

MongoDB 資料庫設計

Chap1 之 MongoDB 安裝設定

什麼是 MongoDB ?

- 1.它是 10gen 開發的 NoSQL (Not only SQL-不只有 SQL)
- 2.不是關聯式資料庫，沒有 Schema
- 3.它是用來處理巨量資料的資料庫，這個資料庫是以文件導向儲存(Document Oriented Storage)的資料庫，以 key:value 表示, 儲存格式與 JSON 完全一樣。

使用 MongoDB 的好處

- 1.提供豐富的查詢
- 2.容易向外擴展
- 3.沒有複雜的關聯
- 4.使用內部記憶體儲存工作集，進而實現更快的數據訪問
- 5.單一物件結構清楚
- 6.不需要應用程式與資料庫物件之間的轉換與對應

什麼時候會需要 MongoDB ?

- 1.Big Data
- 2.資料採礦
- 3.手機及社交平台
- 4.使用者資料管理
- 5.資料中心

MongoDB 資料 v.s. 關聯式資料庫

接下來，有一些 MongoDB 的術語，與我們過去關聯式資料庫所用的名詞，對照一下：

RDBMS	MongoDB
Database	Database
Table	Collection

RDBMS	MongoDB
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (如果沒有設定，mongodb 會自動給予 預設主鍵值 _id 欄位)

在 Windows 上安裝 MongoDB

在安裝前，我們先來確認 windows 版本。

在 command prompt 輸入，並執行：

```
C:\> wmic os get osarchitecture
```

OSArchitecture 64-bit <--顯示出所用的 OS 是哪一種結構。

接著，我們去 [MongoDB Downlad Center](http://www.mongodb.org/downloads) 下載所需套件：

安裝流程

- 可以選 Custom 安裝，這裡就可以設定安裝的路徑。
- 系統預設是安裝在 C:\Program Files\MongoDB\Server\3.0\ 那我們這裡就依照官方的，不改。
-
- 執行啟動 MongoDB

指定 MongoDB 資料庫存放位置是 \data\db，所以，我們可以建立這樣的資料庫路徑：

在 C:\ 以 Command prompt 執行 C:\>md \shared\data\db

新增三層資料夾，

接著我們在 command prompt 要想進入到

```
C:\Program Files\MongoDB\Server\3.0\bin
```

執行

```
D:>cd "C:\Program Files\MongoDB\Server\3.0\bin"
```

目前，我們位在
C:\Program Files\MongoDB\Server\3.0\bin>

在 > 後面輸入
mongod --dbpath c:\shared\data\db
啟動同時，指定資料庫存放位置！

表示 MongoDB 已經正常啟動了!!!!

你就可以開始使用了!!!

Note. 關於資料庫存放位置，其實可以自己設定好一個資料夾位置，指定過去就好囉!!! 不限於一定得在 \data\db 之下!!!!

使用 MongoDB Shell

1. 資料庫啟動以後，我們開啟另一個 command prompt 執行
cd "C:\Program Files\MongoDB\Server\3.0\bin\"

mongo.exe (.exe 可以不用)

啟動 MongoDB Shell 環境：

當我們看到 connecting to: test. Welcome to the MongoDB shell, 代表連線 MongoDB 成功！

2. 我們可以做一些簡單的查詢，比方

show dbs

可以看到底下有哪些資料庫。
要使用資料庫 test，可以用

use test

3. 如何關閉資料庫
離開 shell，使用 exit

關閉資料庫，中斷連線，在 admin 資料庫下，使用 `db.shutdownServer()`

4.查詢指令使用，好用的"help"

Chap2 之 MongoDB 基礎

~ 什麼是 MongoDB ~

MongoDB 一種強大，靈活、且易於擴展的文件導向式(document-oriented)資料庫，與傳統的關聯式導向資料庫相比，它不再有 row 的概念，取而代之的是 document 的概念。

~ MongoDB 的優缺與缺點 ~

優點

- Schema-less : MongoDB 擁有非常彈性的 Schema，這對 RDBMS 來說非常的難以高效能的方法來實現。
- 易於擴展：MongoDB 的設計採用橫向擴展，它的 document 的數據模型使寫能很容易在多台伺服器之間進行數據分割。
- 優透的性能：MongoDB 能預分配，以利用額外的空間換取穩定，同時盡可能把多的內存用作 cache，試圖為每次查詢自動選擇正確的索引。

缺點

- 不支援事務操作：所以通常不適合應用在銀行或會計這種系統上，因為不包證一致性。
- 占用比較多空間：主要是有兩個原因，首先是它會預分配空間，為了提高效能，而第二個原因是欄位所占用的空間。

~ MongoDB 的組成 Document 與 Collection ~

Document

Document 是 **mongodb** 的核心，它就是 **Key** 對應個 **Value** 組合，例如下列範例。

```
{
  name : "mark",
  age : 100 ,
  title : 'Mark BIG BIG'
}
```

document 中的值可以是多種不同的類型，並且 **Key** 有幾個規定，首先它是區分大小寫，例如下面的範例這兩種是不同的，**mongodb** 會存成兩份 **document**。

```
{ name : "mark" }
{ Name : "mark" }
```

