅介绍几个主要的方法：

```
    /**
     * 获得 HTTP 请求中的参数值，该值已经被 URL 解码且
     * 转换成本地要求的字符集；针对 GET 方法，则是获得
     * URL 中 ? 后面的参数值；针对 POST 方法，则可以获得
     * URL 中 ? 后面的参数值或请求体中的参数值
     */
    const char* getParameter(const char* name) const;

    /**
     * 获得 HTTP 客户端请求的某个 cookie 值
     * @param name {const char*} cookie 名称，必须非空
     * @return {const char*} cookie 值，当返回 NULL 时表示
     *  cookie 值不存在
     */
    const char* getCookieValue(const char* name) const;


    /**
     * 获得与该 HTTP 会话相关的 HttpSession 对象引用
     * @return {HttpSession&}
     */
    HttpSession& getSession(void);


    /**
     * 获得与 HTTP 客户端连接关联的输入流对象引用
     * @return {istream&}
     */
    istream& getInputStream(void) const;


    /**
     * 获得 HTTP 请求数据的数据长度
```

```
                 * @return {acl_int64} 返回 -1 表示可能为 GET 方法，
                 *  或 HTTP 请求头中没有 Content-Length 字段
                 */
#ifdef WIN32
                __int64 getContentLength(void) const;
#else
                long long int getContentLength(void) const;
#endif

                /**
                 * 当 HTTP 请求头中的 Content-Type 为
                 * multipart/form-data; boundary=xxx 格式时，说明为文件上传
                 * 数据类型，则可以通过此函数获得 http_mime 对象
                 * @return {const http_mime*} 返回 NULL 则说明没有 MIME 对象，
                 *  返回的值用户不能手工释放，因为在 HttpServletRequest 的析
                 *  构中会自动释放
                 */
                http_mime* getHttpMime(void) const;

                /**
                 * 获得 HTTP 请求数据的类型
                 * @return {http_request_t}，一般对 POST 方法中的上传
                 *  文件应用而言，需要调用该函数获得是否是上传数据类型
                 */
                http_request_t getRequestType(void) const;
```

以上方法一般都是我们在实际对 HttpServletRequest 类方法使用过程中用得较多的。如：

getParmeter： 用来获得 http 请求参数

getCookieValue：获得浏览器的 cookie 值

getSession：获得该 HttpServlet 类对象的 session 会话

getInputStream：获得 http 连接的输入流

getContentLength：针对 HTTP POST 请求，此函数获得 HTTP 请求数据体的长度

getRequestType：针对 HTTP POST 请求，此函数返回 HTTP 请求数据体的传输方式（普通的 name=value 方式，multipart 上传文件格式以及数据流格式）

三、HttpServletResponse 类

该类主要与将您写的程序将处理数据结果返回给浏览器的过程相关，下面也仅介绍该类的一些常用的函数，如果您需要更多的功能，请参数 HttpServletResponse.hpp 头文件。

```
/**
     * 设置 HTTP 响应数据体的 Content-Type 字段值，可字段值可以为：
     * text/html 或 text/html; charset=utf8 格式
     * @param value {const char*} 字段值
     */
    void setContentType(const char* value);

    /**
     * 设置 HTTP 响应数据体中字符集，当已经在 setContentType 设置
     * 了字符集，则就不必再调用本函数设置字符集
     * @param charset {const char*} 响应体数据的字符集
     */
    void setCharacterEncoding(const char* charset);

    /**
     * 设置 HTTP 响应头中的状态码：1xx, 2xx, 3xx, 4xx, 5xx
     * @param status {int} HTTP 响应状态码，如：200
     */
    void setStatus(int status);

    /**
     * 添加 cookie
```

```
     * @param name {const char*} cookie 名
     * @param value {const char*} cookie 值
     * @param domain {const char*} cookie 存储域
     * @param path {const char*} cookie 存储路径
     * @param expires {time_t} cookie 过期时间间隔，当当前时间加
     *  该值为 cookie 的过期时间截(秒)
     */
    void addCookie(const char* name, const char* value,
            const char* domain = NULL, const char* path = NULL,
            time_t expires = 0);

    /**
     * 发送 HTTP 响应头，用户应该发送数据体前调用此函数将 HTTP
     * 响应头发送给客户端
     */
    bool sendHeader(void);

    /**
     * 获得 HTTP 响应对象的输出流对象，用户在调用 sendHeader 发送
     * 完 HTTP 响应头后，通过该输出流来发送 HTTP 数据体
     * @return {ostream&}
     */
    ostream& getOutputStream(void) const;
```

setCharacterEncoding：该方法设置 HTTP 响应头的 HTTP 数据体的字符集，如果通过该函数设置了字符集，即使您在返回的 html 数据中重新设置了其它的字符集，浏览器也会优先使用 HTTP 响应头中设置的字符集，所以用户一定得注意这点；

setContentType：该方法用来设置 HTTP 响应头中的 Content-Type 字段，对于 xml 数据则设置 text/xml，对 html 数据则设置 text/html，当然您也可以设置 image/jpeg 等数据类型；当然，您也可以直接通过该方法在设置数据类型的同时指定数据的字符集，如可以直接写：setContentType("text/html; charset=utf8")，这个用法等同于：setContentType("text/html"); setCharacterEncoding("utf8")。

setStatus：设置 HTTP 响应头的状态码（一般不用设置状态码，除非是您确实需要单独设置）；

addCookie：在 HTTP 响应头中添加 cookie 内容；

sendHeader：发送 HTTP 响应头；

getOutputStream：该函数返回输出流对象，您可以向输出流中直接写 HTTP 响应的数据体（关于 ostream 类的使用请参数头文件：include/ostream.hpp）。

