

SequoiaDB 信息中心

内容

SequoiaDB 数据库概述.....	5
主要特点.....	5
数据模型.....	6
系统架构.....	7
数据库概念.....	11
数据库.....	11
文档.....	11
集合.....	14
集合空间.....	15
数据库服务器.....	15
索引.....	16
聚集.....	17
事务.....	18
最终一致性策略.....	18
实例.....	19
读写分离.....	19
后台任务.....	19
大对象.....	20
集群.....	20
运行模式.....	21
节点.....	21
分区组.....	27
数据分区.....	30
域.....	36
安装指南.....	37
规划数据库部署.....	37
最简部署.....	37
高可用部署.....	38
高性能部署.....	38
SequoiaDB 产品安装的系统要求.....	39
硬件要求.....	39
受支持的操作系统.....	40
软件要求.....	40
准备安装介质.....	42
SequoiaDB 服务器安装部署.....	42
自动化安装部署.....	42
手工安装部署.....	95
SequoiaDB Web 监控.....	99
测试环境.....	110
卸载.....	111
升级.....	111
数据库管理.....	113
数据库管理.....	113
数据库配置.....	113
监控.....	115
引擎调度单元.....	140
日志.....	141
数据库工具.....	142
集群管理.....	154
集群中新增主机.....	154

编目分区组管理.....	155
数据分区组管理.....	156
新增协调节点.....	157
案例.....	157
运维.....	160
集群启停.....	160
备份恢复.....	161
故障恢复.....	168
监控.....	169
系统安全.....	170
Hadoop 集成.....	171
SequoiaDB 与 Hadoop 部署.....	171
与 MapReduce 集成.....	171
与 Hive 集成.....	174
SequoiaDB 支持的 Hive 版本列表.....	174
配置方法.....	175
使用方法.....	175
与 Pig 集成.....	176
开发指南.....	177
SequoiaDB shell.....	177
SequoiaDB shell 入门.....	177
使用 shell 的窍门.....	178
SequoiaDB shell 中的基本操作.....	180
创建.....	180
读取.....	182
更新.....	184
删除.....	185
SequoiaDB 应用程序开发.....	186
C 驱动.....	186
C++ 驱动.....	193
Java 驱动.....	200
PHP 驱动.....	208
C# 驱动.....	212
Python 驱动.....	217
C BSON 简介.....	222
C++ BSON 简介.....	224
C BSON API.....	225
C++ BSON API.....	225
数据库集群控制器.....	225
参考手册.....	227
SequoiaDB JavaScript 方法.....	227
Global.....	229
Sdb.....	230
SdbCS.....	253
SdbCollection.....	256
SdbCursor.....	269
SdbReplicaGroup.....	274
SdbNode.....	278
SdbDomain.....	280
Oma.....	281
操作符.....	284
匹配符.....	286
更新符.....	295
聚集符.....	300
SQL 语法.....	308

sql create collectionspace.....	309
sql drop collectionspace.....	309
sql create collection.....	310
sql drop collection.....	310
sql create index.....	310
sql drop index.....	311
sql list collectionspaces.....	311
sql list collections.....	312
sql insert into.....	312
sql select.....	312
sql update.....	313
sql delete.....	314
sql group by.....	314
sql order by.....	315
sql split by.....	315
sql limit.....	315
sql offset.....	316
sql as.....	316
sql inner join.....	316
sql left outer join.....	317
sql right outer join.....	317
sql sum().....	318
sql count().....	318
sql avg().....	318
sql max().....	319
sql min().....	319
sql first().....	319
sql last().....	320
sql push().....	320
sql addtoset().....	321
sql buildobj().....	321
sql mergearrayset().....	321
SQL to SequoiaDB 映射表.....	322
限制.....	323
Error Code List.....	325

SequoiaDB 数据库概述

SequoiaDB 数据库是一款新型企业级分布式非关系型数据库，帮助企业用户降低 IT 成本，并对大数据的存储与分析提供了一个坚实，可靠，高效与灵活的底层平台。

优势

- 通过非结构化存储与分布式处理，提供了近线性的水平扩张能力，让底层的存储不再成为瓶颈
- 提供了精确到分区级别的高可用性，预防服务器，机房故障以及人为错误，让数据24x7永远在线
- 提供了完善的企业级功能，让用户轻松管理高并发性任务，以及海量数据分析
- 增强的非关系型数据模型，帮助企业快速开发和部署应用程序，做到应用程序的随需应变
- 提供了最终一致性的保障，从根本上杜绝数据缺失
- 提供了在线应用与大数据分析的后台数据库的结合，通过读写分离机制做到同系统中数据分析与在线业务互不干扰

主要特点

随着企业中日益复杂与多变的需求，以及迅速扩展并带来海量数据的业务，IT 部门需要将越来越多的信息提供给其用户。同时在如今的经济环境下，IT 部门还需要在提供高效服务的同时，降低其设备与程序维护所带来的开销。

SequoiaDB 数据库，提供了基于 PC 服务器的大规模集群数据平台，让 IT 部门在提供稳定，可靠以及高效数据服务的同时，大大降低 IT 部门应用程序的开发，部署以及维护成本。

通过部署并使用 SequoiaDB 数据库，用户可以得到：

近线性的水平扩张能力

传统关系型数据库无法做到的水平扩张能力在 SequoiaDB 中得到了完美的解决。通过对数据进行垂直切分，以及应用了新型的非关系型数据模型，SequoiaDB 有效地降低了传统数据库分区中大量数据交换的瓶颈，得到了线性水平扩张的能力。

永不停机的高可用性

SequoiaDB 可以将用户的每一份数据实时保存多份副本，可以有效地防止服务器，机房以及人为的因素所造成的系统停机，保证数据24x7随时在线可用。

完善的企业级支持

SequoiaDB 为企业提供了用户友好并完善的管理，维护以及监控界面，并提供了24x7的电话及现场技术支持。

增强的非关系型模型

SequoiaDB 使用 JSON 数据模型，灵活有效地降低关系模型维护的复杂性，让数据库更加贴近应用程序，从而大大降低应用程序的开发和维护成本。

最终一致性的保障

SequoiaDB 在大规模分布式环境中提供了数据最终一致性的保障，满足用户对实时性与一致性的需求。

在线应用与数据分析的结合

通过分区机制进行读写分离，允许前端在线应用与后台数据分析完美并行互不干扰，并结合 Hadoop 技术进行海量数据分析。

SequoiaDB 作为全球第一家企业级文档式非关系型数据库，致力于为用户提供一个高可扩展性，高可用性，高性能，以及易维护的分布式数据平台。满足用户对大数据，实时分析，以及低成本的需求。

数据模型

SequoiaDB 数据库使用 JSON 数据模型，而非传统的关系型数据模型。

JSON 数据结构的全称为 JavaScript Object Notation，是一种轻量级的数据交换格式，非常易于人阅读和编写，同时也易于机器生成和解析。

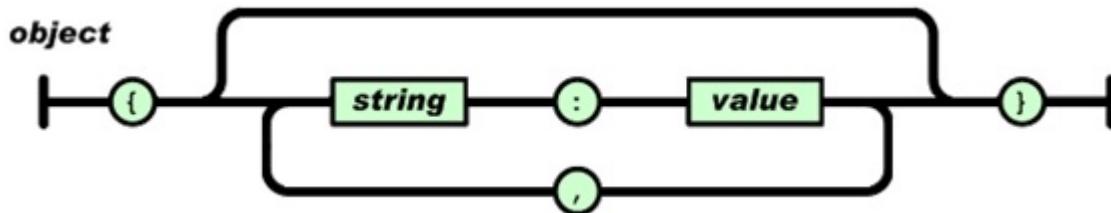
它基于 JavaScript Programming Language, Standard ECMA-262 3rd Edition – December 1999 的一个子集，为纯文本格式，支持嵌套结构与数组。

JSON 建构基于两种结构：

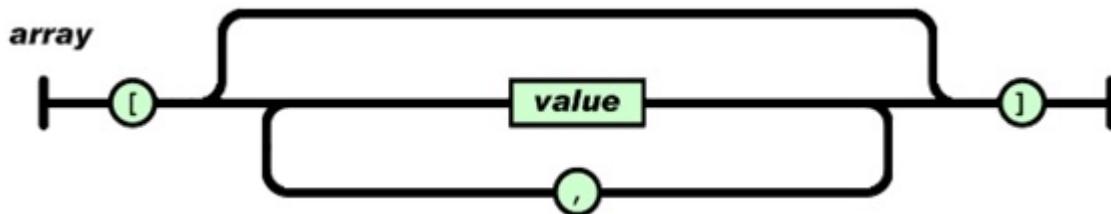
- 键值对集合 —— 在键值对集合结构中，每一个数据元素拥有一个名称与一个数值。数值可以包含数字，字符串等常用结构，或嵌套 JSON 对象和数组。
- 数组 —— 在数组中的每一个元素不包含元素名，其值可以为数字，字符串等常用结构，或者嵌套 JSON 对象和数组。

JSON 具有如下形式：

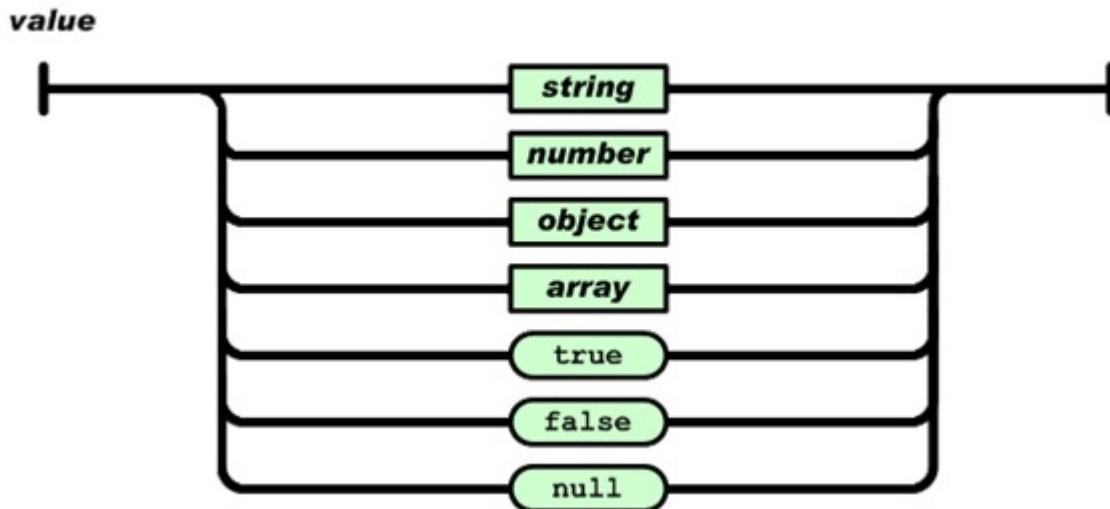
- 对象是一个无序的“键值对”集合，以“{”（左大括号）开始，“}”（右大括号）结束。每一个元素名后跟一个“：“（冒号）；而元素之间使用“,”（逗号）分隔；



- 数组是值的有序集合，以“[”（左中括号）开始，“]”（右中括号）结束。值之间使用“,”（逗号）分隔；



- 值可以为由双引号包裹的字符串，数值，对象，数组，true，false，null，以及 SequoiaDB 数据库特有的数据结构（例如日期，时间等）组成。

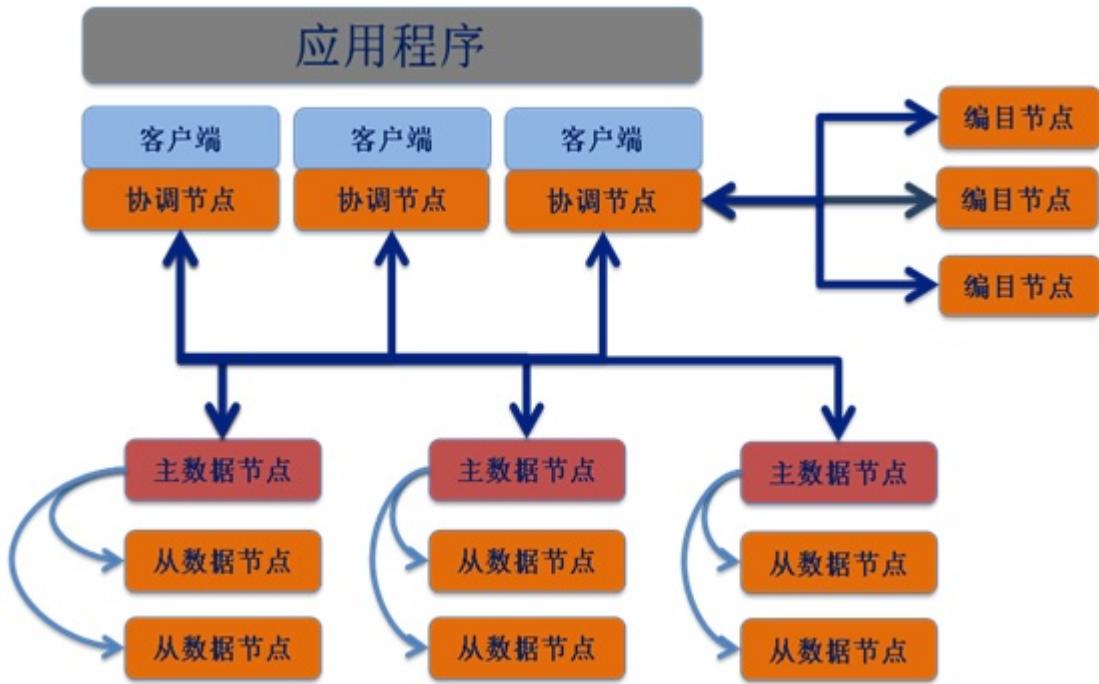


一个典型的嵌套式数据结构如下：



系统架构

SequoiaDB 使用分布式架构，下图提供了对 SequoiaDB 体系结构的一般概述。



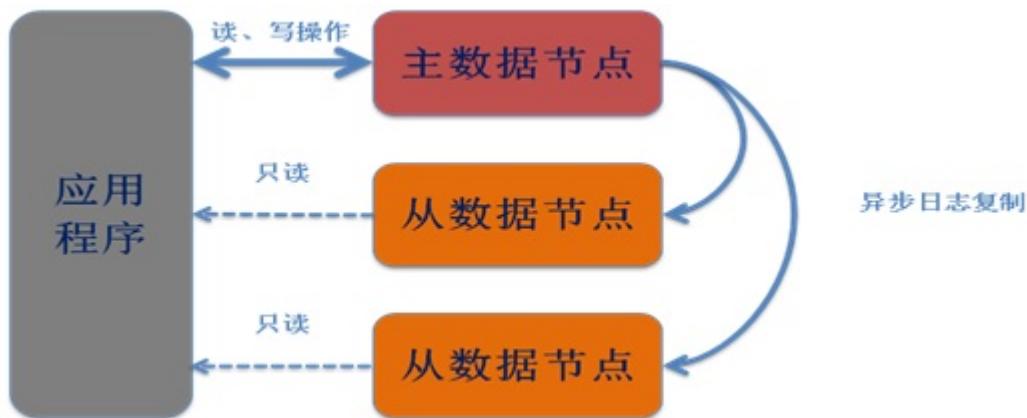
在客户机端（或应用程序端），本地或/和远程应用程序都与 SequoiaDB 客户机库链接。本地与远程客户机使用 TCP/IP 协议与协调节点进行通讯。

协调节点不保存任何用户数据，仅作为请求分发节点将用户请求分发至相应的数据节点。

编目节点保存系统的元数据信息，协调节点通过与编目节点通讯从而了解数据在数据节点中的实际分布。一个或多个编目节点可组成复制组集群。

数据节点保存用户的数据信息。一个或多个数据节点可以构成一个复制组（又称分区组）。复制组中每个数据节点都存储该复制组的一份完整数据，又称为复制组实例（或分区组实例）；复制组中的数据节点之间采用最终一致性同步数据，不同的复制组中保存的数据无重复。

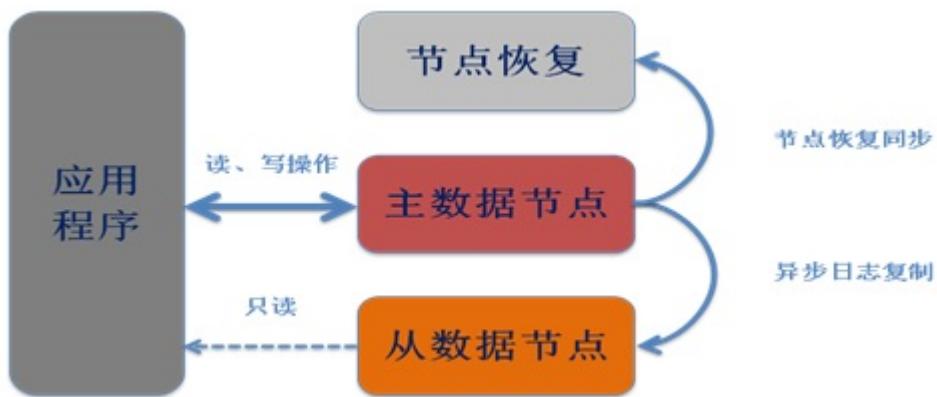
每个复制组中可以包含一个或多个数据节点。当存在多个数据节点时，节点间数据进行异步复制。复制组中可以存在最多一个主节点与若干从节点。其中主节点可以进行读写操作，从节点进行只读操作。



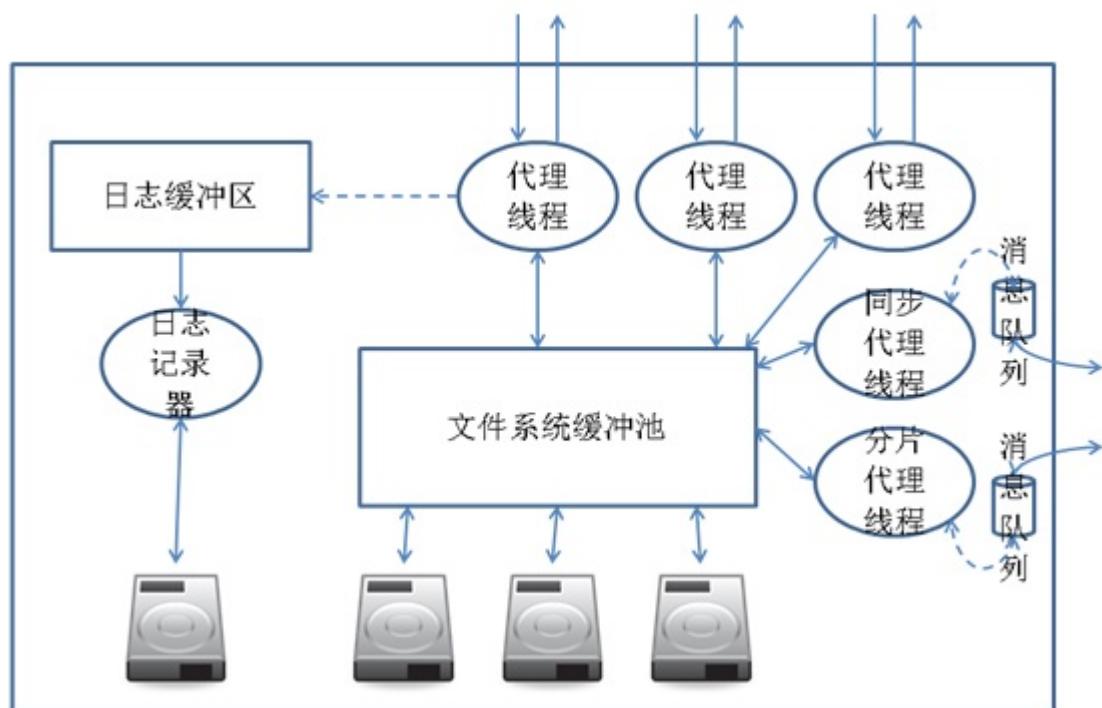
从节点离线不影响主节点的正常工作。主节点离线后会在从节点中自动选举出新的主节点处理写请求。



节点恢复后，或新的节点加入复制组后会进行自动同步，保障数据在同步完成时与主节点一致。



在单个数据节点中的体系结构如下：



在数据节点，活动由引擎可调度单元（EDU）控制。每一个节点为操作系统中的一个进程。每个 EDU 在节点中为一个线程。对于外部用户请求其处理线程为代理线程，对于集群内部请求则由同步代理线程处理分区内同步事件；或分区代理线程处理分区间同步事件。

所有对数据的写操作均会记录入日志缓冲区，通过日志记录器将其异步写入磁盘。

用户数据会由代理线程直接写入文件系统缓冲池，然后由操作系统将其异步写入底层磁盘。

数据库概念

数据库概念相关内容

数据库

数据库的主要对象包括文档，集合，集合空间与索引等。

文档

概念

SequoiaDB 中的文档为 JSON 格式，一般又被称为记录。在数据库内部使用 BSON，即二进制的方式存放 JSON 数据。

一般来说，一条文档由一个或多个字段构成，每个字段分为键值与数值两个部分，如下为包含两个字段的文档：

```
{ "姓名": "张三", "性别": "男" }
```



注:

BSON 文档可能有多个同名的字段，但是，大多数 SequoiaDB 接口不支持重复的字段名，如果需要操作的文档有多个同名的字段，请参阅驱动程序了解更多信息。

SequoiaDB 内部程序创建的一些文档可能含有重名的字段，但是不会向现有的用户文档添加重名的键。

字段类型

每个字段的键值为字符串，而数值则可以为数字，字符串，嵌套 JSON，嵌套数组等对象。SequoiaDB 所支持的数值类型见下表：

数值类型	定义	用例
整数	整数，范围 -2147483648 至 2147483647	{ "key": 123 }
长整数	整数，范围 -9223372036854775808 至 9223372036854775807 如果用户指定的数值无法适用于整数，则 SequoiaDB 自动将其转化为长整数	{ "key": 3000000000 }
浮点数	浮点数，范围 1.7E-308 至 1.7E+308	{ "key": 123.456 } 或 { "key": 123e+50 }
字符串	双引号包含的字符串	{ "key": "value" }
对象 ID (OID)	十二字节对象 ID	{ "key": { "\$oid": "123abcd00ef12358902300ef" } }
布尔	true 或者 false	{ "key": true } 或 { "key": false }
日期	YYYY-MM-DD 的日期形式	{ "key": { "\$date": "2012-01-01" } }
时间戳	YYYY-MM-DD-HH.mm.ss.fffffff 的形式存取	{ "key": { "\$timestamp": "2012-01-01-13.14.26.124233" } }
二进制数据	Base64 形式的二进制数据	{ "key": { "\$binary": "aGVsbG8gd29ybGQ=", "\$type": "1" } }
正则表达式	正则表达式	{ "key": { "\$regex": "^张", "\$options": "i" } }
对象	嵌套 JSON 文档对象	{ "key": { "subobj": "value" } }
数组	嵌套数组对象	{ "key": ["abc", 0, "def"] }
空	null	{ "key": null }

字段顺序

文档中的各字段无排列顺序，在进行数据操作时字段之间的顺序可能会被调换。

当表示嵌套对象中的某一个字段时，可以使用“.”（句号）在字段名之间进行分割。例如给定数据：

```
{ "姓名" : "张三", "地址" : { "街道" : "水蓝街", "城市" : "xx", "省份" : "yy" } }
```

用户可以使用“地址.城市”字段名表示地址子对象中的城市字段。

其他

- 每个文档的最大尺寸为16MB
- 文档中必须包括“_id”字段，如果用户没有提供该字段，系统会自动生成一个对象 ID 类型的字段
- “_id”字段在集合内唯一
- 文档的字段名不可以“\$”字符起始
- 文档的字段名不可以包含“.”字符

数组

概念

SequoiaDB 中的文档为 JSON 格式，一般又被称为记录。

格式

当记录中的某一字段对应多个数值时，用户可以使用数组结构存放数据。数组由“[”（左中括号）开始，至“]”（右中括号）结束，其中包含零个或多个数值。

```
{ 字段名 : [ <数值1>, <数值2>, <数值3> ... ] }
```

示例

数组可以存放完全不相同的数据类型，其中每个记录以从0起始的下标表示。例如：

```
{ "key" : [ "hello", "world" ] }
```

其中“hello”在数组中的下标为0，而“world”在数组中的下标为1。数组之中的数值有序，在进行数据操作时数组中的数值顺序不会改变。表示数组中某个元素时，可以使用“字段名.下标”的方式。

例如：如果希望表示 key 中“world”所在的数值，可以使用“key.1”作为字段名。

对象 ID

概念

对象 ID 为一个12字节的 BSON 数据类型，包括如下内容：

- 4字节精确到秒的时间戳
- 3字节系统（物理机）标示
- 2字节进程 ID
- 3字节由随机数起始的序列号

4字节时间戳	3字节系统标示	2字节进程ID	3字节序列号

该对象 ID 可以在集群环境中，对每台系统中的每个进程，每秒钟标示16777216个不同的数值，因此基本可以认为在集群环境中全局唯一。

在 SequoiaDB 中，每个集合中存放的文档必须拥有一个 _id 字段，并且该字段在集合中唯一。

格式

对象 ID 的表达形式如下：

```
{ "$oid" : "<24字节16进制字符串>" }
```

示例

对象 ID 的显示结果如下：

```
{ "key" : { "$oid" : "5156c192f970aed30c020000" } }
```

日期

概念

SequoiaDB 中的日期使用 YYYY-MM-DD 的形式存取，在存储时将其转换为4字节的整数。

格式

日期的表达形式如下：

```
{ "$date" : "<YYYY-MM-DD>" }
```

示例

例如：

```
{ "createTime" : { "$date" : "2012-05-12" } }
```

时间戳

概念

SequoiaDB 中的时间戳使用 YYYY-MM-DD-HH.mm.ss.fffffff 的形式存取，在存储时将其转换为8字节的整数。

格式

时间戳的表达形式如下：

```
{ "$timestamp" : "<YYYY-MM-DD-HH.mm.ss.fffffff>" }
```

示例

例如：

```
{ "createTime" : { "$timestamp" : "2012-05-12-13.15.21.241523" } }
```

二进制数据

概念

在 SequoiaDB 中的数据使用 JSON 形式访问，因此对于二进制的数据需要用户使用 Base64 方式进行编码，之后以字符串的形式发送至数据库。

格式

二进制数据的表达形式如下：

```
{ "$binary" : "<数据>", "$type" : "<类型>" }
```

其中“数据”必须为 Base64 编码的数据，“类型”为0-255之间的十进制数值，用户可以任意指定该范围之间的类型作为应用程序中的类型标示。

Base64 为一种通用的数据转换形式，主要将二进制数据转化为以纯 ASCII 字符串表示的字节流。一般来说转换之后的数据长度会大于原本的数据长度。

为了节省空间，在 SequoiaDB 的内部存放数据时，会将 Base64 编码后的数据解码为原始数据进行存放。当用户读取数据时会再次将其转化为 Base64 形式发送。

示例

例如：字符串“hello world”被 Base64 编码后的数据为“aGVsbG8gd29ybGQ=”。包含“hello world”二进制数据，且类型为1的 JSON 数据为：

```
{ "key" : { "$binary" : "aGVsbG8gd29ybGQ=", "$type" : "1" } }
```

正则表达式

概念

SequoiaDB 可以使用正则表达式检索用户数据。

格式

正则表达式输入的格式如下：

```
{ "$regex" : "正则表达式", "$options" : "选项" }
```

其中“正则表达式”为一个正则表达式字符串，“选项”则参见下表：

选项	描述
i	匹配时不区分大小写。
m	允许进行多行匹配；当该参数打开时，字符“^”与“&”匹配换行符的之后与之前字符。
x	忽略正则表达式匹配中的空白字符；如果需要使用空白字符，在空白字符之前使用反斜线“\W”进行转意。
s	允许“.”字符匹配换行符。

当使用选项时，用户可以使用任意组合指定其中的选项。

示例

使用正则表达式进行大小写忽略，匹配以字符“W”起始的字符串，可以使用：

```
{ "key" : { "$regex" : "^\W", "$options" : "i" } }
```

关于正则表达式规则，请参阅 [Perl 正则表达式手册](#)。

集合

概念

集合（Collection）是数据库中存放文档的逻辑对象。任何一条文档必须属于一个且仅一个集合。

集合由“<集合空间名>.<集合名>”构成。集合名最大长度127字节，为 UTF-8 编码。一个集合中可以包含零个至任意多个文档（上限为集合空间大小上限）。

在集群环境下，每个集合还可以拥有除名称外的以下属性：

属性名	描述
分区键（ShardingKey）	指定集合的分区键，集合中所有的文档将分区键中指定的字段作为分区信息，文档分别存放在所对应的分区中。
分区类型（ShardingType）	指定集合的分区类型：范围分区（Range）或散列分区（Hash）。
写副本数（ReplSize）	指定该集合默认的写副本数。如果该值 ≤ 1，数据的写请求在一个副本写入成功后就会返回。如果该值 > 1，则需要等到至少指定数量的副本被成功写入数据后才会返回。

属性名	描述
数据压缩 (Compressed)	创建集合时，指定 Compressed 属性的值代表着在做插入操作时，是否以压缩的形式存储数据，它的值有 true 和 false，默認為 false。

集合空间

概念

集合空间 (Collection Space) 是数据库中存放集合的物理对象。任何一个集合必须属于一个且仅一个集合空间。

集合空间名最大长度127字节，为 UTF-8 编码。一个集合空间中可以包含不超过4096个集合；每个数据节点可以包含不超过4096个集合空间。

每一个集合空间在数据节点均对应一个文件，文件名格式为“<集合空间名>.1”。

数据页

集合空间将文件划分为若干个固定大小的数据页 (Page)，在创建集合空间时用户可以指定数据页大小，且创建后不可更改。

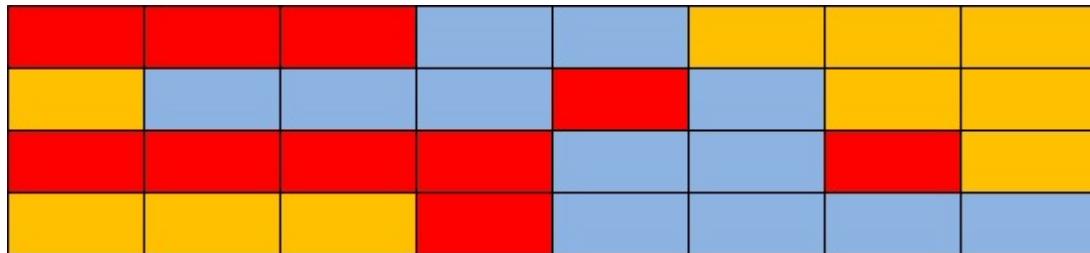
每个数据节点中，单个集合空间可以访问16777216个数据页。因此对应不同数据页大小，单分区中集合空间容量上限为：

数据页大小 (字节)	集合空间最大容量 (GB)
4096	512
8192	1024
16384	2048
32768	4096
65536	8192

数据块

一个或多个数据页组成数据块。集合空间中的每个集合由零个或以上的数据块构成，每个数据块的大小由用户数据长度自动调整。集合中的文档不可跨多个数据块存放。

集合空间中的数据块存放方式如下图所示：



图中显示了一个集合空间中的三个集合，分别用不同的颜色代表。每个集合所对应的数据存放在各自的数据页中。一个或多个数据页可以组成一个数据块，每个数据块中的数据连续，且文档不能跨越多个数据块。

数据库服务器

概念

数据库服务器提供软件服务以便安全、高效地管理信息。SequoiaDB 是文档型非关系型数据库服务器。

数据库服务器是指安装了 SequoiaDB 数据库引擎的计算机。SequoiaDB 引擎为数据存取操作的基本单元，在分布式架构中，每一个数据库作为一个节点 (Node) 存在，节点之间的数据无共享。

在一台计算机中，每一个 SequoiaDB 数据库引擎对应一个数据库路径，该数据库中所有的集合空间均放置在该目录中。

数据库路径包含一个或多个集合空间。对于数据节点和协调节点至少存在一个 SYSTEMP 系统临时集合空间；而对于编目节点至少存在 SYSTEMP 系统临时集合空间、SYSAUTH 系统权限集合空间、SYSPROCEDURES 系统存储过程集合空间与 SYSCAT 系统编目集合空间。

每个数据库引擎可以包含最多4096个集合空间。

索引

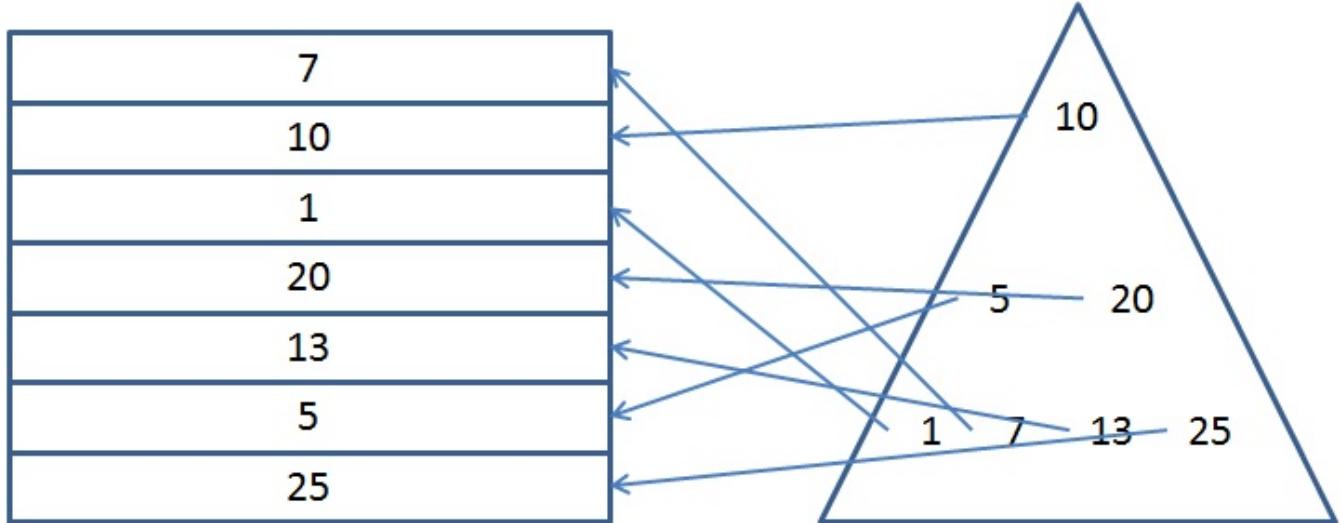
概念

在 SequoiaDB 数据库中，索引是一种特殊的数据对象。索引本身不做为保存用户数据的容器，而是作为一种特殊的元数据，提高数据访问的效率。

每一个索引必须建立在一个集合之中，一个集合最多可以拥有64个索引。

索引可以被认为是将数据按照某一个或多个给定的字段进行排序，从而在其中快速搜索到用户指定查询条件的方式。在 SequoiaDB 中，索引使用 B 树结构。

一个典型的索引结构如下图所示：



图中左边的方形为数据，右边的三角形为索引。索引按照从小到大的树形结构排列，每条索引记录分别指向一条记录文档。

通过进行树形遍历，对于查找某个特定数值的操作，可以使用树遍历在索引中快速定位其所需的数据。

SequoiaDB 可以对任意数据类型进行索引，每一个索引包含几个属性：

属性	描述
name	索引名，同一个集合中的索引名必须唯一。
key	索引键，为一个 JSON 结构，包含一个或多个指定索引字段与方向的字段。其中方向为1代表从小到大排序，-1则为从大到小排序。
unique	索引是否唯一，可选参数，默认 false。设置为 true 时代表该索引为唯一索引。在唯一索引所指定的索引键字段上，集合中不可存在一条以上的记录完全重复。
enforced	索引是否强制唯一，可选参数，在 unique 为 true 时生效，默认 false。设置为 true 时代表该索引在 unique 为 true 的前提下，不可存在一个以上完全空的索引键。

在 SequoiaDB 中，所有集合均包含一个名为“\$id”的强制唯一索引。该索引包含一个“_id”字段的索引键。

所有的分区集合在创建时均会自动生成一个额外的“\$shard”索引，索引键为用户指定的分区键字段。



注：

在分区集合中，所有的唯一索引必须包含集合分区键中所指定的全部字段。

在分区集合中，“\$id”索引仅保证单节点内记录的唯一性。如果用户希望指定全局唯一的字段，需要额外创建唯一索引，且该索引必须包含集合分区键中所指定的全部字段。

格式

索引的定义格式必须包含 name 与 key 两个字段。其中 name 的值必须为字符串，key 则为一个 JSON 对象。

```
{ "name" : "<索引名>", "key" : " { <索引字段1> : <1|-1>, [ <索引字段2> : <1|-1> ... ] },  
[ "unique" : <true|false> ], [ "enforced" : <true|false> ] }
```

key 的对象必须包含至少一个字段，其中字段名为用户需要索引的字段名，数值为1或者-1。其中1代表数据在索引中的排列顺序由小至大，-1则代表由大至小。

示例

- 非唯一索引，索引名“employee_id_key”，索引字段为正向“employee_id”

```
{ "name" : "employee_id_key", "key" : { "employee_id" : 1 } }
```

- 唯一索引，索引名为“record_id_index”，索引字段为正向“product_key”与逆向“record_key”

```
{ "name" : "record_id_index", "key" : { "product_key" : 1, "record_key" : -1 }, "unique" : true }
```

在该索引中，不可存在两条记录拥有同样的 product_key 与 record_key（如果仅 product_key 相同，或者仅 record_key 相同则可以通过唯一判定）

- 强制唯一索引，索引名为“测试索引”，索引字段为正向“测试用例名称”

```
{ "name" : "测试索引", "key" : { "测试用例名称" : 1 }, "unique" : true, "enforced" : true }
```

在强制唯一索引中，所有记录必须遵循唯一索引规则，且不可存在一条以上的数据在“测试用例名称”字段为空。

聚集

聚集框架提供了对集合中的原始数据记录进行统计计算的能力。通过使用聚集框架，用户能够直接从集合中提取数据记录并获取所需的统计结果。聚集框架提供的操作接口类似于集合中的查询操作，不同的是聚集框架还提供了一系列函数及操作对查询结果进行处理。

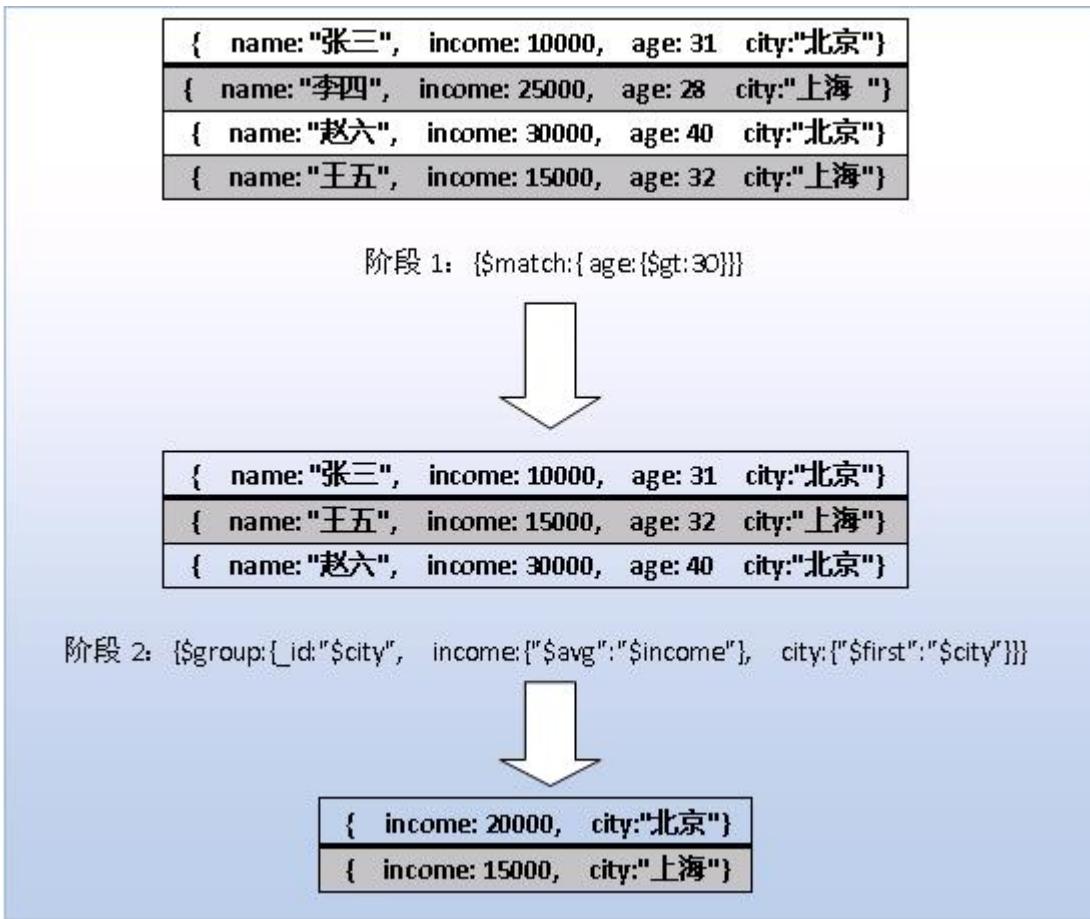
[更多聚集符操作请点击这里](#)

aggregate()

以下是聚集操作举例：

```
db.space.cl.aggregate(  
    {$match: { age: {$gt: 30}}},  
    {$group: {_id: "$city", income: {"$avg": "$income"}, city: {"$first": "$city"}}}  
)
```

上例聚集操作包含了两个子操作，其中“\$match”子操作将集合中年龄大于30的数据记录筛选出来；“\$group”操作从筛选出的数据记录按照城市进行分组，计算出每个城市的人均收入。通过上例聚集操作将得到各城市30岁以上的人均收入。



事务

事务是由一系列操作组成的逻辑工作单元。在同一个会话（或连接）中，同一时刻只允许存在一个事务，也就是说当用户在一次会话中创建了一个事务，在这个事务结束前用户不能再创建新的事务。

事务作为一个完整的工作单元执行，事务中的操作要么全部执行成功要么全部执行失败。SequoiaDB 事务中的操作只能是插入数据、修改数据以及删除数据，在事务过程中执行的其它操作不会纳入事务范畴，也就是说事务回滚时非事务操作不会被执行回滚。如果一个表或表空间中有数据涉及事务操作，则该表或表空间不允许被删除。

默认情况下，事务功能是关闭的。

如要打开事务功能需要在节点的配置文件中配置参数：transactionon = TRUE；在创建数据节点时，增加 JSON 类型的参数：{ "transactionon": "YES" } 或 { "transctionon": true }。

最终一致性策略

为了提升数据的可靠性和实现数据的读写分离，SequoiaDB 中，对于复制组间的数据采用“最终一致性”策略，在读写分离时读取的数据某一个时期内可能不是最新的，但最终是一致的。

名词解释

W：副本写入个数

R：副本读取个数

N：副本个数

在 SequoiaDB 中，设置 R 的值为1，且不可配置。

默认情况下，复制组中的主节点在处理完一个写请求后会立即返回，即 $W = 1$ 。数据同步会在后台异步完成（[同步日志](#)）并达到最终一致。此时外部的读请求获得的数据可能不是最新的。在对数据一致性要求不高的场景中，这种方式可以提供最优的写入性能。

当我们[创建集合](#)时，可以通过 `ReplSize` 属性指定集合的 W 值。

- 默认情况下 $W = 1$ 。
- 当 `ReplSize` 等于 0 时， W 的个数会根据当前复制组的 N 变化而变化。即，如果开始组内有三个节点，则 W 等于 3。当新增加一个入节点时， W 会自动变为 4。
- 当手动指定 W 的个数时，不能超出当前复制组内节点个数。

增大 W 可以有效提高数据的一致性和可靠性。当 $W = N$ 并且写请求处理成功后，后续读到的数据一定是当前组内最新的。但是这样会降低复制组的写入性能。值得注意的是，虽然我们可以将 W 设为 N ，但这并不代表 SequoiaDB 中的数据拥有强一致性。当某个副本写入失败（如磁盘满）时，复制组内可能存在多个版本的数据。此时既可能读到新的数据，也可能读到旧的数据。当失败副本恢复正常后，会继续从主节点上同步最新的数据并达到最终一致。

实例

复制组实例 复制组中的每个数据节点都存储该复制组的一份完整数据，因此也称复制组中的每个节点为复制组实例。复制组实例可根据节点在复制组中的位置分为“主”，“备”或“0~7”标识。

数据库实例 所有复制组中相同位置的复制组实例共同构成数据库实例，因此数据库实例也可以分为“主”，“备”或“0~7”标识。

读写分离

写请求处理

所有写请求都只会发往主节点，如果没有主节点则当前复制组不可处理写请求。

读请求处理

读请求会按照会话（连接）随机选择组内任意一个节点（对外透明），或按照当前会话（连接）配置的优先实例策略选取相应复制组的数据节点。在一次会话中如果上一次查询（包括 `query` 和 `fetch`）返回成功，则下一次查询不会重选节点；如果上一次查询发生失败，则下一次查询将重选节点。如果没有可用节点则返回失败。一次查询中不会重选节点。

后台任务

后台任务是 SequoiaDB 中的一种特殊任务类型，一般用于将特定用户操作置于后台异步执行。在快照中，后台任务的类型（Type）为“Task”。

后台任务类型列表：

任务名	描述
<code>Restore</code>	数据库恢复任务，用于根据日志文件回滚恢复数据库。
<code>Job[PageCleaner]</code>	脏页清除任务，用于异步将未写入磁盘的脏页刷入磁盘。可以使用 <code>-numpagecleaners</code> 控制脏页清除任务数量，默认为 1。
<code>Job[Prefetch]</code>	预取任务，用于在等待客户端接收下一个操作请求时，在后台执行用户接下来可能发生的操作。可以使用 <code>-maxprefpool</code> 控制最大预取任务的数量。
<code>CreateIndex</code>	建立索引任务，用于后台建立索引，多用于备节点重做主节点的建立索引操作日志。
<code>DropIndex</code>	删除索引任务，用于后台删除索引，多用于备节点重做主节点的删除索引操作日志。
<code>CleanUp</code>	数据清理任务，多用于数据切分后，在源数据节点删除被切分数据。
<code>Job[ExtendSegment]</code>	扩展集合空间文件任务，用于当集合空间空闲数据页小于特定阈值后，由后台启动异步扩充集合空间。

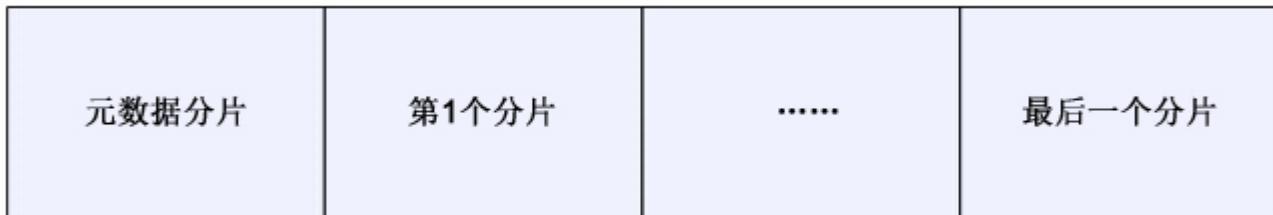
大对象

概念

大对象（LOB）功能旨在突破 SequoiaDB 的单条最大记录长度为 16MB 的限制，为用户写入和读取更大型记录提供便利。LOB 记录的大小目前不受限制。

每一个 LOB 记录拥有一个 OID，通过指定集合及 OID 可以访问一条 LOB 记录。在非分区集合及哈希分区集合中均可使用 LOB 功能。集合间不共享 LOB 记录。当一个集合被删除时，其拥有的 LOB 记录自动删除。

LOB 记录的存储格式：



每个 LOB 记录包含若干个分片。分片所占空间大小均为 LobPageSize（创建集合空间时指定）。在哈希分区中，LOB 记录的每一个分片会被按照 OID 加分片序号分散存储在相应的分区组中。其哈希空间与所属集合的哈希空间相同。

目前 LOB 的存储格式为二进制类型。

支持的操作

操作	备注
创建	LOB 记录一旦创建完毕，其内容无法再做更改。
读取	支持 seek 操作。
删除	无

示例

在 Sdb Shell 中将本地文件 mylob 上传至集合 foo.bar 中：

```
db.foo.bar.putLob('/opt/mylob');
```

在 Sdb Shell 中将集合 foo.bar 中的 OID 为 5435e7b69487faa663000897 的 LOB 记录下载到本地文件 mylob 中：

```
db.foo.bar.getLob('5435e7b69487faa663000897','/opt/newlob')
```

集群

SequoiaDB 集群是指通过并联多台数据库服务器，达到并行计算，以提升数据请求效率的方式。

使用 SequoiaDB 集群，用户可以得到：

高性能的数据访问

通过并行计算的方式，在多台数据库服务器上分布式执行任务，降低了单节点的资源消耗，提升整体吞吐量与数据访问性能。

24x7的高可用性

复制组内数据复制保障了数据的可用性，任何一台系统宕机不影响数据的访问。

数据库的水平扩张能力

通过增加复制组的数量，数据库可以容纳更多的数据；通过增加复制组内节点的数量，数据库可以同时处理更多的读请求。数据库节点数量与整体吞吐量呈近线形上升趋势。

运行模式

概念

运行模式指启动 SequoiaDB 服务时，该服务以独立模式启动还是以集群模式启动。

独立模式

独立模式是启动 SequoiaDB 的最精简模式，仅需要启动一个独立模式的数据节点，即可进行数据服务。

在独立模式中，SequoiaDB 数据库作为一个独立的进程不需要与其他除客户端以外的进程进行通讯。所有的数据均存放在数据节点内。

以独立模式启动的数据库不可进行分区，也不可进行数据复制。因此，在对数据安全性要求较高的环境下不建议使用独立模式。

独立模式的数据库中不存在编目信息。

一般推荐在开发环境中使用独立模式，以减少对硬件资源的需求。

集群模式

集群模式是启动 SequoiaDB 的标准模式，至少需要三个节点。

在集群环境下，SequoiaDB 数据库需要三种角色的节点，分别为：

- [数据节点](#)
- [编目节点](#)
- [协调节点](#)

集群模式的最小配置中，每种角色的节点至少启动一个，才能构成完整的集群模式。

集群模式中客户端或应用程序直接连接到协调节点，其余数据节点与编目节点对应用程序完全透明。

应用程序本身不需关心数据存放在哪个数据节点，协调节点会对接收到的请求解析，自动将其发送到需要的数据节点上进行处理。

在集群模式下，复制组之间的数据无共享，复制组内的节点间进行异步数据复制，保证数据的最终一致性。

节点

概念

在 SequoiaDB 集群环境中，数据库节点分为物理节点与逻辑节点。

逻辑节点

逻辑节点即一个单独的数据库服务，代表一个最基本的数据库服务单元。

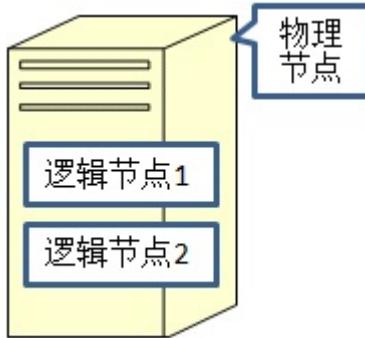
在 Windows 与 Unix 操作系统中，一个逻辑节点即一个 SequoiaDB 进程。一台计算机中可以运行多个逻辑节点。

物理节点

物理节点即一台单独运行操作系统的服务器，代表一个最基本的物理单元。

在虚拟化环境中，物理节点代表一台虚拟机。

一个物理节点中可以运行多个逻辑节点。一般来说，同一个物理节点中的每个逻辑节点监听不同的端口，各自从相应的端口获得请求信息并返回结果。



编目节点

概念

编目节点为一种逻辑节点，其中保存了数据库的元数据信息，而不保存其他用户数据。

编目节点中包含4个集合空间：

- SYSCAT 系统编目集合空间，包含4个系统集合：

集合名	描述
SYSCOLLECTIONS	保存了该集群中所有的用户集合信息
SYSCOLLECTIONSPACES	保存了该集群中所有的用户集合空间信息
SYSNODES	保存了该集群中所有的逻辑节点与复制组信息
SYSTASKS	保存了该集群中所有正在运行的后台任务信息

- SYSTEMP 系统临时集合空间，可以创建最多4096个临时集合
- SYSAUTH 系统认证集合空间，包含一个用户集合，保存当前系统中所有的用户信息

集合名	描述
SYSUSRS	保存了该集群中所有的用户信息

- SYSPROCEDURES 系统存储过程集合空间，包含一个集合，用于存储所有的存储过程函数信息

集合名	描述
STOREPROCEDURES	保存所有存储过程函数信息

除了编目节点外，集群中所有其他的节点不在磁盘中保存任何全局元数据信息。当需要访问其他节点上的数据时，除编目节点外的其他节点需要从本地缓存中寻找集合信息，如果不存在则需要从编目节点获取。

编目节点与其它节点之间主要使用编目服务端口（catalogname参数）进行通讯。

SYSCOLLECTIONS 集合

所属集合空间

SYSCAT

概念

SYSCOLLECTIONS 集合中包含了该集群中所有的用户集合信息。每个用户集合保存为一个文档。

每个文档包含以下字段：

字段名	类型	描述
Name	字符串	集合的完整名，为<集合空间>.〈集合名〉形式。

字段名	类型	描述
Version	整数	集合的版本号，由1起始，每次对该集合的元数据变更会造成版本号+1。
ReplSize	整数	最小复制组，确保任何写操作必须被复制到至少指定数量的节点后返回成功。
ShardingKey	对象	分区键，在分区集合中存在。对象包含一个或多个字段，字段名为分区字段名，数值为1或者-1，代表对该列正向或逆向排序。
ShardingType	字符串	分区类型，在分区集合中存在。分区类型有：范围分区（Range）和散列分区（Hash）两种。
Partition	整数	散列分区的分区大小值，必须为2的幂。
CataInfo	数组	集合所在的逻辑节点信息。在单分区集合中，该数组仅包含一个元素，代表该集合所在的分区组。在多分区集合中，该数组中包含一个或多个元素，代表该集合中的每一个取值范围所在的分区组。每个取值范围包括LowBound与UpBound，代表其下限与上限，闭合关系为左闭右开。

示例

一个典型的单分区集合信息如下：

```
{ "Name" : "test.foo", "Version" : 1, "CataInfo" : [ { "GroupID" : 1000 } ] }
```

一个典型的多分区集合信息如下：

```
{ "Name" : "foo.test",
  "Version" : 1,
  "ShardingKey" : { "Field1" : 1, "Field2" : -1 },
  "ShardingType" : "range",
  "ReplSize": 3,
  "CataInfo" : [
    { "GroupID" : 1000,
      "LowBound" : { "" : MinKey, "" : MaxKey },
      "UpBound" : { "" : MaxKey, "" : MinKey } }
  ]
}
```

SYSCOLLECTIONSPACES 集合

所属集合空间

SYSCAT

概念

SYSCOLLECTIONSPACES 集合中包含了该集群中所有的用户集合空间信息。每个用户集合空间保存为一个文档。

每个文档包含以下字段：

字段名	类型	描述
Name	字符串	集合空间名。
Collection	数组	该集合空间中包含的所有集合名，每个集合为一个 JSON 对象，包含“Name”字段与相应的集合名。
Group	数组	该集合空间所在的复制组 ID。
PageSize	整数	该集合空间的数据页大小。

示例

一个典型的包含一个集合，存放在一个复制组中的集合空间如下：

```
{ "Collection" : [ { "Name" : "foo" } ],
  "Group" : [ { "GroupID" : 1000 } ],
  "Name" : "test",
  "PageSize" : 4096
}
```

SYSNODES 集合

所属集合空间

SYSCAT

概念

SYSNODES 集合中包含了该集群中所有的节点与复制组信息。每个复制组保存为一个文档。

每个文档包含以下字段：

字段名	类型	描述
GroupName	字符串	复制组名。
GroupID	整数	复制组 ID，该 ID 在集群中唯一。
PrimaryNode	整数	该复制组内主节点 ID。
Role	整数	复制组角色，可以为： <ul style="list-style-type: none"> 0：数据节点 2：编目节点
Status	整数	<ul style="list-style-type: none"> 1：已激活复制组 0：未激活复制组 不存在：未激活复制组
Version	整数	版本号，由1起始，任何对该复制组的操作均会对其+1。
Group	数组	复制组中节点信息，见下表：

复制组中如果存在一个以上节点，则每个节点作为一个对象存放在 Group 字段数组中，每个对象的信息如下：

字段名	类型	描述
HostName	字符串	节点所在的系统名，需要完全匹配该节点所在操作系统中“hostname”命令的输出。
dbpath	字符串	数据库路径，为节点所在的物理节点中对应的绝对路径。
NodeID	整数	节点 ID，该 ID 在集群中唯一。
Service	数组	服务名，每个逻辑节点对应4个服务名，每个服务名包括其类型与服务名（可以为端口号或 services 文件中的服务名）。类型如下： <ul style="list-style-type: none"> 0：直连服务，对应数据库参数 svcname 1：复制服务，对应数据库参数 replname 2：复制服务，对应数据库参数 shardname 3：编目服务，对应数据库参数 catalogname

注:

- 编目复制组名固定为“SYSCatalogGroup”，复制组ID固定为1。
- 数据复制组 ID 由1000起始。
- 数据节点 ID 由1000起始。

示例

一个典型的包含单节点的编目复制组为：

```
{ "Group" : [
    { "NodeID" : 2,
      "HostName" : "vmsvr1-rhel-x64",
      "Service" : [
        { "Type" : 3, "Name" : "11803" },
        { "Type" : 1, "Name" : "11801" },
        { "Type" : 2, "Name" : "11802" },
        { "Type" : 0, "Name" : "11800" } ],
      "dbpath" : "/home/sequoiadb/sequoiadb/catalog"
    },
    "GroupID" : 1,
    "GroupName" : "SYSCatalogGroup",
    "PrimaryNode" : 2,
    "Role" : 2,
    "Version" : 1 } }
```

一个典型的包含单节点的数据复制组为：

```
{ "Group" : [
    { "dbpath" : "/home/sequoiadb/sequoiadb/data3",
      "HostName" : "vmsvr1-rhel-x64",
      "Service" : [
        { "Type" : 0, "Name" : "11820" },
        { "Type" : 1, "Name" : "11821" },
        { "Type" : 2, "Name" : "11822" },
        { "Type" : 3, "Name" : "11823" } ],
      "NodeID" : 1001 },
    "GroupID" : 1001,
    "GroupName" : "foo1",
    "PrimaryNode" : 1001,
    "Role" : 0,
    "Status" : 1,
    "Version" : 1 } }
```

SYSTASKS 集合

所属集合空间

SYSCAT

概念

SYSTASKS 集合中包含了该集群中所有正在运行的后台任务信息。每个任务保存为一个文档。

每个文档包含以下字段：

字段名	类型	描述
JobType	整数	任务类型，分别代表： • 0：数据切分
Status	整数	任务状态，分别代表： • 0：准备

字段名	类型	描述
		<ul style="list-style-type: none"> • 1 : 运行 • 2 : 暂停 • 3 : 取消 • 4 : 变更元数据 • 9 : 完成
CollectionSpace	字符串	集合空间名
Collection	字符串	集合名

数据切分

对于数据切分操作，每个文档还存在以下字段：

字段名	类型	描述
SourceName	字符串	源分区所在复制组名
TargetName	字符串	目标分区所在复制组名
SourceID	整数	源分区所在复制组ID
TargetID	整数	目标分区所在复制组ID
SplitValue	对象	数据分区键

SYSUSR 集合

所属集合空间

SYSAUTH

概念

SYSUSR 集合中包含了该集群中所有注册用户的信息。每个用户保存为一个文档。

每个文档包含以下字段：

字段名	类型	描述
User	字符串	用户名。
Password	字符串	对用户密码进行 MD5 散列的结果。



注:

如果该集合为空，则对任何连接不进行身份认证。

STOREPROCEDURES 集合

所属集合空间

SYSPROCEDURES

概念

STOREPROCEDURES 集合中包含了所有的存储过程函数，每一个函数保存为一个文档，每个文档包含以下字段：

字段名	类型	描述
name	字符串	函数名
func	字符串	函数体
funcType	整数	函数类型 <ul style="list-style-type: none"> • 0 : 代表 JavaScript 函数

字段名	类型	描述
		其他类型暂无

示例

一个简单的存储过程函数如下：

```
{
  "_id" : { "$oid" : "5257b115925c31dd16ec4e4a" },
  "name" : "fun",
  "func" : "function fun(num) {
    if (num == 1) {
      return 1;
    } else {
      return fun(num - 1) * num;
    }
  }",
  "funcType" : 0 }
```

协调节点

概念

协调节点为一种逻辑节点，其中并不保存任何用户数据信息。

协调节点作为数据请求部分的协调者，本身并不参与数据的匹配与读写操作，而仅仅是将请求分发到所需要处理的数据节点。

一般来说，协调节点的处理流程如下：

- 得到请求
- 解析请求
- 本地缓存查询该请求对应集合的信息
- 如果信息不存在则从编目节点获取
- 将请求转发至相应的数据节点
- 从数据节点得到结果
- 把结果汇总或直接传递给客户端

协调节点与其它节点之间主要使用分区服务端口（shardname参数）进行通讯。

数据节点

概念

数据节点为一种逻辑节点，其中保存用户数据信息。

数据节点中没有专门的编目信息集合，因此第一次访问集合前需要向编目节点请求该集合的元数据信息。

在独立模式中，数据节点为单独的服务提供者，直接与应用程序或客户端进行通讯，并且不需要访问任何编目信息。

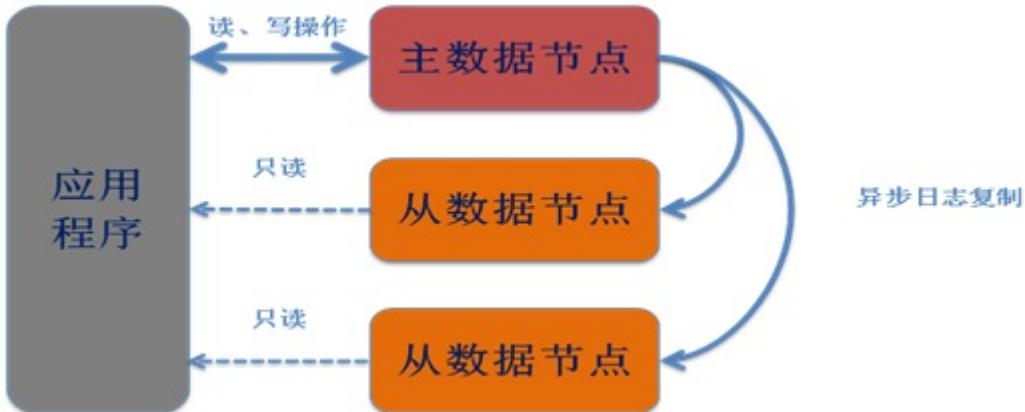
分区组

概念

分区组又被称为复制组，一个复制组内可以包含一个或多个数据节点（或编目节点），节点之间的数据使用异步日志复制机制，保持最终一致。

分区组中所有的节点之间使用复制服务端口（replicname参数）进行通讯，定期相互发送心跳信息以相互验证状态。

分区组结构如下图：



每个分区组内的节点有两种状态：

主节点

主节点可作读写操作。所有写入的数据会同步写入日志文件，日志文件中的日志信息会异步写入从节点。

从节点

从节点可作只读操作。所有主节点写入的数据会异步写入从节点。因此从节点与主节点之间可能存在暂时的数据不一致，但是复制机制可以保证数据的最终一致性。

分区组通过数据复制，读写分离与选举机制实现高可用。

选举

概念

选举机制保障分区组中随时存在一个主节点。当该主节点宕机后会在其余从节点之间自动选举出主节点，进行读写操作。

选举机制的核心为节点状态监测。分区组内所有的节点定期向组内其他成员发送自身状态，因此当主节点宕机后，所有的从节点间会进行投票，当时最匹配原主节点的节点即当选新的主节点。



选举成功的前提条件为组内必须拥有超过半数以上的节点参与投票，否则为了避免“双活”问题（同时存在两个主节点）将无法进行选举。

任何时刻如果组内成员不足半数，则当前的主节点会自动降级为从节点，同时断开当前节点的所有用户连接。

当一个新的节点加入现存的分区组，或者某个故障节点重新加入分区组后，会进行[数据同步](#)。

复制

概念

数据复制为分区组中节点之间的相互同步的机制。

在数据节点和编目节点中，任何数据增删改操作均会写入日志。SequoiaDB 会首先将日志写入日志缓冲区，然后将其异步写入本地磁盘。

每个数据复制会在两个节点间进行：

源节点

为包含新数据的节点。主节点并不一定永远是复制的源节点。

目标节点

为请求进行数据复制的节点。

复制过程中，目标节点选择一个与其最接近的节点，然后向其发送一个复制请求。源节点接到复制请求后，会将目标节点请求的同步点之后的日志记录打包并发送给目标节点，目标节点接收到数据包后会重新处理日志中的所有操作。

节点之间的复制有两个状态：

- 对等状态（PEER）：当目标节点请求的日志依然存在于源节点的日志缓冲区中，两节点之间为对等状态
- 远程追赶状态（Remote Catchup）：当目标节点请求的日志不存在于源节点的日志缓冲区中，但依然存在于源节点的日志文件中，两节点之间为远程追赶状态

如果目标节点请求的日志已经不再存在于源节点的日志文件中，目标节点则进入[恢复同步](#)状态。

当两节点处于对等状态时，同步请求在源节点可以直接从内存中获取数据，因此目标节点选择复制源节点时，总会尝试选择距离自己当前日志点最近的数据节点，使其所包含的日志尽量坐落在内存中。

全量同步

概念

在分区组内，当一个新的节点加入分区组，或者故障节点重新加入分区组，需要进行数据全量同步，以保障新的节点与现有节点之间数据的一致性。

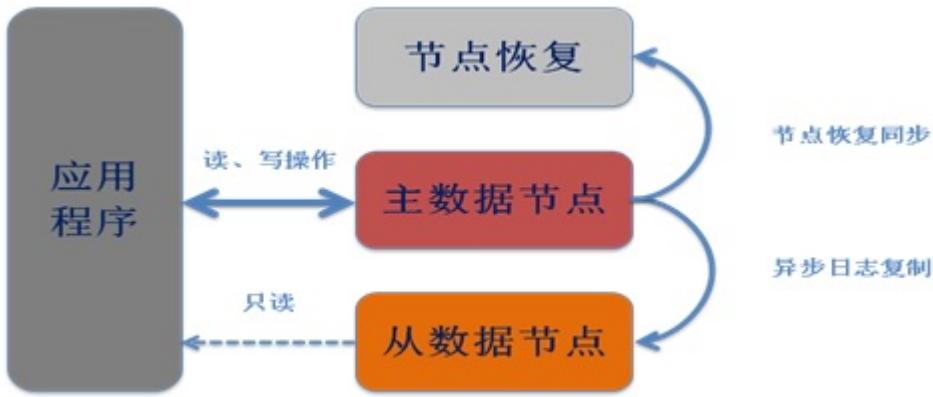
在进行数据全量同步时有两个节点参与：

源节点

为包含有效数据的节点。主节点并不一定永远是同步的源节点。任何与主节点处于同步状态的从节点均可作为源节点进行数据同步。（目前只能主节点作为同步源节点）

目标节点

为新加入组，或重新入组的故障节点。同步时该节点下原有的数据会被废弃。



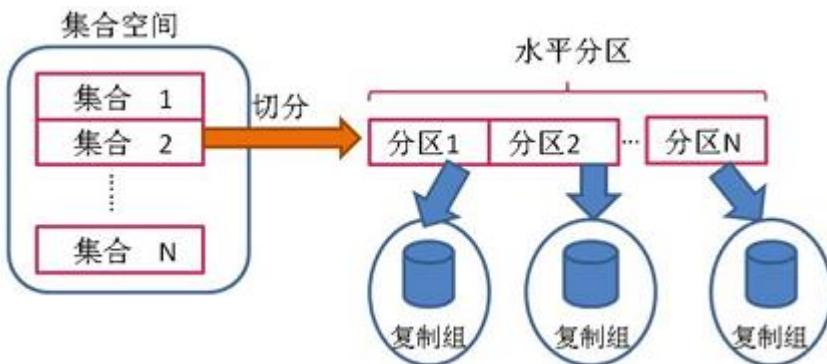
全量同步发生时，目标节点会定期向源节点请求数据。源节点将数据打包后作为大数据块发送给目标节点。
当目标节点重做该数据块内所有数据后，向源节点请求新的数据块。

为保障源节点在同步时可进行写操作，所有已经被发送给目标节点的数据页如果被更改，其更新会被同步到目标节点，以保障全量同步过程中更新的数据不会损失。

数据分区

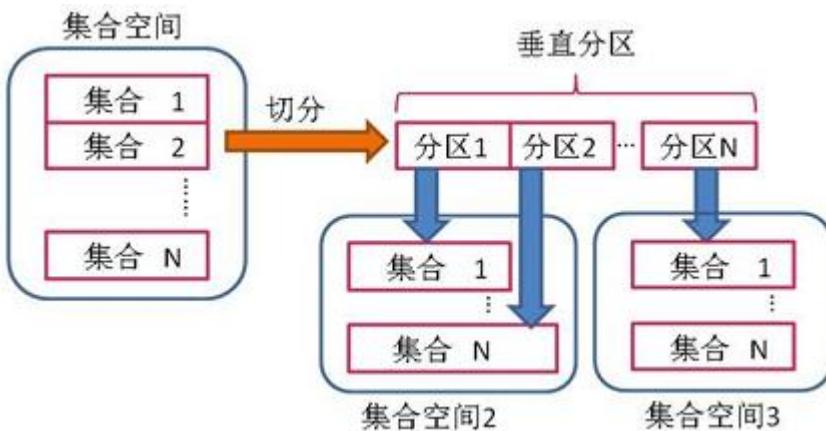
水平分区 水平分区又称为数据库分区或横向分区。

在 SequoiaDB 集群环境中，用户可以通过将一个集合中的数据切分到多个复制组中，以达到并行计算的目的，此数据切分称为水平分区。水平分区是按一定的条件把全局关系的所有元组划分成若干不相交的子集，每个子集为关系的一个片段，称为分区；一个分区只能存在于一个复制组中，但一个复制组可以承载多个分区；分区在复制组之间可以通过水平切分操作进行移动。



垂直分区 垂直分区又称为集合分区或纵向分区。

在 SequoiaDB 集群环境中，用户也可以将一个集合全局关系的属性分成若干子集，并在这些子集上作投影运算，将这些子集映射到另外的集合上，从而实现集合关系的垂直切分；该集合称之为子集合，每个切分的子集称为分区，分区映射的集合称为子集合；一个分区只能映射到一个子集合中，但一个子集合可以承载多个分区；分区在子集合之间可以通过垂直切分操作进行重映射。

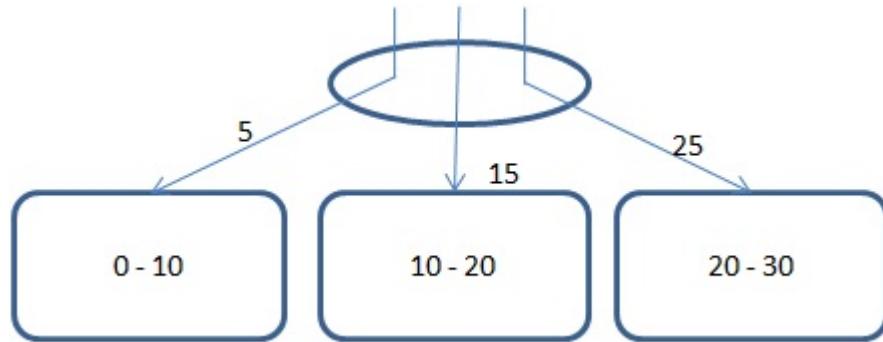


混合分区 在 SequoiaDB 集群环境中，可以将集合先通过垂直分区映射到多个子集合中，再通过水平分区将子集合切分到多个复制组中，从而实现混合分区。

分区方式

无论是水平分区还是垂直分区都有两种方式：散列分区（Hash）和范围分区（Range）。两种分区方式中判定分区划分所依据的字段称为“分区键”。分区键基于集合定义，每个分区键可以包含一个或多个字段。

Range 方式下依据记录中分区键的范围选择所要插入的分区。Hash 方式下根据记录中分区键生成的 hash 值选择所要插入的分区。



在所示图中，为一个 Range 方式分区，方形区域为三个数据节点，椭圆形为协调节点。每个数据节点各自定义了所包含数据的范围。例如对于节点1包含了大于等于0且小于10的数据。

当用户插入一条数据时，协调节点首先判定该数据的分区键应当坐落于哪个分区。如果分区键不存在则定义为 Undefined 类型（Undefined类型也可以与普通数据类型进行对比）。

当查询到该数据所在的分区后，协调节点会将请求直接下发给指定的分区。

分区键

概念

分区键定义了每个集合中所包含数据的分区规则。每一个集合对应一个分区键，分区键中可以包含一个或多个字段。

在编目节点中，每个集合都拥有自己的分区范围，分区范围内每个范围段对应一个分区组，标示该集合的某一数据段坐落于该分区组。



注：

集合的索引键在创建集合时指定，集合创建成功后索引键无法修改。

在[分区集合](#)中，记录插入数据库后无法对分区键值进行更新。

格式

- Range 分区键

Range 分区键的格式类似于索引键，为一个 JSON 对象。JSON 对象中每一个字段对应分区键的字段，数值为1或者-1，代表正向或逆向排序。

```
{ <字段1> : <1|-1>, [ <字段2> : <1|-1> ... ] }
```

- Hash 分区键

Hash 分区的 ShardingKey 组成方式与 Range 分区方式相同（但字段的正向/逆向不起作用）。Partition 代表了 hash 分区的个数。其值必须是2的幂。范围在 $[2^3, 2^{20}]$ 。此字段为可选字段。默认为 2^{12} ，代表我们将整个范围平均划分为4096个分区。设计 hash 分区的目的是让数据分布更灵活，可以根据需要自由设置每个数据分区承担 hash 分区的范围。ShardingType 如果不填则默认为 Range 分区。

```
{ ShardingKey : { <字段1> : <1|-1>, [<字段2> : <1|-1>, ...] }, { ShardingType : "hash" }, [ { Partition : <分区数> } ] }
```

示例

- 一个包含两个字段，分别为正向和逆向排序的 Range 分区键如下：

```
{ Field1 : 1, Field2 : -1 }
```

- Hash 分区键

```
{ { Field1 : 1, Field2 : -1 }, { ShardingType : "hash" }, { Partition : 2^{12} } }
```

分区集合

概念

一个定义了分区键的集合为分区集合。分区集合可以按照分区键所指定的字段，将集合中的数据切分到超过一个数据分区组中。

当集合创建时，用户可以指定分区键。分区集合会在一个随机的数据分区组中创建。用户可以使用手工切分的方式对集合按照某一规则切分至多个数据分区组中。

分区区间

分区集合中每一个区间叫做一个分区区间。

分区集合创建时，其所在的分区组包含全部区间，为所有字段的 MinKey 至 MaxKey。

每一个分区区间为左闭右开规则，也就是包含大于等于低边界，且小于高边界的区域。例如：

```
{ LowBound: { "" : 10 }, UpBound: { "" : 20 } }
```

在该例中，低边界为10，高边界为20，因此本区间包含所有分区字段大于等于10，且小于20的数据。

 注：一个集合中所有边界的定义不包含字段名，其字段应当与分区键所定义的字段，与字段数量保持一致。

当分区键包含多个字段时，其匹配规则为第一字段首先匹配，如果坐落于边界值则匹配下一段。例如：

```
{ LowBound: { "" : 10, "" : 5 }, UpBound: { "" : 20, "" : 1 } }
```

在该分区区间中，如果用户输入的分区键的第一个字段坐落于10与20之间，则立刻判定为该区间内；如果存在于小于10或大于20，则不在该区间内；而如果为10或者20，则需要进行第二个字段的匹配，匹配规则仍为左闭右开。

规则

分区集合定义的规则参见 [SYSCOLLECTIONS 集合定义](#)。

示例

一个存在于两个分区组的典型分区区间如下：

```
[
  {
    "GroupID" : 1000,
    "LowBound" : { "" : MinKey, "" : MaxKey },
    "UpBound" : { "" : 10, "" : 5 }
  },
  {
    "GroupID" : 1001,
    "LowBound" : { "" : 10, "" : 5 },
    "UpBound" : { "" : MaxKey, "" : MinKey }
  }
]
```

其中第一个区间所在分区组 ID 为1000，包含的分区键存在两个字段，分别为：

- 低边界：{ "" : MinKey, "" : MaxKey }
- 高边界：{ "" : 10, "" : 5 }

而第二个区间所在分区组为1001，包含分区区间为：

- 低边界：{ "" : 10, "" : 5 }
- 高边界：{ "" : MaxKey, "" : MinKey }

分区索引

概念

每一个分区集合都会默认创建一个名叫“\$shard”的索引，该索引叫做分区索引。

非分区集合不存在分区索引。

分区索引存在于分区集合所在的每一个分区组中，其字段定义顺序和排列与分区键相同。



注：

任何用户定义的唯一索引必须包含分区索引中所有的字段，其字段顺序无关。

在分区集合中，_id 字段仅保证分区内该字段唯一，无法保证全局唯一。

示例

一个典型的分区索引如下：

```
{
  "IndexDef" : {
    "name" : "$shard",
    "_id" : { "$oid" : "515954bfa88873112fa6bd3a" },
    "key" : { "Field1" : 1, "Field2" : -1 },
    "v" : 0,
    "unique" : false,
    "dropDups" : false,
    "enforced" : false
  },
  "IndexFlag" : "Normal"
}
```

后台任务

概念

后台任务是一种不阻塞前端会话的任务，并不会随着前端会话的中断而停止。

所有的后台任务都会在编目节点的 SYSCAT.SYSTASKS 集合中跟踪，不同类型的后台任务可能包含不同的字段。

以下字段存在于所有后台任务中：

字段名	类型	描述
JobType	整数	任务类型，分别代表： • 0：数据切分
Status	整数	任务状态，分别代表： • 0：准备 • 1：运行 • 2：暂停 • 3：取消 • 4：变更元数据 • 9：完成
CollectionSpace	字符串	集合空间名
Collection	字符串	集合名

后台任务类型包括：

- [数据切分](#)

数据切分

概念

一个分区集合首先会被创建在一个随机的分区组中。如果用户希望对该集合水平切分，将其划分到超过一个分区组中，就需要数据切分功能。

数据切分是一种将数据在线从一个分区组转移到另一个分区组的方式。在数据转移的过程中，查询所得的结果集数据会存在暂时的不一致，但是 SequoiaDB 可以保证磁盘中数据的最终一致性。

Range 分区和 Hash 分区都包含两种切分方式：范围切分和百分比切分。在范围切分时，Range 分区使用精确条件，而 Hash 分区使用 Partition（分区数）条件。切分时起始条件为必填字段，而结束条件为选填条件，结束条件默认为切分源当前包含的最大数据范围。

例如：

Hash:

```
db.foo.bar.split('src', 'dst', {Partition: 10}, {Partition: 20})
```

Range:

```
db.foo.bar.split('src', 'dst', {a: 10}, {a: 20})
```

数据切分及分区上的数据范围皆遵循左闭右开原则。即：{Partition:10}, {Partition:20} 代表迁移数据范围为[10, 20)。

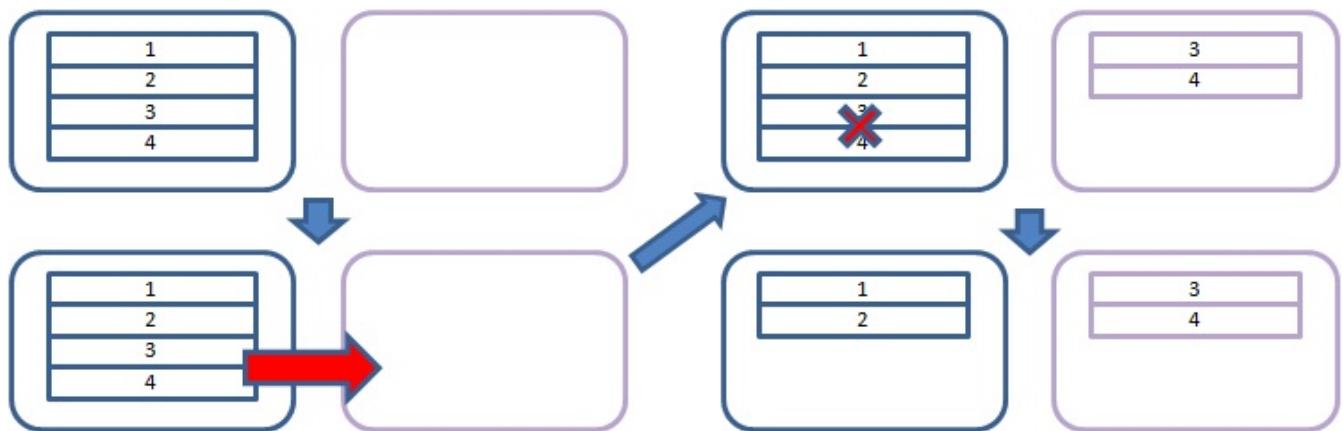
百分比切分：

```
db.foo.bar.split('src', 'dst', 50)
```



注：

- 当切分范围不冲突时，可以做并发切分
- “src”、“dst”分别表示“源分区所在复制组”、“目标分区所在复制组”



图中左上角为系统的起始状态，4条记录均存放在左侧的节点中。切分时定义由3起始，因此数据3与4会被切分至右侧节点。（左下图）

右上图为第三状态，数据在两个分区组中同时存在。此刻数据会有暂时的不一致。最终状态切换到右下图，已经迁移成功的数据从原始节点删除，数据最终恢复一致。

在数据切分过程在两个数据分区组之间进行交互：

- 源分区所在复制组：代表数据原本所存在的分区
- 目标分区所在复制组：代表切分后，所有需要迁移的数据的目标组

后台任务

数据切分属于一个[后台任务](#)。

对于数据切分的后台任务拥有几个特有的字段：

字段名	类型	描述
SourceName	字符串	源分区所在复制组名
TargetName	字符串	目标分区所在复制组名
SourceID	整数	源分区所在复制组 ID
TargetID	整数	目标分区所在复制组 ID
SplitValue	对象	数据切分键

数据切分的后台操作分为几个阶段：

准备阶段

在准备阶段中，并不会向编目节点的 SYSCAT.SYSTASKS 插入任务记录。该阶段首先向编目节点查询，确保该请求合法，并且向源数据节点组请求得到一条包含分区条件的记录或根据规则生成一条包含分区条件的记录。

预备阶段

在预备阶段中，协调节点将分区条件发送至编目节点。编目节点在 SYSCAT.SYSTASKS 集合中插入后台操作记录。

运行阶段

在运行阶段中，协调节点向目标节点发送切分请求，目标节点创建后台任务，从源节点请求数据，并向编目节点上报自身状态。目标节点会在后台任务创建后直接返回给协调节点，并不会长时间阻塞用户会话。

