目录索引

1. 问题背景
2. 功能概要设计
3. 功能详细设计，包括内存的分配、收集和使用

3.1 大块内存分配过程

3.2 小块内存收集过程

3.3 链式内存的使用过程

3.3.1升级文件载入内存

3.3.2在内存中检测升级文件MD5信息的正确性

3.3.3将内存中读取的升级文件进行烧录

1. 问题背景

天津广电项目：IPTV，Linux版本，使用MTK8613芯片，内存256MB。

为了降低生产成本，与客户商定使用“内存阉割”版的方案，即在原有的512MB内存的基础上，降低至256MB内存。但是这样会出现一个问题，在升级过程中，会发现申请大块内存（40MB左右）的时候，出现申请失败的情况，最终导致大概率的升级失败。

项目进度在即，需要在不改变硬件方案的前提下，对问题加以解决。那么最好的解决方案就是，从软件层次对“大内存分配失败”问题加以规避。

1. 功能概要设计

在Linux环境下，申请大块内存的成功率随着系统运行时间的增加而降低，在低物理内存硬件方案的情况下更是如此。规避这个问题大致有两个思路：

1. 采用预分配的方法，即在开机的时候分配一个大的内存池（memPool）。之后如果需要静态内存的话，可以从已分配的内存中加以提取。而在进行升级操作之前，取消其他模块对内存池的绑定，而把内存池的大部分内存用于升级模块，这样就保证了升级过程的稳定。

优点：保证了升级过程的稳定。

缺点：对原有代码修改范围较广，开发周期长。

1. 采用小内存串联的方法，即在申请大内存失败的情况下，进行两次小内存申请，每次申请的大小为之前的1/2。这样可以保证申请的总大小不变，而每次申请的尽可能小。如果第二次分配还是失败，那么就再利用之前的思路，进行“二分”申请。此思路只需要在“内存分配”和“内存使用”的地方进行小范围调整，这样同时也可以降低开发周期。

优点：保证了升级过程的稳定，且开发周期短，容易实现。

为了缩短开发周期，最终选择了第二个方案。

1. 功能详细设计
   1. 大块内存的分配过程

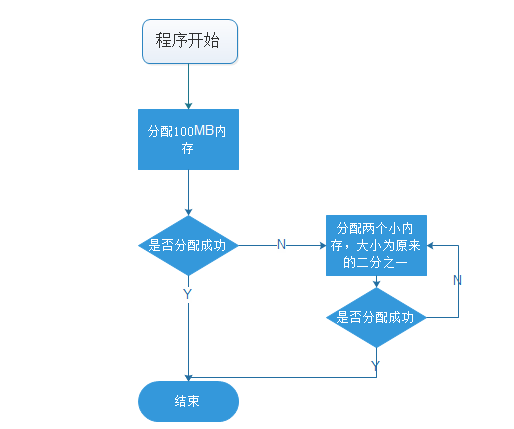


图1

1. 以分配100MB内存为例
2. 如果分配内存成功，则直接结束分配内存过程，进入下一个程序环节
3. 如果分配失败，则重新分配两个小内存，大小是原来的二分之一
4. 分配小内存成功，则直接结束分配内存过程，进入下一个程序环节
5. 若分配小内存失败，则在此内存大小基础上，继续减半，再次进行分配

上述过程类似于一个二叉树建立分支，分配完成后，会形成一个典型的二叉树结构，如图2：

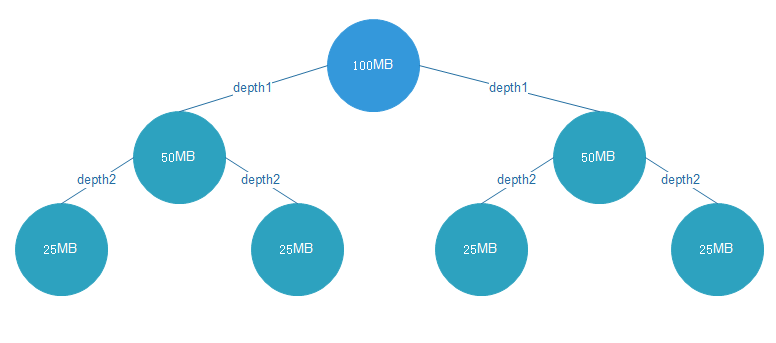


图2

1. 目标内存大小以100MB为例
2. 第一次分配100MB失败以后，会去尝试分配两个50MB，这个时候二叉树的深度为depth1（即深度为1）。这里的深度是为了避免分配循环进入一个死循环而设计的，一般认为最大深度为3，大于3的情况下，可以断定为内存极为不足，以分配失败的错误码返回
3. 若分配50MB也失败了，那么就尝试分配四个25MB，这个时候二叉树的深度为depth2（即深度为2）
4. 假定在25MB的情况下可以分配成功，则程序继续向下执行