除了以上三个类外，还有一个类比较重要：HttpSession 类，该类主要实现与 session 会话相关的功能：

四、HttpSession 类

该类对象实例用户也不必创建与释放，在 HttpServet 类对象内容自动管理该类对象实例。主要用的方法有：

```
/**
     * 获得客户端在服务端存储的对应 session 变量名，子类可以重载该方法
     * @param name {const char*} session 名，非空
     * @return {const char*} session 值，为空说明不存在或内部
     *  查询失败
     */
    virtual const char* getAttribute(const char* name) const;

    /**
     * 设置服务端对应 session 名的 session 值，子类可以重载该方法
     * @param name {const char*} session 名，非空
     * @param name {const char*} session 值，非空
     * @return {bool} 返回 false 说明设置失败
     */
    virtual bool setAttribute(const char* name, const char* value);
```

只所以将这两个方法声明为虚方法，是因为 HttpSession 的 session 数据存储目前仅支持 memcached，您如果有精力的话可以实现一个子类用来支持其它的数据存储方式。当然您也可以在您实现的子类中实现自己的产生唯一 session id 的方法，即实现如下虚方法：

```
    protected:
                /**
                 * 创建某个 session 会话的唯一 ID 号，子类可以重载该方法
                 * @param buf {char*} 存储结果缓冲区
                 * @param len {size_t} buf 缓冲区大小，buf 缓冲区大小建议
                 *  64 字节左右
                 */
                virtual void createSid(char* buf, size_t len);
```

好了，上面说了一大堆类及类函数，下面还是以一个具体的示例来说明这些类的用法：

五、示例

下面的例子是一个 CGI 例子，编译后可执行程序可以直接放在 apache 的 cgi-bin/ 目录，用户可以用浏览器访问。

```
// http_servlet.cpp : 定义控制台应用程序的入口点。
//

#include "lib_acl.hpp"

using namespace acl;

//////////////////////////////////////////////////////////////////////////

class http_servlet : public HttpServlet
{
public:
        // cache_addr: memcached 的服务地址，格式：IP:PORT
        http_servlet(const char* cache_addr)
                : HttpServlet(cache_addr)
```

```
{

}

~http_servlet(void)
{

}

// 实现处理 HTTP GET 请求的功能函数
virtual bool doGet(HttpServletRequest& req, HttpServletResponse& res)
{
        return doPost(req, res);
}

// 实现处理 HTTP POST 请求的功能函数
virtual bool doPost(HttpServletRequest& req, HttpServletResponse& res)
{
        // 获得某浏览器用户的 session 的某个变量值，如果不存在则设置一个
        const char* sid = req.getSession().getAttribute("sid");
        if (sid == NULL || *sid == 0)
                req.getSession().setAttribute("sid", "xxxxxx");

        // 再取一次该浏览器用户的 session 的某个属性值
        sid = req.getSession().getAttribute("sid");

        // 取得浏览器发来的两个 cookie 值
        const char* cookie1 = req.getCookieValue("name1");
        const char* cookie2 = req.getCookieValue("name2");

        // 开始创建 HTTP 响应头

        // 设置 cookie
        res.addCookie("name1", "value1");

        // 设置具有作用域和过期时间的 cookie
        res.addCookie("name2", "value2", ".test.com", "/", 3600 * 24);
```

```
//              res.setStatus(400);  // 可以设置返回的状态码

                // 两种方式都可以设置字符集
                if (0)
                        res.setContentType("text/xml; charset=gb2312");
                else
                {
                        // 先设置数据类型
                        res.setContentType("text/xml");

                        // 再设置数据字符集
                        res.setCharacterEncoding("gb2312");
                }

                // 获得浏览器请求的两个参数值
                const char* param1 = req.getParameter("name1");
                const char* param2 = req.getParameter("name2");

                // 创建 xml 格式的数据体
                xml body;
                body.get_root().add_child("root", true)
                        .add_child("sessions", true)
                                .add_child("session", true)
                                        .add_attr("sid", sid ? sid : "null")
                                        .get_parent()
                                .get_parent()
                        .add_child("cookies", true)
                                .add_child("cookie", true)
                                        .add_attr("name1", cookie1 ? cookie1 : "null")
                                        .get_parent()
                                .add_child("cookie", true)
                                        .add_attr("name2", cookie2 ? cookie2 : "null")
                                        .get_parent()
                                .get_parent()
                        .add_child("params", true)
                                .add_child("param", true)
                                        .add_attr("name1", param1 ? param1 : "null")
```

```
                                        .get_parent()
                                .add_child("param", true)
                                        .add_attr("name2", param2 ? param2 : "null");
                string buf;
                body.build_xml(buf);

                // 发送 http 响应头
                if (res.sendHeader() == false)
                        return false;
                // 发送 http 响应体
                if (res.getOutputStream().write(buf) == -1)
                        return false;
                return true;
        }
protected:
private:
};

////////////////////////////////////////////////////////////////////////

int main(void)
{
#ifdef WIN32
        acl::acl_cpp_init();  // win32 环境下需要初始化库
#endif

        // 开始运行，并设定 memcached 的服务地址为：127.0.0.1:11211
        http_servlet servlet("127.0.0.1:11211");

        // cgi 开始运行
        servlet.doRun();

        // 运行完毕，程序退出
        return 0;
}
```

经常使用 Java HttpServlet 等类进行 web 编程的用户对上面的代码一定不会感到陌生，但它的的确确是一个 CGI 程序，可以放在 Apache 及支持 CGI 的 Webserver 下运行。上面的例子中用到了另外一个类：xml 类，有关该类的使用说明请参数本人的另一篇博客：《acl_cpp 编程之 xml 流式解析与创建》

