数据魅力

崇尚数据魅力，专心数据工程，数据库，数据学，Oracle，SOA，面向服务，云计算，android，移动互联网，创业，人生，感悟，哲理。

个人资料

jakenson

访问：1520次
积分：110分
排名：千里之外

原创：10篇 转载：0篇
译文：0篇 评论：0条

文章搜索

文章分类

数据集成(10)
元数据(7)
Oracle(6)
数据库(9)
数据访问(8)
信息交换(2)
语义(3)
信息集成(5)
业务对象(2)
数据中心(2)
消息(1)
soa(1)
容错(1)
内存数据库(2)
nosql(1)
mongodb(1)

文章存档

2011年12月(10)

阅读排行

关于Mongodb的全面总结，学习mon... (1036)
轻量级内存数据库研究 (75)

关于Mongodb的全面总结，学习mongodb的人，可以从这里开始！

分类： 数据集成 元数据 数据库 数据中心 内存数据库 nosql mongodb 2011-12-10 22:58 1037人阅读 评论(0) 收藏 举报

MongoDB的内部构造《MongoDB The Definitive Guide》

MongoDB的官方文档基本是how to do的介绍，而关于how it worked却少之又少，本人也刚买了《MongoDB TheDefinitive Guide》的影印版，还没来得及看，本文原作者将其书中一些关于MongoDB内部现实方面的一些知识介绍如下，值得一看。

今天下载了《MongoDB The Definitive Guide》电子版，浏览了里面的内容，还是挺丰富的。是官网文档实际应用方面的一个补充。和官方文档类似，介绍MongoDB的内部原理是少之又少，只有在附录的一个章节中介绍了相关内容。

对于大多数的MongoDB的用户来说，MongoDB就像是一个大黑盒，但是如果你能够了解到MongoDB内部一些构造的话，将有利于你更好地理解和使用MongoDB。

BSON

在MongoDB中，文档是对数据的抽象，它被使用在Client端和Server端的交互中。所有的Client端（各种语言的Driver）都会使用这种抽象，它的表现形式就是我们常说的BSON（Binary JSON ）。

BSON是一个轻量级的二进制数据格式。MongoDB能够使用BSON，并将BSON作为数据的存储存放在磁盘中。

当Client端要将写入文档，使用查询等等操作时，需要将文档编码为BSON格式，然后再发送给Server端。同样，Server端的返回结果也是编码为BSON格式再放回给Client端的。

使用BSON格式出于以下3种目的：

效率

BSON是为效率而设计的，它只需要使用很少的空间。即使在最坏的情况下，BSON格式也比JSON格式再最好的情况下存储效率高。

传输性

在某些情况下，BSON会牺牲额外的空间让数据的传输更加方便。比如，字符串的传输的前缀会标识字符串的长度，而不是在字符串的末尾打上结束的标记。这样的传输形式有利于MongoDB修改传输的数据。

性能

最后，BSON格式的编码和解码都是非常快速的。它使用了C风格的数据表现形式，这样在各种语言中都可以高效地使用。

更多关于BSON的介绍，可以参考：http://www.bsonspec.org。

基于Oracle日志分析技术的数据库消息... (41)
元数据实时构建技术的研究与实现 (35)
基于语义网关的信息集成服务 (34)
基于数据中心的业务对象访问服务的设计与实... (30)
数据库性能监控工具的设计 (27)
SOA环境下容错抗毁技术的研究 (22)
基于字段的数据交换技术 (19)
基于语义的数据集成的语义冲突问题研究 (14)

评论排行

元数据实时构建技术的研究与实现 (0)
轻量级内存数据库研究 (0)
SOA环境下容错抗毁技术的研究 (0)
基于Oracle日志分析技术的数据库消息... (0)
基于数据中心的业务对象访问服务的设计与实... (0)
基于语义的数据集成的语义冲突问题研究 (0)
基于语义网关的信息集成服务 (0)
基于字段的数据交换技术 (0)
数据库性能监控工具的设计 (0)
关于Mongodb的全面总结，学习mon... (0)

推荐文章

* PL/SQL 集合的初始化与赋值
* 从LSM-Tree两(多)组件算法谈到StackOverflow
* Eclipse中集成和调试Ant工程
* 再谈.NET Micro Framework移植
* Flex项目与SSH项目整合问题记录
* Apache MINA实战之 对象传输

写入协议

Client端访问Server端使用了轻量级的TCP/IP写入协议。这种协议在MongoDB Wiki中有详细介绍，它其实是在BSON数据上面做了一层简单的包装。比如说，写入数据的命令中包含了1个20字节的消息头（由消息的长度和写入命令标识组成），需要写入的Collection名称和需要写入的数据。

数据文件

在MongoDB的数据文件夹中（默认路径是/data/db）由构成数据库的所有文件。每一个数据库都包含一个.ns文件和一些数据文件，其中数据文件会随着数据量的增加而变多。所以如果有一个数据库名字叫做foo，那么构成foo这个数据库的文件就会由foo.ns，foo.0，foo.1，foo.2等等组成。

数据文件每新增一次，大小都会是上一个数据文件的2倍，每个数据文件最大2G。这样的设计有利于防止数据量较小的数据库浪费过多的空间，同时又能保证数据量较大的数据库有相应的空间使用。

MongoDB会使用预分配方式来保证写入性能的稳定（这种方式可以使用–noprealloc关闭）。预分配在后台进行，并且每个预分配的文件都用0进行填充。这会让MongoDB始终保持额外的空间和空余的数据文件，从而避免了数据增长过快而带来的分配磁盘空间引起的阻塞。

名字空间和盘区

每一个数据库都由多个名字空间组成，每一个名字空间存储了相应类型的数据。数据库中的每一个Collection都有各自对应的名字空间，索引文件同样也有名字空间。所有名字空间的元数据都存储在.ns文件中。

名字空间中的数据在磁盘中分为多个区间，这个叫做盘区。在下图中，foo这个数据库包含3个数据文件，第三个数据文件属于空的预分配文件。头两个数据文件被分为了相应的盘区对应不同的名字空间。

上图显示了名字空间和盘区的相关特点。每一个名字空间可以包含多个不同的盘区，这些盘区并不是连续的。与数据文件的增长相同，每一个名字空间对应的盘区大小的也是随着分配的次数不断增长的。这样做的目的是为了平衡名字空间浪费的空间与保持某一个名字空间中数据的连续性。上图中还有一个需要注意的名字空间：$freelist，这个名字空间用于记录不再使用的盘区（被删除的Collection或索引）。每当名字空间需要分配新的盘区的时候，都会先查看$freelist是否有大小合适的盘区可以使用。

内存映射存储引擎

MongoDB目前支持的存储引擎为内存映射引擎。当MongoDB启动的时候，会将所有的数据文件映射到内存中，然后操作系统会托管所有的磁盘操作。这种存储引擎有以下几种特点：

* MongoDB中关于内存管理的代码非常精简，毕竟相关的工作已经有操作系统进行托管。

* MongoDB服务器使用的虚拟内存将非常巨大，并将超过整个数据文件的大小。不用担心，操作系统会去处理这一切。

* MongoDB无法控制数据写入磁盘的顺序，这样将导致MongoDB无法实现writeahead日志的特性。所以，如果MongoDB希望提供一种durability的特性（这一特性可以参考我写的关于Cassandra文章：http://www.cnblogs.com/gpcuster/tag/Cassandra/），需要实现另外一种存储引擎。

* 32位系统的MongoDB服务器每一个Mongod实例只能使用2G的数据文件。这是由于地址指针只能支持32位。

其他

在《MongoDB The Definitive Guide》中介绍的MongoDB内部构造只有这么多，如果真要把它说清楚，可能需要另外一本书来专门讲述了。比如内部的JS解析，查询的优化，索引的建立等等。有兴趣的朋友可以直接参考源代码

MongoDB的架构

当前架构　　双服务器架构

当前架构为单shard+replica Set模式，双服务器为双Shard+Replica Set模式。同一个Shard中的primary和Secondary存储内容一致。而双Shard则是两个Shard分布式存储不同数据，备份由shard内部进行。

双服务器中的两个Shard各含一个primary，一个secondary，和一个arbiter（arbiter的唯一作用是在primary 宕机后选举新的primary时拥有投票权，用以使存活节点数大于50%，不包括50%，否则系统将整个down掉，以及在票数相同的情况下用以打破选举的平衡，并不存储和读取数据）。

因为同一个shard中，只有primary可以用以写，secondary只是用于对primary节点的备份并用于读操作，然后再primary宕机的情况下接管它的工作。所以，双shard模式下，两个服务器分别包含一个primary，而且同一个shard的arbiter必须和secondary在一个服务器上。这样子既保证了两个服务器都可以进行读、写操作，而且在primary down的时候也能够继续使得选取成功secondary。

后续扩展时，可以再在集群中添加新的shard，然后与老的shard进行balance均衡操作。

MongoDB的特点

MongoDB 是一个面向集合的,模式自由的文档型数据库.

面向集合, 意思是数据被分组到若干集合,这些集合称作聚集(collections). 在数据库里每个聚集有一个唯一的名字,可以包含无限个文档. 聚集是RDBMS中表的同义词,区别是聚集不需要进行模式定义.

模式自由, 意思是数据库并不需要知道你将存入到聚集中的文档的任何结构信息.实际上,你可以在同一个聚集中存储不同结构的文档.

文档型, 意思是我们存储的数据是键-值对的集合,键是字符串,值可以是数据类型集合里的任意类型,包括数组和文档. 我们把这个数据格式称作 "[BSON]"即 "Binary Serialized dOcument Notation."

u 面向文档存储：(类JSON数据模式简单而强大)。

u 高效的传统存储方式：支持二进制数据及大型对象（如照片和视频）。

u 复制及自动故障转移：Mongo数据库支持服务器之间的数据复制，支持主-从模式及服务器之间的相互复制。

u Auto-Sharding自动分片支持云级扩展性（处于早期alpha阶段）：自动分片功能支持水平的数据库集群，可动态添加额外的机器。

u 动态查询：它支持丰富的查询表达式。查询指令使用JSON形式的标记，可轻易查询文档中内嵌的对象及数组。

u 全索引支持：包括文档内嵌对象及数组。Mongo的查询优化器会分析查询表达式，并生成一个高效的查询计划。

u 支持RUBY，PYTHON，JAVA，C++，PHP等多种语言。

u 面向集合存储，易存储对象类型的数据：存储在集合中的文档，被存储为键-值对的形式。键用于唯一标识一个文档，为字符串类型，而值则可以是各中复杂的文件类型；

u *模式自由：存储在mongodb数据库中的文件，我们不需要知道它的任何结构定义；

u *支持完全索引，包含内部对象。

u *支持复制和故障恢复。

u *自动处理碎片: 自动分片功能支持水平的数据库集群，可动态添加额外的机器

u 查询监视：Mongo包含一个监视工具用于分析数据库操作的性能

MongoDB的功能

查询:基于查询对象或者类SQL语句搜索文档. 查询结果可以排序,进行返回大小限制,可以跳过部分结果集,也可以返回文档的一部分.

插入和更新 : 插入新文档,更新已有文档.

索引管理 : 对文档的一个或者多个键(包括子结构)创建索引,删除索引等等

常用命令: 所有MongoDB 操作都可以通过socket传输的DB命令来执行.

MongoDB的局限性与不足

本文来源于对Quora上一个问答的整理，主要列举了MongoDB身上一些局限的功能及目前做得不够好的地方。其中包括了原本就并非MongoDB想做的部分，也包括了MongoDB想做但没做好的方面。

- 在32位系统上，不支持大于2.5G的数据。详见这里
- 单个文档大小限制为 4 M/16 M（1.8版本后升为16M）
- 锁粒度太粗，MongoDB使用的是一把全局的读写锁，详见这里
- 不支持join操作和事务机制，这个确实是非MongoDB要做的领域
- 对内存要求比较大，至少要保证热数据（索引，数据及系统其它开销）都能装进内存
- 用户权限方面比较弱，这一点MongoDB官方推荐的是将机器部署在安全的内网环境中，尽量不要用权限，详见这里
- MapReduce在单个实例上无法并行，只有采用Auto-Sharding才能并行。这是由JS引擎的限制造成的
- MapReduce的结果无法写入到一个被Sharding的Collection中，2.0版本对这个问题的解决好像也不彻底
- 对于数组型的数据操作不够丰富
- Auto-Sharding还存在很多问题，所谓的水平扩展也不是那么理想

适用范围

u 适合实时的插入，更新与查询，并具备应用程序实时数据存储所需的复制及高度伸缩性。

u 适合作为信息基础设施的持久化缓存层。

u 适合由数十或数百台服务器组成的数据库。因为Mongo已经包含对MapReduce引擎的内置支持。

u Mongo的BSON数据格式非常适合文档化格式的存储及查询。

网站数据：Mongo非常适合实时的插入，更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。

u ◆缓存：由于性能很高，Mongo也适合作为信息基础设施的缓存层。在系统重启之后，由Mongo搭建的持久化缓存层可以避免下层的数据源过载。

u ◆大尺寸，低价值的数据：使用传统的关系型数据库存储一些数据时可能会比较昂贵，在此之前，很多时候程序员往往会选择传统的文件进行存储。

u ◆高伸缩性的场景：Mongo非常适合由数十或数百台服务器组成的数据库。Mongo的路线图中已经包含对MapReduce引擎的内置支持。

u ◆用于对象及JSON数据的存储：Mongo的BSON数据格式非常适合文档化格式的存储及查询

MongoDB的不适用范围

· 高度事务性的系统。

· 传统的商业智能应用。

· 级为复杂的SQL查询。

· ◆高度事务性的系统：例如银行或会计系统。传统的关系型数据库目前还是更适用于需要大量原子性复杂事务的应用程序。

· ◆传统的商业智能应用：针对特定问题的BI数据库会对产生高度优化的查询方式。对于此类应用，数据仓库可能是更合适的选择。

· ◆需要SQL的问题

u

要点

跟mysqld一样，一个mongod服务可以有建立多个数据库，每个数据库可以有多张表，这里的表名叫collection，每个collection可以存放多个文档（document），每个文档都以BSON（binary json）的形式存放于硬盘中。跟关系型数据库不一样的地方是，它是的以单文档为单位存储的，你可以任意给一个或一批文档新增或删除字段，而不会对其它文档造成影响，这就是所谓的schema-free，这也是文档型数据库最主要的优点。跟一般的key-value数据库不一样的是，它的value中存储了结构信息，所以你又可以像关系型数据库那样对某些域进行读写、统计等操作。可以说是兼备了key-value数据库的方便高效与关系型数据库的强大功能。

索引

跟关系型数据库类似，mongodb可以对某个字段建立索引，可以建立组合索引、唯一索引，也可以删除索引。当然建立索引就意味着增加空间开销，我的建议是，如果你能把一个文档作为一个对象的来考虑，在线上应用中，你通常只要对对象ID建立一个索引即可，根据ID取出对象某些数据放在memcache即可。如果是后台的分析需要，响应要求不高，查询非索引的字段即便直接扫表也费不了太多时间。如果还受不了，就再建一个索引得了。

默认情况下每个表都会有一个唯一索引：_id，如果插入数据时没有指定_id，服务会自动生成一个_id，为了充分利用已有索引，减少空间开销，最好是自己指定一个unique的key为_id，通常用对象的ID比较合适，比如商品的ID。

capped collection

capped collection是一种特殊的表，它的建表命令为：

db.createCollection("mycoll",{capped:true, size:100000})

允许在建表之初就指定一定的空间大小，接下来的插入操作会不断地按顺序APPEND数据在这个预分配好空间的文件中，如果已经超出空间大小，则回到文件头覆盖原来的数据继续插入。这种结构保证了插入和查询的高效性，它不允许删除单个记录，更新的也有限制：不能超过原有记录的大小。这种表效率很高，它适用于一些暂时保存数据的场合，比如网站中登录用户的session信息，又比如一些程序的监控日志，都是属于过了一定的时间就可以被覆盖的数据。

复制与分片

mongodb的复制架构跟mysql也很类似，除了包括master-slave构型和master-master构型之外，还有一个Replica pairs构型，这种构型在平常可以像master-slave那样工作，一但master出现问题，应用会自动了连接slave。要做复制也很简单，我自己使用过master-slave构型，只要在某一个服务启动时加上–master参数，而另一个服务加上–slave与–source参数，即可实现同步。

分片是个很头疼的问题，数据量大了肯定要分片，mysql下的分片正是成为无数DBA的噩梦。在mongodb下，文档数据库类似key-value数据库那样的易分布特性就显现出来了，无论构造分片服务，新增节点还是删除节点都非常容易实现。但mongodb在这方面做还不足够成熟，现在分片的工作还只做到alpha2版本（mongodb v1.1），估计还有很多问题要解决，所以只能期待，就不多说了。

性能

在我的使用场合下，千万级别的文档对象，近10G的数据，对有索引的ID的查询不会比mysql慢，而对非索引字段的查询，则是全面胜出。mysql实际无法胜任大数据量下任意字段的查询，而mongodb的查询性能实在让我惊

讶。写入性能同样很令人满意，同样写入百万级别的数据，mongodb比我以前试用过的couchdb要快得多，基本10分钟以下可以解决。补上一句，观察过程中mongodb都远算不上是CPU杀手。

GridFS

gridfs是mongodb一个很有趣的类似文件系统的东西，它可以用一大块文件空间来存放大量的小文件，这个对于存储web2.0网站中常见的大量小文件（如大量的用户头像）特别有效。使用起来也很方便，基本上跟一般的文件系统类似。

用合适的数据库做适合的事情

mongodb的文档里提到的user case包括实时分析、logging、全文搜索，国内也有人使用mongodb来存储分析网站日志，但我认为mongodb用来处理有一定规模的网站日志其实并不合适，最主要的就是它占空间过于虚高，原来1G的日志数据它可以存成几个G，如此下去，一个硬盘也存不了几天的日志。另一方面，数据量大了肯定要考虑sharding，而mongodb的sharding到现在为止仍不太成熟。由于日志的不可更新性的，往往只需APPEND即可，又因为对日志的操作往往只集中于一两列，所以最合适作为日志分析的还是列存储型的数据库，特别是像infobright那样的为数据仓库而设计的列存储数据库。

由于mongodb不支持事务操作，所以事务要求严格的系统（如果银行系统）肯定不能用它。

MongoDB分布式复制

一、主从配置（Master Slave）

主从数据库需要两个数据库节点即可，一主一从（并不一定非得两台独立的服务器，可使用--dbpath参数指定数据库目录）。一个从节点可以有多个主节点，这种情况下，`local.sources`中会有多条配置信息。一台服务器可以同时即为主也为从。如果一台从节点与主节点不同步，比如从节点的数据更新远远跟不上主节点或者从节点中断之后重启但主节点中相关的数据更新日志却不可用了。这种情况下，复制操作将会终止，需要管理者的介入，看是否默认需要重启复制操作。管理者可以使用{resync:1} 命令重启复制操作，可选命令行参数 `--autoresync`可使从节点在不同步情况发生10秒钟之后，自动重启复制操作。如果指定了`--autoresync`参数，从节点在10分钟以内自动重新同步数据的操作只会执行一次。
`--oplogSize`命令行参数（与`--master`一同使用）配置用于存储给从节点可用的更新信息占用的磁盘空间（M为单位），如果不指定这个参数，默认大小为当前可用磁盘空间的5%（64位机器最小值为1G，32位机器为50M）。

二、互为主从（Replica Pairs）

数据库自动协调某个时间点上的主从关系。开始的时候，数据库会判断哪个是从哪个是主，一旦主服务器负载过高，另一台就会自动成为主服务器。
*remoteserver*组中的其他服务器host，可加:port指定端口。
arbiterserver 仲裁（arbiter ）的host，也可指定端口。仲裁是一台mongodb服务器，用于协助判断某个时间点上的数据库主从关系。如果同组服务器在同一个交换机或相同的ec2可用区域内，就没必要使用仲裁了。如果同组服务器之间不能通信，可是使用运行在第三方机器上的仲裁，使用“抢七”方式有效地敲定主服务器，也可不使用仲裁，这样所有的服务器都假定是主服务器状态，可通过命令人工检测当前哪台数据库是主数据库：

```
$ ./mongo
> db.$cmd.findOne({ismaster:1});
{ "ismaster" : 0.0 , "remote" : "192.168.58.1:30001" , "ok" : 1.0 }
```

一致性：故障转移机制只能够保障组中的数据库上的数据的最终一致性。如果机器L是主服务器，然后挂了，那么发生在它身上的最后几秒钟的操作信息就到达不了机器R，那么机器R在机器L恢复之前是不能执行这些操作的。
安全性：同主从的操作相同。
数据库服务器替换。当一台服务器失败了，系统能自动在线恢复。但当一台机器彻底挂了，就需要替换机器，而替换机器一开始是没有数据的，怎么办？以下会解释如何替换一组服务器中的一台机器。

MongoDB语法与现有关系型数据库SQL语法比较

MongoDB语法	MySql语法
db.test.find({'name':'foobar'})	<==> select * from test where name='foobar'
db.test.find()	<==> select *from test
db.test.find({'ID':10}).count()	<==> select count(*) from test where ID=10

db.test.find().skip(10).limit(20)<==> select * from test limit 10,20

db.test.find({'ID':{$in:[25,35,45]}})<==> select * from test where ID in (25,35,45)

db.test.find().sort({'ID':-1}) <==> select * from test order by IDdesc

db.test.distinct('name',{'ID':{$lt:20}}) <==> select distinct(name) from testwhere ID<20

db.test.group({key:{'name':true},cond:{'name':'foo'},reduce:function(obj,prev)
{prev.msum+=obj.marks;},initial:{msum:0}}) <==> select name,sum(marks) from testgroup by name

db.test.find('this.ID<20',{name:1}) <==> select name from test whereID<20

db.test.insert({'name':'foobar','age':25})<==>insertinto test ('name','age') values('foobar',25)

db.test.remove({}) <==> delete * from test

db.test.remove({'age':20}) <==> delete test where age=20

db.test.remove({'age':{$lt:20}}) <==> elete test where age<20

db.test.remove({'age':{$lte:20}}) <==> delete test where age<=20

db.test.remove({'age':{$gt:20}}) <==> delete test where age>20

db.test.remove({'age':{$gte:20}})<==> delete test where age>=20

db.test.remove({'age':{$ne:20}}) <==> delete test where age!=20

db.test.update({'name':'foobar'},{$set:{'age':36}})<==> update test set age=36 where name='foobar'

db.test.update({'name':'foobar'},{$inc:{'age':3}})<==> update test set age=age+3 where name='foobar'

Mongodb亿级数据量的性能测试

进行了一下Mongodb亿级数据量的性能测试，分别测试如下几个项目：
（所有插入都是单线程进行，所有读取都是多线程进行）
1） 普通插入性能 （插入的数据每条大约在1KB左右）
2） 批量插入性能 （使用的是官方C#客户端的InsertBatch），这个测的是批量插入性能能有多少提高
3） 安全插入功能 （确保插入成功，使用的是SafeMode.True开关），这个测的是安全插入性能会差多少
4） 查询一个索引后的数字列，返回10条记录（也就是10KB）的性能，这个测的是索引查询的性能
5） 查询两个索引后的数字列，返回10条记录（每条记录只返回20字节左右的2个小字段）的性能，这个测的是返回小数据量以及多一个查询条件对性能的影响
6） 查询一个索引后的数字列，按照另一个索引的日期字段排序（索引建立的时候是倒序，排序也是倒序），并且Skip100条记录后返回10条记录的性能，这个测的是Skip和Order对性能的影响
7） 查询100条记录（也就是100KB）的性能（没有排序，没有条件），这个测的是大数据量的查询结果对性能的影响
8） 统计随着测试的进行，总磁盘占用，索引磁盘占用以及数据磁盘占用的数量

MongoDB CEO：NoSQL的大数据量处理能力

为MongoDB提供技术支持的10gen公司CEO凯文-赖安Dwight Merriman说：“我们公司成立于3月29日，我认为我们选择的不是一个缝隙市场，相反，我认为我们会慢慢改变企业用户市场。现在我们可以看到，MongoDB.org网站每月的下载量达到了3万次，而几个月前，下载量还为零”。

10gen公司CEO Dwight Merriman

MongoDB的名字源自一个形容词humongous（巨大无比的），在向上扩展和快速处理大数据量方面，它会损失一些精度，在旧金山举行的MondoDB大会上，Merriman说：“你不适宜用它来处理复杂的金融事务，如证券交易，数据的一致性可能无法得到保证”。

NoSQL数据库都被贴上不同用途的标签，如MongoDB和CouchDB都是面向文档的数据库，但这并不意味着它们可以象JSON（JavaScript ObjectNotation，JavaScript对象标记）那样以结构化数据形式存储文本文档。

JSON被认为是XML的代替品，它是一个轻量级的，基于文本交换数据的标准，和XML一样具有人类易读的特性。简单的JSON数据结构叫做对象，可能包括多种数据类型，如整型（int），字符串（string），数组（array），日期（date），对象（object）和字节数组（bytearray）。

面向文档的数据库与关系数据库有着显著的区别，面向文档的数据库用一个有组织的文件来存储数据，而不是用行来存储数据，在MongoDB中，一组文档被看作是一个集合，在关系数据库中，许多行的集合被看作是一张表。

但同时它们的操作又是类似的，关系数据库使用select，insert，update和delete操作表中的数据，面向文档的数据库使用query，insert，update和remove做意义相同的操作。

MongoDB中对象的最大尺寸被限制为4MB，但对象的数量不受限制，MongoDB可以通过集群加快操作的执行速度，当数据库变得越来越大时，可以向集群增加服务器解决性能问题。

Wordnik工程副总裁Tony Tam说他的公司有5百万个文档，以前保存在MySQL数据库中，大约有1.5TB，一个月前迁移到MongoDB上了，Wordnik专门收集所有单词的定义和信息，因此数据量是非常大的，迁移到MongoDB后，Tony Tam说他感到更放心。

Tam说使用MySQL数据库时，Wordnik项目一直都象是在颠簸的路上前行，数据表的冻结时间有时甚至超过了10秒，这是任何人都不能容忍的。每天会有大约200个新单词出现，我们要负责收集，并要向数据库增加1500个例子显示它们的用法，我们希望写入数据库的时间只需要1秒。Tam说：“我们不关心一致性，前后两个用户的查询结果不一定非得保持一致，我们本来就是时刻在做着更新，这一点我们无法保证”。

Wordnik系统就象是一个庞大的在线词典，有很多人同时在线查询，但同时我们也在做更新，使用MongoDB后，我们可以保持高速添加数据，不用担心数据库会出现堵塞。Tam在MondoDB大会上曾做过一个题为“Wordnik：从MySQL到MongoDB”的演讲，他说他们公司只花了一天时间就从MySQL迁移到MongoDB上了。

延伸阅读

Mongo是一个高性能，开源，无模式的文档型数据库，它在许多场景下可用于替代传统的关系型数据库或键/值存储方式。Mongo使用C++开发，提供了以下功能：

◆面向集合的存储：适合存储对象及JSON形式的数据。

◆动态查询：Mongo支持丰富的查询表达式。查询指令使用JSON形式的标记，可轻易查询文档中内嵌的对象及数组。

◆完整的索引支持：包括文档内嵌对象及数组。Mongo的查询优化器会分析查询表达式，并生成一个高效的查询计划。

◆查询监视：Mongo包含一个监视工具用于分析数据库操作的性能。

◆复制及自动故障转移：Mongo数据库支持服务器之间的数据复制，支持主-从模式及服务器之间的相互复制。复制的主要目标是提供冗余及自动故障转移。

◆高效的传统存储方式：支持二进制数据及大型对象（如照片或图片）。

◆自动分片以支持云级别的伸缩性（处于早期alpha阶段）：自动分片功能支持水平的数据库集群，可动态添加额外的机器。

MongoDB的主要目标是在键/值存储方式（提供了高性能和高度伸缩性）以及传统的RDBMS系统（丰富的功能）架起一座桥梁，集两者的优势于一身。根据官方网站的描述，Mongo适合用于以下场景：

◆网站数据：Mongo非常适合实时的插入，更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。

◆缓存：由于性能很高，Mongo也适合作为信息基础设施的缓存层。在系统重启之后，由Mongo搭建的持久化缓存层可以避免下层的数据源过载。

◆大尺寸，低价值的数据：使用传统的关系型数据库存储一些数据时可能会比较昂贵，在此之前，很多时候程序员往往会选择传统的文件进行存储。

◆高伸缩性的场景：Mongo非常适合由数十或数百台服务器组成的数据库。Mongo的路线图中已经包含对MapReduce引擎的内置支持。

◆用于对象及JSON数据的存储：Mongo的BSON数据格式非常适合文档化格式的存储及查询。

自然，MongoDB的使用也会有一些限制，例如它不适合：

◆高度事务性的系统：例如银行或会计系统。传统的关系型数据库目前还是更适用于需要大量原子性复杂事务的应用程序。

◆传统的商业智能应用：针对特定问题的BI数据库会对产生高度优化的查询方式。对于此类应用，数据仓库可能是更合适的选择。

◆需要SQL的问题

MongoDB支持OS X、Linux及Windows等操作系统，并提供了Python，PHP，Ruby，Java及C++语言的驱动程序，社区中也提供了对Erlang及.NET等平台的驱动程序。(

漫画：MongoDB身上的优势和劣势

SQL or NoSQL？That's a question!SQL 与 NoSQL 的争论从来没有停息过，但其实任何一种技术都不会是适合一切应用场景的，重要的是你要充分了解自己的需求，再充分了解你要选择的技术的优劣。

下面是一个关于 MongoDB 优缺点的列表，希望对打算使用 MongoDB 的同学，能有一些作用：

优势：

快速！（当然，这和具体的应用方式有关，通常来说，它比一般的关系型数据库快5位左右。）

很高的可扩展性 – 轻轻松松就可实现PB级的存储（但是可能我们并不需要PB级的存储，10TB可能就够了）

他有一个很好的 replication 模式 (replica sets)

有很完善的Java API

他的存储格式是Json的，这对Java来说非常好处理，对javascirpt亦然。

运维起来非常方便，你不用专门为它安排一个管理员。

它有一个非常活跃的社区（我提出的一个bug在20分钟内就能得到修复。多谢Elliot）

他的版本控制非常清楚。

MongoDB 背后的公司（10gen）已经准备好了明天在 MongoDB 上面的投入的资金了。

劣势：

应用经验缺乏，我们都没有相关NoSQL 产品的使用经验。

项目相对来说还比较新。

和以往的存储相比，数据的关系性操作不再存在。

另附趣图一张：

-

详细分析Memcached缓存与Mongodb数据库的优点与作用

本文详细讲下Memcached和Mongodb一些看法，以及结合应用有什么好处，希望看到大家的意见和补充。

Memcached

Memcached的优势我觉得总结下来主要体现在：

1） 分布式。可以由10台拥有4G内存的机器，构成一个40G的内存池，如果觉得还不够大可以增加机器，这样一个大的内存池，完全可以把大部分热点业务数据保存进去，由内存来阻挡大部分对数据库读的请求，对数据库释放可观的压力。

2） 单点。如果Web服务器或App服务器做负载均衡的话，在各自内存中保存的缓存可能各不相同，如果数据需要同步的话，比较麻烦（各自自己过期，还是分发数据同步？），即使数据并不需要同步，用户也可能因为数据的不一致而产生用户体验上的不友好。

3） 性能强。不用怀疑和数据库相比确实是，根源上还是内存的读写和磁盘读写效率上几个数量级的差距。有的时候我们在抱怨数据库读写太差的情况下可以看看磁盘的IO，如果确实是瓶颈的话装啥强劲的数据库估计也档不了，强不强无非是这个数据库多少充分的利用了内存。

但是也不太建议在任何情况下使用Memcached替代任何缓存：

1） 如果Value特别大，不太适合。因为在默认编译下Memcached只支持1M的Value（Key的限制到不是最大的问题）。其实从实践的角度来说也 不建议把非常大的数据保存在Memcached中，因为有序列化反序列化的过程，别小看它消耗的CPU。说到这个就要提一下，我一直觉得 Memcached适合面向输出的内容缓存，而不是面向处理的数据缓存，也就是不太适合把大块数据放进去拿出来处理之后再放进去，而是适合拿出来就直接给输出了，或是拿出来不需要处理直接用。

2） 如果不允许过期，不太适合。Memcached在默认情况下最大30天过期，而且在内存达到使用限制后它也会回收最少使用的数据。因此，如果我们要把它当 作static变量的话就要考虑到这个问题，必须有重新初始化数据的过程。其实应该这么想，既然是缓存就是拿到了存起来，如果没有必定有一个重新获取重新缓存的过程，而不是想着它永远存在。

在使用Memcached的过程中当然也会有一些问题或者说最佳实践：

1） 清除部分数据的问题。Memcached只是一个Key/Value的池，一个公共汽车谁都可以上。我觉得对于类似的公共资源，如果用的人都按照自己的规 则来的话很容易出现问题。因此，最好在Key值的规范上上使用类似命名空间的概念， 每一个用户都能很明确的知道某一块功能的Key的范围，或者说前缀。带来的好处是我们如果需要清空的话可以根据这个规范找到我们自己的一批Key然后再去 清空，而不是清空所有的。当然有人是采用版本升级的概念，老的Key就让它过去吧，到时候自然会清空，这也是一种办法。不过Key有规范总是有好处的，在统计上也方便一点。

2） Value的组织问题。也就是说我们存的数据的粒度，比如要保存一个列表，是一个保存在一个键值还是统一保存为一个键值，这取决于业务。如果粒度很小的话最好是在获取的时候能批量获取，在保存的时候也能批量保存。对于跨网络的调用次数越少越好，可以想一下，如果一个页面需要输出100行数据，每一个数据都需要获取一次，一个页面进行上百次连接这个性能会不会成问题。

那么Memcached主要用在哪些功能上呢？

其实我觉得平时能想到在内存中做缓存的地方我们都可以考虑下是不是可以去适用分布式缓存，但是主要的用途还是用来在前端或中部挡一下读的需求来释放Web服务器App服务器以及DB的压力。

下面讲讲**Mongodb。**

Mongodb

Mongodb是一款比较优良的非关系型数据库的文档型的数据库。它的优势主要体现在：

1） 开源。意味着即使我们不去改也可以充分挖掘它，MS SQL除了看那些文档，谁又知道它内部如何实现。

2） 免费。意味着我们可以在大量垃圾服务器上装大量的实例，即使它性能不怎么高，也架不住非常多的点啊。

3） 性能高。其它没比较过，和MS SQL相比，同样的应用（主要是写操作）一个撑500用户就挂了，一个可以撑到2000。在数据量上到百万之后，即使没索引，MS SQL的插入性能下降的也一塌糊涂。其实任何事物都有相对性的，在变得复杂变得完善了之后会牺牲一部分的性能，MS SQL体现的是非常强的安全性数据完整性，这点是Mongodb办不到的。

4） 配置简单并且灵活。在生产环境中对数据库配置故障转移群集和读写分离的数据库复制是很常见的需求，MS SQL的配置繁琐的步骤还是很恐怖的，而Mongodb可以在五分钟之内配置自己所需要的故障转移组，读写分离更是只需要一分钟。灵活性体现在，我们可以配置一个M一个S，两个M一个S（两个M写入的数据会合并到S上供读取），一个M两个S（一个M写入的数据在两个S上有镜像），甚至是多个M多个S（理论上可以创建10个M，10个S，我们只需要通过轮询方式随便往哪个M上写，需要读的时候也可以轮训任意一个S，当然我们要知道不可能保证在同一时间所有的 S都有一致的数据）。那么也可以配置两个M的对作为一套故障转移群集，然后这样的群集配置两套，再对应两个S，也就是4个M对应2个S，保证M点具有故障 转移。

5） 使用灵活。在之前的文章中我提到甚至可以通过SQL到JS表达式的转换让Mongodb支持SQL语句的查询，不管怎么说Mongodb在查询上还是很方便的。

之前也说过了，并不是所有数据库应用都使用采用Mongodb来替代的，它的主要缺点是：

1） 开源软件的特点：更新快，应用工具不完善。由于更新快，我们的客户端需要随着它的更新来升级才能享受到一些新功能，更新快也意味着很可能在某一阶段会缺乏某个重要功能。另外我们知道MS SQL在DEV/DBA/ADM多个维度都提供了非常好的GUI工具对数据库进行维护。而Mongodb虽然提供了一些程序，但是并不是非常友好。我们的 DBA可能会很郁闷，去优化Mongodb的查询。

2） 操作事务。Mongodb不支持内建的事务（没有内建事务不意味着完全不能有事务的功能），对于某些应用也就不适合。不过对于大部分的互联网应用来说并不存在这个问题。

在使用Mongodb的过程中主要遇到下面的问题：

1） 真正的横向扩展？在使用Memcached的过程中我们已经体会到这种爽了，基本可以无限的增加机器来横向扩展，因为什么，因为我们是通过客户端来决定键值保存在那个实例上，在获取的时候也很明确它在哪个实例上，即使是一次性获取多个键值，也是同样。而对于数据库来说，我们通过各种各样的方式进行了 Sharding，不说其它的，在查询的时候我们根据一定的条件获取批量的数据，怎么样去处理？比如我们按照用户ID去分片，而查询根本不在乎用户ID， 在乎的是用户的年龄和教育程度，最后按照姓名排序，到哪里去取这些数据？不管是基于客户端还是基于服务端的Sharding都是非常难做的，并且即使有了 自动化的Sharding性能不一定能有保障。最简单的是尽量按照功能来分，再下去就是历史数据的概念，真正要做到实时数据分散在各个节点，还是很困难。

2） 多线程，多进程。在写入速度达不到预期的情况下我们多开几个线程同时写，或者多开几个Mongodb进程（同一机器），也就是多个数据库实例，然后向不同 的实例去写。这样是否能提高性能？很遗憾，非常有限，甚至可以说根本不能提高。为什么使用Memcached的时候多开线程可以提高写入速度？那是因为内 存数据交换的瓶颈我们没达到，而对于磁盘来说，IO的瓶颈每秒那么几十兆的是很容易达到的，一旦达到这个瓶颈了，无论是开多少个进程都无法提高性能了。还 好Mongodb使用内存映射，看到内存使用的多了，其实我对它的信心又多了一点（内存占用多了我觉得CPU更容易让它不闲着），怕就怕某个DB不使用什 么内存，看着IO瓶颈到了，内存和CPU还是吃不饱。

Memcached和Mongodb的配合

其实有了Memcached和Mongodb我们甚至可以让80%以上的应用摆脱传统关系型数据库。我能想到它们其实可以互相配合弥补对方的不足：

Memcached适合根据Key保存Value，那么有的时候我们并不知道需要读取哪些Key怎么办呢？我在想是不是可以把Mongodb或 说数据库当作一个原始数据，这份原始数据中分为需要查询的字段（索引字段）和普通的数据字段两部分，把大量的非查询字段保存在Memcached中，小粒 度保存，在查询的时候我们查询数据库知道要获取哪些数据，一般查询页面也就显示20-100条吧，然后一次性从Memcached中获取这些数据。也就是 说，Mongodb的读的压力主要是索引字段，而数据字段只是在缓存失效的时候才有用，使用Memcached挡住大部分实质数据的查询。反过来说，如果我们要清空Memcached中的数据也知道要清空哪些Key。

-

MongoDB 文档阅读笔记 —— 优雅的 NoSQL

NoSQL 数据库在上年炒得很热，于是我也萌生了使用 NoSQL 数据库写一个应用的想法。首先来认识一下 NoSQL。NoSQL 是一个缩写，含义从最初的 No-SQL 到现在已经成为了 Not-Only-SQL。确实后面一种解释比较符合 NoSQL 的使用场景。

现在网络上被人所知的 NoSQL 数据库可以在这个网页（http://nosql-database.org）看到。这个列表林林总总一大堆，要选择哪个数据库入手呢？

1. 选择非关系数据库

在我关注的 Web 领域，特别是 Ruby on Rails 社区，比较多提到的是这几个数据库：

Cassandra， apache基金会下的非关系数据库。早前一段时间传言 Twitter 要用 Cassandra 替代 Mysql，一时间坊间流传“NoSQL 要革 SQL 的命了！”。不过Twitter 博客澄清，Twitter 只是在部分领域使用 Cassandra，存放 Tweets 的主数据库依然是 MySQL。

