2.10. 文件迁移

您的文档结构及其关联的 mongoengine 模式可能会在应用程序的生命周期内发生变化。本节提供有关如何处理迁移的指导和建议。

由于 mongodb 非常灵活的特性，模型的迁移并不是微不足道的，对于了解*sqlalchemy*的*alembic*的人来说，不幸的是没有等效的库可以自动管理 mongoengine 的迁移。

2.10.1. 示例 1：添加字段

让我们从一个简单的模型更改示例开始，并回顾一下您必须处理迁移的不同选项。

假设我们从以下模式开始并保存一个实例：

```
class User(Document):
    name = StringField()

User(name="John Doe").save()

# print the objects as they exist in mongodb
print(User.objects().as_pymongo())    # [{u'_id': ObjectId('5d06b9c3d7c1f18db3e7c874'),
u'name': u'John Doe'}]
```

在您的应用程序的下一个版本中，现在让我们假设一个新的*enabled* `User` 字段被添加到具有*default=True*的现有模型中。因此，您只需将 `User` 类更新为以下内容：

```
class User(Document):
    name = StringField(required=True)
    enabled = BooleanField(default=True)
```

在不应用任何迁移的情况下，我们现在将一个对象从数据库重新加载到类中 `User` 并检查其*启用*属性：

```
assert User.objects.count() == 1
user = User.objects().first()
assert user.enabled is True
assert User.objects(enabled=True).count() == 0    # uh?
assert User.objects(enabled=False).count() == 0   # uh?

# this is consistent with what we have in the database
# in fact, 'enabled' does not exist
print(User.objects().as_pymongo().first())    # {u'_id':
ObjectId('5d06b9c3d7c1f18db3e7c874'), u'name': u'John'}
assert User.objects(enabled=None).count() == 1
```

如您所见，即使文档未更新，mongoengine 在将 pymongo dict 加载到实例中时也会无缝应用默认值`User`。乍一看，添加新字段时似乎不需要迁移现有文档，但这实际上会导致查询时出现不一致。

事实上，在查询时，mongoengine 并没有尝试考虑新字段的默认值，因此如果您实际上没有迁移现有文档，您将面临查询/更新将丢失相关记录的风险。

添加字段/修改默认值时，您可以使用以下任何一种作为独立脚本进行迁移：

```
# Use mongoengine to set a default value for a given field
User.objects().update(enabled=True)
# or use pymongo
user_coll = User._get_collection()
user_coll.update_many({}, {'$set': {'enabled': True}})
```

2.10.2. 示例 2：继承更改

让我们考虑以下示例：

```
class Human(Document):
    name = StringField()
    meta = {"allow_inheritance": True}

class Jedi(Human):
    dark_side = BooleanField()
    light_saber_color = StringField()

Jedi(name="Darth Vader", dark_side=True, light_saber_color="red").save()
Jedi(name="Obi Wan Kenobi", dark_side=False, light_saber_color="blue").save()

assert Human.objects.count() == 2
assert Jedi.objects.count() == 2

# Let's check how these documents got stored in mongodb
print(Jedi.objects.as_pymongo())
# [
#   {'_id': ObjectId('5fac4aaaf61d7fb06046e0f9'), '_cls': 'Human.Jedi', 'name': 'Darth
Vader', 'dark_side': True, 'light_saber_color': 'red'},
#   {'_id': ObjectId('5fac4ac4f61d7fb06046e0fa'), '_cls': 'Human.Jedi', 'name': 'Obi
Wan Kenobi', 'dark_side': False, 'light_saber_color': 'blue'}
# ]
```

如您所见，当您使用继承时，MongoEngine 会在后台存储一个名为“_cls”的字段来跟踪 Document 类。

现在让我们假设您想要重构继承模式并且： - 让 Jedi 的 dark_side=True/False 成为 GoodJedi 的/DarkSith - 摆脱 'dark_side' 字段

移动到以下模式：

```
# unchanged
class Human(Document):
    name = StringField()
    meta = {"allow_inheritance": True}

# attribute 'dark_side' removed
class GoodJedi(Human):
    light_saber_color = StringField()

# new class
class BadSith(Human):
    light_saber_color = StringField()
```

MongoEngine 不知道更改或如何将它们与现有数据映射，因此如果您不应用任何迁移，您将观察到一种奇怪的行为，就好像集合突然变空了一样。

```
# As a reminder, the documents that we inserted
# have the _cls field = 'Human.Jedi'

# Following has no match
# because the query that is used behind the scene is
# filtering on {'_cls': 'Human.GoodJedi'}
assert GoodJedi.objects().count() == 0

# Following has also no match
# because it is filtering on {'_cls': {'$in': ('Human', 'Human.GoodJedi',
'Human.BadSith')}}
# which has no match
assert Human.objects.count() == 0
assert Human.objects.first() is None

# If we bypass MongoEngine and make use of underlying driver (PyMongo)
# we can see that the documents are there
humans_coll = Human._get_collection()
assert humans_coll.count_documents({}) == 2
# print first document
print(humans_coll.find_one())
# {'_id': ObjectId('5fac4aaaf61d7fb06046e0f9'), '_cls': 'Human.Jedi', 'name': 'Darth
Vader', 'dark_side': True, 'light_saber_color': 'red'}
```

