Model

- Model.insertMany()
- Model.inspect()
- Model.listIndexes()
- Model.mapReduce()
- Model.populate()
- Model.prototype.$where
- Model.prototype.$where()
- Model.prototype.base
- Model.prototype.baseModelName
- Model.prototype.collection
- Model.prototype.db
- Model.prototype.deleteOne()
- Model.prototype.discriminators
- Model.prototype.model()
- Model.prototype.modelName
- Model.prototype.remove()
- Model.prototype.save()
- Model.prototype.schema
- Model.remove()
- Model.replaceOne()
- Model.startSession()
- Model.syncIndexes()
- Model.translateAliases()
- Model.update()
- Model.updateMany()
- Model.updateOne()
- Model.validate()
- Model.watch()
- Model.where()
- increment()

Model()

Parameters

- doc «Object» values for initial set

- optional «[fields]» object containing the fields that were selected in the query which returned this document. You do **not** need to set this parameter to ensure Mongoose handles your query projection.
- [skipId=false] «Boolean» optional boolean. If true, mongoose doesn't add an `_id` field to the document.

Inherits:

- «Document http://mongoosejs.com/docs/api/document.html»

A Model is a class that's your primary tool for interacting with MongoDB. An instance of a Model is called a Document.

In Mongoose, the term "Model" refers to subclasses of the `mongoose.Model` class. You should not use the `mongoose.Model` class directly. The `mongoose.model()` and `connection.model()` functions create subclasses of `mongoose.Model` as shown below.

Example:

```
// `UserModel` is a "Model", a subclass of `mongoose.Model`.
const UserModel = mongoose.model('User', new Schema({ name: String }));

// You can use a Model to create new documents using `new`:
const userDoc = new UserModel({ name: 'Foo' });
await userDoc.save();

// You also use a model to create queries:
const userFromDb = await UserModel.findOne({ name: 'Foo' });
```

Model.aggregate()

Parameters

- [pipeline] «Array» aggregation pipeline as an array of objects
- [options] «Object» aggregation options
- [callback] «Function»

Returns:

- «Aggregate»

Performs aggregations on the models collection.

If a `callback` is passed, the `aggregate` is executed and a `Promise` is returned. If a callback is not passed, the `aggregate` itself is returned.

This function triggers the following middleware.

- `aggregate()`

Example:

```
// Find the max balance of all accounts
const res = await Users.aggregate([
  { $group: { _id: null, maxBalance: { $max: '$balance' }}},
  { $project: { _id: 0, maxBalance: 1 }}
]);

console.log(res); // [ { maxBalance: 98000 } ]

// Or use the aggregation pipeline builder.
const res = await Users.aggregate().
  group({ _id: null, maxBalance: { $max: '$balance' } }).
  project('-id maxBalance').
  exec();
console.log(res); // [ { maxBalance: 98 } ]
```

NOTE:

- Mongoose does **not** cast aggregation pipelines to the model's schema because `$project` and `$group` operators allow redefining the "shape" of the documents at any stage of the pipeline, which may leave documents in an incompatible format. You can use the mongoose-cast-aggregation plugin to enable minimal casting for aggregation pipelines.
- The documents returned are plain javascript objects, not mongoose documents (since any shape of document can be returned).

More About Aggregations:

- Mongoose `Aggregate`
- An Introduction to Mongoose Aggregate
- MongoDB Aggregation docs

Model.buildBulkWriteOperations()

Parameters

- options «Object»
- options.skipValidation «Boolean» defaults to `false`, when set to true, building the write operations will bypass validating the documents.

@param {[Document]} documents The array of documents to build write operations of

---

Model.bulkSave()

Parameters

- documents «[Document]»

takes an array of documents, gets the changes and inserts/updates documents in the database according to whether or not the document is new, or whether it has changes or not.

`bulkSave` uses `bulkWrite` under the hood, so it's mostly useful when dealing with many documents (10K+)

Model.bulkWrite()

Parameters

- ops «Array»
  - [ops.insertOne.document] «Object» The document to insert
  - [opts.updateOne.filter] «Object» Update the first document that matches this filter
  - [opts.updateOne.update] «Object» An object containing update operators
  - [opts.updateOne.upsert=false] «Boolean» If true, insert a doc if none match
  - [opts.updateOne.timestamps=true] «Boolean» If false, do not apply timestamps to the operation
  - [opts.updateOne.collation] «Object» The MongoDB collation to use
  - [opts.updateOne.arrayFilters] «Array» The array filters used in `update`
  - [opts.updateMany.filter] «Object» Update all the documents that match this filter
  - [opts.updateMany.update] «Object» An object containing update operators
  - [opts.updateMany.upsert=false] «Boolean» If true, insert a doc if no documents match `filter`
  - [opts.updateMany.timestamps=true] «Boolean» If false, do not apply timestamps to the operation
  - [opts.updateMany.collation] «Object» The MongoDB collation to use
  - [opts.updateMany.arrayFilters] «Array» The array filters used in `update`
  - [opts.deleteOne.filter] «Object» Delete the first document that matches this filter
  - [opts.deleteMany.filter] «Object» Delete all documents that match this filter
  - [opts.replaceOne.filter] «Object» Replace the first document that matches this filter
  - [opts.replaceOne.replacement] «Object» The replacement document
  - [opts.replaceOne.upsert=false] «Boolean» If true, insert a doc if no documents match `filter`
- [options] «Object»

- [options.ordered=true] «Boolean» If true, execute writes in order and stop at the first error. If false, execute writes in parallel and continue until all writes have either succeeded or errored.
- [options.session=null] «ClientSession» The session associated with this bulk write. See transactions docs.
- [options.w=1] «String|number» The write concern. See `Query#w()` for more information.
- [options.wtimeout=null] «number» The write concern timeout.
- [options.j=true] «Boolean» If false, disable journal acknowledgement
- [options.bypassDocumentValidation=false] «Boolean» If true, disable MongoDB server-side schema validation for all writes in this bulk.
- [options.strict=null] «Boolean» Overwrites the `strict` option on schema. If false, allows filtering and writing fields not defined in the schema for all writes in this bulk.

- [callback] «Function» callback `function(error, bulkWriteOpResult) {}`

Returns:

- «Promise» resolves to a `BulkWriteOpResult` if the operation succeeds

Sends multiple `insertOne`, `updateOne`, `updateMany`, `replaceOne`, `deleteOne`, and/or `deleteMany` operations to the MongoDB server in one command. This is faster than sending multiple independent operations (e.g. if you use `create()`) because with `bulkWrite()` there is only one round trip to MongoDB.

Mongoose will perform casting on all operations you provide.

This function does **not** trigger any middleware, neither `save()`, nor `update()`. If you need to trigger `save()` middleware for every document use `create()` instead.

