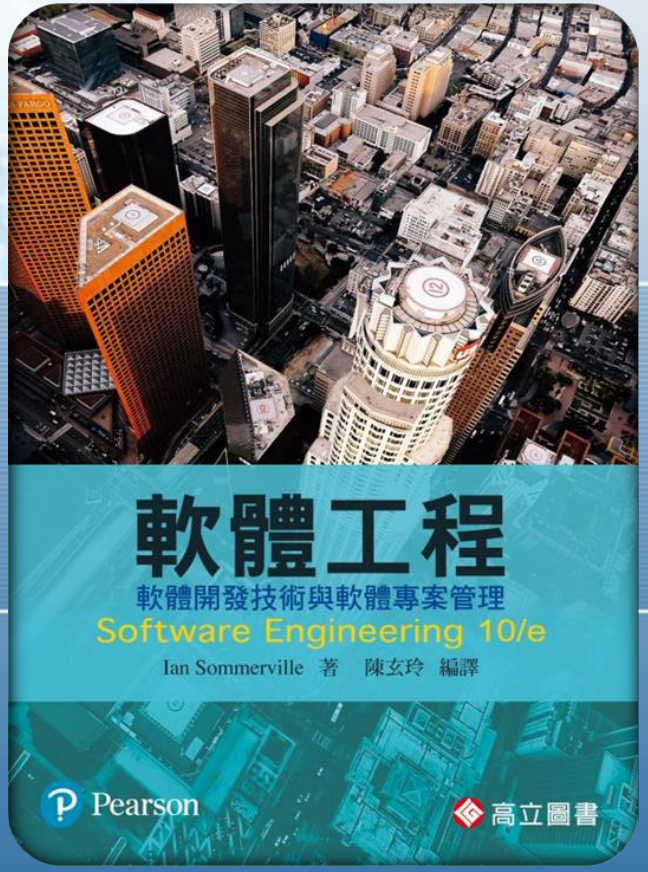
Chapter 11

可靠工程



本章內容

- 我們期望軟體會一直運作，不會發生當機或故障，而且會保存好我們的資料和個人資訊。我們需要能信任我們使用的軟體，意思就是軟體必須是可靠的。
- 在那些由軟體擔負特別關鍵責任的案例，例如在飛機或國家基礎建設中，可能會使用特殊的可靠工程 (reliability engineering) 技術，以達到要求的高標準可靠性 (reliability) 和可用性 (availability)。

- Brian Randell定義所謂的「fault-error-failure」模型 (Randell 2000)，所根據的觀念是由於人為疏失 (human error) 造成缺失 (fault)，缺失會造成錯誤 (error)，而因為錯誤導致系統故障 (failure)。

  1. **人為疏失或失誤 (human error or mistake)**：因為人的行為造成將缺失引進系統。比如說，假設荒野測候系統的某個程式設計人員決定，計算下次傳輸資料的時間是把現在的時間加1小時，這樣做在大部分時候都沒有問題，除了傳輸時間剛好落在23.00到午夜之間的時候 (在24小時制午夜是00.00)。

2. **系統缺失 (system fault)**：軟體系統中某個會導致系統發生錯誤的特性。上例的缺失是因為含有將名稱是Transmission_time的變數加1，而沒有檢查Transmission_time變數的值是否大於或等於23.00的程式碼。
3. **系統錯誤 (system error)**：執行期間錯誤的系統狀態，會導致系統行為不符合使用者期望。此例在執行到缺失程式碼時，將Transmission_time變數的值設錯了 (設定成24.XX而不是00.XX)。
4. **系統故障 (system failure)**：在某個時間點發生的事件，造成系統提供的服務不符合使用者期望。在此例是因為時間值無效，所以沒有傳輸任何氣候資料。

- 系統缺失不一定會造成系統錯誤，而系統錯誤也不一定會造成系統故障。
  1. 程式中並不是所有的程式碼都會被執行到。
  2. 錯誤可能只是暫時的。
  3. 系統可能具有缺失偵測和保護機制，能夠發現這些錯誤行為，並且在系統服務受影響前修正回來。
- 系統缺失不一定會造成系統故障的另一個原因是，使用者會調整自己的行為，避免使用他們已知將引發程式故障的輸入。

- 依據缺失、錯誤和故障這三者的區分，產生出3種互補的方法：
  1. **避免缺失 (fault avoidance)**：軟體的設計與實作程序，應該使用能協助避免設計和程式編寫錯誤，以及儘量減少引進缺失到程式內的軟體開發方法。
  2. **偵測並改正缺失 (fault detection and correction)**：設計一些驗證與確認程序，目的是在系統開始部署正式上線前發現缺失並移除缺失。
  3. **容忍缺失 (fault tolerance，長久以來一般簡稱容錯)**：系統應該設計成在執行期間能夠偵測與管理系統的缺失或非預期行為，讓系統故障不會發生。