如您所见，第一个明显的问题是我们需要根据“dark_side”文档的现有值修改“_cls”值。

```
humans_coll = Human._get_collection()
old_class = 'Human.Jedi'
good_jedi_class = 'Human.GoodJedi'
bad_sith_class = 'Human.BadSith'
humans_coll.update_many({'_cls': old_class, 'dark_side': False}, {'$set': {'_cls':
good_jedi_class}})
humans_coll.update_many({'_cls': old_class, 'dark_side': True}, {'$set': {'_cls':
bad_sith_class}})
```

现在让我们检查 MongoEngine 中的查询是否有所改进：

```
assert GoodJedi.objects().count() == 1  # Hoorah!
assert BadSith.objects().count() == 1   # Hoorah!
assert Human.objects.count() == 2       # Hoorah!

# let's now check that documents load correctly
jedi = GoodJedi.objects().first()
# raises FieldDoesNotExist: The fields "{'dark_side'}" do not exist on the document
"Human.GoodJedi"
```

事实上，我们只负责重命名 _cls 值，但我们没有删除 GoodJedi 和 BadSith 模型中不再存在的“dark_side”字段。让我们从集合中删除该字段：

```
humans_coll = Human._get_collection()
humans_coll.update_many({}, {'$unset': {'dark_side': 1}})
```

笔记

为了示例，我们分 2 个不同的步骤进行了此迁移，但它可以与 _cls 字段的迁移结合使用：

```
humans_coll.update_many(
    {'_cls': old_class, 'dark_side': False},
    {
        '$set': {'_cls': good_jedi_class},
        '$unset': {'dark_side': 1}
    }
)
```

并验证文档现在是否正确加载：

```
jedi = GoodJedi.objects().first()
assert jedi.name == "Obi Wan Kenobi"

sith = BadSith.objects().first()
assert sith.name == "Darth Vader"
```

处理此迁移的另一种方法是遍历文档并逐个更新/替换它们。这速度较慢，但通常对文档模型的复杂迁移很有用。

```
for doc in humans_coll.find():
    if doc['_cls'] == 'Human.Jedi':
        doc['_cls'] =  'Human.BadSith' if doc['dark_side'] else 'Human.GoodJedi'
        doc.pop('dark_side')
        humans_coll.replace_one({'_id': doc['_id']}, doc)
```

警告

如果您在迭代时修改文档，请注意此缺陷

2.10.3. 示例 4：删除索引

如果您从 Document 类中删除索引，或从 Document 类中删除索引字段，则需要手动删除相应的索引。MongoEngine 不会为你做那件事。

处理这种情况的方法是使用index_information()确定要删除的索引的名称，然后使用drop_index()删除它

例如，假设您从以下文档类开始

```
class User(Document):
    name = StringField(index=True)

    meta = {"indexes": ["name"]}

User(name="John Doe").save()
```

一旦您开始与 Document 集合交互（在这种情况下调用*.save()时），它就会创建以下索引：*

```
print(User._get_collection().index_information())
# {
#  '_id_': {'key': [('_id', 1)], 'v': 2},
#  'name_1': {'background': False, 'key': [('name', 1)], 'v': 2},
# }
```

因此：'_id' 是默认索引，'name_1' 是我们的自定义索引。如果要删除“名称”字段或其索引，则必须调用：

```
User._get_collection().drop_index('name_1')
```

笔记

添加新字段或新索引时，MongoEngine 将负责创建它们（除非禁用*auto_create_index* ）

2.10.4. 建议

每当您对模型模式进行更改时编写迁移脚本

使用`DynamicDocument`or可能有助于避免某些迁移或让您的应用程序的 2 个版本共存。`meta = {"strict": False}`

编写后处理检查以验证迁移脚本是否有效。见下文

2.10.5. 后处理检查

以下方法可用于在应用迁移后检查文档集合的健全性。它不会对迁移的内容做任何假设，它会随机获取 1000 个对象并对文档运行一些快速检查以确保文档看起来正常。事实上，它会在第一次出现错误时失败，但这可以根据您的需要进行调整。

```python
def get_random_oids(collection, sample_size):
    pipeline = [{"$project": {'_id': 1}}, {"$sample": {"size": sample_size}}]
    return [s['_id'] for s in collection.aggregate(pipeline)]

def get_random_documents(DocCls, sample_size):
    doc_collection = DocCls._get_collection()
    random_oids = get_random_oids(doc_collection, sample_size)
    return DocCls.objects(id__in=random_oids)

def check_documents(DocCls, sample_size):
    for doc in get_random_documents(DocCls, sample_size):
        # general validation (types and values)
        doc.validate()

        # load all subfields,
        # this may trigger additional queries if you have ReferenceFields
        # so it may be slow
        for field in doc._fields:
            try:
                getattr(doc, field)
            except Exception:
                LOG.warning(f"Could not load field {field} in Document {doc.id}")
                raise

check_documents(Human, sample_size=1000)
```