十年一日，深入成就深度
业精于专，专注成就专业

数字有机体工作平台及抗毁容灾系统

开发手册

成都天心悦高科技发展有限公司

2014 年 11 月

版权声明

数字有机体系统及其附属产品（含 Windows 客户端调度接口库）的版权属于成都天心悦高科技发展有限公司所有。任何组织和个人未经成都天心悦高科技发展有限公司许可与授权，不得擅自复制、更改该软件的内容及其产品包装。

本软件受版权法和国际条约的保护。如未经授权而擅自复制或传播本程序（或其中任何部分），将受到严厉的刑事及民事制裁，并将在法律许可的范围内受到最大可能的起诉！

成都天心悦高科技发展有限公司
地址：成都市武侯区棕南小区
电话：028－83318559
邮编：610054

目 录

1 前言.....1

1.1 编写约定.....1

1.2 如何使用本手册.....1

1.3 相关文档说明.....1

1.4 如何获得技术支持.....2

2 简介.....3

2.1 数字有机体系统简介.....3

2.2 接口概述.....3

3 编程导引.....5

3.1 系统编程.....5

3.1.1 接口调用主要流程.....5

3.1.2 参数配置流程.....5

3.1.3 用户管理.....6

3.1.4 定时器设置流程.....6

3.1.5 服务管理流程.....7

3.1.6 文件系统操作流程.....7

3.2 调度系统.....8

3.2.1 SDK 接口调用的主要流程.....8

3.3 编译说明.....8

4 系统编程接口.....10

4.1 接口定义.....10

4.1.1 共享存储管理.....10

4.1.2 用户组管理.....12

4.1.3 用户管理.....16

4.1.4 节点配置文件管理.....27

4.1.5 站点、节点查询与管理.....28

4.1.6 站点、节点负载管理.....30

4.1.7 站点、节点报警管理.....36

4.1.8 服务管理.....40

4.1.9 虚拟服务管理.....51

4.1.10 副本（块）的位置与属性管理.....52

4.1.11 系统故障后修复.....59

4.1.12 磁盘空间均衡管理.....61

4.1.13 敏感信息存储管理.....73

4.1.14 文件系统的事务管理.....77

4.1.15 其它辅助接口.....79

4.1.16 文件系统 86
4.2 宏定义 88
4.3 数据结构 89
4.3.1 user_landing_host 89
4.3.2 name_list_item 90
4.3.3 p_basic_user_info 90
4.3.4 p_user_authen_info 91
4.3.5 PuserQuota 91
4.3.6 p_group_info 92
4.3.7 pg_list_info 93
4.3.8 SDesc 93
4.3.9 RLDesc 94
4.3.10 Service_Node_Info 95
4.3.11 Node_Load 95
4.3.12 Disk_Info 96
4.3.13 load_info 96
4.3.14 Route_Info 97
4.3.15 routes 98
4.3.16 Service_Ack_Info 98
4.3.17 Item_Location_Info 98
4.3.18 Item_Location 99
4.3.19 Node_Alarm_Msg 99
4.3.20 Station_Alarm_Msg 99
4.3.21 AddTimerAck_Info 100
4.3.22 TimerAck_Msg 100
4.3.23 Read_Config 100
4.3.24 Can_Set_Config 106
4.3.25 Servicename_And_Status_List 108
4.3.26 System_Service_List 108
4.3.27 Table 109
4.3.28 StationAck_Info 109
4.3.29 All_StationAck_Info 109
4.3.30 Station_Info_Store 110
4.3.31 StationAck_Info_Sum 110
4.3.32 NodeAck_Info 111
4.3.33 file_name_item 111
4.3.34 app_files_item 112
4.3.35 PFileConDesc 112
4.3.36 Node_List 114
4.3.37 Com_Route_Info 114
5 调度系统SDK 115
5.1 接口定义 115
5.1.1 SDK 初始化 115

5.1.2 服务注册与注销 116
5.1.3 服务查找 118
5.1.4 其它接口 120
5.2 宏定义 120
5.3 数据结构 121
5.3.1 Service_List 121
5.3.2 Location_Desc 121
5.3.3 IDDATA 122
5.3.4 Load_Info 123
6 分布式并行任务编程范例 125
6.1 概述 125
6.2 分布式并行编程框架结构示例 125
6.3 应用编程示例 126
6.3.1 单词统计 126
6.3.2 数据排序 128
7 常见问题解决 128
7.1 注意事项 128
7.2 常见问题与解答 128

1 前言

1.1 编写约定

非常感谢您使用成都天心悦高科技发展有限公司的产品，本公司将竭诚为您提供最好的服务。

本手册假定用户能够理解并使用 Linux 的基本 shell 命令。

文中出现的‘#’号表示数字有机体系统的命令行提示符。

命令格式描述中的斜体字表示应由用户填充的部分，”[ ]”表示命令中可选的命令参数。

为了阅读方便，文档以灰底黑框的形式呈现某些重要的配置操作。不过，由于数字有机体系统和 Windows 采用不同的字符集和文本规范，请不要直接拷贝文档中的命令行或者配置行到数字有机体系统中，请重新输入。

因软件更新，本手册可能包含技术上不准确的地方或文字错误。

本手册的内容将做定期的更新，恕不另行通知；更新的内容将会在本手册的新版本中加入。

本公司随时会改进或更新本手册中描述的产品或程序。

1.2 如何使用本手册

本手册的阅读对象为数字有机体工作平台的开发人员。请一定要先阅读第三章了解如何使用各种接口函数，然后再分别阅读自己想要的部分。本文第四章介绍数字有机体工作平台的系统接口，这些接口主要给系统管理程序开发人员使用。第五章介绍系统任务调度接口。应用开发人员更多关心这个部分。最后提供开发过程中遇到问题时如何处理的解决方法。

在您需要查找某个或者某类接口时，可以直接按照目录选定要使用的函数并阅读其介绍即可。

1.3 相关文档说明

数字有机体系统包括数字有机体工作平台和数字有机体工作库。本文档是数字有机体工作平台的开发手册。相关的其他文档还有：

- 有关如何配置、管理、维护和使用数字有机体工作平台请参阅《数字有机体工作平台及抗毁容灾系统用户手册》。
- 有关如何维护、管理和使用数字有机体工作库请参考《数字有机体工作库及大规模存储与管理系统用户手册》。
- 有关如何在数字有机体工作库开发应用程序，请参考《数字有机体工作库及大规模存储与管理系统开发手册》。

1.4 如何获得技术支持

在您遇到问题时，请首先联系您的产品提供商。大多数问题都可以在产品提供商的技术支持人员的帮助下得以解决。

您也可以通过产品提供商致电本公司的技术服务热线：028-83318559，获得电话技术支持。您还可以发送邮件，邮件地址是：tianxinyue@126.com。如果您确实需要本公司提供上门服务，本公司将竭诚为您服务。

2 简介

2.1 数字有机体系统简介

数字有机体系统（英文名称为 Digital Organism System，缩写为 DOS）是在刘心松教授带领下，由成都天心悦高科技发展有限公司的研发人员前后千余人次，经过三十多年的技术积累，研发成功的基础系统。

研发这种系统的原始宗旨是向生物特别是人类个体和群体的结构、机理和特性逼近，是一种人能化的新的系统模式。这种系统集操作系统、数据库系统、大规模存储、抗毁容灾、高伸缩、高智能、高灵活、自搜索、自传播、自复制、自修复、自重构、自适应、系统间的兼容性、群体间的协作性、对资源的动态管理调度合理配置、大小新旧机器混合使用等特性为一体，是一个整体解决方案，是面向所有应用的统一的（应用）系统平台。

数字有机体系统主要由数字有机体工作平台、数字有机体抗毁容灾系统、数字有机体工作库、数字有机体大规模存储与管理系统、数字有机体安全系统组成。这是从底层作起的一个一体化平台，可以在此平台上开发任何应用，形成任何应用系统。例如现在已有的应用系统就有数字有机体流媒体系统、数字有机体监控系统、数字有机体会议系统、数字有机体网关、数字有机体管理系统、数字有机体控申系统、数字有机体侦查指挥系统等。

本文有时将数字有机体工作平台及抗毁容灾系统，数字有机体工作库及大规模存储与管理系统和数字有机体安全系统统称为数字有机体系统。数字有机体工作平台及抗毁容灾系统含盖常规操作系统但远高于常规操作系统，是一个在 Linux 之上的、面向很多应用的、统一的、人能化的应用系统平台。数字有机体工作库及大规模存储与管理系统含盖常规数据库系统但远高于常规数据库系统，是一个在 Mysql 之上的、面向很多应用的、统一的、人能化的应用数据平台。

有时将数字有机体工作平台及抗毁容灾系统简称为数字有机体工作平台甚至工作平台。

有时将数字有机体工作库及大规模存储与管理系统简称为数字有机体工作库甚至工作库。

2.2 接口概述

数字有机体工作平台编程接口（以下简称系统编程接口）和数字有机体调度系统 SDK（以下简称调度系统 SDK）属于数字有机体工作平台的开发接口。其中，系统编程接口适用于数字有机体工作平台，调度系统 SDK 适用于目前通用的 Linux 和 Windows 平台，我们提供 Linux 和 Windows 两个版本的 SDK。调度的不同版本 SDK 的组件因平台不同而不一致，但是具有相同的开发接口，因此在以下的接口说明时，不再区分 Linux 和 Windows 两个版本。它们各自包含的组件如下：

表 2-1 系统编程接口

开发库名称	组件名称	组件类型

系统管理	dos.h	头文件
	dos_error.h	头文件
	dos_exernel_share.h	头文件
	libservice.so	动态链接库文件
存储开发库	Linux 标准文件系统的库组件	Linux 标准文件系统的头文件和链接库文件
示例程序	位于“/usr/local/share/doc/lib_exernel/”目录下	标准 c 开发的例子程序

表 2-2 Windows 下调度系统 SDK

开发库名称	组件名称	组件类型
服务调度库	dosschedule.h	头文件
	idtype.h	头文件
	SM_Client_Lib.lib	LIB 库文件
	SM_Client_Lib.dll	DLL 库文件

表 2-3 Linux 下调度系统 SDK

开发库名称	组件名称	组件类型
服务调度库	dosschedule.h	头文件
	libsched.a	静态链接库文件
	libsched.so	动态链接库文件
示例程序	位于“/usr/local/share/doclib_sched/”目录下	标准 c 开发的例子程序

注意：在数字有机体工作平台上开发系统管理及维护的程序时必须安装系统编程接口的各个组件，而在开发存储相关的应用时则只需要存储开发库；在 Linux 和 Windows 系统平台上均可以采用调度系统 SDK 进行开发，但必须安装服务调度库。

3 编程导引

3.1 系统编程

3.1.1 接口调用主要流程

数字有机体工作平台的系统编程主要流程如图 3-1 所示，接口调用前需要先做用户登录操作，然后再调用各种接口，当程序退出时将自动注销已登录用户。

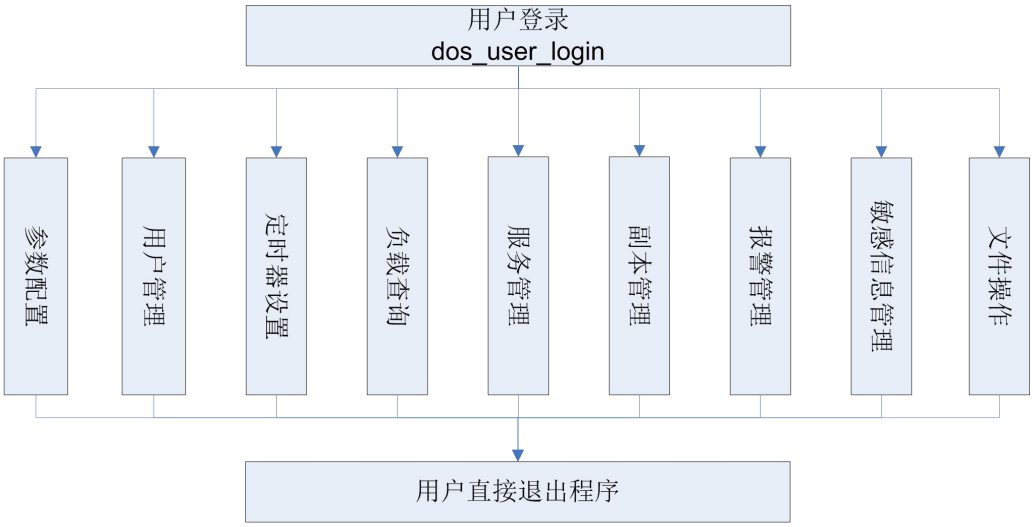

图 3-1 调用系统接口主要流程

3.1.2 参数配置流程

参数的配置如下，可以通过 get_node_config_file 先获取当前的配置参数，再调用 set_node_config_file 接口配置节点参数，重新启动系统后配置参数生效，如图 3-2 所示。

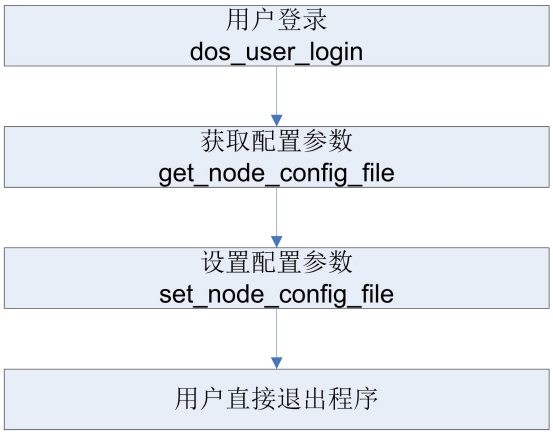

图 3-2 参数配置调用流程

3.1.3 用户管理

数字有机体工作平台的用户管理如图 3-3 所示，接口调用前需要先做用户登录操作，然后再调用各种接口，最后退出程序。登录用户需要有相应的权限才能正确执行各项操作。

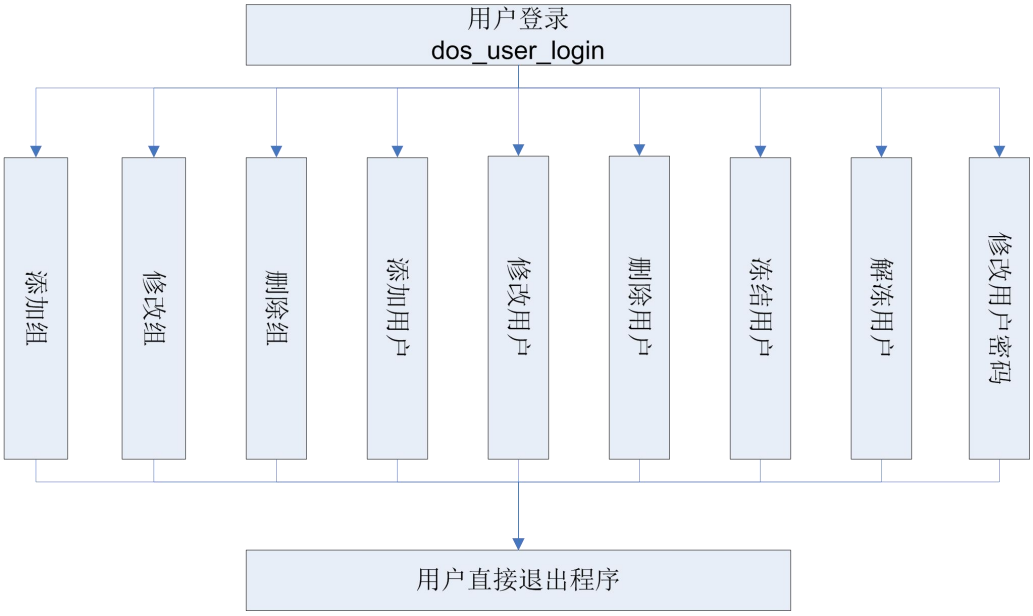

图 3-3 用户管理接口调用流程

3.1.4 定时器设置流程

定时器设置之后，系统将按照设置的时间间隔反复调用定时器，这样就可以根据用户的需求来更新相应的参数，最后调用获取信息的接口获取信息（如报警信息），如图 3-4 所示。

