sysfs - _The_ filesystem for exporting kernel objects.

sysfs - 用于导出内核对象（kobject）的文件系统

Patrick Mochel <mochel@osdl.org>
翻译： tekkamanninja <tekkamanninja@163.com>

10 January 2003

2003年1月10日

翻译时间：2007年12月29日

What it is:

简介：

~~~~~~~~~~~

sysfs is a ram-based filesystem initially based on ramfs. It provides

sysfs 是一个最初基于ramfs的位于内存的文件系统。它提供

a means to export kernel data structures, their attributes, and the

一些方法以导出内核的数据结构、他们的属性和

linkages between them to userspace.

他们与用户空间的连接。

sysfs is tied inherently to the kobject infrastructure. Please read

sysfs 始终与kobject的底层结构紧密相关。请阅读

Documentation/kobject.txt for more information concerning the kobject

Documentation/kobject.txt 文档以获得更多关于 kobject 接口的信息。

interface.

Using sysfs

使用

~~~~~~~~~~~

sysfs is always compiled in. You can access it by doing:

sysfs 通常被编译进内核。你可以通过使用以下命令访问它：

```
mount -t sysfs sysfs /sys
```

（此命令含义是挂载 sysfs 到根目录下的sys目录）

Directory Creation

创建目录

~~~~~~~~~~~~~~~~~~

For every kobject that is registered with the system, a directory is

一旦有 kobject 在系统中注册，就会有一个目录在sysfs中被创建。

created for it in sysfs. That directory is created as a subdirectory
~~~~~~~~~~~~~~~~~~

这个目录是作为 kobject 的 parent 下的子目录创建的，
of the kobject's parent, expressing internal object hierarchies to

以准确的传递内核的对象层次到
userspace. Top-level directories in sysfs represent the common
用户空间。 sysfs中的顶层目录代表着内核对象层次的共同祖先；
ancestors of object hierarchies; i.e. the subsystems the objects

例如：某些对象属于某个子系统。
belong to.

Sysfs internally stores the kobject that owns the directory in the
Sysfs内部存储着 kobject ，这些 kobject 在 d_fsdata 指针（在 kobject
->d_fsdata pointer of the directory's dentry. This allows sysfs to do

的dentry结构体中）中拥有目录。 这使得 sysfs 可以在文件
reference counting directly on the kobject when the file is opened and
打开和关闭时，直接在 kobject 上实现引用计数。
closed.

Attributes
属性
~~~~~~~~~~

Attributes can be exported for kobjects in the form of regular files in
kobject 的属性能在文件系统中以普通文件的形式导出。
the filesystem. Sysfs forwards file I/O operations to methods defined

Sysfs 为属性定义了面向文件 I/O 操作的方法，
for the attributes, providing a means to read and write kernel

以提供对内核属性的读写。
attributes.

Attributes should be ASCII text files, preferably with only one value
属性应为 ASCII 码文本文件，以一个文件只存储一个属性值为宜。
per file. It is noted that it may not be efficient to contain only

但一个文件只包含一个属性值可能影响效率，
value per file, so it is socially acceptable to express an array of

所以一个包含相同数据类型的属性值数组也是被广泛接受的。
values of the same type.

Mixing types, expressing multiple lines of data, and doing fancy
混合类型、表达多行数据以及一些怪异的数据格式是会遭强烈反对。
formatting of data is heavily frowned upon. Doing these things may get

这样做是很丢脸的，而且
you publically humiliated and your code rewritten without notice.

你的代码会在未通知你的情况下被重写。

An attribute definition is simply:
~~~~~~~~~~

一个简单的属性结构定义如下：（到2.6.22.2已添加了struct module * owner;）

```
struct attribute {
        char                    * name;
        mode_t                  mode;
};
```

```
int sysfs_create_file(struct kobject * kobj, struct attribute * attr);
void sysfs_remove_file(struct kobject * kobj, struct attribute * attr);
```

A bare attribute contains no means to read or write the value of the
一个裸的属性并不包含读写其属性值的方法。
attribute. Subsystems are encouraged to define their own attribute

最好为子系统定义自己的属性
structure and wrapper functions for adding and removing attributes for

和 为了增删特殊对象类型的属性而 包装过的函数。
a specific object type.

