Edit by Finshy
Any Question you can contact with me,没事点个小星星啦
Emai:1638529046@qq.com

MongoDB reference

查询和 Project 操作符说明

比较查询操作符:

$eq	匹配与指定值相等的值。
$gt	匹配大于指定值的值。
$gte	匹配大于或等于指定值的值。
$in	匹配数组中指定的任何值。
$lt	匹配小于指定值的值。
$lte	匹配小于或等于指定值的值。
$ne	匹配所有不等于指定值的值。
$nin	不匹配数组中指定的任何值。

$eq

E:inventory collection
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: ["A", "B", "C"] }
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: ["B"] }
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: ["A", "B"] }
{ _id: 4, item: { name: "xy", code: "456" }, qty: 30, tags: ["B", "A"] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [["A", "B"], "C"] }
下面的示例查询 inventory 集合，以选择 qty 字段值为 20 的所有文档:
db.inventory.find({ qty: { $eq: 20 } })=db.inventory.find({ qty: 20 })
R:
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: ["B"] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [["A", "B"], "C"] }
内嵌文档中的字段等于一个值
下面的示例查询 inventory 集合，以选择项目文档中 name 字段的值等于“ab”的所有文档。
要在嵌入文档的字段上指定条件，请使用点符号。

db.inventory.find({ "item.name": { $eq: "ab" } })=db.inventory.find({ "item.name": "ab" })
R:
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: ["A", "B", "C"] }
数组元素等于一个值
下面的示例查询 inventory 集合，以选择标签数组中包含值为“B”[1]的元素的所有文档:
db.inventory.find({ tags: { $eq: "B" } })=db.inventory.find({ tags: "B" })
R:
{ _id: 1, item: { name: "ab", code: "123" }, qty: 15, tags: ["A", "B", "C"] }
{ _id: 2, item: { name: "cd", code: "123" }, qty: 20, tags: ["B"] }
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: ["A", "B"] }
{ _id: 4, item: { name: "xy", code: "456" }, qty: 30, tags: ["B", "A"] }
等于一个数组值
下面的示例查询 inventory 集合，以选择标记数组正好等于指定数组的所有文档，或者标记数组包含一个等于数组的元素["A"， "B"]。
db.inventory.find({ tags: { $eq: ["A", "B"] } })=db.inventory.find({ tags: ["A", "B"] })
R:
{ _id: 3, item: { name: "ij", code: "456" }, qty: 25, tags: ["A", "B"] }
{ _id: 5, item: { name: "mn", code: "000" }, qty: 20, tags: [["A", "B"], "C"] }

$gt

语法:{field: {$gt: value}}
$gt 选择字段值大于(即>)指定值的文档。
对于大多数数据类型，比较操作符只对 BSON 类型与查询值类型匹配的字段执行比较。
MongoDB 通过类型括号支持有限的跨 bson 比较。
考虑下面的例子:
db.inventory.find({ qty: { $gt: 20 } })
此查询将选择库存集合中 qty 字段值大于 20 的所有文档。
考虑下面的例子，它使用$gt 操作符和来自嵌入文档的字段:
db.inventory.update({ "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } })
他的 update()操作将在找到的第一个包含收费字段值大于 2 的嵌入式文档载体的文档中设置 price 字段的值。
若要在包含收费字段值大于 2 的嵌入式文档载体的所有文档中设置 price 字段的值，请在 update()方法中指定 multi:true 选项:

```
db.inventory.update(
    { "carrier.fee": { $gt: 2 } },
    { $set: { price: 9.99 } },
    { multi: true }
)
```

$gte

与 gt 类似。

$lt

与 gt 类似不过是小于

$lte

与 lt 类似不过是小于等于

$ne

与 lt 类似不过是不等于。

$in

$in 操作符选择字段的值等于指定数组中的任何值的文档。要指定$in 表达式，请使用以下原型:

{ field: { $in: [<value1>, <value2>, ... <valueN>] } }

如果字段包含一个数组，那么$in 操作符选择其字段包含一个数组的文档，该数组至少包含一个与指定数组中的值匹配的元素(例如<value1>， <value2>，等等)。

版本 2.6 中的更改:MongoDB 2.6 删除了$in 操作符的组合限制，该操作符的早期版本中存在这种限制。

Use the $in Operator to Match Values

E：db.inventory.find({ qty: { $in: [5, 15] } })

此查询选择库存集合中 qty 字段值为 5 或 15 的所有文档。虽然可以使用$or 操作符来表示此查询，但在对同一字段执行相等性检查时，请选择$in 操作符而不是$or 操作符。

Use the $in Operator to Match Values in an Array

E:inventory collection

{ _id: 1, item: "abc", qty: 10, tags: ["school", "clothing"], sale: false }

然后，下面的 update()操作将把 sale 字段的值设置为 true，其中 tags 字段包含一个数组，其中至少有一个元素匹配“appliance”或“school”。

```
db.inventory.update(
                    { tags: { $in: ["appliances", "school"] } },
                    { $set: { sale:true } }
                  )
```

Use the $in Operator with a Regular Expression

使用带有正则表达式实施例的$in 操作符

$in 操作符可以使用表单/pattern/的正则表达式指定匹配的值。不能在$in 中使用$regex 运算符表达式。

考虑下面的例子:

```
db.inventory.find( { tags: { $in: [ /^be/, /^st/ ] } } )
```

此查询选择 inventory 集合中的所有文档, 其中 tags 字段要么包含以 be 或 st 开头的字符串, 要么包含至少一个以 be 或 st 开头的元素的数组。

$nin

语法:{:{$ nin: [< value1 >, < value2 >⋯<家>]}}
$nin 选择以下文件:
字段值不在指定的数组中
该字段不存在。
有关不同 BSON 类型值的比较，请参见指定的 BSON 比较顺序。
考虑以下查询:
db.inventory.find({ qty: { $nin: [5, 15] } })
此查询将选择库存集合中 **qty 字段值不等于 5 或 15 的所有文档**。所选的文档将包括那些不包含 qty 字段的文档。
如果字段包含一个数组，那么$nin 操作符选择其字段包含一个数组的文档，该数组的元素不等于指定数组中的值(例如<value1>， <value2>，等等)。
考虑以下查询:
db.inventory.update({ tags: { $nin: ["appliances", "school"] } }, { $set: { sale: false } })
他的 update()操作将在 inventory 集合中设置 sales 字段值, 其中 tags 字段包含一个数组, 其中没有与数组中的元素匹配的元素["appliances"、"school"], 或者文档不包含 tags 字段。
不等式运算符$nin 不是很有选择性，因为它通常匹配索引的很大一部分。因此，在许多情况下，带有索引的$nin 查询的性能可能并不比必须扫描集合中的所有文档的$nin 查询好。请参见查询选择性。

逻辑查询操作:

$ AND 将查询子句与一个逻辑连接起来，并返回所有匹配这两个子句条件的文档。
$not 反转查询表达式的效果，并返回与查询表达式不匹配的文档。
$ NOR 将查询子句与逻辑子句连接起来，也不返回与两个子句不匹配的所有文档。
$ OR 将查询子句与逻辑子句连接起来，或者返回与这两个子句的条件匹配的所有文档。

$and

语法:{$and: [{<expression1>}， {<expression2>}，⋯， {<expressionN>}]}
$, 并对一个或多个表达式(例如<expression1>， <expression2>等)的数组执行逻辑和操作, 并选择满足数组中所有表达式的文档。$and 操作符使用短路评估。如果第一个表达式(例如<expression1>)的值为 false, MongoDB 将不计算其余表达式的值。
具有多个指定相同字段实现的表达式的查询
E:
db.inventory.find({ $and: [{ price: { $ne: 1.99 } }, { price: { $exists: true } }] })
此查询将选择库存集合中的所有文档，其中:

price 字段值不等于 1.99 和
price 字段存在。
通过组合 price 字段的运算符表达式，还可以使用隐式和操作构造此查询。例如，这个查询可以写成:

```
db.inventory.find( { price: { $ne: 1.99, $exists: true } } )
```

具有指定相同操作符实现的多个表达式的查询
E:

```
db.inventory.find( {
    $and : [
        { $or : [ { price : 0.99 }, { price : 1.99 } ] },
        { $or : [ { sale : true }, { qty : { $lt : 20 } } ] }
    ]
} )
```

此查询将选择以下所有文档:
price 字段值等于 0.99 或 1.99，并且
sales 字段值等于 true 或 qty 字段值小于 20。
此查询不能使用隐式和操作构造，因为它不止一次使用$or 操作符。

$not

语法:{field: {$not:{<操作符表达式>}}}
$not 对指定的<操作符表达式>执行逻辑 not 操作, 并选择与<操作符表达式>不匹配的文档。这包括不包含字段的文档。

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

此查询将选择库存集合中的所有文档，其中:

price 字段值小于或等于 1.99
price 字段不存在
{$not: {$gt: 1.99}}与$lte 运营商不同。{$lte: 1.99}只返回 price 字段存在且其值小于或等于 1.99 的文档。
请记住，$not 操作符只影响其他操作符，不能单独检查字段和文档。因此，使用$not 操作符进行逻辑分隔，使用$ne 操作符直接测试字段的内容。
$not 和数据类型
$not 操作符的操作与其他操作符的行为一致，但是对于数组之类的数据类型，可能会产生意想不到的结果。
$not 和正则表达式
$not 操作符可以执行逻辑 not 操作:
正则表达式对象(例如/pattern/)
例如，下面的查询选择 inventory 集合中的所有文档，其中 item 字段值不以字母 p 开头。

```
db.inventory.find( { item: { $not: /^p.*/ } } )
```

$regex 操作符表达式(从 MongoDB 4.0.7 开始)
例如，下面的查询选择 inventory 集合中的所有文档，其中 item 字段值不以字母 p 开头。

```
db.inventory.find( { item: { $not: { $regex: "^p.*" } } } )
db.inventory.find( { item: { $not: { $regex: /^p.*/ } } } )
```

驱动程序语言的正则表达式对象
例如，下面的 PyMongo 查询使用 Python 的 re.compile()方法来编译正则表达式:

```
import re
for noMatch in db.inventory.find( { "item": { "$not": re.compile("^p.*") } } ):
    print noMatch
```

$nor

$nor 对一个或多个查询表达式的数组执行逻辑或操作，并选择数组中所有查询表达式都失败的文档。$nor 的语法如下:

```
{ $nor: [ { <expression1> }, { <expression2> }, ...  { <expressionN> } ] }
```

E:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { sale: true } ]  } )
```

此查询将返回以下所有文档:
包含 value 不等于 1.99 的 price 字段，以及 value 不等于 true 的 sale 字段
包含 price 字段，其值不等于 1.99，但不包含 sale 字段或
不包含 price 字段，而是包含其值不等于 true 或
不包含价格字段和不包含销售字段
$nor 和附加的比较说明

```
db.inventory.find( { $nor: [ { price: 1.99 }, { qty: { $lt: 20 } }, { sale: true } ] } )
```

此查询将选择库存集合中的所有文档，其中:
price 字段值不等于 1.99 和
qty 字段值不小于 20 和
sales 字段值不等于 true
包括不包含这些字段的文档。
返回在$nor 表达式中不包含字段的文档时的例外情况是，当$nor 操作符与$exists 操作符一起使用时。
$nor and exists
将其与下面使用$nor 操作符和$exists 操作符的查询进行比较:

```
db.inventory.find( { $nor: [ { price: 1.99 }, { price: { $exists: false } },
                                { sale: true }, { sale: { $exists: false } } ] } )
```

此查询将返回以下所有文档:
包含 value 不等于 1.99 的 price 字段，以及 value 不等于 true 的 sale 字段

$or

$or 操作符对两个或多个<表达式>组成的数组执行逻辑或操作，并选择满足至少一个<表达式>的文档。$or 的语法如下:

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

E:

```
db.inventory.find( { $or: [ { quantity: { $lt: 20 } }, { price: 10 } ] } )
```

为了支持这个查询，而不是一个复合索引，您将创建一个关于数量的索引和另一个关于价格

的索引:
db.inventory.createIndex({ quantity: 1 })
db.inventory.createIndex({ price: 1 })
$or 和文本查询
在 2.6 版本中进行了更改。
如果$or 包含一个$text 查询，则$or 数组中的所有子句必须由索引支持。这是因为$text 查询必须使用索引，而$or 只能在索引支持其所有子句的情况下使用索引。如果$text 查询不能使用索引，查询将返回一个错误。

$or 和地理空间查询
在 2.6 版本中进行了更改。

$or 支持地理空间查询，但 near 子句除外(near 子句包括$near sphere 和$near)。$or 不能与任何其他子句一起包含 near 子句。

$or 和 sort 操作
在 2.6 版本中进行了更改。

当使用 sort()执行$or 查询时，MongoDB 现在可以使用支持$or 子句的索引。以前的版本没有使用索引。

Or versus$in
当使用$or 和<表达式>作为相同字段值的相等性检查时，使用$in 操作符而不是$or 操作符。例如，要选择 quantity 字段值为 20 或 50 的 inventory 集合中的所有文档，请使用$in 操作符:
db.inventory.find ({ quantity: { $in: [20, 50] } })
您可以嵌套$or 操作。

Element Query Operators 要素查询操作

$exists :　匹配具有指定字段的文档。
$type　　如果字段是指定的类型，则选择 document。

$exists

语法:{field: {$exists: <boolean>}}
当<boolean>为真时，$exists 匹配包含该字段的文档，包括字段值为 null 的文档。如果<boolean>为 false，则查询只返回不包含该字段的文档。
MongoDB $exists 不对应于 SQL 操作符 exists。如果 SQL 存在，请参考$in 操作符。
另请参阅
$nin、$in 和查询空字段或缺失字段。
存在且不等于
E:

db.inventory.find({ qty: { $exists: true, $nin: [5, 15] } })
此查询将选择库存集合中存在 qty 字段且其值不等于 5 或 15 的所有文档。
空值
下面的例子使用一个名为 records 的集合，其中包含以下文档:
{ a: 5, b: 5, c: null }
{ a: 3, b: null, c: 8 }
{ a: null, b: 3, c: 9 }
{ a: 1, b: 2, c: 3 }
{ a: 2, c: 5 }
{ a: 3, b: 2 }
{ a: 4 }
{ b: 2, c: 4 }
{ b: 2 }
{ c: 6 }
$exists: true
下面的查询指定查询谓词 a: {$exists: true}:
db.records.find({ a: { $exists: true } })
R:
{ a: 5, b: 5, c: null }
{ a: 3, b: null, c: 8 }
{ a: null, b: 3, c: 9 }
{ a: 1, b: 2, c: 3 }
{ a: 2, c: 5 }
{ a: 3, b: 2 }
{ a: 4 }
下面的查询指定查询谓词 b: {$exists: false}:、
db.records.find({ b: { $exists: false } })
R:
{ a: 2, c: 5 }
{ a: 4 }
{ c: 6 }

$type

$type 选择字段值为指定 BSON 类型实例的文档。在处理数据类型不可预测的高度非结构化数据时，按数据类型查询非常有用。
单个 BSON 类型的$type 表达式具有以下语法:
在 3.2 版本中进行了更改。
{ field: { $type: <BSON type> } }
上面的查询将匹配字段值为列出的任何类型的文档。数组中指定的类型可以是数字别名，也可以是字符串别名。
有关示例，请参见按多个数据类型进行查询。
可用类型描述 BSON 类型及其对应的数字和字符串别名。
另请参阅

如果希望获得操作符表达式返回的 BSON 类型，而不是根据其 BSON 类型过滤文档，请使用$type 聚合操作符。
$type 返回字段的 BSON 类型与传递给$type 的 BSON 类型匹配的文档。
在 3.6 版中进行了更改。
$type 现在可以像处理其他 BSON 类型一样处理数组。以前的版本只匹配字段包含嵌套数组的文档。
可用的类型
版本 3.2 中的更改:除了对应于 BSON 类型的数字之外，$type 操作符还接受 BSON 类型的字符串别名。以前的版本只接受与 BSON 类型对应的数字。

Type	Number	Alias	Notes
Double	1	“double”	
String	2	“string”	
Object	3	“object”	
Array	4	“array”	
Binary data	5	“binData”	
Undefined	6	“undefined”	Deprecated.
ObjectId	7	“objectId”	
Boolean	8	“bool”	
Date	9	“date”	
Null	10	“null”	
Regular Expression	11	“regex”	
DBPointer	12	“dbPointer”	Deprecated.
JavaScript	13	“javascript”	
Symbol	14	“symbol”	Deprecated.
JavaScript (with scope)	15	“javascriptWithScope”	

32-bit integer	16	“int”	
Timestamp	17	“timestamp”	
64-bit integer	18	“long”	
Decimal128	19	“decimal”	New in version 3.4.
Min key	-1	“minKey”	
Max key	127	“maxKey”	

$type 支持数字别名，它将与以下 BSON 类型匹配:
双
32 位整数
64 位整数
小数
参见按数据类型查询
MinKey 和 MaxKey
MinKey 和 MaxKey 用于比较操作，主要用于内部使用。对于所有可能的 BSON 元素值，MinKey 总是最小的值，而 MaxKey 总是最大的值。
使用$type 查询 minKey 或 maxKey 将只返回与特殊的 minKey 或 maxKey 值匹配的字段。
假设数据收集有两个文档，分别使用 MinKey 和 MaxKey:
{ "_id" : 1, x : { "$minKey" : 1 } }
{ "_id" : 2, y : { "$maxKey" : 1 } }
R:
下面的查询将返回_id: 1:
db.data.find({ x: { $type: "minKey" } })
下面的查询将返回_id: 2:
db.data.find({ y: { $type: "maxKey" } })
E:
地址簿包含地址和邮政编码，其中邮政编码有字符串、int、double 和 long 值:

```
db.addressBook.insertMany(
    [
        { "_id" : 1, address : "2030 Martian Way", zipCode : "90698345" },
        { "_id" : 2, address: "156 Lunar Place", zipCode : 43339374 },
        { "_id" : 3, address : "2324 Pluto Place", zipCode: NumberLong(3921412) },
        { "_id" : 4, address : "55 Saturn Ring" , zipCode : NumberInt(88602117) }
    ]
)
```

以下查询返回所有 zipCode 为 BSON 类型字符串的文档:

```
db.addressBook.find( { "zipCode" : { $type : 2 } } );
db.addressBook.find( { "zipCode" : { $type : "string" } } );
```

R:
{ "_id" : 1, "address" : "2030 Martian Way", "zipCode" : "90698345" }

以下查询返回所有 zipCode 为 BSON 类型 double 的文档:

```
db.addressBook.find( { "zipCode" : { $type : 1 } } )
db.addressBook.find( { "zipCode" : { $type : "double" } } )
```

R:

```
{ "_id" : 2, "address" : "156 Lunar Place", "zip" : 43339374 }
```

下面的查询使用数字别名返回 zipCode 为 BSON 类型 double、int 或 long 的文档:

```
db.addressBook.find( { "zipCode" : { $type : "number" } } )
```

R:

```
{ "_id" : 2, address : "156 Lunar Place", zipCode : 43339374 }
{ "_id" : 3, address : "2324 Pluto Place", zipCode: NumberLong(3921412) }
{ "_id" : 4, address : "55 Saturn Ring" , zipCode : 88602117 }
```

根据多个数据类型进行查询

grade 集合包含名称和平均值，其中 classAverage 有 string、int 和 double 值:

```
db.grades.insertMany(
    [
        { "_id" : 1, name : "Alice King" , classAverage : 87.333333333333333 },
        { "_id" : 2, name : "Bob Jenkins", classAverage : "83.52" },
        { "_id" : 3, name : "Cathy Hart", classAverage: "94.06" },
        { "_id" : 4, name : "Drew Williams" , classAverage : 93 }
    ]
)
```

下面的查询返回所有 classAverage 为 BSON 类型字符串或 double 的文档。第一个查询使用数字别名，而第二个查询使用字符串别名。

```
db.grades.find( { "classAverage" : { $type : [ 2 , 1 ] } } );
db.grades.find( { "classAverage" : { $type : [ "string" , "double" ] } } );
```

R;

```
{ "_id" : 1, name : "Alice King" , classAverage : 87.333333333333333 }
{ "_id" : 2, name : "Bob Jenkins", classAverage : "83.52" }
{ "_id" : 3, name : "Cathy Hart", classAverage: "94.06" }
```

通过 MinKey 和 MaxKey 查询

The restaurants collection 对任何不及格的等级都使用 minKey:

```
{
    "_id": 1,
    "address": {
        "building": "230",
        "coord": [ -73.996089, 40.675018 ],
        "street": "Huntington St",
        "zipcode": "11231"
    },
    "borough": "Brooklyn",
    "cuisine": "Bakery",
    "grades": [
        { "date": new Date(1393804800000), "grade": "C", "score": 15 },
        { "date": new Date(1378857600000), "grade": "C", "score": 16 },
```

```
        { "date": new Date(1358985600000), "grade": MinKey(), "score": 30 },
        { "date": new Date(1322006400000), "grade": "C", "score": 15 }
    ],
    "name": "Dirty Dan's Donuts",
    "restaurant_id": "30075445"
}
```

和 maxKey 为任何年级的最高通过分数:

```
{
    "_id": 2,
    "address": {
        "building": "1166",
        "coord": [ -73.955184, 40.738589 ],
        "street": "Manhattan Ave",
        "zipcode": "11222"
    },
    "borough": "Brooklyn",
    "cuisine": "Bakery",
    "grades": [
        { "date": new Date(1393804800000), "grade": MaxKey(), "score": 2 },
        { "date": new Date(1378857600000), "grade": "B", "score": 6 },
        { "date": new Date(1358985600000), "grade": MaxKey(), "score": 3 },
        { "date": new Date(1322006400000), "grade": "B", "score": 5 }
    ],
    "name": "Dainty Daisey's Donuts",
    "restaurant_id": "30075449"
}
```

下面的查询返回其等级的任何餐厅。等级字段包含 minKey:

```
db.restaurants.find(
    { "grades.grade" : { $type : "minKey" } }
)
```

R:

```
{
    "_id" : 1,
    "address" : {
        "building" : "230",
        "coord" : [ -73.996089, 40.675018 ],
        "street" : "Huntington St",
        "zipcode" : "11231"
    },
    "borough" : "Brooklyn",
    "cuisine" : "Bakery",
    "grades" : [
        { "date" : ISODate("2014-03-03T00:00:00Z"), "grade" : "C", "score" : 15 },
        { "date" : ISODate("2013-09-11T00:00:00Z"), "grade" : "C", "score" : 16 },
```

```
        { "date" : ISODate("2013-01-24T00:00:00Z"), "grade" : { "$minKey" : 1 }, "score" :
30 },
        { "date" : ISODate("2011-11-23T00:00:00Z"), "grade" : "C", "score" : 15 }
    ],
    "name" : "Dirty Dan's Donuts",
    "restaurant_id" : "30075445"
}
```

下面的查询返回其等级的任何餐厅。等级字段包含 maxKey:

```
db.restaurants.find(
    { "grades.grade" : { $type : "maxKey" } }
)
R:{
    "_id" : 2,
    "address" : {
        "building" : "1166",
        "coord" : [ -73.955184, 40.738589 ],
        "street" : "Manhattan Ave",
        "zipcode" : "11222"
    },
    "borough" : "Brooklyn",
    "cuisine" : "Bakery",
    "grades" : [
        { "date" : ISODate("2014-03-03T00:00:00Z"), "grade" : { "$maxKey" : 1 }, "score" :
2 },
        { "date" : ISODate("2013-09-11T00:00:00Z"), "grade" : "B", "score" : 6 },
        { "date" : ISODate("2013-01-24T00:00:00Z"), "grade" : { "$maxKey" : 1 }, "score" :
3 },
        { "date" : ISODate("2011-11-23T00:00:00Z"), "grade" : "B", "score" : 5 }
    ],
    "name" : "Dainty Daisey's Donuts",
    "restaurant_id" : "30075449"
}
```

名为 SensorReading 的集合包含以下文档:

```
{
    "_id": 1,
    "readings": [
        25,
        23,
        [ "Warn: High Temp!", 55 ],
        [ "ERROR: SYSTEM SHUTDOWN!", 66 ]
    ]
},
{
    "_id": 2,
```

```
    "readings": [
        25,
        25,
        24,
        23
    ]
},
{
    "_id": 3,
    "readings": [
        22,
        24,
        []
    ]
},
{
    "_id": 4,
    "readings": []
},
{
    "_id": 5,
    "readings": 24
}
```


下面的查询返回 reading 字段为数组(空或非空)的任何文档。

```
db.SensorReading.find( { "readings" : { $type: "array" } } )
```

R:


```
{
    "_id": 1,
    "readings": [
        25,
        23,
        [ "Warn: High Temp!", 55 ],
        [ "ERROR: SYSTEM SHUTDOWN!", 66 ]
    ]
},
{
    "_id": 2,
    "readings": [
        25,
        25,
        24,
        23
    ]
},
```

```
{
    "_id": 3,
    "readings": [
        22,
        24,
        []
    ]
},
{
    "_id": 4,
    "readings": []
}
```

在具有_id: 1、_id: 2、_id: 3 和_id: 4 的文档中，reading 字段是一个数组。

评估查询操作符

$expr　　允许在查询语言中使用聚合表达式。
$ jsonschema　　根据给定的 JSON 模式验证文档。
$mod　　对字段的值执行模操作，并选择具有指定结果的文档。
$regex　　选择值与指定正则表达式匹配的文档。
$text　　执行文本搜索。
$where　　匹配满足 JavaScript 表达式的文档。

$expr

表达式:
{ $expr: { <expression> } }
$expr 可以在$match 阶段构建查询表达式，用于比较来自相同文档的字段。
如果$match 阶段是$lookup 阶段的一部分，那么$expr 可以使用 let 变量比较字段。有关示例，请参见使用$lookup 指定多个连接条件。
$expr 不支持多键索引。
E:monthlyBudget:
{ "_id" : 1, "category" : "food", "budget": 400, "spent": 450 },
{ "_id" : 2, "category" : "drinks", "budget": 100, "spent": 150 },
{ "_id" : 3, "category" : "clothes", "budget": 100, "spent": 50 },
{ "_id" : 4, "category" : "misc", "budget": 500, "spent": 300 },
{ "_id" : 5, "category" : "travel", "budget": 200, "spent": 650 },
以下操作使用$expr 查找超出预算的文件:
db.monthlyBudget.find({ $expr: { $gt: ["$spent" , "$budget"] } })
R:
{ "_id" : 1, "category" : "food", "budget" : 400, "spent" : 450 }
{ "_id" : 2, "category" : "drinks", "budget" : 100, "spent" : 150 }

{ "_id" : 5, "category" : "travel", "budget" : 200, "spent" : 650 }
下面的操作使用$expr 和$cond 来模拟条件语句。它可以找到应用折扣后价格低于 5 的文档，如果文档的 qty 值大于或等于 100，则查询将价格除以 2。否则，价格除以 4:

```
db.supplies.find( {
      $expr: {
          $lt:[ {
              $cond: {
                  if: { $gte: ["$qty", 100] },
                  then: { $divide: ["$price", 2] },
                  else: { $divide: ["$price", 4] }
              }
          },
          5 ] }
} )
```

R:

{ "_id" : 2, "item" : "notebook", "qty": 200 , "price": 8 }
{ "_id" : 3, "item" : "pencil", "qty": 50 , "price": 6 }
{ "_id" : 4, "item" : "eraser", "qty": 150 , "price": 3 }

$jsonschema

$jsonSchema 操作符匹配满足指定 JSON 模式的文档。
$jsonSchema 操作符表达式有以下语法:
{ $jsonSchema: <JSON Schema object> }
其中 JSON 模式对象按照 JSON 模式标准草案 4 进行格式化。
E:

```
{
   $jsonSchema: {
       required: [ "name", "major", "gpa", "address" ],
       properties: {
           name: {
               bsonType: "string",
               description: "must be a string and is required"
           },
           address: {
               bsonType: "object",
               required: [ "zipcode" ],
               properties: {
                   "street": { bsonType: "string" },
                   "zipcode": { bsonType: "string" }
               }
           }
       }
   }
```

}
您可以在查询条件中使用$jsonSchema 进行读写操作，以在集合中找到满足指定模式的文档:

```
db.collection.find( { $jsonSchema: <schema> } )
db.collection.aggregate( [ { $match: { $jsonSchema: <schema> } } ] )
db.collection.updateMany( { $jsonSchema: <schema> }, <update> )
db.collection.deleteOne( { $jsonSchema: <schema> } )
```

要在集合中查找不满足指定模式的文档，请在$nor 表达式中使用$jsonSchema 表达式。例如:

```
db.collection.find( { $nor: [ { $jsonSchema: <schema> } ] } )
db.collection.aggregate( [ { $match: { $nor: [ { $jsonSchema: <schema> } ] } }, ... ] )
db.collection.updateMany( { $nor: [ { $jsonSchema: <schema> } ] }, <update> )
db.collection.deleteOne( { $nor: [ { $jsonSchema: <schema> } ] } )
```

下面的 db.createCollection()方法创建一个名为 students 的集合，并使用$jsonSchema 操作符设置模式验证规则:

```
db.createCollection("students", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: [ "name", "year", "major", "address" ],
            properties: {
                name: {
                    bsonType: "string",
                    description: "must be a string and is required"
                },
                year: {
                    bsonType: "int",
                    minimum: 2017,
                    maximum: 3017,
                    description: "must be an integer in [ 2017, 3017 ] and is required"
                },
                major: {
                    enum: [ "Math", "English", "Computer Science", "History", null ],
                    description: "can only be one of the enum values and is required"
                },
                gpa: {
                    bsonType: [ "double" ],
                    description: "must be a double if the field exists"
                },
                address: {
                    bsonType: "object",
                    required: [ "city" ],
                    properties: {
```

```
                    street: {
                        bsonType: "string",
                        description: "must be a string if the field exists"
                    },
                    city: {
                        bsonType: "string",
                        "description": "must be a string and is required"
                    }
                }
            }
        }
    }
})
```

定为集合创建的验证器，下面的插入操作将失败，因为当验证器需要双精度时，gpa 是整数。

```
db.students.insert({
    name: "Alice",
    year: NumberInt(2019),
    major: "History",
    gpa: NumberInt(3),
    address: {
        city: "NYC",
        street: "33rd Street"
    }
})
```

Erro 信息如下：

```
db.students.insert({
    name: "Alice",
    year: NumberInt(2019),
    major: "History",
    gpa: NumberInt(3),
    address: {
        city: "NYC",
        street: "33rd Street"
    }
})
```

gpa 改为双精度 will be successful：

```
db.students.insert({
    name: "Alice",
    year: NumberInt(2019),
    major: "History",
    gpa: 3.0,
    address: {
        city: "NYC",
```

```
        street: "33rd Street"
    }
})
```

您可以在查询条件中使用$jsonSchema 进行读写操作，以在集合中找到满足指定模式的文档。

例如，用以下文件创建一个样本收集清单:

```
db.inventory.insertMany([
    { item: "journal", qty: NumberInt(25), size: { h: 14, w: 21, uom: "cm" }, instock: true },
    { item: "notebook", qty: NumberInt(50), size: { h: 8.5, w: 11, uom: "in" }, instock: true },
    { item: "paper", qty: NumberInt(100), size: { h: 8.5, w: 11, uom: "in" }, instock: 1 },
    { item: "planner", qty: NumberInt(75), size: { h: 22.85, w: 30, uom: "cm" }, instock: 1 },
    { item: "postcard", qty: NumberInt(45), size: { h: 10, w: 15.25, uom: "cm" }, instock:
true },
    { item: "apple", qty: NumberInt(45), status: "A", instock: true },
    { item: "pears", qty: NumberInt(50), status: "A", instock: true }
])
```

接下来，定义以下示例模式对象:

```
let myschema =   {
        required: [ "item", "qty", "instock" ],
        properties: {
            item: { bsonType: "string" },
            qty: { bsonType: "int" },
            size: {
                bsonType: "object",
                required: [ "uom" ],
                properties: {
                    uom: { bsonType: "string" },
                    h: { bsonType: "double" },
                    w: { bsonType: "double" }
                }
             },
             instock: { bsonType: "bool" }
        }
 }
```

您可以使用$jsonSchema 查找集合中满足模式的所有文档:

```
db.inventory.find( { $jsonSchema: myschema } )
db.inventory.aggregate( [ { $match: { $jsonSchema: myschema } } ] )
```

您可以使用$jsonSchema 与$nor 一起查找所有不满足该模式的文档:

```
db.inventory.find( { $nor: [ { $jsonSchema: myschema } ] } )
```

或者，您可以更新所有不符合模式的文档:

```
db.inventory.updateMany( { $nor: [ { $jsonSchema: myschema } ] }, { $set: { isValid:
false } } )
```

或者，您可以删除所有不符合模式的文档:

```
db.inventory.deleteMany( { $nor: [ { $jsonSchema: myschema } ] } )
```

$mod

选择一个字段除以一个除数的值有指定余数的文档(即执行一个模数操作来选择文档)。要指定$mod 表达式，请使用以下语法:

{ field: { $mod: [divisor, remainder] } }

E:

{ "_id" : 1, "item" : "abc123", "qty" : 0 }

{ "_id" : 2, "item" : "xyz123", "qty" : 5 }

{ "_id" : 3, "item" : "ijk123", "qty" : 12 }

然后，下面的查询选择库存集合中 qty 字段 modulo 4 值为 0 的文档:

db.inventory.find({ qty: { $mod: [4, 0] } })

查询返回以下文件:

{ "_id" : 1, "item" : "abc123", "qty" : 0 }

{ "_id" : 3, "item" : "ijk123", "qty" : 12 }

元素数量不足

$mod 操作符在传递一个少于两个元素的数组时出错。

单元素数组

以下操作错误地向$mod 操作符传递一个包含单个元素的数组:

db.inventory.find({ qty: { $mod: [4] } })

该语句导致以下错误:

```
error: {
        "$err" : "bad query: BadValue malformed mod, not enough elements",
        "code" : 16810
}
```

且，mod 不能为空，不能也不能多个值。

$regex

为查询中的模式匹配字符串提供正则表达式功能。MongoDB 使用 Perl 兼容的正则表达式(即“PCRE”)8.41 版本，并支持 UTF-8。

要使用$regex，请使用以下语法之一:

{ <field>: { $regex: /pattern/, $options: '<options>' } }

{ <field>: { $regex: 'pattern', $options: '<options>' } }

{ <field>: { $regex: /pattern/<options> } }

Options

选项描述语法限制

I:　　病例对上下病例不敏感。例如，请参见执行不区分大小写的正则表达式匹配。

M:

对于包含锚的模式(例如^ For the start，　$ For the end)，在每一行的开头或结尾匹配具有多行值的字符串。如果没有此选项，这些锚将匹配字符串的开头或结尾。例如，请参见多行匹配以指定模式开始的行。

如果模式不包含锚点，或者字符串值没有换行符(例如\n)，则 m 选项没有效果。
X:
“扩展”功能，可以忽略$regex 模式中的所有空白字符，除非转义或包含在字符类中。
此外，它会忽略中间的字符，包括未转义的散列/磅(#)字符和下一行新字符，这样您就可以在复杂的模式中包含注释。这只适用于数据字符;空白字符可能永远不会出现在模式中的特殊字符序列中。
x 选项不影响 VT 字符的处理(即代码 11)。
需要$regex 和$options 语法
S: 允许点字符(即)匹配所有字符，包括换行字符。例如，请参见使用。点字符以匹配新行。
需要$regex 和$options 语法

要在$in 查询表达式中包含正则表达式，您只能使用 JavaScript 正则表达式对象(即/pattern/)。例如:
{ name: { $in: [/^acme/i, /^ack/] } }
不能在$in 中使用$regex 运算符表达式。
隐式和场的条件
要在以逗号分隔的字段查询条件列表中包含正则表达式，请使用$regex 操作符。例如:
{ name: { $regex: /acme.*corp/i, $nin: ['acmeblahcorp'] } }
{ name: { $regex: /acme.*corp/, $options: 'i', $nin: ['acmeblahcorp'] } }
{ name: { $regex: 'acme.*corp', $options: 'i', $nin: ['acmeblahcorp'] } }
x 和 s 选项
要使用 x 选项或 s 选项，必须使用$regex 操作符表达式和$options 操作符。例如，要指定 i 和 s 选项，必须同时使用$options:
{ name: { $regex: /acme.*corp/, $options: "si" } }
{ name: { $regex: 'acme.*corp', $options: "si" } }
PCRE vs JavaScript
要在 regex 模式中使用在 JavaScript 中不受支持的 PCRE 支持特性，必须使用$regex 操作符表达式，并将模式作为字符串。例如，要在模式中使用(?i)打开其余模式的大小写不敏感，(?-i)打开其余模式的大小写不敏感，必须使用$regex 操作符将模式作为字符串:
{ name: { $regex: '(?i)a(?-i)cme' } }
正则表达式和$not
从 4.0.7 开始，$not 操作符可以对这两个对象执行逻辑 not 操作:
正则表达式对象(例如/pattern/)
例如:
db.inventory.find({ item: { $not: /^p.*/ } })
$regex 操作符表达式(从 MongoDB 4.0.7 开始)。
例如:
db.inventory.find({ item: { $not: { $regex: "^p.*" } } })
db.inventory.find({ item: { $not: { $regex: /^p.*/ } } })
E:
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
{ "_id" : 102, "sku" : "xyz456", "description" : "Many spaces before   line" }

{ "_id" : 103, "sku" : "xyz789", "description" : "Multiple\nline description" }
db.products.find({ sku: { $regex: /789$/ } })
这个例子类似于下面的 SQL LIKE 语句:
SELECT * FROM products
WHERE sku like "%789";
执行不区分大小写的正则表达式匹配
下面的示例使用 i 选项对以 ABC 开头的具有 sku 值的文档执行不区分大小写的匹配。
db.products.find({ sku: { $regex: /^ABC/i } })
查询匹配以下文件:
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
多行匹配以指定模式开始的行
下面的例子使用 m 选项来匹配多行字符串中以字母 S 开头的行:
db.products.find({ description: { $regex: /^S/, $options: 'm' } })
R:
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }
{ "_id" : 101, "sku" : "abc789", "description" : "First line\nSecond line" }
使用。点字符以匹配新行
下面的例子使用 s 选项来允许点字符(即)匹配所有字符，包括新行，以及 i 选项来执行不区分大小写的匹配:
db.products.find({ description: { $regex: /m.*line/, $options: 'si' } })
R:
{ "_id" : 102, "sku" : "xyz456", "description" : "Many spaces before line" }
{ "_id" : 103, "sku" : "xyz789", "description" : "Multiple\nline description" }
没有 s 选项将会匹配到如下选项:
{ "_id" : 102, "sku" : "xyz456", "description" : "Many spaces before line" }
忽略模式中的空格

下面的例子使用 x 选项忽略空格和注释，以匹配模式中的#和\n 结尾:
var pattern = "abc #category code\n123 #item number"
db.products.find({ sku: { $regex: pattern, $options: "x" } })
R:
{ "_id" : 100, "sku" : "abc123", "description" : "Single line description." }

$text

$text 对使用文本索引索引的字段的内容执行文本搜索。$text 表达式有以下语法:
在 3.2 版本中进行了更改。

```
{
  $text:
    {
      $search: <string>,
      $language: <string>,
      $caseSensitive: <boolean>,
```

```
        $diacriticSensitive: <boolean>
      }
}
```

E:

下面的例子假设一个集合文章有一个关于字段主题的 version 3 文本索引:

```
db.articles.insert(
   [
     { _id: 1, subject: "coffee", author: "xyz", views: 50 },
     { _id: 2, subject: "Coffee Shopping", author: "efg", views: 5 },
     { _id: 3, subject: "Baking a cake", author: "abc", views: 90  },
     { _id: 4, subject: "baking", author: "xyz", views: 100 },
     { _id: 5, subject: "Café Con Leche", author: "abc", views: 200 },
     { _id: 6, subject: "Сырники", author: "jkl", views: 80 },
     { _id: 7, subject: "coffee and cream", author: "efg", views: 10 },
     { _id: 8, subject: "Cafe con Leche", author: "xyz", views: 10 }
   ]
)
```

搜索一个单词

下面的查询指定了咖啡的$search 字符串:

```
db.articles.find( { $text: { $search: "coffee" } } )
```

这个查询返回在索引的 subject 字段中包含术语 coffee 的文档，或者更准确地说，返回单词的词根版本:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

匹配任何搜索项

如果搜索字符串是空格分隔的字符串，则$text 操作符对每个术语执行逻辑或搜索，并返回包含任何术语的文档。

下面的查询指定了一个$search 字符串，包含三个以空格分隔的单词“bake coffee cake”:

```
db.articles.find( { $text: { $search: "bake coffee cake" } } )
```

这个查询返回在索引主题字段中包含 bake 或 coffee 或 cake 的文档，或者更准确地说是这些单词的词根:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
{ "_id" : 3, "subject" : "Baking a cake", "author" : "abc", "views" : 90 }
{ "_id" : 4, "subject" : "baking", "author" : "xyz", "views" : 100 }
```

搜索一个短语

要将确切的短语匹配为单个术语，请转义引号。

搜索短语 coffee shop 的查询如下:

```
db.articles.find( { $text: { $search: "\"coffee shop\"" } } )
```

E:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

排除包含术语的文档

负项是一个以负号-为前缀的项。如果您否定一个术语，$text 操作符将从结果中排除包含这些术语的文档。
下面的示例搜索包含单词 coffee 但不包含单词 shop 的文档, 或者更准确地说是单词的词根版本:

```
db.articles.find( { $text: { $search: "coffee -shop" } } )
```

E:

```
{ "_id" : 7, "subject" : "coffee and cream", "author" : "efg", "views" : 10 }
{ "_id" : 1, "subject" : "coffee", "author" : "xyz", "views" : 50 }
```

寻找一种不同的语言
在$text 表达式中使用可选的$language 字段指定一种语言，该语言确定停止单词列表以及搜索字符串的 stemmer 和 tokenizer 的规则。

如果指定语言值为“none”，则文本搜索使用简单的标记化，没有停止单词列表，也没有词干分析。

下面的查询指定 es，即西班牙语，作为确定标记化、词干分析和停止词的语言:

```
db.articles.find(
    { $text: { $search: "leche", $language: "es" } }
)
```

R:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz", "views" : 10 }
```

区分大小写搜索一个术语
下面的查询对术语 Coffee 执行区分大小写的搜索:

```
db.articles.find( { $text: { $search: "Coffee", $caseSensitive: true } } )
```

R:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg", "views" : 5 }
```

区分大小写搜索短语
以下查询对短语 Cafe Con Leche 执行区分大小写的搜索:

```
db.articles.find( {
    $text: { $search: "\"Café Con Leche\"", $caseSensitive: true }
} )
```

R:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc", "views" : 200 }
```

带有否定项的大小写敏感性
负项是一个以负号-为前缀的项。如果您否定一个术语，$text 操作符将从结果中排除包含这些术语的文档。还可以为否定项指定大小写敏感性。
下面的示例对包含单词 Coffee 但不包含小写词 shop 的文档执行区分大小写的搜索，或者更准确地说，不包含单词的词根版本:

```
db.articles.find( { $text: { $search: "Coffee -shop", $caseSensitive: true } } )
```

R:

```
{ "_id" : 2, "subject" : "Coffee Shopping", "author" : "efg" }
```

可区别的敏感搜索
在 3.2 版本中进行了更改。

要对 version 3 文本索引启用 diacriticSensitive 搜索，请指定$diacriticSensitive: true。指定 $diacriticSensitive: true 可能会影响性能。
对术语进行变音符号敏感搜索
下面的查询对术语 CAFE 或更准确地说是这个单词的词根版本执行一个变音符号敏感文本搜索:

```
db.articles.find( { $text: { $search: "CAFÉ", $diacriticSensitive: true } } )
```

R:

```
{ "_id" : 5, "subject" : "Café Con Leche", "author" : "abc" }
```

带否定项的变音符号灵敏度
$diacriticSensitive 选项也适用于否定的术语。负项是一个以负号-为前缀的项。如果您否定一个术语，$text 操作符将从结果中排除包含这些术语的文档。

下面的查询对包含术语 leches 但不包含术语 cafe 的文档执行变音符号敏感文本搜索，或者更准确地说是单词的词根版本:

```
db.articles.find(
   { $text: { $search: "leches -cafés", $diacriticSensitive: true } }
)
```

R:

```
{ "_id" : 8, "subject" : "Cafe con Leche", "author" : "xyz" }
```

下面的查询搜索术语 cake 并返回分配给每个匹配文档的分数:

```
db.articles.find(
    { $text: { $search: "cake" } },
    { score: { $meta: "textScore" } }
)
```

返回的文档包含一个额外的字段得分，该字段得分包含与文本搜索关联的文档得分。[1]
按文本搜索得分排序
要按文本得分排序，在投影文档和排序表达式中都包含相同的$meta 表达式。以下查询搜索术语 coffee，并按降序得分对结果进行排序:

```
db.articles.find(
    { $text: { $search: "coffee" } },
    { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

查询返回按降序排序的匹配文档。
返回前 2 个匹配的文档
使用 limit()方法结合 sort()返回最前面的 n 个匹配文档。
下面的查询搜索术语 coffee 并按降序得分对结果进行排序，将结果限制在前两个匹配的文档中:

```
db.articles.find(
    { $text: { $search: "coffee" } },
    { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } ).limit(2)
```

带有附加查询和排序表达式的文本搜索
下面的查询搜索作者等于“xyz”且索引字段主题包含术语 coffee 或 bake 的文档。该操作还指定了升序日期排序，然后降序文本搜索得分:

```
db.articles.find(
    { author: "xyz", $text: { $search: "coffee bake" } },
    { score: { $meta: "textScore" } }
).sort( { date: 1, score: { $meta: "textScore" } } )
```

$where

使用$where 操作符将包含 JavaScript 表达式的字符串或完整的 JavaScript 函数传递给查询系统。$where 提供了更大的灵活性，但要求数据库处理集合中每个文档的 JavaScript 表达式或函数。使用 this 或 obj 在 JavaScript 表达式或函数中引用文档。
E:

```
{
    _id: 12378,
    name: "Steve",
    username: "steveisawesome",
    first_login: "2017-01-01"
}
{
    _id: 2,
    name: "Anya",
    username: "anya",
    first_login: "2001-02-02"
}
```

下面的示例使用$where 和 hex_md5() JavaScript 函数将 name 字段的值与 MD5 散列进行比较，并返回任何匹配的文档。

```
db.players.find( { $where: function() {
    return (hex_md5(this.name) == "9b53e667f30cd329dca1ec9e6a83e994")
} } );
```

R:

```
{
    "_id" : 2,
    "name" : "Anya",
    "username" : "anya",
    "first_login" : "2001-02-02"
}
```

地理空间查询操作符

Operators;
查询选择器:
名称描述
$ geointersects 选择与 GeoJSON 几何相交的几何图形。2dsphere 索引支持

$geoIntersects。
$ geowithin　　选择绑定 GeoJSON 几何中的几何图形。2dsphere 和 2d 索引支持$geoWithin。
$near　　返回接近某个点的地理空间对象。需要地理空间索引。2dsphere 和 2d 索引支持$near。
$near sphere　　返回接近球体上某个点的地理空间对象。需要地理空间索引。2dsphere 和 2d 索引支持$nearSphere。

几何说明符

名称描述
$box　　为$geoWithin 查询指定一个使用遗留坐标对的矩形框。2d 索引支持$box。
$center:　使用平面几何时，$center 使用遗留坐标对到$geoWithin 查询指定一个圆。2d 索引支持$center。
$centerSphere。　在使用球面几何时，$geoWithin 查询使用遗留坐标对或 GeoJSON 格式指定一个圆。2dsphere 和 2d 索引支持$centerSphere。
$geometry　　向地理空间查询操作符指定一个 GeoJSON 格式的几何图形。
$ maxdistance　　指定一个最大距离来限制$near 和$near sphere 查询的结果。2dsphere 和 2d 索引支持$maxDistance。
$ mindistance　　指定一个最小距离来限制$near 和$near sphere 查询的结果。只适用于 2dsphere 索引。
$polygon　　为$geoWithin 查询指定一个使用遗留坐标对的多边形。2d 索引支持$center。
$uniquedocs　　弃用。修改$geoWithin 和$near 查询，以确保即使文档多次匹配查询，查询也只返回文档一次。

$geoIntersects

选择地理空间数据与指定的 GeoJSON 对象相交的文档;即数据与指定对象的交集为非空。$geoIntersects 操作符使用$geometry 操作符指定 GeoJSON 对象。要使用默认坐标引用系统(CRS)指定一个或多个 GeoJSON 多边形，请使用以下语法:

```
{
  <location field>: {
      $geoIntersects: {
          $geometry: {
              type: "<GeoJSON object type>" ,
              coordinates: [ <coordinates> ]
          }
      }
  }
}
```

Intersects a Polygon

下面的示例使用$geoIntersects 选择与坐标数组定义的多边形相交的所有 loc 数据。多边形的面积小于一个半球的面积:

```
db.places.find(
    {
      loc: {
        $geoIntersects: {
            $geometry: {
                type: "Polygon" ,
                coordinates: [
                  [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ]
                ]
            }
        }
      }
    }
)
```

对于面积大于单半球的单环多边形，请参见与“大”多边形相交。

与一个“大”多边形相交

要使用面积大于单个半球的单圈 GeoJSON 多边形进行查询, $geometry 表达式必须指定定制的 MongoDB 坐标引用系统。例如:

```
db.places.find(
    {
      loc: {
        $geoIntersects: {
            $geometry: {
                type : "Polygon",
                coordinates: [
                  [
                    [ -100, 60 ], [ -100, 0 ], [ -100, -60 ], [ 100, -60 ], [ 100, 60 ], [ -100, 60 ]
                  ]
                ],
                crs: {
                    type: "name",
                    properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
                }
            }
        }
      }
    }
)
```

$geoWithin

选择具有完全在指定形状中存在的地理空间数据的文档。
指定的形状可以是 GeoJSON 多边形(单圈或多圈)、GeoJSON 多多边形，也可以是由遗留坐标对定义的形状。$geoWithin 操作符使用$geometry 操作符指定 GeoJSON 对象。
要使用默认坐标引用系统(CRS)指定一个或多个 GeoJSON 多边形，请使用以下语法:

```
{
    <location field>: {
        $geoWithin: {
            $geometry: {
                type: <"Polygon" or "MultiPolygon"> ,
                coordinates: [ <coordinates> ]
            }
        }
    }
}
```

对于指定区域大于单个半球的 GeoJSON 几何图形的$geoWithin 查询，使用默认的 CRS 可以查询互补几何图形。
新版本 3.0:要指定带有自定义 MongoDB CRS 的单圈 GeoJSON 多边形，请使用以下原型，它在$geometry 表达式中指定自定义 MongoDB CRS:

```
{
    <location field>: {
        $geoWithin: {
            $geometry: {
               type: "Polygon" ,
               coordinates: [ <coordinates> ],
               crs: {
                   type: "name",
                   properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
               }
            }
        }
    }
}
```

E:
在一个多边形
下面的示例选择完全存在于 GeoJSON 多边形中的所有 loc 数据。多边形的面积小于一个半球的面积:

```
db.places.find(
    {
      loc: {
        $geoWithin: {
            $geometry: {
                type : "Polygon" ,
```

```
                coordinates: [ [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ] ]
            }
        }
    }
}
)
```

对于面积大于单半球的单环多边形，请参见“大”多边形内。

在一个“大”多边形内

要使用面积大于单个半球的单圈 GeoJSON 多边形进行查询，$geometry 表达式必须指定定制的 MongoDB 坐标引用系统。例如：

```
db.places.find(
    {
      loc: {
        $geoWithin: {
            $geometry: {
                type : "Polygon" ,
                coordinates: [
                  [
                    [ -100, 60 ], [ -100, 0 ], [ -100, -60 ], [ 100, -60 ], [ 100, 60 ], [ -100, 60 ]
                  ]
                ],
                crs: {
                    type: "name",
                    properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
                }
            }
        }
    }
}
)
```

$near

指定地理空间查询从最近到最远返回文档的点。$near 操作符可以指定 GeoJSON 点或遗留坐标点。

$near 需要地理空间索引：

如果指定了一个 GeoJSON 点，

如果使用遗留坐标指定一个点，则使用 2d 索引。

要指定 GeoJSON 点，$near 操作符需要一个 2dsphere 索引，并具有以下语法：

```
{
    <location field>: {
      $near: {
        $geometry: {
```

```
            type: "Point" ,
            coordinates: [ <longitude> , <latitude> ]
        },
        $maxDistance: <distance in meters>,
        $minDistance: <distance in meters>
      }
   }
}
```

如果指定经纬度坐标，首先列出经度，然后列出纬度:
有效经度值在-180 和 180 之间，两者都包括在内。
有效纬度值在-90 和 90 之间，两者都包括在内。
在指定 GeoJSON 点时，可以使用可选的$minDistance 和$maxDistance 规范来限制$near 结果的距离(以米为单位):
$minDistance 将结果限制为那些至少与中心点指定距离的文档。
新版本 2.6。
$maxDistance 将结果限制为那些与中心点最多指定距离的文档。
要使用遗留坐标指定一个点，$near 需要一个 2d 索引，并具有以下语法:

```
{
  $near: [ <x>, <y> ],
  $maxDistance: <distance in radians>
}
```

例子
查询 GeoJSON 数据
重要的
如果指定经纬度坐标，首先列出经度，然后列出纬度:
有效经度值在-180 和 180 之间，两者都包括在内。
有效纬度值在-90 和 90 之间，两者都包括在内。
考虑一个具有 2dsphere 索引的集合位置。
下面的示例返回距离指定 GeoJSON 点至少 1000 米和最多 5000 米的文档，从最近到最远排序:

```
db.places.find(
   {
     location:
       { $near :
          {
            $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
            $minDistance: 1000,
            $maxDistance: 5000
          }
       }
   }
)
```

查询遗留坐标
重要的

如果指定经纬度坐标，首先列出经度，然后列出纬度:
有效经度值在-180 和 180 之间，两者都包括在内。
有效纬度值在-90 和 90 之间，两者都包括在内。
考虑一个具有 2d 索引的集合 legacy2d。
下面的示例返回的文档最多是来自指定的遗留坐标对的 0.10 弧度，从最近到最远排序:

```
db.legacy2d.find(
    { location : { $near : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }
)
```

$nearSphere

指定地理空间查询从最近到最远返回文档的点。MongoDB 使用球面几何计算$nearSphere 的距离。
$nearSphere 需要地理空间索引:
2dsphere 索引，用于定义为 GeoJSON 点的位置数据
定义为遗留坐标对的位置数据的 2d 索引。要在 GeoJSON 点上使用 2d 索引, 请在 GeoJSON 对象的 coordinates 字段上创建索引。
$nearSphere 操作符可以指定 GeoJSON 点或遗留坐标点。
要指定 GeoJSON 点，请使用以下语法:

```
{
  $nearSphere: {
      $geometry: {
          type : "Point",
          coordinates : [ <longitude>, <latitude> ]
      },
      $minDistance: <distance in meters>,
      $maxDistance: <distance in meters>
  }
}
```

可选的$minDistance 将结果限制为那些至少与中心点指定距离的文档。
新版本 2.6。
两个索引都可以使用可选的$maxDistance。
要使用遗留坐标指定一个点，请使用以下语法:

```
{
  $nearSphere: [ <x>, <y> ],
  $minDistance: <distance in radians>,
  $maxDistance: <distance in radians>
}
```

只有当查询使用 2dsphere 索引时，才可以使用可选的$minDistance。$minDistance 将结果限制为那些至少与中心点指定距离的文档。
新版本 2.6。
两个索引都可以使用可选的$maxDistance。
如果您对遗留坐标使用经度和纬度，请首先指定经度，然后指定纬度。
另请参阅

2d 索引和地理空间近查询
E:
使用 GeoJSON 指定中心点
考虑一个集合位置，它包含一个 location 字段和一个 2dsphere 索引的文档。
然后，下面的示例返回其位置距离指定点至少 1000 米，距离指定点最多 5000 米，从最近到最远排序:

```
db.places.find(
   {
      location: {
         $nearSphere: {
            $geometry: {
               type : "Point",
               coordinates : [ -73.9667, 40.78 ]
            },
            $minDistance: 1000,
            $maxDistance: 5000
         }
      }
   }
)
```

使用遗留坐标指定中心点
二维指数
考虑一个集合 legacyPlaces，它在 location 字段中包含具有遗留坐标对的文档，并且具有一个 2d 索引。
然后，下面的例子返回那些位置从指定点到指定点的距离最多为 0.10 弧度的文档，按从最近到最远的顺序排列:\

```
db.legacyPlaces.find(
   { location : { $nearSphere : [ -73.9667, 40.78 ], $maxDistance: 0.10 } }
)
```

2 dsphere 指数
如果集合具有 2dsphere 索引，则还可以指定可选的$minDistance 规范。例如，下面的示例返回位置至少为 0.0004 弧度(从最近到最远)的文档:

$box

为地理空间$geoWithin 查询指定一个矩形，根据其基于点的位置数据返回矩形范围内的文档。当与$box 操作符一起使用时，$geoWithin 根据网格坐标返回文档，并且不查询 GeoJSON 形状。
要使用$box 操作符，必须在数组对象中指定矩形的左下角和右上角:

```
{
  <location field>: {
     $geoWithin: {
        $box: [
```

```
            [ <bottom left coordinates> ],
            [ <upper right coordinates> ]
        ]
    }
  }
}
```

E:下面的示例查询返回框内的所有文档，这些文档的点分别是:[0,0]、[0,100]、[100,0]和[100,100]。

```
db.places.find( {
    loc: { $geoWithin: { $box:  [ [ 0, 0 ], [ 100, 100 ] ] } }
} )
```

$center

$center 操作符为$geoWithin 查询指定一个圆圈。查询返回位于圆圈范围内的遗留坐标对。操作符不返回 GeoJSON 对象。

要使用$center 操作符，请指定一个数组，其中包含:

圆中心点的网格坐标

圆的半径，用坐标系中使用的单位表示。

```
{
    <location field>: {
        $geoWithin: { $center: [ [ <x>, <y> ] , <radius> ] }
    }
}
```

下面的示例查询返回所有文档，这些文档的坐标位于以[-74,40.74]为中心的圆心内，半径为10:

```
db.places.find(
    { loc: { $geoWithin: { $center: [ [-74, 40.74], 10 ] } } }
)
```

$centerSphere

为使用球面几何的地理空间查询定义一个圆。查询返回在圆圈范围内的文档。您可以在GeoJSON 对象和遗留坐标对上使用$centerSphere 操作符。

若要使用$centerSphere，请指定包含以下内容的数组:

圆中心点的网格坐标

圆的半径以弧度为单位。要计算弧度，请参阅使用球面几何计算距离。

```
{
    <location field>: {
        $geoWithin: { $centerSphere: [ [ <x>, <y> ], <radius> ] }
    }
}
```

例子
下面的示例查询网格坐标, 并返回经度 88 W 和纬度 30 N 范围内半径为 10 英里的所有文档。该查询通过除以地球赤道半径 3963.2 英里将距离转换为弧度:

```
db.places.find( {
   loc: { $geoWithin: { $centerSphere: [ [ -88, 30 ], 10/3963.2 ] } }
} )
```

$geometry

在 3.0 版本中进行了更改:添加了对指定面积大于单个半球的单圈 GeoJSON 多边形的支持。
$geometry 操作符指定一个 GeoJSON 几何，用于以下地理空间查询操作符:$geoWithin、$geoIntersects、$near 和$near sphere。$geometry 使用 EPSG:4326 作为默认坐标参考系统(CRS)。
要指定具有默认 CRS 的 GeoJSON 对象，请使用$geometry 的以下原型:

```
$geometry: {
    type: "<GeoJSON object type>",
    coordinates: [ <coordinates> ]
}
```

新版本 3.0:要指定带有自定义 MongoDB CRS 的单圈 GeoJSON 多边形, 请使用以下原型(仅适用于$geoWithin 和$geoIntersects):

```
$geometry: {
    type: "Polygon",
    coordinates: [ <coordinates> ],
    crs: {
        type: "name",
        properties: { name: "urn:x-mongodb:crs:strictwinding:EPSG:4326" }
    }
}
```

自定义 MongoDB 坐标参考系统具有严格的逆时针绕线顺序。

$maxDistance

$maxDistance 操作符将地理空间$near 或$near sphere 查询的结果限制为指定的距离。最大距离的测量单位由所使用的坐标系决定。对于 GeoJSON point 对象，指定以米为单位的距离，而不是弧度。
版本.6 中的更改:为$maxDistance 指定一个非负数。
2dsphere 和 2d 地理空间索引都支持$maxDistance:。
下面的示例查询返回位置值为[-74,40]处 10 个或更少单元的文档。
例子

```
db.places.find( {
    loc: { $near: [ -74 , 40 ],   $maxDistance: 10 }
} )
```

$minDistance

将地理空间$near 或$near sphere 查询的结果筛选到至少与中心点指定距离的文档。
如果$near 或$near sphere 查询将中心点指定为 GeoJSON 点，则将距离指定为以米为单位的非负。
如果$nearSphere 查询将中心点指定为遗留坐标对，则将距离指定为弧度中的非负数。如果查询将中心点指定为 GeoJSON 点，则$near 只能使用 2dsphere 索引。
考虑一个具有 2dsphere 索引的集合位置。
下面的示例返回距离指定 GeoJSON 点至少 1000 米和最多 5000 米的文档，从最近到最远排序:

```
db.places.find(
   {
     location:
       { $near :
          {
            $geometry: { type: "Point",  coordinates: [ -73.9667, 40.78 ] },
            $minDistance: 1000,
            $maxDistance: 5000
          }
       }
   }
)
```

使用$nearSphere
考虑一个集合位置，它包含一个 location 字段和一个 2dsphere 索引的文档。
然后，下面的示例返回其位置距离指定点至少 1000 米，距离指定点最多 5000 米，从最近到最远排序:

```
db.places.find(
   {
     location: {
         $nearSphere: {
            $geometry: {
               type : "Point",
               coordinates : [ -73.9667, 40.78 ]
            },
            $minDistance: 1000,
            $maxDistance: 5000
         }
     }
   }
)
```

$polygon

为遗留坐标对上的地理空间$geoWithin 查询指定一个多边形。查询返回多边形范围内的对。该操作符不查询 GeoJSON 对象。

要定义多边形，请指定一个坐标点数组:

```
{
    <location field>: {
        $geoWithin: {
            $polygon: [ [ <x1> , <y1> ], [ <x2> , <y2> ], [ <x3> , <y3> ], ... ]
        }
    }
}
```

例子

下面的查询返回在[0,0]、[3,6]和[6,0]定义的多边形中存在坐标的所有文档:

```
db.places.find(
  {
      loc: {
        $geoWithin: { $polygon: [ [ 0 , 0 ], [ 3 , 6 ], [ 6 , 0 ] ] }
      }
  }
)
```

$uniqueDocs

2.6 中就没有使用了。

数组查询操作

名称描述

$all:　　匹配包含查询中指定的所有元素的数组。

$elemMatch:　如果数组字段中的元素匹配所有指定的$elemMatch 条件，则选择 document。

$size　　如果数组字段是指定的大小，则选择文档。

$all

$all 操作符选择字段值为包含所有指定元素的数组的文档。要指定$all 表达式，请使用以下原型:

```
{ <field>: { $all: [ <value1> , <value2> ... ] } }
```

相当于$和 Operation
在 2.6 版本中进行了更改。

```
db.coutry.find({country:{$not:{$all:["日本","韩国"]}}})
```

```
WriteResult({ "nInserted" : 1 })
> db.coutry.find()
{ "_id" : ObjectId("5d9bf801921feaec178e4f5a"), "country" : [ "日本", "韩国", "美国
{ "_id" : ObjectId("5d9bf823921feaec178e4f5b"), "country" : [ "韩国", "美国" ] }
{ "_id" : ObjectId("5d9bf8d6921feaec178e4f5c"), "country" : [ "韩国,美国" ] }
{ "_id" : ObjectId("5d9bfba1921feaec178e4f5d"), "country" : [ "中国", "美国" ] }
{ "_id" : ObjectId("5d9bff26921feaec178e4f5e"), "country" : [ "日本" ] }
{ "_id" : ObjectId("5d9c2e87921feaec178e4f5f"), "country" : [ "韩国", "日本" ] }
{ "_id" : ObjectId("5d9c2e95921feaec178e4f60"), "country" : [ "日本", "韩国" ] }
> db.coutry.find({country:{$not:{$all:["日本","韩国"]}}})
{ "_id" : ObjectId("5d9bf823921feaec178e4f5b"), "country" : [ "韩国", "美国" ] }
{ "_id" : ObjectId("5d9bf8d6921feaec178e4f5c"), "country" : [ "韩国,美国" ] }
{ "_id" : ObjectId("5d9bfba1921feaec178e4f5d"), "country" : [ "中国", "美国" ] }
{ "_id" : ObjectId("5d9bff26921feaec178e4f5e"), "country" : [ "日本" ] }
> use
```

$all 等价于指定值的$和运算;即下列声明:

{ tags: { $all: ["ssl" , "security"] } }

等价于:

{ $and: [{ tags: "ssl" }, { tags: "security" }] }

嵌套的数组

在 2.6 版本中进行了更改。

当传递一个嵌套数组的数组(例如[[" a "]])时，$all 现在可以匹配字段包含嵌套数组作为元素的文档(例如 field: [[" a "], …)，或字段等于嵌套数组(例如:field: ["A"])。

例如，考虑以下查询[1]:

db.articles.find({ tags: { $all: [["ssl", "security"]] } })

等价于:

db.articles.find({ $and: [{ tags: ["ssl", "security"] }] })

等价于:

db.articles.find({ tags: ["ssl", "security"] })

因此，$all 表达式可以匹配文档，其中 tags 字段是一个包含嵌套数组["ssl"， "security"]的数组，或者是一个等于嵌套数组的数组:

tags: [["ssl", "security"], ...]

tags: ["ssl", "security"]

E:

以下示例使用包含文档的库存集合:

```
{
    _id: ObjectId("5234cc89687ea597eabee675"),
    code: "xyz",
    tags: [ "school", "book", "bag", "headphone", "appliance" ],
    qty: [
            { size: "S", num: 10, color: "blue" },
            { size: "M", num: 45, color: "blue" },
```

```
            { size: "L", num: 100, color: "green" }
        ]
}

{
    _id: ObjectId("5234cc8a687ea597eabee676"),
    code: "abc",
    tags: [ "appliance", "school", "book" ],
    qty: [
            { size: "6", num: 100, color: "green" },
            { size: "6", num: 50, color: "blue" },
            { size: "8", num: 100, color: "brown" }
        ]
}

{
    _id: ObjectId("5234ccb7687ea597eabee677"),
    code: "efg",
    tags: [ "school", "book" ],
    qty: [
            { size: "S", num: 10, color: "blue" },
            { size: "M", num: 100, color: "blue" },
            { size: "L", num: 100, color: "green" }
        ]
}

{
    _id: ObjectId("52350353b2eff1353b349de9"),
    code: "ijk",
    tags: [ "electronics", "school" ],
    qty: [
            { size: "M", num: 100, color: "green" }
        ]
}
```

使用$all 匹配值

下面的操作使用$all 操作符查询库存集合，其中 tags 字段的值是一个数组，其元素包括 appliance、school 和 book:

```
db.inventory.find( { tags: { $all: [ "appliance", "school", "book" ] } } )
```

R:

```
{
    _id: ObjectId("5234cc89687ea597eabee675"),
    code: "xyz",
    tags: [ "school", "book", "bag", "headphone", "appliance" ],
    qty: [
```

```
            { size: "S", num: 10, color: "blue" },
            { size: "M", num: 45, color: "blue" },
            { size: "L", num: 100, color: "green" }
        ]
}

{
    _id: ObjectId("5234cc8a687ea597eabee676"),
    code: "abc",
    tags: [ "appliance", "school", "book" ],
    qty: [
            { size: "6", num: 100, color: "green" },
            { size: "6", num: 50, color: "blue" },
            { size: "8", num: 100, color: "brown" }
        ]
}
```

