内核数据结构

主要内容：

* 链表
* 队列
* 映射
* 红黑树

# 链表

链表是linux内核中最简单，同时也是应用最广的数据结构。

内核中定义的是双向链表。

1.1头文件简介

Include/linxu/list.h

其实刚开始只要了解一个常用的链表操作（追加，删除，遍历）的实现方法，其他方法基本都是基于这些操作的。

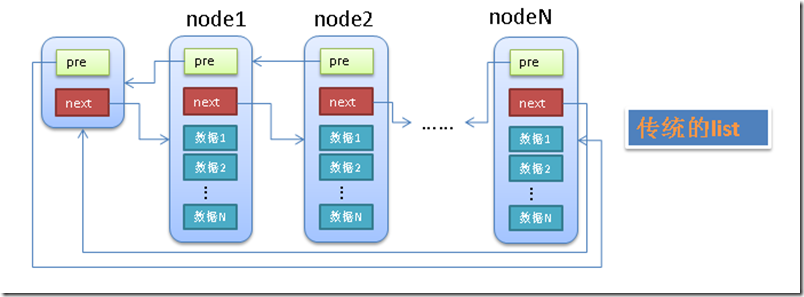
1.2链表代码的注意点

在阅读list.h文件之前，必须注意：linux内核中的链表使用方法和一般数据结构中的链表是有所不同的。

一般是双向链表的如下结构：

* 有个单独的头节点（head）
* 每个节点（node）除了包含必要的数据之外，还有两个指针（pre，next）
* 头节点（head）的pre指针指向链表的最后一个节点
* 最后一个节点的next指针指向头结点（head）

具体见下图：

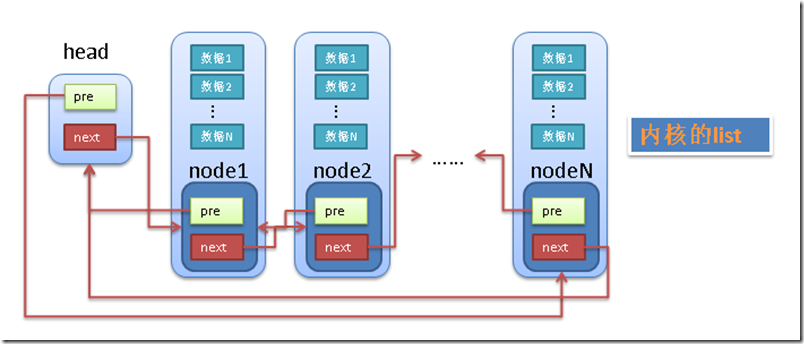


传统的链表有个最大的缺点就是不好共通化，因为每个node中的data1，data2等都是不确定的（无论是个数还是类型）。

Linux中的链表巧妙的解决了这个问题，linux的链表不是将用户数据保存在链表节点中，而是将链表节点保存在用户数据中。

Linux的链表节点只有两个指针（pre和next），这样的话，链表的节点将独立与用户数据之外，便于链表的共同操作。

具体实现见下图：



Linux链表中的最大问题是怎样通过链表的节点来取得用户数据？

和传统链表不同，linux的链表节点node中没有包含用户的用户data1，data2等。

整个list.h文件中，我觉得最复杂的代码就是获取用户数据的宏定义

#define **list\_entry**(ptr, type, member) \

container\_of(ptr, type, member)

这个宏没有什么特别的，主要是container\_of这个宏

#define **container\_of**(ptr, type, member) ({ \

const typeof( ((type \*)0)->member ) \*\_\_mptr = (ptr); \

(type \*)( (char \*)\_\_mptr - offsetof(type,member) );})

这里的type是一个结构体，也就是包含用户数据和链表节点的结构体。

ptr是指向type中链表节点的指针

member则是type中定义链表节点时用的名字

比如：

struct student

{

int id;

char \*name;

struct list\_head list;

}

type是struct student

ptr是指向struct list的指针

membrt就是list

下面分析一下container\_of宏：

/\* 步骤1：将数字0强制转型为type \*，然后取得其中的member元素 \*/

((type \*)0)->member // 相当于((struct student \*)0)->list

/\* 步骤2：定义一个临时变量\_\_mptt，并将其也指向ptr指向的链表节点 \*/

const typeof(((type \*)0)->member)\*\_\_mptr = (ptr);