Example:

```
Character.bulkWrite([
  {
    insertOne: {
      document: {
        name: 'Eddard Stark',
        title: 'Warden of the North'
      }
    }
  },
  {
    updateOne: {
      filter: { name: 'Eddard Stark' },
      // If you were using the MongoDB driver directly, you'd need to do
      // `update: { $set: { title: ... } }` but mongoose adds $set for
      // you.
      update: { title: 'Hand of the King' }
    }
  },
  {
```

```
    deleteOne: {
      {
        filter: { name: 'Eddard Stark' }
      }
    }
  }
]).then(res => {
 // Prints "1 1 1"
 console.log(res.insertedCount, res.modifiedCount, res.deletedCount);
});
```

The supported operations are:

- `insertOne`
- `updateOne`
- `updateMany`
- `deleteOne`
- `deleteMany`
- `replaceOne`

Model.cleanIndexes()

Parameters

- [callback] «Function» optional callback

Returns:

- «Promise,undefined» Returns `undefined` if callback is specified, returns a promise if no callback.

Deletes all indexes that aren't defined in this model's schema. Used by `syncIndexes()`.

The returned promise resolves to a list of the dropped indexes' names as an array

---

Model.count()

Parameters

- filter «Object»
- [callback] «Function»

Returns:

- «Query»

Counts number of documents that match `filter` in a database collection.

This method is deprecated. If you want to count the number of documents in a collection, e.g. `count({})`, use the `estimatedDocumentCount()` function instead. Otherwise, use the `countDocuments()` function instead.

Example:

```
const count = await Adventure.count({ type: 'jungle' });
console.log('there are %d jungle adventures', count);
```

Model.countDocuments()

Parameters

- filter «Object»
- [callback] «Function»

Returns:

- «Query»

Counts number of documents matching `filter` in a database collection.

Example:

```
Adventure.countDocuments({ type: 'jungle' }, function (err, count) {
  console.log('there are %d jungle adventures', count);
});
```

If you want to count all documents in a large collection, use the `estimatedDocumentCount()` function instead. If you call `countDocuments({})`, MongoDB will always execute a full collection scan and **not** use any indexes.

The `countDocuments()` function is similar to `count()`, but there are a few operators that `countDocuments()` does not support. Below are the operators that `count()` supports but `countDocuments()` does not, and the suggested replacement:

- `$where`: `$expr`
- `$near`: `$geoWithin` with `$center`
- `$nearSphere`: `$geoWithin` with `$centerSphere`

Model.create()

Parameters

- docs «Array|Object» Documents to insert, as a spread or array

- [options] «Object» Options passed down to `save()`. To specify `options`, `docs` **must** be an array, not a spread.
- [callback] «Function» callback

Returns:

- «Promise»

Shortcut for saving one or more documents to the database. `MyModel.create(docs)` does `new MyModel(doc).save()` for every doc in docs.

This function triggers the following middleware.

- `save()`

Example:

```
// Insert one new `Character` document
await Character.create({ name: 'Jean-Luc Picard' });

// Insert multiple new `Character` documents
await Character.create([{ name: 'Will Riker' }, { name: 'Geordi LaForge' }]);

// Create a new character within a transaction. Note that you **must**
// pass an array as the first parameter to `create()` if you want to
// specify options.
await Character.create([{ name: 'Jean-Luc Picard' }], { session });
```

Model.createCollection()

Parameters

- [options] «Object» see MongoDB driver docs
- [callback] «Function»

Create the collection for this model. By default, if no indexes are specified, mongoose will not create the collection for the model until any documents are created. Use this method to create the collection explicitly.

Note 1: You may need to call this before starting a transaction See https://docs.mongodb.com/manual/core/transactions/#transactions-and-operations

Note 2: You don't have to call this if your schema contains index or unique field. In that case, just use `Model.init()`

Example:

```
const userSchema = new Schema({ name: String })
const User = mongoose.model('User', userSchema);
```

```
User.createCollection().then(function(collection) {
  console.log('Collection is created!');
});
```

Model.createIndexes()

Parameters

- [options] «Object» internal options
- [cb] «Function» optional callback

Returns:

- «Promise»

Similar to `ensureIndexes()`, except for it uses the `createIndex` function.

Model.deleteMany()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Deletes all of the documents that match `conditions` from the collection. It returns an object with the property `deletedCount` containing the number of documents deleted. Behaves like `remove()`, but deletes all documents that match `conditions` regardless of the `single` option.

Example:

```
await Character.deleteMany({ name: /Stark/, age: { $gte: 18 } }); // returns {deletedCount: x} whe
```

Note:

This function triggers `deleteMany` query hooks. Read the middleware docs to learn more.

Model.deleteOne()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Deletes the first document that matches `conditions` from the collection. It returns an object with the property `deletedCount` indicating how many documents were deleted. Behaves like `remove()`, but deletes at most one document regardless of the `single` option.

Example:

```
await Character.deleteOne({ name: 'Eddard Stark' }); // returns {deletedCount: 1}
```

Note:

This function triggers `deleteOne` query hooks. Read the middleware docs to learn more.

Model.diffIndexes()

Parameters

- options «Object» not used at all.
- callback «Function» optional callback

Does a dry-run of Model.syncIndexes(), meaning that the result of this function would be the result of Model.syncIndexes().

Model.discriminator()

Parameters

- name «String» discriminator model name
- schema «Schema» discriminator model schema
- [options] «Object|String» If string, same as `options.value`.
    - [options.value] «String» the string stored in the `discriminatorKey` property. If not specified, Mongoose uses the `name` parameter.

- [options.clone=true] «Boolean» By default, `discriminator()` clones the given `schema`. Set to `false` to skip cloning.

Returns:

- «Model» The newly created discriminator model

Adds a discriminator type.

Example:

```
function BaseSchema() {
  Schema.apply(this, arguments);

  this.add({
    name: String,
    createdAt: Date
  });
}
util.inherits(BaseSchema, Schema);

const PersonSchema = new BaseSchema();
const BossSchema = new BaseSchema({ department: String });

const Person = mongoose.model('Person', PersonSchema);
const Boss = Person.discriminator('Boss', BossSchema);
new Boss().__t; // "Boss". `__t` is the default `discriminatorKey`

const employeeSchema = new Schema({ boss: ObjectId });
const Employee = Person.discriminator('Employee', employeeSchema, 'staff');
new Employee().__t; // "staff" because of 3rd argument above
```

Model.distinct()

Parameters

- field «String»
- [conditions] «Object» optional
- [callback] «Function»

Returns:

- «Query»

Creates a Query for a `distinct` operation.

Passing a `callback` executes the query.

