

## 第2.1讲 模型设计基础



扫码试看/订阅

《MongoDB 高手课》视频课程

## 本节大纲

内容大纲	学习目标
<p>数据模型设计</p> <p>数据模型元素</p> <p>三层建模流程</p>	<p>了解模型设计的基本要素</p> <p>掌握关系模型的建模过程</p>

# 数据模型

什么是数据模型？

数据模型是一组由符号、文本组成的集合，用以准确表达信息，达到有效交流、沟通的目的。

Steve Hoberman 霍伯曼. 数据建模经典教程

# 数据模型设计的元素

## 实体 Entity

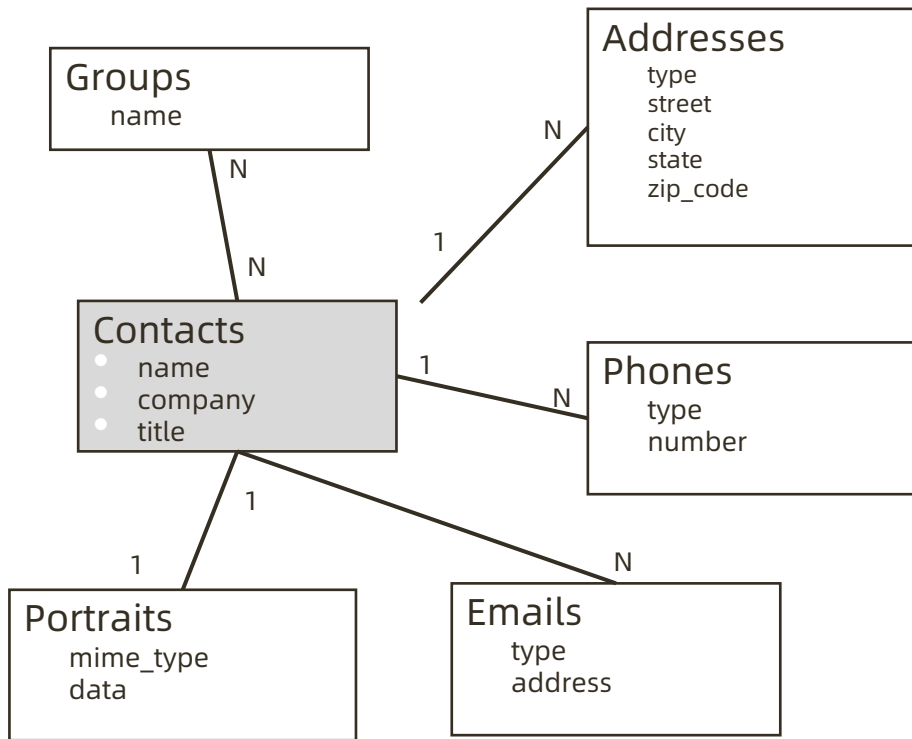
- 描述业务的主要数据集合
- 谁，什么，何时，何地，为何，如何

## 属性 Attribute

- 描述实体里面的单个信息

## 关系 Relationship

- 描述实体与实体之间的数据规则
- 结构规则：1-N, N-1, N-N
- 引用规则：电话号码不能单独存在



# 数据模型设计的元素

## 实体 Entity

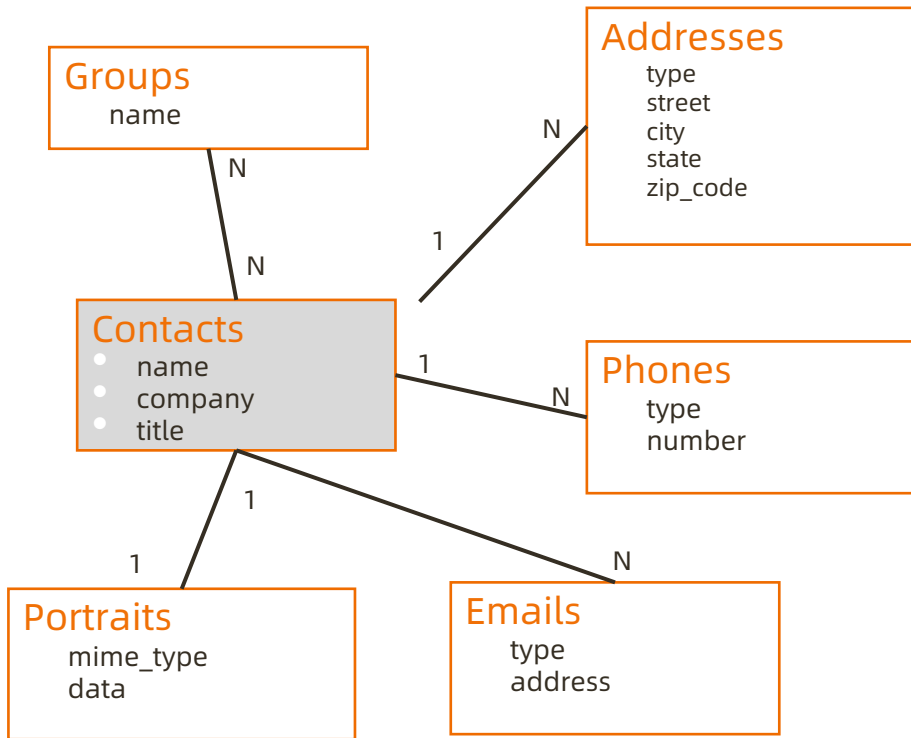
- 描述业务的主要数据集合
- 谁，什么，何时，何地，为何，如何

## 属性 Attribute

- 描述实体里面的单个信息

## 关系 Relationship

- 描述实体与实体之间的数据规则
- 结构规则：1-N, N-1, N-N
- 引用规则：电话号码不能单独存在



# 数据模型设计的元素

## 实体 Entity

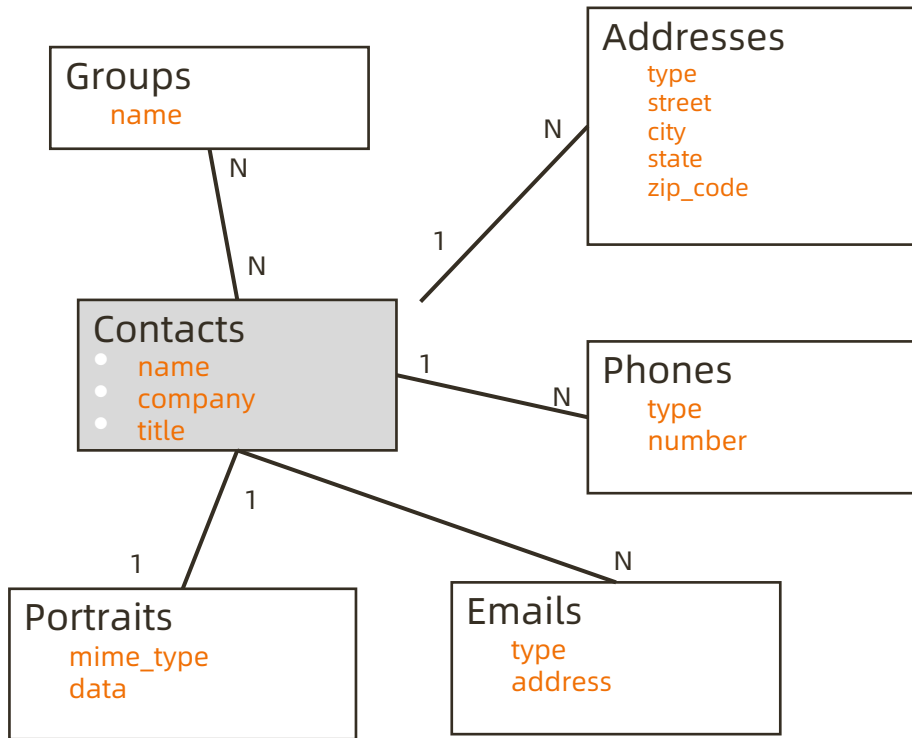
- 描述业务的主要数据集合
- 谁，什么，何时，何地，为何，如何

## 属性 Attribute

- 描述实体里面的单个信息

## 关系 Relationship

- 描述实体与实体之间的数据规则
- 结构规则：1-N, N-1, N-N
- 引用规则：电话号码不能单独存在



# 数据模型设计的元素

## 实体 Entity

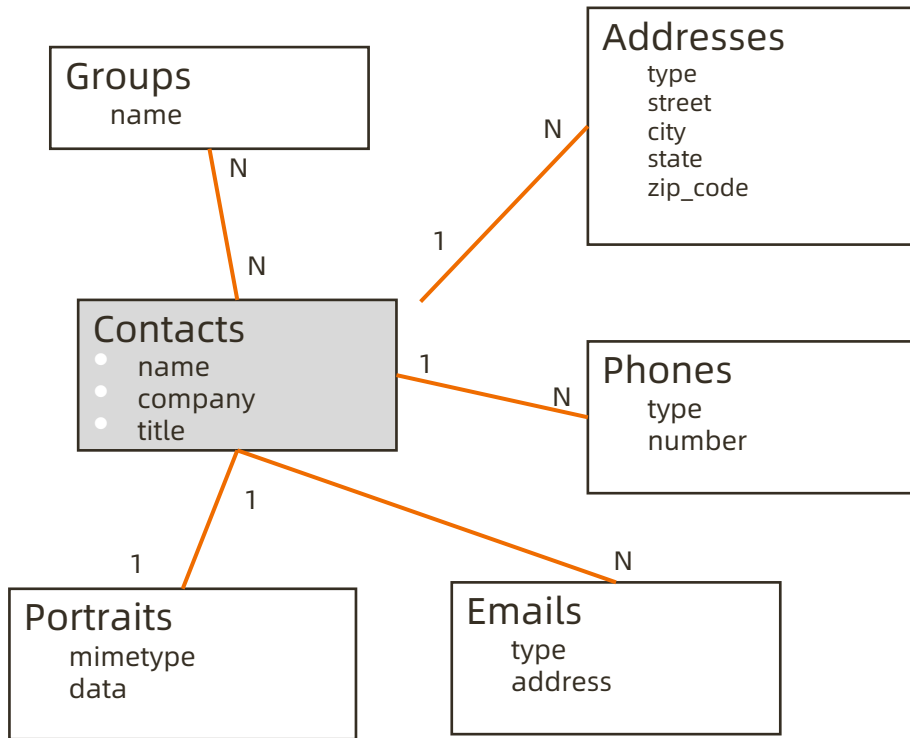
- 描述业务的主要数据集合
- 谁，什么，何时，何地，为何，如何

## 属性 Attribute

- 描述实体里面的单个信息

## 关系 Relationship

- 描述实体与实体之间的数据规则
- 结构规则：1-N, N-1, N-N
- 引用规则：电话号码不能单独存在

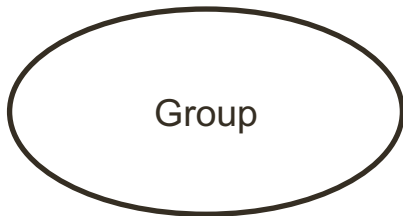
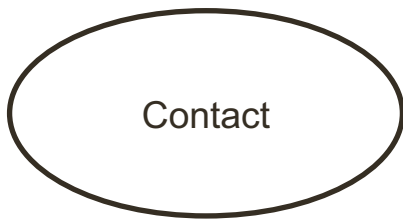




