Table of Contents

mongoose中文文档

> 官方原版：http://mongoosejs.com/docs/guide.html
>
> 翻译：小虾米（QQ:509129）

> 原文：Schemas
> 翻译：小虾米（QQ:509129）

schemas

> 如果你还没有开始，请花一分钟阅读快速入门学习Mongoose 是如何工作的。
> 如果你是从3.x迁移到4.x，请花一点时间来阅读迁移指南。

定义你的schema

Mongoose的一切都始于一个Schema。每个schema映射到MongoDB的集合(collection)和定义该集合(collection)中的文档的形式。

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var blogSchema = new Schema({
  title:  String,
  author: String,
  body:   String,
  comments: [{ body: String, date: Date }],
  date: { type: Date, default: Date.now },
  hidden: Boolean,
  meta: {
    votes: Number,
    favs:  Number
  }
});
```

如果您想稍后添加额外的键（keys），使用Schema#add方法。

在我们的blogSchema每个key在我们的文件将被转换为相关SchemaType定义一个属性。例如，我们定义了一个标题(title)将被转换为字符串(String)的 SchemaType 并将日期(date)转换为日期的 SchemaType 。键（keys）也可以被指定嵌套的对象，包含进一步的键/类型定义（例如，上面的 `meta` 属性）。

允许使用的SchemaTypes：

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array

阅读更多关于他们在这里。

Schemas不仅定义了文档和属性的结构，还定义了文档实例方法、静态模型方法、复合索引和文档被称为中间件的生命周期钩子。

创建一个模型

使用我们的schema定义，我们需要将我们的blogschema转成我们可以用的模型。为此，我们通过 `mongoose.model(modelName, schema)` ：

```
var Blog = mongoose.model('Blog', blogSchema);
// ready to go!
```

实例方法

模型的实例是文档（documents）。文档有许多自己内置的实例方法。我们也可以定义我们自己的自定义文档实例方法。

```
// define a schema
var animalSchema = new Schema({ name: String, type: String });

// assign a function to the "methods" object of our animalSchema
animalSchema.methods.findSimilarTypes = function(cb) {
  return this.model('Animal').find({ type: this.type }, cb);
};
```

现在我们所有的animal的实例有一个 `findsimilartypes` 方法可用。

```
var Animal = mongoose.model('Animal', animalSchema);
var dog = new Animal({ type: 'dog' });

dog.findSimilarTypes(function(err, dogs) {
  console.log(dogs); // woof
});
```

> 重写默认的mongoose文档方法可能会导致不可预测的结果。看到更多的细节。

静态方法（Statics）

给一个模型添加静态方法的也是很简单。继续我们的 `animalschema` ：

```
// assign a function to the "statics" object of our animalSchema
animalSchema.statics.findByName = function(name, cb) {
  return this.find({ name: new RegExp(name, 'i') }, cb);
};

var Animal = mongoose.model('Animal', animalSchema);
Animal.findByName('fido', function(err, animals) {
  console.log(animals);
});
```

查询助手

您还可以像实例方法那样添加查询辅助功能，这是，但对于mongoose的查询。查询辅助方法可以让你扩展mongoose链式查询生成器API。

```
animalSchema.query.byName = function(name) {
  return this.find({ name: new RegExp(name, 'i') });
};

var Animal = mongoose.model('Animal', animalSchema);
Animal.find().byName('fido').exec(function(err, animals) {
  console.log(animals);
});
```

索引（Indexes）

MongoDB支持 secondary indexes 。与 mongoose，我们定义这些indexes在我们的Schema的在路径级别或schema级别。在创建复合索引时，在 schema 级别上定义索引是必要的。

> 译者注：普通索引（index），唯一索引（unique），稀疏索引，时效索引（expires）

```
var animalSchema = new Schema({
  name: String,
  type: String,
  tags: { type: [String], index: true } // field level
});

animalSchema.index({ name: 1, type: -1 }); // schema level
```

> 当应用程序启动时，Mongoose为你的schema定义的每个索引自动调用 `ensureindex` 。Mongoose会按照每个索引的顺序调用ensureindex，并在模型上发出一个 `'index'` 事件，当所有的 `ensureindex` 调用返回成功或当时有一个错误返回。虽然很好的开发，建议这种行为禁用在生产中，因为索引创建可以导致一个显着的性能影响。通过设置schema的 `autoIndex` 选项为 `false` ，或对全局连接的设置选项 `config.autoindex` 为 `false` 。

```
mongoose.connect('mongodb://user:pass@localhost:port/database',
{ config: { autoIndex: false } });
// or
mongoose.createConnection('mongodb://user:pass@localhost:port/da
tabase', { config: { autoIndex: false } });
// or
animalSchema.set('autoIndex', false);
// or
new Schema({..}, { autoIndex: false });
```

查看 Model#ensureIndexes 方法。

虚拟属性

虚拟属性 是文档属性，您可以获取和设置但不保存到MongoDB。用于格式化或组合字段，从而制定者去组成一个单一的值为存储多个值是有用的。

```
// define a schema
var personSchema = new Schema({
  name: {
    first: String,
    last: String
  }
});

// compile our model
var Person = mongoose.model('Person', personSchema);

// create a document
var bad = new Person({
    name: { first: 'Walter', last: 'White' }
});
```

假设我们想打印bad的全名。我们可以这样做：

```
console.log(bad.name.first + ' ' + bad.name.last); // Walter Whi
te
```

或者我们可以在personschema定义 虚拟属性的getter ，这样我们不需要每次写出这个字符串的拼接：

```
personSchema.virtual('name.full').get(function () {
  return this.name.first + ' ' + this.name.last;
});
```

现在，我们进入我们的虚拟”的 `"name.full"` 资源的时候，我们将调用getter函数的返回值：

```
console.log('%s is insane', bad.name.full); // Walter White is i
nsane
```

注意，如果产生的记录转换为一个对象或JSON，virtuals不包括默认。通过 `virtuals` ： `true` 是 `toobject()` 或 `tojson()` 他们返回值。

这也是不错的，能够通过 `this.name.full` 设置 `this.name.first` 和 `this.name.last` 。例如，如果我们想respectively改变 `bad` 的 `name.first` 和 `name.last` 为 `'Breaking'` 和 `'Bad'` ，只需要：

```
bad.name.full = 'Breaking Bad';
```

Mongoose让你这样做也是通过虚拟属性的setter：

```
personSchema.virtual('name.full').set(function (name) {
  var split = name.split(' ');
  this.name.first = split[0];
  this.name.last = split[1];
});

...

mad.name.full = 'Breaking Bad';
console.log(mad.name.first); // Breaking
console.log(mad.name.last);  // Bad
```

[虚拟属性的setter 在其他验证之前使用。因此，上面的例子仍然可以工作，即使第一个和最后一个name字段是必需的。

只有非虚拟属性工作作为查询的一部分和字段选择。

选项（Options）

Schemas有几个可配置的选项，可以直接传递给构造函数或设置：

```
new Schema({..}, options);

// or

var schema = new Schema({..});
schema.set(option, value);
```

有效的选项：

- autoIndex
- capped
- collection
- emitIndexErrors
- id
- _id
- minimize
- read
- safe
- shardKey
- strict
- toJSON
- toObject
- typeKey
- validateBeforeSave
- versionKey
- skipVersioning
- timestamps

选项: autoIndex

在应用程序启动时，Mongoose在你的Schema为每一个索引声明发送一个 `ensureIndex` 命令。在Mongoose V3版本时，索引是默认在后台创建。如果你想禁用自动创建和手动创建索引时，将你的Schemas自动索引（autoIndex）选项设置为 `false` 和在你的模型使用 `ensureindexes` 方法。

```
var schema = new Schema({..}, { autoIndex: false });
var Clock = mongoose.model('Clock', schema);
Clock.ensureIndexes(callback);
```

选项: bufferCommands

默认情况下，mongoose缓冲命令一直存在直到驱动设法重新连接。禁用缓冲，设置 `buffercommands` 为 `false` 。

```
var schema = new Schema({..}, { bufferCommands: false });
```

选项: capped

Mongoose支持 MongoDBs capped集合。指定的MongoDB集合被封顶、设置封顶（capped）选项为文档的最大字节数。

```
new Schema({..}, { capped: 1024 });
```

capped选项也可以设置一个对象，如果你想通过附加选项，像Max或autoindexid。在这种情况下，您必须显式地通过size选项。

```
new Schema({..}, { capped: { size: 1024, max: 1000, autoIndexId:
true } });
```

选项: collection

Mongoose默认情况下通过模型名称的 `utils.tocollectionname` 方法产生的集合名称。这种方法复数名称。设置此选项，如果您需要不同的名称集合。

```
var dataSchema = new Schema({..}, { collection: 'data' });
```

选项: emitIndexErrors

默认情况下，mongoose会建立您在您的模式中指定的任何索引，并在模型中发出一个'index'事件，返回成功或错误。

```
MyModel.on('index', function(error) {
  /* If error is truthy, index build failed */
});
```

然而，这使得它很难捕捉当您的索引建立失败。`emitIndexErrors` 选项可是是你轻松看到索引建立失败。如果此选项打开，当索引建立失败时mongoose会同时在模型上发出一个 `'error'` 事件。

```
MyModel.schema.options.emitIndexErrors; // true
MyModel.on('error', function(error) {
  // gets an error whenever index build fails
});
```

如果一个错误事件被触发并没有听众，Node.js的内置事件抛出一个异常，所以很容易配置快速失败应用程序时的索引构建失败。

选项: id

Mongoose将你schemas id virtual getter 默认返回的文档_id场转换为字符串，或者ObjectIds，它的哈希字符串。如果你不想要一个lid getter加到你的schema，你可以它在schema构建的时通过这个选项禁用。

```
// default behavior
var schema = new Schema({ name: String });
var Page = mongoose.model('Page', schema);
var p = new Page({ name: 'mongodb.org' });
console.log(p.id); // '50341373e894ad16347efe01'

// disabled id
var schema = new Schema({ name: String }, { id: false });
var Page = mongoose.model('Page', schema);
var p = new Page({ name: 'mongodb.org' });
console.log(p.id); // undefined
```

选项: _id

Mongoose默认分配你的每个模式一个_id字段如果没有一个传递到模式构造函数。类型分配一个objectID配合MongoDB的默认行为。如果你不想要一个_id加到你的模式时，你可以使用这个选项禁用它。

您只能在子文档中使用此选项。Mongoose不能保存文档而不知道其id，所以你会保存一个没有_id文档会得到一个错误。

```
// default behavior
var schema = new Schema({ name: String });
var Page = mongoose.model('Page', schema);
var p = new Page({ name: 'mongodb.org' });
console.log(p); // { _id: '50341373e894ad16347efe01', name: 'mon
godb.org' }

// disabled _id
var childSchema = new Schema({ name: String }, { _id: false });
var parentSchema = new Schema({ children: [childSchema] });

var Model = mongoose.model('Model', parentSchema);

Model.create({ children: [{ name: 'Luke' }] }, function(error, d
oc) {
  // doc.children[0]._id will be undefined
});
```

选项: minimize

Mongoose，默认情况下，“minimize”模式通过删除空对象。

```
var schema = new Schema({ name: String, inventory: {} });
var Character = mongoose.model('Character', schema);

// will store `inventory` field if it is not empty
var frodo = new Character({ name: 'Frodo', inventory: { ringOfPo
wer: 1 }});
Character.findOne({ name: 'Frodo' }, function(err, character) {
  console.log(character); // { name: 'Frodo', inventory: { ringO
fPower: 1 }}
});

// will not store `inventory` field if it is empty
var sam = new Character({ name: 'Sam', inventory: {}});
Character.findOne({ name: 'Sam' }, function(err, character) {
  console.log(character); // { name: 'Sam' }
});
```

这种行为可以通过设置minimize选项为false。它将存储空的对象。