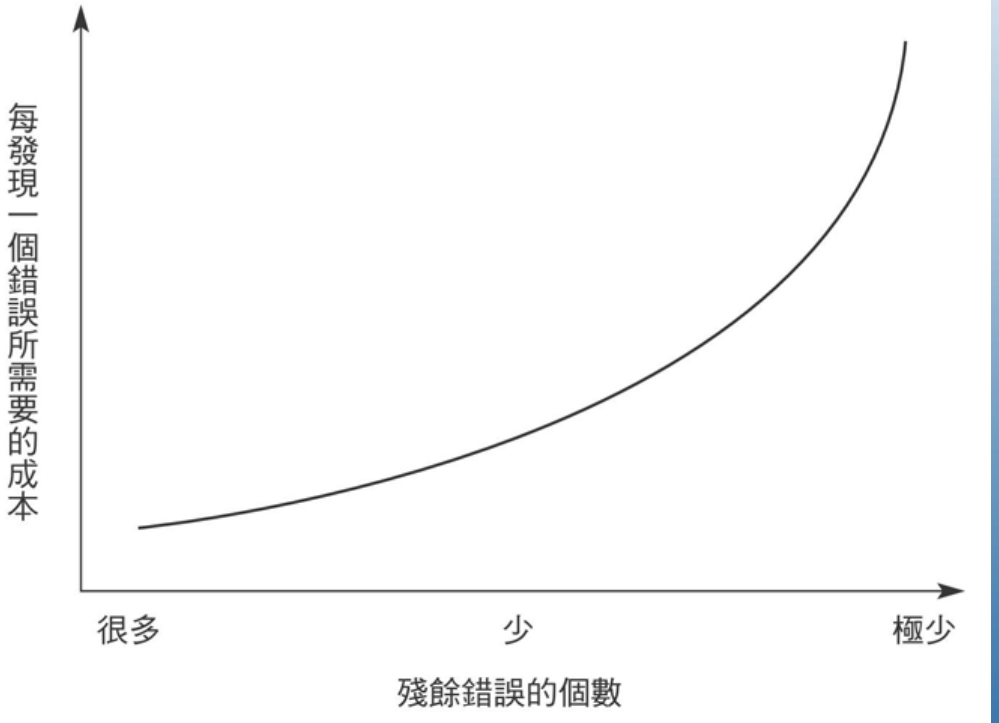


圖 11.1 移除殘餘的錯誤所需要的成本逐漸增加

11.1 可用性與可靠性

- 可用性和可靠性兩者都可以用數值機率來表示。如果可用性等於0.999，這表示經過一段時間的執行，這段期間有99.9%的時候系統都是可用的。

1. **可靠性 (reliability)**：在某段時間內，在某個限定環境中，針對特定目的而執行的運作不發生故障的機率。
2. **可用性 (availability)**：系統在某個時間點能夠運作和提供所要求服務的機率。

11.1 可用性與可靠性

- 系統的可靠性並不是個絕對值，要看系統在哪裡和如何使用而定。比如說，假設在辦公室環境下評估某套應用軟體的可靠性，該環境中多數使用者對軟體的內部運作不感興趣，因此可能會覺得這套系統的可靠性很高。但如果是在大學環境中評估這套系統的可靠性，結果可能就大不相同。

11.1 可用性與可靠性

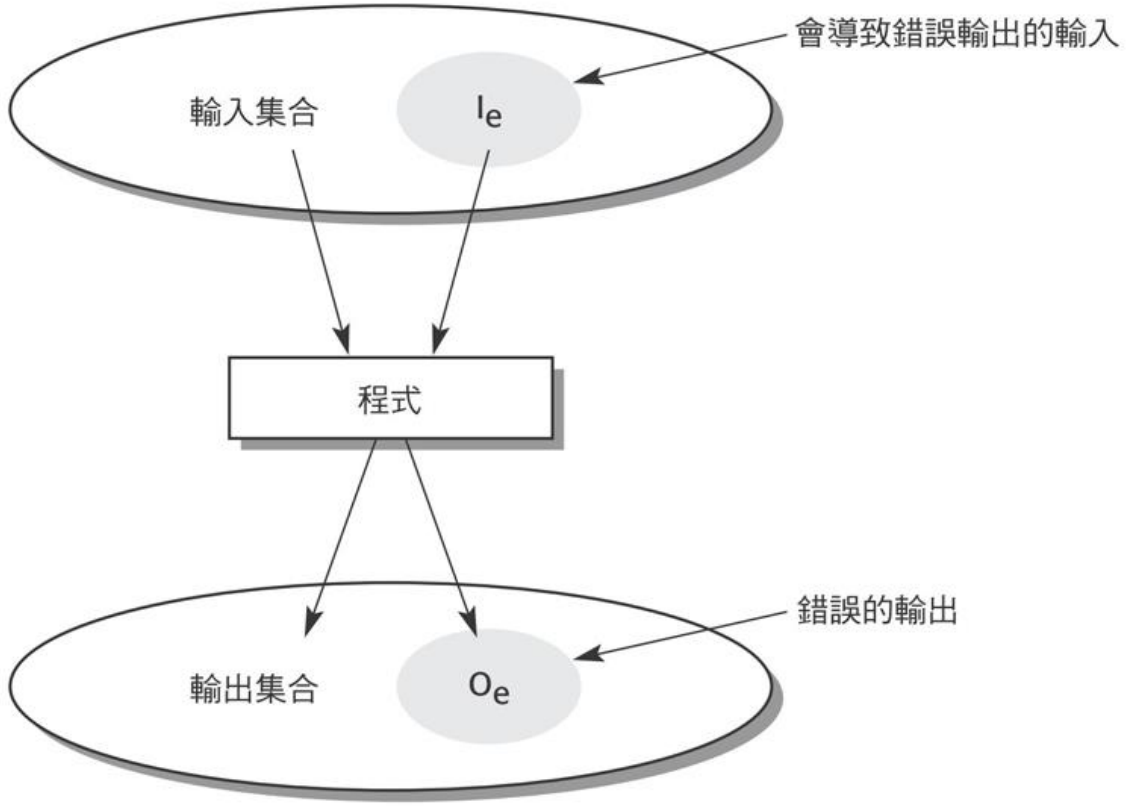


圖 11.2　輸入資訊與輸出資訊的對應關係

11.1 可用性與可靠性

- 大部分的輸入不會導致系統故障，但有些輸入或輸入的組合(如圖11.2中的陰影橢圓形 I_e)可能會導致系統故障或產生錯誤的輸出結果。程式的可靠性與程式在執行時，是否剛好有涵蓋這些錯誤輸入的機率有關。
- 例如有某個會影響可靠性的缺失，也許有些使用者的工作模式會經常執行到，而其他使用者的工作模式可能就不會碰到。

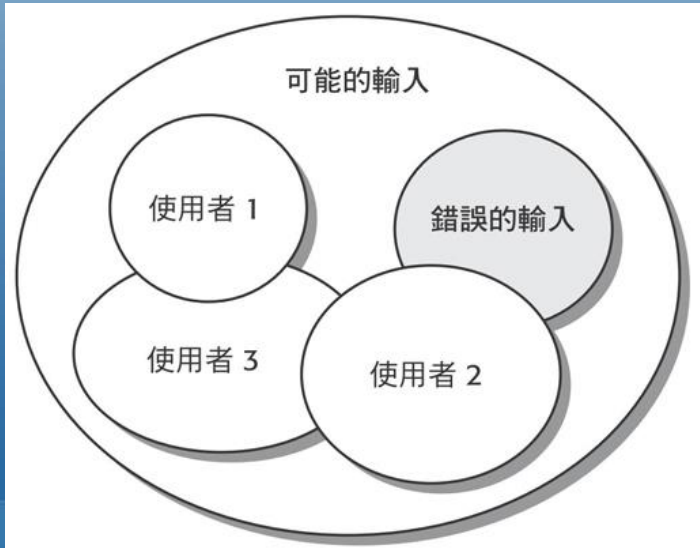


圖 11.3　軟體的使用模式

11.1 可用性與可靠性

- 系統的可用性不只根據系統本身故障的次數而定，它還包括修復會引起系統故障的各個缺失所需的時間。
- 因為系統無法使用而導致的服務中斷，並沒有顯示在簡單的可用性度量方式中，因為它只顯示出系統可使用的時間比例。其實系統發生故障的時間點也很重要。
- 雖然可用性和可靠性的關係密切，但有時某一邊會比另一邊重要。
- 例如電信局的電話交換系統，就是可用性比可靠性重要的一個例子。

