TASKCTL

敏捷批量调度开拓者，开启批量调度工具化时代

敏捷调度技术平台 v7

二次开发手册

成都塔斯克信息技术有限公司

产品网站：www.taskctl.com

1 前言

1.1 文档目的

TASKCTL 不仅是一个独立技术平台，同时也是一个开放的技术平台，它提供一整套完整的二次开发接口，用户可以在该接口基础上，开发一系列应用程序，以满足项目的一些特殊需求。

本文旨在通过对二次开发相关知识的详细介绍，指导用户进行调度应用二次开发。

1.2 读者对象

《TASKCTL 7.0 二次开发》主要适合以下读者对象：

- ✓ 技术开发人员

2 开发总述

应用开发主要针对一些特殊任务程序、比如翻牌任务、特色监控程序（短信监控）以及对核心控制等应用的开发。

在 TASKCTL 调度应用中，都面临流程翻牌的情况。对于翻牌应用，TASKCTL 技术平台并未提供该需求的现有解决方法，而是提供一个解决思路，由客户自己完成。主要思路就是：客户根据平台提供开发接口自己编写一个翻牌任务，并将该任务配置到相应处理流程中，从而达到翻牌的目的。

对于监控问题，平台自身带有相关的应用监控程序，并且功能丰富、操作简单流畅。但实际应用过程，客户可能有更多的特殊需求。为此平台提供相应的接口，以便完成客户更为个性化的监控需求，其中短信监控就是一个典型案例。

2.1 开发环境

开发环境分基本环境与 TASKCTL 技术平台开发环境。

（一）基本环境

TASKCTL 技术平台的开发环境必须具备以下条件：

- ✓ 操作系统 UNIX（AIX、Linux）－ C 开发环境，具备 C 编译器
- ✓ TASKCTL-CIR 技术平台环境（安装 TASKCTL-CIR 技术平台及具备）
- ✓ 对于其它环境要求按实际应用而定（比如数据库开发环境、datastage 开发环境等）。

（二）TASKCTL 技术平台开发环境

技术平台开发环境介绍主要对开发目录、头文件目录、CIR 库目录进行简单说明。

开发目录

开发目录指各种 CIR 开发时文件存放目录（并非必须，只是建议）。需要提醒的是：该目录并非项目任务程序开发目录，而是与 CIR 相关的二次开发目录。

- ✓ 应用开发目录：$TASKCTLDIR/src/app

头文件目录

在 TASKCTL 平台进行二次开发时，需要引用很多与平台相关的头文件，这些头文件分别存放于$TASKCTLDIR/include 下的相关目录之下，在此只说明与二次开发相关的头文件。说明如下：

目录	文件	说明
$TASKCTLDIR/ctl	n_ctl.h	主要包含与事件相关的定义
	n_ctlshm.h	主要包含与核心数据访问的定义
	n_shmstruct.h	主要包含与核心数据数据结构的定义
$TASKCTLDIR/ebase	n_ebase.h	主要包含底层基础处理函数定义

库文件目录

库目录为$TASKCTLDIR/lib，该目录有多个库文件，但与二次应用开发相关的库文件主要包括：

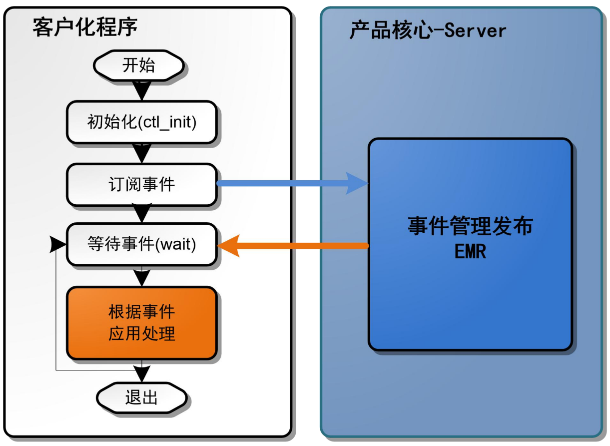
- ✓ **libctlcore.a**：技术平台核心库文件
- ✓ **libebase.a**：底层基础处理库(字符串处理、时间处理等)

