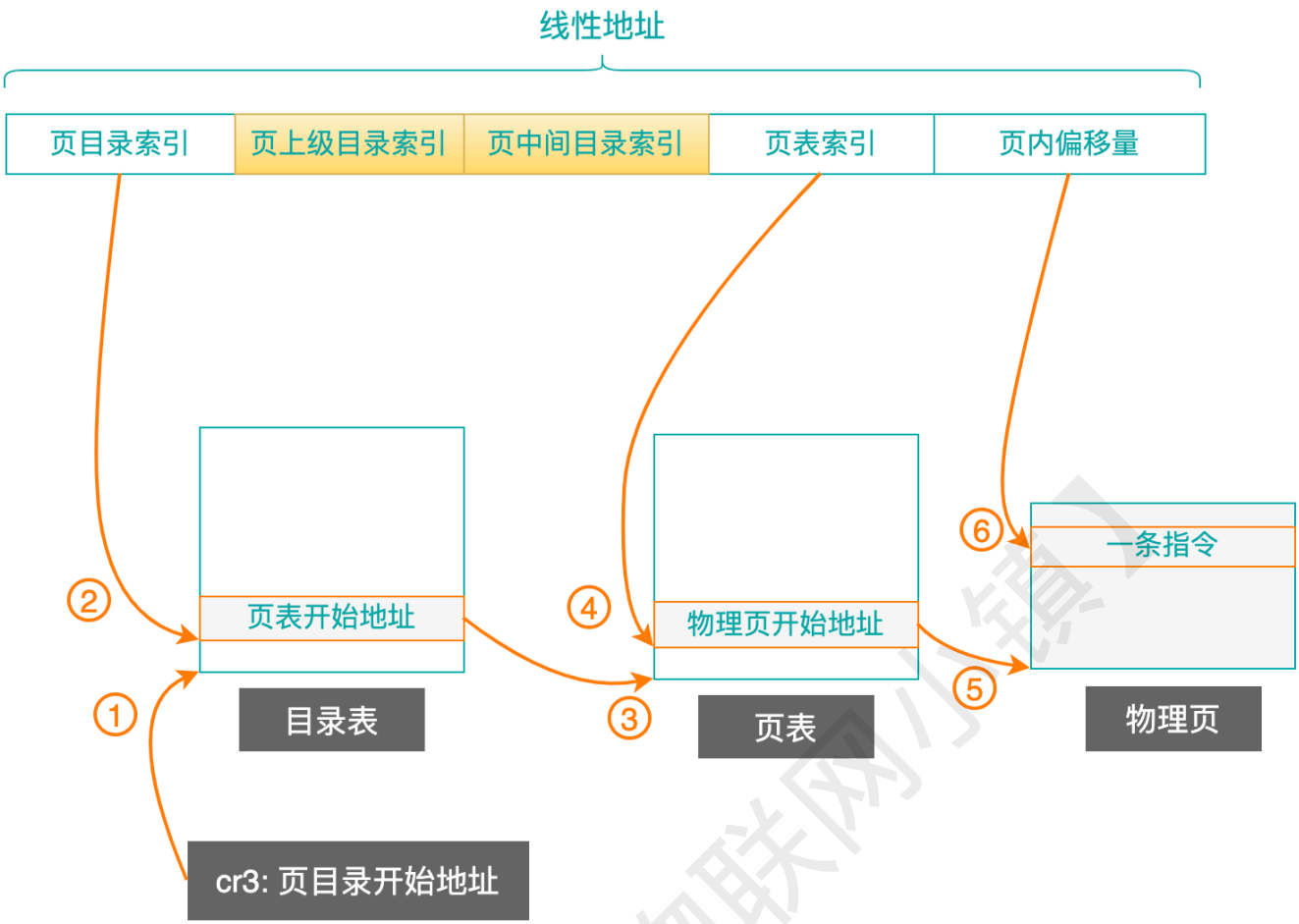
> 作 者：道哥，10+年嵌入式开发老兵，专注于：C/C++、嵌入式、Linux。
>
> 关注下方公众号，回复【书籍】，获取 Linux、嵌入式领域经典书籍；回复【PDF】，获取所有原创文章(PDF 格式)。

目录



在 x86 系统中，内存管理中的分页机制是非常重要的，在Linux操作系统相关的各种书籍中，这部分内容也是重笔浓彩。

如果你看过 Linux 内核相关书籍，一定对下面这张图又熟悉、又恐惧：




这是 Linux 系统中，页处理单元的多级页表查询方式。

其中黄色背景部分：页上级目录索引 和 页中间目录索引，是 Linux 系统自己扩展的，在原本的 x86 处理器中是不存在的，这也是导致 Linux 中相关部分代码更加复杂的原因。

在上一篇文章中，我们主要对 x86 中的页目录和页表的“反向构造”、“正向查找”这两个过程进行了图文并茂的讨论。文章链接在此：Linux从头学15：【页目录和页表】-理论 + 实例 + 图文的最完全、最接地气详解，但是其中有一个环节被特意忽略过去了。

