

组织: 中国互动出版网 (<http://www.china-pub.com/>)

RFC 文档中文翻译计划 (<http://www.china-pub.com/compters/emook/aboutemook.htm>)

E-mail: ouyang@china-pub.com

译者: 马东辉 (eaststone ma_donghui@263.net)

译文发布时间: 2001-4-4

版权: 本中文翻译文档版权归中国互动出版网所有。可以用于非商业用途自由转载, 但必须保留本文档的翻译及版权信息。

Network Working Group
Request for Comments: 1094

Sun Microsystems, Inc.
March 1989

RFC1094 网络文件系统协议

(RFC1094 NFS: Network File System Protocol Specification)

本备忘录状态

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

版权声明

Copyright (C) The Internet Society (1999). All Rights Reserved.

摘要:

网络文件系统可以使访问远程机上的目录和文件象在本地机上一样方便。本文就是介绍网络文件系统协议规范的中文版。

目录

1. 简介	2
1.1 远程过程调用	2
1.2 外部数据描述	2
1.3 无状态服务器	3
2. NFS 协议定义	3
2.1 文件系统模型	3
2.2 服务器过程	4
2.3 基本数据类型	12
3. NFS 实现中的问题	18
3.1 服务器/客户端 的关系	18
3.2 路径名解析	18
3.3 许可问题	18

3.4	RPC 信息.....	19
3.4	XDR 结构的尺寸	20
3.6	设置 RPC 的参数.....	20
附录 A	安装协议定义	21
A.1.	简介	21
A.2	RPC 信息.....	21
A.3	XDR 结构的尺寸	21
A.4	基本数据类型	22
A.5.	服务器过程	23
	作者地址.....	25

1. 简介

Sun 的网络文件系统(NFS)协议提供了对网络中的共享文件进行透明的远程访问。NFS 协议被设计为适合于不同的机器，操作系统，网络体系和传输协议。这种广泛的适应性是通过使用建立在外部数据描述(XDR)之上的远程过程调用（RPC）原语得到的。此协议的实现已经存在于从个人电脑到超级电脑等不同种类的机器之上。

对安装协议的支持允许服务器分发远程访问优先级给一个受限制的客户集。它执行了操作系统特定的功能，以允许把远程目录树链接在本地的文件系统上。

1.1 远程过程调用

Sun 的远程过程调用规范提供了一个面向过程的远程服务的接口。每一个服务器都提供了一个包含着一组过程的“程序”。NFS 就是一种这样的程序。主机地址，程序号和过程号的组合指定了一个远程过程。NFS 的一个目标就是不需要它的下层提供任何特定级别的可靠性。所以，它潜在地可以被使用在许多下层的传输层协议之上，甚至在另一个远程过程调用实现之上。为了便于讨论，本文档的剩余部分假定 NFS 实现在 Sun 的 RPC 上层。

1.2 外部数据描述

外部数据描述(XDR)标准提供了一个在网络上描述数据类型的公用方法。NFS 协议规范就是使用 RPC 数据描述语言撰写的。要想获得更多的信息，请参见 RFC 1014 “XDR:外部数据描述标准”。尽管存在自动化的 RPC/XDR 编译器可以产生服务器和客户端的“桩”（stubs）。NFS 也不需要它们。任何提供相同功能的软件都可以使用，如果编码完全相同的话，它也可以与其它 NFS 实现进行互操作。

1.3 无状态服务器

NFS 协议被希望尽可能无状态。也就是说，服务器应该不必保持关于它的客户端的任何协议状态信息，这是为了功能正确。在失败的事件发生的时候，无状态服务器比有状态服务器有着明显的优点。在无状态服务器中，客户端仅仅需要重发请求直到服务器响应；客户端甚至不需要知道服务器已经崩溃或者是网络临时故障。而有状态服务器的客户端要么需要检测服务器失败，并且在服务器恢复的时候重建服务器状态，要么使客户端操作失败。

这可能听起来不象是一个重要的问题，但是它在一些意想不到的情况下影响着协议。我们认为只要能写一个非常简易的服务器，不需要在崩溃后花费昂贵的代价恢复，即使在协议中多一些额外的复杂性也是值得的。注意：即使使用号称“可靠”的传输协议 TCP 的时候，客户端也必须能够处理当它们超时的时候再次打开连接所产生的服务的中断。因此，无状态协议实际上可以使这个实现简化。

另一方面，NFS 处理文件、目录这样本身就具有状态的对象。如果文件不保持它的内容没有被接触过会有什么好处呢？这样做的目的就是在协议本身不引入任何额外的状态。固有的状态操作，诸如文件或者记录锁定和远程执行都作为分开的服务实现，在此不讨论。

简化恢复的基本方法就是尽可能的采取“幂等”操作（为了它们有被重复的潜力）。这个协议版本中的一些操作并不能达到这个目的；幸运的是，大多数操作（例如 Read 和 Write）是幂等的。而且，多数服务器失败发生在操作之间，而不是发生在收到操作和响应之间。最后，尽管实际上服务器的失败可能很少，但是在复杂的网络中，任何网络，路由器或者网桥的失败与服务器的失败都是很难区分的。

