2.3. 定义文件

在 MongoDB 中，**文档**大致相当于RDBMS 中的**一行。**使用关系数据库时，行存储在**表**中，表具有行遵循的严格**模式。**MongoDB 将文档存储在 **集合**中而不是表中——主要区别在于没有在数据库级别强制执行模式。

2.3.1. 定义文档的架构

MongoEngine 允许您为文档定义模式，因为这有助于减少编码错误，并允许在可能存在的字段上定义实用方法。

要为文档定义架构，请创建一个继承自 `Document`. **通过将字段对象**作为类属性添加到文档类来指定字段：

```
from mongoengine import *
import datetime

class Page(Document):
    title = StringField(max_length=200, required=True)
    date_modified = DateTimeField(default=datetime.datetime.utcnow)
```

由于 BSON（用于在 mongodb 中存储数据的二进制格式）依赖于顺序，因此文档根据其字段顺序进行序列化。

2.3.2. 动态文档模式

MongoDB 的好处之一是集合的动态模式，同时应该计划和组织数据（毕竟显式比隐式好！）在某些情况下需要动态/扩展样式的文档。

`DynamicDocument`文档的工作方式相同， `Document`但设置给它们的任何数据/属性也将被保存

```
from mongoengine import *

class Page(DynamicDocument):
    title = StringField(max_length=200, required=True)

# Create a new page and add tags
>>> page = Page(title='Using MongoEngine')
>>> page.tags = ['mongodb', 'mongoengine']
>>> page.save()

>>> Page.objects(tags='mongoengine').count()
>>> 1
```

笔记

动态文档有一个警告：字段不能以_*开头*

动态字段按创建顺序存储在任何已声明字段*之后*。

2.3.3. 领域

默认情况下，不需要字段。要使字段成为必填字段，请将 `required` 字段的关键字参数设置为 `True`. 字段也可能具有可用的验证约束（例如 `max_length` 在上面的示例中）。字段也可以采用默认值，如果未提供值，将使用默认值。默认值可以可选地是一个可调用的，它将被调用以检索值（例如在上面的示例中）。可用的字段类型如下：

`BinaryField`

`BooleanField`

`ComplexDateTimeField`

`DateTimeField`

`DecimalField`

`DictField`

`DynamicField`

`EmailField`

`EmbeddedDocumentField`

`EmbeddedDocumentListField`

`EnumField`

`FileField`

`FloatField`

`GenericEmbeddedDocumentField`

`GenericReferenceField`

`GenericLazyReferenceField`

`GeoPointField`

`ImageField`

`IntField`

`ListField`

`LongField`

`MapField`

`ObjectIdField`

`ReferenceField`

`LazyReferenceField`

`SequenceField`

`SortedListField`

`StringField`

`URLField`

`UUIDField`

`PointField`

`LineStringField`

`PolygonField`

`MultiPointField`

`MultiLineStringField`

`MultiPolygonField`

2.3.3.1. 字段参数

每个字段类型都可以通过关键字参数进行自定义。可以在所有字段上设置以下关键字参数：

`db_field` **（默认值：无）**

MongoDB 字段名称。

如果设置，MongoDB 中的操作将使用此值而不是类属性执行。

这允许您使用与 MongoDB 中使用的字段名称不同的属性。

```
from mongoengine import *

class Page(Document):
    page_number = IntField(db_field="pageNumber")

# Create a Page and save it
Page(page_number=1).save()

# How 'pageNumber' is stored in MongoDB
Page.objects.as_pymongo() # [{'_id': ObjectId('629dfc45ee4cc407b1586b1f'), 'pageNumber':
1}]

# Retrieve the object
page: Page = Page.objects.first()

print(page.page_number)  # prints 1

print(page.pageNumber) # raises AttributeError
```

