问呢？此时为整个范围创建页表就是一种浪费。因此内存管理器会一直等待，直到线程产生了页面错误后才为相应的页面创建页表。对于保留或提交了大量内存但实际上只零星访问这些内存的进程，这种做法可以大幅提高性能。

这些“暂时不存在”的页表所占用的虚拟地址空间会被计入进程页面文件配额以及系统提交用量。这保证了一旦真正创建后，所需空间是可用的。借助延迟计算算法，即使分配更大块的内存也是一种很快速的操作。当线程分配内存时，内存管理器必须用一段地址作为响应供该线程使用。为此，内存管理器维持了另一组数据结构，用于记录哪些虚拟地址空间已经被进程的地址空间所保留，哪些尚未保留。这种数据结构就叫作虚拟地址描述符（Virtual Address Descriptor，VAD）。VAD 是通过非换页池分配的。

5.9.1 进程的 VAD

内存管理器为每个进程维持了一组用于描述进程地址空间状态的 VAD。VAD 会被组织成一种可以自平衡的 AVL 树（这个名称源自其发明人 Adelson-Velsky 和 Landis，在这种树中，任意节点的两个子树的高度差最大为 1，因此可实现非常快速的插入、查找和删除操作）。平均来说，借此即可在搜索 VAD 对应的虚拟地址时将需要进行的对比次数降至最低。具备相同特征（保留/提交/映射、内存访问保护等）的非空闲虚拟地址中每一块连续的区域，都对应一个 VAD。图 5-49 展示了 VAD 树的一个示意图。

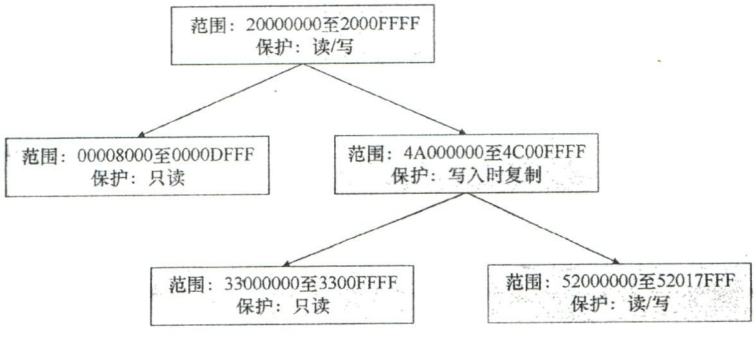

图 5-49 VAD 树的一个示意图

当进程保留了地址空间或映射了节视图后，内存管理器会创建一个 VAD 来存储分配请求所提供的所有信息，例如所保留的地址范围、该范围是共享的还是私有的、子进程是否可继承该范围的内容、应用于该范围中的页面的页面保护。

当线程首次访问一个地址时，内存管理器必须为包含该地址的页面创建 PTE。为此，内存管理器首先会寻找地址范围中包含所访问地址的 VAD，并使用找到的信息填充 PTE。如果该地址落在 VAD 所涵盖的范围之外，或位于已保留但尚未提交的地址范围中，内存管理器会知道该线程在使用这块内存之前尚未申请，因此会生成访问冲突。

实验：查看 VAD

我们可以使用内核模式调试器的!vad 命令查看特定进程的 VAD。首先使用!process 命令找到 VAD 树的根地址，随后将该地址提供给!vad 命令。下例中展示的是 Explorer.exe 进程的 VAD 树。