深入分析 Linux 内核链表

级别： 初级

杨沙洲 (pubb@163.net)国防科技大学计算机学院

2004 年 8 月 01 日

本文详细分析了 2.6.x 内核中链表结构的实现，并通过实例对每个链表操作接口进行了详尽的讲解。

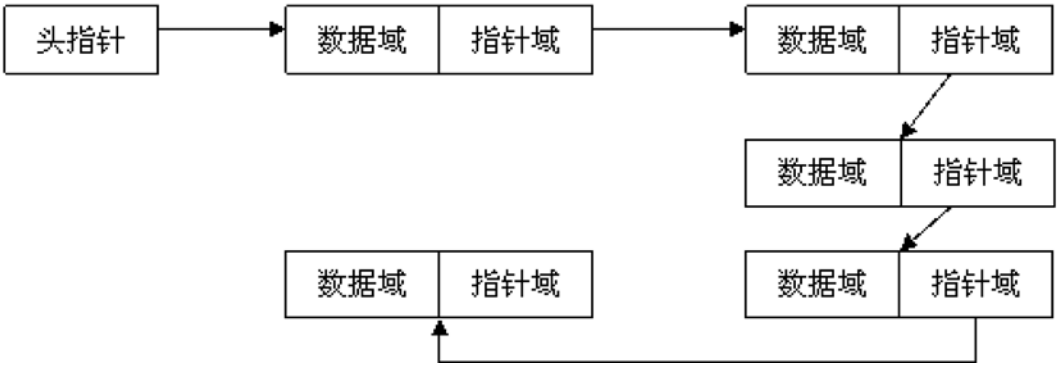
一、 链表数据结构简介

链表是一种常用的组织有序数据的数据结构，它通过指针将一系列数据节点连接成一条数据链，是线性表的一种重要实现方式。相对于数组，链表具有更好的动态性，建立链表时无需预先知道数据总量，可以随机分配空间，可以高效地在链表中的任意位置实时插入或删除数据。链表的开销主要是访问的顺序性和组织链的空间损失。

通常链表数据结构至少应包含两个域：数据域和指针域，数据域用于存储数据，指针域用于建立与下一个节点的联系。按照指针域的组织以及各个节点之间的联系形式，链表又可以分为单链表、双链表、循环链表等多种类型，下面分别给出这几类常见链表类型的示意图：

1. 单链表

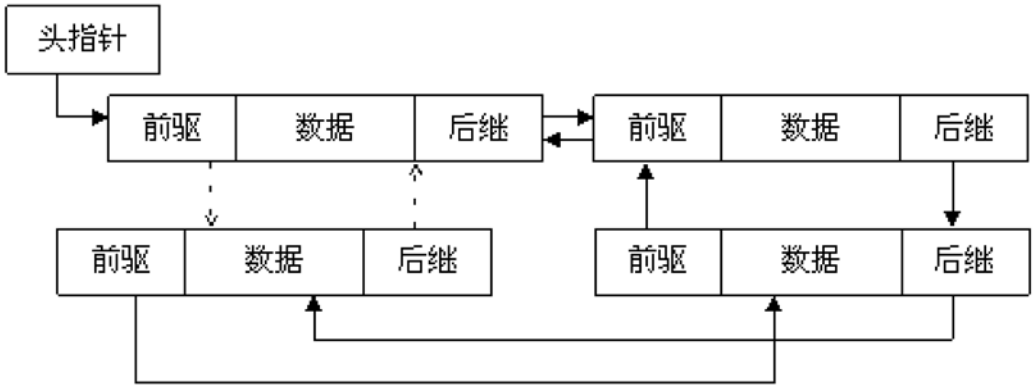
图1 单链表



单链表是最简单的一类链表，它的特点是仅有一个指针域指向后继节点（next），因此，对单链表的遍历只能从头至尾（通常是NULL空指针）顺序进行。

2. 双链表

图2 双链表



通过设计前驱和后继两个指针域，双链表可以从两个方向遍历，这是它区别于单链表的地方。如果打乱前驱、后继的依赖关系，就可以构成"二叉树"；如果再让首节点的前驱指向链表尾节点、尾节点的后继指向首节点（如图2中虚线部分），就构成了循环链表；如果设计更多的指针域，就可以构成各种复杂的树状数据结构。

3. 循环链表

循环链表的特点是尾节点的后继指向首节点。前面已经给出了双循环链表的示意图，它的特点是从任意一个节点出发，沿两个方向的任何一个，都能找到链表中的任意一个数据。如果去掉前驱指针，就是单循环链表。