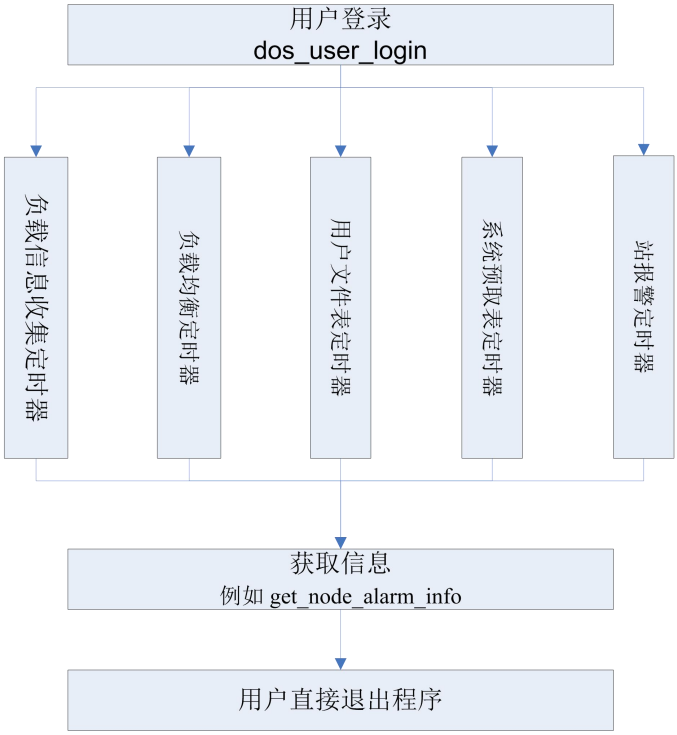

图 3-4 定时器设置流程

3.1.5 服务管理流程

服务的管理，一般体现在服务器系统添加服务节点，客户机查询服务节点，如图 3-5 所示。

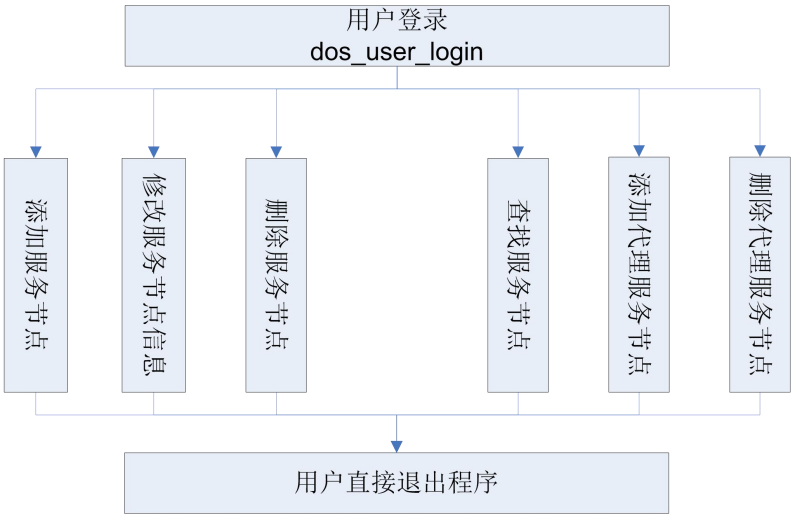

图 3-5 服务管理接口调用流程

3.1.6 文件系统操作流程

数字有机体工作平台的文件的操作和标准的文件操作是完全兼容的。它主要包括目录操作和文件操作两方面，详细内容如图 3-6 所示。

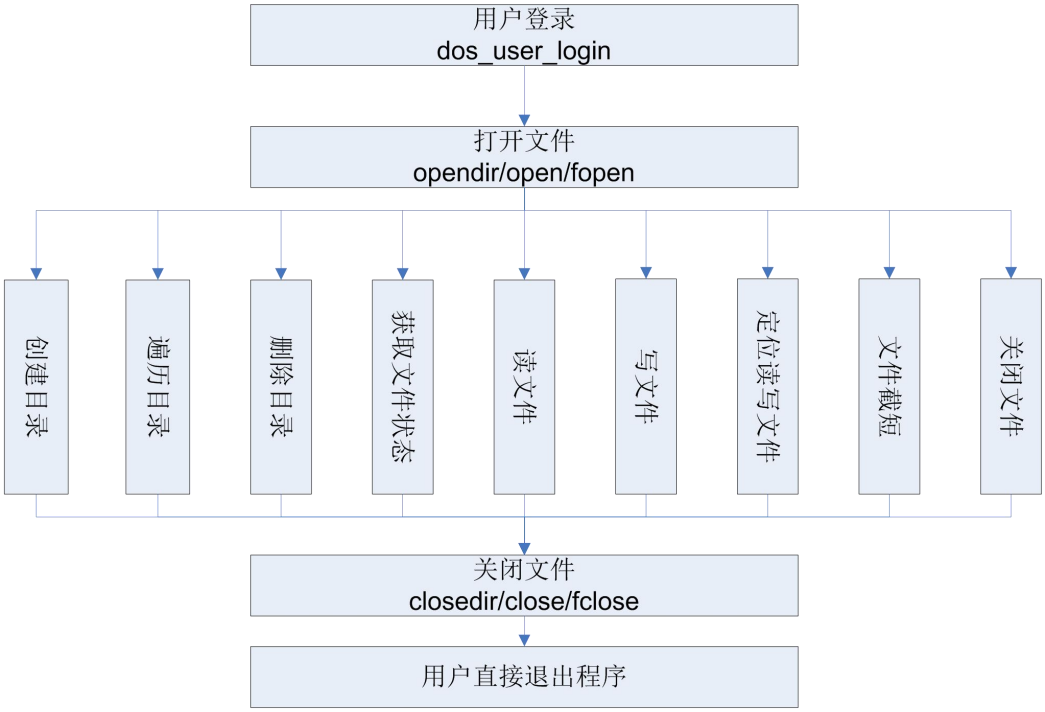

图 3-6 文件系统操作调用流程

3.2 调度系统

Vserver（Linux 版本叫 upd）是独立运行在客户机上的一个守护进程。它也可以由使用者手工启动，也可以在资源注册客户端库或者调度客户端库首次需要获得最近的服务节点等时启动。守护进程一经启动，则一直不停止，直到用户关机或者手动杀死。该程序有以下几个主要功能：

1) 接收客户端接口的注册、注销请求。进程会在内存中维护一张注册服务链表，记录该节点已注册了服务的服务进程，用于检测这些进程是否活跃；
2) 检测在本节点注册的服务的运行情况；
3) 定时获得离本节点最近的服务器节点列表；
4) 检测站内其他注册客户端的活动状态。

Vserver（Linux 版本叫 upd）是由调度 SDK 开发的一个经典应用程序，直接使用它，可以加快用户的应用开发。以下将介绍调度系统开发的基本流程。

3.2.1 SDK 接口调用的主要流程

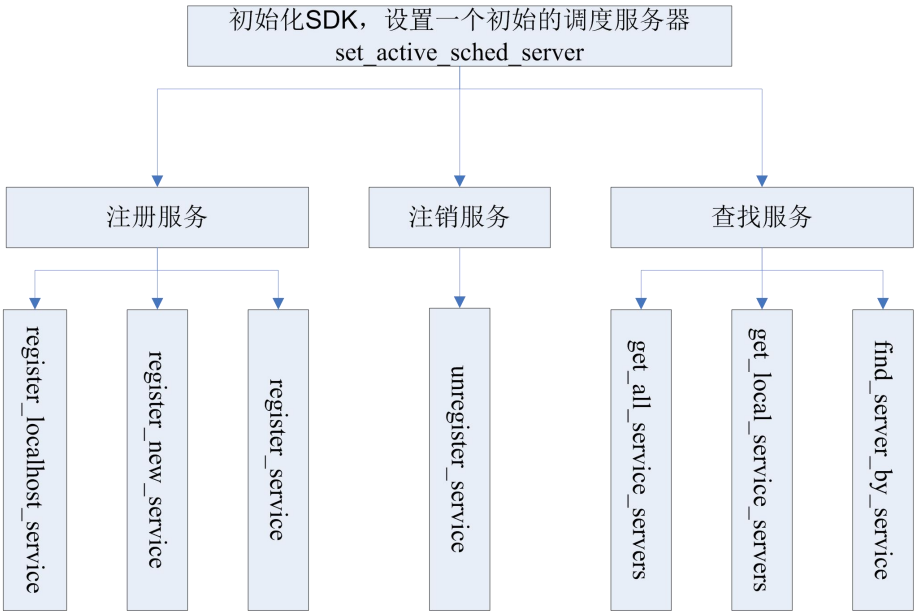

图 3-7 调度 SDK 接口调用关系

调度系统 SDK 开发时，先要设置一个初始的调度服务器地址，设置成功后，就可以调用注册服务、注销服务和查找服务三类接口，如图 3-7 所示。

3.3 编译说明

使用数字有机体系统的 CAPI 时，如果使用了“系统编程接口”，在编译程序时，务必要引用“libservice.so”库，引用的方式是在 gcc/g++中添加“-lservice”参数。具体示例可以参考目录“/usr/local/share/doc/lib_exernel”下的示例。

使用数字有机体系统的 CAPI 时，如果使用了“调度系统 SDK”，在编译程序时，务必要引用“libsched.so”和“libservice.so”库，引用的方式是在 gcc/g++中添加“-lsched -lservice”参数。具体示例可以参考目录“/usr/local/share/doc/lib_sched/”下的示例。

4 系统编程接口

4.1接口定义

4.1.1 共享存储管理

本章节介绍系统共享存储管理接口，所谓“共享存储”是指平台中所有主机都能进行存取的存储空间，每个节点对该存储空间的操作都好像是在对本地的存储空间进行操作一样，而实际上该存储空间的物理位置分布可能相当分散。

4.1.1.1 dos_umount

卸载文件系统。

```
int dos_umount(
    char *local
);
```

参数
[in] local
挂载目录的绝对路径

返回值
0：输出成功
-1：输出失败

备注
请注意当文件系统正处于使用状态时，不能进行卸载操作，必须等工作完成后才能进行卸除。例如，如果在当前文件系统中有打开的文件或某些进程正工作在该文件系统的某个目录下时， 或者是它上面的一个交换文件正在使用。

相关接口
dos_mount

4.1.1.2 dos_mount

挂载文件系统。

```
int dos_mount(
    char *global,
    char *local,
    char *other_options
);
```

参数

[in] global
全局根目录
[in] local
本地挂载点
[in]other_options
加密服务；1 加密， 0 不加密。

返回值

0：输出成功
-1：输出失败

备注

无

相关接口

dos_umount

4.1.1.3 add_share_dir

输出一个本地目录作为全局目录树的一个目录节点。

```
int add_share_dir(
      char *exportname
);
```

参数

[in] exportname
共享目录的绝对路径

返回值

0：输出成功
其他：输出失败

备注

无

相关接口

delete_share_dir

4.1.1.4 delete_share_dir

取消本地输出的共享空间。

```
int delete_share_dir(
      char *exportname
```

);

参数
[in] exportname
共享目录的绝对路径

返回值
0：输出成功
其他：输出失败

备注
无

相关接口
add_share_dir

4.1.2 用户组管理

本节介绍的接口用于数字有机体工作平台用户组的管理，包括用户组的信息查询、增添、修改以及删除。

4.1.2.1 register_group

注册组。

int register_group(
    p_group_info g_info
);

参数
[in] g_info
组信息

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
modify_group_info
unregister_group

4.1.2.2 unregister_group

注销组。

```
int unregister_group(
        char *g_name
);
```

参数

[in] g_name
组名

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

register_group
set_group_info

4.1.2.3 modify_group_info

修改组信息。

```
int modify_group_info (
        p_group_info new_ginfo
);
```

参数

[in] new_ginfo
组基本信息

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

register_group
unregister_group

4.1.2.4 get_group_info

获取组信息。

```
int get_group_info(
    char *g_name,
    p_group_info *out_g_info
);
```

参数

[in] g_name
组名
[out] out_g_info
获取的组信息

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

get_all_group_info

4.1.2.5 get_all_group_info

获取所有的组信息。

```
int get_all_group_info(
    pg_list_info *g_list_info
);
```

参数

[out] g_list_info
指向所有的组信息的指针

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

get_group_info

4.1.2.6 gid_to_gname

实现组 id 到组名的转换。

```
int gid_to_gname(
```

```
    unsigned int g_id,
    char *g_name
);
```

参数

[in] g_id
组 id。
[out] g_name
组名

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

gname_to_gid

4.1.2.7 gname_to_gid

实现组名到组 id 的转换。

```
int gname_to_gid(
    char *g_name,
    unsigned int *g_id
);
```

参数

[in] g_name
组名
[out] g_id
组 id

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

gid_to_gname

4.1.2.8 free_group_info

释放链表的内存资源。

```
void free_group_info(
      p_group_info free_item
);
```

参数
[in] free_item
组用户信息

返回值
无

备注
无

相关接口
get_group_info

4.1.2.9 free_group_list_info

释放链表的内存资源。

```
void free_group_list_info(
      pg_list_info free_item
);
```

参数
[in] free_item
组用户的链表

返回值
无

备注
无

相关接口
get_all_group_info

4.1.3 用户管理

本节介绍的接口用于数字有机体工作平台的用户账户管理，包括用户账户的信息查询、增添、修改、删除及冻结。

4.1.3.1 dos_user_login

数字有机体工作平台的用户登录接口。

```
int dos_user_login(
        char* user_name,
        int name_len,
        char* passwd,
        int passwd _len
);
```

参数

[in] user_name
用户名
[in] name_len
用户名的字符长度
[in] passwd
密码
[in] passwd _len
密码的长度

返回值

0 表示成功，否则表示失败

备注

密码的长度最小为 6 个字节

相关接口

register_user
modify_user_basic_info
modify_user_quota_info
unregister_user
get_user_info
modify_user_auth_info
frozen_user
unfrozen_user

4.1.3.2 reload_credent_conf

当配置文件“/etc/credent.conf”更新后，使用本接口重新加载配置文件的标记到系统内核。

```
reload_credent_conf()
```

参数

无

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
dos_user_login

4.1.3.3 get_dos_uid

获取当前进程的用户 ID。

get_dos_uid()

参数
无

返回值
成功返回用户 ID，否则返回-1

备注
无

相关接口
dos_user_login

4.1.3.4 change_dos_uid

改变当前进程的用户 ID。

```
int change_dos_uid(
    char* user_name,
    int name_len
);
```

参数
[in] user_name
用户名
[in] name_len
用户名的字符长度

返回值
总是返回 0，表示成功

备注
无论用户名是否有效，总是能成功

相关接口
dos_user_login

4.1.3.5 change_dos_gid

改变当前进程的用户组 ID。

```
int change_dos_uid(
    char* group_name,
    int name_len
);
```

参数
[in] group_name
用户名
[in] name_len
用户名的字符长度

返回值
总是返回 0，表示成功

备注
无论用户组名是否有效，总是能成功

相关接口
dos_user_login

4.1.3.6 register_user

注册新用户。

```
int register_user(
    p_basic_user_info u_basic_info,
    p_user_authen_info u_auth_info,
    PuserQuota u_quota
);
```

参数
[in] u_basic_info
用户基本信息
[in] u_auth_info
用户认证信息
[in] u_quota

用户配额信息

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
dos_user_login
unregister_user
modify_user_basic_info
modify_user_quota_info
get_user_info
modify_user_auth_info
frozen_user
unfrozen_user

4.1.3.7 unregister_user

注销用户。

```
int unregister_user(
        char *p_uname
);
```

参数
[in] p_uname
用户名

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
dos_user_login
register_user
modify_user_basic_info
modify_user_quota_info
get_user_info
modify_user_auth_info
frozen_user
unfrozen_user

4.1.3.8 modify_user_basic_info

修改用户信息。

```
int modify_user_basic_info(
    p_basic_user_info new_u_info
);
```

参数

[in] new_u_info
用户基本信息

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

dos_user_login
register_user
modify_user_quota_info
unregister_user
get_user_info
modify_user_auth_info
frozen_user
unfrozen_user

4.1.3.9 modify_user_quota_info

修改用户配额信息。

```
int modify_user_quota_info(
    char *user_name,
    int64_t logic_quota,
    int64_t real_quota
);
```

参数

[in] user_name
用户名
[in] logic_quota
用户逻辑配额
[in] real_quota
用户实际配额

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
dos_user_login
register_user
modify_user_basic_info
unregister_user
get_user_info
modify_user_auth_info
frozen_user
unfrozen_user

4.1.3.10 modify_user_auth_info

设置用户认证信息。

```
int modify_user_auth_info(
        char *name,
        char *passwd,
        int passwd_len
);
```

参数
[in] name
用户名
[in] passwd
密码
[in] passwd_len
密码长度

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
dos_user_login
register_user
modify_user_basic_info

modify_user_quota_info
unregister_user
get_user_info
frozen_user
unfrozen_user

4.1.3.11 get_user_info

获取用户信息。

```
int get_user_info(
        char *u_name,
        p_basic_user_info *out_u_info,
        PuserQuota *out_u_quota
);
```

参数
[in] u_name
用户名
[out] out_u_info
获取的用户信息
[out] out_u_quota
获取的用户配额信息

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
dos_user_login
register_user
modify_user_basic_info
modify_user_quota_info
unregister_user
modify_user_auth_info
frozen_user
unfrozen_user

4.1.3.12 uid_to_uname

实现用户 id 到用户名的转换。

```
int uid_to_uname(
        unsigned int uid,
```

```
    char *name
);
```

参数

[in] uid
用户 id。
[out] u_name
用户名

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

uname_to_uid

4.1.3.13 uname_to_uid

实现用户 id 到用户名的转换。

```
int uname_to_uid(
    char *u_name,
    unsigned int *uid
);
```

参数

[in] u_name
用户名
[out] uid
用户 id

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

uid_to_uname

4.1.3.14 frozen_user

冻结用户。

```
int frozen_user (
      char *u_name
);
```

参数

[in] u_name
用户名

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

dos_user_login
register_user
modify_user_basic_info
modify_user_quota_info
unregister_user
get_user_info
modify_user_auth_info
unfrozen_user

4.1.3.15 unfrozen_user

解冻用户。

```
int unfrozen_user (
      char *u_name
);
```

参数

[in] u_name
用户名

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

dos_user_login
register_user

modify_user_basic_info
modify_user_quota_info
unregister_user
get_user_info
modify_user_auth_info
frozen_user

4.1.3.16 free_user_basic_info

释放内存资源。

```
void free_user_basic_info(
    p_basic_user_info free_item
);
```

参数
[in] free_item
用户基本信息

返回值
无

备注
无

相关接口
get_user_info
register_user

4.1.3.17 free_user_auth_info

释放内存资源。

```
void free_user_auth_info(
    p_user_authen_info free_item
);
```

参数
[in] free_item
用户认证信息

返回值
无

备注
无

相关接口
register_user

4.1.4 节点配置文件管理

本节介绍的接口用于数字有机体工作平台的配置文件（/etc/dos_exernel.cnf）的管理，包括获取、修改配置文件。

4.1.4.1 get_node_config_file

获取 node_ip 指定的节点的外核配置文件/etc/dos_exernel.cnf 中所有配置项的值。

```
int get_node_config_file(
    unsigned int node_ip,
    struct Read_Config **config
);
```

