第 1 章

简介

MongoDB 是一种强大、灵活、可扩展的数据存储方式。它扩展了关系型数据库的众多有用功能，如辅助索引、范围查询（range query）和排序。MongoDB 的功能非常丰富，比如内置的对 MapReduce 式聚合的支持，以及对地理空间索引的支持。

要是不能用的话，再牛的技术也是空谈，MongoDB 致力于容易上手、便于使用。MongoDB 的数据模型对开发者来说非常友好，配置选项对手管理员来说也很轻松，并且由驱动程序和数据库 shell 提供的自然语言式的 API。MongoDB 会为你扫除障碍，让你关注编程本身而不是为存储数据烦恼。

1.1 丰富的数据模型

MongoDB 是面向文档的数据库，不是关系型数据库。放弃关系模型的主要原因就是为了获得更加方便的扩展性，当然还有其他好处。

基本的思路就是将原来“行”（row）的概念换成更加灵活的“文档”（document）模型。面向文档的方式可以将文档或者数组内嵌进来，所以用一条记录就可以表示非常复杂的层次关系。使用面向对象语言的开发者恰恰这么看待数据，所以感觉非常自然。

MongoDB 没有模式：文档的键不会事先定义也不会固定不变。由于没有模式需要更改，通常不需要迁移大量数据。不必将所有数据都放到一个模子里面，应用层可以处理新增或者丢失的键。这样开发者可以非常容易地变更数据模型。

1.2 容易扩展

应用数据集的大小在飞速增加。传感器技术的发展、带宽的增加，以及可联网手持

设备的普及使得当下即便很小的应用也要存储大量数据，量大到很多数据库都应付不来。T 级别的数据原来是闻所未闻的，现在已经司空见惯了。

由于开发者要存储的数据不断增长，他们面临一个非常困难的选择：该如何扩展他们的数据库？升级呢（买台更好的机器），还是扩展呢（将数据分散到很多机器上）？升级通常是最省力气的做法，但是问题也显而易见：大型机一般都非常昂贵，最后达到了物理极限的话花多少钱都买不到更好的机器。对于大多数人希望构建的大型 Web 应用来说，这样做既不现实也不划算。而扩展就不同了，不但经济而且还能持续添加：想要增加存储空间或者提升性能，只需要买台一般的服务器加入集群就好了。

MongoDB 从最初设计的时候就考虑到了扩展的问题。它所采用的面向文档的数据模型使其可以自动在多台服务器之间分割数据。它还可以平衡集群的数据和负载，自动重排文档。这样开发者就可以专注于编写应用，而不是考虑如何扩展。要是需要更大的容量，只需在集群中添加新机器，然后让数据库来处理剩下的事。

1.3 丰富的功能

很难界定什么才算作一个功能：上面提及的算是功能吗？关系型数据库做不到的算吗？都可以说 Memcached 做不到的呢？其他面向文档的数据库做不到的又如何呢？但无论界定的标准是什么，都可以说 MongoDB 拥有一些真正独特的、好用的工具，其他方案不具备或不完全具备这些工具。

- 索引

  MongoDB 支持通用辅助索引，能进行多种快速查询，也提供唯一的、复合的和地理空间索引能力。

- 存储 JavaScript

  开发人员不必使用存储过程了，可以直接在服务端存取 JavaScript 的函数和值。

- 聚合

  MongoDB 支持 MapReduce 和其他聚合工具。

- 固定集合

  集合的大小是有上限的，这对某些类型的数据（比如日志）特别有用。

- 文件存储

  MongoDB 支持用一种容易使用的协议存储大型文件和文件的元数据。

有些关系型数据库的常见功能 MongoDB 并不具备，比如联接（join）和复杂的多

行事务。这个架构上的考虑是为了提高扩展性，因为这两个功能实在很难在一个分布式系统上实现。

1.4 不牺牲速度

卓越的性能是 MongoDB 的主要目标，也极大地影响了设计上的很多决定。MongoDB 使用 MongoDB 传输协议作为与服务器交互的主要方式（与之对应的协议需要更多的开销，如 HTTP/REST）。它对文档进行动态填充，预分配数据文件，用空间换取性能的稳定。默认的存储引擎中使用了内存映射文件，将内存管理工作交给操作系统去处理。动态查询优化器会“记住”执行查询最高效的方式。总之，MongoDB 在各个方面都充分考虑了性能。

虽然 MongoDB 功能强大，尽量保持关系型数据库的众多特性，但是它并不是要具备所有的关系型数据库的功能。它尽可能地将服务器端处理逻辑交给客户端（由驱动程序或者用户的应用程序处理）。这样精简的设计使得 MongoDB 获得了非常好的性能。