```
var schema = new Schema({ name: String, inventory: {} }, { minim
ize: false });
var Character = mongoose.model('Character', schema);

// will store `inventory` if empty
var sam = new Character({ name: 'Sam', inventory: {}});
Character.findOne({ name: 'Sam' }, function(err, character) {
  console.log(character); // { name: 'Sam', inventory: {}}
});
```

选项: read

允许设置query#read 选项在模式层面，为我们提供一种方法来使用默认值ReadPreferences为模型的所有查询。

```
var schema = new Schema({..}, { read: 'primary' });            /
/ also aliased as 'p'
var schema = new Schema({..}, { read: 'primaryPreferred' });   /
/ aliased as 'pp'
var schema = new Schema({..}, { read: 'secondary' });          /
/ aliased as 's'
var schema = new Schema({..}, { read: 'secondaryPreferred' }); /
/ aliased as 'sp'
var schema = new Schema({..}, { read: 'nearest' });            /
/ ali
```

每个首选的别名允许替代而不是必须输入'secondaryPreferred'并得到拼写错误，我们可以简单地通过'sp'。

读选项还允许我们指定标记集。这些告诉驱动程序的副本集的成员应该尝试读取。阅读更多关于标签集这里和这里。

注意：您也可以指定驱动程序读取优先策略选择当连接时：

```
// pings the replset members periodically to track network laten
cy
var options = { replset: { strategy: 'ping' }};
mongoose.connect(uri, options);

var schema = new Schema({..}, { read: ['nearest', { disk: 'ssd'
}] });
mongoose.model('JellyBean', schema);
```

选项: safe

这个选项是通过与MongoDB所有操作并指定如果错误应该回到我们的回调以及调写行为。

```
var safe = true;
new Schema({ .. }, { safe: safe });
```

默认设置为true应用为所有模式，可以保证任何发生的错误被传递回我们的回调函数。通过设置安全的东西像 `{ j: 1, w: 2, wtimeout: 10000 }` ，我们可以保证写致力于MongoDB journal (j: 1)，至少有2个副本 (w: 2)，并写会超时如果需要超过10秒 (wtimeout: 10000)。错误仍然会被传递给我们的回调。

注：在3.6.x，你也需要把版本删掉。在 3.7.x 及以上版本会自动被禁用当安全设置为false。

**注：此设置将覆盖任何设置指定通过传递数据库选项，同时创建一个连接。

还有其他写的问题，如 `{ w: "majority" }` 。看MongoDB文档详情。

```
var safe = { w: "majority", wtimeout: 10000 };
new Schema({ .. }, { safe: safe });
```

选项: shardKey

shardkey选项是用来当我们有一个分片的MongoDB架构。每个分片集提供一个shard key必须存在于所有的插入/更新操作。我们只需要设置此模式选项相同的shard key，我们将所有设置。

```
new Schema({ .. }, { shardKey: { tag: 1, name: 1 }})
```

> 注意，Mongoose不发送 `shardcollection` 命令给你。你必须配置你自己的分片。

选项: strict

严格选项（默认启用），确保传递给我们的模型构造函数的值没有被指定在我们的模式中，不会被保存到数据库中。

```
var thingSchema = new Schema({..})
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing({ iAmNotInTheSchema: true });
thing.save(); // iAmNotInTheSchema is not saved to the db

// set to false..
var thingSchema = new Schema({..}, { strict: false });
var thing = new Thing({ iAmNotInTheSchema: true });
thing.save(); // iAmNotInTheSchema is now saved to the db!!
```

使用 `doc.set()` 来设置属性值这样也会起作用。

```
var thingSchema = new Schema({..})
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing;
thing.set('iAmNotInTheSchema', true);
thing.save(); // iAmNotInTheSchema is not saved to the db
```

这个值可以被覆盖，在模型实例层通过第二个尔参数：

```
var Thing = mongoose.model('Thing');
var thing = new Thing(doc, true);  // enables strict mode
var thing = new Thing(doc, false); // disables strict mode
```

严格的选项也可能被设置为“throw”，这将导致错误产生，而不是丢弃坏数据。

注意：不要设置为false，除非你有很好的理由。

注：在mongoose v2默认是false的。

注意：任何键/值设置在实例上

注意：不管在不在你的模式中存在的任何键/值的实例将被忽略，无论模式选项。

```
var thingSchema = new Schema({..})
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing;
thing.iAmNotInTheSchema = true;
thing.save(); // iAmNotInTheSchema is never saved to the db
```

选项: toJSON

Exactly the same as the toObject option but only applies when the documents toJSON method is called.

完全一样的toObject选项，但只适用于当文件tojson方法称为。

```
var schema = new Schema({ name: String });
schema.path('name').get(function (v) {
  return v + ' is my name';
});
schema.set('toJSON', { getters: true, virtuals: false });
var M = mongoose.model('Person', schema);
var m = new M({ name: 'Max Headroom' });
console.log(m.toObject()); // { _id: 504e0cd7dd992d9be2f20b6f, n
ame: 'Max Headroom' }
console.log(m.toJSON()); // { _id: 504e0cd7dd992d9be2f20b6f, nam
e: 'Max Headroom is my name' }
// since we know toJSON is called whenever a js object is string
ified:
console.log(JSON.stringify(m)); // { "_id": "504e0cd7dd992d9be2f
20b6f", "name": "Max Headroom is my name" }
```

要查看所有可用的toJSON/toObject 选项，读这。

选项: toObject

文档有一个toObject的方法将mongoose文档转成成到一个普通的JavaScript对象。此方法接受一些选项。代替这些选项在一个文档的基础上，我们可以声明这里的选项，并将其应用到所有的这个模式文档为默认。

让所有的虚函数显示在你的console.log输出，设置toObject选项为 { getters: true }：

```
var schema = new Schema({ name: String });
schema.path('name').get(function (v) {
  return v + ' is my name';
});
schema.set('toObject', { getters: true });
var M = mongoose.model('Person', schema);
var m = new M({ name: 'Max Headroom' });
console.log(m); // { _id: 504e0cd7dd992d9be2f20b6f, name: 'Max H
eadroom is my name' }
```

要查看所有可用的toObject选择，读这。

选项: typeKey

默认情况下，如果在你的模式中你有一个对象'type'为键，mongoose只会把它理解为一种类型声明。

```
// Mongoose interprets this as 'loc is a String'
var schema = new Schema({ loc: { type: String, coordinates: [Num
ber] } });
```

然而，对于像GeoJSON这样的应用程序，'typ'属性是很重要的。如果你想控制键mongoose用找到的类型声明，设置'typeKey'模式选项。

```
var schema = new Schema({
  // Mongoose interpets this as 'loc is an object with 2 keys, t
ype and coordinates'
  loc: { type: String, coordinates: [Number] },
  // Mongoose interprets this as 'name is a String'
  name: { $type: String }
}, { typeKey: '$type' }); // A '$type' key means this object is
a type declaration
```

选项: validateBeforeSave

默认情况下，在将文档保存到数据库之前，文档将自动验证。这是为了防止保存无效的文档。如果你想手动处理验证，能够保存不通过验证的对象，您可以设置validatebeforesave为false。

```
var schema = new Schema({ name: String });
schema.set('validateBeforeSave', false);
schema.path('name').validate(function (value) {
    return v != null;
});
var M = mongoose.model('Person', schema);
var m = new M({ name: null });
m.validate(function(err) {
    console.log(err); // Will tell you that null is not allowed.
});
m.save(); // Succeeds despite being invalid
```

选项: versionKey

versionKey是一个设置在每个文档上的属性当第一次被Mongoose创建时。此键值包含文档的内部修订版。versionkey选项是一个字符串，表示使用版本控制路径。默认的是 `__v` 。如果这种与您的应用程序冲突你可以配置：

```
var schema = new Schema({ name: 'string' });
var Thing = mongoose.model('Thing', schema);
var thing = new Thing({ name: 'mongoose v3' });
thing.save(); // { __v: 0, name: 'mongoose v3' }

// customized versionKey
new Schema({..}, { versionKey: '_somethingElse' })
var Thing = mongoose.model('Thing', schema);
var thing = new Thing({ name: 'mongoose v3' });
thing.save(); // { _somethingElse: 0, name: 'mongoose v3' }
```

文档版本也可以通过设置versionkey为false禁用。不禁用版本除非你知道你正在做什么。

```
new Schema({..}, { versionKey: false });
var Thing = mongoose.model('Thing', schema);
var thing = new Thing({ name: 'no versioning please' });
thing.save(); // { name: 'no versioning please' }
```

选项: skipVersioning

skipversioning允许从versioning扣除路径（例如，内部的修改不会增加即使这些路径更新）。不要这样做，除非你知道你在做什么。对于子文档，使用完全限定的路径将此文件包含在父文档中。

```
new Schema({..}, { skipVersioning: { dontVersionMe: true } });
thing.dontVersionMe.push('hey');
thing.save(); // version is not incremented
```

选项: timestamps

如果设置时间戳，mongoose分配 `createdAt` 和 `updatedAt` 字段到你的模式汇总，类型指定为Date。

By default, the name of two fields are createdAt and updatedAt, custom the field name by setting timestamps.createdAt and timestamps.updatedAt.

默认情况下，两个字段的名字是 `createdAt` 和 `updatedAt` ，自定义字段名称设置 `timestamps.createdat` 和 `timestamps.updatedat` 。

```
var thingSchema = new Schema({..}, { timestamps: { createdAt: 'c
reated_at' } });
var Thing = mongoose.model('Thing', thingSchema);
var thing = new Thing();
thing.save(); // `created_at` & `updatedAt` will be included
```

选项: useNestedStrict

在mongoose 4中， `update()` 和 `findoneandupdate()` 只检查顶层模式的严格模式设置。

```
var childSchema = new Schema({}, { strict: false });
var parentSchema = new Schema({ child: childSchema }, { strict:
'throw' });
var Parent = mongoose.model('Parent', parentSchema);
Parent.update({}, { 'child.name': 'Luke Skywalker' }, function(e
rror) {
  // Error because parentSchema has `strict: throw`, even though
  // `childSchema` has `strict: false`
});

var update = { 'child.name': 'Luke Skywalker' };
var opts = { strict: false };
Parent.update({}, update, opts, function(error) {
  // This works because passing `strict: false` to `update()` ov
erwrites
  // the parent schema.
});
```

如果你设置 `useNestedStrict` 为true，mongoose会使用子模式的严格选项铸造更新。

```
var childSchema = new Schema({}, { strict: false });
var parentSchema = new Schema({ child: childSchema },
  { strict: 'throw', useNestedStrict: true });
var Parent = mongoose.model('Parent', parentSchema);
Parent.update({}, { 'child.name': 'Luke Skywalker' }, function(e
rror) {
  // Works!
});
```

Pluggable

模式也可允许我们打包成可重用的功能插件，可以与社区或是你的项目之间共享。

下一步

既然我们已经掌握了 `Schemas` ，让我们看看模式类型（SchemaTypes）。

> 原文：SchemaTypes
> 翻译：小虾米（QQ:509129）

SchemaTypes

SchemaTypes 为查询处理定义path defaults, validation, getters, setters, field selection defaults ，以及为了字符串和数值的其他一般特征。检查他们各自的的更多细节的 API 文档。

以下是所有有效的Schema Types。

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- Objectid
- Array

例子

```
var schema = new Schema({
  name:    String,
  binary:  Buffer,
  living:  Boolean,
  updated: { type: Date, default: Date.now },
  age:     { type: Number, min: 18, max: 65 },
  mixed:   Schema.Types.Mixed,
  _someId: Schema.Types.ObjectId,
  array:      [],
  ofString:   [String],
  ofNumber:   [Number],
  ofDates:    [Date],
  ofBuffer:   [Buffer],
  ofBoolean:  [Boolean],
  ofMixed:    [Schema.Types.Mixed],
  ofObjectId: [Schema.Types.ObjectId],
```