2.2 开发模式

TASKCTL 技术平台对于应用开发主要有两种模式，分别为：事件开发模式、核心数据访问模式。

（一）事件开发模式

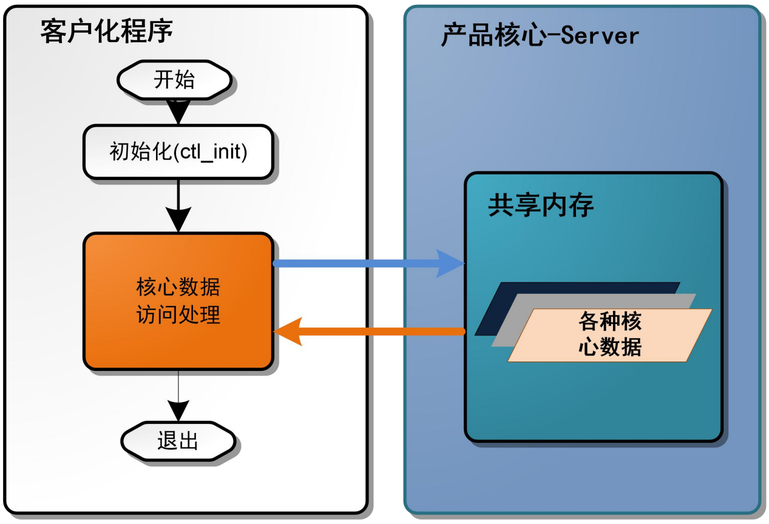
事件开发模式指通过捕获核心的各种事件信息来进行应用开发。如下图所示：



事件开发模式主要用于核心事件访问处理，该模式通过事件相关 API 访问核心事件管理与发布组件(EMR)完成。

（二）核心数据访问模式

核心数据访问模式指通过相应 API 直接访问核心数据来进行应用开发。如下图所示：



核心数据访问模式主要应用于一些控制以及对核心信息的采集。该模式主要通过核心数据操作相关 API 访问共享内存完成。

说明： 事件开发模式与核心数据模式只是两种开发模式，在实际应用中，我们可以将两种模式结合使用。

2.3 编译链接

编译连接可以参考平台安装环境$TASKCTLDIR/src/demo 目录下的 makefile 文件。

应用开发编译连接与 TASKCTL 调度平台相关的库包括： libebase.a、libctlcore.a。如果有其它库链接需求，请自行解决。

2.4 关于 Demo

为了便于用户对应用开发两种模式的理解，TASKCTL 技术平台分别提供两个范例程序，该程序存放于$TASKCTLDIR/demo 目录下面。

（一）事件开发模式范例-mysms.c

该程序是事件开发模式开发的短信应用模拟程序。程序通过捕获核心的短信事件，并通过事件内容获取电话号码以及短信内容，并把相应的信息打印在屏幕上面。

（二）核心数据访问模式范例-passday.c

该程序是利用核心数据访问模式开发的翻牌任务程序。程序通过入口参数 ctlid（流程号）等信息，修改核心流程私有变量 workdate 从而达到翻牌目的。

3 应用开发

本节主要通过对 demo 下相关范例程序进行讲解，使客户分别对二次开发的两种模式有更深入的理解，以便在具体应用中通过这两种开发模式开发更适合的应用程序。

3.1 事件开发模式应用开发

事件开发模式主要通过一个简单的短信范例进行说明，该程序存放于 demo 目下，文件名称为 **mysms.c**。

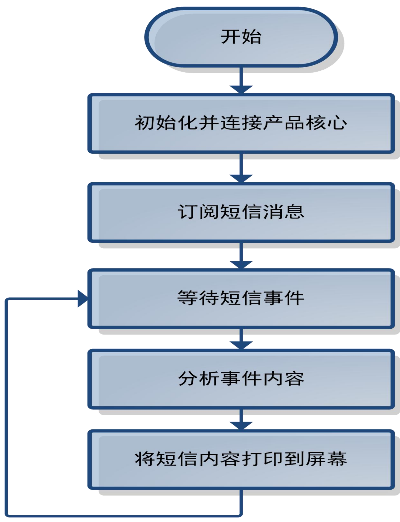
3.1.1 功能描述

该程序是事件开发模式开发的短信应用模拟程序。程序通过破获核心各种事件，按一定需求将相关事件内容生成一定的短信内容，并将该信息打印到屏幕上。

说明： 由于无真实的短信接口，范例程序只能将短信内容通过屏幕的方式打印。但我们可以通过该程序模拟演示事件开发的监控程序效果。

3.1.2 处理流程

该范例程序主要处理流程如下：



3.1.3 代码实现

（一）头文件说明

```
/*********************************************************************
        文件名称:     mysms.c
        功能描述:     CIR技术平台短信应用范例
*********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>

#include "n_ebase.h"
#include "n_cfg.h"
#include "n_ctlmsg.h"
#include "n_ctlshm.h"
#include "n_ctl.h"
```

系统相关头文件

TASKCTL调度平台相关头文件

头文件主要分系统相关头文件以及 TASKCTL 调度平台相关头文件。其中调度平台头文件主要包括：

- ✓ **n_ebase.h**：底层基本处理函数定义
- ✓ **n_cfg.h**：核心基本配置信息处理函数定义
- ✓ **n_ctlmsg.h**：消息通信相关函数定义
- ✓ **n_ctlshm.h**：共享内存相关操作接口定义
- ✓ **n_ctl.h**：二次开发主要接口定义

（二）初始化并连接核心

为了与核心交互，客户化程序首先需要初始化，并与核心建立相关通信通道。通道连接主要包括消息队列连接与共享内存连接。初始化连接主要通过 ctl_init 函数完成。

实现代码如下：

```
    if( ctl_Init() != 0)
    {
        printf("\n程序初始化错误!!");
        fflush(stdout);
        exit(0);
    }
```

ctl_Init 函数说明：

初始化连接主要通过 ctl_init 函数完成。该函数不仅仅完成核心连接，同时，该函数还将后续代码通过子进程运行。在实际应用中，针对调度平台开发的程序主要是守护进程，并以服务程序的方式运行。因此，通过 ctl_init 函数，我们不再需要显示调用 fork 函数来实现进程的后台运行效果。同时，基于该原因，调用该函数初始化失败后，我们需要通过 exit 函数退出子进程。

（三）订阅短信消息

我们与核心建立连接之后，客户化程序就可以与核心通信。由于核心事件比较多，为了使程序更清晰，以及尽量提高访问效率，我们只要求核心给客户化程序发送所关心的相关事件。为此，TASKCTL 通过消息订阅的方式实现该功能。消息订阅时，程序需要显示指出所需的消息类型，并将该信息按一定方式通知核心。这样，核心就只向相关客户化程序发布订阅的消息。

由于核心针对短信封装了专门的事件，所以，在本例中，我们就只需订阅短信消息即可。代码如下：

```
    /*订阅消息，在本例中我们只订阅短信类消息(M_EVENT_SMSNODEMSG)*/
    plist = ctl_CreateSList();
    ctl_AddSubscribe(plist, M_EVENT_SMSNODEMSG);
    ctl_SubscribeMsg(plist);
    ctl_FreeSList(plist);
```

plist 变量

在定于消息之前，首先需要建立一个订阅消息列表，用于存放所订阅的各种事件消息类型，本例中该消息列表通过变量 plist 存放，该变量定义如下：

```
    long ret, pos;
    TEvent event;                /*消息结构*/
    char telecode[100];
    char msg[200];
    TSubscribeList *plist;
```

事件消息列表定义

其中，TSubscribeList 结构体如下：

```
typedef struct _TSubscribeList
{
    long count;
    long list[200];
}TSubscribeList;
```

结构说明：

- ✓ count：订阅事件类型数量。0 表示所有事件类型。
- ✓ list：事件类型列表。该列表通过字符串表达，并用’,’分割各种类型信息。比如：101,102 表示订阅 101 与 102 类型消息。在实际使用中，我们不会对该结构直接赋值，而是通过 ctl_AddSubscribe 函数增加事件类型。

ctl_CreateSList 函数说明

该函数用于创建订阅事件类型队列。在使用队列之前，一定要通过该函数创建队列，并初始化队列。队列不能直接通过定义值类型变量使用。

ctl_AddSubscribe 函数说明

该函数用于订阅类型的添加。参数如下：

- ✓ TSubscribeList *list – 事件列表
- ✓ long msgtype – 事件类型

在本例中，我们只订阅一种事件类型‘M_EVENT_SMSNODEMSG’，即短信类型。

说明：

关于各种事件类型的定义请参见“4.2.2.2 事件类型宏定义”

---

ctl_SubscribeMsg 函数说明

该函数主要功能是向核心订阅消息。参数如下：

- ✓ TSubscribeList *list – 事件列表

通过调用 ctl_SubsribleMsg 函数，程序将准备好的事件消息类型列表中的各种消息类型发向核心，从而完成消息的订阅过程。该函数返回非 0 表示订阅失败。

ctl_FreeSlist 函数说明

该函数主要用于释放消息类型列表。在订阅成功后，我们一定要显示释放事

件消息列表。以便释放相关内存资源。参数如下：

- ✓ TSubscribeList *list – 事件列表

（四）等待短信事件

通过前面步骤，我们完成了向核心的消息订阅过程，接下来，主要工作就是循环等待核心消息并处理。消息的等待，我们通过 ctl_WatiEvent 函数实现。代码如下:

```
        /*等待消息, 以非阻塞方式等待()*/
        ret = ctl_WatiEvent(MS_NOWAIT, &event);
        if(ret == 0)
        {
            sleep(1);
            continue;
        }
        if(ret < 0)
        {
            CTLLOG("系统异常, 程序退出!!");
            fflush(stdout);
            ctl_End();
            exit(0);
        }
```

event 变量

通过 ctl_WaitEvent 等到的事件信息，通过 event 变量存储。该变量定义如下:

```
    long ret, pos;
    TEvent event;                    事件消息内容变量
    char telecode[100];              /*电话号码*/
    char msg[200];                   /*短信内容*/
    TSubscribeList *plist;
```

