十年一日，深入成就深度
业精于专，专注成就专业

数字有机体远程过程调用
（DOSRPC-V1.6）
用户手册

成都天心悦高科技发展有限公司

2015 年 11 月

版权声明

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. 引言

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 内容简介

本文档供数字有机体远程过程调用的开发人员阅读，帮助他们使用数字有机体远程过程调用。

本文档分为 9 个章节，第一章即本章，介绍文档的相关内容。第二章是数字有机体远程过程调用的简介，以帮助用户了解数字有机体远程过程调用。第三章介绍软件的安装。第四章为外部接口介绍。第五章为 dosrpcgen 简介，介绍其使用方法。第六章为 DOSRPC 代码编写，通过两个实例进行介绍。第七章为 dosrpc_manager 的介绍及使用。第八章为编程示例与测试。第九章为出错处理，告诉用户在使用过程中出现问题时怎样解决。最后，请认真阅读并自觉遵守“最终用户许可协议”中的条款。

1.3 相关文档说明

本文档为数字有机体远程过程调用的用户手册，主要阅读对象为使用数字有机体远程过程调用的编程人员。如果对数字有机体工作平台不了解，数字有机体远程过程调用的开发人员可以参阅《数字有机体工作平台及抗毁容灾系统(DOS-X.X)开发手册》和《数字有机体工作平台及抗毁容灾系统(DOS-X.X)用户手册》。

1.4 术语

数据对象：要处理的数据实体。它可以是一个文件或者文件的一个分块，也可以是一个其他的数据存储对象。这里仅表示他是一个可以区别于其他数据对象的实体而已。

远程过程调用：本地过程调用的远程扩展。远程过程调用由两个部分组成，一个是提供调用服务的远程服务实体（当然也可以相互提供调用服务），另一个是进行过程调用的客户实体。远程过程调用以函数调用的方式被应用程序使用，但是函数的执行其实在另一台服务器上完成，这使得分布式程序的编写变得更加容易。

数字有机体远程过程调用：英文缩写为 DOSRPC，是一种面向大数据处理的远程过程调用机制。在数字有机体系统中，文件、数据库或者其他数据对象，被分布式的存储在大量的节点中。当需要处理大量数据对象时，如果能够让各个存储数据的节点处理本地存储的数据对象，然后再把结果汇总到发起处理的节点，则能通过大量节点并行处理数据的方式极大的提升数据处理的效率。数字有机体远程调用即提供这样的功能。它根据用户指定的数据对象集合，自动分解出要处理的数据对象集，并以远程过程调用的方式向存储数据对象的节点传递处理参数，并指定要进行的处理，处理结果以函数返回结果形式返回给调用者。调用者可以汇聚各个数据对象的处理结果，从而得到最终的结果。

数字有机体远程过程调用程序生成器（dosrpcgen）：基于 rpcgen 开发，可以生成 DOSRPC 的大多数代码。开发人员根据不同的需求修改生成的代码，实现自身的需求。

数字有机体远程过程调用单一模式（SINGLE-DOSRPC）：DOSRPC 的一种调用模式，一次只调用一个 DOSRPC。

数字有机体远程过程调用联合模式（UNION-DOSRPC）：DOSRPC 的一种调用模式，一次调用多个 DOSRPC，分步来执行。

1.5 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。

2.2数字有机体远程过程调用简介

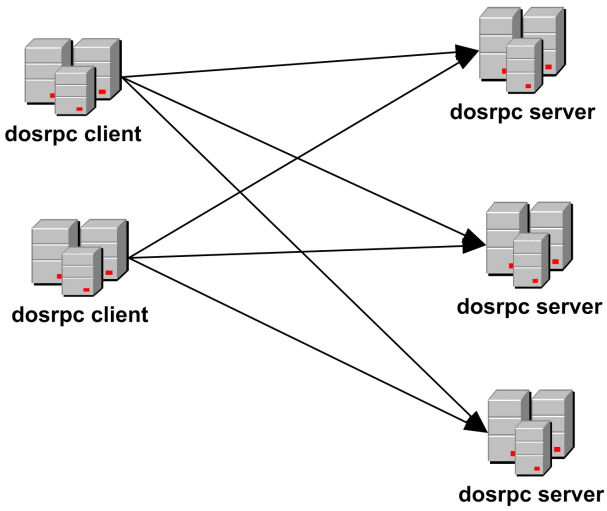

图 2-1 逻辑结构图

数字有机体远程过程调用（DOSRPC）的逻辑结构如图 2-1 所示。其主要由远程调用客户及远程调用服务器组成，每台机器既可以作为客户机进行调用发起，也可以作为服务器进

行处理与执行,每个客户机对应多台服务器，其运行环境为数字有机体工作平台。

数字有机体远程过程调用（DOSRPC）属于应用程序的基础开发库，是一种通用的分布式数据处理框架。通过现有的 C API 接口能够有效地实现各种分布式、并行、联合，以及级联的数据与文件处理。缺省情况下，DOSRPC 只适用于数字有机体工作平台。如果想要适应其他的平台，开发人员需要额外实现任务分解和任务调度。

DOSRPC 开发库支持数据和计算的分布性、并行性处理的能力，以及协同工作的特性，使它天然地具备大数据的处理优势。特别是针对磁盘 IO 密集型，计算密集型的应用，DOSRPC 具备极大的处理能力。

数字有机体具备大规模存储的特性，同时将大的文件自动分割成多个文件块，并把这些文件块均匀地分散存储到多个独立的数字有机体服务器。基于数字有机体文件块的特性，再结合 DOSRPC 的分布式大规模并行处理框架，利用 DOSRPC 库开发的服务端应用程序能够同时在多个数字有机体服务器上协同处理，最终把处理的结果返回给 DOSRPC 库开发的客户端应用程序。此外，数字有机体工作平台还支持文件块的多副本机制，使得基于 DOSRPC 开发的应用程序具备故障后重试的功能，大大地降低了大任务因为小故障而失败的可能性。

2.3 主要功能和特点

欢迎您使用 DOSRPC 进行远程过程调用。本软件有如下功能和特点：

- ✓ 使用简单：使用 dosrpcgen 可以生成大部分代码；
- ✓ 执行效率高：C/C++编写，能更高效的访问文件；
- ✓ 独立性高：各个应用相互之间没有关联，也不统一管理；
- ✓ 对分布式数据处理：依赖数字有机体工作平台及数字有机体工作库，对分布式数据进行处理；
- ✓ 支持多种模式：客户端包括同步与异步调用，服务方包括单线程与多线程接收；
- ✓ 支持数字有机体远程过程调用联合模式：用户可以一次调用多个 DOSRPC 进行分步处理，可根据需要组织处理步骤，可分为任意多个步骤来完成处理；
- ✓ 图形化界面任务跟踪：DOSRPC 提供了一个图形化显示工具，显示每一个分组的情况与分组内任务的详细信息。

3. 软件安装

3.1 系统运行环境要求

安装本软件前，请确认计算机是否满足以下系统要求：

软件要求：数字有机体工作平台

3.2 软件安装步骤

一般情况下，数字有有机体系统发行版中已包含 DOSRPC 的安装包。如果没有，您可以从成都天心悦高科技发展有限公司购买获得。

准备安装前检查目标计算机是否符合“系统运行环境”的要求，如果是，请正确安装 RPM 包，安装的命令是“rpm -ivh --nodeps dosrpc-v1.6-1.x86_64.rpm”。卸载的命令是“rpm -e --nodeps dosrpc-v1.6-1.x86_64”。

4. 外部接口介绍

DOSRPC 在客户端包含两种调用方式，即同步调用与异步调用。不同的调用方式其调用接口也不一样；DOSRPC 的服务端支持单线程接收的方式，也支持多线程接收的方式。一次性调用一个 DOSRPC，我们称之为数字有机体远程过程调用单一模式（SINGLE-DOSRPC），为了满足某些复杂的应用需求（如数据排序），需要一次性调用多个 DOSPRC 分步来进行处理，这里我们提供了另一种调用方式，即数字有机体远程过程调用联合模式（UNION-DOSRPC）。两种调用模式下客户端的调用接口有所不同，而环境设置的参数的接口两者都要用到。（以下若没有特殊标示，DOSRPC 默认为数字有机体远程过程调用单一模式）

4.1 DOSRPC 运行环境接口

这部分接口用于设置 DOSRPC 调用的运行环境。它包括环境初始化函数、一系列环境变量设置函数和销毁环境的函数。

4.1.1 DOSRPC 运行环境初始化函数

每个进程在调用 DOSRPC 的执行函数前，需要先初始化 DOSRPC 模块。该接口函数定义为：

int init_dosrpc(bool fail_exit, rpcprog_t prog, rpcvers_t vers,uint32_t busy_stat_timeout)

该函数的 fail_exit 参数用于表明执行 RPC 调用时，如果有某个任务无法执行是否结束整个 RPC 调用的执行，为真表示要结束，否则表示不结束。prog 和 vers 是本进程使用的 RPC 程序号和版本号。整个进程的 DOSRPC 调用都使用它们。

参数 busy_stat_timeout 为当 RPC 服务器报忙时，客户方再次重试的实际间隔。该间隔越短则客户越频繁的尝试向服务器发送任务，但必然增加系统开销。

4.1.2 DOSRPC 运行环境设置函数

DOSRPC 运行环境包括以下各个方面的内容。

4.1.2.1 跟踪日志输出设置

程序库提供详细记录 DOSRPC 执行过程到日志文件的功能。程序员可以设置记录日志文件的名称，也可以关闭或者开启记录日志功能。因此提供了以下三个函数。

1）extern void SetRunLogFile(char* file_path);

该函数设置日志文件的名称。如果程序员不设置名称，则默认将日志文件放到/var/log 目录下。文件的名称采用“函数号_主机名_进程号_调用任务编号.log”。注意：函数并不检查文件名的合法性。如果文件名是非法的，则将不会产生日志文件，相当于关闭了日志功能。

2）extern void TurnOnLog(void);

开启日志记录功能。这样执行 DOSRPC 时，程序向指定的日志文件输出运行日志，或者向默认日志文件输出运行日志。

3）extern void TurnOffLog(void);

关闭日志功能。即使用户设置了运行日志文件，也不会输出运行日志。

4.1.2.2 是否记录执行检查点

程序库在执行 DOSRPC 过程中，将自动建立和更新检查点文件。检查点文件记录了 DOSRPC 的各个任务信息及任务当前执行的情况。通过读取该检查点文件，可以实时显示执行的进度，也可以在执行节点故障时，在其他节点上重启 DOSRPC 调用的执行。不过系统并不保存中间任务的执行结果。因此，在使用恢复 DOSRPC 执行功能时，应用程序应当自己考虑如何保存中间结果，也应当自己考虑任务是否能够重复执行。

程序库提供一个接口函数来设置是否要建立和更新检查点文件。

extern void set_checkpoint_flag(int flag);

当传入参数为非 0 值时表示要建立和更新检查点文件，为 0 时表示不建立和更新检查点文件。

4.1.2.3 有任务执行失败是否结束 DOSRPC 执行

如前所述，在执行 DOSRPC 调用时，如果某个任务执行失败，则需要根据程序员的需求来决定是否继续执行其他未执行的任务。默认情况是继续执行其他未执行的认为。提供两个接口来设置和获取当前设置。

1）extern void set_end_execute_when_fail(int new_val);

设置有任务执行失败是否结束 DOSRPC 执行的环境变量。传入参数为要设置的值。为非 0 值时表示有任务执行失败则结束 DOSRPC 执行，反之则继续执行。

2）extern int get_end_execute_when_fail(void);

获取当前的设置，返回的含义和设置是相同。

4.1.2.4 发送调用请求线程数

当系统由大量节点构成，且一个 DOSRPC 调用涉及大量节点时，用单个线程来向大量节点发送请求将无法满足需求，因此需要多个线程并行地发送调用请求。编程库提供两个函数，一个函数用于设置最大的发送调用请求线程数，另一个函数则用于获得当前的设置值。如果没有设置，默认的最大发送调用请求线程数为 10。

1）extern void set_send_request_thread_num(uint32_t thread_num);

设置发送调用请求的最大线程数。注意：太大的值没有意义，甚至可能因创建大量线程而耗尽系统内存，因此建议该值不要超过 50 个。在系统节点数量众多时可以大一些。

2）extern uint32_t get_send_request_thread_num(void);

获得当前的发送调用请求最大线程数。

4.1.2.5 节点忙状态超时时间

当服务器采用多线程并行服务时，如果服务器太忙，则可以向客户返回“服务器忙”的消息。这时，客户将暂停向该服务器发送任务，直到服务器返回某个任务的调用结果，或者超过一定的时间。节点忙状态超时时间即这个超时时间。建议将该值设置的大一些，以避免无谓的重试。默认超时时间为 20 秒。

1）extern void set_server_busy_stat_timeout(ulong busy_timeout);

