2.5. 查询数据库

`Document`类有一个`objects`属性，用于访问与类关联的数据库中的对象。该`objects`属性实际上是一个`QuerySetManager`，它在访问时创建并返回一个新`QuerySet`对象。`QuerySet`可以迭代该 对象以从数据库中获取文档：

```
# Prints out the names of all the users in the database
for user in User.objects:
    print user.name
```

笔记

从 MongoEngine 0.8 开始，查询集使用本地缓存。因此多次迭代只会导致单个查询。如果这不是您想要的行为，您可以调用`no_cache`（版本**0.8.3+**）返回一个非缓存查询集。

2.5.1. 过滤查询

可以通过使用字段查找关键字参数调用对象来过滤查询`QuerySet`。关键字参数中的键对应于您正在查询的字段`Document`：

```
# This will return a QuerySet that will only iterate over users whose
# 'country' field is set to 'uk'
uk_users = User.objects(country='uk')
```

嵌入式文档上的字段也可以通过使用双下划线代替对象属性访问语法中的点来使用字段查找语法来引用：

```
# This will return a QuerySet that will only iterate over pages that have
# been written by a user whose 'country' field is set to 'uk'
uk_pages = Page.objects(author__country='uk')
```

笔记

（版本**0.9.1+**）如果您的字段名称类似于 mongodb 运算符名称（例如 type、lte、lt...）并且您想将其放在查找关键字的末尾，mongoengine 会自动将 $ 添加到它的前面。为避免这种情况，请在查找关键字的末尾使用 __。例如，如果您的字段名称是`type`并且您想通过

该字段进行查询，则必须使用`.objects(user__type__="admin")`而不是`.objects(user__type="admin")`

2.5.2. 查询运算符

除了相等之外的运算符也可以在查询中使用——只需将运算符名称附加到带有双下划线的键上：

```
# Only find users whose age is 18 or less
young_users = Users.objects(age__lte=18)
```

可用的运算符如下：

`ne` - 不等于

`lt` - 少于

`lte` - 小于或等于

`gt` - 比...更棒

`gte` - 大于或等于

`not` - 否定标准检查，可以在其他运算符之前使用（例如 ）`Q(age__not__mod=(5, 0))`

`in` - 值在列表中（应提供值列表）

`nin` - 值不在列表中（应提供值列表）

`mod` - ，其中和是两个提供的值`value % x == y``x``y`

`all` - 提供的值列表中的每个项目都在数组中

`size` - 数组的大小是

`exists` - 字段值存在

2.5.2.1。字符串查询

以下运算符可用作使用正则表达式进行查询的快捷方式：

`exact` - 字符串字段与值完全匹配

`iexact` – 字符串字段与值完全匹配（不区分大小写）

`contains` – 字符串字段包含值

`icontains` – 字符串字段包含值（不区分大小写）

`startswith` – 字符串字段以值开头

`istartswith` – 字符串字段以值开头（不区分大小写）

`endswith` – 字符串字段以值结尾

`iendswith` – 字符串字段以值结尾（不区分大小写）

`wholeword` – 字符串字段包含整个单词

`iwholeword` – 字符串字段包含整个单词（不区分大小写）

`regex` – 正则表达式匹配的字符串字段

`iregex` – 正则表达式匹配的字符串字段（不区分大小写）

`match` – 执行 $elemMatch 以便您可以匹配数组中的整个文档

2.5.2.2。地理查询

有一些用于执行地理查询的特殊运算符。`PointField` 在 MongoEngine 0.8 中为，`LineStringField` 和 增加了以下内容 `PolygonField`：

`geo_within` – 检查几何图形是否在多边形内。为了便于使用，它接受 geojson 几何图形或仅接受多边形坐标，例如：

```
loc.objects(point__geo_within=[[[40, 5], [40, 6], [41, 6], [40, 5]]])
loc.objects(point__geo_within={"type": "Polygon",
                        "coordinates": [[[40, 5], [40, 6], [41, 6], [40, 5]]]})
```

`geo_within_box` – 简化的 geo_within 使用框搜索，例如：

```
loc.objects(point__geo_within_box=[(-125.0, 35.0), (-100.0, 40.0)])
loc.objects(point__geo_within_box=[<bottom left coordinates>, <upper right
coordinates>])
```