在Linux内核中使用了大量的链表结构来组织数据，包括设备列表以及各种功能模块中的数据组织。这些链表大多采用在[include/linux/list.h]实现的一个相当精彩的链表数据结构。本文的后继部分就将通过示例详细介绍这一数据结构的组织和使用。

二、 Linux 2.6内核链表数据结构的实现

尽管这里使用2.6内核作为讲解的基础，但实际上2.4内核中的链表结构和2.6并没有什么区别。不同之处在于2.6扩充了两种链表数据结构：链表的读拷贝更新（rcu）和HASH链表（hlist）。这两种扩展都是基于最基本的list结构，因此，本文主要介绍基本链表结构，然后再简要介绍一下rcu和hlist。

链表数据结构的定义很简单（节选自[include/linux/list.h]，以下所有代码，除非加以说明，其余均取自该文件）：

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

list_head结构包含两个指向list_head结构的指针prev和next，由此可见，内核的链表具备双链表功能，实际上，通常它都组织成双循环链表。

和第一节介绍的双链表结构模型不同，这里的list_head没有数据域。在Linux内核链表中，不是在链表结构中包含数据，而是在数据结构中包含链表节点。

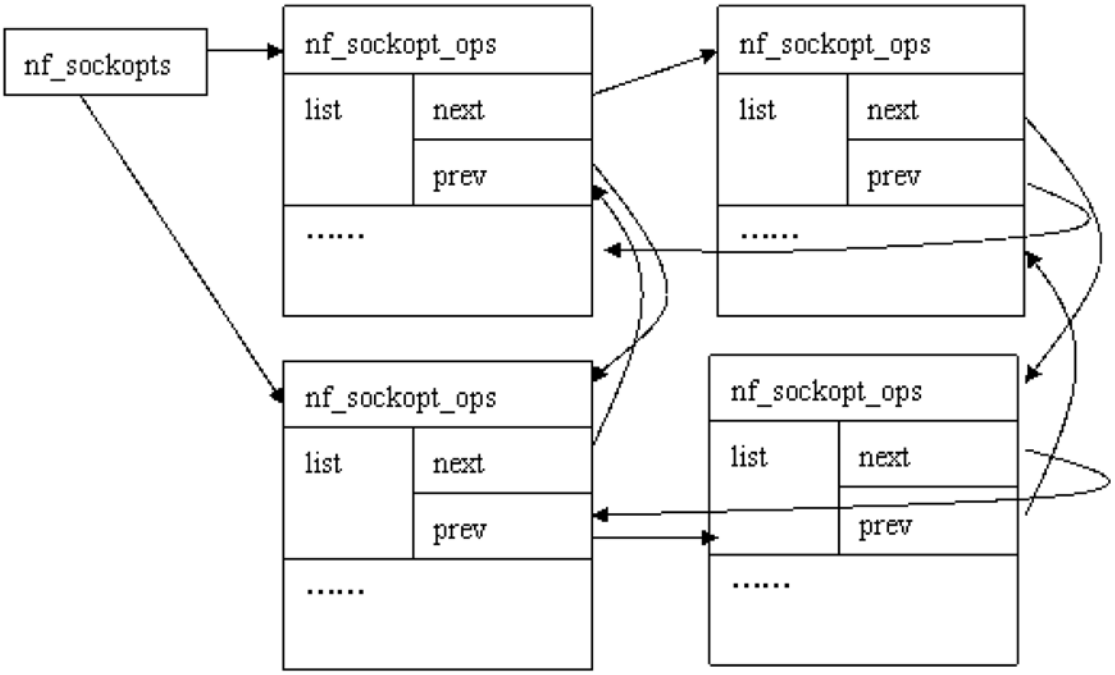
在数据结构课本中，链表的经典定义方式通常是这样的（以单链表为例）：

```
struct list_node {
        struct list_node *next;
        ElemType        data;
};
```

因为ElemType的缘故，对每一种数据项类型都需要定义各自的链表结构。有经验的C++程序员应该知道，标准模板库中的<list>采用的是C++ Template，利用模板抽象出和数据项类型无关的链表操作接口。

在Linux内核链表中，需要用链表组织起来的数据通常会包含一个struct list_head成员，例如在[include/linux/netfilter.h]中定义了一个nf_sockopt_ops结构来描述Netfilter为某一协议族准备的getsockopt/setsockopt接口，其中就有一个（struct list_head list）成员，各个协议族的nf_sockopt_ops结构都通过这个list成员组织在一个链表中，表头是定义在[net/core/netfilter.c]中的nf_sockopts（struct list_head）。从下图中我们可以看到，这种通用的链表结构避免了为每个数据项类型定义自己的链表的麻烦。Linux的简捷实用、不求完美和标准的风格，在这里体现得相当充分。