3.2小块内存收集过程

这里指的小块内存，即分配结束后的数片内存。

接下来要做的，即是把申请的所有的内存片进行收集和串联，最终形成一个单向的链表结构，以便之后的使用。

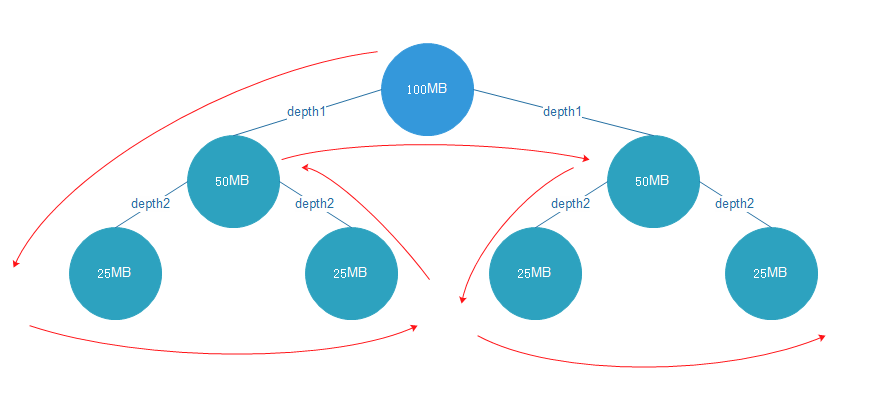
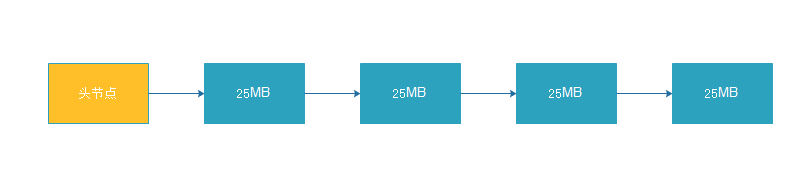


图3

1. 遍历顺序如图的红色箭头部分，即二叉树的“先序遍历”
2. 收集目标：最下层的子孙节点
3. 最终组成一个单向的链表，如图4



3.3 链式内存的使用过程

3.3.1升级文件载入内存

以USB的本地升级为例，讲述升级文件载入内存的过程。

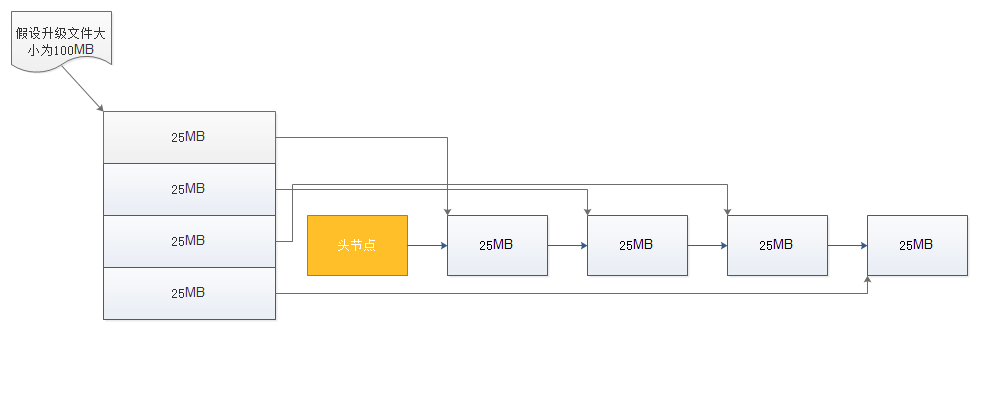


图4

1. 假定升级文件大小为100MB
2. 先读取升级文件的前25MB读取到链表中的第一块内存中
3. 第一块内存用完之后，把升级文件的第二块读取到链表的下一个内存块，然后依次读取，直到读取完毕
4. 此时链式内存中就有了升级文件的全部数据