2. NFS 协议定义

服务器随着时间改变，服务器使用的协议也一样。RPC 对每一个 RPC 请求都提供了一个版本号。RFC 已经定义了 NFS 协议的两个版本。即使在第二版中，也有少部分过时的过程和参数，这将在以后的版本中被删除。NFS 协议第三版的 RPC 当前正在准备之中。（译者注：这是相对此 RFC 文档发布的时间来讲的，此文档发表于 1989，3）

2.1 文件系统模型

NFS 假定文件系统是分层次的，除了最底层是文件，其它层次都是目录。在目录中的每一个条目（文件，目录，设备等）都有一个字符串名。不同的操作系统可能在目录树的深度或者使用的名字上有所限制，就象用不同的语义来描述“路径名”，它是在名字中把所有组成部分（目录和文件名）串联起来。一个“文件系统”就是在一个单一的服务器上（通常是一个磁盘或者物理分区）有一个指定的“根”的树。一些操作系统提供了“安装”操作使所有的文件系统出现在一棵单一的树上。而其它的操作系统保持着一个文件系统“森林”。文件是由无解释字节组成的无结构流。第三版的 NFS 使用更普遍的文件系统模型。

NFS 一次只查询路径名中的一个组成部分。为什么不一次就得到整个路径名，返回一个文件句柄呢？这里有一些不这样做的原因。首先，路径名需要在路径的组成部分之间有分隔符。不同的操作系统使用不同的分隔符。我们可以定义一种网络上标准的路径表示法，但是每一个路径名在每一个终点上将必须进行语法分析和转换。其它的问题在第三节（NFS 实现中的问题）里讨论。

尽管文件和目录在许多方面是相似的对象，但是读目录和读文件也需要不同的过程。这里提供了描述目录的网络标准格式。使用象上面相同的参数来确定一个过程，此过程在每次调用的时候只返回一个目录项。这种方法产生的问题就是效率不高。目录包含着许多目录项，远程调用要返回每一项将是非常缓慢的。

2.2 服务器过程

这个协议被定义为一组过程，这组过程具有用 RPC 语言（XDR 语言在程序，版本，过程声明方面的扩展）定义的参数和结果。每一个过程功能的简要描述都应该提供足够允许实现的信息。2.3 节详细地描述了基本数据类型。

在 NFS 协议中的所有过程都假定是同步的。当一个过程返回给客户端，客户可以假定此操作已经完成，与请求相关的任何数据现在在一个稳定的存储上。例如，客户端的 WRITE 请求可能导致服务器更新数据块，文件系统信息块（比如间接块），和文件属性信息（大小和修改时间）。当 WRITE 返回给客户端，客户端假定这个写操作是可靠的。甚至在服务器崩溃的情况下，它也能丢弃这些已经写的数据。这就是服务器无状态的一个非常重要的部分。如果服务器等待来自远程请求的刷新数据，客户端必须保存这些请求，以便在服务器崩溃的情况下再次发送这些请求。

```

/*
 * 远程文件服务程序
 */
program NFS_PROGRAM {
    version NFS_VERSION {
        void      NFSPROC_NULL(void)           = 0;

        attrstat  NFSPROC_GETATTR(fhandle)     = 1;

        attrstat  NFSPROC_SETATTR(sattrargs)   = 2;

        void      NFSPROC_ROOT(void)           = 3;

        diropres  NFSPROC_LOOKUP(diropargs)    = 4;

        readlinkres  NFSPROC_READLINK(fhandle) = 5;
    }
}

```

```

    readres    NFSPROC_READ(readargs)      = 6;

    void       NFSPROC_WRITECACHE(void)    = 7;

    attrstat   NFSPROC_WRITE(writeargs)    = 8;

    diropres   NFSPROC_CREATE(createargs)  = 9;

    stat       NFSPROC_REMOVE(diropargs)   = 10;

    stat       NFSPROC_RENAME(renameargs)  = 11;

    stat       NFSPROC_LINK(linkargs)      = 12;

    stat       NFSPROC_SYMLINK(symlinkargs) = 13;

    diropres   NFSPROC_MKDIR(createargs)   = 14;

    stat       NFSPROC_RMDIR(diropargs)    = 15;

    readdirres NFSPROC_READDIR(readdirargs) = 16;

    statfsres  NFSPROC_STATFS(fhandle)     = 17;
    } = 2;
} = 100003;

```

2.2.1 不做工作

```
void          NFSPROC_NULL(void) = 0;
```

这个过程不做工作，在所有 RPC 服务中它可以用来允许服务器响应测试和定时。

2.2.2 获得文件属性

```
attrstat     NFSPROC_GETATTR (fhandle) = 1;
```

如果响应状态是 NFS_OK，那么响应属性包含由输入 fhandle 指定的文件的属性。

2.2.3 设置文件属性

```
struct sattrargs {
    fhandle file;
```

```

        sattr attributes;
    };

    attrstat    NFSPROC_SETATTR (sattrargs) = 2;

