



架构师书榜

# 深入实践DDD

## 以DSL驱动复杂软件开发

杨捷锋 著

DEEP INTO DDD  
Driving Complex Software Development by DSL

领域驱动设计里程碑之作，资深技术专家兼技术管理者二十年工作经验结晶  
深度解读DDD思想，揭示使用DSL实现DDD快速落地的方法与技巧，缓解  
复杂软件开发之痛

机械工业出版社  
China Machine Press

架構師書庫

# 深入實踐DDD：以DSL驅動複雜軟件開發

楊捷鋒 著

ISBN : 978-7-111-67771-0

本書紙版由機械工業出版社於2021年出版，電子版由華章分社（北京華章圖文信息有限公司，北京奧維博世圖書發行有限公司）全球範圍內製作與發行。

版權所有，侵權必究

客服熱線：+ 86-10-68995265

客服信箱：[service@bbbvip.com](mailto:service@bbbvip.com)

官方網址：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)

新浪微博 @華章數媒

微信公眾號 華章電子書（微信號：hzebook）

# 目錄

前言

第一部分 概念

第1章 DDD的關鍵概念

1.1 自頂而下、逐步求精

1.1.1 DDD開創全新分析流派

1.1.2 什麼是軟件的核心複雜性

1.2 什麼是領域模型

1.3 戰術層面的關鍵概念

1.3.1 實體

1.3.2 值對象

1.3.3 聚合與聚合根、聚合內部實體

1.3.4 聚合的整體與局部

1.3.5 聚合是數據修改的單元

1.3.6 聚合分析是“拆分”的基礎

1.3.7 服務

1.4 戰略層面的關鍵概念

1.4.1 限界上下文

1.4.2 限界上下文與微服務

1.4.3 防腐層

1.4.4 統一語言

1.5 ER模型、OO模型和關係模型

1.6 概念建模與模型範式

第2章 其他DDD相關概念

2.1 領域ID

2.1.1 自然鍵與代理鍵

2.1.2 DDD實體的ID需要被最終用戶看到

- 2.1.3 什麼時候使用代理鍵
- 2.2 ID、Local ID與Global ID
- 2.3 命令、事件與狀態
- 第3章 CQRS與Event Sourcing
  - 3.1 命令查詢職責分離
  - 3.2 事件溯源
  - 3.3 From-Thru模式
    - 3.3.1 示例：ProductPrice
    - 3.3.2 示例：PartyRelationship
  - 3.4 CQRS、ES與流處理
- 第二部分 設計
- 第4章 DDD的DSL是什麼
  - 4.1 為什麼DDD需要DSL
    - 4.1.1 為什麼實現DDD那麼難
    - 4.1.2 搞定DDD的“錘子”在哪裡
  - 4.2 需要什麼樣的DSL
    - 4.2.1 在“信仰”上保持中立
    - 4.2.2 DDD原生
    - 4.2.3 在複雜和簡單中平衡
    - 4.2.4 通過DSL重塑軟件開發過程
  - 4.3 DDDML——DDD的DSL
    - 4.3.1 DDDML的詞彙表
    - 4.3.2 DDDML的Schema
  - 4.4 DDDML示例：Car
    - 4.4.1 “對象”的名稱在哪裡
    - 4.4.2 使用兩種命名風格：camelCase 與 PascalCase

## 4.4.3 為何引入關鍵字itemType

### 第5章 限界上下文

5.1 DDDML文檔的根結點下有什麼

5.2 限界上下文的配置

5.3 名稱空間

5.3.1 再談PascalCase命名風格

5.3.2 注意兩個字母的首字母縮寫詞

5.4 關於模塊

### 第6章 值對象

6.1 領域基礎類型

6.1.1 例子：從OFBiz借鑑過來的類型系統

6.1.2 例子：任務的觸發器

6.2 數據值對象

6.3 枚舉對象

### 第7章 聚合與實體

7.1 用同一個結點描述聚合及聚合根

7.2 實體之間只有一種基本關係

7.3 關於實體的ID

7.4 不變的實體

7.5 動態對象

7.6 繼承與多態

7.6.1 使用關鍵字inheritedFrom

7.6.2 超對象

7.7 引用

7.7.1 定義實體的引用

7.7.2 屬性的類型與引用類型

7.8 基本屬性與派生屬性

7.8.1 類型為實體集合的派生屬性

7.8.2 類型為值對象的派生屬性

## 7.9 約束

7.9.1 在實體層面的約束

7.9.2 在屬性層面的約束

## 7.10 提供擴展點

# 第8章 超越數據模型

## 8.1 實體的方法

8.1.1 聚合根的方法

8.1.2 非聚合根實體的方法

8.1.3 屬性的命令

8.1.4 命令ID與請求者ID

## 8.2 記錄業務邏輯

8.2.1 關於accountingQuantityTypes

8.2.2 關於derivationLogic

8.2.3 關於filter

8.2.4 使用關鍵字referenceFilter

8.2.5 業務邏輯代碼中的變量

8.2.6 說說區塊鏈

## 8.3 領域服務

## 8.4 在方法定義中使用關鍵字inheritedFrom

## 8.5 方法的安全性

# 第9章 模式

## 9.1 賬務模式

## 9.2 狀態機模式

## 9.3 樹結構模式

9.3.1 簡單的樹

### 9.3.2 使用關鍵字structureType

### 9.3.3 使用關鍵字structureTypeFilter

## 第三部分 實踐

### 第10章 處理限界上下文與值對象

#### 10.1 項目文件

#### 10.2 處理值對象

##### 10.2.1 一個需要處理的數據值對象示例

##### 10.2.2 使用Hibernate存儲數據值對象

##### 10.2.3 處理值對象的集合

##### 10.2.4 在URL中使用數據值對象

##### 10.2.5 處理領域基礎類型

### 第11章 處理聚合與實體

#### 11.1 生成聚合的代碼

##### 11.1.1 接口

##### 11.1.2 代碼中的命名問題

##### 11.1.3 接口的實現

##### 11.1.4 事件存儲與持久化

##### 11.1.5 使用Validation框架

##### 11.1.6 保證靜態方法與模型同步更新

##### 11.1.7 不使用事件溯源

#### 11.2 Override聚合對象的方法

#### 11.3 處理繼承

##### 11.3.1 TPCH

##### 11.3.2 TPCC

##### 11.3.3 TPS

#### 11.4 處理模式

##### 11.4.1 處理賬務模式

## 11.4.2 處理狀態機模式

# 第12章 處理領域服務

## 12.1 處理數據的一致性

### 12.1.1 使用數據庫事務實現一致性

### 12.1.2 使用Saga實現最終一致性

## 12.2 發佈與處理領域事件

### 12.2.1 編寫DDDML文檔

### 12.2.2 生成的事件發佈代碼

### 12.2.3 編寫生產端聚合的業務邏輯

### 12.2.4 實現消費端領域事件的處理

## 12.3 支持基於編制的Saga

### 12.3.1 編寫DDDML文檔

### 12.3.2 生成的Saga命令處理代碼

### 12.3.3 需要我們編寫的Saga代碼

### 12.3.4 需要我們實現的實體方法

# 第13章 RESTful API

## 13.1 RESTful API的最佳實踐

### 13.1.1 沒有必要絞盡腦汁地尋找名詞

### 13.1.2 儘可能使用HTTP作為封包

### 13.1.3 異常處理

## 13.2 聚合的RESTful API

### 13.2.1 GET

### 13.2.2 PUT

### 13.2.3 PATCH

### 13.2.4 DELETE

### 13.2.5 POST

### 13.2.6 事件溯源API

### 13.2.7 樹的查詢接口

## 13.3 服務的RESTful API

### 13.4 身份與訪問管理

#### 13.4.1 獲取OAuth 2.0 Bearer Token

#### 13.4.2 在資源服務器上處理授權

### 13.5 生成Client SDK

#### 13.5.1 創建聚合實例

#### 13.5.2 更新聚合實例

#### 13.5.3 使用Retrofit2

## 第14章 直達UI

### 14.1 兩條路線的鬥爭

#### 14.1.1 前端“知道”領域模型

#### 14.1.2 前端“只知道”RESTful API

### 14.2 生成Admin UI

#### 14.2.1 使用referenceFilter

#### 14.2.2 展示派生的實體集合屬性

#### 14.2.3 使用屬性層面的約束

#### 14.2.4 使用UI層元數據

#### 14.2.5 構建更實時的應用

## 第四部分 建模漫談與DDD隨想

## 第15章 找回敏捷的軟件設計

### 15.1 重構不是萬能靈藥

### 15.2 數據建模示例：訂單的裝運與支付

#### 15.2.1 訂單與訂單行項

#### 15.2.2 訂單與訂單裝運組

#### 15.2.3 訂單與裝運單

#### 15.2.4 訂單的項目發貨

- 15.2.5 訂單的支付
  - 15.3 中臺是一個輪迴
  - 15.4 實例化需求與行為驅動測試
    - 15.4.1 什麼是實例化需求
    - 15.4.2 BDD工具
    - 15.4.3 BDD工具應與DDD相得益彰
    - 15.4.4 不要在驗收測試中使用固件數據
    - 15.4.5 製造“製造數據”的工具
  - 15.5 要領域模型驅動，不要UI驅動
  - 15.6 不要用“我”的視角設計核心模型
    - 15.6.1 讓User消失
    - 15.6.2 認識一下Party
  - 15.7 我們想要的敏捷設計
- 第16章 說說SaaS
- 16.1 何為SaaS
  - 16.2 多租戶技術
  - 16.3 構建成功的SaaS有何難
    - 16.3.1 多租戶系統的構建成本
    - 16.3.2 難以滿足的定製化需求
    - 16.3.3 負重前行的傳統軟件公司
  - 16.4 SaaS需要DDD
- 第17章 更好的“錘子”
- 17.1 我們製作的一個DDDMIL GUI工具
    - 17.1.1 紿領域建模提供起點
    - 17.1.2 創建新的限界上下文
    - 17.1.3 從OFBiz中“借鑑”數據模型
    - 17.1.4 構建項目並運行應用

17.1.5 使用HTTP PUT方法創建實體

17.1.6 級聚合增加方法

17.1.7 生成限界上下文的Demo Admin UI

17.1.8 讓不同層級的開發人員各盡其能

17.2 以統一語言建模

附錄 DDDML示例與縮寫表

# 前言

## 為什麼要寫這本書

2004年，DDD（領域驅動設計）這一軟件開發的方法與願景經由建模專家Eric Evans的經典著作 *Domain-Driven Design: Tackling Complexity in the Heart of Software* 正式面世，當即獲得了廣泛關注和高度評價。16年過去了，我在網上看到越來越多關於 DDD的文章和討論。為什麼我們現在還不停地討論 DDD？為什麼DDD仍然如此重要？

在商業組織中，主張“技術為業務服務”的企業總可以在理論上立於不敗之地。誠然，DDD主張在軟件項目中把領域本身作為關注的焦點（換句話說就是技術人員要懂業務）符合這種思想，但真正難能可貴的是，DDD提供了切實可行的應對軟件核心複雜性的方法。

實踐證明，DDD提出的方法不僅行之有效，而且歷久彌新。關於這一點，我想從當今IT業界的熱詞“雲原生”“中臺”“產業互聯網”說起。

什麼是雲原生？雲原生計算基金會（Cloud Native Computing Foundation，CNCF）[\[1\]](#)對雲原生的定義是：

雲原生技術有利於各組織在公有云、私有云和混合雲等新型動態環境中構建和運行可彈性擴展的應用。雲原生的代表技術包括容器、服務網格、微服務、不可變基礎設施和聲明式API。

這些技術能夠構建容錯性好、易於管理和便於觀察的松耦合系統。結合可靠的自動化手段，雲原生技術使工程師能夠輕鬆地對系統做出頻繁和可預測的重大變更。

而阿里雲發佈的《雲原生架構白皮書》[\[2\]](#)對雲原生架構的定義是：

從技術的角度看，雲原生架構是基於雲原生技術的一組架構原則和設計模式的集合，旨在將雲應用中的非業務代碼部分進行最大化剝離，從而讓雲設施接管應用中原有的大量非功能特性（如彈性、韌性、安全、可觀測性、灰度等），使業務不再有非功能性業務中斷困擾的同時，具備輕量、敏捷、高度自動化的特點。

看了這些定義，你是否還是覺得迷惑？打開CNCF的“landscape”頁面[\[3\]](#)——裡面有很多的項目和成員，難怪有人說雲原生是一個“營銷詞語”。在這個頁面中，在Members（成員）這個標籤頁的左邊，Serverless獨佔了一個標籤頁，十分顯眼。

廣泛認同的Serverless架構是指這樣的應用設計：與第三方的後端即服務（Backend as a Service，BaaS）交互；在函數即服務（Function as a Service，FaaS）平臺上運行函數式業務代碼，一般來說，它們是在受管理的、臨時性的容器中執行的。

雖然新出現的基於容器的Serverless平臺，比如 Knative<sup>[4]</sup>，可以運行開發人員使用傳統方式開發的應用，但FaaS仍然是Serverless中最重要、最具代表性的產品形態，因為它讓我們以一種不同於傳統的方式思考技術架構。

FaaS中的“Function”就是不依賴特定框架和類庫的最純粹的業務代碼——這是真正為業務帶來價值的東西。可以說，FaaS在將雲應用中的非業務代碼部分進行最大化剝離方面做到了極致。我認為它是雲原生“皇冠上的明珠”。

不過，大家普遍認為當前FaaS更適合開發事件驅動風格的、處理少數幾個事件類型的應用組件，而不適合開發傳統的具有很多入口的同步請求/響應風格的應用組件。也就是說，如果你想問：“能不能基於FaaS做出一個SAP ERP？”目前可能大多數人給你的答案會是“NO”。

但是我想給你的答案是“YES”！因為要想達到這個目標，我們需要克服的所有障礙都不屬於FaaS的固有

缺點，而是當前FaaS的實現缺陷，比如啟動延遲、集成測試、調試、交付、監控與觀測等方面的問題。

我認為，以DDD方法實現的應用可以極大地降低FaaS處理這些問題的難度，甚至可以直接忽視某些問題（因為它們對以DDD方法實現的應用來說不是問題）。具體而言，我們可以使用DDD的聚合概念來切分應用組件，每個聚合一個小組件，它們可以很快地被FaaS平臺“拉起”。這些高度內聚的小組件是更復雜的應用組件（比如說領域服務）的構造塊。我們可以使用興起於DDD社區的Event Sourcing（事件溯源，ES）模式，保證應用狀態的每一次變更都會發布領域事件，並以富含業務語義的事件驅動其他應用組件運行。比如，命令查詢職責分離（Command Query Responsibility Segregation，CQRS）模式中的Denormalizer（去規範化器）組件就可以訂閱、消費這些事件，為前端應用構建友好的查詢視圖——這些都是DDD社區在開發嚴肅的商業軟件時一直在做的事情。如果之前你沒有接觸過聚合、ES、CQRS，也許難以理解上面所說的內容，不過沒關係，讀完本書，我相信你就清楚了。

再說“中臺”這個熱詞。以我的理解，中臺是將可複用的代碼抽取到一個平臺中，作為大家共用的軟件組件，它是服務於前臺的規模化創新。中臺（這裡主要指業務中臺）想要好用，必須具備“反映對領域的深度認知”的軟件模型，甚至在某種程度上需要“過度設計”，並且絕對有必要維護良好的概念完整性，構建所

謂的企業級業務架構——這些都是DDD可以大展身手的地方。

關於“產業互聯網即將進入黃金時代”的說法，大多是眾多傳統企業希望借力最新的信息化，特別是互聯網工具（即所謂“互聯網+”），提升內部效率和對外服務的能力。傳統產業中的很多領域概念和業務流程並不一定為普通的開發者所熟知——這與“消費互聯網”不同，顯然人人都是消費者。所以，傳統產業的信息化急需可以快速梳理並深刻地認知領域，以及能構建高質量領域模型的技術人才。DDD可以說是技術人員升職加薪的“神兵利器”。

我在工作中看到的情況是，越來越多的技術人員在自己的求職簡歷上寫上了“熟悉（或精通）DDD”的描述。確實，Eric Evans的經典著作以抽象、凝練著稱，可謂字字珠璣，甚至很多資深技術人員都不能領悟其中玄妙。所以，我也認同掌握DDD是一件足以讓技術人員引以為傲的事情。

可以說，DDD是公認的解決軟件核心複雜性的“大殺器”。但是，在軟件開發中實踐DDD是需要付出相當大的成本的。也就是說，大家的普遍看法是：實踐DDD是一個先苦後甜的過程，一個項目要不要採用DDD，最好先看看它值不值得。

但是一個項目值不值得使用DDD有時不好判斷。大項目往往是由小項目發展而來的，很多從小項目演

化而來的大系統最終變成開發團隊的噩夢，噩夢的根源幾乎無一例外地在於軟件的概念完整性遭到了破壞。而DDD正是維護軟件概念完整性的良藥。如果在項目中實踐DDD的成本不高，那麼即使是小項目，從一開始就使用DDD不是一件很美好的事情嗎？

在軟件開發項目中實踐DDD到底有何難處？16年了，難道在DDD實踐中碰到的問題我們不能從書本中尋得答案？

目前國內已經出版的DDD相關圖書中，除了Eric Evans的經典著作之外，還有《實現領域驅動設計》《領域驅動設計精粹》《領域驅動設計模式、原理與實踐》等，不過寥寥數本。這與DDD的巨大聲望很不匹配，這也許說明了一些問題。

實踐DDD首先需要面對的一個（也許是最大的）難題是：難以描述的領域模型。

DDD想要構建的領域模型是什麼？按照Eric Evans的觀點，領域模型不是一幅具體的圖，而是那幅圖想要傳達的思想；不是一個領域專家頭腦中的知識，而是那些經過嚴格組織並進行選擇性抽象的知識。

聽起來是不是有點玄奧？系統分析師、產品經理到底要拿出什麼樣的領域模型才能說“我的工作已經做到位了”？這個模型到底是不是可以實現的？開發人

員、測試人員到底有沒有理解這個模型？大家的理解是不是一致的？

說到底，一個領域模型要想有用，它必須足夠嚴格。如何使用一種嚴格的方式描述經過嚴格組織並進行選擇性抽象的知識呢？

問題的答案很自然地指向了DSL。

其實，Eric Evans早就意識到了這一點。他曾經在訪[\[5\]](#)談中說：

更多前沿的話題發生在領域專用語言（DSL）領域，我一直深信DSL會是領域驅動設計發展的下一步。現在，還沒有一個工具可以真正給我們想要的東西。但是人們在這一領域比過去了做了更多的實驗，這使我對未來充滿了希望。

通過本書，我想要告訴大家的是：在項目中運用DDD可以不像大家想象的那麼痛苦，DDD並不是只適用於大項目，使用DDD並不一定需要犧牲敏捷性，一切的關鍵在於DSL的運用。

我和我工作過的團隊曾經在多個項目中使用DSL實現了DDD的真正落地。獨樂樂不如眾樂樂！現在，我想把這些實踐經驗分享給大家。

讀者對象

領域模型是一種“思想”，它可以為軟件開發的全過程提供指導，所以我相信本書可以為很多人提供幫助：

- 產品經理與系統分析師。產品經理不僅僅需要保證團隊開發的是“正確的軟件”，也不應該只是關注軟件的界面原型、用戶體驗，更應該讓軟件有內涵。迷人的產品不僅需要漂亮的界面和交互，更需要邏輯自洽，功能處處傳達出一個思想——優美而深刻的領域模型。在有的團隊中，產品經理同時也是系統分析師。作為近十幾年來最有影響力的軟件分析和設計的方法論，系統分析師有必要了解DDD，從中汲取營養。
- 架構師。架構師需要根據業務需求提供技術解決方案。DDD想要構建的領域模型不僅僅是領域業務知識的提煉總結，也包含了對軟件設計的考量，可以用於直接指導軟件的編碼實現。構建這樣的模型，需要架構師的積極參與。
- 開發工程師。開發工程師應該理解領域模型，領域模型應該被忠實地映射到代碼實現中。代碼中對象的命名、對象之間的關係，都應該與領域模型一致。唯有如此，代碼才能具備良好的可讀性、可維護性，敏捷XP方法的狂熱愛好者們所言的“代碼即文檔”才可能實現。

·測試工程師。如今很多測試工程師已經被稱為“測試開發工程師”，他們也應該理解領域模型。測試工程師應該像產品經理一樣瞭解軟件的設計，甚至應該比產品經理更深刻地理解領域模型面向軟件設計所做的考量。實例化需求的自動化測試（或者說行為驅動測試）應該基於領域模型，而非基於UI/UE來構建，因為用戶界面以及用戶交互是易變的，而領域模型相對來說穩定得多。

·項目經理。項目經理是負責“正確開發軟件”的人。如果不深刻理解領域，不知道如何抓住領域模型中的關鍵點，會很難評估任務工作量的大小以及應該在何處投入足夠的資源，甚至無法判斷項目的實際進度。

·高校研究生以及其他有志於從事IT行業的人。

## 如何閱讀本書

本書的第一部分會帶領讀者從戰術層面以及戰略層面重溫領域驅動設計的重要概念，然後進一步闡述Eric Evans經典著作中沒有顯式提出的或者被太多人忽略的但我認為對DDD落地非常重要的若干概念，同時簡要介紹從DDD社區興起的一些軟件架構模式。通過第一部分，讀者可以更完整、更深刻地掌握DDD的知識體系。

第二部分闡述如何設計一種DDD的DSL，包括這個DSL的規範（Specification）支持哪些特性、如何幫助團隊描述領域模型的方方面面、這些特性的選擇基於何種考量等。

這種領域專用語言需要一個名字，我們總不能一直說“我設計的DDD的DSL”吧，於是我就給它起了一個名字：DDDM。我認為這是一個很棒的名字。其實這種語言叫什麼並不太重要，重要的是它可以用一種足夠嚴格的方式描述領域模型。我認為目前它在簡單與複雜之間取得了不錯的平衡。當然，其中還有不小改進的空間。比如，我很樂意讓它支持更多像“賬務模式”這樣的分析模式。

第三部分介紹如何將“思想照進實現”——通過使用工具將描述領域模型的DSL文檔變成可以運行的軟件。這個過程涉及大量的技術工具（工具鏈）的設計與實現。只有將這些技術工具——比如從DSL自動生成應用的源代碼的模板——實現出來，才能減輕開發人員實踐DDD的負擔，進而提升而不是降低軟件團隊的生產效率。本部分會介紹這些技術工具設計與實現的細節。

我和我的同事把自制的DDDM工具鏈稱為DDDM Tools。出於商業原因，我無法展示這些工具的源代碼，但是會詳盡地展示這些工具運行的結果——主要是由工具生成的應用的源代碼。這些源代碼可能經過

一些簡化，但是與我們在生產系統上運行的代碼十分接近，完全可以說明問題。

讀完全書，你將發現其實我們已經全無秘密。你會熟知**DDDM**的規範，見到工具運行的結果，你幾乎可以馬上動手製造自己的**DDDM**工具。其實設計**DDDM**的規範才是整件事情（使用**DSL**實現領域驅動設計）中最難的部分，製作工具不是。雖然想要復刻我們已經做過的所有工具確實需要相當大的工作量，但也僅僅是工作量而已。

幸運的是，你並不需要製造整條**DDDM**的工具鏈才可以享受使用**DSL**的樂趣。比如，你可以先寫一些模板，生成一些持久對象（**Persistant Object**），它們無非是一些簡單的**Java**對象（**Plain Ordinary Java Object**，**POJO**），然後再生成一些**O/R Mapping XML**，就可以馬上使用**JPA/Hibernate**來實現應用的**DAL**（數據訪問層）了。如果你再生成**Repository**，那就更漂亮啦！

第四部分講述的是一些建模案例以及其他與**DDD**相關的話題。領域模型是一種思想，**DSL**是一種工具，但是如何運用、結果如何，因人而異。我希望通過輕鬆的漫談和隨想，將我的一點**DDD**應用經驗分享給大家。

[勘誤和支持](#)

由於作者水平有限，編寫時間倉促，書中難免會出現一些錯誤或者不準確的地方，懇請讀者批評指正。為此，我特意創建了一個微博賬號“@領域驅動設計”，不管你遇到什麼問題，都可以到該微博賬號上諮詢，我將盡量在線上為讀者提供滿意的解答。本書的勘誤信息也會在這個微博賬號發佈。如果你有更多寶貴的意見，歡迎發送郵件至我的郵箱 [lingyuqudong@163.com](mailto:lingyuqudong@163.com)，很期待聽到你們的真摯反饋。

## 致謝

首先要感謝Eric Evans，是他將偉大的DDD帶到世間。

感謝我的朋友何李石，他在百忙之中抽出時間審閱了本書的第一部分和第二部分，並提出了寶貴的修改意見。感謝我的摯友李永春，他仔細通讀了全書，為我挑出了許多錯別字、病句以及行文不流暢之處，當然更提了不少內容上的改進建議。

感謝機械工業出版社華章公司的編輯楊繡國老師，感謝她的魄力和遠見，並在這一年多的時間中始終支持我寫作，是她的鼓勵和幫助引導我順利完成了全部書稿。

特別要感謝我的愛人張麗君，感謝她的陪伴和支持。

楊捷鋒

於上海

[1]

見

<https://github.com/cncf/toc/blob/master/DEFINITION.md>。

[2]

見 <https://developer.aliyun.com/topic/cn-architecture-paper>。

[3]

見 <https://landscape.cncf.io/>。

[4]

見 <https://knative.dev>。

[5]

Eric Evans on why DDD Matters Today,  
<https://www.infoq.com/articles/eric-evans-ddd-matters-today/>。

# 第一部分 概念

- 第1章 DDD的關鍵概念
- 第2章 其他DDD相關概念
- 第3章 CQRS與Event Sourcing

# 第1章 DDD的關鍵概念

開發大型軟件最難的部分並不是實現，而是要深刻理解它所服務的現實世界的領域。領域驅動設計 ( Domain-Driven Design，DDD ) 是一種處理高度複雜領域的願景 ( Vision ) 和方法，它主張在軟件項目中把領域本身作為關注的焦點，維護一個對領域有深度認知的軟件模型。這個願景和方法，經由建模專家Eric Evans於2004年出版的其最具影響力的著名圖書 *Domain-Driven Design: Tackling Complexity in the Heart of Software*<sup>[1]</sup> ( 簡稱DDD ) 正式面世。

 提示部分開發人員是不是覺得這段話不太好理解？

也許大家可以先思考一個問題：為什麼命名是程序員公認的軟件開發中最艱鉅的任務？Quora網站曾進行“最挑戰程序員的任務”的投票，半數程序員認為最難的事情是“*Naming things*”。排在第二位的是“*Explaining what I do(or don't do)*”，其得票數大約只是前者的一半。

其實命名的困難只是表象，構建領域模型的困難才是根本。比如說，“對象”的名字代表著它的職責，你只有把一個對象應該幹什麼想清楚了，才能給它起

一個恰當的名字。“對象應該幹什麼”，這是一個領域建模問題。

我們可以在網上免費獲取該書的縮寫精簡版 *Domain-Driven Design Quickly*<sup>[2]</sup>，在InfoQ中文站中，有這個英文精簡版對應的中文翻譯版本<sup>[3]</sup>。

關於DDD的關鍵概念，還可以參考維基百科的相關詞條<sup>[4]</sup>。

領域驅動設計（DDD），其實就是以領域模型驅動軟件設計。要理解DDD，關鍵是理解什麼是DDD所指的領域模型，但在此之前，還是應該先認識一下軟件開發的過程。然後，基於此認識重溫一下DDD在戰術以及戰略層面的若干關鍵概念。

本章的最後會簡單探討一下軟件開發團隊經常接觸到的幾種模型範式，幫助理解DDD模型和它們的關係。

[1] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003. 見

<https://www.amazon.com/exec/obidos/ASIN/0321125215/domainlanguag-20>。

[2] Floyd Marinescu & Abel Avram. *Domain-Driven Design Quickly*. C4Media, 2007. 見

<https://www.infoq.com/minibooks/domain-driven-design-quickly>。

[3] 領域驅動設計精簡版（全新修訂），  
<https://www.infoq.cn/article/domain-driven-design-quickly-new>。

[4] Domain-Driven Design,  
[https://en.wikipedia.org/wiki/Domain-driven\\_design](https://en.wikipedia.org/wiki/Domain-driven_design)。

## 1.1 自頂而下、逐步求精

我第一次接觸計算機程序設計是在上大學的時候，當時學習的第一門編程語言是Fortran。記得就是在這門課上，老師告訴我們，開發程序應該“自頂而下、逐步求精”。也就是說，不要求一步就編寫出可執行的程序，我們可以面向問題的總體目標，抽象低層的細節，先構造程序的高層結構，然後再一層一層地向下分解和細化，最後一步編寫出來的程序才是可執行程序。這實際上是一種分而治之的思想。

打個比喻，寫程序應該像建造大樓一樣，一開始要有設計任務書（甲方的需求），然後設計單位根據任務書做概念設計、方案設計、初步設計，可能還要進行所謂的擴大初步設計，之後是施工圖設計，最後施工單位拿著施工圖紙才能造出真正的建築。

當時我就有醍醐灌頂之感，“自頂而下、逐步求精”，何其妙哉！

後來我才知道，這句話其實來自瑞士計算機科學家尼古拉斯·沃斯在1971年發表的論文《通過逐步求精方式開發程序》（Program Development by Stepwise Refinement）。這篇論文首次提出了結構化程序設計（Structure Programming）的概念。1983年1月，ACM在紀念Communications of the ACM創刊25週年

時，從其1/4個世紀發表的論文中評選出了其中具有“里程碑意義的研究論文”25篇（每年一篇），沃斯的這篇論文就是其中之一。

### 1.1.1 DDD開創全新分析流派

雖然到了20世紀80年代，面向對象的程序設計方法已經開始大行其道，但是自頂而下、逐步求精的程序開發方法並沒有過時。就像建築材料再怎麼變化，想要建造一個偉大的建築，需求分析、概念設計、方案設計、施工圖設計這些工作恐怕難以避免。那麼，DDD做的到底是哪個階段的事情呢？筆者認為DDD的領域建模階段大致相當於建築設計的需求分析與概念設計階段。

相信任何經歷過大型軟件開發項目的人都會贊成，想要提升軟件開發效率和軟件質量、避免返工浪費，關鍵是要有高質量的分析。高質量的分析是需要有方法論的。軟件的分析也是一個“領域”，那麼基於構建一個“關於分析的模型”的方法，就產生了不同的流派。可以說，Eric Evans的DDD開創了一個全新的分析流派。那麼，這個流派有什麼特點呢？

首先，DDD是一種面向對象分析（Object-Oriented Analysis，OOA）與設計的方法論，可以很好地與現代的面向對象的程序設計（Object-Oriented Programming，OOP）方法相結合，實現軟件的編程方法會反過來影響分析方法。DDD所使用的術語，如對象（DDD將對象分為實體/引用對象、值對象）、屬性等，瞭解OOP的開發人員會感覺很熟悉。

其次，DDD拋棄了分裂業務分析與軟件設計的做法，使用單一的領域模型來同時滿足這兩方面的要求。可以說，DDD致力於構造一個易於編程實現（特別是使用OO語言）的概念模型。



提示這裡說DDD是OOA的一種，可能會引起爭議。有人也許會堅持認為DDD只是面向對象設計（Object-Oriented Design，OOD）。但是，既然DDD是反對分裂業務分析與軟件設計的，那麼，我們就可以說DDD既是OOA也是OOD。我甚至認為，DDD從誕生之日起就是最有分量的OOA方法論。

### 1.1.2 什麼是軟件的核心複雜性

我們知道，Domain-Driven Design: Tackling Complexity in the Heart of Software一書的副標題翻譯過來是“軟件核心複雜性應對之道”，那麼，這個軟件的核心複雜性（Complexity）是怎麼來的呢？

Complexity有繁雜之意。它的解釋之一是：因為存在很多相互關聯的部分而導致的狀況。

對於大多數軟件系統，它們的核心複雜性是由其服務的領域涉及的範圍帶來的。比如說，隨著領域內的名詞概念增多，概念以及概念之間發生關係的可能性也會呈幾何級數增長，於是我們想要全面地理解領域就會變得越來越困難。

所以，解決核心複雜性的關鍵還是在於切分（分而治之），也就是說希望可以縮減每次要解決的領域問題的範圍，簡化概念和概念之間的關係。DDD正是這麼做的。

那麼，DDD要構建的領域模型到底是什麼東西？它有哪些關鍵概念可以幫助我們實現“分而治之”呢？

## 1.2 什麼是領域模型

本書在提到領域模型的時候，一般特指DDD的領域模型，也就是Eric Evans在Domain-Driven Design: Tackling Complexity in the Heart of Software一書中闡述的那種領域模型。DDD要求領域模型既能反映對領域的深度認知，又能直接用於指導軟件的設計與實現。

我們可以說領域模型是系統化的領域知識。不系統化的知識是難以傳授、掌握和應用的。想象一下，一個會計專業的學生如果不去學習《會計基礎》課本，直接上崗記賬會出現什麼樣的狀況，恐怕會記得一塌糊塗吧？

為了將領域知識系統化，我們需要做領域分析，而分析得到的結果是一個體現我們對領域認知的概念模型。

那麼，我們做領域分析的時候，是不是可以只專注於做好業務領域的分析，構建一個只是反映業務知識的分析模型呢？《領域驅動設計精簡版》第13頁中對這個問題有論述：

一個推薦的設計技術是創建分析模型，它被認為是與代碼設計相互分離、通常是由不同的人完成的。分析模型是業務領域分析的結果，此模型不需要考慮

軟件如何實現。這樣的一個模型可用來理解領域，它建立了特定級別的知識，模型在分析層面是正確的。軟件實現不是這個階段要考慮的，因為這會被看作一個導致混亂的因素。這個模型到達開發人員那裡後，由他們來做設計的工作。因為這個模型中沒有包含設計原則，它可能無法很好地為目標服務。因此開發人員不得不修改它，或者創建分離的設計，在模型和代碼之間也不再存在映射關係，最終的結果是分析模型在編碼開始後就被拋棄了。

為什麼“分析模型在編碼開始後就被拋棄”是一個大問題？為什麼領域模型與代碼之間應該存在映射關係？

是時候重溫這句經典名言了：

程序是寫來給人讀的，只會偶爾讓機器執行一下。

——Abelson和Sussman

與建築不同，軟件具有易變性，在軟件開發出來之後，可能還需要經常修改。只有反映高層結構的代碼才是易於閱讀和理解的。我們都知道，開發人員閱讀代碼的時間遠遠多於編寫代碼的時間。Google的工程師每人每天大約只產出100行新代碼！

所以，Evans給出了他認為的更好的分析建模——創建軟件高層結構——的建議：

一種更好的方法是將領域建模和設計緊密關聯起來。模型在構建時就考慮到軟件的實現和設計。開發人員應該被加入到建模的過程中。主要的想法是選擇一個能夠在軟件實現中恰當地表達的模型，這樣設計過程會很順暢並且是基於模型的。將代碼與其所基於的模型緊密關聯，將會使得代碼更有意義，並且與模型保持相關。

——《領域驅動設計精簡版》，第14頁

那麼，DDD的領域模型到底是什麼？還是聽聽Eric Evans的“官方”說法吧：

領域模型不是一幅具體的圖，它是那幅圖想要去傳達的那個思想。它也不是一個領域專家頭腦中的知識，而是一個經過嚴格組織並進行選擇性抽象的知識。一幅圖能夠描繪和傳達一個模型，同樣，經過精心編寫的代碼和一段英語句子都能達到這個目的。

——《領域驅動設計精簡版》，第3頁

## 1.3 戰術層面的關鍵概念

由於本書並不是DDD的入門書，所以未對DDD在戰術層面的一些關鍵概念進行詳細解釋，讀者可以查閱DDD相關圖書或網上文章進行了解。比如《領域驅動設計精簡版》在第3章中詳細介紹了模型驅動設計的基本構成要素，也就是DDD戰術層面的關鍵概念。

本書只結合筆者自身的理解，強調一些筆者認為最重要的概念。

### 1.3.1 實體

有一類對象擁有標識符（簡稱ID），不管對象的狀態如何變化，它的ID總是不變的，這樣的對象稱為實體。

比如說，我們的銀行賬戶（Account）總是有一個編號（賬號）。我們存錢、取錢時賬戶裡面的錢會發生變化，但是賬號不變。我們通過這個賬號，能夠查詢賬戶的餘額。所以，我們可以把銀行賬戶建模為一個實體，選擇它的編號作為這個實體的ID。

對於很多開發人員來說，實體是他們非常熟悉的概念。特別是對於使用過ORM<sup>[1]</sup>框架的開發人員來說，當你提到“實體”，他們可能馬上就會想到“就是需要映射為數據表（Table）的那些對象”。

[1] Object-relational Mapping (ORM),  
[http://en.wikipedia.org/wiki/Object-relational\\_mapping](http://en.wikipedia.org/wiki/Object-relational_mapping)。

### 1.3.2 值對象

下面這幾句話是從Domain-Driven Design: Tackling Complexity in the Heart of Software一書中摘錄的（有少量改寫），是筆者認為理解DDD值對象的關鍵點：

- 用來描述領域的特定方面的、沒有標識符的對象，叫作值對象。
- 忽略其他類型的對象（如Service、Repository、Factory等），假設對象只有實體和值對象兩種，若將那些符合實體定義的對象作為實體，那麼剩下的對象就是值對象。
- 推薦將值對象實現為“不可變的”（**Immutable**）。也就是說，值對象由一個構造器創建，並且在它們的生命週期內永遠不會被修改。實現為不可變的，並且不具有標識符後，值對象就能夠被安全地共享，並且能維持一致性。

對於程序員來說，還可以這麼理解：如果你熟悉的語言中存在所謂的基本類型（Primitive Type），那麼一般來說，它們都可以被理解為值對象。比如Java中的byte、int、long、float、boolean等，都是值對象。

Java基礎類庫的很多類都是值對象。比如對金額進行計算時，在Java代碼中一般會使用**BigDecimal**來實現，而**BigDecimal**就是一個表示大十進制數的值對象。

**BigDecimal**的實現使用了兩個關鍵的私有字段（**private field**），其中字段**intVal**的類型是**BigInteger**，表示“大整數”，另外一個字段**scale**的類型是**int**（整數），用來表示小數的位數。比如，一個**BigDecimal**對象字段**intVal**的值為**10011**，字段**scale**的值為**2**，那麼這個**BigDecimal**對象表示的就是“**100.11**”。

**BigInteger**的實現主要依賴一個**int[]**（整數的數組）類型的字段，通過使用這個數組可以表示任意大整數。**BigInteger**也是一個值對象。

如果我們使用Java語言實現一個表示“錢”的**Money**值對象，內部可以用一個私有的**BigDecimal**類型的字段來表示金額，然後用一個**String**類型的字段來表示貨幣單位的縮寫。它的構造器可能像這樣：

---

```
public Money(BigDecimal amount, String currency);
```

---

當我們需要“人民幣**10**元錢”的時候，可以按如下方式實例化一個對象：

```
Money tenCny = new Money(BigDecimal.valueOf(10), "CNY");
```

---

一旦這筆錢構造出來，就是一個不可變的整體。**Money**的加、減、乘、除，以及（按匯率）轉換為以另一種貨幣單位表示的“錢”，都需要調用相應的方法來完成，這些方法返回的結果會是一個新的不可變的**Money**對象。

在Java開源社區有一個**Joda Money**<sup>[1]</sup>類庫，其中**Money**類的實現就類似上面描述的形式。

我們應該儘可能地多用不可變的對象，而不是隨意地為每個字段創建相應的**getter/setter**方法（也就是可讀可寫的屬性）。

其實在很多領域中都需要這樣的封裝。舉例來說，我們可能希望在開發應用時按如下方式使用值對象：

- 可以聲明一個屬性的類型是**Email**，而不是**String**。
- 可以聲明一個屬性的類型是手機號（**MobileNumber**），而不是**String**。
- 可以聲明一個屬性的類型是郵政編碼（**PostalCode**），而不是長整數（**Long**）。

使用過**Hibernate** **ORM**的開發人員可能會注意到在**Hibernate**中有一個概念是**Dependent Objects**（非獨立的對象），可以認為它所指的就是值對象。**Dependent Objects**是不會被映射為數據庫中的表的，它們會被映射為表中的列。

[1] 見 <https://www.joda.org/joda-money/>。

### 1.3.3 聚合與聚合根、聚合內部實體

在筆者看來，聚合（Aggregate）是DDD在戰術層面最為重要的一個概念。它是DDD可以在戰術層面應對軟件核心複雜性的關鍵。

什麼是聚合？聚合在對象之間，特別是實體與實體之間劃出邊界。聚合內的實體分為兩種：聚合根（Aggregate Root）與聚合內部實體（或者非聚合根實體）。



提示在本書的行文中可能會使用以下幾種說法：

- 聚合內實體，指聚合內包括聚合根的那些實體，一般不會把聚合根排除在外。具體含義讀者可以根據上下文判斷。
- 聚合內部實體，把聚合根排除在外的聚合內部的實體。
- 非聚合根實體，非常明確地把聚合根排除在外的聚合內部的實體。需要避免產生歧義時會使用這個說法。

一個聚合只能包含一個聚合根。當客戶端需要訪問一個聚合內部實體的狀態時，最先能得到的只有聚合根，然後通過這個聚合根，才能進一步訪問到聚合內的其他實體。

從一個聚合根出發能夠訪問到的實體可以認為是一個整體。聚合內部實體的生命週期由它們所屬的聚合根控制。如果聚合根不存在，那麼在它控制下的聚合內部實體也就不存在了。

很多時候，一個聚合內只有聚合根這一個實體。有人聲稱，從經驗上判斷，大約有70%的聚合只有一個實體。

 提示 實體又被稱為引用對象，這個名稱與值對象的概念相對。

本書中提到聚合根、實體、值對象的時候，多數情況下並不會特別說明是指這些對象的“類型”還是指它們的“實例”，請讀者自行根據上下文判斷。

我們應該把一個聚合根的實例以及生命週期完全受它控制的那些聚合內部實體的實例作為一個整體來看待，對於這樣一個整體，有時候書中會使用“聚合實例”這個說法。

為了更好地理解什麼是聚合、聚合根、聚合內部實體，下面舉例說明。這個例子包含訂單（Order）、

訂單頭 ( **OrderHeader** ) 、訂單行項 ( **OrderItem** ) 三個互相關聯的概念。

想象我們在給一個電商系統構建訂單相關的模型，我們可能會得到：

- 一個叫作 **Order** 的聚合。
- 這個訂單聚合的聚合根是一個叫作 **OrderHeader** 的實體。實體 **OrderHeader** 的 ID 叫作 **OrderId** ( 訂單號 ) 。
- 通過 **OrderHeader** 實體，我們可以訪問 **OrderItem** 實體的一個集合。**OrderItem** 這個實體的局部 ID 叫作 **ProductId** ( 產品 ID ) 。因為業務邏輯可能已經明確不允許在同一個訂單的不同訂單行項內出現同一個產品，所以我們可以選擇產品 ID 作為訂單行項的局部 ID ( Local ID ) 。

顯然，在這裡，**Order** 聚合與 **OrderHeader** 聚合根的名字並不相同，這體現了聚合與聚合根這兩個概念的微妙區別。但是確實在大多數時候，聚合和它的聚合根擁有同樣的名字，甚至在大多數時候，一個聚合內就只有聚合根這一個實體。

### 1.3.4 聚合的整體與局部

聚合內的實體，從聚合根到其他實體，往往是一個強烈的整體與局部的關係。這意味著對聚合內部的非聚合根實體而言，它的**ID**（全局**ID**）極有可能最少由兩個部分組成：一部分是聚合根的**ID**，另一部分是它在聚合內的局部**ID**。

比如說，**OrderItem**的局部**ID**可能是**ProductId**。如果允許同一個訂單的不同行項出現相同的產品，那麼最少你還可以創造一個叫作“行號（Sequence Id.）”的概念作為訂單行項的局部**ID**。

再比如，人（作為聚合根）有左手和右手，那麼，“左”或“右”就是“手”（作為聚合內部實體）的局部**ID**。

關於全局**ID**與局部**ID**的問題，後文還會進一步探討。

### 1.3.5 聚合是數據修改的單元

可以從這個角度考慮聚合的邊界：如果兩個實體之間的狀態出現不一致（即使只有非常短暫的一瞬間我們能看到它們的不一致）就讓人難以接受的話，那麼這兩個實體屬於同一個聚合；否則它們就不屬於一個聚合。

比如，也許我們不能接受一個訂單的所有訂單行項的金額之和不等於訂單頭的總金額。那麼，我們可以把訂單頭與訂單行項這兩個實體劃歸到一個聚合內。

對於一個聚合內的對象狀態（數據）的修改，我們需要保證它們總是一致的，也就是說我們要實現它們的強一致性。

聚合是數據修改的單元，這個觀點其實已經被廣泛接受，所以有人說大多數NoSQL數據庫都是聚合型數據庫。因為主流的NoSQL數據庫，不管是文檔型的MongoDB，還是KV型的Redis，它們最少都能保證一個聚合內的數據的強一致性，不管它們把這個聚合叫作文檔還是叫作Value。

你可能聽說過所謂“聚合內強一致，聚合外最終一致”的建議。它的意思是，對於同一個聚合內實體的數據（狀態）的一致性，開發人員可以使用數據庫系統

提供的ACID事務來實現強一致性；對於不同聚合之間的數據，開發人員應該優先考慮自己編碼來實現最終一致性。DDD社區中很多人都贊同這個實踐。



提

示強一致性模型與最終一致性模型是兩種不同的數據一致性模型，這裡的模型可以理解為數據庫系統與開發者之間的契約（Contract）。強一致性模型是指，開發者只要按照數據庫系統的某些規則來做，就可以保證數據絕對不會不一致。最終一致性模型是指，開發者按照一定的規則來做，數據庫中的數據項在處理“過程中”可能會出現不一致，但是在處理過程結束後，它們最終可以達到一致。

### 1.3.6 聚合分析是“拆分”的基礎

分佈式系統設計原則的第一條：不要“分佈”。確切地說，如果沒有足夠的理由就不要做分佈式設計，開發和運維分佈式系統往往需要更高的成本。

為什麼要分佈？常見的理由是為了解決系統的水平擴展問題。這往往意味著我們需要給系統中的軟件元素（構造塊）劃分出邊界，這樣我們就可以更有針對性地對每個構造塊進行獨立的優化。此外，我們可能還需要對同一類型的數據（比如說訂單數據）進行橫向切分（也就是所謂的Sharding），讓每個分佈式系統的結點只負責處理其中的一部分數據。

那麼，到底什麼東西能分、什麼東西不能分，它們之間的邊界如何表述呢？這就是聚合這個概念發揮作用的地方。有些東西互相關聯、密不可分，那麼它們可能應該建模成一個“聚合”。如果從一開始，在軟件設計中就使用了“聚合”這個概念的話，實際上表明我們已經縱向地給其中的軟件元素劃分了邊界，也給數據的橫向切分（Sharding）提供了標準。

比如說，如果我們通過聚合分析已經把OrderHeader和OrderItem劃分到一個聚合內，將它們與其他聚合（比如Product、Payment等）區分開來，我們可能會考慮開發一個叫作Order Service的可獨立部署的微服務。如果要做訂單——OrderHeader以及它

關聯的OrderItem——數據的橫向切分，我們只要根據聚合根的ID（可能我們把這個ID叫作OrderId）來進行SQL數據庫的分庫分表操作就行了。必要的時候，我們可以直接把某些聚合的數據存儲到NoSQL數據庫中。

遺憾的是，有一些DDD的框架和工具，僅僅滿足於使用“一個聚合只有一個實體”的例子來展示它們的“強大”。我們認為，一個DDD工具如果不支持聚合概念，那它就是一個“有殘疾”的DDD工具。雖然據說按照經驗估計，使用DDD方法建模，70%的聚合都只有一個實體（即只有聚合根這個實體），但是這並不等於聚合分析不重要。領域中真正複雜的業務邏輯，可能大部分都集中在另外30%的聚合以及需要處理不同聚合之間的數據一致性的領域服務內，而聚合分析尤其能為數據一致性的處理提供幫助。

如果一個DDD工具不能真正地支持聚合概念，那麼使用它的產品人員、系統分析師、開發人員在分析階段很可能也就不會認真去做聚合分析，因為“即便分析了，大家也未必真的按照這個來做”。

如果一個DDD框架不允許聚合內存在多個“實體”，那麼實際上就沒有了聚合這個概念（當然，若只剩下大部分開發人員都耳熟能詳的“實體”這一概念，確實很“方便”大家理解），如此一來我們採用強一致性模型的邊界在哪裡？如果沒有明確的標準，“哪幾個

實體的數據需要保證強一致”這樣的決定應該如何做出？

可能有人覺得可以做二選一的設計決策：

- 要麼是當要改變狀態時，都先開始一個數據庫事務，操作完若干實體/表後提交數據庫事務，讓數據庫事務來保證強一致性。這其實是我們常見的傳統做法。

- 要麼是認為每個實體就構成了一個聚合，當要改變系統狀態的時候，都自己寫代碼來實現“最終一致性”。如果系統的最終用戶（業務部門）能接受所有實體的狀態只要最終一致就可以，那麼這麼做至少理論上是可行的。

但是現實中這兩種（極端）做法在開發大型應用，特別是互聯網應用時，往往都不太現實。

前者除了很容易導致代碼層面的緊密耦合以外，往往還會造成嚴重的性能和擴展性問題，為了解決這些問題可能需要大規模地重構代碼甚至要重寫整個系統。

可能有人會想：那就不要那麼極端，我們可以不只是“聚合內強一致”，也“適當”地多用數據庫事務來做“聚合外強一致性”。特別是在軟件開發的初始階段，軟件往往還是個單體應用，數據庫可能還是一個單

點，一時半會兒也不會分庫分表，可以讓開發人員“便宜行事”，自行控制數據庫事務來保證在必要範圍內的“強一致性”。

現實情況是，“便宜行事”的開發人員很有可能“做過頭”，因為根本沒有清晰的“邊界”。極端的情況就是臭名昭著的“**Open Session In View**”，也就是服務端收到客戶端請求時立即開啟一個數據庫事務，在將請求的處理結果返回客戶端前的那一刻才提交數據庫事務，這樣所有對系統狀態的查詢和修改操作都會放到同一個數據庫事務中完成——可以說是非常“方便”了。但是，這樣做的結果往往是災難級的系統性能表現，以及剪不斷理還亂的麵條式代碼。

後一種做法，可能有人覺得不是問題。他們可能會說：“我們就是應該堅持原則，一個實體一個聚合，所有實體之間都最終一致。業務部門要的是結果，他們才不管什麼強一致還是最終一致，銀行轉賬這麼重要的事情，大家還不是普遍接受‘最終一致’？所以，經過精心設計的業務邏輯都是可以繞過大多數對強一致性的‘偽需求’的。”



**提  
示**這裡所謂接受“最終一致”，意味著要接受“有時候數據就是不一致的”。比如，我們大多數時候都能接受出現這樣的情況：我們執行了一個轉賬操作，源賬戶減少了500塊錢，目標賬戶沒有馬上增加500塊錢。

首先，我們認為實體之間的強一致性需求一定是存在的，我們不能寄希望於“最終一致性”能滿足所有的業務需要。就算退一萬步，業務部門確認“最終一致性”真的能滿足他們的所有需求，這種做法也會極大地增加軟件開發的工作量與成本。如果大量處理“最終一致性”的代碼都需要程序員來“手動”編碼實現的話，即使有一些框架和工具能提供一定的幫助，程序員也會不堪重負，產出的可能是Bug滿滿的低質量代碼。

所以軟件開發團隊幾乎總是會混合使用強一致性和最終一致性模型。如果開發團隊想就這個問題——什麼時候選擇強一致性模型、什麼時候選擇最終一致性模型——確定標準，那麼即使不使用“聚合”這個術語，遲早也會發明一個相似的概念。

總之，聚合分析的意義就是讓開發人員一開始就在強一致性和最終一致性的選擇上進行足夠的思考和權衡，而不是沒有想清楚就匆忙進入編碼階段。如果先做了聚合分析，即使因為項目工期的要求，沒有完全按照“聚合內強一致，聚合外最終一致”的原則來編寫代碼，至少開發人員也清楚地知道自己在哪些實現代碼上是做了妥協的。也許有人會認為既然要妥協，那就沒有必要做預先的分析和設計。但是筆者認為，就現狀而言，對軟件進行預先設計的價值已經被太多人低估了。

順便說一下，做聚合分析還有其他好處。一個好處是可以根據聚合分析的結果自動生成UI層的代碼

——最少是可以生成腳手架代碼。想一想訂單頭和訂單行項同屬一個訂單聚合的情況，它們在與分屬不同聚合的兩個實體各自獨立、毫不相干的情況比起來，顯然應該是不一樣的吧？這個問題後面可以進一步探討。此外，聚合分析有助於自動生成CQRS的“讀模型”（後面會討論CQRS模式，這裡的讀模型可以先理解為數據庫的視圖），因為我們知道聚合內的實體關係密切，需要一起查詢的可能性很高。比如，因為訂單聚合的存在，我們可以自動生成聯接（Join）了訂單頭和訂單行項兩個表（實體）的視圖（讀模型）。

### 1.3.7 服務

系統中有一些行為是不適合“歸屬於”哪個對象的，DDD建議把這樣的行為定義為服務（Service）。或者說，當有一個操作需要修改多個聚合實例的狀態時，這個操作就很有可能應該被定義為服務。



注意“服務”這個名詞在軟件開發過程中實在是被用“濫”了。當需要與其他“服務”進行區分時，會稱這裡所說的“服務”為“領域服務”。

舉個例子，我們開發一個賬務系統的時候，可能需要支持轉賬功能。所謂的轉賬，就是在一個賬目（Account）上扣減一定金額的錢，在另外一個賬目上增加相應金額的錢。

那麼這個轉賬行為作為賬目的一個方法來定義是不是合適呢？如果是作為賬目的方法，那麼它是應該這樣定義（Java代碼）：

---

```
public class Account {  
    public void transferFrom(Account sourceAccount, Money  
    amount) {  
        //...  
    }  
}
```

---

還是應該這樣定義：

---

```
public class Account {  
    public void transferTo(Account destinationAccount, Money  
amount) {  
        //...  
    }  
}
```

---

我相信，上面兩種做法都會引起（不必要的）爭議。何不按如下形式定義一個轉賬服務呢（Java代碼）：

---

```
public interface TransferService {  
    void transfer(Account sourceAccount, Account  
destinationAccount, Money amount);  
}
```

---

對於這個轉賬服務來說，它要改變的是兩個 **Account** 聚合實例的狀態，雖然這兩個實例的類型都是 **Account**，但是仍然屬於所謂的“跨聚合（實例）”的操作，在這種情況下，將其定義為服務比較合適。

在DDD中，還有其他**Repository**（存儲庫）、**Factory**（工廠）等戰術層面的概念，建議讀者自行閱讀DDD相關圖書或者到Google上查詢資料。

## 1.4 戰略層面的關鍵概念

相信很多人都見過這樣的情形：從小項目演化而來的大系統最終變成開發團隊的噩夢。這些噩夢幾乎無一例外地源於軟件的概念完整性（**Conceptual Integrity**）遭到了破壞（很少是因為單純的技術原因）。代碼可能是一代又一代的開發人員各行其是地堆疊起來的（所謂的“祖傳代碼”），在這個過程中沒有人意識到有必要去維護軟件的概念完整性。而 DDD，特別是DDD在戰略層面提供的概念，是維護軟件概念完整性的良藥。



提

示概念完整性對軟件開發的重要性在弗雷德裡克·布魯克斯（Frederick P. Brooks Jr.）的經典著作《人月神話》一書中被重點提出。

什麼叫作概念完整性？通俗地講，就是所有人對領域內的所有事物持相同的看法。舉例來說，我們把“張三”這個“對象”叫作張三後，就不會再把他叫作李四。我們不會把張三的兒子也叫作張三，我們可以把他叫作“張小三”。大家都知道張三和張小三之間存在父子關係，不會剛才有人說他們是父子，轉臉又有人說他們是兄弟。

在《領域驅動設計精簡版》的第5章中，介紹了 DDD在戰略層面維護模型的概念完整性的方法。筆者

個人認為其中最重要的兩個概念是限界上下文（**Bounded Context**）與防腐層（**Anti-Corruption Layer**），下文打算針對它們以及“統一語言”進行闡述。其他戰略層面的概念讀者可自行閱讀《領域驅動設計精簡版》或搜索其他文章以深入瞭解，在此不再贅述。

### 1.4.1 限界上下文

為了開發軟件，我們需要分析應用要解決的領域問題的範圍，捕獲那些重要的概念，給它們明確的定義——通俗地說，一個“說法”（概念）不要指兩個“東西”，當然一個“東西”也不要有兩個“說法”。我們使用這些概念來構建模型，反映我們對領域的認知。

限界上下文（Bounded Context），顧名思義就是限定了邊界的上下文，它定義了每個模型的應用範圍。在本書的行文中，經常會把限界上下文簡稱為“上下文”。

怎麼理解“上下文”這個說法？為什麼不叫作“子系統”“模塊”？

這麼說吧，很多詞語都可能存在多個含義，它到底是哪個意思，往往需要聯繫上下文才能判斷。比如，當我們說起“Good”這個單詞時，它到底是“貨物”，還是“好的東西”“好的品質”呢？它到底是名詞，還是形容詞呢？如果它不是出現在一個上下文中——比如某篇文章的某一段的某個句子中，我們其實無從判斷。

限界上下文就是這樣的一個上下文、一個邊界，在這個邊界內，所有重要的術語、詞語都有一個明確

的解釋。你看，上下文這個說法是不是比“子系統”“模塊”之類的說法要貼切？

有人可能會提出這樣的問題：限界上下文多大才合適，劃分上下文有沒有什麼可以遵循的規則？

劃分上下文的規則，無非還是放之四海而皆準的“高內聚，低耦合”，這麼說估計大家都會覺得太虛。其實真正讓大家感到糾結、不知如何切分的那些東西之間一定有所關聯，那就乾脆都納入一個上下文中！與其關注上下文的“大小”，不如關注模型的“質量”——概念完整性有沒有被破壞。筆者認為：判斷大小是不是合適，要看應用的開發團隊能在一個多大的範圍內掌控軟件的概念完整性，只要開發團隊沒有問題，那麼這個範圍就算再大，作為一個上下文來處理都是可以的。

## 1.4.2 限界上下文與微服務

那麼，限界上下文與微服務架構<sup>[1]</sup> ( MSA ) 中的那個微服務之間有什麼關係？有人還使用過“物理限界上下文”這個術語，甚至把一個可獨立部署的微服務稱為物理限界上下文，這個叫法合理嗎？

讓我們來設想一下，現在要給一個餐廳開發一個在線外賣 ( Takeout ) 應用。我們使用了微服務架構，決定在後端開發兩個可以獨立部署的微服務：

- **Order Service**。與訂單管理相關的服務，它與前端App進行交互，處理消費者的下單請求。
- **Kitchen Service**。處理後廚相關的業務邏輯。



提

示事實上要創建一個真實的外賣應用，可能要處理的後端業務邏輯遠比這裡列出的多，比如可能我們還需要用**Consumer Service**來處理消費者的信  
息，需要用**Accounting Service**來處理支付邏輯等。這  
裡為了簡化問題，先忽略它們。

假設這兩個微服務是以發佈/訂閱事件的方式——也就是使用所謂的基於協作的**Saga** ( **Choreography-based Saga** ) ——來完成業務事務的，“下單”事務處理的正常路徑如下：

1 ) Order Service接到客戶端App的下單請求，創建一個處於PENDING狀態的訂單，然後發佈OrderCreated ( 訂單已創建 ) 事件。

2 ) Kitchen Service訂閱、消費OrderCreated事件，驗證訂單信息，檢查後廚的食材、物料、人員等信息，創建後廚工單 ( Ticket )，併發布TicketCreated ( 後廚工單已創建 ) 事件。

3 ) Order Service訂閱、消費TicketCreated事件，將訂單狀態置為APPROVED，併發布OrderApproved事件。

假設開發人員編寫了以上事件 ( OrderCreated、TicketCreated ) 以及其他領域對象的靜態類型——Java開發人員稱之為POJO，.NET開發人員稱之為POCO，PHP開發人員稱之為POPO——的代碼，然後把這些代碼都放到一個叫作common-api的類庫項目中——我們可以構建這個項目，打包出jar、dll、phar之類格式的歸檔包文件，供其他項目使用。另外兩個“服務”項目Order Service和Kitchen Service都依賴這個common-api類庫項目，它們會直截了當地使用裡面的靜態類型，使用它們的時候並不會經過什麼轉換，而且會毫不在意地對其他部分 ( 比如客戶端 ) 暴露它們的名字和概念。現在，請問我們是有“一個”限界上下文還是有“兩個”限界上下文？

有些講述在微服務架構中實踐DDD的文章，對於這樣的情形會判斷為兩個限界上下文，每個微服務對應一個上下文。

從限界上下文原本的概念出發，筆者認為這裡只存在一個上下文。在這個例子裡，兩個服務都依賴同一個領域對象的類庫，大家都說統一的語言（Ubiquitous Language），使用同一套術語—OrderCreated、TicketCreated等—來進行溝通，彼此之間並沒有什麼隔閡，顯然大家都處於同一個限界上下文中。

限界上下文是概念的邊界，微服務是物理的軟件組件。微服務架構（MSA）在實踐中可能會產生很多細粒度的微服務，這些微服務的物理邊界與DDD的聚合邊界或領域服務的邊界對齊是比較常見的情況。如果一個微服務與其他微服務共享很多相同的概念，那麼雖然它具有獨立的物理邊界，也很難稱得上是一個限界上下文。

[1] 見  
<https://www.martinfowler.com/articles/microservices.html>。

### 1.4.3 防腐層

我們創建的應用總是免不了要與外部的應用發生交互。與外部應用交互時，應該考慮使用六邊形架構 [1] 風格。

在傳統的分層架構風格中，上層的軟件構造塊（元素）依賴於下層的構造塊。六邊形架構風格摒棄了這種觀點。它認為，一個軟件構造塊如果需要與外部軟件元素進行交互來實現功能、完成自己的使命，那麼它也只應該依賴於抽象。這個抽象就是它對外部軟件元素的期望、設想。這其實也是依賴倒置原則 [2] 的應用。

擴展一下上面的在線外賣應用的例子。假設為了實現這個應用，我們還需要兩個服務參與其中：

- **Consumer Service** (消費者服務)，它的職責是處理對消費者信息的查詢和驗證邏輯。
- **Delivery Service** (送餐服務)，這個服務負責送餐業務流程的管理。

這兩個服務真正的業務邏輯可能並不需要外賣應用的開發團隊去實現。也許送餐本來就是由第三方公司提供的服務，對方已經有了管理送餐業務的軟件（下面假設這個軟件叫送餐系統）；也許有些業務是

公司內其他部門的團隊負責的，並且已經有了“現成的”系統，比如**Consumer Service**需要的信息在公司內的**CRM**系統中本來就存在。不管怎麼說，現在“送餐服務”和“消費者服務”的業務邏輯都不需要外賣應用的開發團隊去實現了，大家要做的只是把已有的應用與外賣應用進行集成。

總之，不管是外部的送餐系統還是公司已經存在的**CRM**系統，都不在外賣應用開發團隊控制的範圍之內，它們是另外兩個上下文。但是外賣應用仍然可以提出它希望這些外部服務“看起來是什麼樣子的”，比如：

- 它希望**Consumer Service**有一個**verifyConsumer**方法。調用這個方法可以驗證某些消費者的信息是否有效。
- 它希望**Delivery Service**有一個**scheduleDelivery**方法。若使用訂單信息作為參數調用這個方法，就可以請求送餐服務的提供者對訂單進行派送。
- 它希望送餐服務的提供者“知道”它提供了一個叫作**NoteOrderDelivered**的通知接口。在送餐員把食物送到消費者手中之後，它希望這個接口被調用，且它能收到一個類型為**OrderDelivered**（訂單已完成派送）的事件，然後根據事件信息做進一步的處理。

外賣應用對外部系統的這些期望都是“單相思”。它才不管外部的**CRM**系統有沒有一個名字叫作

`verifyConsumer`的方法呢，也許在CRM裡面根本就沒有Consumer這個概念，只有Customer的概念。它也不管在送餐員完成派送後送餐系統能不能發佈一個類型為OrderDelivered的事件，也許在送餐系統裡面根本沒有Order的概念，只有Delivery的概念，類似的事件叫作DeliveryCompleted。

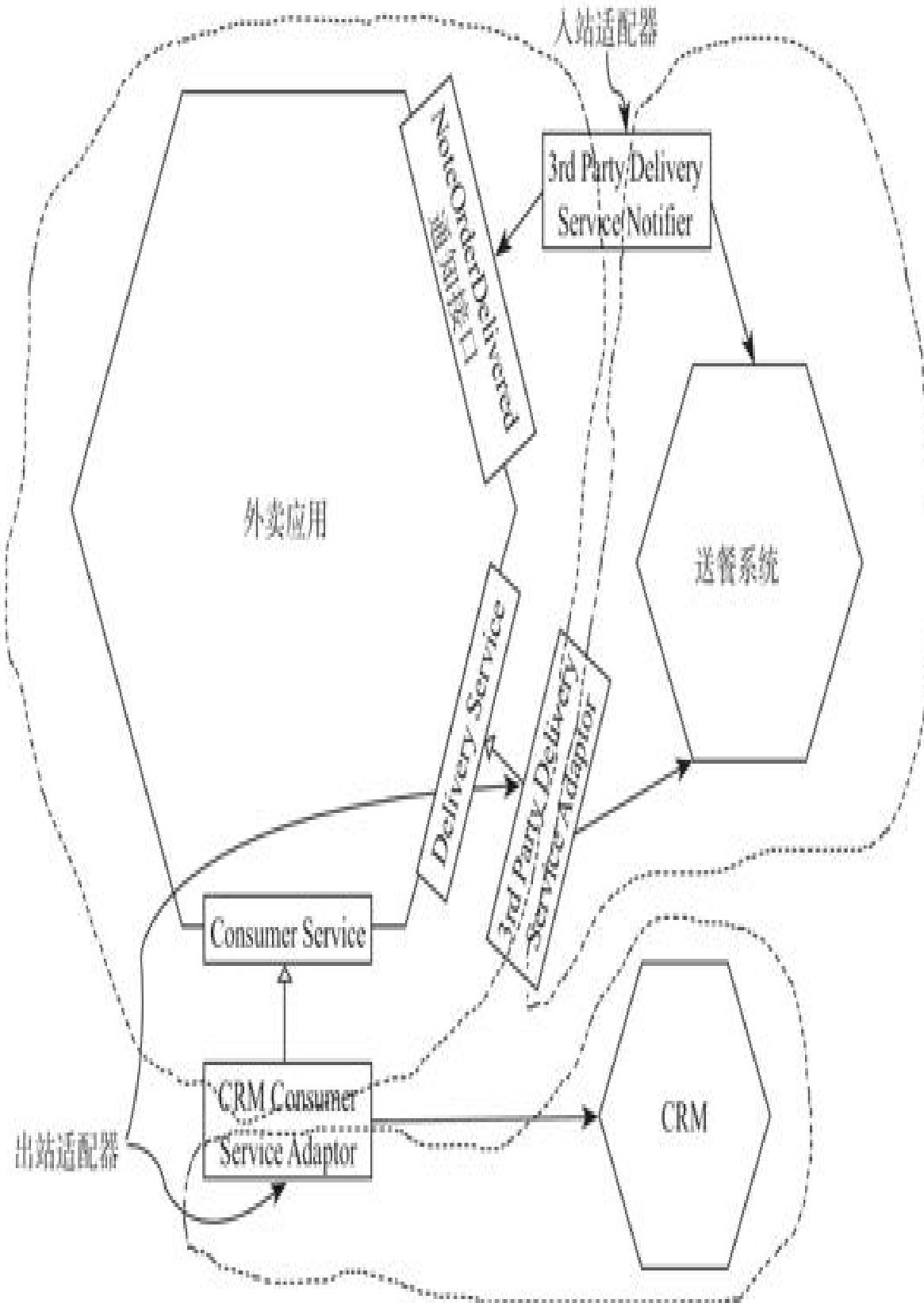
外賣應用不管這些，它就要說“Consumer”“OrderDelivered”，而不管外部系統的叫法。它把“想要的東西”都按照自己的想法定義了接口，也就是說，現在Consumer Service和Delivery Service都是外賣應用定義的接口——這就是所謂的“依賴於抽象，不依賴於實現”。外賣應用的這些抽象，都屬於限界上下文的一部分。在自己的上下文內，只要能保證概念完整性即可，外賣應用沒有什麼理由改變自己，保持純粹就挺好的。

如果外賣應用未能抵制住誘惑，讓外部系統的概念滲透進了內部，比如在代碼裡面一會兒稱客人為Consumer，一會兒又叫人家Customer——如此不正經，那就是“腐化”墮落。

如果外部的CRM系統和送餐系統也不想改變自己，那麼矛盾如何解決？怎樣把三觀不同的系統“整”到一起？這就需要防腐層（Anti-Corruption Layer）在中間牽線搭橋、互相撮合了。

對於採用六邊形架構的應用來說，防腐層一般是以作為適配器來實現的（如圖1-1所示）。

六邊形架構的適配器是限界上下文的核心業務邏輯組件與外部系統之間的交互接口（又稱為端口）的實現。適配器分兩類：出站適配器與入站適配器。一個是適配器是“出站”還是“入站”是站在核心業務邏輯組件的角度看的。如果交互是請求/響應模式的調用，出站適配器實現對外部系統的調用，入站適配器處理外部系統的請求、調用內部的業務邏輯。如果交互是異步消息通信，那麼出站適配器實現對外發送消息，而入站適配器接收外部發送過來的消息。



## 圖1-1 六邊形架構與防腐層

假設，外賣應用上下文在使用CRM或送餐系統提供的服務時屬於“弱勢客戶”，那麼外賣應用的開發團隊就需要自己構造防腐層，防止“強勢的供應商”（CRM上下文或送餐系統上下文）一方的概念侵入自己的上下文中。

在圖1-1中，外賣應用的開發人員最終實現了三個適配器：

- 為Consumer Service接口實現了一個出站適配器CRM Consumer Service Adaptor。
- 為Delivery Service接口實現了一個出站適配器3rd Party Delivery Service Adaptor。
- 為NoteOrderDelivered通知接口實現一個入站適配器3rd Party Delivery Service Notifier。

防腐層的代碼就位於這三個適配器的內部。

我們在圖1-1中用虛線劃出了三個限界上下文的邊界。只有那些包含防腐層代碼的適配器是橫跨不同的限界上下文的邊界的，它們負責在不同上下文之間對概念進行翻譯和轉換。雖然外賣應用最終（在運行時）還是需要使用這些適配器才能實現完整的功能，但是就代碼的依賴關係而言，是適配器依賴外賣應用定義的那些接口——這些接口又叫抽象，它們屬於外賣

應用核心業務邏輯組件的一部分。開發人員都知道這一點：當一個類實現一個接口時，依賴關係是類依賴接口，而不是接口依賴類。

總之，限界上下文是需要時刻保護好的概念的邊界。需要將某個上下文的概念轉換為另一個上下文概念的地方就是防腐層。最好讓防腐層的代碼和其他軟件元素（構造塊）之間存在明顯的物理邊界，倒不是說一定要把防腐層部署為一個個獨立的服務或在獨立的進程中運行，但是，最少我們還可以考慮將一個防腐層作為獨立的類庫項目進行構建和維護。

[1] Hexagonal architecture (software),  
[https://en.wikipedia.org/wiki/Hexagonal\\_architecture\\_\(software\)](https://en.wikipedia.org/wiki/Hexagonal_architecture_(software))。

[2] Dependency Inversion Principle,  
[https://en.wikipedia.org/wiki/Dependency\\_inversion\\_principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle)。

#### 1.4.4 統一語言

領域驅動設計的一個核心原則是使用一種基於模型的語言——統一語言（Ubiquitous Language）。

統一語言使用模型作為語言的主幹，團隊在進行所有的交流時都應該使用它。在共享知識和推敲模型時，團隊在PPT、文字和圖形中使用它。程序員在代碼中也要使用它。

為了創建團隊的統一語言，我們需要發現領域和模型的那些關鍵概念（其中有些概念可能很不容易被發現），並找到適合描述它們的詞彙，然後開始儘可能多地使用它們。

統一語言為何如此重要？眾所周知，在複雜軟件系統的開發過程中，大量甚至是大部分的時間和資源都消耗在思想的溝通和確認上了。統一語言基於“模型”，精心構建的模型歸納總結了各方對領域的一致認知，統一的“說法”可避免歧義、減少溝通中的誤解。所以，統一語言可以成為各方（業務人員、領域專家、產品人員、開發人員、測試人員）進行高效溝通的基礎，從而極大地節約軟件開發的成本。

在實踐上，我們建議使用一個簡潔的詞彙表來記錄一個上下文的統一語言中的那些關鍵概念。

比如，表1-1是GitHub.com的reactive-streams/reactive-streams-jvm<sup>[1]</sup>代碼庫中一個詞彙表的一部分。

表1-1 一個英文詞彙表的示例

Term	Definition
Signal	As a noun: one of the onSubscribe, onNext, onComplete, onError, request(n) or cancel methods. As a verb: calling/invoking a signal
Synchronous(ly)	Executes on the calling Thread
Return normally	Only ever returns a value of the declared type to the caller. The only legal way to signal failure to a Subscriber is via the onError method
Responsivity	Readiness/ability to respond. In this document used to indicate that the different components should not impair each others ability to respond
Terminal state	For a Publisher: When onComplete or onError has been signalled. For a Subscriber: When an onComplete or onError has been received

對於主要由母語為中文的開發人員組成的團隊來說，建議在軟件開發項目中使用中英文對照的詞彙表，並且建議在詞彙表中強調詞條的詞性（名字、動詞、形容詞等），甚至名詞的單複數形式等。因為在中文中很多詞語的詞性都是模糊和多變的，需要根據上下文去辨別，母語為中文的開發人員在文檔和代碼中使用錯誤的詞性的情況實在太過常見。表1-2是中英文對照的詞彙表的一個例子，也許可供借鑑。

表1-2 一箇中英文對照詞彙表的示例

词语	词性	中文词语	定义
Close(document)	v (动词)	关闭 (单据)	关闭单据 (订单等)
Reversed(document)	adj (形容词)	反转的 (单据)	单据被执行反转 (撤销) 操作后, 处于此状态
Personal name	n (名词)	人名	人名包括两部分, 姓 (First name), 名 (Last name)
Parties	n-plural (名词, 复数)	当事人列表	当事人的列表、集合

[1] Reactive Streams Specification for the JVM,  
<https://github.com/reactive-streams/reactive-streams-jvm>。

## 1.5 ER模型、OO模型和關係模型

我們可能有必要先了解一下在軟件開發活動中常見的這幾種模型，或者說模型範式，如下：

- ER模型（實體關係模型）。讀者如果有興趣，可以通過Wikipedia瞭解一下ER模型<sup>[1]</sup>，以及相關的概念數據模型、邏輯數據模型、物理數據模型等。

- OO模型（面向對象模型）。這可能是大多數開發人員最熟悉的一種程序設計模型了。

- 關係模型。這是我們常見的關係型數據庫系統（RDBS）所採用的模型，大部分應用軟件使用的數據庫都是關係型數據庫。

這幾種模型都有一套自己的概念和術語，其中有些概念存在大致的對應關係，我們可以用圖1-2來大致表示。

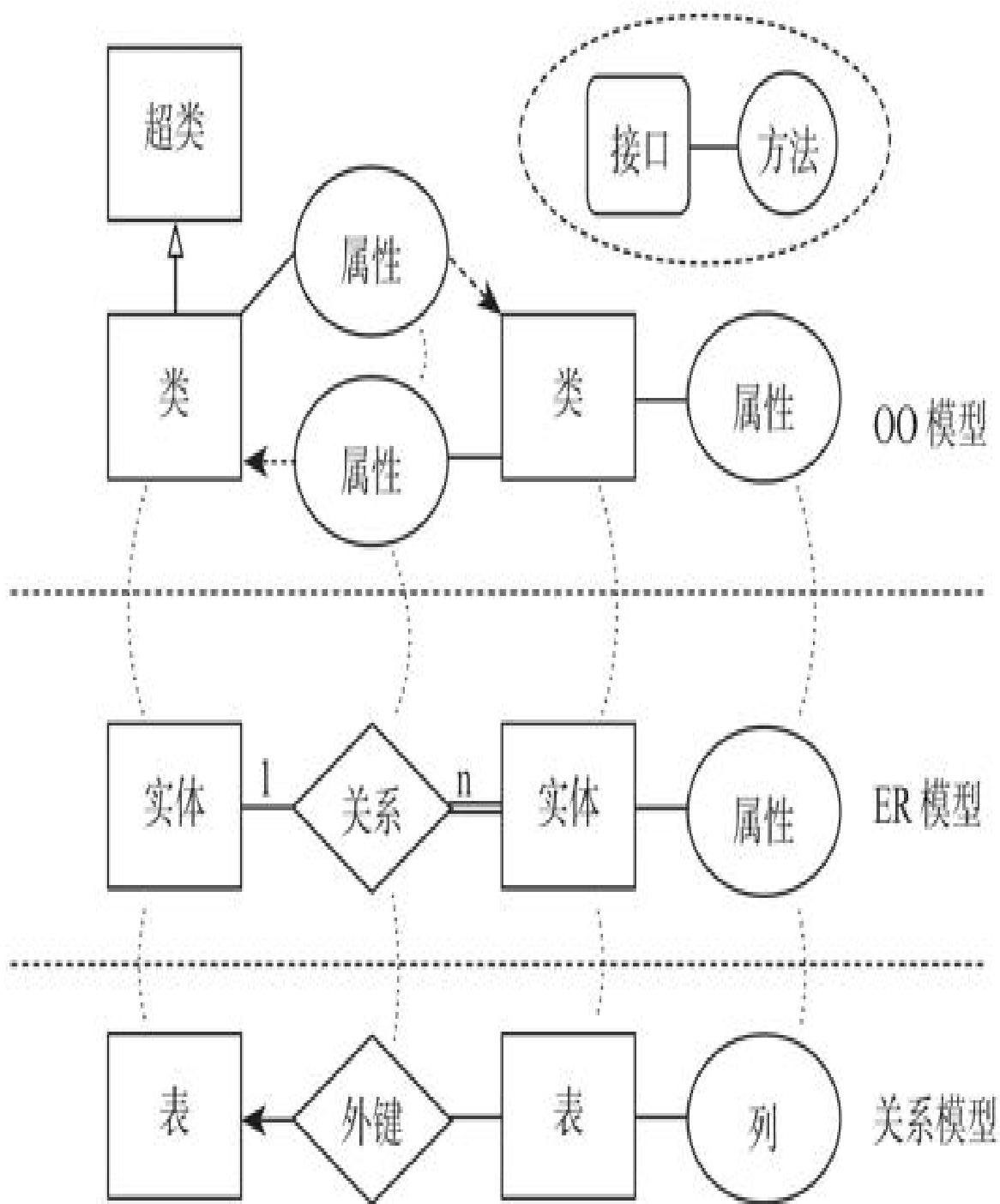


圖1-2 ER模型、OO模型與關係模型



提示其實關係模型使用的“原汁原味”的術語是以下形式。

- 關係 ( Relation )：一個關係對應著一個二維表。
- 元組 ( Tuple )：在二維表中的一行，稱為一個元組。
- 屬性 ( Attribute )：在二維表中的列，稱為屬性。

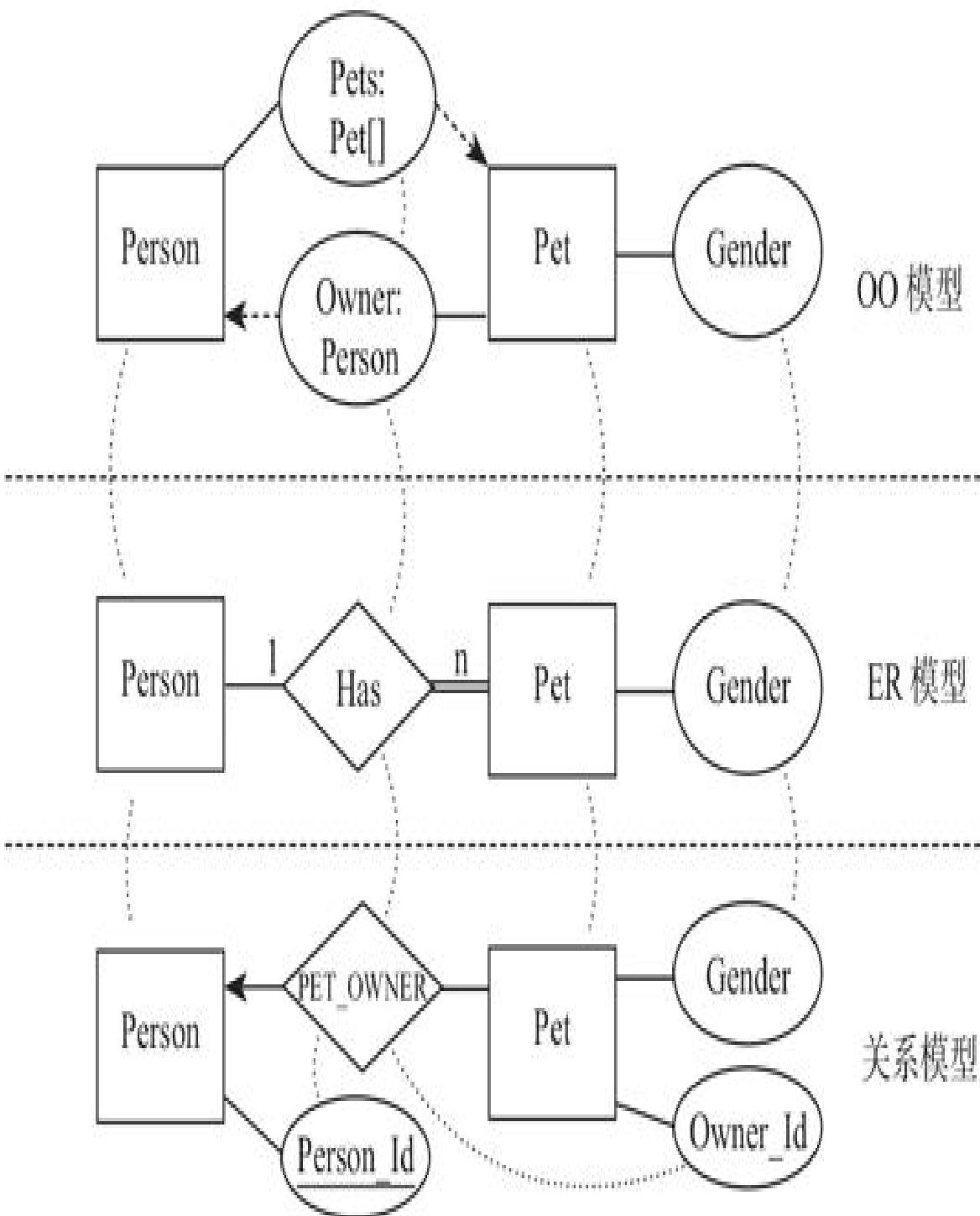
為了照顧大多數開發人員的習慣，書中還是採用了“表”“行”“列”這樣的術語。

模型是用來反映對領域的認知的，對於同樣的認知，我們其實可以使用不同的方式來表述。因為我們的目標是開發軟件，所以最好是使用一些有利於後面編碼“實現”的表述方式。

還是舉個例子吧，假設我們對現實世界（領域）的認知是這樣的：

- 寵物 ( Pet ) 有性別 ( Gender )，有自己的主人 ( Owner )，假設一隻寵物只有一個主人。
- 一個人 ( Person ) 可以擁有多隻寵物。

我們把這些信息（認知）使用不同範式的模型來描述，結果可能大致如圖1-3所示。



## 圖1-3 分別使用ER模型、OO模型與關係模型表述同樣的認知

·對於關係模型，Pet表中存在Gender和Owner\_Id列；Person表中存在Person\_Id列，這一列是Person表的主鍵（PK）。Pet表中存在一個外鍵，可以把這個外鍵命名為PET\_OWNER，它從Pet表的Owner\_Id列指向Person表的Person\_Id列。

·對於ER模型，有一個實體叫作Pet，它有一個屬性叫作Gender；還有一個實體叫作Person。從Person實體到Pet存在一個“一對多”的關係，這個關係的名稱叫作Has（有），即一個人可以擁有多隻寵物的意思。

·對於OO模型，有個類叫作Pet，它除了有一個Gender屬性，還有一個Owner屬性，後者的類型是Person，指向寵物的主人。而類Person有一個屬性叫作Pets，它的類型是Pet的集合（在圖1-3中記作Pet[]），指向一個人可以擁有的寵物（Pets）。Person的Pets屬性與Pet的Owner屬性其實描述的是同一個關係。看一個通俗的示例，無論是說“張小三的爸爸是張三”，還是說“張三有個兒子叫張小三”，都表示“張三與張小三存在父子關係”。

[1] Entity–relationship model (ER model), [https://en.wikipedia.org/wiki/Entity%20%80%93relationship\\_model](https://en.wikipedia.org/wiki/Entity%20%80%93relationship_model)。

## 1.6 概念建模與模型範式

有人認為，在將現實世界映射到信息世界的第一階段，即概念建模階段，適宜使用ER模型。

對此我們有不同看法。實際上不管是ER模型、OO模型還是關係模型，用來做概念建模都沒有問題。所謂的概念建模，主要是選擇忽略領域中那些相對不太重要的細節，只留下最關鍵的部分，這與使用的模型範式無關。

ER模型和關係模型的一些概念本來就存在較好的對應關係。關係數據庫的流行已經證明了關係模型的表述能力。在實踐中，在概念建模階段使用二維表（關係模型）來展示領域中的關鍵概念以及概念之間的聯繫，很多時候對非技術人員來說都是可以理解和接受的，甚至對他們來說是很“親切”的。我們經常看到很多業務人員（軟件的最終用戶）、業務部門製作的Excel極其複雜，我不認為他們理解關係模型有太大的難度。而用於操作關係型數據庫的SQL（結構化查詢語言）一開始也並非是專門面向技術人員而設計的。

UML<sup>[1]</sup>（通用建模語言）作為一門“通用”語言，其野心自然是覆蓋從概念建模到實現建模的需要。UML在設計上傾向於滿足使用OO範式進行建模的需

要，大家一直都在使用UML進行概念建模（構建概念OO模型）。而DDD的領域模型就是基於OO模型的。它是在OO模型的基礎上進一步抽象的，它定義了一些更高層次的概念，比如聚合、聚合根、服務、Factory、Repository等，所以在概念建模階段使用DDD領域模型也毫無問題。

實際上，DDD關注的重點就是在概念建模階段產出那個概念上的“領域模型”，只是DDD強調在做概念建模的時候要朝著軟件的實現方向“睜一隻眼”(*with an eye open*)。如果有一門DDD專用的建模語言，估計大家也會期望它能覆蓋從概念建模到實現建模的需要吧！

[1] 見<http://www.baike.com/wiki/UML>。

## 第2章 其他DDD相關概念

在認識或溫習了DDD在戰略以及戰術層面的部分關鍵概念之後，我們還需要進一步瞭解其他一些相關的概念。因為在實踐DDD的過程中我們會經常使用這些概念，它們也很重要，而在其他DDD圖書中對它們強調得可能還不夠。這些概念包括領域ID、Local ID、Global ID、命令、事件、狀態等。

## 2.1 領域ID

根據DDD對實體的定義來看，實體必然存在一個ID，我們可以把這個ID稱為實體的領域ID。

相信很多人都想問以下問題：

- 這個領域ID是不是應該映射到關係模型的自然鍵？如果一個實體的ID已經是“自然鍵”了，那麼與之對應的關係數據庫的Table中還有必要再引入這個ID之外的代理鍵嗎？
- 如果領域ID可以是代理鍵，那麼它什麼時候應該是代理鍵？
- 在某些軟件系統上，開發人員會不分青紅皂白地給每個表（實體）設計一個代理主鍵。那麼，在代碼中這樣重度地使用代理鍵是合理的做法嗎？
- 如果一個實體需要被其他實體引用，其他實體是不是應該儘可能統一地通過持有它的領域ID的值來引用（指向）它？

接下來我們討論一下這些問題。

### 2.1.1 自然鍵與代理鍵

什麼是自然鍵（Natural Key）？先看看維基百科的定義：

在關係模型數據庫的設計中，自然鍵是指由現實世界中已經存在的屬性所構成的“鍵”。舉個例子，美國公民的社會保險號碼可以用作自然鍵。換句話來說，自然鍵是與那一“行”中的屬性存在邏輯關係的候選鍵。自然鍵有時候也被稱為領域鍵。

從這個定義可知，自然鍵有時候也被稱為領域鍵，領域鍵和領域ID這兩個稱呼都帶著“領域”，已經很接近了，不是嗎？

我們看到，這個定義還依賴於另外一個概念：現實世界（The Real World）。那麼，什麼是現實世界？如果這個概念沒有定義，那麼自然鍵的概念還是“不清不楚”的。



提

示雖然在討論領域模型的時候，我們更願意使用實體ID這個說法，而不是使用關係數據庫中“鍵”的概念，但是在不太嚴格的情況下，有時候我們也會混用實體ID和（自然）鍵這兩個說法。

筆者曾經在Google上搜索到一篇文章Choosing a Primary Key:Natural or Surrogate?<sup>[1]</sup>，也許，我們可以通過文章中的以下描述理解什麼是The Real World（中文翻譯版本）：

不要自然化代理鍵。一旦你向最終用戶顯示了代理鍵的值，或者更壞的是允許他們使用該值（例如搜索該值），實際上你已經給它們賦予了業務含義。這實際上是自然化了代理鍵，從而失去了代理鍵的優點。

根據這個說法，我們可以認為，使用軟件的最終用戶所生活的世界就是現實世界，這個世界沒有代理鍵的容身之地。這麼說來，代理鍵只應該活在技術人員的世界裡？

總之，能不能見“人”（指的是最終用戶），是筆者能找到的用於區別自然鍵和代理鍵的最不讓人困惑的標準了。

[1] 見<http://www.agiledata.org/essays/keys.html>。

## 2.1.2 DDD實體的ID需要被最終用戶看到

DDD所說的實體的ID應不應該見“人”（最終用戶）呢？

顯然，實體ID有99.99%的可能性是需要見“人”的。這個好像已經不需要再展開討論了吧？因為領域模型應該是由技術人員與領域專家共同參與構建的，而領域專家很可能就是最終用戶。實體與值對象最本質的區別就是實體存在ID，這麼至關重要的事情，可不適合對領域專家藏著掖著吧？也許在軟件的內部實現中，技術人員確實會使用一些不會被“軟件所服務的領域”的最終用戶看到的實體，但這些實體是“非主流”，而且對於這些實體來說，“技術領域”就是它服務的領域，技術人員就是它的最終用戶，所以它還是被最終用戶看到了。

如果實體的ID需要被最終用戶看到，那麼，在將領域模型映射到關係模型時，實體ID的對應物就必然是“自然鍵”。雖然Domain-Driven Design: Tackling Complexity in the Heart of Software一書中對此有不同的看法，但是筆者仍然堅持自己的觀點。



提

示簡單總結一下上面的論證過程：根據DDD的基本概念可知，有沒有ID是實體和值對象的本質區別。沒有ID，實體這個概念就不存在了。而沒有實體，就無法進行DDD領域模型在戰術層面的設計。至關重要的是，實體的ID要被最終用戶看見，所以它在數據庫層面對應的東西就是自然鍵。可見，一個實體只有代理鍵沒有自然鍵是不對的，代理鍵不能替代自然鍵。

上面的論證是從DDD的基本概念出發的，那麼，不用DDD能不能開發出一個系統？答案是可以。但是因為這裡的討論就是以實現DDD為前提的，所以讓我們暫時先忘掉沒有DDD的世界……

另外，我想要說的是：不應該把鍵的生成方式當成區分代理鍵和自然鍵的標準。

這樣的情形並不罕見，我們部署一個分佈式的ID生成服務，既可以用它來生成最終用戶看不到的代理鍵，也可以生成最終用戶可以看到的自然鍵。

在給領域建模的時候，完全可以做出這樣的表述：訂單號是訂單實體的ID，這個ID是“任意的”（Arbitrary）的，也就是說我不關心它的編碼規則，只要它是唯一的就好。既然實體的ID有時候可以那麼“隨性”，那麼誰規定自然鍵（領域ID）就不能使用GUID、HiLo Table等方式生成呢？還有，只要最終用

戶可以接受，領域ID當然可以是一個順序增長的整數。

所以，如果一個ID需要被最終用戶看到，即使它使用數據庫的自增長列來生成，那麼也應該被理解為自然鍵。

既然DDD要求實體的領域ID（也就是自然鍵）必然存在，那麼，大多數時候我們直接使用它就可以了。如果一定要給實體對應的表額外加上一個不能見“人”的代理鍵（多個“鍵”必然會加重程序員的選擇困難症，所以最好不要），那麼我們希望它最好“淪落”為偶一為之的優化技巧（Trick），省得大家經常需要為了選擇用哪個鍵而傷腦筋。

那麼，代理鍵這個Trick應該在什麼時候使用，它有什麼好處和代價呢？

### 2.1.3 什麼時候使用代理鍵

同樣是在文章Choosing a Primary Key:Natural or Surrogate?中，這樣舉例說明了使用代理鍵的好處（中文翻譯版本）：

自然鍵的缺點是由於具有業務含義，因此它們與業務直接耦合。於是，你可能在業務需求變更時需要重新指定鍵。例如，當你的用戶決定將CustomerNumber列從數字型改為字母數字型時，除了更新表Customer的模式（這個是不可避免的），還需要修改每一個使用CustomerNumber作為外鍵的表。

我們幾乎可以非常肯定，在這個例子中，即使在Customer表引入一個代理鍵（假設列名為ID），CustomerNumber（客戶編號）仍然是被廣泛使用的備用鍵（即大家都知道它具有唯一性約束，並且會利用這一點）。也就是說，在現實世界裡，CustomerNumber可能會被很多最終用戶看見，公司內部的各個部門，比如客服、訂單處理、技術等部門，都可能使用客戶編號來查詢客戶的信息。如果這個客戶編號沒有唯一性，只會帶來沒有必要的麻煩。

假設Customer表使用了一個代理鍵，當你在數據庫中將CustomerNumber列從數字型改為字母數字型的時候，可能僅僅是修改數據庫模式的工作量會小一

些。但是，不管怎麼樣，你仍然需要仔細地檢查代碼，確認所有使用了Customer Number這個概念的地方，都正確地處理了“CustomerNumber類型的變更”，你可能還要知會通過API使用了Customer Number的其他應用的開發團隊。

## 1.不應讓實體總是使用代理主鍵

有人主張“讓實體總是使用代理主鍵”的原因之一是：假設某個實體/表一開始使用的是自然主鍵，一段時間之後，可能會發現開始作為自然主鍵的那個屬性/列在現實中並不具備唯一性，這時候修改數據庫的模式和程序代碼會比較麻煩。如果一開始這個表就使用了代理主鍵，那麼修改會更容易。

問題是：

- 如果一開始這個表既使用了代理主鍵又使用了自然鍵作為備用鍵，那麼，難道我們只需要把備用鍵那一列或幾列的唯一約束去掉就可以了嗎？

- 如果一開始這個表只使用了代理主鍵，沒有其他備用鍵，這個設計是不是就沒有問題了？

對於第一種情況，與上面修改CustomerNumber的類型的問題一樣，這些讓“修改數據庫模式更容易”的做法可能會誘惑開發人員放棄對領域更深層次的思考，草率地進入編碼階段。當我們需要移除一個備用

自然鍵時，也許應該重新思考一些問題，比如到底是什麼原因導致了當初的認知偏差？難道我們不需要檢查其他代碼，也不需要“告知”其他與當前應用存在交互的軟件開發團隊嗎？如果去掉了某一列或幾列的唯一約束之後，這個實體/表除了代理主鍵，沒有其他備用鍵，那麼應用的最終用戶是不是隻能使用代理主鍵去追蹤實體的狀態，從而自然化了代理鍵（讓代理鍵實際上變成了自然鍵）呢？

對於第二種情況，一個實體/表只有代理主鍵（沒有備用鍵），大多數時候筆者都會反對這麼做。

## 2. 關注領域ID，忘掉代理主鍵

在設計領域模型的時候，建議先關注領域ID，忘掉代理主鍵。

來看一個真實的案例，某個電商系統的會員等級表只有一個自增長列代理主鍵，這個代理鍵是最終用戶看不到的。這個系統曾經出現這樣一個Bug：部分客戶在前端APP上明明看著自己是“二星會員”，在購買商品的時候卻沒有享受到二星會員應該享受的折扣價。出現這個問題的時候用戶和運營人員完全摸不著頭腦。其實引發問題的原因是：數據庫中存在兩個名為“二星會員”的會員等級記錄，其中一個被運營人員設置為“棄用”了，但是在客戶表中，仍然有客戶記錄通過外鍵指向它。

雖然對於這個例子而言，當時開發人員編寫的業務邏輯層代碼確實存在問題，但是，如果當初會員等級表除了存在一個代理鍵之外，還存在備用鍵，比如將表示等級高低的“等級序號”或“等級名稱”作為備用鍵，那麼這個問題就可以避免。因為運營人員將無法往表中添加具有同樣等級序號或等級名稱的多條記錄。

部分開發人員實現代理主鍵的方式是使用數據庫的“自增長列”，也就是說在將記錄數據插入數據庫之前，我們並不知道記錄的ID。大多數時候，這種代理主鍵都是沒有存在的必要的。一個實體/表只使用自增長列代理主鍵（沒有其他備用鍵）往往會給數據庫初始化、單元測試編寫、系統集成、Bug重現、數據分區（Sharding）、灰度發佈等帶來各種麻煩。

比如，當表中只存在一個自增長列代理鍵的時候，會給實現保證“幂等”的接口帶來額外的複雜性。使用不幂等的接口來“導入數據”會導致表中出現大批量重複的數據——這種情況筆者在工作中看到過很多次，有時候處理這些重複數據的代價極大。

我們使用的關係型數據庫擅長保存數據，但不擅長實現“業務邏輯”，業務邏輯一般存在於數據庫之外的應用代碼中（我可不會考慮把業務邏輯放在存儲過程之類的“代碼”中）。我們經常會使用在代碼中定義的一些表示實體ID的“常量”，然後使用它們在數據庫中查找某些特定的記錄，並使用這些記錄數據來實現

某些業務邏輯。如果數據表只存在一個自增長列代理主鍵，會給這類業務邏輯的實現和測試帶來不必要的麻煩。

所以，強烈建議在將領域模型的實體映射到關係模型的數據表時，如果一定要使用代理鍵，那麼在代理鍵之外，務必設計備用的自然鍵，這個自然鍵對應著實體的領域ID。也就是說，一個表就算使用了代理鍵，我們還是需要告訴大家：存在其他具有業務含義的一個列或多個列的組合的鍵，你們可以放心地使用這些鍵的唯一性，我們不會輕易地“取消”這些自然鍵。

大多數人抗拒使用自然主鍵的原因主要是基於性能的考量。比如，如果僅使用代理主鍵，在插入記錄前則不用生成主鍵的值；如果使用自然主鍵，插入前可能需要先調用一個ID Generator Service ( ID生成服務 )。又比如，使用順序增長的代理主鍵可以更有效地利用存儲空間；使用一個整數型的代理主鍵來查找數據庫記錄，比起使用（可能根據最終用戶的要求必須是）字符型的自然主鍵，速度可能快不少。但是，“對軟件的過早優化是萬惡之源”，建議不要在一開始就給每個實體對應的表添加一個代理鍵，可以在後期有絕對必要的時候再考慮作為技術優化的手段來引入代理鍵。

## 2.2 ID、Local ID與Global ID

當我們談到一個聚合內部（非聚合根）實體的ID時，默認指的是這個實體的Local ID（局部ID）。所謂的Local ID，指的是對於這個實體（類型）來說，只要保證在同一個外部實體的實例內、該實體（類型）的不同實例之間，這個ID的值具備唯一性就可以了。

在這個局部ID的基礎上，我們認為這個實體還有一個派生的、可以稱為Global ID（全局ID）的屬性。也就是說，我們可以認為這個派生的Global ID屬性的值，由實體的Local ID的值以及它所有的外部實體的ID的值組合而成。

我們可以給這個Global ID屬性指定一個名稱，這個名稱會佔用屬性所屬的實體（類型）成員的名稱空間。在實踐中，我們也經常會給這個Global ID創建一個專門的值對象（類型）。

強調Global ID這個概念是有意義的。有時候我們可能需要使用一個實體的Global ID的信息，以直接定位聚合內部實體的某個實例。

比如，我們定義了一個聚合，聚合根是訂單頭（訂單號是它的ID），訂單行項是這個聚合內部的實體。假設產品ID是訂單行項的Local ID。有時候，我

們可能會說：“訂單號1000567，產品ID為2000765的那個訂單行項。”

像這樣的情況並不少見，所以，如果由一個值對象（類型）來清晰地體現某個“聚合內部實體的Global ID”這個概念不是很好嗎？並且，在編碼實現、需要做O-R Mapping（對象關係映射）的時候，這個值對象可能就很有用。