参数
[in] node_ip
节点 IP 地址
[out] config
保存节点配置项值的结构体

返回值
0：读取成功
-1：读取失败

备注
返回的配置结构 config 使用结束后需要使用 free（）释放

相关接口
set_node_config_file

4.1.4.2 set_node_config_file

用 config 指向的结构体中各配置项的值设置 node_ip 指定的节点外核配置文件 dos_exernel.cnf 中的各配置项。

```
int set_node_config_file (
    unsigned int node_ip,
    struct Can_Set_Config *config
);
```

参数
[in] node_ip

节点 IP 地址
[in] config
保存节点配置项值的结构体

返回值
0：设置成功
-1：设置失败

备注
无

相关接口
get_node_config_file

4.1.5 站点、节点查询与管理

本节介绍的接口用于数字有机体工作平台的站点、节点的查询与管理，用户可以使用这些接口来获取系统的分布情况以及站首的设置与获取。

4.1.5.1 get_all_station_id

获取系统内站总数以及各站的 ID 号, 调用者需要使用 free()函数释放 ids 指向的空间。

```
int get_all_station_id(
    INTID **ids,
    int *num
);
```

参数
[out] ids
已保存系统内所有站号（标识符）的数组
[out] num
保存系统所有站的数目

返回值
0：成功
-1：失败

备注
需要使用 free()函数释放 ids 指向的空间

相关接口
无

4.1.5.2 get_all_station_node

获取指定站内总节点数以及每个节点的 IP 地址，调用者需要使用 free()函数释放 ips 指向的空间。

```
int get_all_station_node(
    INTID station_id,
    unsigned int **ips,
    int *num
);
```

参数

[in] station_id
站号（标识符）
[out] ips
IP 数组
[out] num
IP 数组中包含的 IP 数量

返回值

0：成功
-1：失败

备注

需要使用 free()函数释放 ips 指向的空间

相关接口

无

4.1.5.3 set_localhost_as_manager

设置本机为主管理机。

```
int set_localhost_as_manager ();
```

参数

无

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

get_current_manager

4.1.5.4 get_current_manager

查询管理节点。

unsigned int get_current_manager();

参数
无

返回值
返回当前管理节点 IP

备注
无

相关接口
set_localhost_as_manager

4.1.5.5 delete_always_dead_node

删除总是出现故障的节点。

```
int delete_always_dead_node (
      unsigned int node_ip,
      INTID station_id
);
```

参数
[in] node_ip
[in] station_id

返回值
返回 0 表示成功，否则表示失败

备注
无

相关接口
无

4.1.6 站点、节点负载管理

本节介绍的接口用于数字有机体工作平台的负载管理，用户使用这些接口能有准确的获取系统内站点、节点的负载情况，从而有效的控制和使用系统。

4.1.6.1 add_collect_load_timer

对系统内所有节点添加节点负载信息收集定时器，并将添加成功的节点信息记录到 status 指向的结构体中。该定时器启动一次则收集一条本节点的负载信息；并将该条负载信息记录到负载信息结构体中，然后插入到内存中负载信息结构体链表中。

```
int add_collect_load_timer (
    unsigned int timer_value,
    struct TimerAck_Msg **status
);
```

参数

[in] timer_value
定时器时间值（单位：秒。默认值为 5，一般不宜过大或过小）
[out] status
指向定时器添加成功的节点信息记录结构体

返回值

0：添加成功
-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

delete_collect_load_timer

4.1.6.2 delete_collect_load_timer

删除系统内所有节点上的节点负载信息收集定时器，并将删除成功的节点信息记录到 status 指向的结构体中。

```
int delete_collect_load_timer (
    struct TimerAck_Msg **status
);
```

参数

[in] status
指向定时器删除成功的节点信息记录结构体

返回值

0：删除成功
-1：删除失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口
add_collect_load_timer

4.1.6.3 get_station_load

获取站 ID 为 station_id 的站负载信息。其中 info_num 取值范围为 1～50。并且该值越大收集到的负载数值越能反映该站中各节点在较长时间内的负载情况。

```
int get_station_info(
    INTID station_id,
    unsigned int info_num,
    struct StationAck_Info **station_info
);
```

参数
[in] station_id
站号（标识符）
[out] info_num
负载信息数组的数量
[out] station_info
负载信息数组

返回值
0：获取成功
-1：获取失败

备注
调用者需要调用 free()函数释放 station_info 指向的空间

相关接口
get_all_station_load
get_station_avg_load

4.1.6.4 get_station_load2

4.1.6.5 get_all_station_load

获取系统内所有站的负载信息。

```
int get_all_station_load (
    unsigned int info_num,
    struct All_StationAck_Info **all_info
);
```

参数

[in] info_num
负载信息数组的数量
[out] all_info
负载信息数组

返回值

0：获取成功
-1：获取失败

备注

调用者需要调用 free()函数释放 all_info 指向的空间

相关接口

get_station_load
get_station_avg_load

4.1.6.6 get_station_avg_load

获取指定站点的平均节点负载信息。其中 info_num 取值范围为 1～50。并且该值越大收集到的负载数值越能反映该节点在较长时间内的负载情况。

```
int get_station_avg_load (
      INTID station_id,
      unsigned int info_num,
      struct StationAck_Info_Sum **info
);
```

参数

[in] station_id
站号（标识符）
[in] info_num
负载信息数组的数量
[out] info
负载信息数组

返回值

0：获取成功
-1：获取失败

备注

调用者需要调用 free()函数释放 info 指向的空间。由于系统内每个节点上都保存了含 50 个负载信息结构体的链表，这些负载信息结构体中保存了每隔 5 秒收集一次的本节点负载值，并且表头结构体保存的负载值是最新的节点负载值。以上参数中 info_num 的数值就是指获

取链表中元素的个数，然后再取这几个元素的平均值。

相关接口
get_station_load
get_all_station_load

4.1.6.7 get_node_load

获取站 ID 号为 station_id 的站内 IP 为 node_ip 的节点负载信息。实际调用中 station_id 可以取任意整数值。

```
int get_node_load (
    unsigned int node_ip,
    INTID station_id,
    struct NodeAck_Info **node_info
);
```

参数
[in] node_ip
节点的 IP 地址
[in] station_id
站号（标识符）
[out] node_info
负载信息

返回值
0：获取成功
-1：获取失败

备注
调用者需要调用 free()函数释放 node_info 指向的空间

相关接口
get_all_node_load
get_node_avg_load

4.1.6.8 get_all_node_load

获取站内所有节点的负载信息，主要供服务查找模块调用。

```
struct load_info * get_all_node_load (
    int *num
);
```

参数
[in] num

用来保存站内节点数目，调用者只需要传入一整型变量指针即可

返回值
成功返回节点负载信息结构体指针，失败返回 NULL

备注
无

相关接口
get_node_load
get_node_avg_load

4.1.6.9 get_node_avg_load

获取指定站的指定节点的平均节点负载信息。其中 info_num 取值范围为 1～50。并且该值越大收集到的负载数值越能反映该节点在较长时间内的负载情况。

```
int get_node_avg_load (
    unsigned int node_ip,
    INTID station_id,
    unsigned int info_num,
    struct NodeAck_Info **node_info
);
```

参数
[in] node_ip
节点 IP 地址
[in] station_id
站号（标识符）
[out] info_num
负载数组的数量
[out] node_info
负载数组

返回值
0：获取成功
-1：获取失败

备注
调用者需要调用 free()函数释放 node_info 指向的空间

相关接口
get_all_node_load
get_node_load

4.1.7 站点、节点报警管理

本章节介绍报警管理接口，用户使用这些接口能有效地控制报警信息的收集时间间隔，并能获取到报警信息。

4.1.7.1 add_node_alarm_info_timer

添加节点报警定时器。在节点 node_ip 上将定时器时间值设为 timer_value 秒；并将添加成功与否的信息记录到 status 指向的结构体中。

```
int add_node_alarm_info_timer(
    unsigned int node_ip,
    unsigned int timer_value,
    struct AddTimerAck_Info **status
);
```

参数

[in] node_ip
节点 IP 地址
[in] timer_value
定时器时间值（单位：秒）
[out] status
指向定时器添加成功与否的信息记录结构体

返回值

0：添加成功
-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

get_node_alarm_info
delete_node_alarm_info_timer
get_station_alarm_info
add_station_alarm_info_timer
delete_station_alarm_info_timer

4.1.7.2 delete_node_alarm_info_timer

删除节点 node_ip 上的节点报警定时器。

```
int delete_node_alarm_info_timer(
    unsigned int node_ip,
    struct AddTimerAck_Info **status
);
```

参数

[in] node_ip
节点 IP 地址
[in] status
指向定时器删除成功与否的信息记录结构体

返回值

0：删除成功
-1：删除失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

get_node_alarm_info
add_node_alarm_info_timer
get_station_alarm_info
add_station_alarm_info_timer
delete_station_alarm_info_timer

4.1.7.3 get_node_alarm_info

当系统内任一节点负载超过设置的报警线，就会发送报警信息，本接口用于获取报警信息。

```
int get_node_alarm_info(
        unsigned int node_ip,
        struct Node_Alarm_Msg **alarm,
        unsigned int begin_pos,
        unsigned int max_count
);
```

参数

[in] node_ip
节点的 IP 地址
[out] alarm
报警信息结构体
[in]begin_pos
符合条件的记录的起始位置
[in]max_count
最多获取的记录条数

返回值

0：有报警
-1：没有报警

备注

此函数返回的报警信息。调用者需要使用 free()函数释放 alarm 指向的空间

相关接口

add_node_alarm_info_timer
delete_node_alarm_info_timer
get_station_alarm_info
add_station_alarm_info_timer
delete_station_alarm_info_timer

4.1.7.4 add_station_alarm_info_timer

向站 station_id 添加站报警定时器。将定时器时间值定为 timer_value 秒，并将添加成功与否的信息记录到 status 指向的结构体中。

```
int add_station_alarm_info_timer(
    INTID station_id,
    unsigned int timer_value,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] timer_value
定时器时间值（单位：秒）
[out] status
指向定时器添加成功与否的信息记录结构体

返回值

0：添加成功
-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间。

相关接口

get_node_alarm_info
add_node_alarm_info_timer
delete_node_alarm_info_timer
get_station_alarm_info
delete_station_alarm_info_timer

4.1.7.5 delete_station_alarm_info_timer

删除 station_id 站上的站报警定时器。

```
int delete_station_alarm_info_timer(
    INTID station_id,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] status
指向定时器删除成功与否的信息记录结构体

返回值

0：删除成功
-1：删除失败

备注

调用者需要调用 free()函数释放 status 指向的空间。

相关接口

get_node_alarm_info
add_node_alarm_info_timer
delete_node_alarm_info_timer
get_station_alarm_info
add_station_alarm_info_timer

4.1.7.6 get_station_alarm_info

当系统内任一站点负载超过设置的报警线，就会发送报警信息，本接口用于获取站点的报警信息。

```
int get_station_alarm_info(
    INTID station_id,
    struct Station_Alarm_Msg **alarm,
    unsigned int begin_pos,
    unsigned int max_count
);
```

参数

[in] station_id
站号（标识符）
[out] alarm

报警信息结构体
[in]begin_pos
符合条件的记录的起始位置
[in]max_count
最多获取的记录条数

返回值
0：有报警
-1：没有报警

备注
此函数返回的报警信息。调用者需要使用 free()函数释放 alarm 指向的空间

相关接口
get_node_alarm_info
add_node_alarm_info_timer
delete_node_alarm_info_timer
add_station_alarm_info_timer
delete_station_alarm_info_timer

4.1.8 服务管理

本节介绍的接口用于数字有机体工作平台的服务管理，包括添加、删除、更新、启动以及停止等功能。服务管理指记录每台服务器可以提供的服务的相关信息。数字有机体工作平台收集、记录和管理系统中的服务信息，为应用程序提供服务查询和服务快速定位服务。通过服务注册机制向数字有机体工作平台注册每台服务器可以提供的服务的信息。数字有机体工作平台将监视每项服务的运行情况，在发现提供服务的服务器故障，或者服务程序死亡时，将自动注销对应服务信息。从而使得应用程序总能获得可用的服务。

4.1.8.1 add_service

增加一个新服务信息。

```
int add_service (
      char *service_name,
      SDesc *service_info
);
```

参数
[in] service_name
服务名
[in] service_info
注册服务节点的信息

返回值

SUCCESS：成功。
SERVICE _INFO_EXIST：服务信息已存在。
SERVICE_NODE_ERROR：服务节点信息错误。
DATABASE_ERROR：文件属性库操作失败。
INPUT_ERROR：未知错误（输入参数不合法）。
MALLOC_ERROR：内存分配错误

备注

无

相关接口

delete_service
add_service_node
delete_service_node
modify_service_node
modify_service
get_node_by_service
startup_node_service
shutdown_node_service

4.1.8.2 delete_service

注销服务。

```
int delete_service (
      char *service_name,
      int node_type
);
```

参数

[in] service_name
服务名
node_type：节点类型，CLIENT_NODE 代表客户节点，SERVER_NODE 代表服务器节点

返回值

SUCCESS：成功
SERVICE_NODE_ERROR：服务节点信息错误
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注

无

相关接口
add_service
add_service_node
delete_service_node
modify_service_node
modify_service
get_node_by_service
startup_node_service
shutdown_node_service

4.1.8.3 modify_service

更新某服务所属的基本信息。

```
int modify_service (
      char *service_name,
      SDesc *service_info
);
```

参数
[in] service_name
服务名
[in] service_info
服务信息

返回值
SUCCESS：成功
SERVICE_INFO_NOT_EXIST：要更新的服务信息不存在
SERVICE_NODE_ERROR：服务节点信息错误。
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注
无

相关接口
add_service
delete_service
add_service_node
delete_service_node
modify_service_node
get_node_by_service
startup_node_service
shutdown_node_service

4.1.8.4 add_service_node

注册节点服务。

```
int add_service_node(
    char *service_name,
    RLDesc *node_info
);
```

参数

[in] service_name
服务名
[in] node_info
节点信息

返回值

SUCCESS：成功
SERVICE_NODE_INFO_EXIST：服务节点信息已存在
SERVICE_NODE_ERROR：服务节点信息错误
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注

无

相关接口

delete_service_node
delete_service
add_service
modify_service_node
modify_service
get_node_by_service
startup_node_service
shutdown_node_service

4.1.8.5 delete_service_node

注销节点服务。

```
int delete_service_node(
    char *service_name,
    RLDesc *node_info,
    int node_type
);
```

参数

[in] service_name
服务名
[in] node_info
节点服务信息
node_type：节点类型，CLIENT_NODE 代表客户节点，SERVER_NODE 代表服务器节点

返回值

SUCCESS：成功
SERVICE_NODE_ERROR：服务节点信息错误
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注

无

相关接口

add_service
delete_service
add_service_node
modify_service_node
modify_service
get_node_by_service
startup_node_service
shutdown_node_service

4.1.8.6 modify_service_node

更新某服务节点所属的某些信息。

```
int modify_service_node (
      char *service_name,
      RLDesc *node_info
);
```

参数

[in] service_name
服务名
[in] node_info
服务描述信息结构体

返回值

SUCCESS：成功

SERVICE_NODE_INFO_NOT_EXIST：服务节点信息不存在
SERVICE_NODE_ERROR：服务节点信息错误
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注
无

相关接口
add_service
delete_service
add_service_node
delete_service_node
modify_service
get_node_by_service
startup_node_service
shutdown_node_service

4.1.8.7 get_node_by_service

查找资源所在服务节点。

```
int get_node_by_service (
        char *service_name,
        struct Service_Node_Info **info,
        char scale,
        int only_server,
        char *route_info
);
```

参数
[in] service_name
服务名
[out] info
查找返回的服务节点信息
[in] scale
查找范围
[in] only_server
提供该服务的节点类型；0：客户和服务；1：服务器节点
[out] route_info
客户节点到接收该消息的服务器节点的路由信息

返回值
查找到的服务数

备注
无

相关接口
add_service
delete_service
add_service_node
delete_service_node
modify_service_node
modify_service
startup_node_service
shutdown_node_service

4.1.8.8 startup_node_service

启动 IP 地址为 node_ip 节点上服务名为 service_name 的服务。

```
int startup_node_service (
        unsigned int node_ip ,
        char *service_name
);
```

参数
[in] node_ip
节点 IP 地址
[in] service_name
服务名

返回值
1：成功
-1：失败

备注
无

相关接口
add_service
delete_service
add_service_node
delete_service_node
modify_service_node
modify_service
get_node_by_service
shutdown_node_service

4.1.8.9 shutdown_node_service

停止 IP 地址为 node_ip 节点上服务名为 service_name 的服务。

```
int shutdown_node_service (
    unsigned int node_ip ,
    char *service_name
);
```

参数

[in] node_ip
节点 IP 地址
[in] service_name
服务名

返回值

1：成功
-1：失败

备注

无

相关接口

add_service
delete_service
add_service_node
delete_service_node
modify_service_node
modify_service
get_node_by_service
startup_node_service

4.1.8.10 get_all_service_list

获取系统内所有服务的服务状态。

```
struct System_Service_List * get_all_service_list (
    void
);
```

参数

无

返回值

成功返回指向服务状态记录链表的指针，失败返回 NULL

备注
调用者需要使用 free_service_list()函数释放服务状态记录链表的空间

相关接口
free_service_list

4.1.8.11 free_service_list

释放服务状态记录链表的空间。

```
int free_service_list(
    struct System_Service_List *list
);
```