```

"attributes"值参数包含着一些字段，这些字段要么是 -1，要么是“file”的文件属性的一个新值。如果响应状态是 NFS_OK，那么响应属性在"SETATTR"操作完成之后具有文件的属性。

注意： -1 指示在“attributes”中一个没有使用的字段，在协议的下一版本将修改。

2.2.4 获得文件系统的根

```

    void        NFSPROC_ROOT(void) = 3;

```

已经过时。这个过程不再使用，因为找到一个文件系统的根文件句柄需要在客户端和服务端之间移动路径名。为了正确的做到这一点，我们必须定义一个路径名网络标准描述。查询根文件句柄已经由 MNTPROC_MNT 过程来实现。（详细情况请参见附录 A，“安装协议定义”）

2.2.5. 查询文件名

```

    diropres    NFSPROC_LOOKUP(diropargs) = 4;

```

如果响应"status"是 NFS_OK,响应“file”和响应“attributes”是参数“dir”给定的目录中的文件名的文件句柄和属性。

2.2.6 从符号链接读

```

    union readlinkres switch (stat status) {
    case NFS_OK:
        path data;
    default:
        void;
    };
    readlinkres    NFSPROC_READLINK(fhandle) = 5;

```

如果"status"的值是 NFS_OK,响应“data”是 fhandle 参数引用的文件的符号链接中的数据。

注意：因为 NFS 总是在客户端解析路径名，如果在不同的客户端或者服务器上使用不同的语义，那么在一个符号链接中的路径名可能有不同的含义（或者无意义）。

2.2.7 从文件中读

```
struct readargs {
    fhandle file;
    unsigned offset;
    unsigned count;
    unsigned totalcount;
};

union readres switch (stat status) {
case NFS_OK:
    fattr attributes;
    nfsdata data;
default:
    void;
};

readres      NFSPROC_READ(readargs) = 6;
```

在由“file”给出的文件中，从“offset”字节偏移开始返回“count”个字节的“data”。这个文件的第一个字节是偏移量 0。在读操作发生后，文件属性从“attributes”中返回。

注意：参数“totalcount”没有使用，在协议的下一修订版中将删除。

2.2.8 写到缓冲区

```
void      NFSPROC_WRITECACHE(void) = 7;
```

将在协议的下一修订版中使用。

2.2.9 写到文件

```
struct writeargs {
    fhandle file;
    unsigned beginoffset;
    unsigned offset;
    unsigned totalcount;
};
```

```

        nfsdata data;
    };

    attrstat    NFSPROC_WRITE(writeargs) = 8;

```

从“file”开头偏移的“offset”字节处开始写数据“data”。文件的第一个字节是在偏移 0 的位置。如果响应状态“status”是 NFS_OK,那么在写操作完成后响应属性“attributes”中包含着文件的属性。写操作是原子的，从这次“WRITE”中写入的数据不会与客户端的另一次“WRITE”写入的数据混合在一起。

注意：参数“beginoffset”和“totalcount”被忽略，在协议的下一修订版中将被删除。

2.2.10 创建文件

```

    struct createargs {
        diropargs where;
        sattr attributes;
    };

    diopres    NFSPROC_CREATE(createargs) = 9;

```

文件“name”创建在由“dir”指定的目录中。新文件的初始属性由“attributes”决定。NFS_OK 的响应状态表明这个文件被创建。响应“file”和响应“attributes”是这个文件的文件句柄和属性。任何其它的响应状态“status”都意味着此操作失败，没有文件被创建。

注意：这个例程可以传递一个排它的创建标志，意味着“仅在文件不存在的时候创建这个文件”。

2.2.11 删除文件

```

    stat    NFSPROC_REMOVE(diropargs) = 10;

```

文件“name”从“dir”确定的目录中删除。NFS_OK 的响应意味着这个目录项被删除。

注意：可能不是幂等地操作。

2.2.12 重命名文件

```

    struct renameargs {
        diropargs from;

```

```

        diropargs to;
    };

    stat    NFSPROC_RENAME(renameargs) = 11;

```

在"from.dir"目录中的"from.name"文件被更名为"to.dir"目录中的文件名"to.name"。如果响应是 NFS_OK,文件被更名。更名操作在服务器上是一个原子操作；它不能在执行中被中断。

注意：可能不是幂等地操作。

2.2.13 创建文件链接

过程 12, 版本 2。

```

    struct linkargs {
        fhandle from;
        diropargs to;
    };

    stat    NFSPROC_LINK(linkargs) = 12;

```

在"to.dir"目录中创建文件"to.name", 它是已存在的文件"from"的一个硬链接。如果返回值是 NFS_OK,一个链接创立。如何返回其它的值表明出错, 这个链接没有创建。