设置节点忙状态超时时间值。函数没有检查传入的值。如果为 0 则没有时间间隔，程序将立即重试。时间单位为秒。

2）extern ulong get_server_busy_stat_timeout(void);

获得当前的节点忙状态超时时间。

4.1.2.6 一个服务节点的最大连接数

在执行一个 DOSRPC 任务时，将占用一个到服务器的连接（即 RPC 的 CLIENT）。程序库提供接口设置和获取一个服务节点允许的最大连接数。默认一个服务节点的最大连接数为 10。如果服务器是多核的，且单个任务处理工作量小，则可以设置大一些，否则应当小一些。如果服务方采用单线程服务，则应当设置为 1 或者 2。

extern void set_max_parell_client_on_node(uint32_t client_num);

extern uint32_t get_max_parell_client_on_node(void);

4.1.2.7 接收调用返回结果的最大线程数

当使用多个线程并行的发送调用请求时，也就会有大量的调用会并行的返回，这时就需要多线程来接收调用返回结果。这里设置允许的最大线程数。默认值为 10，即发送调用请求的默认最大线程数。

1）extern void set_max_parell_recv_threads(uint32_t thread_num);

设置允许的最大线程数。程序库没有检查传入参数，因此不要设置为 0。

2）extern uint32_t get_max_parell_recv_threads(void);

获取当前设置的最大线程数。

4.1.3 销毁运行环境函数

当不再需要指向 DOSRPC 调用时，则可以销毁运行环境。注意，在进程结束时请销毁运行环境，否则某些内存跟踪程序将报告存在内存泄露。

extern void exit_dprpc(void);

该函数没有参数，也没有返回值。注意：在销毁运行环境后，不要希望能够通过重新初始化运行环境来继续执行 DOSRPC 调用。换句话说，如果不是确定不再执行 DOSRPC 调用，否则不要销毁运行环境。

4.2客户方同步调用接口

程序库的 DOSRPC 同步调用接口函数是：

char* files_dprpc_call(struct files_array, rpcproc_t proc, xdrproc_t arg_xdr_proc, caddr_t arg, xdrproc_t res_xdr_proc, resultproc_t eachresult,void* reault_arg, struct timeval timeout);

该函数的第一个参数是一个文件集合，即要被处理的文件的集合。第二个参数为要调用的过程号，第三个参数为流化调用参数的 XDR（外部数据表示，参见 RPC 协议文档）函数指针。第四个参数为函数调用参数的指针。第五个参数是流化调用返回结果的函数指针。第六个参数为任务执行结果的回调函数，用于处理一个文件分块的函数执行结果，相当于 MapReduce 的 reduce 函数。第七个参数是处理结果函数的额外参数，将作为回调函数的第三个参数。最后一个参数为单个任务执行的超时时间，建议设置为最大执行时间的两倍。

文件集合的结构定义如下：

```
struct files_array{
    char **m_file_names;
    int    m_name_num;
};
```

files_array 结构的第一个成员为文件名指针数据，每个成员为一个文件名的指针。第二个成员为数组中成员的个数。考虑到实际应用需要，支持文件名匹配功能。即可以用星号(*)表示任意字符，暂不考虑支持其他匹配符。这样，如果某个文件名为“/dpfs/dir1/f*”，则表示/dpfs/dir1/目录下的所有以 f 开始命名的文件。name_fum 是给出的文件名个数，不是实际匹配的文件个数。

参数 resultproc_t eachresult 用于处理每个文件分块的远程调用返回结果，可以认为是回调函数。resultproc_t 的定义如下：

typedef bool_t (*resultproc_t)(struct file_block_task *file_block, caddr_t result, void *user_arg);

第一个参数为执行远程调用的文件块的结构指针。第二个参数为远程调用返回的结果，即在一个文件分块上执行远程调用函数得到的结果。第三个参数是程序员需要传入的其他参数的结构指针，这样程序员可以为处理结果传入更多的参数。

该回调函数返回的值将影响程序库是否继续执行对应的 DOSRPC 调用。如果返回值为假，则所属的 DOSRPC 调用将继续执行，如果返回值为真则所属的 DOSRPC 调用将停止执行。但这并不表示该回调函数就不再被调用。停止过程是多线程竞争完成的，因此仍然可能再次调用该回调函数。

每个文件块的结构如下所示。

```
struct file_block_task{
    char*     m_file_name;     //文件名称
    uint64_t  m_file_pos;      //块在文件中的起始位置, 以字节计算。
    uint64_t  m_block_size;    //块的大小，以字节计算。
};
```

函数调用参数、函数返回结果的 XDR 流化函数可以让 dosrpcgen 程序自动生成。程序员也可以自己按照 RPC 协议规范编写这些函数。

现在的问题是程序员要实现调用 files_dprpc_call 的其他参数需要做一些繁琐的编程工作。为此，随同程序库提供了一个程序，即 dosrpcgen。该程序根据用户的定义自动生成程序框架和一些程式化的函数。这样，程序员将可以方便的编写程序。当前，dosrepgen 只能用于生成客户方同步调用，服务方单线程服务的程序。

4.3客户方异步调用接口

当程序员不希望长时间等待 DOSRPC 完成执行时，可以使用程序库提供的异步调用接口。这时，DOSRPC 执行函数不等待 RPC 调用执行完成，即不阻塞线程运行，而是在启动任务后就返回一个调用句柄，即 DPRPC_HANDLE。程序员可以使用该句柄获得 DOSRPC 调用执行的情况。要注意的是，程序库没有严格校验句柄的有效性。当使用一个非 0 值的无效句柄调用函数时，将使程序产生段错。

1 ） extern DPRPC_HANDLE files_dprpc_call_async(struct files_array files, rpcproc_t proc, xdrproc_t arg_xdr_proc, caddr_t arg, xdrproc_t res_xdr_proc, resultproc_t eachresult,void* result_arg, struct timeval timeout);

该函数和 files_dprpc_call 的参数是相同的，不同的是，该函数不等待 DOSRPC 执行完成，而是快速返回该 DOSRPC 调用的句柄。后面的函数将使用这个句柄来查询信息。在确定 DOSRPC 执行完成后，应当释放该句柄。

2）extern int check_dprpc_call_end(DPRPC_HANDLE rpc_handle);

该函数用于检查 DOSRPC 任务是否执行完成。返回 0 表示还没有完成，返回 1 表示 DOSRPC 任务已经执行结束，返回-1 则表示句柄是无效的。

3）extern int cancle_dprpc_call(DPRPC_HANDLE rpc_handle);

该函数用于强制结束一个 DOSRPC 调用的执行。除非传入的句柄是无效的，否则将结束句柄对应的 DOSRPC 调用的执行，并返回 0，否则返回-1。

4）extern int get_dprpc_call_end_reason(DPRPC_HANDLE rpc_handle);

该函数用于取得 DOSRPC 调用结束的原因。返回-1 表示句柄无效，返回-2 表示对应的 DOSRPC 调用还没有结束，其他返回值为结束原因。其值对应关系如下：

```
enum rpc_end_reason{
    RPC_END_SUCCESS=0,     //成功结束
    RPC_END_SOME_FAILD,    //部分任务执行失败
    RPC_END_ALL_FAILD,      //全部任务执行失败
    RPC_USER_END,          //回调函数要求结束
    RPC_ONE_FAILD,          //配置要求有任务结束就结束 RPC 执行
    RPC_GROSSERROR    //重大错误
};
```

如果是程序员主动调用 cancle_dprpc_call 函数结束执行的，其原因为 RPC_USER_END。

5 ） extern int get_dprpc_task_info(DPRPC_HANDLE rpc_handle, uint32_t *total_task_num, uint32_t *sucess_task_num, uint32_t *fail_task_num);

该函数用于获取正在执行的 DOSRPC 调用的进度情况。后三个参数都用来返回信息，即分别返回总任务数、成功执行任务数和失败任务数。正在执行和未执行的任务数是总任务数减去成功和失败的任务数。如果需要获得详细的执行情况，应当读取其最新的检查点文件来获得。

6）extern void free_dprpc_handle(DPRPC_HANDLE rpc_handle);

该函数释放 files_dprpc_call_async 函数返回的句柄。注意：在句柄对应的 DOSRPC 执行还没有结束前不能释放该句柄，否则将导致程序段错；而在 DOSRPC 执行结束后必须释放该句柄，否则将造成内存泄露。

4.4客户方联合 DOSRPC 调用接口

当程序员希望一次性调用多个 DOSRPC 分步来执行时，可以考虑使用联合 DOSRPC 模式。他的下一次调用会对本次调用返回的结果（文件链表）进行分组，作为下一次调用的参数，所以它需要一个分组函数。而分组函数又包含两种方式：自定义分组与周期分组。根据分组函数的方式不同，所需接口也不太一样。以下是其外部接口介绍：

1) int files_dprpc_union_call(struct files_array files, struct rpc_union_input *inputs);

该函数为联合 DOSRPC 调用接口，其输入参数 files 为文件列表，inputs 为输入参数列表，其返回结果成功为 0，失败为非 0。

其中 rpc_union_input 的结构体定义如下：

```
struct rpc_union_input{
        rpcproc_t               m_proc;             //函数号
        xdrproc_t               m_arg_xdr_proc;     //调用参数的 XDR 函数指针
        caddr_t                 m_arg;              //函数调用参数
        xdrproc_t               m_res_xdr_proc;     //返回结果的 XDR 函数指针

        //next two item only be used when m_next is NULL.
        dp_resultproc_t         m_result_fun;       //结果处理函数
        void                    * m_result_arg;     //结果处理函数的参数

        //if m_next not NULL, will set the group info.
        groupingproc_t          m_grouping;         //分组函数
        void                    *m_grouping_arg;    //分组函数的参数
```

```
int                 m_grouping_mode;            //分组执行模式
int                 m_grouping_mode_arg[2];     //分组执行模式的参数
enum rpc_node_mode m_node_mode;                 //查找最优节点的模式

struct timeval          m_timeout;              //单个任务的超时时间
struct rpc_union_input  *m_next;
};
```

其中 rpc_node_mode 的定义如下：

```
enum rpc_node_mode{
    RPC_NODE_CAP = 1,//根据节点能力值获取最优节点
    RPC_NODE_FILE_NUM = 2,//根据节点所包含的文件数获取最优节点
    RPC_NODE_ALL = 3 //综合以上两种模式获取最优节点
};
```

以下接口在自定义分组编程中使用：

2) void set_grouping_number_pointer(void *pointer, void *rpc_handle);

该函数为用户保存自定义分组信息，参数 pointer 为自定义分组信息句柄，rpc_handle 为执行控制结构句柄。

3) void *get_grouping_number_pointer(void *rpc_handle);

该函数用于获取用户保存的自定义分组信息，输入参数为执行控制结构句柄。

4) GROUP_HANDLE create_group_with_files(struct file_list *files, void* rpc_handle);

该函数的目的是创建分组，并添加文件到分组，files 为文件列表，rpc_handle 为执行控制结构句柄，其返回一个分组信息句柄。

5) int add_files_to_open_group(struct file_list *files, GROUP_HANDLE group, void* rpc_handle);

该函数用于增加文件到一个已存在的分组，参数 files 为文件列表，group 为分组信息句柄，rpc_handle 执行控制结构句柄。其返回值成功为 0，失败为非 0。

6) int add_files_and_close_group(struct file_list* files, GROUP_HANDLE group, void *rpc_handle);

该函数用于增加文件到一个已存在的分组并结束分组，参数 files 为文件列表，group 为分组信息句柄，rpc_handle 为执行控制结构句柄。其返回值成功为 0，失败为非 0。

7) int create_group_task(struct file_list *files, void* rpc_handle);

该函数用于新建一个分组，添加文件到分组，并组束分组。输入参数 files 为文件列表，group 为分组信息句柄，rpc_handle 为执行控制结构句柄。其返回值成功为 0，失败为非 0。

4.5服务程序实现

4.5.1 基本服务程序编写

如果仅仅实现一个单线程的服务程序，DOSRPC 的服务程序和传统的 RPC 服务程序几乎是相同的。而且 dosrpcgen 可以直接根据 RPC 定义文件生成单线程的服务程序框架。程序员只需编写具体的服务实现函数即可。下面仍然描述出这些不同的地方，以便程序员能够更好的理解和编写程序。

