把 Linux 下的链表 List 移植到 Win32

作者：wilson　e-mail：wilsonwong@126.com　http://blog.chinaunix.net/u1/59572

序言

链表操作在 Linux 系统相当普遍，本文档把 Linux 下的链表操作移植到 Win32 平台下进行详细说明，并附上测试代码。

《链表操作定义文件 list.h》

```
/*Linux下的链表操作移植到Win32*/
#ifndef _LINUX_LIST_H
#define _LINUX_LIST_H
#ifdef __cplusplus
extern "C" {  //如果当前运行环境为C++，则需要加上 extern “C”
#undef  NULL
#define NULL 0
#else
#undef  NULL
#define NULL 0
#endif
//平台定义
#ifndef WIN32
#define WIN32
#endif
//计算结构里的某个成员的偏移量：把地址作为结构的开始地址，因此成员的地址即为偏移地址（相对地址）
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
//通过结构类型及其某一成员的指针，计算结构体变量地址：计算结构体成员的偏移地址，然后用该成员的地址减去偏移地址，便是结构体变量的地址
#ifdef WIN32
#define container_of(ptr, type, member) (type *)( (char *)ptr - offsetof(type,member) )
#else
#define container_of(ptr, type, member) ( { \
const typeof( ((type *)0)->member ) *__mptr = (ptr); \
(type *)( (char *)__mptr - offsetof(type,member) ); } )
/*宏定义分解解析：
    (type *)0--------把0地址转换为结构体type，这种巧妙的0地址类型转换为计算结构体成员的偏移地址提供简单方法
    const typeof( ((type *)0)->member ) *__mptr=(ptr)----获取成员member的数据类型并用此类型定义变量 __mptr，同时把
                    ptr指针传给__mptr（注意__mptr与ptr必须同类型）；typeof是gnu扩展语法，在标准C++里不支持；
                    当本人觉得这一步骤是多余的。因此本文件移植到Win32时，必须作适当调整。
    Offsetof(type,member)---计算结构体type的成员member的地址偏移量（以0地址转换结构体type变量，其成员的绝对地址
                     就是偏移地址（绝对地址-0））
    (char *)__mptr - offsetof(type,member) ----成员绝对地址（指针）减去成员的偏移地址（相对结构体指针）就是结构体
                     地址了。
    (type *)( (char *)__mptr - offsetof(type,member) )----类型转换
*/
#endif

static inline void prefetch(const void *x) {;}
static inline void prefetchw(const void *x) {;}
```

```
#define LIST_POISON1  ((void *) 0x00100100)
#define LIST_POISON2  ((void *) 0x00200200)

struct list_head {
    struct list_head *next, *prev;
};

#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
struct list_head name = LIST_HEAD_INIT(name)

#define INIT_LIST_HEAD(ptr) do { \
    (ptr)->next = (ptr); (ptr)->prev = (ptr); \
} while (0)

static inline void __list_add(struct list_head *newi, struct list_head *prev, struct list_head *next);

static inline void list_add(struct list_head *newi, struct list_head *head);

static inline void list_add_tail(struct list_head *newi, struct list_head *head);
static inline void __list_del(struct list_head * prev, struct list_head * next);
static inline void list_del(struct list_head *entry);
static inline void list_del_init(struct list_head *entry);
static inline void list_move(struct list_head *list, struct list_head *head);
static inline void list_move_tail(struct list_head *list,
  struct list_head *head);
static inline int list_empty(const struct list_head *head);
static inline int list_empty_careful(const struct list_head *head);
static inline void __list_splice(struct list_head *list,
 struct list_head *head);
static inline void list_splice(struct list_head *list, struct list_head *head);
static inline void list_splice_init(struct list_head *list,
    struct list_head *head);

#define list_entry(ptr, type, member) container_of(ptr, type, member)


#define list_for_each(pos, head) \
for (pos = (head)->next; prefetch(pos->next), pos != (head); \
        pos = pos->next)

#define __list_for_each(pos, head) \
for (pos = (head)->next; pos != (head); pos = pos->next)

#define list_for_each_prev(pos, head) \
for (pos = (head)->prev; prefetch(pos->prev), pos != (head); \
        pos = pos->prev)

#define list_for_each_safe(pos, n, head) \
for (pos = (head)->next, n = pos->next; pos != (head); \
pos = n, n = pos->next)
```

```
// for LINUX
#define list_for_each_entry(pos, head, member) \
for (pos = list_entry((head)->next, typeof(*pos), member); \
     prefetch(pos->member.next), &pos->member != (head);  \
     pos = list_entry(pos->member.next, typeof(*pos), member))
// for WIN32，未找到typeof在标准C++的替代方法，因此只能从宏明确传递结构体声明，操作中可行；以下同
#define list_for_each_entry(pos,pos_type, head, member) \
for (pos = list_entry((head)->next, pos_type, member); \