硬链接应该具有这样的属性, 链接的文件中的任何一个改变都将影响到两个文件。当硬链接指向一个文件的时候, 文件属性中应该有一个表示"nlink"的值, 这个值比链接前大。

注意：可能不是幂等地操作。

2.2.14 创建符号链接

```

    struct symlinkargs {
        diropargs from;
        path to;
        sattr attributes;
    };

    stat    NFSPROC_SYMLINK(symlinkargs) = 13;

```

在给定的目录"from.dir"中创建一个文件类型是 NFLNK 的文件"from.name"。这个新文件包含着路径名 "to", 具有"attributes"指定的初始属性。如果返回值是 NFS_OK,一个链接被创建。任何其它的返回值指示错误, 链接没有创建。

符号链接是指向另一个文件的指针。在"to"中给定的名字不被服务器解释，只存储在新建的文件中。当客户端引用一个符号链接文件的时候，符号链接中的内容通常作为一个代替的路径名重新被解释。READLINK 的操作返回给客户端要解释的数据。

注意：在 UNIX 服务器上，attributes 从不使用，因为符号链接总是具有 0777 的模式。

2.2.15 创建目录

```
diropres      NFSPROC_MKDIR (createargs) = 14;
```

新目录"where.name"创建在给定的目录"where.dir"中。新目录的初始属性由"attributes"确定。一个 NFS_OK 的返回值表示新目录被创建，响应"file"和响应"attributes"是这个新目录的文件句柄和属性。返回任何其它的响应状态"status"都意味着操作失败，没有目录被创建。

注意：可能不是幂等地操作。

2.2.16 删除目录

```
stat          NFSPROC_RMDIR(diropargs) = 15;
```

在"dir"指定的目录中的空的目录"name"将被删除，如果响应是 NFS_OK,目录被删除。

注意：可能不是幂等地操作。

2.2.17 从目录中读

```
struct readdirargs {
    fhandle dir;
    nfscookie cookie;
    unsigned count;
};

struct entry {
    unsigned fileid;
    filename name;
    nfscookie cookie;
    entry *nextentry;
};
```

```

union readdirres switch (stat status) {
case NFS_OK:
    struct {
        entry *entries;
        bool eof;
    } readdirok;
default:
    void;
};

readdirres    NFSPROC_READDIR (readdirargs) = 16;

```

从“dir”指定的目录中返回一个可变数目的目录项集合，总的大小是“count”个字节。如果返回的状态值“status”是 NFS_OK,那么后跟一组可变数目的“entry”。每一个“entry”包含着一个“fileid”，这个“fileid”是由文件系统中唯一标识这个文件的号码，文件名和一个指向目录中下一个目录项的不透明指针“cookie”组成。Cookie 使用在下面的 READDIR 调用中，以便在这个目录中从一个指定的开始点获得更多的目录项。特殊的 cookie 值 0（所有比特都是 0）使得从目录的开始点得到目录项。“fileid”字段应该和在文件属性中的“fileid”字段中有同样的值。（见“基本数据类型”中的 2.3.5 节“fattr”）如果在目录中没有更多的目录项，“eof”标志的值是 TRUE。

2.2.18 获得文件系统属性

```

union statfsres (stat status) {
case NFS_OK:
    struct {
        unsigned tsize;
        unsigned bsize;
        unsigned blocks;
        unsigned bfree;
        unsigned bavail;
    } info;
default:
    void;
};

statfsres    NFSPROC_STATFS (fhandle) = 17;

```

如果响应状态“status”是 NFS_OK,那么响应“info”给出包含着由输入文件句柄 fhandle 引用的文件的文件系统的属性。属性字段包含着下列值：

tsize 用字节表示的最优化的传输尺寸。这是服务器在 READ 和 WRITE 请求中的最想要的的数据字节数。

bsize 文件系统用字节表示的块尺寸。

blocks 文件系统中 "bsize"块的总数。

bfree 文件系统中自由的“bsize”块的数目。

bavail 无特权用户可用的"bsize"块的数目。

注意：如果文件系统具有可变尺寸的块，这个调用不能很好的工作。

2.3 基本数据类型

以下 XDR 定义是在描述其它结构中使用的基本的结构和类型。

2.3.1. stat (统计类型)

```
enum stat {
    NFS_OK = 0,
    NFSERR_PERM=1,
    NFSERR_NOENT=2,
    NFSERR_IO=5,
    NFSERR_NXIO=6,
    NFSERR_ACCES=13,
    NFSERR_EXIST=17,
    NFSERR_NODEV=19,
    NFSERR_NOTDIR=20,
    NFSERR_ISDIR=21,
    NFSERR_FBIG=27,
    NFSERR_NOSPC=28,
    NFSERR_ROFS=30,
    NFSERR_NAMETOOLONG=63,
    NFSERR_NOTEMPTY=66,
    NFSERR_DQUOT=69,
    NFSERR_STALE=70,
    NFSERR_WFLUSH=99
};
```