使用$all 和$elemMatch

如果字段包含文档数组，可以使用$all 和$ element match 操作符。

下面的操作查询库存集合，查找 qty 字段的值是一个数组，其元素匹配$elemMatch 条件:

```
db.inventory.find( {
                        qty: { $all: [
                                        { "$elemMatch" : { size: "M", num: { $gt: 50} } },
                                        { "$elemMatch" : { num : 100, color: "green" } }
                                    ] }
                    } )
```

E:

```
{
    "_id" : ObjectId("5234ccb7687ea597eabee677"),
    "code" : "efg",
    "tags" : [ "school", "book"],
    "qty" : [
                { "size" : "S", "num" : 10, "color" : "blue" },
                { "size" : "M", "num" : 100, "color" : "blue" },
                { "size" : "L", "num" : 100, "color" : "green" }
            ]
}

{
    "_id" : ObjectId("52350353b2eff1353b349de9"),
    "code" : "ijk",
    "tags" : [ "electronics", "school" ],
    "qty" : [
                { "size" : "M", "num" : 100, "color" : "green" }
            ]
```

}
$all 操作符的存在是为了支持数组上的查询。但是您可以使用$all 操作符来选择非数组字段，如下面的示例所示:
db.inventory.find({ "qty.num": { $all: [50] } })
但是，请使用下面的表单来表示相同的查询:
db.inventory.find({ "qty.num" : 50 })
这两个查询都将选择 inventory 集合中 num 字段值为 50 的所有文档。

$elemMatch

这两个查询都将选择 inventory 集合中 num 字段值为 50 的所有文档。
{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }
如果在$elemMatch 表达式中只指定一个<query>条件，则不需要使用$elemMatch。

行为
不能在$ elements 匹配中指定$where 表达式。
不能在$elemMatch 中指定$text 查询表达式。
E:score collection
{ _id: 1, results: [82, 85, 88] }
{ _id: 2, results: [75, 88, 89] }
下面的查询只匹配结果数组中至少包含一个大于或等于 80 且小于 85 的元素的文档。
```
db.scores.find(
    { results: { $elemMatch: { $gte: 80, $lt: 85 } } }
)
```
查询返回以下文档，因为元素 82 大于或等于 80，小于 85
{ "_id" : 1, "results" : [82, 85, 88] }
嵌入文档数组
鉴于调查收集的下列文件:
{ _id: 1, results: [{ product: "abc", score: 10 }, { product: "xyz", score: 5 }] }
{ _id: 2, results: [{ product: "abc", score: 8 }, { product: "xyz", score: 7 }] }
{ _id: 3, results: [{ product: "abc", score: 7 }, { product: "xyz", score: 8 }] }
下面的查询只匹配那些结果数组中包含至少一个元素，且产品都等于“
xyz”且得分大于或等于 8 的文档。
```
db.survey.find(
    { results: { $elemMatch: { product: "xyz", score: { $gte: 8 } } } }
)
```
具体来说，查询匹配以下文档:
{ "_id" : 3, "results" : [{ "product" : "abc", "score" : 7 }, { "product" : "xyz", "score" : 8 }] }
单查询条件
如果在$elemMatch 表达式中指定一个查询谓词，则不需要$elemMatch。
例如，考虑下面的例子，其中$elemMatch 只指定一个查询谓词{product: "xyz"}:
```
db.survey.find(
    { results: { $elemMatch: { product: "xyz" } } }
)
```

由于$elemMatch 只指定一个条件，所以不需要使用$elemMatch 表达式，您可以使用以下查询:

```
db.survey.find(
    { "results.product": "xyz" }
)
```

$size

$size 操作符匹配任何与参数指定的元素数量相匹配的数组。例如:

db.collection.find({ field: { $size: 2 } });

返回集合中的所有文档，其中 field 是一个包含 2 个元素的数组。例如，上面的表达式将返回{field: [red, green]}和{field: [apple, lime]}，但不返回{field: fruit}或{field: [orange, lemon, 柚子]}。若要匹配数组中只有一个元素的字段，请使用值为 1 的$size，如下所示:

db.collection.find({ field: { $size: 1 } });

$size 不接受值范围。要基于具有不同数量元素的字段选择文档，请创建一个计数器字段，当您向字段添加元素时，该字段将递增。

查询不能为查询的$size 部分使用索引，尽管查询的其他部分可以在适用时使用索引。

位查询操作符

名称描述

$ bitsallclear 匹配一组位位置都为 0 的数值或二进制值。

$ **bitsallset** 匹配数字或二进制值，其中一组位位置的值都为 1。

$ bitsanyclear 匹配数字或二进制值，其中来自一组位位置的任何位的值都为 0。

$ bitsanyset 匹配数字或二进制值，其中来自一组位位置的任何位的值都为 1。

Bit Value	1	1	1	1	1	1	1	0
Position	7	6	5	4	3	2	1	0

$bitsAllClear

$bitsAllClear 匹配字段中查询给出的所有位位置都为 clear(即 0)的文档。

{ <field>: { $bitsAllClear: <numeric bitmask> } }

{ <field>: { $bitsAllClear: < BinData bitmask> } }

{ <field>: { $bitsAllClear: [<position1>, <position2>, ...] } }

字段值必须是 numeric 或 BinData 实例。否则，$bitsAllClear 将不匹配当前文档。

数字位掩码

您可以提供一个数字位掩码来匹配操作数字段。它必须可以表示为一个非负 32 位带符号整数。否则，$bitsAllClear 将返回一个错误。

BinData 位掩码

还可以使用任意大的 BinData 实例作为位掩码。

职位列表
如果查询位位置列表，每个<position>必须是非负整数。位的位置从最小有效位开始为 0。
例如，十进制数字 254 将具有以下位
E:
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
下面的查询使用$bitsAllClear 操作符来测试字段 a 在位置 1 和位置 5 处是否有清除的位，其中最不重要的位是位置 0。
db.collection.find({ a: { $bitsAllClear: [1, 5] } })
查询匹配以下文件:
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20, "binaryValueofA" : "00010100" }

$bitsallset

新版本 3.2。

$bitsAllSet 匹配字段中设置查询给出的所有位位置(即 1)的文档。

{<field>: {$bitsAllSet: <numeric bitmask>}}
{<field>: {$bitsAllSet: < BinData bitmask>}}
{<field>: {$bitsAllSet: [<position1>, <position2>, ⋯]}}
字段值必须是 numeric 或 BinData 实例。否则，$bitsAllSet 将不匹配当前文档。

数字位掩码
您可以提供一个数字位掩码来匹配操作数字段。它必须可以表示为一个非负 32 位带符号整数。否则，$bitsAllSet 将返回一个错误。
BinData 位掩码
还可以使用任意大的 BinData 实例作为位掩码。
职位列表
如果查询位位置列表，每个<position>必须是非负整数。位的位置从最小有效位开始为 0。
例如，小数 254 的位元位置如下:
E:
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
下面的查询使用$bitsAllSet 操作符来测试字段 a 是否在位置 1 和位置 5 处设置了位，其中最不重要的位是位置 0。
db.collection.find({ a: { $bitsAllSet: [1, 5] } })
R:
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }

{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
整数位掩码
下面的查询使用$bitsAllSet 操作符来测试字段 a 是否在位置 1、4 和 5 处设置了位(位掩码 50 的二进制表示形式是 00110010)。
db.collection.find({ a: { $bitsAllSet: 50 } })
R:
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
BinData 位掩码
下面的查询使用$bitsAllSet 操作符来测试字段 a 在位置 4 和 5 处是否设置了位(BinData 的二进制表示形式(0, “MC==”)为 00110000)。
E:
db.collection.find({ a: { $bitsAllSet: BinData(0, "MC==") } })
R:
{ _id: 1, a: 54, binaryValueofA: "00110110" }

$bitsanyclear

$bitsAnyClear 匹配字段中查询给出的任何位位置为 clear(即 0)的文档。

{<字段>:{$bitsAnyClear: <数值位掩码>}}
{<field>: {$bitsAnyClear: < BinData bitmask>}}
{<field>: {$bitsAnyClear: [<position1>, <position2>, ...]}}
字段值必须是 numeric 或 BinData 实例。否则，$bitsAnyClear 将不匹配当前文档。

数字位掩码
您可以提供一个数字位掩码来匹配操作数字段。它必须可以表示为一个非负 32 位带符号整数。否则，$bitsAnyClear 将返回一个错误。
BinData 位掩码
还可以使用任意大的 BinData 实例作为位掩码。
职位列表
如果查询位位置列表，每个<position>必须是非负整数。位的位置从最小有效位开始为 0。例如，小数 254 的位元位置如下:
符号扩展
数字是加号的。例如，$bitsAnyClear 认为将位位置 200 设置为负数-5，但是将位位置 200 设置为正数+5。

相反，BinData 实例是零扩展的。例如，给定以下文件:
db.collection.save({ x: BinData(0, "ww=="), binaryValueofA: "11000011" })
E:
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
一些位置数组

下面的查询使用$bitsAnyClear 操作符来测试字段 a 的位位置是否为 1 或 5，其中最不重要的位是 0。
db.collection.find({ a: { $bitsAnyClear: [1, 5] } })
R:
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
整数位掩码
下面的查询使用$bitsAnyClear 操作符来测试字段 a 在 0、1 和 5 的位置是否有任何位清除(位掩码 35 的二进制表示形式是 00100011)。
db.collection.find({ a: { $bitsAnyClear: 35 } })
R:
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
BinData 位掩码
下面的查询使用$bitsAnyClear 操作符来测试字段 a 在第 4 和第 5 位置是否有任何位是清除的(BinData 的二进制表示形式(0，“MC==”)是 00110000)。
db.collection.find({ a: { $bitsAnyClear: BinData(0, "MC==") } })
R:
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }

$bitsanyset

新版本 3.2。

$bitsAnySet 匹配在字段中设置查询给出的任何位位置(即 1)的文档。

{<field>: {$bitsAnySet: <numeric bitmask>}}
{<field>: {$bitsAnySet: < BinData bitmask>}}
{<field>: {$bitsAnySet: [<position1>， <position2>，…]}}
字段值必须是 numeric 或 BinData 实例。否则，$bitsAnySet 将不匹配当前文档。

数字位掩码
您可以提供一个数字位掩码来匹配操作数字段。它必须可以表示为一个非负 32 位带符号整数。否则，$bitsAnySet 将返回一个错误。
BinData 位掩码
还可以使用任意大的 BinData 实例作为位掩码。
职位列表
如果查询位位置列表，每个<position>必须是非负整数。位的位置从最小有效位开始为 0。
例如，十进制数字 254 的位位置如下
索引

查询不能为查询的$bitsAnySet 部分使用索引，尽管查询的其他部分可以使用索引(如果适用的话)。
浮点值
$bitsAnySet 将不匹配不能表示为带符号 64 位整数的数值。如果一个值太大或太小，无法放入带符号的 64 位整数中，或者它具有小数部分，则可能出现这种情况。

符号扩展
数字是加号的。例如，$bitsAnySet 认为为负数-5 设置了位位置 200，但是为正数+5 设置了位位置 200。
相反，BinData 实例是零扩展的。例如，给定以下文件:
db.collection.save({ x: BinData(0, "ww=="), binaryValueofA: "11000011" })
E:
db.collection.save({ _id: 1, a: 54, binaryValueofA: "00110110" })
db.collection.save({ _id: 2, a: 20, binaryValueofA: "00010100" })
db.collection.save({ _id: 3, a: 20.0, binaryValueofA: "00010100" })
db.collection.save({ _id: 4, a: BinData(0, "Zg=="), binaryValueofA: "01100110" })
一些位置数组
下面的查询使用$bitsAnySet 操作符来测试字段 a 是否具有位位置 1 或位位置 5 集，其中最不重要的位是位置 0。
R:
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
db.collection.find({ a: { $bitsAnySet: 35 } })
R:
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }
BinData 位掩码
下面的查询使用$bitsAnySet 操作符来测试字段 a 在位置 4 和 5 处是否有任何位集(BinData 的二进制表示形式(0，“MC==”)是 00110000)。’
db.collection.find({ a: { $bitsAnySet: BinData(0, "MC==") } })
R:
{ "_id" : 1, "a" : 54, "binaryValueofA" : "00110110" }
{ "_id" : 2, "a" : 20, "binaryValueofA" : "00010100" }
{ "_id" : 3, "a" : 20.0, "binaryValueofA" : "00010100" }
{ "_id" : 4, "a" : BinData(0,"Zg=="), "binaryValueofA" : "01100110" }

$comment

$comment 查询操作符将注释关联到任何使用查询谓词的表达式。

由于注释传播到概要文件日志，添加注释可以使概要文件数据更容易解释和跟踪。
$comment 操作符有以下形式:
E:附上评论来寻找
下面的例子将$comment 添加到 find()操作中:

```
db.records.find(
    {
      x: { $mod: [ 2, 0 ] },
      $comment: "Find even values."
    }
)
```

将注释附加到聚合表达式

您可以将$comment 与任何带有查询谓词的表达式一起使用。

下面的例子使用$match 阶段的$comment 操作符来阐明操作:

```
db.records.aggregate( [
    { $match: { x: { $gt: 0 }, $comment: "Don't allow negative inputs." } },
    { $group : { _id: { $mod: [ "$x", 2 ] }, total: { $sum: "$x" } } }
] )
```

投影操作

$ (projection)

位置$操作符将查询结果中的<array>的内容限制为只包含匹配查询文档的第一个元素。若要指定要更新的数组元素，请参阅位置$运算符获取更新。

当在选定的文档中只需要一个特定数组元素时，在 find()方法或 findOne()方法的投影文档中使用$。

请参阅聚合运算符$filter，以返回只包含与指定条件匹配的元素的数组。

使用注意事项

$操作符和$ elements match 操作符都根据条件从数组中投射第一个匹配的元素。

$运算符根据查询语句中的某些条件从集合中的每个文档投射第一个匹配的数组元素。

$ element match 投影操作符接受一个显式的条件参数。这允许您基于查询中没有的条件进行项目，或者如果您需要基于数组嵌入文档中的多个字段进行项目。有关示例，请参见数组字段限制。

视图上的 db.collection.find()操作不支持$ projection 操作符。

E:collection student

```
{ "_id" : 1, "semester" : 1, "grades" : [ 70, 87, 90 ] }
{ "_id" : 2, "semester" : 1, "grades" : [ 90, 88, 92 ] }
{ "_id" : 3, "semester" : 1, "grades" : [ 85, 100, 90 ] }
{ "_id" : 4, "semester" : 2, "grades" : [ 79, 85, 80 ] }
{ "_id" : 5, "semester" : 2, "grades" : [ 88, 88, 92 ] }
{ "_id" : 6, "semester" : 2, "grades" : [ 95, 90, 96 ] }
```

在下面的查询中，投影{"grade "。$": 1}只返回 grade 字段中大于或等于 85 的第一个元素。

```
db.students.find( { semester: 1, grades: { $gte: 85 } },
                          { "grades.$": 1 } )
```

R:

```
{ "_id" : 1, "grades" : [ 87 ] }
{ "_id" : 2, "grades" : [ 90 ] }
```

{ "_id" : 3, "grades" : [85] }
虽然数组字段 grade 可能包含多个大于或等于 85 的元素，但是$ projection 操作符只返回数组中第一个匹配的元素。

项目文档数组
一个学生集合包含以下文档，其中 grade 字段是一个文档数组;每个文档包含三个字段名等级、平均值和 std:

```
{ "_id" : 7, semester: 3, "grades" : [ { grade: 80, mean: 75, std: 8 },
                                       { grade: 85, mean: 90, std: 5 },
                                       { grade: 90, mean: 85, std: 3 } ] }

{ "_id" : 8, semester: 3, "grades" : [ { grade: 92, mean: 88, std: 8 },
                                       { grade: 78, mean: 90, std: 5 },
                                       { grade: 88, mean: 85, std: 3 } ] }
```

在下面的查询中，投影{"grade "。$": 1}只返回职系字段中平均值大于 70 的第一个元素:

```
db.students.find(
    { "grades.mean": { $gt: 70 } },
    { "grades.$": 1 }
)
```

R:

```
{ "_id" : 7, "grades" : [   {   "grade" : 80,   "mean" : 75,   "std" : 8 } ] }
{ "_id" : 8, "grades" : [   {   "grade" : 92,   "mean" : 88,   "std" : 8 } ] }
```

$elemMatch

$elemMatch 操作符将查询结果中的<array>字段的内容限制为只包含匹配$elemMatch 条件的第一个元素。

使用注意事项
$操作符和$ elements match 操作符都根据条件从数组中投射第一个匹配的元素。
$运算符根据查询语句中的某些条件从集合中的每个文档投射第一个匹配的数组元素。

$ element match 投影操作符接受一个显式的条件参数。这允许您基于查询中没有的条件进行项目，或者如果您需要基于数组嵌入文档中的多个字段进行项目。有关示例，请参见数组字段限制。
视图上的 db.collection.find()操作不支持$ element match 投影操作符。

不能在$elemMatch 中指定$text 查询表达式。
$elemMatch 投影操作符上的示例假设一个集合学校具有以下文档:

```
{
 _id: 1,
 zipcode: "63109",
 students: [
                { name: "john", school: 102, age: 10 },
```

```
                    { name: "jess", school: 102, age: 11 },
                    { name: "jeff", school: 108, age: 15 }
                ]
}
{
  _id: 2,
  zipcode: "63110",
  students: [
                    { name: "ajax", school: 100, age: 7 },
                    { name: "achilles", school: 100, age: 8 },
                ]
}
{
  _id: 3,
  zipcode: "63109",
  students: [
                    { name: "ajax", school: 100, age: 7 },
                    { name: "achilles", school: 100, age: 8 },
                ]
}
{
  _id: 4,
  zipcode: "63109",
  students: [
                    { name: "barney", school: 102, age: 7 },
                    { name: "ruth", school: 102, age: 16 },
                ]
}
```

Zipcode 搜索

下面的 find()操作查询 zipcode 字段值为 63109 的所有文档。$elemMatch 投影只返回学生数组中第一个匹配的元素，其中 school 字段的值为 102：

```
db.schools.find( { zipcode: "63109" },
                    { students: { $elemMatch: { school: 102 } } } )
```

该操作返回以下 zipcode 等于 63109 的文档, 并使用$elemMatch 对 student 数组进行投影:

```
{ "_id" : 1, "students" : [ { "name" : "john", "school" : 102, "age" : 10 } ] }
{ "_id" : 3 }
{ "_id" : 4, "students" : [ { "name" : "barney", "school" : 102, "age" : 7 } ] }
```

对于_id 等于 1 的文档, student 数组包含多个元素, school 字段等于 102。但是, $elemMatch 投影只返回数组中第一个匹配的元素。

_id = 3 的文档在结果中不包含 students 字段, 因为它的 students 数组中没有匹配$ element match 条件的元素。

$ element 匹配多个字段

$elemMatch 投影可以指定多个字段的条件:

下面的 find()操作查询 zipcode 字段值为 63109 的所有文档。投影包括 student 数组的第一个匹配元素，其中 school 字段的值为 102,age 字段大于 10:

```
db.schools.find( { zipcode: "63109" },
                 { students: { $elemMatch: { school: 102, age: { $gt: 10} } } } )
```

R:

```
{ "_id" : 1, "students" : [ { "name" : "jess", "school" : 102, "age" : 11 } ] }
{ "_id" : 3 }
{ "_id" : 4, "students" : [ { "name" : "ruth", "school" : 102, "age" : 16 } ] }
```

_id 等于 3 的文档不包含 student 字段，因为没有数组元素匹配$elemMatch 标准。

$meta

$meta 投影操作符为每个匹配的文档返回元数据(例如。“textScore”)与查询关联。

$meta 表达式有以下语法:

```
{ $meta: <metaDataKeyword> }
```

E:projection

```
db.collection.find(
    <query>,
    { score: { $meta: "textScore" } }
)
```

Sort:$meta 表达式可以是 sort()表达式的一部分，如:

```
db.collection.find(
    <query>,
    { score: { $meta: "textScore" } }
).sort( { score: { $meta: "textScore" } } )
```

要在 sort()表达式中包含$meta 表达式，必须在投影文档中出现相同的$meta 表达式，包括 <projectedFieldName>。指定的元数据确定排序顺序。例如，“textScore”元数据按降序排序。

有关其他示例，请参见带有附加查询和排序表达式的文本搜索。

$slice

$slice 操作符控制查询返回的数组项数。有关在使用$push 更新期间限制数组大小的信息，请参见$slice 修饰符。

视图上的 db.collection.find()操作不支持$slice 投影操作符。

考虑以下原型查询:

```
db.collection。查找({field: value},  {array: {$slice: count}});
```

此操作选择由一个名为 field 的字段标识的文档集合，该字段保存值，并从存储在数组字段中的数组中返回由 count 值指定的元素数量。如果 count 的值大于数组中的元素数，则查询返回数组的所有元素。

$slice 接受多种格式的参数，包括负值和数组。考虑以下例子:

```
db.posts.find( {}, { comments: { $slice: 5 } } )
```

这里，$slice 在 comments 字段中选择数组中的前五项。
db.posts.find({}, { comments: { $slice: -5 } })
此操作返回数组中的最后五个项。

下面的示例指定一个数组作为$slice 的参数。数组采用[skip, limit]的形式，其中第一个值表示要跳过的数组中的项数，第二个值表示要返回的项数。
db.posts.find({}, { comments: { $slice: [20, 10] } })
在这里，查询将跳过数组的前 20 项后，只返回 10 项。
db.posts.find({}, { comments: { $slice: [-20, 10] } })

Update 操作

FieldUpdate 操作

$currentDate

$currentDate 操作符将字段的值设置为当前日期，即日期或时间戳。默认类型是 Date。
版本 3.0 中的变化:MongoDB 不再将时间戳和日期数据类型作为同等的比较/排序用途。有关详细信息，请参见日期和时间戳比较顺序。
$currentDate 操作符有以下形式:
{ $currentDate: { <field1>: <typeSpecification1>, ... } }
E:User collection
{ _id: 1, status: "a", lastModified: ISODate("2013-10-02T01:11:18.965Z") }
下面的操作将 lastModified 字段更新为当前日期，即“cancel”。将“date”字段更新为当前时间戳，并将状态字段更新为“D”和“cancel”。“用户请求”的原因。

```
db.users.update(
    { _id: 1 },
    {
      $currentDate: {
          lastModified: true,
          "cancellation.date": { $type: "timestamp" }
      },
      $set: {
          status: "D",
          "cancellation.reason": "user request"
      }
    }
)
```

更新后的文件将类似于:

```
{
    "_id" : 1,
```

```
    "status" : "D",
    "lastModified" : ISODate("2014-09-17T23:25:56.314Z"),
    "cancellation" : {
        "date" : Timestamp(1410996356, 1),
        "reason" : "user request"
    }
}
```

$inc

$inc 运算符将字段递增指定的值，其形式如下:
{$. n:行情):{< field1 >: < amount1 >、< field2 >: < amount2 >,…}}
要在嵌入式文档或数组中指定<field>，请使用点符号。
$inc 操作符接受正值和负值。
如果该字段不存在，$inc 将创建该字段并将该字段设置为指定的值。
在一个值为空的字段上使用$inc 操作符将生成一个错误。
$inc 是单个文档中的原子操作。
E:products collections

```
{
  _id: 1,
  sku: "abc123",
  quantity: 10,
  metrics: {
    orders: 2,
    ratings: 3.5
  }
}
```

下面的 update()操作使用$inc 操作符将 quantity 字段减少 2(即增加-2)并增加“metrics.orders“字段由 1:

```
db.products.update(
    { sku: "abc123" },
    { $inc: { quantity: -2, "metrics.orders": 1 } }
)
```

R:

```
{
    "_id" : 1,
    "sku" : "abc123",
    "quantity" : 8,
    "metrics" : {
        "orders" : 3,
        "ratings" : 3.5
    }
}
```

$min

使用$min 比较数字
在收集分数中考虑以下文件:
{ _id: 1, highScore: 800, lowScore: 200 }
文档的最低值当前为 200。下面的操作使用$min 将 200 与指定值 150 进行比较，并将 lowScore 的值更新为 150，因为 150 小于 200:
db.scores.update({ _id: 1 }, { $min: { lowScore: 150 } })
分数收集现在包含以下修改后的文件:
{ _id: 1, highScore: 800, lowScore: 150 }
下一个操作没有效果，因为字段 lowScore 的当前值 i。e150，小于 250:
db.scores.update({ _id: 1 }, { $min: { lowScore: 250 } })
R:
{ _id: 1, highScore: 800, lowScore: 150 }
使用$min 比较日期
在 collection 标签中考虑以下文档:

```
{
   _id: 1,
   desc: "crafts",
   dateEntered: ISODate("2013-10-01T05:00:00Z"),
   dateExpired: ISODate("2013-10-01T16:38:16Z")
}
```

下面的操作将 dateenter 字段的当前值，即 ISODate(“2013-10-01T05:00:00Z”)与指定的 date new date(“2013-09-25”)进行比较，以确定是否更新字段:

```
db.tags.update(
    { _id: 1 },
    { $min: { dateEntered: new Date("2013-09-25") } }
)
```

操作更新 dateenter 字段:

```
{
   _id: 1,
   desc: "crafts",
   dateEntered: ISODate("2013-09-25T00:00:00Z"),
   dateExpired: ISODate("2013-10-01T16:38:16Z")
}
```

$max

如果指定值大于字段的当前值，则$max 操作符将该字段的值更新为指定值。$max 操作符可以使用 BSON 比较顺序比较不同类型的值。

$max 运算符表达式的形式如下:
{ $max: { <field1>: <value1>, ... } }
例子
使用$max 比较数字
在收集分数中考虑以下文件:
{ _id: 1, highScore: 800, lowScore: 200 }
文档的最高分当前为 800。下面的操作使用$max 比较 800 和指定值 950，并将 highScore 的值更新为 950，因为 950 大于 800:
db.scores.update({ _id: 1 }, { $max: { highScore: 950 } })
{ _id: 1, highScore: 950, lowScore: 200 }
下一个操作没有效果，因为字段 highScore 的当前值 950 大于 870:
db.scores.update({ _id: 1 }, { $max: { highScore: 870 } })
本文件在分数收集中保持不变:
{ _id: 1, highScore: 950, lowScore: 200 }
使用$max 比较日期
在 collection 标签中考虑以下文档:

```
{
   _id: 1,
   desc: "crafts",
   dateEntered: ISODate("2013-10-01T05:00:00Z"),
   dateExpired: ISODate("2013-10-01T16:38:16.163Z")
}
```

下面的操作将 dateExpired 字段的当前值，即 ISODate(“2013-10-01T16:38:16.163Z”)与指定的 date new date(“2013-09-30”)进行比较，以确定是否更新字段:

```
db.tags.update(
    { _id: 1 },
    { $max: { dateExpired: new Date("2013-09-30") } }
)
```

操作不更新 dateExpired 字段:

```
{
    _id: 1,
    desc: "decorative arts",
    dateEntered: ISODate("2013-10-01T05:00:00Z"),
    dateExpired: ISODate("2013-10-01T16:38:16.163Z")

}
```

$mul

将字段的值乘以一个数字。要指定$mul 表达式，请使用以下原型:
{ $mul: { <field1>: <number1>, ... } }
行为
失踪的领域

如果文档中不存在字段，$mul 将创建该字段，并将值设置为零，其数值类型与乘数相同。
原子
$mul 是单个文档中的原子操作。
混合类型
与混合数值类型(32 位整数、64 位整数、浮点数)的值相乘可能导致数值类型的转换。若要与混合数值类型的值相乘，请应用以下类型转换规则:
E:
乘以字段的值
考虑一个具有以下文档的集合产品:
{ "_id" : 1, "item" : "ABC", "price" : NumberDecimal("10.99"), "qty" : 25 }
update()操作更新文档，使用$mul 操作符将价格乘以 1.25,qty 字段乘以 2:

```
db.products.update(
    { _id: 1 },
    { $mul: { price: NumberDecimal("1.25"), qty: 2 } }
)
```

操作结果如下文所示，其中 price 的新值为原价 10.99 乘以 1.25,qty 的新值为原价 25 乘以 2:
R:
{ "_id" : 1, "item" : "ABC", "price" : NumberDecimal("13.7375"), "qty" : 50 }
将$mul 操作符应用于不存在的字段
考虑一个具有以下文档的集合产品:
update()操作更新文档，将$mul 操作符应用于文档中不存在的字段价格:

```
db.products.update(
    { _id: 2 },
    { $mul: { price: NumberLong(100) } }
)
```

操作结果如下所示，price 字段设置为数值类型 NumberLong 的值 0，与乘数类型相同:
{ "_id" : 2, "item" : "Unknown", "price" : NumberLong(0) }
多重混合数值类型
考虑一个具有以下文档的集合产品:
E:
{ _id: 3, item: "XYZ", price: NumberLong(10) }
update()操作使用$mul 操作符将 price 字段 NumberLong(10)中的值乘以 NumberInt(5):

```
db.products.update(
    { _id: 3 },
    { $mul: { price: NumberInt(5) } }
)
```

R:
{ "_id" : 3, "item" : "XYZ", "price" : NumberLong(50) }

$rename

$rename 操作符更新字段名，并具有以下表单:

{$重命名:{< field1 >: < newName1 >、< field2 >: < newName2 >,…}}
新字段名必须与现有字段名不同。要在嵌入式文档中指定<field>，请使用点符号。
考虑下面的例子:
db.students.update({ _id: 1 }, { $rename: { 'nickname': 'alias', 'cell': 'mobile' } })
此操作将字段昵称重命名为 alias，并将字段单元格重命名为 mobile。
$rename 操作符逻辑上同时执行旧名称和新名称的$unset，然后使用新名称执行$set 操作。因此，该操作不得保留文档中字段的顺序;也就是说，重命名的字段可以在文档中移动。
如果文档已经有一个具有<newName>的字段，则$rename 操作符删除该字段，并将指定的<field>重命名为<newName>。

如果要重命名的字段在文档中不存在，$rename 什么也不做(即没有操作)。
对于嵌入文档中的字段，$rename 操作符可以重命名这些字段，也可以将这些字段移进或移出嵌入文档。如果这些字段位于数组元素中，则$rename 将不起作用。
E：如果 nmae 字段出现拼写错误，即应该是 name，则集合 student 包含以下文档:

```
{
   "_id": 1,
   "alias": [ "The American Cincinnatus", "The American Fabius" ],
   "mobile": "555-555-5555",
   "nmae": { "first" : "george", "last" : "washington" }
}

{
   "_id": 2,
   "alias": [ "My dearest friend" ],
   "mobile": "222-222-2222",
   "nmae": { "first" : "abigail", "last" : "adams" }
}

{
   "_id": 3,
   "alias": [ "Amazing grace" ],
   "mobile": "111-111-1111",
   "nmae": { "first" : "grace", "last" : "hopper" }
}
```

若要重命名字段，请使用字段的当前名称和新名称调用$rename 操作符:
db.students.updateMany({}, { $rename: { "nmae": "name" } })
此操作将字段 nmae 重命名为集合中所有文档的名称:

```
{
   "_id": 1,
   "alias": [ "The American Cincinnatus", "The American Fabius" ],
   "mobile": "555-555-5555",
   "name": { "first" : "george", "last" : "washington" }
}
```

```
{
    "_id" : 2,
    "alias" : [ "My dearest friend" ],
    "mobile" : "222-222-2222",
    "name" : { "first" : "abigail", "last" : "adams" }
}

{ "_id" : 3,
  "alias" : [ "Amazing grace" ],
  "mobile" : "111-111-1111",
  "name" : { "first" : "grace", "last" : "hopper" } }
```

在嵌入的文档实施例中重命名字段

要在嵌入式文档中重命名字段，请使用点符号调用$rename 操作符来引用该字段。如果字段要保留在相同的嵌入文档中，也要在新名称中使用点符号，如下所示:

```
db.students.update( { _id: 1 }, { $rename: { "name.first": "name.fname" } } )
```

这个操作首先将嵌入的字段重命名为 fname:

重命名不存在的字段

当重命名一个字段，而现有字段名指的是一个不存在的字段时，$rename 操作符什么也不做，如下所示:

重命名不存在的字段

当重命名一个字段，而现有字段名指的是一个不存在的字段时，$rename 操作符什么也不做，如下所示:

```
db.students.update( { _id: 1 }, { $rename: { 'wife': 'spouse' } } )
```

$set

$set 操作符用指定的值替换字段的值。

$set 运算符表达式的形式如下:

E:products collections

```
{
  _id: 100,
  sku: "abc123",
  quantity: 250,
  instock: true,
  reorder: false,
  details: { model: "14Q2", make: "xyz" },
  tags: [ "apparel", "clothing" ],
  ratings: [ { by: "ijk", rating: 4 } ]
}
```

set 顶级字段

对于匹配条件_id 等于 100 的文档，下面的操作使用$set 操作符更新 quantity 字段、details 字段和 tags 字段的值。

```
db.products.update(
    { _id: 100 },
```

```
    { $set:
        {
          quantity: 500,
          details: { model: "14Q3", make: "xyz" },
          tags: [ "coats", "outerwear", "clothing" ]
        }
    }
)
```
操作将值:quantity 替换为 500;details 字段指向新的嵌入文档，tags 字段指向新的数组。

在嵌入式文档中设置字段
要在嵌入式文档或数组中指定<field>，请使用点符号。
对于匹配条件_id = 100 的文档，以下操作更新 details 文档中的 make 字段:\
```
db.products.update(
    { _id: 100 },
    { $set: { "details.make": "zzz" } }
)
```
数组中的元素集
要在嵌入式文档或数组中指定<field>，请使用点符号。
对于匹配条件_id 等于 100 的文档，下面的操作将更新 tags 字段中的第二个元素(数组索引为 1)的值，以及评级数组的第一个元素(数组索引为 0)中的评级字段。

$setOnInsert

如果 upsert: true 的 update 操作导致插入文档，那么$setOnInsert 将指定的值分配给文档中的字段。如果更新操作没有导致插入，则$setOnInsert 不执行任何操作。

您可以为 db.collection.update()或 db.collection.findAndModify()方法指定 upsert 选项。
```
db.collection.update(
    <query>,
    { $setOnInsert: { <field1>: <value1>, ... } },
    { upsert: true }
)
```
E:名为 products 的集合不包含任何文档。
然后，使用 upsert 执行以下 db.collection.update(): true 插入一个新文档。
```
db.products.update(
   { _id: 1 },
   {
       $set: { item: "apple" },
       $setOnInsert: { defaultQty: 100 }
   },
   { upsert: true }
)
```
MongoDB 从<query>条件创建一个_id 等于 1 的新文档，然后将$set 和$setOnInsert 操作

应用到这个文档。
产品集合包含新插入的文档:
{ "_id" : 1, "item" : "apple", "defaultQty" : 100 }
如果 upsert: true 的 db.collection.update()找到了匹配的文档，然后 MongoDB 执行更新，应用$set 操作，但忽略$setOnInsert 操作。

$unset

$unset 操作符删除特定字段。考虑以下语法:
{ $unset: { <field1>: "", ... } }
$unset 表达式(即)不影响操作。
要在嵌入式文档或数组中指定<field>，请使用点符号。
如果字段不存在，那么$unset 什么也不做(即没有操作)。

当与$一起用于匹配数组元素时，$unset 使用 null 替换匹配的元素，而不是从数组中删除匹配的元素。此行为保持数组大小和元素位置一致。

例子
下面的 update()操作使用$unset 操作符从 products 集合的第一个文档中删除字段 quantity 和 instock，其中字段 sku 的值为 unknown。

```
db.products.update(
    { sku: "unknown" },
    { $unset: { quantity: "", instock: "" } }
)
```

Array Update Opreators

更新操作
名称描述
$ 充当占位符，更新匹配查询条件的第一个元素。
$[ ]充当占位符，更新数组中与查询条件匹配的文档的所有元素。
$[<identifier>] 充当占位符，为匹配查询条件的文档更新所有匹配 arrayFilters 条件的元素。
$ addtoset 仅在数组中不存在元素时才向数组添加元素。
$pop 删除数组的第一项或最后一项。
$pull 删除与指定查询匹配的所有数组元素。
$push 将一个项添加到数组中。
$ pullall 从数组中删除所有匹配的值。
更新算子修饰符
名称描述
$each 修改$push 和$addToSet 操作符，为数组更新添加多个项。
$position 修改$push 操作符，以指定要添加元素的数组中的位置。
$slice 修改$push 操作符以限制更新数组的大小。
$sort 修改$push 操作符，以对数组中存储的文档进行重新排序。

$update

position $操作符标识要更新的数组中的元素，而不显式指定元素在数组中的位置。

消歧
要从读取操作投射或返回数组元素，请查看$ projection 操作符。
要更新数组中的所有元素，请查看 all position 操作符$[]。
若要更新与数组筛选条件或多个条件匹配的所有元素，请参见已筛选的位置操作符$[<identifier>]。
位置$运算符有以下形式:
当与更新操作一起使用时，例如 db.collection.update()和 db.collection.findAndModify(),

位置$操作符充当与查询文档匹配的第一个元素的占位符，以及
数组字段必须作为查询文档的一部分出现。
例如:
当与更新操作一起使用时，例如 db.collection.update()和 db.collection.findAndModify(),
位置$操作符充当与查询文档匹配的第一个元素的占位符，以及
数组字段必须作为查询文档的一部分出现。
例如:
```
db.collection.update(
    { <array>: value ... },
    { <update operator>: { "<array>.$" : value } }
)
```
行为
插入
不要在 upsert 操作中使用位置操作符$, 因为插入操作将在插入的文档中使用$作为字段名。
嵌套的数组
位置$操作符不能用于遍历多个数组的查询，例如遍历嵌套在其他数组中的数组的查询，因为$占位符的替换是单个值

附件
当与$unset 操作符一起使用时，位置$操作符不会从数组中删除匹配的元素，而是将其设置为 null。

否定
如果查询使用否定运算符(如$ne、$not 或$nin)匹配数组，则不能使用位置运算符从该数组更新值。

但是, 如果查询的否定部分位于$elemMatch 表达式中, 则可以使用位置操作符更新该字段。
例子
更新数组中的值
创建一个集合学生与以下文件:
```
db.students.insert([
    { "_id" : 1, "grades" : [ 85, 80, 80 ] },
```

```
    { "_id" : 2, "grades" : [ 88, 90, 92 ] },
    { "_id" : 3, "grades" : [ 85, 100, 90 ] }
])
```

若要更新 grade 数组中第一个值为 80 到 82 的元素，如果不知道该元素在数组中的位置，请使用 position $运算符:

```
db.students.updateOne(
    { _id: 1, grades: 80 },
    { $set: { "grades.$" : 82 } }
)
```

更新数组实施例中的文档

位置$操作符有助于更新包含嵌入式文档的数组。使用位置$操作符访问嵌入文档中的字段，并在$操作符上使用点符号。

```
db.collection.update(
    { <query selector> },
    { <update operator>: { "array.$.field" : value } }
)
```

考虑学生集合中的以下文档，其中 grade 元素值是一个嵌入文档数组:

```
{
   _id: 4,
   grades: [
        { grade: 80, mean: 75, std: 8 },
        { grade: 85, mean: 90, std: 5 },
        { grade: 85, mean: 85, std: 8 }
   ]
}
```

使用 position $运算符更新匹配等于 85 条件的第一个数组元素的 std 字段:

必须将数组字段包含在查询文档中。

```
db.students.updateOne(
    { _id: 4, "grades.grade": 85 },
    { $set: { "grades.$.std" : 6 } }
)
```

R:

```
{
    "_id" : 4,
    "grades" : [
        { "grade" : 80, "mean" : 75, "std" : 8 },
        { "grade" : 85, "mean" : 90, "std" : 6 },
        { "grade" : 85, "mean" : 85, "std" : 8 }
    ]
}
```

使用多个字段匹配更新嵌入式文档

$操作符可以更新第一个数组元素，该元素匹配使用$elemMatch()操作符指定的多个查询条件。

考虑学生集合中的以下文档，其 grade 字段值是一个嵌入文档数组:

```
{
   _id: 5,
   grades: [
        { grade: 80, mean: 75, std: 8 },
        { grade: 85, mean: 90, std: 5 },
        { grade: 90, mean: 85, std: 3 }
   ]
}
```

在下面的例子中，$操作符更新第一个嵌入文档中 std 字段的值，该文档的 grade 字段的值小于或等于 90，而平均值字段的值大于 80:

```
db.students.updateOne(
    {
       _id: 5,
       grades: { $elemMatch: { grade: { $lte: 90 }, mean: { $gt: 80 } } }
    },
    { $set: { "grades.$.std" : 6 } }
)
```

此操作更新符合条件的第一个嵌入文档，即数组中的第二个嵌入文档:

```
{
   _id: 5,
   grades: [
      { grade: 80, mean: 75, std: 8 },

      { grade: 85, mean: 90, std: 6 },

      { grade: 90, mean: 85, std: 3 }
   ]
}
```

$[]

all position 操作符$[]表示 update 操作符应该修改指定数组字段中的所有元素。

$[]运算符的形式如下:

```
{ <update operator>: { "<array>.$[]" : value } }
```

插入

如果 upsert 操作导致插入, 查询必须在数组字段上包含一个精确的相等匹配, 以便在 update 语句中使用$[]位置操作符。

例如，下面的 upsert 操作在 update 文档中使用$[]，它在数组字段上指定了一个确切的相等匹配条件:

E:student collection

```
{ "_id" : 1, "grades" : [ 85, 82, 80 ] }
{ "_id" : 2, "grades" : [ 88, 90, 92 ] }
{ "_id" : 3, "grades" : [ 85, 100, 90 ] }
```

若要对集合中的所有文档将 grade 数组中的所有元素增加 10, 请使用 all position $[]运算符:

```
db.students.update(
    { },
    { $inc: { "grades.$[]": 10 } },
    { multi: true }
)
```

all position $[]操作符充当数组字段中所有元素的占位符。

操作完成后，学生作品集包含以下文件:

```
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 102 ] }
{ "_id" : 3, "grades" : [ 95, 110, 100 ] }
```

更新数组中的所有文档

$[]位置操作符有助于更新包含嵌入式文档的数组。要访问嵌入文档中的字段，请使用$[]操作符上的点符号。

```
db.collection.update(
    { <query selector> },
    { <update operator>: { "array.$[].field" : value } }
)
```

结合$[<identifier>]更新嵌套数组

$[]位置操作符与过滤器$[<identifier>]位置操作符可以用于更新嵌套数组。

使用以下文件创建一个集合 students3:

```
db.students3.insert([
    { "_id" : 1,
        "grades" : [
          { type: "quiz", questions: [ 10, 8, 5 ] },
          { type: "quiz", questions: [ 8, 9, 6 ] },
          { type: "hw", questions: [ 5, 4, 3 ] },
          { type: "exam", questions: [ 25, 10, 23, 0 ] },
        ]
    }
])
```

更新嵌套等级中大于或等于 8 的所有值。问题数组，无论类型:

```
db.students3.update(
    {},
    { $inc: { "grades.$[].questions.$[score]": 2 } },
    { arrayFilters: [    { "score": { $gte: 8 } } ], multi: true}
)
```

$[<identifier>]

新版本 3.6。

经过筛选的位置操作符$[<identifier>]标识与更新操作的 arrayFilters 条件匹配的数组元素，

例如 db.collection.update()和 db.collection.findAndModify()。

$[<identifier>]操作符与 arrayFilters 选项一起使用，其形式如下:

```
{ <update operator>: { "<array>.$[<identifier>]" : value } },
{ arrayFilters: [ { <identifier>: <condition> } ] }
```

例子

更新所有匹配 arrayFilters 的数组元素

考虑收集学生的下列文件:

```
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 102 ] }
{ "_id" : 3, "grades" : [ 95, 110, 100 ] }
```

若要更新 grade 数组中所有大于或等于 100 的元素，请使用经过筛选的位置操作符 $[<identifier>]和 arrayFilters:

```
db.students.update(
    { },
    { $set: { "grades.$[element]" : 100 } },
    { multi: true,
      arrayFilters: [ { "element": { $gte: 100 } } ]
    }
)
```

位置$[<identifier>]操作符充当数组字段中与 arrayFilters 中指定的条件匹配的所有元素的占位符。

操作完成后，学生作品集包含以下文件:

```
{ "_id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 100 ] }
{ "_id" : 3, "grades" : [ 95, 100, 100 ] }
```

更新数组中所有匹配 arrayFilters 的文档

$[<identifier>]操作符便于更新包含嵌入式文档的数组。要访问嵌入文档中的字段，请使用 $[<identifier>]上的点符号。

```
db.collection.update(
    { <query selector> },
    { <update operator>: { "array.$[<identifier>].field" : value } },
    { arrayFilters: [ { <identifier>: <condition> } ] }
)
```

考虑一个收集了以下文件的学生 2:

```
{
    "_id" : 1,
    "grades" : [
        { "grade" : 80, "mean" : 75, "std" : 6 },
        { "grade" : 85, "mean" : 90, "std" : 4 },
        { "grade" : 85, "mean" : 85, "std" : 6 }
    ]
}
{
```

```
    "_id" : 2,
    "grades" : [
        { "grade" : 90, "mean" : 75, "std" : 6 },
        { "grade" : 87, "mean" : 90, "std" : 3 },
        { "grade" : 85, "mean" : 85, "std" : 4 }
    ]
}
```

若要修改等级数组中等级大于或等于 85 的所有元素的平均值字段的值，请使用位置 $[<identifier>]运算符和 arrayFilters:

```
db.students2.update(
    { },
    { $set: { "grades.$[elem].mean" : 100 } },
    {
      multi: true,
      arrayFilters: [ { "elem.grade": { $gte: 85 } } ]
    }
)
R:
{
    "_id" : 1,
    "grades" : [
        { "grade" : 80, "mean" : 75, "std" : 6 },
        { "grade" : 85, "mean" : 100, "std" : 4 },
        { "grade" : 85, "mean" : 100, "std" : 6 }
    ]
}
{
    "_id" : 2,
    "grades" : [
        { "grade" : 90, "mean" : 100, "std" : 6 },
        { "grade" : 87, "mean" : 100, "std" : 3 },
        { "grade" : 85, "mean" : 100, "std" : 4 }
    ]
}
```

$addToSet

$addToSet 操作符向数组添加一个值，除非该值已经存在，在这种情况下，$addToSet 不对该数组执行任何操作。

$addToSet 操作符有以下形式:

E:inventory collection

```
{ _id: 1, item: "polarizing_filter", tags: [ "electronics", "camera" ] }
```

添加到数组
以下操作将元素“accessories”添加到标签数组中，因为数组中不存在“accessories”:
```
db.inventory.update(
    { _id: 1 },
    { $addToSet: { tags: "accessories" } }
)
```
值已经存在
下面的$addToSet 操作没有效果，因为“camera”已经是 tags 数组的一个元素:
```
db.inventory.update(
    { _id: 1 },
    { $addToSet: { tags: "camera"   } }
)
```
E:inventory collection
```
{ _id: 2, item: "cable", tags: [ "electronics", "supplies" ] }
```
然后，下面的操作使用$addToSet 操作符和$each 修饰符向标签数组添加多个元素:
```
db.inventory.update(
    { _id: 2 },
    { $addToSet: { tags: { $each: [ "camera", "electronics", "accessories" ] } } }
 )
```
R:该操作只向标签阵列添加“相机”和“附件”，因为“电子”已经存在于该阵列中:
```
{
   _id: 2,
   item: "cable",
   tags: [ "electronics", "supplies", "camera", "accessories" ]
}
```

$pop

$pop 操作符删除数组的第一个或最后一个元素。传递$pop 值-1 以删除数组中的第一个元素，并传递 1 以删除数组中的最后一个元素。

$pop 操作符有以下表单:
```
{ $pop: { <field>: <-1 | 1>, ... } }
```
E:Students
```
{ _id: 1, scores: [ 8, 9, 10 ] }
```
下面的例子删除了 score 数组中的第一个元素(8):
```
db.students.update( { _id: 1 }, { $pop: { scores: -1 } } )
```
操作完成后，更新后的文档从 score 数组中删除第一项 8:
```
{ _id: 1, scores: [ 9, 10 ] }
```
删除数组实施中的最后一项
学生们在收集下列文件时:
```
{ _id: 1, scores: [ 9, 10 ] }
```
下面的例子通过在$pop 表达式中指定 1 来删除 scores 数组中的最后一个元素(10):

db.students.update({ _id: 1 }, { $pop: { scores: 1 } })
操作完成后，更新后的文档将最后一项 10 从其 score 数组中删除：
{ _id: 1, scores: [9] }

$pull

$pull 操作符从现有数组中删除一个或多个与指定条件匹配的值的所有实例。
$pull 操作符有以下表单：
{ $pull: { <field1>: <value|condition>, <field2>: <value|condition>, ... } }
E:store collection

```
{
    _id: 1,
    fruits: [ "apples", "pears", "oranges", "grapes", "bananas" ],
    vegetables: [ "carrots", "celery", "squash", "carrots" ]
}
{
    _id: 2,
    fruits: [ "plums", "kiwis", "oranges", "bananas", "apples" ],
    vegetables: [ "broccoli", "zucchini", "carrots", "onions" ]
}
```

以下操作更新集合中的所有文档，从水果数组中删除“苹果”和“橘子”，从蔬菜数组中删除“胡萝卜”：

```
db.stores.update(
     { },
     { $pull: { fruits: { $in: [ "apples", "oranges" ] }, vegetables: "carrots" } },
     { multi: true }
)
```

操作后，水果数组不再包含任何“苹果”或“橘子”值，蔬菜数组不再包含任何“胡萝卜”值：

```
{
  "_id" : 1,
  "fruits" : [ "pears", "grapes", "bananas" ],
  "vegetables" : [ "celery", "squash" ]
}
{
  "_id" : 2,
  "fruits" : [ "plums", "kiwis", "bananas" ],
  "vegetables" : [ "broccoli", "zucchini", "onions" ]
}
```

删除所有匹配指定$pull 条件的项
在 profiles 集合中给出以下文档：
{ _id: 1, votes: [3, 5, 6, 7, 7, 8] }
以下操作将从投票数组中删除大于或等于($gte) 6 的所有项目：
db.profiles.update({ _id: 1 }, { $pull: { votes: { $gte: 6 } } })
更新操作后，文档值仅小于 6：

从文档数组中删除项
调查资料包括以下文件:

```
{
    _id: 1,
    results: [
        { item: "A", score: 5 },
        { item: "B", score: 8, comment: "Strongly agree" }
    ]
}
{
    _id: 2,
    results: [
        { item: "C", score: 8, comment: "Strongly agree" },
        { item: "B", score: 4 }
    ]
}
```

下面的操作将从结果数组中删除所有同时包含 score 字段等于 8 和 item 字段等于“B”的元素,multi 可选的。如果设置为 true，则更新满足查询条件的多个文档。如果设置为 false，则更新一个文档。默认值为 false。:

```
db.survey.update(
   { },
   { $pull: { results: { score: 8 , item: "B" } } },
   { multi: true }
)
```

$pull 表达式将该条件应用于结果数组的每个元素，就像它是一个顶级文档一样。
操作之后，结果数组不包含同时包含 score 字段等于 8 和 item 字段等于“B”的文档。

```
{
    "_id" : 1,
    "results" : [ { "item" : "A", "score" : 5 } ]
}
{
   "_id" : 2,
   "results" : [
        { "item" : "C", "score" : 8, "comment" : "Strongly agree" },
        { "item" : "B", "score" : 4 }
    ]
}
```

因为$pull 操作符将查询应用于每个元素，就像它是一个顶级对象一样，所以表达式不需要使用$elemMatch 来指定 score 字段等于 8 和 item 字段等于“B”的条件。事实上，下面的操作不会从原始集合中提取任何元素。

```
db.survey.update(
   { },
   { $pull: { results: { $elemMatch: { score: 8 , item: "B" } } } },
   { multi: true }
```

)
但是，如果调查集合包含以下文档，其中 results 数组包含也包含数组的嵌入式文档:

```
{
    _id: 1,
    results: [
        { item: "A", score: 5, answers: [ { q: 1, a: 4 }, { q: 2, a: 6 } ] },
        { item: "B", score: 8, answers: [ { q: 1, a: 8 }, { q: 2, a: 9 } ] }
    ]
}
{
    _id: 2,
    results: [
        { item: "C", score: 8, answers: [ { q: 1, a: 8 }, { q: 2, a: 7 } ] },
        { item: "B", score: 4, answers: [ { q: 1, a: 0 }, { q: 2, a: 8 } ] }
    ]
}
```

然后，您可以使用$elemMatch 在 answers 数组的元素上指定多个条件:

```
db.survey.update(
   { },
   { $pull: { results: { answers: { $elemMatch: { q: 2, a: { $gte: 8 } } } } } },
   { multi: true }
)
```

从结果数组中删除含有 answers 字段的嵌入式文档，其中包含至少一个元素，q = 2 且大于或等于 8:

```
{
    "_id" : 1,
    "results" : [
        { "item" : "A", "score" : 5, "answers" : [ { "q" : 1, "a" : 4 }, { "q" : 2, "a" : 6 } ] }
    ]
}
{
    "_id" : 2,
    "results" : [
        { "item" : "C", "score" : 8, "answers" : [ { "q" : 1, "a" : 8 }, { "q" : 2, "a" : 7 } ] }
    ]
}
```

$push 操作符向数组追加指定的值。

$push 操作符有以下形式:

```
{ $push: { <field1>: <value1>, ... } }
```

如果要更新的文档中没有该字段，$push 将添加该值作为元素的数组字段。

如果字段不是数组，则操作将失败。

如果值是数组，则$push 将整个数组追加为单个元素。若要单独添加值的每个元素，请使用$each 修饰符和$push。例如，请参见向数组追加多个值。有关$push 可用的修饰符列表，请参见修饰符。
您可以使用$push 操作符和以下修饰符:

修饰符描述
$each
向数组字段追加多个值。
当与其他修饰符一起使用时，$each 修饰符不再需要放在第一位。
$slice 限制数组元素的数量。要求使用$each 修饰符。
$sort 对数组的元素排序。要求使用$each 修饰符。
$position 指定数组中插入新元素的位置。要求使用$each 修饰符。如果没有$position 修饰符，$push 将把元素追加到数组的末尾。

$push

例子
向数组追加一个值
下面的示例将 89 追加到 scores 数组:

```
db.students.update(
    { _id: 1 },
    { $push: { scores: 89 } }
)
```

向数组追加多个值
使用$push 和$each 修饰符向数组字段追加多个值。
下面的例子将[90,92,85]的每个元素附加到 name 字段等于 joe 的文档的 scores 数组中:

```
db.students.update(
    { name: "joe" },
    { $push: { scores: { $each: [ 90, 92, 85 ] } } }
)
```

使用带有多个修饰符的$push 操作符
A collection 学生有以下文件:

```
{
    "_id" : 5,
    "quizzes" : [
        { "wk": 1, "score" : 10 },
        { "wk": 2, "score" : 8 },
        { "wk": 3, "score" : 5 },
        { "wk": 4, "score" : 6 }
    ]
}
```

$push 操作的用途如下:
$each 修饰符向 quizzes 数组添加多个文档，
$sort 修饰符，按分数字段降序对修改后的测验数组的所有元素排序，以及

$slice 修饰符只保留 quizzes 数组的前三个排序元素。

```
db.students.update(
    { _id: 5 },
    {
      $push: {
        quizzes: {
            $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
            $sort: { score: -1 },
            $slice: 3
        }
      }
    }
)
```

操作的结果是只保留三个得分最高的测验:

```
{
   "_id" : 5,
   "quizzes" : [
       { "wk" : 1, "score" : 10 },
       { "wk" : 2, "score" : 8 },
       { "wk" : 5, "score" : 8 }
   ]
}
```

$pullAll

$pullAll 操作符从现有数组中删除指定值的所有实例。与通过指定查询删除元素的$pull 操作符不同，$pullAll 删除与列出的值匹配的元素。

$pullAll 操作符有以下形式:

{ $pullAll: { <field1>: [<value1>, <value2> ...], ... } }

如果要删除的<value>是文档或数组，那么$pullAll 只删除数组中与指定的<value>完全匹配的元素，包括 order。

E:survey collection

{ _id: 1, scores: [0, 2, 5, 5, 1, 0] }

下面的操作从 scores 数组中删除值 0 和 5 的所有实例:

```
db.survey.update( { _id: 1 }, { $pullAll: { scores: [ 0, 5 ] } } )
```

R:

```
{ "_id" : 1, "scores" : [ 2, 1 ] }
```

$each

$each 修饰符可用于$addToSet 操作符和$push 操作符。

与$addToSet 操作符一起使用，如果<field>中不存在值，则向数组<field>添加多个值。

{ $addToSet: { <field>: { $each: [<value1>, <value2> ...] } } }

与$push 操作符一起使用，可以向数组<field>追加多个值。$position
E:使用$each 和$push 操作符
下面的例子将[90,92,85]的每个元素附加到 name 字段等于 joe 的文档的 scores 数组中:

```
db.students.update(
    { name: "joe" },
    { $push: { scores: { $each: [ 90, 92, 85 ] } } }
)
```

使用$each 和$addToSet 操作符实施
收集清单有以下文件:

```
{ _id: 2, item: "cable", tags: [ "electronics", "supplies" ] }
```

然后，下面的操作使用$addToSet 操作符和$each 修饰符向标签数组添加多个元素:

```
db.inventory.update(
    { _id: 2 },
    { $addToSet: { tags: { $each: [ "camera", "electronics", "accessories" ] } } }
 )
```

该操作只向标签阵列添加“相机”和“附件”，因为“电子”已经存在于该阵列中:

```
{
  _id: 2,
  item: "cable",
  tags: [ "electronics", "supplies", "camera", "accessories" ]
}
```

$postion

$position 修饰符指定$push 操作符在数组中插入元素的位置。如果没有$position 修饰符，$push 操作符将把元素插入数组的末尾。有关更多信息，请参见$push 修饰符。
要使用$position 修饰符，它必须与$each 修饰符一起出现。

```
{
  $push: {
    <field>: {
        $each: [ <value1>, <value2>, ... ],
        $position: <num>
    }
  }
}
```

例子
在数组的开头添加元素
考虑一个包含以下文档的集合学生:

```
{ "_id" : 1, "scores" : [ 100 ] }
```

下面的操作更新 score 字段，将元素 50、60 和 70 添加到数组的开头:

```
db.students.update(
    { _id: 1 },
    {
      $push: {
```

```
            scores: {
                $each: [ 50, 60, 70 ],
                $position: 0
            }
        }
    }
)
```

R:

```
{ "_id" : 1, "scores" : [    50,    60,    70,    100 ] }
```

将元素添加到数组的中间

考虑一个包含以下文档的集合学生:

```
{ "_id" : 1, "scores" : [    50,    60,    70,    100 ] }
```

下面的操作更新 score 字段，添加数组索引为 2 的元素 20 和 30:

```
db.students.update(
    { _id: 1 },
    {
      $push: {
          scores: {
              $each: [ 20, 30 ],
              $position: 2
          }
      }
    }
)
```

R;

```
{ "_id" : 1, "scores" : [    50,    60,    20,    30,    70,    100 ] }
```

使用负索引向数组添加元素

版本 3.6 中的变化:$position 可以接受一个负数数组索引值来表示从末尾开始的位置，从数组的最后一个元素开始计数(但不包括)。例如，-1 表示数组中最后一个元素之前的位置。

考虑一个包含以下文档的集合学生:

```
{ "_id" : 1, "scores" : [    50,    60,    20,    30,    70,    100 ] }
```

下面的操作为$position 指定-2，使其在最后一个元素前两个位置添加 90，然后在最后一个元素前两个位置添加 80。

```
db.students.update(
    { _id: 1 },
    {
      $push: {
          scores: {
              $each: [ 90, 80 ],
              $position: -2
          }
      }
    }
)
```

R:
{ "_id" : 1, "scores" : [50, 60, 20, 30, 90, 80, 70, 100] }

$slice

$slice 修饰符在$push 操作期间限制数组元素的数量。若要从读取操作投射或返回指定数目的数组元素，请参阅$slice 投影操作符。
要使用$slice 修饰符，它必须与$each 修饰符一起出现。可以将空数组[]传递给$each 修饰符，这样只有$slice 修饰符才有效果。

```
{
   $push: {
      <field>: {
         $each: [ <value1>, <value2>, ... ],
         $slice: <num>
      }
   }
}
```

从数组的末尾切片
学生作品集包含以下文件:
{ "_id" : 1, "scores" : [40, 50, 60] }
下面的操作向 scores 数组添加 3 个新元素，然后使用$slice 修饰符将数组修剪到最后五个元素:

```
db.students.update(
    { _id: 1 },
    {
      $push: {
         scores: {
            $each: [ 80, 78, 86 ],
            $slice: -5
         }
      }
    }
)
```

R:
{ "_id" : 1, "scores" : [50, 60, 80, 78, 86] }
E:
{ "_id" : 2, "scores" : [89, 90] }
F:下面的操作向 scores 数组添加新元素，然后使用$slice 修饰符将数组修剪为前三个元素。

```
db.students.update(
    { _id: 2 },
    {
      $push: {
         scores: {
            $each: [ 100, 20 ],
```

```
            $slice: 3
          }
        }
    }
)
R:
{ "_id" : 2, "scores" : [  89,  90,  100 ] }
E:
{ "_id" : 3, "scores" : [  89,  70,  100,  20 ] }
F:
db.students.update(
  { _id: 3 },
  {
    $push: {
      scores: {
          $each: [ ],
          $slice: -3
      }
    }
  }
)
R:
{ "_id" : 3, "scores" : [  70,  100,  20 ] }
E:
{
    "_id" : 5,
    "quizzes" : [
        { "wk": 1, "score" : 10 },
        { "wk": 2, "score" : 8 },
        { "wk": 3, "score" : 5 },
        { "wk": 4, "score" : 6 }
    ]
}
```

$push 操作的用途如下:

$each 修饰符向 quizzes 数组添加多个文档,
$sort 修饰符，按分数字段降序对修改后的测验数组的所有元素排序，以及
$slice 修饰符只保留 quizzes 数组的前三个排序元素。

```
db.students.update(
    { _id: 5 },
    {
      $push: {
        quizzes: {
            $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
```

```
            $sort: { score: -1 },
            $slice: 3
        }
      }
    }
)
R:
{
   "_id" : 5,
   "quizzes" : [
       { "wk" : 1, "score" : 10 },
       { "wk" : 2, "score" : 8 },
       { "wk" : 5, "score" : 8 }
   ]
}
```

$sort

$sort 修饰符在$push 操作期间对数组的元素进行排序。

要使用$sort 修饰符，它必须与$each 修饰符一起出现。您可以将一个空数组[]传递给$each 修饰符，这样只有$sort 修饰符才有效果。

```
{
   $push: {
       <field>: {
          $each: [ <value1>, <value2>, ... ],
          $sort: <sort specification>
       }
   }
}
```

例子

按文档中的字段对文档数组排序

学生作品集包含以下文件:

```
{
   "_id": 1,
   "quizzes": [
      { "id" : 1, "score" : 6 },
      { "id" : 2, "score" : 9 }
   ]
}
```

以下更新将附加额外的文档到 quizzes 数组中，然后按升序分数字段对数组中的所有元素进行排序:

```
db.students.update(
    { _id: 1 },
```

```
    {
      $push: {
        quizzes: {
          $each: [ { id: 3, score: 8 }, { id: 4, score: 7 }, { id: 5, score: 6 } ],
          $sort: { score: 1 }
        }
      }
    }
)
```

排序文档直接引用文档中的字段，不引用包含数组字段的测验;例如{score: 1}而不是{"quizzes "。分数:1}

更新后，数组元素按升序排列。

```
{
  "_id" : 1,
  "quizzes" : [
      { "id" : 1, "score" : 6 },
      { "id" : 5, "score" : 6 },
      { "id" : 4, "score" : 7 },
      { "id" : 3, "score" : 8 },
      { "id" : 2, "score" : 9 }
  ]
}
```

对非文档的数组元素排序

学生作品集包含以下文件:

```
{ "_id" : 2, "tests" : [  89,  70,  89,  50 ] }
```

下面的操作向 scores 数组中添加两个元素并对元素进行排序:

```
db.students.update(
    { _id: 2 },
    { $push: { tests: { $each: [ 40, 60 ], $sort: 1 } } }
)
```

更新后的文档中，分数数组的元素按升序排列:

```
{ "_id" : 2, "tests" : [  40,  50,  60,  70,  89,  89 ] }
```

只使用排序更新数组

学生作品集包含以下文件:

```
{ "_id" : 3, "tests" : [  89,  70,  100,  20 ] }
```

要更新 test 字段以降序排列其元素，请指定{$sort: -1}，并为$each 修饰符指定一个空数组[]，如下所示:

```
db.students.update(
    { _id: 3 },
    { $push: { tests: { $each: [ ], $sort: -1 } } }
)
```

操作的结果是更新 score 字段，按降序排列其元素:

```
{ "_id" : 3, "tests" : [ 100,  89,  70,  20 ] }
```

使用$sort 和其他$push 修饰符
A collection 学生有以下文件:

```
{
    "_id" : 5,
    "quizzes" : [
        { "wk": 1, "score" : 10 },
        { "wk": 2, "score" : 8 },
        { "wk": 3, "score" : 5 },
        { "wk": 4, "score" : 6 }
    ]
}
```

$push 操作的用途如下:
$each 修饰符向 quizzes 数组添加多个文档,
$sort 修饰符，按分数字段降序对修改后的测验数组的所有元素排序，以及
$slice 修饰符只保留 quizzes 数组的前三个排序元素。

```
db.students.update(
    { _id: 5 },
    {
      $push: {
        quizzes: {
            $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
            $sort: { score: -1 },
            $slice: 3
        }
      }
    }
)
```

R:

```
{
   "_id" : 5,
   "quizzes" : [
       { "wk" : 1, "score" : 10 },
       { "wk" : 2, "score" : 8 },
       { "wk" : 5, "score" : 8 }
   ]
}
```

位更新算子

$bit

$bit 操作符执行字段的位更新。操作符支持位和、位或和位 xor(即独占或)操作。要指定$bit 运算符表达式，请使用以下原型:

只对整数字段(32 位整数或 64 位整数)使用此操作符。
要在嵌入式文档或数组中指定<field>，请使用点符号。
例子
位和 and
考虑一下插入到集合开关中的以下文档:
{ _id: 1, expdata: NumberInt(13) }
下面的 update() 操作将 expdata 字段更新为当前值 NumberInt(13)(即 1101) 和 NumberInt(10)(即 1010)之间按位和操作的结果:

```
db.switches.update(
    { _id: 1 },
    { $bit: { expdata: { and: NumberInt(10) } } }
)
```

按位运算得到整数 8(即 1000):
R:
1101
1010

1000
更新后的文档对 expdata 的值如下:

```
{ "_id" : 1, "expdata" : 8 }
```

mongo shell 将 NumberInt(8)显示为 8。
按位或 or
考虑一下插入到集合开关中的以下文档:
{ _id: 2, expdata: NumberLong(3) }
下面的 update() 操作将 expdata 字段更新为当前值 NumberLong(3)(即 0011) 和 NumberInt(5)(即 0101)之间按位或操作的结果:

```
db.switches.update(
    { _id: 2 },
    { $bit: { expdata: { or: NumberInt(5) } } }
)
```

按位或运算得到整数 7(即 0111):
0011
0101

0111
更新后的文档对 expdata 的值如下:

```
{ "_id" : 2, "expdata" : NumberLong(7) }
```

按位异或 xor
在集合开关中考虑以下文档:
{ _id: 3, expdata: NumberLong(1) }
下面的 update() 操作将 expdata 字段更新为当前值 NumberLong(1)(即 0001) 和 NumberInt(5)(即 0101)之间的位 xor 操作的结果:

```
db.switches.update(
    { _id: 3 },
```