Example

```
Link.distinct('url', { clicks: {$gt: 100}}, function (err, result) {
  if (err) return handleError(err);

  assert(Array.isArray(result));
  console.log('unique urls with more than 100 clicks', result);
})

const query = Link.distinct('url');
query.exec(callback);
```

Model.ensureIndexes()

Parameters

- [options] «Object» internal options
- [cb] «Function» optional callback

Returns:

- «Promise»

Sends `createIndex` commands to mongo for each index declared in the schema. The `createIndex` commands are sent in series.

Example:

```
Event.ensureIndexes(function (err) {
  if (err) return handleError(err);
});
```

After completion, an `index` event is emitted on this `Model` passing an error if one occurred.

Example:

```
const eventSchema = new Schema({ thing: { type: 'string', unique: true }})
const Event = mongoose.model('Event', eventSchema);

Event.on('index', function (err) {
  if (err) console.error(err); // error occurred during index creation
})
```

NOTE: It is not recommended that you run this in production. Index creation may impact database performance depending on your load. Use with caution.

Model.estimatedDocumentCount()

Parameters

- [options] «Object»
- [callback] «Function»

Returns:

- «Query»

Estimates the number of documents in the MongoDB collection. Faster than using `countDocuments()` for large collections because `estimatedDocumentCount()` uses collection metadata rather than scanning the entire collection.

Example:

```
const numAdventures = Adventure.estimatedDocumentCount();
```

Model.events

Type:

- «property»

Event emitter that reports any errors that occurred. Useful for global error handling.

Example:

```
MyModel.events.on('error', err => console.log(err.message));

// Prints a 'CastError' because of the above handler
await MyModel.findOne({ _id: 'notanid' }).catch(noop);
```

Model.exists()

Parameters

- filter «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function» callback

Returns:

- «Query»

Returns true if at least one document exists in the database that matches the given `filter`, and false otherwise.

Under the hood, `MyModel.exists({ answer: 42 })` is equivalent to `MyModel.findOne({ answer: 42 }).select({ _id: 1 }).lean()`

Example:

```
await Character.deleteMany({});
await Character.create({ name: 'Jean-Luc Picard' });

await Character.exists({ name: /picard/i }); // { _id: ... }
await Character.exists({ name: /riker/i }); // null
```

This function triggers the following middleware.

- `findOne()`

Model.find()

Parameters

- filter «Object|ObjectId»
- [projection] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Finds documents.

Mongoose casts the `filter` to match the model's schema before the command is sent. See our query casting tutorial for more information on how Mongoose casts `filter`.

Examples:

```
// find all documents
await MyModel.find({});

// find all documents named john and at least 18
await MyModel.find({ name: 'john', age: { $gte: 18 } }).exec();

// executes, passing results to callback
MyModel.find({ name: 'john', age: { $gte: 18 }}, function (err, docs) {});

// executes, name LIKE john and only selecting the "name" and "friends" fields
await MyModel.find({ name: /john/i }, 'name friends').exec();
```

```
// passing options
await MyModel.find({ name: /john/i }, null, { skip: 10 }).exec();
```

Model.findById()

Parameters

- id «Any» value of `_id` to query by
- [projection] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Finds a single document by its _id field. `findById(id)` is almost* equivalent to `findOne({ _id: id })`. If you want to query by a document's `_id`, use `findById()` instead of `findOne()`.

The `id` is cast based on the Schema before sending the command.

This function triggers the following middleware.

- `findOne()`

* Except for how it treats `undefined`. If you use `findOne()`, you'll see that `findOne(undefined)` and `findOne({ _id: undefined })` are equivalent to `findOne({})` and return arbitrary documents. However, mongoose translates `findById(undefined)` into `findOne({ _id: null })`.

Example:

```
// Find the adventure with the given `id`, or `null` if not found
await Adventure.findById(id).exec();

// using callback
Adventure.findById(id, function (err, adventure) {});

// select only the adventures name and length
await Adventure.findById(id, 'name length').exec();
```

Model.findByIdAndDelete()

Parameters

- id «Object|Number|String» value of `_id` to query by

- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option
- [callback] «Function»

Returns:

- «Query»

Issue a MongoDB `findOneAndDelete()` command by a document's _id field. In other words, `findByIdAndDelete(id)` is a shorthand for `findOneAndDelete({ _id: id })`.

This function triggers the following middleware.

- `findOneAndDelete()`

Model.findByIdAndRemove()

Parameters

- id «Object|Number|String» value of `_id` to query by
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option
    - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
    - [options.projection=null] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [callback] «Function»

Returns:

- «Query»

Issue a mongodb findAndModify remove command by a document's _id field. `findByIdAndRemove(id, ...)` is equivalent to `findOneAndRemove({ _id: id }, ...)`.

Finds a matching document, removes it, passing the found document (if any) to the callback.

Executes the query if `callback` is passed.

This function triggers the following middleware.

- `findOneAndRemove()`

Options:

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `select`: sets the document fields to return

- `rawResult`: if true, returns the raw result from the MongoDB driver
- `strict`: overwrites the schema's strict mode option for this update

Examples:

```
A.findByIdAndRemove(id, options, callback) // executes
A.findByIdAndRemove(id, options)  // return Query
A.findByIdAndRemove(id, callback) // executes
A.findByIdAndRemove(id) // returns Query
A.findByIdAndRemove()           // returns Query
```

Model.findByIdAndUpdate()

Parameters

- id «Object|Number|String» value of `_id` to query by
- [update] «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.returnDocument='before'] «String» Has two possible values, `'before'` and `'after'`. By default, it will return the document before the update was applied.
    - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()` and the Mongoose lean tutorial.
    - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option
    - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.
    - [options.overwrite=false] «Boolean» By default, if you don't include any update operators in `update`, Mongoose will wrap `update` in `$set` for you. This prevents you from accidentally overwriting the document. This option tells Mongoose to skip adding `$set`. An alternative to this would be using Model.findOneAndReplace({ _id: id }, update, options, callback).
- [callback] «Function»

Returns:

- «Query»

Issues a mongodb findAndModify update command by a document's _id field. `findByIdAndUpdate(id, ...)` is equivalent to `findOneAndUpdate({ _id: id }, ...)`.

Finds a matching document, updates it according to the `update` arg, passing any `options`, and returns the found document (if any) to the callback. The query executes if `callback` is passed.

This function triggers the following middleware.