笔记

如果设置，在字典中定义索引时使用属性的名称`meta`而不是`db_field`否则，`LookUpError`将被引发。

`required`（默认值：假）

如果设置为 True 并且未在文档实例上设置该字段，则`ValidationError`在验证文档时将引发 a 。

`default`（默认值：无）

未为此字段设置值时使用的值。

默认参数的定义遵循Python 的一般规则`ListField`，这意味着在处理默认可变对象（如 in或`DictField`）时应注意一些事项：

```
class ExampleFirst(Document):
    # Default an empty list
    values = ListField(IntField(), default=list)

class ExampleSecond(Document):
    # Default a set of values
    values = ListField(IntField(), default=lambda: [1,2,3])

class ExampleDangerous(Document):
    # This can make an .append call to  add values to the default (and all the following
objects),
    # instead to just an object
    values = ListField(IntField(), default=[1,2,3])
```

笔记

取消设置具有默认值的字段将恢复为默认值。

`unique` **（默认值：假）**

当为 True 时，集合中的任何文档都不会具有与此字段相同的值。

`unique_with` **（默认值：无）**

与此字段一起使用的字段名称（或字段名称列表）不会在集合中有两个具有相同值的文档。

`primary_key` **（默认值：假）**

为 True 时，将此字段用作集合的主键。 *DictField* 和*EmbeddedDocuments*都支持作为文档的主键。

笔记

如果设置，该字段也可以通过*pk*字段访问。

`choices` **（默认值：无）**

一个可迭代的（例如列表、元组或集合）选项，该字段的值应限制在该选项中。

可以是嵌套的值元组（存储在 mongo 中）和人类可读的键

```
SIZE = (('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
        ('XL', 'Extra Large'),
        ('XXL', 'Extra Extra Large'))


class Shirt(Document):
    size = StringField(max_length=3, choices=SIZE)
```

或者一个只包含值的平面迭代器

```
SIZE = ('S', 'M', 'L', 'XL', 'XXL')

class Shirt(Document):
    size = StringField(max_length=3, choices=SIZE)
```

`validation` **（选修的）**

用于验证字段值的可调用对象。可调用对象将值作为参数，如果验证失败则应引发 ValidationError

例如

```
def _not_empty(val):
    if not val:
        raise ValidationError('value can not be empty')

class Person(Document):
    name = StringField(validation=_not_empty)
```

`**kwargs` （选修的）

您可以提供额外的元数据作为任意额外的关键字参数。但是，您不能覆盖现有属性。常见的选择包括*help_text*和*verbose_name*，通常由表单和小部件库使用。

2.3.3.2. 列出字段

MongoDB 允许存储项目列表。要将项目列表添加到 a `Document`，请使用`ListField`字段类型。`ListField`将另一个字段对象作为它的第一个参数，它指定哪些类型的元素可以存储在列表中：

```
class Page(Document):
    tags = ListField(StringField(max_length=50))
```

2.3.3.3. 嵌入文档

MongoDB 能够将文档嵌入到其他文档中。可以为这些嵌入式文档定义模式，就像它们可以为常规文档定义一样。要创建嵌入式文档，只需像往常一样定义文档，但继承自`EmbeddedDocument`而不是`Document`：

```
class Comment(EmbeddedDocument):
    content = StringField()
```

要将文档嵌入另一个文档，请使用`EmbeddedDocumentField`字段类型，提供嵌入的文档类作为第一个参数：

```
class Page(Document):
    comments = ListField(EmbeddedDocumentField(Comment))

comment1 = Comment(content='Good work!')
comment2 = Comment(content='Nice article!')
page = Page(comments=[comment1, comment2])
```

嵌入式文档还可以利用动态文档模式的灵活性： 通过继承`DynamicEmbeddedDocument`.

2.3.3.4. 字典字段

通常，可以使用嵌入式文档代替字典——通常建议使用嵌入式文档，因为字典不支持验证或自定义字段类型。然而，有时您并不知道要存储的内容的结构；在这种情况下 a `DictField`是合适的：

```
class SurveyResponse(Document):
    date = DateTimeField()
    user = ReferenceField(User)
    answers = DictField()

survey_response = SurveyResponse(date=datetime.utcnow(), user=request.user)
response_form = ResponseForm(request.POST)
survey_response.answers = response_form.cleaned_data()
survey_response.save()
```