在每一个过程调用结果中都有统计类型“stat”被返回。NFS_OK 的值表示调用执行成功，结果有效。其它值表示服务器一侧在过程服务中产生的某种错误。这些错误值来源于 UNIX 错误号。

NFSERR_PERM

不是所有者，调用者不是所请求操作的正确的所有者。

NFSERR_NOENT

不存在这样的文件或者目录。指定的文件或者目录不存在。

NFSERR_IO

在操作执行的时候出现某种硬件错误。例如，这可能是一个磁盘错误。

NFSERR_NXIO

没有这样的设备或者地址。

NFSERR_ACCES

许可权限拒绝。调用者没有执行请求操作的正确的权限。

NFSERR_EXIST

文件存在。指定的文件已经存在。

NFSERR_NODEV

没有这样的设备。

NFSERR_NOTDIR

不是一个目录。在目录操作中调用者指定一个非目录。

NFSERR_ISDIR

是一个目录。调用者在一个非目录操作中指定一个目录。

NFSERR_FBIG

文件太大。操作造成文件增长超过服务器的限制。

NFSERR_NOSPC

在设备上没有剩余的空间。这个操作导致服务器文件系统达到它的极限。

NFSERR_ROFS

只读文件系统。在一个只读文件系统上试图写。

NFSERR_NAMETOOLONG

文件名太长。在操作中文件名太长。

NFSERR_NOTEMPTY

目录不空。试图删除一个不空的目录。

NFSERR_DQUOT

磁盘限额超出。客户在服务器上的磁盘限额已经超出。

NFSERR_STALE

在参数中给的文件句柄"fhandle"无效。也就是说, 这个文件句柄引用的文件不再存在。或者访问它的设置已经被撤销。

NFSERR_WFLUSH

使用"WRITECACHE"调用中的服务器写缓冲区得到磁盘刷新。

2.3.2. ftype (文件类型)

```
enum ftype {  
    NFNON = 0,  
    NFREG = 1,  
    NFDIR = 2,  
    NFBLK = 3,  
    NFCHR = 4,  
    NFLNK = 5  
};
```

枚举"ftype"类型给出文件的类型。NFNON 类型表示不是一个文件, NFREG 表示一个正常的文件, NFDIR 表示一个目录。NFBLK 表示一个特定的块设备, NFCHR 表示一个特定的字符设备, NFLNK 表示一个符号链接。

2.3.3. fhandle (文件句柄)

```
typedef opaque fhandle[FHSIZE];
```

"fhandle"是一个在服务器和客户端之间传送的文件句柄。所有文件操作都使用文件句柄来引用一个文件或者目录。文件句柄包含着服务器需要的区分一个单独文件的信息。

2.3.4. timeval (时间值)

```
struct timeval {  
    unsigned int seconds;  
    unsigned int useconds;  
};
```

"timeval"结构是一个秒和微秒的数值, 从格林威治时间 1970 年一月一日凌晨起计时。它使用在传递时间和日期的信息中。

2.3.5. fattr (文件属性)

```

struct fattr {
    ftype      type;
    unsigned int mode;
    unsigned int nlink;
    unsigned int uid;
    unsigned int gid;
    unsigned int size;
    unsigned int blocksize;
    unsigned int rdev;
    unsigned int blocks;
    unsigned int fsid;
    unsigned int fileid;
    timeval    atime;
    timeval    mtime;
    timeval    ctime;
};

```

"fattr"结构包含着文件的属性；"type"是文件的类型；"nlink"是一个文件的硬链接数（对同一个文件不同的名字的数目）；"uid"是这个文件的所有者的用户标识号码；"gid"是拥有这个文件的组的组标识号码；"size"是这个文件以字节数计算的大小；"blocksize"是这个文件的一个块以字节计数的大小；如果文件的类型是 NFCHR 或者 NFBLK, "rdev"是这个文件的设备号；"blocks"是文件在磁盘上块的数量；"fsid"是包含这个文件的文件系统的系统标识符；"fileid"是这个文件在它的文件系统中唯一的标识符号码；"atime"是上次文件读访问或者写访问的时间；"mtime"是文件数据上次被修改时的时间（写）；"ctime"是文件状态上次改变的时间。如果文件的尺寸改变，写一个文件也将改变"ctime"。

"Mode"是被编码成一个比特集合的访问模式。注意文件类型要么在模式比特中指定，要么在文件类型中指定。这实际上是此协议中的一个缺陷，将在未来的版本中修订。下面的描述使用八进制数确定比特的位置

0040000 这是一个目录，"type"字段应该是 NFDIR。
 0020000 这是一个字符特殊文件，"type"字段应该是 NFCHR。
 0060000 这是一个块特殊文件，"type"字段应该是 NFBLK。
 0100000 这是一个正常的文件，"type"字段应该是 NFREG。
 0120000 这是一个符号链接文件，"type"字段应该是 NFLNK。
 0140000 这是一个命名的 socket；"type"字段应该是 NFNON。
 0004000 设置在执行中的用户 ID
 0002000 设置在执行中的组 ID
 0001000 在使用后保存交换文本。
 0000400 对所有者的读权限许可。