- `findOneAndUpdate()`

Options:

- `new`: bool - true to return the modified document rather than the original. defaults to false
- `upsert`: bool - creates the object if it doesn't exist. defaults to false.
- `runValidators`: if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert`: `true` by default. If `setDefaultsOnInsert` and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created.
- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `select`: sets the document fields to return
- `rawResult`: if true, returns the raw result from the MongoDB driver
- `strict`: overwrites the schema's strict mode option for this update

Examples:

```
A.findByIdAndUpdate(id, update, options, callback) // executes
A.findByIdAndUpdate(id, update, options)  // returns Query
A.findByIdAndUpdate(id, update, callback) // executes
A.findByIdAndUpdate(id, update)           // returns Query
A.findByIdAndUpdate()                     // returns Query
```

Note:

All top level update keys which are not `atomic` operation names are treated as set operations:

Example:

```
Model.findByIdAndUpdate(id, { name: 'jason bourne' }, options, callback)

// is sent as
Model.findByIdAndUpdate(id, { $set: { name: 'jason bourne' }}, options, callback)
```

This helps prevent accidentally overwriting your document with `{ name: 'jason bourne' }`.

Note:

`findOneAndX` and `findByIdAndX` functions support limited validation. You can enable validation by setting the `runValidators` option.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.findOne()

Parameters

- [conditions] «Object»
- [projection] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [options] «Object» optional see `Query.prototype.setOptions()`
- [callback] «Function»

Returns:

- «Query»

Finds one document.

The `conditions` are cast to their respective SchemaTypes before the command is sent.

Note: `conditions` is optional, and if `conditions` is null or undefined, mongoose will send an empty `findOne` command to MongoDB, which will return an arbitrary document. If you're querying by `_id`, use `findById()` instead.

Example:

```
// Find one adventure whose `country` is 'Croatia', otherwise `null`
await Adventure.findOne({ country: 'Croatia' }).exec();

// using callback
Adventure.findOne({ country: 'Croatia' }, function (err, adventure) {});

// select only the adventures name and length
await Adventure.findOne({ country: 'Croatia' }, 'name length').exec();
```

Model.findOneAndDelete()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`

- [options.strict] «Boolean|String» overwrites the schema's strict mode option
- [options.projection=null] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [options.session=null] «ClientSession» The session associated with this query. See transactions docs.

- [callback] «Function»

Returns:

- «Query»

Issue a MongoDB `findOneAndDelete()` command.

Finds a matching document, removes it, and passes the found document (if any) to the callback.

Executes the query if `callback` is passed.

This function triggers the following middleware.

- `findOneAndDelete()`

This function differs slightly from `Model.findOneAndRemove()` in that `findOneAndRemove()` becomes a MongoDB `findAndModify()` command, as opposed to a `findOneAndDelete()` command. For most mongoose use cases, this distinction is purely pedantic. You should use `findOneAndDelete()` unless you have a good reason not to.

Options:

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `select`: sets the document fields to return, ex. `{ projection: { _id: 0 } }`
- `projection`: equivalent to `select`
- `rawResult`: if true, returns the raw result from the MongoDB driver
- `strict`: overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndDelete(conditions, options, callback) // executes
A.findOneAndDelete(conditions, options)  // return Query
A.findOneAndDelete(conditions, callback) // executes
A.findOneAndDelete(conditions) // returns Query
A.findOneAndDelete()           // returns Query
```

`findOneAndX` and `findByIdAndX` functions support limited validation. You can enable validation by setting the `runValidators` option.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.findOneAndRemove()

Parameters

- conditions «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option
    - [options.projection=null] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [callback] «Function»

Returns:

- «Query»

Issue a mongodb findAndModify remove command.

Finds a matching document, removes it, passing the found document (if any) to the callback.

Executes the query if `callback` is passed.

This function triggers the following middleware.

- `findOneAndRemove()`

Options:

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `select`: sets the document fields to return
- `projection`: like select, it determines which fields to return, ex. `{ projection: { _id: 0 } }`
- `rawResult`: if true, returns the raw result from the MongoDB driver
- `strict`: overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndRemove(conditions, options, callback) // executes
A.findOneAndRemove(conditions, options)  // return Query
A.findOneAndRemove(conditions, callback) // executes
A.findOneAndRemove(conditions) // returns Query
A.findOneAndRemove()           // returns Query
```

`findOneAndX` and `findByIdAndX` functions support limited validation. You can enable validation by setting the `runValidators` option.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
Model.findById(id, function (err, doc) {
  if (err) ..
  doc.name = 'jason bourne';
  doc.save(callback);
});
```

Model.findOneAndReplace()

Parameters

- filter «Object» Replace the first document that matches this filter
- [replacement] «Object» Replace with this document
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.returnDocument='before'] «String» Has two possible values, `'before'` and `'after'`. By default, it will return the document before the update was applied.
    - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()` and the Mongoose lean tutorial.
    - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option
    - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.
    - [options.projection=null] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`
- [callback] «Function»

Returns:

- «Query»

Issue a MongoDB `findOneAndReplace()` command.

Finds a matching document, replaces it with the provided doc, and passes the returned doc to the callback.

Executes the query if `callback` is passed.

This function triggers the following query middleware.

- `findOneAndReplace()`

Options:

- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `select`: sets the document fields to return
- `projection`: like select, it determines which fields to return, ex. `{ projection: { _id: 0 } }`
- `rawResult`: if true, returns the raw result from the MongoDB driver
- `strict`: overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndReplace(conditions, options, callback) // executes
A.findOneAndReplace(conditions, options)  // return Query
A.findOneAndReplace(conditions, callback) // executes
A.findOneAndReplace(conditions) // returns Query
A.findOneAndReplace()           // returns Query
```

Model.findOneAndUpdate()

Parameters

- [conditions] «Object»
- [update] «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.returnDocument='before'] «String» Has two possible values, `'before'` and `'after'`. By default, it will return the document before the update was applied.
    - [options.lean] «Object» if truthy, mongoose will return the document as a plain JavaScript object rather than a mongoose document. See `Query.lean()` and the Mongoose lean tutorial.
    - [options.session=null] «ClientSession» The session associated with this query. See transactions docs.
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option

- [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.
- [options.overwrite=false] «Boolean» By default, if you don't include any update operators in `update`, Mongoose will wrap `update` in `$set` for you. This prevents you from accidentally overwriting the document. This option tells Mongoose to skip adding `$set`. An alternative to this would be using Model.findOneAndReplace(conditions, update, options, callback).
- [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
- [options.projection=null] «Object|String|Array<String>» optional fields to return, see `Query.prototype.select()`

- [callback] «Function»

Returns:

- «Query»

Issues a mongodb findAndModify update command.

Finds a matching document, updates it according to the `update` arg, passing any `options`, and returns the found document (if any) to the callback. The query executes if `callback` is passed else a Query object is returned.