MongoDB，10gen 公司的开源非关系数据库产品，可以选择他们公司的商业支持。RoR 相关的插件挺多。

CouchDB，另一个apache基金会下的非关系数据库。

Redis，特点是运行在内存中，速度很快。相比于用来持久化数据，也许更接近于 memcached 这样的缓存系统，或者用来实现任务队列。（比如resque）

在这些候选名单中我选择了MongoDB。因为它最近在 RoR 社区中的露脸率比较高，网页文档完善，并且项目主页的设计也不错

在陈述 MongoDB 的特性之前，还是给第一次接触 NoSQL 的人提个醒：不要意图用 NoSQL 全盘取代 SQL 数据库。非关系数据库的出现不是为了取代关系数据库。具体的说，MongoDB 并不支持复杂的事务，只支持少量的原子操作，所以不适用于“转帐”等对事务和一致性要求很高的场合。而 MongoDB 适合什么场合，请继续阅读。

2. 文档型数据库初探

关系数据库比如 MySQL，通常将不同的数据划分为一个个“表”，表的数据是按照“行”来储存的。而关系数据库的“关系”是指通过“外键”将表间或者表内的数据关联起来。比如文章-评论 的一对多关系可以用这样的表来实现：

```
posts(id, author_id, content, ... )

comments(id, name, email, web_site, content, post_id)
```

实现关联的关键就是 comments 表的最后一个 post_id 字段，将 comment 数据的 post_id 字段设为评论目标文章的 id 值，就可以用 SQL 语句进行相关查询（假设要查的文章 id 是 1）：

```
SELECT * FROM comments WHERE post_id = 1;
```

相对于关系数据库的行式储存和查询，MongoDB 作为一个文档型数据库，可以支持更具层次感的数据。上面举的 文章-评论 结构，在 MongoDB 里面可以这样设计。

```
{
  _id : ObjectId(...),
  author : 'Rei',
  content : 'content text',
  comments : [ { name : 'Asuka'
                 email : '...',
                 web_site : '...',
                 content : 'comment text'} ]
}
```

comments 项是内嵌在 post 项中的（作为数组）。在 MongoDB 中，一个数据项叫做 Document，一个文档嵌入另一个文档（comment 嵌入 post）叫做 Embed，储存一系列文档的地方叫做 Collections。顺便一提，MongoDB 中也提供类似 SQL 数据库中的表间关联，叫做 Reference。

3. 用文档型数据库储存文档

可以看到，文档性数据库从储存的数据项上就跟 SQL 数据库不同。在 MongoDB 中，文档是以 BSON 格式（类似 JSON）储存的，可以支持丰富的层次的结构。由于数据结构的表达能力更强，用 MongoDB 储存文档型数据可以比 SQL 数据库更直观和高效。

3.1简化模式设计

在 SQL 数据库中，为表达数据的从属关系，通常要将表间关系分为 one-to-one，one-to-many，many-to-many 等模式进行设计，这通常会需要很多链接表的辅助。在MongoDB 中，如果关联文档体积较小，固定不变，并且与另一文档是主从关系，那么通常可以嵌入（Embed）主文档。

常见情景：评论、投票点击数据、Tag。

这类场景的有时单个数据项体积很小，但是数量巨大，与之相应的是查询成本也会上升。如果将这些小数据嵌入所属文档，在查询主文档时一并提取，查询效率要比 SQL 高，后者通常需要开支较大的 JOIN 查询。并且根据文档介绍，每个文档包括 Embed 部分在物理硬盘上都是储存在同一区域的，IO 部分也会比 SQL 数据库快。（注，MongoDB 有单文档大小限制）

3.2动态的文档模式

MongoDB 中的文档其实是没有模式的，不像 SQL 数据库那样在使用前强制定义一个表的每个字段。这样可以避免对空字段的无谓开销。

例如两个用户的联系信息，A 用户带有email 不带 url，B 用户带有 url 不带 email。在 SQL 数据库中需要为两个数据项都提供 email 段和 url 段，而在MongoDB 中可以这样保存：

```
[ { name : 'A', email : 'A email address' }, { :name : 'B', url : 'B url address' } ]
```

在关系数据库中，如果这些不确定的字段很多而且数量很大，为了优化考虑可能又要划分成两个表了，比如users 和 profiles 两个表。在 MongoDB 中，如果这一类不确定信息确实是属于同一文档的，那么可以轻松的放在一起，因为并不需要预先定义模式，也不会有空字段的开销。

不过因为要支持这样的动态性，并且为了性能考虑进行预先分配硬盘空间，数据外的开销也会带来磁盘占用。我还未实测实际中开销有多大，也许未来不久我会写一个应用测试。

4.MongoDB 另外一些特点

4.1JSON 文档式查询

MongoDB 的查询语言看起来是这样的：

```
>db.users.find( { x : 3, y : "abc" } ).sort({x:1});
```

这是在 MongoDB 内置的JavaScript 控制台上的查询，表示在名为 users 的 Collections 中查找 x = 3，y = “abc” 的文档，并且以 x 递增的顺序返回数据。

JSON 文档式查询会让写惯应用层代码的开发者眼前一亮，但对于精通 SQL 查询的关系数据库管理员来说就是一个新的挑战了。

4.2对分布式的支持

MongoDB 对大型网站的最大吸引力也许来源于其对分布式部署的支持。当今互联网最流行的数据库 MySQL 在网站扩大到一定规模之后就会遇到扩展瓶颈，解决方案通常是分表分库、配置主从数据库。很多互联网开拓者前仆后继的为数据库扩展性奋斗，留下了一页页的宝贵经验。

既然年复一年的有人为同一个问题奋斗，为什么不将这些问题在数据库层面就解决了呢？而MongoDB 的优势之一就是内建对分布式的支持。

不过我没有组建数据库群集的需求，所以还未阅读这方面的文档。

5. 总结

没有银弹，这个教诲已经提过太多。由于缺乏对事务的支持，MongoDB 不太适用于金融等行业的关键部分（我想也没有这个人群来看我博客吧）。但对于现在要应对海量细信息的 web 网站来说，MongoDB 可能恰好出现在了正确的时代。NoSQL 的无模式，能让网站开发的迭代更轻盈；MongoDB 对分布式的支持，可以缓解网站快速成长时在数据库端的瓶颈疼痛。

不要激进的用 NoSQL 替代所有 MySQL应用，激进的 NoSQL 化只会像5年前的 all rewrite by RoR 浪潮一样，耗费不必要的精力。但在新项目开发之初，可以考虑是否更适合使用 NoSQL。而我，已经打算在下一个 web 项目里面使用 MongoDB。

通过 MongoDB 推动 NoSQL（第1部分）

自从 2000 年宣布 Microsoft.NET Framework 并在 2002 年首次发行以来的过去近十年中，.NET开发人员一直努力适应 Microsoft 推出的各种新事物。但这似乎还不够，“社区”（包含所有开发人员，无论他们是否每天都使用.NET）也开始行动，创造出更多的新事物来填补Microsoft 未覆盖到的空白，对您而言这可能是制造混乱和干扰。

在 Microsoft 的 *支持* 范围之外，该社区所酝酿出的“新”事物之一就是 NoSQL 运动，一组开发人员公开质疑将所有数据存储于某种形式的关系数据库系统的这种观念。表、行、列、主键、外键约束、关于null 的争论以及有关主键是否应该为自然键或非自然键的辩论……还有什么是神圣不可侵犯的？

在本文及其后续文章中，我将探讨NoSQL 运动所倡导的主要工具之一：MongoDB，根据 MongoDB 网站的陈述，该工具的名称源自于“hu*mongo*us”（并不是我杜撰的）。我基本上会讨论到与 MongoDB 相关的方方面面：安装、浏览以及在.NET Framework 中使用 MongoDB。其中包括其提供的 LINQ 支持；在其他环境（桌面应用程序和Web 应用程序及服务）中使用MongoDB；以及如何设置MongoDB，以免 Windows 生产管理员向您提出严重抗议。

问题（或者，为何我要再次关注？）

在深入了解MongoDB 之前，读者自然要问为什么.NET Framework 开发人员应该牺牲接下来宝贵的大约半小时时间继续待在电脑前阅读本文。毕竟，SQLServer 有免费、可再发行的Express Edition，提供比企业或数据中心绑定的传统关系数据库更精简的数据存储方案，而且毫无疑问，还可以使用大量工具和库来轻松访问SQL Server 数据库，其中包括Microsoft 自己的 LINQ 和实体框架。

但问题在于，关系模型（指关系模型本身）的优点也是其最大的缺点。大多数开发人员（无论是.NET、Java 还是其他开发人员都在此列）在经历短短几年的开发工作之后，就会一一痛诉这种表/行/列的“方正”模型如何不能令其满意。尝试对分层数据进行建模的举动甚至能让最有经验的开发人员完全精神崩溃，类似情况不甚枚举，因此 Joe Celko 还写过一本书《SQLfor Smarties, Third Edition》（Morgan-Kaufmann，2005），其中完全是关于在关系模型中对分层数据建模的概念。如果在此基础之上再增加一个基本前提：关系数据库认为数据的结构（数据库架构）不灵活，则尝试支持数据的临时“添加”功能将变得十分困难。（快速回答下面的问题：你们之中有多少人处理过包含一个Notes 列（乃至 Note1、Note2、Note3……）的数据库？）

NoSQL 运动中没有任何人会说关系模型没有优点，也没有人会说关系数据库将会消失，但过去二十年开发人员生涯的一个最基本的事实是，开发人员经常将数据存储到本质上并非关系模型（有时甚至与这种模型相去甚远）的关系数据库中。

面向文档的数据库便是用于存储“文档”（这是一些紧密结合的数据集合，通常并未关联到系统中的其他数据元素），而非“关系”。例如，博客系统中的博客条目彼此毫无关联，即使出现某一篇博客确实引用到另一篇博客的情况，最常用的关联方法也是通过超链接（旨在由用户浏览器解除引用），而非内部关联。对本博客条目的评论完全局限于本博客条目的内部范围，不管评论的是什么博客条目，极少有用户想查看包含所有评论的内容集合。

此外，面向文档的数据库往往在高性能或高并发性环境中表现突出：MongoDB专门迎合高性能需求，而它的近亲 CouchDB 则更多的是针对高并发性的情况。两者都放弃了对多对象事务的支持，也就是说，尽管它们支持在数据库中对单个对象进行的并发修改，但若尝试一次性对多个对象进行修改，将会在一小段时间内看到这些修改正依序进行。文档以“原子方式”更新，但不存在涉及多文档更新的事务概念。这并不意味着MongoDB 没有任何稳定性，只是说MongoDB 实例与 SQL Server 实例一样不能经受电源故障。需要原子性、一致性、隔离性和持久性 (ACID) 完整要素的系统更适合采用传统的关系数据库系统，因此关键任务数据很可能不会太快出现在MongoDB 实例内，但Web 服务器上的复制数据或缓存数据可能要除外。

一般来说，若应用程序及组件需要存储可快速访问且常用的数据，则采用MongoDB 可以取得较好效果。网站分析、用户首选项和设置（以及包含非完全结构化数据或需采用结构灵活的数据的任何系统类型）都是采用 MongoDB 的自然之选。这并不意味着MongoDB 不能作为操作型数据的主要数据存储库；只是说MongoDB 能在传统 RDBMS 所不擅长的领域内如鱼得水，另外它也能在大量其他适合的领域内大展拳脚。

入门

前面提到过，MongoDB是一款开源软件包，可通过 MongoDB 网站 mongodb.com 轻松下载。在浏览器中打开该网站应该就能找到Windows 可下载二进制包的链接，请在页面右侧查找“Downloads”链接。另外，如果更愿意使用直接链接，请访问 mongodb.org/display/DOCS/Downloads。截至本文撰写之时，其稳定版本为发行版1.2.4。它其实就是一个.zip 文件包，因此相对而言，其安装过程简单得可笑：只需在任何想要的位置解压zip 包的内容。

没开玩笑，就是这样。

该 .zip 文件解压后生成三个目录：bin、include 和 lib。唯一有意义的目录是 bin 目录，其中包含八个可执行文件。除此之外不再需要任何其他的二进制（或运行时）依赖文件，而事实上，现在只需关注其中的两个可执行文件。这两个文件分别是mongod.exe（即 MongoDB 数据库进程本身）和 mongo.exe（即命令行 Shell 客户端，其使用方法通常类似于传统的isql.exe SQL Server 命令行Shell 客户端，用于确保所有内容都已正确安装且能正常运行，并用于直接浏览数据、执行管理任务）。

验证所有内容是否正确安装的过程十分简单，只需在命令行客户端上启动mongod。默认情况下，MongoDB将数据存储在默认的文件系统路径c:\data\db，但该路径是可以配置的，方法是在命令行上通过--config 命令按名称传递一个文本文件。假设mongod 即将启动的位置存在一个名为db 的子目录，验证所有内容是否安装得当的过程很简单，如**图 1** 所示。

图 1 **启动 mongod.exe 以验证安装是否成功**

如果该目录不存在，MongoDB并不会创建它。注意在 Windows 7 界面中，当启动 MongoDB 时，会弹出常见的“该应用程序要打开端口”对话框。请确保能访问到该端口（默认情况下指27017），或者最多是难以连接到该端口。（在后面一篇文章中，我会讨论将MongoDB 投入生产环境，其中将详细论述到这一问题。）

服务器进入运行状态后，通过Shell 连接到该服务器的过程非常简单：mongo.exe应用程序启动一个命令行环境，在该环境中便可直接与服务器交互，如**图 2** 所示。

图 2 **mongo.exe启动一个命令行环境，用于直接与服务器交互**

默认情况下，Shell连接到“test”数据库。由于此处目的只是验证是否一切运行正常，因此使用test 数据库就够了。当然，在这里可以轻松地创建一些简单的示例数据以用于MongoDB，例如创建一个描述某人的快速对象。在 MongoDB 中查看数据的启动过程非常简单，如**图 3** 所示。

图 3 **创建示例数据**

本质上，MongoDB 使用 JavaScript Object Notation (JSON) 作为其数据表示法，这种表示法能表现 MongoDB 的灵活性，并可说明 MongoDB 与客户端的交互方式。在内部，MongoDB 以 BSON（JSON的二进制超集）存储数据，目的是简化存储和索引。JSON 保留了MongoDB 的首选输入/输出格式，并且通常是在 MongoDB 网站和 wiki 上使用的文档格式。如果不熟悉 JSON，最好是在“深陷”MongoDB 之前充一下电。（开个玩笑。）同时，查看 mongod 用来存储数据的目录，您会发现其中一对以“test”命名的文件。

言归正传，该编写一些代码了。退出Shell 简单得只需键入“exit”，而关闭服务器也只需在窗口中按 Ctrl+C 或直接关闭窗口：服务器捕获到关闭信号并正确关闭所有内容，然后退出进程。

MongoDB 的服务器（以及 Shell，尽管它微不足道）是用地道的 C++ 应用程序（还记得吗？）编写的，因此访问该服务器需要使用某种 .NET Framework 驱动程序，此类驱动程序知道如何通过打开的套接字进行连接以向服务器输送命令和数据。MongoDB 程序包中并未绑定 .NET Framework 驱动程序，但有幸的是，社区提供了一个，此处的“社区”指的是名叫 Sam Corder 的开发人员，他构建了一个 .NET Framework 驱动程序以及 LINQ 支持来访问 MongoDB。他的作品同时以源代码形式和二进制形式提供，位于github.com/samus/mongodb-csharp。可以从该页面下载二进制文件（查找页面右上角），也可以下载源代码，然后自行编译。无论采取哪种方式，都会产生两个程序集：MongoDB.Driver.dll 和 MongoDB.Linq.dll。通过向对应项目的“引用”节点快速添加引用后，就可以使用 .NET Framework 了。

编写代码

从根本上来说，打开与正在运行的MongoDB 服务器的连接，同打开与任何其他数据库的连接没有太大差别，如**图 4** 所示。

图 4 **打开与 MongoDB 服务器的连接**

```
using System;

using MongoDB.Driver;


namespace ConsoleApplication1

{
```

```
  class Program
  {
    static void Main(string[] args)
    {
      Mongo db = new Mongo();
      db.Connect(); //Connect to localhost on the default port
      db.Disconnect();
    }
  }
}
```

查找先前创建的对象并不难，只是与以前.NET Framework 开发人员使用过的方法有所不同而已（请参阅**图 5**）。

图 5 **查找创建的 mongo 对象**

```
using System;
using MongoDB.Driver;

namespace ConsoleApplication1
{
  class Program
  {
    static void Main(string[] args)
    {
      Mongo db = new Mongo();
      db.Connect(); //Connect to localhost on the default port.
      Database test = db.getDB("test");
      IMongoCollection things = test.GetCollection("things");
      Document queryDoc = new Document();
      queryDoc.Append("lastname", "Neward");
      Document resultDoc = things.FindOne(queryDoc);
      Console.WriteLine(resultDoc);
      db.Disconnect();
    }
  }
}
```

如果上述内容看起来太突然，别担心，写出这样的代码并非“一日之功”，因为 MongoDB 存储数据的方式与传统数据库是不同的。

对于初学者，请回忆一下，先前插入的数据有三个字段：firstname、lastname 和 age，这三个元素都可作为数据的检索条件。但更重要的是，存储这些数据的行（以强制方式快速完成该过程）为“test.things.save()”，这表示数据被存储在称为“things”的事物中。在 MongoDB 术语中，“things”是一个集合，不言而喻，所有数据都存储在集合中。集合中依次存储着文档，文档则存储着“键/值”对，而其中的“值”又可以是其他集合。在本例中，“things”就是存储在前面提到的 test 数据库内部的集合。

因此，获取数据的过程首先要连接到MongoDB 服务器，再连接到test 数据库，然后查找集合“things”。这就是**图 5** 中前四行的操作：创建一个表示连接的Mongo 对象，连接到服务器，连接到test 数据库，然后获取“things”集合。

返回集合之后，代码可通过调用FindOne 的方式发出一条查询命令来查找单个文档。但与所有数据库一样，该客户端并不想获取集合中的每一个文档，只想查找感兴趣的文档，因此需要对查询进行某种方式的约束。在MongoDB 中，该约束的实现方式是创建一个Document，其中包含字段以及要在这些字段中搜索的数据，这是一种称为示例查询（简称QBE）的概念。由于此处的目标是查找包含lastname 字段（其值设为“Neward”）的文档，因此需要创建一个仅包含一个lastname 字段（及其值）的Document，并作为参数传递给FindOne。如果查询成功，则返回另一个Document，其中包含所有相关数据（外加另一个字段）；否则返回null。

顺便提一句，此描述的缩略版可简化为：

```
Document anotherResult =
         db["test"]["things"].FindOne(
           new Document().Append("lastname", "Neward"));
       Console.WriteLine(anotherResult);
```

运行时，不仅会显示传入的原始值，还会显示一个新值，即一个包含ObjectId 对象的 _id 字段。这是对象的唯一标识符，是在存储新数据时由数据库自动插入的。在尝试修改此对象时，必须避免修改该字段，否则数据库会将该对象视为传入的新对象。通常，这是通过修改由查询返回的Document 来完成的：

```
anotherResult["age"] = 39;
       things.Update(resultDoc);
       Console.WriteLine(
         db["test"]["things"].FindOne(
           new Document().Append("lastname", "Neward")));
```

但是，您始终可以创建新的Document 实例并手动填入_id 字段来匹配ObjectId（如果这样做更合理）：

```
Document ted = new Document();
       ted["_id"] = new MongoDB.Driver.Oid("4b61494aff75000000002e77");
       ted["firstname"] = "Ted";
       ted["lastname"] = "Neward";
       ted["age"] = 40;
       things.Update(ted);
       Console.WriteLine(
         db["test"]["things"].FindOne(
           new Document().Append("lastname", "Neward")));
```

当然，如果_id 已知，那么也可将其用作查询条件。

请注意，由于Document 被有效地非类型化（无类型），因此几乎所有内容均能以任意名称存储在字段中，包括某些核心的.NET Framework 值类型，如DateTime。如前所述，从技术角度上讲，MongoDB用于存储 BSON 数据，其中包括传统 JSON 类型（字符串、整数、布尔值、双精度和null，不过 null 仅允许用于对象，不允许用于集合）的某些扩展，例如上文提到的ObjectId、二进制数据、正则表达式以及嵌入式JavaScript 代码。我们暂时先不管后面两种类型，BSON能存储二进制数据的这种说法是指能存储任何可简化为字节数组的内容，这实际上表示MongoDB 能存储任何内容，但可能无法在该二进制BLOB 中进行查询。

下载代码示例

关于作者：

Ted Neward 是 Neward & Associates 的负责人，这是一家专门研究 .NET Framework 企业系统和 Java 平台系统的独立公司。他曾写作 100 多篇文章，是 C# 领域最优秀的专家之一并且是 INETA 发言人，著作或合著过十几本书，包括即将出版的《Professional F# 2.0》(Wrox)。他定期担任顾问和导师，请通过 ted@tedneward.com 与他联系，或通过 blogs.tedneward.com 访问其博客。

通过 MongoDB 推动 NoSQL（第2部分）

在上一篇文章中，主要介绍了 MongoDB 的基本知识：安装、运行，以及插入和查找数据。不过，这篇文章只介绍了基本知识，所用的数据对象是简单的名称/值对。这是有道理的，因为 MongoDB 的最大优势就包括可使用相对简单的非结构化数据结构。可以肯定地说，这种数据库能存储的不只是简单的名称/值对。

在本文中，我们将通过一种略微不同的方法来研究MongoDB（或任何技术）。这个称为探索测试的过程可帮助我们发现服务器中可能存在的错误，同时可以凸显面向对象开发人员在使用MongoDB 时会遇到的常见问题之一。

前文回顾...

首先，我们要确保讨论同样的问题，还要涉及一些略微不同的新领域。让我们以一种与前一文章(http://cloud.csdn.net/a/20110321/294274.html)相比更加结构化的方式来探讨MongoDB。我们不只是创建简单的应用程序，然后进行调试，我们将采取一举两得的做法，创建探索测试。探索测试的代码段看起来像单元测试，但它们探索功能而不是尝试验证功能。

在研究一项新技术时，编写探索测试可实现几种不同的目的。其一，它们有助于发现所研究的技术在本质上是不是可以测试的（假设如下：如果难于进行探索测试，则难于进行单元测试，而这是一个很严重的问题）。其二，在所研究的技术出现新的版本时，它们可作为一种回归测试，因为它们可在旧功能不再正常工作的情况下发出警告。其三，测试应是相对小型精细的，因此，在本质上，探索测试通过基于以前用例创建新“what-if”用例，使得新技术的学习更为容易。

不过，与单元测试不同，探索测试不是随应用程序连续开发的，因此，一旦考虑所学习的技术，请将这些测试放在一旁。但不要将它们丢弃，它们还可帮助分离应用程序代码中的错误与库或框架中的错误。这些测试通过提供一种与应用程序无关的轻型环境来进行实验，从而完成这种分离，不会产生应用程序开销。

明确了这一点后，我们来创建Visual C# 测试项目MongoDB-Explore。将MongoDB.Driver.dll 添加到程序集引用列表中，然后进行生成，以确保一切正常。（生成时应选择作为项目模板的一部分而生成的TestMethod。默认情况下，该测试将会通过，因此一切正常，这意味着，如果项目无法生成，则环境中存在问题。检查假设是否正确总是很好的方法。）

看起来可以立即着手编写代码了，不过，马上会出现一个问题：MongoDB需要运行外部服务器进程(mongod.exe)，这样客户端代码才能对该进程进行连接，执行有用的操作。我们很容易说“好，好，让我们启动它，然后开始编写代码”，这样还是存在一个必然的问题。几乎可以肯定，15个星期后的某个时候，回头再看这些代码，某些糟糕的开发人员（您、我或团队同事）会尝试运行这些测试，看着它们全部失败，然后浪费两三天努力寻找原因，这才想起看一看服务器是否已运行。

经验教训：尝试以某种方式在测试中捕获所有依赖关系。不管怎样，问题会再次出现在单元测试过程中。因此，我们需要从全新状态的服务器开始，进行一些更改，然后撤消全部更改。要完成这项工作，最简单的方法是停止并启动服务器，现在将问题解决，就为以后节约了时间。

在测试之前（和/或之后）进行运行操作不是什么新方法，Microsoft测试和实验室管理器项目可以使用按测试和按测试套件的初始值设定项和清理方法。这些方法包含适用于按测试套件记帐的自定义属性ClassInitialize 和 ClassCleanup 和适用于按测试记帐的TestInitialize 和 TestCleanup。（有关详细信息，请参见“使用单元测试”。）因此，按测试套件的初始值设定项将启动mongod.exe 进程，而按测试套件的清理方法会关闭该进程，如 **图 1** 所示。

图 1 **测试初始值设定项和清理方法的部分代码**

```
namespace MongoDB_Explore
{
  [TestClass]
  public class UnitTest1
  {
    private static Process serverProcess;

   [ClassInitialize]
   public static void MyClassInitialize(TestContext testContext)
   {
     DirectoryInfo projectRoot =
       new DirectoryInfo(testContext.TestDir).Parent.Parent;
     var mongodbbindir =
       projectRoot.Parent.GetDirectories("mongodb-bin")[0];
     var mongod =
       mongodbbindir.GetFiles("mongod.exe")[0];

     var psi = new ProcessStartInfo
     {
       FileName = mongod.FullName,
       Arguments = "--config mongo.config",
       WorkingDirectory = mongodbbindir.FullName
     };

     serverProcess = Process.Start(psi);
   }
   [ClassCleanup]
   public static void MyClassCleanup()
   {
     serverProcess.CloseMainWindow();
     serverProcess.WaitForExit(5 * 1000);
```

```
    if (!serverProcess.HasExited)
      serverProcess.Kill();
  }
...
```

上述代码第一次运行时，将弹出一个对话框，通知用户正在启动进程。单击“确定”，该对话框就会消失 ... 直到下一次运行该测试。如果不希望显示该对话框，请找到并选中单选框“不再显示此对话框”，以便不再显示该消息。如果正在运行防火墙软件（如Windows 防火墙），也可能出现该对话框，这是因为服务器需要打开一个端口来接收客户端连接。采用同样的方法处理，所有操作都应以无提示方式运行。如果需要，可在清理代码的第一行放置一个断点，验证服务器是否正在运行。

只要服务器正在运行，就可开始测试，除非出现另一个问题：每个测试都需要使用自己的全新数据库，但数据库中预先存在一些数据也是很有用的，这样，更便于进行某些方面（如查询）的测试。每个测试最好都有自己的预先存在的全新数据。包含TestInitializer 和 TestCleanup的方法可以完成这一任务。

对此加以讨论之前，我们来看一看这个快速TestMethod，它尝试确保找到服务器，进行连接，插入、找到和删除对象，使探索测试的速度提高到前一文章所介绍的那样（请参见**图 2**）。

图 2 **TestMethod确保找到服务器并进行连接**

```
[TestMethod]
public void ConnectInsertAndRemove()
{
  Mongo db = new Mongo();
  db.Connect();

  Document ted = new Document();
  ted["firstname"] = "Ted";
  ted["lastname"] = "Neward";
  ted["age"] = 39;
  ted["birthday"] = new DateTime(1971, 2, 7);
  db["exploretests"]["readwrites"].Insert(ted);
  Assert.IsNotNull(ted["_id"]);

  Document result =
    db["exploretests"]["readwrites"].FindOne(
    new Document().Append("lastname", "Neward"));
  Assert.AreEqual(ted["firstname"], result["firstname"]);
  Assert.AreEqual(ted["lastname"], result["lastname"]);
  Assert.AreEqual(ted["age"], result["age"]);
  Assert.AreEqual(ted["birthday"], result["birthday"]);

  db.Disconnect();
```

```
}
```

如果运行上述代码，运行到声明时，测试将失败。具体来说，问题出在最后一条关于“birthday”的声明。很显然，若将 DateTime 发送到没有时间的 MongoDB 数据库中，是不会正确往返的。进入的数据类型是关联时间为午夜的日期，返回的是关联时间为早上8 点的日期，这不符合测试末尾处的AreEqual 声明。

这一点凸显出探索测试的用处，要是不使用探索测试（举例来说，前一文章中的代码就是这样），可能要到项目进行几个星期或几个月后才会注意到MongoDB 的这一小特性。这是不是MongoDB 服务器中的错误是一种价值判断，不需要马上探讨。重要的是，探索测试对技术进行放大观察，有助于隔离这种“有趣的”行为。因此，希望使用该技术的开发人员可以确定这是不是一个重要更改。有备无患。

顺便提一下，若要修复这段代码从而通过测试，需要将从数据库返回的DateTime 转换为本地时间。我曾在一个在线论坛中提出这个问题，MongoDB.Driver的作者 Sam Corder 的回答是：“所有进入的日期都会转换为 UTC，并返回 UTC 时间。”因此，必须将 DateTime 转换为 UTC 时间才能通过DateTime.ToUniversalTime 进行存储，或者通过DateTime.ToLocalTime 将从数据库检索的所有DateTime 转换为本地时区，示例代码如下：

```
Assert.AreEqual(ted["birthday"],
  ((DateTime)result["birthday"]).ToLocalTime());
```

这件事本身凸显了社区的一个极大的优点，即通信双方的距离就是一封电子邮件。

增加复杂性

希望使用 MongoDB 的开发人员需要知道，与最初给人的印象相反，它不是一个对象数据库，也就是说，如果得不到帮助，它无法任意处理复杂对象图。一些常规做法可以提供这种帮助，不过迄今为止，还是需要开发人员才能实现。

例如，考虑**图 3** 所示的简单对象集合，该集合用于反映很多文档的存储情况，而这些文档描述的是一个有名的家庭。至此不会有什么问题。实际上，执行测试时，测试应向数据库查询插入的对象（如**图 4** 所示），这是为了确保这些对象是可以检索的。这样，测试通过。真是太妙了。

图 3 **简单对象集合**

```
[TestMethod]
public void StoreAndCountFamily()
{
  Mongo db = new Mongo();
  db.Connect();

  var peter = new Document();
  peter["firstname"] = "Peter";
  peter["lastname"] = "Griffin";

  var lois = new Document();
  lois["firstname"] = "Lois";
  lois["lastname"] = "Griffin";

  var cast = new[] {peter, lois};
  db["exploretests"]["familyguy"].Insert(cast);
```

```
  Assert.IsNotNull(peter["_id"]);
  Assert.IsNotNull(lois["_id"]);

  db.Disconnect();
}
```

图 4 **向数据库查询对象**

```
[TestMethod]
public void StoreAndCountFamily()
{
  Mongo db = new Mongo();
  db.Connect();

  var peter = new Document();
  peter["firstname"] = "Peter";
  peter["lastname"] = "Griffin";

  var lois = new Document();
  lois["firstname"] = "Lois";
  lois["lastname"] = "Griffin";

  var cast = new[] {peter, lois};
  db["exploretests"]["familyguy"].Insert(cast);
  Assert.IsNotNull(peter["_id"]);
  Assert.IsNotNull(lois["_id"]);

  ICursor griffins =
    db["exploretests"]["familyguy"].Find(
      new Document().Append("lastname", "Griffin"));
  int count = 0;
  foreach (var d in griffins.Documents) count++;
  Assert.AreEqual(2, count);

  db.Disconnect();
}
```

实际上，这种情况可能不完全是真实的。细致的读者如果键入代码就可能发现，说到底，测试并没有通过，因为预期的对象数与2 不匹配。这是因为，正如通常的数据库一样，这个数据库的状态在多次调用中保持不变，此外，由于测试代码不显式删除对象，这些对象在各个测试中都存在。

这凸显了面向文档数据库的另外一个特点：完全可能存在重复项，也允许存在重复项。正因为这，每个文档一经插入，都会由implicit_id 属性进行标记，并且有一个唯一的标识符存储在该属性中，这个唯一标识符实际上会成为文档的主键。

因此，如果要通过测试，需要在运行每个测试之前清除数据库。尽管删除MongoDB 存储文件的目录中的文件十分容易，但最好使测试套件能够自动执行这一任务。每个测试都可在完成后以手动方式完成这一任务，时间一长，这会变得有些乏味。测试代码可利用Microsoft 测试和实验室管理器的TestInitialize 和 TestCleanup功能来捕获常用代码（何不包括数据库连接和断开逻辑），如**图 5** 所示。

图 5 **利用 TestInitialize 和TestCleanup**

```
private Mongo db;

[TestInitialize]
public void DatabaseConnect()
{
  db = new Mongo();
  db.Connect();
}

[TestCleanup]
public void CleanDatabase()
{
  db["exploretests"].MetaData.DropDatabase();

  db.Disconnect();
  db = null;
}
```

CleanDatabase 方法的最后一行不是必不可少的，因为下一个测试会用新的 Mongo 对象覆盖该字段引用，不过，有时最好明确表示出该引用不再有内容。用者自慎。重要的是删除在测试中使用过的数据库，清空MongoDB用于存储数据的文件，一切都以全新的状态迎接下一个测试。

不过，就目前情况看，该家庭模型是不完整的。所引用的两个人是一对伴侣，假设他们应将对方引用为配偶，如下所示：

```
peter["spouse"] = lois;
  lois["spouse"] = peter;
```

如果在测试中运行这段代码，会产生StackOverflowException。MongoDB驱动程序序列化程序本身不理解循环引用的概念，它会无休止地引用下去。天哪。这可不是什么好事。

若要修复这一问题，可以在两种方法中选择其一。一种方法是，配偶字段可使用其他文档的_id 字段来填充（该文档插入后）和更新，如**图 6** 所示。

图 6 **解决循环引用问题**

```
[TestMethod]
public void StoreAndCountFamily()
{
  var peter = new Document();
  peter["firstname"] = "Peter";
  peter["lastname"] = "Griffin";

  var lois = new Document();
  lois["firstname"] = "Lois";
  lois["lastname"] = "Griffin";

  var cast = new[] {peter, lois};
  var fg = db["exploretests"]["familyguy"];
  fg.Insert(cast);
  Assert.IsNotNull(peter["_id"]);
  Assert.IsNotNull(lois["_id"]);

  peter["spouse"] = lois["_id"];
  fg.Update(peter);
  lois["spouse"] = peter["_id"];
  fg.Update(lois);

  Assert.AreEqual(peter["spouse"], lois["_id"]);
  TestContext.WriteLine("peter: {0}", peter.ToString());
  TestContext.WriteLine("lois: {0}", lois.ToString());
  Assert.AreEqual(
    fg.FindOne(new Document().Append("_id",
    peter["spouse"])).ToString(),
    lois.ToString());

  ICursor griffins =
    fg.Find(new Document().Append("lastname", "Griffin"));
  int count = 0;
  foreach (var d in griffins.Documents) count++;
  Assert.AreEqual(2, count);
```

```
}
```

不过，这种方法有一个缺点：它要求将文档插入数据库，并根据需要将它们的_id 值（在MongoDB.Driver 中是Oid 实例）复制到每个对象的配偶字段中。这时，每个文档会再次更新。尽管访问MongoDB 数据库与传统RDBMS 更新相比速度是很快的，这种方法仍有些费时。

第二种方法是为每个文档预先生成Oid 值，填充配偶字段，然后将整个批次发送到数据库，如**图 7** 所示。

图 7 **一种更好的解决循环引用问题的方法**

```
[TestMethod]
public void StoreAndCountFamilyWithOid()
{
  var peter = new Document();
  peter["firstname"] = "Peter";
  peter["lastname"] = "Griffin";
  peter["_id"] = Oid.NewOid();

  var lois = new Document();
  lois["firstname"] = "Lois";
  lois["lastname"] = "Griffin";
  lois["_id"] = Oid.NewOid();

  peter["spouse"] = lois["_id"];
  lois["spouse"] = peter["_id"];

  var cast = new[] { peter, lois };
  var fg = db["exploretests"]["familyguy"];
  fg.Insert(cast);

  Assert.AreEqual(peter["spouse"], lois["_id"]);
  Assert.AreEqual(
    fg.FindOne(new Document().Append("_id",
    peter["spouse"])).ToString(),
    lois.ToString());

  Assert.AreEqual(2,
    fg.Count(new Document().Append("lastname", "Griffin")));
}
```

这种方法仅需要Insert 方法，因为Oid 值是提前已知的。顺便提请注意，对声明测试的ToString 调用是特意进行的，这样，文档会在进行比较之前转换为字符串。

在**图 7** 的代码中，真正务必要注意的是，对通过Oid 引用的文档解除引用可能比较困难和乏味，因为面向文档这种形式假设文档或多或少是独立实体或分层实体，而不是对象图。（请注意，.NET驱动程序提供了 DBRef，后者可通过略微更丰富的方式来引用/解除引用其他文档，但仍无法实现对象图友好的系统。）因此，尽管肯定可以获得一个丰富的对象模型并将其存储到MongoDB 数据库中，仍不建议这样做。请坚持使用Word 或 Excel 这样的文档来存储紧密群集的数据组。如果某些内容可视为大型文档或电子表格，则可能非常适合MongoDB 或其他某种面向文档的数据库。

通过 MongoDB 推动 NoSQL（第3部分）

上一次，我使用探索测试继续对MongoDB 进行探讨。我介绍了如何在测试期间启动和停止服务器，然后介绍了如何获取跨文档引用，并探讨了导致如此麻烦举动的原因。现在我们需要探索更多中间的 MongoDB 功能：谓词查询、聚合函数以及 MongoDB.Linq 程序集提供的 LINQ 支持。我还将提供一些有关在生产环境中安置 MongoDB 的注意事项。

当我们最终离开我们的主角 .. .

在相关的代码包中，我充实了探索测试的内容，使用我最喜欢的电视节目里的人物，加入了一组已存在的示例数据集以供处理。**图 1** 显示了之前的探索测试，已经过刷新器处理。到目前为止一切顺利。

图 1 **示例探索测试**

```
[TestMethod]
        public void StoreAndCountFamilyWithOid()
        {
          var oidGen = new OidGenerator();
          var peter = new Document();
          peter["firstname"] = "Peter";
          peter["lastname"] = "Griffin";
          peter["_id"] = oidGen.Generate();

          var lois = new Document();
          lois["firstname"] = "Lois";
          lois["lastname"] = "Griffin";
          lois["_id"] = oidGen.Generate();

          peter["spouse"] = lois["_id"];
          lois["spouse"] = peter["_id"];

          var cast = new[] { peter, lois };
          var fg = db["exploretests"]["familyguy"];
          fg.Insert(cast);

          Assert.AreEqual(peter["spouse"], lois["_id"]);
```

```
          Assert.AreEqual(
            fg.FindOne(new Document().Append("_id",
              peter["spouse"])).ToString(), lois.ToString());

          Assert.AreEqual(2,
            fg.Count(new Document().Append("lastname", "Griffin")));
        }
```

呼叫所有老人 . . .