1.5 简便的管理

MongoDB 尽量让服务器自治来简化数据库的管理。除了启动数据库服务器之外，几乎没有什么必要的管理操作。如果主服务器挂掉了，MongoDB 会自动切换到备份服务器上，并且将备份服务器提升为活跃服务器。在分布式环境下，集群只需要知道有新增加的节点，就会自动集成和配置新节点。

MongoDB 的管理理念就是尽可能地让服务器自动配置，让用户能在需要的时候调整设置（但不强制）。

1.6 其他内容

在本书中，我们还会花些时间追溯一下在开发 MongoDB 的过程中一些决定的原因和动机，希望通过这种方式来阐释 MongoDB 的理念。毕竟，MongoDB 的愿景是对自身最好的诠释——建立一种灵活、高效、易于扩展、功能完备的数据库。

第 2 章

入门

MongoDB 非常强大，同时也很容易上手。本章会介绍一些 MongoDB 的基本概念。

- *文档*是 MongoDB 中数据的基本单元，非常类似于关系数据库管理系统中的行（但是比行要复杂得多）。
- 类似地，*集合*可以被看做是没有模式的表。
- MongoDB 的单个实例可以容纳多个独立的*数据库*，每一个都有自己的集合和权限。
- MongoDB 自带简洁但功能强大的 JavaScript shell，这个工具对于管理 MongoDB 实例和操作数据作用非常大。
- 每一个文档都有一个特殊的键 `"_id"`，它在文档所处的集合中是唯一的。

2.1 文档

*文档*是 MongoDB 的核心概念。多个键及其关联的值有序地放置在一起便是文档。每种编程语言表示文档的方法不太一样，但大多数编程语言都有相通的一种数据结构，比如映射、散列或字典。例如，在 JavaScript 里面，文档表示为对象：

```
{"greeting" : "Hello, world!"}
```

这个文档只有一个键 `"greeting"`，其对应的值为 `"Hello, world!"`。绝大多数情况下，文档会比这个简单的例子复杂得多，经常会包含多个键 / 值对：

```
{"greeting" : "Hello, world!", "foo" : 3}
```

这个例子很好地解释了几个十分重要的概念。

- 文档中的键 / 值对是有序的，上面的文档和下面的文档是完全不同的：

```
{"foo" : 3, "greeting" : "Hello, world!"}
```

通常文档中键的顺序并不重要。实际上，有些编程语言默认对文档的呈现根本就不顾忌顺序（如 Python 的字典，Perl 和 Ruby 1.8 中的散列）。这些语言的驱动包含特殊的机制，会在少数必要的情况下指定文档的排序。本书会时常提到这些情况。

- 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档，详见 2.6.5 节）。这个例子中 `"greeting"` 的值是个字符串，而 `"foo"` 的值是个整数。

文档的键是字符串。除了少数例外情况，键可以使用任意 UTF-8 字符。

- 键不能含有 \0（空字符）。这个字符用来表示键的结尾。
- . 和 $ 有特别的意义，只有在特定环境下才能使用，后面的章节会详细说明。通常来说就是被保留了，使用不当的话，驱动程序会提示。
- 以下划线“_”开头的键是保留的，虽然这个并不是严格要求的。

MongoDB 不但区分类型，也区分大小写。例如，下面的两个文档是不同的：

```
{"foo" : 3}
{"foo" : "3"}
```

以下的文档也是不同的：

```
{"foo" : 3}
{"Foo" : 3}
```

还有一个非常重要的事项需要注意，MongoDB 的文档不能有重复的键。例如，下面的文档是非法的：

```
{"greeting" : "Hello, world!", "greeting" : "Hello, MongoDB!"}
```

2.2 集合

集合就是一组文档。如果说 MongoDB 中的文档类似于关系型数据库中的行，那么集合就如同表。

2.2.1 无模式

集合是*无模式*的。这意味着一个集合里面的文档可以是各式各样的。例如，下面两个文档可以存在于同一个集合里面：

```
{"greeting" : "Hello, world!"}
{"foo" : 5}
```

注意，上面的文档不光是值的类型不同（字符串和整数），它们的键也是完全不一样的。因为集合里面可以放置任何文档，随之而来的一个问题是：“还有必要使用多个集合吗？”问得好！要是没必要对各种文档划分模式，那么为什么还要使用多个集合呢？下面是一些理由。