     prefetch(pos->member.next), &pos->member != (head);  \
     pos = list_entry(pos->member.next, pos_type, member))
// for LINUX
#define list_for_each_entry_reverse(pos, head, member) \
for (pos = list_entry((head)->prev, typeof(*pos), member); \
     prefetch(pos->member.prev), &pos->member != (head);  \
     pos = list_entry(pos->member.prev, typeof(*pos), member))
// for WIN32
#define list_for_each_entry_reverse(pos,pos_type, head, member) \
for (pos = list_entry((head)->prev, pos_type, member); \
     prefetch(pos->member.prev), &pos->member != (head);  \
     pos = list_entry(pos->member.prev, pos_type, member))
// for LINUX
#define list_prepare_entry(pos, head, member) \
((pos) ? : list_entry(head, typeof(*pos), member))
// for WIN32
#define list_prepare_entry(pos,pos_type, head, member) \
((pos) ? : list_entry(head, pos_type, member))
// for LINUX
#define list_for_each_entry_continue(pos, head, member)  \
for (pos = list_entry(pos->member.next, typeof(*pos), member); \
     prefetch(pos->member.next), &pos->member != (head); \
     pos = list_entry(pos->member.next, typeof(*pos), member))
// for WIN32
#define list_for_each_entry_continue(pos,pos_type, head, member)  \
for (pos = list_entry(pos->member.next, pos_type, member); \
     prefetch(pos->member.next), &pos->member != (head); \
     pos = list_entry(pos->member.next, pos_type, member))
// for LINUX
#define list_for_each_entry_safe(pos, n, head, member) \
for (pos = list_entry((head)->next, typeof(*pos), member), \
n = list_entry(pos->member.next, typeof(*pos), member); \
     &pos->member != (head);  \
     pos = n, n = list_entry(n->member.next, typeof(*n), member))
// for WIN32
#define list_for_each_entry_safe(pos,pos_type, n,n_type, head, member) \
for (pos = list_entry((head)->next, pos_type, member), \
n = list_entry(pos->member.next, pos_type, member); \
     &pos->member != (head);  \
     pos = n, n = list_entry(n->member.next, n_type, member))

//HASH LIST
struct hlist_head {
```

```
    struct hlist_node *first;
};

struct hlist_node {
    struct hlist_node *next, **pprev;
};

#define HLIST_HEAD_INIT { .first = NULL }
#define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)
#define INIT_HLIST_NODE(ptr) ((ptr)->next = NULL, (ptr)->pprev = NULL)

static inline int hlist_unhashed(const struct hlist_node *h);
static inline int hlist_empty(const struct hlist_head *h);
static inline void __hlist_del(struct hlist_node *n);
static inline void hlist_del(struct hlist_node *n);
static inline void hlist_del_init(struct hlist_node *n);
static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h);
static inline void hlist_add_before(struct hlist_node *n,
struct hlist_node *next);
static inline void hlist_add_after(struct hlist_node *n,
struct hlist_node *next);



#define hlist_entry(ptr, type, member) container_of(ptr,type,member)

#define hlist_for_each(pos, head) \
for (pos = (head)->first; pos && ({ prefetch(pos->next); 1; }); \
     pos = pos->next)

#define hlist_for_each_safe(pos, n, head) \
for (pos = (head)->first; pos && ({ n = pos->next; 1; }); \
     pos = n)

// for LINUX
#define hlist_for_each_entry(tpos, pos, head, member)  \
for (pos = (head)->first;  \
     pos && ({ prefetch(pos->next); 1;}) &&  \
({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
     pos = pos->next)
// for WIN32
#define hlist_for_each_entry(tpos,tpos_type, pos, head, member)  \
for (pos = (head)->first;  \
     pos && ({ prefetch(pos->next); 1;}) &&  \
({ tpos = hlist_entry(pos, tpos_type, member); 1;}); \
     pos = pos->next)
// for LINUX
#define hlist_for_each_entry_continue(tpos, pos, member)  \
for (pos = (pos)->next;  \
     pos && ({ prefetch(pos->next); 1;}) &&  \
({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
     pos = pos->next)
```

```
// for WIN32
#define hlist_for_each_entry_continue(tpos,tpos_type, pos, member)  \
for (pos = (pos)->next;  \