1）如何知道当前要处理的是哪个文件块？

如前所述，一个 DOSRPC 调用将根据指定的文件集分解为许多个具体文件块的 RPC 调用。每个具体的文件块用结构 struct file_block_task 表示。它包括文件的名称、起始位置和要处理的数据长度。如果他是一个完成的文件，则起始位置为 0，长度为文件长度。如果他是某个大文件的分块，则起始位置为分块在大文件中的起始位置，大小为文件块的大小。该结构将随着 RPC 调用参数传递到服务程序。服务程序在使用 svc_getargs 函数获得 RPC 调用参数后，再调用 svc_getargs 函数来获得 file_block_task 结构。file_block_task 结构的 XDR 流化函数由程序库提供，即 xdr_file_block_fun。下面代码是获得文件块信息的示例。

```
struct file_block_task *file_block = NULL;
file_block = (struct file_block_task*)malloc(sizeof(struct file_block_task));
file_block->m_file_name = (char*)malloc(1024);
if(!svc_getargs(transp, (xdrproc_t)xdr_file_block_fun, (caddr_t) file_block)){
    svcerr_decode (transp);
    return ;
}
```

2）服务函数增加的参数

和传统 RPC 不同的是，服务函数的参数不仅包括客户调用传入的参数，还包括要处理文件块的信息，即 struct file_block_task 的指针。下面是一个名称为 sample_fun 的服务函数的示例。

```
int _sample_fun_1 (struct file_block_task *file_block, sample_fun_1_argument *argp, void *result, struct svc_req *rqstp)
{
    return (sample_fun_1_svc(file_block, argp->word, argp->time, result, rqstp));
}
```

第一个参数即使增加的文件块信息。后面的参数都是传统服务函数具有的参数。

3）程序编写的其他注意事项

注意事项：DOSRPC 的客户只使用 TCP 协议作为调用网络协议，因此服务方必须创建一个基于 TCP 协议的服务进程，否则客户无法和服务器通信。

4.5.2 多线程服务程序编写

在当前的多核多 CPU 服务器上，单线程的服务程序无法充分利用服务器的性能，因此编写多线程服务程序是常见的需求。无论是 rpcgen 还是 dosrpcgen 都不能生成一个多线程的服务程序框架。因此需要程序员按照如下说明进行编写。

4.5.2.1 DOSRPC 程序库提供的支持

在多线程并发运行的情况下，如果同时接纳大量的任务，则可能造成服务器过载，或者导致大量任务积压。因此应当快速判断是否还能接纳任务。DOSRPC 程序库提供服务程序向客户报“服务器忙”的错误消息的功能。当客户收到“服务器忙”的消息时，将暂停向服务器发送请求，并尝试在其他可以执行任务的节点上执行任务。发送“服务器忙”错误消息的函数为：

void svcerr_systembusy (SVCXPRT *);

服务程序可以调用该函数通知 DOSRPC 执行者当前本服务器忙。

DOSRPC 程序库提供了简单地根据服务器 CPU 核数判定是否可以接受任务的功能。它要求程序员设定每个 CPU 核可以同时处理的任务数。然后在接纳一个任务时必须调用 can_accept_task 函数来判定，只有在函数返回真时才接纳任务，否则立即调用 svcerr_systembusy 函数通知客户，并丢弃该任务。在任务处理完成后，再调用 task_end 函数通知程序库该任务已经结束。

1) extern void set_max_server_a_cpu(int32_t num);

该函数设定每个 CPU 核可以同时处理的任务数。传入参数为要设定的值。注意不要设置为 0。默认值为 1。

2) extern int can_accept_task(SVCXPRT *xprt);

该函数用于判定是否接纳新任务，并处理接纳任务要进行的工作。在多线程服务时，必须调用该函数来接纳任务，否则可能在程序库中产生异常段错。该函数的参数为要接纳任务的服务对象。当服务程序同时接纳的任务数超过 CPU 核数和每个 CPU 核可以同时处理的任务数的乘积时，该函数返回 false；否则该函数进行接纳任务处理，并返回真。

3) extern void task_end(SVCXPRT *xprt);

该函数用于任务结束时通知程序库处理任务结束时的工作。如果一个任务被接纳，则必须调用该函数来通知任务处理已经完成，否则也会产生异常。

4) int get_cpu_num(void);

该函数用来获取 CPU 的个数。

5) void set_need_collect_node_info(int new_val)

该函数用来设置是否收集节点性能指标和负载（默认是）。

6) void set_task_type(int use_type, int *old_type)

该函数设置任务的类型，包括 CPU 型任务、磁盘型任务和综合型任务。调度模块根据任务偏向的类型来计算性能指标。

7) void set_use_current_load(int new_val)

该函数是在计算节点性能时设置是否计算当前的负载值（默认是）。

4.5.2.2 多线程服务程序实现注意事项

传统的 RPC 程序库并没有很好处理多线程服务问题。如果不使用很底层的接口函数，而想在 dosrpcgen 产生的单线程服务框架的基础上修改为多线程的服务程序，则必须注意一下问题。

1）通信守护线程创建和运行

使用 dosrpcgen 生成服务程序时，在程序名_svc.c 文件中有服务程序的 main 函数。该函数使用 rpc 程序库的 svctcp_create、svc_register 和 svc_run 来创建和启动通信守护线程，即服务程序的主线程。

在 svc_run 函数中，服务程序将监听客户的连接，并监听每个连接上到来请求的事件。在连接有请求到达时，svr_run 函数将接收请求并调用注册的回调函数（即服务程序）处理请求。注意，svr_run 是在单一的线程中运行的，因此他串行的处理每个连接。如果每个连接上的服务程序阻塞了，则整个服务程序都会被阻塞。这就是单线程的服务框架。

为了将其改写为多线程的服务程序，需要在回调函数中创建线程来处理具体的请求，而不是等待请求处理完成。当然，也可以使用线程池方式。

2）对 RPC 程序库的修改

要注意的是，传统的 RPC 程序库并没有很好处理多线程服务问题。如果使用 svc_run 并采用多线程服务，将出现诸如无法接收客户请求、发送响应时段错等情况。为此，随同 DOSRPC 程序库，也发布了修改后的 tirpc 程序库（一个有名的传输无关的 RPC 程序库），编译多线程服务程序时必须连接修改后的 tirpc 程序库。

3）严格调用 can_accept_task 和 task_end 函数

如前所述，应当严格调用 can_accept_task 和 task_end 函数。常见的用法是在回调函数的开始就调用 can_accept_task 函数决定是否接纳新任务，如果不能接纳可以直接向客户报错，然后再创建线程来处理请求。在处理请求的函数中调用 task_end 函数来通知任务结束。下面的代码是一个简单的示例。

下面是服务回调函数的示例。这是一个字数统计函数的例子。

```
struct call_proc_arg{ //向线程传递参数的结构
    struct svc_req *rqstp;
    SVCXPRT *transp;
};
static void wordcount_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    struct call_proc_arg *call_arg = NULL;
    pthread_t pthread_id;
    int ret = 0;
//首先判断能否接纳任务
    if(!can_accept_task(transp)) {
        svcerr_systembusy(transp);
        return ;
    }
```

```
//创建线程来处理客户请求。
    call_arg = (struct call_proc_arg *)malloc(sizeof(struct call_proc_arg));
    call_arg->rqstp = rqstp;
    call_arg->transp = transp;
    ret = pthread_create(&pthread_id, NULL, process_call_fun, call_arg);
    if(ret == 0) {
        pthread_detach(pthread_id);
    }else{//创建失败，因已经成功调用 can_accept_task 函数，这里必须调用 task_end 函数。
        task_end(transp);
        svcerr_systemerr (transp);
    }
    return;
}
```

下面是处理函数的示例：

```
void *process_call_fun(void *arg)
{
    //获取参数
    //进行处理；
    //最后线程退出前调用 task_end 函数
}
```

当然，程序员也可以使用线程池来处理请求，而且同样要求在接纳任务时调用 can_accept_task 函数，在请求处理结束时调用 task_end 函数。

5. dosrpcgen 简介

dosrpcgen 根据程序员定义的远程过程调用函数（X 文件定义），生成 DOSRPC 程序代码。不过，如前所述，它只能生成客户端同步调用，服务方单线程服务的程序代码。

5.1 RPC 定义文件

程序员定义远程过程调用函数的文件称为 X 文件，它的名字以“.x”结尾。其格式如下所示：

```
program WORDCOUNT{
    version VERSION1 {
        u_int64_t FILE_WORD_COUNT(string word)=1 /* method_no */;
        bool_t FILE_WORD_FIND(string word)=2          /* method_no */;
    }=1;
}=3 /* program_no */;
```

其中，“program”、“version”为保留字。“program”定义 RPC 程序的名称。用大括号包含所有的程序定义。大括号后的“=”定义程序的编号。该编号在一个系统内要求是唯一的，否则将产生冲突。

“version”部分定义程序的一个版本，大括号后为其版本号。一个程序可以有多个不同的版本。RPC 系统以程序号和版本号作为区分函数的标识之一。

在一个程序版本内，可以定义多个函数。要注意的是，定义函数的参数和返回值类型都必须使用 RPC 的 XDR 数据类型。这些类型定义在 rpc/xdr.h 中。例如布尔型变量需要使用 bool_t，而不是“bool”。一个函数可以有多个参数。函数定义最后的等号后为函数的编号。在同一个版本下的不同函数其编号应当是不同的。RPC 系统以“程序号、版本号和函数号”作为函数的唯一标识。函数的名称一般使用大写，不过生成程序中都会转换为小写表示。

注意：每项定义结束后需要使用分号。这就像编写 C 程序。另外 DOSRPC 只支持一组“程序号、版本号和函数号”，定义时请注意。program_no 示例处为 3，意思是使用数字 3 代表一组程序对（客户端和服务端程序），不属于同一组的程序，应该定义不同的数字，否则，程序运行时将产生异常。method_no 标记为函数（也叫方法）的代号，也是采用数字标记的，同一组程序中，应该使用不相同的数字来标记不同的函数。

5.2 生成的 DOSRPC 代码文件

在编写完成 RPC 定义文件后，即可使用 dosrpcgen 生成程序代码。下面以 test.x 为例进行说明。test.x 文件的内容如下：

```
program TESTPROG{
    version VERSION{
            int test_fun(int,string)=1;                    服务器的处理函数
    }=1;                                                   版本号
}=3;                                                       程序号
```

生成代码文件的使用方法：

./dosrpcgen test.x

“test.x”是 RPC 定义文件。dosrpcgen 将根据该文件生成代码。成功执行命令后，将以定义文件名为前缀，生成如下这些文件：

test.h：该文件包含了服务器与客户端程序变量、常量、类型等定义，是公用的头文件。

test_client.c：该文件为 RPC 客户端程序文件，生成了 main 函数和 test_fun_1 函数。

test_server.c：该文件为服务程序中实现具体服务函数的文件。这里定义了 test_fun_1_svc 函数，即 test_fun 函数的远程执行函数。不过，程序必须自己编写函数的实现代码。dosrpcgen 仅生成了函数实现框架。

test_svc.c：该文件是服务程序的主程序，它定义了服务程序的主函数、服务请求处理函数等。如果仅需要实现一个单线程的服务程序，则无需修改该文件。如果需要编程多线程服务的文件，则需要根据前述“多线程服务程序”章节的要求进行改写。

test_xdr.c：该文件包含了客户机和服务器所需的 XDR 流化函数，一般不用修改。要注意的是，如果所有函数都只使用一个参数，则不会生成该文件。

Makefile.test：该文件用于编译所有客户机、服务器代码。程序员可以根据该文件编写自己的 Makefile 文件。

6. DOSRPC 代码编写

本节分两部分介绍：数字有机体远程过程调用单一模式（SINGLE-DOSRPC）与数字有机体远程过程调用联合模式（UNION-DOSRPC）。

它们分别通过两个例子来进行介绍，其中 wordcount 例子是用于介绍数字有机体远程过程调用单一模式，包括客户端的同步发送与异步发送的代码编写，也包括服务端单线程接收与多线程接收的代码编写。而 ordring 例子是用于介绍数字有机体远程过程调用联合模式，包括客户端系统分组方式的编写与用户自定义分组方式的编写。

6.1 数字有机体远程过程调用单一模式（SINGLE-DOSRPC）

如前所述，使用 dosprcgen 生成代码时，它只能生成客户端同步调用，服务方单线程接收的程序代码，且是 DOSRPC 单一调用模式。这里先介绍由 dosrpcgen 默认生成的代码改写来的例子，然后介绍如何让客户端异步调用的例子，最后介绍如何让服务端多线程服务的例子。

DOSRPC 的安装软件里含有一个例子，文件夹名为 wordcount，下面对这个例子进行分析，介绍 dosrpcgen 程序的编写。

这个 DOSRPC 例子的目的是为了在数字有机体文件系统里，根据所指定的路径下的文件名集合，从中获取某一个单词出现的总的次数（其指定的文件分布在数字有机体工作平台中且只需给出路径即可，语法上支持‘*’号）。

在文件夹 wordcount 里含有一个“.x”文件，使用 dosrpcgen（/dosrpcgen wordcount.x）生成 wordcount.h、wordcount_client.c、wordcount_server.c、wordcount_svc.c、Makefile.wordcount 这五个文件（具体文件代码如下所示），由于 wordcount.x 里所设定的服务器处理函数只有一个参数，没有生成 wordcount_xdr.c 这个文件。这五个文件具体内容如下所示。