Options:

- `new`: bool - if true, return the modified document rather than the original. defaults to false (changed in 4.0)
- `upsert`: bool - creates the object if it doesn't exist. defaults to false.
- `overwrite`: bool - if true, replace the entire document.
- `fields`: {Object|String} - Field selection. Equivalent to `.select(fields).findOneAndUpdate()`
- `maxTimeMS`: puts a time limit on the query - requires mongodb >= 2.6.0
- `sort`: if multiple docs are found by the conditions, sets the sort order to choose which doc to update
- `runValidators`: if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert`: `true` by default. If `setDefaultsOnInsert` and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created.
- `rawResult`: if true, returns the raw result from the MongoDB driver
- `strict`: overwrites the schema's strict mode option for this update

Examples:

```
A.findOneAndUpdate(conditions, update, options, callback) // executes
A.findOneAndUpdate(conditions, update, options)  // returns Query
A.findOneAndUpdate(conditions, update, callback) // executes
A.findOneAndUpdate(conditions, update)           // returns Query
A.findOneAndUpdate()                             // returns Query
```

Note:

All top level update keys which are not `atomic` operation names are treated as set operations:

Example:

```
const query = { name: 'borne' };
Model.findOneAndUpdate(query, { name: 'jason bourne' }, options, callback)

// is sent as
Model.findOneAndUpdate(query, { $set: { name: 'jason bourne' }}, options, callback)
```

This helps prevent accidentally overwriting your document with `{ name: 'jason bourne' }`.

Note:

`findOneAndX` and `findByIdAndX` functions support limited validation that you can enable by setting the `runValidators` option.

If you need full-fledged validation, use the traditional approach of first retrieving the document.

```
const doc = await Model.findById(id);
doc.name = 'jason bourne';
await doc.save();
```

Model.hydrate()

Parameters

- obj «Object»
- [projection] «Object|String|Array<String>» optional projection containing which fields should be selected for this document

Returns:

- «Document» document instance

Shortcut for creating a new Document from existing raw data, pre-saved in the DB. The document returned has no paths marked as modified initially.

Example:

```
// hydrate previous data into a Mongoose document
const mongooseCandy = Candy.hydrate({ _id: '54108337212ffb6d459f854c', type: 'jelly bean' });
```

Model.init()

Parameters

- [callback] «Function»

This function is responsible for building indexes, unless `autoIndex` is turned off.

Mongoose calls this function automatically when a model is created using `mongoose.model()` or `connection.model()`, so you don't need to call it. This function is also idempotent, so you may call it to get back a promise that will resolve when your indexes are finished building as an alternative to `MyModel.on('index')`

Example:

```
const eventSchema = new Schema({ thing: { type: 'string', unique: true }})
// This calls `Event.init()` implicitly, so you don't need to call
// `Event.init()` on your own.
const Event = mongoose.model('Event', eventSchema);

Event.init().then(function(Event) {
  // You can also use `Event.on('index')` if you prefer event emitters
  // over promises.
  console.log('Indexes are done building!');
});
```

Model.insertMany()

Parameters

- doc(s) «Array|Object|*»
- [options] «Object» see the mongodb driver options
- [options.ordered «Boolean» = true] if true, will fail fast on the first error encountered. If false, will insert all the documents it can and report errors later. An `insertMany()` with `ordered = false` is called an "unordered" `insertMany()`.
- [options.rawResult «Boolean» = false] if false, the returned promise resolves to the documents that passed mongoose document validation. If `true`, will return the raw result from the MongoDB driver with a `mongoose` property that contains `validationErrors` if this is an unordered `insertMany`.
- [options.lean «Boolean» = false] if `true`, skips hydrating and validating the documents. This option is useful if you need the extra performance, but Mongoose won't validate the documents before inserting.
- [options.limit «Number» = null] this limits the number of documents being processed (validation/casting) by mongoose in parallel, this does **NOT** send the documents in batches to MongoDB. Use this option if you're processing a large number of documents and your app is running out of memory.

- [options.populate «String|Object|Array» = null] populates the result documents. This option is a no-op if `rawResult` is set.
- [callback] «Function» callback

Returns:

- «Promise» resolving to the raw result from the MongoDB driver if `options.rawResult` was `true`, or the documents that passed validation, otherwise

Shortcut for validating an array of documents and inserting them into MongoDB if they're all valid. This function is faster than `.create()` because it only sends one operation to the server, rather than one for each document.

Mongoose always validates each document **before** sending `insertMany` to MongoDB. So if one document has a validation error, no documents will be saved, unless you set the `ordered` option to false.

This function does **not** trigger save middleware.

This function triggers the following middleware.

- `insertMany()`

Example:

```
const arr = [{ name: 'Star Wars' }, { name: 'The Empire Strikes Back' }];
Movies.insertMany(arr, function(error, docs) {});
```

Model.inspect()

Helper for console.log. Given a model named 'MyModel', returns the string `'Model { MyModel }'`.

Example:

```
const MyModel = mongoose.model('Test', Schema({ name: String }));
MyModel.inspect(); // 'Model { Test }'
console.log(MyModel); // Prints 'Model { Test }'
```

Model.listIndexes()

Parameters

- [cb] «Function» optional callback

Returns:

- «Promise,undefined» Returns `undefined` if callback is specified, returns a promise if no callback.

Lists the indexes currently defined in MongoDB. This may or may not be the same as the indexes defined in your schema depending on whether you use the `autoIndex` option and if you build indexes manually.

Model.mapReduce()

Parameters

- o «Object» an object specifying map-reduce options
- [callback] «Function» optional callback

Returns:

- «Promise»

Executes a mapReduce command.

`o` is an object specifying all mapReduce options as well as the map and reduce functions. All options are delegated to the driver implementation. See node-mongodb-native mapReduce() documentation for more detail about options.

This function does not trigger any middleware.

Example:

```
const o = {};
// `map()` and `reduce()` are run on the MongoDB server, not Node.js,
// these functions are converted to strings
o.map = function () { emit(this.name, 1) };
o.reduce = function (k, vals) { return vals.length };
User.mapReduce(o, function (err, results) {
  console.log(results)
})
```

Other options:

- `query` {Object} query filter object.
- `sort` {Object} sort input objects using this key
- `limit` {Number} max number of documents
- `keeptemp` {Boolean, default:false} keep temporary data
- `finalize` {Function} finalize function
- `scope` {Object} scope variables exposed to map/reduce/finalize during execution
- `jsMode` {Boolean, default:false} it is possible to make the execution stay in JS. Provided in MongoDB > 2.0.X
- `verbose` {Boolean, default:false} provide statistics on job execution time.
- `readPreference` {String}

- `out*` {Object, default: {inline:1}} sets the output target for the map reduce job.

