

SAMPLE CHAPTER

HIBERNATE IN ACTION

HIBERNATE

實作手冊



Christian Bauer
Gavin King

中文翻譯：何孟翰

 MANNING



Hibernate 實作手冊

by Christian Bauer

and

Gavin King

中文翻譯：何孟翰

Chapter 6

進階對應概念

這一章包含了

- Hibernate 型態系統
- 客製化對應型態
- Collection 對應
- 一對一與多對多聯合

在第三章，我們介紹了 Hibernate 所提供的 ORM 最重要的特性。你已經碰到了基本的類別和屬性的對應，繼承對應，元件對應與一對多聯合對應。我們現在繼續瀏覽這些主題而進入更奇特的 collection 和聯合對應。在許多場合，我們會警告你不要未經考慮就使用這特性。舉例來說，只用元件對應和一對多（偶而是一對一）聯合實作任何領域模型是有可能的。奇特的對應特性應該注意使用，或許在大多場合應該避免。

在我們開始討論奇特的特性，你需要一個對 Hibernate 型態系統更精確的了解——特別是個體與值型態之間的差別。

6.1 了解 Hibernate 的型態系統

在第三章，3.5.1 節 " 個體與值型態 "，我們先分辨了個體和值型態，一個 ORM 在 Java 中的中心概念。我們必須詳盡解釋分辨以讓你可以完全了解 Hibernate 個體，值型態與對應型態的型態系統。

個體是系統中的粗粒類別。你通常定義系統的特性是由牽涉的個體而來：" 使用者對了一個項目下了一個標單 " 是一個典型的特性定義，指出了三個個體。值型態的類別通常不會出現在商業需求之中，他們通常是細粒類別表現如字串，數字和財政數量。通常，值型態不會出現在特性定義：" 使用者改變了帳單地址 " 是一個例子，假設 `Address` 是一個值型態，但這並非典型。

更正式的來說，一個個體是任何的類別，它的實例有他們自己的永續辨識元。一個值型態是一個類別他並沒有定義某種永續辨識元。實際上來說，這指的是個體型態是有辨識屬性的類別，而值型態類別是依靠著個體的。

在執行時期，你有一個個體實例的圖穿插著值型態的實例。這個體實例可以是下列三種永續生命週期狀態之一：暫時，分離或永續。我們不考慮將這些生命週期狀態套用到值型態的實例。

然而，個體有他們自己的生命週期。Hibernate 的 `Session` 界面中 `save()` 和 `delete()` 函數可套用到個體類別的實例，而絕不會到值型態的實例。永續生命週期的值型態實例是完全和擁有一個體的實例綁在一起的。舉例來說，使用者名稱當使用者被儲存時也會被永續；它永遠不會獨立於使用者之外而變得永續。

在 Hibernate，一個值型態可能會定義聯合：它有可能由值型態實例走到另外一個個體。然而，絕不可能從另外一個體走訪回值型態實例。聯合關係總是指到個體。這指的是值型態實例在從資料庫取出時，是正好被一個個體所擁有，它絕不被共享。

在資料庫的階層，任何表格都被視為個體。然而，Hibernate 提供了某些建構以在 Java 程式中隱藏了資料庫階層的個體。舉例來說，一個多對多的聯合對應對於應用程式來說隱藏了中間的聯合表格。一個字串的 collection（更精確地來說，一個值型態實例的 collection）的行為對應用程式來說是值型態；然而，它對應到它自己的表格。雖然這些特性一開始看起來很好（他們簡化了 Java 程式碼），我們已經開始懷疑他們。不可避免地，這些隱藏的個體最後在商業需求演化時是會接觸到應用程式的。以多對多聯合表格為例，當應用程式變得成熟時常常需要增加額外的欄位。我們幾乎準備建議每個會接觸到應用程式的資料庫階層個體都是一個體類別。舉例來說，我們已經包含將模型多對多的聯合而成兩個一對多的聯合用一個中界個體類別。我們會留下最後的決定權給你，然而，在這章的之後回到多對多個體聯合的議題。

所以，個體類別總是使用 `<class>`，`<subclass>`，`<joined-subclass>` 這些對應元素對應到資料庫。值型態是如何被對應的呢？

考慮 CaveatEmptor 中 User 和 email 地址的對應：

```
<property
  name="email"
  column="EMAIL"
  type="string"/>
```

讓我們集中注意在 `type="string"` 屬性。在 ORM 中，你必須處理 Java 型態和 SQL 資料型態。這兩者不同的型態系統必須被橋接。這就是 Hibernate 對應型態的工作，且 `string` 是 Hibernate 內建的對應型態。

`string` 對應型態不是 Hibernate 唯一內建的。Hibernate 伴隨著許多不同的對應型態，且對於原始的 Java 型態和某些 JDK 的類別定義了預設的永續策略。

6.1.1 內建對應型態

Hibernate 的內建對應型態通常和所對應的 Java 資料型態共享相同的名稱。然而，對某一個特定的 Java 資料型態可能有多於一種 Hibernate 資料型態。再者，內建

的型態不能執行任何的轉換，如同對應一個 VARCHAR 的欄位到 Java 的 Integer 的屬性值。對於這種事你可以定義你自己的自訂值型態，如同在這裡之後所討論到的。

我們在會討論到基本的，資料和型態，大型物件與其它多樣內建對應型態且展示 Java 和 SQL 的資料型態是如何處理的。

Java 原始對應型態

基本的對應型態在表 6.1 對應 Java 的原始型態（或它們的包裹型態）到適合的內建 SQL 標準型態

Table 6.1 原始型態

對應型態	Java 型態	標準 SQL 內建型態
integer	int or java.lang.Integer	INTEGER
long	long or java.lang.Long	BIGINT
short	short or java.lang.Short	SMALLINT
float	float or java.lang.Float	FLOAT
double	double or java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte or java.lang.Byte	TINYINT
boolean	boolean or java.lang.Boolean	BIT
yes_no	boolean or java.lang.Boolean	CHAR(1) ('Y' or 'N')
true_false	boolean or java.lang.Boolean	CHAR(1) ('T' or 'F')

你可能注意你的資料庫並不支援表 6.1 中列出的型態中的一些。列出的名字是 ANSI 標準資料型態。大部份的資料庫廠商忽視這部份的 SQL 標準（因為他們的資料型態有時比標準來得早）。然而，JDBC 驅動程式提供了一部份廠商自訂 SQL 資料型態的抽向，當執行資料操作語言時，允許 Hibernate 操作 ANSI 標準型態。對於資料庫指定的 DDL 生成，Hibernate 翻譯 ANSI 標準的型態到適合的廠商指定型態，

使用支援的內建 SQL 方言。(如果你正使用 Hibernate 來作為資料存取和資料綱要定義，你通常不需要擔心 SQL 資料型態)。

日期和時間

表格 6.2 列出了 Hibernate 的型態與日期，時間和時間戳記的聯合。在你的領域模型，你可以選擇使用 `java.util.Date` 與 `java.util.Calendar` 或 `java.util.Date` 的子類別，定義在 `java.sql` 的套件兩者之一來表現日期與時間。這是依照個人口味，且我們會讓你自己決定，然而請確保你是一致的。

Table 6.2 日期和時間型態

對應型態	Java 型態	標準 SQL 內建型態
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE

大型物件對應型態

表格 6.3 列出了 Hibernate 對處理二進位物件和大型物件的型態。注意在這些型態沒有任一種可以用作辨識元的屬性。

Table 6.3 二進位與大型物件型態

對應型態	Java 型態	標準 SQL 內建型態
binary	byte[]	VARBINARY (or BLOB)
text	java.lang.String	CLOB
serializable	any Java class that implements java.io.Serializable	VARBINARY (or BLOB)
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB

`java.sql.Blob` 和 `java.sql.Clob` 是處理 Java 大型物件中最有效率的方法。不幸地，`Blob` 和 `Clob` 的實例只有在 JDBC 的交易完成才有用。所以如果你的永續類別定義了一

個 `java.sql.Clob` 或 `java.sql.Blob` 的屬性（無論如何這不是一個好意見），你會被這個類別的實例是如何被使用所限制。特別來說，你將不能使用該類別的實例如同一個分離物件。而且，許多 JDBC 的驅動程式並沒有支援操作 `java.sql.Blob` 與 `java.sql.Clob` 的特性。所以，更合適的是對應大型物件使用 `binary` 或 `text` 的對應屬性，且假設整個大型物件的取出到記憶體中並不是效能的殺手。

注意你可以在 Hibernate 的網站上發現對於大型物件使用最新的設計樣式與訊息，那些是對特定平台的方法。

許多 JDK 中的對應型態

表格 6.4 列出了 Hibernate 的型態，可表現許多其它 JDK 中可能在資料庫中表現為 `VARCHAR` 的 Java 型態。

Table 6.4 其它 JDK 相關的型態

對應型態	Java 型態	標準 SQL 內建型態
class	java.lang.Class	VARCHAR
locale	java.util.Locale	VARCHAR
timezone	java.util.TimeZone	VARCHAR
currency	java.util.Currency	VARCHAR

當然，`<property>` 並不是唯一 Hibernate 對應元素有 `type` 屬性的地方。

6.1.2 使用對應型態

所有基本對應型態會在 Hibernate 對應文件的所有地方出現，在一般的屬性，辨識屬性和其他對應元素。

`<id>`、`<property>`、`<version>`、`<discriminator>`、`<index>` 和 `<element>` 元素定義了 `type` 的屬性。（然而，對於辨識元和鑑別者型態來說，是有哪種對應基本型態是可運作的一些限制。）

你可以看到內建的對應型態在 `BillingDetails` 類別中是如何有用的：

```
<class name="BillingDetails"
      table="BILLING_DETAILS"
      discriminator-value="null">
  <id name="id" type="long" column="BILLING_DETAILS_ID">
```

```
        <generator class="native"/>
    </id>

    <discriminator type="character" column="TYPE"/>

    <property name="number" type="string"/>
    ...
</class>
```

BillingDetails 類別是如同個體被對應。它的 discriminator，辨識元和 number 屬性是值型態，且我們使用內建的 Hibernate 對應型態來指定轉換策略。

通常在 XML 文件中額外指定內建的對應型態並不是必須的。舉例而言，如果你有一個屬性的 Java 型別是 java.lang.String，Hibernate 會使用反射發現這個而預設使用 string。我們可以輕易的簡化之前的對應範例。

```
<class name="BillingDetails"
      table="BILLING_DETAILS"
      discriminator-value="null">

    <id name="id" column="BILLING_DETAILS_ID">
        <generator class="native"/>
    </id>

    <discriminator type="character" column="TYPE"/>

    <property name="number"/>
    ....
</class>
```