更進一步，我們可以認為一個聚合內部的實體在概念上存在一個或者多個派生的外部ID屬性，它們指向其外部實體的ID（通過它們可以獲取外部實體的ID）。

## 2.3 命令、事件與狀態

在後面的討論中可能會經常使用以下概念。

- 命令：我（客戶端）想要系統（服務端）幹什麼。命令應該使用動詞或者動詞性短語來命名。比如 `CreateOrder`（創建訂單）、`Rename`（重命名）等。

- 事件：已經發生的事實。事件應該以動詞的過去分詞形式命名。比如 `OrderCreated`（訂單已創建）、`Renamed`（已重命名）。

- 狀態：系統現在或者某一刻是什麼樣子的。狀態應該是名詞，可以輔以 `State` 作為後綴。比如 `OrderState`、`PersonState`。

舉個例子，我到銀行去取款：

- “我想要把我這張卡的錢全部取出來”——這是一個命令。

- “先生，這是您的900元錢”——這是一個事件（客戶已取款900元）。

- “現在，您的賬戶餘額為0元”——這是狀態。

這三者是不同的概念，它們之間並不必然存在一一對應的關係，雖然有時候它們確實“長得很像”：

- 事件是一種描述“發生了什麼”的對象，很多時候它是因為命令而產生的，所以它可能與命令對象“長得很像”。

- 命令、事件對象也可能與狀態對象長得很像。比如一個CreateProduct（創建產品）命令和ProductState（產品狀態）的屬性列表可能幾乎一樣。

但是它們仍然是有本質區別的，只要考慮這一點就夠了：有些事情（事件）不需要命令也一定會發生，比如時間的流逝。

理解了這幾個概念，你會更容易理解DDD社區在實踐DDD時採用的一些架構模式，比如Event Sourcing（事件溯源）和CQRS模式。

這裡需要注意聚合的整體性。很多時候，客戶端發出的命令只會修改一個聚合實例的狀態，對於這樣的命令，可以稱之為“聚合的命令”。描述這樣的命令產生的後果的那個事件，可以稱為“聚合的事件”。

有時候，你可能覺得自己是在“面向”一個聚合內部的非聚合根實體發出命令，比如：假設我們把訂單頭和訂單行項定義為一個訂單聚合內的實體，那麼，“把這個訂單行項的數量改為30”這個命令就應該被理

解為針對訂單聚合的命令。你應該能想象到這個命令很可能會引起訂單頭的總金額發生變化，我們需要保證訂單頭與訂單行項之間狀態的強一致性。

另外，有時候我們需要區分廣義的命令與狹義的命令。廣義的命令指的是客戶端發出的請求，而狹義的命令是指那些要求改變系統狀態的請求。本書提及的“命令”是廣義的命令還是狹義的命令，需要根據上下文進行分辨（多數時候指的是狹義的命令）。比如，CRQS（命令查詢職責分離）模式所說的命令就是狹義的命令。當我們談到異步的基於消息的通信（Asynchronous Message-Based Communication）機制時，會把請求/異步響應（Request/Asynchronous Response）模式稱為命令模式——這裡的請求有可能是查詢請求，所以這裡的命令是廣義的命令。

## 第3章 CQRS與Event Sourcing

可能有讀者看到，很多討論CQRS（命令查詢職責分離）模式及事件溯源（Event Sourcing，ES）模式的文章都會提及DDD。DDD社區在實踐DDD的時候，也經常會採用CQRS及ES模式來解決一些實際的問題。

因此，很多人混淆了CQRS與ES的概念，認為它們是一回事。但嚴格來說，並非如此。本章就來說明兩者的區別。

“有經驗的”程序員可能經常會在數據模型中添加一些所謂的“冗餘字段”來優化系統在執行查詢時的性能表現。要維護這些字段與核心數據模型之間的一致性是一個苦差事，也許應該使用更系統化的做法，比如CQRS模式。

另外，應用的服務化常常導致用戶需要查看（查詢）的數據分佈在不同的數據庫中，CQRS模式可以將這些數據整合起來，提供一個查詢模型（視圖）以支持查詢功能的開發。

如果我們已經事先知道用戶對某些數據存在審計或可追溯方面的需求，那麼可以設計出相應的數據模型，比如採用後面要介紹的From-Thru模式來滿足這些需求。但是，我們經常是在軟件開發的後期，忽然遭

遇一些審計或可追溯方面的要求，如果在一開始就採用了事件溯源模式，可能就會相對輕鬆不少。

## 3.1 命令查詢職責分離

什麼是命令查詢職責分離 ( Command Query Responsibility Segregation , CQRS ) 模式？網上有 Martin Fowler 對 CQRS 的定義<sup>[1]</sup>，這裡不贅述。筆者認為，CQRS 模式的核心是分離的概念模型。如圖 3-1 所示，CQRS 模式將模型分成兩個，一個叫命令模型（或者稱為寫模型），它是為更新（ update ）操作構建的；另一個叫查詢模型（或者稱為讀模型），它是特別為查詢構建的。這裡“分離的概念模型”一般指的是對象模型。至於使用了幾個物理數據庫，這並非是 CQRS 模式的關鍵。

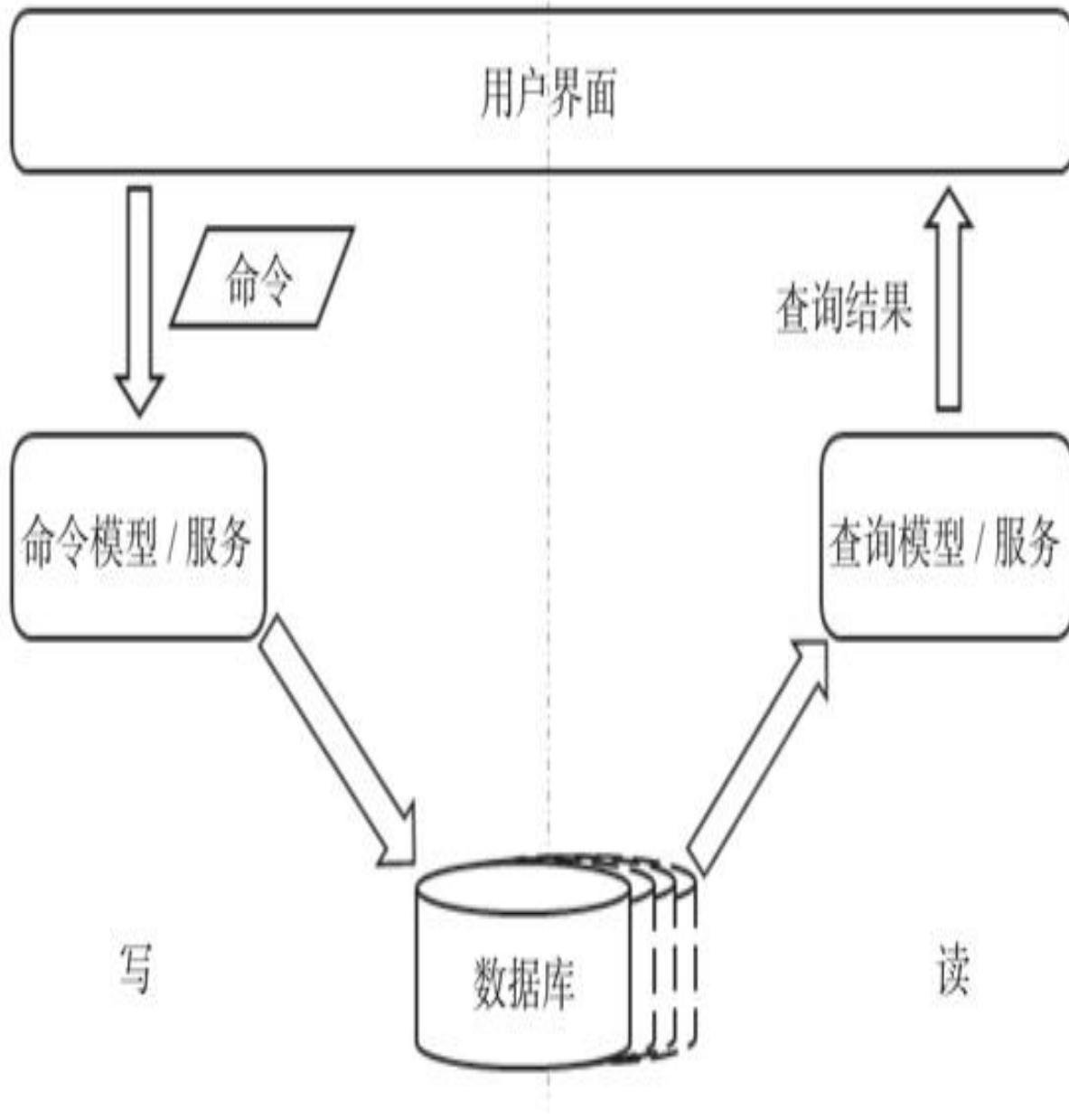


圖3-1 命令查詢職責分離模式

簡單地說，只要讀、寫操作是基於不同的概念模型，或者概念模型是基於讀和寫分別優化過的，那麼基本就可以說是採用了CQRS模式。

不少開發團隊會基於數據庫複製機制來構造在數據庫層讀寫分離的應用。這種方式幾乎不需要“改造”已有的應用代碼，就可以很有效地提高應用的性能表現。有人把這種做法也稱為CQRS，但是筆者難以贊同。CQRS所做的是在概念模型上清晰地分離讀模型與寫模型。不過確實可以同時使用“基於數據庫複製機制的讀寫分離”與CQRS模式。

使用數據庫複製機制的時候，主數據庫處理創建、修改、刪除（Create、Update、Delete）操作，從數據庫處理讀取（Retrieve）操作，主從數據庫之間通過數據庫的複製機制進行同步。在這種方式下，既可以構建分離的命令與查詢模型/服務（這時候就可以說是CQRS），也可以只構建單一的後端模型/服務，如圖3-2所示。

如此說來，CQRS也實在是平平無奇。當我們要構建一個比較複雜的應用時，就算沒有分開構建可各自獨立部署的命令與查詢服務，也很有可能會為查詢接口創建一些靜態類型（對象）代碼（也就是程序員所說的POJO、POCO、POPO等），這些對象其實就可以稱為“讀模型”。這時，其實我們已經有意無意地部分使用了CQRS的思想。

CQRS模式的價值需要在微服務架構（或者其他服務化架構）下，或者與事件溯源模式聯合使用時，才能充分體現。

相對於單體應用，微服務架構往往意味著使用更多的數據庫。

雖然什麼樣的架構可以稱為微服務架構並沒有一個統一的標準，但是，每個微服務使用“自己的數據庫”，這是得到廣泛認同的微服務架構的應有之義。

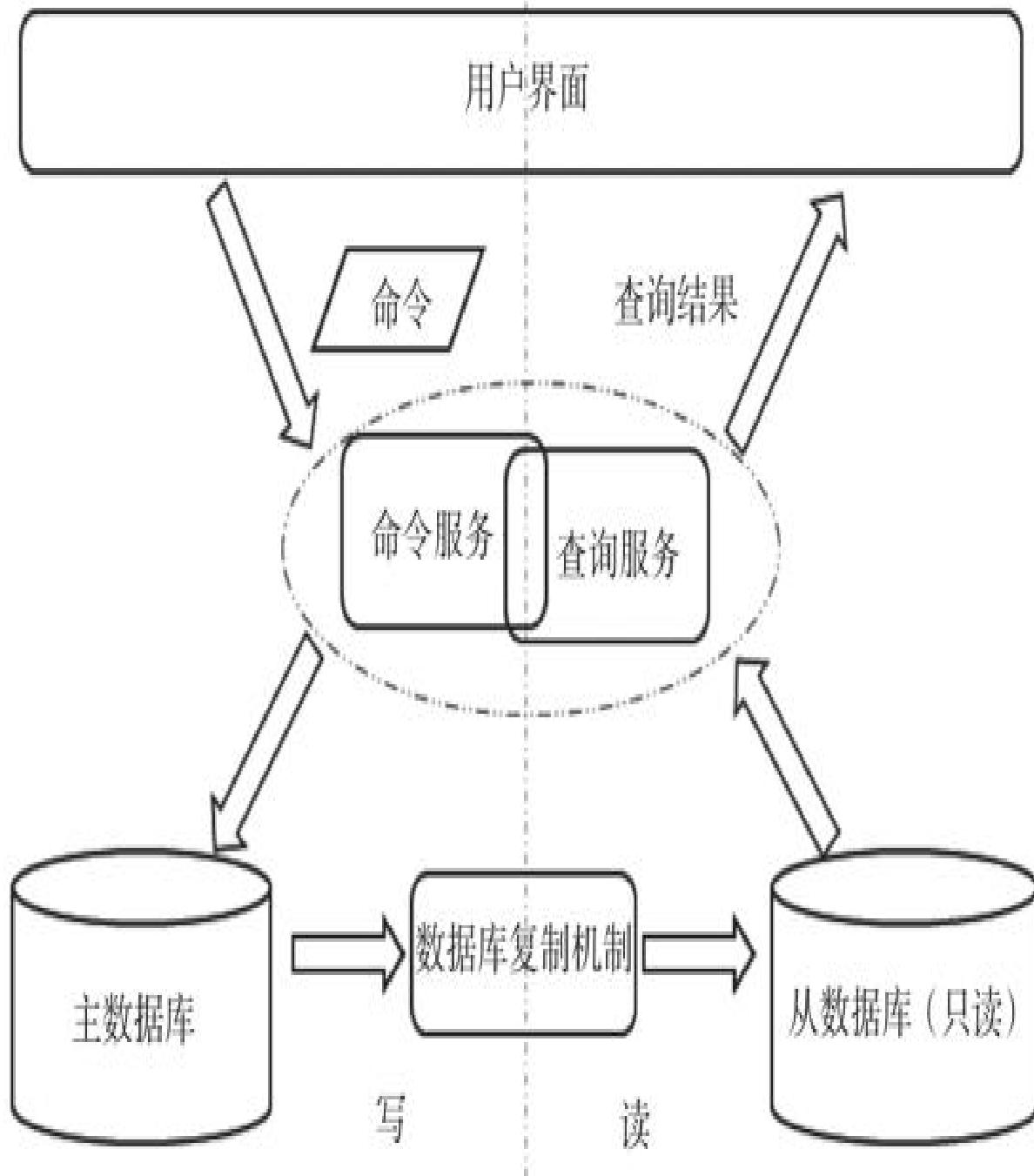


圖3-2 基於數據庫複製機制的讀寫分離

在採用了微服務架構之後，常常需要給前端用戶界面提供跨數據庫的查詢接口。也就是說，往往需要

從多個源頭——不同微服務的數據庫——獲取、整合數據，才能滿足前端UI的展示需求。

那麼，這裡數據的整合用什麼方法實現呢？一個最簡單的方法，就是利用數據庫本身提供的視圖功能，直接在多個微服務的數據庫之上建立視圖，如圖3-3所示。比如說，對於MySQL來說，Database（數據庫）和Schema是指同一個東西，很容易寫出跨Schema的SQL語句；對於Microsoft SQL Server來說，除了可以寫跨Database的SQL語句，還可以把外部數據庫服務器鏈接進來，然後進行Join查詢。

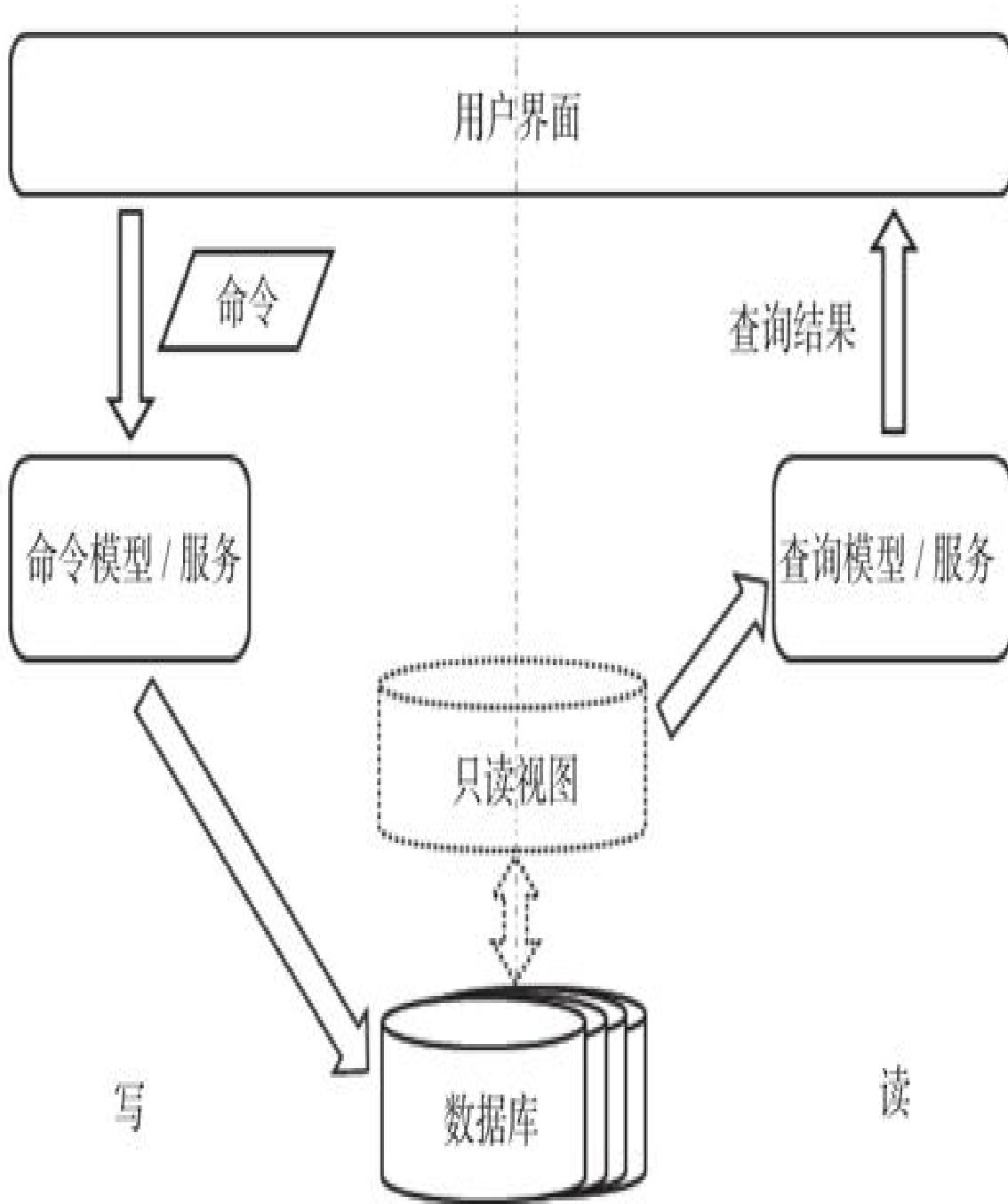


圖3-3 使用數據庫提供的視圖功能實現CQRS

在這種情況下，程序員在對象模型的層面創建與數據庫視圖對應的查詢模型恐怕是再自然不過的事情了，不這麼做反倒略顯奇怪了。筆者認為這樣的情況

可以算是CQRS。也就是說從不同的數據庫整合出來的那個“視圖”對應的對象模型，就是CQRS中的查詢模型。

這樣的CQRS實現直接依賴數據庫提供的功能，很容易保持上層的視圖與底層的數據庫表的同步，看起來確實很簡單，但是也會帶來數據庫技術選型上的限制以及系統的可移植性、運維複雜度等方面的問題。另外，那些需要Join多個表的複雜視圖，在大數據量的情況下性能表現恐怕也不會太好。這時候，也許不只是將概念模型拆分為寫（命令）模型和讀（查詢）模型，將它們依賴的底層物理數據模型也拆分到不同的數據庫中，為讀模型建立相應的物理表（而不是視圖）是一個可以考慮的解決方案——這個做法其實才是很多開發人員印象中“正宗”的CQRS。那麼，此時如何把寫模型數據庫（為寫優化的數據庫）中的變更同步到讀模型數據庫（為讀優化的數據庫）中呢？一個實現方式就是使用事件溯源模式。

要討論CQRS與事件溯源如何結合，應該先了解一下事件溯源是怎麼樣的一個模式。

[1]

見

<https://www.martinfowler.com/bliki/CQRS.html>。

## 3.2 事件溯源

事件溯源簡稱ES。在網上可以找到Martin Fowler對ES模式的定義<sup>[1]</sup>。Martin Fowler對事件溯源的定義很簡單，即將對應用狀態的所有變更作為事件的序列捕獲下來。

當然，既然是捕獲，那麼這些事件自然是需要被可靠地存儲起來的。存儲事件的數據庫稱為事件存儲（Event Store）。

可以想象，既然應用狀態的所有變更都已經作為事件被捕獲下來，那麼不僅當前狀態可以從這些事件中派生出來，對於歷史上每一個事件發生後應用處於什麼樣的狀態其實都可以重現。也就是說，發生某個事件（En）之後，應用的狀態Sn可以表示為：

$$S_n = F(E_1, E_2, E_3, \dots, E_n)$$

函數F就是我們常說的業務邏輯，應用中的業務邏輯都是已知、明確的。

換句話來說，如果有一種方法能追溯在應用中發生過的歷史事件，以及每一個事件發生後應用都處於什麼樣的狀態，就是事件溯源模式。

那麼，在這種情況下，應用的當前狀態其實是沒有必要保存起來的，因為它可以通過重放所有的歷史事件派生出來，如果不保存當前狀態，那麼事件存儲其實等同於CQRS的寫模型數據庫（為寫優化的數據庫）。

在使用ES模式的情況下，改變應用狀態的過程一般是這樣的：

- 1 ) 客戶端 ( Client ) 發出命令。
- 2 ) 服務端收到命令後，一般都需要獲取應用的當前狀態（可能需要通過重放事件來獲得當前狀態），在當前狀態的基礎上確認命令是否合法。
- 3 ) 如果命令合法，則生成事件，把事件追加到事件存儲中。事件被存儲就意味著“已成事實”，接下來的操作都可以認為是“可選項”。
- 4 ) 應用事件，更新內存中的當前狀態，準備接收下一個命令。如果採用了這個方法，那麼就是所謂的 In-Memory 模式。
- 5 ) 在存儲事件的同時在數據庫中保存當前狀態，或者在某個時機（即檢查點）使用數據庫保存當時的狀態快照，這都是可選的。

使用事件溯源模式給CQRS的實現提供了便利。下面來看看結合使用CQRS與ES的系統架構，如圖3-4所

示。

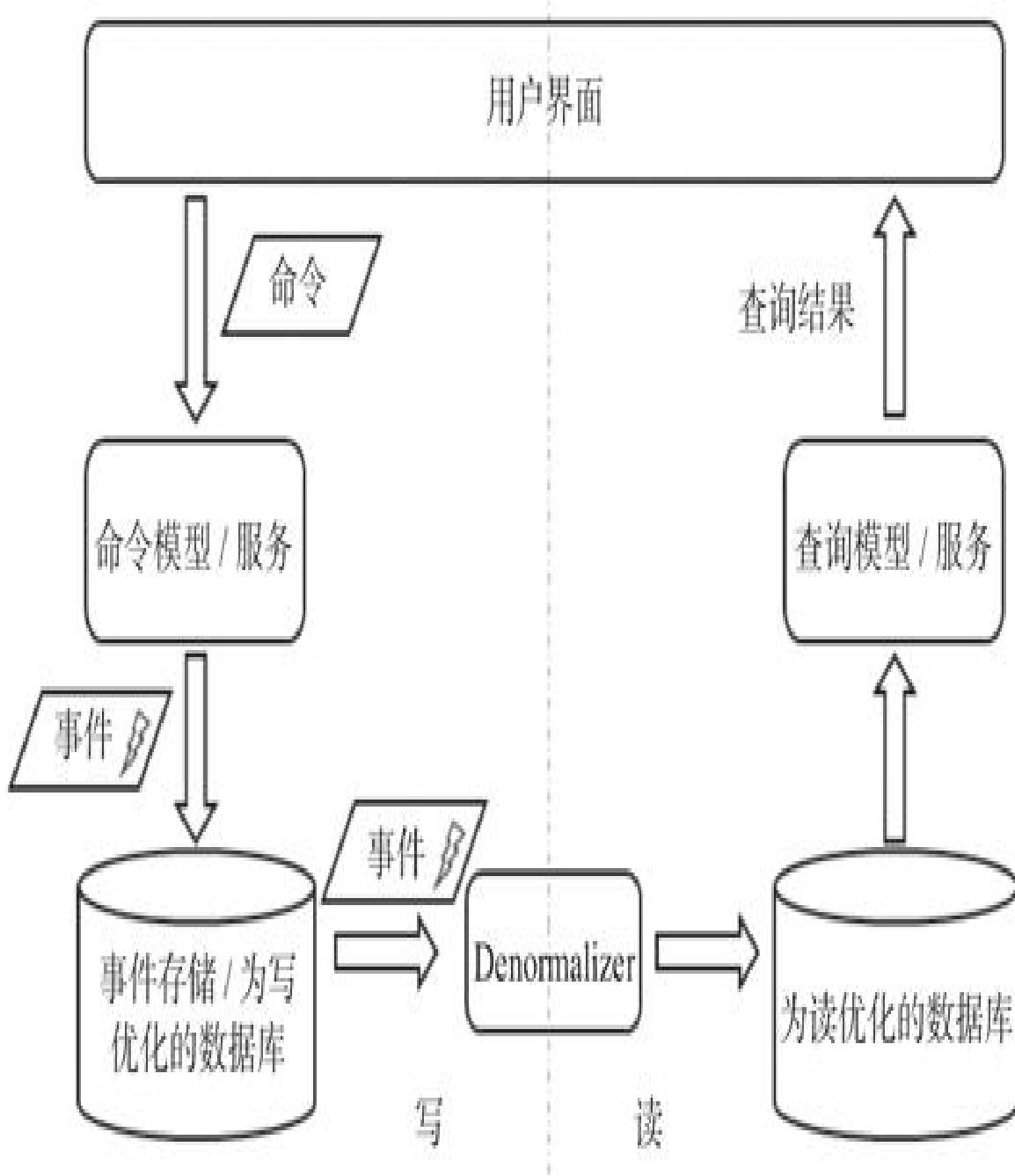


圖3-4 結合使用CQRS與ES的系統架構

因為每次狀態的變更都產生了事件，所以 Denormalizer ( 去規範化器 ) 這個組件通過訂閱、消費這些事件，很容易實現查詢模型數據庫 ( 為讀優化的數據庫 ) 的增量更新。這就是ES與CQRS這兩個名詞總是一起出現的原因。

[1] 見  
<https://martinfowler.com/eaaDev/EventSourcing.html>  
。

### 3.3 From-Thru模式

From-Thru模式來自筆者對開源項目Apache OFBiz<sup>[1]</sup>的數據模型的觀察。在OFBiz的數據模型中，它出現了多次。之所以在這裡提到它，是因為如果採用特定的方法操作這個模式的數據模型，從某種程度上講它就是一個顯式實現的事件溯源（ES）模式。而為了更好地使用它，我們可能又會使用到CQRS模式。

[1] The Apache OFBiz Project, <https://ofbiz.apache.org/>。

### 3.3.1 示例：ProductPrice

以OFBiz中的產品價格（ProductPrice）實體為例，它包含如下主鍵（PK）。

- productId**：產品ID。

- productPriceTypeId**：產品價格類型ID。比如用於表示這個價格是零售價（WHOLESALE\_PRICE）還是標牌價（LIST\_PRICE）等。

- productPricePurposeId**：產品價格目的ID。給價格另外一個分類維度，可以用來表示這是用於採購的價格還是用於電商渠道分銷的價格等。

- currencyUomId**：計價的貨幣單位ID。比如人民幣（CNY）、美元（USD）等。

- productStoreGroupId**：門店組ID。用於將這個價格應用到一組門店，包括線上商店。

- fromDate**：價格的生效時間。

這裡值得特別注意的是**fromDate**這一列（屬性）。相比沒有應用From-Thru模式的“常規”實體，ProductPrice的主鍵主要是多出了此列（屬性）。

除此之外，`ProductPrice`實體還包括如下其他重要的列（屬性）。

- `thruDate`：價格的截止時間。即價格在該時間前有效，如果是`null`，那麼就是沒有截止時間（一直有效）。

- `price`：價格。

- `taxInPrice`：價格是否含稅。

- `taxPercentage`：稅點（百分比）。

這裡需要注意的是`thruDate`這個屬性。有了`fromDate`和`thruDate`，就可以完整地記錄產品價格的“歷史”了。

當產品價格的重要屬性（比如`price`）發生變化的時候，並不會修改原有的記錄——除了那個`thruDate`。我們會把“上一條”價格記錄的`thruDate`改為該價格失效的時間，然後新增一條記錄，`fromDate`是新價格生效的時間。看到這裡，是不是覺得它與事件溯源（ES）模式有些相似呢？

這樣，我們就可以查詢某一個時間點——比如`2019-11-06 17:50:20`——的價格信息：

---

```
SELECT * FROM product_price WHERE '2019-11-06 17:50:20' >=
FROM_DATE and (THRU_DATE is null or '2019-11-06 17:50:20' <
```

```
THRU_DATE);
```

---

這個模式帶來一個問題，就是可能我們需要經常查看“當前”有效的價格記錄，而要獲取這個當前價格的記錄，所寫查詢語句好像還比較煩瑣，大致像這樣（SQL）：

---

```
SELECT * FROM product_price WHERE now() >= FROM_DATE and  
(THRU_DATE is null or now() < THRU_DATE);
```

---

這時，也許需要增加一個表示當前產品價格的實體，我們可以把它命名為ProductPrice-Master（產品價格主記錄）。相對於ProductPrice，這個實體的主鍵去掉了fromDate這一列，只有如下屬性：

- productId
- productPriceTypeId
- productPricePurposeId
- currencyUomId
- productStoreGroupId

那麼，這個實體的數據怎麼維護呢？也許可以考慮採用以下兩種做法之一。

·把ProductPriceMaster和ProductPrice建模成一個聚合。ProductPriceMaster是聚合根，ProductPrice是聚合內部實體，把對聚合的修改封裝成“聚合的方法”，在方法的實現代碼中保證它們之間的數據一致性。

·把ProductPriceMaster當作另外一個聚合的聚合根，並且使用CQRS模式，將ProductPriceMaster看作“讀模型”，ProductPrice看作“寫模型”。然後創建一個軟件組件（“消費者”）去訂閱ProductPrice的領域事件，當ProductPrice的狀態發生變化時，這個組件會接收到事件信息，最後使用事件信息創建或更新ProductPriceMaster的實例。

接下來，看看在OFBiz中另一個採用了From-Thru模式的實體：PartyRelationship。

### 3.3.2 示例：PartyRelationship

在OFBiz中，實體PartyRelationship（業務實體關係）也使用了From-Thru模式。



提

示在OFBiz中的Party指的是什麼？

事實上，在OFBiz中，Party這個概念是一個很強大的抽象。中文有將其翻譯成當事人的，有翻譯成團體的，也有翻譯成會員的。其實它是所有業務流程的參與者的抽象。它的概念子類型包括個人、組織、企事業單位、政府機構，甚至包括非正式的組織，比如家庭等。

個人認為把這個Party翻譯成業務實體比較貼切。其他的譯法感覺都有一些問題，比如，Party這個概念其實是可以包含“個人”的，翻譯成團隊感覺就比較奇怪。

PartyRelationship有如下主鍵。

·partyIdFrom：“從”業務實體ID。

·partyIdTo：“到”業務實體ID。

·roleTypeIdFrom：“從”角色類型ID。

·roleTypeIdTo：“到”角色類型ID。

·fromDate：關係的生效時間。

實體的其他屬性如下。

·thruDate：關係的結束時間。

·partyRelationshipTypeId：關係類型ID。

表3-1是一個業務實體關係的示例。

表3-1 業務實體關係示例

partyIdFrom	partyIdTo	roleTypeIdFrom	roleTypeIdTo	fromDate	thruDate	partyRelationshipTypeId
Company1	Developer1	INTERNAL_ORG	EMPLOYEE	2001-05-02		EMPLOYMENT

表3-1中第一行的意思是，Company1作為一個內部組織（INTERNAL\_ORG），Developer1作為一個僱員（EMPLOYEE），它們之間存在一個僱傭（EMPLOYMENT）關係，僱傭關係開始的時間為2001年5月2日。

## 3.4 CQRS、ES與流處理

前面提到過，很多人認為CQRS就是ES，甚至還有人認為它們同流處理（Stream Processing）是一回事。這些觀點很有意思<sup>[1]</sup>，但是對CQRS與ES的理解並不準確。

CQRS和ES並不是一回事。CQRS的核心是概念模型層面的讀/寫分離，CQRS的實現完全可以不使用ES模式，但是採用ES模式確實有助於實現CQRS。因為ES模式中事件的存在，為實現從寫模型到讀模型的數據“同步”提供了便利。這些在前文已有論述。

CQRS或ES也不是流處理的另外一個說法。“流處理是通過對事件的聚合運算得到結果——這就是ES”，這個說法也有失偏頗。

流處理的結果主要是面向分析的。DDD社區關注的是“領域的複雜性”，數據分析只是其中一部分。DDD很大程度上是關於對象生命週期管理的藝術，或者說DDD是如何對（複雜的）狀態進行有效管理的技巧。ES和CQRS興起於DDD社區，它們的基因與流處理大不相同。它們的關注點不只在於OLAP（在線分析處理），更在於OLTP（在線事務處理）。

**OLTP**應用關注的是當前狀態以及命令。注意，命令和事件不是同一個概念，只有合法的、被系統按照業務規則接受的命令，才會被轉變為事件，這是它們之間微妙但是重要的區別。合法的命令（下一個事件）不能違反當前狀態與業務規則的約束。

而那些使用流處理進行數據分析的互聯網應用很多時候只關注事件，而不關注命令。比如，**Google Analytics**（谷歌分析）只是被動地記錄“點擊”事件——那些已經發生的事實，它並不限制你“在點擊了鏈接A之後不能點擊鏈接B”。

我們很可能在開發**OLTP**應用的時候會使用**ES**或**CQRS**模式。

當我們使用**ES**模式時，服務端在接收到命令（請求）之後，做的第一件事情就是獲取或恢複相關實體的當前狀態——用於表示這個狀態的數據結構，在**CQRS**中這一般會被歸為命令模型（寫模型），並不會認為是查詢模型（讀模型）的一部分。它極其重要，是領域真正的核心模型，一般來說它應該是高度遵循數據庫設計範式的。

**ES**模式通過事件的聚合運算得到當前狀態，其主要目的不是為了分析，而是為了決定接下來發生的命令是否合法、能不能轉變成事件。比如，某個賬戶的當前餘額（當前狀態）決定了能不能從它裡面取出錢來，以及能從它裡面取出多少錢來。

甚至當我們運用CQRS模式，客戶端從讀模型一側獲取數據，並將數據呈現出來的時候，用戶“看著這些數據”，很多時候並不是為了進行數據分析，而是為了進行事務處理，這時候大家關注的仍然是當前狀態。

而進行流處理時，你基本不會想到CQRS模式中寫模型（領域的核心模型）和讀模型之間的區別和聯繫，也不會關注命令與事件這兩個概念的微妙區別，你想做的可能只是事後的分析。就算CQRS或ES與流處理在技術實現上存在一些表面的相似之處，僅僅是意圖不同就足以對它們做出區分。

綜上，我們很難接受“CQRS或ES模式只是流處理另外的一個說法”這個觀點。

[1] Martin Kleppmann. Making Sense of Stream Processing, <https://www.confluent.io/blog/making-sense-of-streamprocessing/>。

## 第二部分 設計

- 第4章 DDD的DSL是什麼
- 第5章 限界上下文
- 第6章 值對象
- 第7章 聚合與實體
- 第8章 超越數據模型
- 第9章 模式

## 第4章 DDD的DSL是什麼

之前的章節已經闡述了什麼是DDD的領域模型，以及DDD對領域模型的期望：既能反映對領域（業務）的認知，又能直接用於指導軟件的設計與實現。

既然按照Eric Evans的觀點，領域模型不是一幅具體的圖，而是那幅圖想要傳達的某個思想，那麼用一幅圖、一段精心編寫的代碼、一段自然語言去描繪和傳達一個模型，還不夠嗎？為什麼非得造一個領域專用語言（Domain-Specific Language，DSL<sup>[1]</sup>）出來呢？

什麼是DSL？維基百科上是這麼說的：

領域專用語言（DSL）是面向特定的應用領域專門化的計算機語言。它與通用語言（GPL）相對應，後者廣泛適用於不同的領域。

本章首先會探討為什麼在實現DDD（領域驅動設計）時使用DSL是有意義的，以及如果要設計一個DDD的DSL我們需要注意什麼問題。然後，正式介紹由筆者設計的一個DDD原生的DSL：領域驅動設計建模語言（Domain-Driven Design Modeling Language，DDDML）。DDDML的核心是一個文檔對象模型（DOM），DDDML文檔可以使用YAML、JSON或其他標記語言來編寫。為方便讀者閱讀，本書展示

的DDML示例代碼默認使用YAML這種對人類友好（相對於JSON具備更高的可讀性）的標記語言來編寫。

為了方便後文的敘述，本章會整理一個DDML的詞彙表，介紹一下在設計DSL及相關工具的過程中使用的一些術語。最後，通過一個Car聚合的模型示例，讓讀者一瞥DSL的大致面貌。

[1] 見 [http://en.wikipedia.org/wiki/Domain-specific\\_language](http://en.wikipedia.org/wiki/Domain-specific_language)。

## 4.1 為什麼DDD需要DSL

DDD的創始人Eric Evans曾經在一次訪談中說：

更多前沿的話題發生在領域專用語言（DSL）領域，我一直深信DSL會是領域驅動設計發展的下一步。現在，我們還沒有一個工具可以真正給我們想要的東西。但是人們在這一領域比過去做了很多的實驗，這使我對未來充滿了希望。

為什麼DDD需要DSL？在回答這個問題之前，先問一個問題：為什麼在現實世界中實現DDD（落地）那麼困難？

### 4.1.1 為什麼實現DDD那麼難

早在2004年，Eric Evans就發表了他最具影響力的經典著作《領域驅動設計：軟件核心複雜性應對之道》。時至今日，DDD在開發領域早已深入人心，它是公認的解決軟件核心複雜性的“大殺器”。但是，一直以來，大家都對在軟件開發過程中實踐DDD心存畏懼，認為那是需要付出相當大成本的。

是因為DDD的概念過於抽象，所以大多數人對DDD理解不深刻嗎？是，也不是。

誠然，Eric Evans的經典著作以抽象、凝練著稱，可說是字字珠璣，初次讀到就能產生強烈共鳴的恐怕只有少數，更多初讀者可能會在心裡嘀咕：領域模型這玩意兒有什麼用啊？我們在開發過程中好像都沒有見過，活不是照樣幹嗎？產品人員交給開發的PRD（Product Requirement Document）、UI設計師出的效果圖，哪個是“領域模型”？

“按照Eric Evans的觀點，領域模型不是一幅具體的圖，而是那幅圖想要去傳達的那個思想。”談思想是不是有點太“玄”了，能不能來點兒實際的東西？

產品經理心裡想的可能是：我們做的產品設計，經常被開發人員抱怨“難以理解”“不可實現”。領域模型對我的工作有何幫助？我到底要拿出一個怎樣的領域

模型，開發人員才能理解？我要做到什麼程度，才好意思告訴大家“我的工作已經做到位了”？我怎麼判斷我做出來的設計到底是不是可以實現？

系統分析師可能會想：理論上，把所謂的分析模型換成領域模型，對分析結果的落地應該有幫助，但是，有了漂亮的模型就夠了嗎？我覺得關鍵還是要能將分析結果映射到實現代碼中啊！我以前又不是沒有出過漂亮的分析結果，但是開發那幫人好像寫代碼的時候完全不按我想的來……

技術管理者可能會想：這玩意兒看起來好像不錯，但是開發人員，特別是那些新手，能不能理解？能不能用得上、玩得轉？如果他們玩不轉，“擦屁股的”還不是我？

項目經理可能會想：搞這個玩意兒會不會很費勁啊？我們需要多少資源才能把這東西弄好？聽說這東西在很多公司都玩不下去呢……

你看，大家的困惑和問題如此之多：

- 作為一種思想，領域模型好像太抽象，難以把握，它能不能更具象化一些？
- 怎麼度量系統分析、領域建模的工作做到位了？能不能提供一些清晰的、嚴格的工作產出結果的標準？

·能不能圖形化展示領域模型？在領域建模之後馬上給大家一個可以運行的、帶UI的軟件？

·怎麼保證代碼忠實地反映了分析結果（領域模型）？畢竟，如果分析結果不能映射到實現代碼中，那麼分析有何意義？

·DDD能不能讓各個層次的技術人員各司其職、各展所長？

·將領域模型忠實地映射到實現代碼需要的工作量如何？成本是不是我們團隊可以承受的？

所以，現實情況就是，一直以來，領域驅動設計都快成為資深技術人員的“不傳之秘”了……是不想傳嗎？並不是，我覺得還是太難傳。

軟件工程進化到今天，我們已經擁有如此多的工具，有數百種編程語言，數不清的軟件類庫、框架，我們熟知OOP、函數式編程等各種編程範式，我們掌握了“敏捷”的軟件開發方法，我們創造了各種管理軟件開發過程的應用……在這些工具中，難道我們還找不到一把可以搞定DDD的“錘子”？

## 4.1.2 搞定DDD的“錘子”在哪裡

曾經有一個Java DDD開發框架的Quickstart Guide文檔是這麼說的：

不像很多其他框架，“我們的框架”不是為了“讓你快速開始”而構建的，它是為了讓你可以長期使用而構建的。

讀到這裡，你可能心裡會想：在大家都追求敏捷的今天，這有點“反動”了吧？再說，軟件開發的快速開始和長期使用難道真的是魚與熊掌——是不可調和的矛盾嗎？

目前諸多“現代”Web應用開發框架的鼻祖是RoR ( Ruby on Rails )。Eric Evans曾在訪談中對RoR讚賞有加。當初正因為RoR具有令人驚豔的開發效率，所以極大地推動了Ruby語言的流行。雖然隨著各個通用編程語言模仿RoR的現代Web開發框架，Ruby語言以及RoR的熱度這幾年已經明顯下去了，但仍然有很多創業公司會先使用RoR來開發一個單體應用，等用戶規模上去後再對單體應用進行拆分和重構。雖然Rails本身不是不能做松耦合的應用，但是大部分RoR單體應用的各個軟件構造塊之間的耦合關係確實很強。我們可以在網上搜索到一些如何在RoR框架下實踐DDD的文章<sup>[1]</sup>，但是顯然大家都已注意到在許多方面RoR

這樣的Web開發框架和主張“分而治之”的DDD之間存在難以調和的矛盾。

筆者的觀點是，想要高效地實踐DDD，不能把希望寄託在RoR這樣並非基於DDD理念設計的開發框架上。試圖將RoR和DDD調和在一起，使用DDD的方法論來指導基於RoR的應用軟件開發，事倍功半。網上總結的那些做法並不是不好，也不是絕對不可行，但是基本只能適用於非常“好”的那部分程序員。因為一個框架（比如RoR）如果不建立約束，想讓開發人員（特別是初階的開發人員）不越過約束（做蠢事）是不可能的。靠傳授各種需要注意的編碼規範、技巧、最佳實踐來實踐DDD，希望初階的開發人員在開發過程中能時刻保持頭腦清醒，將程序一板一眼地執行出來，這實際上非常困難。

除了框架，越來越多原來只知道OOP的開發人員已經開始瞭解和掌握像“函數式編程”這樣“新的”編程語言範式。但是DDD領域模型其實是基於OO模型的，它是關於狀態管理的藝術，函數式編程可以給我們在DDD的部分實現細節上提供啟發和幫助，但是我們不能躲在無狀態的“純函數”的世界裡，迴避真正複雜的狀態管理問題。

歸根到底，基於現有的錘子——工具——去實踐領域驅動設計的問題，在於它們抽象的層次都偏低。比如，現今大多數開發人員都熟悉OO程序設計思想，且掌握了一兩門通用OO語言，雖然DDD的分析和建模方

法是OO範式的，但是聚合、實體、值對象、領域服務等都是更高抽象層次的對象，而通用OO語言缺乏這樣的概念。

另外，要想一個領域模型有用，它就必須足夠嚴格，可以成為編寫代碼的基礎。當我們使用自然語言的句子進行交流時，往往難免隨意、潦草。

所以要想玩轉DDD，我們需要一個DDD原生的工具：它應該是一門DDD建模語言，可使用、強化DDD那些更高抽象層次的概念，可以嚴謹地、“原汁原味”地將DDD領域模型（那個“思想”）表述出來。然後，基於這些領域模型的表述，我們應該能將模型忠實、快速地映射到代碼中；並且在修改、擴展這些代碼的過程中，我們應該為開發人員提供足夠的“約束”，儘可能地讓開發人員在寫代碼的時候不幹出“蠢事”來，尤其是不能破壞代碼與模型之間的映射關係。

說到這裡，也許很多有經驗的開發人員都會想到：這個問題的終極答案可能是DSL。我們可以設計一門DDD的DSL，然後使用DSL描述領域分析和建模的結果——領域模型，並使用軟件工具從DSL文檔生成與領域模型存在映射關係的代碼。

[1] Victor Savkin. DDD for Rails Developers, <https://www.sitepoint.com/ddd-for-rails-developers-part-1-layered-architecture/>。

## 4.2 需要什麼樣的DSL

如果能理解4.1節所述，其實很大程度上已經知道我們需要什麼樣的DSL了。

我們想要一個DSL，它能夠描述DDD風格的領域模型。這個DSL允許我們在一個地方集中記錄和展示領域模型中的關鍵元素。這個DSL應該能在概念層面描述領域模型，也允許我們在其中添加領域模型在實現層面需要的細節。我們希望它能夠支持代碼生成工具以產生與領域模型之間具備親密映射關係的軟件代碼。我們希望軟件的文檔——包括使用Swagger/OpenAPI<sup>[1]</sup>、RAML描述的RESTful API文檔，定義數據庫Schema的DDL代碼，領域模型中關鍵狀態的狀態機圖等——都儘可能自動化地產生。我們希望生成所謂的Client SDK，我們甚至希望能直接生成有UI的客戶端應用，至少生成一些程序員在實現客戶端應用的UI/UE（用戶體驗）時可以使用的腳手架代碼……

除了這些，下面的問題則是筆者在設計DDD的DSL時考慮得比較多的。

[1] 見 <https://swagger.io/specification/>。

## 4.2.1 在“信仰”上保持中立

“我們需要設計一個DDD的DSL。”

當我開始認真地考慮這件事情時，正身處一個超過200人的技術團隊中。當時的背景自然不可避免地影響著我對這個DSL的思考和設計。

當時我所在的公司雖然在技術部門內也一直強調技術要為業務服務，但是很多時候這並不能阻止技術人員的“自嗨”。事實上，就是技術團隊內幾乎也沒有人關注軟件的概念完整性，大家更關注的是在其中使用了什麼時髦的技術。幾乎沒有什麼文檔記錄軟件設計——要“敏捷”嘛。很多應用經過反覆修改後變得複雜而脆弱，模型混亂，技術債務高疊。對於這些軟件的代碼，不少人最愛說的是：“寫得這麼爛，還不如重新寫一個。”當然，說這句話的人並不關心代碼“寫得這麼爛”的根本原因。至於讓他/她重新寫一個，過一段時間後會不會也同樣“變爛”，就更不是他們感興趣的了。

當時公司的應用軟件基本都是基於三個平臺開發的：**PHP**、**Java**和**.NET**。技術人員之間因為語言和工具的“信仰”問題存在很深的隔閡。

很多技術人員不願意走出舒適區，習慣於使用自己熟悉的語言和工具來完成任務。以至於其他團隊已

經做過的東西，非要換成“我最愛”的語言和工具再做一遍。就像俗話說的，“手裡拿了個錘子，看什麼都是釘子”，由此產生了大量的重複建設。

所以，這個DDD的DSL有必要在對技術人員至關重要的“信仰”問題上保持中立，使技術團隊能將更多的注意力轉移到要服務的“領域”上。

更進一步，希望基於這個DSL可以製造一個工具鏈，最終形成一個跨語言的、可以支持大型業務軟件開發的平臺（PaaS）。這個開發平臺對開發人員應該提供足夠的約束，讓他們少幹傻事，少幹超出自己能力範圍的事情。

## 基於開放標準的標記語言

一般來說，DSL是文本化（而非UML那樣的圖形化）的語言，以文本化的表示方式儘可能地將領域分析與建模的成果記錄在同一組DSL文檔中，以減少模型表述的不一致。

基於DDD的DSL應該選擇基於開放標準的標記語言——比如YAML<sup>[1]</sup>、XML<sup>[2]</sup>等——來描述領域模型。開放的標準便於利用現有的工具以及實現各種工具之間的集成。我們可以在此基礎上製作工具軟件，實現對領域模型的可視化呈現、編輯，生成特定的編程語言的應用代碼等。

[1] YAML is a human friendly data serialization standard for all programming languages, <http://yaml.org/> .

[2] Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable , <https://en.wikipedia.org/wiki/XML> .

## 4.2.2 DDD原生

很多人可能都想問這個問題：設計這樣一個DSL是不是又在“重新發明輪子”？

回答：當然不是。因為我們想要做的，是一個“DDD原生”的DSL，應該說這還算是一個新事物。

我們想要在對領域進行分析和建模時就使用這個DSL，用那些“原汁原味”的DDD概念來描述領域模型，然後用它來約束此後的編碼實現。這個過程首先是需要以DDD的理念設計DSL的規範，然後還需要製造合適的工具，用於將DSL描述的模型映射到代碼中。

正如前文所言，複雜軟件的開發理應自頂而下，應該先概念建模再物理建模。但是太多的軟件開發團隊是沒有人做概念建模的：開發人員對於自己本職工作的理解，就是先做數據庫設計（物理建模），然後開始編寫業務邏輯代碼。當新成員加入團隊，想了解系統的設計時，只能去看數據庫。

堅持“DDD原生”可以避免讓我們淪為數據庫CRUD（Create/Retrieve/Update/Delete）程序員，已經有太多工具可以為我們生成CRUD代碼。比如數據庫反轉引擎之類的工具可以從數據庫（這裡的數據庫主要是指關係型數據庫）中讀取Schema信息，然後為你生成一些特定的通用語言代碼。如果生成的目標是OOP語言

代碼，一般來說，數據庫中的表是什麼樣，生成的實體（對象）就是什麼樣。也就是說，表和實體（對象）、列和屬性一般呈現出一一對應的關係。

還是舉個例子吧。設想，在數據庫中有如表4-1所示的這樣一張表（假設表名為Cross）：

表4-1 一個隱含著“點”和“線”對象的表

Id	Line1_Point1_X	Line1_Point1_Y	Line1_Point2_X	Line1_Point2_Y	...
X1	1.1	8.9	...	...	...



示看到這裡，聰明的你可能會從這個表和列的命名去“倒推”領域模型中的概念，你想：也許，在這個領域裡面，可能存在兩個值對象，分別表示二維平面上的“點”和“線”？

你當然是對的。

我們使用數據庫反轉引擎生成的代碼很可能像下面這樣（類Java的偽代碼）：

---

```
public class Cross {  
    String Id;  
    //...  
    float Line1_Point1_X;  
    float Line1_Point1_Y;  
    float Line1_Point2_X;  
    float Line1_Point2_Y;
```

```
//...  
}
```

---

但是，從領域模型出發，你想要的代碼可能是如下形式：

---

```
public class Cross {  
    String Id;  
    //...  
    Line Line1;  
    //...  
}  
  
public class Line {  
    Point Point1;  
    Point Point2;  
}  
  
public class Point {  
    float X;  
    float Y;  
}
```

---

也就是說，數據庫反轉引擎生成的代碼，丟失了“點”和“線”這兩個可能是很重要的領域概念。

可能有人會問：“這算什麼大問題？”

如果我們先設計物理數據模型，然後給開發人員提供數據庫反轉引擎生成的CRUD代碼，讓他來修改，要求體現出領域模型中的概念，那麼開發人員顯然是需要做一些代碼的重構工作的。如果不改，那麼在實現代碼中就丟失了那些重要的領域概念，代碼就失去

了與領域模型之間的映射關係。雖然這只是一個簡單的例子，但是聰明的你應該可以想象到，在一個複雜的大型軟件開發中，這樣的小問題最終可能會累積成大問題。比如，如果有了**Line**和**Point**，我們就可以在這兩個類的方法中實現很多可以共用的對“線”和“點”的操作邏輯；如果沒有這兩個類，那麼這些方法的邏輯很可能會被不同的開發人員重複“編寫”很多次，散落在代碼的各個角落。

### 4.2.3 在複雜和簡單中平衡

設計一個DDD的DSL，我們需要考慮它對領域模型的表達能力：可以描述什麼，以及不能描述什麼（或沒有必要描述什麼）。

想讓DSL描述的東西越多，表現力越豐富，DSL的設計就會越複雜。所有人都討厭複雜性，引入複雜性需要有充分的理由。

比如，很多編程語言都有內置的基本類型，而我們在設計DDD的DSL時決定不預設任何基本類型，這樣做的目的是鼓勵大家儘可能地捕獲領域中的值對象，優先使用這些值對象來構建模型，而不是使用int、byte、long、float、string這樣的技術術語。這其實給DSL的設計和工具的實現在某些方面增加了複雜性，但是我們認為這是值得的。畢竟，我們一直希望的不就是讓大家更多地關注領域本身嗎？

有時候我們也會堅持“簡單”。

比如，我們決定讓這個DDD的DSL僅支持描述唯一的一種實體間的基本關係，即同一個聚合內，從上一級實體指向其直接關聯的下一級實體的導航關係——OO模型需要把這樣一對多的關係體現為類型為“實體的集合”的屬性。

另外，我們還決定，在描述對象的狀態時，這個DSL僅僅支持**Set**語義的集合，而不支持使用**List**、**Map**語義的集合。包括上面提到的類型為“實體的集合”的屬性，以及類型為“值對象的集合”的屬性，這些集合都是**Set**語義的。

同時，我們需要謹記，如果為了簡單，這個DSL決定不支持某個特性，那麼在碰到非解決不可的問題時，它還要能提供“變通”( *Workaround* )的解決方案。

## 避免過度抽象

抽象化（簡稱抽象）是有代價的。抽象化會帶來學習的成本，也就是說，雖然抽象用起來很好，但是使用抽象仍然有必要“知其所以然”。所謂“抽象洩漏”，就是指軟件開發的過程中，本應隱藏實現細節的抽象化不可避免地暴露出底層細節與侷限性。既然抽象總是要洩漏的，那麼在做抽象時我們有必要衡量一下抽象的代價。有時候我們可以選擇取消某些抽象，讓大家直接使用更“原始”的概念來描述模型。比如說，**Set**、**List**、**Map**是不同語義的集合，那麼可以通過引入中間對象，使用**Set**來模擬實現**List**或**Map**的功能。

來看一個例子。NBA賽季“最佳球員”的排名是一個列表 ( *List* )，表現越好的球員應該出現在列表中越靠前的位置。可能一開始，我們為這個模型編寫了類似如下的代碼 ( 類Java偽代碼 )：

---

```
// NBA 賽季
class NbaSeason {
    String Year;
    // “最佳球員”排名列表
    List<Player> TopPlayers;
}

class Player {
```

---

但是，如果我們沒有**List**語義的集合，只有**Set**語義的集合可用呢？其實可以通過增加一箇中間對象，使用**Set**來模擬**List**的效果，示例如下（類Java偽代碼）：

---

```
// NBA 賽季
class NbaSeason {
    String Year;
    // 最佳球員
    Set<TopPlayersItem> TopPlayers;
}

// “中間對象”
class TopPlayersItem {
    int Position;
    Player Player;
}

class Player {
```

---

在上面的代碼中，中間對象**TopPlayersItem**的屬性**Position**用於記錄球員在列表中的位置。

#### 4.2.4 通過DSL重塑軟件開發過程

如果有一門DDD的建模語言，我們期望它能覆蓋從運用DDD方法進行領域建模，到將領域模型映射到代碼的所有需求。那麼，對於DDD的DSL，我們希望使用它之後能達成什麼樣的效果呢？

事實上，我們希望通過這個DSL可以重塑軟件的開發過程：

- 這個DSL可以幫助我們提高軟件分析的效率和質量。
- 我們可以使用這個DSL來準確地描述分析和建模的結果——DDD領域模型。
- 通過使用代碼生成等程序設計自動化工具，我們可以得到從DDD領域模型映射而來的實現代碼。這些代碼不應該只是一個Demo，應該可以擴展並用於生產環境。
- 應該可以對DSL描述的領域模型做圖形化的展現。最好還可以從DSL一鍵生成能馬上運行起來、有UI的軟件。領域專家（業務人員）可以通過UI確認領域模型/概念模型是否正確、合理、可用。

- 以上過程應該非常敏捷。當發現模型有問題的時候，整個過程可以快速從頭再來。

- 領域模型初步得到領域專家以及整個團隊的確認後，生成的代碼要讓初階的開發人員也可以即刻開始擴展、實現業務邏輯。

如果有一個DSL能夠實現上面描述的願景，那麼大家可能都會贊成Eric Evans所言，即“DSL會是領域驅動設計發展的下一大步”。

## 4.3 DDDML——DDD的DSL

領域驅動設計建模語言（Domain-Driven Design Modeling Language · DDDML）是筆者設計的一個DDD的DSL。

DDDML的規範到目前為止幾乎是由筆者個人獨立設計的。圍繞著這個DSL，筆者以及筆者曾經工作過的團隊中的一些同事打造了一個工具鏈（我們把這個工具鏈稱為**DDDML Tools**），以保證領域模型可以被忠實地映射到軟件的實現代碼中。雖然其他人也參與了部分**DDDML**工具的製作，但本書展示的那些由工具生成的代碼，如果沒有特殊說明，都來自筆者自己製作的工具。

接下來的章節將介紹**DDDML**的規範，以及在設計**DDDML**與**DDDML Tools**的過程中我們（筆者以及參與過**DDDML**工具製作的其他人）的思考、討論與決定。不過，在此之前，有必要先介紹一下我們在工作中使用的一個關於**DDDML**的詞彙表，這樣後面使用這些術語時會比較方便。

### 4.3.1 DDDML的詞彙表

#### 1. DDDML DOM

正如DDD的領域模型其實是一個“思想”，DDDML的靈魂並不是以某種標記語言編寫的文檔，而是一個可以使用標記語言來表述的“思想”，即一組抽象的數據結構。

DDDML的核心，是一個我們稱為DDDML文檔對象模型（ Document Object Model，DOM ）的樹結構。

我們規定，這個抽象的數據模型必須可以使用對機器友好的JSON<sup>[1]</sup>來表述。讀者可以通過瀏覽JSON.org網站的首頁快速地瞭解JSON的一些基礎知識。

因為YAML可以看作JSON的超集，且YAML對人類來說具備更好的可讀性，所以我們在實踐中更多地使用了YAML而非JSON來描述DDDML領域模型。如果有必要，很容易把這些YAML轉換為等價的JSON表述，反過來也一樣。

雖然我們在實踐中主要使用YAML來描述DDDML領域模型，但我們應該知道這只是DDDML DOM這種數據結構的表述方式。實際上使用JSON、XML、TOML<sup>[2]</sup>或者其他標記語言，甚至自行創建一種全新的

DSL來表述這個DOM都是可行的，只要這種語言與JSON具備同樣的能力，這實際上並不是一個很高的要求。

## 2. 借鑑自JSON的概念

如前所述，我們要求DDDML DOM必須可以使用JSON來表述，所以在討論DDDML規範的時候，所使用的很多概念與JSON定義的同名概念具有相同的含義。比如，當我們提到string、number、integer、true、false、null、value等時，它們與JSON中的同名概念所指一致。

但是，考慮到JSON中使用的一些術語，比如Object、Array在軟件開發領域實在是用得太“濫”了，為了避免混淆，我們決定對這部分JSON概念的名字進行替換。

## 3. 結點

既然DDDML DOM是一個樹結構，那麼我們將這個結構中的元素稱為結點（Node）就是很自然的事情了。

為了選取結點，我們需要使用路徑。比如，對於下面這個YAML文檔：

---

```
aggregates:  
  Car:
```

```
# ...
properties:
  Tires:
    itemType: Tire
entities:
  Tire:
    # ...
    properties:
      Positions:
        itemType: Position
entities:
  Position:
    # ...
    properties:
      TimePeriod:
        type: TimePeriod
  MileAge:
    type: long
```

---

我們可以使  
用/aggregates/Car/entities/Tire/entities/Position/prop  
erties/MileAge這樣一個路徑，來選取上面的YAML文  
檔倒數第二行的那個結點——這是一個“名/值對”的名  
稱結點。對XPath有所瞭解的讀者應該很熟悉這種描述  
路徑的方式。

## 4.Map

JSON的Object是一個無序的名/值對的集合。在  
DDDDML DOM中，我們把在概念上等同於“JSON的  
Object”的結點的類型稱為Map。

對於Map中的元素，相對於JSON的Object的名/值  
對，我們在更多的時候稱之為鍵/值對 ( key/value )

pair ) 。

如果DDML規範需要規定一個Map中鍵值對的類型，比如，我們要求一個Map中的值的類型必須是整數，這時就會把這個Map的類型記為：

`Map<String, Integer>`。當然，正如JSON中的Object，在DDML的Map中，Key ( 名稱 ) 的類型其實總是 `string` ( 字符串 )，真正允許變化的只是Value的類型而已。

在Map中的Key結點有一個對應的Value ( 值 ) 結點，這個Value結點內部當然可能還存在子結點，比如說它自己也是一個Map。在本書的行文中，我們經常會說某個結點在/`{PATH}`/`{TO}`/`{KEY_NODE}` ( 某個Key結點的路徑 ) 結點下，或者說

在/`{PATH}`/`{TO}`/`{KEY_NODE}`結點中，指的是某個結點是這個"/`{PATH}`/`{TO}`/`{KEY_NODE}`"Key結點對應的Value結點或者Value內部的子結點。

## 5.Object

在DDML DOM中，當我們說一個值的類型是 Object的時候，其實是說這個值的類型可以是 `string`、`number`、`boolean`、`Map`、`List` 等。可以認為Object是所有類型的基類型，如果你熟悉C#、Java這樣的編程語言，應該很容易接受這個概念。

注意，這裡的Object與JSON的Objec含義不同，它相當於JSON的Value。

比如，對於下面這個YAML/DDDML文檔：

---

```
aggregates:
  Package:
    immutable: true
    implements: [Article]
    id:
      name: PackageId
      type: long
    properties:
      RowVersion:
        type: long
        #...
    reservedProperties:
      version: RowVersion
      createdAt: CreationTime
      noDeleted: true
```

---

當我們說  
"/aggregates/Package/reservedProperties這個結點的  
值是一個Map<String, Object>"的時候，指的是對於這  
個Map的值類型，我們沒有做什麼限制。

對於DDDML DOM的Map<String, Object>類型，  
因為它的Key總是String，而Value可以是任意類型，所  
以有時候我們直接簡稱它為Map。

## 6.List

我們都知道，JSON的Array是一個有序的值的集合。在DDDML DOM中，我們把在概念上等同於“JSON的Array”的結點的類型稱為List（列表）。

[1] JavaScript Object Notation is a lightweight data-interchange format, <http://json.org/>。

[2] Tom's Obvious, Minimal Language, <https://github.com/toml-lang/toml>。

### 4.3.2 DDDML的Schema

我們可以把DDDML的規範體現為JSON Schema<sup>[1]</sup>。在如下網址中放了一個不太完整的DDDML的JSON Schema：

<https://github.com/wubuku/dddml-spec/blob/master/schemas/dddml-schema.json>。

在網上讀者可以找到利用JSON Schema來校驗JSON或YAML文檔的工具。

如果你使用的編輯器支持，可以設置一下，讓JSON Schema在編輯JSON文件時起作用。如果你的編輯器有合適的插件，JSON Schema對YAML文件可能也會起作用。

比如，如果你使用的是VS Code編輯器，可以在工作區目錄下的“.vscode/settings.json”文件中做如下設置：

---

```
{  
  "json.schemas": [  
    {  
      "fileMatch": [  
        "/*.json"  
      ],  
      "url": "PATH\\TO\\THE\\dddml-schema.json"  
    }  
  ]  
}
```

---

有了JSON Schema以及支持它的編輯器之後，  
DDDML文件寫起來可能會輕鬆不少。

雖然DDDML文檔一般都是使用YAML格式來編寫的，但是其實用JSON也可以。很多YAML序列化庫一樣可以解析JSON。還有很多工具可以實現YAML和JSON的轉換，在網上通過關鍵字“yaml-to-json”“json-to-yaml”搜索可以找到它們。

DDDML的核心其實是DOM（文檔對象模型），如果我們想要支持用XML來表述DDDML的DOM也不是什麼難事，顯然可以使用XSD（XML Schema Definition）<sup>[2]</sup>來描述一個基於XML的DDDML文檔應有的樣子。

[1] JSON Schema is a vocabulary that allows you to annotate and validate JSON documents, <http://json-schema.org/>。

[2] XSD (XML Schema Definition), [http://en.wikipedia.org/wiki/XMLSchema\\_\(W3C\)](http://en.wikipedia.org/wiki/XMLSchema_(W3C))。

## 4.4 DDDML示例：Car

在《領域驅動設計：軟件核心複雜性應對之道》一書的中文修訂版<sup>[1]</sup>的6.1節中，有一個汽車維修廠使用汽車模型的例子，剪裁這個例子，以DDDML表述，形式如下：

---

```
aggregates:

Car:
    # aggregateRootName: Car
    id:
        name: Id
        type: string

    properties:
        Description:
            type: string
        Wheels:
            itemType: Wheel
        Tires:
            itemType: Tire

    methods:
        # -----
        Rotate:
            eventName: TireWheelPairsRotated
            parameters:
                # 4 tire/wheel ID pairs
                TireWheelIdPairs:
                    itemType: TireWheelIdPair

        # -----
        entities:

        # -----
        Wheel:
```

```
    id:
        name: WheelId
        type: WheelId

    # -----
    Tire:

        id:
            name: TireId
            type: string
            arbitrary: true

        properties:

            Positions:
                itemType: Position

    # -----
    entities:

        # -----
        Position:
            id:
                name: PositionId
                type: long
                arbitrary: true

            properties:
                TimePeriod:
                    type: TimePeriod
                MileAge:
                    type: long

    # -----
    WheelId:
        type: WheelId
        referenceType: Wheel

valueObjects:
    # -----
    TimePeriod:
        properties:
            From:
                type: DateTime
            To:
```

```
        type: DateTime
# -----
TireWheelIdPair:
    properties:
        TireId:
            type: string
        WheelId:
            type: WheelId

enumObjects:
# -----
WheelId:
    baseType: string
    values:
        LF:
            description: left front
        LR:
            description: left rear
        RF:
            description: right front
        RR:
            description: right rear
```

---

在上面的例子中，我們定義了一個叫作**Car**的聚合。這個例子中聚合的名稱與聚合根的名稱一樣，都是**Car**。

**Car**聚合根（實體）的屬性中包含一個名稱為**Wheels**的屬性，它的類型是車輪的集合（**itemType:Wheel**）；聚合根還包含一個名稱為**Tires**的屬性，它的類型是輪胎的集合（**itemType:Tire**）。車輪和輪胎都是**Car**聚合內部的實體。**Position**實體是輪胎（**Tire**）的下一級實體，它用於記錄輪胎什麼時候安裝在什麼位置（即哪個車輪上）、行駛了多少里程。

我們在這個聚合（聚合根）中定義了一個方法 **Rotate**，這個方法對車輪與輪胎進行“輪換”。如果這個方法執行成功，就會產生一個事件：**TireWheelPairsRotated**。

也許有人會認為應該在 DDDML DOM 的聚合定義內再加一個層級，把聚合根的定義單獨作為一個結點體現出來，如下所示（注意增加的 **/aggregates/Car/root** 結點）：

---

```
aggregates:
  Car:
    root:
      name: Car
      id:
        name: Id
        type: string

    properties:
      Wheels:
        itemType: Wheel
      Tires:
        itemType: Tire
    # ...
```

---

我們確實考慮過這個寫法，但是最後並沒有採用，因為在大多數時候聚合以及聚合根的名稱都是相同的；在很多時候聚合內只有聚合根一個實體。這樣做之後，在類似 **/aggregates/Car** 這樣的聚合名稱結點下，大多數時候除了 **root** 這個結點不會再有其他直接的子結點——實在不喜歡這種寫法帶來的更多縮進，當然還有其他理由，我們後文再談。

[1] 埃裡克·埃文斯 ( Eric Evans ) . 領域驅動設計：軟件核心複雜性應對之道 ( 修訂版 ) . 人民郵電出版社, 2016.

#### 4.4.1 “對象”的名稱在哪裡

這裡所說的“對象”只是泛指那些“有名字的東西”，包括DDD的實體（引用對象）、值對象、屬性、方法等。

在DDML中，對於這些“有名字的東西”，都需要在一個Map（也就是JSON的Object）類型的DOM結構中定義。這個Map中元素（鍵/值對）的Key就是這些“東西”的名稱。

因為Map的Key就是對象的名稱，所以就不需要在Map的Value結構中使用“name”再聲明一遍了。

#### 4.4.2 使用兩種命名風格：camelCase與PascalCase

細心的讀者可能已經注意到，在這個DDDML文檔中，有些Map（也就是JSON的Object）的Key是以camelCase風格命名的，有些則是以PascalCase（大寫字母開頭）風格命名。這其實是有意為之。一般來說，我們在DDDML規範中定義的那些需要在特定的位置出現、具有特定含義的Key，都以camelCase風格命名，我們把這樣的Key叫作DDDML的關鍵字。比如，對於上面示例DDDML中存在的一些結點的路徑，我們把關鍵字部分加粗顯示，示例如下：

·/aggregates

·/aggregates/Car/id

·/aggregates/Car/entities

而對於在DDDML規範定義的關鍵字之外的那些對象的名稱，強烈建議使用PascalCase風格命名。比如上面示例DDDML中的一些結點的路徑，我們把對象的名稱部分加粗顯示，示例如下：

·/aggregates/Car

·/aggregates/Car/properties/Wheels

·/aggregates/Car/entities/Wheel

因為我們有意選擇了不同的命名風格，所以當我們通過肉眼閱讀**DDDML**文檔的時候，就很容易從中區分出哪些是**DDDML**的關鍵字，哪些是文檔描述的當前領域模型中的概念。

### 4.4.3 為何引入關鍵字itemType

在上面DDDML的示意代碼中，有四個地方出現了關鍵字itemType，下面列出其中兩個地方：

---

```
aggregates:
  Car:
    # ...
    properties:
      # ...
      Wheels:
        itemType: Wheel
        # ...
    methods:
      Rotate:
        # ...
        parameters:
          TireWheelIdPairs:
            itemType: TireWheelIdPair
        # ...
```

---

在聲明對象的某個屬性（property）或方法的參數（parameter）的類型是一個集合時，為什麼我們引入itemType這個關鍵字，而不是支持像“type:Set<Wheel>”或“type:TireWheelIdPair[]”這樣的寫法呢？

因為像“type:TireWheelIdPair[]”這樣的寫法會給人造成這樣的感覺：我們可以支持數組這種集合，也許還打算支持其他類型的集合？

不，我們不支持。

在需要使用集合來描述對象的狀態時，或者說，對於一個對象的屬性的類型，只支持**Set**語義的集合，**Set**集合內的元素不允許重複。一個屬性的類型如果是某種元素的集合，那麼在**DDML**中就只能用**itemType**關鍵字來聲明這個集合中元素的類型，這就是在強調不允許選擇集合的類型。

而對於一個方法的參數的類型，則只支持**List**語義的集合，**List**是允許元素重複出現的有序的集合。也就是說，在上面例子的/**aggregates/Car/methods/Rotate/parameters/TireWheelIdPairs**結點中，聲明瞭這個參數的類型是“元素類型為**TireWheelIdPair**的**List**集合”。

## 第5章 限界上下文

限界上下文定義了每個模型的應用範圍。在實踐中，我們一般會使用一個目錄來存放一個限界上下文中所有模型的**DDDML ( YAML )**文件，我們習慣把這個目錄叫作限界上下文的“項目目錄”。

一般來說，**DDDML**的工具會讀取這個項目目錄中的所有**DDDML**文件，把它們在邏輯上合併為一個**DDDML**文檔，這個文檔描述的內容被當作同一個限界上下文中的模型來處理。

每個**DDDML**文檔都必須遵循同樣的**Schema**，文檔的根結點下只允許存在少數幾個直接的子結點，本章會介紹它們都是什麼。然後，討論一下在**DDDML**中那些“有名稱的東西”的命名問題，以及**DDD**的“模塊”這一概念在實踐中可以如何運用。

## 5.1 DDDML文檔的根結點下有什麼

每個DDDML文檔都必須遵循同樣的Schema。

在所有需要歸併到同一個限界上下文的不同DDDML文檔中，除了文檔根結點以及少數幾個特殊的文檔根結點的直接子結點

(即/aggregates、/valueObjects、/enumObjects、/superObjects)之外，不允許重複出現路徑相同的結點。比如，不允許在兩個DDDML文檔中都出現/aggregates/Car結點。這是當前DDDML規範的要求，因為我們暫時還沒有完全想好不同DDDML文檔、路徑相同的結點的合併規則。

### 1. aggregates結點

在實踐中，一般會把每個聚合的定義都單獨放在一個DDDML文件中，然後以聚合的名稱為文件命名，但這不是必需的。

你可以把所有這些聚合的DDDML文件的內容直接寫(合併)在一個DDDML文檔內，即把所有的聚合都定義在這個文檔的/aggregates結點下。

### 2. valueObjects結點

可以在聚合的結點中定義和這個聚合聯繫緊密的值對象，比如在`/aggregates/Car/valueObjects`結點中定義和Car聚合聯繫緊密的值對象。

還可以在`/valueObjects`結點下定義限界上下文內公共的值對象（Value Object），示例如下：

```
valueObjects:  
  PersonalName:  
    properties:  
      FirstName:  
        type: string  
        description: First Name  
        length: 50  
      LastName:  
        type: string  
        description: Last Name  
        length: 50
```

### 3.enumObjects結點

可以認為枚舉對象只是一種特殊的值對象。限界上下文內公共的枚舉對象（Enum Object）可以像下面一樣定義在`/enumObjects`結點下：

```
enumObjects:  
  DocumentAction:  
    baseType: string  
    values:  
      Draft:  
        description: Draft  
      Complete:  
        description: Complete  
      Void:
```

```
        description: Void
Close:
        description: Close
Reverse:
        description: Reverse
```

---

## 4.typeDefinitions結點

在值對象中，還有一類作為一個限界上下文的領域基礎類型來定義的值對象，它們一般會放在單獨的文件裡，定義在/**typeDefinitions**結點下。示例如下：

---

```
typeDefinitions:
date-time:
    sqlType: DATETIME
    cSharpType: DateTime?
    javaType: java.sql.Timestamp
currency-amount:
    sqlType: DECIMAL(18,2)
    cSharpType: decimal?
    javaType: java.math.BigDecimal
id:
    sqlType: VARCHAR(20)
    cSharpType: string
    javaType: String
email:
    sqlType: VARCHAR(320)
    cSharpType: string
    javaType: String
decimal:
    sqlType: DECIMAL(18,6)
    cSharpType: decimal?
    javaType: java.math.BigDecimal
Money:
    javaType: org.joda.money.Money
    cSharpType: MyMoneyLib.Money
```

---

## 5.configuration結點

在DDXML文檔根結點下可能還有一個很重要的直接子結點/**configuration**，整個限界上下文的全局配置信息都保存在這個結點下。接下來這一節就介紹它。

## 5.2 限界上下文的配置

限界上下文的全局配置 ( Configuration ) 信息都保存於 /configuration 結點下。一般來說，我們會把 /configuration 單獨保存為一個文件，並稱這個文件為限界上下文的 DDDML 配置文件。

以下是一個限界上下文的 DDDML 配置文件的例子 ( 有刪節 ) :

---

```
#%DDDML 0.1
---

configuration:
    boundedContextName: "Dddml.Wms"
    defaultModule: "Dddml.Wms"

    defaultReservedProperties:
        active: Active
        createdBy: CreatedBy
        createdAt: CreatedAt
        updatedBy: UpdatedBy
        updatedAt: UpdatedAt
        deleted: Deleted
        version: Version

    # -----
    accountingQuantityTypes:
        decimal:
            zeroLogic:
                Java: "BigDecimal.ZERO"
            addLogic:
                Java: "{fst}.add({snd} != null ? {snd} : BigDecimal.ZERO)"
```

```

negateLogic:
    Java: "{0}.negate()"

metadata:
    HttpServicesAuthorizationEnabled: false
    SpringSecurityEnabled: true

clr:
    specializationNamespace: "Dddml.Wms.Specialization"

java:
    boundedContextPackage: "org.dddml.wms"
    specializationPackage:
"org.dddml.wms.specialization"

php:
    boundedContextNamespace: "Dddml\\Wms"

typeScript:
    boundedContextNamespace: "Dddml.Wms"

hibernate:
    hibernateTypes:
        Money:
            mappingType:
"org.dddml.wms.domain.hibernate.usertypes.MoneyType"
            propertyNames: ["Amount", "Currency"]
            propertyTypes: ["decimal", "string"]

nHibernate:
    nHibernateTypes:
        Money:
            mappingType:
"Dddml.Wms.Services.Domain.NHibernate.MyMoneyType,
Dddml.Wms.Services"
            propertyNames: ["Amount", "Currency"]
            propertyTypes: ["decimal", "string"]

```

---

以上面的DDDML代碼為例，對configuration結點下的各個子結點做個簡單的說明：

· `/configuration/boundedContextName`，其值是限界上下文的名稱。

· `/configuration/defaultModule`，其值是上下文中默認模塊的名稱。在實現時，DDD中的模塊可能會映射到具體語言的不同概念上，比如Java的 `package`，.NET ( CLR ) 的 `namespace` 等。筆者認為在實踐中還可以考慮把它們映射到微服務，即每個模塊是一個可以獨立部署的服務組件。

· `/configuration/defaultReservedProperties`，其值類型是 `Map<String, Object>`，它是上下文配置的一個擴展點。一般來說，可以在這裡聲明在上下文中所有實體可能都存在的那些特殊的“保留屬性”的信息。比如，可能上下文中的每個實體都需要一個屬性來表示對象是“被誰創建的” ( `Created By` )，那麼這個屬性的名稱就可以考慮在這裡設置。這裡使用的名詞 `defaultReservedProperties` ( 默認保留屬性 ) 其實也不是一個很嚴格的概念，可以把它理解只是給這個擴展點 ( `Map` ) 起的一個名字，用於和另一個擴展點 ( `/configuration/metadata` ) 區分開來。

· `/configuration/accountingQuantityTypes`，其下是用於賬務處理的數量類型的設置信息。比如一個叫作 `decimal` 的類型 ( 值對象 ) 可能會作為數量類型用於賬務處理。

· /configuration/accountingQuantityTypes/decimal  
· 其下是decimal作為賬務處理的數量類型時需要用到的一些邏輯設置。它的子結點zeroLogic、addLogic、negateLogic的值類型都是Map<String, Object>，分別表示數量的“零值”“增加”“取反”的處理邏輯。在上面的示例中展示了將decimal映射到Java語言的實現（java.math.BigDecimal）所需要的數量處理邏輯（Java代碼模板）。

· /configuration/metadata，其值類型是Map<String, Object>，它是上下文配置的一個擴展點。DDML工具可能需要用到一些在DDML規範中沒有明確定義關鍵字的設置項，它們可以寫在這裡。

· /configuration/clr，其值類型是Map<String, Object>，它是上下文在.NET（CLR）平臺的實現中可能需要使用的設置信息。

· /configuration/java，其值類型是Map<String, Object>，它是上下文在Java（JVM）平臺的實現中可能需要使用的設置信息。

· /configuration/php，其值類型是Map<String, Object>，它是上下文在PHP的實現代碼中可能需要使用的設置信息。

· /configuration/typeScript，其值類型是Map<String, Object>，它是上下文在TypeScript的實現

代碼中可能需要使用的設置信息。

· `/configuration/hibernate`，其值是當使用 Hibernate ORM 框架來實現 Repository、事件存儲等數據訪問層對象時需要使用的設置信息。

· `/configuration/hibernate/hibernateTypes`，其值是使用 Hibernate ORM 持久化值對象時需要使用的類型信息。

· `/configuration/hibernate/hibernateTypes/Money`，其值是使用 Hibernate ORM 持久化名為 Money 的值對象時需要使用的類型信息。

· `/configuration/nHibernate`，其值是使用 NHibernate ORM 框架來實現 Repository、事件存儲等數據訪問層對象時需要使用的設置信息。

## 5.3 名稱空間

DDDML要求在整個限界上下文內，所有對象（包括值對象、實體、服務、工廠、存儲庫等）都有一個獨一無二的名字。

換句話來說，我們認為在同一個限界上下文內的所有對象（類型），不管是值對象還是實體，它們共享同一個名稱空間。為了避免衝突，它們的名字必須各不相同。顯然，如果我們允許這些不同的對象有相同的命名，會給DDDML規範的設計以及DDDML工具的實現帶來額外的複雜性，而且，這也違背了團隊應該儘可能使用統一語言進行交流和建模的理念。

對於DDDML的這個要求，可以理解為：當我們面向特定的領域開發一個應用時，需要建立一張詞彙表，表上的每個詞條（術語）都有且只有一個明確的定義。

正因為同一個限界上下文內所有對象（類型）的名稱不會衝突，所以當我們需要將這些對象映射到特定語言的代碼時，可以把這些代碼生成在同一個目錄下——這裡的目錄，可以理解為Java的package、.Net的namespace等。

限界上下文內的對象，包括那些在DDDML中可能沒有直接描述但按照一定的規則會自動生成的對象，

它們的名稱都不能重複。比如，我們經常會為聚合內部的非聚合根實體生成表示它們的**Global ID**的值對象（類型）。還有，我們製作的代碼生成工具可能會生成每個實體（包括聚合根和非聚合根實體）的**Event ID**值對象。這些對象都需要佔用限界上下文的名稱空間。

每個對象的內部也有一個自己的成員名稱空間，對象的成員，包括屬性、方法、引用、約束等，它們的名稱也不能重複。同樣，那些在DDML中沒有直接描述但工具會自動產生的屬性、方法也會佔用這個名稱空間。

### 5.3.1 再談PascalCase命名風格

推薦在DDML文檔中為對象命名時使用PascalCase的命名風格：

- 對於兩個字母的首字母縮寫詞，應該兩個字母都大寫，比如“InOut”的縮寫為“IO”。
- 超過兩個字母的首字母縮寫詞，只應該將第一個字母大寫，也就是說，我們應該使用“Html”而不是“HTML”。

DDML工具有時需要將PascalCase的命名風格轉換成其他風格，比如camelCase的命名風格，或者underscored\_case的命名風格。這時就需要名稱轉換工具，以下面的代碼中的每一行作為輸入參數為例：

---

```
IOHelper
IAmAlive
XOXOHaha
XOXOHahaIAmAlive
thisIsATest
Base64Encoded
HomeHtmlPageUrl
I18nIsAbbrOfInternationalization
```

---

調用這個名稱轉換工具提供的“**ToHyphenatedString**”（意思是轉換成以“-”分隔的字符串）方法，得到的結果應該是：

---

```
io-helper
i-am-alive
xo-xo-haha
xo-xo-haha-i-am-alive
this-is-a-test
base64-encoded
home-html-page-url
i18n-is-abbr-of-internationalization
```

---

很多用於處理命名規則的工具的代碼實現都過於簡單。比如，有些工具在需要將camelCase或PascalCase的字符串轉為hyphenated-case或underscored\_case的字符串時，只是簡單地用正則表達式去匹配每個大寫字母，將它們轉換成小寫並在前面插入一個分隔符（“-”或“\_”）。也就是說，像“IOHelper”這樣的PascalCase字符串，會被轉成“i-o-helper”這樣的hyphenated-case字符串，這不是我們想要的。

### 5.3.2 注意兩個字母的首字母縮寫詞

如果按照上面推薦的PascalCase命名規則，兩個字母的首字母縮寫詞應該全部大寫，像這樣：**QQNumber**、**IOThroughput**。如果我們把這樣的名稱轉為camelCase風格，結果應該是：**qqNumber**、**ioThroughput**。你可能已經想到，如果想要把此camelCase風格的名稱再轉回PascalCase，可能會碰到麻煩。

舉個例子。如果我們使用一個JSON序列化庫，把如下JSON對象：

```
{  
    "qqNumber": "166651"  
}
```

反序列化為如下Java對象：

```
public class Example {  
    private String qqNumber;  
  
    public String getQQNumber() {  
        return qqNumber;  
    }  
  
    public void setQQNumber(String qqNumber) {  
        this.qqNumber = qqNumber;  
    }  
}
```

若JSON序列化庫在默認情況下認為`qqNumber`對應的PascalCase風格的名稱就是`QqNumber`（沒有考慮到有可能是`QQNumber`），那麼這個反序列化操作可能就會出錯。不同的類庫（工具）可能會為這樣的問題準備不同的解決方案。像Jackson JSON庫<sup>[1]</sup>，也許可以使用`@JsonProperty`註解來解決這個問題。

[1] Jackson Project Home @github,  
<https://github.com/FasterXML/jackson>。

## 5.4 關於模塊

《領域驅動設計：軟件核心複雜性應對之道》一書中雖然提到了模塊的概念，但是隻使用了很小的篇幅。筆者認為模塊不是DDD的核心概念，特別是對DDDML來說，它不是絕對必要的。一個很重要的原因前文已經說過，所有對象（類型）的名字在整個限界上下文中不應該發生衝突，因此，對生成代碼來說，將DDDML中聲明的模塊映射為具體語言的“Namespace”——比如Java的package（包）或C#的namespace——不是必需的，這就讓模塊這個概念在DDDML中的地位進一步邊緣化了。

我們支持在DDDML中以如下形式聲明一個對象所屬的模塊：

---

```
aggregates:
  PhysicalInventory:
    module: "inventory"
    id:
      name: DocumentNumber
      type: string
    properties:
      DocumentStatusId:
        type: string
```

---

在這裡，結點模塊的值可以是點分的字符串（比如wms.inventory），它是可選的。以生成Java代碼為例，在生成代碼的時候，可以考慮如下做法：假設聚

合PhysicalInventory所在的限界上下文是一個倉庫管理系統，該上下文對應的基礎包名是org.dddml.wms，那麼可以選擇（但不是必需）把這個聚合相關的代碼都生成到一個名為inventory的子包（目錄）內。也就是說，這個聚合所在的包全名是org.dddml.wms.inventory。如果有一個聚合在DDDML中沒有聲明它的模塊，那麼這個聚合相關的代碼就放置在限界上下文對應的基礎包（比如org.dddml.wms）內。

但是筆者製作的DDDML工具在生成Java代碼的時候，並沒有選擇將模塊映射為package，它甚至允許把整個限界上下文中所有對象的代碼都生成在一個package（目錄）內。

筆者的同事曾經給DDDML提出過這樣的改進建議：所有的對象都應該聲明它所在的模塊，DDDML DOM的樹結構應該體現模塊的結構。他認為DDDML應該寫成如下形式：

---

```
modules:
  Wms:
    isDefaultModule: true
    # javaPackage: org.dddml.wms
    aggregates:
      PhysicalInventory:
        id:
          name: DocumentNumber
          type: string
        properties:
          DocumentStatusId:
            type: string
```

```
# -----
# Contact 是另一個模塊
Contact:
  # javaPackage: org.dddml.wms.contact
  aggregates:
    ContactMech:
      id:
        name: ContactMech
        type: string
        # ...
# -----
# Contact 模塊下的子模塊
modules:
  Const:
    # javaPackage: org.dddml.wms.contact.const
    aggregates:
      ContactMechType:
        id:
          name: ContactMechTypeId
          type: string
        # ...
```

---

按照這個設計，模塊的概念被大大強化了。這樣做並不太合適，因為不管是在戰術層面還是在戰略層面，模塊都不是DDD的核心概念。DDDML的DOM結構非常重要，對非核心DDD概念的使用應該是可選項而不是必選項。

那麼，模塊在DDDML中還有存在的意義嗎？

筆者認為，模塊的一種使用方式是將它映射為微服務的物理邊界。如前所述，限界上下文是概念的邊界，在一個限界上下文內，大家使用同一種語言（統一語言）進行溝通。很多團隊在實踐微服務架構的時候，每個可以獨立部署的微服務確實都很“微小”，小

到甚至是一個聚合對應一個微服務，或者一個領域服務對應一個微服務。在這種情況下，限界上下文的概念邊界可能遠遠大於微服務的物理邊界。

運維一個由很多細粒度的微服務組成的應用，可能會比運維具備同樣功能的單體應用產生高得多的成本，甚至在開發的時候也是如此。比如說，如果我們在使用微服務架構開發應用時堅持這樣的實踐：

- 每個微服務都使用自己的數據庫。
- 由應用開發人員通過自行編寫代碼來實現微服務之間數據項的最終一致性。

那麼，服務拆得越細，與開發單體應用相比，實現同樣功能所需的工作量就越大。也許有時我們可以考慮讓微服務長大一點，將若干個聯繫相對緊密的聚合及領域服務歸集到一個模塊，然後對應這個模塊構建一個可以獨立部署的服務組件——還是可以稱之為微服務。當我們對這個微服務的水平擴展能力暫時沒有太高的要求時，在這個服務內部可以適當地使用數據庫事務來實現多個聚合的數據的強一致性。從降低應用開發和運維成本的角度來說，這樣做可能是一個不太壞的選擇。

另外，DDD也不是專門為微服務架構而生的，我們完全可以甚至應該使用DDD思想來指導單體應用的開發。比如可以在單體應用內部的對象模型的設計上

使用DDD的戰術關鍵概念，特別是使用聚合這個DDD戰術層面最重要的思想武器，對領域中對象的關係進行分析，得到“高內聚、低耦合”的軟件構造塊。也完全可以運用DDD戰略層面的關鍵概念，特別是限界上下文和防腐層，在將不同的單體應用有效集成的同時，保護每個單體應用的概念完整性。

## 第6章 值對象

在DDML中，值對象一般定義在：

- `/typeDefinitions`結點下，作為領域基礎類型。在一個領域內可能會廣泛使用一些適合抽象為值對象的基礎概念，把它們定義在這個結點下是很合適的。

- `/valueObjects`結點下，這裡放置的是在限界上下文範圍內被各個聚合共用的值對象的定義。

- 某個聚合或聚合內部實體的定義中，關鍵字 `valueObjects` 結點下。這裡放置的一般只是在該聚合範圍內使用的值對象的定義。比如前文介紹的 `Car` 聚合的 DDML 文檔示例，

在 `/aggregates/Car/valueObjects/TimePeriod` 結點下定義了一個 `TimePeriod` 值對象。

另外，我們把枚舉對象作為一種特殊的值對象在 `valueObjects` 結點之外單獨定義。像值對象一樣，它可以定義在 `/enumObjects` 結點下，也可以在某個聚合或聚合內部實體的定義中，關鍵字 `enumObjects` 結點下。

為什麼要把值對象的定義放在不同的地方？比如有的值對象會放在 `valueObjects` 結點下，有的會放在 `typeDefinitions` 下。

因為我們覺察到不同的值對象在代碼中的實現方式有明顯的差異。有一些值對象，可能在某個領域中是大家已經熟知的概念，我們稱之為領域基礎類型，對於這類值對像，一般會在**typeDefinitions**結點下定義。也許可以用通用的編程語言來做一個不是那麼恰當的類比，在很多通用編程語言中會有一些基本類型（**Primitive Type**），比如Java語言的基本類型包括**byte**、**int**、**long**、**float**、**boolean**、**float**等，而其他類型可以在基本類型的基礎上創建出來。

另一種值對象我們經常用到，它們只是在其他值對象的基礎上創建的數據結構，甚至都不需要有什麼方法（**Method**），我們把這樣的值對象稱為數據值對象，一般選擇在**valueObjects**結點下定義它們。一般來說，在這裡定義的值對象時，代碼生成工具應該可以直接生成特定語言的實現代碼，不需要開發人員在**DDML**文檔之外手動編碼去實現它們。

## 6.1 領域基礎類型

是否需要在DDDML規範中定義內置的基本類型（Primitive Type）？筆者曾經為此和當時團隊中的同事有過激烈的爭論。

有些同事認為，DDDML規範應該先定義一些基本類型，就像Java中的byte、int、long、float、boolean、float等一樣，然後在這些基本類型的基礎上再定義其他值對象，這樣才嚴謹。這裡說的嚴謹，大致說的是可以在DDDML中包含更多可以直接映射為實現代碼的細節。比如說，如果把JSON的類型當作DDDML基本類型，或者明確定義JSON類型與DDDML基本類型的轉換規則之後，在DDDML就比較容易聲明一個類型為值對象的屬性的默認值。

具體來說，可以像下面這樣以中立於編程實現語言的方式，給實體Person的Name屬性初始化一個“JOHN DOE”默認值：

---

```
valueObjects:
  PersonalName:
    properties:
      FirstName:
        type: string
      LastName:
        type: string

aggregates:
  Person:
```

```
id:  
  name: PersonId  
  type: string  
properties:  
  Name:  
    type: PersonalName  
    defaultValue:  
      FirstName: JOHN  
      LastName: DOE  
    #...
```

---

這是典型的技術人員的思考方式。**DDDML**規範確實支持使用**defaultValue**關鍵字來聲明屬性的默認值，不過並沒有規定它的值結點必須是什麼樣的**JSON**類型，以及在實現代碼中應該如何使用這個**JSON**表述的默認值去初始化一個屬性。

最終**DDDML**規範中沒有預設任何基本類型。如前文所言，**DDDML**的規範到目前為止還是“獨裁者”（也就是筆者）的作品，這裡主要闡述筆者的觀點。

目前**DDDML**規範對於在**valueObjects**結點下定義的值對象幾乎僅僅是規定了如何描述值對象“有什麼屬性”。如果只盯著本書中所展示的**DDDML**文檔，只注意到在**valueObjects**結點下定義的那些值對象，可能有人會誤以為**DDDML**只支持定義“數據值對象”。

不要忽略了我們在**/typeDefinitions**結點下定義的那些領域基礎類型，它們都是值對象。

有些程序員甚至都沒有意識到平時在代碼中使用的很多類型都是值對象，比如Java程序員常常用到的 BigInteger、BigDecimal等。我們已經習慣了它們的存在，習慣了把這些對象視為一個整體，而不會在做分析的時候考慮它們是由哪些更細粒度的部件組成的、怎麼持久化（在數據庫中有哪些“列”）、怎麼序列化 / 反序列化、有哪些方法等。當我們需要一個新的值對象時，很多時候只是簡單地定義一個類，一個只有屬性、沒有方法的類。當然，很多時候這確實已經足夠了，但是，並非全部的值對象都是如此。按照DDD的定義，假設只有實體和值對象這兩種對象，那麼實體之外的所有對象都是值對象，可見應該歸類為值對象的對象太多了，數不勝數。

在不同的領域，在領域專家的口中，可能本來就存在很多可以認為是一個整體的、沒有ID的、應該被建模為值對象的術語。比如，在金融領域，Money可以被建模為一個值對象；在地理信息領域，Point2D、Point3D、Line等都可能是在這個領域內大家不言自明的“值對象”；在某些領域，我們可以使用Duration這個值對象來表示“持續時間”；做一個任務調度系統的時候，可能大家會發掘出“觸發器”（Trigger）這個值對象……DDD規範不可能預先窮舉各個領域內的值對象。

作為領域驅動設計的建模語言，DDD主張並鼓勵大家使用領域中的術語進行分析和建模。當然，如果在基於領域模型進行交流的時候，大家覺得像

`byte`、`int`、`boolean`、`string`這樣的詞彙並不構成交流障礙，可以作為這個領域統一語言的一部分，那麼筆者並不反對把這些名詞定義為這個領域的基礎類型，只是不想把它們作為**DDDML**規範的一部分。

筆者完全不支持以下做法：在**DDDML**文檔中，必須基於預設的基本類型來定義值對象。除非把**DDDML**變成一門通用的編程語言，否則要想在**DDDML**中“完整”地描述值對象的各種細節是不可能的。就算支持在**DDDML**中聲明值對象的屬性、繼承的基類，並不足以描述“多姿多彩”的值對象。想要在**DDDML**文檔中描述怎麼去創建、操作值對象的種種業務邏輯，即使是專門創造一門**DDDML**的表達式語言（`EL`），恐怕也難以做到完美，而且這完全是吃力不討好的事情。不如在**DDDML**規範中採用一些更靈活的做法，比如在需要描述業務邏輯的時候使用一個`Map<String, Object>`類型的結點。

筆者認為，在**DDDML**規範中不需要預設任何基本類型。也就是說，在**DDDML**文檔中甚至不需要在使用一個值對象之前，先對一個“值對象是什麼樣子”做出任何描述——不需要聲明這個值對象是由哪些屬性組成的，也不需要聲明它是否是繼承自其他值對象的子類型。

**DDDML**應該支持在團隊一致認可某個領域概念是值對象之後，就可以在**DDDML**文件中直接使用它。這時不需要考慮是不是能“生成”值對象的實現。也就是

說在DDML文檔中使用一個沒有顯式定義的值對象是合法的。

領域模型首先需要的是抓住要點，而非技術實現的細節。

表示狀態的數據結構在領域模型中非常重要，很多時候甚至是最重要的組成部分，但是業務邏輯、系統的行為，也是模型的重要組成部分。凡是需要描述業務邏輯的地方，在DDML中，基本上都使用了一個Map<String, Object>類型的結點。業務邏輯不僅是像上面那樣“給屬性初始化默認值”的邏輯，還包括其他業務邏輯——特別是那些需要在分析階段取得共識、落實到文檔的邏輯。DDML作為建模語言，理應凸顯領域中關鍵的業務邏輯。在概念建模階段，“記錄”是第一位的，沒有必要糾結DDML文檔中記錄的業務邏輯是不是能直接生成可以執行的代碼，以及會不會出現“醜陋”的Java或其他語言的代碼或代碼模板的片段。

## 6.1.1 例子：從OFBiz借鑑過來的類型系統

前面都是理論闡述，下面來看一個示例。開源軟件Apache OFBiz<sup>[1]</sup>就是定義了一套自己專用的類型系統。

如果你下載了OFBiz的源代碼，在裡面可以找到OFBiz的類型定義。如果你對MySQL有所瞭解，那麼可以看看這個文件：  
`framework/entity/fieldtype/fieldtypemysql.xml`。該文檔的部分內容如下（有刪減）：

---

```
1  <fieldtypemodel
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  2
  xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/
  fieldtypemodel.xsd">
  3      <field-type-def type="date-time" sql-type="DATETIME"
  java-type="java.sql.Timestamp"/>
  4      <field-type-def type="date" sql-type="DATE" java-
  type="java.sql.Date"/>
  5      <field-type-def type="time" sql-type="TIME" java-
  type="java.sql.Time"/>
  6
  7      <field-type-def type="currency-amount" sql-
  type="DECIMAL(18,2)" java-type="java.math.BigDecimal"/>
  8      <field-type-def type="currency-precise" sql-
  type="DECIMAL(18,3)" java-type="java.math.BigDecimal"/>
  9      <field-type-def type="fixed-point" sql-
  type="DECIMAL(18,6)" java-type="java.math.BigDecimal"/>
  10     <field-type-def type="floating-point" sql-
  type="DOUBLE" java-type="Double"/>
  11     <field-type-def type="numeric" sql-
  type="DECIMAL(20,0)" java-type="Long"/>
```

```
12          <field-type-def type="id" sql-type="VARCHAR(20)"  
13          java-type="String"/>  
14          <field-type-def type="id-long" sql-  
15          type="VARCHAR(60)" java-type="String"/>  
16          <field-type-def type="indicator" sql-  
17          type="CHAR(1)" java-type="String"/>  
18          <field-type-def type="short-varchar" sql-  
19          type="VARCHAR(60)" java-type="String"/>  
20          <field-type-def type="long-varchar" sql-  
21          type="VARCHAR(255)" java-type="String"/>  
22          <field-type-def type="credit-card-number" sql-  
23          type="VARCHAR(255)" java-type="String"/>  
24          <field-type-def type="credit-card-date" sql-  
25          type="VARCHAR(7)" java-type="String"/>  
26          <field-type-def type="email" sql-  
27          type="VARCHAR(320)" java-type="String"/>  
28          <field-type-def type="url" sql-  
29          type="VARCHAR(2000)" java-type="String"/>  
30          <field-type-def type="tel-number" sql-  
31          type="VARCHAR(60)" java-type="String"/>  
32      </fieldtypemodel>
```

