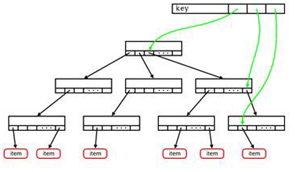
# Radix tree文档

**一、概述**

Linux radix树最广泛的用途是用于内存管理，结构address\_space通过radix树跟踪绑定到地址映射上的核心页，该radix树允许内存管理代码快速查找标识为dirty或writeback的页。Linux radix树的API函数在lib/radix-tree.c中实现。

Linux基数树（radix tree）是将指针与long整数键值相关联的机制，它存储有效率，并且可快速查询，用于指针与整数值的映射（如：IDR机制）、内存管理等。



上图显示了一个有3级结点的radix树，每个数据条目（item）可用3个6位的键值（key）进行索引，键值从左到右分别代表第1~3层结点位置。没有孩子的结点在图中不出现。因此，radix树为稀疏树提供了有效的存储，代替固定尺寸数组提供了键值到指针的快速查找。   
以index=0x5BFB68为例，化为二进制，每6位为一组：10110(22,第一层编号) 111111(63第二次编号) 101101(45第三层编号) 101000(40第四层编号)。

**二、数据结构介绍**

介绍将以linux 4.18.6的内核为例。

**2.1 各个宏定义介绍：**

RADIX\_TREE\_ENTRY\_MASK 3UL

RADIX\_TREE\_INTERNAL\_NODE 1UL

RADIX\_TREE\_EXCEPTIONAL\_ENTRY 2

每个slots[]值的低2bit用于表示当前slot[]的属性，1表示当前是树的树干，0表示是最终的节点，2表示是EXCEPTIONAL，用于tmpfs,shmem保存的特殊节点

=========================================================

RADIX\_TREE\_MAP\_SIZE (1UL << RADIX\_TREE\_MAP\_SHIFT)

表示当前树的node能管理的节点，shift为4或6，具体看配置，因此实际node能管理的节点为16或这个64

=========================================================

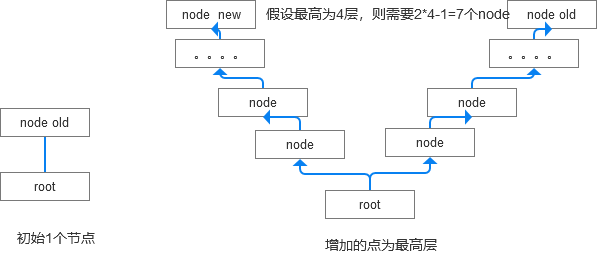
RADIX\_TREE\_MAX\_PATH (DIV\_ROUND\_UP(RADIX\_TREE\_INDEX\_BITS, RADIX\_TREE\_MAP\_SHIFT))

表示一个64位最多能长成的树高，如每个节点可管理64个slot，及shift值为6，则max path= (64 + 6 -1) / 6 = 11

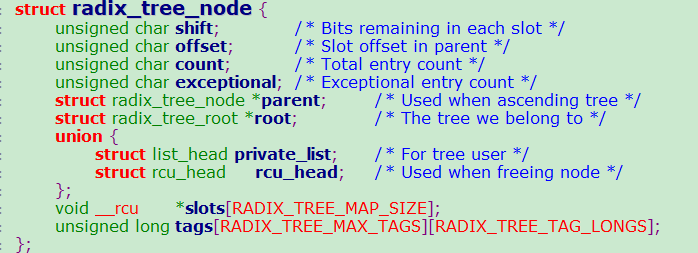
RADIX\_TREE\_PRELOAD\_SIZE (RADIX\_TREE\_MAX\_PATH \* 2 - 1)

为什么preload需要max path \*2-1？？？

如果当前树正好只有1个node，那么如果极限情况下新增了一个node，它的path需要到最高层，那么针对新增的node需要max path个node，但是新的node也需要增加，如果两个点不在树枝上，则也需要增加对应的node，直到连接到老的node，因此需要max path -1个，为什么老的也需要，可以看一下树扩展函数radix\_tree\_extend函数，具体意义如下左图到右图。