/\* 步骤3：计算member字段距离type中第一个字段的距离，也就是tpye地址也member地址之间的差 \*/

/\* offsetof(type, member)也是一个宏，定义如下 \*/

#define offsetof(TYPE, MEMBER) ((size\_t) &((TYPE \*)0)->MEMBER)

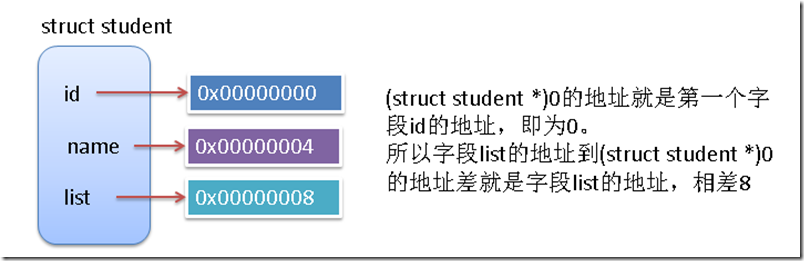
/\* 步骤4：将\_\_mptr的地址 – type地址和member地址之间的差 \*/

/\* 也就是获取type的地址 \*/

步骤1,2,4比较容易理解，下面的图以struct student为例进行说明步骤3：

首先需要知道( (TYPE \*)0 )表示将地址0转换为TYPE类型的地址

由于TYPE的地址的0，所有( (TYPE \*)0 )->MEMBER也就是MEMBER的地址和TYPE的地址的差，图下图所示：



1.3使用示例

构造了一个内核模块来实际使用一些内核中的链表：

#include<linux/init.h>

#include<linux/slab.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/list.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

struct list\_head list;

};

void print\_student(struct student\*);

static int testlist\_init(void)

{

struct student \*stu1, \*stu2, \*stu3, \*stu4;

struct student \*stu;

// init a list head

LIST\_HEAD(stu\_head);

// init four list nodes

stu1 = kmalloc(sizeof(\*stu1), GFP\_KERNEL);

stu1->id = 1;

stu1->name = "wyb";

INIT\_LIST\_HEAD(&stu1->list);

stu2 = kmalloc(sizeof(\*stu2), GFP\_KERNEL);

stu2->id = 2;

stu2->name = "wyb2";

INIT\_LIST\_HEAD(&stu2->list);

stu3 = kmalloc(sizeof(\*stu3), GFP\_KERNEL);

stu3->id = 3;

stu3->name = "wyb3";

INIT\_LIST\_HEAD(&stu3->list);

stu4 = kmalloc(sizeof(\*stu4), GFP\_KERNEL);

stu4->id = 4;

stu4->name = "wyb4";

INIT\_LIST\_HEAD(&stu4->list);

// add the four nodes to head

list\_add (&stu1->list, &stu\_head);

list\_add (&stu2->list, &stu\_head);

list\_add (&stu3->list, &stu\_head);

list\_add (&stu4->list, &stu\_head);

// print each student from 4 to 1

list\_for\_each\_entry(stu, &stu\_head, list)

{

print\_student(stu);

}

// print each student from 1 to 4

list\_for\_each\_entry\_reverse(stu, &stu\_head, list)

{

print\_student(stu);

}

// delete a entry stu2

list\_del(&stu2->list);

list\_for\_each\_entry(stu, &stu\_head, list)

{

print\_student(stu);

}

// replace stu3 with stu2

list\_replace(&stu3->list, &stu2->list);

list\_for\_each\_entry(stu, &stu\_head, list)

{

print\_student(stu);

}

return 0;

}

static void testlist\_exit(void)

{

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "testlist is exited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

void print\_student(struct student \*stu)

{

printk (KERN\_ALERT "======================\n");

printk (KERN\_ALERT "id =%d\n", stu->id);

printk (KERN\_ALERT "name=%s\n", stu->name);

printk (KERN\_ALERT "======================\n");

}

module\_init(testlist\_init);

module\_exit(testlist\_exit);

Makefile：

obj-m += testlist.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

安装,卸载内核模块以及查看内核模块的运行结果：

insmod testlist.ko

rmmod testlist

dmesg | tail -100

# 队列

内核中的队列是以字节形式保存数据的，所有获取数据的时候，需要知道数据的大小。

如果从队列中取得数据时指定的大小不对的话，取得数据会不完整或者过大。

## 2.1头文件简介

内核中关于队列定于的头文件位于：<linux/kfifo.h>

头文件中定义的函数的实现位于<linux.kfifo.c>

## 2.2队列代码的注意点

* 队列的size在初始化时，始终设定为2的n次方
* 使用队列之前将队列结构体中的锁（spinlock）释放

## 2.3使用示例

[复制代码](javascript:void(0);)