{ $bit: { expdata: { xor: NumberInt(5) } } }
)
按位 xor 运算得到整数 4:
0001
0101

0100
更新后的文档对 expdata 的值如下:
{ "_id" : 3, "expdata" : NumberLong(4) }

查询修改器

除了 MongoDB 查询操作符之外，还有许多“元”操作符允许您修改查询的输出或行为。
在 mongo Shell 中，从 v3.2 开始就不支持
从 v3.2 开始，查询“meta”操作符在 mongo shell 中被禁用。在 mongo shell 中，使用游标方法。
驱动程序接口可能提供封装这些选项的游标方法。如果可能的话，使用这些方法;否则，您可以使用以下任何一种语法添加这些选项:
db.collection.find({ <query> })._addSpecial(<option>)
db.collection.find({ $query: { <query> }, <option> })
操作：
名称描述
$comment 向查询添加一条注释，以识别数据库分析器输出中的查询。
$explain 强制 MongoDB 报告查询执行计划。见说明()。
$hint 强制 MongoDB 使用特定的索引。看到提示()
$max 指定要在查询中使用的索引的独占上限。看到 max ()。
$ maxtimems 指定处理游标操作的累积时间限制(以毫秒为单位)。看到 maxTimeMS ()。
$min 指定要在查询中使用的索引的包含下限。看到 min ()。
$orderby 返回一个游标，其中包含根据排序规范排序的文档。看到()。
$query 包装一个查询文档。
$ returnkey 强制光标只返回索引中包含的字段。
$ showdiskloc 修改返回的文档，以包含对每个文档的磁盘位置的引用。
排序顺序
名称描述
$natural 是一种特殊的排序顺序，它使用磁盘上的文档顺序来排序文档。

$comment

$comment 元操作符允许在$query 可能出现的任何上下文中将注释附加到查询。
由于注释传播到概要文件日志，添加注释可以使概要文件数据更容易解释和跟踪。
使用$comment 的方法如下:
db.collection.find({ <query> })._addSpecial("$comment", <comment>)

db.collection.find({ <query> }).comment(<comment>)
db.collection.find({ $query: { <query> }, $comment: <comment> })
要将注释附加到其他上下文中的查询表达式，例如 db.collection.update()，请使用 $comment 查询操作符而不是元操作符。

$explain

$explain 操作符提供关于查询计划的信息。它返回一个文档，描述用于返回查询的过程和索引。这可能在试图优化查询时提供有用的见解。有关输出的详细信息，请参见 cursor.explain()。
你可以用以下任何一种形式指定$explain 运算符:
db.collection.find()._addSpecial("$explain", 1)
db.collection.find({ $query: {}, $explain: 1 })
在 mongo shell 中，还可以通过 explain()方法检索查询计划信息:
db.collection.find().explain()

$hint

$hint 操作符强制查询优化器使用特定索引来完成查询。通过索引名或文档指定索引。
使用$hint 测试查询性能和索引策略。mongo shell 为$hint 操作符提供了一个 helper 方法 hint()。
db.users.find().hint({ age: 1 })
此操作使用 age 字段上的索引返回集合中名为 users 的所有文档。
您还可以使用以下任何一种形式指定提示:
db.users.find()._addSpecial("$hint", { age : 1 })
db.users.find({ $query: {}, $hint: { age : 1 } })
当查询以下列形式指定$hint 时:
db.users。find({$query: {}, $hint: {age: 1}})
得创建索引。

```
db.supplies.find().hint({qty:1})

{ "_id" : 3, "item" : "pencil", "qty" : 50, "price" : 6 }

{ "_id" : 1, "item" : "binder", "qty" : 100, "price" : 12 }

{ "_id" : 4, "item" : "eraser", "qty" : 150, "price" : 3 }

{ "_id" : 2, "item" : "notebook", "qty" : 200, "price" : 8 }
```

然后，为了包含$explain 选项，您必须将$explain 选项添加到文档中，如下所示:
db.users.find({ $query: {}, $hint: { age : 1 }, $explain: 1 })

$max

指定$max 值来指定特定索引的独占上限，以便约束 find()的结果。$max 按顺序指定特定索引的所有键的上限。
mongo shell 提供了 max()包装器方法:
db.collection.find({ <query> }).max({ field1: <max value>, ... fieldN: <max valueN> })
你也可以用这两种表格中的任何一种指定$max:
db.collection.find({ <query> })._addSpecial("$max", { field1: <max value1>, ... fieldN: <max valueN> })
db.collection.find({ $query: { <query> }, $max: { field1: <max value1>, ... fieldN: <max valueN> } })
索引使用
从 MongoDB 4.2 开始，必须使用 hint()方法显式指定特定的索引，以运行 max()/ $max，但有以下例外:如果 find()查询是_id 字段{_id: <value>}上的一个等式条件，则不需要提示。
与索引选择的交互
因为 max()需要一个字段上的索引，并且强制查询使用这个索引，所以如果可能的话，您可能更喜欢使用$lt 操作符来执行查询。考虑下面的例子:
db.collection.find(　{ _id: { $in: [6, 7] } }).max({ age: 25 }).hint({ age: 1 })
查询使用 age 字段上的索引，即使_id 上的索引可能更好。
$max without $min
最小和最大操作符表示系统应该避免正常的查询规划。相反，它们构造一个索引扫描，其中索引边界由 min 和 max 中给出的值显式指定。
指定独占上界
考虑对一个名为 collection 的集合执行以下操作，该集合的索引为{age: 1}:
db.collection.find({ <query> }).max({ age: 100 }).hint({ age: 1 })
使用$min
单独使用$max 或与$min 结合使用，将相同索引的结果限制在特定范围内，如下例所示:
请注意
版本 4.0 中更改:$max 指定的界限必须大于$min 指定的界限。
db.collection.find().min({ age: 20 }).max({ age: 25 }).hint({ age: 1 })

$maxTimeMS

新版本 2.6:$maxTimeMS 操作符指定了处理游标操作的累积时间限制(以毫秒为单位)。MongoDB 在随后最早的中断点中断操作。
mongo shell 提供了 cursor.maxTimeMS()方法
db.collection.find().maxTimeMS(100)
你也可以用以下任何一种形式指定该选项:
db.collection.find({ $query: { }, $maxTimeMS: 100 })
db.collection.find({ })._addSpecial("$maxTimeMS", 100)

劈劈啪啪的操作返回一个类似于下面的错误信息:
error: { "$err" : "operation exceeded time limit", "code" : 50 }

$min

弃用自 V3.2
从 v3.2 开始, $min 操作符在 mongo shell 中被禁用。在 mongo shell 中, 使用 cursor.min()。
指定$min 值来指定特定索引的包含下界，以约束 find()的结果。$min 按顺序指定特定索引的所有键的下界。
mongo shell 提供 min()包装器方法:
db.collection.find({ <query> }).min({ field1: <min value>, ... fieldN: <min valueN>})
你也可以用这两种表格中的任何一种指定选项:
db.collection.find({ <query> })._addSpecial("$min", { field1: <min value1>, ... fieldN: <min valueN> })
db.collection.find({ $query: { <query> }, $min: { field1: <min value1>, ... fieldN: <min valueN> } })
索引使用
从 MongoDB 4.2 开始，必须使用 hint()方法显式指定特定的索引，以运行 min()/ $min，但有以下例外:如果 find()查询是_id 字段{_id: <value>}上的一个等式条件，则不需要提示。
db.collection.find({ _id: { $in: [6, 7] } }).min({ age: 25 }).hint({ age: 1 })
下面的示例使用 mongo shell 包装器。
指定包含下界
考虑对一个名为 collection 的集合执行以下操作，该集合的索引为{age: 1}:
db.collection.find().min({ age: 20 }).hint({ age: 1 })
该操作将查询限制在字段 age 至少为 20 的文档中, 并强制执行一个查询计划, 该计划将{age: 1}索引从 20 扫描到 MaxKey。
使用$max
您可以使用$min 和$max 来将相同索引的结果限制在特定范围内，如下面的示例所示:
db.collection.find().min({ age: 20 }).max({ age: 25 }).hint({ age: 1 })

$orderby

用自 V3.2
从 v3.2 开始，$orderby 操作符在 mongo shell 中被禁用。在 mongo shell 中，使用 cursor.sort()。
$orderby 操作符按升序或降序对查询结果进行排序。
mongo shell 提供了指针.sort()方法:
db.collection.find().sort({ age: -1 })
这些示例返回集合中名为集合的所有文档, 按年龄字段降序排序。指定$orderby 的值为-1(如上所述，例如-1)以降序排序，或为正值(例如 1)以升序排序。
db.collection.find()._addSpecial("$orderby", { age : -1 })
db.collection.find({ $query: {}, $orderby: { age : -1 } })

行为
sort 函数要求整个排序能够在 32 mb 内完成。当 sort 选项消耗超过 32 兆字节时, MongoDB 将返回一个错误。
为了避免这个错误, 创建一个索引来支持排序操作, 或者将$orderby 与 cursor.maxTimeMS() 和/或 cursor.limit()结合使用。limit()通过优化算法提高返回此查询所需的速度和内存量。指定的限制必须导致许多文档位于 32 兆字节限制之内。

$query

$query 操作符强制 MongoDB 将表达式解释为查询。
下面的 mongo 操作是等价的，并且只返回年龄字段为 25 的集合集合中的那些文档。
$query 用于处理包含字段名查询的文档，该字段名查询的值是嵌入的文档，如以下文档:
db.collection.find({ $query: { age : 25 } })
db.collection.find({ age : 25 })
$query 用于处理包含字段名查询的文档，该字段名查询的值是嵌入的文档，如以下文档:
{ _id: 1, age: 25, query: { a: 1 } }
以下不使用$query 操作符的 find 操作将不会返回任何结果:
db.documents.find({ query: { a: 1 } })
要取得文件，你需要使用以下查询:
db.documents.find({ "$query": { query: { a: 1 } } })
不要混合查询表单。如果使用$query 格式，则不要向 find()添加游标方法。要修改查询，可以使用元查询操作符，比如$explain。
因此，以下两个操作是等价的:
db.collection.find({ $query: { age : 25 }, $explain: true })
db.collection.find({ age : 25 }).explain()

$returnKey

只返回索引字段或查询结果的字段。如果$returnKey 被设置为 true，并且查询没有使用索引来执行读取操作，那么返回的文档将不包含任何字段。使用下列其中一种形式:
db.collection.find({ <query> })._addSpecial("$returnKey", true)
db.collection.find({ $query: { <query> }, $returnKey: true })

$showDiskLoc

$showDiskLoc 选项将字段$diskLoc 添加到返回的文档中。增加的$diskLoc 字段的值是一个包含磁盘位置信息的文档:
"$diskLoc": {
  "file": <int>,
  "offset": <int>

}
mongo shell 为$showDiskLoc 提供了指针.showDiskLoc()方法:
db.collection.find().showDiskLoc()
你也可以用以下任何一种形式指定$showDiskLoc 选项:
db.collection.find({ <query> })._addSpecial("$showDiskLoc" , true)
db.collection.find({ $query: { <query> }, $showDiskLoc: true })
有关使用方法，请参见在 cursor.hint()参考页面中查看 Force Collection 扫描示例。
不能在视图上指定$natural 排序。

$natural

聚合操作:

聚合管道 Stages（阶段）：

除了$out 和$geoNear 阶段外，其他阶段都可以在管道中出现多次。
请注意
有关特定运算符的详细信息，包括语法和示例，请单击特定运算符以转到其参考页面。

db.collection.aggregate([{ <stage> }, ...])

Stage:
阶段描述
$ addfields:　　向文档添加新字段。与$project 类似，$addFields 会重新定义流中的每个文档;具体地说，通过向输出文档添加新字段，这些输出文档包含来自输入文档的现有字段和新添加的字段。
$bucket:　　根据指定的表达式和 bucket 边界将传入的文档分类到称为 bucket 的组中。
$ bucketauto:　根据指定的表达式将传入的文档分类为特定数量的组(称为 bucket)。Bucket 边界将自动确定，以便将文档均匀地分布到指定数量的 Bucket 中。
$ collstats:　　返回关于集合或视图的统计信息。
$count:　　返回聚合管道此阶段的文档数量的计数。
$facet:　　在同一组输入文档的一个阶段内处理多个聚合管道。支持创建能够在单个阶段跨多个维度或方面描述数据的多面聚合。
$ geonear:　　根据与地理空间点的接近程度返回有序的文档流。为地理空间数据合并了$match、$sort 和$limit 功能。输出文档包含一个额外的距离字段，并且可以包含一个位置标识符字段。
$ graphlookup;　对集合执行递归搜索。向每个输出文档添加一个新的数组字段，该字段包含对该文档的递归搜索的遍历结果。
$group:　　按指定的标识符表达式对输入文档进行分组，如果指定，则将累加器表

达式应用于每个组。使用所有输入文档，并为每个不同的组输出一个文档。输出文档只包含标识符字段，如果指定，还包含累计字段。
$ indexstats: 返回关于集合中每个索引的使用情况的统计信息。
$limit: 将未修改的前 n 个文档传递给管道，其中 n 是指定的限制。对于每个输入文档，要么输出一个文档(对于前 n 个文档)，要么输出 0 个文档(在前 n 个文档之后)。
$ listsessions: 列出所有活动了足够长时间以传播到系统的会话。会话集合。
$lookup: 执行到同一数据库中另一个集合的左外连接，以便从“已连接”集合中筛选文档进行处理。
$match: 筛选文档流，只允许匹配的文档未经修改地传递到下一个管道阶段。$match 使用标准的 MongoDB 查询。对于每个输入文档，输出一个文档(匹配)或零文档(没有匹配)。
$out: 将聚合管道的结果文档写入集合。要使用$out 阶段，它必须是管道中的最后一个阶段。
$project: 修改流中的每个文档，例如添加新字段或删除现有字段。对于每个输入文档，输出一个文档。
$redact: 根据存储在文档本身中的信息限制每个文档的内容，从而重新构造流中的每个文档。合并了$project 和$match 的功能。可用于实现字段级编校。对于每个输入文档，输出一个或零个文档。
$ replaceroot: 使用指定的嵌入文档替换文档。该操作替换输入文档中的所有现有字段，包括_id 字段。指定嵌入到输入文档中的文档，将嵌入的文档提升到顶层。
$sample: 从其输入中随机选择指定数量的文档。
$skip: 跳过第一个 n 个文档，其中 n 是指定的跳过号，并将其余未修改的文档传递给管道。对于每个输入文档，输出零个文档(对于前 n 个文档)或一个文档(如果在前 n 个文档之后)。
$sort: 按指定的排序键重新排序文档流。只有顺序改变了;文件仍未修改。对于每个输入文档，输出一个文档。
$ sortbycount: 根据指定表达式的值对传入文档进行分组，然后计算每个不同组中的文档数量。
$unwind: 将数组字段从输入文档解构为每个元素的输出文档。每个输出文档都用一个元素值替换数组。对于每个输入文档，输出 n 个文档，其中 n 是数组元素的数量，对于空数组可以为零。
要在管道阶段使用聚合表达式运算符，请参阅聚合管道运算符。
db.aggregate()阶段
从 3.6 版开始，MongoDB 也提供了 db。聚合方法:

db.aggregate([{ <stage> }, ...])

以下阶段使用 db.aggregate()方法，而不是 db.collection.aggregate()方法。
阶段描述
$ currentop: 返回关于 MongoDB 部署的活动和/或休眠操作的信息。
$ listlocalsessions: 列出当前连接的 mongos 或 mongod 实例上最近使用的所有活动会话。这些会话可能还没有传播到系统。会话集合。
按字母顺序列出的阶段实施:
$ addfields: 向文档添加新字段。输出包含来自输入文档和新添加字段的所有现有字段

的文档。
$bucket: 根据指定的表达式和bucket边界将传入的文档分类到称为bucket的组中。
$ bucketauto: 根据指定的表达式将传入的文档分类为特定数量的组(称为 bucket)。Bucket 边界将自动确定，以便将文档均匀地分布到指定数量的 Bucket 中。
$ collstats: 返回关于集合或视图的统计信息。
$count: 返回聚合管道此阶段的文档数量的计数。
$ currentop: 返回关于 MongoDB 部署的活动和/或休眠操作的信息。要运行，请使用db.aggregate()方法。
$facet: 在同一组输入文档的一个阶段内处理多个聚合管道。支持创建能够在单个阶段跨多个维度或方面描述数据的多面聚合。
$ geonear: 根据与地理空间点的接近程度返回有序的文档流。为地理空间数据合并了$match、$sort 和$limit 功能。输出文档包含一个额外的距离字段，并且可以包含一个位置标识符字段。
$ graphlookup: 对集合执行递归搜索。向每个输出文档添加一个新的数组字段，该字段包含对该文档的递归搜索的遍历结果。
$group: 按指定的标识符表达式对输入文档进行分组，如果指定，则将累加器表达式应用于每个组。使用所有输入文档，并为每个不同的组输出一个文档。输出文档只包含标识符字段，如果指定，还包含累计字段。
$ indexstats: 返回关于集合中每个索引的使用情况的统计信息。
$limit: 将未修改的前 n 个文档传递给管道，其中 n 是指定的限制。对于每个输入文档，要么输出一个文档(对于前 n 个文档)，要么输出 0 个文档(在前 n 个文档之后)。
$ listlocalsessions: 列出当前连接的 mongos 或 mongod 实例上最近使用的所有活动会话。这些会话可能还没有传播到系统。会话集合。
$ listsessions: 列出所有活动了足够长时间以传播到系统的会话。会话集合。
$lookup: 执行到同一数据库中另一个集合的左外连接，以便从“已连接”集合中筛选文档进行处理。
$match: 筛选文档流，只允许匹配的文档未经修改地传递到下一个管道阶段。$match 使用标准的 MongoDB 查询。对于每个输入文档，输出一个文档(匹配)或零文档(没有匹配)。
$out: 将聚合管道的结果文档写入集合。要使用$out 阶段，它必须是管道中的最后一个阶段。
$project: 修改流中的每个文档，例如添加新字段或删除现有字段。对于每个输入文档，输出一个文档。
$redact: 根据存储在文档本身中的信息限制每个文档的内容，从而重新构造流中的每个文档。合并了$project 和$match 的功能。可用于实现字段级编校。对于每个输入文档，输出一个或零个文档。
$ replaceroot: 使用指定的嵌入文档替换文档。该操作替换输入文档中的所有现有字段，包括_id 字段。指定嵌入到输入文档中的文档，将嵌入的文档提升到顶层。
$sample 从其输入中随机选择指定数量的文档。
$skip: 跳过第一个 n 个文档，其中 n 是指定的跳过号，并将其余未修改的文档传递给管道。对于每个输入文档，输出零个文档(对于前 n 个文档)或一个文档(如果在前 n 个文档之后)。
$sort: 按指定的排序键重新排序文档流。只有顺序改变了;文件仍未修改。对于每个输入文档，输出一个文档。
$ sortbycount: 根据指定表达式的值对传入文档进行分组，然后计算每个不同组中的文

档数量。
$unwind:　　将数组字段从输入文档解构为每个元素的输出文档。每个输出文档都用一个元素值替换数组。对于每个输入文档，输出 n 个文档，其中 n 是数组元素的数量，对于空数组可以为零。

$addFields:

$ addFields 将新字段附加到现有文档。可以在聚合操作中包含一个或多个$ addFields 阶段。
嵌入文档（包括数组中的文档）添加字段或字段，请使用点表示法。见例子。
使用$ addFields 将元素添加到现有数组字段，请使用$ concatArrays。见例子。
例子：
使用两个$addFields:
文档：

```
db.insertMany({
   _id: 1,
   student: "Maya",
   homework: [ 10, 5, 10 ],
   quiz: [ 10, 8 ],
   extraCredit: 0
},
{
   _id: 2,
   student: "Ryan",
   homework: [ 5, 6, 5 ],
   quiz: [ 8, 8 ],
   extraCredit: 8
})
```

以下操作使用两个$ addFields 阶段在输出文档中包含三个新字段：

```
db.scores.aggregate( [
    {
      $addFields: {
        totalHomework: { $sum: "$homework" } ,
        totalQuiz: { $sum: "$quiz" }
      }
    },
    {
      $addFields: { totalScore:
        { $add: [ "$totalHomework", "$totalQuiz", "$extraCredit" ] } }
    }
] )
```

查询结果如下(不会改变实际内容)：

```
{
   "_id" : 1,
   "student" : "Maya",
   "homework" : [ 10, 5, 10 ],
   "quiz" : [ 10, 8 ],
   "extraCredit" : 0,
   "totalHomework" : 25,
   "totalQuiz" : 18,
   "totalScore" : 43
}
{
   "_id" : 2,
   "student" : "Ryan",
   "homework" : [ 5, 6, 5 ],
   "quiz" : [ 8, 8 ],
   "extraCredit" : 8,
   "totalHomework" : 16,
   "totalQuiz" : 16,
   "totalScore" : 40
}
```

将字段添加到嵌入式文档

使用点表示法向嵌入文档添加新字段。名为 vehicles 的集合包含以下文档：

```
{ _id: 1, type: "car", specs: { doors: 4, wheels: 4 } }
{ _id: 2, type: "motorcycle", specs: { doors: 0, wheels: 2 } }
{ _id: 3, type: "jet ski" }
```

以下聚合操作将新字段 fuel_type 添加到嵌入的文档规范中。

```
db.vehicles.aggregate( [
          {
              $addFields: {
                  "specs.fuel_type": "unleaded"
              }
          }
    ] )
```

该操作返回以下结果：

```
{ _id: 1, type: "car",
    specs: { doors: 4, wheels: 4, fuel_type: "unleaded" } }
{ _id: 2, type: "motorcycle",
    specs: { doors: 0, wheels: 2, fuel_type: "unleaded" } }
{ _id: 3, type: "jet ski",
    specs: { fuel_type: "unleaded" } }
```

覆盖现有字段

在$ addFields 操作中指定现有字段名称会导致替换原始字段。

名为 animals 的集合包含以下文档：

```
{ _id: 1, dogs: 10, cats: 15 }
```

以下$ addFields 操作指定 cats 字段。

```
db.animals.aggregate( [
  {
    $addFields: { "cats": 20 }
  }
] )
```

返回以下文档：

```
{ _id: 1, dogs: 10, cats: 20 }
```

可以用另一个字段替换一个字段。在以下示例中，item 字段替换_id 字段。名为 fruit 的集合包含以下文档：

```
{ "_id" : 1, "item" : "tangerine", "type" : "citrus" }
{ "_id" : 2, "item" : "lemon", "type" : "citrus" }
{ "_id" : 3, "item" : "grapefruit", "type" : "citrus" }
```

以下聚合操作使用$ addFields 将每个文档的_id 字段替换为 item 字段的值，并使用静态值替换 item 字段。

```
db.fruit.aggregate( [
  {
    $addFields: {
      _id : "$item",
      item: "fruit"
    }
  }
] )
```

R:

```
{ "_id" : "tangerine", "item" : "fruit", "type" : "citrus" }
{ "_id" : "lemon", "item" : "fruit", "type" : "citrus" }
{ "_id" : "grapefruit", "item" : "fruit", "type" : "citrus" }
```

将元素添加到数组，使用以下内容创建样本分数集合：

```
db.scores.insertMany([
    { _id: 1, student: "Maya", homework: [ 10, 5, 10 ], quiz: [ 10, 8 ], extraCredit: 0 },
    { _id: 2, student: "Ryan", homework: [ 5, 6, 5 ], quiz: [ 8, 8 ], extraCredit: 8 }
])
```

可以将$ addFields 与$ concatArrays 表达式一起使用，以将元素添加到现有数组字段中。例如，以下操作使用$ addFields 将一个新的数组替换为 homework 字段，该数组的元素是与包含新分数的另一个数组连接的当前作业数组[7]。

```
db.scores.aggregate([
    { $match: { _id: 1 } },
    { $addFields: { homework: { $concatArrays: [ "$homework", [ 7 ] ] } } }
])
```

R:

```
{ "_id" : 1, "student" : "Maya", "homework" : [ 10, 5, 10, 7 ], "quiz" : [ 10, 8 ],
"extraCredit" : 0 }
```

$bucket：

版本 3.4 中的新功能。
根据指定的表达式和存储区边界，将传入的文档分组，称为存储桶。
每个存储桶在输出中表示为文档。每个存储桶的文档包含一个_id 字段，其值指定存储桶的包含下限，以及包含存储桶中文档数的计数字段。未指定输出时，默认情况下包含计数字段。
$ bucket 仅为包含至少一个输入文档的存储桶生成输出文档。

```
{
  $bucket: {
      groupBy: <表达式>,
      boundaries: [ <lowerbound1>, <lowerbound2>, ... ],
      default: <literal>,
      output: {
          <output1>: { <$accumulator expression> },
          ...
          <outputN>: { <$accumulator expression> }
      }
   }
}
```

根据月统计：

```
db.holidays.aggregate([{$project:{math:{$month:"$date"}}},{$bucket:{groupBy:"$math",boundaries:[1,2,3,4,5,6,7,8,9,10,11,12],default:"other",output:{"count":{$sum:1}}}}])
```

GROUPBY 表达式：
用于分组文档的表达式。要指定字段路径，请在字段名称前加上美元符号$，并将其括在引号中。
除非$ bucket 包含默认规范，否则每个输入文档必须将 groupBy 字段路径或表达式解析为属于边界指定的范围之一的值。
边界 boundaries：
基于 groupBy 表达式的值数组，用于指定每个存储桶的边界。每个相邻的值对充当包含的下边界和独占的上边界。您必须至少指定两个边界。
指定的值必须按升序排列，并且所有类型都相同。例外情况的值是混合数字类型，例如：[10，NumberLong（20），NumberInt（30）]
例
[0,5,10]数组创建两个桶：
[0,5]包含下界 0 和独占上界 5。
[5,10]包含下界 5 和独占上界 10。
Default:可选
可选的。一个文字，指定包含其 groupBy 表达式结果不属于由边界指定的存储区的所有文

档的其他存储桶的_id。
如果未指定，则每个输入文档必须将 groupBy 表达式解析为由边界指定的某个存储区范围内的值，否则操作将引发错误。
默认值必须小于最低边界值，或者大于或等于最高边界值。
默认值可以是与边界中的条目不同的类型。

output

可选的。除_id 字段外还指定要包括在输出文档中的字段的文档。要指定要包括的字段，必须使用累加器表达式。
<outputfield1>：{<accumulator>：<expression1>},
<outputfieldN>：{<accumulator>：<expressionN>}
指定输出时，输出文档中不包含默认计数字段。显式指定计数表达式作为输出文档的一部分以包含它：
输出：{
<outputField1>：{<accumulator>：<expression1>},
“数”：{$ sum：1}
“字符串”:{$push:”$name”}
}

$ bucket 至少需要满足以下条件之一，否则操作会抛出错误：
每个输入文档将 groupBy 表达式解析为由边界指定的其中一个存储区范围内的值，或
对于其 groupBy 值在边界之外或与边界中的值不同的 BSON 类型的存储文档，指定了默认值。
如果 groupBy 表达式解析为数组或文档，$ bucket 将使用$ sort 中的比较逻辑将输入文档排列到存储桶中。

E:
文档的集合图稿：
{ "_id" : 1, "title" : "The Pillars of Society", "artist" : "Grosz", "year" : 1926,
"price" : NumberDecimal("199.99") }
{ "_id" : 2, "title" : "Melancholy III", "artist" : "Munch", "year" : 1902,
"price" : NumberDecimal("280.00") }
{ "_id" : 3, "title" : "Dancer", "artist" : "Miro", "year" : 1925,
"price" : NumberDecimal("76.04") }
{ "_id" : 4, "title" : "The Great Wave off Kanagawa", "artist" : "Hokusai",
"price" : NumberDecimal("167.30") }
{ "_id" : 5, "title" : "The Persistence of Memory", "artist" : "Dali", "year" : 1931,
"price" : NumberDecimal("483.00") }
{ "_id" : 6, "title" : "Composition VII", "artist" : "Kandinsky", "year" : 1913,
"price" : NumberDecimal("385.00") }
{ "_id" : 7, "title" : "The Scream", "artist" : "Munch", "year" : 1893
/* No price*/ }
{ "_id" : 8, "title" : "Blue Flower", "artist" : "O'Keefe", "year" : 1918,

```
    "price" : NumberDecimal("118.42") }
```

以下操作使用$ bucket 聚合阶段根据价格将图稿集合排列到存储桶中：

```
db.artwork.aggregate( [
  {
    $bucket: {
      groupBy: "$price",
      boundaries: [ 0, 200, 400 ],
      default: "Other",
      output: {
        "count": { $sum: 1 },
        "titles" : { $push: "$title" }
      }
    }
  }
] )
```

存储桶具有以下边界：

[0,200]，包含下限 0 和独占上限 200。

[200,400]具有包含下限 200 和独占上限 400。

“其他”是没有价格或价格超出上述范围的文档的默认存储桶。

该操作返回以下文档：

```
{
  "_id" : 0,
  "count" : 4,
  "titles" : [
    "The Pillars of Society",
    "Dancer",
    "The Great Wave off Kanagawa",
    "Blue Flower"
  ]
}
{
  "_id" : 200,
  "count" : 2,
  "titles" : [
    "Melancholy III",
    "Composition VII"
  ]
}
{
  "_id" : "Other",
  "count" : 2,
  "titles" : [
    "The Persistence of Memory",
    "The Scream"
```

```
   ]
}
```

使用$ bucket 与$ facet

$ bucket 阶段可以在$ facet 阶段中用于在单个聚合阶段中处理艺术品上的多个聚合管道。

以下操作根据价格和年份将文档中的文档分组到存储桶中：

```
db.artwork.aggregate( [
  {
    $facet: {
      "price": [
        {
          $bucket: {
              groupBy: "$price",
              boundaries: [ 0, 200, 400 ],
              default: "Other",
              output: {
                "count": { $sum: 1 },
                "artwork" : { $push: { "title": "$title", "price": "$price" } }
              }
          }
        }
      ],
      "year": [
        {
          $bucket: {
            groupBy: "$year",
            boundaries: [ 1890, 1910, 1920, 1940 ],
            default: "Unknown",
            output: {
              "count": { $sum: 1 },
              "artwork": { $push: { "title": "$title", "year": "$year" } }
            }
          }
        }
      ]
    }
  }
] )
```

R:

```
{
  "year" : [
    {
      "_id" : 1890,
      "count" : 2,
```

```
    "artwork" : [
      {
        "title" : "Melancholy III",
        "year" : 1902
      },
      {
        "title" : "The Scream",
        "year" : 1893
      }
    ]
  },
  {
    "_id" : 1910,
    "count" : 2,
    "artwork" : [
      {
        "title" : "Composition VII",
        "year" : 1913
      },
      {
        "title" : "Blue Flower",
        "year" : 1918
      }
    ]
  },
  {
    "_id" : 1920,
    "count" : 3,
    "artwork" : [
      {
        "title" : "The Pillars of Society",
        "year" : 1926
      },
      {
        "title" : "Dancer",
        "year" : 1925
      },
      {
        "title" : "The Persistence of Memory",
        "year" : 1931
      }
    ]
  },
  {
```

```
      // Includes the document without a year, e.g., _id: 4
      "_id" : "Unknown",
      "count" : 1,
      "artwork" : [
        {
          "title" : "The Great Wave off Kanagawa"
        }
      ]
    }
],
      "price" : [
    {
      "_id" : 0,
      "count" : 4,
      "artwork" : [
        {
          "title" : "The Pillars of Society",
          "price" : NumberDecimal("199.99")
        },
        {
          "title" : "Dancer",
          "price" : NumberDecimal("76.04")
        },
        {
          "title" : "The Great Wave off Kanagawa",
          "price" : NumberDecimal("167.30")
        },
        {
          "title" : "Blue Flower",
          "price" : NumberDecimal("118.42")
        }
      ]
    },
    {
      "_id" : 200,
      "count" : 2,
      "artwork" : [
        {
          "title" : "Melancholy III",
          "price" : NumberDecimal("280.00")
        },
        {
          "title" : "Composition VII",
          "price" : NumberDecimal("385.00")
```

```
            }
          ]
        },
        {
          // Includes the document without a price, e.g., _id: 7
          "_id" : "Other",
          "count" : 2,
          "artwork" : [
            {
              "title" : "The Persistence of Memory",
              "price" : NumberDecimal("483.00")
            },
            {
              "title" : "The Scream"
            }
          ]
        }
      ]
}
```

$bucketAuto：

版本 3.4 中的新功能。
根据指定的表达式将传入的文档分类为特定数量的组（称为存储桶）。自动确定存储桶边界，以尝试将文档均匀地分配到指定数量的存储桶中。
每个存储桶在输出中表示为文档。每个存储桶的文档包含一个_id 字段，其值指定存储桶的包含下限和独占上限，以及包含存储桶中文档数的计数字段。未指定输出时，默认情况下包含计数字段。
$ bucketAuto 阶段具有以下形式：

```
{
  $bucketAuto: {
      groupBy: <expression>,
      buckets: <number>,
      output: {
          <output1>: { <$accumulator expression> },
          ...
      }
      granularity: <string>
  }
}
```

groupBy:(String)
用于分组文档的表达式。要指定字段路径，请在字段名称前加上美元符号$，并将其括在引号中。
Buckets(integer):

一个正 32 位整数，指定输入文档分组到的存储区数。

Output(Document):

可选的。除_id 字段外还指定要包括在输出文档中的字段的文档。要指定要包含的字段，必须使用累加器表达式：

指定输出时，输出文档中不包含默认计数字段。显式指定计数表达式作为输出文档的一部分以包含它：

```
输出：{
 <outputfield1>：{<accumulator>：<expression1>},
 ...
 数：{$ sum：1}
}
```

Granularity():

可选的。一个字符串，指定用于确保计算的边界边缘以首选圆整数或其幂 10 次方结束的首选数字序列。

仅当所有 groupBy 值都是数字且且它们都不是 NaN 时才可用。

支持的粒度值为：

- "R5"
- "R10"
- "R20"
- "R40"
- "R80"
- "1–2–5"
- "E6"
- "E12"
- "E24"
- "E48"
- "E96"
- "E192"
- "POWERSOF2"

如果出现以下情况，可能会少于指定数量的存储桶：

输入文档的数量小于指定的存储桶数量。

groupBy 表达式的唯一值的数量小于指定的桶数。

粒度比桶的数量少。

粒度不够精确，无法将文档均匀分布到指定数量的存储桶中。

如果 groupBy 表达式引用数组或文档，则在确定存储区边界之前，使用与$ sort 中相同的顺序排列值。

跨桶的均匀文档分布取决于 groupBy 字段的基数或唯一值的数量。如果基数不够高，$ bucketAuto 阶段可能无法在桶中均匀分布结果。

粒度

$ bucketAuto 接受可选的粒度参数，该参数确保所有存储桶的边界都符合指定的首选数字系列。使用首选数字系列可以更好地控制在 groupBy 表达式的值范围中设置存储区边界的位置。**当 groupBy 表达式的范围呈指数级变化时，**它们还可用于对数地帮助并均匀地设置桶边界。

雷纳系列

Renard 数字系列是通过取 10 的第 5，第 10，第 20，第 40 或第 80 根得到的数字集合，然后包括等于 1.0 到 10.0 之间的值的各种权力（10.3 in R80 的情况）。

将粒度设置为 R5, R10, R20, R40 或 R80, 以将存储区边界限制为系列中的值。当 groupBy 值超出 1.0 到 10.0（R80 为 10.3）范围时，系列的值乘以 10 的幂。

例

R5 系列基于 10 的第五个根，即 1.58，包括该根的各种幂（圆形），直到达到 10。 R5 系列推导如下：

- $10^{0/5} = 1$
- $10^{1/5} = 1.584 \sim 1.6$
- $10^{2/5} = 2.511 \sim 2.5$
- $10^{3/5} = 3.981 \sim 4.0$
- $10^{4/5} = 6.309 \sim 6.3$
- $10^{5/5} = 10$

E 数系列与 Renard 系列类似，它们将间隔从 1.0 到 10.0 细分为 10 的第 6，第 12，第 24，第 48，第 96 或第 192 根，具有特定的相对误差。

将粒度设置为 E6，E12，E24，E48，E96 或 E192，以将存储区边界限制为系列中的值。当 groupBy 值超出 1.0 到 10.0 范围时，系列的值乘以 10 的幂。要了解有关 E 系列及其各自相对错误的更多信息，请参阅首选系列。

1-2-5 系列

如果存在这样的系列，1-2-5 系列就像一个三值的 Renard 系列。

将粒度设置为 1-2-5，以将桶边界限制为 10 的第三个根的各种幂，舍入为一个有效数字。

以下数字坚持两个系列的力量：

2 0 = 1
2 1 = 2
2 2 = 4
2 3 = 8
2 4 = 16
2 5 = 32

一个常见的实现是各种计算机组件（如内存）通常如何遵循 POWERSOF2 首选数字集：

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and so on….

比较不同的粒度

以下操作演示了如何为粒度指定不同的值会影响$ bucketAuto 如何确定存储区边界。一组东西的_id 编号从 1 到 100：
粒度的不同值将替换为以下操作：

```
db.things.aggregate( [
  {
    $bucketAuto: {
      groupBy: "$_id",
      buckets: 5,
      granularity: <granularity>
    }
  }
] )
```

下表中的结果演示了粒度的不同值如何产生不同的存储区边界：

Granularity	Results	Notes
No granularity	{ “_id” : { “min” : 0, “max” : 20 }, “count” : 20 } { “_id” : { “min” : 20, “max” : 40 }, “count” : 20 } { “_id” : { “min” : 40, “max” : 60 }, “count” : 20 } { “_id” : { “min” : 60, “max” : 80 }, “count” : 20 } { “_id” : { “min” : 80, “max” : 99 }, “count” : 20 }	
R20	{ “_id” : { “min” : 0, “max” : 20 }, “count” : 20 } { “_id” : { “min” : 20, “max” : 40 }, “count” : 20 } { “_id” : { “min” : 40, “max” : 63 }, “count” : 23 } { “_id” : { “min” : 63, “max” : 90 }, “count” : 27 } { “_id” : { “min” : 90, “max” : 100 }, “count” : 10 }	
E24	{ “_id” : { “min” : 0, “max” : 20 }, “count” : 20 } { “_id” : { “min” : 20, “max” : 43 }, “count” : 23 } { “_id” : { “min” : 43, “max” : 68 }, “count” : 25 } { “_id” : { “min” : 68, “max” : 91 }, “count” : 23 } { “_id” : { “min” : 91, “max” : 100 }, “count” : 9 }	
1-2-5	{ “_id” : { “min” : 0, “max” : 20 }, “count” : 20 } { “_id” : { “min” : 20, “max” : 50 }, “count” : 30 } { “_id” : { “min” : 50, “max” : 100 }, “count” : 50 }	The specified n number of inter
POWERSOF2	{ “_id” : { “min” : 0, “max” : 32 }, “count” : 32 } { “_id” : { “min” : 32, “max” : 64 }, “count” : 32 } { “_id” : { “min” : 64, “max” : 128 }, “count” : 36 }	The specified n number of inter

E:

集合如下

```
{ "_id" : 1, "title" : "The Pillars of Society", "artist" : "Grosz", "year" : 1926,
    "price" : NumberDecimal("199.99"),
    "dimensions" : { "height" : 39, "width" : 21, "units" : "in" } }
{ "_id" : 2, "title" : "Melancholy III", "artist" : "Munch", "year" : 1902,
    "price" : NumberDecimal("280.00"),
    "dimensions" : { "height" : 49, "width" : 32, "units" : "in" } }
{ "_id" : 3, "title" : "Dancer", "artist" : "Miro", "year" : 1925,
    "price" : NumberDecimal("76.04"),
    "dimensions" : { "height" : 25, "width" : 20, "units" : "in" } }
{ "_id" : 4, "title" : "The Great Wave off Kanagawa", "artist" : "Hokusai",
    "price" : NumberDecimal("167.30"),
    "dimensions" : { "height" : 24, "width" : 36, "units" : "in" } }
{ "_id" : 5, "title" : "The Persistence of Memory", "artist" : "Dali", "year" : 1931,
    "price" : NumberDecimal("483.00"),
    "dimensions" : { "height" : 20, "width" : 24, "units" : "in" } }
{ "_id" : 6, "title" : "Composition VII", "artist" : "Kandinsky", "year" : 1913,
    "price" : NumberDecimal("385.00"),
    "dimensions" : { "height" : 30, "width" : 46, "units" : "in" } }
{ "_id" : 7, "title" : "The Scream", "artist" : "Munch",
    "price" : NumberDecimal("159.00"),
    "dimensions" : { "height" : 24, "width" : 18, "units" : "in" } }
{ "_id" : 8, "title" : "Blue Flower", "artist" : "O'Keefe", "year" : 1918,
    "price" : NumberDecimal("118.42"),
    "dimensions" : { "height" : 24, "width" : 20, "units" : "in" } }
```

在以下操作中，输入文档根据价格字段中的值分组为四个桶，且会根据边界值划分为最小值和最大值。：

```
db.artwork.aggregate( [
    {
      $bucketAuto: {
          groupBy: "$price",
          buckets: 4
      }
    }
] )
```

R:

```
{
  "_id" : {
    "min" : NumberDecimal("76.04"),
    "max" : NumberDecimal("159.00")
  },
  "count" : 2
}
{
```

```
    "_id" : {
      "min" : NumberDecimal("159.00"),
      "max" : NumberDecimal("199.99")
    },
    "count" : 2
  }
  {
    "_id" : {
      "min" : NumberDecimal("199.99"),
      "max" : NumberDecimal("385.00")
    },
    "count" : 2
  }
  {
    "_id" : {
      "min" : NumberDecimal("385.00"),
      "max" : NumberDecimal("483.00")
    },
    "count" : 2
  }
```

多面聚合

$ bucketAuto 阶段可以在$ facet 阶段中使用，以处理来自艺术作品的同一组输入文档上的多个聚合管道。

以下聚合管道根据价格，年份和计算区域将艺术品集合中的文档分组到存储桶中：

```
db.artwork.aggregate( [
  {
    $facet: {
      "price": [
        {
          $bucketAuto: {
            groupBy: "$price",
            buckets: 4
          }
        }
      ],
      "year": [
        {
          $bucketAuto: {
            groupBy: "$year",
            buckets: 3,
            output: {
              "count": { $sum: 1 },
              "years": { $push: "$year" }
            }
```

```
            }
          }
        ],
        "area": [
          {
            $bucketAuto: {
              groupBy: {
                $multiply: [ "$dimensions.height", "$dimensions.width" ]
              },
              buckets: 4,
              output: {
                "count": { $sum: 1 },
                "titles": { $push: "$title" }
              }
            }
          }
        ]
      }
    }
] )
```

R:


```
{
   "area" : [
      {
         "_id" : { "min" : 432, "max" : 500 },
         "count" : 3,
         "titles" : [
            "The Scream",
            "The Persistence of Memory",
            "Blue Flower"
         ]
      },
      {
         "_id" : { "min" : 500, "max" : 864 },
         "count" : 2,
         "titles" : [
            "Dancer",
            "The Pillars of Society"
         ]
      },
      {
         "_id" : { "min" : 864, "max" : 1568 },
         "count" : 2,
         "titles" : [
```

```
          "The Great Wave off Kanagawa",
          "Composition VII"
        ]
      },
      {
        "_id" : { "min" : 1568, "max" : 1568 },
        "count" : 1,
        "titles" : [
          "Melancholy III"
        ]
      }
    ],
    "price" : [
      {
        "_id" : { "min" : NumberDecimal("76.04"), "max" : NumberDecimal("159.00") },
        "count" : 2
      },
      {
        "_id" : { "min" : NumberDecimal("159.00"), "max" : NumberDecimal("199.99") },
        "count" : 2
      },
      {
        "_id" : { "min" : NumberDecimal("199.99"), "max" : NumberDecimal("385.00") },
        "count" : 2 },
      {
        "_id" : { "min" : NumberDecimal("385.00"), "max" : NumberDecimal("483.00") },
        "count" : 2
      }
    ],
    "year" : [
      { "_id" : { "min" : null, "max" : 1913 }, "count" : 3, "years" : [ 1902 ] },
      { "_id" : { "min" : 1913, "max" : 1926 }, "count" : 3, "years" : [ 1913, 1918, 1925 ] },
      { "_id" : { "min" : 1926, "max" : 1931 }, "count" : 2, "years" : [ 1926, 1931 ] }
    ]
}
```

$collStats(aggregation):

版本 3.4 中的新功能。
返回有关集合或视图的统计信息。
$ collStats 阶段具有以下原型形式：

```
{
  $collStats:
    {
      latencyStats: { histograms: <boolean> },
      storageStats: {},
      count: {}
    }
}
```

latencyStats 将延迟统计信息添加到返回文档中。
latencyStats.histograms:如果为 true，则将延迟直方图信息添加到 latencyStats 中的嵌入文档中。
storageStats:将存储统计信息添加到返回文档。
count:将集合中的文档总数添加到返回文档中。

对于副本集中的集合或集群中的非分片集合，$ collStats 会输出单个文档。对于分片集合，$ collStats 为每个分片输出一个文档。输出文档包括以下字段：
Ns:
请求的集合或视图的命名空间。
Shard:
输出文档对应的分片的名称。
仅当$ collStats 在分片群集上运行时才会出现。分片和非分片集合都将生成此字段。
版本 3.6 中的新功能。
Host
生成输出文档的 mongod 进程的主机名和端口。
版本 3.6 中的新功能。
localTime :
MongoDB 服务器上的当前时间，表示自 Unix 纪元以来的 UTC 毫秒数。
latencyStats
与集合或视图的请求延迟相关的统计信息集合。有关此文档的详细信息，请参阅 latencyStats 文档。
仅在指定 latencyStats：{}选项时出现。
storageStats:
与集合的存储引擎相关的统计信息集合。有关此文档的详细信息，请参阅 storageStats 文档。
仅在指定 storageStats：{}选项时出现。如果应用于视图，则返回错误。
Count:
集合中的文档总数。 storageStats.count 中也提供了此数据。

注意:

计数基于集合的元数据，为分片集群提供快速但有时不准确的计数。

仅在指定 count：{}选项时出现。如果应用于视图，则返回错误。

行为

$ collStats 必须是聚合管道中的第一个阶段，否则管道会返回错误。

Transactions

Transactions 中不允许使用$ collStats。

latencyStats 文档

如果指定 latencyStats 选项，则 latencyStats 嵌入文档仅存在于输出中。

Read:读取请求的延迟统计信息

Writes：写入请求的延迟统计信息。

Commands：数据库命令的延迟统计信息。

例如，如果在矩阵集合上运行带有 latencyStats：{}选项的$ collStats：

db.matrices.aggregate([{ $collStats: { latencyStats: { histograms: true } } }])

R:


```
{ "ns" : "test.matrices",
   "host" : mongo.example.net:27017",
   "localTime" : ISODate("2017-10-06T19:43:56.599Z"),
   "latencyStats" :
     { "reads" :
          { "histogram" : [
               { "micros" : NumberLong(16),
                 "count" : NumberLong(3) },
               { "micros" : NumberLong(32),
                 "count" : NumberLong(1) },
               { "micros" : NumberLong(128),
                 "count" : NumberLong(1) } ],
            "latency" : NumberLong(264),
            "ops" : NumberLong(5) },
       "writes" :
          { "histogram" : [
               { "micros" : NumberLong(32),
                 "count" : NumberLong(1) },
               { "micros" : NumberLong(64),
                 "count" : NumberLong(3) },
               { "micros" : NumberLong(24576),
                 "count" : NumberLong(1) } ],
            "latency" : NumberLong(27659),
            "ops" : NumberLong(5) },
       "commands" :
          { "histogram" : [ ],
            "latency" : NumberLong(0),
            "ops" : NumberLong(0) },
       "transactions" : {
```

```
            "histogram" : [ ],
            "latency" : NumberLong(0),
            "ops" : NumberLong(0)
        }
    }
}
```

如果指定 storageStats 选项，则 storageStats 嵌入文档仅存在于输出中。

本文档的内容取决于所使用的存储引擎。请参阅输出以获取本文档的参考。

例如，如果使用 WiredTiger 存储引擎在矩阵集合上运行带有 storageStats：{}选项的$ collStats：

例如，如果使用 WiredTiger 存储引擎在矩阵集合上运行带有 storageStats：{}选项的$ collStats：

```
db.matrices.aggregate( [ { $collStats: { storageStats: { } } } ] )
```

R：

```
{
   "ns" : "test.matrices",
   "host" : mongo.example.net:27017",
   "localTime" : ISODate("2017-10-06T19:43:56.599Z"),
   "storageStats" : {
      "size" : 608500363,
      "count" : 1104369,
      "avgObjSize" : 550,
      "storageSize" : 352878592,
      "capped" : false,
      "wiredTiger" : {
         ...
      },
      "nindexes" : 1,
      "indexDetails" : {
         ...
      },
      "totalIndexSize" : 9891840,
      "indexSizes" : {
         "_id_" : 9891840
      }
   }
}
```

Count 领域

版本 3.6 中的新功能。

如果指定 count 选项，则 count 字段仅存在于输出中。

例如，如果在矩阵集合上运行带有 count：{}选项的$ collStats：

```
db.matrices.aggregate( [ { $collStats: { count: { } } } ] )
```

R:

```
{
   "ns" : "test.matrices",
   "host" : mongo.example.net:27017",
   "localTime" : ISODate("2017-10-06T19:43:56.599Z"),
   "count" : 1103869
}
```

当指定 storageStats：{}时，集合中的文档总数也可用作 storageStats.count。有关更多信息，请参阅 storageStats 文档。

在 Sharded Collections 上的$ collStats

$ sharStats 在分片集合上运行时为每个分片输出一个文档。每个输出文档都包含一个分片字段，其中包含文档对应的分片的名称。

例如，如果在分片集合上运行$ collStats，并在名为 matrices 的集合上使用 count：{}选项：

```
db.matrices.aggregate( [ { $collStats: { count: { } } } ] )
```

R;

```
{
   "ns" : "test.matrices",
   "shard" : "s1",
   "host" : "s1-mongo1.example.net:27017",
   "localTime" : ISODate("2017-10-06T15:14:21.258Z"),
   "count" : 661705
}
{
   "ns" : "test.matrices",
   "shard" : "s2",
   "host" : "s2-mongo1.example.net:27017",
   "localTime" : ISODate("2017-10-06T15:14:21.258Z"),
   "count" : 442164
}
```

$count (aggregation)

版本 3.4 中的新功能。

将文档传递到下一个阶段，该阶段包含输入到阶段的文档数量的计数。

$ count 具有以下原型形式：

```
{ $count: <string> }
```

<string>是输出字段的名称，其计数值为其值。 <string>必须是非空字符串，不能以$开头且不得包含。字符。

兄弟成员：

- db.collection.countDocuments()

- $collStats
- db.collection.estimatedDocumentCount()
- count
- db.collection.count()

$ count 阶段相当于以下$ group + $ project 序列：

```
db.collection.aggregate( [
    { $group: { _id: null, myCount: { $sum: 1 } } },
    { $project: { _id: 0 } }
] )
```

其中 myCount 将是包含计数的输出字段。您可以为输出字段指定另一个名称。

E:名为 scores 的集合具有以下文档：

```
{ "_id" : 1, "subject" : "History", "score" : 88 }
{ "_id" : 2, "subject" : "History", "score" : 92 }
{ "_id" : 3, "subject" : "History", "score" : 97 }
{ "_id" : 4, "subject" : "History", "score" : 71 }
{ "_id" : 5, "subject" : "History", "score" : 79 }
{ "_id" : 6, "subject" : "History", "score" : 83 }
```

以下聚合操作有两个阶段：

1$ match 阶段排除得分值小于或等于 80 的文档，将得分大于 80 的文档传递到下一阶段。

2$ count 阶段返回聚合管道中剩余文档的计数，并将值分配给名为 passing_scores 的字段。

```
db.scores.aggregate(
  [
    {
      $match: {
        score: {
          $gt: 80
        }
      }
    },
    {
      $count: "passing_scores"
    }
  ]
)
```

R:
{ "passing_scores" : 4 }

$ currentOp

返回包含有关活动和/或休眠操作的信息的文档流，以及作为事务的一部分持有锁的非活动会话。该阶段为每个操作或会话返回一个文档。要运行$ currentOp，请在 admin 数据库上使用 db.aggregate（）帮助程序。
$ currentOp 聚合阶段优先于 currentOp 命令及其 mongo shell 帮助程序 db.currentOp（）。由于 currentOp 命令和 db.currentOp（）帮助程序在单个文档中返回结果，**因此 currentOp 结果集的总大小受文档的最大 16MB BSON 大小限制。**$ currentOp 阶段在文档流上返回一个光标，每个文档报告一个操作。每个操作文档都受 16MB BSON 限制，但与 currentOp 命令不同，结果集的总体大小没有限制。
$ currentOp 还允许您在文档通过管道时执行结果的任意转换。
$ currentOp 将选项文档作为其操作数：Changed in version 4.0.
格式：
{ $currentOp: { allUsers: <boolean>, idleConnections: <boolean>, idleSessions: <boolean>, localOps: <boolean> } }
allUsers
布尔。如果设置为 false，$ currentOp 将仅报告属于运行该命令的用户的操作。如果设置为 true，$ currentOp 将报告属于所有用户的操作。
注意
使用{allUsers：true}运行$ currentOp 需要 inprog 权限 ,默认为 false。
idleConnections
布尔。如果设置为 false，$ currentOp 将仅报告活动操作。如果设置为 true，则将返回包括空闲连接在内的所有操作。默认为 false。
idleSessions
布尔。如果设置为 true，那么除了报告活动/休眠操作之外，$ currentOp 还会返回有关作为事务一部分持有锁的非活动会话的信息。每个非活动会话将在$ currentOp 流中显示为单独的文档。
会话的文档包括有关 lsid 字段中的会话 ID 和事务字段中的事务的信息。
如果设置为 false，$ currentOp 将不报告非活动会话。
默认为 true。
4.0 版中的新功能。
localOps
布尔。如果对在 mongos 上运行的聚合设置为 true，则$ currentOp 仅报告在该 mongos 上本地运行的操作。如果为 false，则$ currentOp 将报告在分片上运行的操作。
localOps 参数对在 mongod 上运行的$ currentOp 聚合没有影响。
默认为 false。
4.0 版中的新功能。

省略上述任何参数将导致$ currentOp 使用该参数的默认值。指定一个空文档，如下所示，以使用所有参数的默认值。

{ $currentOp: { } }

约束

$ currentOp 必须是管道中的第一个阶段。

以$ currentOp 开头的管道只能在 admin 数据库上运行。

在启用了身份验证的独立或副本集上，如果 allUsers 参数设置为 true，则需要 inprog 权限才能运行$ currentOp。

在分片群集上，需要 inprog 权限才能运行所有$ currentOp 管道。

交易中不允许$ currentOp。

例

以下示例在第一阶段中运行$ currentOp 操作，并在第二阶段中过滤该操作的结果。具体而言，第一阶段返回所有活动操作的文档以及作为事务的一部分持有锁的非活动会话。第二阶段仅筛选与作为事务的一部分持有锁的非活动会话相关的那些文档。

```
db.getSiblingDB("admin").aggregate( [
    { $currentOp : { allUsers: true, idleSessions: true } },
    { $match : { active: false, transaction : { $exists: true } } }
] )
```

R:

```
{
    "host" : "example.mongodb.com:27017",
    "desc" : "inactive transaction",
    "client" : "198.51.100.1:53809",
    "connectionId" : NumberLong(40),
    "appName" : "",
    "clientMetadata" : {
        "driver" : {
            "name" : "PyMongo",
            "version" : "3.7.2"
        },
        "os" : {
            "type" : "Darwin",
            "name" : "Darwin",
            "architecture" : "x86_64",
            "version" : "10.14.2"
        },
        "platform" : "CPython 3.7.1.final.0"
    },
    "lsid" : {
        "id" : UUID("02039c77-3f05-4dd6-a42d-d2168a73fc95"),
        "uid"                                                                    :
BinData(0,"47DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU=")
    },
    "transaction" : {
        "parameters" : {
```

```
            "txnNumber" : NumberLong(1),
            "autocommit" : false,
            "readConcern" : {
                "level" : "snapshot"
            }
        },
        "readTimestamp" : Timestamp(1548869277, 3),
        "startWallClockTime" : "2019-01-30T12:28:06.979-0500",
        "timeOpenMicros" : NumberLong(34930476),
        "timeActiveMicros" : NumberLong(210),
        "timeInactiveMicros" : NumberLong(34930266),
        "expiryTime" : "2019-01-30T12:29:06.979-0500"
    }
    "waitingForLock" : false,
    "active" : false,
    "locks" : {
        "Global" : "w",
        "Database" : "w",
        "Collection" : "w"
    },
    "lockStats" : {
        "Global" : {
            "acquireCount" : {
                "r" : NumberLong(2),
                "w" : NumberLong(1)
            }
        },
        "Database" : {
            "acquireCount" : {
                "r" : NumberLong(1),
                "w" : NumberLong(1)
            }
        },
        "Collection" : {
            "acquireCount" : {
                "r" : NumberLong(1),
                "w" : NumberLong(1)
            }
        }
    }
}
```


Output fileds

每个输出文档可能包含与操作相关的以下字段的子集：

$ currentOp.host
运行操作的主机的名称。
$ currentOp.shard
运行操作的分片的名称。
仅适用于分片群集。
$ currentOp.desc
操作说明。
$ currentOp.connectionId
特定操作发起的连接的标识符。
$ currentOp.client
操作所源自的客户端连接的 IP 地址（或主机名）和临时端口。
对于多文档事务，$ currentOp.client 存储有关在事务内运行操作的最新客户端的信息。
仅适用于标准版和副本集
$ currentOp.client_s
操作源自的 mongos 的 IP 地址（或主机名）和临时端口。
仅适用于分片群集
$ currentOp.clientMetadata
有关客户的其他信息。
对于多文档事务，$ currentOp.client 存储有关在事务内运行操作的最新客户端的信息。
$ currentOp.appName
版本 3.4 中的新功能。
运行该操作的客户端应用程序的标识符。使用 appName 连接字符串选项为 appName 字段设置自定义值。
$ currentOp.active
一个布尔值，指定操作是否已启动。如果操作已启动，则值为 true;如果操作处于空闲状态，则为 false，例如空闲连接，非活动会话或当前空闲的内部线程。即使操作已经产生到另一个操作，操作也可以是活动的。
$ currentOp.currentOpTime
操作的开始时间。
版本 3.6 中的新功能。
$ currentOp.opid
操作的标识符。您可以将此值传递给 mongo shell 中的 db.killOp（）以终止操作。
警告
极其谨慎地终止运行操作。仅使用 db.killOp（）来终止客户端启动的操作，并且不终止内部数据库操作。
$ currentOp.secs_running
以秒为单位的操作持续时间。MongoDB 通过从操作的开始时间减去当前时间来计算此值。仅在操作运行时出现;即如果活动是真的。
$ currentOp.microsecs_running
操作的持续时间以微秒为单位。 MongoDB 通过从操作的开始时间减去当前时间来计算此值。

仅在操作运行时出现;即如果活动是真的。
$ currentOp.lsid

会话标识符。
仅在操作与会话关联时才出现。
版本 3.6 中的新功能。
$ currentOp.transaction
包含多文档事务信息的文档。
仅在操作是事务的一部分时才出现。
4.0 版中的新功能。
$ currentOp.transaction.parameters
包含有关多文档事务的信息的文档。
仅在操作是事务的一部分时才出现。
4.0 版中的新功能。
$ currentOp.transaction.parameters.txnNumber
交易号码。
仅在操作是事务的一部分时才出现。
4.0 版中的新功能。
$ currentOp.transaction.parameters.autocommit
一个布尔标志，指示是否为事务启用了自动提交。
仅在操作是事务的一部分时才出现。
版本 4.0.2 中的新功能。
$ currentOp.transaction.parameters.readConcern
阅读关注交易。
多文档事务支持读取关注“快照”，“本地”和“多数”。
仅在操作是事务的一部分时才出现。
版本 4.0.2 中的新功能。
$ currentOp.transaction.readTimestamp
事务中的操作读取快照的时间戳。
仅在操作是事务的一部分时才出现。
版本 4.0.2 中的新功能。
$ currentOp.transaction.startWallClockTime
交易开始的日期和时间（带时区）。
仅在操作是事务的一部分时才出现。
版本 4.0.2 中的新功能。
$ currentOp.transaction.timeOpenMicros
事务的持续时间（以微秒为单位）。
添加到 timeInactiveMicros 的 timeActiveMicros 值应等于 timeOpenMicros。
仅在操作是事务的一部分时才出现。
版本 4.0.2 中的新功能。
$ currentOp.transaction.timeActiveMicros
交易活跃的总时间;即当交易运行时。
添加到 timeInactiveMicros 的 timeActiveMicros 值应等于 timeOpenMicros。
仅在操作是事务的一部分时才出现。

版本 4.0.2 中的新功能。
$ currentOp.transaction.timeInactiveMicros

交易处于非活动状态的总时间;即当交易没有运行的时候。
添加到 timeActiveMicros 的 timeInactiveMicros 值应等于 timeOpenMicros。
仅在操作是事务的一部分时才出现。
$ currentOp.transaction.expiryTime
交易超时和中止的日期和时间（带时区）。
$ currentOp.transaction.expiryTime 等于 $ currentOp.transaction.startWallClockTime + transactionLifetimeLimitSeconds。
有关更多信息，请参阅事务的运行时限制。
仅在操作是事务的一部分时才出现。
版本 4.0.2 中的新功能。
$ currentOp.op
标识操作类型的字符串。可能的值是：

- `"none"`
- `"update"`
- `"insert"`
- `"query"`
- `"command"`
- `"getmore"`
- `"remove"`
- `"killcursors"`

“命令”操作包括大多数命令，例如 createIndexes，aggregate 和 findandmodify。
“查询”操作包括查找操作和 OP_QUERY 操作。
$ currentOp.ns
操作目标的命名空间。命名空间由数据库名称和与点（。）连接的集合名称组成;也就是“<database>。<collection>”。
$ currentOp.command
版本 3.6 中已更改。
包含与此操作关联的完整命令对象的文档。如果命令文档超过 1 千字节，则文档具有以下格式：
“命令”：{
“$ truncated”：<string>,
“comment”：<string>
}
$ truncated 字段包含文档的字符串摘要，不包括文档的注释字段（如果存在）。如果摘要仍然超过 1 千字节，则会进一步截断，在字符串末尾用省略号（...）表示。
如果将注释传递给操作，则会显示注释字段。

以下示例输出包含名为 test 的数据库中名为 items 的集合上的查找操作的命令对象：
"command" : {

```
   "find" : "items",
   "filter" : {
      "sku" : 1403978
   },
   "$db" : "test"
}
```

以下示例输出包含由名为 test 的数据库中名为 items 的集合上由游标 ID 为 80336119321 的命令生成的 getMore 操作的命令对象：

```
"command" : {
   "getMore" : NumberLong("80336119321"),
   "collection" : "items",
   "$db" : "test"
}
```

$ currentOp.originatingCommand

在版本 3.6 中更改：对于从游标检索下一批结果的“getmore”操作，originatingCommand 字段包含最初创建该游标的完整命令对象（例如，查找或聚合）。

$ currentOp.planSummary

包含查询计划的字符串，以帮助调试慢速查询。

$ currentOp.numYields

numYields 是一个计数器，它报告操作已经产生的次数以允许其他操作完成。

通常，当操作需要访问 MongoDB 尚未完全读入内存的数据时，操作会产生。这允许在 MongoDB 读取数据以进行屈服操作时，在内存中具有数据的其他操作可以快速完成。

$ currentOp.locks

锁文档报告当前操作所持有的锁的类型和模式。可能的锁类型如下：

Global：代表 global 锁定

MMAPV1Journal 表示用于同步日志写入的 MMAPv1 存储引擎特定锁;对于非 MMAPv1 存储引擎，MMAPV1Journal 的模式为空。

数据库表示数据库锁定。

Collection：表示集合锁。

元数据表示元数据锁定。

Oplog：表示对 oplog 的锁定。

$ currentOp.waitingForLock

返回一个布尔值。如果操作正在等待锁定，则 waitingForLock 为 true，如果操作具有所需的锁定，则为 false。

$ currentOp.msg

msg 提供描述操作状态和进度的消息。在索引或 mapReduce 操作的情况下，该字段报告完成百分比。

$ currentOp.progress

报告 mapReduce 或索引操作的进度。进度字段对应于 msg 字段中的完成百分比。进度指定以下信息：

$ currentOp.progress.done

报告已完成的工作项数。

$ currentOp.progress.total

报告工作项的总数。
$ currentOp.killPending
如果操作当前标记为终止，则返回 true。当操作遇到下一个安全终止点时，操作将终止。
$ currentOp.lockStats
对于每种锁类型和模式（有关锁类型和模式的描述，请参阅$ currentOp.locks），返回以下信息：
$ currentOp.lockStats.acquireCount
操作在指定模式下获取锁定的次数。
$ currentOp.lockStats.acquireWaitCount
操作必须等待 acquireCount 锁定获取的次数，因为锁定处于冲突模式。 acquireWaitCount 小于或等于 acquireCount。
$ currentOp.lockStats.timeAcquiringMicros
操作必须等待获取锁的累积时间（以微秒为单位）。
timeAcquiringMicros 除以 acquireWaitCount 给出了特定锁定模式的近似平均等待时间。
$ currentOp.lockStats.deadlockCount
操作在等待锁获取时遇到死锁的次数。

$facet(面):

版本 3.4 中的新功能。
在同一组输入文档的单个阶段内处理多个聚合管道。每个子管道在输出文档中都有自己的字段，其结果存储为文档数组。
$ facet 阶段允许您创建多面聚合，这些聚合可在**单个聚合阶段内跨多个维度或方面表征数据**。多面聚合提**供多个过滤器和分类**，以**指导数据浏览和分析**。零售商**通常使用分面来缩小搜索结果，方法是在产品价格，制造商，尺寸等方面创建过滤器**。
输入文档**仅传递给$ facet 阶段一次**。 $ facet 在同一组输入文档上启用各种聚合，而无需多次检索输入文档。
$ facet 阶段具有以下形式：

```
{ $facet:
    {
        <outputField1>: [ <stage1>, <stage2>, ... ],
        <outputField2>: [ <stage1>, <stage2>, ... ],
        ...

    }
}
```

与**构面相关的聚合阶段对传入的文档进行分类和分组**。在不同的$ facet 子管道的<stage>中指定以下任何与 facet 相关的阶段，以执行多面聚合：

- $bucket
- $bucketAuto

- $sortByCount

任何其他聚合阶段也可以与$ facet 一起使用，除了：

- $facet
- $out
- $geoNear
- $indexStats
- $collStats

$ facet 中的每个子管道都传递完全相同的输入文档集。这些子管道完全相互独立，**每个子管道输出的文档数组存储在输出文档的单独字段中**。一个子管道的输出不能用作同一$ facet 阶段内不同子管道的输入。如果需要进一步聚合，请在$ facet 之后添加其他阶段，并指定所需子管道输出的字段名称<outputField>。

