

The  
Pragmatic  
Programmers

TURING

图灵程序设计丛书

Practices of an Agile Developer

# 高效程序员的45个习惯

## 敏捷开发修炼之道

[美] Venkat Subramaniam 著  
Andy Hunt  
钱安川 郑柯 译



人民邮电出版社  
POSTS & TELECOM PRESS

# 版權信息

書名：高效程序員的45個習慣：敏捷開發修煉之道

作者：Andy Hunt · Venkat Subramaniam

譯者：鄭柯， 錢安川

ISBN：978-7-115-21553-6

本書由北京圖靈文化發展有限公司發行數字版。版權所有，侵權必究。

---

您購買的圖靈電子書僅供您個人使用，未經授權，不得以任何方式複製和傳播本書內容。

我們願意相信讀者具有這樣的良知和覺悟，與我們共同保護知識產權。

如果購買者有侵權行為，我們可能對該用戶實施包括但不限於關閉該帳號等維權措施，並可能追究法律責任。

## 目錄

[版權聲明](#)

[引言](#)

[對本書的讚譽](#)

[推薦序一](#)

[推薦序二](#)

## 譯者序

### 第1章 敏捷——高效軟件開發之道

### 第2章 態度決定一切

- 1 做事
- 2 欲速則不達
- 3 對事不對人
- 4 排除萬難，奮勇前進

### 第3章 學無止境

- 5 跟蹤變化
- 6 對團隊投資
- 7 懂得丟棄
- 8 打破砂鍋問到底
- 9 把握開發節奏

### 第4章 交付用戶想要的軟件

- 10 讓客戶做決定
- 11 讓設計指導而不是操縱開發
- 12 合理地使用技術
- 13 保持可以發佈
- 14 提早集成，頻繁集成
- 15 提早實現自動化部署
- 16 使用演示獲得頻繁反饋
- 17 使用短迭代，增量發佈
- 18 固定的價格就意味著背叛承諾

### 第5章 敏捷反饋

- 19 守護天使
- 20 先用它再實現它
- 21 不同環境，就有不同問題
- 22 自動驗收測試
- 23 度量真實的進度
- 24 傾聽用戶的聲音

### 第6章 敏捷編碼

- 25 代碼要清晰地表達意圖

- 26 用代碼溝通
- 27 動態評估取捨
- 28 增量式編程
- 29 保持簡單
- 30 編寫內聚的代碼
- 31 告知，不要詢問
- 32 根據契約進行替換

## 第7章 敏捷調試

- 33 記錄問題解決日誌
- 34 警告就是錯誤
- 35 對問題各個擊破
- 36 報告所有的異常
- 37 提供有用的錯誤信息

## 第8章 敏捷協作

- 38 定期安排會面時間
- 39 架構師必須寫代碼
- 40 實行代碼集體所有制
- 41 成為指導者
- 42 允許大家自己想辦法
- 43 準備好後再共享代碼
- 44 做代碼複查
- 45 及時通報進展與問題

## 第9章 尾聲：走向敏捷

- 9.1 只要一個新的習慣
- 9.2 拯救瀕臨失敗的項目
- 9.3 引入敏捷：管理者指南
- 9.4 引入敏捷：程序員指南
- 9.5 結束了嗎

## 附錄A 資源

## 索引

# 版權聲明

Copyright © 2006 Venkat Subramaniam and Andy Hunt. Original English language edition, entitled *Practices of an Agile Developer: Working in the Real World*.

Simplified Chinese-language edition copyright © 2010 by Posts & Telecom Press. All rights reserved.

本書中文簡體字版由The Pragmatic Programmers, LLC授權人民郵電出版社獨家出版。未經出版者書面許可，不得以任何方式複製或抄襲本書內容。

版權所有，侵權必究。

## 引言

கற்க கசடறக் கற்பவை கற்றபின்  
நிற்க அதற்குத் தக.

திருக்குறள்-391

學有所成，學有所用。

《提魯庫拉爾》中1330條警句第391條

提魯瓦魯瓦 ( Thiruvalluvar )，印度詩人和哲學家，公元前31年

Almost every wise saying  
has an opposite one,  
no less wise,  
to balance it.

—George Santayana

幾乎每句所謂至理名言都有句意思相反的話與之對應，

而且後者也同樣在理。

——喬治·桑塔耶納（美國哲學家、詩人）

## 對本書的讚譽

書中“切身感受”的內容非常有價值——通過它我們可以做到學有所思，思有所悟，悟有所行。

■ Nathaniel T. Schutta，《Ajax基礎教程》作者

在我眼中，這就是一本名不虛傳的Pragmatic書架系列裡的圖書：簡短、易讀、精煉、深入、深刻且實用。它對於那些想要採用敏捷方法的人很有價值。

■ Forrest Chang，軟件主管

我從一開始看這本書時就一直在想：“哇噻！很多開發者都需要這本書。”我很快就認識到這正是我需要的書，故而向各個級別的開發者強烈推薦。

Guerry A. Semones，Appistry公司資深軟件工程師

此書通過常理和經驗，闡述了為什麼你應該在項目中使用敏捷方法。最難得的是，這些行之有效的實戰經驗，竟然從一本書中得到了。

Matthew Johnson，軟件工程師

我買過Pragmatic書架系列的其他書籍，從中看到過這本書提到的一些習慣。但是，本書把這些思想整合到一起，而且用了簡明、易懂的方式組織起來。我在此向開發新手和想要變得“敏捷”的團隊強烈推薦此書。

Scott Splavec，資深軟件工程師

伴隨著敏捷實踐席捲了整個行業，有越來越多的人需要理解什麼是真正的“敏捷”。這本簡明、實用的書，正符合他們的需求。

Marty Haught，Razorstream公司軟件工程師和架構師

也許，你已經聽說過了敏捷方法學，並在思索著如何才能每天改進自己的工作。我的答案是好好讀這本書，傾聽這天籟之音，融會貫通這些最佳習慣吧。

David Lázaro Saz，軟件開發者

這是一本深入淺出地講解敏捷核心實踐的書。我最欣賞本書的地方在於，它不是在推銷一個具體的敏捷方法學，而是把各種敏捷方法中的有效實踐有機地串聯成一個整體。它適合那些渴望快速而可靠地交付高質量軟件的人們。

Matthew Bass，軟件諮詢師

這本書是《程序員修煉之道》的完美續篇！

Bil Kleb，NASA研究員

# 推薦序一

僅僅還在幾年前，XP還被認為是方法異教，FDD屬於黑客程序方法。如今，敏捷儼然已經成為主流學說，敏捷方法成為人們學習和討論的熱點。敏捷方法的應用也更加廣泛，以至於不少外包項目都要求採用某種敏捷方法。它不僅僅是小團隊和小項目在使用，甚至連微軟都開始使用Scrum。

敏捷已經成為一種炙手可熱的新時尚。

因為火熱，各種不同的說法就多起來；因為時尚，原本有些不認同敏捷的人也開始追捧起來。人們反覆地討論敏捷方法，涉及從哲學思想到實現細節的各個層面。人們不斷地推出各種不同版本的敏捷方法，甚至有些方法顯得如此矛盾、如此不同。

同時，一些誤解也一直在坊間流行。一般誤認為敏捷就是快，越快就是越敏捷——字典上的名詞解釋是其依據。豈不知它本來要以“lightweight processes”（輕量級過程）命名，只不過有些參會者不喜歡被看做是在拳臺上跳來跳去的輕量級拳手，所以才用了“敏捷”這個詞。還有其他一些誤解是，敏捷就是隻寫代碼不寫文檔；敏捷需要重構而無需設計；敏捷迭代就是儘量做到最小，以至於一個小時就好幾次；敏捷需要天才的程序員才能應用，其他人都會水土不服；如此這般。

可以看到，市面上以敏捷為題目的圖書俯拾皆是，似乎軟件開發的書不加上敏捷這個詞就是落伍一樣。敏捷體系下存在多種方法，介紹每種方法的圖書就有一大堆。再加上每種方法採用不同的技術，每本書採用不同的組織形式，存在這麼多書也不奇怪，就更不用提那些僅僅為了跟風而敏捷的作品了。



面對如此百花齊放、百家爭鳴的現象，你該從什麼地方開始呢？有沒有一本圖書可以作為入門的第一讀物呢？

這本書就可以勝任這樣的角色！

這是一本很容易理解並掌握，不需要太多基礎就可以閱讀的書。不管你是開發人員，還是管理人員、財務等後勤人員、學生、編程愛好者，只要你對敏捷有興趣，就可以讀懂這本書。你不會被眾多的概念和曲折的邏輯所迷惑，不會被高難度技巧所困擾。這本書為你打開了瞭解和學習敏捷方法的一扇大門，並指出繼續前進的道路。

你會很悠閒自在地讀完這本小書，然後說：“原來敏捷就是這麼一回事啊！”

自由軟件顧問 劉新生（OZZZZZZ）

## 推薦序二

我很喜歡本書的中文書名——高效程序員的45個習慣，比直譯成“敏捷開發者實踐”含蓄多了。敏捷不是目的，只是手段。只要某個手段適合某個場景，有助於提升質量，提高交付能力，提高開發者水平……總而言之，有好處的事情，我們儘管做就是了，何必冠以敏捷之名？

記得第一次讀本書還是兩年前。這時又細細讀來，越來越覺得“習慣”一詞比“實踐”更有味道。所謂“流水不腐，戶樞不蠹”，廚房髒了就擦一下，總比滿牆都是油煙以後再去清理的代價小得多。有價值的東西——比如回顧、測試、重構，一切有利於團隊建設、提高生產力的實踐都應該頻繁且持續做，然後日積月累就養成了習慣。

有些習慣很容易養成，有些則很難。我們大都常常許願，做計劃，比如要做一個至少100人同時在線的成熟應用，參加義工活動，每週至

少一篇博客.....然後在計劃落空的時候，用各種理由來安慰自己。

李笑來老師在《把時間當作朋友》一書中提到：“所有學習上的成功，都只靠兩件事：策略和堅持，而堅持本身就應該是最重要的策略之一。”那麼，為什麼我們會在某些事情上堅持不下去？或者換個角度來看，哪些事情是容易堅持下去的？

以前我是標準的宅男，CS、網絡小說、魔獸世界幾乎是休閒的全部，等到後來得了腰肌勞損，又得了頸椎病，這才痛定思痛，開始游泳鍛鍊身體。每天遊兩千米，一個月以後，游泳就成了習慣。再舉個例子，我老婆生完孩子以後體型變化很大，立志想要減肥。為了堅持下去，她把懷孕前的照片放在電腦桌面上，時時督促自己。後來，減肥也就變成了一種生活方式。

從我的個人體驗來看，難以堅持下去的事情，基本都是因為沒有迫切的慾望和激情。單說鍛鍊身體，無論是為了減肥、祛病，還是塑形美體等，做這些事情至少都有明確的目的，這樣才能驅使著人們一直堅持下去。沒有動機，沒有慾望，哪裡來的毅力呢？

那麼，當我們決定做一件事情的時候，首先就要多問問自己：為什麼要做這件事情？它所帶來的好處是什麼？如果不做它又會有哪些壞處？有了清晰的目的和思路後再去做事，遇到變化時就知道孰輕孰重，該怎麼調整計劃，同時也不至於被重複和乏味消磨了一時的意氣。翻開本書之後，你同樣也該對自己提問：“為什麼要有自動驗收測試，有了足夠的單元測試是不是就能保證質量了？”“寫自動驗收測試有哪些成本，會帶來哪些收益？”只有明白了“為什麼做”，才能夠解決“如何做”的問題。

本書的兩名譯者與我都是故交。錢安川是我的同事，是ThoughtWorks資深諮詢師，有豐富的敏捷實施經驗。鄭柯與我同是InfoQ中文站敏捷社區的編輯，一起翻譯過數十篇稿件。他翻譯的《項目管理修煉之道》也即將由圖靈公司出版。這次二人聯手的作品，定會給讀者以賞心悅目的閱讀體驗。我有幸已經從樣章中感受到了這一點。

希望你能夠帶著問題，踏上愉快的閱讀之旅。

希望你能夠養成好習慣。

李 劍

ThoughtWorks諮詢師

2009.10.10

## 譯者序

“武功者，包括內功、外功、武術技擊術之總和。有形的動作，如支撐格拒，姿式迴環，變化萬千，外部可見，授受較易，晨操夕練，不難熟練。而無形的內功指內部之靈惠素質，即識、膽、氣、勁、神是也，此乃與學練者整個內在世界的學識水平密切相關，是先天之慧根悟性與後天智能的總成，必需尋得秘籍方可煉成。”

摘自《武林秘籍大全》

公元21世紀，軟件業江湖動盪，人才輩出，各大門派林立，白道黑幫，都欲靠各自門派的武功稱霸武林。

在那些外家功門派（傳統的瀑布開發方法、CMM、ISO和RUP等）和非正統教（中國式太極敏捷UDD等）當道之際，一股新勢力正在崛起——以敏捷方法為總稱的一批內家功門派。

下面的歌訣是對內家武功招數的概述：

迭代開發，價值優先

分解任務，真實進度

站立會議，交流暢通

用戶參與，調整方向

結對編程，代碼質量

測試驅動，安全可靠

持續集成，儘早反饋

自動部署，一鍵安裝

定期回顧，持續改進

不斷學習，提高能力

上面的每種招式，都可尋得一本手冊，介紹其動作要領和攻防章法。幾乎每個內家功門派都有自己的拳法和套路。

但，正所謂“練拳不練功，到老一場空”。學習招數和套路不難，難的是如何練就一身真功夫。內家功，以練內為主，內外結合，以動作引領內氣，以內氣催領動作，通過後天的修煉來彌補先天的不足。

本書是一本內功手冊。它注重於培養軟件開發者的態度、原則、操守、價值觀，即識、膽、氣、勁、神是也。

敏捷的實踐者Venkat Subramaniam和Andy Hunt攜手著下此書。望有志之士有緣得到此書，依法修習，得其精要；由心知到身知，入筋、入骨、入髓，修煉得道。而後，匡扶正義，交付高質量的軟件，為人類造福。

安 川

2008年4月於北京

# 第1章 敏捷——高效軟件開發之道

不管路走了多遠，錯了就要重新返回。

——土耳其諺語

這句土耳其諺語的含義顯而易見，你也會認同這是軟件開發應該遵守的原則。但很多時候，開發人員（包括我們自己）發現自己走錯路後，卻不願意立即回頭，而是抱著遲早會步入正軌的僥倖心理，繼續錯下去。人們會想，或許差不多吧，或許錯誤不像想象的那麼嚴重。假使開發軟件是個確定的、線性的過程，我們隨時可以撤回來，如同諺語中所說的那樣。然而，它卻不是。

相反，軟件開發更像是在衝浪——一直處於動態、不斷變化的環境中。大海本身無法預知，充滿風險，並且海里還可能有鯊魚出沒。

衝浪之所以如此有挑戰性，是因為波浪各不相同。在衝浪現場，每次波浪都是獨一無二的，衝浪的動作也會各不相同。例如，沙灘邊的波浪和峭壁下的波浪就有很大的區別。

在軟件開發領域裡，在項目研發過程中出現的需求變化和挑戰就是你在衝浪時要應對的海浪——它們從不停止並且永遠變化，像波浪一樣。在不同的業務領域和應用下，軟件項目具有不同的形式，帶來了不同的挑戰。甚至還有鯊魚以各種偽裝出沒。

軟件項目的成敗，依賴於整個項目團隊中所有開發成員的技術水平，對他們的培訓，以及他們各自的能力高低。就像成功的衝浪手一樣，開發人員必須也是技術紮實、懂得掌握平衡和能夠敏捷行事的人。不管是預料之外的波浪衝擊，還是預想不到的設計失敗，在這兩種情況下敏捷都意味著可以快速地**適應**變化。

## 敏捷開發宣言

我們正通過親身實踐和幫助他人實踐，揭示了一些更好的軟件開發方法。通過這項工作，我們認為：

- 個體和交互勝過過程和工具
- 可工作的軟件勝過面面俱到的文檔
- 客戶協作勝過合同談判
- 響應變化勝過遵循計劃

雖然右項也有價值，但我們認為左項具有更大的價值。

敏捷宣言作者，2001年版權所有。

更多詳細信息可以訪問[agilemanifesto.org](http://agilemanifesto.org)。

## 敏捷的精神

那麼，到底什麼是敏捷開發方法？整個敏捷開發方法的運動從何而來呢？

2001年2月，17位志願者（包括作者之一Andy在內）聚集在美國猶他州雪鳥度假勝地，討論一個新的軟件開發趨勢，這個趨勢被不嚴格地稱為“**輕量型軟件開發過程**”。

我們都見過了因為開發過程的冗餘、笨重、繁雜而失敗的項目。世上應該有一種更好的軟件開發方法——只關注真正重要的事情，少關注那些佔用大量時間而無甚裨益的不重要的事情。

這些志願者們給這個方法學取名為**敏捷**。他們審視了這種新的軟件開發方法，並且發佈了敏捷開發宣言：一種把以人為本、團隊合作、快

速響應變化和可工作的軟件作為宗旨的開發方法（本頁最開始的方框裡就是宣言的內容）。

敏捷方法可以快速地響應變化，它強調團隊合作，人們專注於具體可行的目標（實現真正可以工作的軟件），這就是敏捷的精神。它打破了那種基於計劃的瀑布式軟件開發方法，將軟件開發的實際重點轉移到一種更加自然和可持續的開發方式上。

它要求團隊中的每一個人（包括與團隊合作的人）都具備職業精神，並積極地期望項目能夠獲得成功。它並不要求所有人都是有經驗的專業人員，但必須具有專業的工作態度——每個人都希望盡最大可能做好自己的工作。

如果在團隊中經常有人曠工、偷懶甚至直接怠工，那麼這樣的方法並不適合你，你需要的是一些重量級的、緩慢的、低生產率的開發方法。如果情況並非如此，你就可以用敏捷的方式進行開發。

這意味著你不會在項目結束的時候才開始測試，不會在月底才進行一次系統集成，也不會在一開始編碼的時候就停止收集需求和反饋。

### 開發要持續不斷，切勿時續時斷

### **Continuous development, not episodic**

相反，這些活動會貫穿項目的整個生命週期。事實上，只要有人繼續使用這個軟件，開發就沒有真正結束。我們進行的是持續開發、持續反饋。你不需要等到好幾個月之後才發現問題：越早發現問題，就越容易修復問題，所以應該就在此時此刻把問題修復。

這就是敏捷的重點所在。

這種持續前進的開發思想根植于敏捷方法中。它不但應用於軟件開發的生命週期，還應用於技術技能的學習、需求採集、產品部署、用戶培訓等方面。它包括了軟件開發各個方面的所有活動。

## 持續注入能量

### Inject energy

為什麼要進行持續開發呢？因為軟件開發是一項非常複雜的智力活動，你遺留下來的任何問題，要麼僥倖不會發生意外，要麼情況會變得更糟糕，慢慢惡化直到變得不可控制。當問題累積到一定程度的時候，事情就更難解決，最後無法扭轉。面對這樣的問題，唯一有效的解決辦法就是持續地推進系統前進和完善（見《程序員修煉之道》一書中的“軟件熵”[HT00]）。

有些人對使用敏捷方法有顧忌，認為它只是另一種**危機管理**而已。事實並非如此。危機管理是指問題累積並且惡化，直到它們變得非常嚴重，以至於你不得不立即放下一切正在做的工作來解決危機。而這樣又會帶來其他的負面影響，你就會陷入危機和恐慌的惡性循環中。這些正是你要避免的問題。

所以，你要防微杜漸，把問題解決在萌芽狀態，你要探索未知領域，在大量成本投入之前先確定其可行性。你要知錯能改，在事實面前主動承認自己的所有錯誤。你要能自我反省，經常編碼實戰，加強團隊協作精神。一開始你可能會覺得不適應，因為這同以往有太多的不同，但是只要能真正地行動起來，習慣了，你就會得心應手。

## 敏捷的修煉之道

下面一句話是對**敏捷**的精闢概括。

**敏捷開發就是在一個高度協作的環境中，不斷地使用反饋進行自我調整和完善。**

下面將扼要講述它的具體含義，以及敏捷的團隊應該採取什麼樣的工作和生活方式。



首先，它要整個團隊一起努力。敏捷團隊往往是一個小型團隊，或者是大團隊分成的若干小團隊（10人左右）。團隊的所有成員在一起工作，如果可能，最好有獨立的工作空間（或者類似bull pen<sup>①</sup>），一起共享代碼和必要的開發任務，而且大部分時間都能在一起工作。同時和客戶或者軟件的用戶緊密工作在一起，並且儘可能早且頻繁地給他們演示最新的系統。

① bull pen原指在棒球比賽中，候補投手的練習場。——譯者注

你要不斷從自己寫的代碼中得到反饋，並且使用自動化工具不斷地構建（持續集成）和測試系統。在前進過程中，你都會有意識地修改一些代碼：在功能不變的情況下，重新設計部分代碼，改善代碼的質量。這就是所謂的**重構**，它是軟件開發中不可或缺的一部分——編碼永遠沒有真正意義上的“結束”。

要以**迭代**的方式進行工作：確定一小塊時間（一週左右）的計劃，然後按時完成它們。給客戶演示每個迭代的工作成果，及時得到他們的反饋（這樣可以保證方向正確），並且根據實際情況儘可能頻繁地發佈系統版本讓用戶使用。

對上述內容有了瞭解之後，我們會從下面幾個方面更深入地走進敏捷開發的實踐。

**第2章：態度決定一切**。軟件開發是一項智力勞動。在此章，我們會講解如何用敏捷的心態開始工作，以及一些有效的個人習慣。這會為你使用敏捷方法打下紮實的基礎。

**第3章：學無止境**。敏捷項目不可能坐享其成。除了開發之外，我們還要在幕後進行其他的訓練，雖然它不屬於開發工作本身，但卻對團隊的發展極其重要。我們還將看到，如何通過培養習慣來幫助個人和團隊成長並自我超越。

**第4章：交付用戶想要的軟件**。如果軟件不符合用戶的需求，無論代碼寫得多麼優美，它都是毫無用處的。這裡將會介紹一些客戶協作的習慣和技巧，讓客戶一直加入到團隊的開發中，學習他們的業務經驗，並且保證項目符合他們的真正需求。

**第5章：敏捷反饋**。敏捷團隊之所以能夠順利開展工作，而不會陷入泥潭掙扎導致項目失敗，就是因為一直使用反饋來糾正軟件和開發過程。最好的反饋源自代碼本身。本章將研究如何獲得反饋，以及如何更好地控制團隊進程和性能。

**第6章：敏捷編碼**。為滿足將來的需求而保持代碼的靈活和可變性，這是敏捷方法成功的關鍵。本章给出了一些習慣，介紹如何讓代碼更加整潔，具有更好的擴展性，防止代碼慢慢變壞，最後變得不可收拾。

**第7章：敏捷調試**。調試錯誤會佔用很多項目開發的時間——時間是經不起浪費的。這裡將會學到一些提高調試效率的技巧，節省項目的開發時間。

**第8章：敏捷協作**。最後，一個敏捷開發者已經能夠獨擋一面，除此之外，你需要一個敏捷團隊。這裡有一些最有效的實踐有助於黏合整個團隊，以及其他一些實踐有助於團隊日常事務和成長。

## 敏捷工具箱

全書中，我們會涉及一些敏捷項目常用的基本工具。也許一些工具你還很陌生，所以這裡做了簡單介紹。想要了解這些工具的詳細信息，可以進一步去讀附錄中的有關參考文獻。

**Wiki**：Wiki<sup>②</sup>是一個網站，用戶通過瀏覽器，就可以編輯網頁內容並創建鏈接到一個新的內容頁面。**Wiki**是一種很好的支持協作的工具，因為團隊中的每一個人都可以根據需要動態地新增和重新組織

網頁中的內容，實現知識共享。關於Wiki的更多詳情，可查閱《Wiki之道》這篇文章[LC01]。

② Wiki是WikiWikiWeb的簡稱，WikiWiki源自夏威夷語，本意是快點快點。

**版本控制**：項目開發中所有的產物——全部的源代碼、文檔、圖標、構建腳本等，都需要放入版本控制系統中，由版本控制系統來統一管理。令人驚訝的是，很多團隊仍然喜歡把這些文件放到一個網絡上共享的設備上，但這是一種很不專業的做法。如果需要一個安裝和使用版本控制系統的詳細說明，可以查閱《版本控制之道——使用CVS》[TH03]或者《版本控制之道——使用Subversion》[Mas05]。

**單元測試**：用代碼來檢查代碼，這是開發者獲得反饋的主要來源。在本書後面會更多地談到它，但要真正知道框架可以處理大部分的繁瑣工作，讓你把精力放到業務代碼的實現中。想要了解單元測試，可以看《單元測試之道Java版》[HT03]和《單元測試之道C#版》[HT04]，你可以在《JUnit Recipes中文版》[Rai04]一書中找到很多寫單元測試的實用技巧。

**自動構建**：不管是在自己的本地機器上實現構建，還是為整個團隊實現構建，都是全自動化並可重複的。因為這些構建一直運行，所以又稱為持續集成。和單元測試一樣，有大量的免費開源產品和商業產品為你提供支持。《項目自動化之道》[Cla04]介紹了所有自動構建的技巧和訣竅（包括使用Java Lamps）。

最後，*Ship It!* [RG05]一書很好地介紹了怎樣將這些基本的開發環境實踐方法結合到一起。

## 魔鬼和這些討厭的細節

如果你翻翻這本書就會注意到，在每節的開頭我們都會引入一段話，旁邊配有一個魔鬼木刻像，誘使你養成不良習慣，如下所示。

“幹吧，就走那個捷徑。真的，它可以為你節省時間。沒有人會知道是你乾的，這樣你就會加快自己的開發進度，並且能夠完成這些任務了。這就是關鍵所在。”



他的有些話聽上去有點兒荒唐，就像是Scott Adams筆下呆伯特（Dilbert）漫畫書中的魔王——“尖發老闆”所說的話一樣。但要記住Adams先生可是從他那些忠實的讀者中得到很多回饋的。

有些事情看上去就會讓人覺得很怪異，但這全部是我們親耳所聞、親眼所見，或者是大家秘而不宣的事情，它們都是擺在我們面前的誘惑，不管怎樣，只要試過就會知道，為了節省項目的時間而走愚蠢的捷徑是會付出巨大代價的。

與這些誘惑相對，在每個習慣最後，會出現一位守護天使，由她給我們認為你應該遵循的一些良策。



**先難後易**。我們首先要解決困難的問題，把簡單的問題留到最後。

現實中的事情很少是黑白分明的。我們將用一些段落描述一個習慣應該帶給你什麼樣的切身感受，並介紹成功實施和保持平衡的技巧。如下所示。

## 切身感受

本段描述培養某個習慣應該有什麼樣的切身感受。如果在實踐中沒有這樣的體會，你就要考慮改變一下實施的方法。

# 平衡的藝術

- 一個習慣很可能會做得過火或者做得不夠。我們會給出一些建議，幫你掌握平衡，並告訴你一些技巧，能使習慣真正為你所用。

畢竟，一件好事做得過火或者被誤用，都是非常危險的（我們見過很多所謂的敏捷項目最後失敗，都是因為團隊在實踐的時候沒有保持好自己的平衡）。所以，我們希望保證你能真正從這些習慣中獲益。

通過遵循這些習慣，把握好平衡的藝術，在真實世界中有效地應用它們，你將會看到你的項目和團隊發生了積極的變化。

好了，你將步入敏捷開發者的修煉之路，更重要的是，你會理解其後的開發原則。

## 致謝

每本書的出版都是一項艱鉅的事業，本書也不例外。除了作者，還有很多幕後英雄。

我們要感謝下面所有的人，正是他們的幫助，本書才得以問世。

感謝**Jim Moore**為本書設計的封面插圖，感謝**Kim Wimpsett**出色的文字編輯工作（如果還有錯，那肯定是最後一刻的修改導致的）。

最後感謝所有付出時間和精力讓本書變得更好的審閱者：**Marcus Ahnve**、**Eldon Alameda**、**Sergei Anikin**、**Matthew Bass**、**David Bock**、**A. Lester Buck III**、**Brandon Campbell**、**Forrest Chang**、**Mike Clark**、**John Cook**、**Ed Gibbs**、**Dave Goodlad**、**Ramamurthy Gopalakrishnan**、**Marty Haught**、**Jack Herrington**、**Ron Jeffries**、**Matthew Johnson**、**Jason Hiltz Laforge**、**Todd Little**、**Ted Neward**、

James Newkirk、Jared Richardson、Frédéric Ros、Bill Rushmore、David Lázaro Saz、Nate Schutta、Matt Secoske、Guerry Semones、Brian Sletten、Mike Stok、Stephen Viles、Leif Wickland和Joe Winter。

## **Venkat Subramaniam致謝**

我要感謝Dave Thomas，他是我的良師益友。如果沒有他的指導、鼓勵和建設性的意見，本書到現在還只是一個空想。

我有幸與Andy Hunt合著本書，從他身上學到了太多的東西。他不僅是一位技術專家（任何注重實效的程序員都知道這一點），還具有令人難以置信的表達能力和優秀品質。我欣賞Pragmatic Programmers出版公司製作本書的每一個環節，他們精通很多有用的工具，具有解決問題的能力，而且最重要的是，他們有很好的工作態度，正因如此，本書才可以如此順利地發佈。

感謝Marc Garbey的鼓勵。他是一位偉大的朋友，他的幽默和敏捷感染了世界上的很多人。我特別感謝那些與我一路同行的俗人們（錯了，是朋友們）——Ben Galbraith、Brian Sletten、Bruce Tate、Dave Thomas、David Geary、Dion Almaer、Eitan Suez、Erik Hatcher、Glenn Vanderburg、Howard Lewis Ship、Jason Hunter、Justin Gehtland、Mark Richards、Neal Ford、Ramnivas Laddad、Scott Davis、Stu Halloway和Ted Neward——這些傢伙棒極了！我感謝Jay Zimmerman（人稱敏捷推動者），NFJS的主管，感謝他的鼓勵，感謝他給我機會去向他的客戶推廣我的敏捷思想。

感謝父親教會我正確的人生價值觀，還有母親給予我創造的靈感。如果不是妻子Kavitha的耐心和鼓勵，還有我的兒子們Karthik和Krupakar的支持，我就沒有今天的一切。謝謝，我愛你們！

## **Andy Hunt致謝**

好的，我想現在每個人都被感謝過了。但是，我要特別感謝Venkat邀請我參與本書的寫作。我不會接受任何其他人要我合作寫這本書的邀請，但卻接受了Venkat的邀請。他最清楚整件事情的經過。

我要感謝當年在雪鳥聚會的那些敏捷精英們。雖然沒有任何哪一個人發明了敏捷，但正是通過所有人的共同努力，才讓敏捷在當今的軟件開發行業中茁壯成長，成為一支重要的力量。

當然，還要感謝我的家人，感謝他們的支持和理解。從最早的《程序員修煉之道》到現在這本書是一個漫長的征途，但也是一段開心的經歷。

現在，演出開始了。

## 第2章 態度決定一切

選定了要走的路，就是選定了它通往的目的地。

——Harry Emerson Fosdick (美國基督教現代主義神學家)

傳統的軟件開發圖書一般先介紹一個項目的角色配置，然後是你需要產生哪些工件 ( artifact ) ——文檔、任務清單、甘特 ( Gantt ) 圖等，接著就是規則制度，往往是這麼寫的：汝當如此①這般.....本書的風格不是這樣的。歡迎進入敏捷方法的世界，我們的做法有些不同。

① 或更通俗地寫成：系統應當如何如何.....。

例如，有一種相當流行的軟件方法學要求對一個項目分配35種不同的角色，包括架構師、設計人員、編碼人員、文檔管理者等。敏捷方法卻背道而馳。只需要一個角色：軟件開發者，也就是你。項目需要什麼你就做什麼，你的任務就是和緊密客戶協作，一起開發軟件。敏捷依賴人，而不是依賴於項目的甘特圖和里程錶。

圖表、集成開發環境或者設計工具，它們本身都無法產生軟件，軟件是從你的大腦中產生的。而且它不是孤立的大腦活動，還會有許多其他方面的因素：個人情緒、辦公室的文化、自我主義、記憶力等。它們混為一體，**態度** 和 **心情** 的瞬息變化都可能導致巨大的差別。

因此態度非常重要，包括你的和團隊的。專業的態度應該著眼於項目和團隊的積極結果，關注個人和團隊的成長，圍繞最後的成功開展工作。由於很容易變成追求不太重要的目標，所以在本章，我們會專注於那些真正的目標。集中精力，你是為**做事** 而工作。（想知道怎樣做嗎？請見下一頁。）

軟件項目時常伴有時間壓力——壓力會迫使你走捷徑，只看眼前利益。但是，任何一個有經驗的開發者都會告訴你，**欲速則不達**（我們在第15頁將介紹如何避免這個問題）。

我們每個人或多或少都有一些自我主義。一些人（暫且不提他們的名字）還美其名曰“健康”的自我主義。如果要我們去解決一個問題，我們會為完成任務而感到驕傲，但這種驕傲有時會導致主觀和脫離實際。你也很可能見過設計方案的討論變成了人身攻擊，而不是就事論事地討論問題。**對事不對人**（第18頁）會讓工作更加有效。

反饋是敏捷的基礎。一旦你意識到走錯了方向，就要立即做出決策，改變方向。但是指出問題往往沒有那麼容易，特別當它涉及一些政治因素的時候。有時候，你需要勇氣去**排除萬難，奮勇前進**（第23頁）。

只有在你對項目、工作、事業有一個專業的態度時，使用敏捷方法才會生效。如果態度不正確，那麼所有的這些習慣都不管用。有了正確的態度，你才可以從這些方法中完全受益。下面我們就來介紹這些對你大有裨益的習慣和建議。

## 1 做事



“出了問題，第一重要的是確定元兇。找到那個白痴！一旦證實了是他的錯誤，就可以保證這樣的問題永遠不會再發生了。”



有時候，這個老魔頭的話聽起來似乎很有道理。毫無疑問，你想把尋找罪魁禍首設為最高優先級，難道不是嗎？肯定的答案是：不。最高優先級應該是解決問題。

也許你不相信，但確實有些人常常不把解決問題放在最高優先級上。也許你也沒有。先自我反省一下，當有問題出現時，“第一”反應究竟是什麼。

如果你說的話只是讓事態更複雜，或者只是一味地抱怨，或者傷害了他人的感情，那麼你無意中在給問題火上澆油。相反，你應該另闢蹊徑，問問“為了解決或緩解這個問題，我能夠做些什麼？”在敏捷的團隊中，大家的重點是做事。你應該把重點放到解決問題上，而不是在指責犯錯者上面糾纏。

## 指責不能修復bug

### Blame doesn't fix bugs

世上最糟糕的工作（除了在馬戲團跟在大象後面打掃衛生）就是和一群愛搬弄是非的人共事。他們對解決問題並沒有興趣，相反，他們愛在別人背後議論是非。他們挖空心思指手畫腳，議論誰應該受到指責。這樣一個團隊的生產力是極其低下的。如果你發現自己是在這樣的團隊中工作，不要從團隊中走開——應該跑開。至少要把對話從負面的指責遊戲引到中性的話題，比如談論體育運動（紐約揚基隊最近怎麼樣）或者天氣。

在敏捷團隊中，情形截然不同。如果你向敏捷團隊中的同事抱怨，他們會說：“好，我能幫你做些什麼？”他們把精力直接放到解決問題上，而不是抱怨。他們的動機很明確，重點就是做事，不是為了自己的面子，也不是為了指責，也無意進行個人智力角鬥。

你可以從自己先做起。如果一個開發者帶著抱怨或問題來找你，你要了解具體的問題，詢問他你能提供什麼樣的幫助。這樣簡單的一個行為就清晰地表明你的目的是解決問題，而不是追究責任，這樣就會消除他的顧慮。你是給他們幫忙的。這樣，他們會知道每次走近你的時候，你會真心幫助他們解決問題。他們可以來找你把問題解決了，當然還可以繼續去別處求助。

## 符合標準不是結果

許多標準化工作強調遵從一個過程，按符合的程度作評判，其理由是：如果過程可行，那麼只要嚴格按這個過程行事，就不會有問題。

但是，現實世界並不是如此運行的。你可以去獲得ISO-9001認證，並生產出一件漂亮的鉛線織就的救生衣。你完全遵循了文檔中約定的過程，糟糕的是到最後所有的用戶都被淹死了。

過程符合標準並不意味結果是正確的。敏捷團隊重結果勝於重過程。

如果你找人幫忙，卻沒有人積極響應，那麼你應該主動引導對話。解釋清楚你想要什麼，並清晰地表明你的目的是解決問題，而不是指責他人或者進行爭辯。



**指責不會修復bug**。把矛頭對準問題的解決辦法，而不是人。這是真正有用處的正面效應。

## 切身感受

勇於承認自己不知道答案，這會讓人感覺放心。一個重大的錯誤應該被當作是一次學習而不是指責他人的機會。團隊成員們在一起工作，應互相幫助，而不是互相指責。

## 平衡的藝術

- “這不是我的錯”，這句話不對。“這都是你的錯”，這句話更不對。
- 如果你沒有犯過任何錯誤，就說明你可能沒有努力去工作。
- 開發者和質量工程師（QA）爭論某個問題是系統本身的缺陷還是系統增強功能導致的，通常沒有多大的意義。與其如此，不如趕緊去修復它。
- 如果一個團隊成員誤解了一個需求、一個API調用，或者最近一次會議做的決策，那麼，也許就意味著團隊的其他成員也有相同的誤解。要確保整個團隊儘快消除誤解。
- 如果一個團隊成員的行為一再傷害了團隊，則他表現得很不職業。那麼，他就不是在幫助團隊向解決問題的方向前進。這種情況下，我們必須要求他離開這個團隊①。

① 不需要解僱他，但是他不能繼續留在這個項目中。同時也要意識到，頻繁的人員變動對整個團隊的平衡也很危險。

- 如果大部分團隊成員（特別是開發領導者）的行為都不職業，並且他們對團隊目標都不感興趣，你就應該主動從這個團隊中離開，尋找更適合自己發展的團隊（這是一個有遠見的想法，沒必要眼睜睜地看著自己陷入一個“死亡之旅”的項目中[You99]）。

## 2 欲速則不達

“你不需要真正地理解那塊代碼，它只要能夠工作就可以了。哦，它需要一個小小的調整。只要在結果中再加上幾行代碼，它就可以工作了。幹吧！就把那幾行代碼加進去，它應該可以工作。”



我們經常會遇到這種情況，出現了一個bug，並且時間緊迫。快速修復確實可以解決它——只要新加一行代碼或者忽略那個列表上的最後一個條目，它就可以工作了。但接下來的做法才能說明，誰是優秀的程序員，誰是拙劣的代碼工人。

拙劣的代碼工人會這樣不假思索地改完代碼，然後快速轉向下一個問題。

優秀的程序員會挖掘更深一層，盡力去理解為什麼這裡必須要加1，更重要的是，他會想明白會產生什麼其他影響。

也許這個例子聽起來有點做作，甚至你會覺得很無聊。但是，真實世界中有大量這樣的事情發生。**Andy**以前的一個客戶正遇到過這樣的問題。沒有一個開發者或者架構師知道他們業務領域的底層數據模型。而且，通過幾年的積累，代碼裡有著成千上萬的+1和-1修正。在這樣髒亂的代碼中添加新的功能或者修復bug，就難逃脫髮的噩運（事實上，很多開發者因此而禿頂）。

千里之堤，潰於蟻穴，大災難是逐步演化來的。一次又一次的快速修復，每一次都不探究問題的根源，久而久之就形成了一個危險的沼澤地，最終會吞噬整個項目的生命。

## 防微杜漸

### **Beware of land mines**

在工作壓力之下，不去深入瞭解真正的問題以及可能的後果，就快速修改代碼，這樣只是解決表面問題，最終會引發大問題。快速修復的誘惑，很容易令人把持不住，墜入其中。短期看，它似乎是有效的。但從長遠來看，它無異於穿越一片流沙，你也許僥倖走過了一半的路程（甚至更遠），一切似乎都很正常。但是轉眼間悲劇就發生了.....