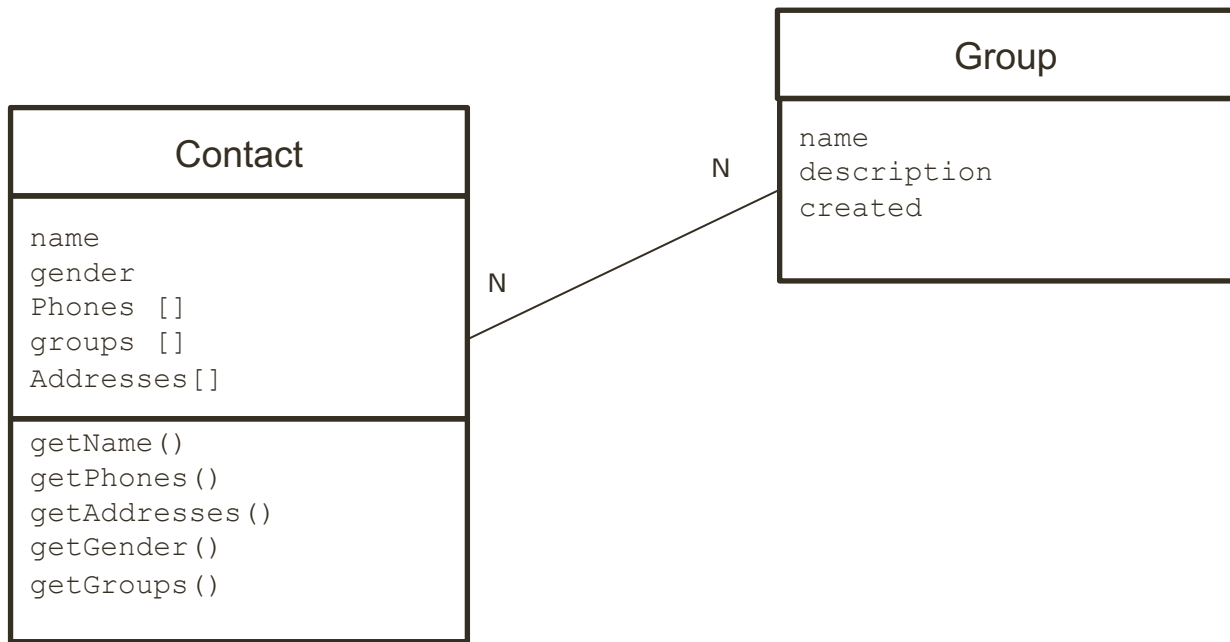
## 传统模型设计：从概念到逻辑到物理

	概念模型 CDM	逻辑模型 LDM	物理模型 PDM
目的	描述业务系统要管理的对象	基于概念模型，详细列出所有实体、实体的属性及关系	根据逻辑模型，结合数据库的物理结构，设计具体的表结构，字段列表及主外键
特点	用概念名词来描述现实中的实体及业务规则，如“联系人”	基于业务的描述和数据库无关	技术实现细节和具体的数据库类型相关
主要使用者	用户 需求分析师	需求分析师 架构师及开发者	开发者 DBA

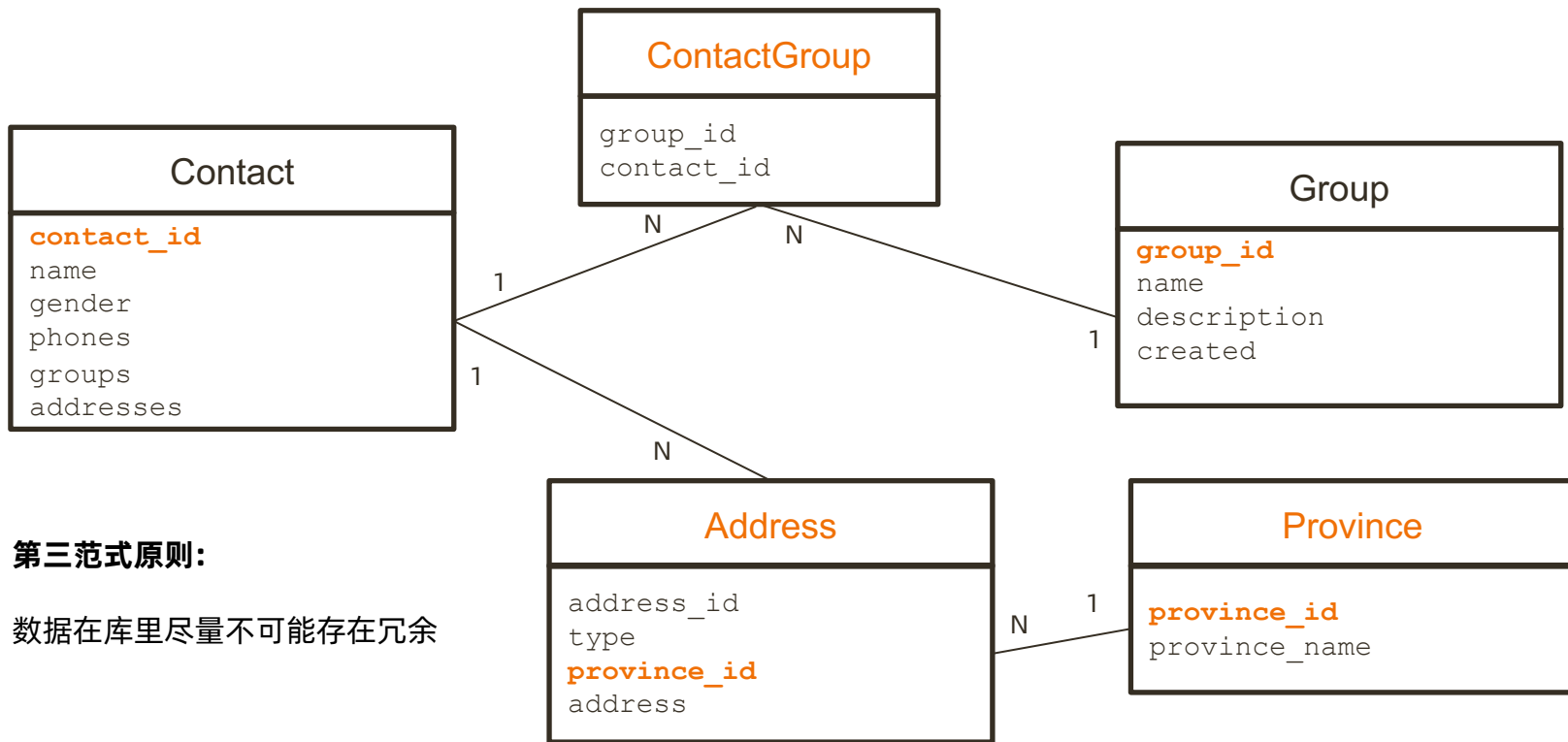
## 从开发者的视角：概念模型



## 从开发者的视角：逻辑模型



## 从开发者的视角：第三范式下的物理模型



## 模型设计小结

数据模型的三要素：

- 实体
- 属性
- 关系

数据模型的三层深度：

- 概念模型，逻辑模型，物理模型
- 一个模型逐步细化的过程

## 第2.2讲 JSON 文档模型设计特点

## 本节大纲

内容大纲	学习目标
文档模型设计目标及原则	掌握文档模型设计的关键考量点
关系模型和文档模型的区别	掌握关系模型和文档模型的区分点

# MongoDB 文档模型设计的三个误区

- 1: 不需要模型设计
- 2: MongoDB 应该用一个超级大文档来组织所有数据
- 3: MongoDB 不支持关联或者事务



# MongoDB 文档模型设计的三个误区

- 1: 不需要模型设计
- 2: MongoDB 应该用一个超级大文档来组织所有数据
- 3: MongoDB 不支持关联或者事务

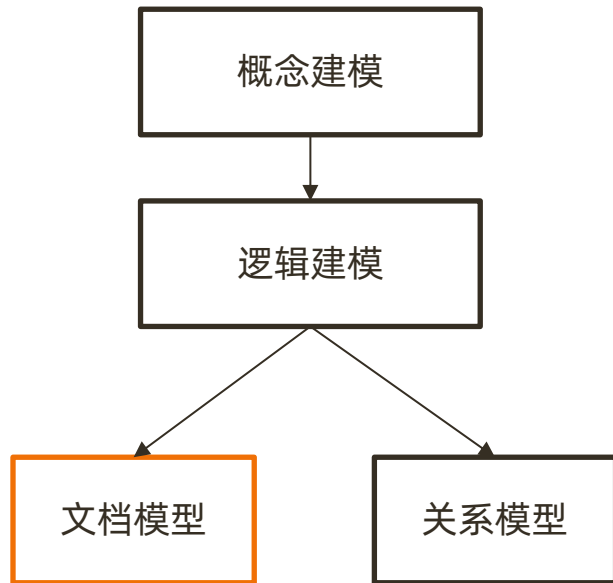
上述陈述均为错!

## 关于 JSON 文档模型设计

文档模型设计处于是物理模型设计阶段（PDM）

JSON 文档模型通过内嵌数组或引用字段来表示关系

文档模型设计不遵从第三范式，允许冗余。



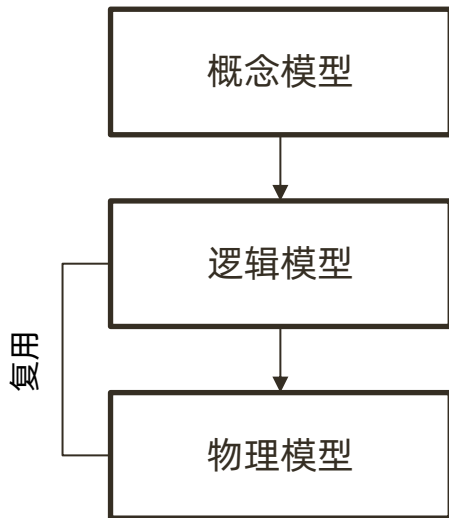
# 为什么人们都说 MongoDB 是无模式？

严格来说，MongoDB 同样需要概念/逻辑建模

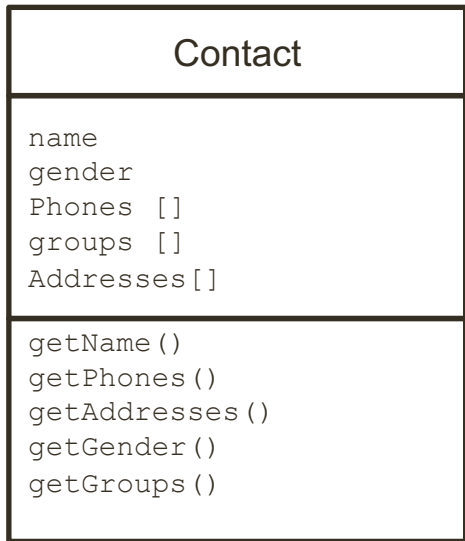
文档模型设计的物理层结构可以和逻辑层类似

MongoDB 无模式由来：

可以省略物理建模的具体过程



## 逻辑模型 - JSON 模型



```
{  
  "name": "TJ Tang",  
  "gender": "M",  
  "created": "2019-01-01",  
  "groups": ["Friends", "Kitesurfers", "校友"]  
  "addresses": [  
    {  
      "type": "home",  
      "province": "广东",  
      "city": "深圳",  
      "address": "望海路1号"  
    },  
    {  
      "type": "work",  
      "province": "广东",  
      "city": "深圳",  
      "address": "前湾路2号"  
    }  
  ]  
}
```

## 文档模型的设计原则：性能和易用

性能

Performance

开发易用

Ease of Development

## 关系模型 vs 文档模型

	关系数据库	JSON 文档模型
模型设计层次	概念模型 逻辑模型 物理模型	概念模型 逻辑模型
模型实体	表	集合
模型属性	列	字段
模型关系	关联关系，主外键	内嵌数组，引用字段