那就是：在操作系统构造页目录和页表的时候，如何对它们自身进行寻址和操作？

这部分内容，也是内存管理中比较复杂的地方，就好比一名医生给病人做手术，但是病人却是“医生自己”。

这篇文章，我们继续通过图片+实例的方式，一起来研究一下内核代码一般都是如何来进行这些“自操作”的。

把这里面的操作机制研究透彻之后，再去看 Linux 内核代码时，就不会晕头转向了。

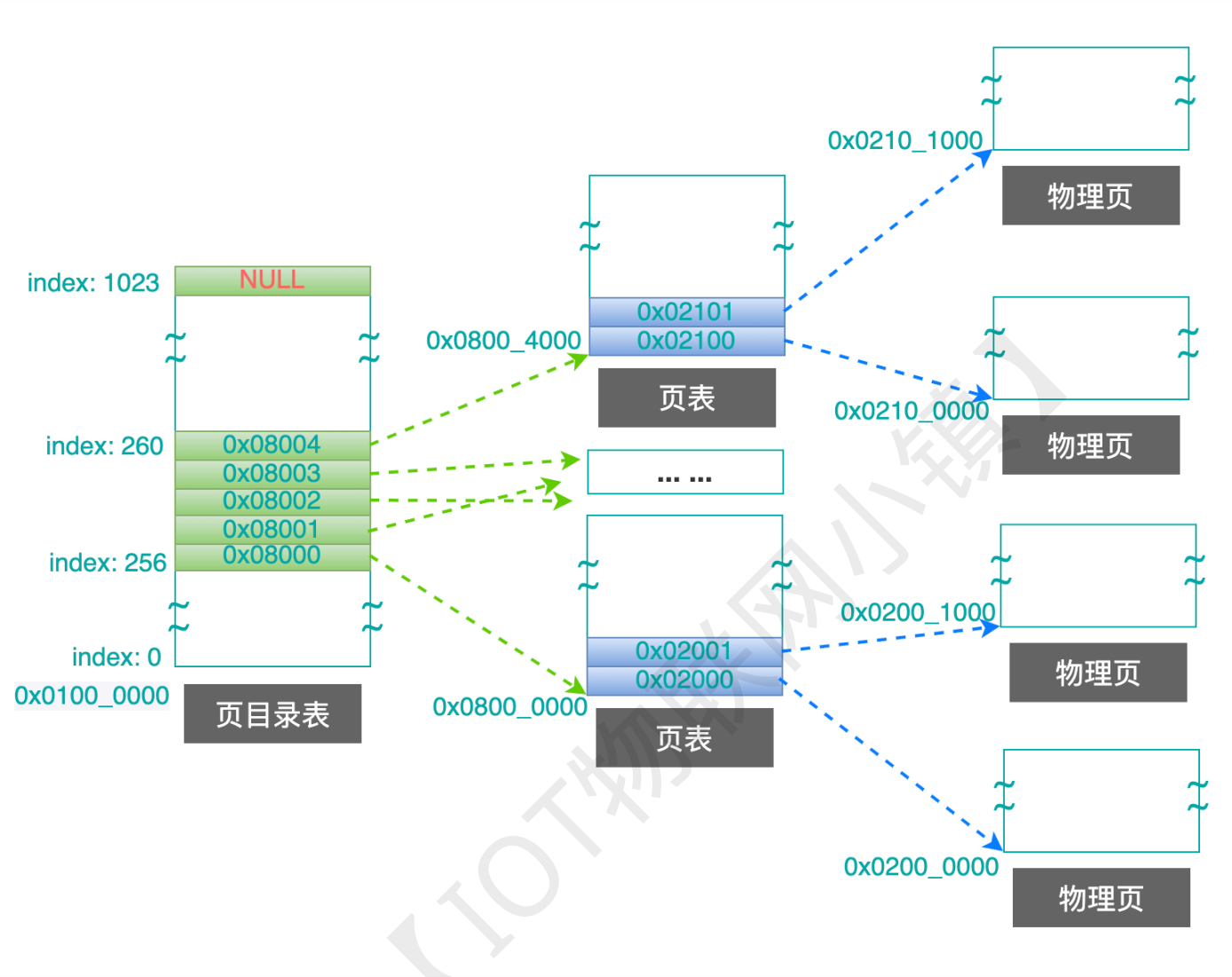
## 问题描述

在上一篇文章中，我们举了这样一个示例：

> 1. 假设实际的物理内存是1 GB;
> 2. 用户程序文件在硬盘上的长度是20 MB;
> 3. 操作系统把用户程序加载到内存中时，从 0x4000_0000 的虚拟内存地址处开始存放;

> 4. 操作系统读取程序结束后，为所有的地址构造好了页目录和页表;

如下图所示：




页目录和页表的每一个有效表项中，存储的地址都是一个个实实在在的物理页的前 20 位(因为一个物理页的长度固定是 4KB，在分配时都是对齐的，末尾的 12 位全部为 0)。