- 把各种各样的文档都混在一个集合里面，无论对于开发者还是管理员来说都是噩梦。开发者要么确保每次查询只返回需要的文档种类，要么让执行查询的应用程序来处理所有不同类型的文档。如果查询博客文章还要剔除那些含有作者数据的文档，就很令人恼火。
- 在一个集合里面查询特定类型的文档在速度上也很不划算，分开做多个集合要快得多。例如，集合里面有个标注类型的键，现在查询其值为“skim”、“whole”或“chunky monkey”的文档，就会非常慢。如果按照名字分割成 3 个集合的话，查询会快很多（参见“子集合”部分）
- 把同种类型的文档放在一个集合里，这样数据会更加集中。从只含有博客文章的集合里面查询几篇文章，会比从含有文章和作者数据的集合里面查出几篇文章少消耗磁盘寻道操作。
- 当创建索引的时候，文档会有附加的结构（尤其是有唯一索引的时候）。索引是按照集合来定义的。把同种类型的文档放入同一个集合里面，可以使索引更加有效。

你已经看到了，的确有很多理由创建一个模式把相关类型的文档规整到一起。但 MongoDB 对此还是不做强制要求，让开发者更有灵活性。

2.2.2 命名

我们可以通过名字来标识集合。集合名可以是满足下列条件的任意 UTF-8 字符串。

- 集合名不能是空字符串 ""。
- 集合名不能含有 \0 字符（空字符），这个字符表示集合名的结尾。
- 集合名不能以“system.”开头，这是为系统集合保留的前缀。例如 system.users 这个集合保存着数据库的用户信息，system.namespaces 集合保存着所有数据库集合的信息。
- 用户创建的集合名字不能含有保留字符 $。有些驱动程序的确支持在集合名里面包含 $，这是因为某些系统生成的集合中包含该字符。除非你要访问这种系统创建的集合，否则千万不要在名字里出现 $。

子集合

组织集合的一种惯例是使用“.”字符分开的按命名空间划分的子集合。例如，一个

带有博客功能的应用可能包含两个集合，分别是 `blog.posts` 和 `blog.authors`。这样做的目的只是为了使组织结构更好些，也就是说 `blog` 这个集合（这里根本就不需要存在）及其子集合没有任何关系。

虽然子集合没有特别的地方，但还是很有用，很多 MongoDB 工具中都包含子集合。

- GridFS 是一种存储大文件的协议，使用子集合来存储文件的元数据，这样就与内容块分开了（关于 GridFS 详见第 7 章）。
- MongoDB 的 Web 控制台通过子集合的方式将数据组织在 DBTOP 部分（关于管理详见第 8 章）。
- 绝大多数驱动程序都提供语法糖，为访问指定集合的子集合提供方便。例如，在数据库 shell 里面，`db.blog` 代表 `blog` 集合，`db.blog.posts` 代表 `blog.posts` 集合。

在 MongoDB 中使用子集合来组织数据是很好的方法，在此强烈推荐。

2.3 数据库

MongoDB 中多个文档组成集合，同样多个集合可以组成*数据库*。一个 MongoDB 实例可以承载多个数据库，它们之间可视为完全独立的。每个数据库都有独立的权限控制，即便是在磁盘上，不同的数据库也放置在不同的文件中。将一个应用的所有数据都存储在同一个数据库中的做法就很好。要想在同一个 MongoDB 服务器上存放多个应用或者用户的数据，就要使用不同的数据库了。

和集合一样，数据库也通过名字来标识。数据库名可以是满足以下条件的任意 UTF-8 字符串。

- 不能是空字符串（""）。
- 不得含有 ' '（空格）、.、$、/、\ 和 \0（空字符）。
- 应全部小写。
- 最多 64 字节。

要记住一点，数据库名最终会变成文件系统里的文件，这也就是有如此多限制的原因。

有一些数据库名是保留的，可以直接访问这些有特殊作用的数据库。这些数据库如下所示。

- admin

  从权限的角度来看，这是“root”数据库。要是将一个用户添加到这个数据库，

这个用户自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。

- local

这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合（关于复制和本地数据库详见第 9 章）

- config

当 Mongo 用于分片设置时（参见第 10 章），config 数据库在内部使用，用于保存分片的相关信息。

把数据库的名字放到集合名前面，得到就是集合的完全限定名，称为命名空间。例如，如果你在 cms 数据库中使用 `blog.posts` 集合，那么这个集合的命名空间就是 `cms.blog.posts`。命名空间的长度不得超过 121 字节，在实际使用当中应该小于 100 字节。关于 MongoDB 中集合的命名空间和内部表示的更多信息，可以参考附录 C