這種方式不能操作良好最重要的地方是 java.util.Date 屬性。預設時，Hibernate 會解釋 Date 成 timestamp 對對應。如果你不想要永續日期時間資訊兩者，你將需要額外指定 type="time" 或 type="date"。

對每一個內建對應型態，常數是被定義在 net.sf.hibernate.Hibernate 之中。舉例而言，Hibernate.STRING 表達了 string 的對應型態。這些常數對於查詢參數連接，如同在第七章中更仔細地討論：

```
session.createQuery("from Item i where i.description like :desc")
    .setParameter("desc", desc, Hibernate.STRING)
    .list();
```

這些常數對於寫程式操作 Hibernate 的對應後設文件也是有用的，如同我們在第三章中所討論過的。

當然，Hibernate 並不限制在內建的對應型態。我們視延伸對應型態系統是讓 Hibernate 如此彈性的一個核心特性和重要的觀點。

創建自訂對應型態

物件導向語言像 Java 使得藉由撰寫新的類別來定義新的型態是簡單的。事實上，這是一個物件導向的基本定義。如果你在宣告永續類別的屬性時限制在 Hibernate 預先訂好的內建對應型態，你會失去許多 Java 的豐富表現。而且，你的領域模型實作會緊緊和物理資料模型聯合，因為新的型態轉換是不可能的。

大部份 ORM 解決方案，那些我們已經看過的，提供對於使用者定義策略的支援來執行型態轉換。這些通常稱為轉換器。舉例而言，使用者可以對永續一個 JDK 型態 Integer 的屬性到一個 VARCHAR 欄位創建新的策略。Hibernate 提供了一個類別，更加強大的特性稱為自訂對應型態。

Hibernate 提供了兩個使用者和善的界面，讓應用程式在定義新的對應型態時可以使用。這些界面減少了定義自訂型態所牽涉的工作且隔離自訂型態以不會改變 Hibernate 的核心。這個允許你簡單的升級 Hibernate 且保持你已存在的自訂對應型態。你可以在 Hibernate 社群的網頁發現許多 Hibernate 對應型態有用的範例。

第一個程式設計界面是 `net.sf.hibernate.UserType.UserType` 適合於所有簡單的情形，且甚至適合許多更複雜的問題。讓我們在簡單的情況使用。

我們的 Bid 類別定義一個 amount 屬性；我們的 Item 類別定義了一個 initialPrice 屬性。兩者都是金融值。至此，我們只使用一個簡單的 BigDecimal 來表達這個值，對應 bigdecimal 到單一的 NUMERIC 欄位。

假設我們想要在拍賣應用程式支援多重貨幣且我們必須為此重構改變此已存在的領域模型（客戶驅動）。一個實作這個改變的方法是增加一個新的屬性到 Bid 與 Item：amountCurrency 和 initialPriceCurrency。我們接著對應這些新的屬性到額外的 VARCHAR 欄位且使用內建的 currency 對應型態。我們希望你永遠不要使用這種方法！

創建一個使用者型態

反之，我們應該創建一個 MonetaryAmount 的類別，其封裝了幣值與數量。注意這是領域模型的類別；它不會和任何 Hibernate 的界面相關：

```
public class MonetaryAmount implements Serializable {  
    private final BigDecimal value;  
    private final Currency currency;
```

```
public MonetaryAmount(BigDecimal value, Currency currency) {
    this.value = value;
    this.currency = currency;
}

public BigDecimal getValue() { return value; }

public Currency getCurrency() { return currency; }

public boolean equals(Object o) { ... }
public int hashCode() { ... }
}
```

我們已經使得 `MonetaryAmount` 是一個不可改變的類別。這是一個 Java 的好常規。注意我們已經實作了 `equals()` 跟 `hashCode()` 來完成這個類別（在這裡沒有特別的考量）。我們使用這個新的 `MonetaryAmount` 來取代 `Item` 中 `BigDecimal` 的 `initial-Price` 屬性。當然，我們可以，在我們的永續類中使用它來取代所有 `BigDecimal` 價錢（如同 `Bid.amount`），且甚至在商業邏輯之中（舉例而言，在帳務系統）。

讓我們對應這個重構過屬性的 `Item` 到資料庫。假設我們已經使用既有的資料庫包含了所有以 USD 計價的帳務，我們的應用程式不再限制到一個單一幣值（就是重構的重點），但是資料庫小組必須花時間來改變。當我們在永續化 `MonetaryAmount` 時我們需要轉換數量到 USD 而當我們載入物件時會轉換回 USD。

爲了這個，我們創建了一個 `MonetaryAmountUserType` 類別，它實作了 Hibernate 的 `UserType` 介面。我們自訂的對應型態，如 6.1 所列：

Listing 6.1 USD 財政數量的自訂對應型態

```
package auction.customtypes;

import ...;

public class MonetaryAmountUserType implements UserType {
    private static final int[] SQL_TYPES = {Types.NUMERIC};

    public int[] sqlTypes() { return SQL_TYPES; } ❶

    public Class returnedClass() { return MonetaryAmount.class; } ❷

    public boolean equals(Object x, Object y) { ❸
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public Object deepCopy(Object value) { return value; } ❹
}
```

```

public boolean isMutable() { return false; } ❸

public Object nullSafeGet(ResultSet resultSet, ❹
    String[] names,
    Object owner)
    throws HibernateException, SQLException {

    if (resultSet.isNull()) return null;
    BigDecimal valueInUSD = resultSet.getBigDecimal(names[0]);

    return new MonetaryAmount(valueInUSD, Currency.getInstance("USD"));
}

public void nullSafeSet(PreparedStatement statement, ❺
    Object value,
    int index)
    throws HibernateException, SQLException {
    if (value == null) {
        statement.setNull(index, Types.NUMERIC);
    } else {
        MonetaryAmount anyCurrency = (MonetaryAmount)value;
        MonetaryAmount amountInUSD =
            MonetaryAmount.convert( anyCurrency,
                Currency.getInstance("USD") );
        // The convert() method isn't shown in our examples
        statement.setBigDecimal(index, amountInUSD.getValue());
    }
}
}
}

```

- ❶ `sqlType()` 函數告訴 Hibernate 哪一種 SQL 欄位型態可用作 DDL 的綱要生成。型態編碼被定義在 `java.sql.Types`。注意這個函數傳回型態碼的陣列。一個 `UserType` 可以對應單一屬性到多重欄位，但是我們傳統的資料模型只有單一的 `NUMERIC`。
- ❷ `returnedClass()` 告訴 Hibernate 哪種 Java 型態是這個 `UserType` 所對應的。
- ❸ `UserType` 負責作屬性值的不良檢查。`equals()` 函數比較了現在的屬性值和之前的快照而決定屬性值是否為不良而必須儲存到資料庫。
- ❹ `UserType` 也部份負責在一開始創建快照。因為 `MonetaryAmount` 是一個不可改變類別，`deepCopy()` 函數傳回它的引數。在可改變型態的情形，傳回一個引數的複製來用作快照值需要的。這個函數當一個型別的實例被二階快取讀寫時也會被呼叫。

- ⑤ 像這樣不可改變的型別會使 Hibernate 可以得到一些較少的效能最佳化。isMutable() 函數告訴 Hibernate 這個型態是不可改變的。
- ⑥ nullSafeGet() 函數由 JDBC 的 ResultSet 中取出屬性值。你也可以存取元件的 owner 如果你需要它以作轉換。所有的資料庫值都是以 USD 為單位，所以在你展示給使用者之前你必須用這個函數的回傳值來轉換 MonetaryAmount。
- ⑦ nullSafeSet() 函數將屬性值寫入 JDBC 的 PreparedStatement。這個函數拿到幣值的任何設定，並且在儲存前將它轉換到簡單的 BigDecimal 的 USD 值。

我們現在對應 Item 的 initialPrice 屬性如下：

```
<property name="initialPrice"
          column="INITIAL_PRICE"
          type="auction.customtypes.MonetaryAmountUserType" />
```

這是最簡單 UserType 能夠執行的轉換。更多複雜精細的事情是有可能的。一個自訂型態可以自行確認；它可以由 LDAP 的目錄中讀寫資料；它甚至可以由另外資料庫的 Hibernate Session 中讀取永續的物件。你主要只受到想像力的限制！

我們偏好使用數量和幣值來表現資料庫的財政數量，特別如果綱要不是既有但可以定義（或很快的更新）。我們仍可以使用 UserType，但接著我們就不能在物件查詢中使用數量（或幣值）。Hibernate 查詢引擎（在下一章討論更仔細）不會知道任何單獨 MonetaryAmount 屬性。你可在 Java 程式中存取屬性（畢竟 MonetaryAmount 只是一個領域模型中一般的類別），而不是在 Hibernate 的查詢之中。

反而，如果我們需要 Hibernate 查詢的完全威力，我們應該使用 CompositeUserType。這個（有一點點複雜）的界面將我們 MonetaryAmount 的屬性接觸到 Hibernate。

創建一個 CompositeUserType

爲了展示自訂對應型態，我們不會改變任何 MonetaryAmount 類別（和其它的永續類別）—我們只改變自訂對應型態，如 6.2 所列

Listing 6.2 在新資料庫綱要下財政數量的自訂型態

```
package auction.customtypes;

import ...;
```

```
public class MonetaryAmountCompositeUserType
    implements CompositeUserType {

    public Class returnedClass() { return MonetaryAmount.class; }

    public boolean equals(Object x, Object y) {
        if (x == y) return true;
        if (x == null || y == null) return false;
        return x.equals(y);
    }

    public Object deepCopy(Object value) {
        return value; // MonetaryAmount is immutable
    }

    public boolean isMutable() { return false; }

    public Object nullSafeGet(ResultSet resultSet,
        String[] names,
        SessionImplementor session,
        Object owner)
        throws HibernateException, SQLException {

        if (resultSet.isNull()) return null;
        BigDecimal value = resultSet.getBigDecimal( names[0] );
        Currency currency =
            Currency.getInstance(resultSet.getString( names[1] ));
        return new MonetaryAmount(value, currency);
    }

    public void nullSafeSet(PreparedStatement statement,
        Object value,
        int index,
        SessionImplementor session)
        throws HibernateException, SQLException {

        if (value==null) {
            statement.setNull(index, Types.NUMERIC);
            statement.setNull(index+1, Types.VARCHAR);
        } else {
            MonetaryAmount amount = (MonetaryAmount) value;
            String currencyCode =
                amount.getCurrency().getCurrencyCode();
            statement.setBigDecimal( index, amount.getValue() );
            statement.setString( index+1, currencyCode );
        }
    }

    public String[] getPropertyNames() { ❶
        return new String[] { "value", "currency" };
    }

    public Type[] getPropertyTypes() { ❷
```

```
        return new Type[] { Hibernate.BIG_DECIMAL, Hibernate.CURRENCY };
    }

    public Object getPropertyValue(Object component, 3
        int property)
        throws HibernateException {
        MonetaryAmount MonetaryAmount = (MonetaryAmount) component;
        if (property == 0)
            return MonetaryAmount.getValue();
        else
            return MonetaryAmount.getCurrency();
    }

    public void setPropertyValue(Object component, 4
        int property,
        Object value) throws HibernateException {
        throw new UnsupportedOperationException("Immutable!");
    }

    public Object assemble(Serializable cached, 5
        SessionImplementor session,
        Object owner)
        throws HibernateException {
        return cached;
    }

    public Serializable disassemble(Object value, 6
        SessionImplementor session)
        throws HibernateException {
        return (Serializable) value;
    }
}
}
```