并且页目录和页表“们”自身，都占用一个物理页的空间，所以它们都有自己的物理地址。

当页目录和页表都构造妥当之后，处理器面对一个线性地址，例如：`0x4100_1800`，页处理单元就会按照分级查表的方式，把这个线性地址转换为一个物理地址：

> 1. 拆分线性地址：0x4100_1800 = 0100_0001_0000_0000___0001_1000_0000_0000;
> 2. 根据线性地址的前 10 位，找到页目录中的索引 260，从而确定页表的物理地址是 0x0800_4000（表项中的值是 0x08004，还要补上低位的 12 个 0）；
> 3. 根据线性地址的中间 10 位，找到 0x0800_4000 这个页表中的索引 1，从而确定普通物理页的物理地址是 0x0210_1000（表项中的值是 0x02101，还要补上低位的 12 个 0）；
> 4. 根据线性地址的最后 12 位，确定普通页内的偏移量是 2048，普通页的开始地址加上这个偏移量，就得到了最终的物理地址 0x0210_1800。

详细的讨论过程，请参考上一篇文章：Linux从头学15：【页目录和页表】-理论 + 实例 + 图文的最完全、最接地气详解。

那么，问题来了：

在页处理单元开启的情况下，处理器面对的是线性地址，那么操作系统在构造页目录中的每一个表项的时候，如何对这个表项进行寻址?

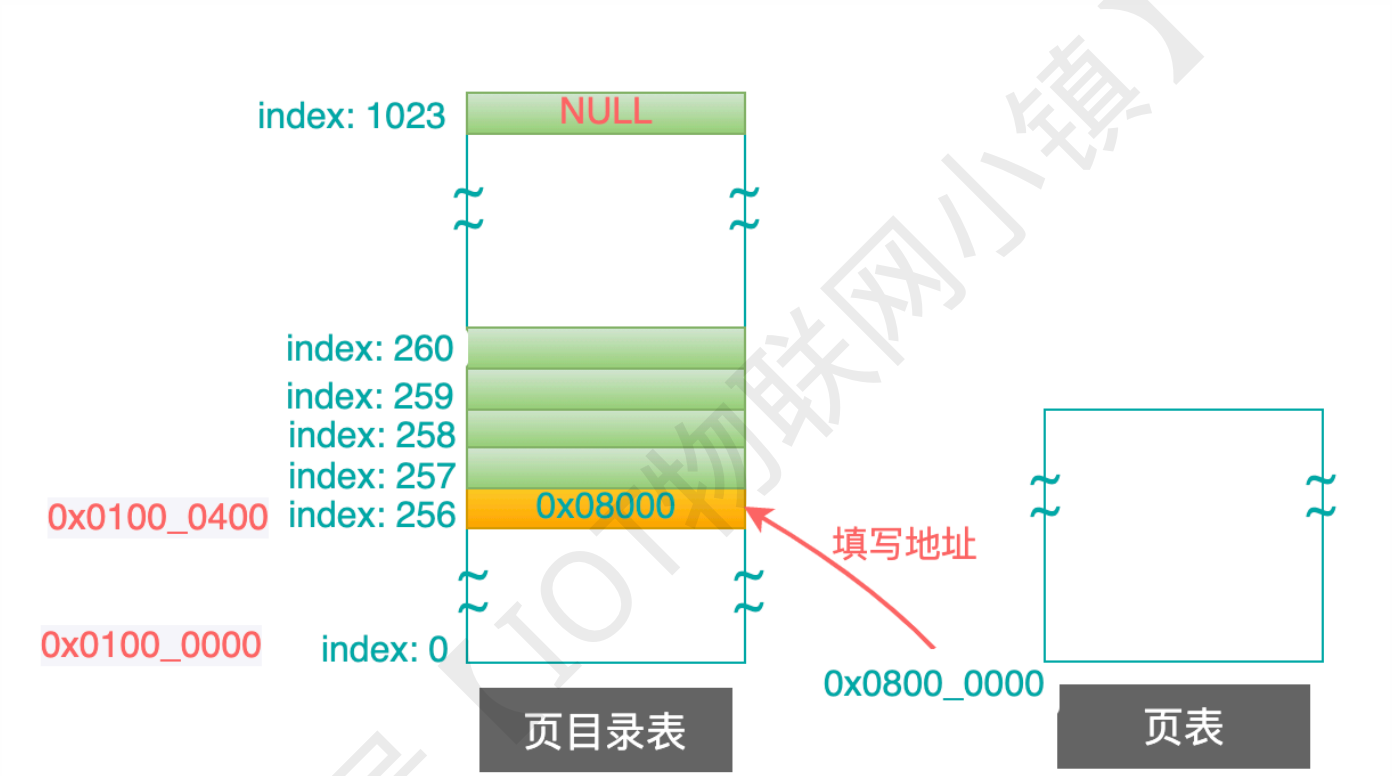
具体到上图来说就是：操作系统想把第一个页表的物理地址 0x0800_0000，填写到页目录的第 256 个表项中时，那么 CPU 就需要找到这个表项，这个表项肯定有物理地址的。

但是，我们不能把这个表项的物理地址直接告诉 CPU，因为 CPU 只接收线性地址，它会自动经过分页单元的处理来得到对应的物理地址。

那么，这个线性地址的值应该是多少呢?

继续用实例来说明，这样容易理解。

假设页目录所处的物理页开始地址是 0x0100_0000，那么第256个表项的物理地址就是 0x0100_0400。




有些小伙伴可能会说：直接把物理地址 0x0100_0400 告诉处理器，不就可以了吗?

