

---

# KingbaseES 数据库管理员指南

---

## 金仓数据库管理系统 KingbaseES

文档版本：V8（V008R006C007B0024）

发布日期：2023 年 4 月 1 日

**KING BASE<sup>®</sup>**

人大金仓

北京人大金仓信息技术股份有限公司

# 目 录

第 1 章 前言	1
1.1 适用读者	1
1.2 相关文档	1
1.3 术语	2
1.4 手册约定	3
第 2 章 数据库管理入门	5
2.1 KingbaseES 数据库的用户类型	5
2.1.1 数据库管理员	5
2.1.2 安全管理员	6
2.1.3 应用开发人员	6
2.1.4 应用管理员	6
2.1.5 数据库用户	6
2.2 数据库管理员的任务	7
2.2.1 评估数据库使用的硬件资源	7
2.2.2 安装数据库软件	7
2.2.3 数据库规划	7
2.2.4 创建数据库	7
2.2.5 备份数据库	8
2.2.6 创建用户	8
2.2.7 实施数据库设计	8
2.2.8 备份功能完整的数据库	8
2.2.9 性能调优	8
2.3 SQL 语句	8
2.3.1 向数据库提交命令和 SQL	9
2.3.2 关于 Ksql	9
2.3.3 连接数据库	9
2.3.4 关于使用 Ksql 连接到数据库	9
2.3.5 Step 1: 打开命令窗口	10
2.3.6 Step 2: 设置操作系统环境变量	10
2.3.7 Step 3: 启动 Ksql	10

2.3.8	Step 4: Ksql 连接到 KingbaseES 数据库	10
2.3.8.1	Ksql 连接数据库命令语法	11
2.4	确定数据库的软件版本	11
2.4.1	KingbaseES 数据库的版本信息	11
2.4.2	获取当前数据库的版本信息	11
2.5	数据库管理员的安全和权限	11
2.5.1	管理员用户	11
2.5.2	管理员权限	12
2.6	数据库管理员认证	12
2.7	数据操作工具	13
2.7.1	数据加载工具	13
2.7.2	数据导出导入工具	13
<b>第 3 章</b>	<b>管理进程</b>	<b>14</b>
3.1	关于独立服务进程和共享服务进程	14
3.2	管理 KingbaseES 的独立服务进程	14
3.3	关于 KingbaseES 的后台进程	14
3.4	终止会话	15
3.4.1	指定需要终止的会话	15
3.4.2	终止会话	15
3.4.3	取消会话执行的 SQL 语句	15
3.5	进程和会话视图	16
<b>第 4 章</b>	<b>管理内存</b>	<b>17</b>
4.1	关于内存管理	17
4.2	内存结构概述	17
4.3	手动配置内存	18
4.3.1	系统全局区配置	18
4.3.2	进程全局区配置	19
<b>第 5 章</b>	<b>用户管理和数据库安全</b>	<b>21</b>
5.1	数据库在使用之前设置安全策略的重要性	21
5.2	管理用户	22
5.3	用户的权限和角色	22
5.4	审计数据库活动	23
5.5	预定义的账户	23
<b>第 6 章</b>	<b>监控数据库</b>	<b>24</b>
6.1	监控错误和警报	24
6.1.1	使用 kbbadger	24
6.2	监控数据库性能	28
6.2.1	监控锁	28
6.2.1.1	使用 sys_locks	29

6.2.1.2	监控死锁	29
6.2.2	监控等待事件	30
6.2.2.1	等待事件	31
6.2.3	性能监控视图	33
<b>第 7 章</b>	<b>管理控制文件</b>	<b>37</b>
7.1	什么是控制文件?	37
7.2	控制文件准则	38
7.2.1	控制文件路径	38
7.2.2	控制文件的大小	38
7.3	创建控制文件	38
7.3.1	控制文件内部信息的创建	38
7.4	备份控制文件	39
7.5	读取控制文件信息	39
<b>第 8 章</b>	<b>管理重做日志</b>	<b>40</b>
8.1	什么是重做日志?	40
8.1.1	WalWriter 进程	41
8.1.2	重做日志的内容	41
8.1.3	KingbaseES 数据库如何写入重做日志	41
8.1.3.1	回收和未回收的重做日志	42
8.1.3.2	日志切换与日志序列号	42
8.2	规划重做日志	42
8.2.1	重做日志段文件的大小	42
8.2.2	重做日志文件的块大小	43
8.2.3	重做日志文件的数量	43
8.2.3.1	max_wal_size	43
8.2.3.2	min_wal_size	43
8.2.4	重做日志记录延迟拷贝	44
8.3	手动切换重做日志	44
8.4	验证重做日志文件中的块	44
8.5	清除重做日志文件	45
<b>第 9 章</b>	<b>管理归档重做日志文件</b>	<b>46</b>
9.1	什么是归档重做日志?	46
9.2	开启和关闭归档重做日志	46
9.2.1	开启归档重做日志	47
9.2.2	关闭归档重做日志	47
9.3	控制归档	47
9.3.1	设置数据库归档模式	48
9.3.2	更改数据库归档模式	48
9.3.3	执行归档	48
9.3.3.1	自动触发归档	48

9.3.3.2	定时触发归档	49
9.3.3.3	手动执行归档	49
9.4	指定归档目标	49
9.4.1	设置存档目标的初始化参数	49
9.4.2	归档的状态	50
9.4.2.1	正常归档状态	50
9.4.2.2	未归档状态	50
9.5	查看归档重做日志状态	50
<b>第 10 章</b>	<b>管理表空间</b>	<b>51</b>
10.1	表空间的管理指南	51
10.1.1	使用多个表空间	52
10.1.2	将表空间分配给用户	52
10.2	创建表空间	52
10.2.1	关于创建表空间	53
10.2.2	加密表空间	53
10.2.3	临时表空间	53
10.3	修改表空间可用性	54
10.3.1	使表空间脱机	54
10.3.2	使表空间联机	55
10.4	使用只读表空间	55
10.4.1	关于只读表空间	55
10.4.2	使表空间只读	56
10.4.3	使只读表空间可读写	56
10.5	修改和维护表空间	56
10.6	重命名表空间	57
10.7	删除表空间	57
10.8	查看表空间相关信息	57
<b>第 11 章</b>	<b>管理数据文件和临时文件</b>	<b>59</b>
11.1	数据文件的管理指南	59
11.1.1	关于数据文件	60
11.1.2	选择适当的文件位置	60
11.1.3	将数据文件和重做日志文件分开存储	60
<b>第 12 章</b>	<b>模式对象的管理</b>	<b>61</b>
12.1	关于默认模式	62
12.2	分析表和索引以及聚类	62
12.2.1	关于分析表、索引和聚类	62
12.2.2	使用 sys_stat_statement 搜集执行语句的统计信息	63
12.2.3	分析表、索引、聚类和物化视图	63
12.3	清空表	64
12.3.1	使用 DELETE 清空表	64

12.3.2	使用 DROP 和 CREATE 清空表	64
12.3.3	使用 TRUNCATE 清空表	65
12.4	启用和禁用触发器	65
12.4.1	关于启用和禁用触发器	66
12.4.2	启用触发器	66
12.4.3	禁用触发器	67
12.5	管理完整性约束	67
12.5.1	完整性约束状态	68
12.5.1.1	关于完整性约束状态	68
12.5.1.2	关于禁用约束	69
12.5.1.3	关于启用约束	69
12.5.1.4	完整性约束的有效使用: 顺序	70
12.5.2	创建约束	70
12.5.2.1	在创建表时指定约束	70
12.5.2.2	在创建表后指定约束	71
12.5.3	修改或删除现有的约束	72
12.5.3.1	禁用和启用约束	73
12.5.3.2	重命名约束	73
12.5.3.3	删除约束	74
12.5.4	延迟约束检查	74
12.5.4.1	将所有约束设置为延迟	74
12.5.4.2	检查提交 (可选)	75
12.5.5	查看约束信息	75
12.6	重命名模式对象	75
12.7	管理对象名称解析	76
12.8	切换到其他模式	76
12.9	显示模式对象的信息	77
<b>第 13 章</b>	<b>表的管理</b>	<b>78</b>
13.1	关于表	79
13.2	表管理指南	79
13.2.1	创建表之前的设计准则	80
13.2.2	指定表的存储位置	80
13.2.3	在创建表时考虑使用 UNLOGGED	80
13.2.4	评估表格大小和相应的规划	81
13.3	创建表	81
13.3.1	例子: 创建表	81
13.3.2	创建临时表	82
13.3.2.1	临时表概述	83
13.3.2.2	创建临时表的注意事项	83
13.3.2.3	临时表事务控制	84
13.3.2.4	创建全局临时表	84

13.3.2.5 创建局部临时表	85
13.4 加载表	85
13.4.1 sys_bulkload 工具快速加载数据	86
13.4.2 使用 CREATE TABLE AS 语句	88
13.4.3 使用 INSERT 语句	88
13.4.4 使用 MERGE 语句	89
13.4.5 使用 COPY 语句	90
13.5 收集数据库的统计信息	90
13.6 修改表	91
13.6.1 增加列	92
13.6.2 移除列	93
13.6.3 增加约束	93
13.6.4 移除约束	93
13.6.5 修改列的默认值	94
13.6.6 修改列的数据类型	94
13.6.7 重命名列	94
13.6.8 重命名表	95
13.7 删除表	95
13.8 使用 FLASHBACK 删除和恢复表	95
13.8.1 什么是回收站	96
13.8.2 启用和禁用回收站	97
13.8.3 从回收站恢复表	97
13.9 管理分区表	97
13.10 表的数据视图	98
<b>第 14 章 索引的管理</b>	<b>99</b>
14.1 关于索引	99
14.2 索引的管理指南	100
14.2.1 在插入表数据后创建索引	101
14.2.2 适合创建索引的表和列	101
14.2.3 考虑排序索引列来改善性能	102
14.2.4 限制每个表的索引的数量	102
14.2.5 删除不再需要的索引	102
14.2.6 为索引指定表空间	102
14.2.7 评估索引大小和设置存储参数	103
14.2.8 在禁用或删除约束之前考虑成本	103
14.3 创建索引	103
14.3.1 显式创建索引	104
14.3.2 创建唯一索引	104
14.3.3 创建指定顺序的 B 树索引	105
14.3.4 创建 Hash 索引	105
14.3.5 创建基于函数的索引	105

14.3.6 创建多列索引 . . . . .	106
14.3.7 创建本地和全局索引 . . . . .	106
14.4 修改索引 . . . . .	107
14.4.1 关于修改索引 . . . . .	107
14.4.2 修改或重建索引 . . . . .	108
14.5 删除索引 . . . . .	109
14.6 索引视图 . . . . .	110
<b>第 15 章 管理视图、序列和同义词</b> . . . . .	<b>111</b>
15.1 管理视图 . . . . .	111
15.1.1 关于视图 . . . . .	112
15.1.2 创建视图和连接视图 . . . . .	112
15.1.2.1 创建视图 . . . . .	113
15.1.2.2 创建连接视图 . . . . .	114
15.1.2.3 创建 FORCE 视图 . . . . .	115
15.1.3 替换视图 . . . . .	115
15.1.4 在查询中使用视图 . . . . .	116
15.1.5 修改视图 . . . . .	117
15.1.6 删除视图 . . . . .	117
15.2 管理序列 . . . . .	118
15.2.1 关于序列 . . . . .	118
15.2.2 创建序列 . . . . .	119
15.2.3 修改序列 . . . . .	119
15.2.4 使用序列 . . . . .	120
15.2.4.1 使用 NEXTVAL 生成序列号 . . . . .	120
15.2.4.2 使用 CURRVAL 序列号 . . . . .	121
15.2.5 缓存序列号 . . . . .	121
15.2.5.1 关于缓存序列号 . . . . .	122
15.2.5.2 序列缓存中的条目数 . . . . .	122
15.2.5.3 缓存中的序列号数量 . . . . .	122
15.2.6 删除序列 . . . . .	122
15.3 管理同义词 . . . . .	123
15.3.1 关于同义词 . . . . .	123
15.3.2 创建同义词 . . . . .	124
15.3.3 在 DML 语句中使用同义词 . . . . .	124
15.3.4 删除同义词 . . . . .	124
15.4 视图、同义词和序列的数据字典视图 . . . . .	125
15.4.1 视图信息的查询 . . . . .	126
15.4.2 序列信息的查询 . . . . .	126
15.4.3 同义词信息的查询 . . . . .	126
<b>第 16 章 管理数据库自动维护任务</b> . . . . .	<b>128</b>
16.1 关于自动维护任务 . . . . .	128



16.2 使用自动作业功能	129
16.3 启用和禁用自动维护任务	129
16.4 修改自动作业间隔时间	130
<b>第 17 章 数据库调度概念</b>	<b>131</b>
17.1 调度器概述	131
17.2 作业和支持的调度对象	132
17.2.1 关于作业和支持调度对象	132
17.2.2 程序	132
17.2.3 调度计划	133
17.2.4 作业	133
17.2.4.1 关于作业	133
17.2.4.2 指定作业操作	134
17.2.4.3 指定作业调度	134
17.3 关于作业的更多描述	134
17.3.1 作业类别	135
17.3.1.1 数据库作业	135
17.3.1.1.1 关于数据库作业	135
17.3.1.1.2 本地数据库的作业	135
17.3.1.1.3 远程数据库作业	136
17.3.1.2 外部作业	136
17.3.1.2.1 关于外部作业	136
17.3.1.2.2 关于本地外部作业	136
17.3.2 作业实例	137
17.3.3 作业参数	137
<b>第 18 章 数据库作业调度</b>	<b>138</b>
18.1 作业调度及其子程序	139
18.2 创建程序	140
18.3 删除程序	140
18.4 创建调度计划	140
18.5 删除调度计划	141
18.6 创建作业	141
18.7 删除作业	141
18.8 运行作业	141
18.9 禁用作业	142
18.10 启用作业	142
18.11 任务的日历表示法	142
18.12 计算符合规则的下一个日期	143
<b>第 19 章 管理数据库调度器</b>	<b>144</b>
19.1 配置调度器	144
19.2 监控和管理调度程序	144

19.2.1 查找当前运行的作业信息	145
19.2.2 监控和管理作业日志	145
19.3 导入/导出计划任务	146
<b>第 20 章 全文搜索</b>	<b>147</b>
20.1 介绍	147
20.1.1 什么是一个文档?	148
20.1.2 基本文本匹配	149
20.1.3 配置	151
20.2 表和索引	152
20.2.1 搜索一个表	152
20.2.2 创建索引	153
20.3 控制文本搜索	154
20.3.1 解析文档	154
20.3.2 解析查询	155
20.3.3 排名搜索结果	158
20.3.4 加亮结果	160
20.4 额外特性	161
20.4.1 操纵文档	162
20.4.2 操纵查询	162
20.4.2.1 查询重写	163
20.4.3 用于自动更新的触发器	165
20.4.4 收集文档统计数据	166
20.5 解析器	167
20.6 词典	169
20.6.1 停用词	170
20.6.2 简单词典	171
20.6.3 同义词词典	172
20.6.4 分类词典	174
20.6.4.1 分类词典配置	175
20.6.4.2 分类词典例子	175
20.6.5 Ispell 词典	177
20.6.6 Snowball 词典	179
20.7 配置例子	179
20.8 中文分词	181
20.8.1 zhparser 中文分词插件支持 utf8 和 gbk 字符集	181
20.8.2 sys_jieba 中文分词插件支持 utf8 字符集	182
20.9 流版式文件内容抽取	182
20.9.1 使用 ftutilx 插件抽取文件文本内容	182
20.9.2 使用 ftutilx 全文检索的联合使用方案	183
20.9.3 使用 ftutilx 插件的注意事项	184
20.10 测试和调试文本搜索	184

20.10.1	配置测试	184
20.10.2	解析器测试	187
20.10.3	词典测试	188
20.11	GIN 和 GiST 索引类型	189
20.12	RUM 索引类型	190
20.12.1	常用操作符和函数	190
20.12.2	操作符类	190
20.12.2.1	rum_tsvector_ops	190
20.12.2.2	rum_tsvector_hash_ops	192
20.12.2.3	rum_TYPE_ops	192
20.12.2.4	rum_tsvector_addon_ops	192
20.12.2.5	rum_tsvector_hash_addon_ops	193
20.12.2.6	rum_tsquery_ops	193
20.12.2.7	rum_anyarray_ops	193
20.12.2.8	rum_anyarray_addon_ops	194
20.13	ksql 支持	194
20.14	限制	195
<b>第 21 章</b>	<b>规则系统</b>	<b>196</b>
21.1	查询树	196
21.2	视图和规则系统	198
21.2.1	SELECT 规则如何工作	198
21.2.2	非 SELECT 语句中的视图规则	204
21.2.3	KingbaseES 中视图的能力	205
21.2.4	更新一个视图	205
21.3	物化视图	206
21.4	INSERT、UPDATE 和 DELETE 上的规则	209
21.4.1	更新规则如何工作	209
21.4.1.1	第一个规则循序渐进	210
21.4.2	与视图合作	214
21.5	规则和权限	221
21.6	规则和命令状态	223
21.7	规则 vs 触发器	223
<b>第 22 章</b>	<b>管理表/索引膨胀</b>	<b>226</b>
22.1	识别表/索引膨胀	226
22.1.1	查询表的更新量	226
22.1.2	查询表/索引的膨胀率	226
22.2	处理表/索引膨胀	227
22.2.1	清理	227
22.2.1.1	Autovacuum	227
22.2.1.1.1	参数配置需要考虑的因素	227
22.2.1.1.2	推荐配置	228

---

22.2.1.1.3 监控自动清理 . . . . .	230
22.2.1.2 VACUUM 命令 . . . . .	230
22.2.2 重建 . . . . .	230
22.2.2.1 sys_squeeze . . . . .	230
22.2.2.1.1 工作方式 . . . . .	230
22.2.2.1.2 使用方式 . . . . .	231
22.2.2.1.3 可能的开销 . . . . .	231
22.2.2.1.4 失败处理 . . . . .	232
22.2.2.2 REINDEX . . . . .	232
22.2.2.2.1 工作方式 . . . . .	232
22.2.2.2.2 使用方式 . . . . .	233
22.2.2.2.3 可能的开销 . . . . .	233
22.2.2.2.4 失败处理 . . . . .	233
22.2.2.3 VACUUM FULL . . . . .	233
22.2.2.3.1 工作方式 . . . . .	233
22.2.2.3.2 使用方式 . . . . .	233
22.2.2.3.3 可能的开销 . . . . .	233
22.2.2.3.4 失败处理 . . . . .	233
22.2.3 查看清理/重建进度 . . . . .	234
版权声明 . . . . .	235
服务周期承诺 . . . . .	236

# 第 1 章 前言

---

本文档描述了如何去创建，配置以及管理 KingbaseES 数据库。

前言部分包含以下主题：

- 适用读者
- 相关文档
- 术语
- 手册约定

## 1.1 适用读者

本文档面向所有使用 KingbaseES 的用户，主要是数据库管理员和应用程序开发人员。

本文档适用于执行以下任务：

- 创建和配置 KingbaseES 数据库
- 监控和优化 KingbaseES 数据库
- KingbaseES 数据库的日常维护操作
- 创建和维护模式对象，例如表、索引、视图和序列等
- 计划系统和用户作业
- 诊断、修复和问题报告

在开始阅读本文档之前，您需要了解关系数据库的概念，同时您也应该熟悉运行 KingbaseES 数据库的操作系统。

## 1.2 相关文档

有关 KingbaseES 数据库更多信息，请参阅以下资源：

- 《KingbaseES 数据库概念》
- 《SQL 语言》
- 《PLSQL 过程语言》
- 《KingbaseES 插件参考手册》
- 《数据库参考手册》
- 《KingbaseES ksql 工具用户指南及参考》
- 《KingbaseES 客户端应用参考手册》
- 《KingbaseES 服务器应用参考手册》
- 《KingbaseES 运维工具参考手册》
- 《KingbaseES 数据库安全指南》
- 《KingbaseES 数据库性能调优指南》
- 《KingbaseES 数据库 SQL 调优指南》

## 1.3 术语

本处罗列一些文档章节涉及的基本数据库术语：

术语	定义
数据库	长期存储在计算机内，有组织、可共享、统一管理的大量数据集合。
表	数据库中数据存储的基本单位，数据存储表的行和列中。在关系型数据库中，和“表格”的概念很类似。
列	一系列数据就是具有相同类型的数据，例如学生的年龄（整数）。
行	一行数据是一组相关的数据，例如一名学生的注册信息，整行数据的各列数据都是该学生的基本信息。
键	主键的值是唯一的，一个表中只能包含一个主键。
外键	用于关联两个表。
索引	对数据库表中的一列或者多列进行特定顺序组织的结构，可以用来快速访问数据库表的特定信息。
视图	建立在表上的一种虚表，从本质上说，视图是对存储的查询。
序列	是一种数据库对象，用来自动生成一组具有一定规律（增加或者减小）变化的连续不同序列号。

见续表

表 1.3.1 – 续表

术语	定义
同义词	其本身并不包含原对象中的数据或者代码，它仅仅充当一个指针，是数据库对象的别名。
表空间	系数据库的逻辑划分，数据库所有的对象都存在指定的表空间中，但主要存放表。
完整性约束	对数据库的数据进行指定条件限制，确保数据库中数据正确且有效，比如值不能为空。
事务	访问并可能操作各种数据项的一个数据库操作序列，要么全部执行，要么全部不执行，是一个不可分割的单位。
模式	一个模式 (schema) 从逻辑上划分了数据库对象的归属，一个模式包含视图、索引、序列、触发器、函数和同义词等等。

数据库相关的术语众多，请参阅文档章节阅读即可。

## 1.4 手册约定

本文档中可能出现“注意、提示、警告、另请参阅”等标志，它们所代表的含义如下：

---

**注意：** 用于突出重要/关键信息、最佳实践等。

---



---

**提示：** 用于突出小窍门、捷径等。

---

**警告：** 用于传递设备或环境安全警示信息，若不可避免，可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。

**另请参阅：**

用于突出参考、参阅等。

以下程序代码书写约定适用于本文档：

符号	说明
[]	表示包含一个或多个可选项。不需要输入中括号本身。

见续表

表 1.4.1 – 续表

符号	说明
{}	表示包含两个以上（含两个）的候选，必须在其中选取一个。不需要输入花括号本身。
	分割中括号或者花括号中的两个或两个以上选项。不需要输入“ ”本身。
...	表示其之前的元素可以被重复。
斜体	表示占位符或者需要提供特定值的变量。
大写	表示系统提供的元素，以便与用户定义的元素相互区分。除出现在方括号中的元素外，应当按照顺序逐字输入。当然，部分元素在系统中是大小写不敏感的，因此用户可以根据系统说明以小写形式输入。
小写	表示由用户提供的元素。



---

## 第 2 章 数据库管理入门

---

本章节包含以下内容：

- KingbaseES 数据库的用户类型
- 数据库管理员的任务
- SQL 语句
- 确定数据库的软件版本
- 数据库管理员的安全和权限
- 数据库管理员认证
- 数据操作工具

### 2.1 KingbaseES 数据库的用户类型

数据库的用户类型和各自的职责取决于数据库支持的业务。小型业务可以只有一位数据库管理员来支持应用开发人员 and 数据库用户；大型的业务对数据库的使用则可能需要多位数据库管理员分别负责不同的数据库管理职责。

#### 2.1.1 数据库管理员

数据库至少需要一位数据库管理员，根据数据库支持的业务规模不同，管理员角色可能由单人或是一个团队承担。数据库管理员可能承担以下职责：

- 安装、升级数据库和相关软件
- 规划、分配存储资源
- 为应用开发人员设计的应用创建存储结构（例如表空间）
- 为应用开发人员设计的应用创建对象（例如表、视图、索引）
- 根据应用开发人员的需要修改存储结构和对象
- 创建用户、维护数据库安全

- 控制、监控用户对数据库的访问
- 监控、优化数据库性能
- 规划、实施备份策略、维护归档
- 处理故障、恢复服务
- 联系 KingbaseES 的技术服务渠道获取技术支持

## 2.1.2 安全管理员

有些系统会由一位或多位独立的安全管理员来创建用户，控制、监控用户对数据库的访问，维护数据库安全。有独立的安全管理员的情况下，数据库管理员就不再负责以上这些职责。

参考 预定义管理用户来分配安全相关的职责。

## 2.1.3 应用开发人员

应用开发人员设计并实现使用数据库的应用系统，他们的职责包括：

- 设计、开发应用系统
- 为应用系统设计数据库对象
- 估算应用对存储的容量需求
- 对数据库对象做对应应用的修改
- 向数据库管理员传递以上需求
- 调试、优化应用
- 明确应用对安全的需求

应用开发人员可以联合数据库管理员共同完成上面的一些工作。

## 2.1.4 应用管理员

数据库支撑的应用可能有一个或多个，每个应用都可以由专门的应用管理员负责。

## 2.1.5 数据库用户

数据库用户通过应用或数据库工具和数据库交互，典型用户的职责包括：

- 在权限允许的情况下登入数据库、修改或删除数据
- 查询数据生成报表

## 2.2 数据库管理员的任务

设计、实现和维护 KingbaseES 数据库需要完成以下一些任务：

### 2.2.1 评估数据库使用的硬件资源

评估数据库所需的硬件资源，应该包括：

- 所需要的磁盘容量和 I/O 能力
- 在需要的情况下，磁带的容量
- 需要的内存容量

### 2.2.2 安装数据库软件

数据库管理员需要在一台或多台服务器上安装数据库和相关的工具软件。

### 2.2.3 数据库规划

数据库管理员需要规划数据库的逻辑存储结构，关系对象存储策略，备份策略。

规划逻辑存储结构时考虑对数据库性能和管理任务的影响非常重要。例如，创建表空间前考虑表空间可能存储的物理文件数量、大小；存储数据的访问模式，表空间所在的物理设备的性能。规划逻辑存储结构时应考虑的影响包括：

- 数据库运行的服务器的性能
- 数据访问时的性能
- 数据库备份恢复操作的效率

关系对象存储策略可以直接影响这些对象访问、存储和管理的效率。规划时注意要考虑数据的增长。规划阶段也应该包括备份策略规划，这个阶段可以调整逻辑存储结构和关系对象存储策略来提升备份效率。

《KingbaseES 数据库概念》中的 存储结构和 数据库对象管理章节中包括了对存储结构和关系对象的进一步描述。

《KingbaseES 数据库开发指南》中的 业务系统开发建议章节包括了一些常见的规划建议。

### 2.2.4 创建数据库

创建数据库并启动服务。

## 2.2.5 备份数据库

实施规划的备份策略，包括配置 WAL、归档，配置备份工具和定时任务，完成首次全量备份。

## 2.2.6 创建用户

创建用户并分配合适的权限，可以参考[用户管理和数据库安全](#)。

## 2.2.7 实施数据库设计

根据规划完成表空间和对象的创建。

## 2.2.8 备份功能完整的数据库

数据库设计实施后，数据库就具备了完整功能，再次备份数据库。除了按备份策略做周期性的备份，最好在变更实施后立刻做一次备份。

## 2.2.9 性能调优

数据库管理员需要持续的优化数据库性能，KingbaseES 数据库提供工具和方法诊断性能问题和优化调整。具体参见《KingbaseES 数据库性能调优指南》。

# 2.3 SQL 语句

与 KingbaseES 数据库通信的主要方式是提交 SQL 语句。

- 向数据库提交命令和 SQL

有几种方法可以将 SQL 语句和命令提交到 KingbaseES 数据库。

- 关于 Ksql

Ksql 是 KingbaseES 数据库的主要命令行界面。您可以使用 Ksql 设置数据库初始化参数、创建和管理用户、创建和更改数据库对象 (例如表和索引)、插入和更新数据、运行 SQL 查询等等。

- 使用 Ksql 连接到数据库

使用 Ksql 连接到 KingbaseES 数据库实例。

## 2.3.1 向数据库提交命令和 SQL

有几种方法可以将 SQL 语句和命令提交到 KingbaseES 数据库。

- 直接使用 Ksql 的命令行界面
- 间接使用图形用户界面

例如通过 KingbaseES 数据库对象管理工具 KStudio 的图形界面来管理数据库，该工具在后台提交 SQL 语句和命令。

## 2.3.2 关于 Ksql

Ksql 是 KingbaseES 数据库的主要命令行界面。您可以使用 Ksql 设置数据库初始化参数、创建和管理用户、创建和更改数据库对象 (例如表和索引)、插入和更新数据、运行 SQL 查询等等。

在提交 SQL 语句和命令之前，您必须连接到数据库。使用 Ksql，您可以本地或者远程连接。本地连接意味着连接到运行 Ksql 的同一台计算机上运行的 KingbaseES 数据库。远程连接意味着通过网络连接到远程计算机上运行的 KingbaseES 数据库。这样的数据库成为远程的数据库。

## 2.3.3 连接数据库

使用 Ksql 连接到 KingbaseES 数据库实例。

- 关于使用 Ksql 连接到数据库

您必须连接到 KingbaseES 数据库才能查询或修改该数据库中的数据。您可以连接到默认数据库或通过网络连接到其他数据库。

- Step 1: 打开命令窗口

在您的平台上执行必要的操作以打开一个窗口，您可以在其中输入操作系统命令。

- Step 2: 设置操作系统环境变量

根据您的平台，您可能必须在启动 Ksql 之前设置环境变量，或者至少验证它们是否正确。

- Step 3: 启动 Ksql

启动 Ksql。

- Step 4: Ksql 连接到 KingbaseES 数据库

使用 Ksql 连接到 KingbaseES 数据库实例，可以用不同用户的身份重新连接。

## 2.3.4 关于使用 Ksql 连接到数据库

KingbaseES 数据库包含以下组件：KingbaseES 数据库实例，它是进程和内存的集合，以及一组包含用户数据和系统数据的磁盘文件。

由于主机上可能有多个 KingbaseES 实例，每个实例都有自己的一组数据文件，因此您必须确定连接的实例。对于本地连接，您可以通过设置操作系统环境变量来识别实例。对于远程连接，您可以通过指定网路和数据库服务名称来识别实例。对于本地和远程连接，您必须设置环境变量以帮助操作系统找到 Ksql 可执行文件并为可执行文件提供支持文件和脚本路径。

### 2.3.5 Step 1: 打开命令窗口

在您的平台上执行必要的操作以打开一个窗口，您可以在其中输入操作系统命令。

### 2.3.6 Step 2: 设置操作系统环境变量

根据您的平台，您可能必须在启动 Ksql 之前设置环境变量，或者至少验证它们是否设置正确。

例如，在大多数平台上，您需要配置 PATH 环境变量以包含 KingbaseES 的 bin 目录。

- 在 Linux 上，通过输入操作系统命令来设置环境变量。

```
-- Setting Environment Variables in Linux (C Shell)
setenv LD_LIBRARY_PATH=/home/test/kb_install/lib/kingbase/

-- Setting Environment Variables in Linux (Bash Shell)
export LD_LIBRARY_PATH=/home/test/kb_install/lib/kingbase/
```