Wordcount.h:

```
* Please do not edit this file.
 * It was generated using rpcgen.
 */


#ifndef _WORDCONT_H_rpcgen
#define _WORDCONT_H_rpcgen


#include <rpc/rpc.h>
#include <dprpc.h>
#include <pthread.h>
```

```
#ifdef __cplusplus
extern "C" {
#endif



#define WORDCOUNT 3
#define VERSION1 1


#if defined(__STDC__) || defined(__cplusplus)
#define FILE_WORD_COUNT 1
extern  bool_t file_word_count_1_svc(struct file_block_task *file_block, char *, u_int64_t *, struct svc_req *);
extern int wordcount_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);


#else /* K&R C */
#define FILE_WORD_COUNT 1
extern  bool_t file_word_count_1_svc();
extern int wordcount_1_freeresult ();
#endif /* K&R C */


#ifdef __cplusplus
}
#endif


#endif /* !_WORDCONT_H_rpcgen */
```

Wordcount_client.c:

```
/*
 * This is sample code generated by rpcgen.
```

```
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "wordcont.h"
#include <dprpc.h>



//Process the each file block's call function returned value.
//The argment res is return value.
//The argment arg is the ponter of process result argment in files_dprpc_call.
//Return false to let dprpc call continue, return true for terminate the dprpc call.
bool_t dp_file_word_count_process_result(struct file_block_task *file_block, caddr_t res, void *arg)
{
	//Processs call result here.
	return FALSE;
}


int file_word_count_1(struct files_array files, u_int64_t *arg, void *process_result_arg)
{
	struct timeval timeout;
	int ret = 0;


	timeout.tv_sec = 100;
	timeout.tv_usec = 0;


	ret = files_dprpc_call(files, FILE_WORD_COUNT, (xdrproc_t)xdr_u_int64_t,(caddr_t)&arg, (xdrproc_t)xdr_int64_t, (dp_resultproc_t)dp_file_word_count_process_result, process_result_arg, timeout);
```

```
        return ret;
}


int
main (int argc, char *argv[])
{
        int ret = 0;
        //You can parse argment here.
        ret = init_dprpc(0, WORDCOUNT, VERSION1, 30);
        if(ret != 0)
                return -1;
        //Now can set the dprpc enverment val.
        TurnOffLog();
        set_max_parell_client_on_node(32);
        //Call the file_word_count_1...
        exit_dprpc();
        return 0;
}
```

Wordcount_server.c:

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */
#include "wordcont.h"


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

```
	bool_t retval;
	/*
	 * insert server code here
	 */

	return retval;
}

int
wordcount_1_freeresult (SVCXPRT *transp, xdrproc_t xdr_result, caddr_t result)
{
	xdr_free (xdr_result, result);
	/*
	 * Insert additional freeing code here, if needed
	 */
	return 1;
}
```

Wordcount_svc.c:

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "wordcont.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
```

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <dprpc.h>


#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif


int
_file_word_count_1 (struct file_block_task *file_block, char * *argp, void *result, struct svc_req *rqstp)
{
	return (file_word_count_1_svc(file_block, *argp, result, rqstp));
}


static void
wordcount_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
	union {
		char *file_word_count_1_arg;
	} argument;
	union {
		u_int64_t file_word_count_1_res;
	} result;
	bool_t retval;
	xdrproc_t _xdr_argument, _xdr_result;
	bool_t (*local)(struct file_block_task *, char *, void *, struct svc_req *);
	struct file_block_task *file_block = NULL;
	switch (rqstp->rq_proc) {
```

```
    case NULLPROC:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
        return;

    case FILE_WORD_COUNT:
        _xdr_argument = (xdrproc_t) xdr_string;
        _xdr_result = (xdrproc_t) xdr_u_int64_t;
        local = (bool_t (*) (struct file_block_task *, char *, void *, struct svc_req *))_file_word_count_1;
        break;

    default:
        svcerr_noproc (transp);
        return;
    }
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
        svcerr_decode (transp);
        return;
    }
    file_block = (struct file_block_task*)malloc(sizeof(struct file_block_task));
    file_block->m_file_name = (char*)malloc(1024);
    if(!svc_getargs(transp, (xdrproc_t)xdr_file_block_fun, (caddr_t) file_block)){
        svcerr_decode (transp);
        return;
    }
    retval = (bool_t) (*local)(file_block, (char *)&argument, (void *)&result, rqstp);
    if (retval > 0 && !svc_sendreply(transp, (xdrproc_t) _xdr_result, (char *)&result)) {
        svcerr_systemerr (transp);
    }
```

```
        if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
                fprintf (stderr, "%s", "unable to free arguments");
                exit (1);
        }
        if (!wordcount_1_freeresult (transp, _xdr_result, (caddr_t) &result))
                fprintf (stderr, "%s", "unable to free results");

        return;
}

int
main (int argc, char **argv)
{
        register SVCXPRT *transp;

        pmap_unset (WORDCOUNT, VERSION1);

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
                fprintf (stderr, "%s", "cannot create udp service.");
                exit(1);
        }
        if (!svc_register(transp, WORDCOUNT, VERSION1, wordcount_1, IPPROTO_UDP)) {
                fprintf (stderr, "%s", "unable to register (WORDCOUNT, VERSION1, udp).");
                exit(1);
        }

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf (stderr, "%s", "cannot create tcp service.");
```

```
            exit(1);
        }
        if (!svc_register(transp, WORDCOUNT, VERSION1, wordcount_1, IPPROTO_TCP)) {
            fprintf (stderr, "%s", "unable to register (WORDCOUNT, VERSION1, tcp).");
            exit(1);
        }

        signal(SIGPIPE, SIG_IGN);
        svc_run ();
        fprintf (stderr, "%s", "svc_run returned");
        exit (1);
        /* NOTREACHED */
}
```

Makefile.wordcount:

```
# This is a template Makefile generated by rpcgen

# Parameters

CLIENT = wordcont_client
SERVER = wordcont_server

SOURCES_CLNT.c =
SOURCES_CLNT.h =
SOURCES_SVC.c =
SOURCES_SVC.h =
SOURCES.x = wordcont.x

TARGETS_SVC.c = wordcont_svc.c wordcont_server.c
TARGETS_CLNT.c =      wordcont_client.c
```

```
TARGETS = wordcont.h      wordcont_svc.c wordcont_client.c wordcont_server.c


OBJECTS_CLNT = $(SOURCES_CLNT.c:%.c=%.o) $(TARGETS_CLNT.c:%.c=%.o)
OBJECTS_SVC = $(SOURCES_SVC.c:%.c=%.o) $(TARGETS_SVC.c:%.c=%.o)
# Compiler flags


CPPFLAGS += -D_REENTRANT
CFLAGS += -g -I/usr/include/tirpc
LDLIBS += -lnsl -lpthread    -ltirpc -ldprpc
rpcgenFLAGS =


# Targets


all : $(CLIENT) $(SERVER)


#$(TARGETS) : $(SOURCES.x)
#      rpcgen $(rpcgenFLAGS) $(SOURCES.x)


$(OBJECTS_CLNT) : $(SOURCES_CLNT.c) $(SOURCES_CLNT.h) $(TARGETS_CLNT.c)


$(OBJECTS_SVC) : $(SOURCES_SVC.c) $(SOURCES_SVC.h) $(TARGETS_SVC.c)


$(CLIENT) : $(OBJECTS_CLNT)
      $(LINK.c) -o $(CLIENT) $(OBJECTS_CLNT) $(LDLIBS)


$(SERVER) : $(OBJECTS_SVC)
      $(LINK.c) -o $(SERVER) $(OBJECTS_SVC) $(LDLIBS)


 clean:
```

```
$(RM) core    $(OBJECTS_CLNT) $(OBJECTS_SVC) $(CLIENT) $(SERVER)
```

这几个文件的作用在上一节已做介绍，这里不再详述。下面介绍如何添加或修改文件代码，并使之编译成我们想要的客户端及服务端。

6.1.1 客户端代码编写

客户端需要修改的代码主要在‘_client.c’文件里，这里需要添加回调函数的代码，若是异步调用，还要修改 DOSRPC 调用接口。使用人员请根据具体需求来选择同步调用还是异步调用。

6.1.1.1 客户端同步调用

由于 dosrpcgen 生成的代码的客户端就是同步调用，这里客户端调用的接口我们可以不做处理。现在只需添加回调函数与参数即可。所修改的文件与代码如下所示（红色字体部分）。

countword_client.c

```
#include "wordcont.h"
#include "dprpc.h"


static int64_t g_word_count = 0;
static    int g_file_num = 0;


bool_t process_result(struct file_block_task *file_block, caddr_t res, void *arg)
{
      int64_t file_word_count = *(int64_t*)res;
      if(file_word_count < 0)
      {
            printf("count word in a file error\n");
            return 0;
      }
      g_word_count += file_word_count;
      g_file_num++;
      return 0;
      }
```

```
int file_word_count_1(struct files_array files, u_int64_t *arg, void *process_result_arg)
{
	struct timeval timeout;
	int ret = 0;

	timeout.tv_sec = 100;
	timeout.tv_usec = 0;

	ret = files_dprpc_call(files, FILE_WORD_COUNT, (xdrproc_t)xdr_u_int64_t,(caddr_t)&arg, (xdrproc_t)xdr_int64_t, (dp_resultproc_t)dp_file_word_count_process_result, process_result_arg, timeout);

	return ret;
}

int main(int argc, char *argv[])
{
	int ret = 0;
	struct files_array files;
	char *arg = "test";
	int i=0;
	time_t begin_time = 0;

	if(argc < 2)
	{
		printf("No check word, use test\n");
	}else{
		arg = strdup(argv[1]);
	}
	ret = init_dprpc(0, WORDCOUNT, VERSION1,30);
```

```
    set_max_parell_client_on_node(16);


    files.m_name_num = 2;
    files.m_file_names = (char**)malloc(sizeof(char*));
    files.m_file_names[0] = strdup("/dpfs/*");
    files.m_file_names[1] = strdup("/dpfs/*/*");


    g_word_count = 0;
    g_file_num =0;
    begin_time = time(NULL);
    printf("begin call files_dprpc_call at %lu\n", begin_time);
    file_word_count_1(files, arg, process_result)
    printf("Now have find %d file , have find word times %d\n", g_file_num, g_word_count);
    printf("last call use time = %d\n", time(NULL) - begin_time);
    exit_dprpc();


    return 0;
}
```

这里在回调函数 bool_t process_result()里添加了返回值的处理代码，主函数 int main()里添加了参数 char *arg（查找的单词）以及 files_array files（文件集合），然后调用接口 file_word_count_1（），这样客户端同步调用的代码就完成了。

6.1.1.2 客户端异步调用

客户端异步调用与同步调用代码的区别主要是接口的不同，其调用函数接口为：

extern DPRPC_HANDLE files_dprpc_call_async(struct files_array files, rpcproc_t proc, xdrproc_t arg_xdr_proc, caddr_t arg, xdrproc_t res_xdr_proc, resultproc_t eachresult,void* result_arg, struct timeval timeout);

除此之外还提供了一些辅助接口来了解任务的执行情况，具体代码如下（红色字体为添加的代码）。

countword_client.c

```
#include "wordcont.h"
```

```
#include "dprpc.h"


static int64_t g_word_count = 0;
static    int g_file_num = 0;


bool_t process_result(struct file_block_task *file_block, caddr_t res, void *arg)
{
        int64_t file_word_count = *(int64_t*)res;
        if(file_word_count < 0)
        {
                printf("count word in a file error\n");
                return 0;
        }
        //printf("%lu,    file    %s    word    %ld\n",    time(NULL),    file_block->m_file_name,
file_word_count);
        g_word_count += file_word_count;
        g_file_num++;
        return 0;
}


int main(int argc, char *argv[])
        struct timeval timeout;
        struct files_array files;
        char *arg = "test";
        int total_num, success_num, fail_num;
        DPRPC_HANDLE handle=NULL;
        int i=0;
        time_t begin_time = 0;


        if(argc < 2)
```

```
{
    printf("No check word, use test\n");
}else{
    arg = strdup(argv[1]);
}
ret = init_dprpc(0, WORDCOUNT, VERSION1,30);

timeout.tv_sec = 50;
timeout.tv_usec = 0;

set_max_parell_client_on_node(128);
set_need_collect_node_info(1);
set_use_current_load(0);

files.m_name_num = 1;
files.m_file_names = (char**)malloc(sizeof(char*));
files.m_file_names[0] = strdup("/dpfs/*/*");

g_file_num = 0;
g_word_count = 0;
begin_time = time(NULL);
handle = files_dprpc_call_async(files, FILE_WORD_COUNT, (xdrproc_t)xdr_string, (caddr_t)&arg, (xdrproc_t)xdr_int64_t, (dp_resultproc_t)process_result, NULL, timeout);
if(handle == NULL)
{
    printf("start rpc call error\n");
    exit_dprpc();
    return -1;
}
```

```
    while(check_dprpc_call_end(handle) == 0)
    {
        sleep(1);
        get_dprpc_task_info(handle, &total_num, &success_num, &fail_num);
        printf("have total task %d, Now success task %d, fail task %d\n", total_num, success_num, fail_num);
    }
    ret = get_dprpc_call_end_reason(handle);
    printf("the dprpc call end reasin is %d\n", ret);
    free_dprpc_handle(handle);
    printf("use time=%d, Now have find %d file , have find word times %d\n", time(NULL)-begin_time, g_file_num, g_word_count);
  exit_dprpc();

    return 0;
}
```