11.2 可靠性需求

- 系統的可信賴度不能只依靠好的程式設計，還必須注意系統規格制訂時的細節，並且納入專門用來確保系統可信賴度的軟體需求。

1. **功能需求**：定義系統中應具備的檢查與回復功能，以及能保護系統避免發生故障和抵禦外在攻擊的保護功能。
2. **非功能需求**：用來定義系統必要的可靠性和可用性。

11.2.1 可靠性的測量值

- 可靠性可以想成是系統在特定作業環境下使用，發生系統故障的機率。

1. **服務故障率 (Probability of failure on demand, POFOD)**：這個測量值是定義對系統提出的服務要求會導致系統故障的機率。
2. **故障發生率 (Rate of occurrence of failures, ROCOF)**：這個測量值用來表示系統可能的故障次數除以某一段時間 (如1小時) ，或除以系統執行次數。
3. **可用率 (AVAIL)**：系統在有服務要求提出時是正常運作的機率。因此，假設可用率等於0.9999，表示系統運作時間有99.99% 可以使用。

11.2.1 可靠性的測量值

可用性	說明
0.9	系統有 90% 的時間是可用的；也就是說，每 24 小時（1440 分鐘）系統將有 144 分鐘無法使用
0.99	系統每 24 小時有 14.4 分鐘無法使用
0.999	系統每 24 小時有 84 秒無法使用
0.9999	系統每 24 小時有 8.4 秒無法使用，大約是每星期 1 分鐘

圖 11.4 可用性規格

11.2.1 可靠性的測量值

- 當失敗發生可能會導致嚴重的系統故障時，應該使用POFOD當作可靠性測量指標。舉例來說，有個監視化學反應爐的保護系統，會在反應爐過熱時關閉反應動作，則它的可靠性應該用POFOD來指定。
- 當對系統的要求是規律發生而不是間歇性時，ROCOF是最適合使用的指標。比如說，假設有個系統負責處理大量的交易，而指定的ROCOF等於每天10次失敗；也就是說，可以接受平均每天有10筆交易沒有成功完成而必須取消和重做。或者是你也可以指定ROCOF等於每1000筆交易的失敗交易個數。

11.2.1 可靠性的測量值

- 假如兩次故障發生之間的絕對時間很重要，也許該使用MTTF來指定可靠性。舉例而言，假如系統的每筆異動時間很長(例如電腦輔助設計系統)，那麼它的可靠性應該指定為平均經過很長時間才發生一次失敗，也就是MTTF應該比使用者在製作模型文件時尚未存檔的平均時間長很多。

11.2.2 可靠性的非功能需求

- 可靠性的非功能 (non-functional) 需求是系統要求的可靠性和可用性的量化規格，使用前一節說明的指標 (POFOD、ROCOF或AVAIL) 其中一個來計算。量化的可靠性和可用性規格，在安全關鍵系統上已經使用多年，不過很少用在商業關鍵系統上。然而，隨著有越來越多公司要求自己的系統提供24/7的服務，他們也會對可靠性和可用性的期望更明確。

11.2.2 可靠性的非功能需求

- 量化的可靠性規格可用在下列這些方面：
  1. 在決定要可靠性到達什麼水準的過程中，有助於利害關係人釐清他們真正要的是什麼。
  2. 它提供評估何時該停止測試系統的參考基準。當系統已經達到要求的可靠性基準時，即可停止測試。
  3. 它是一種評估各種改善系統可靠性的不同設計策略的方法。這有助於判斷哪個策略可以達到要求的可靠性。
  4. 假設系統在上線服務前需要監管機關核可（例如對飛航安全有關鍵影響的系統），那麼能夠證明系統有達到要求的可靠性的相關證據，對於能否取得認證會很重要。

11.2.2 可靠性的非功能需求

1. 先替各種類型的故障分別設定它的可用性和可靠性的需求，高成本故障的機率應該要比後果輕微的故障低才對。
2. 再分別針對不同服務，設定它的可用性和可靠性的需求。最關鍵的服務應該要求最高的可靠性，但是也許你願意容忍非關鍵服務的故障次數多一點沒關係。
3. 思考你是否真的需要很高的可靠性。

11.2.2 可靠性的非功能需求

- 以銀行ATM系統的可靠性和可用性的需求為例。
  1. 要確保它們有確實根據客戶的要求提供服務，而且有適當的將客戶的交易紀錄在帳戶資料庫中。
  2. 要確保這些系統在需要時都可使用。
- ATM機器所要求的可靠性絕對值可能不必很高，每天有幾次故障是可接受的。
- 對銀行(和客戶)而言，ATM網路的可用性比個別的ATM交易有沒有失敗更重要。

11.2.2 可靠性的非功能需求

- 要設定ATM網路的可用性，你應該先找出系統服務，並設定每個服務要求的可用性，尤其是：
  - 客戶的帳戶資料庫服務
  - ATM提供的個人服務如「提款」、「帳戶資訊」等
- 資料庫服務在此是最關鍵的，因為如果這個服務故障，網路上所有ATM機器就全都停擺，因此應該指定這個服務的可用性較高。
- 這個例子在早上7點到晚上11點之間，可接受的資料庫可用性數字(忽略定期維修和升級因素)可能是大約0.9999。這表示每星期停機時間小於1分鐘。

11.2.2 可靠性的非功能需求

- 對於個別ATM機器，它的整體可用性要看機械可靠性和現鈔是否不夠，軟體問題的影響比較小。
- 因此ATM軟體的可用性低一點是可接受的，例如0.999，相當於每天有1到2分鐘這台機器無法使用。
- 控制系統的可靠性通常會以POFOD來指定，也就是發出服務要求時系統故障的機率。以胰島素給藥系統中控制軟體的可靠性規格為例。