字典可以存储复杂的数据、其他字典、列表、对其他对象的引用，因此是最灵活的可用字段类型。

2.3.3.5。参考字段

引用可以存储到数据库中的其他文档，使用 `ReferenceField`. 传入另一个文档类作为构造函数的第一个参数，然后简单地将文档对象分配给该字段：

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

class Page(Document):
    content = StringField()
    author = ReferenceField(User)

john = User(name="John Smith")
john.save()

post = Page(content="Test Page")
post.author = john
post.save()
```

该 `User` 对象在后台自动变成引用，并在 `Page` 检索对象时取消引用。

要添加引用 `ReferenceField` 正在定义的文档的 ，请使用字符串 `'self'` 代替文档类作为 的 `ReferenceField` 构造函数的参数。要引用尚未定义的文档，请使用未定义文档的名称作为构造函数的参数：

```
class Employee(Document):
    name = StringField()
    boss = ReferenceField('self')
    profile_page = ReferenceField('ProfilePage')

class ProfilePage(Document):
    content = StringField()
```

2.3.3.5.1。ListFields 的多对多

如果您通过引用列表实现多对多关系，则引用存储为 DBRefs 并且要查询您需要将对象的实例传递给查询：

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

class Page(Document):
    content = StringField()
    authors = ListField(ReferenceField(User))

bob = User(name="Bob Jones").save()
john = User(name="John Smith").save()

Page(content="Test Page", authors=[bob, john]).save()
Page(content="Another Page", authors=[john]).save()

# Find all pages Bob authored
Page.objects(authors__in=[bob])

# Find all pages that both Bob and John have authored
Page.objects(authors__all=[bob, john])

# Remove Bob from the authors for a page.
Page.objects(id='...').update_one(pull__authors=bob)

# Add John to the authors for a page.
Page.objects(id='...').update_one(push__authors=john)
```

2.3.3.5.2。处理参考文件的删除

默认情况下，MongoDB 不会检查数据的完整性，因此删除其他文档仍然引用的文档会导致一致性问题。Mongoengine `ReferenceField` 添加了一些功能来防止这些类型的数据库完整性问题，为每个引用提供删除规则规范。 `reverse_delete_rule` 通过在定义中 提供属性来指定删除规则 `ReferenceField` ，如下所示：

```
class ProfilePage(Document):
    employee = ReferenceField('Employee', reverse_delete_rule=mongoengine.CASCADE)
```

这个例子中的声明意味着当一个 `Employee` 对象被删除时， `ProfilePage` 引用那个员工的那个也被删除。如果删除了整批员工，则所有链接的个人资料页面也会被删除。

它的值可以采用以下任何常量：

`mongoengine.DO_NOTHING`

这是默认设置，不会执行任何操作。删除速度很快，但可能会导致数据库不一致或悬空引用。

`mongoengine.DENY`

如果仍然存在对被删除对象的引用，则拒绝删除。

`mongoengine.NULLIFY`

仍然引用被删除对象的任何对象的字段都被设置为 None（使用 MongoDB 的“unset”操作），有效地取消了关系。

`mongoengine.CASCADE`

首先删除任何包含引用被删除对象的字段的对象。

`mongoengine.PULL`

`ListField`从( )的任何对象字段中删除对该对象的引用（使用 MongoDB 的“拉”操作）`ReferenceField`。

警告

关于设置这些删除规则的安全注意事项！由于删除规则不是由 MongoDB 本身在数据库级别记录的，而是在运行时、在内存中由 MongoEngine 模块记录的，因此在调用删除之前加载声明关系的模块是最重要的。

例如，如果`Employee`对象存在于`payroll`应用程序中，并且存在于应用`ProfilePage`程序中，那么在删除任何员工之前加载应用程序`people`是极其重要的，因为否则，MongoEngine 永远不会知道这种关系的存在。`people`

在 Django 中，一定要将所有具有此类删除规则声明的应用程序放入它们的`models.py`元`INSTALLED_APPS`组中。

2.3.3.5.3。通用参考字段

还存在第二种参考字段，`GenericReferenceField`. 这允许您引用任何类型的`Document`，因此不会将`Document`子类作为构造函数参数：

```
class Link(Document):
    url = StringField()

class Post(Document):
    title = StringField()

class Bookmark(Document):
    bookmark_object = GenericReferenceField()

link = Link(url='http://hmarr.com/mongoengine/')
link.save()

post = Post(title='Using MongoEngine')
post.save()

Bookmark(bookmark_object=link).save()
Bookmark(bookmark_object=post).save()
```

笔记

使用`GenericReferenceField`s 的效率略低于标准`ReferenceField`s，因此如果您只引用一种文档类型，请首选标准`ReferenceField`.