例

考虑一个在线商店，其库存存储在以下艺术品集合中：

```
{ "_id" : 1, "title" : "The Pillars of Society", "artist" : "Grosz", "year" : 1926,
   "price" : NumberDecimal("199.99"),
   "tags" : [ "painting", "satire", "Expressionism", "caricature" ] }
{ "_id" : 2, "title" : "Melancholy III", "artist" : "Munch", "year" : 1902,
   "price" : NumberDecimal("280.00"),
   "tags" : [ "woodcut", "Expressionism" ] }
{ "_id" : 3, "title" : "Dancer", "artist" : "Miro", "year" : 1925,
   "price" : NumberDecimal("76.04"),
   "tags" : [ "oil", "Surrealism", "painting" ] }
{ "_id" : 4, "title" : "The Great Wave off Kanagawa", "artist" : "Hokusai",
   "price" : NumberDecimal("167.30"),
   "tags" : [ "woodblock", "ukiyo-e" ] }
{ "_id" : 5, "title" : "The Persistence of Memory", "artist" : "Dali", "year" : 1931,
   "price" : NumberDecimal("483.00"),
   "tags" : [ "Surrealism", "painting", "oil" ] }
{ "_id" : 6, "title" : "Composition VII", "artist" : "Kandinsky", "year" : 1913,
   "price" : NumberDecimal("385.00"),
   "tags" : [ "oil", "painting", "abstract" ] }
{ "_id" : 7, "title" : "The Scream", "artist" : "Munch", "year" : 1893,
   "tags" : [ "Expressionism", "painting", "oil" ] }
{ "_id" : 8, "title" : "Blue Flower", "artist" : "O'Keefe", "year" : 1918,
   "price" : NumberDecimal("118.42"),
   "tags" : [ "abstract", "painting" ] }
```

以下操作使用 MongoDB 的分面功能为客户提供商店的库存，这些库存分为多个维度，例如标签，价格和创建的年份。这个$ facet 阶段有三个子管道，它们使用$ sortByCount，$ bucket 或$ bucketAuto 来执行这个多面聚合。来自艺术作品的输入文档仅在操作开始时从数据库中提取一次：

```
db.artworks.aggregate( [
  {
    $facet: {
      "categorizedByTags": [
        { $unwind: "$tags" },
        { $sortByCount: "$tags" }
      ],
      "categorizedByPrice": [
        // Filter out documents without a price e.g., _id: 7
        { $match: { price: { $exists: 1 } } },
        {
          $bucket: {
            groupBy: "$price",
            boundaries: [    0, 150, 200, 300, 400 ],
            default: "Other",
            output: {
              "count": { $sum: 1 },
              "titles": { $push: "$title" }
            }
          }
        }
      ],
      "categorizedByYears(Auto)": [
        {
          $bucketAuto: {
            groupBy: "$year",
            buckets: 4
          }
        }
      ]
    }
  }
])
```

R:

```
{
  "categorizedByYears(Auto)" : [
    // First bucket includes the document without a year, e.g., _id: 4
    { "_id" : { "min" : null, "max" : 1902 }, "count" : 2 },
    { "_id" : { "min" : 1902, "max" : 1918 }, "count" : 2 },
```

```
    { "_id" : { "min" : 1918, "max" : 1926 }, "count" : 2 },
    { "_id" : { "min" : 1926, "max" : 1931 }, "count" : 2 }
  ],
  "categorizedByPrice" : [
    {
      "_id" : 0,
      "count" : 2,
      "titles" : [
        "Dancer",
        "Blue Flower"
      ]
    },
    {
      "_id" : 150,
      "count" : 2,
      "titles" : [
        "The Pillars of Society",
        "The Great Wave off Kanagawa"
      ]
    },
    {
      "_id" : 200,
      "count" : 1,
      "titles" : [
        "Melancholy III"
      ]
    },
    {
      "_id" : 300,
      "count" : 1,
      "titles" : [
        "Composition VII"
      ]
    },
    {
      // Includes document price outside of bucket boundaries, e.g., _id: 5
      "_id" : "Other",
      "count" : 1,
      "titles" : [
        "The Persistence of Memory"
      ]
    }
  ],
  "categorizedByTags" : [
```

```
        { "_id" : "painting", "count" : 6 },
        { "_id" : "oil", "count" : 4 },
        { "_id" : "Expressionism", "count" : 3 },
        { "_id" : "Surrealism", "count" : 2 },
        { "_id" : "abstract", "count" : 2 },
        { "_id" : "woodblock", "count" : 1 },
        { "_id" : "woodcut", "count" : 1 },
        { "_id" : "ukiyo-e", "count" : 1 },
        { "_id" : "satire", "count" : 1 },
        { "_id" : "caricature", "count" : 1 }
    ]
}
```

$geoNear(aggregation)

以距离指定点最近的顺序输出文档。

$ geoNear 阶段具有以下原型形式：

{ $geoNear: { <geoNear options> } }

$ geoNear 运算符接受包含以下$ geoNear 选项的文档。以与处理文档坐标系相同的单位指定所有距离：

Spherical:boolean

确定 MongoDB 如何计算两点之间的距离：

如果为 true，MongoDB 使用$ nearSphere 语义并使用球面几何计算距离。

如果为 false，MongoDB 使用$ near 语义：2dsphere 索引的球形几何和 2d 索引的平面几何。

默认值：false。

Limit:

optional 要返回的最大文档数。默认值为 100.另请参阅 num 选项。

NUM 号可选。 num 选项提供与 limit 选项相同的功能。两者都定义了要返回的最大文档数。如果包含两个选项，则 num 值将覆盖限制值。

maxDistance ： number

可选的。距文档中心点的最大距离。 MongoDB 将结果限制为距离中心点指定距离内的文档。

如果指定的点是 GeoJSON，则以米为单位指定距离;如果指定的点是传统坐标对，则以弧度为单位。

query：

Document:

可选的。将结果限制为与查询匹配的文档。查询语法是通常的 MongoDB 读操作查询语法。您不能在$ geoNear 阶段的查询字段中指定$ near 谓词。

distanceMultiplier:

可选。乘以查询返回的所有距离的因子。例如，使用 distanceMultiplier 将球形查询返回的

弧度转换为乘以地球半径的公里数。

uniqueDocs 布尔

可选的。如果此值为 true，则查询将返回匹配的文档一次，即使文档的多个位置字段与查询匹配也是如此。

自 2.6 版本后不再使用：地理空间查询不再返回重复结果。 $ uniqueDocs 运算符对结果没有影响。

Near:

GeoJSON 点或传统坐标对

找到最接近的文件的关键点。

如果使用 2dsphere 索引，则可以将该点指定为 GeoJSON 点或传统坐标对。

如果使用 2d 索引，请将该点指定为旧坐标对。

distanceField string

包含计算距离的输出字段。要在嵌入文档中指定字段，请使用点表示法。

includeLocs string 可选。这指定了标识用于计算距离的位置的输出字段。当位置字段包含多个位置时，此选项很有用。要在嵌入文档中指定字段，请使用点表示法。

minDistance number

可选的。距文档中心点的最小距离。 MongoDB 将结果限制为超出距中心点指定距离的文档。

指定 GeoJSON 数据的距离（以米为单位）和传统坐标对的弧度。

版本 3.2 中的新功能。

Key

可选的。指定计算距离时要使用的地理空间索引字段。

如果您的集合具有多个 2d 和/或多个 2dsphere 索引，则必须使用 key 选项指定要使用的索引字段路径。指定要使用的地理空间索引提供完整示例。

如果有多个 2d 索引或多个 2dsphere 索引并且您没有指定密钥，MongoDB 将返回错误。

如果您没有指定密钥，并且最多只有一个 2d 索引和/或只有一个 2dsphere 索引，则 MongoDB 首先查找要使用的 2d 索引。如果不存在 2d 索引，则 MongoDB 会查找要使用的 2dphere 索引。

4.0 版中的新功能。

使用$ geoNear 时，请考虑：

1.您只能将$ geoNear 用作管道的第一个阶段。

2.您必须包含 distanceField 选项。 distanceField 选项指定将包含计算距离的字段。

3.$ geoNear 需要地理空间索引。

4.如果集合上有多个地理空间索引，请使用 keys 参数指定要在计算中使用的字段。如果只有一个地理空间索引，$ geoNear 会隐式使用索引字段进行计算。

6.您不能在**$ geoNear 阶**段的查询字段中指定$ near 谓词。

7.视图不支持 geoNear 操作（即$ geoNear 管道阶段和不推荐使用的 geoNear 命令）。

通常，$ geoNear 的选项与不推荐使用的 geoNear 命令类似，但有以下例外：

distanceField 是$ geoNear 管道运算符的必填字段; geoNear 命令中不存在该选项。

includeLocs 接受$ geoNear 管道运算符中的字符串和 geoNear 命令中的布尔值。

例

```
db.shop.insert({"loc": [10, 10]});
db.shop.insert({"loc": [11, 10]});
```

```
db.shop.insert({"loc": [10, 11]});
db.shop.insert({"loc": [12, 15]});
db.shop.insert({"loc": [16, 17]});
db.shop.insert({"loc": [90, 90]});
db.shop.insert({"loc": [120, 130]});
```

```
> db.shop.insert({"loc": [10, 10]});
> db.shop.insert({"loc": [11, 10]});
> db.shop.insert({"loc": [10, 11]});
> db.shop.insert({"loc": [12, 15]});
> db.shop.insert({"loc": [16, 17]});
> db.shop.insert({"loc": [90, 90]});
> db.shop.insert({"loc": [120, 130]});
```

添加索引

```
db.shop.createIndex({"loc": "2d"});
```

db.shop.createIndex({"loc": "2d"});

聚合查询

```
db.shop.aggregate([
...         {"$geoNear": {
...             "near": [11, 12],
...             "distanceField": "loc",
...             "maxDistance": 1,
...             "num": 2,
...             "spherical": true
...         }}
...     ]).pretty();
...     ]).pretty();
{ "_id" : ObjectId("5d9c46b7acee5fcf3543263d"), "loc" : 0.02443600609238405 }
{ "_id" : ObjectId("5d9c46b7acee5fcf3543263c"), "loc" : 0.03490658503988659 }
```

从 4.2 版开始，MongoDB 删除了$geoNear 阶段的 limit()和 num 选项，以及 100 个文档的默认限制。要限制$geoNear 的结果，可以使用$geoNear 阶段和$limit 阶段。

考虑具有 2dsphere 索引的集合位置。以下聚合查找最多 5 个唯一文档，其位置距离中心最多 2 米[-73.99279,40.719296]，类型等于 public：

```
db.places.aggregate([
    {
      $geoNear: {
         near: { type: "Point", coordinates: [ -73.99279 , 40.719296 ] },
```

```
            distanceField: "dist.calculated",
            maxDistance: 2,
            query: { type: "public" },
            includeLocs: "dist.location",
            num: 5,
            spherical: true
        }
    }
])
```

R:

```
{
    "_id" : 8,
    "name" : "Sara D. Roosevelt Park",
    "type" : "public",
    "location" : {
        "type" : "Point",
        "coordinates" : [ -73.9928, 40.7193 ]
    },
    "dist" : {
        "calculated" : 0.9539931676365992,
        "location" : {
            "type" : "Point",
            "coordinates" : [ -73.9928, 40.7193 ]
        }
    }
}
```

匹配文档包含两个新字段：
dist.calculated 字段，包含计算的距离
dist.location 字段，包含计算中使用的位置。

最小距离
版本 3.2 中的新功能。
以下示例使用 minDistance 选项指定距文档中心点的最小距离。 MongoDB 将结果限制为超出距中心点指定距离的文档。

```
db.places.aggregate([
    {
      $geoNear: {
         near: { type: "Point", coordinates: [ -73.99279 , 40.719296 ] },
         distanceField: "dist.calculated",
         minDistance: 2,
         query: { type: "public" },
         includeLocs: "dist.location",
         num: 5,
         spherical: true
```

```
        }
    }
])
```

定要使用的地理空间索引

4.0 版中的新功能。

考虑在位置字段上具有 2dsphere 索引并在旧字段上具有 2d 索引的场所集合。

places 集合中的文档类似于以下内容：

```
{
    "_id" : 3,
    "name" : "Polo Grounds",
    "location": {
        "type" : "Point",
        "coordinates" : [ -73.9375, 40.8303 ]
    },
    "legacy" : [ -73.9375, 40.8303 ],
    "category" : "Stadiums"
}
```

以下示例使用 keys 选项指定聚合应使用$ geoNear 操作的位置字段值而不是旧字段值。

```
db.places.aggregate([
    {
      $geoNear: {
          near: { type: "Point", coordinates: [ -73.98142 , 40.71782 ] },
          key: "location",
          distanceField: "dist.calculated",
          query: { "category": "Parks" }
      }
    }
])
```

R:

```
{
    "_id" : 2,
    "name" : "Sara D. Roosevelt Park",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -73.9928,
            40.7193
        ]
    },
    "category" : "Parks",
    "dist" : {
        "calculated" : 974.175764916902
```

```
    }
}
{
    "_id" : 1,
    "name" : "Central Park",
    "location" : {
        "type" : "Point",
        "coordinates" : [
            -73.97,
            40.77
        ]
    },
    "legacy" : [
        -73.97,
        40.77
    ],
    "category" : "Parks",
    "dist" : {
        "calculated" : 5887.92792958097
    }
}
```

$graphLookup(aggregation)

对集合执行**递归搜索**，其中包含通过递归深度和查询过滤器限制搜索的选项。
$ graphLookup 搜索过程总结如下：
输入文档流入聚合操作的$ graphLookup 阶段。
$ graphLookup 将搜索目标定位到 from 参数指定的集合（请参阅下面的搜索参数的完整列表）。
对于每个输入文档，搜索以 startWith 指定的值开头。
$ graphLookup 将 startWith 值与来自集合中其他文档中 connectToField 指定的字段进行匹配。

对于每个匹配的文档，$ graphLookup 获取 connectFromField 的值，并检查 from 集合中的每个文档以获取匹配的 connectToField 值。对于每个匹配，$ graphLookup 将 from 集合中的匹配文档添加到 as 参数指定的数组字段中。
此步骤以递归方式继续，直到找不到更多匹配的文档，或者直到操作达到 maxDepth 参数指定的递归深度。 $ graphLookup 然后将数组字段附加到输入文档。 $ graphLookup 在完成对所有输入文档的搜索后返回结果。
$ graphLookup 具有以下原型形式：

```
{
    $graphLookup: {
        from: <collection>,
        startWith: <expression>,
```

```
        connectFromField: <string>,
        connectToField: <string>,
        as: <string>,
        maxDepth: <number>,
        depthField: <string>,
        restrictSearchWithMatch: <document>
    }
}
```

from:

目标集合中为$ graphLookup 操作进行搜索，以递归方式将 connectFromField 与 connectToField 匹配。 from 集合**不能分片**，并且**必须与操作中使用的任何其他集**合位于同一数据库中。有关信息，请参阅 Sharded Collections。

startWith Expression:

指定用于**启动递归搜索的 connectFromField 的值**。可选地，startWith 可以是值的数组，每个值分别遵循遍历过程。

connectFromField

其值$ graphLookup 用于**递归匹配集合中其他文档**的 connectToField 的字段名称。如果值是一个数组，则每**个元素都会单独执行遍历过程**。

connectToField:其他文档中的字段名称，与 connectFromField 参数指定的字段值相匹配。

As:添加到每个输出文档的数组字段的名称。包含$ graphLookup 阶段中遍历的文档以访问文档。

注意

在 as 字段中返回的文档不保证按任何顺序排列。

MAXDEPTH:可选。指定**最大递归深度的非负整数。**

depthField:可选。要添加到搜索路径中每个遍历文档的字段的名称。此字段的值是文档的递归深度，表示为 NumberLong。**递归深度值从零开始，因此第一个查找对应于零深度。**

restrictSearchWithMatch

可选的。指定递归搜索的附加条件的文档。语法与查询过滤器语法相同。

Note：

您不能在此过滤器中使用任何聚合表达式。例如，查询文档如

{lastName：{$ ne： “$ lastName”}}

在此上下文中将无法查找 lastName 值与输入文档的 lastName 值不同的文档，因为“$ lastName”将充当字符串文字，而不是字段路径。

注意事项

Sharded Collections

无法对分割中指定的集合进行分片。但是，**可以对运行 aggregate（）方法的集合进行分片。**也就是说，在以下内容中：

```
db.collection.aggregate([
    { $graphLookup: { from: "fromCollection", ... } }
])
```

该集合可以分片。

fromCollection 无法分片。

要加入多个分片集合，请考虑：

修改客户端应用程序以执行手动查找，而不是使用$ graphLookup 聚合阶段。

如果可能，使用嵌入式数据模型，无需加入集合。

Max Depth

将 maxDepth 字段设置为 0 等同于非递归$ graphLookup 搜索阶段。

Memory

$ graphLookup 阶段必须保持在 100 兆字节的内存限制内。如果为 aggregate（）操作指定了 allowDiskUse：true，则$ graphLookup 阶段将忽略该选项。如果 aggregate（）操作中还有其他阶段，则 allowDiskUse：true 选项对这些其他阶段有效。

观点和整理

如果执行涉及多个视图的聚合（例如使用$ lookup 或$ graphLookup），则视图必须具有相同的排序规则。

E:

在单个集合中

名为 employees 的集合具有以下文档

{ "_id" : 1, "name" : "Dev" }
{ "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" }
{ "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }
{ "_id" : 4, "name" : "Andrew", "reportsTo" : "Eliot" }
{ "_id" : 5, "name" : "Asya", "reportsTo" : "Ron" }
{ "_id" : 6, "name" : "Dan", "reportsTo" : "Andrew" }

以下$ graphLookup 操作以递归方式匹配 employees 集合中的 reportsTo 和 name 字段，返回每个人的报告层次结构：reportsTo----name As reportingHierarchy

```
db.employees.aggregate( [
    {
        $graphLookup: {
            from: "employees",
            startWith: "$reportsTo",
            connectFromField: "reportsTo",
            connectToField: "name",
            as: "reportingHierarchy"
        }
    }
] )
```

Result:

```
{
    "_id" : 1,
    "name" : "Dev",
    "reportingHierarchy" : [ ]
}
{
    "_id" : 2,
    "name" : "Eliot",
    "reportsTo" : "Dev",
    "reportingHierarchy" : [
        { "_id" : 1, "name" : "Dev" }
```

```
    ]
}
{
    "_id" : 3,
    "name" : "Ron",
    "reportsTo" : "Eliot",
    "reportingHierarchy" : [
        { "_id" : 1, "name" : "Dev" },
        { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" }
    ]
}
{
    "_id" : 4,
    "name" : "Andrew",
    "reportsTo" : "Eliot",
    "reportingHierarchy" : [
        { "_id" : 1, "name" : "Dev" },
        { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" }
    ]
}
{
    "_id" : 5,
    "name" : "Asya",
    "reportsTo" : "Ron",
    "reportingHierarchy" : [
        { "_id" : 1, "name" : "Dev" },
        { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
        { "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }
    ]
}
{
    "_id" : 6,
    "name" : "Dan",
    "reportsTo" : "Andrew",
    "reportingHierarchy" : [
        { "_id" : 1, "name" : "Dev" },
        { "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" },
        { "_id" : 4, "name" : "Andrew", "reportsTo" : "Eliot" }
    ]
}
```


输出生成层次结构 Asya - > Ron - > Eliot - > Dev。

下表提供了文档{“_ id”的遍历路径：5，“name”：“Asya”，“reportsTo”：“Ron”}：

The following table provides a traversal path for the document `{ "_id" : 5, "name" : "Asya", "reportsTo" : "Ron" }`:

Start value	The `reportsTo` value of the document:
	copy `{ ... "reportsTo" : "Ron" }`
Depth 0	copy `{ "_id" : 3, "name" : "Ron", "reportsTo" : "Eliot" }`
Depth 1	copy `{ "_id" : 2, "name" : "Eliot", "reportsTo" : "Dev" }`
Depth 2	copy `{ "_id" : 1, "name" : "Dev" }`

1:0 -1 - 2 - 3
2:Dev
3:Eliot-Dev
4.Eliot-Dev
5.Ron-Eliot-Dev
6.Andrew-Eliot-Dev

跨多个集合
与$ lookup 一样，$ graphLookup 可以**访问同一数据库中的另一个集合。**
在以下示例中，数据库包含两个集合：
包含以下文件的集合机场：
airports:
{ "_id" : 0, "airport" : "JFK", "connects" : ["BOS", "ORD"] }
{ "_id" : 1, "airport" : "BOS", "connects" : ["JFK", "PWM"] }
{ "_id" : 2, "airport" : "ORD", "connects" : ["JFK"] }
{ "_id" : 3, "airport" : "PWM", "connects" : ["BOS", "LHR"] }
{ "_id" : 4, "airport" : "LHR", "connects" : ["PWM"] }
Travelers:
{ "_id" : 1, "name" : "Dev", "nearestAirport" : "JFK" }
{ "_id" : 2, "name" : "Eliot", "nearestAirport" : "JFK" }
{ "_id" : 3, "name" : "Jeff", "nearestAirport" : "BOS" }

对于旅行者集合中的每个文档，以下聚合操作在机场集合中查找 nearestAirport 值，并递归地将连接字段与机场字段匹配。该操作指定最大递归深度为 2。
E:
db.**travelers**.aggregate([

```
    {
        $graphLookup: {
            from: "airports",
            startWith: "$nearestAirport",
            connectFromField: "connects",
            connectToField: "airport",
            maxDepth: 2,
            depthField: "numConnections",
            as: "destinations"
        }
    }
] )
```

R:

```
{
    "_id" : 1,
    "name" : "Dev",
    "nearestAirport" : "JFK",
    "destinations" : [
        { "_id" : 3,
          "airport" : "PWM",
          "connects" : [ "BOS", "LHR" ],
          "numConnections" : NumberLong(2) },
        { "_id" : 2,
          "airport" : "ORD",
          "connects" : [ "JFK" ],
          "numConnections" : NumberLong(1) },
        { "_id" : 1,
          "airport" : "BOS",
          "connects" : [ "JFK", "PWM" ],
          "numConnections" : NumberLong(1) },
        { "_id" : 0,
          "airport" : "JFK",
          "connects" : [ "BOS", "ORD" ],
          "numConnections" : NumberLong(0) }
    ]
}
{
    "_id" : 2,
    "name" : "Eliot",
    "nearestAirport" : "JFK",
    "destinations" : [
        { "_id" : 3,
          "airport" : "PWM",
```

```
            "connects" : [ "BOS", "LHR" ],
            "numConnections" : NumberLong(2) },
          { "_id" : 2,
            "airport" : "ORD",
            "connects" : [ "JFK" ],
            "numConnections" : NumberLong(1) },
          { "_id" : 1,
            "airport" : "BOS",
            "connects" : [ "JFK", "PWM" ],
            "numConnections" : NumberLong(1) },
          { "_id" : 0,
            "airport" : "JFK",
            "connects" : [ "BOS", "ORD" ],
            "numConnections" : NumberLong(0) } ]
}
{
    "_id" : 3,
    "name" : "Jeff",
    "nearestAirport" : "BOS",
    "destinations" : [
        { "_id" : 2,
          "airport" : "ORD",
          "connects" : [ "JFK" ],
          "numConnections" : NumberLong(2) },
        { "_id" : 3,
          "airport" : "PWM",
          "connects" : [ "BOS", "LHR" ],
          "numConnections" : NumberLong(1) },
        { "_id" : 4,
          "airport" : "LHR",
          "connects" : [ "PWM" ],
          "numConnections" : NumberLong(2) },
        { "_id" : 0,
          "airport" : "JFK",
          "connects" : [ "BOS", "ORD" ],
          "numConnections" : NumberLong(1) },
        { "_id" : 1,
          "airport" : "BOS",
          "connects" : [ "JFK", "PWM" ],
          "numConnections" : NumberLong(0) }
    ]
}
```


下表提供了递归搜索的遍历路径，深度为 2，起始机场为 JFK：

使用查询过滤器

以下示例使用一个包含一组文档的集合，这些文档包含人员的姓名以及他们的朋友和他们的爱好的数组。聚合操作找到一个特定的人并遍历她的连接网络，以找到在他们的爱好中列出高尔夫的人。

名为 people 的集合包含以下文档：

```
{
   "_id" : 1,
   "name" : "Tanya Jordan",
   "friends" : [ "Shirley Soto", "Terry Hawkins", "Carole Hale" ],
   "hobbies" : [ "tennis", "unicycling", "golf" ]
}
{
   "_id" : 2,
   "name" : "Carole Hale",
   "friends" : [ "Joseph Dennis", "Tanya Jordan", "Terry Hawkins" ],
   "hobbies" : [ "archery", "golf", "woodworking" ]
}
{
   "_id" : 3,
   "name" : "Terry Hawkins",
   "friends" : [ "Tanya Jordan", "Carole Hale", "Angelo Ward" ],
   "hobbies" : [ "knitting", "frisbee" ]
}
{
   "_id" : 4,
   "name" : "Joseph Dennis",
   "friends" : [ "Angelo Ward", "Carole Hale" ],
   "hobbies" : [ "tennis", "golf", "topiary" ]
```

```
}
{
   "_id" : 5,
   "name" : "Angelo Ward",
   "friends" : [ "Terry Hawkins", "Shirley Soto", "Joseph Dennis" ],
   "hobbies" : [ "travel", "ceramics", "golf" ]
}
{
    "_id" : 6,
    "name" : "Shirley Soto",
    "friends" : [ "Angelo Ward", "Tanya Jordan", "Carole Hale" ],
    "hobbies" : [ "frisbee", "set theory" ]
 }
```

以下聚合操作使用三个阶段：

$ match 匹配包含字符串“Tanya Jordan”的名称字段的文档。返回一个输出文档。

$ graphLookup 将输出文档的 **friends** 字段与集合中其他文档的 **name** 字段连接起来，以遍历 **Tanya Jordan** 的连接网络。此阶段使用 **restrictSearchWithMatch** 参数仅查找 **hobbies** 数组包含 **golf** 的文档。返回一个输出文档。

$ project 对输出文档进行整形。打高尔夫球的联赛中列出的名字取自输入文件的高尔夫球手阵列中列出的文件的名称字段。

```
db.people.aggregate( [
   { $match: { "name": "Tanya Jordan" } },
   { $graphLookup: {
         from: "people",
         startWith: "$friends",
         connectFromField: "friends",
         connectToField: "name",
         as: "golfers",
         restrictSearchWithMatch: { "hobbies" : "golf" }
      }
   },
   { $project: {
         "name": 1,
         "friends": 1,
         "connections who play golf": "$golfers.name"
      }
   }
] )
RESULT:
{
    "_id" : 1,
    "name" : "Tanya Jordan",
    "friends" : [
        "Shirley Soto",
```

```
        "Terry Hawkins",
        "Carole Hale"
    ],
    "connections who play golf" : [
        "Joseph Dennis",
        "Tanya Jordan",
        "Angelo Ward",
        "Carole Hale"
    ]
}
```

$group:

按一些指定的表达式对文档进行分组，并为每个不同的分组输出到下一个阶段的文档。输出文档包含一个_id 字段，该字段按键包含不同的组。输出文档还可以包含计算字段，这些字段包含按$ group 的_id 字段分组的某些累加器表达式的值。 $ group 不会对其输出文档进行排序。

$ group 阶段具有以下原型形式：

{ $group: { _id: <expression>, <field1>: { <accumulator1> : <expression1> }, ... } }

id 字段是强制性的;但是，您可以指定_id 值 null 或任何其他常量值，以计算所有输入文档的**累计值**

其余的计算字段是可选的，并使用<accumulator>运算符计算。

_id 和<accumulator>表达式可以接受任何有效的表达式。有关表达式的更多信息，请参阅表达式。

注意事项

累加器操作员

<accumulator>运算符必须是以下累加器运算符之一：

名称　　说明

$ addToSet:返回每个组的唯一表达式值的数组。数组元素的顺序未定义。

$ avg:返回数值的平均值。忽略非数字值。

$ first:从每个组的第一个文档返回一个值。仅在文档按定义的顺序定义时才定义订单。

$ last:返回每个组的最后一个文档中的值。仅在文档按定义的顺序定义时才定义订单。

$ max:返回每个组的最高表达式值。

$ mergeObjects:返回通过组合每个组的输入文档创建的文档。

$ min:返回每个组的最低表达式值。

$ push:返回每个组的表达式值数组。

$ stdDevPop:返回输入值的总体标准差。

$ stdDevSamp:返回输入值的样本标准差。

$ sum:返回数值的总和。忽略非数字值。

$ group 运算符和内存

$ group 阶段的 RAM 限制为 100 兆字节。默认情况下，如果阶段超出此限制，$ group 将产生错误。但是，要允许处理大型数据集，请将 allowDiskUse 选项设置为 true 以启用$ group 操作以写入临时文件。有关详细信息，请参阅 db.collection.aggregate（）方法和 aggregate 命令。

在版本 2.6 中更改：MongoDB 为$ group 阶段引入了 100 兆字节的 RAM 限制，以及 allowDiskUse 选项来处理大型数据集的操作。

例子

计算计数，总和和平均值

鉴于集合销售与以下文件：

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
```

按月，日和年分组

以下聚合操作使用$ group 阶段按月，日和年对文档进行分组，并计算总价格和平均数量，并计算每个组的文档：

```
db.sales.aggregate(
    [
        {
          $group : {
             _id : { month: { $month: "$date" }, day: { $dayOfMonth: "$date" }, year: { $year: "$date" } },
             totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
             averageQuantity: { $avg: "$quantity" },
             count: { $sum: 1 }
          }
        }
    ]
)
```

R:

```
{ "_id" : { "month" : 3, "day" : 15, "year" : 2014 }, "totalPrice" : 50, "averageQuantity" : 10, "count" : 1 }
{ "_id" : { "month" : 4, "day" : 4, "year" : 2014 }, "totalPrice" : 200, "averageQuantity" : 15, "count" : 2 }
{ "_id" : { "month" : 3, "day" : 1, "year" : 2014 }, "totalPrice" : 40, "averageQuantity" : 1.5, "count" : 2 }
```

分组为空

以下聚合操作指定组_id 为 null，计算总价格和平均数量以及集合中所有文档的计数：

```
db.sales.aggregate(
    [
        {
          $group : {
```

```
                _id : null,
                totalPrice: { $sum: { $multiply: [ "$price", "$quantity" ] } },
                averageQuantity: { $avg: "$quantity" },
                count: { $sum: 1 }
            }
        }
    ]
)
```

R:

```
{ "_id" : null, "totalPrice" : 290, "averageQuantity" : 8.6, "count" : 5 }
```

检索不同的值

鉴于集合销售与以下文件：

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-03-15T09:00:00Z") }
{ "_id" : 4, "item" : "xyz", "price" : 5, "quantity" : 20, "date" : ISODate("2014-04-04T11:21:39.736Z") }
{ "_id" : 5, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-04-04T21:23:13.331Z") }
```

以下聚合操作使用$ group 阶段按项目对文档进行分组以检索不同的项目值：

必须要有_id

$group : { _id:""}

db.sales.aggregate([{ $group : { _id : "$item" } }])
R:
{ "_id" : "xyz" }
{ "_id" : "jkl" }
{ "_id" : "abc" }

枢轴数据
收藏书籍包含以下文件：
{ "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
{ "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
{ "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
{ "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
{ "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
按作者分组标题:
以下聚合操作将 title 集合中的数据转换为按作者分组的 books。
db.books.aggregate(
    [
      { $group : { _id : "$author", books: { $push: "$title" } } }
    ]
)
R:
{ "_id" : "Homer", "books" : ["The Odyssey", "Iliad"] }
{ "_id" : "Dante", "books" : ["The Banquet", "Divine Comedy", "Eclogues"] }

按作者分组文件
以下聚合操作使用$$ ROOT 系统变量按作者对文档进行分组。生成的文档不得超过 BSON 文档大小限制。
db.books.aggregate(
    [
      { $group : { _id : "$author", books: { $push: "$$ROOT" } } }
    ]
)
db.books.aggregate([{ $group : { _id : "$author", books: { $push: "$$ROOT" } } },{$limit:1}])
R:
{
  "_id" : "Homer",
  "books" :
      [
        { "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 },
        { "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
      ]

}

{
  "_id" : "Dante",
  "books" :
    [
      { "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 },
      { "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 },
      { "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
    ]
}

$indexStats:

返回有关集合的每个索引的使用的统计信息。如果使用访问控制运行，则用户必须具有包含 indexStats 操作的权限。

$ indexStats 阶段采用空文档，并具有以下语法：

{ $indexStats: { } }

out 文档包括以下字段：

Output Field　　Description

Name　　索引名称。

Key　　索引密钥规范。

host　　mongod 进程的主机名和端口。

Accesses　索引使用统计：ops 是使用索引的操作数。因为是 MongoDB 收集统计数据的时间。

索引的统计信息将在 mongod restart 或 index drop and recreation 上重置。

行为

accessses 字段报告的统计信息仅包括由用户请求驱动的索引访问。它不包括内部操作，如通过 TTL 索引删除或块拆分和迁移操作。

$ indexStats 必须是聚合管道中的第一个阶段。

事务中不允许使用$ indexStats。

Example:orders

{ "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2, "type": "apparel" }

{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "type": "electronics" }

{ "_id" : 3, "item" : "abc", "price" : 10, "quantity" : 5, "type": "apparel" }

在集合上创建以下两个索引：

db.orders.createIndex({ item: 1, quantity: 1 })

db.orders.createIndex({ type: 1, item: 1 })

对集合运行一些查询：

db.orders.find({ type: "apparel"})

db.orders.find({ item: "abc" }).sort({ quantity: 1 })

要查看订单集合上索引使用的统计信息，请运行以下聚合操作：

db.orders.aggregate([{ $indexStats: { } }])

R:

```
{
    "name" : "item_1_quantity_1",
    "key" : {
        "item" : 1,
        "quantity" : 1
    },
    "host" : "examplehost.local:27017",
    "accesses" : {
        "ops" : NumberLong(1),
        "since" : ISODate("2015-10-02T14:31:53.685Z")
    }
}
{
    "name" : "_id_",
    "key" : {
        "_id" : 1
    },
    "host" : "examplehost.local:27017",
    "accesses" : {
        "ops" : NumberLong(0),
        "since" : ISODate("2015-10-02T14:31:32.479Z")
    }
}
{
    "name" : "type_1_item_1",
    "key" : {
        "type" : 1,
        "item" : 1
    },
    "host" : "examplehost.local:27017",
    "accesses" : {
        "ops" : NumberLong(1),
        "since" : ISODate("2015-10-02T14:31:58.321Z")
    }
}
```


$limit

限制传递到管道中下一个阶段的文档数。

$ limit 阶段具有以下原型形式:

{ $limit: <positive integer> }

$ limit 采用一个正整数，指定要传递的最大文档数。

E:

```
db.article.aggregate(
    { $limit : 5 }
);
```

此操作仅返回管道传递给它的前 5 个文档。 $ limit 对其传递的文档内容没有影响。

注意

当$ sort 在$ limit 之前并且没有可以修改文档数量的中间阶段时，优化器可以将$ limit 合并到$ sort 中。这允许$ sort 操作仅在进展时保持前 n 个结果，其中 n 是指定的限制，并确保 MongoDB 只需要在内存中存储 n 个项目。当 allowDiskUse 为 true 且 n 个项超过聚合内存限制时，此优化仍适用。

$lookup

版本 3.2 中的新功能。

对同一数据库中的未整数集合执行左外连接，以从“已连接”集合中过滤文档以进行处理。对于每个输入文档，$ lookup 阶段添加一个新的数组字段，其元素是来自“已连接”集合的匹配文档。 $ lookup 阶段将这些重新整形的文档传递给下一个阶段。

$ lookup 阶段具有以下语法：

Equality Match

要在输入文档中的字段与“已连接”集合的文档中的字段之间执行相等匹配，$ lookup 阶段具有以下语法：

```
{
    $lookup:
      {
        from: <collection to join>,
        localField: <field from the input documents>,
        foreignField: <field from the documents of the "from" collection>,
        as: <output array field>
      }
}
```

$ lookup 采用包含以下字段的文档：

From:	指定同一数据库中的集合以执行连接。 from 集合无法分片。有关详细信息，请参阅 Sharded Collection Restrictions。
localField	指定输入到$ lookup 阶段的文档中的字段。 $ lookup 通过 from 集合的文档在 localField 上执行与 foreignField 的相等匹配。如果输入文档不包含 localField，则$ lookup 会将该字段视为具有值 null 以进行匹配。
foreignField	指定 from 集合中文档的字段。 $ lookup 在输入文档中对 foreignField 执行与 localField 的相等匹配。如果 from 集合中的文档不包含 foreignField，则$ lookup 将该值视为 null 以进行匹配。
As	指定要添加到输入文档的新数组字段的名称。新数组字段包含来自集合的匹配文档。如果输入文档中已存在指定的名称，则会覆盖现有字段。

该操作将对应于以下伪 SQL 语句：

```
SELECT *, <output array field>
FROM collection
WHERE <output array field> IN (SELECT *
                               FROM <collection to join>
                               WHERE <foreignField>= <collection.localField>);
```

请参阅以下示例：

使用$ lookup 执行单一等式连接

对数组使用$ lookup

对$ mergeObjects 使用$ lookup

连接条件和不相关的子查询¶

版本 3.6 中的新功能。

要在两个集合之间执行不相关的子查询以及允许除单个相等匹配之外的其他连接条件

$ lookup 阶段具有以下语法：

```
{
   $lookup:
     {
       from: <collection to join>,
       let: { <var_1>: <expression>, …, <var_n>: <expression> },
       pipeline: [ <pipeline to execute on the collection to join> ],
       as: <output array field>
     }
}
```

$ lookup 采用包含以下字段的文档：

From: 指定同一数据库中的集合以执行连接。 from 集合无法分片。有关详细信息，请参阅 Sharded Collection Restrictions。

Let: 可选的。指定要在管道字段阶段中使用的变量。使用变量表达式从输入到 $ lookup 阶段的文档中访问字段。

管道无法直接访问输入文档字段。相反，首先为输入文档字段定义变量，然后引用管道中各阶段的变量。

要访问管道中的 let 变量，请使用$ expr 运算符。

注意

let 变量可由管道中的各个阶段访问，包括嵌套在管道中的其他$ lookup 阶段。

Pipline: 指定要在已连接集合上运行的管道。管道从连接的集合中确定结果文档。要返回所有文档，请指定空管道[]。

管道无法直接访问输入文档字段。相反，首先为输入文档字段定量然后引用管道中各阶段的变量。要访问管道中的 let 变量，请使用 $ expr 运算符。

注意

let 变量可由管道中的各个阶段访问，包括嵌套在管道中的其他$ lookup 阶段。

As: 指定要添加到输入文档的新数组字段的名称。新数组字段包含来自集合的匹配文档。如果输入文档中已存在指定的名称，则会覆盖现有字段。

该操作将对应于以下伪 SQL 语句：

SELECT *, <output array field>

FROM collection

WHERE <output array field> IN (SELECT <documents as determined from the pipeline>

FROM <collection to join>

WHERE <pipeline>);

使用$ lookup 指定多个连接条件

不相关的子查询

观点和整理

如果执行涉及**多个视图的聚合**（例如使用$ lookup 或$ graphLookup），则**视图必须具有相同的排序规则。**

Sharded Collection 限制

在$ lookup 阶段，无法对 from 集合进行分片。但是，可以对运行 aggregate（）方法的集合进行分片。也就是说，在以下内容中：

```
db.collection.aggregate([
    { $lookup: { from: "fromCollection", ... } }
])
```

该集合可以分片。

fromCollection 无法分片。

因此，要将分片集合与未分片集合连接，您可以在分片集合上运行聚合并查找未整理的集合；例如。：

```
db.shardedCollection.aggregate([
    { $lookup: { from: "unshardedCollection", ... } }
])
```

E:

```
db.orders.insert([
    { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
    { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 },
    { "_id" : 3   }
])
```

使用以下文档创建另一个集合清单：

```
db.inventory.insert([
    { "_id" : 1, "sku" : "almonds", description: "product 1", "instock" : 120 },
    { "_id" : 2, "sku" : "bread", description: "product 2", "instock" : 80 },
    { "_id" : 3, "sku" : "cashews", description: "product 3", "instock" : 60 },
    { "_id" : 4, "sku" : "pecans", description: "product 4", "instock" : 70 },
    { "_id" : 5, "sku": null, description: "Incomplete" },
    { "_id" : 6 }
])
```

订单集合上的以下聚合操作使用 Order 集合中的字段项和 inventory 集合中的 sku 字段将订单中的文档与库存集合中的文档连接起来：

```
db.orders.aggregate([
    {
```

```
        $lookup:
          {
            from: "inventory",
            localField: "item",
            foreignField: "sku",
            as: "inventory_docs"
          }
    }
])
```

R:

```
{
    "_id" : 1,
    "item" : "almonds",
    "price" : 12,
    "quantity" : 2,
    "inventory_docs" : [
        { "_id" : 1, "sku" : "almonds", "description" : "product 1", "instock" : 120 }
    ]
}
{
    "_id" : 2,
    "item" : "pecans",
    "price" : 20,
    "quantity" : 1,
    "inventory_docs" : [
        { "_id" : 4, "sku" : "pecans", "description" : "product 4", "instock" : 70 }
    ]
}
{
    "_id" : 3,
    "inventory_docs" : [
        { "_id" : 5, "sku" : null, "description" : "Incomplete" },
        { "_id" : 6 }
    ]
}
```

该操作将对应于以下伪 SQL 语句：

```
SELECT *, inventory_docs
FROM orders
WHERE inventory_docs IN (SELECT *
FROM inventory
WHERE sku= orders.item);
```

对数组使用$ lookup

启动 MongoDB 3.4，如果 localField 是一个数组，则可以将数组元素与标量 foreignField 匹配，而不需要$ unwind 阶段。

例如，使用以下文档创建示例集合类：

```
db.classes.insert( [
    { _id: 1, title: "Reading is ...", enrollmentlist: [ "giraffe2", "pandabear", "artie" ], days:
["M", "W", "F"] },
    { _id: 2, title: "But Writing ...", enrollmentlist: [ "giraffe1", "artie" ], days: ["T", "F"] }
])
```

使用以下文档创建另一个集合成员：

```
db.members.insert( [
    { _id: 1, name: "artie", joined: new Date("2016-05-01"), status: "A" },
    { _id: 2, name: "giraffe", joined: new Date("2017-05-01"), status: "D" },
    { _id: 3, name: "giraffe1", joined: new Date("2017-10-01"), status: "A" },
    { _id: 4, name: "panda", joined: new Date("2018-10-11"), status: "A" },
    { _id: 5, name: "pandabear", joined: new Date("2018-12-01"), status: "A" },
    { _id: 6, name: "giraffe2", joined: new Date("2018-12-01"), status: "D" }
])
```

以下聚合操作将 classes 集合中的文档与 members 集合连接，在成员字段上与 name 字段匹配：

```
db.classes.aggregate([
    {
        $lookup:
            {
                from: "members",
                localField: "enrollmentlist",
                foreignField: "name",
                as: "enrollee_info"
           }{
    "_id" : 1,
    "title" : "Reading is ...",
    "enrollmentlist" : [ "giraffe2", "pandabear", "artie" ],
    "days" : [ "M", "W", "F" ],
    "enrollee_info" : [
        { "_id" : 1, "name" : "artie", "joined" : ISODate("2016-05-01T00:00:00Z"), "status" :
"A" },
        { "_id" : 5, "name" : "pandabear", "joined" : ISODate("2018-12-01T00:00:00Z"),
"status" : "A" },
        { "_id" : 6, "name" : "giraffe2", "joined" : ISODate("2018-12-01T00:00:00Z"),
"status" : "D" }
    ]
}
{
    "_id" : 2,
    "title" : "But Writing ...",
    "enrollmentlist" : [ "giraffe1", "artie" ],
    "days" : [ "T", "F" ],
```

```
    "enrollee_info" : [
        { "_id" : 1, "name" : "artie", "joined" : ISODate("2016-05-01T00:00:00Z"), "status" :
"A" },
        { "_id" : 3, "name" : "giraffe1", "joined" : ISODate("2017-10-01T00:00:00Z"),
"status" : "A" }
    ]
}
    }
])
```

R:

对$ mergeObjects 使用$ lookup

版本 3.6 中已更改：MongoDB 3.6 添加了$ mergeObjects 运算符，以将多个文档合并到一个文档中

使用以下文档创建集合订单：

```
db.orders.insert([
    { "_id" : 1, "item" : "almonds", "price" : 12, "quantity" : 2 },
    { "_id" : 2, "item" : "pecans", "price" : 20, "quantity" : 1 }
])
```

Another

```
db.items.insert([
   { "_id" : 1, "item" : "almonds", description: "almond clusters", "instock" : 120 },
   { "_id" : 2, "item" : "bread", description: "raisin and nut bread", "instock" : 80 },
   { "_id" : 3, "item" : "pecans", description: "candied pecans", "instock" : 60 }
])
```

以下操作首先使用$ lookup 阶段按项目字段连接两个集合，然后使用$ replaceRoot 中的$ mergeObjects 合并项目和订单中的已连接文档：

```
db.orders.aggregate([
    {
        $lookup: {
            from: "items",
            localField: "item",       // field in the orders collection
            foreignField: "item",    // field in the items collection
            as: "fromItems"
        }
    },
    {
        $replaceRoot: { newRoot: { $mergeObjects: [ { $arrayElemAt: [ "$fromItems", 0 ] },
"$$ROOT" ] } }
    },
    { $project: { fromItems: 0 } }
])
```

R:

```
{ "_id" : 1, "item" : "almonds", "description" : "almond clusters", "instock" : 120, "price" :
12, "quantity" : 2 }
```

{ "_id" : 2, "item" : "pecans", "description" : "candied pecans", "instock" : 60, "price" : 20, "quantity" : 1 }

使用$ lookup 指定多个连接条件

在版本 3.6 中更改：MongoDB 3.6 添加了对在已加入集合上执行管道的支持，这允许指定多个连接条件以及不相关的子查询。

使用以下文档创建集合订单：

```
db.orders.insert([
  { "_id" : 1, "item" : "almonds", "price" : 12, "ordered" : 2 },
  { "_id" : 2, "item" : "pecans", "price" : 20, "ordered" : 1 },
  { "_id" : 3, "item" : "cookies", "price" : 10, "ordered" : 60 }
])
```

使用以下文档创建另一个集合仓库：

```
db.warehouses.insert([
  { "_id" : 1, "stock_item" : "almonds", warehouse: "A", "instock" : 120 },
  { "_id" : 2, "stock_item" : "pecans", warehouse: "A", "instock" : 80 },
  { "_id" : 3, "stock_item" : "almonds", warehouse: "B", "instock" : 60 },
  { "_id" : 4, "stock_item" : "cookies", warehouse: "B", "instock" : 40 },
  { "_id" : 5, "stock_item" : "cookies", warehouse: "A", "instock" : 80 }
])
```

以下操作将 orders 集合与项目的 warehouses 集合以及库存数量是否足以涵盖订购数量相结合：

```
db.orders.aggregate([
    {
        $lookup:      管道不能直接访问输入文档字段。相反，首先为输入文档字段定义变量，
                      然后在管道中的各个阶段引用变量。管道要访问 let 变量，请使用$expr
                      操作符。
          {
              from: "warehouses",            1
              let: { order_item: "$item", order_qty: "$ordered" },        2 输入   orders 表
              pipeline: [                          3 管道不能包含$out 阶段或$merge 阶段。
                  { $match:
                      { $expr:
                          { $and:
                              [
                                { $eq: [ "$stock_item",   "$$order_item" ] },   比较 warehouse
的 stock_item
                                { $gte: [ "$instock", "$$order_qty" ] }
                              ]
                          }
                      }
                  },
                  { $project: { stock_item: 0, _id: 0 } }
              ],
```

```
            as: "stockdata"            4
        }
    }
])
```
R:
```
{ "_id" : 1, "item" : "almonds", "price" : 12, "ordered" : 2,
    "stockdata" : [ { "warehouse" : "A", "instock" : 120 }, { "warehouse" : "B", "instock" :
60 } ] }
{ "_id" : 2, "item" : "pecans", "price" : 20, "ordered" : 1,
    "stockdata" : [ { "warehouse" : "A", "instock" : 80 } ] }
{ "_id" : 3, "item" : "cookies", "price" : 10, "ordered" : 60,
    "stockdata" : [ { "warehouse" : "A", "instock" : 80 } ] }
```

Preudo-SQL:
```
SELECT *, stockdata
FROM orders
WHERE stockdata IN (SELECT warehouse, instock
                    FROM warehouses
                    WHERE stock_item= orders.item
                    AND instock >= orders.ordered );
```
不相关的子查询

在版本 3.6 中更改：MongoDB 3.6 添加了对在已加入集合上执行管道的支持，这允许指定多个连接条件以及不相关的子查询。

使用以下文档创建集合 absences：
```
db.absences.insert([
    { "_id" : 1, "student" : "Ann Aardvark", sickdays: [ new Date ("2018-05-01"),new Date
("2018-08-23") ] },
    { "_id" : 2, "student" : "Zoe Zebra", sickdays: [ new Date ("2018-02-01"),new Date
("2018-05-23") ] },
])
```
使用以下文档创建另一个集合 holidays：
```
db.holidays.insert([
    { "_id" : 1, year: 2018, name: "New Years", date: new Date("2018-01-01") },
    { "_id" : 2, year: 2018, name: "Pi Day", date: new Date("2018-03-14") },
    { "_id" : 3, year: 2018, name: "Ice Cream Day", date: new Date("2018-07-15") },
    { "_id" : 4, year: 2017, name: "New Years", date: new Date("2017-01-01") },
    { "_id" : 5, year: 2017, name: "Ice Cream Day", date: new Date("2017-07-16") }
])
```
以下操作将 absences 集合与 holidays 集合中的 2018 年假日信息相结合：
```
db.absences.aggregate([
    {
        $lookup:
            {
              from: "holidays",
```

```
            pipeline: [
                { $match: { year: 2018 } },
                { $project: { _id: 0, date: { name: "$name", date: "$date" } } },
                { $replaceRoot: { newRoot: "$date" } }
            ],
            as: "holidays"
        }
    }
])
```

查询月份为一的

```
db.holidays.aggregate([{$project:{math:{$month:"$date"}}},{$match:{math:1}

}])
```

R:

{ "_id" : 1, "student" : "Ann Aardvark", "sickdays" : [ISODate("2018-05-01T00:00:00Z"), ISODate("2018-08-23T00:00:00Z")],
"holidays" : [{ "name" : "New Years", "date" : ISODate("2018-01-01T00:00:00Z") }, { "name" : "Pi Day", "date" : ISODate("2018-03-14T00:00:00Z") }, { "name" : "Ice Cream Day", "date" : ISODate("2018-07-15T00:00:00Z") }] }
{ "_id" : 2, "student" : "Zoe Zebra", "sickdays" : [ISODate("2018-02-01T00:00:00Z"), ISODate("2018-05-23T00:00:00Z")],
"holidays" : [{ "name" : "New Years", "date" : ISODate("2018-01-01T00:00:00Z") }, { "name" : "Pi Day", "date" : ISODate("2018-03-14T00:00:00Z") }, { "name" : "Ice Cream Day", "date" : ISODate("2018-07-15T00:00:00Z") }] }

Preudo-sql:

```
SELECT *, holidays
FROM absences
WHERE holidays IN (SELECT name, date
                   FROM holidays
                   WHERE year = 2018);
```

$listLocalSessions

版本 3.6 中的新功能。

列出 mongod 或 mongos 实例在内存中缓存的会话。

重要:

当用户在 mongod 或 mongos 实例上创建会话时, 会话的记录最初只存在于实例的内存中; 即记录是实例的本地记录。实例会定期将其缓存的会话同步到 config 数据库中的 system.sessions 集合，此时，$ listSessions 和部署的所有成员都可以看到它们。在 system.sessions 集合中存在会话记录之前，您只能通过$ listLocalSessions 操作列出会

话。

$ listLocalSessions 操作使用 db.aggregate () 方法而不是 db.collection.aggregate ()。要运行$ listLocalSessions，它必须是管道中的第一个阶段。
该阶段具有以下语法：
{ $listLocalSessions: <document> }

{}	如果使用访问控制运行，则返回当前已验证用户的所有会话。如果运行时没有访问控制，则返回所有会话。
{users：[{user：<user>，db：<db>}，...]}	返回指定用户的所有会话。如果使用访问控制运行，则经过身份验证的用户必须具有群集上 listSession 操作的权限才能列出其他用户的会话。
{allUsers：true}	返回所有用户的所有会话。如果使用访问控制运行，则经过身份验证的用户必须具有对群集执行 listSession 操作的权限。

限制
事务中不允许使用$ listLocalSessions。
E:
列出所有本地会话
从连接的 mongod / mongos 实例的会话内存缓存中，以下聚合操作列出了所有会话：
如果使用访问控制运行，则当前用户必须具有对群集执行 listSession 操作的权限。
db.aggregate([{ $listLocalSessions: { allUsers: true } }])
列出指定用户的所有本地会话
从连接的 mongod / mongos 实例的内存缓存中，以下聚合操作列出了指定用户 myAppReader @ test 的所有会话：
注意
如果使用访问控制运行且当前用户不是指定用户，则当前用户必须具有对群集执行 listSession 操作的权限。
db.aggregate([{ $listLocalSessions: { users: [{ user: "myAppReader", db: "test" }] } }])
列出当前用户的所有本地会话
从连接的 mongod / mongos 实例的内存缓存中，如果使用访问控制运行，以下聚合操作将列出当前用户的所有会话：
db.aggregate([{ $listLocalSessions: { } }])
如果在没有访问控制的情况下运行，则该操作将列出所有 session。

$listSessions

版本 3.6 中的新功能。

列出存储在 config 数据库中 system.sessions 集合中的所有会话。 MongoDB 部署的所有成员都可以看到这些会话。

重要

当用户在 mongod 或 mongos 实例上创建会话时，会话的记录最初只存在于实例的内存中；即记录是实例的本地记录。实例会定期将其缓存的会话同步到 config 数据库中的 system.sessions 集合，此时，$ listSessions 和部署的所有成员都可以看到它们。在 system.sessions 集合中存在会话记录之前，您只能通过$ listLocalSessions 操作列出会话。

要运行$ listSessions，它必须是管道中的第一个阶段。

该阶段具有以下语法：

{ $listSessions: <document> }

$ listSessions 阶段采用包含以下内容之一的文档：

{}	如果使用访问控制运行，则返回当前已验证用户的所有会话。如果运行时没有访问控制，则返回所有会话。
{users：[{user：<user>，db：<db>}，...]}	返回指定用户的所有会话。如果使用访问控制运行，则经过身份验证的用户必须具有群集上 listSession 操作的权限才能列出其他用户的会话。
{allUsers：true}	返回所有用户的所有会话。如果使用访问控制运行，则经过身份验证的用户必须具有对群集执行 listSession 操作的权限。

事务中不允许$ listSessions。

E:

列出所有会话

在 system.sessions 集合中，以下聚合操作列出了所有会话：

注意

如果使用访问控制运行，则当前用户必须具有对群集执行 listSession 操作的权限。

```
use config
db.system.sessions.aggregate( [   { $listSessions: { allUsers: true } } ] )
```

列出指定用户的所有会话

在 system.sessions 集合中，以下聚合操作列出了指定用户 myAppReader @ test 的所有会话：

注意

如果使用访问控制运行且当前用户不是指定用户，则当前用户必须具有对群集执行 listSession 操作的权限。

```
use config
db.system.sessions.aggregate( [ { $listSessions: { users: [ {user: "myAppReader", db: "test" } ] } } ] )
```

列出当前用户的所有会话

在 system.sessions 集合中，如果使用访问控制运行，以下聚合操作将列出当前用户的所有会话：

use config

db.system.sessions.aggregate([{ $listSessions: { } }])

$match:

筛选文档，只将匹配指定条件的文档传递到下一个管道阶段。

$match 阶段的原型表单如下:

{ $match: { <query> } }

$match 接受一个指定查询条件的文档。**查询语法与读取操作查询语法相同**;例如$match **不接受原始聚合表达式**。相反，使用**$expr 查询表达式在$match** 中包含聚合表达式。

pipeline 优化:

将$match 尽可能早地放在聚合管道中。因为$match 限制聚合管道中的文档总数，所以早期的$match 操作将管道下的处理量最小化。

如果在管道的最开始放置$match，查询可以像其他 db.collection.find() 或 db.collection.findOne()那样利用索引。

Restrictions

$match 查询语法与 read 操作查询语法相同;例如$match 不接受原始聚合表达式。若要在 $match 中包含聚合表达式，请**使用$expr 查询表达式:**

{ $match: { $expr: { <aggregation expression> } } }

不能将$where in $match 查询用作聚合管道的一部分。

不能在$match 查询中使用$near 或$near sphere 作为聚合管道的一部分。另一种选择是: 使用$geoNear stage 而不是$match stage。

在$match 阶段使用$geoWithin 查询操作符和$center 或$centerSphere。

要在$match 阶段中使用$text， $match 阶段必须是管道的第一阶段。

视图不支持文本搜索。

例子

示例使用一个名为 articles 的集合，其中包含以下文档:

{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }

{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }

{ "_id" : ObjectId("55f5a192d4bede9ac365b257"), "author" : "ahn", "score" : 60, "views" : 1000 }

{ "_id" : ObjectId("55f5a192d4bede9ac365b258"), "author" : "li", "score" : 55, "views" : 5000 }

{ "_id" : ObjectId("55f5a1d3d4bede9ac365b259"), "author" : "annT", "score" : 60, "views" : 50 }

{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25a"), "author" : "li", "score" : 94, "views" : 999 }

{ "_id" : ObjectId("55f5a1d3d4bede9ac365b25b"), "author" : "ty", "score" : 95, "views" : 1000 }
等量 Match：
下面的操作使用$match 执行一个简单的等式匹配:
db.articles.aggregate(
[{ $match : { author : "dave" } }]
);
$match 选择 author 字段等于 dave 的文档，聚合返回以下内容:
{ "_id" : ObjectId("512bc95fe835e68f199c8686"), "author" : "dave", "score" : 80, "views" : 100 }
{ "_id" : ObjectId("512bc962e835e68f199c8687"), "author" : "dave", "score" : 85, "views" : 521 }
执行一个 count
下面的示例选择要使用$match 管道操作符处理的文档，然后将结果管道到$group 管道操作符，以计算文档的数量:
db.articles.aggregate([
{ $match: { $or: [{ score: { $gt: 70, $lt: 90 } }, { views: { $gte: 1000 } }] } },
{ $group: { _id: null, count: { $sum: 1 } } }
]);

在聚合管道中，$match 选择得分大于 70 和小于 90，或者视图大于或等于 1000 的文档。然后将这些文档通过管道传输到$组来执行计数。聚合返回以下内容:
{ "_id" : null, "count" : 5 }

$merge

新版本 4.2。
将聚合管道的结果写入指定的集合。$merge 操作符必须是管道中的最后一个阶段。
语法：

{ $merge: {
into: <collection> -or- { db: <db>, coll: <collection> },
on: <identifier field> -or- [<identifier field1>, ...], // Optional
let: <variables>, // Optional
whenMatched: <replace|keepExisting|merge|fail|pipeline>, // Optional
whenNotMatched: <insert|discard|fail> // Optional
} }
E:
{ $merge: { into: "myOutput", on: "_id", whenMatched: "replace", whenNotMatched: "insert" } }

例子
按需物化视图:初始创建
如果输出集合不存在，$merge 将创建该集合。

例如，zoo 数据库中名为 salary 的集合填充了员工工资和部门历史:

```
db.getSiblingDB("zoo").salaries.insertMany([
    { "_id" : 1, employee: "Ant", dept: "A", salary: 100000, fiscal_year: 2017 },
    { "_id" : 2, employee: "Bee", dept: "A", salary: 120000, fiscal_year: 2017 },
    { "_id" : 3, employee: "Cat", dept: "Z", salary: 115000, fiscal_year: 2017 },
    { "_id" : 4, employee: "Ant", dept: "A", salary: 115000, fiscal_year: 2018 },
    { "_id" : 5, employee: "Bee", dept: "Z", salary: 145000, fiscal_year: 2018 },
    { "_id" : 6, employee: "Cat", dept: "Z", salary: 135000, fiscal_year: 2018 },
    { "_id" : 7, employee: "Gecko", dept: "A", salary: 100000, fiscal_year: 2018 },
    { "_id" : 8, employee: "Ant", dept: "A", salary: 125000, fiscal_year: 2019 },
    { "_id" : 9, employee: "Bee", dept: "Z", salary: 160000, fiscal_year: 2019 },
    { "_id" : 10, employee: "Cat", dept: "Z", salary: 150000, fiscal_year: 2019 }
])
```

您可以使用$group 和$merge 两个阶段，从薪金集合中当前的数据创建一个名为 budget 的集合(在报表数据库中):

请注意

对于复制集或独立部署，如果不存在输出数据库，$merge 也将创建数据库。

对于分片集群部署，指定的输出数据库必须已经存在。

```
db.getSiblingDB("zoo").salaries.aggregate( [
    { $group: { _id: { fiscal_year: "$fiscal_year", dept: "$dept" }, salaries: { $sum: "$salary" } } },
    { $merge : { into: { db: "reporting", coll: "budgets" }, on: "_id",    whenMatched: "replace", whenNotMatched: "insert" } }
] )
```

$group stage 按财政年度和部门对工资进行分组。

$merge 阶段将前面$group 阶段的输出写入报表数据库中的 budget 集合。

查阅新预算册内的文件:

```
db.getSiblingDB("reporting").budgets.find().sort( { _id: 1 } )
```

预算汇编包括下列文件:

R:

```
{ "_id" : { "fiscal_year" : 2017, "dept" : "A" }, "salaries" : 220000 }
{ "_id" : { "fiscal_year" : 2017, "dept" : "Z" }, "salaries" : 115000 }
{ "_id" : { "fiscal_year" : 2018, "dept" : "A" }, "salaries" : 215000 }
{ "_id" : { "fiscal_year" : 2018, "dept" : "Z" }, "salaries" : 280000 }
{ "_id" : { "fiscal_year" : 2019, "dept" : "A" }, "salaries" : 125000 }
{ "_id" : { "fiscal_year" : 2019, "dept" : "Z" }, "salaries" : 310000 }
```

$out:

New in version 2.6.