2.4 启动MongoDB

MongoDB 一般作为网络服务来运行，客户端可以连接到该服务器并执行操作。要启动该服务器，需要运行 `mongod` 可执行文件：

```
$ ./mongod
./mongod --help for help and startup options
Sun Mar 28 12:31:20 Mongo DB : starting : pid = 44978 port = 27017
dbpath = /data/db/ master = 0 slave = 0 64-bit
Sun Mar 28 12:31:20 db version v1.5.0-pre-, pdfile version 4.5
Sun Mar 28 12:31:20 git version: ...
Sun Mar 28 12:31:20 sys info: ...
Sun Mar 28 12:31:20 waiting for connections on port 27017
Sun Mar 28 12:31:20 web admin interface listening on port 28017
```

或者在 Windows 下，这样操作：

```
$ mongod.exe
```

关于安装 MongoDB 的详细信息，参见附录 A。

`mongod` 在没有参数的情况下会使用默认数据目录 /data/db（在 Windows 下是 C:\data\db\），并使用 27017 端口。如果数据目录不存在或者不可写，服务器会启动失败。所以在启动 MongoDB 前，创建数据目录（比如 mkdir -p /data/db），并确保

对该目录有写权限是很重要的。如果端口被占用，启动也会失败。通常这是由于MongoDB 实例已经在运行了。

服务器会打印版本和系统信息，然后等待连接。默认情况下，MongoDB 监听 27017 端口。

mongod 还会启动一个非常基本的 HTTP 服务器，监听数字比主端口号高 1000 的端口，也就是 28017 端口。这意味着你可以通过浏览器访问 http://localhost:28017 来获取数据库的管理信息。

在启动服务器的 shell 下可以键入 Ctrl-C 来完全地停止 mongod 的运行。

想要了解启动和停止 MongoDB 的更多细节，请参见 8.1 节；想要了解管理接口的更多内容，可以参 8.2.1 节。

2.5 MongoDB shell

MongoDB 自带一个 JavaScript shell，可以从命令行与 MongoDB 实例交互。这个 shell 非常有用，通过它可以执行管理操作、检查运行实例，亦或做其他尝试。这个 mongo shell 对于使用 MongoDB 来说是至关重要的工具，本书后面也会经常使用这个工具。

2.5.1 运行shell

运行 mongo 启动 shell：

```
$ ./mongo
MongoDB shell version: 1.6.0
url: test
connecting to: test
type "help" for help
>
```

shell 会在启动时自动连接 MongoDB 服务器，所以要确保在使用 shell 之前启动 mongod。

shell 是功能完备的 JavaScript 解释器，可以运行任何 JavaScript 程序。为了证明这一点，我们运行几个简单的数学运算：

```
> x = 200
200
> x / 5;
40
```

还可以充分利用 JavaScript 的标准库。

```
> Math.sin(Math.PI / 2);
1
> new Date("2010/1/1");
"Fri Jan 01 2010 00:00:00 GMT-0500 (EST)"
> "Hello, World!".replace("World", "MongoDB");
Hello, MongoDB!
```

也可以定义和调用 JavaScript 函数：

```
> function factorial (n) {
... if (n <= 1) return 1;
... return n * factorial(n - 1);
... }
> factorial(5);
120
```

注意，可以使用多行命令。这个 shell 会检测输入的 JavaScript 语句是否写完，如没写完还可以在下一行接着写。

2.5.2 MongoDB客户端

虽然能运行任意 JavaScript 程序很酷，但 shell 的真正威力还在于它是一个独立的 MongoDB 客户端。开启的时候，shell 会连到 MongoDB 服务器的 test 数据库，并将这个数据库连接赋值给全局变量 db。这个变量是通过 shell 访问 MongoDB 的主要入口点。

shell 还有些非 JavaScript 语法的扩展，是为了方便习惯于 SQL shell 的用户而添加的。这些扩展并不提供额外的功能，但它们是很棒的语法糖。例如，最重要的操作之一就是选择要使用的数据库：

```
> use foobar
switched to db foobar
```

现在如果看看 db，会发现其指向 `foobar` 数据库：

```
> db
foobar
```

因为这是一个 JavaScript shell，所以键入一个变量会将变量的值转换为字符串（这里就是数据库名）并打印出来。

可以通过 db 变量来访问其中的集合。例如 `db.baz` 返回当前数据库的 `baz` 集合。既然现在可以在 shell 中访问集合，那么基本上就可以执行几乎所有的数据库操作了。

