每个程序都面对一个或者多个地址空间。你写一个程序，说*(0x1234)=10，这里就索引了一个地址。所有可以被索引的地址，就构成一个地址空间。一个CPU上的程序可能不止一个地址空间，比Intel支持LPC的指令，用inX和outX指令索引的地址空间和用movb索引的地址空间就是两个相对独立的地址空间。有些比如哈佛构架的CPU，访问指令和访问数据内存也会使用不同的地址空间。

一般冯诺伊曼计算机访问内存的是同一套指令，这套指令构成一个地址空间。CPU的指令集就通过这个地址空间要求CPU访问内存。

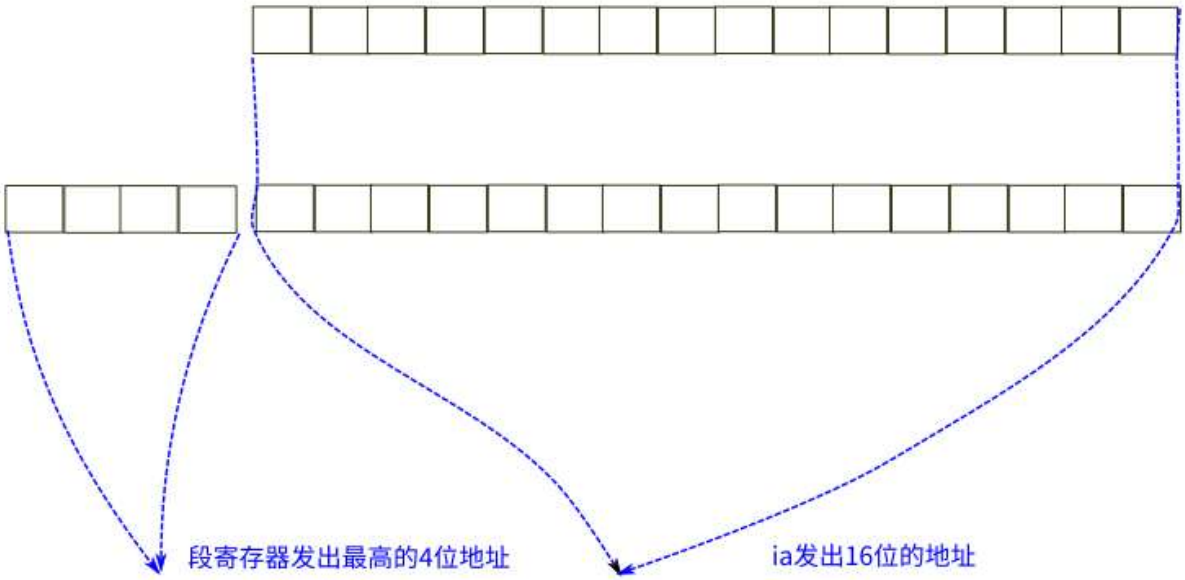
最早的CPU的指令直接索引到物理内存的地址，也就是说，指令说要访问0x1234，访问的就是物理内存下标为0x1234的那个单元的内容。但我要请读者注意的是，这里其实有两个地址空间，一个是指令构造的地址空间（我缩写为ia），另一个是物理地址总线上发出的地址构成的地址空间（我缩写为pa），ia和pa不是完全一一对应的。因为指令的演进具有滞后性，软件是基于指令集写的，写好了，我希望未来做一个更强大的CPU的时候，不要让我还要改写软件。所以比如你的物理地址只有10位，你不可能就做一个指令集，里面的地址都是10位的，以后要支持更多内存，你把物理地址做成11位，你就要重新搞一个指令集。这样软件就没有演进性了。所以，ia空间是相对稳定的，pa则变化得更快一些。我们后面说到的所有故事，都和这两者的互相变化引起的。我需要读者非常注意这个问题。

[段寻址]

PC行业兴起的时候，常常使用16位ia的指令集，16位表示64K的空间，那个时候好大好大了，因为物理内存通常只有16K或者更低的，这时你如果做了14位的物理内存地址空间，CPU发一个16位的地址下来，你只要不看最高两位，你就可以寻址了。

但我们对内存的猜测永远是缺乏想象力的（也是节省资源的需要了，这个我另外找地方介绍），很快物理内存就超过16位了。但立即修改指令集会影响市场，这时，有什么办法扩展指令集，让ia的空间补到和pa空间可以寻址一样的长度呢？