其中，TEvent 结构体如下：

```
typedef struct _TEvent
{
    long    type;
    char    value[200];
}TEvent;
```

结构说明：

- ✓ type：收到的消息类型。该类型值与定义消息类型对应。
- ✓ value：消息类容。该消息内容是'|'分割的字符串，每个分隔符之间代表不同的值。不同消息对应的事件值不同。具体值与消息之间的对应关系参阅“4.2.2.2 事件类型宏定义”

ctl_WaitEvent 函数说明

该函数主要用于等待核心向应用程序发送的各种事件消息。参数如下：

- ✓ long flag – 阻塞标志。MS_NOWAIT 非阻塞等待事件；MS_WAIT 阻塞方式等待事件。
- ✓ event – 返回事件

本例程序中，消息等待采用非阻塞模式，表示没有消息时，该函数同样有返回。Ctl_WaitEvent 返回-1 表示异常；0 表示无事件消息；大于 0 表示得到消息。

（五）分析事件内容

通过 ctl_WaitEvent 等到的消息，我们需要对内容进行分析，并得到具体信息内容，以便后续处理。代码如下：

```
        /* ret > 0 表示有短信，短信格式为 '(电话号码)|(短信内容)'*/
        pos = str_FindCPos(event.value, '|', 1);
        memset(telecode, 0, sizeof(telecode));
        memset(msg, 0, sizeof(msg));
        strncpy(telecode, event.value, pos);
        strcpy(msg, &(event.value[pos+1]));
```

在本例中，得到的消息都是短息消息，该消息分别由电话号码与短息内容构成，我们只需将该消息内容电话号码与短息内容解析出，并分别存放与 telecode 以及 msg 字符串变量之中。

（六）将短信内容输出到屏幕

最后，我们将短信内容通过对应的电话号码发送即可。关于短息具体发送方法，每家企业都有不同方法，都有相应的接口，在此我们不做过多论述。本例只将信息发送到屏幕，用于模拟短信发送，代码如下：

```
        /*
         *现在可以根据实际短信平台接口函数发送短信了，
         *本例直接将短信打印到屏幕
         */
        printf("\n 时间: %s 号码 %s  短信为: '%s'",
            time_GetDateTime("yyyy-mm-dd hh:mi:ss"), telecode, msg);
        fflush(stdout);
```

（七）所有代码

经过以上步骤，我们完成整个短信应用开发编写，所有代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>

#include "n_ebase.h"
#include "n_cfg.h"
#include "n_ctlmsg.h"
#include "n_ctlshm.h"
#include "n_ctl.h"

int main(int argc, char **argv)
{
    long ret, pos;
    TEvent event;                      /*消息结构*/
    char telecode[100];                /*电话号码*/
    char msg[200];                     /*短信内容*/
    TSubscribeList *plist;

    if( ctl_Init() != 0)
    {
        printf("\n程序初始化错误!!");
        fflush(stdout);
        exit(0);
    }


    /*订阅消息，在本例中我们只订阅短信类消息(M_EVENT_SMSNODEMSG)*/
    plist = ctl_CreateSList();
    ctl_AddSubscribe(plist, M_EVENT_SMSNODEMSG);
    ctl_SubscribeMsg(plist);
    ctl_FreeSList(plist);

    /*循环等待*/
    for(;;)
    {
        /*等待消息，以非阻塞方式等待()*/
        ret = ctl_WatiEvent(MS_NOWAIT, &event);
        if(ret == 0)
        {
            sleep(1);
            continue;
        }
        if(ret < 0)
        {
            CTLLOG("系统异常，程序退出!!");
            fflush(stdout);
            ctl_End();
            exit(0);
        }
```

代码片段一

```
        /* ret > 0 表示有短信，短信格式为 '(电话号码)|(短信内容)'*/
        pos = str_FindCPos(event.value, '|', 1);
        memset(telecode, 0, sizeof(telecode));
        memset(msg, 0, sizeof(msg));
        strncpy(telecode, event.value, pos);
        strcpy(msg, &(event.value[pos+1]));

        /*
         *现在可以根据实际短信平台接口函数发送短信了，
         *本例直接将短信打印到屏幕
         */
        printf("\n 时间: %s 号码 %s  短信为: '%s'",
            time_GetDateTime("yyyy-mm-dd hh:mi:ss"), telecode, msg);
        fflush(stdout);
    }
}
```

代码片段二

3.1.4 编译运行

（一）编译

编译命令如下：

```
cc -c -g  -I/home/taskctl/src/taskctl/include/ebase -I/home/taskctl/src
/taskctl/include/ctl  mysms.c ;
cc -I/home/taskctl/src/taskctl/include/ebase -I/home/taskctl/src/
taskctl/include/ctl  -o mysms mysms.o -lcurses -L/home/taskctl/src/
taskctl/lib -lctlcore -lebase
```

说明： 对于编译链接中头文件以及库的目录应根据具体环境而定

（二）运行

在运行该程序之前，需要启动调度平台、各个 CIR 节点以及相应流程。本例在相应平台启动 1 号流程，该程序运行监控结果下：

```
时间: 2012-04-01 12:20:21 号码:13XXXXXXX01 短信为: 1号流程启动
时间: 2012-04-01 12:22:21 号码:13XXXXXXX01 短信为: 1号流程暂停
时间: 2012-04-01 12:23:21 号码:13XXXXXXX01 短信为: 1号流程暂停取消
时间: 2012-04-01 12:50:21 号码:13XXXXXXX01 短信为: 1号流程24号任务错误
时间: 2012-04-01 13:20:21 号码:13XXXXXXX01 短信为: 1号流程结束
```

3.2 核心数据访问模式开发

核心数据访问开发模式主要通过一个简单的翻牌程序进行说明，该程序存放于 demo 目下，文件名称为 **passday.c**。

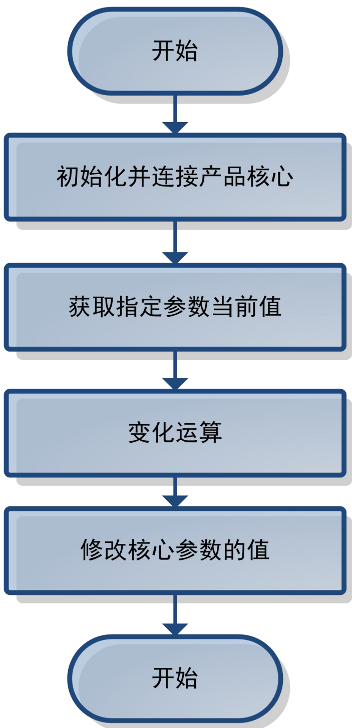
3.2.1 功能描述

该程序是利用核心数据访问模式开发的翻牌任务程序。程序通过入口参数 ctlid（流程号），修改核心变量类型为日期类型的变量。

该程序是一个通用的日期类型变量修改程序。入口参数为

- ✓ 流程号：需要修改流程的编号
- ✓ 日期参数名称：即流程私有变量中类型为日期类型的变量名称。
- ✓ 日期参数格式。比如’yyyy-mm-dd’
- ✓ 变化数：比如 1 表示增加 1 天。-3 表示减少三天。

3.2.2 处理流程



3.2.3 代码实现

该程序的头文件引用与程序初始化与 mysms 程序一样，在此不再论述。本节代码主要讲述“获取指定变量当前值”、“变化运算”以及“修改核心变量值”等三个步骤。

（一）获取指定变量当前值

修改指定变量值之前，先通过 ctl_MemGetParaValue 从共享内存获取指定流程私有变量的当前值，代码如下：