`geo_within_polygon` – 简化的 geo_within 在简单的多边形内搜索，例如：

```
loc.objects(point__geo_within_polygon=[[40, 5], [40, 6], [41, 6], [40, 5]])
loc.objects(point__geo_within_polygon=[ [ <x1> , <y1> ] ,
                                        [ <x2> , <y2> ] ,
                                        [ <x3> , <y3> ] ])
```

`geo_within_center` – 简化的 geo_within 点的平面圆半径内，例如：

```
loc.objects(point__geo_within_center=[(-125.0, 35.0), 1])
loc.objects(point__geo_within_center=[ [ <x>, <y> ] , <radius> ])
```

`geo_within_sphere` – 简化的 geo_within 点的球形圆半径内，例如：

```
loc.objects(point__geo_within_sphere=[(-125.0, 35.0), 1])
loc.objects(point__geo_within_sphere=[ [ <x>, <y> ] , <radius> ])
```

`geo_intersects` – 选择与几何相交的所有位置，例如：

```
# Inferred from provided points lists:
loc.objects(poly__geo_intersects=[40, 6])
loc.objects(poly__geo_intersects=[[40, 5], [40, 6]])
loc.objects(poly__geo_intersects=[[[40, 5], [40, 6], [41, 6], [41, 5], [40, 5]]])

# With geoJson style objects
loc.objects(poly__geo_intersects={"type": "Point", "coordinates": [40, 6]})
loc.objects(poly__geo_intersects={"type": "LineString",
                                  "coordinates": [[40, 5], [40, 6]]})
loc.objects(poly__geo_intersects={"type": "Polygon",
                                  "coordinates": [[[40, 5], [40, 6], [41, 6], [41, 5],
[40, 5]]]})
```

`near` – 找到给定点附近的所有位置：

```
loc.objects(point__near=[40, 5])
loc.objects(point__near={"type": "Point", "coordinates": [40, 5]})
```

您还可以设置以米为单位的最大和/或最小距离：

```
loc.objects(point__near=[40, 5], point__max_distance=1000)
loc.objects(point__near=[40, 5], point__min_distance=100)
```

旧的 2D 索引仍然支持 **`GeoPointField`**：

`within_distance` – 提供包含点和最大距离的列表（例如 [(41.342, -87.653), 5]）

`within_spherical_distance` – 同上，但使用球形地理模型（例如 [(41.342, -87.653), 5/earth_radius]）

`near` – 根据文档与给定点的接近程度对文档进行排序

`near_sphere` – 同上但使用球形地理模型

`within_box` – 将文档过滤到给定边界框内的文档（例如 [(35.0, -125.0), (40.0, -100.0)]）

`within_polygon` – 将文档过滤到给定多边形内的文档（例如 [(41.91,-87.69), (41.92,-87.68), (41.91,-87.65), (41.89,-87.65)]）。

笔记

需要 Mongo 服务器 2.0

`max_distance` – 可以添加到您的位置查询以设置最大距离。

`min_distance` – 可以添加到您的位置查询以设置最小距离。

2.5.2.3。查询列表

在大多数字段上，此语法将查找指定字段与给定值完全匹配的文档，但当字段引用 a 时，`ListField`可能会提供单个项目，在这种情况下，将匹配包含该项目的列表：

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

# This will match all pages that have the word 'coding' as an item in the
# 'tags' list
Page.objects(tags='coding')
```

通过使用数值作为查询运算符，可以按列表中的位置进行查询。因此，如果您想查找第一个标记为 的所有页面`db`，您可以使用以下查询：

```
Page.objects(tags__0='db')
```

如果您只想获取列表的一部分，例如：您想要对列表进行分页，则需要*切片运算符：*

```
# comments - skip 5, limit 10
Page.objects.fields(slice__comments=[5, 10])
```

对于更新文档，如果您不知道列表中的位置，可以使用 $ 位置运算符

```
Post.objects(comments__by="joe").update(**{'inc__comments__$__votes': 1})
```

但是，这并不能很好地映射到语法，因此您也可以改用大写 S