```
  nested: {
    stuff: { type: String, lowercase: true, trim: true }
  }
})

// example use

var Thing = mongoose.model('Thing', schema);

var m = new Thing;
m.name = 'Statue of Liberty';
m.age = 125;
m.updated = new Date;
m.binary = new Buffer(0);
m.living = false;
m.mixed = { any: { thing: 'i want' } };
m.markModified('mixed');
m._someId = new mongoose.Types.ObjectId;
m.array.push(1);
m.ofString.push("strings!");
m.ofNumber.unshift(1,2,3,4);
m.ofDates.addToSet(new Date);
m.ofBuffer.pop();
m.ofMixed = [1, [], 'three', { four: 5 }];
m.nested.stuff = 'good';
m.save(callback);
```

使用说明：

Dates

内置的日期的方法在mongoose用英语改变跟踪逻辑不是钩子，意味着如果您在文档中使用日期，并用类似的方法进行修改如 `setmonth()` ，mongoose不知道这种变化，并且 `doc.save()` 不会坚持这样的修改。如果你必须使用内置的方法修改日期类型，保存之前告诉mongoose关于 `doc.markModified('pathToYourDate')` 的变化。

```
var Assignment = mongoose.model('Assignment', { dueDate: Date })
;
Assignment.findOne(function (err, doc) {
  doc.dueDate.setMonth(3);
  doc.save(callback); // THIS DOES NOT SAVE YOUR CHANGE

  doc.markModified('dueDate');
  doc.save(callback); // works
})
```

Mixed

一个“什么都行”的SchemaType，它的灵活性会带来权衡它难维护。混合类型可通过 `schema.types.mixed` 或通过传递一个空的对象。以下是等价的：

```
var Any = new Schema({ any: {} });
var Any = new Schema({ any: Schema.Types.Mixed });
```

因为它是一个无模式的类型，您可以将值更改为其他任何你喜欢的东西，但是Mongoose失去自动检测和保存这些变化的能力。“告诉”Mongoose，混合型的价值观已经改变了，调用.markModified(path)方法使文档传递路径到您刚刚更改的混合类型。

```
person.anything = { x: [3, 4, { y: "changed" }] };
person.markModified('anything');
person.save(); // anything will now get saved
```

ObjectIds

指定一个ObjectId类型，使用 `Schema.Types` 。ObjectId在你的声明中。

```
var mongoose = require('mongoose');
var ObjectId = mongoose.Schema.Types.ObjectId;
var Car = new Schema({ driver: ObjectId });
// or just Schema.ObjectId for backwards compatibility with v2
```

Arrays

提供创建数组的 `SchemaTypes` 或子文档。

```
var ToySchema = new Schema({ name: String });
var ToyBox = new Schema({
  toys: [ToySchema],
  buffers: [Buffer],
  string:  [String],
  numbers: [Number]
  // ... etc
});
```

注：指定一个空数组是相当于混合类型（ `Mixed` ）。以下所有创建混合数组：

```
var Empty1 = new Schema({ any: [] });
var Empty2 = new Schema({ any: Array });
var Empty3 = new Schema({ any: [Schema.Types.Mixed] });
var Empty4 = new Schema({ any: [{}] });
```

创建自定义类型

Mongoose也可以扩展自定义 SchemaTypes 。搜索兼容类型的插件站点，如，mongoose-long，mongoose-int32以及其他类型。要创建您自己的自定义schema，请看一看创建基本的自定义schema类型。

下一步

既然我们已经掌握了SchemaTypes，让我们看一看模型。

> 原文：Creating a Basic Custom Schema Type
> 翻译：小虾米（QQ:509129）

创建一个基本的自定义模式类型

Mongoose 4.4.0的新特征：Mongoose支持自定义类型。在你达到一个自定义的类型，然而，知道一个自定义的类型是过度对大多数用例。你可以用最基本的任务custom getters/setters，虚函数（virtuals）和单一的嵌入式文档。

让我们看看一个基本schema类型看一个例子：一个字节的整数。创建一个新的schema类型，你需要继承 `mongoose.SchemaType` 并且将相应的属性添加到 `mongoose.Schema.Types` 。你需要实现一个方法是 `cast()` 方法。

```
function Int8(key, options) {
  mongoose.SchemaType.call(this, key, options, 'Int8');
}
Int8.prototype = Object.create(mongoose.SchemaType.prototype
);

// `cast()` takes a parameter that can be anything. You need
to
// validate the provided `val` and throw a `CastError` if you

// can't convert it.
Int8.prototype.cast = function(val) {
  var _val = Number(val);
  if (isNaN(_val)) {
    throw new Error('Int8: ' + val + ' is not a number');
  }
  _val = Math.round(_val);
  if (_val < -0x80 || _val > 0x7F) {
    throw new Error('Int8: ' + val +
      ' is outside of the range of valid 8-bit ints');
  }
  return _val;
};

// Don't forget to add `Int8` to the type registry
mongoose.Schema.Types.Int8 = Int8;
```

```
    var testSchema = new Schema({ test: Int8 });
    var Test = mongoose.model('Test', testSchema);

    var t = new Test();
    t.test = 'abc';
    assert.ok(t.validateSync());
    assert.equal(t.validateSync().errors['test'].name, 'CastErro
r');
    assert.equal(t.validateSync().errors['test'].message,
      'Cast to Int8 failed for value "abc" at path "test"');
    assert.equal(t.validateSync().errors['test'].reason.message,
      'Int8: abc is not a number');
```

> 原文：Models
> 翻译：小虾米（QQ:509129）

Models

从我们的Schema定义模型的构造函数编译。实例这些模型代表文档可以从我们的数据库中保存和检索。从数据库中依靠这些模型来操作所有文档创建和检索。

编译你的第一个模型

```
var schema = new mongoose.Schema({ name: 'string', size: 'string'
 });
var Tank = mongoose.model('Tank', schema);
```

第一个参数是你的模型集合的单数名称。Mongoose会自动寻找你的模型名称的复数形式。因此，对于上面的示例，模型 `Tank` 是用于数据库中的 `tanks` 集合的。.model() 功能使得到schema的副本。确信你已经添加了你想要的一切在调用.model()之前！

构建文档

文档是我们模型的实例。创建它们，并保存到数据库是很容易的：

```
var Tank = mongoose.model('Tank', yourSchema);

var small = new Tank({ size: 'small' });
small.save(function (err) {
  if (err) return handleError(err);
  // saved!
})

// or

Tank.create({ size: 'small' }, function (err, small) {
  if (err) return handleError(err);
  // saved!
})
```

请注意，没有 `tanks` 将创建/删除，直到连接您的模型使用是打开的。每一个模型都有一个关联的连接。当你使用 `mongoose.model()` 。你的模型将要使用默认mongoose连接。

```
mongoose.connect('localhost', 'gettingstarted');
```

如果你创建一个自定义的连接，使用连接的 `model()` 函数代替。

```
var connection = mongoose.createConnection('mongodb://localhost:
27017/test');
var Tank = connection.model('Tank', yourSchema);
```

查询

对于Mongoosecha的查找文档很容易，它支持丰富的查询MongoDB语法。文件可以使用每个模型中使用 find，findById，findOne，或者where，静态方法。

```
Tank.find({ size: 'small' }).where('createdDate').gt(oneYearAgo)
.exec(callback);
```

查看关于如何使用查询API的详细信息的querying章节。

删除

模型有一个静态删除方法，可用于移除所有匹配条件的文档。

```
Tank.remove({ size: 'large' }, function (err) {
  if (err) return handleError(err);
  // removed!
});
```

更新

每个模型都有自己的更新方法，用于修改数据库中的文档，不将它们返回到您的应用程序。

详细看API文档。

如果你想要更新一个文档数据库，并将结果返回给你的应用程序，使用 `findOneAndUpdate` 代替。

然而，更多的

API文档包含了许多额外的方法，像 count，mapReduce，aggregate，更多。

> 原文：Documents
> 翻译：小虾米（QQ:509129）

Documents

Mongoose文档代表了文档存储在MongoDB的一对一映射。每个文档都是它的模型的一个实例。

检索

有许多方法从MongoDB文档检索。我们不会涵盖在在这一部分中。查看查询详细信息的'querying'章节。

更新

有一些方法来更新文档。我们先看看使用 `findById` 的传统方法：

```
Tank.findById(id, function (err, tank) {
  if (err) return handleError(err);

  tank.size = 'large';
  tank.save(function (err) {
    if (err) return handleError(err);
    res.send(tank);
  });
});
```

这种方法包括首先检索文档从Mongo，然后发出更新命令（通过触发调用保存）。然而，如果我们不需要的文档在我们的应用程序中返回，只需要更新一个属性直接在数据库、 `Model#update` 对我们是正确的：

```
Tank.update({ _id: id }, { $set: { size: 'large' }}, callback);
```

如果我们的应用程序需要文档返回，则需要另一个方法，往往更好，选项：

```
Tank.findByIdAndUpdate(id, { $set: { size: 'large' }}, function (
err, tank) {
  if (err) return handleError(err);
  res.send(tank);
});
```

`findAndUpdate/Remove` 静态方法都在最多一个文档中进行更改，并返回它只有一个调用数据库。有几种不同的 `findandmodify` 主题。阅读API文档的更多细节。注意 `findAndUpdate/Remove` 不执行任何钩子或验证在在数据库中进行更改之前。如果你需要钩子和验证,首先查询文档,然后保存它。

> 译者注：findByIdAndRemove，findOneAndUpdate，findAndModify

验证

在保存文档之前，文档进行了验证。详情阅读API文档或validation章节。

下一步

既然我们已经掌握了文档，让我们看看子文档。

> 原文：Sub Docs
> 翻译：小虾米（QQ:509129）
> 进度：未完成

sub docs

Sub-documents are docs with schemas of their own which are elements of a parents document array:

```
var childSchema = new Schema({ name: 'string' });

var parentSchema = new Schema({
  children: [childSchema]
})
```

> 原文：Queries
> 翻译：小虾米（QQ:509129）

Queries

可以通过模型的几个静态辅助方法检索文档。

任何涉及指定查询条件的模型方法都可以执行两种方法：

当一个回调函数：

- 通过，操作将立即执行，结果传递给回调。
- 未通过，查询的一个实例被返回，它提供了一个特殊的查询生成器接口。

> 在mongoose 4，一个查询有一个 `.then()` 功能，因此可以被用来作为一个承诺。

当执行一个查询回调函数，你指定一个JSON文档作为你的查询。JSON文档的语法是MongoDB的shell一样。

```
var Person = mongoose.model('Person', yourSchema);

// find each person with a last name matching 'Ghost', selecting
 the `name` and `occupation` fields
Person.findOne({ 'name.last': 'Ghost' }, 'name occupation', func
tion (err, person) {
  if (err) return handleError(err);
  console.log('%s %s is a %s.', person.name.first, person.name.l
ast, person.occupation) // Space Ghost is a talk show host.
})
```

在这里，我们看到，查询立即执行，并将结果传递给我们的回调。所有在Mongoose的回调使用模式： `callback(error, result)` 。如果执行查询时发生错误，则错误参数将包含一个错误文档，结果将是无效的。如果查询成功，错误参数将为空，结果将与查询的结果填充。

> 在Mongoose中任何一个回调传递给一个查询，回调如下模式 `callback(error, results)` 。什么样的结果是取决于操作：为 findone() 它是一种单文档，find() 一个文档列表，count()文档数量，update() 文件数量等影响，API文档模型提供更加详细的关于什么是传递给回调函数。

现在让我们看看在没有回调时发生了什么：

```
// find each person with a last name matching 'Ghost'
var query = Person.findOne({ 'name.last': 'Ghost' });

// selecting the `name` and `occupation` fields
query.select('name occupation');

