國立政治大學資訊科學系
Department of Computer Science
National Chengchi University

碩士論文
Master's Thesis

基於.NET 平台之可調性多租戶軟體框架

An Adaptable Multi-Tenant Application Framework Based on .NET Platform

研 究 生：莊偉瓏
指導教授：陳　恭

中華民國一0五年七月
July 2016

基於.NET 平台之可調性多租戶軟體框架

An Adaptable Multi-Tenant Application Framework Based on .NET Platform

研 究 生：莊偉瓏　　Student：Wei-Lung Chuang

指導教授：陳　恭　　Advisor：Kung Chen

國立政治大學

資訊科學系

碩士論文

A Thesis

submitted to Department of Computer Science

National Chengchi University

in partial fulfillment of the Requirements

for the degree of

Master

in

Computer Science

中華民國一０五年七月

July 2016

誌　謝

能夠在職場多年後重新回到學校，取得碩士學位完成一個重要的里程，我最感謝我的父母、太太還有兩個女兒的支持與鼓勵，為我分擔許多家庭的責任，讓我無顧慮的在學業及研究上。

我非常感謝陳恭老師，引導我更廣闊的思維，不論在課業上或者研究上，學習了許多在職場多年未曾涉獵的技術與知識，經過論文的洗禮，讓我資訊方面的思維提升到另一個層次，與陳恭老師的互動讓我深刻了解，為何與指導老師的關係並非師生關係，而是師徒關係。另外我要特別感謝上一屆的乃嘉學姊與宗佐學長，時常與我討論在論文使用的技術，感謝這兩位對於技術很有熱誠的學長姐。

此外，我要還感謝同學文聰、舒婷、儀婷、展民、展嘉、世峰大家在課業上的互相扶持，讓我可以順利走完全程。

基於.NET 平台之可調性多租戶軟體框架

摘要

近年來雲端運算蓬勃發展，為資訊系統的建置與服務帶來巨大的改變，其中一個重要趨勢為軟體即服務，並透過多租戶共享資源達到降低成本的優勢。因此如何讓服務可以共享資源，又能兼顧各租戶的客製化需求，這將會是軟體即服務關鍵成功因素。

為了讓租戶客製化自己的綱要，在多租戶的相關研究中，發展出各種綱要映射技術，各適用不同的狀況，但在開發應用程式時候往往無法預估租戶適合使用何種綱要映射技術。本研究提出可以讓多租戶軟體框架具有綱要映射技術的可調性，應用程式開發時候毋需考慮使用何種綱要映射技術，等租用時候再依據應用程式及租戶的特性決定。本研究將以小量租戶效能最佳的 Private Table Layout 與適合用於大量租戶的 Universal Table Layout 為例，提出的可調性多租戶實體模式來建構具有可調性的多租戶軟體框架。此外本研究提出的軟體框架經過技術的封裝，開發者不需要了解多租戶的相關技術，就能完成多租戶應用程式的開發。

關鍵字:軟體即服務、多租戶、綱要映射技術

An Adaptable Multi-Tenant Application Framework Based on .NET Platform

Abstract

Software as a service (SaaS) is an emerging service model of cloud computing. Its central defining characteristic is the ability for clients to use a software application on a pay-as-you-go subscription basis. However, to be economically sustainable, a SaaS application must leverage resource sharing to a large degree by accommodating different clients of the application while making it appear to each that they have the application all to themselves. In other words, a SaaS application must be a multi-tenant application.

An important multi-tenant research topic is the various kinds of schema mapping technology have been developed in order for our tenants to customize their schema. However, it is hard to determine tenants' need for particular schema mapping technology in different circumstances. This thesis proposes an adaptable schema mapping technology for a multi-tenant application (MTA) framework. The application and tenants' characteristics do not need to be considered while applications developers are developing their schema mapping technology. This approach will take examples from the Private Table Layout mapping and the Universal Table Layout mapping to illustrate the features of this adaptable multi-tenant software framework. Furthermore, this thesis argues that, with the approach packaged as a software framework, developers are able

to complete the development of a multi-tenant application without full understanding of the underlying technologies.

Keywords: SaaS、Multi-Tenant、Schema Mapping Technology

目錄

表目錄

圖目錄

程式碼目錄

第一章　緒論

隨著近年來雲端技術的興起，軟體的發展搭乘著這波浪潮產生了變化，許多軟體開發商投入資源提供 SaaS 服務模式推向市場，充分顯示運用網路技術與資源共享已成為資訊服務產業的趨勢。多租戶應用程式(Multi-Tenant Application，MTA)的特色即在於租戶共享資源，以降低成本，並且讓租戶有客製化的能力，不需要為每個租戶的客製化需求改寫程式，最後導致版本過多，難以維護的問題。另外各種綱要映射技術例如 Private Table Layout 與 Universal Table Layout 都有其適用的環境。有鑒於此，本研究提出可調性多租戶軟體框架的建構方式。

1.1　研究背景與動機

我任職於中央研究院，先後於資訊科學研究所與語言學研究所從事資訊系統開發，期間與其它所也有一些接觸，了解到中央研究院內各所對系統的需求都有些差異。

中央研究院內 31 個所皆屬於研究單位，院本部對各所的行政與學術管理方面有統一的管理規範，因此各所對於系統的需求也大致上會是類似，但由於各所規模與研究領域皆有所不同，所以統一管理規範之下還是保留一定程度的自治運作。因此運作上會有著或多或少的差異。院本部的資訊服務處(即一般認知的電算中心)提供給各所行政與學術管理上使用的系統，但由於各所之間的差異，所以資訊服務處所提供的系統通常難以達到完全符合各所的需求。因此各所可能就要使用其它方式，來補足系統上的缺角，但補足缺角的方式將可能會造成一些相關業務運作上的窒礙，並且延伸了系統開發的工作，而這種各單位之間有些許差異的問題正是多租戶的研究議題。

目前多租戶的研究在綱要映射技術部分多數集中在 Universal Table Layout，因為

Universal Table Layout 非常適合在大量租戶共用資源下的運作，它相對於 Private Table Layout 在越大量的租戶且資料表的差異越大的情況之下會有顯著的效能差異，但依前述的中央研究院案例其租戶數並非大量的情況，其最佳的資料綱要選擇會是 Private Table Layout，實務上也有許多的狀況相對不適合 Universal Table Layout 的需求，例如占用較大系統計算與 I/O 的 VIP 租戶，就不適合讓太多這類的租戶共用同一組資源，因此一個多租戶軟體框架應該要可以依據租戶的特性，來選擇適合的綱要映射技術。依所述原因引發投入研究可調性多租戶軟體框架相關技術的動機。

1.2 研究目的

本研究基於前述的研究背景與動機，探討如何實現可調性多租戶軟體框架，這框架必須可應用在本研究提到的中央研究院案例中這種小量租戶共用資源的需求，也可應用在大量租戶共用資源的需求，讓使用可調性多租戶軟體框架所開發的應用程式，可以在不需要修改的狀況下，因應不同情況的租戶選擇不同的資料綱要。使用可調性多租軟體框架開發應用程式的軟體開發過程中不僅不需要考量可能要應用在不同資料綱要之外，開發團隊不需要了解太多的多租戶技術下，不學習與改變太多原本的開發技能下完成應用程式開發。

1.3 研究貢獻

本研究提出可調性多租戶軟體框架概念與實作，使多租戶應用軟體可依需求應用不同的綱要映射技術，可依不同的需要，調整不同的綱要技術。另外提出之可調式多租戶實體設計模式，只要程式語言有 ORM 與動態技術就能依照此設計模式來實現。

第二章　相關研究與技術背景

2.1　多租戶資料層級

多租戶的各種設計模型的經濟效益評估指標就是共用硬體、資料庫與應用程式，並搭配適當的網要映射技術達到有效率的資料存取，便能達到節省成本的目的，這樣的模式讓資料庫能服務最大限度的租戶數，Aulbach et al.【2】 發現，許多多租戶難以解決的問題在資料層。

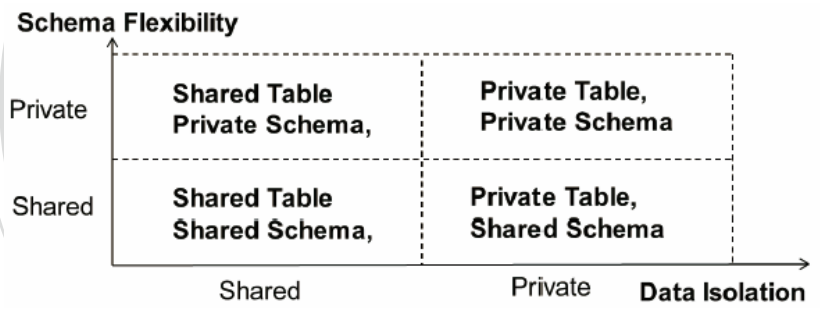


圖 2.1 多租戶資料層級技術分類，本圖取自【4】

2.1.1　多租戶資料架構分類

Shared Table, Shared Schema：又可稱為 Basic Layout【2】，Shared Table 代表示所有租戶們的資料都保存在同一個儲存空間，例如：同一個關聯式資料庫的一組資料表格，而 Shared Schema 則表示所有租戶們的資料綱要都是一樣的，這種方式在實作上最容易，只要將資料表綱要加入識別代碼的欄位，就能有效的將租戶們的資料隔離，這能有效又容易的讓許多租戶共用資源。但是這種方式的缺點是沒有彈性，因為所有租戶的資料綱要都必須相同，所以無法提供租戶變更資料綱要的客製化。此外，Aulbach et al.【2】也發現個別租戶回復資料，在這種框架會非常困難。

Private Table, Shared Schema：因為存放在不同的儲存空間，所以資料區隔性較好，因此有較佳的安全性，但因為還是 Shared Schema，所以還是有缺乏彈性的缺點，由於儲存空間分開在維護就變得相對困難，難以讓應用程式達到提供給大量租戶使用。

Private Table, Private Schema： Private Table Layout 即屬於為此類 這同時具有高安全性及可提供租戶們客製化優點，但由於 Private Table 所以同樣不適合提供給大量租戶使用，但在租戶數量不多的情況，會有最佳的效能。

Shared Table, Private Schema：這屬於最多研究在討論的架構，雖然相較 Private Table 安全性較低，需要透過其它機制來補強，但由於共用資源又具有彈性，所以是最被廣泛討論的架構。

2.2 綱要映射技術

2.2.1 Basic Layout

所有租戶的邏輯資料表都對映同一個實體資料表，租戶的資料僅用租戶編號欄位分辨，實作技術最簡單，但完全沒有擴充性。

2.2.2 Private Table Layout

每個租戶擁有自己的資料表，實作的方式是將資料表加上代號，或者每個租戶的資料表存放在不同資料庫，應用程式使用資料表時只要變換資料表名稱，或者切換連結的資料庫就能達成，其優點是實作技術上相對其它綱要映射技術簡單，且在租戶數量不多時有很好的效能，但相對的缺點為租戶量大時由於每個資料表擁有自己的資料綱要，因此大量租戶時 Buffer 有效利用率不好，導致效能會隨租戶數量的增價而逐漸下降【2】，並且各租戶的儲存空間不同，所以資源不分享，無法有效降低成本。

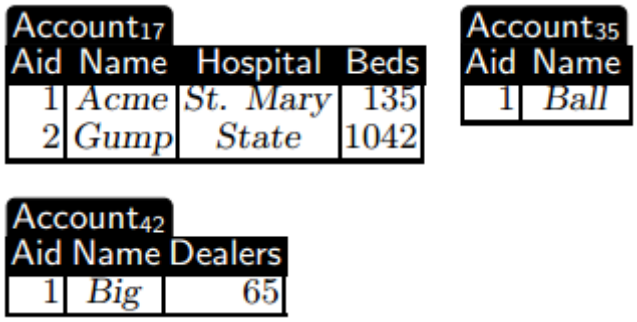


圖 2.2 Private Table Layout，本圖取自【2】

2.2.3 Extension Table Layout

共用欄位在同一個資料表格，並使用租戶編號的欄位區別是哪個租戶的資料，個別租戶客製化的欄位，則寫入各自的資料表格中。

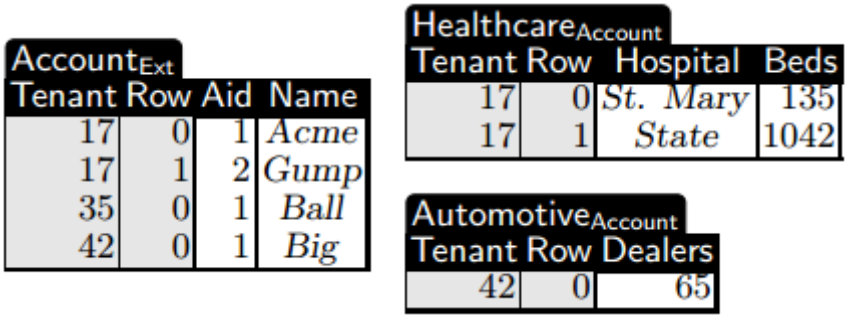


圖 2.3 Extension Table Layout，本圖取自【2】

2.2.4 Universal Table Layout

這是最多論文研究的綱要映射技術，因為在大量租戶時達到很好的資源共用與效能，Universal Table Layout 是將租戶的資料全部放置在一個包含大量租戶的大資料表，這個資料表必須包含租戶編號欄位、邏輯資料表編號與非常多的字串型態欄位，用於儲存各個租戶邏輯資料表內欄位的資料，不論任何型態都使用字串型態儲存，所以實作時可能需要做型態轉換，且邏輯資料表的欄位如果不多會造成稀疏矩陣的狀況。

Universal

Tenant	Table	Col1	Col2	Col3	Col4	Col5	Col6
17	0	1	Acme	St. Mary	135	—	—
17	0	2	Gump	State	1042	—	—
35	1	1	Ball	—	—	—	—
42	2	1	Big	65	—	—	—

圖 2.4 Universal Table Layout，本圖取自【2】

2.2.5 Pivot Table Layout

將不同的欄位型態放置在不同的資料表內，實作時取用資料須將用到的欄位 join 起來。

$Pivot_{int}$

Tenant	Table	Col	Row	Int
17	0	0	0	1
17	0	3	0	135
17	0	0	1	2
17	0	3	1	1042
35	1	0	0	1
42	2	0	0	1
42	2	2	0	65