```
/*获取当前参数值*/
p = (char *)ctl_MemGetParaValue(atol(argv[1]), argv[2]);
if (p == NULL)
{
    printf("\n未设置'%s'参数", argv[2]);
    exit(1);
}
```

ctl_MemGetParaValue 函数说明

该函数用于获取核心共享指定流程指定私有变量的当前值，函数参数如下：

- ✓ in_lCtlID – 流程编号
- ✓ in_szName – 私有变量名称

该函数的返回值为字符串类型，即获取的当前值字符串。

本例中流程编号与变量名称分别从程序入口参数 argv[1]以及 argv[2]获取。

（二）变化运算

变化运算指翻天后相应日期变量的实际值计算。该程序计算根据 argv[4]程序入口参数并通过 time_AddFmtDate 来计算，代码如下：

```
/*加天*/
p1 = time_AddFmtDate(p, argv[3], 'D', atol(argv[4]));
if(p1 == NULL)
{
    printf("\n不合法日期格式或日期参数'%s: %s'", argv[2], argv[3]);
    exit(1);
}
```

time_AddFmtDate 函数说明

该函数用于日期格式字符串的日期加减计算，计算结果通过函数返回值返回。函数参数说明如下：

- ✓ char *in_szDate – 被加日期
- ✓ char *in_szFmt – 被加日期格式
- ✓ char in_cFlag – 相加标志。'D' 按日相加；'M' 按月相加；'Y' 按年相加
- ✓ long in_lNum – 相加增量，负数表示相减

在本例中，我们只按日相加。

（三）修改核心变量值

最后，我们需要将计算过的日期写回核心共享内存，从而达到日期修改目的。代码如下：

```
/*修改日期*/
memset(&stParaInfo, 0, sizeof(stParaInfo));
stParaInfo.iCtlID = atol(argv[1]);
strcpy(stParaInfo.szParaName, argv[2]);
strcpy(stParaInfo.szParaValue, p1);
ctl_MemModiPara(&stParaInfo);
printf("\n翻牌任务成功, 结果为%s=%s", argv[2], p1);
```

stParaInfo 变量

该变量用于存放即将修改流程变量信息。其定义如下：

```
char szWorkDate[12];
char szWDatePath[12];
TParaInfo stParaInfo;
char *p, *p1, *p2;
```

流程变量结构体定义

其中，TParaInfo 结构体定义如下：

```
typedef struct _TParaInfo
{
    int      iCtlID;
    char     szParaName[M_FLEN_PARA_Name +1];
    char     szParaValue[M_FLEN_PARA_Value +1];
    short    iIsDatePara;
    short    iParaFlag;
}TParaInfo;
```

结构说明：

- ✓ iCtlID：流程号
- ✓ szParaName：变量名称
- ✓ szParaValue：变量值

- iIsDataPara：变量类型。1-日期变量；0-普通变量。
- iParaFlag：加密标志。1-加密；0-不加密。

ctl_MemModiPara 函数说明

该函数用于修改变量。函数参数说明如下：

- TParaInfo *P – 需修改参数的结构。在修改时，该函数会根据 P 结构中具体变量名称修改共享内存中的相应变量。

（四）所有代码

经过以上步骤，我们完成整个短信应用开发编写，所有代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
#include <sys/wait.h>
#include "n_ebase.h"
#include "n_cfgxml.h"
#include "n_ctlmsg.h"
#include "n_ctlshm.h"
#include "n_ctl.h"

int main(int argc, char **argv)
{
    char szWorkDate[12];
    char szWDatePath[12];
    TParaInfo stParaInfo;
    char *p, *p1, *p2;

    if(argc!=5)
    {
        printf("请正确输入参数");
        fflush(stdout);
        exit(1);
    }
    if( ctl_Init() != 0)
    {
        printf("\n程序初始化错误!!");
        fflush(stdout);
        exit(0);
    }
```

代码片段一

```
    /*获取当前参数值*/
    p = (char *)ctl_MemGetParaValue(atol(argv[1]), argv[2]);
    if (p == NULL)
    {
        printf("\n未设置'%s'参数", argv[2]);
        exit(1);
    }
    /*加天*/
    p1 = time_AddFmtDate(p, argv[3], 'D', atol(argv[4]));
    if(p1 == NULL)
    {
        printf("\n不合法日期格式或日期参数'%s: %s'", argv[2], argv[3])
        exit(1);
    }
    /*修改日期*/
    memset(&stParaInfo, 0, sizeof(stParaInfo));
    stParaInfo.iCtlID = atol(argv[1]);
    strcpy(stParaInfo.szParaName, argv[2]);
    strcpy(stParaInfo.szParaValue, p1);
    ctl_MemModiPara(&stParaInfo);
    printf("\n翻牌任务成功, 结果为%s=%s", argv[2], p1);
    exit(0);
}
```

代码片段二

3.2.4 编译运行

（一）编译

编译命令如下：

```
cc -c -g  -I/home/taskctl/src/taskctl/include/ebase -I/home/taskctl/src
/taskctl/include/ctl  passday.c ;
cc -I/home/taskctl/src/taskctl/include/ebase -I/home/taskctl/src/taskctl
/include/ctl  -o passday passday.o -lcurses -L/home/taskctl/src/taskctl
/lib -lctlcore -lebase
```

说明： 对于编译链接中头文件以及库的目录应根据具体环境而定

（二）运行

在运行该程序之前，与 mysms 程序一样，需要启动调度平台。另外，一般情况，修改日期变量程序应用与具体流程当中。但在本文档中，我们通过手工运行，以便体验程序运行效果。本例主要修改 1 号流程的 workdate 变量，并且在原有值基础上加 1 天。

运行前

运行前，我们通过后台客户端程序 ctlmoni 查看 1 号流程的变量值，情况如下：

```
MONI_[02][-001]> flowid 1

MONI_[02][0001]> listpara

  类型     变量名称                      变量值
------------------------------------------------------------------------------------------
  私有     sysdate                       2012/06/17
  私有     workdate                      2012-06-18
------------------------------------------------------------------------------------------
                                                                  共 2 条记录
MONI_[02][0001]>
```

workdate变量修改前

运行命令

我们在产品核心服务节点运行以下命令：

```
[taskctl@linux2 demo]$ passday 1 workdate yyyy-mm-dd 1

翻牌任务成功, 结果为workdate=2012-06-19[taskctl@linux2 demo]$
[taskctl@linux2 demo]$
```