### 2.3.7 Step 3: 启动 Ksql

要开始使用 Ksql，您必须首先了解如何启动和停止 Ksql。

- 请确保在您的计算机上已经安装了 Ksql
- 登录到操作系统 (如果需要)
- 输入连接到数据库的命令，然后按 Enter 键

```
ksql -h hostname -U username -d dbname -p port
```

### 2.3.8 Step 4: Ksql 连接到 KingbaseES 数据库

当您启动 Ksql 时，需要一个用户名和密码才能登录到 KingbaseES 数据库模式。您的用户名和密码是 KingbaseES 数据库授权的用户。

您可以使用连接命令对不同的用户进行连接，用户名和密码必须对该数据库有效。例如，连接到 user1 用户：

```
-- Connect to a Local Database User
ksql -U username -d dbname -p port

-- Connect to a Database with the Host IP Address
ksql -h hostname -U username -d dbname -p port
```

### 2.3.8.1 Ksql 连接数据库命令语法

连接命令格式如下:

```
ksql [option...] [dbname [username]]
```

```
ksql -U username -d dbname -p port
```

更多相关 Ksql 使用的信息请参考《KingbaseES ksql 工具用户指南及参考》。

## 2.4 确定数据库的软件版本

### 2.4.1 KingbaseES 数据库的版本信息

KingbaseES 数据库的版本信息含义参见 KingbaseES 版本规则。

### 2.4.2 获取当前数据库的版本信息

可以通过以下 SQL 获取数据库版本信息:

```
SELECT version();
```

## 2.5 数据库管理员的安全和权限

### 2.5.1 管理员用户

KingbaseES 采用了三权分立的安全管理体制，系统初始化时会创建三个管理员用户：数据库管理员、安全管理员、审计管理员。

- 数据库管理员

用户名和密码由初始化时设定，通常用户名指定为 system。主要负责执行数据库日常管理的各种操作和自主存取控制。

- 安全管理员

用户名默认为 sso，主要负责强制访问规则的制定和管理，监督审计管理员和普通用户的操作，不能创建和操作普通对象。

- 审计管理员

用户名默认为 sao，主要负责数据库的审计，监督系统管理员和安全管理员的操作，不能创建和操作普通对象。

三权分立的安全管理体制是为了解决数据库超级用户权力过度集中的问题。此外，还可以创建普通用户来访问数据库，运行数据库应用。

## 2.5.2 管理员权限

管理员执行基本数据库操作所需要的权限是通过特殊的系统权限授予的，在创建用户时可以指定授权。KingbaseES 还提供一些特殊的管理特权，例如，ANY 权限和 SYSBACKUP 特权。

管理权限的内容如下表所示：

表 2.5.1: 管理权限

权限	描述
SUPERUSER	超级用户，越过数据库内的所有访问限制，是最强大的管理权限
CREATEDB	允许创建数据库
CREATEROLE	允许创建用户和角色
LOGIN	允许登录数据库
REPLICATION	是否为复制角色
ANY 权限	允许用户操作所有的某种类型的数据库对象的某种操作，不包括系统对象
SYSBACKUP	允许执行物理备份操作

更多权限管理内容，可参考《KingbaseES 数据库安全指南》。

## 2.6 数据库管理员认证

在 KingbaseES 数据库中，可以通过使用存储在数据库字典中的账户和口令对管理员进行身份验证，即口令验证。同时，还支持结合第三方服务 Kerberos、RADIUS、LDAP、SSL 等认证协议和 CA 等技术对数据库管理员进行强化身份验证。更多身份验证内容，可参考《KingbaseES 数据库安全指南》。



## 2.7 数据操作工具

KingbaseES 提供多种工具用于维护数据。

### 2.7.1 数据加载工具

使用方式参见《KingbaseES 客户端应用参考手册》中的 `bulkload`。

### 2.7.2 数据导出导入工具

使用方式参见《KingbaseES 客户端应用参考手册》中的 `sys_dump`、`sys_restore` 和 `exp/imp` 工具介绍。

---

# 第 3 章 管理进程

---

本章节包含以下内容：

- 关于独立服务进程和共享服务进程
- 管理 KingbaseES 的独立服务进程
- 终止会话
- 进程和会话视图

## 3.1 关于独立服务进程和共享服务进程

KingbaseES 数据库使用客户端/服务器的模型，服务端创建服务进程处理客户端连接。独立服务进程指一个服务进程只处理一个客户端的请求；共享服务进程指一个服务进程处理多个客户端的请求。

## 3.2 管理 KingbaseES 的独立服务进程

KingbaseES 数据库使用独立服务进程模式。控制服务进程数量、资源使用的配置参数参见：

- 连接和认证
- 资源消耗

## 3.3 关于 KingbaseES 的后台进程

为了最大化客户端请求的吞吐量和响应时间，KingbaseES 使用后台进程完成一些工作以避免服务进程因为这些工作被拖慢。

基础的后台进程和在特定时机或使用一些特性时会有的额外后台进程参见：后台进程

## 3.4 终止会话

管理数据库有时需要终止会话，例如终止占用封锁或系统资源过多的会话。

### 3.4.1 指定需要终止的会话

终止会话需要指定会话的 pid。可以通过视图 `sys_stat_activity` 查找特定会话的 pid。

示例，查找用户为 `kingbase` 的会话 pid 和会话状态：

```
SELECT pid, state FROM sys_stat_activity WHERE username = 'kingbase';
 pid | state
-----+-----
 26212 | active
 26292 | active
(4 rows)
```

### 3.4.2 终止会话

终止会话将结束会话并断开客户端到服务端的连接。会话正在执行的事务会由于连接断开回滚。使用 `sys_terminate_backend(integer)` 函数终止会话，参数为会话的 pid。

示例，终止 pid 为 26212 的会话：

```
SELECT sys_terminate_backend(26212);
 sys_terminate_backend
-----
 t
(1 row)
```

### 3.4.3 取消会话执行的 SQL 语句

某一会话执行的 SQL 语句占用过多资源影响业务时，可以通过取消会话执行的 SQL 代替终止会话。会话执行 SQL 所在的事务将回滚。使用 `sys_cancel_backend(integer)` 函数取消会话执行的 SQL 语句，参数为会话的 pid。

示例，取消 pid 为 26212 的会话执行的 SQL 语句：

```
SELECT sys_cancel_backend(26212);
 sys_cancel_backend
-----
 t
(1 row)
```

## 3.5 进程和会话视图

可以通过以下视图获取进程和会话信息：

**sys\_catalog.sys\_stat\_activity** 当前系统中进程/会话的信息。

**sys.sys\_session** 当前系统中的会话信息。

**sys.v\$session** 兼容 Oracle 的 v\$session 视图，提供当前系统中会话信息，详见 v\$session 。

可以通过以上视图中 pid/sess\_id/sid 作为唯一标志和其他视图关联获取进程/会话信息。

例如和 sys\_locks 视图关联获取进程/会话的封锁信息。

---

# 第 4 章 管理内存

---

管理内存涉及维护和变更数据库实例中各项内存结构的大小，以适应数据库的相关需要。

- 关于内存管理

需要被管理的内存结构是指系统全局区（简称为 SGA）和实例进程全局区（简称为 PGA）。KingbaseES 数据库支持以配置参数的形式设置内存区域的大小。

- 内存结构概述

了解 KingbaseES 数据库基础的内存结构。

- 手动配置内存

当你希望对单个内存组件的大小进行直接的控制时，可以手动配置和调整内存组件。

## 4.1 关于内存管理

需要被管理的内存结构是指系统全局区（简称为 SGA）和实例进程全局区（简称为实例 PGA）。目前 KingbaseES 不支持自动内存管理，仅支持手动内存管理，数据库提供一系列配置参数，以支持对每个内存区域的大小进行直接的控制。

父主题：管理内存

## 4.2 内存结构概述

了解 KingbaseES 数据库基础的内存结构。KingbaseES 数据库中，相关的基本内存结构包括：

- 系统全局区（简称为 SGA）SGA 是一组共享内存结构，其中包含一个 KingbaseES 数据库实例的数据和控制信息。SGA 由所有服务器和后台进程共享。存储在 SGA 中的数据包括数据块缓冲区和重做日志缓冲区等。
- 进程全局区（简称为 PGA）PGA 是包含单个服务进程的数据和控制信息的内存区域。它是在启动服务进程时创建的非共享内存。只有该服务进程可以对本 PGA 进行访问。每个服务进程都有一个 PGA，每个后台进程也

会有自己的 PGA。一个实例中所有 PGA 的总和称为实例 PGA。存储在 PGA 中的数据包括排序和哈希操作可使用的工作内存等。

父主题：管理内存

## 4.3 手动配置内存

当你希望对单个内存组件的大小进行直接的控制时，可以手动配置和调整内存组件。你可以通过配置参数指定对应区域的大小。

- 系统全局区配置

手动配置和调整系统全局区中的内存结构大小。

- 进程全局区配置

手动配置和调整进程全局区中的内存结构大小。

父主题：管理内存

### 4.3.1 系统全局区配置

手动配置和调整系统全局区中的内存结构大小。

- 数据页面缓存

在共享内存中缓存的数据页面的大小，由 `shared_buffers` 参数控制其大小。该参数会影响数据库性能。配置的缓存越大，则在内存中保存的数据页面就越多，这通常会减少磁盘读取和写入的次数，并因此提高性能。然而过大的数据页面缓存也可能会占用过多的系统内存并导致内存分页或交换。

出于 IO 资源优化的角度，建议将 `shared_buffers` 配置为操作系统总内存的一半，根据特定需求浮动，但一般不建议设置超过操作系统总内存的 80%。

采用以下方式修改数据页面缓存大小：

```
ALTER SYSTEM SET shared_buffers = '1GB';
```

重启数据库后生效。

采用以下方式查看数据页面缓存大小：

```
SHOW shared_buffers;
```

- 日志页面缓存

数据库操作产生的重做日志内容会存放在日志页面缓存中，并在需要时由日志写进程和服务进程刷写到磁盘。参数 `wal_buffers` 可以设置日志页面缓存大小，。如果每个事务对数据页面修改的数据量较大，占满了日志页面缓存而

不得不提前刷写日志，将导致一定的 IO 资源浪费，此时可以适当增大 `wal_buffers` 的值，以降低日志刷写磁盘的频率。由于日志总是需要在事务提交时和检查点刷盘，因此过大的 `wal_buffers` 并没有实际意义。

采用以下方式修改日志页面缓存大小：

```
ALTER SYSTEM SET wal_buffers = '4MB';
```

重启数据库后生效。

采用以下方式查看日志页面缓存大小：

```
SHOW wal_buffers;
```

- 锁缓存

数据库事务所需的并发控制机制的组成部分之一是锁，而锁的控制信息则存在于系统全局区 SGA 中。锁缓存的大小由数据库内部决定，不可配置，但是参数 `max_locks_per_transaction` 可以影响到锁缓存的大小。参数 `max_locks_per_transaction` 代表了每个事务最大可持有的锁数量，缺省为 64。参数 `max_locks_per_transaction` 越大，锁缓存所占有的空间就越大。

通常锁缓存占用内存在系统中占比不大，不会成为系统的瓶颈，不需要手动配置。

父主题：[手动配置内存](#)

## 4.3.2 进程全局区配置

手动配置和调整进程全局区中的内存结构大小。

- 临时页面缓存

在进程私有内存中用于缓存临时表的数据页面的大小，由 `temp_buffers` 参数控制其大小。

采用以下方式修改临时页面缓存大小：

```
ALTER SYSTEM SET temp_buffers = '8MB';
```

重启数据库后生效。

采用以下方式查看临时页面缓存大小：

```
SHOW temp_buffers;
```

- 工作内存

服务器对元组进行排序或者连接运算时，需要在 PGA 中缓存临时结果集数据，这部分内存称为工作内存。如果工作内存空间不足，数据库会转由临时文件存储这部分数据。参数 `work_mem` 设置每个服务进程的工作内存大小。工作内存只在正在进行排序或者连接运算的服务进程中被分配，并会随着运算结束而释放。

采用以下方式修改工作内存大小：

```
ALTER SYSTEM SET work_mem = '4MB';
```

重启数据库后生效。

采用以下方式查看工作内存大小：

```
SHOW work_mem;
```

- 维护工作内存

在维护性操作 (比如 VACUUM, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY 等) 中使用的最大的内存空间。参数 `maintenance_work_mem` 设置维护工作内存的大小。默认是 64MB。较大的维护工作内存可以有效改进清理和恢复数据的速度。

采用以下方式修改维护工作内存大小：

```
ALTER SYSTEM SET maintenance_work_mem = '64MB';
```

重启数据库后生效。

采用以下方式查看维护工作内存大小：

```
SHOW maintenance_work_mem;
```

- 临时分配的内存

数据库在执行 SQL 过程中，用于临时存放控制信息及数据所使用的内存。此部分内存动态分配和释放，并受到内存上下文管理。此部分内存大小不可配置。

父主题：[手动配置内存](#)



# 第 5 章 用户管理和数据库安全

为每个数据库建立安全策略

- 数据库在使用之前设置安全策略的重要性

制定安全策略非常重要，安全策略的建立是保护数据库免于遭受意外、恶意数据破坏或者数据库基础结构损坏的有效方法

- 管理用户

若要连接数据库，就必须在数据库中注册有效的用户名。

- 用户的权限和角色

KingbaseES 拥有多种权限的角色，是为了控制用户对数据的访问以及区分可执行的 SQL 语句的类型。

- 审计数据库活动

您可以监控和记录包括管理员在内的选定用户的数据库操作，您也可以监控系统级的操作，以及在单个数据库对象上的操作，我们统称该类操作为数据库审计。

- 预定义的账户

KingbaseES 数据库包含多种预定义的用户帐户

## 5.1 数据库在使用之前设置安全策略的重要性

为每个数据库制定安全策略是重要且有必要的。安全策略是为了保护数据库免于遭受意外、恶意数据破坏或者数据库基础结构损坏的有效方法。每个数据库都可以有一个安全管理员，负责实施和维护数据库安全策略。如果数据库系统规模很小，则数据库管理员可以承担安全管理员的职责。相反，如果数据库系统规模很大，则需要指定一个人或者一组人担任安全管理员的职责。

另请参阅:

《KingbaseES 数据库安全指南》，获取有关为数据库建立安全策略的信息

父主题：用户管理和数据库安全

## 5.2 管理用户

若要连接数据库，就必须在数据库中注册有效的用户名。当您创建数据库用户（帐户）时，指定用户以下属性：

- 用户名
- 身份验证方法
- 缺省表空间

另请参阅：

《KingbaseES 数据库安全指南》，了解如何创建和管理用户。

父主题：用户管理和数据库安全

## 5.3 用户的权限和角色

KingbaseES 拥有多种权限的角色，是为了控制用户对数据的访问以及区分可执行的 SQL 语句的类型。

下表描述了三种类型的特权和角色：

类型	描述
系统权限	通常仅由管理员授予的系统定义特权。这些权限允许用户执行特定的数据库操作。
对象权限	一种系统定义的权限，用于控制对特定对象的访问。
角色	权限和角色的集合，存在部分系统定义的角色，但大多数由管理员创建。角色可以将权限和其他角色组合在一起，方便用户拥有多种多样权限的角色

权限和角色可由被授予权限的用户授予其他用户。角色和特权的授予是从管理员级别开始的。在创建数据库后，用户 system 可以向其他用户授予权限和角色，还可以向用户授予向其他用户授权的权限。

另请参阅：

《KingbaseES 数据库安全指南》，了获取更多关于如何管理用户权限和角色的信息

父主题：用户管理和数据库安全

## 5.4 审计数据库活动

您可以监控和记录包括管理员在内的选定用户的数据库操作，您也可以监控系统级的操作，以及在单个数据库对象上的操作，我们统称该类操作为数据库审计。

KingbaseES 数据库通过插件 sysaudit 实现审计功能，您可以通过 SQL 语句创建自定义的审计策略。

另请参阅：

《KingbaseES 数据库安全指南》，了解有关数据库审计的更多信息

父主题：[用户管理和数据库安全](#)

## 5.5 预定义的账户

KingbaseES 数据库安装部署后，会默认创建多种预定义的用户，有以下三种类型：

- 数据库管理员（system）
- 安全管理员（sso）
- 审计管理员（sao）

三个用户将管理权限三权分立，在系统初始化时指定用户名和密码。

在 KingbaseES 数据库中，可通过查询系统表或系统视图查看用户账户信息。用户账户对应的视图 sys\_shadow 和 sys\_user，如下所示：

```
TEST=# select * from sys_user;
username|usesysid|usecreatedb|usesuper|userepl|usebypassrls|passwd |valuntil|
useconfig
-----+-----+-----+-----+-----+-----+-----+-----+
system |    10 | t       | t       | t       | t       | ***** |      |
sso    |     8 | f       | f       | f       | f       | ***** |      |
sao    |     9 | f       | f       | f       | f       | ***** |      |
(3 行记录)
```

另请参阅：

《KingbaseES 数据库安全指南》将提供有关 KingbaseES 数据库所有预定义帐户的信息

父主题：[用户管理和数据库安全](#)

---

# 第 6 章 监控数据库

---

定期监视数据库的操作很重要。这样做不仅可以通知您尚未引起注意的错误，还可以让您更好地了解数据库的正常操作。反过来，熟悉正常行为可以帮助您识别何时出现问题。

- **监控错误和警报**

您可以监控数据库错误和警报以预防、检测和解决问题。

- **监控数据库性能**

监控性能包括监控锁和等待事件以及查询一组数据字典视图。

## 6.1 监控错误和警报

可以监控数据库错误和警报以预防、检测和解决问题。对于数据库运行过程中的错误，除了人工去查看数据库运行过程中产生的日志文件外，也可以使用 kbbadger 工具自动分析。

### 6.1.1 使用 kbbadger

Kbbadger 可以分析大型的日志文件。当日志文件足够长时，kbbadger 可以自动检测日志文件的格式（syslog，stderr，csvlog 或 jsonlog）。

由 kbbadger 生成的所有图表都是可缩放的并且可单独下载为 PNG 文件。另外，在生成的报告中 SQL 查询将突出显示。

Kbbadger 生成的报告中包含的关于 SQL 查询的信息有：

- 总体统计
- 占用时间最多的查询
- 最常见的查询
- 最常见的错误
- 查询时间直方图

- 会话时间直方图
- 参与顶级查询的用户
- 涉及顶级查询的应用程序
- 生成最多取消的查询
- 大多数查询已取消
- 最耗时的准备/绑定查询

报告也会提供每小时的统计图表，其内容包括：

- SQL 查询统计信息
- 临时文件统计
- 检查点统计
- 自动 vacuum 和自动分析统计
- 已取消查询
- 错误事件（死机、致命、错误和警告）
- 错误的类分布

在 kbbadger 的使用过程中，可以使用命令行选项来启用并行处理模式从而加快日志解析速度，使用 -j 命令行选项来指定解析使用的 CPU 核心数，使用 -J 命令行选项来并行解析的文件数，两个选项可以同时使用。另外，可以使用 -A 命令行选项来调整直方图的粒度。默认情况下，它们将报告每小时发生的每个主要查询/错误的平均值。

## Kbbadger 使用方法

### 1. 配置 kingbase.conf

在开始之前，必须在 kingbase.conf 中启用并设置一些配置指令。

您必须首先启用 SQL 查询日志记录才能进行解析：

```
log_min_duration_statement = 0
```

在这里，每条语句都会被记录，在繁忙的服务器上，您可能需要增加此值以仅记录持续时间较长的查询。请注意，不要启用 log\_statement，因为 kbBadger 不会解析它的日志格式并且如果您将 log\_statement 设置为'all'，则不会通过 log\_min\_duration\_statement 指令记录任何内容。如果要保证系统性能，也可以使用 log\_duration 来仅提供有关持续时间和查询数量的报告。有关更多信息，请参见下一章。

kbBadger 支持任何设置到 kingbase.conf 文件的 log\_line\_prefix 中的自定义格式，只要它指定时间转义序列（%t，%m 或%n）和与进程相关的转义序列（%p 或%c）。

例如：

```
--使用“stderr”日志格式，log_line_prefix 必须至少为：
log_line_prefix = '%t [%p]: '
```

--日志行前缀可以添加用户, 数据库名称, 应用程序名称和客户端 IP 地址, 如下所示:

```
log_line_prefix = '%t [%p]: user=%u,db=%d,app=%a,client=%h '
```

--或用于 syslog 日志文件格式:

```
log_line_prefix = 'user=%u,db=%d,app=%a,client=%h '
```

--stderr 输出的日志行前缀也可以是:

```
log_line_prefix = '%t [%p]: db=%d,user=%u,app=%a,client=%h '
```

--或用于 syslog 输出:

```
log_line_prefix = 'db=%d,user=%u,app=%a,client=%h '
```

--您也可以在 kingbase.conf 中启用其他参数, 以从日志文件中获取更多信息:

```
log_checkpoints = on
log_connections = on
log_disconnections = on
log_lock_waits = on
log_temp_files = 0
log_autovacuum_min_duration = 0
log_error_verbosity = default
```

**注意:** 数据库日志中的内容必须为英文, 否则 kbbadger 可能无法解析日志中的内容:

```
lc_messages='en_US.UTF-8'
```

## 2. 使用 kbBadger 生成统计信息

```
kbbadger /var/log/kingbase.log
kbbadger /var/log/kingbase.log.2.gz /var/log/kingbase.log.1.gz /var/log/kingbase.log
kbbadger /var/log/kingbase/kingbase-2012-05-*
kbbadger --exclude-query="^(COPY|COMMIT)" /var/log/kingbase.log
kbbadger -b "2012-06-25 10:56:11" -e "2012-06-25 10:59:11" /var/log/kingbase.log
cat /var/log/kingbase.log | kbbadger -
kbbadger --prefix '%t [%p]: user=%u,db=%d,client=%h ' /sys_log/kingbase-2012-08-21*
kbbadger --prefix '%m %u%d %p %r %a : ' /sys_log/kingbase.log
# Log line prefix with syslog log output
kbbadger --prefix 'user=%u,db=%d,client=%h,appname=%a' /sys_log/kingbase-2012-08-21*
# Use my 8 CPUs to parse my 10GB file faster, much faster
kbbadger -j 8 /sys_log/kingbase-10.1-main.log
```

详细请参见 - help

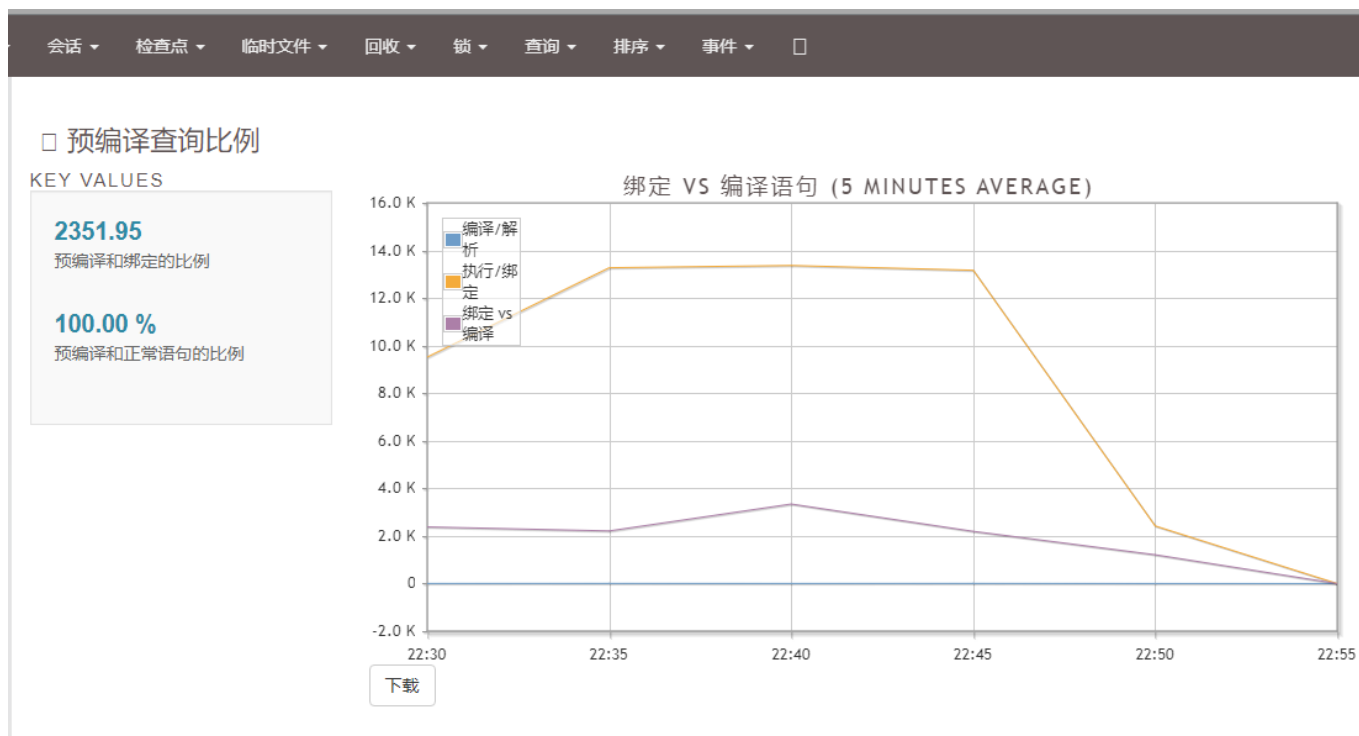
## 3. 举例说明

```
$ ./kbbadger ./ sys_log/kingbase-2019-10-26_1544* -f stderr -J 12 -j 32
[=====] Parsed 283210387 bytes of 283210387 (100.00%),
queries: 327810, events: 51
LOG: Ok, generating html report...
```

生成的统计信息效果图如下：

排名	总持续时间	执行次数	最小持续时间	最大持续时间	平均持续时间	查询
1	39m5s	12,487 <a href="#">细节</a>	100ms	10s10ms	187ms	<code>UPDATE benchmarksql.warehouse SET w_ytd = w_ytd + ' WHERE w_id = ';</code> <a href="#">示例</a> <a href="#">执行用户</a>
2	34m54s	11,195 <a href="#">细节</a>	100ms	2s479ms	187ms	<code>SELECT d_next_o_id, d_tax, ctid, xmin FROM benchmarksql.district WHERE d_id = ' AND d_w_id = ' FOR UPDATE;</code> <a href="#">示例</a> <a href="#">执行用户</a>
3	34m17s	11,415 <a href="#">细节</a>	100ms	2s70ms	180ms	<code>UPDATE benchmarksql.district SET d_ytd = d_ytd + ' WHERE d_w_id = ' AND d_id = ';</code> <a href="#">示例</a> <a href="#">执行用户</a>
4	16m11s	4,098 <a href="#">细节</a>	100ms	1s299ms	237ms	<code>INSERT INTO benchmarksql.order_line (ol_o_id, ol_d_id, ol_w_id, ol_number, ol_i_id, ol_supply_w_id, ol_quantity, ol_amount, ol_dist_info) VALUES ('', '', '', '', '', '', '', '', '');</code> <a href="#">示例</a> <a href="#">执行用户</a>
5	7m36s	3,370 <a href="#">细节</a>	100ms	916ms	135ms	<code>SELECT count (DISTINCT (s_i_id)) AS stock_count FROM benchmarksql.order_line, benchmarksql.stock WHERE ol_w_id = ' AND ol_d_id = ' AND ol_o_id &lt; ' AND ol_o_id &gt;= ' - 0 AND s_w_id = ' AND s_i_id = ol_i_id AND s_quantity &lt; ';</code> <a href="#">示例</a> <a href="#">执行用户</a>

类型	数量	百分比
SELECT	18,396	35.55%
INSERT	5,624	10.87%
UPDATE	26,023	50.29%
DELETE	1,234	2.38%
COPY FROM	0	0.00%
COPY TO	0	0.00%
CTE	0	0.00%
DDL	0	0.00%
TCL	466	0.90%



## 6.2 监控数据库性能

监控性能包括监控锁和等待事件以及查询一组数据字典视图。

- 监控锁

锁是防止访问同一资源的事务之间的破坏性交互的机制。资源可以是用户对象，例如表和行，也可以是用户不可见的系统对象，例如内存中的共享数据结构和数据字典行。

- 监控等待事件

等待事件是由服务器进程递增的统计信息，表明它必须等待事件完成才能继续处理。会话可能出于多种原因等待，包括等待更多输入、等待操作系统完成诸如磁盘写入之类的服务，或者它可能等待锁定或门锁。

- 性能监控视图

您可以查询一组性能视图来监控 KingbaseES 数据库实例。

### 6.2.1 监控锁

锁是防止访问同一资源的事务之间进行破坏性交互的机制。资源可以是用户对象，例如表和行，也可以是用户不可见的系统对象，例如内存中的共享数据结构和数据字典行。

KingbaseES 数据库在执行 SQL 语句时会自动获取并管理必要的锁，因此您无需关心这些细节。但是，数据库还允许您手动锁定数据。



当两个或更多用户正在等待彼此锁定的数据时，可能会发生死锁。死锁会阻止某些事务继续工作。KingbaseES 数据库自动检测死锁情况并通过回滚死锁中涉及的一条语句来解决它们，从而释放一组冲突的行锁。

KingbaseES 数据库旨在避免死锁，它们并不常见。大多数情况下，它们发生在事务显式覆盖数据库的默认锁定时。死锁会影响数据库的性能，因此 KingbaseES 提供了一些脚本和视图，使您能够监控锁。

### 6.2.1.1 使用 sys\_locks

所有活动事务持有的监控锁的基本配置即为系统视图 sys\_locks。这个视图为每个可加锁的对象、已请求的锁模式和相关事务包含一行记录。

```
test=# select * from sys_locks;
-[ RECORD 1 ]-----+-----
locktype           | relation
database           | 16384
relation           | 12168
page               |
tuple              |
virtualxid         |
transactionid      |
classid            |
objid              |
objsubid           |
virtualtransaction | 3/744
pid                | 224515
mode               | AccessShareLock
granted            | t
fastpath           | t
```

### 6.2.1.2 监控死锁

显式锁定的使用可能会增加死锁的可能性，死锁是指两个（或多个）事务相互持有对方想要的锁。

例如，如果事务 1 在表 A 上获得一个排他锁，同时试图获取一个在表 B 上的排他锁，而事务 2 已经持有表 B 的排他锁，同时却正在请求表 A 上的一个排他锁，那么两个事务就都不能进行下去。KingbaseES 能够自动检测到死锁情况并且会通过中断其中一个事务从而允许其它事务完成来解决这个问题（具体哪个事务会被中断是很难预测的，而且也不应该依靠这样的预测）。

要注意死锁也可能作为行级锁的结果而发生（并且因此，它们即使在没有使用显式锁定的情况下也会发生）。考虑如下情况，两个并发事务在修改一个表。第一个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