与同步调用相似，回调函数 bool_t process_result()里添加了返回值的处理代码，主函数 int main()里添加了参数 char *arg（查找的单词）以及 files_array files（文件集合），然后调用异步调用接口 files_dprpc_call_async（），再调用查找信息的接口 get_dprpc_task_info（）显示本次调用的信息，这样客户端异步调用的代码就完成了。

6.1.2 服务端代码编写

服务端需要修改的代码主要在‘_server.c’文件里，这里需要添加处理函数的代码，若是多线程处理，还要修改 DOSRPC 调用接口。使用人员请根据具体需求来选择单线程处理还是多线程处理。

6.1.2.1 服务器单线程处理

由于 dosrpcgen 生成的代码的服务端就是单线程处理，这里服务端调用的接口我们可以不做处理。现在只需添加处理函数代码即可。所修改的文件与代码如下所示（红色字体部分为添加的代码）。

wordcont_server.c

```
#include "wordcont.h"
```

```
void change(char *a)
{
	while(*a!='\0')
	{
		if(*a>='A'&&*a<='Z')
		*a+=32;
		a++;
	}
}

bool_t
file_word_count_1_svc(struct file_block_task *file_block, char *word, u_int64_t *result,    struct svc_req *rqstp)
{
	bool_t retval = 1;
	char *file_name = NULL;
	int n,sum = 0;
	char read[2048], *p = read,ch;
	FILE *fp;
	int    chs = 0;


	file_name = (char *)malloc(strlen(file_block->m_file_name));
	memset(file_name, 0, strlen(file_block->m_file_name));
	file_name = file_block->m_file_name;

	if((fp = fopen(file_name, "rb")) == NULL)
	{
		printf("open file err !\n");
```

```
            retval = 0;
            return retval;
}

change(word);
ch = fgetc(fp);

while(!feof(fp))
{
            chs = 1;
            while((ch>='A'&&ch<='Z')||(ch>='a'&&ch<='z'))
            {
                        *p = ch;
                        p++;
                        ch = fgetc(fp);
                        if(chs++ > 2046)
                                    break;
            }

            *p = '\0';
            change(read);

            if((strlen(word) == strlen(read))&&(strncmp(word, read, strlen(word)) == 0))
            {
                        sum ++;
            }

            if(!feof(fp))
            {
                        ch = fgetc(fp);
```

```
                }

                p = read;
        }

        fclose(fp);
        *result = sum;
        free(file_name);
        file_name = NULL;
        return retval;
}

int
wordcount_1_freeresult (SVCXPRT *transp, xdrproc_t xdr_result, caddr_t result)
{
        xdr_free (xdr_result, result);

        /*
         * Insert additional freeing code here, if needed
         */

        return 1;
}
```

服务端对接收信息的处理是由用户自己实现，这里是一个范例。本例中我们在处理函数file_word_count_1_svc( )里添加处理代码，其内容为查找一个单词在某一指定文件里出现的次数，返回值为查找到的单词的个数。这样服务器单线程调用的代码编写就完成了。

6.1.2.2 服务器多线程处理

服务器多线程编程修改的主要文件是“_server.c”与“_svc.c”这两个文件，其中“_server.c”文件的修改与单线程处理的代码修改一样，具体请参考上一小节，这里不再详细介绍，这里主要介绍文件“_svc.c”的代码添加与修改。所修改的文件与代码如下所示（红色字体为添

加部分）。

wordcont_svc_thread_pool.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "wordcont.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>
#include <signal.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

int
_file_word_count_1 (struct file_block_task *file_block,char * *argp, void *result, struct svc_req *rqstp)
{
        return (file_word_count_1_svc(file_block, *argp, result, rqstp));
}

void real_process_call_fun(struct svc_req *rqstp, register SVCXPRT *transp)
```

```
{
        union {
                char *file_word_count_1_arg;
        } argument;
        union {
                u_int64_t file_word_count_1_res;
        } result;
bool_t retval;
        xdrproc_t _xdr_argument, _xdr_result;
        bool_t (*local)(struct file_block_task *, char *, void *, struct svc_req *);
        struct file_block_task *file_block = NULL;
        uint32_t have_files = 2;


        switch (rqstp->rq_proc) {
        case NULLPROC:
                (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
                return ;


        case FILE_WORD_COUNT:
                _xdr_argument = (xdrproc_t) xdr_wrapstring;
                _xdr_result = (xdrproc_t) xdr_u_int64_t;
                local = (bool_t (*) (struct file_block_task *,char *, void *,    struct svc_req
*))_file_word_count_1;
                break;


        default:
                svcerr_noproc (transp);
                return ;
        }
        memset ((char *)&argument, 0, sizeof (argument));
```

```
        if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
                        printf("decode first argment error\n");
                svcerr_decode (transp);
                return ;
        }
        file_block = (struct file_block_task*)malloc(sizeof(struct file_block_task));
        file_block->m_file_name = (char*)malloc(1024);
        if(!svc_getargs(transp, (xdrproc_t)xdr_file_block_fun, (caddr_t) file_block)){
                        printf("decode file block argment error\n");
                svcerr_decode (transp);
                return ;
}


        retval = (bool_t) (*local)(file_block, (char *)&argument, (void *)&result, rqstp);
        if (retval > 0 && !svc_vc_sendreply(transp, (xdrproc_t) _xdr_result, (char *)&result,
(xdrproc_t)NULL, NULL, &have_fi
les)) {
                        printf("call local function error\n");
                svcerr_systemerr (transp);
        }
        if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
                fprintf (stderr, "%s", "unable to free arguments");
        }
        if (!wordcount_1_freeresult (transp, _xdr_result, (caddr_t) &result))
                fprintf (stderr, "%s", "unable to free results");

        //printf("end use SVCXPRT %p\n", transp);
        return ;
}
```

```
struct call_proc_arg{
        struct svc_req *rqstp;
        SVCXPRT *transp;
        struct call_proc_arg *next;
};
struct call_proc_arg *g_task_list = NULL;
pthread_mutex_t g_task_list_mutex;;
pthread_cond_t    g_task_cond;
int g_end_thread_pool = 0;

void *thread_pool_process(void *arg)
{
        struct call_proc_arg *call_arg = NULL;
        struct svc_req *rqstp = NULL;
        register SVCXPRT *transp = NULL;

        while(!g_end_thread_pool)
        {
                pthread_mutex_lock(&g_task_list_mutex);
                if(g_task_list == NULL)
                {
                        pthread_cond_wait(&g_task_cond, &g_task_list_mutex);
                        pthread_mutex_unlock(&g_task_list_mutex);
                        continue;
                }

                call_arg = g_task_list;
                g_task_list = g_task_list->next;
                pthread_mutex_unlock(&g_task_list_mutex);
```

```
                if(call_arg == NULL)
                        continue;
                rqstp = call_arg->rqstp;
                transp = call_arg->transp;
                real_process_call_fun(rqstp, transp);
                task_end(transp);
        }
        return NULL;
}
static void
wordcount_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        struct call_proc_arg *call_arg = NULL;
        pthread_t pthread_id;
        int ret = 0;

        if(!can_accept_task(transp))
        {
                svcerr_systembusy(transp);
                return ;
        }
        call_arg = (struct call_proc_arg *)malloc(sizeof(struct call_proc_arg));
        call_arg->rqstp = rqstp;
        call_arg->transp = transp;
        pthread_mutex_lock(&g_task_list_mutex);
        call_arg->next = g_task_list;
        g_task_list         = call_arg;
        pthread_mutex_unlock(&g_task_list_mutex);
        pthread_cond_signal(&g_task_cond);
        return ;
```

```
}
int
main (int argc, char **argv)
{
        register SVCXPRT *transp;

        pmap_unset (WORDCOUNT, VERSION1);

        set_max_server_a_cpu(4);

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
                fprintf (stderr, "%s", "cannot create udp service.");
                exit(1);
        }
        if (!svc_register(transp, WORDCOUNT, VERSION1, wordcount_1, IPPROTO_UDP))
{
                perror("regisiter error");
                fprintf (stderr, "%s", "unable to register (WORDCOUNT, VERSION1,
udp).");
                exit(1);
        }

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf (stderr, "%s", "cannot create tcp service.");
                exit(1);
        }
        if (!svc_register(transp, WORDCOUNT, VERSION1, wordcount_1, IPPROTO_TCP)) {
                fprintf (stderr, "%s", "unable to register (WORDCOUNT, VERSION1, tcp).");
```

```
                exit(1);
        }
        signal(SIGPIPE, SIG_IGN);
        pthread_mutex_init(&g_task_list_mutex, NULL);
        pthread_cond_init(&g_task_cond, NULL);
        g_end_thread_pool = 0;
        //create thread pool
        pthread_t *process_thread_ids = NULL;
        int i=0;
        int threads = get_cpu_num()*4;
        process_thread_ids = (pthread_t*)malloc(sizeof(pthread_t) * threads);
        if(process_thread_ids == NULL)
        {
                fprintf (stderr, "%s", "malloc thread pool ids error\n");
                exit(1);
        }
        for(i=0; i<threads; i++)
        {
                pthread_create(&(process_thread_ids[i]), NULL, thread_pool_process,
NULL);
        }
        svc_run ();
        fprintf (stderr, "%s", "svc_run returned");
        g_end_thread_pool = 1;
        pthread_cond_broadcast(&g_task_cond);
        for(i=0; i<threads; i++)
                pthread_join(process_thread_ids[i], NULL);
        exit (1);
        /* NOTREACHED */
}
```

本文件为一个多线程编程例子，用户可自行更改。主要修改的地方是在主函数 int main()里面创建一个线程池 pthread_create()，当每个线程接收到数据后调用接口 real_process_call_fun()进行参数解析并执行处理函数。其中接口 real_process_call_fun()接口内容与利用 dosrpcgen 生成的接口 process_call_fun()内容一样，只需修改接口名字即可。

值得注意的是 have_files 这个参数是针对数字有机体远程过程调用联合模式设定的，本例为数字有机体远程过程调用单一模式，它没有返回的文件，所以设定为‘2’否则为‘1’。

6.2 数字有机体远程过程调用联合模式（UNION-DOSRPC）

DOSRPC 的安装软件里含有一个例子，文件夹名为 ordring，下面对这个例子进行分析，介绍数字有机体远程过程调用联合模式的程序编写。

这个 DOSRPC 例子的目的是为了在数字有机体文件系统里，根据所指定的路径下的文件名链表，对文件里的数字进行排序（其指定的文件分布在数字有机体工作平台中且只需给出路径即可，语法上支持‘*’号）。

6.2.1 客户端代码编写

客户端代码调用的接口因其分组模式的不同而不同，包括系统分组与用户自定义分组两种分组方式。

其中系统分组的选取有三种方式：

1) 默认分组方式：系统根据用户设定的分组大小，按先后顺序进行分组。
2) 周期分组方式：系统根据用户设定的时间与分组大小，周期进行分组。
3) 混合分组方式：中和上面两种分组方式。

自定义分组方式需要用户调用分组接口编写自己的分组函数。

6.2.1.1 系统分组方式

文件 ordering_client.c 里面设置了四组参数,分别对应如下：

1) 系统默认分组，两层 DOSRPC 调用。
2) 系统默认分组，三层 DOSRPC 调用。
3) 周期分组，两层 DOSRPC 调用。
4) 混合分组，三层 DOSRPC 调用。

用户可以参考这几组参数编写自己需要的参数设置。

客户端调用的接口为 files_dprpc_union_call(files, input)，参数 input 即为上面四组参数的设置，files 为文件链表，即要处理的文件名，支持带‘*’号的匹配。（参数的地址会被系统默认会释放，若想查看其信息，请先行拷贝）以下是其内容（红色内容为用户需要修改的地方）：

ordering_client.c

```
/*
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "ordering.h"
#include <stdio.h>
static char *g_result_file_name = NULL;
static pthread_mutex_t    g_result_lock = PTHREAD_MUTEX_INITIALIZER;

static void free_file_list(struct file_list *filelist)
{

        while(filelist)
        {
                free(filelist->m_file_name);
                free(filelist);
                filelist = filelist->m_next;
        }
        return;
}
static struct file_list* cp_file_list(struct file_list *files)
{
        struct file_list *_files = NULL;
        struct file_list *filelist = NULL, *filelist_next = NULL, *filelist_number = NULL;

