MongoDB数据库设计

孟宁

关注孟宁

MongoDB数据库

- MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era. No database is more productive to use.

- MongoDB is a document database, which means it stores data in JSON-like documents. We believe this is the most natural way to think about data, and is much more expressive and powerful than the traditional row/column model.

Rich JSON Documents

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  },
  "hobbies": ["surfing", "coding"]
}
```

Powerful query language

All the power of a relational database, and more...

- Full ACID transactions.
  - 原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务（Transaction）的数据库，必须要具有这四种特性，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。
- Support for joins in queries.
- Two types of relationships instead of one: reference and embedded.

一对多关系建模的三种基础方案

- 当你设计一个MongoDB数据库结构，你需要先问自己一个在使用关系型数据库时不会考虑的问题：这个关系中集合的大小是什么样的规模？你需要意识到一对很少，一对许多，一对非常多，这些细微的区别。不同的情况下你的建模也将不同。
- Modeling One-to-Few
- One-to-Many
- One-to-Squillions

Modeling One-to-Few

针对个人需要保存多个地址进行建模的场景下使用内嵌文档是很合适，可以在person文档中嵌入addresses数组文档

```
> db.person.findOne()
{
  name: 'Kate Monster',
  ssn: '123-456-7890',
  addresses : [
     { street: '123 Sesame St', city: 'Anytown', cc: 'USA' },
     { street: '123 Avenue Q', city: 'New York', cc: 'USA' }
  ]
}
```

One-to-Many

- 以产品零件订货系统为例。每个商品有数百个可替换的零件，但是不会超过数千个。这个用例很适合使用间接引用---将零件的objectid作为数组存放在商品文档中(在这个例子中的ObjectID我使用更加易读的2字节，正常是由12个字节组成的)。

```
> db.parts.findOne()
{
    _id : ObjectID('AAAA'),
    partno : '123-aff-456',
    name : '#4 grommet',
    qty: 94,
    cost: 0.94,
    price: 3.99
```

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [     // array of references to Part documents
        ObjectID('AAAA'),    // reference to the #4 grommet above
        ObjectID('F17C'),    // reference to a different Part
        ObjectID('D2AA'),
        // etc
    ]
```

```
 // Fetch the Product document identified by this catalog number
> product = db.products.findOne({catalog_number: 1234});
   // Fetch all the Parts that are linked to this Product
> product_parts = db.parts.find({_id: { $in : product.parts } } ).toArray() ;
```

One-to-Squillions

- 我们用一个收集各种机器日志的例子来讨论一对非常多的问题。由于每个mongodb的文档有16M的大小限制，所以即使你是存储ObjectID也是不够的。我们可以使用很经典的处理方法“父级引用”---用一个文档存储主机，在每个日志文档中保存这个主机的ObjectID。

```
> db.hosts.findOne()
{
    _id : ObjectID('AAAB'),
    name : 'goofy.example.com',
    ipaddr : '127.66.66.66'
}

>db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    host: ObjectID('AAAB')       // Reference to the Host document
}
```

```
  // find the parent ‘host’ document
> host = db.hosts.findOne({ipaddr : '127.66.66.66'});  // assumes unique index
   // find the most recent 5000 log message documents linked to that host
> last_5k_msg = db.logmsg.find({host: host._id}).sort({time : -1}).limit(5000).toArray()
```

内嵌，子引用，父引用

- 三种基本的设计方案：内嵌，子引用，父引用
- 在选择方案时需要考虑的两个关键因素：1）一对多中的多是否需要一个单独的实体；2)这个关系中集合的规模是一对很少，很多，还是非常多。
  - 一对很少且不需要单独访问内嵌内容的情况下可以使用内嵌多的一方。
  - 一对很多且很多的一端内容因为各种理由需要单独存在的情况下可以通过数组的方式引用多的一方的。
  - 一对非常多的情况下，请将一的那端引用嵌入进多的一端对象中。

双向关联Two-Way Referencing

- 以任务跟踪系统为例。有person和task两个集合，one-to-n的关系是从person端到task端。在需要获取person所有的task这个场景下需要在person这个对象中保存有task的id数组

- 在某些场景中这个应用需要显示任务的列表（例如显示一个多人协作项目中所有的任务），为了能够快速的获取某个用户负责的项目可以在task对象中嵌入附加的person引用关系。

```
db.person.findOne()
{
    _id: ObjectID("AAF1"),
    name: "Kate Monster",
    tasks [     // array of references to Task documents
        ObjectID("ADF9"),
        ObjectID("AE02"),
        ObjectID("AE73")
        // etc
    ]
}
```

```
db.tasks.findOne()
{
    _id: ObjectID("ADF9"),
    description: "Write lesson plan",
    due_date:  ISODate("2014-04-01"),
    owner: ObjectID("AAF1")     // Reference to Person document
}
```

双向关联Two-Way Referencing

- 以任务跟踪系统为例。有person和task两个集合，one-to-n的关系是从person端到task端。在需要获取person所有的task这个场景下需要在person这个对象中保存有task的id数组

- 在某些场景中这个应用需要显示任务的列表（例如显示一个多人协作项目中所有的任务），为了能够快速的获取某个用户负责的项目可以在task对象中嵌入附加的person引用关系。