而另一個規定是 **Key** 不能相同的，例如下面的範例是非法的。

```
{ age : "100" , age : "1000" }
```

Collection

Collection 就是一組 **Document**，如果把它用來與關聯式資料庫比較，他就是 **Table** 裡面存放了很多 **Row**。

Collection 是動態的，這代表這一個 **collection** 裡的 **document** 可以是各種類型，例如下面這幾種文檔都可以存放在同一個 **collection** 裡，不像關聯式規定的好好。

```
{ id :1, name : "mark" }
{ age : 100 }
```

chap3 之 CRUD---新增

~ Insert 方法 ~

單筆資料 Insert

`insert` 函數可以將一個 `document` 建立到 `collection` 裡，我們這裡建立一個簡單的範例來看如何使用 `insert`。

首先我們的需求是要建立一份使用者清單(`collection`)，然後可以存放多筆使用者資料(`document`)，我們假設使用者資料如下。

順到一提，`mongodb` 自帶 `javascript shell`，所以可以在 `shell` 執行 `javascript` 一些語法。

```
user1 = {  
  name : "Mark",  
  age : 18,  
  bmi : 10  
}
```

然後我們要將這筆 `document` 新增至 `user` 的 `collection` 裡。

```
db.user.drop()
```

```
db.user.insert(user1);
```

程式執行過程回傳值如下，代表成功新增一筆。

```
WriteResult({"nInserted" : 1})
```

新增完後，我們可以執行 `find` 指令，來查看 `user` 這 `collection` 中的資料。


```
db.user.find()
```

多筆資料 Insert

Insert 函數同時也可以執行多筆。其中注意 **insert** 有個參數 **ordered**，**true** 時代表如果其中一筆資料有問題，它就會停止下來，後面的資料都不會新增，而 **false** 時，則代表不會停下來，後面的資料會繼續新增，預設是 **true**。

我們用下面範例來看看使用方法。

```
var user1 = {
    name : "Mark",
    age : 18,
    bmi : 10
},
count = 1000,
users = [];

for (var i=0;i<count;i++){
    users.push(user1);
}

db.user.insert(users,{ordered:false})
```

chap4 之 CRUD---新增之 Bulk 與新增效能測試

~ Bulk Insert 方法 ~

Bulk Insert 在 2.6 版時發佈，它也是種新增方法，效能如何等等會比較，基本使用方法有分有兩 **Unordered Operations** 和 **Ordered Operations**。

Ordered Operations

Ordered Operations，mongodb 在執行列表的寫入操作時，如果其中一個發生錯誤，它就會停止下來，不會在繼續執行列表內的其它寫入操作，並且前面的操作不會 **rollback**。

使用範例如下。

```
var bulk = db.user.initializeOrderedBulkOp();
bulk.insert( { name: "mark"} );
bulk.insert( { name: "hoho"} );
bulk.execute();
```

Unordered Operations

Unordered Operations，mongodb 在執行列表的寫入操作時，如果其中一個發生錯誤，它不會停止下來，會繼續執行列表內的其它寫入操作，速度較快。

使用範例如下。

```
var bulk = db.user.initializeUnorderedBulkOp();
```

```
bulk.insert( { name: "mark"} );  
bulk.insert( { name: "hoho"} );  
bulk.execute();
```

Ordered 與 **Unordered** 我們在要如何選擇使用時機呢，記好只要有相關性的操作就要選擇用 **Ordered**，而如果像是 **log** 之類的，流失一兩筆也是沒差，這時可以選用 **Unordered**。

chap5 之 CRUD---更新

~ 基本更新方法 Update ~

Update 函數主要的功用就如同字面所說，更新~，而使用方法如下，**query** 就是指你要先尋找更新的目標條件，**update** 就是你要更新的值。而另外三個參考請考下列。

- **upsert**：這個參數如果是 **true**，代表如果沒有找到該更新的對像，則新增，反之則否，默認是 **false**。
- **multi**：如果是 **false**，則代表你 **query** 出多筆，他就只會更新第一筆，反之則都更新，默認是 **false**(!注意 **multi** 只能在有**修改器**時才能用)。
- **writeConcern**：拋出異常的級別。

```
db.collection.update(  
  <query>,  
  <update>,  
  {  
    upsert: <boolean>,  
    multi: <boolean>,  
    writeConcern: <document>  
  }  
)
```

下面來簡單示範一下用法。首先我們先新增三筆資料。

```
db.user.insert({"name":"mark","age":23});  
db.user.insert({"name":"steven","age":23});  
db.user.insert({"name":"jj","age":23});
```

然後我們將名字為 **mark** 這人的 **age** 改為 **18**，指令如下，**query** 為 `{"name":"mark"}`。

```
db.user.update({"name":"mark"}, {"name":"mark", "age":18})
```

~ 更新修改器 (**\$set**、**\$inc**) ~

修改器**\$set**

\$set 修改器主要的功用就是用來指定一個字段的值，不用像上面一樣整個替換掉。

所以如我們如果要將 **mark** 這位仁兄的 **age** 改為 **18** 只要下達下面的指令。

```
db.user.insert({"name":"mark","age":23});  
db.user.insert({"name":"steven","age":23});  
db.user.insert({"name":"jj","age":23});
```

```
db.user.update({"name":"mark"}, {"$set" : { "age" : 18} })
```

執行結果成功更新為 **age** 為 **18**

修改器**\$inc**

假設一下情景，假如有個投票網站、或是要存放訪客數的功能，每次更新時都是要+1，這種時後就可以用\$inc 來更新你的 document，理論上來說速度應該會優於\$set。

注意\$inc 只能用在數值類型，否則就會提示 Modifier \$inc allowed for numbers only。

我們寫段程式碼來看看他的使用方法，下面範例我們先新增一筆資料，然後我們每次更新時，like 都會加 1，所以我們更新 3 次，理論上 like 會變為 3。

```
db.user.insert({"id" : 1,"like" : 0})

db.user.update({"id" : 1},{"$inc" : {"like" : 1}})
db.user.update({"id" : 1},{"$inc" : {"like" : 1}})
db.user.update({"id" : 1},{"$inc" : {"like" : 1}})
```

