1.教程

本教程通过示例介绍**MongoEngine——我们将逐步介绍如何创建一个简单的Tumblelog**应用程序。tumblelog 是一种支持混合媒体内容的博客，包括文本、图像、链接、视频、音频等。为简单起见，我们将坚持使用文本、图像和链接条目。由于本教程的目的是介绍 MongoEngine，我们将重点关注应用程序的数据建模方面，而忽略用户界面。

1.1. 入门

在我们开始之前，确保 MongoDB 的副本正在可访问的位置运行——在本地运行它会更容易，但如果这不是一个选项，那么它可能会在远程服务器上运行。如果您还没有安装 MongoEngine，只需像这样使用 pip 安装它：

```
$ python -m pip install mongoengine
```

在我们开始使用 MongoEngine 之前，我们需要告诉它如何连接到我们的**mongod**实例。为此，我们使用`connect()`函数。如果在本地运行，我们需要提供的唯一参数是要使用的 MongoDB 数据库的名称：

```
from mongoengine import *

connect('tumblelog')
```

有很多连接到 MongoDB 的选项，有关它们的更多信息，请参阅连接到 MongoDB指南。

1.2. 定义我们的文档

MongoDB 是*无模式的*，这意味着数据库不强制执行任何模式——我们可以根据需要添加和删除字段，MongoDB 不会抱怨。这使生活在许多方面变得更加轻松，尤其是在数据模型发生变化时。然而，为我们的文档定义模式可以帮助消除涉及不正确类型或缺失字段的错误，并且还允许我们以与传统 ORM 相同的方式在我们的文档上定义实用 方法。

在我们的 Tumblelog 应用程序中，我们需要存储几种不同类型的信息。我们需要有一个**users**集合，这样我们就可以将帖子链接到个人。我们还需要 在数据库中存储我们不同类型的**帖子（例如：文本、图像和链接）**。为了帮助导航我们的 Tumblelog，帖子可能有与之关联的**标**

签，因此向用户显示的帖子列表可能仅限于已分配特定标签的帖子。最后，如果可以在帖子中添加**评论**就好了。**我们将从users**开始，因为涉及的其他文档模型稍微多一些。

1.2.1. 用户

就像我们使用带有 ORM 的关系数据库一样，我们需要定义 a `User` 可能有哪些字段，以及它们可能存储什么类型的数据：

```
class User(Document):
    email = StringField(required=True)
    first_name = StringField(max_length=50)
    last_name = StringField(max_length=50)
```

这看起来类似于在常规 ORM 中定义表结构的方式。关键的区别是这个模式永远不会传递给 MongoDB——这只会在应用程序级别强制执行，使未来的更改易于管理。此外，用户文档将存储在 MongoDB*集合*中而不是表中。

1.2.2. 帖子、评论和标签

现在我们将考虑如何存储其余信息。如果我们使用关系数据库，我们很可能会有一个**帖子表、一个评论**表和一个**标签**表。为了将评论与个别帖子相关联，我们将在评论表中放置一列，其中包含帖子表的外键。我们还需要一个链接表来提供帖子和标签之间的多对多关系。然后我们需要解决存储专用帖子类型（文本、图像和链接）的问题。我们可以通过多种方式实现这一目标，但每种方式都有其自身的问题——它们都不是特别直观的解决方案。

1.2.2.1。帖子

幸运的是 MongoDB*不是*关系数据库，所以我们不会那样做。事实证明，我们可以利用 MongoDB 的无模式特性为我们提供更好的解决方案。我们将把所有的帖子存储在*一个集合*中，每个帖子类型将只存储它需要的字段。如果我们稍后想要添加视频帖子，我们根本不必修改集合，我们只需*开始使用*支持视频帖子所需的新字段即可。这很好地符合面向对象的*继承*原则。我们可以将 `Post` 和 视为基类，`TextPost` 并将 `ImagePost` 其 `LinkPost` 视为子类 `Post`. 事实上，MongoEngine 支持这种开箱即用的建模——您需要做的就是通过在以下项中设置 `allow_inheritance` 为 True 来打开继承 `meta`：

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)

    meta = {'allow_inheritance': True}

class TextPost(Post):
    content = StringField()

class ImagePost(Post):
    image_path = StringField()

class LinkPost(Post):
    link_url = StringField()
```

我们正在使用对象存储对帖子作者的引用 `ReferenceField`。这些类似于传统 ORM 中的外键字段，在保存时自动转换为引用，并在加载时解引用。

1.2.2.2。标签

现在我们已经弄清楚了我们的 Post 模型，我们将如何为它们附加标签？MongoDB 允许我们在本地存储项目列表，因此我们可以在每个帖子中存储标签列表而不是链接表。因此，为了效率和简单起见，我们将标签作为字符串直接存储在帖子中，而不是将对标签的引用存储在单独的集合中。特别是由于标签通常很短（通常甚至比文档的 ID 还短），这种反规范化不会对数据库的大小产生很大影响。我们来看看我们修改后的 `Post` 类的代码：

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
```

`ListField` 用于定义 Post 标签的对象将字段对象作为其第一个参数——这意味着您可以拥有任何类型字段（包括列表）的列表。

> **笔记**
>
> 我们不需要修改专门的帖子类型，因为它们都继承自 `Post`.