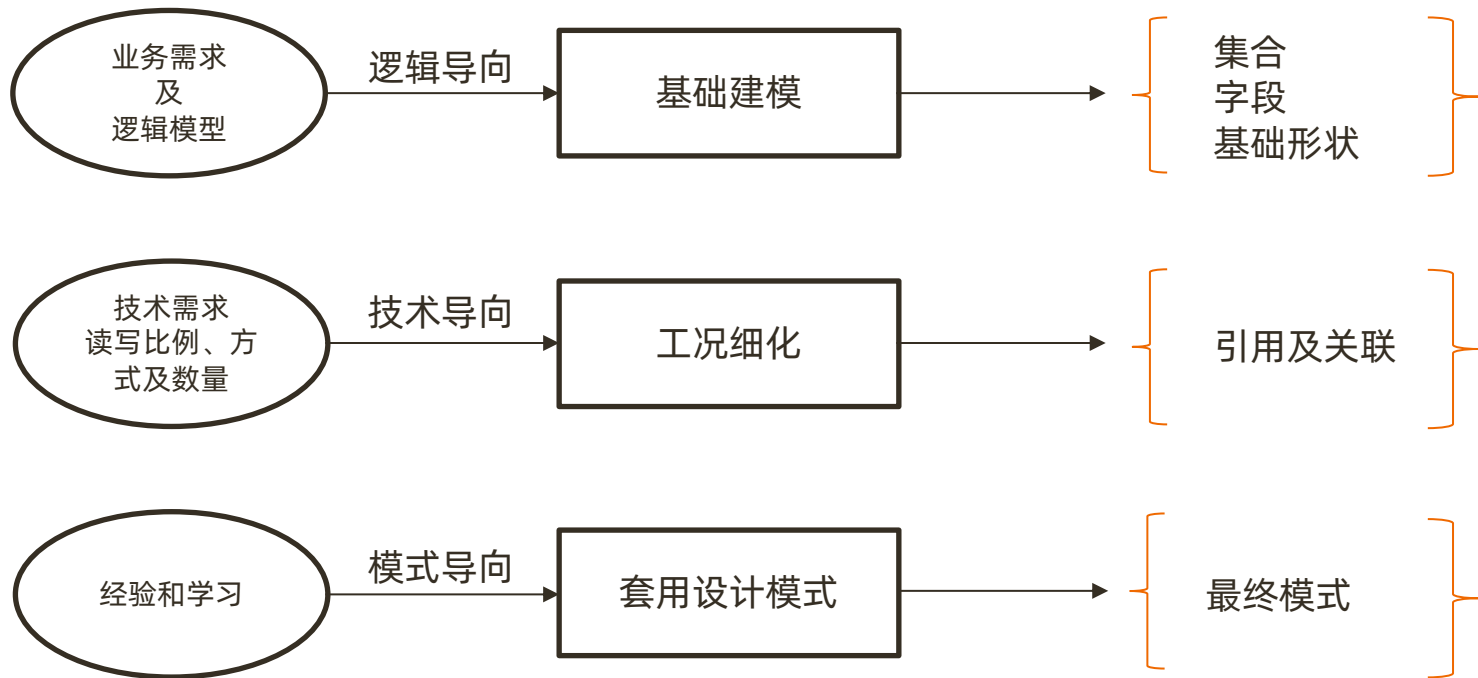
## 第2.3讲 文档模型设计之一：基础设计

## 本节大纲

内容大纲	学习目标
<p>文档模型设计思路</p> <p>基础文档建模方法</p>	<p>了解文档建模的目标</p> <p>掌握第一阶段基础建模方法</p>



# MongoDB 文档模型设计三步曲



## 第一步：建立基础文档模型

1. 根据概念模型或者业务需求推导出逻辑模型 - 找到对象
2. 列出实体之间的关系（及基数） - 明确关系
3. 套用逻辑设计原则来决定内嵌方式 - 进行建模
4. 完成基础模型构建



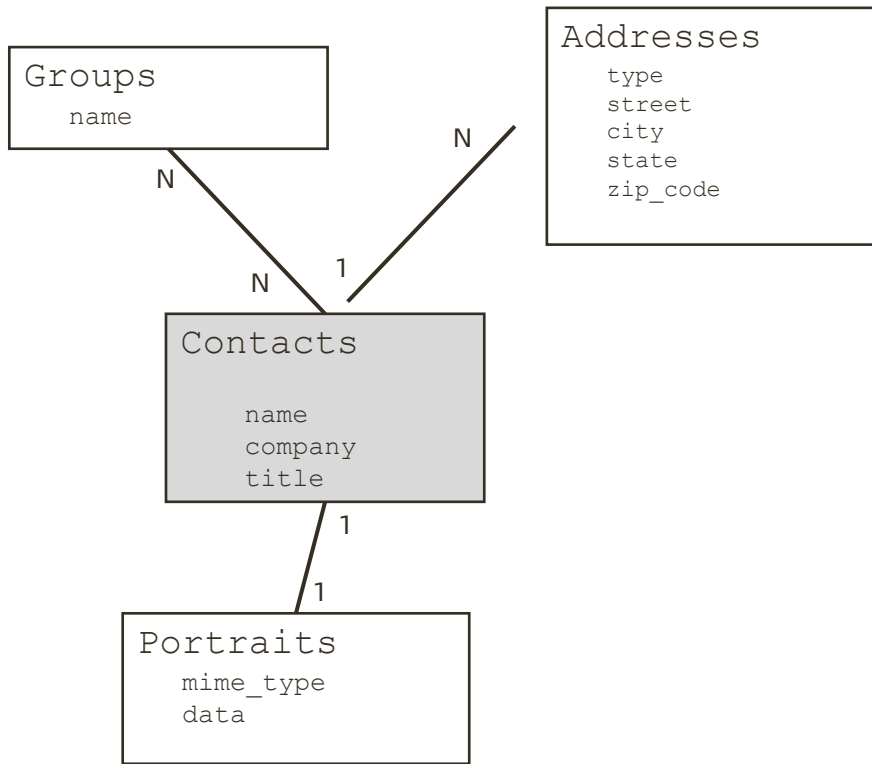
# 一个联系人管理应用的例子

## 1. 找到对象

- Contacts
- Groups
- Address
- Portraits

## 2. 明确关系

- 一个联系人有一个头像 (1-1)
- 一个联系人可以有多个地址 (1-N)
- 一个联系人可以属于多个组，一个组可以有多个联系人 (N - N)



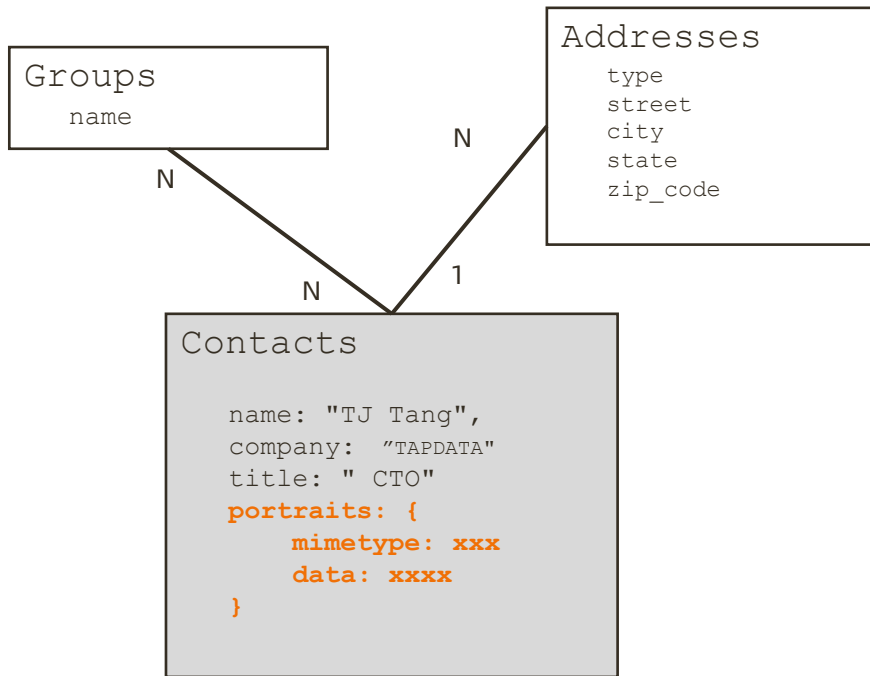
## 1-1 关系建模: portraits

### 基本原则

一对一关系以内嵌为主  
作为子文档形式 或者直接在顶级  
不涉及数据冗余

### 例外情况

如果内嵌后导致文档大小超过16MB



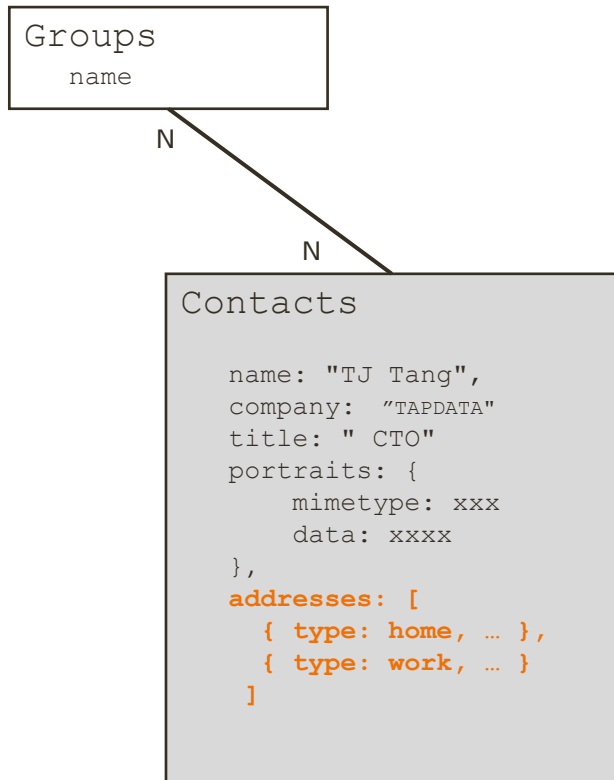
# 1-N 关系建模: Addresses

## 基本原则

一对多关系同样以内嵌为主  
用数组来表示一对多  
不涉及到数据冗余

## 例外情况

内嵌后导致文档大小超过16MB  
数组长度太大（数万或更多）  
数组长度不确定



# N-N 关系建模：内嵌数组模式

## 基本原则

不需要映射表  
一般用内嵌数组来表示一对多  
通过冗余来实现N-N

## 例外情况

内嵌后导致文档大小超过16MB  
数组长度太大（数万或更多）  
数组长度不确定

## Contacts

```
name: "TJ Tang",
company: "TAPDATA"
title: " CTO"
portraits: {
  mimetype: xxx
  data: xxxx
},
addresses: [
  { type: home, ... },
  { type: work, ... }
],
groups: [
  {name: "Friends" },
  {name: "Surfers" },
]
```

## 基础建模小结

90:10 规则： 大部分时候你会使用内嵌来表示 1-1, 1-N, N-N

内嵌类似于预先聚合（关联）

内嵌后对读操作通常有优势（减少关联）

## 第2.4讲 文档模型设计之二：工况细化



## 本节大纲

内容大纲	学习目标
<p>工况驱动的模式细化</p> <p>引用模式及关联查询</p> <p>引用模式的原则及限制</p>	<p>掌握通过技术场景来细化模型设计的方法</p>

## 第二步：根据读写工况细化



- 最频繁的数据查询模式
- 最常用的查询参数
- 最频繁的数据写入模式
- 读写操作的比例
- 数据量的大小

基于内嵌的文档模型  
根据业务需求，  
使用引用来避免性能瓶颈  
使用冗余来优化访问性能

# 联系人管理应用的分组需求

1. 用于客户营销

2. 有千万级联系人

3. 需要频繁变动分组 (group)  
的信息, 如增加分组及修改名称  
及描述以及营销状态

4. 一个分组可以有百万级联系人

一个分组信息的改动意味着  
百万级的 DB 操作

## Contacts

```
name: "TJ Tang",
company: "TAPDATA"
title: " CTO"
portraits: {
  mimetype: xxx
  data: xxxx
},
addresses: [
  { type: home, ...
  { type: work, ...
}],
groups: [
  {name: "本科毕业"
  {name: "金融行业"
]
```

## Contacts

```
name: "Mona Zhang",
company: "HUAXIA"
title: " DIRECTOR"
portraits: {
  mimetype: xxx
  data: xxxx
},
addresses: [
  { type: home, ... },
  { type: work, ... }
],
groups: [
  {name: "本科毕业" },
  {name: "外企" },
]
```