        _files = files;
        while(_files)
        {
                filelist_number = (struct file_list *)malloc(sizeof(struct file_list));
```

```
                filelist_number->m_file_name = strdup(_files->m_file_name);
                filelist_number->m_next = NULL;
                printf("FILES: %s\n", _files->m_file_name);
                if(filelist    == NULL)
                {
filelist = filelist_number;
                        filelist_next = filelist_number;
                }else{
                        filelist_next->m_next = filelist_number;
                        filelist_next = filelist_number;
                }
                filelist_number = NULL;
                _files = _files->m_next;
        }
        return filelist;
}

bool_t process_result(struct file_block_task *file_block, caddr_t res, void *arg)
{
        char *file_name = NULL;
        char *re_file_name = NULL;
        struct file_list *file_list = NULL;
        struct file_list *file_list_next = NULL;
        struct file_list *list = NULL;

        pthread_mutex_lock(&g_result_lock);
        list = cp_file_list((struct file_list *)res);
        printf("\nPROCESS_RESULT: %s\n\n", list->m_file_name);
        if(g_result_file_name == NULL)
        {
```

```
                re_file_name = order_files(list);


        }else{
                file_list = (struct file_list *)malloc(sizeof(struct file_list));
                file_list->m_file_name = strdup(g_result_file_name);
                file_list->m_next = list;
                re_file_name = order_files(file_list);
                free_file_list(file_list);
                free(g_result_file_name);
        }
g_result_file_name = re_file_name;
        pthread_mutex_unlock(&g_result_lock);
        return 0;
}

void init_input(struct rpc_union_input *input)
{
        memset(input, 0, sizeof(struct rpc_union_input));


        input->m_proc = 0;
        input->m_arg_xdr_proc = NULL;
        input->m_arg = NULL;
        input->m_res_xdr_proc = NULL;
        input->m_grouping = NULL;
        input->m_grouping_arg = NULL;
        input->m_grouping_mode = 0;
        input->m_grouping_mode_arg[0] = 0;
        input->m_grouping_mode_arg[1] = 1;
        input->m_timeout.tv_sec = 1000;
        input->m_timeout.tv_usec = 0;
```

```
	input->m_result_fun = NULL;
	input->m_result_arg = NULL;
	input->m_next = NULL;

	return;
}

void free_input(struct rpc_union_input *input)
{
	if(input->m_next != NULL)
		free_input(input->m_next);
	free(input);
	input = NULL;
	return;
}
//default grouping, 2 level
struct rpc_union_input *get_input_default_grouping_2(struct files_array files, struct timeval timeout)
{
	struct rpc_union_input *input = NULL;
	struct rpc_union_input *input_next = NULL;

	input = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
	init_input(input);
	input->m_proc = 1;
	input->m_grouping_mode_arg[0] = 3;

	input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
	init_input(input_next);
	input_next->m_proc = 3;
```

```
        input_next->m_grouping_mode_arg[0] = 3;
        input_next->m_grouping_mode = 0;
        input_next->m_res_xdr_proc = (xdrproc_t)xdr_file_list;
        input_next->m_result_fun = (dp_resultproc_t)process_result;
        input_next->m_next = NULL;
        input->m_next = input_next;

        return input;
}
//default grouping, 3 level
struct rpc_union_input *get_input_default_grouping_3(struct files_array files, struct timeval timeout)
{
        struct rpc_union_input *input = NULL;
        struct rpc_union_input *input_next = NULL;

        input = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input);
        input->m_proc = 1;
        input->m_grouping_mode_arg[0] = 3;

        input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input_next);
        input_next->m_proc = 2;
        input_next->m_grouping_mode_arg[0] = 3;
        input->m_next = input_next;
        input_next = NULL;

        input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input_next);
```

```
        input_next->m_proc = 3;
        input_next->m_grouping_mode_arg[0] = 3;
        input_next->m_res_xdr_proc = (xdrproc_t)xdr_file_list;
        input_next->m_result_fun = (dp_resultproc_t)process_result;
        input_next->m_next = NULL;
        input->m_next->m_next = input_next;

        return input;
}
//cycle grouping, 2 level
struct rpc_union_input *get_input_cycle_grouping_2(struct files_array files, struct timeval timeout)
{
        struct rpc_union_input *input = NULL;
        struct rpc_union_input *input_next = NULL;

        input = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input);
        input->m_proc = 1;
        input->m_grouping_mode_arg[0] = 3;
        input->m_grouping_mode_arg[1] = 1000;
        input->m_grouping_mode =1;

        input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input_next);
        input_next->m_proc = 3;
        input_next->m_grouping_mode_arg[0] = 3;
        input_next->m_grouping_mode = 0;
        input_next->m_res_xdr_proc = (xdrproc_t)xdr_file_list;
        input_next->m_result_fun = (dp_resultproc_t)process_result;
```

```
	input_next->m_next = NULL;
	input->m_next = input_next;

	return input;
}
//admixture, 3 level
struct rpc_union_input *get_input_admixture_grouping_3(struct files_array files, struct timeval timeout)
{
	struct rpc_union_input *input = NULL;
	struct rpc_union_input *input_next = NULL;

	input = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
	init_input(input);
	input->m_proc = 1;
	input->m_grouping_mode_arg[0] = 3;
	input->m_grouping_mode_arg[1] = 1000;
	input->m_grouping_mode =1;

	input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
	init_input(input_next);
	input_next->m_proc = 2;
	input_next->m_grouping_mode_arg[0] = 3;
	input->m_next = input_next;
	input_next = NULL;

	input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
	init_input(input_next);
	input_next->m_proc = 3;
	input_next->m_grouping_mode_arg[0] = 3;
```

```
        input_next->m_res_xdr_proc = (xdrproc_t)xdr_file_list;
        input_next->m_result_fun = (dp_resultproc_t)process_result;
        input_next->m_next = NULL;
        input->m_next->m_next = input_next;


        return input;
}
int main(int argc, char *argv[])
{
        int ret = 0;
        struct timeval timeout;
        struct files_array files;
        time_t begin_time = 0;


        struct rpc_union_input *input = NULL;


        ret = init_dprpc(0, ORDERING_PROG, ORDERING_VERSION,30);


//      set_max_parell_recv_threads(1);
//      set_end_execute_when_fail(0);
        set_send_request_thread_num(1);
        timeout.tv_sec = 100;
        timeout.tv_usec = 0;


        set_max_parell_client_on_node(16);


        files.m_file_names = (char**)malloc(sizeof(char*));
        files.m_file_names[0] = strdup("/dpfs/test_*");
        files.m_name_num = 1;
```

```
        input = get_input_default_grouping_2(files, timeout);
        begin_time = time(NULL);
        printf("begin call files_dprpc_call at %lu\n", begin_time);
        ret = files_dprpc_union_call(files, input);
        if(ret != 0)
                printf("files_dprpc_union_call failed! \n");
        else
                printf("files_dprpc_union_call success! result in file '%s'\n",
g_result_file_name);
         printf("last call use time = %d\n", time(NULL) - begin_time);
        free_input(input);
        free(files.m_file_names);
        free(g_result_file_name);
        return 0;
}
```

其中 bool_t process_result(struct file_block_task *file_block, caddr_t res, void *arg)为结果处理函数。此文件用户主要需要定义分组参数（本文件给出了四种分组模式的参数设置）、调用联合 DOSRPC 接口、编写结果处理函数，结构与单一模式的客户端代码大致相同，由于 dosrpcgen 不会生成 DOSRPC 联合模式的代码，请用户参考本例进行编写。

6.2.1.2 用户自定义分组方式

为了方便用户自定义分组，联合数字有机体远程过程调用给用户提供了一系列接口，这些接口的详细介绍请参考第四小节。以下为文件内容（红色字体为用户主要修改部分）：

ordering_client_user_defined.c：

```
/*
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "ordering.h"
#include <stdio.h>
```

```
#include <dprpc.h>


#define GROUPING_SIZE 3
static char *g_result_file_name = NULL;
static pthread_mutex_t    g_result_lock = PTHREAD_MUTEX_INITIALIZER;
static pthread_mutex_t    g_grouping_lock = PTHREAD_MUTEX_INITIALIZER;


static void free_file_list(struct file_list *filelist)
{
        struct file_list *filelist_next = NULL;


        while(filelist)
        {
                filelist_next = filelist->m_next;
                free(filelist->m_file_name);
                free(filelist);
                filelist = filelist->m_next;
        }
        return;
}
static struct file_list* cp_file_list(struct file_list *files)
{
        struct file_list *_files = NULL;
        struct file_list *filelist = NULL, *filelist_next = NULL, *filelist_number = NULL;


        _files = files;
        while(_files)
{
                filelist_number = (struct file_list *)malloc(sizeof(struct file_list));
                filelist_number->m_file_name = strdup(_files->m_file_name);
```

```
                filelist_number->m_next = NULL;
                printf("FILES: %s\n", _files->m_file_name);
                if(filelist    == NULL)
                {
                        filelist = filelist_number;
                        filelist_next = filelist_number;
                }else{
                        filelist_next->m_next = filelist_number;
                        filelist_next = filelist_number;
                }
                filelist_number = NULL;
                _files = _files->m_next;
        }
        return filelist;
}

void grouping_first(caddr_t result, caddr_t addrs, void* rpc_handle)
{
        struct file_list *filelist = NULL;
        struct file_list *grouping_filelist = NULL;
        struct file_list *tmp_filelist = NULL;
        struct file_list *save = NULL;
        int num = 0;
        int ret = 0;

        pthread_mutex_lock(&g_grouping_lock);
        filelist = cp_file_list((struct file_list *)addrs);
        save = get_grouping_number_pointer(rpc_handle);
        if(filelist == NULL) return;
        if(save == NULL)
```

```
        {
                save =    (struct file_list *)malloc(sizeof(struct file_list));
memset(save, 0, sizeof(struct file_list));
        }
        filelist->m_next = save;
        save = filelist;

        filelist = save;
        num = 0;
        while(filelist)
        {
                num ++;
                if(num == GROUPING_SIZE)
                {
                        grouping_filelist = save;
                        save = filelist->m_next;
                        tmp_filelist = filelist;
                        filelist = filelist->m_next;
                        tmp_filelist->m_next = NULL;
                        ret = create_group_task(grouping_filelist, rpc_handle);
                        free_file_list(grouping_filelist);
                        if(ret == 0)
                        {
                                if(filelist)
                                        free_file_list(filelist);
                                printf("create_group_task error \n\n");
                                pthread_mutex_unlock(&g_grouping_lock);
                                return;
                        }
                        num = 0;
```

```
                }else
                        filelist = filelist->m_next;
        }
        set_grouping_number_pointer((void *)save, rpc_handle);
        pthread_mutex_unlock(&g_grouping_lock);
        return ;
}
bool_t process_result(struct file_block_task *file_block, caddr_t res, void *arg)
{
        char *file_name = NULL;
        char *re_file_name = NULL;
        struct file_list *file_list = NULL;
        struct file_list *file_list_next = NULL;
        struct file_list *list = NULL;

        pthread_mutex_lock(&g_result_lock);
        list = cp_file_list((struct file_list *)res);
        printf("\nPROCESS_RESULT: %s\n\n", list->m_file_name);
        if(g_result_file_name == NULL)
        {
                re_file_name = order_files(list);

        }else{
                file_list = (struct file_list *)malloc(sizeof(struct file_list));
                file_list->m_file_name = strdup(g_result_file_name);
                file_list->m_next = list;
                re_file_name = order_files(file_list);
                free_file_list(file_list);
        }
        g_result_file_name = re_file_name;
```

```
        pthread_mutex_unlock(&g_result_lock);
        return 0;
}
void init_input(struct rpc_union_input *input)
{
        memset(input, 0, sizeof(struct rpc_union_input));

        input->m_proc = 0;
        input->m_arg_xdr_proc = NULL;
        input->m_arg = NULL;
        input->m_res_xdr_proc = NULL;
        input->m_grouping = NULL;
        input->m_grouping_arg = NULL;
        input->m_grouping_mode = 0;
        input->m_grouping_mode_arg[0] = 0;
        input->m_grouping_mode_arg[1] = 1;
        input->m_timeout.tv_sec = 1000;
        input->m_timeout.tv_usec = 0;
        input->m_next = NULL;

        return;
}
void free_input(struct rpc_union_input *input)
{
        if(input->m_next != NULL)
                free_input(input->m_next);
        free(input);
        input = NULL;
        return;
}
```