- 这个方案具有所有的一对多方案的优缺点，但是通过添加附加的引用关系。在task文档对象中添加额外的“owner”引用可以很快的找到某个task的所有者，但是如果想将一个task分配给其他person就需要更新引用中的person和task这两个对象（熟悉关系数据库的童鞋会发现这样就没法保证操作的原子性。当然，这对任务跟踪系统来说并没有什么问题，但是你必须考虑你的用例是否能够容忍）

在一对多关系中应用反范式

- 在你的设计中加入反范式，可以使你避免应用层级别的join读取，当然，代价是这也会让你在更新是需要操作更多数据。

Denormalizing from Many -> One

- 以产品和零件为例，你可以在parts数组中冗余存储零件的名字。
- 反范式化意味着你不需要执行一个应用层级别的join去显示一个产品所有的零件名字，当然如果你同时还需要其他零件信息那这个应用层的join是避免不了的。

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [
        { id : ObjectID('AAAA'), name : '#4 grommet' },
        { id: ObjectID('F17C'), name : 'fan blade assembly' },
        { id: ObjectID('D2AA'), name : 'power switch' },
        // etc
    ]
}
```

反范式化

- 反范式化在节省你读的代价的同时会带来更新的代价：如果你将零件的名字冗余到产品的文档对象中，那么你想更改某个零件的名字你就必须同时更新所有包含这个零件的产品对象。
- 在一个读比写频率高的多的系统里，反范式是有使用的意义的。如果你很经常的需要高效的读取冗余的数据，但是几乎不去变更他的话，那么付出更新上的代价还是值得的。更新的频率越高，这种设计方案的带来的好处越少。
- 例如：假设零件的名字变化的频率很低，但是零件的库存变化很频繁，那么你可以冗余零件的名字到产品对象中，但是别冗余零件的库存。
- 需要注意的是，一旦你冗余了一个字段，那么对于这个字段的更新将不在是原子的。和上面双向引用的例子一样，如果你在零件对象中更新了零件的名字，那么更新产品对象中保存的名字字段前将会存在短时间的不一致。

Denormalizing from One -> Many

- 如果你冗余产品的名字到零件表中，那么一旦更新产品的名字就必须更新所有和这个产品有关的零件，这比起只更新一个产品对象来说代价明显更大。这种情况下，更应该慎重的考虑读写频率。

```
> db.parts.findOne()
{
    _id : ObjectID('AAAA'),
    partno : '123-aff-456',
    name : '#4 grommet',
    product_name : 'left-handed smoke shifter',
    product_catalog_number: 1234,
    qty: 94,
    cost: 0.94,
    price: 3.99
}
```

Denormalizing With “One-To-Squillions” Relationships

- 在日志系统这个一对许多的例子中也可以应用反范式化的技术。你可以将one端（主机对象）冗余到日志对象中
- 如果想获取最近某个ip地址的日志信息就变的很简单，只需要一条语句而不是之前的两条就能完成。
- last_5k_msg = db.logmsg.find({ipaddr : '127.66.66.66'}).sort({time : -1}).limit(5000).toArray()

```
> db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    ipaddr : '127.66.66.66',
    host: ObjectID('AAAB')
}
```

Denormalizing With “One-To-Squillions” Relationships

- 如果one端只有少量的信息存储，你甚至可以全部冗余存储到多端上，合并两个对象。

```
> db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    ipaddr : '127.66.66.66',
    hostname : 'goofy.example.com',
}
```

Denormalizing With “One-To-Squillions” Relationships

- 也可以冗余数据到one端。比如说你想在主机文档中保存最近的1000条日志，可以使用mongodb 2.4中新加入的$eache/$slice功能来保证list有序而且只保存1000条。
- 日志对象保存在logmsg集合中，同时冗余到hosts对象中。这样即使hosts对象中超过1000条的数据也不会导致日志对象丢失。
- 在一对多的情况下，需要慎重的考虑读和更新的频率。冗余日志信息到主机文档对象中只有在日志对象几乎不会发生更新的情况下才是个好的决定。

反范式化总结

- 使用双向引用来优化你的数据库架构，前提是你能接受无法原子更新的代价。
- 可以在引用关系中冗余数据到one端或者N端。
- 在决定是否采用反范式化时需要考虑下面的因素：
- 你将无法对冗余的数据进行原子更新。
- 只有读写比较高的情况下才应该采取反范式化的设计。

总结

- 1、优先考虑内嵌，除非有什么迫不得已的原因。
- 2、需要单独访问一个对象，那这个对象就不适合被内嵌到其他对象中。
- 3、数组不应该无限制增长。如果many端有数百个文档对象就不要去内嵌他们可以采用引用ObjectID的方案；如果有数千个文档对象，那么就不要内嵌ObjectID的数组。该采取哪些方案取决于数组的大小。
- 4、不要害怕应用层级别的join：如果索引建的正确并且通过投影条件限制返回的结果，那么应用层级别的join并不会比关系数据库中join开销大多少。
- 5、在进行反范式设计时请先确认读写比。一个几乎不更改只是读取的字段才适合冗余到其他对象中。
- 6、在mongodb中如何对你的数据建模，取决于你的应用程序如何去访问它们。数据的结构要去适应你的程序的读写场景。

参考资料

- MongoDB官网:https://www.mongodb.com/
- http://blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1