執行結果如下，可以看到 like 增加到 3 了。

chap6 之 CRUD---更新之陣列欄位攻略

本篇文章將要說明陣列修改器`$push`，主要就是針對 `document` 中的陣列進行修改，同時他也可以搭配`$each`、`$slice`、`$ne`、`$addToSet`、`$pop`、`$pull` 來使用。

~ 陣列更新修改器攻略 ~

`$push`

`$push` 是陣列修改器，假如一個 `document` 中已經有陣列的結構，使用 `push` 會在陣列的尾末加入一個新元素，要是本來就沒有這個陣列，則會自動新建一筆。

使用方法如下範例，首先先新增一筆資料，然後新增加一個叫 `jack` 的 `fans`。

```
db.user.insert({
  "name" : "mark",
  "fans" : ["steven","crisis","stanly"]
})
db.user.update({"name":"mark"},
  {$push:{"fans" : "jack"}}
)
```

`$each`

`$push` 一次新增只能新增一筆元素，而搭配`$each` 就可以新增多筆。

使用方法如下範例，一樣首先新增一筆資料，然後這時我們一次新增三個 `fans` 分別為 `jack`、`landry`、`max`。

```
db.user.insert({
```

```

        "name" : "mark",
        "fans" : ["steven","crisis","stanly"]
    })

    db.user.update({"name":"mark"},
        {"$push" : {"fans" : {"$each" : ["jack","lnadry","max"]}}}
    )

```

\$slice

如果你希望限制一個陣列的大小，就算多 **push** 進元素，也不要超過限制大小，這時你就可以用**\$slice**，不過注意它是保留最後 **n** 個元素。

使用方法如下，新增一筆資料，然後我們希望 **fans** 人數不超過 **5** 人，但我們硬多塞一個人進去。

```

    db.user.insert({
        "name" : "mark",
        "fans" : ["steven","crisis","stanly"]
    })

    db.user.update({"name":"mark"},
        {"$push" : {"fans" :
            {"$each" : ["jack","lnadry","max"],
            "$slice" : -5 }}}
    )

```

執行結果如下，可以看到第一位 **steven** 被刪除，只保留了最後 **5** 位。

\$addToSet

你可能有這個需求，假設你要新增一個元素到陣列裡，並且保證陣列內的元素不會重複，這時就可以使用 **\$addToSet**。

使用方法如下範例，新增一筆資料，然後 **fans** 有 **steven**、**landry**、**stanly**，這時我們在新增 **steven** 和 **jack** 進去，預期應該 **steven** 不會被新增進去，也就是不會產生兩個 **steven**。

```
db.user.insert({
  "name" : "mark",
  "fans" : ["steven","crisis","stanly"]
})

db.user.update({"name":"mark"},
{"$addToSet" : {"fans" :
  {"$each" : ["steven","jack"]} }}
)
```

\$pop 與 \$pull

\$pop 與 **\$pull** 這兩個修改器都是用來刪除元素用的，**\$pop** 可以從頭或尾刪除，而 **\$pull** 則是基於特定條件來刪除。

先來看看 **\$pop** 的使用範例。其中 **"fans":1** 代表從 **fans** 陣列尾刪除 **"fans":-1** 則從陣列頭刪除。

```
db.user.insert({
  "name" : "mark",
  "fans" : ["steven","crisis","stanly"]
})

db.user.update({"name":"mark"},
{"$pop" : {"fans":1}}
```

)

這時我們在來看看`$pull` 用法，假設我們要將 `crisis` 這 `fans` 刪除，使用方法如下。

```
db.user.insert({
  "name" : "mark",
  "fans" : ["steven","crisis","stanly"]
})

db.user.update({"name":"mark"},
  {"$pull" : {"fans":"crisis"}}
)
```

chap7 之 CRUD---刪除

~ MongoDB 的刪除方法 ~

remove

`remove` 方法是 `mongodb` 裡最基本的刪除 `document` 的方法，但這邊要注意就算你刪除了

`document` 它的 `index` 與預分配空間都不會刪除。

使用方法與參數如下

- `justOne` 預設 `false`，代表 `query` 到幾個就會刪除幾個，`true` 則只會刪第一個。
- `writeConcern` 為拋出異常的級別。
- `collation` 是 3.4 版開始支持的功能，可依照語言定義來針對文字的內容進行解讀，再還沒支持 `collation` 前一徑依字節來對比。

```
db.collection.remove(  
  <query>,  
  {  
    justOne: <boolean>,  
    writeConcern: <document>,  
    collation: <document>  
  }  
)
```

使用範例如下，我們來新增三筆資料，然後刪除掉 `steven` 該筆資料。

```
db.user.insert({"name":"mark","age":23});  
db.user.insert({"name":"steven","age":23});  
db.user.insert({"name":"jj","age":23});
```

```
db.user.remove({"name":"steven"})
```

刪除所有資料

`remove` 可以用來刪除 `collection` 的所有資料，但還有另一種方法也是刪除 `collection` 的所有資料，那就是 `drop`，但它同時會將 `index` 給全部刪除。

兩種的使用方法如下。

```
db.user.remove({})
```

```
db.user.drop()
```

bulk delete

`bulk` 操作故名思意就是要來衝一下大筆資料刪除的效能方法。

使用方法如下。

```
//先新增二筆資料
```

```
var bulk = db.user.initializeUnorderedBulkOp();  
bulk.insert( { name: "mark" } );  
bulk.insert( { name: "hoho" } );  
bulk.execute();
```

```
//然後再刪除掉 mark 該筆
```

```
var bulk = db.user.initializeUnorderedBulkOp();  
bulk.find( { "name": "mark" } ).remove();  
bulk.execute();
```

chap8 之 CRUD---搜尋之 find 與搜尋操作符號

find 方法基本說明

~ find 的搜尋條件 ~

這邊我們將要說明 **find** 常用搜尋條件，**and**、**or**、大於等於、大於、小於、小於等於、包含、不包含，有了這些條件我們就可以更方便的尋找你所需要的 **document**。