```
Post.objects(comments__by="joe").update(inc__comments__S__votes=1)
```

笔记

由于Mongo，目前 $ 运算符仅适用于查询中的第一个匹配项。

2.5.2.4。原始查询

可以提供原始`PyMongo`查询作为查询参数，它将直接集成到查询中。这是使用`__raw__`关键字参数完成的：

```
Page.objects(__raw__={'tags': 'coding'})
```

同样，可以向方法提供原始更新`update()`：

```
Page.objects(tags='coding').update(__raw__={'$set': {'tags': 'coding'}})
```

并且两者还可以结合使用：

```
Page.objects(__raw__={'tags': 'coding'}).update(__raw__={'$set': {'tags': 'coding'}})
```

2.5.2.5。使用聚合管道更新

可以提供原始`PyMongo`聚合更新参数，该参数将直接集成到更新中。这是通过`__raw__`在更新方法中使用关键字参数并将管道作为列表提供来完成的 Update with Aggregation Pipeline

```
# 'tags' field is set to 'coding is fun'
Page.objects(tags='coding').update(__raw__=[
    {"$set": {"tags": {"$concat": ["$tags", "is fun"]}}}
    ],
)
```

新版本 0.23.2。

2.5.3. 排序/排序结果

可以使用 1 个或多个键对结果进行排序`order_by()`。可以通过在每个键前面加上“+”或“-”来指定顺序。如果没有前缀，则假定升序：

```
# Order by ascending date
blogs = BlogPost.objects().order_by('date')    # equivalent to .order_by('+date')

# Order by ascending date first, then descending title
blogs = BlogPost.objects().order_by('+date', '-title')
```

2.5.4. 限制和跳过结果

与传统 ORM 一样，您可以限制返回结果的数量或跳过查询中的数字或结果。`limit()`和`skip()`方法在对象上可用`QuerySet`，但*数组切片*语法是实现此目的的首选：

```
# Only the first 5 people
users = User.objects[:5]

# All except for the first 5 people
users = User.objects[5:]

# 5 users, starting from the 11th user found
users = User.objects[10:15]
```

您还可以索引查询以检索单个结果。如果该索引处的项目不存在，`IndexError`则会引发 an 。`None`提供了检索第一个结果并在不存在结果时返回的快捷方式(`first()`)：

```
>>> # Make sure there are no users
>>> User.drop_collection()
>>> User.objects[0]
IndexError: list index out of range
>>> User.objects.first() == None
True
>>> User(name='Test User').save()
>>> User.objects[0] == User.objects.first()
True
```

2.5.4.1。检索唯一结果

要检索在集合中应该是唯一的结果，请使用 `get()`. `DoesNotExist`如果没有文档匹配查询，并且 `MultipleObjectsReturned` 有多个文档匹配查询，这将引发 。这些异常被合并到您的文档定义中，例如：*MyDoc.DoesNotExist*

此方法的变体 get_or_create() 存在，但它不安全。它不能保证安全，因为 mongoDB 中没有事务。应研究其他方法，以确保在使用与此方法类似的方法时不会意外复制数据。因此它在 0.8 中被弃用并在 0.10 中被删除。

2.5.5. 默认文档查询

`objects`默认情况下，文档的objects属性返回一个 `QuerySet`不过滤集合的对象——它返回所有对象。这可以通过在修改查询集的文档上定义一个方法来改变。该方法应该接受两个参数——`doc_cls`和 `queryset`。第一个参数是 `Document`定义该方法的类（从这个意义上说，该方法更像是一个 `classmethod()`而不是常规方法），第二个参数是初始查询集。该方法需要装饰 `queryset_manager()`以使其被识别。

```
class BlogPost(Document):
    title = StringField()
    date = DateTimeField()

    @queryset_manager
    def objects(doc_cls, queryset):
        # This may actually also be done by defining a default ordering for
        # the document, but this illustrates the use of manager methods
        return queryset.order_by('-date')
```

您不需要调用您的方法`objects`——您可以定义任意数量的自定义管理器方法：

```
class BlogPost(Document):
    title = StringField()
    published = BooleanField()

    @queryset_manager
    def live_posts(doc_cls, queryset):
        return queryset.filter(published=True)