图3 nf_sockopts链表示意图



三、 链表操作接口

1. 声明和初始化

实际上Linux只定义了链表节点，并没有专门定义链表头，那么一个链表结构是如何建立起来的呢？让我们来看看LIST_HEAD()这个宏：

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) struct list_head name = LIST_HEAD_INIT(name)
```

当我们用LIST_HEAD(nf_sockopts)声明一个名为nf_sockopts的链表头时，它的next、prev指针都初始化为指向自己，这样，我们就有了一个空链表，因为Linux用头指针的next是否指向自己来判断链表是否为空：

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

除了用LIST_HEAD()宏在声明的时候初始化一个链表以外，Linux还提供了一个INIT_LIST_HEAD宏用于运行时初始化链表：

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

我们用INIT_LIST_HEAD(&nf_sockopts)来使用它。

2. 插入/删除/合并

a) 插入

对链表的插入操作有两种：在表头插入和在表尾插入。Linux为此提供了两个接口：

```
static inline void list_add(struct list_head *new, struct list_head *head);
static inline void list_add_tail(struct list_head *new, struct list_head *head);
```

因为Linux链表是循环表，且表头的next、prev分别指向链表中的第一个和最末一个节点，所以，list_add和list_add_tail的区别并不大，实际上，Linux分别用

```
__list_add(new, head, head->next);
```

和

```
__list_add(new, head->prev, head);
```

来实现两个接口，可见，在表头插入是插入在head之后，而在表尾插入是插入在head->prev之后。

假设有一个新nf_sockopt_ops结构变量new_sockopt需要添加到nf_sockopts链表头，我们应当这样操作：

```
list_add(&new_sockopt.list, &nf_sockopts);
```

从这里我们看出，nf_sockopts链表中记录的并不是new_sockopt的地址，而是其中的list元素的地址。如何通过链表访问到new_sockopt呢？下面会有详细介绍。

b) 删除

```
static inline void list_del(struct list_head *entry);
```

当我们需要删除nf_sockopts链表中添加的new_sockopt项时，我们这么操作：

```
list_del(&new_sockopt.list);
```

被剔除下来的new_sockopt.list，prev、next指针分别被设为LIST_POSITION2和LIST_POSITION1两个特殊值，这样设置是为了保证不在链表中的节点项不可访问--对LIST_POSITION1和LIST_POSITION2的访问都将引起页故障。与之相对应，list_del_init()函数将节点从链表中解下来之后，调用LIST_INIT_HEAD()将节点置为空链状态。

c) 搬移

Linux提供了将原本属于一个链表的节点移动到另一个链表的操作，并根据插入到新链表的位置分为两类：

```
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);
```

例如list_move(&new_sockopt.list,&nf_sockopts)会把new_sockopt从它所在的链表上删除，并将其再链入nf_sockopts的表头。

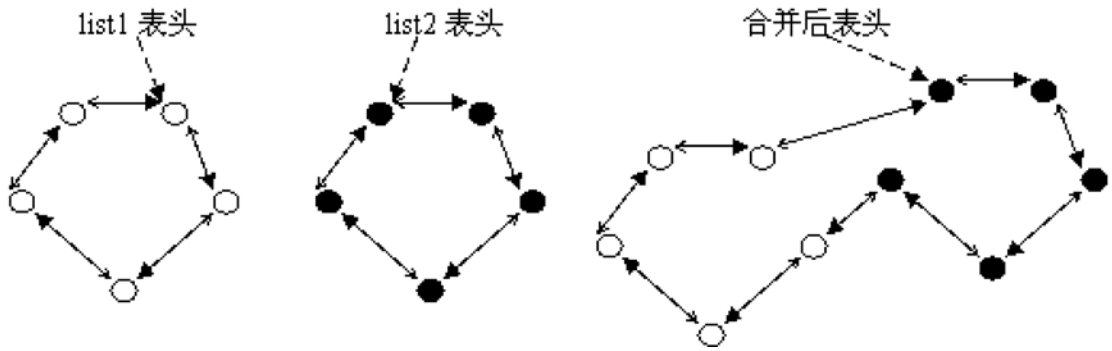
d) 合并

除了针对节点的插入、删除操作，Linux链表还提供了整个链表的插入功能：