這邊簡單的整理成一張表來對應操作符號。

條件	操作符號
AND	\$and ，另一種方法也可以直接在 query 中下 {"key1":"value1","key2":"value2"}
OR	\$or
NOT	\$not
NOR	\$nor
大於	\$gt

條件	操作符號
大於等於	\$gte
小於	\$lt
小於等於	\$lte
包含	\$in
不包含	\$nin

我們接下來會先產生幾筆測試資料，再來測試幾個搜尋故事。

測試資料如下，這是一組使用者資訊，裡面記載了使用者的名稱、年紀、粉絲數以及喜歡數。

//collection 為 user

```
{ "id": "1", "name": "mark", "age": 25, "fans": 100, "likes": 1000 }
{ "id": "2", "name": "steven", "age": 35, "fans": 220, "likes": 50 }
{ "id": "3", "name": "stanly", "age": 30, "fans": 120, "likes": 33 }
{ "id": "4", "name": "max", "age": 60, "fans": 500, "likes": 1000 }
{ "id": "5", "name": "jack", "age": 30, "fans": 130, "likes": 1300 }
{ "id": "6", "name": "crisis", "age": 30, "fans": 130, "likes": 100 }
{ "id": "7", "name": "landry", "age": 25, "fans": 130, "likes": 100 }
```

我想要尋找年紀 30 歲以上(包含 30)，但不滿 60 歲(不包含 60)，fans 又有 200 人以上(包含 200)的人

這時就需要 \$gte、\$lt 和 and 一起用囉，這有兩種寫法。

//這是第一種

```
db.user.find(  
  {"age":{"$gte":30,"$lt":60},  
  "fans":{"$gte":200}})
```

//這是第二種

```
db.user.find(  
  {"$and":[{"age":{"$gte":30,"$lt":60}},{"fans":{"$gte":200}]})
```

結果應該只找到 **steven** 這位仁兄。

我想要尋找 **fans** 小於等於 100，或是 **likes** 小於 100 的人。

這時就需要用到 **or** 和 **\$lt**、**\$lte** 囉。

```
db.user.find(  
  {"$or": [{"fans":{"$lte":100}},{"likes":{"$lt":100}]})
```

結果應該是找到三位 **mark**、**steven**、**stanly**。

我想要尋找 **age** 為 **25**、**60** 的人。

這時可用 **\$in**。

```
db.user.find({"age":{"$in":[25,60]})
```

結果如下，應該是找到三位 **mark**、**max**、**landry**。

我想要尋找 **age** 不為 **25**、**60** 的人，並且只給我它的 **id** 就好。

這時可用 **\$nin**。

```
db.user.find({"age":{"$nin":[25,60]}},{"id":1})
```

結果應該是會找到 4 位。

我想要尋找 **likes** 小於等於 **100** 的人(使用 **\$not**)

這邊事實上可以很簡單的用 **\$lte**，但因為我們要介紹一下 **\$not** 所以會寫的比較麻煩點兒，

而真正可以發揮 **\$not** 功能時，是在和正規表達式聯合使用時，用來查找不匹配的 **document**。

來解釋一下這段，首先它會尋找所有 **likes** 大於 **100** 的 **document**，但這時在配個 **\$not** 就變成完全相反會變成小於等於喔。

```
db.user.find({"likes":{"$not":{"$gt":100}}})
```

所以結果應該是找到 4 筆。

我們想要找同時不滿足 **fans** 大於 **100** 人且 **likes** 大於 **500**。

這邊我們可以用 **\$nor** 來尋找，它的意思就是選出所有不滿足條件的 **document**。

```
db.user.find({"$nor":[{"fans":{"$lt":100}},{"likes":{"$lt":100}]})
```


chap9 之 CRUD---搜尋之陣列欄位與 regex

~ 搜尋陣列內容 ~

這邊我們將要介紹幾個陣列搜尋符號 `$all`、`$size`、`$slice`。

Tables	Are
<code>\$all</code>	當需要尋找多個元素節合的 <code>document</code> 時，就可以使用它
<code>\$size</code>	當要尋找特定長度的陣列時，就可以用它~
<code>\$slice</code>	可以指定回傳的陣列指定的範例 ex. <code>10</code> 就為前十條， <code>-10</code> 就為後十條。
<code>\$elemMatch</code>	它會只針對陣列，進行多組 <code>query</code> 。

假設情況我們 `collection` 中有下列 `document`。

```
{ "id": "1", "name": "mark",  
  "fans": ["steven", "stanly", "max"],  
  "x": [10, 20, 30] };
```

```
{ "id": "2", "name": "steven",  
  "fans": ["max", "stanly"],  
  "x": [5, 6, 30] };
```

```
{ "id": "3", "name": "stanly",  
  "fans": ["steven", "max"],
```

```
"x":[15,6,30,40]];

{"id":"4","name":"max",
 "fans":["steven","stanly"],
 "x":[15,26,330,41,1]};
```

我們這時想要尋找 **fans** 中同時有 **steven**、**max** 的網紅

我們這時就可以使用 **\$all**。

```
db.user.find({"fans":{"$all":["steven","max"]}})
```

結果應該是只找到 **mark**、**stanly** 這兩個人。

```
db.user.find({"fans":{"$all":["steven","max"]}}).pretty()
```

