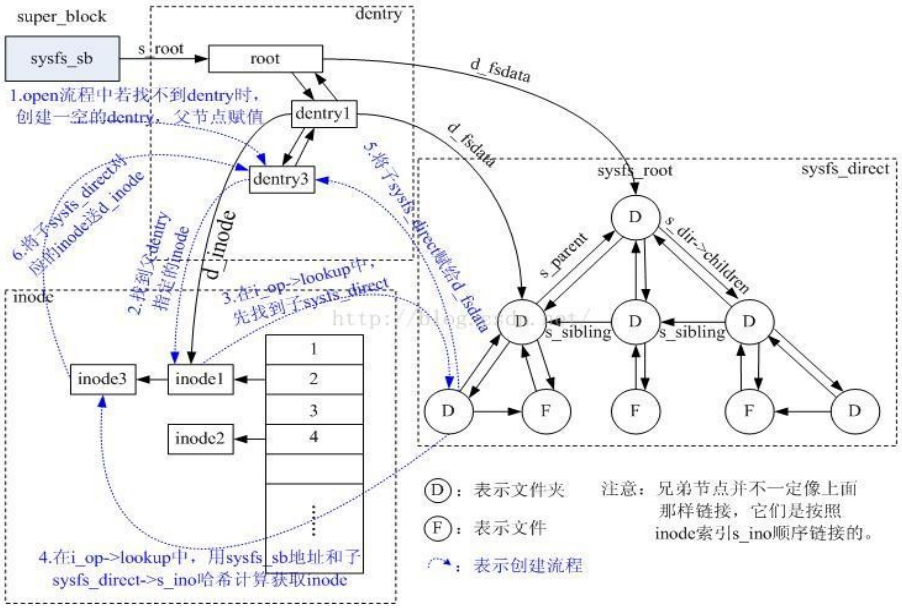
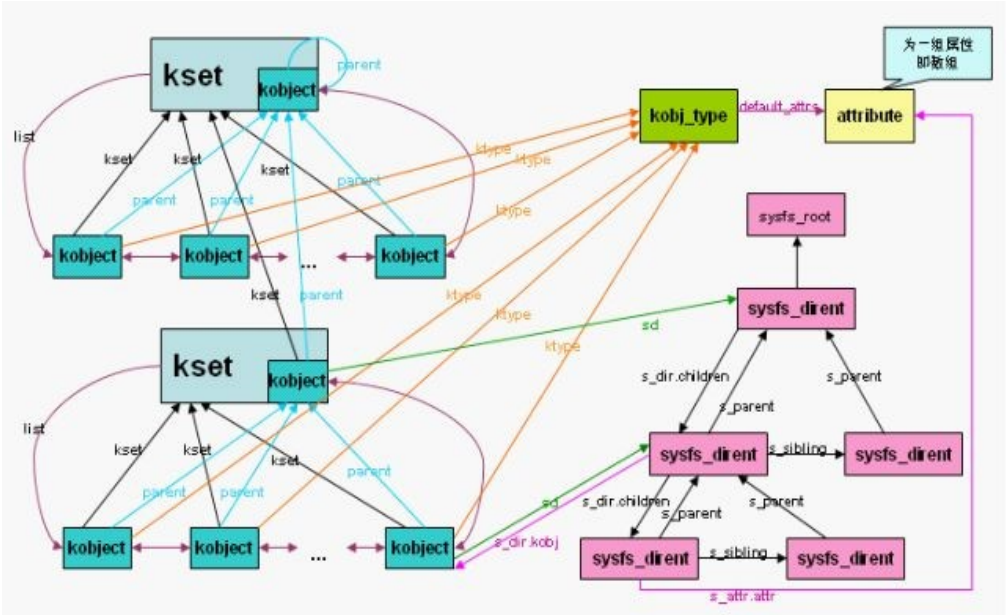
struct file_system_type 只是一个文件系统类型，当该文件系统被挂载时（实例化时），就会生成一个 struct supper_block 对象。以类和对象的思想来理解文件系统的话，file_system_type 就是类，supper_block 就是这个类的一个对象。supper_block 代表一个具体的被挂载的文件系统，file_system_type 代表一个类文件系统。所以 file_system_type 下有一个成员变量 fs_suppers，即每一次挂载 file_system_type 时生成的 supper_block 都链接在 fs_suppers 下面。
file_system_type 主要就是完成两件事，一个是告诉系统自己叫什么，一个是告诉系统怎样解析自己特有的数据结构完成 mount 操作。

supper_block 下最重要的是 root_dentry 和 root_inode。
总结：file_system_type->supper_block->root_inode+root_dentry
有这么一句话好理解类的概念：file_system_type 相关的数据（的**解释权**）由 file_system_type->mount() 函数**解释和处理**，即谁的数据谁解释、处理。
sysfs_dirent、inode、dentry 三者关系：





从上面图中，抓住三棵大树:
1，dentry 树(外加 inode 散列)
2，sysfs_dirent 树
3，kobject 树
dentry 的中文名称是目录项，是 Linux 文件系统中某个索引节点(inode)的链接。这个索引节点可以是文件，也可以是目录。
在内存中, 每个文件都有一个 dentry(目录项)和 inode(索引节点)结构，dentry 记录着文件名，上级目录等信息，正是它形成了我们所看到的树状结构；而有关该文件的组织和管理的信息主要存放 inode 里面，它记录着文件在存储介质上的位置与分布。

sysfs(kernfs)文件系统路径查找的关键：

1.vfs::struct dentry *d_alloc(struct dentry * parent, const struct qstr *name)
{
    struct dentry *dentry = __d_alloc(parent->d_sb, name);
...
    dentry->d_parent = parent;
...
    return dentry;
}
2.sysfs::inode->i_ops->lookup():
static struct dentry *kernfs_iop_lookup(struct inode *dir,
                struct dentry *dentry,
                unsigned int flags)
{
    struct dentry *ret;
    struct kernfs_node *parent = dentry->d_parent->d_fsdata;
    struct kernfs_node *kn;
    struct inode *inode;
    const void *ns = NULL;
...
    kn = kernfs_find_ns(parent, dentry->d_name.name, ns);
...
    kernfs_get(kn);
    dentry->d_fsdata = kn;
    /* attach dentry and inode */
    inode = kernfs_get_inode(dir->i_sb, kn);
    if (!inode) {
        ret = ERR_PTR(-ENOMEM);
        goto out_unlock;
    }
...
    return ret;
}

kernfs_node:

当它代表一个目录时，kernfs_node->priv=kobject;

当它代表一个文件时，kernfs_node->priv=attribute。

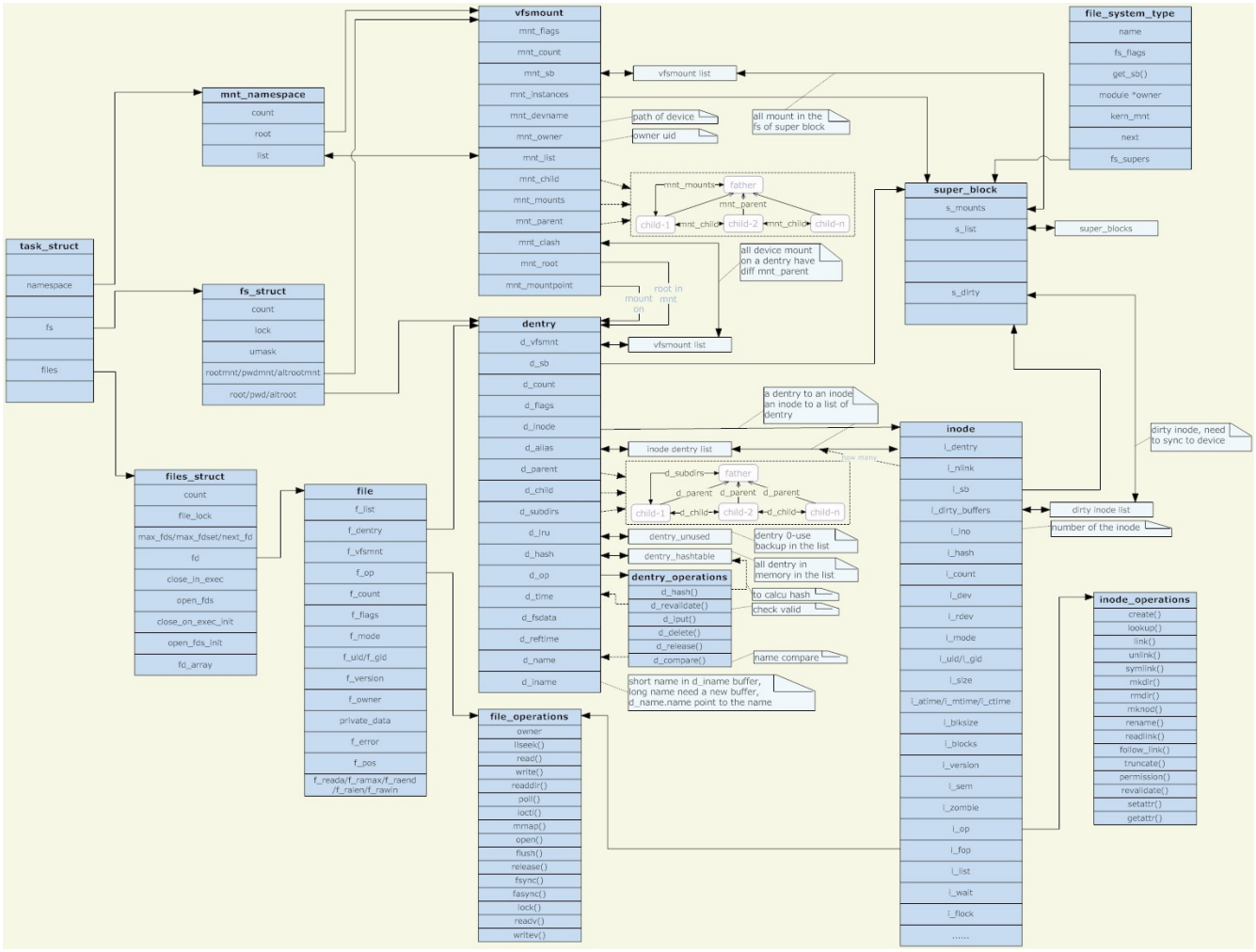
```
*索引节点对象由 inode 结构体表示，定义文件在 linux/fs.h 中
*/
struct inode {
        struct hlist_node       i_hash;           /* 哈希表 */
        struct list_head        i_list;           /* 索引节点链表 */
        struct list_head        i_dentry;         /* 目录项链表 */
        unsigned long           i_ino;            /* 节点号 */
        atomic_t                i_count;          /* 引用记数 */
        umode_t                 i_mode;           /* 访问权限控制 */
        unsigned int            i_nlink;          /* 硬链接数 */
        uid_t                   i_uid;            /* 使用者 id */
        gid_t                   i_gid;            /* 使用者 id 组 */
        kdev_t                  i_rdev;           /* 实设备标识符 */
        loff_t                  i_size;           /* 以字节为单位的文件大小 */
        struct timespec         i_atime;          /* 最后访问时间 */
        struct timespec         i_mtime;          /* 最后修改(modify)时间 */
        struct timespec         i_ctime;          /* 最后改变(change)时间 */
        unsigned int            i_blkbits;        /* 以位为单位的块大小 */
        unsigned long           i_blksize;        /* 以字节为单位的块大小 */
        unsigned long           i_version;        /* 版本号 */
        unsigned long           i_blocks;         /* 文件的块数 */
        unsigned short          i_bytes;          /* 使用的字节数 */
        spinlock_t              i_lock;           /* 自旋锁 */
        struct rw_semaphore     i_alloc_sem;      /* 索引节点信号量 */
        struct inode_operations *i_op;            /* 索引节点操作表 */
        struct file_operations  *i_fop;           /* 默认的索引节点操作 */
        struct super_block      *i_sb;            /* 相关的超级块 */
        struct file_lock        *i_flock;         /* 文件锁链表 */
        struct address_space    *i_mapping;       /* 相关的地址映射 */
        struct address_space    i_data;           /* 设备地址映射 */
        struct dquot            *i_dquot[MAXQUOTAS]; /* 节点的磁盘限额 */
        struct list_head        i_devices;        /* 块设备链表 */
        struct pipe_inode_info  *i_pipe;          /* 管道信息 */
        struct block_device     *i_bdev;          /* 块设备驱动 */
        unsigned long           i_dnotify_mask;   /* 目录通知掩码 */
        struct dnotify_struct   *i_dnotify;       /* 目录通知 */
        unsigned long           i_state;          /* 状态标志 */
        unsigned long           dirtied_when;     /* 首次修改时间 */
        unsigned int            i_flags;          /* 文件系统标志 */
        unsigned char           i_sock;           /* 可能是个套接字吧 */
```

```
        atomic_t            i_writecount;       /* 写者记数 */
        void                *i_security;        /* 安全模块 */
        __u32               i_generation;       /* 索引节点版本号 */
        union {
                void        *generic_ip;        /* 文件特殊信息 */
        } u;
};
/*
*索引节点的操作 inode_operations 定义在 linux/fs.h 中
*/
struct inode_operations {
        int (*create) (struct inode *, struct dentry *,int);
        /*VFS 通过系统调用 create()和 open()来调用该函数，从而为 dentry 对象创建一个新的索引节点。在创建时使用 mode 制定初始模式*/
        struct dentry * (*lookup) (struct inode *, struct dentry *);
        /*该韩式在特定目录中寻找索引节点，该索引节点要对应于 dentry 中给出的文件名*/
        int (*link) (struct dentry *, struct inode *, struct dentry *);
        /*该函数被系统调用 link()电泳，用来创建硬连接。硬链接名称由 dentry 参数指定，连接对象是 dir 目录中 ld_dentry 目录想所代表的文件*/
        int (*unlink) (struct inode *, struct dentry *);
        /*该函数被系统调用 unlink()调用，从目录 dir 中删除由目录项 dentry 制动的索引节点对象*/
        int (*symlink) (struct inode *, struct dentry *, const char *);
        /*该函数被系统电泳 symlik()调用，创建符号连接，该符号连接名称由 symname 指定，连接对象是 dir 目录中的 dentry 目录项*/
        int (*mkdir) (struct inode *, struct dentry *, int);
        /*该函数被 mkdir()调用，创建一个新鲁姆。创建时使用 mode 制定的初始模式*/
        int (*rmdir) (struct inode *, struct dentry *);
        /*该函数被系统调用 rmdir()调用，删除 dir 目录中的 dentry 目录项代表的文件*/
        int (*mknod) (struct inode *, struct dentry *, int, dev_t);
        /*该函数被系统调用 mknod()调用，创建特殊文件(设备文件、命名管道或套接字)。要创建的文件放在 dir 目录中，其目录项问 dentry，关联的设备为 rdev，初始权限由 mode 指定*/
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *);
        /*VFS 调用该函数来移动文件。文件源路径在 old_dir 目录中，源文件由 old_dentry 目录项所指定，目标路径在 new_dir 目录中，目标文件由 new_dentry 指定*/
        int (*readlink) (struct dentry *, char *, int);
        /*该函数被系统调用 readlink()调用，拷贝数据到特定的缓冲 buffer 中。拷贝的数据来自 dentry 指定的符号链接，最大拷贝大小可达到 buflen 字节*/
        int (*follow_link) (struct dentry *, struct nameidata *);
        /*该函数由 VFS 调用，从一个符号连接查找他指向的索引节点，由 dentry 指向的连接被解析*/
        int (*put_link) (struct dentry *, struct nameidata *);
        /*在 follow_link()调用之后，该函数由 vfs 调用进行清楚工作*/
        void (*truncate) (struct inode *);
        /*该函数由 VFS 调用，修改文件的大小，在调用之前，索引节点的 i_size 项必须被设置成预期的大小*/
        int (*permission) (struct inode *, int);
        /*该函数用来检查给低昂的 inode 所代表的文件是否允许特定的访问模式，如果允许特定的访问模式，返回 0，否则返回负值的错误码。多数文件系统都将此区域设置为 null，使用 VFS 提供的通用方法进行检查，
```