运行后结果

运行后，我们通过后台客户端程序 ctlmoni 查看 1 号流程的变量值，情况如下：

```
MONI_[02][-001]> flowid 1

MONI_[02][0001]> listpara

  类型     变量名称                      变量值
------------------------------------------------------------------------------------------
  私有     sysdate                       2012/06/17
  私有     workdate                      2012-06-19
------------------------------------------------------------------------------------------
                                                                  共 2 条记录
MONI_[02][0001]>
```

workdate变量已修改

4 开发函数包说明

4.1 平台连接与断开

4.1.1 概述

所有 TASKCTL-CIR 技术平台二次开发，都需要连接以及断开核心操作。与核心相连才可以和核心通信。

连接和断开分别由 ctl_Init 函数与 ctl_End 函数分别完成。

以上两个函数相关头文件为 n_ctl.h；连接库为 libctlcore.a。

4.1.2 函数说明

4.1.2.1 ctl_Init

- 函数原形：long ctl_Init();
- 功能：初始化，连接核心
- 参数：无
- 返回值：long

    CTLC_SUCCESS - 成功

    CTLC_FAILURE - 失败

4.1.2.2 ctl_End

- 函数原形：void ctl_ End()
- 功能：断开核心，释放资源
- 参数：无
- 返回值： void

4.2 事件类

4.2.1 概述

事件类函数完成与核心事件相关的一系列操作。用户程序要获取核心事件，首先要订阅核心事件，也就是说，用户只能获取自己订阅的核心事件。

事件类函数相关头文件为 n_ctl.h。连接库为 ctlcore.a

4.2.2 相关结构与定义说明

4.2.2.1 数据结构定义

事件类主要包括以下结构类型：

（一）TSubscribeList 类型

此结构为事件订阅列表类型。订阅事件是通过该列表确定的。定义如下：

```
typedef struct _TSubscribeList
{
    long count;
    long list[200];
}TSubscribeList;
```

说明：

1．count – 定义事件种数。0 表示所有事件类型

2．list – 定义事件列表。

（二）TEvent 类型

此结构为事件类型。从核心得到的事件就是通过该结构返回。定义如下：

```
typedef struct _TEvent
{
   long   type;
   char   value[200];
}TEvent;
```

说明：

1．type – 事件类型

2．value – 核心事件值。核心事件值全用‘|’分割符可见字符串表示。

4.2.2.2 事件类型宏定义

以下是有关事件的定义：

事件名称	事件值	事件说明	信息内容
M_EVENT_ALL	100	所有时间	在订阅事件时使用，表示所有时间,核心并不输出该事件
M_EVENT_CTLSTART	101	平台启动	无
M_EVENT_CTLSTOP	102	平台停止	无
M_EVENT_FLOWSTART	103	流程启动	1. 流程号 2. 流程名称
M_EVENT_FLOWSTOP	104	流程停止	1. 流程号 2. 流程名称
M_EVENT_CYCLEBEGIN	105	一个流程循环开始	1. 流程号 2. 流程名称
M_EVENT_CYCLEEND	106	一个流程循环结束	1. 流程号 2. 流程名称
M_EVENT_STATECHANGE	107	任务状态变化	1. 流程号 2. 任务号 3. 任务名称 4. 运行属性(0. 任务 1. 串行 2. 并行) 5. 变化前状态 6. 变化后状态
M_EVENT_FLOWPAUSE	108	流程暂停	1. 流程号 2. 标志 0 暂停 1 取消暂停
M_EVENT_FLOWSTATEERR	109	流程状态异常	1. 流程号 2. 异常原因说明
M_EVENT_SYSNETERR	110	网络错误(远程任务调用会发生)	
M_EVENT_SYSMSG	111	队列异常	
M_EVENT_SYSSHM	112	共享内存错误	
M_EVENT_CFGERR	113	配置错(主要包含一些客户化的东西不合理,比如没有配置 workdate, period 格式错误等)	1. 说明信息
M_EVENT_SYSERR	114	系统异常	1. 说明信息
M_EVENT_USERCOMMMSG	120	用户通过程序 ctlsendmsg 发送消息	1. 流程号 2. 客户信息
M_EVENT_SMSNODEMSG	121	短信息	1. 电话号码 2. 短信内容

4.2.3 函数说明

4.2.3.1 ctl_CreateSList

✧ 函数原形：TSubscribeList *ctl_CreateSList()

✧ 功能：创建事件订阅列表

✧ 参数：无

✧ 返回值： TSubscribeList

返回事件列表指针。NULL 表示创建失败。

4.2.3.2 ctl_FreeSList

✧ 函数原形：void ctl_FreeSList(TSubscribeList *p);

✧ 功能：释放消息订阅列表

✧ 参数：

p – 时间列表指针

✧ 返回值： void

✧ 其他说明：

事件订阅后一定要释放列表。

4.2.3.3 ctl_AddSubscribe

函数原形：long ctl_AddSubscribe(TSubscribeList *list, long msgtype);

✧ 功能：添加订阅事件类型

✧ 参数：

list – 事件列表

msgtype – 事件类型

✧ 返回值： long

0 – 成功

非 0 – 失败

4.2.3.4 ctl_SubscribeMsg

- 函数原形：long ctl_SubscribeMsg(TSubscribeList *list);
- 功能：按订阅列表向核心订阅事件
- 参数：

  list – 事件列表
- 返回值： long

  0 – 成功

  非 0 - 失败
- 其他说明：

  如果不向核心订阅事件，客户端不能收到任何消息。

4.2.3.5 ctl_unSubscribe

- 函数原形：long ctl_unSubscribe();
- 功能：取消事件订阅
- 参数：无
- 返回值： long

  0 – 成功

  非 0 – 失败
- 其他说明：

  如果不需要等待核心事件，应该主动向核心发起事件订阅取消指令。这样会减少核心不必要的资源损耗。

4.2.3.6 ctl_WatiEvent

- 函数原形：long ctl_WatiEvent(long flag, TEvent *event);
- 功能：等待事件
- 参数：

  flag – 阻塞标志。MS_NOWAIT 非阻塞等待事件；MS_WAIT 阻塞方式等待事件。

  event – 返回事件

✧ 返回值： long

0 无消息

>0 成功获取一条消息

-1 异常

-2 消息订阅被取消

4.2.3.7 ctl_GetEventValue

✧ 函数原形：char *ctl_GetEventValue(TEvent *event, long no);

✧ 功能：按事件值序号获取事件值。

✧ 参数：

event – 事件

no - 事件值序号

✧ 返回值： char *

指定序号的事件值。NULL 表示无相应的事件值。

✧ 其他说明：

4.3 核心数据-流程基本信息类

4.3.1 概述

核心数据-流程基本信息类函数主要针对核心流程基本信息数据增删改操作。用户在应用时，只需通过 ctl_Init 函数连接核心后即可通过该类函数进行操作。

核心数据-流程基本信息类函数相关头文件为 n_ctlshm.h、n_shmstruct.h。连接库为 ctlcore.a。

4.3.2 相关结构与定义说明

4.3.2.1 数据结构定义

在对流程基本信息进行操作时，我们主要是通过流程基本信息结构进行信息传递，该结构如下：

```c
typedef struct _TFlowInfo
{
    short    iEventRun;
    short    iEventState;
    int      iAppID;
    int      iCtlID;
    char     szCtlName[M_FLEN_FLOWINFO_Name + 1];
    short    iCtlState;
    short    iQuitState;
    char     szStateMsg[200];
    int      iMaxJob;
    int      iRunningNum;
    char     szWorkDateParaName[60];
    char     szStartModul[40];
    char     szFlowDesc[M_FLEN_FLOWINFO_Desc +1];
    long     lNodeCount;
    long     lParaCount;
    long     lFlowPid;
    char     iIsReleased;
    char     iIsCheckOut;
    char     szCheckOutID[30];
    char     szCheckOutTime[20];
    char     szUpdateTime[20];
    long     lStartFlowID;
    long     lStartJobID;
}TFlowInfo;
```

