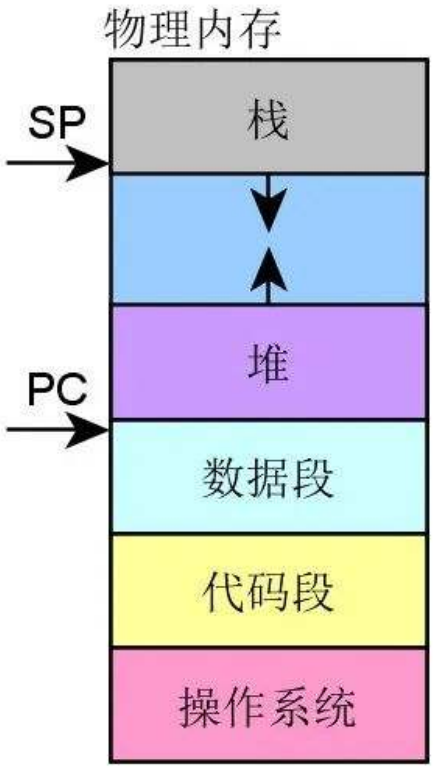
MMU 內存管理單元詳解

https://blog.csdn.net/p1279030826/article/details/105827355

MMU 誕生之前:

在傳統的批處理系統如 DOS 系統,應用程序與操作系統在內存中的佈局大致如下圖：



- 應用程序直接訪問物理內存，操作系統佔用一部分內存區。
- 操作系統的職責是「加載」應用程序，「運行」或「卸載」應用程序。

如果我們一直是單任務處理，則不會有任何問題，也或者應用程序所需的內存總是非常小，則這種架構是不會有任何問題的。然而隨著計算機科學技術的發展，所需解決的問題越來越複雜，單任務批處理已不能滿足需求了。而且應用程序需要的內存量也越來越大。而且伴隨著多任務同時處理的需求，這種技術架構已然不能滿足需求了，早先的多任務處理系統是怎麼運作的呢？

程序員將應用程序分段加載執行，但是分段是一個苦力活。而且死板枯燥。此時聰明的計算機科學家想到了好辦法，提出來虛擬內存的思想。程序所需的內存可以遠超物理內存的大小，將當前需要執行的留在內存中，而不需要執行的部分留在磁盤中，這樣同時就可以滿足多應用程序同時駐留內存能並發執行了。

從總體上而言，需要實現哪些大的策略呢？

- 所有的應用程序能同時駐留內存，並由操作系統調度並發執行。需要提供機制管理 I/O 重疊，CPU 資源競爭訪問。
- 虛實內存映射及交換管理，可以將真實的物理內存，有可變或固定的分區，分頁或者分段與虛擬內存建立交換映射關係，並且有效的管理這種映射，實現交換管理。

這樣，衍生而來的一些實現上的更具體的需求：

- **競爭訪問保護管理需求**：需要嚴格的訪問保護，動態管理哪些內存頁/段或區，為哪些應用程序所用。這屬於資源的競爭訪問管理需求。
- **高效的翻譯轉換管理需求**：需要實現快速高效的映射翻譯轉換，否則系統的運行效率將會低下。
- **高效的虛實內存交換需求**：需要在實際的虛擬內存與物理內存進行內存頁/段交換過程中快速高效。

總之，在這樣的背景下，MMU 應運而生，也由此可見，任何一項技術的發展壯大，都必然是需求驅動的。這是技術本身發展的客觀規律。

內存管理的好處

- 為編程提供方便統一的內存空間抽象，在應用開發而言，好似都完全擁有各自獨立的用戶內存空間的訪問權限，這樣隱藏了底層實現細節，提供了統一可移植用戶抽象。
- 以最小的開銷換取性能最大化，利用 MMU 管理內存肯定不如直接對內存進行訪問效率高，為什麼需要用這樣的機制進行內存管理，是因為並發進程每個進程都擁有完整且相互獨立的內存空間。那麼實際上內存是昂貴的，即使內存成本遠比從前便宜，但是應用進程對內存的尋求仍然無法在實際硬件中，設計足夠大的內存實現直接訪問，即使能滿足，CPU 利用地址總線直接尋址空間也是有限的。