如果您想了解 HTTP 协议，请参考本人写的另一篇文章：《HTTP 协议简介》

该示例所在目录：acl_cpp/samples/cgi

acl_cpp 下载：http://sourceforge.net/projects/aclcpp/

原文地址：http://zsxxsz.iteye.com/blog/1535074

更多文章：http://zsxxsz.iteye.com/

25.2 用C++实现类似于JAVA HttpServlet 的编程接口

发表时间: 2012-05-20 关键字: web编程, C++模拟Java Servlet, HTTP 应用, acl_cpp 库

本人自99年开始接触网络编程，当初最早做的项目都是用 C 语言写一大堆 CGI，可谓是偿尽了编程的苦，因为那时国内的技术水平普遍比较低，如果你会 CGI 编程，就已经算是行业中人了，如果你对 CGI 编程比较熟练，则就可以称得是“专家”了，后来技术不断进步，各种国外的新技术都进入中国并不断得到普及，CGI 就逐渐沦为一种落后的技术，后来的 PHP, JSP/Servlet, ASP 逐渐占领了 WEB 编程的技术市场，这个时候如果你说再用 C 写 CGI，别人会感觉是在和古人对话。现在主流的 WEB 开发语言一个很大的优势就是有各种相对成熟的基础库和框架，开发效率很高，而 CGI 则就逊色很多。当然，这些语言也得有执行效率相对较低的问题，毕竟它们都是脚本语言或半编译语言，需要虚拟机解释执行，象 facebook 的 WEB 前端基本都是用 PHP 写的，他们为了解决执行效率问题，在一位华人的领导下开发了可以将 PHP 代码转成 C++ 代码的工具（hiphop)，从而使执行效率大大提高，这也从另一个侧面反映出技术人员还是希望他们的程序能够运行的更快些。

本文主要描述了 acl_cpp 库中有关 WEB 编程的方法类，为了使大家容易上手，其中的接口设计及命名尽量模仿 JAVA HttpServlet 等相关的类（希望 Oracle 不会告我侵权，呵呵）。如果您会用C/C++编程，同时又有使用 Java Servlet 进行 WEB 编程的经验，则该文您读起来一点不会费力，当然如果您多年从事 WEB 开发，我想理解这些类的设计及用法也不应该有什么难度。好了，下面就开始讲如何使用 acl_cpp 库中的 http/ 模块下的类进行 web 编程。

在 acl_cpp/src/http 模块下，有几个类与 WEB 编程相关：HttpServlet，HttpServletRequest, HttpServletResponse, HttpSession, http_header, http_mime, http_client。如果您掌握了这几个类的用法，则进行 WEB 编程就不会有什么问题了，下面一一介绍这几个类：

一、HttpServlet 类

构造函数及析构函数：

```
/**
 * 构造函数
 * @param cache_addr {const char*} memcached 的服务地址，
 *  不能为空，因为需要用 memcached 来存储与 session 相关的信息
 * @param session_name {const char*} 发给浏览器的与 session
 *  相关的 cookie 名称
 */
```

```
        HttpServlet(const char* cache_addr = "127.0.0.1:11211",
                const char* session_name = "acl_session");

        /**
         * 纯虚析构函数，即该类必须由子类进行实例化
         */
        virtual ~HttpServlet(void) =0;
```

在构建函数中，为了支持 HttpSession 数据的存储，需要用户给出 memcached 的服务器地址（目前仅支持采用 memcached 来存储 session 数据，将来应该会扩展至可以支持 redis 等），同时用户还需要给出 session 的 cookie ID 标识符以发给浏览器。

四个虚接口，需要子类实现以应对不同的浏览器的 HTTP 请求：

```
    /**
         * 当 HTTP 请求为 GET 方式时的虚函数
         */
        virtual bool doGet(HttpServletRequest&, HttpServletResponse&);

        /**
         * 当 HTTP 请求为 POST 方式时的虚函数
         */
        virtual bool doPost(HttpServletRequest&, HttpServletResponse&);

        /**
         * 当 HTTP 请求为 PUT 方式时的虚函数
         */
        virtual bool doPut(HttpServletRequest&, HttpServletResponse&);

        /**
         * 当 HTTP 请求为 CONNECT 方式时的虚函数
         */
        virtual bool doConnect(HttpServletRequest&, HttpServletResponse&);
```

```
        /**
         * 当 HTTP 请求为 PURGE 方式时的虚函数，该方法在清除 SQUID 的缓存
         * 时会用到
         */
        virtual bool doPurge(HttpServletRequest&, HttpServletResponse&);
```

用户实现的 HttpServlet 子类中可以实现以上几个虚接口的一个或者几个，以满足不同的 HTTP 请求。

下面的函数为 HttpServlet 类开始运行的函数：

```
    /**
         * HttpServlet 对象开始运行，接收 HTTP 请求，并回调以下 doXXX 虚函数
         * @param stream {socket_stream*} 当在 acl_master 服务器框架控制下
         *  运行时，该参数必须非空；当在 apache 下以 CGI 方式运行时，该参数
         *  设为 NULL；另外，该函数内部不会关闭流连接，应用应自行处理流对象
         *  的关闭情况，这样可以方便与 acl_master 架构结合
         * @param session {HttpSession*} 将来准备再扩展
         * @return {bool} 返回处理结果
         */
        bool doRun(socket_stream* stream = NULL, HttpSession* session = NULL);
```

从上面五个虚方法中，可以看到两个重要的类：HttpServletRequest 和 HttpServletResponse。这两个类分别表示 http 请求类及 http 响应类，这两个类都是由 HttpServlet 类对象创建并释放的，所以用户不必创建和销毁这两个类对象实例。下面分别介绍这两个类：