在前面的文章中，客户端代码已经获得所有匹配特定标准的文档（例如“lastname”字段匹配特定的 String 或“_id”字段匹配特定的 Oid），但我还没有介绍如何进行谓词查询（例如“找到所有‘age’字段的值大于 18 的文档”）。事实上，MongoDB 并不使用 SQL 风格的接口来描述要执行的查询，而是使用ECMAScript/JavaScript，而且它能接受要在服务器上执行的代码块以筛选或聚合数据，几乎就像存储过程一样。

这提供了一些类似LINQ 的功能，甚至不用了解Mongo.Linq 程序集支持的LINQ 功能我们就能看出来。通过指定包含名为“$where”的字段的文档和描述要执行的ECMAScript 代码的代码片段，可以随意创建复杂的查询：

```
[TestMethod]
        public void Where()
        {
          ICursor oldFolks =
            db["exploretests"]["familyguy"].Find(
            new Document().Append("$where",
            new Code("this.gender === 'F'")));
          bool found = false;
          foreach (var d in oldFolks.Documents)
            found = true;
          Assert.IsTrue(found, "Found people");
        }
```

正如您所看到的，Find调用返回 ICursor 实例，尽管该实例本身不是 IEnumerable（表示它无法用在 ForEach 循环中），却包含一个类型为IEnumerable<Document> 的Documents 属性。如果查询会返回很大的数据集，通过将其Limit 属性设置为 *n*，可以要求 ICursor 返回前 *n* 个结果。

谓词查询的语法共有四种格式，如**图 2** 所示。

图 2 **四种不同的谓词查询语法**

```
[TestMethod]
        public void PredicateQuery()
        {
          ICursor oldFolks =
            db["exploretests"]["familyguy"].Find(
            new Document().Append("age",
```

```
            new Document().Append("$gt", 18)));
          Assert.AreEqual(6, CountDocuments(oldFolks));

          oldFolks =
            db["exploretests"]["familyguy"].Find(
            new Document().Append("$where",
            new Code("this.age > 18")));
          Assert.AreEqual(6, CountDocuments(oldFolks));

          oldFolks =
            db["exploretests"]["familyguy"].Find("this.age > 18");
          Assert.AreEqual(6, CountDocuments(oldFolks));

          oldFolks =
            db["exploretests"]["familyguy"].Find(
            new Document().Append("$where",
            new Code("function(x) { return this.age > 18; }")));
          Assert.AreEqual(6, CountDocuments(oldFolks));
        }
```

在第二种和第三种格式中，“this”指的是要查询的对象。

事实上，您可以使用文档传达查询或命令，以便从驱动程序向数据库发送任意命令（即ECMAScript 代码）。例如，IMongoCollection接口提供的 Count 方法就是对这段冗长代码的简便替代方式：

```
[TestMethod]
        public void CountGriffins()
        {
          var resultDoc = db["exploretests"].SendCommand(
            new Document()
              .Append("count", "familyguy")
              .Append("query",
                new Document().Append("lastname", "Griffin"))
            );
          Assert.AreEqual(6, (double)resultDoc["n"]);
        }
```

这意味着 MongoDB 文档介绍的所有聚合操作（例如“distinct”或“group”）都可以通过同一种机制进行访问，虽然 MongoDB.Driver API 未将它们作为方法提供。

您可以通过“特殊名称”语法“$eval”将查询之外的任意命令发送到数据库，这样就可以对服务器执行任何合法的 ECMAScript 代码块，仍旧很像存储过程：

```
[TestMethod]
        public void UseDatabaseAsCalculator()
        {
          var resultDoc = db["exploretests"].SendCommand(
            new Document()
              .Append("$eval",
                new CodeWScope {
                  Value = "function() { return 3 + 3; }",
                  Scope = new Document() }));
          TestContext.WriteLine("eval returned {0}", resultDoc.ToString());
          Assert.AreEqual(6, (double)resultDoc["retval"]);
        }
```

或者直接对数据库使用所提供的Eval 函数。如果这还不够灵活，MongoDB允许在特殊的数据库集合“system.js”中添加 ECMAScript 函数，从而在数据库实例上存储用户定义的要在查询时执行的ECMAScript 函数以及服务器端执行块，如 MongoDB 网站所述。

缺少的 LINQ

C# MongoDB 驱动程序也有LINQ 支持，允许开发人员编写如**图 3** 所示的 MongoDB 客户端代码。

图 3 **LINQ支持示例**

```
[TestMethod]
        public void LINQQuery()
        {
          var fg = db["exploretests"]["familyguy"];
          var results =
            from d in fg.Linq()
            where ((string)d["lastname"]) == "Brown"
            select d;
          bool found = false;
          foreach (var d in results)
          {
            found = true;
            TestContext.WriteLine("Found {0}", d);
          }
          Assert.IsTrue(found, "No Browns found?");
        }
```

而且为了保持MongoDB 数据库的动态特征，此示例不需要生成代码，只需调用Linq 以返回可以“启用” MongoDBLINQ 提供程序的对象即可。在我撰写本文时，LINQ支持还相当粗略，但在本文发表时，它将得到极大改进。新功能的文档和相关示例将在项目网站的 wiki 栏目发布。

发布是一项功能

最重要的是，如果要将MongoDB 用在生产环境中，对于那些要让生产服务器和服务保持运行的工作人员来说，还有几个问题需要解决，这样才能减轻他们的工作负担。

首先，需要将服务器进程(mongod.exe) 安装为服务，因为在生产服务器上一般不允许在交互式桌面会话中运行该进程。因此，mongod.exe支持一个服务安装选项“--install”，通过该选项可将其安装为服务，然后通过服务面板或命令行“netstart MongoDB”启动。但是，截止本文撰写时，--install命令存在一个小问题：它通过查看执行安装所用的命令行来推断可执行文件的路径，因此必须在命令行中指定完整路径。也就是说，如果MongoDB 安装在C:\Prg\mongodb 中，您必须在命令提示符处（使用管理员权限）使用命令C:\Prg\mongodb\bin\mongod.exe --install 将其安装为服务。

但是，所有的命令行参数，例如“--dbpath”，必须也显示在该安装命令中，这意味着如果端口、数据文件的路径等等设置发生更改，则必须重新安装服务。幸运的是，MongoDB支持一个配置文件选项（通过“--config”命令行选项指定），因此通常最好的做法是将配置文件的完整路径传递到服务安装命令，然后在文件中进行其他所有配置：

```
C:\Prg\mongodb\bin\mongod.exe --config C:\Prg\mongodb\bin\mongo.cfg --install

net start MongoDB
```

像往常一样，测试服务是否成功运行的最简便方法就是使用MongoDB 下载随附的mongo.exe 客户端来连接服务。由于服务器通过套接字与客户端通信，因此您需要在防火墙中设置通道，以便允许与服务器的通信。

没有您需要的数据机器人

当然，对 MongoDB 服务器的不安全访问不可能是什么好事，因此阻止不需要的访问者访问服务器成为一项重要功能。MongoDB支持身份验证，但是它的安全系统不像SQL Server 这样的“大块头”数据库那么精密。

一般来说，第一步是使用mongo.exe 客户端连接数据库并将管理员用户添加到管理数据库中（该数据库中包含用于运行和管理整个MongoDB 服务器的数据），从而创建数据库管理登录，如下所示：

```
> use admin

> db.addUser("dba", "dbapassword")
```

完成之后，所有进一步的操作（甚至是该外壳程序中的操作）都需要经过身份验证，通过在该外壳程序中进行显式身份验证完成：

```
> db.authenticate("dba", "dbapassword")
```

DBA 现在可以更改数据库并使用前面所示的同一个addUser 调用来添加用户，从而将用户添加到MongoDB 数据库中：

```
> use mydatabase

> db.addUser("billg", "password")
```

通过Mongo.Driver 连接数据库时，身份验证信息将作为用于创建Mongo 对象的连接字符串的一部分进行传递，相同的身份验证过程将透明地进行：

```
var mongo = new Mongo("Username=billg;Password=password");
```

很自然，密码不应该直接硬编码在代码中或公开存储，而应该使用与所有基于数据库的应用程序相同的密码规则。实际上，整个配置（主机、端口、密码等）应该保存在配置文件中并通过ConfigurationManager 类进行检索。

扩展到另一些代码

管理员应该定期查看正在运行的实例，以获得正在运行的服务器的相关诊断信息。MongoDB支持一个 HTTP 接口用于与数据库交互，该接口运行的端口号比用于普通客户端通信的端口号高1,000。MongoDB 的默认端口是27017，因此该 HTTP 接口位于端口 28017，如**图 4** 所示。

图 4 **用于与 MongoDB 交互的HTTP 接口**

该 HTTP 接口还允许更加偏向 REST 风格的通信方法，与MongoDB.Driver 和MongoDB.Linq 中的本机驱动程序正相反；MongoDB网站对此有详细介绍，但用于访问集合内容的HTTP URL 中需要添加数据库名称和集合名称（用斜线分隔），如**图 5** 所示。

图 5 **用于访问集合内容的 HTTP URL**

有关使用 WCF 创建 REST 客户端的详细信息，请参见 MSDN 文章“REST in WindowsCommunication Foundation (WCF)”。

专家的忠告

MongoDB 是一款发展迅速的产品，这些旨在探索MongoDB 核心功能的文章有很多内容并未涉猎。尽管MongoDB 不能直接替代SQL Server，但在传统的关系数据库管理系统作用有限的领域，它确实是一种可行的存储替代方案。与MongoDB 相同，mongodb-csharp项目也处在蓬勃发展之中。撰写本文时，很多新的改进功能已经添加到Beta 版中，包括使用普通对象处理强类型化的集合以及对LINQ 支持的重大改进。请密切注意这两个项目。

但现在我们也应该和MongoDB 说再见了。让我们把注意力转向孜孜不倦的程序员可能不熟悉（也应该存在争议）的其他开发人员领域。但在目前，愉快编程，并记住伟大的开发专家曾说过的“开发人员使用源代码获取知识，进行防御，切勿成为黑客”。

下载示例代码

关于作者：

Ted Neward 是 Neward & Associates 的负责人，这是一家专门研究 .NET Framework 企业系统和 Java 平台系统的独立公司。他曾写作 100 多篇文章，是 C# 领域最优秀的专家之一并且是 INETA 发言人，著作或合著过十几本书，包括即将出版的《Professional F# 2.0》(Wrox)。他定期担任顾问和导师，请通过 ted@tedneward.com 与他联系，或通过 blogs.tedneward.com 访问其博客。

MongoDB之父：MongoDB胜过BigTable

Dwight Merriman和他的团队，包括ShopWiki的创始人Eliot Horowitz参加了在纽约10gen启动MongoDB的仪式。现在该公司除了担任该开源项目的主要运营者之外，还提供支持、培训和咨询服务。10gen在旧金山举办了第二届开发者大会，Merriman在上午的大会做了主题演讲，主要介绍了MongoDB的起源，并解释了为何要建立这样的数据库。

“在2007年底，当时的想法是构建一个用于开发、托管并具有自动缩放Web应用程序的在线服务”，谈到MongoDB诞生之目的时，Merriman介绍道。“但是不同于Google App Engine的是，这项服务完全建立在一个开放源代码的软件平台之上。”因此，在关注了Google Bigtable架构很长一段时间后，Merriman和他的团队注意到，尚没有一个开源的数据库平台适合这种服务，这兴许是个机会。

“我们意识到很多现有的数据库并不真正具备‘云计算’的特性。例如弹性、可扩展性以及易管理性。这些特性能够为开发者和运营者带来便利，而MySQL还不完全具备这些特点。

因此，Dwight Merriman以及他的团队的目标是构建一个全新的数据库。新的数据库将会放弃大家所熟悉的关系数据库模型，且是适合现代网络应用并基于分布式的平台。高度事务性的系统可以帮助解决一些棘手的问题，同时还支持云计算架构的伸缩性。Merriman解释到。经过一年的不断努力，这个数据库已经比较完善。他们将它设计为具有为“云计算服务”潜力的数据库。而且还会不断的完善，因为MongoDB本身就是一个开源数据库。

在开源的、面向文档的数据库中，MongoDB经常被誉为具有RDBMS功能的NoSQL数据库。MongoDB还带有交互式shell，这使得访问其数据存储变得简单，且其对于分块的即装即用的支持能够使高可伸缩性跨多个节点。

据悉，MongoDB的API是JSON对象和JavaScript函数的本地混合物。通过shell程序开发人员可与MongoDB进行交互，即允许命令行参数，或通过使用语言驱动程序来访问数据存储实例。这里不存在类JDBC驱动程序，这意味着开发人员不必处理ResultSet或PreparedStatement。

而速度是 MongoDB 的另外一个优势，主要是由于它处理写入的方式：它们存储在内存中，然后通过后台线程写入磁盘。

“由于用户不容易在大规模环境下作分布式的链接，并且在分布式环境下很难做快速的大规模部署，因此，用户需要一些辅助的东西”，Memmiman解释道。

最后他表示同样重要的是为了限制数据库的事务语义你可以使用分布式事务。但当你在1000台机器上运行时它不会那么快。例如银行或会计系统。传统的关系型数据库目前还是更适用于需要大量原子性复杂事务的应用程序。（李智/译）

MongoDB与CouchDB全方位对比

本文见于MongoDB官方网站，MongoDB与CouchDB很相似，他们都是文档型存储，数据存储格式都是JSON型的，都使用Javascript进行操作，都支持Map/Reduce。但是其实二者有着很多本质的区别，本文透过现象追寻本质，让你更好的理解MongoDB与CouchDB。

1.MVCC（Multiversionconcurrency control）

MongoDB与CouchDB的一大区别就是CouchDB是一个MVCC的系统，而MongoDB是一个update-in-place的系统。这二者的区别就是，MongoDB进行写操作时都是即时完成写操作，写操作成功则数据就写成功了，而CouchDB一个支持多版本控制的系统，此类系统通常支持多个结点写，而系统会检测到多个系统的写操作之间的冲突并以一定的算法规则予以解决。

2.水平扩展性

在扩展性方面，CouchDB使用replication去做，而MongoDB的replication仅仅用来增强数据的可靠性，MongoDB在实现水平扩展性方面使用的是Sharding。（据说CouchDB也有开发分片功能的计划）

3.数据查询操作

这个区别在用户接口上了，MongoDB与传统的数据库系统类似，支持动态查询，即使在没有建立索引的行上，也能进行任意的查询。而CouchDB不同，CouchDB不支持动态查询，你必须为你的每一个查询模式建立相应的view，并在此view的基础上进行查询。

4.原子性

这一点上两者比较一致，都支持针对行的原子性修改（concurrentmodifications of single documents），但不支持更多的复杂事务操作。

5.数据可靠性

CouchDB是一个”crash-only”的系统，你可以在任何时候停掉CouchDB并能保证数据的一致性。而MongoDB在不正常的停掉后需要运repairDatabase()命令来修复数据文件，在1.7.5版本后支持单机可靠的–dur命令。

6.Map/Reduce

MongoDB和CouchDB都支持Map/Reduce，不同的是MongoDB只有在数据统计操作中会用到，而CouchDB在变通查询时也是使用Map/Reduce。

7.使用 javascript

MongoDB和CouchDB都支持javascript，CouchDb用javascript来创建view。MongoDB使用JSON作为普通数据库操作的表达式。当然你也可以在操作中包含javascript语句。MongoDB还支持服务端的javascript脚本（running arbitrary javascript functions server-side），当然，MongoDB的Map/Reduce函数也是javascript格式的。

8.REST

CouchDB是一个RESTFul的数据库，其操作完全走HTTP协议，而MongoDB是走的自己的二进制协议。MongoDB Server在启动时可以开放一个HTTP的接口供状态监控。

9.性能

此处主要列举了MongoDB自己具有高性能的原因

采用二进制协议，而非CouchDB REST的HTTP协议

使用Momary Map内存映射的做法

collection-oriented，面向集合的存储，同一个collection的数据是连续存储的

update-in-place直接修改，而非使用MVCC的机制

使用C＋＋编写

10.适用场景

如果你在构建一个 Lotus Notes型的应用，我们推荐使用CouchDB，主要是由于它的MVCC机制。另外如果我们需要master-master的架构，需要基于地理位置的数据分布，或者在数据结点可能不在线的情况下，我们推荐使用CouchDB。

如果你需要高性能的存储服务，那我们推荐MongoDB，比如用于存储大型网站的用户个人信息，比如用于构建在其它存储层之上的Cache层。

如果你的需求中有大量update操作，那么使用MongoDB吧。就像我们在例子updating real time analytics counters中的一样，对于那种经常变化的数据，比如浏览量，访问数之类的数据存储。

Mongodb GridFS 介绍

MongoDB GridFS 是MongoDB用于文件存储的模块，本文简单介绍了期用法。

mongodb GridFS 性能

性能, 网评还不错.

不过在生产环境中,国外有用于存储视频流的.

GridFS的一个优点是可以存储上百万的文件而无需担心扩容性.

通过同步复制,可以解决分布式文件的备份问题.

通过ARP-ping可以实现一个双机热备切换,类mysql的mysql master master replic

使用Nginx module

http://github.com/mdirolf/nginx-gridfs

这是gridfs的nginx module. 可以通过nginx直接访问读取mongo gridfs中的文件.

和nginx对应的mogilefs module类似.

优点: 由于直接通过nginx,速度是最快的.

缺点: 只能通过file_path来查找,目前不支持_id来查找.因此必须在file_path上

建立索引.

其他一些信息:

1.通过runcommand可以直接在mongodb端运行处理脚本. 比如像mapreduce,或者一

些需要读取数据然后进行处理的.

这些command则是使用javascript方式来编写的,很容易. 好处就是避免了数据在服

务端和客户端之间的读取和传输,

提高效率.

2. sharding

sharding在目前开发版中已经具备,但还不成熟. 但是可以自己实现sharding比较

好.因为目前的sharding还是比较硬性的.

3.灵活使用magic操作符和upsert,比如$inc,$all,$in 等等

MongoDB:下一代MySQL?

MongoDB的特性

- 简单的查询语句，没有Join操作
- 文档型存储，其数据是用二进制的Json格式Bson存储的。其数据就像Ruby的hashes，或者Python的字典，或者PHP的数组
- Sharding，MongoDB提供auto-sharding实现数据的扩展性
- GridFS，MongoDB的提供的文件存储API
- 数组索引，你可以对文档中的某个数组属性建立索引
- MapReduce，可以用于进行复杂的统计和并行计算
- 高性能，通过使用mmap和定时fsync的方法，避免了频繁IO，使其性能更高

MongoDB的优点

- 高性能，速度非常快（如果你的内存足够的话）
- 没有固定的表结构，不用为了修改表结构而进行数据迁移
- 查询语言简单，容易上手
- 使用Sharding实现水平扩展
- 部署方便

使用MongoDB，你得记住以下几点：

- MongoDB 假设你有大磁盘空间
- MongoDB 假设你的内存也足够大于放下你的热数据
- MongoDB 假设你是部署在64位系统上的（32位有2G的限制，试用还可以）
- MongoDB 假设你的系统是little-endian的
- MongoDB 假设你有多台机器（并不专注于单机可靠性）
- MongoDB 假设你希望用安全换性能，同时允许你用性能换安全

MongoDB在下面领域不太擅长

- 不太稳定，特别是auto-sharding目前还有很多问题
- 不支持SQL，这意味着你很多通过SQL接口的工具不再适用
- 持久化，MongoDB单机可靠性不太好，宕机可能丢失一段时间的数据
- 相关文档比较少，新功能都有这个问题
- 相关人才比较难找，这也是新功能的问题之一

MongoDB安全性初探

本文是一篇转载文章，文章主要介绍了MongoDB 安全性方面的知识，包括 MongoDB 安全配制、认证机制及认证的网络流程，也简单介绍了可能利用JavaScript引擎执行脚本攻击的可能。

MongoDB，这么火的玩意其实早就想好好研究一下了。之前一直没空仔细学学新的东西，总是感觉精力不足。这次趁着买了一本书，就零零散散地在VPS上搭建、测试、看实现代码。感觉也蛮有意思的一个数据库。虽然感觉它非常简单，尤其是看代码的时候更是感觉如此。但这不也是另一个KISS的典范么，还是简单但是实用的东西最能流行。

既然都看了其实现，也不能不产出点什么。正好多年没更新博文，就简单分析一下 MongoDB 的安全性，凑个数先。

默认配置的安全情况

在默认情况下，mongod是监听在0.0.0.0之上的。而任何客户端都可以直接连接27017，且没有认证。好处是，开发人员或dba可以即时上手，不用担心被一堆配置弄的心烦意乱。坏处是，显而易见，如果你直接在公网服务器上如此搭建MongoDB，那么所有人都可以直接访问并修改你的数据库数据了。

默认情况下，mongod也是没有管理员账户的。因此除非你在admin数据库中使用db.addUser()命令添加了管理员帐号，且使用–auth参数启动mongod，否则在数据库中任何人都可以无需认证执行所有命令。包括delete和shutdown。

此外，mongod还会默认监听28017端口，同样是绑定所有ip。这是一个mongod自带的web监控界面。从中可以获取到数据库当前连接、log、状态、运行系统等信息。如果你开启了–rest参数，甚至可以直接通过web界面查询数据，执行mongod命令。

我试着花了一个晚上扫描了国内一个B段，国外一个B段。结果是国外开了78个MongoDB，而国内有60台。其中我随意挑选了10台尝试连接，而只有一台机器加了管理员账户做了认证，其他则全都是不设防的城市。可见其问题还是比较严重的。

其实MongoDB本身有非常详细的安全配置准则，显然他也是想到了，然而他是将安全的任务推给用户去解决，这本身的策略就是偏向易用性的，对于安全性，则得靠边站了。

用户信息保存及认证过程

类似MySQL将系统用户信息保存在mysql.user表。MongoDB也将系统用户的username、pwd保存在admin.system.users集合中。其中pwd =md5(username + ":mongo:" + real_password)。这本身并没有什么问题。username和:mongo:相当于对原密码加了一个salt值，即使攻击者获取了数据库中保存的md5 hash，也没法简单的从彩虹表中查出原始密码。

我们再来看看MongoDB对客户端的认证交互是如何实现的。mongo client和server交互都是基于明文的，因此很容易被网络嗅探等方式抓取。这里我们使用数据库自带的mongosniff，可以直接dump出客户端和服务端的所有交互数据包：

```
[root@localhost bin]# ./mongosniff --source NET lo

sniffing 27017


...//省略开头的数据包，直接看看认证流程。以下就是当用户输入db.auth(username, real_passwd)后发生的交互。


127.0.0.1:34142  -->> 127.0.0.1:27017 admin.  62 bytes  id:8        8

        query: { getnonce: 1.0 }  ntoreturn: -1 ntoskip: 0

127.0.0.1:27017  <<--  127.0.0.1:34142   81 bytes  id:7 7 - 8

        reply n:1 cursorId: 0

        { nonce: "df97182fb47bd6d0", ok: 1.0 }

127.0.0.1:34142  -->> 127.0.0.1:27017 admin.  152 bytes  id:9       9

        query: { authenticate: 1.0, user: "admin", nonce: "df97182fb47bd6d0", key:
"3d839522b547931057284b6e1cd3a567" }  ntoreturn: -1 ntoskip: 0

127.0.0.1:27017  <<--  127.0.0.1:34142   53 bytes  id:8 8 - 9

        reply n:1 cursorId: 0

        { ok: 1.0 }
```

- 第一步，client向server发送一个命令getnonce，向server申请一个随机值nonce，server返回一个16位的nonce。这里每次返回的值都不相同。
- 第二步，client将用户输入的明文密码通过算法生成一个key，即 key = md5(nonce + username + md5(username + ":mongo:" + real_passwd))，并将之连同用户名、nonce一起返回给server，server收到数

据，首先比对nonce是否为上次生成的nonce，然后比对 key == md5(nonce + username + pwd)。如果相同，则验证通过。

由于至始至终没有密码hash在网络上传输，而是使用了类似挑战的机制，且每一次nonce的值都不同，因此即使攻击者截取到key的值，也没用办法通过重放攻击通过认证。

然而当攻击者获取了数据库中保存的pwd hash，则认证机制就不会起到作用了。即使攻击者没有破解出pwd hash对应的密码原文。但是仍然可以通过发送md5(nonce + username + pwd)的方式直接通过server认证。这里实际上server是将用户的pwd hash当作了真正的密码去验证，并没有基于原文密码验证。在这点上和我之前分析的mysql的认证机制其实没什么本质区别。当然或许这个也不算是认证机制的弱点，但是毕竟要获取MongoDB的username和pwd的可能性会更大一些。

然而在Web的监控界面的认证中则有一些不同。当client来源不是localhost，这里的用户认证过程是基于HTTP 401的。其过程与mongo认证大同小异。但是一个主要区别是：这里的nonce并没有随机化，而是每次都是默认为"abc"。

利用这个特点，如果攻击者抓取了管理员一次成功的登录，那么他就可以重放这个数据包，直接进入Web监控页面。

同样，攻击者还可以通过此接口直接暴力破解 mongo 的用户名密码。实际上27017和28017都没有对密码猜解做限制，但Web由于无需每次获取nonce，因此将会更为简便。

JavaScript的执行与保护

MongoDB 本身最大的特点之一，就是他是使用 JavaScript 语言作为命令驱动的。黑客会比较关注这一点，因为其命令的支持程度，就是获取 MongoDB 权限之后是否能进一步渗透的关键。

JavaScript 本身的标准库其实相当弱。无论是 spidermonkey 或者是 v8 引擎，其实都没有系统操作、文件操作相关的支持。对此，MongoDB做了一定的扩展。可以看到，ls/cat/cd/hostname 甚至 runProgram 都已经在 JavaScript 的上下文中有实现。看到这里是不是已经迫不及待了？mongoshell 中试一下输入ls("./")，看看返回。

等等？结果怎么这么熟悉？哈哈，没错，其实这些 api 都是在 client 的上下文中实现的。一个小小玩笑。

那么在server端是否可以执行js呢？答案是肯定的。利用 db.eval(code) ——实际上底层执行的是 db.$cmd.findOne({$eval: code}) —— 可以在server端执行我们的js代码。

当然在server端也有js的上下文扩展。显然 mongod 考虑到了安全问题（也可能是其他原因），因此在这里并没有提供client中这么强大的功能。当然 MongoDB 还在不断更新，长期关注这个list，说不定以后就有类似 load_file/exec 之类的实现。

一劳永逸解决服务端js执行带来的问题可以使用noscripting参数。直接禁止server-side的js代码执行功能。

白话MongoDB（一）

按照官方的说法，MongoDB是一种可扩展的高性能的开源的面向文档（document-oriented ）的数据库，采用C++开发。注意mongo不是mango（芒果），这个词是从humongous中截取出来的，其野心不言而明，直指海量数据存储。和其他很多NoSQL不太一样，MongoDB背后有一个专门的商业公司在提供支持和推广，有点类似MySQL AB的模式。这一系列文章，是为入门者写的，已经对NoSQL和MongoDB有一定研究和经验的，可以略过，或者看看如有疏漏，请留言指出。

面向文档，那么什么是文档呢？很明显这不是我们常见的word文档。这里说的文档，是一种可以嵌套的数据集合。从关系数据库的范式的概念来说，嵌套是明显的反范式设计。范式设计的好处是消除了依赖，但是增加了关联，查询需要通过关联两张或者多张表来获得所需要的全部数据，但是更改操作是原子的，只需要修改一个地方即可。反范式则是增加了数据冗余来提升查询性能，但更新操作可能需要更新冗余的多处数据，需要注意一致性的问题。

一个典型的例子，如blog，关系数据库中一般可以把文章设计为一张表，评论设计为一张表，那么在页面需要展示一篇文章和其对应的评论的时候，就需要关联查询文章表和评论表。但是面向文档的设计，可以将评论作为文章的一个嵌套文档存放在一起，这不但省去了关联查询，由于存储在一起，查询的性能也可以做到更好。

MongoDB的面向文档采用的是BSON，一种类似JSON的格式，但是是二进制序列化的。如上面提到的blog的文章和评论，可以做如下设计：

```
{ 'id':1, 'author':'NinGoo', 'title':'白话MongoDB（一）', 'content':'按照官方的说法，此处省略一万字',
    comment:[ {'comment-author':'宋兵甲', 'comment-content':'有木有' } ,
              {'comment-author':'尼玛','comment-content':'伤不起啊' }
            ]
}
```

1. 相关数据存放在一起，针对性的查询可以消除join，性能比分散存储要高且方便。
2. 整个结构清晰自解析。所有字段名和值都存储，所以不需要提前设计结构，key的名字和数目可以任意指定，也就是所谓的schema-free。
3. 由于字段名在每一行每一列都需要重复存在，会带来一些额外的存储消耗，这在海量数据及字段较多的时候也需要考虑。
4. 一个document的长度有限，1.7.2之前是4MB，目前是8MB，以后可能增长到32MB。如果有更大的数据，可以使用MongoDB底层的GridFS直接作为文件存储。
5. 如果需要查找某个评论者的所有评论，则相对困难。当然，MongoDB支持任意key的索引，这也不是什么大问题。

像上面的一个结构，为一个文档（document），相当于关系数据库中的一行记录，多个文档组成一个集合（collection），相当于关系数据库的表。多个集合（collection），逻辑上组织在一起，就是数据库（database），一个MongoDB实例支持多个数据库（database）。

大部分的NoSQL产品，为追求性能，一致性等，一般只能支持简单的基于row-key的单条或者范围查询，但是MongoDB可以针对任意列的key创建索引，甚至是内嵌文档里的key，从支持的查询的灵活性上来看，更接近传统的关系数据库，同时还能在性能上向NoSQL看齐，加上支持复制，自动分片和Map/Reduce等功能，非常的吸引眼球，正在成为一款热门的海量存储产品。其背后的商业支持公司10gen，也正在不遗余力的推广，前不久还在北京专门组织了一场技术交流会。在其首页列举的典型客户里，包括foursquare，sourceforge，github等知名互联网公司和应用，当然，也包括淘宝网。

MongoDB调查总结

与关系型数据库相比，MongoDB的优点：

①弱一致性（最终一致），更能保证用户的访问速度：

举例来说，在传统的关系型数据库中，一个COUNT类型的操作会锁定数据集，这样可以保证得到“当前”情况下的精确值。这在某些情况下，例如通过ATM查看账户信息的时候很重要，但对于Wordnik来说，数据是不断更新和增长的，这种“精确”的保证几乎没有任何意义，反而会产生很大的延迟。他们需要的是一个“大约”的数字以及更快的处理速度。

但某些情况下MongoDB会锁住数据库。如果此时正有数百个请求，则它们会堆积起来，造成许多问题。我们使用了下面的优化方式来避免锁定：

每次更新前，我们会先查询记录。查询操作会将对象放入内存，于是更新则会尽可能的迅速。在主/从部署方案中，从节点可以使用“-pretouch”参数运行，这也可以得到相同的效果。

使用多个mongod进程。我们根据访问模式将数据库拆分成多个进程。

②文档结构的存储方式，能够更便捷的获取数据。

对于一个层级式的数据结构来说，如果要将这样的数据使用扁平式的，表状的结构来保存数据，这无论是在查询还是获取数据时都十分困难。

举例1：

就拿一个“字典项”来说，虽然并不十分复杂，但还是会关系到“定义”、“词性”、“发音”或是“引用”等内容。大部分工程师会将这种模型使用关系型数据库中的主键和外键表现出来，但把它看作一个“文档”而不是“一系列有关系的表”岂不更好？使用“dictionary.definition.partOfSpeech='noun'”来查询也比表之间一系列复杂（往往代价也很高）的连接查询方便且快速。

举例2：在一个关系型数据库中，一篇博客（包含文章内容、评论、评论的投票）会被打散在多张数据表中。在MongoDB中，能用一个文档来表示一篇博客，评论与投票作为文档数组，放在正文主文档中。这样数据更易于管理，消除了传统关系型数据库中影响性能和水平扩展性的“JOIN”操作。

CODE↓

```
> db.blogposts.save({ title : "My First Post", author: {name :"Jane", id :1},
  comments : [{ by: "Abe", text: "First" },
          { by : "Ada", text : "Good post" }]
})

>db.blogposts.find( { "author.name" : "Jane" } )

>db.blogposts.findOne({ title : "My First Post","author.name": "Jane",
  comments : [{ by: "Abe", text: "First" },
          { by : "Ada", text : "Good post" } ]
})
> db.blogposts.find( { "comments.by" : "Ada" } )

>db.blogposts.ensureIndex( { "comments.by" : 1 } );
```

举例③：

MongoDB是一个面向文档的数据库，目前由10gen开发并维护，它的功能丰富，齐全，完全可以替代MySQL。在使用MongoDB做产品原型的过程中，我们总结了MonogDB的一些亮点：

使用JSON风格语法，易于掌握和理解：MongoDB使用JSON的变种BSON作为内部存储的格式和语法。针对MongoDB的操作都使用JSON风格语法，客户端提交或接收的数据都使用JSON形式来展现。相对于SQL来说，更加直观，容易理解和掌握。

Schema-less，支持嵌入子文档：MongoDB是一个Schema-free的文档数据库。一个数据库可以有多个Collection，每个Collection是Documents的集合。Collection和Document和传统数据库的Table和Row并不对等。无需事先定义Collection，随时可以创建。

Collection中可以包含具有不同schema的文档记录。 这意味着，你上一条记录中的文档有3个属性，而下一条记录的文档可以有10个属性，属性的类型既可以是基本的数据类型（如数字、字符串、日期等），也可以是数组或者散列，甚至还可以是一个子文档（embeddocument）。这样，可以实现逆规范化（denormalizing）的数据模型，提高查询的速度。

图1 MongoDB是一个Schema-free的文档数据库

图2是一个例子，作品和评论可以设计为一个collection，评论作为子文档内嵌在art的comments属性中，评论的回复则作为comment子文档的子文档内嵌于replies属性。按照这种设计模式，只需要按照作品id检索一次，即可获得所有相关的信息了。在MongoDB中，不强调一定对数据进行Normalize ，很多场合都建议De-normalize，开发人员可以扔掉传统关系数据库各种范式的限制，不需要把所有的实体都映射为一个Collection，只需定义最顶级的class。MongoDB的文档模型可以让我们很轻松就能将自己的Object映射到collection中实现存储。

图2 MongoDB支持嵌入子文档

③内置GridFS，支持大容量的存储。

GridFS是一个出色的分布式文件系统，可以支持海量的数据存储。

内置了GridFS了MongoDB，能够满足对大数据集的快速范围查询。

④内置Sharding。

提供基于Range的Auto Sharding机制：一个collection可按照记录的范围，分成若干个段，切分到不同的Shard上。

Shards可以和复制结合，配合Replica sets能够实现Sharding+fail-over，不同的Shard之间可以负载均衡。查询是对客户端是透明的。客户端执行查询，统计，MapReduce等操作，这些会被MongoDB自动路由到后端的数据节点。这让我们关注于自己的业务，适当的时候可以无痛的升级。MongoDB的Sharding设计能力最大可支持约20 petabytes，足以支撑一般应用。

这可以保证MongoDB运行在便宜的PC服务器集群上。PC集群扩充起来非常方便并且成本很低，避免了“sharding”操作的复杂性和成本。

⑤第三方支持丰富。(这是与其他的NoSQL相比，MongoDB也具有的优势)

现在网络上的很多NoSQL开源数据库完全属于社区型的，没有官方支持，给使用者带来了很大的风险。

而开源文档数据库MongoDB背后有商业公司10gen为其提供供商业培训和支持。

而且MongoDB社区非常活跃，很多开发框架都迅速提供了对MongDB的支持。不少知名大公司和网站也在生产环境中使用MongoDB，越来越多的创新型企业转而使用MongoDB作为和Django，RoR来搭配的技术方案。
⑥性能优越：
在使用场合下，千万级别的文档对象，近10G的数据，对有索引的ID的查询不会比mysql慢，而对非索引字段的查询，则是全面胜出。mysql实际无法胜任大数据量下任意字段的查询，而mongodb的查询性能实在让我惊讶。写入性能同样很令人满意，同样写入百万级别的数据，mongodb比我以前试用过的couchdb要快得多，基本10分钟以下可以解决。补上一句，观察过程中mongodb都远算不上是CPU杀手。

与关系型数据库相比，MongoDB的缺点：
①mongodb不支持事务操作。
所以事务要求严格的系统（如果银行系统）肯定不能用它。(这点和优点①是对应的)
②mongodb占用空间过大。
关于其原因，在官方的FAQ中，提到有如下几个方面：
1、空间的预分配：为避免形成过多的硬盘碎片，mongodb每次空间不足时都会申请生成一大块的硬盘空间，而且申请的量从64M、128M、256M那样的指数递增，直到2G为单个文件的最大体积。随着数据量的增加，你可以在其数据目录里看到这些整块生成容量不断递增的文件。

2、字段名所占用的空间：为了保持每个记录内的结构信息用于查询，mongodb需要把每个字段的key-value都以BSON的形式存储，如果value域相对于key域并不大，比如存放数值型的数据，则数据的overhead是最大的。一种减少空间占用的方法是把字段名尽量取短一些，这样占用空间就小了，但这就要求在易读性与空间占用上作为权衡了。我曾建议作者把字段名作个index，每个字段名用一个字节表示，这样就不用担心字段名取多长了。但作者的担忧也不无道理，这种索引方式需要每次查询得到结果后把索引值跟原值作一个替换，再发送到客户端，这个替换也是挺耗费时间的。现在的实现算是拿空间来换取时间吧。

3、删除记录不释放空间：这很容易理解，为避免记录删除后的数据的大规模挪动，原记录空间不删除，只标记“已删除”即可，以后还可以重复利用。

4、可以定期运行db.repairDatabase()来整理记录，但这个过程会比较缓慢

③MongoDB没有如MySQL那样成熟的维护工具，这对于开发和IT运营都是个值得注意的地方。

###################################

Wordnik的MongoDB使用经验
http://news.cnblogs.com/n/80856/

视觉中国的NoSQL之路：从MySQL到MongoDB
http://news.cnblogs.com/n/77959/

Wordnik的MongoDB使用经验

Wordnik是一项在线字典及百科全书服务，在大约一年前，它们逐渐开始从MySQL迁移至文档型数据库MongoDB，后者是著名的NoSQL产品之一。最近Wordnik的技术团队通过官方博客分享了这12个月来使用MongoDB经验及现状。

据Wordnik技术团队描述，它们起初决定使用MongoDB，是看中了它的弱一致性（最终一致）及文档结构的存储方式。

在传统的关系型数据库中，一个COUNT类型的操作会锁定数据集，这样可以保证得到“当前”情况下的精确值。这在某些情况下，例如通过ATM查看账户信息的时候很重要，但对于Wordnik来说，数据是不断更新和增长的，这种“精确”的保证几乎没有任何意义，反而会产生很大的延迟。他们需要的是一个“大约”的数字已经更快的处理速度。

此外，Worknik的数据结构是“层级”式的，如果要将这样的数据使用扁平式的，表状的结构来保存数据，这无论是在查询还是获取数据时都十分困难：

就拿一个“字典项”来说，虽然并不十分复杂，但还是会关系到“定义”、“词性”、“发音”或是“引用”等内容。大部分工程师会将这种模型使用关系型数据库中的主键和外键表现出来，但把它看作一个“文档”而不是“一系列有关系的表”岂不更好？使用“dictionary.definition.partOfSpeech='noun'”来查询也比表之间一系列复杂（往往代价也很高）的连接查询方便且快速。

经过了一年的使用，Worknik描述了他们从MySQL全面迁移至MongoDB后的感受。

首先是性能上的提高，这也是使用MongoDB的主要原因。MongoDB解决了Worknik在使用MySQL的时候，在存储和数据查询时都遇到的一些问题。下面是一些统计数据：

· MongoDB承受了平均50万每小时的请求（包括周末和夜间），高峰期大约是4倍的量。

· MongoDB中有超过120亿个文档。

· 每个节点大约3TB数据。

· 一般情况下文档插入速度为每条8千条，峰值为每秒5万条。

· 单个Java客户端在千兆带宽下，对单个MongoDB节点的可持续的传输速度为每秒10MB。同一个客户端的四个读取器可以保持每秒40MB的读取速度。

· 各种形式的查询都比MySQL的实现要快许多：

o 示例的获取速度，从400ms减少为60ms。

o 字典项获取速度，从20ms减少为1ms。

o 文档元数据的获取速度，从30ms减少为0.1ms。

o 拼写提示的获取速度，从10ms减少为1.2ms。

Worknik表示，在压力较高的情况下，MongoDB的内置缓存机制，让系统对memcached层的每次调用节省了1-2ms，同时还剩下了许多GB的内存。此外，所有的数据不可能都在内存中，因此获取示例的60ms还包括磁盘访问时间。

其次，使用MongoDB还带来了许多灵活性，除了之前提到的文档型存储让查询变得十分迅速之外，MongoDB还带来了其他一些好处。例如以前Worknik使用集群文件系统保存音频文件，如今这些文件保存在MongoDB的GridFS中。这给IT维护带来了许多方便，例如可以使用相同的方式来维护数据和文件内容，数据库和文件也是保持同步的。

Worknik对MongoDB的可靠性也很满意，从四月起，MongoDB只重启了两次，一次是从1.4.2版升级到1.4.4版，还有一次是由于数据中心断电。

唯一可能的抱怨是对于维护性上的。MongoDB没有如MySQL那样成熟的维护工具，这对于开发和IT运营都是个值得注意的地方。不过幸运的是，MongoDB提供了许多“接入点”，因此Worknit创建了一些辅助工具，并打算开源，他们表示将在十二月份的MongoSV上提供更多信息。

在运营过程中，数据中心断电造成了很大的问题。由于断电发生在写密集的情况下，因此对主从节点都造成了损害。当时主节点正忙于将数据写回磁盘，而从节点正在通过日志获取数据。在电力回复之后，他们花费了超过24小时了来修复主节点上的数据，在这段时间内，他们将从节点提升为主节点使系统得以正常工作。

最后，Worknik还分享了一些经验：

数据尺寸：在四月份的MongoSF会议上，我们曾抱怨MongoDB耗费了4倍的数据空间。之后10gen指出了MongoDB的集合填充机制，以及Worknik某些使用场景上造成的浪费。我们将一些对象作为子文档存储，并去除一些索引之后，则大约使用了MySQL的1.5至2倍的存储空间。

锁：某些情况下MongoDB会锁住数据库。如果此时正有数百个请求，则它们会堆积起来，造成许多问题。我们使用了下面的优化方式来避免锁定：

· 每次更新前，我们会先查询记录。查询操作会将对象放入内存，于是更新则会尽可能的迅速。在主/从部署方案中，从节点可以使用“-pretouch”参数运行，这也可以得到相同的效果。