对了，很自然的想法，加一个寄存器表示地址的高位就可以了：



所以，段地址这种东西，你不要被它那些什么权限啦，内存分段啦这些概念蒙蔽，这就是个简单的地址扩展方法，至于其他功能，段寄存器不可能做成4位，所以剩下的位，闲着也是闲着，就加点功能上去，如此而已。

[页寻址]

段寻址的缺点是显而易见的，你不可能每次寻址都分两次来加载两个数据，这样太麻烦了。所以才需要把高位地址封装成“段”的概念，比如x86的意思是，你访问代码用cs寄存器，访问数据用ds寄存器，这样每次你就不用改这两个寄存器玩了。但这样整个程序也就只有128K而已，如果我的数据很大，超过64K的范围，就又得切换这东西了，x86说，不够是吗？我再给你一个es寄存器……好吧，这样和稀泥的方式是玩不久的。

很快MMU的概念就被提出来了。MMU是一个翻译器，ia发出的地址，都经过MMU进行翻译，变成pa。所以，MMU本质上是一个函数，实现pa=mmuf(ia)。

问题是，MMU基于什么算法来进行翻译?

我们用最“自由”的方式来定义这个算法，最好就是可以指定每个ia的pa是什么。但这样就需要一张ia空间那么大的表。那么大的表，什么地方可以放得下呢？——也只有内存了。但内存填满了这么一张表，还放什么呢?

所以一个ia对应一个pa，这个算法在数理逻辑上就不合理。那我们可以退而求其次，每段连续的ia对应一段连续的pa，这样，这个段的大小，就是可以节省的映射表的项数了。比如，一个16位的ia，用10位作为一段，那么原来需要64K个映射项，现在只需要64K/1K=64项而已。

由于每段的大小都固定了，我们把这种段称为“页”。现在很多平台都是用12位作为一页，所以，现在很多系统都是4K为一页，但这不是固定的，根据需要是可以变的。

基于页寻址的方式，就称为页寻址，实现页寻址的那个映射表，就称为页表，页表放在内存中。在长期的发展过程中，现在的页表不是一个简单的数组，而是一个分层的数组，这些细节，读者可以

自己看芯片编程手册，但本质都只是为了实现ia到pa的对应，所有的优化都是为了节省空间而已。

和段寻址一样，既然页表都发明出来了，和页有关的一些保护措施也会依附上去，让一个页只读啦，决定从这个地址发出去的请求能否合并啦，有没有边界效应啦，这些都可以设置到页表的对应项中。但仍和段寻址一样，这个问题不是页寻址问题的本质。

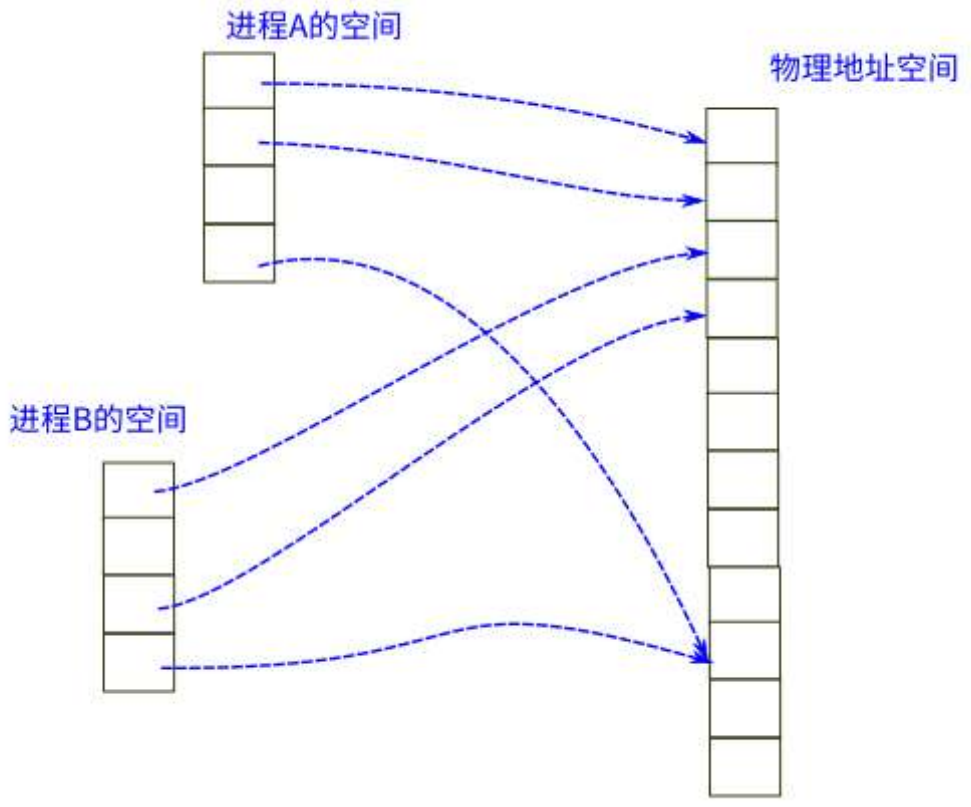
页表的存在，导致所有的ia发出的地址和pa没有直接的映射关系了，为了描述的方便，我们现在开始在这种情形下，把ia称为虚拟地址，va。