3.3.2在内存中检测升级文件的MD5信息的正确性

文件加载入内存之后，首先需要校对MD5的正确性。连续内存数据的MD5计算比较简单，主要使用了MD5Init（初始化MD5的结构性数据），MD5Update（计算MD5加密信息）、MD5Final（输出加密结果）。

幸运的是，MD5Update是可以对同一加密目标进行多次更新的。所以我们只需要将“统一的MD5\_CTX和链式内存中的各个片段”作为MD5Update参数，进行循环的更新，然后把最终的MD5\_CTX作为参数，传入到MD5Final，即可得到最终加密结果。

3.3.3将内存中读取的升级文件进行烧录

烧录过程需要注意两个问题：

1. 跨界问题。烧录过程分为以下几个部分：loader、part\_info、fast\_init\_param、logo、kernel、initrd、rootfs，这样就会出现跨界问题。

如下图5，（仅拿loader，kernel、rootfs1和rootfs2为例）。

观看右侧图，loader和kernel所占空间较小，所以不会超越第一个25MB的内存，不会出现跨界的现象。

而rootfs1和rootfs2所占的空间较大，会出现跨越多于一个内存块（每个内存块是25MB）的现象，这样的话，在编程的时候就应该采用拆分烧录的方式，如图6.

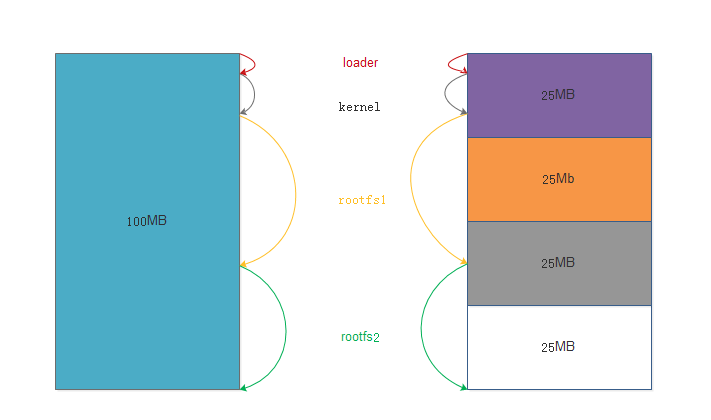


图5

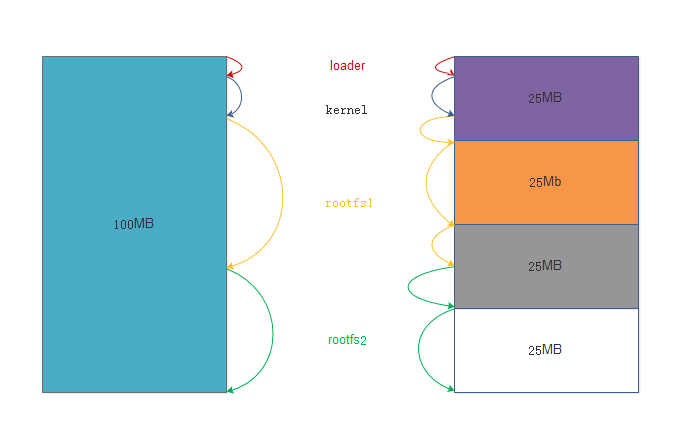


图6

图6的右侧图，把跨界的部分进行了拆分，这样就可以解决跨界烧录的问题了。

1. 跨界问题导致的Erase最小擦除块的问题。这是Flash的特性问题，在进行erase操作的时候，每次erase的大小是一个blocksize（这个值可以从MEMGETINFO中得到），这个值在不同的Flash上有所差异。如图6，在进行rootfs1烧录的时候，会先进行这个Flash区域的erase操作。为了解决跨界问题，已经把rootfs1分成了三次烧录，这样也就需要进行三次不同的erase操作，如图7。这时，第一次erase的大小将非常有可能不是blocksize的整数倍，而导致擦除失败。所以在编程中，一定要进行逻辑控制，以保证每次erase的大小是blocksize的整数倍。

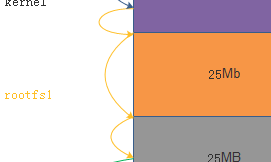


图7