2.5.3 shell中的基本操作

在 shell 查看操作数据会用到 4 个基本操作：创建、读取、更新和删除 (CRUD)。

1. 创建

insert 函数添加一个文档到集合里面。例如，假设要存储一篇博客文章。首先，创建一个局部变量 post，内容是代表文档的 JavaScript 对象。里面会有 "title"，"content" 和 "date"（发表日期）几个键。

```
> post = {"title" : "My Blog Post",
... "content" : "Here's my blog post.",
... "date" : new Date()}
{
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

这个对象是个有效的 MongoDB 文档，所以可以用 insert 方法将其保存到 blog 集合中：

```
> db.blog.insert(post)
```

这篇文章已经被存到数据库里面了。可以调用集合的 find 方法来查看一下：

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

除了我们输入的键 / 值对都完整地被保存下来，还有一个额外添加的键 "_id"。本章的最后会解释 "_id" 突然出现的原因。

2. 读取

find 会返回集合里面所有的文档。若只是想查看一个文档，可以用 findOne：

```
> db.blog.findOne()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
}
```

find 和 findOne 可以接受查询文档形式的限定条件。这将通过查询限制匹配的文档。使用 find 时，shell 自动显示最多 20 个匹配的文档，但可以获取更多文档。关

于查询的更多内容，参见第 4 章。

3. 更新

如果要更改博客文章，就要用到 update 了。update 接受（至少）两个参数：第一个是要更新文档的限定条件，第二个是新的文档。假设决定给我们先前写的文章增加评论内容，则需要增加一个新的键，对应的值是存放评论的数组。

第一步修改变量 post，增加 "comments" 键：

```
> post.comments = []
[ ]
```

然后执行 update 操作，用新版本的文档替换标题为“My Blog Post”的文章：

```
> db.blog.update({title : "My Blog Post"}, post)
```

文档已经有了 "comments" 键。再用 find 查看一下，可以看到新的键：

```
> db.blog.find()
{
    "_id" : ObjectId("4b23c3ca7525f35f94b60a2d"),
    "title" : "My Blog Post",
    "content" : "Here's my blog post.",
    "date" : "Sat Dec 12 2009 11:23:21 GMT-0500 (EST)"
    "comments" : [ ]
}
```

4. 删除

remove 用来从数据库中永久性地删除文档。在不使用参数进行调用的情况下，它会删除一个集合内的所有文档。它也可以接受一个文档以指定限定条件。例如，下面的命令会删除我们刚刚创建的文章：

```
> db.blog.remove({title : "My Blog Post"})
```

集合现在又是空的了。

2.5.4 使用shell的窍门

由于 mongo 是个 JavaScript shell，通过在线查看 JavaScript 的文档能获得很多帮助。shell 本身内置了帮助文档，可以通过 help 命令查看。

```
> help
HELP
    show dbs                show database names
```

```
show collections          show collections in current database
show users                show users in current database
show profile              show recent system.profile entries w. time >= 1ms
use <db name>             set current database to <db name>
db.help()                 help on DB methods
db.foo.help()             help on collection methods
db.foo.find()             list objects in collection foo
db.foo.find( { a : 1 } )  list objects in foo where a == 1
it                        result of the last line evaluated
```

使用 db.help() 可以查看数据库级别的命令的帮助，集合的相关帮助可以通过 db.foo.help() 来查看。

有个了解函数功用的技巧，就是在输入的时候不要输括号。这样就会显示该函数的 JavaScript 源代码。例如，如果想看看 update 的机理，或者就是为了看看参数顺序，可以这么做：

```
> db.foo.update
function (query, obj, upsert, multi) {
    assert(query, "need a query");
    assert(obj, "need an object");
    this._validateObject(obj);
    this._mongo.update(this._fullName, query, obj,
                       upsert ? true : false, multi ? true : false);
    }
```

要查看 shell 提供的所有自动生成的 JavaScript 函数 API 文档，可访问 http://api.mongodb.org/js。

蹩脚的集合名

使用 db.集合名的方式来访问集合一般不会有问题，但如果集合名恰好是数据库类的一个属性就有问题了。例如，要访问 version 这个集合，使用 db.version 就不行，因为 db.version 是个数据库函数（它返回正在运行的 MongoDB 服务器的版本）。

```
> db.version
function () {
    return this.serverBuildInfo().version;
}
```

当 JavaScript 只有在 db 中找不到指定的属性时，才会将其作为集合返回。当有属性与目标集合同名时，可以使用 getCollection 函数：

