# TODO

[linux 内核定时器 timer\_list详解](https://www.cnblogs.com/leaven/archive/2010/08/19/1803382.html)

<https://www.cnblogs.com/leaven/archive/2010/08/19/1803382.html>

中断的学习

<https://blog.csdn.net/xiezhi123456/article/details/80414804>

<http://abcdxyzk.github.io/blog/2017/07/23/kernel-clock-6/>

# timer

## 数据结构

timer\_list

内核定时器的数据结构

struct timer\_list {

struct list\_head entry;

unsigned long expires;

void (\*function)(unsigned long);

unsigned long data;

struct tvec\_base \*base;

/\* ... \*/

};

其中 expires 字段表示期望定时器执行的 jiffies 值，到达该 jiffies 值时，将调用 function 函数，并传递 data 作为参数。当一个定时器被注册到内核之后，entry 字段用来连接该定时器到一个内核链表中。base 字段是内核内部实现所用的。

需要注意的是 expires 的值是32位的，因为内核定时器并不适用于长的未来时间点。

## 初始化

在使用 struct timer\_list 之前，需要初始化该数据结构，确保所有的字段都被正确地设置。初始化有两种方法。

### 方法一：DEFINE\_TIMER

DEFINE\_TIMER(timer\_name, function\_name, expires\_value, data);

该宏会静态创建一个名叫 timer\_name 内核定时器，并初始化其 function, expires, name 和 base 字段。

### 方法二：setup\_timer(

struct timer\_list mytimer;

setup\_timer(&mytimer, (\*function)(unsigned long), unsigned long data);

mytimer.expires = jiffies + 5\*HZ;

### 方法3：init\_timer

struct timer\_list mytimer;

init\_timer(&mytimer);

mytimer ->timer.expires = jiffies + 5\*HZ;

mytimer ->timer.data = (unsigned long) dev;

mytimer ->timer.function = &corkscrew\_timer; /\* timer handler \*/

通过init\_timer()动态地定义一个定时器，此后，将处理函数的地址和参数绑定给一个timer\_list，

注意，无论用哪种方法初始化，其本质都只是给字段赋值，所以只要在运行 add\_timer() 之前，expires, function 和 data 字段都可以直接再修改。

关于上面这些宏和函数的定义，参见 include/linux/timer.h。

## 注册（注销）add\_timer(

定时器要生效，还必须被连接到内核专门的链表中，这可以通过 add\_timer(struct timer\_list \*timer) 来实现。

重新注册

要修改一个定时器的调度时间，可以通过调用 mod\_timer(struct timer\_list \*timer, unsigned long expires)。mod\_timer() 会重新注册定时器到内核，而不管定时器函数是否被运行过。

注销

注销一个定时器，可以通过 del\_timer(struct timer\_list \*timer) 或 del\_timer\_sync(struct timer\_list \*timer)。其中 del\_timer\_sync 是用在 SMP 系统上的（在非SMP系统上，它等于del\_timer），当要被注销的定时器函数正在另一个 cpu 上运行时，del\_timer\_sync() 会等待其运行完，所以这个函数会休眠。另外还应避免它和被调度的函数争用同一个锁。对于一个已经被运行过且没有重新注册自己的定时器而言，注销函数其实也没什么事可做。

int timer\_pending(const struct timer\_list \*timer)

这个函数用来判断一个定时器是否被添加到了内核链表中以等待被调度运行。注意，当一个定时器函数即将要被运行前，内核会把相应的定时器从内核链表中删除（相当于注销）

## 使用说明

LINUX内核定时器是内核用来控制在未来某个时间点（基于jiffies）调度执行某个函数的一种机制，其实现位于 <linux/timer.h> 和 kernel/timer.c 文件中。被调度的函数肯定是异步执行的，它类似于一种“软件中断”，而且是处于非进程的上下文中，所以调度函数必须遵守以下规则：