清除阶段

在清除阶段中，目标节点已经从源节点得到所有的数据，因此向编目节点发送清除请求，并在源数据节点进行数据清除操作。

完成阶段

在源节点清除了所有已经迁移的数据后，会向编目节点发送完成消息。编目节点从 SYSCAT.SYSTASKS 集合中删除该任务。

域

概念

域（Domain）是由若干个复制组（Replica Group）组成的逻辑单元。每个域都可以根据定义好的策略自动管理所属数据，如数据切片和数据隔离等。

当域中的复制组为0个时，称作空域。空域中不能创建集合空间。

一个复制组可以属于多个域。

在逻辑上存在一个系统域称作“SYSDOMAIN”。当前系统所有复制组都属于系统域。用户创建域时不能使用“SYSDOMAIN”作为域名，也不能直接操作系统域。

域拥有除名称外的以下属性：

属性名	描述
AutoSplit	当此属性为 True 时，在该域上创建的散列分区集合会被自动切分至包含的所有复制组上。

安装指南

安装 SequoiaDB 产品的基本步骤如下：

[规划数据库部署](#)

[SequoiaDB 产品安装的系统需求](#)

[准备安装介质](#)

[SequoiaDB 服务器安装部署](#)

[SequoiaDB Web 监控](#)

规划数据库部署

SequoiaDB 是一个全分布式的系统架构，支持各种灵活的部署方式。为了更好的发挥硬软件性能，在安装系统之前，需要对系统如何部署，网络的连接做好提前规划。

SequoiaDB 目前支持两种形式的部署：

- 独立模式

只在一台物理机上，启动一个数据类型的业务进程。这种模式性能高、安装部署简单方便。缺点是不支持分布式部署，不支持高可用。适用于数据总量不大，总 IOPS 吞吐较小，但对单次操作延时低的场景。

- 集群模式

可以分布式部署到多物理机上，最大支持300台物理机。集群模式需要部署编目节点、数据节点、协调调节点以及 Web 管理节点（可选）。每台物理机上可部署任意多个逻辑节点，系统最大支持65535个逻辑节点。

注：独立模式可以迁移到集群模式，迁移过程中需要中断小于10分钟的业务。

用户可根据容量、性能、可靠性、成本方面的因素，规划好部署的方式，如下几种典型的部署方式供参考。

实际上可部署的方式非常灵活，用户可以根据实际需要组合出不同的部署方式。

[最简部署](#)

[高可用部署](#)

[高性能部署](#)

最简部署

最简部署方式适用于对数据库要求不高：数据量不大，总吞吐不高，可靠性要求不高的应用。

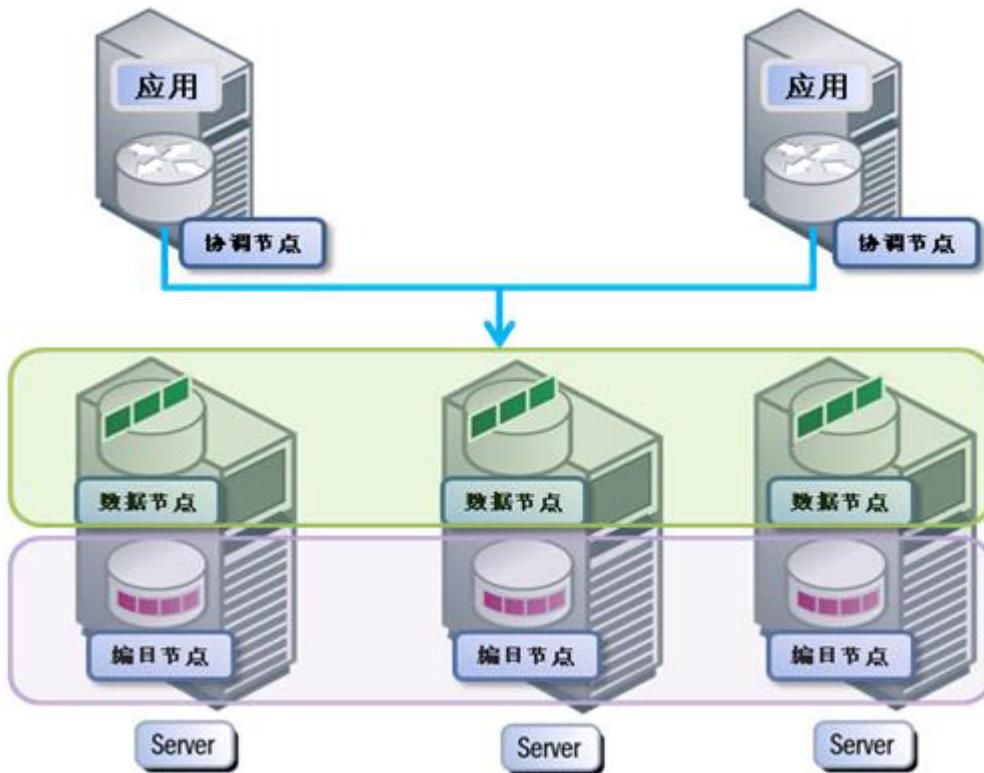
在该部署模式中，SequoiaDB 采用独立模式只启动一个数据库服务进程，业务应用可以与数据库合部，也可以部署在另外一台服务器上。



高可用部署

高可用部署方式适用于对可靠性要求高，但数据量不大、总吞吐要求不高的应用场景。在该部署中，在三台物理服务器上，都部署有数据节点和编目节点，三个数据节点组成一个副本组，三个编目节点组成副本集群。协调节点部署在业务应用的服务器上，也可以将应用/协调节点合部到数据库服务器上。

这种部署方式得优势就是高可靠性，任意一个物理服务器故障，数据的读写都不会受到影响。但数据容量与单个服务器的容量相同，且硬件成本相对较高。

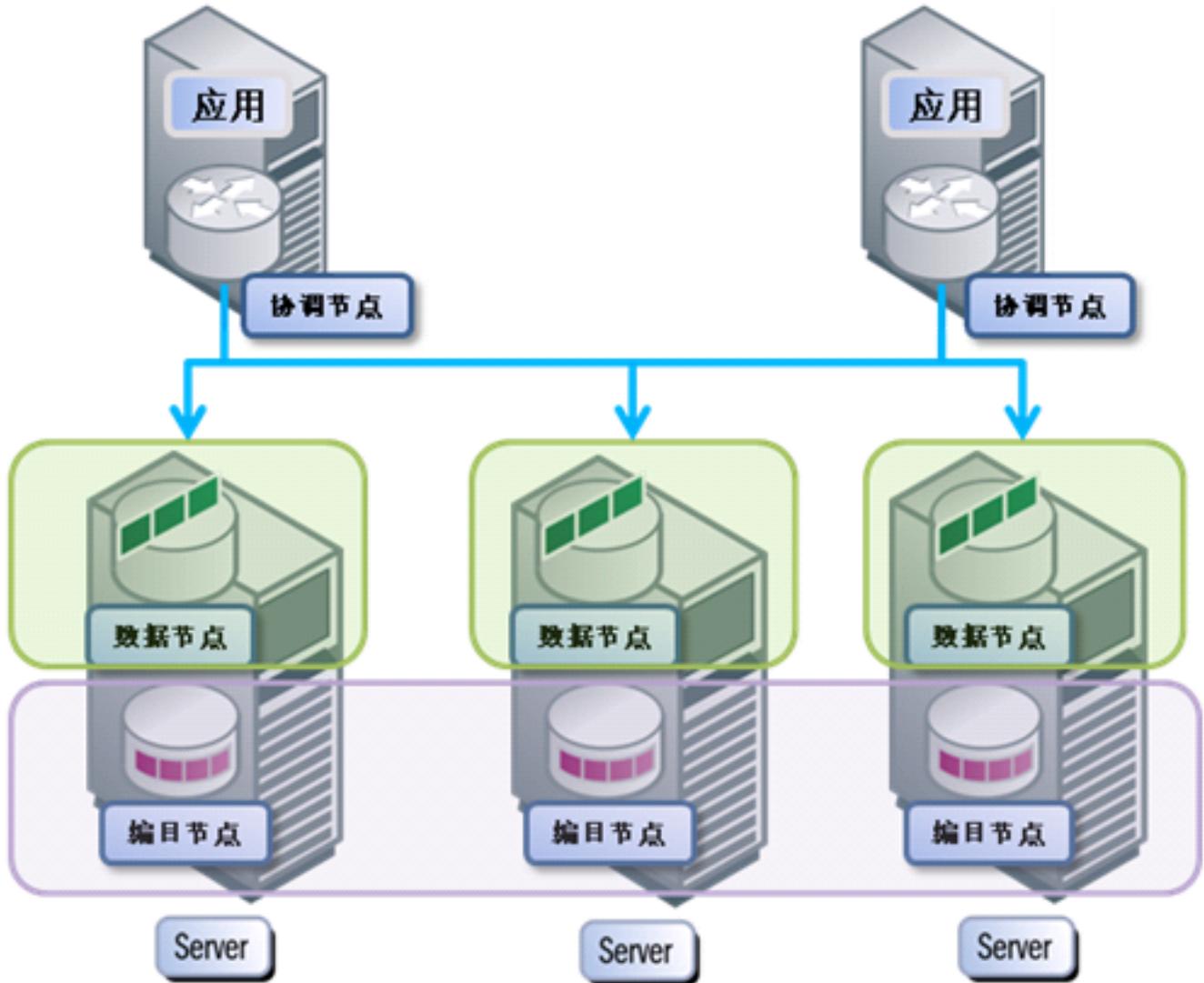


高性能部署

高性能部署方式适用于对总数据吞吐/性能要求高，硬件成本低，但对可靠性要求不高的场景。该部署方式，在三台物理机服务上都部署有编目节点和数据节点，三个编目节点组成一个副本组，每个数据节点单独

组成一个副本组（只有一个副本）。协调节点与应用部署在分离的服务器上，或者也可以部署到数据库服务器上。

这种部署模式可以充分利用所有服务器的存储容量，总的存储容量等于3个服务器的容量总和。但可靠性较低，任意一台服务器故障，都会导致部分数据无法读取和写入。



SequoiaDB 产品安装的系统要求

在安装 SequoiaDB 产品之前，请确保您选择的系统满足必须的操作系统，硬件，通信，磁盘和内存的要求。`sdbpreck` 命令将检查系统是否满足安全先决条件。

[硬件要求](#)

[受支持的操作系统](#)

[软件要求](#)

硬件要求

需求项	要求	建议
CPU	支持以下处理器：	建议采用 X64 (64 位 AMD64 和 Intel EM64T 处理器) 或者 PowerPC 处理器

需求项	要求	建议
	<ul style="list-style-type: none"> x86 (Intel Pentium、Intel Xeon 和 AMD) 32位 Intel 和 AMD 处理器 X64 (64位 AMD64 和 Intel EM64T 处理器) PowerPC 7 或者 PowerPC 7+ 处理器 	
磁盘	至少 10GB 空间	建议大于 100GB 磁盘空间
内存	至少大于 128MB	大于 2GB 物理内存
网卡	配备至少 1 张网卡	建议至少配置 1GE 网卡

受支持的操作系统

系统类型	系统列表
Linux	<ul style="list-style-type: none"> Red Hat Enterprise Linux (RHEL) 6 SUSE Linux Enterprise Server (SLES) 11 Service Pack 1 SUSE Linux Enterprise Server (SLES) 11 Service Pack 2 Ubuntu 12 CentOS 6
Power PC Linux	<ul style="list-style-type: none"> Red Hat Enterprise Linux (RHEL) 6 SUSE Linux Enterprise Server (SLES)11 Service Pack 1 SUSE Linux Enterprise Server (SLES)11 Service Pack 2

软件要求

Linux 系统要求

- 配置主机名

配置项	配置方法	验证方法
配置主机名	<p>1. 使用 root 权限登陆，执行 hostname sdbserver1 (sdbserver1 为主机名称，可根据需要修改。) ；</p> <ul style="list-style-type: none"> 对于 SUSE : <ol style="list-style-type: none"> 打开 /etc/HOSTNAME 文件 ; vi /etc/HOSTNAME 修改文件内容，配置为主机名称 ; sdbserver1 (主机名称) 按 :wq 保存退出 ; 对于 RedHat : <ol style="list-style-type: none"> 打开 /etc/sysconfig/network 文件 ; vi /etc/sysconfig/network 将 HOSTNAME 一行修改为 HOSTNAME = sdbserver1 , 其中 sdbserver1 为新主机名 ; 按 :wq 保存退出 ; 对于 Ubuntu : <ol style="list-style-type: none"> 打开 /etc/hostname 文件 ; vi /etc/hostname 修改文件内容，配置为主机名称 ; sdbserver1 	执行 hostname 命令，确认打印信息是否为“sdbserver1”

配置项	配置方法	验证方法
	4. 按 :wq 保存退出；	
配置物理机之间通过主机名可连接	<ul style="list-style-type: none"> 使用 root 权限，打开 /etc/hosts 文件 vi /etc/hosts 修改 /etc/hosts，将服务器节点的主机名与IP映射关系配置到该文件中 192.168.20.200 sdbserver1 192.168.20.201 sdbserver2 192.168.20.202 sdbserver3 保存退出 	1. ping sdbserver1 (本机主机名) 可以 ping 通 2. ping sdbserver2 (远端主机名) 可以 ping 通

- 关闭防火墙

配置项	配置方法	验证方法
关闭防火墙	关闭防火墙操作，需要管理员权限。 <ul style="list-style-type: none"> 对于 SUSE： 1. SuSEfirewall2 stop ; 2. chkconfig SuSEfirewall2_setup ; 对于 RedHat： 1. service iptables stop ; 2. chkconfig iptables off ; 对于 Ubuntu： 1. ufw disable ; 	<ul style="list-style-type: none"> 对于 SUSE： chkconfig -list grep fire ; 对于 RedHat： service iptables status ; 对于 Ubuntu： ufw status ;

注：每台作为数据库服务器的机器都需要配置。



Linux 推荐配置

- 调整 ulimit

在配置文件 /etc/security/limits.conf 中设置：

```
#<domain>      <type>    <item>      <value>
*              soft      core       0
*              soft      data       unlimited
*              soft      fsize     unlimited
*              soft      rss        unlimited
*              soft      as         unlimited
```

参数说明：

core：数据库出现故障时产生 core 文件用于故障诊断，生产系统建议关闭；

data：数据库进程所允许分配的数据内存大小；

fsize：数据库进程所允许寻址的文件大小；

rss：数据库进程所允许的最大 resident set 大小；

as：数据库进程所允许最大虚拟内存寻址空间限制；

在配置文件 /etc/security/limits.d/90-nproc.conf 中设置：

```
#<domain>      <type>    <item>      <value>
*              soft      nproc     unlimited
```

参数说明：

nproc：数据库所允许的最大线程数限制；

注：1. 每台作为数据库服务器的机器都需要配置；2. 更改配置后需重新登录使得配置生效。

- 调整内核参数

1. 使用下列命令输出当前 vm 配置，并将其归档保存：

```
cat /proc/sys/vm/swappiness
cat /proc/sys/vm/dirty_ratio
cat /proc/sys/vm/dirty_background_ratio
cat /proc/sys/vm/dirty_expire_centisecs
cat /proc/sys/vm/vfs_cache_pressure
cat /proc/sys/vm/min_free_kbytes
```

2. 添加下列参数至 /etc/sysctl.conf 文件调整内核参数：

```
vm.swappiness = 0
vm.dirty_ratio = 100
vm.dirty_background_ratio = 40
vm.dirty_expire_centisecs = 3000
vm.vfs_cache_pressure = 200
vm.min_free_kbytes = <物理内存大小的8%，单位KB>
```

注：当数据库可用物理内存不足 8GB 时不需使用 vm.swappiness = 0；上述 dirty 类参数只是建议值，具体系统设置时请按原则（控制系统的 flush 进程只采用脏页超时机制刷新脏页，而不采用脏页比例超支刷新脏页）进行设置。

3. 执行如下命令，使配置生效：

```
/sbin/sysctl -p
```

注：每台作为数据库服务器的机器都需要配置。

- 数据库目录结构

用户应尽可能使数据目录，索引目录与日志目录存放在不同物理磁盘中，以减少顺序 I/O 与随机 I/O 之间的竞争。

准备安装介质

请到 SequoiaDB 官方网站下载相应的版本。

下载地址：<http://www.sequoiadb.com/index.php?p=downserver>

SequoiaDB 服务器安装部署

安装部署分为自动化安装和手工安装两种方式，用户可以选择其中一种进行安装：

[自动化安装部署](#)

[手工安装部署](#)

自动化安装部署

自动化安装只需要选择一台机器，并在该机器上安装 OM 服务，便可以通过网页连接 OM，进行自动化安装部署集群。

[安装 OM 服务](#)

[通过 OM 部署集群](#)

安装 OM 服务

安装前准备

- 确保系统满足硬件和软件要求
- 使用 root 用户权限来安装 SequoiaDB 数据库服务
- 检查 SequoiaDB 产品软件包与 OS 系统配套
- 如果需要图形界面模式安装，请确保 X Server 服务正在运行
- 服务器配置了主机名，且与其他服务器之间可通过主机名建立网络连接（如 ssh 主机名）

 注: SequoiaDB 的安装向导需要的参数不接受非英文字符。

安装步骤

说明：

- 产品包名字以 sequoiadb-1.0.0-linux-x86_64-installer.run 为例；
- 步骤以命令行方式进行介绍，图形界面按照图像向导提示完成。

 注: 如果有多台服务器，每台机器都需要重复如下步骤安装服务器程序。

- 参照[系统配置需求](#)配置好主机名以及修改系统内核参数
- 运行安装程序

```
./sequoiadb-1.0.0-linux-x86_64-installer.run --mode text --SMS true
```

- 程序提示选择向导语言

```
Language Selection
Please select the installation language
[1] English - English
[2] Simplified Chinese - 简体中文
Please choose an option [1] :2
```

- 输入2，选择中文，显示安装协议，默认忽略阅读，如果需要读取全部文件，输入2

由 BitRockInstallBuilder 评估本所建立

欢迎来到 SequoiaDB Server 安装程序

重要信息：请仔细阅读

下面提供了两个许可协议。

- SequoiaDB 评估程序的最终用户许可协议
- SequoiaDB 最终用户许可协议

如果被许可方为了生产性使用目的（而不是为了评估、测试、试用“先试后买”或演示）获得本程序，单击下面的“接受”按钮即表示被许可方接受 SequoiaDB 最终用户许可协议，且不作任何修改。

如果被许可方为了评估、测试、试用“先试后买”或演示（统称为“评估”）目的获得本程序：单击下面的“接受”按钮即表示被许可方同时接受 (i) SequoiaDB 评估程序的最终用户许可协议（“评估许可”），且不作任何修改；和 (ii) SequoiaDB 最终用户程序许可协议 (SELA)，且不作任何修改。

在被许可方的评估期间将适用“评估许可”。

如果被许可方通过签署采购协议在评估之后选择保留本程序（或者获得附加的本程序副本供评估之后使用），SequoiaDB 评估程序的最终用户许可协议将自动适用。

“评估许可”和 SequoiaDB 最终用户许可协议不能同时有效；两者之间不能互相修改，并且彼此独立。这两个许可协议中每个协议的完整文本如下。

评估程序的最终用户许可协议

- [1] 同意以上协议：了解更多的协议内容，可以在安装后查看协议文件
- [2] 查看详细的协议内容

请选择选项 [1] :

- 是否同意协议：

同意以上协议

按 [Enter] 继续:

您是否接受此软件授权协议？ [y/n]:

- 按 y 表示同意：

请指定 SequoiaDBServer 将会被安装到的目录
安装目录 [/opt/sequoiadb]:

- 输入安装路径后按回车（默认安装在 /opt/sequoiadb），此时系统提示输入用户名，该用户名用于运行 SequoiaDB 服务

数据库管理用户配置
配置用于启动 SequoiaDB 的用户名和密码
用户名 [sdbadmin]:

- 输入用户名后按回车（默认创建 sdbadmin 用户），此时系统提示输入该用户的密码和确认密码
密码 [*****] :
确认密码 [*****] :
- 输入两次密码后（默认密码为 sdbadmin），此时系统提示输入配置服务端口

集群管理服务端口配置
配置SequoiaDB集群管理服务端口，集群管理用于远程启动添加和启停数据库节点
端口 [11790]:

 注：所有服务器的配置服务端口必须相同。

- 输入端口（默认为11790），系统提示开始安装，需要用户确认
- 询问是否允许 SequoiaDB 相关进程开机自启动

- 是否允许 SequoiaDB 相关进程开机自启动
• SequoiaDB 相关进程开机自启动 [Y/n] : Y , 输入 Y , 按回车 , 同意 SequoiaDB 相关进程开机自启动

正在安装 SequoiaDB Server 于您的电脑中，请稍候。

安装中

0% ----- 50% ----- 100%
#####

安装程序已经完成安装 SequoiaDB Server 于你的电脑中。

安装完成后，OM 会自动启动并开启8000端口的 web 服务，用户可以通过浏览器登陆 OM，并进行集群的部署。假设安装 OM 的机器 IP 为192.168.10.10，则在浏览器键入 http://192.168.1.100:8000，访问 OM 服务。

通过 OM 部署集群

OM 支持 IE7/8/9+、chrome、firefox 等主流浏览器。

- 步骤一：登录 OM

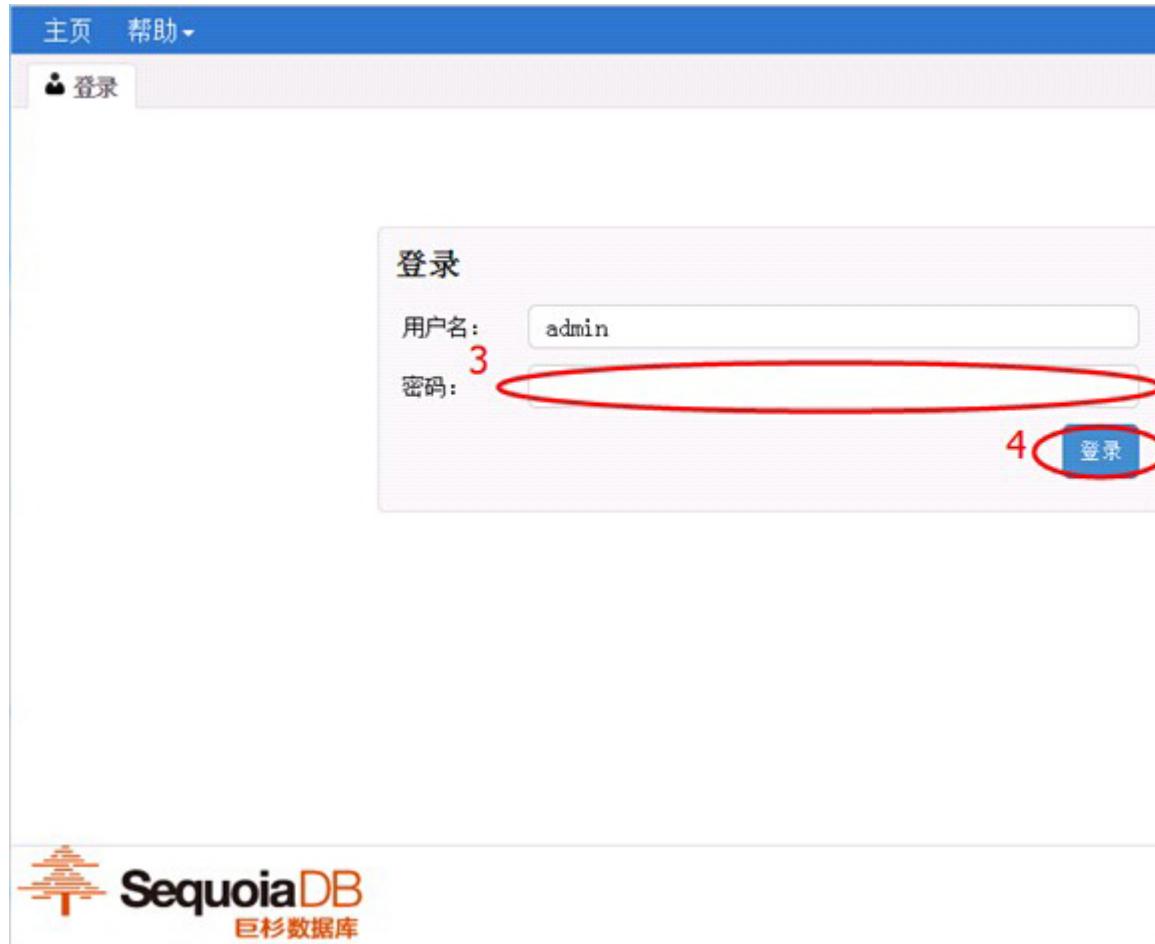
- 1.1 浏览器访问，访问地址为 OM 的地址，访问端口默认8000

- 1.2 例如 <http://192.168.1.100:8000/login.html>

- 1.3 登录用户名默认 admin，初始密码 admin

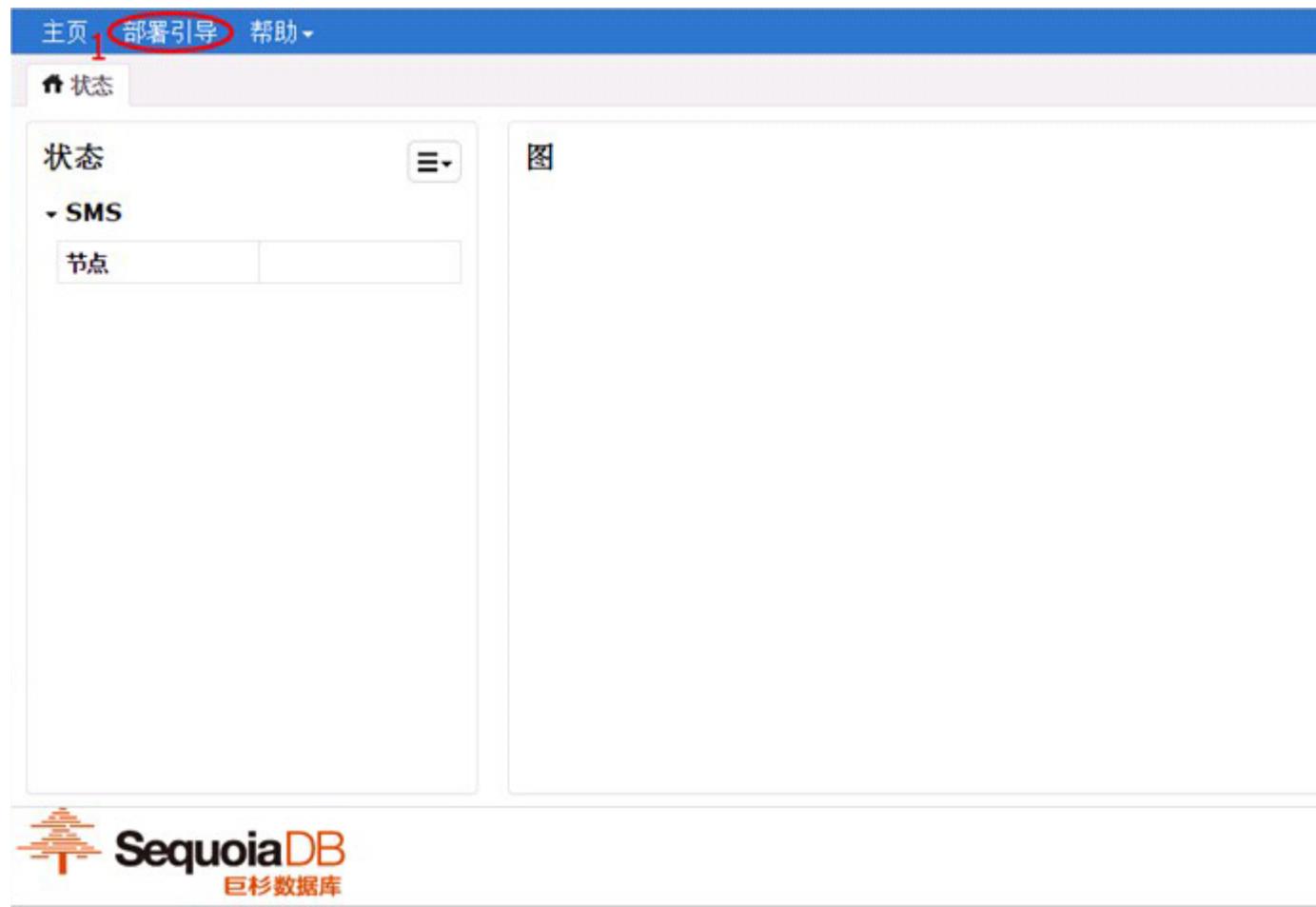
- 1.4 输入默认密码 admin

- 1.5 点击 <登录> 按钮



- 步骤二：开始部署

- 2.1 点击导航栏的 <部署引导>



2.2 集群的参数都有默认值，除了<集群名>和<业务名>，基本上不需要修改其他参数，点击<高级选项>，可以设置更多的参数。修改后点击<确定>。



- 步骤三：添加主机
- 3.1 在左边操作栏，输入 <地址>、<用户名>、<密码>、<SSH 端口>，可以扫描出添加的主机。<地址> 不仅支持 IP、HostName 扫描主机，还可以通过 IP 段和 HostName 段，<地址> 可以参考左边操作栏的提示，点击 <帮助>。



3.2 在左边操作栏，输入 <地址>、<用户名>、<密码>、<SSH 端口>，然后点击 <扫描>。

主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

扫描主机

提示：使用IP段或host段可以批量添加主机。详情请点击[帮助](#)。

地址：

用户名：

密码：

SSH端口：

代理端口：

[扫描](#)

主机名	地址	用户名	密码	SSH端口	代理端口	状态

 SequoiaDB
巨杉数据库

[返回](#) [下一步](#)

3.3 扫描完成后，扫描结果会在右边显示出来，默认正常连接的主机都会选上，有问题的主机不会选择，点击有问题的主机，可以重新输入参数，点击输入框以外的地方，自动重新扫描主机。

主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

扫描主机

提示：使用IP段或host段可以批量添加主机。详情请点击帮助。

地址：
用户名：
密码：
SSH端口：
代理端口：

扫描

主机列表

主机名	地址	用户名	密码	SSH端口	代理端口	状态
ubuntu-test-01	192.168.1.215	root	***	22	117	连接成功
ubuntu-test-02	192.168.1.212	root	***	22	117	连接成功
ubuntu-test-03	192.168.1.213	root	***	22	117	SSH连接失败



SequoiaDB
巨杉数据库

返回 下一步

主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

扫描主机

提示：使用IP段或host段可以批量添加主机。详情请点击帮助。

地址：

用户名：

密码：

SSH端口：

代理端口：

主机列表

主机名	地址	用户名	密码	SSH端口	代理端口	状态
<input checked="" type="checkbox"/> ubuntu-test-01	192.168.1.215	root	***	22	11790	连接成功
<input checked="" type="checkbox"/> ubuntu-test-02	192.168.1.212	root	***	22	11790	连接成功
<input type="checkbox"/> ubuntu-test-03	192.168.1.213	root	1234	21	11791	SSH连接失败

修改好参数后，点击方框以外的地方



3.4 在主机列表选择要添加的主机，图中3台主机都添加，因此全选。点击 <下一步>。

扫描主机

提示：使用IP段或host段可以批量添加主机。详情请点击[帮助](#)。

地址：	
用户名：	root
密码：	
SSH端口：	22
代理端口：	11790

扫描

主机列表

主机名	地址	用户名	密码	SSH 端口	代理 端口	状态
<input checked="" type="checkbox"/> ubuntu-test-01	192.168.1.215	root	***	22	11790	连接成功
<input checked="" type="checkbox"/> ubuntu-test-02	192.168.1.212	root	***	22	11790	连接成功
<input checked="" type="checkbox"/> ubuntu-test-03	192.168.1.213	root	***	22	11790	连接成功

只有打勾的主机会添加到集群中，点击选择框，可以取消打勾

返回
下一步

The screenshot shows the SequoiaDB management interface. At the top, there is a navigation bar with links for '主页' (Home), '帮助' (Help), '扫描主机' (Scan Host), '添加主机' (Add Host), '配置业务信息' (Configure Service Information), '修改业务信息' (Modify Service Information), and '安装业务' (Install Service). Below the navigation bar, there is a search bar with the placeholder text '提示：输入IP或Host可以快速找到对应的主机。' (Tip: Enter IP or Host to quickly find the corresponding host.) and a search icon. To the right of the search bar, there is a '详细信息' (Detailed Information) section with tabs for '选择主机' (Select Host) and 'Host Name'. This section contains fields for 'IP', 'OS', 'Memory', and 'Install Pa21 %'. A circular callout highlights the 'Memory' field. Below this, there is an 'OM 代理' (OM Agent) section with tabs for 'OM Version', 'OM Path', and 'OM Port'. At the bottom left, the SequoiaDB logo is displayed with the text 'SequoiaDB 巨杉数据库'. At the bottom right, there are two buttons: '返回' (Back) and '下一步' (Next Step).

3.6 左边是主机列表栏，右边是主机的详细信息，点击左边列表的主机，可以显示该主机信息。

The screenshot shows the SequoiaDB management interface. At the top, there are tabs for '主页' (Home), '帮助' (Help), '扫描主机' (Scan Host), '添加主机' (Add Host), '配置业务信息' (Configure Service Information), '修改业务信息' (Modify Service Information), and '安装业务' (Install Service). Below these tabs, there is a search bar with the placeholder text '提示：输入IP或Host可以快速找到对应的主机。' (Tip: Enter IP or Host to quickly find the corresponding host.). A list of hosts is displayed, including:

- ubuntu-test-01** 192.168.1.215 (Status: 1 blue, 4 yellow)
- ubuntu-test-02** 192.168.1.212 (Status: 1 blue, 4 yellow)
- ubuntu-test-03** 192.168.1.213 (Status: 2 blue, 4 yellow)

To the right, there is a '详细信息' (Detailed Information) section with the following details:

HostName	ubuntu-test-01
IP	192.168.1.215
OS	Ubuntu 12.04 x64
Memory	421MB / 483MB
Install Path	/opt/sequoiadb/

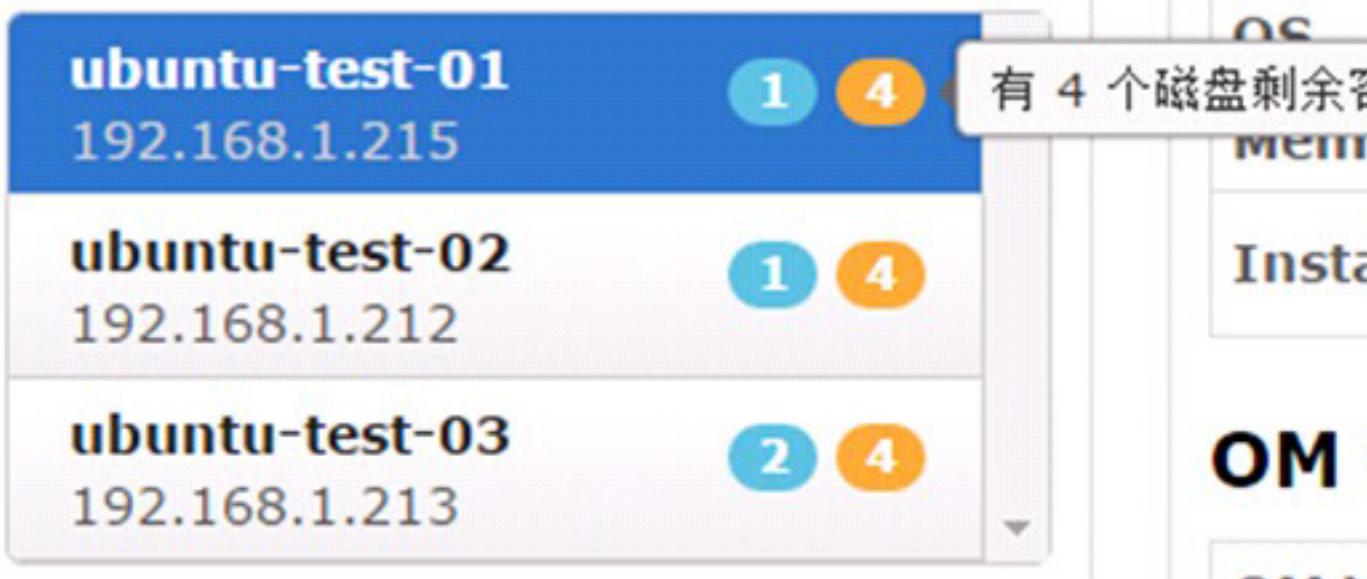
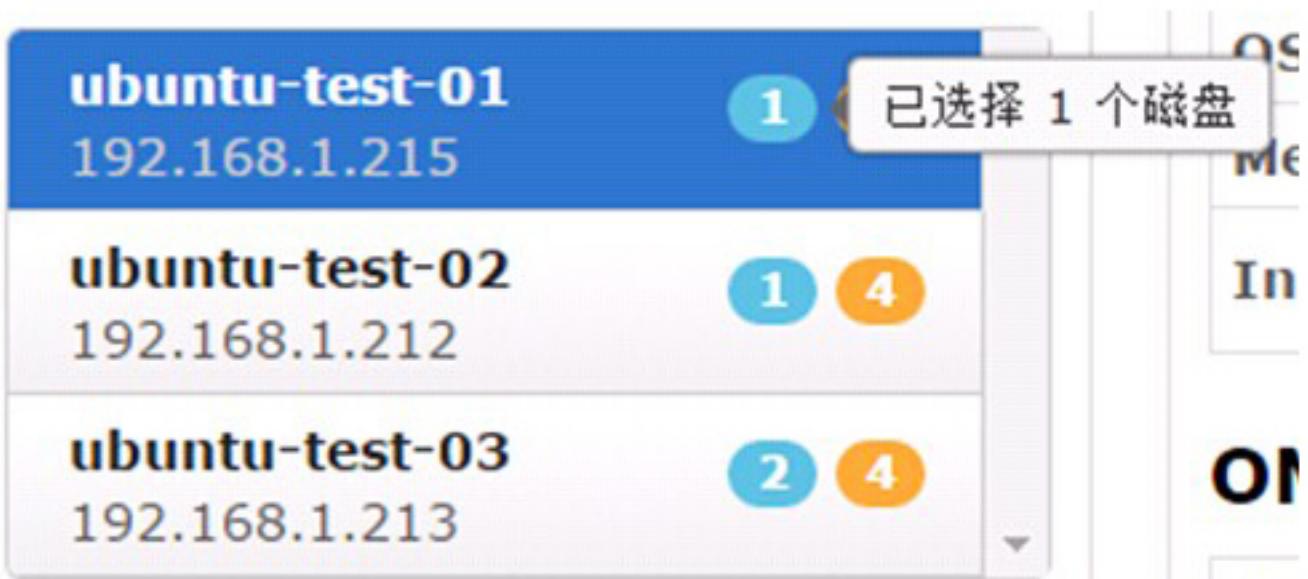
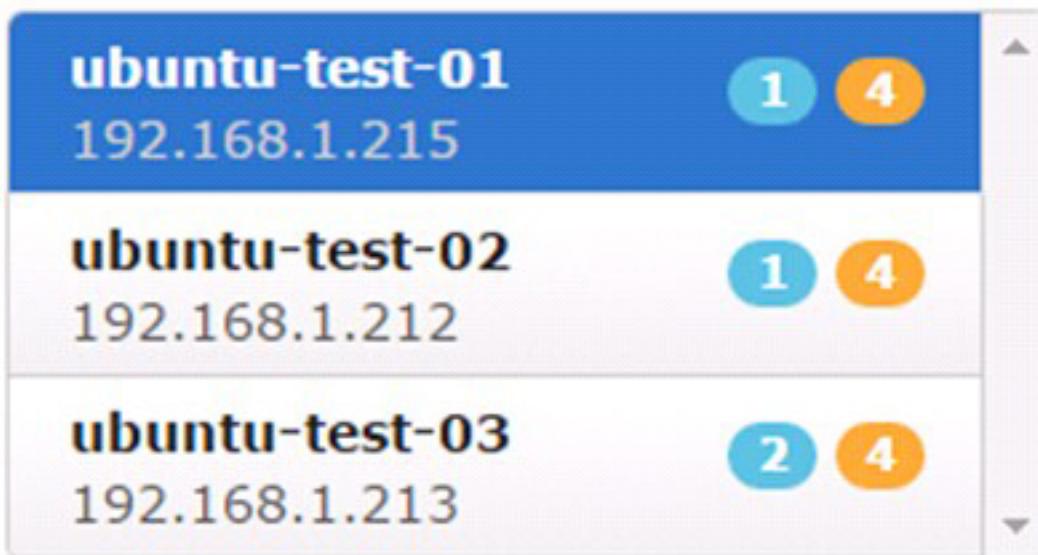
Below this is an 'OM 代理' (OM Agent) section with fields for OM Version, OM Path, and OM Port.

Underneath the host list is a '磁盘' (Disk) section with a table:

磁盘	路径	网络磁盘	容量
<input checked="" type="checkbox"/> /dev/	/	false	5767MB / 18049MB

At the bottom left is the SequoiaDB logo with the text '巨杉数据库'. On the right side, there are '返回' (Back) and '下一步' (Next Step) buttons.

3.7 主机列表，蓝色提示已选择的磁盘数，黄色提示该主机有4个警告，当鼠标移到提示中，会自动显示内容。（注意：图中3台主机都有1个磁盘是满足容量的）



3.8 检查主机没有问题，点击 <下一步>，等待完成。完成后会自动进入下一步操作。

The screenshot shows the SequoiaDB configuration interface. At the top, there are tabs: '主页' (Home), '帮助' (Help), '扫描主机' (Scan Host), '添加主机' (Add Host), '配置业务信息' (Configure Service Information), '修改业务信息' (Modify Service Information), and '安装业务' (Install Service). A search bar is also present.

A message box says: '提示：输入IP或Host可以快速找到对应的主机。' (Tip: Enter IP or Host to quickly find the corresponding host.)

The left sidebar lists three hosts:

- ubuntu-test-01 192.168.1.215 (status: 1 4)
- ubuntu-test-02 192.168.1.212 (status: 1 4)
- ubuntu-test-03 192.168.1.213 (status: 2 4)

The main area contains several sections:

- OM Version**: OM Path, OM Port
- 磁盘 4**: Shows disk usage for /dev/sda1 (5767MB / 18049MB), udev (1MB / 234MB), tmpfs (1MB / 98MB), none (0MB / 5MB), and none (0MB / 242MB).
- CPU**: Shows one core at 3.20GHz.
- SequoiaDB 巨杉数据库**: Includes a logo and the text 'SequoiaDB 巨杉数据库'.
- Buttons**: '返回' (Back) and '下一步' (Next Step), with '下一步' circled in red.

The screenshot shows the SequoiaDB installation interface. At the top, there are tabs for '扫描主机' (Scan Host), '添加主机' (Add Host), '配置业务信息' (Configure Service Information), '修改业务信息' (Modify Service Information), and '安装业务' (Install Service). The '添加主机' tab is selected.

In the center-left, there is a search bar with the placeholder text '提示：输入IP或Host可以快速找到对应的主机。' (Tip: Enter IP or Host to quickly find the corresponding host.) Below it is a list of hosts:

HostName	IP
ubuntu-test-01	192.168.1.215
ubuntu-test-02	192.168.1.212
ubuntu-test-03	192.168.1.213

To the right, under '详细信息' (Detailed Information), the host details are listed:

HostName	ubuntu-test-01
IP	192.168.1.215
OS	Ubuntu 12.04 x64
Memory	421MB / 483MB
Install Path	32% /opt/sequoiadb/

Below this, the 'OM 代理' (OM Agent) section includes fields for 'OM Version', 'OM Path', and 'OM Port'.

Under '磁盘' (Disk), there is a table:

磁盘	路径	网络磁盘	容量
<input checked="" type="checkbox"/> /dev/	/	false	5767MB / 18049MB

At the bottom right are '返回' (Back) and '下一步' (Next) buttons.

SequoiaDB 巨杉数据库

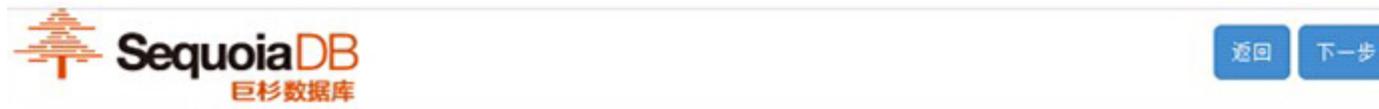
- 步骤四：创建业务

4.1 配置业务信息，可以通过下拉菜单选择 <集群模式> 和 <独立模式>。

SequoiaDB Configuration Interface Screenshot

The screenshot shows the 'Cluster Mode' configuration page. At the top, there are tabs: '扫描主机' (Scan Host), '添加主机' (Add Host), '配置业务信息' (Configure Service Information), '修改业务信息' (Modify Service Information), and '安装业务' (Install Service). The '配置业务信息' tab is active. Below it, there is a section titled '集群模式' (Cluster Mode) with a red circle around it. To its right is the '高级选项' (Advanced Options) button, also circled in red. A table below lists configuration properties:

属性	值	说明
副本数	3	数据拷贝数
分区组数	1	数据分区组的数量
编目节点数	3	编码节点的数量
协调节点数	0	协调节点的数量，填0则所有主机都安装一个协调节点



4.2 <高级选项> , 可以选择指定的1台或多台主机安装业务 , 默认由系统自动分配。

The screenshot shows the OpenTopic installation interface. At the top, there is a navigation bar with links: '主页' (Home), '帮助' (Help), '扫描主机' (Scan Host), '添加主机' (Add Host), '配置业务信息' (Configure Service Information), '修改业务信息' (Modify Service Information), and '安装业务' (Install Service). Below the navigation bar, there is a configuration section for '集群模式' (Cluster Mode) with a dropdown menu and a '高级选项' (Advanced Options) button. A table displays a configuration entry: '属性' (Property) '副本数' (Replica Count) '值' (Value) '3' and '说明' (Description) '数据拷贝数' (Data Copy Number). A modal dialog box titled '选择主机' (Select Host) is open. It contains a table with three host entries: 'ubuntu-test-01' (IP: 192.168.1.215), 'ubuntu-test-02' (IP: 192.168.1.212), and 'ubuntu-test-03' (IP: 192.168.1.213). To the right of the table, it says '共 3 台主机, 已选择 0 台主机' (3 hosts available, 0 hosts selected). At the bottom right of the dialog is a blue '关闭' (Close) button. The background of the main interface features the SequoiaDB logo and navigation buttons: '返回' (Back) and '下一步' (Next Step).

4.3 设置好参数后，点击 <下一步>。

The screenshot shows the 'Cluster Mode' configuration section. It includes a dropdown menu for 'Cluster Mode' and a 'Advanced Options' button. Below is a table with four rows:

属性	值	说明
副本数	3	数据拷贝数
分区组数	1	数据分区组的数量
编目节点数	3	编码节点的数量
协调节点数	0	协调节点的数量，填0则所有主机都安装一个协调节点



4.4 左边是业务和每一个分区组信息，可以通过业务添加分区组，通过分区组添加删除节点、删除分区组。（注意：协调组和编目组是不能删除的）

The screenshot shows the 'Configure Business Information' section. On the left, there's a sidebar with a message: '提示：配置业务信息使用说明，请点击帮助。' (Tip: Use the help link for configuration information). Below it are three groups: '协调组' (Coordinator Group) with 3 nodes, '编目组' (Catalog Group) with 3 nodes, and 'group1' with 3 nodes. On the right, there's a table for managing hosts and their roles across partition groups. The table has columns: 主机名 (Host Name), 端口 (Port), 数据路径 (Data Path), 角色 (Role), and 分区组 (Partition Group). The data in the table is as follows:

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/11810	coord	全部
ubuntu-test-01	11810	/sequoiadb/database/coord/11810	coord	全部
ubuntu-test-02	11810	/sequoiadb/database/coord/11810	coord	全部
ubuntu-test-03	11820	/opt/aaa/sequoiadb/database/catalog/11820	catalog	全部
ubuntu-test-01	11820	/sequoiadb/database/catalog/11820	catalog	全部
ubuntu-test-02	11820	/sequoiadb/database/catalog/11820	catalog	全部
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database/data	data	group1

主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

提示：配置业务信息使用说明，请点击帮助。

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/11810	coord	全部 全部
ubuntu-test-01	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-03	11820	/opt/aaa/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-01	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-02	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database/data	data	group1

业务：myBusiness
总节点数：9

协调组
节点数：3

编目组
节点数：3

group1
节点数：3

添加分区组

SequoiaDB 巨杉数据库

主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

提示：配置业务信息使用说明，请点击帮助。

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/11810	coord	全部 全部
ubuntu-test-01	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-03	11820	/opt/aaa/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-01	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-02	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database/data	data	group1

业务：myBusiness
总节点数：9

协调组
节点数：3

编目组
节点数：3

group1
节点数：3

添加节点
删除节点

SequoiaDB 巨杉数据库

业务: myBusiness
总节点数: 9

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-01	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-03	11820	/opt/aaa/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-01	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-02	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database	data	group1

4.5 右边是节点列表，点击列表每一行，都可以修改该行节点的配置。

业务: myBusiness
总节点数: 9

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-01	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-03	11820	/opt/aaa/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-01	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-02	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database	data	group1

4.6 批量修改节点配置，可以在节点列表每一行的方框打勾来指定要修改的节点，也可以点击 <选择操作> - <全选> - <反选> 来选择，然后点击 <已选定操作> - <修改节点配置>进行修改。

业务: myBusiness
总节点数: 9

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/ 11810	coord	
ubuntu-test-01	11810	/sequoiadb/database/coord/ 11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/ 11810	coord	
ubuntu-test-03	11820	/opt/aaa/sequoiadb/databa se/catalog/11820	catalog	
ubuntu-test-01	11820	/sequoiadb/database/catalo g/11820	catalog	
ubuntu-test-02	11820	/sequoiadb/database/catalo g/11820	catalog	
ubuntu-test-03	11830	/opt/aaa/sequoiadb/databa se/catalog/11830	data	group1

4.7 节点列表第二行是查找条件，通过输入参数，可以快速找到指定条件的节点。

业务: myBusiness
总节点数: 9

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/ 11810	coord	
ubuntu-test-01	11810	/sequoiadb/database/coord/ 11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/ 11810	coord	

The screenshot shows two instances of the SequoiaDB configuration interface. In both instances, the left panel displays a business named 'myBusiness' with 9 nodes. The right panel shows a table for assigning roles to hosts.

Top Screenshot (Host Role Assignment):

主机名	端口	数据路径	角色	分区组
01	11810		全部	全部
ubuntu-test-01	11810	/sequoiadb/database/coord/ 11810	coord	

Bottom Screenshot (Data Node Assignment):

主机名	端口	数据路径	角色	分区组
			data	全部
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database/data/11830	data	group1
ubuntu-test-01	11830	/sequoiadb/database/data/ 11830	data	group1
ubuntu-test-02	11830	/sequoiadb/database/data/ 11830	data	group1

In both screenshots, the 'roles' column is highlighted with a green box, indicating the field where roles are selected. In the bottom screenshot, the 'data' role is selected for all three hosts.

4.8 批量修改节点配置，<数据路径> 和 <服务名> 可以通过规则来设置。点击左边 <提示> - <帮助>，有详细说明。

业务：myBusiness 总节点数：9

协调组 节点数：3

编目组 节点数：3

group1 节点数：3

帮助

关于[已选定操作]-[修改节点配置]。您可以使用特殊规则来批量修改节点的服务名和数据路径：

服务名规则	规则：服务名[+步进]或服务名[-步进]。
普通方式	11810
递增方式	11810[+10]
递减方式	11810[-10]

数据路径规则

规则：可以在路径中任意添加这几个特殊命令，**[role]**--角色，**[svcname]**--服务名，**[groupname]**--分区组名，**[hostname]**--主机名

例子

/opt/sequoiadb/[role]/[svcname]/[groupname]/[hostname]

假设已选定节点配置为：角色：data，服务名：11810，分区组：DATAGROUP1，主机：pcHost1，最终设置的数据路径：/opt/sequoiadb/data/11810/DATAGROUP1/pcHost1，注意：协调节点和编目节点是没有分区组名的，因此 [groupname] 是空字符串

4.9 业务配置修改完成后，点击 <下一步>，开始安装业务。

业务：myBusiness

总节点数：9

主机名	端口	数据路径	角色	分区组
ubuntu-test-03	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-01	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-02	11810	/sequoiadb/database/coord/11810	coord	
ubuntu-test-03	11820	/opt/aaa/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-01	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-02	11820	/sequoiadb/database/catalog/11820	catalog	
ubuntu-test-03	11830	/opt/aaa/sequoiadb/database/data/11830	data	group1
ubuntu-test-01	11830	/sequoiadb/database/data/11830	data	group1

- 步骤五：安装业务

5.1 开始安装业务

业务：myBusiness 正在安装...

共 0 个项目，已成功完成 0 个。

项目	节点数	进度	状态	日志
----	-----	----	----	----



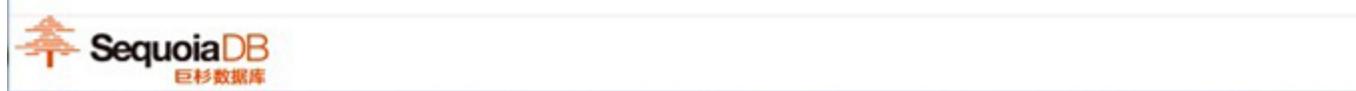
主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

业务: myBusiness 正在安装...

共 3 个项目, 已成功完成 0 个。

项目	节点数	进度	状态	日志
Catalog	3	0%	Installing catalog[ubuntu-test-03: 11820]	日志
Coord	3	0%		日志
group1	3	0%		日志



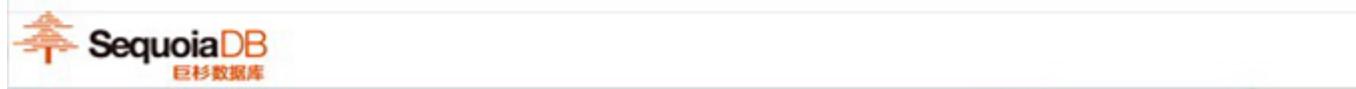
主页 帮助 ▾

扫描主机 添加主机 配置业务信息 修改业务信息 安装业务

业务: myBusiness 正在安装...

共 3 个项目, 已成功完成 0 个。

项目	节点数	进度	状态	日志
Catalog	3	33%	Installing catalog[ubuntu-test-01: 11820]	日志
Coord	3	0%		日志
group1	3	0%		日志



业务: myBusiness 正在安装...

共 3 个项目, 已成功完成 2 个。

项目	节点数	进度	状态	日志
Catalog	3	100%	✓	日志
Coord	3	100%	✓	日志
group1	3	0%	Installing data node[ubuntu-test-03:11830]	日志

5.2 安装完成，点击 <返回>。

业务: myBusiness 安装完成...

共 3 个项目, 已成功完成 3 个。

项目	节点数	进度	状态	日志
Catalog	3	100%	✓	日志
Coord	3	100%	✓	日志
group1	3	100%	✓	日志

安装结果

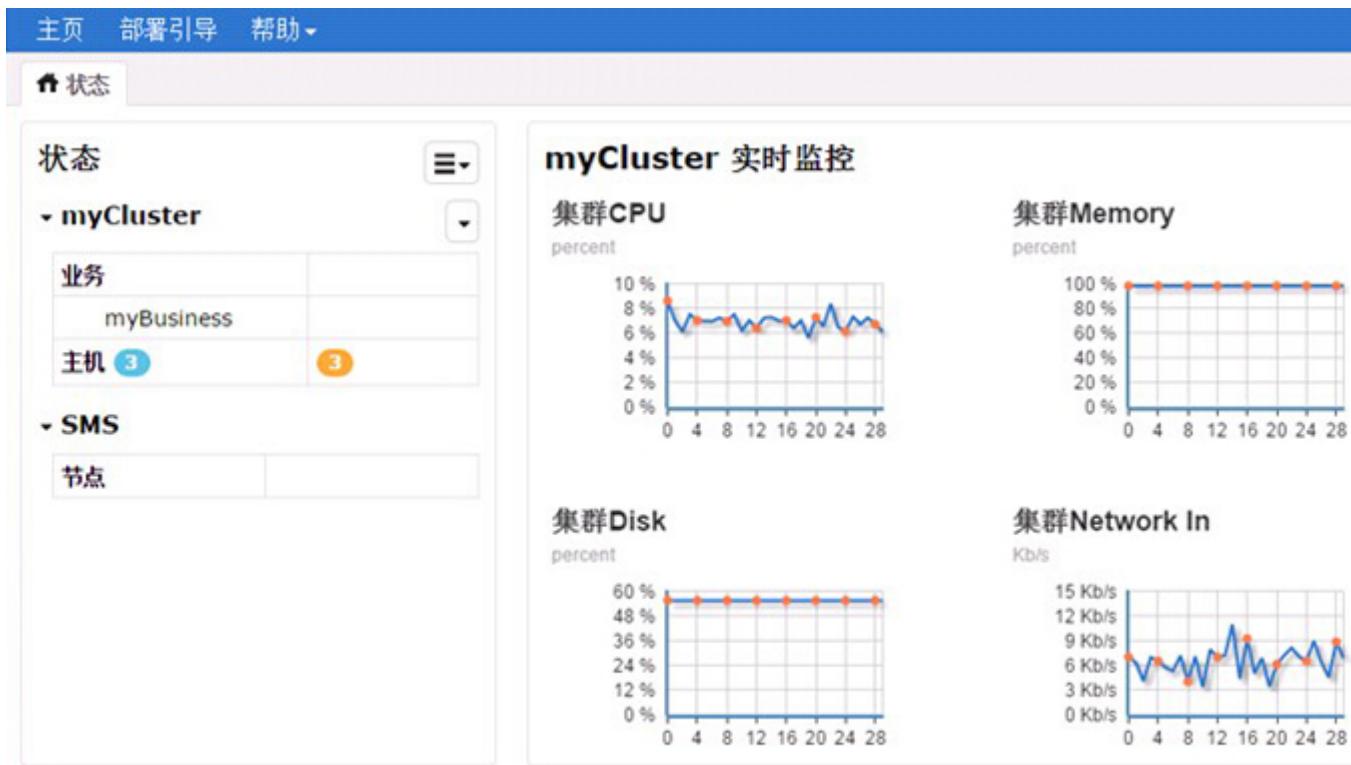
业务安装完成, 系统将会在10秒后返回。或点击下方按钮马上返回。

返回

5.3 业务安装完成。



- 步骤六：主页左边是集群列表，右边是集群实时监控信息



- 步骤七：创建集群

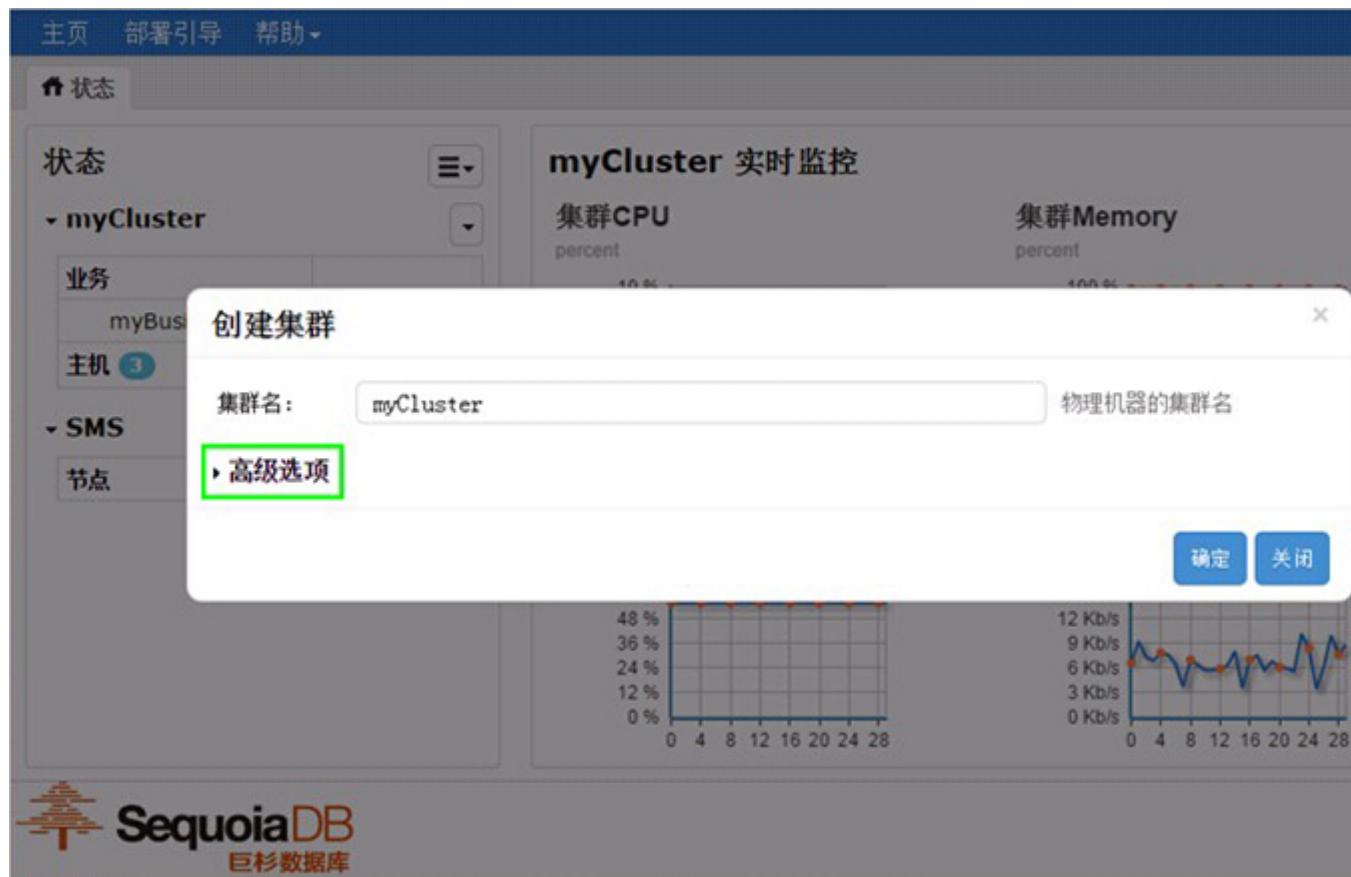
7.1 点击 <状态> 旁边的下拉菜单；



7.2 点击 <创建集群>；



7.3 输入集群参数，点击 <高级选项> 可以设置更多参数；

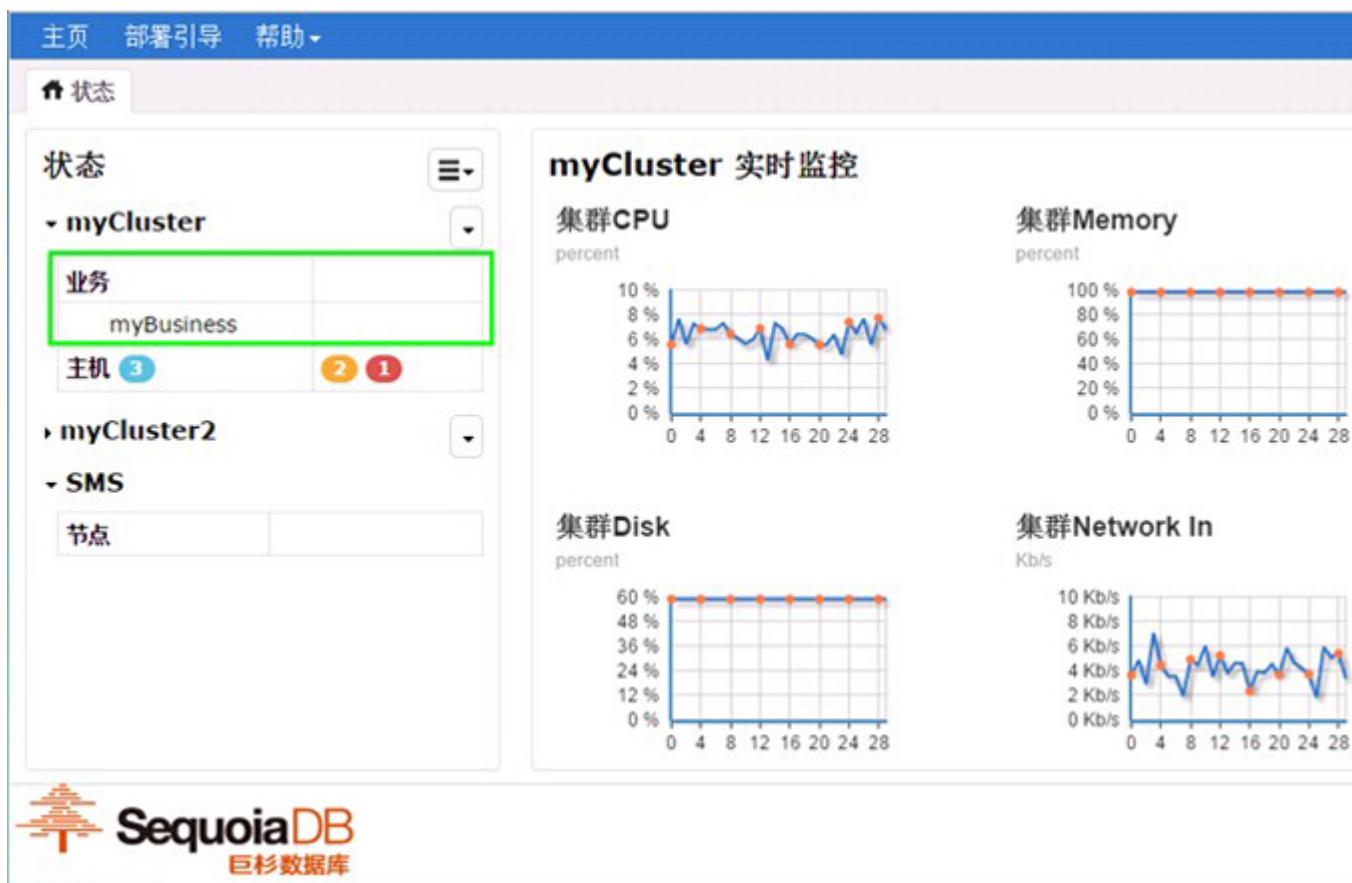


7.4 点击 <确定> , 完成创建。



- 步骤八：查看业务信息和删除业务

8.1 查看当前已安装的业务（图中 myBusiness 是已经安装的一个业务，myBusiness 是业务名）；



8.2 点击表格 <业务> 查看业务信息（一）；

主页 部署引导 帮助 ▾

状态

集群CPU
percent

小时	CPU 使用率 (%)
0	7
4	6
8	7
12	6
16	7
20	6
24	8
28	6

集群Memory
percent

小时	内存使用率 (%)
0	100
4	100
8	100
12	100
16	100
20	100
24	100
28	100

集群Disk
percent

小时	磁盘使用率 (%)
0	60
4	60
8	60
12	60
16	60
20	60
24	60
28	60

集群Network In
Kb/s

小时	网络输入 (Kb/s)
0	4
4	5
8	4
12	5
16	4
20	5
24	6
28	4

SequoiaDB 巨杉数据库

主页 帮助 ▾

业务列表

集群: myCluster

添加业务

业务名	业务类型	部署模式	操作
myBusiness	sequoiadb	distribution	删除业务

SequoiaDB 巨杉数据库

8.3 点击业务名旁边的下拉菜单，点击 <业务列表> 看业务信息（二）；



主页 部署引导 帮助 ▾

状态

myCluster

业务	myBusiness
主机	3 (2 1)

myCluster2

SMS

节点

集群CPU

- 添加主机
- 主机列表
- 添加业务
- 业务列表** (highlighted with red oval)
- 删除集群

集群Memory percent

集群Network In Kbps

SequoiaDB 巨杉数据库

主页 帮助 ▾

业务列表

集群: myCluster

添加业务

业务名	业务类型	部署模式	操作
myBusiness	sequoiadb	distribution	删除业务

SequoiaDB 巨杉数据库

8.4 删除业务，在业务列表，点击业务对应的 <删除业务> 按钮，弹出警告窗口，点击 <确定> 开始删除业务，等待卸载完成，完成后弹出提示，点击 <返回>。



集群: myCluster

业务名	业务类型	部署模式	操作
myBusiness	sequoiadb	distribution	<button>删除业务</button>

 **SequoiaDB**
巨杉数据库

主页 帮助 ▾

业务列表

集群: myCluster

添加业务

业务名	业务类型	部署模式	操作
myBusiness	删除业务		X

警告: 该操作是不可恢复操作, 并且不会保留该业务数据。

确定 关闭

 SequoiaDB
巨杉数据库

主页 帮助 ▾

卸载业务

业务: myBusiness 正在卸载...

共 0 个项目, 已成功完成 0 个。

项目	节点数	进度	状态	日志
----	-----	----	----	----

 SequoiaDB
巨杉数据库

主页 帮助 ▾

卸载业务

业务: **myBusiness** 正在卸载...

共 3 个项目, 已成功完成 0 个。

项目	节点数	进度	状态	日志
Catalog	1	0%		日志
Coord	1	0%		日志
group1	1	0%		日志

 SequoiaDB
巨杉数据库

主页 帮助 ▾

卸载业务

业务: myBusiness 正在卸载...

共 3 个项目, 已成功完成 2 个。

项目	节点数	进度	状态	日志
Catalog	1	0%	Removing catalog group	日志
Coord	1	100%	✓	日志
group1	1	100%	✓	日志

 SequoiaDB
巨杉数据库

主页 帮助 ▾

卸载业务

业务: myBusiness 卸载完成...

共 3 个项目, 已成功完成 3 个。

项目	节点数	进度	状态	日志
卸载结果				

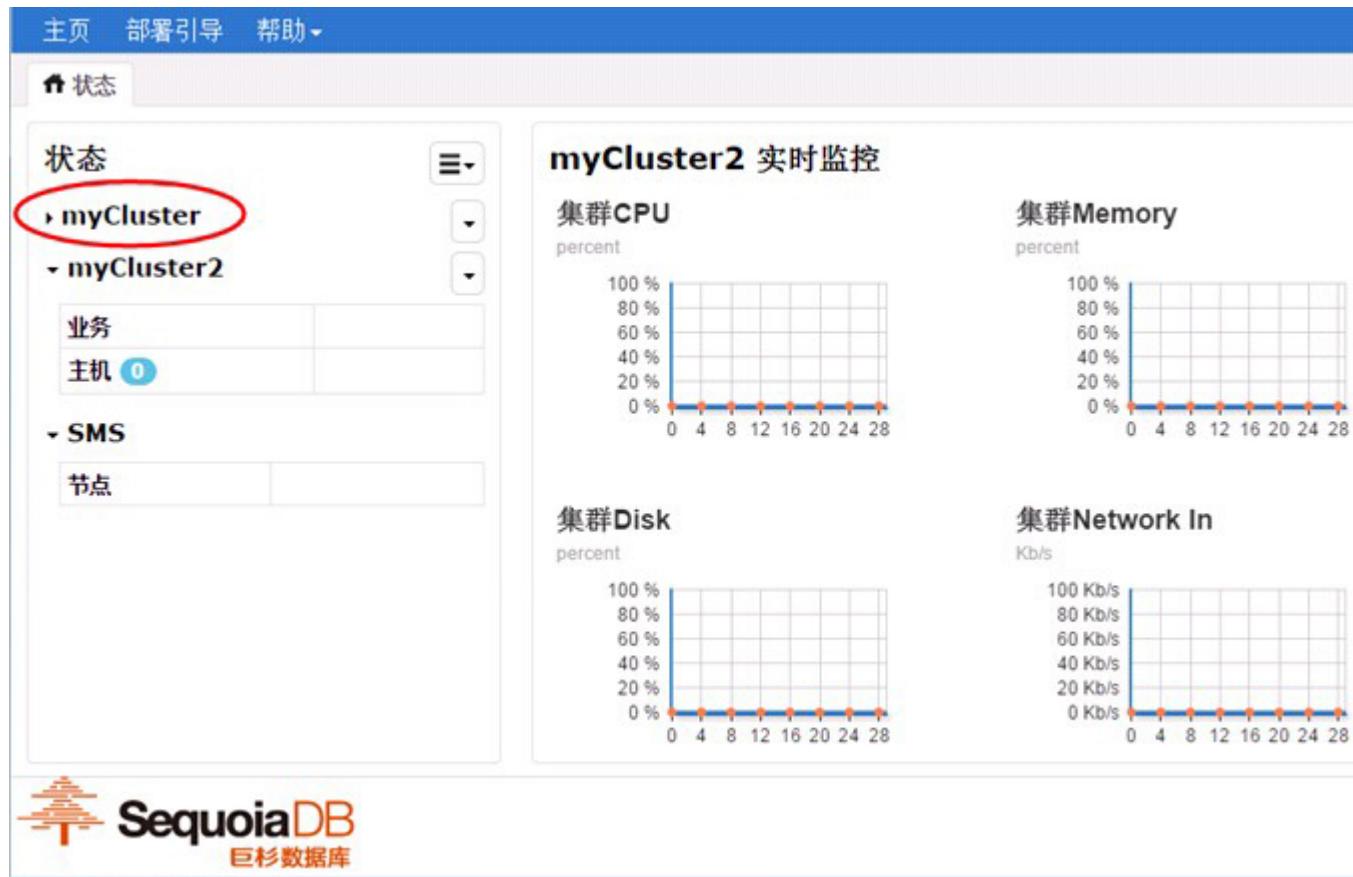
业务卸载完成, 系统将会在10秒后返回。或点击下方按钮马上返回。

[返回](#)

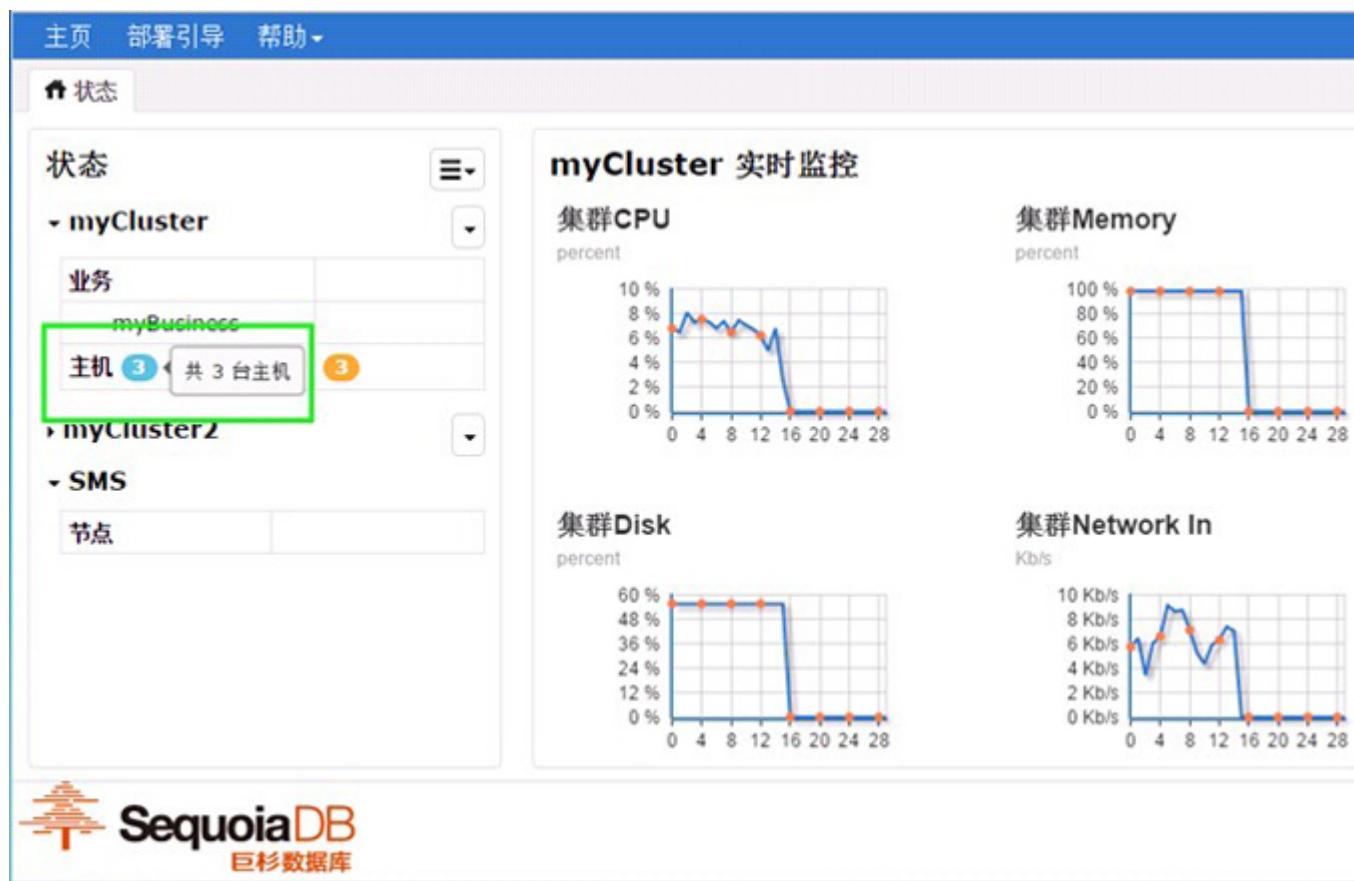
 SequoiaDB
巨杉数据库

- 步骤九：查看集群主机数量和主机详细信息，删除主机

9.1 点击要查看的集群名；



9.2 鼠标移动到主机旁边蓝色的徽章，徽章上面的数字就是当前主机数量；



9.3 表格第二列是显示当前状态信息，黄色表示 warning，红色表示 danger；

主页 部署引导 帮助 ▾

状态

状态

- myCluster

业务
myBusiness

主机 3

3 有 3 台主机的内存达到 90% 以上

myCluster2

SMS

节点

myCluster 实时监控

集群CPU
percent

集群Memory
percent

集群Disk
percent

集群Network In
Kb/s

SequoiaDB 巨杉数据库

主页 部署引导 帮助 ▾

状态

状态

- myCluster

业务
myBusiness

主机 3

2 1 有 1 台主机故障

myCluster2

SMS

节点

myCluster 实时监控

集群CPU
percent

集群Memory
percent

集群Disk
percent

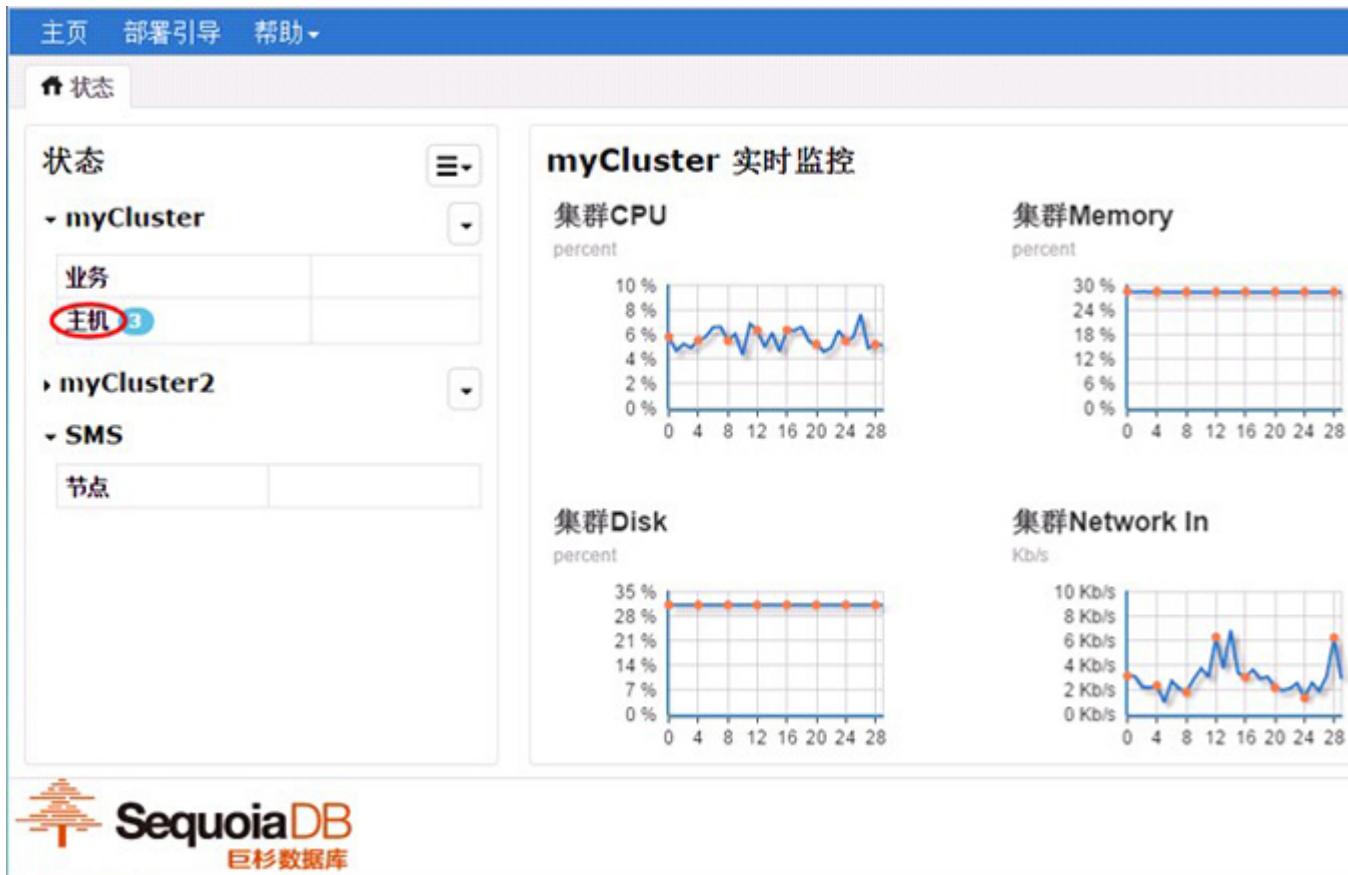
集群Network In
Kb/s

SequoiaDB 巨杉数据库

9.4 主页右边的 <实时监控>，监控集群的平均 CPU 使用率，平均内存使用率，平均磁盘使用率，网络入口流量，网络出口流量，网络每秒接收数据包个数，网络每秒发送数据包个数；



9.5 点击表格 <主机>，查看主机列表；



集群: myCluster

选择操作	已选定操作	添加主机						
状态	主机	IP	安装路径	OM版本	代理端口	系统	CPU核心数	内存
<input type="checkbox"/>	✓ ubuntu-test-01	192.168.1.215	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input type="checkbox"/>	✓ ubuntu-test-02	192.168.1.212	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input type="checkbox"/>	✓ ubuntu-test-03	192.168.1.213	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	

SequoiaDB 巨杉数据库

9.6 点击业务名旁边的下拉菜单，点击 <主机列表> 查看主机列表；

主站 部署引导 帮助 ▾

状态

myCluster ▾ (3)

业务 主机

myCluster2

SMS

节点

myCluster 实时监控

集群CPU percent

集群Memory percent

集群Disk percent

集群Network In Kb/s

SequoiaDB
巨杉数据库

The screenshot shows the SequoiaDB management interface. On the left, there's a sidebar with sections for '状态' (Status), 'myCluster' (with a dropdown menu circled in red), 'myCluster2', and 'SMS'. Under 'myCluster', there are tabs for '业务' (Services) and '主机' (Hosts), with a count of 3. On the right, there are four monitoring charts: '集群CPU' (Cluster CPU) showing usage fluctuating between 4% and 6%; '集群Memory' (Cluster Memory) showing usage constant at about 28%; '集群Disk' (Cluster Disk) showing usage constant at about 28%; and '集群Network In' (Cluster Network In) showing input fluctuating between 0 and 2 Kb/s. At the bottom, the SequoiaDB logo and name are displayed.