---

如果需要在DDML文檔中把上面OFBiz的這些類型都作為一個上下文的領域基礎類型來定義，應該怎麼做呢？下面就是示例：

---

```
typeDefinitions:  
    date-time:  
        sqlType: DATETIME  
        cSharpType: DateTime?  
        javaType: java.sql.Timestamp  
    date:  
        sqlType: DATE  
        cSharpType: DateTime?  
        javaType: java.sql.Date  
    time:  
        sqlType: TIME
```

```
    cSharpType: DateTime?
    javaType: java.sql.Time
currency-amount:
    sqlType: DECIMAL(18,2)
    cSharpType: decimal?
    javaType: java.math.BigDecimal
currency-precise:
    sqlType: DECIMAL(18,3)
    cSharpType: decimal?
    javaType: java.math.BigDecimal
fixed-point:
    sqlType: DECIMAL(18,6)
    cSharpType: decimal?
    javaType: java.math.BigDecimal
floating-point:
    sqlType: DOUBLE
    cSharpType: double?
    javaType: Double
numeric:
    sqlType: DECIMAL(20,0)
    cSharpType: long?
    javaType: Long
id:
    sqlType: VARCHAR(20)
    cSharpType: string
    javaType: String
id-long:
    sqlType: VARCHAR(60)
    cSharpType: string
    javaType: String
id-vlong:
    sqlType: VARCHAR(250)
    cSharpType: string
    javaType: String
indicator:
    sqlType: CHAR(1)
    cSharpType: string
    javaType: String
very-short:
    sqlType: VARCHAR(10)
    cSharpType: string
    javaType: String
short-varchar:
    sqlType: VARCHAR(60)
    cSharpType: string
```

```
    javaType: String
long-varchar:
    sqlType: VARCHAR(255)
    cSharpType: string
    javaType: String
very-long:
    sqlType: LONGTEXT
    cSharpType: string
    javaType: String
comment:
    sqlType: VARCHAR(255)
    cSharpType: string
    javaType: String
description:
    sqlType: VARCHAR(255)
    cSharpType: string
    javaType: String
name:
    sqlType: VARCHAR(100)
    cSharpType: string
    javaType: String
value:
    sqlType: VARCHAR(255)
    cSharpType: string
    javaType: String
credit-card-number:
    sqlType: VARCHAR(255)
    cSharpType: string
    javaType: String
credit-card-date:
    sqlType: VARCHAR(7)
    cSharpType: string
    javaType: String
email:
    sqlType: VARCHAR(320)
    cSharpType: string
    javaType: String
url:
    sqlType: VARCHAR(2000)
    cSharpType: string
    javaType: String
tel-number:
    sqlType: VARCHAR(60)
    cSharpType: string
    javaType: String
```

```
blob:
  sqlType: LONGBLOB
  cSharpType: byte[]
  javaType: java.sql.Blob
byte-array:
  sqlType: LONGBLOB
  cSharpType: byte[]
  javaType: byte[]
object:
  sqlType: LONGBLOB
  cSharpType: object
  javaType: Object
```

---

按照**DDDM**規範，還可以在/**typeDefinitions**的每個子結點中，通過**baseType**、**length**、**precision**之類的關鍵字，進一步對基礎類型做出更詳細的描述，但是這些都是可選的而不是必須的。像上面的例子就已經足以支持我們製作的**DDDM**工具生成可以工作的軟件代碼。



**提**示我們甚至製作了一個基於**OFBiz**實體模型的導入工具，使用它可以直接將**OFBiz**的實體定義文檔（**XML**）轉換成我們需要的**DDDM**（**YAML**）文檔，然後直接生成可以編譯、執行的代碼。

[1] 見 <https://ofbiz.apache.org/>。

## 6.1.2 例子：任務的觸發器

DDDML首先應該是用來做分析和概念建模的。在概念建模完成後，再在DDDML文檔中補充和實現相關的細節。

筆者希望在使用DDDML對一個領域進行分析與建模的時候，直接使用領域中的概念和詞彙。

下面以一個任務調度系統可能需要的Trigger值對象為例，說明在分析和概念建模階段應如何寫DDDML：

---

```
aggregates:
  # 任務聚合
  Task:
    id:
      name: TaskId
      type: id
    properties:
      # 省略其他屬性
      # 任務的觸發器 ( 觸發時機 )
    Trigger:
      type: Trigger
      # 屬性的“默認值”邏輯
      defaultLogic:
        # 每年執行一次
        # 偽代碼
        PseudoCode: "@yearly"
        # # 或者默認值是這樣的：
        # # 每週執行一次
        # PseudoCode: "@monthly"
        # # 每星期執行一次
        # PseudoCode: "@weekly"
```

```
# # 每星期二上午 9 點執行一次
# PseudoCode: "@every Tuesday 9:00 AM"
# # 只執行一次，不再重複
# PseudoCode: "@2009-12-30 8:55:55"
```

---

在這裡，結合/aggregates/Task/properties/Trigger/defaultLogic的值描述的是給Task實體的Trigger屬性提供“默認值”的邏輯。在概念建模階段，可以使用一段偽代碼來描述它。至於DDDMIL代碼生成工具能不能把這段偽代碼變成可以執行的代碼，在這個階段無關緊要。

等到了實現階段，可以修改、完善這個DDDMIL文檔，示例如下（假設我們的應用使用Java語言開發）：

---

```
aggregates:
  # 任務
  Task:
    id:
      name: TaskId
      type: id
    properties:
      # 省略其他屬性
      # 任務的觸發器（觸發時機）
    Trigger:
      type: Trigger
      # 默認值邏輯
      defaultLogic:
        # 每年執行一次
        # 偽代碼
        PseudoCode: "@yearly"
        # Java 代碼
        Java: "Cron.yearly()"
        # # 或者默認值是這樣的：
        # # .....
```

```
# # 每星期二上午 9 點執行一次
# PseudoCode: "@every Tuesday 9:00 AM"
# Java: "Cron.every( TUESDAY ).AM
( \"9:00\" ) "
# # 或者這樣.....
# # 只執行一次，此後不再重複
# PseudoCode: "@2009-12-30 8:55:55"
# Java: "Cron.once( \"2009-12-30
8:55:55\" ) "
```

---

在上面的**DDDML**文檔中，假設在編碼實現階段構建瞭如下**Java**值對象類庫：

- Trigger**是一個接口或者抽象基類，**Cron**是**Trigger**的子類。
- Cron**提供了一系列靜態工廠方法，比如上面的**every**、**once**等，用於創建**Cron**的實例。

在這樣一個**Java**值對象類庫的支持下，**DDDML**中加入的這些**Java**代碼片段不僅讀起來幾乎跟偽代碼一樣直觀，而且可以直接從**DDDML**文檔生成可執行的**Java**代碼。其實甚至在分析和概念建模階段，這個類庫還不存在的時候，我們就可以在**DDDML**中直接寫這樣的**Java**代碼片段，我們在這個過程中對將來需要實現的值對象類庫也進行了設計。

## 6.2 數據值對象

在應用開發的過程中，我們經常會用到一種只有屬性、沒有方法的值對象。這樣的值對象本身沒有什麼重要的行為，只是一個簡單的數據結構、一個數據的容器，所以我們把它叫作數據值對象。

在這裡，有讀者可能想到了貧血模型和充血模型這兩個概念，但是這兩個概念在更多的時候並不是專門針對值對象提出的，所以這裡專門為這樣的值對象起了“數據值對象”這個名字。

這樣的值對象在DDML中可以定義在/**valueObjects**結點下，示例如下：

---

```
valueObjects:
  PersonalName:
    properties:
      FirstName:
        type: string
        description: First Name
        length: 50
      LastName:
        type: string
        description: Last Name
        length: 50
```

---

在這個例子裡，我們看到**PersonalName**這個值對象的定義中只包含了屬性 ( **properties** ) 的描述。在目

前的DDML規範中，值對象的屬性類型只能是值對象。

也可以在某個聚合或聚合內部實體的**valueObjects**結點下定義值對象，如果我們認為主要是在某個聚合的範圍內使用它們。

需要說明的是，筆者在實踐中，到目前為止，在**valueObjects**結點下定義的基本都是純粹的“數據”值對象，即它們只有**properties**，沒有**methods**。即使你在DDML文檔內為這些值對象定義了**methods**。筆者製作的DDML工具也會選擇把它們忽略掉。不排除以後把定義在**valueObjects**結點下值對象的**methods**信息利用起來，並且可能在DDML規範中為這些值對象增加更多可用的描述選項（關鍵字）。

目前的情況，對於那些在實現代碼中既要有屬性又要要有方法的“複雜”的“非數據”值對象，建議在**/typeDefinitions**結點下定義它們，即使它們可能看起來不太像“領域基礎類型”。

在DDML文檔中定義數據值對象還是非常有用的。DDML工具很容易使用它們來生成實現代碼，這可以大大減少開發人員手動編碼的工作量。它們的屬性類型往往可以映射為通用編程語言內置的基本類型或開發者常見的類型。DDML工具在生成實現代碼時，對於這樣的值對象的序列化、持久化問題的處理可能非常簡單，甚至還可以直接生成它們對應的前端

用戶界面 (UI) 組件的代碼。而對於領域基礎類型，代碼生成的難度恐怕就會大一些。

## 6.3 枚舉對象

枚舉對象是一種特殊類型的值對象。我們可以這樣定義一個枚舉對象：

---

```
enumObjects:
  DocumentAction:
    baseType: string
    values:
      Draft:
        description: Draft
      Complete:
        description: Complete
      Void:
        description: Void
      Close:
        description: Close
      Reverse:
        description: Reverse
```

---

在上面的代碼中，**DocumentAction**枚舉對象描述了可以對文檔執行的動作。

或者我們可以把枚舉對象定義在一個聚合內部，示例如下：

---

```
aggregates:
  Car:
    id:
      name: Id
      type: string

    properties:
```

```
    Wheels:
        itemType: Wheel
    # -----
    # 中間省略
    # -----
    entities:
        Wheel:
            id:
                name: WheelId
                type: WheelId

    # -----
    # 中間省略
    # -----
    enumObjects:
        WheelId:
            baseType: string
            values:
                LF:
                    description: left front
                LR:
                    description: left rear
                RF:
                    description: right front
                RR:
                    description: right rear
```

---

**WheelId**枚舉對象描述了汽車四個輪子的標識符（左前、左後、右前、右後）。

這裡的**baseType**並不是必須定義的。它的值可能是“**string**”，可能是“**int**”，也可能是其他值，**DDDML**對此不作限制。

**DDDML**代碼生成工具可以視不同語言能夠提供的特性，以及開發團隊的編碼規範等因素，為**DDDML**定義的枚舉對象生成合適的代碼。

在有些語言，比如Java和C#中，存在enum關鍵字，但在有些語言中可能沒有枚舉的概念。DDDML工具在生成代碼時，可以考慮把枚舉對象（類型）替換成枚舉對象定義中聲明的baseType（基類型），有時候這也不是一個太糟糕的選擇，畢竟這可能帶來序列化、持久化處理方面的便利。

比如，對於上面例子中的WheelId枚舉對象，可以考慮簡單地生成如下Java代碼：

---

```
public class WheelId {
    private WheelId() {}

    public static final String LF = "LF";
    public static final String LR = "LR";
    public static final String RF = "RF";
    public static final String RR = "RR";
}

public interface WheelState {
    String getWheelId();
    //...
}
```

---

另外，DDDML工具可能會視需要在上下文中自動創建枚舉對象，比如，為一個實體繼承結構的“類型的Discriminator（區分標識）”自動創建枚舉對象。

## 第7章 聚合與實體

聚合是DDD在戰術層面最重要的概念。一個聚合只能包含一個聚合根，但是嚴格來說聚合與聚合根是兩個不同的概念，有些時候我們需要區分對待它們，但是有些時候把它們“混為一談”也許有助於編寫DDDML文檔。

在本章中，你會看到DDDML規範規定實體與實體之間只有一種基本關係，這種關係表現為實體的一種基本屬性。即使把這種基本屬性包括在內，實體的基本屬性總共也才三種。實體之間的其他關係，包括多對一的引用關係，都是基於這些基本屬性派生出來的。

DDDML支持描述與實體ID相關的細節信息，比如聚合內部實體Global ID的列名 ( Column Name )，這很大程度上是因為我們希望從DDDML文檔生成代碼時，代碼能像手寫的一樣自然、漂亮。

我們在現實的軟件開發過程中領悟到：DDDML應該支持將一個實體聲明為“不變的”，也應該支持將一個實體聲明為“動態的”。

既然領域模型屬於OO模型，DDDML當然少不了要描述對象的繼承與多態方面的信息。在將概念顯現

出來時，約束（Constraint）是非常有用的概念，有必要記錄它們。

DDDML提供的關鍵字肯定不能覆蓋領域模型的方方面面，建議在DDDML文檔的固定結點中描述那些擴展的模型信息。

## 7.1 用同一個結點描述聚合及聚合根

在大多數情況下，聚合的名稱與聚合根的名稱是一樣的，而且，大多數時候在一個聚合中只存在一個實體（也就是聚合根），這是當前的DDML規範決定使用同一個DOM結點來描述聚合及聚合根的重要原因之一。

比如，對於Car這個聚合的描述如下：

---

```
aggregates:
  Car:
    # 這裡可以但不是必須聲明聚合根的名稱
    # aggregateRootName: Car
    id:
      name: Id
      type: string
    properties:
      Wheels:
        itemType: Wheel
      Tires:
        itemType: Tire
    # -----
    # 聚合內部的實體
    entities:
      Wheel:
        id:
          name: WheelId
          type: WheelId
    # ...
```

---

我們沒有選擇像下面這樣使用不同的結點來定義聚合以及聚合根（注意aggregates/Car/root結點）：

---

```
aggregates:
  # -----
  # 聚合的定義
Car:
  # -----
  # 聚合根的定義
  root:
    name: Car
    id:
      name: Id
      type: string
    properties:
      Wheels:
        itemType: Wheel
      Tires:
        itemType: Tire
  # -----
  # 這是聚合根的元數據
  metadata:
    AggregateRootFoo: Root-Foo-Value1
    AggregateRootBar: Root-Bar-Value2
  # -----
  # 聚合內部(非聚合根)實體
  entities:
    Wheel:
      id:
        name: WheelId
        type: WheelId
      # ...
  # -----
  # 這是聚合的元數據
  metadata:
    AggregateMetaKey1: Aggregate-Meta-Value1
    AggregateMetaKey2: Aggregate-Meta-Value2
```

---

沒有采用上面這種寫法的原因之一，是我們不想讓開發人員在寫**DDDML**的時候需要不時地停下來分辨這個東西到底是屬於聚合還是聚合根。雖然從概念角度出發，可能有些信息放置在聚合根結點內更貼切，

有些信息則放置在聚合根結點外更合理，但是截至目前來看，在這個問題上“佛系”一點好像對實現“工作的軟件”並沒有什麼負面的影響。

下面再來談談聚合與實體的名稱問題。

因為聚合與聚合根是不同的東西，所以它們的名稱可能不同。有時候，聚合的名稱與聚合根的名稱、聚合內部的實體（非聚合根實體）的名稱都不一樣。這種情況前文已經給過一個示例：**Order**是聚合，**OrderHeader**是訂單聚合的聚合根，**OrderItem**是訂單聚合內部的實體。

有時候，聚合的名稱與該聚合內部的非聚合根實體的名稱相同。為什麼會這樣？也許這是在分析和領域建模過程中自然形成的，我們不應該禁止大家選擇這樣的命名。

來看個示例，假設一開始大家同意構造一個產品價格（**ProductPrice**）實體，該實體對應的數據表的列名如下（標註為[PK]的列是主鍵）。

·**productId** [PK]：產品ID。

·**productPriceTypeId** [PK]：價格類型ID。

·**currencyUomId** [PK]：貨幣單位ID。

·**fromDate** [PK]：從何時開始生效。

- thruDate : 截至何時有效。

- price : 價格。

開始的時候大家認為這個**ProductPrice**實體是**ProductPrice**聚合的聚合根。也就是說，**ProductPrice**聚合內只有這一個實體。但是隨著項目的推進，大家可能發現，在代碼的實現上直接使用**ProductPrice**這個實體不太好用，很多時候我們只需要知道當前價格就可以了。於是增加另一個實體，把它命名為**ProductPriceMaster**，該實體對應的數據表的列名如下（它的主鍵不包括fromDate）。

- productId [PK] : 產品ID。

- productPriceTypeId [PK] : 價格類型ID。

- currencyUomId [PK] : 貨幣單位ID。

- activeFromDate : 當前有效的（active）產品價格的開始生效時間。

自然，我們可能會把這個新增的實體**ProductPriceMaster**作為**ProductPrice**聚合的聚合根，因為我們希望能保證聚合根**ProductPriceMaster**與實體**ProductPrice**的狀態一致。

我們可能會給一個名為**ProductPriceApplicationService**的應用服務定義一個立

即更新產品價格的方法（命令），這個方法的參數可能包括：

- `productId`：產品ID。
- `productPriceTypeId`：價格類型ID。
- `currencyUomId`：貨幣單位ID。
- `newPrice`：新價格，即時生效。

該應用服務方法的大致實現邏輯如下：

- 使用`productId`、`productPriceTypeId`、`currencyUomId`參數的值從Repository中獲得`ProductPriceMaster`聚合根的實例（狀態）。
- 使用`ProductPriceMaster`的實例，以`activeFromDate`屬性的值為Local ID，獲得當前生效的那個`ProductPrice`的實例，更新這個`ProductPrice`實例的`thruDate`為當前時間。
- 在`ProductPriceMaster`實例內添加新的`ProductPrice`實例，它的`price`等於方法參數`newPrice`的值，它的`fromDate`為當前時間。
- 將`ProductPriceMaster`實例的`activeFromDate`屬性修改為新添加的`ProductPrice`的`fromDate`，使用

**Repository**保存整個聚合實例（即**ProductPriceMaster**以及它直接關聯的**ProductPrice**的實例）的狀態。

使用**DDDM**工具生成像**ProductPrice**這樣的聚合代碼時，需要特別留意聚合、聚合根、非聚合根實體之間名稱的異同，因為處理不慎很容易導致代碼中對象的命名發生衝突。後文會進一步介紹為了解決這個問題我們在實踐中所採用的一些做法。

## 7.2 實體之間只有一種基本關係

按照當前的**DDML**規範來看，實體與實體之間只存在一種基本關係。這種關係從外層實體指向和它直接關聯的內層實體。也就是說，從聚合根指向它直接關聯的聚合內部實體，或者從聚合內部的非聚合根實體指向它直接關聯的與其在同一個聚合內的另一個非聚合根實體。

**DDML**認為實體之間只有這一種基本關係，實體之間其他類型的關係都是在這種基本關係——或實體與值對象之間的其他基本關係——的基礎上派生出來的。

舉例說明：

---

```
aggregates:
  Car:
    id:
      name: Id
      type: string
    properties:
      Wheels:
        itemType: Wheel
      Tires:
        itemType: Tire

  entities:
    # -----
    Wheel:
      id:
        name: WheelId
```

```

        type: WheelId
    # -----
    # 如果沒有特別聲明，  

    # DDDML 工具可能也會為 Wheel 生成 Global ID 值
    對象，  

    # 值對象名稱是 CarWheelId
    # -----
    # globalId:
    #     name: CarWheelId
    #     type: CarWheelId
    # outerId:
    #     name: CarId

    # -----
    Tire:
        id:
            name: TireId
            type: string
            arbitrary: true
        properties:
            Positions:
                itemType: Position

        entities:
            # -----
            -----
            # Position 是 Tire 直接關聯的實體
            # 它描述“輪胎”什麼時候在哪個“輪子”上，行駛了多少里程
            Position:
                id:
                    name: PositionId
                    type: long
                    arbitrary: true
                properties:
                    TimePeriod:
                        type: TimePeriod
                    MileAge:
                        type: long
                WheelId:
                    type: WheelId
                    referenceType: Wheel
                    referenceName: Wheel

            # -----

```

```
valueObjects:
  TimePeriod:
    properties:
      From:
        type: DateTime
      To:
        type: DateTime

# -----
enumObjects:
  WheelId:
    baseType: string
    values:
      LF:
        description: left front
      LR:
        description: left rear
      RF:
        description: right front
      RR:
        description: right rear
```

---

在上面這個例子中，描述了不同實體之間的三個基本關係：

- 從聚合根**Car**到實體**Wheel**的關係。因為我們使用的是**OO**模型，所以這個關係需要體現為**Car**的一個屬性，該屬性名為**Wheels**，也可以說這個關係的名稱叫**Wheels**。這個屬性是一個**Set**語義的集合，集合的元素類型 (**itemType**) 為**Wheel**。換句話來說，屬性**Wheels**的類型是“**Wheel**的集合”。

- 從聚合根**Car**到實體**Tire**的關係。這個關係體現為**Car**的屬性**Tires**，該屬性的類型是**Tire**的集合。

· 從實體Tire到實體Position的關係。這個關係體現為實體Tire的屬性Positions，該屬性的類型是Position的集合。

也許有讀者看到這裡會心存疑慮：在DDDML中，實體和實體之間只有這一種One to Many的基本關係夠用麼？像在Hibernate ORM框架中，實體的一對多關係還支持幾種不同語義的集合（Set、Bag、List、Map）呢！

實踐證明是夠用的。DDDML鼓勵優先使用值對象而不是引用對象（實體）。

其實，上面的例子中還描述了一個實體Position與實體Wheel之間多對一的關係，只是這個關係是一個派生關係。注意如下結

點：`/aggregates/Car/entities/Tire/entities/Position/properties/WheelId/referenceType`，它的值是Wheel，它的意思是實體Position的屬性WheelId的類型是值對象（枚舉對象）WheelId，通過這個屬性的值，我們可以引用實體Wheel的一個實例——這個從實體Position到實體Wheel的派生關係（引用）的名稱是Wheel（referenceName:Wheel）。後文會進一步討論這裡出現的引用。

## 7.3 關於實體的ID

如前文所言，當我們說起聚合內部（非聚合根）實體的ID時，通常指的是這個實體的Local ID。

在應用的實現代碼中，我們可能有必要為這樣的非聚合根實體創建一個Global ID值對象（類型）。比如說，ORM框架可能需要把這個Global ID映射到數據表的組合主鍵上。

我們希望這些代碼由DDDM的工具自動生成，並且希望生成的代碼可控、漂亮，所以我們支持在DDDM中給這個Global ID值對象指定名稱。這個Global ID值對象包含多個屬性，它們分別對應這個實體的Local ID以及它所有外層實體（一直到聚合根）的ID。

我們還希望映射到數據表中的列名如我們所願，所以我們在DDDM中支持給非聚合根實體的Global ID指定列名（Column Names）。

另外，可以認為一個聚合內部的實體在概念上存在一個或者多個派生的Outer ID屬性，它們指向其外部實體的ID，即通過它們可以獲取到外部實體的ID值。雖然可能不常用，但是我們仍然希望可以指定這些派生的Outer ID屬性的名稱，這些派生屬性也會佔用實體的成員名稱空間。

繼續以前文的Car聚合為例，在DDML中可以按如下形式描述實體ID方面的信息（注意包含“Id”的那些關鍵字）：

---

```
aggregates:
  Car:
    id:
      name: Id
      type: string
    properties:
      Wheels:
        itemType: Wheel
      Tires:
        itemType: Tire

    entities:
      # -----
      Wheel:
        id:
          name: WheelId
          type: WheelId
          # 其實即使沒有聲明，DDML 工具仍可能為 Wheel 自動
          # 生成
          # 一個名為 CarWheelId 的 Global ID 值對象
        globalId:
          name: CarWheelId
          type: CarWheelId
          columnNames:
            - CAR_ID
            - WHEEL_ID
          # 使用一個外部實體的時候，可使用關鍵字 outerId：
        outerId:
          name: CarId
          # 或者不使用 outerId，像這樣使用關鍵字 outerIds：
          # outerIds:
          #   CarId:
          #     referenceType: Car

      # -----
      Tire:
        id:
```

```

        name: TireId
        type: string
        arbitrary: true
    globalId:
        name: CarTireId
        type: CarTireId
        columnNames:
        - CAR_ID
        - TIRE_ID
    outerIds:
        CarId:
            referenceType: Car
    properties:
        Positions:
            itemType: Position

    entities:
        # -----
        -----
        # Position 是 Tire 直接關聯的實體
        # 它描述“輪胎”什麼時候在哪個“輪子”上，行駛了多少里程
        Position:
            id:
                name: PositionId
                type: long
                arbitrary: true
            globalId:
                type: CartirePositionId
                name: CartirePositionId
                columnNames:
                - CAR_ID
                - TIRE_ID
                - POSITION_ID
            outerIds:
                CarId:
                    referenceType: Car
                TireId:
                    referenceType: Tire

            properties:
                TimePeriod:
                    type: TimePeriod
                # ...

```

---

上面的例子為非聚合根實體Wheel、Tire、Position逐一指定了它們的Global ID的屬性名、屬性類型（也就是生成的Global ID值對象的名稱）、在數據庫中對應的列名（columnNames）等。其中Global ID的columnNames的值是String的列表，越外層的實體其ID對應的列名排序越靠前，所以聚合根的ID對應的列名排在第一位。

把這個例子改得更復雜一點。假設聚合根Car的ID以及聚合內部實體Position的ID都由兩部分組成——這兩個ID的類型是名為Pair的數據值對象，Pair的每個屬性都需要映射到數據表中的一個列上，那麼Position的Global ID的columnNames可以採用如下形式指定：

---

```
aggregates:
  Car:
    id:
      name: Id
      type: Pair
      # ...
    entities:
      # -----
      Tire:
        id:
          name: TireId
          type: string
          # ...
        entities:
          # -----
          Position:
            id:
              name: PositionId
              type: Pair
              globalId:
```

```
type: CartTirePositionId
name: CartTirePositionId
columnNames:
- CAR_ID_ITEM_1
- CAR_ID_ITEM_2
- TIRE_ID
- POSITION_ID_ITEM_1
- POSITION_ID_ITEM_2

# -----
valueObjects:
  Pair:
    properties:
      Item1:
        type: string
        # sequenceNumber: 1
      Item2:
        type: string
        # sequenceNumber: 2
```

---

從上面的代碼可以看到，在指定值對象Pair的屬性在數據表中對應的列名時，需要按照這些屬性在DDDML文檔中出現的順序來指定。



**提**示嚴格來說，JSON的Object是一個無序的名/值對的集合。如果JSON或YAML序列化庫在讀入DDDML文檔後，不能保持名/值對在文檔中的順序，那麼可能需要在DDDML文檔中使用關鍵字sequenceNumber顯式地指定屬性的“序號”。

可能讀者仍有疑問：如果某個非聚合根實體，它的外部實體的ID是一個包含多個屬性的數據值對象，並且這個值對象的某個屬性的類型又是包含多個屬性的數據值對象，簡單說就是“數據值對象嵌套數據值對

象”，那麼在DDDM中，這個實體的Global ID的Column Names (列名) 應該按什麼順序排列呢？

答案是以深度優先的原則排列。把上面的例子進一步複雜化，看看Car聚合的內部實體Position的Global ID的Column Names可以如何指定：

---

```
aggregates:
  Car:
    id:
      name: Id
      type: CarId
      # 注意聚合根的 ID 類型是 CarId
      # ...

    entities:
      # -----
      Tire:
        id:
          name: TireId
          type: string
          # ...

        entities:
          # -----
          Position:
            id:
              name: PositionId
              type: Pair
              globalId:
                type: CarTirePositionId
                name: CarTirePositionId
                columnNames:
                  - CAR_ID_PART_1
                  - CAR_ID_PART_2_ITEM_1
                  - CAR_ID_PART_2_ITEM_2
                  - TIRE_ID
                  - POSITION_ID_ITEM_1
                  - POSITION_ID_ITEM_2
```

## 7.4 不變的實體

關於實體與值對象如何區分，有人提出一個觀點：不變的（**Immutable**）對象就是值對象，可變的對象就是實體。

但是在實踐中我們還是經常會碰到很多不可變的實體，所以很多ORM框架都支持在映射設置中將某個實體聲明為不可變的（比如在Hibernate的HBM文件中可以聲明某個實體`mutable="false"`）。

舉個例子吧。筆者曾經給一些工廠開發過WMS（倉庫管理系統）應用。倉庫中的貨品往往需要打包，貨品的打包方式經常是大箱子套小箱子，最外層的大箱子放在托盤上以便於叉車作業。不管是大箱子還是小箱子，系統都需要給每個箱子分配一個“箱號”，箱子外粘貼的標籤上打印著箱號的條碼。可以認為箱號就是“箱子”（**Box**）這個實體的ID。

很多時候，無論是大箱子還是小箱子，一個箱子封好之後，沒有特殊情況是不會打開的。既然箱子不打開，那麼箱子裡的內容（裡面裝的產品、數量）就不會變化，所以，可以認為箱子是一個不變的實體。在DDML中，可以聲明Box實體是immutable的：

---

```
aggregates:  
  Box:
```

```
immutable: true
id:
  name: BoxNumber
  type: id

properties:
  Note:
    type: description
  # ...
```

---

使用不變的實體好處之一是緩存的處理非常簡單，可以不用考慮怎麼適時地讓緩存失效。在這個例子中，因為每個箱子的箱號（ID）關聯的貨物信息都是不變的，所以使用箱號作為Key、貨物的詳細信息作為Value，則可以把這些Mapping記錄緩存到天長地久。一個大箱子中可能打包了很多東西，比如說可能包含數千件產品，每件產品都有自己的序列號（Serial Number），能從緩存中獲取箱子內貨物的詳細信息對提高系統的響應速度非常有效。當用戶在倉庫內使用App掃描箱號條碼時，肯定不願意等待半天才能看到箱子裡面裝了什麼東西。



**提  
示**可能有讀者會問：如果因為特殊的原因一定要拆箱怎麼辦？

在系統中可以把“拆箱”功能實現為一種特殊的“出庫”作業。拆箱後要求倉庫管理員必須塗掉“箱標籤”上的箱號條碼，使它不能再被掃描到。

拆箱後又要重新裝箱呢？那就產生新的箱號、新的“箱子”實例，打印新的箱標籤，然後掃描標籤上的箱號條碼執行新箱子的“入庫”操作。

## 7.5 動態對象

DDDML中所說的動態對象是指可以在運行時添加DDDML中未定義的屬性的對象。這裡的對象主要是指實體。至於值對象有沒有必要也是“動態的”，筆者在實踐中沒有碰到這樣的需求，所以DDDML中暫時不考慮支持聲明值對象是動態的（`isDynamic`）。

這裡說的“在運行時添加”，指的是想要給實體添加一個屬性時，不需要修改DDDML文檔，不需要重新生成或修改代碼，不需要重新編譯代碼、部署應用，這個屬性馬上就可以使用。我們可以把這些在運行時添加的屬性稱為動態屬性。

以下是一個聲明實體為動態對象  
( `isDynamic:true` ) 的例子：

---

```
aggregates:
  Zoo:
    isDynamic: true
    id:
      name: ZooId
      type: id

  properties:
    Description:
      type: description
    # ...
```

---

我們希望在為客戶端提供的**API**層，這些對象的表現儘可能和普通的（非動態）對象一致。那麼怎麼樣才算儘可能和普通的對象一致？**DDDML**規範當前對此沒有具體的規定。或者說能做到多大程度的“動態”取決於實現。

以前面Zoo聚合的例子為例，如果希望服務端提供的**RESTful API**能支持客戶端發送一個**PUT**請求到這個URL：

---

```
{BASE_URL}/zoos/shanghai-zoo
```

---

那麼這個請求的消息體（**JSON**）如下：

---

```
{  
  "description": "Shanghai Zoo",  
  "keeper": "Brouce Lee"  
}
```

---

注意上面的**JSON**對象中使用了**keeper**這個名稱，而在**DDDML**描述的模型中，實體**Zoo**中並不存在名稱為**keeper**的屬性。

服務端在收到這個**PUT**請求後應該創建一個**Zoo**的實例。

然後，繼續使用**RESTful API**，客戶端發送一個**GET**請求到這個URL：

---

```
{BASE_URL}/zoos/shanghai-zoo
```

---

服務端向客戶端回覆的消息體如下（JSON）：

---

```
{  
  "zooId": "shanghai-zoo",  
  "description": "Shanghai Zoo",  
  "keeper": "Brouce Lee"  
}
```

---

這樣的**RESTful API**，筆者在開發某些應用時曾經實現過——為了滿足客戶對系統的“擴展性”要求。

實現動態對象的難度與開發應用所採用的編程語言及技術棧有關。一般來說，使用靜態語言很可能比使用動態語言要難。此外，使用**Schema-Free**的**NoSQL**數據庫可能有助於降低編碼的難度。

對於靜態語言來說，給實體添加一個屬性，讓其類型為**Map**語義的集合，貌似就可以支持動態屬性了，示例如下（C#代碼）：

---

```
public class Zoo  
{  
    public string ZooId { get; set; }  
  
    public string Description { get; set; }  
  
    private IDictionary<string, object> moreProperties = new  
    Dictionary<string, object>();
```

```
public IDictionary<string, object> MoreProperties
{
    get { return moreProperties; }
    set { moreProperties = value; }
}
```

---

一般來說，如果在**RESTful API**的實現中不對所使用的**JSON**序列化組件進行定製化，把上面的**Zoo**對象直接序列化出來的結果很可能是如下形式：

---

```
{
    "zooId": "shanghai-zoo",
    "description": "Shanghai Zoo",
    "moreProperties": {
        "keeper": "Brouce Lee"
    }
}
```

---

這可能不是我們在**DDDM**中定義一個“動態對象”想要得到的結果。

## 7.6 繼承與多態

DDDML支持使用**subtypes**關鍵字定義實體的子類型，但目前僅限於聚合根。並且，暫時規定實體的子類型與父類型的ID必須使用同樣的名稱與類型。

DDDML支持使用**polymorphic**關鍵字來聲明是否生成“多態的”實現代碼。至於**polymorphic**為true時，實現代碼應該做到什麼程度的多態，DDDML規範當前並沒有明確規定。

DDDML支持使用**inheritanceMappingStrategy**關鍵字來聲明實體的繼承映射策略（Inheritance Mapping Strategy）。關鍵字**inheritanceMappingStrategy**的值類型是String（字符串），DDDML沒有把它限制為枚舉（enum）。

以下是筆者製作的工具對DDDML文檔中描述的實體的繼承與多態信息的處理方法（僅僅是筆者的實踐，並非DDDML的規範）：

- 當實體的**polymorphic**的值為false（默認值）時，DDDML工具在處理DDDML文檔時，定義在實體子類的那些成員會被提升（Hoisted）到父類中，然後子類就會被“移除”，這樣生成代碼的時候就不需要考慮繼承關係的存在了。

·當polymorphic的值為true時，才會儘可能地生成多態的實現代碼，比如利用ORM框架支持的繼承映射特性來實現實體狀態持久化方面的多態性。

下面是一個DDML文檔描述實體繼承關係的例子：

---

```
aggregates:
# -----
Party:
  discriminator: PartyTypeId
  discriminatorValue: "*"
  inheritanceMappingStrategy: tpcc
  polymorphic: true
# -----
  id:
    name: PartyId
    type: id
  properties:
    PartyTypeId:
      type: id
    ExternalId:
      type: id
    Description:
      type: very-long
    # ...

subtypes:
# -----
LegalOrganization:
  discriminatorValue: "LegalOrganization"
  properties:
    TaxIdNum:
      type: string
# -----
InformalOrganization:
  discriminatorValue: "InformalOrganization"
  abstract: true

subtypes:
```

```
# -----
Family:
  discriminatorValue: "Family"
  properties:
    Surname:
      type: string
```

---

在這個例子中，實體LegalOrganization（法人組織）和InformalOrganization（非正式組織）是Party的子類型。Family（家庭）是InformalOrganization的子類型。使用的繼承映射策略是TPCC（Table per Concrete Class）。

在實踐中，筆者製作的代碼生成工具在生成Java或C#代碼時支持三種繼承映射策略，關鍵字inheritanceMappingStrategy的值可以是：

- tpch，表示每個繼承結構一個表（Table per class hierarchy）。
- tpcc，表示每個具體類型一個表（Table per concrete class）。
- tps，表示每個子類一個表，並且使用區分標識（Table per subclass:using a discriminator）。

這些命名其實是從Hibernate ORM框架借鑑過來的。所以，對於它們的具體含義可以查閱Hibernate的文檔瞭解。

另外，我們還支持使用JPA風格的別名。以上三種繼承映射策略的JPA風格的別名分別是：

- SINGLE\_TABLE，等於tpch。
- TABLE\_PER\_CLASS，等於tpcc。
- JOINED，等於tps。

另外，PHP開發團隊可能會使用Doctrine ORM框架來開發應用。Doctrine ORM本身支持如下兩種繼承映射策略<sup>[1]</sup>。

- Single Table Inheritance：即tpch策略。
- Class Table Inheritance, using a discriminator：即tps/JOINED策略。

還需要說明的是，在DDDML中定義實體的繼承關係時，必須使用關鍵字discriminator聲明作為“類型的區分標識”的屬性名稱。即使在DDDML中沒有顯式地定義這個屬性，我們仍會保證在實體中有使用這個名稱的屬性存在。這有助於避免在根據DDDML文檔生成代碼時發生一些尷尬的名稱衝突問題。以上面的例子來說，Party實體內一定存在一個名為PartyTypeId的屬性。

[1] Inheritance Mapping, <https://www.doctrine-project.org/projects/doctrine>

orm/en/2.7/reference/inheritance-mapping.html ◦

## 7.6.1 使用關鍵字inheritedFrom

我們可以把實體的子類型寫在另外一個DDDML文檔中，方法是使用inheritedFrom關鍵字。

比如，前面的例子可以拆成兩個DDDML文檔。第一個DDDML文檔如下：

---

```
aggregates:

    Party:
        discriminator: PartyTypeId
        discriminatorValue: "*"
        inheritanceMappingStrategy: tpcc
        polymorphic: true
        # -----
        id:
            name: PartyId
            type: id
        properties:
            PartyTypeId:
                type: id
            ExternalId:
                type: id
            Description:
                type: very-long
            # ...

    subtypes:
        # -----
        InformalOrganization:
            discriminatorValue: "InformalOrganization"
            abstract: true

            subtypes:
                # -----
                Family:
```

```
discriminatorValue: "Family"
properties:
  Surname:
    type: string
```

---

## 第二個DDDM文檔如下：

---

```
aggregates:

  LegalOrganization:
    inheritedFrom: Party
    discriminatorValue: "LegalOrganization"
    # -----
    # 這裡定義的 id 會被忽略
    id:
      name: PartyId
      type: id
    # -----
    properties:
      TaxIdNum:
        type: string
```

---

這種做法和“把它們寫在同一個DDDM文檔中”的效果相同。

另外，`inheritedFrom`還可以用來表示一個方法繼承自另一個方法。它的意思是該方法的參數列表包括了所繼承的方法的所有參數，這個用法後文再討論。

## 7.6.2 超對象

在DDDDML中可以定義Super Objects（超對象）。因為我們發現，有時候不同的實體要實現同樣的抽象或者“接口”，比如有同樣的屬性、方法，我們把這樣的接口稱為Super Objects。這和使用subtypes關鍵字來定義實體之間的繼承關係不同，比如，實現了同一個接口的實體的ID並不需要有相同的名稱和類型。

對於不同的語言來說，超對象在實現代碼中可能會被映射為特定語言的“超類”，可能是接口（比如Java的interface）、抽象基類（比如Java的abstract class）、Mixin（比如Ruby的module、PHP的trait）等，對此DDDDML不作限制。

來看個例子，筆者在給某個企業開發WMS應用的時候，構建的模型中存在如下兩個實體。

·**Package**（包裹）：貨品在倉庫中總是以包裹的形式存在。

·**PackagePart**（包裹部件）：一個包裹可能由多個子包裹（一般是箱子）組成，但也有可能沒有子包裹。我們把“包裹中的內容”抽象為包裹部件（Package Parts）這一概念，這個內容可能是子包裹，也可能是其他描述貨品信息的抽象對象。

PackagePart沒有建模為Package的子類型的現實原因不打算在這裡討論。但是，Package和PackagePart確實應該“共享”一些相同的屬性，於是我們定義了一個超對象Article（物品），把它們共有的那些屬性放置其中。相關的DDML文檔片段如下：

---

```
aggregates:
# -----
Package:
    immutable: true
    implements: [Article]

id:
    name: PackageId
    type: long

properties:
    PackageType:
        type: PackageType
    PackageParts:
        itemType: PackagePart

entities:
# -----
PackagePart:
    implements: [Article]
    id:
        name: PartId
        type: long
    globalId:
        name: PackagePartId

    properties:
        PackagePartType:
            type: PackagePartType

superObjects:
# -----
Article:
    # stereotype: interface
```

```
properties:
  SerialNumber:
    type: string
  MaterialNumber:
    type: string
  CustomerNumber:
    type: string
  WorkOrderNumber:
    type: string
  LotNumber:
    type: string
  Rank:
    type: string
  Version:
    type: string
  Quantity:
    type: int
    notNull: true
  IsMixed:
    type: bool
```

---

從上面的代碼可以看  
出，/aggregates/Package/implements的值類型是  
String的列表，也就是說，一個實體可以“實現”多個超  
對象（接口）。

我們還可以聲明一個超對象的stereotype（雖然在  
上面的DDDDML代碼中這一行是註釋掉的），來幫助  
DDDDML工具在生成特定語言的代碼時選擇超對象的類  
型（比如是“接口”還是Mixin）。



提

示這裡選擇了notNull而不是nullable作為關鍵字，原因是我們希望總是可以假設若在DDML中沒有顯式地聲明一個關鍵字的值，且它的值是布爾類型的，那麼它就是false，並且false是在多數情況下“合理”的那個默認值。

## 7.7 引用

DDDML所說的引用 ( Reference ) , 是指實體 ( 這裡的實體指的是實體的實例 ) 可以通過它自己的一個或多個值對象類型的屬性“指向”另一個實體 ( 可能是和它自己同一個類型的實體 , 甚至是它自己 ) 。這裡的引用類似關係數據庫的“外鍵”概念 , 也就是類似於ER模型中Many to One類型的Relation的概念。

## 7.7.1 定義實體的引用

要定義一個實體的引用，顯然就要描述它的“一個或多個值對象類型的屬性”和另一個實體的**ID**之間的映射信息。現在**DDML**支持兩種定義引用的方法：

- 在實體的（類型為值對象的）屬性結點中使用關鍵字**referenceType**指出這個屬性意圖引用的實體名稱（類型）。
- 在實體結點中，使用關鍵字**references**定義一個或多個引用。

以下是使用第一種方法的例子：

---

```
aggregates:
# -----
ProductType:
  id:
    name: ProductTypeId
    type: id
  properties:
    Description:
      type: description
    # ...

# -----
Product:
  id:
    name: ProductId
    type: id-long
  properties:
    ProductTypeId:
```

```
        type: id
        referenceType: ProductType
    # ...
```

---

以下是使用第二種方法的例子：

---

```
aggregates:
# -----
Product:
    id:
        name: ProductId
        type: id-long
    properties:
        ProductTypeId:
            type: id
            # referenceType: ProductType
        # ...
    references:
        ProductTypeReference:
            type: ProductType
            properties:
            - ProductTypeId
            foreignKeyName: PRODUCT_PRODUCT_TYPE
```

---

在上面兩段DDDML代碼中，第一段代碼中的結點 aggregates/Prooudct/properties/ProductTypeId/refer enceType的值是ProductType，這說明產品實體的 ProductTypeId屬性引用了（指向）實體 ProductType。

第二段DDDML代碼中，在結點 aggregates/Product/references/ProductTypeReferenc e下定義了一個在產品實體中名為 ProductTypeReference的引用，這個引用的類型

( type ) 值是 `ProductType`，它的屬性列表（即結點 `aggregates/Product/references/ProductTypeReference/properties` 的值）只包含了一個屬性的名稱：`ProductTypeId`。這同樣說明產品實體的 `ProductTypeId` 屬性引用了（指向）實體 `ProductType`。

引用的 `foreignKeyName`（外鍵名稱）的聲明是可選的。如果引用所屬實體與引用所指的實體存儲在同一個數據庫內，比如我們構建的是一個單體應用，那麼可以考慮在數據庫中生成外鍵時使用 `DDML` 中聲明的 `foreignKeyName`。

對於上面兩種定義引用的方法，如果第一種可以解決問題，那麼優先推薦第一種。但是有時候第一種方法無法替代第二種。來看一個比較複雜的例子：

---

```
aggregates:
# -----
Party:
  id:
    name: PartyId
    type: id
  properties:
    PartyTypeId:
      type: id
    # ...
# -----
RoleType:
  id:
    name: RoleTypeId
    type: id
  # ...
# -----
```

```

PartyRole:
  id:
    name: PartyRoleId
    type: PartyRoleId
    properties: {}
  valueObjects:
    PartyRoleId:
      properties:
        PartyId:
          type: id
        RoleTypeId:
          type: id

# -----
PartyRelationship:
  id:
    name: PartyRelationshipId
    type: PartyRelationshipId
    columnNames:
      - PARTY_ID_FROM
      - PARTY_ID_TO
      - ROLE_TYPE_ID_FROM
      - ROLE_TYPE_ID_TO
      - FROM_DATE
  properties:
    ThruDate:
      name: ThruDate
      type: date-time
    # ...

  valueObjects:
    PartyRelationshipId:
      properties:
        PartyIdFrom:
          type: id
        PartyIdTo:
          type: id
        RoleTypeIdFrom:
          type: id
        RoleTypeIdTo:
          type: id
        FromDate:
          name: FromDate
          type: date-time

```

```

references:
  PartyFromRef:
    description: From
    type: Party
    properties:
      - PartyRelationshipId.PartyIdFrom
  PartyToRef:
    description: To
    type: Party
    properties:
      - PartyRelationshipId.PartyIdTo
  RoleTypeFromRef:
    description: From
    type: RoleType
    properties:
      - PartyRelationshipId.RoleTypeIdFrom
  RoleTypeToRef:
    description: To
    type: RoleType
    properties:
      - PartyRelationshipId.RoleTypeIdTo
  PartyRoleFromRef:
    description: From
    type: PartyRole
    properties:
      - PartyRelationshipId.PartyIdFrom
      - PartyRelationshipId.RoleTypeIdFrom
      foreignKeyName: PARTY_REL_FPROLE
  PartyRoleToRef:
    description: To
    type: PartyRole
    properties:
      - PartyRelationshipId.PartyIdTo
      - PartyRelationshipId.RoleTypeIdTo
      foreignKeyName: PARTY_REL_TPROLE

```

---

從上面的代碼可以看到：

- 實體PartyRole的ID類型是數據值對象PartyRoleId，這個值對象有兩個屬性，PartyId與

RoleTypeId。

· 實體PartyRelationship的ID（該ID名為PartyRelationshipId）類型也是一個數據值對象PartyRelationshipId。

· 實體PartyRelationship中存在一個名為PartyRoleFromRef的引用，這個引用使用它所屬的實體的ID中的兩個屬性，即PartyRelationshipId.PartyIdFrom與PartyRelationshipId.RoleTypeIdFrom，用以指向實體PartyRole。

 提示這裡簡單解釋一下前面DDML代碼涉及的幾個實體名稱的含義。

· Party：業務實體。參與業務流程的人或組織，比如簽訂協議的“甲方”“乙方”。

· RoleType：角色類型。比如“供應商”“客戶”都是角色類型。

· PartyRole：業務實體角色。業務實體與角色類型的關係，用於記錄業務實體在業務流程中可以扮演的“角色”。

· PartyRelationship：業務實體關係。這裡使用From-Thru模式來記錄業務實體之間的關係。

DDDML規範提供描述引用的能力，但怎麼使用這些引用信息取決於DDDML工具的實現。以筆者使用的DDDML代碼生成工具為例，給實體增加引用基本不會改變工具生成的後端“服務”代碼，但是可能會改變生成的前端UI層的代碼。

## 7.7.2 屬性的類型與引用類型

DDDML中只有一種非集合類型的基本屬性，也就是類型為（單個）值對象的非派生屬性。如前所述，在此屬性的基礎上可以使用關鍵字**referenceType**（引用類型）來聲明這個屬性的意圖，並指向某個聚合根或聚合內部的非聚合根實體。需要注意的是，一個聲明瞭**referenceType**的屬性本質上仍然是個“普通的”值對象類型的屬性，只不過這個屬性的值應該是某個實體實例的**ID**。

在DDDML中，可將一個聚合內的實體（從聚合根到其他實體）組合成一個從外到內的多層級結構，我們把相鄰的兩個層級的實體叫作**Outer Entity/Inner Entity**。顯然，在層級結構的不同位置引用同一個實體可能會使用不同類型的值對象，在有的位置可能需要使用**Global ID**，在有的位置可能使用**Local ID**就夠了。

我們規定，一個屬性的**referenceType**和**type**（類型）的值只能是以下組合：

- referenceType**是某個聚合根的名稱，**type**是**referenceType**所指的聚合根**ID**的那個**type**。此時屬性的**type**允許省略聲明，因為可以根據**referenceType**來推斷出它的**type**就是**referenceType**指向的聚合根**ID**的**type**。

·referenceType是某個聚合內的實體的名稱，type是referenceType所指的實體的Global ID的值對象名稱。

·referenceType是使用屬性所在的實體的Global ID的值，加上一個被引用的實體的Local ID的值就可以引用到同一個聚合內的實體的名稱，type是被引用的實體的Local ID的type。此時屬性的type允許省略聲明。這裡所說的“可以引用到的同一個聚合內的實體”包括屬性所在的當前實體的兄弟（Sibling）實體，當前實體的外層實體（Outer Entities）的兄弟實體。

除了以上組合外，屬性的referenctType和type的其他組合都是非法的，DDML工具應該拒絕處理非法的組合並報告異常。

## 7.8 基本屬性與派生屬性

在DDDM規範，實體只有以下三種基本屬性：

- 類型為值對象。
- 類型是值對象的集合。這個集合是**Set**語義的。
- 類型是直接關聯的同一聚合內部（非聚合根）實體的集合。這個集合也是**Set**語義的。它體現的是實體之間的（唯一的一種）基本關係，這在前文已經做過闡述。

所謂的“基本”，是指這個屬性沒有被顯式地聲明是“派生的”（`isDerived:true`），也沒有聲明它的某種派生邏輯（即它也沒有被“隱式地”派生）。

我們來看個例子：

---

```
aggregates:
  # -----
  Person:
    id:
      name: PersonId
      type: PersonId
    properties:
      BirthDate:
        type: DateTime
        description: 出生日期
    Titles:
      itemType: string
```

```

YearPlans:
    itemType: YearPlan

entities:
# -----
YearPlan:
    id:
        name: Year
        type: int
    properties:
        Description:
            type: string
    MonthPlans:
        itemType: MonthPlan
entities:
# -----
MonthPlan:
    id:
        name: Month
        type: int
    properties:
        Description:
            type: string

valueObjects:
# -----
PersonId:
    properties:
        # 以下省略

```

---

在上面的DDML代碼中，實體Person的以下三個屬性都是基本屬性：

- 屬性BirthDate是第一種基本屬性，它的類型是（單個）值對象。
- 屬性Titles是第二種基本屬性，它的類型是值對象的集合。具體地說，這裡的“頭銜”是一個string的

**Set**。顯然一個人的頭銜可以有多個，比如丹妮莉絲的頭銜就有很多：Daenerys Stormborn（風暴中降生的丹妮莉絲）、the Unburnt（不焚者）、Queen of Meereen（彌林女王）、Queen of the Andals and the Rhoynar and the First Men（安達爾人，羅伊那人和先民的女王）、Lord of the Seven Kingdoms（七國之主）、Protector of the Realm（疆域保護者）、Khaleesi of the Great Grass Sea（大草海的卡麗熙）、Breaker of Shackles（鏽镣粉碎者）、Mother of Dragons（龍之母）。

·屬性YearPlans是第三種基本屬性，它的類型是實體的集合。

其他類型的屬性只能是在上面這三種基本屬性的基礎上派生出來的屬性。

## 7.8.1 類型為實體集合的派生屬性

如果一個實體的派生屬性的itemType是實體的名稱，那麼它可能有幾種不同的派生方式。

### 1. 使用關鍵字filter與inverseOf

繼續使用前文提到的例子，在為某個WMS應用構建的模型中，**Package**（包裹）這個聚合中存在以下兩個實體：

·**Package**是聚合根。貨品在倉庫中總是以包裹的形式存在。

·**PackagePart**是聚合內部實體。一個包裹由多個子包裹（一般是箱子）組成，但也有可能沒有子包裹，所以我們把表示“包裹中的內容”的實體叫作包裹部件。一個包裹內的包裹部件有可能會組成樹結構。既然是樹結構，那麼就有根結點（Root Package Parts），每個結點可能有子結點（Child Package Parts）。

下面使用DDDML來描述這個聚合：

---

```
aggregates:
  Package:
    id:
      name: PackageId
```

```

        type: long
properties:
# ...
PackageParts:
    itemType: PackagePart
RootPackageParts:
    itemType: PackagePart
# isDerived: true
filter:
    CSharp: "e => e.ParentPackagePartId ==
0"

entities:
    PackagePart:
        id:
            name: PartId
            type: long
        properties:
# ...
        ParentPackagePartId:
            # type 可以被推斷為 long
            # type: long
            referenceType: PackagePart
            referenceName: ParentPackagePart
        ChildPackageParts:
            itemType: PackagePart
            # isDerived: true
            inverseOf: ParentPackagePart
# ...

```

---

上面的代碼中，在`/aggregates/Package/properties/RootPackageParts`結點之下，定義了實體`Package`的一個派生屬性。它的`itemType`是`PackagePart`。因為除了`itemType`和`filter`之外沒有更多的說明，所以它默認派生自從聚合根`Package`到實體`PackagePart`的基本關係——也就是名為`PackageParts`的那個基本屬性。

屬性定義中關鍵字filter對應的值結點的類型是Map<String, Object>。在上面的例子，filter中記錄了以C#代碼表達的過濾邏輯，這段代碼很容易理解，那些ParentPackagePartId等於0的包裹部件就是“根包裹部件”( Root PackageParts )，也就是包裹樹結構的根結點。

上面的代碼還演示了使用關鍵字inverseOf從實體的引用關係派生出（類型為實體集合的）屬性。

在使用關鍵字inverse Of派生之前，要定義一個類型為值對象的屬性，並且在這個屬性中存在一個引用。注意前面DDML代碼中的以下幾行：

---

```
# ...
PackagePart:
# ...
    properties:
        #
        ParentPackagePartId:
            referenceType: PackagePart
            referenceName: ParentPackagePart
#...
```

---

它的意思很明顯，實體PackagePart的屬性ParentPackagePartId是“父包裹部件”的ID，引用的類型（實體）也是PackagePart，引用的名稱是ParentPackagePart。我們知道，DDML的引用表示的是Many to One的實體關係。

然後，使用關鍵inverseOf聲明一個派生屬性是這個名為ParentPackagePart的引用的反向關係。注意代碼中的以下幾行：

---

```
#...
ChildPackageParts:
  itemType: PackagePart
  # isDerived: true
  inverseOf: ParentPackagePart
```

---

一個Many to One關係的“反向”自然是One to Many關係，所以這個派生屬性的類型是PackagePart的集合。

## 2. 使用關鍵字itemPropertyMap

上一節在講述引用時，用一段示例DDML代碼定義了Party、PartyRole、Party-Relationship等聚合，現在打算在這些模型的基礎上增加一個聚合Agreement（協議）。此協議需要記錄以下內容：

- 協議的簽訂者PartyIdFrom（甲方）、PartyIdTo（乙方）。
- 協議雙方的角色類型（RoleTypeIdFrom、RoleTypeIdTo），比如甲方的角色類型可能是客戶、乙方的角色類型可能是供應商。
- 協議開始的時間、結束的時間等。

## 我們可以把這些模型信息寫進DDML文檔：

---

```
aggregates:
  Agreement:
    id:
      name: AgreementId
      type: id
    properties:
      ProductId:
        type: id
      PartyIdFrom:
        type: id
      PartyIdTo:
        type: id
      RoleTypeIdFrom:
        type: id
      RoleTypeIdTo:
        type: id
      AgreementTypeId:
        type: id
      AgreementDate:
        type: date-time
      FromDate:
        type: date-time
      ThruDate:
        type: date-time
      Description:
        type: description
      TextData:
        type: very-long
      # ...

    PartyRelationships:
      itemType: PartyRelationship
      isDerived: true
      itemPropertyMap:
        - propertyName: RoleTypeIdFrom
          relatedPropertyName:
            PartyRelationshipId.RoleTypeIdFrom
              - propertyName: RoleTypeIdTo
                relatedPropertyName:
                  PartyRelationshipId.RoleTypeIdTo
                    - propertyName: PartyIdFrom
```

```
    relatedPropertyName:  
PartyRelationshipId.PartyIdFrom  
    - propertyName: PartyIdTo  
    relatedPropertyName:  
PartyRelationshipId.PartyIdTo
```

---

在DDML代碼的最後一部分使用關鍵字 `itemPropertyMap` 定義了一個（類型為 `PartyRelationship` 實體的集合）派生屬性 `PartyRelationships`。它表示協議雙方曾經存在過的業務實體關係（`Party Relationship`）的歷史記錄（只包含雙方當時的角色類型與當前協議的角色類型相同的那些記錄）。

有了這樣的派生屬性的描述信息，DDML代碼生成工具很容易在服務端生成這些集合屬性的查詢方法，免除程序員手動編碼的痛苦。

## 7.8.2 類型為值對象的派生屬性

類型為值對象的屬性也可能是派生的。我們可以使用關鍵字`derivationLogic`來定義屬性“讀”的派生邏輯；還可以使用關鍵字`setterDerivationLogic`來定義屬性“寫”的派生邏輯。這裡兩個關鍵字對應的值結點的類型都是`Map<String, Object>`。

以下是一個示例：

---

```
aggregates:
  Warehouse:
    # -----
    inheritedFrom: Facility
    discriminatorValue: "Warehouse"
    # -----
    id:
      name: FacilityId
      type: id-long

    properties:
      WarehouseId:
        type: id-long
        derivationLogic:
          Java: "{0}.getFacilityId()"
        setterDerivationLogic:
          Java: "{0}.setFacilityId({1})"
      WarehouseName:
        type: Name
    # ...
```

---

在這個例子中，聚合根`Warehouse`（倉庫）是`Facility`（設施）的子類型。設施的ID名稱叫作

**FacilityId**。對於倉庫而言，“倉庫ID”就是“設施ID”的別名。所以這個例子定義了**WarehouseId**這個派生屬性的派生邏輯，對**WarehouseId**屬性的讀/寫操作的實現其實只是簡單地調用**FacilityId**屬性的getter/setter方法。

## 7.9 約束

正如在《領域驅動設計精簡版》第4章中“凸現關鍵概念”一節談到的，在將概念顯現出來時，約束（Constraint）是非常有用的概念。

約束是一個簡單的表達不變性（Invariant）的方式。比如說，是一個簡單的布爾表達式，或者是一個指向布爾表達式的“指針”等。

那麼什麼是不變性？DDD所說的不變性是指在數據發生變化時必須維護的那些規則。

在DDML中，允許使用**constraints**關鍵字來定義在實體層面（Entity-Level）或屬性層面的約束。

## 7.9.1 在實體層面的約束

舉例說明，筆者開發過的某個WMS中，存在一個“庫存移動確認單”（Movement-Confirmation）聚合，實體MovementConfirmationLine是“庫存移動確認單”的“行項”，該聚合的DDDML代碼片段如下：

```
aggregates:
  MovementConfirmation:
    id:
      name: DocumentNumber
      type: string
    properties:
      # ...
    MovementConfirmationLines:
      itemType: MovementConfirmationLine
  entities:
    MovementConfirmationLine:
      id:
        name: LineNumber
        type: string
      properties:
        # ...
        # 目標數量
        TargetQuantity:
          description: The Quantity which
should have been received.
          type: decimal
          defaultValue: 0
          # 確認數量
        ConfirmedQuantity:
          description: Confirmation of a
received quantity.
          type: decimal
          defaultValue: 0
          # 差異數量
        DifferenceQuantity:
```

```

        description: If there is a
difference quantity, a Physical Inventory is created for the
source (from) warehouse.
        type: decimal
        defaultValue: 0
# 破損數量
ScrappedQuantity:
        type: decimal
        defaultValue: 0

constraints:
# ----- 數量之間的約束 -----
QuantitiesConstraint:
# 確認後，目標數量 + 差異數量 = 確認數量
validationLogic:
PseudoCode: "ConfirmedQuantity
== 0 || TargetQuantity + DifferenceQuantity ==
ConfirmedQuantity"
CSharp: "
{this}.ConfirmedQuantity == 0 || {this}.TargetQuantity +
{this}.DifferenceQuantity == {this}.ConfirmedQuantity"
ScrappedQuantityConstraint:
validationLogic:
PseudoCode: "ScrappedQuantity <=
ConfirmedQuantity"
CSharp: "{this}.ScrappedQuantity
<= {this}.ConfirmedQuantity"

```

---

在上面的DDDMIL代碼中，我們在結點/aggregates/MovementConfirmation/entities/MovementConfirmationLine/constraints下定義了聚合內部實體MovementConfirmationLine的兩個約束，它們的名稱分別是QuantitiesConstraint與ScrappedQuantityConstraint。

在約束結點中可以使用validationLogic關鍵字來定義“確認邏輯”——DDDMIL規範不排除以後支持更多的

關鍵字。validationLogic對應的值結點的類型是 `Map<String, Object>`。在這個例子裡，分別用偽代碼與C#代碼（模板）描述了約束的確認邏輯，這兩個約束的確認邏輯很容易理解：

- 如果已經確認（確認數量不等於0），那麼要  
求：目標數量+差異數量=確認數量。
- 報廢數量（ScrappedQuantity）必須小於或者等  
於確認數量。

## 7.9.2 在屬性層面的約束

使用關鍵字constraints在屬性層面定義約束時，它的值類型是String（字符串）的列表。DDDML規範沒有規定在這個列表中的字符串元素的內容必須是什麼，如何使用這些字符串取決於具體的DDDML工具的實現。

我們在實踐中是怎麼使用屬性層面的約束的呢？舉例來說明，以下是我們開發的某個WMS應用的DDDML代碼片段：

---

```
aggregates:
  Locator:
    id:
      name: LocatorId
      type: string
      constraints: [numericDashAlphabetic]

    properties:
      Description:
        type: string
        # ...
```

---

實體Locator表示存放貨物的“貨位”。我們把實體的ID也看作是一種特殊的屬性，所以可以在ID上定義屬性層面的約束。我們希望貨位ID以數字開頭，之後可以是數字、字母或“-”。於是在上面的代碼中使用一個約束：numericDashAlphabetic。這個約束其實是在

這個WMS限界上下文的Configuration中定義的一個“字符串模式”的名稱：

---

```
configuration:  
  boundedContextName: "Dddml.Wms"  
  # -----  
  namedStringPatterns:  
    numericDashAlphabetic: "^[0-9][A-Za-z0-9-]*"  
    # ...
```

---

目前DDDML規範規定結  
點/configuration/namedStringPatterns的值類型是  
Map<String, String>，Key是字符串模式的名稱，  
Value是使用正則表達式（Regex）語法的模式。

以下是我們在應用開發過程中用過的一些命名字  
字符串模式的例子。

·**alphabeticNumeric**：以字母開頭，之後可以是字  
母或數字。例如：AbcdEFG12345。模式為 $^{[A-Za-z]}[A-Za-z0-9]^*$ 。

·**numericAlphabetic**：以數字開頭，之後可以是數  
字或字母。例如：12345AbcdEFG。模式為 $^{[0-9]}[A-Za-z0-9]^*$ 。

·**alphabeticDashNumeric**：以字母開頭，之後可  
以是數字、字母或“-”。例如：AbCD-E-123A。模式為  
 $^{[A-Za-z]}[A-Za-z0-9-]^*$ 。

·**numericDashAlphabetic**：以數字開頭，之後可以是數字、字母或“-”。例如：123456-adcdE。模式為 $^{[0-9]}[A-Za-z0-9-]*$ 。

·**underscoreAlphabeticNumeric**：以下劃線或字母開頭，之後可以是字母、下劃線或數字。例如：AbcdEFG1234\_5678、A\_B\_12456cdefg。模式為 $^{[_A-Za-z]}[A-Za-z0-9_]*$ 。

·**alphabeticUnderscoreNumeric**：以字母開頭，之後可以是字母、下劃線或數字。例如：abcd\_ef12\_3456L。模式為 $^{[A-Za-z]}[A-Za-z0-9_]*$ 。

## 7.10 提供擴展點

當前的DDDM規範為實體的定義提供了兩個擴展點，這兩個擴展點的關鍵字是**reservedProperties**和**metadata**。和限界上下文的**Configuration**中的兩個擴展點（/configuration/defaultReservedProperties以及/configuration/metadata）類似，這兩個關鍵字的值類型都是**Map<String, Object>**。

提供這樣的擴展點很大程度是基於DDDM技術工具（比如代碼生成工具）的需要。這些工具可能希望在實體的定義中加入沒有在DDDM規範中作為關鍵字定義的**Key**，建議把這些**Key**放在這兩個**Map**中。提供兩個**Map**的想法是給這些擴展的元數據做一個不太嚴格的分組。一般來說，與實體的保留屬性相關的擴展元數據應該放在**reservedProperties**結點下，其他擴展元數據放在**metadata**結點下。

雖然DDDM規範沒有明確規定，但是在實現具體的DDDM工具時可以使用如下處理規則：在限界上下文的**Configuration**的擴展點中定義的元數據可以被在實體中定義的擴展點的元數據覆蓋。也就是說，一般認為後者的優先級更高。

來看一個例子。在一個實體中存在一個用於實現樂觀鎖的屬性，這是個技術細節，領域專家很有可能

並不關心它。可能在某些領域中，比如製造工藝管理，像“**Version**”這樣的名詞在領域專家那裡早就有了明確的業務含義。我們可以在限界上下文配置的`/configuration/defaultReservedProperties`結點下指定用於實現“樂觀鎖”的實體的“版本號”屬性的名稱，示例如下：

---

```
configuration:
  # ...
  boundedContextName: "Dddml.Wms"
  defaultReservedProperties:
    version: Version
    # ...
```

---

上面的**DDDML**代碼是在限界上下文內進行全局的設置，但是有時候我們可能並不想修改上下文的全局設置，因為那樣可能會影響很多地方，比如所有已生成的代碼。那麼，我們可能需要針對某個實體指定樂觀鎖使用的“那個版本號”屬性的名稱，以避免和屬於領域概念的“**Version**”發生命名衝突，示例如下：

---

```
aggregates:
  Package:
    id:
      name: PackageId
      type: long
    properties:
      RowVersion:
        type: long
        # ...
      Version:
        type: string
        # ...
```

```
reservedProperties:  
  version: RowVersion
```

---



提

示所謂的“樂觀鎖”，可以按如以下方式理解。

- 我（業務邏輯層代碼）對發生併發訪問衝突導致數據無法更新的可能性持樂觀態度，所以我對數據的“鎖定”方式是：我看到了，先隨它去吧。

- 當我想要更新數據的時候，向數據庫發出這樣一個請求：基於我看到它在某年某月某日某時某刻的樣子，我想要把它變成如此這般；如果它在某年某月某日某時某刻之後已經改變，那麼你什麼也不要作。

- 數據庫會把這個請求的執行結果反饋給我：“如你所願”，抑或“它已改變”。

- 我根據數據庫反饋的結果執行後續的操作。

如果“它已改變”，一個我可能會選擇的操作是通過“回滾”數據庫事務“放棄一切”，然後拋出異常通知上層代碼。

## 第8章 超越數據模型

領域建模不應該只對軟件模型的靜態結構建模，還需要對軟件模型的動態行為建模。

到目前為止，本書主要是介紹了怎麼使用DDDML描述聚合/實體的狀態，也就是對軟件使用的數據結構進行建模。雖然有很多業務軟件都是記錄型系統（SoR），數據結構是系統的核心，但是如果DDDML僅限於此，那實在算不上一門領域建模語言，最多算是加上了聚合概念的數據庫定義語言（Data Definition Language，DDL）而已。DDDML應該具備對系統行為進行建模的能力。

DDDML主要通過定義方法（Method）來給軟件的行為建模。DDDML支持使用關鍵字methods來定義實體的方法及領域服務的方法，它也支持對方法的安全性做出聲明。

另外，DDDML還支持使用一些關鍵字來記錄領域中關鍵的業務邏輯，這些業務邏輯用於描述或者約束系統的行為，本章將針對這些關鍵字進行集中介紹。

## 8.1 實體的方法

一個應用的業務邏輯應該被恰如其分地切分到以下三個地方。

### 1. 值對象的方法上

由於數據值對象基本沒有什麼方法，所以這裡主要指領域基礎類型的方法。值對象是實體的構造塊，但不應該僅僅是實體的數據結構的構造塊，也應該是實體的行為的構造塊。

### 2. 實體的方法上

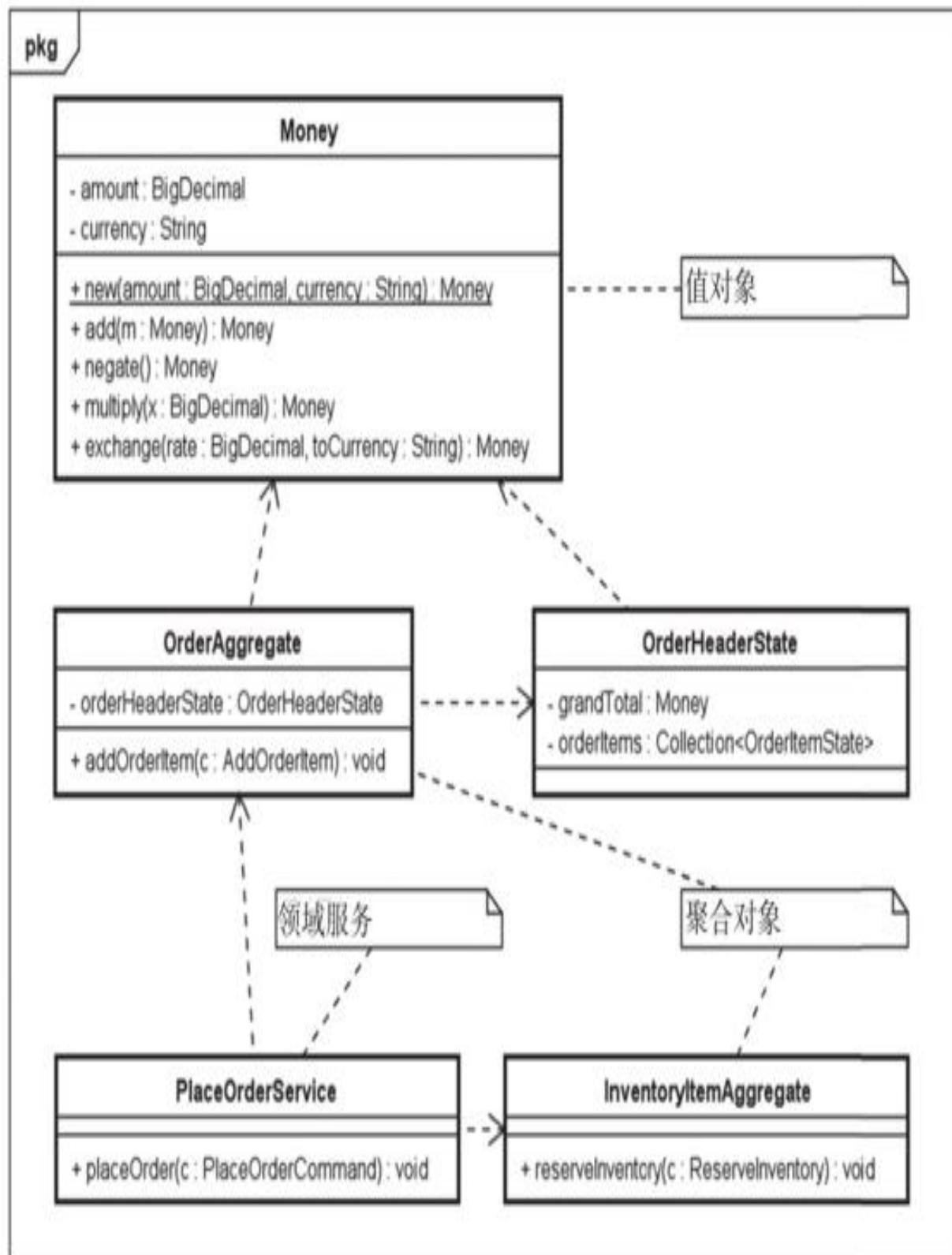
通過定義實體的方法，我們可以把修改單個聚合實例狀態的代碼提取出來，讓這些操作的語義更清晰。並且，我們可以聲明這個操作所產生的領域事件的名稱，這對實現事件驅動架構（Event-Driven Architecture，EDA）很有幫助。

雖然DDDML允許定義非聚合根實體的方法，但是其實並不太推薦這麼做，一般來說，定義聚合根的方法就足夠了。因為一個非聚合根實體的狀態應該被視為某個聚合根實例的狀態的一部分，所以，在實現非聚合根實體的方法時尤其需要注意保證聚合實例的狀態的一致性。

我們可以把實體的方法都稱為“聚合的方法”。在代碼層面，可以委託一個聚合對象去實現DDDML中定義的實體——不管是聚合根還是非聚合根實體——的（命令）方法的業務邏輯。

### 3.跨聚合的領域服務上

舉例來說：假設在我們構建的某個領域的模型中，Money是值對象；Order聚合的聚合根是OrderHeader實體，OrderItem是Order聚合的內部實體；另外可能存在一個InventoryItem（庫存單元）聚合；PlaceOrderService是“下訂單服務”，這是一個領域服務，操作多個聚合（比如Order、InventoryItem等）的業務邏輯就在其中；還有一個InventoryItem（庫存單元）聚合。在代碼層面的UML類圖則如圖8-1所示。



## 圖8-1 不同類型的對象及它們的方法

在圖8-1中，**Money**是一個值對象，包含對“錢”的處理邏輯，比如金額的相加、乘法等；**OrderAggregate**及**InventoryItemAggregate**是聚合對象，它們包含修改單個聚合實例的狀態的方法；**PlaceOrderService**是一個領域服務。下訂單時，很可能需要修改多個聚合實例的狀態，比如除了需要操作訂單聚合，還可能需要操作庫存單元（**InventoryItem**）聚合來保留庫存。

關於怎麼定義實體的方法，先看一個DDML文檔的例子：

---

```
aggregates:
  Person:
    id:
      name: PersonId
      type: PersonId
    properties:
      BirthDate:
        type: DateTime
      # ...
      Email:
        type: Email
    YearPlans:
      itemType: YearPlan
      # ...

  methods:
    # -----
    ChangeEmail:
      eventName: EmailChanged
      # notInstanceMethod: false
      parameters:
        NewEmail:
```

```

        type: Email
    # RequesterId:
    #     type: long
    #     isRequesterId: true
    # CommandId:
    #     type: string
    #     isCommandId: true
    # PersonId:
    #     type: PersonId
    #     isAggregateId: true
    # PersonVersion:
    #     type: long
    #     isAggregateVersion: true

entities:
# -----
YearPlan:
    id:
        name: Year
        type: int
    globalId:
        name: YearPlanId
        type: YearPlanId
    properties:
        Description:
            type: string
            length: 500

methods:
    ChangeDescription:
        parameters:
            Description:
                type: string
            # PersonVersion:
            #     type: long
            #     isAggregateVersion: true
            # YearPlanId:
            #     type: YearPlanId
            #     isEntityGlobalId: true

```

---

在上面的DDDMIL代碼中，定義的兩個實體的方法（`ChangeEmail`與`Change-Description`）都沒有聲明返

回的結果（沒有使用**result**關鍵字），這說明它們都是命令方法。按照CQRS的設計原則，命令方法不應該定義返回結果，有返回結果的應該是查詢方法。下面提到“實體的方法”時，如果沒有特別說明，指的都是實體的命令方法。

如果我們定義的是一個改變狀態的命令方法，那麼可以使用關鍵字**eventName**指定命令可能產生的領域事件的名稱，在這裡的**ChangeEmail**方法產生的事件名是**EmailChanged**（電子郵件已修改）。



提

示領域事件是源自領域層的事件，這些事件有業務含義，表示某些業務邏輯已經被執行。我們經常把領域事件簡稱為事件。

### 8.1.1 聚合根的方法

上一節的DDDDML代碼片段中定義了聚合根Person的一個名為ChangeEmail的方法。顧名思義，這個方法的意圖是要修改一個人（Person）的電子郵件。

這個方法是Person實體的“實例方法”，所以關鍵字notInstanceMethod（非實例方法）的值為false，由於false是默認值，所以並不需要顯式地設置它。注意，目前DDDDML不允許在聚合內部（非聚合根）實體中定義非實例方法。

這個方法顯式定義的參數只有一個，其參數名為NewEmail。客戶端想要調用實體的方法時很有可能還需要傳入其他參數。其中有些技術性參數和應用的領域概念關係不大，比如請求者的ID（RequestId）以及命令ID（CommandId）。服務端可能需要記錄請求者的ID來支持審計，並利用命令ID來實現方法的幂等性。

另外，對於實體的實例方法來說，客戶端調用它們時需要提供聚合（聚合根）的ID；如果客戶端想要調用包含了樂觀鎖檢查邏輯的實體的命令方法，那麼可能還需要傳入聚合（聚合根）的版本號（Version）。

如果在DDDML中沒有顯式地定義方法的如下參數：請求者ID、命令ID、聚合ID、聚合版本號，那麼DDDML工具可能會使用工具的默認設置或限界上下文的全局設置來自動生成它們並添加到方法的參數列表中。DDDML工具在生成代碼的時候，會在必要的地方使用這些參數的名稱。如果覺得這些自動添加的參數的名稱不如人意，那麼可以在DDDML中使用相應的關鍵字isRequesterId、isCommandId、isAggregateId、isAggregateVersion顯式地定義它們（在前面的DDDML代碼中，顯式定義這幾個參數的代碼都被註釋掉了）。



**提  
示**當我們說到“實體的方法”時，如果沒有特別說明，都是指實體的實例方法。實體的實例方法只應該修改一個實體的狀態；而實體的非實例方法可能會修改多個實體的狀態。可以把非實例方法當作領域服務的一種“變體”，只是訪問這個服務的“入口”位於實體成員的名稱空間內。

## 8.1.2 非聚合根實體的方法

我們不僅可以給聚合根定義方法，也可以給聚合內部的非聚合根實體定義方法。前文說過，面向聚合內部的非聚合根實體發出的命令（調用非聚合根實體的方法），同樣應該理解為“聚合的命令”。

如果不想定義一個非聚合根實體的方法，還可以使用以下做法達到類似的效果，即定義一個包含同樣參數的聚合根的方法，然後，在這個聚合根的方法的參數列表中添加表示“從聚合根導航到該非聚合根實體需要的Local ID(s)”的參數。

在前面的DDDML示例代碼中，定義了非聚合根實體YearPlan的ChangeDescription方法，摘錄如下：

---

```
aggregates:
  Person:
    # ...
  entities:
    # ...
  YearPlan:
    # ...
    methods:
      ChangeDescription:
        parameters:
          Description:
            type: string
            # PersonVersion:
            #   type: long
            #   isAggregateVersion: true
            # YearPlanId:
```

```
#     type: YearPlanId  
#     isEntityGlobalId: true
```

---

需要說明的是，因為聚合內部的（非聚合根）實體是不能直接訪問的，所以當客戶端代碼需要調用聚合內部實體的實例方法時，需要提供實體的**Global ID**的信息。另外，我們在聚合內採用的是“強一致性”模型，默認的實現方式是在聚合根上加一個樂觀鎖，如果客戶端要想調用聚合內部實體的方法，那麼需要提供聚合（聚合根）的版本號。如果有必要，可以在DDML中使用相應的關鍵字**isEntityGlobalId**、**isAggregateVersion**顯式地定義這兩個參數，這樣也許可以幫助工具在生成代碼時使用你想要的參數名稱。

### 8.1.3 屬性的命令

需要在DDML文檔中描述的方法大部分都是命令方法。DDML支持在聚合根中定義方法，也支持在聚合內部的非聚合根實體中定義方法，為何不考慮支持在實體的屬性中定義一個命令（方法）呢？



提

示一般來說，如果一個方法沒有返回值，基本就是命令方法無疑。因為如果要嚴格遵循CQRS的職責分離原則，那麼，命令方法應該沒有返回值；有返回值的應該是查詢方法。用戶對查詢的需求往往多變，大部分查詢方法都不需要在DDML中進行記錄。

也許有讀者會問：“有必要嗎？在實體中定義一個方法，然後在這個方法的實現中修改屬性的值不可以了嗎？”

比如，可以給一個名為InOut（入庫/出庫單）的實體定義一個名為ChangeStatus的實例方法，這個方法的業務邏輯可能只會改變InOut的DocumentStatus（單據狀態）屬性。

這種做法確實可以，大多數時候我們確實也是這麼做的。但是筆者認為支持在實體的屬性中定義命令也有它的好處，比如可以使得這個命令的意圖更明顯，它的所作所為就是為了改變這個屬性的狀態。

可能還存在這種情況：有些時候，我們並不想自己完全從零開始實現那些更新實體的命令（方法），也許我們會覺得**DDML**代碼生成工具“自動生成”的創建、更新實體的方法的實現邏輯“大部分”都是好的，我們只是想在這些方法被調用時不能直接修改某些屬性——對這些屬性的修改，我們希望使用自己定製的邏輯。

所以我們考慮支持在**DDML**的屬性結點中定義命令，以表示對這個屬性狀態的修改必須通過發送相應的命令來完成。另外，這個特性還可以和支持“狀態機模式”的特性結合使用，這在後文會進一步介紹。

基於這些考慮，於是**DDML**允許這麼寫：

---

```
aggregates:
  InOut:
    id:
      name: DocumentNumber
      type: string
  properties:
    IsSOTransaction:
      type: bool
    # -----
    DocumentStatus:
      type: string
      commandType: DocumentAction
      commandName: DocumentAction

  # -----
enumObjects:
  # 單據操作
  DocumentAction:
    baseType: string
    values:
```

```
# 起草
Draft:
    description: Draft
# 完成
Complete:
    description: Complete
# 作廢
Void:
    description: Void
# 關閉
Close:
    description: Close
# 反轉 ( 完成後可以反轉 )
Reverse:
    description: Reverse
# 確認 ( 部分單據需要確認 )
Confirm:
    description: Confirm
```

---

在這樣定義之後，DDDMIL代碼生成工具為實體 InOut 生成的創建 ( Create ) 、更新 ( Update ) 實體的命令對象中就不再存在名為 DocumentStatus 的屬性了。前文已經討論過，命令 ( Command ) 和狀態 ( State ) 是兩個不同的概念，Command 對象與 State 對象的屬性很可能是不一致的，這是又一個“不一致”的例子。



提示實際上筆者製作的DDDML工具所生成的Java代碼中，更新InOut實體的命令對象（接口）叫MergePatchInOut。另外，工具生成的Java代碼中包括名為Property-CommandHandler的接口。對於這個例子，開發人員可以編寫Property-Command-Handler的實現類，接收客戶端傳入的DocumentAction，並返回新的DocumentStatus。

### 8.1.4 命令ID與請求者ID

方法的一些特殊的技術性參數——比如命令ID與請求者ID——使用的名稱應該避免和領域中的關鍵概念衝突。所以DDDML支持對這些參數的名稱進行定製。

可以在限界上下文的Configuration中進行全局的設置，示例如下：

```
configuration:  
  commandIdName: CommandId  
  requesterIdName: RequesterId
```

如前文所述，可以在定義某個方法時覆蓋全局的設置，但是一般我們都不這樣做。

命令ID（Command ID）可以用於實現方法的幂等性。服務端在實現實體的命令方法時，可以考慮使用“聚合類型+聚合根ID+Command ID”的組合來檢測出客戶端發送過來的重複命令。

客戶端產生Command ID的方式，可以是在調用方法前生成一個UUID（GUID）作為Command ID。也可以考慮把調用方法使用的所有實參（除Command ID之外）做個Hash摘要，以此作為Command ID。



提示在創建一個聚合根的實例時，可以考慮使用聚合根的ID作為**Command ID**。

不只是實體的方法，調用領域服務的方法時，可能也需要客戶端提供**Command ID**和**Requester ID**。

領域服務可能會改變多個聚合的狀態，實現領域服務通常需要調用實體的方法。如果實體的方法保證了幂等性，那麼採用最終一致性模型實現領域服務的編碼工作可能也會簡單很多。

## 8.2 記錄業務邏輯

目前DDML規範定義了一些關鍵字用於記錄領域的關鍵業務邏輯，這些關鍵字的值類型都是 `Map<String, Object>`。表8-1說明了這些關鍵字在DDML文檔中應該出現的位置，以及它們的值應該描述什麼邏輯。

表8-1 部分用於記錄業務邏輯的關鍵字

关键字	位置	描述的逻辑的含义
<code>defaultLogic</code>	属性结点中	产生属性的默认值的表达式
<code>derivationLogic</code>	属性结点中	实现派生属性的“读”方法的表达式
<code>setterDerivationLogic</code>	属性结点中	实现派生属性“写”方法的逻辑
<code>filter</code>	类型为集合的属性结点中	从其他类型为集合的属性派生出该属性的过滤器
<code>referenceFilter</code>	声明了 <code>referenceType</code> 的属性结点中	限制可以被该属性引用的实体实例的过滤器

关键字	位置	描述的逻辑的含义
validationLogic	实体的约束结点中 (constraints/{CONSTRAINT_NAME})	定义在实体层面 (Entity-Level) 用于确认实体是否满足约束的逻辑
guard	属性状态机的转换结点中 (stateMachine/transitions/{TRAN_NAME})	状态机 (stateMachine) 的转换 (transition) 的守备条件
verificationLogic	方法结点中	对命令的检验逻辑
mutationLogic	方法结点中	应用 (Apply) 事件去修改状态的逻辑

表8-1中列出的一些關鍵字，前文已經有所介紹，後面會進一步說明。

## 8.2.1 關於accountingQuantityTypes

以下是和賬務模式相關的用於描述數量類型的計算邏輯的關鍵字，它們都需要出現  
在/configuration/accountingQuantityTypes/{QUANTITY\_TYPE}結點中。

·zeroLogic：數量“零”的表達式。

·addLogic：數量的相加邏輯。

·negateLogic：對數量“取反”的邏輯。

對於它們的使用，前文展示過DDDML代碼示例，  
其中的decimal就是數量類型：

---

```
configuration:
  # ...
  accountingQuantityTypes:
    decimal:
      zeroLogic:
        Java: "BigDecimal.ZERO"
      addLogic:
        Java: "{fst}.add({snd} != null ? {snd} : BigDecimal.ZERO)"
      negateLogic:
        Java: "{0}.negate()"
```

---

## 8.2.2 關於derivationLogic

關鍵字derivationLogic（派生邏輯）可以用來聲明一個屬性是如何派生出來的，它指的是屬性的讀方法的派生邏輯。

在使用賬務模式的情況下，一個賬目（Account）的餘額其實是不一定需要存儲在數據庫中的，因為它總是等於這個賬目所有分錄（Entries）的數量總和。也就是說，賬目的餘額可以從它的分錄派生出來。

因此，在DDML中可以採用如下方式聲明賬目餘額屬性的派生邏輯（下面使用偽代碼來表述）：

---

```
aggregates:
  Account:
    id:
      name: AccountId
      type: id
    properties:
      Entries:
        itemType: Entry
      Balance:
        type: quantity
        derived: true
        derivationLogic:
          PseudoCode: sumOf(Entries.amount)
    # ...
```

---

### 8.2.3 關於filter

關鍵字filter（過濾器）可以這樣描述屬性的過濾邏輯（使用了filter的屬性是派生屬性）：

---

```
aggregates:
  # 家庭
  Family:
    id:
      name: FamilyId
      type: id
    properties:
      Surname:
        type: name
    Members:
      itemType: FamilyMember
    YoungMembers:
      itemType: FamilyMember
      # -----
      filter:
        PseudoCode: "where age < 18"
        Java: "m -> m.getAge() < 18"
      # derived: true
      # 不需要顯式聲明，有 filter 的就是派生的

entities:
  # 家庭成員
  FamilyMember:
    id:
      name: FirstName
      type: name
    properties:
      Age:
        type: int
      # ...
```

---

## 8.2.4 使用關鍵字referenceFilter

我們可以在一個聲明瞭referenceType的屬性結點中使用referenceFilter關鍵字，限制可以被該屬性引用的實體（Reference Type）的實例。

假設，我們在為一個WMS應用建模，聚合根Locator表示貨物所在的貨位。基於一些現實因素的考量，貨位不是倉庫（Warehouse）聚合內部的實體。如果我們希望Locator的屬性WarehouseId（倉庫ID）引用的是那些已經啟用的（Active）倉庫，那麼可以按如下形式寫出相關代碼。

---

```
aggregates:
  Warehouse:
    id:
      name: WarehouseId
      type: id
    # -----
    Locator:
      id:
        name: LocatorId
        type: id-long
      properties:
        Description:
          type: string
        # ...
      WarehouseId:
        referenceType: Warehouse
        referenceName: Warehouse
      # -----
      # 只引用一個已經啟用 (Active) 的倉庫
      referenceFilter:
        Criterion:
```

```
type: "eq"
property: "active"
value: true
# ...
```

---

## 8.2.5 業務邏輯代碼中的變量

在DDDM文檔中，可能使用不同語言的代碼片段來描述關鍵的業務邏輯。DDDM規範並沒有明確規定DDDM工具應該如何處理這些代碼片段。

在實踐中，DDDM文檔的這些業務邏輯代碼很多時候都是代碼生成工具使用的模板字符串（String），這就需要我們在這些代碼中使用一些變量。在生成代碼的時候，可能需要先把這些模板中的變量替換成具體的值，然後再輸出到代碼文件中。

我們在實踐中對這些變量使用以下命名規範（不是DDDM規範的一部分）。

- 所有變量都以花括號（{}）包圍。
- 對於方法的實現邏輯，以{0}表示第一個參數，{1}表示第二個參數，{2}表示第三個參數，以此類推。
- 對於二元操作符的實現邏輯，兩個操作數（operand）分別使用{fst}與{snd}表示；三元操作符的第三個操作數以{trd}表示；一元操作符的操作數直接使用{0}表示。

·以{this}表示上下文對象。但是這裡的上下文對象指的是什麼，其實是由DDML工具根據需要自行解釋的。

## 8.2.6 說說區塊鏈

這裡打算借用區塊鏈技術中的一些概念來類比說明關鍵字**verificationLogic**、**mutationLogic**、**validationLogic**的含義。

區塊鏈實際上使用了事件溯源模式。簡單地說，“鏈”上那些被確認的“交易”，大致可以類比為事件溯源模式的“事件”。

將**DDDM**與區塊鏈技術相結合是完全有可能的。簡單地說，可以使用**DDDM**工具將一個聚合定義轉變成一個可以在區塊鏈上執行的智能合約。這也許可以大大降低區塊鏈智能合約的開發成本。

以**Corda**<sup>[1]</sup>為例，合約的執行不僅需要確認當前提交的交易合法，當前交易的輸入狀態（**Input States**）也必須有效。這就需要獲得每個輸入狀態之前的交易鏈（**Transaction Chain**），並重新執行這個鏈上的合約（**Contract**）以判斷涉及的歷史交易是否合法。整個交易鏈上的每筆交易都需要使用合約的**Verify**方法來檢查是否合法，檢查的交易信息包括命令、輸入狀態、輸出狀態等。

這和**ES**模式重放（重新應用）整個事件流、恢復聚合的當前狀態、執行命令的過程有幾分相似。在使用**ES**模式時，我們可以：

·先使用實體的方法（命令），或歷史事件關聯的實體的方法的**verificationLogic**（驗證邏輯，它是實體方法的組成部分）來檢驗在上一個狀態（最初的狀態都是些“空”的對象實例）的基礎上執行命令或應用事件是否合法。

- 如果合法則使用**mutationLogic**修改狀態。
- 然後執行**validationLogic**進一步確認獲得的新狀態是否滿足業務邏輯的約束。
- 不管是執行**verificationLogic**出錯，還是執行**validationLogic**發現輸出的狀態違反了約束，命令都會被拒絕執行。如果是在重放歷史事件的過程中，執行**verificationLogic**或**validationLogic**時發現了異常，有些時候可以考慮忽略錯誤。

[1] Corda.net. Corda | Open Source Blockchain Platform for Business, <https://www.corda.net/>。

## 8.3 領域服務

我們可以在DDDML中定義領域服務。在DDDML中，“服務”與“服務方法”是有區別的，可以認為“服務”是“服務方法”的分組。很多時候，我們口頭交談中提到的某個“服務”其實在DDDML中需要體現為一個服務方法。

也就是說，在DDDML中，總是需要在`/services/{SERVICE_NAME_OR_SERVICE_GROUP_NAME}/methods/{METHOD_NAME_OR_SERVICE_NAME}`這樣的結點下定義一個服務方法（或者說服務）。

以前文在介紹DDD戰術層面的關鍵概念時提到的轉賬服務為例，它的DDDML代碼如下：

---

```
services:
  TransferService:
    methods:
      Transfer:
        parameters:
          SourceAccountId:
            type: id
            referenceType: Account
          DestinationAccountId:
            type: id
            referenceType: Account
          Amount:
            type: Money
```

---

這裡的服務方法TransferService.Transfer會操作類型同為Account的兩個實體（實例），再來看看需要操作不同類型的實體（實例）的服務的例子。



注

意定義在實體（包括聚合根或非聚合根實體）結點中的方法，如果這個方法是實例方法（默認就是實例方法，即沒有聲明notInstanceMethod為true時就是實例方法），每次調用這個方法應該最多隻改變一個聚合實例的狀態。如果一個方法要改變多個聚合實例的狀態，那麼應該定義為服務的方法或是聚合根的非實例方法。

這裡說的聚合實例，是指聚合根實例以及通過它可以訪問到的、生命週期完全受它控制的聚合內部其他實體的實例，它們作為一個整體稱為聚合實例。

以下示例來自筆者開發的一個CRM系統。我們在服務端提供了一個線索跟進服務，銷售人員對線索（Lead）執行跟進（比如電話溝通）後，需要在系統中填寫所瞭解的信息，這時客戶端需要調用這個線索跟進服務。這個服務不只是會更新Lead的信息，還要添加跟進（Followup）的記錄，這是一個跨聚合的操作，所以它是一個領域服務。相關的聚合與服務的DDML代碼片段如下：

---

```
aggregates:
```

```
  Lead:
```

```
    id:
```

```
        name: LeadId
        type: id
    # ...
Followup:
    id:
        name: FollowupId
        type: id
    properties:
        Note:
            type: long-varchar

services:
# -----
# 線索跟進服務
LeadFollowupService:
# -----
# 方法
methods:
# -----
# 跟進後更新
UpdateAfterFollowup:
    parameters:
        # 線索 Id.
        LeadId:
            type: id
            referenceType: Lead
        # 稱謂 ( 稱呼 )
        Salutation:
            type: name
        # 跟進備註
        FollowupNote:
            type: long-varchar
    # ...
```

---

## 8.4 在方法定義中使用關鍵字 inheritedFrom

之前已經介紹過，關鍵字 `inheritedFrom` 可以用於表示一個實體（現在僅限於聚合根）是另外一個實體的子類型。DDDML 還允許在定義方法時使用它來聲明一個方法繼承自另外一個方法，這表示該方法的參數列表包括了所繼承的方法的所有參數。

以下是筆者開發的一個 CRM 系統示例：

---

```
aggregates:
# -----
Party:
  id:
    name: PartyId
    type: id-long
  # ...
  methods:
    Create:
      parameters:
        # ...
# -----
# 職位
Position:
  id:
    name: PositionId
    type: id
  # ...
# -----
# 員工履職記錄
PositionFulfillment:
  id:
    name: PositionFulfillmentId
```

```
        type: PositionFulfillmentId
    # ...

services:
# -----
# 員工-職位服務
EmployeePositionService:
# -----
# 方法
methods:
# -----
# 創建員工 ( Party ) 與職位履行記錄
CreateEmployeeAndPositionFulfillments:
    inheritedFrom: "Party.Create"
    parameters:
        # 履行的職位 ID 的集合
        FulfilledPositionIds:
            itemType: id
```

---

首先說明，這裡的Employee（員工）指的是“扮演”了員工角色的那些Party的實例。

## 服務方法

( EmployeePositionService.CreateEmployeeAndPositionFulfillments ) 想要做的是：在創建員工記錄 ( Party 的實例 ) 的同時，創建員工的職位履行記錄。

我們讓這個方法繼承自 ( Inherited From ) Party 實體的Create方法，然後添加一個參數 FulfilledPositionIds——因為一個員工可能在公司中“身兼多職”，所以這裡允許傳入多個職位的ID ( 參數 FulfilledPositionIds的類型是ID的集合 ) 。

客戶端在創建員工記錄時，可能會傳入很多參數（假設實體Party有很多屬性），如果沒有**inheritedFrom**的支持，這個地方可能就要多寫很多冗餘的代碼。

## 8.5 方法的安全性

DDDML支持在方法的定義中聲明客戶端調用方法所需的授權。這裡的授權可以是角色或權限。

以下示例聲明瞭客戶端調用User的Create方法所需的授權：

---

```
aggregates:
  User:
    id:
      name: UserId
      type: id-long
    properties:
      # ...
    # -----
    methods:
      Create:
        authorizationEnabled: true
        requiredAnyRole: [SystemAdministrator]
        requiredAnyPermission: [UserManagement]
      parameters:
        # ...
```

---

關鍵字requiredAnyRole以及requiredAnyPermission的值類型都是String的列表。這裡的DDDML代碼表達的意思是，客戶端想要訪問這個Create方法需要取得相應的授權：擔任requiredAnyRole列表中的任一角色，或者獲得requiredAnyPermission列表中的任一權限。

DDDML代碼生成工具可能會將這些信息映射到使用特定的安全框架的實現代碼中，比如，在使用 **Spring Security** 的情況下，可能會在生成的方法前添加如下註解（Java代碼）：

---

```
@PreAuthorize("hasAnyAuthority('SystemAdministrator',  
'UserManagement')")  
public void create(CreateUser cmd) {  
    // ...  
}
```

---

## 第9章 模式

模式是可複用的解決方案，用於解決軟件設計中反覆遇到的問題。有些問題總是反覆出現，然後一次地被大家“解決”，於是人們對問題進行抽象，在抽象的基礎上思考、總結問題的解決方案，然後為其中的某種解決方案命名，進而得到一個模式。可見，模式是被前人驗證過的解決方案。

**DDDML**支持在領域模型中的某個地方聲明我們想要應用的某種模式，比如像“賬務模式”這樣的分析模式。支持這些模式可以大大增加**DDDML**的表現力，我們往往只需要使用少量的代碼就可以指出所構建的模型的意圖——我們碰到的問題以及決定採用的解決方案，然後讓**DDDML**工具來生成方案的執行細節。

本章會介紹**DDDML**當前支持的三種模式：賬務模式、狀態機模式，以及樹結構模式。

## 9.1 賬務模式

領域中重要數量 (Quantity) 的變化過程必須可追溯。所謂的“可追溯”，也就是說我們需要記錄數量變化的歷史。

賬務模式提供的解決方案是：數量的每次變化都記錄為賬目的一個條目，合計條目得到賬目的當前值（也就是我們常說的餘額）。這裡面有如下兩個重要的概念。

·**Account**：賬目，或者叫作賬戶、科目。什麼是賬目？基本上，領域裡面所有重要的需要追溯其變化過程的數量（或者說數字）都是賬目，一個數量就是一個賬目。

·**Entry**：條目，財務人員可能會把它叫作分錄，俗稱的“流水”往往指的也是它。一般來說，條目應該是個不變的實體。條目需要包含觸發數量變化（也就是條目生成）來源事件 (Event) 的信息。

這裡所說的數量，往往指帶著度量單位 (UoM) 的數字。數量是一個值對象，應該把它看成一個整體。有時候在特定上下文中，度量單位是人盡皆知的，這時候的數量可能僅僅需要使用一個數字 (Amount) 就可以表示。

以上就是一個最簡單的賬務處理模型——所謂的單式記賬法——使用的關鍵概念。在財務領域，重要數量（金額）的變化往往需要使用所謂的複式記賬法進行記錄，只有這樣才能滿足財務審計需求。如需要支持複式記賬法，模型中需要引入更多的概念，比如交易（**Transaction**）。當前的**DDDML**規範不包含描述複式記賬模型需要的關鍵字。但開發團隊可以在**DDDML**描述的單式記賬模型的基礎上，自行實現複式記賬所需要的業務邏輯。

即使**DDDML**只對賬務模式提供了最基本的支持，我們仍然認為意義重大。因為我們已經看到過太多因為“這系統連數都算不對”引發的慘劇。這實在不能過多責怪開發人員，要掌握基本的賬務處理原則和模式需要付出相當大的學習成本。在**Martin Fowler**的《分析模式》[\[1\]](#)一書中，賬務模式是唯一使用了兩章內容來闡述的模式。**DDDML**對賬務模式的支持絕對可以讓開發團隊，特別是缺乏賬務處理經驗的開發人員，在需要處理重要“數字”的時候少走彎路，節省時間。

下面是一個關於庫存單元的例子，出自我們曾經開發的某個**WMS**應用：

---

```
aggregates:
  # 庫存單元
  InventoryItem:
    id:
      name: InventoryItemId
      type: InventoryItemId
```

```

properties:
    # 在庫數量
    OnHandQuantity:
        type: decimal
    # 在途數量
    InTransitQuantity:
        type: decimal
    # 保留數量
    ReservedQuantity:
        type: decimal
    # 佔用數量
    OccupiedQuantity:
        type: decimal
    # 虛擬數量
    VirtualQuantity:
        type: decimal
    # -----
    # 庫存單元條目 ( 分錄 )
    Entries:
        itemType: InventoryItemEntry

reservedProperties:
    noDeleted: true

entities:
    # ----- 庫存單元條目 ( 分錄 ) -----
    -----
    InventoryItemEntry:
        immutable: true
        id:
            name: EntrySeqId
            type: long
            columnName: EntrySeqId

properties:
    # 在庫數量 ( 變化值 )
    OnHandQuantity:
        type: decimal
    # 在途數量 ( 變化值 )
    InTransitQuantity:
        type: decimal
    # 保留數量 ( 變化值 )
    ReservedQuantity:
        type: decimal
    # 佔用數量 ( 變化值 )