只要我們繼續進行快速修復，代碼的清晰度就不斷降低。一旦問題累積到一定程度，清晰的代碼就不復存在，只剩一片混濁。很可能在你

的公司就有人這樣告訴你：“無論如何，千萬不能碰那個模塊的代碼。寫代碼那哥們兒已經不在這兒了，沒有人看得懂他的代碼。”這些代碼根本沒有清晰度可言，它已經成為一團迷霧，無人能懂。

## **Andy如是說.....**

### **要理解開發過程**

儘管我們在談論理解代碼，特別是在修改代碼之前一定要很好地理解它，然而同樣道理，你也需要了解團隊的開發方法或者開發過程。

你必須理解團隊採用的開發方法。你必須理解如何恰如其分地使用這種方法，為何它們是這樣的，以及如何成為這樣的。

只有理解了這些問題，你才能進行有效的改變。

如果在你的團隊中有這樣的事情發生，那麼你是不可能敏捷的。但是敏捷方法中的一些技術可以阻止這樣的事情發生。這裡只是一些概述，後面的章節會有更深入的介紹。

### **不要孤立地編碼**

### **Don't code in isolation**

孤立非常危險，不要讓開發人員完全孤立地編寫代碼（見第155頁，習慣40）。如果團隊成員花些時間閱讀其他同事寫的代碼，他們就能確保代碼是可讀和可理解的，並且不會隨意加入這些“+1或-1”的代碼。閱讀代碼的頻率越高越好。實行**代碼複審**，不僅有助於代碼更好地理解，而且是發現bug最有效的方法之一（見第165頁，習慣44）。

### **使用單元測試**

### **Use unit tests**

另一種防止代碼難懂的重要技術就是單元測試。單元測試幫助你很自然地把代碼分層，分成很多可管理的小塊，這樣就會得到設計更好、更清晰的代碼。更深入項目的時候，你可以直接閱讀單元測試——它們是一種可執行的文檔（見第78頁，習慣19）。有了單元測試，你會看到更小、更易於理解的代碼模塊，運行和使用代碼，能夠幫助你徹底理解這些代碼。



**不要墜入快速的簡單修復之中。** 要投入時間和精力保持代碼的整潔、敞亮。

## 切身感受

在項目中，代碼應該是很亮堂的，不應該有黑暗死角。你也許不知道每塊代碼的每個細節，或者每個算法的每個步驟，但是你對整體的相關知識有很好的瞭解。沒有任何一塊代碼被警戒線或者“切勿入內”的標誌隔離開。


## 平衡的藝術

- 你必須要理解一塊代碼是如何工作的，但是不一定需要成為一位專家。只要你能使用它進行有效的工作就足夠了，不需要把它當作畢生事業。
- 如果有一位團隊成員宣佈，有一塊代碼其他人都很難看懂，這就意味著任何人（包括原作者）都很難維護它。請讓它變得簡單些。
- 不要急於修復一段沒能真正理解的代碼。這種+1/-1的病症始於無形，但是很快就會讓代碼一團糟。要解決真正的問題，不要治標不治本。
- 所有的大型系統都非常複雜，因此沒有一個人能完全明白所有的代碼。除了深入瞭解你正在開發的那部分代碼之外，你還需要從

更高的層面來了解大部分代碼的功能，這樣就可以理解系統各個功能塊之間是如何交互的。

- 如果系統的代碼已經惡化，可以閱讀第23頁習慣4中給出的建議。

## 3 對事不對人

你在這個設計上投入了很多精力，為它付出很多心血。你堅信它比其他任何人的設計都棒。別聽他們的，他們只會把問題變得更糟糕。” 

你很可能見過，對方案設計的討論失控變成了情緒化的指責——做決定是基於誰提出了這個觀點，而不是權衡觀點本身的利弊。我們曾經參與過那樣的會議，最後鬧得大家都很不愉快。

但是，這也很正常。當Lee先生在做一個新方案介紹的時候，下面有人會說：“那樣很蠢！”（這也就暗示著Lee先生也很蠢。）如果把這句話推敲一下，也許會好一點：“那樣很蠢，你忘記考慮它要線程安全。”事實上最適合並且最有效的表達方式應該是：“謝謝，Lee先生。但是我想知道，如果兩個用戶同時登錄會發生什麼情況？”

看出其中的不同了吧！下面我們來看看對一個明顯的錯誤有哪些常見的反應。

- 否定個人能力。
- 指出明顯的缺點，並否定其觀點。
- 詢問你的隊友，並提出你的顧慮。

第一種方法是不可能成功的。即使Lee是一個十足的笨蛋，很小的問題也搞不定，但你那樣指出問題根本不會對他的水平有任何提高，反

而會導致他以後再也不會提出自己的任何想法了。第二種方法至少觀點明確，但也不能給Lee太多的幫助，甚至可能會讓你自己惹火上身。也許Lee能巧妙地回覆你對非線程安全的指責：“哦，不過它不需要多線程。因為它只在Frozbots模塊的環境中使用，它已經運行在自己的線程中了。”哎喲！忘記了Frozbots這一茬了。現在**該是你**覺得自己蠢了，Lee也會因為你罵他笨蛋而生氣。

現在看看第三種方法。沒有譴責，沒有評判，只是簡單地表達自己的觀點。讓Lee自己意識到這個問題，而不是掃他的面子<sup>①</sup>。由此可以開始一次交談，而不是爭辯。

① 通常，這是一個很好的技巧：引導性地提出一個疑問，讓他們自己意識到問題。

## **Venkat如是說.....**

### **要專業而不是自我**

多年以前，在我擔任系統管理員的第一天，一位資深的管理員和我一起安裝一些軟件，我突然按錯了一個按鈕，把服務器給關掉了。沒過幾分鐘，幾位不爽的用戶就在敲門了。

這時，我的導師贏得了我的信任和尊重，他並沒有指責我，而是對他們說：“對不起，我們正在查找是什麼地方出錯了。系統會在幾分鐘之內啟動起來。”這讓我學到了難忘的重要一課。

在一個需要緊密合作的開發團隊中，如果能稍加註意禮貌對待他人，將會有益於整個團隊關注真正有價值的問題，而不是勾心鬥角，誤入歧途。我們每個人都能有一些極好的創新想法，同樣也會萌生一些很愚蠢的想法。

如果你準備提出一個想法，卻擔心有可能被嘲笑，或者你要提出一個建議，卻擔心自己丟面子，那麼你就不會主動提出自己的建議了。然



而，好的軟件開發作品和好的軟件設計，都需要大量的創造力和洞察力。分享並融合各種不同的想法和觀點，遠遠勝於單個想法為項目帶來的價值。

## 消極扼殺創新

### Negativity kills innovation

負面的評論和態度扼殺了創新。現在，我們並不提倡在設計方案的會議上手拉手唱《學習雷鋒好榜樣》，這樣也會降低會議的效率。但是，你必須把重點放在解決問題上，而不是去極力證明誰的主意更好。在團隊中，一個人只是智商高是沒有用的，如果他還很頑固並且拒絕合作，那就更糟糕。在這樣的團隊中，生產率和創新都會頻臨滅亡的邊緣。

我們每個人都會有好的想法，也會有不對的想法，團隊中的每個人都需要自由地表達觀點。即使你的建議不被全盤接受，也能對最終解決問題有所幫助。不要害怕受到批評。記住，任何一個專家都是從這裡開始的。用Les Brown<sup>②</sup>的一句話說就是：“你不需要很出色才能起步，但是你必须起步才能變得很出色。”

② 萊斯·布朗，全球領軍勵志演講家和作家。——編者注

## 團體決策的駱駝

集體決策確實非常有效，但也有一些最好的創新源於很有見地的個人的獨立思考。如果你是一個有遠見的人，就一定要特別尊重別人的意見。你是一個掌舵者，一定要把握方向，深思熟慮，吸取各方的意見。

另一個極端是缺乏生氣的委員會，每個設計方案都需要全票通過。這樣的委員會總是小題大作，如果讓他們造一匹木馬，很可能最後造出的是駱駝。

我們並不是建議你限制會議決策，只是你不應該成為一意孤行的首席架構師的傀儡。這裡建議你牢記亞里士多德的一句格言：“能容納自己並不接受的想法，表明你的頭腦足夠有學識。”

下面是一些有效的特殊技術。

**設定最終期限**。如果你正在參加設計方案討論會，或者是尋找解決方案時遇到問題，請設定一個明確的最終期限，例如午飯時間或者一天的結束。這樣的時間限制可以防止人們陷入無休止的理論爭辯之中，保證團隊工作的順利進行。同時（我們覺得）應現實一些：沒有**最好**的答案，只有更合適的方案。設定期限能夠幫你在為難的時候果斷做出決策，讓工作可以繼續進行。

**逆向思維**。團隊中的每個成員都應該意識到權衡的必要性。一種客觀對待問題的辦法是：先是積極地看到它的正面，然後再努力地從反面去認識它<sup>③</sup>。目的是要找出優點最多缺點最少的那個方案，而這個好辦法可以儘可能地發現其優缺點。這也有助於少帶個人感情。

③ 參見“Debating with Knives”，在<http://blogs.pragprog.com/cgi-bin/pragdave.cgi/Random/FishBow1.rdoc>。

**設立仲裁人**。在會議的開始，選擇一個仲裁人作為本次會議的決策者。每個人都要有機會針對問題暢所欲言。仲裁人的責任就是確保每個人都有發言的機會，並維持會議的正常進行。仲裁人可以防止明星員工操縱會議，並及時打斷假大空式發言。

如果你自己沒有積極參與這次討論活動，那麼你最好退一步做會議的監督者。仲裁人應該專注於調停，而不是發表自己的觀點（理想情況下不應在整個項目中有既得利益）。當然，這項任務不需要嚴格的技术技能，需要的是和他人打交道的能力。

**支持已經做出的決定**。一旦方案被確定了（不管是什麼樣的方案），每個團隊成員都必須通力合作，努力實現這個方案。每個人都要時刻

記住，我們的目標是讓項目成功滿足用戶需求。客戶並不關心這是誰的主意——他們關心的是，這個軟件是否可以工作，並且是否符合他們的期望。結果最重要。

設計充滿了妥協（生活本身也是如此），成功屬於意識到這一點的團隊。工作中不感情用事是需要剋制力的，而你若能展現出成熟大度來，大家一定不會視而不見。這需要有人帶頭，身體力行，去感染另一部分人。



**對事不對人**。讓我們驕傲的應該是解決了問題，而不是比較出誰的主意更好。

## 切身感受

一個團隊能夠很公正地討論一些方案的優點和缺點，你不會因為拒絕了有太多缺陷的方案而傷害別人，也不會因為採納了某個不甚完美（但是更好的）解決方案而被人忌恨。

## 平衡的藝術

- 盡力貢獻自己的好想法，如果你的想法沒有被採納也無需生氣。不要因為只是想體現自己的想法而對擬定的好思路畫蛇添足。
- 脫離實際的反方觀點會使爭論變味。若對一個想法有成見，你很容易提出一堆不太可能發生或不太實際的情形去批駁它。這時，請先捫心自問：類似問題以前發生過嗎？是否經常發生？
- 也就是說，像這樣說是不夠的：我們不能採用這個方案，因為數據庫廠商可能會倒閉。或者：用戶絕對不會接受那個方案。你必須要評判那些場景發生的可能性有多大。想要支持或者反駁一個觀點，有時候你必須先做一個原型或者調查出它有多少的同意者或者反對者。

- 在開始尋找最好的解決方案之前，大家對“**最好**”的含義要先達成共識。在開發者眼中的最好，不一定就是用戶認為最好的，反之亦然。
- 只有**更好**，沒有**最好**。儘管“最佳實踐”這個術語到處在用，但實際上不存在“最佳”，只有在某個特定條件下更好的實踐。
- 不帶個人情緒並不是要盲目地接受所有的觀點。用合適的詞和理由去解釋為什麼你不贊同這個觀點或方案，並提出明確的問題。

## 4 排除萬難，奮勇前進

“如果你發現其他人的代碼有問題，只要你自己心裡知道就可以了。畢竟，你不想傷害他們，或者惹來麻煩。如果他是你的老闆，更要格外謹慎，只要按照他的命令執行就可以了。”



有一則寓言叫“誰去給貓系鈴鐺”(Who Will Bell the Cat)。老鼠們打算在貓的脖子上系一個鈴鐺，這樣貓巡邏靠近的時候，就能預先得到警報。每隻老鼠都點頭，認為這是一個絕妙的想法。這時一隻年老的老鼠問道：“那麼，誰願意挺身而出去系鈴鐺呢？”毫無疑問，沒有一隻老鼠站出來。當然，計劃也就這樣泡湯了。

有時，絕妙的計劃會因為勇氣不足而最終失敗。儘管前方很危險——不管是真的魚雷或者只是一個比喻——你必須有勇氣向前衝鋒，做你認為對的事情。

假如要你修復其他人編寫的代碼，而代碼很難理解也不好使用。你是應該繼續修復工作，保留這些髒亂的代碼呢，還是應該告訴你的老闆，這些代碼太爛了，應該通通扔掉呢？

也許你會跳起來告訴周圍的人，那些代碼是多麼糟糕，但那只是抱怨和發洩，並不能解決問題。相反，你應該重寫這些代碼，並比較重寫前後的優缺點。動手證明（不要只是嚷嚷）最有效的方式，是把糟糕

的代碼放到一邊，立刻重寫。列出重寫的理由，會有助於你的老闆（以及同事）認清當前形勢，幫助他們得到正確的解決方案。

再假定你在處理一個特定的組件。突然，你發現完全弄錯了，你需要推翻重來。當然，你也會很擔心向團隊其他成員說明這個問題，以爭取更多的時間和幫助。

當發現問題時，不要試圖掩蓋這些問題。而要有勇氣站起來，說：“我現在知道了，我過去使用的方法不對。我想到了一些辦法，可以解決這個問題——如果你有更好的想法，我也很樂意聽一聽——但可能會花多些時間。”你已經把所有對問題的負面情緒拋諸腦後，你的意圖很清楚，就是尋找解決方案。既然你提出大家一起努力來解決問題，那就不會有任何爭辯的餘地。這樣會促進大家去解決問題。也許，他們就會主動走近，提供幫助。更重要的是，這顯示出了你的真誠和勇氣，同時你也贏得了他們的信任。

## **Venkat如是說.....**

### **踐行良好習慣**

我曾經開發過一個應用系統。它向服務器程序發送不同類型的文件，再另存為另外一種格式的文件。這應該不難。當我開始工作的時候，我震驚地發現，處理每種類型文件的代碼都是重複的。所以，我也配合了一下，複製了數百行的代碼，改變了其中的兩行代碼，幾分鐘之內就讓它工作起來，但我卻感覺很失落。因為我覺得這有悖於良好的工作習慣。

後來我說服了老闆，告訴他代碼的維護成本很快就會變得非常高，應該重構代碼。一週之內，我們重構了代碼，並立即由此受益，我們需要修改文件的處理方式，這次我們只需要改動一個地方就可以了，而不必遍查整個系統。

你深知怎樣做才是正確的，或者至少知道目前的做法是錯誤的。要有勇氣向其他的項目成員、老闆或者客戶解釋你的不同觀點。當然，這並不容易。也許你會拖延項目的進度，冒犯項目經理，甚至惹惱投資人。但你都要不顧一切，向著正確的方向奮力前進。

美國南北戰爭時的海軍上將**David Farragut**曾經說過一句名言：“別管他媽的魚雷，**Drayton**上校，全速前進。”確實，前面埋伏著水雷（那時叫魚雷），但是要突破防線，只有全速前進①。

① 事實上，**Farragut**的原話往往被簡化為：“別管他媽的魚雷，全速前進！”

他們做得很對！



**做正確的事**。要誠實，要有勇氣去說出實情。有時，這樣做很困難，所以我們要有足夠的勇氣。

## 切身感受

勇氣會讓人覺得有點不自在，提前鼓足勇氣更需要魄力。但有些時候，它是掃除障礙的唯一途徑，否則問題就會進一步惡化下去。鼓起你的勇氣，這能讓你從恐懼中解脫出來。

## 平衡的藝術

- 如果你說天快要塌下來了，但其他團隊成員都不贊同。反思一下，也許你是正確的，但你沒有解釋清楚自己的理由。
- 如果你說天快要塌下來了，但其他團隊成員都不贊同。認真考慮一下，他們也許是對的。
- 如果設計或代碼中出現了奇怪的問題，花時間去理解為什麼代碼會是這樣的。如果你找到了解決辦法，但代碼仍然令人費解，唯

一的解決辦法是重構代碼，讓它可讀性更強。如果你沒有馬上理解那段代碼，不要輕易地否定和重寫它們。那不是勇氣，而是魯莽。

- 當你勇敢地站出來時，如果受到了缺乏背景知識的抉擇者的抵制，你需要用他們能夠聽懂的話語表達。“更清晰的代碼”是無法打動生意人的。節約資金、獲得更好的投資回報，避免訴訟以及增加用戶利益，會讓論點更有說服力。
- 如果你在壓力下要對代碼質量作出妥協，你可以指出，作為一名開發者，你沒有職權毀壞公司的資產（所有的代碼）。

## 第3章 學無止境

即使你已經在正確的軌道上，但如果只是停止不前，也仍然會被淘汰出局。

——**Will Rogers**（美國著名演員）

敏捷需要持續不斷的學習和充電。正如上面引用的**Will Rogers**的話，逆水行舟，不進則退。那不僅是賽馬場上的真理，它更適合我們當今的程序員。

軟件開發行業是一個不停發展和永遠變化的領域。雖然有一些概念一直有用，但還有很多知識很快就會過時。從事軟件開發行業就像是在跑步機上，你必須一直跟上步伐穩步前進，否則就會摔倒出局。

誰會幫助你保持步伐前進呢？在一個企業化的社會中，只有一個人會為你負責——你自己。是否能跟上變化，完全取決於你自己。

許多新技術都基於現有的技術和思想。它們會加入一些新的東西，這些新東西是逐步加入的量。如果你跟蹤技術變化，那麼學習這些新東西對你來說就是了解這些增量變化。如果你不跟蹤變化，技術變化就會顯得很突然並且難以應付。這就好比少小離家老大回，你會發現變化很大，甚至有很多地方都不認識了。然而，居住在那裡的人們，每天只看到小小的變化，所以非常適應。在第28頁我們會介紹一些**跟蹤變化**的方法。

給自己投資，讓自己與時俱進，當然再好不過，但是也要努力**對團隊投資**，這個目標怎麼實現呢？你將從第31頁學到實現這個目標的一些方法。

學習新的技術和新的開發方法很重要，同時你也要能摒棄陳舊和過時的開發方法。換句話說，你需要**懂得丟棄**（請閱讀第34頁）。

當我們談到變化這個話題的時候，要認識到你對問題的理解在整個項目期間也是在變化的。你曾經認為自己已經很明白的事情，現在也許並不是你想象中那樣。你要對沒有完全理解的某些疑問不懈地深入追蹤下去，我們將從第37頁開始講述為什麼要**打破砂鍋問到底**，以及如何有效地提問。

最後，一個活力十足的敏捷開發團隊需要有規律反覆地做很多事情，一旦項目開始運作，你就要**把握開發節奏**，我們會在第40頁介紹這種節奏感。

## 5 跟蹤變化

“軟件技術的變化如此之快，勢不可擋，這是它的本性。繼續用你熟悉的語言做你的老本行吧，你不可能跟上技術變化的腳步。”





赫拉克利特說過：“唯有變化是永恆的。”歷史已經證明了這句真理，在當今快速發展的IT時代尤其如此。你從事的是一項充滿激情且不停變化的工作。如果你畢業於計算機相關的專業，並覺得自己已經學完了所有知識，那你就大錯特錯了。

假設你是10多年前的1995年畢業的，那時，你掌握了哪些技術呢？可能你的C++還學得不錯，你瞭解有一門新的語言叫Java，一種被稱作是設計模式的思想開始引起大家的關注。一些人會談論被稱作因特網的東東。如果那個時候你就不再學習，而在2005年的時候重出江湖。再看看周圍，就會發現變化巨大。就算是在一個相當狹小的技術領域，要學習那些新技術並達到熟練的程度，一年的時間也不夠。

技術發展的步伐如此快速，簡直讓人們難以置信。就以Java為例，你掌握了Java語言及其一系列的最新特性。接著，你要掌握Swing、Servlet、JSP、Struts、Tapestry、JSF、JDBC、JDO、Hibernate、JMS、EJB、Lucene、Spring.....還可以列舉很多。如果你使用的是微軟的技術，要掌握VB、Visual C++、MFC、COM、ATL、.NET、C#、VB.NET、ASP.NET、ADO.NET、WinForm、Enterprise Service、Biztalk.....並且，不要忘記還有UML、Ruby、XML、DOM、SAX、JAXP、JDOM、XSL、Schema、SOAP、Web Service、SOA，同樣還可以繼續列舉下去（我們將會用光所有的縮寫字母）。

不幸的是，如果只是掌握了工作中需要的技術並不夠。那樣的工作也許幾年之後就不再有了——它會被外包或者會過時，那麼你也將會出局<sup>①</sup>。

① 參考*My Job Went to India: 52 Ways to Save Your Job* [Fow05] 一書。新版改名為*Passionate Programmer*。

假設你是Visual C++或者VB程序員，看到COM技術出現了。你花時間去學習它（雖然很痛苦），並且隨時瞭解分佈式對象計算的一切。當XML出現的時候，你花時間學習它。你深入研究ASP，熟知如何用它來開發Web應用。你雖然不是這些技術的專家，但也不是對它們一無

所知。好奇心促使你去了解MVC是什麼，設計模式是什麼。你會使用一點Java，去試試那些讓人興奮的功能。

如果你跟上了這些新技術，接下來學習.NET技術就不再是大問題。你不需要一口氣爬上10樓，而需要一直在攀登，所以最後看起來就像只要再上一二層。如果你對所有這些技術都一無所知，想要馬上登上這10樓，肯定會讓你喘不過氣來。而且，這也會花很長時間，期間還會有更新的技術出現。

如何才能跟上技術變化的步伐呢？幸好，現今有很多方法和工具可以幫助我們繼續充電。下面是一些建議。

**迭代和增量式的學習**。每天計劃用一段時間來學習新技術，它不需要很長時間，但需要經常進行。記下那些你想學習的東西——當你聽到一些不熟悉的術語或者短語時，簡要地把它記錄下來。然後在計劃的時間中深入研究它。

**瞭解最新行情**。互聯網上有大量關於學習新技術的資源。閱讀社區討論和郵件列表，可以瞭解其他人遇到的問題，以及他們發現的很酷的解決方案。選擇一些公認的優秀技術博客，經常去讀一讀，以瞭解那些頂尖的博客作者們正在關注什麼（最新的博客列表請參考 [pragmaticprogrammer.com](http://pragmaticprogrammer.com)）。

**參加本地的用戶組活動**。Java、Ruby、Delphi、.NET、過程改進、面向對象設計、Linux、Mac，以及其他的各種技術在很多地區都會有用戶組。聽講座，然後積極加入到問答環節中。

**參加研討會議**。計算機大會在世界各地舉行，許多知名的顧問或作者主持研討會或者課程。這些聚會是向專家學習的最直接的好機會。

**如飢似渴地閱讀**。找一些關於軟件開發和非技術主題的好書（我們很樂意為你推薦），也可以是一些專業的期刊和商業雜誌，甚至是一些

大眾媒體新聞（有趣的是在那裡常常能看到老技術被吹捧為最新潮流）。



**跟蹤技術變化**。你不需要精通所有技術，但需要清楚知道行業的動向，從而規劃你的項目和職業生涯。

## 切身感受

你能嗅到將要流行的新技術，知道它們已經發布或投入使用。如果必須要把工作切換到一種新的技術領域，你能做到。

## 平衡的藝術

- 許多新想法從未變得羽翼豐滿，成為有用的技術。即使是大型、熱門和資金充裕的項目也會有同樣的下場。你要正確把握自己投入的精力。
- 你不可能精通每一項技術，沒有必要去做這樣的嘗試。只要你在某些方面成為專家，就能使用同樣的方法，很容易地成為新領域的專家。
- 你要明白為什麼需要這項新技術——它試圖解決什麼樣的問題？它可以被用在什麼地方？
- 避免在一時衝動的情況下，只是因為想學習而將應用切換到新的技術、框架或開發語言。在做決策之前，你必須評估新技術的優勢。開發一個小的原型系統，是對付技術狂熱者的一劑良藥。

# 6 對團隊投資

“不要和別人分享你的知識——自己留著。你是因為這些知識而成為團隊中的佼佼者，只要自己聰明就可以了，不用管其他失敗



者。”

團隊中的開發者們各有不同的能力、經驗和技術。每個人都各有所長。不同才能和背景的人混在一起，是一個非常理想的學習環境。

在一個團隊中，如果只是你個人技術很好還遠遠不夠。如果其他團隊成員的知識不夠，團隊也無法發揮其應有的作用：一個學習型的團隊才是較好的團隊。

當開發項目的時候，你需要使用一些術語或者隱喻來清晰地傳達設計的概念和意圖。如果團隊中的大部分成員不熟悉這些，就很難進行高效地工作。再比如你參加了一個課程或者研討班之後，所學的知識如果不用，往往就會忘記。所以，你需要和其他團隊成員分享所學的知識，把這些知識引入團隊中。

找出你或團隊中的高手擅長的領域，幫助其他的團隊成員在這些方面迎頭趕上（這樣做還有一個好處是，可以討論如何將這些東西應用於自己的項目中）。

“午餐會議”是在團隊中分享知識非常好的方式。在一週之中挑選一天，例如星期三（一般來說任何一天都可以，但最好不要是星期一和星期五）。事先計劃午餐時聚集在一起，這樣就不會擔心和其他會議衝突，也不需要特別的申請。為了降低成本，就讓大家自帶午餐。

每週，要求團隊中的一個人主持講座。他會給大家介紹一些概念，演示工具，或者做團隊感興趣的任何一件事情。你可以挑一本書，給大家說說其中一些特別內容、項目或者實踐。①無論什麼主題都可以。

① Pragmatic公司的出版人Andy和Dave曾聽不少人說，他們成立了讀書小組，討論和研究Pragmatic公司的圖書。

---

每個人都比你厲害嗎？嗯，那太好了！

享有盛名的爵士吉他手Pat Methany說過這樣一句話：“總是要成為你所在的那個樂隊中最差的樂手。如果你是樂隊中最好的樂手，就需要重新選擇樂隊了。我認為這也適用於樂隊之外的其他事情。”

為什麼是這樣呢？如果你是團隊中最好的隊員，就沒有動力繼續提高自己。如果周圍的人都比你厲害，你就會有很強的動力去追趕他們。你將會在這樣的遊戲中走向自己的頂峰。

從每週主持講座的人開始，先讓他講15分鐘，然後，進行開放式討論，這樣每個人都可以發表自己的意見，討論這個主題對於項目的意義。討論應該包括所能帶來的益處，提供來自自己應用程序的示例，並準備好聽取進一步的信息。

這些午餐會議非常有用。它促進了整個團隊對這個行業的瞭解，你自己也可以從其他人身上學到很多東西。優秀的管理者會重用那些能提高其他團隊成員價值的人，因此這些活動也直接有助於你的職業生涯。



**提供你和團隊學習的更好平臺**。通過午餐會議可以增進每個人的知識和技能，並幫助大家聚集在一起進行溝通交流。喚起人們對技術和技巧的激情，將會對項目大有裨益。

## 切身感受

這樣做，會讓每個人都覺得自己越來越聰明。整個團隊都要了解新技術，並指出如何使用它，或者指出需要注意的缺陷。

## 平衡的藝術

- 讀書小組逐章一起閱讀一本書，會非常有用，但是要選好書。  
《7天用設計模式和UML精通.....》也許不會是一本好書。

- 不是所有的講座都能引人入勝，有些甚至顯得不合時宜。不管怎麼樣，都要未雨綢繆；諾亞在建造方舟的時候，可並沒有開始下雨，誰能料到後來洪水氾濫呢？
- 儘量讓講座走入團隊中。如果午餐會議在禮堂中進行，有餐飲公司供飯，還要使用幻燈片，那麼就會減少大家接觸和討論的機會。
- 堅持有計劃有規律地舉行講座。持續、小步前進才是敏捷。稀少、間隔時間長的馬拉松式會議非敏捷也。
- 如果一些團隊成員因為吃午飯而缺席，用美食引誘他們。
- 不要侷限於純技術的圖書和主題，相關的非技術主題（項目估算、溝通技巧等）也會對團隊有幫助。
- 午餐會議不是設計會議。總之，你應專注討論那些與應用相關的一般主題。具體的設計問題，最好是留到設計會議中去解決。

## 7 懂得丟棄

“那就是你一貫的工作方法，並且是有原因的。這個方法也很好地為你所用。開始你就掌握了這個方法，很明顯它是最好的方法。真的，從那以後就不要再改變了。”



敏捷的根本之一就是擁抱變化。既然變化是永恆的，你有可能一直使用相同的技術和工具嗎？

不，不可能。我們一直在本章說要學習新技術和新方法。但是記住，你也需要學會如何丟棄。

隨著科技進步，曾經非常有用的東西往往會靠邊站。它們不再有用了，它們還會降低你的效率。當Andy第一次編程的時候，內存佔用是

一個大問題。你通常無法在主存儲器（大約**48KB**）中一次裝載整個程序，所以必須把程序切分成塊。當一個程序塊交換進去的時候，其他一些程序塊必須出來，並且你無法在一個塊中調用另一個塊中的函數。

正是這種實際約束，極大地影響了你的設計和編程技術。

想想在過去，面對處理器之外的循環操作，你必須花費很大精力去手工調整彙編語言的編譯輸出。可以想象，如果是使用**JavaScript**或者**J2EE**代碼，你還需要這麼幹嗎？

對於大多數的商業應用，技術已經有了巨大的變化，不再像過去那樣，處處考慮內存佔用、手動的重複佔位及手工調整彙編語言。<sup>①</sup> 但我們仍然看到很多開發者從未丟棄這些舊習慣。

① 這些技術現在仍然用於嵌入式系統領域的開發。

**Andy**曾經看到過這樣一段**C**語言代碼：一個大的**for** 循環，循環裡面的代碼一共輸出了**60**頁。那個作者“不相信”編譯器的優化，所以決定自己手工實現循環體展開和其他一些技巧。我們只能祝願維護那一大堆代碼的人好運。

在過去，這段代碼也許可以勉強接受。但是，現在絕對不可以了。電腦和**CPU**曾經非常昂貴，而現在它們就是日用品。現在，開發者的時間才是緊缺和昂貴的資源。

這樣的轉變在緩慢地進行著，但是人們也真正認清了這個事實。我們看到，需要耗費**10**人年開發的**J2EE**項目已經從輝煌走向下坡路。使用**PHP**，一個月的時間就可以完成，並能交付大部分的功能。像**PHP**這樣的語言和**Ruby on Rails**這樣的框架越來越受到關注（參見[TH05]），這表明了開發者已經意識到舊的技術再也行不通了。

但丟棄已經會的東西並不容易。很多團隊在猶豫，是因為管理者拒絕用**500**美元購買一臺構建機器（**build machine**），卻寧願花費好幾萬

美元的人工費，讓程序員花時間找出問題。而實際上，買一臺構建機器就可以解決這些問題。如果購買機器需要花費500 000美元，那樣做還情有可原，但現在早已時過境遷了。

**根深蒂固的習慣不可能輕易地就丟棄掉**

## **Expensive mental models aren't discarded lightly**

在學習一門新技術的時候，多問問自己，是否把太多舊的態度和方法用在了新技術上。學習面向對象編程和學習面向過程編程是截然不同的。很容易會發現有人用C語言的方式編寫Java代碼，用VB的方式編寫C#的代碼（或者用Fortran的方式做任何事情）。這樣，你辛苦地轉向一門新的語言，卻失去了期望獲得的益處。

打破舊習慣很難，更難的是自己還沒有意識到這個問題。丟棄的第一步，就是要**意識到**你還在使用過時的方法，這也是最難的部分。另一個難點就是要做到真正地丟棄舊習慣。思維定式是經過多年摸爬滾打才構建成型的，已經根深蒂固，沒有人可以很容易就丟棄它們。

這也不是說你真地要完全丟棄它們。前面那個內存重複佔位的例子，只是在稍大緩存中用手工維護一組工件的特殊案例。儘管實現方式不同了，但以前的技術還在你的大腦中。你不可能撬開大腦，把這一段記憶神經剪掉。其實，根據具體情況還可以運用舊知識。如果環境合適，可以舉一反三地靈活應用，但一定要保證不是習慣性地落入舊習慣。

應該力求儘可能完全轉入新的開發環境。例如，學習一門新的編程語言時，應使用推薦的集成開發環境，而不是你過去開發時用的工具插件。用這個工具編寫一個和過去完全不同類型的項目。轉換的時候，完全不要使用過去的語言開發工具。只有更少被舊習慣牽絆，才更容易養成新習慣。





**學習新的東西，丟棄舊的東西**。在學習一門新技術的時候，要丟棄會阻止你前進的舊習慣。畢竟，汽車要比馬車車廂強得多。

## 切身感受

新技術會讓人感到有一點恐懼。你確實需要學習很多東西。已有的技能和習慣為你打下了很好的基礎，但不能依賴它們。

## 平衡的藝術

- 沉舟側畔千帆過，病樹前頭萬木春。要果斷丟棄舊習慣，一味遵循過時的舊習慣會危害你的職業生涯。
- 不是完全忘記舊的習慣，而是隻在使用適當的技術時才使用它。
- 對於所使用的語言，要總結熟悉的語言特性，並且比較這些特性在新語言或新版本中有什麼不同。

# 8 打破砂鍋問到底

“接受別人給你的解釋。別人告訴你問題出在了什麼地方，你就去看什麼地方。不需要再浪費時間去追根究底。”



前面談到的一些習慣是關於如何提高你和團隊的技術的。下面有一個習慣幾乎總是**有用**，可以用於設計、調試以及理解需求。

假設，應用系統出了大問題，他們找你來修復它。但你不熟悉這個應用系統，所以他們會幫助你，告訴你問題一定是出在哪個特殊的模塊中——你可以放心地忽略應用系統的其他地方。你必須很快地解決這個問題，因為跟你合作的這些人耐心也很有限。

當你受到那些壓力的時候，也許會覺得受到了脅迫，不想去深入瞭解問題，而且別人告訴你的已經夠深入了。然而，為了解決問題，你需要很好地瞭解系統的全局。你需要查看所有你認為和問題相關的部分——即便其他人覺得這並不相干。

觀察一下醫生是如何工作的。當你不舒服的時候，醫生會問你各種各樣的問題——你有什麼習慣，你吃了什麼東西，什麼地方疼痛，你已經服過什麼樣的藥等。人的身體非常複雜，會受到很多因素的影響。如果醫生沒有全面地瞭解狀況，就很可能出現誤診。

例如，住在紐約市的一個病人患有高燒、皮疹、嚴重的頭痛、眼睛後面疼痛，以及肌肉和關節疼痛，他也許是染上了流感或者麻疹。但是，通過全面的檢查，醫生髮現這個倒黴的病人剛去南美洲度假回來。所以，這病也許並不是簡單的流感，還有可能是在新大陸染上的熱帶傳染病登革熱。

在計算機世界中也很相似，很多問題都會影響你的應用系統。為了解決問題，你需要知道許多可能的影響因素。當找人詢問任何相關的問題時，讓他們耐心地回答你的問題，這是你的職責。

或者，假設你和資深的開發者一起工作。他們可能比你更瞭解這個系統。但他們也是人，有時他們也會忘記一些東西。你的問題甚至會幫助他們理清思路。你從一個新人角度提出的問題，給他們提供了一個新的視角，也許就幫助他們解決了一直令人困擾的問題。

“為什麼”是一個非常好的問題。事實上，在一本流行的管理圖書《第五項修煉》中，作者建議，在理解一個問題的時候，需要漸次地問5個以上的“為什麼”。這聽起來就像退回到了4歲，那時對一切都充滿著好奇。它是很好的方式，進一步挖掘簡單直白的答案，通過這個路線，設想就會更加接近事實真相。

在《第五項修煉》一書中就有這樣的例子。諮詢師訪問一個製造設備工廠的經理，就用到了這樣一些追根究底的分析。看到地板上有油漬

的時候，經理的第一反應是命令工人把它打掃乾淨。但是，諮詢師問：“為什麼地板上會有油漬？”經理不熟悉整個流程，就會責備這是清潔隊的疏忽。諮詢師再次問道：“為什麼地板上有油漬？”通過一系列漸次提出的“為什麼”和許多不同部門員工的幫助，諮詢師最後找到了真正的問題所在：採購政策表述不明確，導致大量採購了一批有缺陷的墊圈。

答案出來之後，經理和其他員工都十分震驚，他們對這事一無所知。由此發現了一個重大的隱患，避免了其他方面更大的損失。而諮詢師所做的不過就是問了“為什麼”。

“哎呀，只要每週重啟一次系統，就沒有問題了。”真的嗎？為什麼呀？“你必須依次執行3次構建才能完成構建。”真的嗎？為什麼呀？“我們的用戶根本不想要那個功能。”真的嗎？為什麼呀？

為什麼呀？



**不停地問為什麼**。不能只滿足於別人告訴你的表面現象。要不停地提問直到你明白問題的根源。

## 切身感受

這就好比是從礦石中採掘貴重的珠寶。你不停地篩選掉無關的物質，一次比一次深入，直到找到發光的寶石。你要能感覺到真正地理解了問題，而不是隻知道表面的症狀。

## 平衡的藝術

- 你可能會跑題，問了一些與主題無關的問題。就好比是，如果汽車啟動不了，你問是不是輪胎出了問題，這是沒有任何幫助的。問“為什麼”，但是要問到點子上。

- 當你問“為什麼”的時候，也許你會被反問：“為什麼你問這個問題？”在提問之前，想好你提問的理由，這會有助於你問出恰當的問題。
- “這個，我不知道”是一個好的起點，應該由此進行更進一步的調查，而不應在此戛然結束。

## 9 把握開發節奏

“我們很長時間沒有進行代碼複審，所以這週會複審所有的代碼。此外，我們也要做一個發佈計劃了，那就從星期二開始，用3周時間，做下一個發佈計劃。”



在許多不成功的項目中，基本上都是隨意安排工作計劃，沒有任何的規律。那樣的隨機安排很難處理。你根本不知道明天將會發生什麼，也不知道什麼時候開始下一輪的全體“消防演習”。

但是，敏捷項目會有一個節奏和循環，讓開發更加輕鬆。例如，Scrum約定了30天之內不應發生需求變化，這樣確保團隊有一個良性的開發節奏。這有助於防止一次計劃太多的工作和一些過大的需求變更。

相反，很多敏捷實踐必須一直進行，也就是說，它貫穿於項目的整個生命週期。有人說，上帝發明了時間，就是為了防止所有事情同時發生。因此我們需要更具遠見，保持不同的開發節奏，這樣敏捷項目的所有事情就不會突然同時發生，也不會隨機發生，時間也不會不可預知。

我們先來看某個工作日的情況。你希望每天工作結束的時候，都能完成自己的工作，你手上沒有遺留下任何重要的任務。當然，每天都能這樣是不現實的。但是，你可以做到在每天下班離開公司前運行測試，並提交一天完成的代碼。如果已經很晚了，並且你只是嘗試性地

編寫了一些代碼，那麼也許最好應該刪掉這些代碼，第二天從頭開始。

這個建議聽起來十分極端，也許確實有一點。<sup>①</sup>但是如果你正在開發小塊的任務，這種方式非常有助於你管理自己的時間：如果你工作的時候沒有一個固定的最終期限（例如一天的結束），就應該好好想了想。它會讓你的工作有一個節奏，在每天下班的時候，提交所有的工作，開心地收工。這樣，明天就能開始新的內容，解決下一系列難題。

① Ron Jeffrey告訴我們：“我希望人們敢於經常這麼做。”

---

## 時 間 盒

敏捷開發者可以從多方面得到反饋：用戶、團隊成員和測試代碼。這些反饋會幫助你駕馭項目。但是時間本身就是一個非常重要的反饋。

許多的敏捷技巧來源於時間盒——設定一個短時的期限，為任務設定不能延長的最終期限。你可以選擇放棄其他方面的任務，但是最終期限是不變的。你可能不知道完成所有的任務需要多少個時間盒，但每個時間盒必須是短期的、有限的，並且要完成具體的目標。