主页 部署引导 帮助 ▾

状态

myCluster

业务	
主机	3

myCluster2

SMS

节点	
----	--

集群CPU

添加主机
主机列表

集群Memory
percent

集群Network In
Kb/s

SequoiaDB 巨杉数据库

主页 帮助 ▾

状态

集群: myCluster

选择操作 ▾ 已选定操作 ▾ 添加主机

状态	主机	IP	安装路径	OM版本	代理端口	系统	CPU核心数	内存
<input type="checkbox"/>	ubuntu-test-01	192.168.1.215	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input type="checkbox"/>	ubuntu-test-02	192.168.1.212	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input type="checkbox"/>	ubuntu-test-03	192.168.1.213	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	

SequoiaDB 巨杉数据库

9.7 删除主机，选择要删除的主机，点击 <已选定操作>，点击 <删除主机>，弹出警告窗口，点击 <确定> 开始删除主机，等待完成，完成后弹出删除结果，点击 <关闭>。



The screenshot shows the SequoiaDB management interface with the following details:

- 集群: myCluster**
- 操作按钮:** 选择操作 ▾, 已选定操作 ▾, 添加主机
- 主机列表:** 显示了三台主机的信息，每行都有一个选中框（包含勾选图标）和一个绿色的对勾图标。第一列是选中框，第二列是状态，第三列是主机名，第四列是IP地址，第五列是安装路径，第六列是OM版本，第七列是代理端口，第八列是系统，第九列是CPU核心数，第十列是内存。

状态	主机	IP	安装路径	OM版本	代理端口	系统	CPU核心数	内存
<input checked="" type="checkbox"/>	ubuntu-test-01	192.168.1.215	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input checked="" type="checkbox"/>	ubuntu-test-02	192.168.1.212	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input checked="" type="checkbox"/>	ubuntu-test-03	192.168.1.213	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
- SequoiaDB 巨杉数据库** 标识

主页 帮助 ▾

☰ 状态

集群: myCluster

选择操作 ▾ 已选定操作 ▾ 添加主机

状态	主机	IP	安装路径	OM版本	代理端口	系统	CPU核心数	内存
<input checked="" type="checkbox"/> ✓	ubuntu-test-01	192.168.1.215	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input checked="" type="checkbox"/> ✓	ubuntu-test-02	192.168.1.212	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	
<input checked="" type="checkbox"/> ✓	ubuntu-test-03	192.168.1.213	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB	

 SequoiaDB
巨杉数据库

主页 帮助 ▾

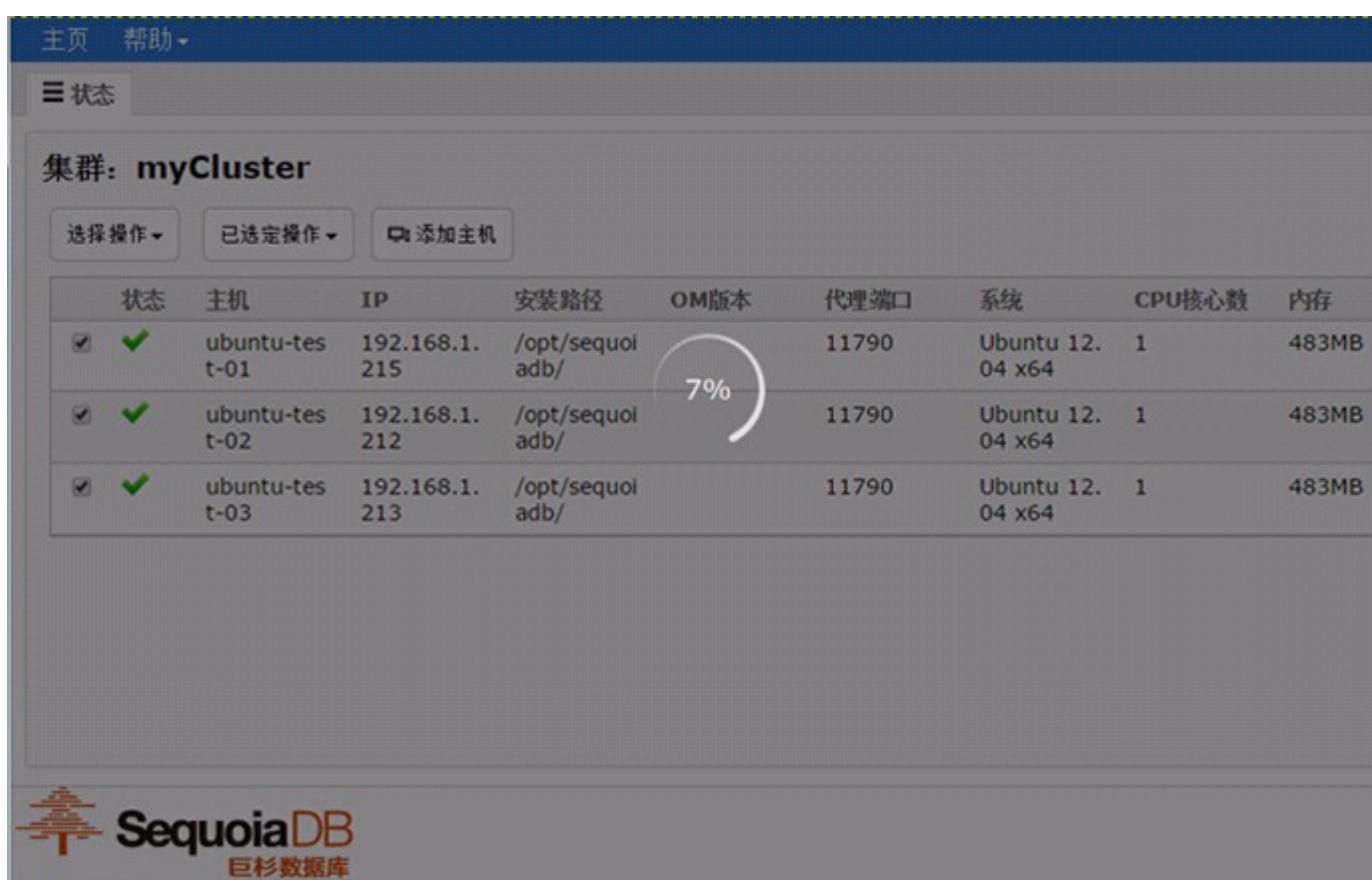
☰ 状态

集群: myCluster

选择操作 ▾ 已选定操作 ▾ 添加主机

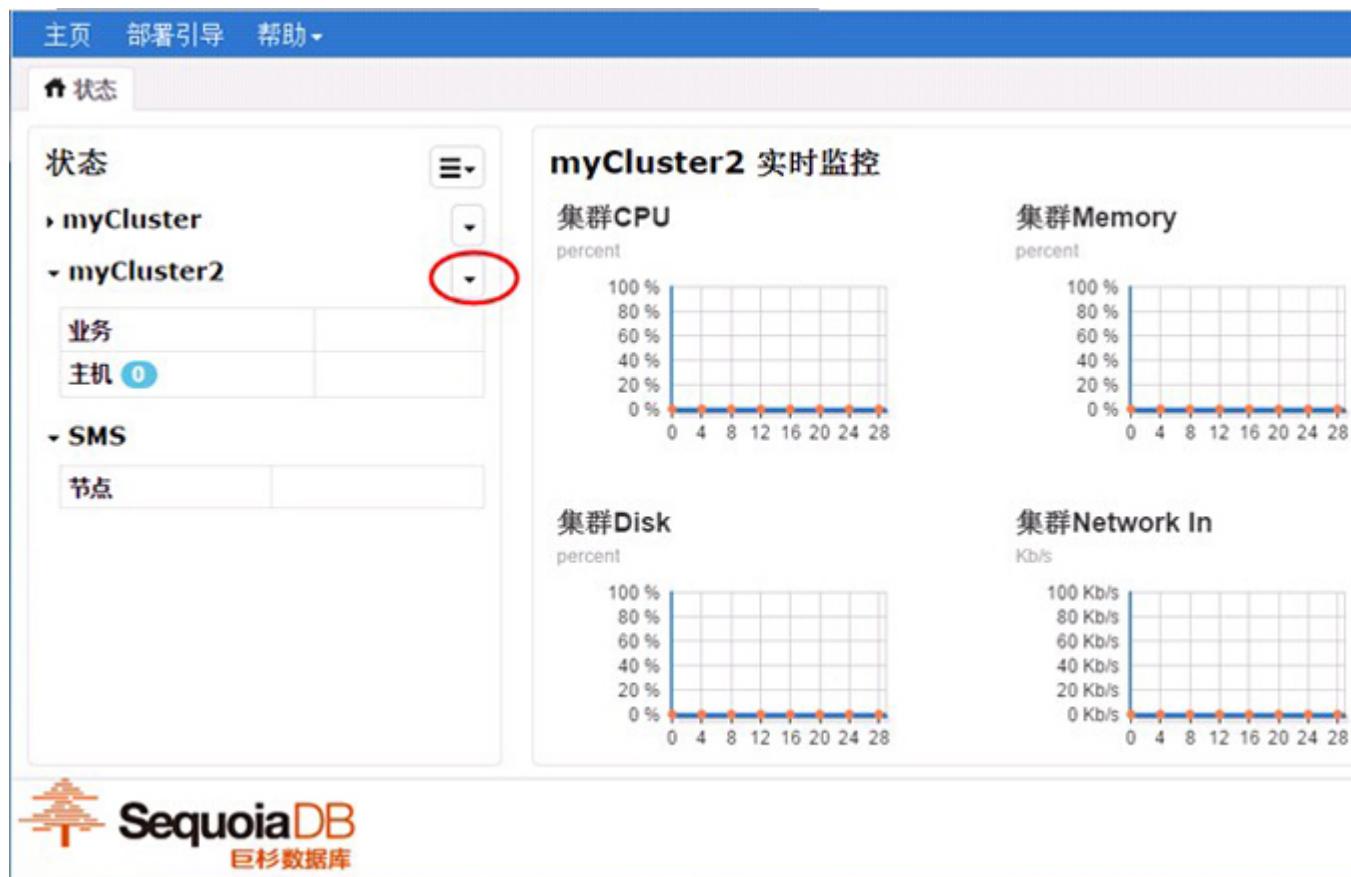
状态	删除主机	安装路径	OM版本	代理端口	系统	CPU核心数	内存
<input checked="" type="checkbox"/> ✓	ubuntu-test-01	192.168.1.215	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB
<input checked="" type="checkbox"/> ✓	ubuntu-test-02	192.168.1.212	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB
<input checked="" type="checkbox"/> ✓	ubuntu-test-03	192.168.1.213	/opt/sequoiadb/	11790	Ubuntu 12.04 x64	1	483MB

 SequoiaDB
巨杉数据库





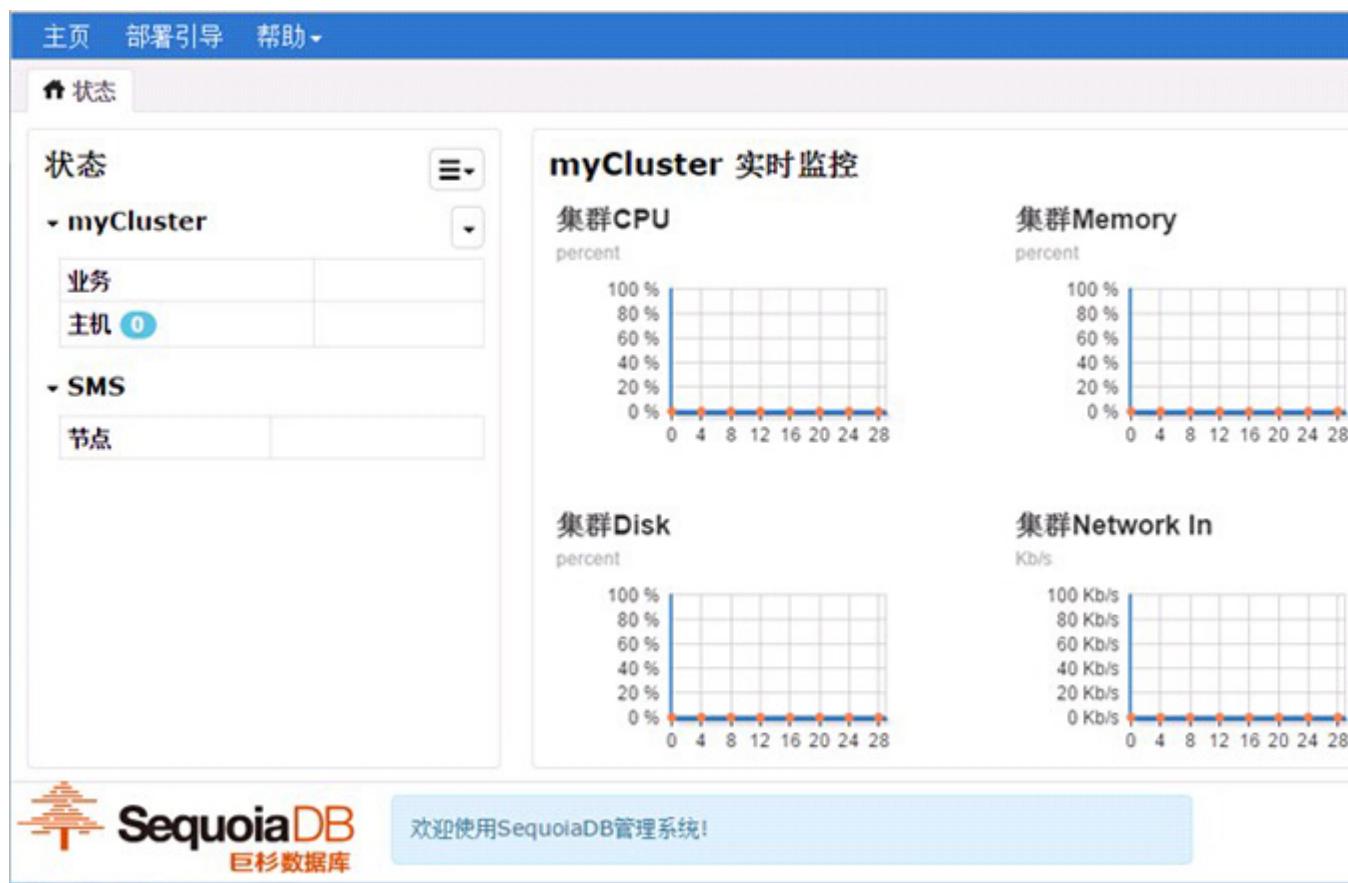
- 步骤十：删除集群
- 10.1 回到主页，点击要删除的集群旁边下拉菜单；



10.2 点击菜单的 <删除集群> ;



10.3 删除完成。



手工安装部署

手工安装需要先分别在每一台主机上安装 SequoiaDB，然后再手工部署集群。

[安装 SequoiaDB](#)

[手工部署集群](#)

[安装 SequoiaDB](#)

安装前准备

- 确保系统满足硬件和软件要求
- 使用 root 用户权限来安装 SequoiaDB 数据库服务
- 检查 SequoiaDB 产品软件包与 OS 系统配套
- 如果需要图形界面模式安装，请确保 X Server 服务正在运行
- 服务器配置了主机名，且与其他服务器之间可通过主机名建立网络连接（如 ssh 主机名）

注: SequoiaDB 的安装向导需要的参数不接受非英文字符。

安装步骤

说明：

- (1) 产品包名字以 sequoiadb-1.0.0-linux-x86_64-installer.run 为例；
- (2) 步骤以命令行方式进行介绍，图形界面按照图像向导提示完成。



注: 如果有多台服务器, 每台机器都需要重复如下步骤安装服务器程序。

- 参照[系统配置需求](#)配置好主机名以及修改系统内核参数
- 运行安装程序

```
./sequoiadb-1.0.0-linux-x86_64-installer.run --mode text --SMS false
```

- 程序提示选择向导语言

```
Language Selection
Please select the installation language
[1] English - English
[2] Simplified Chinese - 简体中文
Please choose an option [1] :2
```

- 输入2, 选择中文, 显示安装协议, 默认忽略阅读, 如果需要读取全部文件, 输入2

由 BitRockInstallBuilder 评估本所建立

欢迎来到 SequoiaDB Server 安装程序

重要信息: 请仔细阅读

下面提供了两个许可协议。

1. SequoiaDB 评估程序的最终用户许可协议
2. SequoiaDB 最终用户许可协议

如果被许可方为了生产性使用目的(而不是为了评估、测试、试用“先试后买”或演示)获得本程序, 单击下面的“接受”按钮即表示被许可方接受 SequoiaDB 最终用户许可协议, 且不作任何修改。

如果被许可方为了评估、测试、试用“先试后买”或演示(统称为“评估”)目的获得本程序: 单击下面的“接受”按钮即表示被许可方同时接受(i) SequoiaDB 评估程序的最终用户许可协议(“评估许可”), 且不作任何修改; 和(ii) SequoiaDB 最终用户程序许可协议(SELA), 且不作任何修改。

在被许可方的评估期间将适用“评估许可”。

如果被许可方通过签署采购协议在评估之后选择保留本程序(或者获得附加的本程序副本供评估之后使用), SequoiaDB 评估程序的最终用户许可协议将自动适用。

“评估许可”和 SequoiaDB 最终用户许可协议不能同时有效; 两者之间不能互相修改, 并且彼此独立。

这两个许可协议中每个协议的完整文本如下。

评估程序的最终用户许可协议

- [1] 同意以上协议: 了解更多的协议内容, 可以在安装后查看协议文件
[2] 查看详细的协议内容

请选择选项 [1] :

- 是否同意协议:

同意以上协议

按 [Enter] 继续:

您是否接受此软件授权协议? [y/n]:

- 按 y 表示同意:

请指定 SequoiaDBServer 将会被安装到的目录
安装目录 [/opt/sequoiadb]:

- 输入安装路径后按回车（默认安装在 /opt/sequoiadb），此时系统提示输入用户名，该用户名用于运行 SequoiaDB 服务

数据库管理用户配置

配置用于启动 SequoiaDB 的用户名和密码

用户名 [sdbadmin]:

- 输入用户名后按回车（默认创建 sdbadmin 用户），此时系统提示输入该用户的密码和确认密码

密码 [*****] :

确认密码 [*****] :

- 输入两次密码后（默认密码为 sdbadmin），此时系统提示输入配置服务端口

集群管理服务端口配置

配置SequoiaDB集群管理服务端口，集群管理用于远程启动添加和启停数据库节点

端口 [11790]:

 注: 所有服务器的配置服务端口必须相同。

- 输入端口（默认为11790），系统提示开始安装，需要用户确认
- 询问是否允许 SequoiaDB 相关进程开机自启动

是否允许 SequoiaDB 相关进程开机自启动

- SequoiaDB 相关进程开机自启动 [Y/n] : Y，输入 Y，按回车，同意 SequoiaDB 相关进程开机自启动

正在安装 SequoiaDB Server 于您的电脑中，请稍候。

安装中

0% ----- 50% ----- 100%
#####
#####

安装程序已经完成安装 SequoiaDB Server 于你的电脑中。

手工部署集群

数据库有两种模式，用户可以根据需求选择其中一种进行安装部署：

独立模式的部署

集群模式的部署

独立模式的部署

说明：

- (1) 本节按照最简部署为例，介绍配置和启动步骤；
- (2) 以下操作步骤假设 SequoiaDB 程序安装在 /opt/sequoiadb 目录下；
- (3) sdb 服务进程全部以 sdbadmin 用户运行，请确保所有数据库目录都赋予 sdbadmin 读写权限。

- 切换到 sdbadmin 用户

su sdbadmin

- 启动 SequoiaDB Shell 控制台（下文以默认安装路径 /opt/sequoiadb 为例）

/opt/sequoiadb/bin/sdb

- 连接到本地的集群管理服务进程 sdbcm

var oma = new Oma ("localhost", 11790)

- 创建独立模式的数据节点

oma.createData(11810, "/opt/sequoiadb/database/standalone/11810")

 注: 其中11810为数据库服务端口名，为避免出现端口冲突等问题，切勿将数据库端口配置在随机端口范围以内。如：多数 Linux 默认随机端口范围为32768 ~ 61000，可将数据库端口配置在32767以下。

- 启动该节点

```
oma.startNode(11810)
```

- 数据库配置启动完成

集群模式的部署

 注：在配置集群模式时，请先确保服务器与主机名的映射关系正确，详细请参考[系统配置需求](#)，确保各节点之间能相互通信，将节点的防火墙关闭。

说明：

- (1) 本节按照高可用部署为例，介绍配置和启动步骤；
- (2) 以下操作步骤假设 SequoiaDB 程序安装在 /opt/sequoiadb 目录下；
- (3) sdb服务进程全部以 sdbadmin 用户运行，请确保所有数据库目录都赋予 sdbadmin 读写权限。

- 步骤一：检查 SequoiaDB 的配置服务状态

在每台数据库服务器上检查 SequoiaDB 配置服务状态：

```
service sdbcm status
```

确认系统提示“sdbcm is running”表示服务正在运行，否则请执行如下命令重新配置服务程序：

```
service sdbcm start
```

- 步骤二：启动一个临时协调节点（该节点只是为了创建其它节点而临时使用，后面会删除）

- 切换到 sdbadmin 用户

```
su sdbadmin
```

- 在任意一台数据库服务器上（以下步骤都只需要在这台服务器上操作），启动 SequoiaDB Shell 控制台 /opt/sequoiadb/bin/sdb

- 连接到本地的集群管理服务进程 sdbcm

```
var oma = new Oma("localhost", 11790)
```

- 创建临时协调节点

```
oma.createCoord(18800, "/opt/sequoiadb/database/coord/18800")
```

- 启动临时协调节点

```
oma.startNode(18800)
```

- 步骤三：通过命令配置和启动编目节点

- 连接到临时协调节点，在 shell 命令中输入：

```
> var db = new Sdb("localhost", 18800)
```

其中18800为协调节点端口号

- 创建一个编目节点组

```
> db.createCataRG("sdbserver1", 11800, "/opt/sequoiadb/database/cata/11800")
```

其中

sdbserver1：第一个服务器主机名；

11800：为编目节点服务端口（该端口配置不要与随机端口冲突，以下其它端口的配置也需要注意）；

/opt/sequoiadb/database/cata/11800：为编目节点的数据文件存放路径；

 注：请确保存放路径的权限，如果 SequoiaDB 采用的默认安装，那么给路径赋予 sdbadmin 权限，下同。

3. 等待5秒，开始添加另外两个编目节点

```
> var cataRG = db.getRG("SYSCatalogGroup");
> var node1 = cataRG.createNode("sdbserver2", 11800, "/opt/sequoiadb/database/cata/11800")
> var node2 = cataRG.createNode("sdbserver3", 11800, "/opt/sequoiadb/database/cata/11800")
```

4. 启动编目节点组

```
> node1.start()
> node2.start()
```

 注：创建节点的第一个参数必须为“主机名”，而不能使主机的 IP。

- 步骤四：通过命令配置和启动数据节点

1. 创建数据节点组

```
> var dataRG = db.createRG("datagroup")
```

2. 添加数据节点

```
> dataRG.createNode("sdbserver1", 11820, "/opt/sequoiadb/database/data/11820")
> dataRG.createNode("sdbserver2", 11820, "/opt/sequoiadb/database/data/11820")
> dataRG.createNode("sdbserver3", 11820, "/opt/sequoiadb/database/data/11820")
```

 注：创建节点的第一个参数必须为“主机名”，而不能是主机的 IP。

3. 启动数据节点组

```
> dataRG.start()
```

- 步骤五：部署启动协调节点

1. 创建协调节点组

```
var rg = db.createCoordRG()
```

2. 创建协调节点

```
rg.createNode("sdbserver1", 11810, "/opt/sequoiadb/database/coord/11810")
rg.createNode("sdbserver2", 11810, "/opt/sequoiadb/database/coord/11810")
rg.createNode("sdbserver3", 11810, "/opt/sequoiadb/database/coord/11810")
```

3. 启动协调节点

```
rg.start()
```

- 步骤六：删除临时协调节点

1. 连接到本地的集群管理服务进程 sdbcm

```
var oma = new Oma("localhost", 11790)
```

2. 删除临时协调节点

```
oma.removeCoord(18800)
```

End

SequoiaDB Web 监控

启动 SequoiaDB Web 服务器

通过启动 SequoiaDB 后台的 Web 服务，也可以对数据库做各种操作，并且方便管理。

- 进入 /opt/sequoiadb/bin 目录
- 执行命令：

```
./sdbwsart -S <server name: port>
```

server name : 指定服务器IP地址，如 192.168.10.10

port : 指定服务器端口号，如 8080

最后在浏览器中输入 <http://192.168.10.10:8080>，即可访问数据库管理页。

监控管理

PHP 监控支持 IE9+、chrome、firefox 等主流浏览器。

- 1. 启动 PHP 服务

1.1 在安装了 SequoiaDB 的主机上，执行 sequoiadb 目录下 bin/sdbwsart。例如：

```
/opt/sequoiadb/bin/sdbwsart
```

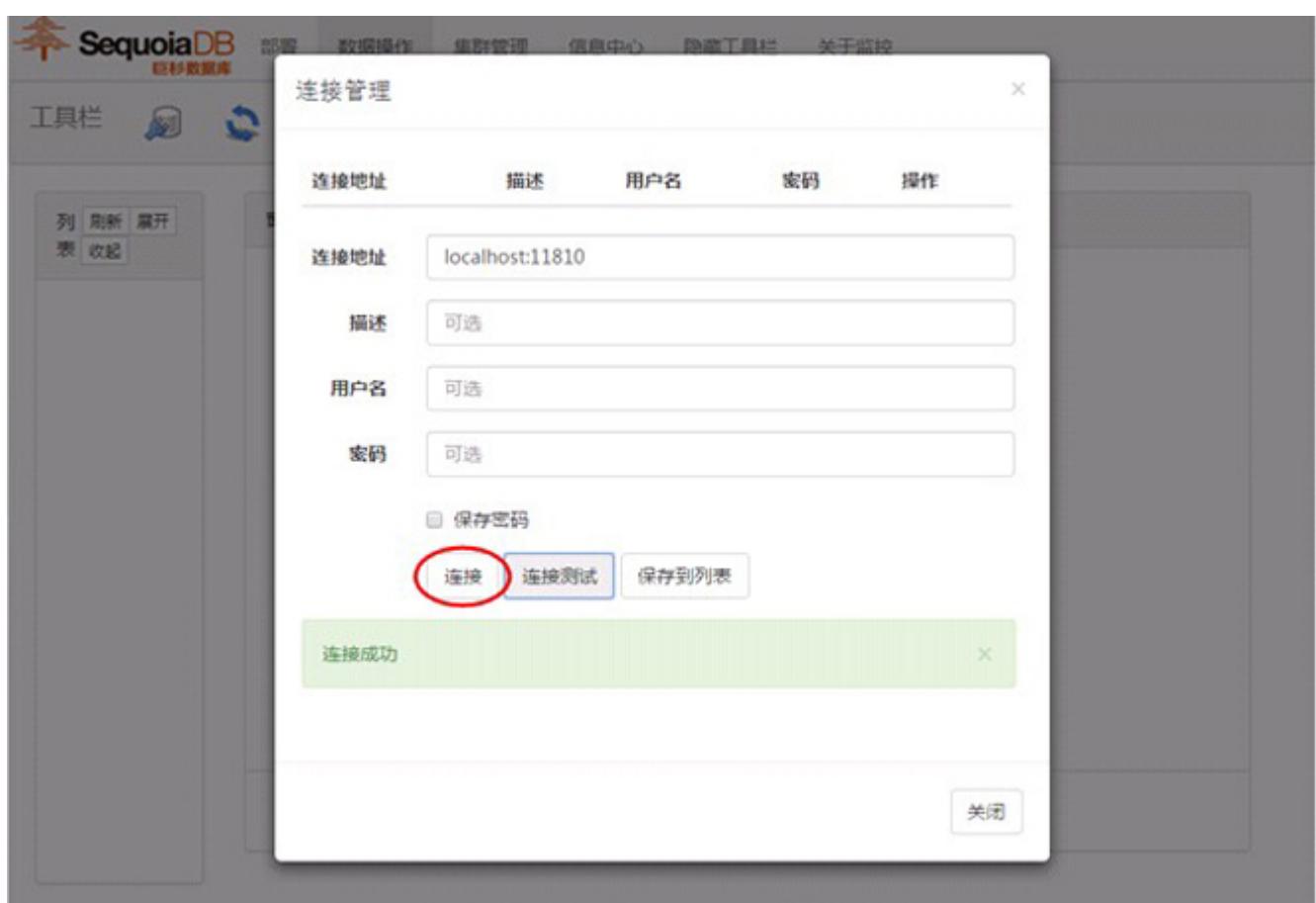
```
root@ubuntu-test-01:~# /opt/sequoiadb/bin/sdbwsart
start with /opt/sequoiadb/bin/sdbwsart -S 0.0.0.0:8080
nohup: appending output to `nohup.out'
start successful
```

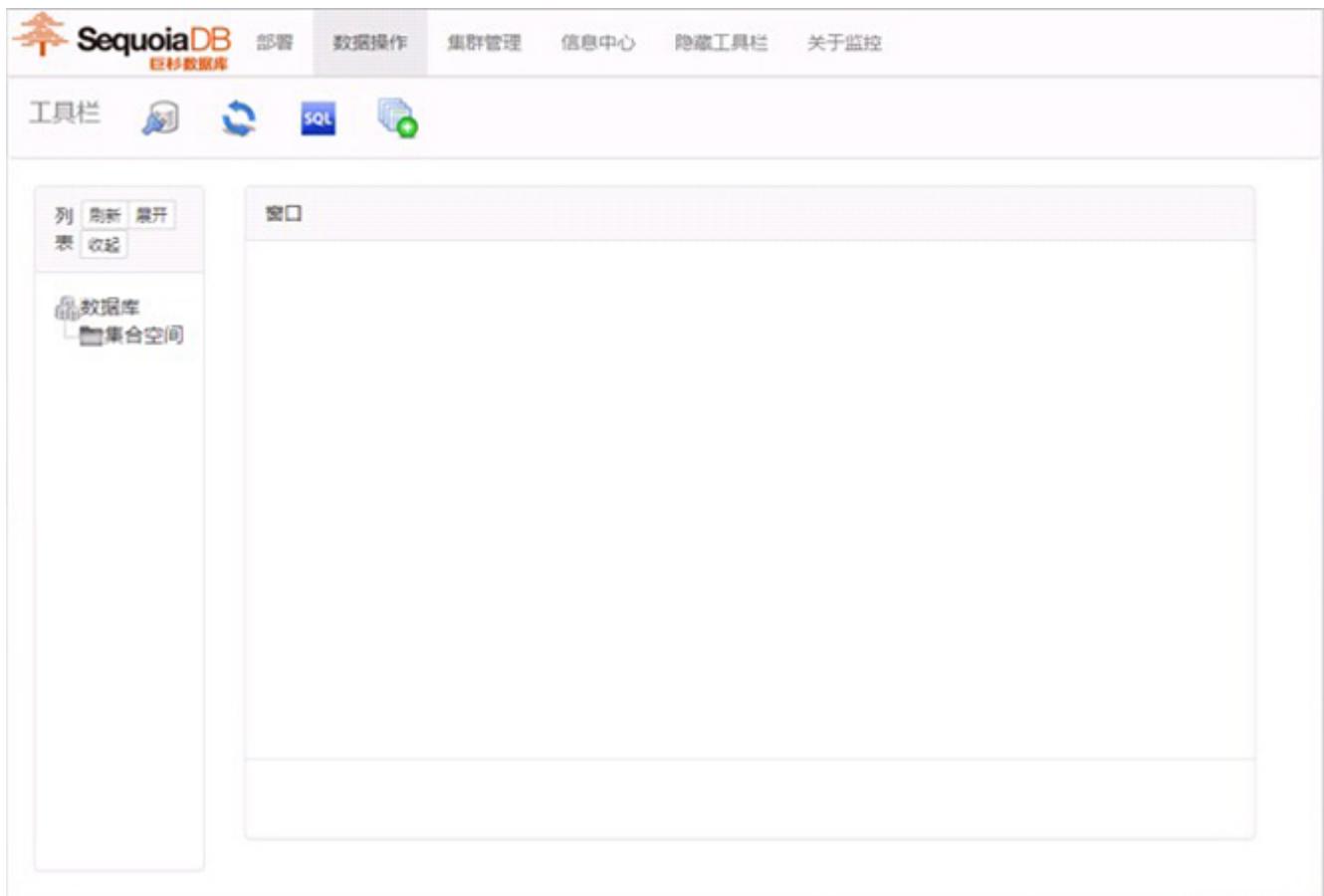
1.2 在浏览器输入 【步骤1.1】 的地址 <http://192.168.1.215:8080/>



- 2. 连接监控页面

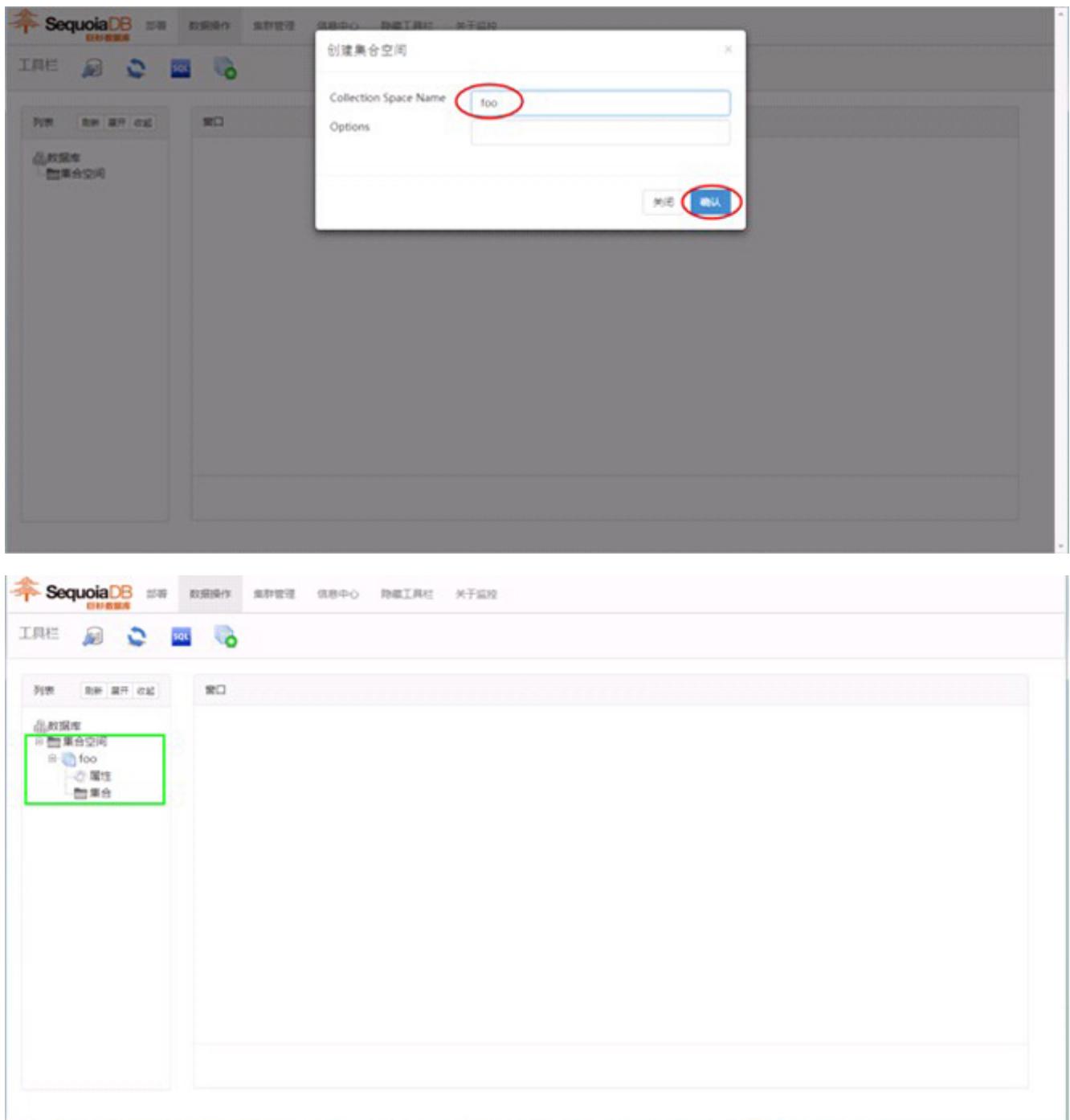
2.1 在连接管理 <连接地址> 输入地址，点击 <连接测试>，连接成功后，点击 <连接>；



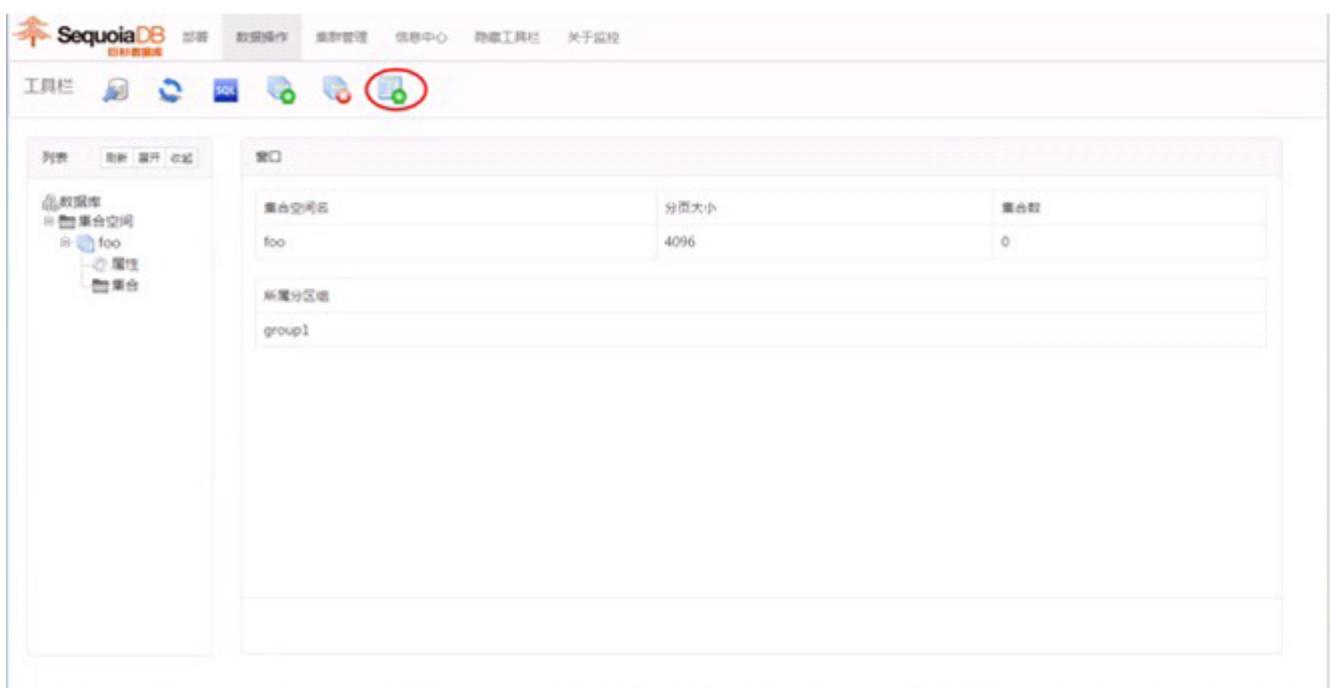
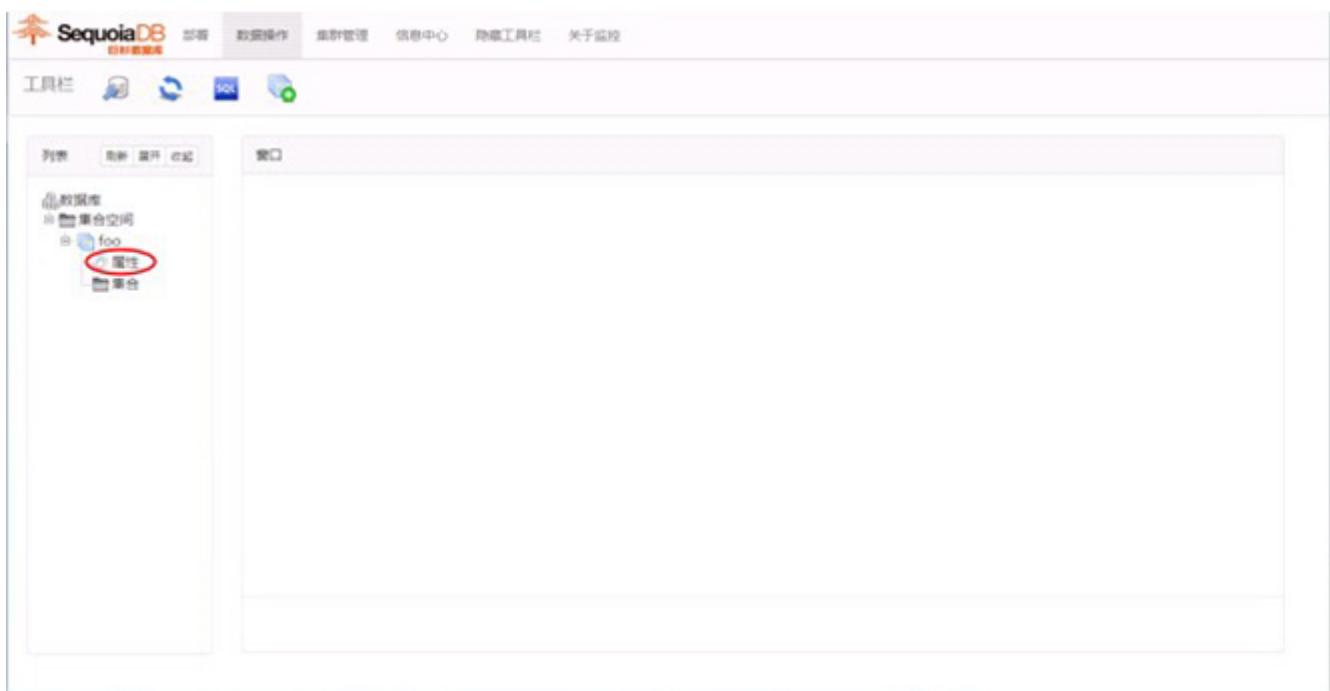


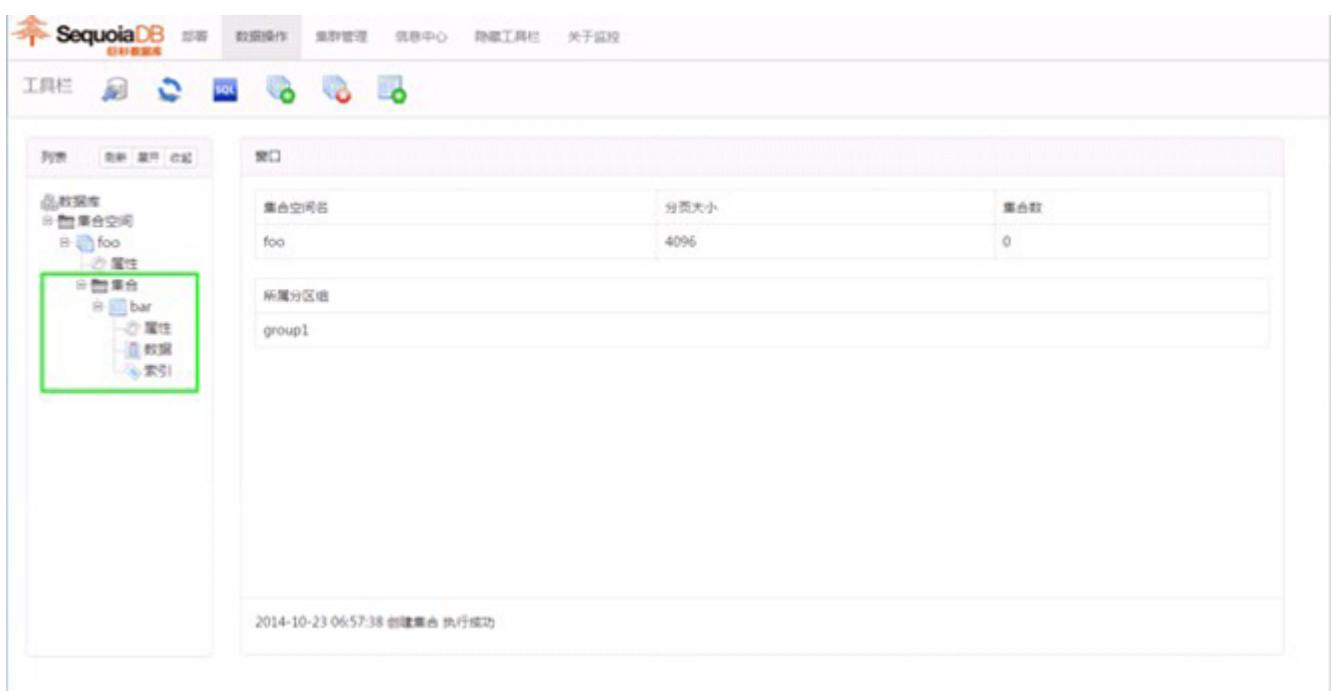
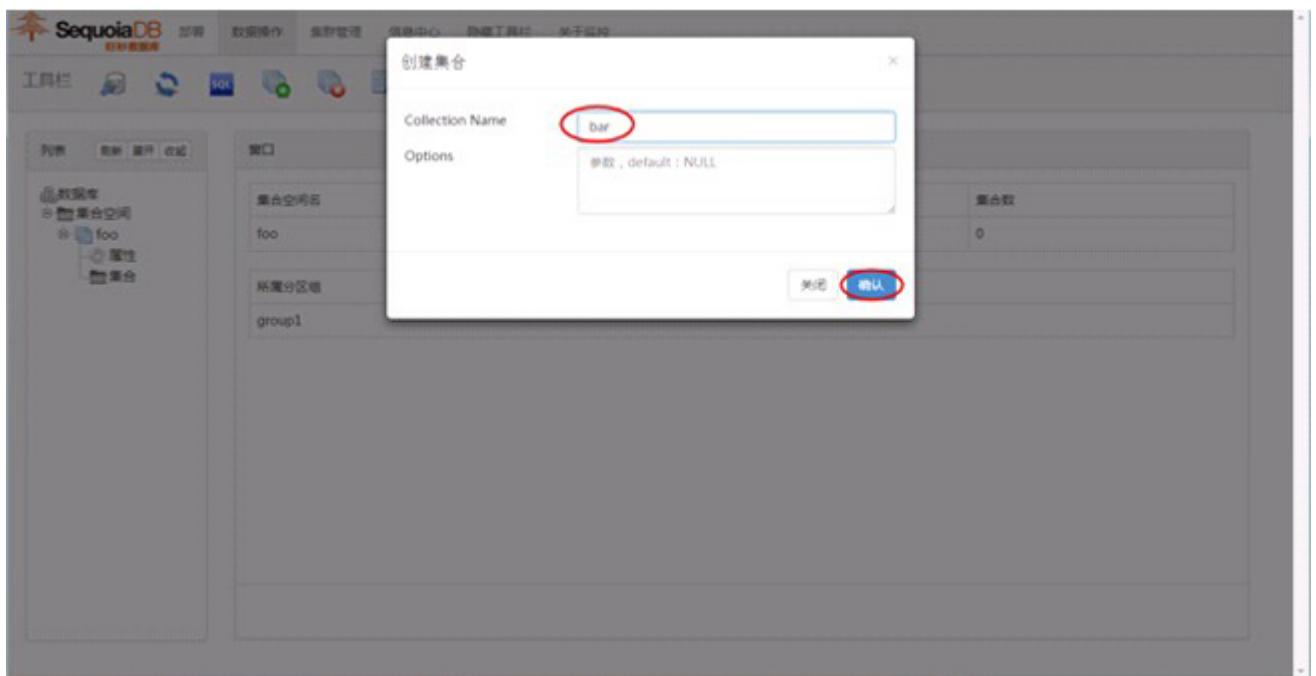
- 3. 创建集合空间、集合、插入数据





3.2 创建集合，点击集合空间 foo 的 <属性>，点击 <工具栏> 的  按钮，输入 <集合名>，点击 <确定>；





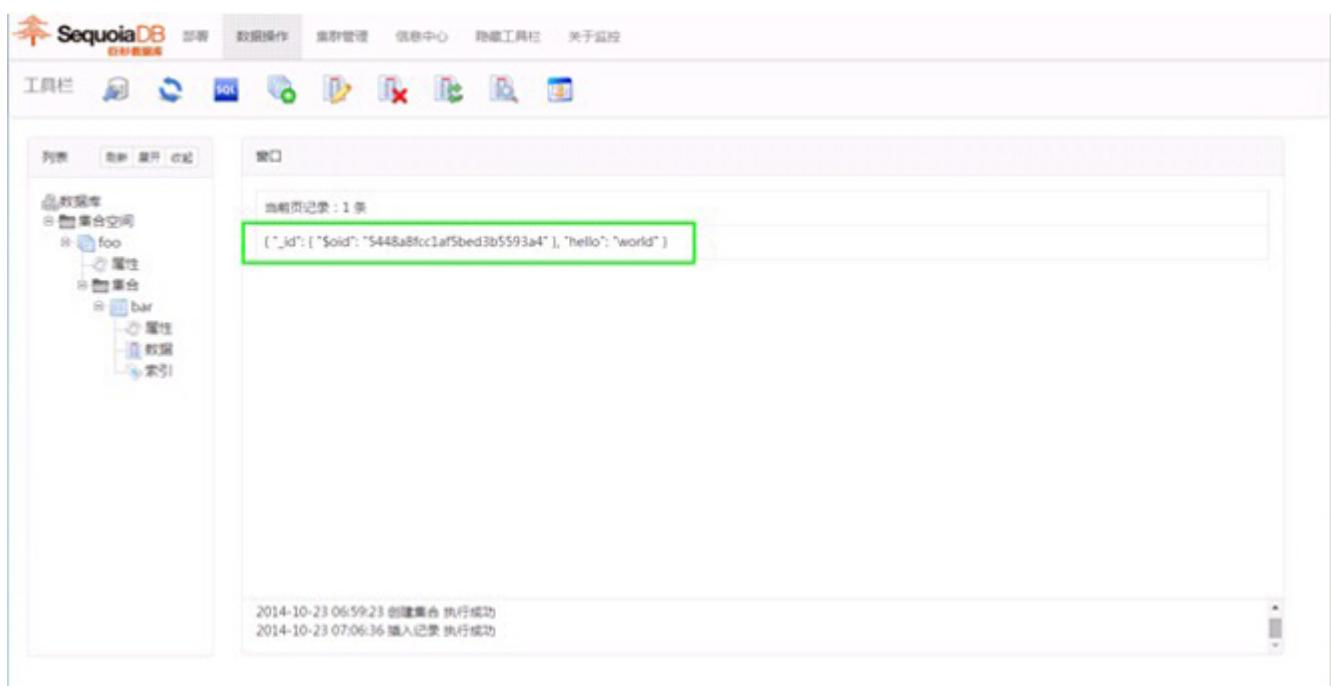
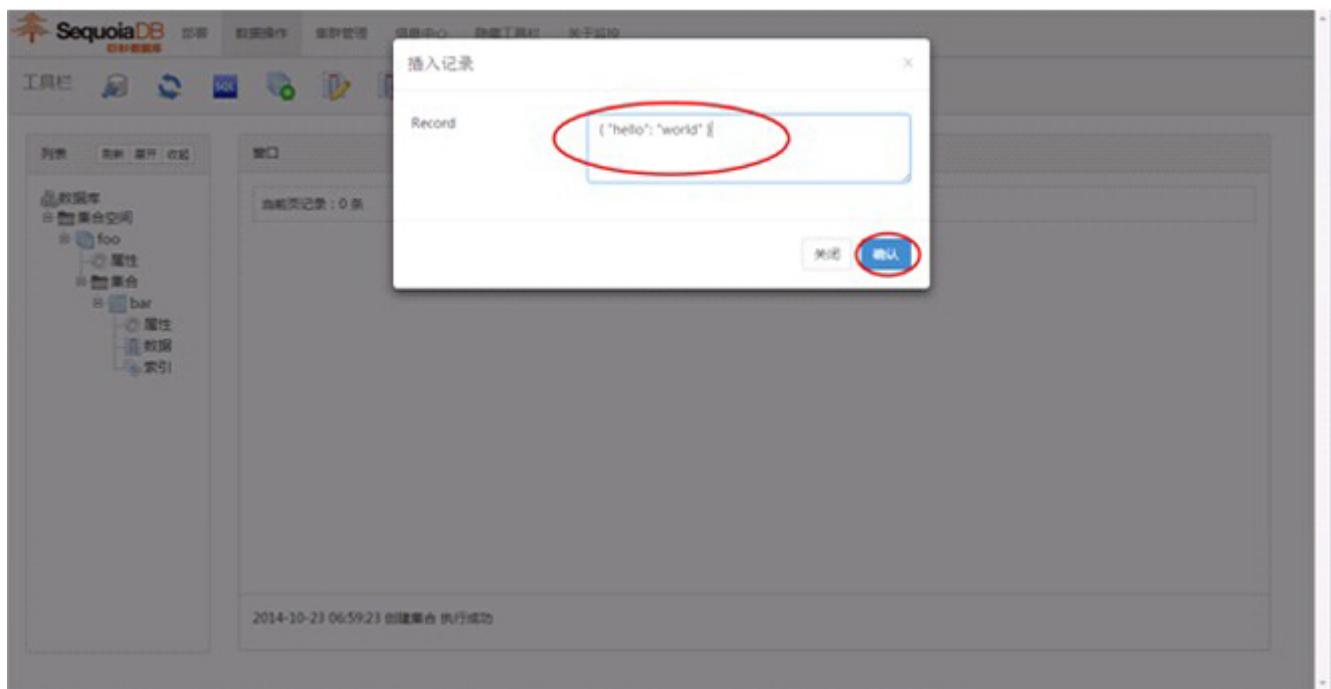
3.3 插入数据，点击集合 bar 的 <数据>，点击 <工具栏> 的 按钮，输入 json 记录，点击 <确定>；

The screenshot shows the SequoiaDB management interface. The left sidebar displays a database structure with a tree view. A red circle highlights the '数据' (Data) icon under the 'bar' collection. The main window shows a table with one row of data:

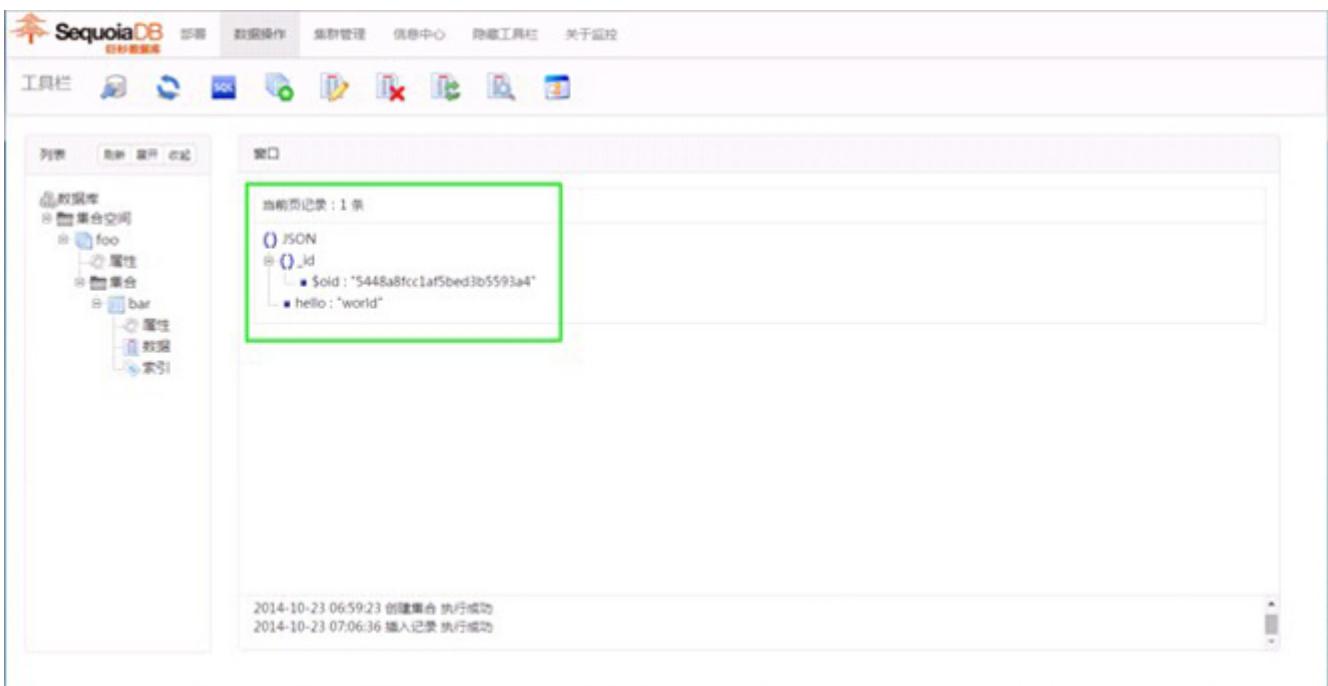
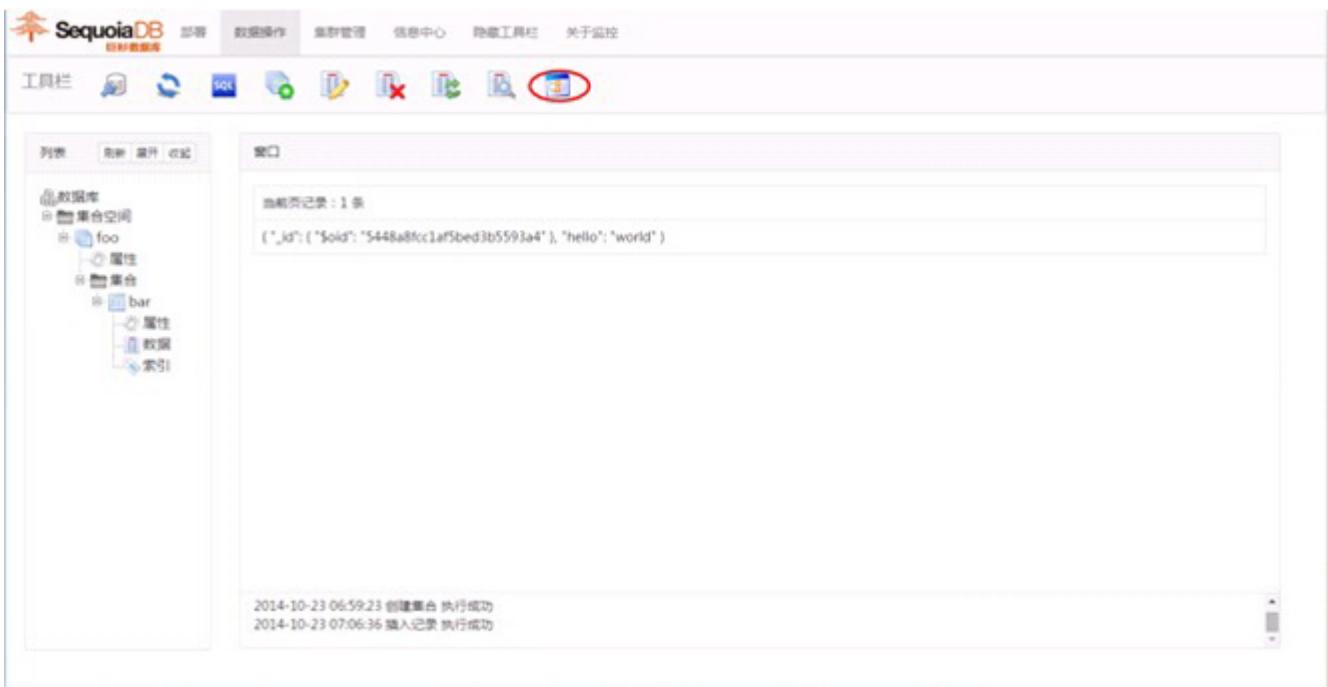
集合空间名	分页大小	集合数
foo	4096	1

Below the table, a message box displays the log entry: "2014-10-23 06:59:23 创建集合 执行成功". The browser address bar shows the URL: "192.168.1.215:8080/index.php?p=data#".

The screenshot shows the SequoiaDB management interface. The left sidebar displays a database structure with a tree view. A red circle highlights the '数据' (Data) icon under the 'bar' collection. The main window shows a message box with the text: "当前页记录：0 条". Below the message box, a log entry is visible: "2014-10-23 06:59:23 创建集合 执行成功". The browser address bar shows the URL: "192.168.1.215:8080/index.php?p=data#".



3.4 格式化 Json 记录，点击 <工具栏> 的 按钮。



- 4. 集群管理

4.1 查看集群，点击导航 <集群管理>。

The screenshot shows the SequoiaDB management interface. The top navigation bar includes tabs for 部署 (Deployment), 数据操作 (Data Operations), **集群管理** (Cluster Management), 信息中心 (Information Center), 跟踪工具栏 (Tracking Toolbar), and 关于监控 (About Monitoring). Below the navigation bar is a toolbar with various icons. On the left, there is a sidebar titled '数据库' (Database) with a tree view showing a database named 'foo' containing '属性' (Properties), '集合' (Collections), 'bar' (another collection), and '索引' (Indexes). The main right panel is titled '窗口' (Window) and displays a JSON document structure. The JSON object contains an '_id' field with the value '5448a8fcc1af5bed3b5593a4' and a 'hello' field with the value 'world'. At the bottom of the main panel, there are two log entries: '2014-10-23 06:59:23 创建集合 执行成功' and '2014-10-23 07:06:36 插入记录 执行成功'.

This screenshot shows the same SequoiaDB management interface as the previous one, but with a different tab selected. The '数据统计' (Data Statistics) tab is highlighted with a green border and circled with a red marker in the toolbar. The other tabs (部署, 数据操作, 集群管理, 信息中心, 跟踪工具栏, 关于监控) are shown in grey. The rest of the interface, including the sidebar and the main window, remains the same as in the first screenshot.



测试环境

1. 进入 SequoiaDB Shell 控制台

```
/opt/sequoiadb/bin/sdb
```

2. 创建一个新的 sdb 连接

```
db = new Sdb("localhost", 11810);
```

3. 创建集合空间

```
db.createCS("foo");
```

4. 创建集合

```
db.foo.createCL("bar");
```

5. 写入记录

```
db.foo.bar.insert({ "name": "sequoiadb" });
```

6. 查询结果

```
db.foo.bar.find();
{
  "_id": {
    "$oid": "53a82aa2c4b970091e000000"
  },
  "name": "sequoiadb"
}
Return 1 row(s).
```

查询结果正确

卸载



注：集群环境需要在每台数据执行如下操作：

- 以 root 身份登陆数据库服务器
- 执行如下命令停止 SequoiaDB 配置服务程序
service sdbcm stop
- 执行如下命令停止 SequoiaDB 数据库服务程序
/opt/sequoiadb/bin/sdbstop
- 执行如下命令卸载 SequoiaDB 软件
/opt/sequoiadb/uninstall
- 回退系统配置参数

1. 删除配置文件 /etc/security/limits.conf 中的如下配置参数：

```
# <#domain>      <type>    <item>      <value>
# *              soft      core       0
# *              soft      data       unlimited
# *              soft      fsize     unlimited
# *              soft      rss       unlimited
# *              soft      as        unlimited
```

2. 删除配置文件 /etc/sysctl.conf 中的如下配置参数：

```
vm.swappiness = 0
vm.dirty_ratio = 100
vm.dirty_background_ratio = 10
vm.dirty_expire_centisecs = 50000
vm.vfs_cache_pressure = 200
vm.min_free_kbytes = <物理内存大小的8%，单位KB>
```

升级

卸载旧版本

- 以 root 身份登陆数据库服务器
- 执行如下命令停止 SequoiaDB 配置服务程序
service sdbcm stop
- 执行如下命令停止 SequoiaDB 数据库服务程序
/opt/sequoiadb/bin/sdbstop
- 执行如下命令命令备份 SequoiaDB 执行程序以及配置文件
scp /opt/sequoiadb/bin/sequoiadb /home/sequoiadb_bak
scp -rf /opt/sequoiadb/conf /home/sequoiadb_conf_bak
- 执行如下命令卸载 SequoiaDB 软件
/opt/sequoiadb/uninstall



注：分别在所有数据库服务器上执行完上述操作后，再执行以下操作：

安装新版本

- 具体安装步骤参见《SequoiaDB 服务器安装》章节

- 待新版本安装成功后，执行以下命令删除备份文件：

```
rm -rf /home/sequoiadb_bak  
rm -rf /home/sequoiadb_conf_bak
```

数据库管理

数据库管理相关内容

数据库管理

此部分提供有关基本管理任务

数据库配置

参数说明

参数名	缩写	类型	说明
--help	-h	--	打印帮助
--dbpath	-d	str	1.指定数据文件存放路径。2.如果不指定，则默认为当前路径。
--indexpath	-i	str	1.指定索引文件存放路径。2.如果不指定，则默认与'dbpath'相同。
--confpath	-c	str	1.指定配置文件路径（不包含文件名），系统会在confpath下寻找sdb.conf。2.sdb.conf中填入需要的配置项，配制方法为：参数名 = 参数值。如 svcname=11810 ; diaglevel=3 3.如果不指定此参数，系统默认在当前路径寻找sdb.conf。4.sdb.conf可以不存在。
--logpath	-l	str	1.副本节点在进行数据同步时会生成同步日志。此参数用来指定同步日志的路径。2.如果不指定，则默认路径为：数据文件路径/replicalog
--diagpath	--	str	1.指定诊断日志存放目录。2.如果不指定，则默认为：数据文件路径/diaglog
--diagnum	--	num	1.指定诊断日志文件最大数量。2.如果不指定，则默认为：20，-1表示不限制。
--bkuppath	--	str	1.指定备份文件生成目录。2.如果不指定，则默认为：数据文件路径/bakfile
--maxpool	--	str	1.指定线程池内线程数量。2.如果不指定，则默认为0。
--svcname	-p	str	1.指定本地服务端口。2.如果不指定则默认为11810端口用于编目节点，11800用于协调节点，11820用于数据节点。
--replname	-r	str	1.指定数据同步平面端口。2.如果不指定则默认为svcname+1。
--shardname	-a	str	1.指定shard平面端口。2.如果不指定则默认为svcname+2。

参数名	缩写	类型	说明
--catalogname	-x	str	1.指定catalog平面端口。 2.如果不指定则默认为svcname+3。
--httpname	-s	str	1.指定http端口。 2.如果不指定则默认为svcname+4。
--diaglevel	-v	num	1.指定诊断日志打印级别。SequoiaDB中诊断日志从0-5分别代表：SEVERE, ERROR, EVENT, WARNING, INFO, DEBUG。 2.如果不指定，则默认为WARNING。
--role	-o	str	1.指定服务角色。SequoiaDB分别以data/coord/catalog/standalone代表：数据节点/协调节点/编目节点/单机。 2.如果不指定则默认为单机。
--catalogaddr	-t	str	1.指定编目节点的地址。配置形式为"hostname1:catalogname1,hostname2:catalogname2"。 2.需要至少指定一个编目节点的地址。
--logfilesz	-f	num	1.指定同步日志文件的大小。合法输入为64 (MB) - 2048 (MB)。 2.如果不指定，则默认为64 (MB)。
--logfilenum	-n	num	1.指定同步日志文件的数量。 2.如果不指定，则默认为20。
--transactionon	-e	boolean	1.指定是否打开事务。 2.如果不指定，则默认为false。
--numpreload	--	num	页面预加载代理数据，默认值为0，取值范围：[0,100]
--maxprefpool	--	num	数据预取代理池最大数量，默认值:200,取值范围:[0,1000]
--maxreplsync	--	num	日志同步最大并发数量，默认值:10,取值范围:[0,200], 0表示不启用日志并发同步
--logbuffsize	--	num	复制日志内存页面数,默认值:1024,取值范围:[512,1024000],但日志总内存大小不能超过日志总文件大小;每个页面大小为64KB
--tmppath	--	num	数据库临时文件目录，默认为'数据库路径'+'/tmp'
--sortbuf	--	num	排序缓存大小(MB),默认值256,最小值128
--hjbuf	--	num	哈希连接缓存大小(MB),默认值128,最小值64
--syncstrategy	--	str	副本组之间数据同步控制策略,取值:none,keepnormal,keepall,默认为keepnormal。
--preferedinstance	--	str	1.指定执行读请求时优先选择的实例 2.如果不指定，则默认为随机选择任意实例。 3.取值列表： M--可读写实例 S--只读实例 A--任意实例 1-7--第n个实例

参数名	缩写	类型	说明
--numpagecleaners	--	num	数据库启动时需要开启的脏页清除器数量 0意味着不启动任何脏页清除器，默认为1，取值范围：[0, 50]。
--pagecleaninterval	--	num	对每个集合空间的进行脏页清除的最长时间间隔 单位：毫秒，默认：10000，最小：1000

👉 注：

SequoiaDB支持命令行方式及配置文件方式。当两种方式并存时，命令行参数将会覆盖配置文件中的相同的配置项。

同步日志的总大小 (logfilesz * logfilenum) 决定了在同步过程中的容错能力。日志越大则进行全量恢复的可能性越小。

监控

概念

监控是一种监视当前系统状态的方式。在 SequoiaDB 中，用户可以使用快照 (SNAPSHOT) 与列表 (LIST) 命令进行系统监控。

👉 注：如果在集群环境下查询快照，连接协调节点就可以获取。

连接协调节点，默认是获取整个集群的快照信息，如：

```
snapshot (SDB_SNAP_SYSTEM)
```

要获取指定分区组的快照信息，使用条件查询，如：

```
snapshot (SDB_SNAP_SYSTEM, { GroupName: "group1" } )
```

要获取指定节点的快照信息，如：

```
snapshot (SDB_SNAP_SYSTEM, { HostName: "host1", svcname: "11820" } )
```

快照

快照是一种得到系统当前状态的命令，主要分为以下类型：

快照标示	快照类型	描述
SDB_SNAP_CONTEXTS	上下文	上下文快照列出当前数据库节点中所有的会话所对应的上下文
SDB_SNAP_CONTEXTS_CURRENT	当前会话上下文	当前上下文快照列出当前数据库节点中当前会话所对应的上下文
SDB_SNAP_SESSIONS	会话	会话快照列出当前数据库节点中所有的会话
SDB_SNAP_SESSIONS_CURRENT	当前会话	当前会话快照列出当前数据库节点中当前的会话
SDB_SNAP_COLLECTIONS	集合	集合快照列出当前数据库节点或集群中所有非临时集合
SDB_SNAP_COLLECTIONSPACES	集合空间	集合空间快照列出当前数据库节点或集群中所有集合空间（编目集合空间除外）
SDB_SNAP_DATABASE	数据库	数据库快照列出当前数据库节点的数据库监视信息
SDB_SNAP_SYSTEM	系统	系统快照列出当前数据库节点的系统监视信息
SDB_SNAP_CATALOG	编目信息	用于查看编目信息

列表

列表是一种轻量级的得到系统当前状态的命令，主要分为以下类型：

列表示	列表类型	描述
SDB_LIST_CONTEXTS	上下文	上下文列表列出当前数据库节点中所有的会话所对应的上下文
SDB_LIST_CONTEXTS_CURRENT	当前会话上下文	当前上下文列表列出当前数据库节点中当前会话所对应的上下文
SDB_LIST_SESSIONS	会话	会话列表列出当前数据库节点中所有的会话
SDB_LIST_SESSIONS_CURRENT	当前会话	当前会话列表列出当前数据库节点中当前的会话
SDB_LIST_COLLECTIONS	集合	集合列表列出当前数据库节点或集群中所有非临时集合
SDB_LIST_COLLECTIONSPACES	集合空间	集合空间列表列出当前数据库节点或集群中所有集合空间（编目集合空间除外）
SDB_LIST_STORAGEUNITS	存储单元	存储单元列表列出当前数据库节点的全部存储单元信息
SDB_LIST_GROUPS	分区组	分区组列表列出当前集群中的所有分区信息

快照

本节介绍各种快照信息：

[上下文快照](#)

[当前上下文快照](#)

[会话快照](#)

[当前会话快照](#)

[集合快照](#)

[集合空间快照](#)

[数据库快照](#)

[操作系统快照](#)

[编目信息快照](#)

[上下文快照](#)

描述

上下文快照列出当前数据库节点中所有的会话所对应的上下文。

每一个会话为一条记录，如果一个会话中包括一个或一个以上的上下文时，其 Contexts 数组字段对每个上下文产生一个对象。

 注：快照操作自身需产生一个上下文，因此结果集中至少会返回一个当前快照的上下文信息。

标示

SDB_SNAP_CONTEXTS

字段信息

字段名	类型	描述
SessionID	字符串	会话 ID (主机名 : 端口号 : ID)
Contexts.ContextID	长整型	上下文 ID

字段名	类型	描述
Contexts.Type	字符串	上下文类型，如：DUMP
Contexts.Description	字符串	上下文的描述信息，如：包含当前的查询条件
Contexts.DataRead	长整型	所读数据
Contexts.IndexRead	长整型	所读索引
Contexts.QueryTimeSpent	浮点数	查询总时间（秒）
Contexts.StartTimestamp	时间戳	创建时间

示例

```
> db.snapshot(SDB_SNAP_CONTEXTS)
{
  "SessionID": "vmsvr2-suse-x64: 11820: 28",
  "Contexts": [
    {
      "ContextID": 12,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimestamp": "2013-09-27-18.06.37.079570"
    }
  ]
}
```

当前上下文快照

描述

当前上下文快照列出数据库节点中，当前连接所对应的会话中的上下文。

返回一条记录，其中 Contexts 数组字段中包含当前会话中所有的上下文。

 注：快照操作自身需产生一个上下文，因此结果集中至少包含一个上下文。

标示

SDB_SNAP_CONTEXTS_CURRENT

字段信息

字段名	类型	描述
SessionID	字符串	会话 ID (Hostname:Port:ID)
Contexts.ContextID	长整型	上下文 ID
Contexts.Type	字符串	上下文类型，如：DUMP
Contexts.Description	字符串	上下文的描述信息，如：包含当前的查询条件
Contexts.DataRead	长整型	所读数据
Contexts.IndexRead	长整型	所读索引
Contexts.QueryTimeSpent	浮点数	查询总时间（秒）
Contexts.StartTimestamp	时间戳	创建时间

示例

```
> db.snapshot(SDB_SNAP_CONTEXTS_CURRENT)
```

```
{
  "SessionID": "vmsvr2-suse-x64: 11820: 28",
  "Contexts": [
    {
      "ContextID": 13,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimeStamp": "2013-09-27-18.25.17.311168"
    }
  ]
}
```

会话快照

描述

会话快照列出当前数据库节点中所有的用户与系统会话，每一个会话为一条记录。

标示

SDB_SNAP_SESSIONS

字段信息

字段名	类型	描述
SessionID	整型或长整型	会话 ID (主机名 : 端口号 : ID)
TID	整型	该会话所对应的系统线程 ID
Status	字符串	会话状态 <ul style="list-style-type: none"> Creating : 创建状态 Running : 运行状态 Waiting : 等待状态 Idle : 线程池待机状态 Destroying : 销毁状态
Type	字符串	EDU 类型
Name	字符串	EDU 名 , 一般系统 EDU 名为空
QueueSize	整型	等待处理请求的队列长度
ProcessEventCount	长整型	已经处理请求的数量
Contexts	长整型数组	上下文 ID 数组 , 为该会话所包含的所有上下文列表
TotalDataRead	长整型	数据记录读
TotalIndexRead	长整型	索引读
TotalDataWrite	长整型	数据记录写
TotalIndexWrite	长整型	索引写
TotalUpdate	长整型	总更新记录数量
TotalDelete	长整型	总删除记录数量
TotalInsert	长整型	总插入记录数量
TotalSelect	长整型	总选取记录数量
TotalRead	长整型	总数据读
TotalReadTime	长整型	总数据读时间 (毫秒)

字段名	类型	描述
TotalWriteTime	长整型	总数据写时间(毫秒)
ReadTimeSpent	长整型	读取记录的时间(毫秒)
WriteTimeSpent	长整型	写入记录的时间(毫秒)
ConnectTimestamp	时间戳	连接发起时间
LastOpType	字符串	最后一次操作的类型,如:insert, update
LastOpBegin	字符串	最后一次操作的起始时间
LastOpEnd	字符串	最后一次操作的结束时间
LastOpInfo	字符串	最后一次操作的详细信息
UserCPU	浮点数	用户CPU(秒)
SysCPU	浮点数	系统CPU(秒)

示例

```
> db.snapshot(SDB_SNAP_SESSIONS)
{
  "SessionID": "vmsvr2-suse-x64: 11820: 1",
  "TID": 8680,
  "Status": "Running",
  "Type": "LogWriter",
  "Name": "",
  "QueueSize": 0,
  "ProcessEventCount": 1,
  "Contexts": [],
  "TotalDataRead": 0,
  "TotalIndexRead": 0,
  "TotalDataWrite": 0,
  "TotalIndexWrite": 0,
  "TotalUpdate": 0,
  "TotalDelete": 0,
  "TotalInsert": 0,
  "TotalSelect": 0,
  "TotalRead": 0,
  "TotalReadTime": 0,
  "TotalWriteTime": 0,
  "ReadTimeSpent": 0,
  "WriteTimeSpent": 0,
  "ConnectTimestamp": "2013-09-27-13.28.38.927465",
  "LastOpType": "unknow",
  "LastOpBegin": "--",
  "LastOpEnd": "--",
  "LastOpInfo": "",
  "UserCPU": "0.410000",
  "SysCPU": "0.150000"
}
```

当前会话快照

描述

当前会话快照列出数据库节点中的当前用户会话，返回一条记录。

标示

SDB_SNAP_SESSIONS_CURRENT

字段信息

字段名	类型	描述
SessionID	字符串	会话 ID (主机名 : 端口号 : ID)
TID	整型	该会话所对应的系统线程 ID
Status	字符串	会话状态 <ul style="list-style-type: none"> • Creating : 创建状态 • Running : 运行状态 • Waiting : 等待状态 • Idle : 线程池待机状态 • Destroying : 销毁状态
Type	字符串	EDU 类型
Name	字符串	EDU 名 , 一般系统 EDU 名为空
QueueSize	整型	等待处理请求的队列长度
ProcessEventCount	长整型	已经处理请求的数量
Contexts	长整型数组	上下文 ID 数组 , 为该会话所包含的所有上下文列表
TotalDataRead	长整型	数据记录读
TotalIndexRead	长整型	索引读
TotalDataWrite	长整型	数据记录写
TotalIndexWrite	长整型	索引写
TotalUpdate	长整型	总更新记录数量
TotalDelete	长整型	总删除记录数量
TotalInsert	长整型	总插入记录数量
TotalSelect	长整型	总选取记录数量
TotalRead	长整型	总数据读
TotalReadTime	长整型	总数据读时间 (毫秒)
TotalWriteTime	长整型	总数据写时间 (毫秒)
ReadTimeSpent	长整型	读取记录的时间 (毫秒)
WriteTimeSpent	长整型	写入记录的时间 (毫秒)
ConnectTimestamp	时间戳	连接发起时间
LastOpType	字符串	最后一次操作的类型 , 如 : insert , update
LastOpBegin	字符串	最后一次操作的起始时间
LastOpEnd	字符串	最后一次操作的结束时间
LastOpInfo	字符串	最后一次操作的详细信息
UserCPU	浮点数	用户 CPU (秒)
SysCPU	浮点数	系统 CPU (秒)

示例

```
> db.snapshot(SDB_SNAP_SESSIONS_CURRENT)
{
  "SessionID": "vmsvr2-suse-x64:11820:28",
  "TID": 9430,
  "Status": "Running",
  "Type": "Agent",
  "Name": "127.0.0.1:60309",
  "QueueSize": 0,
```

```

"ProcessEventCount": 12,
"Contexts": [
    15
],
"TotalDataRead": 0,
"TotalIndexRead": 0,
"TotalDataWrite": 0,
"TotalIndexWrite": 0,
"TotalUpdate": 0,
"TotalDelete": 0,
"TotalInsert": 0,
"TotalSelect": 0,
"TotalRead": 0,
"TotalReadTime": 0,
"TotalWriteTime": 0,
"ReadTimeSpent": 10,
"WriteTimeSpent": 0,
"ConnectTimestamp": "2013-09-27-18.06.25.961090",
"LastOpType": "unknow",
"LastOpBegin": "2014-08-07-14.25.23.550216",
"LastOpEnd": "--",
"LastOpInfo": "",
"UserCPU": "0.910000",
"SysCPU": "2.060000"
}

```

集合快照

描述

集合快照列出当前数据库节点中所有的非临时集合（协调节点中列出用户集合），每个集合为一条记录。

标示

SDB_SNAP_COLLECTIONS

字段信息

由于数据节点与编目节点保存的集合信息不同，集合快照在协调节点与其它节点所返回的结构有所不同：

非协调节点字段信息

字段名	类型	描述
Name	字符串	集合完整名
Details.ID	整型	集合 ID，范围0~4095，集合空间内唯一
Details.LogicalID	整型	集合逻辑 ID
Details.Sequence	整型	序列号
Details.Indexes	整型	该集合所包含的索引数量
Details.Status	字符串	集合当前状态 <ul style="list-style-type: none"> • Free : 空闲 • Normal : 正常 • Dropped : 被删除 • Offline Reorg Shadow Copy Phase : 离线重组复制阶段 • Offline Reorg Truncate Phase : 离线重组清除阶段 • Offline Reorg Copy Back Phase : 离线重组重入阶段

字段名	类型	描述
		<ul style="list-style-type: none"> Offline Reorg Rebuild Phase : 离线重组重建索引阶段
TotalRecords	长整型	集合的记录总数
TotalDataPages	整型	集合的数据页总数
TotalIndexPages	整型	集合的索引页总数
TotalLobPages	整型	集合的大对象页总数
TotalDataFreeSpace	长整型	集合的数据空闲空间
TotalIndexFreeSpace	长整型	集合的索引空闲空间