*** out options:**

- `{inline:1}` the results are returned in an array
- `{replace: 'collectionName'}` add the results to collectionName: the results replace the collection
- `{reduce: 'collectionName'}` add the results to collectionName: if dups are detected, uses the reducer / finalize functions
- `{merge: 'collectionName'}` add the results to collectionName: if dups exist the new docs overwrite the old

If `options.out` is set to `replace`, `merge`, or `reduce`, a Model instance is returned that can be used for further querying. Queries run against this model are all executed with the `lean` option; meaning only the js object is returned and no Mongoose magic is applied (getters, setters, etc).

Example:

```
const o = {};
// You can also define `map()` and `reduce()` as strings if your
// linter complains about `emit()` not being defined
o.map = 'function () { emit(this.name, 1) }';
o.reduce = 'function (k, vals) { return vals.length }';
o.out = { replace: 'createdCollectionNameForResults' }
o.verbose = true;

User.mapReduce(o, function (err, model, stats) {
  console.log('map reduce took %d ms', stats.processtime)
  model.find().where('value').gt(10).exec(function (err, docs) {
    console.log(docs);
  });
})

// `mapReduce()` returns a promise. However, ES6 promises can only
// resolve to exactly one value,
o.resolveToObject = true;
const promise = User.mapReduce(o);
promise.then(function (res) {
  const model = res.model;
  const stats = res.stats;
  console.log('map reduce took %d ms', stats.processtime)
  return model.find().where('value').gt(10).exec();
}).then(function (docs) {
   console.log(docs);
}).then(null, handleError).end()
```

Model.populate()

Parameters

- docs «Document|Array» Either a single document or array of documents to populate.
- options «Object|String» Either the paths to populate or an object specifying all parameters
  - [options.path=null] «string» The path to populate.
  - [options.retainNullValues=false] «boolean» By default, Mongoose removes null and undefined values from populated arrays. Use this option to make `populate()` retain `null` and `undefined` array entries.
  - [options.getters=false] «boolean» If true, Mongoose will call any getters defined on the `localField`. By default, Mongoose gets the raw value of `localField`. For example, you would need to set this option to `true` if you wanted to add a `lowercase` getter to your `localField`.
  - [options.clone=false] «boolean» When you do `BlogPost.find().populate('author')`, blog posts with the same author will share 1 copy of an `author` doc. Enable this option to make Mongoose clone populated docs before assigning them.
  - [options.match=null] «Object|Function» Add an additional filter to the populate query. Can be a filter object containing MongoDB query syntax, or a function that returns a filter object.
  - [options.skipInvalidIds=false] «Boolean» By default, Mongoose throws a cast error if `localField` and `foreignField` schemas don't line up. If you enable this option, Mongoose will instead filter out any `localField` properties that cannot be casted to `foreignField`'s schema type.
  - [options.perDocumentLimit=null] «Number» For legacy reasons, `limit` with `populate()` may give incorrect results because it only executes a single query for every document being populated. If you set `perDocumentLimit`, Mongoose will ensure correct `limit` per document by executing a separate query for each document to `populate()`. For example, `.find().populate({ path: 'test', perDocumentLimit: 2 })` will execute 2 additional queries if `.find()` returns 2 documents.
  - [options.strictPopulate=true] «Boolean» Set to false to allow populating paths that aren't defined in the given model's schema.
  - [options.options=null] «Object» Additional options like `limit` and `lean`.
  - [options.transform=null] «Function» Function that Mongoose will call on every populated document that allows you to transform the populated document.
- [callback(err,doc)] «Function» Optional callback, executed upon completion. Receives `err` and the `doc(s)`.

Returns:

- «Promise»

Populates document references.

Changed in Mongoose 6: the model you call `populate()` on should be the "local field" model, **not** the "foreign field" model.

Available top-level options:

- path: space delimited path(s) to populate
- select: optional fields to select
- match: optional query conditions to match
- model: optional name of the model to use for population
- options: optional query options like sort, limit, etc
- justOne: optional boolean, if true Mongoose will always set `path` to an array. Inferred from schema by default.
- strictPopulate: optional boolean, set to `false` to allow populating paths that aren't in the schema.

Examples:

```
const Dog = mongoose.model('Dog', new Schema({ name: String, breed: String }));
const Person = mongoose.model('Person', new Schema({
  name: String,
  pet: { type: mongoose.ObjectId, ref: 'Dog' }
}));

const pets = await Pet.create([
  { name: 'Daisy', breed: 'Beagle' },
  { name: 'Einstein', breed: 'Catalan Sheepdog' }
]);

// populate many plain objects
const users = [
  { name: 'John Wick', dog: pets[0]._id },
  { name: 'Doc Brown', dog: pets[1]._id }
];
await User.populate(users, { path: 'dog', select: 'name' });
users[0].dog.name; // 'Daisy'
users[0].dog.breed; // undefined because of `select`
```

Model.prototype.$where

Type:

- «property»

Additional properties to attach to the query when calling `save()` and `isNew` is false.

Model.prototype.$where()

Parameters

- argument «String|Function» is a javascript string or anonymous function

Returns:

- «Query»

Creates a `Query` and specifies a `$where` condition.

Sometimes you need to query for things in mongodb using a JavaScript expression. You can do so via `find({ $where: javascript })`, or you can use the mongoose shortcut method $where via a Query chain or from your mongoose Model.

```
Blog.$where('this.username.indexOf("val") !== -1').exec(function (err, docs) {});
```

Model.prototype.base

Type:

- «property»

Base Mongoose instance the model uses.

Model.prototype.baseModelName

Type:

- «property»

If this is a discriminator model, `baseModelName` is the name of the base model.

Model.prototype.collection

Type:

- «property»

Collection the model uses.

This property is read-only. Modifying this property is a no-op.

Model.prototype.db

Type:

- «property»

Connection the model uses.

Model.prototype.deleteOne()

Parameters

- [fn] «function(err|product)» optional callback

Returns:

- «Promise» Promise

Removes this document from the db. Equivalent to `.remove()`.

Example:

```
product = await product.deleteOne();
await Product.findById(product._id); // null
```

Model.prototype.discriminators

Type:

- «property»

Registered discriminators for this model.

Model.prototype.model()

Parameters

- name «String» model name

Returns another Model instance.

Example:

```
const doc = new Tank;
doc.model('User').findById(id, callback);
```

Model.prototype.modelName

Type:

- «property»

The name of the model

Model.prototype.remove()

Parameters

- [options] «Object»
  - [options.session=null] «Session» the session associated with this operation. If not specified, defaults to the document's associated session.
- [fn] «function(err|product)» optional callback

Returns:

- «Promise» Promise

Removes this document from the db.