11.2.2 可靠性的非功能需求

1. **暫時性軟體故障 (transient software failure)**：可由使用者自行修復的故障，例如重新設定或校正機器。這類故障的可接受POFOD值很低 (如0.002)。這表示對機器的每500次要求中只能發生一次故障。
2. **永久性軟體故障 (permanent software failure)**：需要廠商重新安裝軟體的故障。最多每年只能有一次，因此POFOD應該是不能超過0.00002。

11.2.3 可靠性的功能規格

- 系統可靠性的功能需求分成下列4種：
  1. **檢查功能需求 (checking requirement)**：這類需求制訂如何檢查系統的輸入，以確保不正確和超出範圍的輸入值，在系統處理它們之前就能夠先被偵測到。
  2. **回復功能需求 (recovery requirement)**：這類需求是針對協助系統在故障發生後回復。
  3. **重複功能需求 (redundancy requirement)**：這是指定系統的重複功能，以確保單一元件故障不會造成完全無法提供服務。
  4. **程序需求 (process requirement)**：這些是避免缺失的需求，目的是確保在開發程序中會使用良好的實務經驗。

11.2.3 可靠性的功能規格

RR1：操作人員可能輸入的所有輸入值，都必須事先定義數值範圍。系統也應該檢查所有輸入值是否符合定義的範圍。(檢查功能)
RR2：病患資料庫的副本應該保存在兩台分開的伺服器上，而且這兩台伺服器不能在同一棟大樓。(回復、重複功能)
RR3：煞車控制系統在實作時應該使用 N 版本程式設計 (N-version programming) 技術。(重複功能)
RR4：系統必須以 Ada 的安全子集來實作，並且使用靜態分析方式來檢查。(程序)

圖 11.5 可靠性的功能需求範例

11.3 容錯架構

- 容錯做法是讓系統內建即使已經發生軟硬體故障而且系統狀態錯誤後，仍然能讓系統繼續運作的一套機制。
- 對於要求高安全性或保全性的系統，容錯是必要的，這類系統不能在一偵測到錯誤，就讓系統進入安全狀態下而停擺。
- 為了提供容錯功能，系統架構的設計必須內建重複且多樣化的硬體和軟體。例如飛機上的系統就可能需要容錯架構，因為飛機在整個飛行期間，它的通訊系統和重要的命令與控制系統都必須保持運作。

11.3.1 保護系統

- 所謂的保護系統 (protection system) 是一種與某個其他系統有關聯的特殊系統。它通常是某種程序的控制系統 (如化學製造程序) 或儀器控制系統 (如無人駕駛列車上的系統)。
- 例如列車上的保護系統可能負責偵測列車是否快到紅燈，而列車控制系統又沒有減速，則保護系統會自動開始煞車將列車停住。保護系統會獨立的監測它們的環境，如果有感應器發現問題而且控制系統沒有正在處理，那麼保護系統會被啟動將處理器或設備關閉。

11.3.1 保護系統

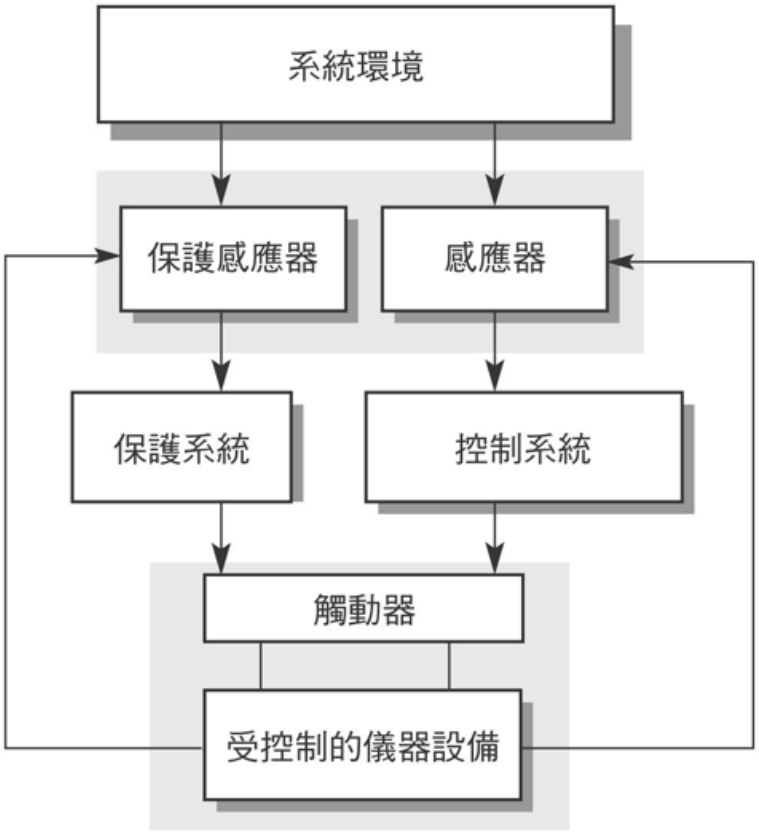


圖 11.6　保護系統的架構

11.3.1 保護系統

- 這種架構的優點是保護系統的軟體，比控制受保護程序的軟體簡單得多。保護系統的唯一功能就是監測運作情況，並確保系統在發生緊急事件時能夠被帶到安全狀態。
- 這樣做的目的是確保保護系統的可靠性，也就是讓它的POFOD非常低 (如0.001)。

11.3.2 自我監測架構

- 自我監測 (self-monitoring) 架構是一種系統架構，這種系統是設計成會監測自己的運作，並且在偵測到問題時採取某些行動 (圖11.7)。
- 做法是在分開的管道 (channel) 上執行計算，然後再比較這些計算的輸出。假如輸出結果同時都完成而且完全相同，那麼表示系統有正確運作。如果輸出不同，就假定有發生故障。