这是不对的！

## 处理器接收的是线性地址，不是物理地址

因为现在已经开启了分页处理单元，0x0100_0400 是我们最后想得到的物理地址，而处理器只接受线性地址，虽然我们知道这是一个物理地址，但是处理器不知道啊！

当我们给处理器一个地址的时候，处理器会按部就班的对这个地址进行[段转换]，再进行[页转换]，这时才得到它认为的物理地址。

> 由于使用的是“平坦型”的段结构，所以这里就忽略了段处理过程，直接讨论页处理过程。

所以，我们应该使用某些方法，构造出一个线性地址 addr，让这个地址经过页处理单元之后，得到 0x0100_0400 这个物理地址：

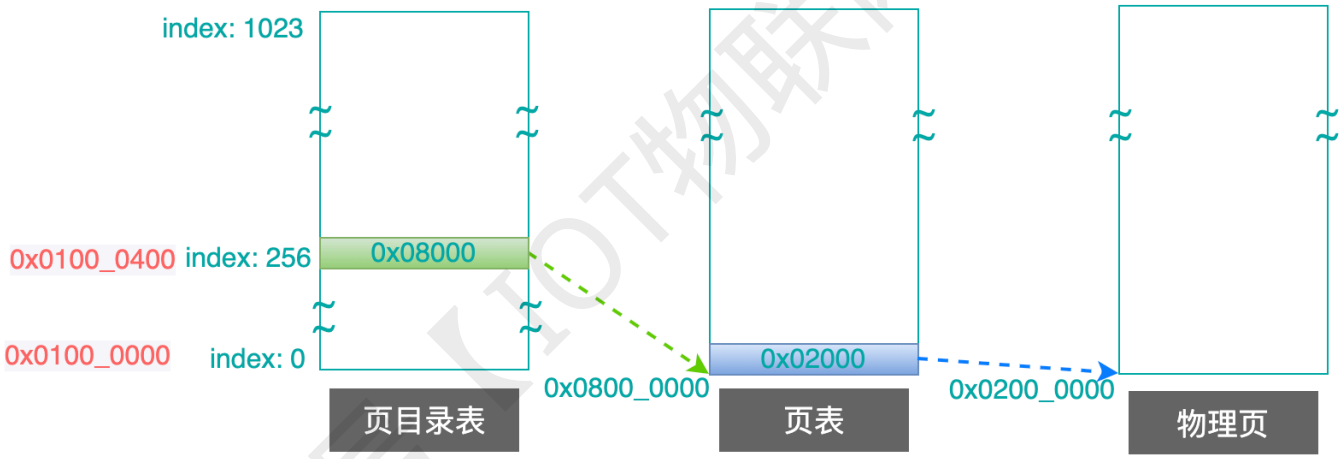

这里有点递归的味道，又有点像一个医生给他自己做一个外科手术！

现在，应该明白面对的问题了吧?

