

数据库对象命名参考

Database Objects Naming Guidelines

张子阳

www.tracefact.net

jimmy_dev@163.com

本文是一个参考，不是一个规范，更不是一个标准。它仅代表了我个人的观点和建议，并只考虑了通常条件下的规则，你可以根据实际情况随意修改它。

引言

编码规范是一个优秀程序员的必备素质，然而，有很多人非常注重程序中变量、方法、类的命名，却忽视了同样重要的数据库对象命名。这篇文章结合许多技术文章和资料，以及我自己的开发经验，对数据库对象的命名规则提出了一点建议，希望能为大家提供一些参考。

NOTE: 虽然这篇文章名为“数据库对象命名参考”，实际上，在这篇文章中我不仅介绍了数据库命名的规则，连带讲述了在数据库设计与开发时所需要特别注意的几个问题。

基本命名规则

表 1. 基本数据库对象命名

数据库对象	前缀	举例
表(Table)	无	Student
字段(Column)	无	Title
视图(View)	v	vActivity
存储过程(Stored procedure)	pr	prDelOrder
触发器(Trigger)	tr	trOrder_D
索引(Index)	ix_	ix_CustomerID
主键(Primary key)	pk_	pk_Admin
外键(Foreign key)	fk_	fk_Order_OrderType
Check 约束(Check Constraint)	ck_	ck_TableColumn
Unique 约束	uq_	uq_TableColumn
用户定义数据类型(User-defined data type)	udt	udtPhone
用户定义函数(User-defined function)	fn	fnDueDate

关于命名的约定

变量(T-SQL 编程中声明的变量)、过程(存储过程或触发器等)、实体(表、字段)应该根据他们所代表的实体意义和进程作用来命名：

表 2.好的命名 和 不好的命名 范例

好的命名	不好的命名
@CurrentDate	@D
@ActivityCount	@ActNum
@EquipmentType	@ET
prCalculateTotalPrice	@prRunCalc

还有一个常见的错误就是只使用面向计算机的术语，而不是面向公司业务的术语，比如 ProcessRecord 就是一个含糊不清的命名，应该使用一个进程业务描述来替换它，比如 CompleteOrder.

如果完全根据上一条的要求，那么根据业务描述的过程名可能会变得很冗长，比如下面：

```
prCountTotalAmountOfMonthlyPayments (计算每月付费的总金额)
prGetParentOrganizationalUnitName (获取上级单位名称)
```

此时则应该考虑使用缩写：

- 如果可以在字典里找到一个词的缩写，就用这个做为缩写，比如：Mon(Monday)、Dec(December)
- 可以删除单词元音(词首字母除外)和每个单词的重复字母来缩写一个单词。比如：Current = Crnt、Address = Adr、Error = Err、Average = Avg
- 不要使用有歧异的缩写(一般是语音上的歧义)。比如 b4(before)、xqt(execute)、4tran(Fortran)

表格、字段的命名：

单数表名、字段名 还是 复数表名、字段名？

可能大家很少会考虑到给表名起单数还是复数，比如，对存储客人信息的表，我们应该起 Customer，还是 Customers？我主张起单数表名，下面是来自《SQL Server 2000 宝典》的一段引用：

主张用复数表名的阵营认为：表是由一组记录构成的，所以应当使用复数名词来命名它。他们经常使用的理由是：客户表是客户们的集合，而集合意味着多个，因此应当称他们为 Customers 表。除非你只有一个客户，但这种情况你根本用不着数据库。

根据笔者的非正式调查，有 3/4 的 SQL Server 开发人员支持使用单数命名。这些开发人员认为，客户表是客户的集合，而不是客户们的集合。一组行不应当也不会被成为 rows set (行们的集合)，而会被称为 row set (行集)。并且，通常在讨论时人们会使用单数名称来称呼表，说 Customer 表比说 Customers 表听起来更为清晰。