#include "kn\_common.h"

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

int id;

char\* name;

};

static void print\_student(struct student\*);

static int testkfifo\_init(void)

{

struct kfifo \*fifo;

struct student \*stu1, \*stu2, \*stu3, \*stu4;

struct student \*stu\_tmp;

char\* c\_tmp;

int i;

// !!importent init a unlocked lock

spinlock\_t sl = SPIN\_LOCK\_UNLOCKED;

// init kfifo

fifo = kfifo\_alloc(4\*sizeof(struct student), GFP\_KERNEL, &sl);

stu1 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu1->id = 1;

stu1->name = "wyb1";

kfifo\_put(fifo, (char \*)stu1, sizeof(struct student));

stu2 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu2->id = 1;

stu2->name = "wyb2";

kfifo\_put(fifo, (char \*)stu2, sizeof(struct student));

stu3 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu3->id = 1;

stu3->name = "wyb3";

kfifo\_put(fifo, (char \*)stu3, sizeof(struct student));

stu4 = kmalloc(sizeof(struct student), GFP\_KERNEL);

stu4->id = 1;

stu4->name = "wyb4";

kfifo\_put(fifo, (char \*)stu4, sizeof(struct student));

c\_tmp = kmalloc(sizeof(struct student), GFP\_KERNEL);

printk(KERN\_ALERT "current fifo length is : %d\n", kfifo\_len(fifo));

for (i=0; i < 4; i++) {

kfifo\_get(fifo, c\_tmp, sizeof(struct student));

stu\_tmp = (struct student \*)c\_tmp;

print\_student(stu\_tmp);

printk(KERN\_ALERT "current fifo length is : %d\n", kfifo\_len(fifo));

}

printk(KERN\_ALERT "current fifo length is : %d\n", kfifo\_len(fifo));

kfifo\_free(fifo);

kfree(c\_tmp);

return 0;

}

static void print\_student(struct student \*stu)

{

printk(KERN\_ALERT "=========================\n");

print\_current\_time(1);

printk(KERN\_ALERT "id = %d\n", stu->id);

printk(KERN\_ALERT "name = %s\n", stu->name);

printk(KERN\_ALERT "=========================\n");

}

static void testkfifo\_exit(void)

{

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printk(KERN\_ALERT "testkfifo is exited!\n");

printk(KERN\_ALERT "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

module\_init(testkfifo\_init);

module\_exit(testkfifo\_exit);

[复制代码](javascript:void(0);)

其中引用的kn\_common.h文件：

[复制代码](javascript:void(0);)

#include<linux/init.h>

#include<linux/slab.h>

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/kfifo.h>

#include<linux/time.h>

void print\_current\_time(int);

[复制代码](javascript:void(0);)

kn\_common.h对应的kn\_common.c：

[复制代码](javascript:void(0);)

#include "kn\_common.h"

void print\_current\_time(int is\_new\_line)

{

struct timeval \*tv;

struct tm \*t;

tv = kmalloc(sizeof(struct timeval), GFP\_KERNEL);

t = kmalloc(sizeof(struct tm), GFP\_KERNEL);

do\_gettimeofday(tv);

time\_to\_tm(tv->tv\_sec, 0, t);

printk(KERN\_ALERT "%ld-%d-%d %d:%d:%d",

t->tm\_year + 1900,

t->tm\_mon + 1,

t->tm\_mday,

(t->tm\_hour + 8) % 24,

t->tm\_min,

t->tm\_sec);

if (is\_new\_line == 1)

printk(KERN\_ALERT "\n");

kfree(tv);

kfree(t);

}

[复制代码](javascript:void(0);)

Makefile：

[复制代码](javascript:void(0);)

obj-m += fifo.o

fifo-objs := testkfifo.o kn\_common.o

#generate the path

CURRENT\_PATH:=$(shell pwd)

#the current kernel version number

LINUX\_KERNEL:=$(shell uname -r)

#the absolute path

LINUX\_KERNEL\_PATH:=/usr/src/kernels/$(LINUX\_KERNEL)