Example:

```
product.remove(function (err, product) {
  if (err) return handleError(err);
  Product.findById(product._id, function (err, product) {
    console.log(product) // null
  })
})
```

As an extra measure of flow control, remove will return a Promise (bound to `fn` if passed) so it could be chained, or hooked to receive errors

Example:

```
product.remove().then(function (product) {
   ...
}).catch(function (err) {
   assert.ok(err)
})
```

Model.prototype.save()

Parameters

- [options] «Object» options optional options
  - [options.session=null] «Session» the session associated with this save operation. If not specified, defaults to the document's associated session.

- [options.safe] «Object» (DEPRECATED) overrides schema's safe option. Use the `w` option instead.
- [options.validateBeforeSave] «Boolean» set to false to save without validating.
- [options.validateModifiedOnly=false] «Boolean» if `true`, Mongoose will only validate modified paths, as opposed to modified paths and `required` paths.
- [options.w] «Number|String» set the write concern. Overrides the schema-level `writeConcern` option
- [options.j] «Boolean» set to true for MongoDB to wait until this `save()` has been journaled before resolving the returned promise. Overrides the schema-level `writeConcern` option
- [options.wtimeout] «Number» sets a timeout for the write concern. Overrides the schema-level `writeConcern` option.
- [options.checkKeys=true] «Boolean» the MongoDB driver prevents you from saving keys that start with '$' or contain '.' by default. Set this option to `false` to skip that check. See restrictions on field names
- [options.timestamps=true] «Boolean» if `false` and timestamps are enabled, skip timestamps for this `save()`.

- [fn] «Function» optional callback

Returns:

- «Promise,undefined» Returns undefined if used with callback or a Promise otherwise.

Saves this document by inserting a new document into the database if document.isNew is `true`, or sends an updateOne operation with just the modified paths if `isNew` is `false`.

Example:

```
product.sold = Date.now();
product = await product.save();
```

If save is successful, the returned promise will fulfill with the document saved.

Example:

```
const newProduct = await product.save();
newProduct === product; // true
```

Model.prototype.schema

Type:

- «property»

Schema the model uses.

Model.remove()

Parameters

- conditions «Object»
- [options] «Object»
  - [options.session=null] «Session» the session associated with this operation.
- [callback] «Function»

Returns:

- «Query»

Removes all documents that match `conditions` from the collection. To remove just the first document that matches `conditions`, set the `single` option to true.

This method is deprecated. See Deprecation Warnings for details.

Example:

```
const res = await Character.remove({ name: 'Eddard Stark' });
res.deletedCount; // Number of documents removed
```

Note:

This method sends a remove command directly to MongoDB, no Mongoose documents are involved. Because no Mongoose documents are involved, Mongoose does not execute document middleware.

Model.replaceOne()

Parameters

- filter «Object»
- doc «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern

    - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.

- [callback] «Function» `function(error, res) {}` where `res` has 3 properties: `n`, `nModified`, `ok`.

Returns:

- «Query»

Same as `update()`, except MongoDB replace the existing document with the given document (no atomic operators like `$set`).

Example:

```
const res = await Person.replaceOne({ _id: 24601 }, { name: 'Jean Valjean' });
res.matchedCount; // Number of documents matched
res.modifiedCount; // Number of documents modified
res.acknowledged; // Boolean indicating everything went smoothly.
res.upsertedId; // null or an id containing a document that had to be upserted.
res.upsertedCount; // Number indicating how many documents had to be upserted. Will either be 0 or
```

This function triggers the following middleware.

- `replaceOne()`

Model.startSession()

Parameters

- [options] «Object» see the mongodb driver options
    - [options.causalConsistency=true] «Boolean» set to false to disable causal consistency
- [callback] «Function»

Returns:

- «Promise<ClientSession>» promise that resolves to a MongoDB driver `ClientSession`

Requires MongoDB >= 3.6.0. Starts a MongoDB session for benefits like causal consistency, retryable writes, and transactions.

Calling `MyModel.startSession()` is equivalent to calling `MyModel.db.startSession()`.

This function does not trigger any middleware.

Example:

```
const session = await Person.startSession();
let doc = await Person.findOne({ name: 'Ned Stark' }, null, { session });
```

```
await doc.remove();
// `doc` will always be null, even if reading from a replica set
// secondary. Without causal consistency, it is possible to
// get a doc back from the below query if the query reads from a
// secondary that is experiencing replication lag.
doc = await Person.findOne({ name: 'Ned Stark' }, null, { session, readPreference: 'secondary' });
```

Model.syncIndexes()

Parameters

- [options] «Object» options to pass to `ensureIndexes()`
  - [options.background=null] «Boolean» if specified, overrides each index's `background` property
- [callback] «Function» optional callback

Returns:

- «Promise,undefined» Returns `undefined` if callback is specified, returns a promise if no callback.

Makes the indexes in MongoDB match the indexes defined in this model's schema. This function will drop any indexes that are not defined in the model's schema except the `_id` index, and build any indexes that are in your schema but not in MongoDB.

See the introductory blog post for more information.

Example:

```
const schema = new Schema({ name: { type: String, unique: true } });
const Customer = mongoose.model('Customer', schema);
await Customer.collection.createIndex({ age: 1 }); // Index is not in schema
// Will drop the 'age' index and create an index on `name`
await Customer.syncIndexes();
```

Model.translateAliases()

Parameters

- raw «Object» fields/conditions that may contain aliased keys

Returns:

- «Object» the translated 'pure' fields/conditions

Translate any aliases fields/conditions so the final query or document object is pure

Example:

```
Character
  .find(Character.translateAliases({
    '名': 'Eddard Stark' // Alias for 'name'
  })
  .exec(function(err, characters) {})
```

Note:

Only translate arguments of object type anything else is returned raw

---

Model.update()

Parameters

- filter «Object»
- doc «Object»
- [options] «Object» optional see `Query.prototype.setOptions()`
    - [options.strict] «Boolean|String» overwrites the schema's strict mode option
    - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
    - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
    - [options.multi=false] «Boolean» whether multiple documents should be updated or just the first one that matches `filter`.
    - [options.runValidators=false] «Boolean» if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
    - [options.setDefaultsOnInsert=false] «Boolean» `true` by default. If `setDefaultsOnInsert` and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created.
    - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.
    - [options.overwrite=false] «Boolean» By default, if you don't include any update operators in `doc`, Mongoose will wrap `doc` in `$set` for you. This prevents you from accidentally overwriting the document. This option tells Mongoose to skip adding `$set`.
- [callback] «Function» params are (error, updateWriteOpResult)
- [callback] «Function»

Returns:

- «Query»

Updates one document in the database without returning it.

This function triggers the following middleware.

- `update()`

This method is deprecated. See Deprecation Warnings for details.

