VxWorks 下任务的控制

任务被 taskSpawn()创建或 taskActivate()激活后，直接进入 Ready 队列。但实际运行时，任务大部分时间处于其它状态，并不是 Ready 态，不然 CPU 的占用率就很高了，功耗也就上去了，那肯定是软件架构的设计出问题了。

今天我们先来看看 VxWorks 的 taskLib 库里关于控制任务状态的一些函数

```
01: STATUS taskSuspend(TASK_ID tid);
02: STATUS taskResume(TASK_ID tid);
03: STATUS taskRestart(TASK_ID tid);
04: STATUS taskDelay(unsigned int ticks);
```

taskSuspend()是将任务放到 Suspend 队列，taskResume()是将任务放到 Ready 队列，这两个 API 通常只是在调试时才会使用。taskSuspend(0)就是挂起任务本身，就像我们在《Task 之常见系统任务》里观察 tRootTask 时所作的。不过 taskResume(0)就是无效操作了，想想为啥？

taskRestart()的主要作用是重启发生严重错误的任务。既然它只有一个参数，说明重启的任务使用的是原有的属性，包括 ID、优先级、选项、入口地址、栈尺寸。其中优先级和选项可能被动态改变过，那就使用最新的值。

taskDelay()通常用在轮询操作之中，它会让当前任务进入 Delay 队列，期限到时之后，再返回 Ready 队列中同优先级任务的尾部。其参数单位是 tick，也就是系统时钟的频率。例如 delay 两秒，通常这样写：

```
01: extern int sysClkRateGet (void);
02: taskDelay (sysClkRateGet ( ) * 2);
```

sysClkRateGet()的作用就是返回系统时钟每秒钟的 tick 数。每个 VxWorks 版本的系统时钟频率默认都是 60，但 taskDelay(120)，不一定是 Delay 两秒。因为可以通过 sysClkRateSet()修改时钟频率

```
01: STATUS sysClkRateSet(int ticksPerSecond);
```

要注意的是，每个 tick 到来时，都会触发系统时钟中断，所以**不建议**把系统时钟频率设置得太快，否则会降低系统性能。

taskDelay()的参数是整型值，所以使用时要注意整除的问题。例如，使用默认的系统时钟频率时，就不可能精确延时 1/7 秒。
sysClkRateGet()/7 = 60/7 = 8，8 个 tick，只是 8/60 秒了。

当 taskDelay()的参数为 0 时，任务并不会进入 Delay 状态，而是直接把 CPU 让给同优先级的任务(如果有的话)，自己排到同优先级任务的尾部，这就相当于 Linux 系统的 sched_yield()操作。

我们在《Task 之任务的创建》里提到过，使用 taskPriorityGet()/taskPrioritySet()，可以查询/调整任务的优先级。另外还有几个 API 可以查询任务的状态信息

```
STATUS taskInfoGet(TASK_ID tid, TASK_DESC *pTaskDesc);
BOOL taskIsReady (TASK_ID tid);
BOOL taskIsPended (TASK_ID tid);
BOOL taskIsDelayed (TASK_ID tid, unsigned int *pDelayTicks);
BOOL taskIsSuspended (TASK_ID tid);
```

从名字就可以看出来 taskInfoGet()最为强大，它可以获得 TCB 里的大部分信息

```
typedef struct {
    TASK_ID td_id;              /* task id */
    RTP_ID  td_rtpId;           /* RTP owning the task */
    char   *td_pExcStackBase;   /* exception stack base */
    char   *td_pExcStackEnd;    /* exception stack end */
    char   *td_pExcStackStart;  /* exception stack start */
    FUNCPTR td_entry;           /* original entry point of task */
    char   *td_sp;              /* saved stack pointer */
    char   *td_pStackBase;      /* the bottom of the stack */
    char   *td_pStackEnd;       /* the actual end of the stack */
    int     td_options;         /* task option bits (see below) */
    int     td_priority;        /* task priority */
    int     td_status;          /* task status */
    int     td_errorStatus;     /* most recent task error status */
    size_t  td_stackSize;       /* size of stack in bytes */
    size_t  td_stackCurrent;    /* current stack usage in bytes */
    size_t  td_stackHigh;       /* maximum stack usage in bytes */
    size_t  td_stackMargin;     /* current stack margin in bytes */
    unsigned int td_delay;      /* delay/timeout ticks */
    EVENTS_DESC  td_events;     /* VxWorks events information */
    char    td_name [VX_TASK_NAME_LENGTH+1];  /* name of task */
    size_t  td_excStackSize;    /* size of exception stack in bytes */
    size_t  td_excStackHigh;    /* exception stack max usage */
    size_t  td_excStackMargin;  /* exception stack margin */
    size_t  td_excStackCurrent; /* current exc stack usage (bytes) */
    int     td_cpuIndex;        /* cpu index running on (if any) */
    cpuset_t td_affinity;       /* task affinity
    } ? end {anonTASK_DESC} ?  TASK_DESC;
```

另外几个就比较简单了，相信只看名字，你就能猜出它们的用法了。

Kernel 里还有很多 API，虽然不属于 taskLib，但是也可以改变任务的状态