例如，迭代一般是兩週的時間。當時間到的時候，迭代就完成了。那部分是固定不變的，但是在一個具體的迭代中完成哪些功能是靈活的。換句話說，你不會改變時間，但是你可以改變功能。相似地，你會為設計討論會設定一個時間盒，即到了指定的時間點，會議就結束，同時必須要做出最終的設計決策。

當你遇到艱難抉擇的時候，固定的時間期限會促使你做決定。你不能在討論或功能上浪費很多時間，這些時間可以用於具體的工作。時間盒會幫助你一直前進。

鯊魚必須不停地向前遊，否則就會死亡。在這方面，軟件項目就像是鯊魚，你需要不停地前進，同時要清楚自己的真實進度。

站立會議（習慣38，第148頁）最好每天在固定的時間和地點舉行，比如說上午10點左右。要養成這樣的習慣，在那時就準備好一切參加站立會議。

最大的節拍就是迭代時間（習慣17，第69頁），一般是1~4周的時間。不管你的一個迭代是多長，都應該堅持——確保每個迭代週期的時間相同很重要。運用有規律的開發節奏，會更容易達到目標，並確保項目不停地前進。



**解決任務，在事情變得一團糟之前**。保持事件之間穩定重複的間隔，更容易解決常見的重複任務。

## 切身感受

項目開發需要有一致和穩定的節奏。編輯，運行測試，代碼複審，一致的迭代，然後發佈。如果知道什麼時候開始下一個節拍，跳舞就會更加容易。

## 平衡的藝術

- 在每天結束的時候，測試代碼，提交代碼，沒有殘留的代碼。
- 不要搞得經常加班。
- 以固定、有規律的長度運行迭代（第69頁，習慣17）。也許剛開始你要調整迭代的長度，找到團隊最舒服可行的時間值，但之後就必須要堅持。
- 如果開發節奏過於密集，你會精疲力竭的。一般來說，當與其他團隊（或組織）合作時，你需要減慢開發節奏。因此人們常說，

互聯網時代發展太快，有害健康。

- 有規律的開發節奏會暴露很多問題，讓你有更多鼓起勇氣的藉口（第23頁，習慣4）。
- 就像是減肥一樣，一點點的成功也是一個很大的激勵。小而可達到的目標會讓每個人全速前進。慶祝每一次難忘的成功：共享美食和啤酒或者團隊聚餐。

## 第4章 交付用戶想要的軟件

沒有任何計劃在遇敵後還能繼續執行。

——**Helmuth von Moltke**（德國陸軍元帥，1848—1916）

客戶把需求交給你了，要你幾年後交付這個系統。然後，你就基於這些需求構建客戶需要的系統，最後按時交付。客戶看到了軟件，連聲稱讚做得好。從此你又多了一個忠實客戶，接著你很開心地進入了下一個項目。你的項目通常都是這樣運作的，是這樣的嗎？

其實，大部分人並不會遇到這樣的項目。通常情況是：客戶最後看到了軟件，要麼震驚要麼不高興。他們不喜歡所看到的軟件，他們認為很多地方需要修改。他們要的功能不在他們給你的原始需求文檔中。這聽起來是不是更具代表性？

Helmuth von Moltke曾說過：“沒有任何計劃在遇敵後還能繼續執行。”我們的敵人不是客戶，不是用戶，不是隊友，也不是管理者。真正的敵人是變化。軟件開發如戰爭，形勢的變化快速而又劇烈。固守昨天的計劃而無視環境的變化會帶來災難。你不可能“戰勝”變化——無論它是設計、架構還是你對需求的理解。敏捷——成功的軟件開發

方法——取決於你識別和適應變化的能力。只有這樣才有可能在預算之內及時完成開發，創建真正符合用戶需求的系統。

在本章中，我們會介紹如何達到敏捷的目標。首先，要介紹為什麼用戶和客戶參與開發如此重要，以及為什麼**讓客戶做決定**（從第45頁開始）。設計是軟件開發的基礎，沒有它很難做好開發，但你也不能被它牽制。從第48頁開始，將介紹如何**讓設計指導而不是操縱開發**。說到牽制，你應確保在項目中引入合適的技術。你需要**合理地使用技術**（第52頁介紹）。

為了讓軟件符合用戶的需求，要一直做下面的準備工作。為了降低集成新代碼帶來的破壞性變化，你要**提早集成，頻繁集成**（第58頁）。當然，你不想破壞已有的代碼，想讓代碼一直保持**可以發佈**（從第55頁開始）。

你不能一次又一次為用戶演示新功能，而浪費寶貴的開發時間，因此你需要**提早實現自動化部署**（第61頁）。只要你的代碼一直可用，並且易於向用戶部署，你就能**使用演示獲得頻繁反饋**（第64頁）。這樣你就能經常向全世界發佈新版本。你想通過**使用短迭代，增量發佈**來幫助經常發佈新功能，與用戶的需求變化聯繫更緊密（從第69頁開始介紹它）。

最後，特別是客戶要求預先簽訂固定價格合約時，很難通過敏捷的方法讓客戶與我們同坐一條船上。而且，事實上是**固定的價格就意味著背叛承諾**，我們會在第73頁瞭解如何處理這種情況。

## 10 讓客戶做決定

“開發者兼具創新和智慧，最瞭解應用程序。因此，所有關鍵決定都應該由開發者定奪。每次業務人員介入的時候，都會弄得一團糟，他們無法理解我們做事的邏輯。”





在設計方面，做決定的時候必須有開發者參與。可是，在一個項目中，他們不應該做所有的決定，特別是業務方面的決定。

就拿項目經理**Pat**的例子來說吧。**Pat**的項目是遠程開發，一切按計劃且在預算內進行著——就像是個可以寫入教科書的明星項目。**Pat**高高興興地把代碼帶到客戶那裡，給客戶演示，卻敗興而歸。

原來，**Pat**的業務分析師沒有和用戶討論，而是自作主張，決定了所有的問題。在整個開發過程中，企業主根本沒有參與低級別的決策。項目離完成還早著呢，就已經不能滿足用戶的需要了。這個項目一定會延期，又成為一個經典的失敗案例。

因而，你只有一個選擇：要麼現在就讓用戶做決定，要麼現在就開始開發，遲些讓用戶決定，不過要付出較高的成本。如果你在開發階段迴避這些問題，就增加了風險，但是你要能越早解決這些問題，就越有可能避免繁重的重新設計和編碼。甚至在接近項目最終期限的時候，也能避免與日俱增的時間壓力。

例如，假設你要完成一個任務，有兩種實現方式。第一種方式的實現比較快，但是對用戶有一點限制。第二種方式實現起來需要更多的時間，但是可以提供更大的靈活性。很顯然，你有時間的壓力（什麼項目沒有時間壓力呢），那麼你就用第一種很快的方式嗎？你憑什麼做出這樣的決定呢？是投硬幣嗎？你詢問了同事或者你的項目經理嗎？

作者之一**Venkat**最近的一個項目就遇到了類似的問題。項目經理為了節約時間，採取了第一種方式。也許你會猜到，在**Beta**版測試的時候，軟件暴露出的侷限讓用戶震驚，甚至憤怒。結果還得重做，花費了團隊更多的金錢、時間和精力。

決定什麼不該決定

**Decide what you shouldn't decide**

開發者（及項目經理）能做的一個最重要的決定就是：判斷哪些是自己決定不了的，應該讓企業主做決定。你不需要自己給業務上的關鍵問題做決定。畢竟，那不是你的事情。如果遇到了一個問題，會影響到系統的行為或者如何使用系統，把這個問題告訴業務負責人。如果項目領導或經理試圖全權負責這些問題，要委婉地勸說他們，這些問題最好還是和真正的業務負責人或者客戶商議（見習慣4，第23頁）。

當你和客戶討論問題的時候，準備好幾種可選擇的方案。不是從技術的角度，而是從業務的角度，介紹每種方案的優缺點，以及潛在的成本和利益。和他們討論每個選擇對時間和預算的影響，以及如何權衡。無論他們做出了什麼決定，他們必須接受它，所以最好讓他們瞭解一切之後再做這些決定。如果事後他們又想要其他的東西，可以公正地就成本和時間重新談判。

畢竟，這是他們的決定。



**讓你的客戶做決定**。開發者、經理或者業務分析師不應該做業務方面的決定。用業務負責人能夠理解的語言，向他們詳細解釋遇到的問題，並讓他們做決定。

## 切身感受

業務應用需要開發者和業務負責人互相配合來開發。這種配合的感覺就應該像一種良好的、誠實的工作關係。

## 平衡的藝術

- 記錄客戶做出的決定，並註明原因。好記性不如爛筆頭。可以使用工程師的工作日記或日誌、Wiki、郵件記錄或者問題跟蹤數據庫。但是也要注意，你選擇的記錄方法不能太笨重或者太繁瑣。

- 不要用低級別和沒有價值的問題打擾繁忙的業務人員。如果問題對他們的業務沒有影響，就應該是沒有價值的。
- 不要隨意假設低級別的問題不會影響他們的業務。如果能影響他們的業務，就是有價值的問題。
- 如果業務負責人回答“我不知道”，這也是一個稱心如意的答案。也許是他們還沒有想到那麼遠，也許是他們只有看到運行的實物才能評估出結果。盡你所能為他們提供建議，實現代碼的時候也要考慮可能出現的變化。

## 11 讓設計指導而不是操縱開發

“設計文檔應該儘可能詳細，這樣，低級的代碼工人只要敲入代碼就可以了。在高層方面，詳細描述對象的關聯關係；在低層方面，詳細描述對象之間的交互。其中一定要包括方法的實現信息和參數的註釋。也不要忘記給出類裡面的所有字段。編寫代碼的時候，無論你發現了什麼，絕不能偏離了設計文檔。”



“設計”是軟件開發過程不可缺少的步驟。它幫助你理解系統的細節，理解部件和子系統之間的關係，並且指導你的實現。一些成熟的方法論很強調設計，例如，統一過程（**Unified Process**，**UP**）十分重視和產品相關的文檔。項目管理者和企業主常常為開發細節困擾，他們希望在開始編碼之前，先有完整的設計和文檔。畢竟，那也是你如何管理橋樑或建築項目的，難道不是嗎？

另一方面，敏捷方法建議你早在開發初期就開始編碼。是否那就意味著沒有設計呢？① 不，絕對不是，好的設計仍然十分重要。畫關鍵工作圖（例如，用**UML**）是必不可少的，因為要使用類及其交互關係來描繪系統是如何組織的。在做設計的時候，你需要花時間去思考（討論）各種不同選擇的缺陷和益處，以及如何做權衡。

① 查閱Martin Fowler的文章*Is Design Dead?*

( <http://www.martinfowler.com/articles/designDead.html> )，它是對本主題深入討論的一篇好文章。

然後，下一步才考慮是否需要開始編碼。如果你在前期沒有考慮清楚這些問題，就草草地開始編碼，很可能會被很多意料之外的問題搞暈。甚至在建築工程方面也有類似的情況。在鋸一根木頭的時候，通常的做法就是先鋸一塊比需要稍微長一點的木塊，最後細緻地修整，直到它正好符合需求。

但是，即使之前已經提交了設計文檔，也還會有一些意料之外的情況出現。時刻謹記，此階段提出的設計只是基於你**目前**對需求的理解而已。一旦開始了編碼，一切都會改變。設計及其代碼實現會不停地發展和變化。

一些項目領導和經理認為設計應該儘可能地詳細，這樣就可以簡單地交付給“代碼工人們”。他們認為代碼工人不需要做任何決定，只要簡單地把設計轉化成代碼就可以了。就作者本人而言，沒有一個願意在這樣的團隊中做純粹的打字員。我們猜想你也不願意。

**設計滿足實現即可，不必過於詳細**

**Design should be only as detailed as needed to implement**

如果設計師們把自己的想法繪製成精美的文檔，然後把它們扔給程序員去編碼，那會發生什麼（查閱習慣39，在第152頁）？程序員會在壓力下，完全按照設計或者圖畫的樣子編碼。如果系統和已有代碼的現狀表明接收到的設計不夠理想，那該怎麼辦？太糟糕了！時間已經花費在設計上，沒有工夫回頭重新設計了。團隊會死撐下去，用代碼實現了明明知道是錯誤的設計。這聽起來是不是很愚蠢？是夠愚蠢的，但是有一些公司真的就是這樣做的。

嚴格的需求—設計—代碼—測試開發流程源於理想化的**瀑布式** ②開發方法，它導致在前面進行了過度的設計。這樣在項目的生命週期中，更新和維護這些詳細的設計文檔變成了主要工作，需要時間和資源方面的巨大投資，卻只有很少的回報。我們本可以做得更好。

② 瀑布式開發方法意味著要遵循一系列有序的開發步驟，前面是定義詳細的需求，然後是詳細的設計，接著是實現，再接著是集成，最後是測試（此時你需要向天祈禱）。那不是作者首先推薦的做法。更多詳情可以查閱[Roy70]。

設計可以分為兩層：**戰略** 和**戰術**。前期的設計屬於戰略，通常只有在沒有深入理解需求的時候需要這樣的設計。更確切地說，它應該只描述總體戰略，不應深入到具體的細節。

### 做到精確

如果你自己都不清楚所談論的東西，就根本不可能精確地描述它。

——約翰·馮·諾依曼

前面剛說過，戰略級別的設計不應該具體說明程序方法、參數、字段和對象交互精確順序的細節。那應該留到戰術設計階段，它應該在項目開發的時候再具體展開。

### 戰略設計與戰術設計

#### Strategic versus tactical design

良好的戰略設計應該扮演地圖的角色，指引你向正確的方向前進。任何設計僅是一個起跑點：它就像你的代碼一樣，在項目的生命週期中，會不停地進一步發展和提煉。

下面的故事會給我們一些啟發。在1804年，Lewis與Clark③進行了橫穿美國的壯舉，他們的“設計”就是穿越蠻荒。但是，他們不知道在穿

越殖民地時會遇到什麼樣的問題。他們只知道自己的目標和制約條件，但是不知道旅途的細節。

③ 這世界真小，Andy還是William Clark的遠親呢。

軟件項目中的設計也與此類似。在沒有穿越殖民地的時候，你不可能知道會出現什麼情況。所以，不要事先浪費時間規劃如何徒步穿越河流，只有當你走到河岸邊的時候，才能真正評估和規劃如何穿越。只有到那時，你才開始真正的戰術設計。

不要一開始就進行戰術設計，它的重點是集中在單個的方法或數據類型上。這時，更適合討論如何設計類的職責。因為這仍然是一個高層次、面向目標的設計。事實上，CRC（類—職責—協作）卡片的設計方法就是用來做這個事情的。每個類按照下面的術語描述。

- 類名。
- 職責：它應該做什麼？
- 協作者：要完成工作它要與其他什麼對象一起工作？

如何知道一個設計是好的設計，或者正合適？代碼很自然地為設計的好壞提供了最好的反饋。如果需求有了小的變化，它仍然容易去實現，那麼它就是好的設計。而如果小的需求變化就帶來一大批基礎代碼的破壞，那麼設計就需要改進。



**好設計是一張地圖，它也會進化**。設計指引你向正確的方向前進，它不是殖民地，它不應該標識具體的路線。你不要被設計（或者設計師）操縱。

切身感受

好的設計應該是正確的，而不是精確的。也就是說，它描述的一切必須是正確的，不應該涉及不確定或者可能會發生變化的細節。它是目標，不是具體的處方。

## 平衡的藝術

- “不要在前期做大量的設計”並不是說不要設計。只是說在沒有經過真正的代碼驗證之前，不要陷入太多的設計任務。當對設計一無所知的時候，投入編碼也是一件危險的事。如果深入編碼只是為了學習或創造原型，只要你隨後能把這些代碼扔掉，那也是一個不錯的辦法。
- 即使初始的設計到後面不再管用，你仍需設計：設計行為是無價的。正如美國總統艾森豪威爾所說：“計劃是沒有價值的，但**計劃的過程**是必不可少的④。”在設計過程中學習是有價值的，但設計本身也許沒有太大的用處。

④ 1957年的演講稿。

- 白板、草圖、便利貼都是非常好的設計工具。複雜的建模工具只會讓你分散精力，而不是啟發你的工作。

## 12 合理地使用技術

“你開始了一個新的項目，在你面前有一長串關於新技術和應用框架的列表。這些都是好東西，你真的需要使用列表中所有的技術。想一想，你的簡歷上將留下漂亮的一筆，用那些偉大的框架，你的新應用將具有極高技術含量。”



---

盲目地為項目選擇技術框架，就好比是為了少交稅而生孩子

## Blindly picking a framework is like having kids to save taxes

從前，作者之一Venkat的同事Lisa向他解釋自己的提議：她打算使用EJB。Venkat表示對EJB有些顧慮，覺得它不適合那個特殊的項目。然後Lisa回答道：“我已經說服了我們經理，這是正確的技術路線，所以現在不要再扔‘炸彈’了。”這是一個典型的“簡歷驅動設計”的例子，之所以選擇這個技術，是因為它很美，也許還能提高程序員的技能。但是，盲目地為項目選擇技術框架，就好比是為了節省稅款而生孩子，這是沒有道理的。

在考慮引入新技術或框架之前，先要把你需要解決的問題找出來。你的表述方式不同，會讓結果有很大差異。如果你說“我們需要xyzzzy技術，是因為.....”，那麼就不太靠譜。你應該這樣說：“.....太難了”或者是“.....花的時間太長了”，或者類似的句子。找到了需要解決的問題，接下來就要考慮如下方面。

- **這個技術框架真能解決這個問題嗎？** 是的，也許這是顯而易見的。但是，這個技術真能解決你面臨的那個問題嗎？或者，更尖銳一點說，你是如何評估這個技術的？是通過市場宣傳還是道聽途說？要確保它能解決你的問題，並沒有任何的毒副作用。如果需要，先做一個小的原型。
- **你將會被它拴住嗎？** 一些技術是賊船，一旦你使用了它，就會被它套牢，再也不可能回頭了。它缺乏**可取消性**（查閱[HT00]），當條件發生變化時，這可能對項目有致命打擊。我們要考慮它是開放技術還是專利技術，如果是開放的技術，那又開放到什麼程度？
- **維護成本是多少？** 會不會隨著時間的推移，它的維護成本會非常昂貴？畢竟，方案的花費不應該高於要解決的問題，否則就是一次失敗的投資。我們聽說，有個項目的合同是支持一個規則引



擎，引擎一年的維護費用是5萬美元，但是這個數據庫只有30條規則。這也太貴了。

當你在考察一個框架（或者任何技術）的時候，也許會被它提供的各種功能吸引。接著，在驗證是否使用這個框架的時候，你可能只會考慮已經發現的另外一些功能。但是，你真的需要這些功能嗎？也許為了迎合你發現的功能，你正在為它們找問題。這很像站在結賬處一時衝動而買些無用的小零碎（那也正是商場把那些小玩意兒放到那裡的原因）。

不久前，Venkat遇到了一個項目。諮詢師Brad把一個專有框架賣給了這個項目的管理者。在Venkat看來，這個框架本身也許還有點兒意思，但是它根本不適合這個項目。

儘管如此，管理者卻堅決認為他們要使用它。Venkat非常禮貌地停手不幹了。他不想成為絆腳石，阻礙他們的工作進度。一年之後項目還沒有完成——他們花了好幾個月的時間編寫代碼來維護這個框架，為了適應這個框架，他們還修改了自己的代碼。

Andy有過相似的經歷：他的客戶想完全透明地利用開源，他們擁有“新技術大雜燴”，其中的東西太多，以至於無法讓所有的部分協同工作。

**不要開發你能下載到的東西**

**Don't build what you can download**

如果你發現自己在做一些花哨的東西（比如從頭創建自己的框架），那就醒醒吧，聞聞煙味有多大，馬上該起火了。你的代碼寫得越少，需要維護的東西就越少。

例如，如果你想開發自己的持久層框架，記住Ted Neward的評論：對象—關係的映射就是計算機科學的越南戰場<sup>①</sup>。你可以把更多的時間和精力投入到應用的開發——領域或具體應用中。

① Ted Neward曾寫過*The Vietnam of Computer Science* 著名文章，逐一探討了對象—關係映射的缺點。——編者注

---



**根據需要選擇技術**。首先決定什麼是你需要的，接著為這些具體的問題評估使用技術。對任何要使用的技術，多問一些挑剔的問題，並真實地作出回答。

## 切身感受

新技術就應該像是新的工具，可以幫助你更好地工作，它自己不應該成為你的工作。

## 平衡的藝術

- 也許在項目中真正評估技術方案還為時太早。那就好。如果你在做系統原型並要演示給客戶看，也許一個簡單的散列表就可以代替數據庫了。如果你還沒有足夠的經驗，不要急於決定用什麼技術。
- 每一門技術都會有優點和缺點，無論它是開源的還是商業產品、框架、工具或者語言，一定要清楚它的利弊。
- 不要開發那些你容易下載到的東西。雖然有時需要從最基礎開發所有你需要的東西，但那是相當危險和昂貴的。

# 13 保持可以發佈

“我們剛試用的時候發現了一個問題，你需要立即修復它。放下你手頭的工作，去修復那個剛發現的問題，不需要經過正規的程序。不用告訴其他任何人——趕快讓它工作就行了。”



這聽起來似乎沒什麼問題。有一個關鍵修復的代碼必須要提交到代碼庫。這只是一件小事，而且又很緊急，所以你就答應了。

修復工作成功地完成了。你提交了代碼，繼續回到以前那個高優先級的任務中。忽然一聲尖叫。太晚了，你發現同事提交的代碼和你的代碼發生了衝突，現在你使得每個人都無法使用系統了。這將會花費很多精力（和時間）才能讓系統重新回到可發佈的狀態。現在你有麻煩了。你必須告訴大家，你不能交付你承諾的修復代碼了。而魔鬼在嘲笑：“哈哈！”

這時候，你的處境會很糟糕：系統無法發佈了。你弄壞了系統，也許會帶來更糟糕的後果。

1836年，當時的墨西哥總統安東尼奧·洛佩斯·德·聖安那將軍，率領部隊穿越得克薩斯州西部，追趕敗退的薩姆·休斯頓將軍。當聖安那的部隊到達得克薩斯州東南方向的布法羅河岸的沼澤地帶的時候，他命令自己的部隊就地休息。傳說中認為他是太過自信，甚至沒有安排哨兵。就在那個傍晚，休斯頓發動了突然襲擊，這時聖安那的部隊已經來不及編隊了。他們潰不成軍，輸掉了這場決定性的戰爭，從此永遠改變了得克薩斯州的歷史<sup>①</sup>。

---

① [http://www.sanjacinto-museum.org/The\\_Battle/April\\_21st\\_1836](http://www.sanjacinto-museum.org/The_Battle/April_21st_1836)。

---

已提交的代碼應該隨時可以行動

### **Checked-in code is always ready for action**

任何時候只要你沒有準備好，那就是敵人進攻你的最佳時機。好好想一想，你的項目進入不可發佈狀態的頻率是多少？你的源代碼服務器中的代碼，是不是像聖安那在那個決定性的黃昏一樣——沒有進行編隊，遇到緊急情況無法立即啟動。

在團隊裡工作，修改一些東西的時候必須很謹慎。你要時刻警惕，每次改動都會影響系統的狀態和整個團隊的工作效率。在辦公室的廚房裡，你不能容忍任何人亂丟垃圾，為什麼就可以容忍一些人給項目帶來垃圾代碼呢？

下面是一個簡單的工作流程，可以防止你提交破壞系統的代碼。

- **在本地運行測試**。先保證你完成的代碼可以編譯，並且能通過所有的單元測試。接著確保系統中的其他測試都可以通過。
- **檢出最新的代碼**。從版本控制系統中更新代碼到最新的版本，再編譯和運行測試。這樣往往會發現讓你吃驚的事情：其他人提交的新代碼和你的代碼發生了衝突。
- **提交代碼**。現在是最新的代碼了，並且通過了編譯和測試，你可以提交它們了。

在做上面事情的時候，也許你會遇到這樣一個問題——其他人提交了一些代碼，但是沒有通過編譯或者測試。如果發生了這樣的事情，要立即讓他們知道，如果有需要，可以同時警告其他的同事。當然，最好的辦法是，你有一個**持續集成**系統，可以自動集成並報告集成結果。

這聽起來似乎有點恐怖，其實很簡單。持續集成系統就是在後臺不停地檢出、構建和測試代碼的應用。你可以自己使用腳本快速實現這樣的方式，但如果你選擇已有的免費、開源的解決方案，它們會提供更多的功能且更加穩定。有興趣的話，可以看一看Martin Fowler的文章②，或者是Mike Clark編著的圖書《項目自動化之道》[Cla04]。

②

<http://www.martinfowler.com/articles/continuousIntegration.html>

。

再深入一點，假設你得知即將進行的一次重大修改很可能會破壞系統，不要任其發生，應該認真地警告大家，在代碼提交之前，找出可以避免破壞系統的方法。選擇可以幫助你平滑地引入和轉換這些修改的方法，從而在開發過程中，系統可以得到持續的測試和反饋。

雖然保持系統可發佈非常重要，但不會總是那麼容易，例如，修改了數據庫的表結構、外部文件的格式，或者消息的格式。這些修改，通常會影響應用的大部分代碼，甚至導致應用暫時不可用，直到大量的代碼修改完成。儘管如此，你還是有辦法減輕這樣的痛苦。

為數據庫的表結構、外部文件，甚至引用它的API提供版本支持，這樣所有相關變化都可以進行測試。有了版本功能，所做的變化可以與其他代碼基相隔離，所以應用的其他方面仍然可以繼續開發和測試。

你也可以在版本控制系統中添加一個分支，專門處理這個問題（使用分支需要十分小心，不好的分支也許會給你帶來更多的麻煩。詳情可以查閱《版本控制之道——CVS》或《版本控制之道——Subversion》）。



**保持你的項目時刻可以發佈**。保證你的系統隨時可以編譯、運行、測試並立即部署。

## 切身感受


你會覺得，不管什麼時候，你的老闆、董事長、質量保障人員、客戶或者你的配偶來公司參觀項目的時候，你都能很自信並毫不猶豫地給他們演示最新構建的軟件。你的項目一直處於可以運行的穩定狀態。

## 平衡的藝術

- 有時候，做一些大的改動後，你無法花費太多的時間和精力去保證系統一直可以發佈。如果總共需要一個月的時間才能保證它一週內可以發佈，那就算了。但這隻應該是例外，不能養成習慣。

- 如果你不得不讓系統長期不可以發佈，那就做一個（代碼和架構的）分支版本，你可以繼續進行自己的實驗，如果不行，還可以撤銷，從頭再來。千萬不能讓系統既不可以發佈，又不可以撤銷。

## 14 提早集成，頻繁集成

“只要沒有到開發的末尾階段，就不要過早地浪費時間去想如何集成你的代碼，至少也要等開發差不多的時候，才開始考慮它。畢竟，還沒有完成開發，為什麼要操心集成的事情呢！在項目的末尾，你有充裕的時間來集成代碼。” 

我們說過，敏捷的一個主要特點就是持續開發，而不是三天打魚兩天曬網似地工作。特別是在幾個人一起開發同一個功能的時候，更應該頻繁地集成代碼。

很多開發者用一些美麗的藉口，推遲集成的時間。有時，不過是為了多寫一些代碼，或者是另一個子系統還有很多的工作要完成。他們很容易就會這樣想：“現在手頭上的工作壓力夠大了，到最後我才能做更多的工作，才能考慮其他人代碼。”經常會聽到這樣的藉口：“我沒有時間進行集成”或者“在我機器上設置集成環境太費事了，我現在不想做它”。

但是，在產品的開發過程中，集成是一個主要的風險區域。讓你的子系統不停地增長，不去做系統集成，就等於一步一步把自己置於越來越大的風險中，世界沒有了你仍然會轉動，潛在的分歧會繼續增加。相反，儘可能早地集成也更容易發現風險，這樣風險及相關的代價就會相當低。而等的時間越長，你也就會越痛苦。

作者之一Venkat小時候生活在印度欽奈市（Chennai），經常趕火車去學校。像其他的大城市一樣，印度的交通非常擁擠。他每次必須在車還沒有停穩的時候，就跳上去或者跳下來。但，你不能從站的地方

一下子跳上運行的火車，我們在物理課上學習過這種運動定律。而應該是，首先你要沿著火車行駛的方向跑，邊跑邊抓住火車上的扶手，然後跳入火車中。

軟件集成就像這一樣。如果你不斷地獨立開發，忽然有一天跳到集成這一步，千萬不要為受到打擊而吃驚。也許你自己在項目中就有這樣的體會：每次到項目結束的時候都覺得非常不爽，大家需要日日夜夜地進行集成。

### 你能集成並且獨立

集成和獨立不是互相矛盾的，你可以一邊進行集成，一邊進行獨立開發。

使用mock對象來隔離對象之間的依賴關係，這樣在集成之前就可以先做測試，用一個mock對象模擬真實的對象（或者子系統）。就像是拍電影時在光線的掩飾下使用替身一樣，mock對象就是真實對象的替身，它並不提供真實對象的功能，但是它更容易控制，能夠模仿需要的行為，使測試更加簡單。

你可以使用mock對象，編寫獨立的單元測試，而不需要立刻就集成和測試其他系統，只有當你自信它能工作的時候，才可以開始集成。

當你在公司昏天黑地地加班時，唯一的好處就是可以享受到免費的披薩。

獨立開發和早期集成之間是具有張力的。當你獨立開發時，會發現開發速度更快，生產率更高，你可以更有效地解決出現的問題（見第136頁，習慣35）。但那並不意味著要你避免或延遲集成（見本頁側邊欄）。你一般需要每天集成幾次，最好不要2~3天才集成一次。

### 決不要做大爆炸式的集成

## Never accept big-bang integration

當早期就進行集成的時候，你會看到子系統之間的交互和影響，你就可以估算它們之間通信和共享的信息數據。你越早弄清楚這些問題，越早解決它們，工作量就越小。就好比是，剛開始有3個開發者，開發著5萬行的代碼，後來是5000個開發者進行3000萬行代碼的開發。相反，如果你推遲集成的時間，解決這些問題就會變得很難，需要大量和大範圍地修改代碼，會造成項目延期和一片混亂。



**提早集成，頻繁集成**。代碼集成是主要的風險來源。要想規避這個風險，只有提早集成，持續而有規律地進行集成。

### 切身感受

如果你真正做對了，集成就不再會是一個繁重的任務。它只是編寫代碼週期中的一部分。集成時產生的問題，都會是小問題並且容易解決。

### 平衡的藝術

- 成功的集成就意味著所有的單元測試不停地通過。正如醫學界希波克拉底的誓言：首先，不要造成傷害。
- 通常，每天要和團隊其他的成員一起集成代碼好幾次，比如平均每天5~10次，甚至更多。但如果你每次修改一行代碼就集成一次，那效用肯定會縮水。如果你發現自己的大部分時間都在集成，而不是寫代碼，那你一定是集成得過於頻繁了。
- 如果你集成得不夠頻繁（比如，你一天集成一次，一週一次，甚至更糟），也許就會發現整天在解決代碼集成帶來的問題，而不是在專心寫代碼。如果你集成的問題很大，那一定是做得不夠頻繁。



- 對那些原型和實驗代碼，也許你想要獨立開發，而不要想在集成上浪費時間。但是不能獨立開發太長的時間。一旦你有了經驗，就要快速地開始集成。

## 15 提早實現自動化部署

“沒問題，可以手工安裝產品，尤其是給質量保證人員安裝。而且你不需要經常自己動手，他們都很擅長複製需要的所有文件。”



系統能在你的機器上運行，或者能在開發者和測試人員的機器上運行，當然很好。但是，它同時也需要能夠部署在用戶的機器上。如果系統能運行在開發服務器上，那很好，但是它同時也要運行在生產環境中。

這就意味著，你要能用一種可重複和可靠的方式，在目標機器上部署你的應用。不幸的是，大部分開發者只會在項目的尾期才開始考慮部署問題。結果經常出現部署失敗，要麼是少了依賴的組件，要麼是少了一些圖片，要麼就是目錄結構有誤。

如果開發者改變了應用的目錄結構，或者是在不同的應用之間創建和共享圖片目錄，很可能會導致安裝過程失敗。當這些變化在人們印象中還很深的時候，你可以快速地找到各種問題。但是幾周或者幾個月之後查找它們，特別是在給客戶演示的時候，可就不是一件鬧著玩的事情了。

**質量保證人員應該測試部署過程**

### **QA should test deployment**

如果現在你還是手工幫助質量保證人員安裝應用，花一些時間，考慮如何將安裝過程自動化。這樣，只要用戶需要，你就可以隨時為他們安裝系統。要提早實現它，這樣讓質量保證團隊既可以測試應用，又

可以測試安裝過程①。如果還是手工安裝應用，那麼最後把應用部署到生產環境時會發生什麼呢？就算公司給你加班費，你也不願意為不同用戶的機器或不同地點的服務器上一遍又一遍地安裝應用。

① 確保他們能提前告訴你運行的軟件版本，避免出現混亂。

有了自動化部署系統後，在項目開發的整個過程中，會更容易適應互相依賴的變化。很可能你在安裝系統的時候，會忘記添加需要的庫或組件——在任意一臺機器上運行自動化安裝程序，你很快就會知道什麼丟失了。如果因為缺少了一些組件或者庫不兼容而導致安裝失敗，這些問題會很快浮現出來。

## Andy如是說.....

### 從第一天起就開始交付

一開始就進行全面部署，而不是等到項目的後期，這會有很多好處。事實上，有些項目在正式開發之前，就設置好了所有的安裝環境。

在我們公司，要求大家為預期客戶實現一個簡單的功能演示——驗證一個概念的可行性。即使項目還沒有正式開始，我們就有了單元測試、持續集成和基於窗口的安裝程序。這樣，我們就可以更容易更簡單地給用戶交付這個演示系統：用戶所要做的工作，就是從我們的網站上點擊一個鏈接，然後就可以自己在各種不同的機器上安裝這個演示系統了。

在簽約之前，就能提供出如此強大的演示，這無疑證明了我們非常專業，具有強大的開發能力。



**一開始就實現自動化部署應用**。使用部署系統安裝你的應用，在不同的機器上用不同的配置文件測試依賴的問題。質量保證人員

要像測試應用一樣測試部署。

## 切身感受

這些工作都應該是無形的。系統的安裝或者部署應該簡單、可靠及可重複。一切都很自然。

## 平衡的藝術

- 一般產品在安裝的時候，都需要有相應的軟、硬件環境。比如，Java或Ruby的某個版本、外部數據庫或者操作系統。這些環境的不同很可能會導致很多技術支持的電話。所以檢查這些依賴關係，也是安裝過程的一部分。
- 在沒有詢問並徵得用戶的同意之前，安裝程序絕對不能刪除用戶的數據。
- 部署一個緊急修復的bug應該很簡單，特別是在生產服務器的環境中。你知道這會發生，而且你不想在壓力之下，在凌晨3點半，你還在手工部署系統。
- 用戶應該可以安全並且完整地卸載安裝程序，特別是在質量保證人員的機器環境中。
- 如果維護安裝腳本變得很困難，那很可能是一個早期警告，預示著——很高的維護成本（或者不好的設計決策）。
- 如果你打算把持續部署系統和產品CD或者DVD刻錄機連接到一起，你就可以自動地為每個構建製作出一個完整且有標籤的光盤。任何人想要最新的構建，只要從架子上拿最上面的一張光盤安裝即可。

# 16 使用演示獲得頻繁反饋

“這不是你的過錯，問題出在我們的客戶——那些麻煩的最終客戶和用戶身上。他們不停地更改需求，導致我們嚴重地延期。他們一次就應該想清楚所有想要的東西，然後把這些需求給我們，這樣我們才能開發出令他們滿意的系統。這才是正確的工作方式。”



## 需求就像是流動著的油墨

### Requirements are as fluid as ink

你時常會聽到一些人想要凍結需求。但是，現實世界中的需求就像是流動著的油墨<sup>①</sup>。你無法凍結需求，正如你無法凍結市場、競爭、知識、進化或者成長一樣。就算你真的凍結了，也很可能是凍結了錯的東西。如果你期望用戶在項目開始之前，就能給你可靠和明確的需求，那就大錯特錯了，趕快醒醒吧！

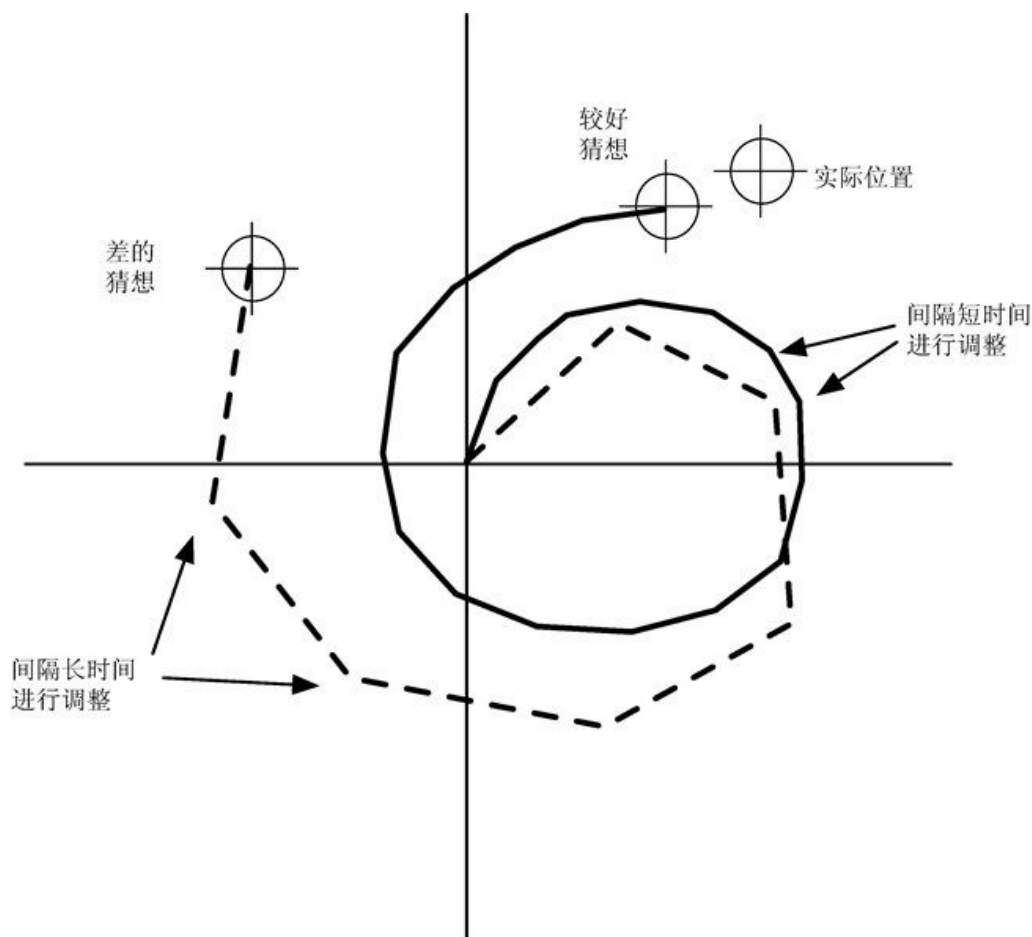
① Edward V. Berard曾經指出：“如果需求能被凍結，那麼開發軟件就如在凍冰上走路一樣簡單。”

沒有人的思想和觀點可以及時凍結，特別是項目的客戶。就算是他們已經告訴你想要的東西了，他們的期望和想法還是在不停地進化——特別是當他們在使用新系統的部分功能時，他們才開始意識到它的影響和可能發生的問題。這就是人的本性。

作為人類，不管是什麼事情，我們都能越做越好，不過是以緩慢而逐步的方式。你的客戶也一樣。在給了你需求之後，他們會不停地研究這些功能，如何才能讓它們變得更好使用。如果，你覺得自己要做的所有工作就是按照用戶最初的需求，並實現了它們，但是在交付的時候，需求已經發生了變化，你的軟件可能不會令他們滿意。在軟件開發過程中，你將自己置於最大的風險中：你生產出了他們曾經要求過的軟件，但卻不是他們現在真正想要的。那最後的結果就是：驚訝、震驚和失望，而不是滿意。

