**主要内容：**

* 链表
* 队列
* 映射
* 红黑树

### **1. 链表**

#### **1.1 头文件简介**

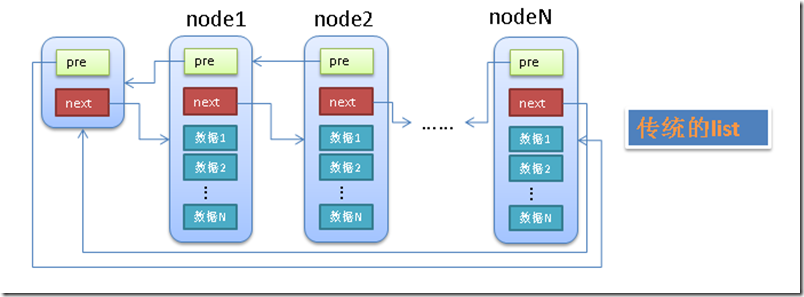
内核中关于链表定义的代码位于： include/linux/list.h

#### **1.2 链表代码的注意点**

在阅读list.h文件之前，有一点必须注意：linux内核中的链表使用方法和一般数据结构中定义的链表是有所不同的。

一般的双向链表一般是如下的结构，

具体见下图：

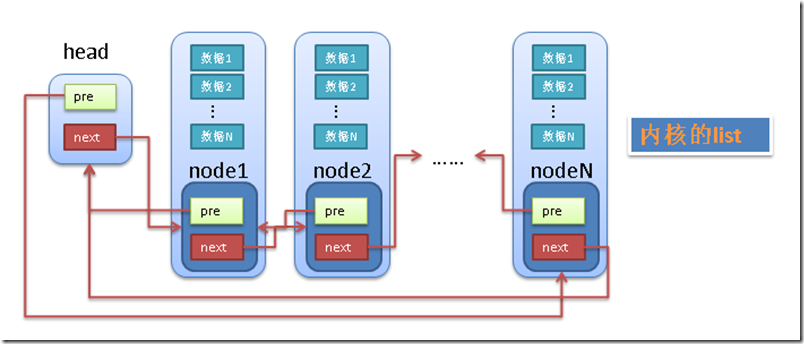
[](https://images0.cnblogs.com/blog/83005/201304/16132037-c83d5c19780d4c13ade4964ed38e6af7.png)

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2个指针(pre和next)，这样的话，链表的节点将独立于用户数据之外，便于实现链表的共同操作。

具体见下图：

[](https://images0.cnblogs.com/blog/83005/201304/16132043-3ae739a1effd40a4afe22a5661018919.png)

#### **1.3 使用示例**

构造了一个内核模块来实际使用一下内核中的链表，代码在CentOS6.3 x64上运行通过。

C代码：

/\* testmodule.c \*/

#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;

    // list\_head 是必须的

    // 内核链表与数据结构无关，所有链表操作通过操作 list\_head 来完成

    struct list\_head list;

};

void print\_student(struct student \*);

static int testlist\_init(void)

{

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

struct student \*stu;

    // 生成链表头

LIST\_HEAD(stu\_head);

    // 生成 stu1 节点

    // kmalloc用于分配内存

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

    stu1->id = 1;

    stu1->name = "wyb";

    // 指定节点的 list\_head

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

    // 生成 stu2 节点

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

    stu2->id = 2;

    stu2->name = "wyb2";

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

    // 生成 stu3 节点

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

    stu3->id = 3;

    stu3->name = "wyb3";

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

    // 生成 stu4 节点

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

    stu4->id = 4;

    stu4->name = "wyb4";

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

    // 追加节点到链表头中

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

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

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

list\_add(&stu4->list, &stu\_head);

    // 循环链表

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

    {

        print\_student(stu);

}

    // 反向循环链表

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

    {

        print\_student(stu);

}

    // 从stu2所在的链表中移除stu2

list\_del(&stu2->list);

    // 将 stu3 替换成 stu2

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

    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 += testmodule.o

# 当前路径

CURRENT\_PATH:=$(shell pwd)

# 当前内核版本

LINUX\_KERNEL:=$(shell uname -r)

# 当前内核路径（注：不是内核源码路径）

LINUX\_KERNEL\_PATH:=/lib/modules/$(LINUX\_KERNEL)/build

# 编译

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：

安装内核模块：insmod testmodule.ko

移除内核模块：rmmod testmodule

查看内核模块输出：dmesg | tail -100

### **2. 队列**

内核中的队列是以字节形式保存数据的，所以获取数据的时候，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过大。

#### **2.1 头文件简介**

内核中关于队列定义的头文件位于：<linux/kfifo.h> include/linux/kfifo.h

头文件中定义的函数的实现位于：kernel/kfifo.c

#### **2.2 队列代码的注意点**

内核队列编程需要注意的是：

* 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**

构造了一个内核模块来实际使用一下内核中的队列，代码在CentOS6.3 x64上运行通过。

C代码:

#include <linux/init.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/kfifo.h>