1) 没有 current 指针、不允许访问用户空间。因为没有进程上下文，相关代码和被中断的进程没有任何联系。

2) 不能执行休眠（或可能引起休眠的函数）和调度。

3) 任何被访问的数据结构都应该针对并发访问进行保护，以防止竞争条件。

内核定时器的调度函数运行过一次后就不会再被运行了（相当于自动注销），但可以通过在被调度的函数中重新调度自己来周期运行。

在SMP系统中，调度函数总是在注册它的同一CPU上运行，以尽可能获得缓存的局域性

## 列子

### 1

#include <linux/module.h>

#include <linux/timer.h>

#include <linux/jiffies.h>

struct timer\_list mytimer;

static void myfunc(unsigned long data)

{

printk("%s/n", (char \*)data);

mod\_timer(&mytimer, jiffies + 2\*HZ);

}

static int \_\_init mytimer\_init(void)

{

setup\_timer(&mytimer, myfunc, (unsigned long)"Hello, world!");

mytimer.expires = jiffies + HZ;

add\_timer(&mytimer);

return 0;

}

static void \_\_exit mytimer\_exit(void)

{

del\_timer(&mytimer);

}

module\_init(mytimer\_init);

module\_exit(mytimer\_exit);

### 2

例子2

static struct timer\_list power\_button\_poll\_timer;

static void power\_button\_poll(unsigned long dummy)

{

if (gpio\_line\_get(N2100\_POWER\_BUTTON) == 0) {

ctrl\_alt\_del();

return;

}

power\_button\_poll\_timer.expires = jiffies + (HZ / 10);

add\_timer(&power\_button\_poll\_timer);

}

static void \_\_init n2100\_init\_machine(void)

{

；

；

init\_timer(&power\_button\_poll\_timer);

power\_button\_poll\_timer.function = power\_button\_poll;

power\_button\_poll\_timer.expires = jiffies + (HZ / 10);

add\_timer(&power\_button\_poll\_timer);

}

### 例子3

设备open时初始化和注册定时器

static int corkscrew\_open(struct net\_device \*dev)

{

；

；

init\_timer(&vp->timer);

vp->timer.expires = jiffies + media\_tbl[dev->if\_port].wait;

vp->timer.data = (unsigned long) dev;

vp->timer.function = &corkscrew\_timer; /\* timer handler \*/

add\_timer(&vp->timer);

：

；

}

定时器超时处理函数，对定时器的超时时间重新赋值

static void corkscrew\_timer(unsigned long data)

{

；

；

vp->timer.expires = jiffies + media\_tbl[dev->if\_port].wait;

add\_timer(&vp->timer);

；

；

}

设备close时删除定时器

static int corkscrew\_close(struct net\_device \*dev)

{

；

；

del\_timer(&vp->timer);

；

；

}

### 4

本例子用DEFINE\_TIMER静态创建定时器

#include <linux/module.h>

#include <linux/jiffies.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/timer.h>

#include <linux/leds.h>

static void ledtrig\_ide\_timerfunc(unsigned long data);

DEFINE\_LED\_TRIGGER(ledtrig\_ide);

static DEFINE\_TIMER(ledtrig\_ide\_timer, ledtrig\_ide\_timerfunc, 0, 0);

static int ide\_activity;

static int ide\_lastactivity;

void ledtrig\_ide\_activity(void)

{

ide\_activity++;

if (!timer\_pending(&ledtrig\_ide\_timer))

mod\_timer(&ledtrig\_ide\_timer, jiffies + msecs\_to\_jiffies(10));

}

EXPORT\_SYMBOL(ledtrig\_ide\_activity);

static void ledtrig\_ide\_timerfunc(unsigned long data)

{

if (ide\_lastactivity != ide\_activity) {

ide\_lastactivity = ide\_activity;

led\_trigger\_event(ledtrig\_ide, LED\_FULL);

mod\_timer(&ledtrig\_ide\_timer, jiffies + msecs\_to\_jiffies(10));

} else {

led\_trigger\_event(ledtrig\_ide, LED\_OFF);

}

}

static int \_\_init ledtrig\_ide\_init(void)

{

led\_trigger\_register\_simple("ide-disk", &ledtrig\_ide);

return 0;

}

static void \_\_exit ledtrig\_ide\_exit(void)

{

led\_trigger\_unregister\_simple(ledtrig\_ide);

}

module\_init(ledtrig\_ide\_init);

module\_exit(ledtrig\_ide\_exit);