· 使用多个mongod进程。我们根据访问模式将数据库拆分成多个进程。

MongoDB是一个可扩展、高性能的下一代数据库。最新版本为1.6.3，并由10gen提供商业支持。

视觉中国的NoSQL之路：从MySQL到MongoDB

起因

视觉中国网站(www.chinavisual.com)是国内最大的创意人群的专业网站。2009年以前，同很多公司一样，我们的CMS和社区产品都构建于PHP+Nginx+MySQL之上；MySQL使用了Master+Master的部署方案；前端使用自己的PHP框架进行开发；Memcached作为缓存；Nginx进行Web服务和负载均衡；Gearman进行异步任务处理。

在传统的基于静态内容（如文章，资讯，帖子）的产品，这个体系运行良好。通过分级的缓存，数据库端实际负载很轻。2009年初，我们进行了新产品的开发。此时，我们遇到了如下一些问题。

用户数据激增：我们的MySQL某个信息表上线1个月的数据就达到千万。我们之前忽略的很多数据，在新形势下需要跟踪记录，这也导致了数据量的激增；

用户对于信息的实时性要求更高：对信息的响应速度和更新频度就要求更高。简单通过缓存解决的灵丹妙药不复存在；

对于Scale-out的要求更高：有些创新产品的增长速度是惊人的。因此要求能够无痛的升级扩展，否则一旦停机，那么用户流失的速度也是惊人的；

大量文件的备份工作：我们面向的是创意人群，产生的内容是以图片为主。需要能够对这些图片及不同尺寸的缩略图进行有效的备份管理。我们之前使用的Linux inotify+rsync的增量备份方案效果不佳；

需求变化频繁：开发要更加敏捷，开发成本和维护成本要更低，要能够快速地更新进化，新功能要在最短的周期内上线。

最初，我们试图完全通过优化现有的技术架构来解决以上问题：对数据时效性进一步分级分层缓存，减小缓存粒度；改进缓存更新机制（线上实时和线下异步更新）提高缓存命中率；尝试对业务数据的特点按照水平和垂直进行分表；使用MogileFS进行分布存储；进一步优化MySQL的性能，同时增加MySQL节点等。但很快发现，即便实施了上述方案，也很难完全解决存在的问题：过度依赖Memcached导致数据表面一致性的维护过于复杂，应用程序开发需要很小心，很多时候出现Memcached的失效会瞬间导致后端数据库压力过大；不同类型数据的特点不同，数据量差别也很大；分表的机制和方式在效率平衡上很难取舍；MogileFS对我们而言是脚小鞋大，维护成本远远超过了实际的效益；引入更多的MySQL数据库节点增大了我们的维护量，如何有效监控和管理这些节点又成了新的问题。虽然虚拟化可以解决部分问题，但还是不能令人满意；

除了MySQL，能否找到一个更为简单、轻便的瑞士军刀呢？我们的目光投向了NoSQL的方案。

候选方案

最初，对于NoSQL的候选方案，我依据关注和熟悉程度，并且在甄别和选择合适的方案时特别制定了一些原则：是否节省系统资源，对于CPU等资源是否消耗过大；客户端/API支持，这直接影响应用开发的效率；文档是否齐全，社区是否活跃；部署是否简单；未来扩展能力。按以上几点经过一段测试后，我们候选名单中剩下Redis、MongoDB和Flare。

Redis对丰富数据类型的操作很吸引人，可以轻松解决一些应用场景，其读写性能也相当高，唯一缺点就是存储能力和内存挂钩，这样如果存储大量的数据需要消耗太多的内存（最新的版本已经不存在这个问题）。

Flare的集群管理能力令人印象深刻，它可以支持节点的动态部署，支持节点的基于权重的负载均衡，支持数据分区。同时允许存储大的数据，其key的长度也不受Memcached的限制。而这些对于客户端是透明的，客户端使用Memcached协议链接到Flare的proxy节点就可以了。由于使用集群，Flare支持fail－over，当某个数据节点宕掉，对于这个节点的访问都会自动被proxy节点forward到对应的后备节点，恢复后还可以自动同步。Flare的缺点是实际应用案例较少，文档较为简单，目前只在Geek使用。

以上方案都打算作为一个优化方案，我从未想过完全放弃MySQL。然而，用MongoDB做产品的设计原型后，我彻底被征服了，决定全面从MySQL迁移到MongoDB。

为什么MongoDB可以替代MySQL？

MongoDB是一个面向文档的数据库，目前由10gen开发并维护，它的功能丰富，齐全，完全可以替代MySQL。在使用MongoDB做产品原型的过程中，我们总结了MonogDB的一些亮点：

使用JSON风格语法，易于掌握和理解：MongoDB使用JSON的变种BSON作为内部存储的格式和语法。针对MongoDB的操作都使用JSON风格语法，客户端提交或接收的数据都使用JSON形式来展现。相对于SQL来说，更加直观，容易理解和掌握。

Schema-less，支持嵌入子文档：MongoDB是一个Schema-free的文档数据库。一个数据库可以有多个Collection，每个Collection是Documents的集合。Collection和Document和传统数据库的Table和Row并不对等。无需事先定义Collection，随时可以创建。

Collection中可以包含具有不同schema的文档记录。 这意味着，你上一条记录中的文档有3个属性，而下一条记录的文档可以有10个属性，属性的类型既可以是基本的数据类型（如数字、字符串、日期等），也可以是数

组或者散列，甚至还可以是一个子文档（embed document）。这样，可以实现逆规范化（denormalizing）的数据模型，提高查询的速度。

图1 MongoDB是一个Schema-free的文档数据库

图2是一个例子，作品和评论可以设计为一个collection，评论作为子文档内嵌在art的comments属性中，评论的回复则作为comment子文档的子文档内嵌于replies属性。按照这种设计模式，只需要按照作品id检索一次，即可获得所有相关的信息了。在MongoDB中，不强调一定对数据进行Normalize ，很多场合都建议De-normalize，开发人员可以扔掉传统关系数据库各种范式的限制，不需要把所有的实体都映射为一个Collection，只需定义最顶级的class。MongoDB的文档模型可以让我们很轻松就能将自己的Object映射到collection中实现存储。

图2 MongoDB支持嵌入子文档

简单易用的查询方式：MongoDB中的查询让人很舒适，没有SQL难记的语法，直接使用JSON，相当的直观。对不同的开发语言，你可以使用它最基本的数组或散列格式进行查询。配合附加的operator，MongoDB支持范围查询，正则表达式查询，对子文档内属性的查询，可以取代原来大多数任务的SQL查询。

CRUD更加简单，支持in-place update：只要定义一个数组，然后传递给MongoDB的insert/update方法就可自动插入或更新；对于更新模式，MongoDB支持一个upsert选项，即：“如果记录存在那么更新，否则插入”。MongoDB的update方法还支持Modifier，通过Modifier可实现在服务端即时更新，省去客户端和服务端的通讯。这些modifer可以让MongoDB具有和Redis、Memcached等KV类似的功能：较之MySQL，MonoDB更加简单快速。Modifier也是MongoDB可以作为对用户行为跟踪的容器。在实际中使用Modifier来将用户的交互行为快速保存到MongoDB中以便后期进行统计分析和个性化定制。

所有的属性类型都支持索引，甚至数组：这可以让某些任务实现起来非常的轻松。在MongoDB中，“_id”属性是主键，默认MongoDB会对_id创建一个唯一索引。

服务端脚本和Map/Reduce：MongoDB允许在服务端执行脚本，可以用Javascript编写某个函数，直接在服务端执行，也可以把函数的定义存储在服务端，下次直接调用即可。MongoDB不支持事务级别的锁定，对于某些需要自定义的“原子性”操作，可以使用Server side脚本来实现，此时整个MongoDB处于锁定状态。Map/Reduce也是MongoDB中比较吸引人的特性。Map/Reduce可以对大数据量的表进行统计、分类、合并的工作，完成原先SQL的GroupBy等聚合函数的功能。并且Mapper和Reducer的定义都是用Javascript来定义服务端脚本。

性能高效，速度快： MongoDB使用c++/boost编写，在多数场合，其查询速度对比MySQL要快的多，对于CPU占用非常小。部署也很简单，对大多数系统，只需下载后二进制包解压就可以直接运行，几乎是零配置。

支持多种复制模式： MongoDB支持不同的服务器间进行复制，包括双机互备的容错方案。

Master-Slave是最常见的。通过Master-Slave可以实现数据的备份。在我们的实践中，我们使用的是Master-Slave模式，Slave只用于后备，实际的读写都是从Master节点执行。

Replica Pairs/Replica Sets允许2个MongoDB相互监听，实现双机互备的容错。

MongoDB只能支持有限的双主模式（Master-Master），实际可用性不强，可忽略。

内置GridFS，支持大容量的存储：这个特点是最吸引我眼球的，也是让我放弃其他NoSQL的一个原因。GridFS具体实现其实很简单，本质仍然是将文件分块后存储到files.file和files.chunk 2个collection中，在各个主流的driver实现中，都封装了对于GridFS的操作。由于GridFS自身也是一个Collection，你可以直接对文件的属性进行定义和管理，通过这些属性就可以快速找到所需要的文件，轻松管理海量的文件，无需费神如何hash才能避免文件系统检索性能问题， 结合下面的Auto-sharding，GridFS的扩展能力是足够我们使用了。在实践中，我们用MongoDB的GridFs存储图片和各种尺寸的缩略图。

图3 MongoDB的Auto-sharding结构

内置Sharding，提供基于Range的Auto Sharding机制：一个collection可按照记录的范围，分成若干个段，切分到不同的Shard上。Shards可以和复制结合，配合Replica sets能够实现Sharding+fail-over，不同的Shard之间可以负载均衡。查询是对客户端是透明的。客户端执行查询，统计，MapReduce等操作，这些会被MongoDB自动路由到后端的数据节点。这让我们关注于自己的业务，适当的时候可以无痛的升级。MongoDB的Sharding设计能力最大可支持约20 petabytes，足以支撑一般应用。

第三方支持丰富： MongoDB社区非常活跃，很多开发框架都迅速提供了对MongDB的支持。不少知名大公司和网站也在生产环境中使用MongoDB，越来越多的创新型企业转而使用MongoDB作为和Django，RoR来搭配的技术方案。

实施结果

实施MonoDB的过程是令人愉快的。我们对自己的PHP开发框架进行了修改以适应MongoDB。在PHP中，对MongoDB的查询、更新都是围绕Array进行的，实现代码变得很简洁。由于无需建表，MonoDB运行测试单元所需要的时间大大缩短，对于TDD敏捷开发的效率也提高了。当然，由于MongoDB的文档模型和关系数据库有很大不同，在实践中也有很多的困惑，幸运的是，MongoDB开源社区给了我们很大帮助。最终，我们使用了2周就完成了从MySQL到MongoDB的代码移植比预期的开发时间大大缩短。从我们的测试结果看也是非常惊人，数据量约2千万，数据库300G的情况下，读写2000rps，CPU等系统消耗是相当的低（我们的数据量还偏小，目前陆续有些公司也展示了他们的经典案例：MongoDB存储的数据量已超过 50亿，>1.5TB）。目前，我们将MongoDB和其他服务共同部署在一起，大大节约了资源。

一些小提示

切实领会MongoDB的Document模型，从实际出发，扔掉关系数据库的范式思维定义，重新设计类；在服务端运行的JavaScript代码避免使用遍历记录这种耗时的操作，相反要用Map/Reduce来完成这种表数据的处理；属性的类型插入和查询时应该保持一致。若插入时是字符串"1"，则查询时用数字1是不匹配的；优化MongoDB的性能可以从磁盘速度和内存着手；MongoDB对每个Document的限制是最大不超过4MB；在符合上述条件下多启用Embed Document, 避免使用DatabaseReference；内部缓存可以避免N+1次查询问题（MongoDB不支持joins）。

用Capped Collection解决需要高速写入的场合，如实时日志；大数据量情况下，新建同步时要调高oplogSize的大小，并且自己预先生成数据文件，避免出现客户端超时；Collection＋Index合计数量默认不能超过24000；当前版本（<v1.6）删除数据的空间不能被回收，如果你频繁删除数据，那么需要定期执行repairDatabase，释放这些空间。

结束语

MongoDB的里程碑是1.6版本，预计今年7月份发布，届时，MongoDB的Sharding将首次具备在生产环境中使用的条件。作为MongoDB的受益者，我们目前也在积极参与MongoDB社区活动，改进Perl/PHP对于MongoDB的技术方案。在1.6版本后也将年内推出基于MongoDB的一些开源项目。

对于那些刚刚起步，或者正在开发创新型互联网应用的公司来说，MongoDB的快速、灵活、轻量和强大扩展性，正适合我们快速开发产品，快速迭代，适应用户迅速变化和更新的种种需求。

总而言之，MongoDB是一个最适合替代MySQL的全功能的NoSQL产品，使用MongoDB＋Perl/PHP/Django/RoR的组合将很快成为开发Web2.0、3.0的产品的最佳组合，就像当年MySQL替代Oracle/DB2/Informix一样，历史总是惊人的相似，让我们拭目以待吧！

作者简介：

潘凡（nightsailer，N.S.），视觉中国网站技术总监，联合创始人，家有1狗2猫。目前负责网站平台设计和底层产品研发工作。当前关注：Apps平台设计、分布式文件存储、NoSQL、高性能后现代的Perl编程。Twitter：@nightsailer Blog：http://nightsailer.com/

Wordnik的MongoDB使用经验

Wordnik是一项在线字典及百科全书服务，在大约一年前，它们逐渐开始从MySQL迁移至文档型数据库MongoDB，后者是著名的NoSQL产品之一。最近Wordnik的技术团队通过官方博客分享了这12个月来使用MongoDB经验及现状。

据Wordnik技术团队描述，它们起初决定使用MongoDB，是看中了它的弱一致性（最终一致）及文档结构的存储方式。

在传统的关系型数据库中，一个COUNT类型的操作会锁定数据集，这样可以保证得到"当前"情况下的精确值。这在某些情况下，例如通过ATM查看账户信息的时候很重要，但对于Wordnik来说，数据是不断更新和增长的，这种"精确"的保证几乎没有任何意义，反而会产生很大的延迟。他们需要的是一个"大约"的数字已经更快的处理速度。

此外，Worknik的数据结构是"层级"式的，如果要将这样的数据使用扁平式的，表状的结构来保存数据，这无论是在查询还是获取数据时都十分困难：

就拿一个"字典项"来说，虽然并不十分复杂，但还是会关系到"定义"、"词性"、"发音"或是"引用"等内容。大部分工程师会将这种模型使用关系型数据库中的主键和外键表现出来，但把它看作一个"文档"而不是"一系列有关系的表"

岂不更好？使用“dictionary.definition.partOfSpeech='noun'”来查询也比表之间一系列复杂（往往代价也很高）的连接查询方便且快速。

经过了一年的使用，Worknik描述了他们从MySQL全面迁移至MongoDB后的感受。

首先是性能上的提高，这也是使用MongoDB的主要原因。MongoDB解决了Worknik在使用MySQL的时候，在存储和数据查询时都遇到的一些问题。下面是一些统计数据：

· MongoDB承受了平均50万每小时的请求（包括周末和夜间），高峰期大约是4倍的量。

· MongoDB中有超过120亿个文档。

· 每个节点大约3TB数据。

· 一般情况下文档插入速度为每条8千条，峰值为每秒5万条。

· 单个Java客户端在千兆带宽下，对单个MongoDB节点的可持续的传输速度为每秒10MB。同一个客户端的四个读取器可以保持每秒40MB的读取速度。

· 各种形式的查询都比MySQL的实现要快许多：

o 示例的获取速度，从400ms减少为60ms。

o 字典项获取速度，从20ms减少为1ms。

o 文档元数据的获取速度，从30ms减少为0.1ms。

o 拼写提示的获取速度，从10ms减少为1.2ms。

Worknik表示，在压力较高的情况下，MongoDB的内置缓存机制，让系统对memcached层的每次调用节省了1-2ms，同时还剩下了许多GB的内存。此外，所有的数据不可能都在内存中，因此获取示例的60ms还包括磁盘访问时间。

其次，使用MongoDB还带来了许多灵活性，除了之前提到的文档型存储让查询变得十分迅速之外，MongoDB还带来了其他一些好处。例如以前Worknik使用集群文件系统保存音频文件，如今这些文件保存在MongoDB的GridFS中。这给IT维护带来了许多方便，例如可以使用相同的方式来维护数据和文件内容，数据库和文件也是保持同步的。

Worknik对MongoDB的可靠性也很满意，从四月起，MongoDB只重启了两次，一次是从1.4.2版升级到1.4.4版，还有一次是由于数据中心断电。

唯一可能的抱怨是对于维护性上的。MongoDB没有如MySQL那样成熟的维护工具，这对于开发和IT运营都是个值得注意的地方。不过幸运的是，MongoDB提供了许多“接入点”，因此Worknit创建了一些辅助工具，并打算开源，他们表示将在十二月份的MongoSV上提供更多信息。

在运营过程中，数据中心断电造成了很大的问题。由于断电发生在写密集的情况下，因此对主从节点都造成了损害。当时主节点正忙于将数据写回磁盘，而从节点正在通过日志获取数据。在电力回复之后，他们花费了超过24小时了来修复主节点上的数据，在这段时间内，他们将从节点提升为主节点使系统得以正常工作。

最后，Worknik还分享了一些经验：

数据尺寸：在四月份的MongoSF会议上，我们曾抱怨MongoDB耗费了4倍的数据空间。之后10gen指出了MongoDB的集合填充机制，以及Worknik某些使用场景上造成的浪费。我们将一些对象作为子文档存储，并去除一些索引之后，则大约使用了MySQL的1.5至2倍的存储空间。

锁：某些情况下MongoDB会锁住数据库。如果此时正有数百个请求，则它们会堆积起来，造成许多问题。我们使用了下面的优化方式来避免锁定：

· 每次更新前，我们会先查询记录。查询操作会将对象放入内存，于是更新则会尽可能的迅速。在主/从部署方案中，从节点可以使用“-pretouch”参数运行，这也可以得到相同的效果。

· 使用多个mongod进程。我们根据访问模式将数据库拆分成多个进程。

MongoDB是一个可扩展、高性能的下一代数据库。最新版本为1.6.3，并由10gen提供商业支持。

关于MongoDB，你可能不知道的十件事

MongoDB 很简单，参照着一些常用的教程下载相应平台的二进制包、创建dbpath然后启动基本上就可以跑了。但是如果你真的打算在生产环境中使用MongoDB，还是请多进行深入的研究，下面是一位MongoDB的爱好者在参加完MongoNYC大会后总结的十个自己了解到的知识点，看看有没有你不知道的吧。

· 1.MongoDB有一个大的全局锁，这使得一个MongoDBDaemon只能同时进行一个写操作，即使是对不同collection的操作，也只得排队。

· 2.MongoDB并没有一个基于统计的查询优化器，对查询并发的执行多个不同的计划，在最快的那个返回后就终止其它任务，并将这个最快的计划指导查询。当然不是每次查询都执行多个不同计划，这个会隔一段时间执行一次。

· 3.Mongos只有在你使用Sharding时才需要，在不用Sharding时，实际上是客户端来实现负载均衡的。

· 4.MongoDB不仅仅只有Replica Sets，还有传统的Master-Slave模式。（实际上你想配置成Master-Master也完全可以）

· 5.MongoDB的同步机器支持“slave-delay”参数，这个参数指定Slave机器延迟Master多长时间。这个参数用来做准备非常合适。

· 6.MongoDB 使用了mmap，在32位系统下数据文件只能达到2G，所以32位系统下的MongoDB玩玩就够了。

· 7.MongoDB会在日志里记录执行时间超过100ms的操作，实际上这个是可以灵活配置的。

· 8.MongoDB可以运行一些耗时较长的统计分析任务。

· 9.MongoDB不支持多主对单从的架构（这个应该是支持的，原文作者可能理解错了）。

· 10.MongoDB的Replica Sets 模式下，可以设置一些节点为Arbiter，它们不存储数据，只在需要重新选Primary时参与投票。

Riak与MongoDB的对比

机制与概念上的异同

Riak和MongoDB在使用特性上有下面几个相同点：

- 都是文档型的数据模型
- 具体存储方式都不是以文档型进行存储
- 写性能及写吞吐都很高

虽然上面几点看起来二者挺像，但在内部实现上两者却是相去甚远。比如Riak是一个分布式的存储，而MongoDB可以理解为是一个单一的数据库系统，同时加上了Replication和Sharding功能。MongoDB的内部数据结构上还是文档，而Riak是不用关心存储内容的二进制。MongoDB提供GridFS机制来存储二进制内容，而Riak的二进制内容与普通内容存储方式一样。MongoDB的写入方式是 in-place方式，修改一个文档是原子性的，而Riak是通过quormNRW的机制保证写入操作安全性的。

- http://www.mongodb.org/display/DOCS/Home
- http://blog.mongodb.org/post/248614779/fast-updates-with-mongodb-update-in-place
- http://www.mongodb.org/display/DOCS/Updating#Updating-Update

复制备份及横向扩展

Riak主要通过一致性hash算法来实现其数据的复制及分片，一致性hash机制是Riak的核心思想之一。在Riak中，每个节点都是对等的，所以其不存在单点故障。

- Add Nodes to Riak
- Consistent Hashing

而MongoDB在1.6版本后也推出了强有力的复制备份功能

1.主从复制

- http://www.mongodb.org/display/DOCS/Master+Slave

2.Replica Sets

Replica Sets是MongoDB的重头功能之一，它让几个节点组成一个集合，在这个集合中的节点中有一个主机提供写入，其它节点会从主机上备份数据，主机故障后会自动在从机中选取产生新的主机。

- http://www.mongodb.org/display/DOCS/Replica+Sets

而在数据分片上，MongoDB提供了一种叫auto-sharding的机制，使数据在多个节点间可以均匀分布，提供动态添加删除节点的功能。

- http://www.mongodb.org/display/DOCS/Sharding
- http://www.mongodb.org/display/DOCS/Sharding+Introduction
- http://en.wikipedia.org/wiki/Sharding

数据分片的自动调整

Riak基于一致性hash策略，在有节点从hash环上移除后，其数据会自动分摊整个环上的其它节点上。其负载也就被均匀分摊了。而MongoDB也支持在Sharding中摘除节点后的自动数据迁移，具体见此文：

- http://www.mongodb.org/display/DOCS/Configuring+Sharding#ConfiguringSharding-Removingashard

性能对比

Riak的存储引擎本身是作为插件的形式挂载的，Riak支持BitCask，InnoDB和LevelDB等存储引擎，使用默认的BitCask引擎，你可以在性能和数据持久化的选择上进行调节。相比之下，MongoDB由于采用了mmap机制，如果索引和热数据能被内存完全装下，那么其操作基本上相当于内存操作，所以MongoDB的当机性能是相当高的。

- http://www.mongodb.org/display/DOCS/Durability+and+Repair
- http://blog.mongodb.org/post/381927266/what-about-durability

数据模型

Riak的数据存储没有特定的格式需求，它允许你存储不同体积的文档型数据，另外Riak还可以在数据间创建link来为数据建立关联。

- Data Storage in Riak

MongoDB的数据是以BSON格式存储的，你可以在MongoDB中存储任意JSON格式的文档，在存储时会被转成BSON进行存储，另外二进制数据也可以转换成相应的一种BSON数据类型进行存储，GridFS正是基于这种类型来实现的。

查询语句及分布式操作

Riak只提供key-value式的数据操作接口，它支持key-value数据的各种操作，也支持link-walking和MapReduce操作，像二级索引这种东西，在Riak里是不存在的，因为Riak根本不关心它存的数据是什么样的，value对它来说只是一串数据。

- *https://wiki.basho.com/display/RIAK/MapReduce*

MongoDB提供与关系型数据库类似的各种数据操作（除了关联查询），其索引机制更是与关系型数据库几乎一模一样。同时MongoDB也提供MapReduce的操作接口，用以处理一些批量任务。

- http://www.mongodb.org/display/DOCS/Indexes
- http://www.mongodb.org/display/DOCS/Querying
- http://www.mongodb.org/display/DOCS/MapReduce

冲突解决策略

Riak使用vector-clock机制来进行冲突检测，所以其冲突解决的选择权是留给应用层来做的。应用层可以决定两个用户对同一行数据的更新哪一个会胜出。

- Vector Clocks

MongoDB使用的是最近更新者胜出的方式，相对来说更简单直接。

- http://www.mongodb.org/display/DOCS/Atomic+Operations

API

Riak提供给非Erlang的客户端两种操作方式

- 1. HTTP
- 2. Protocol Buffers

MongoDB的协议是自己制定的一套特有协议，其客户端由其所属的10gen公司开发并维护，基本主流的语言都有相应的官方客户端。

BSON特性探讨及基于其特性的MongoDB优化

BSON是由10gen开发的一个数据格式，目前主要用于MongoDB中，是MongoDB的数据存储格式。BSON基于JSON格式，选择JSON进行改造的原因主要是JSON的通用性及JSON的schemaless的特性。

BSON主要会实现以下三点目标：

1.更快的遍历速度

对JSON格式来说，太大的JSON结构会导致数据遍历非常慢。在JSON中，要跳过一个文档进行数据读取，需要对此文档进行扫描才行，需要进行麻烦的数据结构匹配，比如括号的匹配，而BSON对JSON的一大改进就是，它会将JSON的每一个元素的长度存在元素的头部，这样你只需要读取到元素长度就能直接seek到指定的点上进行读取了。

MongoDB优化：对于MongoDB来说，由于采用了MMAP来做内存与数据文件的映射，在更新或者获取Document的某一个字段时，如果需要先读取其前面的所有字段，会导致物理内存由于读操作被加载到不必要的字段上，导致资源的不合理分配。而采用BSON只需要读到相应的位置然后跨过无用内容读取需要内容即可。

2.操作更简易

对JSON来说，数据存储是无类型的，比如你要修改基本一个值，从9到10，由于从一个字符变成了两个，所以可能其后面的所有内容都需要往后移一位才可以。而使用BSON，你可以指定这个列为数字列，那么无论数字从9长到10还是100，我们都只是在存储数字的那一位上进行修改，不会导致数据总长变大。当然，在MongoDB中，如果数字从整形增大到长整型，还是会导致数据总长变大的。

MongoDB优化：所以使用MongoDB的一个技巧是将长度可能变化的字段尽量命名靠后（MongoDB在update操作后会将字段按key值按字母顺序重排，所以靠后的意思是按a－z的顺序取名）。这样在更新的时候如果导致数字变长，不需要移动大量数据。一个典型的例子是如果用二进制类型存储文件时，如果文件名或者文件描述可能会变长，那么尽量将这个字段取名靠后是一个明智的选择，否则在文件名或文件描述字段变化时，会导致移动很长的二进制数据，造成不必要的浪费。

3.增加了额外的数据类型

JSON是一个很方便的数据交换格式，但是其类型比较有限。BSON在其基础上增加了“byte array”数据类型。这使得二进制的存储不再需要先base64转换后再存成JSON。大大减少了计算开销和数据大小。

当然，在有的时候，BSON相对JSON来说也并没有空间上的优势，比如对{“field”:7}，在JSON的存储上7只使用了一个字节，而如果用BSON，那就是至少4个字节（32位）

MongoDB优化：在MongoDB中，如果你的字段是数字型，并且涉及到数据加减操作的，那么建议存在int型，但如果是一个固定不变的数字，并且在四位以下的话，可以考虑存成字符串类型。这样会节省空间。

目前在10gen的努力下，BSON已经有了针对多种语言的编码解码包。并且都是Apache 2 license下开源的。并且还在随着MongoDB进一步地发展。关于BSON，你可以在其官方网站 bsonspec.org 上获取更多信息。

MySQL和MongoDB设计实例对比

本文转载自火丁笔记，文章举了一个数据库设计的例子，对MySQL和MongoDB两种存储工具，分别进行了数据库结构设计，在MongoDB的设计上，利用了MongoDB的 schema-free的特性。

虽然文中的例子不一定是最优的选择。但分享此文，希望提醒大家，换个存储，不仅是换一个存储，更重要的是换一套思维。

MySQL是关系型数据库中的明星，MongoDB是文档型数据库中的翘楚。下面通过一个设计实例对比一下二者：假设我们正在维护一个手机产品库，里面除了包含手机的名称，品牌等基本信息，还包含了待机时间，外观设计等参数信息，应该如何存取数据呢？

如果使用MySQL的话，应该如何存取数据呢？

如果使用MySQL话，手机的基本信息单独是一个表，另外由于不同手机的参数信息差异很大，所以还需要一个参数表来单独保存。

```
CREATE TABLE IF NOT EXISTS `mobiles` (

    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,

    `name` VARCHAR(100) NOT NULL,

    `brand` VARCHAR(100) NOT NULL,

    PRIMARY KEY (`id`)

);


CREATE TABLE IF NOT EXISTS `mobile_params` (

    `id` int(10) unsigned NOT NULL AUTO_INCREMENT,

    `mobile_id` int(10) unsigned NOT NULL,

    `name` varchar(100) NOT NULL,

    `value` varchar(100) NOT NULL,

    PRIMARY KEY (`id`)

);


INSERT INTO `mobiles` (`id`, `name`, `brand`) VALUES

(1, 'ME525', '摩托罗拉'),

(2, 'E7'   , '诺基亚');


INSERT INTO `mobile_params` (`id`, `mobile_id`, `name`, `value`) VALUES

(1, 1, '待机时间', '200'),

(2, 1, '外观设计', '直板'),

(3, 2, '待机时间', '500'),

(4, 2, '外观设计', '滑盖');
```

注：为了演示方便，没有严格遵守关系型数据库的范式设计。

如果想查询待机时间大于100小时，并且外观设计是直板的手机，需按照如下方式查询：

```
SELECT * FROM `mobile_params` WHERE name = '待机时间' AND value > 100;

SELECT * FROM `mobile_params` WHERE name = '外观设计' AND value = '直板';
```

注：参数表为了方便，把数值和字符串统一保存成字符串，实际使用时，MySQL允许在字符串类型的字段上进行数值类型的查询，只是需要进行类型转换，多少会影响一点性能。

两条SQL的结果取交集得到想要的MOBILE_IDS，再到mobiles表查询即可：

```
SELECT * FROM `mobiles` WHERE mobile_id IN (MOBILE_IDS)
```

如果使用MongoDB的话，应该如何存取数据呢？

如果使用MongoDB的话，虽然理论上可以采用和MySQL一样的设计方案，但那样的话就显得无趣了，没有发挥出MongoDB作为文档型数据库的优点，实际上使用MongoDB的话，和MySQL相比，形象一点来说，可以合二为一：

```
db.getCollection("mobiles").ensureIndex({

    "params.name": 1,

    "params.value": 1

});


db.getCollection("mobiles").insert({

    "_id": 1,

    "name": "ME525",

    "brand": "摩托罗拉",

    "params": [

        {"name": "待机时间", "value": 200},

        {"name": "外观设计", "value": "直板"}

    ]

});


db.getCollection("mobiles").insert({

    "_id": 2,

    "name": "E7",

    "brand": "诺基亚",

    "params": [

        {"name": "待机时间", "value": 500},

        {"name": "外观设计", "value": "滑盖"}

    ]

});
```

如果想查询待机时间大于100小时，并且外观设计是直板的手机，需按照如下方式查询：

```
db.getCollection("mobiles").find({

    "params": {

        $all: [

            {$elemMatch: {"name": "待机时间", "value": {$gt: 100}}},

            {$elemMatch: {"name": "外观设计", "value": "直板"}}

        ]

    }

});
```

注：查询中用到的$all，$elemMatch等高级用法的详细介绍请参考官方文档中相关说明。

MySQL需要多个表，多次查询才能搞定的问题，MongoDB只需要一个表，一次查询就能搞定，对比完成，相对MySQL而言，MongoDB显得更胜一筹，至少本例如此。

关于NoSQL，你必须知道的九件事

1. 理解ACID与BASE的区别（ACID是关系型数据库强一致性的四个要求，而BASE是NoSQL数据库通常对可用性及一致性的弱要求原则，它们的意思分别是，ACID：atomicity， consistency， isolation, durability;BASE：BasicallyAvailable, Soft-state, Eventually Consistent。同时有意思的是ACID在英语里意为酸，BASE意思为碱）

2. 理解持久化与非持久化的区别。这么说是因为有的NoSQL系统是纯内存存储的。

3. 你必须意识到传统有关系型数据库与NoSQL系统在数据结构上的本质区别。传统关系型数据库通常是基于行的表格型存储，而NoSQL系统包括了列式存储（Cassandra）、key/value存储（Memcached）、文档型存储（CouchDB）以及图结构存储（Neo4j）

4. 与传统关系数据库有统一的SQL语言操作接口不同，NoSQL系统通常有自己特有的API接口。

5. 在架构上，你必须搞清楚，NoSQL系统是被设计用于成百上千台机器的集群中的，而非共享型数据库系统的架构。

6. 在NoSQL系统中，可能你得习惯一下不知道你的数据具体存在何处的情况。

7. 在NoSQL系统中，你最好习惯它的弱一致性。”eventually consistent”(最终一致性)正是BASE原则中的重要一项。比如在Twitter，你在Followers列表中经常会感受到数据的延迟。

8. 在NoSQL系统中，你要理解，很多时候数据并不总是可用的。

9. 你得理解，有的方案是拥有分区容忍性的，有的方案不一定有。

antirez 的Redis 宣言！

1. Redis 是一个操作数据结构的语言工具，它提供基于TCP的协议以操作丰富的数据结构。在Redis中，数据结构这个词的意义不仅表示在某种数据结构上的操作，更包括了结构本身及这些操作的时间空间复制度。

2. Redis 定位于一个内存数据库，正是由于内存的快速访问特性，才使得Redis能够有如此高的性能，才使得Redis能够轻松处理大量复杂的数据结构，Redis会尝试其它的存储方面的选择，但是永远不会改变它是一个内存数据库的角色。

3. Redis 使用基础的API操作基础的数据结构，Redis的API与数据结构一样，都是一些最基础的元素，你几乎可以将任何信息交互使用此API格式表示。作者调侃说，如果有其它非人类的智能生物存在，他们也能理解Redis的API。因为它是如此的基础。

4. Redis 有着诗一般优美的代码，经常有一些不太了解Redis 原则的人会建议Redis采用一些其它人的代码，以实现一些Redis 未实现的功能，但这对我们来说就像是非要给《红楼梦》接上后四十回一样。（作者此处用了莎士比亚的比喻）

5. Redis 始终避免复杂化，我们认为设计一个系统的本质，就是与复杂化作战。我们不会为了一个小功能而往源码里添加上千行代码，解决复杂问题的方法就是让复杂问题永远不要提复杂的问题。

6. Redis 支持两个层成的API，第一个层面包含部分操作API，但它支持用于分布式环境下的Redis。第二个层面的API支持更复杂的multi-key操作。它们各有所长，但是我们不会推出两者都支持的API，但我们希望能够提供实例间数据迁移的命令，并执行multi-key操作。

7. 我们以优化代码为乐，我们相信编码是一件辛苦的工作，唯一对得起这辛苦的就是去享受它。如果我们在编码中失去了乐趣，那最好的解决办法就是停下来。我们决不会选择让Redis不好玩的开发模式。

NoSQL开篇——为什么要使用NoSQL

NoSQL在2010年风生水起，大大小小的Web站点在追求高性能高可靠性方面，不由自主都选择了NoSQL技术作为优先考虑的方面。今年伊始，InfoQ中文站有幸邀请到凤凰网的孙立先生，为大家分享他之于NoSQL方面的经验和体会。

非常荣幸能受邀在InfoQ开辟这样一个关于NoSQL的专栏，InfoQ是我非常尊重的一家技术媒体，同时我也希望借助InfoQ，在国内推动NoSQL的发展，希望跟我一样有兴趣的朋友加入进来。这次的NoSQL专栏系列将先整体介绍NoSQL，然后介绍如何把NoSQL运用到自己的项目中合适的场景中，还会适当地分析一些成功案例，希望有成功使用NoSQL经验的朋友给我提供一些线索和信息。

NoSQL概念

随着web2.0的快速发展，非关系型、分布式数据存储得到了快速的发展，它们不保证关系数据的ACID特性。NoSQL概念在2009年被提了出来。NoSQL最常见的解释是“non-relational”，“Not Only SQL”也被很多人接受。（“NoSQL”一词最早于1998年被用于一个轻量级的关系数据库的名字。）

NoSQL被我们用得最多的当数key-value存储，当然还有其他的文档型的、列存储、图型数据库、xml数据库等。在NoSQL概念提出之前，这些数据库就被用于各种系统当中，但是却很少用于web互联网应用。比如cdb、qdbm、bdb数据库。

传统关系数据库的瓶颈

传统的关系数据库具有不错的性能，高稳定型，久经历史考验，而且使用简单，功能强大，同时也积累了大量的成功案例。在互联网领域，MySQL成为了绝对靠前的王者，毫不夸张的说，MySQL为互联网的发展做出了卓越的贡献。

在90年代，一个网站的访问量一般都不大，用单个数据库完全可以轻松应付。在那个时候，更多的都是静态网页，动态交互类型的网站不多。

到了最近10年，网站开始快速发展。火爆的论坛、博客、sns、微博逐渐引领web领域的潮流。在初期，论坛的流量其实也不大，如果你接触网络比较早，你可能还记得那个时候还有文本型存储的论坛程序，可以想象一般的论坛的流量有多大。

Memcached+MySQL

后来，随着访问量的上升，几乎大部分使用MySQL架构的网站在数据库上都开始出现了性能问题，web程序不再仅仅专注在功能上，同时也在追求性能。程序员们开始大量的使用缓存技术来缓解数据库的压力，优化数据库的结构和索引。开始比较流行的是通过文件缓存来缓解数据库压力，但是当访问量继续增大的时候，多台web机器通过文件缓存不能共享，大量的小文件缓存也带了了比较高的IO压力。在这个时候，Memcached就自然的成为一个非常时尚的技术产品。

Memcached作为一个独立的分布式的缓存服务器，为多个web服务器提供了一个共享的高性能缓存服务，在Memcached服务器上，又发展了根据hash算法来进行多台Memcached缓存服务的扩展，然后又出现了一致性hash来解决增加或减少缓存服务器导致重新hash带来的大量缓存失效的弊端。当时，如果你去面试，你说你有Memcached经验，肯定会加分的。

Mysql主从读写分离

由于数据库的写入压力增加，Memcached只能缓解数据库的读取压力。读写集中在一个数据库上让数据库不堪重负，大部分网站开始使用主从复制技术来达到读写分离，以提高读写性能和读库的可扩展性。Mysql的master-slave模式成为这个时候的网站标配了。

分表分库

随着web2.0的继续高速发展，在Memcached的高速缓存，MySQL的主从复制，读写分离的基础之上，这时MySQL主库的写压力开始出现瓶颈，而数据量的持续猛增，由于MyISAM使用表锁，在高并发下会出现严重的锁问题，大量的高并发MySQL应用开始使用InnoDB引擎代替MyISAM。同时，开始流行使用分表分库来缓解写压力和数据增长的扩展问题。这个时候，分表分库成了一个热门技术，是面试的热门问题也是业界讨论的热门技术问题。也就在这个时候，MySQL推出了还不太稳定的表分区，这也给技术实力一般的公司带来了希望。虽然MySQL推出了MySQL Cluster集群，但是由于在互联网几乎没有成功案例，性能也不能满足互联网的要求，只是在高可靠性上提供了非常大的保证。

MySQL的扩展性瓶颈

在互联网，大部分的MySQL都应该是IO密集型的，事实上，如果你的MySQL是个CPU密集型的话，那么很可能你的MySQL设计得有性能问题，需要优化了。大数据量高并发环境下的MySQL应用开发越来越复杂，也越来越具有技术挑战性。分表分库的规则把握都是需要经验的。虽然有像淘宝这样技术实力强大的公司开发了透明

的中间件层来屏蔽开发者的复杂性，但是避免不了整个架构的复杂性。分库分表的子库到一定阶段又面临扩展问题。还有就是需求的变更，可能又需要一种新的分库方式。

MySQL数据库也经常存储一些大文本字段，导致数据库表非常的大，在做数据库恢复的时候就导致非常的慢，不容易快速恢复数据库。比如1000万4KB大小的文本就接近40GB的大小，如果能把这些数据从MySQL省去，MySQL将变得非常的小。

关系数据库很强大，但是它并不能很好的应付所有的应用场景。MySQL的扩展性差（需要复杂的技术来实现），大数据下IO压力大，表结构更改困难，正是当前使用MySQL的开发人员面临的问题。

NOSQL的优势

易扩展

NoSQL数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展。也无形之间，在架构的层面上带来了可扩展的能力。

大数据量，高性能

NoSQL数据库都具有非常高的读写性能，尤其在大数据量下，同样表现优秀。这得益于它的无关系性，数据库的结构简单。一般MySQL使用Query Cache，每次表的更新Cache就失效，是一种大粒度的Cache，在针对web2.0的交互频繁的应用，Cache性能不高。而NoSQL的Cache是记录级的，是一种细粒度的Cache，所以NoSQL在这个层面上来说就要性能高很多了。