#include <linux/time.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 \*myfifo;

    struct kfifo f;

myfifo = &f;

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

    struct student \*stu\_tmp;

    char \*c\_tmp;

int i;

    // 初始化内核队列

    int result = kfifo\_alloc(myfifo, (4 \* sizeof(struct student)), GFP\_KERNEL);

    if (result)

    {

        return result;

}

    // 生成 stu1

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

    stu1->id = 1;

    stu1->name = "wyb1";

    // stu1 进入队列

kfifo\_in(myfifo, (void \*)stu1, sizeof(struct student));

    // 生存stu2

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

    stu2->id = 1;

    stu2->name = "wyb2";

kfifo\_in(myfifo, (void \*)stu2, sizeof(struct student));

    // 生存stu3

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

    stu3->id = 1;

    stu3->name = "wyb3";

kfifo\_in(myfifo, (void \*)stu3, sizeof(struct student));

    // 生存stu4

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

    stu4->id = 1;

    stu4->name = "wyb4";

kfifo\_in(myfifo, (void \*)stu4, sizeof(struct student));

    // 分配内存存放临时队列成员

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

    // 获取队列的长度

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

    // 遍历队列

    for (i = 0; i < 4; i++)

    {

        // 从fifo队列中取出一个

        kfifo\_out(myfifo, c\_tmp, sizeof(struct student));

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

        print\_student(stu\_tmp);

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

}

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

    // 释放队列

    kfifo\_free(myfifo);

    // 释放用于保存临时数据的内存

    kfree(c\_tmp);

    return 0;

}

static 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");

}

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);

Makefile：

同上

### **3. 映射**

映射的有点想其他语言(C#或者python)中的字典类型，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**

内核中关于映射定义的头文件位于：<linux/idr.h> include/linux/idr.h

头文件中定义的函数的实现位于：lib/idr.c

映射使用一个树状结构，每个节点都有一个标识符uid和用户数据组成，根据uid我们能找到对应的用户数据

#### **3.2 映射代码的注意点**

映射的使用需要注意的是，给自定义的数据结构申请一个id的时候，不能直接申请id，先要分配id(函数idr\_pre\_get)，分配成功后，在获取一个id(函数idr\_get\_new)。

idr的结构比较复杂，我也没有很好的理解，但是csdn上有篇介绍linux idr结构的博客写的挺好，图文并茂：<http://blog.csdn.net/paomadi/article/details/8539794>

#### **3.3 使用示例**

idr代码已有变化，如下是旧版本代码的示例。

C代码（使用的头文件、函数与队列一样）:

#include <linux/init.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/idr.h>

#include <linux/time.h>

MODULE\_LICENSE("Dual BSD/GPL");

struct student

{

    int id;

    char\* name;

};

static int print\_student(int, void\*, void\*);

static int testidr\_init(void)

{

    DEFINE\_IDR(idp);

    struct student \*stu[4];

    //    struct student \*stu\_tmp;

int id, ret, i;

    // init 4 struct student

for (i=0; i<4; i++) {

        stu[i] = kmalloc(sizeof(struct student), GFP\_KERNEL);

        stu[i]->id = i;

        stu[i]->name = "wyb";

}

    // add 4 student to idr

for (i=0; i < 4; i++) {

        do {

            if (!idr\_pre\_get(&idp, GFP\_KERNEL))

                return -ENOSPC;

            ret = idr\_get\_new(&idp, stu[i], &id);

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

        } while(ret == -EAGAIN);

}

    // display all student in idr

idr\_for\_each(&idp, print\_student, NULL);

    idr\_destroy(&idp);

    kfree(stu[0]);

    kfree(stu[1]);

    kfree(stu[2]);

    kfree(stu[3]);

    return 0;

}

static int print\_student(int id, void \*p, void \*data)

{

    struct student\* stu = p;

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

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

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

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

    return 0;

}

static void testidr\_exit(void)

{

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

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

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

}

module\_init(testidr\_init);

module\_exit(testidr\_exit);

### **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。

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，那么它的父节点和子节点都是黑色的
* 从任何一个节点开始，到其下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的2倍。

#### **4.1 头文件简介**

内核中关于红黑树定义的头文件位于：<linux/rbtree.h> include/linux/rbtree.h

头文件中定义的函数的实现位于：lib/rbtree.c

#### **4.2 红黑树代码的注意点**

内核中红黑树的使用和链表(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，我觉得很奇怪，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：

父节点的地址

本节点的颜色

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的倍数分配，所以申请分配的 struct rb\_node 的32位地址的最后2位始终是零，rb\_parent\_color 就是利用最后一位来保存节点的颜色信息的

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

**4.3 红黑树初始化**

struct rb\_root rbroot = RB\_ROOT;

rbtree的实现并没有提供搜索和插入例程，这些例程希望由rbtree的用户自己定义。这是因 为C语言不大容易进行泛型编程，同时Linux内核开发者们相信最有效的捜索和插入方法需要每 个用户自己去实现。