目标就是：通过某种方法，构造出一个线性地址addr，并且通过页处理单元转换之后，得到物理地址0x0100_0400。

## 对页目录进行操作

重新梳理一下思路：如果对一个普通物理页(下文简称为：普通页)里的一个地址处的数据进行操作，需要经过3次查表操作：




从页表的某个表项中，找到的那个物理地址，就是最后要操作的普通物理页。

现在我们的问题是：需要把页目录作为最终的操作对象。

也就是说，从页表中找到的“普通页”的物理地址，应该等于页目录的物理地址！

作为一名软件开发人员，递归思想都是有的。

我们就来构造一个线性地址 addr，让它经过3次查表操作之后，能够指向页目录的物理地址。

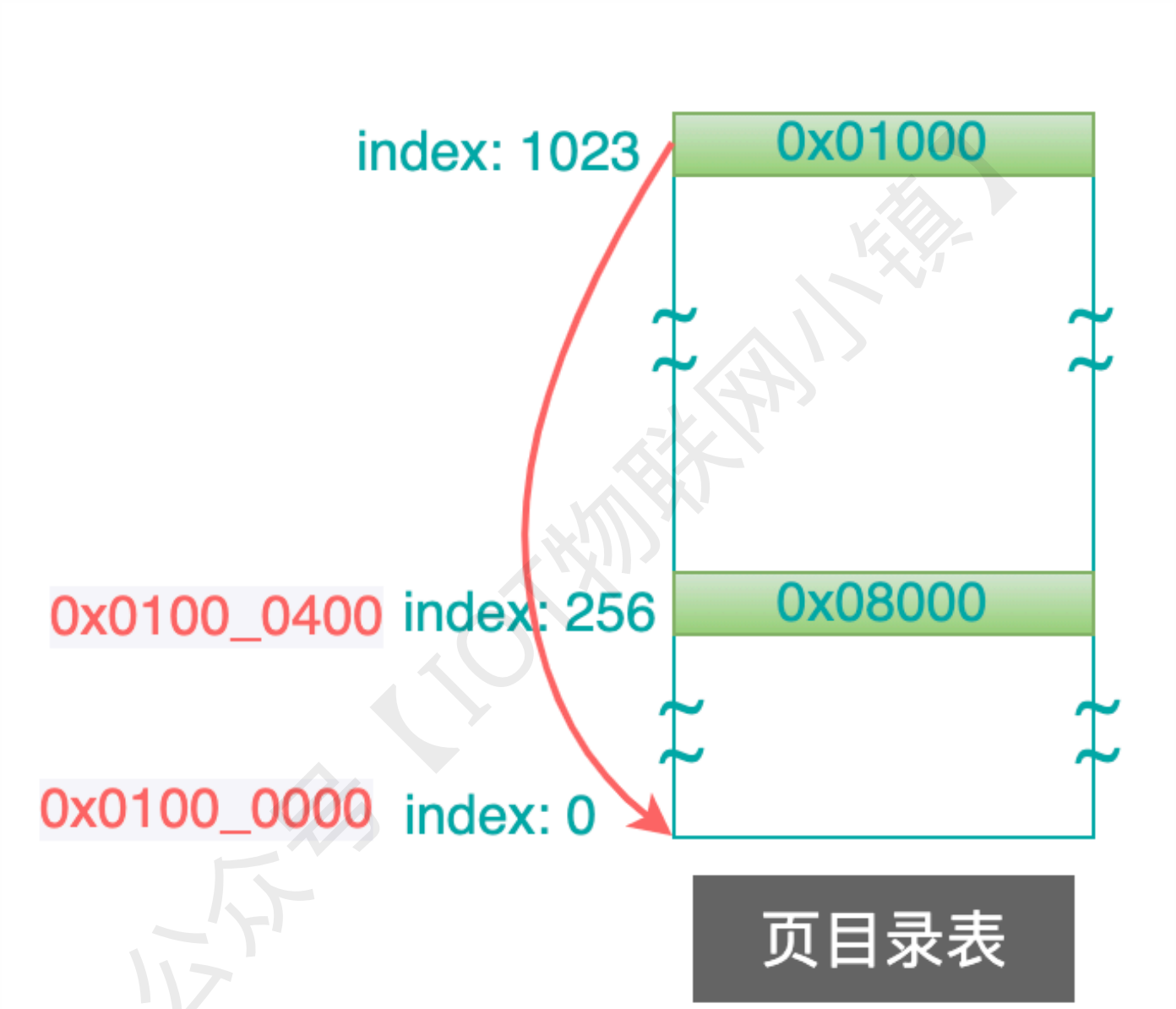
## 一级查表：构造线性地址的前 10 位，来确定页表的物理地址

一级查表：查找的对象是页目录。

线性地址addr的前10位，决定了页目录内的索引。

很显然，需要让这个索引对应的那个表项中所登记的地址，必须是指向页目录自己才可以。

常用的解决方案是：利用页目录中的最后一个表项，让这个表项中记录的地址，指向页目录自己，如下图所示：



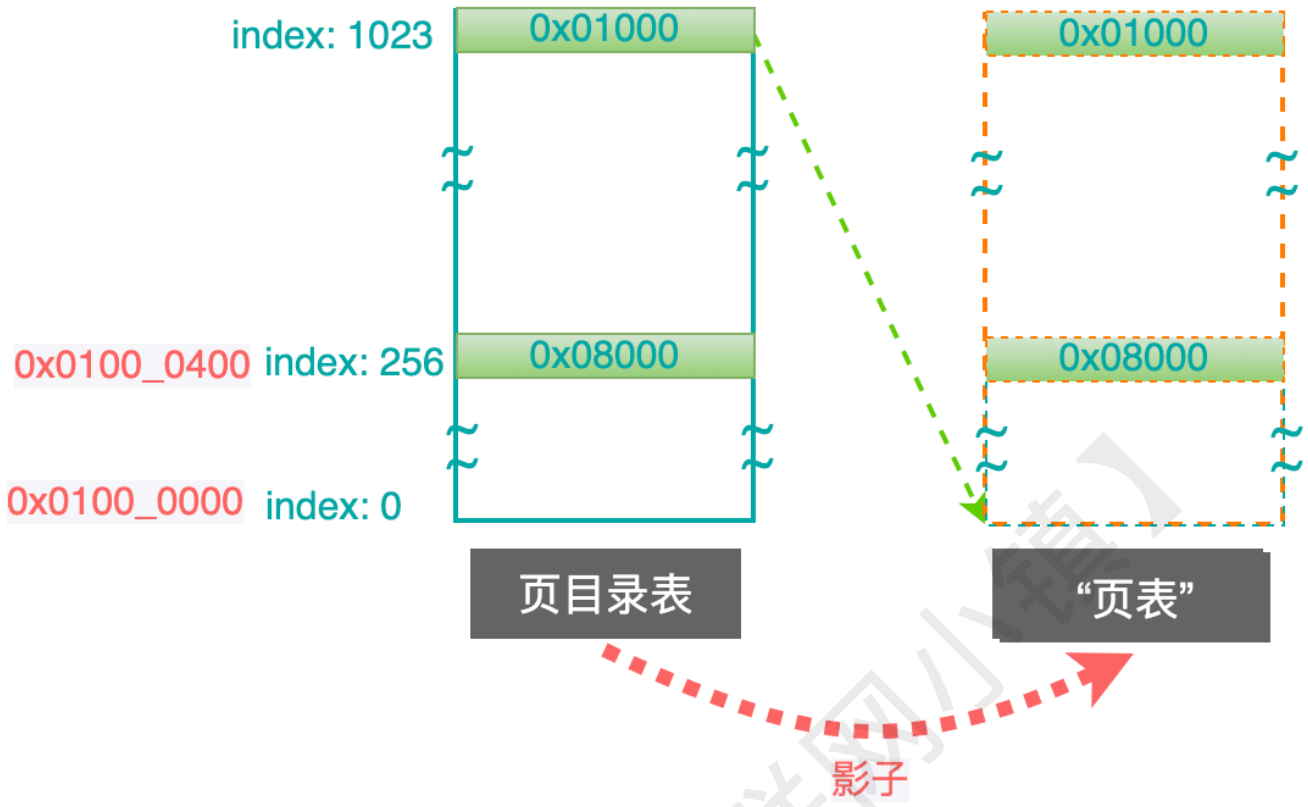