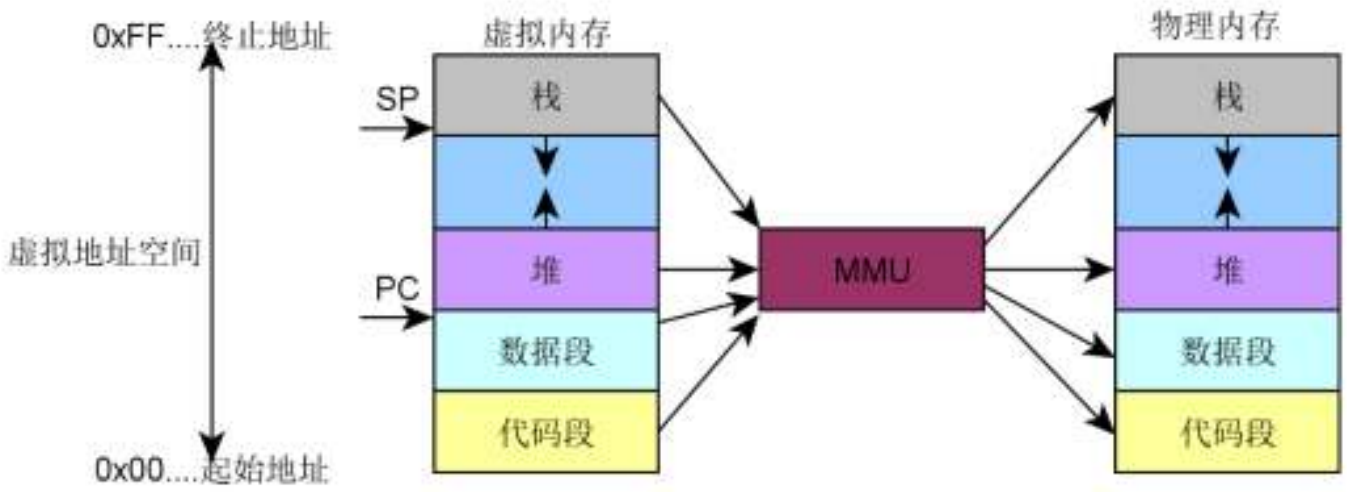
內存管理實現總體策略

從操作系統角度來看，虛擬內存的基本抽象由操作系統實現完成：

- 處理器內存空間不必與真實的所連接的物理內存空間一致。

- 當應用程序請求訪問內存時，操作系統將虛擬內存地址翻譯成物理內存地址，然後完成訪問。

從應用程序角度來看，應用程序（往往是進程）所使用的地址是虛擬內存地址，從概念上就如下示意圖所示，MMU 在**操作系統**的控制下負責將虛擬內存實際翻譯成物理內存。



從而這樣的機制，虛擬內存使得應用程序不用將其全部內容都一次性駐留在內存中執行：

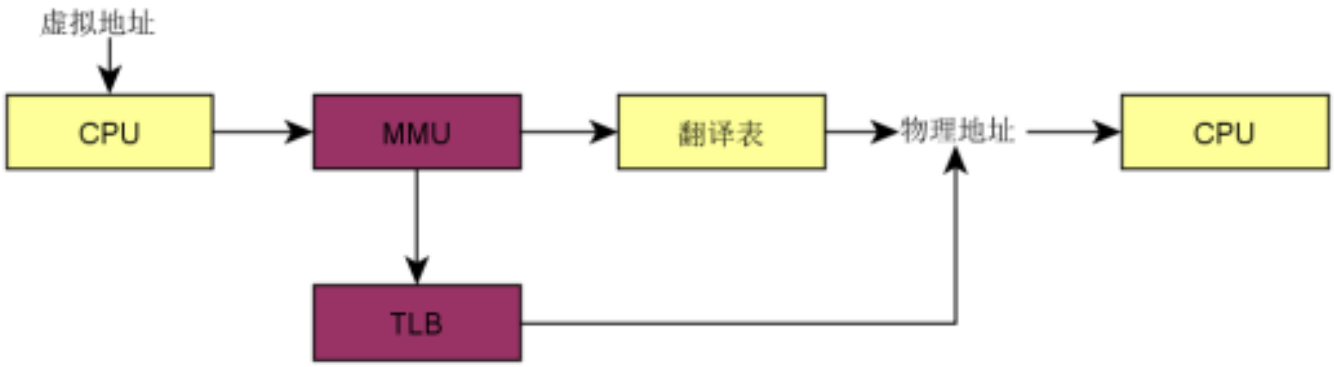
- **節省內存**：很多應用程序都不必讓其全部內容一次性加載駐留在內存中，那麼這樣的好處是顯而易見，即使硬件系統配置多大的內存，內存在系統中仍然是最為珍貴的資源。所以這種技術節省內存的好處是顯而易見的。
- **使得應用程序以及操作系統更具靈活性**。
    - 操作系統根據應用程序的動態運行時行為靈活的分配內存給應用程序。
    - 使得應用程序可以使用比實際物理內存多或少的內存空間。