==================================================================================

add\_timer() -- 将定时器添加到定时器等待队列中

2007年08月04日 星期六 15:30

用add\_timer()函数来看timer\_base的作用

static inline void add\_timer(struct timer\_list \*timer)

{

BUG\_ON(timer\_pending(timer));

\_\_mod\_timer(timer, timer->expires);

}

int \_\_mod\_timer(struct timer\_list \*timer, unsigned long expires)

{

tvec\_base\_t \*base, \*new\_base;

unsigned long flags;

int ret = 0;

timer\_stats\_timer\_set\_start\_info(timer);

BUG\_ON(!timer->function);

base = lock\_timer\_base(timer, &flags);

如果timer已经放到定时链表中,则释放开

|--------------------------------|

| if (timer\_pending(timer)) { -|

| detach\_timer(timer, 0); -|

| ret = 1; |

| } |

|--------------------------------|

获取当前CPU的timer base

|-----------------------------------------|

| new\_base = \_\_get\_cpu\_var(tvec\_bases); |

|-----------------------------------------|

如果当前CPU的timer base不是当前timer中的base, 更新timer的base

|----------------------------------------------------|

| if (base != new\_base) { |

| if (likely(base->running\_timer != timer)) { -|

| timer->base = NULL; |

| spin\_unlock(&base->lock); |

| base = new\_base; |

| spin\_lock(&base->lock); |

| timer->base = base; |

| } |

| } |

|----------------------------------------------------|

给定时器timer设置超时时间；并添加该时钟

|-------------------------------------|

| timer->expires = expires; |

| internal\_add\_timer(base, timer); -|

|-------------------------------------|

spin\_unlock\_irqrestore(&base->lock, flags);

return ret;

}

MODULE\_AUTHOR("Richard Purdie <rpurdie@openedhand.com>");

MODULE\_DESCRIPTION("LED IDE Disk Activity Trigger");

MODULE\_LICENSE("GPL");

## 小结

总的来说，timer的用法还是很简单的。主要需要定义一个timer\_list变量timer、先初始化timer

　　init\_timer(&timer);

　　then 对timer的相关参数赋值：

　　timer.function = fun;

　　timer.expires = jiffies + TIMER\_DELAY;

　　add\_timer(&timer);

　　在定时器时间到的时候，会执行fun，如果继续定时，可以通过

　　在fun中执行

　　mod\_timer(&timer, jiffies + TIMER\_DELAY);

　　在不需要的时候通过调用

　　del\_timer(&timer);

　　删除定时器。

　　简单吧。这样一个简单的定时器就完成了。

　　呵呵。

　　附程序：

　　#include <linux/module.h>

　　#include <linux/types.h>

　　#include <linux/fs.h>

　　#include <linux/errno.h>

　　#include <linux/mm.h>

　　#include <linux/sched.h>

　　#include <linux/init.h>

　　#include <linux/cdev.h>

　　#include <asm/io.h>

　　#include <asm/system.h>

　　#include <asm/uaccess.h>

　　#include <linux/timer.h>

　　#include <asm/atomic.h>

　　#define SECOND\_MAJOR 0

　　static int second\_major = SECOND\_MAJOR;

　　struct second\_dev

　　{

　　struct cdev cdev;

　　atomic\_t counter;

　　struct timer\_list s\_timer;

　　};

　　struct second\_dev \*second\_devp;

　　static void second\_timer\_handle(unsigned long arg)

　　{

　　mod\_timer(&second\_devp->s\_timer, jiffies + HZ);

　　atomic\_inc(&second\_devp->counter);

　　printk(KERN\_ERR "current jiffies is %ld/n",jiffies);

　　}

　　int second\_open(struct inode \*inode, struct file \*filp)

　　{

　　init\_timer(&second\_devp->s\_timer);

　　second\_devp->s\_timer.function = &second\_timer\_handle;

　　second\_devp->s\_timer.expires = jiffies + HZ;

　　add\_timer(&second\_devp->s\_timer);

　　atomic\_set(&second\_devp->counter, 0);

　　return 0;

　　}

　　int second\_release(struct inode \*inode, struct file \*filp)