[进程空间]

有了页寻址，进程的概念很自然就被发明出来了。页表是MMU的输入，那么一个自然的想法，我们有多个页表，一会儿用这个，一会儿用那个，这样我们可以根据需要用不同的内存，这个用来表达不同MMU的概念，体现在软件的观感上，就是进程。

当然，MMU封装为进程，还和软件的“线程”概念进行了一定程度上进行了绑定，所以，软件的进程，不是简单的MMU的封装，但我们在理解寻址的概念的时候，是可以简单这样认为的。

有了页寻址的概念，进程的空间和物理地址看到的空间就很不一样了：



这样为前面谈到的ia空间和pa空间的不匹配制造不少机会。比如32位的处理器，ia空间是32位的，但随着发展，内存越来越大，超过ia可以寻址的4G的空间了。但我们只要保证页表中的pa超过32位，我们就可以让多个进程合起来使用超过4G的物理地址空间。Intel的PAE就是这样的技术，本质上是扩展页表中pa的范围。

[进程的内核空间]

所有从进程发出的va访问，都通过MMU翻译，包括请求从一个进程切换到另一个进程。这样一来，进程的权力太大了。所以，还是很自然的，我们在MMU中可以设置一个标志，说明某些地址是具有“更高优先级”的，一般情况下不允许访问，要访问就要提权，一旦提权，代码执行流就强行切换到这片“高级内存”的特定位置，这样，就给一个进程指定了两个权限级别，一个是所谓的“用户态”，一个是“内核态”，用户态用于一般的程序，内核态用来做进程的管理。

用户态和内核态都在同一个页表中管理，所以，本质上，它们属于同一个进程。只是它们属于进程的不同权限而已。

为了实现的方便，现在通常让所有进程的MMU页表的内核空间指向物理空间的同一个位置，这样，看起来我们就有了一个独立于所有进程的实体，这个实体被称为“操作系统内核”，它是管理所有进程的一个中心软件。

其实操作系统内核并不是必须做成全局唯一的，只是这样设计最方便就是了。

内核空间的存在分少了进程的可用空间，本来一个32位进程可以用4G的空间的，但由于要留一部分给内核用，就不能是4G了。如何切分这个空间，是一个设计上的权衡。给用户空间切少了，作为以用户程序为中心的操作系统来说，这影响了竞争力。给内核空间切少了，内核管理上不方便，能力降低。现在Linux在32位系统上通常使用 3G:1G这样的切分关系。

如果是微内核操作系统，由于内核的功能少，这个比例可以进一步降低，比如我们可以留1M给内核，其他空间全部留给用户态。这也算是微内核操作系统的一个优势吧。

有人不明白为什么Linux内核中需要有Himem这个概念，这样一看也很好理解了，内核有权访问所有内存，但它只有1G的虚拟空间，如果你的物理内存不到1G，在它1G的va空间内，它可以对物理内存做一一映射。但如果你的物理内存超过1G，这种方法就不可行了，你只能根据需要把不同的物理内存映射进来使用。把线性映射的部分分开，不能线性映射的部分，我们就称为Himem了。线性映射的部分和Himem在什么地方分解，又是一种权衡。但这种权衡，老实说，现在我们已经不怎么关心了，因为现在已经切换到64位系统，虚拟空间高达16EB，分给内核用绰绰有余（实际上，现在大部分64位系统的物理地址只有48位），内核虚拟地址总是可以线性映射所有的物理地址的。

[ASID]

不知道读者是否考虑过这个问题：页表放在内存中，MMU收到一个地址访问请求，为了做翻译，它首先要访问内存，得到对应的翻译项，然后再去访问内存。这是不是表示每个地址访问，其实要访问两次内存?