- 1** CompositeUserType 有它自己的屬性，在 `getPropertyName()` 定義。
- 2** 每個屬性有他們自己的型態，在 `getPropertyType()` 定義。
- 3** `getPropertyValue()` 函數傳回一個 `MonetaryAmount` 單獨屬性的值。
- 4** 既然 `MonetaryAmount` 是不可變的，我們不能單獨設定屬性值（沒問題，這個函數是選擇性的）。
- 5** `assemble()` 函數當一實例由二階快取讀出時會被呼叫。
- 6** `disassemble()` 函數當這個型態的實例被寫入二階快取時被呼叫。

屬性的順序在 `getPropertyNames()`、`getPropertyTypes()` 和 `getPropertyValues()` 函數中必需相同。`initialPrice` 屬性現在對應到兩個欄位，所以我們在對應文件中宣告兩者。第一個欄位儲存值；第二個儲存 `MonetaryAmount` 的幣值（欄位的順序必須配合你的型態實作屬性的順序）：

```
<property name="initialPrice"
          type="auction.customtypes.MonetaryAmountCompositeUserType">
  <column name="INITIAL_PRICE" />
  <column name="INITIAL_PRICE_CURRENCY" />
</property>
```

在一個查詢中，我們現在可以提到自訂型態中的 `amount` 與 `currency` 屬性，甚至他們沒有以個別的屬性出現在對應文件中的任何地方。

```
from Item i
where i.initialPrice.value > 100.0
      and i.initialPrice.currency = 'AUD'
```

我們已經使用我們自訂複合型態來延伸了在 Java 物件模型與 SQL 資料庫綱要之中的緩衝區。兩者表達現在都可以更健全地處理改變。

如果實作自訂型態似乎是複雜的，放輕鬆；你很少需要使用自訂對應型態。另一種方式來表達 `MonetaryAmount` 類別是使用元件對應，如 3.5.2 節所示，“使用元件”。決定使用自訂對應型態通常是品味的問題。

讓我們看一個極端重要，自訂對應型態的應用程式。型態安全細目的設計樣式幾乎可在所有企業應用程式中找到。

使用細目型態

細目型態是一個常見的 Java 成語，指的是一個類別有一固定（少量）數量不可改變的實例。

舉例而言，`Comment` 類別（使用者在 `CaveatEmptor` 中給定其它使用者的意見）定義了一個 `rating`。在我們現在的模型，我們有簡單的 `int` 屬性。一個型態安全（且更好）的方式是實作不同的等級（畢竟，我們可能不希望任何的整數值）來創建 `Rating` 類別如下：

```
package auction;

public class Rating implements Serializable {
    private String name;
```

```
public static final Rating EXCELLENT = new Rating("Excellent");
public static final Rating OK = new Rating("OK");
public static final Rating LOW = new Rating("Low");
private static final Map INSTANCES = new HashMap();

static {
    INSTANCES.put(EXCELLENT.toString(), EXCELLENT);
    INSTANCES.put(OK.toString(), OK);
    INSTANCES.put(LOW.toString(), LOW);
}

private Rating(String name) {
    this.name=name;
}

public String toString() {
    return name;
}

Object readResolve() {
    return getInstance(name);
}

public static Rating getInstance(String name) {
    return (Rating) INSTANCES.get(name);
}
}
```

我們接著將 Comment 類別中的 rating 屬性改到新的型態。在資料庫，等級會被表達成 VARCHAR 值。爲了 Rating 值屬性創建一個 UserType 是直觀的：

```
package auction.customtypes;

import ...;
public class RatingUserType implements UserType {

    private static final int[] SQL_TYPES = {Types.VARCHAR};

    public int[] sqlTypes() { return SQL_TYPES; }
    public Class returnedClass() { return Rating.class; }
    public boolean equals(Object x, Object y) { return x == y; }
    public Object deepCopy(Object value) { return value; }
    public boolean isMutable() { return false; }

    public Object nullSafeGet(ResultSet resultSet,
                              String[] names,
                              Object owner)
        throws HibernateException, SQLException {

        String name = resultSet.getString(names[0]);
        return resultSet.isNull() ? null : Rating.getInstance(name);
    }

    public void nullSafeSet(PreparedStatement statement,
                            Object value,
```

```
        int index)
        throws HibernateException, SQLException {
    if (value == null) {
        statement.setNull(index, Types.VARCHAR);
    } else {
        statement.setString(index, value.toString());
    }
}
}
```

這段碼如同之前實作 `UserType` 一樣簡單。實作 `nullSafeGet()` 和 `nullSafeSet()` 再一次是最有趣的部份，包含了邏輯的轉換。

你可能會碰到的問題是在 Hibernate 的查詢中使用細目。考慮下列 Hibernate 的查詢取回了所有被標示為 "Low" 的意見：

```
Query q =
    session.createQuery("from Comment c where c.rating = Rating.LOW");
```

這個查詢不會運作，因為 Hibernate 不知道如何操作 `Rating.LOW` 而會試著當字串運用。我們需要使用 `bind` 參數且動態設定比較的等級（這是我們大部份時間需要的另一個原因）：

```
Query q =
    session.createQuery("from Comment c where c.rating = :rating");
q.setParameter("rating",
    Rating.LOW,
    Hibernate.custom(RatingUserType.class));
```

這個範例的最後一行使用靜態輔助函數 `Hibernate.custom()` 來轉換自訂型態到 Hibernate 的 `Type`，一個簡單的方式告訴 Hibernate 關於你的細目對應和如何處理 `Rating.LOW` 值。

如果在你的應用程式中許多地方使用細目型態，你可能想要用這個 `UserType` 的範例且讓它更為一般。JDK1.5 介紹了新的語言特性來定義細目型態，且我們建議使用自訂對應型態直到 Hibernate 原生支援 JDK1.5 的特性。（注意 Hibernate2 的 `PersistentEnum` 不被贊成使用且不該被使用）。

我們已經討論了各種 Hibernate 對應型態：內建對應型態，使用者自訂客製型態和甚至元件（第三章）。他們都被視為值型態，因為他們對應物件的值型態（非個體）到資料庫。我們現在已經準備好瀏覽值型態實例的 `collection`。

6.2 對應值型態的 collection

你已經在第三章的個體關聯中看過 collection。在這一節，我們討論 collection 包含了值型態的實例，包含了元件的 collection。沿著這條路，你會看到一些 Hibernate 的 collection 對應更進階的特性，也可以用在表達個體聯合的 collection 上，如這章之後所討論的。

6.2.1 Sets, bags, lists 和 maps

假設我們的賣方可以將影像連接到 Item，一個影像只能由包含它的項目來存取；支援我們系統中其它任何個體的聯合不是必須的。在這個情形，模型這個影像變成值型態並不會不合理。Item 會有影像的 collection 而 Hibernate 會考慮這是 Item 的一部份，而沒有它自己的生命週期。

我們會逐一使用 Hibernate 實作這種行為的幾種方式。現在，讓我們假設影像是被儲存在其它檔案系統的地方而我們只想在資料庫中存放檔名。影像是如何被儲存和載入不被討論。

使用 set

最簡單的實作方式是使用 String 檔名的 Set。我們增加一個 Item 類別的 collection 屬性：

```
private Set images = new HashSet();
...
public Set getImages() {
    return this.images;
}

public void setImages(Set images) {
    this.images = images;
}
```

我們在 Item 中使用下列的對應：

```
<set name="images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <element type="string" column="FILENAME" not-null="true"/>
</set>
```

影像檔名儲存在名為 ITEM_IMAGE 的表格。從資料庫的觀點，這個表格是和 ITEM 表格分離的；但是 Hibernate 對我們隱藏這個事實，創建了這是一個單一個體的幻

象。<key> 元素宣告了外部鍵，ITEM_ID 是父個體。<element> 標籤宣告這個 collection 是值型態實例的 collection：在這個情形，是字串。

Set 不能包含重覆的元素，所以 ITEM_IMAGE 表格的主鍵值包含了在 <set> 宣告中的兩個欄位：ITEM_ID 和 FILENAME。看表 6.1 表格綱要範例的例子。

似乎不像我們允許使用者超過一次連接相同的影像，但是假設我們如此作，哪種對應是適當的？

使用 bag

一個沒有次序的 collection 允許重複的元件稱為 bag。奇怪地，Java 的 Collections 框架並沒有定義 Bag 界面。Hibernate 讓你在 Java 中使用 List 來模擬 bag 的行為。這個和 Java 社群一般的使用方式是一致的。注意，然而 List 特定是有順序的 collection；當使用 bag 的語義永續一個 List 時，Hibernate 不會保守順序。要使用 bag，改變型態 Item 中 images 的型態從 Set 到 List，或許使用 ArrayList 作為實作（你也可以使用 Collection 作為這個屬性的型態）。