For example, the driver model defines struct device_attribute like:
例如：驱动程序模型定义的device_attribute 结构体如下：

```
struct device_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device * dev, char * buf);
        ssize_t (*store)(struct device * dev, const char * buf);
};
```

```
int device_create_file(struct device *, struct device_attribute *);
void device_remove_file(struct device *, struct device_attribute *);
```

It also defines this helper for defining device attributes:
它为了定义设备的属性也定义了辅助的宏：

```
#define DEVICE_ATTR(_name, _mode, _show, _store)      \
struct device_attribute dev_attr_##_name = {          \
        .attr = {.name  =       __stringify(_name) , .mode   = _mode },     \
        .show   = _show,                                \
        .store  = _store,                               \
};
```

For example, declaring
例如：声明

```
static DEVICE_ATTR(foo, S_IWUSR | S_IRUGO, show_foo, store_foo);
```

is equivalent to doing:
等同于这样的代码

```
static struct device_attribute dev_attr_foo = {
        .attr           = {
                .name = "foo",
                .mode = S_IWUSR | S_IRUGO,
        },
        .show = show_foo,
        .store = store_foo,
};
```

Subsystem-Specific Callbacks
子系统特有的调用
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

When a subsystem defines a new attribute type, it must implement a
当一个子系统定义一个新属性类型时，
set of sysfs operations for forwarding read and write calls to the
一系列的sysfs操作必须被执行，以帮助读写函数实现
show and store methods of the attribute owners.
属性所有者的显示和储存的方法。

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *);
};
```

[Subsystems should have already defined a struct kobj_type as a
[子系统应已经定义了一个kobj_type 结构体作为
descriptor for this type, which is where the sysfs_ops pointer is
这个类型的描述符，存储 sysfs_ops 的指针。
stored. See the kobject documentation for more information.]

更多的信息参见 kobject 的文档]

When a file is read or written, sysfs calls the appropriate method
当一个文件被读写时， sysfs 会为这个类型调用适当的方法。
for the type. The method then translates the generic struct kobject
这个方法会将一般的 kobject 和 attribute 结构体指针 转换为
and struct attribute pointers to the appropriate pointer types, and
适当的指针类型后
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

calls the associated methods.
调用相关联的函数。

To illustrate:
示例：

```
#define to_dev_attr(_attr) container_of(_attr, struct device_attribute, attr)
#define to_dev(d) container_of(d, struct device, kobj)

static ssize_t
dev_attr_show(struct kobject * kobj, struct attribute * attr, char * buf)
{
        struct device_attribute * dev_attr = to_dev_attr(attr);
        struct device * dev = to_dev(kobj);
        ssize_t ret = 0;

        if (dev_attr->show)
                ret = dev_attr->show(dev, buf);
        return ret;
}
```

Reading/Writing Attribute Data
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
读写属性数据

To read or write attributes, show() or store() methods must be
在声明属性时，show() 或 store() 方法必须被指明，以实现属性的读或写。
specified when declaring the attribute. The method types should be as
这些方法的类型应该和以下的设备属性
simple as those defined for device attributes:
的定义一样简单。

```
        ssize_t (*show)(struct device * dev, char * buf);
        ssize_t (*store)(struct device * dev, const char * buf);
```

IOW, they should take only an object and a buffer as parameters.
也就是说，他们应该只以一个处理对象和一个缓冲指针作为参数。

sysfs allocates a buffer of size (PAGE_SIZE) and passes it to the
sysfs 会分配一个缓冲区的大小（PAGE_SIZE）并传递给这个方法。
method. Sysfs will call the method exactly once for each read or
Sysfs 将会为每次读写操作调用一次这个方法。
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

write. This forces the following behavior on the method

这导致了这些方法的执行会出现以下的行为：
implementations:

- On read(2), the show() method should fill the entire buffer.

-在读方面，show() 方法应该填充整个缓冲区。
Recall that an attribute should only be exporting one value, or an

回想起属性应只导出了一个属性值或是一个同类型的属性值的数组，
array of similar values, so this shouldn't be that expensive.

所以这个代价将不会不太高。

This allows userspace to do partial reads and seeks arbitrarily over

这使得用户空间可以局部地读和任意的搜索整个文件。
the entire file at will.

- On write(2), sysfs expects the entire buffer to be passed during the

-在些方面，sysfs 希望在第一次写操作时得到整个缓冲区。
first write. Sysfs then passes the entire buffer to the store()

之后 Sysfs 传递整个缓冲区给 store()方法。
method.

When writing sysfs files, userspace processes should first read the

当要写 sysfs 文件时，用户空间进程应该首先读整个文件，
entire file, modify the values it wishes to change, then write the

修该想要改变的值，然后回写整个缓冲区。
entire buffer back.

Attribute method implementations should operate on an identical

在读写属性值时，属性方法的执行应操作相同的缓冲区。
buffer when reading and writing values.

Other notes:

注记：

- The buffer will always be PAGE_SIZE bytes in length. On i386, this
  is 4096.

- 缓冲区应总是 PAGE_SIZE 大小。对于i386，这个值为4096。

- show() methods should return the number of bytes printed into the
  buffer. This is the return value of snprintf().

- show() 方法应该返回写入缓冲区的字节数，也就是 snprintf()的返回值。

- show() should always use snprintf().

- show() 应始终使用 snprintf()。

- store() should return the number of bytes used from the buffer. This
  can be done using strlen().
- store() 应返回缓冲区的已用字节数，可使用 strlen()。

- show() or store() can always return errors. If a bad value comes
  through, be sure to return an error.
- show() 或 store() 可以返回错误值。当得到一个非法值，必须返回一个错误值。

- The object passed to the methods will be pinned in memory via sysfs
  referencing counting its embedded object. However, the physical
  entity (e.g. device) the object represents may not be present. Be
  sure to have a way to check this, if necessary.
- 一个传递给方法的对象将会通过 sysfs 调用对象内嵌的引用计数固定在内存中。尽管如此，对象代表的物理实体（如设备）可能已不存在。如有必要，应该实现一个检测机制。

A very simple (and naive) implementation of a device attribute is:
一个简单的（未经实验证实的）设备属性例程如下：