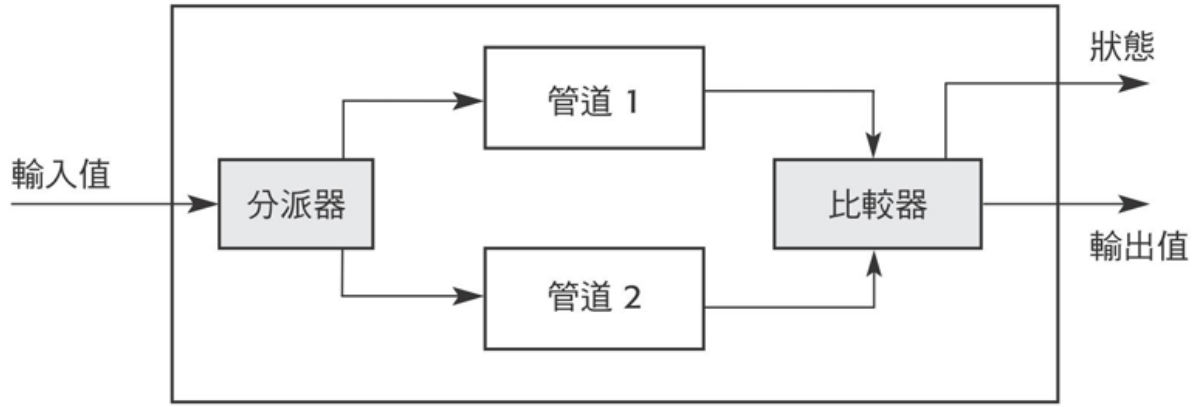


圖 11.7　自我監測架構

11.3.2 自我監測架構

- 自我監測系統必須這樣設計：
  1. 在每個管道使用的硬體應該要多樣化。能降低共同的處理器設計缺失影響到計算結果的機率。
  2. 在每個管道使用的軟體應該要多樣化，否則可能會同時在每個管道發生相同的軟體錯誤。
- 若是要求高可用性的情況，就必須平行使用好幾個自我檢查系統。
- 例如Airbus 340系列飛機的飛航控制系統就使用這種方式，它一共使用5個自我檢查電腦。圖11.8是Airbus飛航控制系統的簡化圖。

11.3.2 自我監測架構

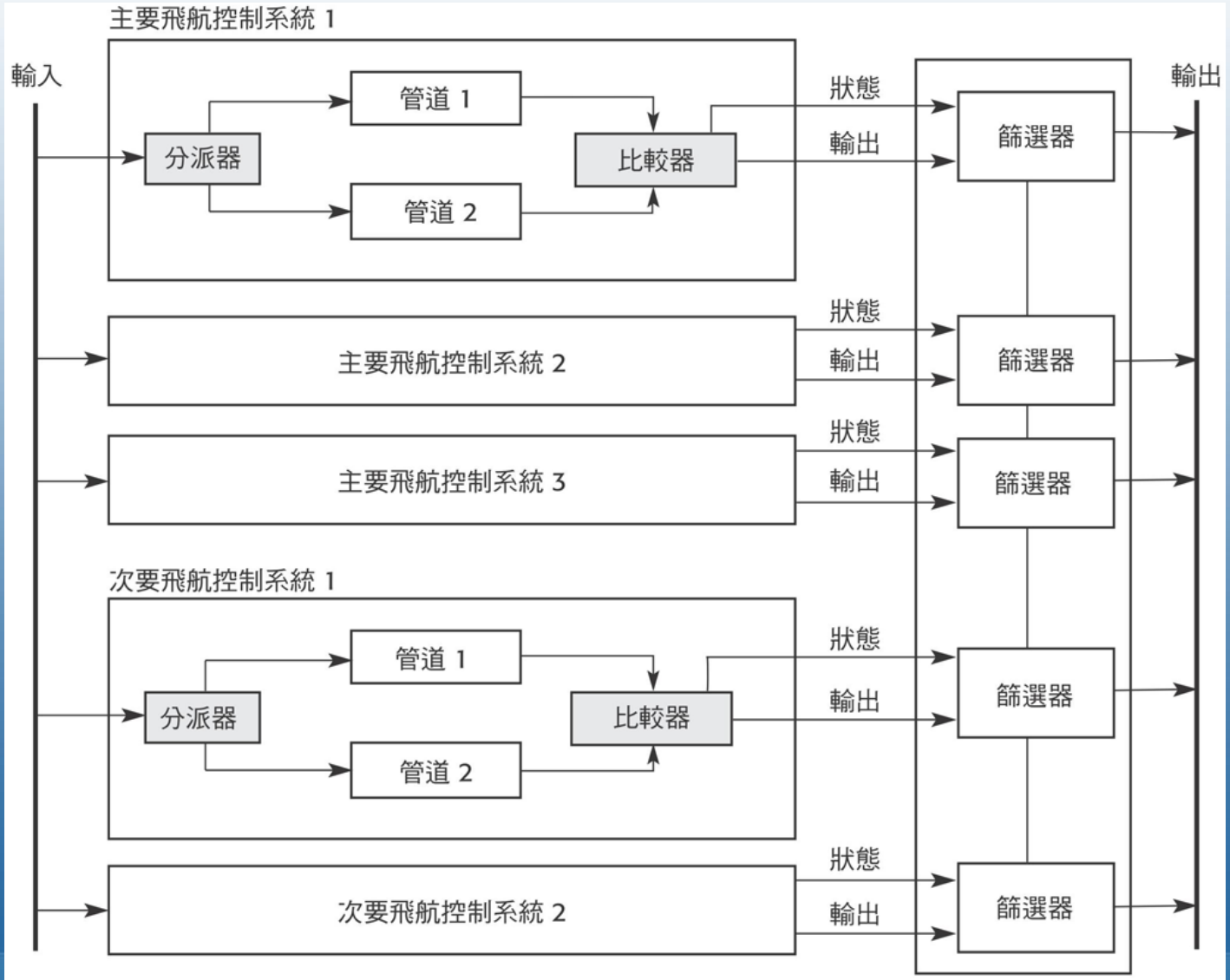


圖 11.8 Airbus 飛航控制系統的架構

11.3.2 自我監測架構

- Airbus系統的設計人員以多種不同方式嘗試達到多樣性：
  1. 主要飛航控制電腦使用的處理器與次要飛航控制系統不同。
  2. 主要和次要系統中的每個管道，所使用的晶片組是由不同的製造商供應。
  3. 次要飛航控制系統的軟體只提供關鍵功能，所以不像主要軟體那麼複雜。
  4. 主要和次要系統中每個管道的軟體，是由不同團隊使用不同的程式語言開發的。
  5. 次要和主要系統使用不同的程式語言。

11.3.3 N版本程式設計

- 多版本程式設計 (multiversion programming) 的觀念是衍生自硬體系統，它們的「三重模組備援」(triple modular redundancy, TMR) 的觀念已經使用多年，用來建構能容忍硬體故障的系統 (圖11.9)。