$Pivot_{str}$

Tenant	Table	Col	Row	Str
17	0	1	0	Acme
17	0	2	0	St. Mary
17	0	1	1	Gump
17	0	2	1	State
35	1	1	0	Ball
42	2	1	0	Big

圖 2.5 Pivot Table Layout，本圖取自【2】

2.2.6 Chunk Table Layout

這種設計方式是將 Universal Table Layout 與 Pivot Table Layout 的概念結合在一起。

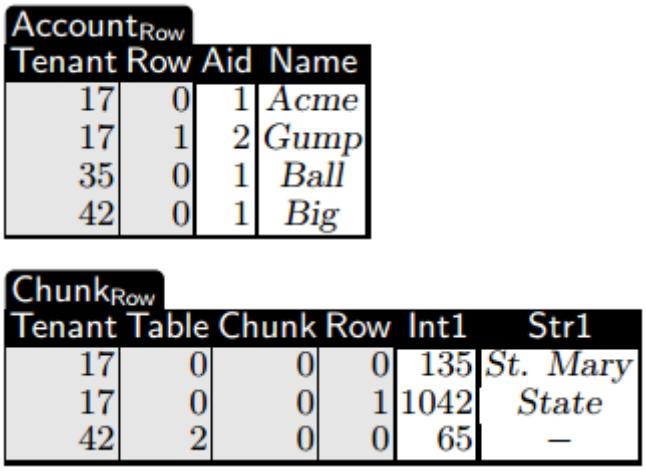
$Chunk_{int|str}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	1	Acme
17	0	1	0	135	St. Mary
17	0	0	1	2	Gump
17	0	1	1	1042	State
35	1	0	0	1	Ball
42	2	0	0	1	Big
42	2	1	0	65	—

圖 2.6 Chunk Table Layout ，本圖取自【2】

2.2.7 Chunk Folding Layout

這種設計方式是將 Extension Table Layout 與 Chunk Table Layout 的概念結合在一起，租戶欄位共用的方式採用 Extension Table Layout，每個租戶的客製化欄位則採用 Chunk Table Layout。



Account$_{Row}$

Tenant	Row	Aid	Name
17	0	1	*Acme*
17	1	2	*Gump*
35	0	1	*Ball*
42	0	1	*Big*

Chunk$_{Row}$

Tenant	Table	Chunk	Row	Int1	Str1
17	0	0	0	135	*St. Mary*
17	0	0	1	1042	*State*
42	2	0	0	65	–

圖 2.7 Chunk Folding Layout，本圖取自【2】

這七種綱要映射技術各有優缺點，一般來說 SaaS 服務提供者應可根據應用程式的特性選用適合的綱要映射技術，但往往選用表格映射技術的考量點並非在應用程式，而是在租戶的特性，例如高使用量的 VIP 客戶就不適合太多的租戶共用資源，因此可能適合 Private Table Layout，而一般的租戶可能最適合 Universal Table Layout，至於完全不變動基本規格，無須客製化的租戶最適合的是 Basic Layout，因此本研究除了實作出多租戶軟體框架之外，還賦予框架具有可調性，讓框架可以針對不同租戶選用不同的綱要映射技術，本研究將以 Private Table 與 Universal Table 為例進行說明。

2.3 Force.com Universal Table 資料架構

Universal Table Layout 是非常適合用於大量租戶共享資源以降低成本的技術，因此被許多論文提出來改良，本研究實作可調性多租戶軟體框架也會使用到 Universal Table Layout 來實現，本研究實作 Universal Table Layout 將參考 Force.com 提出的進行建置，

並在下一章提出改良，本節將介紹 Force.com Universal Table。

Force.com Universal Table 將資料表分為三種類型，分別為 Metadata Tables、Data Tables 以及 Specialized Pivot Tables，利用這三種類型建立出虛擬資料庫 【3】 ，下圖為 Force.com 的設計模型。

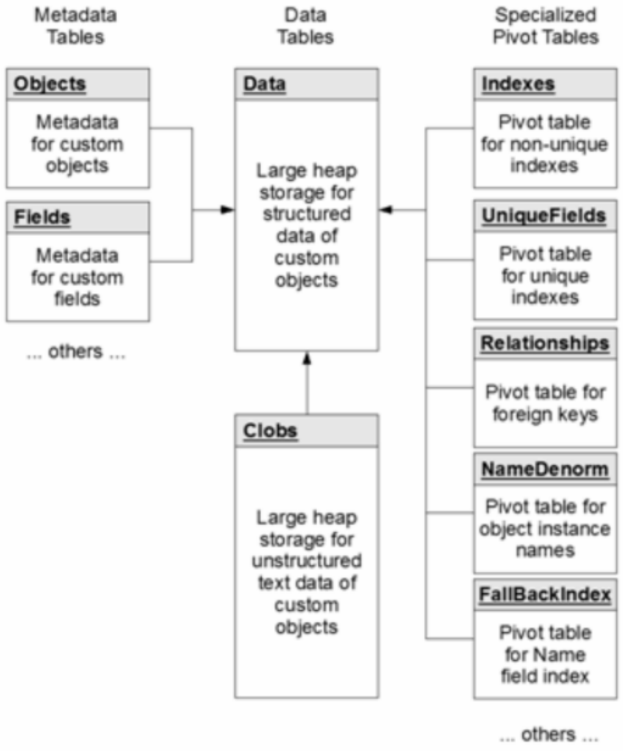


圖 2.8 Force.com Universal Table 資料架構，本圖取自【3】

Metadata 類型資料表：用來描述邏輯資料表的 Schema，其中的 Objects 是描述租戶有哪些邏輯資料表，Fields 則是紀錄邏輯資料表的欄位資訊，藉由 Objects 與 Fields 提供的資訊可以讓邏輯資料表與實體資料表對映。

Data 類型資料表：實際用來儲存租戶的資料，其中的 Data 提供許多字串型態欄位，用以儲存租戶的資料，所有資料都是使用字串形態儲存，使用時則配合 Fields 記載的資訊獲得 Data 中每個欄位的意義及實際資料型態。而 Clobs 則保存了租戶非結構化文字資料。

Specialized Pivot 資料表：主要在維護 Data 中的非正規化資料，並利用當中的資訊加

速取得租戶在 Data 中的資料，雖然 Force.com 並未公布所有 Specialized Pivot 資料表，但主要有 Indexs、UniqueFields、Relationships、NameDenorm 以及 FallBackIndex。由於 Data 中保存的租戶資料都是字串，因此不可能使用關聯式資料庫中的 Index 功能來建置索引以加速查詢速度，所以把 Data 中儲存資料的欄位複製到 Indexs 後再建立關聯式資料庫的索引。UniqueFields 的功能如同 Indexs，但 Indexs 中的索引值是可以重複的，不過 UniqueFields 是不可以重複的，Relactionship 則是記錄了 Objects 所描述的邏輯資料表之間關聯的索引。NameDenorm 則是提供快速查詢 Object 名稱與 Id 的索引。此外當 Force.com 搜索引擎出問題或負載過高時，Force.com 的查詢機制便會使用 FallBackIndex 中的資料來完成查詢。

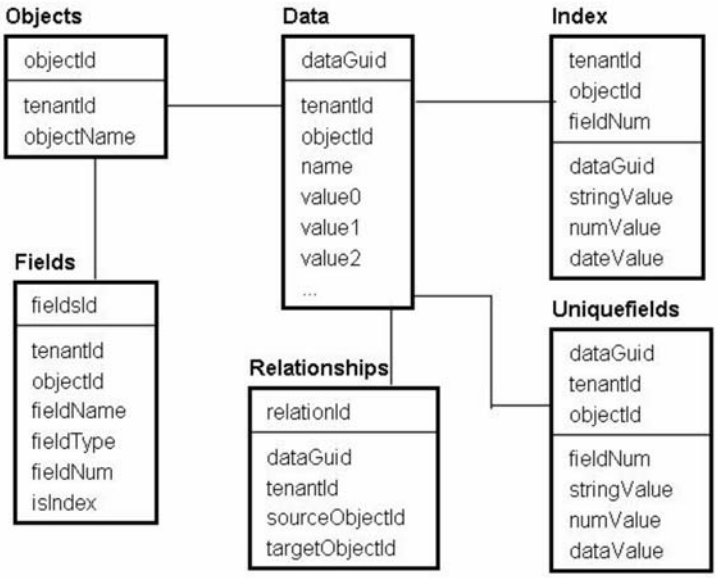


圖 2.9 Universal Table 資料綱要，本圖取自【4】

2.3.1 Objects、Fields、Data 資料表

Data 中保存了租戶的資料，Objects 與 Fields 則保存了詮釋 Data 中資料的 Metadata，Objects 中的每筆資料等同於關聯式資料庫中的一個資料表，他描述的是每個租戶的每個邏輯資料表，而 Fileds 中的資料則是描述租戶邏輯資料表中的欄位，【圖 2.2】 說明了資料表需要的欄位與關聯。

Data 資料表的欄位主要有 dataGuid 作為主鍵(Primary Key)，使用 GUID/UUID 作為其欄位的型態，tenantId 為租戶的 Id，obectId 為租戶所擁用的 object 的 Id，所以我們使用 tenantId 與 objectId 就能知道某筆資料屬於哪一個租戶，並且是屬於哪一個 object 的資料，而租戶邏輯資料表中的資料實際上存在以 value 開頭的欄位，全部都使用字串型態儲存，這種欄位會有多個，是為擴充所預留的，預留越多則可擴充欄位就越多，但也就越容易有稀疏矩陣的情況發生。Objects 的除了有 objectId 作為主鍵之外，還有 tenantId 表示屬於哪一個租戶，objectName 記載邏輯資料表的名稱。至於 Fields 資料表除了有 fieldId 作為主鍵之外，透過 tenantId 與 objectId 欄位可以知道該欄位屬於哪一個租戶的哪一個資料表，fieldName 與 fieldType 則分別表示其租戶的邏輯資料表中的欄位名稱與實際資料型態，fieldNum 則表示邏輯資料表中的欄位順序。

2.3.2 Indexes、UniqueFields 資料表

由於 Data 資料表中每個 Value 欄位中所保存的租戶資料都是字串型態，並不區分型別，因此無法使用關聯式資料庫的索引功能來提升查詢速度。對於 Universal Table Layout 這個問題，Force.com 使用 Indexes、與 UniqueFields 資料表來間接建立索引，以解決無法使用索引的問題。

Indexes 與 UniqueFields 的欄位都是一樣的，差異是 UniqueFields 中的索引值不能重複。Indexes 與 UniqueFields 使用 TenantId、ObjectId、FieldNum 以及 DataGuid 四個欄位作為組合主鍵(Composite Key)，除了這四個欄位之外，另外還有數個欄位名稱為型態加上 Value 的欄位，例如 StringValue、NumValue、DateValue 等等，多租戶軟體框

架將支援多少種欄位類型，就會有多少這種欄位。並且必須為每種類型的欄位設定 Index，例如 NumValue 的 Index 須設定為 TenantId、ObjectId、FieldNum 與 NumValue，如此設定就可以採用間接的方式快速取得租戶的資料。

當 Data 資料表新增資料時，必須將有設定為 Index 或 Unique 的欄位，欄位內容重複寫入到 Indexes 或 UniqueFields 資料表，藉由以上的設計 Force.com 的 Query Optimizer 在接受查詢請求時，會先確認查詢條件中的欄位是否設定為 Unique 或 Index，如果有便會先透過 Indexes 或 UniqueFields 資料表使用索引間接取得 Data 資料表中的資料；由於 Indexes 或 UniqueFields 儲存的資料型態都有依據其型別儲存，並且有建立索引，因此藉由關連式資料庫的索引功能，便可快速取得 Indexes 或 UniqueFields 中的資料，再透過 DataGuid 到 Data 資料表取得完整資料，如此設計就可以彌補 Universal Table Layout 無法支援 Index 的問題。

2.4 LINQ

LINQ (Language-Integrated Query)是由.NET Framework 3.5 版中引進的創新技術，用來填補物件與資料之間的缺口，LINQ 的目的是希望以一種語言，查詢多種不同對象，可以對有實作 IEnumerable 或 IEnumerable<T>介面的物件進行查詢，例如：Array、ArrayList、IList、IList<T>、 Collection<T>、IQueryable<T>、DbSet<T>等等。

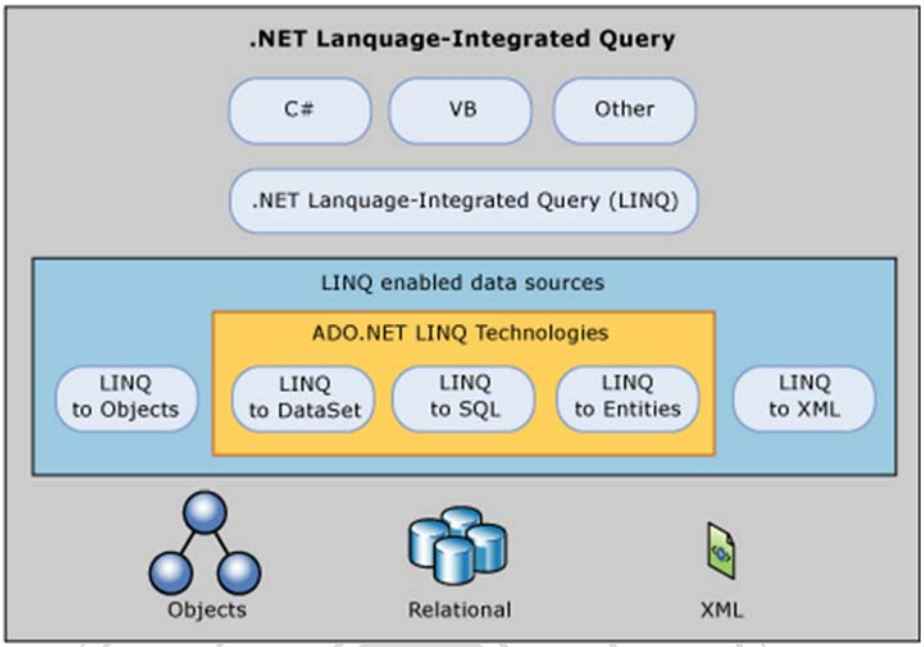


圖 2.10 LINQ 架構圖，本圖取自【6】

LINQ 支援兩種表達式，分別是 Query Expression 與 Fluent， Query Expression 使用類似 SQL 的語法取得資料，如下程式碼示範：

```
from a in Data
where a.TenantID == Guid.Parse("c7a11ee6-4adb-4678-b191-31ea484fdf5a")
&& a.ObjectID == Guid.Parse("7beb593a-309a-4a93-9891-af17700b42ff")
orderby a.Value1
select a
```