二、HttpServletRequest 类

该类主要是与浏览器的请求过程相关，您可以通过该类的方法获得浏览器的请求数据。该类的方法比较多（基本上是参照了 java HttpServlet 的功能方法及名称），所以下面仅介绍几个主要的方法：

```
/**
     * 获得 HTTP 请求中的参数值，该值已经被 URL 解码且
     * 转换成本地要求的字符集；针对 GET 方法，则是获得
     * URL 中 ? 后面的参数值；针对 POST 方法，则可以获得
     * URL 中 ? 后面的参数值或请求体中的参数值
     */
    const char* getParameter(const char* name) const;

    /**
     * 获得 HTTP 客户端请求的某个 cookie 值
     * @param name {const char*} cookie 名称，必须非空
     * @return {const char*} cookie 值，当返回 NULL 时表示
     *  cookie 值不存在
     */
    const char* getCookieValue(const char* name) const;



    /**
     * 获得与该 HTTP 会话相关的 HttpSession 对象引用
     * @return {HttpSession&}
     */
    HttpSession& getSession(void);



    /**
     * 获得与 HTTP 客户端连接关联的输入流对象引用
     * @return {istream&}
     */
    istream& getInputStream(void) const;
```

```
                /**
                 * 获得 HTTP 请求数据的数据长度
                 * @return {acl_int64} 返回 -1 表示可能为 GET 方法，
                 *  或 HTTP 请求头中没有 Content-Length 字段
                 */
#ifdef WIN32
                __int64 getContentLength(void) const;
#else
                long long int getContentLength(void) const;
#endif

                /**
                 * 当 HTTP 请求头中的 Content-Type 为
                 * multipart/form-data; boundary=xxx 格式时，说明为文件上传
                 * 数据类型，则可以通过此函数获得 http_mime 对象
                 * @return {const http_mime*} 返回 NULL 则说明没有 MIME 对象，
                 *  返回的值用户不能手工释放，因为在 HttpServletRequest 的析
                 *  构中会自动释放
                 */
                http_mime* getHttpMime(void) const;

                /**
                 * 获得 HTTP 请求数据的类型
                 * @return {http_request_t}，一般对 POST 方法中的上传
                 *  文件应用而言，需要调用该函数获得是否是上传数据类型
                 */
                http_request_t getRequestType(void) const;
```

以上方法一般都是我们在实际对 HttpServletRequest 类方法使用过程中用得较多的。如：

getParmeter： 用来获得 http 请求参数

getCookieValue：获得浏览器的 cookie 值

getSession：获得该 HttpServlet 类对象的 session 会话

getInputStream：获得 http 连接的输入流

getContentLength：针对 HTTP POST 请求，此函数获得 HTTP 请求数据体的长度

getRequestType：针对 HTTP POST 请求，此函数返回 HTTP 请求数据体的传输方式（普通的 name=value 方式，multipart 上传文件格式以及数据流格式）

三、HttpServletResponse 类

该类主要与将您写的程序将处理数据结果返回给浏览器的过程相关，下面也仅介绍该类的一些常用的函数，如果您需要更多的功能，请参数 HttpServletResponse.hpp 头文件。

```
/**
     * 设置 HTTP 响应数据体的 Content-Type 字段值，可字段值可以为：
     * text/html 或 text/html; charset=utf8 格式
     * @param value {const char*} 字段值
     */
    void setContentType(const char* value);

    /**
     * 设置 HTTP 响应数据体中字符集，当已经在 setContentType 设置
     * 了字符集，则就不必再调用本函数设置字符集
     * @param charset {const char*} 响应体数据的字符集
     */
    void setCharacterEncoding(const char* charset);

    /**
     * 设置 HTTP 响应头中的状态码：1xx, 2xx, 3xx, 4xx, 5xx
     * @param status {int} HTTP 响应状态码，如：200
     */
    void setStatus(int status);
```

```
/**
 * 添加 cookie
 * @param name {const char*} cookie 名
 * @param value {const char*} cookie 值
 * @param domain {const char*} cookie 存储域
 * @param path {const char*} cookie 存储路径
 * @param expires {time_t} cookie 过期时间间隔，当当前时间加
 *  该值为 cookie 的过期时间截(秒)
 */
void addCookie(const char* name, const char* value,
        const char* domain = NULL, const char* path = NULL,
        time_t expires = 0);

/**
 * 发送 HTTP 响应头，用户应该发送数据体前调用此函数将 HTTP
 * 响应头发送给客户端
 */
bool sendHeader(void);

/**
 * 获得 HTTP 响应对象的输出流对象，用户在调用 sendHeader 发送
 * 完 HTTP 响应头后，通过该输出流来发送 HTTP 数据体
 * @return {ostream&}
 */
ostream& getOutputStream(void) const;
```

setCharacterEncoding：该方法设置 HTTP 响应头的 HTTP 数据体的字符集，如果通过该函数设置了字符集，即使您在返回的 html 数据中重新设置了其它的字符集，浏览器也会优先使用 HTTP 响应头中设置的字符集，所以用户一定得注意这点；

setContentType：该方法用来设置 HTTP 响应头中的 Content-Type 字段，对于 xml 数据则设置 text/xml，对 html 数据则设置 text/html，当然您也可以设置 image/jpeg 等数据类型；当然，您也可以直接通过该方法在设置数据类型的同时指定数据的字符集，如可以直接写：setContentType("text/html; charset=utf8")，这个用法等同于：setContentType("text/html"); setCharacterEncoding("utf8")。

setStatus：设置 HTTP 响应头的状态码（一般不用设置状态码，除非是您确实需要单独设置）；

addCookie：在 HTTP 响应头中添加 cookie 内容；

sendHeader：发送 HTTP 响应头；

getOutputStream：该函数返回输出流对象，您可以向输出流中直接写 HTTP 响应的数据体（关于 ostream 类的使用请参数头文件：include/ostream.hpp）。