2.3.3.6. 唯一性约束

`unique=True`MongoEngine 允许您通过提供给 a`Field`的构造函数来指定一个字段在整个集合中应该是唯一的。如果您尝试保存一个文档，该文档的唯一字段值与数据库中已有的文档具有相同的值，`NotUniqueError`则会引发 a 。您还可以使用 指定多字段唯一性约束`unique_with`，它可以是单个字段名称，也可以是字段名称的列表或元组：

```
class User(Document):
    username = StringField(unique=True)
    first_name = StringField()
    last_name = StringField(unique_with='first_name')
```

2.3.4. 文献集

直接继承自的文档类将在数据库中`Document`拥有自己的**集合。**集合的名称默认是转换为 snake_case 的类的名称（例如，如果您的文档类名为*CompanyUser*，则相应的集合将是 *company_user*）。如果您需要更改集合的名称（例如，将 MongoEngine 与现有数据库一起使用），则创建一个`meta`在您的文档上调用的类字典属性，并将其设置`collection`为您希望文档类使用的集合的名称：

```
class Page(Document):
    title = StringField(max_length=200, required=True)
    meta = {'collection': 'cmsPage'}
```

2.3.4.1. 上限集合

A`Document`可以通过在字典中指定 和来使用**Capped Collection**。 是允许存储在集合中的最大文档数，并且是集合的最大大小（以字节为单位）。由 MongoDB 内部和之前的 mongoengine 四舍五入到下一个 256 的倍数。也可以使用 256 的倍数以避免混淆。如果未指定并且 是，则默认为 10485760 字节 (10MB)。以下示例显示的文档将被限制为 1000 个条目和 2MB 的磁盘空间：

`max_documents` `max_size` `meta` `max_documents` `max_size` `max_size` `max_size` `max_documents` `max_size` `Log`

```
class Log(Document):
    ip_address = StringField()
    meta = {'max_documents': 1000, 'max_size': 2000000}
```

2.3.5. 索引

您可以在集合上指定索引以加快查询速度。`indexes`这是通过创建在 字典中调用的索引规范列表来完成的`meta`，其中索引规范可以是单个字段名称、包含多个字段名称的元组或包含完整索引定义的字典。

可以通过在字段名称前加上 **+**（表示升序）或**-**符号（表示降序）来指定字段的方向。请注意，方向仅对复合索引有影响。**可以通过在字段名称前加上$**来指定文本索引。散列索引可以通过在字段名称前加上**#**来指定：

```
class Page(Document):
    category = IntField()
    title = StringField()
    rating = StringField()
    created = DateTimeField()
    meta = {
        'indexes': [
            'title',   # single-field index
            '$title',  # text index
            '#title',  # hashed index
            ('title', '-rating'),  # compound index
            ('category', '_cls'),  # compound index
            {
                'fields': ['created'],
                'expireAfterSeconds': 3600  # ttl index
            }
        ]
    }
```

如果传递字典，则其他选项可用。有效选项包括但不限于：

`fields` **（默认值：无）**

要索引的字段。以与上述相同的格式指定。

`cls` **（默认值：真）**

如果你有继承并开启的多态模型 `allow_inheritance`，你可以配置索引是否应该将字段 `_cls` 自动添加到索引的开头。

`sparse` **（默认值：假）**

索引是否应该是稀疏的。

`unique` **（默认值：假）**

索引是否应该是唯一的。

`expireAfterSeconds` **（选修的）**

允许您通过以秒为单位设置使字段过期的时间来自动使集合中的数据过期。

`name` **（选修的）**

允许您为索引指定名称

`collation` **（选修的）**

允许创建不区分大小写的索引（仅限 MongoDB v3.4+）

笔记

附加选项作为** kwargs 转发给 pymongo 的 create_index 方法。继承添加了额外的字段索引，请参阅：文档继承。

2.3.5.1。全局索引默认选项

可以设置的所有索引都有一些顶级默认值：

```
class Page(Document):
    title = StringField()
    rating = StringField()
    meta = {
        'index_opts': {},
        'index_background': True,
        'index_cls': False,
        'auto_create_index': True,
        'auto_create_index_on_save': False,
    }
```