程式碼 2.1 LINQ Query Expression

而 Fluent 則採用方法串接的方式，並搭配 Lambda 取得資料，如下程式碼示範:

```
Data
.Where(p => p.TenantID == Guid.Parse("c7a11ee6-4adb-4678-b191-31ea484fdf5a")
        && p.ObjectID == Guid.Parse("7beb593a-309a-4a93-9891-af17700b42ff"))
.OrderBy(p => p.Value1)
```

程式碼 2.2 LINQ Fluent

LINQ 依據對象有分為，LINQ to Object、LINQ to SQL、LINQ to XML 與 LINQ to DataSet，其中 LINQ to SQL 是使用 LINQ 技術來存取資料庫，也就是將 LINQ 語法轉

為 SQL Command 查詢資料庫的機制，他著重的是 Entity Mapping Table，而後 LINQ to SQL 的發展路線改由 Entity Framework 取代，Entity Framework 與 LINQ 語法搭配使用可以實作出許多類型的綱要映射技術，實作方式是本研究的重點之一，下節將先簡介 Entity Framework。

2.5 Entity Framework

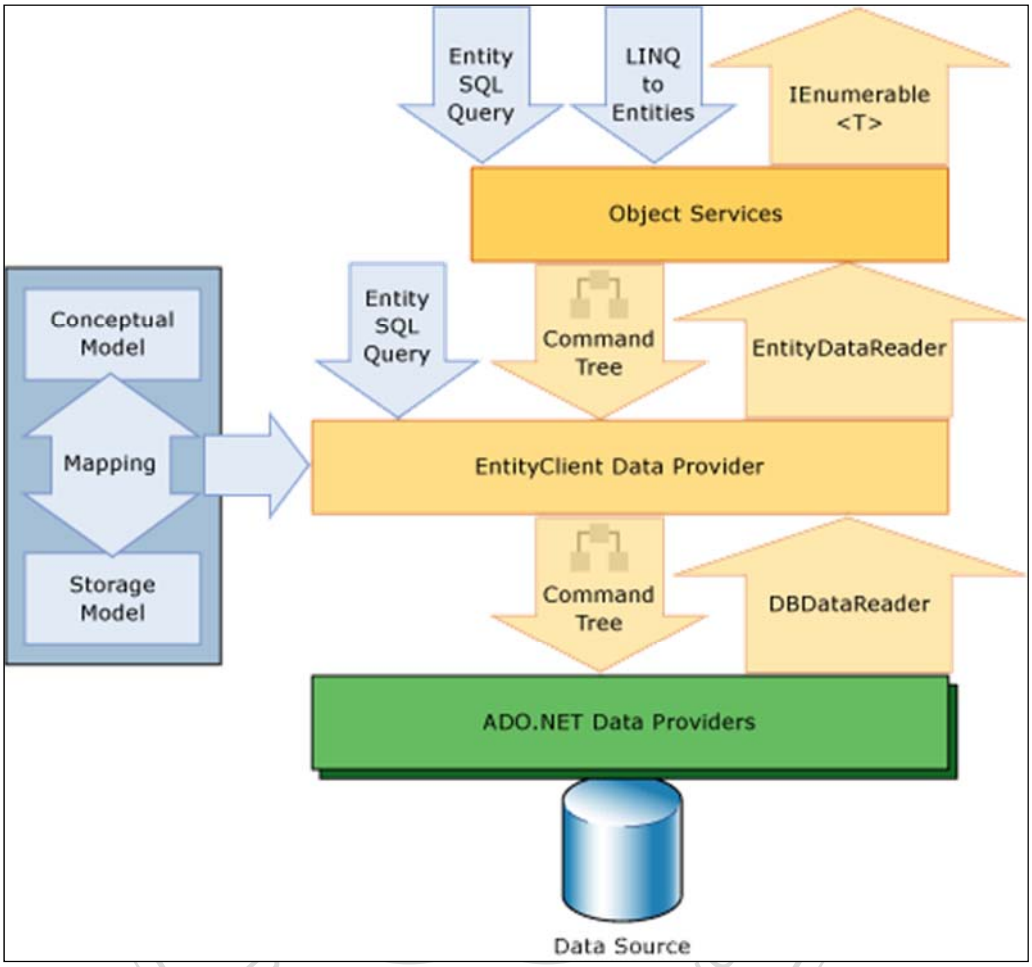


圖 2.11 Entity Framework 架構圖，本圖取自【7】

Entity Framework 是微軟於.NET Framework 3.5 Service Pack 1 推出的 ORM(Object Relational Mapping)框架，讓開發人員可以用特定領域物件(Domain-Specific Objects)和屬性 (如客戶和客戶地址)來處理資料，而不需要顧慮這些資料儲存在哪些基礎資料庫資料表和資料行內。開發人員在處理資料時可以在較高的抽象層級工作，而且能使用比傳統應用程式更少的程式碼來建立及維護資料導向應用程式【7】。

Entity Framework 可以讓開發者使用 LINQ 來查詢資料庫，而不需要自己編寫 SQL Command，以提供使用物件導向的方式存取資料庫，本研究撰寫時已經發展到 5.0 版本，各項功能已經相當完整，依使用方式分為三種，分別為 DB First、Model First 以及

Code First。

DB First 與 Model First 有提供 GUI 來設計 Entity，及 Entity 與 Table 之間的 Mapping，兩者差異在於 DB First 是先建立資料庫和 Tables 之後，透過 GUI 由 Tables 或 Views 自動產生 Entity，而 Model First 則是先使用 GUI 建立 Entity 之後，由 r 產生 Tables，目前 DB First 的開發模式較廣為使用。兩種模式的資訊都儲存在 EDMX 中，EDMX 是一個 XML 檔案，其中包含 CSDL(Conceptual Model)、SSDL(Storage Models) 與 MSL(Mappings)，CSDL 描述 Entity，SSDL 描述資料庫中的 Table，而 MSL 則描述著 CSDL 與 SSDL 之間如何 Mapping。

而 Code First 沒有提供 GUI，設計者必需要自行撰寫 POCO(Plain Old CLR Object)y 做為 Entity，並於 POCO 上設定 Attribute (Attribute 等同於 Java 上稱的 Annotation)，並且將在 DbContext 設定取用 Entity 的 Property，當程式執行時就會自動建立 Tables。

不論是 DB First、Model First 以及 Code First 都是透過 LINQ 操作 Entity，然後最終轉為 SQL Command 操作資料庫的 Tables。

2.6 Reflection

反映 (Reflection) 會提供 Type 型別的物件，用來描述組件、模組和型別。可以使用反映來動態建立型別的執行個體、將型別繫結至現有物件，或從現有物件取得型別，並叫用其方法或存取其欄位和屬性。

2.7 Emit

System.Reflection.Emit 這 namespaces 內包含的 Class 可以在 compiler 或工具去 emit metadata 和 Microsoft intermediate language (MSIL)【9】，更簡明的說就是可以在 Run-Time 產生 Class 及其 Method、Field 及 Property。

第三章　可調性軟體框架之設計

本研究提出之可調性多租戶軟體框架採用使用 ASP.NET MVC 實作，本章將依 Model、Controller 及 View 依序進行說明。其中本研究提出之可調性部分可以讓框架應用於不同的綱要映射技術，此部分建構在框架的 Model 部分，不論租戶採用何種綱要映射技術，都是使用同一個 Entity 工廠。因此對於 MTA Application Designer 來說，設計 Application 時不須預測租戶將要使用何種綱要映射技術。也就是說在本研究提出的可調性多租戶軟體框架下，Application 可應用在不同的綱要映射技術。

3.1　Model

多租戶軟體框架之設計就 Model 層來說，歸納須解決下列幾點問題:

(1) Entity 與實體資料表的對映。

(2) Entity 與實體資料表欄位型態不同的轉換。

(3) 封裝 MTA 技術。

(4) 執行時期產生及實作租戶之 Entity-Class。

(5) Entity 的變數型態問題。

3.1.1　Entity 與實體資料表對映

當我們開發多租戶的 Application 時候，為了使 Application 所使用的關聯式資料庫具有支援多租戶的能力，我們需要依照系統及租戶的特性選擇適合的綱要映射技術，但無論何種選用哪一種映射技術，租戶的邏輯資料表對映到我們選擇的綱要映射技術 SQL Command 都要做些改變，例如 Private Table Layout 需要變更表格或者資料庫名稱，Universal Table Layout 需要增加 TenantId 與 EntityId 的 Where 條件及各 Value 實體欄位

與邏輯欄位名稱的對映，Extension Table Layout 則需要 Join table。

在 OOP 中我們通常透過 ORM 技術來操作資料表，因此實務上我們要解決的邏輯資料表與實體資料表的對映，需要轉換為 Entity 與實體資料表的對映問題，而目前許多 ORM 框架的發展日漸完備，透過一些使用上的技巧是可以辦到 Entity 與實體資料表對映，在.NET 中 LINQ 與 Entity Framework 的搭配就是一個例子，以下將分別探討 Entity 與實體資料表對映在 Universal Table 與 Private Table 如何實做，但暫時不考慮因為租戶之間的 Entity 不同，而需要執行時期動態產生租戶 Entity 的技術，故本小節實作研究屬於接下來給動態產生 Entity 技術的實作藍圖。

3.1.1.1 Entity 對映 Universal Table Layout

我們以一個簡單的例子來說明，假設有一個多租戶的客戶管理系統，系統中有一個 Domain-Entity 叫做 Customer 用來記錄客戶資料，系統原本設計有兩個欄位分別為 Name 與 Phone，而 A 租戶新增了一個 Age 欄位，如此我們使用 Universal Table Layout 會做產生如下的對映。

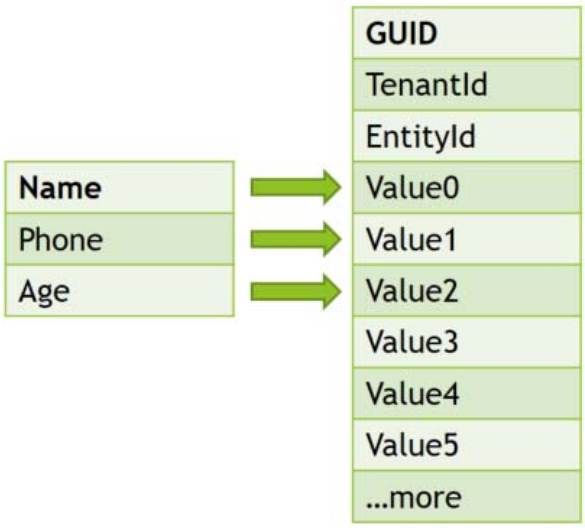


圖 3.1 Entity 與 Universal Table Layout 對映示意圖

左邊是 Entity，右邊是 Universal Table Layout 實體資料表，也就是 Force.com Universal Table 模型中的 Data 資料表。本研究使用可調性多租戶實體模式，來實做 Entity，每個 Entity 必定有的三個 Property，GUID、TenantId 及 EntityId 由 MtaEntity 提供。由 MTA Application 提供的為 Domain-Entity，它繼承 MtaEntity 並提供 Name 與 Phone 這兩個 Property，。而執行時期，真正提供給租戶的為 Tenant-Entity，他繼承 Domain-Entity，並提供客製化的 Property，例如本例 A 租戶新增的 Age。

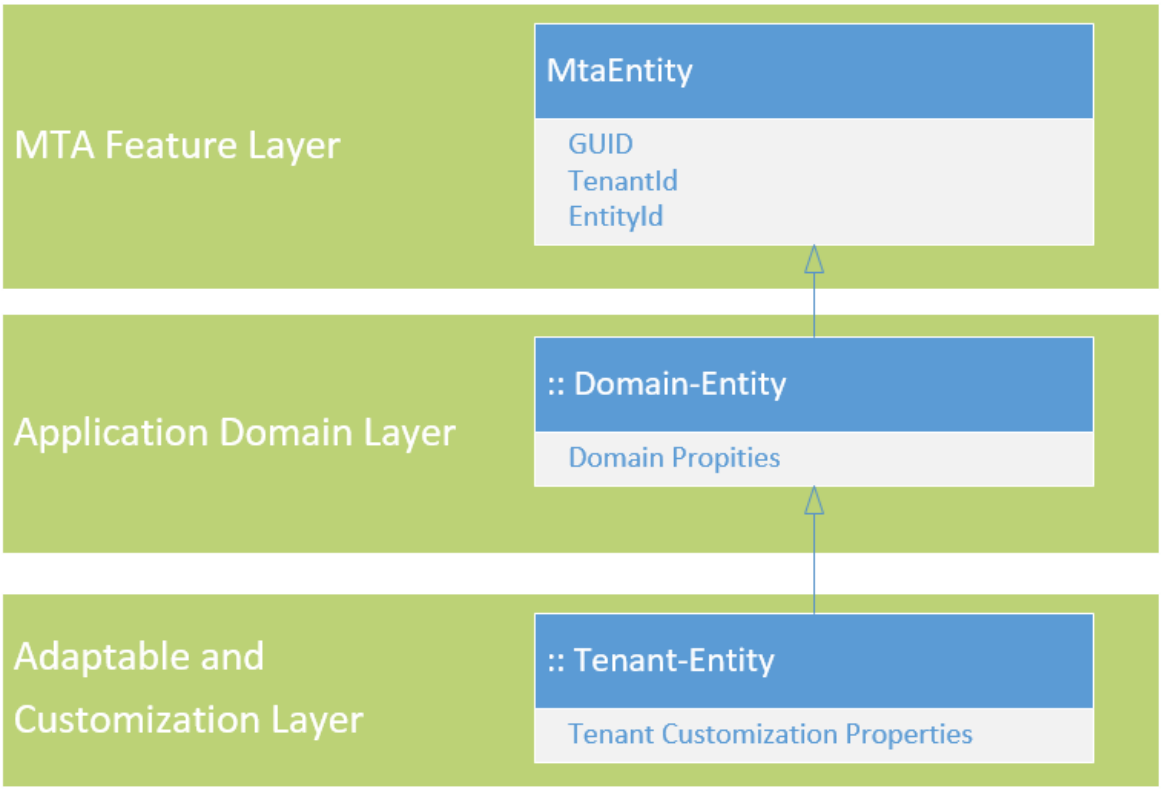


圖 3.2 可調性多租戶實體模式

如下圖範例資料，為了易於說明 TenantId 與 EntityId 使用 int 而不使用 Guid 型態，Value0、Value1 及 Value2 則在 TenantId 與 EntityId 的值皆為 1 的資料中分別對映了 Name、Phone 及 Age。