避免无谓的表格后缀

这两点我想大家都知道：1、表是用来存储数据信息的。2、表是行的集合。那么如果表名已经能够很好地说明其包含的数据信息，就不需要再添加体现上面两点的后缀了。

实际工作中，我看到有的同事对表这样命名：GuestInfo，用于存储客户信息。这个命名与上面所说的第 1 点重复，谁都知道表本来就是存储信息(information)的，再加个 Info 无异于画蛇添足，个人认为直接用 Guest 做表名就可以了。

对于存储航班信息的表，他又命名为 FlightList。这个命名又与之前说的第 2 点相重复，表是行的集合，那么自然是列表(List)，加上 List 后缀显得很多余，命名为 Flight 不是很好么？可见，他给自己都没有订立一个明确的命名规则，不然这两个表一定是要么命名为：GuestList、FlightList 要么命名为 GuestInfo、FlightInfo，而不会是两者的混合。

多对多关系中连接表的命名

大家知道，如果要实现两个实体间的多对多关系，需要三张表，其中一张是解析表。考虑下面这样一个多对多关系，这是一个经典的学生选课问题：一个学生可以选很多门课，一门课可以有多个学生。此时为了实现上面的关系，就需要一张解析表(这张表只存储学生 ID 和课程 ID，而学生的信息和课程信息分别存在各自的表中)，**这个表的起名，建议的写法是将两个表的表名合并(如果表名比较长可做简化)，此处如 StudentCourse。**这个表中字段分别命名为 StudentId、CourseID(既是此表的复合主键，同时分别为连接 Student 表和 Course 表的外键，等下到主键和外键的命名处再说)，这样就实现了学生和课程之间的多对多关系，当然，这个关系还可以加额外的东西，比如给 StudentCourse 表中加 AccessLevel 字段，值域 D{只读, 完全, 禁止}，就可以实现访问级别。

约定俗成的字段名前/后缀

数据库开发的时间久了，慢慢就会摸索出一个规律来：就是很多的字段都有些共同的特性。比如说，有的字段是代表时间的(例如发帖时间，评论时间)，有的是代表数量的(例如浏览数，评论数)，有的是代表真假类型的(例如是否将博客随笔显示在首页)。对于这种同一类型的字段，应该使用统一的前缀 或者 后缀去标识它。

我们来举几个例子看得更明白一点。

以大家都熟悉的论坛来说，需要记录会员最后一次登录的时间，这时候一般人都会把这个字段命名为 LoginTime 或者 LoginDate。这时候，已经产生了一个歧义：对于另一名开发者来说，如果仅看表的字段名称，不去看表的内容，很容易将 LoginTime 理解成 登录的次数，因为，Time 还有一个很常用的意思，就是次数。

为了避免这种情况发生，应该明确的规定：**所有表示时间的字段，统一以 Date 来作为结尾。**

我们经常需要统计发帖数、回帖数信息，这时候，开发人员通常会这样去命名字段：PostAmount、PostTime、PostCount，同样，由于 Time 的歧义，我们首先排除掉不使用 PostTime 作为字段名。接下来，Amount 和 Count 都可以表示计数的意思，用哪个合适呢？这里，我推荐使用 Count。为什么呢？如果你做过 Asp 开发，相信一定知道 RecordCount 这个属性，命名的时候有一个原则：**就是使用约定俗成的名称，而不要去自创名称。**既然微软都用 Count 后缀来表示数目，我们为什么不呢？

于是，**所有表示数目的字段，都应该以 Count 作为结尾。**将这一概念做以推广，很容易得出，浏览次数为 ViewCount，登录次数为 LoginCount 等等。

再举一个例子，我们很少在数据库里直接保存图片等二进制数据，通常是仅保存图片的 URL 路径；在文章管理系统中，如果是转载文章，也会用到记录文章出处的字段。**个人建议所有代表链接的字段，均为 Url 结尾。**于是，图片路径的字段命名为 ImageUrl，文章出处字段的命名为 SourceUrl。

最后一个例子，我们经常需要用到布尔值，比方说，这篇随笔要不要显示到首页，这篇随笔是不是保存到草稿箱等等。**同样，按照微软的建议，布尔类型的值均以 Is、Has 或者 Can 开头。**