```
static inline void list_splice(struct list_head *list, struct list_head *head);
```

假设当前有两个链表，表头分别是list1和list2（都是struct list_head变量），当调用list_splice(&list1,&list2)时，只要list1非空，list1链表的内容将被挂接在list2链表上，位于list2和list2.next（原list2表的第一个节点）之间。新list2链表将以原list1表的第一个节点为首节点，而尾节点不变。如图（虚箭头为next指针）：

图4 链表合并list_splice(&list1,&list2)



当list1被挂接到list2之后，作为原表头指针的list1的next、prev仍然指向原来的节点，为了避免引起混乱，Linux提供了一个list_splice_init()函数：

```
        static inline void list_splice_init(struct list_head *list, struct list_head *head);
```

该函数在将list合并到head链表的基础上，调用INIT_LIST_HEAD(list)将list设置为空链。

3. 遍历

遍历是链表最经常的操作之一，为了方便核心应用遍历链表，Linux链表将遍历操作抽象成几个宏。在介绍遍历宏之前，我们先看看如何从链表中访问到我们真正需要的数据项。

a) 由链表节点到数据项变量

我们知道，Linux链表中仅保存了数据项结构中list_head成员变量的地址，那么我们如何通过这个list_head成员访问到作为它的所有者的节点数据呢？Linux为此提供了一个list_entry(ptr,type,member)宏，其中ptr是指向该数据中list_head成员的指针，也就是存储在链表中的地址值，type是数据项的类型，member则是数据项类型定义中list_head成员的变量名，例如，我们要访问nf_sockopts链表中首个nf_sockopt_ops变量，则如此调用：

```
list_entry(nf_sockopts->next, struct nf_sockopt_ops, list);
```

这里"list"正是nf_sockopt_ops结构中定义的用于链表操作的节点成员变量名。

list_entry的使用相当简单，相比之下，它的实现则有一些难懂：

```
#define list_entry(ptr, type, member) container_of(ptr, type, member)
container_of宏定义在[include/linux/kernel.h]中：
#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})
offsetof宏定义在[include/linux/stddef.h]中：
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

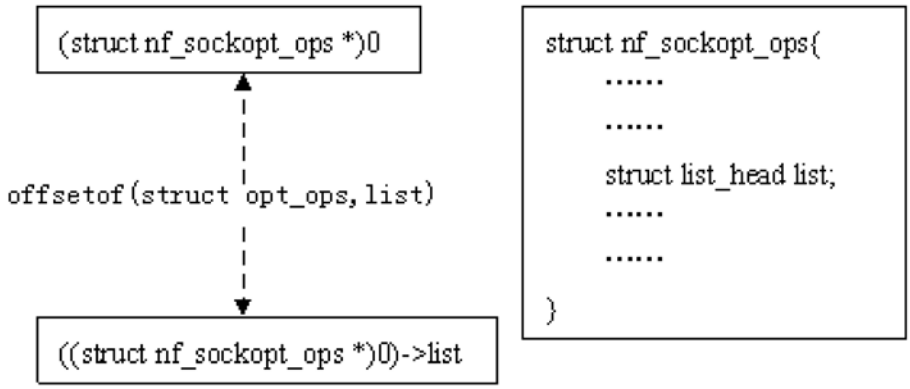
size_t最终定义为unsigned int（i386）。

这里使用的是一个利用编译器技术的小技巧，即先求得结构成员在与结构中的偏移量，然后根据成员变量的地址反过来得出属主结构变量的地址。

container_of()和offsetof()并不仅用于链表操作，这里最有趣的地方是((type *)0)->member，它将0地址强制"转换"为type结构的指针，再访问到type结构中的member成员。在container_of宏中，它用来给typeof()提供参数（typeof()是gcc的扩展，和sizeof()类似），以获得member成员的数据类型；在offsetof()中，这个member成员的地址实际上就是type数据结构中member成员相对于结构变量的偏移量。

如果这么说还不好理解的话，不妨看看下面这张图：

图5 offsetof()宏的原理



对于给定一个结构，offsetof(type,member)是一个常量，list_entry()正是利用这个不变的偏移量来求得链表数据项的变量地址。

b) 遍历宏

在[net/core/netfilter.c]的nf_register_sockopt()函数中有这么一段话：

```
                ……
struct list_head *i;
……
        list_for_each(i, &nf_sockopts) {
                struct nf_sockopt_ops *ops = (struct nf_sockopt_ops *)i;
                ……
        }
        ……
```