	GUID	TenantId	EntityId	Value0	Value1	Value2	Value3
	e62fc622...	1	1	張三	0900-00...	20	*NULL*
	1f266eb7...	1	1	李四	0900-00...	30	*NULL*
	6b513d8...	1	1	王五	0900-00...	15	*NULL*
▸*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*	*NULL*

圖 3.3 Universal Table Layout Data 資料表，資料範例

如果是非多租戶的 Application 中使用 ORM 框架，表示 Entity 可以用相同的名稱及 Property 名稱對映到實體資料表。但若在多租戶 Application 選擇了使用 Universal

Table Layout，那 Entity 對映資料庫之前就需要將 SQL Command 轉換表格名稱與欄位名稱，並加上 TenantId 與 EntityId 的 Where 條件才能正確取得租戶資料。

要實做這樣的轉換技術，本研究使用 Entity Framework 的 Code First 並配合一些技巧來達成。我們依照可調性多租戶實體模式類別圖使用 Entity Framework 來實做 Entity 對映 Universal Table Layout，先撰寫 MtaEntity-Class。

```
public class MtaEntity
{
    [Key]
    public Guid GUID { get; set; }

    public virtual int TenantId { get; set; }

    public virtual int EntityId { get; set; }
}
```

程式碼 3.1 MtaEntity POCO

MtaEntity 共有 GUID、TenantId 及 EntityId 三個 Property，並且將 GUID Property 設定為 Key，MtaEntity 屬於可調性多租戶軟體框架提供的 Class。MTA Application Designer 在開發系統時需撰寫 Domain-Entity 繼承 MtaEntity 並所有的 Property 都須加上 virtual 修飾詞，以提供 Tenant-Entity override。

```
//Domain-Entity
public class DomainCustomer : MtaEntity
{
    [MaxLength(20)]
    public virtual String Name { get; set; }
    [MaxLength(15)]
    public virtual String Phone { get; set; }
}
```

程式碼 3.2 Domain-Entity POCO 範例

```
//Tenant-Entity
[Table("Data")]
public class Customer : DomainCustomer
{
    [Column("Value0")]
    public override String Name { get; set; }

    [Column("Value1")]
    public override String Phone { get; set; }

    [Column("Value2")]
    public String Age { get; set; }
}
```

程式碼 3.3 Universal Table Tenant-Entity POCO 範例

要取得資料時，可以搭配 LINQ 加入 TenantId 與 EntityId 的條件如【程式碼 3.4】所示，取得租戶編號為 1、Entity 編號為 1 且 Phone 為 0900-000-001 的範例程式碼。執行時使用 SQL Server Profiler 觀察，其結果如【圖 3.4】，這段 LINQ 經由 Entity Framework 對 SQL Server 下達 SQL Command，將條件中的 Phone 轉為 Value1。

```
db.Customers
.Where(p => p.TenantId == 1 &&
            p.EntityId == 1 &&
            p.Phone == "0900-000-001")
.ToList());
```

程式碼 3.4 LINQ 查詢租戶資料

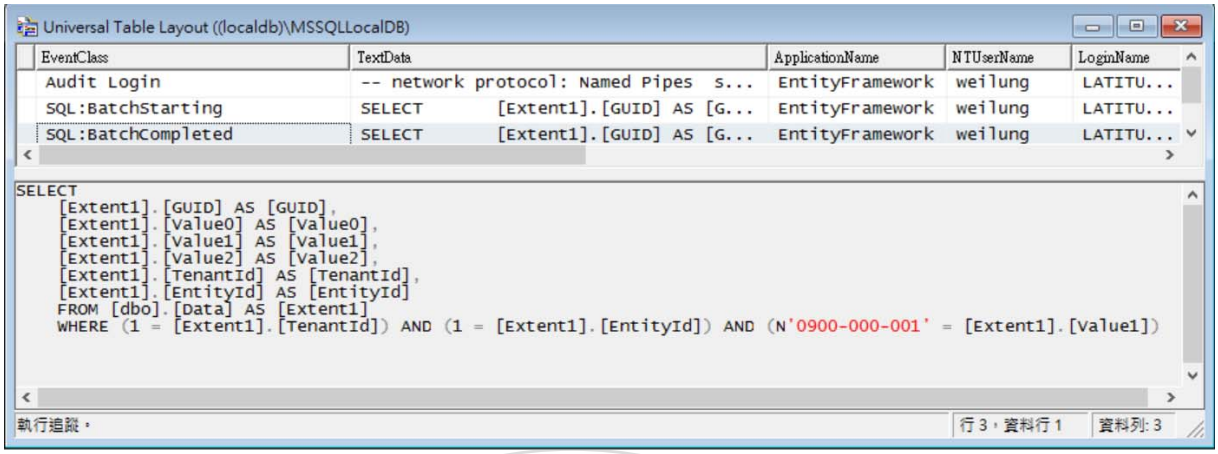


圖 3.4 SQL Server Profiler 觀察查詢資料 SQL Command

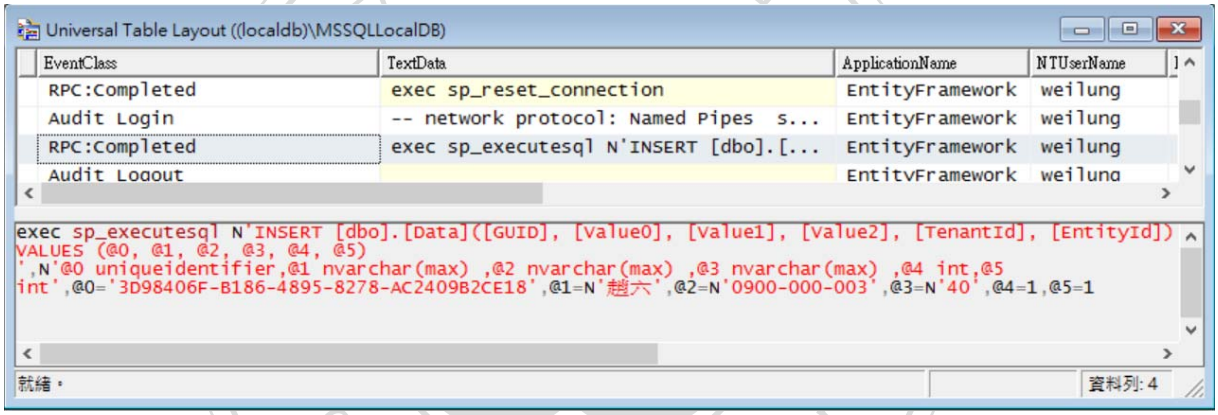


圖 3.5 SQL Server Profiler 觀察新增資料 SQL Command

但由於 Code First 有兩個問題存在，所以使用單純的 Code First 無法確實適用於多租戶的 Entity 對映到實體資料表，須加上一些技巧。

第一個問題，由於 Code First 的概念是先寫 Entity，執行時再自動建立資料庫與程式碼。而 Entity 變更後再使用 Code First 的 Migration 機制，依照 Entity 來變更資料表。由於使用 Universal Table Layout，Entity 的 Property 一定少於 Data 資料表的 Value 欄位，所以會有許多 Value 欄位沒被對映到，因此正規的使用 Code First 無法應付這種狀況，但 Code First 另有一個特性，就是如果 Table 不是經由 Code First 所建立，那 Entity Framework 只會進行對映工作，不會去檢查對映的欄位數量是否一致，因此只要事先建立好 Data 資料表就可以。

第二個問題，Entity Framework 中一個 Table 不允許被多個 Entity 所對映，因此若一個 Request 中只有一個 Entity 對映到 Data 資料表就不會有問題，但如果有兩個 Entity 的狀況，那就需要建立多個 Data 資料表的 View，再將每個 Entity 對映到不同 View，如此就能解決多重對映問題。

3.1.1.2 Entity 對映 Private Table Layout

Private Table Layout 與一般開發 Application 使用的資料表最為接近，要實作 Private Table Layout 在概念上可分為兩種方式。第一種為變更資料表名稱：所有租戶的資料表都放在同一個資料庫內，但資料表各自獨立，租戶的資料表區隔可在資料表名稱前或後加上租戶代號，以識別不同租戶擁有的資料表，取用時資料表需要依租戶變更資料表名稱。第二種為變更連線資料庫：擁有自己的資料庫，因此資料表名稱保留不變，取用時須變更連線到不同的資料庫。變更資料表名稱的方式在上一小段實作 Universal Table Layout 已經使用過，故本小段以變更連線方式為例，依例子可以很容易轉換為變更資料表名稱的做法。

Entity 對映 Private Table Layout 使用 Code First 在對映上非常容易做到，這在 Code First 第一次建立資料表時是沒問題的。本小段持續使用上一小段提出的可調性多租戶實體模式，MtaEntity 與 Domain-Entity 都維持不變，僅對執行時期可調性多租戶軟體框架產生的 Tenant-Entity 提出改變，以下列 Tenant-Entity 產生原則提供參考。

1. TenantId 與 EntityId 沒有對映的必要，可宣告為 NotMapped。
2. Domain-Entity 已經宣告的 Property 無須再 override。
3. Domain-Entity 已經宣告的 Property，如果要增加 Attribute 則需宣告。

相較於 Universal Table Layout 的 Tenant-Entity 產生原則，由於 Tenant-Entity 名稱與 Property 名稱不須轉換，Tenant-Entity 的實作上也簡便很多，依原則實做的程式碼如【程式碼 3.5】所示，其中 Name 因為變更為最大長度 50，所以需要提出來做 override，此段程式碼執行後，會自動建立資料庫與 Table，產生資料綱要如【圖 3.6】

所示。

```
//Tenant-Entity
public class Customer : DomainCustomer
{
    [NotMapped]
    public override int TenantId { get; set; }

    [NotMapped]
    public override int EntityId { get; set; }

    [MaxLength(50)]
    public override String Name { get; set; }

    public int? Age { get; set; }
}
```

程式碼 3.5 Private Table Tenant-Entity POCO 範例

	資料行名稱	資料類型	允許 Null
🔑	GUID	uniqueidentifier	☐
	Name	nvarchar(50)	☑
	Phone	nvarchar(15)	☑
	Age	int	☑

圖 3.6 Code-First 自動產生之資料綱要

但在 Code-First 中為了管理 Entity 與資料庫中 Table 的一致性，Entity 變更後都需要執行 Migrations，將 Entity 的變更修改到 Table。Migrations 區分為三種，分別為自動移轉、手動移轉及程式移轉，預設為自動移轉。自動移轉與手動移轉實務上較常使用，這兩種移轉方式需要在 Visual Studio 中的 Package Manager Console 執行 Migrations 指令。但因為最終我們要產生的 Tenant-Entity 是執行時期動態產生，因此不適合這兩種移轉方式。所以我們需要使用的是程式移轉，讓 Tenant-Entity 變更後可以直接變更資料表。若採用租戶有獨立資料庫的方式，還要解決如何讓每個租戶可以使用不同連線，因此本小段目標是實作程式移轉的 Entity Framework，並且使其加入可變動不同連線的機制，以作為本軟體框架執行時產生 Model 的設計藍圖。

因此我們為 Tenant-Entity 所在的 Model 加入程式移轉的設定與 Class，那 Tenant-Entity 變更就不需要在 Package Manager Console 執行 Migrations 指令，而會在應用程式執行時，由 Entity Framework 變更 Table Schema。我們做一個簡單的模擬實驗，租戶在 Customer 新增一個 Address，Tenant-Entity 模擬程式與執行後 Table Schema 如【程式碼 3.6】與【圖 3.7】所示。

```
//Tenant-Entity
public class Customer : DomainCustomer
{
    [NotMapped]
    public override int TenantId { get; set; }

    [NotMapped]
    public override int EntityId { get; set; }

    [MaxLength(50)]
    public override String Name { get; set; }

    public int? Age { get; set; }

    [MaxLength(150)]
    public String Address { get; set; }
}
```

程式碼 3.6 Private Table Tenant-Entity POCO 範例(修改後)

	資料行名稱	資料類型	允許 Null
🔑	GUID	uniqueidentifier	☐
	Name	nvarchar(50)	☑
	Phone	nvarchar(15)	☑
	Age	int	☑
	Address	nvarchar(150)	☑

圖 3.7 Code-First 自動修改之資料綱要

由以上的模擬實驗可以確認，這適用在 Private Table Layout 來管理租戶的實體資料表綱要。若需要變更連線，這部分只要預先設定可用連線在 Web.config 的 connectionStrings，並變更 DbContext 建構元的傳入參數就可以達到。

本小節提出可調性多租戶實體模式，MTA-Entity 是由可調性多租戶軟體框架提供。它負責提供 Entity 於可調式多租戶軟體框架運作時共同必要的 Property。Domain-Entity 繼承 MTA-Entity，它是 Development-Time 由 MTA 應用程式開發團隊設計，為應用程式提供服務必要的 Property。而 Tenant-Entity 繼承 Domain-Entity 由可調性多租戶軟體框架於 Run-Time 產生出來，除了依據租戶的客製化資訊提供租戶真正需要 Property 之外，還依租戶選用的綱要映射技術調整 Tenant-Entity 的 Attribute 以提供 Entity Framework 完成對映實體資料表。