**2.2结构体定义介绍：**



unsigned char shift; //当前slot在树的第几层，6bit或者4bit一层

unsigned char offset;//Slot offset in parent

unsigned char count; //当前的slot一共有几个有效的节点

unsigned char exceptional; //用于shmem/tmpfs保存swap entries

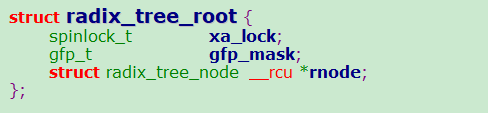
struct radix\_tree\_node \*parent;//父亲node，增加树高的时候有用

struct radix\_tree\_root \*root;//node属于哪棵树

void \*slots[];//当前node管理的子节点，及叶子节点

unsigned long tags[][];//tag最多支持3中类型的tag，二维的第二个参数为每个类型tag需要的long个数，如每个node需要表示64个子节点，系统为32bit，则RADIX\_TREE\_TAG\_LONGS=64 /32=2,如果系统为64bit，则该宏为1

=========================================================



Gfp\_mask表示的是申请node时候的gfp flag标志，gfp的ROOT\_TAG\_SHIFT起至最高位用于标志这个树是什么作用标志。ROOT\_IS\_IDR（4）即gfp\_mask的第2位，如果置位则表示这个树用于id radix

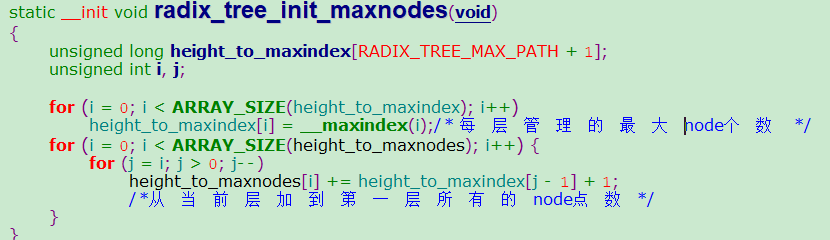
Rnode指向树根node

**2.3函数定义及说明**

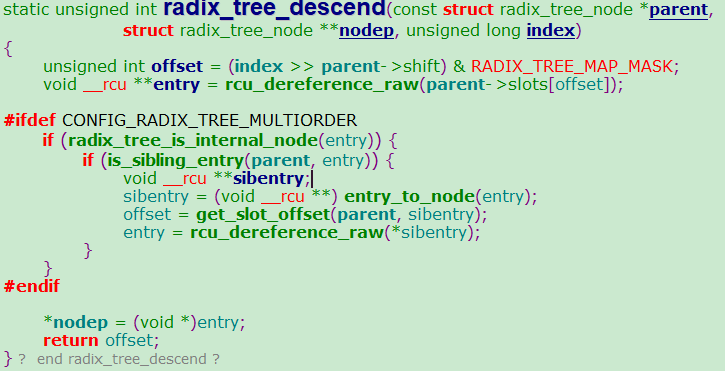
表示树层高需要的node的总数

unsigned long height\_to\_maxnodes[RADIX\_TREE\_MAX\_PATH + 1]

计算代码如下：

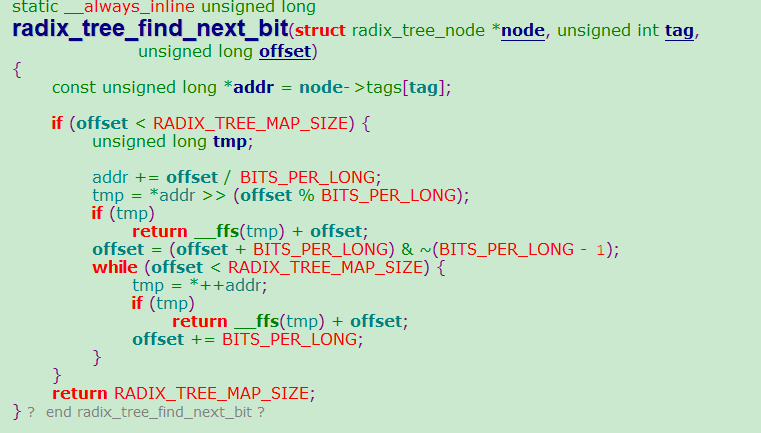
=========================================================

通过index获取对应的叶子节点及指针地址并赋值给nodep,同时返回在当前树干上的offset



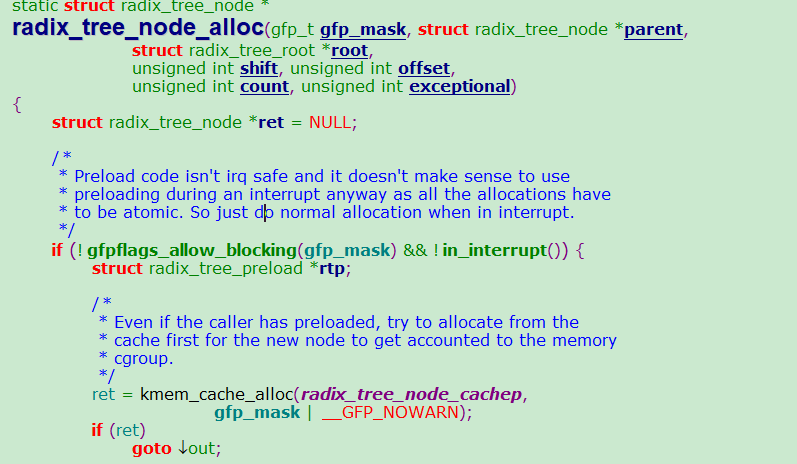
=========================================================