这种检查操作仅仅比较索引及诶但对象中的访问模式位是否和 mask 一致，比较复杂的系统，比如支持访问控制链(ACL)的文件系统，需要使用特殊的 permission()方法*/

```
        int (*setattr) (struct dentry *, struct iattr *);
        /*该函数被 notify_change 调用，在修改索引节点之后，通知发生了改变事件*/
        int (*getattr) (struct vfsmount *, struct dentry *, struct kstat *);
        /*在通知索引节点需要从磁盘中更新时，VFS 会调用该函数*/
        int (*setxattr) (struct dentry *, const char *,
                        const void *, size_t, int);
        /*该函数由 VFS 调用，向 dentry 指定的文件设置扩展属性，属性名为 name，值为 value*/
        ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
        /*该函数被 VFS 调用，向 value 中拷贝给定文件的扩展属性 name 对应的数值*/
        ssize_t (*listxattr) (struct dentry *, char *, size_t);
        /*该函数将特定文件所有属性别表拷贝到一个缓冲列表中*/
        int (*removexattr) (struct dentry *, const char *);
        /*该函数从给定文件中删除指定的属性*/
};
```

inode.i_private：存放的是驱动程序相关的私有数据，和 file.private_data 等同；
dentry.d_fsdata：存放的是文件系统相关的私有数据，如 sysfs_dirent.