参数
[in] list
服务状态链表

返回值
0：指针不为空并且释放空间成功
-1：指针为空

备注
无

相关接口
get_all_service_list

4.1.8.12 add_service_assistant_node

添加代理服务节点信息。

```
int add_service_assistant_node (
    unsigned int private_ip,
    unsigned int public_ip
);
```

参数
[in] private_ip
私网 IP 地址
[in] public_ip
公网 IP 地址

返回值

SUCCESS：成功
SERVICE_NODE_INFO_EXIST：服务节点信息已存在
SERVICE_NODE_ERROR：服务节点信息错误
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注
无

相关接口
delete_service_assistant_node
get_service_assistant_node

4.1.8.13 delete_service_assistant_node

删除代理服务节点信息。

```
int delete_service_assistant_node (
    unsigned int private_ip,
    unsigned int public_ip
);
```

参数
[in] private_ip
私网 IP 地址
[in] public_ip
公网 IP 地址

返回值
SUCCESS：成功
SERVICE_NODE_ERROR：服务节点信息错误
DATABASE_ERROR：文件属性库操作失败
INPUT_ERROR：未知错误（输入参数不合法）
MALLOC_ERROR：内存分配错误

备注
无

相关接口
add_service_assistant_node
get_service_assistant_node

4.1.8.14 get_service_assistant_node

获得代理服务节信息。

```
struct Node_List * get_service_assistant_node (
        unsigned int private_ip,
        unsigned int public_ip,
        int radius,
        struct Com_Route_Info *in_route
);
```

参数

[in] private_ip
私网 IP 地址
[in] public_ip
公网 IP 地址
[in] radius
节点的跳数
[in] in_route
节点到服务器的路由信息结构体

返回值

成功返回 SA 节点信息结构体 Node_List 指针，失败返回 NULL

备注

无

相关接口

add_service_assistant_node
delete_service_assistant_node

4.1.8.15 register_route_info

将路由信息放入路由信息库中。

```
int register_route_info(
        struct Route_Info *info
);
```

参数

[in] info
指向路由信息链表的指针，其中路由信息链表可以包含多个节点信息，也可以只有一个节点信息

返回值

0：成功
-1：失败

备注
无

相关接口
unregister_route_info

4.1.8.16 unregister_route_info

在路由信息库中删除对应的路由信息。

```
int unregister_route_info(
        unsigned int node,
        char type
);
```

参数
[in] node
客户端 ip 或某站站 ID
[in] type
0 代表客户端到服务端的路由，1 代表服务端到服务端的路由

返回值
0：成功
-1：失败

备注
无

相关接口
register_route_info

4.1.9 虚拟服务管理

4.1.9.1 get_service_connection

获取虚拟服务的连接信息。

```
struct ip_vs_get_dests *get_service_connection(
        unsigned int v_id,
        unsigned int s_ip
);
```

参数

[in] v_id
虚拟服务的编号，代表一个服务
[in] v_ip
服务的 IP 地址

返回值
虚拟服务的连接信息

备注
无

相关接口
无

4.1.10 副本（块）的位置与属性管理

本章节介绍文件和块的副本控制，与属性控制接口。通过使用这些接口，用户能得到文件和块的副本分布位置，并能编辑修改副本的放置规则。

4.1.10.1 get_file_block_location

根据文件名获得文件的存储位置信息。调用者需要调用 free_file_block_location()函数释放 list 指向的链表空间。

```
int get_file_block_location(
    char *globalName,
    struct BlockOffStation **blockOffStation
);
```

参数
[in] globalName
文件路径
[out] blockOffStation
调用成功时返回值是指向 struct BlockOffStation 的指针，其中包含了该文件所有数据块在系统中的分布信息。

返回值
负数：调用失败
0：调用成功
备注
文件路径可以是相对路径，也可以是绝对路径，但是文件必须在数字有机体工作平台下

相关接口
add_file_block_to_station
delete_file_block_from_station
add_file_block_to_node

delete_file_block_from_node

4.1.10.2 free_file_block_location

释放由 get_file_block_location 函数获取的文件数据块分布信息结构的空间。

```
void free_file_block_location(
    struct BlockInStation *blockInStation
);
```

参数

[in] blockInStation
文件数据块分布信息结构

返回值

无

备注

无

相关接口

get_file_block_location
add_file_block_to_station
delete_file_block_from_station
add_file_block_to_node
delete_file_block_from_node

4.1.10.3 add_file_block_to_station

向一个站添加一个文件的块。

```
int add_file_block_to_station(
    char *file_name,
    uint64_t blockNum
    INTID station_id
);
```

参数

[in] file_name
文件的路径
[in] blockNum
文件块号
[in] station_id
站号（标识符）

返回值

0：成功
其他：失败

备注
无

相关接口
get_file_block_location
free_file_block_location
delete_file_block_from_station
add_file_block_to_node
delete_file_block_from_node

4.1.10.4 delete_file_block_from_station

从一个站删除一个文件的副本。

```
int delete_file_block_from_station(
        char *file_name,
        uint64_t blocknum,
        INTID station_id
);
```

参数
[in] file_name
文件的路径
[in] blockNum
文件块号
[in] station_id
站号（标识符）

返回值
0：成功
其他：失败

备注
无

相关接口
get_file_block_location
free_file_block_location
add_file_block_to_station
add_file_block_to_node
delete_file_block_from_node

4.1.10.5 add_file_block_to_node

向一个节点添加一个文件的块。

```
int add_file_block_to_node(
    char *file_name,
    uint64_t blockNum
    unsigned int ip
);
```

参数

[in] file_name
要添加副本文件的路径
[in] blockNum
文件块号
[in] ip
要添加副本的节点 IP 地址，必须是网络字节序

返回值

0：成功
其他：失败

备注

ip 必须是网络字节序

相关接口

get_file_block_location
free_file_block_location
add_file_block_to_station
delete_file_block_from_station
delete_file_block_from_node

4.1.10.6 delete_file_block_from_node

从一个节点删除一个文件的副本。

```
int delete_file_block_from_node(
    char *file_name,
    uint64_t blockNum,
    unsigned int ip
);
```

参数

[in] file_name
要删除的副本文件的路径

[in] blockNum
文件块号
[in] ip
要删除的副本文件所在的节点 IP 地址，必须是网络字节序

返回值
0：成功
其他：失败

备注
ip 必须是网络字节序

相关接口
get_file_block_location
free_file_block_location
add_file_block_to_station
delete_file_block_from_station
add_file_block_to_node
delete_file_block_from_node

4.1.10.7 set_file_block_size

设置文件的分块大小，以字节为单位

```
int set_file_block_size(
      char *resName,
      uint64_t blockSize
);
```

参数
[in]resName
文件路径名，必须是目录
[in]blockSize
指定的分块大小

返回值
0：成功
负数：失败

备注
系统内新建文件的分块大小继承自其父目录，设置文件分块大小的时候给出的文件路径名也必须是一个目录且只有当该目录是一个空目录时才能进行分块大小的设置。设置成功后该目录及其子目录下所有新建的文件分块大小都是设置时指定的值。

相关接口

get_file_block_size

4.1.10.8 get_file_block_size

获取文件的分块大小，以字节为单位

```
int get_file_block_size(
    char *resName,
    uint64_t *blockSize
);
```

参数

[in]resName
要查询的文件路径
[out]blockSize
调用成功时包含文件分块大小的值

返回值

0：成功
负数：失败

备注

无

相关接口

set_file_block_size

4.1.10.9 get_file_options

获取文件控制描述符。

```
int get_file_options(
    char *file_name,
    PFileConDesc *out_file_desc
);
```

参数

[in] file_name
文件名
[out] out_file_desc
文件控制描述结构

返回值

成功返回 0，否则返回-1

备注

无

相关接口
set_file_options

4.1.10.10 set_file_options

设置文件控制描述符。

```
int set_file_options(
    char *file_name,
    PFileConDesc file_desc
);
```

参数
[in] file_name
文件名
[in] file_desc
文件控制描述结构

返回值
成功返回 0，否则返回-1

备注
无

相关接口
get_file_options

4.1.10.11 set_dir_crypto_flag

设置目录的加密属性。

```
int set_dir_crypto_flag(
    const char *gdirName,
    uint8_t flag
);
```

参数：

[in]gdirName
要设置的目录名，必须是空目录；
[in] flag
要设置的值，0 表示不加密，1 表示加密。

返回值：

成功返回 0， 否则返回错误码。

备注：

数字有机体文件系统加密服务的规则是：

1）数字有机体文件系统的根目录，即/dpfs 目录初始时是非加密的，如果需要，在根目录空时可以修改其加密属性。

2）当一个目录为空时，可以改变目录的加密属性。但是目录非空时不能修改。

3）目录下的文件继承父目录的加密属性，即文件所在目录的加密属性。新创建的文件的加密属性就是父目录的加密属性。因此，加密目录下的文件都是加密的，反之则是未加密的。

4）目录下的子目录继承父目录的加密属性，即目录所在目录的加密属性，但是目录为空时可以修改加密属性。因此，可能出现加密目录下又有非加密子目录的情况。当然，非加密目录的子目录也可以是加密目录。

5）文件的加密属性继承则父目录，而且不能更改。即无法将加密文件改为不加密的，也不能就非加密文件改为加密文件。

4.1.11 系统故障后修复

本章节介绍系统故障后的修复接口，当文件异常时，用户可以使用这些接口来修复系统，使其变得正常。需要特别说明的是，以下接口返回成功，仅仅说明成功地发起了修复操作，系统修复完成与否，以及花销的时间与系统内文件的数目是息息相关的。

4.1.11.1 check_dir_dentry

对输入的全局路径名下的目录项逐个核对其资源描述信息，并作资源的目录项和资源描述信息的一致性检查。

```
int check_dir_dentry(
    char *path
);
```

参数

[in] path

数字有机体工作平台下的绝对路径目录

返回值

0：成功完成一致性检查及处理

其它：一致性检查失败

备注

此接口调用开销较大，等待的时间较长

相关接口
check_ridi_dentry

4.1.11.2 check_ridi_dentry

检查 ridi 资源描述信息和文件系统内目录内容是否匹配，如果不匹配，则在文件系统目录中写入相应的描述信息，否则直接返回成功。

```
int check_ridi_dentry();
```

参数
无

返回值
0：检查成功
-1：内存分配错误
-5：清除无效的 ridi 信息错误

备注
此接口调用开销较大，等待的时间较长

相关接口
check_dir_dentry
check_file_to_ridi

4.1.11.3 check_file_to_ridi

检查一个站的文件系统内目录内容是否与 ridi 资源描述信息匹配，如果不匹配，则在文件系统目录中写入相应的描述信息，否则直接返回成功。

```
int check_file_to_ridi(
    INTID station_id
);
```

参数
[in] station_id
站号（标识符）

返回值
成功返回 0，否则返回-1

备注
无

相关接口
check_dir_dentry

check_ridi_dentry

4.1.12 磁盘空间均衡管理

本章节介绍磁盘空间的预处理接口。

4.1.12.1 add_poise_info_timer

向站 ID 为 station_id 的站添加站负载均衡定时器，该定时器每隔 timer_value 秒运行一次。

```
int add_poise_info_timer(
    INTID station_id,
    unsigned int timer_value,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] timer_value
定时器时间值(单位：秒)
[out] status
指向定时器添加成功的站内节点信息记录结构体

返回值

0：添加成功
-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

delete_poise_info_timer

4.1.12.2 delete_poise_info_timer

删除站 ID 为 station_id 的负载均衡定时器，删除的定时器是由函数 add_poise_info_timer（）添加。

```
int delete_poise_info_timer(
    INTID station_id,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id

站号（标识符）
[in] status
指向定时器删除成功的站内节点信息记录结构体

返回值
0：删除成功
-1：删除失败

备注
调用者需要调用 free()函数释放 status 指向的空间

相关接口
add_poise_info_timer

4.1.12.3 add_distance_poise_timer

向站 ID 为 station_id 的站添加站负载均衡定时器，该定时器每隔 day 天 timer_value 秒运行一次。

```
int add_distance_poise_timer(
    INTID station_id,
    unsigned int timer_value,
    int day,
    struct TimerAck_Msg **status
);
```

参数
[in] station_id
站号（标识符）
[in] timer_value
定时器时间值(单位：秒)
[in] day
定时器时间值（单位：天）
[out] status
指向定时器添加成功的站内节点信息记录结构体

返回值
0：添加成功
-1：添加失败

备注
调用者需要调用 free()函数释放 status 指向的空间

相关接口
delete_distance_poise_info_timer

4.1.12.4 delete_distance_poise_info_timer

删除站 ID 为 station_id 的负载均衡定时器。删除的定时器是由函数 add_distance_poise_timer（）添加。

```
int delete_distance_poise_info_timer(
    INTID station_id,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] status
指向定时器删除成功的站内节点信息记录结构体

返回值

0：删除成功
-1：删除失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

add_distance_poise_timer

4.1.12.5 add_space_info_timer

向站 ID 为 station_id 的站添加用户文件表定时器，该定时器每隔 timer_value 秒运行一次。

```
int add_space_info_timer(
    INTID station_id,
    unsigned int timer_value,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] timer_value
定时器时间值(单位：秒)
[out] status
指向定时器添加成功的站内节点信息记录结构体

返回值

0：添加成功
-1：添加失败

备注
调用者需要调用 free()函数释放 status 指向的空间。

相关接口
delete_space_info_timer

4.1.12.6 delete_space_info_timer

删除站 ID 为 station_id 的用户文件表定时器，删除的定时器是由函数 add_space_info_timer（）添加。

```
int delete_space_info_timer(
    unsigned int station_id,
    struct TimerAck_Msg **status
);
```

参数
[in] station_id
站号（标识符）
[in] status
指向定时器删除成功的站内节点信息记录结构体

返回值
0：删除成功
-1：删除失败

备注
调用者需要调用 free()函数释放 status 指向的空间。

相关接口
add_space_info_timer

4.1.12.7 add_distance_space_timer

向站 ID 为 station_id 的站添加用户文件表定时器，该定时器每隔 day 天 timer_value 秒运行一次。

```
int add_distance_space_timer(
    INTID station_id,
    unsigned int timer_value,
    int day,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] timer_value
定时器时间值(单位：秒)
[in] day
定时器时间值（单位：24 天）
[out] status
指向定时器添加成功的站内节点信息记录结构体

返回值

0：添加成功
-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

delete_distance_space_info_timer

4.1.12.8 delete_distance_space_info_timer

删除站 ID 为 station_id 的用户文件表定时器。删除的定时器是由函数 add_distance_space_timer（）添加。

```
int delete_distance_space_info_timer(
    INTID station_id,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] status
指向定时器删除成功的站内节点信息记录结构体

返回值

0：删除成功
-1：删除失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

add_distance_space_timer

4.1.12.9 add_fore_info_timer

向站 ID 为 station_id 的站添加系统预取表定时器，该定时器每隔 timer_value 秒运行一次。

```
int add_fore_info_timer(
    INTID station_id,
    unsigned int timer_value,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] timer_value
定时器时间值(单位：秒)
[out] status
指向定时器添加成功的站内节点信息记录结构体

返回值

0：添加成功
-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

delete_fore_info_timer

4.1.12.10 delete_fore_info_timer

删除站 ID 为 station_id 的系统预取表定时器，删除的定时器是由函数 Add_Fore_Info_Timer（）添加。

```
int delete_fore_info_timer(
    INTID station_id,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] status
指向定时器删除成功的站内节点信息记录结构体

返回值

0：添加成功

-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

add_fore_info_timer

4.1.12.11 add_distance_fore_timer

向站 ID 为 station_id 的站添加系统预取表定时器，该定时器每隔 day 天 timer_value 秒运行一次。

```
int add_distance_fore_timer(
    INTID station_id,
    unsigned int timer_value,
    int day,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id

站号（标识符）

[in] timer_value

定时器时间值(单位：秒)

[in] day

定时器时间值（单位：24 天）

[out] status

指向定时器添加成功的站内节点信息记录结构体

返回值

0：添加成功

-1：添加失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

delete_distance_fore_info_timer

4.1.12.12 delete_distance_fore_info_timer

删除站 ID 为 station_id 的系统预取表定时器，删除的定时器是由函数 add_distance_fore_timer（）添加。

```
int delete_distance_fore_info_timer(
    INTID station_id,
    struct TimerAck_Msg **status
);
```

参数

[in] station_id
站号（标识符）
[in] status
指向定时器删除成功的站内节点信息记录结构体

返回值

0：删除成功
-1：删除失败

备注

调用者需要调用 free()函数释放 status 指向的空间

相关接口

add_distance_fore_timer

4.1.12.13 space_get_all_files

按照 which 的值从不同的预取表中获取预取表中所有文件名，并将这些文件名保存到 list 指向的链表中。

```
int space_get_all_files(
    int *which,
    struct file_name_item **list
);
```

参数

[in] which
*which 为 1 表示取 PreTableOfLarge 表中所有文件，*which 为 2 表示取 PreTableOfCretime 表中所有文件，*which 为 3 表示取 PreTableOfAcctimes 表中所有文件
[out] list
指向链表元素类型为 struct file_name_item 的链表表头的指针

返回值

-1：*which 值不在集合{1，2，3}中取值。
大于或等于 0：表示对应预取表中文件数目

备注

链表 list 使用结束后需要 free_file_name_list 释放内存资源

相关接口

space_get_all_files
space_get_all_files_about_user
free_file_name_list
space_get_app_files_about_user
free_app_files

4.1.12.14 space_get_all_files_about_user

获取指定用户 user_name 的所有文件名；并将所有文件名保存到 list 指向的结构体链表中。

