linux使用RAM\_DISK根文件系统基本过程

——何良斌

以下只是讲述了内存文件系统从uboot到内核、最终挂载到根目录的一个基本流程，至于每个环节的细节，尚未深究，有兴趣的可以再深入学习一下。

1. **内存文件系统**
2. 分类

ramdisk、ramfs、tmpfs

1. ramdik

先格式化（格式化成相应的文件系统）再挂载，大小固定，之后不能随便更改。基于虚拟在内存中的其他文件系统，例如：

mke2fs /dev/ram0; mount /dev/ram0 /mnt/mtd

1. ramfs

内存文件系统，处于虚拟文件系统（VFS）层，无需格式化，可以创建多个，只要内存足够，在创建时可以执行最大能使用的内存大小，针对物理内存。例如：

mount –t ramfs /dev/mem /mnt/mtd or

mount –t ramfs none /mnt/mtd –o maxsize=2000(kbyte为单位)

1. tmpfs

虚拟内存文件系统，可以使用物理内存，也可以使用交换分区

1. **基本原理**



Flash的mtdblock1存放根文件系统的相关文件，包括uImage、cramfs.initrd.img等。uboot启动后会将uImage和cramfs.initrd.img加载到内存的相应地址，然后uboot引导内核启动。内核启动根据cramfs.initrd.img存放地址将其挂载到根目录下，因此我们进入shell所看见的根目录就是对应的内存文件系统内容。挂载成功之后，就可以将mtdblock1的其他目录文件挂载到内存文件系统下，进行文件方式访问、操作等。

1. **Uboot阶段加载initrd.img**

1、加载uImage和initrd.img文件到系统内存

在加载uImage的地方添加加载内存根文件系统cramfs.initrd.img文件，两个文件的地址不能重叠，另外不要存放在内核启动的地址0x80008000。这里分别存放在0x82000000和0x81000000的地方，代码如下：

int do\_fsload(cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

…

size = squashfs\_fload((char \*) offset, part, filename); ^M

#ifdef CONFIG\_HI3531\_V100

size = squashfs\_fload((char \*) CONFIG\_INITRDRAMFS\_LOAD\_ADDR, part, "/boot/cramfs.initrd.img");

#endif

…

}

2、修改bootargs为内存根文件系统

#define BOOTARGS\_3531\_V100 "mem=245M, console=ttyAMA0,115200 rootfstype=squashfs " "root=/dev/ram"

3、启动内核bootm

启动内核使用如下命令：bootm 0x82000000 0x81000000。跟随两个参数，第一个为uImage加载的起始地址，第二个参数为cramfs.initrd.img的起始地址，两个参数顺序不能反，这个bootm命令实现决定，具体分析见后面。

（1）启动内核do\_bootm源码解析

int do\_bootm (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

….

if (bootm\_start(cmdtp, flag, argc, argv)) //获取uImage和initrd.img的镜像文件头

return 1;

…

ret = bootm\_load\_os(images.os, &load\_end, 1); //解压uImage，此处不讲解

…

boot\_fn = boot\_os[images.os.os];= do\_bootm\_linux //调用内核最终启动函数，见后面讲解

….

boot\_fn(0, argc, argv, &images); //此处成功的情况下不会执行reset

do\_reset (cmdtp, flag, argc, argv);

}

（2）获取镜像头bootm\_start源码解析

static int bootm\_start(cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

memset ((void \*)&images, 0, sizeof (images)); //对images头文件进行初始化

…

os\_hdr = boot\_get\_kernel (cmdtp, flag, argc, argv,&images, &images.os.image\_start, &images.os.image\_len); //获取内核镜像头，主要是获取起始地址和内核数据长度

….

switch (genimg\_get\_format (os\_hdr)) { //获取image的相关信息

case IMAGE\_FORMAT\_LEGACY:

images.os.type = image\_get\_type (os\_hdr);

images.os.comp = image\_get\_comp (os\_hdr);

images.os.os = image\_get\_os (os\_hdr);

images.os.end = image\_get\_image\_end (os\_hdr);

images.os.load = image\_get\_load (os\_hdr);

break;

…

ret = boot\_get\_ramdisk (argc, argv, &images, IH\_INITRD\_ARCH, &images.rd\_start, &images.rd\_end); //获取initrd的起始结束地址

….