#complie object

all:

make -C $(LINUX\_KERNEL\_PATH) M=$(CURRENT\_PATH) modules

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c .tmp\_versions \*.unsigned

#clean

clean:

rm -rf modules.order Module.symvers .\*.cmd \*.o \*.mod.c \*.ko .tmp\_versions \*.unsigned

[复制代码](javascript:void(0);)

安装,卸载内核模块以及查看内核模块的运行结果：

insmod fifo.ko

rmmod fifo

dmesg | tail -40

# 映射

映射有点像其他语言（c#或者python）中的指点类型，每个唯一的id对应一个自定义的数据结构。

## 3.1头文件简介

头文件：<linux/idr.h>

c文件：<lib/idr.c>

# 4.红黑树

红黑树由于节点颜色的特性，保证其是一种自平衡的二叉搜索树。

红黑树的一系列规则虽然实现起来比较复杂，但是遵循起来却比较简单，而且红黑树的插入，删除性能也还不错。

所以红黑树在内核中的应用非常广泛，掌握好红黑树，即有利于阅读内核源码，也可以在自己的代码中借鉴这种数据结构。

红黑树必须满足的规则：

* 所有节点都有颜色，要么红色，要么黑色
* 根节点是黑色，所有叶子节点也是黑色
* 叶子节点中不包含数据
* 非叶子节点都有2个子节点
* 如果一个节点是红色，那么他的父节点和子节点都是黑色的
* 从任何一个节点开始，到其叶子节点的路径中都包含相同数目的黑节点

红黑树中最长的路径就是红黑树交替的路径，最短的路径是全黑节点的路径，再加上根节点和叶子节点都是黑色，从而可以保证红黑树中最长路径的长度不会超过最短路径。

4.1头文件简介

<linux/rbtree.h>

<lib/rbtree.c>

4.2红黑树代码的注意点

内核中红黑树的使用链表（list）有些类似，是将红黑树的节点放入自定义的数据结构中来使用的。

首先需要注意的一点是红黑树节点的定义：

struct rb\_node

{

unsigned long rb\_parent\_color;

#define RB\_RED 0

#define RB\_BLACK 1

struct rb\_node \*rb\_right;

struct rb\_node \*rb\_left;

} \_\_attribute\_\_((aligned(sizeof(long))));

刚开始看到这个定义的时候，我觉得奇怪，等到看懂了之后，才知道原来作者巧妙的利用内存对齐来将2个内容存入到一个字段中（不服不行啊）。

字段rb\_parent\_color中保存了2个信息：

1. 父节点的地址
2. 子节点的颜色

这2个信息是如何存入一个字段的呢？主要在于\_\_attribute\_\_((aligned(sizeof(long))));

这行代码的意思就是struct rb\_node在内存中的地址需要按照4 bytes或者8 bytes对齐。

注：sizeof(long)在32bit系统中是4 bytes，在64bit系统中是8 bytes

struct tb\_node的地址按4 bytes对齐，意味着分配的地址都是4的倍数。

4的二进制为100，所以申请分配的struct rb\_node的地址的最后两位始终是0，

struct rb\_node的字段rb\_parent\_color就是利用最后一位来保存节点的颜色信息的。

明白了这一点，rb\_tree.h中很多红的定义也就很好懂了。

/\* rb\_parent\_color 保存了父节点的地址和本节点的颜色 \*/

/\* 将 rb\_parent\_color 的最后2位置成0，即将颜色信息去掉，剩下的就是parent节点的地址 \*/

#define rb\_parent(r) ((struct rb\_node \*)((r)->rb\_parent\_color & ~3))

/\* 取得 rb\_parent\_color 二进制表示的最后一位，即用于保存颜色信息的那一位 \*/

#define rb\_color(r) ((r)->rb\_parent\_color & 1)

/\* 将 rb\_parent\_color 二进制表示的最后一位置为0，即置为红色 \*/

#define rb\_set\_red(r) do { (r)->rb\_parent\_color &= ~1; } while (0)

/\* 将 rb\_parent\_color 二进制表示的最后一位置为1，即置为黑色 \*/

#define rb\_set\_black(r) do { (r)->rb\_parent\_color |= 1; } while (0)

还需要重点看的就是rb\_tree.c中的5个函数，下面对着5个函数进行一些注释：

函数1：

http://www.cnblogs.com/wang\_yb/archive/2013/04/16/3023892.html