幾年前的一次數值分析課上，老師要求Venkat使用一些偏微分方程式模擬宇宙飛船的運行軌線。

程序基於時間 $t$ 的座標點，計算出在時間 $t + \delta$ 的位置。程序最後繪出來的軌線圖就是如圖4-1中的虛線。



**圖4-1** 計算宇宙飛船的運行軌線

我們發現，估算出來的宇宙飛船位置遠遠地偏離了它的真實位置。萬有引力不是隻在我們計算的座標點上才起作用。實際上，萬有引力一直起作用：它是連續的，而不是離散的。由於忽略了點之間的作用力，我們的計算不斷引入了誤差，所以宇宙飛船最後到達了錯誤的地方。

縮小點之間的間隔（就是 $\delta$ 的值），再運行計算程序，誤差就會減少。這時，估算的位置（如圖4-1中的實線）就和實際位置很接近了。

同理，你的客戶的期望就像宇宙飛船的實際位置。軟件開發的成功就在於最後你離客戶的期望有多近。你計算的每個精確位置，就是一個給客戶演示目前已經成功功能的機會，也正是得到用戶反饋的時候。在你動身進入下一段旅程的時候，這些反饋可以用來糾正你的方向。

我們經常看到，給客戶演示所成功功能的時間與得到客戶需求的時間間隔越長，那麼你就會離最初需求越來越遠。

應該定期地，每隔一段時間，例如一個迭代的結束，就與客戶會晤，並且演示你已經完成的功能特性。

如果你能與客戶頻繁協商，根據他們的反饋開發，每個人都可以從中受益。客戶會清楚你的工作進度。反過來，他們也會提煉需求，然後趁熱反饋到你的團隊中。這樣，他們就會基於自己進化的期望和理解為你導航，你編寫的程序也就越來越接近他們的真實需求。客戶也會基於可用的預算和時間，根據你們真實的工作進度，排列任務的優先級。

較短的迭代週期，會對頻繁的反饋有負面影響嗎？在宇宙飛船軌線的程序中，當 $\delta$ 降低的時候，程序運行就要花費更長的時間。也許你會覺得，使用短的迭代週期會使工作變慢，延遲項目的交付。

讓我們從這個角度思考：兩年來一直拼命地開發項目，直到快結束的時候，你和你的客戶才發現一個基礎功能有問題，而且它是一個核心的需求。你以為缺貨訂單是這樣處理的，但這完全不是客戶所想的東西。現在，兩年之後，你完成了這個系統，寫下了數百萬行的代碼，卻背離了客戶的期望。再怎麼說，兩年來辛苦寫出的代碼有相當大部分要重寫，代價是沉重的。

相反，如果你一邊開發，一邊向他們演示剛完成的功能。項目進展了兩個月的時候，他們說：“等一下，缺貨訂單根本不是這麼一回事。”於是，召開一個緊急會議：你重新審查需求，評估要做多大的改動。這時只要付很少的代價，就可以避免災難了。

要頻繁地獲得反饋。如果你的迭代週期是一個季節或者一年（那就太長了），就應把週期縮短到一週或者兩週。完成了一些功能和特徵之後，去積極獲得客戶的反饋。

## Andy如是說.....

### 維護項目術語表

不一致的術語是導致需求誤解的一個主要原因。企業喜歡用看似普遍淺顯的詞語來表達非常具體、深刻的意義。

我經常看到這樣的事情：團隊中的程序員們，使用了和用戶或者業務人員不同的術語，最後因為“阻抗失調”導致bug和設計錯誤。

為了避免這類問題，需維護一份項目術語表。人們應該可以公開訪問它，一般是在企業內部網或者Wiki上。這聽起來似乎是一件小事情——只是一個術語列表及其定義。但是，它可以幫助你，確保你真正地和用戶進行溝通。

在項目的開發過程中，從術語表中為程序結構——類、方法、模型、變量等選擇合適的名字，並且要檢查和確保這些定義一直符合用戶的期望。



**清晰可見的開發**。在開發的時候，要保持應用可見（而且客戶心中也要了解）。每隔一週或者兩週，邀請所有的客戶，給他們演示最新完成的功能，積極獲得他們的反饋。

## 切身感受

項目啟動了一段時間之後，你應該進入一種舒適的狀態，團隊和客戶建立了一種健康的富有創造性的關係。

突發事件應極少發生。客戶應該能感覺到，他們可以在一定程度上控制項目的方向。

### 跟蹤問題

隨著項目的進展，你會得到很多反饋——修正、建議、變更要求、功能增強、bug修復等。要注意的信息很多。隨機的郵件和潦草的告示帖是無法應付的。所以，要有一個跟蹤系統記錄所有這些日誌，可能是用Web界面的系統。更多詳情參閱*Ship it !* [RG05]。

## 平衡的藝術

- 當你第一次試圖用這種方法和客戶一起工作的時候，也許他們被這麼多的發佈嚇到了。所以，要讓他們知道，這些都是**內部**的發佈（演示），是為了他們自己的利益，不需要發佈給全部的最終用戶。
- 一些客戶，也許會覺得沒有時間應付每天、每週甚至是每兩週的會議。畢竟，他們還有自己的全職工作。

所以要尊重客戶的時間。如果客戶只可以接受一個月一次會議，那麼就定一個月。

- 一些客戶的聯絡人的全職工作就是參加演示會議。他們巴不得每隔1小時就有一次演示和反饋。你會發現這麼頻繁的會議很難應付，而且還要開發代碼讓他們看。縮減次數，只有在你做完一些東西可以給他們演示的時候，大家才碰面。



- 演示是用來讓客戶提出反饋的，有助於駕馭項目的方向。如果缺少功能或者穩定性的時候，不應該拿來演示，那隻能讓人生氣。可以及早說明期望的功能：讓客戶知道，他們看到的是一個正在開發中的應用，而不是一個最終已經完成的產品。

## 17 使用短迭代，增量發佈

“我們為後面的3年制定了漂亮的項目計劃，列出了所有的任務和可交付的時間表。只要我們那時候發佈了產品，就可以佔領市場。”



統一過程和敏捷方法都使用迭代和**增量**開發<sup>①</sup>。使用**增量**開發一次開發應用功能的幾個小組。每一輪的開發都是基於前一次的功能，增加為產品增值的新功能。這時，你就可以發佈或者演示產品。

① 但是，所有減肥方案都會建議你應該少吃多做運動。然而，每份關於如何達到目標的計劃都會不盡相同。

迭代開發是，在小且重複的週期裡，你完成各種開發任務：分析、設計、實現、測試和獲得反饋，所以叫作迭代。

迭代的結束就標記一個里程碑。這時，產品也許可用，也許不可用。在迭代結束時，新的功能全部完成，你就可以發佈，讓用戶真正地使用，同時提供技術支持、培訓和維護方面的資源。每次增加的新功能都會包含多次迭代。

給我一一份詳細的長期計劃，我就會給你一個註定完蛋的項目

**Show me a detailed long-term plan, and I'll show you a project that's doomed**

根據Capers Jones的格言：“.....大型系統的開發是一件非常危險的事情。”大型系統更容易失敗。它們通常不遵守迭代和增量開發的計劃，

或者迭代時間太長（更多關於迭代和演進開發的討論，以及和風險的關係、生產率和缺點，可以查閱*Agile and Iterative Development : A Manager's Guide* [Lar04]一書）。Larman指出，軟件開發不是精細的製造業，而是創新活動。規劃幾年之後客戶才能真正使用的項目註定是行不通的。

對付大項目，最理想的辦法就是小步前進，這也是敏捷方法的核心。大步跳躍大大地增加了風險，小步前進才可以幫助你很好地把握平衡。

在你周圍，可以看到很多迭代和增量開發的例子。比如W3C（萬維網聯盟）提出的XML規範DTD（Document Type Definitions，文檔類型定義），它用來定義XML文檔的詞彙和結構，作為原規範的部分發布。雖然在DTD設計的時候就解決了很多問題，但是在真正使用的時候，又顯現出很多問題和限制。基於用戶的反饋對規範就有了更深一層的理解，這樣就誕生了更加高效的第二代解決方案，例如Schema。如果他們一開始就試圖進行一些完美的設計，也許就看不到XML成為今天的主流了——我們通過提早發佈獲得了灼見和經驗。

大部分用戶都是希望現在就有一個夠用的軟件，而不是在一年之後得到一個超級好的軟件（可以參見《程序員修煉之道——從小工到專家》“足夠好的軟件”一節[HT00]）。確定使產品可用的核心功能，然後把它們放在生產環境中，越早交到用戶的手裡越好。

根據產品的特性，發佈新的功能需要幾周或者幾個月的時間。如果是打算一年或者兩年再交付，你就應該重新評估和重新計劃。也許你要說，構建複雜的系統需要花費時間，你無法用增量的方式開發一個大型的系統。如果這種情況成立，就不要生產大的系統。可以把它分解成一塊塊有用的小系統——再進行增量開發。即使是美國國家航空航天局（NASA）也使用迭代和增量開發方式開發用於航天飛機的複雜軟件（參見*Design, Development, Integration: Space Shuttle Primary Flight Software System* [MR84]）。

詢問用戶，哪些是使產品可用且不可缺少的核心功能。不要為所有可能需要的華麗功能而分心，不要沉迷於你的想象，去做那些華而不實的用戶界面。

有一堆的理由，值得你儘快把軟件交到用戶手中：只要交到用戶手裡，你就有了收入，這樣就有更好的理由繼續為產品投資了。從用戶那裡得到的反饋，會讓我們進一步理解什麼是用戶真正想要的，以及下一步該實現哪些功能。也許你會發現，一些過去認為重要的功能，現在已經不再重要了——我們都知道市場的變化有多快。儘快發佈你的應用，遲了也許它就沒有用了。

使用短迭代和增量開發，可以讓開發者更加專注於自己的工作。如果別人告訴你有一年的時間來完成系統，你會覺得時間很長。如果目標很遙遠，就很難讓自己去專注於它。在這個快節奏的社會，我們都希望更快地得到結果，希望更快地見到有形的東西。這不一定是壞事，相反，它會是一件好事，只要把它轉化成生產率和正面的反饋。

圖4-2描述了敏捷項目主要的週期關係。根據項目的大小，理想的發佈週期是幾周到幾個月。在每個增量開發週期裡，應該使用短的迭代（不應該超過兩週）。每個迭代都要有演示，選擇可能提供反饋的用戶，給他們每人一份最新的產品副本。

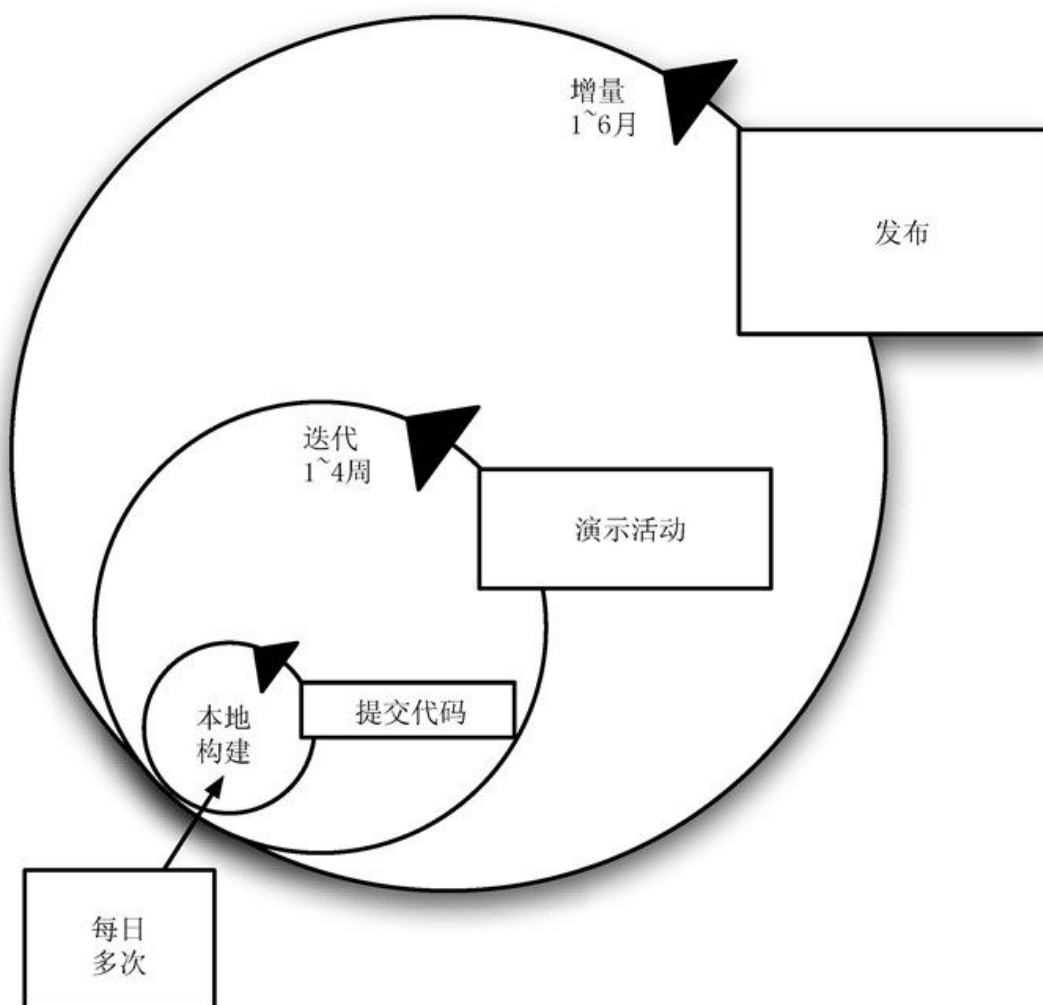


圖4-2 嵌套敏捷開發週期



**增量開發**。發佈帶有最小卻可用功能塊的產品。每個增量開發中，使用1~4周左右迭代週期。

### 切身感受

短迭代讓人感覺非常專注且具效率。你能看到一個實際並且確切的目標。嚴格的最終期限迫使你做出一些艱難的決策，沒有遺留下長期懸而未決的問題。

### 平衡的藝術

- 關於迭代時間長短一直是一個有爭議的問題。Andy曾經遇到這樣一位客戶：他們堅持認為迭代就是4周的時間，因為這是他們學到的。但他們的團隊卻因為這樣的步伐而垂死掙扎，因為他們無法在開發新的代碼的同時又要維護很多已經完成了的代碼。解決方案是，在每4周的迭代中間安排一週的維護任務。沒有規定說迭代必須要緊挨著下一個迭代。
- 如果每個迭代的時間都不夠用，要麼是任務太大，要麼是迭代的時間太短（這是平均數據，不要因為一次迭代的古怪情況而去調整迭代時間）。把握好自己的節奏。
- 如果發佈的功能背離了用戶的需要，那麼多半是因為迭代的週期太長了。用戶的需要、技術和我們對需求的理解，都會隨著時間的推移而變化，在項目發佈的時候，需要清楚地反映出這些變化。如果你發現自己工作時還帶有過時的觀點和陳腐的想法，那麼很可能你等待太長時間做調整了。
- 增量的發佈必須是可用的，並且能為用戶提供價值。你怎麼知道用戶會覺得有價值呢？這當然要去問用戶。

## 18 固定的價格就意味著背叛承諾

“對這個項目，我們必須要有固定的報價。雖然我們還不清楚項目的具體情況，但仍要有一個報價。到星期一，我需要整個團隊的評估，並且我們必須要在年末交付整個項目。”



固定價格的合同會是敏捷團隊的一大難題。我們一直在談論如何用持續、迭代和增量的方式工作。但是現在卻有些人跑過來，想提早知道它會花費多少時間及多少成本。

從客戶方來看，這完全是理所應當的。客戶覺得做軟件就好比是蓋一棟樓房，或者是鋪設一個停車場，等等。為什麼軟件不能像建築業等

其他傳統的行業一樣呢？

也許它真的與建築有很多相似之處——真正的建築行業，但不是我們想象中的建築業。根據英國1998年的一個研究，由於錯誤而返工的成本大約佔整個項目成本的30%<sup>①</sup>。這不是因為客戶的需求變化，也不是物理定律的變化，而是一些簡單錯誤。比如，橫樑太短，窗戶洞太大，等等。這些都是簡單並且為人熟悉的錯誤。

① *Rethinking Construction: The Report of the Construction Task Force*，1998年1月1日，英國副首相辦公室地方政府和地區運輸部文件。

軟件項目會遭遇各種各樣的小錯誤，還要加上基礎需求的變化（不，我要的不是一個工棚，而是一棟摩天大樓），不同個體和團隊的能力差別非常巨大（20倍，甚至更多），當然，還不停地會有新技術出現（從現在開始，釘子就變成圓形的了）。

固定的價格就是保證要背叛承諾

### **A fixed price guarantees a broken promise**

軟件項目天生就是變化無常的，不可重複。如果要提前給出一個固定的價格，就幾乎肯定不能遵守開發上的承諾。那麼我們有什麼可行的辦法呢？我們能做更精確的評估嗎？或者商量出另外一種約定。

根據自己的處境，選擇不同的戰略。如果你的客戶一定要你預先確定項目的報價（比如政府合約），那麼可能你需要研究一些重型的評估技術，比如COCOMO模型或者功能點分析法（Function Point analysis）。但它們不屬於敏捷方法的範疇，並且使用它們也要付出代價。如果這個項目本質上和另一個項目十分相似，並且是同一個團隊開發的，那麼你就好辦了：為一個用戶開發的簡單網站，與下一個會非常相似。

但是，很多項目並不像上面所說的那麼如意。大部分項目都是業務應用，一個用戶和另一個用戶都有著巨大的差別。項目的發掘和創造需要很多配合工作。或許你可以提供稍有不同的安排，試試下面的辦法。

1. 主動提議先構建系統最初的、小的和有用的部分（用建築來打個比方，就是先做一個車庫）。挑選一系列小的功能，這樣完成第一次交付應該不多於6~8周。向客戶解釋，這時候還不是要完成所有的功能，而是要足夠一次交付，並能讓用戶真正使用。
2. 第一個迭代結束時客戶有兩個選擇：可以選擇一系列新的功能，繼續進入下一個迭代；或者可以取消合同，僅需支付第一個迭代的幾周費用，他們要麼把現在的成果扔掉，要麼找其他的團隊來完成它。
3. 如果他們選擇繼續前進。那麼這時候，應該就能很好地預測下一個迭代工作。在下一個迭代結束的時候，用戶仍然有同樣的選擇機會：要麼現在停止，要麼繼續下一個迭代。

對客戶來說，這種方式的好處是項目不可能會死亡。他們可以很早地看到工作的進度（或者不足之處）。他們總是可以控制項目，可以隨時停止項目，不需要繳納任何的違約金。他們可以控制先完成哪些功能，並能精確地知道需要花費多少資金。總而言之，客戶會承擔更低的風險。

而你所做的就是在進行迭代和增量開發。



**基於真實工作的評估**。讓團隊和客戶一起，真正地在當前項目中工作，做具體實際的評估。由客戶控制他們要的功能和預算。

**切身感受**

你的評估數據會在整個項目中發生變化——它們不是固定的。但是，你會覺得自信心在不斷增加，你會越來越清楚每個迭代可以完成的工作。隨著時間的推移，你的評估能力會不斷地提高。

## 平衡的藝術

- 如果你對答案不滿意，那麼看看你是否可以改變問題。
- 如果你是在一個基於計劃的非敏捷環境中工作，那麼要麼考慮一個基於計劃且非敏捷的開發方法，要麼換一個不同的環境。
- 如果你在完成第一個迭代開發之前，拒絕做任何評估，也許你會失去這個合同，讓位於那些提供了評估的人，無論他們做了多麼不切實際的承諾。
- 敏捷不是意味著“開始編碼，我們最終會知道何時可以完成”。你仍然需要根據當前的知識和猜想，做一個大致的評估，解釋如何才能到達這個目標，並給出誤差範圍。
- 如果你現在別無選擇，你不得不提供一個固定的價格，那麼你需要學到真正好的評估技巧。
- 也許你會考慮在合同中確定每個迭代的固定價格，但迭代的數量是可以商量的，它可以根據當前的工作狀況進行調整 [ 又名工作條款說明 ( Statement of Work ) ] 。

## 第5章 敏捷反饋

一步行動，勝過千萬專家的意見。

——**Bill Nye, *The Science Guy*** 科普節目主持人



在敏捷項目中，我們小步前進，不停地收集反饋，時刻矯正自己。但是，這些反饋都是從何而來呢？

在上一章中，我們討論了與用戶一起緊密工作——從他們那裡獲得反饋，並且採取實際的行動。本章中，我們主要討論如何從其他渠道獲得反饋。按照Bill Nye的觀點，實踐是絕對必需的。我們會遵循這一原則，確保你明確知道項目的正確狀態，而不是主觀臆測。

很多項目，都是因為程序代碼失控而陷入困境。修復bug導致了更多的bug，從而又導致了更多的bug修復，成堆的測試卡片最後會把項目壓垮。這時，我們需要的是經常的監督——頻繁反饋以確保代碼不會變壞，如果不會更好，至少能像昨天一樣繼續工作。在第78頁，介紹如何讓**守護天使**來監督你的代碼。

但是，這也不能防止你設計的接口或API變得笨重和難用。這時，你就要先用它再實現它（從第82頁開始介紹）。

當然，不是說一個單元測試在你的機器上能運行，就意味著它可以在其他機器上運行。從第87頁開始，可以看到為什麼**不同環境，就有不同問題**。

現在，你擁有了設計良好的API和乾淨的代碼，就可以看看結果是否符合用戶的期望了。你可以通過**自動驗收測試**來保證代碼是正確的，並且一直都是正確的。我們從第90頁開始談論這個話題。

人人都想清楚瞭解項目的進度狀況，但又很容易誤入歧途，要麼是被一些難懂的指示器誤導，要麼就是錯誤迷信華麗的甘特圖、PERT圖或者日曆工具。其實，你想要的是能**度量真實的進度**，我們會在第93頁介紹它。

儘管，我們已經談論了在開發的時候，與用戶一起工作並及時得到用戶的反饋，但是在其他的比如產品發佈之後的很長一段時間，你還是需要再**傾聽用戶的聲音**，我們會在第96頁詳細解釋。

# 19 守護天使

“你不必為單元測試花費那麼多時間和精力。它只會拖延項目的進度。好歹，你也是一個不錯的程序員——單元測試只會浪費時間，我們現在正處於關鍵時刻。”



## 編寫能產生反饋的代碼

### Coding feedback

代碼在快速地變化。每當你手指敲擊一下鍵盤，代碼就會被改變。敏捷就是管理變化的，而且，代碼可能是變化最頻繁的東西。

為了應對代碼的變化，你需要持續獲得代碼健康狀態的反饋：它是在做你期望的事情嗎？最近一次修改有沒有無意中破壞了什麼功能？這時，你就帶上守護天使，確保所有功能都能正常工作。要做到這樣，就需要自動化單元測試。

現在，一些開發者會對單元測試有意見：畢竟，有“測試”這個詞在裡面，毫無疑問這應該是其他人的工作。從現在開始，忘掉“測試”這個詞。就把它看作是一個極好、編寫能產生反饋的代碼的技術。

先回顧一下，在過去大部分開發者是如何工作的：你寫了一小塊代碼，然後嵌入一些輸出語句，來看一些關鍵變量的值。你也許是在調試器中或者基於一些樁（**stub**）程序來運行代碼。你手工查看所有的運行結果，來修復發現的所有問題，然後扔掉那些樁代碼，或者從調試器中退出，再去解決下一個問題。

敏捷式的單元測試正是採取了相同、相似的過程，並且還讓其更上一層樓。不用扔掉樁程序，你把它保存下來，還要讓其可以自動化地持續運行。你編寫代碼來檢查具體值，而不是手工檢查那些感興趣的變量。

用代碼來測試變量的具體值（以及跟蹤運行了多少個測試），已經是非常普遍的做法。你可以選擇一個標準的測試框架，來幫助你完成簡單的編寫和組織測試的工作，如Java的JUnit、C#或.NET的NUnit、測試Web Service的HttpUnit，等等。實際上，對任何你可以想象到的環境和語言都有對應的單元測試框架，其中的大部分都可以從<http://xprogramming.com/software.htm> 上的列表中找到。

## 清楚自己要測試的內容

讀者David Bock告訴了我們下面這個故事：

“我最近在設計一個特大項目中的一個功能模塊，把構建工具從Ant遷移到Maven。這是在產品中已使用的、沒有任何問題的及經過良好測試的代碼。我不停地工作，一直到深夜，一切都在控制之中。我修改了一部分構建過程，忽然得到了單元測試失敗的警告。我花了很多時間，來查找為什麼修改的代碼會導致測試失敗。最後我放棄了，回滾了修改的代碼，但測試仍然失敗。我開始研究測試代碼，才發現失敗的原因是，測試依賴一個計算次數的工具，而且它還返回一個日期實例，日期設置為第二天中午。我又看了看測試，發現它居然記下了測試執行的時間，並將其作為參數傳遞給另外一個測試。這個方法有個愚蠢的差一錯誤（off-by-one），如果你是在夜裡11點到12點間調用這個方法，它真正的返回值仍然是當天中午，而不是明天。”

從上面的故事中，我們學到了很重要的一課。

- 確保測試是可重複的。使用當前的日期或者時間作為參數，會讓測試依賴運行時間，使用你自己機器上的IP地址同樣會讓它依賴運行時的機器，等等。
- 測試你的邊界條件。11:59:59和0:00:00都是不錯的日期測試邊界條件。

- 不要放過任何一個失敗的測試。在前面的案例中，一個測試一直失敗了，但是因為一段時間內每天都會有幾十個測試失敗，沒有人會注意到這個偽隨機失敗。

只要有了單元測試，就要讓它們自動運行。也就是每次編譯或者構建代碼的時候，就運行一次測試。把單元測試的結果看作是和編譯器一樣——如果測試沒有通過（或者沒有測試），那就像編譯沒有通過一樣糟糕。

接下來就是在後臺架設一個**構建機器**，不斷獲得最新版本的源代碼，然後編譯代碼，並運行單元測試，如果有任何錯誤它會讓你及時知道。

結合本地單元測試，運行每個編譯，構建機器不斷編譯和運行單元測試，這樣你就擁有了一個守護天使。如果出現了問題，你會立刻知道，並且這是最容易修復（也是成本最低）的時候。

一旦單元測試到位，採用這樣的迴歸測試，你就可以隨意重構代碼。可以根據需要進行實驗、重新設計或者重寫代碼：單元測試會確保你不會意外地破壞任何功能。這會讓你心情舒暢，你不用每次寫代碼的時候都如履薄冰。

單元測試是最受歡迎者的一種敏捷實踐，有很多圖書和其他資料可以幫你起步。如果你是一個新手，建議閱讀《單元測試之道》（有Java[HT03]和C# [HT04]版本）。如果要進一步瞭解測試的一些竅門，可以看一下*JUnit Recipes* [Rai04]。

如果想要自動化地連接單元測試（和其他一些有用的東西），可以閱讀《項目自動化之道》[Cla04]。儘管它主要是關於Java的，但也有類似的可以用於.NET環境或者其他環境的工具。

如果你仍然在尋找開始單元測試的理由，下面有很多。

- **單元測試能及時提供反饋** 。你的代碼會重複得到鍛鍊。但若修改或者重寫了代碼，測試用例就會檢查你是否破壞了已有的功能。你可以快速得到反饋，並很容易地修復它們。
- **單元測試讓你的代碼更加健壯** 。測試幫助你全面思考代碼的行為，幫你練習正面、反面以及異常情況。
- **單元測試是有用的設計工具** 。正如我們在實踐20中談論到的，單元測試有助於實現簡單的、注重實效的設計。
- **單元測試是讓你自信の後臺** 。你測試代碼，瞭解它在各種不同條件下的行為。這會讓你在面對新的任務、時間緊迫的巨大壓力之下，找到自信。
- **單元測試是解決問題時的探測器** 。單元測試就像是測試印製電路板的示波鏡。當問題出現的時候，你可以快速地給代碼發送一個脈衝信號。這為你提供了一個很自然的發現和解決問題的方法（見習慣35，第136頁）。
- **單元測試是可信的文檔** 。當你開始學習新API的時候，它的單元測試是最精確和可靠的文檔。
- **單元測試是學習工具** 。在你開始學習新API的時候，可以為這個API寫個單元測試，從而加深自己的理解。這些學習用的測試，不僅能幫助你理解API的行為，還能幫助你快速找到以後可能引入的、無法與現有代碼兼容的變化。



**使用自動化的單元測試** 。好的單元測試能夠為你的代碼問題提供及時的警報。如果沒有到位的單元測試，不要進行任何設計和代碼修改。

**切身感受**

你依賴於單元測試。如果代碼沒有測試，你會覺得很不舒服，就像是在高空作業沒有系安全帶一樣。

## 平衡的藝術

- 單元測試是優質股，值得投資。但一些簡單的屬性訪問方法或者價值不大的方法，是不值得花費時間進行測試的。
- 人們不編寫單元測試的很多借口都是因為代碼中的設計缺陷。通常，抗議越強烈，就說明設計越糟糕。
- 單元測試只有在達到一定測試覆蓋率的時候，才能真正地發揮作用。你可以使用一些測試覆蓋率工具，大致瞭解自己的單元測試的覆蓋情況。
- 不是測試越多質量就會越高，測試必須要有效。如果測試無法發現任何問題，也許它們就是沒有測試對路。

## 20 先用它再實現它

“前進，先完成所有的庫代碼。後面會有大量時間看用戶是如何思考的。現在只要把代碼扔過牆去就可以了，我保證它沒有問題。”



很多成功的公司都是靠著“吃自己的狗食”活著。也就是說，如果要讓你的產品儘可能地好，自己先要積極地使用它。

幸運的是，我們不是在做狗食業務。但是，我們的業務是要創造出能調用的API和可以使用的接口。這就是說，你在說服其他人使用它之前，先得讓自己切實地使用這些接口。事實上，在你剛做完設計但還沒有完成後面的實現的時候，應該使用它。這個可行嗎？

編程之前，先寫測試

## Write tests before writing code

使用被稱為TDD ( Test Driven Development , 測試驅動開發 ) 的技術，你總是在有一個失敗的單元測試後才開始編碼。測試總是先編寫。通常，測試失敗要麼是因為測試的方法不存在，要麼是因為方法的邏輯還不足以讓測試通過。

先寫測試，你就會站在代碼用戶的角度來思考，而不僅僅是一個單純的實現者。這樣做是有很大的區別的，你會發現因為你自己要使用它們，所以能設計一個更有用、更一致的接口。

除此之外，先寫測試有助於消除過度複雜的設計，讓你可以專注於真正需要完成的工作。看看下面編程的例子，這是一個可以兩人玩的“井字棋遊戲”。

開始，你會思考如何為這個遊戲做代碼設計。你也許會考慮需要這些類，例如：`TicTacToeBoard`、`Cell`、`Row`、`Column`、`Player`、`User`、`Peg`、`Score` 和 `Rules`。咱們從 `TicTacToeBoard` 類開始，它就代表了井字棋本身（從遊戲的核心邏輯而不是UI角度說）。

這可能是 `TicTacToeBoard` 類的第一個測試，是用C#在NUnit測試框架下編寫的。它創造了一個遊戲面板，用斷言來檢查遊戲沒有結束。

```
[TestFixture]
public class TicTacToeTest
{
    private TicTacToeBoard board;
    [SetUp]
    public void CreateBoard()
    {
        board = new TicTacToeBoard();
    }
    [Test]
    public void TestCreateBoard()
    {
        Assert.IsNotNull(board);
        Assert.IsFalse(board.GameOver);
    }
}
```

```
}  
}
```

測試失敗，因為類**TicTacToeBoard** 還不存在，你會得到一個編譯錯誤。如果它通過了，你一定很驚訝，不是嗎？這也可能會發生，只是概率很小，但確實可能會發生。在測試通過之前，先要確保測試是失敗的，目的是希望暴露出測試中潛在的**bug**。下面我們來實現這個類。

```
public class TicTacToeBoard {  
    public bool GameOver {  
        get {  
            return false;  
        }  
    }  
}
```

在屬性**GameOver** 中，我們現在只返回**false**。一般情況下，你會用必要的最少代碼讓測試通過。從某種角度上說，這就是在欺騙測試——你知道代碼還沒有完成。但是沒有關係，後面的測試會迫使你再返回來，繼續添加功能。

下一步是什麼呢？首先，你必須決定誰先開始走第一步棋，我們就要設第一個比賽者。先為第一個比賽者寫一個測試。

```
[Test]  
public void TestSetFirstPlayer() {  
    // what should go here?  
}
```



這時，測試會迫使你做一個決定。在完成它之前，你必須決定如何在代碼中表示比賽者，如何把它們分配到面板上。這裡有一個主意。

```
board.SetFirstPlayer(new Player("Mark"), "X");
```

這會告訴面板，遊戲玩家**Mark**使用**X**。

這樣做當然可以，你真的需要**Player** 這個類，或者第一個玩家的名字嗎？也許，稍後你需要知道誰是贏家。但現在它還不是問題。

**YANGI**<sup>①</sup>（你可能永遠都不需要它）原則說過，如果不是真正需要它的時候，你就不應該實現這個功能。基於這一點，現在還沒有足夠的理由表示你需要**Player** 這個類。

① Ron Jeffries創造的詞，它是You Aren't Gonna Need It的縮寫。

別忘了，我們還沒有實現**TicTacToeBoard** 類中的**SetFirstPlayer()** 方法，並且還沒有寫**Player** 這個類。我們仍然是先寫一個測試。我們假設下面的代碼是用來設置第一個玩家的。

```
board.SetFirstPlayer("X");
```

它表示設**X**為第一個玩家，比第一個版本要更簡單。但是，這個版本隱藏著風險：你可以傳任何字母給**SetFirstPlayer()** 方法，這就意味著你必須添加代碼來檢查參數是**O**還是**X**，並且需要知道如果它不

是這兩個值的時候該如何處理。因此要更進一步簡單化。我們有一個簡單的標誌來標明第一個玩家是 *O* 還是 *X*。知道了這個，我們現在就可以寫單元測試了。

```
[Test]
public void TestSetFirstPlayer() {
    board.FirstPlayerPegIsX = true;
    Assert.IsTrue(board.FirstPlayerPegIsX);
}
```

我們可以將 `FirstPlayerPegIsX` 設為布爾類型的屬性，並把它設為期望的值。這看起來挺簡單的，也容易使用，比複雜的 `Player` 類容易很多。測試寫好了，你就可以通過在 `TicTacToeBoard` 類中實現 `FirstPlayerPegIsX` 屬性，讓測試通過。

你看，我們是以 `Player` 類開始，最後卻只使用了簡單的布爾類型屬性。這是如何做到的呢？這種簡化就是在編寫代碼之前讓測試優先實現的。

但記住，我們不是要扔掉好的設計，就只用大量的布爾類型來編碼所有的東西。這裡的重點是：什麼是成功地實現特定功能的最低成本。總之，程序員很容易走向另一個極端——一些不必要的過於複雜的事情——測試優先會幫助我們，防止我們走偏。

消除那些還沒有編寫的類，這會很容易地簡化代碼。相反，一旦你已經編寫了代碼，也許會強迫自己保留這些代碼，並繼續使用它（即使代碼已經過期作廢很久了）。

好的設計並不意味著需要更多的類

**Good design doesn't mean more classes**

當你開發設計面向對象系統的時候，可能會迫使自己使用對象。有一種傾向認為，面向對象的系統應該由對象組成，我們迫使自己創建越來越多的對象類，不管它們是否真的需要。添加無用代碼總是不好的想法。

TDD有機會讓你編寫代碼之前（或者至少在深入到實現之前），可以深思熟慮將如何用它。這會迫使你思考它的可用性和便利性，並讓你的設計更加注重實效。

當然，設計不是在開始編碼的時候就結束了。你需要在它的生命週期中持續地添加測試，添加代碼，並重新設計代碼（更多信息參閱第113頁習慣28）。



**先用它再實現它**。將TDD作為設計工具，它會為你帶來更簡單更有實效的設計。

## 切身感受

這種感覺就是，只有在有具體理由的時候才開始編碼。你可以專注於設計接口，而不會被很多實現的細節干擾。

## 平衡的藝術

- 不要把測試優先和提交代碼之前的測試等同起來。測試先行可以幫助你改進設計，但是你還是需要在提交代碼之前做測試。
- 任何一個設計都可以被改進。
- 你在驗證一個想法或者設計一個原型的時候，單元測試也許並不適合。但是，萬一這些代碼不幸倉促演變成了一個真正的系統，就必須要為它們添加測試（但是最好能重新開始設計系統）。

- 單純的單元測試無法保證好的設計，但它們會對設計有幫助，會讓設計更加簡單。

## 21 不同環境，就有不同問題

“只要代碼能在你的機器上運行就可以了，誰會去關心它是否可以在其他平臺上工作。你又不需其他平臺。”



如果廠商或者同事說了這樣的套話：“哦，那不會有什麼不同。”你可以打賭，他們錯了。只要環境不同，就很可能會有不同的問題。

Venkat真正在項目中學到了這一課。他的一個同事抱怨說，Venkat的代碼失敗了。但奇怪的是，問題在於，這與在Venkat機器上通過的一個測試用例一模一樣。實際上，它在一臺機器上可以工作，在另一臺機器上就不工作。

最後，他們終於找到了罪魁禍首：一個.NET環境下的API在Windows XP和Windows 2003<sup>①</sup>上的行為不同。平臺的不同，造成了結果的不一樣。

① 參見*.NET Gotchas* 中的Gotcha #74[Sub05]。

他們算是幸運的，能夠偶然發現這個問題。否則，很可能在產品投入使用的時候才會發現。如果很晚才發現這個問題，成本會非常昂貴——想象一下產品發佈之後，才發現它並不支持應該支持的平臺，那會怎麼樣。

也許，你會要求測試團隊在所有支持的平臺上進行測試。如果他們是手工進行測試，可能並不是最可靠的測試辦法。我們需要更加面向開發者的測試辦法。

你已經編寫了單元測試，測試你的代碼。每次在修改或者重構代碼的時候，在提交代碼之前，你會運行測試用例。那麼現在所要做的就是

在各種支持的平臺和環境中運行這些測試用例。

如果你的應用程序要在不同的操作系統上運行（例如MacOS、Linux、Windows等），或者一個操作系統的不同版本（例如Windows 2000、Windows XP、Windows 2003等），你需要測試所有的操作系統。如果你的應用程序要在不同版本的Java虛擬機或者不同的.NET CLR中運行，你也需要測試它們。

### **Andy如是說.....**

#### **但是它在我的機器上可以工作**

曾經有這樣一個客戶，他需要提高他們的OS/2系統性能。於是一個莽撞的開發人員打算用匯編從頭開始重寫OS/2的調度程序。

從某種程度上說，事實上它是可以工作的。它在最初的開發人員的機器上工作得非常好，但是在其他人的機器上就不能用。他們甚至嘗試了從同一個廠商那裡購買硬件，安裝相同版本的操作系統、數據庫和其他的工具，但都徒勞無功。

他們甚至嘗試在每天的同一個時間，以同一個方向面朝機器，宰雞向眾神祭祀，希望能有好運（呵呵，這是我杜撰的，但其他都是真實的）。

團隊最終只好放棄了這個方案。與沒有文檔的內部操作系統糾纏在一起，絕對是非常脆弱的。這不是敏捷的做法。

---

### **使用自動化會節省時間**

#### **Automate to save time**

但是，也許你已經有時間壓力了，因此，你怎麼可能有時間在多個平臺上運行測試呢？這就要靠持續集成②來拯救了。

② 閱讀Martin Fowler 寫的一篇重要的文章*Continuous Integration*

<http://www.martinfowler.com/articles/continuousIntegration.html>

。

我們在前面的**保持可以發佈**中學過，用一個持續集成工具，週期性地從源代碼控制系統中取得代碼，並運行代碼。如果有任何測試失敗了，它會通知相關的開發者。通知方式可能是電子郵件、頁面、RSS Feed，或者其他一些新穎的方式。

要在多個平臺上測試，你只要為每個平臺設置持續集成系統就行了。當你或者同事提交了代碼，測試會在每個平臺上自動運行。這樣，提交代碼之後的幾分鐘，你就可以知道它是否可以在不同的平臺上運行！這是多麼英明的辦法呀！

構建機器的硬件成本相當於開發人員的幾個小時而已。如果需要，你甚至可以使用像VMware 或Virtual PC這樣的虛擬機產品，在一臺機器上運行不同版本的操作系統、VM或CLR。