协调节点字段信息

字段名	类型	描述
Name	字符串	集合完整名
Details.GroupName	字符串	节点所在分区组名
Details.Group.ID	整型	集合 ID , 范围0~4096 , 集合空间内唯一
Details.Group.LogicalID	整型	集合逻辑 ID
Details.Group.Sequence	整型	序列号
Details.Group.Indexes	整型	该集合所包含的索引数量
Details.Group.Status	字符串	<p>集合当前状态</p> <ul style="list-style-type: none"> Free : 空闲 Normal : 正常 Dropped : 被删除 Offline Reorg Shadow Copy Phase : 离线重组复制阶段 Offline Reorg Truncate Phase : 离线重组清除阶段 Offline Reorg Copy Back Phase : 离线重组重入阶段 Offline Reorg Rebuild Phase : 离线重组重建索引阶段
Details.Group.TotalRecords	长整型	集合的记录总数
Details.Group.TotalDataPages	整型	集合的数据页总数
Details.Group.TotalIndexPages	整型	集合的索引页总数
Details.Group.TotalDataFreeSpace	长整型	集合的数据空闲空间
Details.Group.TotalIndexFreeSpace	长整型	集合的索引空闲空间
Details.Group.NodeName	字符串	节点名 (主机名 + 端口)

非协调节点示例

```
> db.snapshot(SDB_SNAP_COLLECTIONS)
{
  "Name": "foo.bar",
  "Details": [
    {
      "ID": 0,
      "LogicalID": 0,
      "Sequence": 1,
      "Indexes": 8,
```

```

        "Status": "Normal",
        "TotalRecords": 0,
        "TotalDataPages": 0,
        "TotalIndexPages": 6,
        "TotalLobPages": 0,
        "TotalDataFreeSpace": 0,
        "TotalIndexFreeSpace": 196545
    }
]
}

```

协调节点示例

```

> coord.snapshot(SDB_SNAP_COLLECTIONS)
{
  "Name": "susefoo.susebar",
  "Details": [
    {
      "GroupName": "datagroup1",
      "Group": [
        {
          "ID": 0,
          "LogicalID": 0,
          "Sequence": 1,
          "Indexes": 1,
          "Status": "Normal",
          "TotalRecords": 1,
          "TotalDataPages": 1,
          "TotalIndexPages": 2,
          "TotalLobPages": 0,
          "TotalDataFreeSpace": 4004,
          "TotalIndexFreeSpace": 4046,
          "NodeName": "vmsvr2-suse-x64:11820"
        }
      ]
    }
  ]
}

```

集合空间快照

描述

集合空间快照列出当前数据库节点中所有的集合空间，每个集合空间为一条记录。

标示

SDB_SNAP_COLLECTIONSPACES

字段信息

由于数据节点与编目节点保存的集合空间信息不同，集合空间快照在协调节点与其它节点所返回的结构有所不同：

非协调节点字段信息

字段名	类型	描述
Name	字符串	集合空间名
Collection	字符串数组	集合空间中所包含的所有集合
PageSize	整型	集合空间数据页大小
LobPageSize	整型	集合空间大对象数据页大小

字段名	类型	描述
MaxCapacitySize	长整型	集合空间的最大容量上限
MaxDataCapSize	长整型	集合空间数据文件最大容量上限
MaxIndexCapSize	长整型	集合空间索引文件最大容量上限
MaxLobCapSize	长整型	集合空间大对象文件最大容量上限
NumCollections	整型	集合数量
TotalRecords	整型	集合空间的记录总数
TotalSize	长整型	集合空间的总大小
FreeSize	长整型	集合空间的空闲大小
TotalContentSize	长整型	集合空间数据文件总大小
FreeContentSize	长整型	集合空间数据文件空闲空间大小
TotalIndexSize	长整型	集合空间索引文件总大小
FreeIndexSize	长整型	集合空间索引文件空闲空间大小
TotalLobSize	长整型	集合空间大对象文件总大小
FreeLobSize	长整型	集合空间大对象文件空闲空间大小

协调节点字段信息

字段名	类型	描述
Name	字符串	集合空间名
Collection	字符串数组	集合空间中所包含的所有集合
PageSize	整型	集合空间数据页大小
LobPageSize	整型	集合空间大对象数据页大小
TotalSize	长整型	集合空间的总大小
FreeSize	长整型	集合空间的空闲大小
TotalContentSize	长整型	集合空间数据文件总大小
FreeContentSize	长整型	集合空间数据文件空闲空间大小
TotalIndexSize	长整型	集合空间索引文件总大小
FreeIndexSize	长整型	集合空间索引文件空闲空间大小
TotalLobSize	长整型	集合空间大对象文件总大小
FreeLobSize	长整型	集合空间大对象文件空闲空间大小
Group.GroupName	字符串	该集合空间所在的分区组名列表

非协调节点示例

```
> db.snapshot(SDB_SNAP_COLLECTIONSPACES)
{
  "Collection": [
    {
      "Name": "bar"
    }
  ],
  "PageSize": 65536,
  "LobPageSize": 262144,
  "MaxCapacitySize": 26388279066624,
  "MaxDataCapSize": 8796093022208,
  "MaxIndexCapSize": 8796093022208,
  "MaxLobCapSize": 8796093022208,
  "NumCollections": 4,
  "TotalRecords": 2,
  "TotalSize": 306315264,
  "FreeSize": 265551224,
```

```

    "TotalDataSize": 155254784,
    "FreeDataSize": 133627904,
    "TotalIndexSize": 151060480,
    "FreeIndexSize": 134152171,
    "TotalLobSize": 352714752,
    "FreeLobSize": 140771328,
    "Name": "foo"
}

```

协调节点示例

```

> coord.snapshot (SDB_SNAP_COLLECTIONSPACES)
{
    "Name": "foo",
    "PageSize": 4096,
    "LobPageSize": 262144,
    "TotalSize": 918945792,
    "FreeSize": 805183062,
    "TotalDataSize": 155254784,
    "FreeDataSize": 133627904,
    "TotalIndexSize": 151060480,
    "FreeIndexSize": 134152171,
    "TotalLobSize": 352714752,
    "FreeLobSize": 140771328,
    "Collection": [
        {
            "Name": "bar"
        }
    ],
    "Group": [
        "db2"
    ]
}

```

数据库快照

描述

数据库快照列出当前数据库节点中主要的状态与性能监控参数，输出一条记录。

标示

SDB_SNAP_DATABASE

非协调节点字段信息

字段名	类型	描述
HostName	字符串	数据库节点所在物理节点的主机名
ServiceName	字符串	svcname 所指定的服务名，与 HostName 共同作为一个逻辑节点的标示
NodeName	字符串	节点名，为<HostName>:<ServiceName>
GroupName	字符串	该逻辑节点所属的分区组名，standalone 模式下该字段为空字符串
IsPrimary	布尔	该节点是否为主节点，standalone 模式下该字段为 false
ServiceStatus	布尔	是否为可提供服务状态。一些特殊状态，例如 全量同步 会使该状态为 false
BeginLSN.Offset	长整型	起始 LSN 的偏移
BeginLSN.Version	整型	起始 LSN 的版本号

字段名	类型	描述
CurrentLSN.Offset	长整型	当前 LSN 的偏移
CurrentLSN.Version	整型	当前 LSN 的版本号
TransInfo.BeginLSN	长整型	事务起始 LSN 的偏移
NodeID	数组	[分区组 ID , 节点 ID] , 在 standalone 模式下 , 该字段为 [0 , 0]
Version.Major	整型	数据库主版本号
Version.Minor	整型	数据库子版本号
Version.Release	整型	数据库发行版本号
Version.Build	字符串	数据库编译时间
CurrentActiveSessions	整型	当前活动会话 , 该数量包括用户 EDU 与系统 EDU
CurrentIdleSessions	整型	当前非活动会话 , 一般来说非活动会话意味着 EDU 存在线程池中等待分配
CurrentSystemSessions	整型	当前系统会话 , 为当前活动用户 EDU 数量
CurrentContexts	整型	当前上下文数量
ReceivedEvents	整型	当前分区接收到的事件请求总数
Role	字符串	当前节点角色
Disk.DatabasePath	字符串	数据库所在路径
Disk.LoadPercent	整型	数据库路径磁盘占用率百分比
Disk.TotalSpace	长整型	数据库路径总空间 (字节)
Disk.FreeSpace	长整型	数据库路径空闲空间 (字节) 重要 : 该字段以及以上所有字段仅在数据节点和编目节点显示 , 协调节点不显示
TotalNumConnects	整型	数据库连接请求数量
TotalDataRead	长整型	总数据读请求
TotalIndexRead	长整型	总索引读请求
TotalDataWrite	长整型	总数据写请求
TotalIndexWrite	长整型	总索引写请求
TotalUpdate	长整型	总更新记录数量
TotalDelete	长整型	总删除记录数量
TotalInsert	长整型	总插入记录数量
ReplUpdate	长整型	复制更新记录数量
ReplDelete	长整型	复制删除记录数量
ReplInsert	长整型	复制插入记录数量
TotalSelect	长整型	总选择记录数量
TotalRead	长整型	总读取记录数量
TotalReadTime	长整型	总读取时间 (毫秒)
TotalWriteTime	长整型	总写入时间 (毫秒)
ActivateTimestamp	时间戳	数据库节点启动时间
UserCPU	浮点数	用户 CPU (秒)
SysCPU	浮点数	系统 CPU (秒)
freeLogSpace	长整型	空闲日志空间
vsize	长整型	虚拟内存使用量
rss	长整型	物理内存使用量
fault	长整型	每秒访问失败数 (仅支持 Linux) , 数据被交换出物理内存 , 放到 swap

字段名	类型	描述
TotalMapped	长整型	mmap 的总数据量
svcNetIn	长整型	本地服务端口收到的网络流量
svcNetOut	长整型	本地服务端口发送的网络流量

协调节点字段信息

字段名	类型	描述
TotalNumConnects	整型	数据库连接请求数量
TotalDataRead	长整型	总数据读请求
TotalIndexRead	长整型	总索引读请求
TotalDataWrite	长整型	总数据写请求
TotalIndexWrite	长整型	总索引写请求
TotalUpdate	长整型	总更新记录数量
TotalDelete	长整型	总删除记录数量
TotalInsert	长整型	总插入记录数量
ReplUpdate	长整型	复制更新记录数量
ReplDelete	长整型	复制删除记录数量
ReplInsert	长整型	复制插入记录数量
TotalSelect	长整型	总选择记录数量
TotalRead	长整型	总读取记录数量
TotalReadTime	长整型	总读取时间 (毫秒)
TotalWriteTime	长整型	总写入时间 (毫秒)
freeLogSpace	长整型	空闲日志空间
vsize	长整型	虚拟内存使用量
rss	长整型	物理内存使用量
fault	长整型	每秒访问失败数 (仅支持 Linux) , 数据被交换出物理内存 , 放到 swap
TotalMapped	长整型	mmap 的总数据量
svcNetIn	长整型	本地服务端口收到的网络流量
svcNetOut	长整型	本地服务端口发送的网络流量
shardNetIn	长整型	shard 平面端口收到的网络流量
shardNetOut	长整型	shard 平面端口发送的网络流量
replNetIn	长整型	数据同步平面端口收到的网络流量
replNetOut	长整型	数据同步平面端口发送的网络流量
ErrNodes.NodeName	字符串	返回异常节点名 (主机名 + 端口) 重要 : 此字段仅在协调节点上并且有异常节点时显示
ErrNodes.Flag	整型	错误码 重要 : 此字段仅在协调节点上并且有异常节点时显示

非协调节点示例

```
> db.snapshot(SDB_SNAP_DATABASE)
{
  "NodeName": "ubuntu-dev12: 11810",
  "HostName": "ubuntu-dev12",
  "ServiceName": "11810",
  "GroupName": "",
  "IsPrimary": false,
  "ServiceStatus": true,
```

```

"BeginLSN": {
  "Offset": 0,
  "Version": 1
},
"CurrentLSN": {
  "Offset": -1,
  "Version": 0
},
"TransInfo": {
  "BeginLSN": -1
},
"NodeID": [
  0,
  0
],
"Version": {
  "Major": 1,
  "Minor": 8,
  "Release": 13971,
  "Build": "2014-08-07-11.04.12 (Debug)"
},
"CurrentActiveSessions": 18,
"CurrentIdleSessions": 0,
"CurrentSystemSessions": 5,
"CurrentContexts": 1,
"ReceivedEvents": 0,
"Role": "standalone",
"Disk": {
  "DatabasePath": "/home/users/hejiawen/sequoiadb-new/sequoiadb/trunk/bin",
  "LoadPercent": 46,
  "TotalSpace": 84543193088,
  "FreeSpace": 45332840448
},
"TotalNumConnects": 11,
"TotalDataRead": 0,
"TotalIndexRead": 0,
"TotalDataWrite": 0,
"TotalIndexWrite": 0,
"TotalUpdate": 0,
"TotalDelete": 0,
"TotalInsert": 0,
"Rep1Update": 0,
"Rep1Delete": 0,
"Rep1Insert": 0,
"TotalSelect": 0,
"TotalRead": 0,
"TotalReadTime": 0,
"TotalWriteTime": 0,
"ActivateTimestamp": "2014-08-07-13.04.16.248083",
"UserCPU": "7.980000",
"SysCPU": "10.700000",
"freeLogSpace": 1342177280,
"vsize": 1745002496,
"rss": 12929,
"fault": 12,
"TotalMapped": 918945792,
"svcNetIn": 3051,
"svcNetOut": 9245
}

```

协调节点示例

```

> db.snapshot(SDB_SNAP_DATABASE)
{

```

```

    "TotalNumConnects": 0,
    "TotalDataRead": 4,
    "TotalIndexRead": 0,
    "TotalDataWrite": 3,
    "TotalIndexWrite": 3,
    "TotalUpdate": 0,
    "TotalDelete": 0,
    "TotalInsert": 3,
    "Rep1Update": 0,
    "Rep1Delete": 0,
    "Rep1Insert": 2,
    "TotalSelect": 606,
    "TotalRead": 4,
    "TotalReadTime": 0,
    "TotalWriteTime": 0,
    "freeLogSpace": 5368709120,
    "vsize": 5660057600,
    "rss": 44765,
    "fault": 25,
    "TotalMapped": 2144206848,
    "svcNetIn": 0,
    "svcNetOut": 0,
    "shardNetIn": 38228,
    "shardNetOut": 393997,
    "rep1NetIn": 40743956,
    "rep1NetOut": 40743956,
    "ErrNodes": []
}

```

操作系统快照

描述

操作系统快照列出当前数据库节点所在操作系统中主要的状态与性能监控参数，输出一条记录。

标示

SDB_SNAP_SYSTEM

非协调节点字段信息

字段名	类型	描述
HostName	字符串	数据库节点所在物理节点的主机名
ServiceName	字符串	svcname 所指定的服务名，与 HostName 共同作为一个逻辑节点的标示
NodeName	字符串	节点名，为<HostName>:<ServiceName>
GroupName	字符串	该逻辑节点所属的分区组名，standalone 模式下，该字段为空字符串
IsPrimary	布尔	该节点是否为主节点，standalone 模式下，该字段为 false
ServiceStatus	布尔	是否为可提供服务状态。一些特殊状态，例如 全量同步 会使该状态为 false
BeginLSN.Offset	长整型	起始 LSN 的偏移
BeginLSN.Version	整型	起始 LSN 的版本号
CurrentLSN.Offset	整型	当前 LSN 的偏移
TransInfo.BeginLSN	长整型	事务起始 LSN 的偏移
NodeID	数组	[分区组ID，节点ID]，standalone 模式下，该字段为 [0, 0]

字段名	类型	描述
CurrentLSN.Version	整型	当前 LSN 的版本号
CPU.User	浮点数	操作系统启动后所消耗的总用户 CPU (秒)
CPU.Sys	浮点数	操作系统启动后所消耗的总系统 CPU (秒)
CPU.Idle	浮点数	操作系统启动后所消耗的总空闲 CPU (秒)
CPU.Other	浮点数	操作系统启动后所消耗的总其它 CPU (秒)
Memory.LoadPercent	整型	当前操作系统的内存使用百分比 (包括文件系统缓存) 重要 : 该字段仅在数据节点和编目节点显示 , 协调节点不显示
Memory.TotalRAM	长整型	当前操作系统的总内存空间 (字节)
Memory.FreeRAM	长整型	当前操作系统的空闲内存空间 (字节)
Memory.TotalSwap	长整型	当前操作系统的总交换空间 (字节)
Memory.FreeSwap	长整型	当前操作系统的空闲交换空间 (字节)
Memory.TotalVirtual	长整型	当前操作系统的总虚拟空间 (字节)
Memory.FreeVirtual	长整型	当前操作系统的空闲虚拟空间 (字节)
Disk.DatabasePath	字符串	数据库路径 重要 : 该字段及以上字段仅在数据节点和编目节点显示 , 协调节点不显示
Disk.LoadPercent	整型	数据库路径所在文件系统的空间占用百分比 重要 : 该字段及以上字段仅在数据节点和编目节点显示 , 协调节点不显示
Disk.TotalSpace	长整型	数据库路径总空间 (字节)
Disk.FreeSpace	长整型	数据库路径空闲空间 (字节)

协调节点字段信息

字段名	类型	描述
CPU.User	浮点数	操作系统启动后所消耗的总用户 CPU (秒)
CPU.Sys	浮点数	操作系统启动后所消耗的总系统 CPU (秒)
CPU.Idle	浮点数	操作系统启动后所消耗的总空闲 CPU (秒)
CPU.Other	浮点数	操作系统启动后所消耗的总其它 CPU (秒)
Memory.LoadPercent	整型	当前操作系统的内存使用百分比 (包括文件系统缓存) 重要 : 该字段仅在数据节点和编目节点显示 , 协调节点不显示
Memory.TotalRAM	长整型	当前操作系统的总内存空间 (字节)
Memory.FreeRAM	长整型	当前操作系统的空闲内存空间 (字节)
Memory.TotalSwap	长整型	当前操作系统的总交换空间 (字节)
Memory.FreeSwap	长整型	当前操作系统的空闲交换空间 (字节)
Memory.TotalVirtual	长整型	当前操作系统的总虚拟空间 (字节)
Memory.FreeVirtual	长整型	当前操作系统的空闲虚拟空间 (字节)
Disk.DatabasePath	字符串	数据库路径 重要 : 该字段及以上字段仅在数据节点和编目节点显示 , 协调节点不显示
Disk.LoadPercent	整型	数据库路径所在文件系统的空间占用百分比 重要 : 该字段及以上字段仅在数据节点和编目节点显示 , 协调节点不显示
Disk.TotalSpace	长整型	数据库路径总空间 (字节)
Disk.FreeSpace	长整型	数据库路径空闲空间 (字节)
ErrNodes.NodeName	字符串	返回异常节点名 (主机名 + 端口) 重要 : 此字段仅在协调节点上显示 , 并且有异常节点时才显示

字段名	类型	描述
ErrNodes.Flag	整型	错误码 重要：此字段仅在协调节点上显示，并且有异常节点时才显示

非协调节点示例

```
> db.snapshot(SDB_SNAP_SYSTEM)
{
  "NodeName": "vmsvr2-suse-x64: 11820",
  "HostName": "vmsvr2-suse-x64",
  "ServiceName": "11820",
  "GroupName": "datagroup1",
  "IsPrimary": false,
  "ServiceStatus": true,
  "BeginLSN": {
    "Offset": 0,
    "Version": 1
  },
  "CurrentLSN": {
    "Offset": 3764,
    "Version": 1
  },
  "NodeID": [
    1000,
    1000
  ],
  "TransInfo": {
    "BeginLSN": -1
  },
  "NodeID": [
    0,
    0
  ],
  "CPU": {
    "User": 3947.31,
    "Sys": 715.11,
    "Idle": 331196.41,
    "Other": 771.14
  },
  "Memory": {
    "LoadPercent": 95,
    "TotalRAM": 4155072512,
    "FreeRAM": 202219520,
    "TotalSwap": 2153771008,
    "FreeSwap": 2137071616,
    "TotalVirtual": 6308843520,
    "FreeVirtual": 2339291136
  },
  "Disk": {
    "DatabasePath": "/opt/sequoiadb/database/data/11820",
    "LoadPercent": 78,
    "TotalSpace": 40704466944,
    "FreeSpace": 8615747584
  }
}
```

协调节点示例

```
> db.snapshot(SDB_SNAP_SYSTEM)
{
  "CPU": {
    "User": 36280.72,
    "Sys": 5046.23,
```

```

    "Idle": 7560242.4,
    "Other": 5887.24
},
"Memory": {
    "TotalRAM": 8403730432,
    "FreeRAM": 3075035136,
    "TotalSwap": 25757204480,
    "FreeSwap": 25663799296,
    "TotalVirtual": 34160934912,
    "FreeVirtual": 28738834432
},
"Disk": {
    "TotalSpace": 338172772352,
    "FreeSpace": 181331296256
},
"ErrNodes": []
}

```

编目信息快照

描述

编目信息快照列出当前数据库中所有集合的编目信息，每个集合一条记录。

标示

SDB_SNAP_CATALOG

 注：只能在协调节点执行。

协调节点字段信息

字段名	类型	描述
Name	字符串	集合完整名
EnsureShardingIndex	布尔类型	是否自动为分区键字段创建索引
ReplSize	整型	执行修改操作时需要同步的副本数。当执行更新，插入，删除记录等操作时，仅当指定副本数的节点都完成操作时才返回操作结果。
ShardingKey	对象	分区键定义
ShardingType	字符串	数据分区类型： range：数据按分区键值的范围进行分区存储 hash：数据按分区键的哈希值进行分区存储
Version	整型	集合版本号，当对集合的元数据执行修改操作时递增该版本号（例如数据切分）
CataInfo.GroupID	整型	分区组 ID
CataInfo.GroupName	字符串	分区组名
CataInfo.LowBound	对象	数据分区区间的上限
CataInfo.UpBound	对象	数据分区区间的下限

示例

```

> db.snapshot(SDB_SNAP_CATALOG)
{
    "_id": {
        "$oid": "5247a2bc60080822db1cfba2"
    },
    "Name": "foo.bar",
}

```

```

"Version": 1,
"Rep1Size": 1,
"ShardingKey": {
    "age": 1
},
"EnsureShardingIndex": true,
"ShardingType": "range",
"CataInfo": [
    {
        "GroupID": 1000,
        "GroupName": "datagroup1",
        "LowBound": {
            "": {
                "$minKey": 1
            }
        },
        "UpBound": {
            "": {
                "$maxKey": 1
            }
        }
    }
]
}

```

列表

本节介绍各种列表信息：

[上下文列表](#)

[当前会话上下文列表](#)

[会话列表](#)

[当前会话列表](#)

[集合列表](#)

[集合空间列表](#)

[存储单元列表](#)

[分区组列表](#)

[上下文列表](#)

描述

上下文列表列出当前数据库节点中所有的会话所对应的上下文。

每一个会话为一条记录，如果一个会话中包括一个或一个以上的上下文时，其 Contexts 数组字段对每个上下文产生一个对象。

注：列表操作自身需产生一个上下文，因此结果集中至少会返回一个当前列表的上下文信息。



标示

SDB_LIST_CONTEXTS

字段信息

字段名	类型	描述
SessionID	长整型	会话 ID

字段名	类型	描述
Contexts	长整型数组	上下文 ID 数组，为该会话所包含的所有上下文列表

示例

```
> db.list(SDB_LIST_CONTEXTS)
{
  "SessionID": 21,
  "Contexts": [
    182
  ]
}
```

当前上下文列表

描述

当前上下文列表列出数据库节点中，当前连接所对应的会话中的上下文。

返回一条记录，其中 Contexts 数组字段中包含当前会话中所有的上下文。

注：列表操作自身需产生一个上下文，因此结果集中至少包含一个上下文。

标示

SDB_LIST_CONTEXTS_CURRENT

字段信息

字段名	类型	描述
SessionID	长整型	会话 ID
Contexts	长整型数组	上下文 ID 数组，为该会话所包含的所有上下文列表

示例

```
> db.list(SDB_LIST_CONTEXTS_CURRENT)
{
  "SessionID": 21,
  "Contexts": [
    183
  ]
}
```

会话列表

描述

会话列表列出当前数据库节点中所有的用户与系统会话，每一个会话为一条记录。

标示

SDB_LIST_SESSIONS

字段信息

字段名	类型	描述
SessionID	整型或长整型	会话 ID

字段名	类型	描述
TID	整型	该会话所对应的系统线程 ID
Status	字符串	会话状态 <ul style="list-style-type: none"> Creating : 创建状态 Running : 运行状态 Waiting : 等待状态 Idle : 线程池待机状态 Destroying : 销毁状态
Type	字符串	EDU 类型
Name	字符串	EDU 名，一般系统 EDU 名为空

示例

```
> db.list(SDB_LIST_SESSIONS)
{
  "SessionID": 1,
  "TID": 6168,
  "Status": "Running",
  "Type": "TCPListener",
  "Name": ""
}
{
  "SessionID": 2,
  "TID": 6169,
  "Status": "Running",
  "Type": "HTTPListener",
  "Name": ""
}
...
{
  "SessionID": 21,
  "TID": 6691,
  "Status": "Running",
  "Type": "Agent",
  "Name": "192.168.20.101:52741"
}
```

当前会话列表

描述

当前会话列表列出数据库节点中的当前用户会话，返回一条记录。

标示

SDB_LIST_SESSIONS_CURRENT

字段信息

字段名	类型	描述
SessionID	整型或长整型	会话 ID
TID	整型	该会话所对应的系统线程 ID
Status	字符串	会话状态 <ul style="list-style-type: none"> Creating : 创建状态 Running : 运行状态 Waiting : 等待状态

字段名	类型	描述
		<ul style="list-style-type: none"> Idle : 线程池待机状态 Destroying : 销毁状态
Type	字符串	EDU 类型
Name	字符串	EDU 名 , 一般系统 EDU 名为空

示例

```
> db.list(SDB_LIST_SESSIONS_CURRENT)
{
  "SessionID": 21,
  "TID": 6691,
  "Status": "Running",
  "Type": "Agent",
  "Name": "192.168.20.101:52741"
}
```

集合列表

描述

集合快照列出当前数据库节点中所有的非临时集合（协调节点中列出用户集合），每个集合为一条记录。

标示

SDB_LIST_COLLECTIONS

字段信息

由于数据节点与编目节点保存的集合信息不同，集合列表在协调节点与其它节点所返回的结构有所不同：

非协调节点字段信息

字段名	类型	描述
Name	字符串	集合完整名
Details.ID	整型	集合 ID , 范围0~4095 , 集合空间内唯一
Details.LogicalID	整型	集合逻辑 ID
Details.Sequence	整型	序列号
Details.Indexes	整型	该集合所包含的索引数量
Details.Status	字符串	<p>集合当前状态</p> <ul style="list-style-type: none"> Free : 空闲 Normal : 正常 Dropped : 被删除 Offline Reorg Shadow Copy Phase : 离线重组复制阶段 Offline Reorg Truncate Phase : 离线重组清除阶段 Offline Reorg Copy Back Phase : 离线重组重入阶段 Offline Reorg Rebuild Phase : 离线重组重建索引阶段

协调节点字段信息

字段名	类型	描述
Name	字符串	集合完整名

非协调节点示例

```
> db.list(SDB_LIST_COLLECTIONS)
{
  "Name": "foo.test",
  "Details": [
    {
      "ID": 0,
      "Logical ID": 0,
      "Sequence": 1,
      "Indexes": 2,
      "Status": "Normal"
    }
  ]
}
```

协调节点示例

```
> db.list(SDB_LIST_COLLECTIONS)
{
  "Name": "foo.bar"
}
```

集合空间列表

描述

集合空间列表列出当前数据库节点中所有的集合空间，每个集合空间为一条记录。

标示

SDB_LIST_COLLECTIONSPACES

字段信息

由于数据节点与编目节点保存的集合空间信息不同，集合空间列表在协调节点与其它节点所返回的结构有所不同：

非协调节点字段信息

字段名	类型	描述
Name	字符串	集合空间名
Collection	字符串数组	集合空间中所包含的所有集合
PageSize	整型	集合空间数据页大小

协调节点字段信息

字段名	类型	描述
Name	字符串	集合空间名

非协调节点示例

```
> db.list(SDB_LIST_COLLECTIONSPACES)
{
  "Collection": [
```

```
{
  "Name": "test"
}
],
"Name": "foo",
"PageSize": 4096
}
```

协调节点示例

```
> db.list(SDB_LIST_COLLECTIONSPACES)
{
  "Name": "foo"
}
```

存储单元列表

描述

存储单元列表列出当前数据库节点的全部存储单元信息。

标示

SDB_LIST_STORAGEUNITS

字段信息

字段名	类型	描述
Name	字符串	集合空间名
ID	整型	该集合空间 ID
LogicalID	字符串	集合空间逻辑 ID，为递增顺序
PageSize	整型	集合空间数据页大小
Sequence	整型	序列号，当前版本中为1
NumCollections	整型	集合空间下的集合个数
CollectionHWM	整型	集合高水位，一般来说意味着该集合空间中总共创建过的集合数量（包括被删除的集合）
Size	长整型	存储单元大小（字节）

示例

```
> db.list(SDB_LIST_STORAGEUNITS)
{
  "Name": "testCS",
  "ID": 4095,
  "LogicalID": 0,
  "PageSize": 4096,
  "Sequence": 1,
  "NumCollections": 1,
  "CollectionHWM": 1,
  "Size": 172032000
}
{
  "Name": "foo",
  "ID": 4094,
  "LogicalID": 1,
  "PageSize": 4096,
  "Sequence": 1,
  "NumCollections": 2,
  "CollectionHWM": 3,
  "Size": 172032000
}
```

}

分区组列表

描述

分区组列表列出当前集群中的所有分区信息。

标示

SDB_LIST_GROUPS

字段信息

字段名	类型	描述
Group.dbpath	字符串	分区组中节点的数据文件存放路径
Group.HostName	字符串	分区组中节点的主机名
Group.Service.Type	整型	分区组中节点的服务类型 <ul style="list-style-type: none"> • 0 : 直连服务 , 对应数据库参数 svcname • 1 : 复制服务 , 对应数据库参数 replname • 2 : 分区服务 , 对应数据库参数 shardname • 3 : 编目服务 , 对应数据库参数 catalogname
Group.Service.Name	字符串	分区组中节点的服务名 , 服务名可以为端口号 , 或 services 文件中的服务名
Group.NodeID	整型	分区组中节点的 ID
GroupID	整型	分区组 ID
GroupName	字符串	分区组名称
PrimaryNode	整型	主节点 ID
Role	整型	分区组角色 , 可以为 : <ul style="list-style-type: none"> • 0 : 数据节点 • 2 : 编目节点
Status	字符串	分区组状态 <ul style="list-style-type: none"> • 1 : 已激活分区组 • 0 : 未激活分区组 • 不存在 : 未激活分区组
Version	整型	

示例

```
> db.list(SDB_LIST_GROUPS)
{
  "Group": [
    {
      "dbpath": "/home/users/chenzichuan/sequoiadb/cata",
      "HostName": "ubuntu-dev2",
      "Service": [
        {
          "Type": 0,
          "Name": "11800"
        },
        {
          "Type": 1,
        }
      ]
    }
  ]
}
```

```

        "Name": "11801"
    },
{
    "Type": 2,
    "Name": "11802"
},
{
    "Type": 3,
    "Name": "11803"
}
],
"NodeID": 1
},
],
"GroupID": 1,
"GroupName": "SYSCatalogGroup",
"PrimaryNode": 1,
"Role": 2,
"Status": 1,
"Version": 1,
"_id": {
    "$oid": "51710981d8cb8fbc163d6350"
}
}
}

```

引擎调度单元

概念

引擎调度单元 (Engine Dispatchable Unit) 是 SequoiaDB 数据库中任务运行的载体，一般来说一个 EDU 意味着一个单独的线程。

每个 EDU 可以用来执行用户的请求，或者执行系统内部的维护任务。

EDU 之间相互独立，不同 EDU 单独负责不同的用户会话。一个用户会话与一个 EDU，在一个数据节点中相互绑定。

每个 EDU 拥有一个进程内唯一的64位整数标示，称作“EDU ID”。

EDU 可以分为用户 EDU 与系统 EDU，分别代表执行用户任务的线程，与执行系统任务的线程。

用户 EDU

用户 EDU 为执行用户任务的线程，一般又叫作代理 (Agent) 线程。

在 SequoiaDB 中，主要存在下列代理线程类型：

名称	类型	描述
Agent	代理	代理线程负责由 svcname 服务传入的请求，一般来说该请求由用户直接传入
ShardAgent	分区代理	分区代理线程负责由 shardname 服务传入的请求，一般来说该请求由协调节点传入数据节点
CoordAgent	协调代理	协调代理线程负责由 svcname 服务传入的请求，一般来说该请求由用户直接传入，仅作用于协调节点
ReplAgent	复制代理	复制代理线程负责由 replname 服务传入的请求，一般来说该请求由数据主节点传向从节点，多作用于非协调节点
HTTPAgent	HTTP 代理	HTTP 代理线程负责由 httpname 服务传入的 REST 请求，一般来说该请求由用户直接传入

系统 EDU

系统 EDU 为系统内部维护数据结构及一致性的线程，一般来说对用户完全透明。

在 SequoiaDB 中，存在但不局限于下列系统 EDU：

名称	类型	描述
TCPListener	服务监听	该线程负责监听 svcname 服务，并启动 Agent 代理线程
HTTPListener	HTTP 监听	该线程负责监听 httpname 服务，并启动 Agent 代理线程
Cluster	集群管理	集群管理线程用于维护集群的基本框架，启动 ReplReader 与 ShardReader 线程
ReplReader	复制监听	复制监听线程负责由 replname 服务传入的请求，并启动 ReplAgent 代理线程
ShardReader	分区监听	分区监听线程负责由 shardname 服务传入的请求，并启动 ShardAgent 代理线程
LogWriter	日志写	日志写线程用于将日志缓冲区中的数据写入日志文件
WindowsListener	Windows 事件监听	Windows 环境特有，用于监听 Windows 中 SequoiaDB 定义事件
Task	后台任务处理	后台任务处理线程，一般来说用于处理后台任务请求，例如 数据切分
CatalogMC	编目主控	编目主控线程用于接收和分发编目节点接收到的请求
CatalogNM	编目节点控制	编目节点控制线程用于处理编目节点内部集群信息相关的请求
CatalogManager	编目控制	编目控制线程用于处理编目节点内部元数据相关的请求
CatalogNetwork	编目网络监听	编目网络监听线程用于监听编目服务 catalogname 下的请求
CoordNetwork	协调网络监听	协调网络监听线程用于监听分区的请求

监控

用户可以使用[会话快照](#)监控系统与用户 EDU。

日志

同步日志

同步日志

日志文件

SequoiaDB 采用日志方式进行副本间的数据同步。日志文件存在于 replicalog 目录中。文件大小和个数可以分别通过 logfilesz 和 logfilenum 参数进行设置。默认分别为 64M（不包含头大小）和 20。参数生效后无法修改。（如果要修改必须离线删除全部日志文件，重新配置参数并启动 SequoiaDB。但此举通常会引起全量同步。）

同步

数据组内所有备节点会定期将其他数据节点日志打包下载到本地进行日志回放。同步源并不限于主节点。因为我们期望所有节点的数据版本差距在一个很小的窗口内。当处于这个窗口内时，所有备节点向主节点同步数据。但是当某些节点的数据版本与主节点相差过大时，则选择其他备节点进行同步。当发生版本冲突

时，以当前主节点数据版本为准。如果冲突不能解决则进入全量同步。当组内不存在主节点时，同步无法进行。

全量同步

触发全量同步的原因有：

1. 容机重启。
2. 节点数据版本与其他节点相差过大。
3. 数据不一致并且无法修复。

 注：正常重启后，如果数据版本仍在可同步范围内则不会触发全量同步。

发生全量同步的节点会清空本地所有数据及日志，同时将组内另一个节点（不限于主节点）的数据全部复制到本地。期间同步源发生的数据改变同样会被复制到本地。全量同步期间本节点对外不提供服务。当组内不存在主节点时，全量同步无法进行。全量同步会极大地影响整个组的性能，甚至导致其他备节点同步性能降低。建议通过增加分区及日志容量来避免全量同步。

数据库工具

目录：

[数据迁移 — 导入](#)

[数据迁移 — 导出](#)

[数据库检测工具 — sdbdmsdump](#)

[数据库性能监控工具 — sdbtop](#)

[数据库集群节点数据一致性检测工具 — sdbinspect](#)

[数据库信息收集工具 — sdbsupport](#)

[数据迁移 — 导入](#)

sdbimprt

sdbimprt 是一个 SequoiaDB 数据库导入工具，它可以实现从 JSON 格式或 CSV 格式的数据存储文件导入到 SequoiaDB 数据库。

JSON 格式的记录必须符合 Json 的定义，以左右花括号作为记录的分界符，并且字符串类型的数据必须包含在两个双引号之间，转义字符定义为反斜杠。

CSV 为 Comma-Separated Value 格式，为逗号分割数值。默认情况下每条记录以 0x0D 分割，字段之间以逗号分割。用户能够指定字段之间的分隔符与记录之间的分隔符。

分隔符定义（只支持 ASCII 字符）：

用途	默认	可配
字符串分隔符	"	是
字符分隔符	,	是
记录分割符	'\n' (0x0D)	是

选项

参数	缩写	描述
--help	-h	返回基本帮助和用法文本。
--version		返回版本信息。

参数	缩写	描述
--hostname	-s	从指定主机名的 SequoiaDB 中导入数据。默认情况下 sdbimprt 尝试连接到本地主机。
--svcname	-p	指定的端口号。默认情况下 sdbimprt 尝试连接到端口号11810的主机。
--user	-u	数据库用户名。
--password	-w	数据库密码。
--delchar	-a	指定字符分隔符。默认是 (") , csv 格式有效。
--delfield	-e	指定字段分隔符。默认是 (,) , csv 格式有效。
--delrecord	-r	指定记录分隔符。默认是 (\n) , csv 格式有效。
--csname	-c	指定导入数据的集合空间名。
--cname	-l	指定导入数据的集合名。
--insertnum	-n	批量导入记录，如果设置1，使用普通方式导入，如果大于1，则使用批量方式，设置范围是1~100000。
--file		指定要导入的文件名。
--type		指定的导入数据格式。默认 csv，数据格式可以是 csv 或是 json。
--fields		指定导入数据的字段名。csv 格式有效。
--headerline		指定导入数据首行是否作为字段名，默认 false，csv 格式有效。
--sparse		指定导入数据时，自动添加字段名，默认 true，csv 格式有效。
--extra		指定导入数据时，自动添加值，默认 false，csv 格式有效。
--linepriority		当前分隔符默认的优先级为：记录分隔符，字符分隔符，字段分隔符，默认值是 true；如果设置为 false，那么分隔符的优先级为：字符分隔符，记录分隔符，字段分隔符。
--force		如果数据中有非 utf8 的数据，强制导入数据，默认 false。
--errorstop		如果遇到错误就停止。默认 false。

 注: linepriority 参数需要被特别关注，如果设置不当，可能会导入数据失败。当记录中包含“记录分隔符”并且 linepriority 为 true 时，工具会优先按照“记录分隔符”解析，而导致导入失败。比如：如果记录为 {"name": "Mike\n"}, 应当选择 linepriority 为 false 模式。

返回值

0 : 成功

1 : 成功但有警告

2 : 失败

127 : 参数错误

用法

在下面的例子，sdbimprt 将数据导入到本地数据库端口11810中对应集合空间 foo 和集合 bar，导入类型是 csv，导入文件为 test.csv。

```
sdbimprt -s localhost -p 11810 --type csv --file test.csv -c foo -l bar
```

sdbimprt 自动判断的类型：

CSV 数据	sdbimprt 自动判断的类型	实际数据
123	数值	123
123.1	数值	123.1
+123	数值	123
-123	数值	-123
2E+2	数值	200
-2E+2	数值	-200
0123	数值	123
"123"	字符串	123
123a	字符串	123a
"ab""c"	字符串	ab"c
{"a:1"}	字符串	{a:1}
"true"	字符串	true
"false"	字符串	false
"null"	字符串	null
true	布尔	true
false	布尔	false
null	空	null

从1.8版本开始支持指定类型和默认值：

语法：field [type] [default <default value]

支援数据类型 (type) : int (integer) , long , bool (boolean) , double , string , null

支援特殊数据类型 : timestamp , date

例子：

```
name string default "Jack", age int default 18, phone string
```

数据类型是可选的，不填则程序自动判断类型，判断以上面表格为准。

用法

在下面的例子，sdbimprt 将数据导入到本地数据库端口11810中对应集合空间 foo 和集合 bar ，导入类型是 csv ，导入文件为 test.csv。

```
sdbimprt -s localhost -p 11810 --type csv --file test.csv -c foo -l bar --fields 'name string
default "Jack", age int default 18, phone string'
```

 注: 指定字段可以用命令行指定，也可以在导入文件的首行指定。如果在命令行指定了--fields , --headerline 设为 true , 导入工具将会优先使用命令行的指定字段并且跳过导入文件的首行。

- 例子一：导入文件是 csv ，文件名是 test.csv ，导入至集合空间 foo 的集合 bar 中。

以下是导入文件的内容：

```
"Jack",18,"China"
"Mike",20,"USA"
```

导入命令：

```
sdbimprt -s localhost -p 11810 --type csv --file test.csv -c foo -l bar --fields 'name
string default "Anonymous", age int, country'
```

- 例子二：导入文件是 csv ，文件名是 test.csv ，导入至集合空间 foo 的集合 bar 中。

以下是导入文件的内容：

```
name, age, country
"Jack",18,"China"
"Mike",20,"USA"
```

导入命令：

```
sdbimp -s localhost -p 11810 --type csv --file test.csv -c foo -l bar --fields 'name
string default "Anonymous", age int, country' --headerline true
```



注：

由于例子二首行已经指定字段，但是要重新指定字段，那么在命令行中设置--headerline true，并且设置--fields 'name string default "Anonymous", age int, country'，那么导入工具会使用--fields 为指定字段并忽略文件首行的字段了。

指定的类型与数据：

指定的类型	支援的数据类型	注意
int (integer)	int , string , bool	bool 类型，true 转换1，false 转换0
long	long , string , int , bool	bool 类型，true 转换1，false 转换0
bool (boolean)	bool , string , int , long	int 和 long 类型，0转换 false，不等于0转换 true
double	double , string , int	
string	int , long , bool , double , string , timestamp , date , null	
timestamp	string , long	如果数据是 string，如“2014-01-01-10.30.00.000000”，“1388543400000”，如 果是 long，如 1388543400000，这两种格式是正 确的，数值代表的是毫秒单位
date	string , long	如果数据是 string，如“2014-01-01”，“1388543400”，如果是 long，如1388543400，这两种格式是正确的，数 值代表的秒单位
null		指定了数据类型为 null，那么无论数据是什 么，都是 null

数据类型优先级：

1. 不指定数据类型（不支援特殊数据类型）：程序自动判断类型，优先级是 null > bool > int > double > long > string
2. 指定数据类型：优先级是指定的类型 > 支援的类型（参考指定的类型与数据表格）> null
3. 指定数据类型，并且带默认值：优先级是指定的类型 > 支援的类型（参考指定的类型与数据表格）> 默认
值 > null

数据迁移 — 导出

sdbexprt

sdbexprt 是一个实用的工具。它可以从 SequoiaDB 数据库导出一个 JSON 格式或者 CSV 格式的数据存储文
件。

选项

参数	缩写	描述
--help	-h	返回基本帮助和用法文本。
--version		返回版本信息。

参数	缩写	描述
--hostname	-s	从指定主机名的 SequoiaDB 中导出数据。默认情况下 sdbexprt 尝试连接到本地主机。
--svcname	-p	指定的端口号。默认情况下 sdbexprt 尝试连接到端口号11810的主机。
--user	-u	数据库用户名。
--password	-w	数据库密码。
--delchar	-a	指定字符分隔符。默认是 (") , csv 格式有效。
--delfield	-e	指定字段分隔符。默认是 (,) , csv 格式有效。
--delrecord	-r	指定记录分隔符。默认是 (\n) 。
--csname	-c	指定导出数据的集合空间名。
--cname	-l	指定导出数的集合名。
--fields		指定一个或多个字段来导出数据，使用逗号分隔多个字段。csv 格式有效。
--included		指定是否导出字段名到 csv 首行，默认 true , csv 格式有效。
--file		指定要导出的文件名。
--type		指定的导出数据格式。默认 csv , 数据格式可以是 csv 或 json。
--errorstop		如果遇到错误就停止， 默认 false。

返回值

0 : 成功

1 : 成功但有警告

2 : 失败

127 : 参数错误

用法

在下面的例子，sdbexprt 从本地数据库端口11810中导出集合空间 foo 的集合 bar 的数据，导出类型是 csv , 导出文件为 contact , 导出字段是 field1 和 field2。

```
sdbexprt -s localhost -p 11810 --type csv --file contact --fields field1,field2 -c foo -l bar
```

数据库检测工具 — sdbdmsdump

sdbdmsdump (1.8 版本前名为 sdbinspt , 1.8 版本后更名为 sdbdmsdump) 是一个 SequoiaDB 数据库的数据文件检测工具。它可以检查数据库文件结构的正确性，并且给出结果报告。

权限需求

运行 sdbdmsdump 命令的用户必须对数据库的数据与索引文件拥有读权限。

连接需求

sdbdmsdump 不需要与数据库连接。

选项

参数	缩写	描述
--help	-h	返回基本帮助和用法文本
--dbpath	-d	指定数据库文件所在目录，默认为当前目录

参数	缩写	描述
--output	-o	指定输出文件，默认为屏幕输出
--verbose	-v	是否进行 ASCII 文本输出 (true/false) , 默认为 true
--csname	-c	指定集合空间名，如果未指定则为全部集合空间
--clname	-l	指定集合名，如果未指定则为全部集合
--action	-a	指定操作，为 (inspect/dump/all) 之一，必须指定 inspect：检测并报告任何数据损坏 dump：将数据页格式化并输出 all：检测数据页损坏，并格式化输出数据页
--dumpdata	-t	设定操作数据文件 (true/false) , 默认为 false
--dumpindex	-i	设定操作索引文件 (true/false) , 默认为 false
--pagestart	-s	指定起始数据页，默认为-1
--numpage	-n	指定需要检测或格式化的数据页数量，当指定 -s 参数为非负值时，该参数生效。默认值为1
--record	-p	指定显示格式化输出数据或索引内容 (true/false) , 默认为 false

用法

使用 sdbdmsdump 工具时，请务必保证数据库进程已经停止。

在下面的例子，sdbdmsdump 在当前目录下检测并格式化输出所有集合空间与集合的数据与索引至 output.txt 文件。

```
sdbdmsdump -d . -o output.txt -v true -a all -t true -i true -p true
```

数据库性能监控工具 — sdbtop

sdbtop 是一个 SequoiaDB 数据库的性能监控工具。通过它，可以监控和查看集群中各个节点的监视信息。

选项

参数	缩写	描述
--help	-h	返回基本帮助和用法文本
--confpath	-c	sdbtop 的配置文件，sdbtop 界面形态以及输出字段都依赖该文件（缺省使用默认配置文件）
--hostname	-i	指定需要监控的主机名
--servicename	-s	指定监控的端口服务名
--username	-u	数据库用户名
--password	-p	数据库密码

 注：对于 Ubuntu 等系统，需要安装 Ncurses 库，否则将会提示“Error opening terminal: TERM”

方式一：联网安装

```
sudo apt-get install libncurses5-dev
```

方式二：源码安装

解压 tar -xvzf ncurses-5.5.tar.gz，进入 ncurses-5.5 目录

```
./configure  
sudo make && make install
```

用法

在下面的例子，sdbtop 使用配置文件为“/opt/sequoiadb/conf/sdbtop.xml”，监控主机名为 sdbserver3，端口服务名为 11810，用户名为 test，密码为 test 的数据库集群中的一个节点。

```
sdbtop -c /opt/sequoiadb/conf/sdbtop.xml -i sdbserver3 -s 11810 -u test -p test
```

接着进入主窗口：

```
refresh= 3 secs          sdbtop 1.0          snapshotMode:
GLOBAL                   Main Window        snapshotModeInput:
displayMode: ABSOLUTE      NULL              filtering
NULL
hostname: sdbserver3       servicename: 11810    sortingWay: NULL sortingField:
Number: 0                  NULL
serviceName: 11810         usrName: test      Refresh: F5, Quit: q,
NULL
userName: test             Help: h

##### ###### ##### ###### ##### ###### For help type h or ...
#   #   # #   #   #   # #   #   #   # sdbtop -h: usage
#   #   # #   #   #   # #   #   #
##### #   # ##### #   #   #   # ######
#   #   # #   #   #   #   #   #   #
#   #   # #   #   #   #   #   #   #
##### ##### ##### #   ##### #   #

SDB Interactive Snapshot Monitor V2.0
Use these keys to navigate:
m   - Main Window           s   - Sessions           c   - CollectionSpaces
t   - System                 d   - Database            G   - GLOBAL_SNAPSHOT
g   - GROUP_SNAPSHOT         n   - NODE_SNAPSHOT       r   - reset refreshInterval
A   - Ascending order        D   - Descending order     C   - filter condition
Q   - no filter condition    N   - filter number       W   - no filter number
```

Licensed Materials – Property of SequoiaDB
Copyright SequoiaDB Corp. 2013–2014 All Rights Reserved.

注：在主窗口中按‘h’键可以查看所有工具支持的按键



- 主窗口按键说明

参数	描述
m	返回主窗口
s	列出数据库节点上的所有会话
c	列出数据库节点上的所有集合空间
t	列出数据库节点上的系统资源使用情况
d	列出数据库节点的数据库监视信息
G	global_snapshot，监控所有的数据节点组
g	group_snapshot，指定监控某个数据节点组
n	node_snapshot，列出指定的数据库节点的监视信息
r	设置刷屏的时间间隔，单位秒/s
A	将监视信息按照某列进行顺序排序
D	将监视信息按照某列进行逆序排序
C	将监视信息按照某个条件进行筛选
Q	返回没有使用条件进行筛选前的监视信息
N	将监视信息中对应行号的记录过滤不显示

参数	描述
W	返回没有使用行号进行过滤前的监视信息
h	查看使用帮助
Esc	取消已进入的操作
Enter	返回上一次监视界面，(在已进入 help 帮助输出中有效)
F5	强制刷新后台监视信息
<	向左移动，以查看隐藏的左边列的监视信息
>	向右移动，以查看隐藏的右边列的监视信息
q	退出程序
Tab	切换数据计算的模式(绝对值，平均值，差值三个模式)

- 例子一：

1. 进入主窗口后，按's'键，列出数据库节点的所有会话信息

```
refresh= 3 secs           sdbtop 1.0          snapshotMode:
GLOBAL
displayMode: ABSOLUTE      Sessions
snapshotModeInput: NULL
hostname: sdbserver3        filtering
Number: 0
servicename: 11810          sortingWay: NULL
sortingField: NULL
userName: test
Help: h

SessionID          TID Type          Name
-----          -----
1  sdbserver3:11820:1    10732 LogWriter    ""
2  sdbserver3:11820:10   10741 Task        Job [Prefetcher]
3  sdbserver3:11820:11   10742 Task        Job [Prefetcher]
4  sdbserver3:11820:12   10743 Task        Job [Prefetcher]
5  sdbserver3:11820:13   10744 Cluster     ""
6  sdbserver3:11820:14   10745 ClusterShard ""
7  sdbserver3:11820:15   10746 ClusterLogNotify ""
8  sdbserver3:11820:16   10747 ShardReader  ""
9  sdbserver3:11820:17   10748 Rep1Reader  ""
10 sdbserver3:11820:18   10749 SyncClockWorker ""
11 sdbserver3:11820:19   10750 TCPLListener ""
12 sdbserver3:11820:2    10733 DpsRollback ""
13 sdbserver3:11820:20   10751 RestListener ""
14 sdbserver3:11820:21   10752 Task        Job [PageCleaner]
15 sdbserver3:11820:3    10734 Task        Job [Prefetcher]
16 sdbserver3:11820:4    10735 Task        Job [Prefetcher]
17 sdbserver3:11820:42   10847 Rep1Agent   ""
NodeID: 1000, TID: 1, Start: active
18 sdbserver3:11820:5    10736 Task        Job [Prefetcher]
19 sdbserver3:11820:59   23263 ShardAgent  NetID: 1, TID: 23262
20 sdbserver3:11820:6    10737 Task        Job [Prefetcher]
21 sdbserver3:11820:7    10738 Task        Job [Prefetcher]
22 sdbserver3:11820:8    10739 Task        Job [Prefetcher]
```

2. 按'Tab'键，可以看到屏幕左上方的'displayMode'的值会发生切换

3. 按'r'键，在屏幕最下方输入'2'，回车，设置刷新间隔时间，可以看到屏幕左上方的'refresh'的值变为 2

```
refresh= 2 secs           sdbtop 1.0          snapshotMode:
GLOBAL
displayMode: ABSOLUTE      Sessions
snapshotModeInput: NULL
hostname: sdbserver3        filtering
Number: 0
servicename: 11810          sortingWay: NULL
sortingField: NULL
```

			Refresh: F5, Quit: q,
	SessionID	TID Type	Name
1	sdbserver3: 11820: 1	10732	LogWriter
2	sdbserver3: 11820: 10	10741	Task
3	sdbserver3: 11820: 11	10742	Task
4	sdbserver3: 11820: 12	10743	Task
5	sdbserver3: 11820: 13	10744	Cluster
6	sdbserver3: 11820: 14	10745	ClusterShard
7	sdbserver3: 11820: 15	10746	ClusterLogNotify
8	sdbserver3: 11820: 16	10747	ShardReader
9	sdbserver3: 11820: 17	10748	ReplReader
10	sdbserver3: 11820: 18	10749	SyncClockWorker
11	sdbserver3: 11820: 19	10750	TCPListener
12	sdbserver3: 11820: 2	10733	DpsRollback
13	sdbserver3: 11820: 20	10751	RestListener
14	sdbserver3: 11820: 21	10752	Task
15	sdbserver3: 11820: 3	10734	Task
16	sdbserver3: 11820: 4	10735	Task
17	sdbserver3: 11820: 42	10847	ReplAgent
NodeID: 1000, TID: 1, Start: active			
18	sdbserver3: 11820: 5	10736	Task
19	sdbserver3: 11820: 59	23263	ShardAgent
20	sdbserver3: 11820: 6	10737	Task
21	sdbserver3: 11820: 7	10738	Task
22	sdbserver3: 11820: 8	10739	Task

4. 按'A'键，并输入'TID'，列表结果按照 TID 进行顺序排序

refresh= 2 secs	sdbtop 1.0	snapshotMode:
GLOBAL		
displayMode: ABSOLUTE	Sessions	
snapshotModeInput: NULL		
hostname: sdbserver3		filtering
Number: 0		
servicename: 11810		sortingWay: NULL
sortingField: NULL		
usrName: test		Refresh: F5, Quit: q,
Help: h		
SessionID	TID Type	Name
1 sdbserver3: 11820: 1	10732	LogWriter
2 sdbserver3: 11820: 10	10741	Task
3 sdbserver3: 11820: 11	10742	Task
4 sdbserver3: 11820: 12	10743	Task
5 sdbserver3: 11820: 13	10744	Cluster
6 sdbserver3: 11820: 14	10745	ClusterShard
7 sdbserver3: 11820: 15	10746	ClusterLogNotify
8 sdbserver3: 11820: 16	10747	ShardReader
9 sdbserver3: 11820: 17	10748	ReplReader
10 sdbserver3: 11820: 18	10749	SyncClockWorker
11 sdbserver3: 11820: 19	10750	TCPListener
12 sdbserver3: 11820: 2	10733	DpsRollback
13 sdbserver3: 11820: 20	10751	RestListener
14 sdbserver3: 11820: 21	10752	Task
15 sdbserver3: 11820: 3	10734	Task
16 sdbserver3: 11820: 4	10735	Task
17 sdbserver3: 11820: 42	10847	ReplAgent
NodeID: 1000, TID: 1, Start: active		

```

18 sdbserver3: 11820: 5          10736 Task        Job [Prefetcher]
19 sdbserver3: 11820: 59         23263 ShardAgent NetID: 1, TID: 23262
20 sdbserver3: 11820: 6          10737 Task        Job [Prefetcher]
21 sdbserver3: 11820: 7          10738 Task        Job [Prefetcher]
please input the displayName which need order by asc : TID

```

5. 按'N'键，并输入1，列表中将原来行号为1的记录过滤不显示

6. 按'W'键，返回没有按行号进行过滤前的列表信息

7. 按'C'键，并输入'TID : 10732'进行筛选，则只显示 TID 值为10732的记录

```

refresh= 2 secs                  sdbtop 1.0           snapshotMode:
GLOBAL                           Sessions
displayMode: ABSOLUTE            filtering
snapshotModeInput: NULL
hostname: sdbserver3
Number: 0
servicename: 11810
sortingField: NULL
userName: test
Help: h

SessionID                         TID Type      Name
-----                         -----
1 sdbserver3: 11820: 1           10732 LogWriter  ""
2 sdbserver3: 11820: 10          10741 Task       Job [Prefetcher]
3 sdbserver3: 11820: 11          10742 Task       Job [Prefetcher]
4 sdbserver3: 11820: 12          10743 Task       Job [Prefetcher]
5 sdbserver3: 11820: 13          10744 Cluster    ""
6 sdbserver3: 11820: 14          10745 ClusterShard ""
7 sdbserver3: 11820: 15          10746 ClusterLogNotify ""
8 sdbserver3: 11820: 16          10747 ShardReader  ""
9 sdbserver3: 11820: 17          10748 Rep1Reader   ""
10 sdbserver3: 11820: 18          10749 SyncClockWorker ""
11 sdbserver3: 11820: 19          10750 TCPLListener ""
12 sdbserver3: 11820: 2           10733 DpsRollback ""
13 sdbserver3: 11820: 20          10751 RestListener ""
14 sdbserver3: 11820: 21          10752 Task        Job [PageCleaner]
15 sdbserver3: 11820: 3           10734 Task       Job [Prefetcher]
16 sdbserver3: 11820: 4           10735 Task       Job [Prefetcher]
17 sdbserver3: 11820: 42          10847 Rep1Agent   ""
NodeID: 1000, TID: 1, Start: active
18 sdbserver3: 11820: 5           10736 Task       Job [Prefetcher]
19 sdbserver3: 11820: 59          23263 ShardAgent NetID: 1, TID: 23262
20 sdbserver3: 11820: 6           10737 Task       Job [Prefetcher]
21 sdbserver3: 11820: 7           10738 Task       Job [Prefetcher]
please input the filter condition : TID: 10732

```

8. 按'Q'键，返回没有按照筛选条件前的列表信息

9. 按'<或者>'键，可以查看隐藏在左边或者右边的列

数据库集群节点数据一致性检测工具 — sdbinspect

sdbinspect 是一个 SequoiaDB 数据库的数据节点间数据一致性检测工具。它可以检查节点间数据是否完全一致，并且给出结果报告。

权限需求

无

连接需求

sdbinsepct 需要与数据库（ coord 节点）连接。

选项

参数	缩写	描述
--help	-h	返回基本帮助和用法文本
--version	-v	返回当前工具所附属的数据库的版本
--action	-a	指定检查数据或对已经存在的中间文件生成 report , inspect 和 report 可选，默认是 inspect
--coord	-d	指定 coord 节点的 hostname 和服务端口，格式为 hostname:servicename , 必须指定
--loop	-t	指定迭代检查的次数，默认是5（次）
--group	-g	指定要检查的 group 的名字，若不指定，则检查所有的 group
--collectionspace	-c	指定检查的集合空间名字，不指定则检查所有集合空间
--collection	-l	指定检查的集合名字，不指定则检查所有集合，指定集合时，必须制定集合空间
--file	-f	指定从已存在的（上一次检查的）结果文件开始检查，当指定此选择时，其它选项（除 -o 外）均失效，生效的为文件中保存的 command 选项
--output	-o	指定输出的文件名，默认是 inspect.bin , 报告文件为 inspect.bin.report
--view	-w	指定生成 report 文件的内容按 group 查看和按 collection 查看，默认为 group

用法

在下面的例子，sdbinspect 检查协调节点 ubuntu-dev9:11810 下的全部集群（5次），并将中间文件结果输出到 item.bin 中，同时会解析 item.bin 文件，把文本结果按（默认的）group 划分，输出到 item.bin.report 文件中。

```
sdbinspect -d ubuntu-dev9:11810 #o item.bin
```

在下面的例子，sdbinspect 检查协调节点 ubuntu-dev9:11810 下的全部集群中的集合空间 sports (3次)，并将中间文件结果输出到 item.bin 中，同时会解析 item.bin 文件，把文本结果按 collection 划分，输出到 item.bin.report 文件中。

```
sdbinspect -d ubuntu-dev9:11810 #o item.bin #c sports #w collection #t 3
```

在下面的例子，sdbinspect 检查协调节点 ubuntu-dev9:11810 下的 data_group 集群中的名为 sports 的集合空间，名为 item 的集合（5次），并将中间文件结果输出到 inspect.bin 中，同时会解析 inspect.bin 文件，把文本结果按（默认的）group 划分，输出到 inspect.bin.report 文件中。

```
sdbinspect -d ubuntu-dev9:11810 #g data_group #c sports #l item
```

数据库信息收集工具 — sdbsupport

sdbsupport 是用于收集 SequoiaDB 相关信息的工具，位于目录 /opt/sequoiadb/tools 下面。此工具收集的信息包括：数据库配置信息、数据库日志信息、数据库所在主机的硬件信息和数据库、操作系统信息以及数据库快照信息。

使用此工具需要先为 sdbsupport.sh 赋执行权限：

```
chmod 755 sdbsupport.sh
```

权限需求

数据库用户权限。

选项

参数	缩写	描述
--help		帮助选项
--hostname arg	-N arg	所需要收集的信息的主机名字
--svcport arg	-p arg	指定特定端口收集其配置、日志及快照信息
--snapshot	-s	收集快照信息
--osinfo	-o	收集操作系统信息
--hardware	-h	收集硬件信息
--all		指定收集数据库所有信息
--conf		指定收集配置文件的信息
--log		指定收集日志文件信息
--cm		指定收集 CM 配置、日志信息
--cpu		指定收集 CPU 信息
--memory		指定收集内存信息
--disk		指定收集硬盘信息
--netcard		指定收集网卡信息
--mainboard		指定收集主板信息
--catalog		指定收集编目节点快照
--group		指定收集数据库集群组的信息
--context		指定收集上下文快照信息
--session		指定收集会话快照信息
--collection		指定收集集合快照信息
--collectionspace		指定收集集合空间信息
--database		指定收集数据库快照信息
--system		指定收集系统快照信息
--diskmanage		指定收集操作系统硬盘管理信息
--basicsys		指定收集操作系统基本信息
--module		指定收集内核加载模块信息
--env		指定收集操作系统环境变量信息
--network		指定收集 IP 地址等网络信息
--process		指定收集操作系统进程信息
--login		指定收集用户登陆此机所进行操作的历史信息
--limit		指定收集操作系统用户限制信息
--vmstat		指定收集给定时间间隔内的服务状态值信息

用法示例

1. 获取参数信息。

```
./sdbsupport.sh --help
```

2. 收集本机的数据信息。【包括配置、日志、硬件、操作系统及快照信息】

```
./sdbsupport.sh
```

3. 收集整个数据库集群信息

```
./sdbsupport.sh --all
```

4. 收集指定主机信息。

```
./sdbsuport.sh -N htest2
```

5. 收集指定主机指定端口信息。

```
./sdbsuppor.sh -N htest3 -p 50000
```

6. 收集操作系统信息。

```
./sdbsupport.h --info
```

7. 收集特定主机特定端口的日志信息及快照信息。

```
./sdbsupport.sh -N htest2 -p 11810 --snapshot --log
```

信息归类

通过执行 `./sdbsupport.sh xxx xxx....` 收集的数据库信息信息，会全部收集到本地的 log 文件夹中。收集的信息是以主机为单位打成的压缩包，名称以“主机名·年月日-时分秒”命名。将此文件解压缩后会得四个文件夹：SDBNODES，SDBSNAPS，OSINFO，HARDINFO。

SDBNODES：存放收集的数据库配置、日志信息

SDBSNAPS：存放收集的数据库快照信息

OSINFO：存放收集的操作系统信息

HARDINFO：存放收集的硬件信息

 注：

数据库集群内的机器，如果没有配置信任关系，在收集时，需要输入密码，如：

```
/opt/sequoiadb/tools/sdbsupport/expect/expect
Success to export System environment variable : /opt/sequoiadb/tools/sdbsupport/expect/
Check over Environment!
Complete database database cluster
The host sdbadmin@htest2's password :
```

【此时需要输入 htest2 机器，sdbadmin 用户的密码，然后 Enter】

集群管理

本章描述了集群的一些管理概念和操作，包括编目/数据分区组的创建/删除、协调节点的增加。

通过理解这些概念和操作方法，更好地实现集群规模的扩展或调整。

目录：

[集群中新增主机](#)

[编目分区组管理](#)

[数据分区组管理](#)

[协调分区管理](#)

[案例](#)

集群中新增主机

如果需要在数据库集群中新增一台主机（物理机或者虚拟机），用于部署编目节点或者数据节点，则必须需要按照如下步骤配置好主机系统：

1. 安装好与其他主机相同的操作系统，并配置好IP地址；
2. 按照[系统配置需求](#)一节配置好主机名/内核参数，并将其他的主机名和IP对应关系加入到 /etc/hosts 中；
3. 修改每台集群主机上 /etc/hosts 文件，将新增的主机IP地址与主机对应关系加入到 /etc/hosts 文件中；
4. 按照[系统配置需求](#)一节验证配置的正确性；
5. 按照[SequoiaDB 服务器安装](#)一节，安装 SequoiaDB 软件。安装时，注意配置管理服务端口与现有系统的端口保持一致。

编目分区组管理

本小结介绍在编目分区中新增分区组和新增节点。

目录：

[新建编目分区组](#)

[编目分区组中新增节点](#)

[新建编目分区组](#)

 注：如果新增节点涉及到新增主机，则请首先按照在[集群中新增主机](#)一节完成主机的主机名和参数配置。

一个数据库集群必须有且仅有一个编目分区组，所以新建分区组往往在安装时就已经完成，不需要在安装后执行新建分区组操作。实例见[安装指南](#)[集群模式的配置与启动](#)一节。

操作方法：

```
> db.createReplicaCataGroup(<host>, <service>, <dbpath>, [config])
```

该命令用于创建编目分区组，同时创建并启动一个编目节点，其中：

host：指定编目节点的主机名；

service：指定编目节点的服务端口，请确保该端口号，以及往后延续的3个端口号未被占用；如设置为11800，请确保11800/11801/11802/11803端口都未被占用；

dbpath：数据文件路径，用于存放编目数据文件，请确保数据管理员（安装时创建，默认为 sdbadmin）用户有写权限；

config：该参数为可选参数，用于配置更多细节参数，格式必须为 json 格式，参数参见[数据库配置](#)一节；如需要配置日志大小参数 { logfilesz:64 } 。

[编目分区组中新增节点](#)

 注：如果新增节点涉及到新增主机，则请首先按照在[集群中新增主机](#)一节完成主机的主机名和参数配置。

随着整个集群中的物理设备的扩展，可以通过增加更多的编目节点来提高编目服务的可靠性。

操作方法：

```
> var cataRG = db.getRG("SYSCatalog");
> var node1 = cataRG.createNode(<host>, <service>, <dbpath>, [config]);
> node1.start()
```

第一条命令用于获取编目分区组，“SYSCatalogGroup”为编目分区组组名；

第二条命令用于创建一个新的编目节点，其中：

参数：

host：指定编目节点的主机名；

service：指定编目节点的服务端口，请确保该端口号，以及往后延续的3个端口号未被占用；如设置为11800，请确保11800/11801/11802/11803端口都未被占用；

dbpath : 数据文件路径 , 用于存放编目数据文件 , 请确保数据管理员 (安装时创建 , 默认为 sdbadmin) 用户有写权限 ;

config : 该参数为可选参数 , 用于配置更多细节参数 , 格式必须为 json 格式 , 参数参见[数据库配置](#)一节 ; 如需要配置日志大小参数 { logfilesz:64 } 。

第三条命令用于启动新增的编目节点。

数据分区组管理

本小结介绍在数据分区中新增分区组和新增节点。

目录 :

[新增数据分区组](#)

[分区组中新增节点](#)

新增数据分区组

 注: 如果新增节点涉及到新增主机 , 则请首先按照在[集群中新增主机](#)一节完成主机的主机名和参数配置。

一个集群中可以配置多个分区组 , 最大可配置 60,000 个分区组。通过增加分区组 , 可以充分利用物理设备进行水平扩展 , 理论上 SequoiaDB 可以做到线性的水平扩展能力。

操作方法 :

```
> var dataRG = db.createRG("datagroup1")
> dataRG.createNode("sdbserver1", 11820, "/opt/sequoiadb/database/data/11820")
> dataRG.start()
```

第一条命令用于创建数据分区组 , 与编目分区组不同的是 , 该操作不会创建任何数据节点 , 其中参数为数据组名 ;

第二条命令用户在数据组中新增一个数据节点 , 可以根据需要多次执行该命令来创建多个数据节点。

其中 :

host : 指定数据节点的主机名 ;

service : 指定数据节点的服务端口 , 请确保该端口号 , 以及往后延续的 3 个端口号未被占用 ; 如设置为 11820 , 请确保 11820/11821/11822/11823 端口都未被占用 ;

dbpath : 数据文件路径 , 用于存放数据节点的数据文件 , 请确保数据管理员 (安装时创建 , 默认为 sdbadmin) 用户有写权限 ;

config : 该参数为可选参数 , 用于配置更多细节参数 , 格式必须为 json 格式 , 参数参见[数据库配置](#)一节 ; 如需要配置日志大小参数 { logfilesz:64 } 。

第三条命令用于启动数据分区组 , 该命令将该组的所有节点启动 , 并提供服务。

分区组中新增节点

 注: 如果新增节点涉及到新增主机 , 则请首先按照在[集群中新增主机](#)一节完成主机的主机名和参数配置。

某些分区组可能在创建时设定的副本数较少 , 随着物理设备的增加 , 可能需要增加副本数以提高分区组数据可靠性。

操作方法 :

```
> var dataRG = db.getRG(<groupname>);
> var node1 = dataRG.createNode(<host>, <service>, <dbpath>, [config]);
> node1.start();
```

第一条命令用于获取数据分区组 , 参数 groupname 为数据分区组组名 ;

第二条命令用于创建一个新的数据节点，参数与[新增编目分区组中的创建节点参数相同](#)；

第三条命令用于启动新增的数据节点。

新增协调节点

当集群规模扩大时，协调节点也需要随着规模的增加而进行增加。建议匹配时，一台物理节点，配置一个协调节点。

1. 创建协调节点配置目录；

```
mkdir -p /opt/sequoiadb/conf/local/11810
```

其中11810为协调节点的服务端口，可根据需要配置

2. 拷贝协调节点样例配置文件；

```
cp /opt/sequoiadb/conf/samples/sdb.conf.coord /opt/sequoiadb/conf/local/11810/sdb.conf
```

3. 修改配置文件；

```
vi /opt/sequoiadb/conf/local/11810/sdb.conf
```

修改内容

```
# database path dbpath=/opt/sequoiadb/database/coord
```

该参数为数据库放置路径，可根据需要修改，请确保路径已经存在（不存在请手工创建）

将如下行：

```
# catalog addr (hostname1: servicename1, hostname2: servicename2, ...)  
# catalogaddr=
```

修改

```
# catalog addr (hostname1: servicename1, hostname2: servicename2, ...)  
catalogaddr=sdbserver1:11803, sdbserver2:11803, sdbserver3:11803
```

该参数为Catalog服务地址和端口

4. 按 :wq，保存退出 vi；

5. 创建数据文件存放路径；

```
mkdir -p /opt/sequoiadb/database/coord
```

路径为上一步骤配置的路径

6. 启动协调节点进程。

```
/opt/sequoiadb/bin/sdbstart -c /opt/sequoiadb/conf/local/11810/
```

案例

案例一：新增主机，在当前数据分区中新增一个数据副本节点

- 当前环境：

配置项	配置情况
主机	包含一台主机： 主机一：OS 为 SUSE 11 SP2 64位，主机名为 vmsvr1-suse-x64，IP 为 192.168.1.10
编目分区组	分区组包含一个编目节点，服务端口为 11800
数据分区组	一个数据分区组，组名为 datagroup1，其中包含一个数据节点： 数据节点一：服务主机名为 vmsvr1-suse-x64，端口为 11820
协调节点	一个协调节点，端口为 11810

- 调整后的预期结果

配置项	配置情况
主机	包含两台主机： 主机一：OS 为 SUSE 11 SP2 64位，主机名为 vmsvr1-suse-x64，IP 为192.168.1.10 主机二：OS 为 SUSE 11 SP2 64位，主机名为 vmsvr2-suse-x64，IP 为192.168.1.11
编目分区组	分区组包含一个编目节点，服务端口为11800
数据分区组	一个数据分区组，组名为 datagroup1，其中包含两个数据节点： 数据节点一：服务主机名为 vmsvr1-suse-x64，端口为11820 数据节点二：服务主机名为 vmsvr2-suse-x64，端口为11820
协调节点	一个协调节点，端口为11810

操作步骤

- 步骤一：配置新增的主机

1. 安装操作系统 SUSE 11 SP2 64位（保持与原系统一致）；
2. 使用 root 用户登录系统；
3. 配置 IP 地址为 192.168.1.11；
4. 执行如下命令，配置主机名为 vmsvr2-suse-x64：

```
hostname vmsvr2-suse-x64
```

修改 /etc/hosts 文件，新增两行：

```
192.168.1.10 vmsvr1-suse-x64
192.168.1.11 vmsvr2-suse-x64
```

按 :wq 保存退出；

5. 执行如下命令，检查主机配置是否生效：

```
ping vmsvr1-suse-x64 //检查是否 ping 通;
ping vmsvr2-suse-x64 //检查是否 ping 通;
```

6. 配置系统内核参数：

打开 /etc/profile 文件，在该文件末尾增加如下行：

```
ulimit -Sf unlimited //文件大小限制
ulimit -St unlimited //CPU时间限制
ulimit -Sv unlimited //虚拟内存限制
ulimit -Sn 64000 //文件个数限制
ulimit -Sm unlimited //内存大小限制
ulimit -Su 32000 //线程个数限制
```

按 :wq 保存退出；

执行 source /etc/profile，使得配置生效；

7. 执行如下命令，检查系统内核参数配置是否生效：

```
ulimit -a; //查看配置是否正确;
```

- 步骤二：配置原集群的主机

修改原主机 vmsvr1-suse-x64 的 /etc/hosts 配置文件，在配置文件中新增一行：

```
192.168.1.11 vmsvr2-suse-x64
```

按 :wq 保存退出；

- 步骤三：在新增主机上安装 SequoiaDB 软件

参考 [SequoiaDB 服务器安装](#)一节进行软件安装。

- 步骤四：在当前分区中新增数据节点

1. 在 vmsvr2-suse-x64 上执行如下命令，启动 SequoiaDB Shell 命令行：

```
/opt/sequoiadb/bin/sdb
```

2. 在 SequoiaDB Shell 命令行中，执行如下命令在当前数据分区组新增一副本节点：

```
> var db = new Sdb("localhost", 11810)
> var datarg = db.getRG("datagroup1")
> node=rg.createNode("vmsvr2-suse-x64", 11820, "/opt/sequoiadb /database/data/11820")
> node.start()
```

3. 在 SequoiaDB Shell 命令行中，执行如下命令检查分区组配置情况，可以看到数据组新增了一个数据节点：

```
> db.listReplicaGroups();
...
{
  "Group": [
    {
      "dbpath": "/opt/sequoiadb/database/data/11820",
      "HostName": "vmsvr1-suse-x64",
      "Service": [
        {
          "Type": 0,
          {
            "Type": 1,
            "Name": "11821"
          },
          {
            "Type": 2,
            "Name": "11822"
          },
          {
            "Type": 3,
            "Name": "11823"
          }
        ],
        "NodeID": 1000
      },
      {
        "dbpath": "/opt/sequoiadb/database/data/11820",
        "HostName": "vmsvr2-suse-x64",
        "Service": [
          {
            "Type": 0,
            "Name": "11820"
          },
          {
            "Type": 1,
            "Name": "11821"
          },
          {
            "Type": 2,
            "Name": "11822"
          },
          {
            "Type": 3,
            "Name": "11823"
          }
        ],
        "NodeID": 1001
      }
    }
  ]
}
```

```

],
"GroupID": 1000,
"GroupName": "datagroup1",
"PrimaryNode": 1000,
"Role": 0,
"Status": 1,
"Version": 3,
"_id": {
    "$oid": "51d673c1fde96799fbac6aad"
}
}

```

运维

目录：

[集群启停](#)
[备份恢复](#)
[故障恢复](#)
[监控](#)

集群启停

本节介绍集群的启动和停止。

[集群启动](#)
[集群停止](#)

[集群启动](#)

[操作系统重启动](#)

操作系统启动后会自动启动服务 sdbcm (sequoiadb cluster manager)。该服务启动后会自动启动该物理机中所有注册在 /opt/sequoiadb/conf/local 目录下的节点。使用命令 ps -elf | grep sequoiadb 能看到当前正在启动的节点与启动完毕的节点。启动完毕的进程名为：sequoiadb (服务名) 正在启动的进程名一般为：

/opt/sequoiadb/bin/sequoiadb -c /opt/sequoiadb/conf/local/ (服务名)

[手工启动特定节点](#)

当集群中某个节点失效后，用户可以在 sdb 命令行使用如下步骤启动节点。假设 SequoiaDB 的安装路径为 /opt/sequoiadb

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到分区组

```
> dataRG = db.getRG ( "<datagroup1>" ) ;
```

3. 得到数据节点

```
> dataNode = dataRG.getNode ( "<hostname1>", "<servicename1>" ) ;
```

4. 启动节点

```
> dataNode.start() ;
```

手工启动数据组

当集群中某个数据组被停止后，用户可以在 sdb 命令行使用如下步骤启动数据组。该操作会启动数据组中全部数据节点。

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到分区组

```
> dataRG = db.getRG ( "<datagroup1>" ) ;
```

3. 启动数据组

```
> dataRG.start();
```

集群停止

手工停止特定节点

用户可以在 sdb 命令行使用如下步骤停止数据节点。

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到分区组

```
> dataRG = db.getRG ( "<datagroup1>" ) ;
```

3. 得到数据节点

```
> dataNode = dataRG.getNode ( "<hostname1>", "<servicename1>" ) ;
```

4. 停止节点

```
> dataNode.stop();
```

手工停止数据组

用户可以在 sdb 命令行使用如下步骤停止数据组。该操作会停止数据组中全部数据节点。

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到分区组

```
> dataRG = db.getRG ( "<datagroup1>" ) ;
```

3. 停止数据组

```
> dataRG.stop();
```

使用kill命令停止数据节点

用户可以使用 kill -15 <pid> 正常停止数据节点。以该方式停止的数据节点被认为正常停止。用户使用 kill -9 <pid> 强行停止数据节点。以该方式停止的数据节点被认为非正常停止。如果该节点使用了正常的启动流程，则会被 sdbcm 进程尝试重新启动。启动后会与当前数据组中其它节点进行同步。

备份恢复

本节介绍集群的备份与恢复

数据备份

查看备份信息

数据恢复

[SequoiaDB 备份恢复操作指导书](#)

数据备份

当前版本中，数据备份支持离线备份，即数据备份期间需要中断插入、更新、删除等变更操作，只支持查询操作。当前备份支持两种方式：全量备份和增量备份

- 全量备份：备份整个数据库的配置、数据和日志；
- 增量备份：在上一个全量备份或增量备份的基础上备份新增的日志和配置；

离线备份参数说明

参数	说明
Name	备份名称，缺省则以当前时间格式命名，如“2013-11-13-15:00:00”。
Description	备份用户描述信息。
Path	本次备份的指定路径，缺省为配置参数“bkuppath”中指定的路径。
EnsureInc	备份方式，true 表示增量备份，false 表示全量备份，缺省为 false。
OverWrite	对于同名备份是否覆盖，true 表示覆盖，false 表示不覆盖，如果同名则报错；缺省为 true。
GroupName	对指定组进行备份，缺省为对全系统备份，当需要对多个组进行备份可以指定为数组类型，如：[“datagroup1”, “datagroup2”]。

备份整个数据库

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810);
```

2. 执行备份命令

```
> db.backupOffline({Name: "backupName", Description: "backup for all"})
```

备份指定组的数据库

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810);
```

2. 执行备份命令

```
> db.backupOffline({Name: "backupName", Description: "backup group1", GroupName: "datagroup1"})
```

备份指定节点的数据库

1. 连接到指定节点

```
$ /opt/sequoiadb/bin/sdb
> var dbdata = new Sdb("hostname1", "servicename1");
```

2. 执行备份命令

```
> dbdata.backupOffline({Name: "backupName", Description: "backup data node"}))
```

 注: catalog 编目组的名称固定为 SYSCatalogGroup

查看备份信息

备份信息查看可以通过客户端和手工查看。

查看备份信息参数说明

参数	说明
Name	备份名称，缺省则查看目录下所有备份信息。
Path	查看备份的指定路径，缺省为配置参数“bkuppath”中指定的路径。
GroupName	查看指定组的备份信息，缺省为查看全系统备份信息，当需要查看多个组的备份信息可以指定为数组类型，如：[“datagroup1”,“datagroup2”]。

查看全系统备份信息

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810);
```

2. 执行查看备份信息命令

```
> db.listBackup()
{
  "Name": "test_bk",
  "NodeName": "vmsvr2-suse-x64-1:11800",
  "GroupName": "SYSCatalogGroup",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 18744,
  "StartTime": "2013-11-13-16:06:31",
  "HasError": false
}
{
  "Name": "test_bk",
  "NodeName": "vmsvr2-suse-x64-1:11820",
  "GroupName": "db1",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 920424,
  "StartTime": "2013-11-13-16:06:31",
  "HasError": false
}
```

查看指定名称的备份信息

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810);
```

2. 执行查看备份信息命令

```
> db.listBackup({Name: "backup1"})
{
  "Name": "backup1",
  "NodeName": "vmsvr2-suse-x64-1:11820",
  "GroupName": "group1",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 108744,
  "StartTime": "2013-11-13-16:06:31",
  "HasError": false
}
```

手工查看备份信息

手工查看备份信息直接通过终端登入指定机器，并进入到相应的备份目录中，执行“ls -l”

```
use@vmsvr2-suse-x64-1: /opt/sequoiadb/database/11820/bakfile> ls -l
```

```
total 37328
-rw-r----- 1 sdbadmin sdbadmin 38157784 Nov 13 16:06 test_bk.1
-rw-r----- 1 sdbadmin sdbadmin      65536 Nov 13 16:06 test_bk.bak
```

数据恢复

使用备份的数据恢复某个分区组。执行数据恢复必须确保相应组已停止运行，数据恢复首先会清空原节点的所有数据和日志，然后从备份的数据中恢复配置、数据和日志。

数据恢复工具参数说明

参数	缩写	说明
--bkpath	-p	备份源数据所在路径。
--increaseid	-i	需要恢复到第几次增量备份，缺省恢复到最后一次。
--bkname	-n	需要恢复的备份名。
--action	-a	恢复行为，“restore”表示恢复，“list”表示查看备份信息，缺省为“restore”。
--isSelf		是否为恢复本节点数据，缺省为“true”；当取值为“false”时，根据如下参数将数据恢复至指定路径：
--dbpath		必须配置，数据文件目录。
--confpath		必须配置，配置文件路径。
--svcname		必须配置，本地服务名或端口。
--indexpath		索引文件目录。
--logpath		日志文件目录。
--diagpath		诊断日志文件目录。
--bkuppath		备份文件目录。
--replname		复制通讯服务名或端口。
--shardname		分区通讯服务名或端口。
--catalogname		编目通讯服务名或端口。
--httpname		REST 服务名或端口。

恢复数据

 注：如果一个分区组包含多个数据节点，必须停止该组中每个数据节点并进行恢复。如果将备份的数据恢复至非备份数据节点，须使用 `--isSelf false` 配置参数，同时设置相关的配置参数。

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到分区组

```
> dataRG = db.getRG ( "data" ) ;
```

3. 停止分区组

```
> dataRG.stop()
```

4. 通过终端登入相应分区组的数据节点，执行数据恢复。

```
sdbadmin@vmsvr2-suse-x64-1: /opt/sequoiadb> bin/sdbrestore -p database/11820/bakfile -n
test_bk
Begin to clean dps logs...
Begin to clean dms storages...
Begin to init dps logs...
Begin to restore...
Begin to restore data file: 11820/bakfile/test_bk.1 ...
```

```

Begin to restore su: test.1.data ...
Begin to restore su: test.1.idx ...
Begin to restore dps logs...
*****
Restore succeed!
*****

```

5. 到数据节点目录检查文件是否恢复。

```

sdbadmin @vmsvr2-suse-x64-1: / opt/sequoiadb /database/11820> ls -l
total 299156
drwxr-xr-x 2 sdbadmin sdbadmin      4096 Nov 13 16:06 bakfile
drwxr-xr-x 2 sdbadmin sdbadmin      4096 Nov 13 15:48 diaglog
drwxr-xr-x 2 sdbadmin sdbadmin      4096 Nov 13 17:39 replicalog
-rw-r---- 1 sdbadmin sdbadmin 155254784 Nov 13 17:39 test.1.data
-rw-r---- 1 sdbadmin sdbadmin 151060480 Nov 13 17:39 test.1.idx

```

6. 删除该分区组中其它数据节点的所有数据（或者将该节点的所有 .data 和 .idx 文件拷贝至其它数据节点的数据目录和索引目录下，以及将该节点 replicalog 所有日志拷贝至其它数据节点的日志目录下，或者将备份文件拷贝至其它数据节点，并通过 restored 工具恢复）；重新启动系统。

[SequoiaDB 备份恢复操作指导书](#)

[SequoiaDB 备份及恢复详细样例](#)

备份操作

SequoiaDB 的备份操作是通过在 sdb 客户端上执行相应的方法来完成的。备份操作相关方法函数包含：[db.backupOffline\(\)](#) , [db.listBackup\(\)](#) , [db.removeBackup\(\)](#)。数据备份支持离线备份，在数据备份期间需要中断插入，更新，删除等变更操作，只支持查询操作。

备份有两种方式：全量备份和增量备份。

- 全量备份：备份整个数据库的配置，数据和日志；
- 增量备份：在上一个全量备份或增量备份的基础上备份新增的日志和配置。

备份步骤：

1. 备份前确认数据库无插入，更新，删除等变更操作；
2. 进入 sdb 客户端；

```
# /opt/sequoiadb/bin/sdb
```

3. 连接其中 coord 节点

```
> var db = new Sdb('htest1', 11810)
```

4. 检查数据库是否有备份；

检查数据库集群内所含有的数据：

```

> db.list(4)
{
  "Name": "foo.bar"
}
Return 1 row(s).
Takes 0.2882s.
> db.foo.bar.count()
2880000
Takes 0.2586s.
> db.listBackup()
Return 0 row(s).
Takes 0.6483s.