BlogPost(title='test1', published=False).save()
BlogPost(title='test2', published=True).save()
assert len(BlogPost.objects) == 2
assert len(BlogPost.live_posts()) == 1
```

2.5.6. 自定义查询集

如果您想添加与文档交互或过滤文档的自定义方法，扩展该类`QuerySet`可能是可行的方法。`QuerySet`要在文档上使用自定义类，请在 a的字典`queryset_class`中设置自定义类：`Document` `meta`

```
class AwesomerQuerySet(QuerySet):

    def get_awesome(self):
        return self.filter(awesome=True)

class Page(Document):
    meta = {'queryset_class': AwesomerQuerySet}

# To call:
Page.objects.get_awesome()
```

新版本 0.4。

2.5.7. 聚合

MongoDB 提供了一些开箱即用的聚合方法，但没有 RDBMS 通常提供的那么多。MongoEngine 围绕内置方法提供了一个包装器，并提供了一些它自己的方法，这些方法被实现为在数据库服务器上执行的 Javascript 代码。

2.5.7.1。计数结果

就像限制和跳过结果一样，`QuerySet`对象上有一个方法 - `count()`：

```
num_users = User.objects.count()
```

从技术上讲，您可以使用它`len(User.objects)`来获得相同的结果，但它会比`count()`. 当您执行服务器端计数查询时，您让 MongoDB 完成繁重的工作，并通过网络接收到一个整数。同时，`len()`检索所有结果，放入本地缓存，最后统计。如果我们比较这两个操作的性能，`len()`比 . 慢得多`count()`。

2.5.7.2。进一步聚合

您可以使用以下方法对文档中特定字段的值求和`sum()`：

```
yearly_expense = Employee.objects.sum('salary')
```

笔记

如果文档中不存在该字段，则该文档将从总和中忽略。

要获取文档集合中某个字段的平均值，请使用 `average()`：

```
mean_age = User.objects.average('age')
```

由于 MongoDB 提供本机列表，MongoEngine 提供了一个辅助方法来获取整个集合中列表中项目频率的字典 - `item_frequencies()`. 它的一个使用示例是生成“标签云”：

```
class Article(Document):
    tag = ListField(StringField())

# After adding some tagged articles...
tag_freqs = Article.objects.item_frequencies('tag', normalize=True)

from operator import itemgetter
top_tags = sorted(tag_freqs.items(), key=itemgetter(1), reverse=True)[:10]
```

2.5.7.3。MongoDB 聚合 API

如果您需要运行聚合管道，MongoEngine 通过`aggregate()`. 查看 Pymongo 的语法和管道文档。它的使用示例如下：

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

Person(name='John').save()
Person(name='Bob').save()

pipeline = [
    {"$sort" : {"name" : -1}},
    {"$project": {"_id": 0, "name": {"$toUpper": "$name"}}}
    ]
data = Person.objects().aggregate(pipeline)
assert data == [{'name': 'BOB'}, {'name': 'JOHN'}]
```

2.5.8. 查询效率和性能

有几种方法可以在查询时提高效率，减少查询返回的信息或高效的解引用。

2.5.8.1。检索字段子集

`Document`有时需要a 上的字段子集，为了提高效率，只应从数据库中检索这些字段。这个问题对于 MongoDB 尤其重要，因为字段通常可能非常大（例如，a `ListField` of `EmbeddedDocument`s，代表博客文章上的评论。要仅选择字段的子集，请使用 `only()`，指定

要检索的字段作为其参数.注意如果没有下载的字段被访问，`None`会给出它们的默认值（或者没有提供默认值）：

```
>>> class Film(Document):
...     title = StringField()
...     year = IntField()
...     rating = IntField(default=3)
...
>>> Film(title='The Shawshank Redemption', year=1994, rating=5).save()
>>> f = Film.objects.only('title').first()
>>> f.title
'The Shawshank Redemption'
>>> f.year   # None
>>> f.rating # default value
3
```

笔记

如果你想排除一个字段，则`exclude()`相反 。`only()`

如果您以后需要缺少的字段，只需调用`reload()`您的文档即可。

2.5.8.2。获取相关数据

`ListField`当迭代or 的结果时，`DictField`我们会`DBRef`尽可能高效地自动取消对任何对象的引用，从而减少对 mongo 的查询次数。

有时这种效率还不够， 因为对 MongoDB 的查询数量会迅速增加，因此在顶层具有`ReferenceField`对象或 对象的文档非常昂贵。`GenericReferenceField`