**不同環境，就有不同問題**。使用持續集成工具，在每一種支持的平臺和環境中運行單元測試。要積極地尋找問題，而不是等問題來找你。

## 切身感受

感覺就像是在做單元測試，非但如此，而且還是跨越不同的世界的單元測試。

## 平衡的藝術

- 硬件比開發人員的時間便宜。但如果你有很多配置，要支持大量的平臺，可以選擇哪些平臺需要內部測試。

- 只因為不同的棧層順序、不同的單詞大小寫等，就能發現很多平臺上的bug。因此，即使運行用Solaris的客戶比用Linux的少很多，你仍然要在兩個系統上都進行測試。
- 你不希望因為一個錯誤而收到5次通知轟炸（這就像是雙重徵稅，會導致電子郵件疲勞症）。可以設置一個主構建平臺或者配置，降低其他構建服務器的運行頻率，這樣在它失敗的時候，你就有足夠的時間來修復主構建平臺。或者彙總所有錯誤報告信息到一個地方，進行統一處理。

## 22 自動驗收測試

“很好，你現在用單元測試來驗證代碼是否完成了你期望的行為。發給客戶吧。我們很快會知道這是否是用戶期望的功能。”



你與用戶一起工作，開發他們想要的功能。但現在，你要能確保他們得到的數據是正確的，至少在用戶看來它是正確的。

幾年前，Andy做了一個項目。在項目中，他們的行業標準規定凌晨12:00點是一天的最後一分鐘，12:01是一天最早一分鐘（一般情況下，商業計算機系統認為凌晨11:59是一天的最後一分鐘，12:00是一天最早一分鐘）。在驗收測試的時候，這個很小的細節導致一個嚴重的問題——無法進行正確的合計。

關鍵業務邏輯必須要獨立進行嚴格的測試，並且最後需要通過用戶的審批。

但你也不可能拉著用戶，逐一檢查每個單元測試的運行結果。實際上，你需要能自動比較用戶期望和實際完成的工作。

有一個辦法可以使驗收測試不同於單元測試。你應該讓用戶在不必學習編碼的情況下，根據自己的需要進行添加、更新和修改數據。你有很多方法來實現它。

Andy使用了一些架構，把測試數據放到一個平面文件中，並且用戶可以直接修改這些數據。Venkat使用Excel做過類似的事情。根據環境的不同，也可以找出一種能讓用戶自然接收的方法（數據可以在平面文件、Excel文件、數據庫中）。或者可以考慮選擇一個現成的測試工具，它們會為你完成很多功能。

FIT<sup>①</sup>，即集成測試框架，它很實用，可以更容易地使用HTML表格定義測試用例，並比較測試結果數據。

① <http://fit.c2.com>。

---

## Venkat如是說.....

### 獲取驗收數據

一個客戶以前使用過Excel開發的定價模型。我們就通過寫測試，比較應用的價格輸出結果是否與Excel的一致，然後，必要的話，糾正應用中的邏輯和公式。這樣用戶可以簡單地修改驗收測試標準，定價相關的關鍵業務邏輯是正確的，每個人對項目都很有信心。

使用FIT，客戶可以定義帶有新功能的使用樣本。客戶、測試人員和開發人員（根據樣本）都可以創建表格，為代碼描述可能的輸入和輸出值。開發人員會參照帶有正開發的代碼結果的FIT表格中的樣本編寫測試代碼。測試結果成功或者失敗，都會顯示在HTML頁面中，用戶可以很方便地查閱。

如果領域專家提供了業務的算法、運算或者方程式，為他們實現一套可以獨立運行的測試（參見第136頁習慣35）。要讓這些測試都成為測試套件的一部分，你會在項目生命週期中確保持續為它們提供正確的答案。





**為核心的業務邏輯創建測試**。讓你的客戶單獨驗證這些測試，要讓它們像一般的測試一樣可以自動運行。

## 切身感受

它像是協作完成的單元測試：你仍然是在編寫測試，但從其他人那裡獲得答案。

## 平衡的藝術

- 不是所有客戶都能給你提供正確的數據。如果他們已經有了正確的數據，就根本不需要新系統了。
- 你也許會在舊系統（也許是電腦系統，也許是人工系統）中發現以前根本不知道的bug，或者以前不存在的真正問題。
- 使用客戶的業務邏輯，但是不要陷入無邊無際的文檔寫作之中。

# 23 度量真實的進度

“用自己的時間表報告工作進度。我們會用它做項目計劃。不用管那些實際的工作時間，每週填滿**40**小時就可以了。”



時間的消逝（通常很快）可以證明：判斷工作進度最好是看實際花費的時間而不是估計的時間。

哦，你說早已經用時間表進行了追蹤。不幸的是，幾乎所有公司的時間表都是為工資會計準備的，不是用來度量軟件項目的開發進度的。例如，如果你工作了**60**個小時，也許你的老闆會讓你在時間表上只填寫**40**個小時，這是公司會計想看到的。所以，時間表很難真實地反映工作完成狀況，因此它不可以用來進行項目計劃、評估或表現評估。

## 專注於你的方向

### Focus on where you're going

即使沒有時間表，一些開發人員還是很難面對現實瞭解自己的真實進度。你曾經聽到開發人員報告一個任務完成了**80%**嗎？然而過了一天又一天，一週又一週，那個任務仍然是完成了**80%**？隨意用一個比率進行度量是沒有意義的，這就好比是說**80%**是**對的**（除非你是政客，否則**對**和**錯**應該是布爾條件）。所以，我們不應該去計算工作量完成的百分比，而應該測定還剩下多少工作量沒有完成。如果你最初估計這個任務需要**40**個小時，在開發了**35**個小時之後，你認為還需要另外**30**個小時的工作。那就得到了很重要的度量結果（這裡誠實非常重要，隱瞞真相毫無意義）。

在你最後真正完成一項任務時，要清楚知道完成這個任務**真正**花費的時間。奇怪的是，它花費的時間很可能要比最初估計時間長。沒有關係，我們希望這能作為下一次的參考。在為下一個任務估計工作量時，可以根據這次經驗調整評估。如果你低估了一個任務，評估是**2**天，它最後花費了**6**天，那麼係數就是**3**。除非是異常情況，否則你應該對下次估計乘以係數**3**。你的評估會波動一段時間，有時候過低估計，有時候過高估計。但隨著時間的推移，你的評估會與事實接近，你也會對任務所花費的時間有更清楚的認識。

### Andy如是說.....

#### 登記時間

我的小姨子曾經在某個大型國際諮詢公司中工作。每天每隔**6**分鐘她們就得登記她們的時間。

她們甚至有代碼來專門記錄填表登記時間所花費的時間。這個代碼不是**0**、**9999**或者一些容易記的代碼，而是類似**948247401299-44b**這麼一個臨時的代碼。

這就是為什麼你不願意把會計部門的規則和約束摻合到項目中的原因。

如果能一直讓下一步工作是可見的，會有助於進度度量。最好的做法就是使用**待辦事項**（backlog）。

待辦事項就是等待完成的任務列表。當一個任務被完成了，它就會從列表中移除（邏輯上的，而物理上就是把它從列表中劃掉，或者標識它是完成的狀態）。當添加新任務的時候，先排列它們的優先級，然後加入到待辦事項中。你也可以有個人的待辦事項、當前迭代的待辦事項或者整個項目的待辦事項。①

① 使用待辦事項及個人與項目管理工具的列表的更多信息，參考 *Ship It!* [RG03]。

通過代辦事項，就可以隨時知道下一步最重要的任務是什麼。同時，你的評估技巧也在不停地改進，你也會越來越清楚完成一項任務要花費的時間。

清楚項目的真實進度，是一項強大的技術。



**度量剩下的工作量**。不要用不恰當的度量來欺騙自己或者團隊。要評估那些需要完成的待辦事項。

---

## Scrum方法中的sprint

在Scrum開發方法中（Sch04），每個迭代被稱作**sprint**，通常為30天時間。**sprint**的待辦事項列表是當前迭代任務列表，它會評估剩下的工作量，顯示每個任務還需要多少小時可以完成。

每個工作日，每個團隊成員會重新評估完成一個任務還需要多少小時。不管怎麼樣，只要所有任務的評估總和超過了一個迭代剩餘的

時間，那麼任務就必須移到下一個迭代中開發。

如果每月還有一些剩餘的時間，你還可以添加新的任務。這樣做，客戶一定會非常喜歡。

## 切身感受

你會覺得很舒服，因為你很清楚哪些任務已經完成，哪些是沒有完成的，以及它們的優先級。

## 平衡的藝術

- 6分鐘作為一個時間單位，它的粒度實在太細了，這不是敏捷的做法。
- 一週或者一個月的時間單元，它的粒度太粗了，這也不是敏捷的做法。
- 關注功能，而不是日程表。
- 如果你在一個項目中花費了很多時間來了解你所花費的時間，而沒有足夠的時間進行工作，那麼你在瞭解你所花費的時間上花費的時間就太多了。聽懂了嗎？
- 一週工作40個小時，不是說你就有40個小時的編碼時間。你需要減去會議、電話、電子郵件以及其他相關活動的時間。

# 24 傾聽用戶的聲音

“用戶就是會抱怨。這不是你的過錯，是用戶太愚蠢了，連使用手冊都看不懂。它不是一個bug，只是用戶不明白如何使用而已。他們本應該知道更多。”



Andy曾經在一家大公司工作過，為高端的Unix工作站開發產品。在這個環境中，你不是簡單地運行`setup.exe` 文件或者`pkgadd` 命令，就可以完成軟件的安裝。你必須在工作站上覆制文件並調整各種設置。

Andy和他的團隊成員們覺得一切都工作得很順利。直到一天，Andy走過技術支持部門的工作間，聽到一個技術支持工程師對著電話大笑：“哦，這不是bug，你只是犯了一個每個人都會犯的錯誤。”並且，不只是這一個工程師，整個部門都在嘲笑這些可憐、天真和愚蠢的客戶。

## 這是一個bug

### It is a bug

倒黴的客戶必須要配置那些包含了一些魔術數字的模糊系統文件，否則系統根本不會運行。系統既沒有錯誤提示消息，也不會崩潰，只是顯示大黑屏和一個斗大的“退出”按鈕。事實上，安裝說明書中有一行提到了這樣的問題，但顯然80%的用戶忽略了這個信息，因此只能求助公司的技術支持部門，並遭到他們的嘲笑。

正如我們在第128頁第7章中所說，當出了錯誤，你要儘可能地提供詳細信息。黑屏和含義不明的“退出”按鈕是很不友好的行為。更糟糕的是，在得到用戶反饋的時候，還嘲笑用戶愚蠢，而不去真正地解決問題。

不管它是否是產品的bug，還是文檔的bug，或者是對用戶社區理解的bug，它都是團隊的問題，而不是用戶的問題。

下面一個案例是：一個昂貴的专业車間的控制系統，沒有任何一個用戶會使用。因為，使用系統的第一步是要輸入用戶名和密碼，進行登錄。但這個車間的大部分工人都是文盲，沒有人去問過他們，也沒有去收集他們的反饋。就這樣，為用戶安裝了一個無用的系統。最後，花費巨大的費用，開發人員重新開發了一個基於圖片的使用界面。

我們花費了很大的精力從單元測試之類的代碼中獲得反饋，但卻容易忽略最終用戶的反饋。你不僅需要和真實用戶（不是他們的經理，也不是業務分析師之類的代理人）進行交談，還需要耐心地傾聽。

即使他們說的內容很傻！



**每一個抱怨的背後都隱藏了一個事實。** 找出真相，修復真正的問題。

## 切身感受

對客戶的那些愚蠢抱怨，你既不會生氣，也不會輕視。你會查看一下，找出背後真正的問題。

## 平衡的藝術

- 沒有愚蠢的用戶。
- 只有愚蠢、自大的開發人員。
- “它就是這樣的。”這不是一個好的答案。
- 如果代碼問題解決不了，也許可以考慮通過修改文檔或者培訓來彌補。
- 你的用戶有可能會閱讀所有的文檔，記住其中的所有內容。但也可能不會。

# 第6章 敏捷編碼

任何一個笨蛋都能夠讓事情變得越來越笨重、越來越複雜、越來越極端。需要天才的指點以及許多的勇氣，才能讓事情向相反的方向發展。

### ——John Dryden<sup>①</sup>，書信集10：至Congreve

① John Dryden ( 1631—1700 )，英國第一位受封的“桂冠詩人”，英國古典主義時期重要的批評家和戲劇家，英國古典主義代表人物之一。——譯者注

新項目剛開始著手開發時，它的代碼很容易理解和上手。然而，隨著開發過程的推進，項目不知不覺中演變為一個龐然怪物。發展到最後，往往需要投入更多的精力、人力和物力來讓它繼續下去。

開始看起來非常正常的項目，是什麼讓它最終變得難以掌控？開發人員在完成任務時，可能會難以抵擋誘惑為節省時間而走“捷徑”。然而，這些“捷徑”往往只會推遲問題的爆發時間，而不是把它徹底解決掉（如同第15頁習慣2中的情況一樣）。當項目時間上的壓力增加時，問題最終還是會在項目團隊面前出現，讓大家心煩意亂。

如何保證項目開發過程中壓力正常，而不是在後期面對過多的壓力，以致噩夢連連呢？最簡單的方式，就是在開發過程中便細心“照看”代碼。在編寫代碼時，每天付出一點小的努力，可以避免代碼“腐爛”，並且保證應用程序不至變得難以理解和維護。

開發人員使用本章的實踐習慣，可以保證開發出的代碼無論是在項目進行中還是在項目完成後，都易於理解、擴展和維護。這些習慣會幫助你對代碼進行“健康檢查”，以防止它們變成龐然怪物。

首先，第100頁中的習慣是：**代碼要清晰地表達意圖**。這樣的代碼清晰易懂，僅憑小聰明寫出的程序很難維護。註釋可以幫助理解，也可能導致不好的干擾，應該總是用**代碼溝通**（見第105頁）。在工程項目中沒有免費的午餐，開發人員必須判斷哪些東西更加重要，每個決

策會造成什麼後果，也就是說要**動態評估取捨**（見第110頁）以得到最佳的決策。

項目是以增量式方式進行開發的，寫程序時也應該進行**增量式編程**（見第113頁）。在編寫代碼的時候，要想**保持簡單**很難做到——實際上，想寫出簡單的代碼要遠比寫出令人厭惡的、過分複雜的代碼難得多。不過這樣做絕對值得，見第115頁。

我們將在第117頁談到，良好的面向對象設計原則建議：應該**編寫內聚的代碼**。要保持代碼條理清晰，應該遵循如第121頁上所述的習慣：**告知，不要詢問**。最後，通過設計能夠**根據契約進行替換**的系統（見124頁），可以在不確定的未來中保持代碼的靈活性。

## 25 代碼要清晰地表達意圖

“可以工作而且易於理解的代碼當然好，但是讓人覺得聰明更加重要。別人給你錢是因為你腦子好使，讓我們看看你到底有多聰明。”



### Hoare 談軟件設計

設計軟件有兩種方式。一種是設計得儘量簡單，並且明顯沒有缺陷。另一種方式是設計得儘量複雜，並且沒有明顯的缺陷。

——C.A.R. Hoare<sup>①</sup>

<sup>①</sup> Hoare, 全名Charles Antony Richard Hoare，簡稱C.A.R. Hoare，生於1934年1月11日，英國計算機科學家，發明了排序算法中的“快速排序”算法。圖靈獎得主。——譯者注

我們大概都見過不少難以理解和維護的代碼，而且（最壞的是）還有錯誤。當開發人員們像一群旁觀者見到UFO一樣圍在代碼四周，同樣



也感到恐懼、困惑與無助時，這個代碼的質量就可想而知了。如果沒有人理解一段代碼的工作方式，那這段代碼還有什麼用呢？

開發代碼時，應該更注重可讀性，而不是隻圖自己方便。代碼閱讀的次數要遠遠超過編寫的次數，所以在編寫的時候值得花點功夫讓它讀起來更加簡單。實際上，從衡量標準上來看，代碼清晰程度的優先級應該排在執行效率之前。

例如，如果默認參數或可選參數會影響代碼可讀性，使其更難以理解和調試，那最好明確地指明參數，而不是在以後讓人覺得迷惑。

在改動代碼以修復bug或者添加新功能時，應該有條不紊地進行。首先，應該理解代碼做了什麼，它是如何做的。接下來，搞清楚將要改變哪些部分，然後著手修改並進行測試。作為第1步的理解代碼，往往是最難的。如果別人給你的代碼很容易理解，接下來的工作就省心多了。要敬重這個黃金法則②，你欠他們一份情，因此也要讓你自己的代碼簡單、便於閱讀。

② 黃金法則（Golden Rule），起源於《聖經》（太7:12）：“無論何事，你們願意人怎樣待你們，你們也要怎樣待人。”——編者注

明白地告訴閱讀程序的人，代碼都做了什麼，這是讓其便於理解的一種方式。讓我們看一些例子。

```
coffeeShop.PlaceOrder(2);
```

通過閱讀上面的代碼，可以大致明白這是要在咖啡店中下一個訂單。但是，2到底是什麼意思？是意味著要兩杯咖啡？要再加兩次？還是杯子的大小？要想搞清楚，唯一的方式就是去看方法定義或者文檔，因為這段代碼沒有做到清晰易懂。

所以我們不妨添加一些註釋。

```
coffeeShop.PlaceOrder(2 /* large cup */);
```

現在看起來好一點了，不過請注意，註釋有時候是為了幫寫得不好的代碼補漏（見第105頁習慣26：用代碼溝通）。

Java 5與.NET中有枚舉值的概念，我們不妨使用一下。使用C#，我們可以定義一個名為CoffeeCupSize 的枚舉，如下所示。

```
public enum CoffeeCupSize
{
    Small,
    Medium,
    Large
}
```

接下來就可以用它來下單要咖啡了。

```
coffeeShop.PlaceOrder(CoffeeCupSize.Large);
```

這段代碼就很明白了，我們是要一個大杯 的咖啡。

③ 對星巴克的粉絲來說，這是指*venti*

作為一個開發者，應該時常提醒自己是否有辦法讓寫出的代碼更容易理解。下面是另一個例子。

```
Line 1 public int compute(int val)
      - {
      -     int result = val << 1;
      -     //... more code ...
      5     return result;
      - }
```

第3行中的位移操作符是用來幹什麼的？如果善於進行位運算，或者熟悉邏輯設計或彙編編程，就會明白我們所做的只是把`val` 的值乘以2。

#### PIE④原則

④ PIE=Program Intently and Expressively, 即意圖清楚而且表達明確地編程。——編者注

代碼必須明確說出你的意圖，而且必須富有表達力。這樣可以讓代碼更易於被別人閱讀和理解。代碼不讓人迷惑，也就減少了發生潛在錯誤的可能。一言以蔽之，代碼應意圖清晰，表達明確。

但對沒有類似背景的人們來說，又會如何——他們能明白嗎？也許團隊中有一些剛剛轉行做開發、沒有太多經驗的成員。他們會撓頭不已，直到把頭髮抓下來⑤。代碼執行效率也許很高，但是缺少明確的意圖和表現力。

⑤ 沒錯，那不是一塊禿頂，而是一個編程機器的太陽能電池板。

用位移做乘法，是在對代碼進行不必要且危險的性能優化。  
`result=val*2` 看起來更加清晰，也可以達到目的，而且對於某種給

定的編譯器來說，可能效率更高（**懂得丟棄**，見第34頁習慣7）。不要表現得好像很聰明似的，要遵循**PIE**原則：代碼要清晰地表達意圖。

要是違反了**PIE**原則，造成的問題可就不只是代碼可讀性那麼簡單了——它會影響到代碼的正確性。下列代碼是一個**C#**方法，試圖同步對**CoffeeMaker** 中**MakeCoffee()** 方法進行調用。

```
Public void MakeCoffee()
{
    lock(this)
    {
        // ... operation
    }
}
```

這個方法的作者想設置一個臨界區（**critical section**）——任何時候最多只能有一個線程來執行操作中的代碼。要達到這個目的，作者在**CoffeeMaker** 實例中聲明瞭一個鎖。一個線程只有獲得這個鎖，才能執行這個方法。（在**Java**中，會使用**synchronized** 而不是**lock**，不過想法是一樣的。）

對於**Java**或**.NET**程序員來說，這樣寫順理成章，但是其中有兩個小問題。首先，鎖的使用影響範圍過大；其次，對一個全局可見的對象使用了鎖。我們進一步來看看這兩個問題。

假設**Coffeemaker** 同時可以提供熱水，因為有些人希望早上能夠享用一點伯爵紅茶。我想同步**GetWater()** 方法，因此調用其中的**lock(this)**。這會同步任何在**CoffeeMaker** 上使用**lock** 的代碼，也就意味著不能同時製作咖啡以及獲取熱水。這是開發者原本的意圖嗎？還是鎖的影響範圍太大了？通過閱讀代碼並不能明白這一點，使用代碼的人也就迷惑不已了。

同時，`MakeCoffee()` 方法的實現在 `CoffeeMaker` 對象上聲明瞭一個鎖，而應用的其他部分都可以訪問 `CoffeeMaker` 對象。如果在一個線程中鎖定了 `CoffeeMaker` 對象實例，然後在另外一個線程中調用那個實例之上的 `MakeCoffee()` 方法呢？最好的狀況也會執行效率很差，最壞的狀況會帶來死鎖。

讓我們在這段代碼上應用 **PIE** 原則，通過修改讓它變得更加明確吧。我們不希望同時有兩個或更多的線程來執行 `MakeCoffee()` 方法。那為什麼不能為這個目的創建一個對象並鎖定它呢？

```
Private object makeCoffeeLock = new Object();

Public void MakeCoffee()
{
    lock(makeCoffeeLock)
    {
        // ... operation
    }
}
```

這段代碼解決了上面的兩個問題——我們通過指定一個外部對象來進行同步操作，而且更加明確地表達了意圖。

在編寫代碼時，應該使用語言特性來提升表現力。使用方法名來傳達意向，對方法參數的命名要幫助讀者理解背後的想法。異常傳達的信息是哪些可能會出問題，以及如何進行防禦式編程，要正確地使用和命名異常。好的編碼規範可以讓代碼變得易於理解，同時減少不必要的註釋和文檔。



**要編寫清晰的而不是討巧的代碼**。向代碼閱讀者明確表明你的意圖。可讀性差的代碼一點都不聰明。


## 切身感受

應該讓自己或團隊的其他任何人，可以讀懂自己一年前寫的代碼，而且只讀一遍就知道它的運行機制。

## 平衡的藝術

- 現在對你顯而易見的事情，對別人可能並非如此，對於一年以後的你來說，也不一定顯而易見。不妨將代碼視作不知道會在未來何時打開的一個時間膠囊。
- 不要明日復明日。如果現在不做的話，以後你也不會做的。
- 有意圖的編程並不是意味著創建更多的類或者類型。這不是進行過分抽象的理由。
- 使用符合當時情形的耦合。例如，通過散列表進行松耦合，這種方式適用於在實際狀況中就是松耦合的組件。不要使用散列表存儲緊密耦合的組件，因為這樣沒有明確表示出你的意圖。

## 26 用代碼溝通

“如果代碼太雜亂以至於無法閱讀，就應該使用註釋來說明。精確地解釋代碼做了什麼，每行代碼都要加註釋。不用管為什麼要這樣編碼，只要告訴我們到底是怎麼做的就好了。” 

通常程序員都很討厭寫文檔。這是因為大部分文檔都與代碼沒有關係，並且越來越難以保證其符合目前的最新狀況。這不只違反了DRY原則（不要重複你自己，**Don't Repeat Yourself**，見[HT00]），還會產生使人誤解的文檔，這還不如沒有文檔。

建立代碼文檔無外乎兩種方式：利用代碼本身；利用註釋來溝通代碼之外的問題。

## 不要用註釋來包裹你的代碼

### Don't comment to cover up

如果必須通讀一個方法的代碼才能瞭解它做了什麼，那麼開發人員先要投入大量的時間和精力才能使用它。反過來講，只需短短几行註釋說明方法行為，就可以讓生活變得輕鬆許多。開發人員可以很快瞭解到它的意圖、它的期待結果，以及應該注意之處——這可省了你不少勁兒。

應該文檔化你所有的代碼嗎？在某種程度上說，是的。但這並不意味著要註釋絕大部分代碼，特別是在方法體內部。源代碼可以被讀懂，不是因為其中的註釋，而應該是由於它本身優雅而清晰——變量名運用正確、空格使用得當、邏輯分離清晰，以及表達式非常簡潔。

如何命名很重要。程序元素的命名是代碼讀者必讀的部分。①通過使用細心挑選的名稱，可以向閱讀者傳遞大量的意圖和信息。反過來講，使用人造的命名範式（比如現在已經無人問津的匈牙利表示法）會讓代碼難以閱讀和理解。這些範式中包括的底層數據類型信息，會硬編碼在變量名和方法名中，形成脆弱、僵化的代碼，並會在將來造成麻煩。

① 在《地海巫師》（*The Wizard of Earthsea*）系列書籍中，知道一件事物的真實名稱可以讓一個人對它實施完全的控制。通過名稱來進行魔法控制，是文學和神話中一種常用的主題，在軟件開發中也是如此。

使用細心挑選的名稱和清晰的執行路徑，代碼幾乎不需要註釋。實際上，當Andy和Dave Thomas聯手寫作第一本關於Ruby編程語言的書籍時（即參考文獻[TH01]），他們只要閱讀將會在Ruby解釋器中執行的代碼，幾乎就可以把整個Ruby語言的相關細節記錄下來了。代碼能夠自解釋，而不用依賴註釋，是一件很好的事情。Ruby的創建者松本行

弘是日本人，而Andy和Dave除了“sukiyaki”（一種日式火鍋）和“sake”（日本清酒）之外，一句日語也不會。

如何界定一個良好的命名？良好的命名可以向讀者傳遞大量的正確信息。不好的命名不會傳遞任何信息，糟糕的命名則會傳遞**錯誤**的信息。

例如，一個名為readAccount()的方法實際所做的卻是向硬盤寫入地址信息，這樣的命名就被認為是很糟糕的（是的，這確實發生過，參見[HT00]）。

foo 是一個具有歷史意義、很棒的臨時變量名稱，但是它沒有傳遞作者的任何意圖。要儘量避免使用神秘的變量名。不是說命名短小就等同於神秘：在許多編程語言中通常使用i 來表示循環索引變量，s 常被用來表示一個字符串。這在許多語言中都是慣用法，雖然都很短，但並不神秘。在這些環境中使用s 作為循環索引變量，可真的不是什麼好主意，名為indexvar 的變量也同樣不好。不必費盡心機去用繁複冗長的名字替換大家已習慣的名稱。

對於顯而易見的代碼增加註釋，也會有同樣的問題，比如在一個類的構造方法後面添加註釋//Constructor 就是多此一舉。但很不幸，這種註釋很常見——通常是由過於熱心的IDE插入的。最好的狀況下，它不過是為代碼添加了“噪音”。最壞的狀況下，隨著時間推進，這些註釋則會過時，變得不再正確。

許多註釋沒有傳遞任何有意義的信息。例如，對於passthrough() 方法，它的註釋是“這個方法允許你傳遞”，但讀者能從中得到什麼幫助呢？這種註釋只會分散注意力，而且很容易失去時效性 [ 假使方法最後又被改名為sendToHost() ] 。

註釋可用來為讀者指定一條正確的代碼訪問路線圖。為代碼中的每個類或模塊添加一個短小的描述，說明其目的以及是否有任何特別需求。對於類中的每個方法，可能要說明下列信息。



- 目的：為什麼需要這個方法？
- 需求（前置條件）：方法需要什麼樣的輸入，對象必須處於何種狀態，才能讓這個方法工作？
- 承諾（後置條件）：方法成功執行後，對象現在處於什麼狀態，有哪些返回值？
- 異常：可能會發生什麼樣的問題？會拋出什麼樣的異常？

要感謝如RDoc、javadoc和ndoc這樣的工具，使用它們可以很方便地直接從代碼註釋創建有用的、格式優美的文檔。這些工具抽取註釋，並生成樣式漂亮且帶有超鏈接的HTML輸出。

下面是一段C#文檔化代碼的摘錄。通常的註釋用//開頭，要生成文檔的註釋用///開頭（當然這仍然是合法的註釋）。

```
using System;
namespace Bank
{
    /// <summary>
    /// A BankAccount represents a customer' s non-secured
deposit
    /// account in the domain (see Reg 47.5, section 3).
    /// </summary>
    public class BankAccount
    {
        ...
        /// <summary>
        /// Increases balance by the given amount.
        /// Requirements: can only deposit a positive amount.
        /// </summary>
        ///
        /// <param name="depositAmount">The amount to deposit,
already
        /// validated and converted to a Money object
        /// </param>
        ///
        /// <param name="depositSource">Origination of the monies
```

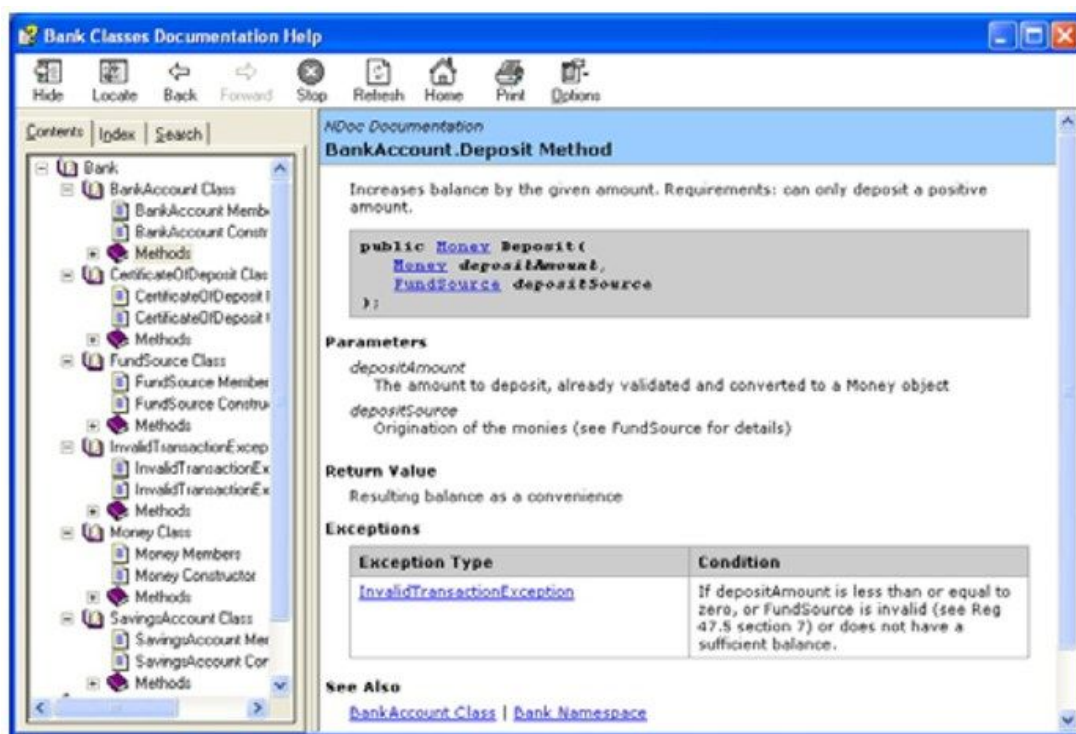
```

    /// (see FundSource for details)
    /// </param>
    ///
    /// <returns>Resulting balance as a convenience
    /// </returns>
    ///
    /// <exception cref="InvalidTransactionException">
    /// If depositAmount is less than or equal to zero, or
FundSource
    /// is invalid (see Reg 47.5 section 7)
    /// or does not have a sufficient balance.
    /// </exception>

    public Money Deposit(Money depositAmount, FundSource
depositSource)
    {
        ...
    }
}

```

圖6-1展示了從C#代碼示例中抽取出來的註釋生成的文檔。用於Java的Javadoc、用於Ruby的Rdoc等工具也都以類似的方式工作。



**圖6-1 使用ndoc 從代碼中抽取出來的文檔**

這種文檔不只是為團隊或組織之外的人準備的。假定你要修改幾個月之前所寫的代碼，如果只要看一下方法頭上的註釋，就知道需要了解的重要細節，那麼修改起來是不是會方便很多？不管怎麼說，如果一個方法只有在發生日全食的時候才能正常工作，那麼先瞭解到這個情況（而不必管代碼細節）是有好處的，否則豈不是要白白等上10年才有這個機會？

代碼被閱讀的次數要遠超過被編寫的次數，所以在編程時多付出一點努力來做好文檔，會讓你在將來受益匪淺。



**用註釋溝通**。使用細心選擇的、有意義的命名。用註釋描述代碼意圖和約束。註釋不能替代優秀的代碼。

**切身感受**

註釋就像是可以帮助你的好朋友，可以先閱讀註釋，然後快速瀏覽代碼，從而完全理解它做了什麼，以及為什麼這樣做。

## 平衡的藝術

- Pascal定理的創始人Blaise Pascal<sup>②</sup>曾說，他總是沒有時間寫短信，所以只好寫長信。請花時間去寫簡明扼要的註釋吧。

② 布萊茲·帕斯卡爾（Blaise Pascal，1623—1662），生於法國奧弗涅，卒於巴黎。他是早慧的神童，早夭的天才。主要的數學成就是射影幾何方面的Pascal定理，他與Fermat是概率論的奠基者。不過對後世影響最大的，是他的宗教性著作《沉思錄》。——譯者注

- 在代碼可以傳遞意圖的地方不要使用註釋。
- 解釋代碼做了什麼的註釋用處不那麼大。相反，註釋要說明為什麼會這樣寫代碼。
- 當重寫方法時，保留描述原有方法意圖和約束的註釋。

## 27 動態評估取捨

“性能、生產力、優雅、成本以及上市時間，在軟件開發過程中都是至關重要的因素。每一項都必須達到最理想狀態。”



你可能曾經身處這樣的團隊：管理層和客戶將很大一部分注意力都放在應用的界面展示上。也有這樣的團隊，其客戶認為性能表現非常重要。在團隊中，你可能會發現，有這樣一個開發主管或者架構師，他會強調遵守“正確”的範式比其他任何事情都重要。對任何單個因素如此獨斷地強調，而不考慮它是否是項目成功的必要因素，必然導致災難的發生。

強調性能的重要性情有可原，因為惡劣的性能表現會讓一個應用在市場上鎩羽而歸。然而，如果應用的性能已經足夠好了，還有必要繼續投入精力讓其運行得更快一點嗎？大概不用了吧。一個應用還有很多其他方面的因素同樣重要。與其花費時間去提升千分之一的性能表現，也許減少開發投入，降低成本，並儘快讓應用程序上市銷售更有價值。

舉例來說，考慮一個必須要與遠程Windows服務器進行通訊的.NET Windows應用程序。可以選擇使用.NET Remoting技術或Web Service來實現這個功能。現在，針對使用Web Service的提議，有些開發者會說：“我們要在Windows之間進行通信，通常此類情況下，推薦使用.NET Remoting。而且，Web Service很慢，我們會遇到性能問題。” 嗯，一般來說確實是這樣。

然而，在這個例子中，使用Web Service很容易開發。對Web Service的性能測試表明XML<sup>①</sup>文檔很小，並且相對應用程序自己的響應時間來講，花在創建和解析XML上的時間幾乎可以忽略不計。使用Web Service不但可以在短期內節省開發時間，且在此後團隊被迫使用第三方提供的服務時，Web Service也是個明智的選擇。

① XML文檔就像人類一樣——它們在小時候很可愛，並且與它們在一起也很有意思，但是長大之後，就會變得特別讓人厭煩。

---

## Andy如是說.....

### 過猶不及

我曾經遇到這樣一個客戶，他們堅信可配置性的重要性，致使他們的應用有大概10 000個可配置變量。新增代碼變得異常艱難，因為要花費大量時間來維護配置應用程序和數據庫。但是他們堅信需要這種程度的靈活性，因為每個客戶都有不同的需求，需要不同的設置。

可實際上，他們只有**19**個客戶，而且預計將來也不會超過**50**個。他們並沒有很好地去權衡。

考慮這樣一個應用，從數據庫中讀取數據，並以表格方式顯示。你可以使用一種優雅的、面向對象的方式，從數據庫中取數據，創建對象，再將它們返回給UI層。在UI層中，你再從對象中拆分出數據，並組織為表格方式顯示。除了看起來優雅之外，這樣做還有什麼好處嗎？

也許你只需要讓數據層返回一個數據集（**dataset**）或數據集合，然後用表格顯示這些數據即可。這樣還可以避免對象創建和銷燬所耗費的資源。如果需要的只是數據展示，為什麼要創建對象去自找麻煩呢？不按書上說的**OO**方式來做，可以減少投入，同時獲得性能上的提升。當然，這種方式有很多缺點，但問題的關鍵是要**多長個心眼兒**，而不是總按照習慣的思路去解決問題。

總而言之，要想讓應用成功，降低開發成本與縮短上市時間，二者的影響同樣重要。由於計算機硬件價格日益便宜，處理速度日益加快，所以可在硬件上多投入以換取性能的提升，並將節省下來的時間放在應用的其他方面。

當然，這也不完全對。如果硬件需求非常龐大，需要一個巨大的計算機網格以及眾多的支持人員才能維持其正常運轉（比如類似**Google**那樣的需求），那麼考慮就要向天平的另一端傾斜了。

但是誰來最終判定性能表現已經足夠好，或是應用的展現已經足夠“炫”了呢？客戶或是利益相關者必須進行評估，並做出相關決定（見第**45**頁習慣**10**）。如果團隊認為性能上還有提升的空間，或者覺得可以讓某些界面看起來更吸引人，那麼就去諮詢一下利益相關者，讓他們決定應將重點放在哪裡。

**沒有最佳解決方案**

## No best solution

沒有適宜所有狀況的最佳解決方案。你必須對手上的問題進行評估，並選出最合適的解決方案。每個設計都是針對特定問題的——只有明確地進行評估和權衡，才能得出更好的解決方案。



**動態評估權衡**。考慮性能、便利性、生產力、成本和上市時間。如果性能表現足夠了，就將注意力放在其他因素上。不要為了感覺上的性能提升或者設計的優雅，而將設計複雜化。

## 切身感受

即使不能面面俱到，你也應該覺得已經得到了最重要的東西——客戶認為有價值的特性。


## 平衡的藝術

- 如果現在投入額外的資源和精力，是為了將來可能得到的好處，要確認投入一定要得到回報（大部分情況下，是不會有回報的）。
- 真正的高性能系統，從一開始設計時就在向這個方向努力。
- 過早的優化是萬惡之源。②

② Donald Knuth對Hoare格言的強有力概括[Knu92]。

- 過去用過的解決方案對當前的問題可能適用，也可能不適用。不要事先預設結論，先看看現在是什麼狀況。

# 28 增量式編程

“真正的程序員寫起代碼來，一干就是幾個小時，根本不停，甚至連頭都不抬。不要停下來去編譯你的代碼，只要一直往下寫就好了！” 

當你開車進行長途旅行時，兩手把住方向盤，固定在一個位置，兩眼直盯前方，油門一踩到底幾個小時，這樣可能嗎？當然不行了，你必須掌控方向，必須經常注意交通狀況，必須檢查油量表，必須停車加油、吃飯，準備其他必需品，以及諸如此類的活動。①

① Kent Beck在《解析極限編程》一書中引入了開車（以及掌控方向盤的重要性）作為比喻。

如果不對自己編寫的代碼進行測試，保證沒有問題，就不要連續幾個小時，甚至連續幾分鐘進行編程。相反，應該採用增量式的編程方式。**增量**式編程可以精煉並結構化你的代碼。代碼被複雜化、變成一團亂麻的幾率減少了。所開發的代碼基於即時的反饋，這些反饋來自以小步幅方式編寫代碼和測試的過程。

採取增量式編程和測試，會傾向於創建更小的方法和更具內聚性的類。你不是在埋頭盲目地一次性編寫一大堆代碼。相反，你會經常評估代碼質量，並不時地進行許多小調整，而不是一次修改許多東西。

在編寫代碼的時候，要經常留心可以改進的微小方面。這可能會改善代碼的可讀性。也許你會發現可以把一個方法拆成幾個更小的方法，使其變得更易於測試。在**重構**的原則指導下，可以做出許多細微改善（見Martin Fowler的《重構：改善既有代碼的設計》②[FBB+99]一書中的相關討論）。可以使用測試優先開發方式（見第82頁習慣20），作為強制進行增量式編程的方式。關鍵在於持續做一些細小而有用的事情，而不是做一段長時間的編程或重構。