文件系统的挂载：

mount -t xfs /dev/sdc1 /mnt
这样看 mountpoint 就是/mnt，对吗？不能说不对，但是不够准确。请参考上一篇文章中讲到的关于挂载关系的内容，当多个文件系统挂载到同一个路径名下时是一种什么样的情况？对，后挂载的文件系统会再挂载到前一次挂载的文件系统的根 dentry 上。lock_mount 这个函数的一部分逻辑就保证了在多文件系统挂载同路径的时候，让每个新挂载的文件系统都顺序的挂载（覆盖）上一次挂载实例的根 dentry。
如何做到这一点得从 lock_mount 的实现看起：
lock_mount 的大部分逻辑似乎不是在做锁操作，是的，它的主要逻辑应该是这样的：
[cpp] view plain copy
while (1) {

```
 mnt = lookup_mnt(path)
 if (!mnt){
   确定新的挂载点了, return
 }
 else
 {
   path->mnt = mnt;
   path->dentry = dget(mnt->mnt_root);
 }
}
```

为了说明白它我们不得不需要进一步说明 lookup_mnt 函数的作用，然后我们需要再反回来看这个逻辑，所以在此我们先命名这个逻辑为 FOLLOW_MNT 逻辑。

FOLLOW_MNT 逻辑

我们就以上述 lookup_mnt 的注释中的例子来解释说明一下，lookup_mnt 是如何被使用的。假如我们已经执行了如下操作：

```
 * mount /dev/sda1 /mnt
 * mount /dev/sda2 /mnt
 * mount /dev/sda3 /mnt
```

现在我要执行：

```
mount /dev/sdb1 /mnt
```

那么 follow_mnt 逻辑的执行顺序就是：

1、path 的初始状态为：

path->mnt = 根文件系统

path->dentry = 根文件系统下的/mnt 的 dentry

2、第一次 lookup_mnt(path)返回的 mnt 的状态是：

mnt 为/dev/sda1 这个挂载实例

mnt->mnt_root 为/dev/sda1 这个文件系统的根 dentry

3、lookup_mnt 返回/dev/sda1 的挂载实例，然后执行：

path->mnt = mnt; 将第一个挂载在/mnt 上的/dev/sda1 的挂载实例赋值给 path->mnt

path->dentry = mnt->mnt_root; 将/dev/sda1 挂载成功后的根 dentry 给 path->dentry

4、重返 lookup_mnt(path)，只是这次 path 中的 mnt 和 dentry 变成了/dev/sda1 的。

5、lookup_mnt 发现/dev/sda1 的根 dentry 上也挂载的文件系统/dev/sda2，于是返回了/dev/sda2 的挂载实例。

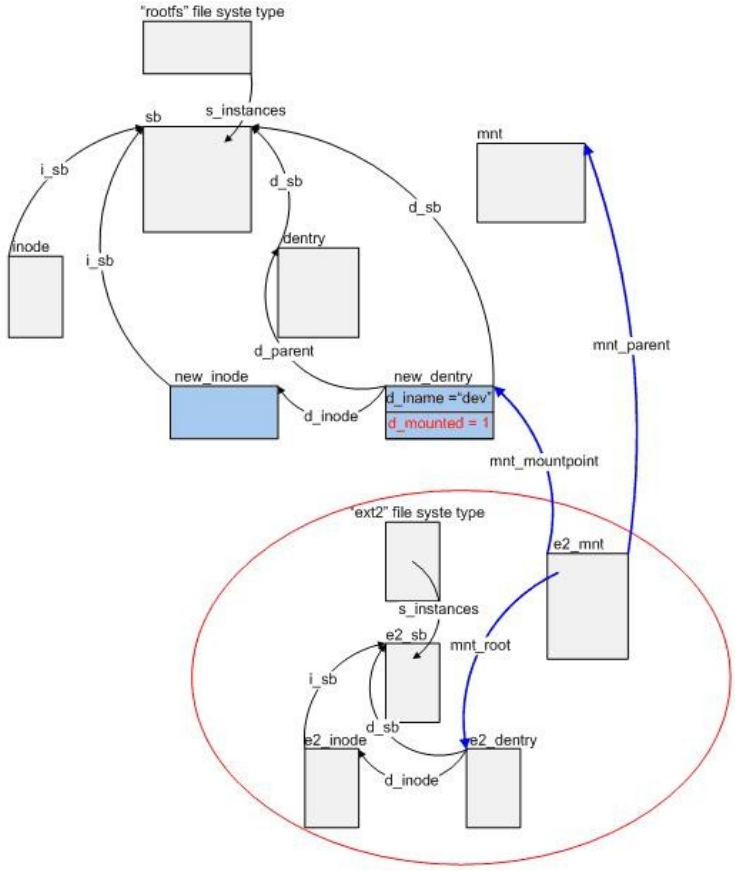
6、又将/dev/sda2 的挂载实例和根 dentry 赋值给 path，重新调用 lookup_mnt。

7、这回 lookup_mnt 返回/dev/sda3 的挂载实例，然后将/dev/sda3 的挂载实例和根 dentry 赋值给 path，再次调用 lookup_mnt。