如果让我来建表示是否将随笔放到首页的字段，它的名字一定是这样的：IsOnIndex

类似的例子是很多的，我在这里仅举出典型的几个范例，大家可以自行拓展，如果我能起到一个抛砖引玉的作用就很满足了。

字段命名时需注意的一个问题

我发现有很多开发人员喜欢给字段加上表名作为它的前缀，举个例子，如果有个表叫 User，那么他就会将这个表中的字段命名为：UserId、UserPassword、UserName、UserPhone 等等。个人认为，这是没有必要的，因为你已经确切的知道了这个表存储的是 User 的信息，那么其中的字段必然是针对于 User 的。而且，在 Join 连接操作中，你的 SQL 代码看上去也会更加的精简一些，诸如 `[User].UserName = Article.ArticleAuthor` 这样的代码完全可以实现为 `[User].Name = Article.Author`。

这里还存在一个特例，就是表的外键包含的字段。在这种情况下，我倾向于使用 表名+ID 的方式，比如 CategoryId、UserId 等。假设有表 Article，那么它的主键我会命名为 Id，关联用户表 User 的外键包含的字段，我会命名为 UserId。之所以这样，是因为在语言(比如 C#)中创建对象时，有时候会使用代码生成器(根据数据库的字段名生成对象的字段、属性名)，此时生成的代码更规整一些。

建表时需要注意的问题

数据库不仅是用来保存数据，还应负责维护数据的完整性和一致性

我看过很多的开发人员设计出来的数据库，给我的感觉就是：在他们眼里，数据库的作用就如同它的名称一样——仅仅是用来存放数据的，除了不得不建的主键以外，什么都没有... 没有 Check 约束，没有索引，没有外键约束，没有视图，甚至没有存储过程。

在这里，我提出如下数据库设计的建议：

1. 如果要写代码来确保表中的行都是唯一的，就为表添加一个主键。
2. 如果要写代码来确保表中的一个单独的列是唯一的，就为表添加一个约束。
3. 如果要写代码确定表中的列的取值只能属于某个范围，就添加一个 Check 约束。
4. 如果要写代码来连接 父—子 表，就创建一个关系。

5. 如果要写代码来维护“一旦父表中的一行发生变化，连带变更子表中的相关行”，就启用级联删除和更新。
6. 如果要调用大量的 Join 来进行一个查询，就创建一个视图。
7. 如果要逐条的写数据库操作的语句来完成一个业务规则，就使用存储过程。

NOTE: 这里我没有提到触发器，实践证明触发器会使数据库迅速变得过于复杂，更重要的是触发器难以调试，如果不小心建了个连环触发器，就更让人头疼了，所以我更倾向于根本就不使用触发器。

以 Not Null 的思路建表

我发现很多开发人员在建表的时候，如果要新建一个字段，他的思路是这样的：默认这个字段是可以为 Null 的，然后去判断是不是非要 Not Null 不可，如果不是这样，OK，这个字段可以为 Null，接着继续进行下一个字段。结果往往是一张表除了主键以外所有的字段都可以为 Null。

之所以会有这样的思路，是因为 Null 好啊，程序不容易出错啊，你插入记录的时候如果不小心忘输了一个字段，程序依然可以 Run，而不会出现“XX 字段不能为 Null”的错误消息。

但是，这样做的结果却是很严重的，也会使你的程序变得更加繁琐，你不得不进行一些无谓的空值处理，以避免程序出错。更糟的是，如果一些重要数据，比如说订单的某一项值为 Null 了，那么大家知道，任何值与 Null 相操作(比如加减乘除)，结果都是 Null，导致的结果就是订单的总金额也为 Null。