```
> db.getCollection("version");
test.version
```

要查看名称中含有无效 JavaScript 字符的集合，这个函数也可以派上用场。比如 foo-

bar 是个有效的集合名，但是在 JavaScript 中就变成了变量相减了。通过 db.get-Collection（"foo-bax"）可以得到 foo-bax 集合。

在 JavaScript 中，x.y 与 x['y'] 完全等价。这就意味着不但可以直“呼”其名，也可以使用变量来访问子集合。也就是说，当需要对 blog 的每个子集合执行操作时，只需要像下面这样迭代就好了：

```
var collections = ["posts", "comments", "authors"];

for (i in collections) {
    doStuff(db.blog[collections[i]]);
}
```

而不是使用下面这种笨笨的写法：

```
doStuff(db.blog.posts);
doStuff(db.blog.comments);
doStuff(db.blog.authors);
```

2.6 数据类型

本章的开始讲了一些文档的基本概念，现在大家应该会启动并运行 MongoDB，在 shell 里面动手试一试。这一部分会更加深入一些。MongoDB 支持将多种数据类型作为文档中的值。本节我们就来逐一看看。

2.6.1 基本数据类型

MongoDB 的文档类似于 JSON，在概念上和 JavaScript 中的对象神似。JSON 是一种简单的表示数据的方式，其规范可用一段文字描述（请到 http://www.json.org 自行验证），仅包含 6 种数据类型。这带来很多好处：易于理解、易于解析、易于记忆。但另外一方面，JSON 的表现力也有限制，因为只有 null、布尔、数字、字符串、数组和对象几种类型。

虽然这些类型的表现力已经足够强大，但是对于绝大多数应用来说还需要另外一些不可或缺的类型，尤其是与数据库打交道的那些应用。例如，JSON 没有日期类型，这会使得处理本来简单的日期问题变得非常繁琐。只有一种数字类型，没法区分浮点数和整数，更不能区分 32 位和 64 位数字。也没有办法表示其他常用类型，如正则表达式或函数。

MongoDB 在保留 JSON 基本的键 / 值对特性的基础上，添加了其他一些数据类型。在不同的编程语言下这些类型的表示有些许差异，下面列出了 MongoDB 通常支持的一些类型，同时说明了在 shell 中这些类型是如何表示为文档的一部分的。

- `null`

`null` 用于表示空值或者不存在的字段。

```
{"x" : null}
```

- 布尔

布尔类型有两个值 `'true'` 和 `'false'`：

```
{"x" : true}
```

- 32 位整数

shell 中这个类型不可用。前面提到，JavaScript 仅支持 64 位浮点数，所以 32 位整数会被自动转换。

- 64 位整数

shell 也不支持这个类型。shell 会使用一个特殊的内嵌文档来显示 64 位整数，详见 2.6.2 节。

- 64 位浮点数

shell 中的数字都是这种类型。下面是一个浮点数：

```
{"x" : 3.14}
```

这个也是浮点数：

```
{"x" : 3}
```

- 字符串

UTF-8 字符串都可表示为字符串类型的数据：

```
{"x" : "foobar"}
```

- 符号

shell 不支持这种类型。shell 将数据库里的符号类型转换成字符串。

- 对象 id

对象 id 是文档的 12 字节的唯一 ID。详见 2.6.6 节：

```
{"x" : ObjectId()}
```

- 日期

日期类型存储的是从标准纪元开始的毫秒数。不存储时区：

```
{"x" : new Date()}
```

- 正则表达式

文档中可以包含正则表达式，采用 JavaScript 的正则表达式语法：

```
{"x" : /foobar/i}
```

- 代码

文档中还可以包含 JavaScript 代码：

```
{"x" : function() { /* ... */ }}
```

- 二进制数据

二进制数据可以由任意字节的串组成。不过 shell 中无法使用。

- 最大值

BSON 包括一个特殊类型，表示可能的最大值。shell 中没有这个类型。

- 最小值

BSON 包括一个特殊类型，表示可能的最小值。shell 中没有这个类型。

- 未定义

文档中也可以使用未定义类型（JavaScript 中 `null` 和 `undefined` 是不同的类型）。