我們想要尋找 **fans** 總共有三位的網紅。

我們這時可以用 **\$size**，不過有點可惜的一件事，**\$size** 無法與搜尋條件(**ex.\$gte**)使用，所以無法尋找 3 人以上之類的，通常要來實現這種需求就只能多加個欄位了。

我們來看看 **\$size** 的使用方法。

```
db.user.find({"fans":{"$size" :3}})
```

```
db.user.find({"fans":{"$size" :3}}).pretty()
```

我們希望尋找 **mark** 的第一個 **fans** 。

\$slice 主要功能就是將陣列切割只回傳你指定的範例。

```
db.user.find({"name":"mark"}, {"fans":{"$slice":1}})
db.user.find({"name":"mark"}, {"fans":{"$slice":1}}).pretty()
```

我們想要尋找 **x** 中至少有一個值為大於 **30** 小於 **100** 的網紅。

```
db.user.find({"x":{"$elemMatch" : {"$gt" : 30 , "$lt" : 100}}})
db.user.find({"x":{"$elemMatch" : {"$gt" : 30 , "$lt" : 100}}}).pretty()
```

~ 正規表達式搜尋 ~

mongodb 當然有提供正規表達式的搜尋，如果你正規表達式夠強，那幾乎可以直接找到你所有想要的資料。

我們想要尋找 **name** 為 **s** 開頭的網紅。

```
db.user.find({"name":/^s/})
db.user.find({"name":/^s/}).pretty()
```

結果應該會尋找到 **steven**、**stanly**。

chap10 之 CRUD---搜尋之 Cursor 運用與搜尋原理

~ Cursor 是啥 ~

`cursor` 是 `find` 時回傳的結果，它可以讓使用者對最終結果進行有效的控制，它事實上也就是 `Iterator` 模式的實作。

除了可以控制最終結果以外，它另一個好處是可以一次查看一條結果，像之前會一次回傳全部的結果，`mongodb shell` 就會自動一直輸出，結果看不到後來執行的東西。

我們實際來看一下 `cursor` 的用法，首先我們還是要先新增一些資料。

```
for (var i=0;i<1000;i++){  
    db.user.insert({x:i})  
}
```

~ Cursor 的方法 ~

`limit`、`skip`、`sort` 這三個是很常用的 `cursor` 方法，主要功能就是限制、忽略、排序。

`limit`

要限制 `find` 結果的數量可以用 `limit`，不過注意 `limit` 是指定上限而不是指定下限，使用方法如下，`limit(10)` 就是代表最多只回傳 10 筆資料。

```
db.user.find().limit(10)
```

skip

當你想要忽略前面幾筆，在開始回傳值時，就是可以用 **skip**，使用方法如下，**skip(10)**，代表忽略前十筆，然後在開始回傳，不過注意『 **skip** 如果數量很多時速度會變很慢 』。

```
db.user.find().skip(10)
```

sort

sort 它主要就是將 **find** 出的資料，根據條件，進行排序。

三個都可以一起使用

這三個條件我們都可以一起使用，例如，你希望尋找先忽略前 10 筆，並且數量限制為 50 筆，最後在進行排序，則指令如下。

```
db.user.find().skip(5).limit(50).sort({x:1})
```

~ 搜尋的原理 ~

在不考慮有索引的條件下，說如果你要找的值是放在資料的最後面，你找到的時間會最久，給個程式實驗看看。


來我們來測試看看找到{"x" : 1}和{"x": 999}速度會差多少，其中加 **limit(1)**是因為只讓它尋找第一個，如果沒限制它會一直繼續找，看還有沒有符合的，這樣兩者速度是相等的，因為都是全文掃描，而 **explain("executionStats")**是叫 **mongodb** 列出詳細的執行結果。

```
db.user.find({"x" : 1}).limit(1).explain("executionStats")
```

```
db.user.find({"x" : 999}).limit(1).explain("executionStats")
```

首先看看下圖，是 { "x" : 1 } 的，可以看到執行時間幾乎沒有，而掃描的 document 之有 2，也就是只找兩個 document 就找到 { "x" : 1 }，而那兩個就是 { "x" : 0 } 和 { "x" : 1 }。

這是尋找 { "x" : 1 } 的結果喔



```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 2,
  "executionStages" : {
    "stage" : "LIMIT",
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 0,
    "works" : 4,
    "advanced" : 1,
    "needTime" : 2,
    "needYield" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "limitAmount" : 1,
    "inputStage" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "x" : {
          "$eq" : 1
        }
      },
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 0,
      "works" : 3,
      "advanced" : 1,
      "needTime" : 2,
      "needYield" : 0,
      "saveState" : 0,
      "restoreState" : 0,
      "isEOF" : 0,
      "invalidates" : 0,
      "direction" : "forward",
      "docsExamined" : 2
    }
  }
}
```

代表該query的執行時間

總共掃描的documentt

代表全部掃描

然後我們在來看看 { "x" : 999 } 的結果，執行時間 413ms 差距和 {"x":1} 差距實在很大，而它幾乎要全部掃描完 document 才找到 { "x" : 999999 }，難怪會著麼慢。

這是 {"x":999999} 的結果喔



```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 1,
  "executionTimeMillis" : 413,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 1000000,
  "executionStages" : {
    "stage" : "LIMIT",
    "nReturned" : 1,
    "executionTimeMillisEstimate" : 350,
    "works" : 1000002,
    "advanced" : 1,
    "needTime" : 1000000,
    "needYield" : 0,
    "saveState" : 7812,
    "restoreState" : 7812,
    "isEOF" : 1,
    "invalidates" : 0,
    "limitAmount" : 1,
    "inputStage" : {
      "stage" : "COLLSCAN",
      "filter" : {
        "x" : {
          "$eq" : 999999
        }
      },
      "nReturned" : 1,
      "executionTimeMillisEstimate" : 320,
      "works" : 1000001,
      "advanced" : 1,
      "needTime" : 1000000,
      "needYield" : 0,
      "saveState" : 7812,
      "restoreState" : 7812,
      "isEOF" : 0,
      "invalidates" : 0,
      "direction" : "forward",
      "docsExamined" : 1000000
    }
  }
}
```