```

5. 执行数据库备份；

数据库集群备份是按组为单位进行备份的，故在备份的时候需要指定组进行备份。如下操作对编目组进行备份，备份完成后并且检查是否备份成功，此备份文件保存在主机 htest1：/opt/sequoiadb/database/cata/30000/bakfile/ 下面。

```
> db.backupOffline({Name: "catalogBackup", GroupName: "SYSCatalogGroup"})
Takes 1.762381s.
> db.listBackup()
{
  "Name": "catalogBackup",
  "NodeName": "htest1:30000",
  "GroupName": "SYSCatalogGroup",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 9716,
  "StartTime": "2014-07-29-22:34:40",
  "HasError": false
}
Return 1 row(s).
Takes 0.7994s.
```

数据组也采用与编目组同样的方式进行备份，其备份文件保存在主机 htest3 上，目录名为：/opt/sequoiadb/database/data/41000/bakfile/

```
> db.backupOffline({Name: "datagroupG1Backup", GroupName: "g1"})
Takes 12.85741s.
> db.listBackup({Name: "datagroupG1Backup", GroupName: "g1"})
{
  "Name": "datagroupG1Backup",
  "NodeName": "htest3:41000",
  "GroupName": "g1",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 460800544,
  "StartTime": "2014-07-29-22:41:40",
  "HasError": false
}
```

注：如果想要一次将整个数据库备份完成，则可执行数据库备份操作且不提供任意参数。如此即可一次备份完成整个数据库集群，包含编目组与数据组。默认备份到相应组主节点的 bakfile 下 /opt/sequoiadb/database/../../dialog/bakfile

```
> db.backupOffline()
```

6. 查看备份；

```
> db.listBackup()
{
  "Name": "catalogBackup",
  "NodeName": "htest1:30000",
  "GroupName": "SYSCatalogGroup",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 9716,
  "StartTime": "2014-07-29-22:34:40",
  "HasError": false
}
{
  "Name": "datagroupG1Backup",
  "NodeName": "htest3:41000",
  "GroupName": "g1",
  "EnsureInc": false,
  "BeginLSNOffset": 0,
  "EndLSNOffset": 460800544,
  "StartTime": "2014-07-29-22:41:40",
  "HasError": false
}
```

```
Return2 row(s).
Takes 0.16428s.
```

如果不指定路径备份，则数据库默认备份到相应组的主节点数据库目录下的 bakfile 文件夹中。若备份在其它路径下，则查看的时候需要带路径参数。

数据恢复

使用数据库的备份文件来恢复某个分区组。执行数据恢复必须确保相应组已停止运行，数据恢复首先会清空原节点的所有数据和日志，然后从备份的数据中恢复配置，数据和日志。数据恢复工具位于：/opt/sequoiadb/bin/sdbrestore，执行 ./sdbrestore --help 可以查看相应的参数。

恢复流程：

1. 停止相关的组的数据库服务进程；

停止数据库服务最好通过 sdb 客户端来完成，这样不需要去集群节点所在的机器去停止数据库，如 g1。当停止的组中包含 catalog 编目组时，要最后再停止编目组。

```
sdbadmin@htest1:~$ /opt/sequoiadb/bin/sdb
> var db = new Sdb('localhost', 11810)
> datarg = db.getRG('g1')
> datarg.stop()
```

2. 将数据库备份文件拷贝到共享目录下；

由于 SequoiaDB 数据的恢复只能对一台机进行恢复，故将数据库备份文件拷贝到数据库集群机器所共享的目录下。如共享目录为 /mnt/xiaojun/BACKUP，则拷贝如下：

```
sdbadmin@htest1:/opt/sequoiadb$ cp -r /opt/sequoiadb/database/cata/30000/bakfile/ /mnt/
xiaojun/BACKUP/bakfile.30000.hertest1
sdbadmin@htest3:/opt/sequoiadb$ cp -r /opt/sequoiadb/database/data/41000/bakfile/ /mnt/
xiaojun/BACKUP/bakfile.41000.hertest3
```

拷贝到共享目录下的好处是在恢复操作时只用一个共享目录即可，而不用每一台要恢复的机器都拷贝一份备份文件到那台机器中去。

3. 数据恢复操作；

恢复工具 (sdbrestore) 位于 SequoiaDB 数据库中的 bin 目录下面（/opt/sequoiadb/bin）。通过 sdb shell 登入分区组的数据节点，执行数据恢复。执行恢复操作时，至少要对一个组中超过半数的节点进行恢复，最好是对组内所有的节点进行恢复操作。此数据库集群包含两个组：编目组“SYSCatalogGroup”和一个数据组“g1”，分别部署在三台主机上：htest1/htest2/htest3。恢复节点操作时需要进到节点所在的主机分别进行恢复。如恢复位于主机 htest1 上的编目节点则要在此主机内执行操作：

```
sdbadmin@htest1:/opt/sequoiadb/bin$ ./sdbrestore -p /mnt/xiaojun/BACKUP/
bakfile.30000.hertest1/ -n catalogBackup --dbpath \
> /opt/sequoiadb/database/cata/30000/ --confpath /opt/sequoiadb/conf/local/30000/ --svcname
30000
2014-07-29-23.09.19.382036           Level: EVENT
PID: 9556                           TID: 9556
Function: pmdRestoreThreadMain       Line: 491
File: SequoiaDB/engine/pmd/sdbrestore.cpp
Message:
Start sdbrestore [Ver: 1.8, Release: 13673, Build: 2014-07-22-00.11.23(Debug)]...
.
.
.
*****
Restore succeed!
*****
```

此时仅恢复了 htest1 的编目节点，采用同样的方式去将 htest2 和 htest3 的编目节点启动起来。

恢复数据组 g1 的数据节点，和恢复编目组节点的方法类似。三台机 htest1/htest2/htest3 中的数据节点均采用如下方法分别进行恢复：

```
sdbadmin@htest1:/opt/sequoiadb/bin$ ./sdbrestore -p /mnt/xiaojun/BACKUP/
bakfile.41000.htest3/ -n datagroupG1Backup --dbpath \
> /opt/sequoiadb/database/data/41000/ --confpath /opt/sequoiadb/conf/local/41000/ --svcname
41000
2014-07-29-23.18.08.966708           Level: EVENT
PID: 9590                           TID: 9590
Function: pmdRestoreThreadMain       Line: 491
File: SequoiaDB/engine/pmd/sdbrestore.cpp
Message:
Start sdbrestore [Ver: 1.8, Release: 13673, Build: 2014-07-22-00.11.23(Debug)]...
.
.
.
*****
Restore succeed!
*****
```

4. 检查数据库集群的恢复是否正确；

通过 sdb 客户端对恢复好的数据库进行简单的数据操作，如增删改查等。

```
sdbadmin@htest1:~$ /opt/sequoiadb/bin/sdb
> var db = new Sdb('htest1', 11810)
Takes 0.8073s.
> db.list(4)
{
  "Name": "foo.bar"
}
Return 1 row(s).
Takes 0.47547s.
> db.foo.bar.count()
2880000
Takes 0.164788s.
> db.foo.bar.find().limit(1)
{
  "_id": 4000000,
  "id": 4000000,
  "ID": "http://www.sequoiadb.com",
  "change": "ccc",
  "info": "email:contact@sequoiadb.com"
}
Return 1 row(s).
Takes 0.28824s.
> db.foo.bar.insert({testing: "testingBackup"})
Takes 0.296942s.
> db.foo.bar.find({testing: "testingBackup"})
{
  "_id": {
    "$oid": "53d7c4f9c6e6fab31f000000"
  },
  "testing": "testingBackup"
}
Return 1 row(s).
Takes 4.311792s.
```

通过此操作可基本确定数据库恢复成功。

故障恢复

本节介绍个类型节点的故障恢复。

协调节点

[数据节点](#)[编目节点](#)[协调节点](#)

由于协调节点不存在用户数据，因此发生故障后可以直接重新启动，不参与任何额外的故障恢复步骤。

[数据节点](#)

数据节点发生故障后，重新启动会自动检测数据库目录下 .SEQUOIADB_STARTUP 隐藏文件。

如果该文件存在则说明上次的执行意外终止（例如 kill -9）。对于意外终止的节点，系统会将该数据节点置入崩溃恢复状态。

在崩溃恢复的过程中，数据节点会与该组中的一个正常节点进行全量同步。在这种情况下，被恢复的节点中所有数据作废，同步到的新数据作为基准。假设该节点没有被意外终止（例如 kill -15），则进入增量同步状态。在这种情况下，如果当前其它数据节点中包含的最老日志已经比被恢复节点新，则进入全量同步状态，否则只同步增量日志。

如果该数据组中所有节点都被意外终止，则需要以独立模式启动一个节点进行本地恢复。在该模式中，数据会被导出并再次导入，以过滤掉所有可能出现的数据损坏。当其中一个节点被本地恢复后，需要将其数据目录拷贝入其它所有数据节点。

[编目节点](#)

编目节点故障恢复策略与数据节点相同。[点击这里](#)

[监控](#)

本节介绍节点和集群的监控信息

[监控节点状态](#)[监控集群](#)[监控节点状态](#)

用户可以使用 snapshot 监控每个节点的状态。

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到分区组

```
> datarg = db.getRG ( "<datagroup1>" ) ;
```

3. 得到数据节点

```
> datanode = datarg.getNode ( "<hostname1>", "<servicename1>" ) ;
```

4. 得到该节点的快照

```
> datanode.connect().snapshot(SDB_SNAP_DATABASE)
```

快照类型分为：

[SDB_SNAP_CONTEXTS](#)

[SDB_SNAP_CONTEXTS_CURRENT](#)

[SDB_SNAP_SESSIONS](#)

[SDB_SNAP_SESSIONS_CURRENT](#)

[SDB_SNAP_COLLECTIONS](#)

[SDB_SNAP_COLLECTIONSPACES](#)

[SDB_SNAP_DATABASE](#)

[SDB_SNAP_SYSTEM](#)

[SDB_SNAP_CATALOG](#)

用户可以使用 Shell 脚本监控，例如：

```
[sequoiadb@vmsvr1-rhel-x64 sequoiadb]$ cat monitor_insert.sh
#!/bin/bash
./sequoiadb/bin/sdb "db=new Sdb('localhost', 11810)" > /dev/null
./sequoiadb/bin/sdb "db.getRG('foo').getNode('vmsvr1-rhel-
x64', 11820).connect().snapshot(SDB_SNAP_DATABASE)" | grep TotalInsert
./sequoiadb/bin/sdb "quit"
[sequoiadb@vmsvr1-rhel-x64 sequoiadb]$ ./monitor_insert.sh
"TotalInsert": 0,
```

监控集群

1. 连接到协调节点

```
$ /opt/sequoiadb/bin/sdb
> var db = new Sdb("localhost", 11810) ;
```

2. 得到集群状态

```
> db.listReplicaGroups()
{
  "Group": [
    {
      "dbpath": "/home/sequoiadb/sequoiadb/cata",
      "HostName": "vmsvr1-rhel-x64",
      "Service": [
        {
          "Type": 0,
          "Name": "11800"
        },
      ]
    }
  ]
}
```

系统安全

概要说明

可以通过安全功能来指定登陆系统的权限。如果没有合法的用户名及密码则无法访问数据库。

详细说明

1. 系统启动时默关闭安全功能。只有在至少创建一个用户后，安全功能才自动激活。
2. 创建用户需要用户指定用户名及密码。用户名全系统唯一。
3. 删除用户需要用户指定用户名及密码。
4. 安全功能激活后，访问数据库需要指定用户名及密码。在一次会话中进行一次登录验证。开启新的会话需要重新进行验证。
5. 用户之间属于平级关系。没有超级用户的概念。用户可以创建删除任意用户。
6. 当删除所有用户后，安全功能自动关闭。
7. 所有密码均以密文形式传输，存储。

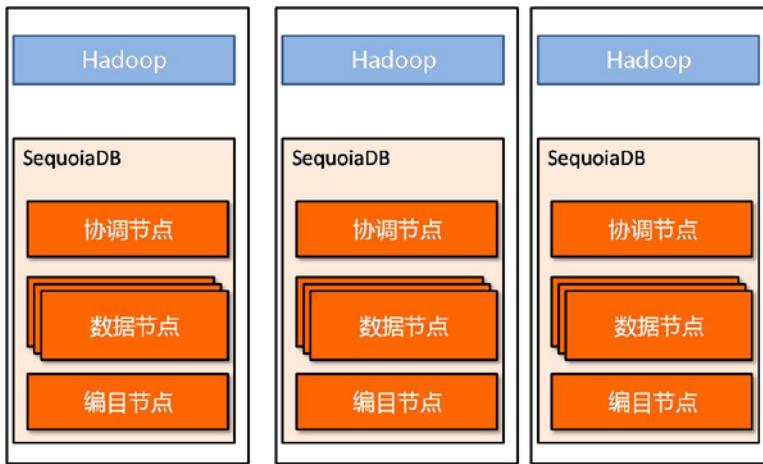
Hadoop 集成

与 Hadoop 集成相关内容。

SequoiaDB 与 Hadoop 部署

SequoiaDB 与 Hadoop 在物理上部署方案简易如下图所示，部署建议如下：

- SequoiaDB 与 Hadoop 部署在相同的物理设备上，以减少 Hadoop 与 SequoiaDB 之间的网络数据传输
- 每个物理设备上都部署一个协调节点和多个数据节点，编目节点可选在任意三台物理设备各部署一个编目节点



与 MapReduce 集成

搭建 Hadoop 环境

我们支持 hadoop 1.x 和 hadoop 2.x。先安装配置好 Hadoop

配置连接环境

与 MapReduce 对接，需要准备 hadoop-connector.jar 和 sequoiadb.jar，这两个 jar 可以在 SequoiaDB 安装目录下面的 hadoop 目录中找到。

因为不同版本的 Hadoop 的 classpath 不一样，所以先查看 hadoop 的 classpath，输入 hadoop classpath，在 classpath 中选择一个目录，把 hadoop-connector.jar 和 sequoiadb.jar 放在目录里面，重启 hadoop 集群。

编写 MapReduce

hadoop-connector.jar 中一些重要的类：

SequoiadbInputFormat：读取SequoiaDB的数据

SequoiadbOutputFormat：向SequoiaDB中写入数据

BSONWritable：BSONObject 的包装类，实现了 WritableComparable 接口。用于序列化 BSONObject 对象。

SequoiaDB 和 MapReduce 的配置：

sequoiadb-hadoop.xml 是配置文件，放在你编写的 MapReduce 工程的源码根目录下面。

sequoiadb.input.url : 指定作为输入的 SequoiaDB 的 URL 路径，格式为：
hostname1:port1,hostname2:port2,

sequoiadb.in.collectionspace : 指定作为输入的集合空间。

sequoiadb.in.collection : 指定作为输入的集合。

sequoiadb.output.url : 指定作为输出的 SequoiaDB 的 URL 路径。

sequoiadb.out.collectionspace : 指定作为输出的集合空间。

sequoiadb.out.collection : 指定作为输出的集合。

sequoiadb.out.bulknum : 指定每次向 SequoiaDB 写入的记录条数，对写入性能进行优化。

示例

- 1. 下面是读取 HDFS 文件，处理后写入到 SequoiaDB 中去：

```
public class HdfsSequoiadbMR {
    static class MobileMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
        private static final IntWritable ONE=new IntWritable(1);
        @Override
        protected void map(LongWritable key, Text value, Context context)
            throws IOException, InterruptedException {
            String valueStr=value.toString();

            String mobile_prefix=valueStr.split(",") [3].substring(0, 3);
            context.write(new Text(mobile_prefix), ONE);
        }
    }

    static class MobileReducer extends Reducer<Text, IntWritable, NullWritable,
    BSONWritable> {

        @Override
        protected void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
            Iterator<IntWritable> iterator=values.iterator();
            long sum=0;
            while(iterator.hasNext()) {
                sum+=iterator.next().get();
            }
            BSONObject bson=new BasicBSONObject();
            bson.put("prefix", key.toString());
            bson.put("count", sum);
            context.write(null, new BSONWritable(bson));
        }
    }

    public static void main(String[] args) throws IOException, InterruptedException,
    ClassNotFoundException {
        if(args.length<1) {
            System.out.print("please set input path ");
            System.exit(1);
        }
        Configuration conf=new Configuration();
        conf.addResource("sequoiadb-hadoop.xml"); //加载配置文件
        Job job=Job.getInstance(conf);
        job.setJarByClass(HdfsSequoiadbMR.class);
        job.setJobName("HdfsSequoiadbMR");
    }
}
```

```

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(SequoiaadbOutputFormat.class); //reduce 输出写入到
SequoiaDB 中
        TextInputFormat.setInputPaths(job, new Path(args[0]));

        job.setMapperClass(MobileMapper.class);
        job.setReducerClass(MobileReducer.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);

        job.setOutputKeyClass(NullWritable.class);
        job.setOutputValueClass(BSONWritable.class);

        job.waitForCompletion(true);
    }
}

```

- 2. 读取 SequoiaDB 中数据处理后写入到 HDFS 中。

```

public class SequoiadbHdfsMR {
    /**
     *
     * @author gaoshengjie
     * read the data, count people in a province
     */
    static class ProvinceMapper extends Mapper<Object, BSONObject, IntWritable, IntWritable>{
        private static final IntWritable ONE=new IntWritable(1);
        @Override
        protected void map(Object key, BSONObject value, Context context)
            throws IOException, InterruptedException {
            int province=(Integer) value.get("province_code");
            context.write(new IntWritable(province), ONE);
        }
    }

    static class ProvinceReducer extends
Reducer<IntWritable, IntWritable, IntWritable, LongWritable>{

        @Override
        protected void reduce(IntWritable key, Iterable<IntWritable> values,
Context context)
            throws IOException, InterruptedException {
            Iterator<IntWritable> iterator=values.iterator();
            long sum=0;
            while(iterator.hasNext()) {
                sum+=iterator.next().get();
            }
            context.write(key, new LongWritable(sum));
        }
    }

    public static void main(String[] args) throws IOException, InterruptedException,
ClassNotFoundException {
        if(args.length<1) {
            System.out.print("please set output path ");
            System.exit(1);
        }
        Configuration conf=new Configuration();
        conf.addResource("sequoiadb-hadoop.xml");
        Job job=Job.getInstance(conf);
        job.setJarByClass(SequoiadbHdfsMR.class);
    }
}

```

```

        job.setJobName("SequoiaadbHdfsMR");
        job.setInputFormatClass(SequoiaadbInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileOutputFormat.setOutputPath(job, new Path(args[0] + "/result"));

        job.setMapperClass(ProvinceMapper.class);
        job.setReducerClass(ProvinceReducer.class);

        job.setMapOutputKeyClass(IntWritable.class);
        job.setMapOutputValueClass(IntWritable.class);

        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(LongWritable.class);

        job.waitForCompletion(true);
    }
}

```

配置信息：

```

<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property>
        <name>sequoiadb.input.url</name>
        <value>localhost:11810</value>
    </property>
    <property>
        <name>sequoiadb.output.url</name>
        <value>localhost:11810</value>
    </property>
    <property>
        <name>sequoiadb.in.collectionspace</name>
        <value>default</value>
    </property>
    <property>
        <name>sequoiadb.in.collect</name>
        <value>student</value>
    </property>
    <property>
        <name>sequoiadb.out.collectionspace</name>
        <value>default</value>
    </property>
    <property>
        <name>sequoiadb.out.collect</name>
        <value>result</value>
    </property>
    <property>
        <name>sequoiadb.out.bulknum</name>
        <value>10</value>
    </property>
</configuration>

```

与 Hive 集成

- SequoiaDB 支持的 Hive 版本列表
- 配置方法

SequoiaDB 支持的 Hive 版本列表

- Apache Hive 0.12.0

- Apache Hive 0.11.0
- Apache Hive 0.10.0
- CDH-5.0.0-beta-2 Hive 0.12.0



注:

hive-sequoiadb-apache.jar 为支持 Apache 版 Hive 的 SequoiaDB-Hive-Connector

hive-sequoiadb-cdh-5.0.0-beta-2.jar 为支持 CDH5.0.0-beta-2 版本下 Hive-0.12 的 SequoiaDB-Hive-Connector

配置方法

1. 安装和配置好 Hadoop/Hive 环境，启动 Hadoop 环境；
2. 拷贝 SequoiaDB 安装目录下（默认在 /opt/sequoiadb）的 hadoop/hive-sequoiadb-{version}.jar 和 java/sequoiadb.jar 两个文件拷贝到 hive/lib 安装目录下；
3. 修改 Hive 安装目录下的 bin/hive-site.xml 文件（如果不存在，可拷贝 \$HIVE_HOME/conf/hive-default.xml.template 为 hive-site.xml 文件），增加如下属性：

```
<property>
  <name>hive.aux.jars.path</name>
  <value>file: //<HIVE_home>/lib/hive-sequoiadb-{version}.jar, file: //<HIVE_HOME>/lib/
sequoiadb.jar</value>
  <description>SequoiaDB store handler jar file</description>
</property>

<property>
  <name>hive.auto.convert.join</name>
  <value>false</value>
</property>
```



注: 需要把这些 jar 包存放 HDFS 上，地址和 file 协议的地址一样。

使用方法

- [创建基于 SequoiaDB 的表](#)
- [从 HDFS 文件中导入数据到 SequoiaDB 表](#)
- [查询数据](#)

创建基于 SequoiaDB 的表

启动 Hive Shell 命令行窗口，执行如下命令创建数据表；

```
hive> create external table sdb_tab(id INT, name STRING, value DOUBLE) stored by
"com.sequoiadb.hive.SdbHiveStorageHandler" tblproperties("sdb.address" = "localhost:11810");
OK
Time taken: 0.386 seconds
```

其中：

sdb.address 用于指定 SequoiaDB 协调节点的 IP 和端口，如果有多个协调节点，可以写入多个，之间用逗号隔开。表的数据库对应 SequoiaDB 的集合空间，表对应集合空间中的集合。

从 HDFS 文件中导入数据到 SequoiaDB 表

从 HDFS 文件中导入数据到 SequoiaDB 表

```
hive> insert overwrite table sdb_tab select * from hdfs_tab;
Total MapReduce jobs = 1
```

```

Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_201310172156_0010, Tracking URL = http://b1465-5:50030/jobdetails.jsp?
jobid=job_201310172156_0010
Kill Command = /opt/hadoop-hive/hadoop-1.2.1/libexec/../bin/hadoop job -kill
job_201310172156_0010
Hadoop job information for Stage-0: number of mappers: 1; number of reducers: 0
2013-10-18 04:44:47,733 Stage-0 map = 0%, reduce = 0%
2013-10-18 04:44:49,763 Stage-0 map = 100%, reduce = 0%, Cumulative CPU 1.85 sec
2013-10-18 04:44:50,777 Stage-0 map = 100%, reduce = 0%, Cumulative CPU 1.85 sec
2013-10-18 04:44:51,795 Stage-0 map = 100%, reduce = 100%, Cumulative CPU 1.85 sec
MapReduce Total cumulative CPU time: 1 seconds 850 msec
Ended Job = job_201310172156_0010
10 Rows loaded to sdb_tab
MapReduce Jobs Launched:
Job 0: Map: 1 Cumulative CPU: 1.85 sec HDFS Read: 2301 HDFS Write: 0 SUCCESS
Total MapReduce CPU Time Spent: 1 seconds 850 msec
OK
Time taken: 12.201 seconds

```



注:

在导入数据到 SequoiaDB 表之前 , 请确保已经创建基于 HDFS 文件的 hdfs_tab 数据表 , 并 load 了数据。

查询数据

查询数据

```

hive> select * from new_tab;
OK
0    false    0.0    ALGERIA
1    true     1.0    ARGENTINA
2    true     1.0    BRAZIL
3    true     1.0    CANADA
4    true     4.0    EGYPT
5    false    0.0    ETHIOPIA
6    true     3.0    FRANCE
7    true     3.0    GERMANY
8    true     2.0    INDIA
9    true     2.0    INDONESIA
Time taken: 0.306 seconds, Fetched: 10 row(s)

```

与 Pig 集成

暂无文档。

开发指南

SequoiaDB 相关开发信息

SequoiaDB shell

目录：

- [SequoiaDB shell入门](#)
- [使用 SequoiaDB shell 的窍门](#)

SequoiaDB shell 入门

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

运行 shell

运行 sequoiadb (./sequoiadb) 启动 shell：

```
$ ./sdb
Welcome to SequoiaDB shell!
help() for help, Ctrl+c or quit to exit
>
```

shell 在启动时需要手动连接到 SequoiaDB 服务器，所以要确保在使用 shell 之前启动 sequoiadb。

shell 是一个功能完备的 JavaScript 解析器，可以运行任何 JavaScript 程序。如：

```
> y=200
200
> y/20
10
>
```

还可以充分利用 JavaScript 的标准库。

```
> new Date("2013/04/17");
Wed Apr 17 2013 00:00:00 GMT+0800 (CST)
> "hello,world".replace("world", "SequoiaDB")
hello,SequoiaDB
>
```

也可以定义和调用 JavaScript 函数：

```
> function sdb(n) {
...   if (n<=1) return 1;
...   else return n*sdb(n-1);
...
> sdb(4);
24
>
```

 注：可以使用多行命令。shell 会检测输入的 JavaScript 语句是否完整，如没有写完还可以接着写下一行。

SequoiaDB 客户端

启动 shell 可以运行任意 JavaScript 程序，但是 shell 的真正威力在于它是一个独立的 SequoiaDB 客户端。开启的时候，可以使用 db = new Sdb("localhost",11810) 命令连接到本地 SequoiaDB 服务器的数据库，并将这个数据库连接赋值给全局变量 db。这个变量是通过 shell 访问 SequoiaDB 的主要入口点。

shell 还有一些非 JavaScript 语法的扩展，例如：

```
> db
localhost: 11810
> db.createCS("foo") // 创建集合空间
localhost: 11810. foo
```

可以通过 db 变量来访问其中的集合空间，如 db.foo 返回当前数据库的 foo 集合空间。再通过集合空间来访问其中的集合，如 db.foo.bar，返回返回当前数据库 foo 集合空间的集合 bar。既然可以在 shell 中访问集合，那么基本上就可以执行几乎所有的数据库操作了。

使用 shell 的窍门

SequoiaDB shell 本身内置了帮助文档，通过 help() 命令可以查看使用介绍。另外，参考手册 SequoiaDB JavaScript 方法一节中，有各方法的详细使用介绍。

- Help

查看使用介绍：

> help()	connect to database use default host 'localhost'
var db = new Sdb()	and default port 11810
var db = new Sdb('localhost', 11810)	connect to database use specified host and port
var db = new Sdb('ubuntu', 11810, '', '')	connect to database with username and password
help(<method>)	help on specified method, e.g. help('createCS')
db.help()	help on db methods
db.cs.help()	help on collection space cs
db.cs.cl	access collection cl on collection space cs
db.cs.cl.help()	help on collection cl
db.cs.cl.find()	list all records
db.cs.cl.find({a: 1})	list records where a=1
db.cs.cl.find().help()	help on find methods
db.cs.cl.count().help()	help on count methods
print(x), println(x)	print out x
traceFmt(<type>, <in>, <out>)	format trace input(in) to output(out) by type
getErr(ret)	print error description for return code
clear	clear the terminal screen
history -c	clear the history
quit	exit
Takes 0.2993s.	

注:



SequoiaDB shell 主要包括 database(db) , collection space(cs) , collection(cl) , cursor(cur) , replica group(rg) , node(nd) , domain(dm) 这7大级别的操作。用户需要理解各级别之间的关系。各级别都有使用帮助命令如下所示：

- Database Help

database 级别主要包括用管理户组，集合空间，副本组，域，快照，存储过程，备份，事务，sql，及错误跟踪等操作。

假设已经连接上数据库，并取得 database 的 javascript 对象 db。

查看 database 所有方法：

```
db.help()
```

查看 database 具体方法：

```
db.help("method")
```

- CollectionSpace Help

collection space 级别主要包括对集合管理的操作。

假设存在名字为“foo”的集合空间。

查看 collection space 所有方法：

```
db.foo.help()
```

查看 collection space 具体方法：

```
db.foo.help("method")
```

- Collection Help

collection 级别主要包括 CRUD，索引管理，数据切分，垂直分区表管理等操作。

假设在集合空间“foo”中存在名字为“bar”的集合。

查看 collection 所有方法：

```
db.foo.bar.help()
```

查看 collection 具体方法：

```
db.foo.bar.help("method")
```

- Cursor Help

cursor 级别主要包括对返回记录（数据）的操作。

在 shell 命令中，与 sequoiadb 引擎交互时，若有记录（数据）返回，都是以游标（cursor）的方式呈现。例如，当使用 db.foo.bar.[find\(\)](#) 方法执行数据库查询操作，将返回一个游标对象，所有查询结果将放在这个游标中。通常的使用方法如下：

```
db.foo.bar.find()
```

或者

```
var cur = db.foo.bar.find()
```

前者直接将所有结果显示在屏幕上，后者将结果放到游标中。

查看 cursor 所有方法：

```
db.foo.bar.find().help()
```

或者

```
cur.help()
```

查看 cursor 具体方法：

```
db.foo.bar.find().help("method")
```

或者

```
cur.help("method")
```

类似于 find() 返回游标的方法，还有 list，snapshot 等等。

- Replica Group Help

replica group 级别主要包括对数据节点的管理的操作。

假设数据库中存在名字为“group1”的副本组，通过 var rg = db.getRG("group1") 获取一个关于副本组的 javascript 对象 rg。

查看 replica group 所有方法：

```
rg.help()
```

查看 replica group 具体方法：

```
rg.help("method")
```

- Node Help

node 级别主要包括对数据节点状态信息获取的操作。

假设在副本组“group1”中创建一个数据节点，`var rn = rg.createNode("ubuntu-dev1", 51000, "/opt/sequoiadb/database/data/51000")`，获取一个关于数据节点的 javascript 对象 rn。

查看 node 所有方法：

```
rn.help()
```

查看 node 具体方法：

```
rn.help("method")
```

- Domain Help

domain 级别主要包括对域更改及获取域信息的操作。

假设在数据库中创建一个名字为“domain1”的域，`var dm = db.createDomain("domain1", ["group1","group2"],{AutoSplit:true})`，获取一个关于域的 javascript 对象 dm。

查看 domain 所有方法：

```
dm.help()
```

查看 domain 具体方法：

```
dm.help("method")
```

 注：以 man page 方式显示帮助文档功能是随 SequoiaDB 1.8 版本发布的，若使用 1.8 版本以下的 sdb shell 客户端，将不具备上述的 help("method") 功能。另外，应该确保 /opt/sequoiadb/doc/manual 目录下有相关方法的 troff 文件，否则，无法显示相应的 man page 介绍。

查看 shell 提供的所有自动生成的 JavaScript 函数 API 文档，可访问 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources

SequoiaDB shell 中的基本操作

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

- [创建](#)
- [读取](#)
- [更新](#)
- [删除](#)

创建

在 SequoiaDB 中，create 操作是向集合中添加新的文档记录。我们可以使用 insert 方法向 SequoiaDB 中的集合中添加记录。

所有的插入操作在 SequoiaDB 中具有如下性质：

- 如果插入的文档记录没有 _id 字段，客户端将会为记录自动添加 _id 字段，并且填充一个唯一值。
- 如果指定 _id 字段，那个在集合中 _id 的值必须唯一；否则出现操作异常。
- 最大的 BSON 文档长度为 16MB。
- 文档结构的字段命名有如下限制：

字段名 _id 作为主键保存在集合中，它的值必须唯一且不可改变，它的值可以是除数组类型以外的其他任何类型。字段的命名不能是空串；不能以 \$ 开始；不能含有（.）。

 注：本文档的所有例子都是使用 SequoiaDB 的 shell 接口。

```
insert()
```

`insert()` 是向 SequoiaDB 集合中插入记录的主要方法，它有以下语法：

```
db.collectionspace.collection.insert(<doc|docs>, [flag])
```

插入第一个文档

如果 **集合空间** 和 **集合** 不存在，首先创建集合空间（如 `db.createCS("foo")`：创建集合空间 `foo`）和集合（如 `db.foo.createCL("bar")`：在集合空间下创建集合 `bar`），之后才能做插入操作。

```
db.foo.bar.insert(
{
    _id: 1,
    name: {first: "Jhon", last: "Black"},
    phone: [1853742000, 1802321000],
    remark: [
        {
            position: "manager",
            year: 2000
        },
        {
            position: "CEO",
            year: 2012
        }
    ]
})
```

可以使用 `find()` 方法确认是否插入成功。

```
db.foo.bar.find()
```

此操作返回结果如下：

```
{
    _id: 1,
    name: {first: "Jhon", last: "Black"},
    phone: [1853742000, 1802321000],
    remark: [
        {
            position: "manager",
            year: 2000
        },
        {
            position: "CEO",
            year: 2012
        }
    ]
}
```

不指定 `_id` 字段

如果新的文档记录不包含 `_id` 字段，`insert()` 方法向文档添加 `_id` 字段并生成一个唯一的 `$oid` 值

```
db.foo.bar.insert({name: "Tom", age: 20})
```

此操作是向集合 `bar` 中插入一条新的记录，记录 `name` 字段的值为“Tom”，`age` 字段的值为20，`_id` 字段被唯一创建：

```
{ "_id": { "$oid": "515152ba49af395200000000" }, "name": "Tom", "age": 20 }
```

插入多条记录

如果向 `insert` 方法中传一个数组类型的文档，`insert()` 方法将会在集合中执行批量插入。

下面的操作是向集合 bar 中插入两条记录。此操作也说明了 SequoiaDB 的动态模式的特点。尽管 _id:20 的记录含有字段名 phone 而在另一条记录中不存在，SequoiaDB 不要求其他记录必须含有此字段。

```
db.foo.bar.insert([{"name": "Mike", "age": 15}, {"_id": 20, "name": "John", "age": 25, "phone": 123}])
```

读取

四大基本数据库操作即（CRUD），读操作是从 SequoiaDB 中的集合中检索文档记录，包括所有响应应用程序请求数据并返回游标的操作。

本文档描述了应用程序从 SequoiaDB 中请求数据的查询语法和结构，以及不同因素影响效率的读取请求。



注：

本文档的所有例子都是使用 SequoiaDB 的 shell 接口。

`find()`

我们使用 `find` 方法读取 SequoiaDB 中的记录。`find` 方法是从集合中选择记录的主要方法，它返回一个包含很多记录的游标。它的语法结构如下：

```
db.collectionspace.collection.find([cond], [sel])
```

在 SQL 中对应的操作：`find()` 的方法与 `SELECT` 语句相似：

- . `[cond]` 参数对应 `WHERE` 语句
- . `[sel]` 参数对应从结果集中选择的字段列表

现集合中有如下一条记录：

```
{
  "_id": 1,
  "name": {
    "first": "Tom",
    "second": "David"
  },
  "age": 23,
  "birth": "1990-04-01",
  "phone": [
    10086,
    10010,
    10000
  ],
  "family": [
    {
      "Dad": "Kobe",
      "phone": 139123456
    },
    {
      "Mom": "Julie",
      "phone": 189123456
    }
  ]
}
```

返回集合所有记录

如果没有 `cond` 参数，方法 `db.collectionspace.collection.find()` 选择集合中所有的记录，如下返回集合空间 `foo` 中集合 `bar` 的所有记录：

```
db.foo.bar.find()
```

返回匹配条件的记录

- Equality 匹配

下面的操作返回集合 bar 中 age 等于23的记录

```
db.foo.bar.find({age: 23})
```

- 使用**匹配符**

下面操作返回集合 bar age 字段值大于20的记录

```
db.foo.bar.find({age: {$gt: 20}})
```

- 嵌套数组匹配

1. 数组元素查询，下面的操作操作返回一个游标，指向集合 bar 中所有数组类型字段 phone 含有元素10086的记录：

```
db.foo.bar.find({"phone": 10086})
```

2. 数组元素为 BSON 对象的查询，下面的操作返回一个游标指向集合 bar 中 family 字段包含的子元素 Dad 字段值为“Kobe”， phone 字段值为139123456的记录：

```
db.foo.bar.find(
  {
    "family": {
      $elemMatch: {
        "Dad": "Kobe",
        "phone": 139123456
      }
    }
  }
)
```

- 嵌套 BSON 对象匹配查询

下面的操作返回一个游标指向集合 bar 中嵌套 BSON 对象的 name 字段匹配{"first":"Tom"}的记录

```
db.foo.bar.find(
  {
    "name": {
      "first": "Tom"
    }
  }
)
```

上面还可以写成：

```
db.foo.bar.find(
  {
    "name.first": "Tom"
  }
)
```

指定返回记录字段

如果指定 find 方法的 sel 参数，那么只返回指定的 sel 参数内的字段名。下面的操作返回记录的 name 字段：

```
db.foo.bar.find(null, {name: ""})
```

注:

如果记录中不存在指定的字段名（如：people），SequoiaDB 默认也返回。如：

```
db.foo.bar.find({}, {name: "", people: ""})
返回: {
  "name": {
    "first": "Tom",
    "second": "David"
  },
  "people": ""
}
```

find() 更多信息

执行 db.foo.bar.find().help() , 会看到 find() 的更多使用方法

- cursor.sort(<sort>)

sort() 方法用来按指定的字段排序 , 语法格式为 : sort({“字段名1” : 1 |-1, “字段名2” : 1 |-1,...}) , 1为升序 , -1为降序。如果 find 方法的 sel 参数不设定内容 , sort() 方法按指定 sort 参数设定的字段排序 , 如果 sel 参数设定了返回的字段名 , 那么 sort() 方法只能对 sel 参数中选定的字段进行排序。如 :

```
db.foo.bar.find().sort({age: 1})
```

对返回的记录按 age 字段值的升序排序

```
db.foo.bar.find(null, {name: ""}).sort({age: 1})
```

此操作实际上对返回的记录达不到排序的效果。

- cursor_hint(<hint>)

添加索引加快查找速度 , 假设存在名为“testIndex”的索引 :

```
db.foo.bar.find().hint({": "testIndex"})
```

- cursor.limit(<num>)

在结果集中限制返回的记录条数 :

```
db.foo.bar.find().limit(3)
```

返回结果集里面的的前三条记录

- cursor.skip(<num>)

skip() 方法控制结果集的开始点 , 即跳过前面的 num 条记录 , 从 num+1 条记录开始返回 :

```
db.foo.bar.find().skip(5)
```

从查询的结果集的第6条记录开始返回

- 使用游标控制 find() 返回的记录

```
db.foo.bar.find().current() //返回当前游标指向的记录
```

```
db.foo.bar.find().next() //返回当前游标指向的下一条记录
```

```
db.foo.bar.find().close() //关闭当前游标, 当前游标不再可用
```

```
db.foo.bar.find().count() //返回当前游标的记录总数
```

```
db.foo.bar.find().size() //返回当前游标到最终游标的距离
```

```
db.foo.bar.find().toArray() //以数组形式返回结果集
```

更新

四大基本数据库操作即 (CRUD) , 更新操作即修改集合中已存在的记录。SequoiaDB 中使用 [update\(\)](#) 方法做更新操作。



注:

本文档的所有例子都是使用 SequoiaDB 的 shell 接口。

update()

update() 方法是修改集合中记录的主要方法 , 它的语法结构为 :

```
db.collectionspace.collection.update(<rule>, [cond], [hint])
```

在 SQL 中对应的操作 : update() 的方法与 update...set 语句相似 :

. <rule>参数对应 set 语句

- . [cond] 参数对应 where 语句
- . [hint] 参数是对应索引表里的名称

使用 update 操作修改记录

如果 update() 方法只有 rule 参数的表达式（例如使用 \$set 更新表达式），那么 update 方法会修改集合记录中所有指定的字段；更新嵌套对象 SequoiaDB 使用点（.）操作符。

- **更新记录字段**

使用 \$set 更新记录字段的值。下面的操作修改集合 bar 中符合条件 _id 字段值等于1的记录，使用 \$set 修改 name 字段的嵌套元素 first 字段的值，将它的值修改为“Mike”：

```
db.foo.bar.update({$set: {"name.first": "Mike"}}, {_id: 1})
```

 注：如果 rule 参数包含的字段名没有在当前的记录中，update() 方法会添加 rule 参数包含的字段到记录中。

- **删除记录字段**

使用 \$unset 删除记录的字段名。下面的操作是删除集合 bar 中所有含有 age 字段的记录，如果记录中没有 age 字段，跳过。

```
db.foo.bar.update({$unset: {"age": ""}})
```

- **数组元素更新**

如果需要更新数组中的元素，SequoiaDB 使用点操作符（.），数组下标从0开始。下面的操作是修改数组字段 arr 的第二个元素的值，将它的值添加为5：

```
db.foo.bar.update({$inc: {"arr.1": 5}})
```

- **hint 参数**

下面操作会通过索引遍历对所有记录的 name 字段内容修改为 Tom，“textIndex”为索引名称。

```
db.foo.bar.update({$set: {"name": "Tom"}}, null, {"": "textIndex"})
```

 注：更多更新操作符查看

删除

四大基本数据库操作即（CRUD），删除操作即移除集合中的记录。SequoiaDB 中使用 remove() 方法做删除操作。

 注：
本文档的所有例子都是使用 SequoiaDB 的 shell 接口。

remove()

remove() 方法是删除集合中记录主要方法，它的语法结构为：

```
db.collectionspace.collection.remove([cond], [hint])
```

在 SQL 中对应的操作：remove() 的方法与 DELETE 语句相似：

- . [cond] 参数对应 where 语句
- . [hint] 参数是对应索引表里的名称

删除集合记录

- **删除集合中的所有记录**

以下操作会删除集合 bar 中所有的记录。

```
db.foo.bar.remove()
```

- 删除集合中匹配条件的记录

以下操作会删除集合 bar 中所有匹配 name 字段值为“Tom”的记录。

```
db.foo.bar.remove({name: "Tom"})
```

- hint 参数

以下操作会通过索引遍历快速删除匹配条件的记录，“textIndex”为索引名称。

```
db.foo.bar.remove({name: "Tom"}, {"": "testIndex"})
```

SequoiaDB 应用程序开发

SequoiaDB 允许用户通过多种方式连接，如C，C++，Java，PHP，C#，Python，本章介绍各个语言的相关驱动信息。

C 驱动

本节介绍 C 驱动的相关驱动信息。

[C 驱动](#)

[C 开发环境搭建](#)

[C 开发基础](#)

[SQL to SequoiaDB shell to C](#)

[C API](#)

[C 驱动](#)

概述

C 客户端程序主要提供了数据库，集合空间，集合，游标，副本组，节点，大对象，域这8个级别的操作。

更多参考 [C 在线 API](#)

句柄

C 客户端驱动的句柄分为两类。一类用于数据库操作，另一类用于集群操作。

- 数据库操作句柄

SequoiaDB 数据库中的数据存放分为三个级别：

1) 数据库

2) 集合空间

3) 集合

每一个数据库中的集合空间没有物理上限，每个集合空间在单系统内存放为一个单独的文件，因此集合空间的数量取决于磁盘和内存的大小。每个集合空间可以包含最多4096个集合。每个集合可以包含多条记录。在一台物理系统内，每个集合空间最大256GB。对比关系型数据库，可以把记录看作关系型数据库的“行”；把集合看作关系型数据库的“表”。因此，在数据库操作中，可用3个句柄分别代表连接，集合空间，集合，1个句柄代表游标，1个句柄表示大对象：

sdbConnectionHandle	数据库连接句柄	代表一个单独的数据库连接
sdbCSHandle	集合空间句柄	代表一个单独的集合空间
sdbCollectionHandle	集合句柄	代表一个单独的集合
sdbCursorHandle	游标句柄	代表一个查询产生的结果集
sdbLobHandle	大对象句柄	代表一个大对象

C 客户端程序需要使用不同的句柄进行操作。譬如读取数据的操作需要游标句柄，而创建集合空间则需要数据库连接句柄。

注：

- 对于每一个连接，其产生的集合空间，集合，与游标句柄公用一个套接字。因此在多线程系统中，必须确保每个线程不会同时针对同一套接字，在同一时间发送或接收数据。
 - 一般来说，不建议使用多个线程共同操作一个连接句柄与其产生的其它句柄。
 - 如果每个线程使用自己的连接句柄以及其它产生的句柄，则可以保证线程安全。
- 集群操作句柄

SequoiaDB 数据库中的集群操作分为三个级别：

- 1) 分区组
- 2) 数据节点
- 3) 域

注：分区组包二种类型：编目分区组、数据分区组。

分区组实例，数据节点实例，域实例可以用以下句柄表示。

sdbReplicaGroupHandle	分区组句柄	代表一个单独的分区组
sdbNodeHandle	数据节点句柄	代表一个单独的数据节点
sdbDomainHandle	域句柄	代表一个单独的域

与集群相关的操作需要使用分区组及数据节点句柄。

`sdbReplicaGroupHandle` 的实例用于管理分区组。其操作包括启动、停止分区组，获取分区组中节点的状态、名称信息、数目信息。

`sdbNodeHandle` 的实例用于管理数据节点。其操作包括启动，停止指定的数据节点，获取指定数据节点实例，获取主从数据节点实例，获取数据节点地址信息。

`sdbDomainHandle` 的实例用于修改，获取域信息。

错误信息

每个函数都有返回值，返回值的定义如下：

`SDB_OK`（数据值为0）：表示执行成功；

< 0 : 表示数据库错误，具体的错误描述在 C 驱动开发包中 `include/ossErr.h` 文件中可以找到；

> 0 : 表示系统错误，请查阅相关系统的错误码信息。

C 开发环境搭建

获取驱动开发包

从 <http://www.sequoiadb.com> 下载对应操作系统版本的 SequoiaDB 驱动开发包。

配置开发环境

- Linux
 1. 解压下来的驱动开发包；
 2. 将压缩包中的 `driver` 目录，拷贝到开发工程目录中（建议放在第三方库目录下），并命名为 `sdbdriver`；
 3. 将 `sdbdriver/include` 目录加入到编译头目录，并将 `sdbdriver/lib` 目录加入连接目录。

动态链接：

使用 lib 目录下的 libsdbsc.so 动态库，gcc 编译参数形式如：

```
cc testClient.c -o testClientC -I <PATH>/sdbdriver/include -L <PATH>/sdbdriver/lib -lsdbc
```

其中：PATH 为 sdbdriver 放置路径；运行程序时，用户需要将 LD_LIBRARY_PATH 路径指定为包含 libsdbsc.so 动态库的路径。

```
export LD_LIBRARY_PATH=<PATH>/sdbdriver/lib
```

-  注：

如果运行程序时会出现错误提示：

```
error while loading shared libraries: libsdbsc.so: cannot open shared object file:  
No such file or directory
```

表示没有正确设置 LD_LIBRARY_PATH，LD_LIBRARY_PATH 是环境变量，建议设置到 /etc/profile 或者应用程序的启动脚本中，避免每次新开终端都需要重新设置。

静态链接：

使用 lib 目录下的 libstaticsdbsc.a 静态库，gcc 编译参数形式如：

```
cc testClient.c -o testClientC -I <PATH>/sdbdriver/include #L <PATH>/sdbdriver/lib/  
libstaticsdbsc.a -lm
```

- Windows

暂未推出 Windows 驱动开发包。

C 开发基础

本节介绍使用 C 程序运行 SequoiaDB。首先安装 SequoiaDB，安装信息请查看 [SequoiaDB 服务器安装](#) 章节。

这里介绍如何使用 C 客户端驱动接口编写使用 SequoiaDB 数据库的程序。为了简单起见，下面的例子不全部是完整的代码，只起示例性作用。可到 /sequoiadb/client/samples/C 下获取相应的完整的代码。

数据库操作

- 数据库连接（Connecting）

编写完整客户端文件 connect.c 演示连接到数据库。文件必须包含“client.h”头文件。

```
#include <stdio.h>  
#include "client.h"  
  
// Display Syntax Error  
void displaySyntax ( CHAR *pCommand )  
{  
    printf ( "Syntax: %s<hostname> <servicename> <username> <password>"  
            OSS_NEWLINE, pCommand ) ;  
}  
  
INT32 main ( INT32 argc, CHAR **argv )  
{  
    // define a connection handle; use to connect to database  
    sdbConnectionHandle connection = 0 ;  
    INT32 rc = SDB_OK ;  
  
    // verify syntax  
    if ( 5 != argc )  
    {  
        displaySyntax ( (CHAR*)argv[0] ) ;  
        exit ( 0 ) ;  
    }
```

```

// read argument
CHAR *pHostName      = (CHAR*) argv[1] ;
CHAR *pServiceName   = (CHAR*) argv[2] ;
CHAR *pUsr           = (CHAR*) argv[3] ;
CHAR *pPasswd         = (CHAR*) argv[4] ;

// connect to database
rc = sdbConnect ( pHostName, pServiceName, pUsr, pPasswd, &connection ) ;
if ( rc!=SDB_OK )
{
    printf("Fail to connet to database, rc = %d" OSS_NEWLINE, rc ) ;
    goto error ;
}

done:
// disconnect from database
sdbDisconnect ( connection ) ;
// release connection
sdbReleaseConnection ( connection ) ;
return 0 ;
error:
goto done ;
}

```

在 Linux 下，可以如下编译及链接动态链接库文件 libsdbs.so。

```
$gcc -o connect connect.c -I /<PATH>/sdbdriver/include -lsdbc -L /<PATH>/sdbdriver/lib
$ ./connect localhost 11810 "" ""
connect success!
```

 注：本例程连接到本地数据库的11810端口，使用的是空的用户名和密码。用户需要根据自己的实际情况配置参数。但如果数据库已经创建用户，可以使用已经创建的用户及密码连接到数据库。

- 创建集合空间，集合

以下创建了一个名字为“foo”的集合空间和一个名字为“bar”的集合，集合空间内的集合的数据页大小为4k。可根据实际情况选择不同的数据页。创建集合后，可对集合做增删改查等操作。

```

// 首先，定义集合空间、集合句柄。
sdbCSHandle collectionspace      = 0 ;
sdbConnectionHandle connection    = 0 ;
// 创建集合空间"foo"
rc = sdbCreateCollectionSpace ( connection, "foo", SDB_PAGESIZE_4K, &collectionspace ) ;
// 在新建立的集合空间中创建集合"bar"
rc = sdbCreateCollection ( collectionspace, "bar", &collection ) ;

```

 注：在创建集合“bar”时并没有附加分区、压缩等信息，关于创建集合的更详细情况，请参考详情
请查阅 [C API](#)

- 插入数据

SequoiaDB 存储数据采用 BSON 的格式，BSON 是一种类似 JSON 的二进制对象。保存数据库中的数据，首先必须创建 bson 对象。下面会将{"name:"Tom",age:24}插入到集合中。

```

// 首先，我们需要创建一个插入的 bson 对象。
INT32 rc = SDB_OK ;
bson obj ;
bson_init( &obj ) ;
bson_append_string( &obj, "name", "tom" ) ;
bson_append_int( &obj, "age", 24 ) ;
rc = bson_finish( &obj ) ;
if ( rc != SDB_OK )
printf("Error.");
// 接着，把此 bson 对象插入集合中
rc = sdbInsert ( collection, &obj ) ;

```

- **查询**

查询操作需要一个游标句柄存放查询的结果到本地。要获得查询的结果需要使用游标操作。本例使用了游标操作的 sdbNext 接口，表示从查询结果中取到一条记录。此示例中没有设置查询条件，筛选条件，排序情况，及仅使用默认索引。

```
// 定义一个游标句柄
sdbCursorHandle cursor = 0 ;
...
// 查询所有记录，查询结果放在游标句柄中
rc = sdbQuery(collection, NULL, NULL, NULL, 0, -1, &cursor) ;
// 从游标中显示所有记录
bson_init(&obj);
while( !( rc=sdbNext( cursor, &obj ) ) )
{
    bson_print( &obj ) ;
    bson_destroy(&obj) ;
    bson_init(&obj);
}
bson_destroy(obj) ;
```

- **索引**

此处，我们在集合句柄 collection 指定的集合中创建一个以“name”为升序，“age”为降序的索引。

```
#define INDEX_NAME "index"
...
// 首先创建一个 bson 对象包含将要创建的索引的信息
bson_init( &obj ) ;
bson_append_int( &obj, "name", 1 ) ;
bson_append_int( &obj, "age", -1 ) ;
rc = bson_finish( &obj ) ;
if ( rc != SDB_OK )
printf("Error.");
// 创建一个以"name"为升序，"age"为降序的索引
rc = sdbCreateIndex ( collection, &obj, INDEX_NAME, FALSE, FALSE ) ;
bson_destroy ( &obj ) ;
```

- **更新**

此处，我们在集合句柄 collection 指定的集合中更新记录。因为没有指定数据匹配规则，所以此示例将更新集合中所有的集合。

```
// 先创建一个包含更新规则的 bson 对象
bson_init( &rule ) ;
bson_append_start_object ( &rule, "$set" ) ;
bson_append_int ( &rule, "age", 19 ) ;
bson_append_finish_object ( &rule ) ;
rc = bson_finish ( &rule ) ;
if ( rc != SDB_OK )
printf("Error.");
// 打印出更新规则
bson_print( &rule ) ;
// 更新记录
rc = sdbUpdate( collection, &rule, NULL, NULL ) ;
bson_destroy(&rule);
```

此处，因为没有指定记录匹配条件，所以此示例将更新集合句柄 collection 指定的集合中所有的记录。

集群操作

- **分区组操作**

分区组操作包括创建分区组 (sdbCreateReplicaGroup) , 得到分区组句柄 (sdbGetReplicaGroup) , 启动分区组 (sdbStartReplicaGroup) , 停止分区组 (sdbStopReplicaGroup) 等。以下为分区组操作示例性的例子。真正的应用应包括错误检测等。

```
// 定义一个分区组句柄
sdbReplicaGroupHandle rg = 0 ;
...
// 先建立一个编目分区组
rc = sdbCreateCataReplicaGroup ( connection, HOST_NAME, SERVICE_NAME, CATALOG_SET_PATH ,
NULL ) ;
// 创建数据分区组
rc = sdbCreateReplicaGroup ( connection, GROUP_NAME, &rg ) ;
// 创建数据节点
rc = sdbCreateNode ( rg, HOST_NAME1, SERVICE_NAME1, DATABASE_PATH1, NULL ) ;
// 启动分区组
rc = sdbStartReplicaGroup( rg ) ;
```

- 数据节点操作

数据节点操作包括创建数据节点 (sdbCreateNode) , 得到主数据节点 (sdbgetNodeMaster) , 得到从数据节点 (sdbgetNodeSlave) , 启动数据节点 (sdbStartNode) , 停止数据节点 (sdbStopNode) 等。以下为数据节点操作示例性的例子。真正的应用应包括错误检测等。

```
// 定义一个数据节点句柄
sdbNodeHandle masternode    = 0 ;
sdbNodeHandle slavenode     = 0 ;
...
// 获取主数据节点
rc = sdbgetNodeMaster ( rg, &masternode ) ;
// 获取从数据节点
rc = sdbgetNodeSlave ( rg, &slavenode ) ;
```

SQL to SequoiaDB shell to C

SequoiaDB 的查询用 json (bson) 对象表示 , 下表以例子的形式显示了 SQL 语句 , SequoiaDB shell 语句和 SequoiaDB C 驱动程序语法之间的对照。

SQL	SequoiaDB shell	C Driver
insert into students(a,b) values(1,-1)	db.foo.bar.insert({a:1,b:-1})	const char *r ="{a:1,b:-1}" ; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (&obj, r); // collection 为集合 bar 的句柄 sdbInsert (collection, &obj);
select a,b from students	db.foo.bar.find(null,{a:"",b:""})	const char *r ="{a:W" W",b:W" W"}" ; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& select, r); // collection 为集合 bar 的句柄 , cursor 为返回查询结果的句柄 sdbQuery (collection, NULL, &select, NULL, NULL, 0, -1, cursor);
select * from students	db.foo.bar.find()	// collection 为集合 bar 的句柄 , cursor 为返回查询结果的句柄 sdbQuery (collection, NULL, NULL, NULL, NULL, 0, -1, cursor);
select * from students where age=20	db.foo.bar.find({age:20})	const char *r ="{age:20}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (&condition, r); // collection 为集合 bar 的句柄 , cursor 为返回查询结果的句柄 sdbQuery (collection, & condition, NULL, NULL, NULL, 0, -1, cursor);

SQL	SequoiaDB shell	C Driver
select * from students where age=20 order by name	db.foo.bar.find({age:20}).sort({name:1})	const char *r1 ="{age:20}"; const char *r2 ="{name:1}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& condition, r1); jsonToBson (&orderBy, r2); // collection 为集合 bar 的句柄 , cursor 为返回查询结果的句柄 sdbQuery (collection, & condition, NULL, & orderBy, NULL, 0, -1, cursor);
select * from students where age>20 and age<30	db.foo.bar.find({age:{\$gt:20,\$lt:30}})	const char *r ="{age:{\$gt:20,\$lt:30}}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& condition, r); // collection 为集合 bar 的句柄 , cursor 为返回查询结果的句柄 sdbQuery (collection, & condition , NULL, NULL, NULL, 0, -1, cursor);
create index testIndex on students(name)	db.foo.bar.createIndex("testIndex", {name:1},false)	const char *r ="{name:1}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& obj, r); // collection 为集合 bar 的句柄 sdbCreateIndex (collection, & obj, "testIndex", FALSE, FALSE)
select * from students limit 20 skip 10	db.foo.bar.find().limit(20).skip(10)	// collection 为集合 bar 的句柄 , cursor 为返回查询结果的句柄 sdbQuery (collection, NULL, NULL, NULL, NULL, 10, 20, cursor);
select count(*) from students where age>20	db.foo.bar.find({age:{\$gt:20}}).count()	const char *r ="{age:{\$gt:20}}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& condition, r); // collection 为集合 bar 的句柄 , count 为返回总数 sdbGetCount (collection, & condition, & count);
update students set a=a+2 where b=-1	db.foo.bar.update({\$set:{a:2}},{b:-1})	const char *r1 ="{\$set:{a:2}}"; const char *r2 ="{b:-1}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& rule, r1); jsonToBson (& condition, r2); // collection 为集合 bar 的句柄 sdbUpdate (collection, &rule, &condition, NULL);
delete from students where a=1	db.foo.bar.remove({a:1})	const char *r ="{a:1}"; // jsonToBson 将一个 json 字符串转换为 bson 对象 jsonToBson (& condition, r); // collection 为集合 bar 的句柄 sdbDelete (collection, & condition, NULL);

C API

此部分是相关 C 的 API 文档。

[C API](#)

历史更新情况：

Version 1.10

1. 添加获取查询访问计划的接口：

sdbExplain, 获取查询的访问计划

2. 添加用于大对象 (lob) 操作的接口：

```
sdbListLobs, 列出集合中的所有lob
sdbOpenLob, 创建或打开一个lob
sdbCloseLob, 关闭一个lob
sdbRemoveLob, 删除一个lob
sdbSeekLob, 设置读起始位置, 该版本中, seek只用于读操作
sdbReadLob, 从lob中读取数据
sdbWriteLob, 把数据写入lob
sdbGetLobSize, 获取lob的大小
sdbGetLobCreateTime, 获取lob的创建时间
```

Version 1.8

新添加接口：

```
sdbConnect1, 可提供多个地址, 接口随机选择一个有效的地址连接。
sdbCreateCollectionSpaceV2, 提供一个 bson 的选项, 使创建集合空间更加灵活
sdbAlterCollection, 修改集合(表)属性
sdbCreateDomain, 创建域
sdbDropDomain, 删除域
sdbGetDomain, 获取域句柄
sdbListDomains, 列出所有域
sdbReleaseDomain, 删除域句柄
sdbAlterDomain, 更改域属性
```

Version 1.6

1. 使用 sdbNodeHandle 来取代原来的 sdbReplicaNodeHandle。sdbReplicaNodeHandle 将在 version 2.x 中被弃用。

2. 使用概念“node”取代原来的“replica node”，和“replica node”相关的 API 接口将保留，直到 version 2.x 会被弃用。

详情请查看相关 API。

C++ 驱动

本节介绍 C++ 驱动的相关驱动信息。

[C++ 驱动](#)

[C++ 开发环境搭建](#)

[C++ 开发基础](#)

[SQL to SequoiaDB shell to C++](#)

[C++ API](#)

[C++ 驱动](#)

概述

C++ 客户端驱动提供了数据库操作和集群操作的接口。主要包括以下8个级别的操作：数据库，集合空间，集合，游标，副本组，节点，域，大对象。更多参考 [C++ 在线 API](#)

C++ 类实例

C++ 客户端驱动的有两种类实例。一种用于数据库操作，另一种用于集群操作。

- 数据库操作实例

SequoiaDB 数据库中的数据存放分为三个级别：

- 1) 数据库
- 2) 集合空间
- 3) 集合

因此，在数据库操作中，可用3个类来分别表示数据库连接，集合空间，集合，1个类表示游标，1个类表达大对象：

Sdb	数据库类	数据库类主要用于管理整个数据库，包括建立连接，创建集合空间等
sdbCollectionSpace	集合空间类	集合空间主要用于管理集合
sdbCollection	集合类	集合类主要用于对数据进行增删改查等操作
sdbCursor	游标类	游标类主要用于遍历查询、快照返回的结果游标 实例代表一个查询产生的游标
sdbLob	大对象类	大对象类用于对大对象进行读写等操作

C++ 客户端需要使用不同的实例进行操作。譬如读取数据的操作需要游标实例，而创建表空间则需要数据库实例。

注:

- 对于每一个连接，其产生的集合空间，集合，与游标句柄公用一个套接字。因此在多线程系统中，必须确保每个线程不会同时针对同一套接字，在同一时间发送或接收数据。
- 一般来说，不建议使用多个线程共同操作一个连接句柄与其产生的其它句柄。
- 如果每个线程使用自己的连接句柄以及其它产生的句柄，则可以保证线程安全。

- 集群操作实例

SequoiaDB 数据库中的集群操作分为三个级别：

- 1) 分区组
- 2) 数据节点
- 3) 域

注:

分区组包两种类型：编目分区组，数据分区组。

分区组实例，数据节点实例，域实例可以用以下三种类的实例表示。

sdbReplicaGroup	分区组类	分区组实例代表一个单独的分区组
sdbNode	数据节点类	数据节点实例代表一个单独的数据节点
sdbDomain	域类	域实例代表一个管理若干个分区组的域

与集群相关的操作需要使用分区组及数据节点实例。

sdbReplicaGroup 的实例用于管理分区组。其操作包括启动，停止分区组，获取分区组中节点的状态，名称信息，数目信息。

sdbNode 的实例用于管理数据节点。其操作包括启动，停止指定的数据节点，获取指定数据节点实例，获取主从数据节点实例，获取数据节点地址信息。

sdbDomain 的实例用于管理域。其包括修改域，获取域信息等操作。

错误信息

每个函数都有返回值，返回值的定义如下：

SDB_OK (数据值为0) : 表示执行成功；

- < 0 : 表示数据库错误，具体的错误描述在 C++ 驱动开发包中 include/ossErr.h 文件中可以找到；
- > 0 : 表示系统错误，请查阅相关系统的错误码信息。

C++ 开发环境搭建

获取驱动开发包

从 <http://www.sequoiadb.com> 下载对应操作系统版本的 SequoiaDB 驱动开发包。

配置开发环境

- Linux

1. 解压下来的驱动开发包；
2. 将压缩包中的 driver 目录，拷贝到开发工程目录中（建议放在第三方库目录下），并命名为 sdbdriver。
3. 将 sdbdriver/include 目录加入到编译头目录，并将 sdbdriver/lib 目录加入连接目录。

动态链接：

使用 lib 目录下的 libsdbcpp.so 动态库，g++ 编译参数形式如：

```
g++ main.cpp -o test -I <PATH>/sdbdriver/include -L <PATH>/sdbdriver/lib -lsdbcpp
```

其中：PATH 为 sdbdriver 放置路径；运行程序时，用户需要将 LD_LIBRARY_PATH 路径指定为包含 libsdbcpp.so 动态库的路径。

```
export LD_LIBRARY_PATH=<PATH>/sdbdriver/lib
```

-  注：

如果运行程序时会出现错误提示：

```
error while loading shared libraries: libsdbcpp.so: cannot open shared object
file: No such file or directory
```

表示没有正确设置 LD_LIBRARY_PATH，LD_LIBRARY_PATH 是环境变量，建议设置到 /etc/profile 或者应用程序的启动脚本中，避免每次新开终端都需要重新设置。

静态链接：

使用 lib 目录下的 libstaticsdbc.a 静态库，g++ 编译参数形式如：

```
g++ main.c -o test -I <path>/sdbdriver/include #L <path>/sdbdriver/lib/
libstaticsdbc.a #lm -lpthread
```

- Windows

暂未推出 Windows 驱动开发包。

C++ 开发基础

这里介绍如何使用 C++ 客户端驱动接口编写使用 SequoiaDB 数据库的程序。为了简单起见，下面的示例不全部是完整的代码，只起示例性作用。可到 /sequoiadb/client/samples/CPP 下获取相应的完整的代码。更多查看 [C++ API](#)

数据库操作

- 连接数据库：Connecting

connect.cpp 演示如何连接到数据库。文件应当包含“client.hpp”头文件及使用命名空间 sdbclient。

```
#include <iostream>
#include "client.hpp"

using namespace std;
```

```

using namespace sdbclient ;

// Display Syntax Error
void displaySyntax ( CHAR *pCommand ) ;

INT32 main ( INT32 argc, CHAR **argv )
{
    // verify syntax
    if ( 5 != argc )
    {
        displaySyntax ( (CHAR*)argv[0] ) ;
        exit ( 0 ) ;
    }
    // read argument
    CHAR *pHostName      = (CHAR*) argv[1] ;
    CHAR *pPort          = (CHAR*) argv[2] ;
    CHAR *pUsr           = (CHAR*) argv[3] ;
    CHAR *pPasswd         = (CHAR*) argv[4] ;

    // define local variable
    sdb connection ;
    INT32 rc = SDB_OK ;

    // connect to database
    rc = connection.connect ( pHostName, pPort, pUsr, pPasswd ) ;
    if ( rc!=SDB_OK )
    {
        cout <<"Fail to connet to database, rc = "<<rc<<endl ;
        goto error ;
    }
    else
        cout<<"Connect success!"<<endl ;

done:
    // disconnect from database
    connection.disconnect () ;
    return 0 ;
error:
    goto done ;
}

// Display Syntax Error
void displaySyntax ( CHAR *pCommand )
{
    cout<<"Syntax: "<<pCommand" <hostname> <servicename>\n
    usernamepassword"<endl ;
}

```

在 Linux下，可以如下编译及链接动态链接库文件 libsdbcpp.so:

```
$g++ -o connect connect.cpp -I <PATH>/sdbdriver/include -lsdbcpp -L <PATH>/sdbdriver/lib
执行结果如下:
$ ./connect localhost 11810 " " "
Connect success!
```

 **注:**

本例程连接到本地数据库的11810端口，使用的是空的用户名和密码。用户需要根据自己的实际情况配置参数。譬如，./connect localhost 11810 "sequoiadb" "sequoiadb"。当数据库已经创建用户时，应该使用正确的用户及密码连接到数据库，否则连接失败。

- **创建集合空间和集合**

```
// 首先，定义集合空间，集合对象。
sdbCollectionSpace collectionspace ;
sdbCollection collection ;
```

```
// 创建集合空间"foo"
rc = connection.createCollectionSpace ( "foo", SDB_PAGESIZE_4K, collectionspace ) ;
// 在新建立的集合空间中创建集合"bar"
rc = collectionspace.createCollection ( "bar", collection ) ;
```

以上创建了一个名字为“foo”的集合空间和一个名字为“bar”的集合，集合空间内的集合的数据页大小为4k。可根据实际情况选择不同大小的数据页。创建集合后，可对集合做增删改查等操作。

注：

在创建集合“bar”时并没有附加分区，压缩等信息，详情请查阅 [C++ API](#)

- 插入数据：insert

```
// 首先，需要创建一个插入的 bson 对象。
BSONObj obj ;
obj = BSON ( "name" << "tom" << "age" << 24 ) ;
// 接着，把此 bson 对象插入集合中
collection.insert ( obj ) ;
```

obj 为输入参数，为要插入的数据。

- 查询：query

```
// 定义一个游标对象
sdbCursor cursor ;
...
// 查询所有记录，并把查询结果放在游标对象中
collection.query ( cursor ) ;
// 从游标中显示所有记录
while( !( rc=cursor.next( obj ) ) )
{
    cout << obj.toString() << endl ;
}
```

查询操作需要一个游标对象存放查询的结果到本地。要获得查询的结果需要使用游标操作。本例使用了游标操作的 next 接口，表示从查询结果中取到一条记录。此示例中没有设置查询条件，筛选条件，排序情况，及仅使用默认索引。

- 创建索引：index

```
#define INDEX_NAME "index"
...
// 首先创建一 BSONObj 对象包含将要创建的索引的信息
BSONObj obj ;
obj = BSON ( "name" << 1 << "age" << -1 ) ;
// 创建一个以"name"为升序，"age"为降序的索引
collection.createIndex ( obj, INDEX_NAME, FALSE, FALSE ) ;
```

集合对象 collection 中创建一个以"name"为升序，"age"为降序的索引。

- 更新：update

```
// 先创建一个包含更新规则的 BSONObj 对象
BSONObj rule = BSON ( "$set" << BSON ( "age" << 19 ) ) ;
// 打印出更新规则
cout << rule.toString() << endl ;
// 更新记录
collection.update( rule ) ;
```

在集合对象 collection 中更新了记录。实例中没有指定数据匹配规则，所以此示例将更新集合中所有的集合。

集群操作

- 分区组操作

分区组操作包括创建分区组（`sdb:createReplicaGroup`），得到分区组实例（`sdb:getReplicaGroup`），启动分区组所有数据节点（`sdbReplicaGroup::start`），停止分区组所有数据节点（`sdbReplicaGroup::stop`）等。

以下为分区组操作示例性的例子。真正的应用应包括错误检测等。

```
// 定义一个分区组实例
sdbReplicaGroup rg ;
// 定义一个空的 map 对象表示创建数据节点没有更多的配置内容
map<string, string> config ;
...
// 先建立一个编目分区组
connection.createCataReplicaGroup ( HOST_NAME, SERVICE_NAME, CATALOG_GROUP_PATH, NULL ) ;
// 创建数据分区组
connection.createRG ( REPLICA_GROUP_NAME, rg ) ;
// 创建第一个数据节点
rg.createNode ( HOST_NAME1, SERVICE_NAME1, DATABASE_PATH1, config ) ;
...
// 启动分区组
rg.start () ;
```

- 数据节点操作

数据节点操作包括创建数据节点（`sdbReplicaGroup::createNode`），得到主数据节点（`sdbReplicaGroup::getMaster`），得到从数据节点（`sdbReplicaGroup::getSlave`），启动数据节点（`sdbNode::start`），停止数据节点（`sdbNode::Stop`）等。

以下为数据节点操作示例性的例子。真正的应用应包括错误检测等。

```
// 定义一个数据节点实例
sdbNode masternode ;
sdbNode slavenode ;
...
// 获取主数据节点
rg.getMaster( masternode ) ;
// 获取从数据节点
rg.getSlave( slavenode ) ;
```

SQL to SequoiaDB shell to C++

SequoiaDB 的查询用 json (bson) 对象表示，下表以例子的形式显示了 SQL 语句，SequoiaDB shell 语句和 SequoiaDB C++ 驱动程序语法之间的对照。

SQL	SequoiaDB shell	C++ Driver
<code>insert into students(a,b) values(1,-1)</code>	<code>db.foo.bar.insert({a:1,b:-1})</code>	<code>sdbCollection collection ;</code> <code> BSONObj obj = BSON("a" << 1 << "b" << -1) ;</code> <code>collection.insert(obj) ;</code>
<code>select a,b from students</code>	<code>db.foo.bar.find(null,{a:"",b:""})</code>	<code>sdbCollection collection ;</code> <code>sdbCursor cursor ;</code> <code> BSONObj selected ;</code> <code> BSONObj obj ;</code> <code> selected = BSON("a" << "" << "b" << "") ;</code> <code> collection .query(cursor, obj, selected) ;</code>
<code>select * from students</code>	<code>db.foo.bar.find()</code>	<code>sdbCollection collection ;</code> <code>sdbCursor cursor ;</code> <code> collection .query (cursor) ;</code>
<code>select * from students where age=20</code>	<code>db.foo.bar.find({age:20})</code>	<code>sdbCollection collection ;</code> <code>sdbCursor cursor ;</code> <code> BSONObj condition ;</code> <code> condition = BSON("age" << 20) ;</code> <code> collection .query (cursor, condition) ;</code>
<code>select * from students where age=20 order by name</code>	<code>db.foo.bar.find({age:20}).sort({name:1})</code>	<code>sdbCollection collection ;</code>

SQL	SequoiaDB shell	C++ Driver
		sdbCursor cursor ; BSONObj obj ; BSONObj condition ; BSONObj orderBy ; condition = BSON("age"<<20) ; orderBy = BSON("name"<<1) ; collection .query (cursor, condition , obj, orderBy , obj) ;
select * from students where age>20 and age<30	db.foo.bar.find({age:{\$gt:20,\$lt:30}})	sdbCollection collection ; sdbCursor cursor ; BSONObj condition ; condition = BSON("age"<<BSON("\$gt"<<20<< \$lt"<<30)) ; collection .query (cursor, condition) ;
create index testIndex on students(name)	db.foo.bar.createIndex("testIndex", {name:1},false)	sdbCollection collection ; BSONObj obj ; obj = BSON("name"<<1) ; collection.createIndex (&obj, "testIndex", FALSE, FALSE)
select * from students limit 20 skip 10	db.foo.bar.find().limit(20).skip(10)	sdbCollection collection ; BSONObj obj ; sdbCursor cursor ; collection .query (cursor,obj, obj, obj, 10, 20) ;
select count(*) from students where age>20	db.foo.bar.find({age:{\$gt:20}}).count()	sdbCollection collection ; SINT64 count = 0 ; BSONObj condition ; Condition = BSON("age"<<BSON("\$gt"<<20)) ; collection.getCount (count, condition);
update students set a=a+2 where b=-1	db.foo.bar.update({\$set:{a:2}},{b:-1})	sdbCollection collection ; BSONObj condition = BSON("b"<<1) ; BSONObj rule = BSON("\$set"<<BSON("a"<<2)) ; BSONObj obj ; collection.update (rule, condition, obj) ;
delete from students where a=1	db.foo.bar.remove({a:1})	sdbCollection collection ; BSONObj condition = BSON("a"<<1) ; collection.del (condition) ;

C++ API

此部分是相关 C++ 的 API 文档。

[C++ API](#)

历史更新情况：

Version 1.10

1. SdbCollection 类添加的接口：

explain, 获取查询的访问计划
 createLob, 创建一个新的lob
 openLob, 打开一个已存在的lob, 该版本中, 打开的lob只用于读操作
 removeLob, 删除一个lob
 listLobs, 列出当前collection中的所有lob

2. 添加类 sdbLob 用于大对象操作, 其接口如下:

read, 从lob中读取数据

```

write, 把数据写入lob中
seek, 设置读起始位置, 该版本中, seek只用于读操作
close, 关闭一个新创建的或打开的lob
getOid, 获取lob的oid
getSize, 获取lob的大小
getCreateTime, 获取lob的创建时间

```

Version 1.8

1. sdb 类新添加的接口 :

```

connect, 可提供多个地址, 接口随机选择一个有效的地址连接。
createCollectionSpace, 提供一个 BSONObject 的选项, 使创建集合空间更加灵活
backupOffline, 离线备份支持更多的选项
createDomain, 创建域
getDomain, 获得域
dropDomain, 删除域
listDomain, 列出所有域

```

2. sdbCollection 类新添加的接口 :

```
alterCollection, 修改集合(表)属性
```

3. 添加 Domain 类用于与域相关的操作

Version 1.6

1. 添加类 Node 来取代原来的类 ReplicaNode。类 ReplicaNode 以及与它相关的方法将在 version 2.x 中被弃用。

详情请查看相关 API。

Java 驱动

本节介绍 Java 的相关驱动信息。

[Java 驱动](#)

[Java 开发环境搭建](#)

[Java 开发基础](#)

[Java BSON 使用](#)

[Java Datasource 介绍](#)

[SQL to SequoiaDB shell to Java](#)

[Java API](#)

[Java 驱动](#)

概述

SequoiaDB Java 驱动提供了数据库操作和集群操作的接口。主要包括以下8个级别的操作：数据库，集合空间，集合，游标，副本组，节点，域，大对象。

Java 驱动的有两种类实例。一种用于数据库操作，另一种用于集群操作。

- 数据库操作实例

SequoiaDB 数据库中的数据存放分为三个级别：

1) 数据库

2) 集合空间

3) 集合

因此，在数据库操作中，可用3个类来分别表示连接，集合空间，集合实例，另2个类分别表示游标实例和大对象实例：

SequoiaDB	数据库实例	代表一个单独的数据库连接
CollectionSpace	集合空间实例	代表一个单独的集合空间
DBCollection	集合实例	代表一个单独的集合
DBCursor	游标实例	代表一个查询产生的结果集
DBBlob	大对象实例	代表一个大对象

Java 驱动需要使用不同的实例进行操作。譬如读取数据的操作需要游标实例，而创建表空间则需要数据库实例。



SequoiaDB 只建立一条 Socket 连接，且内部没有对网络操作加锁。如果需要多线程连接数据库，各个线程必须各自新建一个 SequoiaDB 对象及其之上的 CollectionSpace/DBCollection/DBCursor 对象。

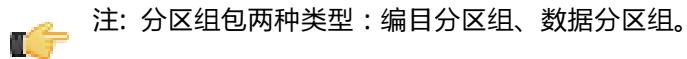
- 集群操作实例

SequoiaDB 数据库中的集群操作分为三个级别：

1) 分区组

2) 数据节点

3) 域



分区组实例和数据节点实例可以用以下三种类的实例表示。

ReplicaGroup	分区组类	分区组实例代表一个单独的分区组
Node	数据节点类	数据节点实例代表一个单独的数据节点
sdbDomain	域类	域实例代表一个管理若干个分区组的域

与集群相关的操作需要使用分区组及数据节点实例。

ReplicaGroup 的实例用于管理分区组。其操作包括启动，停止分区组，获取分区组中节点的状态，名称信息，数目信息。

Node 的实例用于管理节点。其操作包括启动，停止指定的节点，获取指定节点实例，获取主从节点实例，获取数据节点地址信息。

sdbDomain 的实例用于管理域。其包括修改域，获取域信息等操作。

错误信息

- 当执行出现异常时，大部分接口都会抛出 com.sequoiadb.exception.BaseException 和 java.lang.Exception 异常，分别对应于数据库引擎返回的异常信息和客户端本地的异常信息；
- BaseException 的异常信息可以通过该类的 getErrorType，getErrorCode 和 getMessage 方法获取。