灵活的数据模型

NoSQL无需事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。如果是非常大数据量的表，增加字段简直就是一个噩梦。这点在大数据量的web2.0时代尤其明显。

高可用

NoSQL在不太影响性能的情况，就可以方便的实现高可用的架构。比如Cassandra，HBase模型，通过复制模型也能实现高可用。

总结

NoSQL数据库的出现，弥补了关系数据（比如MySQL）在某些方面的不足，在某些方面能极大的节省开发成本和维护成本。

MySQL和NoSQL都有各自的特点和使用的应用场景，两者的紧密结合将会给web2.0的数据库发展带来新的思路。让关系数据库关注在关系上，NoSQL关注在存储上。

你需要知道的NoSQL数据库10件事

NoSQL的5个优势

1.弹性扩展

多年来，数据库管理员一直依赖于向上扩展(scale up)－随着数据库负载的增加购买更大的数据库服务器—而不是向外扩展－随着负载的增加将数据库分不到多个不同的主机上。然而，随着每秒事务数与可用性需求的提高，以及数据库往云或虚拟环境的迁移，向外扩展到廉价硬件的经济优势越来越难以抵挡。

RDBMS或许比较难以在廉价的集群上进行向外扩展，但是，NoSQL数据库的新品从设计之初就是为了利用新节点的优势进行透明扩展，他们通常在设计时就考虑使用低成本的廉价硬件。

2.大数据量

在过去10年，与每秒事务数的增长超出了认知一样，存储的数据的规模也出现了极大的增长。O'Reilly明智的称此为"数据的工业革命。"RDBMS的容量也在增长以匹配这些数据的增长，但是，与每秒事务数一样，单个RDBMS可有效管理的数据规模限制让部分企业越来越难以忍受。今天，大规模数据量可以交由NoSQL系统来处理，比如Hadoop，超过目前最大的RDBMS可以管理的数据规模。

3. 再见了，DBA（回头见，DBA？）

这些年，虽然RDBMS的提供商宣称推出了很多的可管理性方面的改进，高端的RDBMS系统还是只能交由昂贵的、高度受训的DBA来进行维护。高端RDBMS系统从设计到安装以及后续的调优，都需要DBA们深度介入。

从理论上，通常，NoSQL数据库的最初的设计目标就是更少的管理介入：自动修复、数据分布以及更简单的数据模型，从而更少的管理与调优需求。实际上，关于DBA将死的谣言很可能被略微放大了。对于任何关键的数据存储，总是需要有人来关心它的性能以及可用性。

4.经济性

NoSQL数据库通常使用廉价服务器集群来管理暴增的数据与事务规模，而RDBMS倾向于依赖昂贵的专有服务器与存储系统。其结果是，NoSQL数据库的每GB数据或每秒事务数的成本要远远低于RDBMS，使得你可以以更低的价格来存储与处理更多的数据。

5.灵活的数据模型

在大量的生产环境数据库中，变更管理是一个非常棘手的问题。哪怕是对数据模型的很小的变更，在RDBMS中也需要进行小心的管理，甚至还需要停机或降低服务级别。

在数据模型的限制这一点上，NoSQL数据库要宽松的多，或者完全不存在。 NoSQL的键值存储（Keyvalue Store）与文档数据库（Document Database）允许应用在一个数据单元中存入它想要的任何结构。即使是定义更加严格的基于BigTable的NoSQL数据库，通常也允许创建新的字段而不致带来麻烦。

其结果是，应用的变更与数据库结构的变更不需要绑定在一个变更单元中进行管理。理论上，这可以提高应用的迭代速度，然而，显然，如果应用无法管理数据的完整性，它将带来不良的副作用。

NoSQL的5个挑战

NoSQL数据库的可能性空间引发了大量的关注，但是，在它们成为企业级应用的主流之前，还有大量的障碍有待克服。下面是几个主要的挑战。

1.成熟度

RDBMS已经存在了很长一段时间。 NoSQL的支持者认为它们的年纪是它们过时的象征，但是，对于大部分CIO(首席信息官)来讲，RDBMS的成熟度是可以让人放心的。通常，RDBMS 系统都很稳定，功能也很丰富。相比而言，大部分NoSQL的替代品都还处于前一生产环境阶段，还有大量的关键特性有待实现。

生活在科技前沿对于大部分开发人员来讲，是令人兴奋的，但是，企业在实施时必须非常谨慎。

2.支持力度

企业还希望获得保证，当关键系统出现故障时，他们可以获得及时而有效的支持。所有的RDBMS提供商都在竭尽全力地为企业提供高级别的支持。

相比而言，大部分的NoSQL系统都是开源项目，虽然，每一个NoSQL数据库通常都会有一家或多家公司为其提供支持，这些公司通常都是小的创业公司，没有能力提供全球的支持，没有足够的支持资源，或者没有类似于Oracle、Microsoft或者IBM的信用。

3.分析与商业智能

NoSQL数据库经过不断的演化，已经可以满足现代的Web 2。0应用的扩展需求。相应地，它们的大部分功能集也旨在满足这些应用的需求。然而，应用程序中的数据的价值，要超出一个典型的Web应用的插入-阅读-更新-删除的周期。从公司数据库中挖掘信息以提高公司的效率与竞争力的业务，以及商业智能(BI)是所有大中型公司的关键议题。

NoSQL数据库提供了新型的工具来做即时的查询与分析。哪怕是一个简单的查询，也需要可观的编程技能，通常使用的BI工具都无法访问NoSQL数据库。

稍显宽慰的是，还有类似于HIVE与PIG的这类解决方案，通过它们可以较为简单地访问Hadoop集群中的数据，或许最终，可以较为简单的访问其他的 NoSQL数据库。Quest软件公司开发一个产品，Toad For Cloud Database，它提供了对各种不同的NoSQL数据库的即时查询功能。

4.管理

NoSQL的设计目标可能是提供零-管理的解决方案，但是，当前的现实是，此目标远远没有实现。目前的NoSQL系统需要大量的技能来进行安装，以及需要大量的努力来进行维护。

5.专业技能

坦率的讲，目前世界上有上百万的程序员非常熟悉RDBMS的原理与编程，他们分布在各种业务场景中。相比而言，几乎每一个NoSQL开发人员都还处于学习阶段。随着时间的流逝，这种状况将得到解决，但是，现在，寻找一个有经验的RDBMS开发人员与RDBMS管理员要比寻找一个NoSQL专家要容易的多。

结论

NoSQL数据库正在成为越来越多的数据库环境的重要的组成部分，如果使用得当的话，它可以提供实实在在的收益。然而，企业在推进它们的使用时需要非常谨慎，需要明白这些数据库的相关内在限制与问题。

NoSQL生态系统大检阅 不同特性大比拼

虽然NoSQL很火热，但是真正应用NoSQL的用户不多。本文将为大家以对比的形式来介绍不同NoSQL产品的特点，希望对大家有所帮助。

空前的数据量正在驱动商业寻找传统关系型数据库的替代方案,它已经为我们服务30多年了(今年5月份ACM刚刚给关系型数据庆祝40岁生日).总体来讲,这些替代方案就是目前知名的“NoSQL数据库.”

关系型数据库的基本问题是无法处理许多现代的工作负载.有三个具体的问题领域:向外扩展(Scaleout)类似于Digg(3TB的绿色徽章数据)或Facebook(50T的收件箱搜索数据)或Ebay(总共2PB的数据)的数据集,单机性能限制以及僵化的概要设计.

商业上(包含Rackspace Cloud公司)需要寻找新的方式来存储并扩展大规模的数据.我最近写了一篇关于Cassandra的文章,一个我们投入了资源的非关系型数据库.还有另外一些正在运作中的非关系型数据库,它们汇总在一起被我们称为”NoSQL运动”.

“NoSQL”这个术语实际上是由一个Rackspace的员工Eric Evans最先提出的,当时来自Last.fm网站的Johan Oskarsson提议组织一次开源分布式数据库的研讨会.这个名称与概念就一起流行了起来.

有些人反对NoSQL这个说法,因为它听起来像是仅仅表明了我们不做什么,而不是我们在做什么.事实确实是这样,我也基本同意此说法,但是这个术语仍然有其价值,因为当关系型数据库是你所知道的唯一工具时,每个问题看起来都像个拇指(俗语,如果你手里有一个锤子,你看到什么都是钉子,译者补充).NoSQL这个术语起码让人们知道还有其他的选项可供选择.但是,当关系型数据库是解决问题的最佳工具时,我们并不是反关系型数据库者;它的涵义应该是“不仅仅有SQL(Not Only SQL)”而不是“不再有SQL(No SQL atall)”.

有关NoSQL名称的一个真实的忧虑是,它是如此大的一个概念,以致于差异巨大的设计都可以涵盖其中.如果在讨论各种产品时没有搞清楚这一点,就会导致概念混乱.因此,我建议大家沿着下面三个维度来思考这些数据库选项:可伸缩性(scalability)、数据模型与查询模型(data andquery model)以及持久化设计(persistencedesign).

我选择了10种NoSQL数据库作为示例.这不是一份详尽的清单,但是这里讨论的概念对于评估其他的NoSQL数据库也至关重要.

可伸缩性(Scalability)

通过使用复制,就可以轻易扩展读的规模,因此,每当我在此文中谈到规模伸缩(scaling),都是表示通过自动分区将数据分布到多台机器以扩展写的规模.我们将做这种事情的系统称为“分布式数据库”.它们包括Cassandra、HBase、Riak、Scalaris、Voldemort以及其他很多类似的系统.如果你的写容量或写数据大小已经无法在一台机器上进行处理,如果你不想自己手工来管理分区的话,这些就是你的唯一选项了.(你不会这么做吧?)

人们使用分布式数据库主要关注两件事情: 1) 是否支持多个数据中心以及2) 能否在对应用透明的前提下往正在运行的集群中添加新机器的能力.

非分布式NoSQL数据库包括CouchDB、MongoDB、Neo4j、Redis以及TokyoCabinet.它们可作为分布式系统的持久层;MongoDB提供了受限制的数据分片(Sharding)功能,CouchDB也有一个独立的Lounge项目来支持做类似的分片功能,TokyoCabinet可用作Voldemort的存储引擎.

数据模型与查询模型

NoSQL数据库之间的数据模型与查询API千差万别.

部分重点内容介绍:

Cassandra与HBase共同使用的ColumnFamily模型都是受到Google的Bigtable论文第2节的启发.(Cassandra丢弃了历史版本,并增加了超级列(SuperColumn)的概念).在这两个系统中,都有与你之前看到的关系型数据库类似的行/列概念,但是此处的行是稀疏的行:你想要一行有多少列,一行就可以有多少列,这些列并不需要事先定义好.

键值(Key/value)模型是最简单也最容易实现的模型,但是,如果你仅想对值(Value)的一部分进行查询/更新时,它的效率会比较低.要想在一个分布式的键值上,实现更加复杂的结构也会非常困难.

文档数据库实际上是更高级的键/值(Key/Value)数据库,允许在每个键上关联嵌套的值.相对于每次简单地返回整个BLOB(二进制大对象)来讲,文档数据库支持更高效的查询.

Neo4j拥有一个非常独特的数据模型,它以节点与边的形式在图中存储对象与关系.对于适合这个模型(例如,分层数据)的查询,它的性能可能会达到其替代选项的1000倍.

Scalaris的独特之处在于,它可以提供跨越多个键的分布式事务.(关于一致性与可用性的权衡的讨论超出了本文的范围,但是,在评估分布式系统时,它也是需要记住的一方面.)

持久化设计

关于持久化设计,我的意思是“数据在内部是如何存储的?”

持久化模型可以为我们提供大量关于这些数据库适合处理多大工作负载的信息.

内存数据库非常非常快(单台机器上的Redis可以处理100,000次操作/秒),但是无法处理超过可用内存的数据集.持久性(Durability,数据不会由于服务器崩溃或停电而丢失)也是个问题;在两次刷新到磁盘的时间间隔内预期数据丢失量可能非常大.Scalaris是我们此列表中唯一的内存数据库,它通过复制来解决持久性的问题,但是,由于它不支持跨越多个数据中心,因此,如果遇到类似电源故障一类的问题数据仍将非常脆弱.

在为了持久性写入一个仅可追加的提交日志之后,Memtable与SSTable会缓冲内存中的写操作.在接受了足够多的写操作之后(Memtable达到一定的阈值),就会对memtable中的数据进行排序,并一次性写入到磁盘,写入的文件就是一个“sstable.”这样它就可以提供接近于内存处理的性能,因为它不涉及任何检索操作,同时又可以避免纯粹在内存中的方法那样遭遇持久性问题.(在前面引用的Bigtable论文的第5.3与5.4两节,以及论文日志结构的合并树(TheLog-Structuredmerge-tree)中对此都有详细的描述)

几乎从有数据库开始,B-树就开始在数据库中使用了.它们提供健壮的索引支持,但是在旋转磁盘(仍然是目前最经济实用的存储介质)上, 它的性能表现比较差,因为它读写任何内容都会涉及到多次磁盘检索.

CouchDB的仅可做追加操作的B-树(Append-OnlyB-tree)是一个比较有趣的变体,它以限制CouchDB并发写(one write at a time)的代价避免了其检索的开销.

结论

NoSQL运动在2009年取得了爆发性的效果,因为越来越多的企业需要处理大规模的数据.Rackspace Cloud公司很高兴在NoSQL运动扮演了一个较早期的角色,还会持续为Cassandra投入资源并支持与NoSQL East类似的活动.

NoSQL数据库利弊分析：五大优势五大挑战

关系数据库模型已经流行了几十年了，但是一种新类型的数据库——被称为NoSQL，正在引起企业的注意。下面是关于它的优势和劣势的一个概述。二十多年以来，对数据库管理来说，关系数据库(RDBMS)模型一直是一个占统治地位的数据库模型。但是，今天，非关系数据库，“云”数据库，或“NoSQL”数据库作为关系数据库以外的一些选择，正在引起大家的广泛关注。在这篇文章里，我们将主要关注那些非关系的NoSQL数据库的十大利弊：包括五大优势和五大挑战。

NoSQL的五大优势

1.灵活的可扩展性

多年以来，数据库管理员们都是通过“纵向扩展”的方式(当数据库的负载增加的时候，购买更大型的服务器来承载增加的负载)来进行扩展的，而不是通过“横向扩展”的方式(当数据库负载增加的时候，在多台主机上分配增加的负载)来进行扩展。但是，随着交易率和可用性需求的增加，数据库也正在迁移到云端或虚拟化环境中，“横向扩展”在commodity hardware方面的经济优势变得更加明显了，对各大企业来说，这种“诱惑”是无法抗拒的。

在commodity clusters上，要对RDBMS做“横向扩展”，并不是很容易，但是各种新类型的NoSQL数据库主要是为了进行透明的扩展，来利用新节点而设计的，而且，它们通常都是为了低成本的commodity hardware而设计的。

2.大数据

在过去的十年里，正如交易率发生了翻天覆地的增长一样，需要存储的数据量也发生了急剧地膨胀。O'Reilly把这种现象称为：“数据的工业革命”。为了满足数据量增长的需要，RDBMS的容量也在日益增加，但是，对一些企业

来说，随着交易率的增加，单一数据库需要管理的数据约束的数量也变得越来越让人无法忍受了。现在，大量的“大数据”可以通过NoSQL系统(例如：Hadoop)来处理，它们能够处理的数据量远远超出了最大型的RDBMS所能处理的极限。

3.“永别了”！DBA们！

在过去的几年里，虽然一些RDBMS供应商们声称在可管理性方面做出了很多的改进，但是高端的RDBMS系统维护起来仍然十分昂贵，而且还需要训练有素的DBA们的协助。DBA们需要亲自参与高端的RDBMS系统的设计，安装和调优。

NoSQL数据库从一开始就是为了降低管理方面的要求而设计的：从理论上来说，自动修复，数据分配和简单的数据模型的确可以让管理和调优方面的要求降低很多。但是，DBA的死期将至的谣言未免有些过于夸张了。总是需要有人对关键性的数据库的性能和可用性负责的。

4.经济

NoSQL数据库通常使用廉价的commodity servers集群来管理膨胀的数据和事务数量，而RDBMS通常需要依靠昂贵的专有服务器和存储系统来做到这一点。使用NoSQL，每GB的成本或每秒处理的事务的成本都比使用RDBMS的成本少很多倍，这可以让你花费更低的成本存储和处理更多的数据。

5.灵活的数据模型

对于大型的生产性的RDBMS来说，变更管理是一件很令人头痛的事情。即使只对一个RDBMS的数据模型做了很小的改动，也必须要十分小心地管理，也许还需要停机或降低服务水平。NoSQL数据库在数据模型约束方面是更加宽松的，甚至可以说并不存在数据模型约束。NoSQL的主键值数据库和文档数据库可以让应用程序在一个数据元素里存储任何结构的数据。即使是规定更加严格的基于“大表”的NoSQL数据库(例如：Cassandra, HBase)通常也允许创建新列，这并不会造成什么麻烦。

应用程序变更和数据库模式的变更并不需要作为一个复杂的变更单元来管理。从理论上来说，这可以让应用程序迭代的更快，但是，很明显，如果应用程序无法维护数据的完整性，那么这会带来一些不良的副作用。

NoSQL的五大挑战

NoSQL的种种承诺引发了一场热潮，但是在它们得到主流的企业的青睐以前，它们还有许多困难需要克服。下面是NoSQL需要面对的一些挑战。

1.成熟度

RDBMS系统已经发展很长时间了。NoSQL的拥护者们认为，RDBMS系统那超长的发展的年限恰恰表示它们已经过时了，但是对于大多数的CIO们来说，RDBMS的成熟度更加令它们放心。大多数情况下，RDBMS系统更加稳定，而且功能也更加丰富。相比之下，大多数的NoSQL数据库都是pre-production版本，许多关键性的功能还有待实现。

对于大多数开发者来说，处于技术的最前沿的确是很令人兴奋的，但是企业应该怀着极端谨慎的态度来处理此事。

2.支持

企业都希望能得到这样的保证：如果一个关键性的系统出现问题了，他们可以获得及时有效的支持。所有的RDBMS供应商都在竭尽全力地提供高水平的企业支持。

相反，大多数的NoSQL系统都是开源项目，虽然对于每个NoSQL数据库来说，通常也会有一个或多个公司对它们提供支持，但是那些公司通常是小型的创业公司，在支持的范围，支持的资源，或可信度方面，它们和Oracle, Microsoft或IBM是无法相提并论的。

3.分析和商业智能化

NoSQL数据库现在已经可以满足现代的Web2.0应用程序的高度的可扩展性的要求了。这直接导致的结果是，它们的大多数功能都是面向这些应用程序而设计的。但是，在一个应用程序中，具有商业价值的数据早就已经超出了一个标准的Web应用程序需要的“插入-读取-更新-删除”的范畴了。在公司的数据库中进行商业信息的挖掘可以提高企业的效率和竞争力，而且对于所有的中到大型的公司来说，商业智能化(BI)一直是一个至关重要的IT问题。

NoSQL数据库几乎没有提供什么专用的查询和分析工具。即使是一个简单的查询，也要求操作者具有很高超的编程技术，而且，常用的BI工具是无法连接到NoSQL的。

像HIVE或PIG那样的新出现的一些解决方案在这方面可以提供一些帮助，它们可以让访问Hadoop集群中的数据变得更加容易，最后也许还会支持其他的NoSQL数据库。Quest软件已经开发了一个产品——Toad for Cloud Databases——它给各种NoSQL数据库提供了专用的查询功能。

4.管理

NoSQL的设计目标是提供一个“零管理”的解决方案，但是目前来说，还远远没有达到这个目标。安装NoSQL还是需要很多技巧的，同时，维护它也需要付出很多的努力。

5.专业知识

毫不夸张地说，全世界有数百万的开发者，他们都对RDBMS的概念和编程方法很熟悉，在每个业务部门中都有这样的开发者。相反，几乎每一个NoSQL开发者都正处于学习状态中。虽然这种情况会随着时间的推移而改变，但是现在，找到一些有经验的RDBMS程序员或管理员要比找到一个NoSQL专家容易的多。

结论

NoSQL数据库正在逐渐地成为数据库领域中不可或缺的一部分，如果使用方法得当的话，能获得很多的好处。但是，企业应该谨慎行事，要充分地认识到这些数据库的一些限制和问题。

SQL or NoSQL——云计算环境中该选择谁？

NoSQL和SQL之间真正的差异是什么？实质上，是因为不同的访问模式导致了NoSQL和SQL可扩展性和性能上的不同。

NoSQL只允许数据在受限的预定义模式访问。例如DHT （Distributed Hash Table）通过hashtable API访问。其他NoSQL数据服务访问模式同样受限。因此可扩展性和性能结构是可预测和可靠的。

而在SQL中，访问模式预先是不知道的，SQL是一种通用语言，允许数据以各种方式访问，程序员也对SQL语句的执行能力控制有限。

换句话说，在SQL中，数据模型不执行特定的工作方式与数据。强调建立数据完整性、简洁性、标准化和抽象化。这对于所有大型复杂的应用极为重要。

为什么是NoSQL

NoSQL提供的方法对于SQL数据库来说有巨大的优势。因为它允许应用程序扩展的新的水平。新的数据服务基于真正可扩展的结构和体系构建云、构建分布式。这对于应用开发来说是非常有吸引力的。无需DBA，无需复杂的SQL查询。

这是不小的问题，一个好程序员自由选择一个数据模型，使用熟悉的工具写应用程序，减少对他人的依赖于，并测试和优化的代码，而不做猜测或一个黑盒（DB）的计数。

这些都是NoSQL运动的所有主要优势，但NoSQL也非万能，具体而言，数据模型的选择、接口规范以及当前面临的新业务比如移动业务数据的处理问题，都是NoSQL无法回避的。

NoSQL绝非万能

数据模型

如果没有一个统一的、定义良好的数据模型，无论采用何种技术都有缺陷。

SQL的数据模型定义了高度结构化的数据结构，以及对这些结构之间关系的严格定义。在这样的数据模型上执行的查询操作会比较局限，而且可能会导致复杂的数据遍历操作。但是数据结构的复杂性及查询的复杂性，会导致系统产生如下的一些限制：比如当数据量增长到一台机器已经不能容纳，我们需要将不同的数据表分布到不同的机器；如果你的结构化数据并没有那么强，或者对每一行数据的要求比较灵活，那可能关系型的数据模型就太过严格了；再有，使用简单的查询语言可能会导致应用层的逻辑更复杂，但是这样可以将存储系统的工作简单化，让它只需要响应一些简单的请求。

此外，NoSQL数据库并非是唯一适合存储大量数据或大型数据，显然，通过良好的分区设计，SQL数据库也可以获得极好的扩展性。

接口和互操作问题

不可否认，NoSQL的数据服务接口还有待规范。比如DHT，这是一个简单的接口，但仍旧没有标准的语义。每个DHT服务都使用其自己的一套接口。另一个大问题是不同的数据结构，如 DHT和binary tree，只是作为一个例

子，共享数据对象。所有这些服务中，指针没有内在的语义。事实上，这些服务中，处理互操作性是开发者的职责，这一点很很重要，尤其是当需要数据被多个服务访问时。一个简单的例子：后台工作由Java实现，Web服务类工作由PHP实现，数据可以被轻易地从两个域访问数据吗？显然，人们可以使用Web服务作为前端数据访问层，但是，NoSQL有可能让事情变得更复杂，并降低了业务敏捷性，灵活性和性能，同时增加了开发工作量。

移动业务

在移动业务领域，需要一套工具，这套工具不仅要有可扩展性，而且还易于管理并且稳定，并在云上有一个固定的设置服务器。当系统出现问题的时候，可以不需要通过判断整个系统或开发平台来诊断问题，而是通过远程访问——这正是运维经理们所要面对的问题，但是在目前NoSQL所能提供的服务功能来看，很难实现，即便是Amazon的托管环境。

SQL和NoSQL如何结合？

总而言之，在NoSQL和SQL的选择上，需要了解到以下内容：

数据模型及操作模型：你的应用层数据模型是行、对象还是文档型的呢？这个系统是否能支持你进行一些统计工作呢？

可靠性：当你更新数据时，新的数据是否立刻写到持久化存储中去了？新的数据是否同步到多台机器上了？

扩展性：你的数据量有多大，单机是否能容下？你的读写量求单机是否能支持？

分区策略：考虑到你对扩展性，可用性或者持久性的要求，你是否需要一份数据被存在多台机器上？你是否需要知道数据在哪台机器上，以及你能否知道。

一致性：你的数据是否被复制到了多台机器上，这些分布在不同点的数据如何保证一致性？

事务机制：你的业务是否需要ACID的事务机制？

单机性能：如果你打算持久化的将数据存在磁盘上，哪种数据结构能满足你的需求（你的需求是读多还是写多）？写操作是否会成为磁盘瓶颈？

负载可评估：对于一个读多写少的应用，诸如响应用户请求的web应用，我们总会花很多精力来关注负载情况。你可能需要进行数据规模的监控，对多个用户的数据进行汇总统计。你的应用场景是否需要这样的功能呢？

使用NoSQL架构实现SQL数据库？

使用NoSQL的基础架构实现SQL数据库是一个很好的解决方案。一个SQL数据库是可扩展、易管理，云就绪、高度可用的，完全建立在NoSQL的基础结构（分布式）上，但仍然提供SQL数据库的所有优势，如互操作性，定义良好的语义以及更多。

这种混合结构也许不如纯粹的NoSQL的服务，但足以满足需要更稳定系统、可扩展性和云服务的80%的市场需求。

这种解决办法还允许很容易地迁移现有的应用到云环境，从而保护相关组织在这些应用上所付出的巨大的投资。

在我看来，构建于NoSQL基础之上的SQL数据库，可以为那些在其成长期间期望灵活、高效的客户提供最高的价值。

NoSQL架构实践（一）——以NoSQL为辅

怎么样把NoSQL引入到我们的系统架构设计中，需要根据我们系统的业务场景来分析，什么样类型的数据适合存储在NoSQL数据库中，什么样类型的数据必须使用关系数据库存储。明确引入的NoSQL数据库带给系统的作用，它能解决什么问题，以及可能带来的新的问题。下面我们分析几种常见的NoSQL架构。

（一）NoSQL作为镜像

不改变原有的以MySQL作为存储的架构，使用NoSQL作为辅助镜像存储，用NoSQL的优势辅助提升性能。

图 1 -NoSQL为镜像（代码完成模式 ）

```
//写入数据的示例伪代码 //data为我们要存储的数据对象 data.title="title"; data.name="
name"; data.time="2009-12-01 10:10:01"; data.from="1"; id=DB.Insert(data);//写入
MySQL数据库 NoSQL.Add(id,data);//以写入MySQL产生的自增id为主键写入NoSQL数据库
```

如果有数据一致性要求，可以像如下的方式使用

```
//写入数据的示例伪代码 //data为我们要存储的数据对象 bool status=false; DB.startTransaction();//开始事务 id=DB.Insert(data);//写入MySQL数据库 if(id>0) { status=NoSQL.Add(id,data);//以写入MySQL产生的自增id为主键写入NoSQL数据库 } if(id>0 && status==true){ DB.commit();//提交事务 }else{ DB.rollback();//不成功，进行回滚 }
```

上面的代码看起来可能觉得有点麻烦，但是只需要在DB类或者ORM层做一个统一的封装，就能实现重用了，其他代码都不用做任何的修改。

这种架构在原有基于MySQL数据库的架构上增加了一层辅助的NoSQL存储，代码量不大，技术难度小，却在可扩展性和性能上起到了非常大的作用。只需要程序在写入MySQL数据库后，同时写入到NoSQL数据库，让MySQL和NoSQL拥有相同的镜像数据，在某些可以根据主键查询的地方，使用高效的NoSQL数据库查询，这样就节省了MySQL的查询，用NoSQL的高性能来抵挡这些查询。

图 2 -NoSQL为镜像（同步模式）

这种不通过程序代码，而是通过MySQL把数据同步到NoSQL中，这种模式是上面一种的变体，是一种对写入透明但是具有更高技术难度一种模式。这种模式适用于现有的比较复杂的老系统，通过修改代码不易实现，可能引起新的问题。同时也适用于需要把数据同步到多种类型的存储中。

MySQL到NoSQL同步的实现可以使用MySQL UDF函数，MySQL binlog的解析来实现。可以利用现有的开源项目来实现，比如：

- MySQL memcached UDFs：从通过UDF操作Memcached协议。
- 国内张宴开源的mysql-udf-http：通过UDF操作http协议。

有了这两个MySQL UDF函数库，我们就能通过MySQL透明的处理Memcached或者Http协议，这样只要有兼容Memcached或者Http协议的NoSQL数据库，那么我们就能通过MySQL去操作以进行同步数据。再结合lib_mysqludf_json，通过UDF和MySQL触发器功能的结合，就可以实现数据的自动同步。

（二）MySQL和NoSQL组合

MySQL中只存储需要查询的小字段，NoSQL存储所有数据。

图 3 -MySQL和NoSQL组合

```
//写入数据的示例伪代码 //data为我们要存储的数据对象 data.title="title"; data.name="name"; data.time="2009-12-01 10:10:01"; data.from="1"; bool status=false; DB.startTransaction();//开始事务 id=DB.Insert("INSERT INTO table (from) VALUES (data.from)");//写入MySQL数据库,只写from需要where查询的字段 if(id>0){ status=NoSQL.Add(id,data);//以写入MySQL产生的自增id为主键写入NoSQL数据库 } if(id>0 && status==true) { DB.commit();//提交事务 }else{ DB.rollback();//不成功，进行回滚 }
```

把需要查询的字段，一般都是数字，时间等类型的小字段存储于MySQL中，根据查询建立相应的索引，其他不需要的字段，包括大文本字段都存储在NoSQL中。在查询的时候，我们先从MySQL中查询出数据的主键，然后从NoSQL中直接取出对应的数据即可。

这种架构模式把MySQL和NoSQL的作用进行了融合，各司其职，让MySQL专门负责处理擅长的关系存储，NoSQL作为数据的存储。它有以下优点：

- 节省MySQL的IO开销。由于MySQL只存储需要查询的小字段，不再负责存储大文本字段，这样就可以节省MySQL存储的空间开销，从而节省MySQL的磁盘IO。我们曾经通过这种优化，把MySQL一个40G的表缩减到几百M。
- 提高MySQl Query Cache缓存命中率。我们知道query cache缓存失效是表级的，在MySQL表一旦被更新就会失效，经过这种字段的分离，更新的字段如果不是存储在MySQL中，那么对query cache就没有任何影响。而NoSQL的Cache往往都是行级别的，只对更新的记录的缓存失效。
- 提升MySQL主从同步效率。由于MySQL存储空间的减小，同步的数据记录也减小了，而部分数据的更新落在NoSQL而不是MySQL，这样也减少了MySQL数据需要同步的次数。
- 提高MySQL数据备份和恢复的速度。由于MySQL数据库存储的数据的减小，很容易看到数据备份和恢复的速度也将极大的提高。
- 比以前更容易扩展。NoSQL天生就容易扩展。经过这种优化，MySQL性能也得到提高。

MySQL与NoSQL——SQL与NoSQL的融合

写这一篇内容的原因是MySQL5.6.2突然推出了memcached的功能。NoSQL to InnoDB with Memcached的出现，可以看出NoSQL对关系数据库的确产生了巨大的影响，个人觉得这是一个非常大的进步，可以让开发人员更加方便的使用NoSQL和关系数据库。NoSQL一般被认为性能高于关系数据库，那么直接在InnoDB之上提供NoSQL功能并和MySQL共存是否是一个更好的选择呢？

MySQL withHandlerSocket

去年在twitter上看到HandlerSocket的出现，并宣称性能是Memcached的两倍时，非常令人吃惊，居然可以达到750000qps。接着HandlerSocket成为NoSQL领域谈论的焦点之一， 大量的人开始想要尝试，并做过一些自己的性能测试。 下图是HandlerSocket的结构图：

图1 HandlerSocket结构图（来源于官方）

HandlerSocket的出现，给我们眼前一亮的感觉。原来InnoDB的性能已经足够好，并可以直接提供NoSQL的功能。最大的好处就是可以共享MySQL的功能，DBA以前的经验一样可以用。但是有些小小的风险：

•HandlerSocket没有与MySQL一起发布版本，因此对于使用MyISAM引擎的用户是无缘的。不过现在Percona-Server已经集成了HandlerSocket，可以非常方便的使用。

•目前大规模的成功案例并不多，国内也只有少部分公司在尝试，我知道的有飞信开放平台，据说还不错。

•官方给出的测试数据在应用场景上其实并不充分，至少测试的场景跟我们实际使用的场景相差很大。但是毫无疑问， HandlerSocket的性能比直接使用MySQL肯定要高效得多。

InnoDB with Memcached

也许是因为HandlerSocket的火爆的冲击，也许是受HandlerSocket的启发，MySQL开始关注NoSQL领域的应用，并在MySQL5.6.2版本增加了通过Memcached协议直接访问原生Innodb API的功能。

InnoDB with Memcached是在提供MySQL服务的同一进程中提供Memcached服务 ，这与HandlerSocket的架构模式几乎是一样的。虽然目前InnoDB with Memcached还是预览版本，但是我个人更看好它，因为：

•它使用Memcached协议，并同时支持文本和二进制协议，在client的选择和成熟度上就要胜出许多；

•其支持的三种cache模式，不但可以省去开发中使用Memcached来缓存数据的麻烦，并且具有更好的可靠性和数据一致性；

•在应用程序中，可以使用高效的memcached协议来操作数据，同时也可以使用sql进行复杂的查询操作；

注意：目前通过memcached的更新操作不会记录到binlog中，未来的版本会支持。

图二 InnoDB withMemcached

Memcached and MySQLCluster

显而易见，我们会想到MySQL Cluster结合Memcached是一个更好的组合，MySQL Cluster提供了99.999%高可用性，并真正提供了去中心化的无缝高可扩展性。还有什么比这更人兴奋的呢。

MySQL已经提供了这样的功能，源代码在这里。这里有一个O'Reilly MySqlConference大会的PPT演示 ，你也可以看下这个功能开发者的一篇博客。

图三 NDB with Memcached

MySQL Cluster虽然具有高可靠性和无缝扩展的优势，但是对于复杂SQL查询的效率却不能令人满意。不过对于仅仅依赖于key-value查询和写入的海量数据存储需求，MySQL Cluster with Memcached应该是个很好的选择。

总结

Memcached协议由于其简单、协议轻量、存在大量的client，所以提供兼容Memcached协议的产品比较占据先天的优势。

MySQL提供NoSQL的功能，个人觉得并不是MySQL耐不住寂寞，而是的确在响应用户的需求。我前面的文章也说过，“NoSQL只是一个概念，并不是一个数据库 产品，MySQL也可以是NoSQL”，现在也正应了这句话。NoSQL从架构上就约束了开发者的架构和开发方式，从而提高扩展性和性能，而NoSQL和MySQL的融合，也同时提供了复杂查询功能。

虽然MySQL提供了NoSQL功能，如果你要尝试的话，你的数据库设计必须从NoSQL从发，然后再考虑SQL查询功能。

SQL与NoSQL的融合的确会给开发者带来方便，比如最近很流行的Mongodb，它吸引开发最大的点就是支持简单的关系查询。SQL与NoSQL的融合可能是未来很多数据库产品的一个趋势。但是纯NoSQL数据库的优势也是显著的，就是他的简单、高效、易扩展。

解读NoSQL技术代表之作Dynamo

NoSQL背后的两种模式

NoSQL其实并不是什么妖魔鬼怪，相反，NoSQL的真谛其实应该是Not Only SQL，其产生背景是在数据量和访问量逐渐增大的情况下，人为地去添加机器或者切分数据到不同的机器，变得越来越困难，人力成本越来越高，于是便开始有了这样的项目，它们的本意是提高数据存储的自动化程度，减少人为干预的时间，让负载更加均匀等。在国际上，真正的代表之作有来自Google的 BigTable 和Amazon的Dynamo，他们分别使用了不同的基本原理。

MapReduce

这是历史最久的一种模型，典型的代表是BigTable。Map表示映射，Reduce表示化简。MapReduce通过把对数据集的大规模操作分发给网络上的每个节点实现可靠性（Map）；每个节点会周期性地把完成的工作和状态的更新报告回来（Reduce）。大多数分布式运算可以抽象为MapReduce操作。Map是把输入Input分解成中间的Key/Value对，Reduce把Key/Value合成最终输出Output。这两个函数由程序员提供给系统，下层设施把Map和Reduce操作分布在集群上运行。

Dynamo

这里我把Dynamo专门归纳成为了一种，其原因是它与MapReduce有很大的不同，自成一派。先说一下历史，Amazon于2006年推出了自己的云存储服务S3，2007年其CTO公布了S3的设计方案，从此江湖中就不再太平了，开源项目一个个如雨后春笋般地出现了。比较常见的有Facebook开发的Cassandra（如果没有记错，在去年浏览他们项目网页的时候，上面还写着他们之中的一个开发人员是Dynamo的设计人员，现在风头紧，去掉了），还有Linkedin的voldemort，而在国内话，有豆瓣网的beansDB，人人网的nuclear等等。这里我主要讨论的也是Dynamo的方案细节。

入门基础

Dynamo的意思是发电机，顾名思义，这一整套的方案都像发电机一样，源源不断地提供服务，永不间断。以下内容看上去有点教条，但基本上如果你要理解原理，这每一项都是必须知道的。

CAP原则

先来看历史，Eric A. Brewer教授，Inktomi公司的创始人，也是berkeley大学的计算机教授，Inktomi是雅虎搜索现在的台端技术核心支持。最主要的是，他们（Inktomi公司）在最早的时间里，开始研究分布计算。CAP原则的提出，可以追溯到2000年的时候（可以想象有多么早！），Brewer教授在一次谈话中，基于他运作Inktomi以及在伯克利大学里的经验，总结出了CAP原则（文末参考资料中有其演讲资料链接）。图一是来自Brewer教授当年所画的图：

图一：CAP原则当年的PPT

Consistency（一致性）：即数据一致性，简单的说，就是数据复制到了N台机器，如果有更新，要N机器的数据是一起更新的。
Availability（可用性）：好的响应性能，此项意思主要就是速度。
Partition tolerance（分区容错性）：这里是说好的分区方法，体现具体一点，简单地可理解为是节点的可扩展性。

定理：任何分布式系统只可同时满足二点，没法三者兼顾。
忠告：架构师不要将精力浪费在如何设计能满足三者的完美分布式系统，而是应该进行取舍。

DHT——分布式哈希表

DHT（Distributed Hash Table，分布式哈希表），它是一种分布式存储寻址方法的统称。就像普通的哈希表，里面保存了key与value的对应关系，一般都能根据一个key去对应到相应的节点，从而得到相对应的value。

这里随带一提，在DHT算法中，一致性哈希作为第一个实用的算法，在大多数系统中都使用了它。一致性哈希基本解决了在P2P环境中最为关键的问题——如何在动态的网络拓扑中分布存储和路由。每个节点仅需维护少量相邻节点的信息，并且在节点加入/退出系统时，仅有相关的少量节点参与到拓扑的维护中。至于一致性哈希的细节就不在这里详细说了，要指明的一点是，在Dynamo的数据分区方式之后，其实内部已然是一个对一致性哈希的改造了。

进入Dynamo的世界

有了上面一章里的两个基础介绍之后，我们开始进入Dynamo的世界。

Dynamo的数据分区与作用

在Dynamo的实现中提到一个关键的东西，就是数据分区。 假设我们的数据的key的范围是0到2的64次方（不用怀疑你的数据量会超过它，正常甚至变态情况下你都是超不过的，甚至像伏地魔等其他类Dynamo系统是使用的 2的32次方），然后设置一个常数，比如说1000，将我们的key的范围分成1000份。然后再将这1000份key的范围均匀分配到所有的节点（s个节点），这样每个节点负责的分区数就是1000/s份分区。

如图二，假设我们有A、B、C三台机器，然后将我们的分区定义了12个。

图二：三个节点分12个区的数据的情况

因为数据是均匀离散到这个环上的（有人开始会认为数据的key是从1、2、3、4……这样子一直下去的，其实不是的，哈希计算出来的值，都是一个离散的结果），所以我们每个分区的数据量是大致相等的。从图上我们可以得出，每台机器都分到了三个分区里的数据，并且因为分区是均匀的，在分区数量是相当大的时候，数据的分布会更加的均匀，与此同时，负载也被均匀地分开了（当然了，如果硬要说你的负载还是只集中在一个分区里，那就不是在这里要讨论的问题了，有可能是你的哈希函数是不是有什么样的问题了）。