## 解决方案： Group 使用单独的集合

1. 类似于关系型设计
2. 用 id 或者唯一键关联
3. 使用 \$lookup 来提供一次查询多表的能力（类似关联）



## 引用模式下的关联查询

### Contacts

```
name: "TJ Tang",  
company: "TAPDATA"  
group_ids: [1, 2, 3...]
```

### Groups

```
group_id: 1  
name: "Friends"
```

```
db.contacts.aggregate([  
  {  
    $lookup:  
    {  
      from: "groups",  
      localField: "group_ids",  
      foreignField: "group_id",  
      as: "groups"  
    }  
  }  
])
```



```
{  
  "_id" : ObjectId("5de26f197edd62c5d388babb"),  
  "name" : "TJ",  
  "company" : "Tapdata",  
  "group_ids" : [  
    1,  
    3  
  ],  
  "groups" : [  
    {  
      "_id" : ObjectId("5de26f4d7edd62c5d388babc"),  
      "name" : "Friends",  
      "group_id" : 1  
    },  
    {  
      "_id" : ObjectId("5de26f4d7edd62c5d388babe"),  
      "group_id" : 3,  
      "name" : "Surfers"  
    }  
  ]  
}
```

一个查询语句（用 aggregate）

得到 2 个或多个表的输出

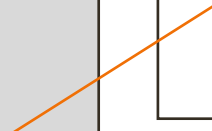
## 联系人的头像： 引用模式

1. 头像使用高保真，大小在 5MB-10MB
2. 头像一旦上传，一个月不可更换
3. 基础信息查询（不含头像）和 头像查询的比例为 9 : 1
4. 建议： 使用引用方式，把头像数据放到另外一个集合，可以显著提升 90% 的查询效率

### Contacts

```
name: "TJ Tang",
company: "TAPDATA"
title: " CTO"
portrait_id: 123,
addresses: [
  { type: home, ... },
  { type: work, ... }
]
```

```
Contact_Portrait
_id: 123,
mimetype: "xxx",
data: "xxxxx..."
```



## 什么时候该使用引用方式?

内嵌文档太大，数 MB 或者超过 16MB

内嵌文档或数组元素会频繁修改

内嵌数组元素会持续增长并且没有封顶

## MongoDB 引用设计的限制

MongoDB 对使用引用的集合之间并无主外键检查

MongoDB 使用聚合框架的 \$lookup 来模仿关联查询

\$lookup 只支持 left outer join

\$lookup 的关联目标 (from) 不能是分片表

```
db.contacts.aggregate([
  {
    $lookup:
    {
      from: "groups",
      localField: "group_ids",
      foreignField: "group_id",
      as: "groups"
    }
  }
])
```



## 第2.5讲 文档模型设计之三：模式套用

## 本节大纲

内容大纲	学习目标
设计模式的价值	了解设计模式
设计模式分类	了解分桶设计模式

### 第三步：套用设计模式

文档模型：无范式，无思维定式，充分发挥想象力

设计模式：实战过屡试不爽的设计技巧，快速应用

举例：一个 IoT 场景的分桶设计模式，可以帮助把存储空间降低 10 倍并且查询效率提升数十倍。



## 问题: 物联网场景下的海量数据处理 - 飞机监控数据

```
{  
  "_id" : "20160101050000:CA2790",  
  "icao" : "CA2790",  
  "callsign" : "CA2790",  
  "ts" : ISODate("2016-01-01T05:00:00.000+0000"),  
  "events" : {  
    "a" : 31418,  
    "b" : 173,  
    "p" : [115, -134],  
    "s" : 91,  
    "v" : 80  
  }  
}
```

## 520亿条，10TB - 海量数据

- 10万架飞机
- 1 年的数据
- 每分钟一条

	每分钟1条	
文档条数	52.6 B	
索引大小	6364 GB	
_id index	1468 GB	
{ts: 1, deviceId: 1}	4895 GB	
文档平均大小	92 Bytes	
数据大小	4503 GB	

## 520亿条，10TB - 海量数据

- 10万架飞机
- 1 年的数据
- 每分钟一条

$$100000 * 365 * 24 * 60$$

	每分钟1条	
文档条数	52.6 B	
索引大小	6364 GB	
_id index	1468 GB	
{ts: 1, deviceId: 1}	4895 GB	
文档平均大小	92 Bytes	
数据大小	4503 GB	

## 520亿条，10TB - 海量数据

- 10万架飞机
- 1 年的数据
- 每分钟一条

$$100000 * 365 * 24 * 60$$

$$100000 * 365 * 24 * 60 * 130$$

	每分钟1条	
文档条数	52.6 B	
索引大小	6364 GB	
_id index	1468 GB	
{ts: 1, deviceId: 1}	4895 GB	
文档平均大小	92 Bytes	
数据大小	4503 GB	

## 520亿条，10TB - 海量数据

- 10万架飞机
- 1 年的数据
- 每分钟一条

$$100000 * 365 * 24 * 60$$

$$100000 * 365 * 24 * 60 * 130$$

	每分钟1条	
文档条数	52.6 B	
索引大小	6364 GB	
_id index	1468 GB	
{ts: 1, deviceId: 1}	4895 GB	
文档平均大小	92 Bytes	
数据大小	4503 GB	

$$100000 * 365 * 24 * 60 * 92$$



## 解决方案: 分桶设计

```
{  
  "_id" : "20160101050000:WG9943",  
  "icao" : "WG9943",  
  "ts" : ISODate("2016-01-01T05:00:00.000+0000"),  
  "events" : [  
    {  
      "a" : 24293, "b" : 319, "p" : [41, 70], "s" : 56,  
      "t" : ISODate("2016-01-01T05:00:00.000+0000")  
    },  
    {  
      "a" : 33663, "b" : 134, "p" : [-38, -30], "s" : 385,  
      "t" : ISODate("2016-01-01T05:00:01.000+0000")  
    },  
    ...  
  ]  
}
```

60 Events  
==  
1 小时数据

一个文档：一架飞机一个小时的数据

## 520亿条，10TB - 海量数据

可视化表现 24 小时的飞行数据

1440 次读

	每分钟1个文档	每小时一个文档
文档条数	52.6 B	876 M
索引大小	6364 GB	106 GB
_id index	1468 GB	24.5 GB
{ts: 1, deviceId: 1}	4895 GB	81.6 GB
文档平均大小	92 Bytes	758 Bytes
数据大小	4503 GB	618 GB

## 模式小结：分桶

场景	痛点	设计模式的方案及优点
时序数据 物联网 智慧城市 智慧交通	数据点采集频繁，数据量太多	利用文档内嵌数组，将一个时间段的数据聚合到一个文档里。  大量减少文档数量  大量减少索引占用空间

## 本讲小结

一个好的设计模式可以显著地：

- 提升数据读写的效率
- 降低资源的需求

更多的 MongoDB 设计模式：

表现形式类	数据访问类	组织结构类
列转行	子集	预聚合
文档版本	近似处理	分桶

## 第2.6讲 设计模式集锦

## 本节大纲

内容大纲	学习目标
<p>列转行模式</p> <p>版本字段</p> <p>近似计算</p> <p>预聚合</p>	<p>理解4个模式应用场景及使用方法</p>

## 问题: 大文档, 很多字段, 很多索引

```
{  
  title: "Dunkirk",  
  ...  
  release_USA: "2017/07/23",  
  release_UK: "2017/08/01",  
  release_France: "2017/08/01",  
  release_Festival_San_Jose:  
    "2017/07/22"  
}
```

### 需要很多索引

```
{ release_USA: 1 }  
  
{ release_UK: 1 }  
  
{ release_France: 1 }  
  
...  
  
{ release_Festival_San_Jose: 1 }  
...
```

## 解决方案: 列转行

```
{  
  title: "Dunkirk",  
  ...  
  release_USA: "2017/07/23",  
  release_UK: "2017/08/01",  
  release_France: "2017/08/01",  
  release_Festival_San_Jose:  
    "2017/07/22"  
}
```



```
{  
  title: "Dunkirk",  
  ...  
  releases: [  
    { country: "USA", date: "2017/07/23"},  
    { country: "UK", date: "2017/08/01"}  
  ]  
}
```

```
db.movies.createIndex({"releases.country":1, "releases.date":1})
```



## 模式小结：列转行

场景	痛点	设计模式方案及优点
产品属性 'color', 'size', 'dimensions', ...  多语言（多国家）属性	文档中有很多类似的字段  会用于组合查询搜索，需要见很多索引	转化为数组，一个索引解决所有查询问题

## 问题: 模型灵活了, 如何管理文档不同版本?

2019.01 版本1.0

```
{  
  "_id" : ObjectId("5de26f197edd62c5d388babb"),  
  "name" : "TJ",  
  "company" : "Tapdata",  
}
```

2019.03 版本2.0

```
{  
  "_id" : ObjectId("5de26f197edd62c5d388babb"),  
  "name" : "TJ",  
  "company" : "Tapdata",  
  "wechat": "tjttang826"  
}
```

## 解决方案: 增加一个版本字段

2019.01 版本1.0

```
{  
  "_id" : ObjectId("5de26f197edd62c5d388babb"),  
  "name" : "TJ",  
  "company" : "Tapdata",  
}
```

2019.03 版本2.0

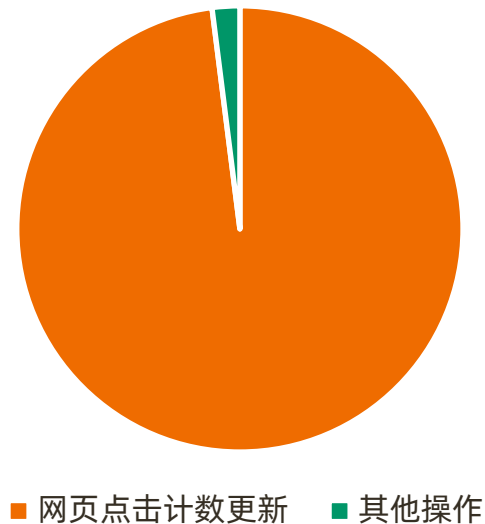
```
{  
  "_id" : ObjectId("5de26f197edd62c5d388babb"),  
  "name" : "TJ",  
  "company" : "Tapdata",  
  "wechat": "tjtang826",  
  "schema_version": "2.0"  
}
```

## 模式小结： 版本字段

场景	痛点	设计模式方案及优点
任何有版本衍变的数据库	文档模型格式多，无法知道其合理性  升级时候需要更新太多文档	增加一个版本号字段  快速过滤掉不需要升级的文档  升级时候对不同版本的文档做不同的处理

## 问题: 统计网页点击流量

数据库写入操作分布



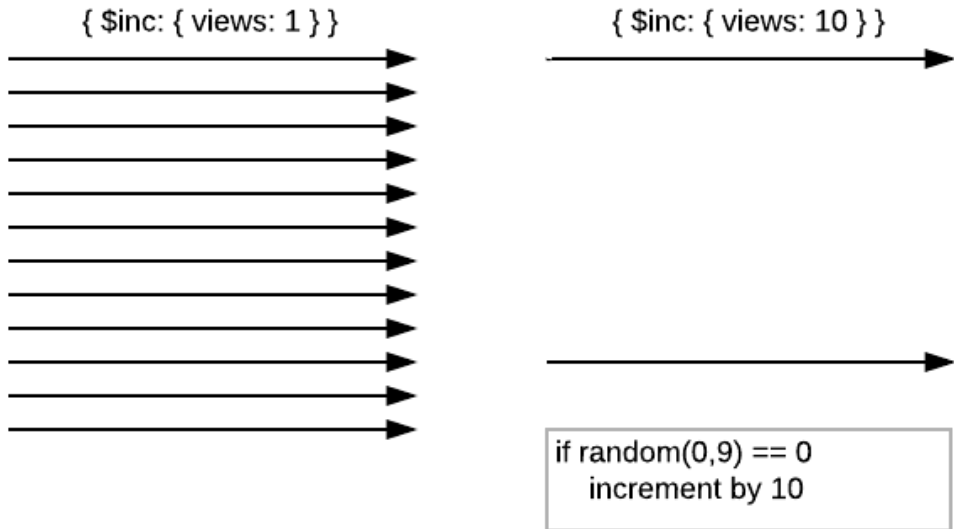
每访问一个页面都会产生一次数据库计数更新操作

统计数字准确性并不十分重要

## 解决方案: 用近似计算

每隔10 (X)次写一次

Increment by 10(X)