② 本書即將由人民郵電出版社出版。——編者注

這就是敏捷的方式。





**在很短的編輯/構建/測試循環中編寫代碼**。這要比花費長時間僅僅做編寫代碼的工作好得多。可以創建更加清晰、簡單、易於維護的代碼。

## 切身感受

在寫了幾行代碼之後，你會迫切地希望進行一次構建/測試循環。在沒有得到反饋時，你不想走得太遠。

## 平衡的藝術

- 如果構建和測試循環花費的時間過長，你就不會希望經常運行它們了。要保證測試可以快速運行。
- 在編譯和測試運行中，停下來想一想，並暫時遠離代碼細節，這是保證不會偏離正確方向的好辦法。
- 要休息的話，就要好好休息。休息時請遠離鍵盤。
- 要像重構你的代碼那樣，重構你的測試，而且要經常重構測試。

# 29 保持簡單

“軟件是很複雜的東西。隨便哪個笨蛋都可以編寫出簡單、優雅的軟件。通過編寫史上最複雜的程序，你將會得到美譽和認可，更不用提保住你的工作了。”



也許你看過這樣一篇文章，其中提到了一個設計想法，表示為一個帶有花哨名稱的模式。放下雜誌，眼前的代碼似乎馬上就可以用到這種模式。這時要捫心自問，是不是真的需要用它，以及它將如何幫你解決眼前的問題。問問自己，是不是特定的問題強迫你使用這個解決方案。不要讓自己被迫進行過分設計，也不要將代碼過分複雜化。

Andy曾經認識一個傢伙，他對設計模式非常著迷，想把它們全都用起來。有一次，要寫一個大概幾百行代碼的程序。在被別人發現之前，他已經成功地將GoF那本書[GHJV95]中的17個模式，都運用到那可憐的程序中。

這不應該是編寫敏捷代碼的方式。

問題在於，許多開發人員傾向於將投入的努力與程序複雜性混同起來。如果你看到別人給出的解決方案，並評價說“非常簡單且易於理解”，很有可能你會讓設計者不高興。許多開發人員以自己程序的複雜性為榮，如果能聽到說：“Wow，這很難，一定是花了很多時間和精力才做出來的吧。”他們就會面帶自豪的微笑了。其實應當恰恰相反，開發人員更應該為自己能夠創建出一個簡單並且可用的設計而驕傲。

## 簡單不是簡陋

### Simple is not simplistic

“簡單性”這個詞彙被人們大大誤解了（在軟件開發工作以及人們的日常生活中，皆是如此）。它並不意味著簡陋、業餘或是能力不足。恰恰相反，相比一個過分複雜、拙劣的解決方案，簡單的方案通常更難以獲得。

簡單性，在編程或是寫作中，就像是廚師的收汁調料。從大量的葡萄酒、主料和配料開始，你小心地進行烹調，到最後得到了最濃縮的精華部分。這就是好的代碼應該帶給人的感覺——不是一大鍋黏糊糊的、亂七八糟的東西，而是真正的、富含營養的、口味上佳的醬汁。

## Andy如是說.....

### 怎樣才算優雅？

優雅的代碼第一眼看上去，就知道它的用處，而且很簡潔。但是這樣的解決方案不是那麼容易想出來的。這就是說，優雅是易於理解

和辨識的，但是要想創建出來就困難得多了。

評價設計質量的最佳方式之一，就是聽從直覺。直覺不是魔術，它是經驗和技能的厚積薄發之產物。在查看一個設計時，聽從頭腦中的聲音。如果覺得什麼地方不對，那就好好想想，是哪裡出了問題。一個好的設計會讓人覺得很舒服。



**開發可以工作的、最簡單的解決方案**。除非有不可辯駁的原因，否則不要使用模式、原則和高難度技術之類的東西。

## 切身感受

當你覺得所編寫的代碼中沒有一行是多餘的，並且仍能交付全部的功能時，這種感覺就對了。這樣的代碼容易理解和改正。

## 平衡的藝術

- 代碼幾乎總是可以得到進一步精煉，但是到了某個點之後，再做改進就不會帶來任何實質性的好處了。這時開發人員就該停下來，去做其他方面的工作了。
- 要將目標牢記在心：簡單、可讀性高的代碼。強行讓代碼變得優雅與過早優化類似，同樣會產生惡劣的影響。
- 當然，簡單的解決方案必須要滿足功能需求。為了簡單而在功能上妥協，這就是過分簡化了。
- 太過簡潔不等於簡單，那樣無法達到溝通的目的。
- 一個人認為簡單的東西，可能對另一個人就意味著複雜。

# 30 編寫內聚的代碼

“你要編寫一些新的代碼，首先要決定的就是把這些代碼放在什麼地方。其實放在什麼地方問題不大，你就趕緊開始吧，看看IDE中現在打開的是哪個類，直接加進去就是了。如果所有的代碼都在一個類或組件裡面，要找起來是很方便的。”



內聚性用來評估一個組件（包、模塊或配件）中成員的功能相關性。內聚程度高，表明各個成員共同完成了一個功能特性或是一組功能特性。內聚程度低的話，表明各個成員提供的功能是互不相干的。

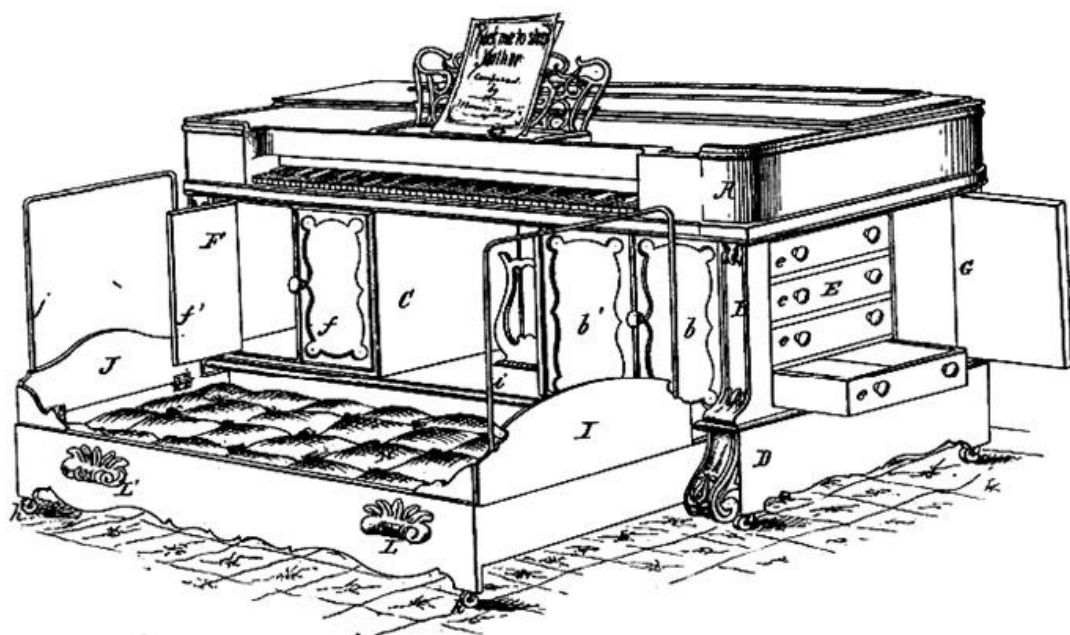
假定把所有的衣服都扔到一個抽屜裡面。當需要找一雙襪子的時候，要翻遍裡面所有的衣服——褲子、內衣、T恤等——才能找到。這很麻煩，特別是在趕時間的時候。現在，假定把所有的襪子都放在一個抽屜裡面（而且是成雙放置的），全部的T恤放在另外一個抽屜中，其他衣服也分門別類。要找到一雙襪子，只要打開正確的抽屜就可以了。

與此類似，如何組織一個組件中的代碼，會對開發人員的生產力和全部代碼的可維護性產生重要影響。在決定創建一個類的時候，問問自己，這個類的功能是不是與組件中其他某個類的功能類似，而且功能緊密相關。這就是組件級的內聚性。

類也要遵循內聚性。如果一個類的方法和屬性共同完成了一個功能（或是一系列緊密相關的功能），這個類就是內聚的。

看看Charles Hess先生於1866年申請的專利，“可變換的鋼琴、睡椅和五斗櫃”（見圖6-2）。根據他的專利說明，它提供了“.....附加的睡椅和五斗櫃.....以填滿鋼琴下未被使用的空間.....”。接下來他說明了為什麼要發明這個可變換的鋼琴。讀者可能已經見過類似這種發明的項目代碼結構了，而且也許其中有你的份。這個發明不具備任何內聚性，任何一個人都可以想象得到，要維護這個怪物（比如換墊子、調鋼琴等）會是多麼困難。

*C. Hess,*  
*Wardrobe, Piano and Bedstead,*  
*Nº 56,413, Patented July 17, 1866.*



**圖6-2 美國專利56 413：可變換的鋼琴、睡椅和五斗櫃**

看看最近的例子。Venkat曾經見過一個用ASP開發的、有20個頁面的Web應用。每個頁面都以HTML開頭，並包含大量VBScript腳本，其中還內嵌了訪問數據庫的SQL語句。客戶當然會認為這個應用的開發已經失去了控制，並且無法維護。如果每個頁面都包括展示邏輯、業務邏輯和訪問數據的代碼，就有太多的東西都堆在一個地方了。

假定要對數據庫的表結構進行一次微調。這個微小的變化會導致應用中所有的頁面發生變化，而且每個頁面中都會有多處改變——這個應用很快就變成了一場災難。

如果應用使用了中間層對象（比如一個COM組件）來訪問數據庫，數據庫表結構變更所造成的影響就可以控制在一定的範圍之內，代碼也更容易維護。

低內聚性的代碼會造成很嚴重的後果。假設有這樣一個類，實現了五種完全不相干的功能。如果這5個功能的需求或細節發生了變化，這個類也必須跟著改變。如果一個類（或者一個組件）變化得過於頻繁，這樣的改變會對整個系統形成“漣漪效應”，並導致更多的維護和成本的發生。考慮另一個只實現了一種功能的類，這個類變化的頻度就沒有那麼高。類似地，一個更具內聚性的組件不會有太多導致其變化的原因，也因此而更加穩定。根據單一職責原則（查看《敏捷軟件開發：原則、模式與實踐》[Mar02]），一個模塊應該只有一個發生變化的原因。

一些設計技巧可以起到幫助作用。舉例來說，我們常常使用模型—視圖—控制器（MVC）模式來分離展示層邏輯、控制器和模型。這個模式非常有效，因為它可以讓開發人員獲得更高的內聚性。模型中的類包含一種功能，在控制器中的類包含另外的功能，而視圖中的類則只關心UI。

內聚性會影響一個組件的可重用性。組件粒度是在設計時要考慮的一個重要因素。根據重用發佈等價原則（[Mar02]）：重用的粒度與發佈的粒度相同。這就是說，程序庫用戶所需要的，是完整的程序庫，而不只是其中的一部分。如果不能遵循這個原則，組件用戶就會被強迫只能使用所發佈組件的一部分。很不幸的是，他們仍然會被不關心的那一部分的更新所影響。軟件包越大，可重用性就越差。



**讓類的功能儘量集中，讓組件儘量小。** 要避免創建很大的類或組件，也不要創建無所不包的大雜燴類。

**切身感受**

感覺類和組件的功能都很集中：每個類或組件只做一件事，而且做得很好。**bug**很容易跟蹤，代碼也易於修改，因為類和組件的責任都很清晰。

## 平衡的藝術


- 有可能會把一些東西拆分成很多微小的部分，而使其失去了實用價值。當你需要一隻襪子的時候，一盒棉線不能帶給你任何幫助。①

① 你可以把這個叫作“意大利麵OO”系統。

- 具有良好內聚性的代碼，可能會根據需求的變化，而成比例地進行變更。考慮一下，實現一個簡單的功能變化需要變更多少代碼。②

② 本書的一位檢閱者告訴我們這樣一個系統，向一個表單中添加一個字段，需要**16**名團隊成員和**6**名經理的同意。這是一個很清晰的警告信號，說明系統的內聚性很差。

## 31 告知，不要詢問

“不要相信其他的對象。畢竟，它們是由別人寫的，甚至有可能是你自己上個月頭腦發昏的時候寫的呢。從別人那裡去拿你需要的信息，然後自己處理，自己決策。不要放棄控制別人的機會！” 

“面向過程的代碼取得信息，然後做出決策。面向對象的代碼讓別的對象去做事情。”**Alec Sharp**[Sha97]通過觀察後，一針見血地指出了這個關鍵點。但是這種說法並不僅限於面向對象的開發，任何敏捷的代碼都應該遵循這個方式。

作為某段代碼的調用者，開發人員絕對不應該基於被調用對象的狀態來做出任何決策，更不能去改變該對象的狀態。這樣的邏輯應該是被

調用對象的責任，而不是你的。在該對象之外替它做決策，就違反了它的封裝原則，而且為bug提供了滋生的土壤。

David Bock使用“送報男孩和錢包的故事”很好地詮釋了這一點。①假定送報男孩來到你的門前，要求付給他本週的報酬。你轉過身去，讓送報男孩從你的後屁股兜裡掏出錢包，並且從中拿走兩美元（你希望是這麼多），再把錢包放回去。然後，送報男孩就會開著他嶄新的美洲豹汽車揚長而去了。

① <http://www.javaguy.org/papers/demeter.pdf> 。

## 將命令與查詢分離開來

### Keep commands separate from queries

在這個過程中，送報男孩作為“調用者”，應該告訴客戶付他兩美元。他不能探詢客戶的財務狀況，或是錢包的薄厚，他也不能代替客戶做任何決策。這都是客戶的責任，而不屬於送報男孩。敏捷代碼也應該以同樣的方式工作。

與告知，不要詢問 相關的一個很有用的技術是：**命令與查詢相分離模式（command-query separation）**。就是要將功能和方法分為“命令”和“查詢”兩類，並在源碼中記錄下來（這樣做可以幫助將所有的“命令”代碼放在一起，並將所有的“查詢”代碼放在一起）。

一個常規的“命令”可能會改變對象的狀態，而且有可能返回一些有用的值，以方便使用。一個“查詢”僅僅提供給開發人員對象的狀態，並不會對其外部的可見狀態進行修改。

## 小心副作用

是不是聽到有人說過：“噢，我們剛調用了那個方法，是因為它的副作用。”這種說法等同於為代碼中的詭異之處辯護說：“嗯，它現在



是這個樣子，是因為過去就是這個樣子.....”

類似這樣的話就是一個明顯的警告信號，表明存在一個敏感易碎的而不是敏捷的設計。

對副作用的依賴，或是與一個不斷扭曲、與現實不符的設計共存，說明你必須馬上開始重新設計以及重構你的代碼了。

這就是說，從外部看來，“查詢”不應該有任何副作用（如果需要的话，開發人員可能想在後臺做一些事先的計算或是緩存處理，但是取得對象中 $X$ 的值，不應該改變 $Y$ 的值）。

像“命令”這種會產生內部影響的方法，強化了**告知，不要詢問**的建議。此外，保證“查詢”沒有副作用，也是很好的編碼實踐，因為開發人員可以在單元測試中自由使用它們，在斷言或者調試器中調用它們，而不會改變應用的狀態。

從外部將“查詢”與“命令”隔離開來，還會給開發人員機會詢問自己為什麼要暴露某些特定的數據。真的需要這麼做嗎？調用者會如何使用它？也許應該有一個相關的“命令”來替代它。



**告知，不要詢問**。不要搶別的對象或是組件的工作。告訴它做什麼，然後盯著你自己的職責就好了。

## 切身感受


Smalltalk使用“信息傳遞”的概念，而不是方法調用。**告知，不要詢問**感覺起來就像你在發送消息，而不是調用函數。

## 平衡的藝術

- 一個對象，如果只是用作大量數據容器，這樣的做法很可疑。有些情況下會需要這樣的東西，但並不像想象的那麼頻繁。

- 一個“命令”返回數據以方便使用是沒有問題的（如果需要的話，創建單獨讀取數據的方法也是可以的）。
- 絕對不能允許一個看起來無辜的“查詢”去修改對象的狀態。

## 32 根據契約進行替換

“深層次的繼承是很棒的。如果你需要其他類的函數，直接繼承它們就好了！不要擔心你創建的新類會造成破壞，你的調用者可以改變他們的代碼。這是他們的問題，而不是你的問題。” 

保持系統靈活性的關鍵方式，是當新代碼取代原有代碼之後，其他已有的代碼不會意識到任何差別。例如，某個開發人員可能想為通信的底層架構添加一種新的加密方式，或者使用同樣的接口實現更好的搜索算法。只要接口保持不變，開發人員就可以隨意修改實現代碼，而不影響其他任何現有代碼。然而，說起來容易，做起來難。所以需要一點指導來幫助我們正確地實現。因此，去看看Barbara Liskov的說法。

Liskov替換原則[Lis88]告訴我們：任何繼承後得到的派生類對象，必須可以替換任何被使用的基類對象，而且使用者不必知道任何差異。換句話說，某段代碼如果使用了基類中的方法，就必須能夠使用派生類的對象，並且自己不必進行任何修改。

這到底意味著什麼？假定某個類中有一個簡單的方法，用來對一個字符串列表進行排序，然後返回一個新的列表。並用如下的方式進行調用：

```
utils = new BasicUtils();  
...  
sortedList = utils.sort(aList);
```

現在假定開發人員派生了一個`BasicUtils` 的子類，並寫了一個新的 `sort()` 方法，使用了更快、更好的排序算法：

```
utils = new FasterUtils();  
...  
sortedList = utils.sort(aList);
```

注意對`sort()` 的調用是完全一樣的，一個`FasterUtils` 對象完美地替換了一個`BasicUtils` 對象。調用`utils.sort()` 的代碼可以處理任何類型的`utils` 對象，而且可以正常工作。

但如果開發人員派生了一個`BasicUtils` 的子類，並改變了排序的意義——也許返回的列表以相反的順序進行排列——那就嚴重違反了 **Liskov**替換原則。

要遵守**Liskov**替換原則，相對基類的對應方法，派生類服務（方法）應該**不要求更多，不承諾更少**；要可以進行自由的替換。在設計類的繼承層次時，這是一個非常重要的考慮因素。

繼承是OO建模和編程中被濫用最多的概念之一。如果違反了**Liskov**替換原則，繼承層次可能仍然可以提供代碼的可重用性，但是將會失去可擴展性。類繼承關係的使用者現在必須要檢查給定對象的類型，以確定如何針對其進行處理。當引入了新的類之後，調用代碼必須經常重新評估並修正。這不是敏捷的方式。

但是可以借用一些幫助。編譯器可以幫助開發人員強制執行**Liskov**替換原則，至少在某種程度上是可以達到的。例如，針對方法的訪問修飾符。在**Java**中，重寫方法的訪問修飾符必須與被重寫方法的修飾符

相同，或者可訪問範圍更加寬大。也就是說，如果基類方法是受保護的，那麼派生重寫方法的修飾符必須是受保護的或者公共的。在C#和VB.NET中，被重寫方法與重寫方法的訪問保護範圍必須完全相同。

考慮一個帶有`findLargest()` 方法的類Base，方法中拋出一個`IndexOutOfRangeException` 異常。基於文檔，類的使用者會準備抓住可能被拋出的異常。現在，假定你從Base 類繼承得到類Derived，並重寫了`findLargest()` 方法，在新的方法中拋出了一個不同的異常。現在，如果某段代碼期待使用Base 類對象，並調用了Derived 類的實例，這段代碼就有可能接收到一個意想不到的異常。你的Derived 類就不能替換使用到Base 類的地方。在Java中，通過不允許重寫方法拋出任何新的檢查異常避免了這個問題，除非異常本身派生自被重寫方法拋出的異常類（當然，對於像`RuntimeException` 這樣的未檢查異常，編譯器就不能幫你了）。

不幸的是，Java也違背了Liskov替換原則。`java.util.Stack` 類派生自`java.util.Vector` 類。如果開發人員（不小心）將Stack 對象發送給一個期待Vector 實例的方法，Stack 中的元素就可能被以與期望的行為不符的順序被插入或刪除。

**針對is-a關係使用繼承；針對has-a或uses-a關係使用委託**

**Use inheritance for is-a;use delegation for has-a or uses-a**

當使用繼承時，要想派生類是否可以替換基類。如果答案是不能，就要問問自己為什麼要使用繼承。如果答案是希望在編寫新類的時候，還要重用基類中的代碼，也許要考慮轉而使用聚合。聚合 是指在類中包含一個對象，並且該對象是其他類的實例，開發人員將責任委託給所包含的對象來完成（該技術同樣被稱為委託）。

圖6-3中展示了委託與繼承之間的差異。在圖中，一個調用者調用了Called Class 中的`methodA()`，而它將會通過繼承直接調用Base

Class 中的方法。在委託的模型中，Called Class 必須要顯式地將方法調用轉向包含的委託方法。

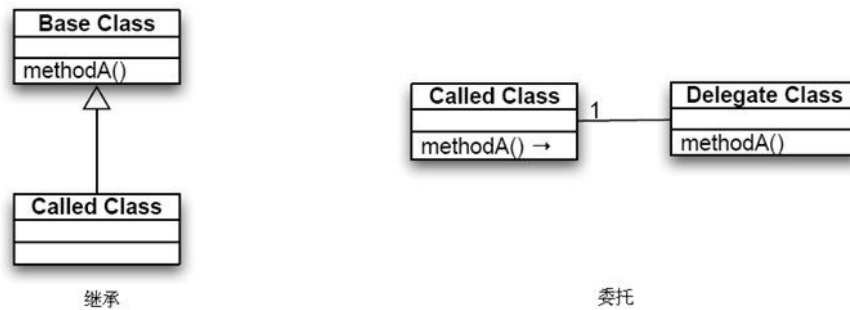


圖6-3 委託與繼承

那麼繼承和委託分別在什麼時候使用呢？

- 如果新類可以替換已有的類，並且它們之間的關係可以通過is-a來描述，就要使用繼承。
- 如果新類只是使用已有的類，並且兩者之間的關係可以描述為has-a或是uses-a，就使用委託吧。

開發人員可能會爭辯說，在使用委託時，必須要寫很多小方法，來將方法調用指向所包含的對象。在繼承中，不需要這樣做，因為基類中的公共方法在派生類中就已經是可用的了。僅憑這一點，並不能構成使用繼承足夠好的理由。

你可以開發一個好的腳本或是好的IDE宏，來幫助編寫這幾行代碼，或者使用一種更好的編程語言/環境，以支持更自動化形式的委託（比如Ruby這一點就做得不錯）。



**通過替換代碼來擴展系統**。通過替換遵循接口契約的類，來添加並改進功能特性。要多使用委託而不是繼承。

## 切身感受

這會讓人覺得有點鬼鬼祟祟的，你可以偷偷地替換組件代碼到代碼庫中，而且其他代碼對此一無所知，它們還獲得了新的或改進後的功能。

## 平衡的藝術

- 相對繼承來說，委託更加靈活，適應力也更強。
- 繼承不是魔鬼，只是長久以來被大家誤解了。
- 如果你不確定一個接口做出了什麼樣的承諾，或是有什麼樣的需求，那就很難提供一個對其有意義的實現了。

# 第7章 敏捷調試

你也許會對木匠那毫無差錯的工作印象深刻。但我向你保證，事實不是這樣的。真正的高手只是知道如何亡羊補牢。

——**Jeff Miller**，傢俱製造者、作家

即使是運作得最好的敏捷項目，也會發生錯誤。bug、錯誤、缺陷——不管被稱作什麼，它們總會發生。

在調試時面對的真正問題，是無法用固定的時間來限制。可以規定設計會議的持續時間，並在時間截止時決定採用最佳的方案。但是調試所耗費的時間，可能是一個小時、一天，甚至一週過去了，還是沒有辦法找到並解決問題。

對於一個項目來說，這種沒有準確把握的時間消耗是不可接受的。不過，我們可以使用一些輔助技術，涵蓋的範圍包括：保留以前的問題解決方案，以及提供發生問題時的更多有用細節。

要想更加有效地重用你的知識和努力，**記錄問題解決日誌** 是很有用的，我們會在下一頁看到如何具體操作。當編譯器警告有問題的時候，要假定**警告就是錯誤**，並且馬上把它們解決掉（第132頁）。

想在一個完整的系統中跟蹤問題非常困難——甚至是不可能的。如果可以**對問題各個擊破**，正如我們在第136頁中看到的那樣，就更容易找到問題了。不同於某些欲蓋彌彰的行為，應該**報告所有的異常**，如第139頁所述。最後，在報告某些事情出錯之時，必須要考慮用戶的感受，並且**提供有用的錯誤信息**。我們會在第141頁看到這是為什麼。

## 33 記錄問題解決日誌

“在開發過程中是不是經常遇到似曾相識的問題？這沒關係。以前解決過的問題，現在還是可以解決掉的。”



面對問題（並解決它們）是開發人員的一種生活方式。當問題發生時，我們希望趕緊把它解決掉。如果一個熟悉的問題再次發生，我們會希望記起第一次是如何解決的，而且希望下次能夠更快地把它搞定。然而，有時一個問題看起來跟以前遇到的完全一樣，但是我們卻不記得是如何修復的了。這種狀況時常發生。

不能通過**Web**搜索獲得答案嗎？畢竟互聯網已經成長為如此令人難以置信的信息來源，我們也應該好好加以利用。從**Web**上尋找答案當然勝過僅靠個人努力解決問題。可這是非常耗費時間的過程。有時可以找到需要的答案，有時除了找到一大堆意見和建議之外，發現不了實質性的解決方案。看到有多少開發人員遇到同樣的問題，也許會感覺不錯，但我們需要的是**一個解決辦法**。

## 不要在同一處跌倒兩次

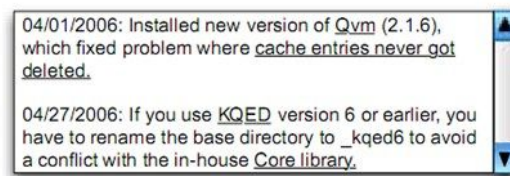
### Don't get burned twice

要想得到更好的效果，不妨維護一個保存曾遇到的問題以及對應解決方案的日誌。這樣，當問題發生時，就不必說：“嘿，我曾碰到過這個問題，但是不記得是怎麼解決的了。”可以快速搜索以前用過的方法。工程師們已經使用這種方式很多年了，他們稱之為**每日日誌**（daylog）。

可以選擇符合需求的任何格式。下面這些條目可能會用得上。

- 問題發生日期。
- 問題簡述。
- 解決方案詳細描述。
- 引用文章或網址，以提供更多細節或相關信息。
- 任何代碼片段、設置或對話框的截屏，只要它們是解決方案的一部分，或者可以幫助更深入地理解相關細節。

要將日誌保存為可供計算機搜索的格式，就可以進行關鍵字搜索以快速查找細節。圖7-1展示了一個簡單的例子，其中帶有超鏈接以提供更多信息。



**圖7-1** 帶有超鏈接的解決方案條目示例



如果面臨的問題無法在日誌中找到解決方案，在問題解決之後，要記得馬上將新的細節記錄到日誌中去。

要共享日誌給其他人，而不僅僅是靠一個人維護。把它放到共享的網絡驅動器中，這樣其他人也可以使用。或者創建一個Wiki，並鼓勵其他開發人員使用和更新其內容。



**維護一個問題及其解決方案的日誌**。保留解決方案是修復問題過程的一部分，以後發生相同或類似問題時，就可以很快找到並使用了。

## 切身感受

解決方案日誌應該作為思考的一個來源，可以在其中發現某些特定問題的細節。對於某些類似但是有差異的問題，也能從中獲得修復的指引。

## 平衡的藝術

- 記錄問題的時間不能超過在解決問題上花費的時間。要保持輕量級和簡單，不必達到對外發布式的質量。
- 找到以前的解決方法非常關鍵。使用足夠的關鍵字，可以幫助你在需要的時候發現需要的條目。
- 如果通過搜索Web，發現沒人曾經遇到同樣的問題，也許搜索的方式有問題。
- 要記錄發生問題時應用程序、應用框架或平臺的特定版本。同樣的問題在不同的平臺或版本上可能表現得不同。
- 要記錄團隊做出一個重要決策的原因。否則，在6~9個月之後，想再重新回顧決策過程的時候，這些細節就很難再記得了，很容

易發生互相指責的情形。

## 34 警告就是錯誤

“編譯器的警告信息只不過是給過分小心和過於書呆子氣的人看的。它們只是警告而已。如果導致的後果很嚴重，它們就是錯誤了，而且會導致無法通過編譯。所以乾脆忽略它們就是了。”



當程序中出現一個編譯錯誤時，編譯器或是構建工具會拒絕產生可執行文件。我們別無選擇——必須要先修正錯誤，再繼續前行。

然而，警告卻是另外一種狀況。即使代碼編譯時產生了警告，我們還是可以運程序。那麼忽略警告信息繼續開發代碼，會導致什麼狀況呢？這樣做等於是坐在了一個嘀嗒作響的定時炸彈上，而且它很有可能在最糟糕的時刻爆炸。

有些警告是過於挑剔的編譯器的良性副產品，有些則不是。例如，一個關於未被使用的變量的警告，可能不會產生什麼惡劣影響，但卻有可能是暗示某些變量被錯誤使用了。

最近在一家客戶那裡，Venkat發現一個開發中的應用有多於300個警告。其中一個被開發人員忽略的警告是這樣：

```
Assignment in conditional expression is always constant;  
did you mean to use == instead of = ?  
條件表達式中的賦值總為常量，你是否要使用==而不是=？
```

相關代碼如下：

```
if (theTextBox.Visible = true)
```

...

也就是說，`if` 語句總是會評估為`true`，無論不幸的`theTextBox` 變量是什麼狀況。看到類似這樣真正的錯誤被當作警告忽略掉，真是令人感到害怕。

看看下面的C#代碼：

```
public class Base
{
    public virtual void foo()
    {
        Console.WriteLine("Base.foo");
    }
}

public class Derived : Base
{
    public virtual void foo()
    {
        Console.WriteLine("Derived.foo");
    }
}

class Test
{
    static void Main(string[] args)
    {
        Derived d = new Derived();
        Base b = d;
        d.foo();
        b.foo();
    }
}
```

在使用Visual Studio 2003默認的項目設置對其進行編譯時，會看到如此信息“構建：1個成功，0失敗，0跳過”顯示在Output窗口的底部。運程序，會得到這樣的輸出：

```
Derived.foo  
Base.foo
```

但這不是我們預期的結果。應該看到兩次對Derived 類中foo 方法的調用。是哪裡出錯了？如果仔細查看Output窗口，可以發現這樣的警告信息：

```
Warning. Derived.foo hides inherited member Base.foo  
To make the current member override that implementation,  
add the override keyword. Otherwise, you' d add the new keyword.
```

這明顯是一個錯誤——在Derived 類的foo() 方法中，應該使用 **override** 而不是**virtual**。①想象一下，有組織地忽略代碼中類似這樣的錯誤會導致什麼樣的後果。代碼的行為會變得無法預測，其質量會直線下降。

① 這對C++程序員來講是一個潛伏的陷阱。在C++中代碼可以按預期方式工作。

可能有人會說優秀的單元測試可以發現這些問題。是的，它們可以起到幫助作用（而且也應該使用優秀的單元測試）。可如果編譯器可以發現這種問題，那為什麼不利用它呢？這可以節省大量的時間和麻煩。

要找到一種方式讓編譯器將警告作為錯誤提示出來。如果編譯器允許調整警告的報告級別，那就把級別調到最高，讓任何警告不能被忽略。例如，GCC編譯器支持 `-Werror` 參數，在Visual Studio中，開發人員可以改變項目設置，將警告視為錯誤。

對於一個項目的警告信息來說，至少也要做到這種地步。然而，如果採取這種方式，就要對創建的每個項目去進行設置。如果可以儘量以全局化的方式來進行設置就好了。

比如，在Visual Studio中，開發人員可以修改項目模板（查看 *.NET Gotchas* [Sub05]獲取更多細節），這樣在計算機上創建的任何項目，都會有同樣的完整項目設置。在當前版本的Eclipse中，可以按照這樣的順序修改設置：Windows→Preferences→Java→Compiler→Errors/Warnings。如果使用其他的語言或IDE，花一些時間來找出如何在其中將警告作為錯誤處理吧。

在修改設置的時候，要記得在構建服務器上使用的持續集成工具中，修改同樣的設置選項。（要詳細瞭解持續集成，查看第87頁習慣21。）這個小小的設置，可以大大提升團隊簽入到源碼控制系統中的代碼質量。

在開始一個項目的時候，要把相關的設置都準備好。在項目進行到一半的時候，突然改變警告設置，有可能會帶來顛覆性的後果，導致難以控制。

編譯器可以輕易處理警告信息，可是你不能。



**將警告視為錯誤**。簽入帶有警告的代碼，就跟簽入有錯誤或者沒有通過測試的代碼一樣，都是極差的做法。簽入構建工具中的代碼不應該產生任何警告信息。

**切身感受**

警告給人的感覺就像.....哦，警告。它們就某些問題給出警告，來吸引開發人員的注意。

## 平衡的藝術

- 雖然這裡探討的主要是編譯語言，解釋型語言通常也有標誌，允許運行時警告。使用相關標誌，然後捕獲輸出，以識別並最終消除警告。
- 由於編譯器的bug或是第三方工具或代碼的原因，有些警告無法消除。如果確實沒有應對之策的話，就不要再浪費更多時間了。但是類似的狀況很少發生。
- 應該經常指示編譯器：要特別注意別將無法避免的警告作為錯誤進行提示，這樣就不用費力去查看所有的提示，以找到真正的錯誤和警告。
- 棄用的方法被棄用是有原因的。不要再使用它們了。至少，安排一個迭代來將它們（以及它們引起的警告信息）安全地移除掉。
- 如果將過去開發完成的方法標記為棄用方法，要記錄當前用戶應該採取何種變通之策，以及被棄用的方法將會在何時一起移除。

## 35 對問題各個擊破

“逐行檢查代碼庫中的代碼確實很令人恐懼。但是要調試一個明顯的錯誤，只有去查看整個系統的代碼，而且要全部過一遍。畢竟你不知道問題可能發生在什麼地方，這樣做是找到它的唯一方式。”



單元測試（在第76頁，第5章）帶來的積極效應之一，是它會強迫形成代碼的分層。要保證代碼可測試，就必須把它從周邊代碼中解脫出來。如果代碼依賴其他模塊，就應該使用mock對象，來把它從其他模

塊中分離開。這樣做不但讓代碼更加健壯，且在發生問題時，也更容易定位來源。

否則，發生問題時有可能無從下手。也許可以先使用調試器，逐行執行代碼，並試圖隔離問題。也許在進入到感興趣的部分之前，要運行多個表單或對話框，這會導致更難發現問題的根源。你會發現自己陷入整個系統之中，徒然增加了壓力，而且降低了工作效率。

大型系統非常複雜——在執行過程中會有很多因素起作用。從整個系統的角度來解決問題，就很難區分開，哪些細節對要定位的特定問題產生影響，而哪些細節沒有。

答案很清晰：不要試圖馬上了解系統的所有細節。要想認真調試，就必須將有問題的組件或模塊與其他代碼庫分離開來。如果有單元測試，這個目的就已經達到了。否則，你就得開動腦筋了。

比如，在一個時間緊急的項目中（哪個項目的時間不緊急呢），Fred和George發現他們面對的是一個嚴重的數據損毀問題。要花很多精力才能知道哪裡出了問題，因為開發團隊沒有將數據庫相關的代碼與其他的應用代碼分離開。他們無法將問題報告給軟件廠商，當然不能把整個代碼庫用電子郵件發給人家！

於是，他們倆開發了一個小型的原型系統，並展示了類似的症狀；然後將其發送給廠商作為實例，並詢問他們的專家意見，使用原型幫助他們對問題理解得更清晰。

而且，如果他們無法在原型中再現問題的話，原型也能告訴他們可以工作的代碼示例，這也有助於分離和發現問題。

## 用原型進行分離

### **Prototype to isolate**

識別複雜問題的第一步，是將它們分離出來。既然不可能在半空中試圖修復飛機引擎，為什麼還要試圖在整個應用中，診斷其中某個組成部分的複雜問題呢？當引擎被從飛機中取出來，而且放在工作臺上之後，就更容易修復了。同理，如果可以隔離出發生問題的模塊，也更容易修復發生問題的代碼。

可是，很多應用的代碼在編寫時沒有注意到這一點，使得分離變得特別困難。應用的各個構成部分之間會彼此糾結：想把這個部分單獨拿出來，其他的會緊隨而至。①在這些狀況下，最好花一些時間把關注的代碼提取出來，而且創建一個可讓其工作的測試環境。

① 這被親切地稱為“大泥球”（Big Ball of Mud）設計反模式。

對問題各個擊破，這樣做有很多好處：通過將問題與應用其他部分隔離開，可以將關注點直接放在與問題相關的議題上；可以通過多種改變，來接近問題發生的核心——你不可能針對正在運行的系統來這樣做。可以更快地發現問題的根源所在，因為只與所需最小數量的相關代碼發生關係。

隔離問題不應該只在交付軟件之後才著手。在構建系統原型、調試和測試時，各個擊破的戰略都可以起到幫助作用。



**對問題各個擊破**。在解決問題時，要將問題域與其周邊隔離開，特別是在大型應用中。

## 切身感受

面對必須要隔離的問題時，感覺就像在一個茶杯中尋找一根針，而不是大海撈針。

## 平衡的藝術



- 如果將代碼從其運行環境中分離後，問題消失不見了，這有助於隔離問題。
- 另一方面，如果將代碼從其運行環境中分離後，問題還在，這也有助於隔離問題。
- 以**二分查找**的方式來定位問題是很有用的。也就是說，將問題空間分為兩半，看看哪一半包含問題。再將包含問題的一半進行二分，並不斷重複這個過程。
- 在向問題發起攻擊之前，先查找你的問題解決日誌（見第129頁習慣33）。

## 36 報告所有的異常

“不要讓程序的調用者看到那些奇怪的異常。處理它們是你的責任。把你調用的一切都包起來，然後發送自己定義的異常——或者乾脆自己解決掉。”



從事任何編程工作，都要考慮事物正常狀況下是如何運作的。不過更應該想一想，當出現問題——也就是事情沒有按計劃進行時，會發生什麼。

在調用別人的代碼時，它也許會拋異常，這時我們可以試著對其處理，並從失敗中恢復。當然，要是在用戶沒有意識到的情況下，可以恢復並繼續正常處理流程，這就最好不過了。要是不能恢復，應該讓調用代碼的用戶知道，到底是哪裡出現了問題。

不過也不盡然。**Venkat**曾經在使用一個非常流行的開源程序庫（這裡就不提它的名字了）時倍受打擊。他調用的一個方法本來應該創建一個對象，可是得到的卻是`null`引用。涉及的代碼量非常少，而且沒有其他代碼發生聯繫，也很簡單。所以從他自己寫的這塊代碼的角度來看，不太可能出問題，他摸不到一點頭緒。

幸好這個庫是開源的，所以他下載了源代碼，然後帶著問題檢查了相關的方法。這個方法調用了另外的方法，那個方法認為他的系統中缺少了某些必要的組件。這個底層方法拋出了帶有相關信息的異常。但是，上層方法卻偷偷地用沒有異常處理代碼的空`catch`代碼塊，把異常給忽略掉了，然後就拋出一個`null`。Venkat所寫的代碼根本不知道到底發生了什麼，只有通過閱讀程序庫的代碼，他才能明白這個問題，並最後安裝了缺失的組件。

像Java中那樣的檢查異常會強迫你捕捉異常，或是把異常傳播出去。可是有些開發人員會採取臨時的做法：捕捉到異常後，為了不看到編譯器的提示，就把異常忽略掉。這樣做很危險——臨時的補救方式很容易被遺忘，並且會進入到生產系統的代碼中。必須要處理所有的異常，倘若可以，從失敗中恢復再好不過。如果不能處理，就要把異常傳播到方法的調用者，這樣調用者就可以嘗試對其進行處理了（或者以優雅的方式將問題的信息告訴給用戶，見習慣37）。