// execute the query at a later time
query.exec(function (err, person) {
  if (err) return handleError(err);
  console.log('%s %s is a %s.', person.name.first, person.name.l
ast, person.occupation) // Space Ghost is a talk show host.
})
```

在上面的代码中，查询变量是类型查询。查询允许您建立一个查询使用链式语法，而不是指定一个JSON对象。下面的2个例子是等价的。

```
// With a JSON doc
Person.
  find({
    occupation: /host/,
    'name.last': 'Ghost',
    age: { $gt: 17, $lt: 66 },
    likes: { $in: ['vaporizing', 'talking'] }
  }).
  limit(10).
  sort({ occupation: -1 }).
  select({ name: 1, occupation: 1 }).
  exec(callback);

// Using query builder
Person.
  find({ occupation: /host/ }).
  where('name.last').equals('Ghost').
  where('age').gt(17).lt(66).
  where('likes').in(['vaporizing', 'talking']).
  limit(10).
  sort('-occupation').
  select('name occupation').
  exec(callback);
```

一个完整的查询辅助功能列表可以在API文档中找到。

其他文档的参考资料

There are no joins in MongoDB but sometimes we still want references to documents in other collections. This is where population comes in. Read more about how to include documents from other collections in your query results here.

Streaming

你可以从MongoDB查询结果。你需要调用Query#cursor()函数代替Query#exec返回一个 QueryCursor实例。

```
var cursor = Person.find({ occupation: /host/ }).cursor();
cursor.on('data', function(doc) {
  // Called once for every document
});
cursor.on('close', function() {
  // Called when done
});
```

下一步

既然我们已经掌握了查询，让我们看看验证。

> 原文：Validation
> 翻译：小虾米（QQ:509129）

Validation

在我们进入验证语法的细节之前，请记住以下的规则：

- 验证是在SchemaType定义
- 验证是中间件。Mongoose寄存器验证作为 `pre('save')` 钩子在每个模式默认情况下。
- 你可以手动使用doc运行验证。 `validate(callback) or doc.validateSync()` 。
- 验证程序不运行在未定义的值上。唯一的例外是 `required` 验证器。
- 验证异步递归；当你调用Model#save，子文档验证也可以执行。如果出现错误，你的 Model#save回调接收它。
- 验证是可定制的。

```
var schema = new Schema({
     name: {
       type: String,
       required: true
     }
   });
   var Cat = db.model('Cat', schema);

   // This cat has no name :(
   var cat = new Cat();
   cat.save(function(error) {
     assert.equal(error.errors['name'].message,
       'Path `name` is required.');

     error = cat.validateSync();
     assert.equal(error.errors['name'].message,
       'Path `name` is required.');
   });
```

内置验证器

Mongoose有几个内置验证器。

- 所有的schematypes有内置的require验证器。所需的验证器使用SchemaType的checkrequired()函数确定的值满足所需的验证器。
- 数值（ Numbers ）有最大（man）和最小（min）的验证器。
- 字符串（String）有枚举，match，maxLength和minLength验证器。

每一个上述的验证链接提供更多的信息关于如何使他们和自定义错误信息。

```
var breakfastSchema = new Schema({
      eggs: {
        type: Number,
        min: [6, 'Too few eggs'],
        max: 12
      },
      bacon: {
        type: Number,
        required: [true, 'Why no bacon?']
      },
      drink: {
        type: String,
        enum: ['Coffee', 'Tea']
      }
    });
    var Breakfast = db.model('Breakfast', breakfastSchema);

    var badBreakfast = new Breakfast({
      eggs: 2,
      bacon: 0,
      drink: 'Milk'
    });
    var error = badBreakfast.validateSync();
    assert.equal(error.errors['eggs'].message,
      'Too few eggs');
    assert.ok(!error.errors['bacon']);
    assert.equal(error.errors['drink'].message,
      '`Milk` is not a valid enum value for path `drink`.');

    badBreakfast.bacon = null;
    error = badBreakfast.validateSync();
    assert.equal(error.errors['bacon'].message, 'Why no bacon?')
;
```

自定义验证器

如果内置验证器是不够的话，你可以自定义验证器来适应你的需求。

自定义验证是通过传递验证函数声明的。你可以找到详细说明如何在 SchemaType#validate() API文档。

```
var userSchema = new Schema({
    phone: {
      type: String,
      validate: {
        validator: function(v) {
          return /\d{3}-\d{3}-\d{4}/.test(v);
        },
        message: '{VALUE} is not a valid phone number!'
      },
      required: [true, 'User phone number required']
    }
  });

  var User = db.model('user', userSchema);
  var user = new User();
  var error;

  user.phone = '555.0123';
  error = user.validateSync();
  assert.equal(error.errors['phone'].message,
    '555.0123 is not a valid phone number!');

  user.phone = '';
  error = user.validateSync();
  assert.equal(error.errors['phone'].message,
    'User phone number required');

  user.phone = '201-555-0123';
  // Validation succeeds! Phone number is defined
  // and fits `DDD-DDD-DDDD`
  error = user.validateSync();
  assert.equal(error, null);
```

异步的自定义验证器

自定义验证器可以异步。如果你的验证函数接受2个参数，mongoose将视为第二个参数是一个回调。

即使你不想使用异步验证，小心，因为mongoose 4 视为所有的带2个参数函数是异步的，像 `validator.isEmail` 功能。

```
var userSchema = new Schema({
      phone: {
        type: String,
        validate: {
          validator: function(v, cb) {
            setTimeout(function() {
              cb(/\d{3}-\d{3}-\d{4}/.test(v));
            }, 5);
          },
          message: '{VALUE} is not a valid phone number!'
        },
        required: [true, 'User phone number required']
      }
    });

    var User = db.model('User', userSchema);
    var user = new User();
    var error;

    user.phone = '555.0123';
    user.validate(function(error) {
      assert.ok(error);
      assert.equal(error.errors['phone'].message,
        '555.0123 is not a valid phone number!');
    });
```

验证错误

验证失败后Errors返回一个错误的对象实际上是validatorerror对象。每个ValidatorError有kind, path, value, and message属性。

```
var toySchema = new Schema({
      color: String,
      name: String
    });

    var Toy = db.model('Toy', toySchema);

    var validator = function (value) {
      return /blue|green|white|red|orange|periwinkle/i.test(valu
e);
    };
    Toy.schema.path('color').validate(validator,
      'Color `{VALUE}` not valid', 'Invalid color');

    var toy = new Toy({ color: 'grease'});

    toy.save(function (err) {
      // err is our ValidationError object
      // err.errors.color is a ValidatorError object
      assert.equal(err.errors.color.message, 'Color `grease` not
 valid');
      assert.equal(err.errors.color.kind, 'Invalid color');
      assert.equal(err.errors.color.path, 'color');
      assert.equal(err.errors.color.value, 'grease');
      assert.equal(err.name, 'ValidationError');
    });
```

Required验证在嵌套的对象

定义嵌套对象验证器在mongoose是是棘手的，因为嵌套对象不完全成熟的路径。

```
var personSchema = new Schema({
      name: {
        first: String,
        last: String
      }
    });

    assert.throws(function() {
      // This throws an error, because 'name' isn't a full fledg
ed path
      personSchema.path('name').required(true);
    }, /Cannot.*'required'/);

    // To make a nested object required, use a single nested sch
ema
    var nameSchema = new Schema({
      first: String,
      last: String
    });

    personSchema = new Schema({
      name: {
        type: nameSchema,
        required: true
      }
    });

    var Person = db.model('Person', personSchema);

    var person = new Person();
    var error = person.validateSync();
    assert.ok(error.errors['name']);
```

更新（update）验证器

上面的例子，你了解了文档的验证。Mongoose也支持 `update()` 和 `findoneandupdate()` 操作的验证。在Mongoose 4.x，更新验证器是默认关闭-您需要指定 `runvalidators` 选项。

开启更新验证器，设置runValidators选项为 `update()` 或 `findOneAndUpdate()` 。小心：更新验证器默认关闭因为他们有几个注意事项。

```
var toySchema = new Schema({
      color: String,
      name: String
    });

    var Toy = db.model('Toys', toySchema);

    Toy.schema.path('color').validate(function (value) {
      return /blue|green|white|red|orange|periwinkle/i.test(valu
e);
    }, 'Invalid color');

    var opts = { runValidators: true };
    Toy.update({}, { color: 'bacon' }, opts, function (err) {
      assert.equal(err.errors.color.message,
        'Invalid color');
    });
```

更新（update）验证器和this

更新（update）验证器和文档（document）验证器有一个主要的区别。在上面的颜色验证功能中，这是指在使用文档（document）验证时所进行的文档的验证。然而，运行更新验证时，被更新的文件可能不在服务器的内存中，所以默认情况下这个值不定义。

```
 var toySchema = new Schema({
      color: String,
      name: String
    });

    toySchema.path('color').validate(function(value) {
      // When running in `validate()` or `validateSync()`, the
      // validator can access the document using `this`.
      // Does **not** work with update validators.
      if (this.name.toLowerCase().indexOf('red') !== -1) {
        return value !== 'red';
      }
      return true;
    });

    var Toy = db.model('ActionFigure', toySchema);

    var toy = new Toy({ color: 'red', name: 'Red Power Ranger' }
);
    var error = toy.validateSync();
    assert.ok(error.errors['color']);

    var update = { color: 'red', name: 'Red Power Ranger' };
    var opts = { runValidators: true };

    Toy.update({}, update, opts, function(error) {
      // The update validator throws an error:
      // "TypeError: Cannot read property 'toLowerCase' of undef
ined",
      // because `this` is **not** the document being updated wh
en using
      // update validators
      assert.ok(error);
    });
```

context 选项

context选项允许你在更新验证器设置此值为相关查询。

```
 toySchema.path('color').validate(function(value) {
      // When running update validators with the `context` optio
n set to
      // 'query', `this` refers to the query object.
      if (this.getUpdate().$set.name.toLowerCase().indexOf('red'
) !== -1) {
        return value === 'red';
      }
      return true;
    });

    var Toy = db.model('Figure', toySchema);

    var update = { color: 'blue', name: 'Red Power Ranger' };
    // Note the context option
    var opts = { runValidators: true, context: 'query' };

    Toy.update({}, update, opts, function(error) {
      assert.ok(error.errors['color']);
    });
```

路径更新验证器 （Update Validator Paths）

另一个主要差别，更新验证器只能运行在指定的路径中的更新。例如，在下面的示例中，因为在更新操作中没有指定“name”，更新验证将成功。

使用更新（update）验证器，required验证器验证失败时，你要明确地 `$unset` 这个key。

```
var kittenSchema = new Schema({
      name: { type: String, required: true },
      age: Number
    });

    var Kitten = db.model('Kitten', kittenSchema);

    var update = { color: 'blue' };
    var opts = { runValidators: true };
    Kitten.update({}, update, opts, function(err) {
      // Operation succeeds despite the fact that 'name' is not
specified
    });

    var unset = { $unset: { name: 1 } };
    Kitten.update({}, unset, opts, function(err) {
      // Operation fails because 'name' is required
      assert.ok(err);
      assert.ok(err.errors['name']);
    });
```

更新指定的路径只运行验证器

（Update Validators Only Run On Specified Paths）

最后值得注意的一个细节：更新（update）验证器只能运行在 $set 和 $unset 选项。例如，下面的更新将成功，无论数字的值。