}

（3）启动内核do\_bootm\_linux

将initrd的相关参数加入内核参数列表传递给内核。

int do\_bootm\_linux(int flag, int argc, char \*argv[], bootm\_headers\_t \*images)

{

…

#if defined (CONFIG\_SETUP\_MEMORY\_TAGS) || \

defined (CONFIG\_CMDLINE\_TAG) || \

defined (CONFIG\_INITRD\_TAG) || \

defined (CONFIG\_SERIAL\_TAG) || \

defined (CONFIG\_REVISION\_TAG)

setup\_start\_tag (bd); //初始化boot参数bd->bi\_boot\_params;

…

#ifdef CONFIG\_INITRD\_TAG //将initrd的相关参数加入参数列表，具体实现见后面

if (images->rd\_start && images->rd\_end)

setup\_initrd\_tag (bd, images->rd\_start, images->rd\_end);

#endif

…

setup\_end\_tag (bd);

#endif

…

theKernel (0, machid, bd->bi\_boot\_params); //进入kernel阶段

}

static void setup\_initrd\_tag (bd\_t \*bd, ulong initrd\_start, ulong initrd\_end)

{

params->hdr.tag = ATAG\_INITRD2; //内核使用initrd2相关函数解析该tag，见内核阶段

params->hdr.size = tag\_size (tag\_initrd);

params->u.initrd.start = initrd\_start; //initrd的起始地址

params->u.initrd.size = initrd\_end - initrd\_start; //initrd的长度

params = tag\_next (params);

}

4、kernel各个镜像地址（hi3536为例）

Images.os的各个信息：

start:0x42000000

end:0x42252000

image\_start:0x42000040

image\_len:0x251fc0,

load:0x40008000

load\_end = load + image\_len

comp:0x0： IH\_COMP\_NONE

type:0x2 ：IH\_TYPE\_KERNEL

os:0x5 ：IH\_OS\_LINUX

images ep:0x40008000

rd\_start:0x41000040

rd\_end:0x4115c040

\*p\_virtaddr++ = 0xe51ff004; /\* ldr pc, [pc, #-4] \*/ 该地址为功能PC对应的指令代码

\*p\_virtaddr++ = jump\_addr; /\* pc jump phy address \*/

1. **Kernel阶段处理initrd.img**

1、获取initrd.img的相关参数

进入内核后，获取uboot传递过来的initrd.img参数的调用流程如下：

start\_kernel() —> setup\_arch() —> setup\_machine\_tags()，该函数会对uboot传递进来的所有tags进行解析：

if (tags->hdr.tag == ATAG\_CORE)

{

if (meminfo.nr\_banks != 0)

squash\_mem\_tags(tags);

save\_atags(tags);

parse\_tags(tags);

}

其中函数parse\_tag\_initrd2就是解析内存文件系统的参数：获取内存文件系统的内存地址和长度，这个可以和前面的uboot对应起来：

static int \_\_init parse\_tag\_initrd2(const struct tag \*tag)

{

//dump\_stack();

phys\_initrd\_start = tag->u.initrd.start;

phys\_initrd\_size = tag->u.initrd.size;

return 0;

}

2、内存文件系统占用的内存处理

调用流程如下：start\_kernel() —> setup\_arch() —> arm\_memblock\_init()，该函数会对上面获取到的内存文件系统占用内存进行处理：

#ifdef CONFIG\_BLK\_DEV\_INITRD

if (phys\_initrd\_size &&!memblock\_is\_region\_memory(phys\_initrd\_start, phys\_initrd\_size)) {

phys\_initrd\_start = phys\_initrd\_size = 0;

}

if (phys\_initrd\_size &&memblock\_is\_region\_reserved(phys\_initrd\_start, phys\_initrd\_size)) {

phys\_initrd\_start = phys\_initrd\_size = 0;

}

if (phys\_initrd\_size) {

memblock\_reserve(phys\_initrd\_start, phys\_initrd\_size);

/\* Now convert initrd to virtual addresses \*/

initrd\_start = \_\_phys\_to\_virt(phys\_initrd\_start);

initrd\_end = initrd\_start + phys\_initrd\_size;