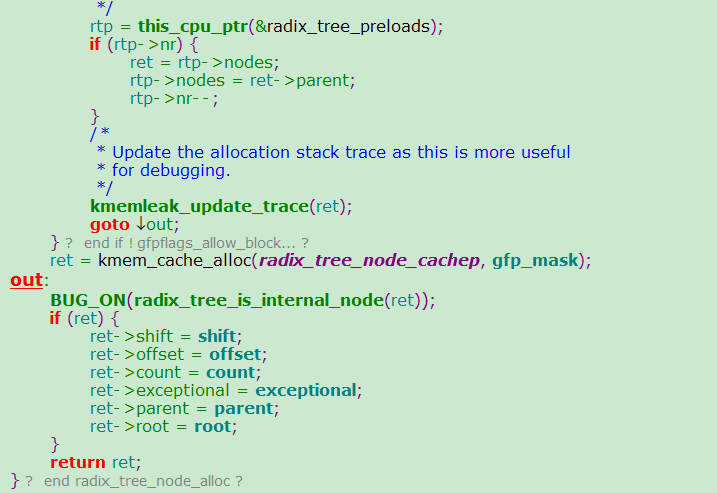
在当前的node的map size范围内找到下一个存在于offset到max size之间的offset。如传入的offset为50，map size为64，那么在50-64之间找tag置位的bit，如找到54，则返回54。



=========================================================

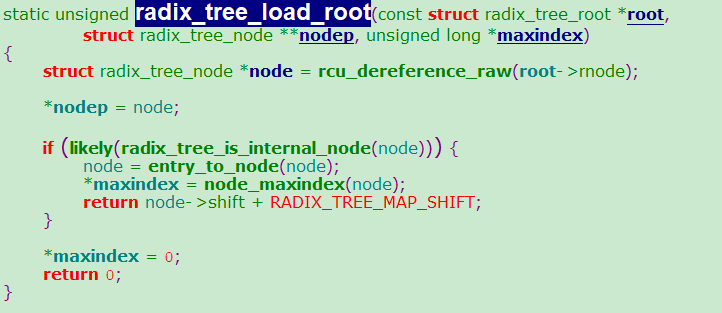
非中断且alloc不允许阻塞的情况下首先尝试从radix\_tree\_node\_cachep高速缓存中获取node，如果失败了则从每个CPU预申请的preload中获取，否则都从radix\_tree\_node\_cachep中获取。获取到之后记录当前node的相关信息。





=========================================================

获取root对应的node，同时返回node所在的树高的shift



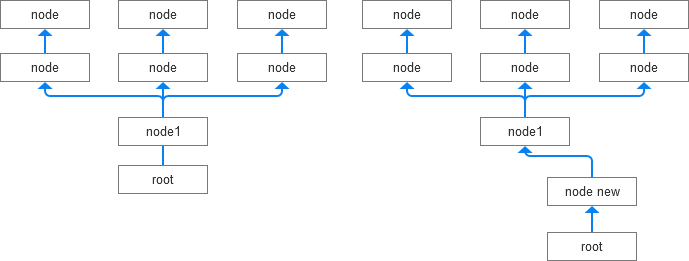
=========================================================