函数首先定义一个(struct list_head *)指针变量i，然后调用list_for_each(i,&nf_sockopts)进行遍历。在[include/linux/list.h]中，list_for_each()宏是这么定义的：

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

它实际上是一个for循环，利用传入的pos作为循环变量，从表头head开始，逐项向后（next方向）移动pos，直至又回到head（prefetch()可以不考虑，用于预取以提高遍历速度）。

那么在nf_register_sockopt()中实际上就是遍历nf_sockopts链表。为什么能直接将获得的list_head成员变量地址当成struct nf_sockopt_ops数据项变量的地址呢？我们注意到在struct nf_sockopt_ops结构中，list是其中的第一项成员，因此，它的地址也就是结构变量的地址。更规范的获得数据变量地址的用法应该是：

```
struct nf_sockopt_ops *ops = list_entry(i, struct nf_sockopt_ops, list);
```

大多数情况下，遍历链表的时候都需要获得链表节点数据项，也就是说list_for_each()和list_entry()总是同时使用。对此Linux给出了一个list_for_each_entry()宏：

```
#define list_for_each_entry(pos, head, member)                  ……
```

与list_for_each()不同，这里的pos是数据项结构指针类型，而不是(struct list_head *)。nf_register_sockopt()函数可以利用这个宏而设计得更简单：

```
……
struct nf_sockopt_ops *ops;
list_for_each_entry(ops,&nf_sockopts,list){
        ……
```

```
}
……
```

某些应用需要反向遍历链表，Linux提供了list_for_each_prev()和list_for_each_entry_reverse()来完成这一操作，使用方法和上面介绍的list_for_each()、list_for_each_entry()完全相同。

如果遍历不是从链表头开始，而是从已知的某个节点pos开始，则可以使用list_for_each_entry_continue(pos,head,member)。有时还会出现这种需求，即经过一系列计算后，如果pos有值，则从pos开始遍历，如果没有，则从链表头开始，为此，Linux专门提供了一个list_prepare_entry(pos,head,member)宏，将它的返回值作为list_for_each_entry_continue()的pos参数，就可以满足这一要求。

4. 安全性考虑

在并发执行的环境下，链表操作通常都应该考虑同步安全性问题，为了方便，Linux将这一操作留给应用自己处理。Linux链表自己考虑的安全性主要有两个方面：

a) list_empty()判断

基本的list_empty()仅以头指针的next是否指向自己来判断链表是否为空，Linux链表另行提供了一个list_empty_careful()宏，它同时判断头指针的next和prev，仅当两者都指向自己时才返回真。这主要是为了应付另一个cpu正在处理同一个链表而造成next、prev不一致的情况。但代码注释也承认，这一安全保障能力有限：除非其他cpu的链表操作只有list_del_init()，否则仍然不能保证安全，也就是说，还是需要加锁保护。

b) 遍历时节点删除

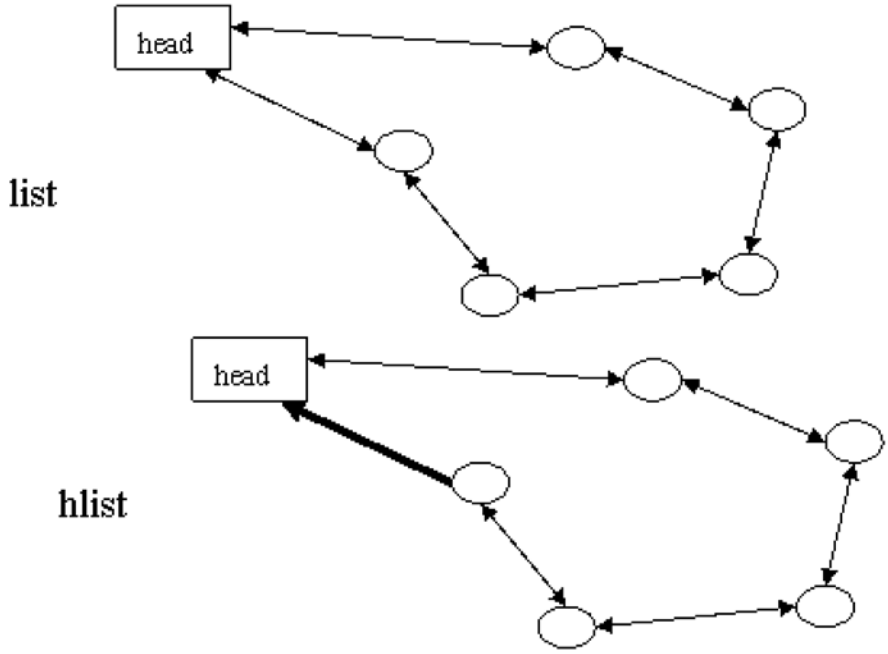
前面介绍了用于链表遍历的几个宏，它们都是通过移动pos指针来达到遍历的目的。但如果遍历的操作中包含删除pos指针所指向的节点，pos指针的移动就会被中断，因为list_del(pos)将把pos的next、prev置成LIST_POSITION2和LIST_POSITION1的特殊值。

