客户机，这里新的k覆盖了原来旧的，只是因为服务器修改了它。实际上，通过引用调用（call-by-reference）的标准调用序列被复制-恢复（copy-restore）所替代了。然而不幸的是，这个技巧并不是总能正常工作的，例如，如果要把指针指向一幅图像或其他的复杂数据结构就不行。由于这个原因，对于被远程调用的过程而言，必须对参数做出某些限制。

第二个问题是，对于弱类型的语言，如C语言，编写一个过程用于计算两个矢量（数组）的内积且不规定其任何一个矢量的大小，这是完全合法的。每个矢量可以由一个指定的值所终止，而只有调用者和被调用的过程掌握该值。在这样的条件下，对于客户端桩而言，基本上没有可能对这种参数进行编排：没有办法能确定它们有多大。

第三个问题是，参数的类型并不总是能够推导出的，甚至不论是从形式化规约还是从代码自身。这方面的一个例子是printf，其参数的数量可以是任意的（至少一个），而且它们的类型可以是整形、短整形、长整形、字符、字符串、各种长度的浮点数以及其他类型的任意混合。试图把printf作为远程过程调用实际上是不可能的，因为C是如此的宽松。然而，如果有一条规则说假如你不使用C或者C++来进行编程才能使用RPC，那么这条规则是不会受欢迎的。

第四个问题与使用全局变量有关。通常，调用者和被调用过程除了使用参数之外，还可以通过全局变量通信。如果被调用过程此刻被移到远程机器上，代码将失效，因为全局变量不再是共享的了。

这里所叙述的问题并不表示RPC就此无望了。事实上，RPC被广泛地使用，不过在实际中为了使RPC正常工作需要有一些限制和仔细的考虑。

8.2.5 分布式共享存储器

虽然RPC有它的吸引力，但即便是在多计算机里，很多程序员仍旧偏爱共享存储器的模型并且愿意使用它。让人相当吃惊的是，采用一种称为*分布式共享存储器*（Distributed Shared Memory，DSM）（Li，1986；Li 和Hudak，1989）的技术，就有可能很好地保留共享存储器的幻觉，尽管这个共享存储器实际并不存在。有了DSM，每个页面都位于如图8-1所示的某一个存储器中。每台机器有其自己的虚拟内存和页表。当一个CPU在一个它并不拥有的页面上进行LOAD和STORE时，会陷入到操作系统当中。然后操作系统对该页面进行定位，并请求当前持有该页面的CPU解除对该页面的映射并通过互连网络发送该页面。在该页面到达时，页面被映射进来，于是出错指令重新启动。事实上，操作系统只是从远程RAM中而不是从本地磁盘中满足了这个缺页异常。对用户而言，机器看起来拥有共享存储器。

实际的共享存储器和DSM之间的差别如图8-21所示。在图8-21a中，是一台配有通过硬件实现的物理共享存储器的真正的多处理机。在图8-21b中，是由操作系统实现的DSM。在图8-21c中，我们看到另一种形式的共享存储器，它通过更高层次的软件实现。在本章的后面部分，我们会讨论第三种方式，不过现在还是专注于讨论DSM。

先考察一些有关DSM是如何工作的细节。在DSM系统中，地址空间被划分为页面（page），这些页面分布在系统中的所有节点上。当一个CPU引用一个非本地的地址时，就产生一个陷阱，DSM软件调取包含该地址的页面并重新开始出错指令。该指令现在可以完整地执行了。这一概念如图8-22a所示，该系统配有16个页面的地址空间，4个节点，每个节点能持有6个页面。

在这个例子中，如果CPU 0引用的指令或数据在页面0、2、5或9中，那么引用在本地完成。引用其他的页面会导致陷入。例如，对页面10的引用会导致陷入到DSM软件，该软件把页面10从节点1移到节点0，如图8-22b所示。

1. 复制

对基本系统的一个改进是复制那些只读页面，如程序代码、只读常量或其他只读数据结构，它可以明显地提高性能。举例来说，如果在图8-22中的页面10是一段程序代码，CPU 0对它的使用可以导致将一个副本送往CPU 0，从而不用打扰CPU 1的原有存储器，如图8-22c所示。在这种方式中，CPU 0和CPU 1两者可以按需要经常同时引用页面10，而不会产生由于引用不存在的存储器页面而导致的陷阱。

另一种可能是，不仅复制只读页面，而且复制所有的页面。只要有读操作在进行，实际上在只读页面的复制和可读写页面的复制之间不存在差别。但是，如果一个被复制的页面突然被修改了，就必须采取必要的措施来避免多个不一致的副本存在。如何避免不一致性将在下面几节中进行讨论。