```

```

        OccupiedQuantity:
            type: decimal
            # 虛擬數量 ( 變化值 )
        VirtualQuantity:
            type: decimal
            # 條目來源 ( 引起數量變化的事件信息 )
        Source:
            type: InventoryItemSourceInfo
            notNull: true

        # -----
        # 庫存事務 ( 交易 ) 的發生時間
        OccurredAt:
            type: date-time
            notNull: true

    reservedProperties:
        noActive: true

    # -----
    # 唯一約束
    uniqueConstraints:
        # 一個“來源” ( 事件 ) 不能重複產生條目 ( 分錄 )
        UniqueInventoryItemSource: [Source]
    # -----


    # ----- Accounts -----
    accounts:
        # 在庫數量。聚合根中的“在庫數量” ( 屬性 ) 是一個“賬目”
        OnHandQuantity:
            # 條目實體名稱
            entryEntityName: "InventoryItemEntry"
            # 條目數額屬性名稱
            entryAmountPropertyName: "OnHandQuantity"
        # 在途數量
        InTransitQuantity:
            entryEntityName: "InventoryItemEntry"
            entryAmountPropertyName: "InTransitQuantity"
        # 保留數量
        ReservedQuantity:
            entryEntityName: "InventoryItemEntry"
            entryAmountPropertyName: "ReservedQuantity"
        # 佔用數量
        OccupiedQuantity:
            entryEntityName: "InventoryItemEntry"

```

```

        entryAmountPropertyName: "OccupiedQuantity"
    # 虛擬數量
    VirtualQuantity:
        entryEntityName: "InventoryItemEntry"
        entryAmountPropertyName: "VirtualQuantity"

# ----- Value Objects -----
valueObjects:
    # -----
    # 庫存單元 Id
    InventoryItemId:
        properties:
            # 產品 Id
            ProductId:
                type: id-long
                columnName: ProductId
                length: 60
            # 貨位 Id.
            LocatorId:
                type: string
                columnName: LocatorId
                length: 50
            # 屬性集實例 Id.
            AttributeSetInstanceId:
                type: string
                columnName: AttributeSetInstanceId
                length: 50

    # -----
    # 庫存單元來源 (事件) 信息
    InventoryItemSourceInfo:
        properties:
            # 單據類型 Id.
            DocumentTypeId:
                type: string
                referenceType: DocumentType
                notNull: true
                columnName: DocumentTypeId
            # 單據號
            DocumentNumber:
                type: string
                notNull: true
                columnName: DocumentNumber
            # 行號
            LineNumber:

```

```
    type: string
    columnName: LineNumber
    # 行的子序列號 ( 一個源單據行項可能產生多個庫存
    事務條目 )
    LineSubSeqId:
        type: int
        columnName: LineSubSeqId
```

---

在這個**DDML**文檔中，聚合**InventoryItem**（庫存單元）表示的是當前的庫存信息。**InventoryItemEntry**是聚合內部實體，表示庫存單元的條目。

從邏輯上說，**OnHandQuantity**、**InTransitQuantity**等數量都是賬目，雖然它們在數據庫中僅對應著同一個表的不同的列。

以庫存單元實體（聚合根）下的屬性**OnHandQuantity**為例，這個重要的數量——它的單位存在於屬性**ProductId**引用的產品信息中——是一個賬目。這個數量是不能直接修改的，想要修改它需通過增加條目（Entry）來完成。在這裡，表示條目的實體就是**InventoryItemEntry**，在條目中表示“變化數量”的屬性名也叫**OnHandQuantity**（聲明該變化數量的屬性名的代碼為：  
`entryAmountPropertyName:"OnHandQuantity" )`。

另外，這裡還使用**uniqueConstraints**關鍵字聲明瞭一個名為**UniqueInventoryItemSource**的唯一約束，因為對於一個觸發條目生成的來源事件我們只應該處理一次。

[1] Martin Fowler. 分析模式：可複用的對象模型. 機械工業出版社，2010. 見 <https://book.douban.com/subject/4832380/>。

## 9.2 狀態機模式

為了幫助開發人員處理領域中那些重要的“數量”的變化，**DDDML**提供了對賬務模式的支持。而領域中有些實體可能具有一些重要的定性屬性，它們的變化規則可以使用有限狀態機（簡稱狀態機）來表示。對軟件開發來說，狀態機是個非常有用的抽象，所以**UML**專門提供了狀態機圖來描述它們，**DDDML**自然也不能錯過對狀態機模式的支持。

以下示例出自我們開發的某個**WMS**應用。入庫/出庫單的單據狀態（**DocumentStatusId**）的轉換（**Transitions**）規則可以使用一個狀態機來表述，這個聚合的**DDDML**文檔如下：

---

```
aggregates:
  # 入庫/出庫單
  InOut:
    id:
      name: DocumentNumber
      type: string
    properties:
      # -----
      # 單據狀態 Id
      DocumentStatusId:
        type: string
        commandType: DocumentAction
        commandName: DocumentAction

      # -----
      # 單據狀態的狀態機
      stateMachine:
        # 轉換
```

```

        transitions:

        - sourceState: null
          trigger: null
          targetState: "Drafted"

        - sourceState: "Drafted"
          trigger: "Complete"
          targetState: "Completed"

        - sourceState: "Drafted"
          trigger: "Void"
          targetState: "Voided"

        - sourceState: "Completed"
          trigger: "Close"
          targetState: "Closed"

        - sourceState: "Completed"
          trigger: "Reverse"
          targetState: "Reversed"

enumObjects:
# -----
DocumentAction:
  baseType: string
  values:
    Draft:
      description: Draft
    Complete:
      description: Complete
    Void:
      description: Void
    Close:
      description: Close
    Reverse:
      description: Reverse

```

---

上面的DDML代碼使用關鍵字stateMachine在屬性DocumentStatusId結點中定義了一個狀態機。狀態機包括一組轉換 ( transitions )，trigger是觸發轉換的事

件或命令。在DDML規範中並沒有限定 stateMachine/transitions中trigger的值類型。

可以看到，這個示例中還存在一個屬性命令（ DocumentAction ）。前文討論過屬性命令的問題。我們在一個屬性中定義命令後，不一定要定義這個屬性的狀態機。但是反過來，如果這個屬性上定義的狀態機的轉換是由命令觸發的，則可以考慮給這個屬性定義一個命令（聲明屬性的命令類型及命令名稱）。生成代碼的時候，DDML工具應該儘可能地實現 trigger的值類型與命令類型之間的轉換。

圖9-1是使用筆者製作的DDML工具通過DDML代碼生成的狀態機圖。

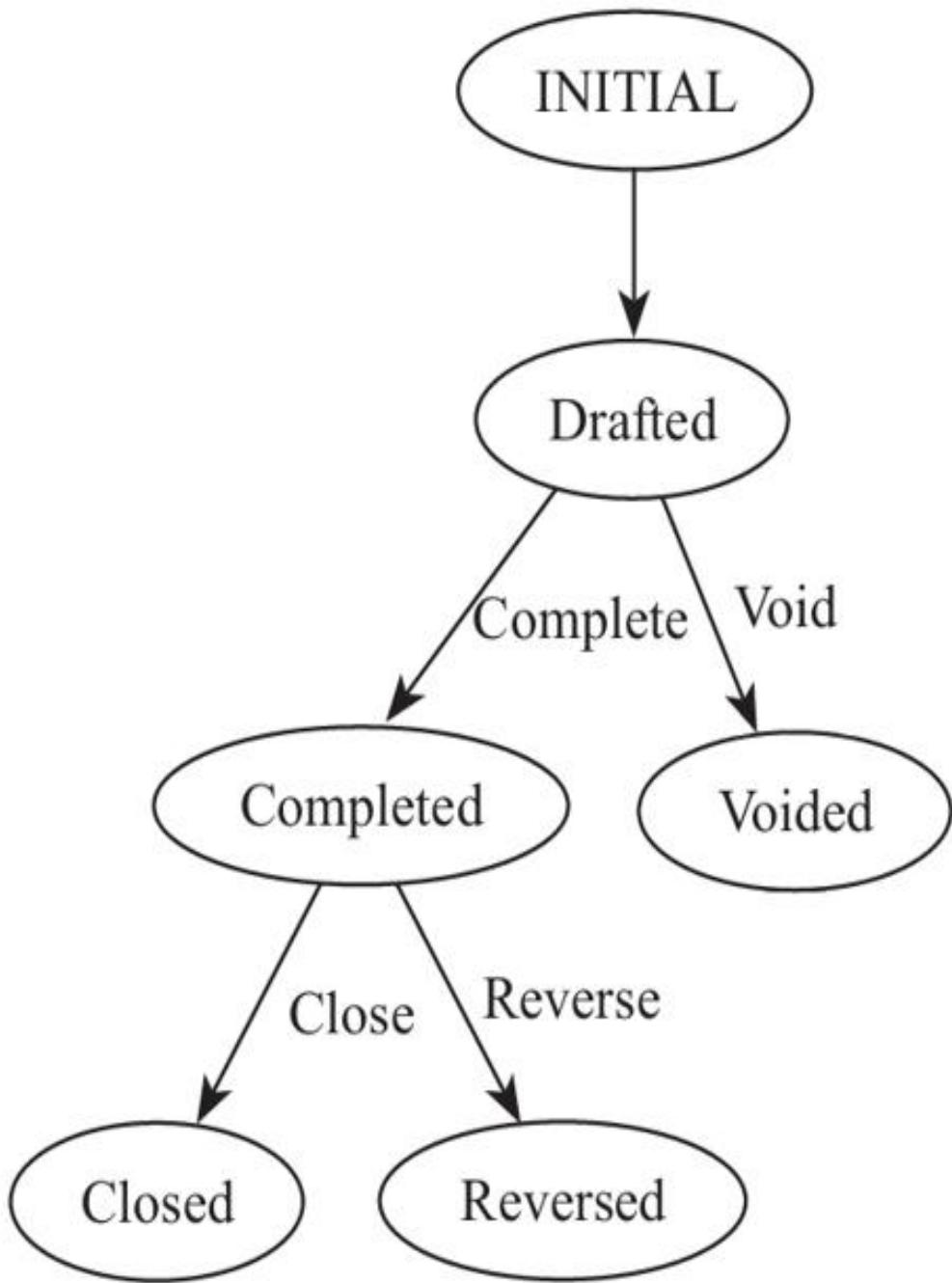


圖9-1 入庫/出庫單的狀態機

從圖9-1可以清楚地看到“入庫/出庫單”實體的單據狀態的轉換規則：

- 單據創建的時候，會被初始化為“已起草” ( Drafted ) 狀態。
- 當有人對已起草的單據執行一個“完成” ( Complete ) 操作時，它就會轉換到“已完成” ( Completed ) 狀態。
- 當有人對已完成的單據執行一個“關閉” ( Close ) 操作時，它就會轉換到“已關閉” ( Closed ) 狀態。
- 當有人對已完成的單據執行一個“反轉” ( Reverse ) 操作時，它就會轉換到“已反轉” ( Reversed ) 狀態。
- 當有人對已起草的單據執行一個“撤銷” ( Void ) 操作時，它就會轉換到“已撤銷” ( Voided ) 狀態。

DDDML還允許給狀態的轉換 ( transition ) 加上守備條件 ( guard )，示例如下。

---

```
aggregates:
  # 庫存移動單 ( 調撥單 )
  Movement:
    id:
      name: DocumentNumber
      type: string

    properties:
      # 單據狀態 Id.
      DocumentStatusId:
        type: string
      commandType: DocumentAction
      commandName: DocumentAction
```

```

stateMachine:
  # 轉換
  transitions:

  - sourceState: null
    trigger: null
    targetState: "Drafted"

  - sourceState: "Drafted"
    trigger: "Void"
    targetState: "Voided"

  - sourceState: "Drafted"
    trigger: "Complete"
    targetState: "Completed"
    # 守備條件 (如果不是在途單據，直接轉換到"已完
    成"狀態)

  guard:
    Java: "{this}.getIsInTransit() ==
false"
    CSharp: "{this}.IsInTransit ==
false"

  - sourceState: "Drafted"
    trigger: "Complete"
    targetState: "InProgress"
    # 守備條件 (如果是在途單據，轉換到"進程中"狀
    態)

  guard:
    Java: "{this}.getIsInTransit() ==
true"
    CSharp: "{this}.IsInTransit ==
true"

  - sourceState: "InProgress"
    trigger: "Confirm"
    targetState: "Completed"

  - sourceState: "Completed"
    trigger: "Close"
    targetState: "Closed"

  - sourceState: "Completed"
    trigger: "Reverse"

```

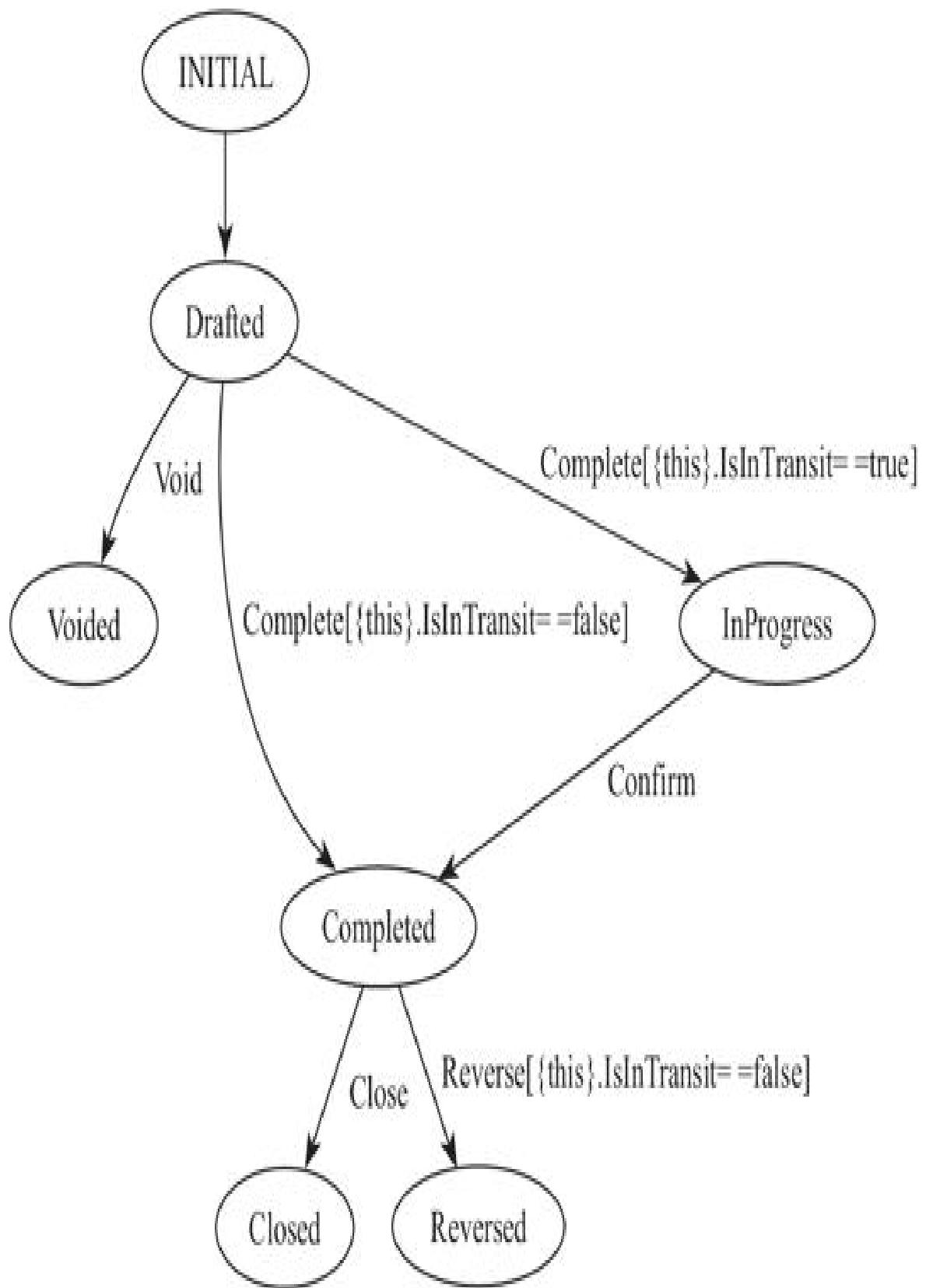
```
targetState: "Reversed"
guard:
    Java: "{this}.getIsInTransit() == false"
    CSharp: "{this}.IsInTransit == false"

MovementDate:
    type: DateTime
    # ...
```

---

在DDDML規範中，狀態機轉換的guard結點的值類型是個Map<String, Object>。可以選擇在Map中記錄Java、CSharp、偽代碼等代碼。

圖9-2是使用筆者製作的DDDML工具基於上面的狀態機定義生成的狀態機圖。



## 圖9-2 庫存移動單的狀態機

領域模型是用來交流的。一圖勝千言，筆者實在是太喜歡狀態機圖這樣的模型信息的圖形化展示了。但是很遺憾，因為所在團隊的資源限制，我們使用的DDML工具在這方面做得還太少。

## 9.3 樹結構模式

在領域建模時，我們經常會碰到一些需要構建層次關係的實體，樹是用來描述層次關係的一種數據結構。比如一個公司的內部組織，它的各個部門、分支機構，可能就會組成一個甚至多個層次結構。

DDDML目前支持使用兩種方式來定義樹。

### 9.3.1 簡單的樹

最簡單的樹，樹結點的內容以及樹結點的關係都體現為同一個實體的屬性。

舉例來說，倉庫中的貨位（ Locator ），它們可能是按照“排（ x ）”“貨架（ y ）”“層（ z ）”這樣的結構來劃分的。“1排3架”是一個貨位，“1排3架第1層”也是一個貨位，前者是後者的父貨位。

下面先定義一個 Locator 的實體（聚合根）：

---

```
aggregates:
  Locator:
    id:
      name: LocatorId
      type: id-long
    properties:
      WarehouseId:
        referenceType: Warehouse
        referenceName: Warehouse
      ParentLocatorId:
        type: id-long
      # ...
```

---

然後定義一個名字叫作 LocatorTree 的樹，在 DDDML 中只需要如下幾行即可：

---

```
trees:
  # -----
  # 貨位樹
```

---

```
LocatorTree:  
  nodeContentType: Locator  
  parentId: ParentLocatorId  
  rootParentIdValues: [null, ""]
```

---

在這個例子，我們：

- 使用關鍵字**nodeContentType**指明樹結點的內容類型是**Locator**（貨位）這個實體。
- 使用關鍵字**parentId**指明（實體**Locator**中）屬性**ParentLocatorId**是一個指向父結點的**ID**（**Parent ID**）。
- 使用關鍵字**rootParentIdValues**指出當一個結點的**Parent ID**是**null**或者是空字符串時，它就會被認為是根結點。根結點是沒有父結點（**Parent**）的，所以“根結點的**Parent ID**”一般都是特殊的值。

## 9.3.2 使用關鍵字structureType

在9.3.1節的例子中，使用nodeContentType指定了樹結點的內容類型（實體），它可以使用一個“Parent ID”屬性來指向自己的父結點。但是也可以使用一個單獨的實體來表示樹結點的關係，我們把它叫作樹的結構類型（structureType），它和結點的內容類型（實體）可以不是同一個實體。

比如說，在一個企業裡面，可能存在“銷售體系”“客服體系”這種類型不同的組織樹（OrganizationTree）。企業的一個部門（內部組織）可能在這兩個體系裡面承擔了不同的職責。我們可以使用DDML描述這樣的情況：

---

```
aggregates:
  Organization:
    id:
      name: OrganizationId
      type: string
    properties:
      Name:
        type: string
      Description:
        type: string
      # ...

    # -----
OrganizationStructureType:
  id:
    name: Id
    type: string
  # ...
```

```

# -----
OrganizationStructure:
  id:
    name: Id
    type: OrganizationStructureId
  properties:
    # ...
  valueObjects:
    # ----- valueObject -----
    OrganizationStructureId:
      properties:
        OrganizationStructureTypeId:
          referenceType:
OrganizationStructureType
  ParentId:
    referenceType: Organization
  SubsidiaryId:
    referenceType: Organization

trees:
# -----
OrganizationTree:
  nodeContentType: Organization
  structureType: OrganizationStructure
  parentId: Id.ParentId
  childId: Id.SubsidiaryId
  #rootParentIdValues: [""]

```

---

相較於前面構造的簡單的貨位樹，在這個組織樹的定義中多了兩行代碼：

- 使用關鍵字 **structureType** 指定用於構造組織樹的結構類型是實體 **Organization-Structure** 。
- 使用關鍵字 **childId** 指明結構類型（實體 **OrganizationStructure** ）的屬性 **Id.SubsidiaryId** 是指向子結點的 **Child ID** 。

為什麼在這個組織樹的定義中**parentId**的值是**Id.ParentId**？你大概已經注意到實體**OrganizationStructure**的**ID**類型是數據值對象**OrganizationStructureId**，它的屬性包括**ParentId**和**SubsidiaryId**。

另外，上面的代碼中沒有指定關鍵字**rootParentIdValues**的值（注意最後一行代碼是註釋掉的）。那麼，如果一個組織（Organization）的**ID**沒有出現在任何**OrganizationStructure**實例的**Id.SubsidiaryId**屬性中，它就應該被認為是**OrganizationTree**的根結點。

其實我們也可以在**OrganizationTree**中使用**rootParentIdValues**，示例如下：

---

```
# -----
trees:
  OrganizationTree:
    nodeContentType: Organization
    structureType: OrganizationStructure
    parentId: Id.ParentId
    childId: Id.SubsidiaryId
    rootParentIdValues: [ "", "_NULL_" ]
```

---

最後一行代碼聲明“Parent ID”的值為空字符串或“\_NULL\_”的**Organization-Structure**的實例的“**Child ID**”指向那些作為根結點的組織。

### 9.3.3 使用關鍵字structureTypeFilter

在上面的DDML示例中，實體OrganizationStructure的ID中包含了“組織結構類型”(Organization Structure Type)的ID，我們可以用不同的組織結構類型來區分各種組織樹。

下面使用關鍵字structureTypeFilter聲明：只有符合一定條件的結構類型的實例，才能用於構成當前的樹結構。關鍵字structureTypeFilter的值類型是Map<String, Object>。以下是DDML示例代碼：

---

```
trees:
  TestOrganizationTree:
    nodeContentType: Organization
    structureType: OrganizationStructure
    parentId: Id.ParentId
    childId: Id.SubsidiaryId
    # -----
    structureTypeFilter:
      Criterion:
        type: "or"
        lhs:
          type: "eq"
          property:
            "Id.OrganizationStructureTypeId"
            value: "test-org-struct-type3"
        rhs:
          type: "eq"
          property:
            "Id.OrganizationStructureTypeId"
            value: "test-org-struct-type1"
```

---

在上面的代碼中，結點/trees/TestOrganizationTree/structureTypeFilter/Criterion的值表明組織結構類型ID（OrganizationStructureTypeId）為test-org-struct-type3或test-org-struct-type1的那些組織結構（OrganizationStructure）的實例，構造了名為TestOrganizationTree的樹。

DDDML代碼生成工具可以根據這些樹的定義生成代碼，包括服務端的樹結點的查詢代碼以及在客戶端應用UI中的樹的展示代碼。

## 第三部分 實踐

- 第10章 處理限界上下文與值對象
- 第11章 處理聚合與實體
- 第12章 處理領域服務
- 第13章 RESTful API
- 第14章 直達UI

## 第10章 處理限界上下文與值對象

前面的章節介紹瞭如何設計一個DDD原生的DSL，並闡述瞭如何使用DDDML去描述DDD風格的領域模型中重要的方面。但是我們不想僅僅停留在這麼“表面”的工作上，從本章開始將講述如何使用DDDML工具將“夢想照進現實”——也就是將DSL描述的領域模型忠實地映射到代碼上。

接下來會展示為了從DDDML文檔產生“工作的軟件”我們所做過的一些工作。也會大量地展示筆者製作的DDDML工具生成的軟件源代碼，它們一般是Java代碼或C#代碼。

筆者對如下說法表示贊同：想要實現DDD，其實不需要那麼多DDD框架，我們需要更多好的示例。而本章提供的正是在真實的生產環境下采用過的一些做法，供讀者參考。

對於這些生成的代碼，有時候你可能會覺得它們略顯煩瑣。確實，其中不少代碼是可以通過某些不太難實現的方式簡化掉的。比如，可以選擇不生成靜態類型的代碼，而是在代碼中使用一些“動態對象”——比如Java的Map、.NET的IDictionary；又比如，只生成接口的代碼，而不生成實現接口的類的代碼，因為我們可以使用動態代理技術在運行時生成接口的實現；

再比如，可以使用一些Bean工具來實現對象之間的複製，而不是生成那麼多執行複制操作的代碼……

如果有必要，完全可以在代碼中採用更“動態”的實現方案。但是，這裡展示靜態類型的代碼，有助於讓你更容易看清楚其中的實現邏輯（How），從而真正理解我們在DDML中設計那些“抽象”的意圖和確切的含義（What）。

到目前為止，筆者實踐中在DDML文檔的valueObjects結點下定義的值對象都是一些沒有定義方法的數據值對象。在本章中讀者會看到筆者製作的DDML工具處理這些相對簡單的值對象的一些做法。

DDML的/typeDefinitions結點下定義的值對象，一般都是所謂的領域基礎類型。在代碼實現層面，除了確定將它們映射為特定語言的類型之外，可能還需要考慮以下問題：

- 這些領域基礎類型怎麼持久化？
- 這些類型怎麼序列化/反序列化？
- 在前端用什麼UI組件/控件來呈現它們？
- 在RESTful API中如何使用它們作為查詢參數進行查詢？

在本章的示例代碼中，主要展示前兩個問題的一些解決方案。對於第三個問題的處理，後面章節會做進一步探討。至於第四個問題，本書沒有過多闡述，下面舉個例子幫助讀者更好地理解這個問題。

假設**Shipment**（裝運單）有一個類型為**Money**的屬性**FreightAmount**，這個**Money**類型在Java代碼中被映射為**Joda Money**類庫的**Money**類，我們可能希望通過向下面的三個URL分別發送HTTP GET請求以獲取符合特定條件的裝運單列表：

---

```
{BASE_URL}/Shipments?FreightAmount.Currency=CNY

{BASE_URL}/Shipments?filter=
{type:"eq",property:"FreightAmount.Currency",value:"CNY"}

{BASE_URL}/Shipments?filter=
{type:"ge",property:"FreightAmount.Amount",value:"400"}
```

---

這個問題如何處理？留待讀者自己尋找答案。

## 10.1 項目文件

首先要做的是將一個限界上下文中所有模型的**DDDML**文檔都作為一個項目組織起來，讓**DDDML**工具可以通過項目找到一個限界上下文的完整模型描述。

因為一個上下文的模型之間多少存在關聯，所以在多數時候，筆者製作的**DDDML**工具會把屬於同一個上下文中的所有**DDDML**文件在邏輯上合併為一個**DDDML**文檔，然後再進行下一步的處理。

我們可以指定**DDDML**工具使用一個項目目錄或者使用一個項目文件來載入一個上下文的完整領域模型。



提示在口頭交流時，筆者有時會把上下文稱為“項目”。

一般來說，一個限界上下文的所有**DDDML**文件都會放到一個目錄下面，我們把這個目錄叫作限界上下文的項目目錄。有時候我們希望指定被合併的**DDDML**文件，不管這些文件是不是都位於一個目錄下——可使用一個項目文件來實現這個功能。項目文件還可以作為我們重用**DDDML**代碼的一種方式，因為它可以引用任意目錄中的**DDDML**文件。

項目文件可以聲明包含或者排除哪些**DDDML**文件。

下面舉例說明。比如，在開發一個**WMS**應用時，在代碼倉庫的**dddml**目錄下，存在兩個項目文件：

**wms.project**

**iam.project**

前者是**WMS**應用的“主”項目文件，後者是相對獨立的“身份與訪問管理”應用（很多人稱之為用戶系統）的項目文件。這兩個應用的**DDDML**文件都放在同一個目錄中，但是通過這兩個不同的項目文件（入口），可以把這些**DDDML**文件作為兩個獨立的項目（上下文）區分開來。

其中**wms.project**項目文件的內容（項目文件也是基於**YAML**的）如下：

---

```
project:
  directories:
    - path: "."
      pattern: "*.yaml"
      fileExclusions:
        - "IamBoundedContextConfig.yaml"
        - "Audience.yaml"
        - "IdentityManagement.yaml"
        - "AccessManagement.yaml"

  files:
    -
"../Dddml.Common.Metadata/AttributeSetInstance.dddml.yaml"
```

```
-  
"../Dddml.Common.Metadata/AttributeSetInstanceExtensionField  
Group.\dddml.yaml"
```

---

上面這個項目文件描述的是：加載當前目錄的所有yaml ( DDDML ) 文件，但是，需要排除掉其中四個文件 ( `IamBoundedContextConfig.yaml` 、`Audience.yaml` 等 ) ；然後，還需要額外加載在結點/`project/files`中指定的兩個yaml文件。

項目文件`iam.project`的內容如下：

---

```
project:  
  files:  
    - "typeDefinations.yaml"  
    - "IamBoundedContextConfig.yaml"  
    - "Audience.yaml"  
    - "IdentityManagement.yaml"  
    - "AccessManagement.yaml"
```

---

上面這個項目文件描述的是：只需要加載五個yaml ( DDDML ) 文件。

## 10.2 處理值對象

到目前為止，筆者實踐中在DDML文檔的 `valueObjects` 結點下定義的值對象都是一些沒有定義方法的數據值對象。DDML工具對於這樣的值對象的處理相對簡單，以生成Java代碼為例，代碼生成工具會按照在DDML中定義的每個屬性生成相應的 `field`、`getter` 與 `setter` 方法。對於這些數據值對象，很多工具，比如JSON序列化類庫的默認處理邏輯可能就是我們想要的。

## 10.2.1 一個需要處理的數據值對象示例

為了說明DDDDML工具如何處理數據值對象，假設有如下DDDDML文檔：

---

```
valueObjects:
  # -----
  PersonalName:
    properties:
      FirstName:
        # sequenceNumber: 0
        type: string
      LastName:
        # sequenceNumber: 1
        type: string
  # -----
  Contact:
    properties:
      PersonalName:
        type: PersonalName
      PhoneNumber:
        type: string
      Address:
        type: string
  # -----
  PersonId:
    properties:
      PersonalName:
        type: PersonalName
      SequenceId:
        type: int

  # -----
aggregates:
  Person:
    id:
      name: PersonId
      type: PersonId
    properties:
```

```
# -----
Titles:
    itemType: string
# ...
EmergencyContact:
    type: Contact
# ...
```

---

在這個例子裡，聚合根**Person**（人）的屬性**EmergencyContact**（緊急聯繫人）的類型是值對象**Contact**。值對象**Contact**的屬性**PersonalName**（人名）的類型（**type**）是一個叫作**PersonalName**的數據值對象。我們經常會碰到類似**Contact**這樣的“嵌套數據值對象的數據值對象”。

我們還可以看到，**Person**的**ID**類型是值對象**PersonId**，它的屬性包括**PersonalName**（人名）與**SequenceId**（序號）——因為不同的人可能有同樣的名字，所以我們需要加上一個序號，使用兩者的組合作為“人”的標識。這是一個略顯生硬的例子，也許你會認為這裡使用一個**UUID**作為**Person**的**ID**就可以了，這裡只是為了展示在代碼層面如何處理這樣“複雜”的數據值對象。

也許有讀者會認為，我們可以避免使用這樣“複雜”的值對象作為實體的**ID**。領域模型應該是整個軟件開發團隊交流的基礎，並且應該被忠實地影射到實現代碼中。如果在領域建模的時候，大家一致認為像**PersonId**這樣“複雜”的數據值對象就應該是某個實體

的ID，那麼DDDML就應該允許這麼做，而不是因為技術上的原因要求大家繞過。

事實上，實體ID的選擇，在概念建模階段就應該仔細考量。那種不分青紅皂白地要求實體必須使用一個沒有任何業務含義的ID（代理主鍵）的做法是值得商榷的，前文對此已經做過討論。我們應該把實體不變的、具備唯一性的屬性建模為它的ID——我們認為ID是實體的一個特殊屬性。如果實體有多個具備唯一性的屬性，那麼選擇在現實世界中有最多機會使用它來查詢、獲取（GET）實體實例（狀態）的那個屬性。在確定好實體的ID之後，就應該儘可能地使用它來引用（獲取）實體的實例。在數據庫層面，考慮給實體的ID建立聚集索引，聚集索引可能是非常寶貴的（一個表只有一個）。恰當地選擇實體的ID也有助於實現底層數據存儲的多樣性（比如使用Cassandra、MongoDB這樣的NoSQL數據庫）。在某些分佈式存儲系統中，除了聚合根的ID，其他屬性的唯一性可能是沒有強一致性保證的。

## 10.2.2 使用Hibernate存儲數據值對象

考慮一個問題，如果我們在Java代碼中，使用了Hibernate ORM框架來實現Person實體的持久化，那麼如何處理像EmergencyContact這樣（類型為Contact）的屬性呢？

先看一下使用筆者製作的DDDML工具為值對象PersonalName生成的Java代碼，大致如下：

---

```
package org.dddml.templates.tests.domain;

import java.io.Serializable;
import org.dddml.templates.tests.domain.*;

public class PersonalName implements Serializable {
    private String firstName;
    public String getFirstName() {
        return this.firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    private String lastName;
    public String getLastName() {
        return this.lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public PersonalName() {
    }
    public PersonalName(String firstName, String lastName) {
        this.firstName = firstName;
    }
}
```

```
        this.lastName = lastName;
    }

@Override
public boolean equals(Object obj) {
    // 省略代碼
}

@Override
public int hashCode() {
    // 省略代碼
}

}
```

---

也許有讀者會建議這裡生成一個不可變的（**immutable**）**PersonalName**類——也就是把它的那些**setter**方法去掉——是一個更好的選擇。筆者也贊同，只是這可能會給**PersonalName**的序列化/反序列化以及持久化帶來更多的挑戰。為了簡化問題，我們還是先選擇為數據值對象的屬性都生成**getter**和**setter**方法。

DDDML工具為**Contact**值對象生成的Java代碼如下：

---

```
public class Contact implements Serializable {
    private PersonalName personalName = new PersonalName();
    public PersonalName getPersonalName() {
        return this.personalName;
    }
    public void setPersonalName(PersonalName personalName) {
        this.personalName = personalName;
    }

    private String phoneNumber;
    public String getPhoneNumber() {
```

```
        return this.phoneNumber;
    }
    public void setPhoneNumber(String phoneNumber) {
        this.phoneNumber = phoneNumber;
    }

    private String address;
    public String getAddress() {
        return this.address;
    }
    public void setAddress(String address) {
        this.address = address;
    }

    // 注意這裡
    protected String getPersonalNameFirstName() {
        return getPersonalName().getFirstName();
    }
    protected void setPersonalNameFirstName(String personalNameFirstName) {
        getPersonalName().setFirstName(personalNameFirstName);
    }

    // 注意這裡
    protected String getPersonalNameLastName() {
        return getPersonalName().getLastName();
    }
    protected void setPersonalNameLastName(String personalNameLastName) {
        getPersonalName().setLastName(personalNameLastName);
    }

    public Contact() {
    }

    public Contact(PersonalName personalName, String phoneNumber, String address) {
        this.personalName = personalName;
        this.phoneNumber = phoneNumber;
        this.address = address;
    }

    @Override
    public boolean equals(Object obj) {
```

```
        // 省略實現代碼
    }

    @Override
    public int hashCode() {
        // 省略實現代碼
    }
}
```

---

從上述Java代碼可以看到，Contact類的屬性 **personalName** 的類型是 **PersonalName** 這個類 ( class )。我們選擇在 Contact 類中生成派生的 **protected** 屬性：**personalNameFirstName**、**personalNameLastName**。通過這樣的方式，可以把值對象 Contact 的屬性“壓平”，以便實現基於 SQL 數據庫的持久化。

然後，DDDMapper 工具生成的 Hibernate 映射文件就可以採用如下方式處理 **emergencyContact** 屬性了：

---

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-
3.0.dtd">
<hibernate-mapping
package="org.dddmapper.templates.tests.domain.person">
    <class name="AbstractPersonState$SimplePersonState"
table="People">
        <!-- 省略其他代碼-->
        <component name="emergencyContact"
class="org.dddmapper.templates.tests.domain.Contact">
            <property name="personalNameFirstName">
                <column
name="emergencyContactPersonalNameFirstName" length="50"/>
```

```
    </property>
    <property name="personalNameLastName">
        <column
name="emergencyContactPersonalNameLastName" length="50"/>
    </property>
    <property name="phoneNumber">
        <column name="emergencyContactPhoneNumber"/>
    </property>
    <property name="address">
        <column name="emergencyContactAddress"/>
    </property>
</component>
<!-- 省略其他代碼-->
</class>
</hibernate-mapping>
```

---

### 10.2.3 處理值對象的集合

在10.2.2節的DDML文檔示例中，聚合根Person的屬性Titles的類型是String的集合。

代碼生成工具為Person生成的狀態對象代碼如下（Java語言）：

---

```
public abstract class AbstractPersonState implements
PersonState.SqlPersonState, Saveable {
    private PersonId personId;
    public PersonId getPersonId() {
        return this.personId;
    }
    public void setPersonId(PersonId personId) {
        this.personId = personId;
    }

    private Set<String> titles;
    public Set<String> getTitles() {
        return this.titles;
    }
    public void setTitles(Set<String> titles) {
        this.titles = titles;
    }
    // 省略其他代碼
}
```

---

我們的工具會為Person實體的狀態對象生成如下Hibernate映射文件，可以留意一下對屬性titles的處理：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-
3.0.dtd">
<hibernate-mapping
package="org.dddml.templates.tests.domain.person">
    <class name="AbstractPersonState$SimplePersonState"
table="People">
        <composite-id name="personId"
            class="org.dddml.templates.tests.domain.PersonId">
                <key-property name="personalNameFirstName">
                    <column name="PersonIdPersonalNameFirstName"
length="50"/>
                </key-property>
                <key-property name="personalNameLastName">
                    <column name="PersonIdPersonalNameLastName"
length="50"/>
                </key-property>
                <key-property name="sequenceId">
                    <column name="PersonIdSequenceId"/>
                </key-property>
            </composite-id>
            <!-- 省略其他代碼-->
            <set name="titles" table="PersonTitles"
lazy="false">
                <!-- 這裡是屬性 titles 的映射信息 -->
                <key>
                    <column
name="PersonIdPersonalNameFirstName"/>
                    <column
name="PersonIdPersonalNameLastName"/>
                    <column name="PersonIdSequenceId"/>
                </key>
                <element column="TitlesItem" type="string" not-
null="true"/>
            </set>
            <!-- 省略其他代碼-->
        </class>
    </hibernate-mapping>
```

## 10.2.4 在URL中使用數據值對象

對於DDDDML文檔中定義的Person聚合，我們希望自動生成相應的RESTful API。比如，希望可以通過發送一個HTTP GET請求到以下URL來獲取一個Person實體的狀態：

---

```
{BASE_URL}/People/{personId}
```

---

Person實體的ID類型是值對象PersonId，那麼，如何把PersonId序列化為URL中使用的路徑字符串（也就是上面URL中的{personId}部分）呢？

誠然，可以選擇將PersonId序列化為JSON字符串，然後將其URL編碼之後用在URL中。

比如，一個PersonId的實例可以序列化為如下形式的JSON：

---

```
{
  "personalName": {
    "firstName": "Yang",
    "lastName": "Jiefeng"
  },
  "sequenceId": 1
}
```

---

將它URL編碼之後，得到的字符串結果如下：

---

```
%7B%0D%0A++++%22personalName%22%3A+%7B%0D%0A++++++%22first
Name%22%3A+%22Yang%22%2C%0D%0A++++++%22lastName%22%3A+%22J
iefeng%22%0D%0A++++%7D%2C%0D%0A++++
%22sequenceId%22%3A+1%0D%0A%7D
```

---

這未免太過煩瑣了，不利於在瀏覽器中輸入URL進行測試。所以，當需要將**PersonId**這樣的（嵌套數據值對象的）數據值對象序列化為一個簡單的字符串時，我們選擇的做法是：按照值對象的屬性在DDDML文檔中出現的先後順序，以深度優先的方式獲取屬性的值，並將其轉換成字符串，然後拼接成以特定的分隔符（默認是逗號“,”）分隔的字符串。也就是說，我們可以通過類似如下一個URL定位到一個Person的實例：

---

```
{BASE_URL}/People/Yang,Jiefeng,1
```

---

DDDML工具生成的**PersonId**值對象的代碼包含了屬性的遍歷邏輯（Java代碼）：

---

```
package org.dddml.templates.tests.domain;

import java.io.Serializable;
import org.dddml.templates.tests.domain.*;

public class PersonId implements Serializable {
    private PersonalName personalName = new PersonalName();
    // 省略 personalName 的 getter/setter 方法的代碼
    private Integer sequenceId;
    // 省略 sequenceId 的 getter/setter 方法的代碼
```

```
protected String getPersonalNameFirstName() {
    return getPersonalName().getFirstName();
}

protected void setPersonalNameFirstName(String
personalNameFirstName) {

getPersonalName().setFirstName(personalNameFirstName);
}

protected String getPersonalNameLastName() {
    return getPersonalName().getLastName();
}

protected void setPersonalNameLastName(String
personalNameLastName) {
    getPersonalName().setLastName(personalNameLastName);
}

public PersonId() {
}

public PersonId(PersonalName personalName, Integer
sequenceId) {
    this.personalName = personalName;
    this.sequenceId = sequenceId;
}

@Override
public boolean equals(Object obj) {
    // ...
}

@Override
public int hashCode() {
    // ...
}

protected static final String[] FLATTENED_PROPERTY_NAMES
= new String[]{
    "personalNameFirstName",
    "personalNameLastName",
    "sequenceId",
};
```

```
protected static final String[] FLATTENED_PROPERTY_TYPES
= new String[]{
    "String",
    "String",
    "Integer",
};

protected static final java.util.Map<String, String>
FLATTENED_PROPERTY_TYPE_MAP;

static {
    java.util.Map<String, String> m = new
java.util.HashMap<String, String>();
    for (int i = 0; i < FLATTENED_PROPERTY_NAMES.length;
i++) {
        m.put(FLATTENED_PROPERTY_NAMES[i],
FLATTENED_PROPERTY_TYPES[i]);
    }
    FLATTENED_PROPERTY_TYPE_MAP = m;
}

protected void
forEachFlattenedProperty(java.util.function.BiConsumer<String,
Object> consumer) {
    for (int i = 0; i < FLATTENED_PROPERTY_NAMES.length;
i++) {
        String pn = FLATTENED_PROPERTY_NAMES[i];
        Object pv = null;
        // 省略使用反射讀取屬性值的代碼
        consumer.accept(pn, pv);
    }
}

protected void setFlattenedPropertyValues(Object...
values) {
    for (int i = 0; i < FLATTENED_PROPERTY_NAMES.length;
i++) {
        String pn = FLATTENED_PROPERTY_NAMES[i];
        // 省略使用反射設置屬性值的代碼
    }
}
// ...
}
```

在需要將 **PersonId** 的實例序列化為一個簡單字符串的時候，我們只需要調用它的 **forEachFlattenedProperty** 方法，就可以遍歷它所有屬性的值，並將它們轉換為字符串（這個轉換可能需要使用某種“類型轉換工具類”），然後進行拼接。

如果需要把一個“簡單”的字符串反序列化為 **PersonId** 的實例，則可以新創建（`new`）一個 **PersonID** 實例，然後將字符串以分隔符進行分割，把分割出來的子字符串轉換成各個屬性的類型，然後調用 **setFlattenedPropertyValues** 方法設置這個 **PersonId** 的實例。

## 10.2.5 處理領域基礎類型

在筆者的實踐中，領域基礎類型都是在DDML文檔的/**typeDefinitions**結點下定義的。舉例來說，可以按如下形式定義一個名為**Money**的類型：

```
typeDefinitions:  
  Money:  
    javaType: org.joda.money.Money  
    cSharpType: MyMoneyLib.Money
```

這幾行代碼的含義如下：

- 在需要生成Java代碼的時候，把屬於領域概念的**Money**類型映射為Joda Money類庫中的**org.joda.money.Money**類。
- 在需要生成C#代碼的時候，把**Money**類型映射為在**MyMoneyLib**這個namespace下的**Money**類。

像Joda Time、Joda Money這樣的類庫，如果不是十分了解相關的領域知識，程序員是不可能在短時間內寫出來的。

構建這樣的領域專用基礎類型值對象對提高代碼質量可能有巨大幫助，雖然實現過程可能並不簡單，但有時候確實非常值得考慮。

## 1. 使用Hibernate持久化

如果打算使用Hibernate ORM框架來實現Joda Money對象的持久化，需要實現Money類對應的 Hibernate映射類型。假設把它命名為MoneyType，因為我們打算在數據庫中使用兩列來存儲Money對象，所以可以讓這個映射類型實現Hibernate提供的 CompositeUserType接口，它的實現代碼如下：

---

```
package org.dddml.wms.domain.hibernate.usertypes;

import org.hibernate.*;
import org.hibernate.engine.spi.SharedSessionContractImplementor;
import org.hibernate.type.*;
import org.hibernate.usertype.CompositeUserType;
import org.joda.money.*;
import java.io.Serializable;
import java.math.BigDecimal;
import java.sql.*;

public class MoneyType implements CompositeUserType {

    @Override
    public String[] getPropertyNames() {
        return new String[]{"amount", "currency"};
    }

    @Override
    public Type[] getPropertyTypes() {
        return new Type[]{BigDecimalType.INSTANCE,
StringType.INSTANCE};
    }

    @Override
    public Object getPropertyValue(Object component, int
propertyIndex) throws HibernateException {
        if (component == null) {

```

```

        return null;
    }
    final Money money = (Money) component;
    switch (propertyIndex) {
        case 0: {
            return money.getAmount();
        }
        case 1: {
            return
money.getCurrencyUnit().getCurrencyCode();
        }
        default: {
            throw new HibernateException("INVALID
property index: " + propertyIndex);
        }
    }
}

@Override
public void setPropertyValue(Object component, int
propertyIndex, Object value) throws HibernateException {
    throw new HibernateException("Component is
immutable.");
}

@Override
public Class returnedClass() {
    return Money.class;
}

@Override
public boolean equals(Object x, Object y) throws
HibernateException {
    if (x == null) {
        return y == null;
    }
    return x.equals(y);
}

@Override
public int hashCode(Object x) throws HibernateException
{
    assert (x != null);
    return x.hashCode();
}

```

```

    @Override
    public Object nullSafeGet(ResultSet rs, String[] names,
SharedSessionContractImplementor
sharedSessionContractImplementor, Object owner) throws
HibernateException, SQLException {
        assert names.length == 2;
        BigDecimal amount =
BigDecimalType.INSTANCE.nullSafeGet(rs, names[0],
sharedSessionContractImplementor);
        String currencyCode =
StringType.INSTANCE.nullSafeGet(rs, names[1],
sharedSessionContractImplementor);
        return amount == null && currencyCode == null
            ? null
            :
        BigMoney.of(CurrencyUnit.getInstance(currencyCode),
amount).toMoney();
    }

    @Override
    public void nullSafeSet(PreparedStatement st, Object
value, int index,
SharedSessionContractImplementor
sharedSessionContractImplementor) throws HibernateException,
SQLException {
        if (value == null) {
            BigDecimalType.INSTANCE.set(st, null, index,
sharedSessionContractImplementor);
            StringType.INSTANCE.set(st, null, index + 1,
sharedSessionContractImplementor);
        } else {
            final Money money = (Money) value;
            BigDecimalType.INSTANCE.set(st,
money.getAmount(), index,
sharedSessionContractImplementor);
            StringType.INSTANCE.set(st,
money.getCurrencyUnit().getCurrencyCode(), index + 1,
sharedSessionContractImplementor);
        }
    }

    @Override
    public Object deepCopy(Object value) throws
HibernateException {

```

```
        if (value == null)
            return null;
        Money money = (Money) value;
        return Money.of(money);
    }

    @Override
    public boolean isMutable() {
        return false;
    }

    @Override
    public Serializable disassemble(Object value,
SharedSessionContractImplementor
sharedSessionContractImplementor) throws HibernateException
{
    return (Serializable) value;
}

    @Override
    public Object assemble(Serializable cached,
SharedSessionContractImplementor
sharedSessionContractImplementor, Object owner) throws
HibernateException {
    return cached;
}

    @Override
    public Object replace(Object original, Object target,
SharedSessionContractImplementor
sharedSessionContractImplementor, Object owner) throws
HibernateException {
    return deepCopy(original);
}
}
```

---

為了生成**Hibernate**映射代碼，可能還需要在限界上下文的**DDML**配置文件中給代碼生成工具提供更多的信息：

---

```
configuration:  
    # 省略部分代碼 ...  
    hibernate:  
        hibernateTypes:  
            Money:  
                mappingType:  
                    "org.dddml.wms.domain.hibernate.usertypes.MoneyType"  
                    propertyNames: ["Amount", "Currency"]  
                    propertyTypes: ["decimal", "string"]  
  
            nHibernate:  
                nHibernateTypes:  
                    Money:  
                        mappingType:  
                            "Dddml.Wms.Services.Domain.NHibernate.MyMoneyType,  
                            Dddml.Wms.Services"  
                            propertyNames: ["Amount", "Currency"]  
                            propertyTypes: ["decimal", "string"]
```

---

作為Hibernate的.NET移植版本，在NHibernate中也存在類似CompositeUserType的ICompositeUserType接口，NHibernate XML映射文件的寫法也與Hibernate大同小異，在此不再贅述。

## 2.序列化

### (1) 使用Jackson JSON庫

如果想要使用Jackson JSON庫來實現Money類的序列化和反序列化，首先需要實現Money的JsonSerializer，示例如下：

---

```
package org.dddml.wms.restful.json;
```

```
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.joda.money.Money;
import java.io.IOException;

public class JodaMoneyJacksonSerializer extends
JsonSerializer<Money> {
    @Override
    public void serialize(Money value, JsonGenerator gen,
SerializerProvider serializers) throws IOException,
JsonProcessingException {
        gen.writeStartObject();
        gen.writeStringField("amount",
value.getAmount().toString());
        gen.writeStringField("currency",
value.getCurrencyUnit().getCurrencyCode());
        gen.writeEndObject();
    }
}
```

---

為了支持Money的反序列化，還需要實現一個  
JsonDeserializer：

---

```
package org.dddml.wms.restful.json;

import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.joda.money.*;
import java.io.IOException;
import java.math.BigDecimal;

public class JodaMoneyJacksonDeserializer extends
JsonDeserializer<Money> {
    @Override
    public Money deserialize(JsonParser p,
DeserializationContext ctxt) throws IOException {
        try {
            JsonNode node = p.getCodec().readTree(p);
            String amount = node.get("amount").asText();
            String currency = node.get("currency").asText();
            Money money =
```

```
        Money.of(CurrencyUnit.of(currency), new BigDecimal(amount));
        return money;
    } catch (Exception ex) {
        throw new JsonParseException(p,
ex.getMessage());
    }
}
```

---

如果你在**Spring MVC**開發的應用中使用了**Jackson JSON**庫來序列化/反序列化**Money**對象，那麼，可能需要客製化**Spring MVC**應用所使用的**ObjectMapper**組件，讓**Money**的序列化器與反序列化器生效。通過**Google**搜索關鍵字“**Spring MVC Customize the Jackson ObjectMapper**”可以找到設置它的方法，這裡不再贅述。

## ( 2 ) 使用**Json.NET**庫

如果想要使用**Json.NET**庫來實現**MyMoneyLib.Money**對象的序列化和反序列化，可能需要先實現一個名為**MoneyJsonConverter**的**JsonConverter**。然後，編寫一個**CustomContractResolver**類（C#代碼），示例如下：

---

```
public class CustomContractResolver :
DefaultContractResolver
{
    private static readonly Type _moneyType = typeof(Money);

    private static readonly JsonConverter
    _moneyJsonConverter = new MoneyJsonConverter();
```

```
protected override JsonConverter
ResolveContractConverter(Type objectType)
{
    if (objectType != null &&
    _moneyType.IsAssignableFrom(objectType))
    {
        return _moneyJsonConverter;
    }
    return base.ResolveContractConverter(objectType);
}
```

---

如果我們想要在ASP.NET Web API應用中使用它們，可能需要以如下方式設置應用所使用的**System.Web.Http.HttpConfiguration**配置對象（如果是使用Visual Studio的ASP.NET Web API模板生成的項目，可能可以在名為“WebApiConfig.cs”的C#代碼文件中找到設置HttpConfiguration的地方）：

---

```
config.Formatters.JsonFormatter.SerializerSettings.ContractResolver = new CustomContractResolver();
```

---

### 3.處理時間值對象

在對領域進行建模的時候，有必要分辨領域中那些看上去相似、互相有聯繫但是存在微妙區別的概念。概念混淆的反面典型之一是Java 8之前的時間處理API。以下是後來Java 8在java.time包下引入的部分類（它們借鑑自Joda Time類庫）：

·**LocalDateTime**，表示“當地日期時間”，它不能表示“時刻”，但是可以通過調用方法，指定時區（“本地”是哪裡）後得到一個“時刻”。

·**LocalDate**，表示當地日期。時區不明確。

·**LocalTime**，表示當地時間。時區不明確。

·**ZonedDateTime**，有時區的日期時間，它可以表示時刻。

這些時間相關的類都可以認為是值對象。當你開發的應用所服務的領域需要使用時間的值對象時，建議考慮一下是否可以使用上面的名詞與概念。另外，也許還可以考慮使用以下概念：

·**ZonedDateTime**，帶時區的日期，不是一個時刻。表示比如“北京時間2020年1月30號”這樣的概念，但是沒有明確是這一天中的哪個時刻。

·**ZonedDateTime**，帶時區的時間，不是一個時刻。表示比如“北京時間早上8點”這樣的概念。

如果你想要在一個界限上下文中把值對象命名為**DateTime**或者**Timestamp**，那應該儘可能地澄清它們的確切含義：

·對於**DateTime**，建議明確說明它是**LocalDateTime**還是**ZonedDateTime**的別名。如果你想

使用**Date**這個名字，也請做類似的考慮。

·對於**Timestamp**，我們經常看到“時間戳”這個詞，它的含義往往取決於上下文。有時它可能是指“時刻”，有時可能是指一個自動增長的整數序號。不同的**SQL**數據庫中**Timestamp**類型所指的概念可能存在明顯差異。請認真考慮是否有必要在限界上下文中使用**Timestamp**作為值對象的名稱，如果確有需要，對其做出準確的定義。

決定將這些與時間相關的領域對象映射為特定語言（**Java**、**C#**、**PHP**等）的何種類型，要從它們代表的領域概念出發。在**JSON**序列化/反序列化這些對象的過程中，首先應該注意不能丢失信息，還應該考慮序列化結果的可讀性。當然，也建議不要“自作多情”地往序列化結果中添加值對象在領域概念中不存在的信息。比如，把時區不明確的**LocalDate**序列化為**JSON**字符串，在序列化結果中添加“時區指示”信息等。

關於時間對象的持久化問題，相信不少讀者都曾為此大傷腦筋。更具體一點說，假如要使用**Hibernate**與**MySQL**開發應用，需要存儲**java.time.ZonedDateTime**類型的值對象，應該如何處理？

如果我們儘可能不做設置（即使用默認設置），那麼**Hibernate**的行為大致是：

· Java 類型 `java.time.ZonedDateTime` 的 Hibernate 映射類型為 `org.hibernate.type.ZonedDateTimeType`。如果使用 MySQL，其對應的列類型默認為 MySQL 的 `DateTime` 類型。

· 不管是什麼時區的 `ZonedDateTime` 實例，在寫入數據庫時都會被統一轉為系統默認時區的日期時間值後存入數據庫。MySQL 的 `DateTime` 類型不包含時區信息，可以將其想象成是一個“沒有時區的日期時間”字符串。

· 當需要從數據庫中取回 `ZonedDateTime` 時，只能先獲得一個“沒有時區的日期時間”值（在概念上可以理解為 `LocalDateTime` 類型），然後指定時區為系統默認時區，並將其轉換為 `ZonedDateTime`。

在這種情況下，如果應用需要在不同時區的機器上運行，很容易引發混亂。

所以，我們可以考慮的一個處理方案是，告訴 Hibernate 總是使用 UTC 時區：

---

```
hibernate.jdbc.time_zone=UTC
```

---

但是這還不夠，我們還需要阻止 MySQL JDBC Connector“自作聰明”。讀者可自行通過 Google 搜索 `How to store date, time, and timestamps in UTC time`

zone with JDBC and Hibernate這篇文章<sup>[1]</sup>，參考文章中的方法進行處理。其中的關鍵點是：

If you're using the MySQL JDBC Connector/J prior to version 8, you need to set the `useLegacyDatetimeCode` connection property to `false` as, otherwise, the `hibernate.jdbc.time_zone` has no effect.

也就是說，如果你使用的是版本8之前的MySQL JDBC Connector/J，需要把“`useLegacyDatetimeCode`”這個連接屬性設置為`false`，否則設置“`hibernate.jdbc.time_zone`”不會生效。

為了更保險，我們可以進一步指定 `serverTimezone` 連接屬性，給應用配置的數據源的 JDBC URL 如下：

---

```
jdbc:mysql://localhost/test?
characterEncoding=utf8&serverTimezone=
GMT%2b0&useLegacyDatetimeCode=false
```

---

[1] 見 <https://vladmirhalcea.com/how-to-store-date-time-and-timestamps-in-utc-time-zone-with-jdbc-and-hibernate/>。

## 第11章 處理聚合與實體

有了值對象，包括數據值對象和領域基礎類型，就有了實體的構造塊。實體總是屬於某個聚合的，因此需要在DDDML文檔的/aggregates結點下定義聚合。

聚合是DDD戰術層面最重要的概念，如何生成聚合的代碼是DDDML代碼生成工具的核心。默認情況下，筆者製作的工具會為聚合生成的代碼使用事件溯源模式。

DDDML工具為聚合生成的代碼大部分都不應該依賴於特定的外部框架。比如說，依賴於Hibernate與Spring框架的代碼都應該被剝離到特定的子類或接口的特定實現類中。

但是我們確實有一些應該在不同的聚合之間共享的代碼，包括一些接口、基類以及工具類。筆者的做法是，將這樣的代碼部分放在限界上下文的“Specialization Namespace”中，這個Namespace是上下文中“基礎Namespace”的一個子空間。



提示映射到具體的語言，這裡說的Namespace是指Java的關鍵字package所代表的概念；對於C#來說，這個Namespace就是C#的關鍵字namespace代表的概念。

我們希望工具生成的代碼無須修改就可以通過編譯，並且可以工作起來。生成的代碼可以分成兩部分：

- 一部分代碼在修改領域模型的**DDDML**文檔之後需要被重新生成、覆蓋，這部分代碼應該預留足夠的擴展點，我們不應該手動去修改它們。對於**Java**來說，擴展方式之一是使用**extends**工具生成的基類；對於**C#**來說，可以使用**partial class**作為生成代碼的擴展方式。

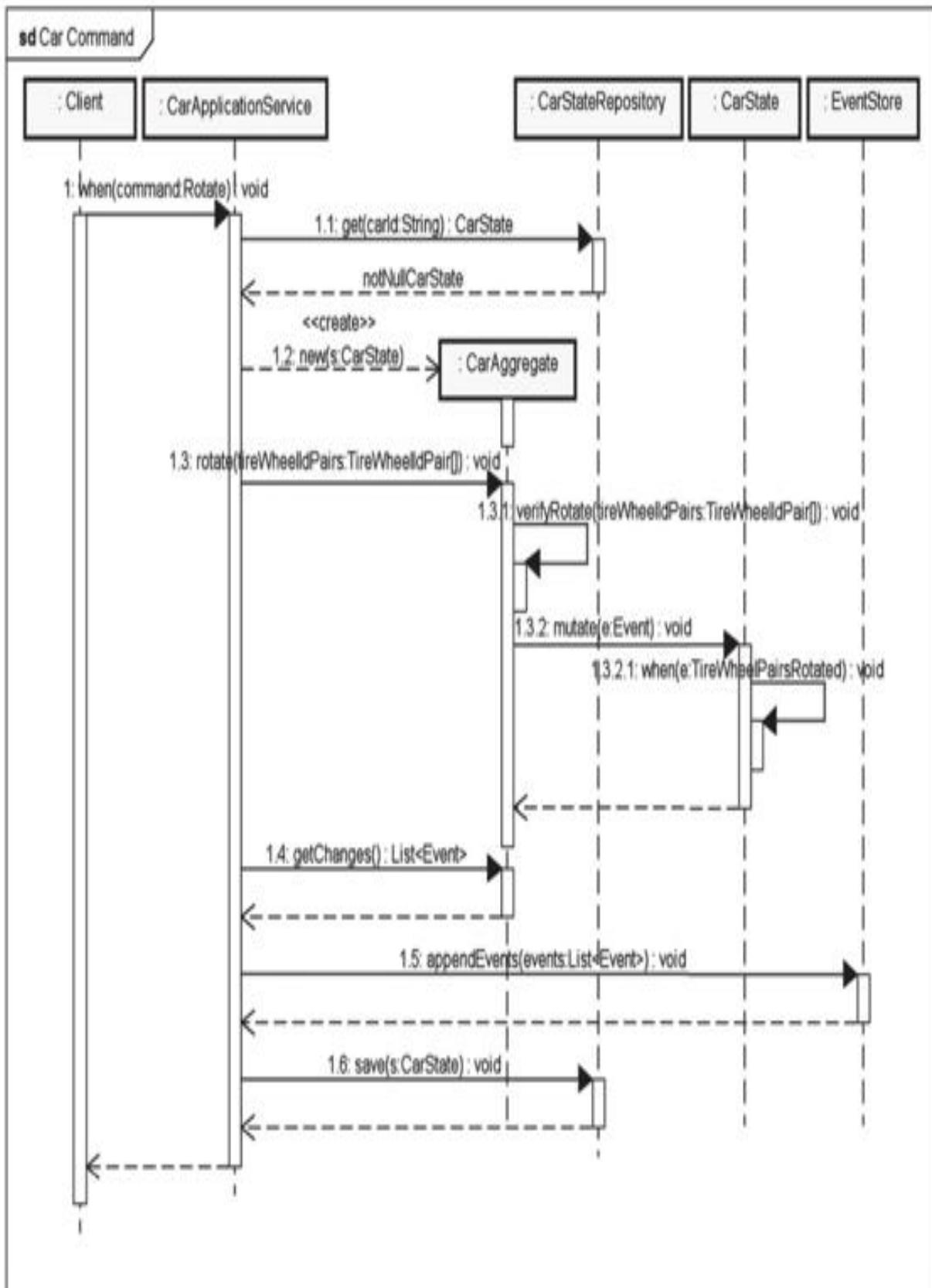
- 另外一部分代碼可以看作是腳手架代碼，在生成之後不會被代碼生成工具靜默地覆蓋。開發人員可以根據領域的需要修改（特別化）它們。我們在**Specialization Namespace**內生成的代碼就屬於這種情況。

對於在**DDDML**中定義的存在繼承關係的實體，本章會演示在使用**Hibernate ORM**框架的情況下該如何處理這些繼承關係。

本章還會介紹代碼生成工具可以如何利用**DDDML**文檔中的模式信息，以第9章中展示過的庫存單元聚合中的那些在庫數量、在途數量之類的賬目，以及入庫/出庫單聚合中的單據狀態的狀態機為例，講解工具能給開發人員的編碼工作提供什麼幫助。

## 11.1 生成聚合的代碼

下面以前文使用過的DDDML文檔中定義的Car聚合為例進行說明，筆者製作的DDDML工具生成的Rotate方法（命令）的默認執行邏輯大致如圖11-1所示（使用UML順序圖表示）。



## 圖11-1 Car聚合Rotate方法的執行順序圖

在默認的情況下，生成的代碼使用了事件溯源模式。要想改變Car的狀態，必須先生成一個事件（Event），然後調用CarState的mutate方法。

可能有讀者已經注意到了，雖然在DDDML中定義了一個名為Car實體，但是在上面的順序圖中並不存在一個名為Car的對象，卻存在一個叫CarAggregate的對象。我們把像CarAggregate這樣的對象叫作聚合對象，把CarState這樣的對象叫作狀態對象。Car聚合的業務邏輯由CarAggregate實現，也就是說它負責維護Car聚合內所有實體狀態的一致性。

需要說明的是，圖11-1其實是概念性的。具體來說，我們使用的DDDML工具為這個聚合生成的Java代碼中會包括一些對象的接口（interface）以及這些接口的實現類，在圖11-1中並沒有體現這些代碼細節。接下來，看看工具生成的代碼中具體都包含些什麼。

## 11.1.1 接口

### 1. 狀態

工具生成的表示Car的狀態接口 ( Java語言 ) 如下：

---

```
package org.dddml.templates.tests.domain.car;

import java.util.*;
import org.dddml.templates.tests.domain.*;
import org.dddml.templates.tests.specialization.Event;

public interface CarState {
    Long VERSION_ZERO = 0L;
    Long VERSION_NULL = VERSION_ZERO - 1;

    String getId();
    String getDescription();
    Long getVersion();
    String getCreatedBy();
    Date getCreatedAt();
    String getUpdatedBy();
    Date getUpdatedAt();
    Boolean getActive();
    Boolean getDeleted();

    EntityStateCollection<String, WheelState> getWheels();
    EntityStateCollection<String, TireState> getTires();

    interface MutableCarState extends CarState {
        void setId(String id);
        void setDescription(String description);
        void setVersion(Long version);
        void setCreatedBy(String createdBy);
        void setCreatedAt(Date createdAt);
        void setUpdatedBy(String updatedBy);
        void setUpdatedAt(Date updatedAt);
    }
}
```

```
        void setActive(Boolean active);
        void setDeleted(Boolean deleted);

        // 注意，這個 mutate 方法會修改 Car 的狀態：
        void mutate(Event e);
    }

    interface SqlCarState extends MutableCarState {
        boolean isStateUnsaved();
        boolean getForReapplying();
    }
}
```

---

在這裡，`org.dddml.templates.tests`是測試限界上下文對應的Java包（ package ）。

Car實體的狀態接口`CarState`的屬性都是隻讀的，我們把修改屬性的那些`setter`方法都放在它的子類型`MutableCarState`接口中。

聚合內可能有多個實體，所謂“聚合的狀態”自然需要包括聚合根的狀態以及聚合內部其他（非聚合根）實體的狀態。

工具生成的接口`EntityStateCollection`表示聚合內部實體的狀態對象的集合，接口的代碼如下：

---

```
package org.dddml.templates.tests.domain;

import java.util.Collection;

public interface EntityStateCollection<TId, TState> extends Collection<TState> {
    TState get(TId entityId);
    boolean isLazy();
```

```
    boolean isAllLoaded();
    Collection<TState> getLoadedStates();

    interface ModifiableEntityStateCollection<TId, TState>
    extends EntityStateCollection<TId, TState> {
        Collection<TState> getRemovedStates();
        TState getOrAdd(TId entityId);
    }
}
```

---

接口**EntityStateCollection**的**get(TId entityId)**方法用於根據實體的ID（這裡指的是聚合內部實體的**Local ID**）返回實體的狀態，如果該ID對應的實體實例不存在，則返回**null**。

接口**ModifiableEntityStateCollection**的方法**getOrAdd(TId entityId)**則一定會返回非**null**的實體狀態對象，如果從參數傳入的**ID**尚未存在對應的實體實例，那麼需要新建一個實體的狀態對象添加到集合中，然後返回該狀態對象。

接口**EntityStateCollection**的實現有可能使用懶（Lazy）加載的方式來加載實體的狀態。已經加載的實體狀態則可以通過**getLoadedStates()**方法獲取。

再來看看聚合內部實體**Wheel**的狀態對象接口（文件**WheelState.java**）：

---

```
public interface WheelState {
    Long VERSION_ZERO = 0L;
    Long VERSION_NULL = VERSION_ZERO - 1;
```

```

String getWheelId();
Long getVersion();
String getCreatedBy();
// 省略部分代碼
String getCarId();

interface MutableWheelState extends WheelState {
    void setWheelId(String wheelId);
    void setVersion(Long version);
    void setCreatedBy(String createdBy);
    // 省略部分代碼
    void setCarId(String carId);

    void mutate(Event e);
}

interface SqlWheelState extends MutableWheelState {
    CarWheelId getCarWheelId();
    void setCarWheelId(CarWheelId carWheelId);
    boolean isStateUnsaved();
    boolean getForReapplying();
}
}

```

---

可以看到，**WheelState**接口的子類型 **SqlWheelState**接口有個**carWheelId**屬性，它表示 **Wheel**實體的**Global ID**。工具為實體**Wheel**生成了它的 **Global ID**值對象（Java代碼，有刪節）：

---

```

public class CarWheelId implements java.io.Serializable {
    private String carId;
    public String getCarId() {
        return this.carId;
    }
    public void setCarId(String carId) {
        this.carId = carId;
    }
}

private String wheelId;

```

```
public String getWheelId() {
    return this.wheelId;
}
public void setWheelId(String wheelId) {
    this.wheelId = wheelId;
}

public CarWheelId() {
}
public CarWheelId(String carId, String wheelId) {
    this.carId = carId;
    this.wheelId = wheelId;
}

@Override
public boolean equals(Object obj) {
    // ...
}
@Override
public int hashCode() {
    // ...
}
// ...
}
```

---

## 2. 命令

代碼生成工具會生成**Car**的命令對象的基類接口（**CarCommand**）。在默認情況下，生成的**Java**代碼還包括創建**Car**（**CreateCar**）、更新**Car**（**MergePatchCar**）、刪除**Car**（**DeleteCar**）的命令對象的接口（**Java**代碼）：

---

```
public interface CarCommand extends Command {
    String getId();
    void setId(String id);
    Long getVersion();
    void setVersion(Long version);
```

```

        static void throwOnInvalidStateTransition(CarState
state, Command c) {
            if (state.getVersion() == null) {
                if (isCommandCreate((CarCommand) c)) {
                    return;
                }
                throw DomainError.named("premature",
"Can't do anything to unexistent aggregate");
            }
            if (state.getDeleted() != null &&
state.getDeleted()) {
                throw DomainError.named("zombie",
"Can't do anything to deleted aggregate.");
            }
            if (isCommandCreate((CarCommand) c))
                throw DomainError.named("rebirth",
"Can't create aggregate that already exists");
        }

        static boolean isCommandCreate(CarCommand c) {
            return c.getVersion().equals(CarState.VERSION_NULL);
        }

        interface CreateOrMergePatchCar extends CarCommand {
            String getDescription();
            void setDescription(String description);
            Boolean getActive();
            void setActive(Boolean active);
        }

        interface CreateCar extends CreateOrMergePatchCar {
            CreateWheelCommandCollection
getCreateWheelCommands();
            WheelCommand.CreateWheel newCreateWheel();
            CreateTireCommandCollection getCreateTireCommands();
            TireCommand.CreateTire newCreateTire();
        }

        interface MergePatchCar extends CreateOrMergePatchCar {
            Boolean getIsPropertyDescriptionRemoved();
            void setIsPropertyDescriptionRemoved(Boolean
removed);
            Boolean getIsPropertyActiveRemoved();
            void setIsPropertyActiveRemoved(Boolean removed);
        }
    }
}

```

```
        WheelCommandCollection getWheelCommands();
        WheelCommand.CreateWheel newCreateWheel();
        WheelCommand.MergePatchWheel newMergePatchWheel();
        WheelCommand.RemoveWheel newRemoveWheel();
        TireCommandCollection getTireCommands();
        TireCommand.CreateTire newCreateTire();
        TireCommand.MergePatchTire newMergePatchTire();
        TireCommand.RemoveTire newRemoveTire();
    }

    interface DeleteCar extends CarCommand {
    }

    interface CreateWheelCommandCollection extends
    Iterable<WheelCommand.CreateWheel> {
        void add(WheelCommand.CreateWheel c);
        void remove(WheelCommand.CreateWheel c);
        void clear();
    }

    interface WheelCommandCollection extends
    Iterable<WheelCommand> {
        void add(WheelCommand c);
        void remove(WheelCommand c);
        void clear();
    }

    interface CreateTireCommandCollection extends
    Iterable<TireCommand.CreateTire> {
        void add(TireCommand.CreateTire c);
        void remove(TireCommand.CreateTire c);
        void clear();
    }

    interface TireCommandCollection extends
    Iterable<TireCommand> {
        void add(TireCommand c);
        void remove(TireCommand c);
        void clear();
    }
}
```

---

這裡生成的用於更新 ( Update ) Car的命令對象的接口名叫MergePatchCar，這是因為服務端在收到MergePatchCar命令後，它的處理邏輯有點類似於JSON Merge Patch<sup>[1]</sup>，但是又有所不同。

按照JSON Merge Patch規範，在客戶端傳過來的JSON實體（對象）中：

- 如果某個屬性（名字）的值被顯式地設置為null，表示需要將該屬性從資源中移除。
- 如果不存在某個屬性（名字），則資源的這個屬性（名字）保留原值。

也就是說，可以認為JSON的對象是動態類型。和JSON Merge Patch不同，在這裡會使用靜態類型的命令對象。對於靜態類型來說，在代碼中定義的屬性總是存在的，無法被移除，最多隻能把它的值設置為null。很多時候，靜態類型的屬性默認值本來就是null。我們希望能很方便地使用“常用”的JSON序列化庫對靜態類型進行序列化/反序列化，並且在序列化/反序列化的過程中不丟失信息。當我們使用JSON序列化庫對這些命令對象進行序列化的時候，如果不針對某個類型進行設置，那麼序列化庫的默認處理方式一般是：值為null的屬性要麼全部被序列化到JSON，要麼全部被忽略。

問題是，工具生成的靜態類型的命令對象 MergePatch{ENTITY\_NAME}需要分別表示：

- {ENTITY\_NAME} State的某個屬性要被移除——對於靜態類型的狀態對象來說，移除屬性也就是將屬性設置為null。
- 讓{ENTITY\_NAME} State的某個屬性保留原值，不做修改。

一個可選的解決方法是給 MergePatch{ENTITY\_NAME}命令對象添加像 isProperty {PROPERTY\_NAME}Removed這樣的屬性。可以根據靜態類型的命令對象的屬性 isProperty{PROPERTY\_NAME}Removed的值來確定是否需要將對應的實體的狀態對象的 {PROPERTY\_NAME}屬性設置為null。

以MergePatchCar為例，當它的 isPropertyDescriptionRemoved為true時，表示要將 CarState的description屬性修改為null；否則，只有當 MergePatchCar的description屬性的值不為null時，才使用這個屬性的值去更新CarState的description。

對於DDDDML中定義的Car實體的Rotate方法，工具生成了對應的命令對象，作為CarCommands類的一個內部類（Java代碼），示例如下：

---

```

public class CarCommands {
    private CarCommands() {
    }

    public static class Rotate extends AbstractCommand
    implements CarCommand {
        public String getCommandType() {
            return "Rotate";
        }
        public void setCommandType(String commandType) {
        }

        /**
         * Car Id.
         */
        private String id;
        public String getId() {
            return this.id;
        }
        public void setId(String id) {
            this.id = id;
        }

        private TireWheelIdPair[] tireWheelIdPairs;
        public TireWheelIdPair[] getTireWheelIdPairs() {
            return this.tireWheelIdPairs;
        }
        public void setTireWheelIdPairs(TireWheelIdPair[]
tireWheelIdPairs) {
            this.tireWheelIdPairs = tireWheelIdPairs;
        }

        private Long version;
        public Long getVersion() { return this.version; }
        public void setVersion(Long version) { this.version
= version; }
    }
}

```

---

默認情況下，工具也會生成聚合內部實體（比如 **Wheel**）的命令對象的接口，這些非聚合根實體的命

令對象不能獨立使用，應該被理解為“聚合的命令對象”的一部分（Java代碼），示例如下：

---

```
public interface WheelCommand extends Command {  
    String getWheelId();  
    void setWheelId(String wheelId);  
  
    interface CreateOrMergePatchWheel extends WheelCommand {  
        Boolean getActive();  
        void setActive(Boolean active);  
    }  
  
    interface CreateWheel extends CreateOrMergePatchWheel {  
    }  
  
    interface MergePatchWheel extends  
CreateOrMergePatchWheel {  
        Boolean getIsPropertyActiveRemoved();  
        void setIsPropertyActiveRemoved(Boolean removed);  
    }  
  
    interface RemoveWheel extends WheelCommand {  
    }  
}
```

---

### 3.事件

工具生成的表示Car的事件對象的接口（Java代碼）如下：

---

```
import org.dddml.templates.tests.specialization.Event;  
  
public interface CarEvent extends Event {  
    interface SqlCarEvent extends CarEvent {  
        CarEventId getCarEventId();  
        boolean getEventReadOnly();  
        void setEventReadOnly(boolean readOnly);  
    }  
}
```

```

    }

    String getId();
    Long getVersion();
    String getCreatedBy();
    void setCreatedBy(String createdBy);
    Date getCreatedAt();
    void setCreatedAt(Date createdAt);
    String getCommandId();
    void setCommandId(String commandId);

    interface CarStateEvent extends CarEvent {
        String getDescription();
        void setDescription(String description);
        Boolean getActive();
        void setActive(Boolean active);
    }

    interface CarStateCreated extends CarStateEvent {
        Iterable<WheelEvent.WheelStateCreated>
getWheelEvents();
        void addWheelEvent(WheelEvent.WheelStateCreated e);
        WheelEvent.WheelStateCreated
newWheelStateCreated(String wheelId);
        Iterable<TireEvent.TireStateCreated>
getTireEvents();
        void addTireEvent(TireEvent.TireStateCreated e);
        TireEvent.TireStateCreated
newTireStateCreated(String tireId);
    }

    interface CarStateMergePatched extends CarStateEvent {
        Boolean getIsPropertyDescriptionRemoved();
        void setIsPropertyDescriptionRemoved(Boolean
removed);
        Boolean getIsPropertyActiveRemoved();
        void setIsPropertyActiveRemoved(Boolean removed);
        Iterable<WheelEvent> getWheelEvents();
        void addWheelEvent(WheelEvent e);
        WheelEvent.WheelStateCreated
newWheelStateCreated(String wheelId);
        WheelEvent.WheelStateMergePatched
newWheelStateMergePatched(String wheelId);
        WheelEvent.WheelStateRemoved
newWheelStateRemoved(String wheelId);
    }
}

```

```

        Iterable<TireEvent> getTireEvents();
        void addTireEvent(TireEvent e);
        TireEvent.TireStateCreated
newTireStateCreated(String tireId);
        TireEvent.TireStateMergePatched
newTireStateMergePatched(String tireId);
        TireEvent.TireStateRemoved
newTireStateRemoved(String tireId);
    }

    interface CarStateDeleted extends CarStateEvent {
        Iterable<WheelEvent.WheelStateRemoved>
getWheelEvents();
        void addWheelEvent(WheelEvent.WheelStateRemoved e);
        WheelEvent.WheelStateRemoved
newWheelStateRemoved(String wheelId);
        Iterable<TireEvent.TireStateRemoved>
getTireEvents();
        void addTireEvent(TireEvent.TireStateRemoved e);
        TireEvent.TireStateRemoved
newTireStateRemoved(String tireId);
    }
}

```

---

我們可以看到，對於DDDML中定義的實體，假設這個實體的名稱為{Xxxx}，默認情況下，工具會生成名稱為{Xxxx}StateEvent的事件對象的接口，以及它的子類型{Xxxx}StateCreated、{Xxxx}StateMergePatched、{Xxxx}StateDeleted。一般來說，這些事件類型表示實體的狀態因為執行Create{Xxxx}、MergePatch{Xxxx}、Delete{Xxxx}命令而發生的變化。但是，如前文所述，事件與命令並非一定要存在嚴格的一一對應關係，所以其他命令的執行也有可能產生這些事件。一個實體的事件類型是可以擴展的，但是如果一個命令產生的後果可以用{Xxxx}StateCreated、{Xxxx}StateMergePatched來表

示，那麼直接使用它們也無妨，在後文會看到這樣的例子。

默認情況下，工具生成的代碼會為每個實體創建對應的事件表，實體的事件會存儲到各自的事件表中。工具會為事件表的主鍵生成對應的值對象的代碼，比如對於**Car**聚合根，工具會生成**CarEventId**：

---

```
public class CarEventId implements java.io.Serializable {
    private String id; //Car Id.
    public String getId() { return this.id; }
    public void setId(String id) { this.id = id; }

    private Long version;
    public Long getVersion() { return this.version; }
    public void setVersion(Long version) { this.version =
version; }

    public CarEventId() {
    }
    public CarEventId(String id, Long version) {
        this.id = id;
        this.version = version;
    }

    @Override
    public boolean equals(Object obj) {
        // ...
    }
    @Override
    public int hashCode() {
        // ...
    }
}
```

---

工具也生成了**WheelEvent**接口，**WheelEvent**是非聚合根實體的事件對象，應該被理解為**CarEvent**的一部分（Java代碼），示例如下：

---

```
public interface WheelEvent extends Event {  
    interface SqlWheelEvent extends WheelEvent {  
        WheelEventId getWheelEventId();  
        boolean getEventReadOnly();  
        void setEventReadOnly(boolean readOnly);  
    }  
  
    String getWheelId();  
    String getCreatedBy();  
    // 省略代碼  
    interface WheelStateEvent extends WheelEvent {  
        // ...  
    }  
    interface WheelStateCreated extends WheelStateEvent {  
        // ...  
    }  
    interface WheelStateMergePatched extends WheelStateEvent {  
        // ...  
    }  
    interface WheelStateRemoved extends WheelStateEvent {  
        // ...  
    }  
}
```

---

為了方便地將**WheelEvent**存儲到SQL數據庫中，工具會生成**WheelEventId**數據值對象（Java代碼），示例如下：

---

```
public class WheelEventId implements Serializable {  
    private String carId;  
    public String getCarId() { return this.carId; }  
    public void setCarId(String carId) { this.carId = carId; }  
}
```

```
private String wheelId;
public String getWheelId() { return this.wheelId; }
public void setWheelId(String wheelId) { this.wheelId =
wheelId; }

private Long carVersion;
public Long getCarVersion() {
    return this.carVersion;
}
public void setCarVersion(Long carVersion) {
    this.carVersion = carVersion;
}

public WheelEventId() {
}
public WheelEventId(String carId, String wheelId, Long
carVersion) {
    this.carId = carId;
    this.wheelId = wheelId;
    this.carVersion = carVersion;
}

@Override
public boolean equals(Object obj) {
    // ...
}
@Override
public int hashCode() {
    // ...
}
}
```

---

如果成功執行實體Car的命令方法Rotate，會產生類型為TireWheelPairsRotated的事件，這是一個Java類（不是接口），它的代碼在本節的後面展示。

在生成的代碼中，那些看著很相似的狀態、命令、事件對象之間是沒有繼承關係，也沒有共同的基

類的。比如CarState、CreateCar、CarStateCreated，從表面上看有很多名字一樣的屬性，但是它們之間沒有繼承關係，也沒有共同的基類。

其實，筆者製作的代碼生成工具在剛開始生成狀態、命令、事件時，它們之間是有繼承關係的，但是筆者很快就發現這是個錯誤，因為它們在概念上是不同的東西，這在前文已經做過闡述。比如，對於使用了賬務模式的聚合來說，賬目的狀態對象（AccountState）中可能存在一個餘額屬性，但是在創建賬目（CreateAccount）這個命令對象中絕不應該存在同名的屬性。如果讓CreateAccount繼承自AccountState是個再明顯不過的錯誤，它打破了賬務模式“用心良苦”的封裝，即使是讓CreateAccount命令對象重寫（override）餘額屬性的setter方法，在方法中拋出異常，這也不是一個好做法。

再比如，我們的工具生成C#代碼時，根據需要，某個名稱為{Xxx}的實體，其狀態對象{Xxx}State中可能存在一個絕對不能為空的值類型屬性Foo。而它的事件對象{Xxx}StateCreated的Foo屬性，可能需要的是一個可空的值類型的包裝類型（這個包裝類型是個引用類型）。這兩個不同對象的Foo屬性的類型並不相同。當一個{Xxx}StateCreated對象的Foo的值為null時，表示創建該實體時沒有顯式地設置Foo屬性的值——所以在{Xxx}StateCreated事件發生後，對應的{Xxx}State的屬性Foo應該會保留在構造對象時獲得的初始值。



提示這裡提到的“值類型”“引用類型”是.NET (CLR) 中的概念，和DDD的“值對象”和“引用對象”概念不同。其實在Java (JVM) 中也存在值類型和引用類型的概念，像int、long這樣的基本類型 (Primitive Types) 都是值類型，它們對應的包裝類Integer、Long則是引用類型。也許有讀者已經注意到，在本書展示的Java示例代碼中，領域值對象都沒有映射為Java的基本類型。

## 4. Aggregate

工具為Car聚合生成名為CarAggregate的聚合對象的接口 (Java代碼) 如下：

---

```
public interface CarAggregate {
    CarState getState();

    List<Event> getChanges();

    void create(CarCommand.CreateCar c);

    void mergePatch(CarCommand.MergePatchCar c);

    void delete(CarCommand.DeleteCar c);

    void rotate(TireWheelIdPair[] tireWheelIdPairs, Long
version, String commandId, String requesterId,
CarCommands.Rotate c);

    void throwOnInvalidStateTransition(Command c);
}
```

---

在CarAggregate接口的方法中，工具在默認情況下會生成的create、mergePatch、delete方法，它們的參數只有一個，因為像CreateCar這樣的用於創建實體的命令對象可能有很多屬性。而對於rotate這樣的開發人員“自定義”的方法（命令），不鼓勵使用太多的參數。

為什麼我們需要CarAggregate與CarState這兩個不同的對象？

聚合對象收到命令後，會根據業務規則檢驗命令，決定是否允許發生對應的“事件”，這是聚合的核心業務邏輯。如果“事件”可以發生，那麼聚合對象會生成“事件對象”，然後把它交給狀態對象，由狀態對象根據事件對象修改（mutate）自身的狀態。也就是說修改狀態的邏輯是由狀態對象負責的。如果把這些邏輯都放在同一個對象裡，那這個對象的職責就太多了。

事件是已經發生的事實。應用事件修改狀態的邏輯一般來說比較簡單。比如，當名稱為{Xxx}的實體的狀態對象（{Xxx}State）收到一個Renamed事件時，它可能只是簡單地設置自身的name屬性（Java代碼），示例如下：

---

```
public class {Xxx}StateImpl implements {Xxx}State {  
    private String name;  
  
    public void setName(String name) {
```

```
        this.name = name;
    }
    // ...

    public void mutate(Event e) {
        if (e instanceof Renamed) {
            Renamed renamed = (Renamed) e;
            setName(renamed.getNewName());
        }
        // ...
    }
    // ...
}
```

---

## 5.事件存儲

DDDML工具生成的表示事件存儲 ( Event Store ) 的接口如下 ( Java 代碼 ) :

---

```
import java.util.Collection;
import java.util.function.Consumer;

public interface EventStore {
    EventStream loadEventStream(EventStoreAggregateId
aggregateId);

    void appendEvents(EventStoreAggregateId aggregateId,
long version, Collection<Event> events,
Consumer<Collection<Event>> afterEventsAppended);

    Event getEvent(Class eventType, EventStoreAggregateId
eventStoreAggregateId, long version);

    Event getEvent(EventStoreAggregateId
eventStoreAggregateId, long version);

    EventStream loadEventStream(Class eventType,
EventStoreAggregateId eventStoreAggregateId, long version);
}
```

---

## 6. 存儲庫 ( Repository )

聚合與實體是兩個不同的概念，因為有了聚合，所以我們才有必要引入對應的Repository的概念。Repository會把一個聚合當成一個整體對待。在沒有聚合這個概念時，在數據訪問層中與實體對應的那個東西往往被稱為DAO（數據訪問對象），我們需要把Repository和DAO這兩個概念區分開。

DDDML工具為聚合Car生成了兩個Repository接口。在實現命令方法的時候，一般只會用到CarStateRepository接口：

---

```
public interface CarStateRepository {  
    CarState get(String id, boolean nullAllowed);  
  
    void save(CarState state);  
}
```

---

默認情況下，Repository返回的實體的狀態對象是隻讀的。調用Repository的客戶端（Client）代碼不能直接設置（set）狀態對象的屬性。想要修改狀態對象，需要生成事件對象，然後調用狀態對象的mutate方法——這是在Aggregate對象中完成的。

另外一個接口CarStateQueryRepository主要用於查詢聚合的狀態，示例如下：

---

```
import java.util.*;
import org.dddml.support.criterion.Criterion;

public interface CarStateQueryRepository {
    CarState get(String id);

    Iterable<CarState> getAll(Integer firstResult, Integer maxResults);
    Iterable<CarState> get(Iterable<Map.Entry<String, Object>> filter, List<String> orders, Integer firstResult, Integer maxResults);
    Iterable<CarState> get(Criterion filter, List<String> orders, Integer firstResult, Integer maxResults);
    Iterable<CarState> getByProperty(String propertyName, Object propertyValue, List<String> orders, Integer firstResult, Integer maxResults);

    CarState getFirst(Iterable<Map.Entry<String, Object>> filter, List<String> orders);
    CarState getFirst(Map.Entry<String, Object> keyValue, List<String> orders);

    long getCount(Iterable<Map.Entry<String, Object>> filter);
    long getCount(Criterion filter);

    WheelState getWheel(String carId, String wheelId);
    Iterable<WheelState> getWheels(String carId, Criterion filter, List<String> orders);

    TireState getTire(String carId, String tireId);
    Iterable<TireState> getTires(String carId, Criterion filter, List<String> orders);

    PositionState getPosition(String carId, String tireId, Long positionId);
    Iterable<PositionState> getPositions(String carId, String tireId, Criterion filter, List<String> orders);
}
```

---

## 7. 應用服務

## 工具生成的Car聚合的應用服務接口 CarApplicationService ( Java代碼 ) 如下：

---

```
import java.util.*;
import org.dddml.support.criterion.Criterion;

public interface CarApplicationService {
    void when(CarCommand.CreateCar c);
    void when(CarCommand.MergePatchCar c);
    void when(CarCommand.DeleteCar c);
    void when(CarCommands.Rotate c);

    CarState get(String id);

    Iterable<CarState> getAll(Integer firstResult, Integer
maxResults);
    Iterable<CarState> get(Iterable<Map.Entry<String,
Object>> filter, List<String> orders, Integer firstResult,
Integer maxResults);
    Iterable<CarState> get(Criterion filter, List<String>
orders, Integer firstResult, Integer maxResults);
    Iterable<CarState> getByProperty(String propertyName,
Object propertyValue, List<String> orders, Integer
firstResult, Integer maxResults);

    long getCount(Iterable<Map.Entry<String, Object>>
filter);
    long getCount(Criterion filter);

    CarEvent getEvent(String id, long version);

    CarState getHistoryState(String id, long version);

    WheelState getWheel(String carId, String wheelId);
    Iterable<WheelState> getWheels(String carId, Criterion
filter, List<String> orders);

    TireState getTire(String carId, String tireId);
    Iterable<TireState> getTires(String carId, Criterion
filter, List<String> orders);

    PositionState getPosition(String carId, String tireId,
```

```
    Long positionId);  
    Iterable<PositionState> getPositions(String carId,  
    String tireId,  
    Criterion filter, List<String> orders);  
}
```

---

要特別說明的是，以上生成的這些代碼的結構，較大程度地受《實現領域驅動設計》[\[2\]](#)一書中提到的示例項目“lokad-iddd-sample”的影響。

[\[1\]](#) 見<https://tools.ietf.org/html/rfc7386>。

[\[2\]](#) Vaughn Vernon. 實現領域驅動設計. 電子工業出版社, 2014. 見<https://book.douban.com/subject/25844633/>。

## 11.1.2 代碼中的命名問題

### 1. Delete與Remove

讀者可能已經注意到，在默認情況下，我們為聚合根Car生成了一個DeleteCar命令對象，而為聚合內部實體Wheel生成的是名為RemoveWheel的內部命令對象。

為什麼命名會不一樣？因為對這兩個命令的處理邏輯是有明顯差異的。我們允許移除（ Remove ）一個聚合內部的（非聚合根）實體，然後再把它添加回去。而刪除（ Delete ）聚合根的時候，默認生成的實現代碼其實只是設置聚合根的狀態對象的一個特殊標記屬性，然後它就變成了“殭屍”。沒有經過特殊的回收處理之前，殭屍會一直躺在那裡，這時是不能重複創建同樣ID的聚合根的實例的。也就是說，我們不允許殭屍隨便復活。這樣做的部分原因是工具生成的代碼默認就使用了事件溯源模式，且聚合的事件ID默認是由聚合根的ID加上事件發生時聚合根的版本號（ Version ）組成的，允許刪除聚合根會導致產生重複的事件ID。

很多應用都是不允許刪除聚合根的，因為允許刪除聚合根可能會導致重要的信息不可追溯，所以這算不上不近人情的決定。

很多時候，在聚合根的**ID**如何生成、是否允許**Delete**這樣重要的問題上，多花時間思考再做決定是非常必要的。

我們的工具生成的代碼是這樣做的：被刪除的（**Deleted**）聚合根不能重新創建（至少不能馬上重新創建），在默認情況下，應用層不提供查詢被刪除的聚合根的方法。如果開發人員實在很討厭這樣，還可以考慮禁止工具生成**Delete**操作的代碼，轉而使用一個標記屬性（比如叫作**Active**）來實現“軟刪除”，即通過設置這個標記屬性來啟用/禁用一個聚合根的實例。

## 2. 處理聚合與實體的命名問題

在第7章中就討論過聚合與實體的命名問題：

- 聚合的名稱一般與聚合根的名稱相同。
- 但是聚合的名稱也有可能與聚合內所有實體的名稱都不相同。比如一個名為**Order**的聚合，其聚合根的名字可能是**OrderHead**，而**OrderItem**則是這個聚合的一個內部實體的名稱。
- 聚合的名稱還有可能和聚合內部的某個非聚合根實體的名稱相同。比如**ProductPrice**聚合的聚合根是**ProductPriceMaster**，**ProductPrice**實體則是這個聚合的內部實體。

那麼，對代碼中的眾多對象應該怎麼命名，以避免名稱的衝突呢？以下是相關建議，供讀者參考。

對於Client來說，它們對聚合發出的各種命令對象，包括默認生成的Create/Update/Delete命令對象，其實都應該理解為針對聚合的命令。所以：

- 當聚合名稱與聚合內的所有實體的名稱都不一樣時，生成聚合的命令對象可以基於聚合的名稱來命名，比如CreateOrder。

- 當聚合名稱與聚合內部的（非聚合根）實體名稱一樣時，生成聚合的命令對象考慮基於聚合根的名稱來命名，比如CreateProductPriceMaster。否則就可能和默認生成的創建聚合內部實體的內部命令對象（比如CreateProductPrice）的名稱衝突。

對於聚合的事件對象，包括默認會生成表示聚合的狀態已經被修改的事件對象（即那些名為{XxxState}Created/{XxxState}Updated/{XxxState}Deleted的對象），採用與聚合的命令對象類似的命名規則即可（記得使用動詞的過去分詞形式哦）。

對於那些狀態對象，如前所述，聚合是一個“邊界”，我們不關心邊界的狀態，所以狀態對象似乎基於實體（聚合根也是一種實體）的名稱來命名比較合適。比如OrderHeaderState、ProductPriceMaterState、ProductPriceState等。

對於在生成的RESTful API的路徑中使用的名詞，可以有如下考慮。

1 ) 總是可以在路徑中使用實體的名稱訪問相應的資源。比如：

- `{BASE_URL}/orderHeaders/{orderId}/orderItems/{orderItemSeqId}`指向某個訂單的某個訂單行項( OrderItem )。

- `{BASE_URL}/productPriceMasters/{masterId}/productPrices/{fromDate}`指向某個產品的價格。

2 ) 對於聚合的名稱和聚合根以及聚合內部實體的名稱都不一樣的情況，還可以考慮支持使用聚合的名稱來訪問相應的資源。比如：

`{BASE_URL}/orders/{orderId}/orderItems/{orderItemSeqId}`同樣指向某個訂單的某個訂單行項。

### 11.1.3 接口的實現

#### 1. 狀態

工具生成的聚合根Car的狀態對象接口的實現類 ( Java 代碼 ) 如下：

```
public abstract class AbstractCarState implements
CarState.SqlCarState, Saveable {
    private String id;
    private String description;
    // getter/seter 方法省略

    private EntityStateCollection<String, WheelState>
wheels;
    public EntityStateCollection<String, WheelState>
getWheels() {
        return this.wheels;
    }
    public void setWheels(EntityStateCollection<String,
WheelState> wheels) {
        this.wheels = wheels;
    }

    // ...
    private EntityStateCollection<String, TireState> tires;
    private Long version;
    private String createdBy;
    private Date createdAt;
    private String updatedBy;
    private Date updatedAt;
    private Boolean active;
    private Boolean deleted;
    private Boolean stateReadOnly;
    private boolean forReapplying;
    // 省略部分 getter/seter 方法的代碼

    public boolean isStateUnsaved() {
```

```

        return this.getVersion() == null;
    }

    public AbstractCarState(List<Event> events) {
        initializeForReapplying();
        if (events != null && events.size() > 0) {
            this.setId(((CarEvent.SqlCarEvent)
events.get(0)).getCarEventId().getId());
            for (Event e : events) {
                mutate(e);
                this.setVersion((this.getVersion() == null ?
CarState.
VERSION_NULL : this.getVersion()) + 1);
            }
        }
    }

    public AbstractCarState() {
        initializeProperties();
    }

    protected void initializeForReapplying() {
        this.forReapplying = true;
        initializeProperties();
    }

    protected void initializeProperties() {
        wheels = new SimpleWheelStateCollection(this);
        tires = new SimpleTireStateCollection(this);
    }

    @Override
    public int hashCode() {
        // 省略實現代碼
    }

    @Override
    public boolean equals(Object obj) {
        // 省略實現代碼 ...
    }

    public void mutate(Event e) {
        setStateReadOnly(false);
        if (e instanceof CarStateCreated) {
            when((CarStateCreated) e);
        }
    }
}

```

```

        } else if (e instanceof CarStateMergePatched) {
            when((CarStateMergePatched) e);
        } else if (e instanceof CarStateDeleted) {
            when((CarStateDeleted) e);
        } else if (e instanceof
AbstractCarEvent.TireWheelPairsRotated) {
            when((AbstractCarEvent.TireWheelPairsRotated)
e);
        } else {
            throw new
UnsupportedOperationException(String.format(
"Unsupported event type: %1$s", e.getClass().getName()));
        }
    }

    public void when(CarStateCreated e) {
        throwOnWrongEvent(e);
        this.setDescription(e.getDescription());
        this.setActive(e.getActive());
        this.setDeleted(false);
        this.setCreatedBy(e.getCreatedBy());
        this.setCreatedAt(e.getCreatedAt());
        for (WheelEvent.WheelStateCreated innerEvent :
e.getWheelEvents()) {
            WheelState innerState = ((EntityStateCollection.
ModifiableEntityStateCollection<String, WheelState>)
this.getWheels()).
                getOrAdd(((WheelEvent.SqlWheelEvent)
innerEvent).
                    getWheelEventId().getWheelId());
            ((WheelState.SqlWheelState)
innerState).mutate(innerEvent);
        }
        for (TireEvent.TireStateCreated innerEvent :
e.getTireEvents()) {
            // 省略部分代码
        }
    }

    protected void merge(CarState s) {
        if (s == this) {
            return;
        }
        this.setDescription(s.getDescription());
        this.setActive(s.getActive());
    }
}

```

```

        if (s.getWheels() != null) {
            Iterable<WheelState> iterable;
            if (s.getWheels().isLazy()) {
                iterable = s.getWheels().getLoadedStates();
            } else {
                iterable = s.getWheels();
            }
            if (iterable != null) {
                for (WheelState ss : iterable) {
                    WheelState thisInnerState =
((EntityStateCollection.
ModifiableEntityStateCollection<String, WheelState>)
this.getWheels()).getOrAdd(ss.getWheelId());
                    ((AbstractWheelState)
thisInnerState).merge(ss);
                }
            }
        }
        if (s.getWheels() != null) {
            if (s.getWheels() instanceof
EntityStateCollection.
ModifiableEntityStateCollection) {
                if
(((EntityStateCollection.ModifiableEntityStateCollection)
s.getWheels()).getRemovedStates() != null) {
                    for (WheelState ss :
((EntityStateCollection.
ModifiableEntityStateCollection<String, WheelState>)
s.getWheels()).getRemovedStates()) {
                        WheelState thisInnerState =
((EntityStateCollection.
ModifiableEntityStateCollection<String, WheelState>)
this.getWheels()).getOrAdd(ss.getWheelId());
                        ((AbstractWheelStateCollection)
this.getWheels()).remove(thisInnerState);
                    }
                }
            } else {
                if (s.getWheels().isAllLoaded()) {
                    Set<String> removedStateIds = new
HashSet<>(this.getWheels().stream().map(i ->
i.getWheelId()).collect(java.util.stream.Collectors.toList()
));
                    s.getWheels().forEach(i ->