Examples:

```
MyModel.update({ age: { $gt: 18 } }, { oldEnough: true }, fn);

const res = await MyModel.update({ name: 'Tobi' }, { ferret: true });
res.n; // Number of documents that matched `{ name: 'Tobi' }`
// Number of documents that were changed. If every doc matched already
// had `ferret` set to `true`, `nModified` will be 0.
res.nModified;
```

Valid options:

- `strict` (boolean): overrides the schema-level `strict` option for this update
- `upsert` (boolean): whether to create the doc if it doesn't match (false)
- `writeConcern` (object): sets the write concern for replica sets. Overrides the schema-level write concern
- `multi` (boolean): whether multiple documents should be updated (false)
- `runValidators`: if true, runs update validators on this command. Update validators validate the update operation against the model's schema.
- `setDefaultsOnInsert` (boolean): if this and `upsert` are true, mongoose will apply the defaults specified in the model's schema if a new document is created. This option only works on MongoDB >= 2.4 because it relies on MongoDB's `$setOnInsert` operator.
- `timestamps` (boolean): If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.
- `overwrite` (boolean): disables update-only mode, allowing you to overwrite the doc (false)

All `update` values are cast to their appropriate SchemaTypes before being sent.

The `callback` function receives `(err, rawResponse)`.

- `err` is the error if any occurred
- `rawResponse` is the full response from Mongo

Note:

All top level keys which are not `atomic` operation names are treated as set operations:

Example:

```
const query = { name: 'borne' };
Model.update(query, { name: 'jason bourne' }, options, callback);

// is sent as
```

```
Model.update(query, { $set: { name: 'jason bourne' }}, options, function(err, res));
// if overwrite option is false. If overwrite is true, sent without the $set wrapper.
```

This helps prevent accidentally overwriting all documents in your collection with `{ name: 'jason bourne' }`.

Note:

Be careful to not use an existing model instance for the update clause (this won't work and can cause weird behavior like infinite loops). Also, ensure that the update clause does not have an _id property, which causes Mongo to return a "Mod on _id not allowed" error.

Model.updateMany()

Parameters

- filter «Object»
- update «Object|Array»
- [options] «Object» optional see `Query.prototype.setOptions()`
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Does nothing if schema-level timestamps are not set.
- [callback] «Function» `function(error, res) {}` where `res` has 5 properties: `modifiedCount`, `matchedCount`, `acknowledged`, `upsertedId`, and `upsertedCount`.

Returns:

- «Query»

Same as `update()`, except MongoDB will update *all* documents that match `filter` (as opposed to just the first one) regardless of the value of the `multi` option.

Note updateMany will *not* fire update middleware. Use `pre('updateMany')` and `post('updateMany')` instead.

Example:

```
const res = await Person.updateMany({ name: /Stark$/ }, { isDeleted: true });
res.matchedCount; // Number of documents matched
res.modifiedCount; // Number of documents modified
res.acknowledged; // Boolean indicating everything went smoothly.
```

```
res.upsertedId; // null or an id containing a document that had to be upserted.
res.upsertedCount; // Number indicating how many documents had to be upserted. Will either be 0 or
```

This function triggers the following middleware.

- `updateMany()`

Model.updateOne()

Parameters

- filter «Object»
- update «Object|Array»
- [options] «Object» optional see `Query.prototype.setOptions()`
  - [options.strict] «Boolean|String» overwrites the schema's strict mode option
  - [options.upsert=false] «Boolean» if true, and no documents found, insert a new document
  - [options.writeConcern=null] «Object» sets the write concern for replica sets. Overrides the schema-level write concern
  - [options.timestamps=null] «Boolean» If set to `false` and schema-level timestamps are enabled, skip timestamps for this update. Note that this allows you to overwrite timestamps. Does nothing if schema-level timestamps are not set.
- [callback] «Function» params are (error, writeOpResult)

Returns:

- «Query»

Same as `update()`, except it does not support the `multi` or `overwrite` options.

- MongoDB will update *only* the first document that matches `filter` regardless of the value of the `multi` option.
- Use `replaceOne()` if you want to overwrite an entire document rather than using atomic operators like `$set`.

Example:

```
const res = await Person.updateOne({ name: 'Jean-Luc Picard' }, { ship: 'USS Enterprise' });
res.matchedCount; // Number of documents matched
res.modifiedCount; // Number of documents modified
res.acknowledged; // Boolean indicating everything went smoothly.
res.upsertedId; // null or an id containing a document that had to be upserted.
res.upsertedCount; // Number indicating how many documents had to be upserted. Will either be 0 or
```

This function triggers the following middleware.

- `updateOne()`

Model.validate()

Parameters

- obj «Object»
- pathsToValidate «Array»
- [context] «Object»
- [callback] «Function»

Returns:

- «Promise,undefined»

Casts and validates the given object against this model's schema, passing the given `context` to custom validators.

Example:

```
const Model = mongoose.model('Test', Schema({
  name: { type: String, required: true },
  age: { type: Number, required: true }
});

try {
  await Model.validate({ name: null }, ['name'])
} catch (err) {
  err instanceof mongoose.Error.ValidationError; // true
  Object.keys(err.errors); // ['name']
}
```

Model.watch()

Parameters

- [pipeline] «Array»
- [options] «Object» see the mongodb driver options

Returns:

- «ChangeStream» mongoose-specific change stream wrapper, inherits from EventEmitter

Requires a replica set running MongoDB >= 3.6.0. Watches the underlying collection for changes using MongoDB change streams.

This function does **not** trigger any middleware. In particular, it does **not** trigger aggregate middleware.

The ChangeStream object is an event emitter that emits the following events

- 'change': A change occurred, see below example
- 'error': An unrecoverable error occurred. In particular, change streams currently error out if they lose connection to the replica set primary. Follow this GitHub issue for updates.
- 'end': Emitted if the underlying stream is closed
- 'close': Emitted if the underlying stream is closed

Example:

```
const doc = await Person.create({ name: 'Ned Stark' });
const changeStream = Person.watch().on('change', change => console.log(change));
// Will print from the above `console.log()`:
// { _id: { _data: ... },
//   operationType: 'delete',
//   ns: { db: 'mydb', coll: 'Person' },
//   documentKey: { _id: 5a51b125c5500f5aa094c7bd } }
await doc.remove();
```

Model.where()

Parameters

- path «String»
- [val] «Object» optional value

Returns:

- «Query»

Creates a Query, applies the passed conditions, and returns the Query.

For example, instead of writing:

```
User.find({age: {$gte: 21, $lte: 65}}, callback);
```

we can instead write:

```
User.where('age').gte(21).lte(65).exec(callback);
```

Since the Query class also supports `where` you can continue chaining

```
User
.where('age').gte(21).lte(65)
.where('name', /^b/i)
... etc
```

increment()

Signal that we desire an increment of this documents version.

Example:

```
Model.findById(id, function (err, doc) {
  doc.increment();
  doc.save(function (err) { .. })
})
```