從前一節中改變表格定義以允許重複的 FILENAME 需要另一個主鍵值。<idbag> 對應允許我們連接一個輔助鍵欄位到 collection 表格。非常像是在個體類別中所使用的綜合辨識元：

```
<idbag name="images" lazy="true" table="ITEM_IMAGE">
  <collection-id type="long" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </collection-id>
  <key column="ITEM_ID"/>
  <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

在這個情形，主鍵值是產生的 ITEM_IMAGE_ID。你可以在圖 6.2 看到資料庫表格的圖形表現。

ITEM		ITEM_IMAGE	
ITEM_ID	NAME	ITEM_ID	FILENAME
1	Foo	1	foimage1.jpg
2	Bar	1	foimage2.jpg
3	Baz	2	barimage1.jpg

Figure 6.1
表格結構和字串 collection
的範例資料

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	Foo	1	1	fooimage1.jpg
2	Bar	2	1	fooimage1.jpg
3	Baz	3	2	barimage1.jpg

Figure 6.2
使用輔助主鍵值的 bag 的
表格結構

你可能會懷疑如果已經有了 `<bag>` 對應，為什麼 Hibernate 對應是 `<idbag>`。你將會知道更多有關 bag，但一個更像的情境牽涉到保守影像連結到 Item 的順序。有很多好的方法來達到這個；一個方式是使用真的 list 而不是使用到 bag。

使用 list

一個 `<list>` 的對應需要資料庫中額外的索引欄位。索引欄位定義了 collection 中元素的位子。因此，如果我們使用 `<list>` 來對應 collection，當由資料庫中取出 collection 時，Hibernate 可以保持 collection 元素的順序：

```
<list name="images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="POSITION"/>
  <element type="string" column="FILENAME" not-null="true"/>
</list>
```

主鍵值是由 ITEM_ID 與 POSITION 欄位所組成。注意 (FILENAME) 重複的元素是可允許的。這和 list 的語義是一致的。(我們不需要改變 Item 的類別，型態和我們之前所使用的 bag 是一樣的。)

如果這個 collection 是 [fooimage1.jpg, fooimage1.jpg, fooimage2.jpg]，這個 POSITION 欄位包含值 0, 1 和 2，如同圖 6.3 所示。

另外，我們可以使用 Java 的陣列而不是一個 list。Hibernate 支援這種方式。事實上，陣列對應的細節差不多和 list 是一致的。然而，我們非常強烈建議不要

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	POSITION	FILENAME
1	Foo	1	0	fooimage1.jpg
2	Bar	1	1	fooimage1.jpg
3	Baz	1	2	fooimage2.jpg

Figure 6.3
有位置元素的 list 的表格

使用陣列，因為陣列不能被延後初始化（沒有在虛擬機器的等級來代理一個陣列的方法）。

現在，假設我們的影像除了檔名之外，有使用者輸入的名字。在 Java 中模型這個的方式是使用 Map，用名字作為鍵值而用檔名作為值。

使用 map

對應一個 `<map>` 和對應一個 `list` 是類似的：

```
<map name="images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="string" column="FILENAME" not-null="true"/>
</map>
```

主鍵值是由 `ITEM_ID` 和 `IMAGE_NAME` 欄位所組成。`IMAGE_NAME` 欄位儲存 `map` 中的鍵值，重複元素是允許的；看圖 6.4 以得到表格的圖形化表現。

`Map` 是沒有順序的。如果我們總是想用影像的名稱來作排序時該如何呢？

排列和有順序的 collection

英文中另人吃驚的濫用，當在 Hibernate 的永續 `collection` 中，排列 (`sorted`) 與有順序 (`ordered`) 是有不同意義的。一個排列的 `collection` 是使用 Java 的比較子在記憶體中排列的。一個有順序的 `collection` 是在資料庫的階層使用 SQL 的 `order by` 子句來查詢。

讓我們讓影像的 `map` 變成有排序的 `map`。這是一個對應文件的簡易改變：

```
<map name="images"
  lazy="true"
  table="ITEM_IMAGE"
  sort="natural">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="string" column="FILENAME" not-null="true"/>
</map>
```

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	IMAGE_NAME	FILENAME
1	Foo	1	Foo Image 1	foimage1.jpg
2	Bar	1	Foo Image One	foimage1.jpg
3	Baz	1	Foo Image 2	foimage2.jpg

Figure 6.4
使用字串作為索引和元素的 `map` 表格

指定 `sort="natural"`，我們告訴 Hibernate 使用 `SortedMap`，依照 `compareTo()` 函數將 `java.lang.String` 作排列。如果你想要其它的排列順序—舉例來說，反字母排列—你可以在 `sort` 屬性中指定實作 `java.util.Comparator` 類別的名稱。舉例來說：

```
<map name="images"
    lazy="true"
    table="ITEM_IMAGE"
    sort="auction.util.comparator.ReverseStringComparator">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Hibernate 有排序的 `map` 行為與 `java.util.TreeMap` 一致。一個排序的 `set` (行為像是 `java.util.TreeSet`) 是用相似的方式對應：

```
<set name="images"
    lazy="true"
    table="ITEM_IMAGE"
    sort="natural">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

`Bag` 不能被排列 (很不幸的沒有 `TreeBag`)，`lists` 也沒有；`list` 元素的順序是由 `list` 的索引所定義。

另外，你可以選擇使用有順序的 `map`，使用資料庫的排列能力而不是使用 (或許較沒有效率的) 記憶體中的排列：

```
<map name="images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

`order-by` 的敘述屬性是 SQL 中 `order by` 子句的一部份。在這個情形，我們依照 `IMAGE_NAME` 欄位作排序，以一個遞增的順序。你甚至可以在 `order-by` 的屬性中寫 SQL 的函數呼叫。

```
<map name="images"
      lazy="true"
      table="ITEM_IMAGE"
      order-by="lower(FILENAME) asc">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="string" column="FILENAME" not-null="true"/>
</map>
```

注意你可以用 collection 表格中的任何欄位來排序。set 和 bag 兩者都允許 order-by 的屬性。但是 list 不允許。這個範例使用了 bag：

```
<idbag name="images"
        lazy="true"
        table="ITEM_IMAGE"
        order-by="ITEM_IMAGE_ID desc">
  <collection-id type="long" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </collection-id>
  <key column="ITEM_ID"/>
  <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

在表面之下，Hibernate 使用 LinkedHashSet 跟 LinkedHashMap 來實作有順序的 set 跟 map，所以這個功能只在 JDK1.4 或之後出現。有次序的 bag 有可能可在所有的 JDK 版本出現。

在一個真實系統，像是我們會需要保存不止影像名稱和檔案名稱；我們或許需要爲了額外的資訊創建一個 Image 類別。我們可以對應 Image 成一個個體類別；但因爲我們已經決定了，這不是絕對必須的。讓我們看不使用 Image 的個體，我們最多可以得到什麼（這可能需要一個聯合對應和更複雜的生命周期管理）。在第三章，你看到 Hibernate 讓你對應使用者自訂類別成元件，這被視爲值型態。這就算當元件實例是 collection 元件時依然爲真。

元件的 collection

我們的 Image 定義了 name, filename, sizeX 跟 sizeY 的屬性。它有單一聯合，連到它的父 Item 類別，如圖 6.5 所示。

如同你可以從聚集聯合型態可以看到的（黑色的菱形），Image 是 Item 的一個元件，且 Item 負責管理 Image 的生命周期。到影像的參照不是共享的，所以我們

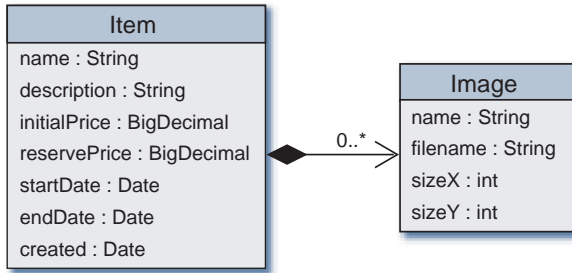


Figure 6.5
Item 中 Image 元件的 collection

第一個選擇是 Hibernate 的元件對應。聯合的多樣性更進一步的宣告了這個聯合是一個多值的，那就是說，同一個 Item 使用許多（或零）個 Images。

撰寫元件類別

首先，我們實作了 Image 類別，這只是單純的 POJO，沒有什麼特別要考量的。如同你在第三章看到的，元件類別沒有辨識屬性。然而，我們必須實作 equals()（與 hashCode()）來比較 name, filename, sizeX 和 sizeY 屬性，以允許 Hibernate 的不良檢查可以正確的運作。嚴格來說，實作 equals() 跟 hashCode() 並不是所有元件都必須的。然而，我們對任何元件類別都建議它因為實作是直觀的且 " 安全比說抱歉好 " 是好的格言。

Item 類別還沒有改變：它仍然有影像的 Set。當然，在這個 collection 的物件不再是 String，讓我們將這對應到資料庫。

對應 collection

元件的 Collection 對應類似於其它值型態的 collection。唯一的不同是使用 <composite-element> 而不是熟悉的 <element> 標籤。一個有順序的影像可以如下對應：

```

<set name="images"
  lazy="true"
  table="ITEM_IMAGE"
  order-by="IMAGE_NAME asc">
  <key column="ITEM_ID" />
  <composite-element class="Image">
    <property name="name" column="IMAGE_NAME" not-null="true"/>
    <property name="filename" column="FILENAME" not-null="true"/>
    <property name="sizeX" column="SIZE_X" not-null="true"/>
  
```

```

        <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>

```

這是 set，所以主鍵值是由鍵值欄位和所有元件欄位組成：ITEM_ID, IMAGE_NAME, FILENAME, SIZEX 跟 SIZEY。因為這些欄位都出現在主鍵值，我們宣告它們是 not-null="true"。(這對特別的對應很清楚是個缺點)。

雙向走訪

從 Item 到 Image 的聯合是單方向的。如果 Image 類別也宣告了 item 的屬性，保持了一個參照回到擁有者 Item，我們會增加 <parent> 標籤到對應：

```

<set name="images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <composite-element class="Image">
        <parent name="item"/>
        <property name="name" column="IMAGE_NAME" not-null="true"/>
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZEX" not-null="true"/>
        <property name="sizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>

```

然而，真正的雙向走訪是不可能的。你不可能單獨取到一個 Image 然後走訪到它的父 Item。這是一個重要的議題：你將會能夠靠著查詢來載入 Image 的實例，但是元件，像是所有的值型態，是用值取出的。Image 物件不會有參照到父親（屬性值是 null）。如果你需要這種功能，你應該使用完整的父 / 子個體聯合，如同在第 3 章所描述的。

再來，宣告所有的屬性成 not-null 是一些你需要避免的事。我們對於 IMAGE 表格需要一個較好的鍵值。

避免 not-null 欄位

如果一個 Image 的 set 不是我們需要的，其它的 collection 方式是有可能的。舉例來說，<idbag> 提供了輔助 collection 鍵：