```
int main(int argc, char *argv[])
{
        int ret = 0;
        struct timeval timeout;
        struct files_array files;
        time_t begin_time = 0;

        struct rpc_union_input *input = NULL;
        struct rpc_union_input *input_next = NULL;

        ret = init_dprpc(0, ORDERING_PROG, ORDERING_VERSION,30);

//      TurnOffLog();
        set_checkpoint_flag(0);
        set_max_parell_recv_threads(1);
        set_end_execute_when_fail(0);

        timeout.tv_sec = 100;
        timeout.tv_usec = 0;

        set_max_parell_client_on_node(16);

        files.m_file_names = (char**)malloc(sizeof(char*));
        files.m_file_names[0] = strdup("/dpfs/test_*");
        files.m_name_num = 1;
        input = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input);
        input->m_proc = 1;
        input->m_grouping_mode = 2;
```

```
        input-> m_grouping = (groupingproc_t)grouping_first;


        input_next = (struct rpc_union_input *)malloc(sizeof(struct rpc_union_input));
        init_input(input_next);
        input_next->m_proc = 2;
        input_next->m_res_xdr_proc = (xdrproc_t)xdr_file_list;
        input_next->m_result_fun = (dp_resultproc_t)process_result;
        input->m_next = input_next;
        input_next = NULL;