也就是说，预先在页目录的最后一个表项中，填入页目录自己的物理地址，然后只要线性地址addr前10位的值为1023，就能够得到这个表项。

很容易就能得到addr的前10位应该是：0x3FF(二进制：1111_1111_11)。

由于这个表项中存储的地址是页目录自己的开始地址(0x0100_0000, 最后的12个0是自动补上的)，这样就相当于：下面进入第二级查找时，页目录即将被当做“页表”来使用。

如下图所示：




这里红色虚线的“页表”其实就是页目录自己，只是一个影子而已。

## 二级查表：构造线性地址的中间 10 位，来确定“普通页”的物理地址

二级查表：查找的对象是页表，也就是一级查表得到的那个“页表”。

虽然一级查表的结果是页目录自己，但是处理器不管这些，它会把这个表当做页表来使用。

现在，来考虑线性地址addr的中间10位，它决定了页表中的索引号。

很显然，需要继续让这个索引号对应的那个表项中，记录的地址必须继续指向页目录自己。

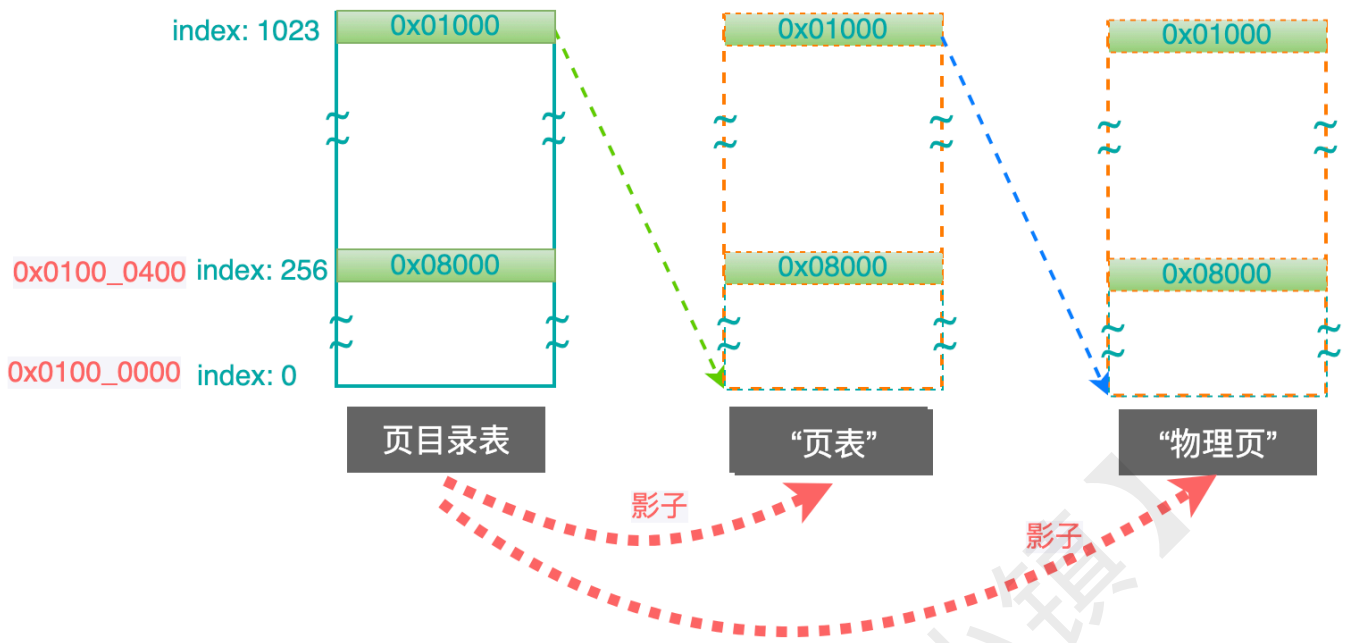
那就继续利用这个“页表”(其实它是页目录)中的最后一个表项呗，就是index = 1023的这个表项。