这样就在指定帐号的行上获得了一个行级锁。然后，第二个事务执行：

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

第一个 UPDATE 语句成功地在指定行上获得了一个行级锁，因此它成功更新了该行。但是第二个 UPDATE 语句发现它试图更新的行已经被锁住了，因此它等待持有该锁的事务结束。事务二现在就在等待事务一结束，然后再继续执行。现在，事务一执行：

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

事务一试图在指定行上获得一个行级锁，但是它得不到：事务二已经持有这样的锁。所以它要等待事务二完成。因此，事务一被事务二阻塞，而事务二也被事务一阻塞：一个死锁。KingbaseES 将检测这样的情况并中断其中一个事务。

## 6.2.2 监控等待事件

可以查看系统视图 `sys_stat_activity` 来进行查看每个连接上的等待事件。

**注意：** 当前执行的查询语句和等待事件状态需要在开启 `track_activities` 参数的情况下才可以查看。开启该参数会带来一定的性能损耗。

`sys_stat_activity` 视图结构如下：

表 6.2.1: 视图结构

字段名	类型	说明
<code>datid</code>	<code>oid</code>	数据库 OID
<code>datname</code>	<code>name</code>	数据库名称
<code>pid</code>	<code>integer</code>	服务于这个连接的进程 ID
<code>usesysid</code>	<code>oid</code>	用户 ID
<code>username</code>	<code>name</code>	用户名
<code>application_name</code>	<code>text</code>	应用名称
<code>client_addr</code>	<code>inet</code>	客户端地址
<code>client_hostname</code>	<code>text</code>	客户端主机名
<code>client_port</code>	<code>integer</code>	客户端端口号

见续表

表 6.2.1 – 续表

字段名	类型	说明
backend_start	timestamp with time zone	该连接的启动时间
xact_start	timestamp with time zone	当前事务开始时间
query_start	timestamp with time zone	当前查询开始时间
state_change	timestamp with time zone	当前状态改变时间
wait_event_type	text	当前等待事件的类型
wait_event	text	当前等待事件
state	text	当前的状态
backend_xid	xid	这个后端的顶层事务标识符
backend_xmin	xid	这个进程的当前事务被启动的时间
query	text	当前查询语句
backend_type	text	后端进程的类型

其中，state 为当前连接的状态，其可能的值主要有：

- active: 后端正在执行一个查询。
- idle: 后端正在等待一个新的客户端命令。
- idle in transaction: 后端在一个事务中，但是当前没有正在执行一个查询。
- idle in transaction (aborted): 这个状态与 idle in transaction 相似，不过在该事务中的一个语句导致了一个错误。
- fastpath function call: 后端正在执行一个 fast-path 函数。
- disabled: 如果在这个后端中 track\_activities 被禁用，则报告这个状态。

---

**注意：** 如果一个查询或者事务执行的时间过长而一直没有结束，很有可能会拖累整个系统的性能表现，需要具体分析问题的原因并采取合适的办法。

---

### 6.2.2.1 等待事件

等待事件是数据库内部记录的一种统计信息，该信息出现表明数据库的服务进程必须在等待事件完成后才能继续处理。等待事件揭示了可能影响性能的各种问题症状，比如缓冲区中争用、锁争用等。

`sys_stat_activity` 里记录的等待事件是瞬时信息，没有对等待事件的时间进行累计，所以量化等待事件上存在一些问题。用户可以考虑多次人工采样来收集等待事件信息，如果在等待的连接较多，则需要等待事件对系统性能造成了较大影响，需要做具体分析。

**注意：**当前执行的查询语句和等待事件状态需要在开启 `track_activities` 参数的情况下才可以查看。开启该参数会带来一定的性能损耗。

`sys_stat_activity` 视图中，`wait_event_type` 和 `wait_event` 字段记录了等待时间相关的内容。当一个会话处于等待状态时，`wait_event` 与 `wait_event_type` 两列非空，表示会话正在等待的事件和等待事件类型。

KingbaseES 的等待事件分类包括：

- LWLock: 轻量级锁
- Lock: 重量级锁
- BufferPin: 数据缓冲区等待
- Activity: 后台辅助进程活动等待
- Client: 客户端等待
- Extension: 扩展插件等待
- IPC: 进程间通讯等待
- TimeOut: 超时等待
- IO: IO 等待

每一个等待事件的名称和产生的原因，可以参考《KingbaseES 数据库参考手册》中的 等待事件。

查询等待事件示例：

```
test=# select * from sys_stat_activity;
-[ RECORD 1 ]-----+-----
datid          |
datname        |
pid            | 236705
usesysid       |
username       |
application_name |
client_addr    |
client_hostname |
client_port    |
backend_start  | 2022-05-27 11:23:30.833022+08
xact_start     |
query_start    |
state_change   |
wait_event_type | Activity
```

wait_event	AutoVacuumMain
state	
backend_xid	
backend_xmin	
query	
backend_type	autovacuum launcher

### 6.2.3 性能监控视图

您可以通过查询一组性能视图来监控 KingbaseES 数据库实例的运行状态。

在使用性能视图的数据时，你必须了解这些信息并非是实时更新的。每个独立的服务器进程只在进入闲置状态之前才向统计收集器传送新的统计计数；因此正在进行的查询或事务并不影响显示出来的总数。同样，收集器本身也最多每 500 毫秒发送一次新的报告。因此显示的信息总是落后于实际活动。但是由 `track_activities` 收集的当前查询信息总是最新的。

当一个服务器进程被要求显示任何这些统计信息时，它首先取得收集器进程最近发出的报告并且接着为所有统计视图和函数使用这个快照，直到它的当前事务的结尾。因此只要你继续当前事务，统计数据将会一直显示静态信息。相似地，当任何关于所有会话的当前查询的信息在一个事务中第一次被请求时，这样的信息将被收集。并且在整个事务期间将显示相同的信息。这是一种特性而非缺陷，因为它允许你在该统计信息上执行多个查询并且关联结果而不用担心那些数字会在你不知情的情况下改变。但是如果你希望用每个查询都看到新结果，要确保在任何事务块之外做那些查询。或者，你可以调用 `sys_stat_clear_snapshot()`，那将丢弃当前事务的统计快照（如果有）。下一次对统计信息的使用将导致获取一个新的快照。

一个事务也可以在视图 `sys_stat_xact_all_tables`、`sys_stat_xact_sys_tables`、`sys_stat_xact_user_tables` 和 `sys_stat_xact_user_functions` 中看到它自己的统计信息（还没有被传送给收集器）。这些数字并不像上面所述的那样行动，相反它们在事务期间持续被更新。

关于每个性能视图的描述，可以参考《KingbaseES 数据库参考手册》中的 动态性能视图。

实时刷新的性能视图：

表 6.2.2: 实时刷新的性能视图

视图名称	描述
<code>sys_stat_activity</code>	每个服务器进程一行，显示与那个进程的当前活动相关的信息，例如状态和当前查询。
<code>sys_stat_replication</code>	每一个 WAL 发送进程一行，显示有关到该发送进程连接的后备服务器的复制的统计信息。
<code>sys_stat_wal_receiver</code>	只有一行，显示来自 WAL 接收器所连接服务器的有关该接收器的统计信息。

见续表

表 6.2.2 – 续表

视图名称	描述
sys_stat_subscription	每个订阅至少一行，显示有关该订阅的工作者的信息。
sys_stat_ssl	每个连接（常规的或者复制）一行，显示在这个连接上使用的 SSL 的信息。
sys_stat_gssapi	每个连接（常规和复制）有一行，显示关于 GSSAPI 验证和加密的信息。
sys_stat_progress_create_index	每个后台运行 CREATE INDEX 或 REINDEX 的后端都有一行，显示当前的进度。
sys_stat_progress_vacuum	每个运行着 VACUUM 的后端（包括 autovacuum 工作者进程）一行，显示当前的进度。
sys_stat_progress_cluster	每个运行着 CLUSTER 或 VACUUM FULL 的后端一行，显示当前进度。

非实时刷新的性能视图：

表 6.2.3: 非实时刷新的性能视图

视图名称	描述
sys_stat_archiver	只有一行，显示有关 WAL 归档进程活动的统计信息。
sys_stat_bgwriter	只有一行，显示有关后台写进程的活动的统计信息。
sys_stat_database	每个数据库一行，显示数据库范围的统计信息。
sys_stat_database_conflicts	每个数据库一行，显示数据库范围的统计信息，这些信息的内容是关于由于与后备服务器的恢复过程发生冲突而被取消的查询。
sys_stat_all_tables	当前数据库中每个表一行，显示有关访问指定表的统计信息。
sys_stat_sys_tables	和 sys_stat_all_tables 一样，但只显示系统表。
sys_stat_user_tables	和 sys_stat_all_tables 一样，但只显示用户表。
sys_stat_xact_all_tables	和 sys_stat_all_tables 相似，但计数动作只在当前事务内发生（还没有被包括在 sys_stat_all_tables 和相关视图中）。
sys_stat_xact_sys_tables	和 sys_stat_xact_all_tables 一样，但只显示系统表。
sys_stat_xact_user_tables	和 sys_stat_xact_all_tables 一样，但只显示用户表。

见续表

表 6.2.3 – 续表

视图名称	描述
sys_stat_all_indexes	当前数据库中的每个索引一行，显示与访问指定索引有关的统计信息。
sys_stat_sys_indexes	和 sys_stat_all_indexes 一样，但只显示系统表上的索引。
sys_stat_user_indexes	和 sys_stat_all_indexes 一样，但只显示用户表上的索引。
sys_statio_all_tables	当前数据库中的每个表一行，显示有关在指定表上 I/O 的统计信息。
sys_statio_sys_tables	和 sys_statio_all_tables 一样，但只显示系统表。
sys_statio_user_tables	和 sys_statio_all_tables 一样，但只显示用户表。
sys_statio_all_indexes	当前数据库中的每个索引一行，显示与指定索引上的 I/O 有关的统计信息。
sys_statio_sys_indexes	和 sys_statio_all_indexes 一样，但只显示系统表上的索引。
sys_statio_user_indexes	和 sys_statio_all_indexes 一样，但只显示用户表上的索引。
sys_statio_all_sequences	当前数据库中的每个序列一行，显示与指定序列上的 I/O 有关的统计信息。
sys_statio_sys_sequences	和 sys_statio_all_sequences 一样，但只显示系统序列（目前没有定义系统序列，因此这个视图总是为空）。
sys_statio_user_sequences	和 sys_statio_all_sequences 一样，但只显示用户序列。
sys_stat_user_functions	每一个被跟踪的函数一行，显示与执行该函数有关的统计信息。
sys_stat_xact_user_functions	和 sys_stat_user_functions 相似，但是只统计在当前事务期间的调用（还没有被包括在 sys_stat_user_functions 中）。

比如，通过 sys\_stat\_bgwriter 视图参考后台写和检查点的统计信息：

```
test=# select * from sys_stat_bgwriter;
-[ RECORD 1 ]-----+-----
checkpoints_timed    | 344
checkpoints_req      | 0
checkpoint_write_time | 3749
checkpoint_sync_time | 1
buffers_checkpoint   | 37
buffers_clean        | 0
maxwritten_clean     | 0
buffers_backend      | 0
```

```

buffers_backend_fsync | 0
buffers_alloc         | 82
stats_reset           | 2022-05-30 09:35:50.464433+08

```

通过 sys\_stat\_database 查看每个库的事务和元组统计信息:

```

test=# select * from sys_stat_database;
-[ RECORD 1 ]-----+-----
datid           | 16384
datname         | test
numbackends     | 1
xact_commit     | 44930
xact_rollback   | 15
blks_read       | 59142
blks_hit        | 11122493
tup_returned    | 160318834
tup_fetched     | 1491817
tup_inserted    | 1380883
tup_updated     | 7324
tup_deleted     | 35999
conflicts       | 0
temp_files      | 288
temp_bytes      | 2359296
deadlocks       | 0
checksum_failures |
checksum_last_failure |
blk_read_time   | 8225.142
blk_write_time  | 6.066
stats_reset     | 2022-05-16 18:06:12.851563+08

```



---

# 第 7 章 管理控制文件

---

您可以通过控制文件读取到 KingbaseES 内部状态的各方面信息

- 什么是控制文件？

每个 KingbaseES 都有一个控制文件，它是一个记录数据库内部状态的重要文件。

- 控制文件准则

您可以通过指南了解控制文件的一些信息。

- 创建控制文件

控制文件在初始化数据库的时候创建。

- 备份控制文件

KingbaseES 使用基础备份的方式备份控制文件。

- 读取控制文件信息

KingbaseES 提供了 `sys_controldata` 工具来读取控制文件的信息。

## 7.1 什么是控制文件？

每个 KingbaseES 都有一个控制文件，它是一个记录数据库内部状态的重要文件。

控制文件中包括：

- 初始化静态信息
- WAL 及检查点的动态信息
- 一些配置信息

当 KingbaseES 启动时，控制文件必须可供数据库正确读取和写入。如果没有控制文件或控制文件不可读，数据库无法启动。

KingbaseES 的控制文件是在初始化数据库的时候创建的。

父主题：管理控制文件

## 7.2 控制文件准则

您可以通过指南了解控制文件的一些信息。

- 控制文件路径 不能改变。
- 控制文件的大小 固定不变的。

父主题：管理控制文件

### 7.2.1 控制文件路径

KingbaseES 控制文件路径不能改变的，控制文件放在 data/global 目录下名为 sys\_control。在启动数据库的时候，KingbaseES 从这个路径下读取控制文件中的相关信息。

父主题：控制文件准则

### 7.2.2 控制文件的大小

KingbaseES 中的控制文件的大小是固定不变的。

KingbaseES 控制文件在内存中尽量保持小于 512 个字节以使其适合一个典型的磁盘驱动的物理簇的大小。

KingbaseES 控制文件的物理大小是 8K，这样做是为了控制文件格式变化时保持物理大小不变。

父主题：控制文件准则

## 7.3 创建控制文件

控制文件在初始化数据库的时候创建。

- 控制文件内部信息的创建，一部分在初始化时创建，一部分在运行时动态修改。

父主题：管理控制文件

### 7.3.1 控制文件内部信息的创建

- 控制文件的静态信息在初始化时自动生成，运行过程中不允许修改，例如系统标识符。
- 控制文件的配置信息允许用户在初始化时一次性定制，不再允许修改，例如重做日志段尺寸。
- 控制文件的 WAL 及检查点的动态信息，在 KingbaseES 运行中动态修改。

父主题：创建控制文件

## 7.4 备份控制文件

KingbaseES 使用基础备份的方式备份控制文件。在基础备份执行的过程中，控制文件随着数据目录一起被备份起来。

父主题：[管理控制文件](#)

## 7.5 读取控制文件信息

KingbaseES 提供了 `sys_controldata` 工具来读取控制文件的信息。

```
sys_controldata -D /path/to/sysdata
```

父主题：[管理控制文件](#)

---

# 第 8 章 管理重做日志

---

- 什么是重做日志？

KingbaseES 中的重做日志依靠预写式日志（WAL）实现的，它是有多个重做日志段文件组成，这些段文件记录了对数据库页面所做的所有更改，任何还没有被应用到数据页面的改变可以根据其日志记录重做。

- 规划重做日志

在配置数据库实例重做日志时，您可以遵循指南。

- 手动切换重做日志

KingbaseES 允许您进行手动切换重做日志段文件。

- 验证重做日志文件中的块

您可以将数据库配置为使用一致性校验来验证重做日志文件中的块。

- 清除重做日志文件

KingbaseES 中，需要安全的删除重做日志文件，可以采用检查点的方式或者设置配置参数 `archive_cleanup_command`。

## 8.1 什么是重做日志？

KingbaseES 中的重做日志依靠预写式日志（WAL）实现的，它是有多个重做日志文件组成，这些段文件记录了对数据库页面所做的所有更改，任何还没有被应用到数据页面的改变可以根据其日志记录重做。

- WalWriter 进程

会周期性地将重做日志块写入到磁盘。

- 重做日志的内容

KingbaseES 中，对数据库所有的修改通过重做日志记录的方式放在重做日志中。

- KingbaseES 数据库如何写入重做日志

KingbaseES 中，采用了共享缓存区的方式将重做日志的 I/O 操作合并在一起。

父主题：[管理重做日志](#)

## 8.1.1 WalWriter 进程

WalWriter 进程会周期性地将重做日志块写入到磁盘。KingbaseES 中，为每个实例分配一个后台进程，所有后台进程采用共享缓存区的方式进行重做日志文件的操作。WalWriter 进程可以避免其他服务进程在事务提交时需要将重做日志同步地写入磁盘造成性能下降，事务提交记录不是在提交时同步地写入磁盘，而是在一个已知的预先设置的时间异步地写入。

父主题：[什么是重做日志？](#)

## 8.1.2 重做日志的内容

KingbaseES 中，对数据库所有的修改通过重做日志记录的方式放在重做日志中。

重做日志被存放在数据目录的 `sys_wal` 目录里，它是作为一个文件段的集合存储的，通常每个段 16MB 大小（不过这个大小可以通过 `initdb` 配置选项 `--wal-segsize` 来修改）。每个段分割成多个页，通常每个页为 8K（该尺寸可以通过 `--block-size` 配置选项来修改）。

每个页中存放多个重做日志记录，重做日志记录内容取决于它记录的事件类型。段文件的名称是不断增长的数字，从 `000000010000000000000000` 开始。目前这些数字不能回卷，不过要把所有可用的数字都用光也需要非常非常长的时间。

父主题：[什么是重做日志？](#)

## 8.1.3 KingbaseES 数据库如何写入重做日志

KingbaseES 中，采用了共享缓存区的方式将重做日志的 I/O 操作合并在一起。

共享缓存区可以理解为一个环形的共享缓存，每次缓存空间满后，会将头部的页面刷新到磁盘，同时写入新的页面，并将头部往后移动一位。具体来说，需要写入新的日志记录时：

- 当共享缓存区中有足够的空间，顺序写入到缓存区中。
- 当共享缓存区写到尾部且空间不足时，从头部刷出信息后重复利用。

父主题：[什么是重做日志？](#)

- [回收和未回收的重做日志](#)

KingbaseES 数据库一次只使用一个重做日志段文件来存储从重做日志缓冲区写入的重做记录。

- [日志切换与日志序列号](#)

日志切换是数据库停止写入当前重做日志段文件并开始写入下一个重做日志段文件。通常，当当前重做日志文件完全填满并且必须继续写入下一个重做日志文件时，会发生日志切换。

### 8.1.3.1 回收和未回收的重做日志

KingbaseES 数据库一次只使用一个重做日志段文件来存储从重做日志缓冲区写入的重做记录。在执行检查点之前，从上一个检查点起产生的重做日志是不能被回收或删除的，如果开启了归档模式，重做日志可以正常归档。执行检查点之后，上一个检查点与此次检查点之间的重做日志可以被回收或删除，具体回收还是删除需要靠配置参数决定。

父主题：[KingbaseES 数据库如何写入重做日志](#)

### 8.1.3.2 日志切换与日志序列号

日志切换是数据库停止写入当前重做日志段文件并开始写入下一个重做日志段文件。通常，当当前重做日志文件完全填满并且必须继续写入下一个重做日志文件时，会发生日志切换。

您也可以手动的切换日志。

当每个新的重做日志记录被写入时，重做日志记录被追加到重做日志中。插入位置由日志序列号（LSN）描述，该日志序列号是日志中的字节偏移量，随每个新记录单调递增。

父主题：[KingbaseES 数据库如何写入重做日志](#)

## 8.2 规划重做日志

在配置数据库实例重做日志时，您可以遵循指南。

- [重做日志段文件的大小](#)

KingbaseES 允许您配置重做日志段文件的大小。

- [重做日志文件的块大小](#)

KingbaseES 允许您配置重做日志文件的块大小。

- [重做日志文件的数量](#)

KingbaseES 允许您配置重做日志文件的数量。

- [重做日志记录延迟拷贝](#)

某些情况下，一台后备服务器使用主服务器提供的重做日志记录尽快进行恢复。使用数据的延时拷贝可以提供纠正数据丢失错误的机会。

父主题：[管理重做日志](#)

### 8.2.1 重做日志段文件的大小

KingbaseES 允许您配置重做日志段文件的大小。

- 在初始化阶段进行配置，通过 `initdb` 配置选项 `--wal-segsize` 进行选择。
- 通过 `sys_resetwal` 工具的选项 `--wal-segsize` 进行选择

---

**注意:**

- `sys_resetwal` 会清除 `sys_wal` 目录的重做日志文件。
- `sys_resetwal` 方式修改 `wal_segment_size` 后，重启后原来的数据都不会丢失，因为修改前数据库是正常关闭的。

---

父主题：规划重做日志

## 8.2.2 重做日志文件的块大小

KingbaseES 允许您配置重做日志文件的块大小。

每个重做日志段文件被分割成多个 block，默认每个 block 为 8K。该尺寸通过 `initdb` 配置选项 `--block-size` 来修改。

父主题：规划重做日志

## 8.2.3 重做日志文件的数量

KingbaseES 允许您配置重做日志文件的数量。

KingbaseES 中，在执行检查点后会删除或回收不需要的重做日志段文件。可以通过配置参数 `max_wal_size` 和 `min_wal_size` 来限制检查点后的重做日志段文件的数量。

### 8.2.3.1 max\_wal\_size

在检查点之间允许重做日志增长到的最大尺寸。这是一个软限制，在特殊的情况下重做文件尺寸可能会超过 `max_wal_size`。如果指定值时没有单位，则以兆字节为单位。默认为 1 GB。增加这个参数可能导致崩溃恢复所需的时间。这个参数只能在 `kingbase.conf` 或者服务器命令行中设置。

### 8.2.3.2 min\_wal\_size

只要重做日志磁盘用量保持在这个设置之下，在检查点时旧的重做日志文件总是被回收以便未来使用，而不是直接被删除。这可以被用来确保有足够的重做日志空间被保留来应付重做日志使用的高峰，例如运行大型的批处理任务。如果指定值时没有单位，则以兆字节为单位。默认是 80 MB。这个参数只能在 `kingbase.conf` 或者服务器命令行中设置。

父主题：规划重做日志

## 8.2.4 重做日志记录延迟拷贝

某些情况下，一台后备服务器使用主服务器提供的重做日志记录尽快进行恢复。使用数据的延时拷贝可以提供纠正数据丢失错误的机会。

通过 `kingbase.conf` 中的配置参数 `recovery_min_apply_delay`，允许备机的恢复延迟一段固定的时间，如果没有指定单位则以毫秒为单位。

注意:

- 只有在事务提交的重做日志记录上才会发生延迟，其他记录还是会被尽可能快地重放。
- 一旦恢复中的数据库已经达到一致状态，延迟就会产生，直到后备机被提升或者触发。在那之后，后备机将会结束恢复并且不再等待。
- 修改后，需要重启 `standby` 节点才能生效。

父主题：规划重做日志

## 8.3 手动切换重做日志

KingbaseES 允许您进行手动切换重做日志段文件。

`sys_wal_switch` 是 KingbaseES 的一个系统函数，它强制 KingbaseES 切换到一个新的重做日志段文件：

```
select sys_switch_wal();
```

注意:

- 返回旧重做日志文件的结尾 LSN + 1
- 如果自上次重做日志文件切换后没有产生重做日志的活动，则不会进行切换并返回当前重做日志文件的起始位置。

父主题：管理重做日志

## 8.4 验证重做日志文件中的块

您可以将数据库配置为使用一致性校验来验证重做日志文件中的块。

在 `kingbase.conf` 中开启配置参数 `wal_consistency_checking` 时，将开启 KingbaseES 的一致性校验功能。



当这个参数被启用时，被修改的任何缓冲区的全页映像及其重做日志记录都被加入到记录中。如果该记录后来被重放，系统将首先应用每个记录然后测试该记录修改的缓冲区是否符合存储的映像。在某些情况下（例如提示位），小的变动是可以接受的，并且会被忽略。任何预期之外的差别都将导致致命错误，最后中止恢复。

这个设置的默认值是空字符串，它将禁用这一特性。它可以被设置为 `all` 以检查所有记录，或者被设置为一个逗号分隔的资源管理器列表用以检查那些资源管理器产生的记录。当前，支持的资源管理器是 `heap`、`heap2`、`btree`、`bitmap`、`hash`、`gin`、`gist`、`sequence`、`spgist`、`brin` 以及 `generic`。只有超级用户可以更改这一设置。

父主题：管理重做日志

## 8.5 清除重做日志文件

KingbaseES 中，需要安全的删除重做日志文件，可以采用检查点的方式或者设置配置参数 `archive_cleanup_command`。

---

**注意：**除特殊的情况外，不要手动删除重做日志文件。

---

在检查点之后，当旧的重做日志文件不再需要时，它们将被删除或回收。

---

**注意：**

- 如果使用了重做日志归档，旧的段在被归档之前不能被删除或回收，直到它们被归档。如果重做日志归档跟不上重做日志记录生成的速度，或者 `archive_command` 重复失败，旧的重做日志文件将在 `sys_wal` 中累积，直到问题解决。
  - `wal_keep_segments` 配置参数可能需要保留一定的数据。
- 

`archive_cleanup_command` 指定了一个 shell 命令，它将在每一个重启点被执行。`archive_cleanup_command` 的目的是提供一种清除不再被后备服务器需要的旧的已归档重做日志文件的机制。

父主题：管理重做日志

---

# 第 9 章 管理归档重做日志文件

---

您可以在数据库启动前决定是否开启归档模式，并管理重做文件的归档。

- 什么是归档重做日志？

KingbaseES 数据库允许您将发生切换的重做日志段文件存储在其他位置，这个过程统称为归档重做日志。

- 开启和关闭归档重做日志

在启动 KingbaseES 数据库之前，必须决定是否开启归档。

- 控制归档

您可以为数据库开启归档模式。

- 指定归档目标

在归档重做日志之前，您必须确定要归档的目标。

- 查看归档重做日志状态

您可以通过状态文件了解归档重做日志状态。

## 9.1 什么是归档重做日志？

KingbaseES 数据库允许您将发生切换的重做日志段文件存储在其他位置，这个过程统称为归档重做日志。

从逻辑上来看，KingbaseES 数据库会产生一个无限长的顺序重做日志记录序列。KingbaseES 在物理上把这个重做日志记录序列分割成多个重做日志（每个重做日志为一个重做日志段），重做日志段文件被重用或删除时进行归档备份操作，把将被重用的重做日志段中的日志记录保存到其他位置。

父主题：管理归档重做日志文件

## 9.2 开启和关闭归档重做日志

在启动 KingbaseES 数据库之前，必须决定是否开启归档。

是否启用归档的选择取决于在数据库上运行的应用程序的可用性和可靠性要求。如果由于故障发生时，您想回退到发生故障之前的位置，请开启归档重做日志文件。

- 开启归档重做日志

当开启归档重做日志文件，重做日志将在发生切换的时候进行归档。

- 关闭归档重做日志

当关闭归档重做日志，重做日志将在回收或删除的时候，没有重做日志的拷贝。

父主题：管理归档重做日志文件

## 9.2.1 开启归档重做日志

当开启归档重做日志文件，重做日志将在发生切换的时候进行归档。

当使用物理备份方式对数据库进行备份，归档重做日志可以使数据库在启动恢复的时候恢复到之前的位置。

父主题：开启和关闭归档重做日志

## 9.2.2 关闭归档重做日志

当关闭归档重做日志，重做日志将在回收或删除的时候，没有重做日志的拷贝。

在没有使用归档重做日志时，KingbaseES 通常只创建少量段文件，并且通过重命名不再使用的重做日志段文件为更高的段编号来“回收”它们。KingbaseES 假设内容位于最后一个检查点之前的重做日志段文件是无用的且可以被回收。

父主题：开启和关闭归档重做日志

## 9.3 控制归档

您可以为数据库开启归档模式。

- 设置数据库归档模式

在数据库启动之前，需要通过配置参数的设置启动归档模式。

- 更改数据库归档模式

归档模式需要在数据库启动之前确认，更改归档模式需要重启数据库。

- 执行归档

为了方便和高效，自动存档通常是最好的。但是，您可以手动归档。

父主题：管理归档重做日志文件

## 9.3.1 设置数据库归档模式

在数据库启动之前，需要通过配置参数的设置启动归档模式。

KingbaseES 开启归档重做日志文件模式需要的配置参数。

可以通过配置参数 `archive_mode` 指定开启/关闭归档，`archive_mode` 的选项：

- `on`：开启归档模式。
- `off`：禁用归档模式。
- `always`：在普通操作期间，与 `on` 参数没有区别。但是当设置为 `always` 时，重做日志归档进程在归档恢复或者后备模式下也会被启用，所有从归档恢复或者用流复制传来的重做日志文件文件将被（再次）归档。

除了 `archive_mode` 配置参数外，需要开启归档重做日志文件，还需要：

- `wal_level` 配置参数设置为 `replica` 或更高。
- 使用 `archive_command` 配置参数指定一个 `shell` 命令。

父主题：控制归档

## 9.3.2 更改数据库归档模式

归档模式需要在数据库启动之前确认，更改归档模式需要重启数据库。

KingbaseES 数据库安装初始化时，默认关闭归档模式。

从关闭归档到启动归档的切换：

1. 关闭数据库。
2. 按照[设置数据库归档模式](#) 章节中进行配置参数。
3. 启动数据库。

父主题：控制归档

## 9.3.3 执行归档

为了方便和高效，自动存档通常是最好的。但是，您可以手动归档。

### 9.3.3.1 自动触发归档

当启动归档模式时，重做日志文件写满后开始写下一个重做日志文件，此时 KingbaseES 将执行 `archive_command` 配置参数指定的 `shell` 将写满的重做日志段文件进行归档。

### 9.3.3.2 定时触发归档

KingbaseES 中可以通过 `archive_timeout` 配置参数来强制服务器定期切换到新的重做日志段文件。如果指定值时没有单位，则以秒为单位。这个参数只能在 `kingbase.conf` 文件中或在服务器命令行上设置。

当 `archive_timeout` 配置参数大于 0 时，只要从上次段文件切换后过了参数所设置的时间量，并且已经有过任何数据库活动（包括一个单一检查点），服务器将切换到一个新的段文件（如果没有数据库活动则会跳过检查点）。写入量很少，只要到达 `archive_timeout` 会触发重做日志文件的强制切换。

如果为 0 表示不启用定时归档。

### 9.3.3.3 手动执行归档

在 KingbaseES 中，当使用 `sys_switch_wal` 系统函数时切换重做日志，这时将触发重做日志归档。

父主题：控制归档

## 9.4 指定归档目标

在归档重做日志之前，您必须确定要归档的目标。

为了给数据库管理员提供最大的灵活性，KingbaseES 不对如何归档做任何假设，而是让管理员提供一个 shell 命令来拷贝一个完整的重做日志文件到备份存储位置。该命令可以就是一个 `cp` 命令，或者是一个复杂的 shell 脚本，所有的操作都由管理员决定。