　　{

　　del\_timer(&second\_devp->s\_timer);

　　return 0;

　　}

　　static ssize\_t second\_read(struct file \*filp, char \_\_user \*buf, size\_t count,

　　loff\_t \*ppos)

　　{

　　int counter;

　　counter = atomic\_read(&second\_devp->counter);

　　if (put\_user(counter, (int \*)buf))

　　{

　　return -EFAULT;

　　}else

　　{

　　return sizeof(unsigned int);

　　}

　　}

　　static const struct file\_operations second\_fops =

　　{

　　.owner = THIS\_MODULE,

　　.open = second\_open,

　　.release = second\_release,

　　.read = second\_read,

　　};

　　static void second\_setup\_cdev(struct second\_dev \*dev, int index)

　　{

　　int err, devno = MKDEV(second\_major, index);

　　cdev\_init(&dev->cdev, &second\_fops);

　　dev->cdev.owner = THIS\_MODULE;

　　dev->cdev.ops = &second\_fops;

　　err = cdev\_add(&dev->cdev, devno, 1);

　　if (err)

　　{

　　printk(KERN\_NOTICE "Error %d add second%d", err, index);

　　}

　　}

　　int second\_init(void)

　　{

　　int ret;

　　dev\_t devno = MKDEV(second\_major, 0);

　　if (second\_major)

　　{

　　ret = register\_chrdev\_region(devno, 1, "second");

　　}else

　　{

　　ret = alloc\_chrdev\_region(&devno, 0, 1, "second");

　　second\_major = MAJOR(devno);

　　}

　　if (ret < 0)

　　{

　　return ret;

　　}

　　second\_devp = kmalloc(sizeof(struct second\_dev), GFP\_KERNEL);

　　if (!second\_devp)

　　{

　　ret = -ENOMEM;

　　goto fail\_malloc;

　　}

　　memset(second\_devp, 0, sizeof(struct second\_dev));

　　second\_setup\_cdev(second\_devp, 0);

　　return 0;

　　fail\_malloc:

　　unregister\_chrdev\_region(devno, 1);

　　}

　　void second\_exit(void)

　　{

　　cdev\_del(&second\_devp->cdev);

　　kfree(second\_devp);

　　unregister\_chrdev\_region(MKDEV(second\_major, 0), 1);

　　}

　　MODULE\_AUTHOR("Song Baohua");

　　MODULE\_LICENSE("Dual BSD/GPL");

　　module\_param(second\_major, int, S\_IRUGO);

　　module\_init(second\_init);

　　module\_exit(second\_exit);

　　附上用户端的测试程序：

　　#include <stdio.h>

　　#include <unistd.h>

　　#include <fcntl.h>

　　int main(void)

　　{

　　int fd, i;

　　int data;

　　fd = open("/dev/second",O\_RDONLY);

　　if (fd < 0)

　　{

　　printf("open /dev/second error/n");

　　}

　　for(i = 0; i < 20; i++)

　　{

　　read(fd, &data, sizeof(data));

　　printf("read /dev/second is %d/n",data);

　　sleep(1);

　　}

　　close(fd);

　　}

[复制代码](javascript:void(0);)

## hrtimer

<http://abcdxyzk.github.io/blog/2017/07/23/kernel-clock-6/>

## 参考

https://www.cnblogs.com/sky-heaven/p/5157653.html

init\_timer

# workqueue

kernel/workqueue.c

## 是什么

workqueue是对内核线程封装的用于处理各种工作项的一种处理方法， 由于处理对象是用链表拼接一个个工作项， 依次取出来处理， 然后从链表删除，就像一个队列排好队依次处理一样， 所以也称工作队列，

所谓封装可以简单理解一个中转站， 一边指向“合适”的内核线程， 一边接受你丢过来的工作项， 用结构体 workqueue\_srtuct表示， 而所谓工作项也是个结构体 --  work\_struct， 里面有个成员指针， 指向你最终要实现的函数，