这个表项中存储的物理地址，即将是最终查表得到的“普通页”的物理地址了。

由于这个表项中，被预先填写了0x01000，补上尾部的12个0之后就是 0x0100_0000，仍然指向页目录自己，完美！

于是，就得到了中间10位的结果：0x3FF(二进制：11_1111_1111)。

如下图所示：



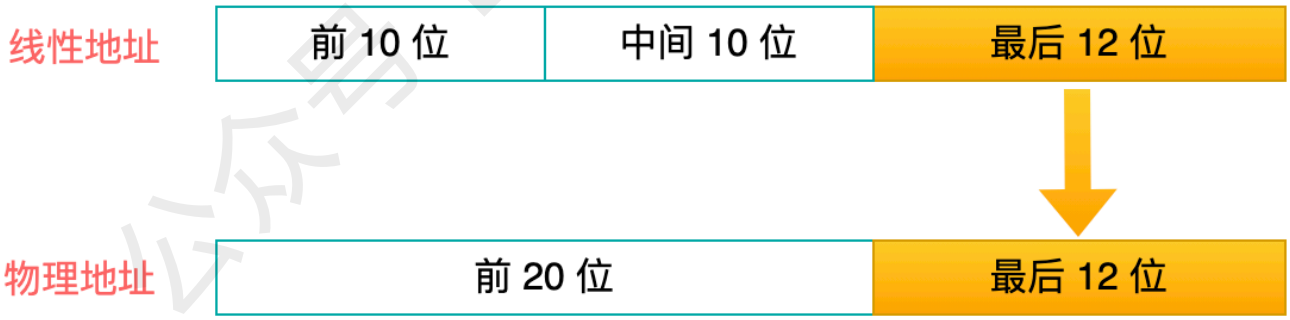

最右面红色虚线的“物理页”，就是二级查找的结果，它本质上仍然是页目录本身，只不过它即将被当做一个普通物理页来使用。

## 三级查表：构造线性地址的最后 12 位，来确定页“普通页”的页内偏移量

现在，已经构造出了线性地址addr（这是我们的最终目标）的前20位，并且经过页表的前两级查表，成功的定位到了页目录自己！

就差最后一步了！

我们知道，从线性地址到物理地址的转换过程中，最后的12位表示页内偏移，是直接从线性地址中取过来的。




也就是说：线性地址 与 物理地址 的最后12位偏移量，值是一样的！

所以，我们就反过来倒推一下：

我们最终想操作的是页目录中第256个表项，它的物理地址是 0x0100_0400，这个物理地址距离这个页目录开始位地址的偏移量是：0x400(0x0100_0400 减去 0x0100_0000)。

因此，线性地址addr中的最后12位的值也应该是0x400。

### 三个地址段合体

把上面三个步骤中，得到的地址聚合在一起：


`0xFFFF_F400` 就是最终想得到的线性地址!

也就是说，我们只要把这个线性地址 `0xFFFF_F400` 告诉处理器，它就会经过页处理单元的转换，最终查找到页目录这个物理页中的第 256个表项，也就是物理地址 `0x0100_0400`。

> 例如：mov [0xFFFF_4000], xxxx

以上就是操作系统在操作页目录自身时，所采取的策略。

具体到每个操作系统来说，可能稍微有差别，但是其中的道理都是差不多的。

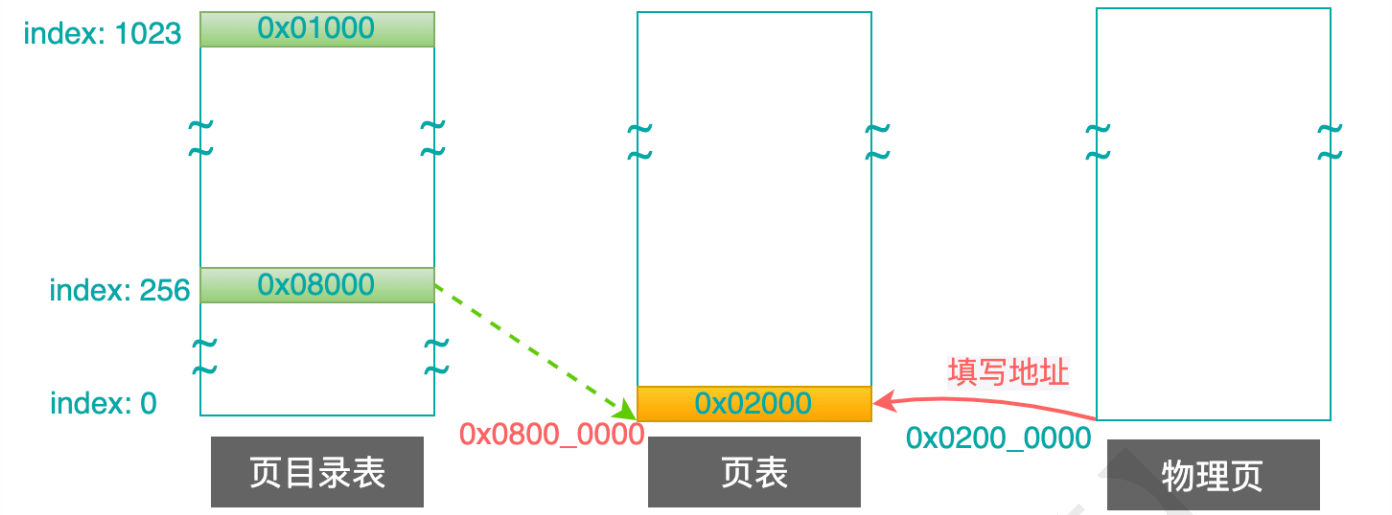
例如本文开头的第一张图中，Linux 使用了4级表格来查找，并且中间的两个表格还可以省略不用。

如何跨过中间的这两个表格，Linux内核代码中的代码更复杂一些，但是策略都是一样的。

## 对页表进行操作

既然已经弄明白了操作系统是如何操作页目录的，那么对页表的操作就不是什么大问题了。

比如下面这张图：




目标：把最右面的普通物理页地址 0x0200_0000，放入 0x0800_0000 这个页表的第一个表项中(只需要存储前20位)，那么应该传递什么样的线性地址给处理器?