}

#endif

其中initrd\_start和initrd\_end为ramfs占用的虚拟内存起始和结束地址。

3、各种init调用顺序

在讲内存文件系统挂载之前，先讲一下各种initcall。arch/arm/kernel/vmlinux.lds文件规定了各个initcall的存放段：

\_\_tagtable\_begin = .;

\*(.taglist.init)

\_\_tagtable\_end = .;

\_\_pv\_table\_begin = .;

\*(.pv\_table)

\_\_pv\_table\_end = .;

. = ALIGN(16); \_\_setup\_start = .; \*(.init.setup) \_\_setup\_end = .;

\_\_initcall\_start = .; \*(.initcallearly.init) \_\_early\_initcall\_end = .; \*(.initcall0.init) \*(.initcall0s.init) \*(.initcall1.init) \*(.initcall1s.init) \*(.initcall2.init) \*(.initcall2s.init) \*(.initcall3.init) \*(.initcall3s.init) \*(.initcall4.init) \* (.initcall4s.init) \*(.initcall5.init) \*(.initcall5s.init) \*(.initcallrootfs.init) \*(.initcall6.init) \*(.initcall6s.init) \*(. initcall7.init) \*(.initcall7s.init) \_\_initcall\_end = .;

\_\_con\_initcall\_start = .; \*(.con\_initcall.init) \_\_con\_initcall\_end = .;

\_\_security\_initcall\_start = .; \*(.security\_initcall.init) \_\_security\_initcall\_end = .;

. = ALIGN(4); \_\_initramfs\_start = .; \*(.init.ramfs) . = ALIGN(8); \*(.init.ramfs.info)

\_\_init\_begin = \_stext;

\*(.init.data) \*(.cpuinit.data) \*(.meminit.data) \*(.init.rodata) \*(.cpuinit.rodata) \*(.meminit.rodata) . = ALIGN(32); \_\_dtb\_start = .; \*(.dtb.init.rodata) \_\_dtb\_end = .;

}

上面列出了tags、setup、initcall所占的各个字段，同时说明了各个初始化顺序（即优先级），(.initcallrootfs.init)为根文件系统的初始化优先级，比较靠后。

4、内核对内存文件系统初始化

调用流程：start\_kernel() —> rest\_init () ，该函数创建kernel\_init，该线程对上面所有注册的initcall按照顺序进行初始化。

下面对线程kernel\_init相关的函数进行说明：

（1）initcallearly初始化

由函数do\_pre\_smp\_initcalls实现，具体如下：

static void \_\_init do\_pre\_smp\_initcalls(void)

{

initcall\_t \*fn;

for (fn = \_\_initcall\_start; fn < \_\_early\_initcall\_end; fn++)

do\_one\_initcall(\*fn);

}

上述蓝色地址见上面vmlinux.lds

（2）其他initcall初始化

do\_basic\_setup()—>do\_initcalls()（do\_basic\_setup函数里还完成驱动等的初始化），该函数具体实现如下：

static void \_\_init do\_initcalls(void)

{

initcall\_t \*fn;

for (fn = \_\_early\_initcall\_end; fn < \_\_initcall\_end; fn++)

do\_one\_initcall(\*fn);

}

这个函数会调用initcallroofs根文件系统的初始化函数populate\_rootfs()

（3）初始化根文件系统populate\_rootfs()（细节未深入）

该函数先判断initrd\_start值为非0，然后将内存文件系统数据写入文件/initrd.image：

fd = sys\_open((const char \_\_user \_\_force \*) "/initrd.image",O\_WRONLY|O\_CREAT, 0700);

if (fd >= 0) {

sys\_write(fd, (char \*)initrd\_start, initrd\_end - initrd\_start);

sys\_close(fd);

free\_initrd();

}

5、内存文件系统挂载/目录

kernel\_init()—>prepare\_namespace()，该函数完成根目录挂载：

void \_\_init prepare\_namespace(void)

{

….

if (initrd\_load())

goto out;

….

mount\_root();

out:

devtmpfs\_mount("dev");

sys\_mount(".", "/", NULL, MS\_MOVE, NULL);

sys\_chroot((const char \_\_user \_\_force \*)".");