MMU 以及 TLB

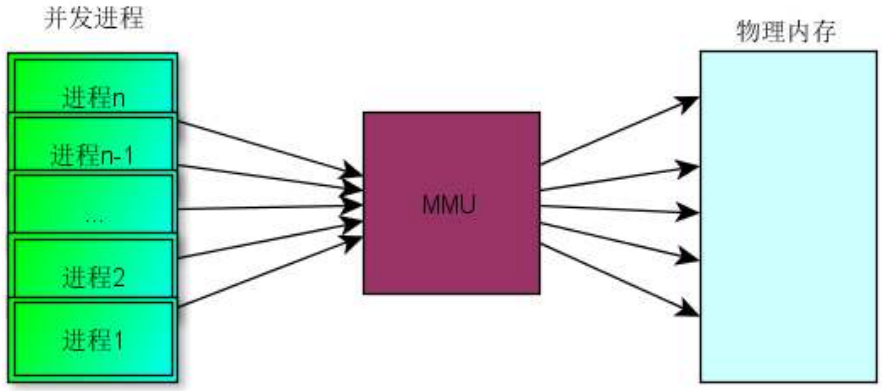
MMU(Memory Management Unit)內存管理單元：

- 一種硬件電路單元負責將虛擬內存地址轉換為物理內存地址
- 所有的內存訪問都將通過 MMU 進行轉換，除非沒有使能 MMU。

TLB(Translation Lookaside Bu・er)轉譯後備緩衝器: 本質上是 MMU 用於虛擬地址到物理地址轉換表的緩存



這樣一種架構，其最終運行時目的，是為主要滿足下面這樣運行需求：



多進程並發同時並發運行在實際物理內存空間中，而 `MMU` 充當了一個至關重要的虛擬內存到物理內存的橋樑作用。

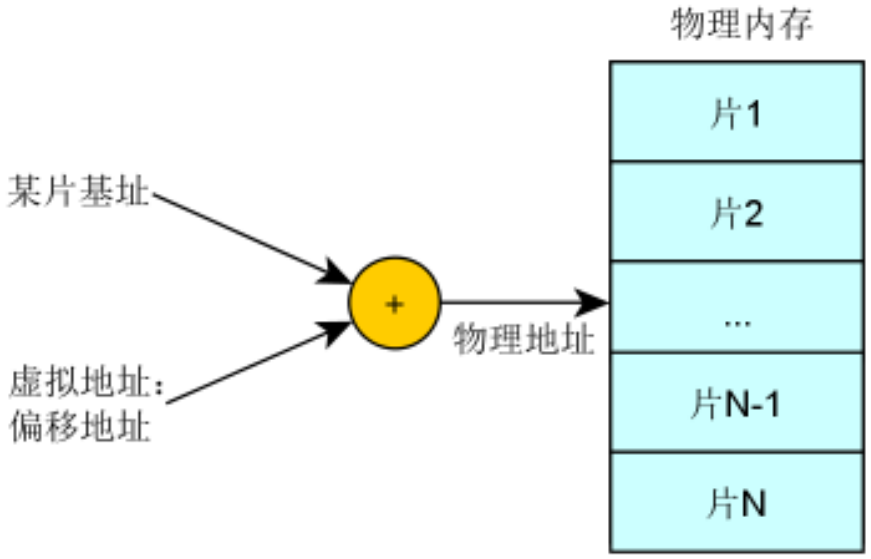
那麼，這種框架具體從高層級的概念上是怎麼做到的呢？事實上，是將物理內存採用分片管理的策略來實現的，那麼，從實現的角度將有兩種可選的策略：