```

<idbag name="images"
    lazy="true"
    table="ITEM_IMAGE"

```

```

        order-by="IMAGE_NAME asc">
<collection-id type="long" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
</collection-id>
<key column="ITEM_ID"/>
<composite-element class="Image">
    <property name="name" column="IMAGE_NAME"/>
    <property name="filename" column="FILENAME" not-null="true"/>
    <property name="sizeX" column="SIZE_X"/>
    <property name="sizeY" column="SIZE_Y"/>
</composite-element>
</idbag>

```

這一次，主鍵是 ITEM_IMAGE_ID 欄位，我們實作 equals() 跟 hashCode() 不是重要的（至少，Hibernate 不要求它）。我們也不需要宣告屬性值是 not-null="true"。他們在 idbag 的情形是可以為空的，如圖 6.6 所示

我們需要指出這個 bag 對應和標準的父 / 子個體關聯並沒有重要的不同。表格是一樣的，甚至 Java 程式碼也非常類似；選擇只是喜好的不同。當然，一個父 / 子聯合支援子個體的分享參照且是一個真正的雙向走訪。

我們甚至可以在 Image 類別中移除 name 屬性且再一次使用影像名稱作為 map 的鍵值：

```

<map name="images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
    <key column="ITEM_ID"/>
    <index type="string" column="IMAGE_NAME"/>
    <composite-element class="Image">
        <property name="filename" column="FILENAME" not-null="true"/>
        <property name="sizeX" column="SIZE_X"/>
        <property name="sizeY" column="SIZE_Y"/>
    </composite-element>
</map>

```

如同之前，主鍵值是由 ITEM_ID 跟 IMAGE_ID 合成的。

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGE_NAME	FILENAME
1	1	Foo Image 1	fooimage1.jpg
2	1	Foo Image 1	fooimage1.jpg
3	2	Bar Image 1	barimage1.jpg

Figure 6.6
Image 元件的 Collection
使用一個 bag 與輔助鍵值

一個合成元件類別如 `Image` 不限制到像 `filename` 簡單屬性的基本型態。它可以包含元件，使用 `<nested-composite-element>` 宣告，且甚至 `<many-to-one>` 聯合到個體。然而它可以不擁有 `collection`。一個有多對一聯合的合成元件是有用的，且我們會在這章之後回到這種對應。

我們終於完成了值型態，我們會繼續個體聯合對應的技巧。我們在第三章對應簡單的父 / 子聯合只是許多可能的聯合對應型態之一。他們大部份被視為特例的且實務中很少使用。

6.3 對應個體聯合

當我們使用聯合這個字，總是表示個體之間的關係。在第 3 章，我們展示了單向多對一聯合，使它雙向，且最後轉到父 / 子關係（一對多與多對一）。

一對多聯合簡單而言是最重要的聯合。事實上，我們到此不鼓勵當一個簡單的雙向多對一 / 一對多能運作時，使用更奇特的聯合形式。特別來說，多對多聯合總是可以由二個多對一聯合到一個中介類別。這個模形通常更簡單的延伸，所以我們不傾向在我們的應用程式使用多對多聯合。

依靠這個拒絕，讓我們由一對一聯合開始調查 `Hibernate` 的豐富對應。

6.3.1 一對一聯合

我們在第三章中討論 `User` 和 `Address` 的關係（使用者有 `billingAddress` 跟 `homeAddress`）是使用 `<component>` 對應最好的表達。這通常是最簡單的方式來表達一對一關聯，因為一個類別的生命周期幾乎總是依靠著另一個類別的生命周期，且聯合是一個合成。

但是如果我們想要用一個獨佔的 `Address` 表格且對應 `User` 跟 `Address` 兩者都是個體時該如何呢？接著，這個類別有一個真正的一對一聯合。在這個情形，我們由下列的 `Address` 對應開始：

```
<class name="Address" table="ADDRESS">
  <id name="id" column="ADDRESS_ID">
    <generator class="native"/>
  </id>
  <property name="street"/>
  <property name="city"/>
  <property name="zipcode"/>
</class>
```

注意 Address 現在需要辨識屬性；它再也不是元件類別。有兩種不同的方式在 Hibernate 之中表達 Address 一對一聯合。第一個方式是增加一個外部鍵欄位到 USER 表格。

使用外部鍵值聯合

最簡單來表達從 User 到 billingAddress 的聯合是使用 <many-to-one> 的對應與獨一限制的外部鍵。這可能令你驚訝，因為 many 看起來不像是一個好的描述，不管是在一對一聯合的哪一端！然而，從 Hibernate 的觀點，這兩種外部鍵聯合並沒有太多的不同。所以，我們增加一個外部鍵欄位叫作 BILLING_ADDRESS_ID 到 USER 表格且如下的對應：

```
<many-to-one name="billingAddress"
  class="Address"
  column="BILLING_ADDRESS_ID"
  cascade="save-update" />
```

注意我們已經選擇了 save-update 成串聯型式。這指的是當我們從一個永續的 User 創建了聯合，Address 會變成永續。或許，cascade="all" 對這個聯合較有意義，因為刪除 User 應該會導致 Address 的刪除。（記住 Address 現在有它自己的個體生命週期。）

我們的資料庫綱要仍然允許 USER 表格中 BILLING_ADDRESS_ID 欄位重複的值，所以使用者可以參照到同一個位址。為了讓這個聯合是真的一對一，我們增加了 unique="true" 到 <many-to-one> 的元素，創建了關聯模型所以一個使用者只能有一個地址：

```
<many-to-one name="billingAddress"
  class="Address"
  column="BILLING_ADDRESS_ID"
  cascade="all"
  unique="true" />
```

這個改變增加了一個限制到 Hibernate 在 DDL 生成 BILLING_ADDRESS_ID 欄位時，導致如 6.7 的表格結構。

但是如果我們想要能在 Java 中從 Address 到 User 的走訪時該如何？從第三章，你知道如何將它轉到雙向一對多的 collection，但是我們決定每個 Address 只有一個 User，所以這不是對的解決方案。我們不需要在 Address 類別內有使用者的

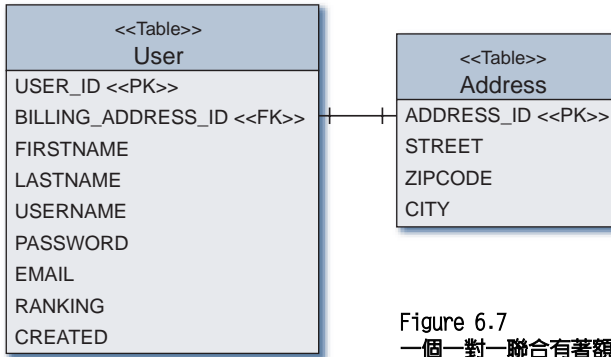


Figure 6.7
一個一對一聯合有著額外的外部鍵欄位

collection。反之，我們增加一個名為 `user` 的屬性（型態是 `User`）到 `Address` 類別，且如同對應 `Address` 般的對應它：

```

<one-to-one name="user"
  class="User"
  property-ref="billingAddress"/>
  
```

這個對應告訴 Hibernate `Address` 中的 `user` 聯合是反方向中 `User` 的 `billingAddress` 聯合。

在程式碼中，我們創建了二個物件的聯合如下：

```

Address address = new Address();
address.setStreet("646 Toorak Rd");
address.setCity("Toorak");
address.setZipcode("3000");

Transaction tx = session.beginTransaction();
User user = (User) session.get(User.class, userId);
address.setUser(user);
user.setBillingAddress(address);
tx.commit();
  
```

爲了完成對應，我們需要對應 `User` 的 `homeAddress` 屬性。這是夠簡單的：我們增加了另外的 `<many-to-one>` 元素到 `User` 的後設資料中，對應一個新的外部鍵欄位，`HOME_ADDRESS_ID`：

```

<many-to-one name="homeAddress"
  class="Address"
  column="HOME_ADDRESS_ID"
  cascade="save-update"
  unique="true"/>
  
```

User 表格現在定義兩個外部鍵參照 ADDRESS 表格的主鍵值：HOME_ADDRESS_ID 跟 BILLING_ADDRESS_ID。

不幸的是，我們不能讓 billingAddress 跟 homeAddress 兩者的聯合雙向化，因為我們不知道特別的地址是否為帳單地址或住家地址。（我們不能決定是哪一個屬性名稱— billingAddress 或 homeAddress，來使用 user 屬性中對應的 property-ref 屬性。）我們可以試著讓 Address 為抽象類別，且有 HomeAddress 與 BillingAddress 子類別且對應聯合到子類別。這個方式會運作，但它很複雜且可能在這個例子並不合理。

我們的建議是在兩個類別之中避免定義超過一個一對一聯合。如果你必須，將這些聯合設成單向。如果你沒有超過一個，也就是對每個 User 而言只有一個 Address 實例，有另一種方式我們剛才展示的方式。在定義 User 表格的外部鍵欄位之外，你可以使用主鍵值聯合。

使用主鍵值聯合

兩個表格用主鍵值聯合的關係共享同一個主鍵值。一個表格的主鍵值同時也是另一個表格的外部鍵。這種方式主要的困難是確保當物件在儲存時，聯合實例是被給定同一個主鍵值。在我們試著解決這個問題前，讓我們看看如何對應主鍵值聯合。

對於一個主鍵值聯合，兩端的聯合都使用 `<one-to-one>` 宣告來對應。這也指出我們不能對應帳單和家裡的地址兩者，只用一個屬性。每個 USER 表格中的一列有一個對應的列在 ADDRESS 表格之中。兩個地址會需要一個額外的表格，且這種對應因此不是恰當的。讓我們呼叫 address 的單一地址屬性且對應它到 User：

```
<one-to-one name="address"
  class="Address"
  cascade="save-update" />
```

接下來，這是 Address 中的 user：

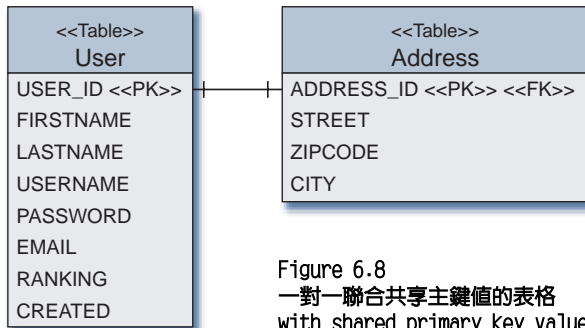
```
<one-to-one name="user"
  class="User"
  constrained="true" />
```

這裡最有趣的事是 `constrained="true"` 的用法。它告訴 Hibernate 有一個外部鍵限制在 ADDRESS 的主鍵值會參考 USER 的主鍵值。

現在我們必須確保新儲存的 Address 實例被給定和他們的 User 相同的辨識元。我們使用一個特別的 Hibernate 辨識元產生策略稱為 foreign：