由以上實作模擬程式的實驗可知，在可調性多租戶實體模式中，租戶選用不同的綱要映射技術，只會在執行時期所產生 Tenant-Entity 的 Attribute 上有所不同，只要依照可調性多租戶實體模式實做，就可以讓多租戶軟體框架具有可調性。但解決 Entity 與實體資料表對映之後還有其他問題，將在後面的小節一一提出並提出解決方案。

3.1.2 Entity 與實體資料表欄位型態不同的轉換

本小節主要是針對 Universal Table Layout 的 Property 型別轉換問題提出解決方案。上一小節中提出了 Universal Table Layout 與 Private Table Layout 實作對映的方案，其中 Universal Table Layout 中 Customer 的 Age 欄位為 String 型態，而 Private Table Layout 為 int 型態，會有這狀況是 Universal Table Layout 中存放資料的欄位 Value 型態皆為 nvarchar，而 ORM 資料對映型態必須符合。以這個例子，成績欄位應該是 int，但卻因為實體資料表的限制，導致 Property 必須跟著宣告為 String 型態。對此本研究提出的解決方式是在 MtaEntity 提供 protected 的字串轉型為指定型態的方法，如下程式碼。

```
public class MtaEntity
{
    [Key]
    public Guid GUID { get; set; }

    public virtual int TenantId { get; set; }

    public virtual int EntityId { get; set; }

    protected Int32? ValueToInt32(String value)
    {
        int retVal;
        if (Int32.TryParse(value, out retVal)) return retVal;
        return retVal;
    }

    protected Guid? ValueToGuid(String value)...

    protected DateTime? ValueToDateTime(String value)...
}
```

程式碼 3.7 支援 Universal Table Layout 欄位轉型之 MtaEntity

Domain-Entity 可以使用正確型態的 Property，Tenant-Entity 的 Property 型態非 String，則必須呼叫 MtaEntity 提供轉型方法。Age 變成三個部分，分別為：

1. String 型別的 Age Filed。
2. String 型別的 Str_Age Property：取用 Age Filed，並對映 Value 欄位。
3. Int32 型別的 Age Property，取用 Age Field，並使用 MtaEntity 中的轉型方法來轉型。

```
//Tenant-Entity
[Table("Data")]
public class Customer : DomainCustomer
{
    private String _Age;

    [Column("Value0")]
    public override String Name { get; set; }

    [Column("Value1")]
    public override String Phone { get; set; }

    [ScriptIgnore]
    [Column("Value2")]
    public String Str_Age { get; set; }

    [NotMapped]
    public int? Age
    {
        get
        {
            return ValueToInt32(_Age);
        }
        set
        {
            _Age = value.ToString();
        }
    }
}
```

程式碼 3.8 Tenant-Entity 屬性轉型之模擬 POCO

3.1.3 動態技術

在前兩個小節中提出 MTA 實體設計模型及轉型解決方案，並經實驗證實符合在多租戶中應用於 Entity 對映實體資料表的需求，但這些僅是假設執行時期動態產生的模擬程式碼，是對可調性對映技術的可行性評估。若要套用到多租戶軟體框架，這些程式碼不可能如前兩節那樣是設計階段撰寫出來的程式碼，而是需要在執行時期依不同的租戶提供不同的 Tenant-Entity，故需要執行時期動態產生 Class 的技術方案。本小節將依序列出下列三點解決方案。

1. 執行時期產生 Tenant-Entity

2. 執行時期實做 Tenant-Entity Object

3. 設計階段 Tenant-Entity 變數型態

3.1.3.1 執行時期產生 Tenant-Entity

在.NET Framework 中可以使用 Emit 來達成執行時期產生程式碼，事實上要透過 Emit 建立的並不只有 Tenant-Entity，在 Entity Framework 的運作中還需要提供 DbContext 來管理 Entity 及提供資料庫連線。因此 Emit 的過程需先建立 DbContext，再建立 Tenant-Entity，並在 Tenant DbContext 上宣告取用 Tenant-Entity 的 Property。DbContext 為了要應付可以提供不同租戶連線不同資料庫的能力，本研究 Emit 出來的 Tenant-DbContext 不直接繼承 Entity Framework 提供的 DbContext，而是由可調性多租戶軟體框架提供具有可提供連線參數的建構元的 MtaDbContext，Emit 出來的 Tenant-Context 再繼承 MtaDbContext，這樣可以減少 Emit 時候的複雜性。

```
public class MtaDbContext: DbContext
{
    public MtaDbContext()
        : base("name=" +
              MtaHelper.GetTenant(
                  (String)System.Web.Routing.RouteTable.Routes.GetRouteData(
                      new System.Web.HttpContextWrapper(System.Web.HttpContext.Current))
                  .Values["id"])
              .Connection)
    {
    }
}
```

程式碼 3.9 變更連線字串之 DbContext

框架為租戶產生 Tenant-Entity 之前須先到 Entity 資料表及 Field 資料表中取得要建立 Entity 的 metadata。有了這些資料再掌握動態時期產生 Class 的技術並配合前面兩小節的模擬程式碼作為設計藍圖，便可在執行時期產生租戶需要的 Tenant-Entity。但並非所有 Tenant-Entity 都是如模擬程式一般的繼承 Domain-Entity，在多租戶軟體框架中，可能會提供一些完全租戶自訂的 Tenant-Entity，這些 Tenant-Entity 雖然沒有 Domain Tenant 存在，但仍須直接繼承 MtaEntity。

```
String baseName = "Domain" + entity.Name;
if (!entity.IsDefault)
{
    baseName = "MtaEntity";
}

//建立Entity
TypeBuilder entityBuilder = modBuilder.DefineType(
            "Mta.Models.MtaEmitModel." + entity.Name,
            TypeAttributes.Public,
            Type.GetType("Mta.Models." + baseName));
```

程式碼 3.10 Tenant-Entity 繼承之程式片段

有了 TypeBuilder 之後，接著就可以經由 ILGnterator 來寫 IL，產生 Tenant-Entity 的 Property。在建立好 Tenant-Entity 的 TypeBuilder 與 Property 後，便可在執行時期產生租戶獨有的 Tenant-Entity。

```
PropertyBuilder propertyBuilder = typeBuilder.DefineProperty(
    name,
    PropertyAttributes.None,
    type,
    null);
MethodBuilder getMethod = typeBuilder.DefineMethod(
    "get_" + name,
    methodAttributes,
    type,
    Type.EmptyTypes);
ILGenerator getIL = getMethod.GetILGenerator();
getIL.Emit(OpCodes.Ldarg_0); //this
getIL.Emit(OpCodes.Ldfld, fieldBuilder); //將指定的值加到堆疊上
getIL.Emit(OpCodes.Ret);
propertyBuilder.SetGetMethod(getMethod);

MethodBuilder setMethod = typeBuilder.DefineMethod(
    "set_" + name, methodAttributes,
    null,
    new Type[] { type });
ILGenerator setIL = setMethod.GetILGenerator();
setIL.Emit(OpCodes.Ldarg_0);//this
setIL.Emit(OpCodes.Ldarg_1);//第一個參數
setIL.Emit(OpCodes.Stfld, fieldBuilder);
setIL.Emit(OpCodes.Ret);
propertyBuilder.SetSetMethod(setMethod);
```

程式碼 3.11 產生單純 Get/Set 的 Property 範例

```
TypeBuilder entityBuilder = CreateEntityBuilder(modBuilder, tenantId, entity);
entityBuilder ▷ entityBuilder {Name = "_User" FullName = "Mta.Models.MtaEmitModel._User"}
```

程式碼 3.12 Visual Studio 中使用中斷點觀察 TypeBuilder 的資訊

3.1.3.2 實做 Class

在前一小段提到使用 Emit 在執行時期產生 DbContext 與 Entity，但產生的 Class 是無法直接使用，我們需要將 Class 實做成 Object，在.NET Framework 中可以使用 Reflection 來可以達成。

```
ConstructorInfo mtaEmitModelCtor = mtaEmitModelType.GetConstructor(Type.EmptyTypes);
var mtaEmitModel = mtaEmitModelCtor.Invoke(null);
```

程式碼 3.13 DbContext 的 Class 實作為 Object 範例

3.1.3.3 開發時期變數型態

在完成 DbContext 與 Entity 的產生與實作之後，還需要解決一個非常重要的問題 DbContext 與 Entity 的型別都是執行時期才出現，開發時期還沒有這個型態的存在，直接宣告為執行時期才會出現的型態，勢必無法通過編譯器的檢查。var 這是許多應用 Reflection 最常搭配的變數型態，但 var 實際上是編譯時期由編譯器決定型態，所以它僅是語法糖衣，在這邊編譯器會將 var 改為 Object，所以這邊直接寫 Object 是一樣的結果，但變數型態宣告為 Object 表示只能使用 Object 的屬性及方法，雖然可以使用 Reflection 提供的 GetProperty 與 GetMethod，但使用上會相當不方便。尤其 Tenant-Entity 將會被傳遞到 Controller 與 View，這會造成整個應用程式開發的不方便性，因此使用應該使用 Dynamic Binding 的技術，只要將型態宣告為 Dynamic，變數使用屬性及方法時，編譯器就會跳過檢查，變數的真實型態會等到執行時期才確定，在那時候變數的型態才會綁定為傳入物件的型態，這就是 Dynamic Binding 或稱延遲綁定，將變數綁定的時間由編譯時期延後到執行時期。

在本小節使用了 Emit、Reflection 及 Dynamic Binding 的技術，解決需要在執行時期客製化 Class 的問題，由產生、實做到開發時期的搭配，提出完整解決方案。

3.1.4 封裝 MTA 技術

在前面幾個小節提出了對映及動態的技術方案，已經足夠做為可調性多租戶軟體框架 Model 層的技術。但對於應用程式開發者來說，額外多出許多與平常開發應用程式差異甚大的部分，例如實作物件時要如【程式碼 3.13】一樣使用 Reflection，取的 Tenant-Entity 資料時後，如果是 Universal Table Layout 就要如【程式碼 3.4】一樣，加上 TenantId 與 EntityId 的過濾條件，這些會讓開發者須要多學習不少技術及 MTA 的觀

念。因此本小節提出使用 Repository Patten 前面幾個小節提出的技術封裝起來。

本研究對 Repository Patten 做了一些調整，首先將 EFRepository 的 Generic 加入 Constraint，使用泛型限制讓 T 只能是 MtaEntity 的子類別。

```
public class EFRepository<T> : IRepository<T> where T : MtaEntity
{
    private IDbSet<T> _objectset;

    private Guid _tenantId;

    private Guid _entityId;

    public TableType TenantTableType { get; set; }

    public IUnitOfWork UnitOfWork { get; set; }
```

程式碼 3.14 IRepository 泛型限制 MtaEntity

這樣的設定可以達到將 MTA 特性的操作封裝起來，租戶隔離的邏輯，可以在 All 方法中。

```
public virtual IQueryable<T> All()
{
    if (MtaHelper.IsTableIsolationByFields(TenantTableType))
    {
        return ObjectSet.Where(p => p.TenantId == _tenantId && p.EntityId == _entityId);
    }
    else
    {
        return ObjectSet;
    }
}
```

程式碼 3.15 租戶隔離過濾

其它所有取用資料的方法都透過 All 方法過濾，例如 EFRepository 中的 FindByGuid 方法。

```
public virtual dynamic FindByGuid(Guid? GUID)
{
    return this.All().FirstOrDefault(p => p.GUID == GUID);
}
```

程式碼 3.16 租戶條件過濾

由於 LINQ 具有延遲執行，所以 All 方法中的條件會在最後真正執行查詢時才使用，而之後繼承 EFRepository 的 Entity 的 Repository 也都先串接 All 方法，如此應用程式開發者只要使用一般的 Repository 使用方法就可以不用理會 MTA 特性。

3.2 強化 Force.com Universal Table 關聯效能

本節將提出若接受資料表的鍵值使用 Guid，那在 Universal Table 關聯效能上可以有很好的提升的方法。

Force.com 提出使用 Indexes 資料表來間接彌補 Universal Table Layout 無法設定索引而造成資料搜尋低落的問題。在使用關聯式資料庫觀念開發的應用程式中，有許多索引的應用是來自於資料表之間的關聯，例如線上購物中會員搜尋其擁有的訂單就會使用索引來增加速度，其關聯如【圖 3.8】，資料範例如【圖 3.9】。

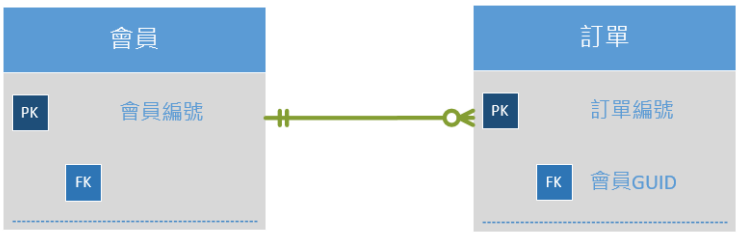


圖 3.8 會員、訂單關聯圖

會員

會員編號(PK)	姓名
1	張三

訂單

訂單編號(PK)	會員編號(FK)	訂單日期
0001	1	2016/5/1
0002	1	2016/5/1

圖 3.9 會員、訂單資料範例

但若使用 Universal Table Layout，資料將會儲存在 Data 資料表，其中屬於訂單的會員編號欄位儲存在 Value2。因為訂單的會員編號欄位為外來鍵，所以資料會複製一份儲存於 Indexes，如下圖所示。

Data

GUID	Tenant	Object	Value1	Value2	Value3
Guid_A	abc.com	會員	1	張三	
Guid_B	abc.com	訂單	0001	1	2016/5/1
Guid_C	abc.com	訂單	0002	1	2016/5/1

Indexes

Tenant	Object	FieldNum	DataGuid	Str	Int	Date
abc.com	訂單	2	Guid_B		1	
abc.com	訂單	2	Guid_C		1	

圖 3.10 會員、訂單於 Force.com Universal Table 資料範例

當應用程式要由會員資料取得其訂單資料時，就會拿著訂單資料的值『1』去 Indexes 找，獲得 GUID 欄位值為 Guid_B 及 Guid_C 的資料，然後再去 Data 取出所要的資料。這在資料量大時，確實可以有效提升速度，彌補沒有索引的缺點。但畢竟還是使用間接的方式取得資料，並且資料新增或異動時還會增加維護 Indexes 資料表的成本。