为什么要进行这样的分布呢，分布的好处在于，在有新机器加入的时候，只需要替换原有分区即可，如图三所示：

图三：加入一个新的节点D的情况

同样是图二里的情况，12个分区分到ABC三个节点，图三中就是再进入了一个新的节点D，从图上的重新分布情况可以得出，所有节点里只需要转移四分之一的数据到新来的节点即可，同时，新节点的负载也伴随分区的转移而转移了（这里的12个分区太少了，如果是1200个分区甚至是12000个分区的话，这个结论就是正确的了，12个分区只为演示用）。

从Dynamo的NRW看CAP法则

在Dynamo系统中，第一次提出来了NRW的方法。
N：复制的次数；
R：读数据的最小节点数；
W：写成功的最小分区数。
这三个数的具体作用是用来灵活地调整Dynamo系统的可用性与一致性。

举个例子来说，如果R=1的话，表示最少只需要去一个节点读数据即可，读到即返回，这时是可用性是很高的，但并不能保证数据的一致性，如果说W同时为1的 话，那可用性更新是最高的一种情况，但这时完全不能保障数据的一致性，因为在可供复制的N个节点里，只需要写成功一次就返回了，也就意味着，有可能在读的这一次并没有真正读到需要的数据（一致性相当的不好）。如果W=R=N=3的话，也就是说，每次写的时候，都保证所有要复制的点都写成功，读的时候也是都读到，这样子读出来的数据一定是正确的，但是其性能大打折扣，也就是说，数据的一致性非常的高，但系统的可用性却非常低了。如果R + W > N能够保证我们“读我们所写”，Dynamo推荐使用322的组合。

Dynamo系统的数据分区让整个网络的可扩展性其实是一个固定值（你分了多少区，实际上网络里扩展节点的上限就是这个数），通过NRW来达到另外两个方 向上的调整。

Dynamo的一些增加可用性的补救

针对一些经常可能出现的问题，Dynamo还提供了一些解决的方法。

第一个是hinted handoff数据的加入：在一个节点出现临时性故障时，数据会自动进入列表中的下一个节点进行写操作，并标记为handoff数据，在收到通知需要原节点恢复时重新把数据推回去。这能使系统的写入成功大大提升。

第二个是向量时钟来做版本控制：用一个向量（比如说[a,1]表示这个数据在a节点第一次写入）来标记数据的版本，这样在有版本冲突的时候，可以追溯到出现问题的地方。这可以使数据的最终一致成为可能。（Cassandra未用vector clock，而只用client timestamps也达到了同样效果。）

第三个是Merkle tree来提速数据变动时的查找：使用Merkle tree为数据建立索引，只要任意数据有变动，都将快速反馈出来。

第四个是Gossip协议：一种通讯协议，目标是让节点与节点之间通信，省略中心节点的存在，使网络达到去中心化。提高系统的可用性。

后记

Dynamo的理论对CAP原则里的可扩展性做到了很方便的实现，通过创造性的NRW来平衡系统的可用性和一致性，增加了系统在实际情况下遇到问题的可选择方案。可以相像，在NoSQL的道路上，这只是个开端，在分布式计算的道路上，已经是MapReduce之后的再次革命。

企业中的NoSQL

什么是NoSQL——快速回顾

像许多关注这一领域的人一样，我不喜欢从本质上将SQL与NoSQL这一术语对立起来。同时我对该术语现有的解释"Not Only SQL"也不甚满意。对我来说，我们这里所讨论的并非是是否使用SQL。**(相反的是，我们仍然可以选择类似SQL这样的查询接口(缺少对join等的支持)来与这些数据库交互，使用现有的资源和技术来管理开发伸缩性和可维护性。)** 这一运动是要找到存储和检索数据的其他高效的途径，而不是盲目地在任何情况下都把关系数据库当作万金油。因此，我认为'Non Relational Database'(非关系型数据库)能够更好的表达这一思想。

无论采用哪个名字，“非关系型数据库”这一范围所传达出来的“囊括所有”类型的意味，使得这一概念比较模糊(并且它还是否定型的)。这又使得人们(特别是企业中的决策者)对于哪些是属于这个范围，哪些不是，更重要的是，对他们来说这到底意味着什么，感到非常迷惑。

为了解答这些疑问，我尝试通过以下几点特征的描述，来刻画“非关系型数据库”的内在本质。

所谓“非关系型数据库”指的是

1. 使用松耦合类型、可扩展的数据模式来对数据进行逻辑建模(Map，列，文档，图表等)，而不是使用固定的关系模式元组来构建数据模型。
2. 以遵循于CAP定理（能保证在一致性，可用性和分区容忍性三者中中达到任意两个）的跨多节点数据分布模型而设计，支持水平伸缩。这意味着对于多数据中心和动态供应（在生产集群中透明地加入/删除节点）的必要支持，也即弹性(Elasticity)。
3. 拥有在磁盘或内存中，或者在这两者中都有的，对数据持久化的能力，有时候还可以使用可热插拔的定制存储。
4. 支持多种的'Non-SQL'接口(通常多于一种)来进行数据访问。

围绕着图中四个特征的（数据持久性、逻辑数据模型、数据分布模型和接口）“非关系型数据库”的各种变形，在最近的一些文章中有详尽的描述，并且在因特网上有着广泛的传播。所以我就不做过多繁复的描述，而是通过一些例子对关键的方向进行总结，供快速参考：

接口——REST (HBase，CouchDB，Riak等)，MapReduce(HBase，CouchDB，MongoDB，Hypertable等)，Get/Put(Voldemort，Scalaris等)，Thrift (HBase，Hypertable，Cassandra等)，语言特定的API(MongoDB)。

逻辑数据模型——面向键值对的(Voldemort，Dynomite等)，面向Column Family的(BigTable，HBase，Hypertable等)，面向文档的(Couch DB，MongoDB等)，面向图的(Neo4j， Infogrid等)

数据分布模型——一致性和可用性(HBase，Hypertable， MongoDB等)， 可用性和可分区性(Cassandra等)。**一致性和可分区性的组合会导致一些非额定的节点产生可用性的损失。有趣的是目前还没有一个“非关系型数据库”支持这一组合。**

数据持久性——基于内存的(如Redis，Scalaris， Terrastore)，基于磁盘的(如MongoDB，Riak等)，或内存及磁盘二者的结合(如HBase，Hypertable，Cassandra)。存储的类型有助于我们辨别该解决方案适用于哪种类型。然而，在大多数情况下人们发现基于组合方案的解决方案是最佳的选择。既能通过内存数据存储支持高性能，又能在写入足够多的数据后存储到磁盘来保证持续性。

如何将其与企业IT融合

如今的企业中，并非所有用例都直观地倾向于使用关系型数据库，或者都需要严格的ACID属性(特别是一致性和隔离性)。在80年代及90年代，绝大部分存储在企业数据库里的数据都是结构化的业务事务的“记录”，必须用受控的方式来生成或访问，而如今它已一去不复返了。无可争辩的是，仍有这一类型的数据在那里，并将继续也应该通过关系型数据库来建模，存储和访问。但对于过去15年以来，随着Web的发展，电子商务和社交计算的兴起所引起的企业里不受控的非结构化并且面向信息的数据大爆炸，该如何应对呢？企业确实不需要关系型数据库来管理这些数据，因为关系型数据库的特点决定了它不适用于这些数据的性质和使用方式。

上图总结了现今以web为中心的企业中信息管理的新兴模式。而“非关系型数据库” 是处理这些趋势的最佳选择(较之关系型数据库来说)，提供了对非结构化数据的支持，拥有支持分区的水平伸缩性，支持高可用性等等。

以下是支持这一观点的一些实际应用场景：

日志挖掘——集群里的多个节点都会产生服务器日志、应用程序日志和用户活动日志等。对于解决生产环境中的问题，日志挖掘工具非常有用，它能访问跨服务器的日志记录，将它们关联起来并进行分析。使用“非关系型数据库”来定制这样的解决方案将会非常容易。

分析社交计算——许多企业如今都为用户(内部用户、客户和合作伙伴)提供通过消息论坛，博客等方式来进行社交计算的能力。挖掘这些非结构化的数据对于获得用户的喜好偏向以及进一步提升服务有着至关重要的作用。使用“非关系型数据库” 可以很好的解决这一需求。

外部数据feed聚合——许多情况下企业需要消费来自合作伙伴的数据。显然，就算经过了多轮的讨论和协商，企业对于来自合作伙伴的数据的格式仍然没有发言权。同时，许多情况下，基于合作伙伴业务的变更，这些数据格式也频繁的发生变化。通过“非关系型数据库”来开发或定制一个ETL解决方案能够非常成功的解决这一问题。

高容量的EAI系统——许多企业的EAI系统都有高容量传输流(不管是基于产品的还是定制开发的)。出于可靠性和审计的目的，这些通过EAI系统的消息流通常都需要持久化。对于这一场景，“非关系型数据库” 再次体现出它十分适用于底层的数据存储，只要能给定环境中源系统和目标系统的数据结构更改和所需的容量。

前端订单处理系统——随着电子商务的膨胀，通过不同渠道流经零售商、银行和保险供应商、娱乐服务供应商、物流供应商等等的订单、应用、服务请求的容量十分巨大。同时，由于不同渠道的所关联的行为模式的限制，每种情况下系统所使用的信息结构都有所差异，需要加上不同的规则类型。在此之上，绝大部分数据不需要即时的处理和后端对帐。所需要的是，当终端用户想要从任何地方推送这些数据时，这些请求都能够被捕获并且不会被打断。随后，通常会有一个对帐系统将其更新到真正的后端源系统并更新终端用户的订单状态。这又是一个可以应用“非关系型数据库”的场景，可用于初期存储终端用户的输入。这一场景是体现“非关系型数据库”的应用的极佳例子，它具有高容量，异构的输入数据类型和对帐的"最终一致性"等等特点。

企业内容管理服务——出于各种各样的目的，内容管理在企业内部得到了广泛的应用，横跨多个不同的功能部门比如销售、市场、零售和人力资源等。企业大多数时间所面临的挑战是用一个公共的内容管理服务平台，将不同部门的需求整合到一起，而它们的元数据是各不相同的。这又是“非关系型数据库”发挥作用的地方。

合并和收购——企业在合并与收购中面临巨大的挑战，因为他们需要将适应于相同功能的系统整合起来。“非关系型数据库” 可解决这一问题，不管是快速地组成一个临时的公共数据存储，或者是架构一个未来的数据存储来调和合并的公司之间现有公共应用程序的结构。

但我们如何才能准确的描述，相对于传统的关系型数据库，企业使用“非关系型数据库”带来的好处呢？下面是可通过非关系型数据库的核心特点(正如上一节所讨论的)而获得的一些主要的好处，即企业的任何IT决策都会参考的核心参数——成本削减，更好的周转时间和更优良的质量。

业务灵活性——更短的周转时间

“非关系型数据库”能够以两种基本的方式带来业务灵活性。

- 模式自由的逻辑数据模型有助于在为任何业务进行调整时带来更快的周转时间，把对现有应用和功能造成影响减到最少。在大多数情况下因任意的变更而给你带来的迁移工作几乎为零。
- 水平伸缩性能够在当越来越多的用户负载造成负载周期性变化，或者应用突然变更的使用模式时，提供坚固的保障。面向水平伸缩性的架构也是迈向基于SLA构建(例如云)的第一步，这样才能保证在不断变化的使用情形下业务的延续性。

更佳的终端用户体验——更优越的质量

在现今企业IT中，应用的质量主要由终端用户的满意度来决定。“非关系型数据库”通过解决如下终端用户的考虑因素，能够达到同样的效果，而这些因素也是最容易发生和最难以处理的。

- “非关系型数据库” 为提升应用的性能带来了极大的机会。分布式数据的核心概念是保证磁盘I/O(寻道速率)绝不能成为应用性能的瓶颈。尽管性能更多的是由传输速率来决定。在此之上，绝大部分解决方案支持各种不同的新一代的高速计算的范式，比如MapReduce，排序列，Bloom Filter，仅可追加的B树，Memtable等。
- 现今用户满意度的另一个重要的方面就是可靠性。终端用户希望在想要访问应用时就能访问到，并且至少是在当他们分配到时间的时候能随时执行他们的任务。所以不可用的应用需要不惜代价的避免。许多现代的“非关系型数据库”都能适应并支持这一类有着严格和最终一致性的可用性的需求。

更低的所有者总成本

在如今的竞争市场中，企业IT支出随时都要仔细审查，以合理的成本获取合理的质量才值得赞许。在这一领域中“非关系型数据库”在一定程度上胜于传统的数据库，特别是当存储和处理的数据容量很大时。

- 水平伸缩性的基本前提保证了它们可以运行于廉价机器之上。这不仅削减了硬件资本的成本，同时还削减了诸如电力，维护等运维成本。同时这还进一步的为利用诸如云、虚拟数据中心等下一代低成本的基础设施打下了基础。
- 从长期来看，更少的维护能带来更多的运维成本优势。对于关系型数据库，这绝对是一个需要存储大容量数据的场景。为大容量的数据调优数据库需要高超的技艺，也就意味着更高的成本。相较之下，“非关系型数据库”始终提供快速和响应的特点，就算是在数据大幅上升的情况下。索引和缓存也以同样的方式工作。开发者不必过多担心硬件、磁盘、重新索引及文件布局等，而是把更多的精力投入了应用程序的开发上。

企业采用中所面临的挑战

抛开所有这些长远的好处，在企业拥抱“非关系型数据库”之前，当然还需要经历各种各样的挑战。

不考虑因现有思想的转换和缺乏信心而产生的来自高层的阻力，目前我认为的最主要的战术性挑战是：

为“非关系型数据库”认定正确的应用/使用场景

尽管从理论上容易论证并非所有的企业数据都需要基于关系和ACID的系统，然而由于关系型数据库与企业数据间多年的绑定关系，要作出所有的数据可以通过非关系的解决方案而解耦的决定仍然有很多困难。许多时候IT经理(以及其它对于应用程序负有核心的底线责任的各级人员)不明白他们将会失去什么，这样的担忧对于从关系型数据库转变出来比较不利。企业IT最有价值的资产就是数据。因此，要作出决定使用一种不太明确或者未被广泛采用的解决方案来管理同样的数据，这种能力不仅需要转换思维方式，同时还需要来自高层的强大的支持(和推动)。

我们如何选择最适合我们的产品/解决方案

另一个重大的挑战是找出合适的产品/工具来提供“非关系型数据库”。正如前面所提到的那样，现今业界里面有多于25种不同的产品和解决方案，它们在四个方面有着不同的特点。正因为每个产品在这四个方面特点各异，所以要选择一个产品来应对所有的需求显得尤为困难。有的时候，可能在企业的不同部门使用到多种类型的非关系型数据库，最后人们可能会完全出于对标准的需要而转向关系型数据库。

如何获得规模经济

这一想法本质上是从前一个问题分支出来的。如果一个组织需要使用多个非关系型数据库解决方案(由于单个方案的适用问题)，那么保证在技术(开发者，管理者，支持人员)，基础设施(硬件成本，软件许可成本，支持成本，咨询成本)，以及工件(公共组件和服务)方面的规模经济就是一个大问题。这一方面与传统的关系型数据库解决方案比较起来确实更为严峻，因为大部分时间组织的数据存储都是以共享服务的模式在运行的。

我们如何保证解决方案的可移植性

从“非关系型数据库”的发展来看，我们可以很直观地推测在未来的几年中这一领域会有许多变化，比如供应商的合并，功能的进步以及标准化。所以对于企业来说一个更好的策略是不要把宝押在某个特定的产品/解决方案上，以后才可以更灵活的转换到一个更好的经过考验的产品。 由于现在的非关系型产品/解决方案大部分是私有的，因此IT决策者在考虑尝试“非关系型数据库”之前，不得不认真考虑可移植性这一重要的问题。这纯粹是出于保护现有投资的需要。

我们如何获得合适的产品支持类型

现在的“非关系型数据库”能通过外部组织而提供支持方案的少之又少。就算有，也无法与Oracle，IBM或者微软等相比。特别是在数据恢复，备份和特定的数据恢复方面，由于许多“非关系型数据库”在这些方面未能提供一个健壮而易于使用的机制，对于企业决策者来说，仍存在很大的问题。

我们如何预算整体成本

与重量级的关系型数据库相比，“非关系型数据库”通常在性能和伸缩性特征方面能提供的数据更少。我也没有发现有TPC基准程序方面和类似的其它方面的数据。这将企业决策者置于了一个“没有方向”的情况下，因为他们不知道需要在硬件、软件许可、基础设施管理和支持等方面支出多大的费用。要得出一个预算估计，缺乏判断的数据就成了一个主要的障碍。因此在项目启动阶段，大部分情况下决策者还是会选择基于熟悉的关系型数据库的解决方案。

有时候，就算可以得到这些数字，但也不足以用来形成TCO模型并与传统的基于关系型数据库的数据存储和非关系型数据存储进行整体的成本分析(Capex+Opex)比较。通常情况下水平伸缩性所要求的大量的硬件机器(以及软件许可成本，支持成本)，如果与垂直伸缩性乍一比较，会让人觉得战战兢兢，除非由此带来的好处经过基于TCO模型的全方位比较仍然被证明是可以持续的。

关于如何采用NoSQL的两点思考

这是否意味着目前来看企业应该对NoSQL运动持观望的态度呢？并非如此。诚然，“非关系型数据库”对于广泛的采用来说还未到完全成熟的阶段。但“非关系型数据库”作为未来企业骨架的潜力仍不能忽视。特别是不远的将来企业将更多地处理大容量的半结构化/非结构化以及最终一致性的数据，而不是相对而言小容量的，严格结构化的遵循ACID的数据。 所以现在而言至关重要的是做企业的关键决策人的思想工作，让他们明白企业的数据处理需要使用“非关系型数据库”。在这一过程中，要采取一些渐进的步骤把“非关系型数据库”应用到企业IT的一些关键的方面(技术，人员和流程)，并产生一定的价值。这样，就可以用一种缓慢而稳健的方式从整体上来解决我们之前所总结出来的一系列问题。

采用一个产品/解决方案

如今市场上的选择非常多样化，可根据“非关系型数据库”侧重的面不同而进行差异化的处理。与此同时，企业应用场景可能需要不同类型的特点。然而以不同的解决方案来处理不同的应用/使用场景从规模经济的角度出发对于企业是不适宜的。因此最好是根据目标应用的需要最终落实到某一个具体的产品/解决方案上。需要注意的是大多数的解决方案在特性上都会有一些折中，有些特性可能在其它的产品中可以获得，有些可能只是在发展路线图当中暂时设定了一个位置。因为大部分的产品会在不久的将来不断趋于成熟，因此可以通过不同配置来提供不同的解决方案。所以只要现有的解决方案能适合目前大部分的需要，不妨作为一个起点将其采纳。

选择产品/解决方案的经验法则

- 支持所需要的逻辑数据模型应当被给予更高的权重。这将从实质上决定该解决方案在当前或未来能否灵活地适应不同的业务需求。
- 调查该产品所支持的物理数据模型的合适与否，据此对这一解决方案所需要的水平伸缩性、可靠性、一致性和分区性作出合理的评估。这同样能表明备份和恢复机制的可能性。
- 接口支持需要与企业标准的运行环境对齐。由于这些产品支持多样的接口，所以这一点可以得到很好的处理。
- 只要产品支持水平伸缩性，对于持久化模型的选择就不再重要了。

这里有一份一系列“非关系型数据库”的对照表。对于现在正认真考虑采用的企业来说，这是一个不错的起点。为了更贴近企业本身的情况，从25+的集合中挑选出的子集所用到的的关键选择标准是：

1. 最重要的一点首先是企业应用程序必须支持有一定复杂程度的数据结构。否则的话，应用程序管理复杂性的责任将变得非常大。我认为比较合理的应当是介于纯粹的键值对与关系型模式中间的一种方案。出于这方面的考虑像Vlodemort，Tokyo Cabinet等产品就排除在了我的列表之外。
2. 第二点是以低成本的分片/分区为大容量数据提供水平支持。缺乏这样的支持就使得解决方案与任何关系型数据库无异了。因此像Neo4J(尽管他有丰富的基于图的模型)，Redis，CouchDB等此时此刻就被过滤出我的列表之外了。
3. 最后一条评判标准，在企业级推广之前我会考虑一定程度的商业支持。否则的话，一旦出现生产环境的问题，我该去找谁呢？出于这一点，我不得不将现在的一些明星产品排除在外，比如Cassandra(尽管有很大的可能不久的将来Rackspace或者Cloudera就会对其提供支持，因为它已经被用于一些生产环境里边了，比如Twitter，Digg，Facebook)。

有了这些过滤标准，我可以精简这一列表，符合目前企业可用的产品有MongoDB(下一版本就会提供shards支持)， Riak，Hypertable和HBase。下面这个表格中总结了这四个产品的主要特性。一个企业可以基于自己具体的实际情况从中作出选择，找到最适合自己需要的特性。

特性	MongoDB	Riak	HyperTable	HBase

逻辑数据模型	富文档，并提供对内嵌文档的支持	富文档	列家族(Column Family)	列家族(Column Family)
CAP支持	CA	AP	CA	CA
动态添加删除节点	支持(很快在下一发布中就会加入)	支持	支持	支持
多DC支持	支持	不支持	支持	支持
接口	多种特定语言API(Java，Python，Perl，C#等)	HTTP之上的JSON	REST，Thrift，Java	C++，Thrift
持久化模型	磁盘	磁盘	内存加磁盘(可调的)	内存加磁盘(可调的)
相对性能	更优(C++编写)	最优(Erlang编写)	更优(C++编写)	优(Java编写)
商业支持	10gen.com	Basho Technologies	Hypertable Inc	Cloudera

数据访问抽象

为数据访问创建一个单独的抽象层对于“非关系型数据库”来说是必须的。它可以带来多方面的好处。首先，应用开发者可以与底层解决方案的细节完全隔离开来。这对于技术方面的伸缩性带来了好处。同时未来如果需要更改底层的解决方案也很方便。这也以一个标准的方式满足了多个应用的要求(即去掉了Join，Group by等复杂特性的SQL)。

为性能和伸缩性创建模型

不管选择怎样的解决方案，使用标准技术(比如排队网络模型，分层排队网络等)来对性能和伸缩性进行建模都是高度推荐的。它能够为基本的服务器规划、拓扑以及整体的软件许可证成本，管理运行等提供必要的数据。这将实质上成为所有预算计划的主要参考数据，并对作出决策提供帮助。

构建显式的冗余

要防止数据丢失，除了将数据复制到备份服务器上，没有其它的办法了。尽管许多非关系型数据库提供自动复制功能，但仍然存在主节点单点失效的风险。因此最好是使用次节点备份，并准备好用于数据恢复和自动数据修复的脚本。出于这样的目的，应当充分的了解目标解决方案的物理数据模型，找出可能的恢复机制备选方案，基于企业的整体需求和实践来对这些选项作出评估。

构建公共数据服务平台

就像公共共享服务的关系型数据库一样，也可以构建非关系型数据库的公共数据服务来促进规模经济效应，满足基础设施和支持的需要。这对于未来进一步演化和更改也有帮助。这可以作为愿望列表上的最终目标，通过中期或长期的努力来达到这一成熟水平。然而，初始阶段就设立这样的远景有助于在整个过程中作出正确的决策。

壮大企业的技术力量

每个组织都有一部分人对于学习新生的和非传统的事物充满热忱。成立这样的小组，并挑选人员(全职的或兼职的)，密切关注这方面的动向，了解问题和挑战，进行前瞻性的思考，能够为使用这些技术的项目提供方向和帮助。同时，这个小组还可以为决策者澄清炒作的疑云，提供来自真实数据的观点。

建立与产品社区的关系

选择了产品之后，与产品社区建立起良好的关系对于双方的成功都有极大的好处。许多非关系型数据库目前都有十分活跃的社区，非常愿意相互帮助。企业与社区之间的良好合作能给大家带来一个双赢的局面。如能提前对问题和解决方案有了解，那么企业在对某些特性或版本作出决策时就能成竹在胸。反过来，企业又能对产品特性的路线图产生影响作用，这对他们自身和社区都是有利的。另一方面，社区也能从实际层次的问题中得到反馈，从而丰富和完善产品。来自大型企业的成功案例同样能让他们处于领先。

迭代前进

考虑到非关系型数据库相对的成熟度，风险最小的采用策略就是遵循迭代开发的方法论。构建公共数据服务平台和标准化数据访问抽象不可能是一蹴而就的。相反，通过迭代和面向重构的方式能更好的达到目标。运用不太成熟的技术进行转型的过程，中途改变解决方案也不会太意外的。与此同时，采用敏捷的方式来看待事物，能够帮助建立起一个能从管理和实现两方面不断吸引改进的开放态度。

然而，在这一问题上实现迭代，非常重要的一点是定义一个决策条件矩阵。比如操作指南(和例子)，来判断一个应用的对象模型是否适合关系型或非关系的范围，对基础设施规划作出指导，列出必需的测试用例等等。

结束语

企业的非关系型数据库采用过程中最大的挑战就是转变决策者的思想观念——让他们相信并非所有的数据/对象都适合关系型数据库。 最能证明这一点就是选择合适的用例去**尝试**非关系型数据库，进而证实在合适的背景下，非关系型数据库是比关系型数据库更有效的解决方案。找到一些“非关键业务”(但能立竿见影的)适合于非关系型数据库的项目。这些项目的成功(甚至失败)都能有助于观念的改变。这也能有助于不断学习如何才能以一种不同的方式来更好的采用非关系型数据库。这些少儿学步般的**尝试**所作出的努力与投入都是值得的，如果企业想要在将来使用“非关系型数据库”来重塑其信息管理体系的话。

关于作者

Sourav Mazumder目前是InfoSys Technologies的首席技术架构师。他在信息技术领域有14年以上的经验。作为Infosys技术顾问团的主要成员，Sourav为Infosys在美国、欧洲、澳洲和日本的主要客户，提供保险、电信、银行、零售、安全、交通以及建筑、工程、施工等多个行业的服务。他曾参与Web项目的技术架构和路线图定义，SOA战略实施，国际战略定义，UI组件化，性能建模，伸缩性分析，非结构化数据管理等等。Sourav参考的Infosys自身的核心银行产品Finacle，也为他提供了丰富的产品开发经验。Sourav还曾参与开发Infosys的J2EE可重用框架，和定义Infosys在架构方面和开发定制应用方面的软件工程方法。Sourav的经历还包括在保证架构合规和开发项目的治理方面的工作。

Sourav是iCMG认证的软件架构师，同时也是TOGAF 8认证的执行者。Sourav最近在LISA伯克利全球化会议上发表了演讲。Sourav关于SOA的最新白皮书在社区里十分流行。

Sourav目前关注NoSQL，Web2.0，治理，性能建构和全球化。

SQL vs NoSQL：数据库并发写入性能比拼

最近听说了很多关于NoSQL的新闻，比如之前Sourceforge改用MongoDB，Digg改用Cassandra等等。再加上之前做数据库比较时有人推荐我mongodb，所以也搜索了一下NoSQL，觉得NoSQL可能真的是未来的趋势。

NoSQL vs SQL

传统SQL数据库为了实现ACID(atomicity,consistency, isolation, durability)，往往需要频繁应用文件锁，这使得其在现代的web2.0应用中越来越捉襟见肘。现在SNS网站每一个点击都是一条/多条查询，对数据库写的并发要求非常之高，而传统数据库无法很好地应对这种需求。而仔细想来SNS中大部分需求并不要求ACID，比如Like/Unlike投票等等。

NoSQL吸取了教训，比如有些NoSQL采用了eventually consistency的概念，在没有Update操作一段时间后，数据库将最终是consistency的，显然这样的数据库将能更好的支持高并发读写。

SQL数据库是基于schema的，这对时时刻刻更新着的web2.0应用开发者来说是个噩梦：随时随地有新的应用出现，旧的数据库无法适应新的应用，只能不停地更新schema，或者做补丁表，如此一来要么schema越发混乱，要么就是数据库频繁升级而耗时耗力耗钱。

NoSQL一般就没有schema这种概念，大部分NoSQL都直接保存json类的Row，比如一个记录可以是{ id = 1, name = Bob, phone = 38492839 }，这样扩展升级非常方便，比如需要地址信息直接加入 address=blahblah 即可。

传统SQL很难进行分布式应用，即使可以也往往代价高昂。而NoSQL则很好地解决了这个问题：他们一般都直接从分布式系统中吸取了Map/Reduce方法，从而很容易就可以处理规模急速增加的问题。

推荐robbin牛的NoSQL数据库探讨之一 — 为什么要用非关系数据库？一文，介绍了主流的一些NoSQL系统，还有这个站http://nosql-database.org/收集了基本上目前所有的NoSQL系统。

总结一下我对NoSQL的看法，NoSQL出现的目的就是为了解决高并发读写的问题，而高并发应用往往需要分布式的数据库来实现高性能和高可靠性，所以NoSQL的关键字就是concurrency和scalability。

我的瓶颈

我之前主要关注数据库的select性能也就是read性能，在读性能方面SQL数据库并没有明显的劣势，应该说纯粹高并发读的性能的话往往要优于NoSQL数据库，然而一旦涉及写，事情就不一样了。

我本来以为自己不会遇到大量写的问题，后来发现即使在simplecd这种简单的应用环境下也会产生大量的并发写：这就是爬VC用户评论的时候。事实上，sqlite3在处理这个问题上非常的力不从心，所以我产生了换个数据库的想法。

既然我是要求能高并发读写，干脆就不用SQL了，但是同时我也想测试一下其他SQL的写性能。

我的数据有180万条，总共350M，测试用了10个线程，每个线程做若干次100个数据的bulk写入，然后记录总共耗时。结果如下:

innodb: 15.19
myiasm: 14.34
pgsql: 23.41
sqlite3: 锁住了
sqlite3(单线程): 300+
mongodb: 3.82
couchdb: 90
couchdb(单线程）:66

作为一个MySQL黑，看到这组测试数据我表示压力很大。在SQL数据库中，mysql意外地取得了最佳的成绩，好于pgsql，远好于sqlite。更令人意外的是myisam居然优于号称insert比较快的innodb。不管如何，对我的应用来说，用mysql保存评论数据是一个更为明智的选择。我对mysql彻底改观了，我宣布我是mysql半黑。以后select-intensive的应用我还是会选择sqlite，但是insert/update-intensive的应用我就会改用mysql了。

MongoDB和CouchDB同为NoSQL，表现却截然相反，MongoDB性能很高，CouchDB的并发性能我只能ORZ，这种性能实在太抱歉了。

NoSQL的碎碎念

其实我本来还打算测试cassandra的，可是cassandra用的是java，这首先让我眉头一皱，内存大户我养不起啊，其次看了cassandra的文档，立刻崩溃，这简直就是没有文档么。（BTW，CouchDB也好不到哪里去，我都是用python-couchdb然后help(couchdb.client)看用法的）

至于CouchDB，可能是因为采用http方式发送请求，所以并发性能糟糕的一塌糊涂，很怀疑它是否有存在的理由。

MongoDB是我用下来最讨人喜欢的一个NoSQL。不但文档丰富，使用简单，性能也非常好，它的Map/Reduce查询(很多NoSQL都有)让我惊叹，数据库可以非常简单地就扩大规模，完全不用理会什么分区分表之类繁琐的问题，可惜这方面我暂时没有需求。但是MongoDB有两大致命问题。

第一是删除锁定问题，当批量删除记录时，数据库还是会锁定不让读写。这意味着进行数据清理时会让网站应用失去响应。见locking problems

第二是内存占用问题，MongoDB用了操作系统的内存文件映射，这导致操作系统会把所有空闲内存都分配给MongoDB，当MongoDB有这个需要时。更可怕的是，MongoDB从来不主动释放已经霸占的内存，它只会滚雪球一样越滚越大，除非重启数据库。这样的上下文环境下，MongoDB只适合一台主机就一个数据库，而没有其他应用的环境，否则一会儿功夫MongoDB就会吃光内存，然后你都fork不出新进程，彻底悲剧。见memory limit

总之NoSQL虽然让我眼前一亮，可是目前尝试的一些产品都让人望而生畏，现在的NoSQL都把目光放在了巨型网站上，而没有一个小型的，可以在VPS里面应用的高性能NoSQL，令我有点失望。NoSQL尚未成熟，很期待它的将来发展，目前来说MySQL还是更好的选择。

谷歌Jeff Dean阐述分布式系统设计模式
分布式系统设计模式

1. 系统失败是很平常的事情：每年有1-5%的硬盘会报废，服务器每年会平均宕机两次，报废几率在2-4%几率。

2. 将一个大而复杂系统切分为多个服务：而且服务之间依赖尽可能的少，这样有助于测试，部署和小团队独立开发。例子：一个google的搜索会依赖100多个服务。吴注：需要一套机制来确保服务的fault-tolerant，不能让一个服务的成败影响全局。

3. 需要有ProtocolDescription Language：比如protocol buffers。吴注：这样能降低通信方面的代码量。

4. 有能力在开发之前，根据系统的设计来预测性能：在最下面有一些重要的数字。

5.在设计系统方面，不要想做的很全面，而是需要抓住重点。

6. 为了增量做设计，但不为无限做设计。比如：要为5-50倍的增量做设计，但超过1000倍了，就需要重写和重新设计了。

7. 使用备份请求来降低延迟：比如一个处理需要涉及1000台机器，通过备份请求这个机制来避免这个处理被一台慢机器延误。吴注：这个机制非常适合MapReduce。

8. 使用范围来分布数据，而不是Hash：因为这样在语义上比较简单，并且容易控制。吴注：在大多数情况下语义比性能更重要，不要为了20%的情况hardcode。

9. 灵活的系统，根据需求来伸缩：并且当需求上来的时候，关闭部分特性，比如：关闭拼写检查。

10. 一个接口，多个实现。

11. 加入足够的观察和调式钩子（hook）。

12. 1000台服务器只需单一Master：通过Master节点来统一指挥全部的行动，但客户端和Master节点的交互很少，以免Master节点Crash，优点是，在语义上面非常清晰，但伸缩性不是非常强，一般最多只能支持上千个节点。

13. 在一台机器上运行多个单位的服务：当一台机器宕机时，能缩短相应的恢复时间，并且支持细粒度的负载均衡，比如在BigTable中，一个Tablet服务器会运行多个Tablet。

ZooKeeper—分布式协同服务

1.概述

ZooKeeper[1]是hadoop的一个分布式协同服务，主要解决分布式应用程序中的局部失败问题，即网络操作过程中发送者与接收者之间无法明确发送操作是否正确无误。在分布式系统中，它能够提供:系统配置信息维护，命名，分布式同步等服务。著名的Hadoop分布式数据库HBase已经采用了ZooKeeper技术为其提供服务[2]，如，ZK存储了HBase中的Region的寻址入口；实时监控Rgeion Server的状态，将Region Server上、下线实时通知给master。

Zookeeper非常简化暴漏一些简单的基本操作，可以将其想象为一个简单的文件系统提供读、写操作服务，此外，它还有预定(ordering)和通知(notification)服务。ZK的可靠性体现在ZK服务是一一个集群的方式提供服务，所以，分布式应用程序不会因为使用ZK服务器而导致单点失败问题，相反，很多分布式应用程序可以通过引入ZK来解决本身系统存在的单点问题，HBase就是这样做的。

2.ZooKeeper安装

1.下载http://www.apache.org/dyn/closer.cgi/zookeeper/最新文档版本。

2.解压：% **tar xzf zookeeper-*x.y.z*.tar.gz** 。配置环境变量：ZOOKEEPER_INSTALL指向ZK的安装目录，在PATH中加入可执行文件目录。简单的办法是在/etc/profile里加入下面两行：

export ZOOKEEPER_INSTALL=/home/tom/zookeeper-x.y.z

export PATH=$PATH:$ZOOKEEPER_INSTALL/bin

ZK的配置文件conf/zoo.cfg三个变量：

trickTime= 2000 //ZK的基本时间单位，单位微秒

dataDir=/usr/tom/zookeeper //数据目录，用于存放ZK的持久化信息

clientPort=2181 //ZK的默认监听端口

3.启动ZK：

%zkServer.sh start

4.确认安装：

%echo ruok | nc localhost 2181

如果返回信息是imok(“Iam ok ”)的话表明安装成功。

以上只是简单的单机情况的配置。关于在实际生成系统中的安装配置，后面将详细介绍。

分布式数据库的具体实现与对比分析

1．前言

随着传统的数据库、计算机网络和数字通信技术的快速发展，以数据分布存储和分布处理为主要特征的分布式数据库系统的研究和开发越来越受到人们的关注。如何在一个数据库系统中实现一个分布式数据库，在实现分布是数据库中采用何种策略以及有那些需要注意的问题，这一直是数据库研究和应用相关领域人员非常关心的问题。本文就在Microsoft SQL系列数据库系统中分布式数据的具体实现进行了阐述，并对相关问题进行深入的分析。

2．分布式数据库简介

分布式数据库系统是在集中式数据库系统的基础上发展起来的，由分布式数据库管理系统和分布式数据库组成，是数据库技术与计算机网络技术的产物。分布式数据库管理系统是具有管理分布数据库功能的计算机系统，分布式数据库则是一组逻辑上属同一系统,但物理上分布在计算机网络的不同结点的结构化数据的集合，由分布于计算机网络上的多个逻辑相关的数据库组成。网络中的每个结点（场地）具有独立处理的能力（称为本地自治），可执行局部应用，同时，每个结点通过网络通讯系统也能执行全局应用。所谓局部应用即仅对本结点的数据库执行某些应用。所谓全局应用（或分布应用）是指对两个以上结点的数据库执行某些应用。支持全局应用的系统才能称为分布式数据库系统。对用户来说，一个分布式数据库系统逻辑上看如同集中式数据库系统一样，用户可在任何一个场地执行全局应用。分布式数据库系统的特点是：

（1）物理分布性：分布式数据库系统中的数据不是存储在一个站点上，而是分散存储在由计算机网络联结起来的多个站点上。

（2）逻辑整体性：分布式数据库系统中的数据物理上是分散在各个站点中，但这些分散的数据逻辑上却是一个整体，它们被分布式数据库系统的所有用户（全局用户）共享，并由一个分布式数据库管理系统统一管理。

（3）站点自治性：站点自治性也称场地自治性，每个站点上的数据由本地的DBMS 管理，具有自治处理能力，完成本站点的局部应用。

分布式数据库系统适合于单位分散的部门，系统的结点可反映公司的逻辑组织，允许各部门将其常用数据存贮在本地，实施就地存放就地使用，降低通讯费用，并可提高响应速度。分布式数据库可将数据分布在多个结点上，增加适当的冗余，可提高系统的可靠性，只要一个数据库和网络可用，那么全局数据库可一部分可用。不会因一个数据库的故障而停止全部操作或引起性能瓶颈。故障恢复通常在单个结点上进行。结点可独立地升级软件。每个局部数据库存在一个数据字典。由于分布式数据库系统结构的特点，它和集中式数据库系统相比具有以下优点：

1) 可靠性高，个别场地发生故障，不致引起整个系统的瘫痪

2) 可扩展性，为扩展系统的处理能力提供了较好的途径。

3) 均衡负载，可避免临界瓶颈

目前，随着计算机体系结构和数据库技术的发展，分布式数据库系统技术已经有了长足发展。基于关系数据模型的分布式数据库技术在商业处理的应用方面已非常成功，如Oracle、MS SQL SERVER、Sybase、IBM DB2等数据库平台，其分布式处理技术能很好地满足大型数据库管理的需要，并能实现一定的分布式实时分析处理和数据更新，能够满足各种不同类型用户对不同数据库功能的要求。但不同的数据库产品在价格、性能、稳定性等方面有着不小的差异，在对分布式数据库理论的支持程度、分布式实现的具体方式也都有各自的特点，用户应当根据各自的实际情况选用合适的数据库产品。

3．分布式数据库的具体实现

3．1实现的理论准备

3．1．1实现流程

由于以上集中式数据库所不能替代的特点，分布式数据库的使用已经越来越成为当前数据库使用的主流。同时如何根据实际情况实现一个架构可靠、运行稳定的分布式数据库也成为许多数据库使用者所面临的问题。综合来讲，设计并实现一个分布式数据库主要有以下几个步骤：

1) 掌握分布式数据库的基本原理。

2) 根据需求和实际情况选取一种合适的分布式数据库系统。

3) 了解在该数据库系统中分布式数据库的实现方式。

4) 在系统中的具体实现。

本文已经就以上几点分别做了简单阐述。下面将集中探讨一下在MS SQL 2000数据库系统中实现分布式数据库的方式。

3．1．2 MS SQL 2000数据库简介

MS SQL 2000数据库系统是微软公司的主流数据库产品，其工作效率、稳定性等指标都非常出色，且在具有友好、简单的操作方式的同时，对分布式数据库的实现有着完整的理论支持。且对不同类型的分布式数据库应用需求都提出了完善的解决方案。所以对MS SQL 2000数据库中分布式数据库实现的掌握对分布式数据库的实际设计以及在其它数据库平台上的实现都有很大的参考作