```
int space_get_all_files_about_user(
    char *user_name,
    struct file_name_item **list
);
```

参数

[in] user_name
用户名
[out] list
指向链表元素类型为 struct file_name_item 的链表表头的指针

返回值

-1：user_name 指针为 NULL。
大于或等于 0：表示该用户拥有的文件数目

备注

调用者需要调用 free_filename_list()函数释放 list 指向的空间

相关接口

space_get_all_files
free_file_name_list
space_get_app_files_about_user
free_app_files

4.1.12.15 free_file_name_list

释放链表元素类型为 struct file_name_item 的链表空间

```
int free_file_name_list(
    struct file_name_item *list
);
```

参数

[in] list

指向链表元素类型为 struct file_name_item 的链表表头的指针

返回值

0：释放成功
-1：list 为 NULL

备注

无

相关接口

space_get_all_files
space_get_all_files_about_user
space_get_app_files_about_user
free_app_files

4.1.12.16 space_get_app_files_about_user

获取指定用户 user_name 的所有文件名；并将所有文件名保存到 list 指向的结构体链表中，指定获取文件的开始编号和数量。

```
int space_get_app_files_about_user(
        char *uname,
        int begin_num,
        int count,
        struct app_files_item **list
);
```

参数

[in] uname
用户名
[in] begin_num
文件的开始编号
[in] count
文件的数量
[out] list
保存文件的链表

返回值

返回文件的数量，如果返回值为负数，表示出现错误

备注

无

相关接口

space_get_all_files

space_get_all_files_about_user
free_file_name_list
free_app_files

4.1.12.17 free_app_files

释放 app_files_item 结构的链表。

```
void free_app_files(
    struct app_files_item *list
);
```

参数
[in] list
app_files_item 结构的链表

返回值
无

备注
无

相关接口
space_get_all_files
space_get_all_files_about_user
free_file_name_list
space_get_app_files_about_user

4.1.12.18 space_get_files_about_callback

按照 which 的值和 username 从不同的预取表中获取预取表中所有文件名，并将这些文件名保存到 list 指向的链表中。

```
int space_get_files_about_callback(
    int which,
    int item_num,
    char *username,
    struct callback_file_desc **list
);
```

参数
[in] witch
which 为 1 表示取 PreTableOfLarge 表中所有文件，which 为 2 表示取 PreTableOfCretime 表中所有文件，which 为 3 表示取 PreTableOfAcctimes 表中所有文件
[in] item_num
获取的文件最大数量，即“list”返回链表的最大节点数

[in] username
用户名
[out] list
用户返回文件的 callback_file_desc 结构的链表

返回值
无

备注
无

相关接口
space_get_all_files
space_get_all_files_about_user
space_get_app_files_about_user
free_callback_file_desc

4.1.12.19 free_callback_file_desc

释放链表的内存资源。

```
void free_callback_file_desc(
    struct callback_file_desc *list
);
```

参数
[in] list
callback_file_desc 结构的链表

返回值
无

备注
无

相关接口
space_get_files_about_callback

4.1.12.20 space_execute_poise

指定站 station_id 执行站内空间均衡。

```
int space_execute_poise(
    INTID    station_id
);
```

参数

[in] station_id
站号（标识符）

返回值

-1：站 ID 号为 0 或空间均衡执行失败
0：空间均衡执行成功

备注

无

相关接口

无

4.1.13 敏感信息存储管理

敏感信息用于保存高度机密的小的数据，比如密码和指纹等。主要包括保存、获取、删除操作。此外，还提供它的权限操作，比如用户和组，模式等。

4.1.13.1 add_object

保存敏感信息。

```
int add_object(
        char *object_name,
        char *object_data,
        int data_len,
        unsigned int mode,
        int replace
);
```

参数

[in] object_name
信息名称
[in] object_data
信息内容
[in] data_len
信息内容的长度
[in] mode
读写权限，同文件的权限一致，分为用户、组和其他的权限，使用八进制来表示，1 表示读权限，用宏 MY_READ 来表示，2 表示写权限，用宏 MY_WRITE 来表示。例如“0300”表示只有本用户才有权限读写。特别注意，“DOSroot”用户具有所有的权限
[in] replace
是否强制替换，1 表示强制替换，删除原有的信息，重写新的信息；0 表示不替换，如果已经存在，该接口将报错

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
get_object
delete_object

4.1.13.2 get_object

获取敏感信息的数据。

```
int get_object(
    char *object_name,
    char ** data_save_ptr,
    int *data_out_len
);
```

参数
[in] object_name
信息名称
[out] data_save_ptr
信息内容，需要调用系统函数 free 来释放
[out] data_out_len
信息内容长度

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
add_object
delete_object

4.1.13.3 delete_object

删除敏感信息。

```
int delete_object(
    char *object_name
);
```

参数
[in] object_name
信息名称

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
add_object
get_object

4.1.13.4 get_object_own

获取敏感信息的所有者，返回所有者的用户名，用户所在的组名。

```
int get_object_own(
        char *object_name,
        char **group,
        char **user
);
```

参数
[in] object_name
信息名称
[out] group
用户组名，需要使用系统调用 free 释放
[out] user
用户名，需要使用系统调用 free 释放

返回值
成功返回 0，否则返回非 0

备注
无

相关接口
chown_object

4.1.13.5 chown_object

修改敏感信息的所有者。

```
int chown_object(
```

```
    char *object_name,
    char *new_group,
    char *new_user
);
```

参数

[in] object_name
信息名称
[in] new_group
新的用户组名
[in] new_user
新的用户名

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

get_object_own

4.1.13.6 get_object_mod

获取敏感信息的模式，即权限。

```
int get_object_mod(
    char *object_name,
    unsigned int *mode
);
```

参数

[in] object_name
信息名称
[out] mode
模式

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

chmod_object

4.1.13.7 chmod_object

修改敏感信息的模式。

```
int chmod_object(
    char *object_name,
    unsigned int mode
);
```

参数

[in] object_name
信息名称
[in] mode
模式

返回值

成功返回 0，否则返回非 0

备注

无

相关接口

get_object_mod

4.1.14 文件系统的事务管理

数字有机体的文件服务支持事务。即保证事务的所有文件操作要么都成功，要么都失败。一个事务总是以“dos_begin”开始，以“dos_commit”或“dos_abort”结束。

4.1.14.1 dos_begin

使用该接口开始一个事务。

```
int dos_begin(
    int fd
);
```

参数

[in] fd
已打开文件的描述符

返回值

返回 0 表示成功，否则表示失败

备注

无

相关接口
dos_commit
dos_abort

4.1.14.2 dos_commit

使用该接口结束一个事务，并表示提交该事务引起的更新。

```
int dos_commit(
    int fd
);
```

参数
[in] fd
已打开文件的描述符

返回值
返回 0 表示成功，否则表示失败

备注
无

相关接口
dos_begin
dos_abort

4.1.14.3 dos_abort

使用该接口结束一个事务，并表示取消该事务引起的更新。

```
int dos_abort(
    int fd
);
```

参数
[in] fd
已打开文件的描述符

返回值
返回 0 表示成功，否则表示失败

备注
无

相关接口

dos_begin
dos_commit

4.1.15 其它辅助接口

本章节介绍一些辅助类的接口，这些接口主要包括参数校验、站 ID 的转换、路径转换和错误描述。

4.1.15.1 check_station_id_valid

检验站号（标识符）是否合法。

```
int check_station_id_valid(
    char* id
);
```

参数
[in] id
站号（标识符）

返回值
返回 0 表示合法，否则不合法

备注
无

相关接口
无

4.1.15.2 check_ipv4_host_valid

检验点进制 IP 字符串是否合法。

```
int check_ipv4_host_valid(
    char *ip
);
```

参数
[in] ip
点进制 IP 字符串

返回值
返回 0 表示合法，否则不合法

备注

无

相关接口
无

4.1.15.3 check_group_name_valid

检验用户组名是否合法。

```
int check_group_name_valid(
    char *group_name
);
```

参数
[in] group_name
用户组名

返回值
返回 0 表示合法，否则不合法

备注
无

相关接口
无

4.1.15.4 check_user_name_valid

检验用户名是否合法。

```
int check_user_name_valid(
    char *user_name
);
```

参数
[in] user_name
用户名

返回值
返回 0 表示合法，否则不合法

备注
无

相关接口
无

4.1.15.5htonll

转换 int64_t 类型的数据为网络字节序的数据。

```
int64_t htonll(
    int64_t val
);
```

参数

[in] val
主机字节序的数字

返回值

网络字节序的数字

备注

无

相关接口

ntohll

4.1.15.6ntohll

转换 int64_t 类型的数据为主机字节序的数据。

```
int64_t ntohll(
    int64_t val
);
```

参数

[in] val
网络字节序的数字

返回值

主机字节序的数字

备注

无

相关接口

htonll

4.1.15.7get_char_id

由高 64 位和低 64 位的数字构建一个字符串格式的站 ID。

```
void get_char_id(
    uint64_t id1,
    uint64_t id2,
    char *out_id
);
```

参数

[in] id1
站的高 64 位数字
[in] id2
站的低 64 位的数字
[out] out_id
字符串格式的站 ID

返回值

无

备注

无

相关接口

get_low64
get_high64

4.1.15.8 get_low64

获取站号（标识符）的低 64 位（用于字符串表示的 ID），参数为字符串形式表示的 ID。

```
uint64_t get_low64(
    char *id
);
```

参数

[in] id
站号（标识符）

返回值

站号（标识符）的低 64 位

备注

无

相关接口

get_high64

4.1.15.9 get_high64

获取站号（标识符）的高 64 位（用于字符串表示的 ID），参数为字符串形式表示的 ID。

```
uint64_t get_high64(
    char *id
);
```

参数
[in] id
站号（标识符）

返回值
站号（标识符）的高 64 位

备注
无

相关接口
get_low64

4.1.15.10 get_id_low64

获取站号（标识符）的低 64 位（用于 INTID 表示的 ID），参数为 INTID 类型的 ID。

```
uint64_t get_id_low64(
    INTID id
);
```

参数
[in] id
站号（标识符）

返回值
站号（标识符）的低 64 位

备注
无

相关接口
get_id_high64

4.1.15.11 get_id_high64

获取站号（标识符）的高 64 位(用于 INTID 表示的 ID)，参数为 INTID 类型的 ID。

```
uint64_t get_id_high64(
    INTID id
);
```

参数
[in] id
站号（标识符）

返回值
站号（标识符）的高 64 位

备注
无

相关接口
get_id_low64

4.1.15.12 string_to_INTID

把字符串类型的站号（标识符）转换为 INTID 类型的站号。

```
int string_to_INTID(
    char* value,
    INTID *id
);
```

参数
[in] valu
字符串类型的站号（标识符）
[out] id
INTID 类型的站号（标识符）

返回值
返回 0 表示成功，否则表示失败

备注
无

相关接口
INTID_to_string

4.1.15.13 INTID_to_string

把 INTID 类型的站号（标识符）转换为字符串类型的站号。

```
int INTID_to_string(
```

```
    INTID *id,
    STRID *pos
);
```

参数

[in] id
INTID 类型的站号（标识符）
[out] pos
字符串类型的站号（标识符）

返回值

返回 0 表示成功，否则表示失败

备注

无

相关接口

string_to_INTID

4.1.15.14 dos_realpath

获取文件的绝对路径，如果文件不在“/dpfs”目录下，该接口将返回失败。

```
int dos_realpath(
    char *name,
    char *real_path,
    int use_mount = 0
);
```

参数

[in] name
文件名
[out] real_path
文件的绝对路径
[in] use_mount
是否需要计算绝对路径，0 表示不需要，1 表示需要

返回值

成功返回 0，否则返回-1

备注

use_mount 一般配置为 0

相关接口

无

4.1.15.15 error_string

根据给定的错误号，返回一个使用英文描述的错误描述字符串。

```
char* error_string(
    int errno
);
```

参数
[in]errno
错误号

返回值
错误号对应的错误描述字符串

备注
无

相关接口
无

4.1.16 文件系统

数字有机体文件系统向用户提供标准的 Linux 文件系统调用接口。使用户可以使用标准的文件操作函数的方式访问数字有机体文件系统中的文件，和本地操作一致。但在数字有机体系统环境下，标准文件系统的一些特性难以保留和实现，所以系统不能完全兼容和支持标准文件系统的一些特性，这将在后续版本逐步改进。

4.1.16.1 目录操作

1) 创建目录（mkdir）

为便于管理，待创建目录的目录项名字长度应小于 126 个字符，目录名命名规则同传统文件系统命名规则相同。除以上不同外，系统对创建目录的支持与标准 Linux 文件系统调用接口完全兼容。

2) 删除目录（rmdir）

系统对删除子目录的支持与标准 Linux 文件系统调用接口完全兼容。

3) 遍历目录树(readdir)

系统对此功能的支持与标准 Linux 文件系统调用接口完全兼容。

4.1.16.2 文件操作

1) 打开文件（open、fopen）

由于目前数字有机体工作平台用户尚未实现用户信息及其权限的记录和划分。因此，打开文

件的权限设置缺乏意义和依据，目前暂不支持 Linux 原有的权限机制。除权限机制外，系统对此功能的支持与标准 Linux 文件系统调用接口完全兼容。

2) 文件读 (read)

系统对文件读的支持与标准 Linux 文件系统调用接口完全兼容。

3) 定位读写位置 (lseek)

系统对定位读写位置的支持与标准 Linux 文件系统调用接口完全兼容。

4) 文件写（write）

系统对文件写的支持与标准 Linux 文件系统调用接口完全兼容。

5) 关闭文件 (close)

系统对关闭文件的支持与标准 Linux 文件系统调用接口完全兼容。

6) 文件截短(truncate、ftruncate)

本系统文件的截短功能与标准 Linux 文件系统调用接口完全兼容。

7) 获取文件大小（stat、fstat）

本系统中获取文件大小与标准 Linux 文件系统调用接口完全兼容。

4.1.16.3 暂不支持的操作

由于广域网环境的复杂性和多样性，传统文件系统的一些特性难以保留和满足，以下是系统暂不支持或对传统文件系统特性支持不够好的地方，我们将在后续版本逐步改进和添加。

1) 文件时间

由于广域网的固有特点，各个节点维持一个全局一致的时间是不现实的。因此，原先单个节点上的时间标记对系统中的其他节点失去了实际意义，文件时间目前只有参考意义。

2) 符号链接和硬链接

目前在系统全局目录/dpfs 下，不能创建符号链接和硬连接。但可以在一般文件系统下创建指向/dpfs 下文件或目录的符号链接。符号连接和硬连接的主要功能都一样，就是给真实文件一个别名。用户可以以这个别名访问文件，以便于文件共享。但是，由于虚拟文件系统中的文件本来就是共享的，因此该功能的优势并不明显。其次从实现上来讲，因为读取和访问符号连接时，实际上是先读取符号连接文件，然后再读取其指向的真实文件。两次操作的时延可能较大。由于在数字有机体文件系统中，一个文件并不仅仅对应硬盘上的一个文件，因此硬连接无法指明是对应那个文件。基于这两个原因，目前不支持硬连接。

3) 不支持目录改名

目前在全局目录下改目录名代价过大，暂时不支持。

4) 不支持 mmap( )

从文件内存映象功能的应用目的看，它不是文件应用的主流，尤其不是网络共享文件的应用主流。如果文件可直接执行，将给病毒侵入和传播提供机会，因此对此功能的支持将建立在安全机制得以实现的基础上。

5) 不支持严格的访问互斥控制（即标准的 unix 语义）

在广域网环境下的访问互斥控制是目前的世界性难题，还没找到能同时有令人可接受的可靠性和效率的有效算法，我们也在不断的修改设计，将在适当的时候在新版本中推出。

4.2 宏定义

宏定义	宏定义值	含义
SINGLEADDR	0x00	搜索规模为 1 个地址
FEWADDR	0x01	搜索规模为 4-16 个地址
COUPLEADDR	0x02	搜索规模为 16-64 个地址
ABUNDANTADDR	0x04	搜索规模为 64-256 个地址
ALLADDR	0x08	搜索规模为 256 以上个地址
NODE_REGGI_TYPE	0	注册
NODE_UNREG_TYPE	1	注销
CLIENT_NODE	0	所有节点
SERVER_NODE	1	服务器节点
SUCCESS	0	成功
SERVICE_INFO_EXIST	-1	服务信息已经存在
INPUT_ERROR	-10	输入参数错误
MALLOC_ERROR	-11	分配内存失败
UNKNOWN_MSG_ERROR	-12	未知的错误
IDDATA	IDData	保存“站”的结构类型
ID_SIZE	16	“站”标识符字符串最大长度
ONLYMINE	0x00	本站搜索
NEIGHBOR	0x10	一跳距离内搜索
NEIGHBOR_UP	0x20	两跳距离内搜索
NEIGHBOR_MUTI	0x40	三跳距离内搜索
GLOBAL	0x80	全局范围内搜索
SERVERSEARCH	1	服务器上搜索

ALLSEARCH	0	客户端和服务器上都搜索
MULTI_TP_NOT_USE	0x0000	随机放置
MULTI_TP_USE_DEF_CREAET_NUM	0x0001	按照默认位置放置，需要创建指定个数，默认位置不足时随机放置
MULTI_TP_USE_DEF_STATION	0x0002	使用默认位置创建副本
MULTI_TP_ONLY_USE_DEF_STATION	0x0004	只按照默认位置放置
MULTI_TP_USE_CREATE_NUM	0x0008	创建指定个数副本
MULTI_TP_USE_ACID	0x00F0	使用事务机制
DEF_STATIONS	5	最大默认站个数
RES_NAME_LEN	1024	默认资源名长度
MAXSITENUMINSTA	256	站内最多节点数
MAX_STATION_NODES	256	站内最多节点数
NODE_NAME_MAX_LEN	32	节点名字最大长度
USER_NAME_LEN	13	用户名最大长度
MAXNAMELEN	1024	最大名字长度
MAX_USER_COM_NAME_LEN	16	通讯时用户名的最大长度
PASS_LEN	1024	密码最大长度
GROUP_NAME_LEN	16	组名最大长度

4.3数据结构

4.3.1 user_landing_host

可登录节点