在許多應用程式的案例中，較多狀況是 Entity 只有被另一個 Entity 所擁有，也就是說一個 Entity 被兩個以上 Entity 所擁有的狀況相對較少。因此本研究藉由這現象為 Data 資料表加入一個 ForeignGUID 的欄位，並設定 GUID 欄位對 ForeignGUID 做一對多自我關聯，如下圖所示。因此 GUID 與 ForeignGUID 都具有關聯式資料庫索引。

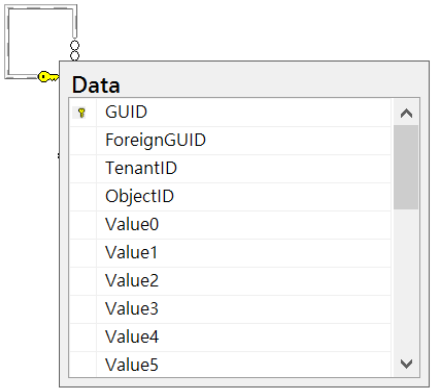


圖 3.11 改良 Force.com Universal Table Data 資料表自我關聯圖

在此設定下，應用程式設計團隊需要知道哪些 Entity 使用這種方式關聯。應用程式必須將資料寫入 ForeignGUID，用這種方式連結的 Entity 就不再使用原本邏輯資料表的關聯欄位，而是使用 GUID 及 ForeignGUID 做關聯，轉換後邏輯概念如【圖 3.12】，範例資料如【圖 3.13】。

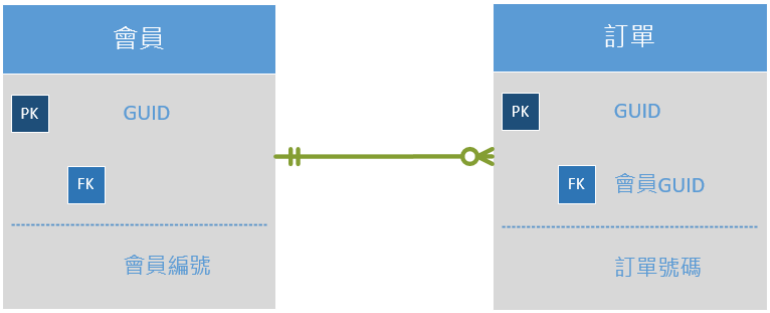


圖 3.12 Data 自我關聯之邏輯資料表概念圖

Data

GUID	Foreigen GUID	Tenant	Object	Value1	Value2	Value3
Guid_A		abc.com	會員	1	張三	
Guid_B	Guid_A	abc.com	訂單	0001	1	2016/5/1
Guid_C	Guid_A	abc.com	訂單	0002	1	2016/5/1

圖 3.13 Data 自我關聯 Data 資料範例

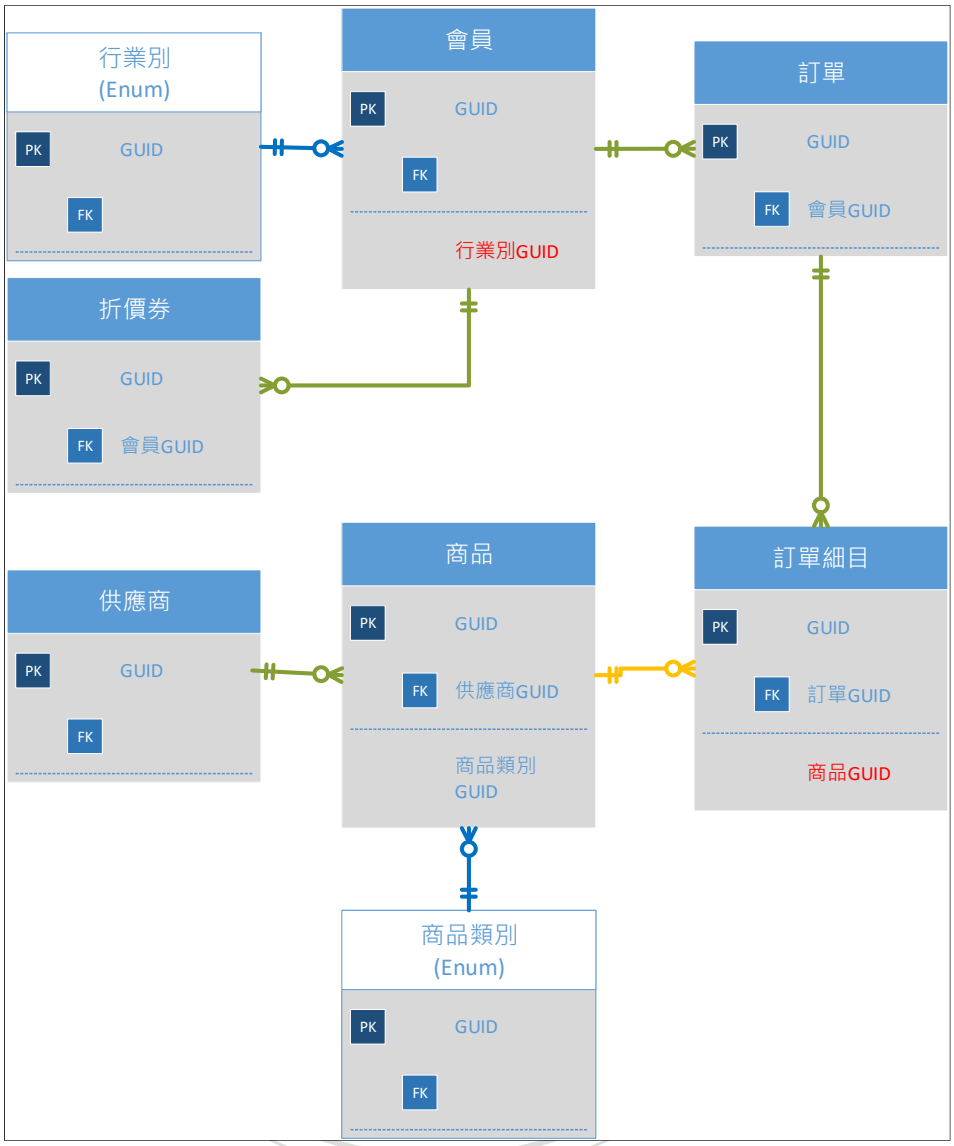


圖 3.14 Data 資料表自我關聯案例分析

【圖 3.14】範例模型來分析，若完全使用 ForeignGUID 的方式來關聯，而不使用 Force.com 提出的方式，綠色線條表示可以雙向使用索引，黃色表示單向，綠色表示單向但通常使用不支援索引的方向不多。雖然有些關聯不能獲得 ForeignGUID 的索引，但這些部分還是可以使用 Force.com 提出的方式，兩者搭配使用

3.3 Controller

前一節對於可調性多租戶軟體框架之 Model 層架構設計，本章將接續說明使用 ASP.NET MVC 框架為基礎來搭配可調性 Model 層的設計。在本框架 Controller 設計中設計一個 MtaController，應用程式中的 Controller 皆繼承 MtaController，其主要提供三項功能，分別為：

1. 方便取用 Repository 的機制。
2. 導向到租戶 Private View 的機制。
3. 常用的 Action 例如 Index、Create、Update、Detail、Delete 及 JSON 相關等等。

MtaController 類別圖如下圖所示。

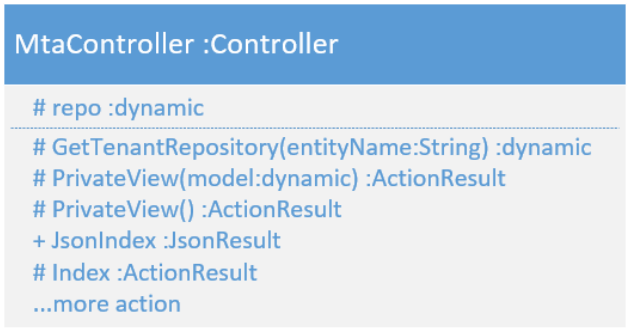


圖 3.15 MtaController 類別圖

其中 repo property 為提供用來存放 Entity 的 Repository，而 GetTenantRepository 方法提供傳入 Entity 名稱以回傳 Entity 的 Repository，PrivateView 方法則用來代替 ASP.NET MVC 中的 View 方法，可以將使用租戶自訂的 View，這部分的設計會在下一節說明。其它 Action 提供制式的 CRUD，除了可以讓 MTA Application Designer 撰寫 Controller 時直接呼叫，另一個很重要的功用，就是提供產生制式化擁有 CRUD Action 的 Controller，來使用租戶自行新增的 Entity。

3.4 View

3.4.1 接收 Tenant-Entity

ASP.NET MVC 中多數使用 Razor 語法作為 View 的技術，View 原始碼上方會宣告傳入 Model 的型態，例如圖中宣告傳入 Model 型態為 IEnumerable 的 Tenant 泛型。

```
@model IEnumerable<Mta.Models.Tenant>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>
```

程式碼 3.17 一般 Razor Model 型態宣告

而在本框架中傳入的 Model 會是 Tenant-Entity，Tenant-Entity 會在執行時期才產生，因此這邊還是需要藉助 Dynamic Binding，因此泛型集合的 Model 可以宣告為：

```
@model IEnumerable<dynamic>
```

若為單一 Tenant-Entity 則可宣告為：

```
@model dynamic
```

3.4.2 客製化 View

在 Model 可以提供客製化之後，最重要的就是 View 也要可搭配客製化。View 的客製化有兩個方向可以實行，分別為前端產生與後端產生，前端產生可以透過 JavaScript、AJAX 等技術存取 JSON 套用客製化 View，而後端產生就是在後端產生畫面回應，通常使用 Razor。本研究在這部分主要提出如何保持使用 ASP.NET MVC 的 View 技術，Razor 來客製化，本研究僅探討核心技術的部份，提供租戶所見及所得視覺化界面轉為 Razor 不在本研究範圍之內。

.cshtml 慣例被存在 View 目錄下的 Controller 同名目錄，檔案名稱與 Action 同名。只要符合這慣例，在 Controller 使用 View()就可以套用慣例位置的 View 回應到客戶

端，例如 User Controller 的 Index Action 預設搭配搭他的 View 存在 /View/User/Index.cshtml，這是他的預設規則，當然也可以透過參數指定，使用其他位置的 View。

在 ASP.NET MVC 中取得 View 是透過 View Engine 向 VirtualPathProvider 取得檔案系統中的 View。在 ASP.NET MVC 的設計中，可以透過 HostingEnvironment 來註冊新的 VirtualPathProvider，使用這方式來回應 View Engine 租戶的 View。如【圖 3.16】，紅色部分為新增的 VirtualPathProvider，接到請求後先由最後加入的 MtaPrivateView 作為第一關，到資料庫尋找客製化的 View，若有就直接回傳，若沒有則往下呼叫使用預設的 View。

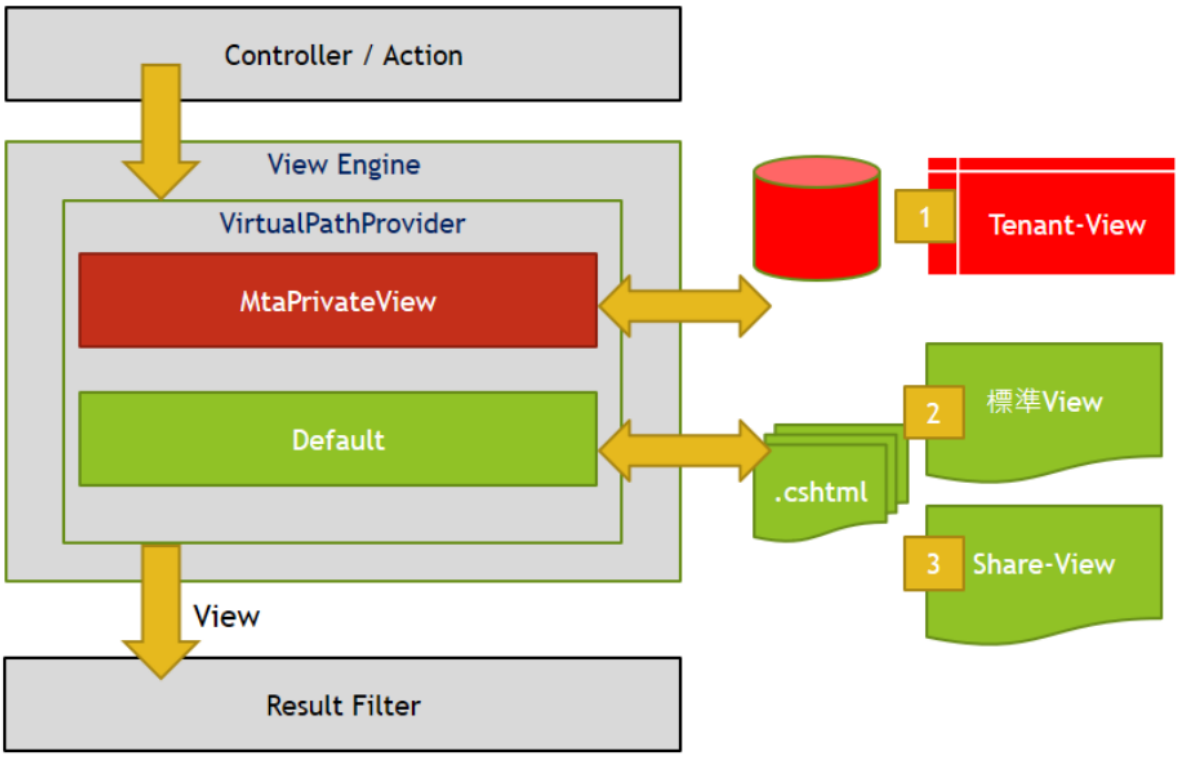


圖 3.16 租戶 View 客製化運作概念圖

3.5 租戶識別

多租戶軟體框架會需要使用到租戶識別，但租戶識別的邏輯會隨應用程式的需求而有所不同，在許多例子中會使用登入帳號作為識別，但有時候也可能使用網址或 Client IP 作為識別。租戶識別的邏輯必須等待多租戶應用程式開發甚至部署時才做決定，但租戶識別的使用卻存在框架中的程式碼中，這種問題可以使用 High-order-function 的觀念將 Function 當作參數傳入。在 C#中一個很好實踐 High-order-function 的方式就是使用 Func 委派。本框架中由 MtaHelper.GetTenantName 提供租戶識別的服務，GetTenantName 的型態為 Func<String,String>，這型態表示可以委派一個傳入值為 String、回傳值為 String 的 Function。如【圖 3.17】所示，框架設計時可以直接由 GetTenantName 取租戶識別，但此時還不知道租戶識別的邏輯。而應用程式設計時 Application Developer 提供租戶識別的邏輯。在【圖 3.17】例子中兩種不同的租戶識別