- 设置存档目标的初始化参数

您可以在 `kingbase.conf` 指定归档重做日志的拷贝命令。

- 归档的状态

在正常启动归档模式后，可能出现正常归档状态和未归档状态。

父主题：管理归档重做日志文件

### 9.4.1 设置存档目标的初始化参数

您可以在 `kingbase.conf` 指定归档重做日志的拷贝命令。

KingbaseES 使用 `archive_command` 配置参数指定本地 shell 命令，执行一个完成的重做日志文件段的归档。这个参数只能在 `kingbase.conf` 文件中或在服务器命令行上设置。

`archive_command` 中的控制符：

- `%p` 被替换成要被归档的文件的路径名
- `%f` 只被文件名替换（路径名是相对于服务器的工作目录，即集簇的数据目录）

- 如果要在命令里嵌入一个真正的 % 字符，可以使用 %%。

父主题：指定归档目标

## 9.4.2 归档的状态

在正常启动归档模式后，可能出现正常归档状态和未归档状态。

### 9.4.2.1 正常归档状态

`archive_command` 配置参数中指定的 shell 命令只在成功时返回一个零作为退出状态。

### 9.4.2.2 未归档状态

- 配置参数 `archive_command` 指定的 shell 命令执行失败，或未指定归档命令（默认为空）。重做日志归档会被临时禁用，但服务器仍会继续累计重做日志段文件，期待着一个命令被提供。
- `archive_command` 不设置拷贝指令。如果将 `archive_command` 设置为一个只返回真但不做任何事的命令（例如 `/bin/true` 或 Windows 上的 `REM`）实际上会禁用归档，也会打破归档恢复所需的重做日志文件链，因此只有在极少数情况下才能用。

父主题：指定归档目标

## 9.5 查看归档重做日志状态

您可以通过状态文件了解归档重做日志状态。

为了标识文件归档状态，KingbaseES 在数据集簇的 `sys_wal/archive_status` 目录下记录了每一重做日志段文件的状态文件。状态文件的前缀与段文件同名，以表示时间顺序的整数形式命名。状态文件后缀为 `.ready` 或者 `.done`，代表段文件的归档状态。

父主题：管理归档重做日志文件

---

# 第 10 章 管理表空间

---

表空间是将有关的数据库逻辑数据组合在一起形成的数据库存储单元。表空间允许用户定义指定的数据库对象在文件系统中存储的位置。关系及其数据文件都存储在其所属的表空间中。

- 表空间的管理指南

供您参考的管理表空间的建议和指南。

- 创建表空间

创建表空间以将有关的数据库逻辑数据（如表或索引等）组合在一起。数据文件都存储在表空间中。

- 修改表空间可用性

你可以使表空间进入脱机状态，以使它暂时不可被访问和使用，数据库的其他表空间仍然可以正常访问。你可以使已脱机的表空间重新进入联机状态，以使它可以再次被访问。

- 使用只读表空间

你可以将表空间设置为只读模式，以防止其中存储的数据被更改。

- 修改和维护表空间

你可以修改表空间的参数和属主等信息。

- 重命名表空间

你可以修改表空间名称。

- 删除表空间

当表空间不再被需要时，你可以删除表空间。在一个表空间能被删除前，其中必须没有任何数据库对象。

- 查看表空间相关信息

KingbaseES 提供表空间信息视图，供查看表空间相关的信息。

## 10.1 表空间的管理指南

供您参考的管理表空间的建议和指南。

- 使用多个表空间

使用多个表空间可以更灵活的执行数据库操作。

- 将表空间分配给用户

授予用户在表空间创建对象的权限。

父主题：管理表空间

### 10.1.1 使用多个表空间

使用多个表空间可以更灵活地执行数据库操作。当数据库具有多个表空间时，您可以：

- 将用户数据与系统表数据分开，以减少 I/O 争用。
- 将一个应用程序的数据与另一个应用程序的数据分开，以防止在必须使表空间脱机时多个应用程序受到影响。
- 将不同表空间的数据文件存储在不同的磁盘驱动器上，以减少 I/O 争用。
- 使单个表空间脱机，而其他表空间保持联机状态，从而提供更好的整体可用性。
- 通过为特定类型的数据库使用（如高更新活动、只读活动或临时段存储）保留表空间来优化表空间使用。
- 备份单个表空间。

父主题：表空间的管理指南

### 10.1.2 将表空间分配给用户

授予将要创建表、物化视图、索引和其他对象的用户在表空间中创建对象的权限。如果启用了表空间限额，则为用户在该表空间上配置合适的空间限制配额。

父主题：表空间的管理指南

## 10.2 创建表空间

创建表空间以将有关的数据库逻辑数据（如表或索引等）组合在一起。数据文件都存储在表空间中。

- 关于创建表空间

要创建新的表空间，请使用 SQL 语句 CREATE TABLESPACE。

- 加密表空间

你可以加密一个表空间来保护敏感数据。

- 临时表空间



临时表空间用于在工作内存不足时存储排序或者连接运算生成的临时关系。指定合适的临时表空间可以改善这种情况下排序和连接操作的性能。

父主题：管理表空间

## 10.2.1 关于创建表空间

要创建新的表空间，请使用 SQL 语句 CREATE TABLESPACE。只有超级用户能创建表空间，但是它们能把表空间的拥有权赋予给非超级用户。

在创建表空间之前，必须创建一个数据库来包含它。任何新建的数据库中都有初始的两个系统表空间：sys\_global 和 sys\_default。这两个表空间的物理位置位于数据集簇目录下。这两个系统表空间中记录了服务器运行的基本信息，比如系统表。系统表空间可以像任何其他表空间一样进行管理，但需要更高级别的特权，并且在某些方面受到限制，例如，不能重命名或删除系统表空间，也不能使其脱机。两个系统表空间的不同之处在于，sys\_global 表空间内的数据在整个数据集簇中对所有数据库共享。而 sys\_default 表空间则只记录本数据库的信息。

创建表空间的步骤因操作系统而异，但第一步始终是在操作系统上创建目录结构，以为该表空间指定存储的物理位置。然后，在多数操作系统上，我们需要为该目录设置正确的所有权。一个超级用户可以执行 CREATE TABLESPACE 语句以创建表空间，我们会指定该目录作为该表空间的 LOCATION。此外选择一个合适的表空间名，该名称不能以 sys\_ 开头，因为 sys\_ 名称是为系统表空间保留的。

可以使用 ALTER TABLESPACE 或者 ALTER DATABASE 语句来更改表空间。同样的，您必须对该 TABLESPACE 或 DATABASE 具备操作的权限。

父主题：创建表空间

## 10.2.2 加密表空间

你可以对表空间中的数据进行加密，以符合安全需求。

父主题：创建表空间

## 10.2.3 临时表空间

临时表空间用于存储会话工作期间的临时数据。这些数据包括：

- 排序和连接操作的中间结果。
- 物化操作存储的数据。
- 临时表和临时索引（缺省状态）。

和普通表空间一样，临时表空间中的数据以临时文件和数据文件的形式存储，其中，排序等中间结果和物化结果会以临时文件存储，而临时表和临时索引则以数据文件存储。

在 KingbaseES 中，临时表空间的创建和使用和普通表空间无异，只是你需要在创建表空间之后，在参数 temp\_tablespaces 中指定哪些表空间将作为临时表空间使用。参数 temp\_tablespaces 的值是一个表空间名的列表，

当列表中有多个成员时，每个临时关系会被随机的存储在其中一个临时表空间中。如果该参数未被配置，数据库将缺省的使用默认表空间，通常为 `sys_default`，兼作为临时表空间使用。

临时表也可以建立在用户指定的表空间中，因此临时表空间只是一个缺省的选择。

父主题：创建表空间

## 10.3 修改表空间可用性

你可以使表空间进入脱机状态，使它暂时不可被访问和使用，数据库的其他表空间仍然可以正常访问。你可以使已脱机的表空间重新进入联机状态，以使它可以再次被访问。

- 使表空间脱机

使一个表空间进入脱机状态，使它暂时不可被访问和使用。

- 使表空间联机

使一个表空间进入联机状态，以使它可以再次被访问。这是表空间的初始状态和通常状态。

父主题：管理表空间

### 10.3.1 使表空间脱机

使表空间脱机以使其中的数据不可被访问。只有表空间属主或超级用户具备执行该操作的权限

你可能希望表空间脱机的原因有：

- 使数据库的一部分数据不可用，同时保持其余部分可以正常访问。
- 在某应用程序维护和升级期间，保护其所属的数据不被更改和访问。

使一个表空间脱机的方式：

- 执行 SQL 语句 `ALTER TABLESPACE`，其中包含 `OFFLINE` 子句。

不能使系统表空间 `sys_default` 和 `sys_global` 进入脱机模式。

在使表空间脱机之前，应小心那些已将默认表空间设置为该表空间的用户，为其变更默认表空间，否则这些用户可能会无法正常的访问数据。

例如，使用以下语句使表空间脱机：

```
ALTER TABLESPACE tablesp OFFLINE;
```

父主题：修改表空间可用性

## 10.3.2 使表空间联机

使表空间联机，使其中的数据可以被正常访问。联机状态是表空间的初始状态和通常状态，以便用户正常的访问表空间中的数据。

使一个表空间联机的方式：

- 执行 SQL 语句 ALTER TABLESPACE，其中包含 ONLINE 子句。

例如，使用以下语句使表空间脱机：

```
ALTER TABLESPACE tablesp ONLINE;
```

父主题：修改表空间可用性

## 10.4 使用只读表空间

你可以将表空间设置为只读模式，以防止其中存储的数据被更改。

- 关于只读表空间

只读表空间是被设置为只读模式的表空间，其中的数据不能被更改。

- 使表空间只读

使用包含 READ ONLY 子句的 ALTER TABLESPACE 命令以使表空间进入只读模式。

- 使只读表空间可读写

使用包含 READ WRITE 子句的 ALTER TABLESPACE 命令以使表空间进入读写模式。

父主题：管理表空间

### 10.4.1 关于只读表空间

将表空间设置为只读模式，以使表空间中的数据只能被读取，不能被更新和写入。

只读表空间的主要目的是消除对数据库大量静态数据执行备份和恢复的需要。只读表空间还提供了一种保护历史数据不被篡改的方法。将表空间设置为只读可防止用户对表空间中的所有表进行更新，而与用户的更新权限无关。

只读表空间中的对象，如表、索引等，不能被创建、删除、更改定义。目前不支持将只读表空间中的数据放在只读设备，如 CD-ROM 等上存储。

父主题：使用只读表空间

## 10.4.2 使表空间只读

通过 ALTER TABLESPACE 命令可以将表空间设置为只读模式。所有表空间被初始化时都是可读写的。只有表空间属主或超级用户具备执行该操作的权限。

使一个表空间只读的方式：

- 执行 SQL 语句 ALTER TABLESPACE，其中包含 READ ONLY 子句。

例如，使用以下语句使表空间只读：

```
ALTER TABLESPACE tablesp READ ONLY;
```

父主题：[使用只读表空间](#)

## 10.4.3 使只读表空间可读写

使只读表空间置为可读写模式，以接受对数据文件的写入操作。

使一个表空间可读写的方式：

- 执行 SQL 语句 ALTER TABLESPACE，其中包含 READ WRITE 子句。

例如，使用以下语句使表空间可读写：

```
ALTER TABLESPACE tablesp READ WRITE;
```

父主题：[使用只读表空间](#)

## 10.5 修改和维护表空间

你可以修改表空间的参数和属主等信息。

只有表空间属主或超级用户具备执行该操作的权限。

修改表空间的参数的方式：

- 执行 SQL 语句 ALTER TABLESPACE，其中包含 SET tablespace\_option = value 子句。

例如，使用以下语句修改表空间的参数：

```
ALTER TABLESPACE tablesp SET tablespace_option = value;
```

合法的 tablespace\_option 有 seq\_page\_cost、random\_page\_cost 和 effective\_io\_concurrency。它们功能的介绍详见：[ALTER TABLESPACE](#)

修改表空间的属主的方式：

- 执行 SQL 语句 ALTER TABLESPACE，其中包含 OWNER TO 子句。

例如，使用以下语句修改表空间的属主：

```
ALTER TABLESPACE tablesp OWNER TO user;
```

父主题：管理表空间

## 10.6 重命名表空间

你可以修改表空间的名称。只有表空间属主或超级用户具备执行该操作的权限。

重命名表空间的方式：

- 执行 SQL 语句 ALTER TABLESPACE，其中包含 RENAME TO 子句。

例如，使用以下语句重命名表空间：

```
ALTER TABLESPACE tablesp RENAME TO newtablesp;
```

父主题：管理表空间

## 10.7 删除表空间

如果不再需要表空间，你可以删除它。在一个表空间能被删除前，其中必须没有任何数据库对象。只有表空间属主或超级用户具备执行该操作的权限。

系统表空间 sys\_default 和 sys\_global 不能被删除。正在被用作临时表空间的表空间不能被删除，因为有临时文件存在其中。

删除表空间的方式：

- 执行 SQL 语句 DROP TABLESPACE。

例如，使用以下语句删除表空间：

```
DROP TABLESPACE tablesp;
```

父主题：管理表空间

## 10.8 查看表空间相关信息

以下系统表和视图提供了表空间有关的相关信息。

视图	描述
SYS_TABLESPACE	记录所有表空间信息的系统表
SYS_DATABASE	查询所有数据库使用的默认表空间
DBA_TABLESPACE、USER_TABLESPACE	显示所有表空间信息的易阅读系统视图

父主题：管理表空间

# 第 11 章 管理数据文件和临时文件

在 KingbaseES 中，数据文件的创建、命名和删除均由数据库内部进行管理，这些操作对用户是透明的。但你仍可以依照以下建议管理数据文件和临时文件。

**注意：**临时文件是一种特殊的数据文件，用于存储临时关系的数据。临时文件所属的表空间由参数 `temp_tablespaces` 控制。

- 数据文件的管理指南

供您参考的管理数据文件的指南，包含了管理数据文件的一些建议和注意事项。

## 11.1 数据文件的管理指南

供您参考的管理数据文件的建议和指南。

- 关于数据文件

数据文件是一系列文件系统上的物理文件，用于存储数据库中的逻辑数据。在 KingbaseES 中的数据文件不需要显式创建，而是随着表和索引等对象操作而自动创建和维护。

- 选择适当的文件位置

数据文件的位置取决于其所属的表空间。为每个表指定合适的表空间就可以为数据文件选择合适的物理位置，合理地利用硬件资源。

- 将数据文件和重做日志文件分开存储

数据文件和重做日志文件不应该存放在同一个磁盘驱动器上。如果数据文件和重做日志文件在同一个磁盘驱动器上，那么在该磁盘驱动器出现故障时，我们将无法通过日志重做而恢复到正确的数据。

父主题：管理数据文件和临时文件

### 11.1.1 关于数据文件

数据文件是存储数据库中逻辑数据的物理文件。KingbaseES 为每个数据库对象，如表、索引或视图，生成至少一个数据文件存储其中的数据。对于普通关系，这些文件会以表或索引的 filenode 号命名。对于临时关系，文件名的形式为 tBBB\_FFF，其中 BBB 是创建该文件的后台会话的后台 ID，FFF 是文件节点号。此外关系中的数据超过 1GB 之后，该关系会被划分成 1G 大小的段，每个段是一个单独的数据文件，第一个段的文件名和文件节点相同；随后的段被命名为 filenode.1、filenode.2 等等。

另请参阅：

《KingbaseES 数据库概念》中的 存储结构。

父主题：数据文件的管理指南

### 11.1.2 选择适当的文件位置

数据文件的位置取决于其所属的表空间。对于普通关系，数据文件会被存放在该关系所属的表空间中。对于临时关系，临时文件会被存放在 temp\_tablespaces 指定的表空间中。为每个表指定合适的表空间就可以为数据文件选择合适的物理位置，合理地利用硬件资源。

例如，如果有多个磁盘驱动器可用于存储数据，每个磁盘驱动器上创建了不同的表空间，请考虑将可能冲突访问的数据文件放在不同的磁盘上，这样用户执行查询时，两个磁盘驱动器可以同时工作，同时检索数据。

父主题：数据文件的管理指南

### 11.1.3 将数据文件和重做日志文件分开存储

数据文件和重做日志文件不应该存放在同一个磁盘驱动器上。如果数据文件和重做日志文件在同一个磁盘驱动器上，那么在该磁盘驱动器出现故障时，我们将无法通过日志重做而恢复到正确的数据，而只能通过备份文件进行恢复。

如果我们的重做日志文件有多份副本，那么丢失所有副本的概率很低，这种情况我们才可以将重做日志文件和数据文件放在相同的磁盘驱动器上。

父主题：数据文件的管理指南



---

# 第 12 章 模式对象的管理

---

您可以在 KingbaseES 数据库中创建和管理多种类型的模式对象。

- 关于默认模式

用户只要有权限就可以访问所连接数据库中的任何模式中的对象，在创建数据库时，KingbaseES 会默认创建两个模式：SYS\_CATALOG 和 PUBLIC。

- 分析表和索引以及聚类

您可以收集模式对象的统计信息、分析统计信息以及验证模式对象。

- 清空表

您可以将一个表中的所有行清空，但表本身保留。

- 启用和禁用触发器

数据库触发器是存储在数据库中并在发生特定情况（如向表中添加行）时激活（“触发”）的过程。

- 管理完整性约束

完整性约束是限制表中一列或多列的值的规则。约束子句可以出现在或语句中，并标识受约束影响的列，并标识约束的条件。CREATE TABLEALTER TABLE

- 重命名模式对象

有多种方法可以重命名对象。

- 管理对象名称解析

SQL 语句中引用的对象名称可以由多个部分组成，以句点分隔。KingbaseES 数据库执行特定操作来解析对象名称。

- 切换到其他模式

使用 SQL 的 ALTER SESSION SET SEARCH\_PATH 语句切换到其他模式。

- 显示模式对象的信息

KingbaseES 数据库提供了一个 PL/SQL 包，使您能够确定创建对象的 DDL 以及可用于显示有关模式对象的信息的视图。

## 12.1 关于默认模式

用户只要有权限就可以访问所连接数据库中的任何模式中的对象，在创建数据库时，KingbaseES 会默认创建两个模式：SYS\_CATALOG 和 PUBLIC。

一个数据库可以包含一个或多个命名的模式（SCHEMA），一个模式内可以包含多个表。不同的模式中的表名可以相同，例如，schema1 和 schema2 中可以分别包含名称为 mytable 的表，但是同一个模式中的表不能同名。模式类似于操作系统级的目录，但模式不能嵌套。其中 SYS\_CATALOG 模式用于存放系统表和所有内置的数据类型、函数和操作符。

当用户没有自己的模式并且其创建或使用数据库对象没有指定模式时，默认使用 PUBLIC 模式。模式通常在以下情况下使用：

- 多个用户使用同一个数据库而不会相互影响。
- 对数据库中的对象进行逻辑分组，更便于管理。
- 各个应用分别使用各自的模式，以避免命名冲突。

另请参阅：

《KingbaseES 安全指南》获取更多关于权限管理和模式对象相关信息

父主题：[模式对象的管理](#)

## 12.2 分析表和索引以及聚类

您可以收集模式对象的统计信息，分析统计信息，并验证模式对象。

- [关于分析表、索引和聚类](#)

您可以收集关于模式对象的信息并分析这些信息。

- [使用 sys\\_stat\\_statement 搜集执行语句的统计信息](#)

sys\_stat\_statement 是 kingbaseES 系统的一个扩展组件，它提供了所有执行语句的统计信息，可以帮助找出哪种类型的查询很慢以及多久调用一次查询，帮助您分析对象的 SQL 语句，选择最有效的执行计划。

父主题：[模式对象的管理](#)

### 12.2.1 关于分析表、索引和聚类

您可以收集关于模式对象的信息并分析这些信息。

分析一个模式对象（表、索引）：

- 收集和管理统计数据
- 验证其存储格式的有效性

- 标识表的迁移行和链接行

父主题：分析表和索引以及聚类

## 12.2.2 使用 sys\_stat\_statement 搜集执行语句的统计信息

sys\_stat\_statement 是 kingbaseES 系统的一个扩展组件，它提供了所有执行语句的统计信息，可以帮助找出哪种类型的查询很慢以及多久调用一次查询，帮助您分析对象的 SQL 语句，选择最有效的执行计划。

使用这个模块的执行步骤：

1. 在 kingbase.conf 里添加预加载项：

```
shared_preload_libraries = 'sys_stat_statements'
```

2. 重启数据库服务器

3. 在执行的数据库里创建扩展：

```
test=# create extension sys_stat_statements;
CREATE EXTENSION
```

**注意：** 从 KingbaseES V8R6 版本开始，sys\_stat\_statements 插件已经内置化，初始化数据库实例的时候，就会被自动加载。不过该功能默认关闭，需要设置：sys\_stat\_statements.track = "top" 才能正常使用。

另请参阅：

《KingbaseES 数据库性能调优指南》了解有关 sys\_stat\_statements 工具的详细描述

父主题：分析表和索引以及聚类

## 12.2.3 分析表、索引、聚类和物化视图

ANALYZE 收集数据库中的表的内容统计信息，并且将分析结果存储至系统表 sys\_statistic。接下来，查询计划就会使用这些统计信息，以助于决定选择出最有效的查询执行计划。

以下语句分析了该表：emp

```
ANALYZE emp;
```

您可以通过包含 VERBOSE 选项来显示进度消息。语句如下：

```
ANALYZE VERBOSE emp;
```

另请参阅：

《KingbaseES SQL 语言参考手册》有关 ANALYZE 语句的详细信息

父主题：分析表和索引以及聚类

## 12.3 清空表

您可以将一个表中的所有行清空，但表本身保留。例如，某单位的一个包含月度数据的表，在每个月底，必须在对其数据进行归档后清空该表（清空所有行）。

保留表情况下清空整个表，存在多种方式：

- 使用 DELETE 清空表

您可以使用 SQL 的 DELETE 语句删除表中的行。

- 使用 DROP 和 CREATE 清空表

您可以先删除一个表，然后重新创建表的方式变相实现清空表的目的。

- 使用 TRUNCATE 清空表

可以使用 TRUNCATE 语句删除表中的所有行。

父主题：模式对象的管理

### 12.3.1 使用 DELETE 清空表

您可以使用 SQL 的 DELETE 语句删除表中的行。

例如，下面的语句从 class 表中删除所有行：

```
DELETE FROM class;
```

在使用 DELETE 语句时，如果表中存在许多行，在删除这些行时将消耗大量系统资源。例如，CPU 时间、重做日志空间、撤销表和相关索引的段空间都需要资源。另外，当删除每一行时，也有可能触发触发器。以前分配给生成的空表空间仍然与该对象关联。DELETE 还可以通过 WHERE 条件子句删除满足条件的行，而 TRUNCATE 和 DROP 直接影响整个对象。

另请参阅：

《KingbaseES SQL 语言参考手册》获取关于 DELETE 语句的语法和其他信息

父主题：清空表

### 12.3.2 使用 DROP 和 CREATE 清空表

您可以先删除一个表，然后重新创建该表的方式变相实现清空表的目的。

例如，以下语句删除并重新创建 class 表:

```
DROP TABLE class;  
CREATE TABLE class ( ... );
```

父主题: 清空表

### 12.3.3 使用 TRUNCATE 清空表

可以使用 TRUNCATE 语句删除表中的所有行。例如，下面的语句清空 class 表:

```
TRUNCATE emp;
```

使用 TRUNCATE 语句提供快速移除表中所有行的机制。TRUNCATE 语句不会生成任何撤消信息，会立即提交，并且它是 DDL 语句。TRUNCATE 语句不会影响与被清空的表相关的任何结构 (约束和触发器) 或授权。您可以清空自己模式中的任何表。

当 TRUNCATE 语句从表中删除行时，不会触发与表关联的触发器。此外，如果启用了审计，那么 TRUNCATE 语句不会生成与 DELETE 语句对应的任何审计信息。相反，将为发出的 TRUNCATE 语句生成单个审计记录。

另请参阅:

《KingbaseES SQL 语言参考手册》获取语法和其他关于 TRUNCATE TABLE 语句的信息

父主题: 清空表

## 12.4 启用和禁用触发器

数据库触发器是存储在数据库中的由事件驱动的特殊过程，并在特定条件 (例如向表中添加一行) 发生时激活 (“触发”)。

您可以使用触发器来补充数据库的标准功能，从而实现高度定制的数据库管理系统。例如，可以创建一个触发器来限制对表的 DML 操作，比如只允许 DML 语句在正常业务时间内执行。

- 关于启用和禁用触发器

触发器相比约束更加灵活，可以实现比外约束更为复杂的检查和操作，更精细和更强大的数据处理能力。

- 启用触发器

您可以使用带有 ENABLE 选项的 ALTER TRIGGER 语句启用禁用状态的触发器。

- 禁用触发器

您可以使用带有 DISABLE 选项的 ALTER TRIGGER 语句禁用触发器。

## 12.4.1 关于启用和禁用触发器

触发器相比约束更加灵活，可以实现比外约束更为复杂的检查和操作，更精细和更强大的数据处理能力。

数据库触发器可以与表、模式或数据库相关联，它们可以在以下情况被隐式触发：

- 对关联的表执行 DML 语句 (INSERT、UPDATE、DELETE)
- 某些 DDL 语句 (例如:ALTER、CREATE、DROP) 在数据库或模式中的对象上执行

创建触发器请使用 CREATE TRIGGER 语句，可以定义触发器函数是要在事件之前、之后被调用、还是取代该事件。下面的示例在表 class 上创建了一个 check\_update\_trigger 触发器。在执行任何指定 UPDATE 语句之前触发该触发器，最终会执行触发器函数 check\_account\_update。

```
CREATE TRIGGER class_update_trigger
  BEFORE
  UPDATE ON class
  FOR EACH ROW
  EXECUTE FUNCTION check_class_update();
```

之后可以通过执行 DROP TRIGGER 语句从数据库中删除触发器。

要使用 ALTER TABLE 语句启用或禁用触发器，您必须是这个表的拥有者或者对该表具有 ALTER 权限。如果使用 ALTER TRIGGER 语句启用或禁用单个触发器，您必须拥有该触发器。

另请参阅：

- 《KingbaseES 数据库概念》获取更详细的触发器描述细节
- 《KingbaseES SQL 语言参考手册》获取关于 CREATE TRIGGER 的语法及其示例信息
- 《KingbaseES PL/SQL 过程语言参考手册》参考获取更多关于创建和使用触发器信息

父主题：启用和禁用触发器

## 12.4.2 启用触发器

您可以使用带有 ENABLE 选项的 ALTER TRIGGER 语句启用禁用状态的触发器。

要在 class 表中启用名为 class\_trigger 的触发器 (禁用状态)，输入以下语句：

```
ALTER TRIGGER class_trigger ENABLE;
```

上述用于启动特定的触发器，如果要启用特定表的所有触发器，请使用带有 ENABLE ALL TRIGGERS 选项的 ALTER TABLE 语句。如下示例，启用 class 表定义的所有触发器，则可以输入以下语句：

```
ALTER TABLE class
  ENABLE TRIGGERS ALL;
```

另请参阅:

《KingbaseES SQL 语言参考手册》获取关于 ALTER TRIGGER 语句的详细信息

父主题: 启用和禁用触发器

### 12.4.3 禁用触发器

可以使用带有 DISABLE 选项的 ALTER TRIGGER 语句禁用触发器。

如果满足以下条件之一, 请考虑暂时禁用触发器:

- 触发器引用的对象不可用。
- 您必须执行大型数据加载, 并希望它在不触发触发器的情况下快速进行。
- 您正在将数据加载到触发器应用的表中。

要禁用 class 表上的 class\_trigger 触发器, 输入以下语句:

```
ALTER TRIGGER class_trigger DISABLE;
```

可以使用带有 DISABLE ALL TRIGGERS 选项的 ALTER TABLE 语句同时禁用与表关联的所有触发器。如下示例, 禁用 class 表中定义的所有触发器, 可以输入以下语句:

```
ALTER TABLE class
DISABLE ALL TRIGGERS;
```

父主题: 启用和禁用触发器

## 12.5 管理完整性约束

完整性约束用于保证关系型数据库中数据的精确性和一致性。对于关系型数据库来说, 数据完整性由参照完整性 (referential integrity, RI) 来保证。完整性约束是限制表中一个或多个列的值的规则。约束子句可以出现在 CREATE TABLE 或 ALTER TABLE 语句中, 并标识一个或多个列受约束影响, 以及标识约束的条件。

- 完整性约束状态

完整性约束强制实施业务规则并防止将无效信息输入到表中。

- 创建约束

约束可以在创建表时同时指定, 也可以在创建表之后在指定。

- 修改或删除现有的约束

可以使用 ALTER TABLE 语句启用、禁用、修改或删除约束。当数据库使用 UNIQUE 或 PRIMARY KEY 索引强制约束时, 如果与该索引相关的约束被删除或禁用, 则索引将被删除, 除非您另行指定。

- 延迟约束检查

当数据库检查约束时，如果约束不满足，它就发出错误信号。当您发出 SET CONSTRAINTS 语句时，SET CONSTRAINTS 模式将在事务的整个过程中持续，或者直到另一个 SET CONSTRAINTS 语句重置该模式。

- [查看约束信息](#)

KingbaseES 数据库提供了一组视图，使您能够看到表上的约束定义，并识别在约束中指定的列。

另请参阅:

《KingbaseES 数据库概念》获取更多关于完整性约束的信息

《KingbaseES 数据库开发指南》关于约束的详细信息和在应用程序中使用完整性约束的示例

## 12.5.1 完整性约束状态

完整性约束强制实施业务规则并防止将无效信息输入到表中。

- [关于完整性约束状态](#)

您可以指定约束是启用 (ENABLE) 或禁用 (DISABLE) 状态。

- [关于禁用约束](#)

要让表数据符合完整性约束定义的规则，则必须保证约束处于启用状态，但在某些情况下您也应考虑禁用约束。

- [关于启用约束](#)

当启用约束时，不满足约束的数据不能插入到表中。

- [完整性约束的有效使用: 顺序](#)

以特定顺序使用完整性约束状态是很重要的。

父主题: [管理完整性约束](#)

### 12.5.1.1 关于完整性约束状态

您可以指定约束是启用 (ENABLE) 或禁用 (DISABLE) 状态。

如果约束状态是 ENABLE，在数据库表中输入或更新数据时要进行约束检查，并阻止输入不符合约束的数据输入数据库。如果该约束状态是 DISABLE，则输入或者更新数据时不会进行约束规则检查。另外，您可以指定表中的现有数据必须符合约束 (VALIDATE)。相反，如果指定 NOVALIDATE 状态，则不能确保现有数据符合要求。

在表上定义的完整性约束可以处于以下状态之一:

- ENABLE, VALIDATE

将对表的现有及后续新加入的数据执行约束检查。

- ENABLE, NOVALIDATE

将仅对后续新加入的数据做强制性约束检查，而不管表中的任何现有数据。



- DISABLE, VALIDATE

约束规则关闭，但为了保证有效性不被破坏，数据库会被置为只读状态，禁止对表的 DML 操作。

- DISABLE, NOVALIDATE

约束规则关闭，可以对表的约束列进行 DML 操作。

一个 VALIDATE 状态的约束表示数据库内现有数据符合约束规则，设置或者修改约束为 ENABLE 状态时如果没指定有效性，则 VALIDATE 将是默认值。相反，NOVALIDATE 状态表示表内现有数据未做约束规则检查，如果设置或者修改约束为 DISABLE 状态时如果没指定有效性，则 NOVALIDATE 将是默认值。

父主题：完整性约束状态

### 12.5.1.2 关于禁用约束

要让表数据符合完整性约束定义的规则，则必须保证约束处于启用状态，但在某些情况下您也应考虑禁用约束。出现以下性能方面的原因，可以考虑暂时禁用表的完整性约束：

- 当将大量数据加载到表中时
- 执行对表进行大量更改的批处理操作时
- 每次导入或导出一个表

上述这三种情况下，临时禁用完整性约束可以提高操作的性能。当某个约束被禁用时，可能会输入不符合该约束的数据。因此，当我们完成上述类似的操作项后应总是重新启用约束。

父主题：完整性约束状态

### 12.5.1.3 关于启用约束

当启用约束时，不满足约束的数据不能插入到表中。

禁用约束后，违反约束的行则可以插入到表中，该行则被认为是约束异常。而且如果约束处于启用非校验状态 (ENABLE,NOVALIDATE)，则在禁用约束时插入的违反约束的数据将依然被保留。违反约束的行必须更新或者删除，才能将约束置于 VALIDATE 状态。

当表的约束处于 ENABLE,NOVALIDATE 状态时，意味着表中现有数据可以不符合约束规则，但后面新增的数据必须进行规则校验，这对于数据仓库来说特别有用。因为基本上数据仓库都是使用增量更新，在停用约束后如果采用启用校验约束状态，则需要对大量数据进行校验，影响性能。

另外说明：校验一个已经启用的约束，在校验期间不需要任何 DML 锁，因此在校验期间可以保证没有引入违反约束的数据。

父主题：完整性约束状态

### 12.5.1.4 完整性约束的有效使用: 顺序

一般可以按照以下的顺序来使用约束，以确保最佳的效益：

停用状态

执行操作 (装载、导出、导入)

启用非校验状态

启用状态

好处是：

不持有锁

所有的约束能够被并发设置为启用状态

约束启用是并行实现的

允许表进行并发操作

父主题：完整性约束状态

## 12.5.2 创建约束

约束可以在创建表时同时指定，也可以在创建表之后在指定。如果与表同时创建，那么在创建表的 CREATE 语句中通过 CONSTRAINT 关键字指定约束的名称和约束类型。

- 在创建表时指定约束

您可以在创建表时同时定义约束。

- 在创建表后指定约束

您也可以在创建表之后再定义约束。

父主题：管理完整性约束

### 12.5.2.1 在创建表时指定约束

您可以在创建表时同时定义约束。

创建表的格式如下：

```
CREATE TABLE 表名 (
    列 1 数据类型 CONSTRAINT 约束名 1 约束类型
    列 2 数据类型 CONSTRAINT 约束名 2 约束类型
);
```

其中约束名必须是唯一指定的名称，如果用户不使用 CONSTRAINT 指定名称，则系统会自动生成。

创建表和定义约束一种常见的方式如下：

```
CREATE TABLE student(
    stu_no number(8) PRIMARE KEY,
    stu_name char(8) NOT NULL,
    gender char(2) CHECK(gender IN ('男','女')),
    birthday date,
    class varchar(40)
);
```

这种方式就是在定义表的列时直接指定该列约束，该方式称之为列级越苏。在创建 student 表时定义了三个约束，其中 stu\_no 列为主键，stu\_name 列约束指定姓名不能为空，gender 列约束指出该列值只能是男或者女。这些约束由于没有名字，系统将自动为它们指定各自的名字。如果想要为每个约束定义名字，可以参考如下方式：

```
CREATE TABLE student(
    stu_no number(8) CONSTRAINT con_sno PRIMARE KEY,
    stu_name char(8) CONSTRAINT con_name NOT NULL,
    gender char(2) CONSTRAINT con_gen CHECK(gender IN ('男','女')),
    birthday date,
    class varchar(40)
);
```

上述方式指定的约束都是和列一起定义的，实际上约束也可以在每列定以后再创建约束，具体的语句如下：

```
CREATE TABLE student(
    stu_no number(8),
    stu_name char(8) CONSTRAINT con_name NOT NULL,
    gender char(2),
    birthday date,
    class varchar(40)
    --定义约束，格式：CONSTRAINT 约束名 约束类型 (列名)
    CONSTRAINT con_sno PRIMARE KEY(stu_no),
    CONSTRAINT con_gen CHECK(gender IN ('男','女'))
);
```

父主题：[创建约束](#)

### 12.5.2.2 在创建表后指定约束

您也可以在创建表之后再定义约束。

此方式创建约束可能会带来这样的问题：如果表中已有的数据不满足即将添加的约束，那么约束是无法添加的。因此通常还是建议在创建表时定义约束，这需要根据您的情况在创建表时充分评估。

创建表后添加约束是通过 ALTER 执行，实际上该方式是对表结构的修改，该方式添加约束不再是列级约束而是表级约束。ALTER 语句添加约束的格式为：

```
ALTER TABLE 表名 ADD CONSTRAINT 约束名 约束类型 (列名)
```

其中 CONSTRAINT 和约束名是可以省略的，省略后由系统自动生成，如果要指定多个约束，可以在 ADD 子句的括号被使用逗号分隔即可。现在假设 student 没有定义任何约束且已创建，您可以为这个

```
ALTER TABLE student
    ADD CONSTRAINT con_sno PRIMARE KEY (stu_no);

ALTER TABLE student
    ADD CONSTRAINT con_gender check(gender in ('男','女'));
```

特别注意，对于 NOT NULL 约束，你需要通过 ALTER COLUMN 子句添加。语法格式如下：

```
ALTER TABLE 表名 ALTER COLUMN 列名 SET NOT NULL;
```

往 student 表添加非空约束，具体的语句如下：

```
ALTER TABLE student ALTER COLUMN stu_name SET NOT NULL;
```

父主题：[创建约束](#)

### 12.5.3 修改或删除现有的约束

可以使用 ALTER TABLE 语句启用、禁用、修改或删除约束。当数据库使用 UNIQUE 或 PRIMARY KEY 索引强约束时，如果与该索引相关的约束被删除或禁用，则索引将被删除，除非您另行指定。当启用外键引用 PRIMARY 或 UNIQUE 键时，不能禁用或删除 PRIMARY 或 UNIQUE 键约束或索引。

- [禁用和启用约束](#)

您可以禁用已启用的完整性约束和启用已禁用的完整性约束。

- [重命名约束](#)

ALTER TABLE……RENAME CONSTRAINT 语句使您能够为表重命名任何当前存在的约束。新的约束名不能与用户的任何现有约束名冲突。

- [删除约束](#)

如果完整性约束所执行的规则不再为真，或者不再需要该约束，则可以删除该约束。

父主题：[管理完整性约束](#)

### 12.5.3.1 禁用和启用约束

您可以禁用已启用的完整性约束和启用已禁用的完整性约束。以下语句禁用完整性约束。第二条语句指定要保留相关的索引。

```
ALTER TABLE dept
  DISABLE CONSTRAINT dname_ukey;

ALTER TABLE dept
  DISABLE PRIMARY KEY KEEP INDEX,
  DISABLE UNIQUE (dname, loc) KEEP INDEX;
```

以下语句启用 novalidate disable 状态的完整性约束:

```
ALTER TABLE dept
  ENABLE NO CONSTRAINT dname_ukey;

ALTER TABLE dept
  ENABLE NO PRIMARY KEY,
  ENABLE NO UNIQUE (dname, loc);
```

以下语句启用或验证禁用的完整性约束:

```
ALTER TABLE dept
  MODIFY CONSTRAINT dname_key ;

ALTER TABLE dept
  MODIFY PRIMARY KEY ENABLE NO;
```

以下语句启用禁用状态的完整性约束:

```
ALTER TABLE dept
  ENABLE CONSTRAINT dname_ukey;

ALTER TABLE dept
  ENABLE PRIMARY KEY,
  ENABLE UNIQUE (dname, loc);
```

父主题: [修改或删除现有的约束](#)

### 12.5.3.2 重命名约束

ALTER TABLE……RENAME CONSTRAINT 语句使您能够为表重命名任何当前存在的约束。新的约束名不能与用户的任何现有约束名冲突。下面的语句重命名表 dept 的 dname\_ukey 约束为 dname\_unikey:

```
ALTER TABLE dept
  RENAME CONSTRAINT dname_ukey TO dname_unikey;
```

父主题：[修改或删除现有的约束](#)

### 12.5.3.3 删除约束

如果完整性约束所执行的规则不再为真，或者不再需要该约束，则可以删除该约束。可以使用带有 DROP CONSTRAINT 子句的 ALTER TABLE 语句删除约束：

以下语句删除了完整性约束 dept\_fkey：

```
ALTER TABLE emp
  DROP CONSTRAINT dept_fkey;
```

父主题：[修改或删除现有的约束](#)

## 12.5.4 延迟约束检查

当数据库检查约束时，如果约束条件不满足，它将报错。当您执行 SET CONSTRAINTS 语句时，SET CONSTRAINTS 模式将在事务的整个过程中持续，延迟约束检查会在事务结束后再检查。

- [将所有约束设置为延迟](#)

当必须为事务延迟约束时，必须在实际开始处理应用程序中用于操作数据的任何数据之前设置所有延迟约束。

- [检查提交 \(可选\)](#)

在事务 COMMIT 之前，可以用 SET CONSTRAINTS ALL IMMEDIATE 语句来手工检查约束的违反情况。虽然在事务结束时会自动隐性执行这条语句，但是也是有一定的存在意义的，例如希望在错误时定义某些操作。

父主题：[管理完整性约束](#)

### 12.5.4.1 将所有约束设置为延迟

当必须为事务延迟约束时，必须在实际开始处理应用程序中用于操作数据的任何数据之前设置所有延迟约束。使用下面的 DML 语句设置所有可延期约束：

```
SET CONSTRAINTS ALL DEFERRED;
```

SET CONSTRAINTS 语句只应用于当前事务。只要约束存在，在创建约束时指定的默认值就保持不变。ALTER SESSION SET CONSTRAINTS 语句只适用于当前会话。

另请参阅：

《KingbaseES SQL 语言参考手册》获取有关 SET CONSTRAINTS 语句详细语法和描述

父主题: [延迟约束检查](#)

### 12.5.4.2 检查提交 (可选)

您可以在提交之前通过发出 SET CONSTRAINTS ALL IMMEDIATE 语句检查约束违例。

如果某个约束有任何问题, 则此语句失败, 并识别出导致错误的约束。如果在违反约束的情况下提交, 则事务将回滚, 并收到一条错误消息。

父主题: [延迟约束检查](#)

### 12.5.5 查看约束信息

KingbaseES 提供了一组视图 (兼容 oracle), 使您能够看到表上的约束定义, 并识别在约束中指定的列。

- DBA\_CONSTRAINTS, DBA 视图描述了数据库中所有的约束定义
- ALL\_CONSTRAINTS, ALL 视图描述了当前用户可访问的约束定义
- USER\_CONSTRAINTS, USER 视图描述了当前用户拥有的定义
- DBA\_CONS\_COLUMNS, DBA 视图描述了约束中指定数据库中所有列
- ALL\_CONS\_COLUMNS, ALL 视图描述了约束中当前用户可访问的列
- USER\_CONS\_COLUMNS, USER 视图只描述了约束中指定的用户当前拥有的列

另请参阅:

《KingbaseES 数据库参考手册》中 KingbaseES 兼容 Oracle 视图章节获取有关上述视图详细描述

父主题: [管理完整性约束](#)

## 12.6 重命名模式对象

重命名对象多种方法。要重命名对象, 它必须在您的模式中。您可以使用以下任一种方式重命名架构对象:

- 删除并重新创建对象
- 可使用 ALTER...RENAME 语句重命名对象 (用于索引和触发器)

如果删除并重新创建对象, 则将丢失授予该对象的所有权限。在重新创建对象时, 必须重新授予权限。

在重命名模式对象之前, 请注意以下影响:

- 所有依赖于重命名对象的视图和 PL/SQL 程序单元都将失效, 在下次使用之前必须重新编译。
- 重命名对象的所有同义词在使用时都返回错误。

另请参阅:

《KingbaseES SQL 语言参考手册》获取关于 RENAME 语句的语法信息

父主题: 模式对象的管理

## 12.7 管理对象名称解析

在 SQL 语句中引用的对象名称可以由多个部分组成, 由点号分隔。KingbaseES 数据库执行特定的操作来解析对象名称。下面描述的是 KingbaseES 数据库管理系统怎样解析对象名字。

- 如果只有一个名字, 而没有点号。则在当前模式下寻找是否存在相同名字的对象, 如果找到, 则返回; 否则报错;
- 如果有点号, 首先检测对象名的第一部分, 如在 scott.emp 中, scott 就是第一部分。
  - 1) 寻找哪一个模式的名称和第一部分相同, 如果找到, 以此模式进行步骤 2); 否则, 以当前模式进行步骤 2);
  - 2) 在模式中需找是否有与对象名第二部分同名的对象, 如果找到且待解析对象名只有两个部分, 则返回; 如果找到但待解析对象名多于两个部分, 则转步骤 3); 否则报错;
  - 3) 在模式中查找包含在前一个对象中并且和待解析对象名当前解析部分相同名字的对象, 如果找到, 循环步骤 3), 直至所有的部分检测结束后再返回; 如果没有找到, 则直接报错。如 scott.emp.deptno.workers, 在 scott 模式下中的 emp 对象中寻找 deptno 的对象, 然后在 deptno 对象中寻找 workers 对象。

另请参阅:

《KingbaseES PL/SQL 过程语言参考手册》获取关于” PL/SQL 名称解析” 详细信息

父主题: 模式对象的管理

## 12.8 切换到其他模式

可以使用 SQL 语句 ALTER SESSION 切换到不同的模式。下面语句将当前会话模式设置为指定的模式名称。

```
ALTER SESSION SET SEARCH_PATH = <schema name>
```

在后续的 SQL 语句中, 如果省略了模式名, 数据库将使用该模式名作为模式限定符。

父主题: 模式对象的管理



## 12.9 显示模式对象的信息

KingbaseES 提供的 PL/SQL 包 DBMS\_METADATA.GET\_DDL 允许您获取关于模式对象的元数据 (以用于创建对象的 DDL 的形式)。DBMS\_METADATA 包是获取模式对象完整定义的强大工具。它使您能够在一次传递中获得对象的所有属性。该对象被描述为 DDL, 可用于 (重新) 创建它。在下面的语句中, GET\_DDL 函数用于获取当前模式中所有表的 DDL。

```
SELECT DBMS_METADATA.GET_DDL('TABLE',u.table_name)
       FROM USER_ALL_TABLES u
       WHERE u.nested='NO'
       AND (u.iot_type is null or u.iot_type='IOT');
```

另请参阅:

《KingbaseES 插件参考手册》获取关于” dbms\_metadata ” 详细信息

父主题: 模式对象的管理

---

# 第 13 章 表的管理

---

表的管理包括创建表、加载表、修改表和删除表等操作，以及临时表，分区表等的管理。

- 关于表

表是数据库中数据存储的基本单元，数据存储在表的行和列中。

- 表管理指南

遵循这些准则可以使表的管理更加容易，并且可以提高创建表、加载表更新以及查询表的性能。

- 创建表

使用 SQL 语句 CREATE TABLE 创建表。

- 加载表

有多种将数据加载到表中的技术。

- 收集数据库的统计信息

sys\_stat\_statement 工具允许您生成数据库的统计信息。您可以使用此工具来收集的数据库性能统计信息。

- 修改表

使用 ALTER TABLE 语句修改表。要修改一个表，表必须包含在您的模式中，或者您必须具有该表的 ALTER 对象权限或 ALTER ANY TABLE 系统权限。

- 删除表

可以使用 DROP TABLE 语句删除不再需要的表。

- 使用 FLASHBACK 删除和恢复表

删除表时，数据库不会立即删除与该表关联的空间。数据库重命名表，并将其和任何关联的对象放在回收站中，万一表被错误地删除，可以在稍后恢复。这个特性被称为 FLASHBACK DROP, FLASHBACK TABLE 语句则用于恢复表。

- 管理分区表

分区表可以将数据分割为更小、更易于管理的小单位，称为分区，甚至是更小的子分区。每个分区可以有单独的物理属性，例如物理存储设置和表空间，能有效提升大型表的访问效率。此外，可以单独管理每个分区，可以简化和减少备份和管理所需的时间。

- 表的数据视图

您可以查询一组数据字典视图，以了解有关表的信息。

## 13.1 关于表

表是数据库中数据存储的基本单位，数据存储在行和列中。表中列的个数和顺序是固定的。在创建表时，要对表中的所有列进行说明，要指明它的名称、数据类型、宽度或精度等。已经说明的列也可以使用 SQL 语句进行修改。

在逻辑结构上，一个表一定是位于某个表空间上，当创建表时，同时将创建一个表段，用于存放表中的数据。在物理结构上，表的数据都放在数据块中，数据块中存放的就是一行的数据结构。同时，一般而言，表的各行数据是以写入的先后顺序存放，而一行的各列一般按照定义表时的指定顺序存放。

列的数据类型限制了该列取值，以及说明了该列进行计算时，存储在该列上的数据的语义。例如，一个列声明为 INTEGER 类型，则其中就不能存放字符串数据，该列的数据可以进行数学运算。同样，一个列声明为字符串类型，则其中可以存放任何数据，但该列的数据不可以进行数学运算。

您可以为表的每一列指定限制规则。这些限制规则称为完整性约束。一个典型例子是 NOT NULL 完整性约束。此约束强制列在每一行中包含一个值。您可以调用透明数据加密来在存储数据之前对数据进行加密。如果用户试图绕过数据库访问控制机制，直接使用操作系统工具查看数据文件，加密会阻止这些用户查看敏感数据。

另请参阅:

- 《KingbaseES SQL 语言参考手册》获取关于数据类型的描述信息
- 《KingbaseES 数据库安全指南》获取关于透明数据加密相关信息

父主题: 表的管理

## 13.2 表管理指南

遵循这些准则可以使表的管理更加容易，并且可以提高创建表、加载表更新以及查询表的性能。

- 创建表之前的设计准则

通常，应用程序开发人员负责设计应用程序的元素，包括表。数据库管理员负责建立用于保存应用程序表的底层表空间的属性。

- 指定表的存储位置

建议在 CREATE TABLE 语句中指定 TABLESPACE 子句，以标识将要存储新表的表空间。对于分区表，您可以选择标识用于存储每个分区的表空间。

- 在创建表时考虑使用 UNLOGGED

要最有效地创建表，可以使用 CREATE UNLOGGED TABLE...AS 语句。如果指定了 UNLOGGED，则创建的表将不做日志。

- 评估表格大小和相应的规划

在创建表之前估计表的大小。最好将此作为数据库规划的一部分。了解数据库表的大小和用途是数据库规划的一个重要部分。

父主题：表的管理

## 13.2.1 创建表之前的设计准则

通常，应用程序开发人员负责设计应用程序的元素，包括表。数据库管理员负责建立用于保存应用程序表的底层表空间的属性。DBA 或应用程序开发人员，或者两者共同工作，可以负责实际创建表，这取决于站点的实践。与应用程序开发人员合作，在设计表时考虑以下准则：

- 对表、列、索引和聚簇使用描述性名称。
- 使用 COMMENT 命令记录每个表及其列的含义。
- 标准化每个表。
- 为每个列选择适当的数据类型。
- 考虑向某些表添加一个或多个虚拟列是否使您的应用程序从中受益。
- 定义允许最后为空的列，以节省存储空间。
- 聚簇表在任何时候都是合适的，可以实现节省存储空间并优化 SQL 语句的性能。

在创建表之前，还应该确定是否使用完整性约束。可以在表的列上定义完整性约束，将自动执行数据库的业务规则。

父主题：表管理指南

## 13.2.2 指定表的存储位置

建议在使用 CREATE TABLE 语句时指定 TABLESPACE 子句，可对表指定存储的表空间。对于分区表，您可以选择指定存储每个分区的表空间。如果没有指定，则该表将创建在用户的默认表空间中。在创建表时，通过指定合适的表空间，有以下优点：

- 提高数据库系统的性能，因为不同的数据库表可能对应不同的数据文件，可减少对相同文件的竞争；
- 减少数据库管理的时间，数据库表分布在不同的表空间中，即使一个表空间损坏，也不影响其他表空间上数据库表的正常访问。

父主题：表管理指南

## 13.2.3 在创建表时考虑使用 UNLOGGED

要最有效地创建表，可以使用 CREATE UNLOGGED TABLE...AS 语句。如果指定了 UNLOGGED，则创建的表将不做日志。

使用 UNLOGGED 子句有以下好处:

- 省去重做日志文件，节省了一定的空间。
- 创建表的时间会减少。
- 并行创建大型表性能得到提升。

父主题：[表管理指南](#)

### 13.2.4 评估表格大小和相应的规划

创建表之前，先进行表大小的评估，并且建议将该评估结果作为数据库规划的一部分。了解数据库表的大小和用途是数据库规划的一个重要部分。

您可以连同索引、撤消空间和重做日志文件的大小一起评估，综合评估表的空间，来确保数据库所需的磁盘空间量。根据估计结果，您就可以购买合适容量的磁盘硬件。

父主题：[表管理指南](#)

## 13.3 创建表

要在所属模式中创建新表，需要有 CREATE TABLE 数据库权限；而要在其他用户的模式中创建新表，则需要有 CREATE ANY TABLE 数据库权限。

创建表时，如果给定了一个模式名（例如 CREATE TABLE myschema.mytable...），那么该表被创建在指定的模式中，否则它被创建在当前模式中。

- [例子: 创建表](#)

若干创建表的例子。

- [创建临时表](#)

临时表在需要缓冲结果集的应用程序中很有用，因为它通过运行多个 DML 操作构建的。您可以创建全局临时表或者局部临时表。

父主题：[表的管理](#)

### 13.3.1 例子: 创建表

若干创建表的例子。

创建 distributors 表，并为 name 列定义一个唯一约束：

```
CREATE TABLE distributors (
  did      integer,
```

```

name    varchar(40) UNIQUE
);

```

执行下列语句后，在 hr 模式中创建了一个名为 emp 的表，并将其存储在 tbs0 表空间中，并定义了主键约束，其中包含定义在不同列上的一个主键：

```

CREATE TABLE hr.emp (
  code      char(12) NOT NULL,
  title     varchar(40) NOT NULL,
  did       integer,
  date_prod date,
  kind      varchar(10),
  len       interval hour to minute,
  CONSTRAINT code_title PRIMARY KEY(code,title)
)TABLESPACE tbs0;

```

创建 distributors 表，指定表和它的唯一索引指定 70% 的填充因子：

```

CREATE TABLE distributors (
  did      integer,
  name     varchar(40),
  UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);

```

创建一个范围分区表：

```

CREATE TABLE measurement (
  logdate      date not null,
  peaktemp     int,
  unitsales    int
) PARTITION BY RANGE (logdate);

```

另请参阅：

《KingbaseES SQL 语言参考手册》获取关于 CREATE TABLE 语句更多创建例子

父主题：[创建表](#)

## 13.3.2 创建临时表

临时表在应用程序需要缓冲结果集时很有用，因为它是通过运行多个 DML 操作构建的。您可以创建全局临时表或者局部临时表。

- [临时表概述](#)

临时表保存的数据仅在事务或会话期间存在。

- 创建临时表的注意事项

在创建临时表时，一些应该关注的注意事项。

- 临时表事务控制

临时表在一个事务块结束时的行为由 ON COMMIT 控制。

- 创建全局临时表

全局临时表是存储在磁盘上的永久数据库对象，对连接到数据库的所有会话都可见。

- 创建局部临时表

局部临时表是在事务或会话结束时删除的临时数据库对象，每个临时表只对创建它的会话可见。

另请参阅：

《KingbaseES SQL 语言参考手册》中 CREATE TABLE 获取关于创建临时表的详细语法和描述

父主题：[创建表](#)

### 13.3.2.1 临时表概述

临时表保存的数据仅在事务或会话期间存在。临时表中的数据是会话私有的。每个会话只能看到和修改自己的数据。简单来看全局临时表定义是持久的，而表数据是临时的，而本地临时表的表定义和表数据都是临时的，会在会话退出后被删除。

本地临时表和全局临时表具体有如下四点区别：

- 本地临时表在临时模式下，用户不可以指定；但是全局临时表创建在用户指定模式下；
- 本地临时表对象本身并不是一直存在，在会话退出后，本地临时表会被删除；全局临时表创建之后，一直存在，除非显示去删除它。
- 删除本地临时表，不受其他会话影响；但是删除全局临时表时，所有会话都不能持有全部临时表的数据。
- 全局临时表不支持外键约束也不支持其他表引用全局临时表作为外键约束，而本地临时表没有这个要求。

父主题：[创建临时表](#)

### 13.3.2.2 创建临时表的注意事项

在创建临时表时，一些应该关注的注意事项。

只有当当前没有会话绑定到一个现有临时表时，才允许在这个临时表上执行 DDL 操作 (TRUNCATE 除外)。如果事务回滚，则输入的数据将丢失，但表定义仍然存在。特定于事务的临时表一次只允许一个事务。如果在一个事务作用域中有多个自治事务，则每个自治事务只能在前一个事务提交之后才能使用该表。

由于临时表中的数据根据定义是临时的，所以当系统发生故障时，临时表数据的备份和恢复功能是不可用的。为了应对这种故障，应该开发时需要其他方法来保存临时表数据。

父主题：[创建临时表](#)

### 13.3.2.3 临时表事务控制

临时表在一个事务块结束时的行为由 ON COMMIT 控制。

三种选项具体是：

- PRESERVE ROWS

在事务结束时不采取特殊的动作。是本地临时表采用的默认行为

- DELETE ROWS

在每一个事务块结束时将删除临时表中的所有行。实质上，在每一次提交时会完成一次自动的 TRUNCATE。在分区表上使用时，不会将其级联到其分区。这是全局临时表的默认行为。

- DROP

在当前事务块结束时将删除临时表。在分区表上使用时，此操作将删除其分区，在具有继承子级的表上使用时，将删除依赖子级。仅本地临时表支持在当前事务块结束时将删除临时表。为兼容 Oracle，全局临时表不支持 ON COMMIT DROP 选项。

父主题：[创建临时表](#)

### 13.3.2.4 创建全局临时表

全局临时表是存储在磁盘上的永久数据库对象，对连接到数据库的所有会话都可见。

全局临时表的定义对所有会话都是可见的，但是全局临时表中的数据只对向表中插入数据的会话可见。您使用 CREATE GLOBAL TEMPORARY TABLE 语句创建一个全局临时表。

下列示例演示了如何创建全局临时表。这条语句创建了一个特定于事务的全局临时表：

```
CREATE GLOBAL TEMPORARY TABLE admin_work_area_trans
  (startdate DATE,
   enddate DATE,
   class CHAR(20))
ON COMMIT DELETE ROWS;
```

这条语句创建了一个特定于会话的全局临时表：

```
CREATE GLOBAL TEMPORARY TABLE admin_work_area_session
  (startdate DATE,
   enddate DATE,
   class CHAR(20))
ON COMMIT PRESERVE ROWS;
```

父主题：[创建临时表](#)



### 13.3.2.5 创建局部临时表

局部临时表的元数据和内容仅在创建该临时表的会话中可见。局部临时表在以下情况下很有用:

- 当应用程序将临时数据存储在临时表中时，这些临时表只填充一次，读取几次，然后在事务或会话结束时删除
- 当一个会话被无限期地维护，并且必须为不同的事务创建不同的临时表时
- 创建临时表时，不能启动新事务或提交现有事务
- 当同一个用户的不同会话必须对一个临时表使用相同的名称时

使用 CREATE LOCAL TEMPORARY TABLE 语句创建一个局部临时表。LOCAL 可以不指定，因为默认是局部临时表。

下列示例演示如何创建局部临时表。这条语句创建了一个特定于事务的局部临时表:

```
CREATE LOCAL TEMPORARY TABLE sales_ptt_transaction
  (time_id      DATE,
   amount_sold NUMBER(10,2))
ON COMMIT DELETE ROWS;
```

这条语句创建了一个局部临时表，它是特定于会话的:

```
CREATE LOCAL TEMPORARY TABLE sales_ptt_session
  (time_id      DATE,
   amount_sold NUMBER(10,2))
ON COMMIT PRESERVE ROWS;
```

默认情况下，局部临时表中的行存储在创建它的用户的默认临时表空间中。但是，在创建临时表的过程中，可以通过 CREATE LOCAL TEMPORARY TABLE 语句的 TABLESPACE 子句将一个局部临时表分配给另一个临时表空间。

父主题: [创建临时表](#)

## 13.4 加载表

有几种方法可以将数据插入或初始加载到表中。

最常用的是以下几种:

- [sys\\_bulkload 工具快速加载数据](#)

sys\_bulkload 是 KingbaseES 提供的快速加载数据的命令行工具，该工具能够把一定格式的文本数据简单、快速的加载到 KingbaseES 数据库中。

- [使用 CREATE TABLE AS 语句](#)

使用此 SQL 语句，您可以创建一个表，并使用从另一个现有表 (包括外部表) 选择的数据填充该表数据。

- 使用 INSERT 语句

INSERT 语句允许您通过指定列值或指定从另一个现有表选择数据的子查询向表添加行。

- 使用 MERGE 语句

通过从另一个现有表中选择行，可以使用 MERGE 语句向表中插入行或更新行。如果新数据中的一行对应于表中已经存在的项，则执行 UPDATE，否则执行 INSERT。

- 使用 COPY 语句

COPY 语句允许您将 KingbaseES 表和标准文件系统文件之间相互转移数据。可以实现将表的内容复制到文件，也可以将文件复制数据到表，甚至复制一个 SELECT 查询结果。

父主题：表的管理

### 13.4.1 sys\_bulkload 工具快速加载数据

sys\_bulkload 是 KingbaseES 提供的快速加载数据的命令行工具，该工具能够把一定格式的文本数据简单、快速的加载到 KingbaseES 数据库中。

创建导入数据表

```
create TABLE test(id int primary key, info text, crt_time timestamp);
```