### workqueue\_struct

**struct** workqueue\_struct {  
 **struct** list\_head pwqs; */\* WR: all pwqs of this wq \*/* **struct** list\_head list; */\* PR: list of all workqueues \*/* **struct** mutex mutex; */\* protects this wq \*/* **int** work\_color; */\* WQ: current work color \*/* **int** flush\_color; */\* WQ: current flush color \*/* atomic\_t nr\_pwqs\_to\_flush; */\* flush in progress \*/* **struct** wq\_flusher \*first\_flusher; */\* WQ: first flusher \*/* **struct** list\_head flusher\_queue; */\* WQ: flush waiters \*/* **struct** list\_head flusher\_overflow; */\* WQ: flush overflow list \*/* **struct** list\_head maydays; */\* MD: pwqs requesting rescue \*/* **struct** worker \*rescuer; */\* I: rescue worker \*/* **int** nr\_drainers; */\* WQ: drain in progress \*/* **int** saved\_max\_active; */\* WQ: saved pwq max\_active \*/* **struct** workqueue\_attrs \*unbound\_attrs; */\* PW: only for unbound wqs \*/* **struct** pool\_workqueue \*dfl\_pwq; */\* PW: only for unbound wqs \*/*#ifdef CONFIG\_SYSFS  
 **struct** wq\_device \*wq\_dev; */\* I: for sysfs interface \*/*#endif  
#ifdef CONFIG\_LOCKDEP  
 **struct** lockdep\_map lockdep\_map;  
#endif  
 **char** name[WQ\_NAME\_LEN]; */\* I: workqueue name \*/  
  
 /\*  
 \* Destruction of workqueue\_struct is sched-RCU protected to allow  
 \* walking the workqueues list without grabbing wq\_pool\_mutex.  
 \* This is used to dump all workqueues from sysrq.  
 \*/* **struct** rcu\_head rcu;  
  
 */\* hot fields used during command issue, aligned to cacheline \*/* **unsigned int** flags \_\_\_\_cacheline\_aligned; */\* WQ: WQ\_\* flags \*/* **struct** pool\_workqueue \_\_percpu \*cpu\_pwqs; */\* I: per-cpu pwqs \*/* **struct** pool\_workqueue \_\_rcu \*numa\_pwq\_tbl[]; */\* PWR: unbound pwqs indexed by node \*/*};

### work\_struct

**struct** work\_struct {  
 atomic\_long\_t data;// //函数的参数  
 **struct** list\_head entry;// /挂到链表  
 work\_func\_t func;// //函数指针，指向你实现的函数功能  
#ifdef CONFIG\_LOCKDEP  
 **struct** lockdep\_map lockdep\_map;  
#endif  
};

当然使用者在实现自己函数功能后可以直接调用，或者通过kthread\_create()把函数当做新线程的主代码， 或者add\_timer添加到一个定时器延时处理，

那为何要弄个work\_struct工作项先封装函数， 然后再丢到workqueue\_srtuct处理呢？ 这就看使用场景了， 如果是一个大函数， 处理事项比较多， 且需要重复处理， 可以单独开辟一个内核线程处理； 对延时敏感的可以用定时器；

如果只是简单的一个函数功能，  且函数里面有延时动作的， 就适合放到工作队列来处理了， 毕竟定时器处理的函数是在中断上下文，不能delay或者引发进程切换的API，  而且开辟一个内核线程是耗时且耗费资源的， 一般用于函数需要while(1) 不断循环处理的，

不然处理一次函数后退出，线程又被销毁， 简直就是浪费！

## 怎么用

一个简单示例：

[复制代码](javascript:void(0);)

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/delay.h>

#include <linux/workqueue.h>

struct workqueue\_struct \*workqueue\_test;

struct work\_struct work\_test;

void work\_test\_func(struct work\_struct \*work)

{

printk("%s()\n", \_\_func\_\_);

//mdelay(1000);

//queue\_work(workqueue\_test, &work\_test);

}

static int test\_init(void)

{

printk("Hello,world!\n");

/\* 1. 自己创建一个workqueue， 中间参数为0，默认配置 \*/

workqueue\_test = alloc\_workqueue("workqueue\_test", 0, 0);

/\* 2. 初始化一个工作项，并添加自己实现的函数 \*/

INIT\_WORK(&work\_test, work\_test\_func);