     pos && ({ prefetch(pos->next); 1;}) &&  \
({ tpos = hlist_entry(pos, tpos_type, member); 1;}); \
     pos = pos->next)
// for LINUX
#define hlist_for_each_entry_from(tpos, pos, member)  \
for (; pos && ({ prefetch(pos->next); 1;}) &&  \
({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
     pos = pos->next)
// for WIN32
#define hlist_for_each_entry_from(tpos,tpos_type, pos, member)  \
for (; pos && ({ prefetch(pos->next); 1;}) &&  \
({ tpos = hlist_entry(pos, tpos_type, member); 1;}); \
     pos = pos->next)
// for LINUX
#define hlist_for_each_entry_safe(tpos, pos, n, head, member)   \
for (pos = (head)->first;  \
     pos && ({ n = pos->next; 1; }) &&   \
({ tpos = hlist_entry(pos, typeof(*tpos), member); 1;}); \
     pos = n)
// for WIN32
#define hlist_for_each_entry_safe(tpos,tpos_type, pos, n, head, member)   \
for (pos = (head)->first;  \
     pos && ({ n = pos->next; 1; }) &&   \
({ tpos = hlist_entry(pos, tpos_type, member); 1;}); \
     pos = n)

#ifdef __cplusplus
}
#endif
//////////////////////////////////////////////////////////////////////////
/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */
static inline void __list_add(struct list_head *newi,
      struct list_head *prev,
      struct list_head *next)
{
     next->prev = newi;
     newi->next = next;
     newi->prev = prev;
     prev->next = newi;
}
/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
```

```
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static inline void list_add(struct list_head *newi, struct list_head *head)
{
     __list_add(newi, head, head->next);
}
/**
 * list_add_tail - add a new entry
 * @new: new entry to be added
 * @head: list head to add it before
 *
 * Insert a new entry before the specified head.
 * This is useful for implementing queues.
 */
static inline void list_add_tail(struct list_head *newi, struct list_head *head)
{
     __list_add(newi, head->prev, head);
}

static inline void __list_del(struct list_head * prev, struct list_head * next)
{
     next->prev = prev;
     prev->next = next;
}

static inline void list_del(struct list_head *entry)
{
     __list_del(entry->prev, entry->next);
     entry->next = (struct list_head*)LIST_POISON1;
     entry->prev = (struct list_head*)LIST_POISON2;
}

static inline void list_del_init(struct list_head *entry)
{
     __list_del(entry->prev, entry->next);
     INIT_LIST_HEAD(entry);
}

static inline void list_move(struct list_head *list, struct list_head *head)
{
        __list_del(list->prev, list->next);
        list_add(list, head);
}

static inline void list_move_tail(struct list_head *list,
  struct list_head *head)
{
        __list_del(list->prev, list->next);
        list_add_tail(list, head);
}
```

```
static inline int list_empty(const struct list_head *head)
{
    return head->next == head;
}

static inline int list_empty_careful(const struct list_head *head)
{
    struct list_head *next = head->next;
    return (next == head) && (next == head->prev);
}

static inline void __list_splice(struct list_head *list,
 struct list_head *head)
{
    struct list_head *first = list->next;
    struct list_head *last = list->prev;
    struct list_head *at = head->next;

    first->prev = head;
    head->next = first;

    last->next = at;
    at->prev = last;
}

/**
 * list_splice - join two lists
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 */
static inline void list_splice(struct list_head *list, struct list_head *head)
{
    if (!list_empty(list))
        __list_splice(list, head);
}

/**
 * list_splice_init - join two lists and reinitialise the emptied list.
 * @list: the new list to add.
 * @head: the place to add it in the first list.
 *
 * The list at @list is reinitialised
 */
static inline void list_splice_init(struct list_head *list,
    struct list_head *head)
{
    if (!list_empty(list)) {
        __list_splice(list, head);
        INIT_LIST_HEAD(list);
    }
}
```