- 固定大小分區機制
- 可變大小分區機制

固定大小區片機制

通過這樣一種概念上的策略，將物理內存分成固定等大小的片：

- 每一個片提供一個基地址
- 實際尋址，物理地址=某片基址+虛擬地址
- 片基址由操作系統在進程動態運行時動態加載



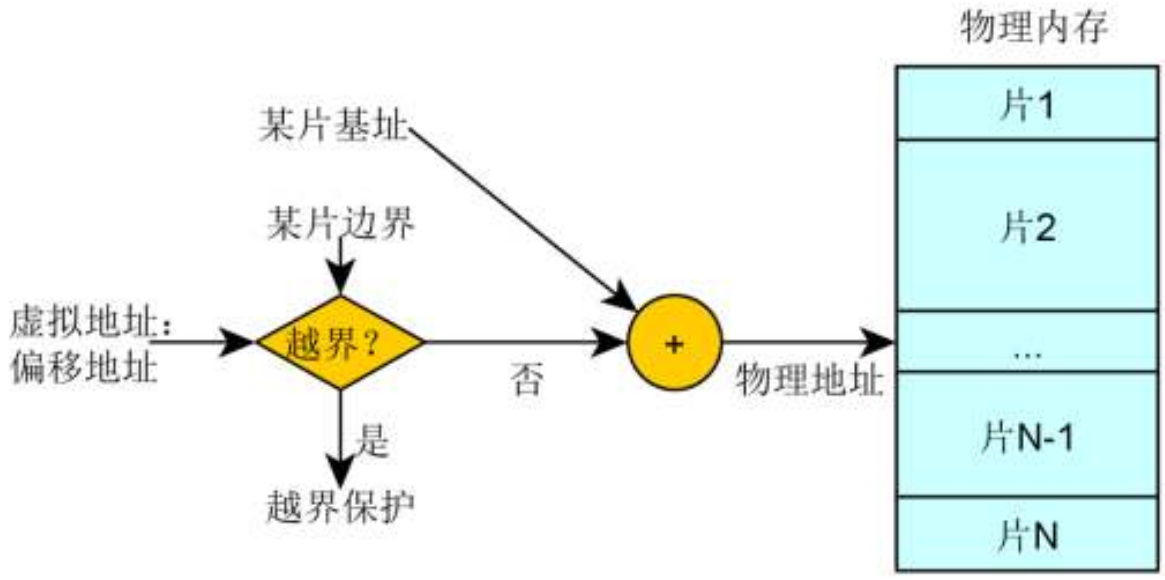
這種策略實現，其優勢在於簡易，切換快速。但是該策略也帶來明顯的劣勢：

- 內部碎片：一個進程不使用的分區中的內存對其他進程而言無法使用
- 一種分區大小並不能滿足所有應用進程所需。

可變大小分區機制

內存被劃分為可變大小的區塊進行映射交換管理：

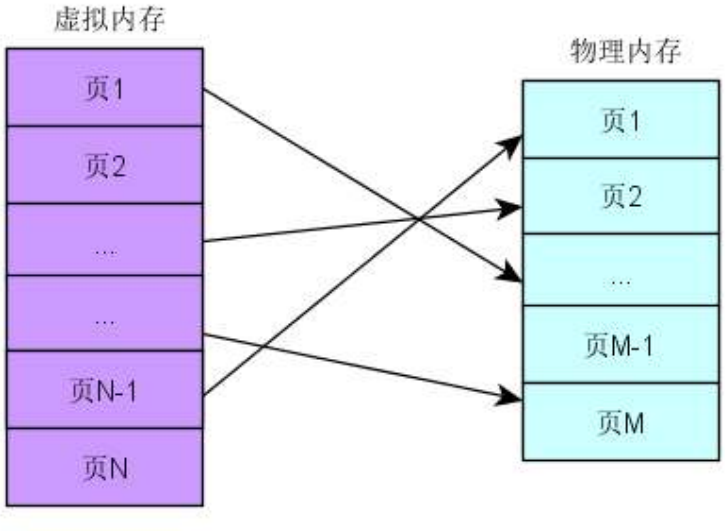
- 需要提供基址以及可變大小邊界，可變大小邊界用於越界保護。
- 實際尋址，物理地址=某片基址+虛擬地址