```
var testSchema = new Schema({
      number: { type: Number, max: 0 },
    });

    var Test = db.model('Test', testSchema);

    var update = { $inc: { number: 1 } };
    var opts = { runValidators: true };
    Test.update({}, update, opts, function(error) {
      // There will never be a validation error here
    });
```

> 原文：Middleware
> 翻译：小虾米（QQ:509129）

Middleware

中间件（也称为前置和后置钩子）是异步函数执行过程中传递的控制的函数。中间件是在schema级别上指定的，并用于编写插件是非常有用的。Mongoose 4.0 有2种类型的中间件：文档（document）中间件和查询（query）中间件。文档（document）中间件支持以下文档方法。

- init
- validate
- save
- remove

查询（query）中间件支持一下模型和查询方法。

- count
- find
- findOne
- findOneAndRemove
- findOneAndUpdate
- update

文档（document）中间件和查询（query）中间件支持前置和后置钩子。前置和后置钩子如何工作更详细的描述如下。

注：这里没有查询的 `remove()` 钩子，只对文档。如果你设定了一个 'remove'钩子，它将解雇当你调用 `myDoc` 时。 `remove()` ，不是当你调用 `MyModel.remove()` 。

Pre（前置钩子）

有两种类型的前置钩子，串行（serial）和并行（parallel）。

Serial（串行）

串行中间件是一个接一个的执行，当每个中间件调用 `next` 。

```
var schema = new Schema(..);
schema.pre('save', function(next) {
  // do stuff
  next();
});
```

Parallel （并行）

并行中间件提供了更多的细粒度的流量控制。

```
var schema = new Schema(..);

// `true` means this is a parallel middleware. You **must** spec
ify `true`
// as the second parameter if you want to use parallel middlewar
e.
schema.pre('save', true, function(next, done) {
  // calling next kicks off the next middleware in parallel
  next();
  setTimeout(done, 100);
});
```

钩子方法，在这种情况下，保存，将不会被执行，直到完成每个中间件。

使用案例

中间件用于雾化模型逻辑和避免嵌套异步代码块。这里有一些其他的想法：

complex validation removing dependent documents (removing a user removes all his blogposts) asynchronous defaults asynchronous tasks that a certain action triggers triggering custom events notifications

- 杂的验证
- 删除相关文件
  - （删除一个用户删除了他所有的博客文章）
- 异步默认
- 异步任务，某些动作触发器
  - 引发自定义事件

- 通知

错误处理

如果任何中间件调用 `next` 或 `done` 一个类型错误的参数，则流被中断，并且将错误传递给回调。

```
schema.pre('save', function(next) {
  // You **must** do `new Error()`. `next('something went wrong')` will
  // **not** work
  var err = new Error('something went wrong');
  next(err);
});

// later...

myDoc.save(function(err) {
  console.log(err.message) // something went wrong
});
```

后置中间件（Post middleware）

后置中间件被执行后，钩子的方法和所有的前置中间件已经完成。后置的中间件不直接接收的流量控制，如：没有 `next` 或 `done` 回调函数传递给它的。后置钩子是是一种来为这些方法注册传统事件侦听器方式。

```
schema.post('init', function(doc) {
  console.log('%s has been initialized from the db', doc._id);
});
schema.post('validate', function(doc) {
  console.log('%s has been validated (but not saved yet)', doc._
id);
});
schema.post('save', function(doc) {
  console.log('%s has been saved', doc._id);
});
schema.post('remove', function(doc) {
  console.log('%s has been removed', doc._id);
});
```

异步后置钩子

虽然后中间件不接收流量控制，但您仍然可以确保异步后置钩子在预定义的命令中执行。如果你的后置钩子函数至少需要2个参数，mongoose将承担第二个参数是一个 `next()` 函数，以触发序列中的下一个中间件。

```
// Takes 2 parameters: this is an asynchronous post hook
schema.post('save', function(doc, next) {
  setTimeout(function() {
    console.log('post1');
    // Kick off the second post hook
    next();
  }, 10);
});

// Will not execute until the first middleware calls `next()`
schema.post('save', function(doc, next) {
  console.log('post2');
  next();
});
```

保存/验证钩子

`save()` 函数触发 `validate()` 钩子，因为 mongoose 有一个内置的 `pre('save')` 钩子叫 `validate()` 。这意味着所有的 `pre('validate')` 和 `post('validate')` 钩子在任何 `pre('save')` 钩子之前被调用到。

```
schema.pre('validate', function() {
  console.log('this gets printed first');
});
schema.post('validate', function() {
  console.log('this gets printed second');
});
schema.pre('save', function() {
  console.log('this gets printed third');
});
schema.post('save', function() {
  console.log('this gets printed fourth');
});
```

注意 findAndUpdate() and 插件中间件

前置和后置的 `save()` 钩子不能执行在 `update()` ， `findOneAndUpdate()` ，等.。你可以看到一个更详细的讨论这个问题为什么在GitHub。Mongoose 4.0有这些功能不同的钩。

```
schema.pre('find', function() {
  console.log(this instanceof mongoose.Query); // true
  this.start = Date.now();
});

schema.post('find', function(result) {
  console.log(this instanceof mongoose.Query); // true
  // prints returned documents
  console.log('find() returned ' + JSON.stringify(result));
  // prints number of milliseconds the query took
  console.log('find() took ' + (Date.now() - this.start) + ' mil
lis');
});
```

查询中间件不同于文档中间件，在一个微妙但重要的方式：在文档中间件中，这是指被更新的文档。在查询中间件，mongoose并不一定参考被更新的文档，所以这是指查询的对象而不是被更新的文档。

例如，如果你想在每次调用 `update()` 时添加一个 `updatedAt` 的时间戳，你可以使用以下的前置。

```
schema.pre('update', function() {
  this.update({},{ $set: { updatedAt: new Date() } });
});
```

下一步

现在我们已经掌握了中间件，让我们去加入它的查询人口助手一看Mongoose的方法。

> 原文：Population
> 翻译：小虾米（QQ:509129）

Population

在MongoDB没有join联表操作但有时我们还想引用其他集合中文档。这是population的出现的缘由。

Population是在文档中自动更换其他集合的文档指定路径的过程。我们可以填充一个单一的文档，多个文档，普通对象，多个简单的对象，或从查询返回的所有对象。让我们看看一些例子。

```
var mongoose = require('mongoose')
  , Schema = mongoose.Schema

var personSchema = Schema({
  _id     : Number,
  name    : String,
  age     : Number,
  stories : [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

var storySchema = Schema({
  _creator : { type: Number, ref: 'Person' },
  title    : String,
  fans     : [{ type: Number, ref: 'Person' }]
});

var Story  = mongoose.model('Story', storySchema);
var Person = mongoose.model('Person', personSchema);
```

到目前为止，我们已经创建了两个模型。我们的Person 模型有stories字段设置为ObjectIds数组。ref 选项告诉Mongoose哪个模型中中使用 population，在我们的案例 Story 模型。所有的_ids我们存储的必须是 Story 模型的_ids。我们还声明 Story 的_creator属性作为数字型，和 _id类型相同用在personSchema中。这是非常重要的匹配 _id类型到 ref 类型。

> 注：ObjectId，Number，String，和Buffer都是作为refs有效的类型。

Saving refs

引用其他文件保存的工作方式相同，通常保存属性，只是分配的_id值：

```
var aaron = new Person({ _id: 0, name: 'Aaron', age: 100 });

aaron.save(function (err) {
  if (err) return handleError(err);

  var story1 = new Story({
    title: "Once upon a timex.",
    _creator: aaron._id    // assign the _id from the person
  });

  story1.save(function (err) {
    if (err) return handleError(err);
    // thats it!
  });
});
```

Population

到目前为止，我们还没有做任何不同的事情。我们只是创造了一个 Person 和一个 Story 。现在让我们使用查询生成器看一下填充story的_creator。

```
Story
.findOne({ title: 'Once upon a timex.' })
.populate('_creator')
.exec(function (err, story) {
  if (err) return handleError(err);
  console.log('The creator is %s', story._creator.name);
  // prints "The creator is Aaron"
});
```

阵列的refs工作方式相同。在查询时调用populate方法和文档数组将在返回在原来的_ids的地方。

> 注：mongoose > = 3.6 在使用population暴露原始的 `_ids` 通过document#populated()方法。

设置Populated字段

在Mongoose > = 4，你也可以手动填充字段。

```
Story.findOne({ title: 'Once upon a timex.' }, function(error, s
tory) {
  if (error) {
    return handleError(error);
  }
  story._creator = aaron;
  console.log(story._creator.name); // prints "Aaron"
});
```

注意，这只工作在单个refs。你现在不能手动填充数组refs。

字段选择

如果我们只想要一些特定的字段返回填充的文档呢？这可以通过将常用的字段名语法作为填充方法的第二个参数来完成：

```
Story
.findOne({ title: /timex/i })
.populate('_creator', 'name') // only return the Persons name
.exec(function (err, story) {
  if (err) return handleError(err);

  console.log('The creator is %s', story._creator.name);
  // prints "The creator is Aaron"

  console.log('The creators age is %s', story._creator.age);
  // prints "The creators age is null'
})
`
```

Populating multiple paths

如果我们想在同一时间填充多个路径呢？

```
Story
.find(...)
.populate('fans _creator') // space delimited path names
.exec()
```

> 在 mongoose > = 3.6，我们可以把一个空格分隔的路径名来填充字符串。3.6 之前，您必须执行 `populate()` 方法多次。

```
Story
.find(...)
.populate('fans')
.populate('_creator')
.exec()
```

查询条件和其他选项

如果我们想根据他们的年龄来填充我们的球迷数组，选择只是他们的名字，并返回最多，其中5个？

```
Story
.find(...)
.populate({
  path: 'fans',
  match: { age: { $gte: 21 }},
  select: 'name -_id',
  options: { limit: 5 }
})
.exec()
```

Refs to children

我们可能会发现，如果我们使用aaron对象，我们无法得到一个列表的stories。这是因为没有story的对象都“推”到 `aaron.stories` 。

这里有两个观点。首先，很高兴aaron知道哪些stories是他的。

```
aaron.stories.push(story1);
aaron.save(callback);
```

这使我们能够执行一个查找和填充组合：

```
Person
.findOne({ name: 'Aaron' })
.populate('stories') // only works if we pushed refs to children
.exec(function (err, person) {
  if (err) return handleError(err);
  console.log(person);
})
```

> 这是值得商榷的，我们真的要两套指针作为他们可能不同步。相反，我们可以跳过，直接填充并 `find()` 我们感兴趣的故事。

```
Story
.find({ _creator: aaron._id })
.exec(function (err, stories) {
  if (err) return handleError(err);
  console.log('The stories are an array: ', stories);
})
```

更新refs

现在我们有一个故事，我们意识到 `_creator` 是错误的。我们可以通过更新refs通过Mongoose的铸件内部任何其他属性一样：

```
var guille = new Person({ name: 'Guillermo' });
guille.save(function (err) {
  if (err) return handleError(err);

  story._creator = guille;
  console.log(story._creator.name);
  // prints "Guillermo" in mongoose >= 3.6
  // see https://github.com/Automattic/mongoose/wiki/3.6-release
-notes

  story.save(function (err) {
    if (err) return handleError(err);

    Story
    .findOne({ title: /timex/i })
    .populate({ path: '_creator', select: 'name' })
    .exec(function (err, story) {
      if (err) return handleError(err);

      console.log('The creator is %s', story._creator.name)
      // prints "The creator is Guillermo"
    })
  })
})
```

> 返回的文档查询population成为功能齐全，可保存文件，除非指定了精益选项。不与子文档混淆。请注意，当调用它的删除方法，因为你会从数据库中删除它，不仅仅是数组。

Populating 一个存在的文档

如果我们有一个现有的mongoose文档并且想要填充的一些它的路径。mongoose >= 3.6支持document#populate()方法。

Populating 多个存在的文档

如果我们有一个或多个mongoose文档，甚至是简单的对象（像mapReduce输出），我们可以让他们使用Model.populate() 方法 mongoose > = 3.6。这是为什么使用 `document#populate()` 和 `query#populate()` 来populate文档。

Populating在多个层面

说你有一个用户模式，跟踪用户的朋友。