1.2.2.3. 评论

评论通常与一个帖子相关联。在关系数据库中，要显示带有评论的帖子，我们必须从数据库中检索帖子，然后再次查询数据库以获取与帖子关联的评论。这可行，但没有真正的理由将评论与其关联的帖子分开存储，除了绕过关系模型。*使用 MongoDB，我们可以将评论作为嵌入文档列表* 直接存储在帖子文档中。嵌入文档的处理方式与常规文档没有区别；它只是在数据库中没有自己的集合。使用 MongoEngine，我们可以定义嵌入式文档的结构以及实用方法，其方式与处理常规文档的方式完全相同：

```
class Comment(EmbeddedDocument):
    content = StringField()
    name = StringField(max_length=120)
```

然后我们可以在我们的帖子文档中存储评论文档列表：

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User)
    tags = ListField(StringField(max_length=30))
    comments = ListField(EmbeddedDocumentField(Comment))
```

1.2.2.4。处理引用的删除

如果引用被删除，该`ReferenceField`对象采用关键字 *reverse_delete_rule*来处理删除规则。要在删除用户后删除所有帖子，请设置规则：

```
class Post(Document):
    title = StringField(max_length=120, required=True)
    author = ReferenceField(User, reverse_delete_rule=CASCADE)
    tags = ListField(StringField(max_length=30))
    comments = ListField(EmbeddedDocumentField(Comment))
```

`ReferenceField`有关详细信息，请参阅。

笔记

MapFields 和 DictFields 目前不支持自动处理已删除的引用

1.3. 将数据添加到我们的 Tumblelog

现在我们已经定义了文档的结构，让我们开始向数据库添加一些文档。首先，我们需要创建一个`User`对象：

```
ross = User(email='ross@example.com', first_name='Ross', last_name='Lawley').save()
```

笔记

我们也可以使用属性语法定义我们的用户：

```
ross = User(email='ross@example.com')
ross.first_name = 'Ross'
ross.last_name = 'Lawley'
ross.save()
```

将另一个用户分配给一个名为 的变量`john`，就像我们在上面对 所做的那样 `ross`。

现在我们已经在数据库中获得了我们的用户，让我们添加一些帖子：

```
post1 = TextPost(title='Fun with MongoEngine', author=john)
post1.content = 'Took a look at MongoEngine today, looks pretty cool.'
post1.tags = ['mongodb', 'mongoengine']
post1.save()

post2 = LinkPost(title='MongoEngine Documentation', author=ross)
post2.link_url = 'http://docs.mongoengine.com/'
post2.tags = ['mongoengine']
post2.save()
```

笔记

如果您更改已保存的对象上的字段然后`save()`再次调用，文档将被更新。

1.4. 访问我们的数据

现在我们的数据库中有几篇文章，我们如何显示它们？每个文档类（即直接或间接继承自 的任何类`Document`）都有一个`objects`属性，用于访问与该类关联的数据库集合中的文档。那么让我们看看如何获取帖子的标题：

```
for post in Post.objects:
    print(post.title)
```

1.4.1. 检索特定于类型的信息

这将打印我们帖子的标题，每行一个。但是，如果我们想要访问特定于类型的数据（link_url、内容等）怎么办？一种方法是简单地使用 的`objects`子类的属性`Post`：

```
for post in TextPost.objects:
    print(post.content)
```

使用 TextPost 的`objects`属性只返回使用创建的文档`TextPost`。实际上，这里有一个更通用的规则：`objects`任何子类的属性`Document`只查找使用该子类或其子类之一创建的文档。

那么我们如何显示我们所有的帖子，只显示对应于每个帖子的特定类型的信息呢？有一种比单独使用每个子类更好的方法。当我们之前使用`Post`的`objects`属性时，返回的对象实际上并不是 的实例——它们是与帖子类型匹配的`Post`子类的实例 。`Post`让我们看看它在实践中是如何工作的：

```
for post in Post.objects:
    print(post.title)
    print('=' * len(post.title))

    if isinstance(post, TextPost):
        print(post.content)

    if isinstance(post, LinkPost):
        print('Link: {}'.format(post.link_url))
```

这将打印每个帖子的标题，如果是文本帖子则打印内容，如果是链接帖子则打印“Link: <url>”。

1.4.2. 按标签搜索我们的帖子

`objects`a的属性其实`Document`就是一个`QuerySet`对象。这仅在您需要数据时才懒惰地查询数据库。它也可能被过滤以缩小您的查询范围。让我们调整我们的查询，以便只返回带有标签“mongodb”的帖子：

```
for post in Post.objects(tags='mongodb'):
    print(post.title)
```

对象上还有一些方法`QuerySet`允许返回不同的结果，例如，调用`first()`属性`objects`将返回单个文档，第一个与您提供的查询匹配。聚合函数也可以用在`QuerySet`对象上：

```
num_posts = Post.objects(tags='mongodb').count()
print('Found {} posts with tag "mongodb"'.format(num_posts))
```

1.4.3. 进一步了解 MongoEngine

如果你走到这一步，你已经有了一个很好的开始，干得好！MongoEngine 之旅的下一步是完整的用户指南，您可以在其中深入了解如何使用 MongoEngine 和 MongoDB。