8、这回 lookup_mnt 发现/dev/sda3 的根 dentry 上没有挂载文件系统，于是返回 NULL。

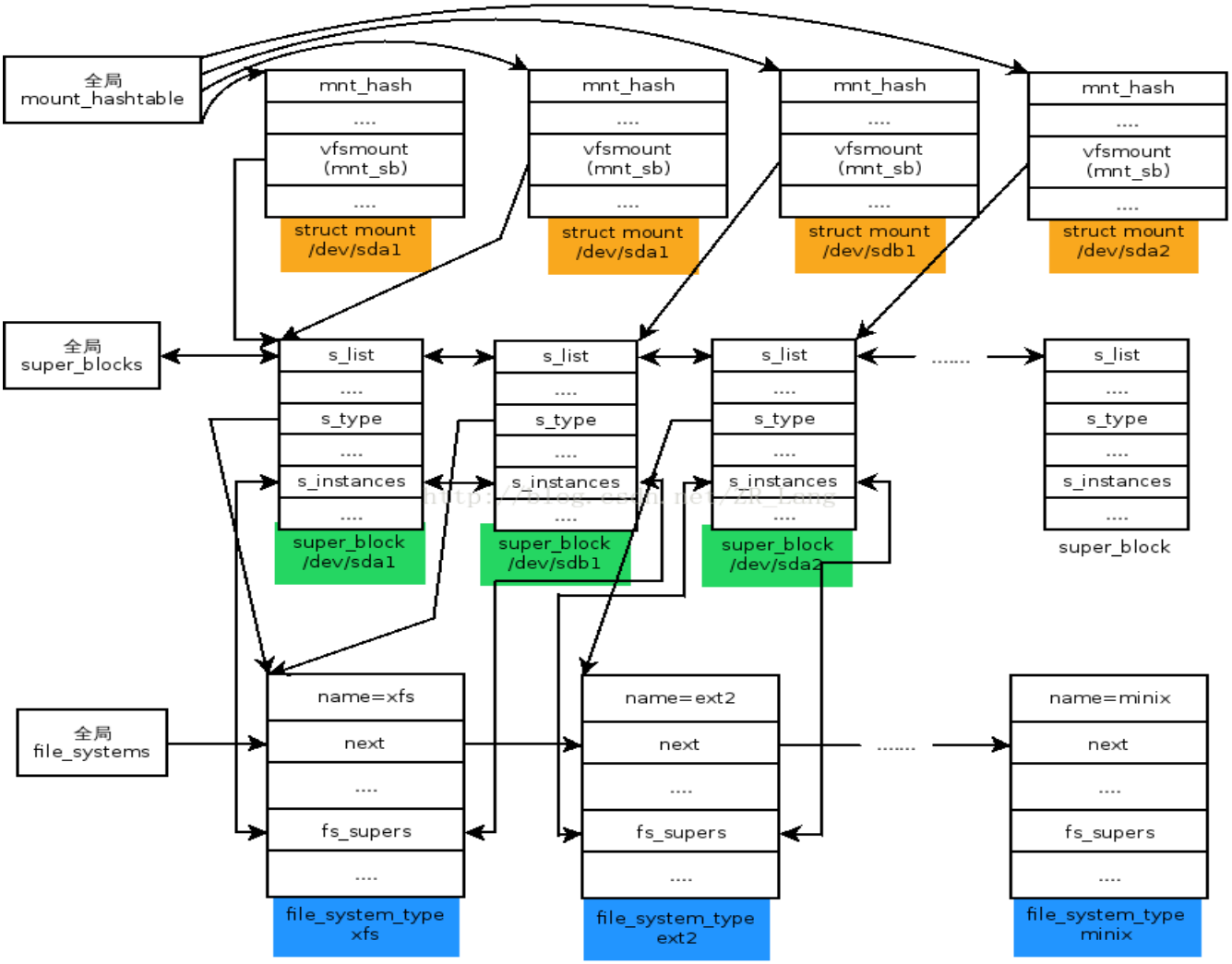
9、lookup_mnt 返回了 NULL，也就是找到了本次要被挂载的挂载点，用/dev/sda3 的根 dentry 构建挂载点结构，并让 lock_mount 返回。新的 /dev/sdb1 将在返回后的操作中被挂载到/dev/sda3 的根 dentry 上。

安装 ext2 类型根文件系统到 "/dev " 目录上



mount 过程分析之六——挂载关系(图解)