```
struct user_landing_host
{
    unsigned int node_ip[MAXSITENUMINSTA]; //节点 ip
};
```

参数

node_ip
可登录的节点 ip 数组

4.3.2 name_list_item

用户列表。

```
struct name_list_item
{
    char pname[USER_NAME_LEN]; 用户名
    struct name_list_item *next;//指向下一个用户
};
```

参数

Pname
用户名
Next
指向下一个用户

4.3.3 p_basic_user_info

这个结构体描述用户的基本信息。

```
typedef struct user_basic_info
{
        int is_valid;
        unsigned int version;
        char p_name[USER_NAME_LEN];
        unsigned int uname_id;
        char p_gname[USER_NAME_LEN];
        unsigned int gname_id;
        unsigned int node_num;
        struct user_landing_host  p_host;
        char preal_name[MAXNAMELEN];
        char puser_des[MAXNAMELEN];
        struct user_basic_info *next;
} __attribute__((aligned(4))) *p_basic_user_info, basic_user_info;
```

参数

is_valid
该用户是否有效
version
版本号
p_name
用户名
uname_id

用户 id
p_gname
所属组名
gname_id
所属组 id
node_num
节点数
p_host
可登录主机
preal_name
真实姓名
puser_des
用户描述信息
next
指向链表下一个节点的地址

4.3.4 p_user_authen_info

这个结构体描述用户的认证信息。

```
typedef struct user_authentic_info
{
        char user_name[USER_NAME_LEN];
        unsigned int auth_info_len;
        char *auth_info;
} __attribute__((aligned(4))) *p_user_authen_info, user_authen_info;
```

参数

user_name
用户名
auth_info_len
密码长度
auth_info
密码

4.3.5 PuserQuota

用户配额结构体。

```
typedef struct User_Quota{
        char user_name[MAX_USER_COM_NAME_LEN];
        int64_t logic_quota;
        int64_t real_quota;
        int64_t remainder_logic_quota;
```

```
        int64_t remainder_real_quota;
        unsigned int version;
        unsigned int pad;
} __attribute__((aligned(4))) UserQuota,*PuserQuota;
```

参数

user_name[MAX_USER_COM_NAME_LEN]
用户名
logic_quota
逻辑配额
real_quota
实际配额
remainder_logic_quota
剩余逻辑配额
remainder_real_quota
剩余实际配额
Version
版本号
Pad
用来使字节对齐，无实际意义

4.3.6 p_group_info

组基本信息。

```
typedef struct group_info
{
        unsigned int gname_id;
        unsigned int version;
        int valid;
        time_t create_time;
        char p_gname[USER_NAME_LEN];
        char p_gdesc[MAXNAMELEN];
        unsigned int user_num;
        struct name_list_item *p_user_name_list;
        struct group_info *next;
} __attribute__((aligned(4))) *p_group_info, ginfo;
```

参数

gname_id
组 id
Version
版本号

Valid
是否有效
p_gname
组名
p_gdesc
组描述信息
user_num
成员数量
p_user_name_list
组成员列表
Next
指向下一个组

4.3.7 pg_list_info

组列表。

```
typedef struct group_list_info
{
        int g_num;
        struct name_list_item *pglist;
} __attribute__((aligned(4))) *pg_list_info, g_list_info;
```

参数

g_num
组的数量
Pglist
组名列表

4.3.8 SDesc

注册器传给服务信息管理模块的数据结构。

```
typedef struct Service_Desc
{
        char *serviceName;                      /* name of service */
        char *serviceOwner;                     /* owner of service */
        time_t createTime;                      /* the service time that provides
service in the beginning. */
        unsigned int useCount;                  /* how many times that service have
been provided */
} __attribute__((aligned(4))) SDesc;
```

参数

serviceName
服务名
serviceOwner
服务所有者
createTime
服务的开始时间
useCount
服务被使用的次数

4.3.9 RLDesc

注册器传给服务信息管理模块的数据结构。

```
typedef struct Resource_Location_Desc
{
        char *serviceName;                      /* name of service */
        char *serviceOwner;                     /* owner of service */
        INTID stationID;                        /* service node's station id */
        unsigned int nodeIP;                    /* service naode' IP */
        unsigned int useCount;                  /* count of used */
        time_t joinTime;                        /* provide service's time */
        int locationType;                       /* CLIENT_NODE or SERVER_NODE */
        unsigned int publicIP;                  /* public node IP if the node is
in a private network */
        short servicePort;                        /* service port that provides
services */
        struct Resource_Location_Desc *next;
} __attribute__((aligned(4))) RLDesc;
```

参数

serviceName
服务名
serviceOwner
服务所有者
stationID
站号（标识符）
nodeIP
节点 IP 地址
useCount
使用数量
joinTime
提供服务的时间
locationType
CLIENT_NODE 或者 SERVER_NODE

publicIP
公网 IP 地址
servicePort
服务端口
next
指向下一个地址的指针

4.3.10 Service_Node_Info

服务节点信息。

```
struct Service_Node_Info
{
        INTID station_id;
        unsigned int public_ip;
        unsigned int private_ip;
        unsigned short port;
        unsigned short flag;    /* CLIENT_NODE or SERVER_NODE */
        int hop;
        struct load_info load;
        unsigned int route[LOCAL_HOP];
} __attribute__((aligned(4)));
```

参数

station_id
站号（标识符）
public_ip
公网 IP 地址
private_ip
私网 IP 地址
port
端口
flag
CLIENT_NODE 或者 SERVER_NODE
hop
跳数
load
负载信息
rout
路由表

4.3.11 Node_Load

节点部分负载信息。

```
struct Node_Load {
    unsigned int cpu_usage;
    unsigned int mem_usage;
    unsigned int disk_speed;
    unsigned int net_flow;
};
```

参数

cpu_usage
cpu 使用率
mem_usage
内存使用率
disk_speed
磁盘 I/O 读写速度（单位：kb/s）
net_flow
网络流量（单位：kb/s）

4.3.12 Disk_Info

磁盘信息结构体。

```
struct Disk_Info {
    unsigned int disk_total; //磁盘总容量（单位：Mb）
    unsigned int disk_free; //磁盘空闲容量（单位：Mb）
    unsigned int disk_usage;//磁盘使用率（单位：%）
};
```

参数

disk_total
磁盘总容量（单位：Mb）
disk_free
磁盘空闲容量（单位：Mb）
disk_usage
磁盘使用率（单位：%）

4.3.13 load_info

节点负载信息描述结构体。

```
struct load_info
{
        int cpu_usage;          /* cpu usage:percent */
        int mem_usage;          /* memory usage:percent */
```

```
        int disk_speed;         /* disk speed:KB/s */
        int net_flow;           /* net flow:KB/s */
        unsigned int ip;        /* node ip */
        unsigned int pad;       /* padding */
} __attribute__((aligned(4)));
```

参数

cpu_usage
已经使用的 CPU 百分比
mem_usage
已经使用的内存百分比
disk_speed
磁盘的转速，单位为 KB/s
net_flow
网络的流量，单位为 KB/s
ip
IP 地址
pad
占位符

4.3.14 Route_Info

路由表信息记录结构体。

```
struct Route_Info
{
        union {
                unsigned int client;
                unsigned int station;
        }node;
        char type;
        int hops;
        struct routes *header;
        struct Route_Info *next;
} __attribute__((aligned(4)));
```

参数

type
0 表示 client，1 表示 station
hops
总跳数
header
路由表

next
路由表信息记录链表的下一个指针。

4.3.15 routes

构建路由表的单跳信息。

```
struct routes{
    unsigned int ip;
    struct routes *next;
};
```

参数

ip
节点 IP 地址
next
下一跳的指针

4.3.16 Service_Ack_Info

记录服务的结构。

```
struct Service_Ack_Info{
        UINT serviceCount;
        struct Service_Info_Item services[1];
} __attribute__((aligned(4)));
```

参数

serviceCount
服务的数量
services
服务数组

4.3.17 Item_Location_Info

副本信息的链表结构。

```
struct Item_Location_Info
{
        struct Item_Location item;
        struct Item_Location_Info *next;
} __attribute__((aligned(4)));
```

参数

item
副本信息存储结构
next
指向链表的下一个地址

4.3.18 Item_Location

本结构用于描述一个副本节点。

```
struct Item_Location{
    INTID stationID;
    unsigned int siteIp;
};
```

参数

stationID
副本所在节点所属的站 ID
siteIp
副本所在节点 IP 地址

4.3.19 Node_Alarm_Msg

节点报警记录。

```
struct Node_Alarm_Msg
{
        int node_num;
        struct Show_Alarm_Info nodealarm[0];
} __attribute__((aligned(4)));
```

参数

node_num
指定节点报警次数
nodealarm
报警节点各项负载报警值

4.3.20 Station_Alarm_Msg

站点报警记录。

```
struct Station_Alarm_Msg
{
        int station_num;
        struct Show_Alarm_Info stationalarm[0];
```

```
} __attribute__((aligned(4)));
```

参数

station_num
指定站报警次数
stationalarm
报警站各项负载报警值

4.3.21 AddTimerAck_Info

指定节点添加或删除定时器返回状态记录。

```
struct AddTimerAck_Info
{
        unsigned int node_ip;
        int curstatus;                          /* 1 is set Ok ,0 is set failed */
} __attribute__((aligned(4)));
```

参数

node_ip
节点 IP 地址
curstatus
1 表示设置成功，0 表示设置失败

4.3.22 TimerAck_Msg

添加或删除定时器返回状态记录。

```
struct TimerAck_Msg
{
        unsigned int node_num;
        struct AddTimerAck_Info timerstatus[0];
} __attribute__((aligned(4)));
```

参数

node_num
添加或删除定时器成功的节点数
timerstatus
/添加或删除定时器成功节点状态记录

4.3.23 Read_Config

读取外核配置文件的所有配置项值。

```
struct Read_Config
{
        unsigned int m_nodeIP;                  /* local node ip */
        unsigned int m_nodeCap;                 /* local node's capacity */
        unsigned int m_groupAddr;               /* broadcast ip */
        unsigned int m_groupPort;               /* broadcast port */
        char m_nodeName[NODE_NAME_MAX_LEN];     /* host name */

        INTID m_stationId;
        INTID m_sucStationID;
        unsigned int m_capacity;                /* station's */
        char m_localityDec[LOCAL_DES_LEN];
        unsigned int m_netPort;                 /* the port used to send data to
other station */
        unsigned int m_schedulePort;

        /* for database config */
        char m_databaseName[SHORT_NAME_LEN];
        char m_dbUser[SHORT_NAME_LEN];
        char m_dbPasswd[SHORT_NAME_LEN];
        char m_dbHost[SHORT_NAME_LEN];
        int  m_dbconnects;

        unsigned int m_criticalCpuUsage;        /* cpu critical line for schedule
optimize */
        unsigned int m_criticalMemUsage;        /* memory critical line for
schedule optimize */
        unsigned int m_criticalDiskUsage;       /* disk read/writeusage critical
line for schedule optimize */
        unsigned int m_criticalNetUsage;        /* network send/reacive critical
line for schedule optimize */

        /* for node alarm line */
        unsigned int m_cpu_alarm_line;
        unsigned int m_mem_alarm_line;
        unsigned int m_speed_alarm_line;
        unsigned int m_disk_alarm_line;
        unsigned int m_net_alarm_line;

        /* for node weight */
        unsigned int m_lcpup;
        unsigned int m_lmemp;
        unsigned int m_ldiskp;
```

```
unsigned int m_cpup;
unsigned int m_memp;
unsigned int m_diskp;
unsigned int m_netp;

/* for station weight */
unsigned int m_scpup;
unsigned int m_smemp;
unsigned int m_sdiskp;

/* for node and station weight line */
unsigned int m_nodeWeight;
unsigned int m_stationWeight;

/* for node timer */
unsigned int m_collect_timer;
unsigned int m_poise_timer;
unsigned int m_poise_day;
unsigned int m_poise_type;
unsigned int m_foretable_timer;
unsigned int m_foretable_day;
unsigned int m_foretable_type;
unsigned int m_fresh_timer;
unsigned int m_fresh_day;
unsigned int m_fresh_type;
unsigned int m_nodealarm_timer;
unsigned int m_stationalarm_timer;

/* for main manager address */
unsigned int m_main_manager_station;
unsigned int m_main_manager_site;

/* for rep num control */
unsigned int m_rep_num_in_station;
unsigned int m_rep_num_of_station;
unsigned int m_space_callback;
int m_uses_quota;

/* for cache system,set in M Bety */
unsigned int m_cache_block_size;
unsigned int m_cache_max_size;
char m_cache_path[RES_NAME_LEN];
```

```
        /* for rep copy speed */
        unsigned int m_rep_copy_speed;

        /* every neighbor station give out a node's IP */
        unsigned int m_NBNodeIPs[MAX_STATION_NODES];
} __attribute__((aligned(4)));
```

参数

m_nodeIP
本机节点 IP 地址
m_nodeCap
本机节点处理能力值
m_groupAddr
节点组播 IP 地址
m_groupPort
站内节点之间的通讯端口号
m_nodeName
节点服务器名称
m_stationId
本节点所属站 ID 号
m_sucStationID
节点所在站的后继站 ID 号
m_capacity
节点所在站的处理能力值
m_localityDec
站点描述符
m_netPort
系统通讯端口号
m_schedulePort
系统调度通信端口号
m_databaseName
文件属性库名
m_dbUser
文件属性库用户名
m_dbPasswd
文件属性库密码
m_dbHost
文件属性库所在节点 IP 地址
m_dbconnects
文件属性库最大连接数
m_criticalCpuUsage
本节点 CPU 负载上限值
m_criticalMemUsage
本节点内存负载上限值

m_criticalDiskUsage
本节点磁盘使用率上限值
m_criticalNetUsage
本节点网络流量上限值
m_cpu_alarm_line
CPU 负载报警线
m_mem_alarm_line
内存负载报警线
m_speed_alarm_line
磁盘 I/O 读写速度报警线（单位：kb/s）
m_disk_alarm_line
磁盘使用率报警线
m_net_alarm_line
网络流量报警线（单位：kb/s）
m_lcpup
计算本节点平均负载值的各项权重，cpu 使用率权重
m_lmemp
计算本节点平均负载值的各项权重，内存使用率权重
m_ldiskp
计算本节点平均负载值的各项权重，磁盘使用率权重
m_cpup
本节点 CPU 权重
m_memp
本节点内存权重
m_diskp
本节点磁盘使用率权重
m_netp
本节点网络流量权重
m_scpup
计算节点所在站的平均负载值的各项权重，cpu 使用率权重
m_smemp
计算节点所在站的平均负载值的各项权重，内存使用率权重
m_sdiskp
计算节点所在站的平均负载值的各项权重，磁盘使用率权重
m_nodeWeight
本节点平均负载报警线
m_stationWeight
本站平均负载报警线
m_collect_timer
系统收集负载信息定时器时间值（单位：秒）
m_poise_timer
站空间均衡定时器时间值（单位：秒）
m_poise_day
站空间均衡定时器时间值（单位：24 小时）

m_poise_type
定时器类型。0 表示相对定时器（包含 day 的时间在内且 day 不为零），1 表示绝对定时器（不包含 day 的时间在内）
m_foretable_timer
站内系统预取表定时器时间值（单位：秒）
m_foretable_day
站内系统预取表定时器时间值（单位：24 小时）
m_foretable_type
定时器类型，同 m_poise_type。
m_fresh_timer
站内用户文件表定时器时间值（单位：秒）
m_fresh_day
站内用户文件表定时器时间值（单位：24 小时）
m_fresh_type
定时器类型，同 m_poise_type。
m_nodealarm_timer
节点报警定时器时间值（单位：秒）
m_stationalarm_timer
站报警定时器时间值（单位：秒）
m_main_manager_station
主管理机所在站的站 ID 号
m_main_manager_site
主管理机的 IP 地址
min_rep_num_in_station
站内最小文件副本数
max_rep_num_in_station
站内最大文件副本数
min_rep_num_of_station
站间最小文件副本数
max_rep_num_of_station
站间最大文件副本数
m_space_callback
空间回收模块使用标记
m_uses_quota
系统启动配额机制标记
m_cache_block_size
本节点 Cache 空间大小（默认单位：Mb）
m_cache_max_size
本节点 Cache 块粒度（默认单位：Mb）
m_rep_copy_speed
站间副本拷贝速度（默认单位：Kb）
m_cache_path
存放 Cache 的路径
m_NBNodeIPs

本节点邻居节点 IP 地址

4.3.24 Can_Set_Config

外核配置文件可配置项，各配置项含义见 struct Read_Config 中注释。

```
struct Can_Set_Config
{
        unsigned int m_nodeCap;                 /* local node's capacity. */
        unsigned int m_groupPort;
        char m_nodeName[NODE_NAME_MAX_LEN];
        unsigned int m_capacity;                /* station's capacity */
        char m_localityDec[LOCAL_DES_LEN];
        unsigned int m_netPort;                 /* The port used to send data to
other station. */
        unsigned int m_schedulePort;