邏輯，帶入不同的 lambda 作為 Function 來提供租戶識別邏輯。

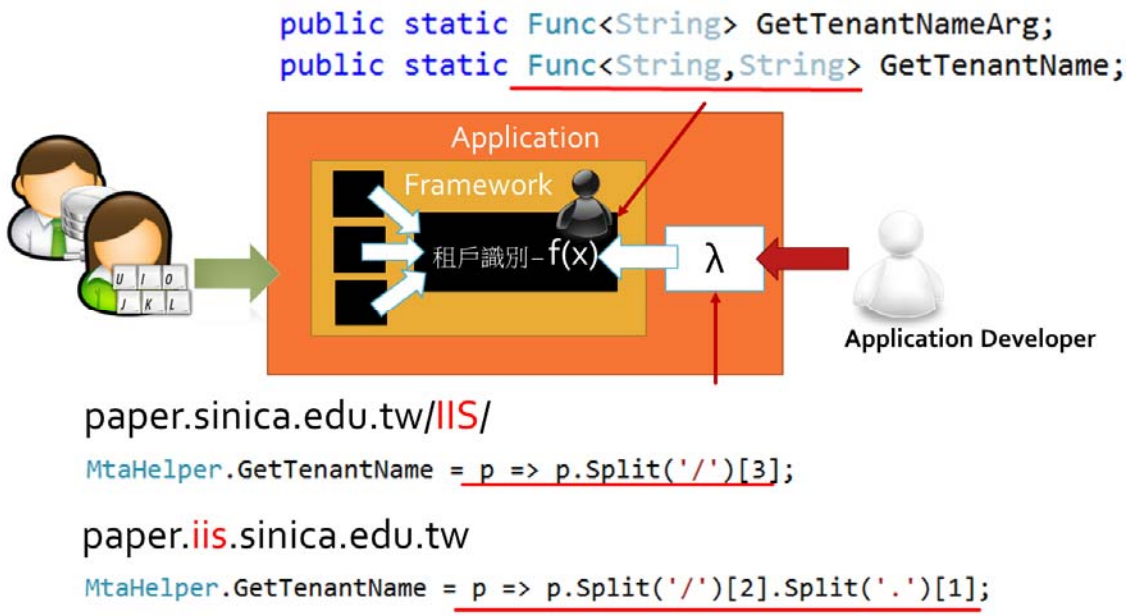


圖 3.17 租戶識別運作概念圖

第四章　可調式多租戶軟體框架使用探討

本章將以使用框架開發應用軟體的角度切入來探討本框架的設計，來證明對於一個熟悉 ASP.NET MVC 的程式設計師來說，是可以很容易使用本框架作為開發。本章將如同第三章依循 Model、Controller 及 View 進行說明。

4.1 Model

使用本框架建立 Model 的步驟為如下:

(1) 撰寫 Domain-Entity POCO。

(2) 撰寫 Domain-Entity Repository。

(3) 於系統中建立 Tenant-Entity 相關資訊。

4.1.1 撰寫 Domain-Entity POCO

撰寫 Domain-Entity POCO 如同在 Entity Framework 中撰寫 Entity POCO 類似，需另外依循的規範只有三個:

(1) 命名加上前置 Domain。

(2) 繼承 MtaEntity。

(3) Property 需加上 virtual 修飾詞，以供 Tenant-Entity 需要時 override。

如下範例程式碼：

```
public class DomainInProceeding : MtaEntity
{
    [Display(Name = "出版狀態")]
    [MaxLength(1)]
    public virtual String PublishStatus { get; set; }

    [Display(Name = "作者")]
    [MaxLength(200)]
    public virtual String Author { get; set; }

    [Display(Name = "篇名")]
    [MaxLength(200)]
    public virtual String PaperTitle { get; set; }

    [Display(Name = "書名")]
    [MaxLength(200)]
    public virtual String Booktitle { get; set; }

    [Display(Name = "發表日期")]
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    public virtual DateTime? PubDate { get; set; }

    [Display(Name = "頁碼")]
    public virtual Int32? Page { get; set; }

    //--More Propertyies
```

程式碼 4.1 Domain-Entity 範例程式碼

4.1.2 撰寫 Domain-Entity Repository

撰寫 Domain-Entity Repository 與一般 Repository 結構類似，僅須注意以下規範：

(1) 使用泛型並限制其 Domain-Entity。

(2) 名稱為 Tenant-Entity 的名稱加上 Repository。

(3) 提供傳入 Tenant GUID 的建構元。

(4) Repository 取資料的方法回傳型態為 dynamic。

```
public class InProceedingRepository<T> : EFRepository<T> where T : DomainInProceeding
{
    public InProceedingRepository(Guid TenantGUID)
        : base(TenantGUID)
    {

    }

    public dynamic FindDataByPaperTitle(String PaperTitle)
    {
        return this.All().FirstOrDefault(p => p.PaperTitle == PaperTitle);
    }
}
```

程式碼 4.2 Domain-Entity Repository 範例程式碼

4.1.3 於系統中建立 Tenant-Entity 相關資訊

每個租戶需要有 Tenant-Entity 的基礎資訊，以提供租戶客製化 Tenant-Entity，包含 Property 的新增、必填限制、長度限制及顯示名稱修改。

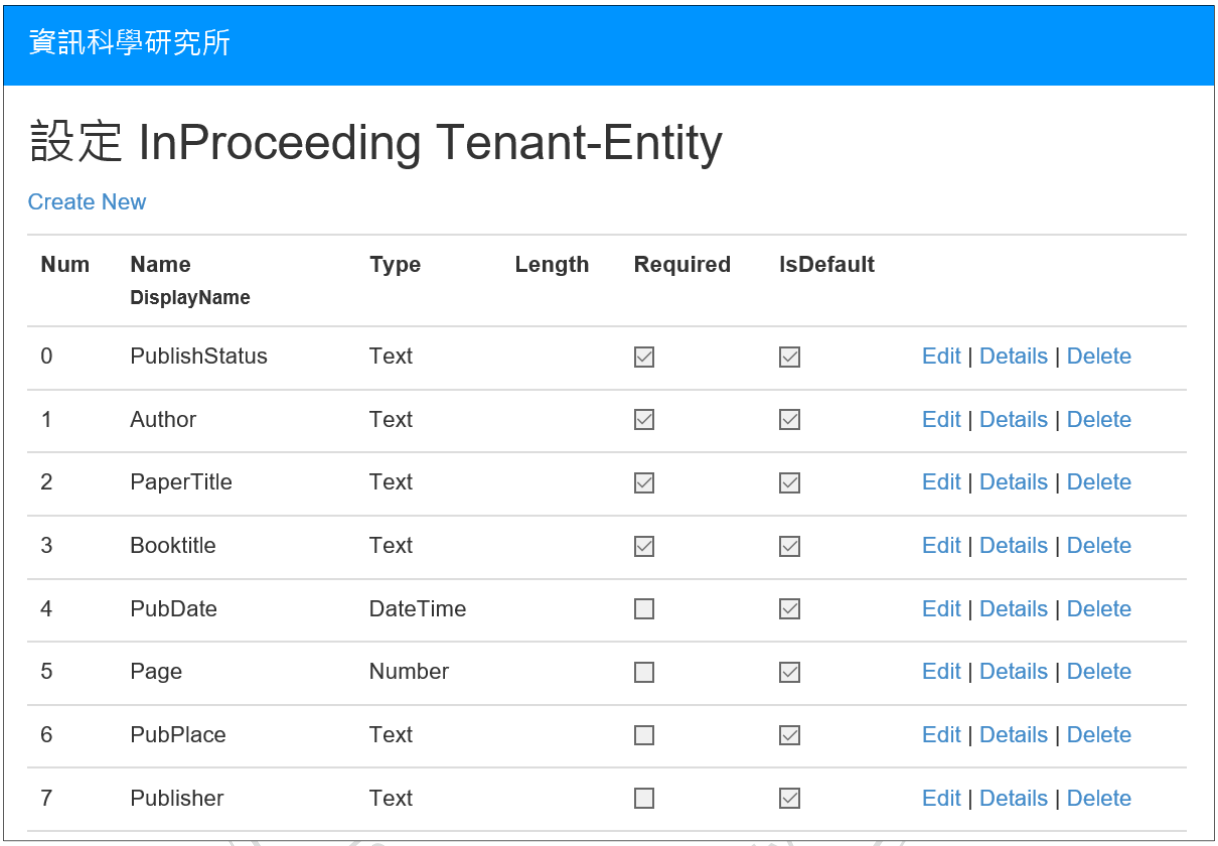


圖 4.1 租戶變更資料綱要介面

4.2 Controller

編寫 Controller 的作法與 Visual Studio 自動產生的 Controller 並且加入 Repository 使用來 Entity 的基本作法差異不大，以下依循 Index、Detials、Create/Get、Create/Post、Edit/Get、Edit/Post、Delete 及 Delete Confirmed 八個在 ASP.NET MVC 常用的 Actioni 做比較說明。

Index	

原本	`public ActionResult Index()`
	`{`
	`  return View(repo.All().ToList());`
	`}`

框架	`public ActionResult Index()`
	`{`
	`  return PrivateView(repo.All());`
	`}`

✓ 改用 PrivateView。 ✓ 不使用 ToList 等擴充方法，將執行時間延後到 View 內。	

程式碼 4.3 Index Action 差異比較

Detials

原本

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    InProceeding inProceeding = repo.Find(id);
    if (inProceeding == null)
    {
        return HttpNotFound();
    }
    return View(inProceeding);
}
```

框架

```
public ActionResult Details(Guid GUID)
{
    if (GUID == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    dynamic inProceeding = repo.Find(GUID);
    if (inProceeding == null)
    {
        return HttpNotFound();
    }
    return PrivateView(inProceeding);
}
```

- ✓ 使用 dynamic 做為 Entity 的型別。
- ✓ 改用 PrivateView。
- ✓ (選擇)傳入 Id 可以改用 MTA-Entity GUID。

程式碼 4.4 Details Action 差異比較

Create/Get	

原本	`public ActionResult Create()`
	`{`
	`  return View();`
	`}`

框架	`public ActionResult Create()`
	`{`
	`  return PrivateView(rep.Create());`
	`}`

✓ 改用 PrivateView。 ✓ 由於 View 的 Model 型態為 dynamic，因此 View 無法知道 Model 型態，所以需要傳送一個空的 Entity 過去。	

程式碼 4.5 Create/Get Action 差異比較

Create/Post

原本

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create
       ([Bind(Include = "欄位1,欄位2")] InProceeding inProceeding)
{
    if (ModelState.IsValid)
    {
        repo.Add(inProceeding);
        repo.UnitOfWork.Commit();
        return RedirectToAction("Index");
    }
    return View(inProceeding);
}
```

框架

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create
       (FormCollection FromValue)
{
    dynamic inProceeding = repo.Create();
    if (TryUpdateModel(inProceeding, new String[] { "欄位1", "欄位2" }) &&
        ModelState.IsValid)
    {
        repo.Add(inProceeding);
        repo.UnitOfWork.Commit();
        return RedirectToAction("Index");
    }
    return PrivateView(inProceeding);
}
```

✓ 改用 PrivateView。

✓ 改用 TryUpdateModel 填入 Post 內容。

程式碼 4.6 Create/Post Action 差異比較

Edit/Get

原本

```
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    InProceeding inProceeding = repo.Find(id);
    if (inProceeding == null)
    {
        return HttpNotFound();
    }
    return View(inProceeding);
}
```

框架

```
public ActionResult Edit(Guid GUID)
{
    if (GUID == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    dynamic inProceeding = repo.Find(GUID);
    if (inProceeding == null)
    {
        return HttpNotFound();
    }
    return PrivateView(inProceeding);
}
```

- ✓ 使用 dynamic 做為 Entity 的型別。
- ✓ 改用 PrivateView。
- ✓ (選擇)傳入 Id 可以改用 MTA-Entity GUID。

程式碼 4.7 Edit/Get Action 差異比較

Edit/Post

原本

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit
       ([Bind(Include = "欄位1,欄位")] InProceeding inProceeding)
{
    if (ModelState.IsValid)
    {
        ((DbContext)repo.UnitOfWork.Context).Entry(inProceeding)
                                                .State = EntityState.Modified;
        repo.UnitOfWork.Commit();
        return RedirectToAction("Index");
    }
    return View(inProceeding);
}
```

框架

```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit(Guid GUID,FormCollection FromValue)
{
    dynamic inProceeding = repo.Find(GUID);
    if (TryUpdateModel(inProceeding, new String[] {"欄位1", "欄位2" }) &&
        ModelState.IsValid)
    {
        ((DbContext)repo.UnitOfWork.Context).Entry(inProceeding)
                                            .State = EntityState.Modified;
        repo.UnitOfWork.Commit();
        return RedirectToAction("Index");
    }

    return PrivateView(inProceeding);
}
```

✓ 改用 TryUpdateModel 填入 Post 內容。

✓ 改用 PrivateView。

程式碼 4.8 Edit/Post Action 差異比較

Delete

原本

```
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    InProceeding inProceeding = repo.Find(id);
    if (inProceeding == null)
    {
        return HttpNotFound();
    }
    return View(inProceeding);
}
```

框架

```
public ActionResult Delete(Guid GUID)
{
    if (GUID == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    dynamic inProceeding = repo.Find(GUID);
    if (inProceeding == null)
    {
        return HttpNotFound();
    }
    return PrivateView(inProceeding);
}
```

✓ 使用 dynamic 做為 Entity 的型別。

✓ 改用 PrivateView。

程式碼 4.9 Delete Action 差異比較

Delete Confirmed	

原本	[HttpPost, ActionName("Delete")]
	[ValidateAntiForgeryToken]
	public ActionResult DeleteConfirmed(int id)
	{
	InProceeding inProceeding = repo.Find(id);
	repo.Delete(inProceeding);
	repo.UnitOfWork.Commit();
	return RedirectToAction("Index");
	}