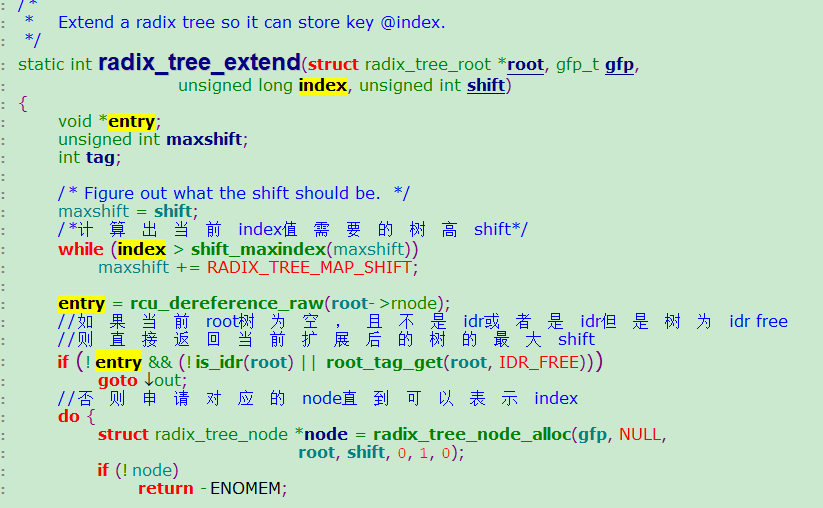
Radix\_tree\_extend函数的作用是扩展树至能表示index范围的树高，