```
var userSchema = new Schema({
  name: String,
  friends: [{ type: ObjectId, ref: 'User' }]
});
```

Populate可以让你得到一个用户的朋友的列表，但如果你也想要一个用户的朋友的朋友呢？指定populate选项告诉mongoose来populate所有用户的朋友的朋友的数组：

```
User.
  findOne({ name: 'Val' }).
  populate({
    path: 'friends',
    // Get friends of friends - populate the 'friends' array for
 every friend
    populate: { path: 'friends' }
  });
```

Populating整个数据库

让我们说，你有一个代表事件的模式，和一个代表对话的模式。每个事件都有一个相应的会话线程。

```
var eventSchema = new Schema({
  name: String,
  // The id of the corresponding conversation
  // Notice there's no ref here!
  conversation: ObjectId
});
var conversationSchema = new Schema({
  numMessages: Number
});
```

同时，假设事件和对话都存储在单独的MongoDB实例。

```
var db1 = mongoose.createConnection('localhost:27000/db1');
var db2 = mongoose.createConnection('localhost:27001/db2');

var Event = db1.model('Event', eventSchema);
var Conversation = db2.model('Conversation', conversationSchema)
;
```

在这种情况下，您将无法正常 `populate()` 。conversation字段永远是空的，因为 `populate()` 不知道使用哪种模式。然而，您可以显式指定模型。

```
Event.
  find().
  populate({ path: 'conversation', model: Conversation }).
  exec(function(error, docs) { /* ... */ });
```

这是一个被称为“跨数据库的populate“，因为它使你populate在MongoDB数据库和通过MongoDB实例。

动态参考

Mongoose也可以同是populate从多个集合。让我们说，你有一个用户模式，有一个“连接”的数组-用户可以连接到其他用户或组织。

```
var userSchema = new Schema({
  name: String,
  connections: [{
    kind: String,
    item: { type: ObjectId, refPath: 'connections.kind' }
  }]
});

var organizationSchema = new Schema({ name: String, kind: String
 });

var User = mongoose.model('User', userSchema);
var Organization = mongoose.model('Organization', organizationSc
hema);
```

以上的refpath属性意味着mongoose会看 `connections.kind` 路径确定模型用于 `populate()` 。换句话说，这 `refpath` 属性使您能够ref属性的动态。

```
// Say we have one organization:
// `{ _id: ObjectId('000000000000000000000001'), name: "Guns N'
Roses", kind: 'Band' }`
// And two users:
// {
//   _id: ObjectId('000000000000000000000002')
//   name: 'Axl Rose',
//   connections: [
//     { kind: 'User', item: ObjectId('000000000000000000000003'
) },
//     { kind: 'Organization', item: ObjectId('00000000000000000
0000001') }
//   ]
// },
// {
//   _id: ObjectId('000000000000000000000003')
//   name: 'Slash',
//   connections: []
// }
User.
  findOne({ name: 'Axl Rose' }).
  populate('connections.item').
  exec(function(error, doc) {
    // doc.connections[0].item is a User doc
    // doc.connections[1].item is an Organization doc
  });
```

Populate虚函数

新的4.5.0中

到目前为止，你只有稀少的基于_id字段。然而，这有时不是正确的选择。特别是，数组，无束缚成长是MongoDB的反模式。用mongoose鼬虚函数，您可以定义文档之间更复杂的关系。

```
var PersonSchema = new Schema({
  name: String,
  band: String
});

var BandSchema = new Schema({
  name: String
});
BandSchema.virtual('members', {
  ref: 'Person', // The model to use
  localField: 'name', // Find people where `localField`
  foreignField: 'band' // is equal to `foreignField`
});

var Person = mongoose.model('Person', personSchema);
var Band = mongoose.model('Band', bandSchema);

/**
 * Suppose you have 2 bands: "Guns N' Roses" and "Motley Crue"
 * And 4 people: "Axl Rose" and "Slash" with "Guns N' Roses", an
d
 * "Vince Neil" and "Nikki Sixx" with "Motley Crue"
 */
Band.find({}).populate('members').exec(function(error, bands) {
  /* `bands.members` is now an array of instances of `Person` */
});

`
```

下一步

现在我们已经掌握了population，让我来看一下连接。

> 原文：Connections
> 翻译：小虾米（QQ:509129）

Connections

我们可以通过利用 `mongoose.connect()` 方法连接MongoDB 。

```
mongoose.connect('mongodb://localhost/myapp');
```

这是最需要的在连接 myapp数据库运行在默认端口（27017）上。如果本地连接失败，然后尝试用127.0.0.1不是localhost。有时可能会出现问题，当本地主机名已更改。

我们还可以视你的环境而定在URI指定几个参数：

```
mongoose.connect('mongodb://username:password@host:port/database
?options...');
```

查看 mongodb connection string spec 了解更多。

选项

该连接方法还接受一个选项对象，该对象将被传递给底层驱动程序。这里包含的所有选项优先于连接字符串中传递的选项。

```
mongoose.connect(uri, options);
```

以下是可用的选项键：

- db - passed to the underlying driver's db instance
- server - passed to the underlying driver's server instance(s)
- replset - passed to the underlying driver's ReplSet instance
- user - username for authentication (if not specified in uri)
- pass - password for authentication (if not specified in uri)
- auth - options for authentication
- mongos - passed to the underlying driver's mongos options

- promiseLibrary - sets the underlying driver's promise library

例子：

```
var options = {
  db: { native_parser: true },
  server: { poolSize: 5 },
  replset: { rs_name: 'myReplicaSetName' },
  user: 'myUserName',
  pass: 'myPassword'
}
mongoose.connect(uri, options);
```

注意：服务器选项auto_reconnect默认为真的可以重写。数据库选项 `forceserverobjectid` 设置为false将不能被重写。

有关可用选项的更多信息，见driver。

注意：如果auto_reconnect设置成on，mongoose会放弃试图恢复一定数量的失败后。设置 `server.reconnectTries` 和 `server.reconnectInterval options` 选项增加mongoose尝试重新连接的次数。

```
// Good way to make sure mongoose never stops trying to reconnect

mongoose.connect(uri, { server: { reconnectTries: Number.MAX_VAL
UE } });
```

连接字符串选项

mongoose支持以下的连接字符串选项。

- ssl
- poolSize
- autoReconnect
- socketTimeoutMS
- connectTimeoutMS
- authSource

- retries
- reconnectWait
- rs_name
- replicaSet
- nativeParser
- w
- journal
- wtimeoutMS
- readPreference
- readPreferenceTags
- sslValidate

关于keepAlive

> 对于长时间运行的应用程序，它往往是谨慎开启keepAlive数毫秒。没有它，在一段时间后，你可能会开始看到“连接关闭”的错误，似乎没有理由。如果是这样的话，读了这些之后，你可能会决定启用KeepAlive：

```
options.server.socketOptions = options.replset.socketOptions = {
 keepAlive: 120 };
mongoose.connect(uri, options);
```

复制集连接

用同样方法连接到一个复制集但是通过逗号分隔uris的列表。

```
mongoose.connect('mongodb://[username:password@]host1[:port1][,h
ost2[:port2],...[,hostN[:portN]]][/[database][?options]]' [, opt
ions]);
```

多mongos的支持

高可用性在多mongoss情况下也支持。通过你的mongos实例的连接字符串和设置mongos选项为true：

```
mongoose.connect('mongodb://mongosA:27501,mongosB:27501', { mong
os: true }, cb);
```

多个连接

到目前为止，我们已经看到了如何连接到使用Mongoose默认连接MongoDB。有时我们可能需要多个连接到Mongo，各有不同的读/写设置，或者只是不同的数据库为例。在这些情况下，我们可以利用 `mongoose.createConnection()` 接受所有已经讨论的争论并返回你一个新的连接。

```
var conn = mongoose.createConnection('mongodb://[username:passwo
rd@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database
][?options]]', options);
```

此连接对象，然后用于创建和检索模型。模型总是局限于单个连接。

连接池

每个连接，使用 `mongoose.connect` 或 `mongoose.createconnection` 创造所有的内部配置连接池的默认大小为5的支持 。使用您的连接选项调整池大小：

```
// single server
var uri = 'mongodb://localhost/test';
mongoose.createConnection(uri, { server: { poolSize: 4 }});

// for a replica set
mongoose.createConnection(uri, { replset: { poolSize: 4 }});

// passing the option in the URI works with single or replica se
ts
var uri = 'mongodb://localhost/test?poolSize=4';
mongoose.createConnection(uri);
```

下一步

现在我们已经掌握了connections，让我们看看我们如何能将我们的功能的碎片变成可重用的并共享插件。

> 原文：Plugins
> 翻译：小虾米（QQ:509129）

Plugins

Schemas是可插拔的，那就是，他们允许使用预包装的能力，扩展他们的功能。这是一个非常强大的功能。

假设我们有几个集合在我们的数据库中，并要为每一个添加最后修改的功能。有了插件，这是很容易。只需要创建一个插件一次，并将其应用到每个Schemas：

```
// lastMod.js
module.exports = exports = function lastModifiedPlugin (schema, options) {
  schema.add({ lastMod: Date })

  schema.pre('save', function (next) {
    this.lastMod = new Date
    next()
  })

  if (options && options.index) {
    schema.path('lastMod').index(options.index)
  }
}

// game-schema.js
var lastMod = require('./lastMod');
var Game = new Schema({ ... });
Game.plugin(lastMod, { index: true });

// player-schema.js
var lastMod = require('./lastMod');
var Player = new Schema({ ... });
Player.plugin(lastMod);
```

我们刚刚添加了最后修改的行为对我们的 `Game` 和 `Player` 的schemas，并声明了一个在我们的Games上的lastMod索引。不坏的几行代码。

全局的Plugins

想注册一个插件为所有schemas吗？mongoose单独有一个 `plugin()` 功能为每一个schema注册插件。例如:

```
var mongoose = require('mongoose');
mongoose.plugin(require('./lastMod'));

var gameSchema = new Schema({ ... });
var playerSchema = new Schema({ ... });
// `lastModifiedPlugin` gets attached to both schemas
var Game = mongoose.model('Game', gameSchema);
var Player = mongoose.model('Player', playerSchema);
```

社区！

你不仅可以在自己的项目架构功能重复使用也可以从Mongoose社区中获利。任何插件发布到NPM，打上mongoose的标签会在我们的搜索结果页面显示。

下一步

现在我们已经掌握了插件和如何参与到伟大的社会成长周围，让我们来看看如何可以帮助贡献Mongoose本身的不断发展。

> 原文：Built-in Promises
> 翻译：小虾米（QQ:509129）

Built-in Promises

Mongoose异步操作，像 `.save()` 和查询，返回Promises/A+ conformant promises。这意味着你可以像 `MyModel.findOne({}).then()` 和 `MyModel.findOne({}).exec()` （如果你使用co包）。

为了向后兼容，Mongoose 4返回默认mpromise承诺。

```
var gnr = new Band({
    name: "Guns N' Roses",
    members: ['Axl', 'Slash']
  });

  var promise = gnr.save();
  assert.ok(promise instanceof require('mpromise'));

  promise.then(function (doc) {
    assert.equal(doc.name, "Guns N' Roses");
  });
```

Queries are not promises

Mongoose查询不是承诺。然而，他们也有一个 `.then()` 功能为产量和异步/等待。如果你需要一个完全成熟的承诺，用 `.exec()` 功能。

```
    var query = Band.findOne({name: "Guns N' Roses"});
    assert.ok(!(query instanceof require('mpromise')));

    // A query is not a fully-fledged promise, but it does have
a `.then()`.
    query.then(function (doc) {
      // use doc
    });