除了以上三个类外，还有一个类比较重要：HttpSession 类，该类主要实现与 session 会话相关的功能：

四、HttpSession 类

该类对象实例用户也不必创建与释放，在 HttpServet 类对象内容自动管理该类对象实例。主要用的方法有：

```
/**
     * 获得客户端在服务端存储的对应 session 变量名，子类可以重载该方法
     * @param name {const char*} session 名，非空
     * @return {const char*} session 值，为空说明不存在或内部
     *  查询失败
     */
    virtual const char* getAttribute(const char* name) const;

    /**
     * 设置服务端对应 session 名的 session 值，子类可以重载该方法
     * @param name {const char*} session 名，非空
     * @param name {const char*} session 值，非空
     * @return {bool} 返回 false 说明设置失败
     */
    virtual bool setAttribute(const char* name, const char* value);
```

只所以将这两个方法声明为虚方法，是因为 HttpSession 的 session 数据存储目前仅支持 memcached，您如果有精力的话可以实现一个子类用来支持其它的数据存储方式。当然您也可以在您实现的子类中实现自己的产生唯一 session id 的方法，即实现如下虚方法：

```
protected:
        /**
         * 创建某个 session 会话的唯一 ID 号，子类可以重载该方法
         * @param buf {char*} 存储结果缓冲区
         * @param len {size_t} buf 缓冲区大小，buf 缓冲区大小建议
         *  64 字节左右
         */
        virtual void createSid(char* buf, size_t len);
```

好了，上面说了一大堆类及类函数，下面还是以一个具体的示例来说明这些类的用法：

五、示例

下面的例子是一个 CGI 例子，编译后可执行程序可以直接放在 apache 的 cgi-bin/ 目录，用户可以用浏览器访问。

```
// http_servlet.cpp : 定义控制台应用程序的入口点。
//

#include "lib_acl.hpp"

using namespace acl;

//////////////////////////////////////////////////////////////////////////

class http_servlet : public HttpServlet
{
public:
```

```
// cache_addr: memcached 的服务地址，格式：IP:PORT
http_servlet(const char* cache_addr)
        : HttpServlet(cache_addr)
{

}

~http_servlet(void)
{

}

// 实现处理 HTTP GET 请求的功能函数
virtual bool doGet(HttpServletRequest& req, HttpServletResponse& res)
{
        return doPost(req, res);
}

// 实现处理 HTTP POST 请求的功能函数
virtual bool doPost(HttpServletRequest& req, HttpServletResponse& res)
{
        // 获得某浏览器用户的 session 的某个变量值，如果不存在则设置一个
        const char* sid = req.getSession().getAttribute("sid");
        if (sid == NULL || *sid == 0)
                req.getSession().setAttribute("sid", "xxxxxx");

        // 再取一次该浏览器用户的 session 的某个属性值
        sid = req.getSession().getAttribute("sid");

        // 取得浏览器发来的两个 cookie 值
        const char* cookie1 = req.getCookieValue("name1");
        const char* cookie2 = req.getCookieValue("name2");

        // 开始创建 HTTP 响应头

        // 设置 cookie
        res.addCookie("name1", "value1");
```

```
		// 设置具有作用域和过期时间的 cookie
		res.addCookie("name2", "value2", ".test.com", "/", 3600 * 24);
//		res.setStatus(400);  // 可以设置返回的状态码

		// 两种方式都可以设置字符集
		if (0)
			res.setContentType("text/xml; charset=gb2312");
		else
		{
			// 先设置数据类型
			res.setContentType("text/xml");

			// 再设置数据字符集
			res.setCharacterEncoding("gb2312");
		}

		// 获得浏览器请求的两个参数值
		const char* param1 = req.getParameter("name1");
		const char* param2 = req.getParameter("name2");

		// 创建 xml 格式的数据体
		xml body;
		body.get_root().add_child("root", true)
			.add_child("sessions", true)
				.add_child("session", true)
					.add_attr("sid", sid ? sid : "null")
					.get_parent()
				.get_parent()
			.add_child("cookies", true)
				.add_child("cookie", true)
					.add_attr("name1", cookie1 ? cookie1 : "null")
					.get_parent()
				.add_child("cookie", true)
					.add_attr("name2", cookie2 ? cookie2 : "null")
					.get_parent()
				.get_parent()
```

```
                        .add_child("params", true)
                                .add_child("param", true)
                                        .add_attr("name1", param1 ? param1 : "null")
                                        .get_parent()
                                .add_child("param", true)
                                        .add_attr("name2", param2 ? param2 : "null");
                string buf;
                body.build_xml(buf);

                // 发送 http 响应头
                if (res.sendHeader() == false)
                        return false;
                // 发送 http 响应体
                if (res.getOutputStream().write(buf) == -1)
                        return false;
                return true;
        }
protected:
private:
};

//////////////////////////////////////////////////////////////////////////

int main(void)
{
#ifdef WIN32
        acl::acl_cpp_init();  // win32 环境下需要初始化库
#endif

        // 开始运行，并设定 memcached 的服务地址为：127.0.0.1:11211
        http_servlet servlet("127.0.0.1:11211");

        // cgi 开始运行
        servlet.doRun();

        // 运行完毕，程序退出
```

```
        return 0;
}
```

经常使用 Java HttpServlet 等类进行 web 编程的用户对上面的代码一定不会感到陌生，但它的的确确是一个 CGI 程序，可以放在 Apache 及支持 CGI 的 Webserver 下运行。当然，大家应该都清楚 CGI 在运行时因进程切换而导致了效率较为低下，在另一篇文章《使用 acl_cpp 的 HttpServlet 类及服务器框架编写WEB服务器程序 》中展示了用上面的 http_servlet 类并结合 acl_cpp 的服务器模型实现的一个WEB服务器的例子，效率比 CGI 要高的多(效率也应比 FCGI高，因为其少了 Webserver 层的过滤)；文章《acl_cpp web 编程之文件上传 》中举例讲述了在服务端如何使用 acl_cpp 库处理浏览器上传文件的功能。