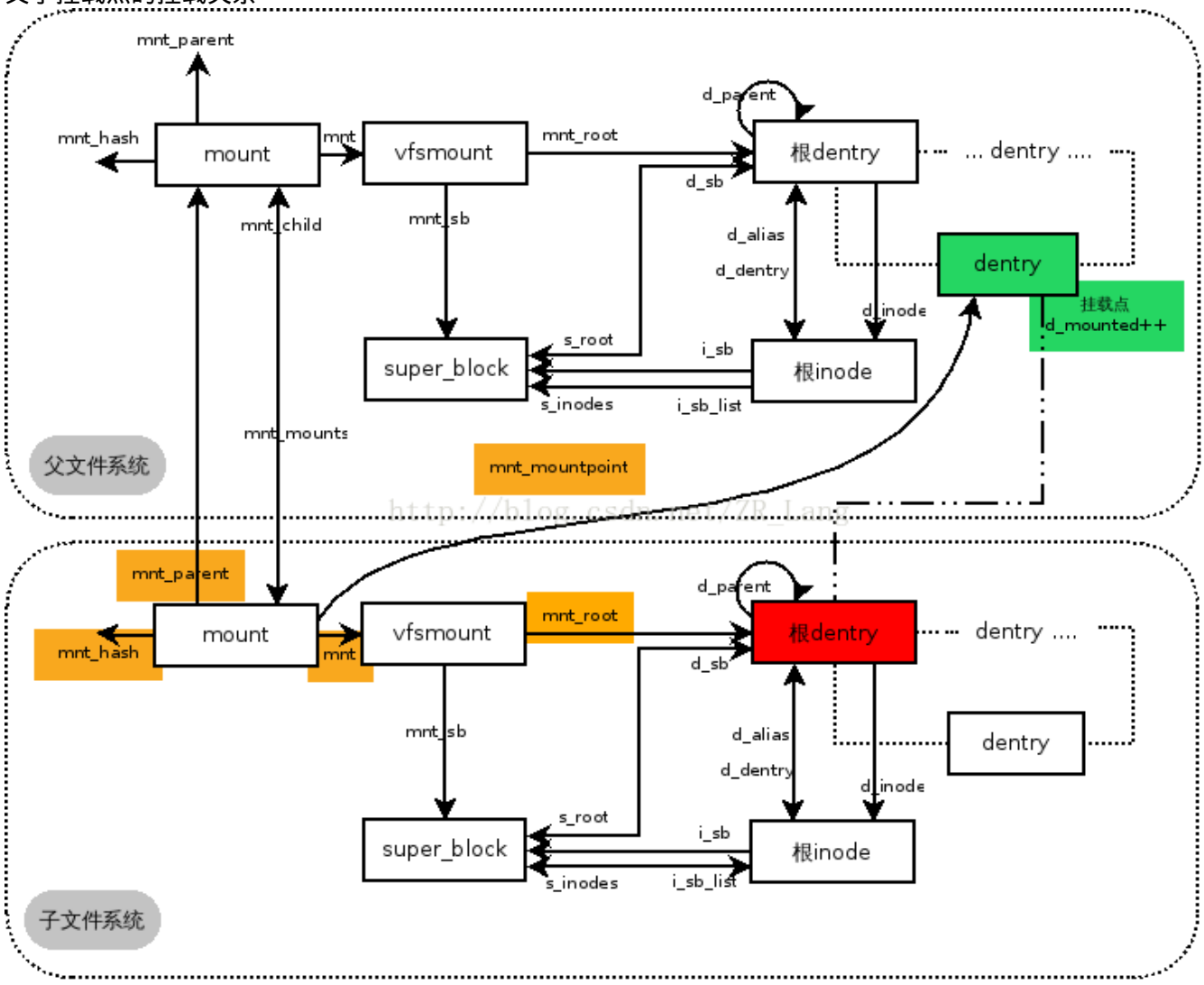
写到这里我们已经从 mount 文件系统调用的入口开始，分析到内核的 mount，通过 VFS 层进入到具体文件系统实现的 mount 函数，最终得到具体文件系统的 super block 信息后构建出一个新的 struct mount 结构，返回给 do_add_mount 继续下面的操作。本来在这节我想从 do_add_mount 的代码开始把 mount 最后一部分——加入全局文件系统树讲完。但是写到一半我发现自己写不下去了，因为单纯的对代码进行注释没有办法解释很多基本的问题，到底挂载实例和文件系统有什么关系？我觉得不把这个说一下没有办法往下写，所以插入这样一节，我们来看一下挂载实例(struct mount/ struct vfsmount)和 super block, dentry, inode, file_system_type 的关系，以及一个文件系统挂载到另一个文件系统下时到底是怎么样的关系。
为了说明的方便，我们下面以这样的场景为例进行描述：
1. 系统中有 xfs, ext2 和 minix 等若干文件系统模块
2. 现有/dev/sda1 和/dev/sdb1 上存在 xfs 文件系统，/dev/sda2 上为 ext2 文件系统，/dev/sdc1 上为 minix 文件系统
3. 将/dev/sda1 挂载到/mnt/a 上，将/dev/sdb1 挂载到/mnt/b 上，将/dev/sdc1 不挂载
4. 在第三步之后，将/dev/sda2 也挂载到/mnt/a 上。再将/dev/sda1 同时挂载到/mnt/x 上
file_system_type + super block + mount 的关系
从 file_system_type 到 super_block 到 mount 实例的角度来看，上述关系大致是这样的：



系统中存在三个文件系统，也就有三种 file_system_type 被注册。sda1 和 sdb1 都是 xfs 文件系统，所以

xfs 的 file_system_type 的 fs_supers 把这两个同为 xfs 文件系统的 super_block 串连在自己下面。sda2 是 ext2 文件系统，所以它挂在 ext2 的 file_system_type 下。sdc1 是 minix 文件系统，在 sdc1 的设备上存在着 super_block 信息，但是我们这里说的 super_block 是指内存中的，由于 sdc1 没有被挂载使用，所以没有它的 super_block 信息被读入内存。
从挂载实例上看，sda1 和 sda2 都挂载到/mnt/a 上，但是从上述关系中很难表述它们的区别，需要借助 mount 和 dentry 的关系来说明，下面再具体说明。sda1 同时挂在了/mnt/a 和/mnt/x 上，所以它有两个挂载实例对应同一个 super_block。sdc1 没有被挂载，所以没有挂载实例和它对应。
mount_hashtable 是一个全局挂载实例的哈希表，系统中除了根挂载点以外所有的挂载实例都记录在它下面，搜索一个 mount 实例时需要借助这个 mount 的父 mount 实例和 dentry 实例来计算的出。比如说/mnt 上挂载着一个文件系统，/mnt/a 和 b 上分别又挂载着文件系统，此时要想检索/mnt/a(或 b)，需要以/mnt 上的挂载实例和/mnt/a(或者 b)的 dentry 结构为依据计算 hash 数值从 mount_hashtable 上得到一个头指针，这个头指针下就是所有父文件系统是在/mnt 上且挂载点是/mnt/a(或 b)的挂载到/mnt/a 或 b 下的 mount 实例。（这里其实有一个不太容易注意到的地方，到底什么情况下同一个父文件系统下的同一个 dentry 下会有多个挂载实例呢？我们不在本文讨论，以后有机会再讨论这个问题。）

上图可以看出 file_system_type, super_block 和 mount 实例之间的关系，但是不能看出来父子文件系统之间的相互关系。下面让我们看一下当一个文件系统挂载到另一个文件系统的子目录下的情况。
父子挂载点的挂载关系



父文件系统代表/mnt 上的文件系统，子文件系统代表/mnt/b 上的文件系统（带颜色的地方为重点要注意的地方）。父子文件系统通过 mnt_parent, mount_child, mnt_mounts 等成员来联系在一起，每个挂载实例的 mnt_sb 都指向这个挂载实例所属文件系统的 super_block。每个挂载实例的 mnt_root 都指向这个文件系统的根 dentry。