要限制查询的数量，请使用`select_related()`将 QuerySet 转换为列表并尽可能高效地取消引用。默认情况下，`select_related()`仅取消引用对 1 级深度的任何引用。如果您有更复杂的文档并且想要一次取消引用更多的对象，那么增加将`max_depth`取消引用文档的更多级别。

2.5.8.3。关闭取消引用

有时出于性能原因，您不想自动取消引用数据。要关闭`no_dereference()`对查询集的查询结果的取消引用，如下所示：

```
post = Post.objects.no_dereference().first()
assert(isinstance(post.author, DBRef))
```

您还可以使用上下文管理器在固定时间段内关闭所有取消引用`no_dereference`：

```
with no_dereference(Post) as Post:
    post = Post.objects.first()
    assert(isinstance(post.author, DBRef))

# Outside the context manager dereferencing occurs.
assert(isinstance(post.author, User))
```

2.5.9. 高级查询

有时调用`QuerySet`带有关键字参数的对象不能完全表达您想要使用的查询——例如，如果您需要使用*and*和*or*组合多个约束。这在 MongoEngine 中通过类成为可能`Q`。对象`Q`代表查询的一部分，并且可以使用与查询文档相同的关键字参数语法进行初始化。要构建复杂查询，您可以 使用(and) 和(or) 运算符组合`Q`对象。要使用对象，请将其作为第一个位置参数传递给您通过使用关键字参数调用它来过滤它时：`&` `|` `Q` `Document.objects`

```
from mongoengine.queryset.visitor import Q

# Get published posts
Post.objects(Q(published=True) | Q(publish_date__lte=datetime.now()))

# Get top posts
Post.objects((Q(featured=True) & Q(hits__gte=1000)) | Q(hits__gte=5000))
```

警告

您必须使用按位运算符。您不能使用`or`,`and`来组合查询，因为它与 不同 。等同于 true 与 相同。`Q(a=a) or Q(b=b)` `Q(a=a) | Q(b=b)` `Q(a=a)` `Q(a=a) or Q(b=b)` `Q(a=a)`

2.5.10。原子更新

文档 可以通过 在a 上使用`update_one()`,`update()`和`modify()`方法`QuerySet`或 在 . 您可以将几种不同的“修饰符”用于这些方法：`modify()` `save()` `save_condition` `Document`

`set` – 设置一个特定的值

`set_on_insert` – 仅当这是新文档时才设置 `需要添加 upsert=True`_

`unset` – 删除特定值（自 MongoDB v1.3 起）

`max` – 仅在值较大时更新

`min` – 仅在值较小时更新

`inc` – 将一个值增加给定的数量

`dec` - 将一个值减少给定的数量

`push` - 将值附加到列表

`push_all` - 将几个值附加到列表

`pop` -根据值删除列表的第一个或最后一个元素

`pull` - 从列表中删除一个值

`pull_all` - 从列表中删除多个值

`add_to_set` - 仅在不在列表中时才向列表添加值

`rename` - 重命名键名

原子更新的语法类似于查询语法，但修饰符出现在字段之前，而不是之后：

```
>>> post = BlogPost(title='Test', page_views=0, tags=['database'])
>>> post.save()
>>> BlogPost.objects(id=post.id).update_one(inc__page_views=1)
>>> post.reload()  # the document has been changed, so we need to reload it
>>> post.page_views
1
>>> BlogPost.objects(id=post.id).update_one(set__title='Example Post')
>>> post.reload()
>>> post.title
'Example Post'
>>> BlogPost.objects(id=post.id).update_one(push__tags='nosql')
>>> post.reload()
>>> post.tags
['database', 'nosql']
```

笔记

如果未指定修饰符运算符，则默认值为`$set`. 所以下面的句子是相同的：

```
>>> BlogPost.objects(id=post.id).update(title='Example Post')
>>> BlogPost.objects(id=post.id).update(set__title='Example Post')
```

笔记

在版本 0.5 中`save()`，通过跟踪对该文档的更改来对更改的文档运行原子更新。

位置运算符允许您在不知道索引位置的情况下更新列表项，因此使更新成为一个原子操作。由于我们不能在关键字参数中使用$语法，它已被映射到*S*：

```
>>> post = BlogPost(title='Test', page_views=0, tags=['database', 'mongo'])
>>> post.save()
>>> BlogPost.objects(id=post.id, tags='mongo').update(set__tags__S='mongodb')
>>> post.reload()
>>> post.tags
['database', 'mongodb']
```