0000200 对所有者的写权限许可。
0000100 对所有者的执行和搜索权限许可。
0000040 对组的读权限许可。
0000020 对组的写权限许可。
0000010 对组的执行和搜索权限许可。
0000004 对其他人的读权限许可。
0000002 对其他人的写权限许可。
0000001 对其他人的执行和搜索权限许可。

注意：这些比特与 UNIX 中 `stat(2)` 系统调用中返回的模式比特是一样的。文件类型要么在模式比特中要么在文件类型中确定。这在未来的版本中将修改。

在属性结构中 `"rdev"` 字段是一个操作系统特定设备的标识符。在这个协议的下一个修订版中将删除。

2.3.6. `sattr` (设置文件属性)

```
struct sattr {
    unsigned int mode;
    unsigned int uid;
    unsigned int gid;
    unsigned int size;
    timeval      atime;
    timeval      mtime;
};
```

`"sattr"` 结构包含着可以从客户端设置的文件的属性。这些字段与上面的 `"fattr"` 中的字段的含义是相同的。`"size"` 值为 0 意味着文件将被截短。一个 `-1` 的值意味着这个字段将被忽略。

2.3.7. `filename` (文件名)

```
typedef string filename<MAXNAMLEN>;
```

`"filename"` 用来传送文件名或者路径名的组成部分。

2.3.8. `path` (路径)

```
typedef string path<MAXPATHLEN>;
```

`"path"` 是一个路径名。服务器把它作为一个没有内部结构的字符串，但是对客户端来

说它是文件系统树中的节点的名字。

2.3.9. attrstat (属性状态)

```
union attrstat switch (stat status) {
case NFS_OK:

    fattr attributes;
default:
    void;
};
```

"attrstat"结构是一个公共的过程结果。它包含着一个"status", 如果这个调用成功, 它将也包含操作作用于那个文件的属性。

2.3.10. diropargs (目录操作参数)

```
struct diropargs {
    fhandle dir;
    filename name;
};
```

"diropargs"结构使用在目录属性操作中。 "dir" 是在其中寻找文件"name"的那个目录。一个目录操作影响这个目录。

2.3.11. diopres (目录操作结果)

```
union diopres switch (stat status) {
case NFS_OK:
    struct {
        fhandle file;
        fattr attributes;
    } diropok;
default:
    void;
};
```

在"diopres"中返回目录操作的结果。如果这个调用成功, 与这个文件相关的新文件句柄"file"和文件属性"attributes"将连同"status"一起返回。

3. NFS 实现中的问题

NFS 协议设计为允许不同的操作系统共享文件。但是，因为它在 UNIX 环境中设计，所以，许多操作与 UNIX 文件系统的操作语义相似。这一节讨论实现中特别的细节和语义的问题。

3.1 服务器/客户端 的关系

NFS 协议被设计为让服务器尽可能的简单和通用。有时候如果客户端想实现复杂的文件系统语义，服务器的简单性可能是一个难题。

例如，一些操作系统允许删除打开的文件。一个进程能够打开文件，当文件打开的时候，从目录中删除它。只要进程保持文件打开，这个文件就可以被读写，即使这个文件在文件系统中没有名字。对于无状态服务器来说，实现这些语义是不可能的。客户端可以使用一些技巧，诸如在删除时重命名这个文件，只在它关闭的时候删除它。我们相信服务器提供了足够的功能在客户端上实现大多数文件系统的语义。

每一个 NFS 客户端也是一个潜在的服务器，远程和本地安装的文件系统可以自由的混合在一起。当客户端浏览远程文件系统的目录树，到达在服务器上另一个远程系统的安装点的时候，将会出现一些有趣的问题。要允许服务器跟随第二个远程安装就需要循环检测，服务器查找和用户重新生效。代替的做法是，我们决定不让客户端越过服务器的安装点。当客户端在一个服务器已经安装了文件系统的目录中查询的时候，客户端将看见下面的目录而不是安装的目录。

例如，如果服务器有一个叫做"/usr"的文件系统，把另一个文件系统安装在"/usr/src"上，如果客户端安装了"/usr"，它将看不到"/usr/src"的安装版本。客户端能做远程安装以配合服务器的安装点保持服务器的视图。在这个例子中，客户端应该除了安装"/usr"之外，还要安装"/usr/src"，即使它们是来自同一台服务器。

3.2 路径名解析

路径名总是在客户端解析的规则有点复杂。例如，符号链接在不同的客户端可以有不同的解释。对于非 UNIX 实现的另一个共同的问题就是路径".."的专门的解释是给定的目录的父目录。此协议的下一个修订版将使用一个明确的标志指示父目录。