```

```

removedStateIds.remove(i.getWheelId()));
        for (String i : removedStateIds) {
            WheelState thisInnerState =
((EntityStateCollection.
ModifiableEntityStateCollection<String, WheelState>)
this.getWheels()).getOrAdd(i);
            ((AbstractWheelStateCollection)
this.getWheels()).remove(thisInnerState);
        }
    }
}
// 省略部分代碼
}

public void when(CarStateMergePatched e) {
    throwOnWrongEvent(e);
    if (e.getDescription() == null) {
        if (e.getIsPropertyDescriptionRemoved() != null
&& e.
getIsPropertyDescriptionRemoved()) {
            this.setDescription(null);
        }
    } else {
        this.setDescription(e.getDescription());
    }
    // 省略部分代碼
    this.setUpdatedBy(e.getCreatedBy());
    this.setUpdatedAt(e.getCreatedAt());

    for (WheelEvent innerEvent : e.getWheelEvents()) {
        WheelState innerState = ((EntityStateCollection.
ModifiableEntityStateCollection<String, WheelState>)
this.getWheels()).
getOrAdd((WheelEvent.SqlWheelEvent) innerEvent).
getWheelEventId().getWheelId());
        ((WheelState.SqlWheelState)
innerState).mutate(innerEvent);
        if (innerEvent instanceof
WheelEvent.WheelStateRemoved) {
            ((AbstractWheelStateCollection)
this.getWheels()).remove(innerState);
        }
    }
    for (TireEvent innerEvent : e.getTireEvents()) {

```

```

        TireState innerState = ((EntityStateCollection.
ModifiableEntityStateCollection<String, TireState>)
this.getTires()) .
getOrAdd(((TireEvent.SqlTireEvent) innerEvent) .
getTireEventId().getTireId());
        ((TireState.SqlTireState)
innerState).mutate(innerEvent);
        if (innerEvent instanceof
TireEvent.TireStateRemoved) {
            ((AbstractTireStateCollection)
this.getTires()).remove(innerState);
        }
    }

    public void when(CarStateDeleted e) {
        // 省略實現代碼
    }

    public void when(AbstractCarEvent.TireWheelPairsRotated
e) {
        throwOnWrongEvent(e);
        TireWheelIdPair[] tireWheelIdPairs =
e.getTireWheelIdPairs();
        if (this.getCreatedBy() == null) {
this.setCreatedBy(e.getCreatedBy()); }
        if (this.getCreatedAt() == null) {
this.setCreatedAt(e.getCreatedAt()); }
        this.setUpdatedBy(e.getCreatedBy());
        this.setUpdatedAt(e.getCreatedAt());

        CarState updatedCarState = (CarState)
ReflectUtils.invokeStaticMethod(
"org.dddml.templates.tests.domain.car.RotateLogic",
        "mutate",
        new Class[]{CarState.class,
TireWheelIdPair[].class, MutationContext.class},
        new Object[]{this, tireWheelIdPairs,
MutationContext.forEvent(e, s -> {
            if (s == this) {
                return this;
            } else {
                throw new
UnsupportedOperationException(); }
        })
    }
}

```

```

        }
    } ) }
);
if (this != updatedCarState) {
    merge(updatedCarState);
}
}

public void save() {
    ((Saveable) wheels).save();
    ((Saveable) tires).save();
}

protected void throwOnWrongEvent(CarEvent event) {
    // 省略實現代碼
}

public static class SimpleCarState extends
AbstractCarState {
    // 省略實現代碼
}
static class SimpleWheelStateCollection extends
AbstractWheelStateCollection {
    // 省略實現代碼
}
static class SimpleTireStateCollection extends
AbstractTireStateCollection {
    // 省略實現代碼
}
}
}
}

```

---

在上面的實體Car的狀態對象實現類的代碼中，**when(AbstractCarEvent.TireWheelPairs-Rotated e)**方法會使用反射機制去調用位於同一個包（ package ）內的**RotateLogic**類的靜態的**mutate**方法。這裡之所以使用反射，部分原因是我們希望代碼生成之後馬上可以編譯和執行。也就是說，需要編寫如下**RotateLogic**類：

```
package org.dddml.templates.tests.domain.car;

public class RotateLogic {
    // ...
    public static CarState mutate(CarState carState,
        TireWheelIdPair[] tireWheelIdPairs,
        MutationContext<CarState, CarState.MutableCarState>
        mutationContext) {
        // 返回一個修改後的 CarState 的實例
    }
}
```

在上面的**mutate**方法中實現了當**TireWheelPairsRotated**事件發生時修改狀態的邏輯。從這個方法的參數**carState**傳入的是事件發生前**Car**的狀態，這個方法需要返回事件發生後**Car**的新狀態。

為什麼不生成一個叫作**RotateLogic**的接口，然後讓開發人員實現這個接口呢？這是因為我們希望開發人員使用函數式編程風格來編寫實體方法的業務邏輯。也就是說，不希望開發人員在這裡使用實例的字段成員，這裡的業務邏輯代碼最好是無狀態的。

 提示想要實現**Car**實體的**Rotate**方法，可以選擇如下方式。

- 像前面這樣寫一個**RotateLogic**類。
- 用**Override**（重寫）**DDDML**工具生成的**CarAggregate**實現類中的**rotate**方法。

·用Override ( 重寫 ) DDDML工具生成的CarApplicationService實現類中的when ( CarCommands.Rotate c ) 方法。

後文可以看到後面兩種做法的例子。

表示車輪 ( Wheel ) 的狀態對象的集合的SimpleWheelStateCollection類擴展自AbstractWheelStateCollection，後者的代碼如下：

---

```
public abstract class AbstractWheelStateCollection
implements
EntityStateCollection.ModifiableEntityStateCollection<String
, WheelState>, Saveable {
    protected WheelStateDao getWheelStateDao() {
        return (WheelStateDao)
ApplicationContext.current.get("wheelStateDao");
    }

    private CarState carState;

    private Map<CarWheelId, WheelState> loadedWheelStates =
new HashMap<CarWheelId, WheelState>();
    private Map<CarWheelId, WheelState> removedWheelStates =
new HashMap<CarWheelId, WheelState>();

    protected Iterable<WheelState> getLoadedWheelStates() {
        return this.loadedWheelStates.values();
    }

    private boolean forReapplying;
    public boolean getForReapplying() {
        return forReapplying;
    }
    public void setForReapplying(boolean forReapplying) {
        this.forReapplying = forReapplying;
    }
}
```

```

private Boolean stateCollectionReadOnly;

public Boolean getStateCollectionReadOnly() {
    if (this.carState instanceof AbstractCarState) {
        if (((AbstractCarState)
this.carState).getStateReadOnly() != null &&
((AbstractCarState) this.carState).getStateReadOnly()) {
            return true;
        }
    }
    if (this.stateCollectionReadOnly == null) {
        return false;
    }
    return this.stateCollectionReadOnly;
}

public void setStateCollectionReadOnly(Boolean readOnly)
{
    this.stateCollectionReadOnly = readOnly;
}

private boolean allLoaded;
public boolean isAllLoaded() {
    return this.allLoaded;
}

protected Iterable<WheelState> getInnerIterable() {
    if (!getForReapplying()) {
        assureAllLoaded();
        return this.loadedWheelStates.values();
    } else {
        List<WheelState> ss = new ArrayList<WheelState>
();
        for (WheelState s : loadedWheelStates.values())
{
            if (!
(removedWheelStates.containsKey(((WheelState.SqlWheelState)
s).getCarWheelId()) && s.getDeleted())) {
                ss.add(s);
            }
        }
        return ss;
    }
}

```

```

public boolean isLazy() {
    return true;
}

protected void assureAllLoaded() {
    if (!allLoaded) {
        Iterable<WheelState> ss =
getWheelStateDao().findByCarId(carState.getId(), carState);
        for (WheelState s : ss) {
            if
(!this.loadedWheelStates.containsKey(((WheelState.SqlWheelState)
s).getCarWheelId()))
                &&
!this.removedWheelStates.containsKey(((WheelState.SqlWheelState)
s).getCarWheelId())) {

this.loadedWheelStates.put(((WheelState.SqlWheelState)
s).getCarWheelId(), s);
        }
    }
    allLoaded = true;
}
}

public AbstractWheelStateCollection(CarState outerState)
{
    this.carState = outerState;
    this.setForReapplying(((CarState.SqlCarState)
outerState).getForReapplying());
}

@Override
public Iterator<WheelState> iterator() {
    return getInnerIterable().iterator();
}

public WheelState get(String wheelId) {
    return get(wheelId, true, false);
}

public WheelState getOrAdd(String wheelId) {
    return get(wheelId, false, false);
}

protected WheelState get(String wheelId, boolean

```

```

nullAllowed, boolean forCreation) {
    CarWheelId globalId = new
CarWheelId(carState.getId(), wheelId);
    if (loadedWheelStates.containsKey(globalId)) {
        WheelState state =
loadedWheelStates.get(globalId);
        if (state instanceof AbstractWheelState) {
            ((AbstractWheelState)
state).setStateReadOnly(
getStateCollectionReadOnly());
        }
        return state;
    }
    boolean justNewIfNotLoaded = forCreation ||
getForReapplying();
    if (justNewIfNotLoaded) {
        if (getStateCollectionReadOnly()) {
            throw new UnsupportedOperationException(
"State collection is ReadOnly.");
        }
        WheelState state =
AbstractWheelState.SimpleWheelState.newForReapplying();
        ((WheelState.SqlWheelState)
state).setCarWheelId(globalId);
        loadedWheelStates.put(globalId, state);
        return state;
    } else {
        WheelState state =
getWheelStateDao().get(globalId, nullAllowed, carState);
        if (state != null) {
            if (state instanceof AbstractWheelState) {
                ((AbstractWheelState)
state).setStateReadOnly
(getStateCollectionReadOnly());
            }
            if (((WheelState.SqlWheelState)
state).isStateUnsaved() && getStateCollectionReadOnly()) {
                return state;
            }
            loadedWheelStates.put(globalId, state);
        }
        return state;
    }
}

```

```

    public boolean remove(WheelState state) {
        if (getStateCollectionReadOnly()) {
            throw new UnsupportedOperationException(
"State collection is ReadOnly.");
        }

        this.loadedWheelStates.remove(((WheelState.SqlWheelState)
state).getCarWheelId());
        if
        (this.removedWheelStates.containsKey(((WheelState.SqlWheelSt
ate) state).getCarWheelId())) {
            return false;
        }

        this.removedWheelStates.put(((WheelState.SqlWheelState)
state).getCarWheelId(), state);
        return true;
    }

    public boolean add(WheelState state) {
        if (getStateCollectionReadOnly()) {
            throw new UnsupportedOperationException(
"State collection is ReadOnly.");
        }

        this.removedWheelStates.remove(((WheelState.SqlWheelState)
state).getCarWheelId());
        if
        (this.loadedWheelStates.containsKey(((WheelState.SqlWheelSta
te) state).getCarWheelId())) {
            return false;
        }

        this.loadedWheelStates.put(((WheelState.SqlWheelState)
state).getCarWheelId(), state);
        return true;
    }

    public Collection<WheelState> getLoadedStates() {
        return
Collections.unmodifiableCollection(this.loadedWheelStates.va
lues());
    }

    public Collection<WheelState> getRemovedStates() {

```

```
        return
Collections.unmodifiableCollection(this.removedWheelStates.v
alues());
    }

    public int size() {
        assureAllLoaded();
        return this.loadedWheelStates.size();
    }

    public boolean isEmpty() {
        assureAllLoaded();
        return this.loadedWheelStates.isEmpty();
    }

    public boolean contains(Object o) {
        if (loadedWheelStates.values().contains(o)) {
            return true;
        }
        assureAllLoaded();
        return this.loadedWheelStates.containsValue(o);
    }

    public Object[] toArray() {
        assureAllLoaded();
        return this.loadedWheelStates.values().toArray();
    }

    public <T> T[] toArray(T[] a) {
        assureAllLoaded();
        return this.loadedWheelStates.values().toArray(a);
    }

    public boolean containsAll(Collection<?> c) {
        assureAllLoaded();
        return
this.loadedWheelStates.values().containsAll(c);
    }

    public boolean addAll(Collection<? extends WheelState>
c) {
        boolean b = false;
        for (WheelState s : c) {
            if (add(s)) { b = true; }
        }
    }
}
```

```
        return b;
    }

    public boolean remove(Object o) {
        return remove((WheelState) o);
    }

    public boolean removeAll(Collection<?> c) {
        boolean b = false;
        for (Object s : c) {
            if (remove(s)) { b = true; }
        }
        return b;
    }

    public boolean retainAll(Collection<?> c) {
        throw new UnsupportedOperationException();
    }

    public void clear() {
        assureAllLoaded();
        this.loadedWheelStates.values().forEach(s ->
this.removedWheelStates.put(((WheelState.SqlWheelState)
s).getCarWheelId(), s));
        this.loadedWheelStates.clear();
    }

    public void save() {
        for (WheelState s : this.getLoadedWheelStates()) {
            getWheelStateDao().save(s);
        }
        for (WheelState s :
this.removedWheelStates.values()) {
            getWheelStateDao().delete(s);
        }
    }
}
```

---

在上面的代碼中，可以看到有個名為 **WheelStateDao** 的接口。這個接口負責 **Wheel** 實體的狀態對象的懶 (Lazy) 加載以及持久化。可以認為，這

個WheelStateDao接口只是基於關係數據庫實現的Car聚合的Repository的一部分。如果我們不使用關係數據庫或者不打算這樣實現聚合內部的實體集合的懶加載，那麼這個接口以及它的實現類可能是沒有必要的。比如說，如果我們使用的是MongoDB這樣的文檔型數據庫，那麼把一個Car以及它的Wheel的狀態都保存到一個文檔中可能是一個不錯的選擇。

另外，在這個例子裡並沒有使用Hibernate的One to Many映射，這是因為我們想讓開發人員直接控制集合的加載邏輯。這樣做的另外一個考慮是，如果有必要，可以更容易地將Repository從使用Hibernate的實現替換為使用MyBatis或直接使用JDBC的實現。

## 2. 命令

工具生成的Car的命令對象的實現類代碼如下：

---

```
public abstract class AbstractCarCommand extends AbstractCommand implements CarCommand {
    private String id;
    private Long version;
    // 省略這些 fields 的 getter/setter 方法

    public static abstract class AbstractCreateOrMergePatchCar extends AbstractCarCommand
        implements CreateOrMergePatchCar {
        private String description;
        private Boolean active;
        // 省略這些 fields 的 getter/setter 方法

        public WheelCommand.CreateWheel newCreateWheel() {
            AbstractWheelCommand.SimpleCreateWheel c = new
```

```

AbstractWheelCommand.SimpleCreateWheel();
    c.setCarId(this.getId());
    return c;
}

public WheelCommand.MergePatchWheel
newMergePatchWheel() {
    AbstractWheelCommand.SimpleMergePatchWheel c =
new AbstractWheelCommand.SimpleMergePatchWheel();
    c.setCarId(this.getId());
    return c;
}

public WheelCommand.RemoveWheel newRemoveWheel() {
    AbstractWheelCommand.SimpleRemoveWheel c = new
AbstractWheelCommand.SimpleRemoveWheel();
    c.setCarId(this.getId());
    return c;
}

public TireCommand.CreateTire newCreateTire() {
    // ...
}

public TireCommand.MergePatchTire
newMergePatchTire() {
    // ...
}

public TireCommand.RemoveTire newRemoveTire() {
    // ...
}

public static abstract class AbstractCreateCar extends
AbstractCreateOrMergePatchCar implements CreateCar {
    @Override
    public String getCommandType() {
        return COMMAND_TYPE_CREATE;
    }

    private CreateWheelCommandCollection
createWheelCommands = new
SimpleCreateWheelCommandCollection();

```

```

        public CreateWheelCommandCollection
getCreateWheelCommands() {
    return this.createWheelCommands;
}

        public CreateWheelCommandCollection getWheels() {
    return this.createWheelCommands;
}

        private CreateTireCommandCollection
createTireCommands =
new SimpleCreateTireCommandCollection();

        public CreateTireCommandCollection
getCreateTireCommands() {
    return this.createTireCommands;
}

        public CreateTireCommandCollection getTires() {
    return this.createTireCommands;
}

}

public static abstract class AbstractMergePatchCar
extends AbstractCreateOrMergePatchCar implements
MergePatchCar {
    @Override
    public String getCommandType() {
        return COMMAND_TYPE_MERGE_PATCH;
    }

    private Boolean isPropertyDescriptionRemoved;
    private Boolean isPropertyActiveRemoved;
    private WheelCommandCollection wheelCommands = new
SimpleWheelCommandCollection();
    // 省略這些 fields 的 getter/setter 方法
}

public static class SimpleCreateCar extends
AbstractCreateCar {
}
public static class SimpleMergePatchCar extends
AbstractMergePatchCar {
}
public static class SimpleDeleteCar extends

```

```
AbstractCarCommand implements DeleteCar {
    @Override
    public String getCommandType() {
        return COMMAND_TYPE_DELETE;
    }
}

public static class SimpleCreateWheelCommandCollection
implements CreateWheelCommandCollection {
    private List<WheelCommand.CreateWheel> innerCommands
= new ArrayList<WheelCommand.CreateWheel>();

    public void add(WheelCommand.CreateWheel c) {
        innerCommands.add(c);
    }

    public void remove(WheelCommand.CreateWheel c) {
        innerCommands.remove(c);
    }

    public void clear() {
        innerCommands.clear();
    }

    @Override
    public Iterator<WheelCommand.CreateWheel> iterator()
{
    return innerCommands.iterator();
}
}

public static class SimpleWheelCommandCollection
implements WheelCommandCollection {
    private List<WheelCommand> innerCommands = new
ArrayList<WheelCommand>();

    public void add(WheelCommand c) {
        innerCommands.add(c);
    }

    public void remove(WheelCommand c) {
        innerCommands.remove(c);
    }

    public void clear() {
```

```
        innerCommands.clear();
    }

    @Override
    public Iterator<WheelCommand> iterator() {
        return innerCommands.iterator();
    }
}

public static class SimpleCreateTireCommandCollection
implements CreateTireCommandCollection {
    // ...
}

public static class SimpleTireCommandCollection
implements TireCommandCollection {
    // ...
}
```

---

這些命令對象的抽象基類**AbstractCommand**的代碼如下：

---

```
package org.dddml.templates.tests.domain;

public abstract class AbstractCommand implements Command {
    protected String commandType;
    private String commandId;
    private String requesterId;
    private java.util.Map<String, Object> commandContext =
new java.util.HashMap<>();
    // 省略這些 fields 的 getter/setter 方法
}
```

---

### 3.事件

實現Car的事件對象接口的抽象基類  
**AbstractCarEvent**的部分代碼如下：

---

```
import org.dddml.templates.tests.domain.AbstractEvent;

public abstract class AbstractCarEvent extends AbstractEvent
implements CarEvent.SqlCarEvent {
    private CarEventId carEventId;
    public CarEventId getCarEventId() {
        return this.carEventId;
    }
    public void setCarEventId(CarEventId eventId) {
        this.carEventId = eventId;
    }

    public String getId() {
        return getCarEventId().getId();
    }
    public void setId(String id) {
        getCarEventId().setId(id);
    }

    private boolean eventReadOnly;
    // 省略 getter/setter 方法

    public Long getVersion() {
        return getCarEventId().getVersion();
    }
    public void setVersion(Long version) {
        getCarEventId().setVersion(version);
    }

    private String createdBy;
    private Date createdAt;
    private String commandId;
    private String commandType;
    // 省略以上 fields 的 getter/setter 方法

    protected AbstractCarEvent() {
    }
    protected AbstractCarEvent(CarEventId eventId) {
        this.carEventId = eventId;
    }
}
```

```
        }
        // 省略部分代碼
    public abstract String getEventType();

    public static class CarClobEvent extends
AbstractCarEvent {
        protected Map<String, Object> getLobProperties() {
            return lobProperties;
        }
        protected void setLobProperties(Map<String, Object>
lobProperties) {
            if (lobProperties == null) {
                throw new
IllegalArgumentException("lobProperties is null.");
            }
            this.lobProperties = lobProperties;
        }

        private Map<String, Object> lobProperties = new
HashMap<>();
        protected String getLobText() {
            return
ApplicationContext.current.getClobConverter().toString(getLo
bProperties());
        }
        protected void setLobText(String text) {
            getLobProperties().clear();
            Map<String, Object> ps =
ApplicationContext.current.getClobConverter().parseLobProper
ties(text);
            if (ps != null) {
                for (Map.Entry<String, Object> kv :
ps.entrySet()) {
                    getLobProperties().put(kv.getKey(),
kv.getValue());
                }
            }
        }

        @Override
    public String getEventType() {
        return "CarClobEvent";
    }
}
```

```

        public static class TireWheelPairsRotated extends
CarClobEvent {
    @Override
    public String getEventType() {
        return "TireWheelPairsRotated";
    }

    public TireWheelIdPair[] getTireWheelIdPairs() {
        Object val =
getLobProperties().get("tireWheelIdPairs");
        if (val instanceof TireWheelIdPair[]) {
            return (TireWheelIdPair[]) val;
        }
        return
ApplicationContext.current.getTypeConverter().convertValue(v
al, TireWheelIdPair[].class);
    }

    public void setTireWheelIdPairs(TireWheelIdPair[]
value) {
        getLobProperties().put("tireWheelIdPairs",
value);
    }
}

```

---

可以看到，對於在DDDM中定義的方法（命令），工具默認生成的命令對象**Rotate**以及執行該命令可能產生的事件對象**TireWheelPairsRotated**具有相似的屬性，比如它們都有**tireWheelIdPairs**屬性。除了一些特殊的屬性，事件對象**TireWheelPairsRotated**內部會使用一個**Map** (**lobProperties**字段) 來實現各個屬性的**getter**和**setter**方法。這樣在使用**SQL**數據庫實現事件的持久化時，直接把這個**Map**序列化後保存到數據庫中的一個列即可，從而避免頻繁更新數據庫**Schema**。

## 4. Aggregate

表示聚合對象的CarAggregate接口的實現類的代碼如下：

---

```
public abstract class AbstractCarAggregate extends AbstractAggregate implements CarAggregate {
    private CarState.MutableCarState state;
    private List<Event> changes = new ArrayList<Event>();

    public AbstractCarAggregate(CarState state) {
        this.state = (CarState.MutableCarState) state;
    }

    public CarState getState() {
        return this.state;
    }
    public List<Event> getChanges() {
        return this.changes;
    }

    public void create(CarCommand.CreateCar c) {
        if (c.getVersion() == null) {
            c.setVersion(CarState.VERSION_NULL);
        }
        CarEvent e = map(c);
        apply(e);
    }

    public void mergePatch(CarCommand.MergePatchCar c) {
        CarEvent e = map(c);
        apply(e);
    }

    public void delete(CarCommand.DeleteCar c) {
        CarEvent e = map(c);
        apply(e);
    }

    public void throwOnInvalidStateTransition(Command c) {
        CarCommand.throwOnInvalidStateTransition(this.state,
```

```

c) ;
}

protected void apply(Event e) {
    onApplying(e);
    state.mutate(e);
    changes.add(e);
}

protected CarEvent map(CarCommand.CreateCar c) {
    CarEventId stateEventId = new CarEventId(c.getId(),
c.getVersion());
    CarEvent.CarStateCreated e =
newCarStateCreated(stateEventId);
    e.setDescription(c.getDescription());
    e.setActive(c.getActive());
    ((AbstractCarEvent)
e).setCommandId(c.getCommandId());
    e.setCreatedBy(c.getRequesterId());
    e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
    Long version = c.getVersion();
    for (WheelCommand.CreateWheel innerCommand :
c.getCreateWheelCommands()) {
        throwOnInconsistentCommands(c, innerCommand);
        WheelEvent.WheelStateCreated innerEvent =
mapCreate(innerCommand, c, version, this.state);
        e.addWheelEvent(innerEvent);
    }
    for (TireCommand.CreateTire innerCommand :
c.getCreateTireCommands()) {
        throwOnInconsistentCommands(c, innerCommand);
        TireEvent.TireStateCreated innerEvent =
mapCreate(innerCommand, c, version, this.state);
        e.addTireEvent(innerEvent);
    }
    return e;
}

protected CarEvent map(CarCommand.MergePatchCar c) {
    CarEventId stateEventId = new CarEventId(c.getId(),
c.getVersion());
    CarEvent.CarStateMergePatched e =
newCarStateMergePatched(stateEventId);

```

```

        e.setDescription(c.getDescription());
        e.setActive(c.getActive());

        e.setPropertyDescriptionRemoved(c.getPropertyDescriptionRemoved());
        e.setPropertyActiveRemoved(c.getPropertyActiveRemoved());
        e.setCommandId(c.getCommandId());
        e.setCreatedBy(c.getRequesterId());
        e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.util.Date.class));
        Long version = c.getVersion();
        for (WheelCommand innerCommand :
c.getWheelCommands()) {
            throwOnInconsistentCommands(c, innerCommand);
            WheelEvent innerEvent = map(innerCommand, c,
version, this.state);
            e.addWheelEvent(innerEvent);
        }
        for (TireCommand innerCommand : c.getTireCommands())
{
            throwOnInconsistentCommands(c, innerCommand);
            TireEvent innerEvent = map(innerCommand, c,
version, this.state);
            e.addTireEvent(innerEvent);
        }
        return e;
    }

    protected CarEvent map(CarCommand.DeleteCar c) {
        CarEventId stateEventId = new CarEventId(c.getId(),
c.getVersion());
        CarEvent.CarStateDeleted e =
newCarStateDeleted(stateEventId);
        ((AbstractCarEvent)
e).setCommandId(c.getCommandId());
        e.setCreatedBy(c.getRequesterId());
        e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.util.Date.class));
        return e;
    }
}

```

```

        protected WheelEvent map(WheelCommand c, CarCommand
outerCommand, Long version, CarState outerState) {
            WheelCommand.CreateWheel create =
(c.getCommandType().equals(CommandType.CREATE)) ?
((WheelCommand.CreateWheel) c) : null;
            if (create != null) {
                return mapCreate(create, outerCommand, version,
outerState);
            }
            WheelCommand.MergePatchWheel merge =
(c.getCommandType().equals(CommandType.MERGE_PATCH)) ?
((WheelCommand.MergePatchWheel) c) : null;
            if (merge != null) {
                return mapMergePatch(merge, outerCommand,
version, outerState);
            }
            WheelCommand.RemoveWheel remove =
(c.getCommandType().equals(CommandType.REMOVE)) ?
((WheelCommand.RemoveWheel) c) : null;
            if (remove != null) {
                return mapRemove(remove, outerCommand, version,
outerState);
            }
            throw new UnsupportedOperationException();
        }

        protected WheelEvent.WheelStateCreated
mapCreate(WheelCommand.CreateWheel c, CarCommand
outerCommand, Long version, CarState outerState) {
        (AbstractCommand)
c).setRequesterId(outerCommand.getRequesterId());
        WheelEventId stateEventId = new
WheelEventId(outerState.getId(), c.getWheelId(), version);
        WheelEvent.WheelStateCreated e =
newWheelStateCreated(stateEventId);
        WheelState s =
((EntityStateCollection.ModifiableEntityStateCollection<
String, WheelState>)
outerState.getWheels()).getOrAdd(c.getWheelId());
        e.setActive(c.getActive());
        e.setCreatedBy(c.getRequesterId());
        e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
    }
}

```

```

        return e;
    } // END map(Create... //////////////////////////////

    protected WheelEvent.WheelStateMergePatched
mapMergePatch(WheelCommand.MergePatchWheel c, CarCommand
outerCommand, Long version, CarState outerState) {
    (AbstractCommand)
c).setRequesterId(outerCommand.getRequesterId());
    WheelEventId stateEventId = new
WheelEventId(outerState.getId(), c.getWheelId(), version);
    WheelEvent.WheelStateMergePatched e =
newWheelStateMergePatched(
stateEventId);
    WheelState s =
((EntityStateCollection.ModifiableEntityStateCollection<
String, WheelState>
outerState.getWheels()).getOrAdd(c.getWheelId()));
    e.setActive(c.getActive());

e.setPropertyActiveRemoved(c.getPropertyActiveRemoved());
;
    e.setCreatedBy(c.getRequesterId());
    e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
    return e;
} // END map(MergePatch... //////////////////////////////

    protected WheelEvent.WheelStateRemoved
mapRemove(WheelCommand.RemoveWheel c, CarCommand
outerCommand, Long version, CarState outerState) {
    (AbstractCommand)
c).setRequesterId(outerCommand.getRequesterId());
    WheelEventId stateEventId = new
WheelEventId(outerState.getId(), c.getWheelId(), version);
    WheelEvent.WheelStateRemoved e =
newWheelStateRemoved(stateEventId);
    e.setCreatedBy(c.getRequesterId());
    e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
    return e;
} // END map(Remove... //////////////////////////////

```

```
protected TireEvent map(TireCommand c, CarCommand
outerCommand, Long version, CarState outerState) {
    // ...
}

protected TireEvent.TireStateCreated
mapCreate(TireCommand.CreateTire c, CarCommand outerCommand,
Long version, CarState outerState) {
    // ...
}

protected TireEvent.TireStateMergePatched
mapMergePatch(TireCommand.MergePatchTire c, CarCommand
outerCommand, Long version, CarState outerState) {
    // ...
}

protected TireEvent.TireStateRemoved
mapRemove(TireCommand.RemoveTire c, CarCommand outerCommand,
Long version, CarState outerState) {
    // ...
}

protected PositionEvent map(PositionCommand c,
TireCommand outerCommand, Long version, TireState
outerState) {
    // ...
}

protected PositionEvent.PositionStateCreated
mapCreate(PositionCommand.CreatePosition c, TireCommand
outerCommand, Long version, TireState outerState) {
    // ...
}

protected PositionEvent.PositionStateMergePatched
mapMergePatch(PositionCommand.MergePatchPosition c,
TireCommand outerCommand, Long version, TireState
outerState) {
    // ...
}

protected PositionEvent.PositionStateRemoved
mapRemove(PositionCommand.RemovePosition c, TireCommand
outerCommand, Long version, TireState outerState) {
```

```

        // ...
    }

    protected void throwOnInconsistentCommands(CarCommand
command, WheelCommand innerCommand) {
        // ...
    }

    protected void throwOnInconsistentCommands(CarCommand
command, TireCommand innerCommand) {
        // ...
    }

    protected void throwOnInconsistentCommands(TireCommand
command, PositionCommand innerCommand) {
        // ...
    }

    protected CarEvent.CarStateCreated
newCarStateCreated(Long version, String commandId, String
requesterId) {
        CarEventId stateEventId = new
CarEventId(this.state.getId(), version);
        CarEvent.CarStateCreated e =
newCarStateCreated(stateEventId);
        ((AbstractCarEvent) e).setCommandId(commandId);
        e.setCreatedBy(requesterId);
        e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
        return e;
    }

    protected CarEvent.CarStateMergePatched
newCarStateMergePatched(Long version, String commandId,
String requesterId) {
        CarEventId stateEventId = new
CarEventId(this.state.getId(), version);
        CarEvent.CarStateMergePatched e =
newCarStateMergePatched(stateEventId);
        ((AbstractCarEvent) e).setCommandId(commandId);
        e.setCreatedBy(requesterId);
        e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
        return e;
    }
}

```

```

        protected CarEvent.CarStateDeleted
newCarStateDeleted(Long version, String commandId, String
requesterId) {
    CarEventId stateEventId = new
CarEventId(this.state.getId(), version);
    CarEvent.CarStateDeleted e =
newCarStateDeleted(stateEventId);
    ((AbstractCarEvent) e).setCommandId(commandId);
    e.setCreatedBy(requesterId);
    e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
    return e;
}

        protected CarEvent.CarStateCreated
newCarStateCreated(CarEventId stateEventId) {
    return new
AbstractCarEvent.SimpleCarStateCreated(stateEventId);
}

        protected CarEvent.CarStateMergePatched
newCarStateMergePatched(CarEventId stateEventId) {
    return new
AbstractCarEvent.SimpleCarStateMergePatched(stateEventId);
}

        protected CarEvent.CarStateDeleted
newCarStateDeleted(CarEventId stateEventId) {
    return new
AbstractCarEvent.SimpleCarStateDeleted(stateEventId);
}

        protected WheelEvent.WheelStateCreated
newWheelStateCreated(WheelEventId stateEventId) {
    return new
AbstractWheelEvent.SimpleWheelStateCreated(stateEventId);
}

        protected WheelEvent.WheelStateMergePatched
newWheelStateMergePatched(
WheelEventId stateEventId) {
    return new
AbstractWheelEvent.SimpleWheelStateMergePatched(
stateEventId);
}

```

```

    protected WheelEvent.WheelStateRemoved
newWheelStateRemoved(WheelEventId stateEventId) {
    return new
AbstractWheelEvent.SimpleWheelStateRemoved(stateEventId);
}
// 省略部分代碼

    public static class SimpleCarAggregate extends
AbstractCarAggregate {
    public SimpleCarAggregate(CarState state) {
        super(state);
    }

    @Override
    public void rotate(TireWheelIdPair[]
tireWheelIdPairs, Long version, String commandId, String
requesterId, CarCommands.Rotate c) {
        try {
            verifyRotate(tireWheelIdPairs, c);
        } catch (Exception ex) {
            throw new DomainError("VerificationFailed",
ex);
        }
        Event e =
newTireWheelPairsRotated(tireWheelIdPairs, version,
commandId, requesterId);
        apply(e);
    }

    protected void verifyRotate(TireWheelIdPair[]
tireWheelIdPairs, CarCommands.Rotate c) {
        TireWheelIdPair[] TireWheelIdPairs =
tireWheelIdPairs;
        ReflectUtils.invokeStaticMethod(
"org.ddm.templates.tests.domain.car.RotateLogic",
"verify",
new Class[]{CarState.class,
TireWheelIdPair[].class, VerificationContext.class},
new Object[]{getState(),
tireWheelIdPairs, VerificationContext.forCommand(c)}
);
    }
}

```

```

        protected AbstractCarEvent.TireWheelPairsRotated
newTireWheelPairsRotated(TireWheelIdPair[] tireWheelIdPairs,
Long version, String commandId, String requesterId) {
    CarEventId eventId = new
CarEventId(getState().getId(), version);
    AbstractCarEvent.TireWheelPairsRotated e = new
AbstractCarEvent.TireWheelPairsRotated();
    e.setTireWheelIdPairs(tireWheelIdPairs);
    e.setCommandId(commandId);
    e.setCreatedBy(requesterId);
    e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.ut
il.Date.class));
    e.setCarEventId(eventId);
    return e;
}
}
}
}

```

---

可以看到，在應用事件之前，**SimpleCarAggregate**使用**verifyRotate**方法對客戶端傳入的命令信息進行了檢驗。這個方法的默認邏輯是使用反射調用位於同一個包內的**RotateLogic**類的靜態**verify**方法。之所以使用反射，部分原因是我們希望代碼生成之後是馬上可以編譯和執行的。也就是說，需要編寫如下**RotateLogic**類：

---

```

package org.dddml.templates.tests.domain.car;

public class RotateLogic {
    public static void verify(CarState carState,
TireWheelIdPair[] tireWheelIdPairs, VerificationContext
verificationContext) {
        // 在這裡檢驗命令信息，如命令“不合法”則拋出異常！
    }

    public static CarState mutate(CarState carState,

```

```
TireWheelIdPair[] tireWheelIdPairs,  
MutationContext<CarState, CarState.MutableCarState>  
mutationContext) {  
    //...  
}  
}
```

---

## 5. 應用服務

表示應用服務的**CarApplicationService**接口的實現類如下：

---

```
import java.util.function.Consumer;  
import org.ddm.support.criterion.Criterion;  
  
public abstract class AbstractCarApplicationService  
implements CarApplicationService {  
    private EventStore eventStore;  
    protected EventStore getEventStore() {  
        return eventStore;  
    }  
  
    private CarStateRepository stateRepository;  
    protected CarStateRepository getStateRepository() {  
        return stateRepository;  
    }  
  
    private CarStateQueryRepository stateQueryRepository;  
    protected CarStateQueryRepository  
getStateQueryRepository() {  
        return stateQueryRepository;  
    }  
  
    public AbstractCarApplicationService(EventStore  
eventStore, CarStateRepository stateRepository,  
CarStateQueryRepository stateQueryRepository) {  
        this.eventStore = eventStore;  
        this.stateRepository = stateRepository;  
        this.stateQueryRepository = stateQueryRepository;  
    }  
}
```

```

public void when(CarCommand.CreateCar c) {
    update(c, ar -> ar.create(c));
}

public void when(CarCommand.MergePatchCar c) {
    update(c, ar -> ar.mergePatch(c));
}

public void when(CarCommand.DeleteCar c) {
    update(c, ar -> ar.delete(c));
}

public void when(CarCommands.Rotate c) {
    update(c, ar -> ar.rotate(c.getTireWheelIdPairs(),
c.getVersion(), c.getCommandId(), c.getRequesterId(), c));
}

public CarState get(String id) {
    CarState state = getStateRepository().get(id, true);
    return state;
}

public Iterable<CarState> getAll(Integer firstResult,
Integer maxResults) {
    return getStateQueryRepository().getAll(firstResult,
maxResults);
}

public Iterable<CarState> get(Iterable<Map.Entry<String,
Object>> filter, List<String> orders, Integer firstResult,
Integer maxResults) {
    return getStateQueryRepository().get(filter, orders,
firstResult, maxResults);
}

public Iterable<CarState> get(Criterion filter,
List<String> orders, Integer firstResult, Integer
maxResults) {
    return getStateQueryRepository().get(filter, orders,
firstResult, maxResults);
}

public Iterable<CarState> getByProperty(String
propertyName, Object propertyValue, List<String> orders,

```

```

        Integer firstResult, Integer maxResults) {
            return
        getStateQueryRepository().getByProperty(propertyName,
        propertyValue, orders, firstResult, maxResults);
    }

    public long getCount(Iterable<Map.Entry<String, Object>>
filter) {
        return getStateQueryRepository().getCount(filter);
    }

    public long getCount(Criterion filter) {
        return getStateQueryRepository().getCount(filter);
    }

    public CarEvent getEvent(String id, long version) {
        CarEvent e = (CarEvent)
getEventStore().getEvent(toEventStoreAggregateId(id),
version);
        if (e != null) {
            ((CarEvent.SqlCarEvent)
e).setEventReadOnly(true);
        } else if (version == -1) {
            return getEvent(id, 0);
        }
        return e;
    }

    public CarState getHistoryState(String id, long version)
{
        EventStream eventStream =
getEventStore().loadEventStream(
AbstractCarEvent.class, toEventStoreAggregateId(id), version
- 1);
        return new
AbstractCarState.SimpleCarState(eventStream.getEvents());
    }

    public WheelState getWheel(String carId, String wheelId)
{
        return getStateQueryRepository().getWheel(carId,
wheelId);
    }

    public Iterable<WheelState> getWheels(String carId,

```

```

Criterion filter, List<String> orders) {
    return getStateQueryRepository().getWheels(carId,
filter, orders);
}

    public TireState getTire(String carId, String tireId) {
        return getStateQueryRepository().getTire(carId,
tireId);
    }

    public Iterable<TireState> getTires(String carId,
Criterion filter, List<String> orders) {
        return getStateQueryRepository().getTires(carId,
filter, orders);
    }

    public PositionState getPosition(String carId, String
tireId, Long positionId) {
        return getStateQueryRepository().getPosition(carId,
tireId, positionId);
    }

    public Iterable<PositionState> getPositions(String
carId, String tireId, Criterion filter, List<String> orders)
{
        return getStateQueryRepository().getPositions(carId,
tireId, filter, orders);
    }

    public CarAggregate getCarAggregate(CarState state) {
        return new
AbstractCarAggregate.SimpleCarAggregate(state);
    }

    public EventStoreAggregateId
toEventStoreAggregateId(String aggregateId) {
        return new
EventStoreAggregateId.SimpleEventStoreAggregateId(
aggregateId);
    }

    protected void update(CarCommand c,
Consumer<CarAggregate> action) {
        String aggregateId = c.getId();

```

```

        EventStoreAggregateId eventStoreAggregateId =
toEventStoreAggregateId(
aggregateId);
        CarState state =
getRepository().get(aggregateId, false);
        boolean duplicate = isDuplicateCommand(c,
eventStoreAggregateId, state);
        if (duplicate) {
            return;
        }

        CarAggregate aggregate = getCarAggregate(state);
        aggregate.throwOnInvalidStateTransition(c);
        action.accept(aggregate);
        persist(eventStoreAggregateId, c.getVersion() ==
null ? CarState.VERSION_NULL : c.getVersion(), aggregate,
state);

    }

    private void persist(EventStoreAggregateId
eventStoreAggregateId, long version, CarAggregate aggregate,
CarState state) {
    getEventStore().appendEvents(eventStoreAggregateId,
version,
        aggregate.getChanges(), (events) -> {
            getRepository().save(state);
        });
}

public void initialize(CarEvent.CarStateCreated
stateCreated) {
    String aggregateId = ((CarEvent.SqlCarEvent)
stateCreated).getCarEventId().getId();
    CarState.SqlCarState state = new
AbstractCarState.SimpleCarState();
    state.setId(aggregateId);

    CarAggregate aggregate = getCarAggregate(state);
    (AbstractCarAggregate)
aggregate).apply(stateCreated);

    EventStoreAggregateId eventStoreAggregateId =
toEventStoreAggregateId(
aggregateId);

```

```

        persist(eventStoreAggregateId,
((CarEvent.SqlCarEvent)
stateCreated).getCarEventId().getVersion(), aggregate,
state);
}

protected boolean isDuplicateCommand(CarCommand command,
EventStoreAggregateId eventStoreAggregateId, CarState state)
{
    boolean duplicate = false;
    if (command.getVersion() == null) {
        command.setVersion(CarState.VERSION_NULL);
    }
    if (state.getVersion() != null && state.getVersion()
> command.getVersion()) {
        Event lastEvent =
getEventStore().getEvent(AbstractCarEvent.class,
eventStoreAggregateId, command.getVersion());
        if (lastEvent != null && lastEvent instanceof
AbstractEvent
            && command.getCommandId() != null &&
command.getCommandId().equals(((AbstractEvent)
lastEvent).getCommandId())) {
            duplicate = true;
        }
    }
    return duplicate;
}

public static class SimpleCarApplicationService extends
AbstractCarApplicationService {
    public SimpleCarApplicationService(EventStore
eventStore, CarStateRepository stateRepository,
CarStateQueryRepository stateQueryRepository) {
        super(eventStore, stateRepository,
stateQueryRepository);
    }
}
}

```

---

在上面的代碼中，方法**isDuplicateCommand**用於判斷聚合是否收到了重複的命令。默認情況下，生成

的代碼的實現邏輯是根據命令中聚合根的**ID**、聚合根的版本號（**Version**）以及**Command ID**來判斷在事件存儲中是否已經存在相應的事件，如果事件已存在則說明命令重複。重複的命令會被忽略，從而實現聚合更新操作的幂等性。如果對這個聚合的更新沒有必要使用離線樂觀鎖，可以在**DDDML**文檔中該聚合的**metadata**結點內增加如下鍵值對聲明：

**IgnoringConcurrencyConflict:true**，這樣客戶端調用該聚合的命令方法時就不需要提供版本號了，服務端在執行命令時會使用最新的聚合根的版本號。這時，如果想修改工具生成的**isDuplicateCommand**方法的實現邏輯，（忽略聚合根的版本號）只使用聚合根的**ID**與**Command ID**來判斷命令是否重複，則需要在**metadata**結點內增加如下聲明：**DetectingDuplicateCommandIdEnabled:true**。

## 11.1.4 事件存儲與持久化

默認情況下，DDDDML工具所生成的事件存儲（Event Store）以及管理聚合狀態的存儲庫的代碼都是基於SQL數據庫實現的。

### 1. 為何不使用NoSQL

絕大部分應用在起步階段使用SQL都是正確的選擇。我們很容易找到足夠的掌握SQL的技術人員。

我們尤其喜歡SQL數據庫的Schema，就像喜歡靜態類型的編程語言一樣。在開發比較複雜的企業應用時，有很多NoSQL的Schema-Free（沒有數據庫模式）可不是什麼好特性。使用Schema會讓程序的Bug更少。其實同樣重要的是Schema具備文檔的作用。Schema拯救了很多代碼即文檔（約等於沒有文檔）的開發團隊。很多新人進入某個軟件開發團隊的第一件事情，就是去“讀”數據庫。Schema是機器可以處理的設計文檔，如果沒有Schema，那麼數據遷移工具的實現難度就會加大很多。數據遷移工具可以比較新舊版本數據庫的Schema差異，自動生成遷移腳本——即使自動生成的這些腳本可能只是腳手架代碼，還需要開發人員或DBA（數據庫管理員）去檢查和手動修改，但是這仍然可以節省相當的工作量。

### 2. 事件存儲

使用了事件溯源模式後，只有不變的、一直追加的事件才是絕對必須存儲（持久化）的東西。聚合的最新狀態（當前狀態）的持久化是可選的，因為通過事件我們可以追溯聚合在任意時間點的狀態。

默認的情況下，我們的工具生成的代碼會啟用事件溯源模式，並且在保存聚合事件的同時持久化聚合的最新狀態，這個過程是在一個數據庫事務內完成。這可能會給想要“手動維護數據”的人帶來麻煩，因為修改數據時需要保證數據庫中聚合事件與聚合當前狀態之間的一致性。如果不使用事件溯源，那麼手動維護數據可能只需要更新一個表（即更新聚合的當前狀態）就可以了。如果想要解決這個問題，可以考慮不持久化聚合的最新狀態，當需要聚合的最新狀態時，可以通過重放事件派生出來。從一個聚合最初的創建事件開始重放它的所有事件可能會存在性能問題，所以我們可以考慮在適當的時間點（“檢查點”）保存聚合的狀態快照，這樣當我們需要聚合的最新狀態時，從最近的那個快照開始重放之後的事件即可。

默認情況下，我們為每個聚合生成了單獨的事件存儲。以下是使用**Hibernate**實現的聚合**Car**的事件存儲：

---

```
package org.dddml.templates.tests.domain.car.hibernate;

import java.io.Serializable;
import java.util.*;
import org.dddml.templates.tests.domain.*;
import org.dddml.templates.tests.specialization.*;
```

```

import
org.dddml.templates.tests.specialization.hibernate.AbstractH
ibernateEventStore;
import org.hibernate.*;
import org.hibernate.criterion.*;
import
org.springframework.transaction.annotation.Transactional;
import org.dddml.templates.tests.domain.car.*;

public class HibernateCarEventStore extends
AbstractHibernateEventStore {
    @Override
    protected Serializable getEventId(EventStoreAggregateId
eventStoreAggregateId, long version) {
        return new CarEventId((String)
eventStoreAggregateId.getId(), version);
    }

    @Override
    protected Class getSupportedEventType() {
        return AbstractCarEvent.class;
    }

    @Transactional(readOnly = true)
    @Override
    public EventStream loadEventStream(Class eventType,
EventStoreAggregateId eventStoreAggregateId, long version) {
        Class supportedEventType = AbstractCarEvent.class;
        if (!eventType.isAssignableFrom(supportedEventType))
{
            throw new UnsupportedOperationException();
        }
        String idObj = (String)
eventStoreAggregateId.getId();
        Criteria criteria =
getCurrentSession().createCriteria(AbstractCarEvent.class);
        criteria.add(Restrictions.eq("carEventId.id",
idObj));
        criteria.add(Restrictions.le("carEventId.version",
version));
        criteria.addOrder(Order.asc("carEventId.version"));
        List es = criteria.list();
        for (Object e : es) {
            ((AbstractCarEvent) e).setEventReadOnly(true);
        }
    }
}

```

```
        EventStream eventStream = new EventStream();
        if (es.size() > 0) {
            eventStream.setSteamVersion(((AbstractCarEvent)
es.get(es.size() - 1)).getCarEventId().getVersion());
        }
        eventStream.setEvents(es);
        return eventStream;
    }
}
```

---

它的基類**AbstractHibernateEventStore**的代碼如下：

---

```
package org.dddml.templates.tests.specialization.hibernate;

import org.dddml.templates.tests.specialization.*;
import org.hibernate.*;
import
org.springframework.transaction.annotation.Transactional;
import java.io.Serializable;
import java.util.Collection;
import java.util.function.Consumer;

public abstract class AbstractHibernateEventStore implements
EventStore {
    private SessionFactory sessionFactory;

    public SessionFactory getSessionFactory() {
        return this.sessionFactory;
    }

    public void setSessionFactory(SessionFactory
sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    protected Session getCurrentSession() {
        return this.sessionFactory.getCurrentSession();
    }

    @Transactional(readOnly = true)
```

```

        public EventStream loadEventStream(EventStoreAggregateId
aggregateId) {
            throw new UnsupportedOperationException();
        }

        @Transactional
        public void appendEvents(EventStoreAggregateId
aggregateId, long version, Collection<Event> events,
Consumer<Collection<Event>> afterEventsAppended) {
            for (Event e : events) {
                getCurrentSession().save(e);
                if (e instanceof Saveable) {
                    Saveable saveable = (Saveable) e;
                    saveable.save();
                }
            }
            afterEventsAppended.accept(events);
        }

        @Transactional(readOnly = true)
        public Event getEvent(Class eventTypeId,
EventStoreAggregateId eventStoreAggregateId, long version) {
            Class supportedEventType = getSupportedEventType();
            if (!eventTypeId.isAssignableFrom(supportedEventType))
{
                throw new UnsupportedOperationException();
            }
            Serializable eventId =
getEventId(eventStoreAggregateId, version);
            return (Event) getCurrentSession().get(eventTypeId,
eventId);
        }

        @Transactional(readOnly = true)
        @Override
        public Event getEvent(EventStoreAggregateId
eventStoreAggregateId, long version) {
            Serializable eventId =
getEventId(eventStoreAggregateId, version);
            return (Event)
getCurrentSession().get(getSupportedEventType(), eventId);
        }

        protected abstract Class getSupportedEventType();
        protected abstract Serializable

```

```
        getEventId(EventStoreAggregateId eventStoreAggregateId, long
version);
    }
```

---

為了實現聚合內部實體的事件的存儲與加載，我們的工具生成了**WheelEventDao**接口：

```
package org.dddml.templates.tests.domain.car;

import java.util.*;
import org.dddml.templates.tests.domain.*;

public interface WheelEventDao {
    void save(WheelEvent e);

    Iterable<WheelEvent> findByCarEventId(CarEventId
carEventId);
}
```

---

和**WheelStateDao**接口類似，可以認為這裡生成的**WheelEventDao**接口只是基於關係數據庫的事件存儲（Event Store）實現的一部分。如果不使用關係數據庫，它可能就不是必須的。實現**WheelEventDao**的**HibernateWheelEventDao**類的代碼比較簡單，這裡不再列出。

Car事件對象的**Hibernate**映射文件（文件**CarEvent.hbm.xml**）如下：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```

        "http://www.hibernate.org/dtd/hibernate-mapping-
3.0.dtd">

<hibernate-mapping
package="org.dddml.templates.tests.domain.car">
    <class name="AbstractCarEvent" table="CarEvents"
mutable="false" abstract="true">
        <composite-id name="carEventId"
class="org.dddml.templates.tests.domain.car.CarEventId">
            <key-property name="id">
                <column name="Id" length="50"/>
            </key-property>
            <key-property name="version"></key-property>
        </composite-id>
        <discriminator column="EventType" type="string"/>
        <property name="createdBy" column="CreatedBy"/>
        <property name="createdAt" column="CreatedAt"/>
        <property name="commandId" column="CommandId"/>
        <property name="commandType" column="CommandType"
length="50"/>

            <subclass name="AbstractCarEvent$CarClobEvent"
discriminator-value="CarClobEvent">
                <property name="LobText" column="LobText"/>
                <subclass
name="AbstractCarEvent$TireWheelPairsRotated" discriminator-
value="TireWheelPairsRotated"/>
                </subclass>

            <subclass
name="AbstractCarEvent$AbstractCarStateEvent"
abstract="true">
                <property name="description"></property>
                <property name="active"></property>

                <subclass
name="AbstractCarEvent$SimpleCarStateCreated" discriminator-
value="Created">
                    </subclass>
                    <subclass
name="AbstractCarEvent$SimpleCarStateMergePatched"
discriminator-value="MergePatched">
                        <property
name="isPropertyDescriptionRemoved"
column="IsPropertyDescriptionRemoved"/>

```

```
        <property name="isPropertyActiveRemoved"
column="IsPropertyActiveRemoved"/>
        </subclass>
        <subclass
name="AbstractCarEvent$SimpleCarStateDeleted" discriminator-
value="Deleted">
        </subclass>
        </subclass>
    </class>
</hibernate-mapping>
```

---

Wheel事件對象的Hibernate映射文件 ( 文件 WheelEvent.hbm.xml ) 如下：

---

```
<hibernate-mapping
package="org.dddml.templates.tests.domain.car">
    <class name="AbstractWheelEvent" table="WheelEvents"
mutable="false" abstract="true">
        <composite-id name="wheelEventId"
class="org.dddml.templates.tests.domain.car.WheelEventId">
            <key-property name="carId">
                <column name="CarWheelIdCarId" length="50"/>
            </key-property>
            <key-property name="wheelId">
                <column name="CarWheelIdWheelId"/>
            </key-property>
            <key-property name="carVersion"></key-property>
        </composite-id>
        <discriminator column="EventType" type="string"/>
        <property name="createdBy" column="CreatedBy"/>
        <property name="createdAt" column="CreatedAt"/>
        <property name="commandId" column="CommandId"/>

        <subclass
name="AbstractWheelEvent$AbstractWheelStateEvent"
abstract="true">
            <property name="active"></property>
            <property name="version" column="Version" not-
null="true"/>
            <subclass
name="AbstractWheelEvent$SimpleWheelStateCreated">
```

```
discriminator-value="Created">
    </subclass>
    <subclass
name="AbstractWheelEvent$SimpleWheelStateMergePatched"
discriminator-value="MergePatched">
    <property name="isPropertyActiveRemoved"
column="IsPropertyActiveRemoved"/>
    </subclass>
    <subclass
name="AbstractWheelEvent$SimpleWheelStateRemoved"
discriminator-value="Removed">
    </subclass>
    </subclass>
</class>
</hibernate-mapping>
```

---

### 3. 狀態的持久化

使用Hibernate實現CarStateRepository接口的  
HibernateCarStateRepository類的代碼如下：

---

```
package org.dddml.templates.tests.domain.car.hibernate;

import java.util.*;
import org.dddml.templates.tests.domain.*;
import org.hibernate.*;
import org.hibernate.Criteria;
import org.hibernate.criterion.*;
import org.dddml.templates.tests.domain.car.*;
import org.dddml.templates.tests.specialization.*;
import org.dddml.templates.tests.specialization.hibernate.*;
import
org.springframework.transaction.annotation.Transactional;

public class HibernateCarStateRepository implements
CarStateRepository {
    private SessionFactory sessionFactory;

    public SessionFactory getSessionFactory() {
        return this.sessionFactory;
```

```
    }

    public void setSessionFactory(SessionFactory
sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    protected Session getCurrentSession() {
        return this.sessionFactory.getCurrentSession();
    }

    @Transactional(readOnly = true)
    public CarState get(String id, boolean nullAllowed) {
        CarState.SqlCarState state = (CarState.SqlCarState)
getCurrentSession().get(AbstractCarState.SimpleCarState.clas
s, id);
        if (!nullAllowed && state == null) {
            state = new AbstractCarState.SimpleCarState();
            state.setId(id);
        }
        return state;
    }

    public void save(CarState state) {
        CarState s = state;
        if (getReadOnlyProxyGenerator() != null) {
            s = (CarState)
getReadOnlyProxyGenerator().getTarget(state);
        }
        if (s.getVersion() == null) {
            getCurrentSession().save(s);
        } else {
            getCurrentSession().update(s);
        }
        if (s instanceof Saveable) {
            Saveable saveable = (Saveable) s;
            saveable.save();
        }
        getCurrentSession().flush();
    }
}
```

---

## 實現WheelStateDao接口的 HibernateWheelStateDao類的代碼如下：

---

```
package org.dddml.templates.tests.domain.car.hibernate;

import org.dddml.templates.tests.domain.*;
import java.util.*;
import org.hibernate.*;
import org.hibernate.criterion.*;
import org.dddml.templates.tests.domain.car.*;
import org.dddml.templates.tests.specialization.*;
import
org.springframework.transaction.annotation.Transactional;

public class HibernateWheelStateDao implements WheelStateDao
{
    private SessionFactory sessionFactory;
    // 省略 getter/setter 方法

    protected Session getCurrentSession() {
        return this.sessionFactory.getCurrentSession();
    }

    @Transactional(readOnly = true)
    @Override
    public WheelState get(CarWheelId id, boolean
nullAllowed, CarState aggregateState) {
        Long aggregateVersion = aggregateState.getVersion();
        WheelState.SqlWheelState state =
(WheelState.SqlWheelState)
getCurrentSession().get(AbstractWheelState.SimpleWheelState.
class, id);
        if (!nullAllowed && state == null) {
            state = new
AbstractWheelState.SimpleWheelState();
            state.setCarWheelId(id);
        }
        if (state != null) {
            ((AbstractWheelState)
state).setCarState(aggregateState);
        }
        if (nullAllowed && aggregateVersion != null) {
```

```

        assertNoConcurrencyConflict(id.getCarId(), aggregateVersion);
    }
    return state;
}

private void assertNoConcurrencyConflict(String aggregateId, Long aggregateVersion) {
    Criteria crit =
getCurrentSession().createCriteria(AbstractCarState.SimpleCarState.class);
    crit.setProjection(Projections.property("version"));
    crit.add(Restrictions.eq("id", aggregateId));
    Long v = (Long) crit.uniqueResult();
    if (!aggregateVersion.equals(v)) {
        throw DomainError.named("concurrencyConflict",
"Conflict between new state version (%1$s) and old version (%2$s)", v, aggregateVersion);
    }
}

@Override
public void save(WheelState state) {
    WheelState s = state;
    if (s.getVersion() == null) {
        getCurrentSession().save(s);
    } else {
        getCurrentSession().update(s);
    }
    if (s instanceof Saveable) {
        Saveable saveable = (Saveable) s;
        saveable.save();
    }
}

@Transactional(readOnly = true)
@Override
public Iterable<WheelState> findByCarId(String carId,
CarState aggregateState) {
    Long aggregateVersion = aggregateState.getVersion();
    Criteria criteria =
getCurrentSession().createCriteria(
AbstractWheelState.SimpleWheelState.class);
    Junction partIdCondition =
Restrictions.conjunction()

```

```

        .add(Restrictions.eq("carWheelId.carId",
carId));
        List<WheelState> list =
criteria.add(partIdCondition).list();
        list.forEach(i -> ((AbstractWheelState)
i).setCarState(aggregateState));
        if (aggregateVersion != null) {
            assertNoConcurrencyConflict(carId,
aggregateVersion);
        }
        return list;
    }

    @Override
    public void delete(WheelState state) {
        WheelState s = state;
        if (s instanceof Saveable) {
            Saveable saveable = (Saveable) s;
            saveable.save();
        }
        getCurrentSession().delete(s);
    }
}

```

---

Car實體的狀態對象的Hibernate映射文件（文件 **CarState.hbm.xml** ）如下：

---

```

<hibernate-mapping
package="org.dddml.templates.tests.domain.car">
    <class
name="org.dddml.templates.tests.domain.car.AbstractCarState$"
SimpleCarState"
table="Cars"> <id name="id" length="50" column="Id">
        <generator class="assigned"/>
    </id>
        <version name="version" column="Version"
type="long"/>
        <property name="description"></property>
        <property name="createdBy"></property>
        <property name="updatedBy"></property>
        <property name="active"></property>

```

```
<property name="deleted"></property>
<property name="createdAt" column="CreatedAt"/>
<property name="updatedAt" column="UpdatedAt"/>
</class>
</hibernate-mapping>
```

---

Wheel實體的狀態對象的Hibernate映射文件（文件WheelState.hbm.xml）如下：

---

```
<hibernate-mapping
package="org.dddml.templates.tests.domain.car">
<class
name="org.dddml.templates.tests.domain.car.
AbstractWheelState$SimpleWheelState"
table="Wheels"> <composite-id name="carWheelId"
class="org.dddml.templates.tests.domain.car.CarWheelId">
    <key-property name="carId">
        <column name="CarWheelIdCarId" length="50"/>
    </key-property>
    <key-property name="wheelId">
        <column name="CarWheelIdWheelId"/>
    </key-property>
</composite-id>
<version name="version" column="Version"
type="long"/>
    <property name="createdBy"></property>
    <property name="updatedBy"></property>
    <property name="active"></property>
    <property name="deleted"></property>
    <property name="createdAt" column="CreatedAt"/>
    <property name="updatedAt" column="UpdatedAt"/>
</class>
</hibernate-mapping>
```

---

## 4.處理數據庫的名稱

我們可以在限界上下文的**DDML**配置文件內指定一個全局的數據庫命名規則，示例如下：

---

```
configuration:  
  # ...  
  databaseNamingConvention:  
  "SimpleUnderscoredNamingConvention"
```

---

這個名為**SimpleUnderscoredNamingConvention**的命名規則，其對名稱的處理邏輯借鑑自開源項目**Apache OFBiz**<sup>[1]</sup>的如下代碼：

---

```
public static String javaNameToDbName(String javaName) {  
    if (javaName == null) return null;  
    if (javaName.length() <= 0) return "";  
    StringBuilder dbName = new StringBuilder();  
  
    dbName.append(Character.toUpperCase(javaName.charAt(0)));  
    int namePos = 1;  
    while (namePos < javaName.length()) {  
        char curChar = javaName.charAt(namePos);  
        if (Character.isUpperCase(curChar))  
            dbName.append('_');  
        dbName.append(Character.toUpperCase(curChar));  
        namePos++;  
    }  
    return dbName.toString();  
}
```

---

也可以在聚合的定義內聲明實體的數據表命名，示例如下：

---

```
aggregates:
  Person:
    tableName: PERSON_T
    # ...
    properties:
      BirthDate:
        type: DateTime
    metadata:
      EventDatabaseTableName: PERSON_EVENT_T
```

---

我們的工具以這樣的優先級獲取實體相關對象的數據庫名稱：

- 如果在DDDM中指定了這些對象對應的數據表的名稱，那麼優先使用這些名稱。比如，上面DDDM示例代碼中的**Person**實體，它的狀態表名稱為**PERSON\_T**，事件表的名稱為**PERSON\_EVENT\_T**。
- 如果在DDDM中指定了限界上下文的全局數據庫命名規則，那麼，在實體的名稱上應用這個命名規則，得到它的狀態表的名稱；將實體的名稱拼接上“Event”後，應用這個命名規則，得到實體的事件表名稱。比如，對“Person”這個字符串應用**SimpleUnderscoredNamingConvention**命名規則進行轉換後，得到的結果是“PERSON”；對“PersonEvent”應用這個命名規則後，得到的結果是“PERSON\_EVENT”。
- 如果上面所說的關於數據庫名稱的設置在DDDM中都不存在，代碼生成工具就會使用內置的默認命名規則。即以實體名稱的複數形式命名實體的狀

態表，以實體的名稱拼接“Events”命名實體的事件表。

## 5. Hibernate與NHibernate的差異

在筆者製作的工具生成的Java版本的服務端代碼中，使用**Hibernate**來實現事件存儲以及聚合的狀態持久化時，存儲的第一個聚合事件的**Version**（版本號）是**-1**。而在使用**NHibernate**的.NET版本的服務端，數據庫中保存的第一個聚合事件的**Version**是**0**。

這是因為在使用**Hibernate**支持的樂觀鎖特性時，一個實體的狀態對象被創建出來之後，它的**Version**屬性的默認值是**null**（如果**Version**類型是**Long/Integer**）或**0**（如果**Version**類型是**long/int**）。第一次把實體的狀態對象保存到數據庫之後，它在數據庫中的**Version**就是**0**。此後每次修改、保存實體的狀態，它在數據庫中的**Version**都會加**1**。

而在使用**NHibernate**的情況下，一個實體的狀態對象被創建出來之後，它的**Version**屬性的默認值是**0**（如果**Version**的類型是**long**或**int**）。第一次保存狀態對象到數據庫，它在數據庫中**Version**的值會變成**1**。

其實在這個問題上，我認為**NHibernate**的處理更合理、更具有一致性。但不管是使用**Hibernate**還是使用**NHibernate**，我們希望保存在事件存儲/數據庫中的某個聚合的最後一個事件的**Version**總是比保存在數據

庫中的聚合根的最新狀態的**Version**小1，那麼，如果使用的是**Hibernate**，就讓聚合第一個事件的**Version**屬性的值是-1。

[1] 見<https://ofbiz.apache.org/>。

## 11.1.5 使用Validation框架

正如前文所言，在將概念顯現出來時，約束（Constraints）是非常有用的概念。我們可以使用Validation框架來確認實體的狀態在修改後仍然滿足那些重要的約束。如果已經選擇了將實體的最新狀態存儲到數據庫，那麼可以在將實體的狀態存儲到數據庫之前執行確認。

### 1. 使用Hibernate Validator

對於Java項目，在resources目錄下的“META-INF/validation.xml”的文件形式如下：

---

```
<validation-config
  xmlns="http://jboss.org/xml/ns/javax/validation/configuration"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  "http://jboss.org/xml/ns/javax/validation/configuration"
  version="1.1">

  <constraint-mapping>META-INF/validation/state-
constraints.xml
  </constraint-mapping>
  <property
  name="hibernate.validator.fail_fast">false</property>
</validation-config>
```

---

在resources目錄下的META-INF/validation/state-constraints.xml文件是由DDDML工具生成的。對於第7章的“約束”一節中使用的Locator聚合示例來說，工具可能會生成如下代碼（文件state-constraints.xml）：

---

```
<constraint-mappings
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

    xsi:schemaLocation=
    "http://jboss.org/xml/ns/javax/validation/mapping
validation-mapping-1.1.xsd"

    xmlns="http://jboss.org/xml/ns/javax/validation/mapping"
    version="1.1">

    <default-
package>org.dddml.templates.tests.domain</default-package>
    <bean
    class="org.dddml.templates.tests.domain.locator.AbstractLoca-
torState">
        <field name="locatorId">
            <constraint
annotation="javax.validation.constraints.Pattern">
                <element name="regexp">^ [0-9] [A-Za-z0-9-] *
            </element>
            </constraint>
        </field>
    </bean>
</constraint-mappings>
```

---

## 2. 使用NHibernate Validator

在.NET中，可以考慮使用NHibernate Validator。可以在Spring.NET的XML配置文件中啟用它，示例如下：

---

```
<object id="NHibernateSessionFactory" type="Spring.Data.NHibernate.LocalSessionFactoryObject, Spring.Data.NHibernate4">
    <!-- ... -->
    <property name="EventListeners">
        <dictionary>
            <entry key="PreUpdate">
                <object type="NHibernate.Validator.Event.ValidatePreUpdateEventListener, NHibernate.Validator" />
            </entry>
            <entry key="PreInsert">
                <object type="NHibernate.Validator.Event.ValidatePreInsertEventListener, NHibernate.Validator" />
            </entry>
        </dictionary>
    </property>
    <!-- ... -->
</object>
```

---

工具生成的Validation映射文件（文件 **LocatorState.nhv.xml**）如下：

---

```
<?xml version="1.0" encoding="utf-8" ?>
<nhv-mapping xmlns="urn:nhibernate-validator-1.0">
    <class
name="Dddml.T4.Templates.Tests.Generated.Domain.Locator.LocatorState, Dddml.T4.Templates.Tests">
        <property name="LocatorId">
            <pattern regex="^ [0-9] [A-Za-z0-9-]*" message="Must match numericDashAlphabetic"/>
        </property>
    </class>
</nhv-mapping>
```

---

### 11.1.6 保證靜態方法與模型同步更新

在前面生成的代碼中可以看到，為了實現Car實體的Rotate方法，我們使用反射調用了位於org.dddml.templates.tests.domain.car包內RotateLogic類中的兩個靜態方法（verify與mutate）。這種做法帶來一個問題，如果在DDML文檔中，我們對Rotate方法的定義進行了修改，怎麼保證開發人員“手寫”的RotateLogic類也做了相應的修改？

還應該考慮這種情況，有時我們並沒有修改模型中方法的簽名（一般來說，方法的簽名不包括參數的名稱），也就是說方法名、參數列表的長度、每個參數的類型都沒有被修改，但是我們修改了方法中某個參數的名稱。修改參數的名稱有可能意味著參數的含義已經出現了變化，客戶端可能會改變這個參數的傳入內容，我們需要讓實現這個方法業務邏輯的開發人員注意到這一點，由他們來判斷自己編寫的代碼是否需要修改。

所以有時需要對開發人員編寫的靜態方法的簽名、參數的名稱進行檢查。代碼生成工具會生成類似如下形式的檢查代碼：

---

```
public class StaticMethodConstraints {  
    public static void
```

```
assertStaticVerificationAndMutationMethods() {
    ReflectUtils.assertStaticMethodIfExists(
        "org.dddml.templates.tests.domain.car.RotateLogic",
        "verify",
        new Class[] {CarState.class,
TireWheelIdPair[].class, VerificationContext.class},
        new String[] {"_", "tireWheelIdPairs"}
    ); // 下劃線“_”表示可以匹配任何名稱
    ReflectUtils.assertStaticMethodIfExists(
        "org.dddml.templates.tests.domain.car.RotateLogic",
        "mutate",
        new Class[] {CarState.class,
TireWheelIdPair[].class, MutationContext.class},
        new String[] {"_", "tireWheelIdPairs"}
    );
}
```

---

可以在必要的時候（比如應用啟動的時候），調用**assertStaticVerificationAndMutation-Methods**方法來保證開發人員編寫的靜態方法已經同步更新。



注意在編譯**RotateLogic**這樣的Java類時，需要使用編譯器參數（Compiler Argument）- **parameters**，使得源代碼中方法參數的名稱能保留在Java字節碼中，這樣才能夠在運行時獲取到它們。

這裡用到的工具類**ReflectUtils**的代碼大致如下：

---

```
package org.dddml.templates.tests.specialization;

import java.lang.reflect.*;
import java.util.*;
```

```
public class ReflectUtils {
    public static Method
assertStaticMethodIfClassExists(String className, String
methodName,
                                Class<?>[] parameterTypes,
                                String[] parameterNames) {
    return assertStaticMethod(className, methodName,
parameterTypes, parameterNames, false);
}

    private static Method assertStaticMethod(String
className, String methodName,
                                Class<?>[] parameterTypes,
                                String[] parameterNames, boolean
throwOnClassNotFound) {
    Class clazz = null;
    try {
        clazz = Class.forName(className);
    } catch (ClassNotFoundException e) {
        if (throwOnClassNotFound) { throw new
RuntimeException(e); }
    }
    try {
        if (clazz != null) {
            return assertStaticMethod(clazz, methodName,
parameterTypes, parameterNames);
        } else { return null; }
    } catch (NoSuchMethodException e) { throw new
RuntimeException(e); }
}

    private static Method assertStaticMethod(Class clazz,
String methodName, Class<?>[] parameterTypes,
String[] parameterNames) throws
NoSuchMethodException {
    Method method = clazz.getDeclaredMethod(methodName,
parameterTypes);
    if (!Modifier.isStatic(method.getModifiers())) {
        throw new RuntimeException(String.format(
"Method: %1$s.%2$s, MUST be static.",
                                clazz.getName(), methodName));
    }
    if (parameterNames != null) {
        for (int i = 0; i < parameterNames.length; i++)
```

```
{  
    if ("_".equals(parameterNames[i])) {  
        continue;  
    }  
    if (i < method.getParameters().length) {  
        if  
(!parameterNames[i].equals(method.getParameters()  
[i].getName())) {  
            throw new  
RuntimeException(String.format(  
                "Method: %1$s.%2$s, parameters[%3$s] MUST be  
named as \"%4$s\".",  
                clazz.getName(), methodName, i,  
                parameterNames[i]);  
        }  
    }  
}  
return method;  
}  
}  
}
```

---

### 11.1.7 不使用事件溯源

使用了事件溯源之後，生成的代碼似乎有點煩瑣，其實也可以聲明一個聚合不使用數據溯源（ES），示例如下：

```
aggregates:
  JobLevel:
    id:
      name: JobLevelId
      type: id-ne
    properties:
      Description:
        name: Description
        type: very-long
    metadata:
      AbsolutelyNoEventSourcing: true
```

在上面的DDML代碼的最後一行，已把這個**JobLevel**聚合聲明為不使用事件溯源，於是，DDML工具為這個聚合生成的服務端Java代碼就不再包含**Event**相關的類，只剩下如下Java文件（這裡不包括工具可能會生成的RESTful API層的代碼）：

- **JobLevelApplicationService.java**
- **JobLevelCommand.java**
- **JobLevelState.java**

- JobLevelStateQueryRepository.java
- JobLevelStateRepository.java
- JobLevelStateDto.java
- AbstractJobLevelApplicationService.java
- AbstractJobLevelCommandDto.java
- AbstractJobLevelState.java
- CreateOrMergePatchJobLevelDto.java
- DeleteJobLevelDto.java
- HibernateJobLevelStateQueryRepository.java
- HibernateJobLevelStateRepository.java

其中後綴為“Dto”的那些類，被用作服務端 RESTful API方法的參數類型或返回值類型。工具生成的Java Client SDK中也使用了它們。也就是說，在生成Java Client SDK時，只用到了 AbstractJobLevelCommandDto類和它的兩個子類 CreateOrMergePatchJobLevelDto、DeleteJobLevelDto，以及JobLevelStateDto類。

## 11.2 Override聚合對象的方法

在代碼中實現DDDML中定義的實體方法的方式之一，是重寫（Override）工具生成的聚合對象的相應方法。

例如，我們開發過的一個CRM應用，在DDDML文檔中給聚合根Lead（線索）定義了一個Discard方法：

---

```
aggregates:
  Lead:
    id:
      name: LeadId
      type: id
    # ...
    methods:
      # 丟棄線索
      Discard:
        parameters:
          LeadStatusId:
            type: id
            referenceType: LeadStatus
```

---



提

示定義在實體（聚合根或非聚合根的實體）中的方法，表示這個方法只會改變一個聚合實例的狀態（一個聚合根的實例以及生命週期由其控制的其他實體的實例可以看作一個整體，我們稱之為“聚合實例”）。如果一個方法要改變多個聚合實例的狀態，那麼應該被定義為領域服務或領域服務的變體——聚合根的非實例方法。

如果我們告訴DDDDML工具在生成Java代碼時不要生成聚合對象的**discard**方法的默認實現，那麼在源代碼文件中會有一個直接拋出異常的**discard**方法（文件**AbstractLeadAggregate.java**）：

---

```
public class AbstractLeadAggregate {
    public static class SimpleLeadAggregate extends
AbstractLeadAggregate {
        public SimpleLeadAggregate(LeadState state) {
            super(state);
        }

        @Override
        public void discard(String leadStatusId, String
commandId, String requesterId) {
            throw new UnsupportedOperationException();
        }
    }
}
```

---

下一步打算通過擴展工具生成的**SimpleLeadAggregate**類去真正實現這個方法。我們可以增加一個擴展自**SimpleLeadApplicationService**的類

LeadApplicationServiceImpl，並在裡面實現一個內部類LeadAggregateImpl，示例如下：

---

```
public class LeadApplicationServiceImpl extends
AbstractLeadApplicationService.SimpleLeadApplicationService
{
    // 省略部分代碼
    @Override
    public LeadAggregate getLeadAggregate(LeadState state) {
        return new LeadAggregateImpl(state);
    }

    public static class LeadAggregateImpl extends
AbstractLeadAggregate.SimpleLeadAggregate{
        // 省略部分代碼
        @Override
        public void discard(String leadStatusId, String
commandId, String requesterId) {
            // 生成事件
            LeadStateEvent.LeadStateMergePatched e =
newLeadStateMergePatched(
commandId, requesterId);
            e.setActive(false);
            e.setLeadStatusId(leadStatusId);
            // 應用事件
            apply(e);
        }
    }
}
```

---

這幾行代碼在IDE的幫助下可以很快完成。

在上面展示的代碼中，方法discard的實現邏輯是使用已經生成了代碼的LeadState-MergePatched事件對象，而不是創建更多的事件類型，來完成Lead實體的狀態修改。

默認情況下，我們還會給這個Discard方法生成 RESTful API。這樣就可以發送HTTP PUT請求到這樣的一個URL中去調用它：

---

```
{BASE_URL}/Leads/{leadId}/_commands/Discard
```

---

只要客戶端在請求的消息體中包含了合適的 Command ID，那麼對Discard方法的調用是幂等的，所以這裡支持PUT方法（HTTP PUT應該是幂等的）。需要說明的是，這個示例中URL的風格確實“不怎麼 RESTful”，但是在DDDM文檔沒有提供更多設置信息的情況下，使用這樣的URL也是可以接受的。

以上演示的是使用Java語言的做法，我們通過 **Override**基類的方法來擴展DDDM工具生成的Java代碼。說到這裡，不由得讓人想念C#的“partial class”特性。對於C#來說，我們不需要Override（重寫）工具生成的類，只需要在一個獨立的C#代碼文件中直接實現ILeadAggregate接口的Discard方法即可，示例如下：

---

```
// ...
public partial class LeadAggregate : ILeadAggregate
{
    public void Discard(string leadStatusId, string
commandId, string requesterId)
    {
        //業務邏輯實現
    }
}
```



## 11.3 處理繼承

正如前文所述，筆者製作的DDML工具在生成Java或C#的服務端代碼時支持如下三種繼承映射策略：Table per class hierarchy ( TPCH ) 、Table per concrete class ( TPCC ) 、Table per subclass:using a discriminator ( TPS ) 。

如果在代碼中使用了ORM框架，支持這三種映射策略需要生成的Java或C#代碼很可能大同小異，主要差異體現在ORM的映射代碼上。下面以我們的工具生成的Hibernate XML映射文件來舉例說明。

### 11.3.1 TPCH

在我們開發的一個網上商城應用中，曾把產品組（ProductGroup）建模為產品（Product）實體的子類型。這樣，產品組就可以像產品一樣關聯產品圖片、供應商、價格等信息。這裡的繼承映射策略使用的是TPCH。描述模型的DDDML代碼大致如下：

```
aggregates:
  Product:
    id:
      name: ProductId
      type: id-ne
    properties:
      ProductTypeId:
        type: id
        # ...
    inheritanceMappingStrategy: tpch
    discriminator: ProductTypeId
    discriminatorValue: "PRODUCT"
    subtypes:
      ProductGroup:
        discriminatorValue: "PRODUCT_GROUP"
```

DDDML工具為產品的狀態對象生成的HBM文件（文件ProductState.hbm.xml）大致如下：

```
<hibernate-mapping package="org.dddml.pmall.domain.product">
  <class
  name="org.dddml.pmall.domain.product.AbstractProductState"
  table="PRODUCT" abstract="true">
    <id name="productId" length="20"
    column="PRODUCT_ID">
```

```
        <generator class="assigned"/>
    </id>
    <discriminator column="PRODUCT_TYPE_ID"
type="string"/>
        <version name="version" column="VERSION"
type="long"/>
        <property name="manufacturerPartyId">
            <column name="MANUFACTURER_PARTY_ID" sql-
type="VARCHAR(20)"/>
        </property>
        <!-- 更多屬性省略-->
        <subclass name=
"org.dddml.pmall.domain.product.
AbstractProductState$SimpleProductState"
            discriminator-value="PRODUCT">
        </subclass>

        <subclass name=

"org.dddml.pmall.domain.product.AbstractProductGroupState"
abstract="true">
<subclass
name="org.dddml.pmall.domain.product.AbstractProductGroupSta-
te$SimpleProductGroupState"
            discriminator-value="PRODUCT_GROUP"
abstract="false">
            </subclass>
        </subclass>

    </class>
</hibernate-mapping>
```

---

### 11.3.2 TPCC

在我們開發的一個CRM系統中，對Party ( Party是參與業務流程的業務實體 ) 聚合採用如下方式建模：

- Person ( 個人 ) 是Party的子類型。
- Organization ( 組織 ) 是Party的子類型，LegalOrganization ( 法人組織 ) 是Organization的子類型。
- Company ( 公司 ) 是LegalOrganization的子類型，InformalOrganization ( 非正式組織 ) 是Organization的子類型。
- Family ( 家庭 ) 是InformalOrganization的子類型。

這裡的繼承映射策略選擇使用TPCC。描述模型的DDDMIL代碼這裡不再列出。

DDDMIL工具為Party的狀態對象生成的HBM文件 ( 文件PartyState.hbm.xml ) 大致如下：

---

```
<hibernate-mapping package="org.dddml.crm.domain.party">
  <class
    name="org.dddml.crm.domain.party.AbstractPartyState$SimplePartyState"
    table="PARTY">
    <id name="partyId" length="20" column="PARTY_ID">
```

```

        <generator class="assigned"/>
    </id>
    <version name="version" column="VERSION"
type="long"/>
        <property name="externalId">
            <column name="EXTERNAL_ID" sql-
type="VARCHAR(20)"/>
        </property>

        <union-subclass name=
"org.dddml.crm.domain.party.
AbstractPersonState$SimplePersonState"
table="PERSON"
            abstract="false">
            <property name="salutation">
                <column name="SALUTATION" sql-
type="VARCHAR(100)"/>
            </property>
            <property name="firstName">
                <column name="FIRST_NAME" sql-
type="VARCHAR(100)"/>
            </property>
            <property name="middleName">
                <column name="MIDDLE_NAME" sql-
type="VARCHAR(100)"/>
            </property>
        </union-subclass>

        <union-subclass name=
"org.dddml.crm.domain.party.AbstractOrganizationState$Simple
OrganizationState"
            table="ORGANIZATION"
            abstract="false">
            <property name="organizationName">
                <column name="ORGANIZATION_NAME" sql-
type="VARCHAR(250)"/>
            </property>
            <union-subclass name=
"org.dddml.crm.domain.party.AbstractLegalOrganizationState$SimpleLegalOrganizationState"
                table="LEGAL_ORGANIZATION"
                abstract="false">
                <property name="taxIdNum">
                    <column name="TAX_ID_NUM"/>
                </property>
            </union-subclass>
        </union-subclass>
    </version>

```

```
        <union-subclass name=
"org.dddml.crm.domain.party.AbstractCompanyState$SimpleCompa
nyState" table="COMPANY"
                abstract="false">
            </union-subclass>
        </union-subclass>

        <union-subclass name=
"org.dddml.crm.domain.party.AbstractInformalOrganizationStat
e"
                table="INFORMAL_ORGANIZATION"
                abstract="true">
            <union-subclass name=
"org.dddml.crm.domain.party.AbstractFamilyState$SimpleFamily
State" table="FAMILY"
                abstract="false">
                <property name="familyName">
                    <column name="FAMILY_NAME" sql-
type="VARCHAR(60)"/>
                </property>
            </union-subclass>
        </union-subclass>
    </union-subclass>

</class>
</hibernate-mapping>
```

---

### 11.3.3 TPS

在我們開發的一個CRM系統中，曾把投資項目的潛在投資者在“項目頁面”下的留言

( **InvestmentComment** ) 建模為銷售線索 ( **Lead** ) 的子類型，這裡使用的是TPS繼承映射策略。因為 **DDDML**比較簡單，這裡不再列出。

**DDDML**工具為**Lead**的狀態對象生成的**HBM**映射文件 ( 文件**LeadState.hbm.xml** ) 大致如下：

---

```
<hibernate-mapping package="org.dddml.crm.domain.lead">
  <class
    name="org.dddml.crm.domain.lead.AbstractLeadState"
    table="LEAD" abstract="true">
    <id name="leadId" length="60" column="LEAD_ID">
      <generator class="assigned"/>
    </id>
    <discriminator column="LEAD_TYPE_ID" type="string"/>
    <version name="version" column="VERSION"
      type="long"/>
    <property name="contactMechId">
      <column name="CONTACT_MECH_ID" sql-
        type="VARCHAR(20)"/>
    </property>
    <property name="leadStatusId">
      <column name="LEAD_STATUS_ID" sql-
        type="VARCHAR(60)"/>
    </property>
    <!-- 更多屬性省略-->
    <subclass name=
      "org.dddml.crm.domain.lead.AbstractLeadState$SimpleLeadState
      " discriminator-value="Lead">
    </subclass>
```

```
    <subclass
name="org.dddml.crm.domain.lead.AbstractInvestmentCommentState" abstract="true">
        <join table="INVESTMENT_COMMENT">
            <key column="LEAD_ID"/>
            <property name="intendedInvestmentAmount">
                <column
name="INTENDED_INVESTMENT_AMOUNT" sql-type="DECIMAL(18,2)"/>
                </property>
            </join>
            <subclass name=
"org.dddml.crm.domain.lead.AbstractInvestmentCommentState$Si
mpleInvestmentCommentState"
                    discriminator-value="InvComment">
                </subclass>
        </subclass>
    </class>
</hibernate-mapping>
```

---

## 11.4 處理模式

第9章展示了在DDML文檔中如何定義賬目（使用關鍵字**accounts**）和狀態機（使用關鍵字**stateMachine**），接下來看看DDML代碼生成工具可以如何利用這些模式信息。

## 11.4.1 處理賬務模式

第9章曾展示一個描述InventoryItem（庫存單元）聚合的DDDML文檔。這裡繼續以這個聚合為例，看看DDDML代碼工具如何為它生成賬務模式相關的代碼。

默認情況下，工具生成的代碼並不會把賬目的當前值（餘額）屬性當作派生屬性來處理。

### 1.生成DDL

使用DDDML代碼工具為庫存單元生成的DDL如下（假設我們使用的目標數據庫是MySQL）：

---

```
CREATE TABLE 'InventoryItems' (
  'ProductId' varchar(60) NOT NULL,
  'LocatorId' varchar(50) NOT NULL,
  'AttributeSetInstanceId' varchar(50) NOT NULL,
  'Version' bigint(20) NOT NULL,
  'OnHandQuantity' decimal(18,6) DEFAULT NULL, #在庫數量
  'InTransitQuantity' decimal(18,6) DEFAULT NULL, #在途數量
  'ReservedQuantity' decimal(18,6) DEFAULT NULL, #保留數量
  'OccupiedQuantity' decimal(18,6) DEFAULT NULL, #佔用數量
  'VirtualQuantity' decimal(18,6) DEFAULT NULL, #虛擬數量
  'createdBy' varchar(255) DEFAULT NULL,
  'updatedBy' varchar(255) DEFAULT NULL,
  'CreatedAt' datetime DEFAULT NULL,
  'UpdatedAt' datetime DEFAULT NULL,
  PRIMARY KEY
  ('ProductId', 'LocatorId', 'AttributeSetInstanceId')
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

---

它的主鍵包含3列。

- **ProductId**：產品Id。
- **LocatorId**：貨位Id。
- **AttributeSetInstanceId**：屬性集實例Id。庫存單元可能需要一些描述它的擴展屬性信息，比如序列號、批次等，這些信息保存在**AttributeSetInstanceId**實體裡。

工具為庫存單元條目生成的DDL如下：

---

```
CREATE TABLE 'InventoryItemEntries' (
    'ProductId' varchar(60) NOT NULL,
    'LocatorId' varchar(50) NOT NULL,
    'AttributeSetInstanceId' varchar(50) NOT NULL,
    'entrySeqId' bigint(20) NOT NULL,
    'Version' bigint(20) NOT NULL,
    'OnHandQuantity' decimal(18, 6) DEFAULT NULL, #在庫數量
    'InTransitQuantity' decimal(18, 6) DEFAULT NULL, #在途數量
    'ReservedQuantity' decimal(18, 6) DEFAULT NULL, #保留數量
    'OccupiedQuantity' decimal(18, 6) DEFAULT NULL, #佔用數量
    'VirtualQuantity' decimal(18, 6) DEFAULT NULL, #虛擬數量
    'sourceDocumentTypeId' varchar(255) NOT NULL,
    'sourceDocumentNumber' varchar(255) NOT NULL,
    'sourceLineNumber' varchar(255) DEFAULT NULL,
    'sourceLineSubSeqId' int(11) DEFAULT NULL,
    'OccurredAt' datetime NOT NULL,
    'createdBy' varchar(255) DEFAULT NULL,
    'updatedBy' varchar(255) DEFAULT NULL,
    'CreatedAt' datetime DEFAULT NULL,
    'UpdatedAt' datetime DEFAULT NULL,
    'CommandId' varchar(255) DEFAULT NULL,
    PRIMARY KEY
    ('ProductId', 'LocatorId', 'AttributeSetInstanceId', 'entrySeqId'
    ),
```

```
UNIQUE KEY 'UK_8rt76iv8h9j6vel1hnw1pcahk'  
( 'sourceDocumentTypeId', 'sourceDocument  
Number', 'sourceLineNumber', 'sourceLineSubSeqId' )  
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
```

---

它的主鍵相比庫存單元表多了一列，即**entrySeqId**（條目序號）。這個表中的數量列（**OnHandQuantity**、**InTransitQuantity**等）都是在庫存單元表中相應數量的變化值（變化值有可能是正數也有可能是負數）。

因為在**DDML**中聲明瞭唯一約束，所以這個條目表中存在一個唯一索引，它由以下列組成：

·**sourceDocumentTypeId**：源單據類型**ID**。引起庫存變化的源單據的類型**ID**，它的值可能是生產單（**Production**）、盤點單（**PhysicalInventory**）、入庫出庫單（**InOut**）、移動單（**Movement**）等。

·**sourceDocumentNumber**：源單據號。比如生產單號、盤點單號、入庫出庫單號等。

·**sourceLineNumber**：源單據行號。引起庫存變化的單據中的行項序號。比如生產單的行號。

·**sourceLineSubSeqId**：源單據行的“子序號”。

為什麼需要在唯一約束中設置**sourceLineSubSeqId**這一列？這是因為一個單據有可能在完成後還會被反轉，在完成單據或反轉

( Reverse ) 單據時都需要觸發庫存數量的更新。比如，使用出庫單 ( InOut ) 執行出庫作業，如果在確認 ( 完成 ) 出庫後才發現操作有誤，這時可能需要執行反轉操作 ( 只允許反轉一次 )。那麼，在確認出庫的時候，這個 `sourceLineSubSeqId` 可能會記為 0；在“反轉”的時候，可能會記為 1，如表 11-1 所示。

表 11-1 一個出庫單在“完成後反轉”產生的條目記錄

...	source DocumentTypeId	source DocumentNumber	source LineNumber	source LineSubSeqId
...	IN_OUT	OUT001234	PRD_00001	0
...	IN_OUT	OUT001234	PRD_00001	1

顯然，如果那個唯一約束中只有前三列，沒有 `sourceLineSubSeqId` 這一列，那麼反轉出庫單的時候，就會因為違反了唯一約束導致條目記錄插入失敗。

## 2. 處理庫存單元的命令對象

默認情況下，DDDML 代碼生成工具可以為聚合生成 CRUD 代碼。比如，對於上面所說的 `InventoryItem` 聚合，我們的工具可能會生成用於“創建庫存單元”的 `CreateInventoryItem` 命令對象 ( 接口以及實現類 )，以及用於修改庫存單元的 `MergePatchInventoryItem` 命令對象。

以CreateInventoryItem這個命令對象為例，因為在DDML中我們將InventoryItem的OnHandQuantity屬性聲明為一個“賬目”，所以在CreateInventoryItem中不會存在名字叫作onHandQuantity的屬性。客戶端不能通過設置發送給服務端的那個實體的onHandQuantity屬性去修改InventoryItemState（表示庫存單元狀態的對象）的onHandQuantity屬性。客戶端只能發送CreateInventoryItemEntry命令，並通過創建庫存單元條目間接地改變它（InventoryItemState.onHandQuantity屬性）。

我們可以看看為InventoryItem生成的聚合對象的實現類AbstractInventoryItemAggregate有何特別之處（Java代碼）：

---

```
package org.dddml.wms.domain.inventoryitem;

import java.util.*;
import java.math.BigDecimal;
import org.dddml.wms.domain.*;
import org.dddml.wms.specialization.*;

public abstract class AbstractInventoryItemAggregate extends
AbstractAggregate implements InventoryItemAggregate {
    // ...
    protected InventoryItemEvent
map(InventoryItemCommand.CreateInventoryItem c) {
    InventoryItemId stateEventId = new
InventoryItemId(c.getInventoryItemId(), c.getVersion());
    InventoryItemEvent.InventoryItemStateCreated e =
newInventoryItemStateCreated(stateEventId);
    ((AbstractInventoryItemEvent)
e).setCommandId(c.getCommandId());
    e.setCreatedBy(c.getRequesterId());
    e.setCreatedAt((java.util.Date)
```

```

ApplicationContext.current.getTimestampService().now(java.util.Date.class));
        // 注意這裡對“在庫數量”這個賬目的處理：
        BigDecimal onHandQuantity = BigDecimal.ZERO;
        // 省略其他數量的處理代碼
        Long version = c.getVersion();
        for
        (InventoryItemEntryCommand.CreateInventoryItemEntry
        innerCommand : c.getCreateInventoryItemEntryCommands()) {
            throwOnInconsistentCommands(c, innerCommand);

        InventoryItemEntryEvent.InventoryItemEntryStateCreated
        innerEvent = mapCreate(innerCommand, c, version, this.state);
            e.addInventoryItemEntryEvent(innerEvent);

            onHandQuantity =
        onHandQuantity.add(innerEvent.getOnHandQuantity() != null ?
        innerEvent.getOnHandQuantity() : BigDecimal.ZERO);
            // 省略其他數量的處理代碼
        }
        e.setOnHandQuantity(onHandQuantity);
        // 省略其他數量的處理代碼
        return e;
    }

    protected InventoryItemEvent map(InventoryItemCommand.
MergePatchInventoryItem c) {
    InventoryItemEventId stateEventId = new
InventoryItemEventId(c.getInventoryItemId(), c.getVersion());
    InventoryItemEvent.InventoryItemStateMergePatched e =
newInventoryItemStateMergePatched(stateEventId);
        ((AbstractInventoryItemEvent)
e).setCommandId(c.getCommandId());
        e.setCreatedBy(c.getRequesterId());
        e.setCreatedAt((java.util.Date)
ApplicationContext.current.getTimestampService().now(java.util.Date.class));

        BigDecimal onHandQuantity =
this.state.getOnHandQuantity();
        // 省略其他數量的處理代碼
        Long version = c.getVersion();
        for (InventoryItemEntryCommand innerCommand : c.
getInventoryItemEntryCommands()) {
            throwOnInconsistentCommands(c, innerCommand);

```

```
        InventoryItemEntryEvent innerEvent =
map(innerCommand, c, version, this.state);
        e.addInventoryItemEntryEvent(innerEvent);
        if (!(innerEvent instanceof
InventoryItemEntryEvent.
InventoryItemEntryStateCreated)) {
            continue;
        }

InventoryItemEntryEvent.InventoryItemEntryStateCreated
entryCreated =
(InventoryItemEntryEvent.InventoryItemEntryStateCreated)
innerEvent;
        onHandQuantity =
onHandQuantity.add(entryCreated.getOnHandQuantity() != null ?
entryCreated.getOnHandQuantity() : BigDecimal.ZERO);
        // 省略其他數量的處理代碼
    }
    e.setOnHandQuantity(onHandQuantity);
    // 省略其他數量的處理代碼
    return e;
}
//...
}
```

---

開發人員使用這些生成的代碼，很容易保證一個賬目的當前數量（餘額）與它的條目的合計（Sum）數量總是一致的。

### 3. 確認約束

賬務模式要求賬目與條目之間滿足如下約束：

賬目（Account）的餘額=賬目的所有條目（Entries）的數量的合計（Sum）。

即不應該存在條目的合計數量不等於餘額的賬目的情況。以前面的庫存單元為例，我們可以隨時執行以下SQL對約束進行確認，如果沒有意外，這個SQL查詢的結果應該總是為0：

---

```
select count(*) from
(
SELECT
    i.ProductId,
    i.LocatorId,
    i.AttributeSetInstanceId,
    ifnull(i.OnHandQuantity, 0) as OnHandQuantity,
    ifnull((select sum(e.OnHandQuantity)
            FROM InventoryItemEntries e # 條目表
           where e.ProductId = i.ProductId
                 and e.LocatorId = i.LocatorId
                 and e.AttributeSetInstanceId =
i.AttributeSetInstanceId
            ), 0) as SumOfEntryOnHandQuantity
FROM InventoryItems i # 賬目表
) a
where OnHandQuantity != a.SumOfEntryOnHandQuantity
;
```

---

## 11.4.2 處理狀態機模式

第9章曾展示一個描述InOut（入庫/出庫單）聚合的DDDML文檔。這裡繼續以這個聚合為例，看看DDDML工具如何為它生成狀態機模式相關的代碼。

與賬務模式類似，工具生成的命令對象CreateInOut、MergePatchInOut中並不存在DocumentStatusId屬性。也就是說，只有“觸發”與InOut實體的DocumentStatusId屬性關聯的那個狀態機執行轉換，才能間接地改變InOut的狀態對象的DocumentStatusId屬性的值。

工具生成的狀態機的處理邏輯主要位於InOut的聚合對象的實現類AbstractInOut-Aggregate中（Java代碼），示例如下：

---

```
import org.dddml.wms.domain.*;
import org.dddml.wms.specialization.*;

public abstract class AbstractInOutAggregate extends
AbstractAggregate implements InOutAggregate {
    // ...

    protected InOutEvent map(InOutCommand.CreateInOut c) {
        InOutEventId stateEventId = new
        InOutEventId(c.getDocumentNumber(), c.getVersion());
        InOutEvent.InOutStateCreated e =
        newInOutStateCreated(stateEventId);
        newInOutDocumentActionCommandAndExecute(c, state,
        e);
        // ...
    }
}
```

```

        e.setDescription(c.getDescription());
        // ...
        return e;
    }

    protected InOutEvent map(InOutCommand.MergePatchInOut c)
{
    InOutEventId stateEventId = new
InOutEventId(c.getDocumentNumber(), c.getVersion());
    InOutEvent.InOutStateMergePatched e =
newInOutStateMergePatched(
stateEventId);
    if (c.getDocumentAction() != null)
        newInOutDocumentActionCommandAndExecute(c,
state, e);
    // ...
    e.setDescription(c.getDescription());

    e.setIsPropertyDescriptionRemoved(c.getIsPropertyDescription
Removed());
    // ...
    return e;
}
// ...

protected void
newInOutDocumentActionCommandAndExecute(InOutCommand.MergePa
tchInOut c, InOutState s, InOutEvent.InOutStateMergePatched
e) {
    PropertyCommandHandler<String, String>
pCommandHandler =
this.getInOutDocumentActionCommandHandler();
    String pCmdContent = c.getDocumentAction();
    PropertyCommand<String, String> pCmd = new
AbstractPropertyCommand.SimplePropertyCommand<String,
String>();
    pCmd.setContent(pCmdContent);
    pCmd.setStateGetter(() -> s.getDocumentStatusId());
    pCmd.setStateSetter(p -> e.setDocumentStatusId(p));
    pCmd.setOuterCommandType(CommandType.MERGE_PATCH);
    pCmd.setContext(getContext());
    pCommandHandler.execute(pCmd);
}

protected void

```

```

newInOutDocumentActionCommandAndExecute(InOutCommand.CreateI
nOut c, InOutState s, InOutEvent.InOutStateCreated e) {
    PropertyCommandHandler<String, String>
pCommandHandler =
this.getInOutDocumentActionCommandHandler();
    String pCmdContent = null;
    PropertyCommand<String, String> pCmd = new
AbstractPropertyCommand.SimplePropertyCommand<String,
String>();
    pCmd.setContent(pCmdContent);
    pCmd.setStateGetter(() -> s.getDocumentStatusId());
    pCmd.setStateSetter(p -> e.setDocumentStatusId(p));
    pCmd.setOuterCommandType(CommandType.CREATE);
    pCmd.setContext(getContext());
    pCommandHandler.execute(pCmd);
}

private PropertyCommandHandler<String, String>
InOutDocumentActionCommandHandler = new
SimpleInOutDocumentActionCommandHandler();

public void
setInOutDocumentActionCommandHandler(PropertyCommandHandler<
String, String> h) {
    this.inOutDocumentActionCommandHandler = h;
}

protected PropertyCommandHandler<String, String>
getInOutDocumentActionCommandHandler() {
    if (this.inOutDocumentActionCommandHandler != null)
{
        return this.inOutDocumentActionCommandHandler;
    }
    Object h = ApplicationContext.current.get(""
InOutDocumentActionCommandHandler");
    if (h instanceof PropertyCommandHandler) {
        return (PropertyCommandHandler<String, String>)
h;
    }
    return null;
}

public class SimpleInOutDocumentActionCommandHandler
implements PropertyCommandHandler<String, String> {
    public void execute(PropertyCommand<String, String>

```

```
command) {
        String trigger = command.getContent();
        if (Objects.equals(null,
command.getStateGetter().get()) && Objects.equals(null,
trigger)) {
            command.getStateSetter().accept("Drafted");
            return;
        }
        if (Objects.equals("Drafted",
command.getStateGetter().get()) &&
Objects.equals("Complete", trigger)) {

            command.getStateSetter().accept("Completed");
            return;
        }
        if (Objects.equals("Drafted",
command.getStateGetter().get()) && Objects.equals("Void",
trigger)) {
            command.getStateSetter().accept("Voided");
            return;
        }
        if (Objects.equals("Completed",
command.getStateGetter().get()) && Objects.equals("Close",
trigger)) {
            command.getStateSetter().accept("Closed");
            return;
        }
        if (Objects.equals("Completed",
command.getStateGetter().get()) && Objects.equals("Reverse",
trigger)) {
            command.getStateSetter().accept("Reversed");
            return;
        }
        throw new
IllegalArgumentException(String.format(
        "State: %1$s, command: %2$s",
command.getStateGetter().get(), trigger));
    }
}
// ...
}
```

---

上面的代碼依賴的接口PropertyCommandHandler的代碼如下：

---

```
package org.dddml.wms.specialization;

public interface PropertyCommandHandler<TContent, TState> {
    void execute(PropertyCommand<TContent, TState> command);
}
```

---

接口PropertyCommand的代碼如下：

---

```
package org.dddml.wms.specialization;

import java.util.function.Consumer;
import java.util.function.Supplier;

public interface PropertyCommand<TContent, TState> {
    TContent getContent();
    void setContent(TContent content);

    Supplier<TState> getStateGetter();
    void setStateGetter(Supplier<TState> stateGetter);

    Consumer<TState> getStateSetter();
    void setStateSetter(Consumer<TState> stateSetter);

    String getOuterCommandType();
    void setOuterCommandType(String type);

    Object getContext();
    void setContext(Object context);
}
```

---

## 第12章 處理領域服務

定義在實體中的方法一般來說只應該改變一個聚合實例的狀態，如果一個方法要改變多個聚合實例的狀態，那麼它應該被定義為一個領域服務。對於只會修改單個聚合實例的狀態的方法，我們自然可以使用數據庫本地事務來保證所修改數據項的強一致性。但對於修改多個聚合實例的狀態的領域服務，建議優先使用最終一致性模型。但是，部分初級應用開發人員在開發工作中會極度依賴數據庫事務，無法在保證基本的開發效率的前提下手動編碼實現數據的最終一致性。

雖然DDD的領域服務不等同於MSA的微服務，但是很多開發團隊不敢把已經非常臃腫複雜的單體應用拆分開，不敢實踐微服務架構（MSA），很大一部分原因也是對採用最終一致性模型心存畏懼。雖然微服務一詞沒有標準的定義，但是還是存在一些基本的實踐上的共識：每個微服務應該使用自己獨立的數據庫，微服務架構應該採用最終一致性模型去實現分佈在不同數據庫中的數據項的最終一致性。

雖然大家都覺得聚合外最終一致是最好的，但是有時在做好聚合分析的前提下，也是可以妥協的。也就是說，在實現跨聚合操作的領域服務時，可以考慮使用數據庫本地事務支持的強一致性模型。

如果打算使用最終一致性模型來實現領域服務，可以考慮使用Saga模式，包括基於協作的Saga（Choreography-based saga）以及基於編制的Saga（Orchestration-based saga）。DDDML工具可以為這兩種Saga的實現提供幫助。

我們的DDDML工具可以生成發佈、訂閱領域事件的代碼，使開發人員在實現基於協作的Saga時可以專注於業務邏輯的編碼。此外，它還可以生成支持使用基於消息的命令（消息通信的請求/異步響應模式）遠程調用實體方法或服務方法的代碼，這些代碼使用了支持DSL的Saga框架，可以大大減少開發人員實現基於編制的Saga所需的編碼工作量。

## 12.1 處理數據的一致性

為了實現可能會修改多個聚合實例狀態的領域服務，我們可能需要在採用何種一致性模型上做出選擇。一般來說，應該優先考慮使用最終一致性模型。

但是，離開了數據庫事務，開發人員要自己實現一個業務事務（Business Transaction）涉及的所有數據項的最終一致並不簡單。他們需要考慮：如果業務事務中的某個步驟（Step）在執行時發生了異常，是不是應該重試？這個步驟（方法）的執行邏輯是幂等的嗎？如果不是幂等的怎麼辦？

此外，開發人員還要考慮業務事務之間的隔離性問題。業務事務沒法提供像數據庫本地事務那樣強的隔離性。業務事務中的一個步驟結束之後其結果是確定的，所有人都能看到這個結果，沒有數據庫事務那樣的回滾功能可用。大家可能會在這個執行結果的基礎上疊加更多的操作，這為開發人員正確地實現已執行步驟的“補償”操作帶來了更大的挑戰。

來看個例子，假設在一個WMS應用的領域模型中，`InventoryItem`（庫存單元）實體（聚合根）表示“某個產品在某個貨位上的庫存數量”。如果我們打算遵循聚合外最終一致的原則來實現一個庫存調撥

( Inventory Movement ) 服務，那麼需要考慮執行這個調撥服務時可能存在如下場景：

- 在源貨位上，某產品A本來的庫存數量是1000個。
- 我們執行了一個調撥操作，打算把100個產品A轉移到目標貨位上。
- 一開始，庫存調撥服務扣減了源貨位的庫存數量，在這個貨位上產品A的庫存數量變成了900——這個結果是確定的，不能通過數據庫事務回滾，但此時目標貨位的庫存數量還沒有增加，調撥還沒有最終完成。
- 接著，其他人因為生產加工的需要，用掉（出庫）了在源貨位上的100個產品A，庫存數量變成了800——這個結果也是確定的，不能使用數據庫事務回滾。
- 然後，因為某些原因，我們沒法在目標貨位上增加產品A的庫存數量，所以需要取消這次調撥操作。
- 這時，應該把源貨位上產品A的庫存數量改為900個，也就是在數量800個的基礎上加回100個，而不能直接將庫存數量改回調撥操作發生前的數量（1000個）。

可見，沒有了數據庫事務的幫助，開發人員在開發功能時需要面對很多問題。對於初級開發人員來說，這個過程很容易犯錯，以至於有人說：引入最終一致性的結果一定是最終不一致。

我們希望DDDMIL工具能幫助我們降低採用最終一致性模型的成本。如前面展示過的，工具生成的聚合代碼已經為此提供了一些基礎支持。其中一個很關鍵的地方是，在默認情況下，為實體的方法所生成的代碼中包含了基本的幂等性處理邏輯。在這些幂等的實體方法的基礎上實現不同聚合的數據的最終一致性，能少犧牲很多腦細胞。並且，默認情況下，工具生成的實體命令方法的代碼會使用離線樂觀鎖來檢測併發衝突。我們以此提示開發人員：“你所做的”必須基於“你看到的”——並且中間沒有人改變過它。這對實現一個需要修改在不同的數據庫內數據項的長時間運行的事務來說尤其有用，因為我們沒法通過提高數據庫事務的隔離級別或者使用數據庫層面的鎖機制（悲觀鎖）來簡單地繞過問題。我們需要開發人員在編寫調用這些實體方法的客戶端（Client）代碼時直面問題，而不是先“湊合”搞出一個可能會產生錯誤結果的實現。

## 12.1.1 使用數據庫事務實現一致性

如果開發人員能完全按照聚合外最終一致的原則來編碼，那麼系統會具備良好的水平擴展能力，因為只要按聚合來拆分數據庫（微服務）就可以了。但很多時候，確實沒有必要上來就搞“一個聚合一個數據庫”——先開發一個單體應用也是可以考慮的。如果領域服務訪問的各項數據都在同一個數據庫內，基於可投入資源、項目期限等因素，直接使用數據庫本地事務就可以支持的強一致性模型來實現領域服務並非完全不能接受。

以一個CRM應用為例，這裡定義了一個線索跟進服務用於記錄線索跟進的結果，DDDML代碼片段如下：

---

```
services:
  LeadFollowupService:
    methods:
      UpdateAfterFollowup:
        parameters:
          LeadId:
            type: id
            referenceType: Lead
          Salutation:
            type: name
          # ...
```

---

然後使用代碼生成工具從這個DDDML文檔生成服務端的接口（LeadFollowup-ApplicationService，Java

代碼)：

---

```
package org.ddm1.crm.domain.service;

import org.ddm1.crm.domain.*;

public interface LeadFollowupApplicationService {
    void
when(LeadFollowupServiceCommands.UpdateAfterFollowup c);
}
```

---

默認情況下，工具會自動生成這個服務的**RESTful API**層的代碼。這樣，客戶端可以發送**POST**請求到如下URL調用服務：

---

```
{BASE_URL}/LeadFollowupService/UpdateAfterFollowup
```

---

添加一個**LeadFollowupApplicationService**接口的實現類（Java代碼）：

---

```
package org.ddm1.crm.domain.service;

import
org.springframework.transaction.annotation.Transactional;

public class LeadFollowupApplicationServiceImpl implements
LeadFollowupApplicationService {

    @Transactional(readOnly = true)
    public void
when(LeadFollowupServiceCommands.UpdateAfterFollowup c) {
        // 在這裡實現業務邏輯
    }
}
```

}

---

上面展示的LeadFollowupApplicationServiceImpl 類中的when方法使用了Spring框架的AOP事務註解（@Transactional），也就是說，這個服務方法打算使用數據庫事務來保證多個聚合狀態之間的強一致。

如果打算在領域服務上使用數據庫事務，如下問題可能需要注意：

- 儘可能讓客戶端通過參數傳入服務執行所需的信息，在服務執行過程中儘可能減少對數據庫的查詢。如果需要查詢，理想情況是隻依賴於寫模型的存儲庫（Repository）的“Get By ID”方法來查詢單個實體的狀態。

- 有些時候，一個服務方法可能會調用多個標註了@Transactional的其他方法，可能這些方法並不需要在同一個數據庫事務內執行才能得到正確的結果，但是通過在服務方法上聲明AOP事務可以減少事務開啟和關閉的次數，從而提升性能表現。

- 如果使用只讀的事務（@Transactional(readOnly=true)）就能解決問題，那麼不要使用非只讀的事務。

## 12.1.2 使用Saga實現最終一致性

如果打算使用最終一致性模型來實現多個聚合之間狀態（數據）的一致性，有必要考慮使用Saga模式 [1] 。

這裡說的Saga是什麼？

Saga又叫“長時間運行的事務”（Long-running transaction），它是跨越多個服務的業務事務的實現。Saga由一系列本地事務組成，每個本地事務在更新數據庫之後通過發佈消息或事件觸發下一個本地事務執行。如果一個本地事務因為違反了業務規則而失敗，那麼就對之前已執行的那些本地事務執行相應的補償事務，以撤銷它們對數據庫做過的修改。

顯然，Saga實現業務事務所採用的是最終一致性模型。保證數據的最終一致需要開發人員自己實現的業務邏輯。

 提示Saga所說的服務並不是特指DDD領域服務，可以理解為使用自己的獨立數據庫的軟件組件——微服務一般就是這麼做的。

Saga所說的業務事務也不是指數據庫事務，而是指應用的開發人員通過編寫多個步驟的業務邏輯代碼

完成的業務/商業上的事務/交易。這裡業務、商業的英文都是**business**，事務、交易的英文都是**transaction**。

基於協作的Saga沒有中心協調者，服務之間通過公開地發佈消息/事件來推進業務流程。比如：

- 客戶下了一個訂單後，訂單服務會發佈一個**OrderPlaced**事件——它其實並不在意誰對這個事件感興趣，發佈事件的意思就相當於大喊：“有人下單了！”
- 也許庫存服務會對這個事件感興趣，它會訂閱這個事件。當收到這個事件時，它可能會按照訂單上的產品以及數量信息預留庫存（**Reserve Inventory**），併為接下來的發貨操作做準備。同樣地，在預留好庫存之後，它也會發佈一個**InventoryReserved**事件——相當於大喊：“庫存預留好了！”
- 也許，揀貨服務一直就在監聽這類消息，因為預留好庫存之後，接下來就應該通知揀貨員幹活了.....