## 模式小结：近似计算

场景	痛点	设计模式方案及优点
网页计数 各种结果不需要准确的排名	写入太频繁，消耗系统资源	间隔写入，每隔10次或者100次 大量减少写入需求

## 问题：业绩排名，游戏排名，商品统计等精确统计

热销榜：某个商品今天卖了多少，这个星期卖了多少，这个月卖了多少？

电影排行：观影者，场次统计

传统解决方案：通过聚合计算

痛点：消耗资源多，聚合计算时间长



## 解决方案: 用预聚合字段

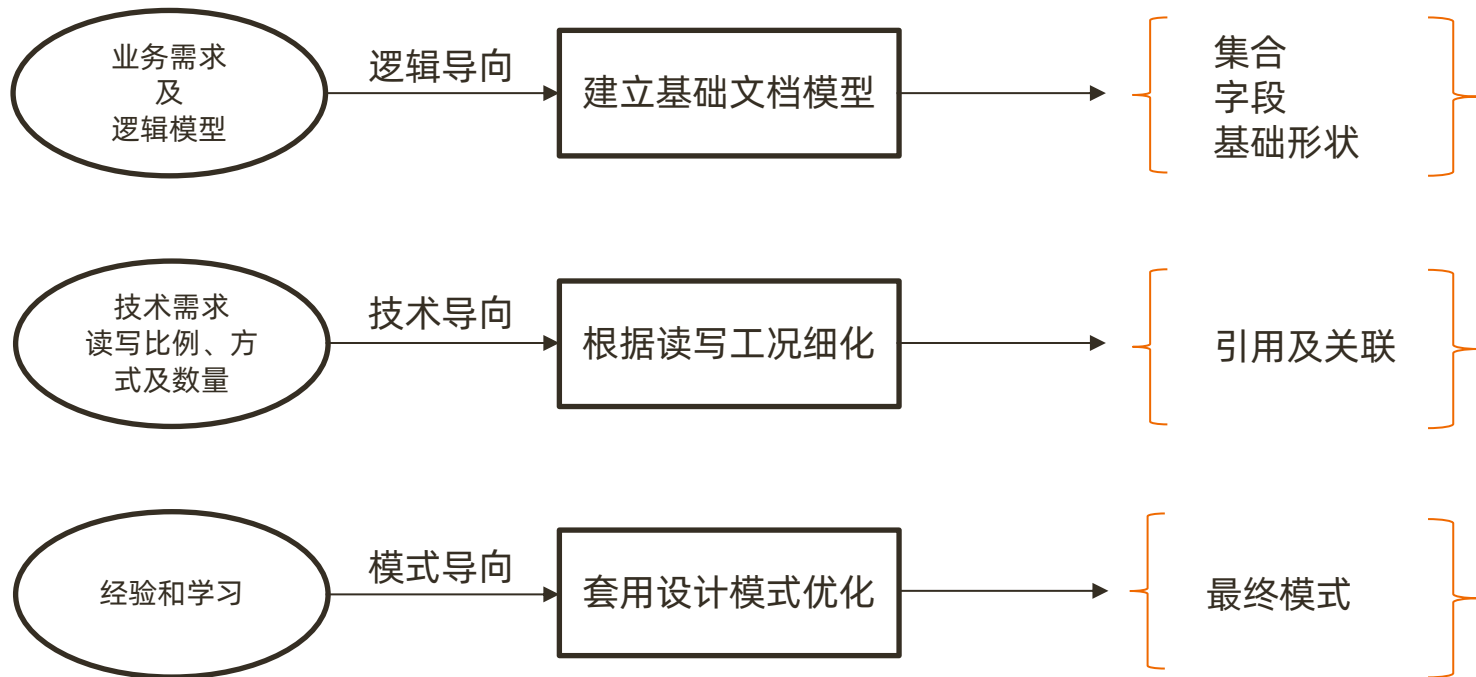
```
{  
  product: "Bike",  
  sku: "abc123456",  
  quantity: 20394,  
  daily_sales: 40,  
  weekly_sales: 302,  
  monthly_sales: 1419  
}
```

```
db.inventory.update({_id:123},  
  
  {$inc: {  
  
    quantity: -1,  
    daily_sales: 1,  
    weekly_sales: 1,  
    monthly_sales: 1,  
  
  }}  
)
```

## 模式小结：预聚合

场景	痛点	设计模式方案及优点
准确排名 排行榜	统计计算耗时，计算时间长	模型中直接增加统计字段  每次更新数据时候同时更新统计值

# MongoDB 文档模型设计：小结



## 第2.7讲 事务开发：写操作事务

## 本节大纲

内容大纲	学习目标
<p>什么是 writeConcern</p> <p>writeConcern 的意义</p> <p>writeConcern vs journal</p>	<p>了解并学会正确使用 writeConcern</p>

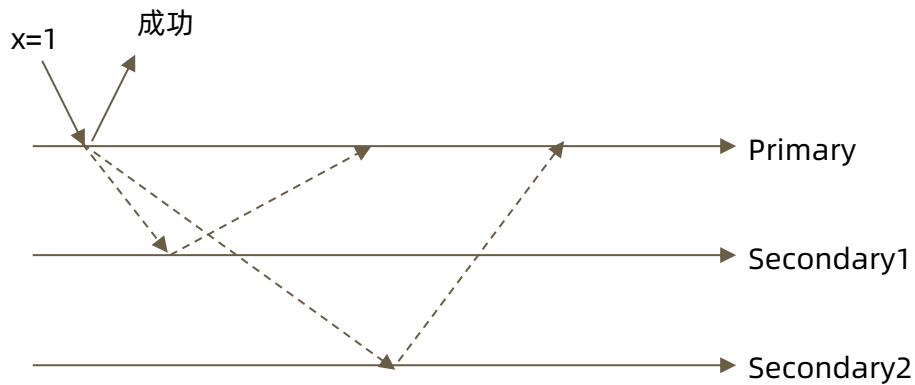
## 什么是 writeConcern ?

writeConcern 决定一个写操作落到多少个节点上才算成功。writeConcern 的取值包括：

- 0：发起写操作，不关心是否成功；
- 1~集群最大数据节点数：写操作需要被复制到指定节点数才算成功；
- majority：写操作需要被复制到大多数节点上才算成功。

发起写操作的程序将阻塞到写操作到达指定的节点数为止

3 节点复制集不作任何特别设定（默认值）：

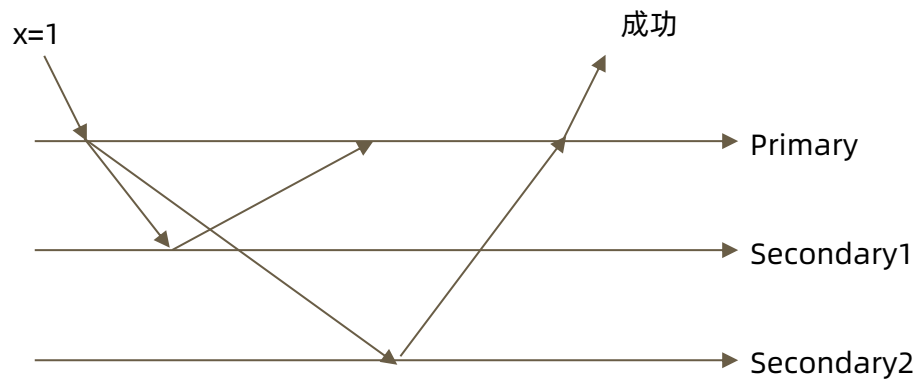






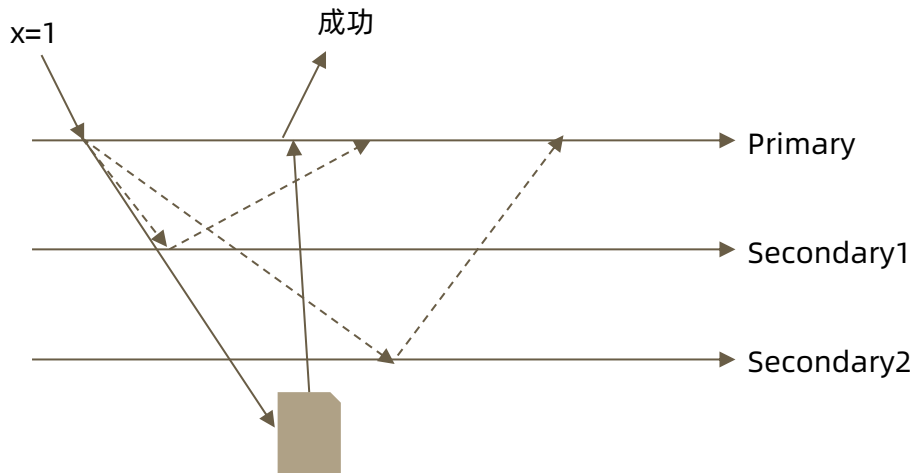
w: "all"

全部节点确认模式



writeConcern 可以决定写操作到达多少个节点才算成功，journal 则定义如何才算成功。取值包括：

- true: 写操作落到 journal 文件中才算成功；
- false: 写操作到达内存即算作成功。



## writeConcern 的意义

对于5个节点的复制集来说，写操作落到多少个节点上才算是安全的？

- 1
- 2
- 3
- 4
- 5
- majority

## writeConcern 的意义

对于5个节点的复制集来说，写操作落到多少个节点上才算是安全的？

- 1
- 2
- 3 ✓
- 4 ✓
- 5 ✓
- majority ✓

## writeConcern 实验

### 在复制集测试writeConcern参数

```
db.test.insert( {count: 1}, {writeConcern: {w: "majority"}})
```

```
db.test.insert( {count: 1}, {writeConcern: {w: 3 }})
```

```
db.test.insert( {count: 1}, {writeConcern: {w: 4 }})
```

### 配置延迟节点，模拟网络延迟（复制延迟）

```
conf=rs.conf()
```

```
conf.members[2].slaveDelay = 5
```

```
conf.members[2].priority = 0
```

```
rs.reconfig(conf)
```

### 观察复制延迟下的写入，以及timeout参数

```
db.test.insert( {count: 1}, {writeConcern: {w: 3}})
```

```
db.test.insert( {count: 1}, {writeConcern: {w: 3, wtimeout:3000 }})
```

## 注意事项

- 虽然多于半数的 writeConcern 都是安全的，但通常只会设置 majority，因为这是等待写入延迟时间最短的选择；
- 不要设置 writeConcern 等于总节点数，因为一旦有一个节点故障，所有写操作都将失败；
- writeConcern 虽然会增加写操作延迟时间，但并不会显著增加集群压力，因此无论是否等待，写操作最终都会复制到所有节点上。设置 writeConcern 只是让写操作等待复制后再返回而已；
- 应对重要数据应用 {w: “majority” }，普通数据可以应用 {w: 1} 以确保最佳性能。

## 第2.8讲 事务开发：读操作事务之一

readPreference

## 本节大纲

内容大纲	学习目标
<p>什么是 readPreference</p> <p>readPreference 的使用场景</p>	<p>了解并学会正确使用 readPreference</p>