3.3 许可问题

严格的讲，NFS 协议没有定义服务器使用的许可权限检查。但是，也希望使用

AUTH_UNIX 类型认证这一基础的保护机制作正常的操作系统许可权限检查, 服务器得到客户的有效"uid", 有效"gid"和每次调用上的组, 使用它们来检查许可。使用这种方法产生的问题可以用一些有趣的途径来解决。

使用"uid" 和 "gid"暗示着客户端和服务器分享相同的"uid"列表。每一对服务器和客户端必须有相同的用户到"uid", 组到"gid"的映射。因为每一个客户端也可能是一台服务器, 这意味着整个网络共享相同的"uid/gid"空间。AUTH_DES (和 NFS 协议的下一个修订版) 使用字符串来代替数字, 但是仍有复杂的问题要解决。

由于打开操作有状态, 所以产生了另一个问题。大多数操作系统在打开文件的时候检查许可权限, 在每一次读写请求的时候检查文件是否打开。在无状态的服务器中, 服务器没有办法知道文件是否打开, 必须在每次读写调用的时候检查许可权限。在一个本地文件系统中, 用户可以打开文件, 然后改变权限不允许别人接触它, 但是仍然能够写文件, 因为文件是打开的。相反, 在远程文件系统中, 写操作将失败。为了避免这种问题, 服务器的许可检查算法将允许文件的所有者访问文件, 而不管许可的设置。

在从网络中的文件上进行页面调度的时候, 也会出现相似的问题。操作系统在打开一个文件进行页面调度之前, 总是检查执行许可权限, 然后从打开的文件中读取块。文件可能没有读许可权限, 但是在文件打开后, 这就不是一个问题了。NFS 服务器不能区分在正常文件读和页面调入请求读之间的区别。为了使这个可以工作, 如果在调用中被给的"uid"在文件上有执行或者读许可权限, 服务器将允许读文件。

在大多数操作系统中, 一个特别的用户(在 UNIX 上, 用户 ID 为 0)有访问所有文件的权限, 而不管文件中的所有权和设定的许可权限。"super-user"权限在服务器上不可能被允许, 因为在自己工作站上的任何具有超级用户权限的人都能访问所有的远程文件。UNIX 服务器在访问检查前, 默认把用户 ID 0 映射为 -2。这个工作在 NFS 的根文件系统中例外, 在那里超级用户访问不能避免。

3.4 RPC 信息

认证

NFS 服务使用 AUTH_UNIX, AUTH_DES 或者 AUTH_SHORT 类型的认证, 在 NULL 过程中例外, 在那里 AUTH_NONE 也被允许。

传输协议

NFS 通常由 UDP 支持。

端口号

NFS 协议当前使用 UDP 端口号 2049。这不是一个正式分配的端口号, 所以, 这个协议的后继版本使用 RPC 的“端口映射”工具。

3.4 XDR 结构的尺寸

这里有一些使用在此协议中不同的 XDR 结构的尺寸，用十进制字节给出。

```
/*  
 * 在读写请求中的数据的最小字节数。  
 */  
const MAXDATA = 8192;  
  
/*在路径参数中的最大字节数 */  
const MAXPATHLEN = 1024;  
  
/*在文件名参数中的最大字节数*/  
const MAXNAMLEN = 255;  
  
/*被 READDIR 传送的"cookie"字节数的大小*/  
const COOKIESIZE = 4;  
  
/*不透明文件句柄的字节数的大小*/  
const FHSIZE = 32;
```

3.6 设置 RPC 的参数

不同的文件系统参数和选项应该在安装的时候设置。安装协议在附录中描述。例如，象“硬”安装一样，“软”安装也被提供。当 RPC 操作失败（在给出一个重传的选项号后），软安装文件系统返回错误，而硬安装文件系统一直继续重传。最大的传输尺寸依赖于实现。对于在一个本地网的有效操作来说，通常使用 8192 字节的数据。这可能导致下层的分段（诸如在 IP 层）。既然一些网络接口不允许这样的包，对于在低速网络、主机上的操作，或者通过网关的操作，512 或 1024 字节总是提供较好的结果。

客户机和服务器可能需要把当前的操作保存在缓冲区中，以帮助避免因为非幂等的操作产生的问题。例如，如果传输协议丢失了删除文件操作的响应，在重传的时候，服务器可能返回一个 NFSERR_NOENT 来代替 NFS_OK。但是，如果服务器保持上次的请求操作和结果，它可能返回正确的成功的代码。当然，服务器在重传之间可能崩溃、重启。但是一块很小的缓冲区（甚至只是容纳一个条目）将解决大部分的问题。

附录 A 安装协议定义

A.1. 简介

安装协议与 NFS 协议分离，但是与 NFS 协议相关。它提供了操作系统特定的服务来扩展 NFS 的功能：查询服务器路径名，使用户身份有效，检查访问权限。客户端使用安装协议得到第一个文件句柄，这允许客户进入一个远程的文件系统。

