與錯誤打交道

主要參考資料來源：《失控》，Kevin Kelly著，東西文庫譯，新星出版社，2010

 在網路時代，大型軟體可能是現在人類能夠製造出的最複雜東西了。以微軟的操作系統來說，就有超過400萬條程式，系統中出現的bug，必需不斷修正更新。那麼，網路經濟是否可能製造出沒有缺陷的超級複雜系統，還是只能建立一個有漏洞的複雜系統？

 加拿大電腦學家David Parnas提出連續和非連續的二種複雜系統觀念。汽車公司在測試新車應對急轉彎的性能時，會以不同時速下進行測試，如果50、60、70哩時能通過測試，那麼在其中間插值如55、67…等的速度下，也肯定能通過，因為這是一個連續的系統。

 然而電腦軟體、分布式網路以及絕大多數的活系統都是非連續系統。這就表示，在複雜的適應性系統中，我們並不能依賴插值函數來判斷系統的行為。因為斷點始終都是存在的，雖然我們測試了所有的鄰近值，可是仍不能保證系統不會在某特定點崩壞，甚至連有沒有這種隱患都不知道。對於非連續系統，是不能依靠抽樣的方式對系統進行測試的！

 在網路經濟中，研發新產品的費用主要源自於生產流程的設計，而非產品設計。在工業化生產中，日本人特別擅長生產流程的設計和改進，而美國人則擅長產品的開發和設計。我們要把軟體看作是一個生產流程而不是產品，也就是說，我們賣給客戶的相當於一個工廠(一套程式)，在客戶需要的時候為他製造出一個他需要的產品。

問題就在於要怎麼才能生產一個「零缺陷」的工廠呢？

 通常，編寫軟體的步驟有三：首先，設計一個全景圖；然後用程式語言寫出細節；最後進行測試。要怎麼做到「零缺陷」呢？重點是要把整個系統分散成上千個小步驟，設計、編寫、測試工作每天都在成百上千個小工作間裡進行，這等於每個人都有一個客戶(即你的工作伙伴)，你必需把自己的部份做好才能交下去給你的工作伙伴。你的伙伴立刻對它進行檢測，並把錯誤回饋給你，讓你知道並修改，這是一個由下向上的發展過程，而每個步驟就是一個小模塊。

 可是，單靠這些小步驟還不能得到「零缺陷」的軟體系統。所謂的「缺陷」是指被交付出去的錯誤，如果在交付之前能修正錯誤就不能算是缺陷。因此，我們雖然不可能絕對的避免錯誤，但是我們可以避免錯誤成為缺陷。所以，所謂的「零缺陷」的任務就是儘早發現錯誤，儘早改正錯誤！然而，真正的改進在於儘早發現產生錯誤的原因並儘早清除。請注意，犯錯的是人，但處理錯誤的則是系統。

 在軟體編寫工作有一種防錯設計叫做「拼寫錯誤檢查器」，它不允許程式工程師輸入任何拼寫錯誤的指令。還有一些高階的研發工具可以對程式的邏輯性進行分析和評價，並提供合乎邏輯的改錯選擇。另外一種非常重要的防錯手段就是對複雜的軟體進模塊化。1982年一項研究顯示，在其他條件完全相同的情況下，程式碼總行數相同的大型軟體和把它拆分為子程序後加以比較，一個一萬行的程式如果是一個整體，它有317個錯誤，可是如果把它拆分為3組子程式的話，總錯誤就可減少為265個，而且每拆一次都可減少錯誤量，大致上符合線性關係。結論就是：模塊化雖然不能完全解決問題，但它是一種有效的手段！理論上來說，當子程式小到某個閾限以下，就可以達到沒有錯誤的狀態。

 物件導向程式設計(Object-Oriented Programming, OOP)是一種程式開發的法則，大陸稱為「面向對象程序設計」，實際上就是一個相對去中心化的、模塊式的程式設計方法。對於一個OOP，它就是一個獨立完整的單元，可以和其他的OOP整合在一起，形成一個可分解的指令結構。也就是說OOP把每一個功能都限制在可掌控的單元內，即使一個對象崩潰了，程式的其他部分也可以繼續運轉，不會像傳統程式系統只要一個地方出了問題，就會全部崩盤。OOP就像是帶了小智能的樂高(Lego)積木玩具小塊，它知道自己能幹什麼不能幹什麼，同時也能和其他的對象橫向交流。它有一定的抗錯性、能快速修復，並能通過有效單元的組裝實現擴充。

 程式的缺陷還有一個現象，那就是「錯誤扎堆」，也就是說會出錯的總是會出錯，從未出過錯的模塊也常會一直保持不敗金身。應對之策就是：「不要把錢花在錯誤百出的模塊上，應該直接放棄掉並重寫，因為重寫和修補一個老是出錯的代價差不多。程式工程師必需知道有很多東西我們並無法完全理解，所以要做到「零缺陷」免不了的代價就是「過度設計」，犧牲簡潔性來換取可靠性，不求最優但求夠好。

 網路經濟的未來在於設計出可靠的流程，而不是可靠的產品。本質上意味著這種程式是不可能最優化的。在一個分布式的、半活性的世界中，我們的目標只能「滿意化」，而什麼是滿意也是隨著形勢而變的。