那麼這種策略其優勢在於沒有內部內存碎片，分配剛好夠進程所需的大小。但是劣勢在於，在加載和卸載的動態過程中會產生碎片。

分頁機制

分頁機制採用在虛擬內存空間以及物理內存空間都使用固定大小的分區進行映射管理。

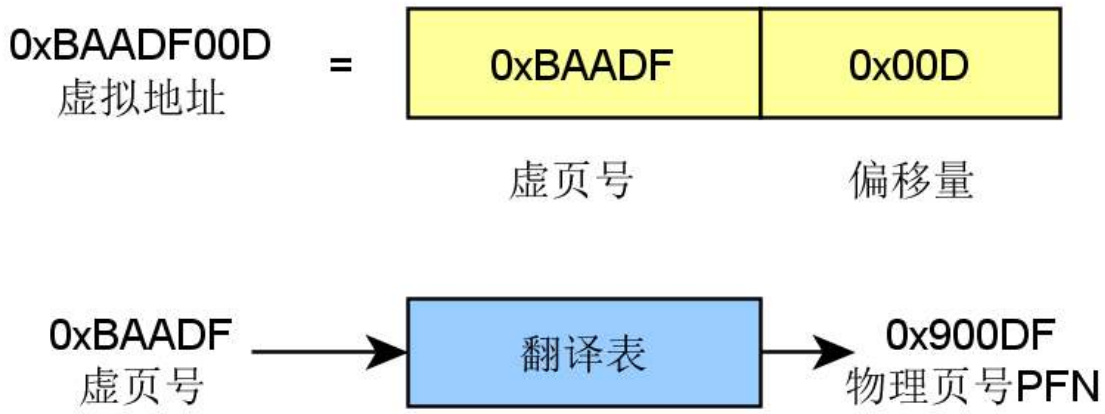


- 從應用程序(進程)角度看內存是連續的 `0-N` 的分頁的虛擬地址空間。
- 物理內存角度看，內存頁是分散在整個物理存儲中
- 這種映射關係對應用程序不可見，隱藏了實現細節。

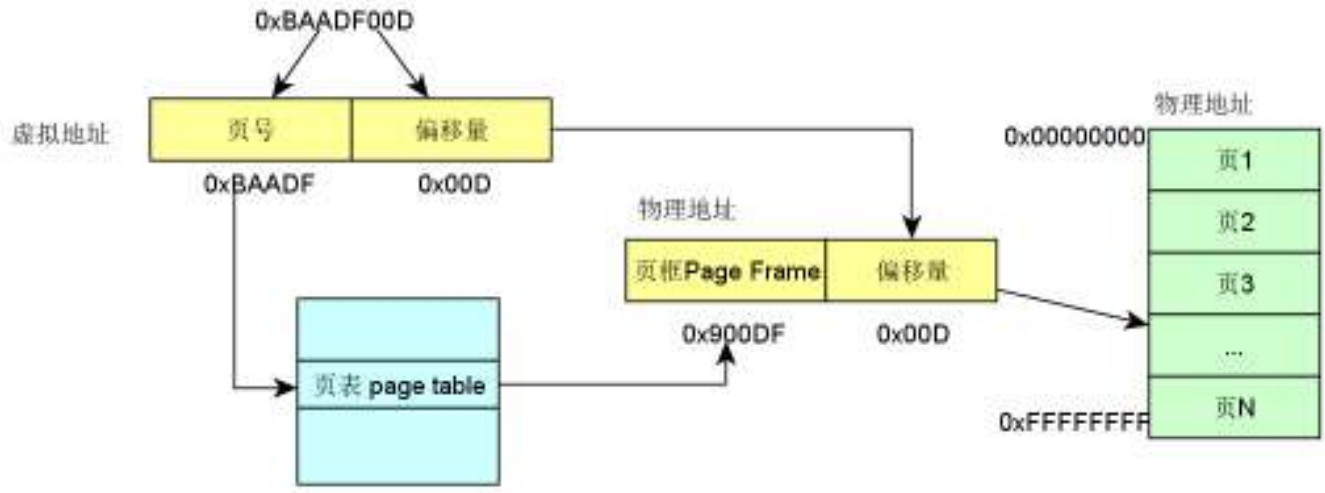
分頁機制是如何尋址的呢？這裡介紹的設計理念，具體的處理器實現各有細微差異：

- 虛擬地址包含了兩個部分：**虛擬頁序號 VPN（virtual paging number)以及偏移量**
- **虛擬頁序號 VPN** 是**頁表（Page Table）**的索引
- **頁表（Page Table）維護了頁框號(Page frame number PFN)**
- 物理地址由 **PFN::Offset** 進行解析。