}

上面代码末尾就是挂载文件系统到根目录，initrd\_load()就是加载内存文件系统。

（2）initrd\_load()实现

int \_\_init initrd\_load(void)

{

printk("\*\*\* mount\_initrd:=%d\n", mount\_initrd);

if (mount\_initrd) {

create\_dev("/dev/ram", Root\_RAM0);

if (rd\_load\_image("/initrd.image") && ROOT\_DEV != Root\_RAM0) {

sys\_unlink("/initrd.image");

handle\_initrd();

return 1;

}

}

sys\_unlink("/initrd.image");

return 0;

}

将内存文件系统数据加载到/dev/ram以便后续挂在到根目录下。

# 五、linux使用RAM\_DISK根文件系统使用方法

注下面以hi3531+3532平台为例，进行简单讲解，如有遗漏请帮忙补充

1、uboot修改

（1）bootargs修改

#define BOOTARGS\_3531\_V100 "mem=135M@0x80000000 mem=135M@0xc0000000, console=ttyAMA0,115200 rootfstype=squashfs " \

"root=/dev/ram"

rootfstype以打包时使用的压缩方式为准；使用内存文件建系统，所以root不再是flash的mtdblock分区而是内存设备

（2）fsload修改

加载/boot/uImage之后，还需要将cramfs.initrd.img加载到指定的位置，注意uImage和cramfs加载空间不能重叠，且不要将cramfs加载到uImage启动地址。

if(squashfs\_check(part))

{

size = squashfs\_fload((char \*) offset, part, filename); //加载uImage

printf("### %s loading '%s' to 0x%lx\n", fsname, "/boot/cramfs.initrd.img",\

offset-0x1000000);

size = squashfs\_fload((char \*)(offset - 0x1000000), part, "/boot/cramfs.initrd.img"); //加载cramfs.initrd.img

}

（3）bootm修改

修改bootm —> bootm 0x82000000 0x8100000：当没有参数时bootm只解析uImage；第一个参数为内核加载的地址，第二参数为cramfs加载的地址，顺序不能反。

2、kernel修改

内核配置添加对initrd的支持：

General setup --->

[\*] Initial RAM filesystem and RAM disk (initramfs/initrd) support

() Initramfs source file(s)

[\*] Support initial ramdisks compressed using gzip

[ ] Support initial ramdisks compressed using bzip2

[\*] Support initial ramdisks compressed using LZMA

[ ] Support initial ramdisks compressed using XZ

[ ] Support initial ramdisks compressed using LZO

Device Drivers --->

[\*] Block devices --->

<\*>RAM block device support

(16) Default number of RAM disks

(4096) Default RAM disk size (kbytes)

3、打包环境修改

（1）打包目录修改

新增一个ramfs目录用于生成ramfs.initrd.img，该目录类似原来的romfs但是只保留了根文件系统所需的基本要素（节省内存），而占大内存和flash的Challenge.7z、lib.7z、usr目录等内容仍然保留在romfs目录。

调整后romfs值保留了几个目录，如下所示：

/boot:

cramfs.initrd.img uImage ： 主从共用的内存文件系统、主片kernel

/slave:

u-boot.bin uImage ：如果没有从片，该目录可以节省；从片的uboot/kernel

/usr:

/bin /data /etc /lib /sbin ：用户所需：challenge.7z lib.7z 图片 字库等内容

（2）打包工具修改

$TOOL\_DIR/mkimage -A arm -O linux -T ramdisk -a 0x0 \

-e 0x83000000 -n "initrd in cramfs" -d $TMP\_DIR/$TMP\_FILE\_RAM $TARGET\_DIR/cramfs.initrd.img

cp $TARGET\_DIR/cramfs.initrd.img $SRC\_DIR/boot/

$TOOL\_DIR/mkimage -A arm -O linux -T kernel -C gzip -a A0050000 \

-e 0xA0d70000 -n 3531romfs -d $TMP\_DIR/$TMP\_FILE $TARGET\_DIR/$TARGET\_FILE

在生成romfs-x.cramfs.img前，要先生成cramfs.initrd.img，并拷贝到romfs的boot目录，然后再生成最终额romfs-x.cramfs.img。

以上就是内存文件系统使用的基本过程，实际情况有些偏差，仅供参考。