```
static ssize_t show_name(struct device *dev, struct device_attribute *attr, char *buf)
{
        return snprintf(buf, PAGE_SIZE, "%s\n", dev->name);
}

static ssize_t store_name(struct device * dev, const char * buf)
{
        sscanf(buf, "%20s", dev->name);
        return strnlen(buf, PAGE_SIZE);
}

static DEVICE_ATTR(name, S_IRUGO, show_name, store_name);
```

(Note that the real implementation doesn't allow userspace to set the
name for a device.)
（注意：真实的程序不允许用户空间设置设备名。）

Top Level Directory Layout
~~~~~~~~~~~~~~~~~~~~~~~~~~
顶层目录

The sysfs directory arrangement exposes the relationship of kernel
sysfs 目录的安排显示了内核数据结构之间的关系。
data structures.

The top level sysfs directory looks like:
~~~~~~~~~~~~~~~~~~~~~~~~~~

顶层 sysfs 目录如下：

block/
bus/
class/
devices/
firmware/
net/
fs/

devices/ contains a filesystem representation of the device tree. It maps

devices/ 包含了一个设备树的文件系统表示。 他直接
directly to the internal kernel device tree, which is a hierarchy of
以内核设备树的形式反映了设备的层次结构。
struct device.

bus/ contains flat directory layout of the various bus types in the
bus/ 包含了各种内核总线类型的固定目录布局。
kernel. Each bus's directory contains two subdirectories:

每个总线目录包含两个子目录:

devices/
drivers/

devices/ contains symlinks for each device discovered in the system
devices/ 包含了每个系统中出现的设备 指向 设备目录/dev 的动态链接。
that point to the device's directory under root/.

drivers/ contains a directory for each device driver that is loaded
drivers/ 包含了一个每个已为特定总线上的设备而挂载的驱动程序的目录
for devices on that particular bus (this assumes that drivers do not
（这里假定驱动没有多个总线类型）。
span multiple bus types).

fs/ contains a directory for some filesystems. Currently each

fs/ 包含了一个为文件系统设立的目录。 现在每个想要导出属性的
filesystem wanting to export attributes must create its own hierarchy
文件系统必须在 fs/ 下创建自己的层次结构
below fs/ (see ./fuse.txt for an example).
（可参见./fuse.txt 作为参考）。

More information can driver-model specific features can be found in
更多有关driver-model 的特性信息可以在

Documentation/driver-model/.

Documentation/driver-model/中找到。

TODO: Finish this section.

Current Interfaces

~~~~~~~~~~~~~~~~~~

当前接口

The following interface layers currently exist in sysfs:

以下的接口层普遍出现在sysfs中：

- devices (include/linux/device.h)

- 设备

Structure:

结构体：

```
struct device_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device * dev, char * buf);
        ssize_t (*store)(struct device * dev, const char * buf);
};
```

Declaring:

声明：

```
DEVICE_ATTR(_name, _str, _mode, _show, _store);
```

Creation/Removal:

增/删属性：

```
int device_create_file(struct device *device, struct device_attribute * attr);
void device_remove_file(struct device * dev, struct device_attribute * attr);
```

- bus drivers (include/linux/device.h)

- 总线驱动程序

Structure:

结构体：

```
struct bus_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct bus_type *, char * buf);
        ssize_t (*store)(struct bus_type *, const char * buf);
};
```

Declaring:
~~~~~~~~~~~~~~~~~~

声明：

BUS_ATTR(_name, _mode, _show, _store)

Creation/Removal:

增/删属性：

int bus_create_file(struct bus_type *, struct bus_attribute *);

void bus_remove_file(struct bus_type *, struct bus_attribute *);

- device drivers (include/linux/device.h)

- 设备驱动程序

Structure:

结构体：

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *, char * buf);
        ssize_t (*store)(struct device_driver *, const char * buf);
};
```

Declaring:

声明：

DRIVER_ATTR(_name, _mode, _show, _store)

Creation/Removal:

增/删属性

int driver_create_file(struct device_driver *, struct driver_attribute *);

void driver_remove_file(struct device_driver *, struct driver_attribute *);