執行的時間

掃描數量

chap11 之索引(1)

~ 什麼是索引? ~

索引是什麼?最常見的說法是，一本字典中，你要找單字，會先去前面的索引找他在第幾頁，是的這就是索引，可以幫助我們更快速的尋找到 **document**。

~ 索引的優缺點 ~

索引竟然可以幫助我們著麼快的找到目標。

優點

- 搜尋速度更(飛)快 ~
- 在使用分組或排度時更快 ~

缺點

- 每次進行操作(新增、更新、刪除)時，都會更費時，因為也要修改索引。
- 索引需要佔據空間。

使用時機

所以根據以上的優缺點可知，不是什麼都要建立索引的，通常只有下列時機才會使用。

- 搜尋結果佔原 **collection** 越小，才越適合(下面會說明更清楚)。
- 常用的搜尋。
- 該搜尋造成性能瓶頸。
- 在經常需要排序的搜尋。
- 當索引性能大於操作性能時。

~ 索引的建立 ~

然後這時我們建立 **x** 欄位的索引。

```
db.user.ensureIndex({ "x" : 1 })
```

然後我們可以達行下列指令，來查看有沒有建立成功。

```
db.user.getIndexes()
```

結果建立成功 **x** 的索引，其中 **_id** 那個是預設的，**mongodb** 會自動幫 **objectId** 建立索引。

刪除指定的索引 **dropIndex()**

```
db.user.dropIndex ( "x_1")
```

刪除所有索引 **dropIndexes ()**

```
db.user.dropIndexes()
```

~ 索引與非索引搜尋比較 ~

在 **mongodb** 中排序是非常的耗費內存資源，如果排序時內存耗費到 **32mb**，**mongodb** 就會報錯，如果超出值，那麼必須使用索引來獲取經過排序的結果。

我們這裡建立些資料，來比較看看兩者的資源耗費不同點。

```
for (var i=0;i<1000;i++){  
    db.test.insert({  
        "x" : i  
    })  
}
```

然後建立 **x** 的索引。

```
db.test.ensureIndex({ "x" : 1 })  
db.test.getIndexes()
```

然後我們在有索引與無索引的情況下指行下列指令。

```
db.test.find({ "x" : { "$gt" : 5000 } }).sort({ "x" : -  
1 }).explain("executionStats")
```

首先來看看無索引的，可知它耗用了不少的內存，並且速度也比較慢。

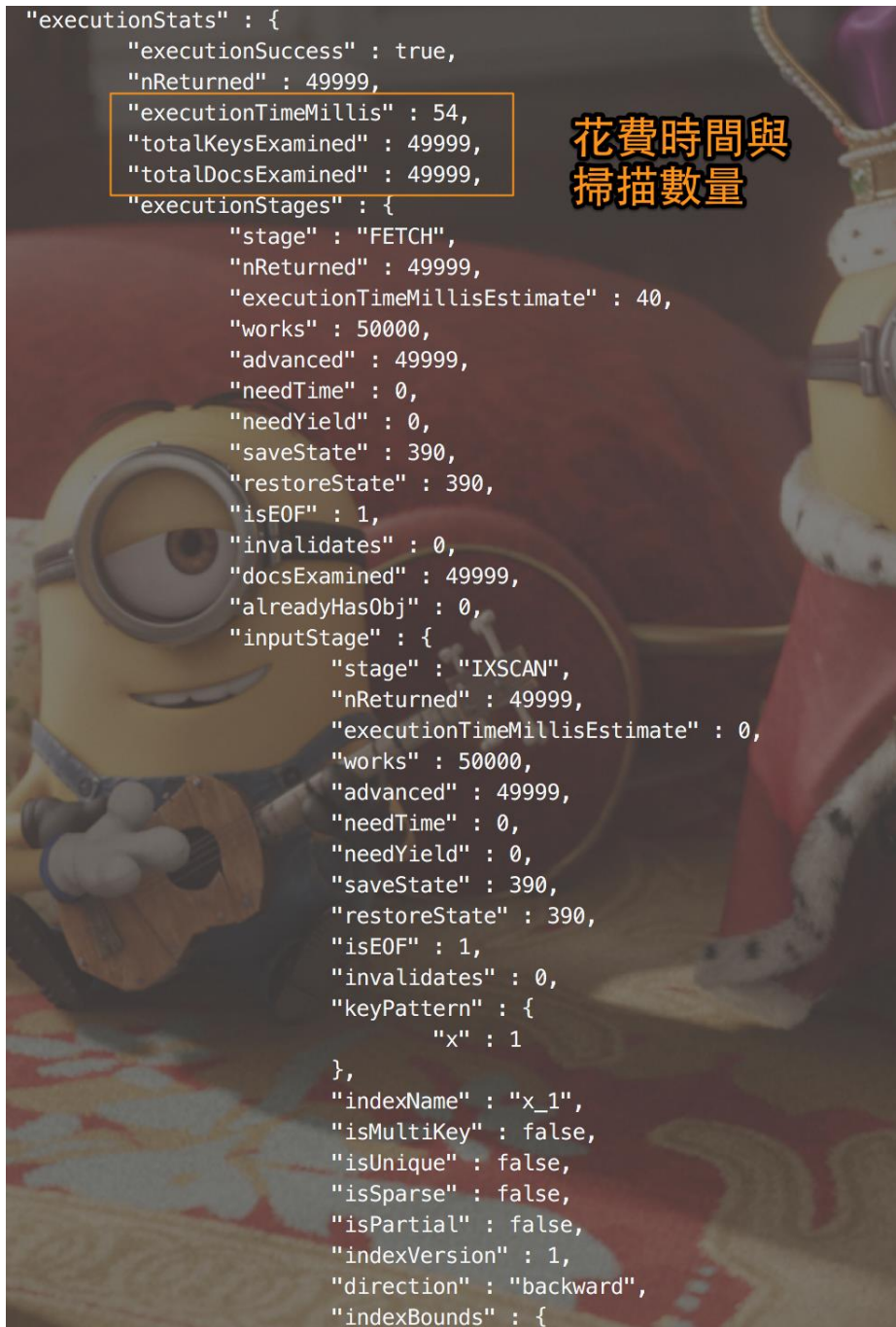


```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 49999,
  "executionTimeMillis" : 205,
  "totalKeysExamined" : 0,
  "totalDocsExamined" : 100000,
  "executionStages" : {
    "stage" : "SORT",
    "nReturned" : 49999,
    "executionTimeMillisEstimate" : 170,
    "works" : 150003,
    "advanced" : 49999,
    "needTime" : 100003,
    "needYield" : 0,
    "saveState" : 1172,
    "restoreState" : 1172,
    "isEOF" : 1,
    "invalidates" : 0,
    "sortPattern" : {
      "x" : -1
    },
    "memUsage" : 2049959,
    "memLimit" : 33554432,
    "inputStage" : {
      "stage" : "SORT_KEY_GENERATOR",
      "nReturned" : 0,
      "executionTimeMillisEstimate" : 130,
      "works" : 100003,
      "advanced" : 0,
      "needTime" : 50003,
      "needYield" : 0,
      "saveState" : 1172,
      "restoreState" : 1172,
      "isEOF" : 1,
      "invalidates" : 0,
      "inputStage" : {
        "stage" : "COLLSCAN",
        "filter" : {
          "x" : {
            "$gt" : 50000
          }
        }
      }
    }
  }
},
```