访问内存是很慢的，MMU显然不会干这种蠢事。所以，它使用了CPU设计者的万能法宝：加个Cache呗。

MMU的Cache称为TLB，MMU进行翻译的时候，首先从TLB里读页表项，如果页表项不在TLB中，再读内存，把内存中的对应项写到TLB中，然后在用TLB中的数据进行翻译，这样就不用每次做翻译都要读一次内存了。

这样的算法可以得以实施，取决于一个现实，称为程序的局部性原理，即假设程序在一段时间内，访问的都是特定范围内的内存，这样MMU就不用经常重新加载TLB了。

但这个局部性原理在特定的场景上是不成立的，一种情形是大型数据库。数据库有个很大的表在内存中，经常要跨越页进行随机访问，这样就有很大的机会造成TLB失效（这种情形称为Cache污染），操作系统解决这个问题的方法常常是使用HugePage，也就是让特定的页不止4K，比如可以是256M，这样，256M的内存仅仅需要一个TLB项，甚至可以把这个TLB项锁死在TLB中（不允许替换），这样数据库的性能就可以提高。

第二个影响局部性原理的场景是微内核操作系统。微内核操作系统不从内核请求操作系统服务（微内核操作系统的内核仅仅提供进程切换和进程间通讯手段），而是直接从另一个进程获得这个服务，这大大提高了进程切换的频度。问题是，A进程向B进程发出一个请求，B进程1ms就搞定了，又切换回A进程。但你切换进程，我就得把整个TLB清掉，所以，为了这个1ms的请求，MMU就得重新加载进程A需要的所有页表项，这个对性能的影响也太大了。

解决这个问题的方法是引入ASID，也就是让MMU认识进程id。所以，在TLB的表项中，每个项都有一个ASID，当你切换进程的时候，你不需要清空TLB的，B进程用不了里面A进程的页表项，这样你切换回A进程的时候就不需要重新加载对应的项了。

ASID这个概念在我们后面讨论IOMMU/SMMU的时候有特别的作用。

[DMA]

好了，CPU和内存的恩怨告一段落了。现在我们该看看外设了。Intel最早的时候给了外设独立的地址空间， 要访问外设就用io指令，独立指定要访问的地址，和内存访问没有关系。但这样划分没有什么意义，所以现代的CPU通常把这两个地址空间合并。只是在做页表映射的时候，给外设空间特殊的设定（比如内存读，如果要求读一个字节，我读64个字节也不会有错，但外设就不行了，所以要进行特殊的设定）。

读写外设空间是个很慢的动作，因为从总线上发出一个读写操作，都要等被读写一方响应的，CPU做了很多优化来优化内存访问的QoS，但对外设是没有什么办法的（这玩意儿可以动态插进来的，没法做任何假设），所以，遇到一个不好的设备，进行io读写，可以导致整个指令停住不能动。

现在更常见的方法是尽量不去碰外设的IO空间，而是把数据放在内存中，让外设自己去读。这个动作就称为DMA，现代SoC中外设用来访问内存的部件，通常就叫DMA控制器。

早期的外设通常很简陋，它自己的DMA访问内存的总线常常比CPU的物理总线要短。比如你一个32位的CPU，物理总线是40位，但外设的DMA就只有16位。这样如果CPU发起一个DMA请求，让外设自己去读一个超过16位的物理内存地址的空间，这个外设根本就没有能力访问。

为了解决这个问题，Linux内核为驱动提供了DMA专门的内存分配函数。保证你分配的空间在外设DMA可以访问的范围内（这就是内核分区ZONE_DMA的由来），这样，你做DMA就能保证这个物理地址是可以被外设访问到的。

现在的外设越来越强了，很多外设的DMA都拥有和CPU一样长的地址，对于这样的设备，就不需要用那个分配函数来解决问题了。

[IOMMU和SMMU]