你可以运行下面的代码尝试一下：

```
Select Null + 5 As Result
```

你可能会说，就算我将字段设置成 Not Null，但是它依然可以接受空字符串，这样一来在程序中还是要进行空值处理。请别忘了，数据库还赋予你一个强力武器，就是 Check 约束，当你需要确保一个字段既不可以为 Null，又不可以为空的时候，可以这么写：

```
ColumnName      Varchar(50)      Not Null Constraint ck_ColumnName  
Check(Len(ColumnName) > 0)
```

所以，合理的思维方式应该是这样的：默认这个字段是 Not Null 的，然后判断这个字段是不是非为 Null 不可，如果不是这样，OK，这个字段是 Not Null 的，进行下一个字段。

一个建表的范例脚本

我正在建立我自己的个人空间，其中的文章表是这样写的：

```
Create Table Article  
(
```

```

    Id            Int Identity(1,1)  Not Null,
    Title         Varchar(50)      Not Null Constraint uq_ArticleTitle Unique,
    Keywords      Varchar(50)      Not Null,
    Abstract      Varchar(500)     Not Null,
    Author        Varchar(50)     Not Null Default '张子阳',
    Type          TinyInt      Not Null Default 0 Constraint ck_ArticleType
Check(Type in (0,1,2)), -- 0,原创; 1,编译; 2,翻译
    IsOnIndex     Bit          Not Null Default 1, -- 是否显示在首页
    Content       Text         Not Null,
    SourceCode    Varchar(100)  Null,    -- 程序源码的下载路径
    Source        Varchar(50)   Not Null Default 'TraceFact', -- 文章出处
    SrcUrl        Varchar(150)  Null,    -- 文章出处的 URL
    PostDate      DateTime     Not Null Default GetDate(),
    ViewCount     Int           Not Null Default 0,
    ClassId       Int           Not Null    -- 外键包含的字段, 文章类别

    Constraint pk_Article Primary Key(Id) -- 建立主键
)

```

可以看到，在这里我使用了 Check 约束，以确保文章类型只能为 0, 1, 2。这里，我想说的是 Check 约束的命名规则：尽管 Check 约束是针对字段的，但在同一数据库中，却不能有同名的 Check 约束。所以，建议使用 ck_ + 表名 + 字段名 来命名它，比如这个范例脚本中的 ck_ArticleType。

除此以外，我还使用了 Unique 约束，以确保文章标题的唯一性。由于这是我的博客文章表，不应该出现重复的题目，这样可以避免在使用 Insert 语句时插入重复值。类似于 Check 约束，这里的命名规则是：uq_ + 表名 + 字段名。

主键的命名

按照 SQL Server 的默认规范(使用企业管理器创建主键时默认产生的主键名)，主键的命名为 pk_TableName。主键是针对一个表的，而不是针对一个字段的，大家有时候在企业管理器中会见到一个表的两个字段前面都会有钥匙的图标(比如 SQL Server 2000 自带的 NorthWind 范例数据库的 EmployeeTerritories 表)，就会误以为主键是针对字段的，即是说一个表上有两个主键，其实错了，只有一个主键，但包含了两个字段，这就是常说的复合主键。为了有个更生动的认识，看下建立复合主键的 SQL 语句，以上面说到的多对多连接表 StudentCourse 为例：

```

Alter Table StudentCourse
Add Constraint pk_StudentCourse Primary key(StudentId, CourseId)

```

可见，对于主键 pk_StudentCourse，包含了两个字段 StudentId 和 CourseId。

外键的命名

外键的命名为 **fk_外键所在的表名_外键引用的表名**。因为外键所在的表为从表，所以上式可以写为 **fk_从表名_主表名**。

外键包含的字段的命名，外键包含的字段和外键是完全不同的概念。外键包含字段的命名，建议为：**外键所在的表名 + Id**。

考虑这样一个关系，表 Hotel，字段 Id, Name, CityId。表 City, 字段 Id, Name。因为一个城市可能有好多家酒店，所以是一个一对多的关系，City 是主表(1方)，Hotel 是从表(多方)。在 Hotel 表中，CityId 是做为外键使用。

在实现外键的时候我们可以这样写：

```
Alter Table HotelInfo
Add Constraint fk_HotelInfo_City Foreign Key (CityID) References City(ID)
On Delete No Action On update No Action
```