```
<class name="Address" table="ADDRESS">
  <id name="id" column="ADDRESS_ID">
    <generator class="foreign">
      <param name="property">user</param>
    </generator>
  </id>
  ...
  <one-to-one name="user"
    class="User"
    constrained="true"/>
</class>
```

foreign 產生器中 <param> 中名稱爲 property 的允許我們提出 Address 類別中一個一對一的聯合，在這個情形，是 user 聯合。foreign 產生器檢查聯合物件 (User) 且使用它的辨識元爲新的 Address 的辨識元。可以看圖 6.8 的表格結構。



創建物件聯合的程式碼對主鍵值聯合是沒有改變的；和我們之前多對一對應型態的是相同程式碼。

還有剩下一個個體聯合多樣性我們還沒有討論到的：多對多。

6.3.2 多對多聯合

Category 和 Item 的聯合是一個多對多聯合，如你在圖 6.9 所看到的。

在一個真實系統，我們可以不使用多對多聯合。在我們的經驗，總是幾乎有其它的資訊必須夾帶在兩個聯合實例的連結（舉例來說，當一個項目被設定在類別中需要日期和時間），而最好表達這個資訊的方式是由一個中介聯合類別。在

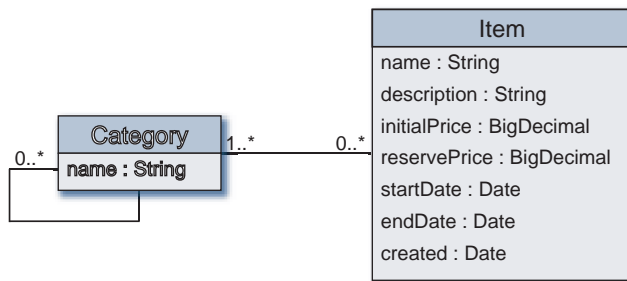


Figure 6.9
在 Category 與 Item 中多對多的值聯合

Hibernate，我們可以對應聯合類別成一個個體而對每一邊使用兩個一對多聯合。或許更便利地，我們可以使用一個合成元素類別，一個我們之後會展示的技巧。

不過，實作一個真實多對對個體聯合是這一節的目的地。讓我們從單向的例子開始。

一個單方向多對多聯合

如果你只需要單向的走訪，對應是直觀的。單向多對多聯合不會比我們之前所包含的值型態實例的 collection 來得複雜。舉例來說，如果 Category 有一個 Item 的 set，我們可以使用這個對應：

```

<set name="items"
    table="CATEGORY_ITEM"
    lazy="true"
    cascade="save-update">
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</set>
  
```

如同值型態實例的 collection，多對多聯合有它自己的表格，就是連結表格或聯合表格。在這個情形，連結表格有兩個欄位：CATEGORY 與 ITEM 表格的外部鍵值。主鍵值是由欄位兩者所合成。主表格的結構如圖 6.10 所示。

我們也可以使用 bag 與分別的主鍵值欄位：

```

<idbag name="items"
    table="CATEGORY_ITEM"
    lazy="true"
    cascade="save-update">
    <collection-id type="long" column="CATEGORY_ITEM_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="CATEGORY_ID"/>
    <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
  
```

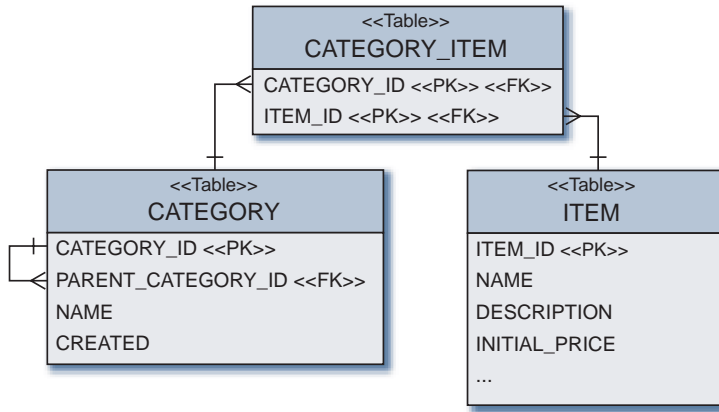


Figure 6.10
多對多個體聯合對應到一個聯合表格

如同使用 `<idbag>` 對應，主鍵值是輔助鍵欄位，`CATEGORY_ITEM_ID`。複製連結因此是允許的；同一個 `Item` 可以增加兩次到某一特定的 `Category`。（這似乎不是一個很有用的特性。）

我們甚至可以使用索引 `collection`（一個 `map` 或 `list`）。下列的範例使用 `list`：

```

<list name="items"
      table="CATEGORY_ITEM"
      lazy="true"
      cascade="save-update">
  <key column="CATEGORY_ID"/>
  <index column="DISPLAY_POSITION"/>
  <many-to-many class="Item" column="ITEM_ID"/>
</list>

```

主鍵值是由 `CATEGORY_ID` 和 `DISPLAY_POSITION` 欄位所組成，這個對應保證每一個 `Item` 知道它在 `Category` 中的位子。

創建一個物件聯合是簡單的：

```

Transaction tx = session.beginTransaction();
Category cat = (Category) session.get(Category.class, categoryId);
Item item = (Item) session.get(Item.class, itemId);

cat.getItems().add(item);

tx.commit();

```

雙向多對多聯合有一點點更困難

一個雙向多對多聯合

當我們在第三章 (3.7 節, " 介紹聯合 ") 對應到雙向一對多聯合, 我們已解釋過為什麼一端的聯合必須要用 `inverse="true"` 來對應。我們建議你現在能複習解釋。

同樣的原理套用到雙向多對多聯合: 每一個連結表格的列是由兩個 `collection` 元素所表達, 是聯合兩個的每一個元素。一個 `Item` 跟 `Category` 的聯合是表現在記憶體中, 是由 `Category` 中 `item` 的 `collection` 中的 `Item` 實例, 但同時也是由 `Item` 中 `categories` 的 `collection` 中的 `Category` 實例。

在我們討論雙向對應的例子之前, 你必須知道創建物件聯合的程式碼也改變了:

```
cat.getItems.add(item);
item.getCategories().add(category);
```

如同長久以來, 一個雙向聯合 (不管是怎樣的多樣性) 需要你設定聯合的兩個端點。

當你對應一個雙向多對多聯合, 你必須宣告聯合中的一個端點是 `inverse="true"` 來定義哪一端的狀態是用來更新連結表格的。你可以自己選擇應該是哪一端。

回想在前一節 `items` 的 `collection` 對應:

```
<class name="Category" table="CATEGORY">
  ... <
  <set name="items"
    table="CATEGORY_ITEM"
    lazy="true"
    cascade="save-update">
    <key column="CATEGORY_ID" />
    <many-to-many class="Item" column="ITEM_ID" />
  </set>
</class>
```

我們可以重複使用對應雙向聯合在 `Category` 端。我們對應 `Item` 這一端如下:

```
<class name="Item" table="ITEM">
  ...
  <set name="categories"
    table="CATEGORY_ITEM"
    lazy="true"
    inverse="true"
    cascade="save-update">
    <key column="ITEM_ID" />
    <many-to-many class="Item" column="CATEGORY_ID" />
  </set>
</class>
```

注意 `inverse="true"` 的使用。再一次，這個設定告訴 Hibernate 如果我們在 Java 程式碼中操作這個聯合，忽視由 `categories` 端的 `collection` 所作的改變而使用另一端的聯合 (`item` 的 `collection`) 為表達應該要與資料庫作同步。

我們已經選擇 `cascade="save-update"` 在 `collection` 的兩個端點，這不會是不合理的。另一方面，`cascade="all"°Acascade="delete"`，與 `cascade="all-delete-orphans"` 在多對多的聯合中沒有意義，因為有著多個父親的實例在只有一個父親被刪除時不應該被刪除。

哪一種 `collection` 可以被使用在雙向多對多聯合之中？你需要在 `collection` 的二端使用相同的型態？以下的使用是合理的，舉例而言，一個 `list` 在沒有被標記為 `inverse="true"` 的那一端（或額外設定成 `false`）且一個 `bag` 在被標記為 `inverse="true"` 的這一端。

你可以我們在單向多對多聯合中所展示的，使用任何的對應作為雙向聯合不可反轉的那一端。`<set>`，`<idbag>`，`<list>`，與 `<map>` 都是可能的，且對應和之前所展示的是一致的。

對於反轉端，`<set>` 是允許的，如同下列的 `bag` 對應：

```
<class name="Item" table="ITEM">
  ...
  <bag name="categories"
        table="CATEGORY_ITEM"
        lazy="true"
        inverse="true" cascade="save-update">
    <key column="ITEM_ID"/>
    <many-to-many class="Item" column="CATEGORY_ID"/>
  </bag>
</class>
```

這是第一次我們看到 `<bag>` 的宣告：它和 `<idbag>` 對應相似，但它並沒有牽涉輔助鍵值欄位。它讓你使用 `List`（使用 `bag` 的語義）在一個永續類別之中而不是使用 `set`。因此，如果多對多聯合對應的不可逆端使用 `map`，`list` 或 `bag` 是較偏好的。記住 `bag` 並不會保持元件的順序，儘管這是 `List` 型態在 Java 屬性中的定義。

沒有其它的對應應該在多對多聯合的可逆端被使用。索引的 `collection` (`list` 和 `map`) 不應該被使用。因為如果 `inverse="true"`，Hibernate 不會初始化或維持索引欄位。記住對所有其它牽涉到 `collection` 的聯合對應下列都是真的且重要的：一個索引的 `collection`（或甚至是陣列）不能被設定為 `inverse="true"`。

我們已經對使用多對多聯合感到不滿，且建議使用合成元件對應作為替代。讓我們看看這如何運作。

對多對多聯合使用元件的 collection

假設我們在每次增加一個 Item 到 Category 時需要記錄一些資訊。舉例來說，我們需要儲存日期和增加這個項目到類別的使用者名稱。我們需要一個 Java 類別來表達這個資訊：

```
public class CategorizedItem {
    private String username;
    private Date dateAdded;
    private Item item;
    private Category category;
    ....
}
```

(我們刪去了存取函數和 equals() 和 hashCode() 函數，但是他們對於元件類別是必要的。)

我們對應 items 的 collection 到 Category 如下：