以上结构中相关字段分用户使用字段与系统内部使用字段，内部使用字段用户可以查看，但不能修改。用户使用字段为：

iCtlID	流程号
szCtlName	流程名称
iMaxJob	流程最大任务并行数
szXMLFileName	流程对应的入口流程控制 XML 文件名称
szCtlLogPath	流程调度日志目录

4.3.2.2 宏定义

以下为流程状态宏定义：

状态名称	状态值	状态说明
M_FLOW_STATE_INIT	0	初始化
M_FLOW_STATE_PAUSE	1	暂停状态
M_FLOW_STATE_RUNNING	2	正在运行状态
M_FLOW_STATE_BEGINNING	3	正在开始状态
M_FLOW_STATE_ENDDING	4	正在结束状态
M_FLOW_STATE_IDLE	5	空闲

M_FLOW_STATE_QUIT	6	退出

详细解释：

- M_FLOW_STATE_INIT　初始化状态。每个流程对应一个调度中心，当中心刚启动还在初始化相关数据的阶段，我们就定义为初始化状态。
- M_FLOW_STATE_PAUSE　暂停状态。在暂停状态时，调度中心依然存在，但不做任务调度动作。
- M_FLOW_STATE_RUNNING　正在运行状态。表示一个流程对某一批次已经开始运行，并且开始任务已经成功运行，但还没到结束任务。我们把这一阶段的状态定义为正在运行状态。
- M_FLOW_STATE_BEGINNING　正在开始状态。开始任务正在运行阶段，我们就称为正在开始阶段。
- M_FLOW_STATE_ENDDING　正在结束状态。只剩下结束任务没运行的阶段，我们就成为正在结束阶段。
- M_FLOW_STATE_IDLE　空闲状态。当一个流程的结束任务已经完成，既流程一个批次运行完成，但下一批次又没开始的阶段，我们就称为状态。

M_FLOW_STATE_QUIT　退出状态。当一个流程所对应的调度中心被退出或未启动，这种状态我们称为退出状态。

4.3.3 函数说明

4.3.3.1 ctl_MemGetCountOfFlow

- 函数原形：long ctl_MemGetCountOfFlow()
- 功能：获取流程基本信息记录条数
- 参数：无
- 返回值：　long 返回记录数

4.3.3.2 ctl_MemGetFlowByID

- 函数原形：long ctl_MemGetFlowByID(int in_iCtlID, TFlowInfo *out_pstFlowInfo, long *out_lPos);

✧ 功能：根据流程号获取流程基本信息

✧ 参数：

in_iCtlID – 需要获取的流程号

out_pstFlowInfo – 输出获取的流程基本信息

out_lPos – 输出当前流程基本信息在核心的相对位置

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.3.3.3 ctl_MemGetFirstFlow

✧ 函数原形： long ctl_MemGetFirstFlow(TFlowInfo *out_pstFlowInfo, long *out_lPos);

✧ 功能：获取第一条流程基本信息

✧ 参数：

out_pstFlowInfo -输出获取的流程基本信息

out_lPos -输出当前流程基本信息在核心的相对位置

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.3.3.4 ctl_MemGetNextFlow

✧ 函数原形： long ctl_MemGetNextFlow(int in_iPos, TFlowInfo *out_pstFlowInfo, long *out_lPos);

✧ 功能：获取指定位置的下一条流程基本信息

✧ 参数：

in_iPos – 指定的位置

out_pstFlowInfo -输出获取的流程基本信息

out_lPos -输出当前流程基本信息在核心的相对位置

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.3.3.5 ctl_MemModiFlow

✧ 函数原形：long ctl_MemModiFlow(TFlowInfo *p);

✧ 功能：修改流程基本信息

✧ 参数：

P – 需修改的流程基本信息数据结构

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.3.3.6 ctl_MemAddFlow

✧ 函数原形：long ctl_MemAddFlow(TFlowInfo *p);

✧ 功能：向核心增加流程基本信息

✧ 参数：

P – 需增加的流程基本信息

✧ 返回值： long

CTLC_SUCCESS 增加成功

CTLC_UNIQUEERR 流程号重复，不能增加

CTLC_FAILURE 增加异常

4.3.3.7 ctl_MemDeleteOneFlow

✧ 函数原形：long ctl_MemDeleteOneFlow(long in_iCtlID);

✧ 功能：从核心删除指定编号的流程基本信息

✧ 参数：

in_iCtlID – 需删除的流程号

✧ 返回值： long

CTLC_SUCCESS 删除成功

CTLC_NODATA 无数据

✧ 其他说明：流程基本信息删除后，但其对应的流程任务节点信息以及参数信息并为从核心数据删除。

4.4 核心数据-流程私有变量类

4.4.1 概述

核心数据-流程私有变量类函数主要针对核心流程私有变量信息数据增删改操作。用户在应用时，只需通过 ctl_Init 函数连接核心后即可通过该类函数进行操作。

核心数据-流程私有变量信息类函数相关头文件为 n_ctlshm.h、n_shmstruct.h。连接库为 ctlcore.a。

4.4.2 相关结构与定义说明

在对流程私有参数信息进行操作时，我们主要是通过流程私有参数信息结构进行信息传递，该结构如下：

```
typedef struct _TParaInfo
{
    int      iCtlID;
    char     szParaName[M_FLEN_PARA_Name +1];
    char     szParaValue[M_FLEN_PARA_Value +1];
    short    iIsDatePara;
    short    iParaFlag;
}TParaInfo;
```

4.4.3 函数说明

4.4.3.1 ctl_MemGetCountOfPara

✧ 函数原形：long ctl_MemGetCountOfPara(long in_lCtlID);

✧ 功能：获取指定流程的私有参数记录数

✧ 参数：

in_lCtlID - 流程编号

✧ 返回值： long 记录数

4.4.3.2 ctl_MemGetParaByName

✧ 函数原形： long ctl_MemGetParaByName(long in_lCtlID, char *in_szName, TParaInfo *out_pNode, long *out_lPos);

✧ 功能：根据参数名称获取参数记录数据结构

✧ 参数：

in_lCtlID – 流程编号

in_szName – 参数名称

out_pNode – 输出获取的参数信息结构

out_lPos – 输出参数在核心的相对位置

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.4.3.3 ctl_MemGetParaValue

✧ 函数原形：char *ctl_MemGetParaValue(long in_lCtlID, char *in_szName);

✧ 功能：根据参数名称直接指定获取参数值

✧ 参数：

in_lCtlID – 流程编号

in_szName – 参数名称

✧ 返回值： char *

返回参数值，NULL 表示无此参数

4.4.3.4 ctl_MemGetFirstPara

✧ 函数原形：long ctl_MemGetFirstPara(long in_lCtlID, TParaInfo *out_pPara, long *out_lPos);