## 综述

在读取数据的过程中我们需要关注以下两个问题：

- 从哪里读？
- 什么样的数据可以读？

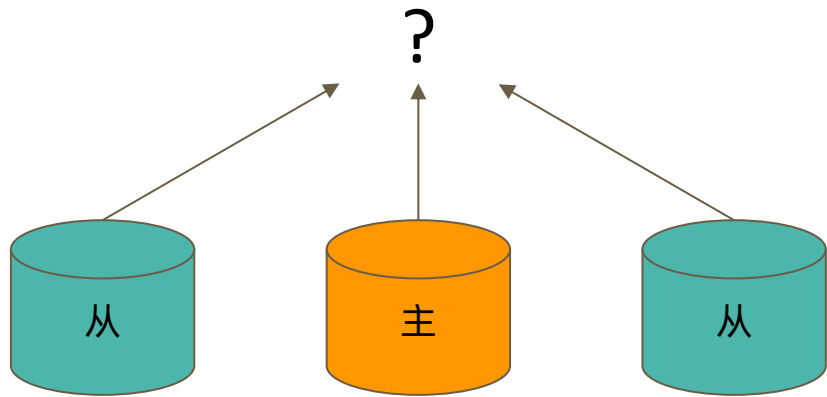
第一个问题是是由 `readPreference` 来解决

第二个问题则是由 `readConcern` 来解决

## 什么是 readPreference?

readPreference 决定使用哪一个节点来满足正在发起的读请求。可选值包括：

- primary: 只选择主节点；
- primaryPreferred: 优先选择主节点，如果不可用则选择从节点；
- secondary: 只选择从节点；
- secondaryPreferred: 优先选择从节点，如果从节点不可用则选择主节点；
- nearest: 选择最近的节点；



## readPreference 场景举例

- 用户下订单后马上将用户转到订单详情页——primary/primaryPreferred。因为此时从节点可能还没复制到新订单；
- 用户查询自己下过的订单——secondary/secondaryPreferred。查询历史订单对时效性通常没有太高要求；
- 生成报表——secondary。报表对时效性要求不高，但资源需求大，可以在从节点单独处理，避免对线上用户造成影响；
- 将用户上传的图片分发到全世界，让各地用户能够就近读取——nearest。每个地区的应用选择最近的节点读取数据。

## readPreference 与 Tag

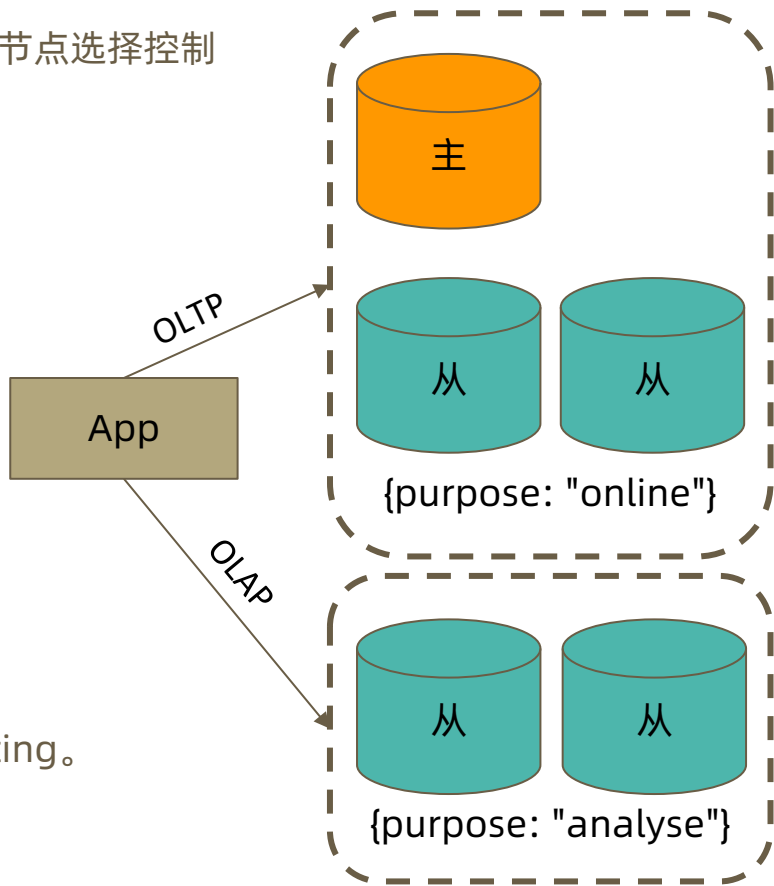
readPreference 只能控制使用一类节点。Tag 则可以将节点选择控制到一个或几个节点。考虑以下场景：

- 一个 5 个节点的复制集；
- 3 个节点硬件较好，专用于服务线上客户；
- 2 个节点硬件较差，专用于生成报表；

可以使用 Tag 来达到这样的控制目的：

- 为 3 个较好的节点打上 {purpose: "online"}；
- 为 2 个较差的节点打上 {purpose: "analyse"}；
- 在线应用读取时指定 online，报表读取时指定 reporting。

更多信息请参考文档：[readPreference](#)



## readPreference 配置

### 通过 MongoDB 的连接串参数:

- `mongodb://host1:27107,host2:27107,host3:27017/?replicaSet=rs&readPreference=secondary`

### 通过 MongoDB 驱动程序 API:

- `MongoCollection.withReadPreference(ReadPreference readPref)`

### Mongo Shell:

- `db.collection.find({}).readPref( "secondary" )`

## readPreference 实验: 从节点读

- 主节点写入 {x:1}, 观察该条数据在各个节点均可见
- 在两个从节点分别执行 `db.fsyncLock()` 来锁定写入 (同步)
- 主节点写入 {x:2}
  - `db.test.find({a: 123})`
  - `db.test.find({a: 123}).readPref( "secondary" )`
- 解除从节点锁定 `db.fsyncUnlock()`
  - `db.test.find({a: 123}).readPref( "secondary" )`

## 注意事项

- 指定 readPreference 时也应注意高可用问题。例如将 readPreference 指定 primary，则发生故障转移不存在 primary 期间将没有节点可读。如果业务允许，则应选择 primaryPreferred；
- 使用 Tag 时也会遇到同样的问题，如果只有一个节点拥有一个特定 Tag，则在这个节点失效时将无节点可读。这在有时候是期望的结果，有时候不是。例如：
  - 如果报表使用的节点失效，即使不生成报表，通常也不希望将报表负载转移到其他节点上，此时只有一个节点有报表 Tag 是合理的选择；
  - 如果线上节点失效，通常希望有替代节点，所以应该保持多个节点有同样的 Tag；
- Tag 有时需要与优先级、选举权综合考虑。例如做报表的节点通常不会希望它成为主节点，则优先级应为 0。

## 第2.9讲 事务开发：读操作事务之二

readConcern



## 本节大纲

内容大纲	学习目标
<p>什么是 readConcern</p> <p>readConcern 的不同级别</p> <p>readPreference 的使用场景</p> <p>readPreference 的演示</p>	<p>了解并学会正确使用 readConcern</p>

## 什么是 readConcern?

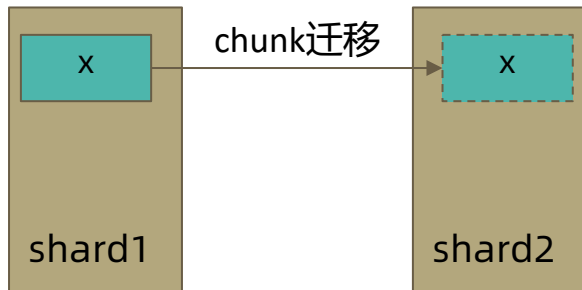
在 readPreference 选择了指定的节点后，readConcern 决定这个节点上的数据哪些是可读的，类似于关系数据库的隔离级别。可选值包括：

- available：读取所有可用的数据；
- local：读取所有可用且属于当前分片的数据；
- majority：读取在大多数节点上提交完成的数据；
- linearizable：可线性化读取文档；
- snapshot：读取最近快照中的数据；

## readConcern: local 和 available

在复制集中 local 和 available 是没有区别的。两者的区别主要体现在分片集上。考虑以下场景：

- 一个 chunk x 正在从 shard1 向 shard2 迁移；
- 整个迁移过程中 chunk x 中的部分数据会在 shard1 和 shard2 中同时存在，但源分片 shard1 仍然是 chunk x 的负责方：
  - 所有对 chunk x 的读写操作仍然进入 shard1；
  - config 中记录的信息 chunk x 仍然属于 shard1；
- 此时如果读 shard2，则会体现出 local 和 available 的区别：
  - local：只取应该由 shard2 负责的数据（不包括 x）；
  - available：shard2 上有什么就读什么（包括 x）；



## readConcern: local 和 available

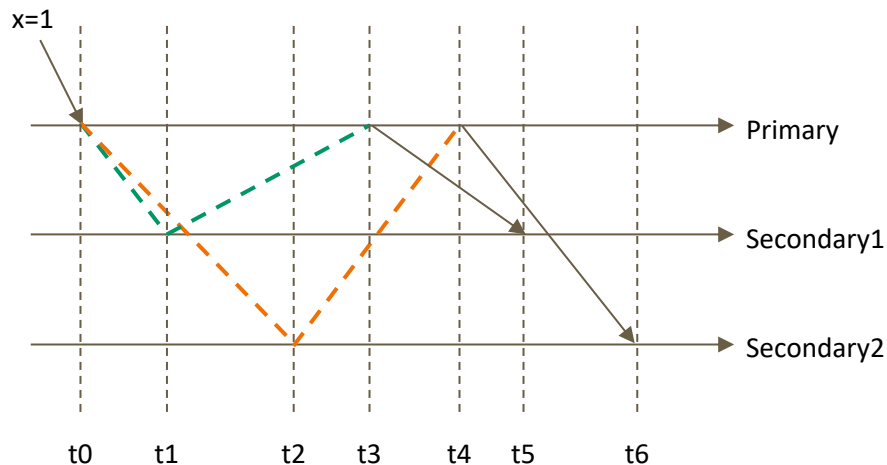
注意事项：

- 虽然看上去总是应该选择 local，但毕竟对结果集进行过滤会造成额外消耗。在一些无关紧要的场景（例如统计）下，也可以考虑 available；
- MongoDB <=3.6 不支持对从节点使用 {readConcern: "local"}；
- 从主节点读取数据时默认 readConcern 是 local，从从节点读取数据时默认 readConcern 是 available（向前兼容原因）。

## readConcern: majority

只读取大多数数据节点上都提交了的数据。考虑如下场景：

- 集合中原有文档 {x: 0};
- 将x值更新为 1;



如果在各节点上应用  
{readConcern: "majority"} 来读取数据：

	P	S1	S2
t0	x=0	x=0	x=0
t1	x=0	x=0	x=0
t2	x=0	x=0	x=0
t3	x=1	x=0	x=0
t4	x=1	x=0	x=0
t5	x=1	x=1	x=0
t6	x=1	x=1	x=1

## readConcern: majority 的实现方式

考虑  $t_3$  时刻的 Secondary1, 此时:

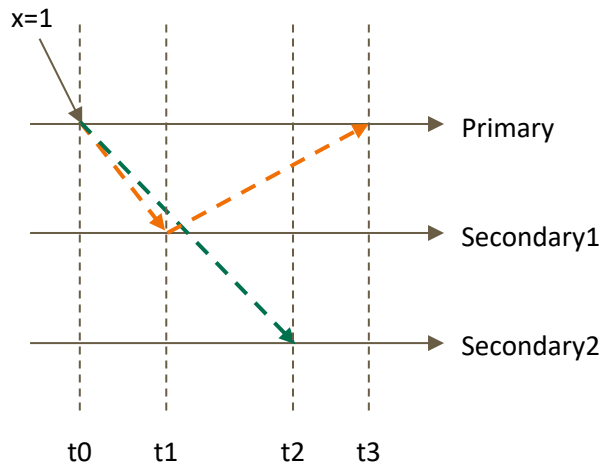
- 对于要求 majority 的读操作, 它将返回  $x=0$ ;
- 对于不要求 majority 的读操作, 它将返回  $x=1$ ;

### 如何实现?

节点上维护多个  $x$  版本, MVCC 机制

MongoDB 通过维护多个快照来链接不同的版本:

- 每个被大多数节点确认过的版本都将是一个快照;
- 快照持续到没有人使用为止才被删除;



## 实验：readConcern：“majority” vs “local”

- 安装 3 节点复制集。
- 注意配置文件内 server 参数 enableMajorityReadConcern

```
replication:  
  replSetName: rs0  
  enableMajorityReadConcern: true
```