上面的例子中用到了另外一个类：xml 类，有关该类的使用说明请参考博客：《acl_cpp 编程之 xml 流式解析与创建 》；如果您想了解 HTTP 协议，请参考文章：《HTTP 协议简介 》

该示例所在目录：acl_cpp/samples/cgi

原文地址

acl_cpp 下载

acl_cpp 的编译与使用

更多文章

25.3 使用 acl_cpp 的 HttpServlet 类及服务器框架编写WEB服务器程序

发表时间: 2012-05-21 关键字: web 应用, HTTP应用, 服务器编程, acl_cpp 库

在 《用C++实现类似于JAVA HttpServlet 的编程接口 》 文章中讲了如何用 HttpServlet 等相关类编写 CGI 程序，于是有网友提出了 CGI 程序低效性，不错，确实 CGI 程序的进程开销是比较大的，本文就将说明依然是这些 HTTP 相关的类，如果在使用 acl_cpp/src/master 下的服务器框架类的条件下，可以非常方便地转为服务器程序。现在依然是使用 《用C++实现类似于JAVA HttpServlet 的编程接口 》示例中的 http_servlet 类，只是稍微修改一下 main 函数，就变成下面的情形：

```
////////////////////////////////////////////////////////////////////////

class master_service : public acl::master_proc
{
public:
        master_service() {}
        ~master_service() {}
protected:
        // 基类虚函数，当接收到一个 HTTP 客户端请求时，服务器
        // 框架回调此函数将连接流传入
        virtual void on_accept(acl::socket_stream* stream)
        {
                // HttpServlet 的子类实例
                http_servlet servlet("127.0.0.1:11211");
                servlet.setLocalCharset("gb2312");  // 设置本地字符集
                servlet.doRun(stream);  // 开始处理浏览器请求过程
        }
};

////////////////////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{
        acl::acl_cpp_init();  // 初始化 acl_cpp 库
```

```
        master_service service;  // 半驻留进程池服务类对象

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                // 当在手工调试时一般采用此方式
                printf("listen: 127.0.0.1:8888 ...\r\n");
                service.run_alone("127.0.0.1:8888", NULL, 1);  // 单独运行方式
        }
        else  // 生产环境中以半驻留进程池模式运行
                service.run_daemon(argc, argv);  // acl_master 控制模式运行

        return 0;
}
```

上面的例子是一个结合 HttpServlet 类及 master_service 进程池服务类的 HTTP 服务器程序，关于进程池的例子，可以先结合本人原来写过的基于C语言库 acl 的一篇文章《快速创建你的服务器程序 - - single进程池模型 》。

不仅可以非常容易地将 HttpServlet 写成进程池方式，同时还可以结合 acl_cpp 的线程池框架模板，将 HttpServlet 类实现为半驻留线程池实例，下面就显示了这一过程：

```
class master_threads_test : public acl::master_threads
{
public:
        master_threads_test() {}

        ~master_threads_test() {}

protected:
        // 基类纯虚函数：当客户端连接有数据可读或关闭时回调此函数，返回 true 表示
        // 继续与客户端保持长连接，否则表示需要关闭客户端连接
```

```
        virtual bool thread_on_read(acl::socket_stream* stream)
        {
                // HttpServlet 的子类实例
                http_servlet servlet("127.0.0.1:11211");
                servlet.setLocalCharset("gb2312");  // 设置本地字符集
                servlet.doRun(stream);  // 开始处理浏览器请求过程
        }

        // 基类虚函数：当接收到一个客户端请求时，调用此函数，允许
        // 子类事先对客户端连接进行处理，返回 true 表示继续，否则
        // 要求关闭该客户端连接
        virtual bool thread_on_accept(acl::socket_stream*)
        {
                return true;  // 返回 true 以允许服务器框架继续调用 thread_on_read 过程
        }
};

// 字符串类配置参数项

static char *var_cfg_debug_msg;

static acl::master_str_tbl var_conf_str_tab[] = {
        { "debug_msg", "test_msg", &var_cfg_debug_msg },

        { 0, 0, 0 }
};

// 布尔配置参数项
static int  var_cfg_debug_enable;
static int  var_cfg_keep_alive;
static int  var_cfg_loop;

static acl::master_bool_tbl var_conf_bool_tab[] = {
        { "debug_enable", 1, &var_cfg_debug_enable },
        { "keep_alive", 1, &var_cfg_keep_alive },
        { "loop_read", 1, &var_cfg_loop },
```

```
        { 0, 0, 0 }
};

// 整数配置参数项
static int  var_cfg_io_timeout;

static acl::master_int_tbl var_conf_int_tab[] = {
        { "io_timeout", 120, &var_cfg_io_timeout, 0, 0 },

        { 0, 0 , 0 , 0, 0 }
};

int main(int argc, char* argv[])
{
        master_threads_test mt;  // 半驻留线程池服务器实例

        // 设置配置参数表
        mt.set_cfg_int(var_conf_int_tab);
        mt.set_cfg_int64(NULL);
        mt.set_cfg_str(var_conf_str_tab);
        mt.set_cfg_bool(var_conf_bool_tab);

        // 开始运行

        if (argc >= 2 && strcmp(argv[1], "alone") == 0)
        {
                // 当在手工调试时一般采用此方式
                printf("listen: 127.0.0.1:8888\r\n");
                mt.run_alone("127.0.0.1:8888", NULL, 2, 10);  // 单独运行方式
        }
        else  // 生产环境中以半驻留线程池模式运行
                mt.run_daemon(argc, argv);  // acl_master 控制模式运行
        return 0;
}
```