当然，调用者完全可以自己缓存next指针使遍历操作能够连贯起来，但为了编程的一致性，Linux链表仍然提供了两个对应于基本遍历操作的"_safe"接口：list_for_each_safe(pos, n, head)、list_for_each_entry_safe(pos, n, head, member)，它们要求调用者另外提供一个与pos同类型的指针n，在for循环中暂存pos下一个节点的地址，避免因pos节点被释放而造成的断链。

四、 扩展

1. hlist

图6 list和hlist



精益求精的Linux链表设计者（因为list.h没有署名，所以很可能就是Linus Torvalds）认为双头（next、prev）的双链表对于HASH表来说"过于浪费"，因而另行设计了一套用于HASH表应用的hlist数据结构--单指针表头双循环链表，从上图可以看出，hlist的表头仅有一个指向首节点的指针，而没有指向尾节点的指针，这样在可能是海量的HASH表中存储的表头就能减少一半的空间消耗。

因为表头和节点的数据结构不同，插入操作如果发生在表头和首节点之间，以往的方法就行不通了：表头的first指针必须修改指向新插入的节点，却不能使用类似list_add()这样统一的描述。为此，hlist节点的prev不再是指向前一个节点的指针，而是指向前一个节点（可能是表头）中的next（对于表头则是first）指针（struct list_head **pprev），从而在表头插入的操作可以通过一致的"*(node->pprev)"访问和修改前驱节点的next（或first）指针。

2. read-copy update

在Linux链表功能接口中还有一系列以"_rcu"结尾的宏，与以上介绍的很多函数一一对应。RCU（Read-Copy Update）是2.5/2.6内核中引入的新技术，它通过延迟写操作来提高同步性能。

我们知道，系统中数据读取操作远多于写操作，而rwlock机制在smp环境下随着处理机增多性能会迅速下降（见参考资料4）。针对这一应用背景，IBM Linux技术中心的Paul E. McKenney提出了"读拷贝更新"的技术，并将其应用于Linux内核中。RCU技术的核心是写操作分为写-更新两步，允许读操作在任何时候无阻访问，当系统有写操作时，更新动作一直延迟到对该数据的所有读操作完成为止。Linux链表中的RCU功能只是Linux RCU的很小一部分，对于RCU的实现分析已超出了本文所及，有兴趣的读者可以自行参阅本文的参考资料；而对RCU链表的使用和基本链表的使用方法基本相同。

五、 示例

附件中的程序除了能正向、反向输出文件以外，并无实际作用，仅用于演示Linux链表的使用。

为了简便，例子采用的是用户态程序模板，如果需要运行，可采用如下命令编译：

```
gcc -D__KERNEL__ -I/usr/src/linux-2.6.7/include pfile.c -o pfile
```

因为内核链表限制在内核态使用，但实际上对于数据结构本身而言并非只能在核态运行，因此，在笔者的编译中使用"-D__KERNEL__"开关"欺骗"编译器。

参考资料

1. 维基百科 http://zh.wikipedia.org，一个在GNU Documentation License下发布的网络辞典，自由软件理念的延伸，本文的"链表"概念即使用它的版本。
2. 《Linux内核情景分析》，毛德操先生的这本关于Linux内核的巨著几乎可以回答绝大部分关于内核的问题，其中也包括内核链表的几个关键数据结构。
3. Linux内核2.6.7源代码，所有不明白的问题，只要潜心看代码，总能清楚。
4. Kernel Korner: Using RCU in the Linux 2.5 Kernel，RCU主要开发者Paul McKenney 2003年10月发表于Linux Journal上的一篇介绍RCU的文章。在 http://www.rdrop.com/users/paulmck/rclock/上可以获得更多关于RCU的帮助。

关于作者

杨沙洲，目前在国防科技大学计算机学院攻读软件方向博士学位。对文中存在的技术问题，欢迎向 pubb@163.net质疑。