        begin_time = time(NULL);
        printf("begin call files_dprpc_call at %lu\n", begin_time);
        ret = files_dprpc_union_call(files, input);
        if(ret != 0)
                        printf("files_dprpc_union_call failed! \n");
        else
                        printf("files_dprpc_union_call success! result in file '%s'\n",
g_result_file_name);
          printf("last call use time = %d\n", time(NULL) - begin_time);
        free_input(input);
        free(files.m_file_names);
        free(g_result_file_name);
        return 0;
}
```

本文件与系统分组相比只是分组函数不一样，分组函数须由用户自定义填写。这里给出了一个样例，用户可参考本例编写自己的分组函数。

6.2.2 服务端代码编写

联合数字有机体远程过程调用的服务端与单一数字有机体远程过程调用内容大致相同，只有“_svc.c”文件里对参数的解析有所不同，这里不做详细介绍，具体请参考单一数字有

机体远程过程调用的服务端编写。

6.3 代码的编译

DOSRPC 的初始代码是采用 dosrpcgen 自动生成的，附带生成 Makefile 文件，名称为 Makefile.sample，其中 sample 为用户自定义的程序名称。用户可以选择将编译文件 Makefile.sample 更名为 Makefile，然后执行“make”命令进行程序的编译。

Makefile.sample 文件的 CLIENT 变量定义了客户端程序的名称，SERVER 定义了服务端程序的名称，开发人员想要修改，可以直接编辑这两个名称。编译文件 Makefile 已自动包含了 DOSRPC 的动态链接库“libdprpc.so”。

7. dosrpc_manager 介绍

7.1 使用

在安装目录下输入：./dosrpc_manager 进入如下页面。（注意只能在客户端使用，否则找不到日志文件）

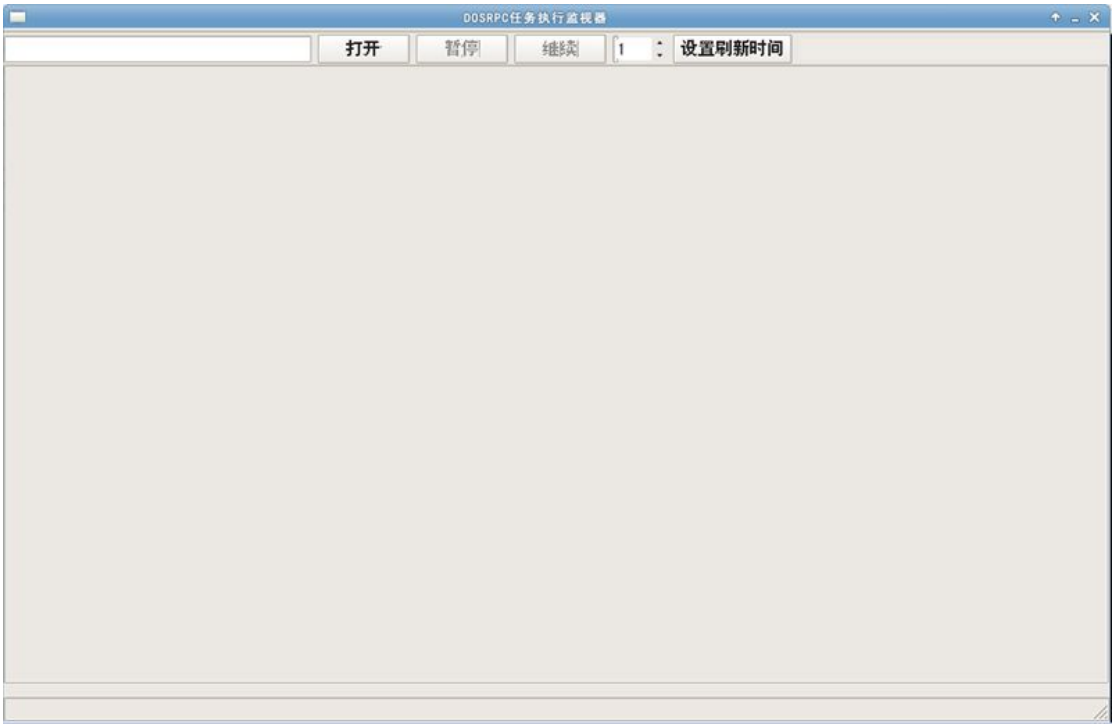

图 7-1 dos_manager 的显示界面

7.2 页面介绍

界面包括一个菜单栏与一个显示框。其中菜单栏包括：

输入框：用来输入文件的位置。

“打开”按钮：点击后显示栏将加载输入框的文件。

“暂停”按钮：暂停读取文件。

“继续”按钮：继续读取文件。

“设置刷新时间”按钮：设置更新的时间，单位为秒。

显示框：显示读取文件的状态。

点击“打开”按钮将弹出选择文件对话框，选择要加载的文件，检查点的日志文件在“/var/checkpoint/”目录下，所以弹出框默认此目录。如下图所示：

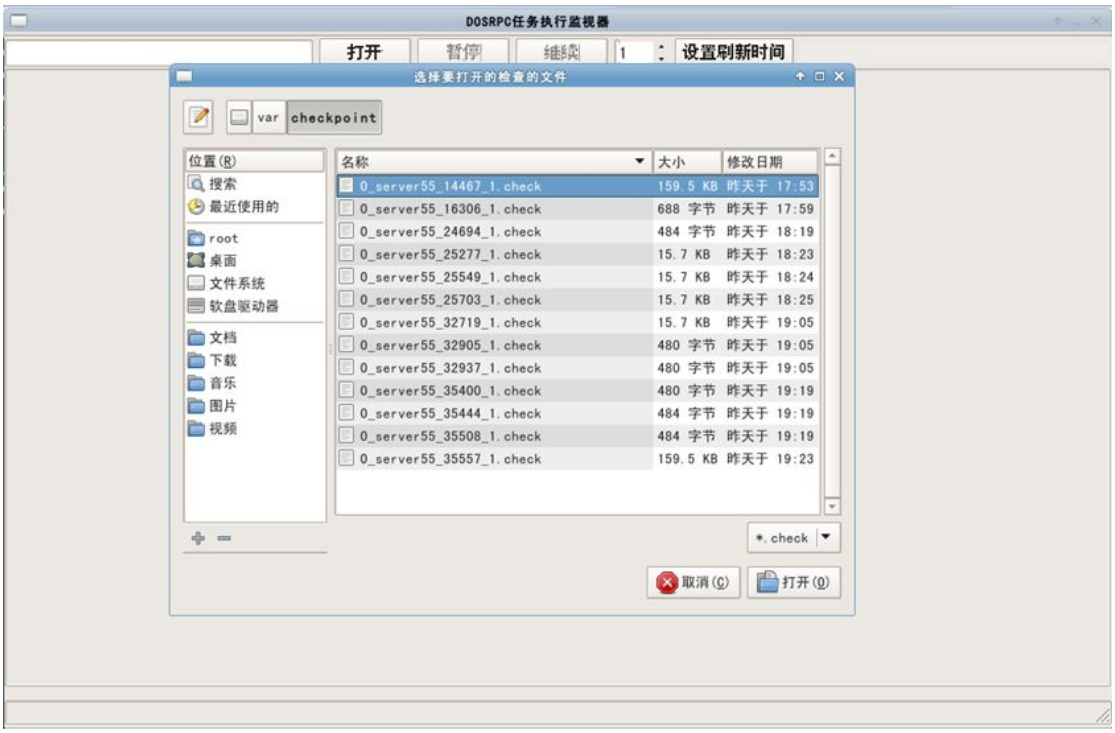

图 7-2 dos_manager 选择日志文件

当文件加载后，显示如下图所示：

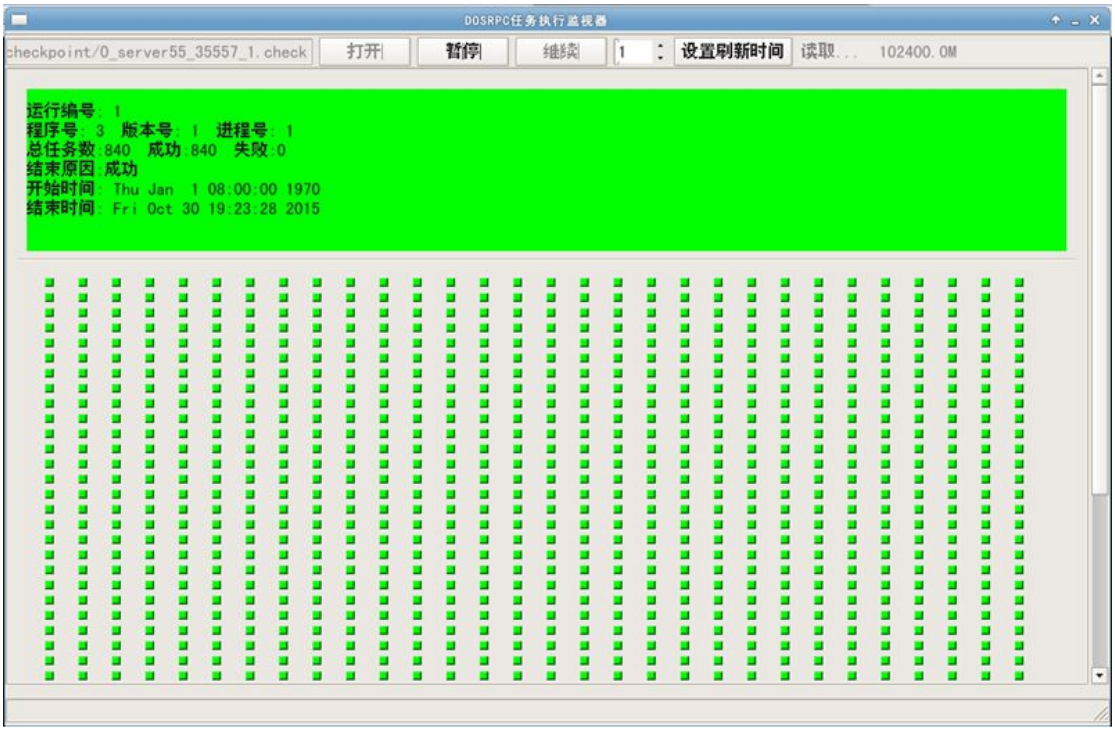

图 7-3 dos_manager 文件加载信息

每一个显示框代表一个 DOSRPC 调用，当其显示区域为绿色代表此 DOSRPC 调用执行完成，没有颜色代表此 DOSRPC 调用正在执行，当显示为红色代表此 DOSRPC 调用执行失败。

其显示的内容包括：DOSRPC 的 ID、程序号、版本号、进程号、全部任务数、成功的

任务数、失败的任务数、结束原因、开始时间、结束时间。

显示框下面的一个圆点代表一个任务，绿色代表任务执行完成，没有颜色代表正在执行，红色代表任务执行失败。

单击一个任务即圆点，会显示这个任务的详细信息，如图所示：

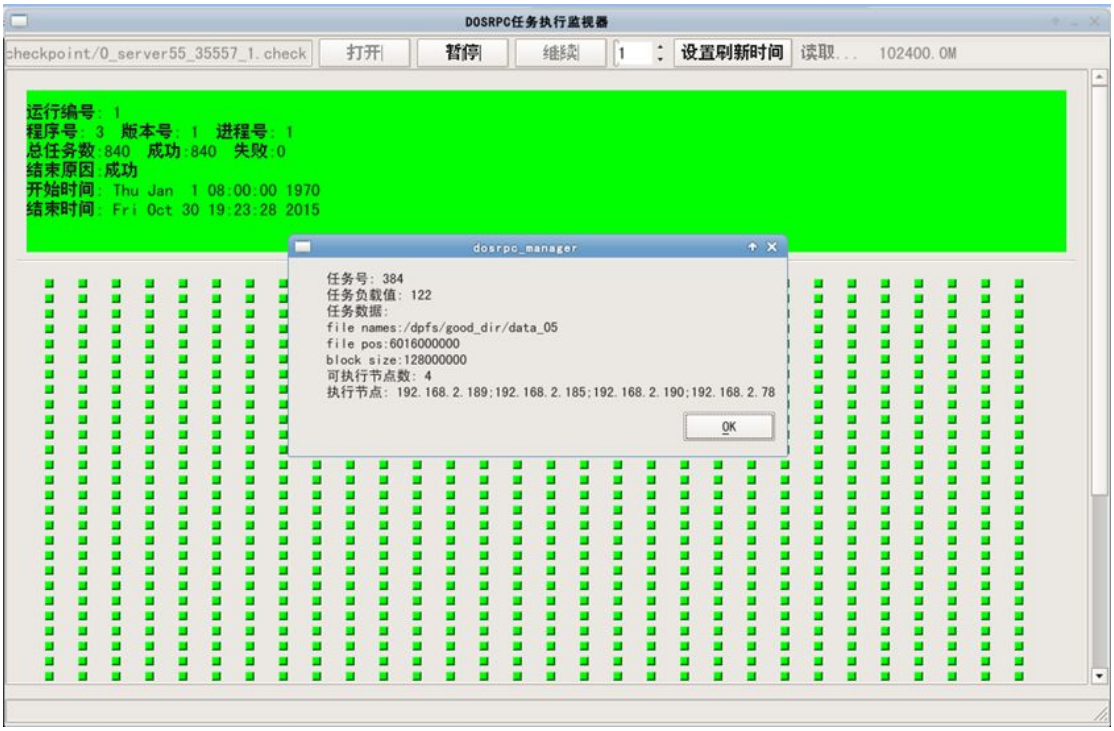

图 7-4 节点信息

显示的任务信息包括：此任务的 ID 号、负载量、包含的文件名、文件的起始位置、文件的大小、包含此文件的节点信息、节点 IP。

8. 编程示例与测试

DOSRPC 具备超强的能力，能有效地将多台服务器的能力整合起来，一起完成用户指定的任务。正确安装 DOSRPC 安装包后，两个完整的编程示例及源代码（ordering 和 wordcount）被放在/usr/share/dosrpc/目录中。

其中示例程序 ordering 用于对文件中的单词和整形数字进行排序，最终把结果记录到文件中。示例程序 wordcount 用于查找文件中的指定字符串，并统计包含该字符串的文件数量和出现的次数。

为了方便测试，并显示 DOSRPC 示例的正确性，测试所用的数据文件建议采用 gen_rpc_data 程序来自动生成。本公司采用该程序生成过 10 个 10GB 的文件，并对示例程序（ordering 和 wordcount）进行了测试，最终完全正确。

为了展示采用 DOSRPC 应用程序的性能远超单机系统的应用程序，在 wordcount 示例程序所在的目录中，通过执行脚本“build_test_wordcount.sh”可以编译得到 test_wordcount 测试程序；然后使用 test_wordcount 来测试单机查找并统计字符串的最佳性能。通过单机最佳性能和 wordcount 的性能比较，可以很好地证明 DOSRPC 的强大。

一般而言，数字有机体服务器越多，数据资源越分散，DOSRPC 展现的能力就越强大。

每个示例程序都由服务端程序和客户端程序组成。编程示例程序（ordering 和 wordcount）必须运行在数字有机体工作平台上。测试时，务必在所有的数字有机体工作平台上先执行“do_su”登录，然后再运行服务端程序（如 wordcont_svc_single_thread），最后再在客户端中“do_su”登录后，执行客户端程序（如 wordcont_client）。以下是本公司测试人员得出的一组数据，仅作参考：

测试项目	是否单机	数据总量	服务程序	客户程序	花费时间（单位秒）
个人计算机	单机	10G×10	无	test_wordcount	3229
普通服务器	单机	10G×10	无	test_wordcount	1953
4 台个人计算机和 2 台普通服务器，客户端采用消息的同步模式	6 台服务器	10G×10	wordcont_svc_single_thread	wordcont_client	652
	6 台服务器	10G×10	wordcont_svc_mult_thread	wordcont_client	313
	6 台服务器	10G×10	wordcont_svc_thread_pool	wordcont_client	341
4 台个人计算机和 2 台普通服务器，客户端采用消息的异步模式	6 台服务器	10G×10	wordcont_svc_single_thread	wordcont_client_async	640
	6 台服务器	10G×10	wordcont_svc_mult_thread	wordcont_client_async	313
	6 台服务器	10G×10	wordcont_svc_thread_pool	wordcont_client_async	503

9. 出错处理

如果在使用本产品时，您发现了某些缺陷与不足，或是需要帮助，请即时致函我们。我们的电子邮箱是：tianxinyue@126.com。

要想获取最新的在线用户手册和了解最新的相关信息，您可访问我们的公司 Web 站点 www.tianxinyue.com。

最终用户许可协议

请务必仔细阅读和理解本数字有机体系统软件最终用户许可协议（以下简称《协议》）中规定的所有权利和限制。在安装时，用户需要仔细阅读并决定接受或不接受本《协议》的条款。除非或直至用户接受本《协议》的条款，否则本软件不得安装在用户的计算机上。

作为参考，用户现在就可以从本页起打印出本《协议》的文本。

本《协议》是用户与成都天心悦高科技发展有限公司之间有关随附本《协议》的数字有机体系统软件的法律协议。本软件包括随附的计算机软件，并可能包括相关文档印刷材料。用户一旦安装本“软件”，即表示用户同意接受本《协议》各项条款的约束。如用户不同意本《协议》中的条款，则用户不可以安装或使用本“软件”。

本“软件”受著作权法及国际著作权条约和其它知识产权法和条约的保护。本“软件”权利只许可使用，而不出售。

一、成都天心悦高科技发展有限公司将本软件在中国大陆地区（除香港、澳门、台湾外的中国境内）的非专有的使用权授予用户。用户可以：

1. 在一台计算机、工作站、终端机、手持式计算机或其它数字电子仪器（“计算机”）上安装、使用、显示、运行（“运行”）本“软件” 的一份副本。
2. 为了防止复制品损坏而制作备份复制品。这些备份复制品不得通过任何方式提供给他人使用，并在用户丧失该合法复制品的所有权时，负责将备份复制品销毁。
3. 为了把该软件用于实际的计算机应用环境或改进其功能、性能而进行必要的修改；但是，除合同另有约定外，未经成都天心悦高科技发展有限公司许可，不得向任何第三方提供修改后的软件。

二、用户保证：

1. 不在本协议规定的条款之外，使用、复制、修改、租赁或转让本软件或其中的任一部份。
2. 只在一台计算机上使用本软件；一份“软件”许可不得在不同的计算机共同或同时使用。

3. 只在以下之一前提下，将本软件用于多用户环境或网络软件。本软件明文许可用于多用户环境或网络软件上；或者，使用本软件的每一节点及终端都已购买使用许可。
4. 不得对本“软件”进行反向工程、反向编译或反汇编。
5. 不出租、租赁或出借本“软件”产品。
6. 在本“软件”的所有副本上包含所有的版权标识。

三、 软件转让:

用户可将用户在本《协议》项下的所有权利作永久性一次转让，转让后用户的许可权即自行终止。转让的条件是：

1. 用户不得保留副本。
2. 转让“软件产品”（包括全部组件、媒体及印刷材料，任何升级版本和本《协议》）的所有部分。
3. 受让人接受本《协议》的各项条款。
4. 如果“软件产品”为升级版本，任何转让必须包括本“软件产品”的所有前版本。

四、 支持服务:

1. 成都天心悦高科技发展有限公司为用户提供与“软件”有关的支持服务（“支持服务”）。
2. 支持服务的使用受用户手册或其它成都天心悦高科技发展有限公司提供的材料中所述的各项政策和计划的制约。
3. 提供给用户作为支持服务的一部分的任何附加软件代码应被视为本“软件”的一部分，并须符合本《协议》中的各项条款。
4. 用户提供给成都天心悦高科技发展有限公司作为支持服务的一部分的技术信息，成都天心悦高科技发展有限公司可将其用于商业用途，包括产品支持和开发。除了在为用户提供支持时必须的情况外，成都天心悦高科技发展有限公司在使用这些技术信息时不会以个人形式提及用户。

五、 软件的替换、修改和升级:

1. 成都天心悦高科技发展有限公司保留在任何时候通过为用户提供本“软件”的替换版本或修改版本或这类升级版本以替换、修改或使本“软件”升级的权利和为这类替换、修改或升级收取费用的权利。

2. 成都天心悦高科技发展有限公司提供给用户的本“软件”的任何替换版本或修改软件代码或升级版本，将被视为本“软件”的一部分并且要受到本《协议》条款的制约（除非本《协议》被随附本“软件”的替换或修改版本或升级版本的另外一份《协议》取代）。
3. 如果成都天心悦高科技发展有限公司提供本“软件”的一个替换或修改版本或任何升级版本，则用户对本“软件”的继续使用条件是用户接受本“软件”的这类替换或修改版本或升级版本以及任何随附的取代《协议》，并且就替换或修改版本的“软件”而言，用户对“软件”的所有先前版本的使用将被终止。

六、 权利的保留:

未明示授予的一切其它权利均为成都天心悦高科技发展有限公司所有。

七、 本“软件”的著作权:

1. 本“软件产品”及其所有复制品的名称，与光盘上或本软件中注明的公司同在。
2. 本“软件产品”（包括但不限于本“软件”中所含的任何图像、照片、动画、录像、录音、音乐、文字和附加程序）、随附的印刷材料、及本“软件”的任何副本的产权和著作权，均由成都天心悦高科技发展有限公司拥有。
3. 本软件及文档享有版权，并受国家版权法及国际协约条款的保护。
4. 用户不可以从本软件中去掉其版权声明；并保证为本软件的复制品（全部或部分）复制版权声明。用户同意制止以任何形式非法复制本软件及文档。
5. 用户不可复制本“软件”随附的印刷材料。

八、 出口限制:

用户同意不将本“软件”、其任何部分或任何属“软件”的直接成果的任何程序或服务出口或转口给任何中国大陆外的任何国家或者地区。

九、 售后担保:

1. 成都天心悦高科技发展有限公司担保，在正常使用的情况下，自售出之日起九十天内，其软件载体无材料或工艺缺陷。经验证确有缺陷时，成都天心悦高科技发展有限公司的全部责任就是退换其软件载体作为对用户的补偿。
2. 因事故、滥用或错误应用导致的载体缺陷，售后担保无效。
3. 退换的载体享受原担保期剩余时间，或三十天的担保，取其长者优先。
4. 除上述之外，本软件不享受任何其他形式的售后担保。

十、 责任有限:

上述担保，无论是明示或暗喻的，为担保的全部内容，包括对特殊应用目的的商品性和适应性担保。在适用法律所允许的最大范围内，成都天心悦高科技发展有限公司或其供应商绝不就因使用或不能使用本“软件”所引起的或有关的任何间接的、意外的、直接的、非直接的、特殊的、惩罚性的或其它任何损害赔偿（包括但不限于因人身伤害或财产损坏而造成的损害赔偿，因利润损失、营业中断、商业信息的遗失而造成的损害赔偿，因未能履行包括诚信或相当注意在内的任何责任致使隐私泄露而造成的损害赔偿，因疏忽而造成的损害赔偿，或因任何金钱上的损失或任何其损它损失而造成的损害赔偿）承担赔偿责任，即使成都天心悦高科技发展有限公司或其任何供应商事先被告知该损害发生的可能性。即使补救措施未能达到预定目的，本损害赔偿排除条款将仍然有效。

十一、 许可终止：

1. 如用户未遵守本《协议》的各项条款和条件，在不损害其它权利的情况下，成都天心悦高科技发展有限公司可终止本《协议》。终止《协议》时，用户必须立即销毁本软件的所有复制品，或者归还给成都天心悦高科技发展有限公司。
2. 通过向用户提供本“软件”或本“软件”的任何替换或修改版本或升级版本的一份取代《协议》，并规定用户继续使用本“软件”或这类替换、修改或升级版本的条件是用户接受这类取代《协议》，成都天心悦高科技发展有限公司可以终止本《协议》。

十二、 适用、管辖法律：

本协议适用《中华人民共和国著作权法》、《中华人民共和国计算机软件保护条例》、《中华人民共和国商标法》、《中华人民共和国专利法》等法律法规。本《协议》受中华人民共和国法律管辖。

至此，用户肯定已经详细阅读并已理解本协议，并同意严格遵守各条款和条件。

注：本软件附赠其它软件的知识产权以及法律责任由该附赠软件提供商享有及承担。