很明显，fk_HotelInfo_City 是外键的名字，CityId 是外键包含的字段的名字。

NOTE: 在创建数据库表的时候，一般需要写成三个 SQL 脚本文件。第一个文件仅包含所有的创建表的 SQL 语句，即 Create Table 语句。第二个文件包含删除关系和表的语句，其中，所有删除关系的语句，即 Drop Constraint 语句集中在这个文件的上半部分，所有删除表的语句，Drop Table 语句，集中在这个文件的下半部分。第三个文件包含建立表之间关系的语句。这种做法会在你移植数据库的时候产生较大的便利，原因我就不解释了，您一试便知。

而对于多对多关系中解析表的外键包含的字段，顺理往下推，我们可以这样写(再次回到学生选课的多对多例子中)：

建立解析表 StudentCourse 与 Student 表的外键关系：

```
Alter Table StudentCourse
Add Constraint fk_StudentCourse_Student Foreign Key (StudentId) References Student
(Id)
On Delete No Action On Update No Action
```

建立解析表 StudentCourse 与 Course 表的外键关系：

```
Alter Table StudentCourse
Add Constraint fk_StudentCourse_Course Foreign Key (CourseId) References Course(Id)
On Delete No Action On Update No Action
```

触发器的命名

由三部分构成：

1. 前缀(tr), 描述了数据库对象的类型。
2. 基本部分, 描述触发器所加的表。
3. 后缀(_I、_U、_D), 显示了修改语句(Insert, Update 及 Delete)

存储过程的命名

大家知道, 系统存储过程的前缀是 `sp_`, 为了避免将用户存储过程与系统存储过程混淆, 这里我推荐大家使用 `pr` 作为自己定义的存储过程的命名。

同时, 命名的规则是: 采用自解释型的命名, 比如: `prGetItemById`。

这里, 有个有意思的地方值得深思。我们按上面规则命名存储过程的时候, 可以用两种方式:

1. 动词放前面, 名词放后面。
2. 名词放前面, 动词放后面。

我个人推荐使用方式 2, 现在说说原因:

以 `NorthWind` 为例, 假如对于 `Employees` 表你有 4 个存储过程, 分别命名为: `prEmployeeInsert`、`prEmployeeUpdate`、`prEmployeeDelById`、`prEmployeeGetById`

同时对于 `Products` 表你有类似的 4 个存储过程, 分别命名为: `prProductInsert`、`prProductUpdate`、`prProductDelById`、`prProductGetById`

这时, 你用企业管理器查看时, 会发现存储过程像下面这样整整齐齐的排列:

```
prEmployeeDelById
prEmployeeGetById
prEmployeeInsert
prEmployeeUpdate
prProductDelById
prProductGetById
prProductInsert
prProductUpdate
```

很容易就会发现, 当你的存储过程越多时, 这种命名方法的优势就越明显。

存储过程中参数的命名

存储过程中的入口参数,我建议与其对应的字段名相同,这里,假设要写一个更新 Northwind 数据库 Employees 表的存储过程(做了简化),可以这么写:

```
Create Procedure prEmployeeUpdateById
    @EmployeeId      Int,
    @LastName         NVarchar(20),
    @FirstName        NVarchar(10)
As
Update Employees Set
    LastName = @LastName,
    FirstName = @FirstName
Where
    EmployeeId = @EmployeeId

If @@error <> 0 or @@RowCount = 0
    Raiserror 16001 '更新用户失败'
```

总结

在这篇文章中,我首先提出了开发人员对数据库对象命名不够重视的问题,随后列出了一张数据对象命名的简表。

接着我按照 表、字段、主键、外键、触发器、存储过程的顺序,详细讲述了数据库对象命名的规则。

其间,我还穿插着讲述了在数据库开发中常见的一些问题,包括建表时需要注意的问题,以及在管理存储过程时可以采取的技巧。

希望这篇文章能给你带来帮助。