外设没有MMU，所以CPU要提供一个地址给外设访问，必须使用物理地址。但使用物理地址是件很麻烦的事情，因为在虚拟地址上连续的地址，在物理地址上不一定连续。

所以，又作为一个“自然”的想法，有人就考虑把MMU移到设备DMA上来，这个概念就是IOMMU（AMD的概念）和SMMU（ARM的概念）了。由于我对ARM比较熟一点，我们用SMMU举例。

SMMU的原理和MMU一样，只是作用在设备一侧，SMMU可以复用MMU的页表，也可以自己创建页表。通常我们不会选择复用页表，因为SMMU是针对每个设备的，我们显然希望设备只看到CPU希望给它看到的空间，而不是所有空间，否则一不小心我们插一个黑客设备进来，整个操作系统就暴露了。

有了SMMU，我们对外设做DMA请求，就不需要使用物理地址了，使用和进程一样的虚拟地址就可以了。只要保证SMMU的页表中对应的设置和进程一致就行。Linux内核中做DMA请求之前，要求做dma_map，就是为了完成SMMU的对应设置。

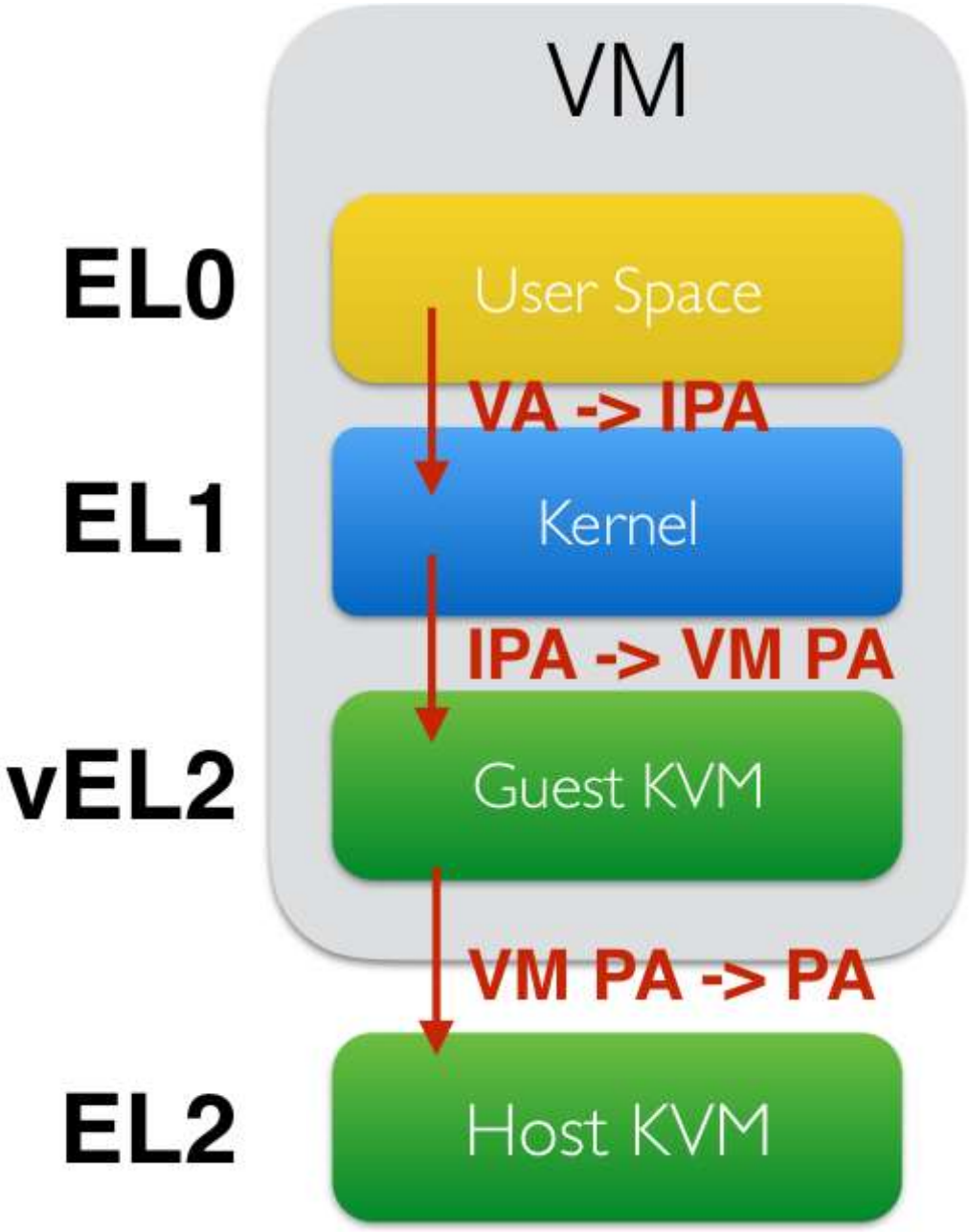
实际上，SMMU的问题比MMU的问题复杂。考虑这么一种情况：进程A发起一个DMA请求，我们把这个页表项放到设备的SMMU上了，现在CPU切换到进程B，CPU的页表变了，但设备不知道啊，设备还是向当初的虚拟地址上写。这时进程B又发起一个DMA请求，在同一个虚拟地址上要求做DMA，设备不是晕菜了？