/\* 3. 将自己的工作项添加到指定的工作队列去， 同时唤醒相应线程处理 \*/

queue\_work(workqueue\_test, &work\_test);

return 0;

}

static void test\_exit(void)

{

printk("Goodbye,cruel world!\n");

destroy\_workqueue(workqueue\_test);

}

module\_init(test\_init);

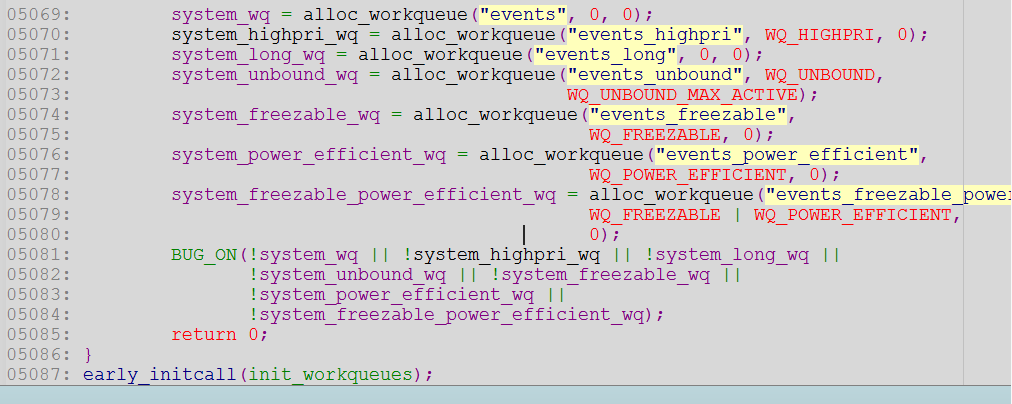
module\_exit(test\_exit);

MODULE\_AUTHOR("Vedic <FZKmxcz@163.com>");

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

### work\_queue

 当然内核已经为我们创建了几个工作队列， 我们可以直接将自己的工作项挂到相应的队列即可



所以代码可以改为：

[复制代码](javascript:void(0);)

static int test\_init(void)

{

printk("Hello,world!\n");

/\* 2. 初始化一个工作项，并添加自己实现的函数 \*/

INIT\_WORK(&work\_test, work\_test\_func);

/\* 3. 将自己的工作项添加到指定的工作队列去， 同时唤醒相应线程处理 \*/

queue\_work(system\_wq, &work\_test);

return 0;

}  
  
如果workqueue对象是 system\_wq， 可以使用另一个封装函数schedule\_work(&work\_test)

static inline bool schedule\_work(struct work\_struct \*work)  
{  
　　return queue\_work(system\_wq, work);  
}

将自己的工作项挂到已有的工作队列需要注意的是由于这些队列是共享的， 各个驱动都有可能将自己的工作项放到同个队列， 会导致队列的项拥挤， 当有些项写的代码耗时久或者调用delay()延时特别久， 你的项将会迟迟得不到执行！

所以早期很多驱动开发人员都是自己创建workqueue， 添加自己的work

创建workqueue时会创建属于workqueue自己的内核线程， 这些线程是“私有的”， 虽然是方便了驱动开发人员， 但每个驱动都“一言不合”就

创建workqueue导致太多线程， 严重占用系统资源和效率， 所以在Linux-3.XXX时代， 社区开发人员将workqueue和内核线程剥离！ 内核会自己事先创建相应数量的线程（后面详解）， 被所有驱动共享使用。  用户调用alloc\_workqueue()

只是创建workqueue这个空壳， 其主要作用：

　　a. 兼容Linux-2.XXX时代代码

　　b. 新增flag字段表明这个workqueue的属性（普通优先级还是高优先级等）， 方便在queue\_work()时寻找“合适的”线程， 因为事先创建的线程分普通优先级、高优先级、绑定CPU线程， 非绑定CPU线程等

当然这对驱动开发人员是透明的， 驱动人员只需关注调用queue\_work()让线程执行自己的工作项， 至于是这个workqueue的私有线程还是现在的共享线程， 不重要！ 这样就限制了系统工作线程的暴涨， 唯一的缺点就是前面提到的， 跟别人共享会增加

自己的工作项被执行的不确定性。 只能说各个驱动开发人员自我约束， 尽量使得工作项函数简短快速， 如果我们需要等自己的工作项被执行完才能处理其他事情， 可以调用flush\_work() 等待work被执行完：

### flush\_work

gdh

/\*\*

\* flush\_work - wait for a work to finish executing the last queueing instance

\* @work: the work to flush

\*

\* Wait until @work has finished execution. @work is guaranteed to be idle

\* on return if it hasn't been requeued since flush started.

