一、 上下文的概念

人为地划分两个部分：上文、 下文；

例如中断服务函数： 紧急操作放在中断上文；

可以推后的工作放在下文执行，目的是让 CPU 尽快的出中断。

例：中断当中： 获取键值、 修改定时器、……

中断上文： 紧急的操作(确定键值)；

中断下文： 修改定时器、……(注册一个小任务、工作队列)；

内核当中： 提供了俩种可以推后执行的机制： tasklet(小任务)；

work(工作队列)；

二、 tasklet 小任务

使用 tasklet_struct 描述小任务：

```
struct tasklet_struct
{
    struct tasklet_struct *next;        //下一个小任务；
    unsigned long state;                //状态；
    atomic_t count;                     //引用计数器      0：可执行状态； 1：不可执行状态；
    void (*func)(unsigned long);        //指定小任务执行的函数；
    unsigned long data;                 //传递给任务的参数；
};
```

填充 tasklet_struct 结构体，在合适的时候调度小任务；

三、 相关的 API

1、 创建小任务

1） 静态创建

a、DECLARE_TASKLET(name,func,data)； //引用计数器为 0，可运行的状态；

```
#define DECLARE_TASKLET(name,func,data)
                struct tasklet_struct name ={NULL,0,ATOMIC_INIT(0),func,data}
```

b、DECLARE_TASKLET_DISABLED(name,func, data) //引用计数器为 1，不可运行状态；

```
#define DECLARE_TASKLET_DISABLED(name, func, data)
                struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data}
```

2）动态创建

```
void tasklet_init(struct tasklet_struct *tasklet,void (*func)(unsigned long),unsigned long data);
                                                                    //可运行状态；
```

2、 调度小任务

```
tasklet_schedule(struct tasklet_struct *tasklet);
```

3、 终止小任务

```
tasklet_kill(struct tasklet_struct *t)；
```

四、 工作队列

使用 struct work_struct 来描述一个工作队列。

```
struct work_struct
{
        atomic_long_t data;             //内核填充
        struct list_head entry;
        work_func_t func;               //工作队列执行的函数
        #ifdef CONFIG_LOCKDEP
            struct lockdep_map lockdep_map;
        #endif
 };
 typedef void (*work_func_t)(struct work_struct *work);
```

五、相关 API

1、创建工作队列：

静态创建：

```
DECLARE_WORK(name,void (*func)(struct work_struct *));
```

动态创建：

```
struct work_struct my_work;                 //定义工作结构体
INIT_WORK(struct work_struct *work, void (*func)(struct work_struct *));
```

2、调度工作队列：

```
int schedule_work(struct work_struct *work);
```

六、区别：

工作队列里面可以休眠， tasklet 禁止休眠。

七、将驱动编译进内核

1、将驱动源文件拷贝到内核相应目录

例：leddrv.c --> linux-3.5\drivers\char

2、进入 linux-3.5\drivers\char，打开 Kconfig，参考已有的选项添加配置菜单选项

```
config MYLEDDRV
bool "MY LEDDRV"
```

3、linux-3.5\drivers\char 目录：

```
#vim Makefile
```

增加编译选项： obj-$(CONFIG_MYLEDDRV) += leddrv.o

4、内核根目录：

```
#make menuconfig
    Device Drivers --->
        Character devices --->
            [*] MY LEDDRV (NEW)
```

已经有我们添加的选项，选中保存退出；

5、根目录#vim .config

搜索 MYLEDDRV 可以看到

```
CONFIG_MYLEDDRV=y
```

6、内核根目录，#make && make zImage

7、将新的内核烧写到开发板启动，可查看驱动已存在；