舉個栗子，如下圖所示：



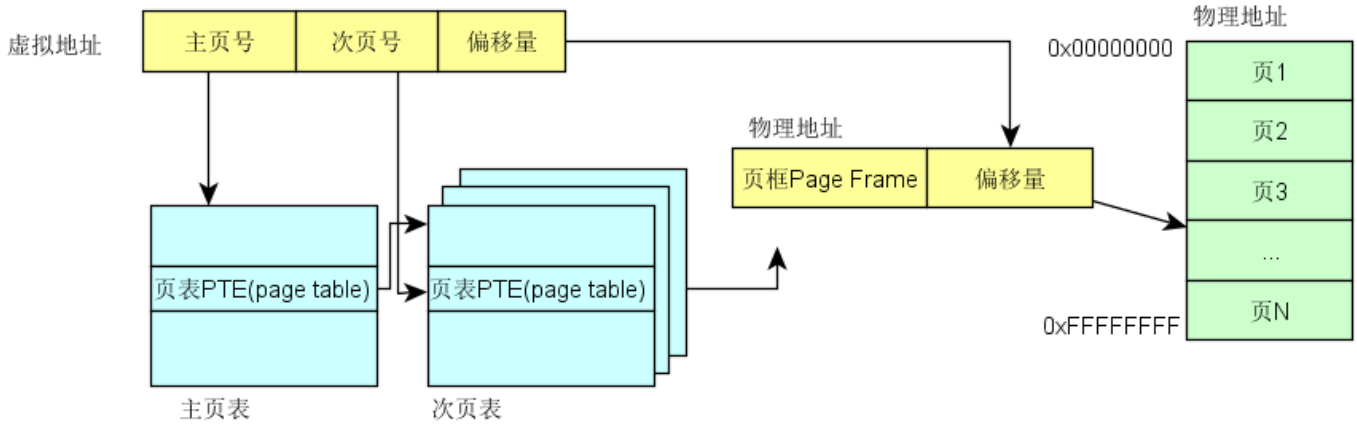
還沒有查到具體的物理地址，憋急，再看一下完整解析示例：



如何管理頁表

對於 32 位地址空間而言，假定 4K 為分頁大小，則頁表的大小為 100MB，這對於頁表的查詢而言是一個很大的開銷。那麼如何減小這種開銷呢？實際運行過程中發現，事實上只需要映射實際使用的很小一部分地址空間。那麼在一級頁機制基礎上，延伸出多級頁表機制。

以二級分頁機製為例：



單級頁表已然有不小的開銷，查詢頁表以及取數，而二級分頁機制，因為需要查詢兩次頁表，則將這種開銷再加一倍。那麼如何提高效率呢？其實前面提到一個概念一直還沒有深入描述 TLB,將翻譯工作由硬件緩存 cache,這就是 TLB 存在的意義。

- TLB 將虛擬頁翻譯成 PTE，這個工作可在單週期指令完成。
- TLB 由硬件實現
  - 完全關聯緩存(並行查找所有條目)
  - 緩存索引是虛擬頁碼
  - 緩存內容是 PTE
  - 則由 PTE+offset，可直接計算出物理地址

TLB 加載

誰負責加載 TLB 呢？這裡可供選擇的有兩種策略：

- 由操作系統加載，操作系統找到對應的 PTE，而後加載到 TLB。格式比較靈活。
- MMU 硬件負責，由操作系統維護頁表，MMU 直接訪問頁表，頁表格式嚴格依賴硬件設計格式。

總結一下

從計算機大致發展歷程來瞭解內存管理的大致發展策略，如何衍生出 MMU，以及固定分片管理、可變分片管理等不同機制的差異，最後衍生出單級分頁管理機制、多級分頁管理機制、TLB 的作用。從概念上相對比較易懂的角度描述了 MMU 的誕生、機制，而忽略了處理器的具體實現細節。作為從概念上更深入的理解 MMU 的工作機理的角度，還是不失為一篇淺顯易懂的文章。