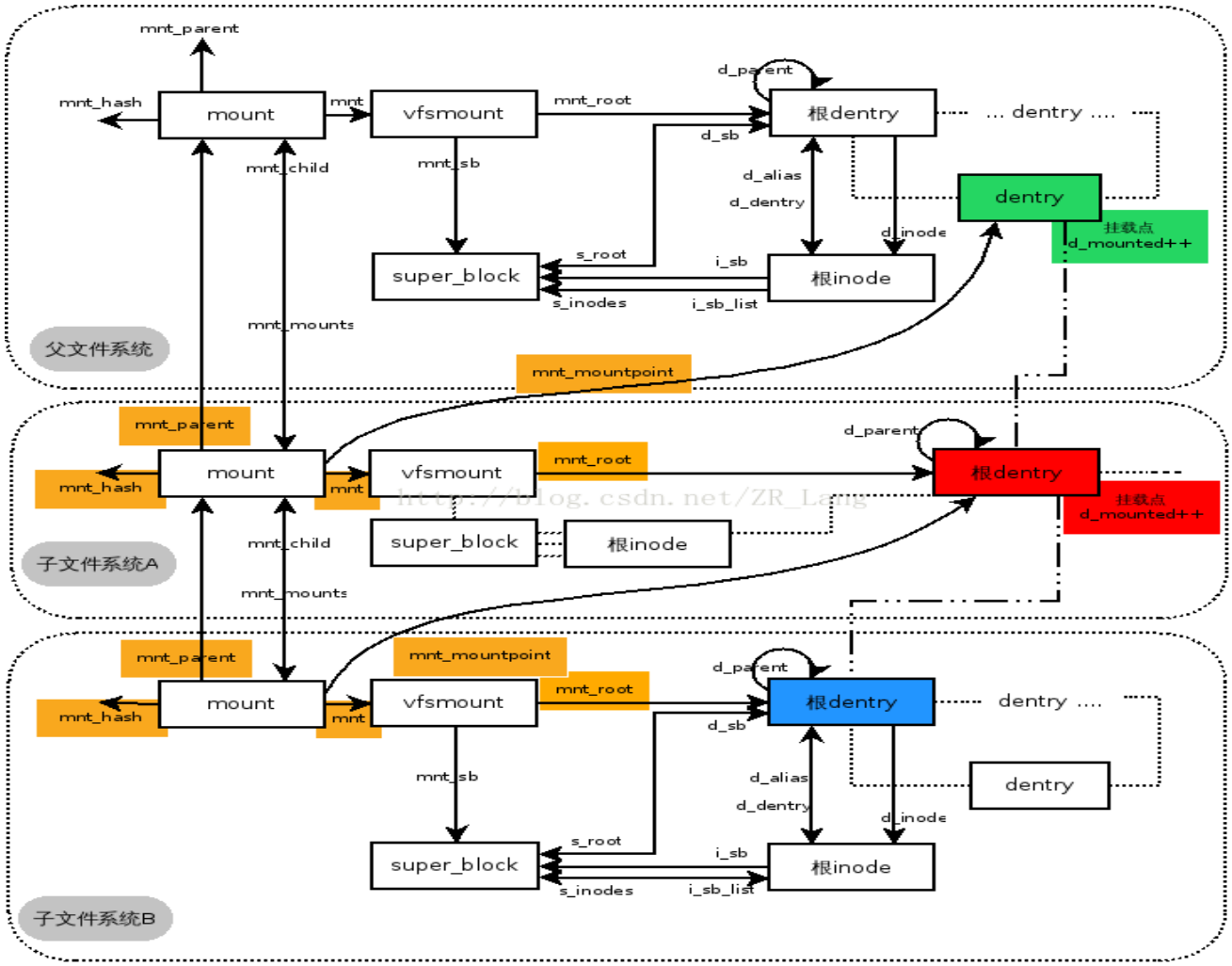
根 dentry 就是一个文件系统的路径的起始，也就是"/"。比如一个路径名/mnt/a/dir/file。在/mnt/b 这个文件系统下看这个文件是/dir/file，这个起始的"/"代表/mnt/a 下挂载的文件系统的根，也就是如上图红色所示的 dentry，它是这一文件系统的起始 dentry。当发现到了一个文件系统的根后，如果想继续探寻完整路径应该根据/mnt/b 的挂载实例向上找到其父文件系统，也就是/mnt 下挂载的文件系统。/dev/sda1 挂载在了/mnt/a 上，这里的/mnt/a 代表/mnt 下文件系统的一个子 dentry，如图绿色部分所示。注意红色和绿色是两个文件系统下的两个不同的 dentry，虽然不是很恰当的说它们从全局来看是一个路径名。那么从/mnt 所在的文件系统看/mnt/a 就是/a。最后再往上就到了 rootfs 文件系统，也就是最上层的根"/"。所以我们之前说过，表示一个文件的路径需要<mount, dentry>二元组来共同确定。

子文件系统的 mnt_mountpoint 就指向了父文件系统的一个 dentry，这个 dentry 也就是子文件系统的真正挂载点。可以说子文件系统在挂载后会新创建一个 dentry，并在此构建这个文件系统下的路径结构。

宗上所述，/mnt/b 上这个新挂载的文件系统创建了一个新的 mount, super_block, 根 inode 和根 dentry。在看懂了一个简单的父子文件系统挂载关系后，我们来看下多个文件系统挂载到同一路径名下时又是什么样子。