`index_opts` **（选修的）**

设置任何默认索引选项 - 查看完整选项列表

`index_background` **（选修的）**

设置索引是否应在后台编制索引的默认值

`index_cls` **（选修的）**

一种为 _cls 关闭特定索引的方法。

`auto_create_index` **（选修的）**

当这是 True （默认）时，MongoEngine 将确保在首次使用 Document 时 MongoDB 中存在正确的索引。这可以在索引单独管理的系统中禁用。禁用此功能将提高性能。

`auto_create_index_on_save` **（选修的）**

当这是 True 时，MongoEngine 将确保每次 `save()` 运行时 MongoDB 中存在正确的索引。启用它会降低性能。默认值为假。这个选项是在 0.25 版本中添加的。

2.3.5.2。复合索引和索引子文档

可以通过将嵌入式字段或字典字段名称添加到索引定义来创建复合索引。

有时对嵌入式/字典字段的部分进行索引更有效，在这种情况下使用“点”符号来标识要索引的值，例如：*rank.title*

2.3.5.3. 地理空间索引

mongodb 最好的地理索引是新的“2dsphere”，它具有改进的球形模型并在查询时提供更好的性能和更多选项。以下字段将明确添加“2dsphere”索引：

`PointField`

`LineStringField`

`PolygonField`

`MultiPointField`

`MultiLineStringField`

`MultiPolygonField`

由于“2dsphere”索引可以是复合索引的一部分，您可能不想要自动索引但更喜欢复合索引。在这个例子中，我们关闭了自动索引并在`location`and上显式声明了一个复合索引`datetime`：

```
class Log(Document):
    location = PointField(auto_index=False)
    datetime = DateTimeField()

    meta = {
        'indexes': [[("location", "2dsphere"), ("datetime", 1)]]
    }
```

2.3.5.3.1。Pre MongoDB 2.4 地理

笔记

对于 MongoDB < 2.4 这仍然是最新的，但是新的 2dsphere 索引比以前的 2D 模型有了很大的改进 - 所以建议升级。

将为所有`GeoPointField`s自动创建地理空间索引

也可以明确定义地理空间索引。如果您需要在`DictField`包含点的自定义字段的子字段上定义地理空间索引，这将非常有用。要创建地理空间索引，您必须在字段前加上 *号。

```
class Place(Document):
    location = DictField()
    meta = {
        'indexes': [
            '*location.point',
        ],
    }
```

2.3.5.4。生存时间 (TTL) 指标

一种特殊的索引类型，允许您在给定时间段后自动使集合中的数据过期。 有关详细信息，请参阅官方 ttl文档。一个常见的用例可能是会话数据：

```
class Session(Document):
    created = DateTimeField(default=datetime.utcnow)
    meta = {
        'indexes': [
            {'fields': ['created'], 'expireAfterSeconds': 3600}
        ]
    }
```

警告

TTL 索引发生在 MongoDB 服务器上而不是应用程序代码中，因此在删除文档时不会触发任何信号。如果您需要在删除时触发信号，那么您必须在应用程序代码中处理文档的删除。

2.3.5.5。比较指标

用于`mongoengine.Document.compare_indexes()`将数据库中的实际索引与文档定义定义的索引进行比较。这对于维护目的很有用，并确保您拥有适合您的架构的正确索引。