框架	[HttpPost, ActionName("Delete")]
	[ValidateAntiForgeryToken]
	public ActionResult DeleteConfirmed(Guid GUID)
	{
	dynamic inProceeding = repo.Find(GUID);
	repo.Delete(inProceeding);
	repo.UnitOfWork.Commit();
	return RedirectToAction("Index");
	}

✓	使用 dynamic 做為 Entity 的型別。

程式碼 4.10 Delete Confirmed Action 差異比較

在以上的比較中可以發現主要的改變只有三項，分別為：

(1) 使用 dynamic 作為 Entity 的型別。

(2) 改用 PrivateView。

(3) 改用 TryUpdateModel 填入 Post 內容。

這樣的改變非常容易達到，對於已經會使用 ASP.NET MVC 的程式設計師來說，可以很容易的改用框架設計程式。

在上面的比較中，使用 Repository 是直接使用 repo 變數，這在前一章已經提過。這是 MtaController 類別提供的 dynamic 型態變數，會自動依 Controller 的名稱將 Repository 指定給 repo；若 Controller 名稱與要使用的 Tenant-Entity 名稱不一致，那可以使用 AuotImplementRepositoryActionFilter Attribute 來指定要使用的 Tenant-Entity 名

稱，框架會依指定的 Tenant-Entity 名稱，將其 Repository 指定給 repo。

```
[AuotImplementRepositoryActionFilter(entityName = "User")]
public class UserRepController : MtaController
{
    public ActionResult Index()
    {
        return PrivateView(repo.All());
    }
```

程式碼 4.11 AutoImplementRepositoryActionFilter 指定使用 Entity

4.3 View

框架提供將租戶客製化 View 存於資料庫內，當 Controller 使用 PrivateView 方法套用 View 時，框架會先優先選擇租戶客製化 View 來套用，若租戶沒有客製化 View 則套用公用 View。

客製化 View 的原始碼與 ASP.NET MVC 通常使用的 View 做法可以說是相同，但由於 Tenant-Entity 執行時期才產生，所以 View 宣告 Entity(Model)型態的部分依然是使用 dynamic。但由於 dynamic 無法使用 lambda，所以無法使用 HtmlHelper 的 DisplayFor()、LabelFor()及 EditorFor()等等這些支援使用 lambda 的方法，所以須改用 Display()、Label()或 Editor()等等這些弱型別的方法。

4.4 效能測試

本研究提出之框架使用了 Emit、Reflection 及 Dynamic 等技術，讓使用框架開發的應用程式可以客製化 Entity，帶來了彈性，相對的也會造成效能降低。本節將對本框架使用的主要技術，造成的效能降低進行測試研究，以了解需要多少效能來換取彈性。

4.4.1 測試環境

本框架模擬實作部分，參考中央研究院的研究成果系統之功能實作，並建立數個租戶，表示中央研究院內的各研究單位(所/中心)。為測試不同的綱要映射技術，將資訊

所設定為使用 Universal Table Layout，另將語言所設定為使用 Private Table Layout，以下將稱之為 MTA 組與 MTA Private 組。另使用單純的 ASP.NET MVC 與 Entity Framework 建立一個相同功能規格之研究成果系統作為基準，以下將稱之為 Not MTA 組。由於 Not MTA 組也是使用 Code-First 來建立資料庫與資料表，故其資料綱要可以與 MTA Private 組視為相同。因此 MTA Private 組與 Not MTA 組的測試數據將不受資料綱的差異影響，最具比較意義。

以下將使用 Visual Studio Enterprise 2015 中的效能測試工具進行測試。基於測試案例主要針對框架使用的動態技術帶來效能降低進行測試，比對各組織間的相對差距，為了降低網路、資料庫等其它不相關因素所帶來的影響，測試採用單機，並且資料量維持一筆，以降低資料庫在量大資料筆數時會帶來的效能差距。在資料量大的狀況下，不僅不同的資料綱要之間資料筆數與效能的差異為正相關。另外由於多租戶系統為了可能需要租戶分離到不同主機或合併到同一個主機，最適合使用 Guid 作為 Key，但 Guid 有相對的缺點，因為 Guid 並不像識別值增量(IDENTITY)這種循序式資料值一樣，後來的資料值必定會大於之前的資料值。MS SQL Server 資料表的索引會有一個是 Clustered Index，決定資料表儲存資料的實際順序，這索引一般會設定在 Key。然而這種 Clustered Index 最適合有序的資料值，MS SQL Server 的索引使用 B-Tree 資料結構作為搜尋，這種資料結構使用 Guid 時很容易導致索引破碎(Index Fragment)，而降低些許效能。在使用 Guid 的系統中，DBA 通常需要定時執行 Index Rebuild，因為 Guid 新舊值間屬於無序資料值，降低效能的狀況也隨之不穩定。因此在各組效能差異非常逼近的實驗中，執行連續新增、連續刪除或大量資料中查詢、修改的測試實驗中，會讓實驗數據失去意義。

4.4.2 Web 效能測試

本小節使用 Visual Studio Enterprise 2015 中的 Web 效能測試，來了解各組在效能在 CRUD 上的差異：

	第一次	第二次	第三次	第四次	第五次	平均
Create	0.032	0.038	0.034	0.038	0.027	0.0338
Delete	0.031	0.028	0.024	0.036	0.029	0.0296
Update	0.019	0.021	0.023	0.031	0.032	0.0252
Select	0.027	0.027	0.036	0.026	0.028	0.0288

表 4.1 Not MTA CRUD Web 效能測試報告

	第一次	第二次	第三次	第四次	第五次	平均
Create	0.032	0.034	0.033	0.037	0.027	0.0326
Delete	0.037	0.033	0.034	0.031	0.029	0.0328
Update	0.021	0.035	0.024	0.033	0.031	0.0288
Select	0.027	0.022	0.029	0.031	0.033	0.0284

表 4.2 MTA Private CRUD Web 效能測試報告

	第一次	第二次	第三次	第四次	第五次	平均
Create	0.033	0.034	0.038	0.029	0.031	0.033
Delete	0.028	0.031	0.028	0.031	0.03	0.0296
Update	0.026	0.037	0.028	0.026	0.029	0.0292
Select	0.029	0.032	0.029	0.028	0.026	0.0288

表 4.3 MTA Universal CRUD Web 效能測試報告

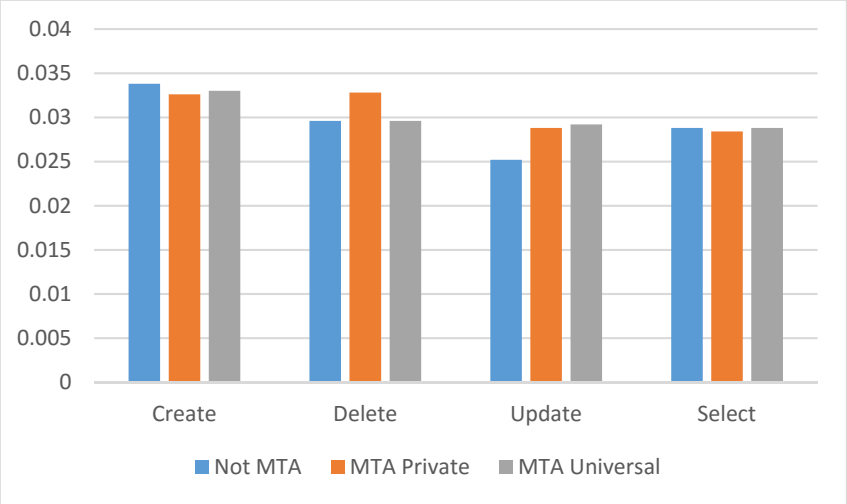


圖 4.2 Web 效能測試結果

由【圖 4.2】我們得知，三組的 Web 效能測試結果各組幾乎難分高低，這已經不是使用幾次效能測試就能得出細微的差異。因此接下來將以單筆查詢來做壓力測試，使用較大的數量統計出細微差異。

4.4.3 壓力測試

為了減少資料庫因為綱要不同而造成資料庫的效能差異，資料庫內僅提供一筆資料查詢。每組共測試 5 次，每次模擬 250 人進行 5 分鐘的 Request，獲得 Request 的次數，結果如【表 4.4】。由測試數據我們獲得 MTA 兩組僅與 Not MTA 的 Request 數量近差距 1:0.98。經由本實驗得知本論文提出之框架使用的技術雖然會降低效能，但降低的效能僅不到 2%，更精確的以相同 Schema 的 Private MTA 這組來比較，僅差距不到 1.5%，此測試結果非常理想。這結果表示，僅犧牲 1.5%~2%的效能，能帶來租戶客製化的彈性。

	五分鐘，250 人 Request 數量					合計	與 Not MTA 比例
	第一次	第二次	第三次	第四次	第五次		
Not MTA	29,660	30,351	30,051	30,384	31,431	151,877	
Private MTA	30,262	29,305	29,646	30,426	30,013	149,652	1 : 0.985
Universe MTA	29,621	29,011	29,031	31,101	30,349	149,113	1 : 0.981

表 4.4 壓力測試報告

第五章　結論與未來研究方向

5.1　結論

本研究建立在可以依照租戶及多租戶應用程式的特性來選擇不同的綱要映射技術，以達到較佳的效能與資源利用進而達到確實節省本。因此依本研究提出的可調性多租戶實體模式並參考各種綱要映射技術的特性，將可以讓框架或應用程式應用在各種狀況，無須為了不同狀況而需要重新開發框架或應用程式。

由第四章說明可以得知，對於使用本框架開發應用程式的程式設計來說，需要做的開發習慣改變與學習並不多，除了依照規則設計 Domain-Entity 之外，就是使用 dynamic 貫穿 MVC，並因 dynamic 的特性做了些微的改變，因此很適合用來作為多租戶應用程式的開發。

5.2　未來研究方向

5.2.1　客製化商業邏輯

應用程式需要客製化的部分除了欄位與畫面呈現之外，還有一些系統會涉及到客製化商業邏輯，例如客製化的欄位不是只有儲存，還需要用來判斷或處理，另外流程的改變也是在一些需要審核流程的系統常見的狀況。

5.2.2　特定 View 語言轉換技術

本研究僅提出在 Server-Side 產生 View 的技術，以完整貫穿 MVC，但還缺乏一個讓租戶客製化的介面，事實上 View 技術並非僅限定於 Server-Side，有些應用程式的特性更適合讓 View 呈現技術是在 Client-Side，例如搭配使用 AngularJS 或 React。不論是

Server-Side 的技術或 Client-Side 的技術，都不適合讓租戶直接接觸，除了技術門檻之外，這也牽扯到安全性問題。因此需要制定出一個描述 View 的專用語言，除了系統提供所見即所得或簡易的設定介面儲存為專用語言之外，專用語言可轉換為各種 View 技術，如下圖所示：

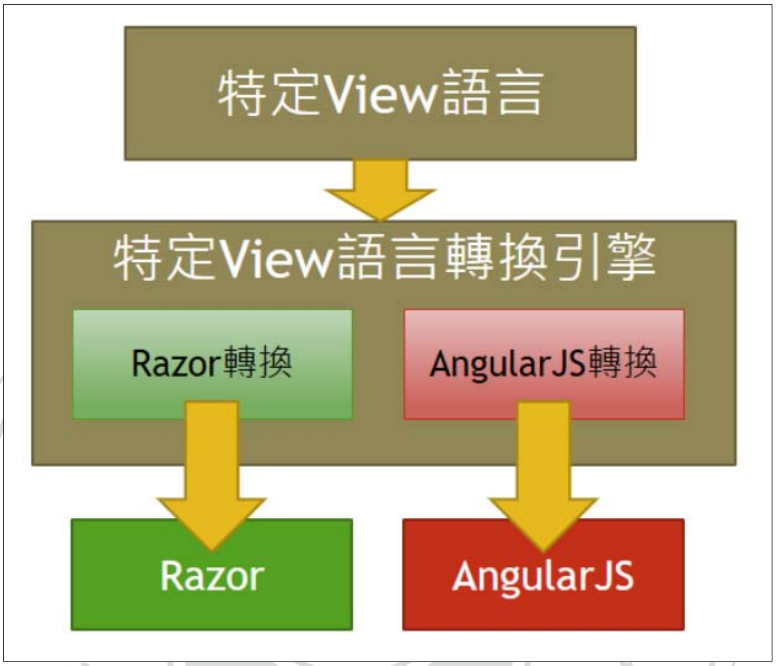


圖 5.1 特定 View 語言轉換引擎

參 考 文 獻

【1】 AlAlwan, M. H., & Zaghloul, S. (2013). ANALYSIS OF SaaS MULTI-TENANT DATABASE IN A CLOUD ENVIRONMENT. In *The Third International Conference on Digital Information Processing and Communications* (pp. 523-528). The Society of Digital Information and Wireless Communication.

【2】 Aulbach, S., Grust, T., Jacobs, D., Kemper, A., & Rittinger, J. (2008, June). Multi-tenant databases for software as a service: schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (pp. 1195-1206). ACM.

【3】 Weissman, C. D., & Bobrowski, S. (2009, June). The design of the force. com multitenant internet application development platform. In *SIGMOD Conference*(pp. 889-896).

【4】 Liao, C. F., Chen, K., & Chen, J. J. (2012, December). Toward a tenant-aware query rewriting engine for universal table schema-mapping. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on* (pp. 833-838). IEEE.

【5】 Multi-Tenant Data Architecture (2015)
https://msdn.microsoft.com/en-us/library/aa479086.aspx

【6】 LINQ 和 ADO.NET (2016)
https://msdn.microsoft.com/zh-tw/library/bb399365(v=vs.110).aspx

【7】 Entity Framework 概觀 (2016)
https://msdn.microsoft.com/zh-tw/library/bb399567(v=vs.110).aspx

【8】 用 Repository Pattern 抽離對 Entity Framework 的依賴
http://ithelp.ithome.com.tw/articles/10157484

【9】 反映 (C# 和 Visual Basic)

https://msdn.microsoft.com/zh-tw/library/ms173183.aspx