\*

\* RETURNS:

\* %true if flush\_work() waited for the work to finish execution,

\* %false if it was already idle.

\*/

bool flush\_work(struct work\_struct \*work)

{

struct wq\_barrier barr;

lock\_map\_acquire(&work->lockdep\_map);

lock\_map\_release(&work->lockdep\_map);

if (start\_flush\_work(work, &barr)) {

wait\_for\_completion(&barr.done);

destroy\_work\_on\_stack(&barr.work);

return true;

} else {

return false;

}

}

EXPORT\_SYMBOL\_GPL(flush\_work);

[复制代码](javascript:void(0);)

\

## 部分源码解析

### init\_workqueues

**static int** \_\_init init\_workqueues(**void**)  
{  
 **int** std\_nice[NR\_STD\_WORKER\_POOLS] = { 0, HIGHPRI\_NICE\_LEVEL };//

**//NR\_STD\_WORKER\_POOLS = 2**

//线程两种优先级： nice=0普通级； nice=-20高优先级

**int** i, cpu;  
  
 WARN\_ON(**\_\_alignof\_\_**(**struct** pool\_workqueue) < **\_\_alignof\_\_**(**long long**));  
  
 BUG\_ON(!alloc\_cpumask\_var(&wq\_unbound\_cpumask, GFP\_KERNEL));  
 cpumask\_copy(wq\_unbound\_cpumask, cpu\_possible\_mask);  
  
 pwq\_cache = KMEM\_CACHE(pool\_workqueue, SLAB\_PANIC);  
  
 cpu\_notifier(workqueue\_cpu\_up\_callback, CPU\_PRI\_WORKQUEUE\_UP);  
 hotcpu\_notifier(workqueue\_cpu\_down\_callback, CPU\_PRI\_WORKQUEUE\_DOWN);  
  
 wq\_numa\_init();  
  
 */\* initialize CPU pools \*/* for\_each\_possible\_cpu(cpu) {  
 **struct** worker\_pool \*pool;  
  
 i = 0;  
 for\_each\_cpu\_worker\_pool(pool, cpu) {  
 BUG\_ON(init\_worker\_pool(pool));  
 pool->cpu = cpu;  
 cpumask\_copy(pool->attrs->cpumask, cpumask\_of(cpu));  
 pool->attrs->nice = std\_nice[i++];  
 pool->node = cpu\_to\_node(cpu);  
  
 */\* alloc pool ID \*/* mutex\_lock(&wq\_pool\_mutex);  
 BUG\_ON(worker\_pool\_assign\_id(pool));  
 mutex\_unlock(&wq\_pool\_mutex);  
 }  
 }  
  
 */\* create the initial worker \*/* for\_each\_online\_cpu(cpu) {  
 **struct** worker\_pool \*pool;  
  
 for\_each\_cpu\_worker\_pool(pool, cpu) {  
 pool->flags &= ~POOL\_DISASSOCIATED;  
 BUG\_ON(!create\_worker(pool));  
 }  
 }  
  
 */\* create default unbound and ordered wq attrs \*/* **for** (i = 0; i < NR\_STD\_WORKER\_POOLS; i++) {  
 **struct** workqueue\_attrs \*attrs;  
  
 BUG\_ON(!(attrs = alloc\_workqueue\_attrs(GFP\_KERNEL)));  
 attrs->nice = std\_nice[i];  
 unbound\_std\_wq\_attrs[i] = attrs;  
  