```
<set name="items" lazy="true" table="CATEGORY_ITEMS">
  <key column="CATEGORY_ID"/>
  <composite-element class="CategorizedItem">
    <parent name="category"/>
    <many-to-one name="item"
      class="Item"
      column="ITEM_ID"
      not-null="true"/>
    <property name="username" column="USERNAME" not-null="true"/>
    <property name="dateAdded" column="DATE_ADDED" not-null="true"/>
  </composite-element>
</set>
```

我們使用 <many-to-one> 元件來宣告 Item 的聯合，且我們使用 <property> 對應來宣告額外聯合相關的資訊。這個連結表格現在有四個欄位：CATEGORY_ID, ITEM_ID, USERNAME 和 DATE_ADDED。CategorizedItem 屬性的欄位應該永遠不是 null：不然我們不能辨別一個單一的連結項目，因為他們都是表格主鍵值的一部份。你可以在圖 6.11 中看到表格的結構。

事實上，不只是對應 username，我們想要保持一個真正 User 物件的參照。在這個情形，我們有下列三元聯合對應：

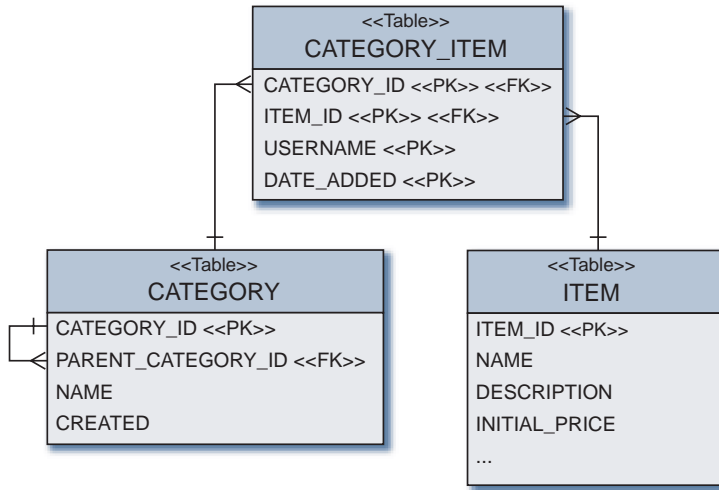


Figure 6.11
使用一個元件的多對多
個體聯合

```

<set name="items" lazy="true" table="CATEGORY_ITEMS">
  <key column="CATEGORY_ID" />
  <composite-element class="CategorizedItem">
    <parent name="category" />
    <many-to-one name="item"
      class="Item"
      column="ITEM_ID"
      not-null="true" />
    <many-to-one name="user"
      class="User"
      column="USER_ID"
      not-null="true" />
    <property name="dateAdded" column="DATE_ADDED" not-null="true" />
  </composite-element>
</set>

```

這是一個奇怪的情況！如果你發現你自己有一個像這樣的對應，你應該要問是否對應 `CategorizedItem` 成一個個體類別且使用兩個一對多的聯合會更好。而且，沒有方法使這個對應雙向化：一個元件（如同一個 `CategorizedItem`）不能，依照其定義，有共用的參照。你不能從 `Item` 走訪到 `CategorizedItem`。

我們在前一節討論到了一些多對多對應的限制。如果他們是雙向的，他們其中的一個，限制非索引 `collection` 作為聯合的可逆端，也套用到一對多聯合。讓我們再一次仔細看看一對多和多對一，來恢復你的記憶且詳細說明我們在第三章中所學到的。

一對多聯合

你已經從第三章知道一對多聯合中大部份你需要知道的。我們在兩個個體永續類別對應一個典型的父 / 子聯合，Item 和 Bid。這是一個雙向聯合，使用 <one-to-many> 和 <many-to-one> 對應。聯合中 "許多" 的這一端是用 Set 在 Java 中實作的；我們在 Item 類別中有一個 bids 的 collection。讓我們重新思考這個對應且走過一些特別的情形。

使用有 set 語義的 bag

舉例而言，如果你絕對需要在父 Java 類別中有 List 型態的子類別，是有可能使用 <bag> 對應來代替 set。在我們的範例，首先我們需要取代 Item 中永續類別中的 bids 的 collection 型態為 List。這個 Item 和 Bid 聯合的對應本質上是沒有改變的：

```
<class
  name="Bid"
  table="BID">
  ...
  <many-to-one
    name="item"
    column="ITEM_ID"
    class="Item"
    not-null="true"/>
</class>

<class
  name="Item"
  table="ITEM">
  ...
  <bag
    name="bids"
    inverse="true"
    cascade="all-delete-orphan">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
  </bag>
</class>
```

我們重新命名 <set> 為 <bag>，沒有作其它改變。注意，然而，這個改變並不是有用的：底層的表格結構並不支援重複，所以 <set> 對應導致聯合有 set 的語義。有些偏好甚至是當聯合有 set 的語義時也使用 List，但我們的不是這樣，我們建議在典型的父 / 子關係中使用 <set> 對應。

明顯（且是錯誤的）解決方案是對 bids 用真正的 <list> 對應，且加上額外的欄位來存放元素的位置。記住我們在這章之前所介紹 Hibernate 的限制。inverse="true" 這一端的聯合當 Hibernate 儲存物件的狀態時是不會被考量的，所以 Hibernate 會忽略元素的索引而不會更新欄位的位置。

然而，如果你的父 / 子關係只會是單一方向的（只有可能由父往子），你甚至可以使用索引的 collection 型態（因為 " 許多 " 這一端不再是相反的那一端）。實務中能夠好好使用單方向的聯合並不常見，且我們在拍賣應用程式中一個也沒有。你可能記得我們在第三章中由 Item 和 Bid 的對應開始。一開始讓它單方向，但我們很快地介紹了另一方向的對應。

讓我們找另一個例子來實作有索引 collection 的單方向一對多聯合。

單方向對應

爲了這一節，我們現在假設在 Category 與 Item 的聯合重新模型成一對多的聯合（一個項目現在最多屬於一個目錄），且 Item 並不再有參照到它現在的目錄。在 Java 程式，我們模型這個成一個名爲 items 的 collection 在 Category 類別之中；如果我們沒有使用 collection，我們不需要改變任何東西。如果 items 實作了 Set，我們使用了下列的對應：

```
<set name="items" lazy="true">
  <key column="CATEGORY_ID" />
  <one-to-many class="Item" />
</set>
```

記住一對多聯合對應不需要宣告表格名稱。Hibernate 已經在 collection 對應時知道了欄位的名稱（在這個情形，只有 CATEGORY_ID 屬於 ITEM 表格。這個表格結構如圖 6.12 所示。

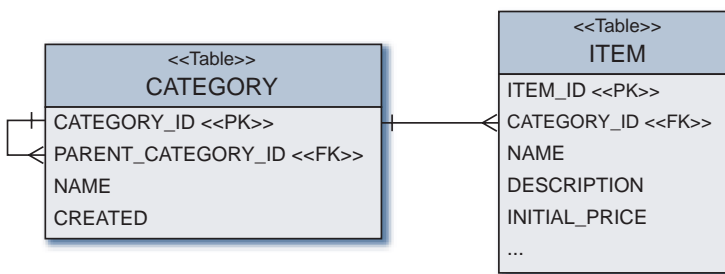


Figure 6.12
一個使用外部鍵欄位的
獨立一對多聯合

另一端的聯合，Item 類別沒有到 Category 的對應參照。我們現在也在 Category 中使用索引 collection，舉例來說，當我們改變了 List 的屬性之後：

```
<list name="items" lazy="true">
  <key>
    <column name="CATEGORY_ID" not-null="false"/>
  </key>
  <index column="DISPLAY_POSITION"/>
  <one-to-many class="Item"/>
</list>
```

注意在 ITEM 表格中新的 DISPLAY_POSITION 欄位，它持有 Item 元素在 collection 中的位置。

有一個重要的議題需要考量，在我們的經驗中一開始困擾許多 Hibernate 的使用者。在一個單向一對多的聯合，ITEM 中的外部鍵欄位 CATEGORY_ID 必須可為空值。Item 可以不用知道任何有關 Category 的事就被儲存，它是一個獨立的個體！這是一個一致的模型和對應，且如果你處理 not-null 外部鍵和父 / 子關係時你或許會思考它兩次。使用一個雙向聯合（和一個 Set）是一個正確的方案。

現在你知道對於一般個體有關聯合對應的技巧，我們仍然需要考慮繼承和不同等級繼承階層的聯合。我們真正想要的是多型的行為。讓我們看 Hibernate 如何處理多型個體聯合。

6.4 對應多型聯合

多型是一個物件導向語言，如 Java 關鍵的特性。支援多型聯合和多型查詢是像 Hibernate 般的 ORM 的基本功能。另人驚訝地，我們設法走這麼遠而不用討論太多的多型。更另人驚訝地，這個主題沒有太多需要說，在 Hibernate 中多型是如此易使用所以我們不需要花太多精力來解釋這個特性。

爲了得到一個概況，我們先考慮一個可能有子類別的類別多對一聯合。在這個情形，Hibernate 保證你可以創建連接到任何子類別實例，就如同你想要對父類別的實例所作的。

6.4.1 多型多對一聯合

一個多型聯合是一個聯合，可以指到子類別的實例，其父類別是我們在對應後設資料中額外指定的。對於這個範例，想像我們每個User沒有很多的BillingDetails，但是只有一個，如圖 6.13 所示。

我們對應到抽象類別 BillingDetails 的聯合如下：

```
<many-to-one name="billingDetails"
  class="BillingDetails"
  column="BILLING_DETAILS_ID"
  cascade="save-update" />
```

但因為 BillingDetails 是抽象的，在執行時期，聯合必須到它子類別的實例，CreditCard 或 BankAccount。

所有至此在這個章節中我們介紹的聯合對應支援多型。你並不需要作任何特別的事才能在 Hibernate 中支援多型聯合；指定任何對應的永續類別的名稱在你的聯合對應(或讓Hibernate用反射來發現)；接著，如果那個類別宣告任何<subclass>或 <joined-subclass> 元素，聯合自然就是多型的。

下列程式碼展示創建一個聯合到 CreditCard 子類別的實例：

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
```