思路是完全一样的。

## 一级查表

按照正常的分页查找流程，从页目录的某个表项中，查找我们想操作的那个页表。

页目录中的这个表项位于索引值256的地方，因此可以构造出线性地址的前10位是：0100_0000_00(0x100)。

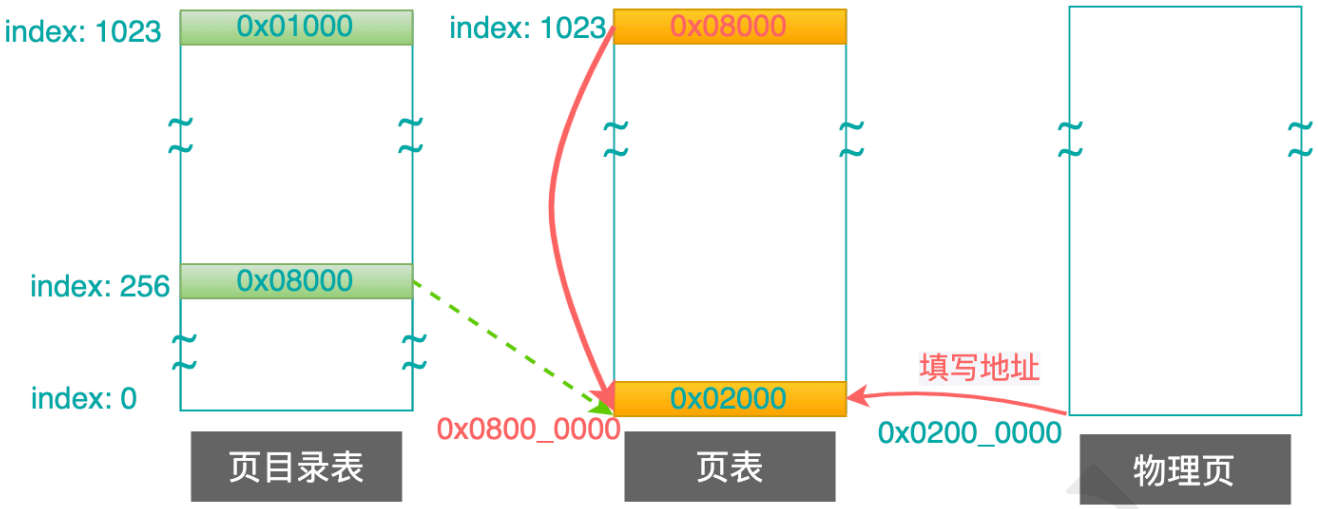
所以，经过一级查表得到的这个页表的物理地址是 0x0800_0000。

## 二级查表

利用这个页表的最后一个表项(index = 1023)，预先填写一个地址(0x08000)，让它指向这个页表自己的开始物理地址。

于是，可以构造出线性地址的中间10位是：11_1111_1111(0x3FF)。

由于这个表项中存储的地址是0x0800_0000，指向的正是页表自己，只不过马上它就被当作普通物理页被使用。




## 三级查表

此时，已经找到最后的普通物理页了(其实它是一个页表，被当作普通物理页使用)。

线性地址的最后12位，可以直接从最后想操作的那个目标物理地址中最后12位直接拿过来。

我们的目标是：操作页表中的第 0 个表项，这个表项的物理地址是 `0x0800_0000`，最后的12位偏移量是 `0000_0000_0000`。

把以上3个地址段合体，即可得到正确的线性地址：




------ End ------

这里讨论的方法，并不是处理页目录和页表的唯一方式。

当处理逻辑更加复杂时，可能需要对页目录或页表中更多的表项，进行一些特殊的预处理。

如果你想挑战一下，可以看一下Linux内核中的相关文档或代码！

在这个系列中，关于页目录和页表的知识点就介绍结束了。

如果文中有错误或者误导的地方，非常期待与您一起探讨、学习！

写这篇文章真不容易，让我深深的体会到那句话：

> 写作就是：将网状的思考-通过树状的结构-用线性的语言清晰的表达出来。

如果您觉得还不错，请点个赞，鼓励一下，转发给身边的技术小伙伴，真心的感谢！

推荐阅读

【1】《Linux 从头学》系列文章

【2】C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻

【3】原来gdb的底层调试原理这么简单

【4】内联汇编很可怕吗？看完这篇文章，终结它！

其他系列专辑：精选文章、应用程序设计、物联网、 C语言。


星标公众号，能更快找到我！

C/C++、物联网、嵌入式、Lua语言

Linux 操作系统、应用程序开发设计


扫码关注公众号


道哥 个人微信

喜欢请分享，满意点个赞，最后点在看。