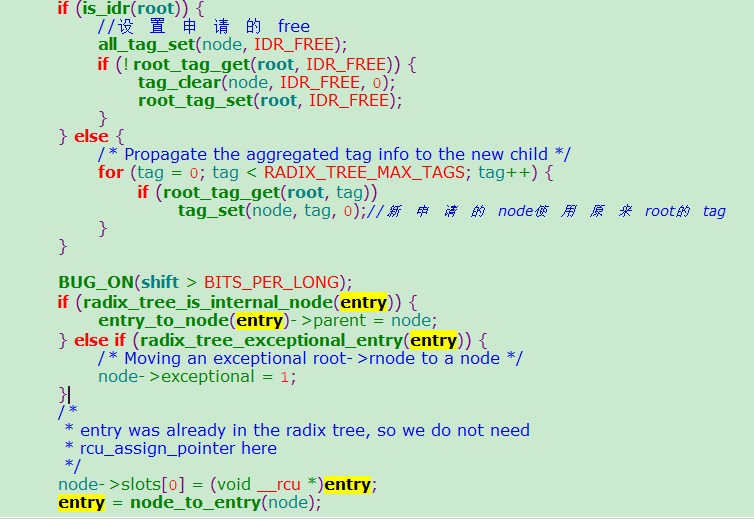
方法是申请一个node，然后node接在树根跟老的树根跟root之间。

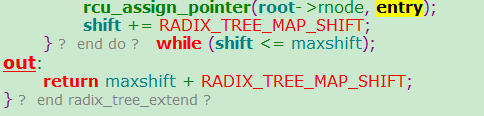
示意图如下，作图为原始树，右侧为扩展树之后的状态。

新的node的slot指向原来的node1，同时对应的tag标志根据root的tag标志填写。





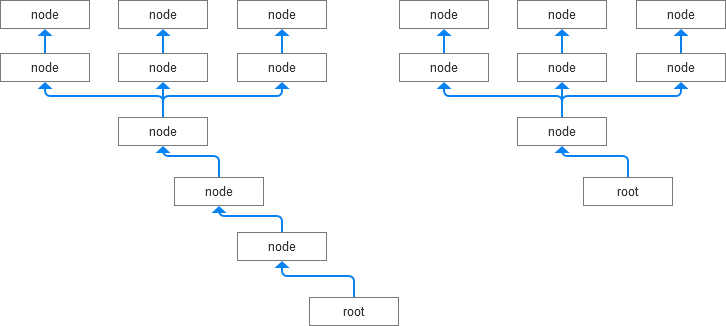


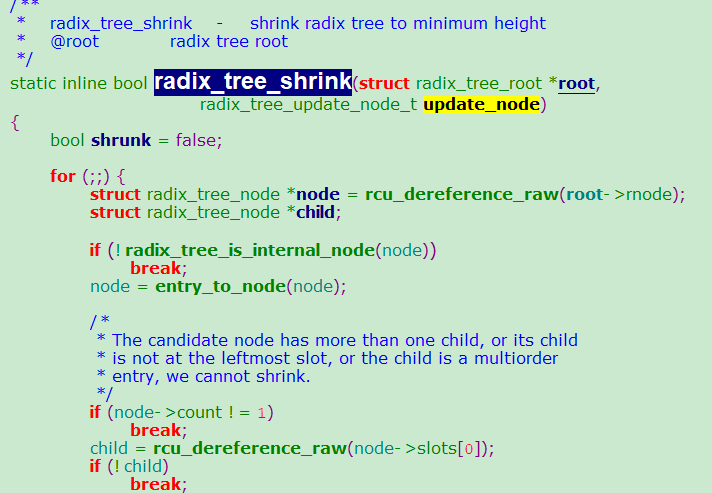


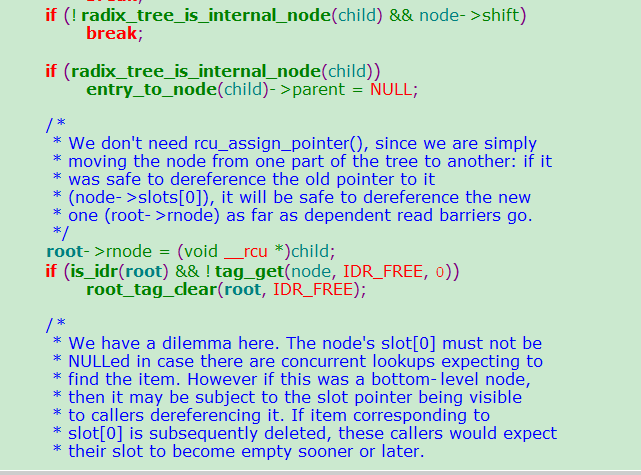
=========================================================

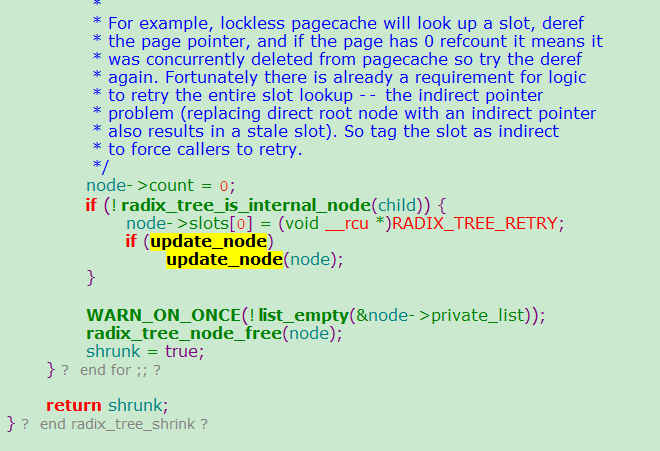
Radix\_tree\_shrink 用于将树高缩短，缩短的条件是当前的树干node的子count只有1才能删除，否则停止缩短。

如下图左侧为原树，发现只有下面的两层count为1，因此将这两个node删除，变成右侧的树形状。









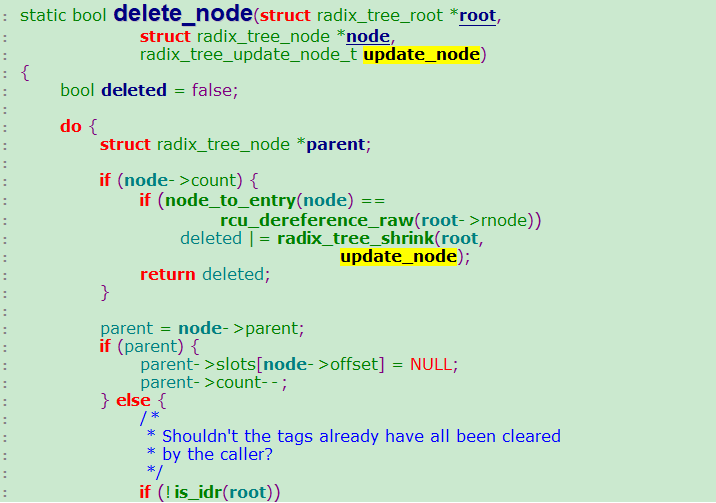
=========================================================

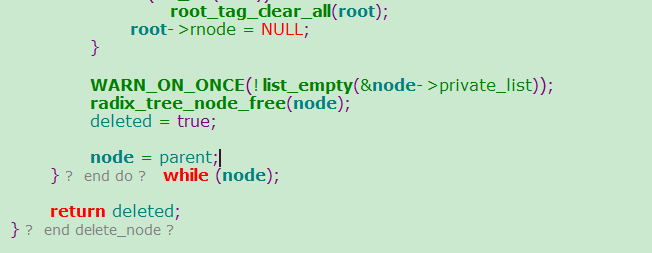
删除一个node，如果当前node的slot个数非0，则判断是不是树根node，

如果这个node是树根节点，则需要调整树高，调用radix\_tree\_shrink实现，如果树根的子节点大于1个则会删除节点失败。

如当前node的slot为0，父node将对应offset清空，并删除node，计数减1。

如node是树根，且count为0则直接删除树，将所有的tag标志清零。





=========================================================

=========================================================