花費時間

使用內存與上限

在來看看有索引的，由於有使用到索引進行排序，所以不需要在內存中進行排序。



從上面兩張圖的結果可知有用索引的速度較快，也較省內存，但要注意並不是建立了索引就代表它一定會用索引排序，這在下一章複合索引會提到。

~ 不要使用索引的時機 ~

我們這邊將使用時機的搜尋結果佔原 collection 越小，才越適合來進行分析一下。

我們來試試結果佔原 collection 比大於 60% 會如何

我們這邊將要來驗證一下，在這種情況下，索引搜尋和全文搜尋(未使用索引)那個比較快。

首先來建立資料一百萬筆，然後有 60% 的 x 都為 1。

```
for (var i=0;i<10000;i++){
  var value = (i<6000)?"1":"2";
  db.test2.insert({
    "x" : value
  })
}
```

然後建立 x 的索引。

```
db.test2.ensureIndex({ "x" : 1 })
```

```
db.user2.find({"x" : 1}).explain("executionStats")
```

然後我們來比較看看兩者的搜尋速度，我們要尋找 x 為 1 的。

首先看看沒有用索引的速度，

在來看看有索引的速度

所以記好當你要找的結果可能會佔你原資料太多部份的，請不要用索引

~

chap12 之索引(2)---複合索引

~ 複合索引的運用~

索引建立的不好反而會浪費更多資源，以下舉個例子來說明

假設我們有以下的資料

```
{ "name" : "mark00" , age:20  }  
{ "name" : "mark01" , age:25  }  
{ "name" : "mark02" , age:10  }  
{ "name" : "mark03" , age:18  }  
{ "name" : "mark04" , age:26  }  
{ "name" : "mark05" , age:40  }  
{ "name" : "mark06" , age:51  }  
{ "name" : "mark07" , age:20  }  
{ "name" : "mark08" , age:51  }  
{ "name" : "mark00" , age:30  }  
{ "name" : "mark00" , age:100 }
```

這時我們要來思考一件事情，我們是要建立索引{ "name" : 1, "age" :1 }還是{"age":1,"name" :1 }，這兩個是不同的，記好。

首先下列索引列表為 { "name" : 1, "age" :1 }，索引的值都按一定順序排序，所以它會先依 name 的值進行排序，然後相同的 name 再按 age 進行排序。

```
db.user.ensureIndex({ "name" : 1 , "age" : 1 })
```

```
["mark00",20] -> xxxxxxxx
```

```
["mark00",30] -> xxxxxxxx
```

```
["mark00",100] -> xxxxxxxx
["mark01",25] -> xxxxxxxx
["mark02",10] -> xxxxxxxx
["mark03",18] -> xxxxxxxx
["mark04",26] -> xxxxxxxx
["mark05",40] -> xxxxxxxx
["mark06",51] -> xxxxxxxx
["mark07",20] -> xxxxxxxx
["mark08",51] -> xxxxxxxx
```

然後在來是{ "age": 1 , "name" : 1 }的索引列表。

```
db.user.ensureIndex({ "age": 1 , "name" : 1 })
```

```
[10,"mark02"] -> xxxxxxxx
[18,"mark03"] -> xxxxxxxx
[20,"mark00"] -> xxxxxxxx
[20,"mark07"] -> xxxxxxxx
[25,"mark01"] -> xxxxxxxx
[26,"mark04"] -> xxxxxxxx
[30,"mark00"] -> xxxxxxxx
[40,"mark05"] -> xxxxxxxx
[51,"mark06"] -> xxxxxxxx
[51,"mark08"] -> xxxxxxxx
[100,"mark00"] -> xxxxxxxx
```