获取聚合管道返回的文档，并将其写入指定的集合。**$out 操作符必须是管道中的最后一个阶段。**

版本 3.2.0 中的更改:MongoDB 3.2 增加了对文档验证的支持。bypassDocumentValidation

字段允许您在聚合操作的$out 阶段绕过文档验证。这允许插入不满足验证要求的文档。将 bypassDocumentValidation 指定为聚合方法或命令上的选项。

$out 阶段的原型形式如下:

{ $out: "<output-collection>" }

Important:

不能将切分集合指定为输出集合。可以对管道的输入集合进行分片。

$out 操作符不能将结果写入有上限的集合。

创建新的集合

$out 操作在当前数据库中**创建一个新的集合**(如果一个集合还不存在)。在聚合完成之前，集合是不可见的。如果**聚合失败，MongoDB 将不创建集合**。

取代现有的集合

如果$out 操作指**定的集合已经存在**，那么在聚合完成之后，**$out 阶段将原子性地用新的结果集合替换现有的集合**。具体来说

$out 操作:

1 创建一个临时集合。

2 将索引从现有集合复制到临时集合。

3 将文档插入到 temp 集合中。

4 调用 db.collection。使用 dropTarget 重命名一个集合:true 将临时集合重命名为目标集合。

$out 操作不会更改上一个集合上存在的任何索引。如果聚合失败，$out 操作不更改已存在的集合。

会生成一个新的集合，实际存在的。

Index Constraints

如果管道生成的文档违反任何惟一索引，包括原始输出集合_id 字段上的索引，则管道将无法完成。

Transactions

Transactions 中不允许使用$out。

例子

A collection books 包含以下文件:

```
{ "_id" : 8751, "title" : "The Banquet", "author" : "Dante", "copies" : 2 }
{ "_id" : 8752, "title" : "Divine Comedy", "author" : "Dante", "copies" : 1 }
{ "_id" : 8645, "title" : "Eclogues", "author" : "Dante", "copies" : 2 }
{ "_id" : 7000, "title" : "The Odyssey", "author" : "Homer", "copies" : 10 }
{ "_id" : 7020, "title" : "Iliad", "author" : "Homer", "copies" : 10 }
```

下面的聚合操作以**图书集合中的数据为轴心**，使其具有**按作者分组的标题**，然后将结果写入 authors 集合。

```
db.books.aggregate( [
                        { $group : { _id : "$author", books: { $push: "$title" } } },
                        { $out : "authors" }
                    ] )
```

操作完成后，authors collection 包含以下文档:

```
{ "_id" : "Homer", "books" : [ "The Odyssey", "Iliad" ] }
{ "_id" : "Dante", "books" : [ "The Banquet", "Divine Comedy", "Eclogues" ] }
```

$planCacheStats

Can be find in document 4.2
返回集合的计划缓存信息。该阶段为每个计划缓存条目返回一个文档。
$planCacheStats 阶段必须是管道中的第一个阶段。该阶段以一个空文档为参数，语法如下:
{ $planCacheStats: { } }
E:orders collection
db.orders.insert([
{ "_id" : 1, "item" : "abc", "price" : NumberDecimal("12"), "quantity" : 2, "type": "apparel" },
{ "_id" : 2, "item" : "jkl", "price" : NumberDecimal("20"), "quantity" : 1, "type": "electronics" },
{ "_id" : 3, "item" : "abc", "price" : NumberDecimal("10"), "quantity" : 5, "type": "apparel" },
{ "_id" : 4, "item" : "abc", "price" : NumberDecimal("8"), "quantity" : 10, "type": "apparel" },
{ "_id" : 5, "item" : "jkl", "price" : NumberDecimal("15"), "quantity" : 15, "type": "electronics" }
])
Create index on this collection
db.orders.createIndex({ item: 1 });
db.orders.createIndex({ item: 1, quantity: 1 });
db.orders.createIndex({ item: 1, price: 1 }, { partialFilterExpression: { price: { $gte: NumberDecimal("10")} } });
db.orders.createIndex({ quantity: 1 });
db.orders.createIndex({ quantity: 1, type: 1 });
Index {item: 1, price: 1} 是一个部分索引，仅索引 price 字段大于或等于 NumberDecimal(“10”)的文档。
返回查询。
db.orders.find({ item: "abc", price: { $gte: NumberDecimal("10") } })
db.orders.find({ item: "abc", price: { $gte: NumberDecimal("5") } })
db.orders.find({ quantity: { $gte: 20 } })
db.orders.find({ quantity: { $gte: 5 }, type: "apparel" })
返回查询缓存中所有条目的信息
下面的聚合管道使用$planCacheStats 返回关于集合的计划缓存条目的信息:
db.orders.aggregate([
{ $planCacheStats: { } }
])
R:
{ // Plan Cache Entry 1
"createdFromQuery" : {
"query" : { "quantity" : { "$gte" : 5 }, "type" : "apparel" },
"sort" : { },
"projection" : { }

```
    },
    "queryHash" : "4D151C4C",
    "planCacheKey" : "DD67E353",
    "isActive" : true,
    "works" : NumberLong(4),
    "cachedPlan" : {
        ...
    },
    "timeOfCreation" : ISODate("2019-02-04T20:30:10.414Z"),
    "creationExecStats" : [
        {
            ... // Exec Stats for Candidate 1
        },
        {
            ... // Exec Stats for Candidate 2
        }
    ],
    "candidatePlanScores" : [
        1.5003000000000002,
        1.5003000000000002
    ],
    "indexFilterSet" : false
}
{                                                                   // Plan Cache Entry 2
    "createdFromQuery" : {
        "query" : { "quantity" : { "$gte" : 20 } },
        "sort" : { },
        "projection" : { }
    },
    "queryHash" : "23B19B75",
    "planCacheKey" : "6F23F858",
    "isActive" : true,
    "works" : NumberLong(1),
    "cachedPlan" : {
        ...
    },
    "timeOfCreation" : ISODate("2019-02-04T20:30:10.412Z"),
    "creationExecStats" : [
        {
            ... // Exec Stats for Candidate 1
        },
        {
            ... // Exec Stats for Candidate 2
        }
```

```
    ],
    "candidatePlanScores" : [
        1.0003000000000002,
        1.0003000000000002
    ],
    "indexFilterSet" : false
}
{                                                                   // Plan Cache Entry 3
    "createdFromQuery" : {
        "query" : { "item" : "abc", "price" : { "$gte" : NumberDecimal("5") } },
        "sort" : { },
        "projection" : { }
    },
    "queryHash" : "117A6B10",
    "planCacheKey" : "A1824628",
    "isActive" : true,
    "works" : NumberLong(4),
    "cachedPlan" : {
        ...
    },
    "timeOfCreation" : ISODate("2019-02-04T20:30:10.410Z"),
    "creationExecStats" : [
        {
            ... // Exec Stats for Candidate 1
        },
        {
            ... // Exec Stats for Candidate 2
        }
    ],
    "candidatePlanScores" : [
        1.7503000000000002,
        1.7503000000000002
    ],
    "indexFilterSet" : false
}
{                                                                   // Plan Cache Entry 4
    "createdFromQuery" : {
        "query" : { "item" : "abc", "price" : { "$gte" : NumberDecimal("10") } },
        "sort" : { },
        "projection" : { }
    },
    "queryHash" : "117A6B10",
    "planCacheKey" : "2E6E536B",
    "isActive" : true,
```

```
    "works" : NumberLong(3),
    "cachedPlan" : {
        ...
    },
    "timeOfCreation" : ISODate("2019-02-04T20:30:10.408Z"),
    "creationExecStats" : [
        {
            ... // Exec Stats for Candidate 1
        },
        {
            ... // Exec Stats for Candidate 2
        },
        {
            ... // Exec Stats for Candidate 3
        }
    ],
    "candidatePlanScores" : [
        1.6669666666666663,
        1.6669666666666665,
        1.6669666666666665
    ],
    "indexFilterSet" : false
}
```

$project:

将带有请求字段的文档传递到管道中的下一阶段。指定的字段可以是输入文档中的现有字段，也可以是新计算的字段。

$project 阶段的原型形式如下:

{ $project: { <specification(s)> } }

$project 接受一个文档，该文档可以指定字段的包含、_id 字段的抑制、新字段的添加和现有字段值的重置。或者，您可以指定字段的排除。

$项目规格书的格式如下:

<field>:　　<1 or true>指定包含一个字段。

_id:　　<0 或 false>

　　指定对_id 字段的抑制。

　　要有条件地排除字段，可以使用 REMOVE 变量。有关详细信息，请参见有条件地排除字段。

<filed>:<表达式>

　　添加新字段或重置现有字段的值。

　　版本 3.6 中更改:MongoDB 3.6 添加了变量 REMOVE。如果表达式的计算值为$$REMOVE，则该字段将被排除在输出中。有关详细信息，请参见有条件地排除字段。

<字段>:< 0 或 false>:

新版本 3.4。
指定字段的排除。
要有条件地排除字段，可以使用 REMOVE 变量。有关详细信息,请参见有条件地排除字段。
如果指定排除_id 以外的字段，则不能使用任何其他$project 规范表单。此限制不适用于使用 REMOVE 变量有条件地排除字段。

注意事项
包括现有油田
默认情况下，_id 字段包含在输出文档中。要在输出文档中包含来自输入文档的任何其他字段，必须显式地指定包含在$project 中的内容。
如果指定包含文档中不存在的字段，$project 将忽略该字段包含，并且不将该字段添加到文档中。
禁止_id 字段
默认情况下，_id 字段包含在输出文档中。要从输出文档中排除_id 字段，必须在$project 中显式指定对_id 字段的抑制。

有条件地排除字段
新版本 3.6。
从 MongoDB 3.6 开始，可以使用聚合表达式中的变量 REMOVE 有条件地抑制字段。例如，请参见条件排除字段。
添加新字段或重置现有字段
MongoDB 还提供了$addFields 来向文档添加新字段。
若要添加新字段或重置现有字段的值，请指定字段名称并将其值设置为某个表达式。有关表达式的更多信息，请参见表达式。
文字值
要将字段值直接设置为数值或布尔文字，而不是将字段设置为解析为文字的表达式，请使用$literal 操作符。否则，$project 将 numeric 或 boolean 文字作为包含或排除字段的标志。
场重命名
通过指定新字段并将其值设置为现有字段的字段路径，可以有效地重命名字段。
新数组字段
从 MongoDB 3.2 开始，$project stage 支持使用方括号[]直接创建新的数组字段。如果数组规范包含文档中不存在的字段，则操作将 null 替换为该字段的值。例如，请参见 Project New Array 字段。
嵌入式文档字段
在嵌入文档中投影或添加/重置字段时，可以使用点符号，如 in
"contact.address.country": <1 or 0 or expression>
或者：
contact: { address: { country: <1 or 0 or expression> } }
嵌套字段时，不能在嵌入的文档中使用点符号来指定字段，例如:contact: {"address "。: <1 或 0 或表达式>}无效。
Restrictions
在 3.4 版本中进行了更改。
如果$project 规范是一个空文档，则稍后会产生一个错误。
例子

在输出文档中包含特定的字段
下面的$project 阶段不包含_id 字段，但包含 title，以及输出文档中的 author 字段:
db.books.aggregate([{ $project : { _id: 0, title : 1 , author : 1 } }])
R:
{ "title" : "abc123", "author" : { "last" : "zzz", "first" : "aaa" } }
从输出文档中将字段排除
新版本 3.4。
考虑以下列文件收集书籍:

```
{
   "_id" : 1,
   title: "abc123",
   isbn: "0001122223334",
   author: { last: "zzz", first: "aaa" },
   copies: 5,
   lastModified: "2016-07-28"
}
```

下面的$project 阶段将 lastModified 字段从输出中排除:
db.books.aggregate([{ $project : { "lastModified": 0 } }])
排除嵌入文档中的字段
新版本 3.4。
考虑以下列文件收集书籍:

```
{
   "_id" : 1,
   title: "abc123",
   isbn: "0001122223334",
   author: { last: "zzz", first: "aaa" },
   copies: 5,
   lastModified: "2016-07-28"
}
```

下面的$project 阶段不包括作者。第一个和最后一个修改字段的输出:
db.books.aggregate([{ $project : { "author.first" : 0, "lastModified" : 0 } }])
或者，您可以将排除规范嵌套在文档中:
db.bookmarks.aggregate([{ $project: { "author": { "first": 0}, "lastModified" : 0 } }])
这两种规格的结果是相同的输出

```
{
    "_id" : 1,
    "title" : "abc123",
    "isbn" : "0001122223334",
    "author" : {
        "last" : "zzz"
    },
    "copies" : 5,
}:
```

有条件地排除字段

新版本 3.6。
从 MongoDB 3.6 开始，可以使用聚合表达式中的变量 REMOVE 有条件地抑制字段。
考虑以下列文件收集书籍:

```
{
   "_id" : 1,
   title: "abc123",
   isbn: "0001122223334",
   author: { last: "zzz", first: "aaa" },
   copies: 5,
   lastModified: "2016-07-28"
}
{
   "_id" : 2,
   title: "Baked Goods",
   isbn: "9999999999999",
   author: { last: "xyz", first: "abc", middle: "" },
   copies: 2,
   lastModified: "2017-07-21"
}
{
   "_id" : 3,
   title: "Ice Cream Cakes",
   isbn: "8888888888888",
   author: { last: "xyz", first: "abc", middle: "mmm" },
   copies: 5,
   lastModified: "2017-07-22"
}
```

下面的$project 阶段使用 REMOVE 变量来排除作者。只有当它等于""时，中间字段:

```
db.books.aggregate( [
    {
        $project: {
            title: 1,
            "author.first": 1,
            "author.last" : 1,
            "author.middle": {
                $cond: {
                    if: { $eq: [ "", "$author.middle" ] },
                    then: "$$REMOVE",
                    else: "$author.middle"
                }
            }
        }
    }
] )
```

R:
{ "_id" : 1, "title" : "abc123", "author" : { "last" : "zzz", "first" : "aaa" } }
{ "_id" : 2, "title" : "Baked Goods", "author" : { "last" : "xyz", "first" : "abc" } }
{ "_id" : 3, "title" : "Ice Cream Cakes", "author" : { "last" : "xyz", "first" : "abc", "middle" : "mmm" } }
包含嵌入文档中的特定字段
考虑收藏下列文件的书签:
{ _id: 1, user: "1234", stop: { title: "book1", author: "xyz", page: 32 } }
{ _id: 2, user: "7890", stop: [{ title: "book2", author: "abc", page: 5 }, { title: "book3", author: "ijk", page: 100 }] }
要在 stop 字段中只包含嵌入文档中的 title 字段，可以使用点符号:
db.bookmarks.aggregate([{ $project: { "stop.title": 1 } }])
或者，您可以将包含规范嵌套在文档中:
db.bookmarks.aggregate([{ $project: { stop: { title: 1 } } }])
两种规格都产生了以下文件:
{ "_id" : 1, "stop" : { "title" : "book1" } }
{ "_id" : 2, "stop" : [{ "title" : "book2" }, { "title" : "book3" }] }
包括计算字段
考虑以下列文件收集书籍:

```
{
   "_id" : 1,
   title: "abc123",
   isbn: "0001122223334",
   author: { last: "zzz", first: "aaa" },
   copies: 5
}
```

下面的$project 阶段添加了新的字段 isbn、lastName 和 copiesSold:

```
db.books.aggregate(
    [
        {
            $project: {
                title: 1,
                isbn: {
                    prefix: { $substr: [ "$isbn", 0, 3 ] },
                    group: { $substr: [ "$isbn", 3, 2 ] },
                    publisher: { $substr: [ "$isbn", 5, 4 ] },
                    title: { $substr: [ "$isbn", 9, 3 ] },
                    checkDigit: { $substr: [ "$isbn", 12, 1] }
                },
                lastName: "$author.last",
                copiesSold: "$copies"
            }
        }
    ]
```

)
R:

```
{
    "_id" : 1,
    "title" : "abc123",
    "isbn" : {
        "prefix" : "000",
        "group" : "11",
        "publisher" : "2222",
        "title" : "333",
        "checkDigit" : "4"
    },
    "lastName" : "zzz",
    "copiesSold" : 5
}
```

项目新数组字段
例如，如果一个集合包含以下文档:

```
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "x" : 1, "y" : 1 }
```

下面的操作将字段 x 和 y 投影为一个新的字段 myArray 中的元素:

```
db.collection.aggregate( [ { $project: { myArray: [ "$x", "$y" ] } } ] )
```

R:

```
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "myArray" : [ 1, 1 ] }
```

如果数组规范包含文档中不存在的字段，则操作将 null 替换为该字段的值。
例如，给定与上面相同的文档，下面的操作将字段 x、y 和一个不存在的字段$someField 作为新字段 myArray 中的元素进行投影:

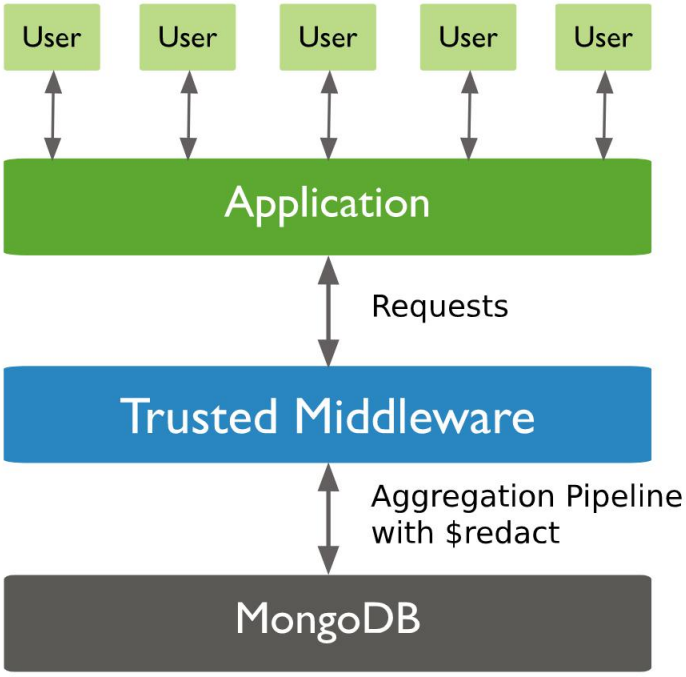
```
db.collection.aggregate( [ { $project: { myArray: [ "$x", "$y", "$someField" ] } } ] )
```

R:

```
{ "_id" : ObjectId("55ad167f320c6be244eb3b95"), "myArray" : [ 1, 1, null ] }
```


$redact：

出现在新版本 2.6 中。
根据存储在文档本身中的信息限制文档的内容。



Folloing prototype form:

{ $redact: <expression> }

参数**可以是任何有效的表达式**，只要它解析为$$ descent、$$PRUNE 或$$KEEP 系统变量。有关表达式的更多信息，请参见表达式。

$redact	返回当前文档级别的字段，不包括嵌入的文档。要将嵌入式文档和嵌入式文档包含在数组中，请将$cond 表达式应用于嵌入式文档，以确定这些嵌入式文档的访问权限。
$$prune	$redact 排除了当前文档/嵌入文档级别的所有字段，没有进一步检查任何被排除的字段。即使被排除的字段包含可能具有不同访问级别的嵌入式文档，这也适用。
$$KEEP	$redact 返回值，或者保持当前文档/嵌入文档级别的所有字段，而不需要在此级别进一步检查字段。即使 include 字段包含可能具有不同访问级别的嵌入式文档，这也适用。

E:

本节中的示例使用了 mongo shell 2.6 版本中提供的 db.collection.aggregate() helper。

评估每个文档级别的访问

forecast 集合包含以下格式的文档，其中 tags 字段列出该文档/嵌入文档级别的不同访问值；即["G"， "STLW"]的值指定"G"或"STLW"中的任何一个都可以访问数据：

```
{
  _id: 1,
  title: "123 Department Report",
  tags: [ "G", "STLW" ],
  year: 2014,
```

```
  subsections: [
    {
      subtitle: "Section 1: Overview",
      tags: [ "SI", "G" ],
      content:  "Section 1: This is the content of section 1."
    },
    {
      subtitle: "Section 2: Analysis",
      tags: [ "STLW" ],
      content: "Section 2: This is the content of section 2."
    },
    {
      subtitle: "Section 3: Budgeting",
      tags: [ "TK" ],
      content: {
        text: "Section 3: This is the content of section3.",
        tags: [ "HCS" ]
      }
    }
  ]
}
```

用户可以使用标签“STLW”或“G”查看信息。要对该用户 2014 年的所有文档运行查询，包括$redact 阶段，如下所示：

```
var userAccess = [ "STLW", "G" ];
db.forecasts.aggregate(
    [
      { $match: { year: 2014 } },
      { $redact: {
          $cond: {
              if: { $gt: [ { $size: { $setIntersection: [ "$tags", userAccess ] } }, 0 ] },
              then: "$$DESCEND",
              else: "$$PRUNE"
            }
         }
      }
    ]
);
```

R:

```
{
  "_id" : 1,
  "title" : "123 Department Report",
  "tags" : [ "G", "STLW" ],
  "year" : 2014,
```

```
  "subsections" : [
    {
      "subtitle" : "Section 1: Overview",
      "tags" : [ "SI", "G" ],
      "content" : "Section 1: This is the content of section 1."
    },
    {
      "subtitle" : "Section 2: Analysis",
      "tags" : [ "STLW" ],
      "content" : "Section 2: This is the content of section 2."
    }
  ]
}
```

排除给定级别上的所有字段

收款帐户包含以下文件:

```
{
  _id: 1,
  level: 1,
  acct_id: "xyz123",
  cc: {
    level: 5,
    type: "yy",
    num: 000000000000,
    exp_date: ISODate("2015-11-01T00:00:00.000Z"),
    billing_addr: {
      level: 5,
      addr1: "123 ABC Street",
      city: "Some City"
    },
    shipping_addr: [
      {
        level: 3,
        addr1: "987 XYZ Ave",
        city: "Some City"
      },
      {
        level: 3,
        addr1: "PO Box 0123",
        city: "Some City"
      }
    ]
  },
  status: "A"
}
```

在这个示例文档中，level 字段确定查看数据所需的访问级别。
若要对所有状态为 a 的文档运行查询，并排除级别为 5 的文档/嵌入文档中包含的所有字段，请在 then 字段中包含一个$redact 阶段，该阶段指定系统变量“$$PRUNE”：

```
db.accounts.aggregate(
  [
    { $match: { status: "A" } },
    {
      $redact: {
        $cond: {
          if: { $eq: [ "$level", 5 ] },
          then: "$$PRUNE",
          else: "$$DESCEND"
        }
      }
    }
  ]
);
```

$redact 阶段评估 level 字段以确定访问权限。如果 level 字段等于 5，那么排除该级别的所有字段，即使被排除的字段包含可能具有不同级别值的嵌入式文档，比如 shipping_addr 字段。
聚合操作返回以下“修订过的”文档:

```
{
  "_id" : 1,
  "level" : 1,
  "acct_id" : "xyz123",
  "status" : "A"
}
```

结果集显示，$redact 阶段将字段 cc 作为一个整体排除在外，包括包含嵌入文档的 shipping_addr 字段，该文档的级别字段值为 3 而不是 5。

$replaceRoot:

新版本 3.4。
用**指定的文档替换输入文档**。该操作**替换输入文档中的所有现有字段**，包括_id 字段。您可以将现有的嵌入式文档提升到顶层，或者创建一个新文档进行提升(参见示例)。
$replaceRoot 阶段的形式如下:

```
{ $replaceRoot: { newRoot: <replacementDocument> } }
```

替换文档可以是解析为文档的任何有效表达式。**如果<replacementDocument>不是文档**，则阶段错误并失败。有关表达式的更多信息，请参见表达式。
Behavior:
如果<replacementDocument>**不是文档**，则$replaceRoot **将出错并失败**。
如果<replacementDocument>**解析为丢失的文档(即文档不存在)**，则**$replaceRoot 错误**并

失败。例如，用以下文件创建一个集合:

```
db.collection.insertMany([
    { "_id": 1, "name" : { "first" : "John", "last" : "Backus" } },
    { "_id": 2, "name" : { "first" : "John", "last" : "McCarthy" } },
    { "_id": 3, "name": { "first" : "Grace", "last" : "Hopper" } },
    { "_id": 4, "firstname": "Ole-Johan", "lastname" : "Dahl" },
])
```

然后，下面的$replaceRoot 操作失败，因为其中一个文档没有 name 字段:

```
db.collection.aggregate([
    { $replaceRoot: { newRoot: "$name" } }
])
```

为了避免错误，可以使用$mergeObjects 将名称文档合并到某个默认文档中;例如:

```
db.collection.aggregate([
    { $replaceRoot: { newRoot: { $mergeObjects: [ { _id: "$_id", first: "", last: "" }, "$name" ] } } }
])
```

或者，在将文档传递到$replaceRoot 阶段之前，可以通过**包含$match 阶段来检查文档字段**是否存在，从而**跳过缺少 name 字段**的文档:

```
db.collection.aggregate([
    { $match: { name : { $exists: true, $not: { $type: "array" }, $type: "object" } } },
    { $replaceRoot: { newRoot: "$name" } }
])
```

或者，可以使用**$ifNull 表达式指定其他一些文档作为根**;例如:

```
db.collection.aggregate([
    { $replaceRoot: { newRoot: { $ifNull: [ "$name", { _id: "$_id", missingName: true} ] } } }
])
```

E:

collection named people

```
{ "_id" : 1, "name" : "Arlene", "age" : 34, "pets" : { "dogs" : 2, "cats" : 1 } }
{ "_id" : 2, "name" : "Sam", "age" : 41, "pets" : { "cats" : 1, "fish" : 3 } }
{ "_id" : 3, "name" : "Maria", "age" : 25 }
```

下面的操作**使用$replaceRoot 阶段用 1$mergeObjects 操作的结果替换每个输入文档**。2**$mergeObjects 表达式将**指定的默认文档与 pets 文档合并。

```
db.people.aggregate( [
    { $replaceRoot: { newRoot: { $mergeObjects:    [ { dogs: 0, cats: 0, birds: 0, fish: 0 }, "$pets" ] }} }
] )
```

R:

```
{ "dogs" : 2, "cats" : 1, "birds" : 0, "fish" : 0 }
{ "dogs" : 0, "cats" : 1, "birds" : 0, "fish" : 3 }
{ "dogs" : 0, "cats" : 0, "birds" : 0, "fish" : 0 }
```

用嵌套在数组中的文档替换

一个名为 students 的集合包含以下文件:

```
db.students.insertMany([
```

```
    {
        "_id" : 1,
        "grades" : [
            { "test": 1, "grade" : 80, "mean" : 75, "std" : 6 },
            { "test": 2, "grade" : 85, "mean" : 90, "std" : 4 },
            { "test": 3, "grade" : 95, "mean" : 85, "std" : 6 }
        ]
    },
    {
        "_id" : 2,
        "grades" : [
            { "test": 1, "grade" : 90, "mean" : 75, "std" : 6 },
            { "test": 2, "grade" : 87, "mean" : 90, "std" : 3 },
            { "test": 3, "grade" : 91, "mean" : 85, "std" : 4 }
        ]
    }
])
```

以下操作将等级字段大于或等于 90 的嵌入式文档提升到顶层:

```
db.students.aggregate( [
    { $unwind: "$grades" },
    { $match: { "grades.grade" : { $gte: 90 } } },
    { $replaceRoot: { newRoot: "$grades" } }
] )
```

R:

```
{ "test" : 3, "grade" : 95, "mean" : 85, "std" : 6 }
{ "test" : 1, "grade" : 90, "mean" : 75, "std" : 6 }
{ "test" : 3, "grade" : 91, "mean" : 85, "std" : 4 }
```

用一个新创建的文档替换

您还可以作为$replaceRoot 阶段的一部分创建新文档，并使用它们替换所有其他字段。

名为 contacts 的集合包含以下文档:

```
{ "_id" : 1, "first_name" : "Gary", "last_name" : "Sheffield", "city" : "New York" }
{ "_id" : 2, "first_name" : "Nancy", "last_name" : "Walker", "city" : "Anaheim" }
{ "_id" : 3, "first_name" : "Peter", "last_name" : "Sumner", "city" : "Toledo" }
```

下面的操作从 first_name 和 last_name 字段创建一个新文档。

```
db.contacts.aggregate( [
    {
        $replaceRoot: {
            newRoot: {
                full_name: {                        /合并后的字段 id
                    $concat : [ "$first_name", " ", "$last_name" ]      /合并。
                }
            }
        }
```

```
    }
] )
```
R:
```
{ "full_name" : "Gary Sheffield" }
{ "full_name" : "Nancy Walker" }
{ "full_name" : "Peter Sumner" }
```
使用一个由$$ROOT 创建的新文档和一个默认的文档实施
使用以下文件创建一个名为 contacts 的集合:
```
db.contacts.insert([
    { "_id" : 1, name: "Fred", email: "fred@example.net" },
    { "_id" : 2, name: "Frank N. Stine", cell: "012-345-9999" },
    { "_id" : 3, name: "Gren Dell", home: "987-654-3210", email: "beo@example.net" }
]);
```
下面的操作使用$replaceRoot 和$mergeObjects 来输出当前的文档，这些文档的默认值是缺失字段的值:
```
db.contacts.aggregate([
    { $replaceRoot: { newRoot: { $mergeObjects: [ { _id: "", name: "", email: "", cell: "", home: "" }, "$$ROOT" ] } } }
])
```
R:
```
{ "_id" : 1, "name" : "Fred", "email" : "fred@example.net", "cell" : "", "home" : "" }
{ "_id" : 2, "name" : "Frank N. Stine", "email" : "", "cell" : "012-345-9999", "home" : "" }
{ "_id" : 3, "name" : "Gren Dell", "email" : "beo@example.net", "cell" : "", "home" : "987-654-3210" }
```

$sample

新版本 3.2。
从其输入中随机选择指定数量的文档。
$sample 阶段的语法如下:
```
{ $sample: { size: <positive integer> } }
```
行为
$sample 使用两种方法中的一种来获取 N 个随机文档，具体取决于集合的大小、N 的大小和$sample 在管道中的位置。
如果满足以下所有条件，$sample 使用伪随机游标选择文档:
$sample 是管道的第一阶段
N 小于集合中所有文档的 5%
该集合包含 100 多个文档
如果不满足上述任何条件，$sample 执行集合扫描，然后进行随机排序，以选择 N 个文档。
在本例中，$sample 阶段受排序内存限制。
警告
$sample 可以在其结果集中多次输出相同的文档。有关更多信息，请参见游标隔离。
E:

Collection users:
{ "_id" : 1, "name" : "dave123", "q1" : true, "q2" : true }
{ "_id" : 2, "name" : "dave2", "q1" : false, "q2" : false }
{ "_id" : 3, "name" : "ahn", "q1" : true, "q2" : true }
{ "_id" : 4, "name" : "li", "q1" : true, "q2" : false }
{ "_id" : 5, "name" : "annT", "q1" : false, "q2" : true }
{ "_id" : 6, "name" : "li", "q1" : true, "q2" : true }
{ "_id" : 7, "name" : "ty", "q1" : false, "q2" : true }
E:
db.users.aggregate(
[{ $sample: { size: 3 } }]
)
R:
Returns three random documents
{ "_id" : 1, "name" : "dave123", "q1" : true, "q2" : true }
{ "_id" : 2, "name" : "dave2", "q1" : false, "q2" : false }
{ "_id" : 4, "name" : "li", "q1" : true, "q2" : false }

$set

New in version 4.2
E:向文档添加新字段。$set 输出文档, 其中包含来自输入文档和新添加字段的所有现有字段。$set 阶段是$addFields 的别名。
这两个阶段都相当于一个$project 阶段, 该阶段显式地指定输入文档中的所有现有字段并添加新字段。
E: scores collection
db.scores.insertMany([
{ _id: 1, student: "Maya", homework: [10, 5, 10], quiz: [10, 8], extraCredit: 0 },
{ _id: 2, student: "Ryan", homework: [5, 6, 5], quiz: [8, 8], extraCredit: 8 }
])
下面的操作使用两个$set stage 在输出文档中包含三个新字段:

```
db.scores.aggregate( [
    {
      $set: {
          totalHomework: { $sum: "$homework" },
          totalQuiz: { $sum: "$quiz" }
      }
    },
    {
      $set: {
          totalScore: { $add: [ "$totalHomework", "$totalQuiz", "$extraCredit" ] } }
    }
] )
```

R:

```
{
   "_id" : 1,
   "student" : "Maya",
   "homework" : [ 10, 5, 10 ],
   "quiz" : [ 10, 8 ],
   "extraCredit" : 0,
   "totalHomework" : 25,
   "totalQuiz" : 18,
   "totalScore" : 43
}
{
   "_id" : 2,
   "student" : "Ryan",
   "homework" : [ 5, 6, 5 ],
   "quiz" : [ 8, 8 ],
   "extraCredit" : 8,
   "totalHomework" : 16,
   "totalQuiz" : 16,
   "totalScore" : 40
}
```

$skip

跳过传递到该阶段的指定数量的文档，并将其余文档传递到管道中的下一个阶段。
$skip 阶段的原型表单如下:
{ $skip: <positive integer> }
$skip 接受一个**正整数**，该整数指定要**跳过的文档的最大数量**。
E:

```
db.test1.aggregate(
      { $skip : 5 }
);
```

{ "_id" : 6, "name" : "li", "q1" : true, "q2" : true }
{ "_id" : 7, "name" : "ty", "q1" : false, "q2" : true }
该操作跳过管道传递给它的前 5 个文档。$skip 对它沿着管道传递的文档的内容没有影响。

$sort

对所有输入文档进行排序，并按排序的顺序将它们返回给管道。
$sort 阶段的原型表单如下:
{ $sort: { <field1>: <sort order>, <field2>: <sort order> ... } }
$sort 接受一个文档，该文档指定要排序的字段和相应的排序顺序。<sort order>可以有以下值之一:
指定升序。

-1 指定降序。
{$meta: "textScore"}按计算出的 textScore 元数据降序排序。有关示例，请参见元数据排序。
$sort 操作符和内存
$sort + $limit 内存优化
当$sort 位于$limit 之前，并且没有修改文档数量的中间阶段时，优**化器可以将$limit 合并到$sort 中**。这允许$sort 操作只在进程中维护前 n 个结果，其中 n 是指定的限制，并确保 MongoDB 只需要在内存中存储 n 个条目。**当 allowDiskUse 为真且 n 项超过聚合内存限制时，此优化仍然适用。**
优化可能会随着版本的不同而变化。
$sort 和内存限制
$sort 阶段的内存限制为 100 mb。默认情况下，如果 stage 超过这个限制，$sort 将产生一个错误。要允许处理大型数据集，请将 allowDiskUse 选项设置为 true，以启用$sort 操作来写入临时文件。有关详细信息，请参阅 db.collection.aggregate()方法中的 allowDiskUse 选项和聚合命令。
版本 2.6 中的更改:$sort 的内存限制从 10%的 RAM 更改为 100 mb 的 RAM。
$sort 操作符和性能
当将$sort 操作符放置在管道的开始处或放置在$project、$unwind 和$group 聚合操作符之前时，$sort 操作符可以利用索引。如果$project、$unwind 或$group 发生在$sort 操作之前，那么$sort 不能使用任何索引。

$SortByCount

新版本 3.4。
根据**指定表达式的值对传入文档进行分组**，然后**计算每个不同组中的文档数量**。
每个**输出文档包含两个字段**:一个_id 字段包含不同的分组值，一个 count 字段包含属于该分组或类别的文档数量。
文档按 count 降序排列。
$sortByCount 阶段的原型表单如下:
{ $sortByCount: <expression> }
Expression:
对分组的表达式。您可以指定除文档文字外的任何表达式。
要指定字段路径，请在字段名称前面加上美元符号$并将其括在引号中。例如，要按字段 employee 进行分组，请指定“$employee”作为表达式。
{$sortByCount: "$employee"}
虽然不能为 group by 表达式指定文档文本，但是可以指定计算结果为文档的字段或表达式。例如，如果 employee 字段和 business 字段是文档字段，那么下面的$mergeObjects 表达式是$sortByCounts 的有效参数，它的计算结果是一个文档:
{ $sortByCount: { $mergeObjects: ["$employee", "$business"] } }
但是，下面的文档文字表达式示例是无效的:
{ $sortByCount: { lname: "$employee.last", fname: "$employee.first" } }

Behavior:
$sortByCount 阶段相当于$group + $sort 序列:
{ $group: { _id: <expression>, count: { $sum: 1 } } },

{ $sort: { count: -1 } }
例子
考虑一套附有下列文件的展品:
{ "_id" : 1, "title" : "The Pillars of Society", "artist" : "Grosz", "year" : 1926, "tags" : ["painting", "satire", "Expressionism", "caricature"] }
{ "_id" : 2, "title" : "Melancholy III", "artist" : "Munch", "year" : 1902, "tags" : ["woodcut", "Expressionism"] }
{ "_id" : 3, "title" : "Dancer", "artist" : "Miro", "year" : 1925, "tags" : ["oil", "Surrealism", "painting"] }
{ "_id" : 4, "title" : "The Great Wave off Kanagawa", "artist" : "Hokusai", "tags" : ["woodblock", "ukiyo-e"] }
{ "_id" : 5, "title" : "The Persistence of Memory", "artist" : "Dali", "year" : 1931, "tags" : ["Surrealism", "painting", "oil"] }
{ "_id" : 6, "title" : "Composition VII", "artist" : "Kandinsky", "year" : 1913, "tags" : ["oil", "painting", "abstract"] }
{ "_id" : 7, "title" : "The Scream", "artist" : "Munch", "year" : 1893, "tags" : ["Expressionism", "painting", "oil"] }
{ "_id" : 8, "title" : "Blue Flower", "artist" : "O'Keefe", "year" : 1918, "tags" : ["abstract", "painting"] }
下面的操作展开标签数组，并使用$sortByCount 阶段来计算与每个标签关联的文档数量:
db.exhibits.aggregate([{ $unwind: "$tags" }, { $sortByCount: "$tags" }])
R:db.test2.aggregate([{$unwind:"$tags"},{$group:{_id:"$tags",count:{$sum:1}}},{$sort:{count:-1}}])
{ "_id" : "painting", "count" : 6 }
{ "_id" : "oil", "count" : 4 }
{ "_id" : "Expressionism", "count" : 3 }
{ "_id" : "Surrealism", "count" : 2 }
{ "_id" : "abstract", "count" : 2 }
{ "_id" : "woodblock", "count" : 1 }
{ "_id" : "woodcut", "count" : 1 }
{ "_id" : "ukiyo-e", "count" : 1 }
{ "_id" : "satire", "count" : 1 }
{ "_id" : "caricature", "count" : 1 }

$unwind

将数组字段从输入文档解构为每个元素的输出文档。每个输出文档都是输入文档，数组字段的值由元素替换。
$unwind 阶段有两种语法:
操作数是字段路径:
{$unwind: <field path>}
要指定字段路径，请在字段名称前面加上美元符号$并在引号中括起。
操作数是一个文档:
New in version 3.2.

```
{
  $unwind:
    {
      path: <field path>,
      includeArrayIndex: <string>,
      preserveNullAndEmptyArrays: <boolean>
    }
}
```

字段类型描述

Path: 字符串字 段到数组字段的路径。要指定字段路径，请在字段名称前面加上美元符号$并在引号中括起。

includearrayindex 字符串 可选的。用于保存元素数组索引的新字段的名称。名称不能以美元符号$开头。

Preservenullandemptyarrays 布尔

可选的。如果为真, 如果路径为 null、缺失或一个空数组, $unwind 将输出文档。如果为 false, 如果路径为空、丢失或空数组，$unwind 将不输出文档。

默认值为 false。

行为

非数组字段路径

在 3.2 版中更改:**$unwind 阶段不再出现非数组操作数上的错误。**如果**操作数没有解析为数组但没有丢失**、null 或空数组，**$unwind 将操作数视为单个元素数组。**

以前，**如果字段路径指定的字段中的值不是数组**，db.collection.aggregate()将生成一个错误。

Missing Field

如果您**为输入文档中不存在的字段指定了路径**，或者该字段是一个空数组，那么$unwind 默认情况下会**忽略输入文档**，并且不**会输出该输入文档的文档**。

新版本 3.2: 要输出缺少数组字段(null 或空数组)的文档，请使用 preserveNullAndEmptyArrays 选项。

E:

Unwind Array

考虑使用以下文件的清单:

```
{ "_id" : 1, "item" : "ABC1", sizes: [ "S", "M", "L"] }
```

下面的聚合使用$unwind 阶段为 size 数组中的每个元素输出一个文档:

```
db.inventory.aggregate( [ { $unwind : "$sizes" } ] )
```

R:

```
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

除了 size 字段的值之外，每个文档都与输入文档相同，而 size 字段现在包含来自原始 size 数组的值。

另请参阅

使用邮政编码数据集进行聚合，使用用户首选项数据进行聚合

includeArrayIndex 和 preserveNullAndEmptyArrays¶
新版本 3.2。
收集清单有以下文件:
{ "_id" : 1, "item" : "ABC", "sizes": ["S", "M", "L"] }
{ "_id" : 2, "item" : "EFG", "sizes" : [ ] }
{ "_id" : 3, "item" : "IJK", "sizes": "M" }
{ "_id" : 4, "item" : "LMN" }
{ "_id" : 5, "item" : "XYZ", "sizes" : null }
下面的$unwind 操作是等效的，并为 size 字段中的每个元素返回一个文档。如果 size 字段没有解析为数组但没有丢失、null 或空数组，$unwind 将非数组操作数视为单个元素数组。
db.inventory.aggregate([{ $unwind: "$sizes" }])
db.inventory.aggregate([{ $unwind: { path: "$sizes" } }])
R:
{ "_id" : 1, "item" : "ABC", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC", "sizes" : "L" }
{ "_id" : 3, "item" : "IJK", "sizes" : "M" }
下面的$unwind 操作使用 includeArrayIndex 选项也输出数组元素的数组索引。
db.inventory.aggregate([{ $unwind: { path: "$sizes", includeArrayIndex: "arrayIndex" } }])
该操作展开 size 数组，并在 new arrayIndex 字段中包含数组索引的数组索引。如果 size 字段没有解析为数组，但没有丢失、null 或空数组，则 arrayIndex 字段为 null。
{ "_id" : 1, "item" : "ABC", "sizes" : "S", "arrayIndex" : NumberLong(0) }
{ "_id" : 1, "item" : "ABC", "sizes" : "M", "arrayIndex" : NumberLong(1) }
{ "_id" : 1, "item" : "ABC", "sizes" : "L", "arrayIndex" : NumberLong(2) }
{ "_id" : 3, "item" : "IJK", "sizes" : "M", "arrayIndex" : null }
下面的$unwind 操作使用 preserveNullAndEmptyArrays 选项在输出中包含 size 字段缺失、null 或空数组的文档。
db.inventory.aggregate([
    { $unwind: { path: "$sizes", preserveNullAndEmptyArrays: true } }
])
除对 size 为元素数组或非空、非数组字段的文档进行解卷外，在不进行修改的情况下，对 size 字段缺失、null 或空数组的文档进行操作输出:
{ "_id" : 1, "item" : "ABC", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC", "sizes" : "L" }
{ "_id" : 2, "item" : "EFG" }
{ "_id" : 3, "item" : "IJK", "sizes" : "M" }
{ "_id" : 4, "item" : "LMN" }
{ "_id" : 5, "item" : "XYZ", "sizes" : null }

聚合管道 Operators

在这一节中
In this section
Arithmetic Expression Operators　算术表达式运算符
Array Expression Operators　数组表达式运算符
Boolean Expression Operators　布尔表达式运算符
Comparison Expression Operators 比较表达式操作符
Conditional Expression Operators　条件表达式运算符
Date Expression Operators　日期表达运算符
Literal Expression Operator　文字表达式运算符
Object Expression Operators　对象表达式运营商
Set Expression Operators　组表达运营商
String Expression Operators　字符串表达式运算符
Text Expression Operator　文本表达式运算符
Type Expression Operators　类型表达式运算符
Accumulators ($group)　累加器（组）
Accumulators ($project)　累加器（项目）
Variable Expression Operators　变量表达式运算符

这些表达式运算符可用于构造用于聚合管道阶段的表达式。
运算符表达式类似于接受参数的函数。一般来说，这些表达式采用参数数组，形式如下:
{ <operator>: [<argument1>, <argument2> ...] }
如果运算符接受单个参数，可以省略指定参数列表的外部数组:
{<操作符>:[<argument1>,　<argument2>…]}
如果参数是文字数组，为了避免解析歧义，您必须将文字数组包装在$literal 表达式中，或者保留指定参数列表的外部数组。

Arithmetic Expression Operators(算术表达式运算符)

算术表达式对数字进行数学运算。一些算术表达式也可以支持日期算术。
$abs：　返回一个数字的绝对值。
$add：添加数字来返回总数，或者添加数字和日期来返回新日期。如果添加数字和日期，则将这些数字视为毫秒。接受任意数量的参数表达式，但最多只能解析一个表达式到日期。
$ceil：返回大于或等于指定数字的最小整数。
$divide：返回**第一个数字除以第二个数字的结果**。接受两个参数表达式。
$exp：将 e 提升到指定的指数。
$floor：返回**小于或等于指定数字的最大整数**。
$ln：计算一个数字的自然对数。
$log 计算指定基数中数字的**日志**。
$log10：计算以 **10 为底的对数**。
$mod：返回**第一个数字除以第二个数字的余数**。接受。
$multiply：乘以数字来返回产品。接受任意**两个参数表达式**数量的参数表达式。
$pow：将一个数字**提升到指定的指数。**
$sqrt：计算**平方根。**
$ minus 返回从**第一个值减去第二个值的结果**。如果这**两个值是数字**，则返回差值。如果这

两个值是日期，则以毫秒为单位返回差值。如果这两个值是一个日期 和一个以毫秒为单位的数字，则返回结果日期。接受两个参数表达式。如果这两个值是日期和数字，请首先指定 date 参数，因为从数字中减去日期没有意义。
$trunc 将数字截断为整数。

Array Expression Operators(数组表达式运算符)

$ arrayElemAt: 返回指定数组索引处的元素。
$ arrayToObject: 将键值对数组转换为文档。
$ concatArrays: 连接数组以返回连接的数组。
$filter: 选择数组的一个子集来返回一个数组，该数组只包含与筛选条件匹配的元素。
$in: 返回一个布尔值，指示指定的值是否在数组中。
$ indexOfarray: 在数组中搜索指定值的出现，并返回第一次出现的数组索引。如果没有找到子字符串，则返回-1。
$ isarray: 确定操作数是否为数组。返回一个布尔值。
$map: 对数组的**每个元素应用子表达式**，并**按顺序返回结果值的数组**。接受命名参数。
$ objecttoarray: 将文档转换为表示键值对的**文档数组**。
$range: 根据用户定义的**输入输出一个包含整数序列的数组**。
$reduce: 对数组中的**每个元素应用一个表达式**，并将它们组合成一个值。
$ reverseArray: 返回一个数组，其中的元素按相反的顺序排列。
返回数组中元素的数量。接受单个表达式作为参数。
$slice: 返回数组的子集。
$zip: 将两个数组合并在一起。

Boolean Expression Operators(布尔表达式运算符)

布尔表达式将它们的参数表达式计算为布尔值，并返回一个布尔值作为结果。
除了伪布尔值之外，布尔表达式的计算结果为 false，如下所示:null、0 和未定义的值。布尔表达式计算所有其他值为真，包括非零数值和数组。
名称描述
$and: 仅当所有表达式的值都为 true 时才返回 true。接受任意数量的参数表达式。
$not: 返回与参数表达式相反的布尔值。接受单个参数表达式。
$or: 返回 true。接受任意数量的参数表达式。当其中一个表达式的值为 true 时，

Comparison Expression Operators(比较表达式操作符)

比较表达式**返回一个布尔值**，但**$cmp 返回一个数字。**
比较表达式**接受两个参数表达式**，并使用**指定的 BSON 比较顺序比较值和类型**。

$cmp: 如果两个值相等，返回 0;如果第一个值大于第二个值，返回 1;如果第一个值小于第二个值，返回-1。
$eq: 如果值相等，则返回 true。

$gt: 如果第一个值大于第二个值，则返回 true。
$gte: 如果第一个值大于或等于第二个值，则返回 true。
$gt: 如果第一个值小于第二个值，则$lt 返回 true。
$gte: 如果第一个值小于或等于第二个值，则返回 true。
$ne: 如果值不相等，则返回 true。

String Expression Operators（字符串表达式运算符）

除了$concat 之外，字符串表达式只有定义良好的 ASCII 字符字符串行为。
无论使用什么字符，都可以定义$concat 行为。

名称描述
$concat: 连接任意数量的字符串。
$ datefromstring: 将日期/时间字符串转换为日期对象。
$ datetostring: 以格式化字符串的形式返回日期。
$ indexofbytes: 搜索一个字符串，查找子字符串的发生情况，并返回第一个发生情况的 UTF-8 字节索引。如果没有找到子字符串，则返回-1。
$ indexofcp: 搜索一个字符串，查找子字符串的发生情况，并返回第一个发生情况的 UTF-8 代码点索引。如果没有找到子字符串，则返回-1
$ltrim: 删除字符串开头的空格**或指定的字符**。新版本 4.0。
$rtrim: 从字符串末尾删除空白或指定的字符。新版本 4.0。
$split: 根据分隔符将字符串分割为子字符串。返回子字符串数组。如果在字符串中没有找到分隔符，则返回包含原始字符串的数组。
$ strlenbytes: 返回字符串中 UTF-8 编码的字节数。
$ strlencp: 返回字符串中 UTF-8 代码点的数量。
$strcasecmp: 执行不区分大小写的字符串比较，如果两个字符串相等，返回 0;如果第一个字符串大于第二个字符串，返回 1;如果第一个字符串小于第二个字符串，返回-1。
substr 弃用美元。使用$substrBytes 或$substrCP。
$ substrbytes: 返回字符串的子字符串。从字符串中指定 UTF-8 字节索引(从零开始)处的字符开始，并继续指定字节数。
$ substrcp: 返回字符串的子字符串。从字符串中指定的 UTF-8 代码点(CP)索引(从零开始)处的字符开始，并继续指定代码点的数量。
$ tolower: 将字符串转换为小写。接受单个参数表达式。
$toString: 将值转换为字符串。新版本 4.0。
$trim: 删除字符串开头和结尾的空格或指定字符。新版本 4.0。
$ toupper: 将字符串转换为大写。接受单个参数表达式。

Conditional Expression Operators（条件表达式运算符）

名称描述
$cond: 三元运算符，它计算一个表达式，根据结果返回另外两个表达式之一的值。接受有序列表中的三个表达式或三个命名参数。

$ ifnull： 返回第一个表达式的非空结果，如果第一个表达式结果为空，则返回第二个表达式的结果。Null result 包含未定义值或缺少字段的实例。接受两个表达式作为参数。第二个表达式的结果可以是 null。
$switch： 计算一系列大小写表达式。当它找到一个计算结果为 true 的表达式时，$switch 执行一个指定的表达式并跳出控制流。
Date Expression Operators(日期表达运营商)
以下运算符返回日期对象或日期对象的组件:
名称描述
$ datefromparts: 给定日期的组成部分，构造一个 BSON Date 对象。
$ datefromstring 将日期/时间字符串转换为日期对象。
$ datetopart 返回包含日期组成部分的文档。
$ datetostring 以格式化字符串的形式返回日期。
$ dayofmonth 将日期的月日作为 1 到 31 之间的数字返回。
$ dayofweek 以 1(星期日)到 7(星期六)之间的数字返回日期的星期几。
$ dayofyear 以 1 到 366(闰年)之间的数字返回日期的年份。
$hour 将日期的小时作为 0 到 23 之间的数字返回。
$ isodayofweek 返回 ISO 8601 格式的工作日号，范围从 1(周一)到 7(周日)。
$ isoweek 以 ISO 8601 格式返回周数，范围从 1 到 53。周数从 1 开始，包含一年的第一个星期四的那一周(周一到周日)。
$ isoweekyear 以 ISO 8601 格式返回年份号。一年从第一周的星期一(ISO 8601)开始，到最后一周的星期日(ISO 8601)结束。
$ ond 以 0 到 999 之间的数字返回日期的毫秒数。
$minute 将日期的分钟作为 0 到 59 之间的数字返回。
$month 将日期的月份作为 1(一月)到 12(十二月)之间的数字返回。
$second 以 0 到 60 之间的数字(闰秒)返回日期的秒。
$toDate: 将值转换为日期。新版本 4.0。
$week: 将日期的周数返回为 0(一年的第一个星期日之前的部分周)和 53(闰年)之间的数字。
$year: 以数字的形式返回日期的年份(例如 2014 年)。
以下算术运算符可以接受日期操作数:
名称描述
$add: 添加数字和日期以返回新的日期。如果添加数字和日期，则将这些数字视为毫秒。接受任意数量的参数表达式，但最多只能解析一个表达式到日期。
$ minus: 返回从第一个值减去第二个值的结果。如果这两个值是日期，则以毫秒为单位返回差值。如果这两个值是一个日期和一个以毫秒为单位的数字，则返回结果日期。接受两个参数表达式。如果这两个值是日期和数字，请首先指定 date 参数，因为从数字中减去日期没有意义。

文字表达运算符
名称描述
$literal： 返回一个不需要解析的值。用于聚合管道可能解释为表达式的值。例如，对以$开头的字符串使用$literal 表达式，以避免将其解析为字段路径。
Object Expression Operators 对象表达式运营商
名称描述

$mergeObjects:　将多个文档合并到一个文档中。新版本 3.6。
$objectToArray:　将文档转换为表示键值对的文档数组。新版本 3.6。

组表达运算符
Set 表达式对数组执行 Set 操作，将数组视为集合。Set 表达式忽略每个输入数组中的重复项和元素的顺序。
如果 set 操作返回一个集合，则该操作过滤掉结果中的重复项，以输出只包含唯一条目的数组。输出数组中元素的顺序未指定。
如果集合包含嵌套数组元素，则集合表达式不会下降到嵌套数组中，而是在顶层计算数组。

$ allElementsTrue:　如果没有一个集合的元素计算为 false，则返回 true，否则返回 false。接受单个参数表达式。
$ anyelemenTtrue:　如果集合中的任何元素的值为 true，则返回 true;否则,返回 false。接受单个参数表达式。
$ setDifference:　返回一个集合，其中的元素出现在第一个集合中，但不在第二个集合中;即执行第二组相对于第一组的相对补码。只接受两个参数表达式。
$ setequals:　如果输入集具有相同的不同元素，则返回 true。接受两个或多个参数表达式。
$ setIntersection 返回一个集合，其中包含出现在所有输入集中的元素。接受任意数量的参数表达式。
如果第一个集合的所有元素都出现在第二个集合中，包括当第一个集合等于第二个集合时，
$ setisIsSubset:　返回 true;也就是说，不是一个严格的子集。只接受两个参数表达式。
$ setUnion:　返回一个集合，其中包含出现在任何输入集中的元素。

Text Expression Operator(文本表达式运算符)

名称描述
$meta:　访问文本搜索元数据。

Type Expression Operators 类型表达式运算符

名称描述
$convert:　将值转换为指定的类型。新版本 4.0。

$toBool:　将值转换为布尔值。新版本 4.0。

$toDate:　将值转换为日期。新版本 4.0。

$toDecimal:　将值转换为十进制 128。新版本 4.0。
$toDouble:　将值转换为 double。新版本 4.0。
$toInt;　将值转换为整数。新版本 4.0。
$toLong:将值转换为 long。新版本 4.0。
$toObjectId:　将值转换为 ObjectId。新版本 4.0。
$toString:　将值转换为字符串。新版本 4.0。

$type:　返回字段的 BSON 数据类型。

Accumulators 累加器（组)

累加器可以在$group 阶段中使用，它是一些操作符，当文档在管道中进行时，它们维护自己的状态(例如 total、maximums、minimums 和相关数据)。

在$group 阶段中用作累加器时，这些操作符将单个表达式作为输入，对每个输入文档计算表达式一次，并为共享相同组键的文档组维护它们的阶段。
名称描述
$ addtoset:　返回每个组的唯一表达式值数组。数组元素的顺序未定义。
$avg:　返回数值的平均值。忽略了非数字值。
$first:　为每个组从第一个文档返回一个值。只有当文档按照已定义的顺序时才定义 Order。
$last:　从每个组的上一个文档返回一个值。只有当文档按照已定义的顺序时才定义 Order。
$max:　返回每个组的最高表达式值。
$ mergeobjects:　返回通过组合每个组的输入文档创建的文档。
$min:　返回每个组的最低表达式值。
$push:　返回每个组的表达式值数组。
$ stddevpop:　返回输入值的总体标准差。
$ stddevsamp:　返回输入值的样本标准差。
$sum:　返回数值的和。忽略了非数字值。

Accumulators ($project)累加器（(项目)）

在$group 阶段可以作为累加器使用的一些操作符也可以在$project 阶段中使用，但不能作为累加器。当在$project 阶段使用时，这些操作符不维护它们的状态，可以接受单个参数或多个参数作为输入。

在 3.2 版本中进行了更改。

在$project 和$addFields 阶段还可以使用以下累加器操作符。
$avg：　返回每个文档指定表达式或表达式列表的平均值。忽略了非数字值。
$max：　返回每个文档的指定表达式或表达式列表的最大值
$min：　返回每个文档的指定表达式或表达式列表的最小值
$ stddevpop：　返回输入值的总体标准差。
$ stddevsamp：　返回输入值的样本标准差。
$sum：　返回数值的和。忽略了非数字值。

Variable Expression Operators 变量表达式运算符

名称描述
$let:　　定义用于子表达式范围内的变量，并返回子表达式的结果。接受命名参数。
接受任意数量的参数表达式。

Alphabetical Listing of Expression Operators（）字母的清单的表达式运算符()

名称描述
$abs　　返回一个数字的绝对值。
$add　　添加数字来返回总数，或者添加数字和日期来返回新日期。如果添加数字和日期，则将这些数字视为毫秒。接受任意数量的参数表达式，但最多只能解析一个表达式到日期。
$addToSet：返回每个组的唯一表达式值数组。数组元素的顺序未定义。只适用于$group stage。
$ allelementstrue　　如果没有一个集合的元素计算为 false，则返回 true，否则返回 false。接受单个参数表达式。
仅当所有表达式的值都为 true 时才返回 true。接受任意数量的参数表达式。
$ anyelementtrue　　如果集合中的任何元素的值为 true，则返回 true;否则,返回 false。接受单个参数表达式。
$ arrayelemat　　返回指定数组索引处的元素。
$ arraytoobject　　将键值对数组转换为文档。
$avg　　返回数值的平均值。忽略了非数字值。在 3.2 版本中进行了更改:在$group 和 $project　　阶段都可以使用。
$ceil　　返回大于或等于指定数字的最小整数。
$cmp　　返回:如果两个值相等，则返回 0;如果第一个值大于第二个值，返回 1;如果第一个值小于第二个值，返回-1。
$concat　　连接任意数量的字符串。
$ concatarrays　　连接数组以返回连接的数组。
$cond　　三元运算符，它计算一个表达式，根据结果返回另外两个表达式之一的值。接受有序列表中的三个表达式或三个命名参数。
$convert　　将值转换为指定的类型。
给定日期的组成部分，$ datefromparts 构造一个 BSON Date 对象。
$ datetopart　　返回包含日期组成部分的文档。
$ datefromstring　　返回日期/时间作为日期对象。
$ datetostring　　以格式化字符串的形式返回日期。
$ dayofmonth　　将日期的月日作为 1 到 31 之间的数字返回。
$ dayofweek　　以 1(星期日)到 7(星期六)之间的数字返回日期的星期几。
$ dayofyear　　以 1 到 366(闰年)之间的数字返回日期的年份。
$divide　　返回第一个数字除以第二个数字的结果。接受两个参数表达式。
如果值相等，则$eq 返回 true。
$exp　　将 e 提升到指定的指数。

$filter　选择数组的一个子集来返回一个数组，该数组只包含与筛选条件匹配的元素。
$first:　从第一个文档中为每个组返回一个值。只有当文档按照已定义的顺序时才定义 Order。只适用于$group stage。
$floor:　返回小于或等于指定数字的最大整数。
$gt:　如果第一个值大于第二个值，则返回 true。
$gte:　如果第一个值大于或等于第二个值，则返回 true。
$hour　将日期的小时作为 0 到 23 之间的数字返回。
$ ifnull　返回第一个表达式的非空结果，如果第一个表达式结果为空，则返回第二个表达式的结果。Null result 包含未定义值或缺少字段的实例。接受两个表达式作为参数。第二个表达式的结果可以是 null。
$in　返回一个布尔值，指示指定的值是否在数组中。
$ indexofarray　在数组中搜索指定值的出现，并返回第一次出现的数组索引。如果没有找到子字符串，则返回-1。
$ indexofbytes　搜索一个字符串，查找子字符串的发生情况，并返回第一个发生情况的 UTF-8 字节索引。如果没有找到子字符串，则返回-1。
$ indexofcp　搜索一个字符串，查找子字符串的发生情况，并返回第一个发生情况的 UTF-8 代码点索引。如果没有找到子字符串，则返回-1。
$ isarray　确定操作数是否为数组。返回一个布尔值。
$ isodayofweek　返回 ISO 8601 格式的工作日号，范围从 1(周一)到 7(周日)。
$ isoweek　以 ISO 8601 格式返回周数，范围从 1 到 53。周数从 1 开始，包含一年的第一个星期四的那一周(周一到周日)。
$ isoweekyear　以 ISO 8601 格式返回年份号。一年从第一周的星期一(ISO 8601)开始，到最后一周的星期日(ISO 8601)结束。
$last:　为每个组从上一个文档返回一个值。只有当文档按照已定义的顺序时才定义 Order 只适用于$group stage。

$let:　定义用于子表达式范围内的变量，并返回子表达式的结果。接受命名参数。接受任意数量的参数表达式。
$literal:　返回一个不需要解析的值。用于聚合管道可能解释为表达式的值。例如，对以$开头的字符串使用$literal 表达式，以避免将其解析为字段路径。
$ln:　计算一个数字的自然对数。
$log:　计算指定基数中数字的日志。
$log10:　计算以 10 为底的对数。
$lt:　如果第一个值小于第二个值，则返回 true。
$lte:　:如果第一个值小于或等于第二个值，则返回 true。
$ltrim:　删除字符串开头的空格或指定的字符。
$map:　对数组的每个元素应用子表达式，并按顺序返回结果值的数组。接受命名参数。
$max:　返回每个组的最高表达式值。在 3.2 版本中进行了更改:在$group 和$project 阶段都可以使用。
$ mergeobjects:　将多个文档组合成一个文档。
$meta:　访问文本搜索元数据。
$min:　返回每个组的最低表达式值。在 3.2 版本中进行了更改:在$group 和$project 阶段都可以使用。

$ millisecond:　以 0 到 999 之间的数字返回日期的毫秒数。
$minute:　将日期的分钟作为 0 到 59 之间的数字返回。
$mod:　返回第一个数字除以第二个数字的余数。接受两个参数表达式。
$month:　将日期的月份作为 1(一月)到 12(十二月)之间的数字返回。
$multiply:　乘以数字来返回产品。接受任意数量的参数表达式。
如果值不相等，则$ne 返回 true。
$not:　返回与参数表达式相反的布尔值。接受单个参数表达式。
$ objecttoarray:　将文档转换为表示键值对的文档数组。
当其中一个表达式的值为 true 时，$or 返回 true。接受任意数量的参数表达式。
$pow:　将一个数字提升到指定的指数。
$push　返回每个组的表达式值数组。只适用于$group stage。
$range　根据用户定义的输入输出一个包含整数序列的数组。
$reduce　对数组中的每个元素应用一个表达式，并将它们组合成一个值。
$ reversearray　返回一个数组，其中的元素按相反的顺序排列。
$rtrim　删除字符串末尾的空格或指定字符。
$second　以 0 到 60 之间的数字(闰秒)返回日期的秒。
$ setdifference　返回一个集合，其中的元素出现在第一个集合中，但不在第二个集合中;即执行第二组相对于第一组的相对补码。只接受两个参数表达式。
$ setequals:　如果输入集具有相同的不同元素，则返回 true。接受两个或多个参数表达式。
$ setIntersection:　返回一个集合，其中包含出现在所有输入集中的元素。接受任意数量的参数表达式。
$ setisSubset　如果第一个集合的所有元素都出现在第二个集合中，包括当第一个集合等于第二个集合时，子集返回 true;也就是说，不是一个严格的子集。只接受两个参数表达式。
$ setunion:　返回一个集合，其中包含出现在任何输入集中的元素。
返回数组中元素的数量。接受单个表达式作为参数。
$slice　返回数组的子集。
$split　根据分隔符将字符串分割为子字符串。返回子字符串数组。如果在字符串中没有找到分隔符，则返回包含原始字符串的数组。
$sqrt　计算平方根。
$stdDevPop:　返回输入值的总体标准差。在 3.2 版本中进行了更改:在$group 和$project 阶段都可以使用。
$stdDevSamp:　返回输入值的样本标准差。在 3.2 版本中进行了更改:在$group 和$project 阶段都可以使用。

$strcasecmp　执行不区分大小写的字符串比较，如果两个字符串相等，返回 0;如果第一个字符串大于第二个字符串，返回 1;如果第一个字符串小于第二个字符串，返回-1。
$ strlenbytes　返回字符串中 UTF-8 编码的字节数。
$ strlencp　返回字符串中 UTF-8 代码点的数量。
substr 弃用美元。使用$substrBytes 或$substrCP。
$ substrbytes　返回字符串的子字符串。从字符串中指定 UTF-8 字节索引(从零开始)处的字符开始，并继续指定字节数。
$ substrcp　返回字符串的子字符串。从字符串中指定的 UTF-8 代码点(CP)索引(从零开

始)处的字符开始，并继续指定代码点的数量。
$ subtract　　返回从第一个值减去第二个值的结果。如果这两个值是数字，则返回差值。如果这两个值是日期，则以毫秒为单位返回差值。如果这两个值是一个日期和一个以毫秒为单位的数字，则返回结果日期。接受两个参数表达式。如果这两个值是日期和数字，请首先指定 date 参数，因为从数字中减去日期没有意义。

$sum:　　返回数值的和。忽略了非数字值。在 3.2 版本中进行了更改:在$group 和$project 阶段都可以使用。
$switch　　计算一系列大小写表达式。当它找到一个计算结果为 true 的表达式时，$switch 执行一个指定的表达式并跳出控制流。
$ tobool　　将值转换为布尔值。
$ todate　　将值转换为日期。
$ todecimal　　将值转换为 Decimal128。
$ todouble　　将值转换为 double。
$ toint　　将值转换为整数。
$ tolong　　将价值转换为 long。
$ toobjectid　　将值转换为 ObjectId。
$ tostring　　将值转换为字符串。
$ tolower　　将字符串转换为小写。接受单个参数表达式。
$ toupper　　将字符串转换为大写。接受单个参数表达式。
$trim　　删除字符串开头和结尾的空格或指定字符。
$trunc　　将数字截断为整数。返回字段的 BSON 数据类型。
$week　　将日期的周数返回为 0(一年的第一个星期日之前的部分周)和 53(闰年)之间的数字。
$year　　以数字的形式返回日期的年份(例如 2014 年)。
$zip　　将两个数组合并在一起。

聚合管道 Operators（操作）

$abs

新版本 3.2。
返回一个数字的绝对值。
$abs 的语法如下:
{ $abs: <number> }
<number>表达式可以是任何有效的表达式，只要它解析为一个数字。有关表达式的更多信息，请参见表达式。
行为
如果参数解析为 null 值或引用缺失的字段，$abs 返回 null。如果参数解析为 NaN，　$abs 返回 NaN。

Example	Results
{ $abs: –1 }	1
{ $abs: 1 }	1
{ $abs: null }	null

E:收集评级包括以下文件:
{ _id: 1, start: 5, end: 8 }
{ _id: 2, start: 4, end: 4 }
{ _id: 3, start: 9, end: 7 }
{ _id: 4, start: 6, end: 7 }
下面的例子计算了开始和结束评级之间的差异大小:

```
db.ratings.aggregate([
    {
      $project: { delta: { $abs: { $subtract: [ "$start", "$end" ] } } }
    }
])
```

R:
{ "_id" : 1, "delta" : 3 }
{ "_id" : 2, "delta" : 0 }
{ "_id" : 3, "delta" : 2 }
{ "_id" : 4, "delta" : 1 }

$add

将数字相加或将数字和日期相加。如果其中一个参数是日期，$add 将其他参数视为要添加到日期中的毫秒。
$add 表达式的语法如下:
{ $add: [<expression1>, <expression2>, ...] }
参数可以是**任何有效的表达式**，只要**它们解析为所有数字或数字和日期**。有关表达式的更多信息，请参见表达式。
E:
下面的例子使用一个 sales 集合，其中包含以下文档:
{ "_id" : 1, "item" : "abc", "price" : 10, "fee" : 2, date: ISODate("2014-03-01T08:00:00Z") }

{ "_id" : 2, "item" : "jkl", "price" : 20, "fee" : 1, date: ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "fee" : 0, date: ISODate("2014-03-15T09:00:00Z") }

Add Numbers:

下面的聚合使用$project 管道中的$add 表达式来计算总成本:

```
db.sales.aggregate(
    [
      { $project: { item: 1, total: { $add: [ "$price", "$fee" ] } } }
    ]
)
```

R:

{ "_id" : 1, "item" : "abc", "total" : 12 }
{ "_id" : 2, "item" : "jkl", "total" : 21 }
{ "_id" : 3, "item" : "xyz", "total" : 5 }

在日期实施添加

下面的聚合使用$add 表达式计算 billing_date，方法是向 **date 字段添加 3*24*60*60000 毫秒**(即 3 天):

```
db.sales.aggregate(
    [
      { $project: { item: 1, billing_date: { $add: [ "$date", 3*24*60*60000 ] } } }
    ]
)
```

R:

{ "_id" : 1, "item" : "abc", "billing_date" : ISODate("2014-03-04T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "billing_date" : ISODate("2014-03-04T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "billing_date" : ISODate("2014-03-18T09:00:00Z") }

$addToSet:

返回一个由所有惟一值组成的数组, 这些值是将表达式应用于一组按键共享同一组的文档中的每个文档而得到的。输出数组中元素的顺序未指定。

$addToSet 只在$group 阶段可用。

$addToSet 有以下语法:

{ $addToSet: <expression> }

Behavior：

如果表达式的值是数组，则$addToSet 将整个数组附加为单个元素。

如果表达式的值是一个文档, 如果数组中的另一个文档与要添加的文档完全匹配, MongoDB 确定该文档是一个副本;也就是说，现有文档具有完全相同的字段和值，它们的顺序完全相同。

E:

考虑一个包含以下文件的 sales 集合:

{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }

{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:12:00Z") }
按日期字段的日期和年份对文档进行分组，下面的操作使用**$addToSet 累加器计算每个组出售的惟一项目列表**:
$ dayofyear 以返回日期中 1 到 366(闰年)之间的数字

```
db.sales.aggregate(
    [
      {
        $group:
          {
            _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
            itemsSold: { $addToSet: "$item" }
          }
      }
    ]
)
```

操作返回以下结果:

```
{ "_id" : { "day" : 46, "year" : 2014 }, "itemsSold" : [ "xyz", "abc" ] }
{ "_id" : { "day" : 34, "year" : 2014 }, "itemsSold" : [ "xyz", "jkl" ] }
{ "_id" : { "day" : 1, "year" : 2014 }, "itemsSold" : [ "abc" ] }
```

$allElementsTrue

新版本 2.6。
将数组计算为一个集合，如果数组中没有元素为假，则返回 true。否则,返回 false。空数组返回 true
$allElementsTrue 有以下语法:

```
{ $allElementsTrue: [ <expression> ] }
```

<表达式>本身必须解析为一个数组，与表示参数列表的外部数组分开。有关表达式的更多信息，请参见表达式。
Behavior:
如果集合包含嵌套数组元素，则$allElementsTrue 不会下降到嵌套数组中，而是在顶层计算数组。

除了 false boolean 值之外，$allElementsTrue 还将以下值计算为 false: null、0 和未定义的值。$allElementsTrue 将所有其他值计算为 true，包括非零数值和数组。

Example	Result

Example	Result
{ $allElementsTrue: [[true, 1, "someString"]] }	true
{ $allElementsTrue: [[[false]]] }	true
{ $allElementsTrue: [[ ]] }	true
{ $allElementsTrue: [[null, false, 0]] }	false

E:
survey collection
{ "_id" : 1, "responses" : [true] }
{ "_id" : 2, "responses" : [true, false] }
{ "_id" : 3, "responses" : [ ] }
{ "_id" : 4, "responses" : [1, true, "seven"] }
{ "_id" : 5, "responses" : [0] }
{ "_id" : 6, "responses" : [[ ]] }
{ "_id" : 7, "responses" : [[0]] }
{ "_id" : 8, "responses" : [[false]] }
{ "_id" : 9, "responses" : [null] }
{ "_id" : 10, "responses" : [undefined] }
下面的操作使用$allElementsTrue 操作符来确定响应数组是否只包含求值为 true 的值:

```
db.survey.aggregate(
    [
      { $project: { responses: 1, isAllTrue: { $allElementsTrue: [ "$responses" ] }, _id:
0 } }
    ]
)
```

R:
{ "responses" : [true], "isAllTrue" : true }
{ "responses" : [true, false], "isAllTrue" : false }
{ "responses" : [ ], "isAllTrue" : true }
{ "responses" : [1, true, "seven"], "isAllTrue" : true }
{ "responses" : [0], "isAllTrue" : false }
{ "responses" : [[ ]], "isAllTrue" : true }
{ "responses" : [[0]], "isAllTrue" : true }

{ "responses" : [[false]], "isAllTrue" : true }
{ "responses" : [null], "isAllTrue" : false }
{ "responses" : [null], "isAllTrue" : false }

$and

计算一个或多个表达式，如果所有表达式都为真，或者没有参数表达式，则返回 true。否则，$和返回 false。
$的语法如下:
{$and: [<expression1>， <expression2>，…]}
有关表达式的更多信息，请参见表达式。
Behavior
$and 使用短路逻辑:操作在遇到第一个错误表达式后停止计算。
除了 false boolean 值之外，$和的计算结果为 false，如下所示:null、0 和未定义的值。$和计算所有其他值为 true，包括非零数值和数组。

Example	Result
{ $and: [1, "green"] }	true
{ $and: [ ] }	true
{ $and: [[null], [false], [0]] }	true
{ $and: [null, true] }	false
{ $and: [0, true] }	false

E:
inventory collection

```
db.inventory.insertMany([
    { "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 },
    { "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 },
    { "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 },
    { "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 },
    { "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

])
下面的操作使用$and 操作符来确定 qty 是否大于 100 和小于 250:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
            qty: 1,
            result: { $and: [ { $gt: [ "$qty", 100 ] }, { $lt: [ "$qty", 250 ] } ] }
          }
      }
    ]
)
```

R:

```
{ "_id" : 1, "item" : "abc1", "qty" : 300, "result" : false }
{ "_id" : 2, "item" : "abc2", "qty" : 200, "result" : true }
{ "_id" : 3, "item" : "xyz1", "qty" : 250, "result" : false }
{ "_id" : 4, "item" : "VWZ1", "qty" : 300, "result" : false }
{ "_id" : 5, "item" : "VWZ2", "qty" : 180, "result" : true }
```

$anyElementTrue

新版本 2.6。

将数组计算为一个集合，如果其中任何元素为真或为假，则返回 true。空数组返回 false。

$anyElementTrue 的语法如下:

{$anyElementTrue: [<expression>]}

<表达式>本身必须解析为一个数组，与表示参数列表的外部数组分开。有关表达式的更多信息，请参见表达式。

如果集合包含嵌套数组元素，则$anyElementTrue 不会下降到嵌套数组中，而是在顶层计算数组。

除了 false boolean 值之外，$anyElementTrue 的计算结果为 false，如下所示:null、0 和未定义的值。$anyElementTrue 将所有其他值计算为 true，包括非零数值和数组。

Example	Result
{ $anyElementTrue: [[true, false]] }	true
{ $anyElementTrue: [[[false]]] }	true
{ $anyElementTrue: [[null, false, 0]] }	false
{ $anyElementTrue: [[ ]] }	false

E:

survey collection

```
{ "_id" : 1, "responses" : [ true ] }
{ "_id" : 2, "responses" : [ true, false ] }
{ "_id" : 3, "responses" : [ ] }
{ "_id" : 4, "responses" : [ 1, true, "seven" ] }
```

{ "_id" : 5, "responses" : [0] }
{ "_id" : 6, "responses" : [[ ]] }
{ "_id" : 7, "responses" : [[0]] }
{ "_id" : 8, "responses" : [[false]] }
{ "_id" : 9, "responses" : [null] }
{ "_id" : 10, "responses" : [undefined] }
下面的操作使用$anyElementTrue 操作符来确定响应数组是否包含任何计算结果为 true 的值:

```
db.survey.aggregate(
    [
      { $project: { responses: 1, isAnyTrue: { $anyElementTrue: [ "$responses" ] }, _id: 0 } }
    ]
)
```

R:
{ "responses" : [true], "isAnyTrue" : true }
{ "responses" : [true, false], "isAnyTrue" : true }
{ "responses" : [ ], "isAnyTrue" : false }
{ "responses" : [1, true, "seven"], "isAnyTrue" : true }
{ "responses" : [0], "isAnyTrue" : false }
{ "responses" : [[ ]], "isAnyTrue" : true }
{ "responses" : [[0]], "isAnyTrue" : true }
{ "responses" : [[false]], "isAnyTrue" : true }
{ "responses" : [null], "isAnyTrue" : false }
{ "responses" : [null], "isAnyTrue" : false }

$arrayElemAt:

新版本 3.2。
返回指定数组索引处的元素。
$arrayElemAt 的语法如下:
{$arrayElemAt: [<array>, <idx>]]
只要解析为数组，<array>表达式可以是任何有效的表达式。
<idx>表达式可以是任何有效的表达式，只要它解析为整数。
如果是正数，$arrayElemAt 返回 idx 位置的元素，从数组的开始计数。
如果是负数，$arrayElemAt 返回 idx 位置的元素，从数组末尾开始计数。
如果 idx 超过数组边界，$arrayElemAt 不会返回任何结果。
有关表达式的更多信息，请参见表达式。

E	R
{ $arrayElemAt: [[1, 2, 3], 0] }	1
{ $arrayElemAt: [[1, 2, 3], -2] }	2
{ $arrayElemAt: [[1, 2, 3], 15] }	

E:
collection named users

{ "_id" : 1, "name" : "dave123", favorites: ["chocolate", "cake", "butter", "apples"] }
{ "_id" : 2, "name" : "li", favorites: ["apples", "pudding", "pie"] }
{ "_id" : 3, "name" : "ahn", favorites: ["pears", "pecans", "chocolate", "cherries"] }
{ "_id" : 4, "name" : "ty", favorites: ["ice cream"] }、
下面的例子返回收藏夹数组中的第一个和最后一个元素:

```
db.users.aggregate([
    {
      $project:
       {
           name: 1,
           first: { $arrayElemAt: [ "$favorites", 0 ] },
           last: { $arrayElemAt: [ "$favorites", -1 ] }
       }
    }
])
```

R:
{ "_id" : 1, "name" : "dave123", "first" : "chocolate", "last" : "apples" }
{ "_id" : 2, "name" : "li", "first" : "apples", "last" : "pie" }
{ "_id" : 3, "name" : "ahn", "first" : "pears", "last" : "cherries" }
{ "_id" : 4, "name" : "ty", "first" : "ice cream", "last" : "ice cream" }

$arrayToObject:

新版本 3.4.4。
将数组转换为单个文档;数组必须是:
一个由两个元素组成的数组，其中第一个元素是字段名，第二个元素是字段值:
[["item", "abc123"], ["qty", 25]]
——或
包含两个字段 k 和 v 的文档数组，其中:
k 字段包含字段名。
v 字段包含字段的值。
$arrayToObject:
{ $arrayToObject: <expression> }
<expression>可以是任何解析为包含“k”和“v”字段的双元素数组或文档数组的有效表达式。
Behavior:
如果字段名在数组中重复，
从 4.0.5 开始，$arrayToObject 使用该字段的最后一个值。对于 4.0.0-4.0.4，使用的值取决于驱动程序。
从 3.6.10 开始，$arrayToObject 使用该字段的最后一个值。对于 3.6.0-3.6.9，使用的值取决于驱动程序。
从 3.4.19 开始，$arrayToObject 使用该字段的最后一个值。对于 3.4.0-3.4.19，使用的值取决于驱动程序。
例子]

```
{ $arrayToObject:   { $literal: [
        { "k": "item", "v": "abc123"},
        { "k": "qty", "v": 25 }
] } }
                                                    { "item" : "abc123", "qty" : 25 }
{ $arrayToObject: { $literal:   [
    [ "item", "abc123"], [ "qty", 25 ]
] } }
                                                    { "item" : "abc123", "qty" : 25 }
{ $arrayToObject: { $literal: [
    { "k": "item", "v": "123abc"},
    { "k": "item", "v": "abc123" }
] } }
                                                    { "item" : "abc123" }
```

从版本 4.0.5+(3.6.10+和 3.4.19+)开始，如果数组中重复出现字段名，$arrayToObject 将使用**该字段的最后一个值。**

E:inventory collection

```
{ "_id" : 1, "item" : "ABC1",   dimensions: [ { "k": "l", "v": 25} , { "k": "w", "v": 10 }, { "k": "uom", "v": "cm" } ] }
{ "_id" : 2, "item" : "ABC2",   dimensions: [ [ "l", 50 ], [ "w",   25 ], [ "uom", "cm" ] ] }
{ "_id" : 3, "item" : "ABC3",   dimensions: [ [ "l", 25 ], [ "l",   "cm" ], [ "l", 50 ] ] }
```

下面的聚合管道操作使用$arrayToObject 将 dimensions 字段作为文档返回:

```
db.inventory.aggregate(
    [
        {
            $project: {
                item: 1,
                dimensions: { $arrayToObject: "$dimensions" }
            }
        }
    ]
)
```

R:$arrayToObject 将使用**该字段的最后一个值。**

```
{ "_id" : 1, "item" : "ABC1", "dimensions" : { "l" : 25, "w" : 10, "uom" : "cm" } }
{ "_id" : 2, "item" : "ABC2", "dimensions" : { "l" : 50, "w" : 25, "uom" : "cm" } }
{ "_id" : 3, "item" : "ABC3", "dimensions" : { "l" : 50 } }
```

从版本 4.0.5+(3.6.10+和 3.4.19+)开始，如果数组中重复出现字段名，$arrayToObject 将使用该字段的最后一个值。

$objectToArray + $arrayToObject 示例实施

考虑以下列文件收集存货:

```
{ "_id" : 1, "item" : "ABC1", instock: { warehouse1: 2500, warehouse2: 500 } }
{ "_id" : 2, "item" : "ABC2", instock: { warehouse2: 500, warehouse3: 200} }
```

下面的聚合管道操作计算每个项目的总库存，并添加到 instock 文档:

```
db.inventory.aggregate( [
```

```
    { $addFields: { instock: { $objectToArray: "$instock" } } },
    { $addFields: { instock: { $concatArrays: [ "$instock", [ { "k": "total", "v": { $sum: "$instock.v" } } ] ] } } },
    { $addFields: { instock: { $arrayToObject: "$instock" } } }
] )
```

R:

```
{ "_id" : 1, "item" : "ABC1", "instock" : { "warehouse1" : 2500, "warehouse2" : 500, "total" : 3000 } }
{ "_id" : 2, "item" : "ABC2", "instock" : { "warehouse2" : 500, "warehouse3" : 200, "total" : 700 } }
```

$avg:

返回数值的平均值。$avg 忽略非数值。

在 3.2 版本中进行了更改:$avg 可用于$group 和$project 阶段。在以前的 MongoDB 版本中，$avg 只在$group 阶段可用。

在$group 阶段使用$avg 时, $avg 具有以下语法, 并返回将指定的表达式应用于一组文档(按键共享同一组文档)中的每个文档所产生的所有数值的集合平均值:

```
{ $avg: <expression> }
```

$avg 在$project 阶段使用时，返回每个文档指定表达式或表达式列表的平均值，并具有以下两种语法之一:

$avg 有一个指定的表达式作为操作数:

```
{ $avg: <expression> }
```

$avg 有一个指定表达式列表作为操作数:

```
{ $avg: [ <expression1>, <expression2> ... ]  }
```

非数值或缺少值

$avg 忽略非数值值，包括丢失的值。如果平均值的所有操作数都是非数值的，$avg 返回 null，因为零值的平均值没有定义。

Array Operand 数组操作数

在$group 阶段，如果表达式解析为数组，$avg 将操作数视为非数值。

在$project 阶段:

以单个表达式作为操作数，如果表达式解析为数组，$avg 将遍历数组，对数组的数字元素进行操作，以返回单个值。

以表达式列表作为操作数，如果任何表达式解析为数组，$avg 不遍历数组，而是将数组视为非数值。

E:

Use in $group Stage

sales collection:

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
```

{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:12:00Z") }
按照 **item 字段对文档进行分组**, 下面的操作使用$avg 累加器计算每个分组的平均数量和平均数量。

```
db.sales.aggregate(
    [
      {
        $group:
          {
            _id: "$item",
            avgAmount: { $avg: { $multiply: [ "$price", "$quantity" ] } },
            avgQuantity: { $avg: "$quantity" }
          }
      }
    ]
)
```

R:

```
{ "_id" : "xyz", "avgAmount" : 37.5, "avgQuantity" : 7.5 }
{ "_id" : "jkl", "avgAmount" : 20, "avgQuantity" : 1 }
{ "_id" : "abc", "avgAmount" : 60, "avgQuantity" : 6 }
```

Use in $project Stage

collection students

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

下面的例子使用$project 阶段中的$avg 来计算测验的平均分数、实验室的平均分数、期末和期中考试的平均分数:

```
db.students.aggregate([
    {
      $project: {
        quizAvg: { $avg: "$quizzes"},
        labAvg: { $avg: "$labs" },
        examAvg: { $avg: [ "$final", "$midterm" ] }
      }
    }
])
```

R:

```
{ "_id" : 1, "quizAvg" : 7.666666666666667, "labAvg" : 6.5, "examAvg" : 77.5 }
{ "_id" : 2, "quizAvg" : 9.5, "labAvg" : 8, "examAvg" : 87.5 }
{ "_id" : 3, "quizAvg" : 4.666666666666667, "labAvg" : 5.5, "examAvg" : 74 }
```

在$project 阶段:

以单个表达式作为操作数，如果表达式解析为数组，$avg 将遍历数组，对数组的数字元素

进行操作，以返回单个值。
以表达式列表作为操作数，如果任何表达式解析为数组，$avg 不遍历数组，而是将数组视为非数值。

$ceil:

新版本 3.2。
返回大于或等于指定数字的最小整数。
$ceil 的语法如下:
{ $ceil: <number> }
<number>表达式可以是任何有效的表达式，只要它解析为一个数字。有关表达式的更多信息，请参见表达式。
行为
如果参数解析为 null 值或引用缺失的字段，$ceil 返回 null。如果参数解析为 NaN， $ceil 返回 NaN。
Example Results
{ $ceil: 1 } 1
{ $ceil: 7.80 } 8
{ $ceil: -2.8 } -2
E:
{ _id: 1, value: 9.25 }
{ _id: 2, value: 8.73 }
{ _id: 3, value: 4.32 }
{ _id: 4, value: -5.34 }
下面的示例返回原始值和上限值:

```
db.samples.aggregate([
    { $project: { value: 1, ceilingValue: { $ceil: "$value" } } }
])
```

R:

```
{ "_id" : 1, "value" : 9.25, "ceilingValue" : 10 }
{ "_id" : 2, "value" : 8.73, "ceilingValue" : 9 }
{ "_id" : 3, "value" : 4.32, "ceilingValue" : 5 }
{ "_id" : 4, "value" : -5.34, "ceilingValue" : -5 }
```

$cmp

比较两个值和返回值:
-1 如果第一个值小于第二个值。
1 如果第一个值大于第二个值。
0 如果这两个值相等。
$cmp 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。

$cmp 的语法如下:

{ $cmp: [<expression1>, <expression2>] }
E:
inventory collection
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
下面的操作使用$cmp 操作符将 qty 值与 250 进行比较:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
            qty: 1,
            cmpTo250: { $cmp: [ "$qty", 250 ] },
            _id: 0
          }
      }
    ]
)
```

R:

```
{ "item" : "abc1", "qty" : 300, "cmpTo250" : 1 }
{ "item" : "abc2", "qty" : 200, "cmpTo250" : -1 }
{ "item" : "xyz1", "qty" : 250, "cmpTo250" : 0 }
{ "item" : "VWZ1", "qty" : 300, "cmpTo250" : 1 }
{ "item" : "VWZ2", "qty" : 180, "cmpTo250" : -1 }
```

$concat:

连接字符串并返回连接后的字符串。
$concat 的语法如下:
{ $concat: [<expression1>, <expression2>, ...] }
参数可以是任何有效的表达式，只要它们解析为字符串。有关表达式的更多信息，请参见表达式。
如果参数解析为 null 值或引用缺失的字段，$concat 返回 null。
E:
inventory collection:
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
下面的操作使用$concat 操作符将 item 字段和 description 字段用“-”分隔符连接起来。

```
db.inventory.aggregate(
```

```
    [
        { $project: { itemDescription: { $concat: [ "$item", " - ", "$description" ] } } }
    ]
)
```

R:

```
{ "_id" : 1, "itemDescription" : "ABC1 - product 1" }
{ "_id" : 2, "itemDescription" : "ABC2 - product 2" }
{ "_id" : 3, "itemDescription" : null }
```

$concatArrays:

新版本 3.2。

连接数组以返回连接的数组。

$concatArrays 有以下语法

```
{ $concatArrays: [ <array1>, <array2>, ... ] }
```

<array>表达式可以是任何有效的表达式，只要它们解析为一个数组。有关表达式的更多信息，请参见表达式。

如果任何参数解析为 null 值或引用缺失的字段，$concatArrays 返回 null。

```
{ $concatArrays: [
    [ "hello", " "], [ "world" ]
] }
                                    [ "hello", " ", "world" ]
{ $concatArrays: [
    [ "hello", " "],
    [ [ "world" ], "again"]
] }
                                    [ "hello", " ", [ "world" ], "again" ]
```

E:

collection named warehouses

```
{ "_id" : 1, instock: [ "chocolate" ], ordered: [ "butter", "apples" ] }
{ "_id" : 2, instock: [ "apples", "pudding", "pie" ] }
{ "_id" : 3, instock: [ "pears", "pecans"], ordered: [ "cherries" ] }
{ "_id" : 4, instock: [ "ice cream" ], ordered: [ ] }
```

下面的例子连接了 instock 和 ordered 数组:

```
db.warehouses.aggregate([
    { $project: { items: { $concatArrays: [ "$instock", "$ordered" ] } } }
])
```

R:

```
{ "_id" : 1, "items" : [ "chocolate", "butter", "apples" ] }
{ "_id" : 2, "items" : null }
{ "_id" : 3, "items" : [ "pears", "pecans", "cherries" ] }
{ "_id" : 4, "items" : [ "ice cream" ] }
```

$cond

计算布尔表达式以返回指定的两个返回表达式之一。
$cond 表达式有两种语法:
新版本 2.6。
{ $cond: { if: <boolean-expression>, then: <true-case>, else: <false-case-> } }
OR:
{ $cond: [<boolean-expression>, <true-case>, <false-case>] }
对于任何一种语法，$cond 都需要所有三个参数(if-then-else)。
如果<boolean-expression>的值为 true, 那么$cond 将计算并返回<true-case>表达式的值。
否则，$cond 将计算并返回<false-case>表达式的值。
参数可以是任何有效的表达式。有关表达式的更多信息，请参见表达式。
$switch
E:
inventory collection
{ "_id" : 1, "item" : "abc1", qty: 300 }
{ "_id" : 2, "item" : "abc2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", qty: 250 }
下面的聚合操作使用$cond 表达式在 qty 值大于或等于 250 时将折扣值设置为 30，在 qty 值小于 250 时将折扣值设置为 20:

```
db.inventory.aggregate(
    [
        {
            $project:
              {
                item: 1,
                discount:
                  {
                    $cond: { if: { $gte: [ "$qty", 250 ] }, then: 30, else: 20 }
                  }
              }
        }
    ]
)
```

R:
{ "_id" : 1, "item" : "abc1", "discount" : 30 }
{ "_id" : 2, "item" : "abc2", "discount" : 20 }
{ "_id" : 3, "item" : "xyz1", "discount" : 30 }
下面的操作使用$cond 表达式的数组语法，并返回相同的结果:

```
db.inventory.aggregate(
    [
        {
            $project:
              {
```

```
                item: 1,
                discount:
                  {
                    $cond: [ { $gte: [ "$qty", 250 ] }, 30, 20 ]
                  }
              }
        }
    ]
)
```

$convert

新版本 4.0。
将值转换为指定的类型。
$convert 有以下语法:

```
{
    $convert:
        {
            input: <expression>,
            to: <type expression>,
            onError: <expression>,    // Optional.
            onNull: <expression>       // Optional.
        }
}
```

$convert 接受具有以下字段的文档:
字段描述
Input　　参数可以是任何有效的表达式。有关表达式的更多信息，请参见表达式。
to　　参数可以是任何解析为以下数字或字符串标识符之一的有效表达式:
字符串标识符、数字标识符
“double”　　1 有关转换为 double 的更多信息，请参见转换为 double。
“string”　　2 有关转换为字符串的更多信息，请参见转换为字符串。
“objectid”　　7 有关转换为 objectId 的更多信息，请参见转换为 objectId。
“bool”　　8 有关转换为布尔值的更多信息，请参见转换为布尔值。
“date”　　9 有关转换为日期的更多信息，请参见转换为日期。
“int”　　16 有关转换为整数的更多信息，请参见转换为整数。
“long”　　18 有关将“long”转换为 long 的更多信息，请参见将“long”转换为 long。
“decimal”　　19 有关十进制转换的更多信息，请参见十进制转换。

onError　　可选的。在转换期间遇到错误时返回的值，包括不支持的类型转换。参数可以是任何有效的表达式。
如果未指定，操作将在遇到错误时抛出错误并停止。
onNull　　可选的。如果输入为空或缺失，返回的值。参数可以是任何有效的表达式.如果未指定，如果输入为空或缺失，$convert 返回 null。
除了$convert，当默认的“onError”和“onNull”行为可以接受时，MongoDB 还提供以下

聚合操作符作为简写:
$toBool
$toDate
$toDecimal
$toDouble
$toInt
$toLong
$toObjectId
$toString
行为
转换为布尔值
下表列出了可以转换成布尔值的输入类型:

Boolean	No-op . Returns the boolean value.
Double	Returns true if not zero. Return false if zero.
Decimal	Returns true if not zero. Return false if zero.
Integer	Returns true if not zero. Return false if zero.
Long	Returns true if not zero. Return false if zero.
ObjectId	Returns true.
String	returns true.
Date	Returns true.

Converting to a Boolean:
下表列出了一些转换为布尔型的例子:

Example	Results
{ input: true, to: "bool"}	true
{ input: false, to: "bool" }	false
{ input: 1.99999, to: "bool" }	true
{ input: NumberDecimal("5"), to: "bool"}	true
{ input: NumberDecimal("0"), to: "bool"}	false
{ input: 100, to: "bool" }	true
{ input: ISODate("2018-03-26T04:38:28.044Z"), to: "bool" }	true
{ input: "hello", to: "bool" }	true
{ input: "false", to: "bool" }	true
{ input: "", to: "bool" }	true
{ input: null, to: "bool" }	null

Converting to a Decimal
下表列出了可以转换成小数的输入类型:
{ input: true, to: "decimal"} NumberDecimal(“1”)

{ input: false, to: "decimal" } NumberDecimal(“0”)
{ input: 2.5, to: "decimal" } NumberDecimal(“2.50000000000000”)
{ input: NumberInt(5), to: "decimal"} NumberDecimal(“5”)
{ input: NumberLong(10000), to: "decimal"} NumberDecimal(“10000”)
{ input: "-5.5", to: "decimal" } NumberDecimal(“-5.5”)
{ input: ISODate("2018-03-27T05:04:47.890Z"), to: "decimal" }:
NumberDecimal(“1522127087890”)

Converting to a Double

{ input: true, to: "double"} 1
{ input: false, to: "double" } 0
{ input: 2.5, to: "double" } 2.5
{ input: NumberInt(5), to: "double"} 5
{ input: NumberLong(10000), to: "double"} 10000
{ input: "-5.5", to: "double" } -5.5
{ input: "5e10", to: "double" } 50000000000
{
  input: "5e550",
  to: "double",
  onError: "Could not convert to type double."
} “Could not convert to type double.”
{ input: ISODate("2018-03-27T05:04:47.890Z"), to: "double" } 1522127087890

Converting to a Long

{ input: true, to: "long" } NumberLong(“1”)
{ input: false, to: "long" } NumberLong(“0”)
{ input: 1.99999, to: "long" } NumberLong(“1”)
{ input: NumberDecimal("5.5000"), to: "long" } NumberLong(“5”)
{ input: NumberDecimal("9223372036854775808.0"), to: "long" } Error
{
  input: NumberDecimal("9223372036854775808.000"),
  to: "long",
  onError: "Could not convert to type long."
} “Could not convert to type long.”

{ input: NumberInt(8), to: "long" } NumberLong(8):
{ input: ISODate("2018-03-26T04:38:28.044Z"), to: "long" }

{ input: "-2", to: "long" } NumberLong(“-2”)
{ input: "2.5", to: "long" } Error
{ input: null, to: "long" } null

Converting to a Date

{ input: 120000000000.5, to: "date"} ISODate(“1973-10-20T21:20:00Z”)

{ input: NumberDecimal("1253372036000.50"), to: "date"}:
ISODate(“2009-09-19T14:53:56Z”)

{ input: NumberLong("1100000000000"), to: "date"}
ISODate(“2004-11-09T11:33:20Z”)

{ input: NumberLong("-1100000000000"), to: "date"}
ISODate(“1935-02-22T12:26:40Z”)

{ input: ObjectId("5ab9c3da31c2ab715d421285"), to: "date" }
ISODate(“2018-03-27T04:08:58Z”)

{ input: "2018-03-03", to: "date" }
ISODate(“2018-03-03T00:00:00Z”)

{ input: "2018-03-20 11:00:06 +0500", to: "date" }
ISODate(“2018-03-20T06:00:06Z”)

{ input: "Friday", to: "date" } Error

```
{
    input: "Friday",
    to: "date",
    onError: "Could not convert to type date."
}
```

“Could not convert to type date.”

Converting to a String

{ input: true, to: "string" } “true”

{ input: false, to: "string" } “false”

{ input: 2.5, to: "string"} “2.5”

{ input: NumberInt(2), to: "string"} “2”

{ input: NumberLong(1000), to: "string"} “1000”

{ input: ObjectId("5ab9c3da31c2ab715d421285"), to: "string" }
“5ab9c3da31c2ab715d421285”

{ input: ISODate("2018-03-27T16:58:51.538Z"), to: "string" }
“2018-03-27T16:58:51.538Z”

E:

SEE ALSO

$toString operator. $dateToString

E:collection orders

```
db.orders.insert( [
    { _id: 1, item: "apple", qty: 5, price: 10 },
    { _id: 2, item: "pie", qty: 10, price: NumberDecimal("20.0") },
    { _id: 3, item: "ice cream", qty: 2, price: "4.99" },
    { _id: 4, item: "almonds" },
    { _id: 5, item: "bananas", qty: 5000000000, price: NumberDecimal("1.25") }
] )
```

订单集合的以下聚合操作将价格转换为小数:

```
//定义阶段，使用转换后的价格和 qty 值添加 convertedPrice 和 convertedQty 字段
//如果缺少 price 或 qty 值，则转换返回一个 decimal 值或 int 值为 0 的值。
//如果不能转换 price 或 qty 值，则转换返回一个字符串
priceQtyConversionStage = {
    $addFields: {
        convertedPrice: { $convert: { input: "$price", to: "decimal", onError: "Error", onNull:
NumberDecimal("0") } },
        convertedQty: { $convert: {
            input: "$qty", to: "int",
            onError:{$concat:["Could not convert ", {$toString:"$qty"}, " to type integer."]},
            onNull: NumberInt("0")
        } },
    }
};
totalPriceCalculationStage = {
    $project: { totalPrice: {
       $switch: {
           branches: [
              { case: { $eq: [ { $type: "$convertedPrice" }, "string" ] }, then: "NaN" },
              { case: { $eq: [ { $type: "$convertedQty" }, "string" ] }, then: "NaN" },
           ],
           default: { $multiply: [ "$convertedPrice", "$convertedQty" ] }
       }
} } };

db.orders.aggregate( [
    priceQtyConversionStage,
    totalPriceCalculationStage
])
```

R:

```
{ "_id" : 1, "totalPrice" : NumberDecimal("50.0000000000000") }
{ "_id" : 2, "totalPrice" : NumberDecimal("200.0") }
{ "_id" : 3, "totalPrice" : NumberDecimal("9.98") }
{ "_id" : 4, "totalPrice" : NumberDecimal("0") }
{ "_id" : 5, "totalPrice" : "NaN" }
```

$dateFromParts:

新版本 3.6。

给定日期的组成属性构造并返回日期对象。

$dateFromParts 表达式有以下语法:

```
{
    $dateFromParts : {
        'year': <year>, 'month': <month>, 'day': <day>,
        'hour': <hour>, 'minute': <minute>, 'second': <second>,
        'millisecond': <ms>, 'timezone': <tzExpression>
    }
}
```

您还可以使用以下语法以 ISO 周日期格式指定您的组成日期字段:

```
{
    $dateFromParts : {
        'isoWeekYear': <year>, 'isoWeek': <week>, 'isoDayOfWeek': <day>,
        'hour': <hour>, 'minute': <minute>, 'second': <second>,
        'millisecond': <ms>, 'timezone': <tzExpression>
    }
}
```

$dateFromParts 接受一个具有以下字段的文档:

重要的

在构造$dateFromParts 输入文档时，不能结合使用日历日期和 ISO 周日期字段。

行为

值范围

从 MongoDB 4.0 开始，如果为 year、isoYear 和时区以外的字段指定的值超出了有效范围，$dateFromParts 将携带或减去与其他日期部分的差值来计算日期。

值大于范围

考虑下面的$dateFromParts 表达式，其中 month 字段值为 14，比 12 个月(或 1 年)的最大值大 2 个月:

```
{ $dateFromParts: { 'year' : 2017, 'month' : 14, 'day': 1, 'hour' : 12   } }
```

表达式通过将年份增加 1 并将月份设置为 2 来计算日期:

```
ISODate("2018-02-01T12:00:00Z")
```

值小于范围

考虑下面的$dateFromParts 表达式，其中 month 字段值为 0，比 1 个月的最小值小 1 个月:

```
{ $dateFromParts: { 'year' : 2017, 'month' : 0, 'day': 1, 'hour' : 12   } }
```

表达式通过将年份减少 1，并将月份设置为 12 来计算日期:

```
ISODate("2016-12-01T12:00:00Z")
```

时区

当在< Timezone >字段中使用 Olson 时区标识符时，MongoDB 应用 DST 偏移量(如果适用于指定的时区)。

例如，考虑一个包含以下文档的 sales 集合:

```
{
    "_id" : 1,
    "item" : "abc",
    "price" : 20,
    "quantity" : 5,
    "date" : ISODate("2017-05-20T10:24:51.303Z")
}
```

下面的聚合演示了 MongoDB 如何处理 Olson 时区标识符的 DST 偏移量。该示例使用$hour和$minute 操作符返回 date 字段的相应部分:

```
db.sales.aggregate([
{
    $project: {
        "nycHour": {
            $hour: { date: "$date", timezone: "-05:00" }
         },
         "nycMinute": {
            $minute: { date: "$date", timezone: "-05:00" }
         },
         "gmtHour": {
            $hour: { date: "$date", timezone: "GMT" }
         },
         "gmtMinute": {
            $minute: { date: "$date", timezone: "GMT" } },
         "nycOlsonHour": {
            $hour: { date: "$date", timezone: "America/New_York" }
         },
         "nycOlsonMinute": {
            $minute: { date: "$date", timezone: "America/New_York" }
         }
    }
}])
```

R:

```
{
    "_id": 1,
    "nycHour" : 5,
    "nycMinute" : 24,
    "gmtHour" : 10,
    "gmtMinute" : 24,
    "nycOlsonHour" : 6,
    "nycOlsonMinute" : 24
}
```

E:

下面的聚合使用$dateFromParts 从提供的输入字段构造三个日期对象:

```
db.sales.aggregate([
{
    $project: {
        date: {
            $dateFromParts: {
                'year' : 2017, 'month' : 2, 'day': 8, 'hour' : 12
            }
        },
        date_iso: {
            $dateFromParts: {
                'isoWeekYear' : 2017, 'isoWeek' : 6, 'isoDayOfWeek' : 3, 'hour' : 12
            }
        },
        date_timezone: {
            $dateFromParts: {
                'year' : 2016, 'month' : 12, 'day' : 31, 'hour' : 23,
                'minute' : 46, 'second' : 12, 'timezone' : 'America/New_York'
            }
        }
    }
}])
```

R:

```
{
   "_id" : 1,
   "date" : ISODate("2017-02-08T12:00:00Z"),
   "date_iso" : ISODate("2017-02-08T12:00:00Z"),
   "date_timezone" : ISODate("2017-01-01T04:46:12Z")
}
```

$dateToParts:

新版本 3.6。

返回一个文档，该文档包含作为单个属性的给定 BSON 日期值的组成部分。返回的属性是年、月、日、小时、分钟、秒和毫秒。

您可以将 iso8601 属性设置为 true，以返回表示 ISO 周日期的部件。这将返回一个属性为 isoWeekYear、isoWeek、isoDayOfWeek、hour、minute、second 和 ond 的文档。

$ datetopart 表达式的语法如下:

```
{
     $dateToParts: {
          'date' : <dateExpression>,
          'timezone' : <timezone>,
          'iso8601' : <boolean>
     }
}
```

date Required
在 3.6 版中进行了更改。
返回零件的输入日期。<dateExpression>可以是任何解析为日期、时间戳或 ObjectID 的表达式。有关表达式的更多信息，请参见表达式。
Timezone Optional
用于格式化日期的时区。默认情况下，$ datetopart 使用 UTC。
<timezone>可以是任何计算为字符串的表达式，其值可以是:
Olson 时区标识符，如“欧洲/伦敦”或“美国/纽约”，或
表格中的 UTC 偏移量:
+ / - (hh): (mm)。“+ 04:45”或
+ / - (hh)(毫米),例如:“-0530”,或
+ / - (hh)。“+ 03”。
有关表达式的更多信息，请参见表达式。
iso8601 Optional
如果设置为 true，则修改输出文档以使用 ISO 周日期字段。默认值为 false。
Behavior:
当在< Timezone >字段中使用 Olson 时区标识符时，MongoDB 应用 DST 偏移量(如果适用于指定的时区)。
例如，考虑一个包含以下文档的 sales 集合:

```
{
    "_id" : 1,
    "item" : "abc",
    "price" : 20,
    "quantity" : 5,
    "date" : ISODate("2017-05-20T10:24:51.303Z")
}
```

下面的聚合演示了 MongoDB 如何处理 Olson 时区标识符的 DST 偏移量。该示例使用$hour 和$minute 操作符返回 date 字段的相应部分:

```
db.sales.aggregate([
{
    $project: {
        "nycHour": {
            $hour: { date: "$date", timezone: "-05:00" }
        },
        "nycMinute": {
            $minute: { date: "$date", timezone: "-05:00" }
        },
        "gmtHour": {
            $hour: { date: "$date", timezone: "GMT" }
        },
        "gmtMinute": {
            $minute: { date: "$date", timezone: "GMT" } },
        "nycOlsonHour": {
            $hour: { date: "$date", timezone: "America/New_York" }
```

```
        },
        "nycOlsonMinute": {
            $minute: { date: "$date", timezone: "America/New_York" }
        }
    }
}])
```

R:

```
{
    "_id": 1,
    "nycHour" : 5,
    "nycMinute" : 24,
    "gmtHour" : 10,
    "gmtMinute" : 24,
    "nycOlsonHour" : 6,
    "nycOlsonMinute" : 24
}
```

E:sales collection

```
{
   "_id" : 2,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2017-01-01T01:29:09.123Z")
}
```

下面的聚合使用$dateToParts 返回包含 date 字段组成部分的文档。

```
db.sales.aggregate([
 {
     $project: {
         date: {
             $dateToParts: { date: "$date" }
         },
         date_iso: {
             $dateToParts: { date: "$date", iso8601: true }
         },
         date_timezone: {
             $dateToParts: { date: "$date", timezone: "America/New_York" }
         }
     }
}])
```

R:

```
{
    "_id" : 2,
    "date" : {
        "year" : 2017,
```

```
        "month" : 1,
        "day" : 1,
        "hour" : 1,
        "minute" : 29,
        "second" : 9,
        "millisecond" : 123
    },
    "date_iso" : {
        "isoWeekYear" : 2016,
        "isoWeek" : 52,
        "isoDayOfWeek" : 7,
        "hour" : 1,
        "minute" : 29,
        "second" : 9,
        "millisecond" : 123
    },
    "date_timezone" : {
        "year" : 2016,
        "month" : 12,
        "day" : 31,
        "hour" : 20,
        "minute" : 29,
        "second" : 9,
        "millisecond" : 123
    }
}
```

$dateFromString:

新版本 3.6。

将日期/时间字符串转换为日期对象。

$dateFromString 表达式有以下语法:

```
{ $dateFromString: {
        dateString: <dateStringExpression>,
        format: <formatStringExpression>,
        timezone: <tzExpression>,
        onError: <onErrorExpression>,
        onNull: <onNullExpression>
} }
```

Specifiers	Description	Possible Values

Specifiers	Description	Possible Values
%d	Day of Month (2 digits, zero padded)	01–31
%G	Year in ISO 8601 format	0000–9999
%H	Hour (2 digits, zero padded, 24–hour clock)	00–23
%L	Millisecond (3 digits, zero padded)	000–999
%m	Month (2 digits, zero padded)	01–12
%M	Minute (2 digits, zero padded)	00–59
%S	Second (2 digits, zero padded)	00–60
%u	Day of week number in ISO 8601 format (1–Monday, 7–Sunday)	1–7
%V	Week of Year in ISO 8601 format	1–53

Specifiers	Description	Possible Values
%Y	Year (4 digits, zero padded)	0000–9999
%z	The timezone offset from UTC.	+/–[hh][mm]
%Z	The minutes offset from UTC as a number. For example, if the timezone offset (+/–[hhmm]) was 聽+0445, the minutes offset is 聽+285.	+/–mmm
%%	Percent Character as a Literal	

E:collection logmessages
{ _id: 1, date: "2017-02-08T12:10:40.787", timezone: "America/New_York", message: "Step 1: Started" },
{ _id: 2, date: "2017-02-08", timezone: "-05:00", message: "Step 1: Ended" },
{ _id: 3, message: " Step 1: Ended " },
{ _id: 4, date: "2017-02-09", timezone: "Europe/London", message: "Step 2: Started"}
{ _id: 5, date: "2017-02-09T03:35:02.055", timezone: "+0530", message: "Step 2: In Progress"}
下面的聚合使用$dateFromString 将日期值转换为日期对象:

```
db.logmessages.aggregate( [ {
    $project: {
        date: {
            $dateFromString: {
                dateString: '$date',
                timezone: 'America/New_York'
            }
        }
    }
} ] )
```

上述聚合返回以下文档，并将每个日期字段转换为东部时区:

```
{ "_id" : 1, "date" : ISODate("2017-02-08T17:10:40.787Z") }
{ "_id" : 2, "date" : ISODate("2017-02-08T05:00:00Z") }
{ "_id" : 3, "date" : null }
{ "_id" : 4, "date" : ISODate("2017-02-09T05:00:00Z") }
{ "_id" : 5, "date" : ISODate("2017-02-09T08:35:02.055Z") }
```

时区参数也可以通过文档字段而不是硬编码的参数提供。例如:

```
db.logmessages.aggregate( [ {
    $project: {
        date: {
            $dateFromString: {
                dateString: '$date',
                timezone: '$timezone'
            }
        }
    }
} ] )
```

上述聚合返回以下文档，并将每个日期字段转换为各自的 UTC 表示。

```
{ "_id" : 1, "date" : ISODate("2017-02-08T17:10:40.787Z") }
{ "_id" : 2, "date" : ISODate("2017-02-08T05:00:00Z") }
{ "_id" : 3, "date" : null }
{ "_id" : 4, "date" : ISODate("2017-02-09T00:00:00Z") }
{ "_id" : 5, "date" : ISODate("2017-02-08T22:05:02.055Z") }
```

onError

如果集合包含具有不可解析日期字符串的文档，则$dateFromString 将抛出一个错误，除非为可选的 onError 参数提供聚合表达式。

例如，给定一个包含以下文档的集合日期:

```
{ "_id" : 1, "date" : "2017-02-08T12:10:40.787", timezone: "America/New_York" },
{ "_id" : 2, "date" : "20177-02-09T03:35:02.055", timezone: "America/New_York" }
```

您可以使用 onError 参数返回原始字符串形式的无效日期:

```
db.dates.aggregate( [ {
    $project: {
        date: {
            $dateFromString: {
                dateString: '$date',
                timezone: '$timezone',
                onError: '$date'
            }
        }
    }
} ] )
```

R:

```
{ "_id" : 1, "date" : ISODate("2017-02-08T17:10:40.787Z") }
{ "_id" : 2, "date" : "20177-02-09T03:35:02.055" }
```

onNull
如果集合包含具有 null 日期字符串的文档，$dateFromString 将返回 null，除非为可选的 onNull 参数提供聚合表达式。
例如，给定一个包含以下文档的集合日期:
{ "_id" : 1, "date" : "2017-02-08T12:10:40.787", timezone: "America/New_York" },
{ "_id" : 2, "date" : null, timezone: "America/New_York" }
您可以使用 onNull 参数让$dateFromString 返回一个表示 unix 历元的日期，而不是 null:

```
db.dates.aggregate( [ {
    $project: {
        date: {
            $dateFromString: {
                dateString: '$date',
                timezone: '$timezone',
                onNull: new Date(0)
            }
        }
    }
} ] )
```

R:
{ "_id" : 1, "date" : ISODate("2017-02-08T17:10:40.787Z") }
{ "_id" : 2, "date" : ISODate("1970-01-01T00:00:00Z") }

$dateToString

新版本 3.0。
根据用户指定的格式将日期对象转换为字符串。
$dateToString 表达式具有以下操作符表达式语法:

```
{ $dateToString: {
      date: <dateExpression>,
      format: <formatString>,
      timezone: <tzExpression>,
      onNull: <expression>
} }
```

E:sales colletion

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$dateToString 以格式化字符串的形式返回日期字段:

```
db.sales.aggregate(
    [
```

```
        {
          $project: {
              yearMonthDayUTC:  {  $dateToString:  {  format:  "%Y-%m-%d",  date:
"$date" } },
              timewithOffsetNY:  {  $dateToString:  {  format:  "%H:%M:%S:%L%z",  date:
"$date", timezone: "America/New_York"} },
              timewithOffset430:  {  $dateToString:  {  format:  "%H:%M:%S:%L%z",  date:
"$date", timezone: "+04:30" } },
              minutesOffsetNY: { $dateToString: { format: "%Z", date: "$date", timezone:
"America/New_York" } },
              minutesOffset430: { $dateToString: { format: "%Z", date: "$date", timezone:
"+04:30" } }
          }
        }
    ]
)
R:
{
    "_id" : 1,
    "yearMonthDayUTC" : "2014-01-01",
    "timewithOffsetNY" : "03:15:39:736-0500",
    "timewithOffset430" : "12:45:39:736+0430",
    "minutesOffsetNY" : "-300",
    "minutesOffset430" : "270"
}
```

$dayOfMonth

$dayOfWeek

$dayOfYear

以 1 到 31 之间的数字返回月份的日期。
以 1(星期日)到 7(星期六)之间的数字返回日期的星期几。
以 1 到 366 之间的数字返回日期的年份。
$dayOfMonth 表达式有以下操作符表达式语法:
{ $dayOfMonth: <dateExpression> }
在 3.6 版中进行了更改。
该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。

{ date: <dateExpression>, timezone: <tzExpression> }

E: sales collection

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$dayOfMonth 和其他日期操作符来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

R:

```
{
   "_id" : 1,
   "year" : 2014,
   "month" : 1,
   "day" : 1,
   "hour" : 8,
   "minutes" : 15,
   "seconds" : 39,
   "milliseconds" : 736,
   "dayOfYear" : 1,
   "dayOfWeek" : 4,
   "week" : 0
}
```

$divide

将一个数字除以另一个数字并返回结果。将参数传递给数组中的$divide。
$divide 表达式的语法如下:
{ $divide: [<expression1>, <expression2>] }
第一个参数是被除数，第二个参数是除数;即第一个参数除以第二个参数。
参数可以是任何有效的表达式，只要它们解析为数字。有关表达式的更多信息，请参见表达式。
E:planning collection
{ "_id" : 1, "name" : "A", "hours" : 80, "resources" : 7 },
{ "_id" : 2, "name" : "B", "hours" : 40, "resources" : 4 }
下面的聚合使用$divide 表达式将 hours 字段除以一个文字 8 来计算工作日的数量:

```
db.planning.aggregate(
    [
      { $project: { name: 1, workdays: { $divide: [ "$hours", 8 ] } } }
    ]
)
```

R:
{ "_id" : 1, "name" : "A", "workdays" : 10 }
{ "_id" : 2, "name" : "B", "workdays" : 5 }

$eq:

比较两个值和返回值:
当值相等时为真。
当值不相等时为 false。
$eq 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。
$eq 的语法如下:
{ $eq: [<expression1>, <expression2>] }
参数可以是任何有效的表达式。有关表达式的更多信息，请参见表达式。
例子 inventory collection
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
下面的操作使用$eq 操作符来确定 qty 是否等于 250:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
```

```
                qty: 1,
                qtyEq250: { $eq: [ "$qty", 250 ] },
                _id: 0
            }
        }
    ]
)
```

R:

```
{ "item" : "abc1", "qty" : 300, "qtyEq250" : false }
{ "item" : "abc2", "qty" : 200, "qtyEq250" : false }
{ "item" : "xyz1", "qty" : 250, "qtyEq250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtyEq250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyEq250" : false }
```

$exp

新版本 3.2。

将欧拉数(即 e)提高到指定的指数并返回结果。

$exp 的语法如下:

{ $exp: <exponent> }

<exponent>表达式可以是任何有效的表达式，只要它解析为一个数字。有关表达式的更多信息，请参见表达式。

Behavior:

如果参数解析为 null 值或引用缺失的字段，$exp 返回 null。如果参数解析为 NaN， $exp 返回 NaN。

{ $exp: 0 } 1

{ $exp: 2 } 7.38905609893065

{ $exp: –2 } 0.1353352832366127

A collection named accounts

```
{ _id: 1, rate: .08, pv: 10000 }
{ _id: 2, rate: .0825, pv: 250000 }
{ _id: 3, rate: .0425, pv: 1000 }
```

下面的例子计算连续复利的有效利率:

```
db.accounts.aggregate( [ { $project: { effectiveRate: { $subtract: [ { $exp: "$rate"}, 1 ] } } } ] )
```

R:
{ "_id" : 1, "effectiveRate" : 0.08328706767495864 }
{ "_id" : 2, "effectiveRate" : 0.08599867343905654 }
{ "_id" : 3, "effectiveRate" : 0.04341605637367807 }

$filter

New in 3.2
根据指定的条件选择要返回的数组子集。返回一个数组，其中只包含与条件匹配的元素。返回的元素按原始顺序排列。
$filter 有如下几个语法：
{ $filter: { input: <array>, as: <string>, cond: <expression> } }
E:

```
{
   $filter: {
       input: [ 1, "a", 2, null, 3.1, NumberLong(4), "5" ],
       as: "num",
       cond: { $and: [
           { $gte: [ "$$num", NumberLong("-9223372036854775807") ] },
           { $lte: [ "$$num", NumberLong("9223372036854775807") ] }
        ] }
   }
}
```

R:
[1, 2, 3.1, NumberLong(4)]

E:Sales collection:

```
{
    _id: 0,
    items: [
       { item_id: 43, quantity: 2, price: 10 },
       { item_id: 2, quantity: 1, price: 240 }
    ]
}
{
    _id: 1,
    items: [
       { item_id: 23, quantity: 3, price: 110 },
       { item_id: 103, quantity: 4, price: 5 },
       { item_id: 38, quantity: 1, price: 300 }
    ]
}
{
     _id: 2,
```

```
    items: [
        { item_id: 4, quantity: 1, price: 23 }
    ]
}
```

下面的示例筛选 items 数组，使其只包含价格大于或等于 100 的文档:

E:

```
db.sales.aggregate([
    {
        $project: {
            items: {
                $filter: {
                    input: "$items",
                    as: "item",
                    cond: { $gte: [ "$$item.price", 100 ] }
                }
            }
        }
    }
])
```

R:

```
{
    "_id" : 0,
    "items" : [
        { "item_id" : 2, "quantity" : 1, "price" : 240 }
    ]
}
{
    "_id" : 1,
    "items" : [
        { "item_id" : 23, "quantity" : 3, "price" : 110 },
        { "item_id" : 38, "quantity" : 1, "price" : 300 }
    ]
}
{ "_id" : 2, "items" : [ ] }
```

$first

返回将表达式应用于共享同一组 by 键的一组文档中的第一个文档所产生的值。只有当文档按定义的顺序排列时才有意义。

$first 只在$group 阶段可用。

$first 的语法如下:

```
{ $first: <expression> }
```

Behavior：

当在$group 阶段中首先使用$first 时，$group 阶段应该遵循$sort 阶段，以定义输入文档的

顺序。
虽然$sort 阶段将有序的文档作为输入传递给$group 阶段，但是$group 不能保证在自己的输出中保持这种排序顺序。
E:sales collection
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
{ "_id" : 6, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-15T12:05:10Z") }
{ "_id" : 7, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T14:12:12Z") }
按项目字段分组文档，下面的操作**使用$first 累加器计算每个项目的第一个销售日期:**

```
db.sales.aggregate(
    [
      { $sort: { item: 1, date: 1 } },
      {
        $group:
          {
            _id: "$item",
            firstSalesDate: { $first: "$date" }
          }
      }
    ]
)
```

R:
{ "_id" : "xyz", "firstSalesDate" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : "jkl", "firstSalesDate" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : "abc", "firstSalesDate" : ISODate("2014-01-01T08:00:00Z") }

$Floor

新版本 3.2。
返回小于或等于指定数字的最大整数。
$floor 有以下语法:
{ $floor: <number> }
<number>表达式可以是任何有效的表达式，只要它解析为一个数字。有关表达式的更多信息，请参见表达式。

Behavior
如果参数解析为 null 值或引用缺失的字段, 则$floor 返回 null。如果参数解析为 NaN，$floor 返回 NaN。
Example Results
{ $floor: 1 } 1
{ $floor: 7.80 } 7
{ $floor: -2.8 } -3
E:collection named samples
{ _id: 1, value: 9.25 }
{ _id: 2, value: 8.73 }
{ _id: 3, value: 4.32 }
{ _id: 4, value: -5.34 }
下面的示例返回原始值和下限值:

```
db.samples.aggregate([
    { $project: { value: 1, floorValue: { $floor: "$value" } } }
])
```

R:

```
{ "_id" : 1, "value" : 9.25, "floorValue" : 9 }
{ "_id" : 2, "value" : 8.73, "floorValue" : 8 }
{ "_id" : 3, "value" : 4.32, "floorValue" : 4 }
{ "_id" : 4, "value" : -5.34, "floorValue" : -6 }
```

$gt

比较两个值和返回值:
当第一个值大于第二个值时为真。
当第一个值小于或等于第二个值时为 false。
$gt 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。
$gt 的语法如下:

```
{ $gt: [ <expression1>, <expression2> ] }
```

E:inventory collection

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

下面的操作使用$gt 操作符来确定 qty 是否大于 250:

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            qty: 1,
```

```
            qtyGt250: { $gt: [ "$qty", 250 ] },
            _id: 0
        }
    }
]
)
```

R:

```
{ "item" : "abc1", "qty" : 300, "qtyGt250" : true }
{ "item" : "abc2", "qty" : 200, "qtyGt250" : false }
{ "item" : "xyz1", "qty" : 250, "qtyGt250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyGt250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyGt250" : false }
```

$gte

比较两个值和返回值:
当第一个值大于或等于第二个值时为真。
当第一个值小于第二个值时为 false。
$gte 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。

```
{ $gte: [ <expression1>, <expression2> ] }
```

E:inventory collection

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

下面的操作使用$gte 操作符来确定 qty 是否大于或等于 250:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
            qty: 1,
            qtyGte250: { $gte: [ "$qty", 250 ] },
            _id: 0
          }
      }
    ]
)
```

R:

```
{ "item" : "abc1", "qty" : 300, "qtyGte250" : true }
{ "item" : "abc2", "qty" : 200, "qtyGte250" : false }
{ "item" : "xyz1", "qty" : 250, "qtyGte250" : true }
```

{ "item" : "VWZ1", "qty" : 300, "qtyGte250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyGte250" : false }

$hour

以 0 到 23 之间的数字返回日期的小时部分。
$hour 表达式有以下操作符表达式语法:
{ $hour: <dateExpression> }
在 3.6 版中进行了更改。
该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。
{ date: <dateExpression>, timezone: <tzExpression> }
E:sales collection

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$hour 和其他日期表达式来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" }
          }
      }
    ]
)
```

R:

```
{
   "_id" : 1,
```

```
    "year" : 2014,
    "month" : 1,
    "day" : 1,
    "hour" : 8,
    "minutes" : 15,
    "seconds" : 39,
    "milliseconds" : 736,
    "dayOfYear" : 1,
    "dayOfWeek" : 4,
    "week" : 0
}
```

$ifnull

对表达式求值，如果表达式求值为非空值，则返回表达式的值。如果表达式计算为空值，包括未定义值或缺失字段的实例，则返回替换表达式的值。
$ifNull 表达式有以下语法:
{ $ifNull: [<expression>, <replacement-expression-if-null>] }
参数可以是任何有效的表达式。有关表达式的更多信息，请参见表达式。
E:inventory collection
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: null, qty: 200 }
{ "_id" : 3, "item" : "xyz1", qty: 250 }
下面的操作使用$ifNull 表达式返回非空描述字段值，如果描述字段为空或不存在，则返回字符串“Unspecified”：

```
db.inventory.aggregate(
    [
        {
            $project: {
                item: 1,
                description: { $ifNull: [ "$description", "Unspecified" ] }
            }
        }
    ]
)
```

R:

```
{ "_id" : 1, "item" : "abc1", "description" : "product 1" }
{ "_id" : 2, "item" : "abc2", "description" : "Unspecified" }
{ "_id" : 3, "item" : "xyz1", "description" : "Unspecified" }
```

$in:

新版本 3.4。

返回一个布尔值，指示指定值是否在数组中。
$in 有以下运算符表达式语法:
{ $in: [<expression>, <array expression>] }
{ $in: [2, [1, 2, 3]] } true
{ $in: ["abc", ["xyz", "abc"]] } true
{ $in: ["xy", ["xyz", "abc"]] } false
{ $in: [["a"], ["a"]] } false
{ $in: [["a"], [["a"]]] } true
{ $in: [/^a/, ["a"]] } false
{ $in: [/^a/, [/^a/]] } true
如果$in 表达式没有得到两个参数，或者第二个参数没有解析为一个数组，那么在以下情况之一中，$in 失败并出现错误。
E:collection fruit:
```
{ "_id" : 1, "location" : "24th Street",
   "in_stock" : [ "apples", "oranges", "bananas" ] }
{ "_id" : 2, "location" : "36th Street",
   "in_stock" : [ "bananas", "pears", "grapes" ] }
{ "_id" : 3, "location" : "82nd Street",
   "in_stock" : [ "cantaloupes", "watermelons", "apples" ] }
```
下面的聚合操作查看每个文档中的 in_stock 数组，并确定字符串 banana 是否存在。
```
db.fruit.aggregate([
  {
    $project: {
      "store location" : "$location",
      "has bananas" : {
        $in: [ "bananas", "$in_stock" ]
      }
    }
  }
])
```
R:
```
{ "_id" : 1, "store location" : "24th Street", "has bananas" : true }
{ "_id" : 2, "store location" : "36th Street", "has bananas" : true }
{ "_id" : 3, "store location" : "82nd Street", "has bananas" : false }
```

$indexOfArray:

新版本 3.4。
搜索数组中指定值的发生情况，并返回第一次发生的数组索引(从零开始)。如果没有找到该值，则返回-1。
$indexOfArray 有以下操作符表达式语法:
{ $indexOfArray: [<array expression>, <search expression>, <start>, <end>] }

{ $indexOfArray: [["a", "abc"], "a"] } 0
{ $indexOfArray: [["a", "abc", "de", ["de"]], ["de"]] } 3
{ $indexOfArray: [[1, 2], 5] } -1
{ $indexOfArray: [[1, 2, 3], [1, 2]] } -1
{ $indexOfArray: [[10, 9, 9, 8, 9], 9, 3] } 4
{ $indexOfArray: [["a", "abc", "b"], "b", 0, 1] } -1
{ $indexOfArray: [["a", "abc", "b"], "b", 1, 0] } -1
{ $indexOfArray: [["a", "abc", "b"], "b", 20] } -1
{ $indexOfArray: [[null, null, null], null] } 0
{ $indexOfArray: [null, "foo"] } null
{ $indexOfArray: ["foo", "foo"] }
E:inventory collection
{ "_id" : 1, "items" : ["one", "two", "three"] }
{ "_id" : 2, "items" : [1, 2, 3] }
{ "_id" : 3, "items" : [null, null, 2] }
{ "_id" : 4, "items" : null }
{ "_id" : 5, "amount" : 3 }
下面的操作使用$indexOfArray 操作符返回数组索引，其中字符串 foo 位于每个项目数组中:

```
db.inventory.aggregate(
    [
      {
        $project:
            {
              index: { $indexOfArray: [ "$items", 2 ] },
            }
       }
    ]
)
```

R:
{ "_id" : 1, "index" : "-1" }
{ "_id" : 2, "index" : "1" }
{ "_id" : 3, "index" : "2" }
{ "_id" : 4, "index" : null }
{ "_id" : 5, "index" : null }

$indexOfBytes

新版本 3.4。

搜索字符串以查找子字符串的发生情况，并返回第一次发生的 UTF-8 字节索引(从零开始)。
如果没有找到子字符串，则返回-1。
$indexOfBytes 有以下运算符表达式语法:
{ $indexOfBytes: [<string expression>, <substring expression>, <start>, <end>] }

{ $indexOfBytes: ["cafeteria", "e"] } 3
{ $indexOfBytes: ["cafétéria", "é"] } 3
{ $indexOfBytes: ["cafétéria", "e"] } -1
{ $indexOfBytes: ["cafétéria", "t"] }5
{ $indexOfBytes: ["foo.bar.fi", ".", 5] } 7
{ $indexOfBytes: ["vanilla", "ll", 0, 2] } -1
{ $indexOfBytes: ["vanilla", "ll", -1] } -1
{ $indexOfBytes: ["vanilla", "ll", 12] } -1
{ $indexOfBytes: ["vanilla", "ll", 5, 2] } -1
{ $indexOfBytes: ["vanilla", "nilla", 3] } -1
{ $indexOfBytes: [null, "foo"] }null
E:Inventory collection
{ "_id" : 1, "item" : "foo" }
{ "_id" : 2, "item" : "fóofoo" }
{ "_id" : 3, "item" : "the foo bar" }
{ "_id" : 4, "item" : "hello world fóo" }
{ "_id" : 5, "item" : null }
{ "_id" : 6, "amount" : 3 }
下面的操作使用$indexOfBytes 操作符来检索字符串 foo 位于每个条目中的索引:

```
db.inventory.aggregate(
    [
      {
        $project:
            {
              byteLocation: { $indexOfBytes: [ "$item", "foo" ] },
            }
       }
    ]
)
```

R:
{ "_id" : 1, "byteLocation" : "0" }
{ "_id" : 2, "byteLocation" : "4" }
{ "_id" : 3, "byteLocation" : "4" }
{ "_id" : 4, "byteLocation" : "-1" }
{ "_id" : 5, "byteLocation" : null }
{ "_id" : 6, "byteLocation" : null }

$indexOfCP

新版本 3.4。
搜索字符串以查找子字符串的发生情况，并返回第一次发生的 UTF-8 代码点索引(从零开始)。如果没有找到子字符串，则返回-1。
$indexOfCP 有以下操作符表达式语法:
{ $indexOfCP: [<string expression>, <substring expression>, <start>, <end>] }

{ $indexOfCP: ["cafeteria", "e"] } 3
{ $indexOfCP: ["cafétéria", "é"] } 3
{ $indexOfCP: ["cafétéria", "e"] } -1
{ $indexOfCP: ["cafétéria", "t"] } 4
{ $indexOfCP: ["foo.bar.fi", ".", 5] } 7
{ $indexOfCP: ["vanilla", "ll", 0, 2] } -1
{ $indexOfCP: ["vanilla", "ll", -1] } Error
{ $indexOfCP: ["vanilla", "ll", 12] } -1
{ $indexOfCP: ["vanilla", "ll", 5, 2] } -1
{ $indexOfCP: ["vanilla", "nilla", 3] } -1
{ $indexOfCP: [null, "foo"] } null

E:inventory collection

```
{ "_id" : 1, "item" : "foo" }
{ "_id" : 2, "item" : "fóofoo" }
{ "_id" : 3, "item" : "the foo bar" }
{ "_id" : 4, "item" : "hello world fóo" }
{ "_id" : 5, "item" : null }
{ "_id" : 6, "amount" : 3 }
```

下面的操作使用$indexOfCP 操作符返回每个项目字符串中 foo 字符串所在的代码点索引:

```
db.inventory.aggregate(
    [
      {
        $project:
            {
              cpLocation: { $indexOfCP: [ "$item", "foo" ] },
            }
       }
    ]
)
```

R:

```
{ "_id" : 1, "cpLocation" : "0" }
{ "_id" : 2, "cpLocation" : "3" }
{ "_id" : 3, "cpLocation" : "4" }
{ "_id" : 4, "cpLocation" : "-1" }
{ "_id" : 5, "cpLocation" : null }
{ "_id" : 6, "cpLocation" : null }
```

$isArray

新版本 3.2。

确定操作数是否为数组。返回一个布尔值。

$isArray 的语法如下:

```
{ $isArray: [ <expression> ] }
```

{ $isArray: ["hello"] } false
{ $isArray: [["hello", "world"]] } true
E: collection named warehouses
{ "_id" : 1, instock: ["chocolate"], ordered: ["butter", "apples"] }
{ "_id" : 2, instock: ["apples", "pudding", "pie"] }
{ "_id" : 3, instock: ["pears", "pecans"], ordered: ["cherries"] }
{ "_id" : 4, instock: ["ice cream"], ordered: [ ] }
下面的示例在连接两个字段之前检查 instock 和有序字段是否是数组:

```
db.warehouses.aggregate([
    { $project:
        { items:
            { $cond:
              {
                if: { $and: [ { $isArray: "$instock" }, { $isArray: "$ordered" } ] },
                then: { $concatArrays: [ "$instock", "$ordered" ] },
                else: "One or more fields is not an array."
              }
            }
        }
    }
])
```

R:
{ "_id" : 1, "items" : ["chocolate", "butter", "apples"] }
{ "_id" : 2, "items" : "One or more fields is not an array." }
{ "_id" : 3, "items" : ["pears", "pecans", "cherries"] }
{ "_id" : 4, "items" : ["ice cream"] }

$isDayOfWeek

新版本 3.4。
返回 ISO 8601 格式的工作日编号，范围从 1(周一)到 7(周日)。
$isoDayOfWeek 表达式有以下操作符表达式语法:
{ $isoDayOfWeek: <dateExpression> }
在 3.6 版中进行了更改。
该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:

新版本 3.6。
{ date: <dateExpression>, timezone: <tzExpression> }
{ $isoDayOfWeek: new Date("2016-01-01") } 5

{ $isoDayOfWeek: { date: new Date("Jan 7, 2003") } } 2

```
{ $isoDayOfWeek: {
      date: new Date("August 14, 2011"),
      timezone: "America/Chicago"                          7
} }

{ $isoDayOfWeek: ISODate("1998-11-07T00:00:00Z") }        6

{ $isoDayOfWeek: {
      date: ISODate("1998-11-07T00:00:00Z"),                5
      timezone: "-0400"
} }

{ $isoDayOfWeek: "March 28, 1976" }                       error

{ $isoDayOfWeek: Date("2016-01-01") }                     error

{ $isoDayOfWeek: "2009-04-09" }                           error
```

E:birthdays collection

```
{ "_id" : 1, "name" : "Betty", "birthday" : ISODate("1993-09-21T00:00:00Z") }
{ "_id" : 2, "name" : "Veronica", "birthday" : ISODate("1981-11-07T00:00:00Z") }
```

下面的操作返回每个生日字段的工作日号。

```
db.dates.aggregate( [
  {
    $project: {
      _id: 0,
      name: "$name",
      dayOfWeek: { $isoDayOfWeek: "$birthday" }
    }
  }
] )
```

R:

```
{ "name" : "Betty", "dayOfWeek" : 2 }
{ "name" : "Veronica", "dayOfWeek" : 6 }
```

$isoWeek

新版本 3.4。

返回 ISO 8601 格式的周数，范围从 1 到 53。周数从 1 开始，包含一年的第一个星期四的那一周(周一到周日)。

$isoWeek 表达式有以下操作符表达式语法:

{ $isoWeek: <dateExpression> }

在 3.6 版中进行了更改。

该参数必须是一个有效的表达式，解析为以下之一:

日期、时间戳或 ObjectID。
A collection called deliveries
{ "_id" : 1, "**date**" : ISODate("2006-10-24T00:00:00Z"), "city" : "Boston" }
{ "_id" : 2, "date" : ISODate("2011-08-18T00:00:00Z"), "city" : "Detroit" }
下面的操作返回每个日期字段的星期号。

```
db.deliveries.aggregate( [
  {
    $project: {
      _id: 0,
      city: "$city",
      weekNumber: { $isoWeek: "$date" }
    }
  }
] )
```

R:
{ "city" : "Boston", "weekNumber" : 43 }
{ "city" : "Detroit", "weekNumber" : 33 }

$last

返回通过字段将表达式应用于共享同一组文档的一组文档中的最后一个文档所产生的值。只有当文档按定义的顺序排列时才有意义。
$last 只在$group 阶段可用。
$last 的语法如下:

```
{ $last: <expression> }
```

在$group 阶段中使用$last 时，$group 阶段应该遵循$sort 阶段，使输入文档按照定义的顺序排列。
请注意
虽然$sort 阶段将有序的文档作为输入传递给$group 阶段，但是$group 不能保证在自己的输出中保持这种排序顺序。
E:sales collection
{ "_id" : 1, "item" : "abc", "date" : ISODate("2014-01-01T08:00:00Z"), "price" : 10, "quantity" : 2 }
{ "_id" : 2, "item" : "jkl", "date" : ISODate("2014-02-03T09:00:00Z"), "price" : 20, "quantity" : 1 }
{ "_id" : 3, "item" : "xyz", "date" : ISODate("2014-02-03T09:05:00Z"), "price" : 5, "quantity" : 5 }
{ "_id" : 4, "item" : "abc", "date" : ISODate("2014-02-15T08:00:00Z"), "price" : 10, "quantity" : 10 }
{ "_id" : 5, "item" : "xyz", "date" : ISODate("2014-02-15T09:05:00Z"), "price" : 5, "quantity" : 10 }
{ "_id" : 6, "item" : "xyz", "date" : ISODate("2014-02-15T12:05:10Z"), "price" : 5, "quantity" : 5 }
{ "_id" : 7, "item" : "xyz", "date" : ISODate("2014-02-15T14:12:12Z"), "price" : 5,

"quantity" : 10 }
下面的操作首先按项目和日期对文档进行排序，然后在接下来的$group 阶段中，按项目字段对现在已排序的文档进行分组，并使用**$last 累加器**计算每个项目的**最近销售日期:**

```
db.sales.aggregate(
    [
      { $sort: { item: 1, date: 1 } },
      {
        $group:
          {
            _id: "$item",
            lastSalesDate: { $last: "$date" }
          }
      }
    ]
)
```

R:

```
{ "_id" : "xyz", "lastSalesDate" : ISODate("2014-02-15T14:12:12Z") }
{ "_id" : "jkl", "lastSalesDate" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : "abc", "lastSalesDate" : ISODate("2014-02-15T08:00:00Z") }
```

$let

绑定变量以供在指定的表达式中使用，并返回表达式的结果。
$let 表达式的语法如下:

```
{
  $let:
      {
        vars: { <var1>: <expression>, ... },
        in: <expression>
      }
}
```

vars
在 in 表达式中可访问的变量的赋值块。要分配变量，请为变量名指定一个字符串，并为值分配一个有效表达式。

变量赋值在 in 表达式之外没有任何意义，甚至在 vars 块本身内部也没有意义。
in
在表达式中求值。
$let 可以访问定义在其表达式块之外的变量，包括系统变量。
如果修改 vars 块中外部定义变量的值，则新值仅在 in 表达式中生效。在 in 表达式之外，变量保留它们以前的值。
在 vars 赋值块中，赋值的顺序并不重要，变量赋值只在 In 表达式中有意义。因此，在 vars 赋值块中访问变量的值是指在 vars 块外部定义的变量的值，而不是在同一个 vars 块内部定义的变量的值。

例如，考虑下面的$let 表达式:

```
{
   $let:
      {
         vars: { low: 1, high: "$$low" },
         in: { $gt: [ "$$low", "$$high" ] }
      }
}
```

在 vars 赋值块中，“$$low”是指外部定义的变量 low 的值，而不是同一个 vars 块中定义的变量。如果在这个$let 表达式块之外没有定义 low，则该表达式无效。

E:

Sales collection

```
{ _id: 1, price: 10, tax: 0.50, applyDiscount: true }
{ _id: 2, price: 10, tax: 0.25, applyDiscount: false }
```

下面的聚合使用$let 在$project pipeline 阶段计算并返回每个文档的 finalTotal:

```
db.sales.aggregate( [
    {
        $project: {
            finalTotal: {
                $let: {
                    vars: {
                        total: { $add: [ '$price', '$tax' ] },
                        discounted: { $cond: { if: '$applyDiscount', then: 0.9, else: 1 } }
                    },
                    in: { $multiply: [ "$$total", "$$discounted" ] }
                }
            }
        }
    }
] )
```

R:

```
{ "_id" : 1, "finalTotal" : 9.450000000000001 }
{ "_id" : 2, "finalTotal" : 10.25 }
```

$literal

返回一个不需要解析的值。用于聚合管道可能解释为表达式的值。

$literal 表达式的语法如下:

```
{ $literal: <value> }
```

如果<value>是一个表达式，$literal 不计算表达式的值，而是返回未解析的表达式。

```
{ $literal: { $add: [ 2, 3 ] } }       { "$add" : [ 2, 3 ] }
```

{ $literal: { $literal: 1 } }　　{ "$literal" : 1 }

Example:
把$当作文字
在表达式中，$符号$计算为字段路径;即提供对该字段的访问。例如，$eq 表达式$eq: ["$price", "$1"]在指定 price 字段中的值和文档中指定 1 字段中的值之间执行相等检查。下面的示例使用$literal 表达式将包含$符号“$1”的字符串视为常量。
A collection records 有以下文件:
{ "_id" : 1, "item" : "abc123", price: "$2.50" }
{ "_id" : 2, "item" : "xyz123", price: "1" }
{ "_id" : 3, "item" : "ijk123", price: "$1" }
db.records.aggregate([
　{ $project: { costsOneDollar: { $eq: ["$price", { $literal: "$1" }] } } }
])
该操作将投射一个名为 costsOneDollar 的字段，该字段持有一个布尔值，指示 price 字段的值是否等于字符串“$1”：
{ "_id" : 1, "costsOneDollar" : false }
{ "_id" : 2, "costsOneDollar" : false }
{ "_id" : 3, "costsOneDollar" : true }
项目一个值为 1 的新字段
$project 阶段使用表达式<field>: 1 在输出中包含<field>。下面的示例使用$literal 返回一个值为 1 的新字段。
A collection bid 有以下文件:
{ "_id" : 1, "item" : "abc123", condition: "new" }
{ "_id" : 2, "item" : "xyz123", condition: "new" }
下面的聚合计算表达式项:1 表示返回输出中已有的字段项，但是使用{$literal: 1}表达式返回一个新的字段 startAt，该字段的值设置为 1:
db.bids.aggregate([
　{ $project: { item: 1, startAt: { $literal: 1 } } }
])
R:
{ "_id" : 1, "item" : "abc123", "startAt" : 1 }
{ "_id" : 2, "item" : "xyz123", "startAt" : 1 }

$ln

新版本 3.2。
计算 ln (i)的自然对数。)，并以双精度返回结果。
$ln 的语法如下:
{$ln: <number>}
<number>表达式可以是任何有效的表达式，只要它解析为一个非负数。有关表达式的更多信息，请参见表达式。

$ln 等于$log: [<number>, Math。E]表达式，其中有数学。E 是欧拉数 E 的 JavaScript 表示。
Behavior:
如果参数解析为 null 值或引用缺失的字段，$ln 返回 null。如果参数解析为 NaN， $ln 返回 NaN。
{ $ln: 1 } 0
{ $ln: Math.E } where Math.E is a JavaScript representation for e. 1
{ $ln: 10 } 2.302585092994046
E:sales collection
{ _id: 1, year: "2000", sales: 8700000 }
{ _id: 2, year: "2005", sales: 5000000 }
{ _id: 3, year: "2010", sales: 6250000 }
下面的示例转换 sales 数据:
db.sales.aggregate([{ $project: { x: "$year", y: { $ln: "$sales" } } }])
R:
{ "_id" : 1, "x" : "2000", "y" : 15.978833583624812 }
{ "_id" : 2, "x" : "2005", "y" : 15.424948470398375 }
{ "_id" : 3, "x" : "2010", "y" : 15.648092021712584 }

$log

新版本 3.2。
计算指定基数中数字的日志，并以双精度返回结果。
$log 有以下语法:
{ $log: [<number>, <base>] }
<number>表达式可以是任何有效的表达式，只要它解析为一个非负数。
<base>表达式可以是任何有效的表达式，只要它解析为一个大于 1 的正数。
有关表达式的更多信息，请参见表达式。
行为
如果参数解析为 null 值或引用缺少的字段，$log 返回 null。如果其中一个参数解析为 NaN，$log 将返回 NaN。

{ $log: [100, 10] } 2
{ $log: [100, Math.E] } where Math.E is a JavaScript representation for e. 4.605170185988092
E:A collection examples
{ _id: 1, positiveInt: 5 }
{ _id: 2, positiveInt: 2 }
{ _id: 3, positiveInt: 23 }
{ _id: 4, positiveInt: 10 }
下面的示例在计算中使用 log2 来确定表示 positiveInt 值所需的比特数。
db.examples.aggregate([
    { $project: { bitsNeeded:

```
        {
            $floor: { $add: [ 1, { $log: [ "$positiveInt", 2 ] } ] } }
        }
])
```
R:
```
{ "_id" : 1, "bitsNeeded" : 3 }
{ "_id" : 2, "bitsNeeded" : 2 }
{ "_id" : 3, "bitsNeeded" : 5 }
{ "_id" : 4, "bitsNeeded" : 4 }
```

$log10

新版本 3.2。
计算以 10 为底的对数，并以双精度返回结果。
$log10 的语法如下:
```
{ $log10: <number> }
```
<number>表达式可以是任何有效的表达式，只要它解析为一个非负数。有关表达式的更多信息，请参见表达式。
$log10 等价于$log: [<number>， 10]表达式。
Behavior:
如果参数解析为 null 值或引用缺失的字段, $log10 返回 null。如果参数解析为 NaN，$log10 将返回 NaN。
```
{ $log10: 1 }       0
{ $log10: 10 }      1
{ $log10: 100 }     2
{ $log10: 1000 }    3
```
E:collection samples
```
{ _id: 1, H3O: 0.0025 }
{ _id: 2, H3O: 0.001 }
{ _id: 3, H3O: 0.02 }
```
下面的例子计算了样品的 pH 值:
```
db.samples.aggregate( [
    { $project: { pH: { $multiply: [ -1, { $log10: "$H3O" } ] } } }
] )
```
R:
```
{ "_id" : 1, "pH" : 2.6020599913279625 }
{ "_id" : 2, "pH" : 3 }
{ "_id" : 3, "pH" : 1.6989700043360187 }
```

$lt

比较两个值和返回值:
当第一个值小于第二个值时为真。

当第一个值大于或等于第二个值时为 false。
$lt 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。
$lt 的语法如下:
{ $lt: [<expression1>, <expression2>] }
E:inventory collection
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
以下操作使用$lt 操作符确定 qty 是否小于 250:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
            qty: 1,
            qtyLt250: { $lt: [ "$qty", 250 ] },
            _id: 0
          }
      }
    ]
)
```

R:
{ "item" : "abc1", "qty" : 300, "qtyLt250" : false }
{ "item" : "abc2", "qty" : 200, "qtyLt250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyLt250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyLt250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyLt250" : true }

$lte

比较两个值和返回值:
当第一个值小于或等于第二个值时为真。
当第一个值大于第二个值时为 false。
$lte 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。
$lte 有以下语法:
{ $lte: [<expression1>, <expression2>] }
E:inventory collection
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }

{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
下面的操作使用$lte 操作符来确定 qty 是否小于或等于 250:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
            qty: 1,
            qtyLte250: { $lte: [ "$qty", 250 ] },
            _id: 0
          }
      }
    ]
)
```

E:

```
{ "item" : "abc1", "qty" : 300, "qtyLte250" : false }
{ "item" : "abc2", "qty" : 200, "qtyLte250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyLte250" : true }
{ "item" : "VWZ1", "qty" : 300, "qtyLte250" : false }
{ "item" : "VWZ2", "qty" : 180, "qtyLte250" : true }
```

$ltrim

删除字符串开头的空白字符(包括 null)或指定的字符。
$ltrim 有以下语法:

```
{ $ltrim: { input: <string>,    chars: <string> } }
```

Example	Results
{ $ltrim: { input: " ggggoodbyeeeee ", chars: " gd" } }	"oodbyeeeee "

E inventory collection:

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : " product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2 \n The product is in stock.   \n\n   " }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
```

下面的操作使用$ltrim 操作符从 description 字段中删除前导空格:

```
db.inventory.aggregate([
    { $project: { item: 1, description: { $ltrim: { input: "$description" } } } }
])
```

R:

```
{ "_id" : 1, "item" : "ABC1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", "description" : "product 2 \n The product is in stock.   \n\n   " }
{ "_id" : 3, "item" : "XYZ1", "description" : null }
```

$map：

将表达式**应用于数组中的每个项，并返回具有应用结果的数组**。
$map 表达式的语法如下:
{ $map: { input: <expression>, as: <string>, in: <expression> } }
使用$map 实施向数组的每个元素添加
一套名为“grades”的资料包括以下文件:
{ _id: 1, quizzes: [5, 6, 7] }
{ _id: 2, quizzes: [ ] }
{ _id: 3, quizzes: [3, 8, 9] }
下面的聚合操作输出文档，其中 quizzes 数组的每个成员都增加了 2。

```
db.grades.aggregate(
    [
        { $project:
            { adjustedGrades:
                {
                  $map:
                     {
                       input: "$quizzes",
                       as: "grade",
                       in: { $add: [ "$$grade", 2 ] }
                     }
                }
            }
        }
    ]
)
```

R:
{ "_id" : 1, "adjustedGrades" : [7, 8, 9] }
{ "_id" : 2, "adjustedGrades" : [ ] }
{ "_id" : 3, "adjustedGrades" : [5, 10, 11] }
使用$map 截断每个数组元素
一个名为 delivery 的集合包含以下文档:
{ "_id" : 1, "city" : "Bakersfield", "distances" : [34.57, 81.96, 44.24] }
{ "_id" : 2, "city" : "Barstow", "distances" : [73.28, 9.67, 124.36] }
{ "_id" : 3, "city" : "San Bernadino", "distances" : [16.04, 3.25, 6.82] }
下面的聚合操作使用$trunc 操作符将距离数组的每个成员截断为其整数值。

```
db.deliveries.aggregate(
    [
        { $project:
            {   city: "$city",
                integerValues:
                   { $map:
                      {
```

```
                        input: "$distances",
                        as: "integerValue",
                        in: { $trunc: "$$integerValue" }
                    }
                }
            }
        }
    ]
)
```

R:

```
{ "_id" : 1, "city" : "Bakersfield", "integerValues" : [ 34, 81, 44 ] }
{ "_id" : 2, "city" : "Barstow", "integerValues" : [ 73, 9, 124 ] }
{ "_id" : 3, "city" : "San Bernadino", "integerValues" : [ 16, 3, 6 ] }
```

$max

返回最大值。$max 比较值和类型，**使用指定的 BSON 比较顺序比较不同类型的值**。
在 3.2 版本中进行了更改:$max 可用于$group 和$project 阶段。在以前版本的 MongoDB 中，$max 只在$group 阶段可用。
在$group 阶段使用时，$max 具有以下语法，并返回将表达式应用于一组按键共享相同组的文档中的每个文档所产生的最大值:
{ $max: <expression> }
在$project 阶段使用时，$max 返回每个文档指定表达式或表达式列表的最大值，并具有以下两种语法之一:
$max 有一个指定的表达式作为它的操作数:
{ $max: <expression> }
$max 的操作数是指定的表达式列表:
{ $max: [<expression1>, <expression2> ...] }

行为
空值或缺失值
如果$max 操作的一些(但不是所有)文档的字段值为 null，或者缺少字段，那么$max 操作符只考虑字段的非 null 值和非缺少值。
如果$max 操作的所有文档的字段都有 null 值，或者缺少该字段，那么$max 操作符将返回 null 作为最大值。
数组操作数
在$group 阶段，如果表达式解析为数组，则$max 不遍历数组并将数组作为一个整体进行比较。
在$project 阶段:
使用单个表达式作为操作数，如果表达式解析为数组，$max 将遍历数组，对数组的数字元素进行操作，以返回单个值。
以表达式列表作为操作数，如果任何表达式解析为数组，$max 将不遍历数组，而是将数组视为非数值。

Use in $group Stage

{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }

按照 item 字段对文档进行分组，下面的操作使用$max 累加器计算每组文档的最大总金额和最大数量。

```
db.sales.aggregate(
    [
      {
        $group:
          {
            _id: "$item",
            maxTotalAmount: { $max: { $multiply: [ "$price", "$quantity" ] } },
            maxQuantity: { $max: "$quantity" }
          }
      }
    ]
)
```

R:

{ "_id" : "xyz", "maxTotalAmount" : 50, "maxQuantity" : 10 }
{ "_id" : "jkl", "maxTotalAmount" : 20, "maxQuantity" : 1 }
{ "_id" : "abc", "maxTotalAmount" : 100, "maxQuantity" : 10 }

Use in $project Stage

A collection students

{ "_id": 1, "quizzes": [10, 6, 7], "labs": [5, 8], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [9, 10], "labs": [8, 8], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [4, 5, 5], "labs": [6, 5], "final": 78, "midterm": 70 }

以下示例使用$ project 阶段中的$ max 来计算最大测验分数，最大实验室分数以及最终和期中考试的最大值：

```
db.students.aggregate([
    {
      $project: {
        quizMax: { $max: "$quizzes"},
        labMax: { $max: "$labs" },
        examMax: { $max: [ "$final", "$midterm" ] }
      }
    }
```

])
{ "_id" : 1, "quizMax" : 10, "labMax" : 8, "examMax" : 80 }
{ "_id" : 2, "quizMax" : 10, "labMax" : 8, "examMax" : 95 }
{ "_id" : 3, "quizMax" : 5, "labMax" : 6, "examMax" : 78 }
在$project 阶段:
使用单个表达式作为操作数，如果表达式解析为数组，$max 将遍历数组，对数组的数字元素进行操作，以返回单个值。
以表达式列表作为操作数，如果任何表达式解析为数组，$max 将不遍历数组，而是将数组视为非数值。

$mergeObjects

新版本 3.6。
将多个文档合并到一个文档中。
当$mergeObjects 用作$group stage 累加器时，它的形式如下:
{ $mergeObjects: <document> }
在其他表达式中使用时，包括在$group 阶段，但不作为累加器:
{ $mergeObjects: [<document1>, <document2>, ...] }
<document>可以是解析为文档的任何有效表达式。
Behavior：
合并对象忽略空操作数。如果合并对象的所有操作数都解析为 null，则合并对象返回一个空文档{}。
合并对象在合并文档时覆盖字段值。如果要合并的文档包含相同的字段名，则生成的文档中的字段具有上次为该字段合并的文档的值。
E:

Example	Results
`{ $mergeObjects: [ { a: 1 }, null ] }`	`{ a: 1 }`
`{ $mergeObjects: [ null, null ] }`	`{ }`
`{ $mergeObjects: [ { a: 1 }, { a: 2, b: 2 }, { a: 3, c: 3 } ] }`	`{ a: 3, b: 2, c: 3 }`
`{ $mergeObjects: [ { a: 1 }, { a: 2, b: 2 }, { a: 3, b: null, c: 3 } ] }`	`{ a: 3, b: null, c: 3 }`

E:$mergeObjects
Orders collection
db.orders.insert([

```
   { "_id" : 1, "item" : "abc", "price" : 12, "ordered" : 2 },
   { "_id" : 2, "item" : "jkl", "price" : 20, "ordered" : 1 }
])
```

使用以下文件创建另一个集合项:

```
db.items.insert([
   { "_id" : 1, "item" : "abc", description: "product 1", "instock" : 120 },
   { "_id" : 2, "item" : "def", description: "product 2", "instock" : 80 },
   { "_id" : 3, "item" : "jkl", description: "product 3", "instock" : 60 }
])
```

下面的操作首先使用$lookup stage 通过 item 字段将两个集合连接起来，然后在 $replaceRoot 中使用$mergeObjects 将已连接的文档从 items 和 order 中合并起来:

```
db.orders.aggregate([
    {
        $lookup: {
            from: "items",
            localField: "item",      // field in the orders collection
            foreignField: "item",    // field in the items collection
            as: "fromItems"
        }
    },
    {
        $replaceRoot: { newRoot: { $mergeObjects: [ { $arrayElemAt: [ "$fromItems", 0 ] }, "$$ROOT" ] } }
    },
    { $project: { fromItems: 0 } }
])
```

R:

```
{ "_id" : 1, "item" : "abc", "description" : "product 1", "instock" : 120, "price" : 12, "ordered" : 2 }
{ "_id" : 2, "item" : "jkl", "description" : "product 3", "instock" : 60, "price" : 20, "ordered" : 1 }
```

$mergeObjects as an Accumulator¶

Create a collection sales with the following documents:

```
db.sales.insert( [
    { _id: 1, year: 2017, item: "A", quantity: { "2017Q1": 500, "2017Q2": 500 } },
    { _id: 2, year: 2016, item: "A", quantity: { "2016Q1": 400, "2016Q2": 300, "2016Q3": 0, "2016Q4": 0 } } ,
    { _id: 3, year: 2017, item: "B", quantity: { "2017Q1": 300 } },
    { _id: 4, year: 2016, item: "B", quantity: { "2016Q3": 100, "2016Q4": 250 } }
] )
```

下面的操作使用$mergeObjects 作为累加器，在$group 阶段按项目字段对文档进行分组:

请注意

当用作累加器时，$mergeObjects 操作符接受一个操作数时，$mergeObjects 操作符接受一个操作数。

```
db.sales.aggregate( [
    { $group: { _id: "$item", mergedSales: { $mergeObjects: "$quantity" } } }
])
```

R:

```
{ "_id" : "B", "mergedSales" : { "2017Q1" : 300, "2016Q3" : 100, "2016Q4" : 250 } }
{ "_id" : "A", "mergedSales" : { "2017Q1" : 500, "2017Q2" : 500, "2016Q1" : 400, "2016Q2" : 300, "2016Q3" : 0, "2016Q4" : 0 } }
```

请注意

如果要合并的文档包含相同的字段名，则结果文档中的字段具有该字段的上一个合并文档的值。

$meta:

新版本 2.6。

返回管道操作中与文档关联的元数据，例如。“textScore”执行文本搜索时。

$meta 表达式有以下语法

```
{ $meta: <metaDataKeyword> }
```

$meta: "textScore"}表达式是$sort 阶段接受的唯一表达式。

虽然可以在管道中接受的所有地方使用表达式，但是{$meta: "textScore"}表达式只在管道中有意义，管道中包含一个$match 阶段和一个$text 查询。

视图不支持文本搜索。

关键词描述排序顺序

“textScore”返回与每个匹配文档对应的$text 查询相关联的分数。文本得分表示文档与搜索项或搜索项的匹配程度。如果不与$text 查询一起使用，则返回一个 null 分数。

articles collection

```
{ "_id" : 1, "title" : "cakes and ale" }
{ "_id" : 2, "title" : "more cakes" }
{ "_id" : 3, "title" : "bread" }
{ "_id" : 4, "title" : "some cakes" }
```

下面的聚合操作执行文本搜索，并使用$meta 操作符根据文本搜索得分进行分组:

```
db.articles.aggregate(
    [
      { $match: { $text: { $search: "cake" } } },
      { $group: { _id: { $meta: "textScore" }, count: { $sum: 1 } } }
    ]
)
```

R:

```
{ "_id" : 0.75, "count" : 1 }
{ "_id" : 1, "count" : 2 }
```

$min

返回最小值。$min 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。

在 3.2 版本中进行了更改:$min 可用于$group 和$project 阶段。在以前版本的 MongoDB 中，$min 只在$group 阶段可用。
在$group 阶段使用$min 时，$min 具有以下语法，并返回将表达式应用于一组按键共享相同组的文档中的每个文档所产生的最小值:
{ $min: <expression> }
在$project 阶段使用时，$min 返回每个文档的指定表达式或表达式列表的最小值，并具有以下两种语法之一:
$min 有一个指定的表达式作为操作数:
{ $min: <expression> }
$min 的操作数是指定的表达式列表:
{ $min: [<expression1>, <expression2> ...] }
Behavior:
空值或缺失值
如果$min 操作的一些(但不是所有)文档的字段值为空，或者缺少字段，那么$min 操作符只考虑字段的非空值和非缺少值。
如果$min 操作的所有文档的字段都有 null 值，或者缺少该字段，那么$min 操作符将为最小值返回 null。
数组操作数
在$group 阶段，如果表达式解析为数组，$min 不会遍历数组并将数组作为一个整体进行比较。
在$project 阶段:
使用一个表达式作为操作数，如果表达式解析为一个数组，$min 将遍历数组，对数组的数字元素进行操作，以返回一个值。
以表达式列表作为操作数，如果任何表达式解析为数组，$min 不遍历数组，而是将数组视为非数值。
E:

Use in $group Stage
Collection sales:
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
按照 item 字段对文档进行分组，下面的操作使用$min 累加器计算每个分组的最小数量和最小数量。

```
db.sales.aggregate(
    [
      {
```

```
        $group:
          {
            _id: "$item",
            minQuantity: { $min: "$quantity" }
          }
      }
   ]
)
```

R:

```
{ "_id" : "xyz", "minQuantity" : 5 }
{ "_id" : "jkl", "minQuantity" : 1 }
{ "_id" : "abc", "minQuantity" : 2 }
```

Use in $project Stage:

Collection students:

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

下面的例子使用$project 阶段中的$min 来计算最低的测验分数、最低的实验室分数以及期末和期中考试的最低分数:

```
db.students.aggregate([
   {
     $project: {
        quizMin: { $min: "$quizzes"},
        labMin: { $min: "$labs" },
        examMin: { $min: [ "$final", "$midterm" ] }
     }
   }
])
```

R:

```
{ "_id" : 1, "quizMin" : 6, "labMin" : 5, "examMin" : 75 }
{ "_id" : 2, "quizMin" : 9, "labMin" : 8, "examMin" : 80 }
{ "_id" : 3, "quizMin" : 4, "labMin" : 5, "examMin" : 70 }
```

在$project 阶段:

使用一个表达式作为操作数，如果表达式解析为一个数组，$min 将遍历数组，对数组的数字元素进行操作，以返回一个值。

以表达式列表作为操作数，如果任何表达式解析为数组，$min 不遍历数组，而是将数组视为非数值。

$millinsecond

以 0 到 999 之间的整数返回日期的毫秒部分。

$ ond 表达式有以下操作符表达式语法:

```
{ $millisecond: <dateExpression> }
```

该参数必须是一个有效的表达式，解析为以下之一:

日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。
{ date: <dateExpression>, timezone: <tzExpression> }
E:sales collection

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$ ond 和其他日期操作符来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

R:

```
{
   "_id" : 1,
   "year" : 2014,
   "month" : 1,
   "day" : 1,
   "hour" : 8,
   "minutes" : 15,
   "seconds" : 39,
   "milliseconds" : 736,
   "dayOfYear" : 1,
   "dayOfWeek" : 4,
```

```
  "week" : 0
}
```

$minute

以 0 到 59 之间的数字返回日期的分钟部分。
$minute 表达式有以下操作符表达式语法:
{ $minute: <dateExpression> }
该参数必须是一个有效的表达式，解析为以下之一
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。
{ date: <dateExpression>, timezone: <tzExpression> }
E:sales collection

```
{
  "_id" : 1,
  "item" : "abc",
  "price" : 10,
  "quantity" : 2,
  "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$minute 和其他日期表达式来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

E:

```
{
  "_id" : 1,
```

```
    "year" : 2014,
    "month" : 1,
    "day" : 1,
    "hour" : 8,
    "minutes" : 15,
    "seconds" : 39,
    "milliseconds" : 736,
    "dayOfYear" : 1,
    "dayOfWeek" : 4,
    "week" : 0
}
```

$mod

将一个数字除以另一个数字并返回余数。
$ mod 表达式具有以下语法：
{$ mod：[<expression1>，<expression2>]}
第一个参数是红利，第二个参数是除数;即第一个参数除以第二个参数。
参数可以是任何有效的表达式，只要它们解析为数字即可。有关表达式的更多信息，请参阅表达式。
Planning collection:

```
{ "_id" : 1, "project" : "A", "hours" : 80, "tasks" : 7 }
{ "_id" : 2, "project" : "B", "hours" : 40, "tasks" : 4 }
```

下面的聚合使用$mod 表达式返回剩余的 hours 字段除以 tasks 字段:

```
db.planning.aggregate(
    [
      { $project: { remainder: { $mod: [ "$hours", "$tasks" ] } } }
    ]
)
```

R:

```
{ "_id" : 1, "remainder" : 3 }
{ "_id" : 2, "remainder" : 0 }
```

$month

Returns the month of a date as a number between 1 and 12.
The $month expression has the following operator expression syntax:
{ $month: <dateExpression> }
在 3.6 版中进行了更改。
该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。

{ date: <dateExpression>, timezone: <tzExpression> }

Sales collection:

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$month 和其他日期操作符来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

R:

```
{
   "_id" : 1,
   "year" : 2014,
   "month" : 1,
   "day" : 1,
   "hour" : 8,
   "minutes" : 15,
   "seconds" : 39,
   "milliseconds" : 736,
   "dayOfYear" : 1,
   "dayOfWeek" : 4,
   "week" : 0
}
```

$multiply

将数字相乘并返回结果。将参数传递给数组中的$multiply。
$multiply 表达的语法如下:
{$multiply: [<expression1>, <expression2>, ...]}
参数可以是任何有效的表达式，只要它们解析为数字。有关表达式的更多信息，请参见表达式。
E:sales collection
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity": 2, date: ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity": 1, date: ISODate("2014-03-01T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity": 10, date: ISODate("2014-03-15T09:00:00Z") }
下面的聚合使用$project 管道中的$multiply 表达式将 price 和 quantity 字段相乘:

```
db.sales.aggregate(
    [
      { $project: { date: 1, item: 1, total: { $multiply: [ "$price", "$quantity" ] } } }
    ]
)
```

R:

```
{ "_id" : 1, "item" : "abc", "date" : ISODate("2014-03-01T08:00:00Z"), "total" : 20 }
{ "_id" : 2, "item" : "jkl", "date" : ISODate("2014-03-01T09:00:00Z"), "total" : 20 }
{ "_id" : 3, "item" : "xyz", "date" : ISODate("2014-03-15T09:00:00Z"), "total" : 50 }
```

$ne

比较两个值和返回值:
当值不相等时为真。
当值相等时为 false。
$lte 比较值和类型，使用指定的 BSON 比较顺序比较不同类型的值。
$ne 的语法如下:
{ $ne: [<expression1>, <expression2>] }
E:inventory collection

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

下面的操作使用$ne 操作符来确定 qty 是否不等于 250:

```
db.inventory.aggregate(
    [
      {
```

```
        $project:
            {
              item: 1,
              qty: 1,
              qtyNe250: { $ne: [ "$qty", 250 ] },
              _id: 0
            }
      }
   ]
)
```

R:

```
{ "item" : "abc1", "qty" : 300, "qtyNe250" : true }
{ "item" : "abc2", "qty" : 200, "qtyNe250" : true }
{ "item" : "xyz1", "qty" : 250, "qtyNe250" : false }
{ "item" : "VWZ1", "qty" : 300, "qtyNe250" : true }
{ "item" : "VWZ2", "qty" : 180, "qtyNe250" : true }
```

$not

计算布尔值并返回相反的布尔值;也就是说，当传递一个计算结果为 true 的表达式时，$not 返回 false;当传递一个计算结果为 false 的表达式时，$not 返回 true。

db.t11.find({country:{$not:{$all:["日本","韩国"]}}})

$not 的语法如下:

{ $not: [<expression>] }

Example	Result
`{ $not: [ true ] }`	`false`
`{ $not: [ [ false ] ] }`	`false`
`{ $not: [ false ] }`	`true`
`{ $not: [ null ] }`	`true`
`{ $not: [ 0 ] }`	`true`

E:inventory collection

```
{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }
{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }
{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
```

下面的操作使用$not 操作符来确定 qty 是否不大于 250:

```
db.inventory.aggregate(
   [
     {
```

```
        $project:
            {
              item: 1,
              result: { $not: [ { $gt: [ "$qty", 250 ] } ] }
            }
       }
    ]
)
```

R:

```
{ "_id" : 1, "item" : "abc1", "result" : false }
{ "_id" : 2, "item" : "abc2", "result" : true }
{ "_id" : 3, "item" : "xyz1", "result" : true }
{ "_id" : 4, "item" : "VWZ1", "result" : false }
{ "_id" : 5, "item" : "VWZ2", "result" : true }
```

$objectToArray

新版本 3.4.4。

将文档转换为数组。返回数组包含原始文档中每个字段/值对的元素。返回数组中的每个元素都是一个包含两个字段 k 和 v 的文档:

k 字段包含原始文档中的字段名。

v 字段包含原始文档中字段的值。

$objectToArray 有以下语法:

```
{ $objectToArray: <object> }
```

只要解析为文档对象，<object>表达式可以是任何有效的表达式。$objectToArray 应用于其参数的顶级字段。如果参数本身是包含嵌入式文档字段的文档，则$objectToArray 不会递归地应用于嵌入式文档字段。

Example	Results
`{ $objectToArray: { item: "foo", qty: 25 } }`	`[ { "k" : "item", "v" : "foo" }, { "k" : "qty", "v" : 25 } ]`
`{ $objectToArray: { item: "foo", qty: 25, size: { len: 25, w: 10, uom: "cm" } } }`	`[ { "k" : "item", "v" : "foo" }, { "k" : "qty", "v" : 25 }, { "k" : "size", "v" : { "len" : 25, "w" : 10, "uom" : "cm" } } ]`

$objectToArray E：inventory collection

{ "_id" : 1, "item" : "ABC1", dimensions: { l: 25, w: 10, uom: "cm" } }

{ "_id" : 2, "item" : "ABC2", dimensions: { l: 50, w: 25, uom: "cm" } }

{ "_id" : 3, "item" : "XYZ1", dimensions: { l: 70, w: 75, uom: "cm" } }

下面的聚合管道操作使用$objectToArray 将 dimensions 字段作为数组返回:

```
db.inventory.aggregate(
    [
        {
            $project: {
                item: 1,
                dimensions: { $objectToArray: "$dimensions" }
            }
        }
    ]
)
```

R:

{ "_id" : 1, "item" : "ABC1", "dimensions" : [{ "k" : "l", "v" : 25 }, { "k" : "w", "v" : 10 }, { "k" : "uom", "v" : "cm" }] }

{ "_id" : 2, "item" : "ABC2", "dimensions" : [{ "k" : "l", "v" : 50 }, { "k" : "w", "v" : 25 }, { "k" : "uom", "v" : "cm" }] }

{ "_id" : 3, "item" : "XYZ1", "dimensions" : [{ "k" : "l", "v" : 70 }, { "k" : "w", "v" : 75 }, { "k" : "uom", "v" : "cm" }] }

$objectToArray to Sum Nested Fields

Inventory collection

{ "_id" : 1, "item" : "ABC1", instock: { warehouse1: 2500, warehouse2: 500 } }

{ "_id" : 2, "item" : "ABC2", instock: { warehouse2: 500, warehouse3: 200} }

下面的聚合管道操作使用$objectToArray 以及$unwind 和$group 来计算每个仓库的总库存项。

```
db.inventory.aggregate([
    { $project: { warehouses: { $objectToArray: "$instock" } } },          1
    { $unwind: "$warehouses" },                                            2
    { $group: { _id: "$warehouses.k", total: { $sum: "$warehouses.v" } } }        3
])
```

R:

{ "_id" : "warehouse3", "total" : 200 }

{ "_id" : "warehouse2", "total" : 1000 }

{ "_id" : "warehouse1", "total" : 2500 }

$objectToArray + $arrayToObject Example

Inventory collection

{ "_id" : 1, "item" : "ABC1", instock: { warehouse1: 2500, warehouse2: 500 } }

{ "_id" : 2, "item" : "ABC2", instock: { warehouse2: 500, warehouse3: 200} }

下面的聚合管道操作计算每个项目的总库存，并添加到 instock 文档:

```
db.inventory.aggregate( [
    { $addFields: { instock: { $objectToArray: "$instock" } } },
    { $addFields: { instock: { $concatArrays: [ "$instock", [ { "k": "total", "v": { $sum: "$instock.v" } } ] ] } } } ,
    { $addFields: { instock: { $arrayToObject: "$instock" } } }
] )
```

R:

{ "_id" : 1, "item" : "ABC1", "instock" : { "warehouse1" : 2500, "warehouse2" : 500, "total" : 3000 } }

{ "_id" : 2, "item" : "ABC2", "instock" : { "warehouse2" : 500, "warehouse3" : 200, "total" : 700 } }

$or

计算一个或多个表达式，如果其中任何表达式为真，则返回 true。否则，$or 返回 false。

$or 的语法如下:

{ $or: [<expression1>, <expression2>, ...] }

Example	Result
`{ $or: [ true, false ] }`	`true`
`{ $or: [ [ false ], false ] }`	`true`
`{ $or: [ null, 0, undefined ] }`	`false`
`{ $or: [ ] }`	`false`

E:inventory collection

{ "_id" : 1, "item" : "abc1", description: "product 1", qty: 300 }

{ "_id" : 2, "item" : "abc2", description: "product 2", qty: 200 }

{ "_id" : 3, "item" : "xyz1", description: "product 3", qty: 250 }
{ "_id" : 4, "item" : "VWZ1", description: "product 4", qty: 300 }
{ "_id" : 5, "item" : "VWZ2", description: "product 5", qty: 180 }
下面的操作使用$or 操作符来确定 qty 是大于 250 还是小于 200:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: 1,
            result: { $or: [ { $gt: [ "$qty", 250 ] }, { $lt: [ "$qty", 200 ] } ] }
          }
      }
    ]
)
```

R:

```
{ "_id" : 1, "item" : "abc1", "result" : true }
{ "_id" : 2, "item" : "abc2", "result" : false }
{ "_id" : 3, "item" : "xyz1", "result" : false }
{ "_id" : 4, "item" : "VWZ1", "result" : true }
{ "_id" : 5, "item" : "VWZ2", "result" : true }
```

$pow

新版本 3.2。
将数字提升到指定的指数并返回结果。$pow 的语法如下:
{$pow: [<number>, <exponent>]]
<number>表达式可以是任何有效的表达式，只要它解析为一个数字。
<exponent>表达式可以是任何有效的表达式，只要它解析为一个数字。
0 不能取负数。
Behavior:
结果将具有与输入相同的类型，除非它不能在该类型中精确表示。在这些情况下:
如果结果可以表示为 64 位整数，则 32 位整数将转换为 64 位整数。
如果结果不能表示为 64 位整数，则 32 位整数将转换为双精度整数。
如果结果不能表示为 64 位整数，则将 64 位整数转换为 double。
如果参数解析为 null 值或引用缺失的字段, $pow 返回 null。如果其中一个参数解析为 NaN, $pow 将返回 NaN。

```
{ $pow: [ 5, 0 ] }        1
{ $pow: [ 5, 2 ] }        25
{ $pow: [ 5, -2 ] }       0.04
{ $pow: [ -5, 0.5 ] }     NaN
```

E：quizzes collection

```
{
    "_id" : 1,
```

```
    "scores" : [
        {
            "name" : "dave123",
            "score" : 85
        },
        {
            "name" : "dave2",
            "score" : 90
        },
        {
            "name" : "ahn",
            "score" : 71
        }
    ]
}
{
    "_id" : 2,
    "scores" : [
        {
            "name" : "li",
            "quiz" : 2,
            "score" : 96
        },
        {
            "name" : "annT",
            "score" : 77
        },
        {
            "name" : "ty",
            "score" : 82
        }
    ]
}
```

下面的例子计算每个测验的方差:

```
db.quizzes.aggregate([
    { $project: { variance: { $pow: [ { $stdDevPop: "$scores.score" }, 2 ] } } }
])
```

R:

```
{ "_id" : 1, "variance" : 64.66666666666667 }
{ "_id" : 2, "variance" : 64.66666666666667 }
```

$push

返回一个包含所有值的数组, 这些值是将一个表达式应用于一组按键共享同一组的文档中的每个文档而得到的。

$push 只在$group 阶段可用。

$push 有以下语法:

```
{ $push: <expression> }
```

E:sales

```
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
{ "_id" : 6, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-15T12:05:10Z") }
{ "_id" : 7, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T14:12:12Z") }
```

按日期字段的日期和年份对文档进行分组, 下面的操作使用$push 累加器计算每组销售的项目和数量列表:

```
db.sales.aggregate(
    [
      {
        $group:
          {
            _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
            itemsSold: { $push:  { item: "$item", quantity: "$quantity" } }
          }
      }
    ]
)
```

R:

```
{
    "_id" : { "day" : 46, "year" : 2014 },
    "itemsSold" : [
        { "item" : "abc", "quantity" : 10 },
        { "item" : "xyz", "quantity" : 10 },
        { "item" : "xyz", "quantity" : 5 },
        { "item" : "xyz", "quantity" : 10 }
    ]
```

```
}
{
    "_id" : { "day" : 34, "year" : 2014 },
    "itemsSold" : [
        { "item" : "jkl", "quantity" : 1 },
        { "item" : "xyz", "quantity" : 5 }
    ]
}
{
    "_id" : { "day" : 1, "year" : 2014 },
    "itemsSold" : [ { "item" : "abc", "quantity" : 2 } ]
}
```

$range

返回一个数组，该数组的元素是生成的数字序列。$range 通过将起始数按指定的步骤值递增，直到但不包括终点，从而生成从指定的起始数开始的序列。
$range 有以下运算符表达式语法:
{ $range: [<start>, <end>, <non-zero step>] }

Example	Results
`{ $range: [ 0, 10, 2 ] }`	`[ 0, 2, 4, 6, 8 ]`
`{ $range: [ 10, 0, -2 ] }`	`[ 10, 8, 6, 4, 2 ]`
`{ $range: [ 0, 10, -2 ] }`	`[ ]`
`{ $range: [ 0, 5 ] }`	`[ 0, 1, 2, 3, 4 ]`

E:distances collection
{ _id: 0, city: "San Jose", distance: 42 }
{ _id: 1, city: "Sacramento", distance: 88 }
{ _id: 2, city: "Reno", distance: 218 }
{ _id: 3, city: "Los Angeles", distance: 383 }
一位骑自行车的人计划从旧金山骑到名单上列出的每个城市，并希望每 25 英里停下来休息一次。下面的聚合管道操作使用$range 操作符确定每次旅行的停止点。

```
db.distances.aggregate([{
    $project: {
        _id: 0,
        city: 1,
        "Rest stops": { $range: [ 0, "$distance", 25 ] }
    }
}])
```

R:
{ "city" : "San Jose", "Rest stops" : [0, 25] }
{ "city" : "Sacramento", "Rest stops" : [0, 25, 50, 75] }
{ "city" : "Reno", "Rest stops" : [0, 25, 50, 75, 100, 125, 150, 175, 200] }
{ "city" : "Los Angeles", "Rest stops" : [0, 25, 50, 75, 100, 125, 150, 175, 200, 225, 250, 275, 300, 325, 350, 375] }

$reduce

新版本 3.4。
将表达式应用于数组中的每个元素，并将它们组合成单个值。
$reduce 有以下语法:

```
{
    $reduce: {
        input: <array>,
        initialValue: <expression>,
        in: <expression>
    }
}
```

Input:
可以是解析为数组的**任何有效表达式**。有关表达式的更多信息，请参见表达式。
如果参数解析为 **null 值或引用缺失的字段，$reduce 返回 null。**
如果参数**没有解析为数组或 null，也没有引用丢失的字段，则$reduce 返回一个错误。**

Initialvalue ：
in 之前的**初始累积值集应用于输入数组的第一个元素**。

In:
一个有效的表达式，**$reduce 按从左到右的顺序应用于输入数组中的每个元素**。用 $reverseArray 包装输入值，**得到从右到左应用组合表达式的等价结果。**
在计算 in 表达式时，有两个变量可用:
value 是表示表达式的累积值的变量。
这是引用正在处理的元素的变量。

E:

`` { $reduce: { input: ["a", "b", "c"], initialValue: "", in: { $concat : ["$$value", "$$this"] } } } ```	"abc"
`` { $reduce: { input: [1, 2, 3, 4], initialValue: { sum: 5, product: 2 }, in: { sum: { $add : ["$$value.sum", "$$this"] }, product: { $multiply: ["$$value.product", "$$this"] } } } } ```	{ "sum" : 15, "product" : 48 }
`` { $reduce: { input: [[3, 4], [5, 6]], initialValue: [1, 2], in: { $concatArrays : ["$$value", "$$this"] } } } ```	[1, 2, 3, 4, 5, 6]

乘法

概率

一个名为 events 的集合包含一个概率实验的事件。每个实验都可以有多个事件，例如多次滚动一个骰子或连续绘制几张卡片(不替换)，以达到预期的结果。为了得到实验的整体概率，我们需要将实验中每个事件的概率相乘。

{_id:1, "type":"die", "experimentId":"r5", "description":"Roll a 5", "eventNum":1, "probability":0.16666666666667}

{_id:2, "type":"card", "experimentId":"d3rc", "description":"Draw 3 red cards", "eventNum":1, "probability":0.5}

{_id:3, "type":"card", "experimentId":"d3rc", "description":"Draw 3 red cards", "eventNum":2, "probability":0.49019607843137}

{_id:4, "type":"card", "experimentId":"d3rc", "description":"Draw 3 red cards", "eventNum":3, "probability":0.48}

{_id:5, "type":"die", "experimentId":"r16", "description":"Roll a 1 then a 6", "eventNum":1, "probability":0.16666666666667}

{_id:6, "type":"die", "experimentId":"r16", "description":"Roll a 1 then a 6", "eventNum":2, "probability":0.16666666666667}

{_id:7, "type":"card", "experimentId":"dak", "description":"Draw an ace, then a king", "eventNum":1, "probability":0.07692307692308}

{_id:8, "type":"card", "experimentId":"dak", "description":"Draw an ace, then a king", "eventNum":2, "probability":0.07843137254902}
步骤:
使用$group 对实验者进行分组，并使用$push 创建一个具有每个事件概率的数组。
使用$reduce 和$正乘来相乘，并将或然性 yarr 的元素组合成一个单独的值并对其进行投影。

```
db.probability.aggregate(
  [
    {
      $group: {
        _id: "$experimentId",
        "probabilityArr": { $push: "$probability" }
      }
    },
    {
      $project: {
        "description": 1,
        "results": {
          $reduce: {
            input: "$probabilityArr",
            initialValue: 1,
            in: { $multiply: [ "$$value", "$$this" ] }
          }
        }
      }
    }
  ]
)
```

R:results=initialvalue*this

```
{ "_id" : "dak", "results" : 0.00603318250377101 }
{ "_id" : "r5", "results" : 0.16666666666667 }
{ "_id" : "r16", "results" : 0.027777777777778886 }
{ "_id" : "d3rc", "results" : 0.11764705882352879 }
```

E2:打折的商品 collection clothes

```
{ "_id" : 1, "productId" : "ts1", "description" : "T-Shirt", "color" : "black", "size" : "M", "price" : 20, "discounts" : [ 0.5, 0.1 ] }
{ "_id" : 2, "productId" : "j1", "description" : "Jeans", "color" : "blue", "size" : "36", "price" : 40, "discounts" : [ 0.25, 0.15, 0.05 ] }
{ "_id" : 3, "productId" : "s1", "description" : "Shorts", "color" : "beige", "size" : "32", "price" : 30, "discounts" : [ 0.15, 0.05 ] }
{ "_id" : 4, "productId" : "ts2", "description" : "Cool T-Shirt", "color" : "White", "size" : "L", "price" : 25, "discounts" : [ 0.3 ] }
{ "_id" : 5, "productId" : "j2", "description" : "Designer Jeans", "color" : "blue", "size" : "30", "price" : 80, "discounts" : [ 0.1, 0.25 ] }
```

每个文档包含一个折扣数组，其中包含每个项目当前可用的折扣券。如果每个折扣可以应用

于产品一次, 我们可以使用$reduce 对折扣数组中的每个元素应用以下公式计算最低价格:(1 - discount) * price。

```
db.clothes.aggregate(
  [
    {
      $project: {
        "discountedPrice": {
          $reduce: {
            input: "$discounts",
            initialValue: "$price",
            in: { $multiply: [ "$$value", { $subtract: [ 1, "$$this" ] } ] }
          }
        }
      }
    }
  ]
)
```

R:

```
{ "_id" : ObjectId("57c893067054e6e47674ce01"), "discountedPrice" : 9 }
{ "_id" : ObjectId("57c9932b7054e6e47674ce12"), "discountedPrice" : 24.224999999999998 }
{ "_id" : ObjectId("57c993457054e6e47674ce13"), "discountedPrice" : 24.224999999999998 }
{ "_id" : ObjectId("57c993687054e6e47674ce14"), "discountedPrice" : 17.5 }
{ "_id" : ObjectId("57c993837054e6e47674ce15"), "discountedPrice" : 54 }
```

String Concatenation

People colletion

```
{ "_id" : 1, "name" : "Melissa", "hobbies" : [ "softball", "drawing", "reading" ] }
{ "_id" : 2, "name" : "Brad", "hobbies" : [ "gaming", "skateboarding" ] }
{ "_id" : 3, "name" : "Scott", "hobbies" : [ "basketball", "music", "fishing" ] }
{ "_id" : 4, "name" : "Tracey", "hobbies" : [ "acting", "yoga" ] }
{ "_id" : 5, "name" : "Josh", "hobbies" : [ "programming" ] }
{ "_id" : 6, "name" : "Claire" }
```

下面的例子将字符串的 hobbies 数组简化为一个字符串 bio:

```
db.people.aggregate(
    [
      // Filter to return only non-empty arrays
      { $match: { "hobbies": { $gt: [ ] } } },
      {
        $project: {
          "name": 1,
          "bio": {
            $reduce: {
              input: "$hobbies",
```

```
                initialValue: "My hobbies include:",
                in: {
                  $concat: [
                    "$$value",
                    {
                      $cond: {
                        if: { $eq: [ "$$value", "My hobbies include:" ] },
                        then: " ",
                        else: ", "
                      }
                    },
                    "$$this"
                  ]
                }
              }
            }
          }
        }
    ]
)
```

R:

```
{ "_id" : 1, "name" : "Melissa", "bio" : "My hobbies include: softball, drawing, reading" }
{ "_id" : 2, "name" : "Brad", "bio" : "My hobbies include: gaming, skateboarding" }
{ "_id" : 3, "name" : "Scott", "bio" : "My hobbies include: basketball, music, fishing" }
{ "_id" : 4, "name" : "Tracey", "bio" : "My hobbies include: acting, yoga" }
{ "_id" : 5, "name" : "Josh", "bio" : "My hobbies include: programming" }
```

Array Concatenation

collection named matrices

```
{ "_id" : 1, "arr" : [ [ 24, 55, 79 ], [ 14, 78, 35 ], [ 84, 90, 3 ], [ 50, 89, 70 ] ] }
{ "_id" : 2, "arr" : [ [ 39, 32, 43, 7 ], [ 62, 17, 80, 64 ], [ 17, 88, 11, 73 ] ] }
{ "_id" : 3, "arr" : [ [ 42 ], [ 26, 59 ], [ 17 ], [ 72, 19, 35 ] ] }
{ "_id" : 4 }
```

计算单个简化步骤

下面的例子将二维数组折叠成单个数组:

```
db.arrayconcat.aggregate(
  [
    {
      $project: {
        "collapsed": {
          $reduce: {
            input: "$arr",
            initialValue: [ ],
            in: { $concatArrays: [ "$$value", "$$this" ] }
          }
```

```
          }
        }
      }
  ]
)
```

R:

```
{ "_id" : 1, "collapsed" : [ 24, 55, 79, 14, 78, 35, 84, 90, 3, 50, 89, 70 ] }
{ "_id" : 2, "collapsed" : [ 39, 32, 43, 7, 62, 17, 80, 64, 17, 88, 11, 73 ] }
{ "_id" : 3, "collapsed" : [ 42, 26, 59, 17, 72, 19, 35 ] }
{ "_id" : 4, "collapsed" : null }
```

Computing a Multiple Reductions

```
db.arrayconcat.aggregate(
  [
    {
      $project: {
        "results": {
          $reduce: {
            input: "$arr",
            initialValue: [ ],
            in: {
              "collapsed": {
                $concatArrays: [ "$$value.collapsed", "$$this" ]
              },
              "firstValues": {
                $concatArrays: [ "$$value.firstValues", { $slice: [ "$$this", 1 ] } ]
              }
            }
          }
        }
      }
    }
  ]
)
```

R:

```
{ "_id" : 1, "results" : { "collapsed" : [ 24, 55, 79, 14, 78, 35, 84, 90, 3, 50, 89, 70 ],
"firstValues" : [ 24, 14, 84, 50 ] } }
{ "_id" : 2, "results" : { "collapsed" : [ 39, 32, 43, 7, 62, 17, 80, 64, 17, 88, 11, 73 ],
"firstValues" : [ 39, 62, 17 ] } }
{ "_id" : 3, "results" : { "collapsed" : [ 42, 26, 59, 17, 72, 19, 35 ], "firstValues" : [ 42, 26, 17,
72 ] } }
{ "_id" : 4, "results" : null }
```

$reverseArray

新版本 3.4。
接受数组表达式作为参数，并返回元素顺序相反的数组。
$reverseArray 有以下操作符表达式语法:
{ $reverseArray: <array expression> }
行为
如果参数解析为 null 值或引用缺失的字段，$reverseArray 返回 null。
如果参数没有解析为数组或 null，也没有引用丢失的字段，$reverseArray 将返回一个错误。
当参数为空数组时，$reverseArray 返回一个空数组。
如果参数包含子数组，$reverseArray 只在顶层数组元素上操作，不会反转子数组的内容。
E:users
{ "_id" : 1, "name" : "dave123", "favorites" : ["chocolate", "cake", "butter", "apples"] }
{ "_id" : 2, "name" : "li", "favorites" : ["apples", "pudding", "pie"] }
{ "_id" : 3, "name" : "ahn", "favorites" : [ ] }
{ "_id" : 4, "name" : "ty" }
下面的示例以相反的顺序返回一个包含 favorites 数组元素的数组:

```
db.users.aggregate([
    {
      $project:
       {
           name: 1,
           reverseFavorites: { $reverseArray: "$favorites" }
       }
    }
])
```

R:
{ "_id" : 1, "name" : "dave123", "reverseFavorites" : ["apples", "butter", "cake", "chocolate"] }
{ "_id" : 2, "name" : "li", "reverseFavorites" : ["pie", "pudding", "apples"] }
{ "_id" : 3, "name" : "ahn", "reverseFavorites" : [ ] }
{ "_id" : 4, "name" : "ty", "reverseFavorites" : null }

$rtrim

新版本 4.0。
从字符串末尾删除空白字符(包括 null)或指定的字符。
$rtrim 有以下语法:
{ $rtrim: { input: <string>, chars: <string> } }
E:inventory collection
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : " product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2 \n The product is in stock. \n\n " }

{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
下面的操作使用$rtrim 操作符从 description 字段中删除尾随空格:

```
db.inventory.aggregate([
    { $project: { item: 1, description: { $rtrim: { input: "$description" } } } }
])
```

R:

```
{ "_id" : 1, "item" : "ABC1", "description" : " product 1" }
{ "_id" : 2, "item" : "ABC2", "description" : "product 2 \n The product is in stock." }
{ "_id" : 3, "item" : "XYZ1", "description" : null }
```

$second

以 0 到 59 之间的数字返回日期的第二部分，但可以用 60 表示闰秒。
$second 表达式有以下操作符表达式语法:

```
{ $second: <dateExpression> }
```

在 3.6 版中进行了更改。
该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6

```
{ date: <dateExpression>, timezone: <tzExpression> }
```

$second cannot take a string as an argument.
E:sales

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$second 和其他日期表达式来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
```

```
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
        }
    }
  ]
)
```

R:

```
{
    "_id" : 1,
    "year" : 2014,
    "month" : 1,
    "day" : 1,
    "hour" : 8,
    "minutes" : 15,
    "seconds" : 39,
    "milliseconds" : 736,
    "dayOfYear" : 1,
    "dayOfWeek" : 4,
    "week" : 0
}
```

$setDifference

新版本 2.6

获取两个集合并返回一个数组，其中包含仅存在于第一个集合中的元素;即执行第二组相对于第一组的相对补码。

$setDifference 有以下语法:

{ $setDifference: [<expression1>, <expression2>] }

`{ $setDifference: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`	`[ ]`
`{ $setDifference: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`	`[ "a", "b" ]`

E:experiments collection

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

下面的操作使用$setDifference 操作符返回 B 数组中但不在 A 数组中的元素数组:

```
db.experiments.aggregate(
    [
      { $project: { A: 1, B: 1, inBOnly: { $setDifference: [ "$B", "$A" ] }, _id: 0 } }
    ]
)
```

R:

```
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue" ], "inBOnly" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ], "inBOnly" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ], "inBOnly" : [ "green" ] }
{ "A" : [ "red", "blue" ], "B" : [ "green", "red" ], "inBOnly" : [ "green" ] }
{ "A" : [ "red", "blue" ], "B" : [ ], "inBOnly" : [ ] }
{ "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ], "inBOnly" : [ [ "red" ], [ "blue" ] ] }
{ "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ], "inBOnly" : [ [ "red", "blue" ] ] }
{ "A" : [ ], "B" : [ ], "inBOnly" : [ ] }
{ "A" : [ ], "B" : [ "red" ], "inBOnly" : [ "red" ] }
```

$setEquals

新版本 2.6。

比较两个或多个数组，**如果它们具有相同元素，则返回 true，否则返回 false**。

$setEquals 有以下语法:

```
{ $setEquals: [ <expression1>, <expression2>, ... ] }
```

`{ $setEquals: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`	`true`
`{ $setEquals: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`	`false`

E:experiments

```
{ "_id" : 1, "A" : [ "red", "blue" ], "B" : [ "red", "blue" ] }
{ "_id" : 2, "A" : [ "red", "blue" ], "B" : [ "blue", "red", "blue" ] }
{ "_id" : 3, "A" : [ "red", "blue" ], "B" : [ "red", "blue", "green" ] }
{ "_id" : 4, "A" : [ "red", "blue" ], "B" : [ "green", "red" ] }
{ "_id" : 5, "A" : [ "red", "blue" ], "B" : [ ] }
{ "_id" : 6, "A" : [ "red", "blue" ], "B" : [ [ "red" ], [ "blue" ] ] }
{ "_id" : 7, "A" : [ "red", "blue" ], "B" : [ [ "red", "blue" ] ] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : [ "red" ] }
```

下面的操作使用$setEquals 操作符来确定 A 数组和 B 数组是否包含相同的元素:

```
db.experiments.aggregate(
    [
      { $project: { A: 1, B: 1, sameElements: { $setEquals: [ "$A", "$B" ] }, _id: 0 } }
    ]
)
```

R:
{ "A" : ["red", "blue"], "B" : ["red", "blue"], "sameElements" : true }
{ "A" : ["red", "blue"], "B" : ["blue", "red", "blue"], "sameElements" : true }
{ "A" : ["red", "blue"], "B" : ["red", "blue", "green"], "sameElements" : false }
{ "A" : ["red", "blue"], "B" : ["green", "red"], "sameElements" : false }
{ "A" : ["red", "blue"], "B" : [ ], "sameElements" : false }
{ "A" : ["red", "blue"], "B" : [["red"], ["blue"]], "sameElements" : false }
{ "A" : ["red", "blue"], "B" : [["red", "blue"]], "sameElements" : false }
{ "A" : [ ], "B" : [ ], "sameElements" : true }
{ "A" : [ ], "B" : ["red"], "sameElements" : false }

$setIntersection

新版本 2.6。
获取两个或多个数组，并返回一个数组，该数组包含每个输入数组中出现的元素。
$ set 交集的语法如下:
{ $setIntersection: [<array1>, <array2>, ...] }
E:experiments
{ "_id" : 1, "A" : ["red", "blue"], "B" : ["red", "blue"] }
{ "_id" : 2, "A" : ["red", "blue"], "B" : ["blue", "red", "blue"] }
{ "_id" : 3, "A" : ["red", "blue"], "B" : ["red", "blue", "green"] }
{ "_id" : 4, "A" : ["red", "blue"], "B" : ["green", "red"] }
{ "_id" : 5, "A" : ["red", "blue"], "B" : [ ] }
{ "_id" : 6, "A" : ["red", "blue"], "B" : [["red"], ["blue"]] }
{ "_id" : 7, "A" : ["red", "blue"], "B" : [["red", "blue"]] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : ["red"] }
下面的操作使用$ set 交集运算符返回 A 数组和 B 数组共有的元素数组:

```
db.experiments.aggregate(
    [
      { $project: { A: 1, B: 1, commonToBoth: { $setIntersection: [ "$A", "$B" ] }, _id: 0 } }
    ]
)
```

R:
{ "A" : ["red", "blue"], "B" : ["red", "blue"], "commonToBoth" : ["blue", "red"] }
{ "A" : ["red", "blue"], "B" : ["blue", "red", "blue"], "commonToBoth" : ["blue", "red"] }
{ "A" : ["red", "blue"], "B" : ["red", "blue", "green"], "commonToBoth" : ["blue", "red"] }
{ "A" : ["red", "blue"], "B" : ["green", "red"], "commonToBoth" : ["red"] }
{ "A" : ["red", "blue"], "B" : [ ], "commonToBoth" : [ ] }
{ "A" : ["red", "blue"], "B" : [["red"], ["blue"]], "commonToBoth" : [ ] }
{ "A" : ["red", "blue"], "B" : [["red", "blue"]], "commonToBoth" : [ ] }
{ "A" : [ ], "B" : [ ], "commonToBoth" : [ ] }
{ "A" : [ ], "B" : ["red"], "commonToBoth" : [ ] }

$setIsSubset

新版本 2.6。

获取两个数组，当第一个数组是第二个数组的子集时返回 true，包括当第一个数组等于第二个数组时返回 true，否则返回 false。
$ setisSubset 有以下语法:
{ $setIsSubset: [<expression1>, <expression2>] }
参数可以是任何有效的表达式，只要它们各自解析为一个数组。有关表达式的更多信息，请参见表达式。

Example	Result
`{ $setIsSubset: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`	`true`
`{ $setIsSubset: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`	`false`

E:experiments
{ "_id" : 1, "A" : ["red", "blue"], "B" : ["red", "blue"] }
{ "_id" : 2, "A" : ["red", "blue"], "B" : ["blue", "red", "blue"] }
{ "_id" : 3, "A" : ["red", "blue"], "B" : ["red", "blue", "green"] }
{ "_id" : 4, "A" : ["red", "blue"], "B" : ["green", "red"] }
{ "_id" : 5, "A" : ["red", "blue"], "B" : [ ] }
{ "_id" : 6, "A" : ["red", "blue"], "B" : [["red"], ["blue"]] }
{ "_id" : 7, "A" : ["red", "blue"], "B" : [["red", "blue"]] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : ["red"] }
下面的操作使用$ setissubset 操作符来确定 A 数组是否是 B 数组的子集:

```
db.experiments.aggregate(
    [
      { $project: { A:1, B: 1, AisSubset: { $setIsSubset: [ "$A", "$B" ] }, _id:0 } }
    ]
)
```

R:
{ "A" : ["red", "blue"], "B" : ["red", "blue"], "AisSubset" : true }
{ "A" : ["red", "blue"], "B" : ["blue", "red", "blue"], "AisSubset" : true }
{ "A" : ["red", "blue"], "B" : ["red", "blue", "green"], "AisSubset" : true }
{ "A" : ["red", "blue"], "B" : ["green", "red"], "AisSubset" : false }
{ "A" : ["red", "blue"], "B" : [ ], "AisSubset" : false }
{ "A" : ["red", "blue"], "B" : [["red"], ["blue"]], "AisSubset" : false }
{ "A" : ["red", "blue"], "B" : [["red", "blue"]], "AisSubset" : false }
{ "A" : [ ], "B" : [ ], "AisSubset" : true }
{ "A" : [ ], "B" : ["red"], "AisSubset" : true }

$setUnion

新版本 2.6。

获取两个或多个数组，并返回一个数组，其中包含出现在任何输入数组中的元素。

$setUnion 有以下语法:

参数可以是任何有效的表达式，只要它们各自解析为一个数组。有关表达式的更多信息，请参见表达式。

Behavior

$setUnion 对数组执行集合操作，将数组视为集合。如果数组包含重复项，$setUnion 将忽略重复项。$setUnion 忽略元素的顺序。

$setUnion 过滤掉结果中的重复项，以输出只包含唯一条目的数组。输出数组中元素的顺序未指定。

如果集合包含嵌套数组元素，则$setUnion 不会下降到嵌套数组中，而是在顶层计算数组。

`{ $setUnion: [ [ "a", "b", "a" ], [ "b", "a" ] ] }`	`[ "b", "a" ]`
`{ $setUnion: [ [ "a", "b" ], [ [ "a", "b" ] ] ] }`	`[ [ "a", "b" ], "b", "a" ]`

Experiments:

{ "_id" : 1, "A" : ["red", "blue"], "B" : ["red", "blue"] }
{ "_id" : 2, "A" : ["red", "blue"], "B" : ["blue", "red", "blue"] }
{ "_id" : 3, "A" : ["red", "blue"], "B" : ["red", "blue", "green"] }
{ "_id" : 4, "A" : ["red", "blue"], "B" : ["green", "red"] }
{ "_id" : 5, "A" : ["red", "blue"], "B" : [ ] }
{ "_id" : 6, "A" : ["red", "blue"], "B" : [["red"], ["blue"]] }
{ "_id" : 7, "A" : ["red", "blue"], "B" : [["red", "blue"]] }
{ "_id" : 8, "A" : [ ], "B" : [ ] }
{ "_id" : 9, "A" : [ ], "B" : ["red"] }

下面的操作使用$setUnion 操作符返回在 A 数组或 B 数组中找到的元素数组，或者两者都返回:

```
db.experiments.aggregate(
    [
      { $project: { A:1, B: 1, allValues: { $setUnion: [ "$A", "$B" ] }, _id: 0 } }
    ]
)
```

R:

{ "A": ["red", "blue"], "B": ["red", "blue"], "allValues": ["blue", "red"] }
{ "A": ["red", "blue"], "B": ["blue", "red", "blue"], "allValues": ["blue", "red"] }
{ "A": ["red", "blue"], "B": ["red", "blue", "green"], "allValues": ["blue", "red", "green"] }
{ "A": ["red", "blue"], "B": ["green", "red"], "allValues": ["blue", "red", "green"] }
{ "A": ["red", "blue"], "B": [ ], "allValues": ["blue", "red"] }
{ "A": ["red", "blue"], "B": [["red"], ["blue"]], "allValues": ["blue", "red", ["red"], ["blue"]] }
{ "A": ["red", "blue"], "B": [["red", "blue"]], "allValues": ["blue", "red", ["red",

"blue"]] }
{ "A": [ ], "B": [ ], "allValues": [ ] }
{ "A": [ ], "B": ["red"], "allValues": ["red"] }
$size
计算并返回数组中项的总数。
$size 的语法如下:
{ $size: <expression> }
$size 的参数可以是任何表达式，只要它解析为数组。有关表达式的更多信息，请参见表达式。
Behavior
$size 的参数必须解析为数组。如果$size 的参数丢失或没有解析为数组，则$size 错误。
E；inventory
{ "_id" : 1, "item" : "ABC1", "description" : "product 1", colors: ["blue", "black", "red"] }
{ "_id" : 2, "item" : "ABC2", "description" : "product 2", colors: ["purple"] }
{ "_id" : 3, "item" : "XYZ1", "description" : "product 3", colors: [ ] }
{ "_id" : 4, "item" : "ZZZ1", "description" : "product 4 - missing colors" }
{ "_id" : 5, "item" : "ZZZ2", "description" : "product 5 - colors is string", colors: "blue,red" }
下面的聚合管道操作使用$size 返回颜色数组中的元素数量:

```
db.inventory.aggregate([
    {
        $project: {
            item: 1,
            numberOfColors: { $cond: { if: { $isArray: "$colors" }, then: { $size: "$colors" }, else: "NA"} }
        }
    }
])
```

R:
{ "_id" : 1, "item" : "ABC1", "numberOfColors" : 3 }
{ "_id" : 2, "item" : "ABC2", "numberOfColors" : 1 }
{ "_id" : 3, "item" : "XYZ1", "numberOfColors" : 0 }
{ "_id" : 4, "item" : "ZZZ1", "numberOfColors" : "NA" }
{ "_id" : 5, "item" : "ZZZ2", "numberOfColors" : "NA" }·

$slice

新版本 3.2。
返回数组的子集。
$slice 有两种语法形式:
下面的语法从数组的开始或结束返回元素:
{ $slice: [<array>, <n>] }
下面的语法返回数组中指定位置的元素:
{ $slice: [<array>, <position>, <n>] }]

Position:可选的。任何有效表达式，只要它解析为整数。
如果为正数，$slice 将从数组的开始确定起始位置。如果<position>大于元素的数量，$slice 返回一个空数组。
如果是负数，$slice 将从数组的末尾确定起始位置。如果<position>的绝对值大于元素个数，则起始位置为数组的起始位置。

Example	Results
{ $slice: [[1, 2, 3], 1, 1] }	[2]
{ $slice: [[1, 2, 3], -2] }	[2, 3]
{ $slice: [[1, 2, 3], 15, 2] }	[ ]
{ $slice: [[1, 2, 3], -15, 2] }	[1, 2]

E:users
{ "_id" : 1, "name" : "dave123", favorites: ["chocolate", "cake", "butter", "apples"] }
{ "_id" : 2, "name" : "li", favorites: ["apples", "pudding", "pie"] }
{ "_id" : 3, "name" : "ahn", favorites: ["pears", "pecans", "chocolate", "cherries"] }
{ "_id" : 4, "name" : "ty", favorites: ["ice cream"] }

下面的示例最多为每个用户返回收藏夹数组中的前三个元素:

```
db.users.aggregate([
    { $project: { name: 1, threeFavorites: { $slice: [ "$favorites", 3 ] } } }
])
```

{ "_id" : 1, "name" : "dave123", "threeFavorites" : ["chocolate", "cake", "butter"] }
{ "_id" : 2, "name" : "li", "threeFavorites" : ["apples", "pudding", "pie"] }
{ "_id" : 3, "name" : "ahn", "threeFavorites" : ["pears", "pecans", "chocolate"] }
{ "_id" : 4, "name" : "ty", "threeFavorites" : ["ice cream"] }

$split

根据分隔符将字符串划分为子字符串数组。$split 删除分隔符，并返回作为数组元素的结果子字符串。如果在字符串中没有找到分隔符，$split 将返回原始字符串作为数组的唯一元素。
$split 有以下操作符表达式语法:
{ $split: [<string expression>, <delimiter>] }
例子如下

`{ $split: [ "June-15-2013", "-" ] }`	`[ "June", "15", "2013" ]`
`{ $split: [ "banana split", "a" ] }`	`[ "b", "n", "n", " split" ]`
`{ $split: [ "Hello World", " " ] }`	`[ "Hello", "World" ]`
`{ $split: [ "astronomical", "astro" ] }`	`[ "", "nomical" ]`
`{ $split: [ "pea green boat", "owl" ] }`	`[ "pea green boat" ]`
`{ $split: [ "headphone jack", 7 ] }`	Errors with message: `"$split requires an expression that evaluates to a string as a second argument, found: double"`
`{ $split: [ "headphone jack", /jack/ ] }`	Errors with message: `"$split requires an expression that evaluates to a string as a second argument, found: regex"`

E:deliveries

{ "_id" : 1, "city" : "Berkeley, CA", "qty" : 648 }
{ "_id" : 2, "city" : "Bend, OR", "qty" : 491 }
{ "_id" : 3, "city" : "Kensington, CA", "qty" : 233 }
{ "_id" : 4, "city" : "Eugene, OR", "qty" : 842 }
{ "_id" : 5, "city" : "Reno, NV", "qty" : 655 }
{ "_id" : 6, "city" : "Portland, OR", "qty" : 408 }
{ "_id" : 7, "city" : "Sacramento, CA", "qty" : 574 }

下面的聚合操作的目标是找到每个状态的交付总量，并按降序对列表进行排序。它分为五个阶段:

$project 阶段生成包含两个字段的文档，qty (integer)和 city_state (array)。$split 操作符使用空格(" ")作为分隔符，通过分割 city 字段创建字符串数组。

$unwind 阶段为 city_state 字段中的每个元素创建一个单独的记录。

$match 阶段使用正则表达式过滤城市文档，只留下包含状态的文档。

$group stage 将所有状态组合在一起，并对 qty 字段进行求和。

$sort 阶段按 total_qty 降序排列结果。

```
db.deliveries.aggregate([
   { $project : { city_state : { $split: ["$city", ", "] }, qty : 1 } },
   { $unwind : "$city_state" },
   { $match : { city_state : /[A-Z]{2}/ } },
   { $group : { _id: { "state" : "$city_state" }, total_qty : { "$sum" : "$qty" } } },
   { $sort : { total_qty : -1 } }
]);
```

R:

{ "_id" : { "state" : "OR" }, "total_qty" : 1741 }
{ "_id" : { "state" : "CA" }, "total_qty" : 1455 }
{ "_id" : { "state" : "NV" }, "total_qty" : 655 }

$sqrt:

新版本 3.2。
计算一个正数的平方根，并以双精度返回结果。
$sqrt 有以下语法:
参数可以是任何有效的表达式，只要它解析为一个非负数。有关表达式的更多信息，请参见表达式。
如果参数解析为 null 值或引用缺失的字段，$sqrt 返回 null。如果参数解析为 NaN， $sqrt 返回 NaN。
$sqrt 负数错误。

`{ $sqrt: 25 }`	`5`
`{ $sqrt: 30 }`	`5.477225575051661`
`{ $sqrt: null }`	`null`

E:points collection
{ _id: 1, p1: { x: 5, y: 8 }, p2: { x: 0, y: 5} }
{ _id: 2, p1: { x: -2, y: 1 }, p2: { x: 1, y: 5} }
{ _id: 3, p1: { x: 4, y: 4 }, p2: { x: 4, y: 0} }
下面的例子使用$sqrt 计算 p1 和 p2 之间的距离:

```
db.points.aggregate([
    {
       $project: {
           distance: {
               $sqrt: {
                    $add: [
                        { $pow: [ { $subtract: [ "$p2.y", "$p1.y" ] }, 2 ] },
                        { $pow: [ { $subtract: [ "$p2.x", "$p1.x" ] }, 2 ] }
                    ]
               }
           }
       }
    }
])
```

R:
{ "_id" : 1, "distance" : 5.830951894845301 }
{ "_id" : 2, "distance" : 5 }
{ "_id" : 3, "distance" : 4 }

$stdDevPop

新版本 3.2。
计算输入值的总体标准差。如果值包含要表示的数据的整个总体，而不希望泛化较大的总体，

则使用 if。$stdDevPop 忽略非数值。
如果这些值只表示用于概括总体的总体数据的样本，则使用**$stdDevSamp。**
$stdDevPop 在$group 和$project 阶段中可用。
在$group 阶段使用时，$stdDevPop 返回一组文档的指定表达式的总体标准差，这些文档按键共享相同的组，语法如下:
$stdDevPop 有一个指定的表达式作为操作数:
{ $stdDevPop: <expression> }
在$project 阶段使用时，$stdDevPop 返回每个文档的指定表达式或表达式列表的标准差，并具有以下两种语法之一:
$stdDevPop 有一个指定的表达式作为操作数:
{ $stdDevPop: <expression> }
$stdDevPop has a list of specified expressions as its operand:
{ $stdDevPop: [<expression1>, <expression2> ...] }
$stdDevPop 的参数可以是任何表达式，只要它解析为一个数组。有关表达式的更多信息，请参见表达式
Behavior
非数字值
$stdDevPop 忽略非数值。如果$stdDevPop 的所有操作数都是非数值的，则$stdDevPop 返回 null。
单值
如果示例包含一个数值，则$stdDevPop 返回 0。
数组操作数
在$group 阶段，如果表达式解析为数组，$stdDevPop 将操作数视为非数值。
在$project 阶段:
使用单个表达式作为操作数，如果表达式解析为数组，则$stdDevPop 将遍历数组，对数组的数字元素进行操作，以返回单个值。
以表达式列表作为操作数，如果任何表达式解析为数组，则$stdDevPop 不遍历数组，而是将数组视为非数值。

Use in $group Stage¶
Users collection
{ "_id" : 1, "name" : "dave123", "quiz" : 1, "score" : 85 }
{ "_id" : 2, "name" : "dave2", "quiz" : 1, "score" : 90 }
{ "_id" : 3, "name" : "ahn", "quiz" : 1, "score" : 71 }
{ "_id" : 4, "name" : "li", "quiz" : 2, "score" : 96 }
{ "_id" : 5, "name" : "annT", "quiz" : 2, "score" : 77 }
{ "_id" : 6, "name" : "ty", "quiz" : 2, "score" : 82 }
下面的例子计算了每个测验的标准差:

```
db.users.aggregate([
    { $group: { _id: "$quiz", stdDev: { $stdDevPop: "$score" } } }
])
```

R:
{ "_id" : 2, "stdDev" : 8.04155872120988 }
{ "_id" : 1, "stdDev" : 8.04155872120988 }

在$project Stage 实施阶段中使用

Quizzes collection

```
{
    "_id" : 1,
    "scores" : [
        {
            "name" : "dave123",
            "score" : 85
        },
        {
            "name" : "dave2",
            "score" : 90
        },
        {
            "name" : "ahn",
            "score" : 71
        }
    ]
}
{
    "_id" : 2,
    "scores" : [
        {
            "name" : "li",
            "quiz" : 2,
            "score" : 96
        },
        {
            "name" : "annT",
            "score" : 77
        },
        {
            "name" : "ty",
            "score" : 82
        }
    ]
}
```

下面的例子计算了每个测验的标准差:

```
db.quizzes.aggregate([
    { $project: { stdDev: { $stdDevPop: "$scores.score" } } }
])
```

R:

```
{ "_id" : 1, "stdDev" : 8.04155872120988 }
{ "_id" : 2, "stdDev" : 8.04155872120988 }
```

$stdDevSamp

新版本 3.2。
计算输入值的样本标准差。如果值包含一个总体数据的样本，则使用 if 来概括该总体。
$stdDevSamp 忽略非数值。
如果这些值表示数据的整个总体，或者您不希望泛化更大的总体，则使用$stdDevPop。
$stdDevSamp 可用于$group 和$project 阶段。
在$group 阶段使用$stdDevSamp 时，$stdDevSamp 具有以下语法，并返回按键共享相同组的一组文档的指定表达式的样本标准差:
{ $stdDevSamp: <expression> }
在$project 阶段使用时，$stdDevSamp 返回每个文档指定表达式或表达式列表的样本标准差，并具有以下两种语法之一:
$stdDevSamp 有一个指定的表达式作为它的操作数:
{ $stdDevSamp: <expression> }
$stdDevSamp 有一个指定表达式列表作为它的操作数:
{ $stdDevSamp: [<expression1>, <expression2> ...] }
$stdDevSamp 的参数可以是任何表达式，只要它解析为一个数组。有关表达式的更多信息，请参见表达式。
非数字值
$stdDevSamp 忽略非数值。如果 sum 的所有操作数都是非数值的，则$stdDevSamp 返回 null。
单值
如果示例包含一个数值，则$stdDevSamp 返回 null。
数组操作数
在$group 阶段，如果表达式解析为数组，$stdDevSamp 将操作数视为非数值。
在$project 阶段:
使用单个表达式作为操作数，如果表达式解析为数组，则$stdDevSamp 将遍历数组，对数组的数字元素进行操作，以返回单个值。
以表达式列表作为操作数，如果任何表达式解析为数组，则$stdDevSamp 不遍历数组，而是将数组视为非数值。
E:user
{_id: 0, username: "user0", age: 20}
{_id: 1, username: "user1", age: 42}
{_id: 2, username: "user2", age: 28}...
要计算用户样本的标准差，在进行聚合操作之后，首先使用$sample 管道对 100 个用户进行抽样，然后使用$stdDevSamp 计算抽样用户的标准差。

```
db.users.aggregate(
    [
        { $sample: { size: 100 } },
        { $group: { _id: null, ageStdDev: { $stdDevSamp: "$age" } } }
    ]
)
```

R:

```
{ "_id" : null, "ageStdDev" : 7.811258386185771 }
```

$strcasecmp

对两个字符串执行不区分大小写的比较。返回
1 如果第一个字符串“大于”第二个字符串。
0 如果两个字符串相等。
-1 如果第一个字符串“小于”第二个字符串。
$strcasecmp 的语法如下:
{ $strcasecmp: [<expression1>, <expression2>] }
Behavior
$strcasecmp 只对 ASCII 字符字符串具有定义良好的行为。
有关区分大小写的比较，请参见$cmp。
inventory collection
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
下面的操作使用$strcasecmp 操作符将 quarter 字段值与字符串“13q4”进行不区分大小写的比较:

```
db.inventory.aggregate(
    [
      {
        $project:
            {
              item: 1,
              comparisonResult: { $strcasecmp: [ "$quarter", "13q4" ] }
            }
       }
    ]
)
```

R:
{ "_id" : 1, "item" : "ABC1", "comparisonResult" : -1 }
{ "_id" : 2, "item" : "ABC2", "comparisonResult" : 0 }
{ "_id" : 3, "item" : "XYZ1", "comparisonResult" : 1 }

$strLenBytes

新版本 3.4。
返回指定字符串中 UTF-8 编码的字节数。
$strLenBytes 有以下运算符表达式语法:
{ $strLenBytes: <string expression> }
参数可以是任何有效的表达式，只要它解析为字符串。有关表达式的更多信息，请参见表达式。
如果参数解析为 null 值或引用缺失的字段，则$strLenBytes 返回一个错误。
Behavior

$strLenBytes 运算符计算字符串中 UTF-8 编码的字节数，其中每个字符可以使用 1 到 4 个字节。
例如，US-ASCII 字符使用一个字节编码。带有变音符号和附加拉丁字母字符(即英文字母之外的拉丁字符)的字符使用两个字节编码。中文、日文和韩文字符通常需要三个字节，unicode 的其他平面(表情符号、数学符号等)需要四个字节。
$strLenBytes 操作符与$strLenCP 操作符不同，后者计算指定字符串中的代码点，而不管每个字符使用多少字节。

`{ $strLenBytes: "abcde" }`	5	Each character is encoded using one byte.
`{ $strLenBytes: "Hello World!" }`	12	Each character is encoded using one byte.
`{ $strLenBytes: "cafeteria" }`	9	Each character is encoded using one byte.
`{ $strLenBytes: "cafétéria" }`	11	é is encoded using two bytes.
`{ $strLenBytes: "" }`	0	Empty strings return 0.
`{ $strLenBytes: "$€λG" }`	7	€ is encoded using three bytes. λ is encoded using two bytes.
`{ $strLenBytes: "寿司" }`	6	Each character is encoded using three bytes.

E:单字节和多字节字符集
Food collection
{ "_id" : 1, "name" : "apple" }
{ "_id" : 2, "name" : "banana" }
{ "_id" : 3, "name" : "éclair" }
{ "_id" : 4, "name" : "hamburger" }
{ "_id" : 5, "name" : "jalapeño" }
{ "_id" : 6, "name" : "pizza" }
{ "_id" : 7, "name" : "tacos" }
{ "_id" : 8, "name" : "寿司" }
下面的操作使用$strLenBytes 操作符来计算每个 name 值的长度:

```
db.food.aggregate(
  [
    {
      $project: {
        "name": 1,
        "length": { $strLenBytes: "$name" }
      }
    }
  ]
)
```

R:

{ "_id" : 1, "name" : "apple", "length" : 5 }
{ "_id" : 2, "name" : "banana", "length" : 6 }
{ "_id" : 3, "name" : "éclair", "length" : 7 }
{ "_id" : 4, "name" : "hamburger", "length" : 9 }
{ "_id" : 5, "name" : "jalapeño", "length" : 9 }
{ "_id" : 6, "name" : "pizza", "length" : 5 }
{ "_id" : 7, "name" : "tacos", "length" : 5 }
{ "_id" : 8, "name" : "寿司", "length" : 6 }
具有_id: 3和_id: 5的文档都包含一个需要编码两个字节的变音符号(分别为e和n)。具有_id: 8的文档包含两个日语字符，每个字符使用三个字节进行编码。这使得具有_id: 3、_id: 5和_id: 8的文档的长度大于名称中的字符数。

$strLenCP

新版本 3.4。
返回指定字符串中 UTF-8 代码点的数量。
$strLenCP 有以下运算符表达式语法:
{ $strLenCP: <string expression> }
参数可以是任何有效的表达式，只要它解析为字符串。有关表达式的更多信息，请参见表达式。
如果参数解析为 null 值或引用缺失的字段，$strLenCP 将返回一个错误。

`{ $strLenCP: "abcde" }`	5
`{ $strLenCP: "Hello World!" }`	12
`{ $strLenCP: "cafeteria" }`	9
`{ $strLenCP: "cafétéria" }`	9
`{ $strLenCP: "" }`	0
`{ $strLenCP: "$€ΛA" }`	4
`{ $strLenCP: "寿司" }`	2

Behavior
$strLenCP 操作符计算指定字符串中的代码点数量。这种行为与$strLenBytes 操作符不同，后者计算字符串中的字节数，其中每个字符使用 1 到 4 个字节。
E:单字节和多字节字符集
Food collection
{ "_id" : 1, "name" : "apple" }
{ "_id" : 2, "name" : "banana" }
{ "_id" : 3, "name" : "éclair" }
{ "_id" : 4, "name" : "hamburger" }
{ "_id" : 5, "name" : "jalapeño" }
{ "_id" : 6, "name" : "pizza" }
{ "_id" : 7, "name" : "tacos" }

{ "_id" : 8, "name" : "寿司" }
下面的操作使用$strLenCP 操作符来计算每个 name 值的长度:

```
db.food.aggregate(
  [
    {
      $project: {
        "name": 1,
        "length": { $strLenCP: "$name" }
      }
    }
  ]
)
```

R:

```
{ "_id" : 1, "name" : "apple", "length" : 5 }
{ "_id" : 2, "name" : "banana", "length" : 6 }
{ "_id" : 3, "name" : "éclair", "length" : 6 }
{ "_id" : 4, "name" : "hamburger", "length" : 9 }
{ "_id" : 5, "name" : "jalapeño", "length" : 8 }
{ "_id" : 6, "name" : "pizza", "length" : 5 }
{ "_id" : 7, "name" : "tacos", "length" : 5 }
{ "_id" : 8, "name" : "寿司", "length" : 2 }
```

$substr

自 3.4 版以来一直不推荐使用:$substr 现在是$substrBytes 的别名。
返回字符串的子字符串，从指定索引位置开始，并包含指定的字符数。索引是从零开始的。
$substr 有以下语法:

```
{ $substr: [ <string>, <start>, <length> ] }
```

如果<start>是负数，$substr 返回一个空字符串""。
如果<length>是负数，$substr 返回从指定索引开始的子字符串，并包含字符串的其余部分。
$substr 只对 ASCII 字符字符串具有定义良好的行为。
E:inventory collection

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
```

下面的操作使用$substr 操作符将 quarter 值分隔为 yearSubstring 和 quarterSubstring:
索引 0 开始 2 位，2 开始到后面的。

```
db.inventory.aggregate(
   [
     {
       $project:
          {
            item: 1,
            yearSubstring: { $substr: [ "$quarter", 0, 2 ] },
```

```
            quarterSubtring: { $substr: [ "$quarter", 2, -1 ] }
        }
    }
  ]
)
```

R:

```
{ "_id" : 1, "item" : "ABC1", "yearSubstring" : "13", "quarterSubtring" : "Q1" }
{ "_id" : 2, "item" : "ABC2", "yearSubstring" : "13", "quarterSubtring" : "Q4" }
{ "_id" : 3, "item" : "XYZ1", "yearSubstring" : "14", "quarterSubtring" : "Q2" }
```

$substrBytes

新版本 3.4。

返回字符串的子字符串。子字符串以字符串中指定的 UTF-8 字节索引(从零开始)处的字符开始，并继续指定字节数。

$substrBytes 有以下运算符表达式语法:

{ $substrBytes: [<string expression>, <byte index>, <byte count>] }

Behavior

$substrBytes 操作符使用 UTF-8 编码字节的索引，其中每个代码点或字符可以使用 1 到 4 个字节进行编码。

例如，US-ASCII 字符使用一个字节编码。**带有变音符号和附加拉丁字母字符(即英文字母之外的拉丁字符)的字符使用两个字节编码。**中文、日文和韩文字符通常需要三个字节, unicode 的其他平面(表情符号、数学符号等)需要四个字节。

注意字符串表达式中的内容很重要，因为提供位于 UTF-8 字符中间的字节索引或字节计数将导致错误。

$substrBytes 与$substrCP 的不同之处在于，$substrBytes 计算每个字符的字节数，而 $substrCP 计算代码点或字符，而不管字符使用了多少字节。

`{ $substrBytes: [ "abcde", 1, 2 ] }`	`"bc"`
`{ $substrBytes: [ "Hello World!", 6, 5 ] }`	`"World"`
`{ $substrBytes: [ "cafétéria", 0, 5 ] }`	`"café"`
`{ $substrBytes: [ "cafétéria", 5, 4 ] }`	`"tér"`
`{ $substrBytes: [ "cafétéria", 7, 3 ] }`	Errors with message: `"Error: Invalid range, starting index is a UTF-8 continuation byte."`
`{ $substrBytes: [ "cafétéria", 3, 1 ] }`	Errors with message: `"Error: Invalid range, ending index is in the middle of a UTF-8 character."`

E:single-byte character set
Inventory collection
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
下面的操作使用$substrBytes 操作符将 quarter 值(只包含单个字节的 US-ASCII 字符)分隔为 yearSubstring 和 quarterSubstring。quarterSubstring 字段表示 yearSubstring 后面指定字节索引的字符串的其余部分。它是通过使用$strLenBytes 从字符串长度中减去字节索引来计算的。

```
db.inventory.aggregate(
  [
    {
      $project: {
        item: 1,
        yearSubstring: { $substrBytes: [ "$quarter", 0, 2 ] },
        quarterSubtring: {
          $substrBytes: [
            "$quarter", 2, { $subtract: [ { $strLenBytes: "$quarter" }, 2 ] }
          ]
        }
      }
    }
  ]
)
```

R:
{ "_id" : 1, "item" : "ABC1", "yearSubstring" : "13", "quarterSubtring" : "Q1" }
{ "_id" : 2, "item" : "ABC2", "yearSubstring" : "13", "quarterSubtring" : "Q4" }
{ "_id" : 3, "item" : "XYZ1", "yearSubstring" : "14", "quarterSubtring" : "Q2" }
单字节和多字节字符集
名为 food 的集合包含以下文件:
{ "_id" : 1, "name" : "apple" }
{ "_id" : 2, "name" : "banana" }
{ "_id" : 3, "name" : "éclair" }
{ "_id" : 4, "name" : "hamburger" }
{ "_id" : 5, "name" : "jalapeño" }
{ "_id" : 6, "name" : "pizza" }
{ "_id" : 7, "name" : "tacos" }
{ "_id" : 8, "name" : "寿司 sushi" }
下面的操作使用$substrBytes 操作符从 name 值创建一个三个字节的 menuCode:

```
db.food.aggregate(
  [
    {
      $project: {
        "name": 1,
```

```
            "menuCode": { $substrBytes: [ "$name", 0, 3 ] }
        }
    }
  ]
)
```

R:

```
{ "_id" : 1, "name" : "apple", "menuCode" : "app" }
{ "_id" : 2, "name" : "banana", "menuCode" : "ban" }
{ "_id" : 3, "name" : "éclair", "menuCode" : "éc" }
{ "_id" : 4, "name" : "hamburger", "menuCode" : "ham" }
{ "_id" : 5, "name" : "jalapeño", "menuCode" : "jal" }
{ "_id" : 6, "name" : "pizza", "menuCode" : "piz" }
{ "_id" : 7, "name" : "tacos", "menuCode" : "tac" }
{ "_id" : 8, "name" : "寿司 sushi", "menuCode" : "寿" }
```

$substrCP

返回字符串的子字符串。子字符串以字符串中指定的 UTF-8 代码点(CP)索引(从零开始)处的字符开始，用于指定代码点的数量。

$substrCP 有以下运算符表达式语法:

{ $substrCP: [<string expression>, <code point index>, <code point count>] }

```
{ $substrCP: [ "abcde", 1, 2 ] }         "bc"

{ $substrCP: [ "Hello World!", 6, 5 ] }  "World"

{ $substrCP: [ "cafétéria", 0, 5 ] }     "café"

{ $substrCP: [ "cafétéria", 5, 4 ] }     "téri"

{ $substrCP: [ "cafétéria", 7, 3 ] }     "ia"

{ $substrCP: [ "cafétéria", 3, 1 ] }     "é"
```

Behavior

$substrCP 操作符使用代码点来提取子字符串。这种行为与$substrBytes 操作符不同，后者根据字节数提取子字符串，其中每个字符使用 1 到 4 个字节。

E:single-byte character set

Inventory collection

```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2" }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
```

下面的操作使用$substrCP 操作符将 quarter 值分隔为 yearSubstring 和 quarterSubstring。

quarterSubstring 字段表示 yearSubstring 后面指定字节索引的字符串的其余部分。它是通过使用$strLenCP 从字符串长度中减去字节索引来计算的。

```
db.inventory.aggregate(
  [
    {
      $project: {
        item: 1,
        yearSubstring: { $substrCP: [ "$quarter", 0, 2 ] },
        quarterSubtring: {
          $substrCP: [
            "$quarter", 2, { $subtract: [ { $strLenCP: "$quarter" }, 2 ] }
          ]
        }
      }
    }
  ]
)
```

R:

```
{ "_id" : 1, "item" : "ABC1", "yearSubstring" : "13", "quarterSubtring" : "Q1" }
{ "_id" : 2, "item" : "ABC2", "yearSubstring" : "13", "quarterSubtring" : "Q4" }
{ "_id" : 3, "item" : "XYZ1", "yearSubstring" : "14", "quarterSubtring" : "Q2" }
```

Single-Byte and Multibyte Character Set

Collection food

```
{ "_id" : 1, "name" : "apple" }
{ "_id" : 2, "name" : "banana" }
{ "_id" : 3, "name" : "éclair" }
{ "_id" : 4, "name" : "hamburger" }
{ "_id" : 5, "name" : "jalapeño" }
{ "_id" : 6, "name" : "pizza" }
{ "_id" : 7, "name" : "tacos" }
{ "_id" : 8, "name" : "寿司 sushi" }
```

下面的例子使用$substrCP 操作符从 name 值创建一个三个字节的 menuCode:

```
db.food.aggregate(
  [
    {
      $project: {
          "name": 1,
          "menuCode": { $substrCP: [ "$name", 0, 3 ] }
      }
    }
  ]
)
```

R:

```
{ "_id" : 1, "name" : "apple", "menuCode" : "app" }
```

{ "_id" : 2, "name" : "banana", "menuCode" : "ban" }
{ "_id" : 3, "name" : "éclair", "menuCode" : "écl" }
{ "_id" : 4, "name" : "hamburger", "menuCode" : "ham" }
{ "_id" : 5, "name" : "jalapeño", "menuCode" : "jal" }
{ "_id" : 6, "name" : "pizza", "menuCode" : "piz" }
{ "_id" : 7, "name" : "tacos", "menuCode" : "tac" }
{ "_id" : 8, "name" : "寿司 sushi", "menuCode" : "寿司 s" }

$subtract

减去两个数字以返回差值，或者两个日期以毫秒为单位返回差值，或者一个日期和一个数字以毫秒为单位返回结果日期。
$ minus 表达式的语法如下:
{ $subtract: [<expression1>, <expression2>] }
第二个参数从第一个参数中减去。
参数可以是任何有效的表达式，只要它们解析为数字和/或日期。要从日期中减去一个数字，日期必须是第一个参数。有关表达式的更多信息，请参见表达式。
E:sales collection
{ "_id" : 1, "item" : "abc", "price" : 10, "fee" : 2, "discount" : 5, "date" : ISODate("2014-03-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "fee" : 1, "discount" : 2, "date" : ISODate("2014-03-01T09:00:00Z") }
减去数字
下面的聚合使用$ minus 表达式，通过从价格和费用的小计中减去折扣来计算总数。
db.sales.aggregate([{ $project: { item: 1, total: { $subtract: [{ $add: ["$price", "$fee"] }, "$discount"] } } }])
R:
{ "_id" : 1, "item" : "abc", "total" : 7 }
{ "_id" : 2, "item" : "jkl", "total" : 19 }
减去两个日期
下面的聚合使用$ minus 表达式从当前日期中减去$date，并以毫秒为单位返回差值:
db.sales.aggregate([{ $project: { item: 1, dateDifference: { $subtract: [new Date(), "$date"] } } }])
R:
{ "_id" : 1, "item" : "abc", "dateDifference" : NumberLong("11713985194") }
{ "_id" : 2, "item" : "jkl", "dateDifference" : NumberLong("11710385194") }
从日期中减去几毫秒
下面的聚合使用$ minus 表达式从“$date”字段中减去 5 * 60 * 1000 毫秒(5 分钟):
db.sales.aggregate([{ $project: { item: 1, dateDifference: { $subtract: ["$date", 5 * 60 * 1000] } } }])
R:
{ "_id" : 1, "item" : "abc", "dateDifference" : ISODate("2014-03-01T07:55:00Z") }
{ "_id" : 2, "item" : "jkl", "dateDifference" : ISODate("2014-03-01T08:55:00Z") }

$Sum：

计算并返回数值的和。$sum 忽略非数值。
在 3.2 版本中进行了更改:$sum 可用于$group 和$project 阶段。在以前版本的 MongoDB 中，$sum 只在$group 阶段可用。
在$group 阶段使用$sum 时，$sum 具有以下语法，并返回将指定的表达式应用于一组按键共享相同组的文档中的每个文档所产生的所有数值的总和:
{ $sum: <expression> }
在$project 阶段使用时，$sum 返回每个文档指定表达式或表达式列表的总和，并具有以下两种语法之一:
$sum 有一个指定的表达式作为操作数:
{ $sum: <expression> }
$sum 有列表作为操作数:
{ $sum: [<expression1>, <expression2> ...] }

Example	Field Values	Results
{ $sum : <field> }	Numeric	Sum of Values
{ $sum : <field> }	Numeric and Non-Numeric	Sum of Numeric Values
{ $sum : <field> }	Non-Numeric or Non-Existent	0

数组操作数
在$group 阶段，如果表达式解析为数组，$sum 将操作数视为非数值。
在$project 阶段:
使用单个表达式作为操作数，如果表达式解析为数组，$sum 将遍历数组，对数组的数字元素进行操作，以返回单个值。
以表达式列表作为操作数，如果任何表达式解析为数组，$sum 将不遍历数组，而是将数组视为非数值。
Use in $group stage
Sale collection
{ "_id" : 1, "item" : "abc", "price" : 10, "quantity" : 2, "date" : ISODate("2014-01-01T08:00:00Z") }
{ "_id" : 2, "item" : "jkl", "price" : 20, "quantity" : 1, "date" : ISODate("2014-02-03T09:00:00Z") }
{ "_id" : 3, "item" : "xyz", "price" : 5, "quantity" : 5, "date" : ISODate("2014-02-03T09:05:00Z") }
{ "_id" : 4, "item" : "abc", "price" : 10, "quantity" : 10, "date" : ISODate("2014-02-15T08:00:00Z") }
{ "_id" : 5, "item" : "xyz", "price" : 5, "quantity" : 10, "date" : ISODate("2014-02-15T09:05:00Z") }
按日期字段中的日期和年份对文档进行分组，下面的操作使用$sum 累加器计算每组文档的总金额和计数。
db.sales.aggregate(

```
    [
      {
        $group:
          {
            _id: { day: { $dayOfYear: "$date"}, year: { $year: "$date" } },
            totalAmount: { $sum: { $multiply: [ "$price", "$quantity" ] } },
            count: { $sum: 1 }
          }
      }
    ]
)
```

R:

```
{ "_id" : { "day" : 46, "year" : 2014 }, "totalAmount" : 150, "count" : 2 }
{ "_id" : { "day" : 34, "year" : 2014 }, "totalAmount" : 45, "count" : 2 }
{ "_id" : { "day" : 1, "year" : 2014 }, "totalAmount" : 20, "count" : 1 }
```

对不存在的字段使用$sum 将返回 0 值。以下操作试图对 qty 进行$sum 操作

User in $project stage

Collection students

```
{ "_id": 1, "quizzes": [ 10, 6, 7 ], "labs": [ 5, 8 ], "final": 80, "midterm": 75 }
{ "_id": 2, "quizzes": [ 9, 10 ], "labs": [ 8, 8 ], "final": 95, "midterm": 80 }
{ "_id": 3, "quizzes": [ 4, 5, 5 ], "labs": [ 6, 5 ], "final": 78, "midterm": 70 }
```

下面的例子使用$project 阶段中的$sum 来计算测试总分、实验室总分、期末和期中总分:

```
db.students.aggregate([
    {
      $project: {
        quizTotal: { $sum: "$quizzes"},
        labTotal: { $sum: "$labs" },
        examTotal: { $sum: [ "$final", "$midterm" ] }
      }
    }
])
```

R:

```
{ "_id" : 1, "quizTotal" : 23, "labTotal" : 13, "examTotal" : 155 }
{ "_id" : 2, "quizTotal" : 19, "labTotal" : 16, "examTotal" : 175 }
{ "_id" : 3, "quizTotal" : 14, "labTotal" : 11, "examTotal" : 148 }
```

在$project 阶段:

使用单个表达式作为操作数，如果表达式解析为数组，$sum 将遍历数组，对数组的数字元素进行操作，以返回单个值。

以表达式列表作为操作数，如果任何表达式解析为数组，$sum 将不遍历数组，而是将数组视为非数值。

$switch

新版本 3.4。

计算一系列大小写表达式。当它找到一个计算结果为 true 的表达式时，$switch 执行一个指定的表达式并跳出控制流。

$switch 的语法如下:

```
$switch: {
    branches: [
        { case: <expression>, then: <expression> },
        { case: <expression>, then: <expression> },
        ...
    ],
    default: <expression>
}
```

branches

操作数描述

分支机构

控件分支文档的数组。每个分支都是一个包含以下字段的文档:

case

可以是任何解析为布尔值的有效表达式。如果结果不是布尔值，则强制转换为布尔值。有关 MongoDB 如何将表达式计算为 true 或 false 的更多信息可以在这里找到。

then

可以是任何有效的表达式。

分支数组必须包含至少一个分支文档。

Behavior

不同的 case 语句不需要相互排斥。$switch 执行它找到的第一个求值为 true 的分支。如果没有一个分支计算为 true，则$switch 执行默认选项。

以下条件导致$switch 失败并出现错误:

分支字段丢失或不是一个至少有一个条目的数组。

分支数组中的对象不包含大小写字段。

分支数组中的对象不包含 then 字段。

分支数组中的对象包含 case 或 then 之外的字段。

没有指定默认值，也没有大小写计算为 true。

```
{
   $switch: {
      branches: [
         { case: { $eq: [ 0, 5 ] }, then: "equals" },
         { case: { $gt: [ 0, 5 ] }, then: "greater than" },
         { case: { $lt: [ 0, 5 ] }, then: "less than" }
      ]
   }
}
```

```
"less than"
```

```
{
   $switch: {
      branches: [
         { case: { $eq: [ 0, 5 ] }, then: "equals" },
         { case: { $gt: [ 0, 5 ] }, then: "greater than" }
      ],
      default: "Did not match"
   }
}
```

```
"Did not match"
```

```
{
   $switch: {
      branches: [
         { case: "this is true", then: "first case" },
         { case: false, then: "second case" }
      ],
      default: "Did not match"
   }
}
```

```
"First case"
```

Grades collection

```
{ "_id" : 1, "name" : "Susan Wilkes", "scores" : [ 87, 86, 78 ] }
{ "_id" : 2, "name" : "Bob Hanna", "scores" : [ 71, 64, 81 ] }
{ "_id" : 3, "name" : "James Torrelio", "scores" : [ 91, 84, 97 ] }
```

下面的聚合操作使用$switch 根据每个学生的平均分显示特定的消息。

```
db.grades.aggregate( [
  {
    $project:
      {
        "name" : 1,
        "summary" :
        {
          $switch:
            {
              branches: [
                {
                  case: { $gte : [ { $avg : "$scores" }, 90 ] },
                  then: "Doing great!"
                },
                {
                  case: { $and : [ { $gte : [ { $avg : "$scores" }, 80 ] },
                                   { $lt : [ { $avg : "$scores" }, 90 ] } ] },
                  then: "Doing pretty well."
                },
                {
                  case: { $lt : [ { $avg : "$scores" }, 80 ] },
                  then: "Needs improvement."
                }
              ],
              default: "No scores found."
            }
        }
      }
  }
] )
```

E:

```
{ "_id" : 1, "name" : "Susan Wilkes", "summary" : "Doing pretty well." }
{ "_id" : 2, "name" : "Bob Hanna", "summary" : "Needs improvement." }
{ "_id" : 3, "name" : "James Torrelio", "summary" : "Doing great!" }
```

$toBool

新版本 4.0。

将值转换为布尔值。
$toBool 有以下语法:

```
{
    $toBool: <expression>
}
```

$toBool 接受任何有效表达式。
$toBool 是以下$convert 表达式的缩写:

```
{ $convert: { input: <expression>, to: "bool" } }
```

`{$toBool: false}`	false
`{$toBool: 1.99999}`	true
`{$toBool: NumberDecimal("5")}`	true
`{$toBool: NumberDecimal("0")}`	false
`{$toBool: 100}`	true
`{$toBool: ISODate("2018-03-26T04:38:28.044Z")}`	true
`{$toBool: "false"}`	true
`{$toBool: ""}`	true
`{$toBool: null}`	null

E:orders

```
db.orders.insert( [
    { _id: 1, item: "apple",    qty: 5, shipped: true },
    { _id: 2, item: "pie",    qty: 10, shipped: 0    },
    { _id: 3, item: "ice cream", shipped: 1 },
    { _id: 4, item: "almonds", qty: 2, shipped: "true" },
    { _id: 5, item: "pecans", shipped: "false" },    // Note: All strings convert to true
    { _id: 6, item: "nougat", shipped: ""    }            // Note: All strings convert to true
])
```

在找到未发货订单之前，订单集合上的以下聚合操作将发货的订单转换为布尔值:

```
// Define stage to add convertedShippedFlag field with the converted shipped value
// Because all strings convert to true, include specific handling for "false" and ""

shippedConversionStage = {
    $addFields: {
        convertedShippedFlag: {
            $switch: {
                branches: [
                   { case: { $eq: [ "$shipped", "false" ] }, then: false } ,
                   { case: { $eq: [ "$shipped", "" ] }, then: false }
```

```
                ],
                default: { $toBool: "$shipped" }
            }
        }
    }
};

// Define stage to filter documents and pass only the unshipped orders

unshippedMatchStage = {
    $match: { "convertedShippedFlag": false }
};

db.orders.aggregate( [
  shippedConversionStage,
  unshippedMatchStage
])
```

R:

```
{ "_id" : 2, "item" : "pie", "qty" : 10, "shipped" : 0, "convertedShippedFlag" : false }
{ "_id" : 5, "item" : "pecans", "shipped" : "false", "convertedShippedFlag" : false }
{ "_id" : 6, "item" : "nougat", "shipped" : "", "convertedShippedFlag" : false }
```

如果转换操作遇到错误，聚合操作将停止并抛出错误。要覆盖此行为，请使用$convert。

$toDate

新版本 4.0。

将值转换为日期。如果值不能转换为日期，则$toDate 错误。如果该值为 null 或缺失，则 $toDate 返回 null。

$toDate 的语法如下:

```
{
    $toDate: <expression>
}
```

$toDate 接受任何有效表达式。

$toDate 是以下$convert 表达式的缩写:

```
{ $convert: { input: <expression>, to: "date" } }
```

{$toDate: 120000000000.5}	ISODate("1973-10-20T21:20:00Z")
{$toDate: NumberDecimal("1253372036000.50")}	ISODate("2009-09-19T14:53:56Z")
{$toDate: NumberLong("1100000000000")}	ISODate("2004-11-09T11:33:20Z")
{$toDate: NumberLong("-1100000000000")}	ISODate("1935-02-22T12:26:40Z")
{$toDate: ObjectId("5ab9c3da31c2ab715d421285")}	ISODate("2018-03-27T04:08:58Z")
{$toDate: "2018-03-03"}	ISODate("2018-03-03T00:00:00Z")
{$toDate: "2018-03-20 11:00:06 +0500"}	ISODate("2018-03-20T06:00:06Z")
{$toDate: "Friday"}	Error

E：orders

```
db.orders.insert( [
    { _id: 1, item: "apple", qty: 5, order_date: new Date("2018-03-10") },
    { _id: 2, item: "pie", qty: 10,   order_date: new Date("2018-03-12")},
    { _id: 3, item: "ice cream", qty: 2, price: "4.99", order_date: "2018-03-05" },
    { _id: 4, item: "almonds" ,   qty: 5, price: 5,   order_date: "2018-03-05 +10:00"}
] )
```

以下对 orders 集合的聚合操作将 order_date 转换为 date，然后按 date 值排序:

```
// Define stage to add convertedDate field with the converted order_date value

dateConversionStage = {
    $addFields: {
        convertedDate: { $toDate: "$order_date" }
    }
};

// Define stage to sort documents by the converted date

sortStage = {
    $sort: { "convertedDate": 1 }
};

db.orders.aggregate( [
    dateConversionStage,
    sortStage
])
```

R:
{ "_id" : 4, "item" : "almonds", "qty" : 5, "price" : 5, "order_date" : "2018-03-05 +10:00", "convertedDate" : ISODate("2018-03-04T14:00:00Z") }
{ "_id" : 3, "item" : "ice cream", "qty" : 2, "price" : "4.99", "order_date" : "2018-03-05", "convertedDate" : ISODate("2018-03-05T00:00:00Z") }
{ "_id" : 1, "item" : "apple", "qty" : 5, "order_date" : ISODate("2018-03-10T00:00:00Z"), "convertedDate" : ISODate("2018-03-10T00:00:00Z") }
{ "_id" : 2, "item" : "pie", "qty" : 10, "order_date" : ISODate("2018-03-12T00:00:00Z"), "convertedDate" : ISODate("2018-03-12T00:00:00Z") }

$toDecimal

新版本 4.0。
将值转换为小数。如果该值不能转换为十进制，则$toDecimal 错误。如果该值为 null 或缺失，$toDecimal 返回 null。
$toDecimal 有以下语法:

```
{
    $toDecimal: <expression>
}
```

$toDecimal 接受任何有效表达式。
$toDecimal 是以下$convert 表达式的简写:

Example	Results
{$toDecimal: true}	NumberDecimal("1")
{$toDecimal: false}	NumberDecimal("0")
{$toDecimal: 2.5}	NumberDecimal("2.50000000000000")
{$toDecimal: NumberInt(5)}	NumberDecimal("5")
{$toDecimal: NumberLong(10000)}	NumberDecimal("10000")
{$toDecimal: "-5.5"}	NumberDecimal("-5.5")
{$toDecimal: ISODate("2018-03-27T05:04:47.890Z")}	NumberDecimal("1522127087890")

E:orders collcetion

```
db.orders.insert( [
    { _id: 1, item: "apple", qty: 5, price: 10 },
    { _id: 2, item: "pie", qty: 10, price: NumberDecimal("20.0") },
    { _id: 3, item: "ice cream", qty: 2, price: "4.99" },
    { _id: 4, item: "almonds",   qty: 5, price: 5 }
] )
```

订单集合的下面聚合操作在计算总价之前将价格转换为小数，将 qty 转换为整数:

```
// Define stage to add convertedPrice and convertedQty fields with the converted price and qty values
```

```
priceQtyConversionStage = {
    $addFields: {
        convertedPrice: { $toDecimal: "$price" },
        convertedQty: { $toInt: "$qty" },
    }
};

// Define stage to calculate total price by multiplying convertedPrice and convertedQty
fields


totalPriceCalculationStage = {
    $project: { item: 1, totalPrice: { $multiply: [ "$convertedPrice", "$convertedQty" ] } }
};

db.orders.aggregate( [
    priceQtyConversionStage,
    totalPriceCalculationStage
])
R:
{ "_id" : 1, "item" : "apple", "totalPrice" : NumberDecimal("50.0000000000000") }
{ "_id" : 2, "item" : "pie", "totalPrice" : NumberDecimal("200.0") }
{ "_id" : 3, "item" : "ice cream", "totalPrice" : NumberDecimal("9.98") }
{ "_id" : 4, "item" : "almonds", "totalPrice" : NumberDecimal("25.00000000000000") }
```

$toDouble

新版本 4.0。
将值转换为 double。如果值不能转换为 double，则$toDouble 错误。如果该值为 null 或缺失，$toDouble 返回 null。
$toDouble 有以下语法:

```
{
    $toDouble: <expression>
}
```

$toDouble 接受任何有效表达式。
$toDouble 是以下$convert 表达式的缩写:

```
{ $convert: { input: <expression>, to: "double" } }
```

Example	Results
`$toDouble: true`	1
`$toDouble: false`	0
`$toDouble: 2.5`	2.5
`$toDouble: NumberInt(5)`	5
`$toDouble: NumberLong(10000)`	10000
`$toDouble: "-5.5"`	-5.5
`$toDouble: ISODate("2018-03-27T05:04:47.890Z")`	1522127087890

E:weather

```
db.weather.insert( [
    { _id: 1, date: new Date("2018-06-01"), temp: "26.1C" },
    { _id: 2,   date: new Date("2018-06-02"), temp: "25.1C" },
    { _id: 3,   date: new Date("2018-06-03"), temp: "25.4C" },
] )
```

下面对 weather collection 的聚合操作解析 temp 值并将其转换为 double:

```
// Define stage to add degrees field with converted value

tempConversionStage = {
    $addFields: {
        degrees: { $toDouble: { $substrBytes: [ "$temp", 0, 4 ] } }
    }
};

db.weather.aggregate( [
    tempConversionStage,
])
```

R:

```
{ "_id" : 1, "date" : ISODate("2018-06-01T00:00:00Z"), "temp" : "26.1C", "degrees" : 26.1 }
{ "_id" : 2, "date" : ISODate("2018-06-02T00:00:00Z"), "temp" : "25.1C", "degrees" : 25.1 }
{ "_id" : 3, "date" : ISODate("2018-06-03T00:00:00Z"), "temp" : "25.4C", "degrees" : 25.4 }
```

$toInt

新版本 4.0。

将值转换为整数。如果该值不能转换为整数，则$toInt 错误。如果该值为 null 或缺失，$toInt 返回 null。

$toInt 有以下语法:

```
{
    $toInt: <expression>
}
```

$toInt 接受任何有效表达式。

$toInt 是以下$convert 表达式的缩写:

{ $convert: { input: <expression>, to: "int" } }

`$toInt: true`	1
`$toInt: false`	0
`$toInt: 1.99999`	1
`$toInt: NumberDecimal("5.5000")`	5
`$toInt: NumberDecimal("9223372036000.000")`	Error
`$toInt: NumberLong("5000")`	5000
`$toInt: NumberLong("922337203600")`	Error
`$toInt: "-2"`	-2
`$toInt: "2.5"`	Error
`$toInt: null`	null

E:orders

```
db.orders.insert( [
    { _id: 1, item: "apple", qty: 5, price: 10 },
    { _id: 2, item: "pie", qty: 10, price: NumberDecimal("20.0") },
    { _id: 3, item: "ice cream", qty: 2, price: "4.99" },
    { _id: 4, item: "almonds" ,   qty: 5, price: 5 }
] )
```

订单集合的以下聚合操作将 qty 转换为整数，并在计算总价之前将 price 转换为小数:

```
// Define stage to add convertedPrice and convertedQty fields with the converted price and qty values

priceQtyConversionStage = {
    $addFields: {
        convertedPrice: { $toDecimal: "$price" },
        convertedQty: { $toInt: "$qty" },
    }
};

// Define stage to calculate total price by multiplying convertedPrice and convertedQty fields
```

```
totalPriceCalculationStage = {
    $project: { item: 1, totalPrice: { $multiply: [ "$convertedPrice", "$convertedQty" ] } }
};

db.orders.aggregate( [
    priceQtyConversionStage,
    totalPriceCalculationStage
])
```

R:

```
{ "_id" : 1, "item" : "apple", "totalPrice" : NumberDecimal("50.0000000000000") }
{ "_id" : 2, "item" : "pie", "totalPrice" : NumberDecimal("200.0") }
{ "_id" : 3, "item" : "ice cream", "totalPrice" : NumberDecimal("9.98") }
{ "_id" : 4, "item" : "almonds", "totalPrice" : NumberDecimal("25.00000000000000") }
```

$toLong

新版本 4.0。

将值转换为 long。如果值不能转换为长错误，则为$toLong 错误。如果值为 null 或缺失，$toLong 返回 null

$toLong 有以下语法:

```
{
    $toLong: <expression>
}
```

$toLong 接受任何有效的表达式。

$toLong 是以下$convert 表达式的缩写:

```
{ $convert: { input: <expression>, to: "long" } }
```

`{ $toLong: true }`	NumberLong(“1”)
`{ $toLong: false }`	NumberLong(“0”)
`{ $toLong: 1.99999 }`	NumberLong(“1”)
`{ $toLong: NumberDecimal("5.5000") }`	NumberLong(“5”)
`{ $toLong: NumberDecimal("9223372036854775808.0") }`	Error
`{ $toLong: NumberInt(8) }`	NumberLong(8)
`{ $toLong: ISODate("2018-03-26T04:38:28.044Z") }`	NumberLong(“1522039108044”)
`{ $toLong: "-2" }`	NumberLong(“-2”)
`{ $toLong: "2.5" }`	Error
`{ $toLong: null }`	null

E:orders collection

```
db.orders.insert( [
    { _id: 1, item: "apple", qty: NumberInt(5) },
```

```
    { _id: 2, item: "pie", qty: "100" },
    { _id: 3, item: "ice cream", qty: NumberLong(500) },
    { _id: 4, item: "almonds", qty: "50" },
] )
```

对 orders 集合执行以下聚合操作，将 qty 转换为 long，然后按值排序:

```
// Define stage to add convertedQty field with converted qty value

qtyConversionStage = {
    $addFields: {
        convertedQty: { $toLong: "$qty" }
    }
};

// Define stage to sort documents by the converted qty values

sortStage = {
    $sort: { "convertedQty": -1 }
};


db.orders.aggregate( [
    qtyConversionStage,
    sortStage
])
```

E:

```
{ "_id" : 1, "item" : "apple", "qty" : 5, "convertedQty" : NumberLong(5) }
{ "_id" : 2, "item" : "pie", "qty" : "100", "convertedQty" : NumberLong(100) }
{ "_id" : 3, "item" : "ice cream", "qty" : NumberLong(500), "convertedQty" : NumberLong(500) }
{ "_id" : 4, "item" : "almonds", "qty" : "50", "convertedQty" : NumberLong(50) }
```

$toObjectId

新版本 4.0。

将值转换为 ObjectId。如果该值不能转换为 ObjectId，则$toObjectId 错误。如果该值为 null 或缺失，$toObjectId 返回 null。

$toObjectId 有以下语法:

```
{
    $toObjectId: <expression>
}
```

$toObjectId 接受任何有效表达式。

$toObjectId 是以下$convert 表达式的简写:

```
{ $convert: { input: <expression>, to: "objectId" } }
```

{$toObjectId: "5ab9cbfa31c2ab715d42129e"}	ObjectId("5ab9cbfa31c2ab715d42129e")
{$toObjectId: "5ab9cbfa31c2ab715d42129"}	Error

E:orders

```
db.orders.insert( [
    { _id: "5ab9cbe531c2ab715d42129a", item: "apple", qty: 10 },
    { _id: ObjectId("5ab9d0b831c2ab715d4212a8"), item: "pie", qty: 5 },
    { _id: ObjectId("5ab9d2d331c2ab715d4212b3"), item: "ice cream", qty: 20 },
    { _id: "5ab9e16431c2ab715d4212b4", item: "almonds", qty: 50 },
] )
```

订单集合上的以下聚合操作在按值排序之前将_id 转换为 ObjectId:

```
// Define stage to add convertedId field with converted _id value

idConversionStage = {
    $addFields: {
        convertedId: { $toObjectId: "$_id" }
    }
};

// Define stage to sort documents by the converted qty values

sortStage = {
    $sort: { "convertedId": -1 }
};
db.orders.aggregate( [
    idConversionStage,
    sortStage
])
```

R:

```
{ "_id" : "5ab9e16431c2ab715d4212b4", "item" : "almonds", "qty" : 50, "convertedId" : ObjectId("5ab9e16431c2ab715d4212b4") }
{ "_id" : ObjectId("5ab9d2d331c2ab715d4212b3"), "item" : "ice cream", "qty" : 20, "convertedId" : ObjectId("5ab9d2d331c2ab715d4212b3") }
{ "_id" : ObjectId("5ab9d0b831c2ab715d4212a8"), "item" : "pie", "qty" : 5, "convertedId" : ObjectId("5ab9d0b831c2ab715d4212a8") }
{ "_id" : "5ab9cbe531c2ab715d42129a", "item" : "apple", "qty" : 10, "convertedId" : ObjectId("5ab9cbe531c2ab715d42129a") }
```

如果转换操作遇到错误，聚合操作将停止并抛出错误。要覆盖此行为，请使用$convert。

$toString

新版本 4.0。

将值转换为字符串。如果该值不能转换为字符串，则为$toString 错误。如果该值为 null 或缺失，$toString 返回 null。

$toString 有以下语法:

```
{
    $toString: <expression>
}
```

$toString 接受任何有效表达式。

$toString 是以下$convert 表达式的缩写:

```
{ $convert: { input: <expression>, to: "string" } }
```

Example	Results
`{$toString: true}`	“true”
`{$toString: false}`	“false”
`{$toString: 2.5}`	“2.5”
`{$toString: NumberInt(2)}`	“2”
`{$toString:  NumberLong(1000)}`	“1000”
`{$toString: ObjectId("5ab9c3da31c2ab715d421285")}`	“5ab9c3da31c2ab715d421285”
`{$toString:  ISODate("2018-03-27T16:58:51.538Z")}`	“2018-03-27T16:58:51.538Z”

orders collection

```
db.orders.insert( [
    { _id: 1, item: "apple",   qty: 5, zipcode: 12345 },
    { _id: 2, item: "pie",   qty: 10, zipcode: 11111 },
    { _id: 3, item: "ice cream",   zipcode: "12345" },
    { _id: 4, item: "almonds", qty: 2, zipcode: "12345-0030" },
])
```

订单集合上的以下聚合操作在按字符串值排序之前将 zipcode 转换为 string:

```
// Define stage to add convertedZipCode field with the converted zipcode value

zipConversionStage = {
    $addFields: {
        convertedZipCode: { $toString: "$zipcode" }
    }
};

// Define stage to sort documents by the converted zipcode

sortStage = {
    $sort: { "convertedZipCode": 1 }
};
```

```
db.orders.aggregate( [
   zipConversionStage,
   sortStage
])
```
E:
```
{ "_id" : 2, "item" : "pie", "qty" : 10, "zipcode" : 11111, "convertedZipCode" : "11111" }
{ "_id" : 1, "item" : "apple", "qty" : 5, "zipcode" : 12345, "convertedZipCode" : "12345" }
{ "_id" : 3, "item" : "ice cream", "zipcode" : "12345", "convertedZipCode" : "12345" }
{ "_id" : 4, "item" : "almonds", "qty" : 2, "zipcode" : "12345-0030", "convertedZipCode" :
"12345-0030" }
```
如果转换操作遇到错误，聚合操作将停止并抛出错误。要覆盖此行为，请使用$convert。

$toLower

将字符串转换为小写，返回结果。
$toLower 有以下语法:
参数可以是任何表达式，只要它解析为字符串。有关表达式的更多信息，请参见表达式。
如果参数解析为 null， $toLower 返回一个空字符串“”。
Behavior
$toLower 只对 ASCII 字符字符串具有定义良好的行为。
E:inventory collection
```
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "PRODUCT 1" }
{ "_id" : 2, "item" : "abc2", quarter: "13Q4", "description" : "Product 2" }
{ "_id" : 3, "item" : "xyz1", quarter: "14Q2", "description" : null }
```
下面的操作使用$toLower 操作符返回小写项和小写描述值:
```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: { $toLower: "$item" },
            description: { $toLower: "$description" }
          }
      }
    ]
)
```
R:
```
{ "_id" : 1, "item" : "abc1", "description" : "product 1" }
{ "_id" : 2, "item" : "abc2", "description" : "product 2" }
{ "_id" : 3, "item" : "xyz1", "description" : "" }
```

$toUpper

美元 toUpper
将字符串转换为大写，返回结果。
$toUpper 的语法如下:
{ $toUpper: <expression> }
参数可以是任何表达式，只要它解析为字符串。有关表达式的更多信息，请参见表达式。
如果参数解析为 null， $toUpper 返回一个空字符串“”。
Behavior
$toUpper 只对 ASCII 字符字符串具有定义良好的行为。
E:inventory collection
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : "PRODUCT 1" }
{ "_id" : 2, "item" : "abc2", quarter: "13Q4", "description" : "Product 2" }
{ "_id" : 3, "item" : "xyz1", quarter: "14Q2", "description" : null }
下面的操作使用$toUpper 操作符返回大写项和大写描述值:

```
db.inventory.aggregate(
    [
      {
        $project:
          {
            item: { $toUpper: "$item" },
            description: { $toUpper: "$description" }
          }
      }
    ]
)
```

R:
{ "_id" : 1, "item" : "ABC1", "description" : "PRODUCT 1" }
{ "_id" : 2, "item" : "ABC2", "description" : "PRODUCT 2" }
{ "_id" : 3, "item" : "XYZ1", "description" : "" }

$trim

新版本 4.0。
删除字符串开头和结尾的空白字符(包括 null)或指定的字符。
$trim 有以下语法:
{ $trim: { input: <string>, chars: <string> } }
$ltrim and $rtrim
Chars:可选的。从输入中删除的字符。
参数可以是任何解析为字符串的有效表达式。$trim 操作符将字符串分解为各个 UTF 代码点，以便从输入中进行修剪。
如果未指定，$trim 将删除空白字符，包括空字符。有关空白字符列表，请参见空白字符。

- By default, `$trim` removes whitespace characters, including the null character:

Example	Results
`{ $trim: { input: "  \n good  bye \t  " } }`	`"good  bye"`

- You can override the default characters to trim using the `chars` field.
  For example, the following trims any **g** and **e** from the start and end of the input. Since the input starts with a whitespace, neither character can be trimmed from the start of the string.

Example	Results
`{ $trim: { input: " ggggoodbyeeeee", chars: "ge" } }`	`" ggggoodby"`

- If overriding the default characters to trim, you can explicitly include the whitespace character(s) to trim in the `chars` field.
  For example, the following trims any space, **g**, **e** from the start and end of the input.

Example	Results
`{ $trim: { input: "    ggggoodbyeeeee", chars: " ge" } }`	`"oodby"`

E inventory collection
{ "_id" : 1, "item" : "ABC1", quarter: "13Q1", "description" : " product 1" }
{ "_id" : 2, "item" : "ABC2", quarter: "13Q4", "description" : "product 2 \n The product is in stock. \n\n " }
{ "_id" : 3, "item" : "XYZ1", quarter: "14Q2", "description" : null }
下面的操作使用$trim 操作符从 description 字段中删除前面和后面的空格:
db.inventory.aggregate([
  { $project: { item: 1, description: { $trim: { input: "$description" } } } }
])
R:
{ "_id" : 1, "item" : "ABC1", "description" : "product 1" }
{ "_id" : 3, "item" : "XYZ1", "description" : null }
{ "_id" : 2, "item" : "ABC2", "description" : "product 2 \n The product is in stock." }

$trunc

新版本 3.2。
将数字截断为整数。
$trunc 的语法如下:

Example	Results
`{ $trunc: 0 }`	`0`
`{ $trunc: 7.80 }`	`7`
`{ $trunc: -2.3 }`	`-2`

E:sample
{ _id: 1, value: 9.25 }
{ _id: 2, value: 8.73 }
{ _id: 3, value: 4.32 }
{ _id: 4, value: -5.34 }
下面的示例返回原始值和截断值:

```
db.samples.aggregate([
    { $project: { value: 1, truncatedValue: { $trunc: "$value" } } }
])
```

R:
{ "_id" : 1, "value" : 9.25, "truncatedValue" : 9 }
{ "_id" : 2, "value" : 8.73, "truncatedValue" : 8 }
{ "_id" : 3, "value" : 4.32, "truncatedValue" : 4 }
{ "_id" : 4, "value" : -5.34, "truncatedValue" : -5 }

$type

新版本 3.4。
返回指定参数的 BSON 类型的字符串。
$type 有以下操作符表达式语法:
{ $type: <expression> }
参数可以是任何有效的表达式。
另请参阅
如果希望按文档的 BSON 类型过滤文档，而不是返回表达式的类型，请使用$type 查询操作符。

Example	Results
`{ $type: "a" }`	`"string"`
`{ $type: /a/ }`	`"regex"`
`{ $type: 1 }`	`"double"`
`{ $type: NumberLong(627) }`	`"long"`
`{ $type: { x: 1 } }`	`"object"`
`{ $type: [ [ 1, 2, 3 ] ] }`	`"array"`

等一系列文字的情况下(1、2、3),附上一组外的数组括号中的表达式防止 MongoDB 解析(1、2、3)与三个参数作为参数列表(1、2、3)。包装数组(1、2、3)文字表达美元达到了相同的效果。
有关更多信息，请参见操作符表达式语法表单。
E:coll collection
{ _id: 0, a : 8 }

{ _id: 1, a : [41.63, 88.19] }
{ _id: 2, a : { a : "apple", b : "banana", c: "carrot" } }
{ _id: 3, a : "caribou" }
{ _id: 4, a : NumberLong(71) }
{ _id: 5 }
下面的聚合操作使用$type 操作符显示作为$project 阶段一部分的所有文档的字段 a 的类型。

```
db.coll.aggregate([{
      $project: {
          a : { $type: "$a" }
      }
}])
```

R:

{ _id: 0, "a" : "double" }
{ _id: 1, "a" : "array" }
{ _id: 2, "a" : "object" }
{ _id: 3, "a" : "string" }
{ _id: 4, "a" : "long" }
{ _id: 5, "a" : "missing" }

$week

以 0 到 53 之间的数字返回日期的一年中的星期。
星期从星期日开始，第一周从一年的第一个星期日开始。一年中第一个星期日之前的日子在第 0 周。此行为与 strftime 标准库函数的“%U”操作符相同。
$week 表达式有以下操作符表达式语法:
{ $week: <dateExpression> }
在 3.6 版中进行了更改。
该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。
{ date: <dateExpression>, timezone: <tzExpression> }
Date: 操作符应用到的日期。<dateExpression>必须是解析为日期、时间戳或 ObjectID 的有效表达式。
Tiemzone: 可选的。操作结果的时区。<tzExpression>必须是一个有效的表达式，它解析为一个格式化为 Olson 时区标识符或 UTC 偏移量的字符串。如果没有提供时区，则以 UTC 显示结果。

Format	Examples
Olson Timezone Identifier	"America/New_York" "Europe/London" "GMT"
UTC Offset	+/-[hh]:[mm], e.g. "+04:45" +/-[hh][mm], e.g. "-0530" +/-[hh], e.g. "+03"

Example	Result
`{ $week: new Date("Jan 1, 2016") }`	0
`{ $week: { date: new Date("2016-01-04") } }`	1
`{ $week: { date: new Date("August 14, 2011"), timezone: "America/Chicago" } }`	33
`{ $week: ISODate("1998-11-01T00:00:00Z") }`	44
`{ $week: { date: ISODate("1998-11-01T00:00:00Z"), timezone: "-0500" } }`	43
`{ $week: "March 28, 1976" }`	error
`{ $week: Date("2016-01-01") }`	error
`{ $week: "2009-04-09" }`	error

$week 不能接受字符串作为参数。

E:sales collection

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$week 和其他日期操作符来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
```

```
            week: { $week: "$date" }

        }
    }
  ]
)
```

R:

```
{
    "_id" : 1,
    "year" : 2014,
    "month" : 1,
    "day" : 1,
    "hour" : 8,
    "minutes" : 15,
    "seconds" : 39,
    "milliseconds" : 736,
    "dayOfYear" : 1,
    "dayOfWeek" : 4,
    "week" : 0
}
```

$year

返回日期的年份部分。
$year 表达式有以下操作符表达式语法:
{ $year: <dateExpression> }
在 3.6 版中进行了更改。

该参数必须是一个有效的表达式，解析为以下之一:
日期、时间戳或 ObjectID。
下列格式的文件:
新版本 3.6。

`{ $year: new Date("2016-01-01") }`	
`{ $year: { date: new Date("Jan 7, 2003") } }`	2003
`{ $year: {` `  date: new Date("August 14, 2011"),` `  timezone: "America/Chicago"` `} }`	2011
`{ $year: ISODate("1998-11-07T00:00:00Z") }`	1998

`{ $year: { date: ISODate("1998-11-07T00:00:00Z"), timezone: "-0400" } }`	1998
`{ $year: "March 28, 1976" }`	error
`{ $year: Date("2016-01-01") }`	error
`{ $year: "2009-04-09" }`	error

$year 不能接受字符串作为参数。

Sales collection

```
{
   "_id" : 1,
   "item" : "abc",
   "price" : 10,
   "quantity" : 2,
   "date" : ISODate("2014-01-01T08:15:39.736Z")
}
```

下面的聚合使用$year 和其他日期操作符来分解日期字段:

```
db.sales.aggregate(
    [
      {
        $project:
          {
            year: { $year: "$date" },
            month: { $month: "$date" },
            day: { $dayOfMonth: "$date" },
            hour: { $hour: "$date" },
            minutes: { $minute: "$date" },
            seconds: { $second: "$date" },
            milliseconds: { $millisecond: "$date" },
            dayOfYear: { $dayOfYear: "$date" },
            dayOfWeek: { $dayOfWeek: "$date" },
            week: { $week: "$date" }
          }
      }
    ]
)
```

R:

```
{
   "_id" : 1,
```

```
    "year" : 2014,
    "month" : 1,
    "day" : 1,
    "hour" : 8,
    "minutes" : 15,
    "seconds" : 39,
    "milliseconds" : 736,
    "dayOfYear" : 1,
    "dayOfWeek" : 4,
    "week" : 0
}
```

$zip

新版本 3.4。

置换一个输入数组的数组，使输出数组的第一个元素是一个包含的数组，第一个输入数组的第一个元素，第二个输入数组的第一个元素，等等。

例如,$ zip 将[[1,2,3],[“a”、“b”、“c”]]到[[1,”一个“],[2,“b”], [3,“c”]]。

$zip 的语法如下:

```
{
     $zip: {
          inputs: [ <array expression1>,   ... ],
          useLongestLength: <boolean>,
          defaults:   <array expression>
     }
}
```

`{ $zip: { inputs: [ [ "a" ], [ "b" ], [ "c" ] ] } }`	`[ [ "a", "b", "c" ] ]`
`{ $zip: { inputs: [ [ "a" ], [ "b", "c" ] ] } }`	`[ [ "a", "b" ] ]`
`{ $zip: { inputs: [ [ 1 ], [ 2, 3 ] ], useLongestLength: true } }`	`[ [ 1, 2 ], [ null, 3 ] ]`
`{ $zip: { inputs: [ [ 1 ], [ 2, 3 ], [ 4 ] ], useLongestLength: true, defaults: [ "a", "b", "c" ] } }`	Because `useLongestLength: true`, `$zip` will pad the shorter input arrays with the corresponding `defaults` elements. This yields `[ [ 1, 2, 4 ], [ "a", 3, "c" ] ]`.

E:
Matrix Transposition
矩阵换位
E:matrices collection:

```
db.matrices.insertMany([
   { matrix: [[1, 2], [2, 3], [3, 4]] },
   { matrix: [[8, 7], [7, 6], [5, 4]] },
])
```

要计算这个集合中每个 3x2 矩阵的转置，可以使用以下聚合操作:

```
db.matrices.aggregate([{
   $project: {
      _id: false,
      transposed: {
         $zip: {
            inputs: [
               { $arrayElemAt: [ "$matrix", 0 ] },
               { $arrayElemAt: [ "$matrix", 1 ] },
               { $arrayElemAt: [ "$matrix", 2 ] },
            ]
         }
      }
   }
}])
```

这将返回以下 2x3 矩阵:

```
{ "transposed" : [ [ 1, 2, 3 ], [ 2, 3, 4 ] ] }
{ "transposed" : [ [ 8, 7, 5 ], [ 7, 6, 4 ] ] }
```

Filtering and Preserving Indexes 过滤和保存索引
您可以使用$zip 和$filter 来获得数组中元素的子集，将原始索引保存在每个保留元素旁边。
名为 pages 的集合包含以下文档:

```
db.pages.save( {
   "category": "unix",
   "pages": [
      { "title": "awk for beginners", reviews: 5 },
      { "title": "sed for newbies", reviews: 0 },
      { "title": "grep made simple", reviews: 2 },
] } )
```

下面的聚合管道将首先压缩页面数组的元素及其索引，然后只过滤至少有一次查看的页面:

```
db.pages.aggregate([{
   $project: {
      _id: false,
      pages: {
         $filter: {
            input: {
               $zip: {
```

```
                inputs: [ "$pages", { $range: [0, { $size: "$pages" }] } ]
              }
            },
            as: "pageWithIndex",
            cond: {
              $let: {
                vars: {
                  page: { $arrayElemAt: [ "$$pageWithIndex", 0 ] }
                },
                in: { $gte: [ "$$page.reviews", 1 ] }
              }
            }
          }
        }
      }
    }])
R:
{
   "pages" : [
     [ { "title" : "awk for beginners", "reviews" : 5 }, 0 ],
     [ { "title" : "grep made simple", "reviews" : 2 }, 2 ] ]
}
```