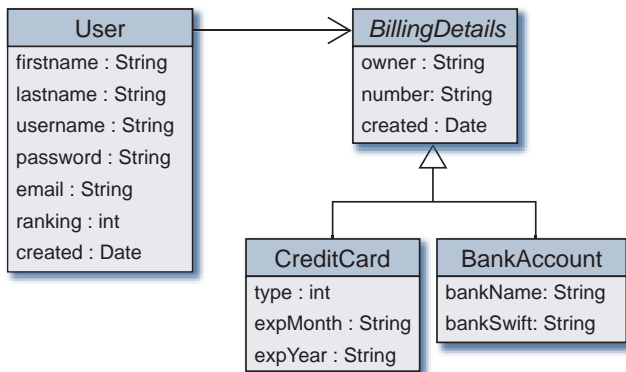


Figure 6.13
使用者只有一個帳單資訊

```
user.setBillingDetails(cc);  
  
tx.commit();  
session.close();
```

現在當我們在第二個交易時走訪聯合，Hibernate 會自動取出 CreditCard 的實例：

```
Session session = sessions.openSession();  
Transaction tx = session.beginTransaction();  
  
User user = (User) session.get(User.class, uid);  
// Invoke the pay() method on the actual subclass  
user.getBillingDetails().pay(paymentAmount);  
  
tx.commit();  
session.close();
```

有一件事要注意：如果 BillingDetails 是用 lazy="true" 對應，Hibernate 會代理 billingDetails 聯合。在這個情形，我們不需要在執行時期執行型別轉換到 CreditCard 的具體類別，且甚至 instanceof 運算元的行為會很奇怪：

```
User user = (User) session.get(User.class, uid);  
BillingDetails bd = user.getBillingDetails();  
System.out.println( bd instanceof CreditCard ); // prints "false"  
CreditCard cc = (CreditCard) bd; // ClassCastException!
```

在這個程式碼中，型別轉換之所以失敗是因為 bd 是一個代理實例。當一個函數被代理所呼叫，呼叫被委託到 CreditCard 實例是延後取出的。要執行一個代理安全的型別轉換，使用 Session.load()：

```
User user = (User) session.get(User.class, uid);  
BillingDetails bd = user.getBillingDetails();  
// Get a proxy of the subclass, doesn't hit the database  
CreditCard cc =  
    (CreditCard) session.load( CreditCard.class, bd.getId() );  
expiryDate = cc.getExpiryDate();
```

當這個呼叫載入後，bc 跟 cc 指到兩個不同的代理實例，它們都代理到相同的底層 CreditCard 實例。

注意你可以避免延後取出來避免這些議題，如同在下列的程式碼，使用在下一章會討論的查詢技巧：

```
User user = (User) session.createCriteria(User.class)  
    .add( Expression.eq("id", uid) )  
    .setFetchMode("billingDetails", FetchMode.EAGER)  
    .uniqueResult();  
// The user's billingDetails were fetched eagerly
```

```
CreditCard cc = (CreditCard) user.getBillingDetails();
expiryDate = cc.getExpiryDate();
```

真正的物件導向程式碼不應該使用 `instanceof` 或許多的型態轉換。如果你發現自己在使用代理時遇到問題，你該分析你的設計，詢問是否有一個更多型的方式。

一對一聯合是用同樣的方法處理，那麼多值聯合該如何呢？

6.4.2 多型 collection

讓我們重構之前的例子到它原本 `CaveatEmptor` 的範例。如果 `User` 擁有許多的 `BillingDetails`，我們使用一個雙向一對多。在 `BillingDetails`，我們有如下的：

```
<many-to-one name="user"
  class="User"
  column="USER_ID" />
```

在 `User` 對應，我們有這個：

```
<set name="billingDetails"
  lazy="true"
  cascade="save-update"
  inverse="true">
  <key column="USER_ID" />
  <one-to-many class="BillingDetails" />
</set>
```

增加一個 `CreditCard` 是簡單的：

```
CreditCard cc = new CreditCard();
cc.setNumber(ccNumber);
cc.setType(ccType);
cc.setExpiryDate(ccExpiryDate);

Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
// Call convenience method that sets both "ends"
user.addBillingDetails(cc);

tx.commit();
session.close();
```

如同一般，`addBillingDetails()` 呼叫 `getBillingDetails().add(cc)` 與 `cc.setUser(this)`。

我們可以循序取出 `collection` 且處理 `CreditCard` 的實例與多型化 `BankAccount` (我們不希望在我們最後的系統要使用者多次付款)：

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();

User user = (User) session.get(User.class, uid);
Iterator iter = user.getBillingDetails().iterator();
while ( iter.hasNext() ) {
    BillingDetails bd = (BillingDetails) iter.next();
    // Invoke CreditCard.pay() or BankAccount.pay()
    bd.pay(ccPaymentAmount);
}

tx.commit();
session.close();
```

這個例子到此，我們假設 `BillingDetails` 是一個類別在 Hibernate 對應文件中額外指定，且繼承對應策略是每個階層一個表格或每個子類別一個表格。我們還沒有考慮到每個具體類別一個表格這種對應策略的例子，在那裡 `BillingDetails` 不會額外在對應檔案中被提及（只在子類別的 Java 定義之中）。

6.4.3 多型聯合與每個具體類別一個表格

在 3.6.1 節，"每個具體類別一個表格" 中，我們定義了每個具體類別一個表格的對應策略且觀察到這種對應策略使得表達多型聯合變得困難，因為你不能對應對應外部鍵的關係到抽象父類別的表格。在這種策略下父類別是沒有表格的；你只有具體類別的表格。

假設我們想要表達從 `User` 到 `BillingDetails` 的多對一多型聯合，在那裡 `BillingDetails` 類別階層是用每個具體類別有一個表格的策略。有一個 `CREDIT_CARD` 表格和 `BANK_ACCOUNT` 表格，但是沒有 `BILLING_DETAILS` 表格。我們需要兩個 `USER` 表格中的資訊來獨一辨識聯合的 `CreditCard` 或 `BankAccount`：

- 被聯合實例所在的表格名稱
- 被聯合實例的辨識元

`USER` 表格需要額外的 `BILLIND_DETAIL_TYPE` 欄位，除了 `BILLING_DETAIL_ID` 之外。我們使用 `HIBERNATE <any>` 元素來對應這個聯合：

```
<any name="billingDetails"
      meta-type="string"
      id-type="long"
      cascade="save-update">
  <meta-value value="CREDIT_CARD" class="CreditCard"/>
  <meta-value value="BANK_ACCOUNT" class="BankAccount"/>
```

```

    <column name="BILLING_DETAILS_TYPE"/>
    <column name="BILLING_DETAILS_ID" />
  </any>

```

meta-type 屬性指定了 Hibernate 的型態在 BILLING_DETAIL_TYPE 欄位；id-type 屬性指定 BILLING_DETAIL_ID 的型態（CreditCard 和 BankAccount 必須有相同的辨識型態）。注意欄位的次序是重要的：第一是型態，再來是辨識元。

<meta-value> 元素告 Hibernate 如何解讀 BILLING_DETAIL_TYPE 欄位。我們不需要使用完整的表格名稱，我們可以使用任何我們想要的值作為型態鑑識元。舉例來說，我們可以將資訊編碼成兩個字元：

```

<any name="billingDetails"
    meta-type="string"
    id-type="long"
    cascade="save-update">
  <meta-value value="CC" class="CreditCard"/>
  <meta-value value="CA" class="BankAccount"/>
  <column name="BILLING_DETAILS_TYPE"/>
  <column name="BILLING_DETAILS_ID" />
</any>

```

一個表格結構的範例可以看 6.14。

這裡有一個這種聯合第一個主要的問題：我們不能增加一個外部鍵限制到 BILLING_DETAILS_ID 欄位，因為一些值參照到 BANK_ACCOUNT 表格和其他會到

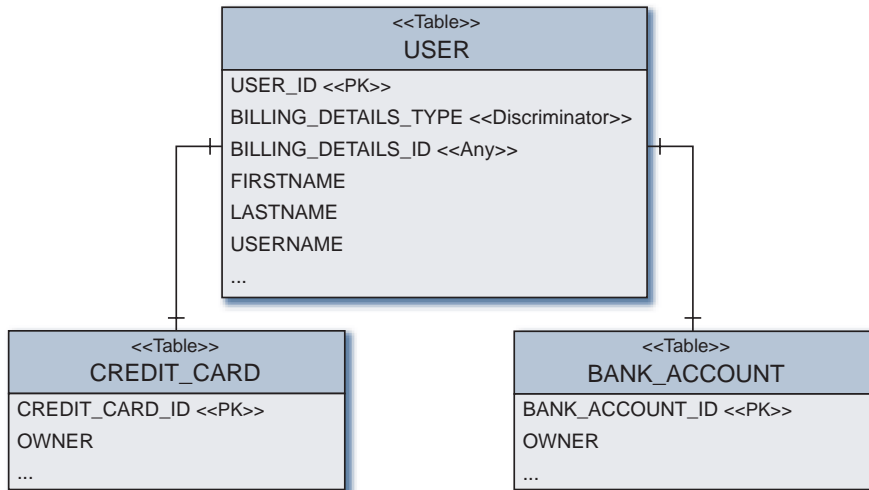


Figure 6.14 使用鑑識元欄位到任何聯合

CREDIT_CARD 表格。所以，我們必須提供一些其它的方式來確保一致性（例如扳機（trigger））。

而且，爲了聯合寫 SQL 表格連接是困難的。特別是，Hibernate 查詢能力並不支援這種聯合對應，也不能用外部連結來取出這個聯合。除了最特別的情形，我們不鼓勵在所有場合使用 <any> 聯合。

如同你可以看到的，多型在每個具體類別一個表格的繼承對應策略的情形是較亂的。當多型聯合是需要時，我們通常不會使用這種策略。只要你忠於其它的繼承對應策略，多型是直觀的，且你通常不需要思考它。

6.5 結論

這一章包含了 ORM 的細部觀點和需要解決結構上不協調的問題所需要的技巧。我們現在可以完整對應所有在 CaveatEmptor 領域模型的個體與聯合。

Hibernate 型態系統從分辨了個體與值型態。一個個體有它自己的生命週期和永續辨識元；一個值型態的實例完全依靠擁有者的個體。

Hibernate 定義了豐富多樣的內建值對應型態。當預訂型態是不夠的，你可以輕易使用自訂型態或複合元件對應延伸他們且甚至實作任意從 Java 到 SQL 資料型態的轉換。

Collection 值型態的屬性被視爲值型態。一個 collection 沒有它自己的永續辨識元且屬於一個單一擁有者的個體。你已經看到如何對應 collection，包括值型態實例的 collection 和多值個體聯合。

Hibernate 支援一對一，一對多，和多對多的個體間聯合。實際上，我們建議不要過度使用多對多聯合。Hibernate 中的聯合自然是多型的。我們也討論這種關係的雙向行爲。