    // `.exec()` gives you a fully-fledged promise
    var promise = query.exec();
    assert.ok(promise instanceof require('mpromise'));

    promise.then(function (doc) {
      // use doc
    });
```

Plugging in your own Promises Library

> Mongoose 4.1.0 新特性

而mpromise满足基本使用的情况下，高级用户可能想插上自己喜欢的ES6风格承诺类库像bluebird，或只使用本地的ES6承诺。只需要设置 `mongoose.Promise` 到你喜欢的ES6风格承诺构造函数，mongoose会使用它。

随着本土承诺ES6Mongoose试验，bluebird和q。任何承诺类库，理论上导出一个ES6风格的构造函数应该工作，但理论往往不同于实践。如果你发现一个bug，开放式问题在GitHub上。

```
  var query = Band.findOne({name: "Guns N' Roses"});

    // Use native promises
    mongoose.Promise = global.Promise;
    assert.equal(query.exec().constructor, global.Promise);

    // Use bluebird
    mongoose.Promise = require('bluebird');
    assert.equal(query.exec().constructor, require('bluebird'));

    // Use q. Note that you **must** use `require('q').Promise`.
    mongoose.Promise = require('q').Promise;
    assert.ok(query.exec() instanceof require('q').makePromise);
```

Promises for the MongoDB Driver

mongoose。承诺的属性集的mongoose用承诺。然而，这并不影响底层的MongoDB的驱动程序;。如果你使用了底层的驱动程序，例如 `Model.collection.db.insert()` ，你需要做一些额外的工作来改变其所承诺的类库。注意，下面的代码假定mongoose> = 4.4.4。

```
    var uri = 'mongodb://localhost:27017/mongoose_test';
    // Use bluebird
    var options = { promiseLibrary: require('bluebird') };
    var db = mongoose.createConnection(uri, options);

    Band = db.model('band-promises', { name: String });

    db.on('open', function() {
      assert.equal(Band.collection.findOne().constructor, require
('bluebird'));
    });
```

> 原文：The model.discriminator() function
> 翻译：小虾米（QQ:509129）

The `model.discriminator()` function

鉴别器是一个模式继承机制。他们使你重叠模式上同一标的MongoDB集合有多个模型。

假设你想在一个集合中跟踪不同类型的事件。每一件事件都会有一个时间戳，但事件表示点击链接应该有一个URL。你可以使用 `model.discriminator()` 函数。这个函数需要2个参数，一个模型的名字和一个鉴频器模式。它返回一个模型是基础模式的结合和鉴频器模式。

```
 var options = {discriminatorKey: 'kind'};

    var eventSchema = new mongoose.Schema({time: Date}, options)
;
    var Event = mongoose.model('Event', eventSchema);

    // ClickedLinkEvent is a special type of Event that has
    // a URL.
    var ClickedLinkEvent = Event.discriminator('ClickedLink',
      new mongoose.Schema({url: String}, options));

    // When you create a generic event, it can't have a URL fiel
d...
    var genericEvent = new Event({time: Date.now(), url: 'google
.com'});
    assert.ok(!genericEvent.url);

    // But a ClickedLinkEvent can
    var clickedEvent =
      new ClickedLinkEvent({time: Date.now(), url: 'google.com'}
);
    assert.ok(clickedEvent.url);
```

鉴别器保存事件模型的集合

假设你创建一个鉴别器跟踪事件，新用户注册。这些 `SignedUpEvent` 实例将被存储在相同的集合作为通用的事件和 `ClickedLinkEvent` 实例。

```
var event1 = new Event({time: Date.now()});
    var event2 = new ClickedLinkEvent({time: Date.now(), url: 'g
oogle.com'});
    var event3 = new SignedUpEvent({time: Date.now(), user: 'tes
tuser'});

    var save = function (doc, callback) {
      doc.save(function (error, doc) {
        callback(error, doc);
      });
    };

    async.map([event1, event2, event3], save, function (error) {

      Event.count({}, function (error, count) {
        assert.equal(count, 3);
      });
    });
```

鉴别器的键

mongoose讲述不同的鉴别模型之间的差异是由“鉴频器的键”，默认是 `__t` 。Mongoose添加一个叫做 `__t` 字符串路径到你的模式中，它采用追踪鉴别本文档实例。

```
var event1 = new Event({time: Date.now()});
    var event2 = new ClickedLinkEvent({time: Date.now(), url: 'g
oogle.com'});
    var event3 = new SignedUpEvent({time: Date.now(), user: 'tes
tuser'});

    assert.ok(!event1.__t);
    assert.equal(event2.__t, 'ClickedLink');
    assert.equal(event3.__t, 'SignedUp');
```

鉴别器添加鉴别键查询

鉴别器模型是特殊的；他们重视鉴别键查询。换句话说，`find(), count(), aggregate(),` 等等，有足够的智慧来解释鉴别器。

```
 var event1 = new Event({time: Date.now()});
    var event2 = new ClickedLinkEvent({time: Date.now(), url: 'g
oogle.com'});
    var event3 = new SignedUpEvent({time: Date.now(), user: 'tes
tuser'});

    var save = function (doc, callback) {
      doc.save(function (error, doc) {
        callback(error, doc);
      });
    };

    async.map([event1, event2, event3], save, function (error) {

      ClickedLinkEvent.find({}, function (error, docs) {
        assert.equal(docs.length, 1);
        assert.equal(docs[0]._id.toString(), event2._id.toString
());
        assert.equal(docs[0].url, 'google.com');
      });
    });
```

鉴别器复制的前置和后置钩子

作者也使用他们的基础模式的前置和后置的中间件。然而，你也可以把中间件来鉴别模式不影响基础模式。

```
 var options = {discriminatorKey: 'kind'};

    var eventSchema = new mongoose.Schema({time: Date}, options)
;
    var eventSchemaCalls = 0;
    eventSchema.pre('validate', function (next) {
      ++eventSchemaCalls;
      next();
    });
    var Event = mongoose.model('GenericEvent', eventSchema);

    var clickedLinkSchema = new mongoose.Schema({url: String}, o
ptions);
    var clickedSchemaCalls = 0;
    clickedLinkSchema.pre('validate', function (next) {
      ++clickedSchemaCalls;
      next();
    });
    var ClickedLinkEvent = Event.discriminator('ClickedLinkEvent'
,
      clickedLinkSchema);

    var event1 = new ClickedLinkEvent();
    event1.validate(function () {
      assert.equal(eventSchemaCalls, 1);
      assert.equal(clickedSchemaCalls, 1);

      var generic = new Event();
      generic.validate(function () {
        assert.equal(eventSchemaCalls, 2);
        assert.equal(clickedSchemaCalls, 1);
      });
    });
```

处理自定义_id字段

鉴别器的字段是基础模式的字段和鉴别器模式的字段的结合，并且鉴频器模式的字段优先。这种行为变得古怪当你有一个自定义 `_id` 字段。一个模式默认情况下得有 `_id` 字段，所以基础模式的 `_id` 字段将得到由鉴别器模式的默认 `_id` 字段覆盖。

你可以通过设置 `_id` 选项为false 在鉴别器的模式如下图所示。

```
var options = {discriminatorKey: 'kind'};

    // Base schema has a String _id...
    var eventSchema = new mongoose.Schema({_id: String, time: Da
te},
      options);
    var Event = mongoose.model('BaseEvent', eventSchema);

    var clickedLinkSchema = new mongoose.Schema({url: String}, o
ptions);
    var ClickedLinkEvent = Event.discriminator('ChildEventBad',
      clickedLinkSchema);

    var event1 = new ClickedLinkEvent();
    // Woops, clickedLinkSchema overwrote the custom _id
    assert.ok(event1._id instanceof mongoose.Types.ObjectId);

    // But if you set `_id` option to false...
    clickedLinkSchema = new mongoose.Schema({url: String},
      {discriminatorKey: 'kind', _id: false});
    ClickedLinkEvent = Event.discriminator('ChildEventGood',
      clickedLinkSchema);

    // The custom _id from the base schema comes through
    var event2 = new ClickedLinkEvent({_id: 'test'});
    assert.ok(event2._id.toString() === event2._id);
```

贡献

> 原文：Documents in ES6
> 翻译：小虾米（QQ:509129）
> 进度：未完成

文档在 ES6

Asynchronous document functions return promises, and so are compatible with the ES6 yield keyword and libraries like co.

Note that the yield keyword is currently only supported in NodeJS 0.11.x with the --harmony flag.

validate()结合CO和产量的关键词

```
co(function*() {
      var schema = null;
      var called = false;
      var shouldSucceed = true;
      var error;

      var validate = {
        validator: function() {
          called = true;
          return shouldSucceed;
        },
        message: 'BAM'
      };

      schema = new Schema({
        eggs: {type: String, required: true, validate: validate}
,
        bacon: {type: Boolean, required: true}
      });

      var M = db.model('validateSchema', schema, getCollectionNa
me());
      var m = new M({eggs: 'Sunny side up', bacon: false});
```

```
      try {
        yield m.validate();
      } catch (e) {
        error = e;
      }

      assert.ok(!error);
      assert.equal(called, true);
      called = false;

      // The validator function above should now fail
      shouldSucceed = false;
      try {
        yield m.validate();
      } catch (e) {
        error = e;
      }

      assert.ok(error);
      assert.ok(error instanceof ValidationError);

      done();
    })();
```

save() integrates with co and the yield keyword

```
co(function*() {
      var error;
      var schema = new Schema({
        description: {type: String, required: true}
      });

      var Breakfast = db.model('breakfast', schema, getCollectio
nName());

      var goodBreakfast = new Breakfast({description: 'eggs & ba
con'});

      try {
```

```
        yield goodBreakfast.save();
      } catch (e) {
        error = e;
      }

      assert.ifError(error);
      var result;
      try {
        result = yield Breakfast.findOne().exec();
      } catch (e) {
        error = e;
      }
      assert.ifError(error);
      assert.equal(result.description, 'eggs & bacon');

      // Should cause a validation error because `description` i
s required
      var badBreakfast = new Breakfast({});
      try {
        yield badBreakfast.save();
      } catch (e) {
        error = e;
      }

      assert.ok(error);
      assert.ok(error instanceof ValidationError);

      done();
    })();
```

populate() requires execPopulate() to work with the yield keyword

```
/**
     *  Because the `populate()` function supports chaining, it'
s difficult
     *  to determine when the chain is 'done'. Therefore, you ne
ed to call
     *  `execPopulate()` to use `populate()` with `yield`.
```

```
     */
    co(function*() {
      var error;
      var breakfastCollectionName = getCollectionName();
      var foodCollectionName = getCollectionName();
      var breakfastSchema = new Schema({
        foods: [{type: mongoose.Schema.ObjectId, ref: foodCollec
tionName}]
      });

      var foodSchema = new Schema({
        name: String
      });

      var Food = db.model(foodCollectionName, foodSchema, foodCo
llectionName);
      var Breakfast = db.model(breakfastCollectionName, breakfas
tSchema, breakfastCollectionName);

      var bacon = new Food({name: 'bacon'});
      var eggs = new Food({name: 'eggs'});
      var goodBreakfast = new Breakfast({foods: [bacon, eggs]});

      try {
        yield [bacon.save(), eggs.save(), goodBreakfast.save()];
      } catch (e) {
        error = e;
      }

      var result;
      try {
        result = yield Breakfast.findOne().exec();
      } catch (e) {
        error = e;
      }
      assert.ifError(error);
      assert.equal(result.foods.length, 2);

      try {
        result = yield result.populate('foods').execPopulate();
```

```
    } catch (e) {
      error = e;
    }
    assert.ifError(error);
    assert.equal(result.foods.length, 2);
    assert.equal(result.foods[0].name, 'bacon');
    assert.equal(result.foods[1].name, 'eggs');

    done();
  })();
```

update() works with co and yield

```
 co(function*() {
      var schema = new Schema({
        steak: String,
        eggs: String
      });

      var Breakfast = db.model('breakfast', schema, getCollectio
nName());

      var breakfast = new Breakfast({});
      var error;

      try {
        yield breakfast.update({steak: 'Ribeye', eggs: 'Scramble
d'}, {upsert: true}).exec();
      } catch (e) {
        error = e;
      }

      assert.ifError(error);
      var result;
      try {
        result = yield Breakfast.findOne().exec();
      } catch (e) {
        error = e;
      }
      assert.ifError(error);
      assert.equal(breakfast._id.toString(), result._id.toString
());
      assert.equal(result.steak, 'Ribeye');
      assert.equal(result.eggs, 'Scrambled');
      done();
    })();
```

Queries in ES6

Mongoose queries' .exec() function returns a promise, and so its compatible with the ES6 yield keyword and libraries like co.

Note that the yield keyword is currently only supported in NodeJS 0.11.x with the `--harmony` flag.

exec() integrates with co and the yield keyword