该例子显示了一个基于线程池服务器模型的WEB实例，可以依然使用了文章 《用C++实现类似于JAVA HttpServlet 的编程接口 》示例中的 http_servlet 类，但采用的是由文章 《 开发多线程进程池服务器程序---acl 服务器框架应用 》所介绍的多进程多线程服务器框架模板。

acl_cpp 下载

原文地址

更多文章

25.4 web 编程中实现文件上传的服务端实例

发表时间: 2012-05-22 关键字: http 应用, web 编程, 文件上传, CGI 编程, acl_cpp

在文章《用C++实现类似于JAVA HttpServlet 的编程接口 》中讲了如何用 acl_cpp 的 HttpServlet 等类来实现 WEB CGI 的功能，同时在文章《使用 acl_cpp 的 HttpServlet 类及服务器框架编写WEB服务器程序 》中也举例说明如何将基于 HttpServlet 编写的 CGI 程序快速地转为服务器程序的过程。本文主要讲如何用 acl_cpp 的 WEB 编程类实现 HTTP 文件上传过程。为了实现 HTTP 协议的文件上传过程，引入了两个类：http_mime 和 http_mime_node。

http_mime 类是有关 HTTP 协议中 mime 格式的流式解析器（即每次仅输入部分 HTTP MIME 数据，等数据输入完毕时，该解析器也解析完毕，流式解析的好处是它可以适用于阻塞或非阻塞的IO模式）；http_mime_node 类对象表示 http mime 数据中每一个 mime 结点对象，该结点的数据可能是文件内容数据，也可能是参数数据。

一、http_mime 类

该类一般由 HttpServletRequest 类内部自动管理（负责分配与释放 http_mide 类对象），当然用户可以在测试 http_mime 类时，自己创建与释放该类对象。下面是该类的构造函数及常用方法：

```
/**
 * 构建函数
 * @param boundary {const char*} 分隔符，不能为空
 * @param local_charset {const char*} 本地字符集，非空时会自动将
 *  参数内容转为本地字符集
 */
http_mime(const char* boundary, const char* local_charset  = "gb2312");
```

尤其需要指出的是 http mime 的 boundary(分隔符）与邮件的 mime 的分隔符规则略有不同，如邮件的相关头部字段为：Content-Type: multipart/mixed; charset="GB2312"; boundary="0_11119_1331286082"，HTTP MIME 的相关头部字段为：Content-Type: multipart/form-data; boundary="--0_11119_1331286082"。其中，最大的区别就是在 HTTP 头中获得的分隔符与 HTTP 数据体的分隔符（除结尾分隔符多了两个 '-' 后缀）完全相同，而邮件的 mime 的分隔符在头部和 mime 体中是不一样的，mime 体中的分隔符是由头部的分隔符加两个 '-' 作为前导符（结尾分隔符为头部分隔符前面加两个

'-'，尾部加两个 '-' ），一定得注意这些不同。在 acl_cpp 中的 http mime 解析模块原来主要是作邮件 mime 解析的，现在依然支持 HTTP 的 mime 解析，唯一不同就是区分分隔符的不同。（当然，邮件的 MIME 数据体还与 HTTP MIME 数据体有另外一个区别：邮件的 MIME 数据一般都是要经过 BASE64 来编码的，而 HTTP MIME 却很少编码）。

http_mime 的几个常用方法接口如下：

```
/**
 * 设置 MIME 数据的存储路径，当分析完 MIME 数据后，如果想要
 * 从中提取数据，则必须给出该 MIME 的原始数据的存储位置，否则
 * 无法获得相应数据，即 save_xxx/get_nodes/get_node 函数均无法
 * 正常使用
 * @param path {const char*} 文件路径名，如果该参数为空，则不能
 *  获得数据体数据，也不能调用 save_xxx 相关的接口
 */
void set_saved_path(const char* path);

/**
 * 调用此函数进行流式方式解析数据体内容
 * @param data {const char*} 数据体(可能是数据头也可能是数据体，
 *  并且不必是完整的数据行)
 * @param len {size_t} data 数据长度
 * @return {bool} 针对 multipart 数据，返回 true 表示解析完毕；
 *  对于非 multipart 文件，该返回值永远为 false，没有任何意义，
 *  需要调用者自己判断数据体的结束位置
 * 注意：调用完此函数后一定需要调用 update_end 函数通知解析器
 * 解析完毕
 */
bool update(const char* data, size_t len);

/**
 * 获得所有的 MIME 结点
 * @return {const std::list<http_mimde_node*>&}
 */
```

```
        const std::list<http_mime_node*>& get_nodes(void) const;

        /**
         * 根据变量名取得 HTTP MIME 结点
         * @param param name {const char*} 变量名
         * @return {http_mime_node*} 返回空则说明对应变量名的结点
         *  不存在
         */
        const http_mime_node* get_node(const char* name) const;
```

二、http_mime_node 类

该类实例存储 HTTP MIME 数据体中每个数据结点，同时该类的实例是由 http_mime 类对象自动维护的，所以您一般不必关心该类对象的创建与销毁；另外，http_mime_node 类的继承关系为：http_mime_node -> mime_attach -> mime_node。

该类的构造函数如下：

```
    /**
         * 原始文件存放路径，不能为空
         * @param node {MIME_NODE*} 对应的 MIME 结点，非空
         * @param decodeIt {bool} 是否对 MIME 结点的头部数据
         *  或数据体数据进行解码
         * @param toCharset {const char*} 本机的字符集
         * @param off {off_t} 偏移数据位置
         */
        http_mime_node(const char* path, const MIME_NODE* node,
                bool decodeIt = true, const char* toCharset = "gb2312",
                off_t off = 0);
```