3．1．3"复制"技术与分布式事务处理

设计一个分布式计算解决方案首先需要考虑的问题就是应用的完整性、复杂性、性能和可用性以及响应时间等，同时还需要考虑的是对于不同的应用需求是采用实时存取远程数据（分布式事务处理）还是采用延迟存取远程数据（"复制"）。

MS SQL 2000数据库中的分布式事务处理即通过事务处理机制采用实时数据存取，能够保证数据的一致性。是一种实时远程存取和实时更新数据的技术。这种技术可以保证应用的完整性并降低了应用的复杂性，但是如果系统存在网络存取速度很慢这样的问题，相应响应时间就会很慢。这是一种同步分发数据库技术。

"复制"，顾名思义就是将数据库中的数据拷贝到不同物理地点的数据库中以支持分布式应用，它是整个分布式计算解决方案的一个重要组成部分。这里针对"复制"也存在同步"复制"和异步"复制"的问题。同步"复制"，复制数据在任何时间在任何复制节点均保持一致。如果复制环境中的任何一个节点的复制数据发生了更新操作，这种变化会立刻反映到其他所有的复制节点。这种技术适用于那些对于实时性要求较高的商业应用中。异步"复制"，所有复制节点的数据在一定时间内是不同步的。如果复制环境中的其中的一个节点的复制数据发生了更新操作，这种改变将在不同的事务中被传播和应用到其他所有复制节点。这些不同的事务间可以间隔几秒，几分种，几小时，也可以是几天之后。复制节点之间的数据是不同步的，但传播最终将保证所有复制节点间的数据一致。这是一种延迟远程存取和延迟传播对数据更新的技术，有很高的可用性和很短的响应时间，相比同步分发数据库技术就显得复杂一些，为了确保应用的完整性需要仔细考虑和设计。

对于实际的商业问题，必须权衡这两种技术的利弊最终选择最佳的解决方案，有些问题选用分布式事务处理比较适合，也有一些问题采用"复制"是比较好的解决方案，还有一些问题必须综合这两种技术。

3．1．4 "发行－分布－订阅"结构

MS SQL 2000数据库系统中分布式数据库的具体实现采用"发行－分布－订阅"结构。具体来讲就是将实现分布式数据的角色分为三种，即发行者、分布者、订阅者。这三中角色在数据的分布中其着不同的作用：

1) 发行者：是发行项目的集合。发行项目也就是数据库中表、存储过程或特定的行或列等我们要作为分布式数据的这部分内容。

2) 订阅者：从发行者上进行数据的订阅，即按照需求使用发行者上的数据更新自己本地的数据。其订阅方式包括推出式订阅（属于主动式发布，发行者主动将数据传送到订阅者处）、引进式订阅（属于被动式发布，即当订阅者需要更新数据时再向发行者请求进行更新）

3) 发布者：负责管理，将发行者上发行的数据安计划传送到订阅者处。

通过采用"发行－分布－订阅"结构，分布式数据库中每一个角色的功能更加明确。同时应该注意的是，三种角色指的是再分布式数据实现中所起到的作用，并不是指具体的计算机。同一台计算机可能扮演多种角色，即一台主机在分布式数据库中有可能即使发行者，也是订阅者，或者三种都是。

3．2 "复制"策略的选择

MS SQL 2000数据库采用的"复制"方式有快照复制（Snapshot Replication）、事务复制（transaction replication）和合并复制(merge replication)三种，三者都是按照实现制定的"复制"策略，在一定的时间，按照一定的规则进行一定数量发布内容的复制，三者的具体差别及特点如下：

1) 快照复制：在复制时将所有发布数据全部重新传送一遍。此方法的特点是具有周期性、进行批量复制处理、高延迟、高度自治。但由于一次传送数据较多，灵活性差，故不适宜大型数据库。但同时此方法不需持续监控，故代价较低。

2) 事务复制：属于增量复制，即在复制时仅复制增减或变更的内容，特点是低延迟、低度自治，适宜大型数据库。但此方法需要持续监控，故代价较高。另外，此方法复制的方式也可设置为即时更新。

3) 合并复制：复制时即根据发行者上的数据，也根据订阅者上来进行更新。此方法高度自治，在过程中使用使用触发器，但由于其自身的实现方式，故不能保证数据一致性

不同的"复制"策略适用于不同的分布式数据库实现要求，我们可以根据延迟、站点自治、事务一致性、数据更新冲突、"复制"发生的频率需求和网络特性的各方面的实际情况来决定使用那一种或那几种"复制"策略。

3．3 "复制"的实现

在决定好"复制"的策略后，要在系统中进行具体的实现。其主要步骤包括：

3．3．1调整、设定发行者和分布者

通过对预发布的数据的了解和对接受数据者的判断等，我们已经确定了"复制"的策略，这一步我们就需要确定"复制"发生的频率需求并根据网络特性，如拓扑结构、"复制"类型的影响、空间需求等对发行者的发行内容和分布者的分布策略进行具体的设置。

3．3．2设置订阅者

通过分布式数据的要求和订阅者特征决定订阅者的属性，如选择订阅方式是"推出式订阅"还是"引进式订阅"等，此属性为最重要的内容。然后进行其他订阅服务属性设置。

3．3．3 设定代理服务

分布式数据库中的代理服务是进行"复制"管理的核心，它是一种始终运行的服务，负责全部"复制"任务的控制和管理。这些服务中主要包括：快照代理、分布代理、日志代理和合并代理等。每一种"复制"方法都需要一种或集中代理服务的配合。

通过以上几部分内容的设置，我们就可以建立并运行起一个完整、可行的分布式数据库。但经管这个数据库能够正确的实现并运行，并不一定意味者其"复制"机制能够始终保持稳定并一直符合我们的要求，所以，我们还需要对"复制"服务进行有效的管理。这就包括使用服务器的"复制"服务监视工具来对"复制"服务的维护，并在"复制"服务发生故障时进行的问题解决。

4．MS SQL 2000数据库对异类数据库"复制"的支持

MS SQL 2000数据库可以与其他异类数据库实现数据直接"复制"，包括Microsoft Access数据库、Oracle系列数据库、IBM DB2数据库以及其他支持 SQL Server ODBC接口标准的数据库。这些异类数据库可以通过"复制"代理直接连接起来，组成一个大的分布式数据库，自由进行分布式数据的"复制"处理。

5．小结

分布式数据库以其高度的可扩展性和可伸缩性，同时由于资源共享提高了系统的性价比，已经得到广泛的研究和应用。本文就在MS SQL 2000数据库中实现分布式数据库的策略与具体方式做了详细的分析。MS SQL 2000数据库是一款常用的数据库产品，其分布式数据功能针对各种实际情况都有着完善的解决方案。通过对其分布式数据库实现策略、方法和特点的学习，可以加深我们对分布式数据库理论的认识，加强我们通过具体途径，解决实际问题的能力。

Hadoop：Google分布式存储/计算/查询系统的开源实现

Google的伟大很大程度上得益于其强大的数据存储和计算能力，GFS和Bigtable使得其基本摆脱了昂贵的人力运维，并节省了机器资源；MapReduce使其可以很快看到各种搜索策略试验的效果。鉴于此，国内外出现了很多的模仿者，它们都是所谓的“高科技”企业，且往往还打上“云计算”的标签。从头到尾实现一套Google的存储/计算/查询系统是极其复杂的，也只有寥寥无几的几个巨头可以做到，Hadoop做为一种开源的简化实现，帮了很多科技公司的大忙。前些时候，Yahoo将Hadoop的创始人收于麾下，使得Hadoop完成华丽大转身，性能实现了一个飞跃式提升。

Hadoop主要包括HDFS(分布式文件系统，对应GFS)，MapReduce(分布式计算系统)和HBase(分布式查询系统，对应Bigtable)，其中以HDFS和MapReduce较为成熟。另外，Hadoop还包括一些辅助系统，如分布式锁服务ZooKeeper，对应GoogleChubby。这一套系统的设计目标如下：

1. 简化运维：在大规模集群中，机器宕机，网络异常，磁盘错都属于正常现象，因此错误检查，自动恢复是核心架构目标。Google的解决方案就已经做到了机器随时加入/离开集群。

2. 高吞吐量：高吞吐量和低延迟是两个矛盾的目标，Hadoop优先追求高吞吐量，设计和实现中采用了小操作合并，基于操作日志的更新等提高吞吐量的技术。

3. 节省机器成本：Hadoop鼓励部署时利用大容量的廉价机器(性价比高但是机器故障概率大)，数据的存储和服务也分为HDFS和HBase两个层次，从而最大限制地利用机器资源。

4. 采用单Master的设计：单Master的设计极大地简化了系统的设计和实现，由此带来了机器规模限制和单点失效问题。对于机器规模问题，由于Hadoop是数据/计算密集型系统，而不是元数据密集型系统，单Master设计的单个集群可以支持成千上万台机器，对于现在的几乎所有应用都不成问题；而单点失效问题可以通过分布式锁服务或其它机制有效地解决。

Google的其它模仿者包括，Microsoftdyrad(模范GoogleMapReduce)，Hypertable(HadoopHBase开发团队核心成员开始的一个开源项目，C++实现的Bigtable)。Google的解决方案不是万能的，然而相对我们普通人已经是几乎不可逾越了。Hadoop做为Google的这个模型的简化实现，有很多不足，这里先列出几点，以后将通过阅读Hadoop源代码和论文逐渐展开分析。Hadoop的几个明显缺点如下：

1. 采用Java实现。Java的IO处理虽然没有性能瓶颈，但是对于CPU密集型的任务是一个噩耗。这点可以通过对比HBase和Hypertable两个开源的Bigtable实现来做初步的验证。

2. 开源项目。开源本身是一柄双刃剑，它方便了大多数人，但是对于一个有一定规模的公司，项目发展方向的把握，技术保密，技术支持等都是采用Hadoop这种开源项目必须考虑的问题。另外，Hadoop作为一个比较新的项目，性能和稳定性的提升还需要一定时间。

Moosefs介绍

MooseFS是一种分布式文件系统，MooseFS文件系统结构包括以下四种角色：

1 管理服务器managingserver (master)

2 元数据日志服务器Metaloggerserver（Metalogger）

3 数据存储服务器data servers(chunkservers)

4 客户机挂载使用clientcomputers

各种角色作用:

1 管理服务器:负责各个数据存储服务器的管理,文件读写调度,文件空间回收以及恢复.多节点拷贝

2 元数据日志服务器: 负责备份master服务器的变化日志文件，文件类型为changelog_ml.*.mfs，以便于在master server出问题的时候接替其进行工作

3 数据存储服务器:负责连接管理服务器,听从管理服务器调度,提供存储空间，并为客户提供数据传输.

4 客户端: 通过fuse内核接口挂接远程管理服务器上所管理的数据存储服务器,.看起来共享的文件系统和本地unix文件系统使用一样的效果.吧。

1 Moosefs简介

1.1 角色构成

整个mfs共计四种角色：master、metalogger、chunk和client

1、master：只有一台。

2、metalogger：可以有多台。它负责定期从master下载metadata，并实时同步changelog。metadata和changelog的关系类似于sfrd里面基准和增量的关系。当master挂了的时候，metalogger利用下载下来的metadata和实时同步的changelog来恢复master挂掉时候的metadata。并且接管master的功能。

3、chunk：提供存储的服务器，可以有多台。这些服务器负责提供存储，它可以自由的启动和停止。在chunk启动后，会主动与master联系，master知道有多少chunk在网络中，并且会定期检查chunk的状态。

4、client：使用mfs的服务器，可以有多台。它需要运行mfsmount命令，将网络上的存储挂载到本地，看起来类似nfs。client就像读写本地磁盘那样读写mfsmount挂载的网络存储。

解读：基于Hadoop的大规模数据处理系统

2010年8月27日下午，由中国电子学会云计算专家委员会主办、CSDN承办的“高端云计算课程”在中关村软件园进行了免费的公开课程，公开课上座无虚席，原定几十人的教室，最终挤满了上百人。本次高端课程以“普及云计算，利用云计算，发展云计算”为基本原则，旨在为云计算领域培养和选拔更多更优秀的技术和管理人才奠定坚实的基础。

在本次公开课上，中科院计算所副研究员查礼博士做了主题演讲，解密了基于Hadoop的大规模数据处理系统的组成及原理。

Hadoop的组成部分

Hadoop是Google的MapReduce一个Java实现。MapReduce是一种简化的分布式编程模式，让程序自动分布到一个由普通机器组成的超大集群上并发执行。

Hadoop主要由HDFS、MapReduce和HBase等组成。具体的组成如下图：

Hadoop的组成图

1. Hadoop HDFS是Google GFS存储系统的开源实现，主要应用场景是作为并行计算环境（MapReduce）的基础组件，同时也是BigTable（如HBase、HyperTable）的底层分布式文件系统。HDFS采用master/slave架构。一个HDFS集群是有由一个Namenode和一定数目的Datanode组成。Namenode是一个中心服务器，负责管理文件系统的namespace和客户端对文件的访问。Datanode在集群中一般是一个节点一个，负责管理节点上它们附带的存储。在内部，一个文件其实分成一个或多个block，这些block存储在Datanode集合里。Namenode执行文件系统的namespace操作，例如打开、关闭、重命名文件和目录，同时决定block到具体Datanode节点的映射。Datanode在Namenode的指挥下进行block的创建、删除和复制。Namenode和Datanode都是设计成可以跑在普通的廉价的运行Linux的机器上。HDFS采用Java语言开发，因此可以部署在很大范围的机器上。一个典型的部署场景是一台机器跑一个单独的Namenode节点，集群中的其他机器各跑一个Datanode实例。这个架构并不排除一台机器上跑多个Datanode，不过这比较少见。

HDFS体系结构图

2. Hadoop MapReduce是一个使用简易的软件框架，基于它写出来的应用程序能够运行在由上千个商用机器组成的大型集群上，并以一种可靠容错的方式并行处理上TB级别的数据集。

一个MapReduce作业（job）通常会把输入的数据集切分为若干独立的数据块，由 Map任务（task）以完全并行的方式处理它们。框架会对Map的输出先进行排序，然后把结果输入给Reduce任务。通常作业的输入和输出都会被存储在文件系统中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

Hadoop MapReduce处理流程图

3. Hive是基于Hadoop的一个数据仓库工具，处理能力强而且成本低廉。

主要特点：

存储方式是将结构化的数据文件映射为一张数据库表。

提供类SQL语言，实现完整的SQL查询功能。

1.可以将SQL语句转换为MapReduce任务运行，十分适合数据仓库的统计分析。

不足之处：

1.采用行存储的方式（SequenceFile）来存储和读取数据。

2.效率低：当要读取数据表某一列数据时需要先取出所有数据然后再提取出某一列的数据，效率很低。

3.占用较多的磁盘空间。

由于以上的不足，查礼博士介绍了一种将分布式数据处理系统中以记录为单位的存储结构变为以列为单位的存储结构，进而减少磁盘访问数量，提高查询处理性能。这样，由于相同属性值具有相同数据类型和相近的数据特性，以属性值为单位进行压缩存储的压缩比更高，能节省更多的存储空间。

行列存储的比较图

HBase是一个分布式的、面向列的开源数据库，它不同于一般的关系数据库,是一个适合于非结构化数据存储的数据库。另一个不同的是HBase基于列的而不是基于行的模式。HBase使用和 BigTable非常相同的数据模型。用户存储数据行在一个表里。一个数据行拥有一个可选择的键和任意数量的列，一个或多个列组成一个ColumnFamily，一个Fmaily下的列位于一个HFile中，易于缓存数据。表是疏松的存储的，因此用户可以给行定义各种不同的列。在HBase中数据按主键排序，同时表按主键划分为多个HRegion。

HBase数据表结构图

DRDA 分布式关系数据库体系结构

DRDA（Distributed Relational Database Architecture ）分布式关系数据库体系结构
分布式关系数据库体系结构(DRDA)是一个跨IBM平台访问、遵循SQL标准的数据库信息的IBM标准。它是IBM的信息仓库框架中的重要组成部分，该框架定义了庞大的后台服务器，客户机可通过较小的基于工作组的中介服务器来访问它。DRDA具有下列功能：
定义了客户机和后台数据库之间的接口协议。
提供了IBM的DB2、DBM、SQL/DS和SQL/400数据库系统的互连框架。
支持多供应商提供的数据库系统。
支持分布式数据库上的事务(工作单元)处理。
在DRDA中，客户机叫做应用请求器(ARS)，后台服务器叫做应用服务器(AS)，协议叫做应用支持协议(ASP)，提供AR和AS间的接口。整个过程操作在SNA网上，但也计划支持OSI和TCP/IP。有一个附加的协议叫做数据库支持协议(DSP)，它使一个AS能对另一服务器扮演AR的角色。通过这种方法服务器之间能相互通话并传递来自客户AR的请求，如图D-25所示。最初的协议对一个数据库只支持一个结构化查询语言(SQL)的语句，但未来的版本将对一个或多个数据库提供多个语句的支持。
DRDA是IBM环境中建立客户机/服务器计算的基础之一。其它基础是高级的对等联网(APPN)和分布式数据管理

(DDM)。通过信息仓库和DRDA，IBM计算机将它的企业中心组成部分的大型计算机，用作各种类型信息(包括多媒体信息)的存储平台。

Google文件系统GFS

Google文件系统（Google File System，GFS）是一个大型的分布式文件系统。它为Google云计算提供海量存储，并且与Chubby、MapReduce以及Bigtable等技术结合十分紧密，处于所有核心技术的底层。由于GFS并不是一个开源的系统，我们仅仅能从Google公布的技术文档来获得一点了解，而无法进行深入的研究。

当前主流分布式文件系统有RedHat的GFS[3]（Global File System）、IBM的GPFS[4]、Sun的Lustre[5]等。这些系统通常用于高性能计算或大型数据中心，对硬件设施条件要求较高。以Lustre文件系统为例，它只对元数据管理器MDS提供容错解决方案，而对于具体的数据存储节点OST来说，则依赖其自身来解决容错的问题。例如，Lustre推荐OST节点采用RAID技术或SAN存储区域网来容错，但由于Lustre自身不能提供数据存储的容错，一旦OST发生故障就无法恢复，因此对OST的稳定性就提出了相当高的要求，从而大大增加了存储的成本，而且成本会随着规模的扩大线性增长。

正如李开复所说的那样，创新固然重要，但有用的创新更重要。创新的价值，取决于一项创新在新颖、有用和可行性这三个方面的综合表现。Google GFS的新颖之处并不在于它采用了多么令人惊讶的技术，而在于它采用廉价的商用机器构建分布式文件系统，同时将GFS的设计与Google应用的特点紧密结合，并简化其实现，使之可行，最终达到创意新颖、有用、可行的完美组合。GFS使用廉价的商用机器构建分布式文件系统，将容错的任务交由文件系统来完成，利用软件的方法解决系统可靠性问题，这样可以使得存储的成本成倍下降。由于GFS中服务器数目众多，在GFS中服务器死机是经常发生的事情，甚至都不应当将其视为异常现象，那么如何在频繁的故障中确保数据存储的安全、保证提供不间断的数据存储服务是GFS最核心的问题。GFS的精彩在于它采用了多种方法，从多个角度，使用不同的容错措施来确保整个系统的可靠性。

2.1.1 系统架构

GFS的系统架构如图2-1[1]所示。GFS将整个系统的节点分为三类角色：Client（客户端）、Master（主服务器）和Chunk Server（数据块服务器）。Client是GFS提供给应用程序的访问接口，它是一组专用接口，不遵守POSIX规范，以库文件的形式提供。应用程序直接调用这些库函数，并与该库链接在一起。Master是GFS的管理节点，在逻辑上只有一个，它保存系统的元数据，负责整个文件系统的管理，是GFS文件系统中的“大脑”。Chunk Server负责具体的存储工作。数据以文件的形式存储在Chunk Server上，Chunk Server的个数可以有多个，它的数目直接决定了GFS的规模。GFS将文件按照固定大小进行分块，默认是64MB，每一块称为一个Chunk（数据块），每个Chunk都有一个对应的索引号（Index）。

图2-1 GFS体系结构

客户端在访问GFS时，首先访问Master节点，获取将要与之进行交互的Chunk Server信息，然后直接访问这些Chunk Server完成数据存取。GFS的这种设计方法实现了控制流和数据流的分离。Client与Master之间只有控制流，而无数据流，这样就极大地降低了Master的负载，使之不成为系统性能的一个瓶颈。Client与Chunk Server之间直接传输数据流，同时由于文件被分成多个Chunk进行分布式存储，Client可以同时访问多个Chunk Server，从而使得整个系统的I/O高度并行，系统整体性能得到提高。

相对于传统的分布式文件系统，GFS针对Google应用的特点从多个方面进行了简化，从而在一定规模下达到成本、可靠性和性能的最佳平衡。具体来说，它具有以下几个特点。

1．采用中心服务器模式

GFS采用中心服务器模式来管理整个文件系统，可以大大简化设计，从而降低实现难度。Master管理了分布式文件系统中的所有元数据。文件划分为Chunk进行存储，对于Master来说，每个Chunk Server只是一个存储空间。Client发起的所有操作都需要先通过Master才能执行。这样做有许多好处，增加新的Chunk Server是一件十分容易的事情，Chunk Server只需要注册到Master上即可，Chunk Server之间无任何关系。如果采用完全对等的、无中心的模式，那么如何将Chunk Server的更新信息通知到每一个Chunk Server，会是设计的一个难点，而这也将在一定程度上影响系统的扩展性。Master维护了一个统一的命名空间，同时掌握整个系统内Chunk Server的情况，据此可以实现整个系统范围内数据存储的负载均衡。由于只有一个中心服务器，元数据的一致性问题自然解决。当然，中心服务器模式也带来一些固有的缺点，比如极易成为整个系统的瓶颈等。GFS采用多种机制来避免Master成为系统性能和可靠性上的瓶颈，如尽量控制元数据的规模、对Master进行远程备份、控制信息和数据分流等。

2．不缓存数据

缓存（Cache）机制是提升文件系统性能的一个重要手段，通用文件系统为了提高性能，一般需要实现复杂的缓存机制。GFS文件系统根据应用的特点，没有实现缓存，这是从必要性和可行性两方面考虑的。从必要性上讲，客户端大部分是流式顺序读写，并不存在大量的重复读写，缓存这部分数据对系统整体性能的提高作用不大；而对于Chunk Server，由于GFS的数据在Chunk Server上以文件的形式存储，如果对某块数据读取频繁，本地的文件系统自然会将其缓存。从可行性上讲，如何维护缓存与实际数据之间的一致性是一个极其复杂的问题，在GFS中各个Chunk Server的稳定性都无法确保，加之网络等多种不确定因素，一致性问题尤为复杂。此外由于读取的数据量巨大，以当前的内存容量无法完全缓存。对于存储在Master中的元数据，GFS采取了缓存策略，GFS中Client发起的所有操作都需要先经过Master。Master需要对其元数据进行频繁操作，为了提高操作的效率，Master的元数据都是直接保存在内存中进行操作。同时采用相应的压缩机制降低元数据占用空间的大小，提高内存的利用率。

3．在用户态下实现

文件系统作为操作系统的重要组成部分，其实现通常位于操作系统底层。以Linux为例，无论是本地文件系统如Ext3文件系统，还是分布式文件系统如Lustre等，都是在内核态实现的。在内核态实现文件系统，可以更好地和操作系统本身结合，向上提供兼容的POSIX接口。然而，GFS却选择在用户态下实现，主要基于以下考虑。

在用户态下实现，直接利用操作系统提供的POSIX编程接口就可以存取数据，无需了解操作系统的内部实现机制和接口，从而降低了实现的难度，并提高了通用性。

> POSIX接口提供的功能更为丰富，在实现过程中可以利用更多的特性，而不像内核编程那样受限。
>
> 用户态下有多种调试工具，而在内核态中调试相对比较困难。
>
> 用户态下，Master和Chunk Server都以进程的方式运行，单个进程不会影响到整个操作系统，从而可以对其进行充分优化。在内核态下，如果不能很 好地掌握其特性，效率不但不会高，甚至还会影响到整个系统运行的稳定性。
>
> 用户态下，GFS和操作系统运行在不同的空间，两者耦合性降低，从而方便GFS自身和内核的单独升级。

4．只提供专用接口

通常的分布式文件系统一般都会提供一组与POSIX规范兼容的接口。其优点是应用程序可以通过操作系统的统一接口来透明地访问文件系统，而不需要重新编译程序。GFS在设计之初，是完全面向Google的应用的，采用了专用的文件系统访问接口。接口以库文件的形式提供，应用程序与库文件一起编译，Google应用程序在代码中通过调用这些库文件的API，完成对GFS文件系统的访问。采用专用接口有以下好处。

> 降低了实现的难度。通常与POSIX兼容的接口需要在操作系统内核一级实现，而GFS是在应用层实现的。
>
> 采用专用接口可以根据应用的特点对应用提供一些特殊支持，如支持多个文件并发追加的接口等。
>
> 专用接口直接和Client、Master、Chunk Server交互，减少了操作系统之间上下文的切换，降低了复杂度，提高了效率。

2.1.2 容错机制

1．Master容错

具体来说，Master上保存了GFS文件系统的三种元数据。

> 命名空间（Name Space），也就是整个文件系统的目录结构。
>
> Chunk与文件名的映射表。
>
> Chunk副本的位置信息，每一个Chunk默认有三个副本。

首先就单个Master来说，对于前两种元数据，GFS通过操作日志来提供容错功能。第三种元数据信息则直接保存在各个Chunk Server上，当Master启动或Chunk Server向Master注册时自动生成。因此当Master发生故障时，在磁盘数据保存完好的情况下，可以迅速恢复以上元数据。为了防止Master彻底死机的情况，GFS还提供了Master

远程的实时备份，这样在当前的GFS Master出现故障无法工作的时候，另外一台GFS Master可以迅速接替其工作。

2．Chunk Server容错

GFS采用副本的方式实现Chunk Server的容错。每一个Chunk有多个存储副本（默认为三个），分布存储在不同的Chunk Server上。副本的分布策略需要考虑多种因素，如网络的拓扑、机架的分布、磁盘的利用率等。对于每一个Chunk，必须将所有的副本全部写入成功，才视为成功写入。在其后的过程中，如果相关的副本出现丢失或不可恢复等状况，Master会自动将该副本复制到其他Chunk Server，从而确保副本保持一定的个数。尽管一份数据需要存储三份，好像磁盘空间的利用率不高，但综合比较多种因素，加之磁盘的成本不断下降，采用副本无疑是最简单、最可靠、最有效，而且实现的难度也最小的一种方法。

GFS中的每一个文件被划分成多个Chunk，Chunk的默认大小是64MB，这是因为Google应用中处理的文件都比较大，以64MB为单位进行划分，是一个较为合理的选择。Chunk Server存储的是Chunk的副本，副本以文件的形式进行存储。每一个Chunk以Block为单位进行划分，大小为64KB，每一个Block对应一个32bit的校验和。当读取一个Chunk副本时，Chunk Server会将读取的数据和校验和进行比较，如果不匹配，就会返回错误，从而使Client选择其他Chunk Server上的副本。

2.1.3 系统管理技术

严格意义上来说，GFS是一个分布式文件系统，包含从硬件到软件的整套解决方案。除了上面提到的GFS的一些关键技术外，还有相应的系统管理技术来支持整个GFS的应用，这些技术可能并不一定为GFS所独有。

1．大规模集群安装技术

安装GFS的集群中通常有非常多的节点，文献[1]中最大的集群超过1000个节点，而现在的Google数据中心动辄有万台以上的机器在运行。因此迅速地安装、部署一个GFS的系统，以及迅速地进行节点的系统升级等，都需要相应的技术支撑。

2．故障检测技术

GFS是构建在不可靠的廉价计算机之上的文件系统，由于节点数目众多，故障发生十分频繁，如何在最短的时间内发现并确定发生故障的Chunk Server，需要相关的集群监控技术。

3．节点动态加入技术

当有新的Chunk Server加入时，如果需要事先安装好系统，那么系统扩展将是一件十分烦琐的事情。如果能够做到只需将裸机加入，就会自动获取系统并安装运行，那么将会大大减少GFS维护的工作量。

4．节能技术

有关数据表明，服务器的耗电成本大于当初的购买成本，因此Google采用了多种机制来降低服务器的能耗，例如对服务器主板进行修改，采用蓄电池代替昂贵的UPS（不间断电源系统），提高能量的利用率。Rich Miller 在一篇关于数据中心的博客文章中表示，这个设计让 Google 的 UPS 利用率达到99.9%，而一般数据中心只能达到92%～95%。

分布式系统存储设计

曾几何时，分布式计算成为一种潮流，伴之而来的分散存储所带来的高额管理费用已成为企业的一大笔开支，且给企业的业务发展带来许多负面影响。存储整合解决方案正是针对这类情况所设计。

现状及问题

随着计算机应用的不断深入，目前企业的业务系统采用计算机来实现和存储关键数据，已是非常普遍的现象。在过去的几年，有这么一种观点：企业采用计算机系统和存储系统最好采用“分布式计算”的方式。这种“分布式计算”的核心是：企业中每个部门选择各自不同的电脑系统。因为每个部门各有各的业务，对不同品牌和平台也各有偏好，选择各自熟悉的平台，以部门为单位分开，可以方便管理，同时节约成本。随着这几年的实践，人们发现这一想法走向了它初衷的反面，具体体现在：

1. 数据格式的不统一所导致的管理困难，不利于业务的发展。多年以来，各地方各自为政，按照自己的喜好采购硬件，开发软件系统，导致不同的系统平台，数据格式也不完全统一，数据之间的迁移 / 转换更加复杂，需要额外的硬件和软件支持；而且，最为关键的是总公司没有一本完整的账目，不能做出及时的决策，对企业的发展和竞争极为不利；

2. 分布式管理所带的巨额管理费用已经得不偿失。以前，企业往往会陷入一个误区，认为分布式管理会节省费用，但一旦购买了一定量的存储容量后，管理费用（维护费用，升级费用，人工费用等）成为最大的支出项目，而且这笔支出将贯穿整个产品的生命周期，因此管理费用是整套产品的最大投资，而且这个管理费用还在不断上

升。右图比较了三种不同存储分布的情况下，花相同的管理费用实际可管理到的存储容量。第一种情况为一个跨地域的企业将存储分散在不同的地域分开管理；第二种情况为一个跨地域的企业将存储集中在一个地方，但依然分为不同系统管理。第三种情况为完全存储整合方案，即同一系统同一地域的管理。如图所示，第三种方式可极大的降低成本同时方便管理。

图：同样的开支在不同环境下所能管理的容量
因此，银行，电信等均在着手对存储数据进行整合。

今天来说，集中化的存储管理思想是一种非常有效的经济的存储管理解决方案。因为对于磁盘阵列来说，只有一套管理系统，这样就可以极为方便地进行磁盘监控，性能调试。增加或者重新配置磁盘也变得非常简单。最大化的合成集中设备，也使得存储系统的宕机风险降至最低。同时，一套完善的备份方案就可以有效地进行数据备份及恢复。
惠普提供两种存储整合解决方案。
1. 单一存储设备整合：

将数据库和OA的数据集中存放在同一存储设备上。该存储设备支持开放标准，可连结不同平台的服务器(hp,sun,IBM、NTServer等），如hp的SureStore系列存储设备中的XP512、XP48等。下图为整合后的逻辑示意图：
2. 存储区域网（SAN）整合
存储整合的终极目标是构成存储区域网，即由存储区域网完成数据的读、写、备份和容灾，无需占用企业内部的通信网络（LAN-Free）。前端的用户无需考虑数据的存放位置，数据保护机制，备份策略等，仅仅象用自来水那样拧开水龙头。具体请参照存储网络方案介绍。

下图为整合后的存储区域网的示意图：
对比两种存储解决方案，两者均采用惠普的同一存储设备，因而能够
1. 有效利用总存储空间，节约投资；假设某台主机设备的存储空间不足，均可动态地进行调整，将其他主机暂时未占用的空间进行分配，从而避免重复投资。
2. 能够集中备份和维护，轻松管理；设备集中在某一中心，一方面，只需少量的专业管理人员，维护起来也更加容易，另一方面，总公司也便于提起数据来支持决策支持系统。
3. 只需与一家厂商沟通，便于得到最好的服务。和以前方式不同，并非每家主机系统配置自己的存储设备，因而，一旦出现问题，需要和多家厂商进行沟通，甚至还会出现踢皮球现象，现在存储设备能够支持多平台，不会存在此类现象。
但前一种解决方案，需要通过局域网实现某些存储和备份功能，存在网络瓶颈，而采用SAN整合方式，则解决了网络瓶颈问题，最大限度地提高整套系统的性能。
方案的优点
存储整合一直是惠普公司关注的焦点，同时惠普公司在用户存储系统整合的方案设计和实施方面积累了丰富的经验和大量的成功案例。通过建立惠普存储整合计算环境后：
1. 可以极大地保护用户的投资，降低用户管理费用，从而
确保整体拥有成本最低。
2. 降低管理难度，维护数据管理的统一性。
3. 提高了电子化数据管理的可靠性。数据的集中化管理，
能够确保数据的一致性和完整性，保证电子化数据的可
靠性。
4. 开放的标准，能够支持多平台的整合，包括：hp UX,
IBM AIX, Sun Solaris和Microsoft,Windows NT, W2K.
方案配置
单一存储设备整合配置：
服务器：
Option1: hp 9000 Unix服务器；
hp NetServer PC服务器；
Option2: IBM AIX或COMPAQ ALPHA,
SUN Solaris其他NT服务器
存储设备：
hp SureStore XP48或XP512磁盘阵列
hp SureStore Tape Library 6/140,10/180,20/700

软件：
hp SureSoft软件
Option1: hp OmnibackII备份软件及选件
Option2: Veritas NetBackup(hp UX, Solaris, NT)
备份软件及选件
Option3: Veritas Backup Exec(NT)备份软件及选件
Option4:CA ARCserve备份软件及选件
Option5: Legato Networker备份软件及选件
hp SAN存储设备整合配置：
服务器：
Option1: hp 9000 Unix服务器；
hp NetServer PC服务器；
Option2: IBM AIX 或COMPAQALPHA,
SUN Solaris其他NT服务器
存储设备：
hp SureStore XP48或XP512磁盘阵列
hp SureStore Tape Library 6/140,10/180,20/700
其他硬件设备：
hp SureStore SCSI Bridge FC4/2
Brocade Silkworm Switch 2400/2800
Emulex LP8000 主机总线适配卡（NT/WIN2K/AIX）
Qlogic QLA2200F主机总线适配卡（NT/WIN2K）
JNI FCE6410-N主机总线适配卡（NT/WIN2K）
JNI FC641063主机总线适配卡（Solaris, Sbus总线）
JNI FCI-1063 主机总线适配卡（Solaris,PCI总线）
光纤通道或广域网接口
软件：
hp SureSoft软件
Option1: hp OmnibackII备份软件及选件
Option2: Veritas NetBackup(hp UX, Solaris, NT)
备份软件及选件
Option3: Veritas Backup Exec(NT)备份软件及选件
Option4: CA ARCserve备份软件及选件
Option5: Legato Networker备份软件及选件
适用范围
惠普的存储整合方案适用于企业计算环境中有不同的存储平台，或存储设备分散在不同场所，希望通过存储整合降低成本，同时提高可管理性的场合。

可维护性分布式存储系统和分离的分布式系统

1.分布式系统的应该有两种基本的层次的架构。

1.1.普通的分布式系统架构，是典型的三层的架构，如下图的分离的分布式系统的一个子系统。

1.2.多个分布式系统构成的分布式系统的超级，可以构建云服务的分布式系统。

2.普通的分布式系统的构成

2.1简单分布式的组合构成的服务系统

一般的分布式系统都具有三层架构层次，hand，master，svr。master保存路由表，hand为接入，svr为真正的逻辑服务器。可以对master的路由表操作实现分布。master动态探测svr的存活，进行路由表的更新。

如果单纯的逻辑服务器（不带cache），这里的路由表的更新较为方便和简洁。如果是带有cache 的逻辑服务器则需要根据是脏cache还是干净cache，并迁移cache 的数据。由于在迁移过程中一般会对服务有影响，如果是脏cache，则需要等到迁移完成才能进行服务，这里就有个格子锁定的状态。因此对于具有脏cache 的服务分布式系统，复杂程度大了很多。并且在迁移的时候会影响到服务。

图一 各个分布式的简单组合构成的服务系统典型架构

上图中有些系统没有hand模块是因为其master可能只需要管理其路由表。而路由表的拉取直接放在上一层的逻辑服务器。其接入机是镜像的接入。这种逻辑服务系统一般只能较小规模的业务接入和应用。原有有以下几点：

2.1.1.各个自己系统的具有自己的简单容灾机制，但是没有总体的容灾机制，

2.1.2.所有的接入是镜像的，则其配置都是一样的，如果这个系统承载多个服务，则无法轻松对其近区别服务，迁移和管理。即一个系统一旦搭建也运行，则很难对业务应用的服务进行控制，控制的粒度则一个系统级别。

2.1.3.其承载的业务愈多，则风险系数陡增。其接入服务器的bug会 导致所有服务的中断，其后端的逻辑服务器的bug会导致其一个系统的瘫痪。其业务的灰度升级无法做到真正意义上的灰度。因为其子系统没有业务管理的功能。无论灰度那一台机器，其影响都是系统的所有业务，一旦出现bug，将会对所有的业务产生影响。无法做到业务和机器的灰度，只能做到机器这个低级别的灰度。

因此这种分布式的简单组合无法承载较大业务的运行，只有将一定数量的业务放在一个系统，这样风险或许可承受，但是如果系统有所发展，接入的业务和应用越来越多，将会维护这样若干的系统。造成运维上的麻烦和低效率。

3.可维护的分布式系统

3.1 下面介绍可维护的分布式系统的构成和基本设计思想。目标就是要构建支持大规模业务应用的存储分布式存储系统。

既然是存储系统，那么就会容纳许多业务，这个系统才有意义。当前的能够容纳很多业务的成熟概念就是云的概念，没错，我么就是要构建这样一个具有云系统概念的分布式存储系统。

其基本目标就是要能够轻松的做到对不同业务的管理和维护，调度。在大规模的业务的场景下，能够很好的支撑和运营。

下面是其基本的架构思想。

3.2基本设计思想：

3.2.1 整个系统由统一的接入，统一的控制中心组成，后端的逻辑分布式子系统组成

3.2.2 整个系统的master都具有容灾，路由和按照一个中心master的指令来进行操作的功能，即中心master具有管理的功能，能够对业务进行控制和管理，然后根据策略中心生成的策略对整个系统进行均衡，迁移和管理操作。

3.2.3 支持大规模的业务接入，提高系统的安全性。如果某一个业务有故障，中心master可以根据一定的策略将其布置在单独的机器行，并且将其他业务部署在其他机器上，实现业务的快速隔离。

3.2.4 其对业务级别和机器级别，模块级别的灰度将表现非常优秀。中心master可以部署某个业务的某些访问在灰度的机器上。灰度的控制非常灵活和到位。

4.总结

具有可维护的分布式系统需要一个统一的控制中心。即cmaster（中心控制器）。可以对业务进行很好的调度和控制，所有的master

都应该具有这样的功能，具有调度和控制的功能。这样的系统才具有云服务系统的基础

集中式，分布式，协作式数据处理的区别

1）集中式数据处理

集中式计算机网络由一个大型的中央系统，其终端是客户机，数据全部存储在中央系统，由数据库管理系统进行管理，所有的处理都由该大型系统完成，终端只是用来输入和输出。终端自己不作任何处理，所有任务都在主机上进行处理。

集中式数据存储的主要特点是能把所有数据保存在一个地方，各地办公室的远程终端通过电缆同中央计算机（主机）相联，保证了每个终端使用的都是同一信息。备份数据容易，因为他们都存储在服务器上，而服务器是唯一需要备份的系统。这还意味这服务器是唯一需要安全保护的系统，终端没有任何数据。银行的自动提款机（ATM）采用的就是集中式计算机网络。另外所有的事务都在主机上进行处理，终端也不需要软驱，所以网络感染病毒的可能性很低。这种类型的网络总费用比较低，因为主机拥有大量存储空间、功能强大的系统，而使终端可以使用功能简单而便宜的微机和其他终端设备。

这类网络不利的一面是来自所有终端的计算都由主机完成，这类网络处理速度可能有些慢。另外，如果用户有各种不同的需要，在集中式计算机网络上满足这些需要可能是十分困难的，因为每个用户的应用程序和资源都必须单独设置，而让这些应用程序和资源都在同一台集中式计算机上操作，使得系统效率不高。还有，因为所有用户都必须连接到一台中央计算机，集中连接可能成为集中式网络的一个大问题。由于这些限制，如今的大多数网络都采用了分布式和协作式网络计算模型。

2）分布式数据处理

由于个人计算机的性能得到极大的提高及其使用的普及，使处理能力分布到网络上的所有计算机成为可能。分布式计算是和集中式计算相对立的概念，分布式计算的数据可以分布在很大区域。

分布式网络中，数据的存储和处理都是在本地工作站上进行的。数据输出可以打印，也可保存在软盘上。通过网络主要是得到更快、更便捷的数据访问。因为每台计算机都能够存储和处理数据，所以不要求服务器功能十分强大，其价格也就不必过于昂贵。这种类型的网络可以适应用户的各种需要，同时允许他们共享网络的数据、资源和服务。在分布式网络中使用的计算机既能够作为独立的系统使用，也可以把它们连接在一起得到更强的网络功能。

分布式计算的优点是可以快速访问、多用户使用。每台计算机可以访问系统内其他计算机的信息文件；系统设计上具有更大的灵活性，既可为独立的计算机的地区用户的特殊需求服务，也可为联网的企业需求服务，实现系统内不同计算机之间的通信；每台计算机都可以拥有和保持所需要的最大数据和文件；减少了数据传输的成本和风险。为分散地区和中心办公室双方提供更迅速的信息通信和处理方式，为每个分散的数据库提供作用域，数据存储于许多存储单元中，但任何用户都可以进行全局访问，使故障的不利影响最小化，以较低的成本来满足企业的特定要求。

分布式计算的缺点是：对病毒比较敏感，任何用户都可能引入被病毒感染的文件，并将病毒扩散到整个网络。备份困难，如果用户将数据存储在各自的系统上，而不是将他们存储在中央系统中，难于制定一项有效的备份计划。这种情况还可能导致用户使用同一文件的不同版本。为了运行程序要求性能更好的PC机；要求使用适当的程序；不同计算机的文件数据需要复制；对某些PC机要求有足够的存储容量，形成不必要的存储成本；管理和维护比较复杂；设备必须要互相兼容。

3）协作式数据处理

协作式数据处理系统内的计算机能够联合处理数据，处理既可集中实施，也可分区实施。协作式计算允许各个客户计算机合作处理一项共同的任务，采用这种方法，任务完成的速度要快于仅在一个客户计算机运行。协作式计算允许计算机在整个网络内共享处理能力，可以使用其它计算机上的处理能力完成任务。除了具有在多个计算机系统上处理任务的能力，该类型的网络在共享资源方面类似于分布式计算。

协作式计算和分布式计算具有相似的优缺点。例如协作式网络上可以容纳各种不同的客户，协作式计算的优点是处理能力强，允许多用户使用。缺点是病毒可迅速扩散到整个网络。因为数据能够在整个网络内存储，形成多个副本，文件同步困难。并且也使得备份所有的重要数据比较困难。

Google Megastore分布式存储技术全揭秘

Megastore是谷歌一个内部的存储系统，它的底层数据存储依赖Bigtable，也就是基于NoSql实现的，但是和传统的NoSql不同的 是，它实现了类似RDBMS的数据模型(便捷性)，同时提供数据的强一致性解决方案(同一个datacenter，基于MVCC的事务实现)，并且将数据 进行细颗粒度的分区(这里的分区是指在同一个datacenter，所有datacenter都有相同的分区数据)，然后将数据更新在机房间进行同步复制 (这个保证所有datacenter中的数据一致)。

Megastore的数据复制是通过paxos进行同步复制的，也就是如果更新一个数据，所有机房都会进行同步更新，因为使用paxos进行复制， 所以不同机房针对同一条数据的更新复制到所有机房的更新顺序都是一致的，同步复制保证数据的实时可见性，采用paxos算法则保证了所有机房更新的一致 性，所以个人认为megastore的更新可能会比较慢，而所有读都是实时读(对于不同机房是一致的)，因为部署有多个机房，并且数据总是最新。

为了达到高可用性，megastore实现了一个同步的，容错的，适合长距离连接的日志同步器

为了达到高可扩展性，megastore将数据分区成一个个小的数据库，每一个数据库都有它们自己的日志，这些日志存储在NoSql中

Megastore将数据分区为一个Entity Groups的集合，这里的Entity Groups相当于一个按id切分的分库，这个Entity Groups里面有多个Entity Group(相当于分库里面的表)，而一个Entity Group有多个Entity(相当于表中的记录)

在同一个Entity Group中(相当于单库)的多个Entity的更新事务采用single-phase ACID事务，而跨Entity Group(相当于跨库)的Entity更新事务采用two-phase ACID事务(2段提交)，但更多使用Megastore提供的高效异步消息实现。需要说明的一点是，这些事务都是在同一个机房的，机房之间的数据交互都 是通过数据复制来实现的。

传统关系型数据库使用join来满足用户的需求，对于Megastore来说，这种模型(也就是完全依赖join的模型)是不合适的。原因包括

1.高负载交互性型应用能够从可预期的性能提升得到的好处多于使用一种代价高昂的查询语言所带来的好处。

2.Megastore目标应用是读远远多于写的，所以更好的方案是将读操作所需要做的工作转移到写操作上面(比如通过具体值代替外键以消除join)

3.因为megastore底层存储是采用BigTable，而类似BigTable的key-value存储对于存取级联数据是直接的

所以基于以上几个原因，Megastore设计了一种数据模型和模式语言来提供基于物理地点的细颗粒度控制，级联布局，以及申明式的不正规数据存储来帮助消除大部分joins。查询时只要指定特定表和索引即可。

当然可能有时候不得不使用到join，Megastore提供了一种合并连接算法实现，具体算法这里我还是没弄清楚，原文是[the user providesmultiple queries that return primary keys for the same table in the same order;we then return the intersection of keys for all the provided queries.]