在上面描述的業務流程的執行過程中，並沒有一個協調者負責：

- “命令”每個服務做什麼。
- 記錄每個服務完成任務後的結果。

- 決定這個服務完成任務後由哪個服務接著幹，或者這個服務徹底幹不下去了應該怎麼辦。

而基於編制的**Saga**就存在這個居中指揮的協調者，我們可以把這個協調者稱為**Saga Manager**。**Saga Manager**與服務之間的交互可能使用異步的基於消息的通信機制，也可能使用同步的RPC方式。

接下來的兩節中，會使用示例來說明如何使用異步消息通信（**Asynchronous Messaging**）實現**Saga**，此方法適用於如下場景：

- 實現一個領域服務所涉及的多個聚合實例狀態的最終一致性。即使這些聚合都使用了同一個數據庫，且更新聚合狀態的代碼都屬於同一個微服務項目，我們仍然應該優先考慮採用最終一致性模型。

- 實現同一個限界上下文內、不同微服務的數據庫中各項數據的最終一致性。

- 實現大粒度的服務甚至是單體應用之間的集成，在不同的應用之間實現數據的最終一致性。

基於消息的異步通信機制通常會使用消息代理<sup>[2]</sup>（**Message Broker**）。使用消息代理有必要考慮限界上下文的邊界，建議如下：

- 明確哪些消息代理是位於限界上下文之內的內部消息代理。同一限界上下文的不同服務/微服務之間可共享一個內部消息代理，它們使用異步的基於消息的通信機制進行交互。這些內部消息代理應該看作所屬的限界上下文專用的技術基礎設施，不要在不同的限界上下文之間共享它們。不過，可以考慮允許（跨越多個限界上下文的）防腐層的代碼訪問內部消息代理，這是個小小的例外，此時防腐層應該只使用一個限界上下文的內部消息代理。建議把這個防腐層交給該消息代理所屬的限界上下文的開發團隊進行開發和維護。

- 不同的限界上下文之間優先使用RESTful API或其他RPC方式進行交互。

[1]

見

<https://microservices.io/patterns/data/saga.html>。

[2] 見<https://zh.wikipedia.org/zh-hans/> 消息代理。

## 12.2 發佈與處理領域事件

如前文所述，基於協作的Saga是指各個服務通過各自發布、訂閱消息/事件的方式來推進執行業務事務，沒有居中的協調者。

在這個過程中產生的消息有一部分是所謂的“偽事件”。一般來說，事件是指那些改變了系統狀態的已發生的事實，但偽事件不代表系統的狀態發生了改變。一個偽事件可能只是表示有人發出了一個查詢請求，或者表示有人給出了查詢的結果。舉例來說，訂單服務（Order Service）可能會發佈一個Get-Product Requested偽事件來請求產品的詳細信息，而產品服務（Product Service）可能會發佈一個Get-Product Replied偽事件來對這個查詢請求做出響應。這裡不對偽事件進一步展開討論。

既然在默認情況下，筆者製作的DDDMIL工具為聚合生成的代碼已經採用了事件溯源模式，那麼，如果我們只需要將這些聚合的領域事件發佈出來，應該就提供了實現基於協作的Saga所需的部分事件。

下面通過一個例子來看看DDDMIL代碼生成工具在發佈和處理領域事件方面可以幫助開發人員做些什麼。

在下面展示的工具生成的代碼中，使用Eventuate Tram<sup>[1]</sup>框架來實現領域事件的發佈。Eventuate Tram是一個用於實現可靠的事務性消息通信（Transactional Messaging）的開發框架。

[1]

參

見

<https://eventuate.io/abouteventuatetram.html>。

## 12.2.1 編寫DDDML文檔

假設領域模型中存在一個StatusItem（狀態項）聚合，描述它的DDDML代碼如下：

---

```
aggregates:
  StatusItem:
    id:
      name: StatusId
      type: id-ne
    properties:
      StatusTypeId:
        type: id-ne
      StatusCode:
        type: short-varchar
      SequenceId:
        type: id
      Description:
        type: description

    methods:
      ChangeCode:
        parameters:
          NewCode:
            type: short-varchar
          eventName: CodeChanged

    metadata:
      # -----
      # 設置為需要發佈領域事件
      PublishingEventEnabled: true
      # -----
      # 可以在發佈事件時只發布事件的引用（事件 Id.）
      # OnlyPublishingEventReference: true
```

---

在這個DDML文檔中可以看到：當ChangeCode方法被調用時，如果執行成功將產生CodeChanged事件。另外，在聚合的metadata結點中存在一個鍵值對 PublishingEventEnabled:true，這表示當StatusItem的狀態發生變化時，需要對外發布事件。

然後，我們打算創建一個名為TestDomainEventConsumerService的測試服務，用以消費和處理ItemStatus聚合發佈的領域事件。描述該服務的DDML文檔如下：

---

```
services:
  TestDomainEventConsumerService:
    methods:
      # -----
      HelloChangeCode:
        parameters:
          StatusId:
            type: id-ne
          NewCode:
            type: short-varchar
        domainEventHandler:
          forAggregateType: StatusItem
          onEvent: CodeChanged

      # -----
      HelloStatusItemStateCreated:
        parameters:
          StatusId:
            type: id-ne
          StatusTypeId:
            type: id-ne
          StatusCode:
            type: short-varchar
          SequenceId:
            type: id
          Description:
```

```
type: description
domainEventHandler:
  forAggregateType: StatusItem
  onEvent: StatusItemStateCreated
```

---

這個服務有兩個方法，在這兩個方法的定義中各存在一個**domainEventHandler**結點，它們的意思如下：

- HelloChangeCode**方法是**StatusItem**聚合發佈的**CodeChanged**事件的處理器。
- HelloStatusItemStateCreated**方法是**StatusItem**聚合發佈的**StatusItemStateCreated**事件的處理器。

## 12.2.2 生成的事件發佈代碼

在工具生成的代碼中，會使用一個領域事件發佈器接口來發布事件，這個接口的代碼（文件 **DomainEventPublisher.java**）如下：

---

```
package org.dddml.wms.specialization;

import java.util.*;

public interface DomainEventPublisher {
    void publish(String aggregateType, Object aggregateId,
List<Event> domainEvents);

    default void publish(Class<?> aggregateType, Object
aggregateId, List<Event> domainEvents) {
        publish(aggregateType.getName(), aggregateId,
domainEvents);
    }
}
```

---

上面這個**DomainEventPublisher**只是一個接口，我們還需要通過它的實現類真正把事件發佈出去。筆者製作的**DDDML**工具可以生成一個使用了**Eventuate Tram**框架的該接口的實現類，後面會展示這個實現類的代碼。

使用**Eventuate Tram**發佈的領域事件對象必須實現**Eventuate Tram**定義的那個**DomainEvent**接口。我們並不希望這個接口侵入已經生成的聚合事件對象的

代碼，所以定義了一個實現這個接口的封包類（文件 **EventEnvelope.java**）：

---

```
package org.dddml.wms.specialization.eventuate.tram;

import io.eventuate.tram.events.common.DomainEvent;

public class EventEnvelope<T> implements DomainEvent {
    private T data;

    public T getData() {
        return data;
    }

    public void setData(T data) {
        this.data = data;
    }

    public EventEnvelope() {
    }

    public EventEnvelope(T data) {
        this.data = data;
    }
}
```

---

有時候，可能我們僅僅會發布領域事件的引用，關注這個事件的訂閱者可以在接收到事件的引用後，使用引用中的事件ID及URL去拉取完整的事件信息。為此，我們定義了一個實現**Eventuate Tram**的**DomainEvent**接口的事件引用抽象基類（文件 **AbstractEventReference.java**）：

---

```
package org.dddml.wms.specialization.eventuate.tram;
```

```
import io.eventuate.tram.events.common.DomainEvent;

public abstract class AbstractEventReference<TId, T>
implements DomainEvent {
    private TId eventId;

    private String url;

    public TId getEventId() {
        return eventId;
    }

    public void setEventId(TId eventId) {
        this.eventId = eventId;
    }

    public String getUrl() {
        return url;
    }

    public void setUrl(String url) {
        this.url = url;
    }

    public AbstractEventReference(TId eventId, String url) {
        this.eventId = eventId;
        this.url = url;
    }

    public AbstractEventReference() {
    }
}
```

---

現在，可以基於Eventuate Tram框架實現前面定義的那個領域事件發佈器（Domain-EventPublisher）接口了（Java類EventuateTramDomainEventPublisher的代碼）：

---

```
package org.dddml.wms.specialization.eventuate.tram;

import io.eventuate.tram.events.common.DomainEvent;
import org.dddml.wms.specialization.DomainEventPublisher;
import org.dddml.wms.specialization.Event;
import org.dddml.wms.specialization.EventReference;
import java.lang.reflect.Field;
import java.util.*;
import java.util.stream.Collectors;

public class EventuateTramDomainEventPublisher implements
DomainEventPublisher {
    private
io.eventuate.tram.events.publisher.DomainEventPublisher
innerDomainEventPublisher;

    public
EventuateTramDomainEventPublisher(io.eventuate.tram.events.p
ublisher.DomainEventPublisher publisher) {
        this.innerDomainEventPublisher = publisher;
    }

    @Override
    public void publish(String aggregateType, Object
aggregateId, List<Event> domainEvents) {
        innerDomainEventPublisher
            .publish(aggregateType, aggregateId,
domainEvents.stream().map(
                e -> {
                    try {
                        if (e instanceof
EventReference) {
                            return
convertEventReference((
EventReference) e);
                    } else {
                        return
createEventEnvelope(e);
                    }
                }
            ) catch (ClassNotFoundException
e1) {
                throw new
RuntimeException(e1);
            } catch (NoSuchFieldException
e1) {

```

```

                    throw new
RuntimeException(e1);
                } catch (IllegalAccessException
e1) {
                    throw new
RuntimeException(e1);
                } catch (InstantiationException
e1) {
                    throw new
RuntimeException(e1);
                }
            }).collect(Collectors.toList())
        );
    }

    private static EventEnvelope createEventEnvelope(Object
data) throws ClassNotFoundException, NoSuchFieldException,
IllegalAccessException, InstantiationException {
        String dataClassName = data.getClass().getName();
        Class eventEnvelopeClass =
getEventEnvelopeClass(dataClassName);
        EventEnvelope eventEnvelope = (EventEnvelope)
eventEnvelopeClass.newInstance();
        eventEnvelope.setData(data);
        return eventEnvelope;
    }

    private static Class getEventEnvelopeClass(String
dataClassName) throws ClassNotFoundException,
NoSuchFieldException, IllegalAccessException {
        String thisClassName =
EventuateTramDomainEventPublisher.class.getName();
        String basePackage =
getBoundedContextBasePackageName(thisClassName);
        String envelopesClassName = basePackage +
".domain.eventuate.tram.EventEnvelopes";
        Class envelopesClass =
Class.forName(envelopesClassName);
        Field classMapField = envelopesClass.getField(
"EVENT_ENVELOPE_CLASS_MAP");
        Map<String, Class> envelopClassMap = (Map<String,
Class>) classMapField.get(null);
        return envelopClassMap.get(dataClassName);
    }
}

```

```

        private static AbstractEventReference
convertEventReference(
EventReference oer) throws ClassNotFoundException,
NoSuchFieldException, IllegalAccessException,
InstantiationException {
    String dataClassName = oer.getEventType().getName();
    Class eventReferenceClass =
getEventReferenceClass(dataClassName);
    AbstractEventReference eventReference =
(AbstractEventReference) eventReferenceClass.newInstance();
    eventReference.setEventId(oer.getEventId());
    eventReference.setUrl(oer.getUrl());
    return eventReference;
}

        private static Class getEventReferenceClass(String
dataClassName) throws ClassNotFoundException,
NoSuchFieldException, IllegalAccessException {
    String thisClassName =
EventuateTramDomainEventPublisher.class.getName();
    String basePackage =
getBoundedContextBasePackageName(thisClassName);
    String referencesClassName = basePackage +
".domain.eventuate.tram.EventReferences";
    Class referencesClass =
Class.forName(referencesClassName);
    Field classMapField = referencesClass.getField(
"EVENT_REFERENCE_CLASS_MAP");
    Map<String, Class> envelopClassMap = (Map<String,
Class>) classMapField.get(null);
    return envelopClassMap.get(dataClassName);
}

        private static String
getBoundedContextBasePackageName(String thisClassName) {
    return thisClassName.substring(0,
        thisClassName.length() -
".specialization.eventuate.tram.EventuateTramDomainEventPubl
isher".length());
}

```

---

工具還生成了這個領域事件發佈器的**Spring Boot**配置代碼：

---

```
package org.dddml.wms.specialization.eventuate.tram;

import io.eventuate.tram.events.common.*;
import
io.eventuate.tram.events.publisher.TramEventsPublisherConfig
uration;
import org.springframework.context.annotation.*;

@Configuration
@Import({TramEventsPublisherConfiguration.class})
public class EventuateTramDomainEventPublisherConfiguration
{

    @Bean
    public EventuateTramDomainEventPublisher
eventuateTramDomainEventPublisher(io.eventuate.tram.events.p
ublisher.DomainEventPublisher publisher) {
        return new
EventuateTramDomainEventPublisher(publisher);
    }
}
```

---

上面由**DDDML**工具生成的代碼一般在一個限界上下文中只需要生成一次，因為這些代碼和領域模型存在哪些值對象、聚合及服務沒有直接關係。而下面的代碼在**DDDML**文檔更新後可能需要重新生成。

工具會為那些需要發佈的聚合的領域事件生成實現**Eventuate Tram**領域事件接口的封包類（文件**EventEnvelopes.java**）：

---

```

package org.dddml.wms.domain.eventuate.tram;

import
org.dddml.wms.specialization.eventuate.tram.EventEnvelope;
import org.dddml.wms.domain.statusitem.*;
import java.util.*;

public class EventEnvelopes {
    public interface StatusItem {
        public static class StatusItemStateCreatedEnvelope
extends
EventEnvelope<AbstractStatusItemEvent.SimpleStatusItemStateC
reated> {
            }

        public static class
StatusItemStateMergePatchedEnvelope extends

EventEnvelope<AbstractStatusItemEvent.SimpleStatusItemStateM
ergePatched> {
            }

        public static class StatusItemBlobEventEnvelope
extends EventEnvelope<
AbstractStatusItemEvent.StatusItemBlobEvent> {
            }

        public static class CodeChangedEnvelope extends
EventEnvelope<AbstractStatusItemEvent.CodeChanged> {
            }
    }

    public static Map<String, Class<?>>
EVENT_ENVELOPE_CLASS_MAP;
    static {
        Map<String, Class<?>> map = new HashMap<>();

        map.put(AbstractStatusItemEvent.SimpleStatusItemStateCreated
.class.getName(),
StatusItem.StatusItemStateCreatedEnvelope.class);

        map.put(AbstractStatusItemEvent.SimpleStatusItemStateMergePa
tched.class.getName(),
StatusItem.StatusItemStateMergePatchedEnvelope.class);
    }
}

```

```

map.put(AbstractStatusItemEvent.StatusItemClobEvent.class.get
tName(), StatusItem.StatusItemClobEventEnvelope.class);

map.put(AbstractStatusItemEvent.CodeChanged.class.getName(),
StatusItem.CodeChangedEnvelope.class);
    EVENT_ENVELOPE_CLASS_MAP = map;
}
}

```

---

我們來看一看對聚合**StatusItem**啟用事件發佈  
(**PublishingEventEnabled:true**) 之後，**DDML**工具  
生成的應用服務接口  
(**StatusItemApplicationService**) 的實現類代碼會發  
生什麼變化：

```

public abstract class AbstractStatusItemApplicationService
implements StatusItemApplicationService {
    // ...
    private void persist(EventStoreAggregateId
eventStoreAggregateId, long version, StatusItemAggregate
aggregate, StatusItemState state) {
        getEventStore().appendEvents(eventStoreAggregateId,
version,
            aggregate.getChanges(), (events) -> {
                getStateRepository().save(state);
                // 注意這裡，使用 DomainEventPublisher 發佈領域
事件：
                getDomainEventPublisher().publish(
"org.ddml.wms.domain.statusitem.StatusItem",
                    eventStoreAggregateId.getId(),
                    (List<Event>)events);
            });
        if (aggregateEventListener != null) {
            aggregateEventListener.eventAppended(new
AggregateEvent<>(aggregate, state, aggregate.getChanges()));
        }
    }
}

```

```
// ...
}
```

---

生成了上面的這些代碼之後，當**StatusItem**聚合的狀態發生變化時，相應的領域事件就會被髮布到消息代理中。

因為我們在**DDDM**中定義的領域服務**TestDomainEventConsumerService**想要消費這些**StatusItem**聚合的領域事件，所以工具為此生成了相應的領域事件處理器（Domain Event Handler），代碼如下：

---

```
package org.dddml.wms.domain.eventuate.tram;

import
io.eventuate.tram.events.subscriber.DomainEventEnvelope;
import
io.eventuate.tram.events.subscriber.DomainEventHandlers;
import
io.eventuate.tram.events.subscriber.DomainEventHandlersBuilder;
import org.dddml.wms.domain.statusitem.*;
import org.dddml.wms.domain.service.*;
import java.util.*;

public class TestDomainEventConsumerServiceEventConsumer {
    public TestDomainEventConsumerApplicationService
testDomainEventConsumerApplicationService;

    public TestDomainEventConsumerServiceEventConsumer(
TestDomainEventConsumerApplicationService
testDomainEventConsumerApplicationService) {
        this.testDomainEventConsumerApplicationService =
testDomainEventConsumerApplicationService;
    }
}
```

```

public DomainEventHandlers domainEventHandlers() {
    return DomainEventHandlersBuilder
        .forAggregateType("org.dddml.wms.domain.statusitem.StatusItem")
        .onEvent(EventEnvelopes.StatusItem.CodeChangedEnvelope.class,
            this::handleStatusItemCodeChanged)
            .andForAggregateType(
"org.dddml.wms.domain.statusitem.StatusItem")

        .onEvent(EventEnvelopes.StatusItem.StatusItemStateCreatedEnvelope.class, this::
handleStatusItemStateCreated)
            .build();
}

void
handleStatusItemCodeChanged(DomainEventEnvelope<EventEnvelopes.StatusItem.CodeChangedEnvelope> ee) {
    AbstractStatusItemEvent.CodeChanged e =
ee.getEvent().getData();

TestDomainEventConsumerServiceCommands.HelloChangeCode c =
new
TestDomainEventConsumerServiceCommands.HelloChangeCode();
    c.setStatusId(e.getStatusId());
    c.setNewCode(e.getNewCode());
    c.setCommandId(e.getCommandId());
    c.setRequesterId(e.getCreatedBy());
    testDomainEventConsumerApplicationService.when(c);
}

void
handleStatusItemStateCreated(DomainEventEnvelope<EventEnvelopes.StatusItem.StatusItemStateCreatedEnvelope> ee) {
    AbstractStatusItemEvent.SimpleStatusItemStateCreated e =
ee.getEvent().getData();

TestDomainEventConsumerServiceCommands.HelloStatusItemStateCreated c =
new
TestDomainEventConsumerServiceCommands.HelloStatusItemStateCreated();

```

```

        c.setStatusId(e.getStatusId());
        c.setStatusTypeId(e.getStatusTypeId());
        c.setStatusCode(e.getStatusCode());
        c.setSequenceId(e.getSequenceId());
        c.setDescription(e.getDescription());
        c.setCommandId(e.getCommandId());
        c.setRequesterId(e.getCreatedBy());
        testDomainEventConsumerApplicationService.when(c);
    }
}

```

---

筆者製作的DDML工具生成的領域事件消費端的  
Spring Boot配置代碼如下：

```

package org.ddm.wms.domain.eventuate.tram;

import io.eventuate.tram.events.common.DomainEvent;
import
io.eventuate.tram.events.publisher.TramEventsPublisherConfig
uration;
import io.eventuate.tram.events.subscriber.*;
import io.eventuate.tram.messaging.consumer.MessageConsumer;
import org.springframework.beans.factory.annotation.*;
import org.springframework.context.annotation.*;
import java.util.*;
import
org.ddm.wms.specialization.eventuate.tram.EventEnvelope;
import org.ddm.wms.domain.service.*;

@Configuration
@Import({TramEventSubscriberConfiguration.class})
public class EventuateTramDomainEventConsumersConfiguration
{
    @Bean
    public DomainEventDispatcher
testDomainEventConsumerServiceDomainEventDispatcher(
DomainEventDispatcherFactory domainEventDispatcherFactory,
    MessageConsumer messageConsumer,
    TestDomainEventConsumerServiceEventConsumer
testDomainEventConsumerServiceEventConsumer) {
        return domainEventDispatcherFactory.make(

```

```
"testDomainEventConsumerServiceDomainEventDispatcher",  
  
testDomainEventConsumerServiceEventConsumer.domainEventHand  
lers());  
}  
  
@Bean  
public TestDomainEventConsumerServiceEventConsumer  
testDomainEventConsumerServiceEventConsumer(  
TestDomainEventConsumerApplicationService  
testDomainEventConsumerApplicationService){  
    return new  
TestDomainEventConsumerServiceEventConsumer(  
testDomainEventConsumerApplicationService);  
}  
}  
}
```

---

可以看到，這些DDDML工具生成的代碼主要是 **TestDomainEventConsumerServiceEventConsumer** 類 及 **EventuateTramDomainEventConsumersConfiguration** 類，幫助 **TestDomain-EventConsumerService** 服務完成 對 **StatusItem** 聚合所發佈的 **CodeChanged** 與 **StatusItemState-Created** 事件的訂閱。

至此，DDDML工具已經為我們生成了領域事件生產端發佈事件的代碼，還生成了領域事件消費端訂閱事件的代碼，只需要我們自己動手寫業務邏輯代碼。接下來就看一看需要我們手動編寫的這部分代碼。

### 12.2.3 編寫生產端聚合的業務邏輯

在默認情況下，工具會生成創建 StatusItem ( 產生 StatusItemStateCreated 事件 ) 的代碼。但我們還是需要自己實現 StatusItem 聚合根的 ChangeCode 方法的業務邏輯，可以按如下方式編寫代碼 ( 文件 ChangeCodeLogic.java ) :

---

```
package org.dedml.wms.domain.statusitem;

import org.dedml.wms.specialization.MutationContext;
import org.dedml.wms.specialization.VerificationContext;
import java.util.Date;

public class ChangeCodeLogic {
    public static void verify(StatusItemState
statusItemState, String newCode, VerificationContext
verificationContext) {
    }

    public static StatusItemState mutate(StatusItemState
statusItemState, String newCode,
MutationContext<StatusItemState,
StatusItemState.MutableStatusItemState> mutationContext) {
        return new StatusItemState() {
            @Override
            public String getStatusId() {
                return statusItemState.getStatusId();
            }

            @Override
            public String getStatusTypeId() {
                // ChangeCode 方法不改變 statusTypeId 屬性
                return statusItemState.getStatusTypeId();
            }

            @Override
```

```
    public String getStatusCode() {
        // 使用了新的狀態代碼
        return newCode;
    }
    // 省略其他代碼
};

}
```

---

可以看到，這些代碼是開發人員不得不寫的純粹的業務邏輯。在編寫它們的時候，開發人員完全可以不考慮如何發佈領域事件這樣的小事情。

## 12.2.4 實現消費端領域事件的處理

我們需要實現**DDDML**中定義的**TestDomainEventConsumerService**服務，這個服務的兩個方法是**StatusItem**聚合發佈的領域事件**CodeChanged**與**StatusItemStateCreated**的真正處理器。

如果使用**Java**語言來實現，可以按如下方式編寫**TestDomainEventConsumerApplication-Service**接口的實現類：

---

```
package org.dddml.wms.domain.service;

public class TestDomainEventConsumerApplicationServiceImpl
implements
TestDomainEventConsumerApplicationService{
    @Override
    public void
when(TestDomainEventConsumerServiceCommands.HelloChangeCode
c) {
        System.out.println("Hello, new code: " +
c.getNewCode());
    }

    @Override
    public void
when(TestDomainEventConsumerServiceCommands.HelloStatusItems
stateCreated c) {
        System.out.println("Hello, status Id.: " +
c.getStatusId());
    }
}
```

---

## 12.3 支持基於編制的Saga

繼續舉例說明DDDML的代碼生成工具可以給實現基於編制的Saga提供什麼幫助。

使用DSL來定義Saga可以讓代碼更簡潔，這裡考慮使用支持DSL的基於編制的Saga框架，比如Eventuate Tram Saga<sup>[1]</sup>這樣的框架。

假設，我們在開發一個WMS應用時，在領域模型中創建了兩個聚合，即InventoryItem與InOut。關於這兩個聚合，第9章中使用DDDML描述過它們的部分模型信息。

現在，我們想要一個“硬生生地”直接修改庫存單元的“在庫數量”的服務方法。雖然是直接修改庫存單元的在庫數量，但是仍然希望使用InOut（入庫/出庫單）來保存庫存數量的修改記錄，所以這個方法會涉及兩個聚合。這裡打算使用聚合外最終一致的策略來實現這個修改在庫數量的方法。這可能有點麻煩，但是我們想以此為代價換來應用水平擴展能力的提升。因為，如果採用聚合外最終一致，在必要的時候，我們就可以很容易地將方法涉及的每個聚合都拆分出來單獨部署為一個微服務，而這個過程不需要對代碼做太多的修改。

首先，設計這個方法的實現思路，也就是這個業務事務的各個步驟，大致如下：

1 ) 查詢庫存單元 ( **InventoryItem** ) 信息。根據查詢的結果，判斷到底是需要新建一條庫存單元記錄還是更新已有的庫存單元記錄，以及入庫/出庫單的行項的 **MovementQuantity** ( 移動數量 ) 的應該是多少。

2 ) 創建一個入庫/出庫單。這個單據只有一行，行項的 **MovementQuantity** 是更新後的在庫數量與當前在庫數量 ( 我們在上一個步驟看到的在庫數量 ) 的差值。

3 ) 添加一個庫存單元條目 ( **InventoryItemEntry** ) 。庫存單元聚合使用了賬務模式，所以我們需要通過這個方式去間接地更新庫存單元的在庫數量。

4 ) 如果更新庫存單元成功，那麼將入庫/出庫單的狀態更新為“已完成”。

5 ) 如果更新庫存單元失敗，那麼將入庫 / 出庫單更新為“已取消”——這是第2個步驟的補償操作。

可以注意到，相對於簡單地使用數據庫本地事務來保證強一致性的做法，這裡明顯多了第4項及第5項編碼任務。

我們可以在DDDML中為以上步驟定義相應的實體方法，這些方法是用於實現基於編制的**Saga**的構造塊，DDDML工具會為我們生成支持通過“消息命令”遠程調用這些方法的代碼。

[1] Sagas for microservices,  
<https://github.com/eventuate-tram/eventuate-tram-sagas>。

### 12.3.1 編寫DDDML文檔

在庫存單元聚合的DDDML文檔中我們將定義以下三個方法：

·**CreateOrUpdateInventoryItem**，這個方法是更新在庫數量的服務方法的入口。

·**Get**，這個方法是通過聚合根ID獲取聚合狀態的查詢方法。其實即使不在DDDML中定義，代碼生成工具默認也會生成它。

·**AddInventoryItemEntry**，這個方法會添加一個庫存單元條目，這是（間接地）修改庫存單元的那些數量屬性（賬目）的唯一方式。

定義這個庫存單元聚合的DDDML代碼如下：

---

```
aggregates:
  InventoryItem:
    id:
      name: InventoryItemId
      type: InventoryItemId
    properties:
      # 在庫數量
      OnHandQuantity:
        type: decimal
      # ...
      # 條目（分錄）
    Entries:
      itemType: InventoryItemEntry
```

```

entities:
  InventoryItemEntry:
    immutable: true
    id:
      name: EntrySeqId
      type: long
    properties:
      # 在庫數量 (變化)
      OnHandQuantity:
        type: decimal
      # 來源信息
      Source:
        type: InventoryItemSourceInfo
        notNull: true
      OccurredAt:
        type: date-time
        notNull: true
    # -----
    # 唯一約束
    uniqueConstraints:
      # 一個來源不能重複產生庫存事務 (分錄)
      UniqueInventoryItemSource: [Source]

    # 定義賬目
    accounts:
      # 在庫數量
      OnHandQuantity:
        # 條目實體名稱
        entryEntityName: "InventoryItemEntry"
        # 條目數額屬性名稱
        entryAmountPropertyName: "OnHandQuantity"

methods:
  # -----
  CreateOrUpdateInventoryItem:
    notInstanceMethod: true
    parameters:
      ProductId:
        type: id-long
      LocatorId:
        type: string
      AttributeSetInstanceId:
        type: string
      OnHandQuantity:
        type: decimal

```

```

        InOutDocumentNumber:
            type: string
# -----
Get:
    metadata:
        MessagingCommandEnabled: true
# -----
AddInventoryItemEntry:
    isInternal: true
    parameters:
        EntryOnHandQuantity:
            type: decimal
        Source:
            type: InventoryItemSourceInfo

valueObjects:
# -----
# 庫存單元 Id
InventoryItemId:
    properties:
        ProductId:
            type: id-long
            length: 60
        # 貨位 Id.
        LocatorId:
            type: string
            length: 50
        # 屬性集實例 Id.
        AttributeSetInstanceId:
            type: string
            length: 50
# -----
# 庫存單元來源信息
InventoryItemSourceInfo:
    properties:
        # 單據類型 Id.
        DocumentTypeId:
            type: string
        # 單據號
        DocumentNumber:
            type: string
        # 行號
        LineNumber:
            type: string
        # 行的子序列號 (一個源單據行項可能產生多個庫存

```

事務條目 )

```
LineSubSeqId:  
    type: int
```

---

需要注意的是，實體InventoryItem的方法CreateOrUpdateInventoryItem是一個非實例方法(notInstanceMethod:true)，這往往意味著它很有可能改變多個聚合實例的狀態，所以可以把非實例方法理解為領域服務的一種變體。雖然可以在DDML文檔中為這個方法單獨定義一個服務，但有時候這種做法有點煩人，不如在聚合內定義一個非實例方法清爽。

方法CreateOrUpdateInventoryItem的參數列表如下。

- **ProductId**：產品的ID。
- **LocatorId**：貨位的ID。
- **AttributeSetInstanceId**：庫存單元的屬性集實例的ID。
- **OnHandQuantity**：在庫數量（即，我們要把庫存單元的在庫數量修改成為這個數量）。
- **InOutDocumentNumber**：生成的入庫/出庫單的單號。

另外，在`DDDML`中需要把`Get`方法支持“消息命令（請求 / 異步響應模式）”的選項打開（`MessagingCommandEnabled:true`），這樣筆者製作的`DDDML`工具就會為這個方法生成支持使用異步消息通信進行遠程調用的代碼。因為這個方法是`DDDML`工具會默認生成的方法，所以這裡不用對它做其他更多的設置（比如聲明它的返回值類型等）。



注意`DDDML`工具會把以下這些名稱當成保留的方法名稱：`Get`、`Create`、`Patch`、`Delete`、`Remove`和`MergePatch`。默認情況下，工具會自動為實體生成部分使用這些名稱的方法。我們可以在`DDDML`中為這些方法添加一些聲明以調整工具生成的代碼。

現在給入庫/出庫單（`InOut`）聚合增加以下三個方法：

- `InternalCreateSingleLineInOut`，這個方法會創建一個入庫/出庫單（`InOut`），這個單據只有一行（`InOutLine`）。
- `InternalComplete`，這個方法會將入庫/出庫單的狀態更新為“已完成”。
- `InternalVoid`，這個方法會將入庫/出庫單更新為“已取消”。

`InOut`聚合的`DDDML`代碼如下：

---

```
aggregates:
  InOut:
    id:
      name: DocumentNumber
      type: string

    properties:
      # 單據狀態 Id
      DocumentStatusId:
        type: string
        commandType: DocumentAction
        commandName: DocumentAction
        # -----
        # 單據狀態的狀態機
        stateMachine:
          # 轉換
          transitions:
            - sourceState: null
              trigger: null
              targetState: "Drafted"
            - sourceState: "Drafted"
              trigger: "Complete"
              targetState: "Completed"
            - sourceState: "Drafted"
              trigger: "Void"
              targetState: "Voided"
            - sourceState: "Completed"
              trigger: "Close"
              targetState: "Closed"
            - sourceState: "Completed"
              trigger: "Reverse"
              targetState: "Reversed"

          # 單據類型
          DocumentTypeId:
            referenceType: DocumentType
          # 描述
          Description:
            type: string
          # 省略一些屬性

        # 出入庫行項
        InOutLines:
```

```
        itemType: InOutLine

entities:
    # 入庫/出庫行項
    InOutLine:
        id:
            name: LineNumber
            type: string
        properties:
            # 貨位 Id
            LocatorId:
                type: string
            # 產品 Id
            ProductId:
                type: id-long
            # 屬性集實例 Id
            AttributeSetInstanceId:
                type: string
            # 出入庫數量
            MovementQuantity:
                type: decimal

methods:
    # -----
    InternalCreateSingleLineInOut:
        isInternal: true
        parameters:
            # 貨位 Id
            LocatorId:
                type: string
            # 產品 Id
            ProductId:
                type: id-long
            # 屬性集實例 Id
            AttributeSetInstanceId:
                type: string
            # 入庫/出庫數量
            MovementQuantity:
                type: decimal
    # -----
    InternalComplete:
        isInternal: true
    # -----
    InternalVoid:
        isInternal: true
```

```
    isCompensationMethod: true  
    # ...
```

---

在上面的DDML代碼中，三個方法都被聲明為內部方法（`isInternal:true`），這表示它們不想被外部的客戶端（Client）調用。在實踐中，筆者對於內部方法的處理方式是不將它們暴露到RESTful API層，因為筆者希望外部Client統一通過RESTful API來使用那些對外服務。

DDML定義完畢。接下來以Eventuate Tram Saga框架為例，瞭解代碼生成工具可以給開發人員實現編制式的Saga提供哪些幫助。

## 12.3.2 生成的Saga命令處理代碼

Eventuate Tram Saga依賴於Eventuate Tram框架，Eventuate Tram框架為消息通信的命令模式（請求/異步響應模式）提供了基礎支持。Eventuate Tram要求發送的“命令”必須實現由它定義的那個Command接口。我們不想讓這個“外來”的接口侵入已經生成的聚合命令對象，所以定義了一個實現了Eventuate Tram的Command接口的封包類（文件CommandEnvelope.java），示例如下：

---

```
package org.dddml.wms.specialization.eventuate.tram;

import io.eventuate.tram.commands.common.Command;

public class CommandEnvelope<T> implements Command {
    private T data;

    public T getData() {
        return data;
    }

    public void setData(T command) {
        this.data = command;
    }

    public CommandEnvelope() {
    }

    public CommandEnvelope(T command) {
        this.data = command;
    }
}
```

---

代碼生成工具會根據DDDML中定義的實體方法自動生成這個封包類的一些子類，示例代碼如下：

---

```
package org.dddml.wms.domain.eventuate.tram;

import
org.dddml.wms.specialization.eventuate.tram.CommandEnvelope;
import org.dddml.wms.domain.inout.*;
import org.dddml.wms.domain.inventoryitem.*;
import java.util.*;

public class CommandEnvelopes {
    public interface InOut {
        public static final String SERVICE_NAME =
"InOutService";

        public static class
InternalCreateSingleLineInOutEnvelope extends
CommandEnvelope<InOutCommands.InternalCreateSingleLineInOut>
{
            public InternalCreateSingleLineInOutEnvelope() {
            }
            public
InternalCreateSingleLineInOutEnvelope(InOutCommands.Internal
CreateSingleLineInOut c) {
                super(c);
            }
        }
    }

    public static class InternalCompleteEnvelope extends
CommandEnvelope<InOutCommands.InternalComplete> {
        public InternalCompleteEnvelope() {
        }
        public
InternalCompleteEnvelope(InOutCommands.InternalComplete c) {
            super(c);
        }
    }

    public static class InternalVoidEnvelope extends
CommandEnvelope<InOutCommands.InternalVoid> {
        public InternalVoidEnvelope() {
        }
    }
}
```

```

        }
    public
InternalVoidEnvelope(InOutCommands.InternalVoid c) {
        super(c);
    }
}

public interface InventoryItem {
    public static final String SERVICE_NAME =
"inventoryItemService";

    public static class CreateInventoryItemEnvelope
extends
CommandEnvelope<CreateOrMergePatchInventoryItemDto.CreateInv
entoryItemDto> {
        public CreateInventoryItemEnvelope() {
        }
        public CreateInventoryItemEnvelope(
CreateOrMergePatchInventoryItemDto.CreateInventoryItemDto c)
{
        super(c);
    }
}

public static class MergePatchInventoryItemEnvelope
extends
CommandEnvelope<CreateOrMergePatchInventoryItemDto.MergePac
hInventoryItemDto> {
        public MergePatchInventoryItemEnvelope() {
        }
        public MergePatchInventoryItemEnvelope(
CreateOrMergePatchInventoryItemDto.MergePatchInventoryItemDt
o c) {
        super(c);
    }
}

public static class GetInventoryItemEnvelope extends
CommandEnvelope<
InventoryItemId> {
        public GetInventoryItemEnvelope() {
        }
        public GetInventoryItemEnvelope(InventoryItemId
id) {

```

```
        super(id);
    }
}

public static class AddInventoryItemEntryEnvelope
extends
CommandEnvelope<InventoryItemCommands.AddInventoryItemEntry>
{
    public AddInventoryItemEntryEnvelope() {
    }
    public
AddInventoryItemEntryEnvelope(InventoryItemCommands.AddInven
toryItemEntry c) {
        super(c);
    }
}
}
```

---

為了讓服務端能夠處理Eventuate Tram Saga命令，需要用到Saga命令處理器（ Saga Command Handlers ）。



提示如果你知道如何使用Spring MVC框架（或者其他類似的服務框架）來實現RESTful Services，可以用Spring MVC的REST Controller（@RestController）來類比理解這裡所說的“命令處理器”。它們都是請求/響應模式的服務端實現代碼，只不過這裡的命令處理器要實現的是基於消息通信的請求/異步響應模式。下一章會展示DDDDML工具生成的RESTful Services的代碼，它們與這裡展示的Saga命令處理器的代碼都是不含“業務邏輯”的很薄的一層“膠水”代碼。

筆者製作的DDML工具還會生成庫存單元  
( **InventoryItem** ) 的Saga命令處理器 ( 文件  
**InventoryItemCommandHandler.java** ) , 示例如下：

---

```
package org.ddm.wms.domain.eventuate.tram;

import io.eventuate.tram.commands.consumer.CommandHandlers;
import io.eventuate.tram.commands.consumer.CommandMessage;
import io.eventuate.tram.messaging.common.Message;
import
io.eventuate.tram.sagas.participant.SagaCommandHandlersBuild
er;
import org.ddm.wms.domain.inventoryitem.*;
import org.ddm.wms.specialization.DomainErrorFailure;
import static
io.eventuate.tram.commands.consumer.CommandHandlerReplyBuild
er.withFailure;
import static
io.eventuate.tram.commands.consumer.CommandHandlerReplyBuild
er.withSuccess;

public class InventoryItemCommandHandler {
    public InventoryItemApplicationService
inventoryItemApplicationService;

    public
InventoryItemCommandHandler(InventoryItemApplicationService
inventoryItemApplicationService) {
        this.inventoryItemApplicationService =
inventoryItemApplicationService;
    }

    public CommandHandlers commandHandlerDefinitions() {
        return SagaCommandHandlersBuilder

        .fromChannel(CommandEnvelopes.InventoryItem.SERVICE_NAME)

        .onMessage(CommandEnvelopes.InventoryItem.CreateInventoryIte
mEnvelope.class, this::createInventoryItem)

        .onMessage(CommandEnvelopes.InventoryItem.MergePatchInventor
```

```

yItemEnvelope.class, this::mergePatchInventoryItem)

.onMessage(CommandEnvelopes.InventoryItem.GetInventoryItemEn
velope.class, this::getInventoryItem)

.onMessage(CommandEnvelopes.InventoryItem.AddInventoryItemEn
tryEnvelope.class, this::addInventoryItemEntry)
    .build();
}

private Message
createInventoryItem(CommandMessage<CommandEnvelopes.Inventor
yItem.CreateInventoryItemEnvelope> ce) {
    try {

CreateOrMergePatchInventoryItemDto.CreateInventoryItemDto
cmd = ce.getCommand().getData();
    inventoryItemApplicationService.when(cmd);
    return withSuccess();
} catch (Exception e) {
    return
withFailure(DomainErrorFailure.ofDomainErrorOrThrow(e));
}
}

private Message
mergePatchInventoryItem(CommandMessage<CommandEnvelopes.Inve
ntoryItem.MergePatchInventoryItemEnvelope> ce) {
    try {

CreateOrMergePatchInventoryItemDto.MergePatchInventoryItemDt
o cmd = ce.getCommand().getData();
    inventoryItemApplicationService.when(cmd);
    return withSuccess();
} catch (Exception e) {
    return
withFailure(DomainErrorFailure.ofDomainErrorOrThrow(e));
}
}

public Message
getInventoryItem(CommandMessage<CommandEnvelopes.InventoryIt
em.GetInventoryItemEnvelope> cm) {
    try {

```

```

CommandEnvelopes.InventoryItem.GetInventoryItemEnvelope cmd
= cm.getCommand();
    InventoryItemStateDto stateDto = when(cmd);
    return withSuccess(stateDto);
} catch (Exception e) {
    return
withFailure(DomainErrorFailure.ofDomainErrorOrThrow(e));
}
}

private InventoryItemStateDto
when(CommandEnvelopes.InventoryItem.GetInventoryItemEnvelope
cmd) {
    InventoryItemState state =
inventoryItemApplicationService.get(cmd.getData());
    InventoryItemStateDto stateDto = null;
    if (state != null) {
        InventoryItemStateDto.DtoConverter dtoConverter
= new InventoryItemStateDto.DtoConverter();
        stateDto =
dtoConverter.toInventoryItemStateDto(state);
    }
    return stateDto;
}

private Message
addInventoryItemEntry(CommandMessage<CommandEnvelopes.InventoryItem.AddInventoryItemEntryEnvelope> ce) {
    try {
        InventoryItemCommands.AddInventoryItemEntry cmd
= ce.getCommand().getData();
        inventoryItemApplicationService.when(cmd);
        return withSuccess();
    } catch (Exception e) {
        return
withFailure(DomainErrorFailure.ofDomainErrorOrThrow(e));
    }
}
}

```

---

DDDML工具也可生成入庫/出庫單應用服務  
( InOutApplicationService ) 的Saga命令處理器 ( 文

件InOutCommandHandler.java ) , 示例如下 :

---

```
package org.dddml.wms.domain.eventuate.tram;

import io.eventuate.tram.commands.consumer.CommandHandlers;
import io.eventuate.tram.commands.consumer.CommandMessage;
import io.eventuate.tram.messaging.common.Message;
import
io.eventuate.tram.sagas.participant.SagaCommandHandlersBuild
er;
import org.dddml.wms.domain.inout.*;
import org.dddml.wms.specialization.DomainErrorFailure;
import static
io.eventuate.tram.commands.consumer.CommandHandlerReplyBuild
er.withFailure;
import static
io.eventuate.tram.commands.consumer.CommandHandlerReplyBuild
er.withSuccess;

public class InOutCommandHandler {
    public InOutApplicationService inOutApplicationService;

    public InOutCommandHandler(InOutApplicationService
inOutApplicationService) {
        this.inOutApplicationService =
inOutApplicationService;
    }

    public CommandHandlers commandHandlerDefinitions() {
        return SagaCommandHandlersBuilder

.fromChannel(CommandEnvelopes.InOut.SERVICE_NAME)
        .onMessage(CommandEnvelopes.InOut.
InternalCreateSingleLineInOutEnvelope.class,
this::internalCreateSingleLineInOut)

        .onMessage(CommandEnvelopes.InOut.InternalCompleteEnvelope.c
lass, this::internalComplete)
        .onMessage(CommandEnvelopes.InOut.
InternalVoidEnvelope.class, this::internalVoid)
        .build();
    }
}
```

```

    private Message
internalCreateSingleLineInOut(CommandMessage<CommandEnvelope
s. InOut.InternalCreateSingleLineInOutEnvelope> ce) {
    try {
        InOutCommands.InternalCreateSingleLineInOut cmd
= ce.getCommand().getData();
        inOutApplicationService.when(cmd);
        return withSuccess();
    } catch (Exception e) {
        return
withFailure(DomainErrorFailure.ofDomainErrorOrThrow(e));
    }
}

    private Message
internalComplete(CommandMessage<CommandEnvelopes.InOut.InternalCompleteEnvelope> ce) {
    try {
        InOutCommands.InternalComplete cmd =
ce.getCommand().getData();
        inOutApplicationService.when(cmd);
        return withSuccess();
    } catch (Exception e) {
        return
withFailure(DomainErrorFailure.ofDomainErrorOrThrow(e));
    }
}

    private Message
internalVoid(CommandMessage<CommandEnvelopes.InOut.InternalVoidEnvelope> ce) {
    try {
        InOutCommands.InternalVoid cmd =
ce.getCommand().getData();
        inOutApplicationService.when(cmd);
        return withSuccess();
    } catch (Exception e) {
        throw e;
    }
}
}

```

---

## DDDML工具生成的Saga命令處理器的Spring Boot配置代碼如下：

---

```
package org.dddml.wms.domain.eventuate.tram;

import
io.eventuate.tram.commands.consumer.CommandDispatcher;
import io.eventuate.tram.messaging.common.*;
import io.eventuate.tram.sagas.participant.*;
import org.springframework.context.annotation.*;
import
org.dddml.wms.specialization.eventuate.tram.EventEnvelope;
import org.dddml.wms.domain.inout.*;
import org.dddml.wms.domain.inventoryitem.*;

@Configuration
@Import(SagaParticipantConfiguration.class)
public class EventuateTramCommandHandlersConfiguration {
    @Bean
    public ChannelMapping channelMapping() {
        return DefaultChannelMapping.builder().build();
    }

    @Bean
    public CommandDispatcher
    inOutCommandDispatcher(InOutCommandHandler target,
    SagaCommandDispatcherFactory sagaCommandDispatcherFactory) {
        return
    sagaCommandDispatcherFactory.make("inOutCommandDispatcher",
    target.commandHandlerDefinitions());
    }

    @Bean
    public InOutCommandHandler
    inOutCommandHandler(InOutApplicationService
    inOutApplicationService) {
        return new
    InOutCommandHandler(inOutApplicationService);
    }

    @Bean
```

```
public CommandDispatcher
inventoryItemCommandDispatcher(InventoryItemCommandHandler
target,

SagaCommandDispatcherFactory sagaCommandDispatcherFactory) {
    return sagaCommandDispatcherFactory.make(
"inventoryItemCommandDispatcher",
target.commandHandlerDefinitions());
}

@Bean
public InventoryItemCommandHandler
inventoryItemCommandHandler(
InventoryItemApplicationService
inventoryItemApplicationService) {
    return new
InventoryItemCommandHandler(inventoryItemApplicationService)
;
}
}
```

---

上面這些代碼都是由**DDDM**工具自動生成的，除了**DDDM**文檔中的內容，我們一行代碼沒寫。下面是需要我們手寫的代碼。

### 12.3.3 需要我們編寫的Saga代碼

下面照著前面設計的Saga執行步驟，編寫一個Saga類（文件CreateOrUpdate-Inventory-ItemSaga.java）：

```
package org.ddm.wms.domain.inventoryitem;

import
io.eventuate.tram.commands.consumer.CommandWithDestination;
import io.eventuate.tram.sagas.orchestration.SagaDefinition;
import io.eventuate.tram.sagas.simpledsl.SimpleSaga;
import org.ddm.wms.domain.documenttype.DocumentTypeIds;
import org.ddm.wms.domain.eventuate.tram.CommandEnvelopes;
import org.ddm.wms.domain.inout.InOutCommands;
import java.math.BigDecimal;
import static
io.eventuate.tram.commands.consumer.CommandWithDestinationBuilder.send;

public class CreateOrUpdateInventoryItemSaga implements
SimpleSaga<CreateOrUpdateInventoryItemSagaData> {

    private
SagaDefinition<CreateOrUpdateInventoryItemSagaData>
sagaDefinition =
    step()
        .invokeParticipant(this::getInventoryItem)
        .onReply(InventoryItemStateDto.class,
this::getInventoryItemOnReply)
    .step()
        .invokeParticipant(this::createSingleLineInOut)
        .withCompensation(this::voidInOut)
    .step()
        .invokeParticipant(this::addInventoryItemEntry)
    .step()
        .invokeParticipant(this::completeInOut)
.build();
```

```

    @Override
    public
SagaDefinition<CreateOrUpdateInventoryItemSagaData>
getSagaDefinition() {
    return this.sagaDefinition;
}

    private CommandWithDestination getInventoryItem(
CreateOrUpdateInventoryItemSagaData sagaData) {
    InventoryItemCommands.CreateOrUpdateInventoryItem c
= sagaData.
getCreateOrUpdateInventoryItem();
    InventoryItemId inventoryItemId = new
InventoryItemId(c.getProductId(), c.getLocatorId(),
c.getAttributeSetInstanceId());

CommandEnvelopes.InventoryItem.GetInventoryItemEnvelope
getInventoryItem = new
CommandEnvelopes.InventoryItem.GetInventoryItemEnvelope();
    getInventoryItem.setData(inventoryItemId);
    return send(getInventoryItem)

.to(CommandEnvelopes.InventoryItem.SERVICE_NAME)
    .build();
}

    private <T> void getInventoryItemOnReply(
CreateOrUpdateInventoryItemSagaData sagaData,
    InventoryItemStateDto inventoryItemState) {
    sagaData.setInventoryItemVersion(inventoryItemState
== null ? null : inventoryItemState.getVersion());
    BigDecimal targetOnHandQty =
sagaData.getCreateOrUpdateInventoryItem().getOnHandQuantity(
);
    sagaData.setEntryOnHandQuantity(targetOnHandQty ==
null ? BigDecimal.ZERO : targetOnHandQty.subtract(
    inventoryItemState == null ||
inventoryItemState.getOnHandQuantity() == null
        ? BigDecimal.ZERO :
inventoryItemState.getOnHandQuantity()
));
}

    private CommandWithDestination createSingleLineInOut(

```

```

CreateOrUpdateInventoryItemSagaData sagaData) {
    InventoryItemCommands.CreateOrUpdateInventoryItem c
    =
    sagaData.getCreateOrUpdateInventoryItem();
    InOutCommands.InternalCreateSingleLineInOut
    createSingleLineInOut = new
    InOutCommands.InternalCreateSingleLineInOut();

    createSingleLineInOut.setDocumentNumber(c.getInOutDocumentNu
    mber());

    createSingleLineInOut.setProductId(c.getProductId());

    createSingleLineInOut.setLocatorId(c.getLocatorId());

    createSingleLineInOut.setAttributeSetInstanceId(c.getAttribute
    SetInstanceId());

    createSingleLineInOut.setMovementQuantity(sagaData.getEntryO
    nHandQuantity());
    createSingleLineInOut.setCommandId("Create " +
    c.getInOutDocumentNumber());
    return send(new CommandEnvelopes.InOut.
    InternalCreateSingleLineInOutEnvelope(createSingleLineInOut)
    )
        .to(CommandEnvelopes.InOut.SERVICE_NAME)
        .build();
}

private CommandWithDestination addInventoryItemEntry(
CreateOrUpdateInventoryItemSagaData sagaData) {
    InventoryItemCommands.AddInventoryItemEntry
    addInventoryItemEntry = new
    InventoryItemCommands.AddInventoryItemEntry();
    addInventoryItemEntry.setInventoryItemId(new
    InventoryItemId(
        sagaData.getCreateOrUpdateInventoryItem().getProductId(),
        sagaData.getCreateOrUpdateInventoryItem().getLocatorId(),
        sagaData.getCreateOrUpdateInventoryItem().getAttributeSetIns
        tanceId()
    )));
}

```

```

        addInventoryItemEntry.setVersion(sagaData.getInventoryItemVersion());
        InventoryItemSourceInfo source = new
        InventoryItemSourceInfo();

        source.setDocumentNumber(sagaData.getCreateOrUpdateInventory
        Item().getInOutDocumentNumber());
        source.setDocumentTypeId(DocumentTypeIds.IN_OUT);
        source.setLineNumber("0");
        source.setLineSubSeqId(0);
        addInventoryItemEntry.setSource(source);

        addInventoryItemEntry.setEntryOnHandQuantity(sagaData.getEnt
        ryOnHandQuantity());
        addInventoryItemEntry.setCommandId(sagaData.
        getCreateOrUpdateInventoryItem().getCommandId());
        return send(new CommandEnvelopes.InventoryItem.
        AddInventoryItemEntryEnvelope(addInventoryItemEntry))

        .to(CommandEnvelopes.InventoryItem.SERVICE_NAME)
        .build();
    }

    private CommandWithDestination completeInOut(
CreateOrUpdateInventoryItemSagaData sagaData) {
    InventoryItemCommands.CreateOrUpdateInventoryItem c
    =
    sagaData.getCreateOrUpdateInventoryItem();
    InOutCommands.InternalComplete completeInOut = new
    InOutCommands.InternalComplete();

    completeInOut.setDocumentNumber(c.getInOutDocumentNumber());
    completeInOut.setVersion(0L);
    completeInOut.setCommandId("Complete " +
    c.getInOutDocumentNumber());
    return send(new
    CommandEnvelopes.InOut.InternalCompleteEnvelope(
    completeInOut))
    .to(CommandEnvelopes.InOut.SERVICE_NAME)
    .build();
}

private CommandWithDestination voidInOut(
CreateOrUpdateInventoryItemSagaData sagaData) {

```

```
        InventoryItemCommands.CreateOrUpdateInventoryItem c
        =
        sagaData.getCreateOrUpdateInventoryItem();
        InOutCommands.InternalVoid voidInOut = new
        InOutCommands.InternalVoid();

        voidInOut.setDocumentNumber(c.getInOutDocumentNumber());
        voidInOut.setVersion(0L);
        voidInOut.setCommandId("Void " +
        c.getInOutDocumentNumber());
        return send(new
        CommandEnvelopes.InOut.InternalVoidEnvelope(voidInOut))
        .to(CommandEnvelopes.InOut.SERVICE_NAME)
        .build();
    }
}
```

---

可以看到，在上面的Saga類的sagaDefinition字段的初始化代碼中，使用Fluent DSL創建了Saga的定義。這個Saga在每一步驟裡都會使用消息命令（以請求/異步響應模式）去調用遠程服務，其實Eventuate Tram Saga還支持定義調用本地服務的步驟，但是在上面的代碼中沒有表現出來。

在上面的代碼中還使用了一個Saga Data類（CreateOrUpdateInventoryItemSagaData），用以表示在Saga執行過程中需要保存的信息，示例如下：

---

```
package org.dddml.wms.domain.inventoryitem;

import java.math.BigDecimal;

public class CreateOrUpdateInventoryItemSagaData {
    private
    InventoryItemCommands.CreateOrUpdateInventoryItem
    createOrUpdateInventoryItem;
```

```
private BigDecimal entryOnHandQuantity;
private Long inventoryItemVersion;
// 省略這三個 fields 的 getter/setter 方法代碼

public CreateOrUpdateInventoryItemSagaData() { }
public
CreateOrUpdateInventoryItemSagaData(InventoryItemCommands.CreateOrUpdateInventoryItem c) {
    this.createOrUpdateInventoryItem = c;
}
}
```

---

也許你已經注意到，在這個**Saga Data**類中，存在一個**inventoryItemVersion**字段，它用於保存**Saga**執行第一步（**getInventoryItem**）查詢得到的庫存單元的版本號，如果庫存單元還不存在，那麼它的值為**null**。在後面執行更新庫存數量的步驟

（**addInventoryItemEntry**）時需要使用這個版本號，因為工具生成的實體的方法代碼會使用它來檢測對庫存單元的更新是否發生了併發衝突。這就是前文所說的，要讓程序員直面重要的問題並給出解決方案。

然後，**InventoryItemApplicationServiceImpl**使用了**Eventuate Tram Saga**的**SagaManager**來實現在**DDDM**中定義的那個**CreateOrUpdateInventoryItem**方法：

---

```
package org.dddml.wms.domain.inventoryitem;

import io.eventuate.tram.sagas.orchestration.SagaManager;
import org.dddml.wms.specialization.EventStore;
import
org.springframework.beans.factory.annotation.Autowired;
```

```
import
org.springframework.transaction.annotation.Transactional;
import java.math.BigDecimal;
import java.sql.Timestamp;

public class InventoryItemApplicationServiceImpl extends
AbstractInventoryItemApplicationService.
SimpleInventoryItemApplicationService {

    @Autowired
    private SagaManager<CreateOrUpdateInventoryItemSagaData>
createOrUpdateInventoryItemSagaManager;

    public InventoryItemApplicationServiceImpl(EventStore
eventStore, InventoryItemStateRepository stateRepository,
InventoryItemStateQueryRepository stateQueryRepository) {
        super(eventStore, stateRepository,
stateQueryRepository);
    }

    @Transactional
    @Override
    public void
when(InventoryItemCommands.CreateOrUpdateInventoryItem c) {
        CreateOrUpdateInventoryItemSagaData data = new
CreateOrUpdateInventoryItemSagaData(c);
        createOrUpdateInventoryItemSagaManager.create(data);
    }

    @Override
    public void
when(InventoryItemCommands.AddInventoryItemEntry c) {
        // 方法的實現代碼略
    }
}
```

---

Saga Manager的create方法會創建Saga的實例  
( Saga Instance )，並將Saga的實例持久化到數據庫  
中，然後執行此實例。這裡的create方法創建的Saga

實例中包含了從參數傳入的Saga數據（數據對象的類型為CreateOrUpdateInventoryItemSagaData）。

需要注意的是，在這個InventoryItemApplicationServiceImpl類裡，通過重寫（Override）工具生成的基類的方法，實現了InventoryItem實體的AddInventoryItemEntry（添加庫存單元條目）方法。

觀察前面編寫的Saga實現代碼，你可能會發現其中存在的問題：這些代碼是直接依賴Eventuate Tram Saga框架的。

對於工具生成的代碼來說，依賴Eventuate Tram或Eventuate Tram Saga並不是什麼問題，因為如果需要更換框架，使用不同的代碼模板重新生成代碼就好了。但是對於需要程序員手寫的代碼，這樣的依賴確實不太完美。

“計算機科學中的每個問題都可以用一個間接層解決。”也許我們應該定義一套自己的Saga接口（抽象），把Eventuate Tram Saga包裝為這套接口的一個實現。

另外，支持使用DDDML定義Saga也許是非常有意義的。也就是說，將前面展示的Java代碼中創建SagaDefinition的那些邏輯挪到DDDML文檔中，且以語言中立的方式來編寫（以更好地滿足技術多樣性的

要求），這種做法是值得考慮的。因為Saga的定義屬於相當重要的業務邏輯，我們希望能夠更多地使用DDDML集中呈現領域中重要的業務邏輯。

關於這個問題，本書中並不打算做進一步的探討，如果讀者感興趣，可以自行嘗試。

### 12.3.4 需要我們實現的實體方法

實現實體方法的那些業務邏輯，除了補償操作以外，其他大部分是我們即使沒有采用最終一致性模型也應該手動編寫的。

實現添加庫存單元條目這一步驟（即 **InventoryItem** 實體的 **AddInventoryItemEntry** 方法）的代碼如下：

---

```
// ...
public class InventoryItemApplicationServiceImpl extends
AbstractInventoryItemApplicationService.SimpleInventoryItemA
pplication
Service {

    @Override
    public void
when(InventoryItemCommands.AddInventoryItemEntry c) {
        CreateOrMergePatchInventoryItemDto
createOrMergePatchInventoryItem = null;
        if (c.getVersion() == null) {
            createOrMergePatchInventoryItem = new
CreateOrMergePatchInventoryItemDto.CreateInventoryItemDto();
        } else {
            createOrMergePatchInventoryItem = new
CreateOrMergePatchInventoryItemDto.MergePatchInventoryItemDt
o();
        }

        createOrMergePatchInventoryItem.setCommandId(c.getCommandId(
));
        createOrMergePatchInventoryItem.setRequesterId(c.getRequeste
rId());
    }
}
```

```

createOrMergePatchInventoryItem.setInventoryItemId(c.getInventoryItemId());

CreateOrMergePatchInventoryItemEntryDto.CreateInventoryItemEntry
    createInventoryItemEntry
        = new
CreateOrMergePatchInventoryItemEntryDto.CreateInventoryItemEntry
    createInventoryItemEntry();

createInventoryItemEntry.setOnHandQuantity(c.getEntryOnHandQuantity());
    createInventoryItemEntry.setSource(c.getSource());
    createInventoryItemEntry.setOccurredAt(new
Timestamp(System.currentTimeMillis()));
    createOrMergePatchInventoryItem.setEntries(
        new
CreateOrMergePatchInventoryItemEntryDto[]
{createInventoryItemEntry});

    if (c.getVersion() == null) {
        createInventoryItemEntry.setEntrySeqId(-1L);
        when((InventoryItemCommand.CreateInventoryItem)
createOrMergePatchInventoryItem);
    } else {
        long minSeqId = getMinEntrySeqId(c);

        createInventoryItemEntry.setEntrySeqId(minSeqId);

        when((InventoryItemCommand.MergePatchInventoryItem)
createOrMergePatchInventoryItem);
    }
}

private long
getMinEntrySeqId(InventoryItemCommands.AddInventoryItemEntry
c) {
    InventoryItemState iis =
get(c.getInventoryItemId());
    if
(c.getEntryOnHandQuantity().add(iis.getOnHandQuantity()).compareTo(BigDecimal.ZERO) < 0) {
        throw new IllegalArgumentException(
"c.EntryOnHandQuantity + inventoryItem.OnHandQuantity < 0");
    }
}

```

```

        long minSeqId = -1L;
        for (InventoryItemEntryState e : iis.getEntries()) {
            if (e.getEntrySeqId().compareTo(minSeqId) <= 0)
        {
            minSeqId = e.getEntrySeqId() - 1;
        }
    }
    return minSeqId;
}

```

---

實現庫存單元實體的三個方法的代碼如下：

---

```

package org.dddml.wms.domain.inout;

import org.dddml.wms.domain.*;
import org.dddml.wms.domain.documenttype.DocumentTypeIds;
import org.dddml.wms.domain.inventoryitem.*;
import org.dddml.wms.specialization.*;
import
org.springframework.transaction.annotation.Transactional;
import java.math.BigDecimal;
import java.sql.Timestamp;
import java.util.*;

public class InOutApplicationServiceImpl extends
AbstractInOutApplicationService.SimpleInOutApplicationService {
    @Override
    public void
when(InOutCommands.InternalCreateSingleLineInOut c) {
        if (c.getDocumentNumber() == null) {
            throw new NullPointerException("c.DocumentNumber
is null.");
        }
        super.when(c);
    }

    public static class InOutAggregateImpl extends
AbstractInOutAggregate.SimpleInOutAggregate {
        public InOutAggregateImpl(InOutState state) {

```

```

        super(state);
    }

    @Override
    public void internalCreateSingleLineInOut(String locatorId,
                                              String productId,
                                              String attributeSetInstanceId,
                                              BigDecimal movementQuantity,
                                              Long version, String commandId, String requesterId,
    InOutCommands.InternalCreateSingleLineInOut c) {
        InOutEvent.InOutStateCreated e =
    newInOutStateCreated(version, commandId, requesterId);

        e.setDocumentStatusId(DocumentStatusIds.DRAFTED);
        InOutLineEvent.InOutLineStateCreated inOutLineStateCreated =
    e.newInOutLineStateCreated("0");
        inOutLineStateCreated.setLocatorId(locatorId);
        inOutLineStateCreated.setProductId(productId);
        inOutLineStateCreated.setAttributeSetInstanceId(
    attributeSetInstanceId);

        inOutLineStateCreated.setMovementQuantity(movementQuantity);
        e.addInOutLineEvent(inOutLineStateCreated);
        apply(e);
    }

    @Override
    public void internalComplete(Long version, String commandId, String requesterId,
    InOutCommands.InternalComplete c) {
        documentAction(DocumentAction.COMPLETE, version,
    commandId, requesterId, null);
    }

    @Override
    public void internalVoid(Long version, String commandId, String requesterId, InOutCommands.InternalVoid c)
    {
        documentAction(DocumentAction.VOID, version,

```

```
    commandId, requesterId, null);  
    }  
}  
}
```

---

在將入庫/出庫單修改為“已完成”或“已取消”狀態的方法（`internalComplete`與`internalVoid`）的實現代碼中，調用了工具生成的`InOutAggregateImpl`基類中的`documentAction`方法，這個方法會觸發狀態機執行轉換，從而修改入庫/出庫單的單據狀態。

對比`DDDM`工具為我們生成的成噸的代碼，我們需要擡起袖子親手乾的活實在是少太多了。基於工具生成的靜態類型的代碼，在`IDE`的幫助下，大多數工作我們都可以輕鬆實現。

## 第13章 RESTful API

前面的章節展示的DDDML工具生成的代碼大部分屬於服務端的領域層代碼，這些代碼實現了領域的業務邏輯。為了能讓遠程客戶方便地使用它們，一般來說，我們還應該提供某種形式的應用程序接口（API）。這些接口可以基於各種協議構建，基於HTTP協議構建的API常常被稱為Web Services，其中佔據著當今主流地位的是所謂的RESTful Web Services，又稱RESTful API——REST<sup>[1]</sup>架構風格的API。

這裡不想過多地討論REST架構風格，讀者可以通過Google找到很多關於REST架構風格以及RESTful API設計的文章。接下來會展示一些使用我們製作的工具根據DDDML描述的領域模型（主要是聚合及領域服務的定義）生成RESTful API的實現代碼細節，以及我們在這個實踐過程中的一些思考和決定。

除了可以使用工具生成RESTful API的服務端代碼以外，還可以生成RESTful API的Client SDK，本章會給出在客戶端使用它們的一些簡單示例。

[1] Representational State Transfer,  
[http://en.wikipedia.org/wiki/Representationalstate transfer](http://en.wikipedia.org/wiki/Representational_state_transfer)。

## 13.1 RESTful API的最佳實踐

使用關鍵字“restful best practices”就能通過Google搜索到不少關於如何應用REST架構風格的建議。

但是，也有一些自稱是RESTful API的其實完全沒有遵循REST架構風格進行設計。並不是說基於HTTP+JSON構建的Web Service非得是嚴格的REST風格，但至少我們應該瞭解常用的HTTP方法的語義。

·**GET**：表示獲取數據。它不會改變資源的狀態，所以這是一個安全的操作。

·**PUT**：表示使用請求發送過來的內容完全代替現有的數據。如果資源還不存在，那麼就創建它。PUT方法應該保證幂等性。

·**PATCH**：表示部分更新現有的數據。提交過來的實體（對象）描述了現有資源的數據與新版本之間的差異。

·**DELETE**：刪除指定的資源。這個操作應該是幂等的。

·**POST**：向指定的資源提交一個實體，通常會引起系統的狀態出現某種變化。

### 13.1.1 沒有必要絞盡腦汁地尋找名詞

想要使用純正的REST架構風格，其中一個難點是如何把各種動作都抽象為對資源（名詞）的CRUD（Create、Retrieve、Update、Delete）。

比如，GitHub的API允許你發送一個PUT請求到/gists/{gistId}/star給gist“加星”，發送一個DELETE請求到/gists/{gistId}/star給gist“去星”（unstar）。這樣的抽象十分的RESTful。

但是，有時候在這種抽象上花太多的心思並不值得，就在URI裡面部分地使用動詞也無所謂，也就是說在REST風格中混搭點RPC風格。

在做好DDD分析之後，我們已經知道領域內有哪些聚合、哪些實體的方法，以及有哪些服務，那麼在默認情況下，代碼生成工具應該能自動地為它們生成RESTful API，不需要大家絞盡腦汁地給它們想什麼“名詞”。

筆者認為應該可以發送一個HTTP PUT請求到如下URL，去關閉一個訂單（假設Order是一個聚合）：

---

```
{BASE_URL}/Orders/{orderId}/_commands/Close
```

---

在這個URL中使用`_commands`是為了避免和Order實體的屬性名稱衝突。默認情況下，我們的工具為實體的方法生成的代碼包含了幂等處理邏輯，所以這裡使用的是HTTP PUT方法。幂等操作可以使實現最終一致性變得更簡單。

對於領域服務，比如之前說過的轉賬服務，我們支持客戶端發送一個HTTP POST請求到如下URL去調用這個服務：

---

```
{BASE_URL}/TransferService/Transfer
```

---

### 13.1.2 儘可能使用HTTP作為封包

經常看到有些號稱是RESTful的API會“再次發明HTTP”。比如，當服務端的代碼發生錯誤或拋出異常時，它們經常會統一返回500的HTTP狀態碼，然後在消息體裡再包含自己定義的狀態碼、異常信息等。甚至還有更誇張的，就是不管服務端對請求的處理正常還是異常，都統一返回200的HTTP狀態碼，消息體則是一個既可以包含正常數據，也可以包含異常信息的聯合（Union）數據結構。

這些做法都可以理解為在“按URI語義理應返回的”數據之外添加封包（Envelope）。其實大部分情況下這都是沒有必要的，因為HTTP協議已經設計好了類似的封包機制。

建議儘可能使用HTTP作為封包。使用HTTP標準，就不用去寫非必需的文檔，只要按照眾所周知的標準做，一切都不言自明。更重要的是，這可能有利於使用很多已有的技術基礎設施，比如說Service Mesh。

筆者個人不太喜歡封包，包括常見的Page（分頁）封包。比如，希望通過發送一個HTTP GET請求到如下URL：

---

{BASE\_URL}/Orders

---

就可以得到訂單的列表，而不是包含訂單列表的一個Page封包。如果這裡返回的是訂單列表的Page封包，那麼很可能和獲取訂單的訂單行項的接口出現不一致的情況。一般來說，一個訂單內的訂單行項是很有限的，所以發送HTTP GET請求到如下URL：

---

{BASE\_URL}/Orders/{orderId}/OrderItems

---

返回的結果是某個訂單的所有訂單行項是合理的，而返回訂單行項的Page封包並不常見。

現實世界中糟糕的“RESTful API”設計比比皆是。下面的示例是某知名電商SaaS提供的一個“商品信息查詢”接口。嚴格來說，這個電商SaaS的開放API採用了一套有點類似JSON-RPC的自定義規範，並沒有宣稱自己提供的接口就是REST風格的。但不管是哪種風格的Web Services，筆者認為其設計中存在的問題都是應該避免的。

這個商品信息查詢接口返回的JSON消息體如下（有大量刪節）：

---

```
{  
  "status": {  
    "status_code": 0,  
    "status_reason": ""  
  },  
  "result": {
```

```
        "result": {
            "itemID": "2258064210",
            "point_price_range": null,
            "collectCount": 0,
            "bg_cate": {
                "path": "休閒娛樂->室內休閒玩樂->其他室內休閒"
            },
            "_imgs": [
                "https://xxx.xxx.com/xxxxx395640-1390204649-
2.jpg"
            ],
            "price": "12.00"
        },
        "status": {
            "status_reason": "",
            "status_code": 0
        }
    }
}
```

---

這個API不管後端服務正常還是異常，統一返回200 HTTP狀態碼。僅是JSON消息體就使用了兩層封包（Envelope），result之內還有result。此外，一眼就能看到命名風格有問題，一會兒是camelCase（如itemID），一會兒是PascalCase（如\_imgs），一會兒又是snake\_case（如point\_price\_range）。

更誇張的是，這個SaaS的開放API中所有的方法，包括修改狀態的方法，都可以使用HTTP GET調用。比如，可以通過發送一個GET請求到這個URL來更新商品信息（注意裡面的方法名xxxx.item.update）：

---

```
https://api.xxxx.com/api?param=
{"purchase_fee":68,"itemID":"2250117463",
"sku":
```

```
[{"id":7376581286,"title":"updatew","stock":33,"price":34,"attr_ids":[],"img":"http://xx.xxxx.com/vshop640-1390204649-1.jpg","status":1,"sku_merchant_code":"updateskummcTest","purchase_fee":2211}],"merchant_code":"merchant_code_2","bigImg": ["http://xx.xxxx.com/xxxxx395640-1390204649-2.jpg","http://xx.xxxx.com/xxxxx395640-1390204649-1.jpg"],"price":0.22,"item_comment":"itemupdatetest.", "attr_list":[]}, "free_delivery":1,"remote_free_delivery":1,"titles":["圖片1","圖片0"],"status":1,"cate_id":"121","stock":22,"detail_id": "id923284842152232256112"}&public={"access_token": "xxxxx","version": "1.3","method": "xxxx.item.update","format": "json","auth_userid": "923284842"}
```

---

需要說明的是，為了展示得更清楚，在上面的URL中，筆者沒有對查詢參數**param**與**public**進行URL編碼（實際請求的時候是需要的）。

### 13.1.3 異常處理

服務端向客戶端返回請求的處理結果時，應該優先使用標準的HTTP狀態碼。要注意的是，有些狀態碼是允許在響應中包含消息體（Body）的，有些則不允許存在消息體。

當服務端處理客戶端請求的過程中發生錯誤時，不建議（但是經常看到）統一返回500 HTTP狀態碼，然後在響應消息體裡面使用“自己的狀態碼”。

在使用了服務框架（比如Apache CXF、Spring MVC等）的RESTful API的服務端實現代碼中，服務方法的返回值類型應該是代碼按正常路徑（Happy Path）執行時要返回的那個POJO（對於Java語言而言）。我們儘可能不要手寫構造HTTP Response（包含狀態碼、錯誤信息等）的代碼。一般來說，RESTful服務框架已經包含了可擴展的異常處理機制。

比如，如果使用的是Spring MVC，那麼可以使用關鍵字“spring mvc異常處理restful”去Google上搜索，很容易找到一些很好的實踐建議。

建議對異常進行分類處理。服務端應該針對內部發生的各種異常類型，向客戶端返回不同的HTTP狀態碼。

在服務端代碼的領域層，如果因為客戶端的請求不符合業務邏輯而拒絕處理，那麼可以拋出 **DomainError** (領域錯誤) 異常。在RESTful層，可以考慮捕獲**DomainError**，然後向客戶端回應40X HTTP 狀態碼。客戶端收到40X狀態碼之後，是沒有必要重試的，因為服務端已經告訴它“重試也沒有用”。

RESTful層捕獲從服務端的底層技術基礎設施（比如因為網絡問題、數據庫暫時不可用等原因）拋出的異常後，可以向客戶端回應50X狀態碼。我們可以使用50X狀態碼表示服務端發生了未預料到的 **Exception**。可以認為這種異常發生後，服務端處於一個未知（Unknown）的狀態。客戶端在收到50X狀態碼後可以考慮重試請求。

## 13.2 聚合的RESTful API

本節主要展示由DDDDML工具生成的RESTful API的Java服務端代碼，這些代碼使用了javax.ws.rs包下的註解，這樣我們就可以使用像Apache CXF<sup>[1]</sup>這樣支持JAX-RS規範的服務框架來實現RESTful Services了。在Spring MVC中存在相似的註解，我們的代碼生成工具也支持生成Spring MVC版本的RESTful API服務端代碼。

繼續使用前文展示過的Car聚合的DDDDML示例，看看生成的Java服務端支持哪些RESTful接口，並瞭解實現這些接口的RESTful層的部分代碼細節。

[1] Apache CXF: An Open-Source Services Framework, <http://cxf.apache.org/>。

## 13.2.1 GET

### 1. 獲取聚合實例的列表

支持通過發送HTTP GET請求到如下URL來查詢Car的實例列表：

---

{BASE\_URL}/Cars

---

為了支持這個方法的調用，服務端生成的RESTful服務方法的代碼如下（以Java為例）：

---

```
@Path("Cars")
@Produces(MediaType.APPLICATION_JSON)
public class CarResource {
    @Autowired
    private CarApplicationService carApplicationService;

    @GET
    public CarStateDto[] getAll(@Context HttpServletRequest
request,
        @QueryParam("sort") String sort,
        @QueryParam("fields") String fields,
        @QueryParam("firstResult") @DefaultValue("0")
Integer firstResult,
        @QueryParam("maxResults")
@DefaultValue("2147483647") Integer maxResults,
        @QueryParam("filter") String filter) {
    try {
        Iterable<CarState> states = null;
        CriterionDto criterion = null;
        if (!StringHelper.isNullOrEmpty(filter)) {
            criterion = JSON.parseObject(filter,
CriterionDto.class);
```

```

        } else {
            criterion =
QueryParamUtils.getQueryCriterionDto(request.getParameterMap()
()).entrySet().stream()
            .filter(kv ->
CarResourceUtils.getFilterPropertyName(kv.getKey()) != null)
            .collect(Collectors.toMap(kv ->
kv.getKey(), kv -> kv.getValue())));
        }
        Criterion c = CriterionDto.toSubclass(criterion,
getCriterionTypeConverter(), getPropertyTypeResolver(),
n ->
(CarMetadata.aliasMap.containsKey(n) ?
CarMetadata.aliasMap.get(n) : n));
        states = carApplicationService.get(
            c,
            CarResourceUtils.getQuerySorts(request.getParameterMap()),
            firstResult, maxResults);

        CarStateDto.DtoConverter dtoConverter = new
CarStateDto.DtoConverter();
        if (StringHelper.isNullOrEmpty(fields)) {
            dtoConverter.setAllFieldsReturned(true);
        } else {

            dtoConverter.setReturnedFieldsString(fields);
        }
        return dtoConverter.toCarStateDtoArray(states);
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
// ...
}

```

---

這裡的URL中支持的查詢參數如下。

·**sort**：用於排序的屬性名稱。多個屬性名稱可以英文逗號分隔。屬性名稱前面有“-”則表示倒序排列。

查詢參數sort還可以多次出現，比如，  
sort=fisrtName&sort=lastN-ame,desc。

- fields**：需要返回的字段（屬性）名稱。多個名稱可以逗號分隔。
- filter**：返回結果的過濾器，後文會進一步解釋。
- firstResult**：返回結果中第一條記錄的序號，從0開始計算。
- maxResults**：返回結果的最大記錄數量。

除了上面這些在方法的註解中出現的查詢參數，其餘查詢參數都被認為是針對實體屬性的查詢規格。下面舉例說明查詢參數的使用方法。

例1，可以發送HTTP GET請求到下面的URL，查詢firstName以“Yang”開頭，age大於18的“人們”：

---

```
http://localhost:8080/people?  
firstName=like(Yang%25) &age=gt(18)
```

---

例2，在URL可以重複使用一個屬性名作為查詢參數名，此做法相當於在SQL中使用in子句，比如：

---

```
http://localhost:8080/people?  
firstName=XXXX&firstName=YYYY&firstName=ZZZZ
```

---

例3，可以通過HTTP GET請求到如下的URL來查詢產品的列表，指定按sort查詢參數進行排序後返回：

---

```
http://localhost:8080/products/_page?  
size=10&productId=gt(1533)&productId=lt(  
9933)&sort=productId,desc&sort=productName&productName=like(  
Test1%25)
```

---

注意，這裡URL中出現的\_page，它的意思是要獲取符合條件的產品列表的分頁封包。而且，這個URL中的排序參數sort還可以寫成如下形式：

---

```
http://localhost:8080/products/_page?  
size=10&productId=gt(1533)&productId=lt(  
9933)&sort=-productId,productName&productName=like(Test1%25)
```

---

例4，支持屬性的“Not Equals”查詢，比如：

---

```
http://localhost:8080/attributes?  
attributeId=notEq(airDryMetricTon)
```

---

筆者還構建了一個用於序列化/反序列化查詢規格(Criterion)的Java類庫，它的Criterion接口在設計上類似於Hibernate中的Criterion( org.hibernate.criterion包 )。在筆者製作的DDDMML工具生成的Java代碼中使用了這個自制的類庫。可以使用它來生成這裡的GET查詢方法使用的filter參數。

當然，我們也製作了這個工具的.NET移植版。按照.NET命名慣例，表示Criterion的接口我們命名為ICriterion。以下是生成一個比較複雜的ICriterion接口實例的示意代碼（C#代碼）：

---

```
private static ICriterion _Test_GetFilter()
{
    var conjunctionAll = Restrictions.Conjunction();
    var c0_IsNull = Restrictions.IsNull("Address");
    var c1_Eq = Restrictions.Eq("City", "Shanghai");
    var c2_Between = Restrictions.Between("CreatedAt",
DateTime.Now.Date, DateTime.Now.AddDays(1));
    conjunctionAll.Add(c0_IsNull);
    conjunctionAll.Add(c1_Eq);
    conjunctionAll.Add(c2_Between);

    var c3_Ge = Restrictions.Ge("CreatedAt",
DateTime.Now.Date);
    var c4_Le = Restrictions.Le("CreatedAt",
DateTime.Now.AddDays(1));
    var c5_And = Restrictions.And(c3_Ge, c4_Le);

    var c6_Disjunction = Restrictions.Disjunction();
    c6_Disjunction.Add(c3_Ge);
    c6_Disjunction.Add(c4_Le);
    c6_Disjunction.Add(c5_And);

    var c7_NotEqProperty =
Restrictions.NotEqProperty("Address", "City");

    var c8_Like = Restrictions.Like("City", "%Shanghai%");

    var c9_IsNull = Restrictions.IsNull("CreatedAt");
    var c10_IsNotNull = Restrictions.IsNotNull("CreatedAt");
    var c11_LtProperty =
Restrictions.LtProperty("CreatedAt", "UpdatedAt");
    var c12_Or = Restrictions.Or(c9_IsNull,
Restrictions.And(c10_IsNotNull,
c11_LtProperty));

    var c15_In = Restrictions.In("City", new object[] {
```

```
    "Beijing", "Shanghai", "Shenzhen", "Guangzhou" });

    conjunctionAll.Add(c6_Disjunction);
    conjunctionAll.Add(c7_NotEqProperty);
    conjunctionAll.Add(c8_Like);
    conjunctionAll.Add(c12_Or);
    conjunctionAll.Add(c15_In);
    var c99_IsNotNull = Restrictions.IsNotNull("CreatedBy");
    var filter = Restrictions.Or(conjunctionAll,
c99_IsNotNull);
    return conjunctionAll;
}
```

---

接口ICriterion的所有實現都可以簡單地序列化為JSON。以下是將上面的方法返回的結果序列化後，得到的JSON的樣子（在這裡，序列化結果中的屬性名使用了PascalCase命名風格）：

```
{
  "Type": "conjunction",
  "Criteria": [
    {
      "Type": "isNull",
      "Property": "Address"
    },
    {
      "Type": "eq",
      "Property": "City",
      "Value": "Shanghai"
    },
    {
      "Type": "between",
      "Property": "CreatedAt",
      "Hi": "2016-06-29T16:51:33.3824763+08:00",
      "Lo": "2016-06-28T00:00:00+08:00"
    },
    {
      "Type": "disjunction",
      "Criteria": [
        {
          "Type": "ge",
          "Property": "CreatedAt",
          "Value": "2016-06-28T00:00:00+08:00"
        },
        {
          "Type": "eq",
          "Property": "CreatedBy",
          "Value": "John Doe"
        }
      ]
    }
  ]
}
```

```
        "Type": "le",
        "Property": "CreatedAt",
        "Value": "2016-06-29T16:51:33.383478+08:00"
    } , {
        "Type": "and",
        "Lhs": {
            "Type": "ge",
            "Property": "CreatedAt",
            "Value": "2016-06-28T00:00:00+08:00"
        } ,
        "Rhs": {
            "Type": "le",
            "Property": "CreatedAt",
            "Value": "2016-06-29T16:51:33.383478+08:00"
        }
    } ],
} , {
    "Type": "not",
    "Criterion": {
        "Type": "eqProperty",
        "Property": "Address",
        "OtherProperty": "City"
    }
} , {
    "Type": "like",
    "Property": "City",
    "Value": "%Shanghai%"
} , {
    "Type": "or",
    "Lhs": {
        "Type": "isNull",
        "Property": "CreatedAt"
    } ,
    "Rhs": {
        "Type": "and",
        "Lhs": {
            "Type": "isNotNull",
            "Property": "CreatedAt"
        } ,
        "Rhs": {
            "Type": "ltProperty",
            "Property": "CreatedAt",
            "OtherProperty": "UpdatedAt"
        } ,
    }
} ,
```

```
    },
    {
        "Type": "in",
        "Property": "City",
        "Values": [
            "Beijing",
            "Shanghai",
            "Shenzhen",
            "Guangzhou"
        ]
    }
}
```

---

我們可以將這個JSON字符串URL編碼之後，作為獲取資源列表的GET請求的filter查詢參數的值。

## 2. 獲取聚合實例的列表的Page封包

雖然筆者並不喜歡封包，但是基於部分前端開發人員強烈要求，我們還是支持發送GET請求到如下URL以獲取Car列表的Page（分頁）封包：

---

```
{BASE_URL}/Cars/_page?page={page}
```

---

為了支持該方法的調用，生成的服務端代碼（Java）如下：

---

```
@Path("_page")
@GET
public Page<CarStateDto> getPage(@Context
HttpServletRequest request,
        @QueryParam("fields") String fields,
        @QueryParam("page") @DefaultValue("0") Integer
page,
```

```
    @QueryParam("size") @DefaultValue("20") Integer
size,
    @QueryParam("filter") String filter) {
try {
    // 省略部分代碼
    Page<PageImpl<CarStateDto>> statePage = new
Page<PageImpl<>>(dtoConverter.toCarStateDtoList(states),
count);
    statePage.setSize(size);
    statePage.setNumber(page);
    return statePage;
} catch (Exception ex) {
    throw DomainErrorUtils.convertException(ex);
}
}
```

---

支持的分頁相關的查詢參數如下。

·**page**：獲取第幾頁（從0開始）。

·**size**：每頁的大小，即**Page size**。

返回的**Page**封包中的屬性包括以下內容。

·**number**：當前頁碼（從0開始）。

·**size**：每頁的大小，即**Page size**。

·**totalElements**：所有頁的總元素數量（總行數）。

·**content**：當前頁的內容。

### 3.獲取單個聚合實例

可以發送HTTP GET請求到如下URL，使用聚合根的ID作為路徑參數去獲取一個Car聚合實例的狀態：

---

```
{BASE_URL}/Cars/{id}
```

---

生成的Java代碼如下：

---

```
@Path("{id}")
@GET
public CarStateDto get(@PathParam("id") String id,
@QueryParam("fields") String fields) {
    try {
        String idObj = id;
        CarState state =
carApplicationService.get(idObj);
        if (state == null) {
            return null;
        }
        CarStateDto.DtoConverter dtoConverter = new
CarStateDto.DtoConverter();
        if (StringHelper.isNullOrEmpty(fields)) {
            dtoConverter.setAllFieldsReturned(true);
        } else {

            dtoConverter.setReturnedFieldsString(fields);
        }
        return dtoConverter.toCarStateDto(state);
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

## 4. 獲取聚合實例的數量

可以通過HTTP GET請求查詢符合某些條件  
( filter ) 的聚合根實例的數量，URL如下：

---

```
{BASE_URL}/Cars/_count?filter={filter_string}
```

---

為支持這個方法而生成的代碼如下：

---

```
@Path("_count")
@GET
public long getCount(@Context HttpServletRequest
request,
                      @QueryParam("filter") String
filter) {
    try {
        long count = 0;
        CriterionDto criterion = null;
        if (!StringHelper.isNullOrEmpty(filter)) {
            criterion = JSONObject.parseObject(filter,
CriterionDto.class);
        } else {
            criterion =
QueryParamUtils.getQueryCriterionDto(request.getParameterMap
());
        }
        Criterion c = CriterionDto.toSubclass(criterion,
getCriterionTypeConverter(),
get.PropertyTypeResolver(),
n ->
(CarMetadata.aliasMap.containsKey(n) ?
CarMetadata.aliasMap.get(n) : n));
        count = carApplicationService.getCount(c);
        return count;
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

## 5. 獲取聚合內部實體的實例的列表

可以發送HTTP GET請求到如下URL，獲取某輛汽車（Car）的車輪（Wheels）信息：

---

```
{BASE_URL}/Cars/{id}/Wheels
```

---

為支持這個方法而生成的代碼如下：

---

```
@Path("{id}/Wheels")
@GET
public WheelStateDto[] getWheels(@PathParam("id") String
id,
                                  @QueryParam("sort") String sort,
                                  @QueryParam("fields") String
fields,
                                  @QueryParam("filter") String
filter,
                                  @Context HttpServletRequest
request) {
    try {
        CriterionDto criterion = null;
        if (!StringHelper.isNullOrEmpty(filter)) {
            criterion = JSON.parseObject(filter,
CriterionDto.class);
        } else {
            criterion =
QueryParamUtils.getQueryCriterionDto(request.getParameterMap
().entrySet().stream()
.filter(kv ->
CarResourceUtils.getWheelFilterPropertyName(kv.getKey()) !=
null)
.collect(Collectors.toMap(kv ->
kv.getKey(), kv -> kv.getValue())));
        }
        Criterion c = CriterionDto.toSubclass(criterion,
getCriterionTypeConverter(), getPropertyTypeResolver(),
n ->
```

```
(WheelMetadata.aliasMap.containsKey(n) ?  
WheelMetadata.aliasMap.get(n) : n));  
        Iterable<WheelState> states =  
carApplicationService.getWheels(id, c,  
  
CarResourceUtils.getWheelQuerySorts(request.getParameterMap()  
));  
        if (states == null) {  
            return null;  
        }  
        WheelStateDto.DtoConverter dtoConverter = new  
WheelStateDto.DtoConverter();  
        if (StringHelper.isNullOrEmpty(fields)) {  
            dtoConverter.setAllFieldsReturned(true);  
        } else {  
  
dtoConverter.setReturnedFieldsString(fields);  
        }  
        return  
dtoConverter.toWheelStateDtoArray(states);  
    } catch (Exception ex) {  
        throw DomainErrorUtils.convertException(ex);  
    }  
}
```

---

## 6. 獲取聚合內部實體的單個實例

可以發送HTTP GET請求到如下URL，獲取某個車輪的狀態：

---

{BASE\_URL}/Cars/{id}/Wheels/{wheelId}

---

為支持這個方法而生成的代碼如下：

---

```
@Path("{id}/Wheels/{wheelId}")  
@GET
```

```
    public WheelStateDto getWheel(@PathParam("id") String
id,
                                  @PathParam("wheelId")
String wheelId) {
    try {
        WheelState state =
carApplicationService.getWheel(id, wheelId);
        if (state == null) {
            return null;
        }
        WheelStateDto.DtoConverter dtoConverter = new
WheelStateDto.DtoConverter();
        WheelStateDto stateDto =
dtoConverter.toWheelStateDto(state);
        dtoConverter.setAllFieldsReturned(true);
        return stateDto;
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

## 7. 獲取派生的實體集合屬性

這裡回顧一下在第11章中展示過的一個DDML示例文檔，文檔描述了Package聚合根擁有兩個派生的、類型為實體PackagePart的集合的屬性，即RootPackageParts和ChildPackageParts：

---