        /* for database config. */
        char m_databaseName[SHORT_NAME_LEN];
        char m_dbUser[SHORT_NAME_LEN];
        char m_dbPasswd[SHORT_NAME_LEN];
        char m_dbHost[SHORT_NAME_LEN];
        int  m_dbconnects;

        unsigned int m_criticalCpuUsage;        /* cpu critical line for schedule
optimize */
        unsigned int m_criticalMemUsage;          /* memory critical line for
schedule optimize */
        unsigned int m_criticalDiskUsage;       /* disk read/writeusage critical
line for schedule optimize */
        unsigned int m_criticalNetUsage;        /* network send/reacive critical
line for schedule optimize */

        /* for node alarm line */
        unsigned int m_cpu_alarm_line;
        unsigned int m_mem_alarm_line;
        unsigned int m_speed_alarm_line;
        unsigned int m_disk_alarm_line;
        unsigned int m_net_alarm_line;

        /* for node weight */
        unsigned int m_lcpup;
        unsigned int m_lmemp;
        unsigned int m_ldiskp;                  /* disk usage (%) */
```

```
        /* for station weight */
        unsigned int m_scpup;
        unsigned int m_smemp;
        unsigned int m_sdiskp;

        /* for node and station weight line */
        unsigned int m_nodeWeight;
        unsigned int m_stationWeight;

        /* for rep num control */
        unsigned int min_rep_num_in_station;
        unsigned int max_rep_num_in_station;
        unsigned int min_rep_num_of_station;
        unsigned int max_rep_num_of_station;
        unsigned int m_space_callback;
        int m_uses_quota;

        /* for timer system's default timer value */
        unsigned int m_collect_timer;
        unsigned int m_poise_timer;
        unsigned int m_poise_day;
        unsigned int m_poise_type;
        unsigned int m_foretable_timer;
        unsigned int m_foretable_day;
        unsigned int m_foretable_type;
        unsigned int m_fresh_timer;
        unsigned int m_fresh_day;
        unsigned int m_fresh_type;
        unsigned int m_nodealarm_timer;
        unsigned int m_stationalarm_timer;

        /* for cache system, set in M Bety */
        unsigned int m_cache_block_size;
        unsigned int m_cache_max_size;
        char m_cache_path[RES_NAME_LEN];

        /* for rep copy speed */
        unsigned int m_rep_copy_speed;

        /* every neighbor station give out a node's IP */
        unsigned int m_NBNodeIPs[1];
} __attribute__((aligned(4)));
```

参数

见 struct Read_Config

4.3.25 Servicename_And_Status_List

服务信息记录。

```
struct Servicename_And_Status_List
{
    INTID StationID;//服务所在站 ID 号
    unsigned int node_ip;//服务所在节点 IP 地址
    char servicename[1024];//服务名
    int status;//服务状态，1 表示服务处于运行状态
    struct Servicename_And_Status_List *next;
};
```

参数

StationID
服务所在站 ID 号
node_ip
服务所在节点 IP 地址
Servicename
服务名
Status
服务状态，1 表示服务处于运行状态
next
指向下一地址的指针

4.3.26 System_Service_List

系统内服务列表。

```
struct System_Service_List
{
        unsigned int systemsnum;
        struct Servicename_And_Status_List *syslisthead;
} __attribute__((aligned(4)));
```

参数

systemsnum
系统内服务数目
syslisthead
指向系统内所有运行着的服务信息记录形成的链表

4.3.27 Table

Table 是一个枚举型变量，三个成员为三种类型的文件预取表表名。

```
enum Table
{
    PreTableOfLarge,
    PreTableOfCretime,
    PreTableOfAcctimes
};
```

参数

PreTableOfLarge
按文件大小排列的文件预取表表名
PreTableOfCretime
按文件创建时间排列的文件预取表表名
PreTableOfAcctimes
按文件被访问次数排列的文件预取表表名

4.3.28 StationAck_Info

站的节点和负载信息描述。

```
struct StationAck_Info
{
        unsigned int node_num;
        INTID StationID;
        struct Station_Info_Store nodeinfo[0];
} __attribute__((aligned(4)));
```

参数

node_num
站内节点个数
StationID
站 ID 号
nodeinfo[0]
记录站内各个节点的负载信息

4.3.29 All_StationAck_Info

系统内的站个数和站负载描述。

```
struct All_StationAck_Info
{
```

```
        int station_num;
        struct StationAck_Info_Sum stationinfo[0];
} __attribute__((aligned(4)));
```

参数

station_num
系统内站的个数
Stationinfo
记录每个站的平均负载信息

4.3.30 Station_Info_Store

记录节点的负载信息。

```
struct Station_Info_Store{
    struct Node_Load node_inf;
    struct Disk_Info disk_inf;
    unsigned int node_ip;
};
```

参数

node_inf
节点部分负载信息
disk_inf
节点磁盘使用情况
node_ip
节点 IP 地址

4.3.31 StationAck_Info_Sum

站点负载及磁盘信息。

```
struct StationAck_Info_Sum
{
        unsigned int node_num;
        INTID StationID;
        struct Node_Load station_load;
        struct Disk_Info sdisk_info;
} __attribute__((aligned(4)));
```

参数

node_num
站内节点个数

StationID
站 ID 号
station_load
站内所有节点的平均负载信息
sdisk_info
站内磁盘空间使用情况

4.3.32 NodeAck_Info

节点负载及磁盘信息。

```
struct NodeAck_Info
{
        struct Node_Load node_inf;
        struct Disk_Info disk_inf;
        INTID StationID;
        unsigned int node_ip;
} __attribute__((aligned(4)));
```

参数

node_inf
节点部分负载信息
disk_inf
节点磁盘使用情况
StationID
节点所在站 ID 号
node_ip
节点 IP 地址

4.3.33 file_name_item

记录文件的链表结构。

```
struct file_name_item
{
        char *name;
        struct file_name_item *next;
} __attribute__((aligned(4)));
```

参数

name
文件名
next
指向下一个地址的指针

4.3.34 app_files_item

用户文件描述。

```
struct app_files_item
{
        int total;
        int cur_count;
        struct file_name_item *file_list;
} __attribute__((aligned(4)));
```

参数

Total
总数量
cur_count
当前数量
file_list
文件列表

4.3.35 PFileConDesc

副本规则控制信息结构体。

```
typedef struct file_control_desc
{
        char create_id[ID_SIZE];
        uint64_t create_type;
        unsigned int create_ip;
        unsigned int file_def_create_num;
        unsigned int min_rep_num_in_station;
        unsigned int max_rep_num_in_station;
        unsigned int min_rep_num_of_station;
        unsigned int max_rep_num_of_station;
        char file_create_recover_def[DEF_STATIONS][ID_SIZE];
} __attribute__((aligned(4))) FileConDesc,*PFileConDesc;
```

参数

create_id
创建文件的站的 ID
create_type
创建类型，值为:
MULTI_TP_NOT_USE：不使用目录或者文件的控制信息，这时使用各站自己的配置，即在 dos_exernel.cnf 中的配置。

MULTI_TP_USE_DEF_CREAET_NUM：表示设置了创建文件时初始的副本数。这个副本数是站副本数。即使设置了其他站副本数参数，如果该标志设置，则创建时也仅创建这么多个站副本。
MULTI_TP_USE_DEF_STATION：设置了副本默认放置的站。这时结构中的 file_create_recover_def 有值。
MULTI_TP_ONLY_USE_DEF_STATION：设置此标志时，系统仅在指定的默认放置站点上创建副本。
MULTI_TP_USE_CREATE_NUM：表示设置了站内和站间副本数限制。这时结构中的 min、max 样式的成员有效。
MULTI_TP_ONLY_CREATE_STATION：仅放置在创建站点上。
MULTI_TP_USE_ACID：表示要使用写事务机制，用以保证写的原子性。
create_ip
创建文件的服务器 IP 地址
file_def_create_num
file_create_recover_def 数组中保存的数量
min_rep_num_in_station
站内最小的副本数
max_rep_num_in_station
站内最大的副本数
min_rep_num_of_station
站间最小副本数
max_rep_num_of_station
站间最大副本数
file_create_recover_def
数组，用于保存默认的副本放置位置

说明：

创建文件时，副本建立的规则是:
1)如果父目录的 create_type 有 MULTI_TP_NOT_USE，则按照 dos_exernel.cnf 中的配置创建副本。
2）如果父目录的 create_type 有 MULTI_TP_ONLY_USE_DEF_STATION，则仅使用父目录文件控制信息中设置的缺省目录来创建副本，这时 file_create_recover_def 必须有合法的站点 ID，可以是多个。
3）如果不是上述两种情况，则创建的站副本数为：如果 create_type 有 MULTI_TP_USE_DEF_CREAET_NUM 标志则使用设置的数字作为站副本数，否则若有 MULTI_TP_USE_CREATE_NUM 标志则使用设置的站间最小副本数，都没有才使用 dos_exernel.cnf 中配置的站间最小副本数。对目录来说，副本数必须大于 2。
4）如果有 MULTI_TP_USE_DEF_STATION 标志，则优先在文件控制信息中指定的缺省站上创建。最多指定 5 个。否则，优先在 dos_exernel.cnf 中配置的缺省站点上创建（最多 10 个)。如果都还不够则本站创建，如果都还不够则随机选择站。

创建文件时，如果父目录的 create_type 有 MULTI_TP_NOT_USE 标志，则使用配置文件中的参数创建副本。这时，在 dos_exernel.cnf 中有 only_local_set，如果设置为 1 则仅

在创建文件的站点上创建副本，且将在文件的 create_type 中增加 MULTI_TP_ONLY_CREATE_STATION，表示增加副本时也仅在创建站上。
要注意的是：这些参数仅仅是指导自动增减副本时的行为。你可以用命令或者接口强制在任意站点上创建副本。

4.3.36 Node_List

节点数组的存放结构。

```
struct Node_List
{
        unsigned int nodeNum;
        unsigned int pad;
        struct Node_Info nodeList[1];
}__attribute__((aligned(4)));
```

参数

nodeNum
节点数量
pad
用于占位，字节序对齐
nodeList
节点数组

4.3.37 Com_Route_Info

IP 路由信息。

```
struct Com_Route_Info
{
        unsigned int hops;
        unsigned int pad;
        unsigned int ip[MAX_HOPS];
}__attribute__((aligned(4)));
```

参数

Hops
跳数
pad
用于占位，字节序对齐
ip
路由 IP 数组

5 调度系统 SDK

5.1 接口定义

5.1.1 SDK 初始化

5.1.1.1 set_active_sched_server

配置可以提供任务调度的服务器的地址。

```
int set_active_sched_server(
    unsigned int sched_host
);
```

参数
[in] sched_host
设备 IP 地址

返回值
成功设置时返回 0，否则返回-1

备注
本接口传入的参数 sched_host 必须有效，才能进行后续的操作

相关接口
get_active_sched_server

5.1.1.2 get_active_sched_server

获取可以提供任务调度的服务器的地址。

```
int get_active_sched_server(
    unsigned int* sched_host
);
```

参数
[out] sched_host
设备 IP 地址

返回值
返回 0，否则返回-1

备注

本接口传入的参数 sched_host 必须有效，地址不能为空

相关接口

set_active_sched_server

5.1.2 服务注册与注销

5.1.2.1 register_service

该函数注册一个服务提供者。如果本服务提供者提供的是一个系统中不存在的服务，则将在系统中建立该新服务的描述。服务名是编程人员约定的名字。例如约定流媒体服务的名字为 streaming_server 等。服务名字只能采用字母、数字和下划线组成。

```
int register_service(
        DWORD service_pid,
        unsigned int node_ip,
        unsigned short service_port,
        int node_type,
        const char *service_name,
        const char *service_owner
);
```

参数

[in] service_pid

服务程序运行时的进程号

[in] node_ip

服务程序所在的节点 IP 地址，以 IPV4 网络地址方式传递

[in] service_port

注册服务程序的端口

[in] node_type

节点的类型，值为宏 CLIENT_NODE 或者 SERVER_NODE

[in] service_name

服务的名称，最大长度不能超过 1024 个字节

[in] service_owner

服务的所有者

返回值

SUCCESS：注册新服务成功

SERVICE_INFO_EXIST：新服务信息已经存在

INPUT_ERROR：输入参数错误

MALLOC_ERROR：分配空间失败

UNKNOWN_MSG_ERROR：未知错误信息

备注

上述参数都不能为空

相关接口
register_new_service
unregister_service

5.1.2.2 register_new_service

该函数用于注册一个新的服务资源。这里注册的只是服务的基本信息，即服务的名字。服务的其他信息和节点信息在这里并不加入。该函数使用的场合较少，只用于用户想建立一个服务基本信息。

```
int register_new_service(
    const char *service_name,
    char **new_service_name
);
```

参数
[in] service_name
服务的名称，最大长度不能超过 1024 个字节
[out] new_service_name
如果有重名，用于保存建议使用的新服务名字

返回值
SUCCESS：注册新服务成功
SERVICE_INFO_EXIST：新服务信息已经存在
INPUT_ERROR：输入参数错误
MALLOC_ERROR：分配空间失败
UNKNOWN_MSG_ERROR：未知错误信息

备注
上述参数都不能为空

相关接口
register_ service
unregister_service

5.1.2.3 unregister_service

该函数用来删除一个服务节点的信息，或删除整个服务资源。当指定 node_ip 时，即其不为 0，则表示要删除一个服务节点的信息。如果 node_ip 为 0，则表示要删除整个服务资源，包括所有服务节点信息和服务信息。不过，删除整个服务时，它只通知本站和缺省站点删除服务信息。如果某个站点不是存储服务信息的缺省站点，也不是本站，则可能仍然遗留有服务信息及节点信息。因此该函数不能保证删除掉所有服务信息

```
int unregister_service(
    DWORD service_pid,
```

```
    unsigned int node_ip,
    unsigned short service_port,
    int node_type,
    const char *service_name,
    const char *service_owner
);
```

参数

[in] service_pid
服务程序运行时的进程号
[in] node_ip
服务程序所在的节点 IP 地址，以 IPV4 网络地址方式传递
[in] service_port
注册服务程序的端口
[in] node_type
节点的类型，值为宏 CLIENT_NODE 或者 SERVER_NODE
[in] service_name
服务的名称，最大长度不能超过 1024 个字节
[in] service_owner
服务的所有者

返回值

SUCCESS：注销成功
INPUT_ERROR：输入参数错误
MALLOC_ERROR：分配空间失败
UNKNOWN_MSG_ERROR：未知错误信息

备注

上述参数中，node_ip 和 service_name 是必须的，实际上服务器也只以这两个参数来确定要注销的服务节点。如果 node_ip 为 0，则表示要注销整个服务，包括所有服务节点信息和服务信息

相关接口

register_ service
register_new_service

5.1.3 服务查找

5.1.3.1 find_server_by_service

该函数也是一个兼容函数。它的目的是在本站的服务器中查找一个可以提供某个服务的节点。

```
unsigned long find_server_by_service(
    char* service_name
```

);

参数

service_name

[in] 服务的名称，最大长度不能超过 1024 个字节

返回值

当查找到可以提供服务的服务者时，该函数返回最优的服务提供者的地址。如果返回值为 0，则表示没有找到可以提供服务的服务者或者执行查找失败。

备注

上述参数不能为空

相关接口

get_all_service_servers

5.1.3.2 get_all_service_servers

该函数是常用的资源查找函数，它根据指定的要查找的服务名，获得能提供所需服务资源的服务器的链表。返回链表按照优先关系排列，最前面的为最优先的服务提供者。返回的服务提供者信息中也有负载和距离信息，调用者可以根据这些信息再进行优选。

```
int get_all_service_servers(
        struct Service_List *list,
        char scale,
        struct Location_Desc **station_list,
        int node_type
);
```

参数

[in] list

要搜索的服务资源链表，指定搜索多个服务资源，则返回能提供所有服务资源的服务节点

[in] scale

搜索范围参数，参数可以为 SINGLEADDR、FEWADDR、COUPLEADDR、ABUNDANTADDR、ALLADDR

[out] station_list

返回搜索结果的参数。用于保存资源所在位置信息链表头的指针地址

[in] node_type

所要搜索节点类型，可以为 CLIENT_NODE 或者 SERVER_NODE

返回值

0：成功

其它：失败

备注

上述参数不能为空

相关接口
find_server_by_service
free_location_list

5.1.4 其它接口

5.1.4.1 free_location_list

释放链表的内存资源。

```
void free_location_list(
    struct Location_Desc* list
);
```

参数
[out] list
位置链表指针

返回值
无

备注
上述参数不能为空

相关接口
get_all_service_servers

5.2 宏定义

宏定义	宏定义值	含义
SINGLEADDR	0x00	搜索规模为 1 个地址
FEWADDR	0x01	搜索规模为 4-16 个地址
COUPLEADDR	0x02	搜索规模为 16-64 个地址
ABUNDANTADDR	0x04	搜索规模为 64-256 个地址
ALLADDR	0x08	搜索规模为 256 以上个地址
NODE_REGGI_TYPE	0	注册
NODE_UNREG_TYPE	1	注销
CLIENT_NODE	0	所有节点
SERVER_NODE	1	服务器节点

SUCCESS	0	成功
SERVICE_INFO_EXIST	-1	服务信息已经存在
INPUT_ERROR	-10	输入参数错误
MALLOC_ERROR	-11	分配内存失败
UNKNOWN_MSG_ERROR	-12	未知的错误
IDDATA	IDData	保存“站”的结构类型
ID_SIZE	16	“站”标识符字符串最大长度

5.3数据结构

5.3.1 Service_List

服务名称的装载容器链表结构。该数据结构用于查找服务提供者（任务调度）调用。查找服务提供者时，将指明要查找的服务的名字。服务也可以是文件名，即要查找有某个文件的服务提供者。如果是文件名，则必须是其在数字有机体文件系统中的全路径名，即以/dpfs 开始的文件名。该结构用于形成一多个要查找的服务名的链表。url 即用来存储服务的名字，其最大长度为 1024 个字节。