```
{"x" : undefined}
```

- 数组

值的集合或者列表可以表示成数组：

```
{"x" : ["a", "b", "c"]}
```

- 内嵌文档

文档可以包含别的文档，也可以作为值嵌入到父文档中：

```
{"x" : {"foo" : "bar"}}
```

2.6.2 数字

JavaScript 中只有一种“数字”类型。因为 MongoDB 中有 3 种数字类型（32 位整数、64 位整数和 64 位浮点数），shell 必须绕过 JavaScript 的限制。默认情况下，shell 中的数字都被 MongoDB 当做是双精度数。这意味着如果你从数据库中获得的是一个 32 位整数，修改文档后，将文档存回数据库的时候，这个整数也被转换成了浮点数，即便保持这个整数原封不动也会这样的。所以明智的做法是尽量不要在 shell 下覆盖整个文档。（关于修改指定键的值的内容参见第 3 章。）

数字只能表示为双精度数（64 位浮点数）的另外一个问题是，有些 64 位的整数并不能

精确地表示为 64 位浮点数。所以，要是存入了一个 64 位整数，然后在 shell 中查看，它会显示一个内嵌文档，表示可能不准确。例如，保存一个文档[1]，其中 "myInteger" 键的值设为一个 64 位整数——3，然后在 shell 中查看，应该是下面这样的：

```
> doc = db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa08"),
    "myInteger" : {
        "floatApprox" : 3
    }
}
```

数据库中的数字是不会改变的（除非你修改了，尔后又通过 shell 保存回去了，这样它就会被转换成浮点类型）；内嵌文档只表示 shell 显示的是一个用 64 位浮点数近似表示的 64 位整数。若是内嵌文档只有一个键的话，实际上这个值是准确的。

要是插入的 64 位整数不能精确地作为双精度数显示，shell 会添加两个键，"top" 和 "bottom"，分别表示高 32 位和低 32 位。例如，如果插入 9 223 372 036 854 775 807，shell 会这样显示：

```
> db.nums.findOne()
{
    "_id" : ObjectId("4c0beecfd096a2580fe6fa09"),
    "myInteger" : {
        "floatApprox" : 9223372036854776000,
        "top" : 2147483647,
        "bottom" : 4294967295
    }
}
```

"floatApprox" 是一种特殊的内嵌文档，可以作为值和文档来操作：

```
> doc.myInteger + 1
4

> doc.myInteger.floatApprox
3
```

32 位的整数都能用 64 位的浮点数精确表示，所以显示起来没什么特别的。

2.6.3 日期

在 JavaScript 中，Date 对象用做 MongoDB 的日期类型，创建一个新的 Date 对象时，通常会调用 new Data(…) 而不只是 Date(…)。调用构造函数（也就是说不包括 new）实际上会返回对日期的字符串表示，而不是真正的 Date 对象。这不是 MongoDB 的特性，而是 JavaScript 本身的特性。要是不小心忘了使用 Date 构造函

译注1：显然不是在shell中保存的。

数，最后就会导致日期和字符串混淆。字符串和日期不能互相匹配，所以这会给删除、更新、查询等很多操作带来问题。

关于 JavaScript 中 `Date` 类的详细说明和可接受的构造函数的格式，请参见 ECMA-Script 规范 15.9 节（可在 http://www.ecmascript.org 下载）。

shell 中的日期显示时使用本地时区设置。但是，日期在数据中是以从标准纪元开始的毫秒数的形式存储的，没有与之相关的时区信息（当然可以把时区信息作为其他键的值存储）。

2.6.4 数组

数组是一组值，既可以作为有序对象（像列表、栈或队列）来操作，也可以作为无序对象（像集合）操作。

在下面的文档中，`"things"` 这个键的值就是一个数组：

```
{"things" : ["pie", 3.14]}
```

从这个例子可以看到，数组可以包含不同数据类型的元素（这个例子中是一个字符串和一个浮点数）。实际上，常规键 / 值对支持的值都可以作为数组的元素，甚至是套嵌数组。

文档中的数组有个奇妙的特性，就是 MongoDB 能“理解”其结构，并知道如何“深入”数组内部对其内容进行操作。这样就能用内容对数组进行查询和构建索引了。例如，之前的例子中，MongoDB 可以查询所有 `"things"` 数组中含有 3.14 的文档。要是经常使用这个查询，可以对 `"things"` 创建索引，来提高性能。

MongoDB 可以使用原子更新修改数组中的内容，比如深入数组内部将 pie 改为 pi。在本书后面还会介绍更多这种操作的例子。

2.6.5 内嵌文档

内嵌文档就是把整个 MongoDB 文档作为另一个文档中键的一个值。这样数据可以组织得更自然些，不用非得存成扁平结构的。

例如，用一个文档来表示一个人，同时还要保存他的地址，可以将地址内嵌到 `"add- ress"` 文档中：