```
aggregates:
  Package:
    # ...
  properties:
    PackageParts:
      itemType: PackagePart
    RootPackageParts:
      itemType: PackagePart
      isDerived: true
      filter:
```

```
CSharp: "e => e.ParentPackagePartId ==  
0"  
  
entities:  
  PackagePart:  
    # ...  
    properties:  
      # ...  
      ParentPackagePartId:  
        referenceType: PackagePart  
        referenceName: ParentPackagePart  
      # ...  
      ChildPackageParts:  
        itemType: PackagePart  
        inverseOf: ParentPackagePart
```

---

筆者製作的DDML工具會為這樣的派生屬性生成相應的RESTful API。想要獲取某個Package（假設其ID為636157963396968305）下的RootPackageParts（作為根結點的PackagePart的集合），可以發送HTTP GET請求到以下的URL：

---

<http://test.localhost:8080/Packages/636157963396968305/RootPackageParts>

---

服務端回應的JSON消息體如下：

---

```
[  
  {  
    "partId": "636157962507732244",  
    "packagePartType": 2,  
    "parentPackagePartId": "0",  
    "active": false,  
    "serialNumber": "Box_578151e0-8816-4b6c-bfdc-  
7ea3007a1418",  
    "materialNumber": "TEST_M_1QWRTYUIOP",
```

```
        "quantity": 1,
        "isMixed": false,
        "rowVersion": "1",
        "packageId": "636157963396968305",
        "createdBy": "0",
        "createdAt": "2016-11-27T06:30:28+08:00",
        "updatedBy": "0",
        "allFieldsReturned": true
    }, {
        "partId": "636157964361898321",
        "packagePartType": 2,
        "parentPackagePartId": "0",
        "active": false,
        "serialNumber": "Box_f7fc5960-f597-4c28-910b-1ba17b290c6",
        "materialNumber": "TEST_M_1QWRTYUIOP",
        "quantity": 2,
        "isMixed": false,
        "rowVersion": "1",
        "packageId": "636157963396968305",
        "createdBy": "0",
        "createdAt": "2016-11-27T06:30:28+08:00",
        "updatedBy": "0",
        "allFieldsReturned": true
    }
]
```

---

為了獲得某個PackagePart的子結點  
( ChildPackageParts )，可以發送GET請求到如下  
URL：

---

<http://test.localhost:8080/Packages/636157963396968305/PackageParts/636157964361898321/ChildPackageParts>

---

服務端回應的JSON消息體如下：

---

```
[{
    "partId": "636157963653896794",
```

```
        "packagePartType": 1,
        "parentPackagePartId": "636157964361898321",
        "active": false,
        "serialNumber": "Piece_11886080-b6c1-47fb-b487-
f1701b827340",
        "materialNumber": "TEST_M_1QWRTYUIOP",
        "quantity": 1,
        "isMixed": false,
        "rowVersion": "1",
        "packageId": "636157963396968305",
        "createdBy": "0",
        "createdAt": "2016-11-27T06:30:28+08:00",
        "updatedBy": "0",
        "allFieldsReturned": true
    } , {
        "partId": "636157964358830704",
        "packagePartType": 1,
        "parentPackagePartId": "636157964361898321",
        "active": false,
        "serialNumber": "Piece_838e8f19-d421-490b-8c92-
19b2ce1fecc2",
        "materialNumber": "TEST_M_1QWRTYUIOP",
        "quantity": 1,
        "isMixed": false,
        "rowVersion": "1",
        "packageId": "636157963396968305",
        "createdBy": "0",
        "createdAt": "2016-11-27T06:30:28+08:00",
        "updatedBy": "0",
        "allFieldsReturned": true
    }
]
```

---

## 13.2.2 PUT

### 1. 創建一個聚合實例

想要“創建”一輛汽車（Car），可以發送PUT請求到如下URL：

---

```
{BASE_URL}/Cars/{id}
```

---

為支持這個方法而生成的代碼（Java）如下：

---

```
@Path("{id}")
@PUT
public void put(@PathParam("id") String id,
CreateOrMergePatchCarDto.CreateCarDto value) {
    try {
        CarCommand.CreateCar cmd = value;

        CarResourceUtils.setNullIdOrThrowOnInconsistentIds(id, cmd);
        carApplicationService.when(cmd);
    } catch (Exception ex) {
        logger.info(ex.getMessage(), ex);
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

在上面的代碼中，put方法使用的參數類型CreateOrMergePatchCarDto.CreateCarDto實現了CarCommand.CreateCar接口，它的代碼如下（與

## CreateOrMergePatchCarDto.MergePatchCarDto 類的代碼一起列出，有較多刪節 ) :

---

```
package org.dddml.templates.tests.domain.car;

import java.util.*;
import org.dddml.templates.tests.domain.*;

public class CreateOrMergePatchCarDto extends
AbstractCarCommandDto
implements CarCommand.CreateOrMergePatchCar {
    private String description;
    private CreateOrMergePatchWheelDto[] wheels = new
CreateOrMergePatchWheelDto[0];
    private CreateOrMergePatchTireDto[] tires = new
CreateOrMergePatchTireDto[0];
    private Boolean isPropertyDescriptionRemoved;
    // ...
    // 省略部分 fields
    // 省略 files 對應的 getter/setter 方法
    // ...

    public CarCommand toSubclass() {
        if (COMMAND_TYPE_CREATE.equals(getCommandType()) ||
null == getCommandType()) {
            // 省略代碼
        } else if
(COMMAND_TYPE_MERGE_PATCH.equals(getCommandType())) {
            // 省略代碼
        }
        throw new UnsupportedOperationException("Unknown
command type:" + getCommandType());
    }

    public static class CreateCarDto extends
CreateOrMergePatchCarDto
implements CarCommand.CreateCar {
        public CreateCarDto() {
            this.commandType = COMMAND_TYPE_CREATE;
        }

    @Override
```

```
public String getCommandType() {
    return COMMAND_TYPE_CREATE;
}

@Override
public CreateWheelCommandCollection
getCreateWheelCommands() {
    return new CreateWheelCommandCollection() {
        // ...
    };
}
// ...
}

public static class MergePatchCarDto extends
CreateOrMergePatchCarDto implements CarCommand.MergePatchCar
{
    public MergePatchCarDto() {
        this.commandType = COMMAND_TYPE_MERGE_PATCH;
    }

    @Override
    public String getCommandType() {
        return COMMAND_TYPE_MERGE_PATCH;
    }

    @Override
    public WheelCommandCollection getWheelCommands() {
        return new WheelCommandCollection() {
            // ...
        };
    }
}
// ...
}
```

---

在生成僅在RESTful Client SDK中使用的命令DTO  
( 那些{COMMAND\_NAME}Dto類 ) 的Java代碼時，可以考慮不讓它們實現對應的命令接口

( {COMMAND\_NAME} ) , 這樣就不需要在Client SDK中包含那些命令接口的代碼，可減小SDK包的大小。

## 2. 調用實體的命令方法

通過HTTP PUT方法可以調用在聚合內定義的實體的命令方法。比如，我們可以發送PUT請求到如下URL，調用Car的Rotate方法：

---

```
{BASE_URL}/Cars/{id}/_commands/Rotate
```

---

為支持這個方法而生成的代碼如下：

---

```
@Path("/{id}/_commands/Rotate")
@PUT
public void rotate(@PathParam("id") String id,
CarCommands.Rotate content) {
    try {
        CarCommands.Rotate cmd = content;
        String idObj = id;
        if (cmd.getId() == null) {
            cmd.setId(idObj);
        } else if (!cmd.getId().equals(idObj)) {
            throw DomainError.named("inconsistentId",
"Argument Id %1$s NOT equals body Id %2$s", id,
cmd.getId());
        }
        carApplicationService.when(cmd);
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

默認情況下，客戶端想要修改聚合的狀態，需要在請求中包含**Command ID**以及聚合根的版本號。對於存在消息體（**Body**）的請求，這些信息應該包含在消息體內。

要客戶端傳入聚合根的版本號是為了實現離線樂觀鎖。但是我們也允許在**DDML**中設置忽略併發衝突，這需要在聚合的**metadata**結點下增加鍵值對：**IgnoringConcurrencyConflict:true**，這樣客戶端在調用這個聚合的命令方法時就可以不提供聚合根的版本號了。

### 13.2.3 PATCH

可以通過發送HTTP PATCH請求到如下URL更新Car的聚合實例：

---

```
{BASE_URL}/Cars/{id}
```

---

為了支持這樣的請求生成的Java代碼如下：

```
@Path("{id}")
@org.apache.cxf.jaxrs.ext.PATCH
public void patch(@PathParam("id") String id,
CreateOrMergePatchCarDto.MergePatchCarDto value) {
    try {
        CarCommand.MergePatchCar cmd = value;

        CarResourceUtils.setNullIdOrThrowOnInconsistentIds(id, cmd);
        carApplicationService.when(cmd);
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

這裡的  
`CreateOrMergePatchCarDto.MergePatchCarDto`類實現了第11章中展示的`CarCommand.MergePatchCar`接口。

雖然這裡使用了MergePatch這個說法，但是展示的（由作者製作的DDML工具生成的）RESTful API並

沒有真正支持JSON Merge Patch規範。這裡的PATCH就是一個普普通通的Patch，客戶端不需要在請求中添加如下的頭信息：

---

Content-Type: application/merge-patch+json

---

如果想要實現“原汁原味”的JSON Merge Patch呢？以下服務端的實現思路。假設服務端收到的是“純正”的JSON Merge Patch消息，我們不一定需要將其反序列化為一個動態對象（比如Java的Map，C#的IDictionary），仍然可以將這樣的JSON消息反序列化為靜態類型的命令對象以方便後續使用。以C#為例，可以考慮生成如下的靜態類型代碼：

---

```
//...
public class MergePatchCarDto : CreateOrMergePatchCarDto
{
    public virtual bool IsPropertyDescriptionRemoved { get;
set; }

    public override string Description
    {
        get { return base.Description; }
        set
        {
            if (value == null)
            {
                IsPropertyDescriptionRemoved = true;
            }
            base.Description = value;
        }
    }
//...
```

```
}  
//...
```

---

如果使用**ASP.Net Web API**來實現**RESTful Services**，可能需要通過如下方式設置服務應用的**JsonFormatter**：

---

```
config.Formatters.JsonFormatter.SerializerSettings.NullValueHandling = NullValueHandling.Include;
```

---

我們可以直接在修改**Car**聚合的**RESTful API**方法的參數中使用**MergePatchCarDto**這個靜態類型，示例如下（**C#**代碼）：

---

```
[HttpPatch]  
public void Patch(string id, [FromBody]MergePatchCarDto  
value)  
{  
    CarsControllerUtils.SetId(id, value);  
    _carApplicationService.When(value as IMergePatchCar);  
}
```

---

另外，筆者的前同事製作的**DDML**代碼生成工具曾嘗試過生成真正支持**JSON Patch**以及**JSON Merge Patch**規範的**Java**版本的**RESTful Services**，生成的代碼在這裡不做展示。

## 13.2.4 DELETE

支持客戶端通過發送DELETE請求到如下URL刪除一個Car的聚合實例：

---

```
{BASE_URL}/Cars/{id}?commandId={commandId}&version={version}&requesterId={requesterId}
```

---

為支持這個方法而生成的Java代碼如下：

---

```
@Path("{id}")
@DELETE
public void delete(@PathParam("id") String id,
                    @NotNull @QueryParam("commandId") String commandId,
                    @NotNull @QueryParam("version") @Min(value = -1) Long version,
                    @QueryParam("requesterId") String requesterId) {
    try {
        CarCommand.DeleteCar deleteCmd = new DeleteCarDto();
        deleteCmd.setCommandId(commandId);
        deleteCmd.setRequesterId(requesterId);
        deleteCmd.setVersion(version);

        CarResourceUtils.setNullIdOrThrowOnInconsistentIds(id, deleteCmd);
        carApplicationService.when(deleteCmd);
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

因為**HTTP DELETE**請求一般來說是不應該攜帶消息體的，所以需要通過查詢參數來傳遞聚合根的版本號等信息。

### 13.2.5 POST

默認情況下，DDDML代碼生成工具不會生成使用HTTP POST創建聚合實例的RESTful API代碼。但是，如果需要，也可以支持客戶端在不提供（不知道）聚合根的ID的情況下，通過POST方法創建聚合實例。這時，DDDML文檔中該聚合的metadata結點下的設置如下：

```
aggregates:
  Car:
    # ...
    metadata:
      CreationWithoutIdEnabled: true
      HttpPostCreationEnabled: true
```

從metadata結點中Key的命名就可以大致猜到：

- 當CreationWithoutIdEnabled為true時，表明客戶端創建Car聚合實例時不需要提供ID。那麼後端/服務端需要自己實現這個實體的ID的生成邏輯。
- 當HttpPostCreationEnabled為true時，表示需要支持客戶端使用HTTP POST方法創建Car聚合實例。

這樣設置後，在服務端的代碼中添加Car的ID生成邏輯（這部分代碼不在這裡展示），客戶端就可以發送HTTP POST請求到如下URL去創建Car的實例了：

---

```
{BASE_URL}/Cars
```

---

生成的支持POST方法的服務端代碼如下：

---

```
@POST
public String post(CreateOrMergePatchCarDto.CreateCarDto
value,
@Context HttpServletResponse response) {
    try {
        CarCommand.CreateCar cmd = value;
        String idObj =
termApplicationService.createWithoutId(cmd);

        response.setStatus(HttpStatus.SC_CREATED);
        return idObj;
    } catch (Exception ex) {
        throw DomainErrorUtils.convertException(ex);
    }
}
```

---

不過，我們曾碰到有些客戶端開發人員習慣用**POST**而不是**PUT**來創建實體，即使在默認情況下我們要求在客戶端發送過來的請求的消息體中必須包含聚合根的**ID**。為了增加對這些客戶端的兼容性，這裡又增加了一個限界上下文（**BoundedContext**）的全局設置：

---

```
configuration:
metadata:
    HttpPostCreationWithIdEnabled: true
```

---

當`HttpPostCreationWithIdEnabled`為`true`時，會給所有聚合生成支持“使用`POST`方法創建聚合實例”的接口。默認情況下，需要客戶端在`POST`過來的消息體中提供聚合根的**ID**，這個方法服務端的實現邏輯和使用`PUT`創建的實現邏輯是一樣的。

## 13.2.6 事件溯源API

對於啟用了事件溯源的聚合，我們的工具會在 **RESTful API** 中生成獲取事件溯源信息的查詢方法。比如，客戶端可以發送 **HTTP GET** 請求到如下 **URL**，獲得某個產品的實例的創建事件：

---

```
{BASE_URL}/Products/{productId}/Events/-1
```

---

可以通過請求以下 **URL** 獲取創建某個產品後第一個版本的歷史狀態：

---

```
{BASE_URL}/Products/{productId}/_historyStates/0
```

---

可以通過請求以下 **URL** 獲取在某個產品處於第一個版本 (**Version**為0) 的狀態下，發生的改變其狀態的事件信息：

---

```
{BASE_URL}/Products/{productId}/Events/0
```

---

可以通過請求以下 **URL** 獲取某個產品的第二個版本的歷史狀態：

---

```
{BASE_URL}/Products/{productId}/_historyStates/1
```

---

### 13.2.7 樹的查詢接口

對於在DDML中定義的樹結構，代碼生成工具會在RESTful API層為它們生成一些查詢接口。

對於在第9章中舉例的“貨位樹（LocatorTree）”，可以通過向以下URL發送HTTP GET請求，獲取作為根結點的貨位：

---

```
{BASE_URL}/LocatorTrees
```

---

我們可以發送HTTP GET請求到如下URL，獲取某個父貨位下的子貨位：

---

```
{BASE_URL}/LocatorTrees?parentId={parentLocatorId}
```

---

使用單獨的結構類型構造的樹，比如前文舉例的組織樹（OrganizationTree），可以使用其結構類型的屬性作為查詢參數來進行查詢。比如，我們可以訪問如下URL，獲取“分銷渠道（DISTRIBUTION-CHANNEL）組織樹”的那些根結點組織的信息：

---

```
http://localhost:8080/OrganizationTrees?  
id.organizationStructureTypeId=  
DISTRIBUTION-CHANNEL
```

---

## 13.3 服務的RESTful API

在我們開發的某個CRM應用中，曾提供一個線索跟進服務，這個服務的DDML描述大致如下：

```
services:
  LeadFollowupService:
    methods:
      UpdateAfterFollowup:
        parameters:
          LeadId:
            type: id-long
            referenceType: Lead
          Salutation:
            type: name
          FirstName:
            type: name
          LastName:
            type: name
          Gender:
            type: indicator
          ScheduledContactDate:
            type: DateTime
          PreferredContactPhoneNumber:
            type: id
          LeadFollowupNote:
            type: comment
        #
      ...
    ...
  ...
}
```

DDML工具生成的服務端的RESTful API代碼如下（這裡的代碼使用了Spring MVC作為RESTful服務框架）：

```
package org.dddml.crm.restful.resource;

import java.util.*;
import
org.springframework.beans.factory.annotation.Autowired;
// 省略部分代碼

@RequestMapping(path = "LeadFollowupService", produces =
MediaType.APPLICATION_JSON_VALUE)
@RestController
public class LeadFollowupServiceResource {
    @Autowired
    private LeadFollowupApplicationService
leadFollowupApplicationService;

    @PostMapping("UpdateAfterFollowup")
    public void updateAfterFollowup(@RequestBody
LeadFollowupServiceCommands.UpdateAfterFollowup
requestContent) {

    leadFollowupApplicationService.when(requestContent);
    }
}
```

---

在默認情況下，我們不確認這個服務方法的實現是不是幂等的，所以這裡生成的是**HTTP POST**方法。客戶端可以發送**HTTP POST**請求到如下URL調用這個服務方法：

---

{BASE\_URL}/LeadFollowupService/UpdateAfterFollowup

---

## 13.4 身份與訪問管理

在開發應用時，我們經常聽到的“用戶管理”其實指的是對訪問資源服務器的客戶端進行身份與訪問管理（Identity and Access Management），簡稱IAM。

有必要對身份管理（IM）與訪問管理（AM）的職責進行區分，也就是說要搞清楚它們分別都管什麼（以及不管什麼）：

- 身份管理的核心問題是搞清楚客戶端所代表的用戶“是什麼人”或“不是什麼人”以及客戶端是不是可以代表用戶，然後做出“聲明”。

- 訪問管理的核心問題是解決代表用戶的客戶端“能幹什麼”或“不能幹什麼”的問題。

混淆身份管理與訪問管理會產生很多糟糕的代碼。比如微軟的ASP.NET Membership框架一做好多年，被開發者群嘲說是“人盡皆知”的Sucks。後來微軟不得不重新搞了個ASP.Net Identity框架，這個框架聲稱自己是Claims-Based Identity。關於Claims-Based Identity，可以參考維基百科的詞條[\[1\]](#)。

在IM上下文中不應該存在訪問控制的邏輯，也就是說，將聲明（Claims）映射到用戶“能幹什麼”，這不是Identity Management應該管的事情。角色

( **Role** ) 不應該是IM上下文的關鍵概念，就算在IM上下文的代碼中出現了Role這個詞，那麼它也只應該是一個Claim ( 聲明 ) 的名稱，除此之外沒有更多特別之處。

實現IAM的方法之一，是使用類似OAuth 2.0<sup>[2]</sup>的思路：

- 構建一個頒發Token ( 令牌 ) 的服務器，在OAuth 2.0規範中，這個服務器被稱為Authorization Server。
- 客戶端向這個服務器提供資源所有者的授權證明 ( 比如用戶名、密碼等 )，獲得一個經過Authorization Server簽名的Token。
- 這個Token的Claims中一般包含Subject ( 可能是用戶的ID )、Issuer ( 發行者 )、過期時間以及其他經過認證的聲明。
- 客戶端在訪問資源時，需要向資源服務器提供這個Token。資源服務器檢查這個Token時，一般是通過數字“簽名”來判斷它是否可信，如果認為可信，那麼根據Token中的Claims來決定客戶端“能幹什麼”或“不能幹什麼”。

在本節接下來的討論中，假設DDML工具生成的代碼是使用OAuth2.0 Bearer Token+JWT來實現IAM

的。

[1] 見 [https://en.wikipedia.org/wiki/Claims-basedidentity](https://en.wikipedia.org/wiki/Claims-based_identity)。

[2] The OAuth 2.0 Authorization Framework, <https://tools.ietf.org/html/rfc6749>。

### 13.4.1 獲取OAuth 2.0 Bearer Token

JSON Web Token ( JWT ) 是一個開放標準 ( RFC 7519<sup>[1]</sup> ) 的Claims的表述方法。可以在OAuth 2.0 Bearer Token中使用JWT，在網站Oauth.net上有OAuth 2.0 Bearer Token使用方法的介紹<sup>[2]</sup>。



提示一個Bearer Token是一個“不透明的”字符串，不打算具有任何讓客戶端可以使用的意義。有些服務器分發的tokens是以十六機制字符表示的短字符串，而有些服務器使用結構化的tokens，比如JSON Web Tokens。

客戶端可以通過不同的授權方式獲得OAuth 2.0 Bearer Token，其中通過密碼 ( password ) 授權方式 ( grant\_type ) 向Authorization Server獲得Token的代碼如下 ( C#代碼 )：

---

```
public string GetOAuthBearerToken(string loginId, string password)
{
    var client = new HttpClient { BaseAddress = new
    Uri(AuthzServerEndpointUrl) };
    var url = "oauth2/token";
    string client_id =
    ConfigurationManager.AppSettings["self.ClientId"];

    var postContent = new Dictionary<string, string>();
    postContent["grant_type"] = "password";
    postContent["username"] = loginId;
```

```

postContent["password"] = password;
postContent["client_id"] = client_id;

var req = new HttpRequestMessage(HttpMethod.Post, url);
req.Content = new FormUrlEncodedContent(postContent);
var response =
client.SendAsync(req).GetAwaiter().GetResult();
if (!HttpStatusCode.OK.Equals(response.StatusCode))
{
    throw new
ApplicationException("!HttpStatusCode.OK.Equals(response.Sta
tusCode)");
}

dynamic result = response.Content.ReadAsAsync< JObject>
(new MediaTypeFormatter[] { new JsonMediaTypeFormatter()
}).GetAwaiter().GetResult();

return (string)result.access_token;
}

```

---

注意，這裡消息體的編碼方式是“Form URL Encoded”。

服務器返回的JWT的Claims如下：

```

{
    "sub": "TEST-USER-1",
    "exp": 1539273021,
    "iss": "ISSUER-1",
    "aud": "wms",
    "role": "SystemAdministrator,LocatorManagement",
    "user_groups": "WAREHOUSE1,WAREHOUSE2"
}

```

---

其中sub、exp、iss、aud這幾個名稱是註冊的Claim名稱 ( Registered Claim Names ) ，而這裡的

role和user\_groups是私有的Claim名稱 ( Private Claim Names ) 。

[1] JSON Web Token (JWT),  
<https://tools.ietf.org/html/rfc7519>。

[2] OAuth 2.0 Bearer Token Usage,  
<https://oauth.net/2/bearer-tokens/>。

### 13.4.2 在資源服務器上處理授權

以上是客戶端獲取OAuth 2.0 Bearer Token的示例代碼。接下來看看在資源服務器上如何處理方法的安全性聲明。

以下示例說明在DDDML中聲明客戶端調用一個方法時需要得到的授權（Roles或Permissions）：

```
aggregates:  
  Locator:  
    # ...  
  methods:  
    Create:  
      authorizationEnabled: true  
      requiredAnyRole: [SystemAdministrator]  
      requiredAnyPermission: [LocatorManagement]  
      # ...
```

DDDML工具在生成RESTful API的服務端代碼時應該使用這些安全性的聲明。

#### 1. 使用ASP.Net Web API的Authorize特性

基於ASP.Net Web API實現RESTful Services的時候，可以利用AuthorizeAttribute這個特性（Attribute，相當於Java的註解）來對客戶端的授權情況進行檢查。

ASP.Net Web API內置的這個Attribute只具備檢查用戶是否屬於某個角色（“Is In Roles”）的能力。有人吐槽說：這簡直不是活在現實世界中，對於稍微複雜一點的應用，授權需要在Action/Permission這個級別進行。

為了實現Permission級別的授權，一個簡單的做法是把Permission當成Role來處理。把Permission當成Role來處理並不是筆者的獨創，在StackOverflow上就有類似的建議。如果我們決定這麼做，那麼“Authorization server”在生成JWT Token的時候，只要合併一下Roles和Permissions，並把它們都作為Role這個Claim的值就可以了。然後，就可以在Web API方法上通過如下方式使用AuthorizeAttribute了：

---

```
[HttpPut]  
[Authorize(Roles ="SystemAdministrator, LocatorManagement")]  
public void Put(string id, [FromBody]CreateLocatorDto value)  
{  
    // ...  
}
```

---

 提示在C#中使用特性時，可以把特性名稱中的Attribute後綴省略掉，所以[AuthorizeAttribute]又可以寫作[Authorize]。

## 2. 使用Spring Security的PreAuthorize註解

如果在以Java實現的**RESTful API**服務端代碼中使用**Spring Security**框架，那麼代碼生成工具可能會根據**DDML**中方法的安全性聲明，在**RESTful**服務方法前添加**Spring Security**的註解，類似如下形式：

---

```
@PreAuthorize("hasAnyAuthority('SystemAdministrator',  
'LocatorManagement')")  
@PutMapping("{locatorId}")  
public void put(@PathVariable("locatorId") String locatorId,  
@RequestBody CreateLocatorDto value) {  
    // ...  
}
```

---

有些權限管理的需求可能不太容易以註解的方式實現，有時候我們可能需要手動編碼獲取JWT的信息來實現權限的控制。

比如，一個需求可能是“只有某個倉庫的管理員，才能新建該倉庫的貨位”。如果使用了**Spring Security**框架，我們可能會採用如下方式手動編寫代碼來實現這個功能：

---

```
@PostMapping  
 @ResponseStatus(HttpStatus.CREATED)  
public String post(@RequestBody  
CreateOrMergePatchLocatorDto.  
CreateLocatorDto value,  HttpServletResponse response) {  
    Jwt jwt = ((JwtAuthenticationToken)  
SecurityContextHolder.getContext().getAuthentication()).getT  
oken();  
  
    if(!jwt.getClaims().get("user_groups").toString().contains(v  
alue.getWarehouseId())) {
```

```
        throw new AccessDeniedException("You CANNOT create
locator of this warehouse.");
    }
    // 省略下面的代碼
}
```

---

在上面的代碼中，假設user\_groups這個Claim的值是用戶“可管理的倉庫的ID”的列表。

## 13.5 生成Client SDK

除了可以生成服務端的**RESTful API**層的代碼（**Java**或**C#**）以外，我們製作的**DDDML**工具還可以生成若干種語言的**RESTful Client SDK**。在為靜態語言（比如**Java**、**C#**等）生成的**Client SDK**中包含並使用各種對象（包括狀態對象、命令對象、事件對象等）的靜態類型是絕對有必要的。在**IDE**的幫助下，靜態類型可以幫助開發人員極大地提升開發效率。

回想多年以前那些使用**SOAP Web Services**開發的軟件項目，如果當時沒有工具可以從**WSDL**生成靜態類型的代碼，筆者絕不願意參與其中。即使基於**JSON**的**RESTful Web Services**相對於**SOAP Web Services**已經大大簡化，但是對於稍微複雜的應用，大約也沒有幾個前端開發人員在經歷過使用靜態類型的便利後還願意退回去使用動態類型（這裡說的動態類型，可以理解為**Java**中的**Map**、**.NET**中的**IDictionary**這樣的集合）。

我們在前文已經看到，在默認情況下，工具生成的服務端**RESTful API**層的代碼中，輸入參數以及返回結果的類型都是靜態類型（它們一般帶有**Dto**後綴），這些靜態類型也是生成**Client SDK**的基礎。

### 13.5.1 創建聚合實例

當客戶端需要創建一個聚合實例時，可以使用包含在Client SDK中的靜態類型來生成一個命令對象。

以附錄DDML示例中的Person聚合為例，下面的方法創建了一個CreatePersonDto命令對象（以下展示的是C#代碼，Java版的代碼也是差不多的）：

---

```
static CreatePersonDto GetCreatePersonCommand()
{
    CreatePersonDto person = new CreatePersonDto();
    //person.PersonId = new PersonId(new
    PersonalName("Yang", "Jiefeng"), 1);
    person.Email = "yangjiefeng@gmail.com";
    person.Titles = new string[] { "Programer" };

    CreateYearPlanDto yearPlan2020 = new
    CreateYearPlanDto();
    yearPlan2020.Year = 2020;
    yearPlan2020.Description = "Do something.";

    CreateYearPlanDto yearPlan2021 = new
    CreateYearPlanDto();
    yearPlan2021.Year = 2021;
    yearPlan2021.Description = "Do more than something.";

    person.YearPlans = new
    CreateOrMergePatchOrRemoveYearPlanDto[] { yearPlan2020,
    yearPlan2021 };

    CreateMonthPlanDto monthPlan202001 = new
    CreateMonthPlanDto();
    monthPlan202001.Month = 1;
    monthPlan202001.Description = "My 2020/1 month plan.";
    yearPlan2020.MonthPlans = new
    CreateOrMergePatchOrRemoveMonthPlanDto[] { monthPlan202001 }
```

```
};

person.RequesterId = 1111111L; ;
person.CommandId = Guid.NewGuid().ToString();
return person;
}
```

---

把上面方法返回的**CreatePersonDto**命令對象“隨便”用一個常用的**JSON**工具庫序列化，得到的**JSON**如下：

---

```
{
  "commandType": "Create",
  "requesterId": 1111111,
  "commandId": "7b518538-d052-46c2-90d4-4dd55b1ac5d7",
  "email": "yangjiefeng@gmail.com",
  "titles": [
    "Programer"
  ],
  "yearPlans": [
    {
      "commandType": "Create",
      "requesterId": 0,
      "year": 2020,
      "description": "Do something.",
      "monthPlans": [
        {
          "commandType": "Create",
          "requesterId": 0,
          "month": 1,
          "description": "My 2020/1 month plan.",
          "dayPlans": []
        }
      ]
    },
    {
      "commandType": "Create",
      "requesterId": 0,
      "year": 2021,
      "description": "Do more than something."
    }
  ]
}
```

```
        "monthPlans": []
    }
]
}
```

---

把上面的**JSON**作為**HTTP**請求的消息體，通過向以下**URL**發送**PUT**請求，就可以創建一個**Person**聚合實例：

---

```
{BASE_URL}/People/Yang,Jiefeng,1
```

---

## 13.5.2 更新聚合實例

當需要更新Person聚合實例時，可以通過如下方式使用Client SDK中的命令對象（C#代碼）：

```
static MergePatchPersonDto GetMergePatchPersonDto()
{
    MergePatchPersonDto updatePerson = new
    MergePatchPersonDto();
    updatePerson.BirthDate = new DateTime(2020, 1, 1);

    MergePatchYearPlanDto updateYearPlan2020 = new
    MergePatchYearPlanDto();
    updateYearPlan2020.Year = 2020;
    updateYearPlan2020.IsPropertyDescriptionRemoved = true;
    updateYearPlan2020.Active = true;
    updatePerson.YearPlans = new
    CreateOrMergePatchOrRemoveYearPlanDto[] { updateYearPlan2020
    };

    MergePatchMonthPlanDto updateMonthPlan202001 = new
    MergePatchMonthPlanDto();
    updateMonthPlan202001.Month = 1;
    updateMonthPlan202001.Description = "Updated 2020/01
month plan.';

    CreateMonthPlanDto monthPlan202002 = new
    CreateMonthPlanDto();
    monthPlan202002.Description = "My 2020/02 month plan.';

    updateYearPlan2020.MonthPlans = new
    CreateOrMergePatchOrRemoveMonthPlanDto[] {
        updateMonthPlan202001, monthPlan202002
    };

    updatePerson.Version = 1;
    updatePerson.CommandId = Guid.NewGuid().ToString();
    updatePerson.RequesterId = 111111L;
```

```
        return updatePerson;
    }
```

---

把上面方法返回的命令對象（類型為 `MergePatchPersonDto`）用JSON庫序列化，得到的 JSON如下：

---

```
{
    "commandType": "MergePatch",
    "version": 1,
    "requesterId": 111111,
    "commandId": "05805f21-66ed-4ff1-8d32-40c375fd333b",
    "birthDate": "2020-01-01T00:00:00",
    "yearPlans": [
        {
            "commandType": "MergePatch",
            "requesterId": 0,
            "year": 2020,
            "active": true,
            "isPropertyDescriptionRemoved": true,
            "monthPlans": [
                {
                    "commandType": "MergePatch",
                    "requesterId": 0,
                    "month": 1,
                    "description": "Updated 2020/01 month
plan.",
                    "dayPlans": []
                },
                {
                    "commandType": "Create",
                    "requesterId": 0,
                    "month": 0,
                    "description": "My 2020/02 month plan.",
                    "dayPlans": []
                }
            ]
        }
    ]
}
```

---

把這個JSON作為HTTP請求的消息體，通過向以下URL發送PATCH請求即可更新Person聚合實例：

---

```
{BASE_URL}/People/Yang,Jiefeng,1
```

---

### 13.5.3 使用Retrofit2

廣受歡迎的Retrofit<sup>[1]</sup>框架是一個類型安全的HTTP客戶端（A type-safe HTTP client），可見人人都愛靜態類型（靜態類型才是安全的）。

基於DDML工具生成的各種Java靜態類型，我們可以進一步生成Retrofit2客戶端使用的接口。

對於前文展示過的Car聚合的DDML文檔示例，工具生成的Retrofit2客戶端可使用的接口如下（Java代碼，有刪節）：

---

```
package org.dddml.templates.tests.rest.clientinterface;

import java.util.*;
import retrofit2.*;
import retrofit2.http.*;
import org.dddml.support.criterion.*;
import org.dddml.templates.tests.domain.*;
import org.dddml.templates.tests.specialization.*;
import org.dddml.templates.tests.domain.car.*;

public interface CarsClient {
    @Headers("Accept: application/json")
    @GET("Cars")
    Call<List<CarStateDto>> getAll(@Query("sort") String
sort,
    @Query("fields") String fields, @Query("firstResult") int
firstResult,
    @Query("maxResults") int maxResults, @Query("filter") String
filter);

    // ...
}
```

```

@Headers("Accept: application/json")
@GET("Cars/{carId}/Wheels/{wheelId}")
Call<WheelStateDto> getWheel(@Path("carId") String
carId, @Path("wheelId") String wheelId);

// ...
@Headers("Accept: application/json")
@PUT("Cars/{id}")
Call<String> put(@Path("id") String id, @Body
CreateOrMergePatchCarDto.CreateCarDto value);

@Headers("Accept: application/json")
@PATCH("Cars/{id}")
Call<String> patch(@Path("id") String id, @Body
CreateOrMergePatchCarDto.MergePatchCarDto value);

@Headers("Accept: application/json")
@DELETE("Cars/{id}")
Call<String> delete(@Path("id") String id,
@Query("commandId") String commandId, @Query("version")
String version, @Query("requesterId") String requesterId);

@Headers("Accept: application/json")
@PUT("Cars/{id}/_commands/Rotate")
Call<String> rotate(@Path("id") String id, @Body
CarCommandDtos.RotateRequestContent content);
}

```

---

[1] A type-safe HTTP client for Android and Java, <https://square.github.io/retrofit/> .

## 第14章 直達UI

開發大型業務軟件往往需要使用所謂“元數據驅動”的開發框架（平臺）。傳統的元數據驅動的業務軟件開發框架往往是基於關係模型或者ER模型的，而非DDD風格的對象模型。在使用這些框架的時候，開發UI層所需要的工作量經常讓筆者感到困擾。

使用這些框架開發出來的面向最終用戶的應用程序往往包括一個可能叫作管理門戶/管理後臺/控制檯之類的客戶端應用。為了方便，後文把這樣的客戶端應用叫作**Admin UI**。這些應用往往有很多相似的特點，比如相似的功能佈局：界面的左側是分類的功能菜單，點擊菜單後進入一個實體的列表頁，點擊列表中的一項，進入實體的詳情頁……這樣的客戶端很大程度上是為了在PC上運行而設計的，如果在移動端的瀏覽器上使用它們，用戶體驗往往不會太好。但不管怎麼說，我們總是要一遍又一遍地開發這樣乏味的應用。

我們在設計**DDDM**的時候，關注的是它對領域模型的表達能力，也就是怎樣才能凸顯領域中的關鍵概念，並沒有關注如何給軟件的最終用戶提供一個操作界面。所以筆者基本沒有想過把對UI層的描述作為**DDDM**規範的一部分，但還是經常會想到：既然領域模型已經抓住了領域的核心和關鍵，那麼，能不能僅通過**DDDM**對領域模型的描述就推導出一個合理的默

認的Admin UI——即使這個Admin UI的用戶體驗可能不是那麼好但至少基本可用？那些傳統的“元數據驅動”的開發框架沒有這麼做，是不是隻是因為受限於它們使用的建模語言的表達能力？使用它們所要做的UI層的編碼工作很多時候是否只是為了實現某些領域知識與用戶界面（UI）的映射關係而已？

很多時候，筆者得到的答案是肯定的。

接下來介紹我們做過的一些嘗試，也許你會對“從DDML直接生成Admin UI”能做到什麼程度有一個大致的判斷。

## 14.1 兩條路線的鬥爭

在團隊已經使用**DDDML**描述領域模型的情況下，怎麼快速地開發面向最終用戶的客戶端應用？

是直接基於描述領域模型的**DDDML**文檔，還是基於**RESTful API**的服務描述文檔來開發？也就是說，前端開發人員是需要理解**DDDML**描述的那個領域模型呢？還是只需要“知道”後端可以提供的**RESTful API**，實現**UI/UE**所需的其他信息通過其他方式獲得？



提示如果不太理解這裡所說的“服務描述文檔”是什麼，也許可以先去了解一下**WSDL**（網絡服務描述語言，*Web Services Description Language*<sup>[1]</sup>）。

筆者所在的團隊曾經就這個問題有過激烈的討論甚至爭吵，結果好像誰也沒有最終說服誰。後來我們針對這兩種思路都進行了嘗試。按照前一種思路，有人基於**Vue.js**<sup>[2]</sup>框架實驗性地從**DDDML**直接生成了**Admin UI**的實現，不過並沒有最終將其應用到實際的生產環境中。按照後一種思路，我們走得更遠一點，基於**Angular**<sup>[3]</sup>框架開發出了實際用於生產的**Admin UI**。

[1] Web Services Description Language (WSDL)

Version 2.0 Part 1: Core Language,

<https://www.w3.org/TR/wsdl/>。

[2] 見 <https://vuejs.org/>。

[3] 見 <https://angular.io/>。

### 14.1.1 前端“知道”領域模型

我們可以考慮將DDDML描述的領域模型以更方便前端處理的格式（一般是JSON）直接提供給前端應用使用。這些JSON和後端使用基於YAML的DDDML文檔在結構上完全一致，可以遵循同樣的JSON Schema。

 提示我們把基於YAML的DDDML文檔簡寫為YAML/DDDML；基於JSON的DDDML文檔簡寫為JSON/DDDML。

對於前端開發人員來說，這些JSON/DDDML文檔可以被稱為領域模型元數據。在這些領域模型元數據的基礎上，可以再設計另一種元數據來聲明UI元素與領域模型之間的關係，以幫助我們快速地構建應用的用戶界面。我們把後一種元數據稱為領域模型UI層元數據，簡稱UI層元數據，它們其實不是純粹的UI描述，而是屬於“領域模型與前端UI結合的那部分”。比如，UI層元數據可能包括領域中的實體、值對象、領域服務、“樹”等各種“對象”以及它們的“屬性”的標籤文本（Label Text），因為我們在UI上可能並不想顯示它們在DDDML文檔中的名字（name）或者長長的描述（description）。

為了向前端應用提供領域模型元數據，我們可能需要製作合適的工具。因為給前端提供的

JSON/DDDML文檔，其Schema和後端使用的基於YAML的DDDML文檔應該是完全一致的，所以，實際上我們可以使用工具將YAML/DDDML文檔反序列化為內存中的DDDML DOM文檔對象，然後剪裁掉前端不需要的那部分，再使用一個JSON序列化類庫輸出為JSON文檔即可。

筆者採用的具體做法是，在DDDML .NET工具箱中實現一個簡單的Mapper工具（下文的DictionaryMapper），可以將複雜的對象映射（剪裁、轉換）成一個IDictionary（.NET的“字典集合”，相當於Java中的Map）的實例，然後把它交給一個“普通的”JSON庫序列化為JSON文本。

這個工具的接口被設計成所謂的Fluent Interface<sup>[1]</sup>。也就是說，可以通過鏈式調用來聲明它“想要什麼”。因為Fluent Interface是通用語言內部的一種DSL，所以有時又被稱為Fluent DSL。C#的Lambda Expression給Fluent DSL的實現提供了極大的便利。

以下示例說明了如何使用這個Mapper工具（C#代碼）：

---

```
private static void TestMapper1()
{
    var dictMapper =
ObjectUtils.CreateDictionaryMapper<Parent>();

    dictMapper.Select<String>(p => p.Foo, f => f == null ?
```

```

null : f.ToUpperInvariant()).CamelCasePropertyName();

dictMapper.Cascade<Parent.Child>(p => p.Bar).Cascade(c => c.Haha);

dictMapper.CascadeCollection<Parent.Child>(p => p.BarCollection).CamelCaseKey();
    .ItemMapper.Cascade<Parent.Child.ChildChild>(c => c.Chacha).CamelCasePropertyName().Select(cc => cc.Hello);

var parentObj = new Parent()
{
    Foo = "I am foo.",
    BarCollection = new Parent.Child[] {
        new Parent.Child() {
            Haha = "Haha",
            Chacha = new Parent.Child.ChildChild() {
                Hello = "Hello!"
            }
        }
    }.ToDictionary(c => c.Haha),
};

var dict = dictMapper.Map(parentObj);
Console.WriteLine(JsonUtils.SerializeObject(parentObj));
Console.WriteLine("-----");
Console.WriteLine(JsonUtils.SerializeObject(dict));
}

class Parent
{
    public string Foo { get; set; }

    public Child Bar { get; set; }

    public IDictionary<string, Child> BarCollection { get; set; }
}

public class Child
{
    public string Haha { get; set; }

    public ChildChild Chacha { get; set; }
}

```

```
public class ChildChild
{
    public string Hello { get; set; }
}
}
```

---

上面的**TestMapper1**方法中，創建了一個**Parent**類的**DictionaryMapper**，然後定義其映射規則，調用**Map**方法把**Parent**類的實例映射成一個C#的**IDictionary**集合的實例，最後分別打印這個**Parent**實例的**JSON**序列化結果以及映射出來的**IDictionary**實例的**JSON**序列化結果。執行這個方法，控制檯會輸出如下文本內容：

---

```
{
    "Foo": "I am foo.",
    "BarCollection": {
        "Haha": {
            "Haha": "Haha",
            "Chacha": {
                "Hello": "Hello!"
            }
        }
    }
}
-----
{
    "foo": "I AM FOO.",
    "barCollection": {
        "haha": {
            "chacha": {
                "hello": "Hello!"
            }
        }
    }
}
```

---

使用這個工具，我們就可以很方便地向前端輸出  
基於JSON格式的DDML文檔。

[1]

見

<https://www.martinfowler.com/bliki/FluentInterface.html>。

### 14.1.2 前端“只知道”RESTful API

關於開發一個客戶端應用，筆者的看法是：前端開發人員如果能理解領域模型以及使用DDDML這樣的DSL是最好的。但是，有些開發人員可能會覺得他們只需要拿到遵循RAML<sup>[1]</sup>、Swagger<sup>[2]</sup> ( OpenAPI )<sup>[3]</sup>這樣的服務描述規範的RESTful API文檔就夠了，至於用戶界面 ( UI ) 和用戶體驗 ( UE ) 怎麼做，“讓產品經理來跟我們說吧”。

當然，基於我們生成RESTful API使用的一些慣例 ( Conventions )，也能從RESTful API的服務描述中“猜測”出部分領域模型信息。

比如說，當我們在API的服務描述文檔中看到存在如下三個URL模板時：

- {BASE\_URL}/OrderHeaders/{orderId}
- {BASE\_URL}/OrderHeaders/{orderId}/OrderItems
- {BASE\_URL}/OrderHeaders/{orderId}/OrderItems/{itemSeqId}

我們可以大膽猜測OrderHeader是聚合根，而OrderItem是OrderHeader直接關聯的聚合內部實體。

於是我們可以在“訂單詳情頁”展示訂單頭信息的同時，展示一個訂單行項的列表——在PC端，我們可能會使用一個Table組件來展示這個訂單行項列表。基於此猜測得到的領域模型信息八九不離十，這樣就能夠在前端應用中實現部分UI的“自動”生成了。

雖然筆者不太看好這條路線，但是不否認主要基於RESTful API的服務描述來構建客戶端應用也有它的優勢。首先就是那些對業務（領域）沒有興趣的前端工程師會很高興——終於又可以不用去“瞭解業務”了。另外，這樣的解決方案也更通用一些，可以適用於那些沒有使用DDDML進行領域建模的開發團隊。而且，在使用DDDML時，我們可能會更多地專注於描述領域模型中的核心部分，而忽略了非核心部分。比如CQRS模式中的命令模型往往比查詢模型處於更核心的位置，可能存在描述命令模型的DDDML文檔，而查詢模型可能就沒有。這時候，一個基於“RESTful API服務描述”的工具能更有效地構建“與查詢模型相關的”那些用戶界面和客戶端功能。

不過，下文還是繼續討論基於“將領域模型的DDDML描述輸出到前端”，我們可以怎樣快速開發像Admin UI這樣的客戶端應用。

[1] 見<https://raml.org/>。

[2] The Best APIs are Built with Swagger Tools | Swagger, <https://swagger.io/>。

[3] OpenAPI Specification, <https://swagger.io/specification/> °

## 14.2 生成Admin UI

直接基於使用DDDM<sub>L</sub>描述的領域模型，我們可以生成什麼樣的Admin UI？可能大家很容易想到以下示例：

1 ) 假設我們在DDDM<sub>L</sub>中定義了枚舉對象，如果某個實體的屬性類型是枚舉對象的集合，那麼用戶在Admin UI上就應該可以使用“多選框”組件來展示和編輯這個屬性。

2 ) 在DDDM<sub>L</sub>中定義LocatorTree樹結構的例子裡（見第9章），我們在Admin UI中自動生成了如圖14-1所示的界面。

LocatorTrees

LocatorTrees	Locator
一 号库位	locatorId 一号库位
二 号库位	warehouseId 仓库 ID
三 号库位	parentLocatorId
四 号库位	locatorType
五 号库位	priorityNumber 1
六 号库位	isDefault true
七 号库位	x 1
八 号库位	y
九 号库位	z
十 号库位	version 1
十一 号库位	createdBy test@dddml.org
十二 号库位	createdAt 23 20
十三 号库位	updatedBy

LocatorTrees

- 一 号库位
- 二 号库位
- 三 号库位
- 四 号库位
- 五 号库位
- 六 号库位
- 七 号库位
- 八 号库位
- 九 号库位
- 十 号库位
- 十一 号库位
- 十二 号库位
- 十三 号库位

Locator

- locatorId 一号库位
- warehouseId 仓库 ID
- parentLocatorId
- locatorType
- priorityNumber 1
- isDefault true
- x 1
- y
- z
- version 1
- createdBy test@dddml.org
- createdAt 23 20
- updatedBy

## 圖14-1 根據DDML中定義的LocatorTree生成用戶界面

3 ) 對於在DDML中定義的實體的命令方法，可在 Admin UI 上展示該實體的某個實例的地方（比如在實體的“詳情頁”中）生成相應的命令按鈕。當用戶點擊這個命令按鈕時，可根據DDML中該命令方法的參數列表信息生成、彈出方法參數的錄入界面；用戶輸入（或選擇）參數值後，點擊“確認”按鈕即可調用後端服務提供的方法接口。

4 ) 若一個類型為值對象的屬性聲明瞭引用類型（referenceType），則表明該屬性意圖引用某個實體（可以用SQL數據庫的外鍵概念去類比這裡所說的引用）。那麼，我們自然而然會想到，當用戶在編輯這個屬性的時候，可以彈出一個“被引用的實體”的實例列表，讓用戶選擇列表中的某一項（實例）即可，而不是需要用戶使用鍵盤輸入實體的ID。關於這個特性，下面舉一個更復雜的例子。

## 14.2.1 使用referenceFilter

在DDDM中，實體的屬性如果聲明瞭引用類型，那麼還可以使用**referenceFilter**關鍵字進一步聲明對被引用的實體的過濾規則。

比如，以下是一段描述貨位（Locator）實體的DDDM代碼（有刪節）：

```
aggregates:
  Locator:
    properties:
      WarehouseId:
        referenceType: Warehouse
        referenceName: Warehouse
        notNull: true
        # only reference to "Active" warehouse.
        referenceFilter:
          Criterion:
            type: "eq"
            property: "active"
            value: true
```

作為領域模型元數據，我們可以向前端應用提供實體Locator的屬性WarehouseId（這個屬性表示Locator所屬的倉庫的ID）的定義（以JSON格式）：

```
{  
  "referenceType": "Warehouse",  
  "referenceName": "Warehouse",  
  "notNull": true,  
  "referenceFilter": {
```

```
        "Criterion": {  
            "type": "eq",  
            "property": "active",  
            "value": true  
        }  
    }  
}
```

---

當需要用戶錄入“**Locator**所屬的倉庫的**ID**”時，客戶端可以將上面的**JSON**代碼中的**Criterion**的值進行**URL**編碼（**Encode**）：

---

```
{"type": "eq", "property": "active", "value": "true"}
```

---

之後會得到如下字符串：

---

```
%7B%22type%22%3A+%22eq%22%2C+%22property%22%3A+%22active%22%  
2C+%22value%22%3A+%22true%22%7D
```

---

然後把這個字符串作為查詢參數**filter**的值，通過**GET**請求發送到後端服務的如下**URL**上：

---

```
{BASE_URL}/Warehouses?  
filter=%7B%22type%22%3A+%22eq%22%2C+%22property%22%3A+%22act  
ive%22%2C+%22value%22%3A+%22true%22%7D
```

---

這樣，客戶端**App**就可以獲得已經啟用（**Active**）的倉庫列表，將其呈現到**UI**上，讓用戶從中選擇一個倉庫。

## 14.2.2 展示派生的實體集合屬性

假設，在我們構建的某個應用的領域模型中，存在一個叫會議記錄（Minutes of Meeting，MoM）的實體。

一個會議記錄可能有一個或者多個討論事項（Discussion Details），討論事項有如下四種狀態：

- Clarification（說明）。
- Open（開放）。
- Closed（關閉）。
- On Hold（擱置）。

經過討論，大家都認同如下概念：Pending Discussion（未決的討論事項）指的是那些狀態為Open（開放）或On Hold（擱置）的討論事項。

於是，我們在DDML中可以使用一個派生屬性（Derived Property）來描述這個概念，示例如下：

---

```
aggregates:  
  Mom:  
    id:  
      name: MomId  
      type: id
```

```

properties:
  DiscussionDetails:
    itemType: DiscussionDetail
    # ----- 注意這裡 -----
  PendingDiscussionDetails:
    itemType: DiscussionDetail
    # isDerived: ture
    # filter is also a type of derivation logic
    filter:
      Java: d -> d.getStatusId() == "OPEN" ||
d.getStatusId() ==
"ON_HOLD"
    # ...

entities:
  DiscussionDetail:
    id:
      name: LineNumber
      type: number
    properties:
      StatusId:
        type: id

```

---

在這個建模的過程中，我們更多關注的是領域中的知識和概念，而不是UI層面的問題。但是根據上面的DDDM文檔，我們不僅可以生成後端的業務邏輯代碼，還可以直接在前端的Admin UI上呈現如下界面效果：在一個“會議記錄（MoM）詳情”頁面中，我們可以使用兩個Tab（標籤頁）組件。一個Tab的標籤是“Discussion Details”，它呈現所有的討論事項；另一個Tab的標籤是“Pending Discussion Details”，只呈現那些“未決的討論事項”。

要實現上面的功能，某些基於關係模型的應用開發框架可能得大費周章，而我們只需要在DDDM中寫

四行代碼（ PendingDiscussionDetails 結點），而且凸顯了領域概念。

### 14.2.3 使用屬性層面的約束

在後端提供給前端的領域模型元數據 ( JSON/DDDML ) 中，可能包含對象屬性的約束 ( constraints ) 信息。比如，某個實體可能有個屬性叫作 `email`，前端應用可以拿到這個屬性的元數據 ( JSON 代碼片段 )：

```
{  
  "email": {  
    "type": "string",  
    "constraints": [  
      "Email"  
    ]  
  }  
}
```

另外一個例子，前端可能得知某個實體的 ID 是按如下方式定義的 ( JSON 代碼片段 )：

```
{  
  "id": {  
    "name": "locatorId",  
    "type": "string",  
    "constraints": [  
      "numericDashAlphabetic"  
    ]  
  }  
}
```

前端還可以從後端提供的領域模型元數據 ( JSON/DDDML ) 中獲得在界限上下文配置 ( /configuration 結點 ) 中定義的字符串模式：

---

```
{  
  "configuration": {  
    "boundedContextName": "Dddml.Wms",  
    "namedStringPatterns": {  
      "numericDashAlphabetic": "^[0-9][A-Za-z0-9-]*"  
    }  
  }  
}
```

---

**Email**地址的正確格式是大家都知道的，所以可以不用在上面的代碼**namedStringPatterns**結點中定義它的字符串模式。

有了這些關於屬性約束的領域模型元數據，**Admin UI**在用戶輸入屬性值的時候，就可以利用它們在前端直接執行校驗，第一時間提示用戶輸入正確或錯誤了。

#### 14.2.4 使用UI層元數據

當然，僅僅依賴於領域模型元數據來生成Admin UI還是具有很大的侷限性的。因為默認情況下生成的UI，對於某個對象類型，往往只能簡單地綁定到某種默認的UI組件上，而前端的UI組件可能是非常豐富多彩的。

比如，對於一個`LocalDateTime`類型，前端可以使用不同的UI組件來展示和編輯。比如，可以給用戶提供一個簡單的純文本的輸入框，也可以給用戶提供一個複雜的“日期時間選擇器”( `DateTime Picker` )——所以，我們需要使用UI層元數據來“告訴”前端應用我們想要使用什麼樣的UI組件。

又比如，對於聚合內部的（非聚合根）實體，Admin UI的默認展示方式可能是：在該實體的上一級實體的“詳情頁”中嵌入一個Tab（標籤頁），在Tab中使用一個Table（表格）組件來展示這個實體的實例集合。這是一個比較通用的UI展示方案，但是某些時候我們可能想要更簡潔的解決方案，下面就是一個例子。

假設，有這樣一個YAML/DDDML文檔：

---

```
aggregates:
  Term:
    id:
```

```
        name: TermId
        type: string
    properties:
        # ...
    Tags:
        itemType: TermTag
entities:
    # -----
    TermTag:
        id:
            name: TagId
            type: string
            referenceType: Tag
        properties:
            SequenceNumber:
                type: int
    # -----
    Tag:
        id:
            name: TagId
            type: string
        properties:
            Name:
                type: string
                length: 64
```

---

上面展示的DDDML代碼描述的是：

- 有一個**Term**聚合。聚合根**Term**有一個名為**Tags**的屬性，其類型為實體**TermTag**的集合  
( itemType:TermTag ) 。
- TermTag**是**Term**聚合內部的實體。
- Tag**是另外一個聚合的聚合根。

像TermTag這樣用於表示實體之間多對多關係的簡單“關聯實體”其實很常見。在Admin UI上，也許我們希望使用一個輕量級的標籤組件來對Term.Tags屬性進行展示和編輯，而不是使用默認的重量級的表格組件。這裡所說的標籤組件如圖14-2所示。



圖14-2 使用標籤組件展示實體之間的多對多關係

那麼，對於這個例子，我們需要在前端的UI層元數據中聲明一些信息，比如：

- 在關聯實體（這裡是TermTag）中，用於確定其在UI上的展示順序的屬性是哪一個（這裡很可能是SequenceNumber）。
- 顯示在標籤組件上的文本應該取自被引用實體（這裡是Tag）的哪個屬性（這裡可能是Name）。

## 14.2.5 構建更實時的應用

需要說明的是，這一節記錄的是我們想做但是還沒做好的事情——為了提升工具生成的Admin UI的用戶體驗我們應該做的事情。

我們考慮過這樣的問題：用戶在Admin UI應用中創建（或修改）一個實體時，HTTP請求被髮送到服務端，這個請求是一個“命令”。前文展示過的DDML工具生成的RESTful API服務端在處理完“命令”之後，並不會將實體的最新狀態返回給客戶端，那麼，客戶端如果想要準確地知道實體的真實狀態，需要再次向服務端發送GET請求。

我們可能需要修改代碼生成工具，生成比前一章所展示的更復雜的RESTful API層的實現代碼，可能還需要考慮生成服務端推送（Server-Push）相關的代碼。

比如，如果客戶端使用HTTP POST方法請求創建一個聚合根的實例，之前展示的RESTful API服務端代碼的做法是：在成功處理客戶端請求後，返回的HTTP狀態碼是“201 Created”，響應不包括消息體。其實我們可以考慮讓服務端返回HTTP狀態碼“200 OK”，表示請求已成功處理，同時在響應消息體中返回聚合根的ID以及它在服務端的最新狀態。或者，服務端可以考慮先直接返回HTTP狀態碼“202 Accepted”，然後以異

步的方式處理客戶端請求，在聚合根創建完成之後，使用服務端推送機制將所創建的聚合根的狀態推送給客戶端。

甚至，在客戶端當前UI上顯示的實體實例其服務端的狀態被其他客戶端修改之後，最新狀態應該被服務端直接推送到當前客戶端。在不同的業務場景下，客戶端此時選擇的處理策略可能不同，比如，可能直接覆蓋當前UI上顯示的實體的狀態信息，或者彈出窗口詢問用戶，經用戶確認後忽略或覆蓋當前UI上顯示的實體的狀態信息，等等。

## 第四部分 建模漫談與DDD隨想

- 第15章 找回敏捷的軟件設計
- 第16章 說說SaaS
- 第17章 更好的“錘子”

## 第15章 找回敏捷的軟件設計

我們當然是擁護“敏捷軟件開發宣言”的價值觀的：

個體和互動 高於 流程和工具

工作的軟件 高於 詳盡的文檔

客戶合作 高於 合同談判

響應變化 高於 遵循計劃

也就是說，儘管右項有其價值，

我們更重視左項的價值。

但是，現在敏捷好像已經快要變成貶義詞了。

敏捷開發到底出了什麼問題？也許是因為太多“敏捷”團隊在開發軟件的時候，過早地開始編碼？

畢竟，回到二十多年前，軟件工程界的主流思想是：先做出深思熟慮的良好設計。具體的做法是推遲編碼、推遲編碼、再推遲編碼，讓設計階段在軟件開發週期中所佔比例儘可能大，大家認為這有助於提高軟件開發項目的成功率。



注

意這裡的設計不是程序設計語言的“設計”。它指的是軟件設計，也就是在程序員編碼之前應該進行的需求分析、領域建模以及產品定義等軟件開發活動。這裡把需求分析也作為設計工作的一部分，畢竟，筆者贊成DDD創始人Eric Evans的觀點，分析和設計不應割裂，“（分析）模型在構建時就應考慮到軟件實現和設計”。

多年以前我讀過《最後期限》<sup>[1]</sup>，其故事梗概是一個團隊把開始編碼的時間延後、延後、再延後，於是好產品就誕生了。

注意，人家只是主張把開始編碼的時間延後，不是說開發人員在設計階段應該無所事事——開發人員也應該參與設計。這不是沒有道理的，因為軟件開發越往後進行，改變設計需要付出的代價就越大。

但是這樣的高度設計做過頭之後可能會導致分析癱瘓，特別是對那些系統分析能力無法匹配需要解決的領域複雜性的軟件開發團隊而言。也就是說，大家在設計階段可能會身陷領域的複雜性中，越是分析越是覺得“我們還沒有分析清楚”，越分析越畏懼動手開始編碼。其實這時候，不管三七二十一，先開始編碼再說可能更好，也許“做著做著一切就都清楚了”——很多時候，條條大路通羅馬，此路不通（其實未必不通），也許換條路就通了（其實可能是其他的路“也通”）。

設計免不了要寫文檔，大概是因為開發人員都討厭寫文檔吧，於是敏捷漸成潮流。我們看到很多這樣的“敏捷”團隊：預期兩個月的開發項目（當然一開始的時候工期也只能是粗略估算），產品人員花了不到一個星期去做產品設計——好著急，因為客戶等著呢，領導等著呢，程序員們都等著呢。產品設計的結果無非是幾個流程圖與一堆UI原型，除此之外可能什麼都沒有。大家對這些文檔走過場式地評審完之後，就開始進入編碼階段。經常是程序員寫著寫著覺得不對勁，反過來找產品人員，而產品人員則可能質問：“當初評審你們怎麼不說？”……幾番相愛相殺之後代碼終於勉強成功上線，大家心知肚明肩上的技術債又加重了一噸。

對於這樣的團隊，可能需要適時地開一下歷史的倒車，放下“不盡快開始編碼項目就延期了”的恐懼，讓更多的人參與到需求分析與軟件設計中去。畢竟，不管是瀑布還是敏捷都不能改變：軟件開發越到後期，修改設計的成本越高。

越是複雜的軟件，背後越需要有“反映對領域深度認知的模型”作為支撐。不管這個模型是一開始精心設計出來的，還是通過重構代碼逐步演進出來的。問題是，一個反映對領域深度認知的模型真的總是可以通過重構在代碼中逐步呈現出來嗎？我想，現實世界中往往並非如此。

[1] 湯姆·迪馬可. 最後期限. 清華大學出版社, 2002.  
見<https://book.douban.com/subject/1231972/>。

## 15.1 重構不是萬能靈藥

對於一個已經在運行的系統，如果沒有資源可以支持徹底重寫（重新造一個），那麼，補上自動化測試（保證對最終用戶呈現的功能不變），然後重構（小幅、多次地修改代碼），恐怕確實是改進軟件質量唯一可行的方法。

有些不缺資源的團隊，背地裡重寫了整個系統，然後對外吹噓他們完成了多麼了不起的“重構”壯舉，說他們工作的難度堪比“給行駛中的汽車換輪子”，甚至是“給飛行中的飛機換發動機”——鑑於這種事情好像大家都沒有見過，所以“你懂的”……

一味重構往往不足以挽救那些“身患重症、垂死掙扎”的代碼。中醫開方講究“君臣佐使”，很多敏捷的實踐方法不應該是孤立的，需要相互成全。在筆者看來，能稱之為文檔的代碼，是那些經過精心編寫的測試代碼，最好是像BDD工具Cucumber<sup>[1]</sup>使用的Gherkin<sup>[2]</sup>文檔那樣以自然語言編寫的“實例化需求”——它們是文檔，也是需要頻繁運行的測試代碼的組成部分。

在軟件開發的敏捷運動中，我們確實發明了更多的工具，讓程序員可以通過寫代碼——特別使用業務

人員也有望看懂的DSL編寫的代碼，參與到軟件設計中去。

但是那些可以稱之為文檔的代碼在現實中其實並不多見，這對開發人員系統分析和建模能力是有要求的。代碼即文檔，就是把軟件設計的責任更多地放到開發工程師或者測試開發工程師身上，這實際上是變相地保證整個團隊在軟件設計上的投入。這同二十多年前大家主張“推遲編碼”以給軟件的設計階段留出更多時間根本就是異曲同工。

很多開發人員贊成代碼即文檔，不僅僅是因為討厭寫文檔，而是討厭軟件設計的勞心費力。畢竟，代碼即文檔不等於沒有文檔啊！再怎麼討厭文案工作，也不應該輕視系統分析、領域建模。領域模型從來不是固定的一種或幾種格式的文檔，而是文檔“想要去傳達的那個思想”。你確實可以用代碼表達你的設計思想，但是當你需要和開發人員之外的其他人交流設計思想時，代碼，特別是以通用編程語言編寫的代碼，往往具有很大的侷限性。

敏捷宣言主張“工作的軟件勝過面面俱到的文檔”，“響應變化高於遵循計劃”。有些“敏捷”軟件開發團隊以為：減少文案工作有助於早日開發出可以工作的軟件。

更有人聲稱要避免過度設計，卻說不清楚“過度”的“度”在哪裡。他們說，以KISS ( Keep it simple

stupid ) 開始的軟件可以通過響應變化達到幸福的彼岸。在筆者看來，很多時候，這不過是懶惰的藉口。很多軟件，第一次設計如果做不到90分以上，那麼在面對隨後頻發的“變化”時，可能連挽救的價值都沒有。

很多軟件的設計決策需要從一開始就做對，超前的設計是必要和重要的。我們可以看一個與訂單的裝運、支付相關的數據模型設計示例。

[1] BDD Testing & Collaboration Tools for Teams | Cucumber, <https://cucumber.io> 。

[2] Gherkin Syntax, <https://cucumber.io/docs/gherkin/> 。

## 15.2 數據建模示例：訂單的裝運與支付

假設你現在身處一個軟件外包公司，你是一個軟件開發團隊的首席——這個團隊的所有重要決策都是你說了算。這一次，你們要做的是一个電商App的開發項目。甲方說他要的“其實很簡單”，就模仿某電商App來做就行。你們領導拍著胸脯保證兩個星期後交付。

時間如此緊迫，你決定從KISS開始。於是打開了手機上的某個購物App，如圖15-1所示。

10:39



我的亲情账号 >



1941

280

421

17

收藏夹

关注店铺

足迹

红包卡券

会员中心

直 800

专享10元寄快递优惠券

最快1小时上门取件

10 元

无门槛使用

我的订单

查看全部订单 >



待付款



待发货



待收货



评价



退款/售后

最新物流

16:15



运输中

【镇江市】快件已到达 镇江丹阳集散中心

我的频道

全部频道 >



首页



...



消息



购物车



我的

## 圖15-1 某電商App“我的”界面

App裡“我的”頁面中，“我的訂單”被分成了如下類別：

- 待付款
- 待發貨
- 待收貨
- (待)評價
- 退款/售后



提

示假設你的團隊並不熟悉DDD，所以在下文的討論中不會使用DDD特有的概念和術語。

你打算就從這裡開始“借鑑”，設計與訂單相關的數據模型。作為開發團隊的首席，你是這麼決定的.....

### 15.2.1 訂單與訂單行項

你認為，訂單的職責是記錄客戶訂購了什麼東西（也就是“產品”）。

一個訂單可能有多個行項，表示在這個訂單中，客戶訂購了什麼東西，以及多少數量。

比如，張三可能下了兩個訂單，兩個訂單的信息如表15-1所示。

表15-1 訂單與訂單行項

Order (订单)	OrderItemSeqId (订单行项序号)	Product	订单行项数量
19033101 (张三的订单)	1-101	iPhone XR 手机	2
	1-102	火爆小酒	1
	1-103	星爸爸猫爪杯	1
19033102 (张三的订单)	2-101	DELL 显示器	1
	2-102	火爆小酒	4
	2-103	星爸爸猫爪杯	2

然後，你決定給訂單這個實體增加幾個物流和支付相關的屬性。

現在，Order的屬性是這樣的：

- OrderId：訂單的ID。
- IsPaid：是否已支付。
- PaymentNumber：支付編號。
- IsShipped：是否已發運。
- IsReceipted：客戶是否已收貨。
- ShipmentNumber：發貨單編號。

你覺得設計的訂單模型如此清晰明瞭，你都快愛上你自己了……

兩個星期後，系統開發測試完畢。你的團隊對自己編寫的代碼的質量有十二分的自信，它們都遵循互聯網大廠Google使用的代碼規範。你們交付的App也滿足了甲方提出的全部需求。你甚至相信甲方用了它以後，分分鐘可以在電商行業獨佔鰲頭，三年納斯達克上市，走上人生巔峰。

然後，你們拿著系統到甲方公司做演示。

看了你們的演示，甲方倉儲部門的人提出問題：

客戶下的訂單，並不總是能夠一次性地把訂單中的所有東西都發貨給客戶的。也許因為沒有貨，也許

因為要發貨的時候發現倉庫裡的貨已經損壞了，比如屋頂漏水、被雨淋壞了。碰到這樣的情況，我們一般會跟客戶商量，很多時候，我們是需要先把一部分產品發給客戶的。

聽了這個說法，你感覺事情好像有點不妙。

你們領導也參加了會議，他對你說：“把這個需求記錄下來，回去馬上改！”

這時，甲方市場部門的人發話：

能不能加個儲值功能？我們的客戶可以先充值（充值到儲值賬戶），然後用儲值賬戶裡的錢消費。一個訂單，如果儲值賬戶的錢不夠，那麼可以部分使用儲值賬戶支付，部分使用支付寶或者微信支付。對了，做個積分兌換功能應該也很簡單吧？每100個積分能抵一塊錢。還有，應該讓我們可以對一個商品設置積分最多能夠抵扣多少錢。

你們領導說：“這個事情很簡單，一兩天我們就能搞定。”

你的後背開始流汗，你知道這個事情一點都不簡單.....

難熬的會議終於開完了。會後你找到領導，告訴他會議上客戶提出的需求你們最少需要做兩個月。他聽後暴跳如雷，他本來還指望你們一個星期內搞定，

好跟甲方要剩下的錢。現在你居然說最少還要兩個月？？

你很委屈：“這麼多需求，比我們預想的複雜太多，系統幾乎要推翻了重做，你找誰來也不可能一兩個星期內做完。”

領導很生氣，甚至開始懷疑你在訛他：“誰來也不可能？我給你們找個‘外腦’，給你們參謀參謀，你們一起來做這個事情。”

人脈資源豐富的領導立馬抓起電話，請來了一個賢者（Smart Guy，司馬蓋）。

賢者聽了你們的故事，捻鬚微笑，他說：“其實，你們是可以抄作業的。這樣吧，我給你們介紹一下別人做過的模型。”

下面進入賢者時間（以下的話都是賢者說的）。

讓我們先從訂單發貨說起。

## 15.2.2 訂單與訂單裝運組

一個訂單可以分多次發貨 ( Ship ) ，從而形成多個訂單裝運組 ( OrderShipGroup ) 。也就是說，“訂單”和“訂單裝運組”實體是一對多的關係。

訂單裝運組的ID由兩部分組成，OrderId以及OrderShipGroupSeqId ( 訂單裝運組序號 ) 。我們使用訂單裝運組序號來標識“訂單的一次發貨”。

就像訂單有行項，訂單裝運組也有自己的行項，我們就叫它OrderItemShip-GroupAssociation ( 訂單行項與裝運組關聯 ) 吧，它的ID由三部分組成：OrderId、OrderShipGroupSeqId、OrderItemSeqId ( 訂單行項序號 ) 。

像前面張三的兩個訂單，可能需要做一個發貨計劃，具體見表15-2。

表15-2 訂單與訂單裝運組

OrderItemShipGroupAssociation Id. (订单行项与装运组关联 Id.)			OrderItem Ship Group Association Quantity (或剩余数量)	备注 / 说明
Order (订单)	OrderShipGroup (订单装运组)	订单行项序号 (产品)		
19033101 (张三的订单)	SG-1-1 (订单第1次发货)	1-101 (iPhone XR 手机)	2	< 这些有货先发
		1-103 (星爸爸猫爪杯)	1	
	(待定)	1-102 (火爆小酒)	(1)	< 这些先不发
19033102 (张三的订单)	SG-2-1 (订单第1次发货)	2-101 (DELL 显示器)	1	
		2-103 (星爸爸猫爪杯)	1	
	(待定)	2-102 (火爆小酒)	(4)	< 这些先不发
		2-103 (星爸爸猫爪杯)	(1)	< 这些先不发

對了，訂單裝運組存在的意義，還有利於按庫存情況進行庫存保留、備貨、揀貨等操作。這些業務流程的信息，可以考慮記錄在OrderShipGroup及OrderItemShip-GroupAssociation實體中。

既然，這兩個訂單都是張三的，收貨地址都一樣，為什麼我們不一次性發給張三呢？

顯然，我們完全應該支持把多個訂單/訂單裝運組的發貨需求合併到一個裝運單 ( Shipment )，提高作

業效率、節約資源。

### 15.2.3 訂單與裝運單

一個訂單可以多次發貨，一個裝運單 ( Shipment ) 又可以裝運多個訂單/訂單裝運組的“貨”。

於是，“訂單裝運組”和“裝運單”是多對多的關係，自然訂單和裝運單之間也是多對多的關係。

比如，我們可以把上面兩個“訂單裝運組”的東西，一次性發給客戶張三，從而形成一次裝運 ( Shipment )，如表15-3所示。

表15-3 訂單與裝運單

Shipment (装运单)	Shipment Item SeqId (装运项序号)	装运项 数量	OrderItemShipGroupAssociation (订单行项与装运组关联)	
			订单 / 订单装运组 / 订单行项 (产品)	OrderItem ShipGroup Association Quantity
S1903310037 (发给张三的装运单)	SI01	2	19033101 / SG-1-1 / 1-101 (iPhone XR 手机)	2
	SI02	2	19033101 / SG-1-1 / 1-103 (星爸爸猫爪杯)	1
			19033102 / SG-2-1 / 2-103 (星爸爸猫爪杯)	1
	SI03	1	19033102 / SG-2-1 / 2-101 (DELL 显示器)	1

如上所述，訂單與裝運單之間既然是個“多對多”的關係，那麼對於關係模型來說，這需要一箇中間的關聯實體（數據表）。那麼，這個關聯實體怎麼設計？

我們是不是可以給 OrderItemShipGroupAssociation 增加兩個屬性：  
ShipmentId 和 ShipmentItemSeqId 屬性，讓它作為訂單行項與裝運項之間的關聯實體？

另外，OrderItemShipGroupAssociation（訂單行項與裝運組關聯）的數量，其實是個“計劃發貨數量”，那麼，實際的發貨數量又記錄在哪裡？是否要在

OrderItemShip-GroupAssociation裡面增加一個“發貨數量”屬性，用於記錄實際發貨數量？

還有，我們是不是要考慮一下是否存在訂單不存在對應的“裝運單”，只是需要記錄訂單的發貨信息這樣的情況？

這些信息都保存在OrderItemShipGroupAssociation中，這個實體承擔的職責是不是有點多了？乾脆，我們再增加一個實體吧？

### 15.2.4 訂單的項目發貨

於是我們增加了一個獨立的實體，叫 **ItemIssuance**（項目發貨）。

這裡假設我們沒有為**ItemIssuance**選擇使用組合 ID（組合主鍵），它有一個單列的ID（主鍵）。

除了這個ID之外，**ItemIssuance**實體還包括如下屬性：

- **OrderId**：訂單ID（訂單號）。
- **OrderShipGroupSeqId**：訂單裝運組序號。
- **OrderItemSeqId**：訂單行項序號。
- **ShipmentId**：裝運單ID。
- **ShipmentItemSeqId**：裝運單行項序號。
- **Quantity**：（實際的）發貨數量。

上面所列的前三個屬性，指向（Reference）**OrderItemShipGroupAssociation**；屬性**ShipmentId**與**ShipmentItemSeqId**指向裝運行項。

ItemIssuance是處於裝運單與訂單之間的關聯實體，如表15-4所示。

表15-4 裝運單與訂單之間的關聯實體（1）

Shipment (装运单)	Shipment Item	装运行项 数量	ItemIssuance		OrderItemShipGroupAssociation (订单行项与装运组关联)	
			Id	发货 数量	订单 / 订单装运组 / 订单行项 (产品)	OrderItem ShipGroup Association Quantity
S1903310037 (发给张三)	S101	2	IS0001	2	19033101 / SG-1-1 / 1-101 (iPhone XR 手机)	2
	S102	2	IS0002	1	19033101 / SG-1-1 / 1-103 (星爸爸猫爪杯)	1
			IS0003	1	19033102 / SG-2-1 / 2-103 (星爸爸猫爪杯)	1
	S103	1	IS0004	1	19033102 / SG-2-1 / 2-101 (DELL 显示器)	1

沒有使用組合主鍵，ItemIssuance使用起來會更靈活一些。比如像表15-5，很容易將上一個表格的第一行拆成兩行。

表15-5 裝運單與訂單之間的關聯實體（2）

Shipment / Shipment Item	装运 行项 数量	ItemIssuance			OrderItemShipGroupAssociation (订单行项与装运组关联)	
		Id.	发货 数量	发货 详情	订单 / 订单装运组 / 订单行项 (产品)	OrderItem ShipGroup Association Quantity
S1903310037 / SI01	2	IS0001-1	1	SN: XX01XXXX	19033101 / SG-1-1 / 1-101 (iPhone XR 手机)	2
		IS0001-2	1	SN: XX02XXXX		

(续)

Shipment / Shipment Item	装运 行项 数量	ItemIssuance			OrderItemShipGroupAssociation (订单行项与装运组关联)	
		Id.	发货 数量	发货 详情	订单 / 订单装运组 / 订单行项 (产品)	OrderItem ShipGroup Association Quantity
S1903310037 / SI01	2	IS0002	1		19033101 / SG-1-1 / 1-103 (星爸爸猫爪杯)	1
		IS0003	1		19033102 / SG-2-1 / 2-103 (星爸爸猫爪杯)	1
S1903310037 / SI01	1	IS0004	1		19033102 / SG-2-1 / 2-101 (DELL 显示器)	1

如表15-5所示，我們可在ItemIssuance這個實體中記錄更多的實際“發貨詳情”信息，比如發出的產品的序列號 (Serial Number，即SN)。

至此，我們已經構建了與訂單裝運 (發貨) 相關的數據模型。

可以認為，訂單本身其實是沒有“發貨狀態”的。它的發貨狀態，是從其他實體 (Shipment等) 的狀態計

算（派生）出來的。當然，從效率角度考慮，有些派生的狀態計算出來之後可以緩存起來。

一個訂單可能既處於“待發貨”的狀態，同時也處於“已發貨（待收貨）”的狀態。

我們應該可以想象到，這樣的模型可以很好地支持如下業務操作：訂單的一部分產品使用企業自有的庫存來發貨，另一部分產品由供應商“一件代發”。

聰明如你們，可能已經看出來了，上面的數據模型實際上是從開源項目Apache OFBiz借鑑而來的。

搞定了物流的問題，我們再來看看資金流的問題。

## 15.2.5 訂單的支付

訂單支付相關的數據模型，我們也可以看看OFBiz是怎樣設計的：

·**Invoice**表示“付款請求”。有時稱之為發票也無妨，但是這裡不要把**Invoice**單純地理解成我們拿來報銷、抵稅的那個發票，具體請看看**Invoice**詞條的英文解釋。

·**Invoice**和訂單之間是多對多的關係。可以使用**OrderItemBilling**（訂單行項計費）表示**Invoice**和**Order**相關實體之間的多對多關係。除了**OrderItemBilling**實體以外，還有其他“計費”（**Billing**）實體，比如**ShipmentItemBilling**（裝運項計費）實體。

·**Invoice**（請求付款）並不一定會形成真正的付款（**Payment**）。付款請求的款項不一定一次就付完；或者，付款時可以一次付多個付款請求的款項。所以，付款本身需要使用獨立的實體（**Payment**）來表示。

·引入**PaymentApplication**實體，用於表示實際付款（**Payment**）和付款請求（**Invoice**）之間的多對多關係。

在“簡單”的情形下，“訂單”和“付款請求”及“**Payment**”之間，可能是一一對應的關係。但稍微多想

一下就可能會意識到，訂單、賬單、付款，顯然有時候它們不是一一對應、嚴絲合縫的：

- 我（客戶）要買某商品、要買多少，這是訂單。
- 你把賬單給我，要我付款，這是付款請求。
- 付款是Payment。我先付一部分行不行？我一次付多個賬單行不行？一般情況來說，當然可以。誰會有錢不收嗎？

訂單和支付的關係，對比訂單和裝運（物流單據）的關係，其實是有相似的地方的。就像訂單的物流狀態不是訂單本身的狀態一樣，訂單本身沒有支付狀態屬性。如果一定說有，那麼它的支付狀態只能算是一個派生狀態，是從其他實體的狀態計算（派生）出來的。

如果我們按照這個模型來開發，假設一個訂單已支付，那麼其實表示：

- 通過相關的計費（XxxxBilling）實體，可以找到該訂單關聯的付款請求。
- 這些付款請求關聯著已完成的付款記錄，且Payment通過PaymentApplication分配到付款請求的實際付款金額已經等於（或大於）付款請求的應付金額。

和訂單的發貨狀態類似，一個訂單完全有可能處於未支付與已支付的中間狀態——也就是部分支付的狀態。

考慮一下現實存在的組合支付需求：我們可能需要支持客戶使用（在我們的電商系統中的）儲值賬戶的餘額進行支付，當餘額不足的時候，可以使用支付寶或者微信支付另一部分款項。我們還可能使用積分這樣的虛擬貨幣來支付（抵扣）部分款項。

我們的電商系統和支付寶、微信支付之間不可能使用數據庫事務來保證一致性，所以出現不一致的中間狀態不可避免，上面的模型也可以很好地支持最終一致性的實現……

大家可能會問：在項目初期，費心巴力地去構建一個如此“複雜”的模型，有什麼好處？

回頭看看一開始的簡單設計，我們試圖在Order中使用幾個屬性來搞定一切：

- **IsPaid**：是否已支付。
- **IsShipped**：是否已發運。
- **IsReceipted**：是否已收貨。

對比後來我們可能需要增加的實體、引入的概念（表15-6僅列出部分），我們可以問問自己：如果我

們面對的領域的現實就是如此複雜，如果我們只是“簡單地”開始編碼，然後隨著業務發展，被動地響應客戶提出的需求，我們有多大的可能性通過重構得到這樣一個反映對領域深度認知的模型？這樣的模型雖然複雜但仍然清晰、合理、具備良好的概念完整性。這裡說的“得到這樣一個……模型”的意思，不僅僅是“大家知道模型應該是什麼樣的”、把模型記錄在文檔中就完事，而是要保證開發人員重構出來的代碼和這個模型之間存在清晰的映射關係。

表15-6 增加的實體和引入的概念

术语 (英文, 代码中使用)	解释 (中文)
OrderShipGroup	订单装运组
OrderShipGroupSeqId	订单装运组序号
OrderItemShipGroupAssociation	订单行项与装运组关联 (可以理解为订单装运组的行项)
Shipment	装运单
ShipmentItem	装运单行项
ShipmentItemSeqId	装运单行项序号
ItemIssuance	(订单的) 项目发货
Invoice	付款请求 (发票)
{Xxxx} Billing	(各种) 计费
Payment	支付
PaymentApplication	支付应用 (支付分配)

理論上這是可能的，但是，大概是我孤陋寡聞，目前還沒見過這樣的事情發生。

我們是不是可以一直保持簡單？這取決於領域。有時候，領域的現實就是複雜的。具體需要結合領域的現實來判斷。

簡單和複雜是相對的，不知道什麼是複雜，怎麼知道如何保持簡單？

我見過經過數年演進、有800多張表的電商系統——如果以合理的模型實現，大概300張表就可以實現同樣的功能需求，並且系統會更健壯、更具可擴展性——維護這樣的“祖傳代碼”對開發人員來說簡直是一場噩夢。這樣的系統，代碼即使能重構，整個模型也應該被預先、精心、重新設計，然後把設計結果放在團隊所有人都能看到的地方，並以此作為一個目標（立個flag），每一步重構都應該朝著這個目標前進。

很多時候，要獲得一個良好的模型，並不一定需要先被現實打得鼻青臉腫，要做的只是預先學習。我的個人看法是，不管是B to b還是b to C，天底下所有的公司其實都是差不多的。以為自己業務很特別的、值得從零開始去“定製開發”軟件的，大多數時候，其實只是自己“看得少”和“想得少”。

很多團隊聲稱的KISS，不過是偷懶的藉口。如果項目不死，總有一天團隊需要為前面的潦草、Simple、Naive和Stupid埋單。

Keep it simple,smart！閱盡世間繁華，才有資格說自己真心喜歡坐在旋轉木馬上的簡單快樂。

以上，是賢者司馬蓋的話。

## 15.3 中臺是一個輪迴

多年以前，當我們開發ERP這樣的大型業務軟件時，致力於在一個很大的範圍內維護概念完整性。一個未經定製的ERP軟件（所謂的標準版）在安裝之後，數據庫中有七八百張表（意味著幾乎同樣數量的實體）稀鬆平常。也就是說，有七八百個實體處於同一個限界上下文內。

有些ERP軟件在這樣一個巨大的限界上下文內仍然很好地維護了概念完整性，這實在讓人歎服。那麼多的對象，那麼多的對象與對象之間的關係，那麼多的概念，名詞、動詞、形容詞，光是命名，就是一項了不起的成就。

實現這樣規模的概念完整性很了不起，破壞它們卻很容易。一套ERP實施下來，數據庫裡面可能又多了幾十甚至幾百張表，其中使用的命名可能千奇百怪。那些ERP實施顧問與ERP開發工程師們，大多數人都對維護概念完整性這個事情缺乏興趣與動力。

把那些辣眼睛的“定製”代碼和漂亮的“產品”代碼放在一起實在是大煞風景，而且，我們總不能讓單體系統無限長大（“機器受不了”），於是我們又一次祭起“分而治之”大法，像SOA（面向服務的架構）這樣的軟件組件化技術給我們提供了拆分的工具。在概念

上，我們把一個大的限界上下文按照領域拆分成幾個相對來說小一些的限界上下文；在物理上，我們把一個大的單體應用軟件拆分成若干服務組件。一般來說，我們會讓軟件組件的物理邊界和限界上下文的概念邊界基本對齊，一個限界上下文對應一個或少數兩三個可以獨立部署的服務項目，服務項目中包含了限界上下文的核心業務邏輯的實現代碼。開發人員在工作的時候，只需要在IDE上打開一個服務項目的代碼，在這個項目/上下文的邊界內思考問題。服務組件的物理邊界給服務之間的調用增加了障礙，促使開發人員簡化對象之間的關係，編寫更高內聚、更低耦合的代碼。當服務組件不多的時候，構建防腐層的工作量也不會很大，我們只需要小心地處理組件之間的集成代碼就好。這些都有利於維護每個限界上下文的概念完整性。

但是，技術人員實在是太愛“分而治之”了，MSA（微服務架構）的出現就是證明。我們看到的很多微服務真的很“微”，幾乎是一個DDD的聚合就對應著一個可以獨立部署的微服務。這樣的服務單靠“自己”是做不了太多事情的，為了對外部客戶提供有意義的服務，它往往還需要調用其他的微服務。MSA技術基礎設施的發展也為服務之間的調用提供了更多的便利，跨越微服務的邊界變成了常態，區分“同一個上下文內的服務調用”與“上下文之間的防腐層”需要開發人員保持頭腦的清醒，限界上下文的概念邊界與微服務的物理邊界往往很難保持對齊，這些必然會增加維護每個限界上下文的概念完整性的難度。

既然維護一個個獨立的限界上下文的概念完整性變得越來越難，那麼，乾脆我們將它們重新融合在一起吧？將它們融合為一個大大的限界上下文，在這個上下文內維護概念完整性，這就是所謂的企業級業務架構。鑑於“企業級”這個概念不夠時髦，於是給它取了一個新名字：“中臺”。

注意，這裡說的“融合”是指概念上的融合。當然，也許會在中臺中使用一些“新”的組件化技術，但是為了更好地服務客戶，中臺必然需要打破軟件組件之間的物理邊界，在一個更大的上下文內維護概念的完整性。想要獲得企業級的大和諧，難度不會因為採用某種拆分物理組件的“新技術”而降低，因為領域的範圍就在那裡。並不是說建設中臺的理念毫無意義，最少對於使用中臺的客戶來說，不需要理解不同上下文之間的概念的差異，免去了對不同上下文的概念進行轉換的麻煩，這不是挺美好的事兒嗎？

軟件工程是妥協的藝術，是中庸之道。要不要中臺、要多大的中臺，不管企業大小，都應該結合自身的業務目標以及擁有的資源，在“維護更大範圍的概念完整性”與“維護更多的防腐層代碼”之間做出平衡。

## 15.4 實例化需求與行為驅動測試

開發軟件要做的第一件事情，就是定義產品，也就是搞清楚“客戶想要的是什麼東西”。這個時候，實例化需求（Specification by Example）是一個你應該擁有的強大工具。

實話化需求的理念因2012年Jolt大獎圖書 Specification by Example:How Successful Teams Deliver the Right Software<sup>[1]</sup>的推介而在軟件業界廣為人知。

[1] Gojko Adzic. Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications, June 2011. 見 <https://www.amazon.com/Specification-Example-Successful-Deliver-Software/dp/1617290084>。

## 15.4.1 什麼是實例化需求

什麼是實例化需求？這裡的解釋引用自InfoQ.cn上的文章“《實例化需求》採訪與書評”（中文翻譯版）[\[1\]](#)：

實例化需求是一組軟件開發過程模式，可以確保有效地交付正確的產品，協助軟件產品的變更。這裡所說的正確的產品，指的是該軟件能滿足用戶提出的商業需求，或能達成預定的商業目標；同時它要具備一定的靈活性，未來能以相對平穩的投入接受後續改進。

《實例化需求》的作者Gojko Adzic在書中詳細描述了該過程的多個步驟：

- 1 ) 從目標中獲取範圍。
- 2 ) 從協作中制訂需求說明。
- 3 ) 用實例進行描述。
- 4 ) 精煉需求說明。
- 5 ) 自動化驗證，無須改變需求說明。需求說明以自然語言表達，並且在此前提下可以用自動化的測試來驗證它。

6 ) 頻繁驗證 。

7 ) 演進出一個活文檔系統 。

通過這些步驟，我們可以獲得準確的需求並將其轉化成一份活文檔。活文檔是實例化需求的最終產物。所謂的活文檔，指的是這些文檔實時追蹤著系統的功能，準確地描述了系統當前的行為。這些文檔是驗收測試（Acceptance Test）代碼的一部分——你可以理解為“文檔即代碼”，通過頻繁執行這些測試，我們可以確信這些文檔是“活”的。

[1] 見 <https://www.infoq.cn/article/specification-by-example-book/> 。

## 15.4.2 BDD工具

實例化需求有時又被稱為BDD ( Behavior-Driven Development , 行為驅動開發 ) , 雖然Gojko Adzic本人認為BDD是一個還沒有精確定義的方法論。如果我們想了解有哪些工具可以幫助我們實踐實例化需求，可以用BDD作為關鍵字進行搜索。

### 1.Cucumber

Cucumber<sup>[1]</sup>是一個老牌的BDD軟件工具。Cucumber會讀取以純文本編寫的可執行的需求說明，並驗證軟件的行為是否恰如需求說明所言。

可以被Cucumber驗證的需求說明一般使用一種被稱為Gherkin的DSL來編寫。以下是一個使用Gherkin編寫的計算器的需求示例：

---

```
Feature: 計算器
    作為一個數學不好的人,
    為了避免一些愚蠢的錯誤,
    我想要一個計算器軟件可以告訴我兩個數字相加的和。
```

```
Scenario: 相加兩個數字
    Given 我在計算器中輸入 50
    And 我還在計算器中輸入 70
    When 我按下"相加"按鈕
    Then 屏幕上的結果是 120
```

---

以上示例中英文關鍵字的含義：

- **Feature**：功能、特性。
- **Scenario**：場景。一個功能可以分多個場景進行闡述，比如可能有成功的場景，還有失敗的場景。
- **Given**：假設。測試的前置條件。
- **And**：而且。
- **When**：當。可以用於描述用戶的輸入動作或其他觸發程序執行的事件。
- **Then**：然後。一般在這裡描述系統的輸出結果。

只要理解了這幾個關鍵字，我們就可以看到，以Gherkin編寫的需求說明非常接近自然語言。其實也可以使用中文關鍵字，但是一般沒有必要這麼做。

## 2.SpecFlow

我們在開發工作中，還使用過SpecFlow<sup>[2]</sup>這個BDD工具。SpecFlow是Cucumber的.NET移植版。使用SpecFlow可以在.NET項目中定義、管理、自動化執行人類可讀的驗收測試。它同樣支持主要的Cucumber/Gherkin語法。

[1] BDD Testing & Collaboration Tools for Teams | Cucumber, <https://cucumber.io> .

[2] SpecFlow - Binding Business Requirements to .NET Code, <http://specflow.org> .

### 15.4.3 BDD工具應與DDD相得益彰

實例化需求與BDD工具回答了“如何釐清客戶想要的是什麼”這個問題，但是它沒有回答應該如何進行系統分析、領域建模的問題。

嚴格來說，“正確的軟件”只需要在功能（“表面”）上滿足客戶所需就可以了，並不意味著軟件的代碼一定要和“反映對領域深度認知的模型”緊密關聯，也就不意味著高質量的實現。

我們不應該滿足於編寫一個只是可以讓機器（BDD工具）執行驗證的需求說明，更應該關注如何寫出易於人類閱讀、便於團隊維護、有助於加深開發人員對領域（“業務”）的認知、指導開發人員編碼（最少給他們提供一些“好名字”）的需求說明。DDD的統一語言（Ubiquitous Language）對此尤其有益。

整個團隊應該使用基於領域模型的統一語言進行溝通。產品人員、開發人員、測試人員、業務人員應該使用統一語言合作編寫需求說明。一個精心編寫的Gherkin文檔應該是可以拿給客戶（業務部門）去簽字確認的。

我們應該直截了當地在需求說明中描述領域中的業務邏輯，儘量避免描述用戶與應用界面（UI）的交互細節。因為後者是不穩定的、易變的。上面“計算

器”需求說明的例子有些地方可能會誤導你，要知道那只是一個以展示Gherkin語法為主要目的的簡單演示而已。

因為上面的“計算器”的例子非常簡單，所以你可能會懷疑使用Gherkin編寫實例化需求說明的“實用性”。事實上，現實世界中存在很多複雜而精彩的實例化需求說明，不在這裡展示只是因為商業上的以及本書篇幅的限制。當然，我們不否認編寫實例化需求說明是需要一定投入成本的，需要團隊的協作，特別是需要技術人員的參與。

#### 15.4.4 不要在驗收測試中使用固件數據

敏捷開發方法的一個非常重要的實踐是TDD（測試驅動開發）。很多開發人員理解的TDD其實是UTDD（單元測試驅動開發），事實上，TDD更應該是ATDD（驗收測試驅動開發）。對實例化需求來說，只有單元測試是遠遠不夠的，實例化需求需要的是驗收測試的自動化。不少人認為實例化需求（SbE）、ATDD、BDD三者的內涵基本相同。

現實情況是願意寫單元測試的人已經很多，但驗收測試就不太招人待見。先說說大家比較認可的“什麼樣的測試是單元測試”：

- 單元測試不使用數據庫。
- 單元測試不使用網絡通信。
- 單元測試不讀寫文件系統。
- 單元測試可以和其他任何單元測試同時運行。
- 不需要為了運行單元測試而對測試的運行環境做任何特殊設置。

雖然實例化需求需要的是驗收測試，但是我們仍然希望關鍵的驗收測試儘可能像單元測試一樣敏捷。

也就是說，我們希望驗收測試儘可能自行設置一個運行它自己所需要的獨立的環境。

驗收測試一般不應該使用內存數據庫，而應該使用與生產環境儘可能相似的數據庫以及其他外部環境，因為我們需要向負責驗收的產品人員或客戶展示測試結果。

如果我是一個開發團隊的主管，在需要對軟件功能進行驗收測試的時候，最不願意聽到的話（之一）就是：“等我從備份還原一下數據庫。”這個數據庫備份屬於固件數據。《實例化需求》認為使用固件數據的做法屬於反模式。這裡的固件即測試固件（**Test Fixture**），指為了運行一個測試你需要準備好的所有東西——這些東西是運行測試的前置條件，也就是在BDD中“**Given**”所指的那部分東西。所謂的使用固件數據，可以理解為：在運行測試前，我們需要把數據庫設置成一個固定的初始狀態。

測試的運行如果依賴於這個固定的數據庫狀態，那麼，可能測試場景運行的先後順序就會決定測試通過或通不過，因為在測試運行過程中外部數據庫的狀態會發生變化。如果測試有時候通過、有時候通不過，不是因為代碼的修改，而是受到外部環境變化的影響，這就是所謂“閃爍的場景”。

使用數據庫備份來進行測試的另外一個問題是，維護用於測試的數據庫備份是個麻煩事。在開發過程

中你可能需要經常改變數據庫的Schema，這時可能需要更新那些過期的數據庫備份。

所以，在開發驗收測試時，我們應該儘可能地在代碼中先自己構造一套僅供當前測試使用的數據。我們想要的是：在運行驗收測試之前無須還原數據庫備份，運行之後也無須執行“清理數據庫”的動作，測試可以一遍又一遍地重複運行，每次都是可以通過的。

高效開發團隊應該做到：人與測試的唯一交互就是在測試的開發過程中，其他一切應該由測試自動化工具或其他測試代碼來管理。更進一步說，驗收測試應該是CI/CD（持續集成/持續交付）與DevOps實踐的一部分，人與應用的唯一交互就是在開發過程中，其他一切應該由基礎架構或其他應用程序來管理（只有在應用發生故障時例外，故障恢復後應修復軟件以便下次不需要人工干預）。

## 15.4.5 製造“製造數據”的工具

很多開發團隊熱衷於以還原數據庫的方式準備測試的前置條件，是因為在之前的開發過程中缺少對“製造數據”的測試代碼的積累。筆者的建議是早日補上這些代碼。

因為在數據庫的數據之間可能存在複雜的“勾稽”關係，所以編寫SQL語句來製造數據有可能是件十分麻煩的事情，通過編寫近似端到端測試的方式來製造數據可能是個更好的選擇。我們可以考慮在儘可能靠近前端的地方，向後端的服務組件發送命令，利用後端已經存在的業務邏輯製造出儘可能接近真實的數據。

饒是如此，編寫這些發送命令的代碼仍然十分枯燥和無聊，我們應該考慮使用工具來輔助完成這樣的工作。沒有工具？大不了我們製造工具嘛，最少製造工具的過程還比較有趣。

那個“為了測試需要還原的數據庫”往往來自於生產環境的備份，裡面可能包含了大量真實、生動的可用於測試的數據，難怪有人對它們“愛不釋手”。也許我們可以考慮將這些數據提取出來作為測試代碼的一部分。下面我們就看看這樣做的一個例子。

這個例子可行的前提是：在應用已有的代碼中，對某個聚合而言，很大概率存在一個創建這個聚合——聚合根以及它關聯的聚合內部實體的實例——的接口，這個接口接受的參數可能是一個和聚合的狀態對象非常相似的命令對象。沒有這樣的接口？也許可以考慮添加一個這樣的接口.....



提

示這裡頻繁地出現了“聚合”這個術語，如果你對它還很陌生，需要先去了解DDD的“聚合”概念。使用聚合作為數據訪問的單元，我們幾乎總是會自然地設計出獲取聚合狀態的接口以及創建聚合的接口。

這個例子來自於我們開發過的一個真實的WMS應用。在這個應用裡，存在一個叫作**InOut**（入庫/出庫單）的聚合，聚合根（實體）**InOut**的**ID**叫作**DocumentNumber**。創建入庫/出庫單的命令對象**CreateInOut**和狀態對象**InOutState**“長得很像”，也就是說兩者的大部分屬性是一樣的。

首先，我們可以編寫這樣一個**Java**應用程序類，連接測試數據庫，將一個在測試數據庫中已經存在的入庫/出庫單的狀態以**JSON**格式打印到控制檯（這裡假設該入庫/出庫單的單號為**1578130488381**）：

---

```
package org.dddml.wms;

import org.dddml.wms.tool.jackson.JacksonEntityDataTool;
import
```

```
org.springframework.context.support.ClassPathXmlApplicationC
ontext;

public class EntityStatePrintApp {
    public static void main(String[] args) {
        org.springframework.context.ApplicationContext
springFrameworkApplicationContext
            = new ClassPathXmlApplicationContext(
                "config/SpringConfigs.xml",
                "config/TestDataSourceConfig.xml");
        String entityName = "InOut"; // = args[0]
        String entityId = "1578130488381"; // = args[1]
        String json =
JacksonEntityDataTool.getStateJsonForCreationCommand(
entityName, entityId);
        System.out.println(json);
    }
}
```

---

打印出來的**JSON**內容見下文。此內容不僅包含聚合根 (**InOut**) 的狀態信息，還包括與聚合根關聯的入庫/出庫單行項的狀態信息。我們可以將打印出來的**JSON**保存為文本文件。



**提**  
示事實上，通過讀取參數 (**args**)，應用程序**EntityStatePrintApp**可以打印出系統中各個聚合根（實體）的狀態信息，而不是僅限於**InOut**實體。

以上代碼中使用到的工具類  
**JacksonEntityDataTool**大致如下（Java代碼）：

---