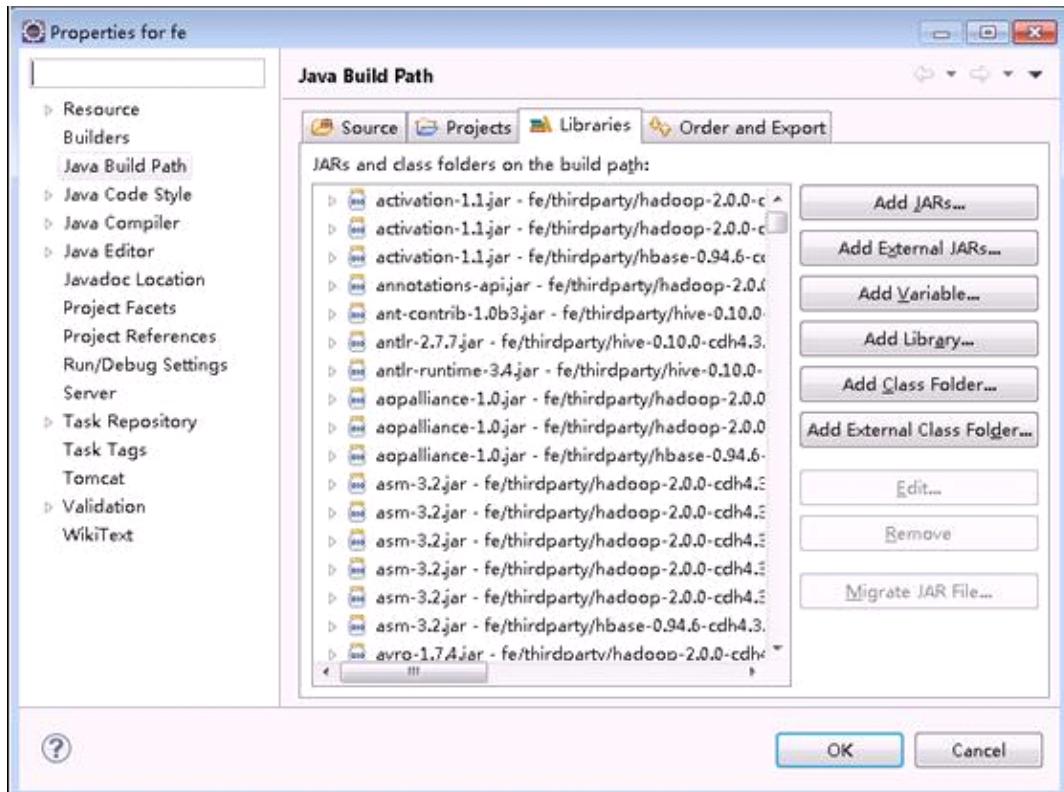
Java 开发环境搭建

获取驱动开发包

从 <http://www.sequoiadb.com> 下载对应操作系统版本的 SequoiaDB 驱动开发包；解压驱动开发包，从 driver/java/ 目录中获取 sequoiadb.jar 文件。

配置 Eclipse 开发环境

1. 将 SequoiaDB 驱动开发包中的 sequoiadb.jar 文件拷贝到工程文件目录下（建议将其放置在其他所有依赖库目录，如 lib 目录）；
2. 在 Eclipse 界面中，创建/打开开发工程；
3. 在 Eclipse 主窗口左侧的“Package Explore”窗口中，选择开发工程，并点击鼠标右键；
4. 在菜单中选择“properties”菜单项；
5. 在弹出的“property for project ...”窗口中，选择“Java Build Path”->“Libraries”，如下图所示：



6. 点击“Add JARs..”按钮，选择添加 sequoiadb.jar 到工程中；
7. 点击“OK”完成环境配置。

更多操作请参考[Java 开发基础](#)

Java 开发基础

这里介绍如何使用Java驱动接口编写使用SequoiaDB数据库的程序。为了简单起见，下面的示例不全部是完整的代码，只起示例性作用。可到安装目录/client/samples/java下获取相应的完整的代码。[更多查看Java API](#)

数据操作

- 连接数据库：Connecting 如下是一个连接数据库，并列出所有集合信息的一个例子：

```
import com.sequoiadb.base.DBCursor;
import com.sequoiadb.base.Sequoiadb;
import com.sequoiadb.exception.BaseException;

public class Sample {
    public static void main(String[] args) {

        String connString = "192.168.1.2:11810";
        try {
            // 建立 SequoiaDB 数据库连接

```

```
Sequoiadb sdb = new Sequoiadb(connString, "", "");  
// 获取所有 Collection 信息，并打印出来  
DBCursor cursor = sdb.listCollections();  
while(cursor.hasNext()) {  
    System.out.println(cursor.getCurrent());  
}  
} catch (BaseException e) {  
    System.out.println("Sequoiadb driver error, error description: " + e.getErrorType());  
}  
}
```



注：

- 1) 本例程连接到本地数据库的11810端口是协调节点的服务端口，使用的是空的用户名和密码。用户需要根据自己的实际情况配置参数。
 - 2) SequoiaDB 类为非线程安全类，每个线程必须单独建立自己的 SequoiaDB 对象，不能传递给多个线程同时操作。

• 插入数据

```
String connString = "192.168.1.2:11810";
try {
    Sequoiadb sdb = new Sequoiadb(connString, "", "");
    CollectionSpace db = sdb.createCollectionSpace("space");
    DBCollection cl = db.createCollection("collection");

    // 创建一个插入的 bson 对象
    BSONObject obj = new BasicBSONObject();
    obj.put("name", "tom");
    obj.put("age", 24);

    cl.insert(obj);
}

} catch (BaseException e) {
    System.out.println("Sequoiadb driver error, error description: " + e.getErrorType());
}
```



注：本例程连接到本地数据库的11810端口是协调节点的服务端口，使用的是空的用户名和密码。用户需要根据自己的实际情况配置参数。

• 查询数据

```
// 定义一个游标对象
DBCursor cursor;

BSONObject queryCondition = new BasicBSONObject();
queryCondition = (BSONObject) JSON.parse("({age: {$ne: 20}})");
// 查询所有记录，并把查询结果放在游标对象中
cursor = cl.query(queryCondition, null, null, null);
// 从游标中显示所有记录
while (cursor.hasNext()) {
    BSONObject record = cursor.getNext();
    String name = (String) record.get("name");
    System.out.println("name=" + name);
}
```



注:

- 1) 此示例中设置了简单的查询条件，实际上还可以设置筛选条件，排序情况，及仅使用默认索引等选项。
 - 2) 游标对象将数据表中的部分数据缓存在本地进程的内存中，如果本地数据读取完了，游标对象会通过网络从服务器再次获取部分数据缓存在本地。

集群操作

- 创建分区组

```
String connString = "192.168.1.2:11810";
try {
    Sequoiadb sdb = new Sequoiadb(connString, "", "");
    ReplicaGroup rg = sdb.createRG("group1");
    rg.createNode("dbserver-1", 11820, "/opt/sequoiadb/database/data/11820", null);
    rg.start();
} catch (BaseException e) {
    System.out.println("Sequoiadb driver error, error description" + e.getErrorType());
}
```

注:



1) rg.createNode() 方法的第一个参数为新增节点所在的主机名，注意这里必须是主机名（暂时不支持使用 IP 地址），第三个参数为数据文件存放路径，SequoiaDB 将自动新建该目录，但需要确保 SequoiaDB 管理员用户（默认 sdbadmin ）有写权限。

2) rg.start() 方法用于启动一个分区组的所有节点，该函数一般需要等待10秒钟左右才可完成。该方法不保证分区组选举完成，为了保证分区组可以正常使用，start 完成后，还需要等待30秒时间才可以正常使用新建的分区组。

- 在分区组增加节点

```
String connString = "192.168.1.2:11810";
try {
    Sequoiadb sdb = new Sequoiadb(connString, "", "");
    ReplicaGroup rg = sdb.getReplicaGroup("group1");
    Node node = rg.createNode("dbserver-1", 11830, "/var/sequoiadb/database/data/11830",
        null);
    node.start();
} catch (BaseException e) {
    System.out.println("Sequoiadb driver error, error description" + e.getErrorType());
}
```

注:



rg.createNode() 方法的第一个参数为新增节点所在的主机名，注意这里必须是主机名（暂时不支持使用IP地址），第三个参数为数据文件存放路径，SequoiaDB将自动新建该目录，但需要确保SequoiaDB管理员用户（默认sdbadmin）有写权限。

Java BSON 使用

Java BSON 数据类型

目前，SequoiaDB 支持多种 BSON 数据类型。详情请查看[数据库概念 - 数据库 - 文档](#)一节。

Java 构造 BSON 数据类型

- 整数/符浮点数

Java BSON 构造整数/符浮点数类型

```
// {a: 123, b: 3.14}
BasicDBObject obj = new BasicDBObject();
obj.put("a", 123);
obj.put("b", 3.14);
```

- 字符串

Java BSON 构造字符串类型

```
// {a: "hi"}
```

```
BSONObject obj = new BasicBSONObject();
obj.put("a", "hi");
```

- 空类型

Java BSON 构造空类型

```
// {a: null}
BSONObject obj = new BasicBSONObject();
obj.put("a", null);
```

- 对象

Java BSON 构造嵌套对象类型

```
// {b: {a: 1}}
BSONObject subObj = new BasicBSONObject();
subObj.put("a", 1);
BSONObject obj = new BasicBSONObject();
obj.put("b", subObj);
```

- 数组

Java BSON 使用 org.bson.types.BasicBSONList 来构造数组类型

```
// {a: [0, 1, 2]}
BSONObject obj = new BasicBSONObject();
BSONObject arr = new BasicBSONList();
arr.put("0", 0);
arr.put("1", 1);
arr.put("2", 2);
obj.put("a", arr);
```

- 布尔

Java BSON 构造布尔类型

```
// {a: true, b: false}
BSONObject obj = new BasicBSONObject();
obj.put("a", true);
obj.put("b", false);
```

- 对象 ID

Java BSON 使用 org.bson.types.ObjectId 来生成每条记录的“_id”字段内容。Java BSON 12 字节的 ObjectId 与[数据库概念 - 数据库 - 文档 - 对象 ID](#)一节介绍的对象 ID 略有不同，目前，Java ObjectId 的12字节内容由三部分组成：4字节精确到秒的时间戳，4字节系统（物理机）标示，4字节由随机数起始的序列号。默认情况下，数据库为每条记录生成一个字段名为“_id”的唯一对象 ID。

```
BSONObject obj = new BasicBSONObject();
ObjectId id1 = new ObjectId();
ObjectId id2 = new ObjectId("53bb5667c5d061d6f579d0bb");
obj.put("_id", id1);
```

- 正则表达式

Java BSON 使用 java.util.regex.Pattern 来构造正则表达式数据类型。

```
BSONObject matcher = new BasicBSONObject();
Pattern obj = Pattern.compile("^2001", Pattern.CASE_INSENSITIVE);
matcher.put("serial_num", obj);
BSONObject modifier = new BasicBSONObject("$set", new BasicBSONObject("count", 1000));
c1.update(matcher, modifier, null);
```

以上使用正则表达式构造了一个匹配条件，将序列号以“2001”开头的记录的“count”字段内容改为“1000”。

注：以上使用 Patten 构造的 bson matcher，当使用 matcher.toString()，内容为：

```
{ "serial_num" : { "$options" : "i" , "$regex" : "^2001"}}
```

通过以下 bson 构造方式也可以得到相同的内容：

```
BSONObject matcher2 = new BasicBSONObject();
BSONObject obj2 = new BasicBSONObject();
obj2.put("$regex", "^2001");
```

```
obj2.put("$options", "i");
matcher2.put("serial_num", obj2);
```

但是，通过后者构造出的 matcher2 的数据类型是一个普通的对象嵌套类型，而不是正则表达式类型。

- 日期

Java BSON 使用 `java.util.Date` 来构造日期类型。

```
BasicDBObject obj = new BasicDBObject();
Date now = new Date();
obj.put("date", now);
```

- 二进制

Java BSON 使用 `org.bson.types.Binary` 来构造二进制类型。

```
BasicDBObject obj = new BasicDBObject();
String str = "hello world";
byte[] arr = str.getBytes();
Binary bindata = new Binary(arr);
obj.put("bindata", bindata);
```

- 时间戳

Java BSON 使用 `org.bson.types.BSONTimestamp` 来构造时间戳类型。

```
int sec = 1404189030; // 2014-07-01 12:30:30
BasicDBObject obj = new BasicDBObject();
BSONTimestamp ts = new BSONTimestamp(sec, 0);
obj.put("timestamp", ts);
```

Java Datasource 介绍

Java 驱动的 Datasource 提供给用户一个快速获取有效连接实例的途径。

连接池用法

使用类 `SequoiadbDatasource` 的 `getConnection` 方法从连接池中获取一个连接，使用 `close` 方法把取出的连接放回连接池。当连接池使用的连接数到达连接上限时，下一个请求连接的操作将会等待一段时间，若在规定的时间内无空闲的连接可用，将抛出异常。类 `ConfigOptions` 可以设置连接的各项参数。类 `SequoiadbOption` 中可以设置连接池的各种参数。

详情请查看相关API介绍。

例子

```
SequoiadbDatasource ds = null;
Sequoiadb db = null;
ArrayList<String> urls = new ArrayList<String>();
ConfigOptions nwOpt = new ConfigOptions();           // 定义连接选项
SequoiadbOption dsOpt = new SequoiadbOption();        // 定义连接池选项

urls.add("ubuntu-dev1:11810");
urls.add("ubuntu-dev2:11810");
urls.add("ubuntu-dev3:11810");

nwOpt.setConnectTimeout(500);                         // 设置若连接失败，超时时间 (ms)
nwOpt.setMaxAutoConnectRetryTime(0);                  // 设置若连接失败，重试次数

// 以下设置的都是 SequoiadbOption 的默认值
dsOpt.setMaxConnectionNum(500);                      // 设置连接池最大连接数
dsOpt.setInitConnectionNum(10);                       // 初始化连接池时，创建连接的数量
dsOpt.setDeltaIncCount(10);                          // 当池中没有可用连接时，增加连接的数量
dsOpt.setMaxIdleNum(10);                            // 周期清理多余的空闲连接时，应保留连接的数量
dsOpt.setTimeout(5 * 1000);                           // 当已使用的连接数到达设置的最大连接数时 (500)，请求连接的等待时间。
```

```

dsOpt.setAbandonTime(10 * 60 * 1000); // 连接存活时间，当连接空闲时间超过连接存活时间，将被连接池丢弃
dsOpt.setRecheckCyclePeriod(1 * 60 * 1000); // 清除多余空闲连接的周期
dsOpt.setRecaptureConnPeriod(10 * 60 * 1000); // 检测并取回异常地址的周期

ds = new SequoiadbDatasource(urls, "", "", nwOpt, dsOpt); // 创建连接池
db = ds.getConnection(); // 从连接池获取连接
// do something else // 使用连接进行业务操作
ds.close(db);

```

SQL to SequoiaDB shell to Java

SequoiaDB 的查询用 json (bson) 对象表示，下表以例子的形式显示了 SQL 语句，SequoiaDB shell 语句和 SequoiaDB Java 驱动程序语法之间的对照。

SQL	SequoiaDB shell	Java Driver
insert into students(a,b) values(1,-1)	db.foo.bar.insert({a:1,b:-1})	bar.insert("{'a':1,'b':-1}")
select a,b from students	db.foo.bar.find(null,{a:"",b:""})	bar.query("", {"a":"","b":""}, "", "")
select * from students	db.foo.bar.find()	bar.query()
select * from students where age=20	db.foo.bar.find({age:20})	bar.query("{'age':20}", "", "", "")
select * from students where age=20 order by name	db.foo.bar.find({age:20}).sort({name:1})	bar.query("{'age':20}", "", {"name":1}, "")
select * from students where age>20 and age<30	db.foo.bar.find({age:{\$gt:20,\$lt:30}})	bar.query("{'age':{'\$gt':20,'\$lt':30}}", "", "", "")
create index testIndex on students(name)	db.foo.bar.createIndex("testIndex", {name:1}, false)	bar.createIndex("testIndex", {"name":1}, false, false)
select * from students limit 20 skip 10	db.foo.bar.find().limit(20).skip(10)	bar.query("", "", "", "", 10, 20)
select count(*) from students where age>20	db.foo.bar.find({age:{\$gt:20}}).count()	bar.getCount("{'age':{'\$gt:20}}")
update students set a=a+2 where b=-1	db.foo.bar.update({\$set:{a:2},{b:-1}})	bar.update("{'b':-1}", "{\$inc:{a:2}}", "")
delete from students where a=1	db.foo.bar.remove({a:1})	bar.delete("{'a':1}")

Java API

此部分是相关 Java 的 API 文档。

Java API

历史更新情况：

Version 1.10

1. DBCollection 类新添加的接口：

```

createLob, 创建一个大对象
openLob, 打开一个已存在的大对象
removeLob, 删除一个大对象
listLobs, 列出所有大对象
explain, 获取执行访问计划

```

2. 新增大对象类 DBlob，用于操作大对象：

```

write, 向一个大对象写入数据
read, 从大对象中读取数据
seek, 指定读取数据的偏移
close, 关闭一个大对象
getID, 获取大对象的标识ID
getSize, 获取大对象的大小
getCreateTime, 获取大对象的创建时间

```

Version 1.8

1. Sequoiadb 类新添加的接口：

```

isValid, 判断当前连接是否有效
createCollectionSpace, 提供一个 BSONObject 的选项，使创建集合空间更加灵活
backupOffline, 离线备份支持更多的选项
evalJS, 执行 js 代码

```

```

createDomain, 创建域
getDomain, 获取域
dropDomain, 删除域
isDomainExist, 域是否存在
listDomain, 列出所有域

```

2. DBCollection 类新添加的接口：

```

alterCollection, 修改集合(表)属性
setMainKeys, 设置主键。此接口只与 save 接口配合使用，它设置的主键并不对其他接口起作用
save, 可使用默认的主键"_id"或者指定其他主键，同时插入或更新多条记录

```

3. 添加 Domain 类用于与域相关的操作

4. SequoiadbDatasource类新添加的接口：

```

SequoiadbDatasource, 可提供多个地址的构造器，便于机器负载均衡
getIdleConnNum, 获取当前可用的连接数量
getUsedConnNum, 获取当前已使用的连接数量
getNormalAddrNum, 获取当前正常的地址数量
getAbnormalAddrNum, 获取当前异常的地址数量

```

5. SequoiadbOption 类新添加接口：

```

setRecaptureConnPeriod, 设置周期检测异常地址是否重新可用的时间
getRecaptureConnPeriod, 获取周期检测异常地址是否重新可用的时间

```

Version 1.6

1. 添加类 Node 来取代原来的类 ReplicaGroup。类 ReplicaNode 以及与它们相关的方法将在 version 2.x 中被弃用。

详情请查看相关 API。

PHP 驱动

本节介绍PHP的相关驱动信息。

PHP 驱动

[PHP 开发环境搭建](#)

[PHP 开发基础](#)

[SQL to SequoiaDB shell to PHP](#)

[PHP API](#)

PHP 驱动

概述

SequoiaDB PHP 驱动提供了数据库操作和集群操作的 PHP 接口。数据库操作包括数据库的连接，用户的创建删除，数据的增删改查，索引的创建删除，快照的获取与重置，以及集合与集合空间的创建删除操作等操作。集群操作包括管理分区组和数据节点的各种操作，譬如启动，停止分区组，启动，停止数据节点，获取主从数据节点，集合分区等。

PHP 驱动的有两种类实例。一种用于数据库操作，另一种用于集群操作。

- 数据库操作实例

SequoiaDB 数据库中的数据存放分为三个级别：

1) 数据库

2) 集合空间

3) 集合

因此，在数据库操作中，可用3个类来分别表示连接，集合空间，集合实例，另1个类表示游标实例：

SequoiaDB	数据库类
SequoiaCS	集合空间类
SequoiaCollection	集合类
SequoiaCursor	游标类

PHP 驱动需要使用不同的实例进行操作。譬如读取数据的操作需要游标实例，而创建表空间则需要数据库实例。

- 集群操作实例

SequoiaDB 数据库中的集群操作分为两个级别：1) 分区组 2) 数据节点

 注：分区组包三种类型：协调分区组，编目分区组，数据分区组。

分区组实例和数据节点实例可以用以下两种类的实例表示。

SequoiaGroup	分区组类	分区组实例代表一个单独的分区组
SequoiaNode	数据节点类	数据节点实例代表一个单独的数据节点

与集群相关的操作需要使用分区组及数据节点实例。

SequoiaGroup 的实例用于管理分区组。其操作包括启动，停止分区组，获取分区组中节点的状态，名称信息，数目信息。

SequoiaNode 的实例用于管理节点。其操作包括启动，停止指定的节点，获取指定节点实例，获取主从节点实例，获取数据节点地址信息。

错误信息

- 一个函数被成功调用则返回 true（或整型1），否则返回值为 false（或整型0）
- 如果用户需要知道详细的错误信息，可以调用 getError() 获取错误信息，如果没有错误，则会输出“No error message”

PHP 开发环境搭建

获取驱动开发包

从 <http://www.sequoiadb.com> 下载对应操作系统版本的 SequoiaDB 驱动开发包；解压驱动开发包，从 driver/lib/phpliblibsdbphp-x.x.x.so (x.x.x 为版本号，请根据 PHP 版本选择，前两位需相同版本，第三位版本要小于或等于 PHP 的版本) 文件。

配置开发环境

- Linux

准备工作：安装 Apache 和 PHP 环境，PHP 要求5.3.3及以上版本

配置步骤：

- 打开 /etc/php5/apache2/php.ini 文件；

- 在该文件的 [PHP] 配置段中新增如下行：

```
extension=<PATH>/libsdbphp-x.x.x.so
```

其中 PATH 为 libsdbphp-x.x.x.so 文件放置路径。

- 保存关闭文件；

- 重新启动 apache2 服务；

```
service apache2 restart (SUSE/Redhat) 或 service httpd restart (CentOS)
```

5. 编写包含如下内容 PHP 测试脚本，包存为 test.php 文件，并放在在 Web 服务目录下；

```
<?php phpinfo(); ?>
```

6. 通过浏览器打开 <http://localhost/test.php>，在打开的页面中查看是否包含 SequoiaDB 模块。

- Windows

暂未提供 Windows 驱动开发包

PHP 开发基础

这里介绍如何使用 PHP 驱动接口编写使用 SequoiaDB 数据库的程序。为了简单起见，下面的示例不全部是完整的代码，只起示例性作用。可到安装目录 /client/samples/php 下获取相应的完整的代码。更多查看 [PHP API](#)

数据操作

- 连接数据库

```
//创建 SequoiaDB 对象
$db = new Sequoiadb();
//连接数据库
$array = $db -> connect("localhost:11810");
//检验连接结果，返回的默认是 php 数组类型，数据是 array(0) {"errno">>0}
//如果 errno 为0，那么连接成功
if($array['errno'] !=0 )
{
    exit();
}
```

- 选择集合空间

```
//选择名称为"foo"的集合空间，如果不存在，则自动创建
//返回 SequoiaCS 对象
$cs = $db -> selectCs("foo");
//检验结果，如果成功返回对象，失败返回 NULL
if( empty($cs) )
{
    exit();
}
```

- 选择集合

```
//选择名称为"big"的集合，如果不存在，则自动创建
//返回 SequoiaCollection 对象
$c1 = $cs -> selectCollection("big");
//检验结果，如果成功返回对象，失败返回 NULL
if( empty($c1) )
{
    exit();
}
```

- 插入

```
//插入 json
$arr = $c1 -> insert(" {test: 1} ");
//检测结果
if($array['errno'] !=0 )
{
    exit();
}
//插入数组
$arr = $c1 -> insert(array("test">>=2));
//检测结果
if($arr['errno'] !=0 )
{
    exit();
}
```

- **查询**

```
//查询集合中的所有记录
//返回 SequoiaCursor 对象
$cursor = $c1 -> find();
//遍历所有记录
while($record = $cursor -> getNext())
{
    var_dump($record);
}
```

- **更新**

```
//修改集合中的多条记录，把字段 test 的值修改为0
$arr = $c1 -> update(" {$set: {test: 0}} ");
//检测结果
if($arr['errno'] !=0 )
{
    exit();
}
```

- **删除**

```
//删除集合中的所有记录
$arr = $c1 -> remove();
//检测结果
if($arr['errno'] !=0 )
{
    exit();
}
```

集群操作

- **选择组**

```
//选择名称为"group"的组，如果不存在，则自动创建
//返回 SequoiaGroup 对象
$group = $db -> selectGroup("group");
//检验结果，如果成功返回对象，失败返回 NULL
if( empty($group) )
{
    exit();
}
```

- **启动分区组**

```
//启动分区组，首次会自动激活
//返回操作信息
$arr = $group -> start() ;
//检查结果
If ( $arr['errno'] != 0 )
{
    Exit();
}
```

- **选择节点**

```
//获取名称为"node"的节点
//返回 SequoiaNode 对象
$node = $group -> getNode('node') ;
//检查对象是否空
If ( empty( $node ) )
{
    Exit();
}
```

SQL to SequoiaDB shell to PHP

SequoiaDB 的查询用 json (bson) 对象表示，下表以例子的形式显示了 SQL 语句，SequoiaDB shell 语句和 SequoiaDB PHP 驱动程序语法之间的对照。

SQL	SequoiaDB shell	PHP Driver
insert into students(a,b) values(1,-1)	db.foo.bar.insert({a:1,b:-1})	\$bar->insert(" {a:1,b:-1}")
select a,b from students	db.foo.bar.find(null,{a:"",b:""})	\$bar->find(NULL,'a:','');
select * from students	db.foo.bar.find()	\$bar->find()
select * from students where age=20	db.foo.bar.find({age:20})	\$bar->find("{age:20}")
select * from students where age=20 order by name	db.foo.bar.find({age:20}).sort({name:1})	\$bar->find("{age:20}",NULL,"{name:1}")
select * from students where age>20 and age<30	db.foo.bar.find({age:{\$gt:20,\$lt:30}})	\$bar->find("{age:{\$gt:20,\$lt:30}}")
create index testIndex on students(name)	db.foo.bar.createIndex("testIndex", {name:1},false)	\$bar->createIndex("{name:1}","testIndex",false)
select * from students limit 20 skip 10	db.foo.bar.find().limit(20).skip(10)	\$bar->find(NULL,NULL,NULL,NULL,10,20)
select count(*) from students where age>20	db.foo.bar.find({age:{\$gt:20}}).count()	\$bar->count("{age:{\$gt:20}}")
update students set a=a+2 where b=-1	db.foo.bar.update({\$set:{a:2},{b:-1}})	\$bar->update("{\$set:{a:2}}", "{b:-1}")
delete from students where a=1	db.foo.bar.remove({a:1})	\$bar->remove("{a:1}")

PHP API

此部分是相关 PHP 的 API 文档。

PHP API

C# 驱动

本节介绍 C# 的相关驱动信息。

C# 驱动

C# 开发环境搭建

C# 开发基础

SQL to SequoiaDB shell to C#

C# API

C# 驱动

概述

SequoiaDB C# 驱动提供了数据库操作和集群操作的接口。数据库操作包括数据库的连接，用户的创建删除，数据的增删改查，索引的创建删除，快照的获取与重置，以及集合与集合空间的创建删除操作等操作。集群操作包括管理分区组和数据节点的各种操作，譬如启动，停止分区组，启动，停止数据节点，获取主从数据节点，集合分区等。

C# 类实例

C# 驱动的有两种类实例。一种用于数据库操作，另一种用于集群操作。

- 数据库操作实例

SequoiaDB 数据库中的数据存放分为三个级别：

1) 数据库

2) 集合空间

3) 集合

因此，在数据库操作中，可用3个类来分别表示连接，集合空间，集合实例，另1个类表示游标实例：

SequoiaDB	数据库实例	代表一个单独的数据库连接
CollectionSpace	集合空间实例	代表一个单独的集合空间
DBCollection	集合实例	代表一个单独的集合
DBCursor	游标实例	代表一个查询产生的结果集

C# 驱动需要使用不同的实例进行操作。譬如读取数据的操作需要游标实例，而创建表空间则需要数据库实例。

- 集群操作实例

SequoiaDB 数据库中的集群操作分为两个级别：1) 分区组 2) 数据节点

注：分区组包含三种类型：协调分区组，编目分区组，数据分区组。

分区组实例和数据节点实例可以用以下两种类的实例表示。

ReplicaGroup	分区组类	分区组实例代表一个单独的分区组
Node	数据节点类	数据节点实例代表一个单独的数据节点

无疑与集群相关的操作需要使用分区组及数据节点实例。

ReplicaGroup 的实例用于管理分区组。其操作包括启动，停止分区组，获取分区组中节点的状态，名称信息，数目信息。

Node 的实例用于管理节点。其操作包括启动，停止指定的节点，获取指定节点实例，获取主从节点实例，获取数据节点地址信息。

线程安全性

对于每一个连接，其产生的集合空间，集合公用一个套接字。因此在多线程系统中，必须确保每个线程不会同时针对同一套接字，在同一时间发送或接收数据。一般来说，不建议使用多个线程共同操作一个连接实例与其产生的其它实例。如果每个线程使用自己的连接实例以及其它产生的实例，则可以保证线程安全。

错误信息

每一个接口都会抛出 SequoiaDB.BaseException 和 System.Exception 异常，分别对应于数据库引擎返回的异常信息和客户端本地的异常信息，其中 BaseException 的异常信息可以通过该异常类的 ErrorCode，ErrorType 和 Message 属性获得。

C# 开发环境搭建

获取驱动开发包

从 <http://www.sequoiadb.com> 下载对应操作系统版本的 SequoiaDB 驱动开发包；解压驱动开发包，从 driver/CSharp/ 目录中获取 sequoiadb.dll 链接库，然后，在 Visual Studio 中引用该链接库，或者在命令行编译时指定引用该链接库，比如“csc /target:exe /reference:sequoiadb.dll Find.cs Common.cs”，即可使用相关 API。在安装目录下的 smaples\WC# 目录可以找到 C# 驱动的完整示例。

BSON 库 API

SequoiaDB 数据库的 C# 驱动使用了第三方公司 MongoDB 提供的 C# BSON 库，详细介绍可以参照 MongoDB 官方文档：<http://docs.mongodb.org/ecosystem/tutorial/use-csharp-driver/#the-bson-library>

Visual Studio 版本支持

当前版本的 C# 驱动可在以下版本的 Visual Studio 中使用

Visual Studio 2008

Visual Studio 2010

.NET Framework 版本支持

当前版本的 C# 驱动在 .NET Framework3.5 中生成，可在以下版本的 .NET Framework 中使用

.NET Framework 3.5

.NET Framework 4.0

C# 开发基础

这里介绍如何使用 C# 驱动接口编写使用 SequoiaDB 数据库的程序。该文档介绍了 SequoiaDB 数据库 C# 驱动的简单示例，详细的使用规范可参照官方的 [C# API 文档](#)。

命名空间

在使用 C# 驱动的相关 API 之前，你必须在源代码中添加如下的 using 申明：

```
using SequoiaDB;
using SequoiaDB.Bson;
```

数据操作

- 连接数据库和身份验证

若数据库没有创建用户，则可以匿名连接到数据库：

```
string addr = "127.0.0.1:11810";
Sequoiadb sdb = new Sequoiadb(addr);
try
{
    sdb.Connect();
}
catch (BaseException e)
{
    Console.WriteLine("ErrorCode: {0}, ErrorCode: {1}", e.ErrorCode, e.ErrorType);
    Console.WriteLine(e.Message);
}
catch (System.Exception e)
{
    Console.WriteLine(e.StackTrace);
}
```

否则，连接的时候必须指定用户名和密码：

```
string addr = "127.0.0.1:11810";
Sequoiadb sdb = new Sequoiadb(addr);
try
{
    sdb.Connect("testusr", "testpwd");
}
catch (BaseException e)
{
    Console.WriteLine("ErrorCode: {0}, ErrorCode: {1}", e.ErrorCode, e.ErrorType);
    Console.WriteLine(e.Message);
}
catch (System.Exception e)
{
    Console.WriteLine(e.StackTrace);
}
```

这里给出了异常信息的 try 和 catch 块，下面的所有操作都会抛出同样的异常信息，因此不再给出相关的 try 和 catch 块。

- 断开与数据库连接

```
// do not forget to disconnect from sdb
sdb.Disconnect();
```

- 得到或创建集合空间和集合

根据名字，得到对应的 CollectionSpace，如果不存在，则创建：

```
// create collectionspace, if collectionspace exists get it
string csName = "TestCS";
```

```
CollectionSpace cs = sdb.GetCollectionSpace(csName);
if (cs == null)
    cs = sdb.CreateCollectionSpace(csName);
// or sdb.CreateCollectionSpace(csName, pageSize), need to specify the pageSize
```

根据名字，得到对应的 Collection，如果不存在，则创建：

```
// create collection, if collection exists get it
string c1Name = "TestCL";
DBCollection dbc = cs.GetCollection(c1Name);
if (dbc == null)
    dbc = cs.CreateCollection(c1Name);
//or cs.createCollection(collectionName, options), create collection with some options
```

- 对 Collection 进行插入操作

创建需要插入的数据 BsonDocument 并插入：

```
BsonDocument insertor = new BsonDocument();
string date = DateTime.Now.ToString();
insertor.Add("operation", "Insert");
insertor.Add("date", date);
ObjectId id = dbc.Insert(insertor);
```

当然，BsonDocument 中还可以嵌套 BsonDocument 对象；而且你还可以直接 new 一个完整的 BsonDocument，而不需要通过 Add 方法：

```
BsonDocument insertor = new BsonDocument
{
    {"FirstName", "John"},
    {"LastName", "Smith"},
    {"Age", 50},
    {"id", i},
    {"Address",
        new BsonDocument
        {
            {"StreetAddress", "212ndStreet"},
            {"City", "NewYork"},
            {"State", "NY"},
            {"PostalCode", "10021"}
        }
    },
    {"PhoneNumber",
        new BsonDocument
        {
            {"Type", "Home"},
            {"Number", "212555-1234"}
        }
    }
};
```

插入多条数据：

```
//bulkinsert
List<BsonDocument> insertor=new List <BsonDocument> ();
for (int i=0; i<10; i++)
{
    BsonDocument obj=new BsonDocument();
    obj.Add("operation", "BulkInsert");
    obj.Add("date", DateTime.Now.ToString());
    insertor.Add(obj);
}
dbc.BulkInsert(insertor, 0);
```

- 索引的相关操作

创建索引：

```
//createindexkey, indexonattribute' Id' byASC (1) /DESC (-1)
BsonDocument key = new BsonDocument();
```

```
key.Add("id", 1);
string name = "index name";
bool isUnique = true;
bool isEnforced = true;
dbc.CreateIndex(name, key, isUnique, isEnforced);
```

删除索引：

```
string name = "index name";
dbc DropIndex(name);
```

- **查询操作**

进行查询操作，需要使用游标对查询结果进行遍历，而且可以先得到当前 Collection 的索引，如果不为空，可作为制定访问计划 (hint) 用于查询：

```
DBCursor icursor = dbc.GetIndex(name);
BsonDocument index = icursor.Current();
```

构建相应的 BsonDocument 对象用于查询，包括：查询匹配规则 (matcher，包含相应的查询条件)，域选择 (selector)，排序规则 (orderBy，增序或降序)，制定访问计划 (hint)，跳过记录个数 (0)，返回记录个数 (-1：返回所有数据)。查询后，得到对应的 Cursor，用于遍历查询得到的结果：

```
BsonDocument matcher = new BsonDocument();
BsonDocument conditon = new BsonDocument();
conditon.Add("$gte", 0);
conditon.Add("$lte", 9);
matcher.Add("id", conditon);
BsonDocument selector = new BsonDocument();
selector.Add("id", null);
selector.Add("Age", null);
BsonDocument orderBy = new BsonDocument();
orderBy.Add("id", -1);
BsonDocument hint = null;
if (index != null)
    hint = index;
else
    hint = new BsonDocument();
DBCursor cursor = dbc.Query(matcher, selector, orderBy, hint, 0, -1);
```

使用 DBCursor 游标进行遍历：

```
while (cursor.Next() != null)
Console.WriteLine(cursor.Current());
```

- **删除操作**

构建相应的 BsonDocument 对象，用于设置删除的条件：

```
//createtheDeletecondition
BsonDocument drop = new BsonDocument();
drop.Add("Last Name", "Smith");
coll.Delete(drop);
```

- **更新操作**

构建相应的 BsonDocument 对象，用于设置更新条件，你还可以创建 DBQuery 对象封装所有的查询或更新规则：

```
DBQuery query = new DBQuery();
BsonDocument upater = new BsonDocument();
BsonDocument matcher = new BsonDocument();
BsonDocument modifier = new BsonDocument();
upater.Add("Age", 25);
modifier.Add("$set", upater);
matcher.Add("First Name", "John");
query.Matcher = matcher;
query.Modifier = modifier;
dbc.Update(query);
```

更新操作，如果没有满足 matcher 的条件，则插入此记录：

```
dbc.Upsert(query);
```

SQL to SequoiaDB shell to C#

SequoiaDB 的查询用 json (bson) 对象表示，下表以例子的形式显示了 SQL 语句，SequoiaDB shell 语句和 SequoiaDB C# 驱动程序语法之间的对照。

SQL	SequoiaDB shell	C# Driver
insert into students(a,b) values(1,-1)	db.foo.bar.insert({a:1,b:-1})	bar.insert("{'a':1,'b':-1}")
select a,b from students	db.foo.bar.find(null,{a:"",b:""})	bar.query("", {"a":"","b":""}, "", "")
select * from students	db.foo.bar.find()	bar.query()
select * from students where age=20	db.foo.bar.find({age:20})	bar.query("{'age':20}", "", "", "")
select * from students where age=20 order by name	db.foo.bar.find({age:20}).sort({name:1})	bar.query("{'age':20}", "", {"name":1}, "")
select * from students where age>20 and age<30	db.foo.bar.find({age:{\$gt:20,\$lt:30}})	bar.query("{'age':{'\$gt':20,'\$lt':30}}", "", "", "")
create index testIndex on students(name)	db.foo.bar.createIndex("testIndex", {name:1},false)	bar.createIndex("testIndex", {"name":1}, false, false)
select * from students limit 20 skip 10	db.foo.bar.find().limit(20).skip(10)	bar.query("", "", "", 10, 20)
select count(*) from students where age>20	db.foo.bar.find({age:{\$gt:20}}).count()	bar.getCount("{'age':{'\$gt:20}}")
update students set a=a+2 where b=-1	db.foo.bar.update({\$set:{a:2},{b:-1}})	bar.update("{'b':-1}", {"\$inc": {"a":2}}, "")
delete from students where a=1	db.foo.bar.remove({a:1})	bar.delete("{'a':1}")

C# API

此部分是相关 C# 的 API 文档。

[C# API](#)

历史更新情况：

Python 驱动

本节介绍 Python 的相关驱动信息。

[Python 驱动](#)

[Python 开发环境搭建](#)

[Python 开发基础](#)

[SQL to SequoiaDB shell to Python](#)

[Python API](#)

Python 驱动

概述

Python 客户端驱动提供了数据库操作和集群操作的接口。数据库操作包括数据库的连接，用户的创建删除，数据的增删改查，索引的创建删除，快照的获取与重置，以及集合与集合空间的创建删除操作等操作。集群操作包括管理分区组和数据节点的各种操作，譬如启动、停止分区组，启动、停止数据节点，获取主从数据节点，集合分区等。更多参考 [Python 在线 API](#)

Python 类实例

Python 客户端驱动的有两种类实例。一种用于数据库操作，另一种用于集群操作。

- [数据库操作实例](#)

SequoiaDB 数据库中的数据存放分为三个级别：

1) 数据库

2) 集合空间

3) 集合

因此，在数据库操作中，可用3个类来分别表示连接，集合空间，集合实例，另1个类表示游标实例：

client	数据库类	连接实例代表一个单独的数据库连接
collectionspace	集合空间类	集合空间实例代表一个单独的集合空间
collection	集合类	集合实例代表一个单独的集合
cursor	游标类	游标实例代表一个查询产生的游标

Python 客户端需要使用不同的实例进行操作。譬如读取数据的操作需要游标实例，而创建表空间则需要数据库实例。

- 集群操作实例

SequoiaDB数据库中的集群操作分为两个级别：1) 分区组 2) 数据节点

 注：分区组包三种类型：编目分区组，数据分区组。

分区组实例和数据节点实例可以用以下两种类的实例表示。

replicagroup	分区组类	分区组实例代表一个单独的分区组
replicaode	数据节点类	数据节点实例代表一个单独的数据节点

与集群相关的操作需要使用分区组及数据节点实例。

replicagroup 的实例用于管理分区组。其操作包括启动，停止分区组，获取分区组中节点的状态，名称信息，数目信息。

replicanode 的实例用于管理节点。其操作包括启动，停止指定的节点，获取指定节点实例，获取主从节点实例，获取数据节点地址信息。

错误信息

每个函数都有返回值，返回值的定义如下：

SDB_OK (数据值为0) : 表示执行成功；

< 0 : 表示数据库错误，具体的错误描述在 err.prop 文件中可以找到，也可以用 pysequoiadb.getErr(error_no) 获取；

> 0 : 表示系统错误，请查阅相关系统的错误码信息。

Python 开发环境搭建

获取驱动开发包

从 <http://www.sequoiadb.com> 下载对应操作系统版本的 SequoiaDB 驱动开发包。

配置开发环境

- Linux

1. 解压下来的驱动开发包；得到 python 目录。

2. 将 python 目录下的 bson 和 pysequoiadb 目录拷贝到开发工程目录中（建议放在第三方库目录下）。

- Windows

暂未推出 Windows 驱动开发包。

Python 开发基础

本节介绍使用 Python 运行 SequoiaDB。首先安装 SequoiaDB，安装信息请查看 [SequoiaDB 服务器安装章节](#)。

这里介绍如何使用 Python 客户端驱动接口编写使用 SequoiaDB 数据库的程序。为了简单起见，下面的示例不全部是完整的代码，只起示例性作用。可到 /sequoiadb/client/samples/python 下获取相应的完整的代码。更多查看 [Python API](#)

数据库操作

- **数据库连接 (Connecting)**

connect.py 演示如何连接到数据库。文件应当 import “pysequoiadb”中的 client , const 等模块，以及 error 模块中的 SequoiaDBError 类。

```
import pysequoiadb
from pysequoiadb import client
from pysequoiadb import const
from pysequoiadb.error import SequoiaDBError

# connect to local db, using default args value.
# host= 'localhost', port= 11810, user= '', password= ''
try:
    db = client()
except DBBaseError, e:
    pysequoiadb._print(e)
    del db
    exit()

# if no error occurs, connect to specified server successfully
print 'Connect success'
db.disconnect()
# Need to release client whether it connected db server successfully or not
del db
```

在 Linux 下，可以直接运行 python 解释执行 connect.py。



注:

本例程连接到本地数据库的11810端口，使用的是空的用户名和密码。用户需要根据自己的实际情况配置参数。譬如，将上述代码中的 db = client() 修改为 db = client('192.168.10.188', 11810)。当数据库已经创建用户时，应该使用正确的用户及密码连接到数据库，否则连接失败。

- **创建集合空间和集合**

以下创建了一个名字为“foo”的集合空间和一个名字为“bar”的集合，集合空间内的集合的数据页大小为16k。可根据实际情况选择不同的数据页。创建集合后，可对集合做增删改查等操作。

```
# 连接到数据库
try:
    db = client()
except SDBBaseError, e:
    pysequoiadb._print(e)

# success to connect to db
try:
    cs_name = 'foo'
    cs = db.create_collection_space(cs_name, 16384) except SDBBaseError, e:
    pysequoiadb._print(e)

# success to create collection space
cl_name = 'bar'
try:
    cs = cs.create_collection(cl_name)
```

```
except SDBBaseError, e:
    pysequoiadb._print(e)
• 插入数据 (insert)
```

```
# 首先，需要创建一个插入的 dict 对象。
record = {"name": "Tom", "age": 24}
# 接着，把此 dict 对象插入集合中
oid = cl.insert ( record ) ;
```

record 为输入参数，为要插入的数据。dict 对象将会被转换成 bson 插入到集合中。oid 是插入该记录，返回的 bson 结构的 objectid。

- 查询 (query)

```
# 查询所有记录，把结果放入游标中，并循环打印游标中的每条记录
try:
    cr = cl.query()

    while True:
        try:
            record = cr.next()
        except SDBEndOfCursor:
            break
        except SDBBaseError, e:
            pysequoiadb._print(e)
```

查询操作需要一个游标对象存放查询的结果到本地。要获得查询的结果需要使用游标操作。本例使用了游标操作的 next 接口，表示从查询结果中取到一条记录。此示例中没有设置查询条件，筛选条件，排序情况，及仅使用默认索引。

- 索引 (index)

```
index_name = "index_name"
# 首先创建一个 dict 对象包含将要创建的索引的信息
idx = { 'name':1, 'age':-1 }
# 创建一个以"name"为升序，"age"为降序的索引
cl.create_index ( idx, index_name, FALSE, FALSE ) ;
```

集合对象 collection 中创建一个以“name”为升序，“age”为降序的索引。

- 更新 (update)

```
# 先创建一个包含更新规则的 BSONObj 对象
rule = {"$set": { "age":19}}
# 打印出更新规则
print rule
# 更新记录
cl.update( rule )
```

在集合对象 collection 中更新了记录。实例中没有指定数据匹配规则，所以此示例将更新集合中所有的集合。

集群操作

- 分区组操作

分区组操作包括创建分区组 (client::creat_replica_group)，得到分区组实例 (client::get_replica_group_by_name 和 client::get_replica_group_by_id)，启动分区组所有数据节点 (replicagroup::start)，停止分区组所有数据节点 (replicagroup::stop) 等。

以下为分区组操作示例性的例子。真正的应用应包括错误检测等。

```
# 定义一个空的 map 对象表示创建数据节点没有更多的配置内容
config = {}
...
# 先建立一个编目分区组
```

```

db.create_cata_replica_group ( HOST_NAME, SERVICE_NAME, CATALOG_GROUP_PATH , None)

# 创建数据分区组
rg = db.create_replica_group ( REPLICA_GROUP_NAME)

# 创建第一个数据节点
rg.create_node ( HOST_NAME1, SERVICE_NAME1, DATABASE_PATH1, config )
...
# 启动分区组
rg.start ()

```

- 数据节点操作

数据节点操作包括创建数据节点 (`sdbReplicaGroup::createNode`) , 得到主数据节点 (`sdbReplicaGroup::getMaster`) , 得到从数据节点 (`sdbReplicaGroup::getSlave`) , 启动数据节点 (`sdbNode::start`) , 停止数据节点 (`sdb::Stop`) 等。

以下为数据节点操作示例性的例子。真正的应用应包括错误检测等。

```

# 获取主数据节点
master = rg.get_master() ;

# 获取从数据节点
slave = rg.get_slave() ;

```

SQL to SequoiaDB shell to Python

SequoiaDB 的查询用 `dict (bson)` 对象表示 , 下表以例子的形式显示了 SQL 语句 , SequoiaDB shell 语句和 SequoiaDB Python 驱动程序语法之间的对照。

SQL	SequoiaDB shell	Python Driver
<code>insert into students(a,b) values(1,-1)</code>	<code>db.foo.bar.insert({a:1,b:-1})</code>	<code>cl = collection() obj = { "a":1, "b":-1 } cl.insert(obj)</code>
<code>select a,b from students</code>	<code>db.foo.bar.find(null,{a:"",b:""})</code>	<code>cl = collection() obj = {} selected = { "a":"", "b":"" } cr = cl.query(obj, selected)</code>
<code>select * from students</code>	<code>db.foo.bar.find()</code>	<code>cl = collection() cr = cl.query () ;</code>
<code>select * from students where age=20</code>	<code>db.foo.bar.find({age:20})</code>	<code>cl = collection() condition ={"age":20} cr = cl.query (condition)</code>
<code>select * from students where age=20 order by name</code>	<code>db.foo.bar.find({age:20}).sort({name:1})</code>	<code>cl = collection() condition ={"age":20} orderBy = {"name":1} cr = cl .query (condition , None, orderBy , None)</code>
<code>select * from students where age>20 and age<30</code>	<code>db.foo.bar.find({age:{\$gt:20,\$lt:30}})</code>	<code>cl = collection() condition = {"age": {"\$gt":20,"\$lt":30}} cr = cl .query (condition) ;</code>
<code>create index testIndex on students(name)</code>	<code>db.foo.bar.createIndex("testIndex", {name:1},false)</code>	<code>cl = collection() obj = { "name":1 } cl.create_index (&obj, "testIndex", FALSE, FALSE)</code>
<code>select * from students limit 20 skip 10</code>	<code>db.foo.bar.find().limit(20).skip(10)</code>	<code>cl = collection() cr = cl .query (None, None, None, None, 10, 20) ;</code>
<code>select count(*) from students where age>20</code>	<code>db.foo.bar.find({age:{\$gt:20}}).count()</code>	<code>cl = collection() count = 0L condition = { "age": {"\$gt":20}}</code>

SQL	SequoiaDB shell	Python Driver
update students set a=a+2 where b=-1	db.foo.bar.update({\$set:{a:2}},{b:-1})	cl = collection() condition = { "b":1 } rule = { "\$set":{"a":2} } cl.update (rule, condition, None);
delete from students where a=1	db.foo.bar.remove({a:1})	cl = collection() condition = {"a":1} cl.delete (condition)

Python API

此部分是相关 Python 的 API 文档。

Python API

历史更新情况：

Version 1.10

1. 新增接口类 lob :

```
close, 关闭创建的lob对象, 用以刷新数据
read, 可从lob对象中读取数据
write, 可把数据写入lob
seek, 可跳转到到指定数据位置
get_oid, 可获取lob对象的oid
get_size, 可获取lob对象的大小(bytes)
get_create_time, 可获取lob对象的创建时间
```

2. collection 新增接口 :

```
create_lob, 可在当前的collection中创建一个lob对象
remove_lob, 可在当前的collection中删除指定lob对象
get_lob, 可获取当前collection中指定oid的lob对象
list_lobs, 可列出当前collection中所有的lob
```

详情请查看相关 API。

C BSON 简介

BSON 是 JSON 的二进制表现形式，通过记录每个对象，元素，以及嵌套元素和数组的类型以及长度，能够高速有效地进行某个元素的查找。因此，在 C 和 C++ 中使用 BSON 官方提供的 BSON 接口进行数据存储。更多参考 [C BSON 在线 API](#)。

与普通的 JSON 不同，BSON 提供更多的数据类型，以满足 C/C++ 语言多种多样的需求。SequoiaDB 提供了包括8字节浮点数（DOUBLE），字符串，嵌套对象，嵌套数组，对象 ID（数据库中每个集合中每条记录都有一个唯一 ID），布尔值，日期，NULL，正则表达式，4字节整数（INT），时间戳，以及8字节整数等数据类型。这些类型的定义可以在 bson.h 中的 bson_type 找到。注意：使用 C BSON API 函数在建立 BSON 出错时，将返回错误码，应当适当检测函数返回值。详情请查看 C BSON API。

在用户程序使用 BSON 对象时，主要分为建立对象和读取对象两个操作。

建立对象

总的来说，一个 BSON 对象的创建主要分为三大步操作：

- 1) 创建对象 (bson_create ; bson_init)
- 2) 使用对象
- 3) 清除对象 (bson_dispose ; bson_destory)

- 创建一个简单的 BSON 对象{age:20}。

```
INT32 rc = SDB_OK;
bson obj;
bson_init(&obj);
bson_append_int(obj, "age", 20);
if ( bson_finish(obj) != SDB_OK )
printf("Error. ");
bson_destory(obj);
```

- 创建一个复杂的 BSON 对象

```
/* 创建一个包含 {name: "tom", colors: ["red", "blue", "green"]}, address: {city: "Toronto,
province: "Ontario"} 的对象 */
bson_iterator bi ;
bson *newobj = bson_create () ;
bson_append_string ( newobj, "name", "tom" ) ;
bson_append_start_object ( newobj, "address" ) ;
bson_append_string ( newobj, "city", "Toronto" ) ;
bson_append_string ( newobj, "provice", "Ontario" ) ;
bson_append_start_array(newobj,"colors");
bson_append_string(newobj, "0", "red");
bson_append_string(newobj, "1", "blue");
bson_append_string(newobj, "2", "green");
bson_append_finish_object ( newobj ) ;
if( bson_finish ( newobj ) != BSON_OK )
printf("Error. ");
```

读取对象

读取 BSON 对象使用一个 `bson_iterator`，对一个完整的例子，可以使用 `bson_print_raw` 方法来读取。但是首先得初始化 `bson_iterator` 对象，然后使用 `bson_iterator_next` 遍历每一个元素。

例如：

```
bson_iterator i[1] ;
bson_type type ;
const char * key;

bson_iterator_init(i, newobj) ;

type = bson_iterator_type (i);
key = bson_iterator_key (i);

printf( "Type: %d, Key: %s\n", type, key) ;
```

对于每个 `bson_iterator`，使用 `bson_iterator_type` 函数可以得到其类型，使用 `bson_iterator_string` 等函数可以得到其相对应类型的数值。

```
printf( "Value: %s, bson_iterator_string(i)) ;
```

- 遍历每个连续的 BSON 对象元素，可以使用 `bson_find` 函数直接跳转得到元素的名称。如果该元素不存在于 `bson` 之内，则 `bson_find` 函数返回 `BSON_EOO`。

例如想得到 `name` 元素名可以这样使用：

```
bson_iterator i[1] ,sub[i] ;
bson_type type ;

bson_find ( i, newobj, "name" )
```

- 读取数组元素或嵌套对象，因为“`address`”是一个嵌套对象，需要特殊遍历。首先得到 `address` 值，再初始化一个新的 BSON 迭代器：

```
type = bson_find(i,newobj,"address");
bson_iterator_subiterator(i,sub);
```

方法 `bson_iterator_subiterator` 初始化迭代器 `sub`，并且指向子对象的开始位置，从这里开始可以遍历 `sub` 中的所有元素，直到子对象的结束位置。

C++ BSON 简介

C++ BSON 主要类

C++ BSON 用到4个类：

bson::BSONObj : 创建 BSONObj 对象。

bson::BSONElement : BSONObj 对象由 BSONElement 对象组成，即 BSONElement 对象为 BSONObj 对象的字段或者元素，它是键值对。

bson::BSONObjBuilder : BSONObjBuilder 用来实例化 BSONObj 对象。

bson::BSONObjIterator : BSONObjIterator 用来遍历 BSONObj 对象中的元素。

命名空间 bson 中定义了这些类的类型为：

```
typedef bson::BSONElement be;
```

```
typedef bson::BSONObj bo;
```

```
typedef bson::BSONObjBuilder bob;
```

另外，可以使用 bo::iterator 代替 BSONObjIterator。

建立对象

以下简单介绍如何创建用 CPP BSON 实例。详细内容请查阅 [C++ BSON API](#)

- 使用 BSONObject , BSONObjBuilder 建立对象

```
#include "client.hpp"
...
using namespace bson ;
BSONObj obj ;
BSONObjBuilder b ;
b.append("name", "sam") ;
b.append("age", "24") ;
obj = b.obj() ;
或者
obj = BSONObjBuilder().genOID().append("name", "sam").append("age", 24).obj() ;
```

另外，可以使用数据流的方法建立 BSONObj 对象。

```
BSONObj obj ;
BSONObjBuilder b ;
b<<"name"<<"sam"<<"age"<<"24" ;
obj = b.obj() ;
```

- 使用宏 BSON 建立对象

C++ BSON 中定义还定义了一个 BSON 的宏，可以用它来快速地建立 BSONObj 对象。

```
BSONObj obj ;
// int
obj = BSON( "a" << 1 ) ;
// float
obj = BSON( "b" << 3.14159265359 ) ;
// string
obj = BSON( "foo" << "bar" ) ;
// OID
obj = BSON( GENOID ) ;
// bool
obj = BSON( "flag" << true "ret" << false ) ;
// object
obj = BSON( "d" << BSON("e" << "hi!") ) ;
// array
```

```
obj = BSON( "phone" << BSON_ARRAY( "13800138123" << "13800138124" ) ) ;
// others, less than, greater than, etc
obj = BSON( "g" << LT << 99 ) ;
```

- 使用 fromjson 接口建立对象

此外，可以使用 fromjson.hpp 中的 fromjson() 将 json 字符串转换成 BSONObj 对象。

```
string s (" {name: \"sam\"} ") ;
fromjson ( s, obj ) ;
或者
const char *r ="{
    firstName: \"Sam\",
    lastName: \"Smith\", age: 25, id: \"count\",
    address: {streetAddress: \"25 3ndStreet\",
    city: \"NewYork\", state: \"NY\", postalCode: \"10021\"},
    phoneNumber: [{type: \"home\", number: \"212555-1234\"}]}" ;
fromjson ( r, obj ) ;
```

C BSON API

此部分是相关 C BSON 的 API 文档。

[C BSON API](#)

C++ BSON API

此部分是相关 C++ BSON 的 API 文档。

[C++ BSON API](#)

数据库集群控制器

sdbcm 概述

数据库集群控制器 (SequoiaDB Cluster Manager) 是一个守护进程，在 Windows 中它是以服务的方式常驻系统后台。SequoiaDB 的所有集群管理操作都必须有 sdbcm 的参与，目前每一台物理机器上只能启动一个 sdbcm 进程，负责执行远程的集群管理命令和监控本地的 SequoiaDB 数据库。sdbcm 主要有两大功能：

- 远程启动，关闭，创建和修改节点：通过 SequoiaDB 客户端或者驱动连接数据库时，可以执行启动，关闭，创建和修改节点的操作，该操作向指定节点物理机器上的 sdbcm 发送远程命令，并得到 sdbcm 的执行结果。
- 本地监控：对于通过 sdbcm 启动的节点，都会维护一张节点列表，其中保存了所有本地节点的服务名和启动信息，如启动时间、运行状态等。如果某个节点是非正常终止的，如进程被强制终止，引擎异常退出等，sdbcm 会尝试重启该节点。

sdbcm 操作

- 配置文件

在数据库安装目录的 conf 子目录下，有一个 sdbcm.conf 的配置文件，该文件给出了启动 sdbcm 时的配置信息，如下所示：

参数	描述	示例
defaultPort	sdbcm 的默认监听端口	defaultPort=11790
<hostname>_Port	物理主机 hostname 上 sdbcm 的监听端口。若在该配置文件中找不到对应主机的参数，sdbcm 会以 defaultPort 启动；若 defaultPort 也不存在，则 sdbcm 以默认端口11790启动。	<hostname>_Port=11790
RestartCount	重启次数，即定义 sdbcm 对节点的最大重启次数。该参数不存在时默认置为-1，即不断重启。	RestartCount=5

参数	描述	示例
RestartInterval	重启间隔，即定义 sdbcm 的最大重启间隔，单位是分钟。该参数与 RestartCount 结合定义了重启间隔内 sdbcm 对节点的最大重启次数，超出时则不再重启。该参数不存在时默认置为0，即不考虑重启间隔。	RestartInterval=0

- 启动 sdbcm

运行 sdbcmart 命令可以启动 sdbcm。

- 关闭 sdbcm

运行 sdbcmtop 命令可以关闭 sdbcm。

参考手册

有关 SequoiaDB 数据库的 JavaScript 操作方法和操作符，以及词汇表、错误代码信息和一些命名限定。

SequoiaDB JavaScript 方法

SequoiaDB 是一种面向文档型的非关系型数据库，使用 [JSON](#) 数据模型。

- 如果需要匹配嵌套对象中的元素，可以使用“.”（ ASCII 码 0x2E ）将所有的父元素名与子元素名连接。例如下列对象：

```
{ name: "tom", address: { street: { street1: "1024 Wall Street", street2: "University Drive" }, zipcode: 100000 } }
```

如果需要匹配所有存在 address 元素中的 street 元素里面的 street1 元素，则可以使用：

```
{ "address.street.street1": { $exists : 1 } }
```

- 如果对象中存在数组，对数组中每一个元素的引用使用“0”，“1”，“2”……等标记第一个，第二个，第三个……元素。例如包含数组的对象：

```
{ name: "tom", phone: [ "13175824", 1528345 ] }
```

可以使用下列匹配条件进行数组中元素匹配：

```
{ "phone.0": "13175824" }
```

数据库操作方法

方法名	描述	示例
db.createCS()	创建集合空间	db.createCS("foo")
db.dropCS()	删除集合空间	db.dropCS("foo")
db.getCS()	获得某个集合空间的引用	db.getCS("foo")
db.createRG()	创建一个分区组	db.createRG("rg")
db.removeRG()	删除一个分区组	db.removeRG("rg")
db.getRG()	获得一个分区组的引用	db.getRG("rg")
db.list()	枚举列表	db.list()
db.listCollectionSpaces()	枚举集合空间	db.listCollectionSpaces()
db.listCollections()	枚举集合	db.listCollections()
db.listReplicaGroups()	枚举分区组	db.listReplicaGroups()
db.snapshot()	枚举快照	db.snapshot(0)
db.resetSnapshot()	重置快照	db.resetSnapshot(0)
db.startRG()	启动分区组	db.startRG("rgName")
db.createUsr()	创建数据库用户	db.createUsr("root","admin")
db.dropUsr()	删除数据库用户	db.dropUsr("root","admin")
db.transBegin()	开启事务	db.transBegin()
db.transCommit()	提交事务	db.transCommit()
db.transRollback()	事务回滚	db.transRollback()
db.createProcedure()	创建存储过程函数	db.createProcedure(function sum(i,j){return i+j;})
db.removeProcedure()	删除已存在的存储过程函数	db.removeProcedure("sum")
db.listProcedures()	枚举存储过程函数	db.listProcedures()

方法名	描述	示例
<code>db.eval()</code>	调用存储过程函数或直接使用 JavaScript 语句	<code>db.eval('db.foo.bar')</code>
<code>db.exec()</code>	执行 SQL Select 语句	<code>db.exec("select * from foo.bar")</code>
<code>db.execUpdate()</code>	执行 SQL 其它语句	<code>db.execUpdate("insert into foo.bar(name,age) values('test',20)")</code>
<code>db.backupOffline()</code>	离线备份	<code>db.backupOffline({Name:"backuptest"})</code>
<code>db.listBackup()</code>	查看备份	<code>db.listBackup()</code>
<code>db.removeBackup()</code>	删除备份	<code>db.removeBackup()</code>
<code>db.listTasks()</code>	查看后台任务	<code>db.listTasks()</code>
<code>db.waitTasks()</code>	同步等待后台任务结束或取消	<code>db.waitTasks(taskid)</code>
<code>db.cancelTask()</code>	取消后台任务	<code>db.cancelTask(taskid)</code>
<code>db.flushConfigure()</code>	刷盘配置	<code>db.flushConfigure({Global:true})</code>
<code>db.setSessionAttr()</code>	设置当前会话/连接属性	<code>db.setSessionAttr({PreferredInstance:"M"})</code>
<code>db.createDomain()</code>	创建一个域	<code>db.createDomain('mydomain', ['datagroup1','datagroup2'])</code>
<code>db.listDomains()</code>	枚举域	<code>db.listDomains()</code>
<code>db.dropDomain()</code>	删除指定域	<code>db.dropDomain('mydomain')</code>
<code>db.getDomain()</code>	获取指定域	<code>var domain = db.getDomain('mydomain')</code>
<code>domain.alter()</code>	修改域的属性	<code>domain.alter({Groups:['data1','data3']})</code>

集合空间方法

方法名	描述	示例
<code>db.collectionspace.createCL()</code>	创建集合	<code>db.foo.createCL("bar")</code>
<code>db.collectionspace.dropCL()</code>	删除集合	<code>db.foo.dropCL("bar")</code>
<code>db.collectionspace.getCL()</code>	获取集合的引用	<code>db.foo.getCL("bar")</code>

集合方法

方法名	描述	示例
<code>db.collectionspace.collection.alter()</code>	修改集合的属性	<code>db.foo.bar.alter({ShardingKey: {a:1}, ShardingType:"hash"})</code>
<code>db.collectionspace.collection.insert()</code>	向集合中插入记录	<code>db.foo.bar.insert({name:"Tom",age:20,phone:[123,345]})</code>
<code>db.collectionspace.collection.count()</code>	统计集合记录数	<code>db.foo.bar.count({age:\$gt:20})</code>
<code>db.collectionspace.collection.find()</code>	查询记录	<code>db.foo.bar.find()</code>
<code>db.collectionspace.collection.aggregate()</code>	聚集	<code>db.foo.bar.aggregate({\$project:{name:1,age:1}}, {\$group:{_id:"\$sex"}})</code>
<code>db.collectionspace.collection.remove()</code>	删除记录	<code>db.foo.bar.remove()</code>
<code>db.collectionspace.collection.createIndex()</code>	创建索引	<code>db.foo.bar.createIndex("myIndex",{age:-1},false)</code>
<code>db.collectionspace.collection.dropIndex()</code>	删除索引	<code>db.foo.bar.dropIndex("myIndex")</code>
<code>db.collectionspace.collection.getIndex()</code>	获取索引引用	<code>db.foo.bar.getIndex("myIndex")</code>
<code>db.collectionspace.collection.listIndexes()</code>	枚举索引	<code>db.foo.bar.listIndexes()</code>
<code>db.collectionspace.collection.update()</code>	更新记录	<code>db.foo.bar.update({\$set:{age:25}},{age:\$lte:20}, {"": "myIndex"})</code>
<code>db.collectionspace.collection.upsert()</code>	更新记录	<code>db.foo.bar.update({\$set:{age:25}},{a:1}, {"": "myIndex"})</code>
<code>db.collectionspace.collection.split()</code>	集合分区	<code>db.foo.bar.split("sourceRG","targetRG", {age:20})</code>

方法名	描述	示例
<code>db.collectionspace.collection.splitAsync()</code>	异步分区	<code>db.foo.bar.splitAsync('sourceRG', 'targetRG', {id:1000})</code>
<code>db.collectionspace.collection.attachCL()</code>	在主分区集合下挂载子分区集合	<code>db.foo.year.attachCL("foo2.January",{LowBound:{date:"20130101"},UpBound:{date:"20130131"}})</code>
<code>db.collectionspace.collection.detachCL()</code>	从主分区集合中分离出子分区集合	<code>db.foo.year.detachCL("foo2.January")</code>

集群方法

方法名	描述	示例
<code>rg.start()</code>	启动分区组	<code>rg.start()</code>
<code>rg.getDetail()</code>	查看分区组信息	<code>rg.getDetail()</code>
<code>rg.createNode()</code>	创建节点	<code>rg.createNode(<host>,<service>,<dbpath>,[<config>])</code>
<code>rg.removeNode()</code>	删除节点	<code>rg.removeNode(<host>,<service>,[<config>])</code>
<code>rg.getMaster()</code>	查看主节点信息	<code>rg.getMaster()</code>
<code>rg.getSlave()</code>	查看从节点信息	<code>rg.getSlave()</code>
<code>rg.getNode()</code>	查看所有节点信息	<code>rg.getNode()</code>
<code>rg.stop()</code>	停止分区组	<code>rg.stop()</code>

节点方法

方法名	描述	示例
<code>node.start()</code>	启动节点	<code>node.start()</code>
<code>node.connect()</code>	返回连接节点的主机名和端口号	<code>node.connect()</code>
<code>node.getHostName()</code>	获取节点的主机名	<code>node.getHostName()</code>
<code>node.getServiceName()</code>	获得节点的服务器名称	<code>node.getServiceName()</code>
<code>node.getNodeDetail()</code>	获得节点信息	<code>node.getNodeDetail()</code>
<code>node.stop()</code>	停止节点	<code>node.stop()</code>

Cursor 方法

方法名	描述	示例
<code>cursor.sort()</code>	对结果集按指定字段排序	<code>db.foo.bar.find().sort({age:1})</code>
<code>cursor.hint()</code>	按指定的索引遍历结果集	<code>db.foo.bar.find().hint({"":""Index"})</code>
<code>cursor.limit()</code>	指定结果集返回记录的条数	<code>db.foo.bar.find().limit(10)</code>
<code>cursor.skip()</code>	指定结果集从哪条记录开始返回	<code>db.foo.bar.find().skip(10)</code>
<code>cursor.current()</code>	返回当前游标指向的记录	<code>db.foo.bar.find().current()</code>
<code>cursor.next()</code>	返回当前游标的下一条记录	<code>db.foo.bar.find().next()</code>
<code>cursor.size()</code>	返回当前游标到最终游标的距离	<code>db.foo.bar.find().size()</code>
<code>cursor.toArray()</code>	以数组的形式返回结果集	<code>db.foo.bar.find.toArray()</code>

Global

全局方法

`traceFmt()`

`traceFmt(<formatType>,<input>,<output>)`

将 `db.traceOff()` 导出来的二进制文件格式化为另外文件输出。

参数描述

参数名	参数类型	描述	是否必填
formatType	int	格式类型。	是
input	string	输入文件。	是
output	string	输出文件。	是

示例

- 格式化输出文件：

```
traceFmt(0, "/opt/sequoiadb/trace.dump", "/opt/sequoiadb/trace.flw")
```

Sdb

数据库操作方法

db.backupOffline()

backupOffline([options])

备份数据库。

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	设定备份名，指定复制组，备份方式等参数	否

Options格式

属性名	描述	格式
GroupID	指定备份的复制组 ID，缺省为所有复制组	GroupID:1000 或 GroupID:[1000, 1001]
GroupName	指定备份的复制组名，缺省为所有复制组	GroupName:"data1" 或 GroupName:["data1", "data2"]
Name	备份名称，缺省为“YYYY-MM-DD-HH:mm:SS”时间格式的备份名	Name:"backup-2014-1-1"
Path	备份路径，缺省为配置参数指定的备份路径。该路径支持通配符 (%g/%G: group name, %h/%H: host name, %s/%S:service name)	Path:"/opt/sequoiadb/backup/%g"
IsSubDir	上述 Path 参数所配置的路径是否为配置参数指定的备份路径的子目录，缺省为 false	IsSubDir:false
Prefix	备份前缀名，支持通配符 (%g,%G,%h,%H,%s,%S) ，缺省为空	Prefix:"%g_bk_"
EnableDataDir	是否开启日期子目录功能，如果开启则会自动根据当前日期创建“YYYY-MM-DD”的子目录，缺省为 false	EnableDataDir:false
Description	备份描述	Description:"First backup"
EnsureInc	是否开启增量备份，缺省为 false	EnsureInc:false
OverWrite	存在同名备份是否覆盖，缺省为 false	OverWrite:false

示例

- 对整个数据库进行全量备份

```
db.backupOffline({Name: "FullBackup1"})
```

`db.cancelTask()`

`db.cancelTask(<id>,[isAsync])`

取消任务

参数描述

参数名	参数类型	描述	是否必填
id	整数	任务 ID	是
isAsync	布尔	是否异步	否

示例

- 停止切分任务

```
var taskid1 = db.test.test.splitAsync("db1", "db2", 50);
db.cancelTask( taskid1, true )
```

`db.createCataRG()`

`db.createCataRG(<host>,<service>,<dbpath>,[config])`

新建一个编目分区组，同时创建并启动一个编目节点。

参数描述

参数名	参数类型	描述	是否必填
host	string	指定编目节点的主机名。	是
service	int	指定编目节点的服务端口，请确保该端口号，以及往后延续的3个端口号未被占用；如设置为11800，请确保11800/11801/11802/11803端口都未被占用。	是
dbpath	string	数据文件路径，用于存放编目数据文件，需确保数据管理员（安装时创建，默认为 sdbadmin）用户有写权限。	是
config	json	参数为可选参数，用于配置更多细节参数，格式必须为 json 格式，参数参见数据库配置一节；如需要配置日志大小参数{logfilesz:64}。	否

格式

`createCataRG()` 方法的定义格式有 host , service , dbpath , config 四个参数，host , dbpath 为字符串类型，service 为整型，config 为 json 对象，格式如下：

```
{"<主机名>,<端口号>,<数据文件路径>,[数据库配置参数对象]}
```

注:

- 
- 请确保数据文件存放路径的权限，如果 SequoiaDB 采用的默认安装，那么给路径赋予 sdbadmin 权限。

示例

- 在名为 : sdbserver1 的主机上创建一个编目节点组 , 服务端口为 : 11800 , 数据文件存放路径为 : /opt/sequoiadb/database/cata/11800

```
db.createCataRG("sdbserver1", 11800, "/opt/sequoiadb/database/cata/11800")
```

db.createCS()

db.createCS(<name>,[options])

在数据库对象中创建集合空间。

参数描述

参数名	参数类型	描述	是否必填
name	string	集合空间名。同一个数据库对象中 , 集合空间名必须唯一。	是
options	Json 对象	集合空间可选属性。	否

options 格式

属性名	描述	格式
PageSize	数据页大小。默认为65536B。	PageSize:<int32>
Domain	所属域	Domain:<string>
LobPageSize	Lob 数据页大小。默认262144B	LobPageSize:<int32>

注:

- 
- name 字段的值不能是空串 , 含点 (.) 或者美元符号 (\$) 。且长度不超过127B。
 - 同一个数据库对象集合空间名必须唯一。
 - 在创建集合空间时用户可以指定数据页大小 , 指定后不可更改。如果不指定默认为65536B。
 - PageSize 只能选填0 , 4096 , 8192 , 16384 , 32768 , 65536之一 , 0即为默认值65536。
 - 所属域必须已经存在 , 且不能为 SYS DOMAIN。
 - 为兼容较早版本接口 , db.createCS(<name>,[PageSize]) 同样可以工作。
 - LobPageSize 只能选填0 , 4096 , 8192 , 16384 , 32768 , 65536 , 131072 , 262144 , 524288之一 , 0即为默认值262144。

示例

- 创建名为 foo 的集合空间 , 不指定数据页大小 , 即数据页大小为默认值65536B

```
db.createCS("foo")
```

- 创建名为 foo 的集合空间 , 指定数据页大小为4096B , 所属域为“mydomain”

```
db.createCS("foo", {PageSize: 4096, Domain: "mydomain"})
```

db.createDomain()

db.createDomain(<name>,<groups>,[options])

创建一个域。域中可以包含若干个复制组 (Replica Group) 。

参数描述

参数名	参数类型	描述	是否必填
name	string	域名 , 全局唯一。	是
groups	Json 数组	域包含的复制组。	是

参数名	参数类型	描述	是否必填
options	Json 对象	在创建域时可以通过 options 设置其他属性。	否

格式

目前通过 options 可设置域的属性有：

属性名	描述	格式
AutoSplit	自动切分散列分区集合。	AutoSplit:true false



注：

- AutoSplit 只作用于散列分区集合。
- 不能在空域（不包含复制组）创建集合空间。

示例

- 创建一个域，包含两个复制组。

```
db.createDomain('mydomain', ['datagroup1', 'datagroup2'])
```

- 创建一个域，包含两个复制组，并且指定自动切分。

```
db.createDomain('mydomain', ['datagroup1', 'datagroup2'], {AutoSplit: true})
```

```
db.createProcedure()
```

```
db.createProcedure(<code>)
```

在数据库对象中创建存储过程。

参数描述

参数名	参数类型	描述	是否必填
code	自定义函数	标准函数定义，不是字符串类型，在输入参数时不能使用引号。	是

格式

createProcedure() 方法的定义格式包含 code 参数，参数值为标准函数定义。

```
createProcedure(<code>)
```

说明：

- 推荐直接使用存储过程中已初始化全局的 db，且全局 db 采用当前执行该存储过程的会话的鉴权信息，如：db.createProcedure(function getAll(){return db.foo.find();})。
- 自己初始化 db 的形式为 var db = new Sdb()，db 采用当前执行该存储过程的会话的鉴权信息。如果需要加入其它用户名和密码，为 var db = new Sdb('username','passwd')。这里需要注意的是，存储过程只能运行在已连接上的 db，不提供远程连接其他 db 的方法。在不需要鉴权的情况下，即使如 var db = new Sdb('hostname', 'servicename') 语句正常执行。得到的 db 仍然是本地 db。
- db 角色必须为协调节点。standalone 模式不提供存储过程功能。

函数定义

- 函数定义

1. 函数必须包含函数名。不能使用如：function(x,y){return x+y;}

2. 在函数定义时可以调用其他函数甚至是不存在的函数。但需要保证运行时所有函数已存在。

3. 函数名全局唯一。不提供重载。

4. 每个函数均在全系统可用。随意删除一个存储过程可能导致他人运行失败。

- 函数参数

native type of JS

- 函数输出

函数中所有标准输出，标准错误会被屏蔽。同时不建议在函数定义或执行时加入输出语句。大量的输出可能会导致存储过程运行失败

- 函数返回值

函数返回值可以是除 db 以外任意类型数据。如：function getCL(){return db.foo.bar;}

示例

- 创建 sum 函数

```
db.createProcedure(function sum(x, y) {return x+y; })
```

创建之后可以使用 [db.listProcedures\(\)](#) 查看函数信息。

`db.createRG()`

`db.createRG(<name>)`

新建一个分区组。创建后系统自动为分区组分配一个 GroupId。

参数描述

参数名	参数类型	描述	是否必填
name	string	分区组名，同一个数据库对象中，分区组名唯一。	是

格式

`createRG()` 方法的定义格式只有 name 字段，name 的值是字符串型，必填。

`(<"分区组名">)`

注：

- 分区组名不能是空串，含点（.）或者美元符号（\$），并且长度不能超过127B。

示例

- 新建名为“group”的分区组

```
db.createRG("group")
```

`db.createUsr()`

`db.createUsr(<name>,<password>)`

为数据库创建用户名和密码。防止非法用户对数据库进行非法操作。

参数描述

参数名	参数类型	描述	是否必填
name	string	用户名	是
password	string	密码	是

格式

`createUsr()` 方法的定义格式 name 和 password 两个参数，两个参数都是字符串类型，且可以为空串。

```
("<用户名>", "<密码>")
```

注:

当为数据库设置了用户名和密码时，那么只能使用正确的用户名和密码才能登录数据库进行相关操作。此时登录数据库的命令如下格式：

```
db = new Sdb("<hostname>", "<port>", "<name>", "<password>")
```

示例

- 为数据库创建用户名为 root，密码为 admin 的命令如下：

```
db.createUsr("root", "admin")
```

```
db.dropCS()
```

```
db.dropCS(<name>)
```

在数据库对象中删除指定的集合空间。

参数描述

参数名	参数类型	描述	是否必填
name	string	集合空间名。同一个数据库对象中集合空间名唯一。	是

格式

删除集合空间的定义格式只有 name 字段，name 的值为 string 类型，指定的集合空间名必须要在数据库对象中存在，否则操作异常。

```
("<集合空间名>")
```

注:

- name 字段的值不能使空串，含点（.），或者美元符号（\$）。且长度不超过127B。
- 集合空间在数据库对象中存在。

示例

- 删除名为 foo 的集合空间，假定 foo 已存在

```
db.dropCS("foo")
```

```
db.dropDomain()
```

```
db.dropDomain(<name>)
```

删除指定域。

参数描述

参数名	参数类型	描述	是否必填
name	string	域名。	是

格式

`dropDomain()` 方法的定义格式必须指定 `name` 参数，并且 `name` 的值在系统中存在，否则操作异常。

```
{"name": "<域名>"}
```



注:

- 删除域前必须保证域中不存在任何数据。
- 不能删除系统域。

示例

- 删除一个之前创建的域。

```
db.dropDomain('mydomain')
```

- 删除一个包含集合空间的域，返回错误：

```
> db.dropDomain('hello')
(nofile):0 uncaught exception: -256
Takes 0.1865s.
> getErr(-256)
Domain is not empty
```

`db.dropUsr()`

`db.dropUsr(<name>, <password>)`

删除数据库已有的用户名和密码。

参数描述

参数名	参数类型	描述	是否必填
<code>name</code>	<code>string</code>	用户名	是
<code>password</code>	<code>string</code>	密码	是

格式

`dropUsr()` 方法的定义格式 `name` 和 `password` 两个参数，两个参数都是字符串类型。

```
("<用户名>", "<密码>")
```

示例

- 删除用户名为 `root`，密码为 `admin` 的数据库权限。

```
db.dropUsr("root", "admin")
```

`db.eval()`

`db.eval(<code>)`

根据需要填入 JavaScript 语句。同时可以在语句中调用已经创建好的存储过程。

参数描述

参数名	参数类型	描述	是否必填
<code>code</code>	字符串	JavaScript 语句或者创建好的存储过程函数。	是

说明：

1. 执行成功则按照语句返回结果。可以将返回值直接赋值给另一个变量。如：var a = db.eval(' db.foo.bar'); a.find();
2. 执行失败会返回错误码及错误信息。如：{ "errmsg": "(nofile):1 ReferenceError: sum is not defined","retCode": -152 }
3. 在函数执行结束前操作不会返回。中途退出则终止整个执行，但已经执行的代码不会被回滚。
4. 自定义返回值的长度有一定限制，参考 SequoiaDB 插入记录的最大长度。
5. 支持定义临时函数，如：db.eval(' function sum(x,y){return x+y;} sum(1,2)')
6. 全局 db 使用方式与 [createProcedure](#) 相同。

 注：虽然语句中的所有输出都会被屏蔽，但还是建议不要加入任何打印语句。

示例

- 在 eval() 方法中调用存储过程函数 sum

```
//初始时 sum() 方法不存在，返回异常信息
> var a = db.eval('sum(1, 2)');
{ "errmsg": "(nofile):1 ReferenceError: getCL is not defined",
  "retCode": -152 }
(nofile):0 uncaught exception: -152
//初始化 sum()
>db.createProcedure(function sum(x, y) {return x+y; })
//调用 sum()
>db.eval('sum(1, 2)')
3
```

- 在 eval() 方法中填写 JavaScript 语句

```
>var rc = db.eval("db.foo.bar")
>rc.find()
{
  "_id": {
    "$oid": "5248d3867159ae144a000000"
  },
  "a": 1
}
{
  "_id": {
    "$oid": "5248d3897159ae144a000001"
  },
  "a": 2
}...
```

db.exec()

db.exec(<select sql>)

执行 SQL 的 select 语句

示例

- 从集合 my.my 中查所 age = 20 的记录

```
db.exec("select * from my.my where age = 20")
```

`db.execUpdate()`

`db.execUpdate(<other sql>)`

执行 SQL 的其它语句

示例

- 向集合 my.my 中插入新的记录

```
db.execUpdate("insert into my.my(name, age) values ('zhangshang', 30)")
```

`db.flushConfigure()`

`db.flushConfigure(<rule>)`

刷盘配置至配置文件

参数描述

参数名	参数类型	描述	是否必填
rule	Json 对象	刷盘规则	是

rule 格式

属性名	描述	格式
Global	true 表示否刷盘全系统配置 , false 表示只刷盘本节点配置	Global:true

示例

- 刷盘数据库配置

```
db.flushConfigure({Global: true})
```

`db.forceSession()`

`db.forceSession(<sessionID>)`

终止指定会话的当前操作。

参数描述

参数名	参数类型	描述	是否必填
sessionId	int	会话编号。	是

注:

- 只有用户会话可以被终止。

示例

- 终止编号为100的会话。

```
db.forceSession(100)
```

`db.getCS()`

`db.getCS(<name>)`

返回指定集合空间对象的引用。

参数描述

参数名	参数类型	描述	是否必填
name	string	集合空间名。同一个数据库对象中集合空间名唯一。	是

格式

`getCS()` 方法的定义格式只有 name 字段，name 的值是字符串型。

`("<集合空间名>")`



注:

- name 字段的值不能使空串，含点（.），或者美元符号（\$）。且长度不超过127B。
- 集合空间在数据库对象中存在

示例

- 返回集合空间 foo 的引用，假定 foo 已存在。

`db.getCS("foo")`

`db.getDomain()`

`db.getDomain(<name>)`

获取指定域。

参数描述

参数名	参数类型	描述	是否必填
name	string	域名。	是

格式

`getDomain()` 方法的定义格式必须指定 name 参数，并且 name 的值在系统中存在，否则操作异常。

`{"name": "<域名>"}`



注:

- 不能获取系统域。

示例

- 获取一个之前创建的域。

`var domain = db.getDomain('mydomain')`

`db.getRG()`

`db.getRG(<name>|<id>)`

返回分区组的引用。

参数描述

参数名	参数类型	描述	是否必填
name	string	分区组名。同一个数据库对象中，分区组名唯一。	name 和 id 任选
id	int	分区组 id，创建分区组时系统自动分配。	id 和 name 任选

格式

getRG() 方法定于格式包含 name 或 id 字段，name 为字符串型，id 为 int 型。指定的分区组名或 id 值要在数据库对象中存在，否则出现操作异常。

```
("<分区组名>" | <id>)
```



注:

- name 字段的值不能使空串，含点（.），或者美元符号（\$）。且长度不超过127B。

示例

- 指定 name 值，返回分区组 rg1 的引用

```
db.getRG("rg1")
```

- 指定 id 值，返回分区组 rg1 的引用

```
db.getRG(1000)
```

`db.invalidateCache()`

`db.invalidateCache([options])`

清除节点（数据节点/协调节点）的缓存。

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	清除缓存的选项。	否

格式

目前通过 options 可设置域的属性有：

属性名	描述	格式
Groups	需要清除缓存的目标。	Groups:null -- 当前协调节点； Groups:[‘data1’,‘data2’] -- 当前协调节点和指定的两个数据组； Groups:‘data1’ -- 当前协调节点和指定的一个数据组。



注:

- 当不指定 Groups 时，作用域为当前协调节点和所有数据节点。

示例

- 清除当前协调节点和数据组‘data1’的缓存信息。

```
db.invalidateCache({Groups: ‘data1’})
```

db.list()

`db.list(<listType>,[con],[sel],[sort])`

枚举列表。列表是一种轻量级得到当前系统状态的命令。查看更多有关[列表信息](#)

参数描述

参数名	参数类型	描述	是否必填
listType	枚举	列表类型 。	是
con	Json 对象	选择条件，只返回符合 con 字段值的记录，为 null 时，返回所有。	否
sel	Json 对象	选择返回字段名。为 null 时，返回所有的字段名。	否
sort	Json 对象	对返回的记录按选定的字段排序。1为升序；-1为降序。	否

格式

list() 方法定义格式有 listType , con , sel , sort四个参数，listType 为枚举类型，其他全部为 Json 对象，格式如下：

```
{"listType": "<列表类型>", ["con": {"字段名1": {"操作符1": "值1"}, "字段名2": {"操作符2": "值2"}...}], ["sel": {"字段名1": "", "字段名2": "", ...}], ["sort": {"字段名1": -1|1, "字段名2": 1|-1, ...}]}
```



注:

- listType 字段的值请参考[列表类型](#)
- sel 参数是一个 json 结构，字段名的值一般指定为空串。如果指定为如下结构：{"字段名1": "值1", "字段名2": "值2", ...}，对于记录中存在所选字段名的话，设定的值其实无效；对于记录中不存在所选字段名的话，返回{"字段名1": "值1", "字段名2": "值2", ...}
- 字段的值是数组的话，我们用".操作符引用数组内的元素。并加上双引号("")

示例

- 指定 listType 的值为 SDB_LIST_CONTEXTS :

```
db.list(SDB_LIST_CONTEXTS)
```

返回：

```
{
  "SessionID": 4,
  "Contexts": [ 0 ]
} ...
```

- 指定 listType 的值为 SDB_LIST_STORAGEUNITS :

```
db.list(SDB_LIST_STORAGEUNITS)
```

返回：

```
{
  "Name": "foo",
  "ID": 4094,
  "Logical ID": 1,
  "PageSize": 4096,
  "Sequence": 1,
  "NumCollections": 1,
  "CollectionHWM": 3,
  "Size": 172032000
}
```

- db.list(SDB_LIST_STORAGEUNITS, {"Logical ID": {\$gt: 1}}, {Name: "space", ID: 2}, {Name: 1})

返回符合条件 Logical ID 大于1的记录，并且每条记录只返回 Name 和 ID 这两个字段，记录按 Name 字段的值升序排序

```
{
  "ID": 4091,
  "Name": "foo"
}
{
  "ID": 4093,
  "Name": "name"
}...
```

`db.listBackup()`

`db.listBackup([options], [cond], [sel])`

查看数据库备份

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	设定备份名，指定复制组，路径等参数	否
cond	Json 对象	备份过滤条件	否
sel	Json 对象	查看备份选择输出的字段	否

Options格式

属性名	描述	格式
GroupID	指定备份的复制组 ID，缺省为所有复制组	GroupID:1000 或 GroupID:[1000, 1001]
GroupName	指定备份的复制组名，缺省为所有复制组	GroupName:"data1" 或 GroupName:["data1", "data2"]
Name	备份名称，缺省查看所有备份	Name:"backup-2014-1-1"
Path	备份路径，缺省为配置参数指定的备份路径。 该路径支持通配符 (%g/%G: group name, %h/%H: host name, %s/%S:service name)	Path:"/opt/sequoiadb/backup/%g"
IsSubDir	上述 Path 参数所配置的路径是否为配置参数指定的备份路径的子目录，缺省为 false	IsSubDir:false
Prefix	备份前缀名，支持通配符 (%g,%G,%h,%H,%s,%S)，缺省为空	Prefix:"%g_bk_"

示例

- 查看数据库所有备份信息

```
db.listBackup()
```

`db.listCollections()`

`db.listCollections()`

枚举集合，执行此方法会将指定集合空间下的集合信息全部显示出来。

示例

- db.listCollections()

返回：

```
{
  "Name": "foo.bar",
  "Details": [
    {
      "ID": 0,
      "Logical ID": 1,
      "Sequence": 1,
      "Indexes": 2,
      "Status": "Normal"
    }
  ]
}
```

`db.listCollectionSpaces()`

`db.listCollectionSpaces()`

枚举数据库中所有的集合空间。

示例

- `db.listCollectionSpaces()`

返回：

```
{
  "Name": "foo"
}
```

`db.listDomains()`

`db.listDomains()`

枚举域，执行此方法会显示系统中所有由用户创建的域。

示例

- `db.listDomains()`

返回：

```
{
  "_id": {
    "$oid": "539ea19669d195f36432111a"
  },
  "Name": "hello",
  "Groups": [
    {
      "GroupName": "data1",
      "GroupID": 1000
    },
    {
      "GroupName": "data2",
      "GroupID": 1001
    }
  ]
}
```

`db.listProcedures()`

`db.listProcedures([cond])`

枚举所有的存储过程函数。

参数描述

参数名	参数类型	描述	是否必填
cond	Json 对象	条件为空时，枚举所有的函数，不为空时，枚举符合条件函数。	是

`listProcedures()` 方法的定义，只有一个 Json 对象类型的参数名 `cond`，输入值时返回符合指定值的函数，否则的话返回所有的函数。

示例

- 返回所有的函数信息

```
>db.listProcedures()
{ "_id" : { "$oid" : "52480389f5ce8d5817c4c353" }, "name" : "sum", "func" : "function
sum(x, y) {
    return x + y;
}", "funcType" : 0
{ "_id" : { "$oid" : "52480d3ef5ce8d5817c4c354" }, "name" : "getA11", "func" : "function
getA11() {
    return db.foo.bar.find();
}", "funcType" : 0
...
...
```

- 指定返回函数名为 `sum` 的记录

```
>db.listProcedures({name: "sum"})
{ "_id" : { "$oid" : "52480389f5ce8d5817c4c353" }, "name" : "sum", "func" : "function
sum(x, y) {
    return x + y;
}", "funcType" : 0 }
```

`db.listReplicaGroups()`

`db.listReplicaGroups()`

枚举分区组。

示例

- 返回所有分区组信息，命令如下：

```
> db.listReplicaGroups()
返回: [
  "Group": [
    {
      "dbpath": "/opt/sequoiadb/data/11800",
      "HostName": "vmsvr2-suse-x64",
      "Service": [
        {
          "Type": 0,
          "Name": "11800"
        },
        {
          "Type": 1,
          "Name": "11801"
        }
      ]
    }
  ]
}
```

```
{
    "Type": 2,
    "Name": "11802"
},
{
    "Type": 3,
    "Name": "11803"
}
],
"NodeID": 1000
},
{
    "dbpath": "/opt/sequoiadb/data/11850",
    "HostName": "vmsvr2-suse-x64",
    "Service": [
        {
            "Type": 0,
            "Name": "11850"
        },
        {
            "Type": 1,
            "Name": "11851"
        },
        {
            "Type": 2,
            "Name": "11852"
        },
        {
            "Type": 3,
            "Name": "11853"
        }
    ],
    "NodeID": 1001
}
],
"GroupID": 1001,
"GroupName": "group",
"PrimaryNode": 1001,
"Role": 0,
>Status": 1,
"Version": 5,
"_id": {
    "$oid": "517b2fc33d7e6f820fc0eb57"
}
}
```

这个分区组有有两个节点：11800和11850，其中11850为主节点。[分区组详细信息请点击这里](#)

`db.listTasks()`

`db.listTasks([cond],[sel],[orderBy],[hint])`

查看数据库所有后台任务

参数描述

参数名	参数类型	描述	是否必填
cond	Json 对象	任务过滤条件	否
sel	Json 对象	任务选择字段	否
hint	Json 对象	保留项	否

示例

- 列出系统所有后台任务

```
db.listTasks()
```

```
db.removeBackup()
```

db.removeBackup([options])

删除数据库备份

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	设定备份名，指定复制组，备份路径等参数	否

options 格式

属性名	描述	格式
GroupID	指定备份的复制组 ID，缺省为所有复制组	GroupID:1000 或 GroupID:[1000, 1001]
GroupName	指定备份的复制组名，缺省为所有复制组	GroupName:"data1" 或 GroupName:["data1", "data2"]
Name	备份名称，缺省删除所有备份	Name:"backup-2014-1-1"
Path	备份路径，缺省为配置参数指定的备份路径。该路径支持通配符 (%g/%G: group name, %h/%H: host name, %s/%S:service name)	Path:"/opt/sequoiadb/backup/%g"
IsSubDir	上述 Path 参数所配置的路径是否为配置参数指定的备份路径的子目录，缺省为 false	IsSubDir:false
Prefix	备份前缀名，支持通配符 (%g,%G,%h,%H,%s,%S)，缺省为空	Prefix:"%g_bk_"

示例

- 删除数据库中备份名为“backup-2014-1-1”的备份信息

```
db.removeBackup({Name: "backup-2014-1-1"})
```

```
db.removeProcedure()
```

db.removeProcedure(<function name>)

删除指定的函数名，函数名必须存在，否则出现异常信息。

参数描述

参数名	参数类型	描述	是否必填
function name	字符串	函数名	是

removeProcedure() 方法的定义，只有一个字符串类型的参数名 function name，它的值必须已存在，否则异常。

示例

- 删除 sum 函数

```
db.removeProcedure("sum")
```

必须保证待删除函数的函数名和定义时的一致。诸如 db.removeProcedure('sum') 的调用将返回失败。

`db.removeRG()`

`db.removeRG(<name>)`

删除数据库中指定的分区组。

参数描述

参数名	参数类型	描述	是否必填
name	string	分区组名，同一个数据库对象中，分区组名唯一。	是

格式

`removeRG()` 方法的定义格式只有 `name` 字段，`name` 的值是字符串型，必填。

`(<"分区组名 ">)`



注:

- 分区组名必须存在。

示例

- 删除名为“group”的分区组

`db.removeRG("group")`

`db.resetSnapshot()`

`db.resetSnapshot([cond])`

重置快照。

参数描述

参数名	参数类型	描述	是否必填
cond	Json 对象	选择条件，只重置符合 cond 条件的快照记录，为 null 时，重置所有。	否

格式

`resetSnapshot()` 方法定义格式有 `cond` 参数，它是一个 Json 对象。

`{["cond": {"字段名1": {"匹配符1": "值1"}, "字段名2": {"匹配符2": "值2"}...}]}`

示例

- 重置 SessionID 大于1的快照。

`db.resetSnapshot({SessionID: {$gt: 1}})`

`db.setSessionAttr()`

`db.setSessionAttr (<options>)`

设置会话属性

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	会话属性选项	是

options格式

属性名	描述	格式
PreferedInstance	会话读操作优先选取的数据库实例标识；取值"m"/"M"/"s"/"S"/"a"/"A"/1-7，分别表示master/slave/anyone/node1-node7	PreferedInstance:"M"

示例

- 设置会话优先从“主”数据库实例获取数据

```
db.setSessionAttr({PreferedInstance: "M"})
```

```
db.snapshot()
```

```
db.snapshot(<snapType>,[cond],[sel],[sort])
```

枚举快照，快照是一种得到当前系统状态的命令。[查看更多有关快照信息](#)

参数描述

参数名	参数类型	描述	是否必填
snapType	枚举	快照类型。	是
cond	Json 对象	选择条件，只返回 cond 字段指定的节点或分区组的快照信息，为 null 时，返回整个集群的快照信息。	否
sel	Json 对象	选择返回字段名。为 null 时，返回所有的字段名。	否
sort	Json 对象	对返回的记录按选定的字段排序。1为升序；-1为降序。	否

格式

snapshot() 方法定义格式有 snapType , cond两个字段，snapType 为枚举类型，cond 为 Json 对象，格式如下：

```
{"snapType": "<快照类型>", ["cond": {"字段名1": {"操作符1": "值1"}, "字段名2": {"操作符2": "值2"}...} ]}
```



注:

- snapType 字段的值请参考[快照类型](#)
- sel 参数是一个 json 结构，字段名的值一般指定为空串。如果指定为如下结构：{"字段名1": "值1", "字段名2": "值2", ...}，对于记录中存在所选字段名的话，设定的值其实无效；对于记录中不存在所选字段名的话，返回 {"字段名1": "值1", "字段名2": "值2", ...}
- 字段的值是数组的话，我们用“.”操作符引用数组内的元素。并加上双引号（" "）

示例

- 指定 snapType 的值为 SDB_SNAP_CONTEXTS :

```
db.snapshot(SDB_SNAP_CONTEXTS)
```

返回整个集群的上下文快照：

```
{
```

```

"SessionID": "vmsvr1-cent-x64-1: 11820: 22",
"Contexts": [
  {
    "ContextID": 8,
    "Type": "DUMP",
    "Description": "BufferSize: 0",
    "DataRead": 0,
    "IndexRead": 0,
    "QueryTimeSpent": 0,
    "StartTimeStamp": "2013-12-28-16. 07. 59. 146399"
  }
]
}
{
  "SessionID": "vmsvr1-cent-x64-1: 11830: 22",
  "Contexts": [
    {
      "ContextID": 6,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimeStamp": "2013-12-28-16. 07. 59. 147576"
    }
  ]
}
{
  "SessionID": "vmsvr1-cent-x64-1: 11840: 23",
  "Contexts": [
    {
      "ContextID": 7,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimeStamp": "2013-12-28-16. 07. 59. 148603"
    }
  ]
}

```

- 通过组名或组 ID 查询某个分区组的快照信息，如：

```
db.snapshot(SDB_SNAP_CONTEXTS, {GroupName: 'data1'})
```

返回组名为“data1”的分区组快照信息

```
db.snapshot(SDB_SNAP_CONTEXTS, {GroupID: 1000})
```

返回组 ID 为“1000”的分区组快照信息

```
{
  "SessionID": "vmsvr1-cent-x64-1: 11820: 22",
  "Contexts": [
    {
      "ContextID": 11,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimeStamp": "2013-12-28-16. 13. 57. 864245"
    }
  ]
}
```

```
{
  "SessionID": "vmsvr1-cent-x64-1: 11840: 23",
  "Contexts": [
    {
      "ContextID": 10,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimeStamp": "2013-12-28-16.13.57.865103"
    }
  ]
}
```

- 通过“组名+主机名+服务名”或“组 ID+节点 ID”查询某个节点的快照信息，如：

```
db.snapshot(SDB_SNAP_CONTEXTS, {GroupName: 'data1', HostName: "vmsvr1-cent-x64-1", svcname: "11820"})
```

```
db.snapshot(SDB_SNAP_CONTEXTS, {GroupID: 1000, NodeID: 1001})
```

```
{
  "SessionID": "vmsvr1-cent-x64-1: 11820: 22",
  "Contexts": [
    {
      "ContextID": 11,
      "Type": "DUMP",
      "Description": "BufferSize: 0",
      "DataRead": 0,
      "IndexRead": 0,
      "QueryTimeSpent": 0,
      "StartTimeStamp": "2013-12-28-16.13.57.864245"
    }
  ]
}
```

- 通过“主机名+服务名”查询某个节点的快照信息，如：

```
db.snapshot(SDB_SNAP_CONTEXTS, {HostName: "vmsvr1-cent-x64-1", svcname: "11820"})
```

`db.startRG()`

`db.startRG(<name>)`

启动指定的分区组。分区组启动后才能在分区组上创建节点。这个方法等价于 `rg.start()`。

参数描述

参数名	参数类型	描述	是否必填
name	string	分区组的名称	是

格式

`db.startRG()` 的方法定义包含 `name` 一个参数，参数类型为字符串，为要启动的分区组名。

```
("<分区组名>")
```

注:

 指定的分区组名必须存在，不然返回异常；如果指定的分区组已经启动，再使用该方法同样会出现异常。

示例

- 启动分区组名为 group 的命令如下：

```
db.startRG("group")
```

db.traceOff()

db.traceOff(<dumpFile>)

关闭数据库引擎跟踪功能，并将跟踪情况导出二进制文件，如：/opt/sequoiadb/trace.dump

参数描述

参数名	参数类型	描述	是否必填
dump file	string	dump 的文件名称。	是

示例

- 关闭数据库引擎跟踪/opt/sequoiadb/trace.dump

```
db.traceOff("/opt/sequoiadb/trace.dump");
```

db.traceOn()

db.traceOn(<bufferSize>,[strComp],[strBreakPoint])

开启数据库引擎跟踪功能。

参数描述

参数名	参数类型	描述	是否必填
bufferSize	int	开启追踪的文件大小。	是
strComp	string	指定模块。	否
strBreakPoint	string	于函数处打断点进行跟踪。	否

示例

- 开启数据库引擎程序跟踪的功能，默认为所有模块：

```
db.traceOn(10000000);
```

- 开户数据库引擎程序跟踪功能，指定跟踪的模块名称和指定断点进行跟踪：

```
db.traceOn(10000000, "cls, dms mth", "_dmsTempCB::init");
```

db.traceResume()

db.traceResume()

重新开启断点跟踪程序。

参数描述

参数名	参数类型	描述	是否必填
-	-	-	-

示例

- 重新开启断点跟踪程序：

```
db.traceResume()
```

`db.traceStatus()`

`db.traceStatus()`

查看当前程序跟踪的状态。

参数描述

参数名	参数类型	描述	是否必填
-	-	-	-

示例

- 查看当前程序跟踪的状态：

```
db.traceStatus()
{
  "TraceStarted": true,
  "Wrapped": false,
  "Size": 524288,
  "Mask": [
    "auth",
    "bps",
    "cat",
    "cls",
    "dps",
    "mig",
    "msg",
    "net",
    "oss",
    "pd",
    "rtn",
    "sql",
    "tools",
    "bar",
    "client",
    "coord",
    "dms",
    "ixm",
    "mon",
    "mth",
    "opt",
    "pmd",
    "rest",
    "spt",
    "util",
    "aggr",
    "spd",
    "qgm"
  ],
  "BreakPoint": []
}
```

`db.transBegin()`

`db.transBegin()`

开启事务。SequoiaDB 数据库事务是指作为单个逻辑工作单元执行的一系列操作。事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。

示例

- 开启事务命令：

```
db.transBegin()
```

```
db.transCommit()
```

```
db.transCommit()
```

事务提交。在开启事务之后，如果单个逻辑工作单元执行的操作无异常，执行事务提交命令，那么数据库的数据将被更新。

示例

- 事务提交命令：

```
db.transCommit()
```

```
db.transRollback()
```

```
db.transRollback()
```

事务回滚。在开启事务之后，如果单个逻辑工作单元执行的操作出现异常，执行事务回滚命令，那么数据库回到原来状态。

示例

- 事务回滚命令：

```
db.transRollback()
```

```
db.waitTasks()
```

```
db.waitTasks(<id1>,[id2],...)
```

同步等待指定任务结束或取消

参数描述

参数名	参数类型	描述	是否必填
id1, id2...	整数	任务 ID	是

示例

- 同步等待数据切分任务完成

```
var taskid1 = db.test.test.splitAsync("db1", "db2", 50);
var taskid2 = db.my.my.splitAsync("db3", "db4", 50) ;
db.waitTasks( taskid1, taskid2 )
```

SdbCS

集合空间方法

```
db.collectionspace.createCL()
```

```
db.collectionspace.createCL(<name>,[options])
```

在指定集合空间下创建集合（Collection）。集合是数据库中存放文档记录的逻辑对象。任何一条文档记录必须属于一个且仅一个集合。

参数描述

参数名	参数类型	描述	是否必填
name	string	集合名，在同一个集合空间中，集合名必须唯一。	是
options	Json 对象	在创建集合时，可以通过 options 参数设置集合的其他属性，如指定集合的分区键，是否以压缩的形式插入数据等。	否

格式

createCL() 方法的定义格式包含 name 和 options 两个参数。name 的值为字符串类型，必须有值。options 是设置集合其他属性参数，目前通过 options 可设置集合的属性有：

属性名	描述	格式
ShardingKey	分区键。	ShardingKey:{<字段1>:<1 -1>,[<字段2>:<1 -1>, ...]}
ShardingType	分区方式，默认为 range 分区。	ShardingType:"hash" "range"
Partition	分区数，hash 分区时填写，代表了 hash 分区的个数。其值必须是2的幂。范围在[2^3, 2^20]。	Partition:<分区数>
ReplSize	副本数，默认情况下，副本写入个数为1。	ReplSize:<int num>
Compressed	数据压缩。	Compressed:true false
IsMainCL	主分区集合。标识是否为主分区集合，默认为否。	IsMainCL:true false
AutoSplit	自动切分。	AutoSplit:true false
Group	指定创建在某个复制组。	Group:<group name>

创建集合的格式为：

```
{"name": "<集合名>", [options]}
```

注:

- ShardingKey是一个 JSON 对象，JSON 对象中每一个字段对应分区键的字段，其值为1或者-1，代表正向或逆向排序。需要分区，必须指定 ShardingKey。
- ReplSize是一个 JSON 对象，它的值是 int 类型，设置写入数据节点的个数，默认为1，当 ReplSize 等于0时，副本写入个数会根据当前数据组中节点数变化而变化；手动指定副本写入个数时，不能超出当前组内节点个数。
- Compressed也是一个 JSON 对象，它的值为 boolean 类型，为“true”时，表示集合中的数据压缩存储，“false”时表示正常存储数据。
- 当 options 内设置了多个参数时，用逗号 (,) 隔开。
- name 的值不能是空串，含点 (.) 或者美元符号 (\$)，并且长度不能超过127B，否则操作失败。
- AutoSplit 必须配合散列分区和域使用，且不能与 Group 同时使用。
- AutoSplit 不能与 Group 同时使用。
- 如果在集合中没有指定 AutoSplit，则使用所属域中的 AutoSplit 参数。
- Group 必须存在于集合空间所属的域中（所有复制组均属于 SYSDOMAIN，即如果集合空间没有指定域，则系统内任意复制组均可）。

示例

- 在集合空间 foo 下创建集合 bar，不指定分区键

```
db.foo.createCL("bar")
```

- 在集合空间 foo 下创建集合 bar，指定字段 age 为分区键，升序排序

```
db.foo.createCL("bar", {ShardingKey:  
  {"age": 1}, ShardingType: "hash", Partition: 1024, Rep1Size: 1, Compressed: true})
```

`db.collectionspace.dropCL()`

`db.collectionspace.dropCL(<name>)`

删除指定集合空间下指定的集合。

参数描述

参数名	参数类型	描述	是否必填
name	string	集合名，在同一个集合空间中，集合名必须唯一。	是

格式

`dropCL()` 方法的定义格式必须指定 name 参数，并且 name 的值在集合空间中存在，否则操作异常。

```
{"name": "<集合名>"}
```



注:

- name 的值不能是空串、含点（.）或者美元符号（\$），并且长度不能超过127B，否则操作失败。
- 集合名必须在集合空间中存在，否则操作异常。

示例

- 删除集合空间 foo 下的集合 bar。假定集合存在

```
db.foo.dropCL("bar")
```

`db.collectionspace.getCL()`

`db.collectionspace.getCL(<name>)`

返回指定集合空间下集合的引用。

参数描述

参数名	参数类型	描述	是否必填
name	string	集合名，在同一个集合空间中，集合名必须唯一。	是

格式

`getCL()` 方法的定义格式必须指定 name 参数，并且 name 的值在集合空间中存在，否则操作异常。

```
{"name": "<集合名>"}
```



注:

- name 的值不能是空串、含点（.）或者美元符号（\$），并且长度不能超过127B，否则操作失败。
- 集合名必须在集合空间中存在，否则操作异常。

示例

- 返回集合空间 foo 下集合 bar 的引用。假定集合存在。

```
db.foo.getCL("bar")
```

SdbCollection

集合方法

`db.collectionspace.collection.aggregate()`

`db.collectionspace.collection.aggregate(<subOp>...)`

`aggregate()` 方法与 [find\(\)](#) 方法功能比较接近，也是从 SequoiaDB 的集合中检索文档记录，并返回游标。

参数描述

参数名	参数类型	描述	是否必填
subOp	json对象	subOp表示子操作，在 aggregate() 方法中可以填写 1~N 个子操作。	是

格式

`aggregate()` 方法只有一个参数 `subOp`，它表示 1~N 个子操作，每个子操作是一个 JSON 对象，子操作之间用逗号隔开。聚集框架支持以下子操作参数：

参数名	描述	示例
<code>\$project</code>	选择需要输出的字段名，“1”表示输出，“0”表示不输出，还可以实现字段的重命名。	<code>{\$project:{field1:1,field0:0,alias:"\$field3"}}</code>
<code>\$match</code>	实现从集合中选择匹配条件的记录，相当与 SQL 语句的 where。	<code>{\$match:{field:{\$lte:value}}}</code>
<code>\$limit</code>	限制返回的记录条数。	<code> {\$limit:10}</code>
<code>\$skip</code>	控制结果集的开始点，即跳过结果集中指定条数的记录。	<code> {\$skip:5}</code>
<code>\$group</code>	实现对记录的分组，类似与 SQL 的 group by 语句，“_id”指定分组字段。	<code> {\$group:{_id:"\$field"}}</code>
<code>\$sort</code>	实现对结果集的排序，“1”代表升序，“-1”代表降序。	<code> {\$sort:{field1:1,field2:-1,...}}</code>

说明：`aggregate()` 方法可以有任意多个子操作，但是注意各子操作的参数名的语法规则。

示例

假设集合 `collection` 包含如下格式的记录：

```
{
  no: 1000,
  score: 80,
  interest: ["basketball", "football"],
  major: "计算机科学与技术",
  dep: "计算机学院",
  info: {
    name: "Tom",
    age: 25,
    gender: "男"
  }
}
```

- 按条件选择记录，并指定返回字段名

```
db.collectionspace.collection.aggregate({$match: {$and: [ {no: {$gt: 1002}}, {no: {$lt: 1015}} ],
  {dep: "计算机学院"} ]}}, {$project: {no: 1, "info.name": 1, major: 1}})
```

此聚集操作操作首先使用 `$match` 选择匹配条件的记录，然后使用 `$project` 只返回指定的字段名。返回结果集如下：

```
{
  "no": 1003,
  "info.name": "Sam",
  "major": "计算机软件与理论"
}
{
  "no": 1004,
  "info.name": "Col1",
  "major": "计算机工程"
}
{
  "no": 1005,
  "info.name": "Jim",
  "major": "计算机工程"
}
```

- 按条件选择记录，并对记录进行分组

```
db.collectionspace.collection.aggregate({$match: {dep: "计算机学院"}}, {$group:
{_id: "$major", Major: {$first: "$major"}, avg-age: {$avg: "$info.age"}}})
```

此聚集操作首先使用 `$match` 选择匹配条件的记录，然后使用 `$group` 对记录按字段 `major` 进行分组，并使用 `$avg` 返回每个分组中嵌套对象 `age` 字段的平均值。

```
{
  "Major": "计算机工程",
  "avg-age": 25
}
{
  "Major": "计算机科学与技术",
  "avg-age": 22.5
}
{
  "Major": "计算机软件与理论",
  "avg-age": 26
}
```

- 按条件选择记录，并对记录进行分组、排序、限制返回记录的起始位置和返回记录数

```
db.collectionspace.collection.aggregate({$match: {interest: {$exists: 1}}}, {$group:
{_id: "$major", avg-age: {$avg: "$info.age"}, major: {$first: "$major"}}, {$sort: {avg-age: -1, major: -1}}, {$skip: 2}, {$limit: 3}})
```

此聚集操作首先按 `$match` 选择匹配条件的记录；然后使用 `$group` 按 `major` 进行分组，并使用 `$avg` 返回每个分组中嵌套对象 `age` 字段的平均值，输出字段名为 `avg_age`；最后使用 `$sort` 按 `avg_age` 字段值（降序），`major` 字段值（降序）对结果集进行排序，使用 `$skip` 确定返回记录的起始位置，使用 `$limit` 限制返回记录的条数。

```
{
  "avg-age": 25,
  "major": "计算机科学与技术"
}
{
  "avg-age": 22,
  "major": "计算机软件与理论"
}
{
  "avg-age": 22,
  "major": "物理学"
}
```

`db.collectionspace.collection.alter()`

`db.collectionspace.collection.alter(<options>)`

修改集合的属性。

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	修改的属性。	是

options 中可选的属性格式

属性名	描述	格式
ReplSize	一次写请求完成副本数。	ReplSize:<int32>
ShardingKey	分区键	ShardingKey:{<字段1>:<1 -1>,[<字段2>:<1 -1>, ...]}
ShardingType	分区方式，默認為 range 分区。	ShardingType:"hash" "range"
Partition	分区数，hash 分区时填写，代表了 hash 分区的个数。其值必须是2的幂。范围在[2^3,2^20]。	Partition:<分区数>

注:

- ShardingKey , ShardingType , Partition 的使用方式见 `db.collectionspace.createCL()`。
- 分区集合不能修改与分区相关的属性。
- 修改为分区集合后需要手动进行 split。

示例

- 创建一个普通集合；

```
db.foo.createCL('bar')
```

- 修改为分区集合

```
db.foo.bar.alter({ShardingKey: {a: 1}, ShardingType: "hash"})
```

`db.collectionspace.collection.attachCL()`

`db.collectionspace.collection.attachCL(<subCLFullName>, <options>)`

在主分区集合下挂载子分区集合。

参数描述

参数名	参数类型	描述	是否必填
subCLFullName	string	子分区集合名(包含集合空间名)	是
options	Json 对象	分区范围，包含两个字 段“LowBound”(区间左值)以 及“UpBound”(区间右值)，例 如：{LowBound:{a:0},UpBound: {a:100}}表示取字段“a”的范围区间： [0, 100)	是

示例

- 在主分区集合的指定区间下挂载子分区集合

```
db.foo.year.attachCL("foo2.January", {LowBound: {date: "20130101"}, UpBound: {date: "20130131"}})
```

`db.collectionspace.collection.count()`

`db.collectionspace.collection.count([cond])`

统计指定集合空间下指定集合的记录总数。

参数描述

参数名	参数类型	描述	是否必填
cond	Json 对象	选择条件。为空时，统计集合下所有的记录总数；不为空时，统计符合条件的记录总数。	否

格式

`count()` 方法的定义格式包含 `cond` 字段，它是一个 JSON 对象。

```
{[{"字段名1": {"匹配符1": "值1"}, "字段名2": {"匹配符2": "值2"}, ...}]}
```

示例

- 统计集合 `bar` 所有的记录数，即不指定参数 `cond`

```
db.foo.bar.count()
```

- 统计符合条件 `name` 字段的值为“Tom”且 `age` 字段的值大于25的记录数

```
db.foo.bar.count({name: "Tom", age: {$gt: 25}})
```

`db.collectionspace.collection.createIndex()`

`db.collectionspace.collection.createIndex(<name>,<indexDef>,[isUnique],[enforced])`

为集合创建索引，提高查询速度。

参数描述

参数名	参数类型	描述	是否必填
name	string	索引名，同一个集合中的索引名必须唯一。	是
indexDef	Json 对象	索引键，包含一个或多个指定索引字段与方向的对象。其中方向为1代表从小到大排序，-1则为从大到小排序。	是
isUnique	Boolean	索引是否唯一，默认 false。设置为 true 时代表该索引为唯一索引。	否
enforced	Boolean	索引是否强制唯一，可选参数，在 isUnique 为 true 时生效，默认 false。设置为 true 时代表该索引在 isUnique 为 true 的前提下，不可存在一个以上全空的索引键。	否

格式

`createIndex()` 方法定义包含 `name`, `indexDef`, `isUnique` 三个参数, 其中 `name` 的值必须为字符串; `indexDef` 则为一个 JSON 对象, `indexDef` 的对象必须包含至少一个字段, 其中字段名为用户需要索引的字段名, 其值为1或者-1。其中1代表升序, -1代表降序; `isUnique` 为布尔类型, 默认 false。

```
{"name": "<索引名>", "indexDef": {"<索引字段1>": <1|-1> [, "<索引字段2>": <1|-1>... ] },  
["isUnique": <true|false>], ["enforced": <true|false>]}
```

 **注:**

- 在唯一索引所指定的索引键字段上, 集合中不可存在一条以上的记录完全重复
- 索引名不能是空串, 含点(.) 或者美元符号(\$)。且长度不超过127B

示例

- 在集合 `bar` 下为字段名 `age` 创建名为 `ageIndex` 的唯一索引, 记录按 `age` 字段值的升序排序。

```
db.foo.bar.createIndex("ageIndex", {age: 1}, true)
```

```
db.collectionspace.collection.deleteLob()
```

```
db.collectionspace.collection.deleteLob(<oid>)
```

删除集合中的大对象。

参数描述

参数名	参数类型	描述	是否必填
<code>oid</code>	<code>string</code>	大对象的唯一描述符。	是

示例

- 删除一个描述符为 5435e7b69487faa663000897 的大对象

```
db.foo.bar.deleteLob('5435e7b69487faa663000897')
```

```
db.collectionspace.collection.detachCL()
```

```
db.collectionspace.collection.detachCL(<subCLFullName>)
```

从主分区集合中分离出子分区集合。

参数描述

参数名	参数类型	描述	是否必填
<code>partitionName</code>	<code>string</code>	子分区名(原子分区集合名)	是

示例

- 从主分区集合中分离指定子分区

```
db.foo.year.detachCL("foo2.January")
```

```
db.collectionspace.collection.dropIndex()
```

```
db.collectionspace.collection.dropIndex(<name>)
```

删除集合中指定的索引。

参数描述

参数名	参数类型	描述	是否必填
name	string	索引名，同一个集合中的索引名必须唯一。	是

格式

dropIndex() 方法的定义格式必须包含 name 字段。其中 name 的值必须为字符串。

```
{"name": "<索引名>"}
```

注:

- 做删除索引操作时，索引名必须在集合中存在。
- 索引名不能是空串，含点（.）或者美元符号（\$），且长度不超过127B。

示例

- 删除集合 bar 下名为 ageIndex 的索引，假设 ageIndex 已存在。

```
db.foo.bar.dropIndex("ageIndex")
```

```
db.collectionspace.collection.find()
```

```
db.collectionspace.collection.find([cond],[sel])
```

选择集合记录，对选择的记录返回一个游标（cursor）。在 SequoiaDB 中游标是一个指针，指向一个查询结果集，客户端可以遍历检索结果。

参数描述

参数名	参数类型	描述	是否必填
cond	Json 对象	选择条件。为空时，查询所有记录，不为空时，查询符合条件记录。	否
sel	Json 对象	控制返回字段名。为空时，返回记录的所有字段，如果指定的字段名不在记录中，返回。	否

格式

find() 的定义格式包含 cond 和 sel 两个参数，都是 JSON 对象类型。cond 控制符合条件的记录，sel 控制返回记录的字段名。

```
[{"字段名1": {"匹配符1": "值1", "字段名2": {"匹配符2": "值2"}, ...}], [{"字段名1": "", "字段名2": "", ...}]}
```

注:

sel 是一个 Json 对象，字段的值一般设定为空。而如果指定值：{"字段名1": "值1", "字段名2": "值2", ...}，如果记录中存在所选字段，设定的值（值1，值2...）不生效；如果记录中不存在所选字段，则按指定的值输出。

示例

- 查询所有记录，不指定 cond 和 sel 字段。

```
db.foo.bar.find()
```

- 查询匹配条件的记录，即设置 cond 参数的内容。

```
db.foo.bar.find({age: {$gt: 25}, name: "Tom"})
```

此操作返回集合 bar 中符合条件 age 字段值大于25且 name 字段值为“Tom”的记录。

- 指定返回的字段名，即设置 sel 参数的内容。如有记

录`{age:25,type:"system"}`和`{age:20,name:"Tom",type:"normal"}`

```
db.foo.bar.find(null, {age: "", name: ""})
```

此操作返回记录的 age 字段和 name 字段，执行后返回：`{age:25,name:""}`，`{age:20,name:"Tom"}`。虽然第一条记录没有 name 字段，还是会返回 name`:`""。

```
db.collectionspace.collection.findOne()
```

```
db.collectionspace.collection.findOne([cond],[sel])
```

此方法的使用与 `db.collectionspace.collection.find([cond],[sel])` 相同，具体的使用可以参照 `find()` 方法。但该操作方法只返回符合查询条件的一条记录。

```
db.collectionspace.collection.getIndex()
```

```
db.collectionspace.collection.getIndex(<name>)
```

返回指定索引的引用。

参数描述

参数名	参数类型	描述	是否必填
name	string	索引名，同一个集合中的索引名必须唯一。	是

格式

`getIndex()` 方法的定义格式必须包含 name 字段。其中 name 的值必须为字符串。

```
{"name": "<索引名>"}
```



注：

- 在做返回索引引用操作时，索引名必须在集合中存在。
- 索引名不能是空串，含点（.）或者美元符号（\$），且长度不超过127B。

示例

- 返回集合 bar 下名为 ageIndex 索引的引用，假设 ageIndex 已存在。

```
db.foo.bar.getIndex("ageIndex")
```

```
db.collectionspace.collection.getLob()
```

```
db.collectionspace.collection.getLob(<oid>,<file path>,[forced])
```

读取集合中的大对象。

参数描述

参数名	参数类型	描述	是否必填
oid	string	大对象的唯一描述符。	是
file path	string	待写入的本地文件全路径。	是
forced	bool	本地文件如果已经存在是否强制覆盖。	否



注：

- 本地文件不需要事先手工创建。

- forced 默认为 false。

示例

- 将标示符为 5435e7b69487faa663000897 的 lob 写入本地 /opt/newlob 文件

```
db.foo.bar.getLob('5435e7b69487faa663000897','/opt/newlob')
```

```
db.collectionspace.collection.insert()
```

```
db.collectionspace.collection.insert(<doc|docs>,[flag])
```

向指定集合中插入记录。如果集合空间或集合不存在，首先需要手动创建一个集合空间，如 db.createCS("foo")，再在该集合空间下手动创建集合，如 db.foo.createCL("bar")。然后在集合中插入记录。

参数描述

参数名	参数类型	描述	是否必填
doc docs	Json 对象	文档记录。doc 为一条记录，docs 为多条记录。	是
flag	Int	可取 SDB_INSERT_RETURN_ID 或者 SDB_INSERT_CONTONDUP。 前者在插入单条记录时有效，表示插入记录后返回记录中"_id"字段内容；后者在插入多条记录时有效，表示在插入的记录中，若存在"_id"字段内容重复的记录时，将跳过这些存在重复"_id"的记录继续插入后面记录。默认情况下，当存在重复"_id"字段内容的记录时，将停止插入后面的记录。	否

格式

insert() 方法的定义格式包含 doc|docs 和 flag 两个字段。

doc :

```
{"< 字段名 1>": "< 值 >", "< 字段名 2>": "< 值 >", ...}
```

docs :

```
{ [
  {"< 字段名 1>": "< 值 >", "< 字段名 2>": "< 值 >", ...},
  {"< 字段名 1>": "< 值 >", "< 字段名 2>": "< 值 >", ...},
  ...
]
```

注:

- 
- 如果插入的记录不指定 _id 字段时，SequoiaDB 会自动为记录添加一个 _id 字段来标识记录的唯一性。

示例

- 不指定 _id 字段，插入一条记录。

```
db.foo.bar.insert({name: "Tom", age: 20})
```

此操作是向集合 bar 中插入一条新的记录，name 字段的值为“Tom”，age 字段的值为 20，_id 字段被唯一创建：

```
{ "_id": { "$oid": "515152ba49af395200000000" }, "name": "Tom", "age": 20 }
```

- 插入一条带有 _id 字段的记录。

```
db.foo.bar.insert({_id: 10, age: 20})
```

此操作是向集合 bar 中插入一条新的记录，_id 字段的值为 10，age 字段的值为 20：

```
{ "_id": 10, "age": 20 }
```

- 插入多条记录。

```
db.foo.bar.insert([{_id: 20, name: "Mike", age: 15}, {name: "John", age: 25, phone: 123}])
```

此操作将会在集合 bar 中插入两条记录：

1) 其中一条记录 _id 字段的值为 20，name 字段的值为“Mike”，age 字段的值为 15。

2) 一条记录系统自动为 _id 字段生成唯一值，name 字段的值为“John”，age 字段的值为 25，phone 字段的值为 123。

```
{
  "_id": 20,
  "name": "Mike",
  "age": 15
}
```

```
{
  "_id": { "$oid": "5151557a49af395200000001" },
  "name": "John",
  "age": 25,
  "phone": 123
}
```

- 插入拥有重复“_id”键的多条记录。

```
db.foo.bar.insert([{_id: 1, a: 1}, {_id: 1, b: 2}, {_id: 3, c: 3}], SDB_INSERT_CONTONDUP)
```

此操作将会在集合 bar 中插入两条记录：

```
{
  "_id": 1,
  "a": 1,
}
{
  "_id": 3,
  "c": 3
}
```

```
db.collectionspace.collection.listIndexes()
```

```
db.collectionspace.collection.listIndexes()
```

枚举索引，执行此方法会将指定集合下的索引信息全部显示出来。

示例

- 返回集合 bar 下的所有索引信息

```
db.foo.bar.listIndexes()
```

```
db.collectionspace.collection.listLobs()
```

```
db.collectionspace.collection.putLob()
```

枚举集合中的大对象。

参数描述

参数名	参数类型	描述	是否必填
-	-	-	-



注:

- 此方法暂不支持排序等查询操作。

示例

- 枚举 foo.bar 中的所有大对象

```
db.foo.bar.listLobs()
```

```
db.collectionspace.collection.putLob()
```

```
db.collectionspace.collection.putLob(<file path>)
```

在集合中插入大对象。

参数描述

参数名	参数类型	描述	是否必填
file path	string	待上传的文件全路径。	是



注:

- 上传大对象成功后会返回其 OID。
- 需要拥有文件的读权限。

示例

- 创建集合空间与集合

```
db.createCS('foo')
db.foo.createCL('bar')
```

- 上传大对象文件

```
db.foo.bar.putLob('/opt/mylob');
```

```
db.collectionspace.collection.remove()
```

```
db.collectionspace.collection.remove([cond],[hint])
```

删除集合中的记录。

参数描述

参数名	参数类型	描述	是否必填
cond	Json 对象	选择条件。为空时，删除所有记录，不为空时，删除符合条件的记录。	否
hint	Json 对象	指定访问计划。	否

格式

cond 参数是一个Json 的对象。当它的内容为空（例如{}）时，删除集合下所有的记录。hint 参数是一个包含一个单一元素的 Json 对象，该元素的字段名会被忽略，而其字段值则指定为需要访问索引的名称，当字段值为 null 时，则遍历集合中所有的记录。

```
[{"字段名1": {"匹配符1": "值1", "字段名2": {"匹配符2": "值2"}, ...}], [{"": "索引名"}]
```

示例

- 删除集合所有记录

```
db.foo.bar.remove()
```

- 按访问计划删除匹配 cond 条件的记录

```
db.foo.bar.remove({age: {$gte: 20}}, {"": "myIndex"})
```

此操作按照索引名为“myIndex”的索引遍历集合中的记录，在遍历得到的记录中删除符合条件 age 字段值大于等于20的记录。

`db.collectionspace.collection.split()`

```
db.collectionspace.collection.split(<source group>,<target group>,<percent(0~100)|condition>,[endcondition])
```

在至少存在两个分区组的环境下，将数据记录按指定的条件切分到不同的分区组中。该操作为同步操作，直到数据切分完成才返回。

参数描述

参数名	参数类型	描述	是否必填
source group	string	源分区组。	是
target group	string	目标分区组。	是
percent(0~100)	double	百分比切分条件。	percent 和 condition 二选一
condition	Json 对象	范围切分条件。	condition 和 percent 二选一
endcondition	Json 对象	结束范围条件。	可选，且只在按条件切分时有效，按百分比切分时无效

格式

数据切分分为范围切分和百分比切分，其中“source group”和“target group”是公共参数，都是字符串类型；参数 condition 和 endcondition 为范围切分时填入，是一个 Json 结构的对象；参数 percent 为百分比切分时填入，是一个双精度浮点型数值。

- 范围切分

范围切分时，Range 分区使用精确条件，而 Hash 分区使用 Partition（分区数）条件。切分时起始条件为必填字段，而结束条件为选填条件，结束条件默认认为切分源当前包含的最大数据范围。

```
("<源分区组>","<目标分区组>",<condition>|<Partition>)
```

 注：范围分区时，如果指定分区键字段为降序时，如：`{groupingKey:{<字段名>:-1}}`，condition（或 Partition）中的起始条件中的范围应该大于终止条件中的范围。Hash 分区使用的 Partition（分区数）必须为整形，不能为其他的类型。

- 百分比切分

```
db.foo.bar.split("<源分区组>","<目标分区组>",<percent>)
```

示例

- Hash 分区范围切分

```
db.foo.bar.split("group1", "group2", {Partition: 10}, {Partition: 20})
```

- Range 分区范围切分

```
db.foo.bar.split("group1", "group2", {a: 10}, {a: 10000})
```

- 百分比切分

```
db.foo.bar.split("group1", "group2", 50)
```



注：百分比切分时，需要保证源分区组中含有数据，即集合不为空。

`db.collectionspace.collection.splitAsync()`

`db.collectionspace.collection.splitAsync(<source group>,<target group>,<percent(0~100)|condition>,[endcondition])`

该操作与 `db.collectionspace.collection.split` 功能相同，但该操作为异步分区操作，分区任务建立后立即返回任务 ID。

`db.collectionspace.collection.update()`

`db.collectionspace.collection.update(<rule>,[cond],[hint])`

更新集合记录。

参数描述

参数名	参数类型	描述	是否必填
rule	Json 对象	更新规则。记录按 rule 的内容更新。	是
cond	Json 对象	选择条件。为空时，更新所有记录，不为空时，更新符合条件的记录。	否
hint	Json 对象	指定访问计划。	否

格式

`update()` 方法的定义必须包含 `rule` 字段，`rule` 是一个 Json 对象。`cond` 和 `hint` 字段可选。`hint` 参数是一个包含一个单一字段的 Json 对象，字段名会被忽略，而其字段值则指定为需要访问索引的名称，当字段值为 `null` 时，则遍历集合中所有的记录，它的格式为`{"":null}`或者`{"":<indexname>}`。

```
{<""更新符1"": {"字段名1: "值"}, "更新符2": {"字段名2": "值2"}, ...>, [{"字段名1": {"匹配符1": "值1"}, "字段名2": {"匹配符2": "值2"}, ...}], [{"": "索引名"}]}
```

注：`update` 本版本暂不支持对表分区键（ShardingKey）字段更新，如果包含对分区键的更新操作，将自动剔除掉对分区键的更新，但其他字段更新生效，且不会发生错误。

示例

- 按指定的更新规则更新集合中所有记录，即设置 `rule` 参数，不设定 `cond` 和 `hint` 参数的内容

```
db.foo.bar.update({$inc: {age: 1}})
```

此操作更新集合 `bar` 下的 `age` 字段，将 `age` 字段的值增加1。

- 选择符合匹配条件的记录，对这些记录按更新规则更新，即设定 `rule` 和 `cond` 参数

```
db.foo.bar.update({$unset: {age: ""}}, {age: {$exists: 1}, name: {$exists: 0}})
```

此操作更新集合 `bar` 中存在 `age` 字段而不存在 `name` 字段的记录，将这些记录的 `age` 字段删除。

- 按访问计划更新记录，假设集合中存在指定的索引名

```
db.foo.bar.update({$inc: {age: 1}, {age: {$gt: 20}}, {"": "testIndex"})>
```

此操作使用索引名为 `testIndex` 的索引访问集合 `bar` 中 `age` 字段值大于20的记录，将这些记录的 `age` 字段加1。

`db.collectionspace.collection.upsert()`

`db.collectionspace.collection.upsert(<rule>,[cond],[hint])`

更新集合记录。`upsert` 方法跟 `update` 方法都是对记录进行更新，不同的是当使用 `cond` 参数在集合中匹配不到记录时，`update` 不做任何操作，而 `upsert` 方法会做一次插入操作。

参数描述

参数名	参数类型	描述	是否必填
rule	Json 对象	更新规则。记录按 rule 的内容更新。	是
cond	Json 对象	选择条件。为空时，更新所有记录，不为空时，更新符合条件的记录。	否
hint	Json 对象	指定访问计划。	否

格式

`upsert()` 方法的定义必须包含 `rule` 字段，`rule` 是一个 Json 对象。`cond` 和 `hint` 字段可选。`hint` 参数是一个包含一个单一字段的 Json 对象，字段名会被忽略，而其字段值则指定为需要访问索引的名称，当字段值为 `null` 时，则遍历集合中所有的记录，它的格式为`{"":null}`或者`{"":<indexname>}`。

```
<{ "更新符1": {"字段名1": "值"}, "更新符2": {"字段名2": "值2"}, ... }>, [{"字段名1": {"匹配符1": "值1"}, "字段名2": {"匹配符2": "值2"}, ...}], [{"": "索引名"}|null]}
```

 注：`upsert` 本版本暂不支持对表分区键（ShardingKey）字段更新，如果包含对分区键的更新操作，将自动剔除掉对分区键的更新，但其他字段更新生效，且不会发生错误。

示例

假设集合 `bar` 中有两条记录：

```
{
  "_id": {
    "$oid": "516a76a1c9565daf06030000"
  },
  "age": 10,
  "name": "Tom"
}
{
  "_id": {
    "$oid": "516a76a1c9565daf06050000"
  },
  "a": 10,
  "age": 21
}
```

- 按指定的更新规则更新集合中所有记录，即设置 `rule` 参数，不设定 `cond` 和 `hint` 参数的内容

```
db.foo.bar.upsert({$inc: {age: 1}, $set: {name: "Mike"}})
```

此操作等效于使用 `update` 方法，更新集合 `bar` 中的所有记录，将记录的 `age` 字段值加1，`name` 字段值更改为“Mike”，对不存在 `name` 字段的记录，`$set` 操作符会将 `name` 字段和其设定的值插入到记录中，使用 `find` 方法返回：

```
{
  "_id": {
    "$oid": "516a76a1c9565daf06030000"
  },
  "age": 11,
```

```

        "name": "Mike"
    }
{
    "_id": {
        "$oid": "516a76a1c9565daf06050000"
    },
    "a": 10,
    "age": 22,
    "name": "Mike"
}

```

- 选择符合匹配条件的记录，对这些记录按更新规则更新，即设定 rule 和 cond 参数

```
db.foo.bar.upsert({$inc: {age: 3}}, {type: {$exists: 1}})
```

此操作更新集合 bar 中存在 type 字段的记录，将这些记录的 age 字段值加3。在上面给出的两条记录中，都没有 type 字段，此时，upsert 操作会插入一条新的记录，新记录只有 _id 字段和 age 字段名，_id 字段值自动生成，而 age 字段值为3。

```

{
    "_id": {
        "$oid": "516a76a1c9565daf06030000"
    },
    "age": 11,
    "name": "Mike"
}
{
    "_id": {
        "$oid": "516a76a1c9565daf06050000"
    },
    "a": 10,
    "age": 22,
    "name": "Mike"
}
{
    "_id": {
        "$oid": "516cfcc334630a7f338c169b0"
    },
    "age": 3
}

```

- 按访问计划更新记录，假设集合中存在指定的索引名 testIndex

```
db.foo.bar.upsert({$inc: {age: 1}}, {age: {$gt: 20}}, {"": "testIndex"})>
```

此操作等效于使用 update 方法，使用索引名为 testIndex 的索引访问集合 bar 中 age 字段值大于20的记录，将这些记录的 age 字段名加1。此时返回：

```
{
    "_id": {
        "$oid": "516a76a1c9565daf06050000"
    },
    "a": 10,
    "age": 23,
    "name": "Mike"
}
```

SdbCursor

游标方法

`cursor.close()`

`cursor.close()`

关闭当前游标，当前游标不再可用。

示例

- 插入10条记录

```
for(i = 0; i < 10; i++) { db.foo.bar.insert({a: i}) }
```

查询集合 foo.bar 的所有记录

```
var cur = db.foo.bar.find()
```

使用游标取出一条记录

```
cur.next()
```

```
{
  "_id": {
    "$oid": "53b3c2d7bb65d2f74c000000"
  },
  "a": 0
}
```

关闭游标

```
cur.close()
```

再次获取下一条记录

```
cur.next()
```

无结果返回

`cursor.current()`

`cursor.current()`

返回当前游标指向的记录。更多查看 [cursor.next\(\)](#) 方法。

示例

- 选择集合 bar 下 age 大于10的记录，返回当前游标指向的记录。

```
db.foo.bar.find({age: {$gt: 10}}).current()
```

`cursor.explain()`

`cursor.explain([opation])`

返回查询的访问计划。

参数描述

参数名	参数类型	描述	是否必填
option	json 对象	访问计划执行参数，目前有 Run 字段项，表示是否执行访问计划，true 表示执行访问计划，获取数据和时间信息，false 表示只获取访问计划的信息，并不执行	否，默认为 false

访问计划描述

字段名	类型	描述
Name	string	集合名
ScanType	string	扫描方式——表扫描：“tbscan”；索引扫描：“ixscan”

字段名	类型	描述
IndexName	string	使用索引的名称
UseExtSort	bool	是否使用非索引排序
NodeName	string	节点名
ReturnNum	int64	返回记录数量
Elapsed Time	float64	查询耗时 (秒)
IndexRead	int64	索引记录扫描条数
DataRead	int64	数据记录扫描条数
UserCPU	float64	用户态 CPU 使用时间 (秒)
SysCPU	float64	内核态 CPU 使用时间 (秒)
SubCollections	Json Array	垂直分区表中各子表访问计划

格式



注: 如果集合经过 split 分布在多个复制组 , 访问计划会按照一组一记录的方式返回。

示例

- foo.bar 是一个水平分区集合 , 分布在三个复制组上。

```
db.foo.bar.find().sort({b:1}).explain()
```

返回 :

```
{
  "Name": "foo.bar",
  "ScanType": "tbscan",
  "IndexName": "",
  "UseExtSort": true,
  "NodeName": "vmsvr2-cent-x64: 40020",
  "ReturnNum": 38,
  "Elapsed Time": 0.000477,
  "IndexRead": 0,
  "DataRead": 38,
  "UserCPU": 0,
  "SysCPU": 0
}
{
  "Name": "foo.bar",
  "ScanType": "tbscan",
  "IndexName": "",
  "UseExtSort": true,
  "NodeName": "vmsvr2-cent-x64: 40000",
  "ReturnNum": 34,
  "Elapsed Time": 0.000415,
  "IndexRead": 0,
  "DataRead": 34,
  "UserCPU": 0,
  "SysCPU": 0
}
{
  "Name": "foo.bar",
  "ScanType": "tbscan",
  "IndexName": "",
  "UseExtSort": true,
  "NodeName": "vmsvr2-cent-x64: 40010",
  "ReturnNum": 28,
  "Elapsed Time": 0.000517,
```

```

    "IndexRead": 0,
    "DataRead": 28,
    "UserCPU": 0,
    "SysCPU": 0
}

```

`cursor.hint()`

`cursor.hint(<hint>)`

按指定的索引遍历结果集。

参数描述

参数名	参数类型	描述	是否必填
hint	Json 对象	指定访问计划，加快查询速度。	否

格式

`cursor.hint()` 的方法定义包含 `hint` 参数，如果不设定 `hint` 参数的内容相当于不使用索引遍历结果集。`hint` 参数是一个包含一个单一字段的 Json 对象，字段名会被忽略，而其字段值则指定为需要访问索引的名称，当字段值为 `null` 时，则遍历集合中所有的记录。

格式如下：

```
{"": null} 或者 {"": "<indexname>"}
```

示例

- 使用索引 `ageIndex` 遍历集合 `bar` 下存在 `age` 字段的记录，并返回。

```
db.foo.bar.find({age: {$exists: 1}}).hint({"": "ageIndex"})
```

`cursor.limit()`

`cursor.limit(<num>)`

控制结果集返回记录的条数。

参数描述

参数名	参数类型	描述	是否必填
num	int	自定义返回结果集的记录条数。	否

格式

`cousor.limit()` 方法的定义格式包含 `num` 参数，它是 `int` 类型。如果不设定 `num` 的内容，相当于返回所有的结果集记录。如果想返回结果集的前5条记录，可是设置 `num` 的值为5。

示例

- 选择集合 `bar` 下 `age` 字段值大于10的记录，控制返回前面10条记录。

```
db.foo.bar.find({age: {$gt: 10}}).limit(10)
```

注:

如果结果集的记录数小于10，按实际的记录数返回，如果结果集的记录数大于10，则返回前10条。

`cursor.next()`

`cursor.next()`

返回当前游标指向的下一条记录。更多查看 [cursor.current\(\)](#) 方法。

示例

- 选择集合 bar 下 age 大于10的记录，返回当前游标指向的下一条记录。

```
db.foo.bar.find({age: {$gt: 10}}).next()
```

`cursor.size()`

`cursor.size()`

返回当前游标到最终游标的记录条数。

示例

- 选择集合 bar 下 age 大于10的记录，返回当前游标到最终游标的记录条数。

```
db.foo.bar.find({age: {$gt: 10}}).size()
```

`cursor.skip()`

`cursor.skip(<num>)`

指定结果集从哪条记录开始返回。

参数描述

参数名	参数类型	描述	是否必填
num	int	自定义结果集从哪条记录返回。	否

格式

`cousor.skip()` 方法的定义格式包含 num 参数，它是 int 类型。如果不设定 num 的内容或者设定 num 的值为0，相当于返回所有的结果集；如果想从结果集的第3条记录开始返回，可是设置 num 的值等于2。

示例

- 选择集合 bar 下 age 字段值大于10的记录，从第5条记录开始返回，即跳过前面的四条记录

```
db.foo.bar.find({age: {$gt: 10}}).skip(4)
```



注：

如果结果集的记录数小于5，那么无记录返回；如果结果集的记录数大于5，从第5条开始返回。

`cursor.sort()`

`cursor.sort(<sort>)`

对结果集按指定字段排序。

参数描述

参数名	参数类型	描述	是否必填
sort	Json 对象	对结果集按指定字段排序。字段名的值为1或者-1,1代表升序；-1代表降序。	否

格式

cursor.sort() 方法的定义格式包含 sort 参数，它是一个 Json 对象。如果不设定 sort 的内容，相当于对返回的结果集不排序。

设定 sort 参数的话，格式如下：

```
{<字段名1>: <-1|1>, <字段名2>: <-1|1>, ...}
```

示例

- 返回集合 bar 中 age 字段值大于20的记录，设置只返回记录的 name 和 age 字段，并按 age 字段值的升序排序。

```
db.foo.bar.find({age: {$gt: 20}}, {age: "", name: ""}).sort({age: 1})
```



通过 [find\(\)](#) 方法，我们能任意选择我们想要返回的字段名，在上例中我们选择了返回记录的 age 和 name 字段，此时用 sort() 方法时，只能对记录的 age 或 name 字段排序。而如果我们选择返回记录的所有字段，即不设置 find 方法的 sel 参数内容时，那么 sort() 能对任意字段排序。

cursor.toArray()

cursor.toArray()

以数组的形式返回结果集。

示例

- 以数组的形式返回集合 bar 中 age 字段值大于5的记录。

```
db.foo.bar.find({age: {$gt: 10}}).toArray()
```

```
返回: [
  {
    "_id": {
      "$oid": "516a76a1c9565daf06030000"
    },
    "age": 10,
    "name": "Tom"
  },
  {
    "_id": {
      "$oid": "516a76a1c9565daf06050000"
    },
    "age": 20,
    "a": 10
  },
  {
    "_id": {
      "$oid": "516a76a1c9565daf06040000"
    },
    "age": 15
  }
]
```

SdbReplicaGroup

集群方法

```
rg.createNode()
```

`rg.createNode(<host>,<service>,<dbpath>,[config])`

在分区组中创建节点。



注:

只有在分区组启动之后，才能创建节点操作。

参数描述

参数名	参数类型	描述	是否必填
host	string	指定节点的主机名。	是
service	string	节点端口号。	是
dbpath	string	节点路径。	是
config	Json 对象	节点配置信息，如配置日志大小，是否打开事务等，具体可参考 数据库配置 。	否

格式

`rg.createNode()` 方法的定义格式有四个参数：`host`，`service`，`dbpath`，`config`，如上表所示，前三个参数的类型都是字符串型，必填；最后一个也是 Json 对象，选填。

```
("<主机名>,<端口号>,<节点路径>,"[ {<configParam>: value,...} ])
```

示例

- 在分区组 group 中创建一个端口号为11800的节点，指定日志文件大小为64MB

```
rg.createNode("vmsvr2-suse-x64",11800,"/opt/sequoiadb/data/11800", {logfilesz: 64})
```



注:

在一个分区组中能创建多个节点，但是连个节点的端口号必须相差5以上。因为系统为每个节点后台控制了5个通信接口。

```
rg.getDetail()
```

`rg.getDetail()`

返回分区组的信息。

示例

- > rg.getDetail()

```
{
  "Group": [
    {
      "dbpath": "/opt/sequoiadb/data/11800",
      "HostName": "vmsvr2-suse-x64",
      "Service": [
        {
          "Type": 0,
          "Name": "11800"
        },
        {
          "Type": 1,
          "Name": "11801"
        }
      ]
    }
  ]
}
```

```
{
  "Type": 2,
  "Name": "11802"
},
{
  "Type": 3,
  "Name": "11803"
},
],
"NodeID": 1000
},
{
  "dbpath": "/opt/sequoiadb/data/11850",
  "HostName": "vmsvr2-suse-x64",
  "Service": [
    {
      "Type": 0,
      "Name": "11850"
    },
    {
      "Type": 1,
      "Name": "11851"
    },
    {
      "Type": 2,
      "Name": "11852"
    },
    {
      "Type": 3,
      "Name": "11853"
    }
  ],
  "NodeID": 1001
}
],
"GroupID": 1001,
"GroupName": "group",
"PrimaryNode": 1001,
"Role": 0,
"Status": 1,
"Version": 3,
"_id": {
  "$oid": "517b2fc33d7e6f820fc0eb57"
}
}
```

`rg.getMaster()`

`rg.getMaster()`

返回分区组的主节点。

示例

- 返回分区组的主节点

```
> rg.getMaster()
返回:
vmsvr2-suse-x64: 11850
```

`rg.getNode()`

`rg.getNode(<nodename|hostname>,<servicename>)`

返回指定节点信息。

参数描述

参数名	参数类型	描述	是否必填
nodename	string	节点名称。	nodename 与 hostname 二选一
hostname	string	主机名。	hostname 与 nodename 二选一
servicename	string	服务器名称。	是

格式

`rg.getNode()` 方法定义了两个参数，第一个参数可是节点名称也可以是主机名，第二个参数为服务器名称。两个参数的类型都是字符串型，且必填。

```
("<节点名称>|<主机名>,<服务器名称>")
```

示例

- 返回指定主机名和服务器名的节点

```
> rg.getNode("vmsvr2-suse-x64", "11800")
返回:
vmsvr2-suse-x64: 11800
```

`rg.getSlave()`

`rg.getSlave()`

返回分区组的从节点。

示例

- 返回分区组的从节点

```
> rg.getSlave()
返回:
vmsvr2-suse-x64: 11800
```

`rg.removeNode()`

`rg.removeNode(<host>,<service>,[config])`

删除分区组中的指定节点。

参数描述

参数名	参数类型	描述	是否必填
host	string	节点主机名。	是
service	string	节点端口号。	是
config	Json 对象	节点配置信息。	否

格式

`rg.removeNode()` 方法的定义格式有三个参数：`host` , `service` , `config` , 如上表所示，格式如下：

```
("<主机名>,<端口号>[, {<configParam>: value,...}])
```

示例

- 在分区组 group 中删除节点命令如下

```
rg.removeNode ("vmsvr2-suse-x64", 11800)
```



注:

指定删除的节点必须存在，否则出现异常。

```
rg.start()
```

```
rg.start()
```

启动分区组。分区组启动之后才能创建节点及其他操作。也可以使用方法 [db.startRG\(<name>\)](#) 启动指定的节点。

示例

- 启动分区组命令：

```
rg.start() //等价于 db.startRG("group")
```

```
rg.stop()
```

```
rg.stop()
```

停止分区组。停止之后就不能创建节点等相关操作。

示例

- 停止分区组

```
rg.stop()
```

SdbNode

节点方法

```
node.connect()
```

```
node.connect()
```

将数据库连接到指定节点。连接之后能进行一系列的操作，可以使用 `node.connect().help()` 查看相关的操作。

示例

- 将数据库连接到节点名为 node 上

```
> node.connect()
返回:
vmsvr2-suse-x64: 11800
```

```
node.getHostName()
```

```
node.getHostName()
```

返回节点的主机名。

示例

- 返回节点名称为 node 的主机名

```
> node.getHostName()
返回:
vmsvr2-suse-x64
```

`node.getNodeDetail()`

`node.getNodeDetail()`

返回当前节点信息。

示例

- 返回节点名称为 node 的信息

```
> node.getNodeDetail()
返回:
1000: vmsvr2-suse-x64: 11800 (group)
其中 "1000" 为节点 ID (NodeID) ; "vmsvr2-suse-x64" 为主机名 (HostName) ; "11800" 为服务器
名 (ServiceName) , " (group)" 为节点所在的分区组名。
```

`node.getServiceName()`

`node.getServiceName()`

返回节点的服务器名。

示例

- 返回节点名为 node 的服务器名

```
> node.getServiceName()
返回:
11800
```

`node.start()`

`node.start()`

启动当前节点。

示例

- 启动当前名称为 node 的节点。

```
node.start()
```

`node.stop()`

`node.stop()`

停止当前节点。

示例

- 停止当前名称为 node 的节点。

```
node.stop()
```

SdbDomain

域方法

`domain.alter()`

`domain.alter(<options>)`

修改域的属性。

参数描述

参数名	参数类型	描述	是否必填
options	Json 对象	需要修改的属性列表。	是

格式

目前通过 options 可设置域的属性有：

属性名	描述	格式
Groups	包含的复制组。	Groups:[‘data1’,‘data2’]
AutoSplit	自动切分。	AutoSplit:true false

 **注:**

- 删除复制组前必须保证其不包含任何数据。
- AutoSplit 的更改不对之前创建的集合和集合空间产生影响。

示例

- 首先创建一个域，包含两个复制组，开启自动切分。

```
> var domain = db.createDomain('mydomain', ['data1', 'data2'], {AutoSplit: true})
```

从域中删除一个复制组 data2，添加另一个复制组 data3，最后域中包含 data1 和 data3 两个复制组。

```
> domain.alter({Groups: ['data1', 'data3']})
```

- 首先创建一个域，包含一个复制组，复制组中包含表 foo.bar。

```
> var domain = db.createDomain('mydomain', ['group1'])
```

从域中删除原复制组，添加另一个复制组，将因把拥有数据的 group1 从域中删除而报错。

```
> domain.alter({Groups: ['group2']})
(node):0 uncaught exception: -256
> getErr(-256)
Domain is not empty
```

`domain.listCollections()`

`domain.listCollections()`

枚举集合，执行此方法会将指定域下的集合信息全部显示出来。

示例

- `domain.listCollections()`

返回：

```
{
  "Name": "foo.bar"
```

```

    }
domain.listCollectionSpaces()

```

domain.listCollectionSpaces()
枚举域中所有的集合空间。

示例

- domain.listCollectionSpaces()

返回：

```
{
  "Name": "foo"
}
```

Oma

操作维护代理方法

oma.close()

oma.close()

关闭 oma

参数描述

参数名	参数类型	描述	是否必填
-	-	-	-



注:

- 关闭的 oma 必须存在，否则出现异常。

示例

- 关闭 oma

```
oma.close()
```

oma.createCoord()

oma.createCoord(<svcname>,<dbpath>,[config obj])

在集群中创建一个 coord 节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是
dbpath	string	节点路径。	是
config obj	Json 对象	节点配置信息，如配置日志大小，是否打开事务等，具体可参考 数据库配置 。	否



注:

- 在一个集群中可以创建多个节点，但是连个节点的端口号必须相差5以上，因为系统为每个节点后台控制了5个通信接口。

示例

- 在集群中创建一个端口号为11810的 coord 节点，指定日志文件大小为64MB

```
oma.createCoord(11810, "/opt/sequoiadb/coord/11810", {logfilesz: 64})
```

oma.createData()

oma.createData(<svcname>,<dbpath>,[config obj])

在 standalone 中创建一个 data 节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是
dbpath	string	节点路径。	是
config obj	Json 对象	节点配置信息，如配置日志大小，是否打开事务等，具体可参考 数据库配置 。	否



注:

- 在一个 standalone 中可以创建多个节点，但是连个节点的端口号必须相差5以上，因为系统为每个节点后台控制了5个通信接口。

示例

- 在 standalone 中创建一个端口号为11820的 data 节点，指定日志文件大小为64MB

```
oma.createData(11820, "/opt/sequoiadb/data/11820", {logfilesz: 64})
```

oma.createOM()

oma.createOM(<svcname>,<dbpath>,[config obj])

创建一个 om 节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是
dbpath	string	节点路径。	是
config obj	Json 对象	节点配置信息，如配置日志大小，是否打开事务等，具体可参考 数据库配置 。	否



注:

- 在一个集群中可以创建多个 om 节点，但是连个节点的端口号必须相差5以上，因为系统为每个节点后台控制了5个通信接口。

示例

- 在集群中创建一个端口号为11830的 om 节点，指定日志文件大小为64MB

```
oma.createOM(11830, "/opt/sequoiadb/om/11830", {logfilesz: 64})
```

oma.removeCoord()

oma.removeCoord(<svcname>)

在集群中删除一个 coord 节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是

注:

- 指定删除的节点必须存在，否则出现异常。

示例

- 在集群中删除一个端口号为11810的 coord 节点

```
oma.removeCoord(11810)
```

oma.removeData()

oma.removeData(<svcname>)

在 standalone 中删除一个 data 节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是

注:

- 指定删除的节点必须存在，否则出现异常。

示例

- 在 standalone 中删除一个端口号为11820的 data 节点

```
oma.removeData(11820)
```

oma.removeOM()

oma.removeOM(<svcname>)

删除一个 om 节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是

注:

- 指定删除的节点必须存在，否则出现异常。

示例

- 在集群中删除一个端口号为11830的 om 节点

```
oma.removeOM(11830)
```

`oma.startNode()`

`oma.startNode(<svcname>)`

启动一个节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是

注:

- 指定启动的节点必须存在，否则出现异常。

示例

- 在集群中启动一个端口号为11830的节点

```
oma.startNode(11830)
```

`oma.stopNode()`

`oma.stopNode(<svcname>)`

停止一个节点。

参数描述

参数名	参数类型	描述	是否必填
svcname	string	节点端口号。	是

注:

- 指定停止的节点必须存在，否则出现异常。

示例

- 在集群中停止一个端口号为11830的节点

```
oma.stopNode(11830)
```

操作符

匹配规则

匹配符	描述	示例
\$gt	大于	db.foo.bar.find({age:\$gt:20})
\$gte	大于等于	db.foo.bar.find({age:\$gte:20})
\$lt	小于	foo.bar.find({age:\$lt:20})
\$lte	小于等于	db.foo.bar.find({age:\$lte:20})
\$ne	不等于	db.foo.bar.find({age:\$ne:20})
\$et	等于	db.foo.bar.find({age:\$et:20})
\$mod	取模	db.foo.bar.find({age:\$mod:[6,5]})
\$in	集合内存在	db.foo.bar.find({age:\$in:[20,21]})
\$isnull	为 null 或不存在	db.foo.bar.find({\$age:\$isnull:1})

匹配符	描述	示例
\$nin	集合内不存在	db.foo.bar.find({age:\$nin:[20,21]})
\$all	全部	db.foo.bar.find({age:\$all:[20,21]})
\$and	与	db.foo.bar.find({\$and:{age:20},name:"Tom"})
\$not	非	db.foo.bar.find({\$not:{age:20},name:"Tom"})
\$or	或	db.foo.bar.find({\$or:{age:20},name:"Tom"})
\$type	数据类型	db.foo.bar.find({age:\$type:16})
\$exists	存在	db.foo.bar.find({age:\$exists:1})
\$elemMatch	元素匹配	db.foo.bar.find({age:\$elemMatch:20})
\$+标识符	数组元素匹配	db.foo.bar.find({"array.\$2":10})
\$size	大小	db.foo.bar.find({array:\$size:3})
\$regex	正则表达式	db.foo.bar.find({str:\$regex:'dh.*fj',\$options:'i'})

更新规则

更新符	描述	信息	示例
\$inc	添加	将指定字段名的值增加给定的值。	db.foo.bar.update({\$inc:{age:25}})
\$set	设置指定字段	将给定字段名设定为给定值。	db.foo.bar.update({\$set:{age:10}})
\$unset	删除指定字段	删除对象中的指定字段。	db.foo.bar.update({\$unset:{age:""}})
\$addToSet	加入集合	如果添加元素不存在于数组中则添加，否则跳过。目标字段必须为数组。	db.foo.bar.update({\$addToSet:{array:[3,4,5]}})
\$pop	消除数组中最后 N 个数值	删除数组中最后 N 个数值，目标字段必须为数组（如果 N 小于 0 意味着从数组起始删除第 -N 个数值）。	db.foo.bar.update({\$pop:{array:2}})
\$pull	消除数值	目标数组中清除给定数值，目标元素必须为数组。	db.foo.bar.update({\$pull:{array:2}})
\$pull_all	消除数组	目标数组中清除给定数组中每个元素的数值，目标元素必须为数组。	db.foo.bar.update({\$pull_all:{array:[2,3,4]}})
\$push	推入数值	将数值插入目标数组，目标元素必须为数组。	db.foo.bar.update({\$push:{array:2}})
\$push_all	推入数组	将给定数组中每一个值插入目标数组，目标元素必须为数组。	db.foo.bar.update({\$push_all:{array:[2,3,4]}})

聚集符

参数名	描述	示例
\$project	选择需要输出的字段名，“1”表示输出，“0”表示不输出，还可以实现字段的重命名	{\$project:{field1:1,field:0,alias:"\$field3"}}
\$match	实现从集合中选择匹配条件的记录，相当与 SQL 语句的 where	{\$match:{field:\$lte:value}}}
\$limit	限制返回的记录条数	{\$limit:10}
\$skip	控制结果集的开始点，即跳过结果集中指定条数的记录	{\$skip:5}
\$group	实现对记录的分组，类似与 SQL 的 group by 语句，“_id”指定分组字段	{\$group:{_id:"\$field"}}
\$sort	实现对结果集的排序，“1”代表升序，“-1”代表降序。	{\$sort:{field1:1,field2:-1,...}}

\$group 聚集符支持以下聚集函数：

函数名	描述
\$addtoset	将指定字段值添加到数组中，相同的字段值只会添加一次
\$first	取分组中第一条记录中的字段值
\$last	取分组中最后一条记录中的字段值
\$max	取分组中字段值最大的
\$min	取分组中字段值最小的
\$avg	取分组中字段值的平均值
\$push	将所有字段添加到数组中，即使数组中已经存在相同的字段值，也继续添加
\$sum	取分组中字段值的总和

👉 注: 在 SequoiaDB 的查询语言中表达相等 (=) 时，使用 JSON{字段名 : 值} 的结构。当然也可以使用 \$eq 操作符

示例：

```
db.collectionspace.collection.find({a: 42}) 等价于 db.collectionspace.collection.find({a: {$eq: 42}})
```

查询集合下符合条件 a 等于42的记录。

匹配符

匹配符	描述	示例
\$gt	大于	db.foo.bar.find({age:{\$gt:20}})
\$gte	大于等于	db.foo.bar.find({age:{\$gte:20}})
\$lt	小于	foo.bar.find({age:{\$lt:20}})
\$lte	小于等于	db.foo.bar.find({age:{\$lte:20}})
\$ne	不等于	db.foo.bar.find({age:{\$ne:20}})
\$in	集合内存在	db.foo.bar.find({age:{\$in:[20,21]}})
\$nin	集合内不存在	db.foo.bar.find({age:{\$nin:[20,21]}})
\$all	全部	db.foo.bar.find({age:{\$all:[20,21]}})
\$and	与	db.foo.bar.find({\$and:[{age:20},{name:"Tom"}]})
\$not	非	db.foo.bar.find({\$not:{age:20},{name:"Tom"}})
\$or	或	db.foo.bar.find({\$or:{age:20},{name:"Tom"}})
\$type	数据类型	db.foo.bar.find({age:{\$type:16}})
\$exists	存在	db.foo.bar.find({age:{\$exists:1}})
\$elemMatch	元素匹配	db.foo.bar.find({age:{\$elemMatch:20}})
\$+标识符	数组元素匹配	db.foo.bar.find({"array.\$2":10})
\$size	大小	db.foo.bar.find({array:{\$size:3}})
\$regex	正则表达式	db.foo.bar.find({str:{\$regex:'dh.*fj'},options:'i'})

\$gt

语法

```
{<字段名>: {$gt: <值>}}
```

描述

\$gt 选择满足“字段名”的值大于 (>) 指定“值”的记录。

示例

- 返回集合 bar 中 age 字段值大于20的记录。

```
db.foo.bar.find({age: {$gt: 20}})
```

- \$gt 匹配嵌套对象中的字段名。使用 [update\(\)](#) 方法更新嵌套对象 service 中的 ID 字段值大于2的记录，将这些记录的 age 字段值设定为25。

```
db.foo.bar.update({$set: {age: 25}}, {"service.ID": {$gt: 2}})
```

\$gte

语法

```
{<字段名>: {$gte: <值>}}
```

描述

\$gte 选择满足“字段名”的值大于等于 (\geq) 指定“值”的记录。

示例

- 选择查询集合空间 foo 下集合 bar 中字段名为 age 的值大于等于20的记录。

```
db.foo.bar.find({age: {$gte: 20}})
```

- \$gte 匹配嵌套对象中的字段名。使用 [update\(\)](#) 方法更新嵌套对象 service 中的 ID 字段值大于等于2的记录，将这些记录的 age 字段值设定为25。

```
db.foo.bar.update({$set: {age: 25}}, {"service.ID": {$gte: 2}})
```

\$lt

语法

```
{<字段名>: {$lt: <值>}}
```

描述

\$lt 选择满足“字段名”的值小于 (<) 指定“值”的记录。

示例

- 查询集合 bar 中字段名为 age , 其值小于20的记录。

```
db.foo.bar.find({age: {$lt: 20}})
```

- \$lt 匹配嵌套对象中的字段名。使用 [update\(\)](#) 方法更新嵌套对象 service 中的 ID 字段值小于2的记录，将这些记录的 age 字段值设定为25。

```
db.foo.bar.update({$set: {age: 25}}, {"service.ID": {$lt: 15}})
```

\$lte

语法

```
{<字段名>: {$lte: <值>}}
```

描述

\$lte 选择满足“字段名”的值小于等于 (\leq) 指定“值”的记录。

示例

- 选择查询集合空间 foo 下集合 bar 中字段名为 age 的值小于等于20的记录。
db.foo.bar.find({age: {\$lte: 20}})
- \$lte 匹配一个嵌套对象中的字段名。使用 [update\(\)](#) 方法更新嵌套对象 service 中的 ID 字段值小于等于2的记录，将这些记录的 age 字段值设定为25。
db.foo.bar.update({\$set: {age: 25}}, {"service.ID": {\$lte: 2}})

\$ne

语法

```
{<字段名>: {$ne: <值>}}
```

描述

\$ne 选择满足“字段名”的值不等于 (!=) 指定“值”的记录。

示例

- 返回集合 bar 中 age 字段值等于 20 的记录。
db.foo.bar.find({age: {\$ne: 20}})
- \$ne 匹配嵌套对象中的字段名。使用 [update\(\)](#) 方法更新嵌套对象 service 中的 type 字段值不等于15的记录，将这些记录的 age 字段值设定为25。
db.foo.bar.update({\$set: {age: 25}}, {"service.type": {\$ne: 15}})

\$et

语法

```
{<字段名>: {$et: <值>}}
```

描述

\$et 选择满足“字段名”的值等于 (=) 指定“值”的记录。

示例

- 返回集合 bar 中 age 字段值等于 20 的记录。
db.foo.bar.find({age: {\$et: 20}}) 等价于 db.foo.bar.find({age: 20})
- \$et 匹配嵌套对象中的字段名。使用 [update\(\)](#) 方法更新嵌套对象 service 中的 type 字段值等于15的记录，将这些记录的 age 字段值设定为25。
db.foo.bar.update({\$set: {age: 25}}, {"service.type": {\$et: 15}})

\$mod

语法

```
{<字段名>: {$mod: [value1, value2]}, ...}
```

描述

\$mod 是取模匹配符，返回指定字段名的值对 value1 取模的值等于 value2 的记录。



注:

- 参数 value1 是除0以外的整型数；如果是浮点型，那只会截取整数部分；不能为其他基础类型。
- 参数 value2 是整型数；如果是浮点型，也只截取整数部分；其他类型以0处理。

