文档数据库的事务处理

通常，事务是数据库的逻辑工作单位。只要数据库管理系统能够保证系统中一切事务的ACID特性，就能保证数据库处于一致的状态。但是，现在市面上的非关系型数据库管理系统（NoSQL数据库）大多不支持事务。以文档数据库管理系统MongoDB为例，它一开始并不支持事务，只能保证单文档上更新操作的原子性，直到MongoDB 4.2版本才开始支持多文档事务。

当数据库管理系统不支持事务时，业务系统的程序开发者就不得不在业务代码层面来保证系统故障时相关数据处理逻辑的正确性。本小节以不支持事务的文档数据库MongoDB为例介绍两种在应用层面保证数据正确性的技术。

标志位的使用

以博客博主更改自己用户名为例介绍如何使用标志位避免数据库的不一致。

该业务中包含博主和博客两类数据实体，博主和博客之间存在一对多的关系，即一个博主可以有多篇博客。数据库的设计为：每类实体对应一类文档（一个文档集），则数据库中包含博主文档集和博客文档集。博客文档集中冗余存储了博主的ID和博主名。下面展示了博主和博客的实例：

```
博主文档集user的文档实例：
{
    "id": "3333" ,
    "name": "Jason",
    "gender": "male"
}
博客文档集doc的文档实例：
{
    "id": "D1111" ,
    "uid": "3333",
    "uname": "Jason",
    "title": "The first blog"
}

{
    "id": "D222" ,
    "uid": "3333",
    "uname": "Jason",
    "title": "The second blog"
}
```

博主更改用户名的应用程序中包含两个操作：首先修改user文档集中某位博主的用户名，然后修改该博主所有博客中的用户名。以修改博主3333的用户名为Bob为例，其应用程序主要包括以下两个操作：

```
/*第一步：更新博主文档集中博主id为3333的名字*/
db.user.updateOne(
    {"id": "3333" },
    { $set: { "name": "Bob"}}
```

```
)

/*第二步：更新该博主所有博客中的用户名字*/
db.doc.updateMany(
    {"uid": "3333" },
    { $set: { "uname": "Bob"}}
)
```

假如当应用程序执行完第一个更新操作之后，发生了故障，如web服务器宕机或者数据库系统故障等，那么此时博主文档中的用户名已经修改，但是该博主的所有博客中的用户名却未修改，即数据库的状态不一致。由于数据库管理系统不支持事务，它不能保证应用程序中的两个操作要么都做要么都不做，所以仅仅依靠数据库管理系统是无法保证数据的正确性，需要在应用程序层面引入新的机制。

解决该数据异常的关键在于能够判断出故障发生时数据库的状态。数据库有三种状态：（1）博主文档和博客文档中的所有用户名已更新；（2）只有博主文档中的用户名被更新；（3）博主文档和博客文档的所有用户名未被更新。如果数据库处于第二种状态，那么当系统重启后重新执行应用程序的第二个操作，即更新该博主所有博客中的用户名。

实现判断数据库状态的方法是在博主文档中增加一个标志位属性namesync表示用户名同步状态。namesync有两个值：Done和Todo，其中Done是默认值，表示博主的用户名已经同步到所有博客中；Todo表示需要将博主的用户名同步到所有博客中。当执行修改博主用户名时，首先修改博主文档中的用户名，同时将同步标志位修改为Todo，然后修改该博主所有博客中的用户名，最后再将博主文档中的同步标志位修改为Done。新的博主文档实例及应用程序如下：

```
博主文档集user的文档实例：
{
    "id": "3333" ,
    "name": "Jason",
    "gender": "male",
    "namesync": "Done"
}
```

```
/*第一步：更新博主文档集中博主id为3333的名字，同时修改同步标志位为Todo*/
db.user.updateOne(
    {"id": "3333" },
    { $set: { "name": "Bob", "namesync": "Todo"}}
)

/*第二步：更新该博主所有博客中的用户名字*/
db.doc.updateMany(
    {"uid": "3333" },
    { $set: { "uname": "Bob"}}
)

/*第三步：修改同步标志位为Done*/
db.user.updateOne(
    {"id": "3333" },
```

```
    { $set: { "namesync": "Todo"}}
)
```