✧ 功能：获取指定流程的第一个私有参数

✧ 参数：

in_lCtlID -流程编号

out_pPara – 输出获取的参数信息结构

out_lPos – 输出参数在核心的相对位置

✧ 返回值：long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.4.3.5 ctl_MemGetNextPara

✧ 函数原形：long ctl_MemGetNextPara(long in_lCtlID, long in_lPos, TParaInfo *out_pPara, long *out_lPos);

✧ 功能：获取指定流程指定位置的下一条参数信息记录结构

✧ 参数：

in_lCtlID-流程编号

in_lPos – 参数位置

out_pPara – 输出获取的参数信息结构

out_lPos – 输出参数在核心的相对位置

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.4.3.6 ctl_MemModiPara

✧ 函数原形：long ctl_MemModiPara(TParaInfo *p)

✧ 功能：修改参数(实际上只能修改参数的参数值，参数名称与流程好都是索引值)

✧ 参数：

P – 需修改参数的结构

✧ 返回值： long

CTLC_NODATA 无数据

CTLC_SUCCESS 修改成功

4.4.3.7 ctl_MemAddPara

- ✧ 函数原形：long ctl_MemAddPara(TParaInfo *p);
- ✧ 功能：增加参数
- ✧ 参数：

  P – 需增加参数的结构
- ✧ 返回值： long

  CTLC_FAILURE 异常

  CTLC_UNIQUEERR 参数名重复

  CTLC_SUCCESS 增加成功

4.4.3.8 ctl_MemDeleteOneParaByName

- ✧ 函数原形：long ctl_MemDeleteOneParaByName(long in_lCtlID, char *in_szParaName);
- ✧ 功能：根据指定流程号以及参数名称删除核心流程参数
- ✧ 参数：

  in_lCtlID - 流程号

  in_szParaName – 参数名称
- ✧ 返回值： long

  CTLC_NODATA 无数据

  CTLC_SUCCESS 删除成功

4.5 核心数据-流程任务节点类

4.5.1 概述

核心数据-流程任务节点类函数主要针对核心流程节点信息数据查询访问改操作， 用户不能对其增删改。用户在应用时，只需通过 ctl_Init 函数连接核心后即可通过该类函数进行操作。

核心数据-流程任务节点类函数相关头文件为 n_ctlshm.h、n_shmstruct.h。连接库为 ctlcore.a。

4.5.2 相关结构与定义说明

4.5.2.1 数据结构定义

在对流程节点信息进行查询访问操作时，我们主要是通过流程节点信息结构进行信息传递，该结构如下：

```
typedef struct _TNodeInfoAll
{
    /*基本属性*/
    short   iCtlID;
    short   iNodeNo;
    char    szNodeName[M_FLEN_Node_NodeName+1];
    short   iType;
    char    szTypeName[M_FLEN_Node_TypeName+1];
    char    szProgName[M_FLEN_Node_ProgName+1];
    char    szPara[M_FLEN_Node_Para+1];
    char    szExpPara[M_FLEN_Node_ExpPara+1];
    short   iAgentID;
    char    szMoniTitle[M_FLEN_Node_MoniTitle+1];
    char    szGroupName[40];
    char    szJobDesc[M_FLEN_Node_JobDesc+1];
    short   iIsEnd;

    /*控制属性*/
    short   iRunAttr;
    short   iNodeAttr;
    short   iLastNode;
    char    szDateType[20];
    char    szPeriod[M_FLEN_Node_Period+1];
    char    szUnPeriod[M_FLEN_Node_Period+1];
    short   iPeriodEnb;
    short   iMaxNum;
    char    szCondition[M_FLEN_Node_Condition+1];
    char    szOstr[M_FLEN_Node_Ostr+1];
    char    szLean[M_FLEN_Node_Lean+1];
    short   iCycle;
    short   iIsPause;
    short   iLock;
    short   iEnable;
    short   iEgnoreErr;
    short   iAgentCtlFlag;

    /*运行动态属性*/
    short   iRunState;
    short   iRunResult;
    char    szBeginTime[M_FLEN_Node_BeginTime+1];
    char    szEndTime[M_FLEN_Node_EndTime+1];
    short   iReNum;
    short   iOKNum;
    char    szRunMsg[M_FLEN_Node_RunMsg+1];
    long    lLastTC;
    long    lHostoryNum;
    long    lAverageTC;
}TNodeInfoAll;
```

4.5.2.2 宏定义

状态名称	状态值	状态说明
M_JOB_STATE_NETERR	-1	网络错误
M_JOB_STATE_NOCOMMFILE	-2	通信文件不存在（其实该情况根本无法返回）
M_JOB_STATE_NOPLUGIN	-3	无执行插件
M_JOB_STATE_CTLNODEERR	-4	控制接点冲
M_JOB_STATE_MSGERR	-5	控制接点冲
M_JOB_STATE_SYSBUSS	-6	系统忙
M_JOB_STATE_SYSOERR	-7	系统其它错
M_JOB_STATE_NOISSUE	-8	程序未发布
M_JOB_STATE_ROUTEERR	-9	错误路由
M_JOB_STATE_NONETINFO	-10	无网络配置信息
M_JOB_STATE_INIT	1	未执行
M_JOB_STATE_RUNNING	2	正在执行
M_JOB_STATE_CYCLE	3	正在循环
M_JOB_STATE_ERR	4	错误需再次执行， 次数为满
M_JOB_STATE_FAILED	10	错误不再执行
M_JOB_STATE_SUCCESS	11	成功执行
M_JOB_STATE_OKPERIOD	12	因周期本期不执行通过
M_JOB_STATE_IGNOREERR	13	忽略错误通过
M_JOB_STATE_DISABLE	14	该任务因为无效通过

4.5.3 函数说明

4.5.3.1 ctl_MemGetCountOfNode

- 函数原形：long ctl_MemGetCountOfNode(long in_lCtlID);
- 功能：获取指定流程任务（组）节点记录数
- 参数：

  in_lCtlID - 流程号
- 返回值：long

  返回记录数

4.5.3.2 ctl_MemGetNodeByID

- 函数原形：long ctl_MemGetNodeByID(long in_lCtlID, long in_lNodeNo, TNodeInfoAll *out_pNode, long *out_lPos);

✧ 功能：根据流程号以及任务（组）节点号获取任务（组）节点信息

✧ 参数：

in_lCtlID – 流程号

in_lNodeNo - 任务（组）节点号

out_pNode – 输出获取的任务（组）节点信息

out_lPos – 输出获取任务（组）节点信息记录核心相对位置

✧ 返回值：

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.5.3.3 ctl_MemGetNodeByName

✧ 函数原形：long ctl_MemGetNodeByName(long in_lCtlID, char *in_szName, TNodeInfoAll *out_pNode, long *out_lPos);

✧ 功能：根据指定流程号以及任务（组）节点名称获取任务（组）节点信息

✧ 参数：

in_lCtlID – 流程号

in_szName - 任务（组）节点名称

out_pNode – 输出获取的任务（组）节点信息

out_lPos – 输出获取任务（组）节点信息记录核心相对位置

✧ 返回值：

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.5.3.4 ctl_MemGetFirstNode

✧ 函数原形：long ctl_MemGetFirstNode(long in_lCtlID, TNodeInfoAll *out_pNode, long *out_lPos);