示例

- 返回集合 bar 中 age 字段值对5取模后的值等于3的记录。

```
db.foo.bar.find({age: {$mod: [5, 3]}})
返回
{
  "_id": {
    "$oid": "521d5446e2d3c4e31c000000"
  },
  "age": 3
}...
```

```
db.foo.bar.find({age: {$mod: [2, 3, 1, 5]}})
返回
{
  "_id": {
    "$oid": "521d5446e2d3c4e31c000000"
  },
  "age": 3
}
{
  "_id": {
    "$oid": "521d544ee2d3c4e31c000002"
  },
  "age": 5
}
```

对数组[2.3,1.5]中的两个元素只截取了整数部分。

\$in

语法

```
{<字段名>: {$in: [<值1>, <值2>, ... <值N>]}}
```

描述

选择集合中“<字段名>”值匹配给定数组 (`[<值1>, <值2>, ..., <值N>]`) 中任意一个值的记录；如果“<字段名>”本身是数组类型，那么只要满足“<字段名>”中任意一个值等于给定数组 (`[<值1>, <值2>, ..., <值N>]`) 中值的记录都会返回。

示例

- 选择集合 bar 下 age 字段的值是20或25的记录。

```
db.foo.bar.find({age: {$in: [20, 25]}})
```

- \$in 匹配嵌套数组对象中的元素。选择集合 bar 中数组对象 name 存在元素“Tom”或“Mike”的记录，并将这些记录的 age 字段删除。

```
db.foo.bar.update({$unset: {age: ""}}, {name: {$in: ["Tom", "Mike"]}})
```

 注：当给定数组只有一个值时，即`{<字段名>: {$in: [<值>]}}`，等价于`{<字段名>: <值>}`

```
db.foo.bar.find({age: {$in: [20]}}) 等价于 db.foo.bar.find({age: 20})
```

\$isnull

语法

```
{<字段名>: {$isnull: <0|1>}}
```

描述

选择集合中指定的“<字段名>”是否为空，或不存在。“0”代表期望该字段存在且不为 null；“1”代表期望该字段不存在或为 null。

示例

- 选择集合 bar 中 age 字段不为空且存在的记录。

```
db.foo.bar.find({age: {$isnull: 0}})
```

\$nin

语法

```
{<字段名>: {$nin: [<值1>, <值2>, ... <值N>]}}
```

描述

选择集合中“<字段名>”值不等于给定数组 ($[<\text{值1}>, <\text{值2}>, \dots <\text{值N}>]$) 中任意一个值的记录或者不存在给定字段名的记录；如果“<字段名>”本身是数组类型，那么选择“<字段名>”中任意一个值都不等于给定数组 ($[<\text{值1}>, <\text{值2}>, \dots <\text{值N}>]$) 中值的记录。

示例

- 选择集合 bar 下 age 字段的值不等于20和25或集合 bar 下不存在 age 字段的记录。

```
db.foo.bar.find({age: {$nin: [20, 25]}})
```

- `$nin` 匹配数组对象中的元素。选择集合 bar 中存在数组对象 name 且其元素不包含“Tom”和“Mike”或者选择集合 bar 中不存在数组对象 name 的记录，并将这些记录的 age 字段删除。

```
db.foo.bar.update({$unset: {age: ""}}, {name: {$nin: ["Tom", "Mike"]}})
```

 注：当给定数组只有一个值时，即`{<字段名>: {$nin: [<值>]}}`，等价于`{<字段名>: {$ne: <值>}}`

```
db.foo.bar.find({age: {$nin: [20]}}) 等价于 db.foo.bar.find({age: {$ne: 20}})
```

\$all

语法

```
{<字段名>: {$all: [<值1>, <值2>, ... <值N>]}}
```

描述

`$all` 的操作对象是数组类型的字段名，选择“<字段名>”包含所有给定数组 ($[<\text{值1}>, <\text{值2}>, \dots <\text{值N}>]$) 中的值。

示例

- 选择集合 bar 下 name 字段的值包含“Tom”和“Mike”的记录。

```
db.foo.bar.find({name: {$all: ["Tom", "Mike"]}})
```

因此，上面的语句会匹配集合 bar 中有 name 字段，且值形如下面数组的记录：

```
["Tom", "Mike", ...]
["Tom", "Jhon", "Mike", ...]
```

但是不会匹配集合 bar 下 name 字段值形如下面数组的记录

```
["Tom", "Jhon"]
```



注:

使用 \$all 操作一个非数组类型的字段的话，例如：

```
db.foo.bar.find({age: {$all: [20]}}) 它等价于 db.foo.bar.find({age: 20})
```

\$and

语法

```
{$and: [<表达式1>, <表达式2>, ..., <表达式N>]}
```

描述

\$and 是一个逻辑“与”操作。它的作用是选择满足所有表达式 (<表达式1>, <表达式2>, ..., <表达式N>) 的记录，但是如果第一个表达式 (<表达式1>) 的计算结果为 false，SequoiaDB 将不会再执行后面的表达式。

示例

- 选择集合 bar 下 age 字段值为20，price 字段值小于10的记录

```
db.foo.bar.find({$and: [{age: 20}, {price: {$lt: 10}}]})
```



SequoiaDB 提供了一种隐式的 and 操作，用逗号 (,) 隔开个表达式，例如

```
db.foo.bar.find({age: 20, price: {$lt: 10}})
```

- 当使用 and 操作对同一个字段名时，如{age : {\$lt:20}}and{age:{\$exists:1}}。那么可以用 \$and 操作两个分开的表达式，也可以合并这两个表达式{age:{\$lt:20,\$exists:1}}。

```
db.foo.bar.update({$inc: {salary: 200}}, {$and: [{age: {$lt: 20}}, {age: {$exists: 1}}]})  
db.foo.bar.update({$inc: {salary: 200}}, {age: {$lt: 20, $exists: 1}})
```

两个操作的结果相同，首先查询集合 bar 下存在 age 字段并且 age 的值小于20的记录，然后对这些记录的 salary 字段的值增加200。

\$not

语法

```
{$not: [<表达式1>, <表达式2>, ..., <表达式N>]}
```

描述

\$not 是一个逻辑“非”操作。它的作用是选择不匹配表达式 (<表达式1><表达式2>, ..., <表达式N>) 的记录。只要不满足其中的任意一个表达式，记录就会返回。

示例

- 选择集合 bar 下 age 字段值不等于20或 price 字段值不小于10的记录。

```
db.foo.bar.find({$not: [{age: 20}, {price: {$lt: 10}}]})
```

\$or

语法

```
{$or: [<表达式1>, <表达式2>, ..., <表达式N>]}
```

描述

\$or 是一个逻辑“或”操作。它的作用是选择满足表达式 (<表达式1>, <表达式2>, ..., <表达式N>) 其中一个表达式的记录。只要有一个表达式的计算结果为 true，记录就会返回。

示例

- 选择集合 bar 下 name 字段值为“Tom”，且 age 字段值为20或 price 字段值小于10的记录。
`db.foo.bar.find({name: "Tom", $or: [{age: 20}, {price: {$lt: 10}}]})`
- \$or 匹配嵌套对象中的字段名。选择 age 字段值小于20或者嵌套对象 snapshot 中的 type 字段值为“system”的记录，并使用 \$inc 更新这些记录的 salary 字段值。
`db.foo.bar.update({$inc: {salary: 200}}, {$or: [{age: {$lt: 20}}, {"snapshot.type": "system"}]})`

\$type

语法

{<字段名>:{\$type:<BSON type>}}

描述

选择集合中的“<字段名>”值的类型等于指定“<BSON type>”的值。

BSON Type

Type	描述	值
32-bit integer	整型，范围-2147483648至2147483647	16
64-bit integer	长整型，范 围-9223372036854775808至9223372036854775807。 如果用户指定的数值无法适用于整数，则 SequoiaDB 自动将其转化为长整数。	18
double	浮点数，范围1.7E-308至1.7E+308	1
string	字符串	2
ObjectID	十二字节对象 ID	7
boolean	布尔 (true false)	8
date	日期 (YYYY-MM-DD)	9
timestamp	时间戳 (YYYY-MM-DD-HH.mm.ss.fffffff)	17
Binary data	Base64 形式的二进制数据	5
Regular expression	正则表达式	11
Object	嵌套 JSON 文档对象	3
Array	嵌套数组对象	4
null	空	10

示例

- 选择集合 bar 下 age 字段是整型的记录。
`db.foo.bar.find({age: {$type: 16}})`
- 选择集合 bar 下嵌套对象 content 中的 arr 字段是数组类型的记录。
`db.foo.bar.find({"content.arr": {$type: 4}})`

\$exists

语法

{<字段名>:{\$exists:<0|1>}}

描述

选择集合中是否存在指定“<字段名>”的记录。“0”表示选择不存在指定“<字段名>”的记录，“1”表示选择存在指定“<字段名>”的记录。

示例

- 选择集合 bar 中存在字段 age 的记录

```
db.foo.bar.find({age: {$exists: 1}})
```

- 选择集合 bar 中嵌套对象 content 不存在 name 字段的记录

```
db.foo.bar.find({"content.name": {$exists: 0}})
```

\$elemMatch

语法

```
{<字段名>:{$elemMatch:{子字段名:<值>,....}}}
```

描述

选择集合中“<字段名>”匹配指定“{子字段名:<值>,....}”的记录。

示例

- 嵌套 JSON 对象匹配

```
db.foo.bar.find({content: {$elemMatch: {name: "Tom", phone: 123}}})
```

字段 content 是一个 JSON 嵌套对象，此操作匹配 content 内字段 name 值为“Tom”，phone 值为123的记录。

\$+标识符

语法

```
{"<字段名>.$+标识符": value}
```

描述

\$+标识符是一种特殊的命令符，这种命令符只作用于数组对象，标识符是一个整数，如 \$1, \$3，标识符相当于一个临时的存储，会把匹配成功的数组元素的索引存储起来。下面这些是错误的书写格式：\$5.4, \$a2, \$3c, \$MA。

这种命令符只作用于数组，用来代替数组的索引 Key，并且可以把匹配到的第一个索引值传递到方法 [update](#) 的 rule 参数中。

示例

- 查询

有记录 : {a:[1,2,3,4,5]};{a:[1,4,5]};{a:[4,2,1]}现在要查询出数组中存在元素5的记录，使用如下命令

```
db.foo.bar.find({"a.$1": 5}, {a: 1})
```

只要记录中数组对象 a 存在元素5，就能返回。返回结果如下：

```
{ "a": [ 1, 4, 5 ] }
{ "a": [ 1, 2, 3, 4, 5 ] }
```

- 更新

1. 有记录 { a : [1, 2, 3, 4, 5] }，现在要修改数组 a 中的元素，把值为4的元素改成100，使用如下命令

```
db.foo.bar.update({$set: {"a.$1": 100}}, {"a.$1": 4})
```

在匹配时元素4的索引 Key 是3，因此在更新规则 { "\$set": { "a.\$1": 100 } } 中，\$1的值为3，系统会自动把更新规则转换成 { "\$set": { "a.3": 100 } }

更新后记录为：

```
{ a : [ 1, 2, 3, 100, 5 ] }
```

2. 有记录 { a:[1, 2, 3, 4, 5], b:[6, 7, 8] }，现要修改数组 a 中的元素，把值为4的元素改成100，且把数组 b 中值为6的元素修改为200，使用如下命令

```
db.foo.bar.update({ "$set" : { "a.$1" : 100, "b.$2" : 200 } }, { "a.$1": 4, "b.$2": 6 })
```

更新后记录为：

```
{ a : [ 1, 2, 3, 100, 5 ], b : [ 200, 7, 8 ] }
```

 注：如果有多个元素符合规则，那么只会修改第一个。如下例：

3. 有记录 { a:[1, 2, 2, 2, 5] }，现要修改数组 a 中的元素，把值为2的元素改成100，使用如下命令

```
db.foo.bar.update({ "$set" : { "a.$1" : 100 } }, { "a.$1": 2 })
```

更新后记录为：

```
{ a : [ 1, 100, 2, 2, 5 ] }
```

\$size

语法

```
{"<字段名>":{$size:<值>}}
```

描述

\$size 的操作对象为数组型字段，匹配数组长度为指定“<值>”的记录。

示例

- 返回集合 bar 中数组类型字段 arr 的长度为2的记录。

```
db.foo.bar.find({arr: {$size: 2}})
```

\$regex

描述

\$regex 操作提供正则表达式模式匹配字符串查询功能。SequoiaDB 使用的是 PCRE 正则表达式。

 注：\$regex 与 \$options 配套使用。

\$options

\$options 提供四种选择标志：

- i：设置这个修饰符，模式中的字母进行大小写不敏感匹配。
- m：默认情况下，pcre 认为目标字符串是由单行字符组成的，“行首”元字符 (^) 仅匹配字符串的开始位置，而“行末”元字符 (\$) 仅匹配字符串末尾，或者最后的换行符。当这个修饰符设置之后，“行首”和“行末”就会匹配目标字符串中任意换行符之前或之后，另外，还分别匹配目标字符串的最开始和最末尾位置，如果目标字符串中没有“\n”，或者模式中没出现“^”或“\$”，设置这个修饰符不产生任何影响。
- x：设置这个修饰符，模式中没有经过转义的或不在字符类中的空白数据字符总会被忽略，并且位于一个未转义的字符类外部的#字符和下一行换行符之间的字符也被忽略。

- `s`：设置这个修饰符，模式中的点号元字符匹配所有字符，包含换行符，如果没有这个修饰符，点号不匹配换行符。

示例

- 返回集合 `bar` 下 `str` 字段值匹配不区分大小写的正则表达式 `dh.*fj` 的记录

```
db.foo.bar.find({str: {$regex: 'dh.*fj', $options: 'i'}})
```

更新符

更新符	描述	信息	示例
<code>\$inc</code>	添加	将指定字段名的值增加给定的值	<code>db.foo.bar.update({\$inc:{age:25}})</code>
<code>\$set</code>	设置指定字段	将给定字段名设定为给定值	<code>db.foo.bar.update({\$set:{age:10}})</code>
<code>\$unset</code>	删除指定字段	删除对象中的指定字段	<code>db.foo.bar.update({\$unset:{age:""}})</code>
<code>\$addToSet</code>	加入集合	如果添加元素不存在于数组中则添加，否则跳过。目标字段必须为数组。	<code>db.foo.bar.update({\$addToSet:{array:[3,4,5]}})</code>
<code>\$pop</code>	消除数组中最后 N 个数值	删除数组中最后 N 个数值，目标字段必须为数组（如果 N 小于 0 意味着从数组起始删除第 -N 个数值）。	<code>db.foo.bar.update({\$pop:{array:2}})</code>
<code>\$pull</code>	消除数值	目标数组中清除给定数值，目标元素必须为数组	<code>db.foo.bar.update({\$pull:{array:2}})</code>
<code>\$pull_all</code>	消除数组	目标数组中清除给定数组中每个元素的数值，目标元素必须为数组。	<code>db.foo.bar.update({\$pull_all:{array:[2,3,4]}})</code>
<code>\$push</code>	推入数值	将数值插入目标数组，目标元素必须为数组。	<code>db.foo.bar.update({\$push:{array:2}})</code>
<code>\$push_all</code>	推入数组	将给定数组中每一个值插入目标数组，目标元素必须为数组。	<code>db.foo.bar.update({\$push_all:{array:[2,3,4]}})</code>

`$inc`

语法

```
{$inc:{<字段名1>:<值1>,<字段名2>:<值2>,...}}
```

描述

`$inc` 操作是给指定“`<字段名>`”增加指定的“`<值>`”。如果原记录中没有指定的字段名，那将字段名和值填充到记录中；如果原记录中存在指定的字段名，那么将字段名的值加上指定的值。

示例

- 选择集合 `bar` 下 `age` 字段值大于15的记录，然后更新这些记录，将 `age` 字段的值增加5、`ID` 的值添加1。

```
db.foo.bar.update({$inc: {age: 5, ID: 1}}, {age: {$gt: 15}})
```

- 选择集合 `bar` 中存在数组对象 `arr` 的记录，将数组对象 `arr` 的第二个元素值添加1。

```
db.foo.bar.update({$inc: {"arr.1": 1}}, {arr: {$exists: 1}})
```

`$set`

语法

```
{$set:{<字段名1>:<值1>,<字段名2>:<值2>,...}}
```

描述

`$set` 操作是将指定的“<字段名>”更新为指定的“<值>”。如果原记录中没有指定的字段名，那将字段名和值填充到记录中；如果原记录中存在指定的字段名，那么将字段名的值更新为指定的值。

示例

- 选择集合 `bar` 下不存在 `age` 字段的记录，使用 `$set` 更新这些记录

```
db.foo.bar.update({$set: {age: 5, ID: 10}}, {age: {$exists: 0}})
```

- 更新集合 `bar` 下的所有记录，使所有记录的字段 `str` 的值更新为“`abc`”

```
db.foo.bar.update({$set: {str: "abd"}})
```

- 使用 `$set` 更新嵌套数组对象里面的元素。字段名 `arr` 在集合 `bar` 中是一个嵌套数组对象，例如有两条记录：`{arr:[1,2,3],name:"Tom"},{name:"Mike",age:20}` 第二条记录没有 `arr` 字段名

```
db.foo.bar.update({$set: {"arr.1": 4}}, {name: {$exists: 1}})
```

此操作是选择含有 `name` 字段的所有记录，然后使用 `$set` 更新这些记录的数组对象 `arr`。如果原记录中没有数组对象 `arr`，使用 `$set` 会将 `arr` 字段以嵌套对象的方式插入到记录中。上面两条记录更新之后为：

```
{arr: [1, 4, 3], name: "Tom"}, {arr: {"1": 4}, name: "Mike", age: 20}
```

\$unset

语法

```
{$unset:{<字段名1>:"",<字段名2>:"",...}}
```

描述

`$unset` 操作是删除集合中指定的字段名。如果记录中没有指定的字段名，跳过。

示例

- 删除集合 `bar` 下记录的 `name` 字段和 `age` 字段，如果记录中没有字段 `name` 或 `age`，跳过，不做任何处理

```
db.foo.bar.update({$unset: {name: "", age: ""}})
```

- `$unset` 删除数组对象中的元素。如有一条记录：

```
{arr: [1, 2, 3], name: "Tom"}
```

使用 `$unset` 删除第二个元素操作如下：

```
db.foo.bar.update({$unset: {"arr.2": ""}})
```

此操作后，记录更新为

```
{arr: [1, null, 3], name: "Tom"}
```

- `$unset` 删除嵌套对象中的字段。如有一条记录：

```
{content: {ID: 1, type: "system", position: "manager"}, name: "Tom"}
```

`content` 是一个嵌套对象，它有 `ID`，`type`，`position` 三个字段。使用 `$unset` 删除 `type` 字段操作如下：

```
db.foo.bar.update({$unset: {"content.type": ""}})
```

此操作后，记录更新为

```
{content: {ID: 1, position: "manager"}, name: "Tom"}
```

\$addtoset

语法

```
{$addtoset:{<字段名1>:[<值1>,<值2>,...,<值N>] , <字段名2>:[<值1>,<值2>,...,<值N>],...}}
```

描述

`$addtoset` 是向数组对象中添加元素和值，操作对象必须为数组类型的字段。`$addtoset` 有如下规则：

- 记录中有指定的字段名（`<字段名1>,<字段名2>,...`）。

如果指定的值（`[<值1>,<值2>,...,<值N>]`）在记录中存在，跳过不做任何操作，只向目标数组对象中添加不存在的值。
- 记录中不存在指定的字段名。

如果记录本身不存在指定的字段名（`<字段名1>,<字段名2>,...`），那么将指定的字段名和值更新到记录中。

示例

- 记录中存在目标数组对象。如有记录：

```
{arr: [1, 2, 4], age: 10, name: "Tom"}
```

```
db.foo.bar.update({$addtoset: {arr: [1, 3, 5]}}, {arr: {$exists: 1}})
```

此操作后，记录更新为：

```
{arr: [1, 2, 4, 3, 5], age: 10, name: "Tom"}
```

将原记录 arr 数组没有的元素3和5，使用 `$addtoset` 之后更新到 arr 数组内。

- 记录中不存在指定的数组对象，如有记录：

```
{name: "Mike", age: 12}
```

```
db.foo.bar.update({$addtoset: {arr: [1, 3, 5]}}, {arr: {$exists: 0}})
```

此操作后，记录更新为：

```
{arr: [1, 3, 5], age: 12, name: "Mike"}
```

原记录中没有数组对象 arr 字段，`$addtoset` 操作将 arr 字段和值更新到记录中。

\$pop

语法

```
{$pop:{<字段名1>:<N>,<字段名2>:<N>,...}}
```

描述

`$pop` 操作是删除指定数组对象（`<字段名1>,<字段名2>,...`）最后 N 个元素，操作对象必须为数组类型的字段。如果记录中不存在指定的数组对象，跳过不做任何操作；如果指定的 N 值大于数组对象的长度，数组对象的长度更新为0，即它的元素全部被删除；如果指定的 N 值 < 0 ，意味着从数组起始删除第 $-N$ 个元素。

示例

- 删除集合 bar 下数组对象 arr 的最后两个元素。如有记录：

```
{arr: [1, 2, 3, 4], age: 20, name: "Tom"}
```

```
db.foo.bar.update({$pop: {arr: 2}})
```

此操作后，记录更新为：

```
{arr: [1, 2], age: 20, name: "Tom"}
```

- 删除集合 bar 下数组对象 arr 的最后10个元素。如有记录：

```
{arr: [1, 2, 3, 4], age: 20, name: "Tom"}
```

```
db.foo.bar.update({$pop: {arr: 10}})
```

此操作后，记录更新为：

```
{arr: [], age: 20, name: "Tom"}
```

- 删除集合 bar 下数组对象 arr 的前两个元素，即设置N的值为-2。如有记录：

```
{arr: [1, 2, 3, 4], age: 20, name: "Tom"}
```

```
db.foo.bar.update({$pop: {arr: -2}})
```

此操作后，记录更新为：

```
{arr: [3, 4], age: 20, name: "Tom"}
```

\$pull

语法

```
{$pull:{<字段名1>:<值1>,<字段名2>:<值2>,...}}
```

描述

\$pull 清除指定数组对象 (<字段名1>,<字段名2>,...) 的指定值 (<值1>,<值2>,...)。操作对象必须为数组类型的字段。如果记录中不存在指定的数组对象，跳过不做任何操作；如果指定的值不存在数组对象中，也不做任何操作。

示例

- 清除集合 bar 下数组对象 arr 值为2的元素以及数组对象 name 中元素值为“Tom”的元素。如有记录：

```
{arr[1, 2, 4, 5], age: 10, name: ["Tom", "Mike"]}
```

```
db.foo.bar.update({$pull: {arr: 2, name: "Tom"}})
```

此操作后，记录更新为：

```
{arr[1, 4, 5], age: 10, name: ["Mike"]}
```

- 清除集合 bar 下数组对象 arr 中元素值等于2的元素以及数组对象 name 中元素值为“Tom”的元素。如有记录：

```
{arr[1, 3, 4, 5], age: 10, name: ["Tom", "Mike"]}
```

```
db.foo.bar.update({$pull: {arr: 2, name: "Tom"}})
```

此操作后，记录更新为：

```
{arr[1, 3, 4, 5], age: 10, name: ["Mike"]}
```

由于 arr 数组对象没有元素值为2的元素，因此对 arr 对象不做任何操作。

\$pull_all

语法

```
{$pull_all:{<字段名1>:[<值1>,<值2>,...,<值N>],<字段名2>:[<值1>,<值2>,...,<值N>],...}}
```

描述

\$pull_all 清除指定数组对象 (如<字段名1>) 的指定值 ([<值1>,<值2>,...,<值N>])。操作对象必须为数组类型的字段。如果记录中不存在指定的数组对象，跳过不做任何操作；如果指定的值不存在数组对象中，也不做任何操作。

示例

- 清除集合 bar 中数组对象 arr 中值为2和3的元素以及数组对象 name 中元素值为“Tom”的元素。如有记录：

```
{arr[1, 2, 4, 5], age: 10, name: ["Tom", "Mike"]}
```

```
db.foo.bar.update({$pull_all: {arr: [2, 3], name: ["Tom"]}})
```

此操作后，记录更新为：

```
{arr[1, 4, 5], age: 10, name: ["Mike"]}
```

- 删除集合 bar 中数组对象 arr 里面的元素值为4和5的元素。如有记录：

```
{arr[1, 3, 4, 5], age: 10, name: ["Tom", "Mike"]}
```

```
db.foo.bar.update({$pull_all: {arr: [4, 5]}})
```

此操作后，记录更新为：

```
{arr[1, 3], age: 10, name: ["Tom", "Mike"]}
```

\$push

语法

```
{$push:{<字段名1>:<值1>,<字段名2>:<值2>,...}}
```

描述

`$push` 将给定数值（`<值1>`）插入到目标数组（`<字段名1>`）中，操作对象必须为数组类型的字段。如果记录中不存在指定的字段名，将指定的字段名以数组对象的形式推入到记录中并填充其指定的数值；如果记录中存在指定的字段名，且字段名存在指定的数值，指定的数值也会被推入到记录中。

示例

- 向集合 bar 下的 arr 数组对象推入数值1。原记录中 arr 数组对象存在元素1，如有记录：

```
{arr[1, 2, 4], age: 10, name: ["Tom", "Mike"]}
```

```
db.foo.bar.update({$push: {arr: 1}})
```

此操作后，记录更新为：

```
{arr[1, 2, 4, 1], age: 10, name: ["Tom", "Mike"]}
```

虽然原来 arr 中有元素1，使用 `$push` 操作符，还是会将元素1推入到 arr 数组对象中。

- 向集合 bar 中推入不存在的数组对象和值。原记录中不存在数组对象 name，如有记录：

```
{arr[1, 2], age: 20}
```

```
db.foo.bar.update({$push: {name: "Tom"}}, {name: {$exists: 0}})
```

此操作后，记录更新为：

```
{arr[1, 2], age: 20, name: ["Tom"]}
```

原记录中不存在数组对象 name，使用 `$push` 操作符，会将 name 以数组对象的形式推入到记录中。

\$push_all

语法

```
{$push_all:{<字段名1>:[<值1>,<值2>,...,<值N>],<字段名2>:[<值1>,<值2>,...,<值N>],...}}
```

描述

`$push_all` 向指定数组对象（如`<字段名1>`）推入每一个指定值（`[<值1>,<值2>,...,<值N>]`）。操作对象必须为数组类型的字段。如果记录中不存在指定的数组对象，向记录推入指定的数组对象和每一个指定的值（`[<值1>,<值2>,...,<值N>]`）；如果指定的值存在数组对象中，同样被推入到数组对象中。

示例

- 向集合 `bar` 下的 `arr` 数组对象推入`[1,2,8,9]`数组。如有记录：

```
{arr [1, 2, 4, 5], age: 10, name: ["Tom", "Mike"]}
```

```
db.foo.bar.update({$push_all: {arr: [1, 2, 8, 9]}})
```

此操作后，记录更新为：

```
{arr [1, 2, 4, 5, 1, 2, 8, 9], age: 10, name: ["Mike"]}
```

虽然原来记录 `arr` 对象有元素1和2，使用 `$push_all` 操作符，会将`[1,2,8,9]`全部值推入到数组对象 `arr` 中。

- 向集合 `bar` 中推入数组对象 `name`，假设原记录不存在数组对象 `name`。如有记录：

```
{arr [1, 3, 4, 5], age: 10}
```

```
db.foo.bar.update({$push_all: {name: ["Tom", "Jhon"]}}, {name: {$exists: 0}})
```

此操作后，记录更新为：

```
{arr [1, 3, 4, 5], age: 10, name: ["Tom", "Mike"]}
```

聚集符

参数名	描述	示例
<code>\$project</code>	选择需要输出的字段名，“1”表示输出，“0”表示不输出，还可以实现字段的重命名	<code>{\$project:{field1:1,field:0,alias:"\$field3"}}</code>
<code>\$match</code>	实现从集合中选择匹配条件的记录，相当与 SQL 语句的 where	<code>{\$match:{field:{\$lte:value}}}</code>
<code>\$limit</code>	限制返回的记录条数	<code>{\$limit:10}</code>
<code>\$skip</code>	控制结果集的开始点，即跳过结果集中指定条数的记录	<code>{\$skip:5}</code>
<code>\$group</code>	实现对记录的分组，类似与 SQL 的 group by 语句，“_id”指定分组字段	<code>{\$group:{_id:"\$field"}}</code>
<code>\$sort</code>	实现对结果集的排序，“1”代表升序，“-1”代表降序。	<code>{\$sort:{field1:1,field2:-1,...}}</code>

`$group` 聚集符支持以下聚集函数：

函数名	描述
<code>\$addtoset</code>	将指定字段值添加到数组中，相同的字段值只会添加一次
<code>\$first</code>	取分组中第一条记录中的字段值
<code>\$last</code>	取分组中最后一条记录中的字段值
<code>\$max</code>	取分组中字段值最大的
<code>\$min</code>	取分组中字段值最小的
<code>\$avg</code>	取分组中字段值的平均值
<code>\$push</code>	将所有字段添加到数组中，即使数组中已经存在相同的字段值，也继续添加
<code>\$sum</code>	取分组中字段值的总和

\$project

描述

\$project 类似 SQL 中的 select 语句，通过使用 \$project 操作可以从记录中筛选出所需字段，字段名的值如果为1，表示选出，为0表示不选；还可以实现字段的重命名。



注：如果记录不存在所选字段，则以如下格式输出："field":null，field 为不存在的字段名。对嵌套对象使用点操作符（.）引用字段名。

示例

- 使用 \$project 快速地从结果集中选取所需字段

```
db.collectionspace.collection.aggregate({ $project : {title: 0, author: 1}})
```

此操作是选出 author 字段，而 title 字段在结果集中不输出。

- 使用 \$project 重命名字段名，如下：

```
db.collectionspace.collection.aggregate({ $project : {author: 1, name: "$studentName", dep: "$info.department"}})
```

此操作将字段名 studentName 重命名为 name 输出，将 info 对象中的子对象 department 字段重命名为 dep。对嵌套对象，字段引用使用点操作符（.）指向。

- 下面的示例使用 \$project 选择输出字段，然后使用 \$match 按条件匹配记录

```
db.collectionspace.collection.aggregate({ $project: {score: 1, author: 1}}, {$match: {score: {$gt: 80}}})
```

此操作使用 \$project 输出所有记录的 score 和 author 字段值，然后按 \$match 输出匹配条件的记录。



注：由于 \$project 选取了输出字段名，所以 \$match 中字段名必须是 \$project 中选出的字段名。

\$match

描述

\$match 与 find() 方法中的 cond 参数完全相同，通过 \$match 可以实现从集合中选择匹配条件的记录。

\$match 的语法规则请参考读取操作 [find\(\)](#) 方法的 cond 参数介绍。

示例

- 下面的示例使用 \$match 执行简单的匹配

```
db.collectionspace.collection.aggregate({$match: {$and: [{score: 80}, {"info.name": {$exists: 1}}]}})
```

该操作表示从集合 collection 中返回符合条件 score 等于80且 info 对象中的子对象 name 字段存在的记录。

- 下面的示例使用 \$match 匹配符合条件的记录，然后使用 \$group 对结果集分组，最后使用 \$project 输出结果集中指定的字段名

```
db.collectionspace.collection.aggregate({$match: {$and: [{score: 80}, {"info.name": {$exists: 1}}]}}, {$group: {_id: "$major"}}, {$project: {major: 1, dep: 1}})
```

该操作首先集合 collection 中返回符合条件 score 等于80且 info 对象中的子对象 name 字段存在的记录，然后按 major 字段进行分组，最后选择输出结果集中的 major 和 dep 字段。

\$group

描述

\$group 实现对结果集的分组，类似 SQL 中的 group by 语句。首先指定分组键 (_id) ，通过“_id”来标识分组字段，分组字段可以是单个，也可以是多个，格式如下：

单个分组键： {_id: "\$field"}

多个分组键： {_id: {field1: "\$field1", field2: "\$field2", ...}}

 注：使用 \$group 必须指定 _id 字段，当 _id 的值为 null 时，即 {_id:null}，表示不分组。对嵌套对象使用点操作符（.）引用字段名。

示例

- \$group 使用如下

```
db.collectionspace.collection.aggregate({$group: {_id: "$major", avg_score: {$avg: "$score"}, Major: {$first: "$major"}}})
```

该操作表示从集合 collection 中读取记录，并按 major 字段进行分组。在返回的结果集中，取各分组的第一条记录的 major 字段，重命名为 Major；对各分组中的 score 字段值求平均值，重命名为 avg_score。返回如下所示：

```
{
  "avg_score": 82,
  "major": "光学"
}
{
  "avg_score": 77.25,
  "major": "物理学"
}
```

\$group 支持的聚集函数：

函数名	描述
\$addtoset	将字段添加到数组中，相同的字段值只会添加一次
\$first	取分组中第一条记录中的字段值
\$last	取分组中最后一条记录中的字段值
\$max	取分组中字段值最大的
\$min	取分组中字段值最小的
\$avg	取分组中字段值的平均值
\$push	将所有字段添加到数组中，即使数组中已经存在相同的字段值，也继续添加
\$sum	取分组中字段值的总和
\$count	对记录分组后，返回表所有的记录条数

\$addtoset

描述

记录分组后，使用 \$addtoset 将指定字段值添加到数组中，相同的字段值只会添加一次。对嵌套对象使用点操作符（.）引用字段名。

示例

- 如下操作对记录分组后将指定字段值添加到数组中输出

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", dep: {$first: "$dep"}, addtoset_major: {$addtoset: "$major"}}})
```

此操作对记录按 dep 字段值进行分组，并使用 `$first` 输出每个组第一条记录的 dep 字段，输出字段名为 Dep；又将 major 字段的值使用 `$addtoset` 放入数组中返回，输出字段名为 addtoset_major，如下：

```
{
  "Dep": "物电学院",
  "addtoset_major": [
    "物理学",
    "光学",
    "电学"
  ]
}
{
  "Dep": "计算机学院",
  "addtoset_major": [
    "计算机科学与技术",
    "计算机软件与理论",
    "计算机工程"
  ]
}
```

`$first`

描述

记录分组后，取分组中第一条记录指定的字段值，对嵌套对象使用点操作符（.）引用字段名。

示例

- 对记录分组后，输出每个分组第一条记录的指定字段值

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", Dep: {$first: "$dep"}, Name: {$first: "$info.name"}}})
```

此操作对记录按 dep 字段分组，取每个分组中第一条记录的 dep 字段值和嵌套对象 name 字段值，输出字段名分别为 Dep 和 Name，记录返回如下：

```
{
  "Dep": "物电学院",
  "Name": "Lily"
}
{
  "Dep": "计算机学院",
  "Name": "Tom"
}
```

`$last`

描述

记录分组后，取分组中最后一条记录指定的字段值，对嵌套对象使用点操作符（.）引用字段名。

示例

- 对记录分组后，输出每个分组最后一条记录的指定字段值

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", Major: {$addtoset: "$major"}, Name: {$last: "$info.name"}}})
```

此操作对记录按 dep 字段分组，使用 `$last` 取每个分组中最后一条记录嵌套对象 name 字段值，输出字段名为 Name，并且将每个分组中的 major 字段值使用 `$addtoset` 填充到数组中返回，返回字段名为 Major；记录返回如下：

```
{
  "Major": [
    "物理学",
    "光学",
    "电学"
  ]
}
```

```

        "光学",
        "电学"
    ],
    "Name": "Kate"
}
{
    "Major": [
        "计算机科学与技术",
        "计算机软件与理论",
        "计算机工程"
    ],
    "Name": "Jim"
}

```

\$max

描述

记录分组后，取分组中指定字段的最大值返回，对嵌套对象使用点操作符（.）引用字段名。

示例

- 对记录分组后，返回分组中指定字段的最大值

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", max_score: {$max: "$score"}, Name: {$last: "$info.name"}}})
```

此操作对记录按 dep 字段分组，使用 \$max 返回每个分组中 score 字段的最大值，输出字段名为 max_score，又使用 \$last 取每个分组中最后一条记录嵌套对象 name 字段值，输出字段名为 Name。记录返回如下：

```
{
    "max_score": 93,
    "Name": "Kate"
}
{
    "max_score": 90,
    "Name": "Jim"
}
```

\$min

描述

记录分组后，取分组中指定字段的最小值返回，对嵌套对象使用点操作符（.）引用字段名。

示例

- 对记录分组后，返回分组中指定字段的最小值

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", min_score: {$min: "$score"}, Name: {$last: "$info.name"}}})
```

此操作对记录按 dep 字段分组，使用 \$min 返回每个分组中 score 字段的最小值，输出字段名为 min_score，又使用 \$last 取每个分组中最后一条记录嵌套对象 name 字段值，输出字段名为 Name。记录返回如下：

```
{
    "min_score": 72,
    "Name": "Kate"
}
{
    "min_score": 69,
    "Name": "Jim"
}
```

\$avg

描述

记录分组后，取分组中指定字段的平均值返回，对嵌套对象使用点操作符（.）引用字段名。

示例

- 对记录分组后，返回分组中指定字段的平均值

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", avg_age: {$avg: "$info.age"}, max_age: {$max: "$info.age"}, min_age: {$min: "$info.age"}}})
```

此操作对记录按 dep 字段分组，使用 \$avg 返回每个分组中的嵌套对象 age 字段的平均值，输出字段名为 avg_age；又使用 \$min 返回每个分组中嵌套对象 age 字段的最小值，输出字段名为 min_age，使用 \$max 返回每个分组中嵌套对象 age 字段的最大值，输出字段名为 max_age。记录返回如下：

```
{
  "avg_age": 23.727273,
  "max_age": 36,
  "min_age": 15
}
{
  "avg_age": 24.5,
  "max_age": 30,
  "min_age": 20
}
```

\$sum

描述

记录分组后，返回每个分组中指定字段值的总和，对嵌套对象使用点操作符（.）引用字段名。

示例

- 对记录分组后，返回分组中指定字段值的总和

```
db.collectionspace.collection.aggregate({$group: {_id: "$dep", sum_score: {$sum: "$score"}, Dep: {$first: "$dep"}}})
```

此操作对记录按 dep 字段分组，使用 \$sum 返回每个分组中 score 字段值的总和，输出字段名为 sum_score；又使用 \$first 取每个分组中第一条记录的 dep 字段值，输出字段名为 Dep。记录返回如下：

```
{
  "sum_score": 888,
  "Dep": "物电学院"
}
{
  "sum_score": 476,
  "Dep": "计算机学院"
}
```

\$push

描述

记录分组后，使用 \$push 将指定字段值添加到数组中，即使数组中已经存在相同的值，也继续添加。对嵌套对象使用点操作符（.）引用字段名。

示例

- 如下操作对记录分组后将指定字段值添加到数组中输出

```
db.collectionspace.collection.aggregate( {$group: { _id: "$dep", Dep: { $first: "$dep" }, push_age: { $push: "$info.age" } } })
```

此操作对记录按 dep 字段值进行分组，每个分组中嵌套对象 age 字段的值使用 \$push 放入数组中返回，输出字段名为 push_age，如下：

```
{
  "Dep": "物电学院",
  "push_age": [
    28,
    18,
    20,
    30,
    28,
    20
  ]
}
{
  "Dep": "计算机学院",
  "push_age": [
    25,
    20,
    22
  ]
}
```

\$count

描述

记录分组后，用 \$count 取出分组所包含的总记录条数。

示例

- 对记录分组后，返回表所有的记录条数

```
db.foo.bar.count( {$group: { $count: "$dep" } })
{
  "count": 1001
}
```

\$limit

描述

\$limit 实现在结果集中限制返回的记录条数。如果指定的记录条数大于实际的记录总数，那么返回实际的记录总数。

示例

- 限制返回结果集中的前10条记录

```
db.collectionspace.collection.aggregate( { $limit : 10 } )
```

该操作表示集合 collection 中读取前10条记录。

\$sort

描述

\$sort 用来指定结果集的排序规则。对嵌套对象使用点操作符 (.) 引用字段名。

示例

```
db.collectionspace.collection.aggregate({$sort: {score: -1, name: 1}});
```

该操作表示从集合 collection 中读取记录，并以 score 的字段值进行降序排序（1表示升序，-1表示降序）；当记录间 score 字段值相同时，则以 name 字段值进行升序排序。

\$skip

描述

\$skip 参数控制结果集的开始点，即跳过结果集中指定条数的记录。如果跳过的记录数大于总记录数，返回0条记录。

示例

- 跳过10条记录返回

```
db.collectionspace.collection.aggregate( { $skip : 10 } );
```

该操作表示从集合 collection 中读取记录，并跳过前面10条，从第11条记录开始返回。

SQL to Aggregate 映射表

下表主要是描述 SQL 关键字与 SequoiaDB 聚集操作符的对照表。

SQL 关键字	SequoiaDB 聚集操作符
where	\$match
group by	\$group
having	\$match
select	\$project
order by	\$sort
top	\$limit
offset	\$skip

下表主要描述标准 SQL 语句与 SequoiaDB 聚集语句之间的对照。

SQL 语句	SequoiaDB 聚集语句	描述
select product_id as p_id, price from table	db.cs.table.aggregate({\$project: {p_id:"\$product_id",price:1,date:0}})	返回所有记录的 product_id 和 price 字段，其中 product_id 重命名为 p_id，对记录中的 date 字段不返回。
select sum(price) as total from table	db.cs.table.aggregate({\$group:{_id:null,total:{\$sum:"\$price"}}})	对 table 中的字段 price 值求和，并重命名为 total。
select product_id, sum(price) as total from table group by product_id	db.cs.table.aggregate({\$group: {_id:"\$product_id",product_id:{\$first:"\$product_id"},total:{\$sum:"\$price"}}})	对表 table 中的记录按 product_id 字段分组；求每个分组中字段 price 值的累加和，并重命名为 total。
select product_id, sum(price) as total from table group by product_id order by total	db.cs.table.aggregate({\$group: {_id:"\$product_id",product_id:{\$first:"\$product_id"},total:{\$sum:"\$price"}},{\$sort:{total:1}}})	对表 table 中的记录按 product_id 字段分组；求每个分组中字段 price 值的累加和，并重命名为 total；对结果集按字段名 total 的值升序排序。
select product_type_id, product_id, sum(price) as total from table group by product_type_id, product_id	db.cs.table.aggregate({\$group: {_id:{\$product_type_id:"\$product_type_id",product_id:"\$product_id"},{\$first:"\$product_id"},total:{\$sum:"\$price"}}})	对表 table 中的记录按首先按 product_type_id 字段分组，再按 product_id 字段分组；求每个分组中字段 price 值的累加和，并重命名为 total。
select product_id, sum(price) as total from table group by product_id having total > 1000	db.cs.table.aggregate({\$group: {_id:"\$product_id",product_id:{\$first:"\$product_id"},total:{\$sum:"\$price"}},{\$match:{total:{\$gt:1000}}}})	对表 table 中的记录按 product_id 字段分组；求每个分组中字段 price 值的累加和，并重命名为 total；只返回满足条件 total 字段值大于1000的分组。
select product_id, sum(price) as total from table where product_type_id = 1001 group by product_id	db.cs.table.aggregate({\$match: {product_type_id:1001}},{\$group: {_id:"\$product_id",product_id:{\$first:"\$product_id"},total:{\$sum:"\$price"}}})	选择符合条件 product_type_id = 1001 的记录；对选出的记录按 product_id 进行分组；对每个分组中的 price 字段值求和，并重命名为 total。
select product_id, sum(price) as total from table where product_type_id = 1001 group by product_id having total > 1000	db.cs.table.aggregate({\$match: {product_type_id:1001}},{\$group: {_id:"\$product_id",product_id:{\$first:"\$product_id"},total:{\$sum:"\$price"}},{\$match:{total:{\$gt:1000}}}})	选择符合条件 product_type_id = 1001 的记录；对选出的记录按 product_id 进行分组；对每个分组中的 price 字段值求和，并重命名为 total；只返回满足条件 total 字段值大于1000的分组。

SQL 语句	SequoiaDB 聚集语句	描述
select top 10 * from table	db.cs.table.aggregate({\$group:{_id:null}}, {\$limit:10})	返回结果集中的前10条记录。
select * from table offset 50 rows fetch next 10	db.cs.table.aggregate({\$group:{_id:null}}, {\$skip:50}, {\$limit:10})	跳过结果集中前50条记录之后，返回接下来的10条记录。

SQL 语法

SequoiaDB 是一种面向文档型的非关系型数据库，在本节中主要介绍如何使用 SQL 访问和处理 SequoiaDB 数据库系统中的数据。



注: SequoiaDB 中，SQL 语句不区分大小写。

SQL语法表

语句	描述	示例
<code>create collectionspace</code>	创建集合空间	db.executeUpdate("create collectionspace foo")
<code>drop collectionspace</code>	删除集合空间	db.executeUpdate("drop collectionspace foo")
<code>create collection</code>	创建集合	db.executeUpdate("create collection foo.bar")
<code>drop collection</code>	删除集合	db.executeUpdate("drop collection foo.bar")
<code>create index</code>	创建索引	db.executeUpdate("create index test_index on foo.bar (age)")
<code>drop index</code>	删除索引	db.executeUpdate("drop index test_index on foo.bar")
<code>list collectionspaces</code>	枚举集合空间	db.exec("list collectionspaces")
<code>list collections</code>	枚举集合	db.exec("list collections")
<code>insert into</code>	插入	db.executeUpdate("insert into foo.bar(age,name) values(20,�"Tom�)")")
<code>select from</code>	查询	db.exec("select * from foo.bar")
<code>update set</code>	更新	db.executeUpdate("update foo.bar set age=25")
<code>delete from</code>	删除	db.executeUpdate("delete from foo.bar")
<code>group by</code>	分组	db.exec("select dept_no,count(emp_no) as 员工总数 from foo.bar group by dept_no ")
<code>order by</code>	排序	db.exec("select * from foo.bar order by age desc")
<code>split by</code>	记录拆分	
<code>limit</code>	限制返回记录数	db.exec("select * from foo.bar limit 5")
<code>offset</code>	跳过记录数	db.exec("select * from foo.bar offset 5")
<code>as</code>	别名	db.exec("select age as 年龄 from foo.bar where age>10")
<code>inner join</code>	连接	db.exec("select E.emp_no,D.dept_name from foo.emp as E inner join foo.dept as D on E.dept_no=D.dept_no")
<code>left outer join on</code>	左连	db.exec("select E.emp_no,D.dept_name from foo.emp as E left outer join foo.dept as D on E.dept_no=D.dept_no where D.dept_no<4")
<code>right outer join on</code>	右连	db.exec("select E.emp_no,D.dept_name from foo.emp as E right outer join foo.dept as D on E.dept_no=D.dept_no where E.emp_no<10")

语句	描述	示例
<code>count()</code>	计数	<code>db.exec("select count(age) as 数量 from foo.bar")</code>
<code>sum()</code>	求和	<code>db.exec("select sum(age) as 年龄总和 from foo.bar")</code>
<code>avg()</code>	求平均数	<code>db.exec("select avg(age) as 平均年龄 from foo.bar")</code>
<code>max()</code>	最大数	<code>db.exec("select max(age) as 最大年龄 from foo.bar")</code>
<code>min()</code>	最小数	<code>db.exec("select min(age) as 最小年龄 from foo.bar")</code>
<code>first</code>	选择第一条数据	
<code>last</code>	选择最后一条数据	
<code>push</code>	合并数组	
<code>addtoset</code>	合并没有重复值数组	
<code>buildobj</code>	合并对象	
<code>mergearrayset</code>	合并不包含重复字段数组	

sql create collectionspace

create collectionspace 语句

用于创建数据库中的集合空间。

语法

```
create collectionspace <cs_name>
```

<cs_name> : 集合空间名称，集合空间名的最大长度为127Byte，并且不能为空。

示例

- 创建名为 foo 的集合空间

```
db.executeUpdate("create collectionspace foo") //等价于 db.createCS("foo")
```

sql drop collectionspace

drop collectionspace 语句

用于删除数据库中的集合空间。

语法

```
drop collectionspace <cs_name>
```

<cs_name> : 集合空间名，集合空间名必须在数据库中存在。

示例

- 删除名为 foo 的集合空间

```
db.executeUpdate("drop collectionspace foo") //等价于 db.dropCS("foo")
```

sql create collection

create collection 语句

用于创建集合，必须指定集合所在的集合空间。

语法

```
create collection <cs_name>. <cl_name>
```

<cs_name> : 数据库中的集合空间名称。

<cl_name> : 集合名，集合名长度不能超过127Byte，并且不能为空，在同一个集合空间中不能存在相同的集合名。

示例

- 在集合空间foo下创建集合bar。

```
db.executeUpdate("create collection foo.bar") //等价于 db.foo.createCL("bar")
```

sql drop collection

drop collection 语句

用于删除集合空间中的集合。

语法

```
drop collection <cs_name>. <cl_name>
```

<cs_name> : 数据库中的集合空间名，集合空间名必须在数据库中存在；

<cl_name> : 集合名，集合名也必须在指定的集合空间中存在。

示例

- 删除集合空间 foo 中的集合 bar

```
db.executeUpdate("drop collection foo.bar") //等价于 db.foo.dropCL("bar")
```

sql create index

create index 语句

用于在集合中创建索引。在不读取整个集合的情况下，索引使数据库应用程序可以更快地查找数据。

语法

```
create [unique] index <index_name> on <cs_name>. <cl_name> (field1_name [asc|desc],...)
```

[unique] : 标识创建的索引是否唯一。在唯一索引所指定的索引键字段上，集合中不可存在一条以上的记录完全重复。

<index_name> : 索引名称

<cs_name> : 集合空间名称

<cl_name> : 集合名称

field1_name : 创建索引所在的字段名，同一个索引名可以在多个字段名上创建

[asc|desc] : 排序 , asc 表示升序索引某个字段中的值 , desc 表示降序索引某个字段中的值 , 默认为升序。

示例

- 本例会创建一个简单的索引 , 名为“test_index” , 在 foo 集合空间的 bar 集合上的 age 字段 :

```
db.execUpdate("create index test_index on foo.bar (age)")
```

如果希望以降序索引某个字段中的值 , 可以在字段名后面添加保留字 desc :

```
db.execUpdate("create index test_index on foo.bar (age desc)")
```

如果希望索引不止在一个字段上 , 可以在括号中列出这些字段的名称 , 用逗号隔开 :

```
db.execUpdate("create index test_index on foo.bar (age desc, name asc)")
```

- 下面的实例会创建一个唯一索引 :

```
db.execUpdate("create unique index test_index on foo.bar (age)")
```

sql drop index

drop index 语句

用于删除集合中的索引。

语法

```
drop index <index_name> on <cs_name>. <cl_name>
```

<index_name> : 索引名

<cs_name> : 集合空间名

<cl_name> : 集合名

示例

- 删除集合空间 foo 中集合 bar 下名为 test_index 的索引名

```
db.execUpdate("drop index test_index on foo.bar") //等价于  
db.foo.bar.dropIndex("test_index")
```

sql list collectionspaces

list collectionspaces 语句

枚举数据库中的集合空间。

语法

```
list collectionspaces
```

示例

- 本例会返回数据库中的所有集合空间

```
db.exec("list collectionspaces")
```

结果:

```
{
  "Name": "testfoo"
  "Name": "big"
  ...
}
```

sql list collections

list collections 语句

枚举集合空间中的集合。

语法

```
list collections
```

示例

- 本例会返回集合空间中的所有集合

```
db.exec("list collections")
结果:
{
    "Name": "testfoo.testbar"
    "Name": "big.small"
    ...
}
```

sql insert into

insert into 语句

用于向集合中插入新的数据。

语法

```
insert into <cs_name>.<cl_name>(<field1_name>,<field2_name>,...) values(<value1>,<value2>,...)
或者
insert into <cs_name>.<cl_name> <select_set>
```

<cs_name> : 集合空间名

<cl_name> : 集合名

<field_name> : 字段名

<value> : 字段名所对应的值

<select_set> : 查询结果集

示例

- 本例会向集合 bar 中插入一条新的数据 , 字段名为 age 和 name , 对应的值分别为 (25 , "Tom") :

```
db.executeUpdate("insert into foo.bar(age,name) values(25,\\"Tom\\")")
```

- 本例会向集合 bar 中插入批量的数据 , 这些数据为集合 small 中的查询结果集 :

```
db.executeUpdate("insert into foo.bar select * from big.small")
```

sql select

select 语句

用于从集合中选取数据 , 结果被存储在一个结果集中。

语法

```
select * from <cs_name>.<cl_name>
```

或者

```
select <field1_name, field2_name, ...> from <cs_name>. <cl_name>
<cs_name> : 集合空间名
<cl_name> : 集合名
<field_name> : 字段名
```

示例

- 本例会选择指定的字段名返回，如果某条符合条件的记录没有指定的字段名，那么返回它的值为 null：

```
db.exec("select age, name from foo.bar")
结果:
{
  "age": 10,
  "name": null
}
{
  "age": 10,
  "name": "Tom"
}
}...
```

- 本例返回集合中的所有记录的所有字段名

```
db.exec("select * from foo.bar")
结果:
{
  "_id": {
    "$oid": "51c909b0c5b855e029000000"
  },
  "age": 10
}
{
  "_id": {
    "$oid": "51c909b9c5b855e029000001"
  },
  "age": 10,
  "name": "Tom"
}
{
  "_id": {
    "$oid": "51c909c2c5b855e029000002"
  },
  "age": 10,
  "name": "Tom",
  "phone": 123456
}...
```

注:



- 可以选择类似 `where`, `group by`, `order by`, `limit`, `offset` 的关键字对要选择的记录做控制。
 - 如果查询源不为集合，则本层查询中所有字段均需要引用别名（*除外），例如：
- ```
select T.a, T.b from (select * from foo.bar) as T where T.a<10
```
- 子查询必须包含别名，子查询中出现的别名只作用于上一层。

`sql update`

update 语句

用于修改集合中的记录。

## 语法

```
update <cs_name>. <cl_name> set (<field1_name>=<value1>, ...) [where <condition>]
```

<cs\_name> : 集合空间名

<cl\_name> : 集合名

<condition> : 条件 , 只对符合条件的记录更新

## 示例

- 本例会修改集合中全部的记录 , 将记录中的 age 字段值修改为20 , 如果记录中不存在 age 字段 , 则将 age : 20添加到记录中 :

```
db.execUpdate("update foo.bar set age=20")
```

- 本例会修改符合条件的记录 , 只对符合条件 age < 10 的记录做修改操作 :

```
db.execUpdate("update foo.bar set age=20 where age<10")
```

## sql delete

### delete 语句

用于删除集合中的记录。

## 语法

```
delete from <cs_name>. <cl_name> [where <condition>]
```

<cs\_name> : 集合空间名

<cl\_name> : 集合名

<condition> : 条件 , 只对符合条件的记录删除

## 示例

- 本例会删除集合中的所有记录 :

```
db.execUpdate("delete from foo.bar")
```

- 本例会删除符合条件 age <10 的记录 :

```
db.execUpdate("delete from foo.bar where age<10")
```

## sql group by

### group by 语句

用于结合合计函数 , 根据一个或多个字段名对结果集进行分组。

## 语法

```
group by <field1_name> [ASC|DESC], ...>
```

<field\_name> : 字段名

[asc|desc] : 排序 , asc 表示升序 , desc 表示降序 , 默认为 asc

## 示例

- 希望计算每个部门的员工数 , 并按字段名 dept\_no 分组 :

```
db.exec("select dept_no, count(emp_no) as 员工总数 from foo.bar group by dept_no")
```



注:

像 `sum` , `count` , `min` , `max` , `avg` 这样的计数函数必须使用别名。

## sql order by

`order by` 语句

用于根据指定的字段名对结果集进行排序，默认为升序排序。

语法

```
order by <field1_name [ASC|DESC] , ...>
```

`<field_name>` : 字段名

[asc|desc] : 排序，asc 表示升序，desc 表示降序，默认为 asc

示例

- 希望计算每个部门的员工数，并按字段名 `dept_no` 分组，并按字段名的降序排序：

```
db.exec("select dept_no, count(emp_no) as 员工总数 from foo.bar group by dept_no order by dept_no desc")
```



注:

像 `sum` , `count` , `min` , `max` , `avg` 这样的计数函数必须使用别名。

## sql split by

`split by` 语句

按照某个数组字段将记录拆分。

语法

```
split by <field name>
```

示例

- 拆分表中原始记录 { a:1,b:2,c:[3,4,5] }

```
SELECT * FROM foo.bar SPLIT BY c
```

得到结果为：

```
{a: 1, b: 2, c: 3}
{a: 1, b: 2, c: 4}
{a: 1, b: 2, c: 5}
```

## sql limit

`limit`语句

用于限制返回记录个数。

语法

```
limit<limit_num>
```

`<limit_num>` : 限制数

## 示例

- 希望返回集合中前10条记录：

```
db.exec("select * from foo.bar limit 10")
```

## sql offset

### offset语句

用于设置跳过的记录个数。

### 语法

```
offset<offset_num>
```

<offset\_num> : 跳过记录数

### 示例

- 希望跳过前5条记录，从第5条后面开始返回：

```
db.exec("select * from foo.bar offset 5")
```

## sql as

### as语句

用于为集合名或者字段名指定别名 ( alias )。

### 语法

```
<cs_name.cl_name | (select_set) | field_name> AS <alias_name>
```

<cs\_name> : 集合空间名

<cl\_name> : 集合名

select\_set : 结果集

field\_name : 字段名

<alias\_name> : 别名

### 示例

- 集合别名

```
db.exec("select T1.age, T1.name from foo.bar as T1 where T1.age>10")
```

- 字段别名

```
db.exec("select age as 年龄 from foo.bar where age>10")
```

- 结果集别名

```
db.exec("select T.age, T.name from (select age, name from foo.bar) as T where T.age>10")
```

## sql inner join

### inner join语句

用于根据两个或多个集合中的字段名之间的关系，从这些集合中查询数据。

## 语法

```
<collection1_name | (select_set1) as <alias1_name>
inner join
<collection2_name | (select_set2)> as <alias2_name>
[ON condition]
```

## 示例

- 有员工信息表 foo.emp 和部门信息表 foo.dept , 查询员工号 emp\_no 所在的部门名 dept\_name :

```
db.exec("select E.emp_no, D.dept_name from foo.emp as E inner join foo.dept as D on
E.dept_no=D.dept_no")
```



注:

- 不能包含非联合条件 , 如下写法是错误的 :

```
select T1.a, T2.b from foo.bar1 as T1 inner join foo.bar2 as T2 on T1.a<10
```

- 不能在 join 本层使用 select \* 语句。

## sql left outer join

### left outer join 语句

left outer join 会从左边的集合名 ( collection1\_name ) 中返回所有的记录 , 即使在右边的集合名 ( collection2\_name ) 中没有匹配的记录。

## 语法

```
<collection1_name | (select_set1) as <alias1_name>
left outer join
<collection2_name | (select_set2)> as <alias2_name>
[ON condition]
```

## 示例

- 有员工信息表 foo.emp 和部门信息表 foo.dept , 查询员工号 emp\_no 所在的部门名 dept\_name :

```
db.exec("select E.emp_no, D.dept_name from foo.emp as E left outer join foo.dept as D on
E.dept_no=D.dept_no where D.dept_no<4")
```

## sql right outer join

### right outer join 语句

right outer join 会从右边的集合名 ( collection2\_name ) 中返回所有的记录 , 即使在左边的集合名 ( collection1\_name ) 中没有匹配的记录。

## 语法

```
<collection1_name | (select_set1) as <alias1_name>
right outer join
<collection2_name | (select_set2)> as <alias2_name>
[ON condition]
```

## 示例

- 有员工信息表 foo.emp 和部门信息表 foo.dept , 查询员工号 emp\_no < 10 所在的部门名 dept\_name :

```
db.exec("select E.emp_no, D.dept_name from foo.emp as E right outer join foo.dept as D on
E.dept_no=D.dept_no where E.emp_no<10")
```

## sql sum()

sum() 函数

用于求和。

语法

```
sum(field_name) as <alisa_name>
```

注:

1. 使用 sum 函数对字段名求和，必须使用别名。
2. 对非数值型字段自动跳过。

示例

- 对集合 bar 中 age 字段进行求和：

```
db.exec("select sum(age) as 年龄总和 from foo.bar")
```

## sql count()

count() 函数

用于计数，返回匹配指定字段名的条数。

语法

```
count(field_name) as <alisa_name>
```

注: 1. 使用 count 函数对字段名计数，必须使用别名。

示例

- 对集合 bar 中 age 字段进行计数：

```
db.exec("select count(age) as 数量 from foo.bar")
```

## sql avg()

avg() 函数

用于求指定字段名的平均值。

语法

```
avg(field_name) as <alisa_name>
```

注:

1. 使用 avg 函数对字段名求平均值，必须使用别名。
2. 对非数值型字段自动跳过。

## 示例

- 对集合 bar 中 age 字段进行求平均值：

```
db.exec("select avg(age) as 平均年龄 from foo.bar")
```

## sql max()

### max() 函数

用于返回指定字段名的最大值。

#### 语法

```
max(field_name) as <alias_name>
```



注: 1. 使用 max 函数返回字段名的最大值时，必须使用别名。

## 示例

- 对集合 bar 中 age 字段返回最大值：

```
db.exec("select max(age) as 最大年龄 from foo.bar")
```

## sql min()

### min() 函数

用于返回指定字段名的最小值。

#### 语法

```
min(field_name) as <alias_name>
```



注: 1. 使用 min 函数返回字段名的最小值时，必须使用别名。

## 示例

- 对集合 bar 中 age 字段返回最小值：

```
db.exec("select min(age) as 最小年龄 from foo.bar")
```

## sql first()

### first() 函数

选择范围内第一条数据。

#### 语法

```
first(field_name)
```

## 示例

- 选择表中第一条数据

表中原始记录  
{a: 1, b: 2}  
{a: 2, b: 3}

```
{a: 3, b: 3}

SELECT FIRST(a) AS a, b FROM foo.bar GROUP BY b

得到记录
{a: 1, b: 2}
{a: 2, b: 3}
```

## sql last()

### last() 函数

选择范围内最后一条数据。

#### 语法

```
last(field name)
```

#### 示例

- 选择表中最后一条数据

表中原始记录

```
{a: 1, b: 2}
{a: 2, b: 3}
{a: 3, b: 3}

SELECT LAST(a) AS a, b FROM foo.bar GROUP BY b

得到记录
{a: 1, b: 2}
{a: 3, b: 3}
```

## sql push()

### push() 函数

将多个记录中的字段合并为一个数组。

#### 语法

```
push(field name)
```

#### 示例

- 将表中多个记录中的字段合并为一个数组

表中原始记录

```
{a: 1, b: 1}
{a: 2, b: 2}
{a: 2, b: 3}

SELECT a, PUSH(b) AS b FROM foo.bar GROUP BY a

得到记录
{a: 1, b: [1]}
{a: 2, b: [2, 3]}
```

## sql addtoset()

### addtoset() 函数

将多个记录中的字段合并为一个没有重复值的数组。

#### 语法

```
addtoset(field_name)
```

#### 示例

- 将表中多个记录中的字段合并为一个没有重复值的数组

表中原始记录

```
{a: 1, b: 1}
{a: 2, b: 2}
{a: 2, b: 3}
{a: 2, b: 3}
```

```
SELECT a, ADDTOSET(b) AS b FROM foo.bar GROUP BY a
```

得到记录

```
{a: 1, b: [1]}
{a: 2, b: [2, 3]}
```

## sql buildobj()

### buildobj() 函数

将记录中多个字段合并为一个对象。

#### 语法

```
buildobj(field_name1, fieldname2, ...)
```

#### 示例

- 将表中记录中多个字段合并为一个对象

表中原始记录

```
{a: 1, b: 1, c: 1}
{a: 2, b: 2, c: 2}
{a: 3, b: 3, c: 3}
```

```
SELECT a, buildobj(b, c) AS d FROM foo.bar
```

得到记录

```
{a: 1, d: {b: 1, c: 1}}
{a: 2, d: {b: 2, c: 2}}
{a: 3, d: {b: 3, c: 3}}
```

## sql mergearrayset()

### mergearrayset() 函数

将多个数组字段合并为一个不包含重复字段的数组。

## 语法

```
mergearrayset (field name)
```

## 示例

- 将表中多个数组字段合并为一个不包含重复字段的数组

表中原始记录

```
{a: 1, b: [1, 2, 3]}
{a: 1, b: [2, 2, 3]}
```

```
SELECT a, MERGEARRAYSET (b) AS b FROM foo.bar GROUP BY a
```

得到记录

```
{a: 1, b: [1, 2, 3]}
```

## SQL to SequoiaDB 映射表

### 概念和术语

| SQL                        | SequoiaDB                                                  |
|----------------------------|------------------------------------------------------------|
| database                   | database                                                   |
| table                      | collection                                                 |
| row                        | document / BSON document                                   |
| column                     | field                                                      |
| index                      | index                                                      |
| table joins                | embedded documents                                         |
| primary key (指定任何唯一的列作为主键) | primary key (在 SequoiaDB 中，primary key 是自动创建到记录的 _id 字段名中) |

### Create 和 Alter

下表列出了各种 SQL 语句表级别的操作和在 SequoiaDB 中对应的操作：

| SQL 语句                                                                             | SequoiaDB 语句                                                                                                                                           | 相关链接                                |
|------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------|
| create table student (id not null, stu_id varchar(50), age number primary key(id)) | 在第一次做插入操作时隐式创建集合，如果未指定 _id 字段，_id 字段自动添加<br>db.collectionspace.student({stu_id:"01",age:20})，当然你也可以明确的创建一个集合<br>db.collectionspace.createCL("student") | <a href="#">insert(),createCL()</a> |
| alter table student add name varchar(50)                                           | 集合不描述或强制执行文档的结构，即在集合上没有结构的改动操作，但是 update() 方法可以使用 \$set 向文档记录添加不存在的字段。<br>db.collectionspace.student.update({},{\$set:{name:"Tom"}})                   | <a href="#">update(),\$set</a>      |
| alter table student drop column name                                               | 集合不描述或强制执行文档的结构，即在集合上没有结构的改动操作，但是 update() 方法可以使用 \$unset 向文档记录删除存在的字段。<br>db.collectionspace.student.update({},{\$unset:{name:"Tom"}})                | <a href="#">update(),\$unset</a>    |
| create index index_stu_id on student (stu_id)                                      | db.collectionspace.student.createIndex("index_stu_id",{stu_id:-1})                                                                                     | <a href="#">createIndex(),index</a> |
| drop table student                                                                 | db.collectionspace.dropCL("student")                                                                                                                   | <a href="#">dropCL()</a>            |

## Insert

下表给出了各种 SQL 语句在表级上的插入操作和 SequoiaDB 上相应的操作：

| SQL 语句                                          | SequoiaDB 语句                                            | 相关链接                     |
|-------------------------------------------------|---------------------------------------------------------|--------------------------|
| insert into student(stu_id,age) values("01",20) | db.collectionspace.student.insert({stu_id:"01",age:20}) | <a href="#">insert()</a> |

## Select

下表给出了各种 SQL 语句在表级上的读操作和 SequoiaDB 上相应的操作：

| SQL 语句                                                  | SequoiaDB 语句                                                   | 相关链接                             |
|---------------------------------------------------------|----------------------------------------------------------------|----------------------------------|
| select * from student                                   | db.collectionspace.student.find()                              | <a href="#">find()</a>           |
| select stu_id, age from student                         | db.collectionspace.student.find({}, {stu_id:"01",age:20})      | <a href="#">find()</a>           |
| select * from student where age > 25                    | db.collectionspace.student.find({age:{\$gt:25}})               | <a href="#">find(),\$gt</a>      |
| select age from student where age = 25 and stu_id= "01" | db.collectionspace.student.find({age:25,stu_id:"01"},{age:25}) | <a href="#">find()</a>           |
| select count(*) from student                            | db.collectionspace.student.count()                             | <a href="#">count()</a>          |
| select count(stu_id) from student                       | db.collectionspace.student.count({stu_id:{\$exists:1}})        | <a href="#">count(),\$exists</a> |

## Update

下表给出了各种 SQL 语句在表级上的更新操作和 SequoiaDB 上相应的操作：

| SQL 语句                                               | SequoiaDB 语句                                                       | 相关链接                           |
|------------------------------------------------------|--------------------------------------------------------------------|--------------------------------|
| update student set age = 25 where stu_id = "01"      | db.collectionspace.student.update({stu_id:"01"}, {\$set:{age:25}}) | <a href="#">update(),\$set</a> |
| update student set age = age + 2 where stu_id = "01" | db.collectionspace.student.update({stu_id:"01"}, {\$inc:{age:2}})  | <a href="#">update(),\$inc</a> |

## Delete

下表给出了各种 SQL 语句在表级上的删除记录操作和 SequoiaDB 上相应的操作：

| SQL 语句                             | SequoiaDB 语句                                | 相关链接                     |
|------------------------------------|---------------------------------------------|--------------------------|
| delete from student where age = 20 | db.collectionspace.student.remove({age:20}) | <a href="#">remove()</a> |
| delete from student                | db.collectionspace.student.remove()         | <a href="#">remove()</a> |

## 限制

---

### 文档

|        |                         |
|--------|-------------------------|
| 描述     | 限制                      |
| 文档最小长度 | 至少包含一个字段                |
| 文档最大长度 | 转为 BSON 结构后 16777168 字节 |
| 字段名    | 不以“\$”起始，不包含“.”         |

### 集合

|         |                       |
|---------|-----------------------|
| 描述      | 限制                    |
| 集合名最大长度 | 127 字节                |
| 集合名     | 不以“\$”或“SYS”起始，不包含“.” |

|             |           |
|-------------|-----------|
| 描述          | 限制        |
| 单节点集合最大容量   | 为集合空间最大容量 |
| 单集合空间集合最大数量 | 4096      |

### 集合空间

|             |                                    |
|-------------|------------------------------------|
| 描述          | 限制                                 |
| 集合空间名最大长度   | 127字节                              |
| 集合空间名       | 不以“\$”或“SYS”起始，不包含“.”              |
| 数据页大小       | 4096、8192、16384、32768、65536        |
| 单节点集合空间最大容量 | 对应每种数据页大小，分别为512GB、1TB、2TB、4TB、8TB |
| 单节点集合空间最大数量 | 4096                               |

### 索引

|                      |                           |
|----------------------|---------------------------|
| 描述                   | 限制                        |
| 每条数据索引键最大长度          | 1024字节                    |
| 索引定义总长度（包括索引名，索引键名等） | 转为 BSON 后小于等于数据页大小-48字节   |
| 复合索引                 | 文档里符合索引所定义的字段中，最多一个字段包含数组 |
| 索引键定义排序值             | 1或者-1                     |
| 单集合最大索引数量            | 64                        |

### 数据库

|        |      |
|--------|------|
| 描述     | 限制   |
| 日志文件最小 | 64MB |
| 日志文件最大 | 2GB  |

### 节点

|            |                             |
|------------|-----------------------------|
| 描述         | 限制                          |
| 每分区组最大节点数量 | 7                           |
| 创建节点       | 必须使用 hostname，而不是 IP 地址     |
| 网络         | 集群中所有系统必须能够使用 hostname 互相访问 |
| 主节点选举条件    | 分区组内至少存在超过半数节点参与选举          |

### 分区

|      |                             |
|------|-----------------------------|
| 描述   | 限制                          |
| 数据切分 | 同一时刻每个集合只能进行一个范围的切分         |
| 分区键  | 分区键数值在数据插入后不可修改             |
| _id  | 分区集合中 _id 仅保证分区组内唯一，不保证全局唯一 |
| 唯一索引 | 必须包含分区键中所有字段                |

### 驱动

|      |                                   |
|------|-----------------------------------|
| 描述   | 限制                                |
| 线程安全 | 每个连接对象与其下属的子对象为非线程安全不同连接对象之间为线程安全 |

## Error Code List

| Description     | Error Code |
|-----------------|------------|
| IO 错误           | -1         |
| 无可用内存           | -2         |
| 权限错误            | -3         |
| 文件不存在           | -4         |
| 文件已存在           | -5         |
| 非法输入参数          | -6         |
| 非法长度            | -7         |
| 中断错误            | -8         |
| 文件结束            | -9         |
| 系统错误            | -10        |
| 无剩余空间           | -11        |
| 引擎调度单元状态错误      | -12        |
| 超时错误            | -13        |
| 数据库已暂停          | -14        |
| 网络错误            | -15        |
| 网络已从远程关闭        | -16        |
| 数据库正在关闭         | -17        |
| 应用被强制退出         | -18        |
| 非法路径错误          | -19        |
| 非预期文件类型         | -20        |
| 存储单元中无可用空间      | -21        |
| 集合已存在           | -22        |
| 集合不存在           | -23        |
| 数据记录过大          | -24        |
| 数据记录不存在         | -25        |
| 溢出记录已存在         | -26        |
| 非法记录            | -27        |
| 存储单元需要重组        | -28        |
| 集合结尾            | -29        |
| 上下文已打开          | -30        |
| 上下文已关闭          | -31        |
| 选项暂不支持          | -32        |
| 集合空间已存在         | -33        |
| 集合空间不存在         | -34        |
| 非法存储单元          | -35        |
| 上下文不存在          | -36        |
| 超过一个索引字段包含数组    | -37        |
| 索引键已存在          | -38        |
| 索引键过大           | -39        |
| 索引块无空间          | -40        |
| 索引键不存在          | -41        |
| 最大数量索引已存在       | -42        |
| 初始化索引失败         | -43        |
| 集合已被删除          | -44        |
| 两条记录拥有同样的键值和RID | -45        |
| 同名索引已存在         | -46        |
| 索引不存在           | -47        |
| 非预期索引状态         | -48        |
| 索引结束            | -49        |
| 去重缓冲区已满         | -50        |
| 非法谓词            | -51        |
| 索引不存在           | -52        |
| 索引提示非法          | -53        |
| 无可用临时集合         | -54        |

| Description   | Error Code |
|---------------|------------|
| 存储空间数量已最大     | -55        |
| \$id索引不可被删除   | -56        |
| 日志不在缓冲区内      | -57        |
| 日志不在文件中       | -58        |
| 复制组不存在        | -59        |
| 复制组已存在        | -60        |
| 非法请求ID        | -61        |
| 会话ID不存在       | -62        |
| 系统引擎调度单元不可被终止 | -63        |
| 数据库未连接        | -64        |
| 非预期结果         | -65        |
| 记录损坏          | -66        |
| 备份已开始         | -67        |
| 备份未结束         | -68        |
| 备份正在进行        | -69        |
| 备份文件损坏        | -70        |
| 主节点不存在        | -71        |
| 请求节点不存在       | -72        |
| 引擎帮助参数        | -73        |
| 非法连接状态        | -74        |
| 非法句柄          | -75        |
| 对象已释放或不存在     | -76        |
| 监听端口已被占用      | -77        |
| 无法监听端口        | -78        |
| 无法连接到指定地址     | -79        |
| 连接不存在         | -80        |
| 发送失败          | -81        |
| 定时器标示不存在      | -82        |
| 路由信息不存在       | -83        |
| 消息错误          | -84        |
| 非法网络句柄        | -85        |
| 重组文件不合法       | -86        |
| 重组文件为只读模式     | -87        |
| 集合状态非法        | -88        |
| 集合不为重组状态      | -89        |
| 复制组未激活        | -90        |
| 非法复制组成员       | -91        |
| 集合状态不兼容       | -92        |
| 存储单元版本不兼容     | -93        |
| 本地组版本信息过期     | -94        |
| 非法数据页大小       | -95        |
| 远程组版本信息过期     | -96        |
| 投票失败          | -97        |
| 日志记录损坏        | -98        |
| LSN超出边界       | -99        |
| 未知消息          | -100       |
| 更新信息无变化       | -101       |
| 未知消息          | -102       |
| 空栈错误          | -103       |
| 非主节点          | -104       |
| 数据节点不足        | -105       |
| 数据节点不存在编目信息   | -106       |
| 数据节点编目版本过旧    | -107       |
| 协调节点编目版本过旧    | -108       |
| 超出最大组上限       | -109       |
| 同步日志失败        | -110       |
| 执行日志失败        | -111       |
| HTTP头结构错误     | -112       |

| Description     | Error Code |
|-----------------|------------|
| 协商失败            | -113       |
| 日志元数据移动失败       | -114       |
| 数据文件空间管理段错误     | -115       |
| 应用程序中断          | -116       |
| 应用程序断开连接        | -117       |
| 字符编码错误          | -118       |
| 协调节点查询失败        | -119       |
| 缓冲区数组满          | -120       |
| 子上下文冲突          | -121       |
| 协调节点接收到集合结尾消息   | -122       |
| 日志文件大小不统一       | -123       |
| 日志文件不可识别        | -124       |
| 无可用资源           | -125       |
| 非法LSN编号         | -126       |
| 命名管道请求发送的数据过大   | -127       |
| 编目授权错误          | -128       |
| 节点间正在全量同步       | -129       |
| 协调节点分配数据节点失败    | -130       |
| PHP驱动内部错误       | -131       |
| 协调节点发送失败        | -132       |
| 节点组信息不存在        | -133       |
| 远程节点断开连接        | -134       |
| 无法找到匹配的协调节点信息   | -135       |
| 更新协调节点失败        | -136       |
| 未知操作请求          | -137       |
| 协调节点无法找到本地节点组信息 | -138       |
| DMS数据块损坏        | -139       |
| 远程集群管理失败        | -140       |
| 远程引擎已部分被停止      | -141       |
| 服务正在启动          | -142       |
| 服务已经启动          | -143       |
| 服务正在重启          | -144       |
| 节点已存在           | -145       |
| 节点不存在           | -146       |
| 锁定失败            | -147       |
| DMS状态与当前请求不兼容   | -148       |
| 数据库重建已开始        | -149       |
| 数据库重建正在进行       | -150       |
| 协调节点的缓存中无数据     | -151       |
| 求值过程发生错误        | -152       |
| 分区组已经存在         | -153       |
| 分区组不存在          | -154       |
| 节点不存在           | -155       |
| 启动节点失败          | -156       |
| 节点配置冲突          | -157       |
| 空分区组            | -158       |
| 该操作仅适用于协调节点     | -159       |
| 在节点上执行操作失败      | -160       |
| 已经存在互斥任务        | -161       |
| 指定任务不存在         | -162       |
| 系统集合数据损坏        | -163       |
| \$shard索引不可被删除  | -164       |
| 该节点不能运行该命令      | -165       |
| 该服务平面不能运行该命令    | -166       |
| 该Group信息不存在     | -167       |
| Group名称冲突       | -168       |
| 非分区集合           | -169       |
| 记录不包含合法的分区键     | -170       |

| Description         | Error Code |
|---------------------|------------|
| 存在一个不兼容的任务          | -171       |
| 集合在指定复制组中不存在        | -172       |
| 指定的任务不存在            | -173       |
| 记录包含超过一条分区键         | -174       |
| 已存在互斥任务             | -175       |
| 指定的分区键不合法或不在源节点范围内  | -176       |
| 唯一索引必须包含分区键中的所有字段   | -177       |
| 分区键不可被更新            | -178       |
| 没有权限                | -179       |
| 编目节点地址未指定           | -180       |
| 当前记录已被删除            | -181       |
| 搜索条件无法满足任何条件的匹配     | -182       |
| 索引页重组后指定位置存在不同左子节点  | -183       |
| 记录内含有重复字段名          | -184       |
| 插入操作尝试写入过多数据        | -185       |
| 合并连接只接受相等谓词         | -186       |
| 跟踪已经启动              | -187       |
| 跟踪缓冲区不存在            | -188       |
| 跟踪文件不合法             | -189       |
| 请求的事务锁不兼容           | -190       |
| 系统正在执行回滚操作          | -191       |
| 导入数据库时遇到无效的记录       | -192       |
| 发现相同的变量名            | -193       |
| 列名存在歧义              | -194       |
| SQL中存在语法错误          | -195       |
| 无效的事务操作             | -196       |
| 加入锁的等待队列中           | -197       |
| 记录已被删除              | -198       |
| 索引被删除或非法状态          | -199       |
| 重复创建编目节点集群          | -200       |
| 解析json文件错误          | -201       |
| 解析CSV文件错误           | -202       |
| 日志文件超长              | -203       |
| 不能删除组内唯一的节点         | -204       |
| 需要手工完成清理工作          | -205       |
| 系统存在其它组时不能删除编目节点和组  | -206       |
| 分区组不存在              | -207       |
| 无法删除存在非空组           | -208       |
| 到达Queue队列结尾         | -209       |
| 集合不存在分区键索引,不能按百分比分区 | -210       |
| 指定参数字段不存在           | -211       |
| 跟踪断点数量过多            | -212       |
| 预取器繁忙               | -213       |
| 域不存在                | -214       |
| 域已存在                | -215       |
| 组不存在指定域中            | -216       |
| 分区类型不为哈希            | -217       |
| 分区百分比过低             | -218       |
| 后台任务已完成             | -219       |
| 集合已处于装载状态           | -220       |
| 进行装载操作回滚            | -221       |
| RouteID与当前节点不一致     | -222       |
| 服务已经存在              | -223       |
| 未找到字段               | -224       |
| csv字段行结束            | -225       |
| 未知的文件类型             | -226       |
| 部分节点导出配置文件失败        | -227       |
| 非主空节点               | -228       |

| Description      | Error Code |
|------------------|------------|
| 索引文件特征值与数据文件不匹配  | -229       |
| 引擎版本参数           | -230       |
| 客户端帮助参数          | -231       |
| 客户端版本参数          | -232       |
| 存储过程不存在          | -233       |
| 非法删除集合分区         | -234       |
| 重复关联集合分区         | -235       |
| 无效的分区集合          | -236       |
| 新增区间与现有区间冲突      | -237       |
| 新增区间不合法          | -238       |
| 达到高水位            | -239       |
| 该备份已存在           | -240       |
| 该备份不存在           | -241       |
| 无效的集合分区          | -242       |
| 后台任务被取消          | -243       |
| 分区集合的分区类型必须是范围分区 | -244       |
| 未包含合法的分区键字段      | -245       |
| 分区集合不支持此操作       | -246       |
| 重复定义索引           | -247       |
| 正在删除CS           | -248       |
| 节点数量达到上限         | -249       |
| 节点处于业务故障状态       | -250       |
| 节点信息过期           | -251       |
| 等待备节点同步该操作失败     | -252       |
| 未开启事务功能          | -253       |
| 客户端连接池已满         | -254       |
| 文件描述符已达到上限       | -255       |
| 域非空              | -256       |
| REST接收的数据大小超过最大值 | -257       |
| 构建bson失败         | -258       |
| 存储过程参数越界         | -259       |
| 未知的REST命令        | -260       |
| 在数据节点上执行命令失败     | -261       |
| 域中不包含任何数据组       | -262       |
| 提示用户修改登陆密码       | -263       |
| 部分节点未返回成功        | -264       |
| 不同版本的OMAgent已运行  | -265       |
| 无法找到后台任务信息       | -266       |
| 后台任务正在回滚         | -267       |
| 大对象的序列不存在        | -268       |
| 大对象处于不可用状态       | -269       |
| 数据格式非UTF-8编码     | -270       |
| 后台任务失败           | -271       |