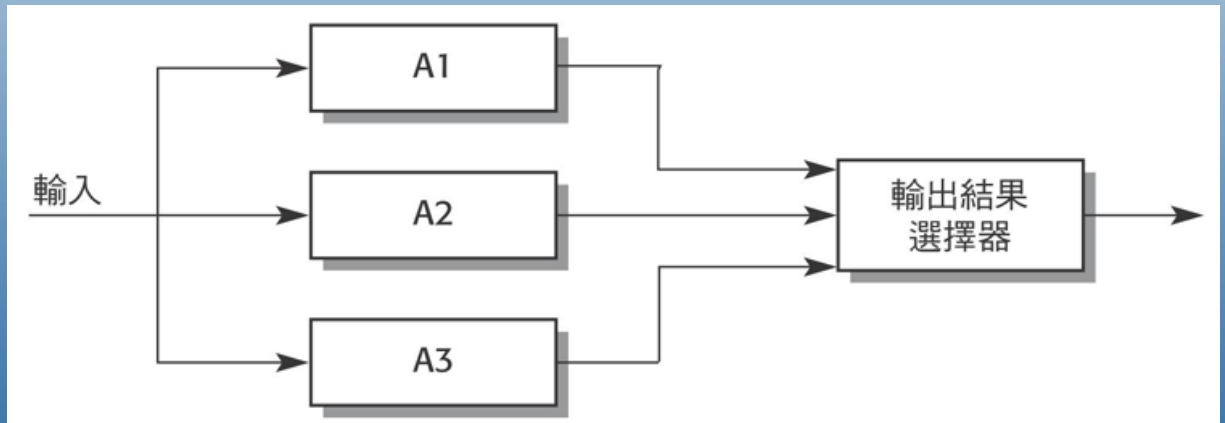


圖 11.9　三重模組備援 (TMR) 的觀念

11.3.3 N版本程式設計

- 在TMR系統中，硬體單元會被複製3份 (有時候更多)，每個單元的輸出都會被傳到輸出結果比較器 (comparator) 進行比較，它的實作就像是投票系統。此系統會比較所有的輸出，假如其中兩個或更多個相同，那麼那個值就是輸出。如果其中一個單元失敗，而無法產生和其他單元相同的輸出，便忽略它的輸出值。
- 如果採用不同製造商根據一個共用規格設計與製造出來的硬體單元，就可以減少這種故障發生的機會。這是假設不同團隊犯下相同的設計或製造錯誤的機率是非常低的。

11.3.3 N版本程式設計

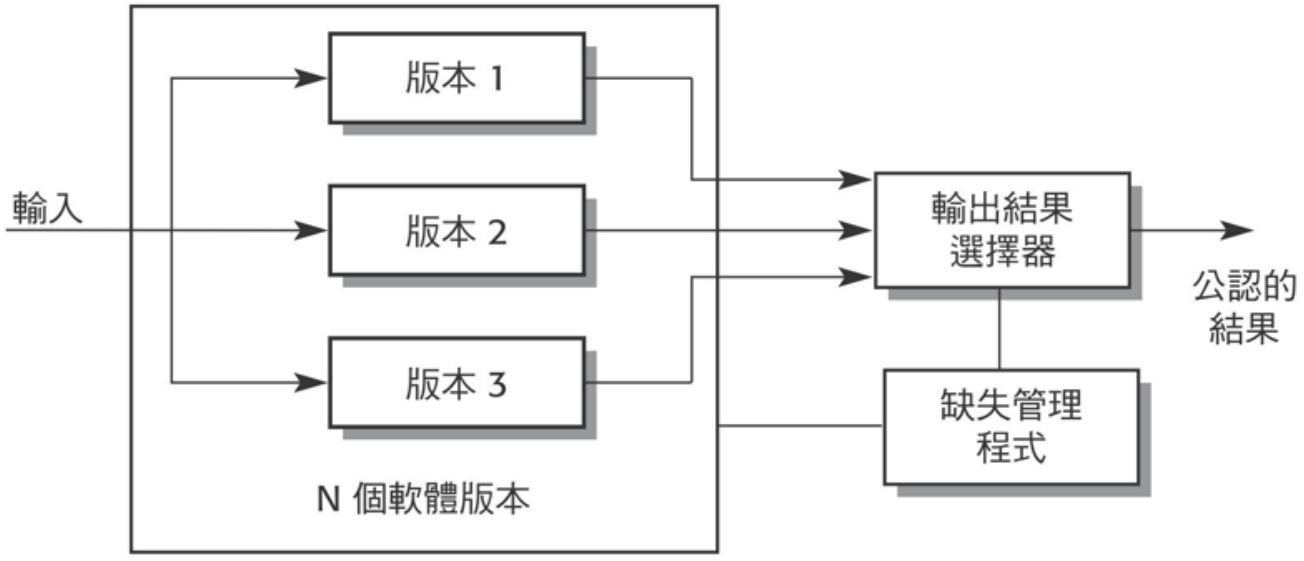


圖 11.10 N 版本程式設計

11.3.4 軟體多樣性

- 以上所有的容錯架構都是依賴軟體多樣性來達到容錯的效果，它所根據的假設是對同一規格(或部分規格，如保護系統)進行多樣化實作的結果是相互獨立的，也就是它們不應該有同樣的錯誤，而且不會同時以同樣的方式故障。
- 理想上系統的多樣化版本應該彼此間沒有相依性，而且故障方式也應該完全不同。
- 然而在實務上，要達到管道完全互不相依是不可能的。不相依的設計團隊卻經常犯下相同的錯誤。

11.3.4 軟體多樣性

1. 不同團隊的成員卻來自相同文化背景的情形很常見，說不定還是用同樣的方式和同一本教科書教育出來的。所以大家可能會覺得同樣的東西很難懂，而且在與領域專家溝通時也有同樣的困難。因此他們很可能會犯同樣的錯誤。
2. 假如需求不正確或對系統環境有同樣的誤解，那麼這些錯誤可能會出現在系統的每個實作版本上。
3. 在關鍵系統中，細部 (detailed) 系統規格是從系統的需求衍生出來的。假如某項規格有模稜兩可的情況，那麼不同團隊對該項規格可能會有同樣的誤解。

11.3.4 軟體多樣性

- 要降低共同規格錯誤發生的可能性，其中一個方法是獨立開發系統的細部規格，並且以不同語言來定義這些規格。
- 在某個實驗的分析報告中，Hatton (Hatton 1997) 的結論是3管道系統比單管道系統大約可靠5到9倍。他的結論是對單一版本投注更多資源，可靠性無法提升這麼多，因此N版本方法可產生出比單版本方法更可靠的系統。
- 多版本系統其可靠性的提升程度，是否值得額外花這些開發成本。