```
typedef struct Service_List
{
        char    url[SERVICE_MAX_LEN];
        struct  Service_List *next;
}service_list_st;
```

参数

url
服务的名称，名称的长度不能超过 SERVICE_MAX_LEN
next
指向链表下一个节点的地址

5.3.2 Location_Desc

这个结构是查找服务提供者返回结果的描述结构，它表示一个服务提供者。多个服务提供者通过 next 成员变量形成链表。

```
typedef struct Location_Desc
{
        IDDATA  stationID;
        unsigned int nodeIP;
        short   Location_type;  /* CLIENT_NODE or SERVER_NODE */
        short   hops;
```

```
        struct  Load_Info load;
        unsigned int public_Ip;
        short   service_port;
        struct Location_Desc *next;
}location_desc_st;
```

参数

stationID
服务提供者的站号，由两个 64 位数字的字符串用“+”衔接
nodeIP
IPv4 的网络地址
Location_type
位置类型，值为 CLIENT_NODE 或者 SERVER_NODE
Hops
服务提供者的距离，跳数
Load
服务提供者的负载信息
public_Ip
服务提供者的公网地址。当服务提供者在私网内时，该私网外的计算机无法直接访问此服务提供者。公网地址是外部计算机看到的地址
service_port
服务提供者提供服务的端口号
next
指向链表下一个节点的地址

5.3.3 IDDATA

本结构定义站编号（站标识符）的类型。在数字有机体系统中，每个数字有机体站都有一个全局唯一的编号。

```
struct IDData
{
        char id[ID_SIZE];

        IDData();
        IDData(unsigned int uid);
        IDData(char* a);
        ~IDData();

        IDData&    operator = (IDData& a);
        IDData&    operator = (char *a);
        IDDATA&    operator +(IDData &a);
        IDDATA&    operator -(IDData &a);
```

```
        operator IDDATA();

        int operator > (IDData &a);
        int operator < (IDData &a);
        int operator == (IDData &a);
        int operator == (char *id);
        int operator >= (IDData &a);
        int operator <= (IDData &a);
        int operator != (IDData &a);

        IDData& SetId(unsigned int low32);
        IDDATA& SetId(unsigned long long id1,unsigned long long id2);
        IDData& clear(void);
        IDDATA& GetMaxId(void);

        void IDDisplay(void);
        int IsZero(void);
        static int IsZero(char *);

        unsigned int GetLast32(void);
        unsigned long long    GetHigh64(void);
        unsigned    long long GetLow64(void);
        static unsigned long long    GetHigh64(char id_str[ID_SIZE]);
        static unsigned    long long GetLow64(char id_str[ID_SIZE]);

        static void GetCharId(unsigned long long id1,unsigned long long id2,char *out_id);
} __attribute__((aligned(4)));
```

参数

Id
站号字符串，由两个 64 位数字的字符串用“+”衔接，字符最大长度为 ID_SIZE

5.3.4 Load_Info

该数据结构用于描述一个服务提供者的负载信息。对于客户机作为服务提供者来说，其负载信息是无效的。cpu_usage 是一个综合参数，它用于描述整个机器各方面的繁忙程度，即不仅仅指处理器的繁忙程度。选择服务提供者是，该变量可以作为选择参考，其值越大表示其负载越重。其后三个参数分别指明内存、磁盘和网络的使用情况。内存使用情况指被使用内存占总内存的比例。磁盘使用情况指当前磁盘读写的繁忙程度，用百分比表示。网络使用情况指当前输出和输入的数据流量占可最大使用量的百分比。

```
typedef struct Load_Info
{
        int     cpu_usage;      /* cpu usage:percent */
```

```
        int     mem_usage;      /* memory usage:percent */
        int     disk_speed;     /* disk speed:KB/s */
        int     net_flow;       /* net flow:KB/s */
        unsigned int ip;        /* node ip */
}load_info_st;
```

参数

cpu_usage
CPU 的使用量，使用百分比表示
mem_usage
内存的使用量，使用百分比表示
disk_speed
磁盘的转速，单位是 KB/s
net_flow
网络流量大小，单位是 KB/s
ip
节点的 IPv4 网络地址

6 分布式并行任务编程范例

6.1 概述

数字有机体工作平台并不直接为分布式并行编程提供接口，而是为分布式并行编程框架提供底层支持，包括文件块的分布查询、文件分块的参数设置、系统站信息查询等功能。应用编程人员不应该关注这些底层的接口，而应该把目光转到下面将介绍的数字有机体远程过程调用所提供的分布式并行编程功能上；框架编写人员则可以参考数字有机体远程过程调用所展示的编程框架中与数字有机体工作平台交互的部分，选择适合的接口以完成自己的框架。

6.2 分布式并行编程框架结构示例

数字有机体系统为应用程序员提供了数字有机体远程过程调用（dosrpc）作为分布式并行编程的框架。该框架将分布式并行任务编程分解为客户端与服务器两个部分，客户端部分作为任务发起方，dosrpc 的客户端部分使用数字有机体文件系统提供的接口获取整个系统的结构以及资源的分布，然后根据这些信息将上层程序员提交的任务进行有效的分解，分解后的任务被派发至系统中指定主机上；服务端部分作为任务接受方，在接收到客户端部分发送过来的请求后，dosrpc 的服务端部分使用数字有机体文件系统提供的 IO 接口获取执行任务所需要的资源数据并执行相应的逻辑以完成任务并将结果返回给客户。

图 6.1 展示了 dosrpc 与数字有机体工作平台交互的结构示意

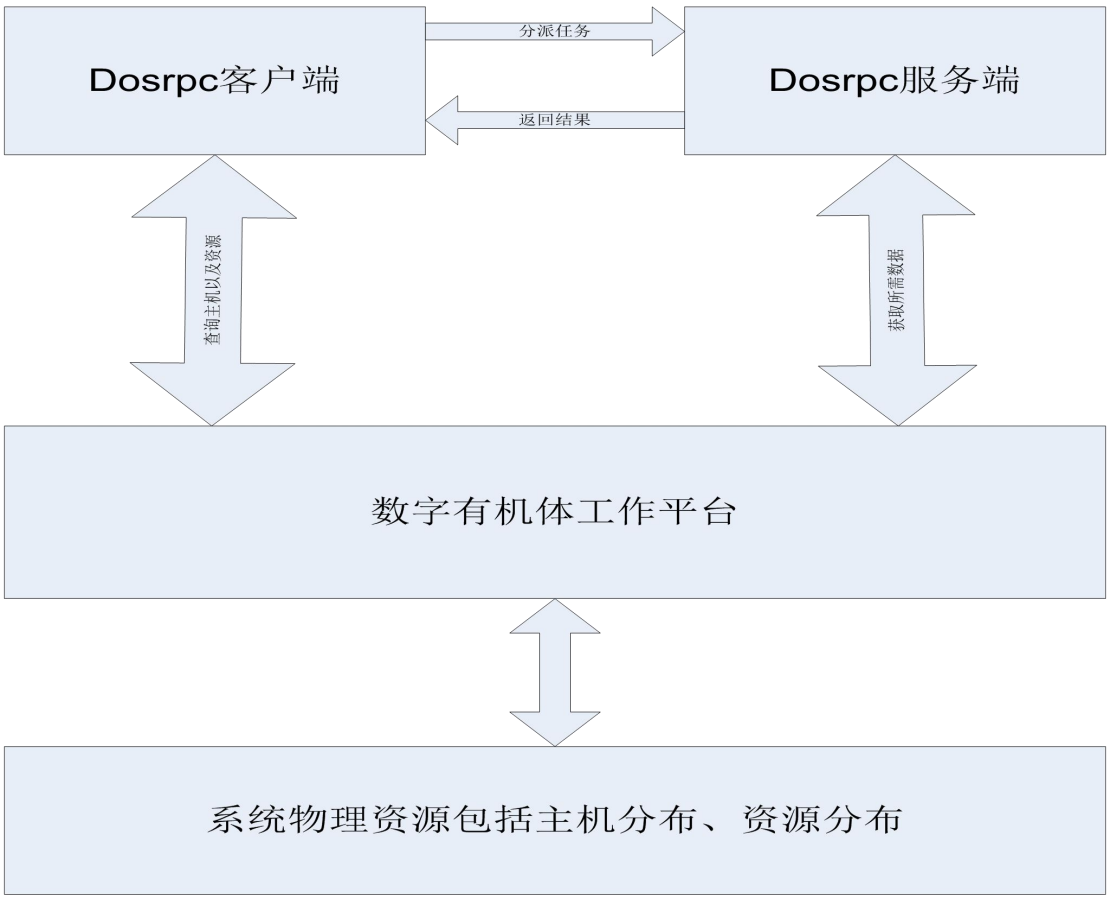

图 6.1 dosrpc 与数字有机体工作平台的交互结构

6.3 应用编程示例

6.3.1 单词统计

该应用要在大量文件中统计指定单词出现的次数。这些文件大小不一，都存储在数字有机体文件系统的某个目录下。这些文件由数字有机体系统分布式的存储在大量节点中，比如20台服务器。现在程序员将编写一个程序来快速完成指定单词出现次数的统计。

为此，程序员将定义统计文件中指定单词出现次数的远程调用函数，即编写一个远程调用描述文件（以“.x”结尾）wordcont.x。其内容如下：

```
program WORDCOUNT{
    version VERSION1 {
        u_int64_t FILE_WORD_COUNT(string word)=1;
    }=1;
}=1;
```

这个文件的格式是标准的RPC调用描述文件格式。只是这里无需指定要处理的文件，因为它将作为一个默认的参数出现。

然后程序员使用dosrpcgen生成DOSRPC程序框架，其命令形如：

```
dosrpcgen    wordcont.x
```

执行该命令后，将产生DOSRPC调用的客户端程序，即发起DOSRPC调用的应用程序，

以及处理远程过程调用的服务程序，即DOSRPC服务程序。

现在，程序员要做的事是在DOSRPC服务程序框架中实现具体统计一个文件中指定单词出现次数的函数，即如下的函数：

```
bool_t file_word_count_1_svc(struct file_block_task *file_block, char *word, u_int64_t *result, struct svc_req *rqstp)
{
    ……..
}
```

这个函数将统计的结果保存在result指针指向的整数中，系统将自动把其值作为函数结果返回给远程调用者。

然后，程序员还需设计一个函数来汇集每个文件的统计结果，从而可以作为整个数据处理的结果提供给DOSRPC的调用者。这个汇集每个文件的统计结果的函数我们将其称为结果处理函数。它的定义形式如下：

```
bool_t process_result(struct file_block_task *file_block, caddr_t res, void *arg)
```

其中第一个参数是被处理的文件或者文件分块信息，第二个参数是远程过程调用的返回值。在这里是一个整数的指针，指向的地址中保存了被处理文件中找到的指定单词的个数。第三个参数是程序员指定的指针参数。对本例子来说，可以是统计结果保存结构的指针。

结果处理函数在每个具体文件或者文件分块被处理完成后都会被调用，因此可以统计出所有的处理结果。

最后，程序员需要编写发起DOSRPC调用的代码。这段代码可以分为执行环境初始化部分、调用参数设置部分和DOSRPC调用执行部分。在执行环境初始化部分，将调用系统接口函数初始化DOSRPC环境，然后可以调用“环境参数设置”函数设置环境参数，例如设置是否开启日志功能，每台服务器最大的并行执行任务数等。然后是调用参数设置。需要设置的调用参数主要是要处理的文件集合和每个文件的最长处理时间。设置完参数后，即可调用接口函数files_dprpc_call启动和执行指定文件集的分析处理。这个函数将一直等到所有文件被处理完成。系统也提供异步调用函数，不过系统的工作原理是相同的，这里不再单独描述。

调用files_dprpc_call函数后，系统将自动完成以下工作：

1）按照指定的文件集合分解出每个具体的数据对象，这里主要是对大文件按照其分块划分为多个数据对象。

2）确定每个数据对象的存储节点集，即明确每个数据对象的处理任务可以在那些节点上执行（系统只在有数据对象的节点上处理该数据对象）。

3）控制和执行每个任务。系统将按照负载均衡的原则在合适的节点上执行每个数据对象的处理任务。每个数据对象的处理任务以一个远程过程调用的方式执行，因此执行任务的节点将获得一个远程过程调用请求。服务节点从请求中解出要处理的文件对象信息以及函数执行参数。这里的函数执行参数即要查询的单词。服务节点调用前面实现的file_word_count_1_svc函数完成任务处理，并按照RPC调用规则将结果返回给客户。客户自动调用“结果处理函数”处理返回结果。DOSRPC将并行的在系统的各台节点上执行各个任务，这样可以缩短整体执行时间。同时系统将跟踪每个任务的执行情况，如果某个任务在一台节点上执行失败了，就选择其他可以执行该任务的节点来再次执行。在所有任务都完成后，DOSRPC调用才结束。

4）以检查点文件记录每个任务执行的情况，以便发起者故障时可以恢复执行。

在files_dprpc_call函数执行结束后，从指定文件集合中统计指定单词出现次数的任务也就结束。

从测试的结果看，如果每个文件都比较大，例如是10MB以上的，则并行执行效率几乎

随节点数线性增长的。

6.3.2 数据排序

和单词统计不同，数据排序处理过程更加复杂，而且处理后的结果很大，难以汇聚到发起者节点进行再处理。因此，需要功能更强的编程接口。

其调用接口为 int files_dprpc_union_call(struct files_array files, struct rpc_union_input *inputs)，其输入参数 files 为文件列表，inputs 为输入参数列表，。其中 rpc_union_input 结构体保存了函数参数、分组参数、分组函数、分组执行模式等信息。

发送内容由分组函数而定，而分组函数根据输入参数来决定分组模式，其中分组又分为系统分组模式与自定义分组模式（本模式的分组函数需要用户自己写）。当分组完成后，其操作与一次 DOSRPC 调用相似（具体请参考上例），不同的是它会选一个最优节点处理接收到的文件列表，返回的结果会包含文件列表，继续由分组函数对其进行分组，重复上述操作。

当返回的结果没有文件列表时，被认为是最后一次 DOSRPC 调用，它的回调函数不再是分组函数，而是处理函数，其得到的结果即为所求。

根据上面的原理，我们将排序的文件名、输入参数列表作为参数传入。每一层 DOSRPC 调用的服务端都会对分组内其包含的文件进行排序，其内生成的排序文件将作为下一层 DOSRPC 调用的参数（文件列表）。这样直至最后一层 DOSRPC 调用时，服务端生成的文件就只有一个，即为排好序的所需文件。

7 常见问题解决

7.1 注意事项

- 使用登录接口

使用数字有机体开发接口时需要登录，否则会出现错误。

- 配置调度服务器

使用调度服务器时首先配置调度服务器，否则无法调度成功，安装有 vserver 或者 upd，也可以手动配置一个调度服务器。

- 释放内存资源

一些接口在使用结束后需要释放内存资源，否则造成内存泄露。

7.2 常见问题与解答

(1) 问题：登录接口中没有返回成功后的描述符，后续操作能继续吗？

答案：成功后返回描述符是常见的应用程序处理方式，数字有机体是系统程序，在系统底层

标记了登录用户名。因此，只要登录成功的用户，系统都记录了它，后续操作可以正常进行。

(2) 问题：数字有机体调度和域名服务有什么区别？

答案：首先数字有机体调度是基于服务的，而域名服务是基于主机的；其次数字有机体调度总能调度到可用的服务器用于服务，而域名服务器则无法保证。

(3) 问题：使用数字有机体调度 SDK 开发时，可以不使用 vserver 或 upd 吗？vserver 或 upd 能干些什么？

答案：使用数字有机体调度 SDK 开发时，vserver 和 upd 不是必须的，可以不使用。vserver 和 upd 也是使用调度 SDK 开发的一个具体应用，它们实现相同的功能，不同点在于 vserver 是 windows 版本的，upd 是 linux 版本的，它们的功能都是从客户端维护调度服务端的服务器地址，使数字有机体调度系统在调度时总是和“活”的服务器通信，避免与“死”的服务器通信，从而提高调度的响应速度，因此还是建议使用。

(4) 问题：本手册描述了 C 开发接口，那么数字有机体工作平台支持 WEB 开发吗？

答案：数字有机体工作平台支持 WEB 开发，程序认证登录的方式可以参阅数字有机体工作平台的用户手册。如果需要用到 C 开发接口，可以采用 JNI 等技术直接调用 C 开发接口。

(5) 问题：C 开发接口可以采用调度 SDK 开发出业务容灾的程序，WEB 开发怎么办？

答案：针对目前广泛使用 WEB 技术的情况，数字有机体工作平台提供虚拟服务的技术，可以实现 HTTP 协议的 Session 复制，并实现服务器故障后任务自动迁移，并做到负载均衡。因此，采用 WEB 开发可以更容易、更好地实现调度功能，考虑到复制的需要，WEB 开发人员需要实现 WEB 中结构的序列化。

(6) 问题：数字有机体工作平台的 B/S 的调度实现了类似进程迁移的功能，WEB 应用可以直接使用。为什么 C/S 的调度没有实现？

答案：B/S 采用的是 HTTP 协议，协议与模式是固定的，应此可以实现，而 C/S 是一种比较笼统的描述，没有固定的协议，没有固定的算法，也没有固定的编程语言，因此无法实现。但是数字有机体工作平台向 C/S 模式提供了有效的服务调度功能，并实现了数据的自动备份，因此基于它能够方便地开发出具有业务容灾的系统。