這兩種所建立出來的索引會完全的不同，但這在搜尋時會有什麼差別呢，首先我們先來試試看下列的搜尋指令會有什麼不同。

情境 1

我們執行下列的指令來進行搜尋，主要就是先全部抓出來，然後在根據 **age** 進行排序。

```
db.user.find({}).sort({"age" : 1}).explain("executionStats")
```


首是{ "name" : 1, "age" : 1 }的索引尋找過程與執行結果，
memUsage : 660 代表有使用到內存進行排序。

執行結果

```
"executionStages" : {  
  "stage" : "SORT",  
  "nReturned" : 11,  
  "executionTimeMillisEstimate" : 0,  
  "works" : 26,  
  "advanced" : 11,  
  "needTime" : 14,  
  "needYield" : 0,  
  "saveState" : 0,  
  "restoreState" : 0,  
  "isEOF" : 1,  
  "invalidates" : 0,  
  "sortPattern" : {  
    "age" : 1  
  },  
  "memUsage" : 660,  
  "memLimit" : 33554432,  
}
```

再來看看{ "age": 1 , "name" : 1 }的執行過程與執行結果。

執行結果

```
"executionStages" : {  
  "stage" : "FETCH",  
  "nReturned" : 11,  
  "executionTimeMillisEstimate" : 0,  
  "works" : 12,  
  "advanced" : 11,  
  "needTime" : 0,  
  "needYield" : 0,  
  "saveState" : 0,  
  "restoreState" : 0,  
  "isEOF" : 1,  
  "invalidates" : 0,  
  "docsExamined" : 11,  
  "alreadyHasObj" : 0,  
}
```


是的，明明都有建立索引，但只有{ "age": 1 , "name" : 1 }有利用到索引進行排序，而另一個還是需要用到內存來進行排序，主因就在於 age 先行的索引，它本來就依照 age 的大小先排序好，而 name 先行的索引，只先排序好 name，後排序 age，但後排序的 age 只是在同樣 name 下進行排序，所以如果是找『全部』的資料再進行排序，age 先行較快。

情境 2

```
db.user.find({"name" : "mark00"}).sort({"age" : 1}).explain("executionStats")
```

先來看看 { "name" : 1, "age" :1 }的執行過程與結果，有使用索引進行尋找。

執行結果

{ "name" : 1 , "age" : 1 } 索引

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 3,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 3,
  "totalDocsExamined" : 3,
  "executionStages" : {
    "stage" : "FETCH",
    "nReturned" : 3,
    "executionTimeMillisEstimate" : 0,
    "works" : 4,
    "advanced" : 3,
    "needTime" : 0,
    "needYield" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "docsExamined" : 3,
    "alreadyHasObj" : 0,
```

再來看看{ "age": 1 , "name" : 1 }的執行結果，也有使用索引進行尋找。

執行結果

{ "age": 1 , "name" : 1 } 索引

```
"executionStats" : {
  "executionSuccess" : true,
  "nReturned" : 3,
  "executionTimeMillis" : 0,
  "totalKeysExamined" : 11,
  "totalDocsExamined" : 11,
  "executionStages" : {
    "stage" : "FETCH",
    "filter" : {
      "name" : {
        "$eq" : "mark00"
      }
    },
    "nReturned" : 3,
    "executionTimeMillisEstimate" : 0,
    "works" : 12,
    "advanced" : 3,
    "needTime" : 8,
    "needYield" : 0,
    "saveState" : 0,
    "restoreState" : 0,
    "isEOF" : 1,
    "invalidates" : 0,
    "docsExamined" : 11,
    "alreadyHasObj" : 0,
  },
}
```

從上面兩張結果可以看出，他們都有使用到索引進行搜尋與排序，但 **name** 先行的索引只花了 3 次就得出結果，而 **age** 先行的卻花 11 次才得出結果，主要原因 **name** 先行的 **name** 已經排序好，三個 **mark00** 就堆在一起，要找到全部的 **mark00** 非常快，而 **age** 先行的就要全部慢慢找，才能找出全部的 **mark00**。

chap13 – MongoDB 備份&復原

前言：

這章節將介紹[匯入匯出](#)指令與[備份還原](#)指令，雖然備份資料可以也可以透過

匯入匯出方式達到可能的效果，

但官方網站不建議這樣使用，因為匯出的 json(或其他)檔案，沒有詳細紀錄

資料類型，利用匯入的方式進行備援，可能會造成一些資料面的問題。

執行步驟：

- **匯入與匯出：**指資料匯出匯入使用，可以使用的格式有 json, csv, tsv 等

- **Export command : mongoexport**

step.1 Enter command as below (out: file path and file name)

```
mongoexport --db test --collection Currency --out Currency.json
```

- **Import Command : mongoimport**

step.1 Enter command as below (file: file path and file name)

```
mongoimport --db test --collection Currency --file Currency.json
```

- 備份與還原：

備份常用的方式如下：

1.mongodump、mongorestore：

Including BSON(指令進行備份與還原，包含 BSON 檔案，紀錄所有資料型態)

- Dump command : mongodump

- Dump a collection with parameter "out"

Step.1 Enter command as below (out: file path)

```
mongodump --collection Currency --db test --out "c:\share"
```

- Restore command : mongorestore

Step.1 Enter command as below

```
mongorestore --db test --collection Currency
```

```
c:\share\test\Currency.bson
```