```
co(function*() {
      var schema = new Schema({
        eggs: {type: Number, required: true},
        bacon: {type: Number, required: true}
      });

      var Breakfast = db.model('BreakfastHarmony', schema, getCo
llectionName());

      try {
        yield Breakfast.create(
          {eggs: 4, bacon: 2},
          {eggs: 3, bacon: 3},
          {eggs: 2, bacon: 4});
      } catch (e) {
        return done(e);
      }

      var result;
      try {
        result = yield Breakfast.findOne({eggs: 4}).exec();
      } catch (e) {
        return done(e);
      }

      assert.equal(result.bacon, 2);

      var results;
      try {
        results = yield Breakfast.find({eggs: {$gt: 2}}).sort({b
acon: 1}).exec();
      } catch (e) {
        return done(e);
      }
```

```
    assert.equal(results.length, 2);
    assert.equal(results[0].bacon, 2);
    assert.equal(results[1].bacon, 3);

    var count;
    try {
      count = yield Breakfast.count({eggs: {$gt: 2}}).exec();
    } catch (e) {
      return done(e);
    }

    assert.equal(count, 2);

    done();
  })();
```

can call populate() with exec()

```
co(function*() {
      var bookSchema = new Schema({
        author: {type: mongoose.Schema.ObjectId, ref: 'AuthorHar
mony'},
        title: String
      });

      var authorSchema = new Schema({
        name: String
      });

      var Book = db.model('BookHarmony', bookSchema, getCollecti
onName());
      var Author = db.model('AuthorHarmony', authorSchema, getCo
llectionName());

      try {
        var hugo = yield Author.create({name: 'Victor Hugo'});
        yield Book.create({author: hugo._id, title: 'Les Miserab
les'});
      } catch (e) {
        return done(e);
      }

      var result;
      try {
        result = yield Book.findOne({title: 'Les Miserables'}).p
opulate('author').exec();
      } catch (e) {
        return done(e);
      }

      assert.equal(result.author.name, 'Victor Hugo');

      done();
    })();
```

Models in ES6

Asynchronous functions on Model return promises, and so are compatible with the ES6 yield keyword and libraries like co.

Note that the functions find(), findOne(), findById(), count(), findOneAndUpdate(), remove(), distinct(), findByIdAndUpdate(), findOneAndRemove(), update(), and findByIdAndRemove() return Query objects, and so you need to use .exec() to use these functions with yield as described above.

create() integrates with co and the yield keyword

```
    co(function * () {
      var schema = new Schema({
        eggs: {type: String, required: true},
        bacon: {type: Boolean, required: true}
      });

      var M = db.model('harmonyCreate', schema, getCollectionNam
e());

      var results;
      try {
        results = yield M.create([
          {eggs: 'sunny-side up', bacon: false},
          {eggs: 'scrambled', bacon: true}]);
      } catch (e) {
        return done(e);
      }

      assert.equal(results.length, 2);
      assert.equal(results[0].eggs, 'sunny-side up');
      assert.equal(results[1].eggs, 'scrambled');

      done();
    })();
```

aggregate() integrates with co and the yield keyword

```
 co(function*() {
      var schema = new Schema({
        eggs: {type: String, required: true},
        bacon: {type: Boolean, required: true}
      });

      var M = db.model('harmonyAggregate', schema, getCollection
Name());

      try {
        yield M.create([
          {eggs: 'sunny-side up', bacon: false},
          {eggs: 'scrambled', bacon: true}]);
      } catch (e) {
        return done(e);
      }

      var results;
      try {
        results = yield M.aggregate([
          {$group: {_id: '$bacon', eggs: {$first: '$eggs'}}},
          {$sort: {_id: 1}}
        ]).exec();
      } catch (e) {
        return done(e);
      }

      assert.equal(results.length, 2);
      assert.equal(results[0]._id, false);
      assert.equal(results[0].eggs, 'sunny-side up');
      assert.equal(results[1]._id, true);
      assert.equal(results[1].eggs, 'scrambled');

      done();
    })();
```

mapReduce() can also be used with co and yield

```
 co(function*() {
      var schema = new Schema({
        eggs: {type: String, required: true},
        bacon: {type: Boolean, required: true}
      });

      var M = db.model('harmonyMapreduce', schema, getCollection
Name());

      try {
        yield M.create([
          {eggs: 'sunny-side up', bacon: false},
          {eggs: 'sunny-side up', bacon: true},
          {eggs: 'scrambled', bacon: true}]);
      } catch (e) {
        return done(e);
      }

      var results;
      try {
        results = yield M.mapReduce({
          map: function() { emit(this.eggs, 1); },
          reduce: function(k, vals) { return vals.length; }
        });
      } catch (e) {
        return done(e);
      }

      assert.equal(results.length, 2);
      assert.ok(results[0]._id === 'sunny-side up' || results[1]
._id === 'sunny-side up');
      assert.ok(results[0]._id === 'scrambled' || results[1]._id
 === 'scrambled');

      done();
    })();
```

> 原文：Mongoose in the browser
> 翻译：小虾米（QQ:509129）

浏览器中的Mongoose

在3.9.3，Mongoose模式的声明是同构的，那就是，你可以用mongoose的浏览器组件来验证对象在浏览器中对你的mongoose模式。

包括mongoose在你的浏览器代码，你可以使用 `require('mongoose')` 如果你使用的是Browserify或可通过script标签引入的mongoose。下面的例子使用mongoose的Amazon的CloudFront CDN。

```
<script type="text/javascript" src="//d1l4stvdmqmdzl.cloudfront.
net/4.0.2/mongoose.js">
</script>
```

浏览器中声明模式

When you include the mongoose.js file in a script tag, mongoose will attach a mongoose object to the global window. This object includes a Schema constructor that you can use to define schemas much like in NodeJS.

当你有一个脚本标签的mongoose.js文件，mongoose会附上mongoose对象全局窗口。此对象包含模式的构造函数，您可以使用来定义模式就像在NodeJS。

注：Mongoose的浏览器组件需要一个ECMAScript 5标准的浏览器。特别是，它不会在Internet Explorer 8或Safari 5的工作。

允许你使用mongoose类型

```
var foodSchema = new mongoose.Schema({name: String});
      var breakfastSchema = new mongoose.Schema({
        foods: [foodSchema],
        date: {type: Date, default: Date.now}
      });

      assert.ok(foodSchema.path('name') instanceof mongoose.Sche
ma.Types.String);
      assert.ok(breakfastSchema.path('foods') instanceof mongoos
e.Schema.Types.DocumentArray);
      assert.ok(breakfastSchema.path('date') instanceof mongoose
.Schema.Types.Date);
```

在浏览器中验证文档

mongoose的浏览器组件的主要目的是验证对一个给定的模式验证文档。因为mongoose浏览器组件目前不支持任何形式的查询，你负责创建自己的文档。

允许您创建一个模式并使用它来验证文档

```
  var schema = new mongoose.Schema({
        name: {type: String, required: true},
        quest: {type: String, match: /Holy Grail/i, required: tr
ue},
        favoriteColor: {type: String, enum: ['Red', 'Blue'], req
uired: true}
      });

      /* `mongoose.Document` is different in the browser than in
 NodeJS.
       * the constructor takes an initial state and a schema. Yo
u can
       * then modify the document and call `validate()` to make
sure it
       * passes validation rules. */
      var doc = new mongoose.Document({}, schema);
      doc.validate(function(error) {
        assert.ok(error);
```

```
        assert.equal('Path `name` is required.', error.errors['n
ame'].message);
        assert.equal('Path `quest` is required.', error.errors['
quest'].message);
        assert.equal('Path `favoriteColor` is required.',
          error.errors['favoriteColor'].message);

        doc.name = 'Sir Lancelot of Camelot';
        doc.quest = 'To seek the holy grail';
        doc.favoriteColor = 'Blue';
        doc.validate(function(error) {
          assert.ifError(error);

          doc.name = 'Sir Galahad of Camelot';
          doc.quest = 'I seek the grail'; // Invalid, must conta
in 'holy grail'
          doc.favoriteColor = 'Yellow'; // Invalid, not 'Red' or
 'Blue'
          doc.validate(function(error) {
            assert.ok(error);
            assert.ok(!error.errors['name']);
            assert.equal('Path `quest` is invalid (I seek the gr
ail).',
              error.errors['quest'].message);
            assert.equal('`Yellow` is not a valid enum value for
 path `favoriteColor`.',
              error.errors['favoriteColor'].message);
            done();
          });
        });
      });
```

> 原文：Creating a Basic Custom Schema Type
> 翻译：小虾米（QQ:509129）

创建一个基本的自定义模式类型

在Mongoose 4.4.0的新特性：Mongoose支持自定义类型。在你到达一个自定义的类型之前，然而，知道一个自定义类型是大多数情况下矫枉过正。你可以用最基本的任务采用自定义的 `getters/setters` ，虚函数，和单一的嵌入式文档。

让我们看看一个基本模式类型例子：一个字节的整数。创建一个新的模式类型，你需要继承 `mongoose.SchemaType` 和添加相应的属性
到 `mongoose.Schema.Types` 。你需要实现一个方法是 `cast()` 方法。

```
function Int8(key, options) {
      mongoose.SchemaType.call(this, key, options, 'Int8');
    }
    Int8.prototype = Object.create(mongoose.SchemaType.prototype
);

    // `cast()` takes a parameter that can be anything. You need
 to
    // validate the provided `val` and throw a `CastError` if you

    // can't convert it.
    Int8.prototype.cast = function(val) {
      var _val = Number(val);
      if (isNaN(_val)) {
        throw new Error('Int8: ' + val + ' is not a number');
      }
      _val = Math.round(_val);
      if (_val < -0x80 || _val > 0x7F) {
        throw new Error('Int8: ' + val +
          ' is outside of the range of valid 8-bit ints');
      }
      return _val;
    };

    // Don't forget to add `Int8` to the type registry
    mongoose.Schema.Types.Int8 = Int8;
```

```
    var testSchema = new Schema({ test: Int8 });
    var Test = mongoose.model('Test', testSchema);

    var t = new Test();
    t.test = 'abc';
    assert.ok(t.validateSync());
    assert.equal(t.validateSync().errors['test'].name, 'CastErro
r');
    assert.equal(t.validateSync().errors['test'].message,
      'Cast to Int8 failed for value "abc" at path "test"');
    assert.equal(t.validateSync().errors['test'].reason.message,
      'Int8: abc is not a number');
```

MongoDB服务器版本的兼容性

Mongoose依赖MongoDB Node.js驱动和MongoDB的交流。你可以参考这个表的最新信息以MongoDB驱动程序支持哪个版本的MongoDB版本。

下面是semver范围代表哪个版本的mongoose与MongoDB服务器版本兼容。

- MongoDB Server 2.4.x: mongoose ~3.8, 4.x
- MongoDB Server 2.6.x: mongoose ~3.8.8, 4.x
- MongoDB Server 3.0.x: mongoose ~3.8.22, 4.x
- MongoDB Server 3.2.x: mongoose >=4.3.0

3.6 发布说明

3.8 发布说明

4.0 发布说明