该类的常用方法为：

```
        /**
              * 获得该结点的类型
              * @return {http_mime_t}
              */
             http_mime_t get_mime_type(void) const;

             /**
              * 当 get_mime_type 返回的类型为 HTTP_MIME_PARAM 时，可以
              * 调用此函数获得参数值；参数名可以通过基类的 get_name() 获得
              * @return {const char*} 返回 NULL 表示参数不存在
              */
             const char* get_value(void) const;
```

http_mime_t 为枚举类型，如：

```
typedef enum
     {
             HTTP_MIME_PARAM,        // http mime 结点为参数类型
             HTTP_MIME_FILE          // http mime 结点为文件类型
     } http_mime_t;
```

加上两个基类的一些方法，有几个方法也是比较常用的，如下：

mime_node::get_name: 获得该 mime 结点的名称

mime_attach::get_filename: 当结点为上传文件类型时，此函数获得上传文件的文件名

三、示例

```
#include "lib_acl.hpp"

using namespace acl;

class http_servlet : public HttpServlet
{
public:
        http_servlet(const char* cache_addr)
                : HttpServlet(cache_addr)
        {
                ...
        }

        ...
        // 基类虚方法：HTTP POST 方法接口
        virtual bool doPost(HttpServletRequest& req, HttpServletResponse& res)
        {
                ...
                return doUpload(req, res);
        }

        // 处理文件上传的函数
        bool doUpload(HttpServletRequest& req, HttpServletResponse& res)
        {
                // 先获得 Content-Type 对应的 http_ctype 对象
                http_mime* mime = req.getHttpMime();
                if (mime == NULL)
                {
                        logger_error("http_mime null");
                        return false;
                }

                // 获得数据体的长度
```

```
long long int len = req.getContentLength();
if (len <= 0)
{
        logger_error("body empty");
        return false;
}

// 获得输入流
istream& in = req.getInputStream();
char  buf[8192];
int   ret;
bool  n = false;

const char* filepath = "./var/mime_file";
ofstream out;
// 只写方式打开存储上传文件的临时文件句柄
out.open_write(filepath);

// 设置原始文件存入路径
mime->set_saved_path(filepath);

// 读取 HTTP 客户端请求数据
while (len > 0)
{
        // 从 HTTP 输入流中读取数据
        ret = in.read(buf, sizeof(buf), false);
        if (ret == -1)
        {
                logger_error("read POST data error");
                return false;
        }
        // 将数据写入临时文件中
        out.write(buf, ret);
        len -= ret;

        // 将读得到的数据输入至解析器进行解析
        if (mime->update(buf, ret) == true)
```

```
        {
                n = true;
                break;
        }
}
out.close();

if (len != 0 || n == false)
        logger_warn("not read all data from client");

string path;

// 遍历所有的 MIME 结点，找出其中为文件结点的部分进行转储
const std::list<http_mime_node*>& nodes = mime->get_nodes();
std::list<http_mime_node*>::const_iterator cit = nodes.begin();
for (; cit != nodes.end(); ++cit)
{
        // HTTP MIME 结点的变量名
        const char* name = (*cit)->get_name();

        // HTTP MIME 结点的类型
        http_mime_t mime_type = (*cit)->get_mime_type();
        if (mime_type == HTTP_MIME_FILE)
        {
                // 当该结点为文件数据结点时
                // 取得上传文件名
                const char* filename = (*cit)->get_filename();
                if (filename == NULL)
                {
                        logger("filename null");
                        continue;
                }

                if (strcmp(name, "file1") == 0)
                        file1_ = filename;
                else if (strcmp(name, "file2") == 0)
                        file2_ = filename;
```

```
                        else if (strcmp(name, "file3") == 0)
                                file3_ = filename;

                        // 将文件内容转存
                        path.format("./var/%s", filename);
                        (void) (*cit)->save(path.c_str());
                }
        }

        // 查找上载的某个文件并转储
        const http_mime_node* node = mime->get_node("file1");
        if (node && node->get_mime_type() == HTTP_MIME_FILE)
        {
                const char* ptr = node->get_filename();
                if (ptr)
                {
                        path.format("./var/1_%s", ptr);
                        (void) node->save(path.c_str());
                }
        }

        // 删除临时文件
        :unlink(filepath);

        // 发送 http 响应头
        if (res.sendHeader() == false)
                return false;
        // 发送 http 响应体
        if (res.getOutputStream().write("ok") == -1)
                return false;
        return true;
    }

private:
    const char* file1_;
    const char* file2_;
    const char* file3_;
```

```
};

int main(void)
{
#ifdef WIN32
        acl::acl_cpp_init();
#endif

        // 开始运行
        http_servlet servlet("127.0.0.1:11211");
        servlet.doRun();
        return 0;
}
```

与上面例子对应的 HTML 页面如下：

```
<html>
<head>
<meta content="text/html; charset=gb2312" http-equiv="Content-Type">
</head>
<body>
<form enctype="multipart/form-data" method=POST action="/cgi-bin/test/upload?name1=中国人">
<input type=hidden name="name2" value="美国人"><br>
<input type=hidden name="name3" value="英国人"><br>
<input type=submit name="submit", value="提交"><br>
文件一：<input type=file name="file1" value=""><br>
文件二：<input type=file name="file2" value=""><br>
文件三：<input type=file name="file3" value=""><br>
</form>
</body>
</html>
```

上面例子比较简单地说明了如果使用 acl_cpp 中的 HttpServlet/http_mime 等类来实现文件上传的功能，完整的例子请参考：acl_cpp/samples/cig_upload。该例子虽然是一个 CGI 程序，但您依然可以不费吹灰之力

将其改变成一个服务器程序，转换方法可参考：《使用 acl_cpp 的 HttpServlet 类及服务器框架编写WEB服务器程序》。

原文地址

acl_cpp 下载

acl_cpp 的编译与使用

更多文章