```
package org.dddml.wms.tool.jackson;

import com.fasterxml.jackson.core.JsonProcessingException;
```

```

import com.fasterxml.jackson.databind.ObjectMapper;
import
org.dddml.wms.domain.jackson.StateToCreationCommandMixIns;
import org.dddml.wms.domain.meta.M;
import org.dddml.wms.tool.ApplicationServiceReflectUtils;
import java.io.IOException;
import java.lang.reflect.*;
import java.util.*;

public class JacksonEntityDataTool {
    private JacksonEntityDataTool() {
    }

    public static String
getStateJsonForCreationCommand(String entityName, String id)
{
    Object idObj = getIdObject(entityName, id);
    try {
        Object entityState =
ApplicationServiceReflectUtils.
invokeApplicationServiceGetMethod(entityName, idObj);
        ObjectMapper objectMapper = new ObjectMapper();

StateToCreationCommandMixIns.setUpObjectMapper(objectMapper)
;
        String aggregateName = M.BoundedContextMetadata.
TYPE_NAME_TO_AGGREGATE_NAME_MAP.get(entityName);
        Class stateClass =
getEntityStateInterfaceType(aggregateName, entityName);
        String json =
objectMapper.writerFor(stateClass).writeValueAsString(entity
State);
        return json;
    } catch (NoSuchMethodException |
InvocationTargetException | IllegalAccessException |
ClassNotFoundException |
JsonProcessingException e) {
        throw new RuntimeException(e);
    }
}

private static Object getIdObject(String entityName,
String id) {
    Object idObj;
    try {

```

```

        Class metadataClass =
Class.forName(M.class.getName() + "$" + entityName +
"Metadata");
        Field idClassField =
metadataClass.getField("ID_CLASS");
        Class idClass = (Class)idClassField.get(null);
        if (idClass.equals(String.class)) {
            idObj = id;
        } else {
            ObjectMapper objectMapper = new
ObjectMapper();
            idObj = objectMapper.readValue(id, idClass);
        }
    } catch (ClassNotFoundException |
NoSuchFieldException | IllegalAccessException | IOException
e) {
        throw new RuntimeException(e);
    }
    return idObj;
}

private static Class getEntityStateInterfaceType(String
aggregateName, String entityName) throws
ClassNotFoundException {
    String paramTypeName =
String.format("%1$s.domain.%2$s.%3$sState",
                getBoundedContextPackageName(),
aggregateName.toLowerCase(), entityName);
    return Class.forName(paramTypeName);
}

private static String getBoundedContextPackageName() {
    // 省略代碼
}
}

```

---

在方法`getStateJsonForCreationCommand`中，我們使用`ApplicationServiceReflectUtils`工具類——這個類使用了`Java`的反射機制——來獲取某個聚合根（實

體 ) 的狀態，然後使用Jackson JSON序列化庫的 ObjectMapper將實體的狀態對象序列化為JSON文本。

因為後面打算使用序列化的結果來創建實體的命令對象，所以在這個JSON字符串中不應該包括在創建入庫/出庫單的命令 ( CreateInOut ) 中用不上的屬性。為了做到這一點，上面的代碼使用到一個 StateToCreationCommandMixIns類，調用它的 setUpObjectMapper方法對ObjectMapper進行設置，可以使狀態對象的序列化結果如我們所願：

---

```
package org.dddml.wms.domain.jackson;

import com.fasterxml.jackson.annotation.*;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.annotation.JsonSerialize;
import org.dddml.wms.domain.*;
// 省略部分代碼

public class StateToCreationCommandMixIns {
    private StateToCreationCommandMixIns() {
    }

    public static ObjectMapper
setUpObjectMapper(ObjectMapper objectMapper) {

    objectMapper.setSerializationInclusion(JsonInclude.Include.N
ON_NULL);
        objectMapper.addMixIn(InOutState.class,
InOutStateToCreationCommandMixIn.class);
        objectMapper.addMixIn(InOutLineState.class,
InOutLineStateToCreationCommandMixIn.class);
        // 省略更多 addMixIn 的代碼
        return objectMapper;
    }
}
```

```

        public static class InOutStateToCreationCommandMixIn {
            @JsonIgnore public Long getVersion() { return null; }
            @JsonIgnore public String getCreatedBy() { return null; }
            @JsonIgnore public Date getCreatedAt() { return null; }
            @JsonIgnore public String getUpdatedBy() { return null; }
            @JsonIgnore public Date getUpdatedAt() { return null; }
            @JsonIgnore public Boolean getDeleted() { return null; }

            // 注意：忽略“單據狀態 ID”
            @JsonIgnore public String getDocumentStatusId() {
                return null; }

            @JsonSerialize(contentAs = InOutLineState.class) // 只序列化接口的屬性
            EntityStateCollection<String, InOutLineState>
            getInOutLines() { return null; }
        }

        public static class InOutLineStateToCreationCommandMixIn
        {
            @JsonIgnore public Long getVersion() { return null; }
            @JsonIgnore public String getCreatedBy() { return null; }
            @JsonIgnore public Date getCreatedAt() { return null; }
            @JsonIgnore public String getUpdatedBy() { return null; }
            @JsonIgnore public Date getUpdatedAt() { return null; }
            @JsonIgnore public Boolean getDeleted() { return null; }

            @JsonIgnore public String getInOutDocumentNumber() {
                return null; }

            @JsonSerialize(contentAs =
            InOutLineImageState.class) // 只序列化接口的屬性
            EntityStateCollection<String, InOutLineImageState>
        }
    }
}

```

```
getInOutLineImages() { return null; }
}
}
```

---

從以上代碼可以看出，在將入庫/出庫單的狀態（`InOutState`）序列化為JSON時，“單據狀態ID”（`documentStatusId`屬性）會被忽略（`@JsonIgnore`）。這是因為當使用一個命令（`CreateInOut`）去創建入庫/出庫單的時候，我們並不能指定單據的狀態。

需要說明的是，這個 `StateToCreationCommandMixIns` 類的代碼是由工具自動生成的。如果你像我們一樣使用DSL來描述領域模型——包括領域中靜態的數據結構以及動態的行為（比如創建實體的方法），那麼可能很容易通過DSL文檔知道狀態對象中的哪些屬性在實體的創建命令中並不存在（同名的屬性）。當然，其實也可以通過反射來生成類似的代碼。

在上面的`getStateJsonForCreationCommand`方法中，還使用了一個由工具生成的類 `org.dddml.wms.domain.meta.M`，裡面保存著限界上下文的元數據以及聚合/實體的元數據（Java代碼）：

---

```
package org.dddml.wms.domain.meta;

import java.util.*;
import org.dddml.wms.specialization.*;
```

```
public class M {
    public static class BoundedContextMetadata {
        // 類型 (主要是實體) 名稱到聚合名稱的映射表
        public static final Map<String, String>
TYPE_NAME_TO_AGGREGATE_NAME_MAP;
        static {
            // 省略代碼
        }
        // 省略代碼
    }

    public static class InOutMetadata {
        private InOutMetadata() {
        }
        // 入庫/出庫單的 ID 的類型
        public static final Class ID_CLASS = String.class;
        // 省略代碼
    }
}
```

---

在

**M.BoundedContextMetadata.TYPE\_NAME\_TO\_AGGREGATE\_NAME\_MAP**這個Map中，保存著從實體（類型）的名稱到它所屬的聚合的名稱的映射關係。而**M.InOutMetadata.ID\_CLASS**這個字段則指明瞭入庫/出庫單實體的ID的類型信息。

在方法**getStateJsonForCreationCommand**中使用到的應用服務反射工具類（**Application-ServiceReflectUtils**）的代碼大致如下：

---

```
package org.dddml.wms.tool;

import org.dddml.wms.domain.meta.M.BoundedContextMetadata;
import org.dddml.wms.specialization.*;
import java.lang.reflect.*;
```

```

import java.util.*;

public class ApplicationServiceReflectUtils {
    private ApplicationServiceReflectUtils() {
    }

    public static Object
invokeApplicationServiceGetMethod(String entityName, Object
id) throws NoSuchMethodException, InvocationTargetException,
IllegalAccessException, ClassNotFoundException {
        String aggregateName = BoundedContextMetadata.
TYPE_NAME_TO_AGGREGATE_NAME_MAP.get(entityName);
        Object appSvr =
getApplicationService(aggregateName);
        Class appSrvClass = appSvr.getClass();
        Method m = null;
        m = appSrvClass.getMethod("get", id.getClass());
        return m.invoke(appSvr, id);
    }

    public static void
invokeApplicationServiceCreateMethod(String entityName,
Object e) throws NoSuchMethodException,
InvocationTargetException, IllegalAccessException,
ClassNotFoundException {
        // 省略代碼
    }

    private static Object getApplicationService(String
aggregateName) {
        // 省略代碼
    }
}

```

---

其中，方法**invokeApplicationServiceGetMethod**用於獲取某個聚合根（實體）的狀態信息。而調用方法**invokeApplicationServiceCreateMethod**傳入聚合根（實體）的名稱以及一個創建命令對象（比如

`CreateInOut`），則可以創建出一個聚合的實例（比如一個包含多個行項的入庫/出庫單的實例）。

我們將應用程序類`EntityStatePrintApp`打印出來的JSON保存為文件（文件名為`testIn-OutData.json`），作為創建入庫/出庫單的命令對象的模板。為了防止替換過程中可能發生意外的錯誤，還可以把JSON文本中的“單號”**1578130488381**替換為 `${1578130488381}`，這是一個需要被新單號替換的佔位符，得到的JSON文件的內容如下：

---

```
{  
  "documentNumber": " ${1578130488381}",  
  "warehouseId": "W1",  
  "active": true,  
  "documentTypeId": "Out",  
  "inOutLines": [  
    {  
      "lineNumber": "1578155769040",  
      "productId": "1573353943865",  
      "locatorId": "W1-15635-6845-1",  
      "quantityUomId": "kg",  
      "attributeSetInstanceId": "b32d5c40-130e-4d52-  
b8f3-3d03b7432418",  
      "movementQuantity": -8285  
    },  
    {  
      "lineNumber": "1578156102657",  
      "productId": "1573353943865",  
      "locatorId": "W1-15635-6845-1",  
      "quantityUomId": "kg",  
      "attributeSetInstanceId": "93bd7cbe-72b9-4735-  
9e9a-ade9c5654e3e",  
      "movementQuantity": -8582  
    }  
  ]  
}
```

---

這樣的**JSON**代碼具備不錯的可讀性——最少比維護**SQL**容易。

以這個**JSON**文件為模板，在必要的時候，可以使用如下代碼創建出一個全新的入庫/出庫單：

---

```
// 省略代碼
public class JacksonTests {
    @Test
    public void doSomeTest() {
        // 省略代碼
        String docNumberHolder = "${1578130488381}";
        try {
            // 隨機生成一個新的入庫/出庫單號：
            String docNumber = "I" +
                (UUID.randomUUID().hashCode() + System.currentTimeMillis());
            InputStream inputStream =
                JacksonTests.class.getResourceAsStream("/data/testInOutData.
                json");
            String json2 =
                readText(inputStream).replace(docNumberHolder, docNumber);
            InOutCommand.CreateInOut createInOut =
                toInOutCreationCommand(json2);
            createInOut.setCommandId(docNumber);

            ApplicationServiceReflectUtils.invokeApplicationServiceCreat
            eMethod("InOut", createInOut);
            // 現在數據庫中存在一個新的入庫/出庫單了，接下來的測試代
            碼中可以使用它
        } catch (IOException | NoSuchMethodException
            | InvocationTargetException |
            IllegalAccessException | ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }

    public static InOutCommand.CreateInOut
    toInOutCreationCommand(String json) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper();
```

```
        CreateOrMergePatchInOutDto.CreateInOutDto
createInOutDto =
        objectMapper.readValue(json,
CreateOrMergePatchInOutDto.CreateInOutDto.class);
        return createInOutDto;
    }

    private static String readText(InputStream inputStream)
throws IOException {
    // 省略代碼
}
}
```

---

以上展示出來的代碼的大致邏輯是：

- 生成一個隨機的新的入庫/出庫單號 ( DocumentNumber ) 。
- 讀入文件 **testInOutData.json** 中的 **JSON** 文本內容，將其中的  **\${1578130488381}** 替換為新的入庫/出庫單號。
- 將 **JSON** 文本反序列化為一個 **CreateInOut** 命令對象。
- 向 **InOutApplicationService** 應用服務發送這個命令對象，這將產生一個新的入庫/出庫單。

在接下來的測試代碼中就可以使用這個全新的入庫/出庫單，避免了對固件數據的依賴。

## 15.5 要領域模型驅動，不要UI驅動

很多軟件的開發過程是這樣的：產品經理分析需求，給出產品原型以及產品需求說明文檔，然後召集開發人員和測試人員對需求進行評審，如果大家沒有異議，開發就進入編碼階段。開發人員開始看著產品原型去做數據建模、決定數據庫的Schema，編寫業務邏輯層、UI層代碼……

在這個過程中幾乎沒有人去構建領域模型，大家直接從用戶體驗出發，“一步到位”開發出工作的軟件。筆者把這樣的開發模式叫作UI（用戶界面）驅動開發。

其實很多大廠中的軟件開發團隊在開發方法論上都談不上先進：產品人員先做產品設計、做原型，然後團隊反覆評審，等確認下來，一個月過去了；然後前端開發人員開始介入，按照產品的最終的用戶體驗（UE）要求來做App UI部分的開發，半個月過去了；然後再次評審，評審通過才算“正式”進入開發階段。整個過程是瀑布式的，這其實也還沒有太大的問題，但是其中並沒有人有意識地構建模型、維護概念完整性，開發過程基本上是基於UI驅動的，這才是大問題。

不要以為大廠的代碼質量就一定高，如果仔細看看諸多大廠的那些“Open API”，慘不忍睹的細節比比皆是。當然對於大廠這可能不成問題，它們有足夠的資源來“維護”那些代碼，但是對資源不足的小團隊來說這可能是大問題。

UI驅動開發可以在“摸著石頭過河”的過程中修補出可以工作的軟件，但是軟件的代碼和“一個反映對領域深度認知的模型”之間往往缺少關聯，對於稍微複雜的應用來說，這就意味著難以維護的低質量的代碼。

比如，在前面“數據建模示例：訂單的裝運與支付”一節中，我們就看到某個購物App的“我的訂單”UI可能會誤導開發團隊的建模思路。

下面是筆者在現實中見過的可能是照著UI做出來的幼稚模型：

- 產品和促銷（活動）不分。
- “發佈活動”就是錄入新的產品和價格，造成數據庫內大量重複的產品記錄。
- 因為UI上有客戶（Customer）、門店（Store）、供應商（Supplier）、製造商（Manufacturer）等概念，為它們都定義了對應的獨立實體。

·更有甚者，認為它們都是不同類型的“用戶”，因為需要“讓TA們使用我們的系統”，於是創造出“門店用戶”“供應商用戶”等概念。

·用戶賬號和登錄賬號概念不分。App每增加一種登錄方式，比如微信登錄、微博登錄、手機號登錄、郵箱登錄、暱稱登錄，就在用戶表中增加一列。甚至，在這個數據模型的基礎上又實現了“一個手機號登入多個用戶賬號”的功能，於是App的用戶在輸入手機號登錄時，要先想清楚“我是誰”，然後在彈出的“用戶列表”中選擇其中的一個，然後才能登入系統。

·各種產品屬性滿天飛。一開始，模型中只存在產品（Product）這個概念。後來，為了支持前端App的在購買某個產品時選擇不同的規格而引入SKU的概念。比如說，某款襯衫有“尺寸”和“顏色”兩種規格可選，這個規格也被稱產品屬性。所謂的SKU就是“同一個產品不同規格的實例的組合”。比如說，某款襯衫產品有一個SKU是“尺寸：XL；顏色：白”。這個SKU實體有一個Local ID叫作SkuId，也就是說SKU表的主鍵是ProductId與SkuId兩列的組合。於是原來那些通過ProductId屬性引用產品的實體（OrderItem、ShipmentItem、Agreement、InventoryItem、ReturnItem等）全部增加一個SkuId屬性。然後，因為要“以智能製造滿足客戶的個性化需求”，產品人員在用戶下單購買產品的流程中加入定製屬性頁面。於是開發人員再引入一個CustomizedAttributeSetInstance（定製屬性集實例）實體，用於記錄用戶選擇了什麼

定製屬性。原來引用產品的大多數實體（OrderItem、ShipmentItem、InventoryItem、ReturnItem等），都再增加一個CustomizedAttribute-SetInstanceId屬性。

如果你是一個有良好品味的開發人員，看到這樣的模型，你心裡不苦嗎？

## 15.6 不要用“我”的視角設計核心模型

在筆者的職業生涯前期，經常從“我”的視角去設計軟件的數據模型。閱讀那些數據模型，你看到的是“第一人稱”敘述：這是我的銷售訂單，這是我的採購訂單；這是我的銷售發票，這是我的採購發票；這是我的銷售發貨單（Sales Shipment），這是我的採購發貨單（Purchase Shipment）……

這裡的“我”是業務流程的參與者，很多時候也是軟件提供的功能的使用者（“用戶”），因此從“我”的視角看過去，訂單才會分成“銷售訂單”與“採購訂單”。

UI為“我”提供觀察和使用系統的視角，從“我”的視角去設計，某種程度也可以說是UI驅動設計。比如說，我在“董小姐的店”這個購物App上買了一個MacBook筆記本電腦，我在App的UI上看到我的“採購訂單”。對於董小姐來說，她在管理後臺（Admin UI）上看到的是她的“銷售訂單”——你是不是已經從中看出點問題來了？

從“我”的角度設計的模型有什麼問題？比如，在為“我”的公司開發的某個應用的模型中，銷售訂單（Sales Order）與採購訂單（Purchase Order）一開始被設計成兩個獨立的實體/表。其原因可能是：在開發這個應用的時候，公司的採購和銷售流程是分離

的，產品被批量地採購進來，放到倉庫裡面，然後一件一件地往外賣。開始的時候，應用完全滿足了這些業務流程運作的需要，一切都好，直到某一天，“我”的公司增加了Dropship（直運）銷售業務。也就是說，有些商品公司不需要屯貨（庫存），公司就等著客戶下單，在客戶下單後，把訂單和裝運細節告知商品的供應商，供貨商直接向客戶發貨。那麼，這時我們的數據模型如何“改造”呢？我們是增加一個新類型的訂單實體/表（比如叫作Dropship Order），還是不修改數據模型，只是修改業務邏輯層的代碼？當用戶下單的時候，我們是否在創建一個銷售訂單的同時再創建一個採購訂單？

無論哪種做法，都不對勁。這樣的不對勁一再發生，我想是我的腦袋有問題，但是，我看別人做的很多系統不也是這麼做的嗎？

直到某一天，我看了《數據模型資源手冊（卷1）》<sup>[1]</sup>，閱讀了OFBiz<sup>[2]</sup>的數模模型，於是一下子豁然開朗了。原來在建模的時候，應該放棄第一人稱敘述，使用第三人稱敘述。

也就是說，在設計業務軟件的核心數據模型時，不要總是把自己代入到某個業務參與者的視角，要想象自己就是和參與業務流程的各方都毫無關係的旁觀者，只是想要對“他們”做出客觀的記錄。

讓我們看看OFBiz是怎麼做的。

- [1] 希爾瓦斯頓. 數據模型資源手冊 (卷1). 機械工業  
出 版 社 , 2004-6. 見  
<https://book.douban.com/subject/1230513/>。
- [2] 見<https://ofbiz.apache.org/>。

## 15.6.1 讓User消失

如果仔細觀察一下OFBiz的核心數據模型，就會發現其中完全沒有User（用戶）、Role（角色）這些概念。

想想也對。用戶是使用軟件的“人”，有時，用戶甚至不是“人”，而是另外一個程序。業務軟件的核心數據模型應該客觀地記錄各方（Party）——甲方、乙方、丙方、丁方等——的業務數據。很多時候，用戶使用軟件，需要錄入的是TA（他/她/它）並不直接參與的業務流程的數據。除非你是真的在開發一個用戶管理系統，否則引入User的概念毫無必要。

筆者多次見識過引入“用戶”概念給建模工作造成的干擾。比如說，把“供應商”建模為一個“用戶”的子類型。因為那個系統的開發人員是這麼想的：“你看，現在用戶表裡面有郵箱、手機號、名稱等信息，供應商也需要有這些屬性不是嗎？而且，確實有部分供應商會使用我們的系統，那顯然供應商就是用戶嘛。”但是，實際上還有更多的供應商根本就不使用那個系統！

所以筆者認為：在業務軟件的核心數據模型，以及核心業務邏輯代碼中，應該徹底讓User消失。除非，你真的是在做一個用戶管理系統。

讓User消失後，讓Party帶著它的小夥伴們登場。

## 15.6.2 認識一下Party

Party是業務流程的參與者。這個參與者可能是自然人，可能是法人，也有可能是其他非正式的組織。

 提示  
示法人指具有民事權利能力和民事行為能力，依法獨立享有民事權利和承擔民事義務的組織。

筆者見過的Party的中文翻譯包括：當事人、團體、會員。推薦使用業務實體這個名稱。

在OFBiz中，Party這個概念是個很強大的抽象。它的類型（PartyType）可包括個人、組織、企事業單位、政府機構、公司的部門、臨時組織的團隊、家庭等。

 提示  
示開發人員需要注意區分“業務實體”的這個“實體”和他們熟悉的表示“擁有不變的ID的對象”的那個“實體”。

### 1. PartyIdentification與PartyIdentificationType

PartyIdentification表示業務實體的標識，而業務實體的標識有不同的類型（PartyIdentificationType）。

以Person（自然人，Party的子類）為例，在現實世界中，可以有多個標識（即PID，個人標識）指向同一個人。

比如，我的護照號、我的身份證號、我的駕照，都指向我。甚至，我的個人手機號也可以算是我的一個PID，我有可能更換我的個人手機號，所以，不要使用這些PID作為“我”的Party ID。

PartyIdentification實體的重要屬性（帶有“[PK]”後綴的對應著數據表構成主鍵的列）如下。

·partyId [PK]：業務實體ID。

·partyIdentificationTypeId [PK]：業務實體標識類型，比如“護照”。

·idValue：業務實體標識的ID值，比如護照的“護照號”、身份證的“身份證號”。

## 2. RoleType

一個業務實體可以扮演多個角色類型（RoleType）。

比如說，角色類型可能是：

·Customer（客戶）

- Manufacturer ( 製造商 )
- Employee ( 僱員 )
- Distributor ( 分銷商 )
- Family Member ( 家庭成員 )



注

意這裡的Role Type和我們做權限管理時經常碰到的那個Role (“角色”) 含義不同。那個Role和用戶的權限相關，可以理解成權限集。比如，我可以創建一個管理員的角色，這個角色具有出庫、入庫、裝運單導入等權限。然後我們可能讓角色與用戶賬號相關聯，系統以此來判斷用戶能做什麼/不能做什麼。

### 3. Party與RoleType的關聯

在OFBiz中，角色類型 ( Role Type ) 和業務實體 ( Party ) 之間是多對多的關係，這就需要使用一個 PartyRole 作為它們之間的關聯實體。

不過，在很多系統裡面，一個業務實體可能確實就是承擔一種角色類型。比如有的Party就只是供應商。怎麼把問題簡化下來？一個可以考慮的做法：在使用PartyRole實體關聯Party和RoleType的同時，給Party實體增加一個PrimaryRoleId屬性，表示它主要扮演的角色類型。如果你覺得真的需要這麼做，也請注意不要隨意濫用這種冗餘字段模式。儘可能把這

個PrimaryRoleTypeId當成一個Party的備註屬性來用就對了。



提示如果你瞭解CQRS模式，也許會想到，像這裡的“Party的PrimaryRoleTypeId”這樣的信息，可以考慮作為查詢模型來維護。

## 4. OrderRole

OFBiz為了增加擴展性，使用了OrderRole（訂單角色）實體來表示訂單和Party之間的關聯。

一個訂單，它的下單客戶（PLACING\_CUSTOMER）可能是張三，但可能需要發貨給李四（它的SHIP\_TO\_CUSTOMER是李四）。在訂單表中，並沒有PLACING\_CUSTOMER\_PARTY\_ID這一列，也沒有SHIP\_TO\_CUSTOMER\_PARTY\_ID這一列。這些信息是記錄在OrderRole表中的。

OrderRole實體的屬性如下。

- orderId [PK]：訂單ID（訂單號）。
- partyId [PK]：業務實體ID。
- roleTypeId [PK]：角色類型ID。

像OrderRole這樣的實體在OFBiz中還有很多，比如InvoiceRole（發票角色）、ShipmentRole（裝運單角色）等。

## 5.PartyRelationship與PartyRelationshipType

不管是個人還是組織，一個業務實體總是會跟其他業務實體發生關係，這樣的關係在OFBiz中使用PartyRelationship（業務實體關係）實體來表示。這個實體有如下屬性。

- partyIdFrom [PK]：從什麼業務實體。
- partyIdTo [PK]：到什麼業務實體。
- roleTypeIdFrom [PK]：從什麼角色類型。
- roleTypeIdTo [PK]：到什麼角色類型。
- fromDate [PK]：關係的生效時間。
- thruDate：關係的結束時間。
- partyRelationshipTypeId：業務實體關係類型（PartyRelationshipType）的ID。

其中，partyRelationshipTypeId可能為：  
EMPLOYMENT（僱傭關係）、

DISTRIBUTION\_CHANNEL ( 分銷渠道 ) 、  
PARTNERSHIP ( 合作伙伴 ) 等。

在第3章的表3-1中存在一條PartyRelationship的示例記錄。

## 15.7 我們想要的敏捷設計

對於複雜領域，一個反映對領域的深度認知的模型具有巨大的價值，找到它卻是一項艱鉅的任務。尋找它的過程可以自下而上——儘快開始編碼，產出工作的軟件，然後重構、改進代碼，在這個過程中摸索出深度認知的模型；也可以自上而下——在編碼開始之前進行充分的分析和設計，包括學習和汲取前人的經驗。

當然，我們可以結合使用這兩種方法，不管是使用哪種方法，我們希望過程都是敏捷的。但現實情況是，很多實踐敏捷的團隊已經過度依賴前者，而大大低估了進入編碼階段前軟件設計的價值。也許是時候來一場矯枉過正的“敏捷反動”了……

我們想要如下這樣的敏捷軟件設計方法：

它“打通”需求分析、領域建模與編碼實現。

它幫助我們穿過UI（用戶界面）、UE（用戶體驗）的“迷霧”，直達領域的本質。

它應該在我們進行需求分析、領域建模後，幫助我們以最快的速度產出工作的軟件，以寬慰我們以及客戶焦慮的心。

它應該幫助領域專家儘快確認我們所設計的東西是否是正確的軟件。

如果我們認為軟件的設計有問題，它應該幫助我們馬上重頭再來，重新分析、調整模型、再次產出工作的軟件、再次確認設計。

顯然，這一切有賴於它給我們提供低代碼（Low-Code）的軟件生產工具——它讓我們敢於在設計上投入重注。

當設計階段告一段落，所有艱難的決定都已做出時，它能為開發人員留下可以繼承的優質資產：一個反映對領域深度認知的模型以及基於此模型產生的代碼。開發人員可以在不破壞代碼與模型映射關係的前提下擴展代碼、完善軟件。

## 第16章 說說SaaS

SaaS是互聯網化的應用軟件。SaaS軟件廠商向客戶提供的是工具的價值。SaaS軟件的用戶使用該軟件的目的為了生產，所以他們又被叫作B端（Business）用戶，而使用App通過互聯網進行消費的用戶，叫作C端（Consumer）用戶。

這些年，每隔一段時間，SaaS就會被人拿出來熱炒一番，然後又沉寂下去。筆者認為大家普遍低估SaaS需要的技術難度——倒不是其中有多高深的算法，主要是工程難度。少數依賴商業模式驅動“成功”的SaaS給大家造成了“SaaS沒有技術含量”的片面印象。

雖然SaaS“不過是應用程序而已”，但是這個商業模式需要用到與之相匹配的底層技術基礎設施。而構建技術基礎設施的很多細節，如果不在其中摸爬滾打，恐怕是難以掌握的。

## 16.1 何為SaaS

以下是百度百科的解釋<sup>[1]</sup> ( 有刪節和少量修改 ) :

SaaS是Software-as-a-Service ( 軟件即服務 ) 的簡稱.....一種完全創新的軟件應用模式。.....通過Internet提供軟件的模式，廠商將應用軟件統一部署在自己的服務器上，客戶可以根據自己實際需求，通過互聯網向廠商定購所需的應用軟件服務，按定購的服務多少和時間長短向廠商支付費用，並通過互聯網獲得廠商提供的服務。用戶不用再購買軟件，而改用向提供商租用基於Web的軟件，來管理企業經營活動，且無須對軟件進行維護，服務提供商會全權管理和維護軟件.....SaaS是採用先進技術的最好途徑，它消除了企業購買、構建和維護基礎設施和應用程序的需要。

SaaS軟件能打動客戶（企業）的關鍵其實還是兩個字：“省錢”。用SaaS為什麼能省錢呢？

·租用。通過租用軟件，企業使用軟件的成本被攤薄，這大大降低了企業購買軟件的決策門檻。“好不好用試試看”“如果這家不行，這點錢大不了就當打水漂了”，這是很多企業主或者企業的IT負責人在決定購買SaaS軟件時的心態。

·低運維成本。企業不再需要建立自己的IT團隊，去構建和維護各種軟硬件基礎設施和應用程序。理論上，SaaS廠商規模化運維的成本會更低。

[1] 見<https://baike.baidu.com/item/SaaS>。

## 16.2 多租戶技術

仔細思考前文所述，其實可以發現：SaaS本質上只是一種商業模式。只是為了實現這樣的商業模式，我們還需要匹配相應的技術解決方案。

如果企業不想買服務器、不想養IT團隊，想把原來要自己去做的工作以更低的成本轉嫁給SaaS軟件供應商，那麼後者如果按照老辦法來做事情，怎麼賺錢？顯然，想要賺錢就必須採用新辦法——多租戶技術就是SaaS必須採用的一種新辦法。理論上來說，SaaS並不是一定需要使用多租戶技術來實現。但是，只有使用了多租戶技術，才能有效降低SaaS軟件的運維成本，使SaaS在商業上具備可行性。

多租戶技術（Multi-Tenancy Technology）或稱多重租賃技術，是一種軟件架構技術，它研究與實現：如何於多個租戶（租用軟件的客戶）的環境下共用相同的系統或程序組件，仍可確保各租戶間數據的隔離性。

如果用一句話總結，就是一套代碼服務所有客戶（租戶）。

只有在多個租戶之間共用、共享一套應用程序的核心代碼，才能有效降低運維應用程序的成本。當應

用需要升級時，只要重新部署一套代碼，所有租戶就可以同時享受到升級後的新版本。

購買**SaaS**軟件需要警惕那些自稱可以“交鑰匙”的軟件供應商。一些自稱做的是**SaaS**的供應商承諾給客戶提供軟件的全部源代碼，“源代碼隨便改”，他們把這個叫作“交鑰匙”。但筆者認為，這麼說的供應商，賣的幾乎就不可能是**SaaS**。

你能想象**Salesforce**會把整個平臺的全部源代碼交給你嗎？大部分**SaaS**一般只是開放**API**（應用程序接口），讓你可以通過**API**自己開發部分外圍功能。

**SaaS**不能給客戶源代碼（或者就算給了客戶源代碼，也不會讓客戶修改他們租用的線上系統的那些源代碼），不只是怕洩露商業、技術機密，或者是“敝帚自珍”的問題，而是由**SaaS**的業務模式決定的。**SaaS**只有堅持一套代碼服務所有客戶，才能把運維成本降下來。

想象一下，倘若給每個客戶源代碼，客戶可以自行修改源代碼、部署獨立的應用，那麼軟件供應商服務一百個客戶，就有一百套不同的源代碼——雖然其中可能只是略有差異。假設，現在發現了一個**Bug**，這個**Bug**在一百套源代碼中都存在，會影響所有的客戶，那麼需要幹什麼？需要修改一百套源代碼、重新部署一百套源代碼到服務器。這對開發人員、運維人員來說是一場噩夢。

所以SaaS不能真正“交鑰匙”（讓客戶完全自由地修改源代碼）。一旦這麼做，SaaS低成本運維的優勢就不再存在。

## 16.3 構建成功的SaaS有何難

為什麼中國成功的SaaS軟件廠商如此之少？為什麼中國沒有出現Salesforce？為什麼幾乎所有的傳統軟件廠商都看到“SaaS是未來”，但是能成功轉型的少之又少？

筆者認為這是因為構建SaaS的方法論並不成熟。對於一個小軟件廠商來說，獨立構建一個可以良好運作的SaaS目前還是有很難跨越的門檻的。

### 16.3.1 多租戶系統的構建成本

要開發一個支持多租戶技術的軟件系統，初期的投入遠遠高於“給這個客戶做一個系統”這樣的私有部署的（On-Premises）傳統軟件的開發投入，大家對此心知肚明。軟件廠商前期願意燒錢投入開發SaaS軟件，是為了未來服務更多客戶（租戶）的邊際成本的下降。但是大多數人仍然大大低估了開發SaaS系統所需的資源和成本，所以很多SaaS項目甚至都沒能撐到開發出一個可以運行的MVP（最小價值產品）就中途夭折了。

那麼，怎麼降低SaaS系統的初期開發成本？是否可以存在這樣一種理想的開發模式——應用軟件的開發人員能以“給一個客戶做一個系統”的思維模式進行SaaS系統的開發？

什麼面向多租戶、什麼數據隔離、什麼系統運維，這些因素能不能不要打擾應用開發人員的日常編碼工作？

這是一個SaaS技術基礎設施（有人稱之為技術底座）需要解決的問題。100%的理想狀態很難達到，但是我想，最少在80%的應用開發時間裡，多租戶技術的實現問題不應該困擾到應用開發人員吧——目前想要構建這樣的一個技術基礎設施還真的挺費錢的。

### 16.3.2 難以滿足的定製化需求

一般來說，SaaS軟件追求的是通用性、標準化，基於“天底下所有的公司都是差不多的”的理念設計產品。但是，客戶往往認為自己是獨特的。

大多數SaaS軟件只有在這種方式下才能給客戶省到錢：我有什麼功能你就用什麼功能。

客戶要的功能沒有怎麼辦？能不能讓SaaS廠商修改應用的核心代碼、增加新功能來滿足自己的需求？

如果SaaS廠商認為這個功能是通用的，看在錢的份上願意優先開發，那麼客戶需要付出的成本也不會低。因為SaaS為了解決一套代碼服務很多客戶的問題，所採用的技術棧可能對開發人員提出了更高的要求——需要更“貴”的開發人員。如果客戶想要的功能確實就是獨特的，SaaS廠商覺得根本不應該作為SaaS標準產品的一部分呢？這對很多SaaS廠商來說，做也不是，不做也不是，實在是太難了。

我們可以看到的現狀是，在整個企業管理應用軟件領域有太多類似的系統，這些系統也許只有少量業務邏輯上的差異。很多標準化的軟件往往因為沒有很好地解決這些少量的差異化的需求，而錯失很多高價值的客戶。而許多客戶（公司）也依舊會選擇立項開發定製化的系統。

SaaS廠商當然也很想在不影響客戶使用標準化產品的前提下，為這些高價值的企業客戶提供定製化的功能。那麼，要怎樣才能滿足這些客戶的定製化需求？

SaaS軟件實現定製化的方法之一，是先對領域內的企業的“各種做法”進行抽象——提高模型的抽象層次，然後允許不同的客戶（租戶）根據自身的需求對系統進行配置。這裡的配置（也可以稱為元數據）應該是“聲明式”的，即支持客戶說出“我想要什麼”。理想的情況是，客戶的差異化需求都應該體現在租戶的配置信息上。

我們可以看到這裡的難點是：

- 首先是對產品人員（包括需求分析師、系統分析員、業務架構師等）的分析、建模、產品設計的能力有要求。

- 能力再強的產品人員也不可能一次就設計出完美的模型，這就帶來第二個難題，即怎麼降低SaaS系統的修改成本。也就是說，SaaS的技術基礎設施、採用的技術棧，需要在業務需求發生變化，特別是領域模型需要調整的時候，支持開發人員快速、低成本地改進應用程序。

有人聲稱PaaS可以解決這些問題。PaaS是Platform-as-a-Service（平臺即服務）的簡稱。PaaS是

技術工具平臺，主要是面向軟件的開發者。SaaS是應用軟件，面向“普通人”。

SaaS廠商可以基於PaaS製造出SaaS。或者說SaaS可以內置PaaS，用於實現SaaS的擴展和定製。有時候，“SaaS提供的PaaS”，和上面說的“SaaS通過提供配置功能滿足客戶（租戶）的差異化需求”，兩者之間並沒有明確的界限，所以有人還提出所謂“可配置化的PaaS”的說法。

一般來說，SaaS的配置信息（元數據）是聲明式的，就是描述“我想要什麼”，而不是系統“怎麼做”。它能提供多大的靈活性，很大程度上取決於產品人員當初是否“想過客戶可能要什麼”。PaaS可能提供更強大的功能，比如圖靈完備的開發語言，能讓開發者創造產品人員“沒有想過”的新東西。當然，如果可能，大家都希望PaaS能具有更多聲明式開發的能力，畢竟開發人員也不想多敲代碼。

開發人員是最挑剔的，SaaS軟件廠商如果想要通過對外開放PaaS來為自己的SaaS產品提供定製化、可擴展的能力，其難度顯然比在SaaS產品上提供一些“可配置功能”要大多了。

### 16.3.3 負重前行的傳統軟件公司

**SaaS**這個商業模式需要有與之相匹配的底層技術基礎設施。我們看到，很多傳統的企業軟件公司雖然知道**SaaS**是軟件的未來，但是卻遲遲不能向**SaaS**方向成功轉型，原因是不知道**SaaS**的技術基礎設施需要怎麼構建，或者沒有資源可以在較長時間內持續地投入到技術基礎設施的建設上。

很多傳統的企業軟件廠商已經習慣了以私有部署的方式為客戶提供服務。多年以前開發的軟件產品在架構上往往存在各種缺陷，比如，可能採用了過時的技術，沒有為**SaaS**做過預先的設計和考量，實現代碼經過反覆修改之後可能冗餘複雜、難以修改和維護。技術團隊身上已經揹負著沉重的技術債，但是為滿足已有客戶的“新需求”，還要繼續進行定製化開發、私有部署、人肉運維，在巨大的工作壓力下疲於奔命。這種軟件廠商想要向**SaaS**轉型談何容易？

## 16.4 SaaS需要DDD

很多傳統企業軟件廠商開發的軟件都是單體應用。想要將這樣的軟件變成服務成千上萬個企業的互聯網應用，尤其需要它們的開發團隊掌握“分而治之”的方法。

即使是私有部署的軟件，組件化的好處也無須多言，只是對互聯網應用來說尤其必要。通過將單體應用拆分為多個軟件組件，我們可以更好地對應用進行局部的優化——不管是軟件功能的改進，還是非功能性的優化，都會容易一些。這對降低SaaS軟件的開發和運維成本尤其重要。

DDD是一個有力的“分而治之”的思想武器。我們可以看到，很多介紹SOA或MSA的文章都會提到DDD。DDD在軟件開發的戰略及戰術層面都給開發人員提供了非常有益的“拆分”複雜軟件的思路。

在戰略層面，DDD提供了限界上下文與防腐層的概念，軟件開發團隊可以用它們來規劃應用軟件的範圍，明確它們之間的邊界，在緊密集成不同應用的同時，仍然保證應用邊界內的概念完整性。

在戰術層面，DDD提供了聚合的概念，開發人員可以用它來在應用內構建高內聚、低耦合的軟件構造塊。基於聚合分析的結果，開發人員可以在恰當地採

用強一致性模型來簡化開發工作的同時，採用最終一致性模型來處理不同聚合之間的數據一致性，以滿足 SaaS 應用對水平擴展能力的需要。

DDD 確實是構建大型 SaaS 的“大殺器”。

## 第17章 更好的“錘子”

我們（筆者以及筆者曾經的同事）認為，DDD不僅可以解決大型軟件的核心複雜問題，而且還可以在保證軟件質量的前提下，讓很多中小型應用的開發進程加速。

我們創造了DDDM<sub>L</sub>這個DDD原生的DSL，並圍繞著它製造了一個工具鏈——我們稱為DDDM<sub>L</sub> Tools，其中包括多種語言的應用代碼的生成工具。它們可以完成的工作包括：

- Java後端服務的生成。因為筆者和同事對生成的代碼風格有不同的偏好，所以我們分別做了各自的實現。筆者開發的代碼生成工具可以生成基於Hibernate ORM的Repository與事件存儲（Event Store），基於JAX-RS或Spring MVC的RESTful Services。另一位同事使用完全不同的模板引擎獨立完成了另外一個實現，其生成的後端代碼的架構風格與筆者的實現有明顯的差異，比如沒有默認使用事件溯源模式。

- .NET後端服務的生成。筆者開發的代碼生成工具可以生成基於NHibernate ORM的Repository與事件存儲，基於ASP.NET Web API的RESTful Services。

- PHP版的RESTful Client SDK的生成。

- .NET 版的 RESTful Client SDK 的生成。
- Java/Android 版的 RESTful Client SDK 的生成。
- Web 前端 TypeScript Client SDK 的生成。
- Java RESTful Services 的 RAML 文檔的生成。
- 管理後臺 Web 應用（我們稱之為 Admin UI ）的生成。不止一個實現，第一個實現使用的是 Vue.js 前端框架，第二個實現使用的是 Angular。它們內在的設計理念非常不同，一個是基於 JSON/DDDML 描述的領域模型元數據，一個是基於 API 的服務描述文檔（使用像 Swagger/OpenAPI、RAML 這樣的服務描述語言）。

基於這些工具開發出來的應用，從前端到後端，都經過了實際生產環境的檢驗。正是因為經過了這些實踐檢驗，我們可以很確信地說：基於 DSL 實現 DDD 並不是花架子。

但關於 DDDML Tools，因為可投入資源的限制，我們還有很多想做卻沒有做好的東西：

- 首先我們想要對 DDDML 文檔進行圖形化的展示。這個工具可以稱為 DDDML Viewer。它可以讀入 DDDML 文檔，展示成圖形（可以用 UML 類圖、狀態機圖等去想象我們想要的圖形）。領域模型應該由整個軟件開發團隊合作構建，被所有團隊成員理解。大家只能看著 DSL 代碼去理解模型顯然不夠完美，“一圖勝

千言”，圖形化展示能幫我們儘快確認DDDML文檔描述的是不是大家所想的那個“領域模型”。

- 我們想要圖形用戶界面的DDDML設計器。不過，這個工具的重要性不如DDDML Viewer的高。因為技術人員手寫YAML/DDDML文檔其實是沒有問題的，而技術人員與非技術人員之間的交流最好有圖形化的展示作為輔助。
- 我們需要可以生成更多語言（Ruby、Python、Go、Swift等）的代碼的工具。
- 我們希望工具生成的代碼質量更好、具備更高的可擴展性。我們需要更多好的代碼示例，然後把它們製作成代碼模板。
- 我們希望工具可以生成更多UI層的代碼，提升由工具生成的GUI應用程序的用戶體驗。
- 我們希望可以生成更多的基於NoSQL數據庫的代碼，讓應用的數據庫可以在SQL和NoSQL之間進行平滑的轉換。
- 我們希望給產品以及測試團隊提供更好的工具，比如從實例化需求文檔的編寫、驗收測試的自動化，到測試數據的構造等方面的開發工具。
- 我們希望這些工具是支持團隊協作的在線應用。

也許，我們需要的是一個基於DDD理念構建的  
PaaS平臺。

## 17.1 我們製作的一個DDDML GUI工具

筆者認為實現具備圖形用戶界面的DDDML工具是很有必要的，這裡介紹一下我們製作的一個**DDDML GUI工具**，我們把它叫作**DDDML Builder**（DDDML構建器）。當然，其實它離我們想要的工具還差得很遠，但是看一看，也許便於你理解我們想要的大概是什麼。

### 17.1.1 紿領域建模提供起點

企業的經營並非無章可循。知名開源ERP系統Apache OFBiz的數據模型就是基於經典圖書《數據模型資源手冊》(The Data Model Resource Book，簡稱DMRB)的描述實現的。DMRB的第一卷闡述的是通用的數據模型，這些數據模型可以在不同行業中使用；第二卷闡述的是面向服務的領域專門化數據模型，即怎麼修改通用的數據模型以應用於特定的行業；第三卷講述數據建模的模式。

在很多開源應用軟件中，還存在各種經典的數據模型，它們一般是關係範式（或ER範式）的。我們可以考慮將這些經典的數據模型以DDD範式重新整理，使用DDDML描述出來。也就是說，我們可以建立一個包羅萬象的DDDML模型庫，為軟件開發團隊的業務分析、領域建模工作提供一個起點。

或者更簡單一點，我們直接收集各個領域的關係數據模型（或ER模型），並將DDDML工具——將關係模型轉換到DDDML模型、編輯DDDML模型的工具——開放出來，給需要的人使用，然後鼓勵大家共享DDDML模型。

目前，我們的DDDML Builder支持從OFBiz開源項目中搜索、轉換其中的數據模型。

下面以一個迷你Cookbook的方式，看一下怎麼使用這個DDML Builder。

### 17.1.2 創建新的限界上下文

當需要創建一個新的限界上下文時（有時候我們把它叫作項目），並不需要使用事先製作的項目模板，可直接使用之前已有的（特別是在生產環境中實際使用的）應用的開發過程中用過的生成配置文件。我們通過替換這個配置文件中與已有項目（源項目）相關的關鍵字或設置項，產生一個新的限界上下文的生成配置文件。然後，調用我們製作的項目創建工具，基於這個新的生成配置文件，生成新的項目。在創建新項目的過程中，當需要從已有項目複製部分實現代碼的時候，工具也會替換代碼中與源項目相關的關鍵字或設置項。

我們這麼做，是為了避免維護項目模板的麻煩以及從過時的項目模板中創建新項目可能導致的缺陷。

使用圖17-1所示的用戶界面創建一個新的限界上下文的過程大致如下：

1 ) 點擊“Read”按鈕，讀入作為模板已有的限界上下文的生成配置文件的信息。

2 ) 然後在“BC Root Directory”輸入框中，輸入想要創建的新的限界上下文（項目）的根目錄的路徑。

3 ) 在標籤“BC Name(PascalCase.Plz)”右邊的文本框中，輸入需要創建的新的限界上下文的名稱。這個名稱必須是PascalCase風格的（如果不是，會提示輸入錯誤），一般由“組織名稱”與“軟件名稱”兩部分以點分的形式組成，比如：Lingyuqudong.Crm。

4 ) 檢查需要創建的新限界上下文的其他選項，視需要進行修改。比如修改測試數據庫的連接URL、用戶名、密碼等。然後，點擊“Gen. Config File”按鈕，創建新的限界上下文的生成配置文件。

5 ) 點擊“Create BC”按鈕，根據上一步的生成配置文件，創建出新的限界上下文（項目）。

6 ) 點擊“Watch”按鈕，運行DDDM<sub>L</sub>模型目錄監視器程序。這個程序會監視新創建的限界上下文的DDDM<sub>L</sub>模型目錄（該目錄用於存放描述領域模型的DDDM<sub>L</sub>文件，一般命名為“dddml”，位於項目根目錄下）。當該目錄中的文件發生變化時，監視器程序會調用代碼模板轉換引擎，生成或更新項目中與模型相關的源代碼。

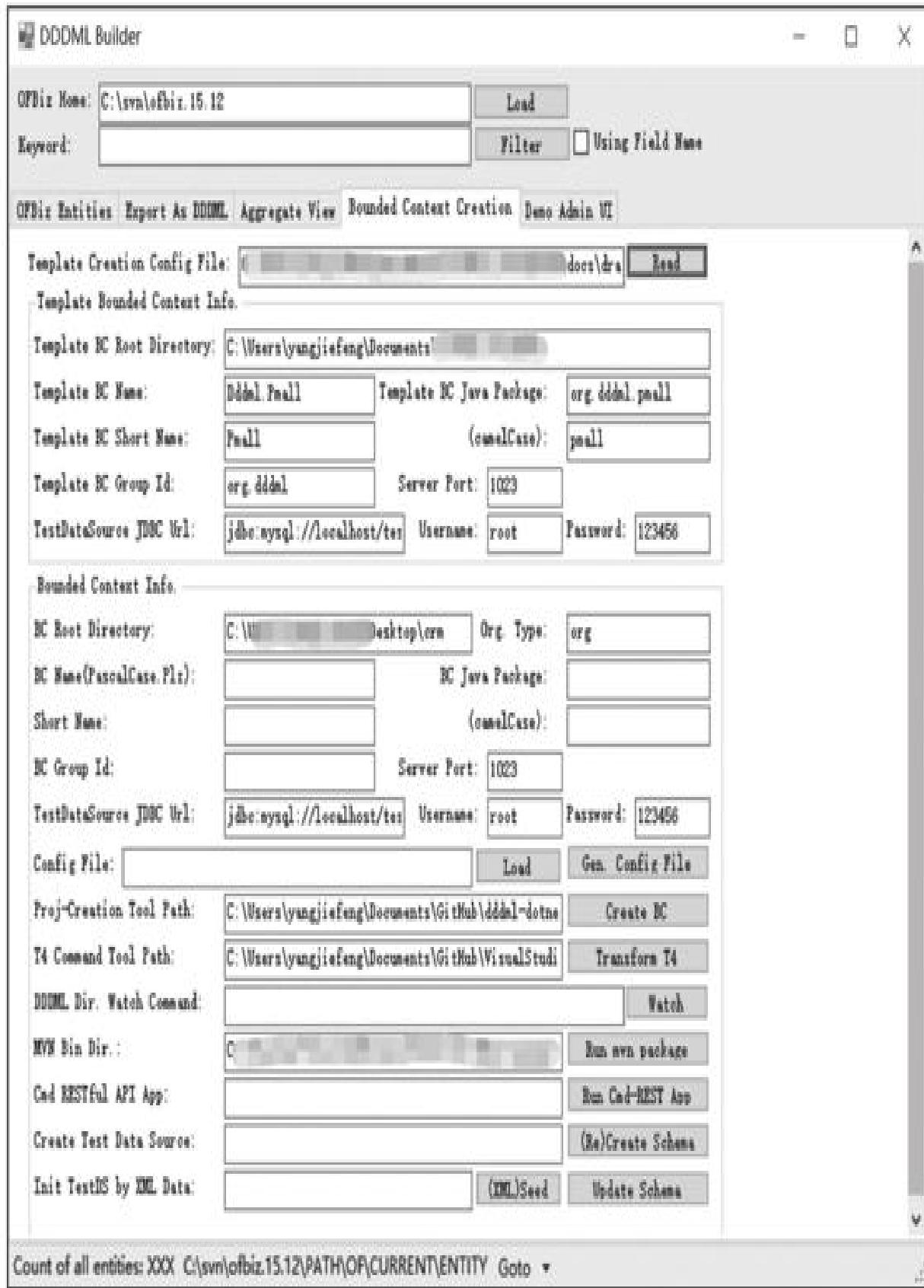


圖17-1 創建新的限界上下文 ( DDDML項目 )

### 17.1.3 從OFBiz中“借鑑”數據模型

目前DDDML Builder工具僅支持從開源項目OFBiz的源代碼中搜索數據模型，但我們的理想是構建一個包羅萬象的模型庫。模型庫中的模型以“限界上下文”進行分組，每個限界上下文的模型可能來源於各行各業、不同領域的開源軟件項目。

OFBiz的數據模型是ER範式的，使用XML描述。下面嘗試從OFBiz中選擇一個實體，轉換為DDDML模型，導入到上面新創建的限界上下文（Lingyuqudong.Crm）中。

切換到如圖17-2中所示的“OFBiz Entities”標籤頁。確保在“OFBiz Home”輸入框中正確地輸入了OFBiz源代碼所在目錄的路徑，點擊“Load”按鈕，載入OFBiz的實體模型信息。

在“Keyword”輸入框中輸入StatusItem，敲回車鍵或點擊“Filter”按鈕，以StatusItem作為關鍵字過濾實體。然後點擊左邊實體樹形視圖的StatusItem結點，查看該實體在OFBiz中的實體模型定義（以XML格式描述），以及在OFBiz源代碼目錄中存在的該實體的數據（XML格式），結果如圖17-3所示。

DDML Builder

Office Name: C:\svn\ofbiz.15.12    Using Field Name

OFBiz Entities [Export As DDML](#) [Aggregate View](#) [Bounded Context Creation](#) [Done](#) [Admin UI](#)

AccommodationClass  
 AccommodationMap  
 AccommodationType  
 AccommodationSpot  
 AcctgTrans  
 AcctgTransAttribute  
 AcctgTransEntry  
 AcctgTransEntryType  
 AcctgTransType  
 AcctgTransTypeAttr  
 Addressbook  
 AddressbookEntry  
 Affiliate  
 Agreement  
 AgreementAttribute  
 AgreementEmploymentAppl  
 AgreementFacilityAppl  
 AgreementGeographicalAppl  
 AgreementItem  
 AgreementItemAttribute  
 AgreementItemType  
 AgreementItemTypeAttr  
 AgreementPartyAppl  
 AgreementProductAppl  
 AgreementFromAppl  
 AgreementRole  
 AgreementTerm  
 AgreementTermAttribute  
 AgreementType  
 AgreementTypeAttr  
 AgreementWorkEffortAppl  
 ApplicationSandbox  
 AuditDataset  
 BenefitType  
 BillingAccount  
 BillingAccountType  
 BillingAccountTypeTerm

Start-Point Entities

Current Entity Data

Count of all entities: 880 C:\svn\ofbiz.15.12\PATH\OF\CURRENT ENTITY Goto ▾

圖17-2 載入OFBiz的實體模型信息

OFBiz Entities Export As DDDML Aggregate View Bounded Context Creation Demo Admin UI

OFBiz Name: C:\svn\ofbiz.15.12 Keyword: StatusItem Load Filter  Using Field Name

OK! To Export

Start-Point Entities Current Entity Model

```
<entity entity-name="StatusItem" package="com.terracotta.ofbiz.15.12">
  <field name="statusId" type="id">
  </field>
  <field name="statusTypeId" type="id">
  </field>
  <field name="statusCode" type="short">
  </field>
  <field name="sequenceId" type="id">
  </field>
  <field name="description" type="string">
  </field>
  <primary field="statusId" />
  <relation type="one" fk-name="STATUS_D">
    <key field-name="statusTypeId" />
  </relation>

```

Remove < >

Current Entity Data

```
<?xml version="1.0" encoding="UTF-8"?>
<entity>
  <!-- C:\svn\ofbiz.15.12\applications\accounting\data\Accounting.15.12\StatusItem.xml -->
  <!-- StatusItem description="Not Reconciled" sequenceId="01" statusCode="01" -->
  <!-- StatusItem description="Partly Reconciled" sequenceId="02" statusCode="02" -->
  <!-- StatusItem description="Reconciled" sequenceId="03" statusCode="03" -->
  <!-- StatusItem description="Active" sequenceId="01" statusCode="ACTI" -->
  <!-- StatusItem description="Negative Pending Replenishment" sequenceId="04" statusCode="NPR" -->

```

(Re)Load Data Save As...

Count of all entities: 880 C:\svn\ofbiz.15.12\framework\common\entitydef\entitymodel.xml Goto ...

### 圖17-3 以StatusItem作為關鍵字過濾OFBiz實體

點擊“Save As...”按鈕，可以保存實體的數據文件。這裡可以將其保存到新創建的限界上下文根目錄下的Data目錄中，命名為StatusItemData.xml，以便後面測試的時候可以使用它。

然後，使用鼠標將樹形視圖的StatusItem結點拖入“Start-Point Entities”列表，結果如圖17-4所示。



#### 圖17-4 將StatusItem結點拖入“Start-Point Entities”列表

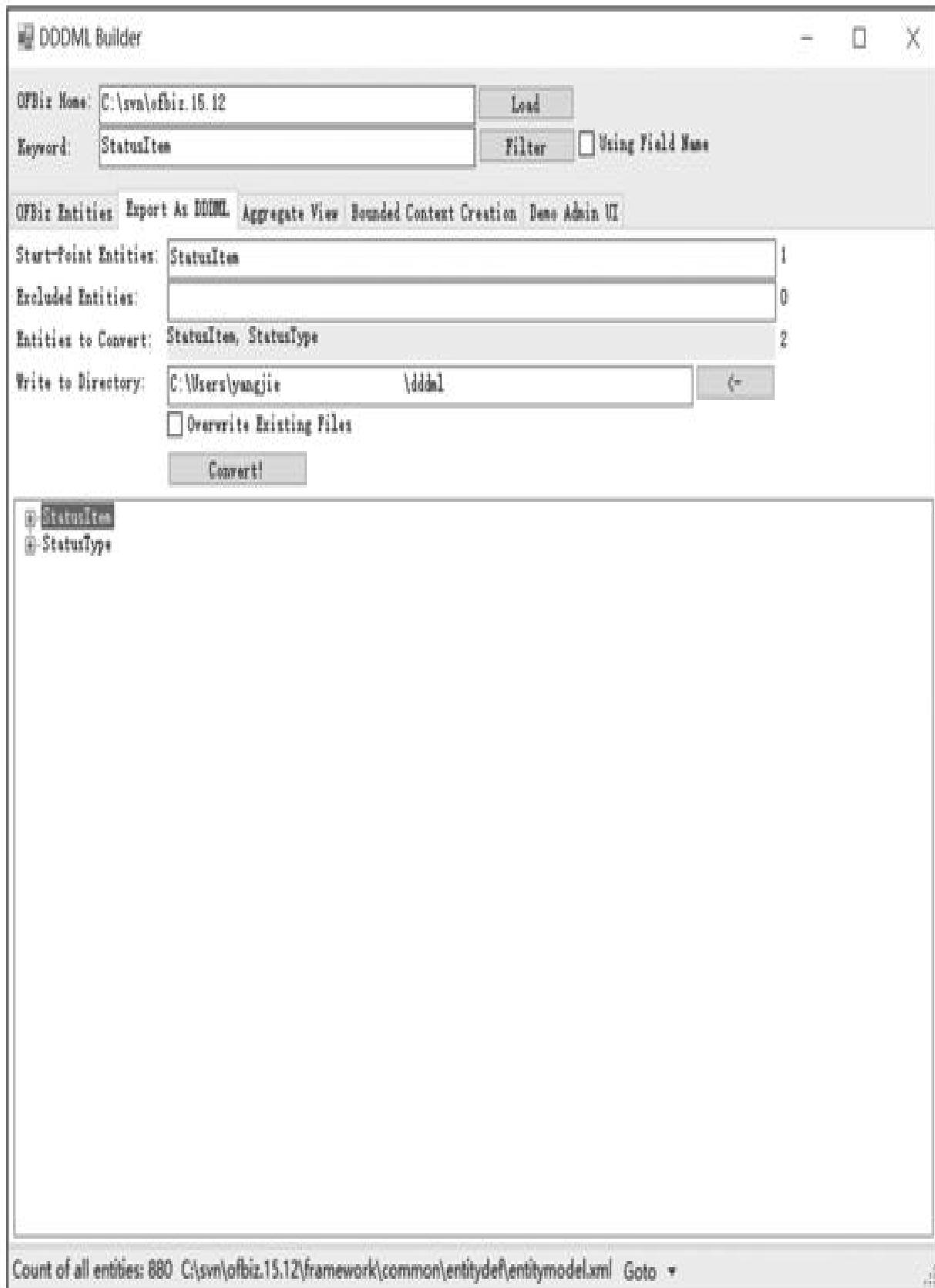
點擊“OK! To Export”按鈕，進入另外一個標籤頁（該標籤頁提供將實體模型導出為DDML文件的功能），如圖17-5所示。

這裡假設我們並不希望導出與StatusItem實體存在關聯的StatusType實體，可以將圖17-5下方的樹形視圖的StatusType結點拖入“Excluded Entities”輸入框，將其排除。

然後，點擊“Convert!”按鈕，將選中實體（這裡只有一個StatusItem）轉換為DDML文檔並導出到新創建的限界上下文的DDML模型目錄中。DDML模型目錄監視器程序在檢測到目錄中的文件發生變化後，會自動生成或更新模型相關的源代碼。

#### 17.1.4 構建項目並運行應用

切換回“Bounded Context Creation”標籤頁，點擊“Run mvn package”按鈕，使用Maven<sup>[1]</sup>工具構建項目。



## 圖17-5 將實體導出為DDDMIL文件的標籤頁

在看到彈出構建成功的提示信息之後，關閉提示窗口，點擊“(Re)Create Schema”按鈕，創建運行應用需要的數據庫模式（Schema）。

在Schema創建成功之後，接著可以點擊“(XML)Seed”按鈕，導入在Data目錄中保存的實體數據文件（這裡會導入之前保存的StatusItemData.xml文件的內容）。

在看到彈出“初始化數據成功”的信息提示之後，關閉提示窗口，然後點擊“Run Cmd-REST App”，啟動新創建的限界上下文的RESTful API服務。

打開瀏覽器，在地址欄輸入如下命令：

---

```
http://localhost:1023/api/StatusItems
```

---

應該可以看到如圖17-6所示的頁面，證明我們已經可以通過HTTP GET請求獲取已導入的StatusItem實體的數據。

[1] 見<http://maven.apache.org/>。

## 17.1.5 使用HTTP PUT方法創建實體

我們可以試試使用HTTP PUT方法創建StatusItem 實體。向以下URL：

---

```
http://localhost:1023/api/StatusItems/TEST_STATUS_1
```

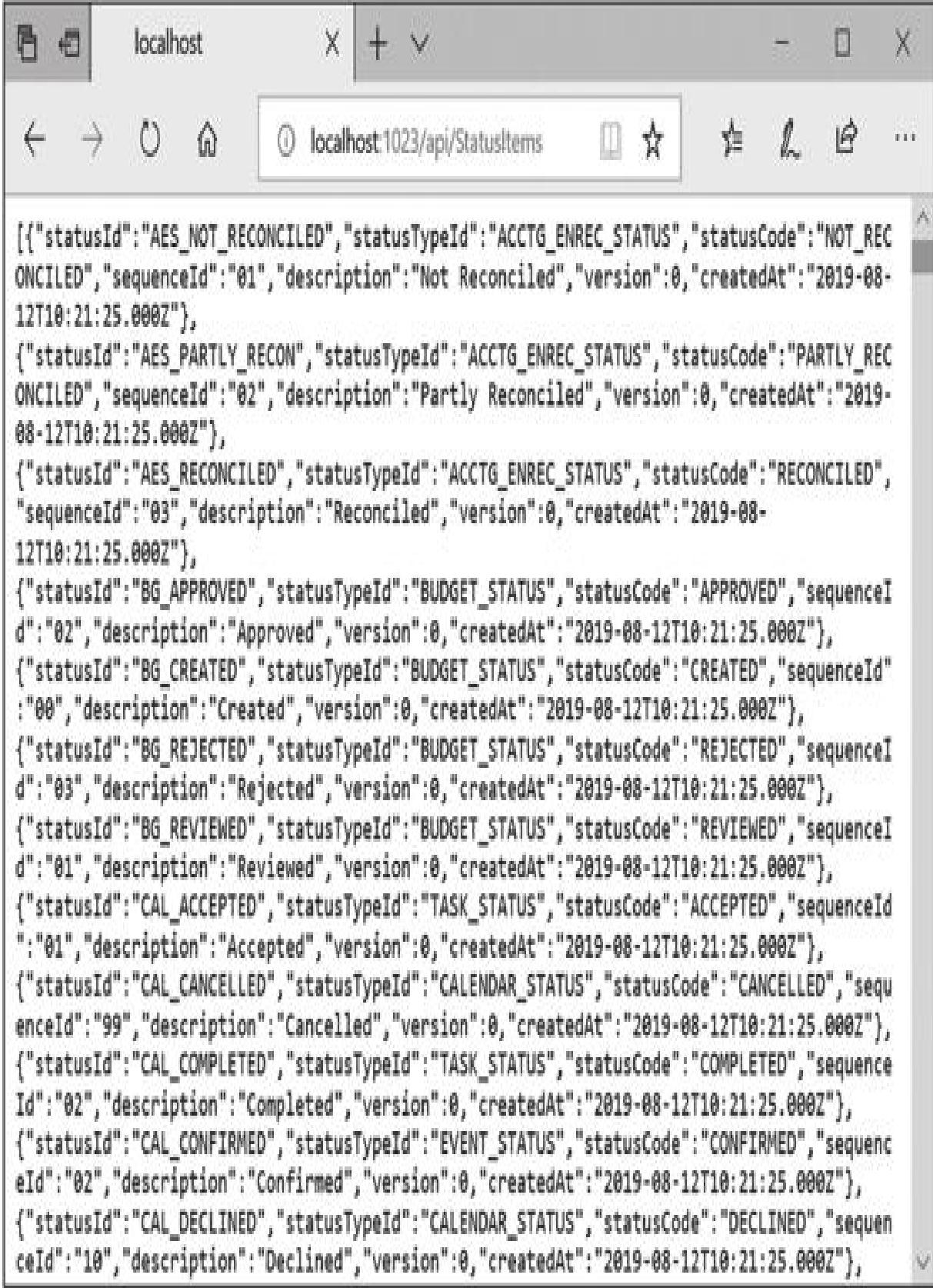
---

發送HTTP PUT請求，請求的JSON消息體如下：

---

```
{  
  "statusTypeId": "TEST_STATUS",  
  "statusCode": "TEST STATUS CODE",  
  "sequenceId": "01",  
  "description": "Test status"  
}
```

---



A screenshot of a Microsoft Edge browser window. The address bar shows the URL `localhost:1023/api/StatusItems`. The main content area displays a JSON array of 18 status items, each with a unique status ID, type ID, code, description, version, and creation timestamp. The items are categorized into four main types: ACCTG\_ENREC\_STATUS, BUDGET\_STATUS, TASK\_STATUS, and CALENDAR\_STATUS, with sequence IDs ranging from 01 to 99.

```
[{"statusId": "AES_NOT_RECONCILED", "statusTypeId": "ACCTG_ENREC_STATUS", "statusCode": "NOT_RECONCILED", "sequenceId": "01", "description": "Not Reconciled", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "AES_PARTLY_RECON", "statusTypeId": "ACCTG_ENREC_STATUS", "statusCode": "PARTLY_RECONCILED", "sequenceId": "02", "description": "Partly Reconciled", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "AES_RECONCILED", "statusTypeId": "ACCTG_ENREC_STATUS", "statusCode": "RECONCILED", "sequenceId": "03", "description": "Reconciled", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "BG_APPROVED", "statusTypeId": "BUDGET_STATUS", "statusCode": "APPROVED", "sequenceId": "02", "description": "Approved", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "BG_CREATED", "statusTypeId": "BUDGET_STATUS", "statusCode": "CREATED", "sequenceId": "00", "description": "Created", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "BG_REJECTED", "statusTypeId": "BUDGET_STATUS", "statusCode": "REJECTED", "sequenceId": "03", "description": "Rejected", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "BG_REVIEWED", "statusTypeId": "BUDGET_STATUS", "statusCode": "REVIEWED", "sequenceId": "01", "description": "Reviewed", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "CAL_ACCEPTED", "statusTypeId": "TASK_STATUS", "statusCode": "ACCEPTED", "sequenceId": "01", "description": "Accepted", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "CAL_CANCELLED", "statusTypeId": "CALENDAR_STATUS", "statusCode": "CANCELLED", "sequenceId": "99", "description": "Cancelled", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "CAL_COMPLETED", "statusTypeId": "TASK_STATUS", "statusCode": "COMPLETED", "sequenceId": "02", "description": "Completed", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "CAL_CONFIRMED", "statusTypeId": "EVENT_STATUS", "statusCode": "CONFIRMED", "sequenceId": "02", "description": "Confirmed", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}, {"statusId": "CAL_DECLINED", "statusTypeId": "CALENDAR_STATUS", "statusCode": "DECLINED", "sequenceId": "10", "description": "Declined", "version": 0, "createdAt": "2019-08-12T10:21:25.000Z"}]
```

## 圖17-6 通過HTTP GET請求獲取StatusItem實體的數據

應該可以得到從服務端返回的狀態碼為200的響應。然後，在瀏覽器地址欄輸入如下URL：

---

```
http://localhost:1023/api/StatusItems/TEST_STATUS_1
```

---

可以獲得類似如下的響應消息體：

---

```
{"statusId": "TEST_STATUS_1", "statusTypeId": "TEST_STATUS", "statusCode": "TEST STATUS CODE", "sequenceId": "01", "description": "Test status", "version": 0, "createdAt": "2019-08-13T03:17:18.000Z"}
```

---

這說明我們已經成功創建了一個新的StatusItem實例。

### 17.1.6 給聚合增加方法

切換到“Aggregate View”標籤頁，確保在“DDDML Directory or Project File”輸入框中輸入的是之前新創建的限界上下文的DDDML模型目錄的路徑，點擊“Load”按鈕加載上下文的模型信息。

在“Aggregate Name”下拉列表中選中“StatusItem”這個聚合，點擊“View”按鈕，結果如圖17-7所示。

鼠標右鍵點擊左邊的聚合樹形視圖中的任意結點，在彈出的快捷菜單中選擇“Add Method”，如圖17-8所示。

DDOML Builder

OFBiz Name: C:\svn\ofbiz.15.12

Keyword: StatusItem   Using Field Name

OFBiz Entities Export as DDOML Aggregate View Bounded Context Creation Demo Admin UI

DDOML Directory or Project File: C:\Users\yang\Iefeng\Desktop\workspace\src\ddml

Aggregate Name: StatusItem    Only Conceptual

StatusItem (AGGREGATE)  
   StatusItem (AGGREGATE ROOT)  
     StatusId: id -ne (ID)  
     properties  
       StatusTypeId: id-ne  
       StatusCode: short-varchar  
       SequenceId: id  
       Description: description  
       Version: long  
       CreatedBy: string  
       CreatedAt: Date/Time  
       UpdatedBy: string  
       UpdatedAt: Date/Time  
       Active: bool  
       Deleted: bool

Wkspace Directory: C:\Users\yang\Iefeng\Desktop\workspace

Current Agg Method:

Source Project Dir.: C:\Users\yang\Iefeng\Desktop\workspace\src\ddml

Current Agg. File: C:\Users\yang\Iefeng\Desktop\workspace\src\ddml\StatusItem.yaml

Node Properties

Name:

Type:

Count of all entities: 880 C:\svn\ofbiz.15.12\framework\common\entitydef\entitymodel.xml Goto

圖17-7 查看聚合StatusItem

DDML Builder

OFBiz Home: C:\svn\ofbiz.15.12  Keyword: StatusItem   Using Field Name

OFBiz Entities Export As DDML Aggregate View Bounded Context Creation Demo Admin UI

DDML Directory or Project File: C:\Users\yangji\fang\Desktop\src\ddml  Aggregate Name: StatusItem    Only Conceptual

StatusItem (AGGREGATE)  
   StatusItem (ACCUMULATE STATE)  
    StatusID   
    properties  
      >StatusTypeID: id<ns  
      -StatusCode: short-varchar  
      -SequenceID: id  
      -Description: description  
      -Version: long  
      -CreatedBy: string  
      -CreatedAt: Date/Time  
      -UpdatedBy: string  
      -UpdatedAt: Date/Time  
      -Active: bool  
      -Deleted: bool

Workspace Directory: C:\Users\yangji\workspace  
Current Agg Method:  
Source Project Dir.: C:\Users\yangji\src-common

Current Agg File: C:\Users\yangji\src\ddml>StatusItem.yaml

Node Properties

Name:   
Type:

Count of all entities: 880 C:\svn\ofbiz.15.12\framework\common\entitydef\entitymodel.xml Goto ▾

圖17-8 快捷菜單Add Method

現在，為實體添加的方法名為ChangeCode。該方法的參數只有一個，參數名為Code，其類型選擇為short-varchar。添加方法的界面如圖17-9所示。

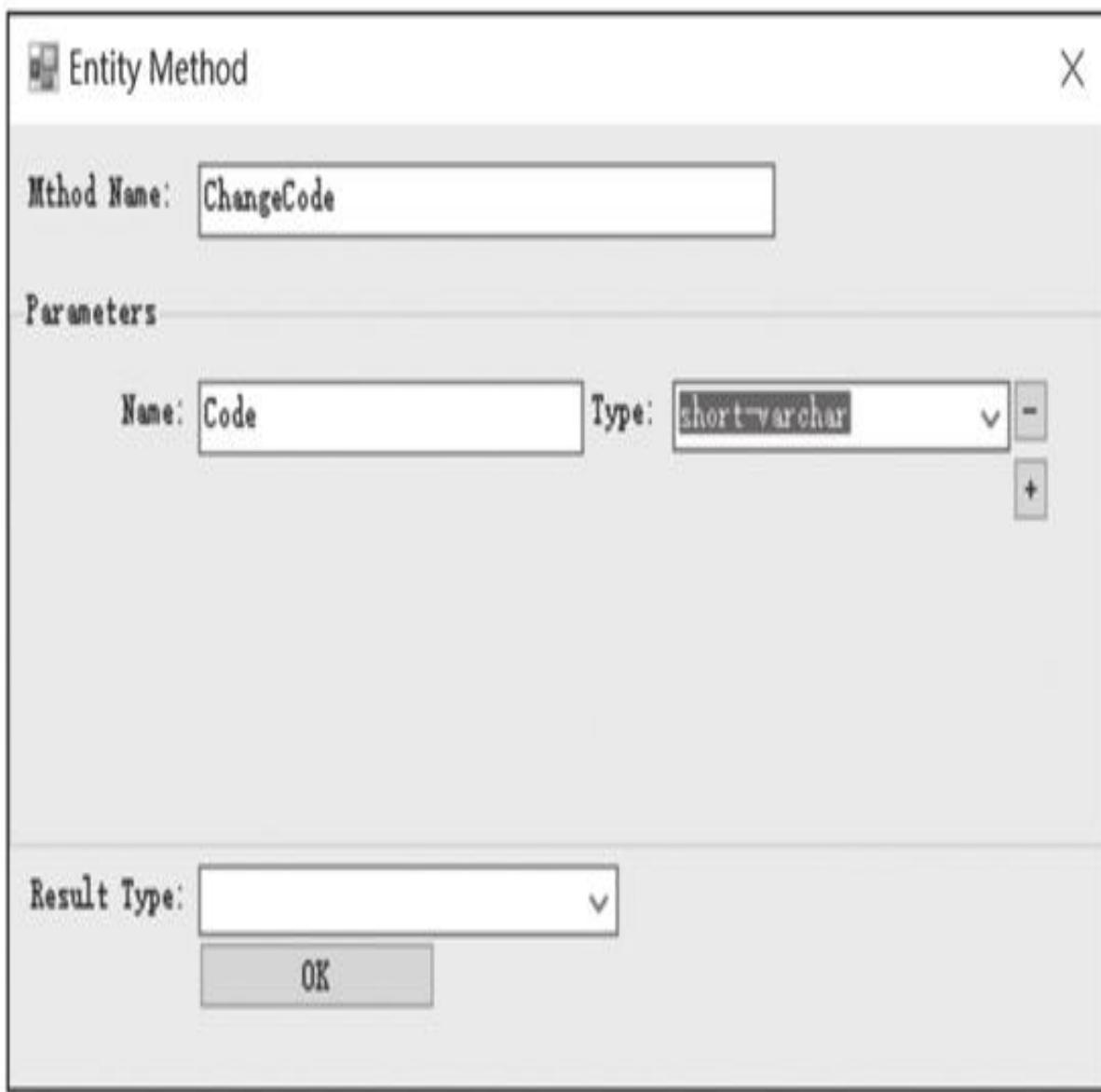


圖17-9 紿StatusItem添加方法ChangeCode

在這個類型下拉列表中，可以看到OFBiz抽象的各種基礎類型（值對象），雖然這些類型仍然有點“技術化”，但是這表明為一個界限上下文構建一套領域基礎類型是可行的。這些類型之所以能夠映射到具體語言的實現，是因為存在（我們已經事先編寫好的）如下的類型定義信息：

---

```
typeDefinitions:  
  id:  
    sqlType: VARCHAR(20)  
    cSharpType: string  
    javaType: String  
  short-varchar:  
    sqlType: VARCHAR(60)  
    cSharpType: string  
    javaType: String  
  long-varchar:  
    sqlType: VARCHAR(255)  
    cSharpType: string  
    javaType: String  
  # ...
```

---

點擊添加方法窗口的“OK”按鈕，返回前一個窗口，應該可以看到ChangeCode方法已經添加好了，如圖17-10所示。

接下來，就是實現ChangeCode方法的業務邏輯了。

## 1. 實現實體方法的業務邏輯

確保左邊聚合樹形視圖中的“ChangeCode”結點被選中，在“Workspace Directory”中輸入你想要的工作空

間目錄的路徑，這個目錄用來存放實現該方法業務邏輯的Maven項目（“Method POM”）。

DOOML Builder

Office Name: C:\svn\ofbiz 15.12  Keyword: StatusItem   Bring Field Name

Office Entities Export As DOOML Aggregate View Bounded Context Creation Demo Admin UI

DOOML Directory or Project File: C:\Users\yungjifeng\Desktop\src\ddml  Aggregate Name: StatusItem    Only Conceptual

StatusItem (AGGREGATE)  
 StatusItem (AGGREGATE ROOT)  
  - StatusId: id-na (ID)  
   properties  
    - StatusTypeId: id-na  
    - StatusCode: short-varchar  
    - SequenceId: id  
    - Description: description  
    - Version: long  
    - CreatedBy: string  
    - CreatedAt: Datetime  
    - UpdatedBy: string  
    - UpdatedAt: Datetime  
    - Active: bool  
    - Deleted: bool  
   methods  
     ChangeCode (METHOD)  
       parameters  
        - StatusId: id-na  
        - Code: short-varchar  
        - Version: long  
        - CommandId: string  
        - RequesterId: string

Workspace Directory: C:\Users\yungjifeng\Desktop\workspace  
Current Agg Method: StatusItem/ChangeCode  
Source Project Dir.: C:\Users\yungjifeng\Desktop\src\src-common  
   
  
Current Agg. File: C:\Users\yungjifeng\Desktop\src\ddml>StatusItem.yml

Node Properties

Name:   
Type:

Count of all entities: 880 C:\svn\ofbiz 15.12\framework\common\entitydef\entitymodel.xml Goto ▾

## 圖17-10 添加了ChangeCode方法的StatusItem

點擊“Generate Method POM”按鈕，應該可以看到“Maven項目已生成”的提示信息。

點擊“Show in Explorer”，可以打開系統的文件瀏覽器直接定位到該項目的POM文件。

可以使用你喜歡的IDE，比如IntelliJ IDEA，打開這個Maven項目。找到如下文件：

src/main/java/org/lingyuqudong/crm/domain/statusitem/ChangeCodeLogic.java，你需要在這個文件中編寫StatusItem.ChangeCode方法的業務邏輯。

這個ChangeCodeLogic類中的verify方法可以暫時先不管它：

---

```
public static void verify(StatusItemState
statusItemState, String code, VerificationContext
verificationContext) {
}
```

---

在它的mutate方法中需要填入ChangeCode方法“修改StatusItem的狀態”那部分的業務邏輯代碼，示例如下：

---

```
public static StatusItemState mutate(StatusItemState
statusItemState, String code,
MutationContext<StatusItemState,
StatusItemState.MutableStatusItemState> mutationContext) {
```

```
        StatusItemState.MutableStatusItemState
mutableStatusItemState =
mutationContext.createMutableState(statusItemState);
        mutableStatusItemState.setStatusCode(code);
        return mutableStatusItemState;
    }
```

---

但是，筆者更喜歡如下的函數式編程風格：

---

```
    public static StatusItemState mutate(StatusItemState
statusItemState, String code,
MutationContext<StatusItemState,
StatusItemState.MutableStatusItemState> mutationContext) {
    return new StatusItemState() {
        @Override
        public String getStatusId() {
            return statusItemState.getStatusId();
        }

        @Override
        public String getStatusTypeId() {
            return statusItemState.getStatusTypeId();
        }

        @Override
        public String getStatusCode() {
            return code;
        }
        // 省略其他代碼
    };
}
```

---

你可以試著在**IDE**中編譯、測試這個方法。想要對這個方法做個單元測試？可以簡單地在這個文件中寫一個靜態的**main**方法，然後執行一下看看，這裡連單元測試框架都不需要。

現在，我們將實現好的方法邏輯複製到DDDML項目，只需要點擊DDDML Builder中的“Copy Back to Src.Project”按鈕即可。

如果DDDML Builder是一個在線的工具，那麼這一步應該是將代碼上傳到服務器——如果資源允許，其實我們特別想基於DDDML構建一個Serverless平臺。

## 2. 測試添加的方法

關閉之前啟動的RESTful API服務。點擊“Run mvn package”按鈕，重新編譯、打包項目。然後點擊“Run Cmd-REST App”按鈕，確保RESTful API服務已經重新啟動。

發送HTTP PUT消息到如下URL：

---

```
http://localhost:1023/api/StatusItems/TEST_STATUS_1/_commands/ChangeCode
```

---

這個PUT請求的JSON消息體是：

---

```
{  
  "code": "NEW TEST STATUS CODE",  
  "version": 0  
}
```

---

如果得到200狀態碼的回應，則說明方法已經調用成功。

然後在瀏覽器地址欄輸入如下URL：

---

`http://localhost:1023/api/StatusItems/TEST_STATUS_1`

---

獲得類似的響應消息：

---

```
{"statusId": "TEST_STATUS_1", "statusTypeId": "TEST_STATUS", "statusCode": "NEW TEST STATUS CODE", "sequenceId": "01", "description": "Test status", "version": 1, "createdAt": "2019-08-13T03:17:18.000Z", "updatedAt": "2019-08-13T05:57:05.000Z"}
```

---

可以看到，這個StatusItem的statusCode屬性已經更新（已經被修改為“NEW TEST STATUS CODE”）。

### 17.1.7 生成限界上下文的Demo Admin UI

切換到“Demo Admin UI”標籤頁，點擊“<- From Created BC”按鈕，快速輸入最近創建的限界上下文的信息，如圖17-11所示。

DDDML Builder

OFBiz Home: C:\svn\ofbiz.15.12  Keyword: StatusItem   Using Field Name

OFBiz Entities Export As DDDML Aggregate View Bounded Content Creation Demo Admin UI

Source Project Directory: C:\Users\lys \pall\4

Target Project Directory: C:\Users\lys \crn\demo-admin-ui

Download Client From: http://47.105.144.230:8081

Download Client Name: '@metadata-driven-admin/demo-for-yang'

Proxy Target API Url: http://localhost:1023/api

Metadata JDBC Url: jdbc:mysql://localhost/test Username: root Password: 123456

Server Port: 1034

Demo Admin UI App: java -Xms128m -Xmx128m -jar C:\Users\lys \crn\demo-admin-

Count of all entities: 880 C:\svn\ofbiz.15.12\framework\common\entitydef\entitymodel.xml Goto ▾

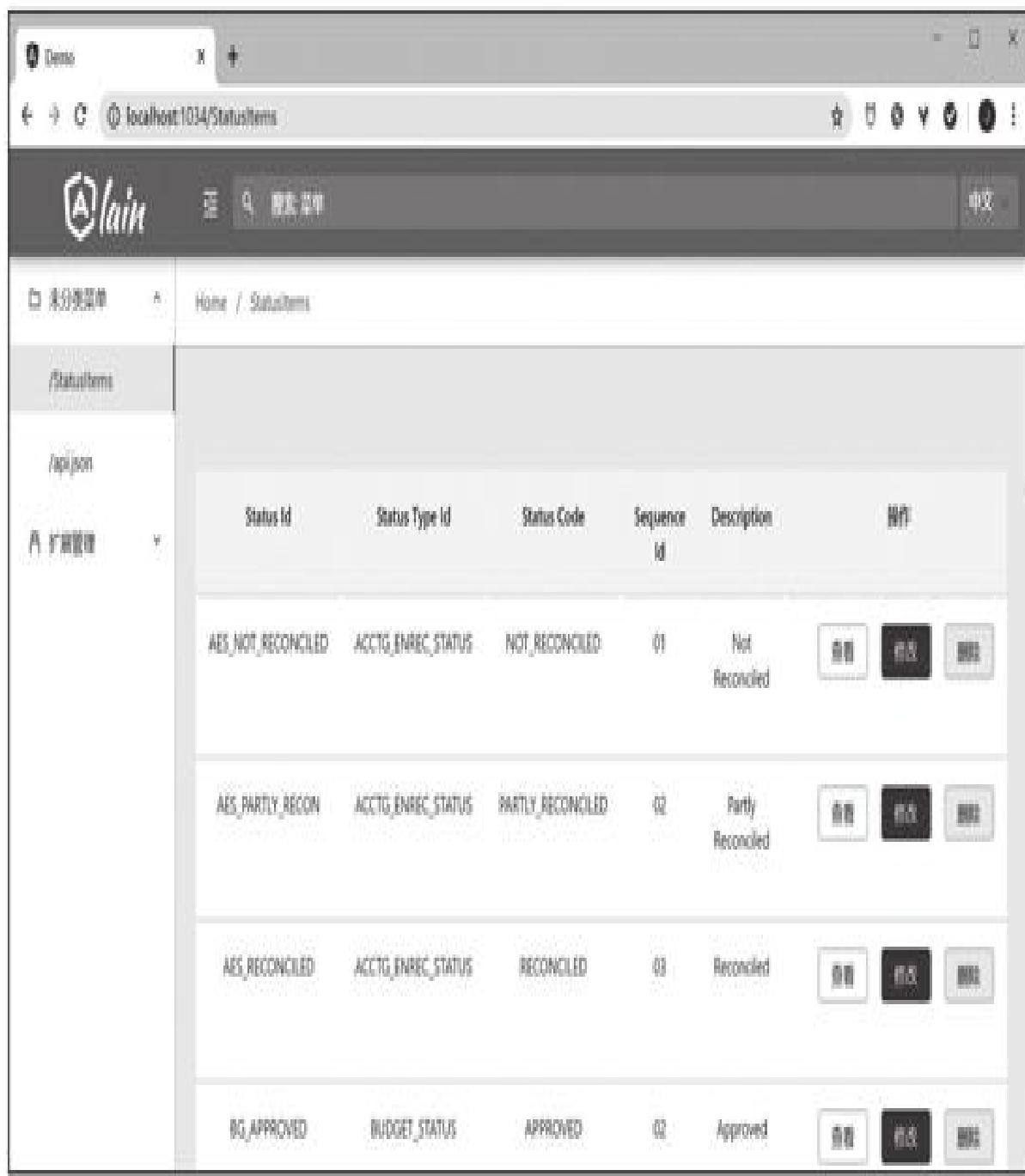
## 圖17-11 創建Demo Admin UI的標籤頁

點擊“Create Project”按鈕，生成這個Demo Admin UI的項目，這個項目是獨立於之前生成的RESTful API 服務項目的。

看到創建項目成功的提示後，點擊“Run mvn pacakge”，編譯、打包這個項目。

看到打包成功的提示後，點擊“Run Admin App”，運行這個Demo Admin UI項目。

打開瀏覽器，地址欄輸入`http://localhost:1034`，進入Admin UI。點擊“未分類菜單”下的“/StatusItems”子菜單，瀏覽顯示頁面如圖17-12所示。



The screenshot shows a web-based administration interface for a system named 'Demo'. The browser address bar indicates the URL is `localhost:1034/StatusItems`. The main content area displays a table of 'StatusItem' entities. The table has columns: Status Id, Status Type Id, Status Code, Sequence Id, Description, and three small icons (Edit, Delete, and Details) for each row. There are four rows of data in the table.

	Status Id	Status Type Id	Status Code	Sequence Id	Description	
	00000000-0000-0000-0000-000000000000	ACCTG_ENREC_STATUS	NOT_RECONCILED	01	Not Reconciled	  
	00000000-0000-0000-0000-000000000001	ACCTG_ENREC_STATUS	PARTLY_RECONCILED	02	Partly Reconciled	  
	00000000-0000-0000-0000-000000000002	ACCTG_ENREC_STATUS	RECONCILED	03	Reconciled	  
	00000000-0000-0000-0000-000000000003	BUDGET_STATUS	APPROVED	02	Approved	  

圖17-12 Demo Admin UI中的StatusItem列表

## 17.1.8 讓不同層級的開發人員各盡其能

想要控制好複雜應用軟件的質量，我們需要更明確的分工，讓不同能力的人幹不同的事情。通過上面使用**DDDML Builder**的演示，也許你已經看到了這樣的可能性：使用**DDDML**和基於**DDDML**的工具，可以將複雜應用軟件的開發任務進行合理分解，讓能力處於不同層級的開發人員各盡其能。

### 1. 初級開發人員做兩件事

初級開發人員只需要（也只能）做兩件事：

- 參與以**DDDML**描述的領域模型的設計。**DDDML**文件可以手動編寫，也可以通過圖形化設計器來產生。領域模型的構建、維護領域模型的概念完整性需要整個團隊的參與。領域模型使用獨立的**DDDML**文件進行描述，和其他代碼文件有效隔離，便於團隊進行管理和評審。
- 編寫實體的方法的實現邏輯。

初級開發人員不需要掌握越來越複雜的**IDE**，不需要了解**Maven**這樣的項目管理工具的諸多“高級”用法，不需要了解**IoC**、**AOP**、**ORM**、**Web MVC**等開發框架，基本上只需要掌握初級的編程語言語法，就可以承擔相當部分業務邏輯的編碼工作。

## 2.高級開發人員應該做的

我們把這樣的開發任務留給高級開發人員：

·封裝高質量的領域基礎類型類庫。像**Joda Time**、**Joda Money**這樣的類庫絕不是初級開發人員短時間內可以寫出來的。在很多領域我們可能都需要這樣的類庫。有了這樣的類庫，我們在**DDDM**文檔中聲明屬性、參數的類型時就有了更具表現力的選擇，而不是隻能使用類似**string**、**int**、**long**、**boolean**這些基本類型。初級開發人員基於這些類庫實現業務邏輯也會少犯很多錯誤。

·在實體的方法的基礎上，實現跨聚合的領域服務。這裡可能涉及最令開發人員頭痛的最終一致性的處理。不過，一些簡單的領域服務也可以考慮留給初中級開發人員實現。

·**DDDM**工具鏈、技術基礎設施的開發。原則就是將在不同領域中重複碰到的共性問題的解決方案“下沉”到**DDDM**的規範、工具鏈以及相關的技術基礎設施中。比如我們已經做過的對賬務模式的初步支持、對樹結構的支持等。

·其他更有挑戰性的開發任務。

也就是說，一個應用的業務邏輯需要被切分到不同的地方：領域基礎類型（它們屬於值對象）的方

法；實體的方法；跨聚合的服務。坦白地講，我們認為很多初級程序員一開始可能僅具備把第二件事情做好的能力。

## 17.2 以統一語言建模

筆者所在的技術團隊曾經想過開發一個叫作“詞彙表”的App。我們開發這個App的目的，是想幫助開發團隊在開發應用時更容易地構建和維護一個“詞彙表”。同樣遺憾的是，由於資源不足，這個App也是做了個開頭就放到一邊去了。

為什麼我們想要這個App呢？因為它能解決最難的事情：

在計算機科學中只有兩件難事：清除緩存以及命名問題。

對於大多數開發人員來說，命名問題比清除緩存還要難。在Quora網站上有過“程序員最難的任務”的投票，半數程序員認為最難的事情是“Naming things”。

要想提高代碼質量，好的命名至關重要。

筆者在國內程序員寫的代碼中看到過很多奇奇怪怪的命名。比如，用“Hair”表示“發送（Send）”的“發”。不過，這畢竟是少數，但是，詞性（名詞、動詞、形容詞、副詞等）、名詞單複數形式的使用錯誤就非常普遍了。這大概與我們的母語有關。中文的很多詞語的詞性是很靈活的，需要放到特定的上下文中才能確定。而我們要使用名詞的複數形式時，一般只

需要在名詞的單數形式的後面加一個“們”字就可以了，有的時候“們”都可以沒有。比如，“小夥伴們來吃飯”這句話完全可以簡化為“小夥伴來吃飯”。

對很多軟件開發團隊來說，都有必要給自己負責開發的每個應用弄一個“單詞表”，大夥兒經常拿出來看一看，像中考、高考、過四六級背單詞一樣。

當然，語法問題雖然重要，更重要的是，我們可以通過討論“命名問題”來對領域建模。所謂的領域建模，某種程度上就是努力尋找那些“好名字”的過程。DDD所說的“統一語言”可以體現為一個詞彙表。

下面介紹我們想做的這個App都有什麼功能。

- 我們可以在App中創建多個“上下文”。什麼是“上下文”？也許你可以去了解一下DDD中“界限上下文”的概念。

- 我們可以在上下文中記錄“筆記”。我們還想要開發一些聊天機器人程序，把這些聊天機器人添加到工作微信群、QQ群裡面，機器人自動從群聊中抓取聊天消息作為筆記。

- 每個上下文中存在一個“詞彙表”。詞彙表中的每一個詞條的英文形式是它的ID，並且我們要求必須存在對應的中文翻譯。當我們編輯詞條時，App會根據當前上下文中存在的筆記智能地給出提示。

為了開發這個應用，我們需要使用如下技術：

- 使用“中文分詞”技術，將筆記中的中文句子分解為詞語，作為詞彙表的詞條的備選。
- 使用中英文翻譯服務。在我們錄入詞條的英文形式後，App給出可能的中文翻譯供用戶選擇，反過來也可以。
- 使用代碼搜索服務。當我們選中某個詞條，或者在某些地方選中某個詞語時，App會顯示這個詞語在實際的軟件源代碼中是怎麼被使用的。像 [SearchCode.com](https://SearchCode.com) 網站就可以提供代碼搜索API<sup>[1]</sup>。

我們在使用這個App尋找好名字的過程中——特別是通過查看使用了這些名字的代碼，會自然而然地學習到更多的領域知識和其他人的建模經驗，加深對領域的認知。

我們還希望有了這個詞彙表App之後，它可以與我們製作的DDML工具結合起來使用。比如，與圖形化的DDML設計器集成。當設計器中需要輸入聚合名稱的時候，設計器可以在詞彙表的數據庫中查找那些名詞性的詞條，為用戶提供自動完成功能。我們也希望有工具可以檢查DDML文檔描述的領域模型的關鍵概念——那些關鍵的對象、服務、方法、狀態的名字——是否都進入了詞彙表並且得到了準確的定義。

這樣，當我們使用**DDDML**工具生成大量的實現代碼時，這些代碼中的名字都會和詞彙表保持一致。開發人員在以後擴展或修改這些代碼的時候，也會自然而然地受到影響，沿用正確的命名方法。通過這些做法，我們可以有效地保證代碼和名字背後的領域模型之間存在映射關係。

如果**DDDML**文檔使用的各種名字都是正確的，那麼我們使用**DDDML**工具生成的用戶界面上的文本元素在默認情況下就具備良好的可讀性。比如，假設產品（*Product*）實體有個名為 **SupportDiscontinuationDate**（支持終止日期）的屬性，那麼我們的**DDDML**工具為這個屬性生成的標籤（*Label*）文本可能是“**Support Discontinuation Date**”，這樣的文本是可以直接在UI上使用的。

總之，命名問題真的很重要，為了找到那些好名字並且用好它們，值得我們多付出一些努力。

[1] 參見<https://searchcode.com/api/>。

# 附錄 DDDML示例與縮寫表

## DDDML示例：Package

---

```
aggregates:
# -----
Package:
    immutable: true
    implements: [Article]

id:
    name: PackageId
    type: long

properties:
    RowVersion:
        type: long

    PackageType:
        type: PackageType
    PackageParts:
        itemType: PackagePart

RootPackageParts:
    itemType: PackagePart
    isDerived: true
    filter:
        CSharp: "e => e.ParentPackagePartId == 0"
        Java: "e -> e.getParentPackagePartId() == null || e.getParentPackagePartId().equals(0L)"

reservedProperties:
    version: RowVersion

# -----
entities:

# -----
PackagePart:
    implements: [Article]
```

```
        id:
            name: PartId
            type: long
        globalId:
            name: PackagePartId

        properties:
            RowVersion:
                type: long
            PackagePartType:
                type: PackagePartType
            ParentPackagePartId:
                referenceType: PackagePart
                referenceName: ParentPackagePart
            ChildPackageParts:
                itemType: PackagePart
                inverseOf: ParentPackagePart

        reservedProperties:
            version: RowVersion

# -----
enumObjects:
    PackageType:
        baseType: int
        values:
            Piece:
                intValue: 1
            Box:
                intValue: 2
            BigBox:
                intValue: 3

    PackagePartType:
        baseType: int
        values:
            Piece:
                intValue: 1
            Box:
                intValue: 2
            Lot:
                intValue: 4

# -----
superObjects:
```

```
Article:
  properties:
    SerialNumber:
      type: string
    MaterialNumber:
      type: string
    CustomerNumber:
      type: string
    WorkOrderNumber:
      type: string
    LotNumber:
      type: string
    Rank:
      type: string
    Version:
      type: string
    Quantity:
      type: int
      notNull: true
    IsMixed:
      type: bool
```

---

## DDML示例 : Party

---

```
aggregates:
  Party:
    tableName: PARTY
    # -----
    discriminator: PartyTypeId
    discriminatorValue: "*"
    inheritanceMappingStrategy: tpcc
    polymorphic: true
    # -----
    id:
      name: PartyId
      columnName: PARTY_ID
      type: id
    properties:
      PartyTypeId:
        columnName: PARTY_TYPE_ID
        type: id
```

```
ExternalId:
  columnName: EXTERNAL_ID
  type: id
PreferredCurrencyUomId:
  columnName: PREFERRED_CURRENCY_UOM_ID
  type: id
Description:
  columnName: DESCRIPTION
  type: very-long
StatusId:
  columnName: STATUS_ID
  type: id
CreatedDate:
  columnName: CREATED_DATE
  type: date-time
CreatedByUserLogin:
  columnName: CREATED_BY_USER_LOGIN
  type: id-vlong
LastModifiedDate:
  columnName: LAST_MODIFIED_DATE
  type: date-time
LastModifiedByUserLogin:
  columnName: LAST_MODIFIED_BY_USER_LOGIN
  type: id-vlong
DataSourceId:
  columnName: DATA_SOURCE_ID
  type: id
IsUnread:
  columnName: IS_UNREAD
  type: indicator
reservedProperties:
  active: Active
  createdBy: CreatedByUserLogin
  createdAt: CreatedDate
  updatedBy: LastModifiedByUserLogin
  updatedAt: LastModifiedDate
  deleted: Deleted
  version: Version

subtypes:

LegalOrganization:
  discriminatorValue: "LegalOrganization"
  properties:
    TaxIdNum:
```

```

        type: string

InformalOrganization:
    discriminatorValue: "InformalOrganization"
    abstract: true

    subtypes:

        Family:
            discriminatorValue: "Family"
            properties:
                Surname:
                    type: string
            methods:
                ChangeSurname:
                    parameters:
                        NewName:
                            type: string

# -----
metadata: {}

```

---

## DDML示例 : Person

---

```

# -----
Person:
    id:
        name: PersonId
        type: PersonId
    properties:
        BirthDate:
            type: DateTime
            description: 出生日期
        Titles:
            itemType: string
        Email:
            type: Email
    YearPlans:
        itemType: YearPlan

    methods:
        # -----

```

```

ChangeEmail:
  parameters:
    NewEmail:
      type: Email
      # 下面這些都不是必需的
      # PersonId:
      #   type: PersonId
      #   isAggregateId: true
      # PersonVersion:
      #   type: long
      #   isAggregateVersion: true
      # CommandId:
      #   type: string
      #   isCommandId: true
      # RequesterId:
      #   isRequesterId: true

  entities:
    # -----
    YearPlan:
      id:
        name: Year
        type: int
      outerId:
        name: PersonId
      globalId:
        name: YearPlanId
      properties:
        Description:
          type: string
          length: 500
      MonthPlans:
        itemType: MonthPlan

    entities:
      # -----
      MonthPlan:
        id:
          name: Month
          type: int
        # outerIds:
          # PersonId:
          #   referenceType: Person
        # Year:
          # referenceType: YearPlan

```

```

        globalId:
            name: MonthPlanId
        properties:
            Description:
                type: string
                length: 500
        DayPlans:
            itemType: DayPlan

        entities:
# -----
        DayPlan:
            id:
                name: Day
                type: int
        outerIds:
            PersonId:
                referenceType: Person
            Year:
                referenceType: YearPlan
            Month:
                referenceType: MonthPlan
        globalId:
            name: DayPlanId
        properties:
            Description:
                type: string
                length: 500

# -----
valueObjects:

# -----
PersonalName:
    properties:
        FirstName:
            type: string
            description: First Name
            length: 50
        LastName:
            type: string
            description: Last Name
            length: 50

# -----

```

```

PersonId:
  properties:
    PersonalName:
      type: PersonalName
    SequenceId:
      type: int

```

---

## 縮寫表

英文縮寫	英文	中文譯文及解釋
DDD	Domain-Driven Design	領域驅動設計
DOM	Document Object Model	文档对象模型
DSL	Domain-Specific Language	領域專用語言
JSON	JavaScript Object Notation	JavaScript 對象标记
OOA	Object-Oriented Analysis	面向對象分析方法
OOD	Object-Oriented Design	面向對象設計
UML	Unified Modeling Language	統一建模語言或標準建模語言
XML	Extensible Markup Language	可擴展標記語言
YAML	YAML Ain't Markup Language	其名直譯過來是“YAML 不是標記語言”。其實 YAML 使用的是一種遞歸且戲謔的命名方法，它就是一種標記語言
BDD	Behavior-Driven Development	行為驅動開發
ES	Event Sourcing	事件溯源。指的是這樣一種程序設計模式：它保證所有對應用系統的狀態的變更都被存儲到一個事件的序列中