```
{
    "name" : "John Doe",
    "address" : {
```

```
            "street" : "123 Park Street",
            "city" : "Anytown",
            "state" : "NY"
        }
    }
```

上面例子中"address"的值是另一个文档，这个文档有自己的"street"、"city"和"state"键值。

同数组一样，MongoDB能够“理解”内嵌文档的结构，并能“深入”其中构建索引、执行查询，或者更新。

我们会在后面深入讨论模式设计，但就算是从这个简单的例子也可以看出内嵌文档可以改变处理数据的方式。在关系型数据库中，之前的文档一般会被拆分成两个表（“people”和“address”）中的两个行。在MongoDB中，就可以将地址文档直接嵌入人员文档中。使用得当的话，内嵌文档会使信息表示得更加自然（通常也会更高效）。

这样做也有坏处，因为MongoDB会储存更多重复的数据，这样是反规范化的。如果在关系数据库中“address”在一个独立的表中，要修复地址中的拼写错误。当我们对“people”和“address”执行连接操作时，每一个使用这个地址的人的信息都会得到更新。但是在MongoDB中，则需要在每个人的文档中修正拼写错误。

2.6.6 _id和ObjectId

MongoDB中存储的文档必须有一个"_id"键。这个键的值可以是任何类型的，默认是个ObjectId对象。在一个集合里面，每个文档都有唯一的"_id"值，来确保集合里面每个文档都能被唯一标识。如果有两个集合的话，两个集合可以都有一个值为123的"_id"键，但是每个集合里面只能有一个"_id"是123的文档。

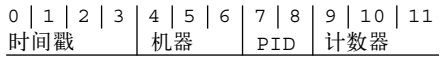
1. ObjectId

ObjectId是"_id"的默认类型。它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。这是MongoDB采用ObjectId，而不是其他比较常规的做法（比如自动增加的主键）的主要原因，因为在多个服务器上同步自动增加主键值既费力还费时。MongoDB从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求。后面会看到ObjectId类型在分片环境中要容易生成得多。

ObjectId使用12字节的存储空间，每个字节两位十六进制数字，是一个24位的字符串。由于看起来很长，不少人会觉得难以处理。但关键是要知道这个长长的ObjectId是实际存储数据的两倍长。

如果快速连续创建多个ObjectId，会发现每次只有最后几位数字有变化。另外，

中间的几位数字也会变化（要是在创建的过程中停顿几秒钟）。这是 ObjectId 的创建方式导致的。12 字节按照如下方式生成：



前 4 个字节是从标准纪元开始的时间戳，单位为秒。这会带来一些有用的属性。

- 时间戳，与随后的 5 个字节组合起来，提供了秒级别的唯一性。
- 由于时间戳在前，这意味着 ObjectId 大致会按照插入的顺序排列。这对于某些方面很有用，如将其作为索引提高效率，但是这个是没有保证的，仅仅是“大致”。
- 这 4 个字节也隐含了文档创建的时间。绝大多数驱动都会公开一个方法从 ObjectId 获取这个信息。

因为使用的是当前时间，很多用户担心要对服务器进行时间同步。其实没有这个必要，因为时间戳的实际值并不重要，只要其总是不停增加就好了（每秒一次）。

接下来的 3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的 ObjectId，不产生冲突。

为了确保在同一台机器上并发的多个进程产生的 ObjectId 是唯一的，接下来的两字节来自产生 ObjectId 的进程标识符（PID）。

前 9 字节保证了同一秒钟不同机器不同进程产生的 ObjectId 是唯一的。后 3 字节就是一个自动增加的计数器，确保相同进程同一秒产生的 ObjectId 也是不一样的。同一秒钟最多允许每个进程拥有 256^3（16 777 216）个不同的 ObjectId。

2. 自动生成_id

前面讲到，如果插入文档的时候没有 "_id" 键，系统会自动帮你创建一个。可以由 MongoDB 服务器来做这件事情，但通常会在客户端由驱动程序完成。理由如下。

- 虽然 ObjectId 设计成轻量型的，易于生成，但是毕竟生成的时候还是产生开销。在客户端生成体现了 MongoDB 的设计理念：能从服务器端转移到驱动程序来做的事，就尽量转移。这种理念背后的原因是，即便是像 MongoDB 这样的可扩展数据库，扩展应用层也要比扩展数据库层容易得多。将事务交由客户端来处理，就减轻了数据库扩展的负担。
- 在客户端生成 ObjectId，驱动程序能够提供更加丰富的 API。例如，驱动程序可以有自己的 insert 方法，可以返回生成的 ObjectId，也可以直接将其插入文档。如果驱动程序允许服务器生成 ObjectId，那么将需要单独的查询，以确定插入的文档中的 "_id" 值。