为了解决这个问题，那个ASID又被捡起来了，SMMU的页表中带上ASID，这样，CPU一个时刻只有一个进程的页表在用，而设备却同时用着好多的页表。

[两层MMU和SMMU]

MMU和SMMU的概念在虚拟化时代继续进行着演进。如果用户程序运行在虚拟机中会怎么样？ARMv7/v8提供两层地址映射，虚拟机中的程序发出一个va，第一次翻译为ipa（中间pa），如果下面是虚拟机创建的，虚拟机也可以给MMU/SMMU指定一个页表，说明ipa如何翻译为pa。这样，虚拟机的效率就提高了，硬件就认知了虚拟机的概念。va一次使用两层的页表，直接得到pa。这种情况下，实际上就有进程，Hypervisor和物理三个地址空间了。

问题是，如果我要在虚拟机里面再跑一个虚拟机怎么办呢？这个方案也在做。显然我们不能无限增加MMU/SMMU的翻译层数了，所以就只能在Hypervisor的调度上下功夫，虚拟机的页表根据Hypervisor的调度进行动态的切换。类似这样：



[从设备侧发起缺页]

设备使用虚拟地址，又再带来一个问题，根据现代操作系统的管理方式，虚拟地址很可能是没有分配物理空间的，在CPU一侧如果发生这种事情，就通过缺页例程来解决。但在设备侧怎么办呢？

一种简单的方法是发起DMA前，把虚拟空间先pin死到物理内存中。

但这样是有缺点的，因为如果一大片内存给了设备，设备只需要一头一尾两页的内存呢？那我不是白干了？

所以，更好的办法是让设备也有缺页的能力。但设备缺页是比CPU缺页复杂的。CPU缺页你只要在缺页中断中把页加载进来。但设备缺页，设备可没有能把缺掉的页加载进来。所以，CPU上需要一个代理（在PCIE的ATS标准中称为TA，转换代理），设备上进行页面转换的装置（称为ATC，地址转化客户端）在发现缺页的时候，通过对TA发送缺页信息，让TA通知CPU产生设备缺页中断，然后把对应的LTB项发送给ATC，才能继续下去。

x86在Linux中加入svm框架来处理这个过程（参考Linux内核代码中的driver/iommu/intel-svm.c），ARM也在准备合入到这个框架中（参考：spinics.net/lists/linux）。其实SVM是个比较容易搞定的事情，这个设计的主要难度在总线系统上，CPU的缺页到异常的过程几乎是原子或者同步的，但如果总线系统要经过几次消息交换来完成这个过程，就有很多出错的机会了。。

[统一内存模型]

把上面的考虑综合在一起，我们发现我们逐步走向一种“全局地址”的内存模型了，也就是说，要加工的数据，统统放在内存里，CPU来动一部分，GPU来动一部分，FPGA来动一部分，硬加速加速器来动一部分。

这个设计，就叫“统一内存模型”，ARM联盟的CCIX，HPE的Gen-Z，现在都在为最终的这种模型填砖加瓦，这也是我们未来追求的统一异构计算的软件模型。

所谓CCIX，其实是是PCIE的一个变种，PCIE总线的协议栈分4层，应用，会话，链路和物理。CCIX替换了其中的应用层和部分的会话层，实现可以深度不是那么深，但对于内存是Cache Coherence的高速外设总线。从而让基于CCIX的外设可以用更方便简单的方法修改加入内存的协同访问中。

而Gen-Z，是通过一个Fabric访问更大的，非CC的内存，让全局的系统可以访问到更大的内存空间（比如256T）。