 */\*  
 \* An ordered wq should have only one pwq as ordering is  
 \* guaranteed by max\_active which is enforced by pwqs.  
 \* Turn off NUMA so that dfl\_pwq is used for all nodes.  
 \*/* BUG\_ON(!(attrs = alloc\_workqueue\_attrs(GFP\_KERNEL)));  
 attrs->nice = std\_nice[i];  
 attrs->no\_numa = true;  
 ordered\_wq\_attrs[i] = attrs;  
 }  
  
 system\_wq = alloc\_workqueue(**"events"**, 0, 0);  
 system\_highpri\_wq = alloc\_workqueue(**"events\_highpri"**, WQ\_HIGHPRI, 0);  
 system\_long\_wq = alloc\_workqueue(**"events\_long"**, 0, 0);  
 system\_unbound\_wq = alloc\_workqueue(**"events\_unbound"**, WQ\_UNBOUND,  
 WQ\_UNBOUND\_MAX\_ACTIVE);  
 system\_freezable\_wq = alloc\_workqueue(**"events\_freezable"**,  
 WQ\_FREEZABLE, 0);  
 system\_power\_efficient\_wq = alloc\_workqueue(**"events\_power\_efficient"**,  
 WQ\_POWER\_EFFICIENT, 0);  
 system\_freezable\_power\_efficient\_wq = alloc\_workqueue(**"events\_freezable\_power\_efficient"**,  
 WQ\_FREEZABLE | WQ\_POWER\_EFFICIENT,  
 0);  
 BUG\_ON(!system\_wq || !system\_highpri\_wq || !system\_long\_wq ||  
 !system\_unbound\_wq || !system\_freezable\_wq ||  
 !system\_power\_efficient\_wq ||  
 !system\_freezable\_power\_efficient\_wq);  
 **return** 0;  
}  
early\_initcall(init\_workqueues);

### INIT\_WORK

INIT\_WORK会在你定义的\_work工作队列里面增加一个工作任务，该任务就是\_func。\_func这个任务会需要一些数据作为参数，这个参数就是通过\_data传递的。任务的启动是通过调度程序schedule\_work来处理的。INIT\_LIST\_HEAD是初始化一个链表，就是在此之前，链表是不存在的。  
(\_work)->pending是调度程序需要使用的一个标志，没有仔细研究用处。  
PREPARE\_WORK就是进行赋值，确定链表中这个对象的任务和数据。  
init\_timer是初始化这个对象的任务时间，具体作用没有研究。

#define INIT\_WORK(\_work, \_func) \  
 \_\_INIT\_WORK((\_work), (\_func), 0)

#define \_\_INIT\_WORK(\_work, \_func, \_onstack) \  
 **do** { \  
 \_\_init\_work((\_work), \_onstack); \  
 (\_work)->data = (atomic\_long\_t) WORK\_DATA\_INIT(); \  
 INIT\_LIST\_HEAD(&(\_work)->entry); \  
 (\_work)->func = (\_func); \  
 } **while** (0)  
#endif

**void** \_\_init\_work(**struct** work\_struct \*work, **int** onstack)  
{  
 **if** (onstack)  
 debug\_object\_init\_on\_stack(work, &work\_debug\_descr);  
 **else** debug\_object\_init(work, &work\_debug\_descr);  
}

#### for\_each\_cpu\_worker\_pool

其相关代码在：

[复制代码](javascript:void(0);)

#define for\_each\_cpu\_worker\_pool(pool, cpu) \

for ((pool) = &per\_cpu(cpu\_worker\_pools, cpu)[0]; \

(pool) < &per\_cpu(cpu\_worker\_pools, cpu)[NR\_STD\_WORKER\_POOLS]; \

(pool)++)

static DEFINE\_PER\_CPU\_SHARED\_ALIGNED(struct worker\_pool [NR\_STD\_WORKER\_POOLS], cpu\_worker\_pools);

[复制代码](javascript:void(0);)

　　从这里可以看出， 每个CPU都有两个私有结构体 struct worker\_pool ， 用变量cpu\_worker\_pools 表示， 而这两个worker\_pool最大区别就是nice赋值， 以及对worker\_pool 编号

https://www.cnblogs.com/vedic/p/11069249.html

对online CPU 每个worker\_pool创建worker， 也即前面讲到的工作线程

#### system\_wq

# irq.drivers

devm\_request\_threaded\_irq

<https://blog.csdn.net/xiezhi123456/article/details/80414804>