11.4 提升可靠性的程式設計

- 有一組公認良好的程式設計實務經驗法則，可應用在幾乎每種程式語言上，有助於減少系統的缺失。

可信賴的程式設計法則

1. 限制程式中資訊的能見度。
2. 檢查所有輸入的有效性。
3. 為所有例外狀況提供處理函式。
4. 儘量減少使用容易出錯的構件。
5. 提供重新啟動的能力。
6. 檢查陣列邊界。
7. 呼叫外部元件時要包含逾時 (timeout) 處理。
8. 所有代表真實世界數值的常數都要命名。

圖 11.11 可信賴的程式設計的良好實務經驗法則

11.4 提升可靠性的程式設計

- **法則1：限制程式中資訊的能見度**
- 程式元件應該也只能存取實作所需的資料，其他資料就不應該能存取到，而且要隱藏起來，這樣就不會被不應該用到它的程式元件將它損毀。
- 要達到這個目的，可將資料結構實作成抽象資料型別 (abstract data type)。
- 舉例而言，假設有個抽象資料型別是儲存針對某服務的請求佇列 (queue)，則它的運算動作應該有get和put，分別是新增和移除佇列項目；還有一個運算是傳回目前佇列中的項目個數。

11.4 提升可靠性的程式設計

- **法則2：檢查所有輸入的有效性**
- 所有程式都會從它們的環境取得輸入並加以處理，很多時候系統規格並未定義輸入不正確要採取什麼行動。然而使用者免不了會犯錯，有時會輸入錯的資料；有些惡意攻擊手法就是故意輸入不正確的資料來攻擊系統。
- 因此一旦從程式的作業環境讀取到輸入值，應該要馬上檢查它的有效性。

11.4 提升可靠性的程式設計

1. **範圍檢查 (range check)**：程式預期輸入值在某個範圍內。
2. **長度檢查 (size check)**：程式預期輸入值的長度是在多少個字元以內，例如用8個字元來表示銀行帳戶。
3. **表示法檢查 (representation check)**：程式可能預期輸入值是屬於某個型別。舉例來說，人名中不能包括數字。
4. **合理性檢查 (reasonableness check)**：程式可能預期輸入值是屬於某個集合，否則就不合理。

11.4 提升可靠性的程式設計

- **法則3：為所有例外狀況提供處理函式**
- 在程式執行期間發生的錯誤或非預期事件稱為例外狀況 (exception)，例如系統電源故障。
- 系統可以讓程式本身來處理，或是將控制權轉移給系統的例外處理 (exception handling) 機制來接手。
- 為了確保程式的例外狀況不會引起系統故障，應該要針對每個可能的例外狀況，定義一個例外處理函式 (exception handler)。

11.4 提升可靠性的程式設計

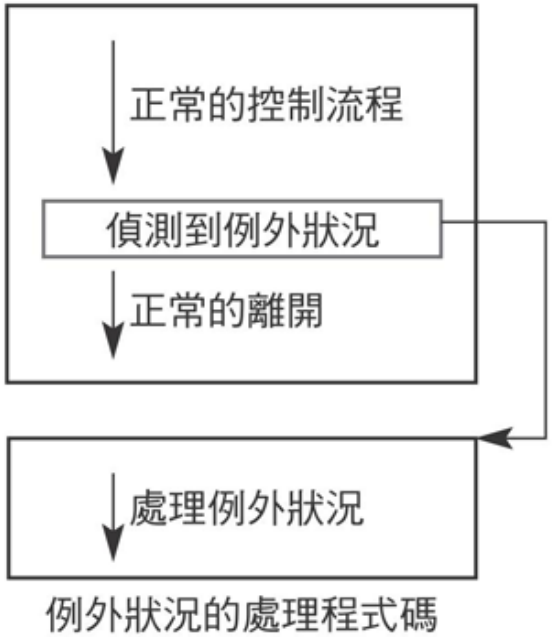


圖 11.12 例外狀況的處理

11.4 提升可靠性的程式設計

1. 發出信號通知高階元件此例外狀況已經發生，並提供例外狀況類型的資訊給元件。
2. 對原來打算進行的事進行其他的處理方式。
3. 將控制權轉移給負責處理例外狀況的程式語言執行期間 (runtime) 支援系統，這經常是程式出現缺失時的預設做法 (例如數值超出範圍)。

11.4 提升可靠性的程式設計

- **法則4：儘量減少使用容易出錯的構件**
- 程式中的缺失以及所導致的許多程式故障，通常都是人為疏失的後果。程式設計人員可能會因為沒有追蹤所有狀態變數之間的關係而造成一些錯誤，也可能會寫出一些產生非預期行為和改變系統狀態的程式敘述。
- 有些程式設計方法的確比較容易引進錯誤到程式中。
- 舉例來說，你應該儘量避免使用浮點數，因為它的精確度受到硬體的限制，所以極大或極小的浮點數的比較運算結果並不可靠。是動態記憶體配置，也就是由你在程式中明確的管理儲存空間。

11.4 提升可靠性的程式設計

- **法則5：提供重新啟動的能力**
- 因此在這類系統上，應該提供重新啟動 (restart) 的功能，它會在處理過程中收集資料並加以保存，必要時可利用這些資料繼續往下做，而不是從頭開始。
- 這些保存副本有時叫做檢查點 (checkpoint)，例如：
  1. 在電子商務系統上，應該要保存使用者所填的表單內容，讓他們可以存取這些表單並重新送出，不必全部重填。
  2. 在長久異動或計算繁重的系統上，應該每隔幾分鐘自動儲存資料，這樣在發生系統故障時，就可使用最新儲存的資料重新啟動。

11.4 提升可靠性的程式設計

- **法則6：檢查陣列邊界**
- 所有程式語言都有陣列 (array)，它是一種循序的資料結構，透過數字索引值 (index) 來存取。
- 有些程式語言如C和C++ 沒有自動內建陣列邊界檢查功能，只有直接計算從陣列開頭的位移，因此A [12345] 會存取到從陣列開頭位置開始算第12345個位置，不管到底是不是在陣列內。
- 然而省略邊界檢查會導致保全上的漏洞，例如緩衝區溢載 (參見第13章)，這可能會讓系統有漏洞而導致故障。
- 假如使用的語言沒有內建陣列邊界檢查 (如C或C++)，你就一定要額外撰寫這部分的程式碼，以確保陣列索引值有在邊界內。