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

static inline int hlist_unhashed(const struct hlist_node *h)
{
    return !h->pprev;
}

static inline int hlist_empty(const struct hlist_head *h)
{
    return !h->first;
}

static inline void __hlist_del(struct hlist_node *n)
{
    struct hlist_node *next = n->next;
    struct hlist_node **pprev = n->pprev;
    *pprev = next;
    if (next)
        next->pprev = pprev;
}

static inline void hlist_del(struct hlist_node *n)
{
    __hlist_del(n);
    n->next = (struct hlist_node*)LIST_POISON1;
    n->pprev = (struct hlist_node**)LIST_POISON2;
}

static inline void hlist_del_init(struct hlist_node *n)
{
    if (n->pprev)  {
        __hlist_del(n);
        INIT_HLIST_NODE(n);
    }
}
static inline void hlist_add_head(struct hlist_node *n, struct hlist_head *h)
{
    struct hlist_node *first = h->first;
    n->next = first;
    if (first)
        first->pprev = &n->next;
    h->first = n;
    n->pprev = &h->first;
}



/* next must be != NULL */
static inline void hlist_add_before(struct hlist_node *n,
struct hlist_node *next)
{
    n->pprev = next->pprev;
    n->next = next;
```

```
    next->pprev = &n->next;
    *(n->pprev) = n;
}

static inline void hlist_add_after(struct hlist_node *n,
struct hlist_node *next)
{
    next->next = n->next;
    n->next = next;
    next->pprev = &n->next;

    if(next->next)
        next->next->pprev  = &next->next;
}

#endif
```

《链表测试代码》

```
    m_strList+="定义链表！\r\n";
    struct list_head list; //定义链表（头）

    m_strList+="初始化链表！\r\n";
    INIT_LIST_HEAD(&list); //初始化链表（头尾相接，形成空链表循环）

    //判断链表是否为空
    m_strList+="判断链表是否为空：";
    if(list_empty(&list)){
        m_strList+="判断链表是否为空：空\r\n";
    }else{
        m_strList+="判断链表是否为空：非空\r\n";
    }

    //批量添加节点
    m_strList+="批量添加节点：\r\n";
    for(int i=0;i<10;i++){
        int key=i;          //key
        int data=i*10;      //data
        CString node;
        node.Format("\t new node:key(%d),data(%d)\r\n",key,data);
        m_strList+=node;
        struct list_test_struct *st=(struct list_test_struct*)malloc(sizeof(struct list_test_struct));
        st->key=key;
        st->data=data;
        list_add(&st->list,&list);
    }

    //显示列表所有节点
    m_strList+="显示列表所有节点：\r\n";
    struct list_head *pos;
    list_for_each(pos,&list){
        struct list_test_struct *st=list_entry(pos,struct list_test_struct,list);
```

```
        CString node;
        node.Format("\t node:key(%d),data(%d)\r\n",st->key,st->data);
        m_strList+=node;
    }

    //释放所有节点资源
    m_strList+="释放所有节点资源！\r\n";
    struct list_head *n;
    list_for_each_safe(pos,n,&list){
        struct list_test_struct *st=list_entry(pos,struct list_test_struct,list);
        list_del(pos);     //删除节点，删除节点必须在删除节点内存之前
        free(st);     //释放节点内存
    }

    //判断链表是否为空
    m_strList+="判断链表是否为空：";
    if(list_empty(&list)){
        m_strList+="判断链表是否为空：空\r\n";
    }else{
        m_strList+="判断链表是否为空：非空\r\n";
    }
```