- 将复制集中的两个从节点使用 db.fsyncLock() 锁住写入（模拟同步延迟）

## readConcern 验证

- `db.test.save({ "A" :1})`
- `db.test.find().readConcern( "local" )`
- `db.test.find().readConcern( "majority" )`
- 在某一个从节点上执行 `db.fsyncUnlock()`
- 结论：
  - 使用 `local` 参数，则可以直接查询到写入数据
  - 使用 `majority`，只能查询到已经被多数节点确认过的数据
  - `update` 与 `remove` 与上同理。



## readConcern: majority 与脏读

MongoDB 中的回滚：

- 写操作到达大多数节点之前都是不安全的，一旦主节点崩溃，而从节点还没复制到该次操作，刚才的写操作就丢失了；
- 把一次写操作视为一个事务，从事务的角度，可以认为事务被回滚了。

所以从分布式系统的角度来看，事务的提交被提升到了分布式集群的多个节点级别的“提交”，而不再是单个节点上的“提交”。

在可能发生回滚的前提下考虑脏读问题：

- 如果在一次写操作到达大多数节点前读取了这个写操作，然后因为系统故障该操作回滚了，则发生了脏读问题；

使用 {readConcern: “majority” } 可以有效避免脏读

## readConcern: 如何实现安全的读写分离

考虑如下场景：

向主节点写入一条数据；  
立即从从节点读取这条数据。

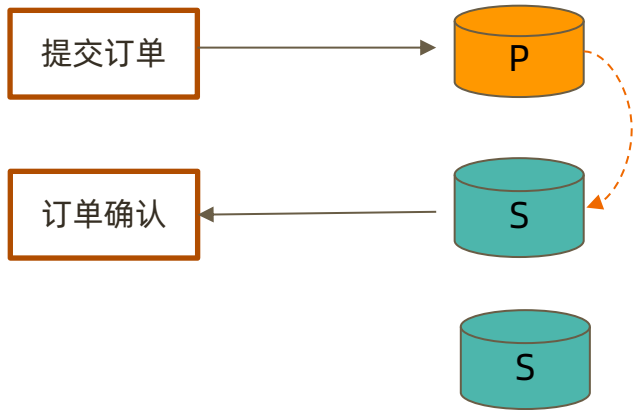
如何保证自己能够读到刚刚写入的数据？

下述方式有可能读不到刚写入的订单

```
db.orders.insert({ oid: 101, sku: "kite", q: 1})  
db.orders.find({oid:101}).readPref("secondary")
```

使用 writeConcern + readConcern majority 来解决

```
db.orders.insert({ oid: 101, sku: "kiteboard", q: 1}, {writeConcern:{w: "majority"}})  
db.orders.find({oid:101}).readPref("secondary").readConcern("majority")
```



readConcern 主要关注读的隔离性，ACID 中的 Isolation，但是是分布式数据库里特有的概念

readCocnern: majority 对应于事务中隔离级别中的哪一级？

Read Uncommitted

Read Committed

Repeatable Read

Seriazable

readConcern 主要关注读的隔离性，ACID 中的 Isolation，但是是分布式数据库里特有的概念

readCocnern: majority 对应于事务中隔离级别中的哪一级？

Read Uncommitted

**Read Committed**

Repeatable Read

Seriazable

## readConcern: linearizable

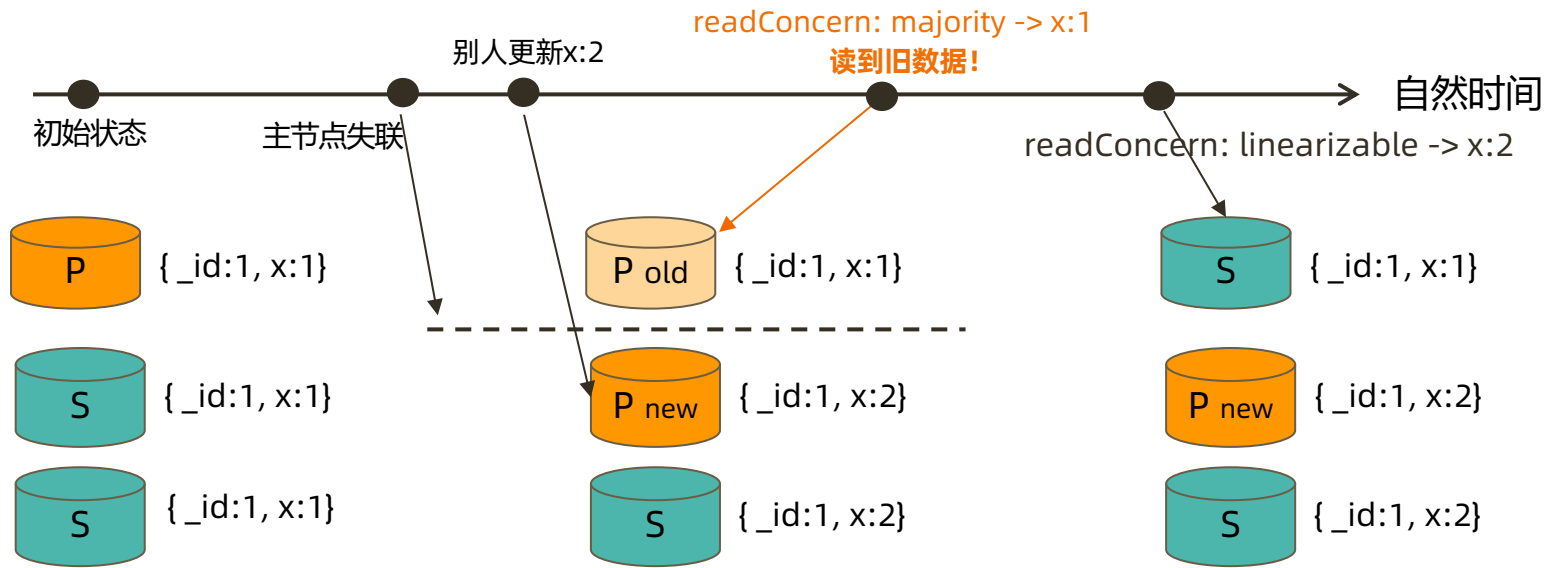
只读取大多数节点确认过的数据。和 majority 最大差别是保证绝对的操作线性顺序 - 在写操作自然时间后面发生的读，一定可以读到之前的写

- 只对读取单个文档时有效；
- 可能导致非常慢的读，因此总是建议配合使用 maxTimeMS；

## readConcern: linearizable

只读取大多数节点确认过的数据。和 majority 最大差别是保证绝对的操作线性顺序 - 在写操作自然时间后面发生的读，一定可以读到之前的写

- 只对读取单个文档时有效；
- 可能导致非常慢的读，因此总是建议配合使用 maxTimeMS；



## readConcern: snapshot

{readConcern: "snapshot" } 只在多文档事务中生效。将一个事务的 readConcern 设置为 snapshot，将保证在事务中的读：

- 不出现脏读；
- 不出现不可重复读；
- 不出现幻读。

因为所有的读都将使用同一个快照，直到事务提交为止该快照才被释放。

## readConcern: 小结

- available: 读取所有可用的数据
- local: 读取所有可用且属于当前分片的数据, 默认设置
- majority: 数据读一致性的充分保证, 可能你最需要关注的
- linearizable: 增强处理 majority 情况下主节点失联时候的例外情况
- snapshot: 最高隔离级别, 接近于 Serializable



## 第2.10讲 事务开发：多文档事务

## 开始之前.....

MongoDB 虽然已经在 4.2 开始全面支持了多文档事务，但并不代表大家应该毫无节制地使用它。相反，对事务的使用原则应该是：能不用尽量不用。

通过合理地设计文档模型，可以规避绝大部分使用事务的必要性

为什么？事务 = 锁，节点协调，额外开销，性能影响

# MongoDB ACID 多文档事务支持

事务属性	支持程度
<b>Atomocity 原子性</b>	单表单文档：1.x 就支持 复制集多表多行：4.0 复制集 分片集群多表多行4.2
<b>Consistency 一致性</b>	writeConcern, readConcern (3.2)
<b>Isolation 隔离性</b>	readConcern (3.2)
<b>Durability 持久性</b>	Journal and Replication

## 使用方法

MongoDB 多文档事务的使用方式与关系数据库非常相似：

```
try (ClientSession clientSession = client.startSession()) {  
    clientSession.startTransaction();  
    collection.insertOne(clientSession, docOne);  
    collection.insertOne(clientSession, docTwo);  
    clientSession.commitTransaction();  
}
```

## 事务的隔离级别

- 事务完成前，事务外的操作对该事务所做的修改不可访问
- 如果事务内使用 {readConcern: "snapshot" }，则可以达到可重复读  
Repeatable Read

## 实验：启用事务后的隔离性

```
db.tx.insertMany([{ x: 1 }, { x: 2 }]);  
var session = db.getMongo().startSession();  
session.startTransaction();  
var coll = session.getDatabase('test').getCollection("tx");  
coll.updateOne({x: 1}, {$set: {y: 1}});  
coll.findOne({x: 1});  
db.tx.findOne({x: 1});  
session.abortTransaction();
```

} 事务内操作 {x:1, y:1}

// 事务外的操作 {x:1}

## 实验：可重复读 Repeatabl Read

```
var session = db.getMongo().startSession();  
session.startTransaction({  
    readConcern: {level: "snapshot"},  
    writeConcern: {w: "majority"}});  
var coll = session.getDatabase('test').getCollection("tx");
```

```
coll.findOne({x: 1}); // 返回: {x: 1}
```

```
db.tx.updateOne({x: 1}, {$set: {y: 1}});
```

```
db.tx.findOne({x: 1}); // 返回: {x: 1, y: 1}
```

```
coll.findOne({x: 1}); // 返回: {x: 1}
```

```
session.abortTransaction();
```



Repeatabl Read

MongoDB 的事务错误处理机制不同于关系数据库：

- 当一个事务开始后，如果事务要修改的文档在事务外部被修改过，则事务修改这个文档时会触发 Abort 错误，因为此时的修改冲突了；
- 这种情况下，只需要简单地重做事务就可以了；
- 如果一个事务已经开始修改一个文档，在事务以外尝试修改同一个文档，则事务以外的修改会等待事务完成才能继续进行（write-wait.md 实验）。



## 实验：写冲突

继续使用上个实验的tx集合

开两个 mongo shell 均执行下述语句

```
var session = db.getMongo().startSession();  
session.startTransaction({ readConcern: {level: "snapshot"},  
                           writeConcern: {w: "majority"}});  
var coll = session.getDatabase('test').getCollection("tx");
```

窗口1:

```
coll.updateOne({x: 1}, {$set: {y: 1}});  
// 正常结束
```

窗口2:

```
coll.updateOne({x: 1}, {$set: {y: 2}});  
// 异常 - 解决方案：重启事务
```

## 实验：写冲突 (续)

窗口1：第一个事务，正常提交

```
coll.updateOne({x: 1}, {$set: {y: 1}});
```

窗口2：另一个事务更新同一条数据，异常

```
coll.updateOne({x: 1}, {$set: {y: 2}});
```

窗口3：事务外更新，需等待

```
db.tx.updateOne({x: 1}, {$set: {y: 3}});
```

## 注意事项

- 可以实现和关系型数据库类似的事务场景
- 必须使用与 MongoDB 4.2 兼容的驱动；
- 事务默认必须在 60 秒（可调）内完成，否则将被取消；
- 涉及事务的分片不能使用仲裁节点；
- 事务会影响 chunk 迁移效率。正在迁移的 chunk 也可能造成事务提交失败（重试即可）；
- 多文档事务中的读操作必须使用主节点读；
- readConcern 只应该在事务级别设置，不能设置在每次读写操作上。

## 第2.11讲 Change Stream

## 本节大纲

内容大纲	学习目标
<p>什么是 Change Stream</p> <p>Change Stream 的实现原理；</p> <p>Change Stream 的使用场景；</p> <p>Change Stream 注意事项</p>	<p>了解并学会正确使用 Change Stream</p>