11.4 提升可靠性的程式設計

- **法則7：呼叫外部元件時要包含逾時 (timeout) 處理**
- 在分散式系統中，系統的各元件可能是在不同電腦上執行，而元件之間會經過網路互相呼叫。
- 為了避免這種情況，在呼叫外部元件時應該要包含逾時處理。所謂逾時 (timeout) 是時間過了就自動假設被呼叫元件已故障不會發出回應。
- 如果超過時限還沒得到回應，就假設發生故障而將控制權從被呼叫元件那邊取回。

11.4 提升可靠性的程式設計

- **法則8：所有代表真實世界數值的常數都要命名**
- 只要不是太陽春的程式，都會包含一些代表真實世界實體的常數值。這些值在程式執行期間不會被修改。其中有些是絕對常數永遠不會變(如光速)，不過大部分是很少改(如稅率可能每隔幾年才改)。
- 在程式中一定要有一個區塊是專門放會用到的常數。使用這些常數時應該參考名稱，而不是它們的值。有兩個優點：
  1. 使用名稱比較不會打錯字。數字很容易打錯字，系統也不會偵測出來。
  2. 如果常數值需要變更，就不必找遍整個程式一一修改，只要修改宣告常數的地方即可。

11.5 測量可靠性

- 在評估系統可靠性時，必須收集它的作業資料。要求的資料可能包含：
  1. 對系統發出固定次數的服務要求中，系統發生故障的次數。這個方法可以用來測量POFOD。
  2. 系統發生故障之間的時間(或交易個數)。這個方法可以用來測量ROCOF和MTTF。
  3. 當系統發生故障修復所花的時間或重新啟動的所需時間。這是用來測量可用率，可用率不只與兩次故障之間的間隔時間有關，同時也與讓系統回復運作所需的時間長短有關。

11.5 測量可靠性

- 可靠性測試 (reliability testing) 是一種專門測量系統可靠性的測試程序。可靠性測試程序中可檢查系統是否達到規定的可靠性水準。

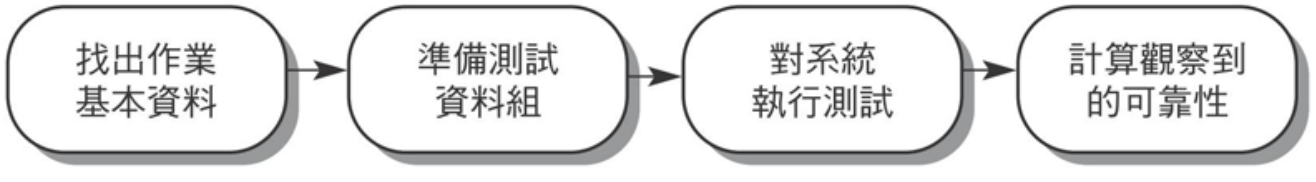


圖 11.13　測量可靠性的統計測試

11.5 測量可靠性

- 最主要的困難可歸因於下列幾個原因：
  1. **作業基本資料不確定**：以其他系統的經驗為基礎的作業基本資料，可能無法精確的反映出系統真正使用的習慣。
  2. **產生測試資料的成本很高**：除非測試資料能全自動產生，否則要產生作業基本資料要求的大量測試資料，可能需要很高的成本。
  3. **在需要非常高規格的可靠性時，統計上有不確定性**：如果要精確的測量可靠性，必須有大量的故障次數才能做統計。但是當系統已經很可靠時，故障就很不容易發生，所以很難產生出新故障。
  4. **辨識故障**：是否有發生系統故障並不一定很明顯看得出來，假如有正規化規格，那麼也許可由規格推導而辨識出。

11.5.1 作業基本資料

- 軟體系統的作業基本資料可反映出它實際上是如何被使用的，這份基本資料是由輸入值分類的規格和出現這些輸入值的機率所組成。
- 當有新的軟體系統取代現存的自動化系統時，應該可以很容易評估新軟體的可能使用模式。它應該與現行的使用模式大致相同，另外再加上一些新功能的使用模式。

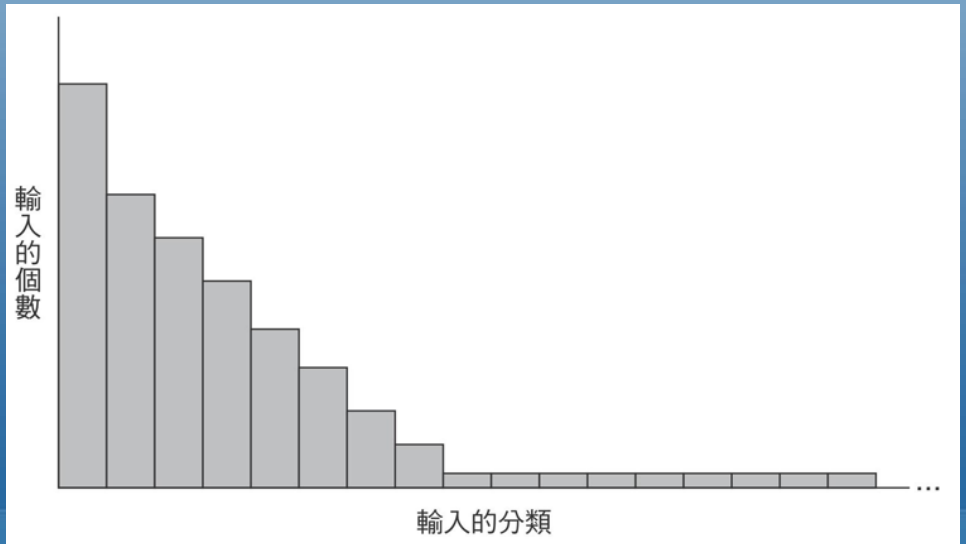


圖 11.14　作業基本資料中的輸入分布情況

11.5.1 作業基本資料

- 假設軟體系統是全新的或創新的，就比較難預期它會如何被使用，因此實際上不可能建立出精確的作業基本資料。
- 對其他系統而言也許很困難或甚至做不到：
  1. 因為系統可能有許多不同使用者，而各個使用者有自己使用系統的方式。
  2. 使用者對系統的使用方式會逐漸改變。