2.3.6. 订购

可以`QuerySet`使用 的`ordering`属性 为您指定默认排序`meta`。排序将在创建时应用`QuerySet`，并且可以被后续调用覆盖`order_by()`。

```
from datetime import datetime

class BlogPost(Document):
    title = StringField()
    published_date = DateTimeField()

    meta = {
        'ordering': ['-published_date']
    }

blog_post_1 = BlogPost(title="Blog Post #1")
blog_post_1.published_date = datetime(2010, 1, 5, 0, 0 ,0)

blog_post_2 = BlogPost(title="Blog Post #2")
blog_post_2.published_date = datetime(2010, 1, 6, 0, 0 ,0)

blog_post_3 = BlogPost(title="Blog Post #3")
blog_post_3.published_date = datetime(2010, 1, 7, 0, 0 ,0)

blog_post_1.save()
blog_post_2.save()
blog_post_3.save()

# get the "first" BlogPost using default ordering
# from BlogPost.meta.ordering
latest_post = BlogPost.objects.first()
assert latest_post.title == "Blog Post #3"

# override default ordering, order BlogPosts by "published_date"
first_post = BlogPost.objects.order_by("+published_date").first()
assert first_post.title == "Blog Post #1"
```

2.3.7. 片键

如果你的集合被多个键分片，那么你可以通过指定分片键，`shard_key`使用`meta`. 分片键应定义为元组。

`save()`这确保在调用诸如、`update()`、`modify()`或`delete()`现有实例上的 方法时，完整的分片键与查询一起发送`Document`：

```
class LogEntry(Document):
    machine = StringField()
    app = StringField()
    timestamp = DateTimeField()
    data = StringField()

    meta = {
        'shard_key': ('machine', 'timestamp'),
        'indexes': ('machine', 'timestamp'),
    }
```

2.3.8. 文档继承

`Document`要创建您定义的a 的特殊类型，您可以将其子类化并添加您可能需要的任何额外字段或方法。由于这个新类不是 的直接子类`Document`，因此不会存储在自己的集合中；它将使用与其超类使用的相同的集合。这允许更方便和高效地检索相关文档——您需要做的就是在文档的数据中将其设置`allow_inheritance`为 True 。：`meta`

```
# Stored in a collection named 'page'
class Page(Document):
    title = StringField(max_length=200, required=True)

    meta = {'allow_inheritance': True}

# Also stored in the collection named 'page'
class DatedPage(Page):
    date = DateTimeField()
```

笔记

从 0.8 开始`allow_inheritance`默认为 False，这意味着您必须将其设置为 True 才能使用继承。

设置`allow_inheritance`为 True 也应该在类中使用`EmbeddedDocument`，以防您需要对其进行子类化

在使用 进行查询时`objects()`，查询*Page.objects()*将同时查询*Page*和*DatedPage*，而查询*DatedPage*将仅查询*DatedPage*文档。在幕后，MongoEngine 通过`_cls`在每个文档中添加一个包含类名的属性来处理继承。加载文档时，MongoEngine 检查它的`_cls`属性并使用该类来构造实例：

```
Page(title='a funky title').save()
DatedPage(title='another title', date=datetime.utcnow()).save()

print(Page.objects().count())         # 2
print(DatedPage.objects().count())    # 1

# print documents in their native form
# we remove 'id' to avoid polluting the output with unnecessary detail
qs = Page.objects.exclude('id').as_pymongo()
print(list(qs))
# [
#   {'_cls': u 'Page', 'title': 'a funky title'},
#   {'_cls': u 'Page.DatedPage', 'title': u 'another title', 'date':
datetime.datetime(2019, 12, 13, 20, 16, 59, 993000)}
# ]
```

2.3.8.1. 使用现有数据

由于 MongoEngine 不再默认需要`_cls`，您可以快速轻松地使用现有数据。只需定义文档以匹配数据库中的预期模式

```
# Will work with data in an existing collection named 'cmsPage'
class Page(Document):
    title = StringField(max_length=200, required=True)
    meta = {
        'collection': 'cmsPage'
    }
```

如果您的架构变化很大，那么使用 a `DynamicDocument`可能更合适，而不是定义所有可能的字段类型。

如果您使用`Document`并且数据库包含未定义的数据，那么该数据将存储在*document._data*字典中。

2.3.9. 抽象类

如果你想向一组文档类添加一些额外的功能，但你不需要或不想要继承的开销，你可以`abstract`使用`meta`. 这不会打开文档继承，但可以让您保持代码干燥：

```
class BaseDocument(Document):
    meta = {
        'abstract': True,
    }
    def check_permissions(self):
        ...

class User(BaseDocument):
   ...
```

现在 User 类将可以访问继承的*check_permissions*方法，并且不会存储任何额外的*_cls*信息。