安装协议与 NFS 协议保持分离，使得在不改变 NFS 服务器协议的情况下很容易加进新的访问检查和确认的方法。

注意：这个协议的定义暗示着服务器有状态，因为服务器保持着一个客户端安装请求的列表。安装列表信息对客户机或者服务器的功能没有危害性。仅仅建议在服务器出现故障时，给客户机可能的警告。

安装协议的第一版与 NFS 协议的第二版一起使用，在两个协议中唯一通信的信息是 "fhandle" 结构。

A.2 RPC 信息

认证

安装服务仅使用 AUTH_UNIX 和 AUTH_NONE 类型的认证

传输协议

安装服务被 UDP 和 TCP 中支持。

端口号

与服务器的端口映射器协商，来发现安装服务注册的端口号。

A.3 XDR 结构的尺寸

这里有使用在此协议中不同的 XDR 结构的尺寸，用十进制表示。

```
/* 路径名参数的最大字节数*/  
const MNTPATHLEN = 1024;
```

```
/*一个名字参数的最大字节数 */  
const MNTNAMLEN = 255;
```

```
/* 不透明文件句柄的字节数*/
```

```
const FHSIZE = 32;
```

A.4 基本数据类型

这一节描述使用在安装协议中的数据类型。在许多情况下它们与使用在 NFS 中的数据类型相似。

A.4.1. fhandle

```
typedef opaque fhandle[FHSIZE];
```

"fhandle"类型是服务器传递给客户端的文件句柄。所有文件操作都使用文件句柄来引用文件或者目录。文件句柄包含着服务器需要的区分单个文件的信息。

这与 NFS 协议第二版中的"fhandle"XDR 定义一样；见在 "基本数据类型"之中的"2.3.3. fhandle"。

A.4.2. fhstatus

```
union fhstatus switch (unsigned status) {  
  case 0:  
    fhandle directory;  
  default:  
    void;  
}
```

"fhstatus"类型是一个联合结构。如果返回的"status"为 0，调用执行成功，接着的是给"directory"分配的文件句柄。一个非 0 值表示一些种类的错误。在这种情况下，status 是一个 UNIX 错误号。

A.4.3. dirpath

```
typedef string dirpath<MNTPATHLEN>;
```

"dirpath"类型是一个目录的服务器路径名。

A.4.4. name

```
typedef string name<MNTNAMLEN>;
```

"name"类型是一个使用在不同名字中的二进制串。

A.5. 服务器过程

下面的部分定义了安装服务器提供的 RPC 过程。

```

/*
 *安装程序的协议定义
 */
program MOUNTPROG {
    /*
     *安装协议的第一版与 NFS 协议的第二版使用在一起。
     */
    version MOUNTVERS {

        void    MOUNTPROC_NULL(void) = 0;

        fhstatus MOUNTPROC_MNT(dirpath) = 1;

        mountlist    MOUNTPROC_DUMP(void) = 2;

        void        MOUNTPROC_UMNT(dirpath) = 3;

        void        MOUNTPROC_UMNTALL(void) = 4;

        exportlist    MOUNTPROC_EXPORT(void) = 5;
    } = 1;
} = 100005;

```

A.5.1. 不作工作

```
void    MNTPROC_NULL(void) = 0;
```

这个过程什么也不作，在所有的 RPC 服务中使用它来允许服务器响应测试和定时。

A.5.2 加入安装条目

```
fhstatus    MNTPROC_MNT(dirpath) = 1;
```

如果响应状态"status"是 0，那么响应的"directory"包含着目录"dirname"的文件句柄。这个文件句柄可能使用在 NFS 协议中。这个过程也在客户安装的"dirname"安装列表中加一个新的项目。

A.5.3 返回安装条目

```

struct *mountlist {
    name      hostname;
    dirpath   directory;
    mountlist nextentry;
};

mountlist    MNTPROC_DUMP(void) = 2;

```

返回一个远程安装文件系统的列表。"mountlist"对每一对"hostname"和"directory"都包含着一个条目。

A.5.4. 删除安装条目

```

void        MNTPROC_UMNT(dirpath) = 3;

```

从输入"dirpath"中删除安装列表条目

A.5.5. 删除所有的安装条目

```

void        MNTPROC_UMNTALL(void) = 4;

```

为客户端删除所有安装列表的条目

A.5.6. 返回输出列表

```

struct *groups {
    name gname;
    groups grnext;
};

struct *exportlist {
    dirpath filesys;
    groups groups;
    exportlist next;
};

exportlist    MNTPROC_EXPORT(void) = 5;

```

返回一组可变数目的输出列表条目。每一个条目包含着文件系统名和一个被允许导入的组列表。文件系统名在"filesys"当中，组名在列表"groups"中。

注意：输出列表应该包含关于文件系统状态的更多的信息，例如只读标志。

作者地址

Bill Nowicki
Sun Microsystems, Inc.
Mail Stop 1-40
2550 Garcia Avenue
Mountain View, CA 94043

Phone: (415) 336-7278

Email: nowicki@SUN.COM