如果应用程序执行过程中发生了故障，当系统重启时，检查所有博主文档的namesync值，如果某博主文档的namesync=Done，则不用做任何操作；如果某博主文档的namesync=Todo，则重新执行应用程序的第二步和第三步，使得博客文档中的用户名与博主文档中的用户名一致，从而保证数据库的正确性。

消息队列的使用

以博主互加好友为例介绍如何使用消息队列来保证数据库的正确性。

博主互加好友应用中只涉及博主实体，博主实体中包含用户id，用户名name以及好友列表friends。下面给出两个博主的文档实例：

```
博主文档集user中的两个实例：
{
    "id": "3333" ,
    "name": "Jason",
    "friends": ["1234", "4567"]
}

{
    "id": "7777" ,
    "name": "May",
    "friends": ["1234", "6789"]
}
```

假设Jason和Jessie互相添加为好友，首先需要将Jason的用户id添加到May的好友列表中，然后再将May的用户id添加到Jason的好友列表中，其应用程序如下所示：

```
/*第一步：向Jason的好友列表中添加May的用户id7777*/
db.user.updateOne(
    {"id": "3333" },
    { $push: { "friends": "7777"}}
)

/*第二步：向May的好友列表中添加Jason的用户id3333*/
db.user.updateOne(
    {"id": "7777" },
    { $push: { "friends": "3333"}}
)
```

假如当程序执行完第一步之后发生了系统故障，那么此时May已经成为Jason的好友，但是Jason还不是May的好友。因此，数据库的状态不正确。

为了防止数据异常，可以在数据库中创建一个任务文档集。该任务文档集用于描述新的任务，包含任务id，任务状态status，任务描述content三个属性。当博主互加好友时，首先在任务文档集中新增一条互加好友的任务

文档，任务状态status的初始值为Todo，然后分别向博主的好友列表中加入新好友的用户id，最后更新该任务状态status为Done，表示添加好友任务已完成。任务文档实例以及新应用程序如下：

```
任务文档集task中的实例：
{
    "id": "T001" ,
    "status": "Todo",
    "content": "Add Friends: 3333 and 7777 become friends"
}
```

```
/*第一步：向任务文档集中添加一个互加好友的任务*/
db.task.insertOne(
    "id" : "T001",
    "status": "Todo",
    "content": "Add Friends: 3333 and 7777 become friends"
)

/*第二步：向Jason的好友列表中添加May的用户id7777*/
db.user.updateOne(
    {"id": "3333" },
    { $push: { "friends": "7777"}}
)

/*第三步：向May的好友列表中添加Jason的用户id3333*/
db.user.updateOne(
    {"id": "7777" },
    { $push: { "friends": "3333"}}
)

/*第四步：修改T001任务的状态为Done*/
db.task.updateOne(
    {"id": "T001" },
    { $set: { "status": "Done"}}
)
```

如果应用程序执行过程中发生故障，系统重启之后，检查任务文档集中所有文档实例的status值，如果某任务文档的status=Done，则不做任何操作；如果某任务文档的status=Todo，表示该任务在发生故障时没有执行完，那么需重新执行添加好友程序的第二步到第四步，最终保证了数据库的一致性。

这种添加任务文档的方法称为消息列队技术。消息队列技术广泛运用于基于微服务的互联网应用，它采用异步的方式在多个服务之间传递信息来完成业务流程。以秒杀购物为例，在购物应用服务之前添加一个消息队列用于存储用户的请求，当存储的用户请求数大于消息队列长度后则直接拒接后续的用户请求，之后秒杀购物应用从消息队列中读取用户请求，进行购物逻辑处理。在秒杀应用中，消息队列能够避免因瞬间激增的流量导致的应用服务崩溃。当然，消息队列技术并不适用于所有场景，只有当应用程序具备幂等性时，才能使用消息队列。所谓的幂等性是指应用程序无论执行多少次，它最终的结果都是一样的。