✧ 功能：根据流程号获取第一条任务（组）节点信息

✧ 参数：

in_lCtlID – 流程号

out_pNode – 输出获取的任务（组）节点信息

out_lPos – 输出获取任务（组）节点信息记录核心相对位置

✧ 返回值：

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.5.3.5 ctl_MemGetNextNode

✧ 函数原形：long ctl_MemGetNextNode(long in_lCtlID, long in_lPos, TNodeInfoAll *out_pNode, long *out_lPos);

✧ 功能：根据流程号获取以及任务节点位置获取下一条任务（组）节点信息

✧ 参数：

in_lCtlID – 流程号

in_lPos – 任务（组）节点信息位置

out_pNode – 输出获取的任务（组）节点信息

out_lPos – 输出获取任务（组）节点信息记录核心相对位置

✧ 返回值：

CTLC_NODATA 无数据

CTLC_SUCCESS 成功

4.5.3.6 ctl_MemGetPrevNode

✧ 函数原形：long ctl_MemGetPrevNode(long in_lCtlID, long in_lPos, TNodeInfoAll *out_pNode, long *out_lPos);

✧ 功能：根据流程号获取以及任务节点位置获取上一条任务（组）节点信息

✧ 参数：

in_lCtlID – 流程号

in_lPos – 任务（组）节点信息位置

out_pNode – 输出获取的任务（组）节点信息

out_lPos – 输出获取任务（组）节点信息记录核心相对位置

✧ 返回值：

CTLC_NODATA　　无数据

CTLC_SUCCESS　成功

4.6 EBASE 基础操作类

4.6.1 概述

Ebase 基础操作类主要分两类：字符串处理类、时间日期处理类。

对于字符串处理函数，字符串必须满足两个条件：

1．所有处理的字串均为可见字符串

2．字串处理长度不能超过 5000

对于时间日期处理函数，所有日期格式串按如下定义：

yyyy - 四位年

yy - 两位年

mm - 月

dd - 天

hh - 小时

mi - 分钟

ss - 秒

wd - 礼拜几

4.6.2 字串处理类

4.6.2.1 str_LTrim

✧ 函数原形：char *str_LTrim(char *str)

✧ 功能：　删除字串左空格

✧ 参数：

str – 需删除空格的字串

✧ 返回值：

删除后字串指针

4.6.2.2 str_RTrim

✧ 函数原形：char *str_RTrim(char *str)

✧ 功能：删除字串右空格

✧ 参数：

str – 需删除空格的字串

✧ 返回值：

删除后字串指针

4.6.2.3 str_LRTrim

✧ 函数原形：char *str_LRTrim(char *str)

✧ 功能：删除字串左右空格

✧ 参数：

str – 需删除空格的字串

✧ 返回值：

删除后字串指针

4.6.2.4 str_TrimCRLF

✧ 函数原形：char* str_TrimCRLF(char* in_psz)

✧ 功能：去掉输入字符串尾部的换行符以及空格和制表符

✧ 参数：

in_psz– 目标串

✧ 返回值：

处理后字串指针

4.6.2.5 str_GetValue

✧ 函数原形：char *str_GetValue(char *str, char c, int No)

✧ 功能：取分割符字符串中指定位置的值

✧ 参数：

str - 分割符字串

c - 分割符值

no - 需获取值的序号

✧ 返回值：

分割符字符串中指定位置的值

4.6.2.6 str_ReplaceStr

✧ 函数原形：char *str_ReplaceStr(char *str1, char *str2, char *str3, int num)

✧ 功能： 字符串替换

✧ 参数：

str1 – 被替换的字符串

str2 – 被替换的指定字符串

str3 – 用于替换的指定字符串

num – 替换的相对位置。从 1 开始； 0 表示替换整个字符串。

✧ 返回值：

替换后的字符串指针

4.6.2.7 str_GridField

✧ 函数原形：char * str_GridField(char *in_szStr, int in_iLen, int in_iFlag)

✧ 功能： 用空格填充对齐字符串

✧ 参数：

in_szStr – 需对齐的字串

in_iLen – 对齐长度

in_iFlag – 对齐标志。-1 左对齐；0 中间对齐 1 右对齐

✧ 返回值：

对齐后字串指针

4.6.3 时间处理类

4.6.3.1 time_GetDateTime

- 函数原形：char *time_GetDateTime(char *fmt)
- 功能： 获取系统日期时间
- 参数：

  Fmt – 日期时间格式
- 返回值：

  时间日期指针

4.6.3.2 time_IsDate

- 函数原形：long time_IsDate(char* in_szDate, char *in_szFmt)
- 功能： 判断日期是否合法
- 参数：

  in_szDate – 日期字串

  in_szFmt – 输入日期字串格式
- 返回值：

  0 合法

  -1 不合法

4.6.3.3 time_FmtDateToYYYYMMDD

- 函数原形：char *time_FmtDateToYYYYMMDD(char *in_szDate, char *in_szFmt)
- 功能： 将格式日期串转变成 yyyymmdd 格式日期串
- 参数：

  in_szDate – 被转换日期字串

  in_szFmt – 被转换日期格式
- 返回值：

  转换后的日期串指针

4.6.3.4 time_YYYYMMDDToFmtDate

- ✧ 函数原形：char *time_YYYYMMDDToFmtDate(char *in_szDate, char *in_szFmt)
- ✧ 功能： 将 yyyymmdd 格式日期传转变成指定格式的日期串
- ✧ 参数：

  in_szDate – 被转换日期

  in_szFmt – 转换后的格式
- ✧ 返回值：

  转换后的日期字串指针

4.6.3.5 time_DateToLong

- ✧ 函数原形：long time_DateToLong(char *in_szDate, char *in_szFmt)
- ✧ 功能： 将格式日期字串转换成整数日期。1 表示'1900-01-01'，以后每天加 1
- ✧ 参数：

  in_szDate - 被转换的日期字串

  in_szFmt - 转换的日期格式
- ✧ 返回值：

  转换后的整数日期，如果小于 0，表示输入日期不合法。

4.6.3.6 time_LongToDate

- ✧ 函数原形：char * time_LongToDate(long in_iDate, char *in_szFmt)
- ✧ 功能： 将整数日期转换成格式日期串
- ✧ 参数：

  in_iDate – 需转换的整数日期

  in_szFmt – 转换后的日期格式
- ✧ 返回值：

  转换后格式日期串指针

4.6.3.7 time_AddFmtDate

- 函数原形：char *time_AddFmtDate(char *in_szDate, char *in_szFmt, char in_cFlag, long in_lNum)
- 功能： 对输入日期按日、月、年相加
- 参数：

  in_szDate – 被加日期

  in_szFmt – 被加日期格式

  in_cFlag – 相加标志。’D’ 按日相加；’M’ 按月相加；’Y’ 按年相加

  in_lNum – 相加增量，负数表示相减
- 返回值：

  相加后的日期串指针，格式不变。