使用Megastore的应用通过并行查询实现了outer joins。通常先进行一个初始的查询，然后利用这个查询结果进行并行索引查询，这个过程我理解的是，初始查询查出一条数据，就马上根据这个结果进行并行 查询，这个时候初始查询继续取出下一条数据，再根据这个结果并行查询(可能前面那个外键查询还在继续，使用不同的线程)。这种方法在初始查询数据量较小并 且外键查询使用并行方式的情况下，是一种有效的并且具有sql风格的joins。

Megastore的数据结构介于传统的RDBMS和NoSql之间的，前者主要体现在他的schema表示上，而后者体现在具体的数据存储上 (BigTable)。和RDBMS一样，Megastore的数据模型是定义schema中并且是强类型的。每一个schema有一个表集合，每个表包 含一个实体集合(相当于record)，每个实体有一系列的属性(相当于列属性)，属性是命名的，并且指定类型，这些类型包括字符串，各种数字类型，或者google的protocol buffer。这些属性可以被设置成必需的，可选的，或者可重复的(一个属性上可以具有多个值)。一个或者多个属性可以组成一个主键。

在上图中，User和Photo共享了一个公共属性user_id，IN TABLE User这个标记直接将Photo和User这两张表组织到了同一个BigTable中，并且键的顺序(PRIMARY KEY(user_id,photo_id)？是这个还是schema中定义的顺序？)保证Photo的实体存储在对应的User实体邻接位置上。这个机 制可以递归的应用，加速任意深度的join查询速度。这样，用户能够通过操作键的顺序强行改变数据级联的布局。其他标签请参考原文。

Megastore支持事务和并发控制。一个事务写操作会首先写入对应Entity Group的日志中，然后才会更新具体数据。BigTable具有一项在相同row/column中存储多个版本带有不同时间戳的数据。正是因为有这个特 性，Megastore实现了多版本并发控制(MVCC，这个包括oracle，innodb都是使用这种方式实现ACID，当然具体方式

会有所不同)： 当一个事务的多个更新实施时，写入的值会带有这个事务的时间戳。读操作会使用最后一个完全生效事务的时间戳以避免看到不完整的数据.读写操作不相互阻塞， 并且读操作在写事务进行中会被隔离(？)。

Megastore 提供了current，snapshot，和inconsistent读，current和snapshot级别通常是读取单个entity group。当开始一个current读操作时，事务系统会首先确认所有之前提交的写已经生效了；然后系统从最后一个成功提交的事务时间戳位置读取数据。 对于snapshot读取，系统拿到已经知道的完整提交的事务时间戳并且从那个位置直接读取数据，和current读取不同的是，这个时候可能提交的事务 更新数据还没有完全生效(提交和生效是不同的)。Megastore提供的第三种读就是inconsistent读，这种读无视日志状态并且直接读取最后 一个值。这种方式的读对于那些对减少延迟有强烈需求，并且能够容忍数据过期或者不完整的读操作是非常有用的。

一个写事务通常开始于一个current读操作以便确定下一个可用的日志位置。提交操作将数据变更聚集到日志，并且分配一个比之前任何一个都高的时 间戳，并且使用Paxos将这个log entry加入到日志中。这个协议使用了乐观并发：即使有可能有多个写操作同时试图写同一个日志位置，但只会有1个成功。所有失败的写都会观察到成功的写 操作，然后中止，并且重试它们的操作。咨询式的锁定能够减少争用所带来的影响。通过特定的前端服务器分批写入似乎能够完全避免竞争(这几句有些不能理解) [Advisory lockingis available to reduce the effects of contention. Batching writes throughsession affinity to a particular front-end server can avoid contentionaltogether.]。

完整事务生命周期包括以下步骤：

1.读：获取时间戳和最后一个提交事务的日志位置

2.应用逻辑：从BigTable读取并且聚集写操作到一个日志Entry

3.提交：使用Paxos将日志Entry加到日志中

4.生效：将数据更新到BigTable的实体和索引中

5.清理：删除不再需要的数据

写操作能够在提交之后的任何点返回，但是最好还是等到最近的副本(replica)生效(再返回)。

Megastore提供的消息队列提供了在不同Entity Group之间的事务消息。它们能被用作跨Entity Group的操作，在一个事务中分批执行多个更新，或者延缓工作(？)。一个在单个Entity Group上的事务能够原子性地发送或者收到多个信息除了更新它自己的实体。每个消息都有一个发送和接收的Entity Group；如果这两个Entity Group是不同的，那么传输将会是异步的。

消息队列提供了一种将会影响到多个Entity Group的操作的途径，举个例子，日历应用中，每一个日历有一个独立的Entity Group，并且我们现在需要发送一个邀请到多个其他人的日历中，一个事务能够原子地发送邀请消息到多个独立日历中。每个日历收到消息都会把邀请加入到它 自己的事务中，并且这个事务会更新被邀请人状态然后删除这个消息。Megastore大规模使用了这种模式：声明一个队列后会自动在每一个Entity Group上创建一个收件箱。

Megastore支持使用二段提交进行跨Entity Group的原子更新操作。因为这些事务有比较高的延迟并且增加了竞争的风险，一般不鼓励使用。

接下来内容具体来介绍下Megastore最核心的同步复制模式：一个低延迟的Paxos实现。Megastore的复制系统向外提供了一个单一 的，一致的数据视图，读和写能够从任何副本(repli ca)开始，并且无论从哪个副本的客户端开始，都能保证ACID语义。每个Entity Group复制结束标志是将这个Entity Group事务日志同步地复制到一组副本中。写操作通常需要一个数据中心内部的网络交互，并且会跑检查健康状况的读操作。current级别的读操作会有以下保证：

1.一个读总是能够看到最后一个被确认的写。(可见性)

2.在一个写被确认后，所有将来的读都能够观察到这个写的结果。(持久性，一个写可能在确认之前就被观察到)

数据库典型使用Paxos一般是用来做事务日志的复制，日志中每个位置都由一个Paxos实例来负责。新的值将会被写入到之前最后一个被选中的位置之后。

Megastore在事先Paxos过程中，首先设定了一个需求，就是current reads可能在任何副本中进行，并且不需要任何副本之间的RPC交互。因为写操作一般会在所有副本上成功，所以允许在任何地方进行本地读取是现实的。这些本地读取能够很好地被利用，所有区域的低延迟，细颗粒度的读取failover，还有简单的编程体验。

Megastore设计实现了一个叫做Coordinator(协调者)的服务，这个服务分布在每个副本的数据中心里面。一个Coordinator服务器跟踪一个Entity Groups集合，这个集合中的Entity Groups需要具备的条件就是它们的副本已经观察到了所有的Paxos写。在这个集合中的Entity Groups，它们的副本能够进行本地读取(local read)。

写操作算法有责任保持Coordinator状态是保守的，如果一个写在一个副本上失败了，那么这次操作就不能认为是提交的，直到这个entity group的key从这个副本的coordinator中去除。(这里不明白)

为了达到快速的单次交互的写操作，Megastore采用了一种Master-Slave方式的优化，如果一次写成功了，那么会顺带下一次写的保证 (也就是下一次写就不需要prepare去申请一个log position)，下一次写的时候，跳过prepare过程，直接进入accept阶段。Megastore没有使用专用的Masters，但是使用 Leaders。

Megastore为每一个日志位置运行一个Paxos算法实例。[The leader for each log position is a

distinguished replicachosen alongside the preceding log position's consensus value.] Leader仲裁在0号提议中使用哪一个值。第一个写入者向Leader提交一个值会赢得一个向所有副本请求接收这个值做为0号提议最终值的机会。所有其 他写入者必需退回到Paxos的第二阶段。

因为一个写入在提交值到其他副本之前必需和Leader交互，所以必需尽量减少写入者和Leader之间的延迟。Megastore设计了它们自己 的选取下一个写入Leader的规则，以同一地区多数应用提交的写操作来决定。这个产生了一个简单但是有效的原则：使用最近的副本。(这里我理解的是哪个 位置提交的写多，那么使用离这个位置最近的副本做为Leader)

Megastore的副本中除了有日志有Entity数据和索引数据的副本外，还有两种角色，其中一种叫做观察者(Witnesses)，它们只写 日志，并且不会让日志生效，也没有数据，但是当副本不足以组成一个quorum的时候，它们就可以加入进来。另外一种叫只读副本(Read-Only)， 它刚刚和观察者相反，它们只有数据的镜像，在这些副本上只能读取到最近过去某一个时间点的一致性数据。如果读操作能够容忍这些过期数据，只读副本能够在广阔的地理空间上进行数据传输并且不会加剧写的延迟。

上图显示了Megastore的关键组件，包括两个完整的副本和一个观察者。应用连接到客户端库，这个库实现了Paxos和其他一些算法：选择一个副本进行读，延迟副本的追赶，等等。

Each applicationserver has a designated local replica. The client library makes Paxosoperations on that replica durable by submitting transactions directly to thelocal Bigtable.To minimize wide-area roundtrips, the library submits remotePaxos operations to stateless intermediary replication servers communicatingwith their local Bigtables.

客户端，网络，或者BigTable失败可能让一个写操作停止在一个中间状态。复制的服务器会定期扫描未完成的写入并且通过Paxos提议没有操作的值来让写入完成。

接下来介绍下Megastore的数据结构和算法，每一个副本存有更新和日志Entries的元数据。为了保证一个副本能够参与到一个写入的投票中 即使是它正从一个之前的宕机中恢复数据，Megastore允许这个副本接收不符合顺序的提议。Megastore将日志以独立的Cells存储在 BigTable中。

当日志的前缀不完整时(这个前缀可能就是一个日志是否真正写入的标记，分为2段，第一段是在写入日志之前先写入的几个字节，然后写入日志，第二段是 在写入日志之后写入的几个字节，只有这个日志前缀是完整的，这个日志才是有效的)，日志将会留下holes。下图表示了一个单独Megastore Entity Group的日志副本典型场景。0-99的日志位置已经被清除了，100的日志位置是部分被清除，因为每个副本都会被通知到其他副本已经不需要这个日志 了。101日志位置被所有的副本接受了(accepted)，102日志位置被Y所获得，103日志位置被A和C副本接受，B副本留下了一个 hole，104日志位置因为副本A和B的不一致，复本C的没有响应而没有一致结果。

在一个current读的准备阶段(写之前也一样)，必需有一个副本要是最新的：所有之前更新必需提交到那个副本的日志并且在该副本上生效。我们叫这个过程为catchup。

省略一些截止超时的管理，一个current读算法步骤如下：

1.本地查询：查询本地副本的Coordinator，判定当前副本的Entity Group是最新的

2.查找位置：确定最高的可能已提交的日志位置，然后选择一个已经将这个日志位置生效的副本

a.(Local read) 如果步骤1发现本地副本是最新的，那么从本地副本中读取最高的被接受(accepted)的日志位置和时间戳。

b.(Majority read)如果本地副本不是最新的(或者步骤1或步骤2a超时)，那么从一个多数派副本中发现最大的日志位置，然后选取一个读取。我们选取一个最可靠的或者最新的副本，不一定总是是本地副本

3.追赶：当一个副本选中之后，按照下面的步骤追赶到已知的日志位置:

a.对于被选中的不知道共识值的副本中的每一个日志位置，从另外一个副本中读取值。对于任何一个没有已知已提交的值的日志位置，发起一个没有操作的写操作。Paxos将会驱动多数副本在一个值上打成共识-----可能是none-op的写操作或者是之前提议的写操作

b.顺序地将所有没有生效的日志位置生效成共识的值，并将副本的状态变为到分布式共识状态(应该是Coordinator的状态更新)

如果失败，在另外一个副本上重试。

4.验证：如果本地副本被选中并且之前没有最新，发送一个验证消息到coordinator断定(entitygroup,replica)能够反馈(reflects)所有提交的写操作。不要等待回应----如果请求失败，下一个读操作会重试。

5.查询数据：从选中的副本中使用日志位置所有的时间戳读取数据。如果选中的副本不可用，选取另外一个副本重新开始执行追赶，然后从它那里读取。一个大的读取结果有可能从多个副本中透明地读取并且组装返回

注意在实际使用中 1和2a通常是并行执行的。

在完整的读操作算法执行后，Megastore发现了下一个没有使用的日志位置，最后一个写 操作的时间戳，还有下一个leader副本。在提交时刻，所有更新的状态都变为打包的(packaged)和提议(proposed)，并且包含一个时间戳和下一个leader 候选人，做为下一个日志位置的共识值。如果这个值赢得了分布式共识，那么这个值将会在所有完整的副本中生效。否则整个事务将会终止并且必需重新从读阶段开 始。

就像上面所描述的，Coordinators跟踪Entity Groups在它们的副本中是否最新。如果一个写操作没有被一个副本接受，我们必需将这个Entity Group的键从这个副本的Coordinator中移除。这个步骤叫做invalidation(失效)。在一个写操作被认为提交的并且准备生效，所有 副本必需已经接受或者让这个Entity Group在它们coordinator上失效。

写算法的步骤如下：

1.接受Leader：请求Leader接受值做为0号提议的值。如果成功。跳到第三步

2.准备：在所有副本上执行Paxos Prepare阶段，使用一个关于当前log位置更高的提议号。将值替换成拥有最高提议号的那个值。[Replace the valuebeing written withthe highest-numbered proposal discovered, if any]

3.接受：请求余下的副本接受这个值。如果多数副本失败，转到第二步。

4.失效：将没有接受值的副本coordinator失效掉。错误处理将在接下来描述

5.生效：将更新在尽可能多的副本上生效。如果选择的值不同于原始提议的，返回冲突错误[？]

Coordinator进程在每一个数据中心运行并且只保持其本地副本的状态。在上述的写入算法中，每一个完整的副本必需接受或者让其 coordinator失效，所以这个可能会出现任何单个副本失效就会引起不可用。在实际使用中这个不是一个寻常的问题。Coordinator是一个简 单的进程，没有其他额外的依赖并且没有持久存储，所以它表现得比一个BigTable服务器更高的稳定性。然而，网络和主机失败仍然能够让 coordinator不可用。

Megastore使用了Chubby锁服务：Coordinators在启动的时候从远程数据中心获取指定的Chubby locks。为了处理请求，一个Coordinator必需持有其多数locks。一旦因为宕机或者网络问题导致它丢失了大部分锁，它就会恢复到一个默认 保守状态----认为所有在它所能看见的Entity Groups都是失效的。随后(该Coordinator对应的)副本中的读操作必需从多数其他副本中得到日志位置直到Coordinator重新获取到 锁并且Coordinator的Entries重新验证的。

写入者通过测试一个Coordinator是否丢失了它的锁从而让其在Coordinator不可用过程中得到保护：在这个场景中，一个写入者知道在恢复之前Coordinator会认为自己是失效的。

在一个数据中心活着的Coordinator突然不可用时，这个算法需要面对一个短暂(几十秒)的写停顿风险---所有的写入者必需等待 Coordinator的Chubby locks过期(相当于等待一个master failover后重新启动)，不同于master failover，写入和读取都能够在coordinator状态重建前继续平滑进行。

除了可用性问题，对于Coordinator的读写协议必需满足一系列的竞争条件。失效的信息总是安全的，但是生效的信息必需小心处理。在 coordinator中较早的写操作生效和较晚的写操作失效之间的竞争通过带有日志位置而被保护起来。标有较高位置的失效操作总是胜过标有较低位置的生 效操作。一个在位置n的失效操作和一个在位置m<n的生效操作之间的竞争常常和一个crash联系在一起。Megastore通过一个具有时间期限 的数字代表

Coordinator来侦测crashes：生效操作只允许在最近一次对Coordinator进行的读取操作以来时间期限数字没变化的情况 下修改Coordinator的状态。

总体来说，使用Coordinator从而能够在任何数据中心进行快速的本地读取对于可用性的影响并不是完全没有的。但是实际上，以下因素能够减轻使用Coordinator所带来的问题。

1.Coordinators是比任何的BigTable 服务器更加简单进程，机会没有依赖，所以可用性更高。

2.Coordinators简单，均匀的工作负载让它们能够低成本地进行预防措施。

3.Coordinators轻量的网络传输允许使用高可用连接进行服务质量监控。

4.管理员能够在维护期或者非安全期集中地让一批Coordinators失效。对于默写信号的监测是自动的。

5.一个Chubby qunrum能够监测到大多数网络问题和节点不可用。

总结

文章总体介绍了下google megastore的实现思路，其主要解决的问题就是如何在复杂的环境下(网络问题,节点失效等等)保证数据存取服务的可用性。对于多机房，多节点，以及ACID事务支持，实时非实时读取，错误处理等等关键问题上给出了具体方案。

Oceanbase——千亿级海量数据库

从大学的数据结构课程可以知道，数据量比较大时，有两种数据结构很常用：哈希表和B+树，分布式系统也是类似的。如下图：

Amazon的系统实现了一个分布式哈希表，而Google Bigtable, Yahoo PNUTS，Microsoft SQL Azure实现了一颗分布式B+树。分布式哈希表实现相对简单，但只支持随机读取；而分布式B+树支持范围查询，但实现比较复杂，主要有两个难点：

1, 状态数据的持久化和迁移。更新操作改变系统的状态，数据库系统中更新操作首先以事务提交日志（MySQL称为binlog, NOSQL称为commit log）写入到磁盘，为了保证可靠性，commit log需要复制多份并保证它们之间的一致性。另外，机器宕机时需要通过commit log记录的状态修改信息将服务迁移到集群中的其它节点。

2, 子表的分裂和合并。B+树实现的难点在于树节点的分裂与合并，在分布式系统中，数据被顺序划分为大小在几十到几百MB大小的数据范围，一般称为子表，相当于B+树结构中的叶子节点。由于每个子表在系统中存储多份，需要保证多个副本之间的分裂点是一致的。由于子表在分裂的同时也有更新操作，保证多个副本之间一致是比较困难的。

对于这两个问题，不同的系统有不同的解决方法：

1. 状态维持。

Google Bigtable将状态数据写入到GFS中，由GFS提供可靠性保证，但GFS本身是一个巨大的工程；Yahoo PNUTS将状态数据写入到分布式消息中间件，Yahoo内部称为Yahoo Message Broker；Microsoft SQL Azure直接通过网络将数据复制到多机，由于一台机器服务多个子表，这些子表的副本可能分布在整个集群中，因此，任何两台机器都可能建立数据复制的网络通道，需要处理与这些通道有关的异常情况。

2.子表分裂。

由于底层有GFS保证可靠性，Google Bigtable设计时保证每一个子表同时只被一台机器(Tablet Server)服务；Yahoo PNUTS通过引入复杂的两节点提交(Two-phase commit)协议协调多个副本之间的一致性，使得他们的分裂点相同；Microsoft SQL Azure干脆不支持子表分裂，牺牲一部分扩展性从而简化系统设计。

淘宝Oceanbase设计之初对淘宝的在线存储需求进行分析发现：淘宝的数据总量比较大，未来一段时间，比如五年之内的数据规模为百TB级别，千亿条记录，另外，数据膨胀很快，传统的分库分表对业务造成很大的压力，必须设计自动化的分布式系统；然而，在线存储每天的修改量很小，大多数情况下单机的内存就能存放下。因此，我们采用将动态数据和静态数据分离的办法。动态数据的数据量小，采用集中式的方法解决，这样，状态数据维持从一个分布式的问题转化为单机的问题；静态数据的数据量大，采用分布式的方法解决，因为静态数据基本不变，实现时不需要复杂的线程同步机制，另外，保证静态数据的多个副本之间一致性是比较容易的，简化了子表的分裂和合并操作。通过这样的权衡，淘宝Oceanbase以一种很简单的方式满足了未来一段时间的在线存储需求，并且还获得了一些其它特性，如高效支持跨行跨表事务，这对于淘宝的业务是非常重要的。另外，我们之所以敢于做这样的权衡，还有一个重要的原因：我们内部已经思考了很多关于动态数据由集中式变为分布式的方案，即使我们对需求估计有些偏差，也可以很快修改原有系统进一步提高可扩展性

分布式计算开源框架Hadoop入门实践

Hadoop是Apache开源组织的一个分布式计算开源框架，在很多大型网站上都已经得到了应用，如亚马逊、Facebook和Yahoo等等。

在SIP项目设计的过程中，对于它庞大的日志在开始时就考虑使用任务分解的多线程处理模式来分析统计，在我从前写的文章《Tiger ConcurrentPractice --日志分析并行分解设计与实现》中有所提到。但是由于统计的内容暂时还是十分简单，所以就采用Memcache作为计数器，结合MySQL就完成了访问控制以及统计的工作。然而未来，对于海量日志分析的工作，还是需要有所准备。现在最火的技术词汇莫过于“云计算”，在Open API日益盛行的今天，互联网应用的数据将会越来越有价值，如何去分析这些数据，挖掘其内在价值，就需要分布式计算来支撑海量数据的分析工作。

回过头来看，早先那种多线程，多任务分解的日志分析设计，其实是分布式计算的一个单机版缩略，如何将这种单机的工作进行分拆，变成协同工作的集群，其实就是分布式计算框架设计所涉及的。在去年参加BEA大会的时候，BEA和VMWare合作采用虚拟机来构建集群，无非就是希望使得计算机硬件能够类似于应用程序中资源池的资源，使用者无需关心资源的分配情况，从而最大化了硬件资源的使用价值。分布式计算也是如此，具体的计算任务交由哪一台机器执行，执行后由谁来汇总，这都由分布式框架的Master来抉择，而使用者只需简单地将待分析内容提供给分布式计算系统作为输入，就可以得到分布式计算后的结果。

Hadoop是Apache开源组织的一个分布式计算开源框架，在很多大型网站上都已经得到了应用，如亚马逊、Facebook和Yahoo等等。对于我来说，最近的一个使用点就是服务集成平台的日志分析。服务集成平台的日志量将会很大，而这也正好符合了分布式计算的适用场景（日志分析和索引建立就是两大应用场景）。

当前没有正式确定使用，所以也是自己业余摸索，后续所写的相关内容，都是一个新手的学习过程，难免会有一些错误，只是希望记录下来可以分享给更多志同道合的朋友。

什么是Hadoop？

搞什么东西之前，第一步是要知道What（是什么），然后是Why（为什么），最后才是How（怎么做）。但很多开发的朋友在做了多年项目以后，都习惯是先How，然后What，最后才是Why，这样只会让自己变得浮躁，同时往往会将技术误用于不适合的场景。

Hadoop框架中最核心的设计就是：MapReduce和HDFS。MapReduce的思想是由Google的一篇论文所提及而被广为流传的，简单的一句话解释MapReduce就是“任务的分解与结果的汇总”。HDFS是Hadoop分布式文件系统（Hadoop Distributed File System）的缩写，为分布式计算存储提供了底层支持。

MapReduce从它名字上来看就大致可以看出个缘由，两个动词Map和Reduce，“Map（展开）”就是将一个任务分解成为多个任务，“Reduce”就是将分解后多任务处理的结果汇总起来，得出最后的分析结果。这不是什么新思想，其实在前面提到的多线程，多任务的设计就可以找到这种思想的影子。不论是现实社会，还是在程序设计中，一项工作往往可以被拆分成为多个任务，任务之间的关系可以分为两种：一种是不相关的任务，可以并行执行；另一种是任务之间有相互的依赖，先后顺序不能够颠倒，这类任务是无法并行处理的。回到大学时期，教授上课时让大家去分析关键路径，无非就是找最省时的任务分解执行方式。在分布式系统中，机器集群就可以看作硬件资源池，将并行的任务拆分，然后交由每一个空闲机器资源去处理，能够极大地提高计算效率，同时这种资源无关性，对于计算集群的扩展无疑提供了最好的设计保证。（其实我一直认为Hadoop的卡通图标不应该是一个小象，应该是蚂蚁，分布式计算就好比蚂蚁吃大象，廉价的机器群可以匹敌任何高性能的计算机，纵向扩展的曲线始终敌不过横向扩展的斜线）。任务分解处理以后，那就需要将处理以后的结果再汇总起来，这就是Reduce要做的工作。

MapReduce结构示意图

上图就是MapReduce大致的结构图，在Map前还可能会对输入的数据有Split（分割）的过程，保证任务并行效率，在Map之后还会有Shuffle（混合）的过程，对于提高Reduce的效率以及减小数据传输的压力有很大的帮助。后面会具体提及这些部分的细节。

HDFS是分布式计算的存储基石，Hadoop的分布式文件系统和其他分布式文件系统有很多类似的特质。分布式文件系统基本的几个特点：

对于整个集群有单一的命名空间。

数据一致性。适合一次写入多次读取的模型，客户端在文件没有被成功创建之前无法看到文件存在。

文件会被分割成多个文件块，每个文件块被分配存储到数据节点上，而且根据配置会由复制文件块来保证数据的安全性。

上图中展现了整个HDFS三个重要角色：NameNode、DataNode和Client。NameNode可以看作是分布式文件系统中的管理者，主要负责管理文件系统的命名空间、集群配置信息和存储块的复制等。NameNode会将文件系统的Meta-data存储在内存中，这些信息主要包括了文件信息、每一个文件对应的文件块的信息和每一个文件块在DataNode的信息等。DataNode是文件存储的基本单元，它将Block存储在本地文件系统中，保存了Block的Meta-data，同时周期性地将所有存在的Block信息发送给NameNode。Client就是需要获取分布式文件系统文件的应用程序。这里通过三个操作来说明他们之间的交互关系。

文件写入：

Client向NameNode发起文件写入的请求。

NameNode根据文件大小和文件块配置情况，返回给Client它所管理部分DataNode的信息。

Client将文件划分为多个Block，根据DataNode的地址信息，按顺序写入到每一个DataNode块中。

文件读取：

Client向NameNode发起文件读取的请求。

NameNode返回文件存储的DataNode的信息。

Client读取文件信息。

文件Block复制：

NameNode发现部分文件的Block不符合最小复制数或者部分DataNode失效。

通知DataNode相互复制Block。

DataNode开始直接相互复制。

最后再说一下HDFS的几个设计特点（对于框架设计值得借鉴）：

Block的放置：默认不配置。一个Block会有三份备份，一份放在NameNode指定的DataNode，另一份放在与指定DataNode非同一Rack上的DataNode，最后一份放在与指定DataNode同一Rack上的DataNode上。备份无非就是为了数据安全，考虑同一Rack的失败情况以及不同Rack之间数据拷贝性能问题就采用这种配置方式。

心跳检测DataNode的健康状况，如果发现问题就采取数据备份的方式来保证数据的安全性。

数据复制（场景为DataNode失败、需要平衡DataNode的存储利用率和需要平衡DataNode数据交互压力等情况）：这里先说一下，使用HDFS的balancer命令，可以配置一个Threshold来平衡每一个DataNode磁盘利用率。例如设置了Threshold为10%，那么执行balancer命令的时候，首先统计所有DataNode的磁盘利用率的均值，然后判断如果某一个DataNode的磁盘利用率超过这个均值Threshold以上，那么将会把这个DataNode的block转移到磁盘利用率低的DataNode，这对于新节点的加入来说十分有用。

数据交验：采用CRC32作数据交验。在文件Block写入的时候除了写入数据还会写入交验信息，在读取的时候需要交验后再读入。

NameNode是单点：如果失败的话，任务处理信息将会纪录在本地文件系统和远端的文件系统中。

数据管道性的写入：当客户端要写入文件到DataNode上，首先客户端读取一个Block然后写到第一个DataNode上，然后由第一个DataNode传递到备份的DataNode上，一直到所有需要写入这个Block的NataNode都成功写入，客户端才会继续开始写下一个Block。

安全模式：在分布式文件系统启动的时候，开始的时候会有安全模式，当分布式文件系统处于安全模式的情况下，文件系统中的内容不允许修改也不允许删除，直到安全模式结束。安全模式主要是为了系统启动的时候检查各个DataNode上数据块的有效性，同时根据策略必要的复制或者删除部分数据块。运行期通过命令也可以进入安全模式。在实践过程中，系统启动的时候去修改和删除文件也会有安全模式不允许修改的出错提示，只需要等待一会儿即可。

下面综合MapReduce和HDFS来看Hadoop的结构：

在Hadoop的系统中，会有一台Master，主要负责NameNode的工作以及JobTracker的工作。JobTracker的主要职责就是启动、跟踪和调度各个Slave的任务执行。还会有多台Slave，每一台Slave通常具有DataNode的功能并负责TaskTracker的工作。TaskTracker根据应用要求来结合本地数据执行Map任务以及Reduce任务。

说到这里，就要提到分布式计算最重要的一个设计点：Moving Computation is Cheaper than Moving Data。就是在分布式处理中，移动数据的代价总是高于转移计算的代价。简单来说就是分而治之的工作，需要将数据也分而存储，本地任务处理本地数据然后归总，这样才会保证分布式计算的高效性。

为什么要选择Hadoop?

说完了What，简单地说一下Why。官方网站已经给了很多的说明，这里就大致说一下其优点及使用的场景（没有不好的工具，只用不适用的工具，因此选择好场景才能够真正发挥分布式计算的作用）：

可扩展：不论是存储的可扩展还是计算的可扩展都是Hadoop的设计根本。

经济：框架可以运行在任何普通的PC上。

可靠：分布式文件系统的备份恢复机制以及MapReduce的任务监控保证了分布式处理的可靠性。

高效：分布式文件系统的高效数据交互实现以及MapReduce结合Local Data处理的模式，为高效处理海量的信息作了基础准备。

使用场景：个人觉得最适合的就是海量数据的分析，其实Google最早提出MapReduce也就是为了海量数据分析。同时HDFS最早是为了搜索引擎实现而开发的，后来才被用于分布式计算框架中。海量数据被分割于多个节点，然后由每一个节点并行计算，将得出的结果归并到输出。同时第一阶段的输出又可以作为下一阶段计算的输入，因此可以想象到一个树状结构的分布式计算图，在不同阶段都有不同产出，同时并行和串行结合的计算也可以很好地在分布式集群的资源下得以高效的处理。

文档型数据库设计模式-如何存储树形数据

作者：nosqlfan on 星期二, 三月 8, 2011 · 8条评论 【阅读：1,299 次】

在数据库中存储树形结构的数据，这是一个非常普遍的需求，典型的比如论坛系统的版块关系。在传统的关系型数据库中，就已经产生了各种解决方案。

此文以存储树形结构数据为需求，分别描述了利用关系型数据库和文档型数据库作为存储的几种设计模式。

A.关系型数据库设计模式1

id name parent_id

1 A NULL

2 B 1

3 C 1

4 D 2

上图表示了传统的设计方法之一，就是将树形结构的每一个结点作为关系型数据库中的一行进行存储，每一个结点保存一个其父结点的指针。

- **优点**：结构简单易懂，插入修改操作都很简单
- **缺点**：如果要获取某个结点的所有子结点，将是一件很恶心的事

B.关系型数据库设计模式2

id name parent_id left right

1 A NULL 1 8

2 B 1 2 5

3 C 1 6 7

```
4 D    2        3  4
```

上图在模式1的基础上多了两列，left和right，相当于btree中的左右分支，分别存储了左右分支结点的最大值和最小值。

- **优点**：要查找一个结点的子结点很容易，只需要做一个范围查询就行了（比如B节点的子结点，只需要查询id >=2 && id<=5）
- **缺点**：由于树结构存在在这里面了，所以添加或修改已存在结点将可能产生连锁反应，操作过于复杂

C.文档型数据库设计模式1

```
{
  "name": "A",
  "children": [
    {"name": "B", "children": [{"name": "D"}]},
    {"name": "C"}
  ]
}
```

将整个树结构存成一个文档，文档结构既树型结构，简明易懂。

- **优点**：简明易懂
- **缺点**：文档会越来越大，对所有结点的修改都集中到这一个文档中，并发操作受限

D.文档型数据库设计模式2

```
{"_id": "A", "children": ["B", "C"]}
{"_id": "B", "children": ["D"]}
{"_id": "C"}
{"_id": "D"}
```

将每个结点的所有子结点存起来

- **优点**：结构简单，查找子结点方便
- **缺点**：查找父结点会比较麻烦

E.文档型数据库设计模式3

```
{
  "leaf": "A",
  "children": [
    {"leaf": "B", "children": [{"leaf": "D"}] },
    {"leaf": "C"}
  ]
}
{"_id": "A", ...}
{"_id": "B", ...}
{"_id": "C", ...}
{"_id": "D", ...}
```

充分利用文档型存储schema-less的优点，先利用上面C方案存存储一个大的树形文档，再将每一个结点的其他信息单独存储。

- **优点**：操作方便，结构上的操作可以直接操作大的树形文档，数据上的操作也只需要操作单条数据
- **缺点**：对所有结点的修改都集中到这一个文档中，并发操作受限

10gen工程师讲数据库索引实现

MongoDB尽管在数据存储上与传统关系数据库很不一样，但是在索引构建上却几乎是与传统关系型数据库一致。主要是使用了B-tree作为索引结构。

Indexes in MongoDBare conceptually similar to those in RDBMSes like MySQL. You will want an indexin MongoDB in the same sort of situations where you would have wanted an indexin MySQL.

下面一篇文章是10gen工程师 KyleBanker 所写，他拿食谱举例，讲解了在数据库系统中索引的基本实现，非常形像。本站简单翻译如下，更详细的内容，请直接查看原文：

原文链接：TheJoy Of Indexing

1.唯一索引

想像你要在一本没有目录的食谱上查找某一道菜的做法，那么你唯一的办法可能就是从头到尾把这本几百页的书看一遍，直到找到你想找的菜。

而一个快速的方法就是给菜谱加上一个目录，目录中有每一道菜的名字与相应页数的对应，并且菜名是按字母顺序排列的如下，这样你就可以按首字母快速地找到你要找的菜谱的页数了。（这个基本上对应的是唯一索引的形式）

举例：

水饺
- 45
水煮肉片
- 4,011
水煮鱼
- 9432.非唯一索引

但是如果我今天买了条鱼，想查一下所有跟鱼相关的做法，那么用上面的索引就没办法了。于是出现下面一种索引构建方法，即将所有菜的材料构建目录，这时候一个材料对应多道菜。

举例：

鱼
- 3, 20, 42, 88, 103, 1,215…
猪肉
- 2, 47, 88, 89, 90, 275…
白菜
- 7, 9, 80, 81, 82, 83, 84…

3.联合索引

下面还有另一种需求，如果我知道材料里有鱼，而我又知道这道菜的名字，就不用查找到材料后去翻看后面几个具体页的菜再来选择了。满足这种需求的是联合索引：

鱼
- 水煮鱼
— 1,215
- 红烧带鱼
— 88
- 酸菜鱼
— 103
猪肉
- 红烧肉
— 875

- 小炒肉

—— 89

- 回锅肉

—— 47

白菜

- 白菜汤

—— 2,000

- 素炒白菜

—— 2,133

- 白菜炖豆腐

—— 1,050

上一篇：轻量级内存数据库研究

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场