## 什么是 Change Stream

Change Stream 是 MongoDB 用于实现变更追踪的解决方案，类似于关系数据库的触发器，但原理不完全相同：

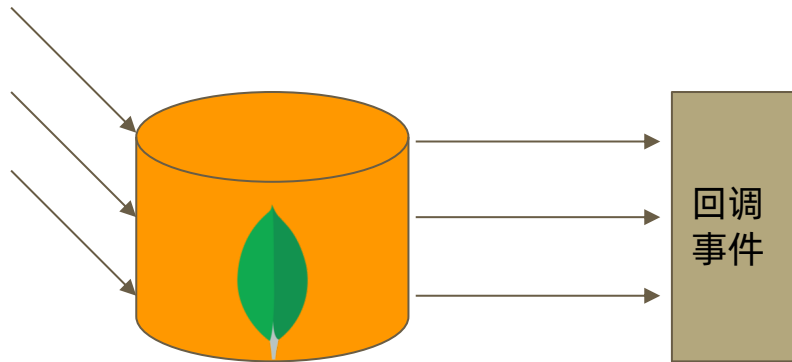
	Change Stream	触发器
触发方式	异步	同步（事务保证）
触发位置	应用回调事件	数据库触发器
触发次数	每个订阅事件的客户端	1次（触发器）
故障恢复	从上次断点重新触发	事务回滚

## Change Stream 的实现原理

Change Stream 是基于 oplog 实现的。它在 oplog 上开启一个 tailable cursor 来追踪所有复制集上的变更操作，最终调用应用中定义的回调函数。

被追踪的变更事件主要包括：

- insert/update/delete：插入、更新、删除；
- drop：集合被删除；
- rename：集合被重命名；
- dropDatabase：数据库被删除；
- invalidate：drop/rename/dropDatabase 将导致 invalidate 被触发，并关闭 change stream；



## Change Stream 与可重复读

Change Stream 只推送已经在大多数节点上提交的变更操作。即“可重复读”的变更。这个验证是通过 {readConcern: “majority”} 实现的。因此：

- 未开启 majority readConcern 的集群无法使用 Change Stream;
- 当集群无法满足 {w: “majority”} 时，不会触发 Change Stream（例如 PSA 架构中的 S 因故障宕机）。



## Change Stream 变更过滤

如果只对某些类型的变更事件感兴趣，可以使用使用聚合管道的过滤步骤过滤事件。

例如：

```
var cs = db.collection.watch([{\n  $match: {\n    operationType: {\n      $in: ['insert', 'delete']\n    }\n  }\n}])
```

## Change Stream 示例

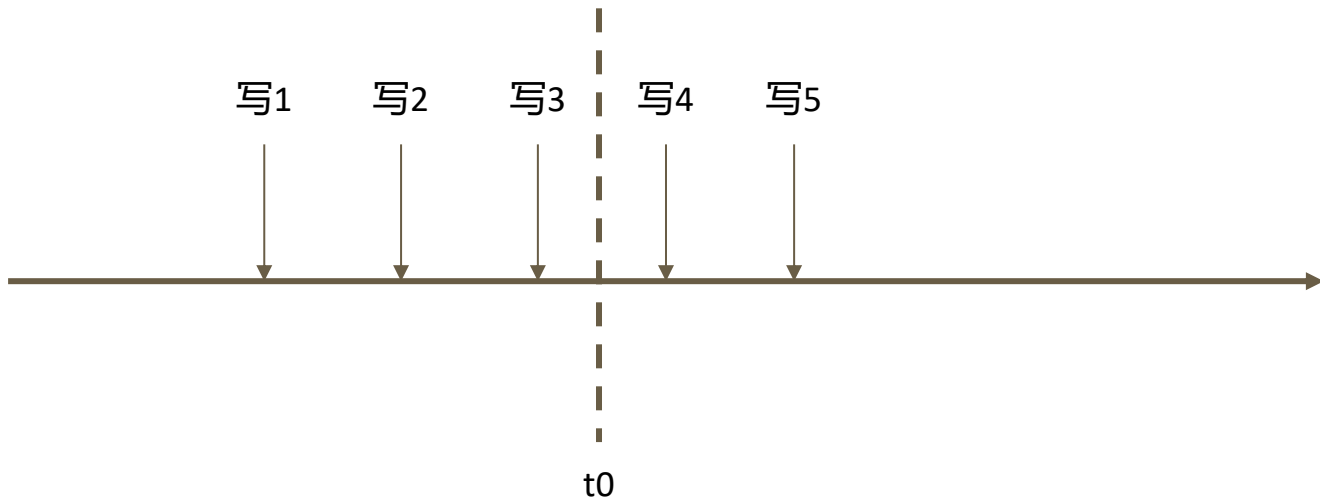
```
> db.collection.watch([],  
  {maxAwaitTimeMS: 30000}).pretty()
```

```
> db.collection.insert({  
  _id: 1,  
  text: "hello"  
})
```

```
{  
  _id : (resumeToken),  
  operationType : "insert",  
  ...  
  fullDocument : {  
    _id : 1,  
    text: "hello"  
  }  
}
```

## Change Stream 故障恢复

假设在一系列写入操作的过程中，订阅 Change Stream 的应用在接收到“写3”之后于  $t_0$  时刻崩溃，重启后后续的变更怎么办？



## Change Stream 故障恢复

想要从上次中断的地方继续获取变更流，只需要保留上次变更通知中的 `_id` 即可。

右侧所示是一次 Change Stream 回调所返回的数据。每条这样的数据都带有一个 `_id`，这个 `_id` 可以用于断点恢复。例如：

```
var cs = db.collection.watch([], {resumeAfter: <_id>})
```

即可从上一条通知中断处继续获取后续的变更通知。

```
{
  "_id": { "_data":
    BinData(0,"gl3dBycAAAABRmRfaWQAZF3dB
    ycMPJGagxioigBaEASFV+UNajtBdbj2j5fOD+3
    GBA==") },
  operationType: "insert",
  ns: {
    db: "test",
    coll: "collection"
  },
  documentKey: {
    _id: 1
  },
  fullDocument: {
    _id: 1,
    text: "hello"
  }
}
```

## Change Stream 使用场景

- 跨集群的变更复制——在源集群中订阅 Change Stream，一旦得到任何变更立即写入目标集群。
- 微服务联动——当一个微服务变更数据库时，其他微服务得到通知并做出相应的变更。
- 其他任何需要系统联动的场景。

## 注意事项

- Change Stream 依赖于 oplog，因此中断时间不可超过 oplog 回收的最大时间窗；
- 在执行 update 操作时，如果只更新了部分数据，那么 Change Stream 通知的也是增量部分；
- 同理，删除数据时通知的仅是删除数据的 `_id`。

## 第2.12讲 MongoDB 开发最佳实践

## 本章内容

关于连接到 MongoDB

关于查询及索引

关于写入操作

关于文档结构

关于事务



## 连接到 MongoDB

- 关于驱动程序：总是选择与所用之 MongoDB 相兼容的驱动程序。这可以很容易地从[驱动兼容对照表](#)中查到；
  - 如果使用第三方框架（如 Spring Data），则还需要考虑框架版本与驱动的兼容性；
- 关于连接对象 MongoClient：使用 MongoClient 对象连接到 MongoDB 实例时总是应该保证它单例，并且在整个生命周期中都从它获取其他操作对象。
- 关于连接字符串：连接字符串中可以配置大部分连接选项，建议总是在连接字符串中配置这些选项；

// 连接到复制集

mongodb://节点1,节点2,节点3.../database?[options]

// 连接到分片集

mongodb://mongos1,mongos2,mongos3.../database?[options]

## 常见连接字符串参数

- maxPoolSize
  - 连接池大小
- Max Wait Time
  - 建议设置，自动杀掉太慢的查询
- Write Concern
  - 建议 majority 保证数据安全
- Read Concern
  - 对于数据一致性要求高的场景适当使用

## 连接字符串节点和地址

- 无论对于复制集或分片集，连接字符串中都应尽可能多地提供节点地址，建议全部列出；
  - 复制集利用这些地址可以更有效地发现集群成员；
  - 分片集利用这些地址可以更有效地分散负载；
- 连接字符串中尽可能使用与复制集内部配置相同的域名或 IP；

## 使用域名连接集群

在配置集群时使用域名可以为集群变更时提供一层额外的保护。例如需要将集群整体迁移到新网段，直接修改域名解析即可。

另外，MongoDB 提供的 `mongodb+srv://` 协议可以提供额外一层的保护。该协议允许通过域名解析得到所有 mongos 或节点的地址，而不是写在连接字符串中。

```
mongodb+srv://server.example.com/
```

```
Record TTL Class Priority Weight Port Target _mongodb._tcp.server.example.com. 86400  
IN SRV 0 5 27317 mongodb1.example.com.
```

```
_mongodb._tcp.server.example.com. 86400 IN SRV 0 5 27017 mongodb2.example.com.
```

## 不要在 mongos 前面使用负载均衡

基于前面提到的原因，驱动已经知晓在不同的 mongos 之间实现负载均衡，而复制集则需要根据节点的角色来选择发送请求的目标。如果在 mongos 或复制集上层部署负载均衡：

- 驱动会无法探测具体哪个节点存活，从而无法完成自动故障恢复；
- 驱动会无法判断游标是在哪个节点创建的，从而遍历游标时出错；

结论：不要在 mongos 或复制集上层放置负载均衡器，让驱动处理负载均衡和自动故障恢复。

## 游标使用

如果一个游标已经遍历完，则会自动关闭；如果没有遍历完，则需要**手动调用 `close()`** 方法，否则该游标将在服务器上存在 10 分钟（默认值）后超时释放，造成不必要的资源浪费。

但是，如果不能遍历完一个游标，通常意味着查询条件太宽泛，更应该考虑的问题是如何将条件收紧。

## 关于查询及索引

- 每一个查询都必须要有对应的索引
- 尽量使用覆盖索引 Covered Indexes （可以避免读数据文件）
- 使用 projection 来减少返回到客户端的文档的内容

## 关于写入

- 在 update 语句里只包括需要更新的字段
- 尽可能使用批量插入来提升写入性能
- 使用TTL自动过期日志类型的数据



## 关于文档结构

- 防止使用太长的字段名（浪费空间）
- 防止使用太深的数组嵌套（超过2层操作比较复杂）
- 不使用中文，标点符号等非拉丁字母作为字段名

## 处理分页问题 - 避免使用 count

尽可能不要计算总页数，特别是数据量大和查询条件不能完全命中索引时。

考虑以下场景：假设集合总共有 1000w 条数据，在没有索引的情况下考虑以下查询：

```
db.coll.find({x: 100}).limit(50);
```

```
db.coll.count({x: 100});
```

- 前者只需要遍历前 n 条，直到找到 50 条队伍 x=100 的文档即可结束；
- 后者需要遍历完 1000w 条找到所有符合要求的文档才能得到结果。

为了计算总页数而进行的 count() 往往是拖慢页面整体加载速度的原因

## 处理分页问题 - 巧分页

避免使用skip/limit形式的分页，特别是数据量大的时候；

替代方案：使用查询条件+唯一排序条件；

例如：

第一页：db.posts.find({}).sort({\_id: 1}).limit(20);

第二页：db.posts.find({\_id: {\$gt: <第一页最后一个\_id>}}).sort({\_id: 1}).limit(20);

第三页：db.posts.find({\_id: {\$gt: <第二页最后一个\_id>}}).sort({\_id: 1}).limit(20);

.....

## 关于事务

使用事务的原则：

- 无论何时，事务的使用总是能避免则避免；
- 模型设计先于事务，尽可能用模型设计规避事务；
- 不要使用过大的事务（尽量控制在 1000 个文档更新以内）；
- 当必须使用事务时，尽可能让涉及事务的文档分布在同一个分片上，这将有效地提高效率；

## 本章小结

文档模型

读写操作的事务处理

Change Stream

开发最佳实践



扫码试看/订阅

《MongoDB 高手课》视频课程