聽起來很明白，是吧？其實不像想象得那麼容易。不久前有一條新聞，提到一套大型航空訂票系統中發生了嚴重的問題。系統崩潰，飛機停飛，上千名旅客滯留機場，整個航空運輸系統數天之內都亂作一團。原因是什麼？在一臺應用服務器上發生了一個未檢查異常。

也許你很享受CNN新聞上提到你名字的感覺，但是你不太可能希望發生這樣的情形。



**處理或是向上傳播所有的異常**。不要將它們壓制不管，就算是臨時這樣做也不行。在寫代碼時要估計到會發生的問題。

## 切身感受

當出現問題時，心裡知道能夠得到拋出的異常。而且沒有空的異常處理方法。

## 平衡的藝術

- 決定由誰來負責處理異常是設計工作的一部分。
- 不是所有的問題都應該拋出異常。
- 報告的異常應該在代碼的上下文中有實際意義。在前述的例子中，拋出一個`NullPointerException` 看起來也許不錯，不過這就像拋出一個`null` 對象一樣，起不到任何幫助作用。
- 如果代碼中會記錄運行時調試日誌，當捕獲或是拋出異常時，都要記錄日誌信息；這樣做對以後的跟蹤工作很有幫助。
- 檢查異常處理起來很麻煩。沒人願意調用拋出31種不同檢查異常的方法。這是設計上的問題：要把它解決掉，而不是隨便打個補丁就算了。
- 要傳播不能處理的異常。

## 37 提供有用的錯誤信息

“不要嚇著用戶，嚇程序員也不行。要提供給他們乾淨整潔的錯誤信息。要使用如下這樣讓人舒服的詞句：‘用戶錯誤。替換，然後繼續’。”



當應用發佈並且在真實世界中得到使用之後，仍然會發生這樣那樣的問題。比如計算模塊可能出錯，與數據庫服務器之間的連接也可能丟失。當無法滿足用戶需求時，要以優雅的方式進行處理。

類似的錯誤發生時，是不是隻要彈出一條優雅且帶有歉意的信息給用戶就足夠了？並不盡然。當然了，顯示通用的信息，告訴用戶發生了問題，要好過由於系統崩潰造成應用執行錯誤的動作，或者直接關閉（用戶會因此感到困惑，並希望知道問題所在）。然而，類似“出錯

了”這樣的消息，無法幫助團隊針對問題做出診斷。用戶在給支持團隊打電話報告問題時，我們希望他們提供足夠多且好的信息，以幫助儘快識別問題所在。遺憾的是，用很通用的錯誤消息，是無法提供足夠的數據的。

針對這個問題，常用的解決方案是記錄日誌：當發生問題時，讓應用詳細記錄錯誤的相關數據。錯誤日誌最起碼應該以文本文件的形式維護。不過也許可以發佈到一個系統級別的事件日誌中。可以使用工具來瀏覽日誌，產生所有日誌信息的RSS Feed，以及諸如此類的輔助方式。

記錄日誌很有用，可是單單這樣做是不夠的：開發人員認真分析日誌，可以得到需要的數據；但對於不幸的用戶來說，起不到任何幫助作用。如果展示給他們類似下圖中的信息，他們還是一點頭緒都沒有——不知道自己到底做錯了什麼，應該怎麼做可以繞過這個錯誤，或者在給技術支持打電話時，應該報告什麼。

如果你注意的話，在開發階段就能發現這個問題的早期警告。作為開發人員，經常要將自己假定為用戶來測試新功能。要是錯誤信息很難理解，或者無助於定位錯誤的話，就可以想想真正的用戶和支持團隊，遇到這個問題時會有多麼困難了（見圖7-2）。



**圖7-2** 無用的異常信息

例如，假定登錄UI調用了應用的中間層，後臺向數據訪問層發送了一個請求。由於無法連接數據庫，數據訪問層拋出一個異常。這個異常被中間層用自己的異常包裹起來，並繼續向上傳遞。那麼UI層應該怎麼做呢？它至少應該讓用戶知道發生了系統錯誤，而不是由用戶的輸入引起的。

接下來，用戶會打電話並且告訴我們他無法登錄。我們怎麼知道問題的實質是什麼呢？日誌文件可能有上百個條目，要找到相關的細節非常困難。

實際上，不妨在顯示給用戶的信息中提供更多細節。好比說，可以看到是哪條SQL查詢或存儲過程發生了錯誤；這樣可以很快找到問題並且修正，而不是浪費大把的時間去盲目地碰運氣。不過另一方面，在生產系統中，向用戶顯示數據連接問題的特定信息，不會對他們有多大幫助。而且有可能嚇他們一跳。

一方面要提供給用戶清晰、易於理解的問題描述和解釋，使他們有可能尋求變通之法。另一方面，還要提供具備關於錯誤的詳細技術細節給用戶，這樣方便開發人員尋找代碼中真正的問題所在。

下面是一種同時實現上述兩個目的的方式：圖中顯示了清晰的錯誤說明信息。該錯誤信息不只是簡單的文本，還包括了一個超鏈接。用戶、開發人員、測試人員都可以由此鏈接得到更多信息，如圖7-3、圖7-4所示。

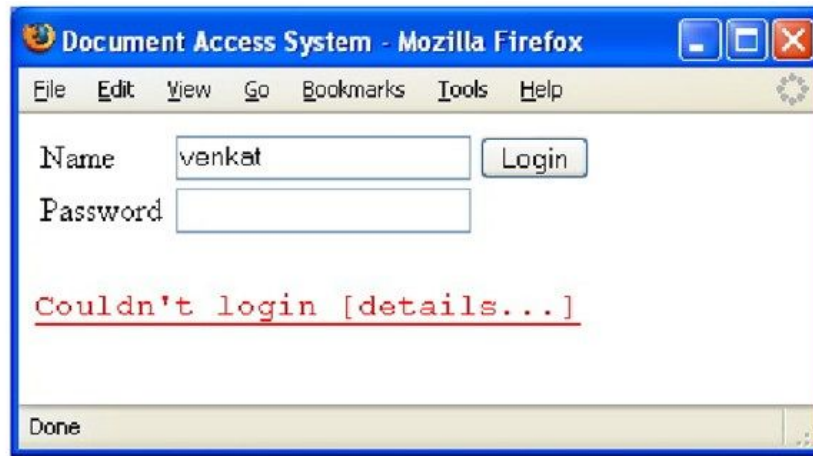


圖7-3 帶有更多細節鏈接的異常信息

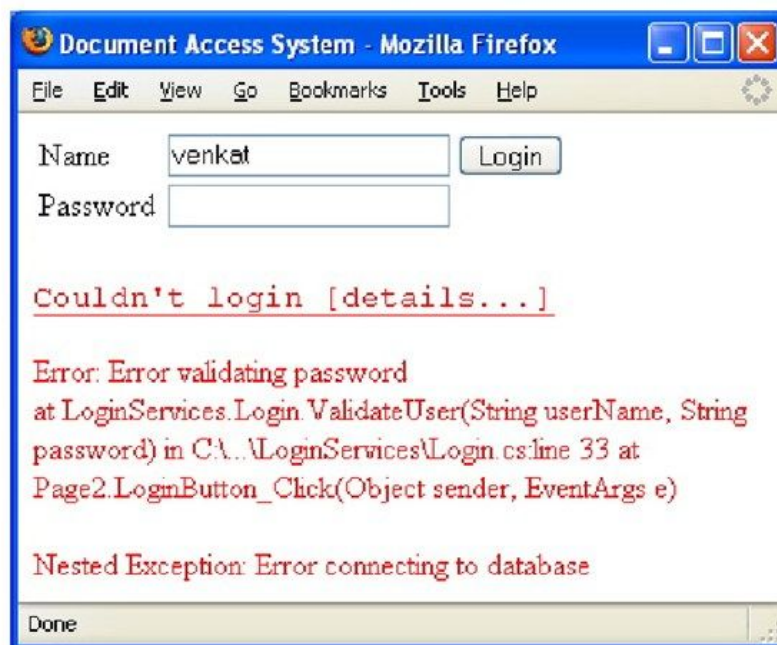


圖7-4 供調試用的完整詳細信息

進入鏈接的頁面，可以看到異常（以及所有嵌套異常）的詳細信息。在開發時，我們可能希望只要看到這些細節就好了。不過，當應用進入生產系統後，就不能把這些底層細節直接暴露給用戶了，而要提供鏈接，或是某些訪問錯誤日誌的入口。支持團隊可以請用戶點擊錯誤

信息，並讀出錯誤日誌入口的相關信息，這樣支持團隊可以很快找到錯誤日誌中的特定細節。對於獨立系統來說，點擊鏈接，有可能會將錯誤信息通過電子郵件發送到支持部門。

除了包括出現問題的詳細數據外，日誌中記錄的信息可能還有當時系統狀態的一個快照（例如Web應用的會話狀態）。①

① 有些安全敏感的信息不應該被暴露，甚至不可以記錄到日誌中去，這其中包括密碼、銀行賬戶等。

使用上述信息，系統支持團隊可以重建發生問題的系統狀態，這樣對查找和修復問題非常有效。

錯誤報告對於開發人員的生產率，以及最終的支持活動消耗成本，都有很大的影響。在開發過程中，如果定位和修復問題讓人倍受挫折，就考慮使用更加積極主動的錯誤報告方式吧。調試信息非常寶貴，而且不易獲得。不要輕易將其丟棄。



**展示有用的錯誤信息**。提供更易於查找錯誤細節的方式。發生問題時，要展示出儘量多的支持細節，不過別讓用戶陷入其中。

---

## 區分錯誤類型

**程序缺陷**。這些是真正的bug，比如NullPointerException、缺少主鍵等。用戶或者系統管理員對此束手無策。

**環境問題**。該類別包括數據庫連接失敗，或是無法連接遠程Web Service、磁盤空間滿、權限不足，以及類似的問題。程序員對此沒有應對之策，但是用戶也許可以找到變通的方法，如果提供足夠詳細的信息，系統管理員應該可以解決這些問題。

**用戶錯誤**。程序員與系統管理員不必擔心這些問題。在告知是哪裡操作的問題後，用戶可以重新來過。

通過追蹤記錄報告的錯誤類型，可以為受眾提供更加合適的建議。

## 切身感受

錯誤信息有助於問題的解決。當問題發生時，可以詳細研究問題的細節描述和發生上下文。

## 平衡的藝術

- 像“無法找到文件”這樣的錯誤信息，就其本身而言無助於問題的解決。“無法打開/andy/project/main.yaml以供讀取”這樣的信息更有效。
- 沒有必要等待拋出異常來發現問題。在代碼關鍵點使用斷言以保證一切正常。當斷言失敗時，要提供與異常報告同樣詳細的信息。
- 在提供更多信息的同時，不要洩露安全信息、個人隱私、商業機密，或其他敏感信息（對於基於Web的應用，這一點尤其重要）。
- 提供給用戶的信息可以包含一個主鍵，以便於在日誌文件或是審核記錄中定位相關內容。

# 第8章 敏捷協作

我不僅發揮了自己的全部能力，還將我所仰仗的人的能力發揮到極致。



## ——伍德羅●威爾遜，美國第28任總統（1856—1924）

只要是具備一定規模的項目，就必然需要一個團隊。靠單打獨鬥在車庫裡面開發出一個完整產品的日子早已不再。然而，在團隊中工作與單兵作戰，二者是完全不同的。一個人會突然發現，自己的行為會對團隊以及整個項目的生產效率和進度產生影響。

項目的成功與否，依賴於團隊中的成員如何一起有效地工作，如何互動，如何管理他們的活動。全體成員的行動必須要與項目相關，反過來每個人的行為又會影響項目的環境。

高效的協作是敏捷開發的基石，下面這些習慣將會幫助所有的團隊成員全心投入到項目中，並且大家一起向著正確的方向努力。

首先要做的是**定期安排會面時間**，見第148頁。面對面的會議仍然是最有效的溝通方式，所以我們將以此作為本章的開篇。接下來，希望每個人都能投入到開發過程中來。也就是說**架構師必須寫代碼**（我們會在第152頁看到為什麼要這樣做）。既然整個團隊都是項目工作的一部分，我們希望**實行代碼集體所有制**（見第155頁），以保證任何團隊成員的缺席不會對項目造成影響。這就是協作的效果，還記得嗎？

但是高效的協作並不只是寫出代碼就好了。隨著時間的流逝，團隊中每個人都要強化和提高他們的技能，並且推進各自的職業發展。即使一個人剛剛加入團隊，他也可以**成為指導者**，將會在第157頁談到應該怎麼做。團隊中一個人的知識，經常可以解決另外一名團隊成員的問題。只要**允許大家自己想辦法**，就可以幫助團隊不斷成長，就像在第160頁上看到的那樣。

最後，由於大家都是團隊中一起工作，每個人就要修改自己的個人編碼習慣，來適應團隊的其他成員。對於初學者來說，**準備好後再共享代碼** 才是有禮貌的做法（見第162頁），這樣才不會用未完成的工作來給團隊成員造成麻煩。當準備好之後，我們應該與其他團隊成員

一起做代碼複查（見第165頁）。隨著項目的推進，我們會不斷地完成舊任務，並且領取新任務。應該**及時通報進展與問題**，讓大家瞭解彼此的進度、遇到的問題，以及在開發過程中發現的有意思的東西。我們將在第168頁討論該習慣並結束本章。

## 38 定期安排會面時間

“會議安排得越多越好。實際上，我們要安排更多的會議，直到發現為什麼工作總是完不成。”



也許你個人很討厭開會，但是溝通是項目成功的關鍵。我們不只要跟客戶談話，還應該與開發人員進行良好的溝通。要知道其他人在做什麼——如果Bernie知道如何解決你的問題，你肯定希望早點搞清楚她是怎麼做的，不是嗎？

**立會**（站著開的會議，Scrum最早引入並被極限編程所強調的一個實踐）是將團隊召集在一起，並讓每個人瞭解當下進展狀況的好辦法。顧名思義，參與者們不允許在立會中就坐，這可以保證會議快速進行。一個人坐下來之後，會由於感到舒適而讓會議持續更長的時間。

Andy曾遇到一個客戶，他和Dave Thomas通過電話遠程參與客戶的站立會議。一切都看起來很順利，直到有一天，會議時間比平時多了一倍。你猜怎麼著？客戶那邊，與會者都挪到了會議室，舒舒服服地坐在扶椅上開會。

坐著開的會議通常會持續更久，大部分人不喜歡站著進行長時間的談話。

要保證會議議題不會發散，每個人都應該只回答下述三個問題。

- 昨天有什麼收穫？
- 今天計劃要做哪些工作？

- 面臨著哪些障礙？

只能給予每個參與者很少的時間發言（大約兩分鐘）。也許要用計時器來幫助某些收不住話頭的人。如果要詳細討論某些問題，可以在立會結束之後，再召集相關人員（在會議中說“我需要跟Fred和Wilma討論一下數據庫”是沒有問題的，但是不要深入討論細節）。

通常，立會都是在每個工作日的早些時候，且大家都在上班時舉行。但是不要把它安排為上班後的第一件事。要讓大家有機會從剛才混亂的交通狀況中恢復狀態，喝點咖啡，刪除一些垃圾郵件什麼的。要保證會議結束後有足夠的時間，讓大家在午餐之前做不少工作，同時也不要開始得過早，讓每個人都巴不得趕緊結束會議，去喝點東西。一般來說，在大家到公司之後的半個小時到一個小時之內舉行，是個不錯的選擇。

## 豬 與 雞

Scrum將團隊成員與非團隊成員這兩種角色命名為豬和雞。團隊成員是豬（自尊何在啊），非團隊成員（管理層、支持人員、QA等）是雞。這兩個用語來自一個寓言，講的是農場裡的動物們打算一起開飯店，並且準備用燻肉和雞蛋作為早餐提供。對於雞來說，當然是要參與進來了，可對於豬來講，可就是放血投入了。

只有“豬”才允許參與Scrum的每日立會。

參加會議的人要遵守一些規則，以保證彼此不會分神，而且會議也不會跑題。這些規則有：只有團隊成員——開發人員、產品所有者和協調者可以發言（查看上面對“豬”和“雞”的描述）。他們必須回答上面的3個問題，而且不能展開深入討論（討論可以安排在會後進行）。管理層可以把要解決的問題記下來，但是不能試圖將會議從每個人要回答的三個問題引開。

每日立會有諸多好處。

- 讓大家儘快投入到一天的工作中來。
- 如果某個開發人員在某一點上有問題，他可以趁此機會將問題公開，並積極尋求幫助。
- 幫助團隊帶頭人或管理層瞭解哪些領域需要更多的幫助，並重新分配人手。
- 讓團隊成員知道項目其他部分的進展情況。
- 幫助團隊識別是否在某些東西上有重複勞動而耗費了精力，或者是不是某個問題有人已有現成的解決方案。
- 通過促進代碼和思路的共享，來提升開發速度。
- 鼓勵向前的動力：看到別人報告的進度都在前進，會對彼此形成激勵。

## 使用廚房計時器

開發者Nancy Davis告訴我們她使用廚房計時器召開立會的經驗。

“我們使用了妹妹去年聖誕節送給我的一個廚房計時器。它在運行時不會發出‘嘀嗒’的聲音，只會在時間到達後發出‘叮’的一聲。如果計時器停止了，我們就再加兩分鐘，並讓下一個成員發言。有時會忘掉計時器的存在，並讓會議持續需要的時間，但是大部分情況下，我們都會遵守計時器的提醒。”

採取立會的形式需要管理層的承諾和參與。不過，團隊中的開發人員可以幫助推行這個實踐。如果開發人員無法說服管理層的參與，他們自己可以用非正式的形式召開立會。



**使用立會**。立會可以讓團隊達成共識。保證會議短小精悍不跑題。

## 切身感受

大家都盼望著立會。希望彼此瞭解各自的進度和手上的工作，而且不怕把各自遇到的問題拿出來公開討論。

## 平衡的藝術

- 會議會佔用開發時間，所以要儘量保證投入的時間有較大的產出。立會的時間最長不能超出**30分鐘**，**10~15分鐘**比較理想。
- 如果要使用需提前預定的會議室，就把預定的時間設定為一個小時吧。這樣就有機會在**15分鐘**的立會結束後，馬上召開更小規模的會議。
- 雖然大多數團隊需要每天都碰頭，但對於小型團隊來說，這樣做可能有點過頭了。不妨兩天舉行一次，或者一週兩次，這對小團隊來說足夠了。
- 要注意報告的細節。在會議中要給出具體的進度，但是不要陷入細節之中。例如，“我在開發登錄頁面”就不夠詳細。“登錄頁面目前接受`guest/guest`作為登錄用戶名和密碼，我明天會連接數據庫來做登錄驗證”，這樣的詳細程度才行。
- 迅速地開始可以保證會議短小。不要浪費時間等著會議開始。
- 如果覺得立會是在浪費時間，那可能是大家還沒有形成真正的團隊意識。這並不是壞事，有利於針對問題進行改進。

## 39 架構師必須寫代碼

“我們的專家級架構師Fred會提供設計好的架構，供你編寫代碼。他經驗豐富，拿的薪水很高，所以不要用一些愚蠢的問題或者實現上的難點來浪費他的時間。”



## 不可能在PowerPoint 幻燈片中進行編程

### You can't code in PowerPoint

軟件開發業界中有許多掛著架構師稱號的人。作為作者的我們不喜歡這個稱號，為什麼呢？**架構師** 應該負責設計和指導，但是許多名片上印著“架構師”的人配不上這個稱號。作為架構師，不應該只是畫一些看起來很漂亮的設計圖，說一些像“黑話”一樣的詞彙，使用一大堆設計模式——這樣的設計通常不會有效的。

這些架構師通常在項目開始時介入，繪製各種各樣的設計圖，然後在重要的代碼實現開始之前離開。有太多這種“PowerPoint架構師”了，由於得不到反饋，他們的架構設計工作也不會有很好的收效。

一個設計要解決的是眼前面臨的特定問題，隨著設計的實現，對問題的理解也會發生改變。想在開始實現之前，就做出一個很有效的詳細設計是非常困難的（見第48頁習慣11）。因為沒有足夠的上下文，能得到的反饋也很少，甚至沒有。設計會隨著時間而演進，如果忽略了應用的現狀（它的具體實現），要想設計一個新的功能，或者完成某個功能的提升是不可能的。

作為設計人員，如果不能理解系統的具體細節，就不可能做出有效的設計。只通過一些高度概括的、粗略的設計圖無法很好地理解系統。

這就像是嘗試僅僅通過查看地圖來指揮一場戰役——一旦開打，僅有計劃是不夠的。戰略上的決策也許可以在後方進行，但是戰術決策——影響成敗的決策需要對戰場狀況的明確瞭解。①

① 第一次世界大戰中，所門戰役（the Battle of the Somme）本應成為一個有決定性意義的突破。實際上，它卻成為了20世紀最愚蠢的軍事行動。最重要的原因是，由於斷絕了通信聯繫，面對的戰場情況與早先的預測已經完全不同了，指揮官仍然堅持按照原計劃展開戰役。請查看<http://www.worldwar1.com/sfsomme.htm>。

---

## 可 逆 性

《程序員修煉之道》中指出不存在所謂的最終決策。沒有哪個決策做出之後就是板上釘釘了。實際上，就時間性來看，不妨把每個重要的決策，都看作沙上堆砌的城堡，它們都是在變化之前所做出的預先規劃。

---

## 新系統的設計者

新系統的設計者必須要親自投入到實現中去。

——Donald E. Knuth②

② 計算機科學大師，圖靈獎得主，經典著作《計算機程序設計藝術》作者。——編者注

正像Knuth說的，好的設計者必須能夠捲起袖子，加入開發隊伍，毫不猶豫地參與實際編程。真正的架構師，如果不允許參與編碼的話，他們會提出強烈的抗議。

有一句泰米爾諺語說：“只有一張蔬菜圖無法做出好的咖喱菜。”與之類似，紙上的設計也無法產生優秀的應用。應該根據設計開發出原型，經過測試，當然還有驗證——它是要演化的。實現可用的設計，這是設計者或者說架構師的責任。

Martin Fowler在題為“Who Needs an Architect?”<sup>③</sup>的文章中提到：一個真正的架構師“.....應該指導開發團隊，提升他們的水平，以解決更為複雜的問題”。他接著說：“我認為架構師最重要的任務是：通過找到移除軟件設計不可逆性的方式，從而去除所謂架構的概念。”增強可逆性是注重實效的軟件實現方式的關鍵構成部分。

③

<http://www.martinfowler.com/ieeesoftware/whoNeedsArchitect.pdf>

。

要鼓勵程序員參與設計。主力程序員應該試著擔任架構師的角色，而且可以從事多種不同的角色。他會負責解決設計上的問題，同時也不會放棄編碼的工作。如果開發人員不願意承擔設計的責任，要給他們配備一個有良好設計能力的人。程序員在拒絕設計的同時，也就放棄了思考。



**優秀的設計從積極的程序員那裡開始演化**。積極的編程可以帶來深入的理解。不要使用不願意編程的架構師——不知道系統的真實情況，是無法展開設計的。

## 切身感受

架構、設計、編碼和測試，這些工作給人的感覺就像是同一個活動——開發的不同方面。感覺它們彼此之間應該是不可分割的。

## 平衡的藝術

- 如果有一位首席架構師，他可能沒有足夠的時間來參與編碼工作。還是要讓他參與，但是別讓他開發在項目關鍵路徑上的、工作量最大的代碼。
- 不要允許任何人單獨進行設計，特別是你自己。



## 40 實行代碼集體所有制

“不用擔心那個煩人的bug，Joe下週假期結束回來後會把它解決掉的。在此之前先想個權宜之計應付一下吧。”



任何具備一定規模的應用，都需要多人協作進行開發。在這種狀況下，不應該像國家宣稱對領土的所有權一樣，聲明個人對代碼的所有權。任何一位團隊成員，只要理解某段代碼的來龍去脈，就應該可以對其進行處理。如果某一段代碼只有一位開發人員能夠處理，項目的風險無形中也就增加了。

相比找出誰的主意最好、誰的代碼實現很爛而言，解決問題，並讓應用滿足用戶的期望要更為重要。

當多人同時開發時，代碼會被頻繁地檢查、重構以及維護。如果需要修復bug，任何一名開發人員都可以完成這項工作。同時有兩個或兩個以上的人，可以處理應用中不同部分的代碼，可以讓項目的日程安排也變得更為容易。

在團隊中實行任務輪換制，讓每個成員都可以接觸到不同部分的代碼，可以提升團隊整體的知識和專業技能。當Joe接過Sally的代碼，他可以對其進行重構，消除待處理的問題。在試圖理解代碼的時候，他會問些有用的問題，儘早開始對問題領域的深入理解。

另一方面，知道別人將會接過自己的代碼，就意味著自己要更守規矩。當知道別人在注意時，一定會更加小心。

可能有人會說，如果一個開發者專門應對某一個領域中的任務，他就可以精通該領域，並讓後續的開發任務更加高效。這沒錯，但是眼光放長遠一點，有好幾雙眼睛盯著某一段代碼，是一定可以帶來好處的。這樣可以提升代碼的整體質量，使其易於維護和理解，並降低出錯率。



**要強調代碼的集體所有制**。讓開發人員輪換完成系統不同領域中不同模塊的不同任務。

## 切身感受

項目中絕大部分的代碼都可以輕鬆應對。

## 平衡的藝術

- 不要無意間喪失了團隊的專家技能。如果某個開發人員在某個領域中極其精通，不妨讓他作為這方面的駐留專家，而且系統的其他部分代碼也對他開放，這樣對團隊和項目都很有幫助。
- 在大型項目中，如果每個人都可以隨意改變任何代碼，一定會把項目弄得一團糟。代碼集體所有制並不意味著可以隨心所欲、到處破壞。
- 開發人員不必瞭解項目每一部分的每個細節，但是也不能因為要處理某個模塊的代碼而感到驚恐。
- 有些場合是不能採用代碼集體所有制的。也許代碼需要某些特定的知識、對特定問題域的瞭解，比如一個高難度的實時控制系統。這些時候，人多了反而容易誤事。
- 任何人都可能遭遇到諸如車禍等突發的災難事故，或者有可能被競爭對手僱傭。如果不向整個團隊分享知識，反而增加了喪失知識的風險。

# 41 成為指導者

“你花費了大量的時間和精力，才達到目前的水平。對別人要有所保留，這樣讓你看起來更有水平。讓隊友對你超群的技能感到



恐懼吧。”

## 教學相長

### Knowledge grows when given

我們有時會發現自己在某些方面，比其他團隊成員知道得更多。那要怎麼對待這種新發現的“權威地位”呢？當然，可以用它來質疑別人，取笑他人做出的決策和開發的代碼——有些人就是這樣做的。不過，我們可以共享自己的知識，讓身邊的人變得更好。

好的想法不會因為被許多人瞭解而削弱。當我聽到你的主意時，我得到了知識，你的主意也還是很棒。同樣的道理，如果你用你的蠟燭點燃了我的，我在得到光明的同時，也沒有讓你的周圍變暗。好主意就像火，可以引領這個世界，同時不削弱自己。①

① 托馬斯·傑弗遜，美國第三任總統，獨立宣言起草人。

與團隊其他人一起共事是很好的學習機會。知識有一些很獨特的屬性；假設你給別人錢的話，最後你的錢會變少，而他們的財富會增多。但如果是去教育別人，那雙方都可以得到更多的知識。

通過詳細解釋自己知道的東西，可以使自己的理解更深入。當別人提出問題時，也可以發現不同的角度。也許可以發現一些新技巧——聽到一個聲音這樣告訴自己：“我以前還沒有這樣思考過這個問題。”

與別人共事，激勵他們變得更出色，同時可以提升團隊的整體實力。遇到無法回答的問題時，說明這個領域的知識還不夠完善，需要在這方面進一步增強。好的指導者在為他人提供建議時會做筆記。如果遇到需要花時間進一步觀察和思考的問題，不妨先草草記錄下來。此後將這些筆記加入到每日日誌中（見第129頁習慣33）。

成為指導者，並不意味著要手把手教團隊成員怎麼做（見第160頁習慣42），也不是說要在白板前進行講座，或是開展小測驗什麼的，可以在進行自備午餐會時展開討論。多數時候，成為指導者，是指在幫助團隊成員提升水平的同時也提高自己。

這個過程不必侷限於自己的團隊。可以開設個人博客，貼一些代碼和技術在上面。不一定是多麼偉大的項目，即使是一小段代碼和解釋，對別人也可能是有幫助的。

成為指導者意味著要分享——而不是固守——自己的知識、經驗和體會。意味著要對別人的所學和工作感興趣，同時願意為團隊增加價值。一切都是為了提高隊友和你的能力與水平，而不是為了毀掉團隊。

然而，努力爬到高處，再以蔑視的眼神輕視其他人，這似乎是人類本性。也許在沒有意識到的情況下，溝通的障礙就已經建立起來了。團隊中的其他人可能出於畏懼或尷尬，而不願提出問題，這樣就無法完成知識的交換了。這類團隊中的專家，就像是擁有無數金銀財寶的有錢人，卻因健康原因無福享受。我們要成為指導別人的人，而不是折磨別人的人。



**成為指導者**。分享自己的知識很有趣——付出的同時便有收穫。還可以激勵別人獲得更好的成果，而且提升了整個團隊的實力。

## 切身感受

你會感到給予別人教導，也是提升自己學識的一種方式，並且其他人亦開始相信你可以幫助他們。

## 平衡的藝術

- 如果一直在就同一個主題向不同的人反覆闡述，不妨記錄筆記，此後就此主題寫一篇文章，甚至是一本書。
- 成為指導者是向團隊進行投資的一種極佳的方式。（見第31頁習慣6。）
- 結對編程（見第165頁習慣44）是一種進行高效指導的、很自然的環境。
- 如果總是被一些懶於自己尋找答案的人打擾（查看下一頁習慣42）。
- 為團隊成員在尋求幫助之前陷入某個問題的時間設定一個時限，一個小時應該是不錯的選擇。

## 42 允許大家自己想辦法

“你這麼聰明，直接把乾淨利落的解決方案告訴團隊其他人就好了。不用浪費時間告訴他們為什麼這樣做。”



“授人以魚，三餐之需；授人以漁，終生之用。”告訴團隊成員解決問題的方法，也要讓他們知道如何解決問題的思路，這也是成為指導者的一部分。

瞭解上個實踐——**成為指導者**——之後，也許有人會傾向於直接給同事一個答案，以繼續完成工作任務。要是隻提供一些指引給他們，讓他們自己想辦法找到答案，又會如何？

這並不是多麼麻煩的事情；不要直接給出像“42”這樣的答案，應該問你的隊友：“你有沒有查看在事務管理者與應用的鎖處理程序之間的交互關係？”

這樣做有下面幾點好處。

- 你在幫助他們學會如何解決問題。
- 除了答案之外，他們可以學到更多東西。
- 他們不會再就類似的問題反覆問你。
- 這樣做，可以幫助他們在你不能回答問題時自己想辦法。
- 他們可能想出你沒有考慮到的解決方法或者主意。這是最有趣的——你也可以學到新東西。

如果有人還是沒有任何線索，那就給更多提示吧（或者甚至是答案）。如果有人提出來某些想法，不妨幫他們分析每種想法的優劣之處。如果有人給出的答案或解決方法更好，那就從中汲取經驗，然後分享你的體會吧。這對雙方來說都是極佳的學習經驗。

作為指導者，應該鼓勵、引領大家思考如何解決問題。前面提到過亞里士多德的話：“接納別人的想法，而不是盲目接受，這是受過教育的頭腦的標誌。”應該接納別人的想法和看問題的角度，在這個過程中，自己的頭腦也得到了拓展。

如果整個團隊都能夠採納這樣的態度，可以發現團隊的知識資本在快速提升，而且將會完成一些極其出色的工作成果。



**給別人解決問題的機會**。指給他們正確的方向，而不是直接提供解決方案。每個人都能從中學到不少東西。

## 切身感受

感覺不是在以填鴨式的方式給予別人幫助。不是有意掩飾，更非諱莫如深，而是帶領大家找到自己的解決方案。

## 平衡的藝術

- 用問題來回答問題，可以引導提問的人走上正確的道路。
- 如果有人真的陷入膠著狀態，就不要折磨他們了。告訴他們答案，再解釋為什麼是這樣。

## 43 準備好後再共享代碼

“別管是不是達到代碼簽入的要求，要儘可能頻繁地提交代碼，特別是在要下班的時候。”



讓你猜個謎語：相對不使用版本控制系統，更壞的狀況是什麼？答案是：錯誤地使用了版本控制系統。使用版本控制系統的方式，會影響生產力、產品穩定性、產品質量和開發日程。特別地，諸如代碼提交頻率這樣簡單的東西都會有很大影響。

完成一項任務後，應該馬上提交代碼，不應該讓代碼在開發機器上多停留一分鐘。如果代碼不能被別人集成使用，那又有什麼用處呢？應該趕緊發佈出去，並開始收集反饋。①

① 而且，不應該將僅有的一份代碼保存在只有“90天有限質保”的硬盤中。

很明顯，每週或每月一次提交代碼，並不是令人滿意的做法——這樣源代碼控制系統就不能發揮其作用了。也許總有種種原因來為這種懶散的做法解釋。有人說開發人員是採取異地開發（**off-site**）或離岸開發（**offshore**）的方式，訪問源代碼控制系統的速度很慢。這就是**環境黏性**（**environmental viscosity**）的例子——把事情做糟要比做好更容易。很明顯，這是一個亟待解決的簡單技術問題。

另一方面，如果在任務完成之前就提交代碼又會如何？也許你正在開發一些至關重要的代碼，而且你想在下班回家晚飯之後再繼續開發。要想在家裡得到代碼，最簡單的方式就是將其提交到源代碼控制系統，到家之後再把代碼簽出。

向代碼庫中提交仍在開發的代碼，會帶來很多風險。這些代碼可能還有編譯錯誤，或者對其所做的某些變化與系統其他部分的代碼不兼容。當其他開發者獲取最新版本的代碼時，也會受到這些代碼的影響。

通常情況下，提交的文件應該與一個特定的任務或是一個bug的解決相關。而且應該是同時提交相關的文件，並注有日誌信息，將來也能夠知道修改了哪些地方，以及為什麼要做修改。一旦需要對變更採取回滾操作，這種“原子”提交也是有幫助的。

要保證在提交代碼之前，所有的單元測試都是可以通過的。使用持續集成是保證源代碼控制系統中代碼沒有問題的一種良好方式。

### 代碼不執行提交操作的其他安全選擇

如果需要將尚未完成的源代碼傳輸或是保存起來，有如下選擇。

**使用遠程訪問**。將代碼留在工作地點，然後在家裡使用遠程訪問獲取，而不是將完成了一半的代碼提交，再從家裡簽出。

**隨身攜帶**。將代碼複製到U盤、CD或DVD中，以達到異地開發的目的。

**使用帶有底座擴展的筆記本電腦**。如果是由於在多臺電腦上開發造成的延續性問題，不妨考慮使用帶有底座擴展的筆記本電腦，這樣就可以帶著代碼到處走了。

**使用源代碼控制系統的特性**。Microsoft Visual Team System 2005有一個“shelving”特性，因為有些產品的某些代碼在提交之前，需要被其他部分調用。在CVS和Subversion中，可以將尚未允許合併到主幹的代碼，設定為開發者的分支（查看[TH03]和[Mas05]）。

---





**準備好後再共享代碼**。絕不要提交尚未完成的代碼。故意簽入編譯未通過或是沒有通過單元測試的代碼，對項目來說，應被視作翫忽職守的犯罪行為。

## 切身感受

感覺好像整個團隊就在源代碼控制系統的另一端盯著你。要知道一旦提交代碼，別人就都可以訪問了。

## 平衡的藝術

- 有些源代碼控制系統會區分“提交”和“可公開訪問”兩種代碼權限。此時，可以進行臨時的提交操作（比如在工作地點和家之間來回奔波時），不會因為完全提交未完成的代碼，而讓團隊的其他成員感到鬱悶。
- 有些人希望代碼在提交之前可以進行復查操作。只要不會過久拖延提交代碼的時間就沒有問題。如果流程的某個部分產生了拖延，那就修正流程吧。
- 仍然應該頻繁提交代碼。不能用“代碼尚未完成”作為避免提交代碼的藉口。

# 44 做代碼複查

“用戶是最好的測試人員。別擔心——如果哪裡出錯了，他們會告訴我們的。”



代碼剛剛完成時，是尋找問題的最佳時機。如果放任不管，它也不會變得更好。

## 代碼複查和缺陷移除

要尋找深藏不露的程序bug，正式地進行代碼檢查，其效果是任何已知形式測試的兩倍，而且是移除80%缺陷的唯一已知方法。

——Capers Jones的《估算軟件成本》[Jon98]

正如Capers Jones指出的，代碼複查或許是找到並解決問題的最佳方式。然而，有時很難說服管理層和開發人員使用它來完成開發工作。

管理層擔心進行代碼複查所耗費的時間。他們不希望團隊停止編碼，而去參加長時間的代碼複查會議。開發人員對代碼複查感到擔心，允許別人看他們的代碼，會讓他們有受威脅的感覺。這影響了他們的自尊心。他們擔心在情感上受到打擊。

作者參與過的項目中，只要實施了代碼複查，其成果都是非常顯著的。

Venkat最近參與了一個日程安排非常緊湊的項目，團隊不少成員都是沒有多少經驗的開發者。通過嚴格的代碼複查過程，他們可以提交質量極高而且穩定的代碼。當開發人員完成某項任務的編碼和測試後，在簽入源代碼控制系統之前，會有另一名開發人員對代碼做徹底的複查。

這個過程修復了很多問題。噢，代碼複查不只針對初級開發者編寫的代碼——團隊中每個開發人員的代碼都應該進行複查，無論其經驗豐富與否。

那該如何進行代碼複查呢？可以從下面這些不同的基本方式中進行選擇。

- **通宵複查**。可以將整個團隊召集在一起，預定好美食，每個月進行一次“恐怖的代碼複查之夜”。但這可能不是進行代碼複查最有效的方式（而且聽起來也不太敏捷）。大規模團隊的複查會議很容易陷入無休止的討論之中。大範圍的複查不僅沒有必要，而且有可能對整個流程造成損害。我們不建議這種方式。

- **撿拾遊戲**。當某些代碼編寫完成、通過編譯、完成測試，並已經準備簽入時，其他開發人員就可以“撿拾”起這些代碼開始複查。類似的“提交複查”是一種快速而非正式的方式，保證代碼在提交之前是可以被接受的。為了消除行為上的慣性，要在開發人員之間進行輪換。比如，如果Joey的代碼上次是由Jane複查的，這次不妨讓Mark來複查。這是一種很有效的技術。①

① 要了解這種方式的更多細節，查看*Ship It !* [RG05]一書。

- **結對編程**。在極限編程中，不存在一個人獨立進行編碼的情況。編程總是成對進行的：一個人在鍵盤旁邊（擔任**司機**的角色），另一個人坐在後面擔任**導航員**。他們會不時變換角色。有第二雙眼睛在旁邊盯著，就像是在進行持續的代碼複查活動，也就不必安排單獨的特定複查時間了。

在代碼複查中要看什麼呢？你可能會制訂出要檢查的一些特定問題列表（所有的異常處理程序不允許空，所有的數據庫調用都要在包的事務中進行，等等），不過這裡是一個可供啟動的最基本的檢查列表。

- 代碼能否被讀懂和理解？
- 是否有任何明顯的錯誤？
- 代碼是否會對應用的其他部分產生不良影響？
- 是否存在重複的代碼（在複查的這部分代碼中，或是在系統的其他部分代碼）？
- 是否存在可以改進或重構的部分？

此外，還可以考慮使用諸如**Similarity Analyzer**或**Jester**這樣的代碼分析工具。如果這些工具產生的靜態分析結果對項目有幫助，就把它們集成到持續構建中去吧。



**複查所有的代碼**。對於提升代碼質量和降低錯誤率來說，代碼複查是無價之寶。如果以正確的方式進行，複查可以產生非常實用而高效的成果。要讓不同的開發人員在每個任務完成後複查代碼。

## 切身感受

代碼複查隨著開發活動持續進行，而且每次針對的代碼量相對較少。感覺複查活動就像是項目正在進行的一部分，而不是一種令人畏懼的事情。

## 平衡的藝術

- 不進行思考、類似於橡皮圖章一樣的代碼複查沒有任何價值。
- 代碼複查需要積極評估代碼的設計和清晰程度，而不只是考量變量名和代碼格式是否符合組織的標準。
- 同樣的功能，不同開發人員的代碼實現可能不同。差異並不意味著不好。除非你可以讓某段代碼明確變得更好，否則不要隨意批評別人的代碼。
- 如果不及時跟進討論中給出的建議，代碼複查是沒有實際價值的。可以安排跟進會議，或者使用代碼標記系統，來標識需要完成的工作，跟蹤已經處理完的部分。
- 要確保代碼複查參與人員得到每次複查活動的反饋。作為結果，要讓每個人知道複查完成後所採取的行動。