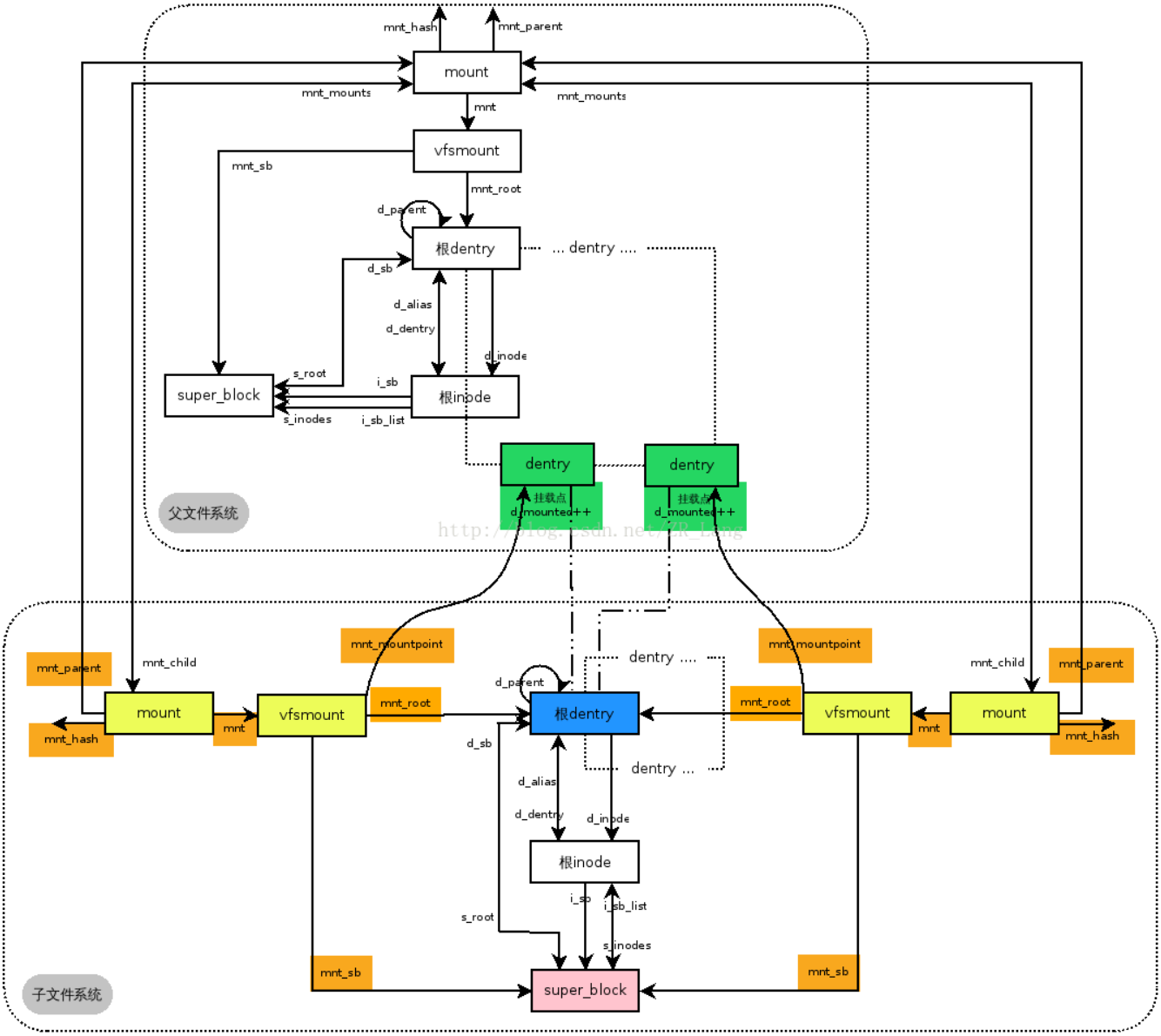
多文件系统单挂载点的挂载关系

就像上面所叙述的/dev/sda1 挂载到了/mnt/a 上，之后/dev/sda2 也挂载到/mnt/a 上的情况，当两个以上的文件系统先后挂载到同一个路径名下时会是怎样一种情况呢？如下图所示：



如上图，子文件系统 A 代表/dev/sda1，子文件系统 B 代表/dev/sda2。父文件系统和子文件系统 A 的挂载在之前一节已经说明过了，当/dev/sda2 在 sda1 之后也挂载到/mnt/a 上时，其关系就像在上一节基础上又添加了子文件系统 B 的关系。实际上子文件系统 A 就是子文件系统 B 的父文件系统，而唯一不同的是子文件系统 B 的 mnt_mountpoint 指向了子文件系统 A 的根 dentry。而新的子文件系统 B 还是有自己的 mount, super_block, 根 dentry 和根 inode。
两个文件系统挂载在同一路径名下会造成之前挂载的文件系统被隐藏，这种实现我们在后续讲解。其实从上面的关系也应该可以想到一些解决的方式。最后再来看一下一个文件系统被挂载到多个不同的路径下的情况。
单文件系统多挂载点的挂载关系

同一个文件系统被挂载到不同的路经下，就像上面例子中/dev/sda1 被挂载到/mnt/a 和/mnt/x 两个位置一样，如下图所示：



一个文件系统对应一个 super_block，所以同一个文件系统当然只有一个 super_block。但是因为挂载了两次，所有每一次挂载对应一个挂载实例 struct mount，也就是有两个 mount 实例。此外同一个文件系统只有一个根，也就是两个挂载实例公用一个根 dentry。但是因为挂载在两个不同的路经下，所以每个挂载实例的 mnt_mountpoint 指向不同的 dentry。由于/mnt/a 和/mnt/x 都属于同一文件系统的下的两个子目录，所以两个子 mount 才指向同一个父 mount（这个不是必须的）。

如果我们像讨论的再变态一点，在上述/mnt/a 和/mnt/x 的基础上在将两个不同的文件系统分别挂载到/mnt/a/dir 和/mnt/x/dir 上，想象一下这将是怎样一种情况和关系。这个我就不画出来了，有兴趣的可以思考一下。

本文讲到这里就先结束了，由于是个后文做一下必要知识的铺垫，在 mount 章节最后分析将 mount 加入全局文件系统树，以及后面的路径名解析中都有一些铺垫的作用。所以只是简单的罗列了一下不同情况下挂载关系的情况。上述的分析也只是代表我自己分析的观点，可能有些地方并不十分准确，还望各路大神纠正。