从 MongoDB 2.6 版开始，push 运算符支持 $position 值，它允许使用索引推送值：

```
>>> post = BlogPost(title="Test", tags=["mongo"])
>>> post.save()
>>> post.update(push__tags__0=["database", "code"])
>>> post.reload()
>>> post.tags
['database', 'code', 'mongo']
```

笔记

目前只处理顶级列表，mongodb/pymongo 的未来版本计划支持嵌套位置运算符。请参阅 $ 位置运算符。

2.5.11。服务器端javascript执行

可以编写 Javascript 函数并将其发送到服务器执行。其结果是 Javascript 函数的返回值。`exec_js()` 通过对象上的方法 访问此功能 `QuerySet()`。传入一个包含 Javascript 函数的字符串作为第一个参数。

剩余的位置参数是将作为参数传递给您的 Javascript 函数的字段名称。这允许编写可以在集合中的任何字段上执行的函数（例如 `sum()`，接受要求和的字段名称作为其参数的方法）。请注意，以这种方式传入的字段名称会自动转换为数据库中使用的名称（使用 `name` 字段构造函数的关键字参数设置）。

的关键字参数 `exec_js()` 组合到一个名为 的对象中 `options`，该对象在 Javascript 函数中可用。这可用于为您的函数定义特定参数。

一些变量在 Javascript 函数的范围内可用：

`collection` `Document` – 与正在使用的类相对应的集合的名称；这应该用于 `Collection` 从 `db` Javascript 代码中获取对象

`query` – 对象生成的查询 `QuerySet`；this 可以传递到Javascript 函数中对象 `find()` 的方法中 `Collection`

`options` – 包含传递给的关键字参数的对象 `exec_js()`

以下示例 `exec_js()` 通过定义一个函数来演示 的预期用法，该函数对文档的一个字段求和（此功能已经可用， `sum()` 但为了举例而在此处显示）：

```
def sum_field(document, field_name, include_negatives=True):
    code = """
    function(sumField) {
        var total = 0.0;
        db[collection].find(query).forEach(function(doc) {
            var val = doc[sumField];
            if (val >= 0.0 || options.includeNegatives) {
                total += val;
            }
        });
        return total;
    }
    """
    options = {'includeNegatives': include_negatives}
    return document.objects.exec_js(code, field_name, **options)
```

由于 MongoEngine 中的字段可能在数据库中使用不同的名称（使用 构造 `db_field` 函数的关键字参数设置 `Field` ），因此存在一种机制，可以用 Javascript 代码中的数据库字段名称替换 MongoEngine 字段名称。访问集合对象上的字段时，使用方括号表示法，并在 MongoEngine 字段名称前加上波浪号。波浪号后面的字段名称将被转换为数据库中使用的名称。请注意，在引用嵌入文档中的字段时， `EmbeddedDocumentField` 应在嵌入文档中的字段名称之前使用 的名称，后跟一个点。以下示例显示了如何进行替换：

```
class Comment(EmbeddedDocument):
    content = StringField(db_field='body')

class BlogPost(Document):
    title = StringField(db_field='doctitle')
    comments = ListField(EmbeddedDocumentField(Comment), name='cs')

# Returns a list of dictionaries. Each dictionary contains a value named
# "document", which corresponds to the "title" field on a BlogPost, and
# "comment", which corresponds to an individual comment. The substitutions
# made are shown in the comments.
BlogPost.objects.exec_js("""
function() {
    var comments = [];
    db[collection].find(query).forEach(function(doc) {
        // doc[~comments] -> doc["cs"]
        var docComments = doc[~comments];

        for (var i = 0; i < docComments.length; i++) {
            // doc[~comments][i] -> doc["cs"][i]
            var comment = doc[~comments][i];

            comments.push({
                // doc[~title] -> doc["doctitle"]
                'document': doc[~title],

                // comment[~comments.content] -> comment["body"]
                'comment': comment[~comments.content]
            });
        }
    });
    return comments;
}
""")
```