# 45 及時通報進展與問題

“管理層、項目團隊以及業務所有方，都仰仗你來完成任務。如果他們想知道進展狀況，會主動找你要的。還是埋頭繼續做事



吧。”

接受一個任務，也就意味著做出了要準時交付的承諾。不過，遇到各種問題從而導致延遲，這種情形並不少見。截止日期來臨，大家都等著你在演示會議上展示工作成果。如果你到會後通知大家工作還沒有完成，會有什麼後果？除了感到窘迫，這對你的事業發展也沒有什麼好處。

如果等到截止時間才發佈壞消息，就等於是為經理和技術主管提供了對你進行微觀管理（**micromanagement**）的機會。他們會擔心你再次讓他們失望，並開始每天多次檢查你的工作進度。你的生活就開始變得像呆伯特的漫畫一樣了。

假定現在你手上有一個進行了一半的任務，由於技術上的難題，看起來不能準時完成了。如果這時積極通知其他相關各方，就等於給機會讓他們提前找出解決問題的方案。也許他們可以向另外的開發人員尋求幫助，也許他們可以將工作重新分配給更加熟悉相關技術的人，也許他們可以提供更多需要的資源，或者調整目前這個迭代中要完成的工作範圍。客戶會願意將這個任務用其他同等重要的任務進行交換的。

及時通報進展與問題，有情況發生時，就不會讓別人感到突然，而且他們也很願意瞭解目前的進展狀況。他們會知道何時應提供幫助，而且你也獲得了他們的信任。

發送電子郵件，用即時貼傳遞信息，或快速電話通知，這都是通報大家的傳統方式。還可以使用Alistair Cockburn提出的“信息輻射器”。<sup>①</sup>信息輻射器類似於牆上的海報，提供變更的信息。路人可以很方便地瞭解其中的內容。以推送的方式傳遞信息，他們就不必再來問問題了。信息輻射器中可以展示目前的任務進度，和團隊、管理層或客戶可能會感興趣的其他內容。

① 查看<http://c2.com/cgi-bin/wiki?InformationRadiator>。

也可以使用海報、網站、Wiki、博客或者RSS Feed。只要讓人們可以有規律地 查看到需要的信息，這就可以了。

整個團隊可以使用信息輻射器來發布他們的狀態、代碼設計、研究出的好點子等內容。現在只要繞著團隊的工作區走一圈，就可以學到不少新東西，而且管理層也就可以知道目前的狀況如何了。



**及時通報進展與問題**。發佈進展狀況、新的想法和目前正在關注的主題。不要等著別人來問項目狀態如何。

## 切身感受

當經理或同事來詢問工作進展、最新的設計，或研究狀況時，不會感到頭痛。

## 平衡的藝術

- 每日立會（見第148頁習慣38）可以讓每個人都能明確瞭解最新的進展和形勢。
- 在展示進度狀況時，要照顧到受眾關注的細節程度。舉例來說，CEO和企業主是不會關心抽象基類設計的具體細節的。
- 別花費太多時間在進展與問題通報上面，還是應該保證開發任務的順利完成。
- 經常抬頭看看四周，而不是隻埋頭於自己的工作。

# 第9章 尾聲：走向敏捷

一燈能除千年暗，一智能滅萬年愚。

——慧能，中國禪宗第6代祖師

一點智慧，只要有它便足夠了。我們希望大家喜歡對這些敏捷實踐的描述，而且其中有一到兩個可以點燃諸位智慧的火花。

無論經驗是否豐富，不管過去有什麼樣的成功，遇到過什麼樣的挑戰，只要進行一個新的實踐，就可以讓人頭腦清醒，並讓你的工作與生活從此發生改變。使用這些實踐的子集，能夠救瀕臨失敗的項目於水火，也可以使得從此往後的項目變得完全不同。

## 9.1 只要一個新的習慣

舉個例子，看看Andy曾經服務過的一個客戶的故事。他們的辦公室位於一座具有玻璃外牆的、高聳的寫字樓中，團隊的辦公室沿著外牆排列，形成一條優雅的曲線。每個人都能看到窗外的風景，整個團隊的分佈幾乎佔用了樓層一半的牆內空間。但是這個團隊有不少問題：版本發佈總在延期，團隊逐漸失去了對不斷增多的bug的控制。

按照通常的工作方式，Andy和注重實效的程序員們，坐在辦公室的一端，開始對團隊進行訪談，以瞭解他們的工作是如何開展的，有哪些進展順利，哪些構成了障礙。第一位成員解釋說，他們在開發一個C/S應用，客戶端非常瘦，所有的業務邏輯和數據庫訪問都放在服務器一端。

然而，隨著訪談的不斷進行，故事卻慢慢發生了變化。每個人對項目的方向和目標的瞭解都有所偏差。最終，最後一個成員驕傲地宣稱：系統的構成包括一個包含全部GUI和業務邏輯的胖客戶端，以及僅僅包含一個簡單數據庫的服務器！

現在問題就很清楚了，團隊從來沒有坐在一起討論過項目。實際上，每位成員僅僅與坐在旁邊的人有過談論。就像是學校裡的孩子們玩過

的“傳話”遊戲，信息在人與人之間傳遞時產生了偏差，最終偏離了本意。

需要哪種有實效的建議？馬上開始使用立會吧（見第148頁習慣38）。這種做法很快就收到了令人驚異的效果。不只很快解決了架構上的問題，而且產生了更深遠的影響。團隊開始變得更有凝聚力，彼此緊密配合，共同工作。**bug**產生率降低了，產品變得越來越穩定，截止日期也不再像以前那樣令人窒息。

沒有花費太多時間，也沒有耗費多少精力，只需要一些規矩來保證立會的舉行，不過這很快就形成習慣了。**只要一個新的習慣**，就讓團隊發生了巨大的變化。

## 9.2 拯救瀕臨失敗的項目

如果採納一個習慣可以產生好的效果，那麼採納所有的習慣，就應該產生更好的影響，是嗎？最終一定是這樣的，但是不能一下子全部上馬——特別是對一個已經處於困境的項目。突然改變某個項目的全部開發習慣，是讓項目突然死亡的最佳方式。

用一個醫學上的比喻，假定有一個胸部疼痛的病人。當然，如果病人經常運動而且保持健康飲食的話，他們不會生病。但是不能因此就馬上說：“別賴在床上了，爬起來開始在跑步機上運動吧。”這有可能要了病人的命，而且你的瀆職保險賠償率一定會升高。

必須要穩定病人的狀況，並使用最小劑量的（但是必要的）藥物和治療過程。只有在病人身體狀況恢復且趨於穩定之後，才能讓他按照良好的飲食起居制度來安排自己的生活，保證身體的健康。

當項目岌岌可危時，應該先引入一系列習慣來穩定目前的狀況。看這個例子：一個潛在的客戶曾經以驚恐的聲調打電話給**Venkat**，說他們的項目陷入危機。他們已經耗費了一半的時間，而項目還有**90%**的成



果要交付。管理層對於開發人員不能及時完成任務感到很不高興。開發人員對於管理層總是逼得這麼急也覺得很不爽。剩下的時間，他們是應該用來修補bug，還是開發新功能？不管危機發展到什麼程度，團隊總是希望獲得成功，然而他們不知道該怎麼辦。所做的每件事情都讓他們落後更遠。他們感到了威脅並且不願再做任何決策。

Venkat沒有試圖一次解決全部的問題，他必須先穩定病人的狀況，使用了下面這些促進溝通和協作的敏捷習慣作為開始：第18頁習慣3，第148頁習慣38，第162頁習慣43，以及第168頁習慣45。以此為起點，下一步要引入一些與發佈相關的習慣，比如第55頁習慣13，第58頁習慣14。最終，他們採納了一些與編碼相關的習慣，比如第132頁習慣34，第136頁習慣35。這就足夠解決目前的危機了，項目比預定時間早兩週完成，並得到了管理層的認可。

這就是緊急救助的模型。如果事態沒有那麼糟，可以採取更加全面、整齊的方式來引入敏捷習慣。無論你是經理，還是團隊帶頭人，或者只是一個對敏捷感興趣、試圖從組織內部發起敏捷過程的程序員，我們都有一些針對性的建議。

## 9.3 引入敏捷：管理者指南

作為一個管理者或者團隊的帶頭人，有責任先讓整個團隊知道接下來將要發生什麼。向大家說明**敏捷開發是要讓開發人員的工作變得更加輕鬆**，這主要是為了他們好（根本上來看，對用戶和組織也是有益處的）。如果沒有達到這樣的效果，那就是有些地方出了問題。

要慢慢來。記得領導所做的每一個小動作，都會隨著時間對團隊產生巨大的影響。①

① 可以查看*Behind Closed Doors* [RD05]，這是一本關於如何領導團隊和提升管理技能的好書。

將這些主意介紹給團隊的時候，要說明在第10頁第2章中的幾條基本原則。確保每個人都知道項目將會以此運轉——而不只是口頭上說說而已。

從立會開始（見第148頁習慣38）。這可以讓團隊有機會進行彼此討論，並對一些重大問題達成共識。把之前相對孤立的架構師帶到團隊中，並讓他們參與到日常開發工作（見第152頁習慣39）。開展非正式的代碼複查（見第165頁習慣44），並做出計劃，讓客戶與用戶也參與到項目中來（見第45頁習慣10）。

接下來要準備好開發的基本環境。也就是說要開始採納（或改進）基本的入門級別習慣。

- 版本控制
- 單元測試
- 自動構建

版本控制是第一要務。在任何項目中，它都必須是要最先搭建好的基本架構。設置好後，就要為每個開發人員安排好各自要使用的本地構建項目，這些項目要與服務器保持一致，可以通過腳本運行構建操作，還要能夠運行任何可用的單元測試。這些都搞定之後，就可以開始為正在開發的新代碼創建單元測試，並按需為已有代碼創建新的測試了。最後，要準備一臺供後臺運行持續構建的機器，使之起到棒球比賽中“擋球網”的作用，以捕獲任何發生的問題。

如果你對這些領域不熟悉，到最近的書店（或 [www.PragmaticBookshelf.com](http://www.PragmaticBookshelf.com)）去買一本 *Ship It!* [RG05]，它會告訴你如何設置相關的環境和運行機制。入門工具箱（**Starter Kit**）系列圖書可以幫你完成如何在特定環境下配置版本控制、單元測試，以及自動化等具體細節。

基礎架構搭建好後，就要考慮如何將項目和團隊帶入到固定的節奏中了。可以再次閱讀第43頁第4章，來了解項目的時間安排和節奏相關的內容。

現在應該已經對基本知識都有所瞭解了，接下來應該調整習慣，以讓它們適用於你的團隊。在設定環境時，可以回顧一下在第76頁第5章，接下來再看看在第98頁第6章和第138頁第7章，瞭解如何以敏捷的方式來解決日常問題。

最後，開始引入在第26頁第3章提到的自備午餐會和其他習慣，並開始使用在第146頁第8章**敏捷協作**的習慣，讓團隊可以緊密配合，共同工作。但這並不是結束，還有很多其他工作可以開展，很多習慣可以採納。

要不時——也許是在每個迭代結束後，或每個版本發佈完成後——舉辦項目回顧會議。從團隊處得到反饋：哪些做得不錯，哪些需要調整，哪些不起作用。如果之前採納的某個習慣沒有達到預期效果，翻回頭查閱本書中對應習慣的“**切身感受**”和“**平衡的藝術**”兩個部分，看看是不是有哪些細節方面出了問題，並且進行修正。

## 9.4 引入敏捷：程序員指南

如果你不負責帶領團隊，但是希望讓大家向敏捷的方向努力，就要面臨不少挑戰了。不單要完成前一節列出的各種事項，還應該通過實際的例子，而不是行政命令來引領大家。

有句老話說得好：“你可以把馬帶到水邊……但是你不能強迫它使用你最鍾愛的代碼編輯器。”<sup>①</sup>當然，除非你已經用得非常熟練了。只要好處明顯，團隊成員肯定會希望儘快著手使用的。

① 西方諺語，原文為：You can lead a horse to water, but you cannot make him drink（你可以帶馬到水邊，但不能勉強它喝）

水)，其意指：善意不足以成事。——譯者注

舉例來說，從單元測試開始是一個不錯的選擇。可以先針對自己的代碼開始使用。在短時間之內（幾周甚至更少），就可以看到代碼質量提升了——減少了錯誤的數目，提高了質量，健壯性也有所提升。你下午5點就可以下班回家，而且所有的任務都可以順利完成——不必擔心半夜被電話叫醒，去修復bug。旁邊的開發人員想知道你是如何做到的，而且消息也漸漸傳開了。整個團隊現在都想嘗試這些新奇的習慣，而不需要你努力去說服他們。

如果要將團隊帶入新的領域，必須首先以身作則。所以不妨從可以馬上著手的習慣做起。在第10頁第2章中的習慣是個不錯的起點，比如這幾個偏重編碼的習慣：第78頁習慣19和第82頁習慣20；還有第98頁第6章和第128頁第7章中的習慣。還可以在自己機器上運行一個持續構建服務器，發生問題時可以馬上知道。隊友可能會覺得你有“千里眼”呢。

過些時間之後，可以試著開始一些非正式的自備午餐會（見第31頁習慣6），與大家一起討論關於敏捷項目的節奏（見第40頁習慣9）和其他感興趣的話題。

## 9.5 結束了嗎

本書內容馬上就要結束了，下面該怎麼做就看你自己了。不妨應用這些習慣，看看對自己有哪些好處，也可以帶領整個團隊著手，以更加輕鬆和快速的方式開發更好的軟件。

可以訪問我們的網站，在那裡可以找到作者的博客以及其他文章，包括相關資源的鏈接。

感謝你的閱讀。

## 附錄A 資源

### A.1 Web資源

#### 敏捷開發人員

<http://www.agiledeveloper.com/download.aspx>

Agile Developer下載頁面，從中可以找到Venkat Subramaniam的文章和演示。

#### Andy的博客

<http://toolshed.com/blog>

Andy Hunt的博客，覆蓋了很多話題，甚至還有一點關於軟件開發的內容。

#### Anthill

<http://www.urbancode.com/projects/anthill/default.jsp>

控制構建過程、達到持續集成效果的工具，可以提升組織內部的知識共享程度。

#### Unix編程藝術

<http://www.faqs.org/docs/artu/ch04s02.html>

Eric Steven Raymond的《Unix編程藝術》一書節選。

## 持續集成

<http://www.martinfowler.com/articles/continuousIntegration.html>

告訴你持續集成好處所在的文章。

## CruiseControl

<http://cruisecontrol.sourceforge.net>

主要供開發Java應用使用的持續集成工具。供.NET平臺使用的C#版本名為CruiseControl.NET，可從<http://sourceforge.net/projects/ccnet>下載。

## Damage Control

<http://dev.buildpatterns.com/trac/wiki/DamageControl>

用Ruby on Rails編寫的持續集成工具。

## Draco.NET

<http://draconet.sourceforge.net>

供.NET平臺使用的持續集成工具，通過Windows服務的方式運行。

## 依賴倒置 ( **Dependency Inversion** ) 原則

<http://c2.com/cgi/wiki?DependencyInversionPrinciple>

介紹依賴倒置原則的短文。

## FIT集成測試框架

<http://fit.c2.com>

可以自動對比客戶期望結果與應用實際運行結果的協作工具。

## **Google Groups**

<http://groups.google.com>

訪問用戶組討論的站點。

## **信息輻射器**

<http://c2.com/cgi-bin/wiki?InformationRadiator>

對Alistair Cockburn信息輻射器概念的討論。

## **設計已死？**

<http://www.martinfowler.com/articles/designDead.html>

Martin Fowler所寫，關於設計在敏捷開發中的重要意義和角色的文章。

## **JUnit**

<http://www.junit.org>

供使用JUnit或其他語言測試工具XUnit系列測試框架的軟件開發人員使用的站點。

## **JUnitPerf**

<http://www.clarkware.com/software/JUnitPerf.html>

一系列JUnit測試用的decorator模式代碼，用來測算現有JUnit測試用例中功能的性能和可伸縮性。

## **NUnit**

<http://sourceforge.net/projects/nunit>

專供使用NUnit的軟件開發人員使用的站點。

## 面向對象設計原則

<http://c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign>

集合了多個極佳的面向對象程序設計原則的網頁。

## 對象—關係映射

<http://www.neward.net/ted/weblog/index.jsp?date=20041003#1096871640048>

Ted Neward對框架的討論，包括他著名的引語“對象關係映射就是計算機科學中的越南戰場”。

## 開放—封閉原則

<http://www.objectmentor.com/resources/articles/ocp.pdf>

介紹了開放—封閉原則的實例和限制。

## 開放—封閉原則簡要介紹

<http://c2.com/cgi/wiki?OpenClosedPrinciple>

關於開放—封閉原則優劣的討論。

## 注重實效的編程

<http://www.pragmaticprogrammer.com>

Pragmatic Programmer公司的主頁，可以從中找到Programtic Bookshelf書籍（包括本書）的鏈接，包括供開發人員和管理層使用的



信息。

### **單一職責原則**

<http://c2.com/cgi-bin/wiki?SingleResponsibilityPrinciple>

描述了單一職責原則，並提供了相關文章和討論的鏈接。

### **軟件項目管理實踐：失敗與成功**

<http://www.stsc.hill.af.mil/crosstalk/2004/10/0410Jones.html>

Capers Jones對250個軟件項目成敗的分析。

### **測試驅動開發**

<http://c2.com/cgi/wiki?TestDrivenDevelopment>

對測試驅動開發的介紹。

### **軟件工程的末日和經濟合作博弈的黎明**

<http://alistair.cockburn.us/crystal/articles/eoseatsoecg/theendofsoftwareengineering>

Alistair Cockburn對為什麼軟件開發應被劃歸工程學領域的質疑，以及對新模型的引入。

### **所門戰役的悲劇：第二個巴拉克拉瓦戰役**

<http://www.worldwar1.com/sfsomme.htm>

本站點討論了第一次世界大戰中所門戰役的結果。

### **為什麼你的代碼很爛**

<http://www.artima.com/weblogs/viewpost.jsp?thread=71730>

Dave Astels討論代碼質量的一篇博客文章。

## XProgramming.com

<http://www.xprogramming.com/software.htm>

包括測試工具在內的資源集合。

你不會需要它

<http://c2.com/cgi/wiki?YouArentGonnaNeedIt>

對於“你不會需要它”原則優劣的討論。

## A.2 參考書目

[Bec00]	Kent Beck. <i>Extreme Programming Explained: Embrace Change</i> . Addison- Wesley, Reading, MA, 2000.
[Cla04]	Mike Clark. <i>Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications</i> . The Pragmatic Pro- grammers, LLC, Raleigh, NC, and Dallas, TX, 2004.
[FBB+99]	Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. <i>Refactoring: Improving the Design of Existing Code</i> . Addison Wesley Longman, Reading, MA, 1999.
[Fow05]	Chad Fowler. <i>MyJobWent to India: 52 Ways to Save Your Job</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.
[GHJV95]	Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. <i>Design Patterns: Elements of Reusable Object-Oriented Software</i> . Addison- Wesley, Reading, MA, 1995.
[HT00]	Andrew Hunt and David Thomas. <i>The Pragmatic Programmer: From Journeyman to Master</i> . Addison-Wesley, Reading, MA, 2000.
[HT03]	Andrew Hunt and David Thomas. <i>Pragmatic Unit Testing in Java with JUnit</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.
[HT04]	Andrew Hunt and David Thomas. <i>Pragmatic Unit Testing in C# with NUnit</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2004.

[Jon98]	Capers Jones. <i>Estimating Software Costs</i> . McGraw Hill, 1998.
[Knu92]	Donald Ervin Knuth. <i>Literate Programming</i> . Center for the Study of Language and Information, Stanford, CA, 1992.
[Lar04]	Craig Larman. <i>Agile and Iterative Development: A Manager's Guide</i> . Addison-Wesley, Reading, MA, 2004.
[LC01]	Bo Leuf and Ward Cunningham. <i>The Wiki Way: Collaboration and Sharing on the Internet</i> . Addison-Wesley, Reading, MA, 2001.
[Lis88]	Barbara Liskov. <i>Data abstraction and hierarchy</i> . SIGPLAN Notices, 23(5), May 1988.
[Mar02]	Robert C. Martin. <i>Agile Software Development, Principles, Patterns, and Practices</i> . Prentice Hall, Englewood Cliffs, NJ, 2002.
[Mas05]	Mike Mason. <i>Pragmatic Version Control Using Subversion</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.
[Mey97]	Bertrand Meyer. <i>Object-Oriented Software Construction</i> . Prentice Hall, Englewood Cliffs, NJ, second edition, 1997.
[MR84]	William A Madden and Kyle Y. Rone. <i>Design, development, integration: space shuttle primary flight software system</i> . Communications of the ACM, 27(9):914–925, 1984.
[Rai04]	J. B. Rainsberger. <i>JUnit Recipes: Practical Methods for Programmer Testing</i> . Manning Publications Co., Greenwich, CT, 2004.
[RD05]	Johanna Rothman and Esther Derby. <i>Behind Closed Doors: Secrets of Great Management</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.
[RG05]	Jared Richardson and Will Gwaltney. <i>Ship It! A Practical Guide to Successful Software Projects</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.
[Roy70]	Winston W. Royce. <i>Managing the development of large software systems</i> . Proceedings, IEEE WECON, pages 1–9, August 1970.
[Sch04]	Ken Schwaber. <i>Agile Project Management with Scrum</i> . Microsoft Press, Redmond, WA, 2004.
[Sen90]	Peter Senge. <i>The Fifth Discipline: The Art and Practice of the Learning Organization</i> . Currency/Doubleday, New York, NY, 1990.
[Sha97]	Alec Sharp. <i>Smalltalk by Example: The Developer's Guide</i> . McGraw-Hill, New York, NY, 1997.
[Sub05]	Venkat Subramaniam. <i>.NET Gotchas</i> . O'Reilly & Associates, Inc., Sebastopol, CA, 2005.
[TH01]	David Thomas and Andrew Hunt. <i>Programming Ruby: The Pragmatic Programmer's Guide</i> . Addison-Wesley, Reading, MA, 2001.

[TH03]	David Thomas and Andrew Hunt. <i>Pragmatic Version Control Using CVS</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2003.
[TH05]	David Thomas and David Heinemeier Hansson. <i>Agile Web Development with Rails</i> . The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2005.
[You99]	Edward Yourdon. <i>Death March: The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects</i> . Prentice Hall, Englewood Cliffs, NJ, 1999.

# 索引

## A

Acceptance testing ( 驗收測試 ) · 90, 177

Acronyms ( 縮寫 ) · 28

Agile Developer web site ( 敏捷開發人員網站 ) · 176

Agile Manifesto ( 敏捷宣言 ) · 2

Agile vs. fragile ( 敏捷與脆弱 ) · 88, 122

agilemanifesto.org ( 敏捷宣言網站 agilemanifesto.org ) · 2

Agility ( 敏捷性 ) · 1

definition of ( 敏捷性定義 ) · 4

All-nighter ( 通宵 ) · 165

Andy Hunt's blog ( Andy Hunt的官方博客 ) · 176

Architect ( 架構師 ) , 152

Aristotle ( 亞里士多德 ) , 20, 161

Audit trail ( 審核記錄 ) , 145

Automated testing ( 自動化測試 ) , 90

Avoiding lawsuits ( 避免訴訟 ) , 25

## **B**

Backlog ( 待辦事項 ) , 94, 95

Battle ( 戰役 )

Battle of Somme (WWI) ( 第一次世界大戰，所門戰役 ) , 179

Be the worst ( 成為最差的 ) , 32

Beck, Kent ( Kent Beck ) , 113

Belling the cat ( 給貓系鈴鐺 ) , 23

Best vs. better ( 最好與更好 ) , 20, 22

Big ball of mud ( 大泥球 ) , 137

Big design up front ( 在前期做大量設計 ) , 51

Bill Nye ( Bill Nye , 科普節目主持人 ) , 76

Binary chop ( 二分查找 ) , 138

Blame ( 指責 ) , 12

Blog ( 博客 ) · 158, 169

Blogs ( 多個博客 ) · 29

Bock, David ( David Bock ) · 121

Broken Windows ( 破窗 ) · 3

Brown bag sessions ( 午餐會議 ) · 31

Bug fix ( bug修復 ) · 63

Bugs

inevitability of ( 不可避免的bug ) · 128

Build automation ( 自動構建 ) · 6, 56

Build error ( 構建錯誤 ) · 132

Build machine ( 構建機器 ) · 79

Building construction ( 建築業 ) · 73

Burnout ( 精疲力竭 ) · 42

Business decisions ( 業務決策 ) · 45

Business logic ( 業務邏輯 ) · 90

testing ( 業務邏輯測試 ) · 91

## C

Change ( 變化 )

coping with ( 擁抱變化 ) , 34

Code ( 代碼 )

inspections ( 代碼檢查 ) , 165

procedural vs. object-oriented ( 過程化代碼與面向對象代碼 ) , 121

quality ( 代碼質量 ) , 25

reviews ( 代碼複查 ) , 164

rot ( 腐爛代碼 ) , 165

sharing ( 共享代碼 ) , 163

understanding ( 理解代碼 ) , 100

Coding ( 編碼 )

incrementally ( 增量式編碼 ) , 113

Cohesion ( 內聚 ) , 117

Command-query separation ( 命令與查詢相分離模式 ) , 121

Comments ( 註釋 ) , 106

Commitment ( 承諾 ) , 149

Compiler error ( 編譯錯誤 ) , 132

Compliance ( 符合標準 ) , 13

Composition ( 聚合 ) , 126

Conference room ( 會議室 ) , 150

Conferences ( 會議 ) , 29

Continuous build ( 持續構建 ) , 173

Continuous development ( 持續開發 ) , 58

Continuous integration ( 持續集成 ) , 6, 56, 60, 63, 134, 176

multiple platform ( 多平臺 ) , 88

Contracts, fixed price ( 固定價格合同 ) , 73

Coupling ( 耦合 ) , 104

Courage ( 勇氣 ) , 25

Courtesy ( 禮貌 ) , 19

CRC card design ( CRC卡設計 ) , 50

Crisis management ( 危機管理 ) , 4

Customer demos ( 客戶演示 ) , 68

Customers ( 客戶 )

working with ( 與客戶一起工作 ) , 66

CVS ( CVS ) , 163

## **D**

Data holders ( 數據容器 ) , 123



Daylog ( 日誌 ) , 157

see also Solutions log ( 另見解決方案日誌 )

Deadlines ( 最終期限 ) , 41

setting ( 設定最終期限 ) , 20

Death march ( 死亡之旅 ) , 14

Debug log ( 調試日誌 ) , 140

Debugger ( 調試器 ) , 78, 136

Debugging information ( 調試信息 ) , 144

Decision making ( 制定決策 )

tracking ( 記錄決策制定過程 ) , 131

Decisions ( 決策 )

business ( 業務決策 ) , 45

Delegation ( 委託 ) , 126

Deployment ( 部署 ) , 61

Deprecated methods ( 棄用方法 ) , 135

Design ( 設計 ) , 48

Evolution ( 設計演進 ) , 152

flaws ( 設計缺陷 ) , 81

hand-off ( 設計交付 ) , 49

Is design dead? ( 設計已死 ? ) , 177

Patterns ( 設計模式 ) , 115

using testing for ( 使用測試完成設計 ) , 85

Development methodology ( 開發方法論 )

understanding ( 理解開發方法論 ) , 16

Devil ( 魔鬼 )

about ( 關於魔鬼 ) , 7

Diagnosing problems ( 問題診斷 ) , 37

Differences ( 不同 ) , 87

Docking ( 筆記本電腦底座擴展 ) , 163

Documentation ( 文檔 ) , 81, 92, 105, 108

Don't Repeat Yourself ( “不要重複你自己”原則 ) , 105

Done ( 完成 ) , 3, 4, 41, 93, 163

DRY, see Don't Repeat Yourself ( DRY原則 , 參見“不要重複你自己”原則 )

## **E**

Eisenhower ( 艾森豪威爾 )

U.S. president ( 美國總統艾森豪威爾 ) , 51

Elegance ( 優雅 ) , 116

Emergency project rescue ( 緊急救助項目 ) , 172

Emerging technologies ( 正在湧現的技術 ) , 30

Encapsulation ( 封裝 ) , 121

Environmental problems ( 環境問題 ) , 145

Environmental viscosity ( 環境黏性 ) , 162

Error messages ( 錯誤信息 )

details in ( 詳細錯誤信息 ) , 143

Error reporting ( 錯誤報告 ) , 144

Estimation ( 估算 ) , 74, 93

Evolutionary development ( 演進開發 ) , 69

Exceptions ( 異常 ) , 140

Expertise ( 專家知識 ) , 156

Extreme programming ( 極限編程 ) , 113n, 148, 166

## **F**

Failure, graceful ( 優雅的失敗 ) , 141

Farragut, Admiral David ( 海軍上將David Farragut ) , 24

Feedback ( 反饋 ) · 3, 41, 50, 66, 76, 78, 80, 96, 114

Fire drills ( 消防演習 ) · 40

FIT ( FIT自動驗收測試框架 ) · 90

Fixed-price contracts ( 固定價格合同 ) · 73

Fragile vs. agile, see Agile vs. fragile ( 脆弱與敏捷 · 參見敏捷與脆弱 )

Framework ( 框架 )

justify ( 證明框架可用 ) · 178

Friction ( 摩擦、不和 ) · 3

## **G**

Gang of Four ( GoF · 設計模式“四人幫” ) · 115

Glossary ( 詞彙表 ) · 67

Goal-driven design ( 目標驅動設計 ) · 50

## **H**

Habits ( 習慣 ) · 35

Hacking, crude ( 未經認真考慮的隨意解決方案 ) · 15

Hand-off ( 交付 ) · 49

Hardware vs. software ( 硬件與軟件 )

relative costs ( 硬件與軟件的相對成本 ) , 111

Heraclitus ( 赫拉克利特 , 希臘哲學家 ) , 28

Hippocratic oath ( 希波克拉底的誓言 ) , 60

Hoare, C.A.R. ( C.A.R. Hoare英國著名計算機科學家 ) , 100

Hungarian notation ( 匈牙利表示法 ) , 105

## **I**

Impatience ( 焦急 ) , 25

Incremental ( 增量的 )

development ( 增量開發 ) , 113

Incremental development ( 增量開發 ) , 69

Information radiators ( 信息輻射器 ) , 168, 177

Inheritance ( 繼承 ) , 125

Installer ( 安裝程序 ) , 62, 63

Integration ( 集成 ) , 58

Intellectual capital ( 知識資本 ) , 161

Investment ( 投入 ) , 112

ISO-9001 ( ISO-9001標準 ) , 13

Isolation ( 隔離 ) , 16

Issue tracking ( 問題跟蹤 ) · 68

Iteration ( 迭代 ) · 4, 41, 65, 67

Iterative development ( 迭代開發 ) · 69

## **J**

Javadoc ( Javadoc工具 ) · 107

Jefferson, Thomas ( 托馬斯·傑弗遜 ) · 157

Jones, Capers ( 卡珀斯·瓊斯 ) · 165

JUnit ( JUnit單元測試工具 ) · 78

## **K**

Kitchen timer ( 廚房計時器 ) · 150

Knowledge ( 知識 )

sharing ( 知識分享 ) · 157

Knuth, Donald ( 唐納德·努斯 ) · 112, 153

## **L**

Learning ( 學習 )

and diversity ( 學習和多樣性 ) · 31

iteratively ( 迭代式學習 ) · 29

unit tests as aid ( 用單元測試輔助學習 ) · 81

and unlearning ( 學習與忘記 ) · 35

Lewis and Clark ( 劉易斯與克拉克 ) · 50

Life cycle ( 生命週期 ) · 3

Liskov, Barbara ( Liskov替換原則發明人 ) · 124

List,The ( 列表 ) · 94

Log, see Solutions log ( 日誌 · 參見解決方案日誌 )

Log messages ( 日誌信息 )

in version control ( 版本控制系統中的日誌信息 ) · 163

## **M**

Maintenance costs ( 維護成本 ) · 53

Manager role ( 管理者的角色 )

in meetings ( 管理者在會議中的角色 ) · 149

Mediator ( 仲裁人 ) · 20

Meetings, stand up ( 站立會議 ) · 148

Mentoring ( 指導 ) · 157

Messaging ( 消息發送 ) · 123

Metaphors ( 隱喻 )

using ( 使用隱喻 ) , 31

Methodology ( 方法論 )

understanding ( 理解方法論 ) , 16

Metrics ( 度量指標 ) , 94

Milestone ( 里程碑 ) , 69

Mistakes ( 錯誤 )

making enough ( 犯下足夠的錯誤 ) , 14

Mock objects ( mock對象 ) , 59, 136

Model-View-Controller (MVC) ( MVC設計模式 ) , 119

## **N**

Naming ( 命名 ) , 67, 103, 105

NDoc ( NDoc文檔生成工具 ) , 107

Negativity ( 消極 ) , 19

Next most important thing ( 下一件最重要的事 ) , 94

Noah ( 建造方舟的諾亞 ) , 33

## **O**

Object-oriented vs. procedural code ( 面向對象與面向過程 ) , 121



OO design principles ( 面向對象設計原則 ) , 177

Opaque code ( 意圖不清的代碼 ) , 16

Options ( 選擇 )

providing ( 提供選擇 ) , 46

Outcome ( 產出 ) , 12

Overcomplication ( 過於複雜 ) , 85, 115

## **P**

Pair programming ( 結對編程 ) , 159, 166

Paperboy and the wallet ( 送報紙的男孩和錢包 ) , 121

Pascal, Blaise ( 布萊士·帕斯卡 ) , 109

Patent claim ( 專利聲明 ) , 117

Patterns ( 模式 ) , 119

Pick-up game ( 撿拾遊戲 ) , 166

PIE principle ( 按表達意圖編程原則 ) , 102

Pizza bribes ( 用比薩餅行賄 ) , 33, 42

Plan vs. planning ( 計劃與做計劃 ) , 51

Plan-based projects ( 按計劃行事的項目 ) , 3

Planning ( 做計劃 ) , 43

Politics ( 政治 ) , 19

PowerPoint architects ( 用PowerPoint的架構師 ) , 152

Pragmatic Programmer ( 《程序員修煉之道》 )

web site ( 《程序員修煉之道》網站 ) , 178

Pressure ( 壓力 ) , 37, 45

Privacy ( 隱私 ) , 145

Problem ( 問題 )

diagnosis ( 診斷問題 ) , 129

identi?cation ( 識別問題 ) , 52, 97

isolating ( 隔離問題 ) , 137

solving, chance for ( 解決問題的機會 ) , 161

Procedural vs. object-oriented code ( 過程化與面向對象的代碼 ) , 121

Production environment ( 生產環境 ) , 61

Program defects ( 程序缺陷 ) , 145

Program intently and expressively ( 按表達意圖編程 ) , 102

Progress reporting ( 進度報告 ) , 168

Project automation ( 項目自動化 ) , 56

Project glossary ( 項目詞彙表 ) , 67

Project roles ( 項目職責 ) , 10

Prototype ( 原型 ) , 51, 60, 86, 136

## Q

QA testing ( 質量保證測試 ) , 61

Quick fix ( 快速修復 ) , 15

## R

Résumé-driven-design ( 簡歷驅動的設計 ) , 52

Raising the bar ( 提升標準 ) , 32

RDoc ( RDoc文檔生成工具 ) , 107

Readability ( 可讀性 ) , 100

Refactoring ( 重構 ) , 4, 80, 113

Regression tests ( 迴歸測試 ) , 80

Remote access ( 遠程訪問 ) , 163

Repeatable tests ( 可重複的測試 ) , 79

Requirements ( 需求 )

freezing ( 需求凍結 ) , 64

Responsibilities ( 職責 ) , 120

Retrospectives ( 回顧 ) , 174

Return on investment ( 投資回報 ) , 25

Reuse Release Equivalency ( 重用發佈等價原則 ) , 119

Reversibility ( 可逆轉性 ) , 52, 153

Risk ( 風險 ) , 55

and integration ( 風險與集成 ) , 58

Roles ( 職責 ) , 10

Root cause analysis ( 根本原因分析 ) , 38

RSS feed ( RSS Feed ) , 169

Ruby ( Ruby編程語言 ) , 106

Ruby on Rails ( Ruby on Rails框架 ) , 35

Rut vs. grave ( 車轍與墳墓 ) , 36

## S

Saboteurs ( 怠工分子 ) , 3

Sam Houston ( 薩姆·休斯頓將軍 ) , 55

Santa Anna ( 聖安那將軍 ) , 55

Scrum ( Scrum敏捷方法論 ) , 40, 95

Security concerns ( 安全考慮 ) , 143

Sharing code ( 分享代碼 ) , 163

Sharing learning ( 分享經驗和體會 ) , 31

Side effects ( 副作用 ) , 122

Simple vs. simplistic ( 簡單與簡化 ) , 115

Single Responsibility principle ( 單一職責原則 ) , 119

Slackers ( 懶惰之人 ) , 3

Smalltalk ( Smalltalk編程語言 ) , 123

Snowbird ( 猶他州雪鳥市 ) , 2

Software architect ( 軟件架構師 ) , 152

Solutions log ( 解決方案日誌 ) , 46, 129, 138, 157

Sprint ( Sprint迭代 ) , 40, 95

Stand-up meeting ( 站立會議 ) , 148, 171

Starter Kit ( 入門工具箱 ) , 173

Strategic ( 戰略的 )

decisions ( 戰略決策 ) , 152

design ( 戰略設計 ) , 49

Stub program ( 樁程序 ) , 78

Stupid users ( 愚蠢的用戶 ) , 97

Subclass ( 子類 ) , 124

Subversion ( Subversion版本控制軟件 ) , 163

## T

Tactical ( 戰術的 )

decisions ( 戰術決策 ) , 152

design ( 戰術設計 ) , 49

Teaching ( 講授 ) , 158

Teams ( 團隊 )

size ( 團隊大小 ) , 4

Teamwork ( 團隊精神 ) , 151

Technology stew ( 技術大雜燴 ) , 53

Tell, Don't Ask ( 告知 , 不要詢問 ) , 121

Territorial code ownership ( 代碼“領土”所有權 ) , 155

Test coverage ( 測試覆蓋率 ) , 81

Test Driven Development ( 測試驅動開發 ) , 82

Testing ( 測試 )

user involvement ( 用戶參與測試 ) , 90

Testing frameworks ( 測試框架 ) , 78

Time boxing ( 時間盒 ) , 20, 41, 128

Time sheets ( 時間表 )

problems with ( 時間表的問題 ) , 93

Tools ( 工具 ) , 6

Track issues ( 跟蹤問題 ) , 68

Trade-offs ( 權衡 ) , 54, 111

Transitioning ( 轉換 ) , 36

## U

Unit testing ( 單元測試 ) , 6, 16, 56, 163, 174, 177

automated ( 自動化單元測試 ) , 78

jUnit ( JUnit工具 ) , 177

jUnitPerf ( JunitPerf工具 ) , 177

nUnit ( NUnit工具 ) , 177

Unlearning ( 丟棄 ) , 35

User errors ( 用戶錯誤 ) , 145

User groups ( 用戶組 ) , 29

## V

Version control ( 版本控制 ) · 6, 162, 173

Versioning ( 建立版本 ) · 57

## W

Warnings ( 警告 ) · 132

Waterfall ( 瀑布式開發 ) · 49

Why ( 為什麼 )

benefits of asking ( 問“為什麼”的好處 ) · 38

Wiki ( 維基 ) · 6, 46, 67, 130, 169

Working overtime ( 加班工作 ) · 42

## X

XP, see Extreme programming ( XP · 參見“極限編程” )

xUnit ( 各種語言的單元測試工具 ) · 78

## Y

YAGNI ( “你可能永遠都不需要它”原則 ) · 84

## Z

Zoom-out ( 縮小 ) · 114