数据文件示例：将下列数据以 test.csv 为文件名保存到 KingbaseES 服务器所在目录

```
1,29b35ff06c949e7e442c929e1df86396,2017-10-08 10:52:47.746062
2,06fde814525395de5ab85f6d92b04e87,2017-10-08 10:52:47.746573
3,c93f02e8677c9cd7c906c6ad5dbd450e,2017-10-08 10:52:47.746627
4,6541700070ae3d051f965fcef43baf45,2017-10-08 10:52:47.746835
5,3d7e7246016acaa842526b6614d0edf5,2017-10-08 10:52:47.746869
6,1d1ae5a03ef0bad3bc14cd5449ba0985,2017-10-08 10:52:47.746894
7,7745c57c54b97656bec80a502ec13ec7,2017-10-08 10:52:47.746918
8,3c377131f6ef82c3284dc77a3b4ffdf7,2017-10-08 10:52:47.746942
9,5ef98d40aeeadf65eb1f0d7fd86ed585,2017-10-08 10:52:47.746968
10,312c0a0188da9e34fe45aa19d0d07427,2017-10-08 10:52:47.746993
```

导入数据

1. 以 BUFFERED 方式导入 TEXT 文件配置文件示例（以 test.ctl 为名保存到服务器所在目录，也可自行指定其他目录。）

```
TABLE = test
INPUT = /home/kingbase/test.csv
TYPE = CSV
SKIP = 2
LIMIT = 5
```

```
WRITER = BUFFERED
PROCESSOR_COUNT = 3
```

## 2. 使用配置文件导入命令示例

```
./sys_bulkload -h localhost -p 54321 -d TEST /home/kingbase/test.ctl -U SYSTEM -W 123456
```

## 3. 使用配置文件导入结果

导入成功提示信息:

```
NOTICE: BULK LOAD START
NOTICE: BULK LOAD END
    2 Rows skipped.
    5 Rows successfully loaded.
    0 Rows not loaded due to parse errors.
    0 Rows not loaded due to duplicate errors.
    0 Rows replaced with new rows.
log path: /home/test/test.log
parse error path: /home/test/test.bad
duplicate error path: /home/test/test.dupbad
ctrl file path: /home/test/test.ctl
data file path: /home/test/test.csv
Run began on 2021-03-27 22:53:26.690476+08
Run ended on 2021-03-27 22:53:27.496344+08
```

导入结果查询（配置指定跳过前 2 行，限制导入条数 5 条）:

```
test=# select * from test;
 id |          info          |          crt_time
-----+-----+-----
  3 | c93f02e8677c9cd7c906c6ad5dbd450e | 2017-10-08 10:52:47.746627
  4 | 6541700070ae3d051f965fcef43baf45 | 2017-10-08 10:52:47.746835
  5 | 3d7e7246016acaa842526b6614d0edf5 | 2017-10-08 10:52:47.746869
  6 | 1d1ae5a03ef0bad3bc14cd5449ba0985 | 2017-10-08 10:52:47.746894
  7 | 7745c57c54b97656bec80a502ec13ec7 | 2017-10-08 10:52:47.746918
(5 行记录)
```

另请参阅:

《KingbaseES 客户端应用参考手册》获取关于 sys\_bulkload 详细选项描述和示例

父主题: [加载表](#)

## 13.4.2 使用 CREATE TABLE AS 语句

CREATE TABLE AS 创建一个表，并且用由一个 SELECT 命令计算出来的数据填充该表。该表的列具有和 SELECT 的输出列相关的名称和数据类型（不过可以通过给出一个显式的新列名列表来覆盖这些列名）。

CREATE TABLE AS 和创建一个视图有些相似，但是实际上非常不同：它会创建一个新表并且只计算该查询一次用来初始填充新表。这个新表将不会跟踪该查询源表的后续变化。相反，一个视图只要被查询，它的定义 SELECT 语句就会被重新计算。

创建一个新表 films\_new，它只由表“films”中最近一段时间的项组成：

```
CREATE TABLE films_new AS
SELECT * FROM films WHERE date_prod >= '2022-01-01';
```

要完全地复制一个表，也可以使用 TABLE 命令的简短形式：

```
CREATE TABLE films_copy AS
TABLE films;
```

用一个预备语句创建一个新的临时表 films\_new，它仅由表 films 中最近的项组成。新表将在提交时被删除：

```
PREPARE recentfilms(date) AS
SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_new ON COMMIT DROP AS
EXECUTE recentfilms('2022-01-01');
```

另请参阅：

《KingbaseES SQL 语言参考手册》获取关于 CREATE TABLE AS 详细语法描述和示例

父主题：[加载表](#)

## 13.4.3 使用 INSERT 语句

INSERT 语句允许您通过指定列值或指定从另一个现有表选择数据的子查询向表添加行。

例如，当在表之间拷贝值时，这个 OVERRIDING USER VALUE 子句就能派上用场，下面这个例子将从 tbl1 中拷贝所有在 tbl2 中不是标识列的列，而 tbl2 中标识列的值将由与 tbl2 关联的序列产生。

```
INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1;
```

这个例子从表 tmp\_films 中获得一些行插入到表“films”中，两个表具有相同的列布局：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2022-05-07';
```

另请参阅：

《KingbaseES SQL 语言参考手册》获取关于 INSERT 语句详细语法描述和示例

父主题: [加载表](#)

## 13.4.4 使用 MERGE 语句

使用 MERGE 语法可合并 UPDATE 和 INSERT 语句功能。

通过 MERGE 语句, 根据一张表 (或视图) 的连接条件对另外一张表 (或视图) 进行查询, 连接条件匹配上的执行 UPDATE (可能含有 DELETE), 无法匹配的执行 INSERT。其中数据表包括: 普通表、分区表。

这个例子创建了两个表 test1 和 test2, 并且进行条件匹配, 将结果合并到 test2 中:

```
create table test1(a int, b int);
insert into test1 values(1, 1);
insert into test1 values(2, 2);
insert into test1 values(3, 3);

create table test2(x int, y int);
insert into test2 values(1, 1);
insert into test2 values(3, 3);
insert into test2 values(5, 5);

merge into test2 using test1
on (test1.a = test2.x)
when matched then
update set y = y * -1 where test1.a > 1
when not matched then
insert values(test1.a, test1.b);
```

执行后 test2 结果:

```
x | y
---+---
1 | 1
5 | 5
2 | 2
3 | -3
(4 rows)
```

执行后 test1 结果:

```
a | b
---+---
1 | 1
2 | 2
3 | 3
(3 rows)
```

父主题: [加载表](#)

## 13.4.5 使用 COPY 语句

COPY 语句允许您将 KingbaseES 表和标准文件系统文件之间相互转移数据。可以实现将表的内容复制到文件，也可以将文件复制数据到表，甚至复制一个 SELECT 查询结果。

- COPY TO 把一个表的内容复制到一个文件;
- COPY FROM 则从一个文件复制数据到一个表（把数据追加到表中原有数据）;
- COPY TO 也能复制一个 SELECT 查询的结果。

COPY TO 只能被用于纯粹的表，不能用于视图。不过您可以写 COPY (SELECT \* FROM viewname) TO... 来拷贝一个视图的当前内容。

COPY FROM 可以被用于普通表、外部表、分区表或者具有 INSTEAD OF INSERT 触发器的视图。

COPY 只处理提到的表，它不会从子表复制数据或者复制数据到子表中。例如 COPY table TO 会显示与 SELECT \* FROM ONLY table 相同的数据。而 COPY (SELECT \* FROM table) TO ... 可以被用来转储一个继承层次中的所有数据。

1. 下面的例子使用竖线 (|) 作为域定界符把 country 表复制到客户端:

```
COPY country TO STDOUT (DELIMITER '|');
```

2. 从文件 country\_data 中复制数据到 country 表中:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

3. 只把名称以'A'开头的国家复制到文件 a\_list\_countries.copy 中:

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO  
'/usr1/proj/bray/sql/a_list_countries.copy';
```

另请参阅:

《KingbaseES SQL 语言参考手册》获取关于 COPY 语句详细语法和示例信息

父主题: [加载表](#)

## 13.5 收集数据库的统计信息

数据库内部提供了多种性能视图和工具，将允许您生成数据库的统计信息。您可以使用诸如 sys\_stat\_statements 工具来收集相关的性能统计信息。数据库运行时的状态对分析某些性能问题的分析至关重要，我们可用通过对系统状态统计信息、日志信息的分析，来监控整个数据库的运行状态，并能从其中发现一些性能相关的问题。

跟数据库性能相关的信息和工具主要包括：

整体分析	sys_kwr 报告工具（基本覆盖所有优化点分析） sys_ksh 工具
CPU 优化点分析	耗时语句统计： sys_stat_statements kbbadger
会话/负载/等待事件分析	sys_stat_activity 视图 (实时) sys_ksh 报告工具 (时间区间)
锁优化点分析	sys_locks 视图
IO 优化点分析	sys_statio_user_tables 视图 sys_statio_user_indexes 视图
应用特点分析	sys_stat_user_tables 视图 sys_stat_user_indexes 视图 sys_stat_database 视图

另请参阅：

《KingbaseES 数据库性能调优指南》手册的性能诊断部分获取关于数据库资源分析详细信息

父主题：表的管理

## 13.6 修改表

和修改表中所包含的数据不同，有时当创建表并且填充数据后，有可能需要修改表的定义或者说结构，这一类操作可以使用 alter TABLE 语句修改表。

要修改一个表，表必须包含在您的模式中，或者您必须具有该表的 ALTER 对象权限或 ALTER ANY TABLE 系统权限。KingbaseES 提供了一族命令来对已有的表进行修改。通过更改表，用户可以对数据库中的表作如下修改：

- 增加列

为表新增加一列。

- 移除列

移除表中的现有列。

- 增加约束

给表增加约束条件。

- 移除约束

移除表中的现有约束。

- 修改列的默认值

修改一个列的默认值。

- 修改列的数据类型

将一个列的数据类型转换为另一种数据类型。

- 重命名列

重命名一个现有列。

- 重命名表

修改表的名字。

另请参阅:

《KingbaseES SQL 语言参考手册》的 ALTER TABLE 语句获取更详细的信息。

父主题: [表的管理](#)

## 13.6.1 增加列

要增加一个列，可以使用这样的命令：

```
ALTER TABLE products ADD COLUMN description text;
```

新列将被默认值所填充（如果没有指定 DEFAULT 子句，则会填充空值）。

**提示：** 在 KingbaseES，添加具有常量默认值的列不再意味着在执行 ALTER TABLE 语句时需要更新表的每一行。相反，默认值将在下一次访问该行时返回，并在重写表时应用，这使得 ALTER TABLE 非常快，即使在大型表上也是如此。

但是，如果默认值是 volatile (e.g. clock\_timestamp()), 每一行都需要在执行 ALTER TABLE 时计算出的值来更新。以避免潜在的冗长的更新操作，特别是如果你打算填列主要是默认的值，它可能是更可取的添加没有默认的列，使用 UPDATE 插入正确的值，然后添加任何所需的默认如下所述。

也可以同时为列定义约束，语法：

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

事实上 CREATE TABLE 中关于一列的描述都可以应用在这里。记住不管怎样，默认值必须满足给定的约束，否则 ADD 将会失败。也可以先将新列正确地填充好，然后再增加约束。

父主题: [修改表](#)



## 13.6.2 移除列

为了移除一个列，使用如下的命令：

```
ALTER TABLE products DROP COLUMN description;
```

列中的数据将会消失，涉及到该列的表约束也会被移除。然而，如果该列被另一个表的外键所引用，KingbaseES 不会移除该约束。我们可以通过增加 CASCADE 来授权移除任何依赖于被删除列的所有东西：

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

父主题：[修改表](#)

## 13.6.3 增加约束

在现有表基础上新增加约束，可以使用表约束的语法，例如：

```
ALTER TABLE products ADD CHECK (name <> '');  
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);  
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

要增加一个不能写成表约束的非空约束，可使用语法：

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

该约束会立即被检查，所以表中的数据必须在约束被增加之前就已经符合约束。

父主题：[修改表](#)

## 13.6.4 移除约束

为了移除一个约束首先需要知道它的名称。如果在创建时已经给它指定了名称，那么事情就变得很容易。否则约束的名称由系统生成，我们必须先找出这个名称。ksql 的命令 d 表名将会对此有所帮助，其他接口也会提供方法来查看表的细节。因此命令是：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

（如果处理的是自动生成的约束名称，如 \$2，别忘了用双引号使它变成一个合法的标识符。）

和移除一个列相似，如果需要移除一个被某些对象依赖的约束，也需要加上 CASCADE。一个典型的例子是一个外键约束依赖于被引用列上的一个唯一主键约束。

这对除了非空约束之外的所有约束类型都一样有效。为了移除一个非空约束可以用：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

（回忆一下，非空约束是没有名称的，所以不能用第一种方式。）

父主题：修改表

### 13.6.5 修改列的默认值

要为一个列设置一个新默认值，使用命令：

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

注意这不会影响任何表中已经存在的行，它只是为未来的 INSERT 命令改变了默认值。要移除任何默认值，使用：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

这等同于将默认值设置为空值。相应的，试图删除一个未被定义的默认值并不会引发错误，因为默认值已经被隐式地设置为空值。

父主题：修改表

### 13.6.6 修改列的数据类型

为了将一个列转换为一种不同的数据类型，使用如下命令：

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

只有当列中的每一个项都能通过一个隐式造型转换为新的类型时该操作才能成功。如果需要一种更复杂的转换，应该加上一个 USING 子句来指定应该如何把旧值转换为新值。

KingbaseES 将尝试把列的默认值转换为新类型，其他涉及到该列的任何约束也是一样。但是这些转换可能失败或者产生奇特的结果。因此最好在修改类型之前先删除该列上所有的约束，然后在修改完类型后重新加上相应修改过的约束。

父主题：修改表

### 13.6.7 重命名列

要重命名一个现有表的列，使用如下命令：

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

父主题：修改表

## 13.6.8 重命名表

要重命名一个现有表，使用如下命令：

```
ALTER TABLE products_table RENAME TO products_items;
```

products\_table 表将被重命名为 products\_items。

父主题：[修改表](#)

## 13.7 删除表

当一个表不再使用时，可以将其删除。删除表时，将产生以下结果：

- 表的结构信息从数据字典中删除，表中的数据不可访问；
- 表上的所有索引和触发器被一起清除；
- 所有建立在该表上的同义词、视图和存储过程变为无效；

一般情况下，普通用户只能删除自己模式下的表。若要删除其他模式下的表，则必须具有 DROP ANY TABLE 数据库权限。

以下语句可删除 emp 表：

```
DROP TABLE emp;
```

如果要删除的表被其他表引用，即其他表的外键引用了表的任何主键或唯一键，则需要在 DROP TABLE 语句中包含 CASCADE 选项，如：

```
DROP TABLE emp CASCADE
```

当您删除一个表时，数据库通常不会立即释放与该表关联的空间。相反，数据库会重命名表并将其放入回收站，如果您发现错误地删除了该表，则可以使用 FLASHBACK TABLE 语句在回收站中恢复该表。如果您希望在发出 DROP TABLE 语句时立即释放与表相关的空间，请包含如下语句所示的 PURGE 子句：

```
DROP TABLE hr.emp PURGE;
```

父主题：[表的管理](#)

## 13.8 使用 FLASHBACK 删除和恢复表

删除表时，数据库不会立即删除与该表关联的空间。数据库重命名表，并将其和任何关联的对象放在回收站中，万一表被错误地删除，可以在稍后恢复。这个特性被称为 FLASHBACK DROP, FLASHBACK TABLE 语句用于恢复表。

- 什么是回收站

回收站实际上是一个数据字典表，包含关于被删除对象的信息。被删除的表和任何关联的对象 (如索引、约束、嵌套表等) 没有被删除，仍然会占用空间。

- 启用和禁用回收站

当回收站被启用时，被删除的表及其相关对象将被放入回收站。当回收站被禁用时，被删除的表及其相关对象不会被放入回收站；它们已被删除，您必须使用其他方法来恢复它们 (例如从备份中恢复)

- 从回收站恢复表

使用 `FLASHBACK TABLE ... TO BEFORE DROP` 语句从回收站中恢复表。

另请参阅:

《KingbaseES SQL 语言参考手册》中获取关于 `FLASHBACK DROP TABLE` 语句更多详细信息

父主题: [表的管理](#)

## 13.8.1 什么是回收站

回收站实际上是一个数据字典表，包含关于被删除对象的信息。被删除的表和任何关联的对象 (如索引、约束、嵌套表等) 没有被删除，仍然会占用空间。这些已删除的对象会继续占用用户空间，直到从回收站中清除它们。

用户可以使用以下语句查看回收站中的对象:

```
SELECT * FROM RECYCLEBIN;
```

只有 SQL 语句，`DROP TABLE` 语句会将对象放入回收站。它添加表及其关联对象，以便它们可以作为一个组数据一起进行恢复。除了表本身，添加到回收站的关联对象可以包括以下类型的对象:

- 嵌套表
- LOB 数据库字段
- 索引
- 约束 (不包括外键约束)
- 触发器
- 聚簇

`DROP TABLE` 若是开启了 GUC 参数: `kdb_flashback.db_recyclebin=on`，且未指定 `PURGE` 选项，则删除的表及其关联的索引、规则、触发器、约束等将被放到 `SYS_RECYCLEBIN` 系统表中。在需要时可以将回收站中的表闪回到删除之前的状态。

父主题: [使用 FLASHBACK 删除和恢复表](#)

## 13.8.2 启用和禁用回收站

回收站功能默认关闭，可以通过下列命令查看是否开启回收站功能：

```
show kdb_flashback.db_recyclebin;
```

开启回收站功能，在登录数据库后执行以下命令即可

```
alter system set kdb_flashback.db_recyclebin = on;
```

父主题：[使用 FLASHBACK 删除和恢复表](#)

## 13.8.3 从回收站恢复表

开启了回收站参数后，如果不允许一个删除的表进回收站，需要在 DROP 语句后面加上 “PUGRE“X 选项。flashback 支持将一个已经删除到回收站中的表及相关对象闪回到删除之前状态。

如果在当前 schema 下已经存在同名的表，则闪回失败，需要使用 rename to 选项指定一个新的表名。成功闪回一个表后，回收站视图 recyclebin 和回收站系统表 sys\_recyclebin 中对象将被清除。

一个 FLASHBACK 命令：

```
FLASHBACK TABLE TABLE_name TO BEFORE DROP;
```

这个命令将在回收站中的 TABLE\_name 闪回到闪回之前的状态：

```
FLASHBACK TABLE TABLE_name TO BEFORE DROP RENAME TO newname;
```

将回收站中的表闪回到一个新的表名：

---

提示：要将删除的表放到回收站需要在 kingbase.conf 配置文件末尾添加 kdb\_flashback.db\_recyclebin=on。

---

父主题：[使用 FLASHBACK 删除和恢复表](#)

## 13.9 管理分区表

分区表可以将数据分割为更小、更易于管理的小单位，称为分区，甚至是更小的子分区。每个分区可以有单独的物理属性，例如物理存储设置和表空间，能有效提升大型表的访问效率。此外，可以单独管理每个分区，可以简化和减少备份和管理所需的时间。

另请参阅：

《KingbaseES 数据库概念》手册获取关于分区表管理的详细描述

父主题：表的管理

## 13.10 表的数据视图

KingbaseES 提供了一组兼容 Oracle 的视图，您可以用来查询一组数据字典视图，获取表的信息。

视图	描述
DBA_TABLES ALL_TABLES USER_TABLES	DBA 视图描述数据库中的所有关系表。ALL 视图描述用户可访问的所有表。USER 视图仅限于用户拥有的表。
DBA_TAB_COLUMNS ALL_TAB_COLUMNS USER_TAB_COLUMNS	这些视图描述当前所在数据库中，所有的表、视图的非隐藏列信息
DBA_TAB_COMMENTS ALL_TAB_COMMENTS USER_TAB_COMMENTS	这些视图显示表和视图的注释。使用 COMMENT 语句输入注释。
DBA_COL_COMMENTS ALL_COL_COMMENTS USER_COL_COMMENTS	这些视图显示表和视图列的注释。使用 COMMENT 语句输入注释。

另请参阅:

《数据库参考手册》获取上述兼容 Oracle 视图的详细列表和信息

父主题：表的管理

---

# 第 14 章 索引的管理

---

索引是一种建立在表上的数据库对象，主要作用就是加快对表的查询速度，合理的使用索引将大大减少磁盘访问次数，从而提升数据库性能。本章节索引的管理主要包括创建、修改、删除和查询索引。

- 关于索引

索引是和表关联的可选结构，可以实现对表更快的 SQL 查询速度。

- 索引的管理指南

可供参考的若干索引管理指南。

- 创建索引

您可以创建多种不同类型的索引。您可以显式地创建索引，也可以创建与约束关联的索引。

- 修改索引

若要更改索引，模式中必须包含该索引。

- 删除索引

使用 DROP INDEX 语句删除索引。

- 索引视图

通过查询索引视图，可以了解索引的相关信息。

## 14.1 关于索引

索引是和表关联的可选结构，可以实现对表更快的 SQL 查询速度。

就像本手册中的文章目录索引可以帮助您更快的查看到关注的章节，KingbaseES 数据库的索引也起到类似的作用，提供更快的访问表数据的方法。索引的新创建和使用无需重写任何查询语句，获取的结果与创建索引前是一样的，但您查询到结果的速度将更快了。

索引是如何加快查询速度的呢？那是因为，索引是建立在某个列或者几个列上（即索引列），创建索引时，数据库服务器将根据索引列进行排序，并将排序的结果存储在索引指定的存储空间中。在执行查询时，数据库服务器首先

在索引中查询，然后再到表中查询，因为索引中的数据进行了排序，因此只需要很少的查询次数就可以找到指定的数据。

KingbaseES 数据库提供了几种功能互补的索引方案:

- b-tree 索引: 默认也是最常用的索引
- Hash 索引: 只能处理简单等值比较
- GiST 索引: 可以实现很多不同索引策略
- SP-GiST 索引: 允许实现众多不同的非平衡的基于磁盘的数据结构, 例如四叉树、k-d 树和 radix 树。
- GIN 索引是“倒排索引”: 它适合于包含多个组成值的数据值
- BRIN 索引(块范围索引的缩写): 存储有关存放在一个表的连续物理块范围上的值摘要信息。
- BITMAP 索引: 以位图形式存储列中特定数据的出现位置信息。
- 全局索引和本地索引: 与分区表和索引相关
- 基于函数的索引: 基于函数或者表达式的索引

索引在逻辑上和物理上都是独立于关联表中的数据。作为独立的结构, 它们固然需要额外的存储空间。您的创建或删除索引操作, 是不会影响基表、数据库应用程序或其他索引。并且当您插入、更新和删除相关表的行时, 数据库将自动维护索引。如果索引被删除, 所有应用程序仍然继续正常工作, 但访问数据可能会因此变慢。

父主题: [索引的管理](#)

## 14.2 索引的管理指南

可供参考的若干索引管理指南。

- [在插入表数据后创建索引](#)

一般情况下, 在插入或加载数据之后, 再为表创建索引会更有效率。

- [适合创建索引的表和列](#)

若干适合表和列创建索引的指导原则。

- [考虑排序索引列来改善性能](#)

CREATE INDEX 语句中的列顺序会影响查询性能。通常, 首先指定最常用的列。

- [限制每个表的索引的数量](#)

一个表可以有任意数量的索引。但是, 索引越多, 修改表时产生的开销就越大, 磁盘占用也越多。

- [删除不再需要的索引](#)

实践来看, 当不再需要索引时, 应及时将其删除。

- [为索引指定表空间](#)



可以在表空间中创建索引。

- 评估索引大小和设置存储参数

在创建索引之前评估索引的占用空间的大小，将有益于磁盘空间规划和管理。

- 在禁用或删除约束之前考虑成本

因为唯一键和主键有关联的索引，所以在考虑是否禁用或删除 UNIQUE 或 PRIMARY KEY 约束时，应该考虑删除和创建索引的成本。

父主题：索引的管理

## 14.2.1 在插入表数据后创建索引

一般情况下，应在插入或加载数据之后，再为表创建索引会更有效率。如果您在加载数据之前创建了一个或多个索引，那么数据库必须在插入每一行数据时更新每个索引，导致插入效率降低。

父主题：索引的管理指南

## 14.2.2 适合创建索引的表和列

若干适合表和列创建索引的指导原则。可以参考下面的准则来决定何时为表创建索引：

- 如果需要经常地检索大表中的少量的行，就为查询键创建索引；
- 为了改善多个表的连接的性能，可为连接列创建索引；
- 主键和唯一键自动具有索引，在外键上很多情况下也创建索引；
- 如果查询语句花费的时间太长，那么就检查表大小。如果它发生了显著变化，则需要评估现有的索引 (如果有的话)。

选取表中的索引列时可以参考以下几点原则，越多符合越适合作为索引列：

- 列中的值相对比较唯一；
- 取值范围大，适合建立索引 (适合普通索引)；
- 列包含许多空值，但是查询通常会选择所有具有值的行。

反之，以下情况不应创建索引：

- 如果该表频繁的进行 DML 操作，不应建立索引，或者建立少量索引；
- 有很多重复值的列，一般不建议作为索引列；
- 太小的表，不用建立索引，也没有必要。

父主题：索引的管理指南

### 14.2.3 考虑排序索引列来改善性能

在 CREATE INDEX 语句中列的排序会影响查询的性能。通常，将最常用的列放在最前面。

如果查询中有多个字段组合定位，则不应为每个字段单独创建索引，而应该创建一个组合索引。当两个或多个字段都是等值查询时，组合索引中各个列的前后关系是无关紧要的。但是如果是非等值查询时，要想有效利用组合索引，则应该按等值字段在前，非等值字段在后的原则创建组合索引，查询时只能利用一个非等值的字段。

父主题：索引的管理指南

### 14.2.4 限制每个表的索引的数量

一个表可以有任意数量的索引。但是，索引越多，修改表数据的开销就越大，占用的磁盘空间也越大。

当插入或删除行时，表上的所有索引也要被更改；更改一个列时，包含该列的所有索引也要被更改。因此，在从表中检索数据的速度和更新表的速度之间需要权衡。典型的：如果一个表主要仅用于读，则索引多就有好处；如果一个表经常被更新，则索引不宜多建。

父主题：索引的管理指南

### 14.2.5 删除不再需要的索引

实践来看，当不再需要索引时，应及时将其删除。一方面能避免修改表数据的额外索引开销，另一方面也能减少磁盘空间占用。

要删除索引，您应该具有删除权限，或者该索引在您的模式中。在以下情况下应考虑将索引删除：

- 索引没有加速查询。比如，表非常小，或者表中可能有很多列，但索引项很少。
- 应用程序中的查询不使用索引，或者索引已无效。
- 重建索引之前，必须删除索引。
- 索引变得过于零散，应考虑重新创建索引，此时可以将其删除

父主题：索引的管理指南

### 14.2.6 为索引指定表空间

可以在表空间中创建索引。

如果表及其索引使用相同的表空间能更方便地对数据库进行管理（如表空间或文件备份）或保证应用的可用性，因为所有有关的数据总是在一起联机。然而，将表及其索引放在不同的表空间（在不同磁盘上）产生的性能比放在相同的表空间更好，因为这样做减少了磁盘竞争。但是将表及其索引放在不同的表空间时，如果一个表上某索引所在的表空间脱机了，则涉及这张表的 SQL 语句可能由于执行计划仍旧需要使用被脱机的索引而不能成功执行

父主题：索引的管理指南

## 14.2.7 评估索引大小和设置存储参数

在创建索引之前评估索引的占用空间的大小，将有益于磁盘空间规划和管理。

您可以使用综合评估的索引大小，结果对表、撤销表空间和重做日志文件的大小评估，以此来确定数据库所需的磁盘空间量，从而购买合适的硬件。

CREATE INDEX 语句创建索引时，可选的 WITH 子句可以为索引设置存储参数，每一种索引方法都有自己的存储参数集合。适当的存储参数可以改善使用该索引应用程序的 I/O 性能。例如，假设在创建索引之前估计索引的最大大小，之后就可以在创建该索引时设置适当的存储参数，就能很少为表的数据段分配簇。并且，所有的该索引的数据都被保存在相对连续的磁盘空间扇区中，这就减少了使用该索引的磁盘 I/O 操作所需的时间。

另请参阅：

请参阅《KingbaseES SQL 语言参考手册》中关于 CREATE INDEX 语句设置存储参数更多详细的描述和信息

父主题：索引的管理指南

## 14.2.8 在禁用或删除约束之前考虑成本

因为唯一键和主键有关联的索引，所以在考虑是否禁用或删除 UNIQUE 或 PRIMARY KEY 约束时，应该考虑删除和创建索引的成本。

如果 UNIQUE 键或 PRIMARY KEY 约束的关联索引非常大，那么您可以通过启用约束来节省时间，而不是删除并重新创建索引。

您还可以选择在删除或禁用 UNIQUE 或 PRIMARY KEY 约束时显式指定想要保留或删除索引。

父主题：索引的管理指南

## 14.3 创建索引

您可以创建多种不同类型的索引。您可以显式地创建索引，也可以创建基于函数或者表达式的索引等。要在用户自己的模式中创建索引，至少要满足如下条件之一：

- 要被索引的表是在自己的模式中；
- 在要被索引的表上有 CREATE INDEX 权限；

本节将描述如何创建不同类型的索引。

- 显式创建索引

可以使用 SQL 语句 CREATE INDEX 显式地创建索引。

- 创建唯一索引

索引可以是唯一的，也可以是非唯一的。

- 创建指定顺序的 B 树索引

一个索引可能还需要将它们以指定的顺序传递。这使得查询中的 ORDER BY 不需要独立的排序步骤。

- 创建 Hash 索引

Hash 索引只能处理简单等值比较。

- 创建基于函数的索引

基于函数的索引时将索引建立在某个函数或者表达式的基础上。

- 创建多列索引

一个索引可以定义在表的多个列上。

- 创建本地和全局索引

在分区表上创建的索引可分为全局索引和本地索引。

另请参阅:

您也可以在《KingbaseES SQL 语言参考手册》中获取关于不同类型索引的详细描述

父主题: [索引的管理](#)

## 14.3.1 显式创建索引

可以使用 SQL 语句 CREATE INDEX 显式地创建索引。

下面的语句为 emp 表的 ename 列创建了一个名为 emp\_ename\_index 的索引，并存放在 users\_space 表空间上:

```
CREATE INDEX emp_ename_index ON emp(ename)
    TABLESPACE users_space;
```

父主题: [创建索引](#)

## 14.3.2 创建唯一索引

索引可以是唯一的，也可以是非唯一的。唯一索引保证表中没有两行在键列中有重复的值。非唯一索引则不会对列值施加此限制。使用 CREATE UNIQUE INDEX 语句创建一个唯一索引。下面的例子会为 dept 表的 ename 列创建唯一索引 dept\_unique\_index:

```
CREATE UNIQUE INDEX dept_unique_index ON dept (dname)
    TABLESPACE users_space;
```

**注意:** 当前，只有 B-tree 能够被声明为唯一。

当一个索引被声明为唯一时，索引中不允许多个表行具有相同的索引值。空值被视为不相同。一个多列唯一索引将会拒绝在所有索引列上具有相同组合值的表行。

KingbaseES 会自动为定义了一个唯一约束或主键的表创建一个唯一索引。该索引包含组成主键或唯一约束的所有列（可能是一个多列索引），它也是用于强制这些约束的机制。

父主题：创建索引

### 14.3.3 创建指定顺序的 B 树索引

一个索引可能还需要将它们以指定的顺序传递。这使得查询中的 ORDER BY 不需要独立的排序步骤。默认情况下，CREATE INDEX 命令创建适合于大部分情况的 B-tree 索引。如下列命令可以用来在表 test 上的 id 列上创建一个索引：

```
CREATE INDEX test_id_index ON test (id);
```

索引的名字 test\_id\_index 可以自由选择，但最好选择一个有相关性、能让用户想起该索引用途的名字。在 KingbaseES 当前支持的索引类型中，只有 B-tree 可以产生排序后的输出，其他索引类型会把行以一种没有指定的且与实现相关的顺序返回。B-tree 索引默认将表项以升序方式存储，并将空值放在最后（在其他相等的项之间，表 TID 被视为平分符列）。我们可以在创建 B-tree 索引时通过下列选项来改变索引的排序

- ASC，指定上升排序（默认）。
- DESC，指定下降排序。
- NULLS FIRST，指定把空值排序在非空值前面。在指定 DESC 时，这是默认行为。
- NULLS LAST，指定把空值排序在非空值后面。在没有指定 DESC 时，这是默认行为。

那么典型的示例就是：

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

父主题：创建索引

### 14.3.4 创建 Hash 索引

Hash 索引只能处理简单等值比较。不论何时当一个索引列涉及到一个使用了 = 操作符的比较时，查询规划器将考虑使用一个 Hash 索引。下面的命令将创建一个 Hash 索引：

```
CREATE INDEX name ON table USING HASH (column);
```

父主题：创建索引

### 14.3.5 创建基于函数的索引

基于函数的索引时将索引建立在某个函数或者表达式的基础上。一个索引列并不一定是底层表的一个列，也可以是从表的一列或多列计算而来的一个函数或者标量表达式。这种特性对于根据计算结果快速获取表中内容是有用的。

有一些查询，需要对表的列进行计算，此时就适合创建基于函数的索引。例如，一种进行大小写不敏感比较的常用方法是使用 lower 函数：

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

由于 col1 没有索引，在执行语句时就需要先把列的值修改为小写，再和 value 进行匹配。为了加快速度，可以再 col1 列上创建索引，先对该列的值进行小写转换，然后再把索引存储在索引中，这样执行查询时就不再需要转换。函数结果之上的索引：

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

如果我们将该索引声明为 UNIQUE，它将阻止创建在 col1 值上只有大小写不同的行。

父主题：[创建索引](#)

### 14.3.6 创建多列索引

一个索引可以定义在表的多个列上。

例如，我们有这样一个表：

```
CREATE TABLE test2(  
  major int,  
  minor int,  
  name varchar  
);
```

如果我们经常根据多列的相交 (通常为多个 and 操作) 或者联合 (通常为多个 or 操作) 操作做查询时，例如从表 test2 中查询 major 和 minor 均为常量 constant 的 name 信息：

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

那么我们可以在 major 和 minor 上定义一个多列索引：

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

相比于在每个列上创建索引，多列索引占用的磁盘相对更少，索引数据更新时也将更快。目前，只有 B-tree、Bitmap、GiST、GIN 和 BRIN 索引类型支持多列索引，最多可以指定 512 个列。

父主题：[创建索引](#)

### 14.3.7 创建本地和全局索引

在分区表上创建的索引可分为全局索引和本地索引。

全局索引包括全局非分区索引（Global Nonpartitioned Indexes）和全局分区索引（Global Partitioned Indexes），是指与分区表有不同分区方式的索引，它是在分区表的所有分区数据基础上创建的索引。

本地索引（本地分区索引，Local Partitioned Indexes），是指在每个表分区上单独创建的索引，是一种局部索引，也是一种分区索引，某一个索引分区只能索引到一个表分区。

例如，在分区表上创建全局索引：

```
CREATE TABLE sales_table (COL1 INT, COL2 INT) PARTITION BY RANGE (COL1);
CREATE TABLE q1 partition of sales_table for values from (0) to (100);
CREATE TABLE q2 partition of sales_table for values from (101) to (1000);
CREATE unique INDEX GLOBAL_INDEX ON sales_table (COL2) GLOBAL;
```

对于非分区表而言，指定 GLOBAL/LOCAL 行为没有差别。对于分区表而言，当指定 GLOBAL 时创建全局索引，当不指定 GLOBAL 或者 LOCAL 时默认创建本地索引。

父主题：[创建索引](#)

## 14.4 修改索引

若要修改索引，模式中必须包含该索引。

- [关于修改索引](#)

可以使用 ALTER INDEX 或者 REINDEX 语句修改索引，两者的使用场景有区别。

- [修改或重建索引](#)

通过 ALTER INDEX 或者 REINDEX 语句，可以实现修改索引的定义或者重建索引。

父主题：[索引的管理](#)

### 14.4.1 关于修改索引

可以使用 ALTER INDEX 或者 REINDEX 语句修改索引，两者的使用场景有区别。

ALTER INDEX 用于更改一个现有索引的属性定义，如修改索引名称，变更表空间等。而 REINDEX 则根据表重建一个索引，并且替换旧索引。如下场景应考虑使用 REINDEX：

- 一个索引已经损坏，并且不再包含合法数据。尽管理论上这不会发生，实际上索引会因为软件缺陷或硬件失效损坏。REINDEX 提供了一种恢复方法。
- 一个索引变得“臃肿”，其中包含很多空的或者近乎为空的页面。KingbaseES 中的 B-树索引在特定的非常规访问模式下可能会发生这种情况。REINDEX 提供了一种方法来减少索引的空间消耗，即制造一个新版本的索引，其中没有死亡页面。
- 修改了一个索引的存储参数（例如填充因子），并且希望确保这种修改完全生效。

- 如果 CONCURRENTLY 选项的索引构建失败，该索引被保留为“invalid”。这样的索引是无用的，但是可以方便地使用 REINDEX 来重建它们。注意，只有 REINDEX INDEX 能够在无效索引上执行并发构建。

父主题：[修改索引](#)

## 14.4.2 修改或重建索引

通过 ALTER INDEX 或者 REINDEX 语句，可以实现修改索引的定义或者重建索引。

可以实现如下功能：

- 更改索引的名称
- 更改该索引的表空间为其他指定的表空间
- 把该索引标记为依赖于扩展
- 把和主表有相同表结构的主表变成该主表的一个分区
- 更改一个或者多个索引相关的存储参数
- 重置存储参数为默认值
- 为索引设置统计信息收集目标
- 重建索引

---

**注意：**不能更改索引列结构。

---

具体的修改索引典型示例如下：

重命名一个现有索引：

```
ALTER INDEX distributors RENAME TO new_distributors;
```

把一个索引移动到一个不同的表空间：

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

更改一个索引的填充因子（假设该索引方法支持填充因子）：

```
ALTER INDEX distributors SET (fillfactor = 75);  
REINDEX INDEX distributors;
```

为一个表达式索引设置统计信息收集目标：

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));  
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

重建单个索引：



```
REINDEX INDEX my_index;
```

重建表 my\_table 上的所有索引:

```
REINDEX TABLE my_table;
```

KingbaseES 将重新构建索引，而不采取任何锁来防止表上的并发插入、更新或删除:

```
REINDEX TABLE CONCURRENTLY my_broken_table;
```

另请参阅:

《KingbaseES SQL 语言参考手册》中获取关于 ALTER INDEX 和 REINDEX 语句的详细语法和描述信息

父主题: [修改索引](#)

## 14.5 删除索引

可以使用 DROP INDEX 语句删除索引。要执行这个命令你必须是该索引的拥有者。

删除索引的一些原因包括:

- 不再需要该索引;
- 该索引未能实现预期的性能改进。例如，表可能非常小;
- 应用程序不使用索引来查询数据;
- 索引已无效，必须在重新生成之前删除该索引;

当您删除一个索引时，该索引的在表空间上的所有索引段将得到释放。删除索引主要有两大类情况，如果使用 CREATE INDEX 语句显式创建索引，则可以用 DROP INDEX 语句删除该索引。如下面的语句删除 emp\_ename 索引:

```
DROP INDEX emp_ename;
```

如果是通过在表上定义键约束隐式的创建索引。则不能直接删除与已启用的 UNIQUE KEY 键或 PRIMARY KEY 键约束相关的索引。此情况，必须停用或删除该约束本身。如下面的语句删除主键约束 pk\_emp\_name，同时删除其对应的索引:

```
ALTER TABLE emp DROP CONSTRAINT pk_emp_name;
```

当然，删除表也能实现删除索引，将自动删除了所有与该表相关的索引。

另请参阅:

《KingbaseES SQL 语言参考手册》中获取关于 DROP INDEX 语句的详细语法和描述信息

父主题: [索引的管理](#)

## 14.6 索引视图

通过查询数据视图，您可以获取到索引相关的信息：

视图	描述
DBA_INDEXES ALL_INDEXES USER_INDEXES	DBA 视图会显示数据库中所有表上的索引信息。ALL 视图显示用户可访问的所有表上的索引信息。USER 视图仅限于用户拥有的索引。这些视图其中的一些列包含由 sys_stat_statements 工具或 ANALYZE 语句生成的统计信息。
DBA_IND_COLUMNS ALL_IND_COLUMNS USER_IND_COLUMNS	这些视图显示表上索引列的信息。这些视图其中的一些列包含由 sys_stat_statements 工具或 ANALYZE 语句生成的统计信息。

父主题：[索引的管理](#)

---

# 第 15 章 管理视图、序列和同义词

---

视图、序列和同义词都是数据库中重要的模式对象概念，您可以创建和管理数据库视图、序列和同义词。KingbaseES 数据库支持对这些模式对象的管理。

- 管理视图

您可以执行创建视图、替换视图、修改视图和删除视图等任务。

- 管理序列

您可以执行创建序列、修改序列、使用序列和删除序列等任务。

- 管理同义词

您可以执行创建同义词、使用同义词和删除同义词等任务。

- 视图、同义词和序列的数据字典视图

您可以查询指定数据字典视图，了解有关视图、同义词和序列的信息。

## 15.1 管理视图

您可以执行创建视图、替换视图、更改视图和删除视图等任务。

- 关于视图

视图的形式类似于普通表，我们可以从视图中查询数据，实际上它是建立在表上的一种虚表，从本质上说，视图是对存储的查询。

- 创建视图和连接视图

可以使用 CREATE VIEW 语句创建视图。每个视图可以由引用表、物化视图或其他视图的查询来定义。还可以创建连接视图，即在 FROM 子句中指定多个基表或视图。

- 替换视图

可以通过删除并重新创建视图，或者通过执行带有 OR REPLACE 子句的 CREATE VIEW 语句实现替换现有视图的定义。

- 在查询中使用视图

可以像查询表一样查询视图，您还可以在视图上执行数据操作语句 (DML) 相关的操作，但有一些限制。

- 修改视图

使用 ALTER VIEW 语句只能显式地重新编译无效的视图。

- 删除视图

您可以使用 DROP VIEW 语句删除视图。

父主题：管理视图、序列和同义词

## 15.1.1 关于视图

视图的形式类似于普通表，我们可以从视图中查询数据，实际上它是建立在表上的一种虚表，从本质上说，视图是对存储的查询。

视图从它所基于的表中获取数据，这些表称为基表。基表可以是实际的表，也可以是其他视图。在视图上执行的所有操作实际上都会影响视图的基表。可以像使用表一样使用视图。您可以对视图进行查询、更新、插入和删除，就像对表进行查询一样。

对视图的访问会转化成对其基表的访问，并且将所查询的结果以视图所规定的格式和顺序返回。因此当基表中的数据发生变化时，从视图中查询出的数据也随之改变了。

对用户而言通常要进行查询操作，甚至是复杂且频繁的查询，那么就可以为这个查询定义一个视图。假设用户经常执行下面的查询语句：

```
SELECT student_id, student_name
FROM student
WHERE age > 20 and gender = '男';
```

如果已经为该查询定义了一个名为 student\_age\_view 的视图，那么您就可以执行一条简单的语句获取您想要的结果。

```
SELECT * FROM student_age_view;
```

另外值得注意的是视图本身不保存对基表的查询结果，仅仅保存一条 SELECT 语句。从上面这个例子可以看出使用视图的目的是为了方便用户访问基表数据。实际上视图的另一个重要作用是确保用户对基表的安全访问。对于同一个表，不同的用户而言，我们可以定义不同规则的视图，从而保证用户只能进行允许的操作。

父主题：管理视图

## 15.1.2 创建视图和连接视图

可以使用 CREATE VIEW 语句创建视图。每个视图可以由引用表、物化视图或其他视图的查询来定义。还可以创建连接视图，即在 FROM 子句中指定多个基表或视图。

- 创建视图

可以使用 CREATE VIEW 语句创建视图。

- 创建连接视图

您还可以在 CREATE VIEW 语句的 FROM 子句中指定多个基表或视图进行视图创建。

- 创建 FORCE 视图

如果 CREATE VIEW 语句带有 FORCE 关键字，意味着无论视图的依赖检查是否成功，只要语句没有语法错误，数据库也会创建该视图。

父主题：管理视图

### 15.1.2.1 创建视图

可以使用 CREATE VIEW 语句创建视图。

创建视图需要满足以下要求：

- 要在模式中创建视图，必须具有 CREATE VIEW 权限。如果要在其他用户的模式中创建视图，必须具有 CREATE ANY VIEW 系统权限。您可以显式地或通过角色获得这些特权。
- 视图的所有者（无论是您还是其他用户）必须被授予访问视图定义中引用的所有对象的权限。所有者不能通过角色获得这些特权。另外，视图的功能依赖于视图所有者的权限。例如，如果视图的所有者只有 test 表的 INSERT 权限，那么视图只能用于向 test 表插入新行，而不能用于 SELECT、UPDATE 或 DELETE 行。
- 如果视图的所有者打算将视图的访问权限授予其他用户，则该所有者必须通过 GRANT OPTION 获得基本对象的对象权限，或者通过 ADMIN OPTION 获得系统权限。

每个视图可以由引用表、物化视图或其他视图的查询定义。与所有子查询一样，定义视图的查询不能包含 FOR UPDATE 子句。为了说明视图，定义一张实验表，模拟学校的学生信息：

```
CREATE TABLE student (
  student_id NUMBER(4),
  student_name CHAR(8) NOT NULL,
  gender VARCHAR2(2),
  age NUMBER(2),
  grade VARCHAR2(8),
  classid NUMBER(2),
  email VARCHAR2(20),
  sdate DATE,
  CONSTRAINT pk_nmb PRIMARY KEY (student_id),
  CONSTRAINT ck_check CHECK (gender in ('男','女')),
  CONSTRAINT ck_age CHECK (age BETWEEN 12 AND 60)
);
```

下面的语句在 student 表中的数据子集上创建一个视图：

```
CREATE VIEW student_age_view AS
SELECT student_id,student_name,gender,age
FROM student
WHERE age > 20 and gender = '男'
WITH LOCAL CHECK OPTION;
```

上述例子创建了 student\_age\_view 视图，用于查询年龄大于 20 且性别为男的学生 ID 和姓名信息。LOCAL CHECK OPTION 子句是可选的，可以用来限定 DML 操作，并且只根据当前视图的 WHERE 条件进行限定。换句话说就是，针对该视图如果想执行 INSERT 和 UPDATE 语句，应确保新增加的数据可以被该视图重新查询到。例如，下面的 INSERT 语句根据 student\_age\_view 视图成功地将一行性别为男，年龄 21（大于 20）的学生数据插入到 student 表中：

```
INSERT INTO student_age_view VALUES (1726,'王猛','男',21);
```

但是，下面的 INSERT 语句将会返回一个错误，因为它试图插入一个性别为女，年龄为 19 的女学生信息，这是因为 student\_age\_view 视图的查询只引用性别为男且年龄大于 20 的行：

```
INSERT INTO student_age_view VALUES (1739,'王萌','女',19);
```

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 CREATE VIEW 语句详细语法描述

父主题：[创建视图和连接视图](#)

### 15.1.2.2 创建连接视图

您还可以在 CREATE VIEW 语句的 FROM 子句中指定多个基表或视图进行视图创建。这些基于多个表创建的视图就被称为连接视图，主要目的在于简化连接查询。

为了清楚的说明连接视图，下列是本章节视图中使用到的 user\_info 和 practice\_detail 两个表的详细定义：

```
CREATE TABLE user_info (
  id int NOT NULL,
  device_id int NOT NULL,
  gender varchar(14) NOT NULL,
  age int ,
  university varchar(32) NOT NULL,
  gpa float,
  active_days_within_30 int ,
  question_cnt int ,
  answer_cnt int
);

CREATE TABLE question_detail (
  id int NOT NULL,
```

```

device_id int NOT NULL,
question_id int NOT NULL,
result varchar(32) NOT NULL
);

```

下面的创建视图语句创建了连接 user\_info 和 practice\_detail 表数据的 avg\_answer\_view 视图:

```

CREATE VIEW avg_answer_view AS
SELECT user_info.university,
       COUNT(question_id)/COUNT(distinct question_detail.device_id) AS avg_answer_cnt
FROM question_detail INNER JOIN user_info
ON user_info.device_id = question_detail.device_id
GROUP BY university;

```

上述视图例子根据 user\_info 和 practice\_detail 两表获取了指定的计算结果，但值得一提的是由于该视图中使用了 GROUP BY 子句和 DISTINCT 关键字，因此将不能通过该视图 avg\_answer\_view 进行 DML 相关操作。本节后面的内容将详细指出该限制条件。

父主题：[创建视图和连接视图](#)

### 15.1.2.3 创建 FORCE 视图

如果 CREATE VIEW 语句带有 FORCE 关键字，意味着无论视图的依赖检查是否成功，只要语句没有语法错误，数据库也会创建该视图。如果依赖检查成功则为有效视图，否则创建出来的就是无效视图。

比如视图引用了一个不存在的表或现有表中的无效列，或者视图所有者没有所需的权限时，数据库依然能创建该视图并将其保存到数据字典中。只是该视图还不能使用，当引用的表恢复（或者创建）后，该视图重新编译后能恢复使用。

要创建 FORCE 视图，CREATE VIEW 语句必须包含 FORCE 关键字，具体的命令如下：

```

CREATE FORCE VIEW AS ...;

```

另请参阅:

《KingbaseES SQL 语言参考手册》中关于 CREATE VIEW 语句中关于 FORCE 关键字的详细描述

父主题：[创建视图和连接视图](#)

### 15.1.3 替换视图

可以通过删除并重新创建视图，或者通过执行带有 OR REPLACE 子句的 CREATE VIEW 语句实现替换现有视图的定义。

要替换视图，您必须拥有删除和创建视图所需的所有权限。如果视图的 SELECT 定义必须修改，可以优先考虑使用带有 OR REPLACE 子句的 CREATE VIEW 语句执行视图替换，其他拥有关系、权限和非 SELECT 规则在内

的视图属性不会被更改; 视图不能使用 ALTER VIEW 语句更改 SELECT 定义。

替换视图的方式具体如下:

- 您可以直接删除并重新创建视图。

**注意:** 当一个视图被删除时, 所有相应的对象权限将从角色和用户中被撤销。在重新创建视图之后, 必须重新授权。

- 可以使用包含 OR REPLACE 子句的 CREATE VIEW 语句重新定义视图。OR REPLACE 子句会替换当前定义的视图, 并保留当前授权。

例如, 在前面小节所示创建的 student\_age\_view 视图, 如果需要重新定义视图 WHERE 子句指定的年龄和性别以及新增查询结果。可以使用以下语句替换 student\_age\_view 视图为当前定义:

```
CREATE OR REPLACE VIEW student_age_view AS
SELECT student_id, student_name, age
FROM student
WHERE age > 19 and gender = '女'
WITH LOCAL CHECK OPTION;
```

在替换视图之前, 请注意可能出现的影响:

- 替换一个视图将替换数据字典中的视图定义。视图引用的所有基础对象都不受影响。
- 如果 CHECK OPTION 中不再指定 LOCAL 关键字, 则将会根据当前新视图和所有基视图条件检查新数据。

父主题: [管理视图](#)

## 15.1.4 在查询中使用视图

可以像表一样查询视图, 您还可以在视图上执行数据操作语句 (DML) 相关的操作, 但有一些限制。

要对视图执行查询或 INSERT、UPDATE 或 DELETE 语句, 必须显式地或通过角色分别为视图拥有 SELECT、READ、INSERT、UPDATE 或 DELETE 对象权限。

类似于普通的表, 可以以相同的方式进行查询。例如, 要查询重新定义以后的 student\_age\_view 视图 (大于 19 岁的女生), 输入引用该视图的 SELECT 语句, 获取结果如下:

```
SELECT * FROM student_age_view;
```

student_id	student_name	gender	age
7844	宋义	男	20
7782	梅田田	女	21
7672	张华乐	男	20
7781	吉祥	男	21
7742	张丽莉	女	20



7900	黄金美	女	21
7654	倪文	女	20
7694	向玲	女	20
7698	向柯	男	20

可以使用视图向基表中插入、更新或从基表中删除行，但存在一些限制。下面的语句使用 `student_age_view` 视图向 `student` 表插入一个新行：

```
INSERT INTO student_age_view
VALUES (7950, '王雯雯','女',20);
```

上述数据之所以能被成功插入到基表，是因为该行数据能被视图重新选中查询到，即满足（只允许将年龄大于 19 且为女性），同时不存在以下限制：

- 如果视图是包含了 `DISTINCT` 操作符、`GROUP BY` 子句、`HAVING`、`LIMIT` 或 `OFFSET` 的查询定义，则不能使用该视图向基表插入、更新或从基表删除行。
- 如果视图是用 `WITH CHECK OPTION` 定义的，则查询必须满足一定条件，如果视图不能从基表中选择行，则不能使用视图向基表插入或更新行。
- 如果不包含 `DEFAULT` 子句的 `NOT NULL` 列在视图中的新数据不满足该约束，则不能使用视图将行插入到基表中。
- 视图的选择列不能包含任何聚合、窗口函数或者集合返回函数。

父主题：[管理视图](#)

## 15.1.5 修改视图

使用 `ALTER VIEW` 语句只能显式地重新编译无效的视图。

`ALTER VIEW` 语句允许您在运行前定位重新编译错误。要确保更改不会影响视图或依赖于它的其他对象，可以在更改视图的一个基表之后显式地重新编译视图。要使用 `ALTER VIEW` 语句，视图必须在您的模式中，或者您必须具有 `ALTER ANY TABLE` 系统权限。

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 `ALTER VIEW` 语句详细语法描述

父主题：[管理视图](#)

## 15.1.6 删除视图

您可以使用 `DROP VIEW` 语句删除视图。

可以删除模式中包含的任何视图。要删除另一个用户模式中的视图，必须具有 `DROP ANY VIEW` 系统权限。例如，下面的语句删除了 `student_age_view` 视图：

```
DROP VIEW student_age_view;
```

另请参阅:

《KingbaseES SQL 语言参考手册》中关于 DROP VIEW 语句详细语法描述

父主题: 管理视图

## 15.2 管理序列

您可以执行创建序列、修改序列、使用序列和删除序列等任务。

- 关于序列

序列是一种数据库对象，用来自动生成一组具有一定规律 (增加或者减小) 变化的连续不同序列号，一般应用于表的主键列。

- 创建序列

使用 CREATE SEQUENCE 语句创建序列。

- 修改序列

使用 ALTER SEQUENCE 语句修改序列。

- 使用序列

对用户而言，序列中的可用资源就是序列号，序列号可以被多个用户访问并递增或者递减。

- 序列信息的查询

序列作为一种数据库对象，其相关信息存储在数据字典中。

- 删除序列

如果不再需要某个序列，可以使用 DROP SEQUENCE 语句删除该序列。

父主题: 管理视图、序列和同义词

### 15.2.1 关于序列

序列是一种数据库对象，用来自动生成一组具有一定规律 (增加或者减小) 变化的连续不同序列号。

序列一般应用于表的主键列，可以被多个用户使用，也可以为多个表生成主键。设想，如果没有序列，则只能编写程序生成顺序值，而在使用序列的情况下，当向表插入数据时，主键列就直接使用序列号赋值，保证了主键列没有重复值，也可以获得更可靠的主键值。

父主题: 管理序列

## 15.2.2 创建序列

使用 CREATE SEQUENCE 语句创建一个序列。

要在模式中创建序列，您必须具有 CREATE SEQUENCE 系统权限。要在其他用户的模式中创建序列，必须具有 CREATE ANY SEQUENCE 的权限。例如，下面的语句创建了一个序列，用于为 student 表的 student\_id 主键列生成四位数的学生编号：

```
CREATE SEQUENCE stu_sequence
  INCREMENT BY 1
  START WITH 1000
  MINVALUE 1000
  MAXVALUE 9999
  CACHE 100
  NOCYCLE;
```

通过上述例子可以看出，您可以通过各种选项参数实现对序列的控制，指示序列是升序还是降序、序列号的起始值 (START WITH)、最小值 (MINVALUE) 和最大值 (MAXVALUE) 以及序列值之间的递增幅度 (INCREMENT BY) 等等。

其中 NOCYCLE 选项表示序列在达到最大值或最小值后不会生成更多值。如果希望序列号可以循环使用，请使用 CYCLE 选项。

CACHE 子句预先分配 100 个序列号保存在内存中。用户每使用一次序列，都需要对序列进行一次查询，当保存在内存中可以大大加快获取序列号，当缓存中的最后一个序列号被使用时，数据库将向缓存读入另一组序列号。

如果选择缓存一组序列号，则数据库可能会跳过该组序列号。例如，当实例异常关闭时 (例如，当发生实例故障或发出 SHUTDOWN ABORT 语句时)，已缓存但未使用的序列号将丢失。此外，已使用但未保存的序列号也会丢失。

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 CREATE SEQUENCE 语句详细语法描述

父主题：[管理序列](#)

## 15.2.3 修改序列

使用 ALTER SEQUENCE 语句修改序列。

要更改序列，您的模式必须包含该序列，您必须对该序列具有 ALTER 对象权限，或者必须具有 ALTER ANY SEQUENCE 系统权限。您可以通过修改序列的生成数值的定义参数来实现序列修改。

下面的例子改变了 stu\_sequence 序列的增长幅度为 2，循环使用序列号以及缓存数量变更为 20：

```
ALTER SEQUENCE stu_sequence
  INCREMENT BY 2
```

```
CYCLE
CACHE 20;
```

同时，您在修改序列的时候应注意到数值的合理性，比如下列语句则明显是不合理的，因为起始值 (START WITH) 是 1000，而下列修改试图将最小值修改为 2000：

```
ALTER SEQUENCE stu_sequence
INCREMENT BY 2
MINVALUE 2000
MAXVALUE 8000
CYCLE
CACHE 20;
```

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 ALTER SEQUENCE 语句详细语法描述

父主题：管理序列

## 15.2.4 使用序列

对用户而言，序列中的可用资源就是序列号，序列号可以被多个用户访问并递增或者递减。

要使用序列，您的模式必须包含该序列，或者您必须已被授予其他用户序列的 SELECT 对象权限。一旦序列完成创建，序列就可以被多个用户 (对包含该序列的序列拥有 SELECT 对象权限) 访问和递增，而无需等待。

- 使用 NEXTVAL 生成序列号

序列一旦生成，用户就可以在 SQL 语句中用 CURRVAL 和 NEXTVAL 伪列来存取序列的值。

- 使用 CURRVAL 序列号

要使用当前序列值，请在 SQL 语句中引用 seq\_name.CURRVAL。

父主题：管理序列

### 15.2.4.1 使用 NEXTVAL 生成序列号

序列一旦创建，用户就可以使用 SQL 语句中从 CURRVAL 和 NEXTVAL 伪列中获取序列的值。

CURRVAL 和 NEXTVAL 伪列就如同序列的指针一样，NEXTVAL 每获得一个值，指针就向后移动一个位置，这是就可以用 CURRVAL 获取当前序列号。即，其意义如下：

- CURRVAL 返回当前的序列值；
- NEXTVAL 返回下一个可用的序列号。

如上描述，NEXTVAL 和 CURRVAL 并不是保留字或关键字，可以在 SELECT、INSERT 或 UPDATE 等 SQL 语句中用作伪列名。在第一次使用序列号时，必须先访问 NEXTVAL 伪列。要生成和使用序列号，请引用 SQL 语句中的 NEXTVAL，如下示例，从伪表 dual 中查看序列号：

```
SELECT stu_sequence.NEXTVAL FROM dual;
```

这条语句的执行结果为：

```
NEXTVAL
-----
2000
```

例如，假设有新的学生注册，序列号可以在值列表中直接引用：

```
INSERT INTO student VALUES (stu_sequence.NEXTVAL, '张丽','女',18,'高中二年级',2,'105@qq.com',SYSDATE);
```

当需要更新某位同学的序列号时，可以在 UPDATE 语句的 SET 子句中引用序列号。例如：

```
UPDATE student SET student_id = stu_sequence.NEXTVAL WHERE student_id = 2022;
```

父主题：使用序列

## 15.2.4.2 使用 CURRVAL 序列号

要使用当前序列值，请在 SQL 语句中引用 seq\_name.CURRVAL。

使用 CURRVAL 之前，seq\_name.NEXTVAL 必须在用户会话中已被引用 (在当前或以前的事务中) 至少一次。可以根据需要多次引用 CURRVAL，包括在同一条语句中多次引用。在引用 NEXTVAL 之前，不会生成下一个序列号。继续前面的示例，您将通过插入新学生数据，进行注册：

```
INSERT INTO student VALUES (stu_sequence.CURRVAL, '刘意','女',17,'高中一年级',4,'106@qq.com',SYSDATE);
INSERT INTO student VALUES (stu_sequence.CURRVAL, '赵文','男',19,'高中一年级',5,'107@qq.com',SYSDATE);
```

假设上一节给出的 INSERT 语句生成了一个新的序号 2022，那么本节中的语句插入的两行都插入了序号为 2022 的行。当然，如果引用 CURRVAL 的列定义了主键约束，此方法的引用会约束报错。

父主题：使用序列

## 15.2.5 缓存序列号

缓存序列号可以提高访问效率。

- 关于缓存序列号

在序列缓存中访问序列号比从磁盘读取序列号更快。

- 序列缓存中的条目数

当应用程序访问序列缓存中的序列时，将快速读取序列号。

- 缓存中的序列号数量

当序列被读入缓存时，指定数量的序列号就已经被生成并存储在缓存中。

父主题: 管理序列

### 15.2.5.1 关于缓存序列号

在序列缓存中访问序列号比从磁盘读取序列号更快。序列缓存由条目组成，每个条目可以保存单个序列的多个序列号。

遵循这些指南快速访问所有序列号:

- 请确保序列缓存了足够的序列号，保证应用程序并发使用所有序列。
- 高并发情况应增加缓存中序列号的数量。

父主题: 缓存序列号

### 15.2.5.2 序列缓存中的条目数

当应用程序访问序列缓存中的序列时，将快速读取缓存中的序列号。

如果应用程序访问的序列不在缓存中，那么在使用序列号之前，必须将序列先从磁盘读取到缓存中。如果应用程序并发地使用许多序列，那么序列缓存可能不够大，不能容纳所有序列。在这种情况下，访问序列号可能经常需要读取磁盘。为了快速访问所有序列，请确保缓存中有足够数量的序列号，从而保证应用程序并发使用所有序列。

父主题: 缓存序列号

### 15.2.5.3 缓存中的序列号数量

当序列被读入缓存时，指定数量的序列号就已经被生成并存储在缓存中。

然后可以快速地访问这些值。存储在缓存中的序列号的数量由 CREATE SEQUENCE 语句中的 CACHE 参数决定，该参数的默认值为 1。CREATE SEQUENCE 语句创建 seq\_test2 序列，该序列的 50 个值就存储在 SEQUENCE 缓存中:

```
CREATE SEQUENCE seq_test2
  CACHE 50;
```

父主题: 缓存序列号

## 15.2.6 删除序列

如果不再需要某个序列，可以使用 DROP SEQUENCE 语句删除该序列。

用户可以删除自己创建的任何序列。要在删除其他用户的序列，必须具有 DROP ANY SEQUENCE 系统权限。序列删除后，数据字典中不再保留相关信息。

例如，下面的语句删除了前面章节定义的 stu\_sequence 序列：

```
DROP SEQUENCE stu_sequence;
```

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 DROP SEQUENCE 语句详细语法描述

父主题：管理序列

## 15.3 管理同义词

您可以执行创建同义词、使用同义词和删除同义词等任务。

- 关于同义词

可以为表、视图、存储过程，序列等数据库对象建立同义词，也可以为同义词再次建立同义词，甚至可以为一个不存在的对象建立同义词。

- 创建同义词

使用 CREATE SYNONYM 语句创建同义词。

- 在 DML 语句中使用同义词

在 DML 语句中，同义词的使用方式和基对象的使用方式完全相同。

- 删除同义词

您可以通过 DROP SYNONYM 语句来删除一个不再需要的同义词。

父主题：管理视图、序列和同义词

### 15.3.1 关于同义词

可以为表、视图、存储过程，序列等数据库对象建立同义词，也可以为同义词再次建立同义词，甚至可以为一个不存在的对象建立同义词。

同义词本身并不包含原对象中的数据或者代码，它仅仅充当一个指针，是数据库对象的别名。此外，同义词易于使用，并能降低了数据库用户 SQL 语句的复杂性。同义词允许底层对象被重命名或移动，但必须重定义同义词，基于同义词的应用程序无需修改即可继续工作。您可以创建公共和私有的同义词。

公共同义词由名为 PUBLIC 的特殊用户组拥有，一般由 DBA 创建，数据库中的每个用户都可以访问它。私有同义词包含在特定用户的模式中（非 public 模式），仅对该用户和底层对象的被授予者可用。同义词本身不涉及安全，当您赋予一个同义词对象权限时，您实质上是在给同义词的基对象赋予权限。

父主题：管理同义词

## 15.3.2 创建同义词

使用 CREATE SYNONYM 语句创建同义词。

创建同义词时，底层模式对象不需要存在，也不需要访问对象的权限，也能创建同义词。下面的语句在包含在 schm 模式中的 tab1 表上创建一个名为 public\_syn\_tab1 的公共同义词：

```
CREATE PUBLIC SYNONYM public_syn_tab1 FOR schm.tab1;
```

您也可以使用带有 OR REPLACE 子句的 CREATE SYNONYM 将一个现有同义词定义替换掉：

```
CREATE OR REPLACE PUBLIC SYNONYM public_syn_tab1 FOR schm.tab1;
```

在创建同义词应注意的：

- 同义词不能与同一模式中其他同义词同名，但不同模式下可以有同名的同义词
- 创建公有同义词不能指定模式
- 创建私有同义词不能指定 public 模式
- 创建私有同义词必须指定模式 (在当前模式路径为 public 时)

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 CREATE SYNONYM 语句详细语法描述

父主题：[管理同义词](#)

## 15.3.3 在 DML 语句中使用同义词

在 DML 语句中，同义词的使用方式和基对象的使用方式完全相同。

只要您被赋予了基对象权限，您就可以使用其他用户的私有同义词。例如，您只拥有 student 表的 SELECT 权限，student 表有一个同义词 stu，这时，您就只能对 stu 同义词进行查询操作，而不能使用它进行插入数据。同理，如果拥有基对象的插入权限，则可以使用其同义词进行插入操作。

例如，一个名称为 student 的同义词，下面语句是有效的：

```
INSERT INTO stu VALUES (stu_sequence.CURRVAL, '刘恩斯',18,'高中二年级',4,'108@qq.com',SYSDATE);
```

父主题：[管理同义词](#)

## 15.3.4 删除同义词

您可以通过 DROP SYNONYM 语句来删除一个不再需要的同义词。

如果您要删除公共同义词，您必须拥有 DROP PUBLIC SYNONYM 系统权限。当删除私有同义词时，省略 PUBLIC 关键字；当删除公共同义词时，需要包含 PUBLIC 关键字。



例如，下面语句删除名为 `syn_test` 的同义词：

```
DROP SYNONYM syn_test;
```

下面的语句删除名为 `public_syn` 的公共同义词：

```
DROP PUBLIC SYNONYM public_syn;
```

另请参阅：

《KingbaseES SQL 语言参考手册》中关于 `DROP SYNONYM` 语句详细语法描述

父主题：管理同义词

## 15.4 视图、同义词和序列的数据字典视图

您可以查询指定的数据字典视图，了解有关视图、同义词和序列的信息。

以下视图记录了显示关于视图、同义词和序列的信息：

视图	描述
DBA_VIEWS, ALL_VIEWS, USER_VIEWS	DBA 视图描述数据库中的所有视图。ALL 视图被限制为当前用户可以访问的视图。USER 视图仅限于当前用户拥有的视图。
DBA_SYNONYMS, ALL_SYNONYMS, USER_SYNONYMS	描述同义词的视图。
DBA_SEQUENCES, ALL_SEQUENCES, USER_SEQUENCES	描述序列的视图。

- 视图信息的查询

视图作为一种数据库对象，其相关信息存储在数据字典中。

- 序列信息的查询

序列作为一种数据库对象，其相关信息存储在数据字典中。

- 同义词信息的查询

同义词作为一种数据库对象，其相关信息存储在数据字典中。

父主题：管理视图、序列和同义词

### 15.4.1 视图信息的查询

视图作为一种数据库对象，其相关信息存储在数据字典中。

与视图相关的视图有三个：

- USER\_VIEWS，当前用户有关的视图信息
- DBA\_VIEWS，系统中所有的视图信息
- ALL\_VIEWS，当前用户可以访问的所有视图信息

例如要查询视图 stu\_age\_view 的相关信息，可以执行下面的 SELECT 语句：

```
SELECT text FROM user_views WHERE view_name = 'stu_age_view';
```

其中 text 中存储的是创建视图时使用的 SELECT 语句。

父主题：[视图、同义词和序列的数据字典视图](#)

### 15.4.2 序列信息的查询

序列作为一种数据库对象，其相关信息存储在数据字典中。

与序列相关的视图有三个：

- USER\_SEQUENCES，当前用户有关的序列信息
- DBA\_SEQUENCES，系统中所有的序列信息
- ALL\_SEQUENCES，当前用户可以访问的所有序列信息

例如如果需要了解 stu\_sequence 序列的信息，可以执行下面的 SELECT 语句：

```
SELECT min_value,max_value,increment_by,last_number,cycle_flag
FROM dba_sequences
WHERE sequence_name = 'stu_sequence';
--或者
SELECT *
FROM dba_sequences
WHERE sequence_name = 'stu_sequence';
```

父主题：[视图、同义词和序列的数据字典视图](#)

### 15.4.3 同义词信息的查询

同义词作为一种数据库对象，其相关信息存储在数据字典中。

与同义词相关的视图有三个：

- USER\_SYNONYMS, 当前用户有关的同义词信息
- DBA\_SYNONYMS, 系统中所有的同义词信息, 包括每个用户创建的同义词
- ALL\_SYNONYMS, 当前用户可以访问的所有同义词信息, 包括私有和共有同义词

要查询当前用户创建了那些同义词, 各代表那个用户的对象, 可以执行下列 SELECT 语句:

```
SELECT synonym_name,table_name,table_owner
FROM user_synonyms;
```

如果要在整个数据库范围内查询 system 用户创建的同义词, 可以执行下面的 SELECT 语句:

```
SELECT synonym_name,table_name,table_owner
FROM dba_synonyms
WHERE owner='system';
```

```
SELECT synonym_name,table_name,table_owner
FROM dba_synonyms
WHERE owner='PUBLIC';
```

父主题: 视图、同义词和序列的数据字典视图

# 第 16 章 管理数据库自动维护任务

KingbaseES 基于后台进程来实现自动作业功能。您可以创建、修改和删除自动作业，以及启用或禁用某个自动作业。

- 关于自动维护任务

自动维护任务是指自动定期的对数据库进行维护操作的任务。

- 使用自动作业功能

KingbaseES 通过 `kdb_schedule` 插件提供了 `DBMS_JOB` 和 `DBMS_SCHEDULER` 包，其中定义了自动作业功能的相关函数，在使用 `kdb_schedule` 之前，我们需要将它添加到 `kingbase.conf` 文件的 `shared_preload_libraries` 中。

- 启用和禁用自动维护任务

如果使用 `DBMS_JOB` 包创建的自动维护任务，则可以使用 `BROKEN` 子程序将 `job` 任务设置为中止，`job` 将不会再运行但仍然存在，后续可以重新启用。

- 修改自动作业间隔时间

创建自动作业任务后，可以通过 `DBMS_JOB` 的 `INTERVAL` 子程序更改 `job` 的作业间隔时间（涉及日历表示法）。

## 16.1 关于自动维护任务

自动维护任务是指自动定期的对数据库进行维护操作的任务。您可以启用和禁用某个自动作业，创建、修改和删除自动作业。

KingbaseES 的自动作业功能是通过 `kdb_schedule` 插件提供的 `DBMS_JOB` 和 `DBMS_SCHEDULER` 包实现，其中 `DBMS_JOB` 模式提供调度和管理计划任务的功能，该包功能可以由 `DBMS_SCHEDULE` 替代，本章节主要结合 `DBMS_JOB` 说明自动作业功能。

父主题：管理数据库自动维护任务

## 16.2 使用自动作业功能

KingbaseES 通过 `kdb_schedule` 插件提供了 `DBMS_JOB` 和 `DBMS_SCHEDULER` 包，其中定义了自动作业功能的相关函数，在使用 `kdb_shchedule` 之前，我们需要将他添加到 `kingbase.conf` 文件的 `shared_preload_libraries` 中。配置命令如下：

```
shared_preload_libraries = 'kdb_schedule' # (change requires restart)
```

并重启数据库，并在数据库中通过下列 `create extension` 命令创建该插件：

```
create extension kdb_schedule;
```

以下三个配置选项可以在配置文件中指定：

- `job_queue_processes`：允许用户启动的最大并发数，当其值设置为 0 时，表示不启动自动作业功能，默认为 0，不开启自动作业。
- `sys_job.log_level`：用于设置 `JOB` 后台进程的日志级别，更改后需要重新加载配置文件，可选项：`LOG_ERROR`，`LOG_WARNING`，`LOG_DEBUG`，默认为 `LOG_ERROR`。
- `sys_job.poll_time`：用于设置轮询系统表间隔时间，单位秒，默认值为 10 秒。

父主题：[管理数据库自动维护任务](#)

## 16.3 启用和禁用自动维护任务

如果使用 `DBMS_JOB` 包创建的自动维护任务，则可以使用 `BROKEN` 子程序将 `job` 任务设置为中止，`job` 将不会再运行但任然存在，后续可以重新启用。

示例如下，将 `job ID` 为 1 的作业任务立刻中止：

```
call dbms_job.broken(1,TRUE,now());
```

如果想恢复使用（`true` 为禁用，`false` 为启用），示例如下：

```
call dbms_job.broken(1,FALSE,now());
```

如果使用 `DBMS_SCHEDULER` 包创建的维护任务，则可以使用 `DISABLE` 子程序将 `job` 任务设置为禁用，`job` 将不会再运行但任然存在。

示例如下，将禁用名为 `myjob` 的维护任务

```
call DBMS_SCHEDULER.DISABLE('myjob', false, 'STOP_ON_FIRST_ERROR');
```

另请参阅：

《KingbaseES 插件参考手册》中 kdb\_schedule 版块获取关于子程序 DBMS\_JOB 和 DBMS\_SCHEDULER 的详细说明

父主题：管理数据库自动维护任务

## 16.4 修改自动作业间隔时间

采用 DBMS\_JOB 创建自动作业任务后，可以通过 DBMS\_JOB 的 INTERVAL 子程序更改 job 的作业间隔时间涉及日历表示法。

示例如下，将 job ID 为 1 的作业，更改间隔时间属性为每天晚上 23 点：

```
call dbms_job.interval(1, 'Freq=daily;BYHOUR=23;BYMINUTE=0;BYSECOND=0');
```

另请参阅：

《KingbaseES 插件参考手册》中 kdb\_schedule 版块获取关于子程序 interval 和日历表示法的详细说明

父主题：管理数据库自动维护任务

---

# 第 17 章 数据库调度概念

---

您可以使用调度器 (Scheduler) 调度任务。

- 调度器概述

KingbaseES 数据库拥有作业调度功能，可以帮助您简化数百甚至数千个任务的调度。KingbaseES 调度器 (Scheduler) 依赖于 PL/SQL 提供的 DBMS\_SCHEDULER 相关函数和存储过程。

- 作业和支持的调度对象

您可以使用作业和其他调度对象进行任务调度。

- 关于作业的更多描述

作业存在多种类型，一个作业实例表示一个具体运行的作业。

## 17.1 调度器概述

KingbaseES 数据库拥有作业调度功能，可以帮助您简化数百甚至数千个任务的调度。KingbaseES 调度器 (Scheduler) 依赖于 PL/SQL 提供的 DBMS\_SCHEDULER 相关函数和存储过程。

您可以使用调度器在生产环境中自主控制在何时生成何种计算任务，同时可以帮助您有效地管理这些任务。无需人工干预的自动任务，可以有效的降低人工操作成本，同时可靠的例行程序，可以最大限度地减少人为错误。

作业调度器提供了复杂、灵活的作业调度功能，您可以使用它来：

- 运行数据库程序单元

您可以在本地数据库上运行程序单元，即 PL/SQL 匿名块、PL/SQL 存储过程。

- 运行外部可执行文件 (数据库外部的可执行文件)

您可以在本地系统上运行外部可执行文件，例如应用程序、shell 脚本。

- 基于时间的调度：

您可以将作业在特定的日期和时间运行，可以配置为只运行一次，也可以配置为规律性的重复运行。

父主题：数据库调度概念

## 17.2 作业和支持的调度对象

您可以使用作业和其他调度对象进行任务调度。

- 关于作业和支持调度对象

要使用调度器，则需要创建调度对象。调度对象定义了作业调度的内容、时间。调度对象支持以模块化的方式管理任务。模块化方法的一个优点是，当创建与现有任务类似的新任务时，可以复用对象。

- 程序

程序对象描述调度器要运行的内容。

- 调度计划

调度计划对象指定作业何时运行以及运行多少次。

- 作业

作业就是用户定义的任务。

父主题：数据库调度概念

### 17.2.1 关于作业和支持调度对象

要使用调度器，则需要创建调度对象。调度对象定义了作业调度的内容、时间。调度对象支持以模块化的方式管理任务。模块化方法的一个优点是，当创建与现有任务类似的新任务时，可以复用对象。

主要的调度对象是作业。作业定义了要执行的动作、动作的时间表。大多数其他调度对象创建来是为了支持作业。

KingbaseES 的自动作业功能是通过 `kdb_schedule` 插件提供的 `DBMS_JOB` 和 `DBMS_SCHEDULER` 包实现，其中 `DBMS_JOB` 模式提供调度和管理调度任务的功能，该包功能可以由 `DBMS_SCHEDULER` 替代。

这些调度对象将在本节后面进行描述。

父主题：作业和支持的调度对象

### 17.2.2 程序

程序对象描述调度器要运行的内容。

程序包括：

- 操作：例如，存储过程的名称、操作系统文件系统中可执行文件的名称（“外部可执行文件”）或 PL/SQL 匿名块。
- 类型：          PLSQL\_BLOCK、STORED\_PROCEDURE、SQL\_SCRIPT、EXECUTABLE、EXTERNAL\_SCRIPT、BACKUP\_SCRIPT 或可执行文件，其中可执行文件表示外部可执行文件。
- 参数数量：存储过程或外部可执行文件接受的参数数量。



程序是一个独立于作业的实体。作业则在某个确定的时间运行，并调用某个程序。您可以创建指向现有程序对象的作业，这意味着不同的作业可以使用相同的程序，并在不同的时间以不同的设置运行该程序。有了正确的权限，不同的用户可以使用相同的程序，而不必重新定义它。因此，您可以创建程序库，用户可以从现有程序列表中进行选择。

父主题：作业和支持的调度对象

### 17.2.3 调度计划

调度计划对象 (schedule) 指定作业何时运行以及运行多少次。

调度计划可以被多个作业共享。例如，对于许多作业来说，一个商业季度的结束可能是一个常见的时间框架。与每次季度未定义新作业不同，作业创建者可以指向一个已命名的调度计划。KingbaseES 支持基于时间的调度，您可以让作业立即运行或稍后运行。时间调度包括开始日期及时间、结束日期及时间、重复间隔。

父主题：作业和支持的调度对象

### 17.2.4 作业

作业就是用户定义的任务

- 关于作业

作业对象是描述用户定义任务的元数据集合。它定义了必须执行什么 (动作)、何时 (一次性或重复执行的调度或触发事件)。作业有一个所有者，即创建作业的模式。

- 指定作业操作

您可以通过指定数据库程序或外部可执行文件或现有程序对象 (程序) 的名称来指定作业的操作。

- 指定作业调度

您可以通过设置作业对象的属性或现有调度对象的名称来指定作业调度。

父主题：作业和支持的调度对象

#### 17.2.4.1 关于作业

作业对象是描述用户定义任务的元数据集合。它定义了必须执行什么 (动作)、何时 (一次性或重复执行的调度或触发事件)。作业有一个所有者，即创建作业的模式。

运行数据库程序的作业称为数据库作业。运行外部可执行文件的作业称为外部作业。

父主题：作业

### 17.2.4.2 指定作业操作

您可以通过指定数据库程序或外部可执行文件或现有程序对象 (程序) 的名称来指定作业的操作。

您可以通过以下方式之一指定作业操作:

- 通过将数据库程序单元或要运行的外部可执行文件指定为作业属性。这称为内联指定作业操作。
- 通过将现有程序的名称指定为作业属性, 该程序指定要运行的数据库程序单元或外部可执行文件。

父主题: [作业](#)

### 17.2.4.3 指定作业调度

可以通过设置作业对象的属性或现有调度对象 (schedule) 的名称来指定作业调度。

您可以通过以下方式之一指定作业调度:

- 通过设置作业对象的属性来定义开始和结束日期以及重复间隔, 或定义触发作业的事件。
- 通过将现有调度的名称指定为作业属性, 该名称定义了开始和结束日期、重复时间间隔或定义了事件。

父主题: [作业](#)

## 17.3 关于作业的更多描述

作业存在多种类型。一个作业实例表示一个具体运行的作业。您可以自定义作业参数来覆盖默认的参数值。

- [作业类别](#)

KingbaseES 数据库的 Scheduler 支持多种类型的作业。

- [作业实例](#)

一个作业实例表示一个具体运行的作业。一个只调度运行一次的作业只有一个实例。具有重复调度或每次事件发生时运行的作业有多个实例, 作业的每次运行代表一个实例。

- [作业参数](#)

当作业引用程序对象 (程序) 时, 可以提供作业参数来覆盖默认的程序参数值, 或者为没有默认值的程序参数提供值。

另请参阅:

请参考《KingbaseES 插件参考手册》获取 CREATE\_SCHEDULE 包的更多详细信息

父主题: [数据库调度概念](#)

## 17.3.1 作业类别

KingbaseES 支持多种类型的作业。

- 数据库作业

数据库作业运行数据库程序单元。可以运行本地和远程数据库作业。

- 外部作业

外部作业在数据库外部运行可执行程序。您可以运行本地的外部作业。

父主题：[关于作业的更多描述](#)

### 17.3.1.1 数据库作业

数据库作业运行数据库程序单元。可以运行本地和远程数据库作业。

- 关于数据库作业

数据库作业运行数据库程序单元，包括 PL/SQL 匿名块、PL/SQL 存储过程。

- 本地数据库的作业

本地数据库作业以数据库用户（即作业所有者）的身份在原始数据库上运行。作业所有者是创建作业的模式名称。

- 远程数据库作业

远程数据库作业的位置数据库可以是远程主机上的数据库，也可以是原始数据库所在主机上的另一个数据库实例。

父主题：[作业类别](#)

#### 17.3.1.1.1 关于数据库作业

数据库作业运行的数据库程序单元，包括了 PL/SQL 匿名块、PL/SQL 存储过程。对于内联指定操作的数据库作业，`job_type` 被设置为 ‘PLSQL\_BLOCK’ 或 ‘STORED\_PROCEDURE’，并且 `job_action` 包含 PL/SQL 匿名块的文本或存储过程的名称。（如果程序是一个命名程序对象，而不是内联指定的程序操作，则必须设置相应的 `program_type` 和 `program_action`。）在原始数据库上运行的数据库作业称为本地数据库作业。在初始数据库以外的位置数据库上运行的数据库作业称为远程数据库作业。可以在原始数据库上的作业日志视图中查看本地数据库和远程数据库作业的运行结果。

父主题：[数据库作业](#)

#### 17.3.1.1.2 本地数据库的作业

本地数据库作业以数据库用户（即作业所有者）的身份在原始数据库上运行。作业所有者是创建作业的模式名称。

父主题: [数据库作业](#)

### 17.3.1.1.3 远程数据库作业

远程数据库作业的位置数据库可以是远程主机上的 KingbaseES 数据库, 也可以是原始数据库所在主机上的另一个数据库实例。

另请参阅:

《KingbaseES 插件参考手册》获取关于 DBMS\_SCHEDULER 包的更多信息。

父主题: [数据库作业](#)

## 17.3.1.2 外部作业

外部作业在数据库外部运行可执行程序。您可以运行本地外部作业。

- [关于外部作业](#)

外部作业运行外部可执行文件。外部可执行文件是运行在数据库外部 (即数据库外部) 的操作系统可执行文件。

- [关于本地外部作业](#)

本地外部作业与调度该作业的 KingbaseES 数据库在同一台计算机上运行其外部可执行文件。

父主题: [作业类别](#)

### 17.3.1.2.1 关于外部作业

外部作业运行外部可执行文件。外部可执行文件是运行在数据库外部的操作系统的可执行文件。对于外部作业, job\_type 指定为 'EXECUTABLE'。(如果使用命名程序, 对应的 program\_type 将是 'EXECUTABLE'。)job\_action(或对应的 program\_action) 是所需的外部可执行文件的完全依赖于操作系统的路径, 不包括任何命令行参数。例如: /usr/local/bin/perl。

请注意, Windows 批处理文件不是直接可执行的, 必须在命令提示符 (cmd.exe) 中运行。

外部可执行文件必须作为某个操作系统用户运行。因此, Scheduler 使您能够将操作系统凭据分配给您创建的任何外部作业。与远程数据库作业一样, 可以使用凭据对象 (凭据) 指定这些凭据, 并将凭据分配给外部作业。

本地外部作业与调度该作业的数据库在同一台计算机上运行其外部可执行文件。

父主题: [外部作业](#)

### 17.3.1.2.2 关于本地外部作业

本地外部作业与调度该作业的 KingbaseES 数据库在同一台计算机上运行其外部可执行文件。

父主题: [外部作业](#)

## 17.3.2 作业实例

一个作业实例就表示一个特定运行的作业。只调度运行一次的作业只有一个实例。具有重复调度或每次事件发生时运行的作业有多个实例，作业的每次运行代表一个实例。

例如，一个只调度在 2009 年 10 月 8 日运行的作业只有一个实例，调度一周的每天中午运行的作业有 7 个实例。

父主题：[关于作业的更多描述](#)

## 17.3.3 作业参数

当作业引用程序对象 (程序) 时，可以设置作业参数来覆盖默认的程序参数值，或者为没有默认值的程序参数提供值。您还可以为作业指定的内联操作 (例如，存储过程) 提供参数值。

在定义了所有必需的程序参数值 (作为引用的程序对象中的默认值或作为作业参数) 之前，不能启用作业。

作业的一个常见示例是运行一组夜间报告。如果不同部门需要不同的报表，可以为该任务创建程序，供不同部门的不同用户共享。程序操作运行一个报告脚本，程序有一个参数: 部门编号。然后，每个用户都可以创建指向该程序的作业，并可以将部门编号指定为作业参数。

另请参阅:

《KingbaseES 插件参考手册》获取 CREATE\_JOB 包的更多详细信息

父主题：[关于作业的更多描述](#)

---

# 第 18 章 数据库作业调度

---

您可以使用 DBMS\_SCHEDULER 创建、运行和管理作业。

本章描述了如何使用 DBMS\_SCHEDULER 包来处理 Scheduler 对象。

- 作业调度及其子程序

您可以使用 DBMS\_SCHEDULER 包中的子程序来管理常见任务和调度计划。

- 创建程序

创建程序为了告诉调度器要运行什么。

- 删除程序

您可以通过 DBMS\_SCHEDULER 的 DROP\_PROGRAM 删除一个或多个程序。

- 创建调度计划

您可以通过 DBMS\_SCHEDULER 的 CREATE\_SCHEDULE 创建调度计划。

- 删除调度计划

您可以通过 DBMS\_SCHEDULER 的 DROP\_SCHEDULE 删除调度计划。

- 创建作业

当您完成了程序和调度计划的创建，意味着可以基于两者创建一个有计划的作业（JOB）。

- 删除作业

当我们不再需要作业时，可以使用 DROP\_JOB 删除一个指定的作业。

- 运行作业

通过调用 DBMS\_SCHEDULER.RUN\_JOB，您可以使用 RUN\_JOB 程序来测试作业或者在调度计划之外运行它。

- 禁用作业

您可以通过 DBMS\_SCHEDULER 的 DISABLE 禁用作业。

- 启用作业

您可以通过 DBMS\_SCHEDULER 的 ENABLE 禁用作业。

- 任务的日历表示法

创建的任务中会涉及使用日历表示法进行时间定义，日历表示法是用来定义重复计划的实现方法，能实现一些复杂的时间计划。

- 计算符合规则的下一个日期

您可以通过 DBMS\_SCHEDULER 的 EVALUATE\_CALENDAR\_STRING 计算符合规则的下一次计划运行时间。

另请参阅:

有关 DBMS\_SCHEDULER 的详细信息，请参见《KingbaseES 插件参考手册》中的 kdb\_schedule 模块。

## 18.1 作业调度及其子程序

您可以使用 DBMS\_SCHEDULER 包中的子程序来管理常见任务和调度计划。

下图展示了常见任务及其 DBMS\_SCHEDULER 子程序：

表 18.1.1: DBMS\_SCHEDULER 软件包子程序

子程序	描述
CREATE_PROGRAM Procedure	创建一个程序
DROP_PROGRAM Procedure	删除一个程序
CREATE_SCHEDULE Procedure	创建一个调度计划
DROP_SCHEDULE Procedure	删除一个调度计划
EVALUTE_CALENDAR_STRING Procedure	计算符合规则的下一个日期
CREATE_JOB Procedure	创建一个作业
DROP_JOB Procedure	删除一个作业
RUN_JOB Procedure	运行一个作业
DISABLE Procedure	将一个作业状态设置为 DISABLE
ENABLE Procedure	将一个作业状态设置为 ENABLE

DBMS\_SCHEDULER 具有如下限制：

- 需要创建 kdb\_schedule 扩展。
- 间隔时间采用日历表示法。
- 使用该功能需要必要的 GUC 参数。

父主题：数据库作业调度

## 18.2 创建程序

创建程序为了告诉调度器要运行什么。

默认情况下，程序在创建者的模式中创建。要在另一个用户的模式中创建程序，必须使用模式名称对程序名进行限定。要让其他用户使用您的程序，他们必须具有执行权限，因此一旦创建了程序，您必须授予该程序执行权限。

下面的例子创建一个计划任务的程序，名字为 program1，程序的类型为 PLSQL\_BLOCK，程序执行创建表 t2。

```
call DBMS_SCHEDULER.CREATE_PROGRAM('program1', 'PLSQL_BLOCK', 'create table t2(A int)', 'user=system
dbname=test port=54321 password=123456', 'test', 0, true, 'this is test program');
```

创建后您可以禁用或者启用该任务，后面将描述该内容。

## 18.3 删除程序

您可以通过 DBMS\_SCHEDULER 的 DROP\_PROGRAM 删除一个或多个程序。

当程序被删除时，与之相关的任何参数都会被删除，如下示例，可以在一次调用中通过逗号删除多个程序，也可以只删除一个程序。

```
call DBMS_SCHEDULER.DROP_PROGRAM('program1');
```

父主题：数据库作业调度

## 18.4 创建调度计划

您可以通过 DBMS\_SCHEDULER 的 CREATE\_SCHEDULE 创建调度计划。

调度计划是在创建该任务时的用户模式下创建的，并且第一次创建时即启动。您可以在其他用户模式下创建计划任务，一旦被创建，它就可以被其他用户使用。

下列示例创建一个名为 schedule1 的计划任务，规划在每天的早上 10 点 10 分 10 秒时。

```
call DBMS_SCHEDULER.CREATE_SCHEDULE('schedule1', now(), 'FREQ=MINUTELY;BYSECOND=10', NULL, 'this is test
schedule');
```

父主题：数据库作业调度



## 18.5 删除调度计划

您可以通过 DBMS\_SCHEDULER 的 DROP\_SCHEDULE 删除调度计划。

```
call DBMS_SCHEDULER.DROP_SCHEDULE('schedule1');
```

父主题：数据库作业调度

## 18.6 创建作业

当您完成了程序和调度计划的创建，意味着可以基于两者创建一个有计划的作业（job）。

如下示例，schedule1 定义了调度计划，program1 定义了要执行的什么内容（程序），创建的作业将两者“组合”形成一个有调度计划的作业。

```
call DBMS_SCHEDULER.CREATE_JOB('job1', 'program1', 'schedule1', 'Routine Maintenance', true, true, 'this is test job');
```

父主题：数据库作业调度

## 18.7 删除作业

当我们不再需要作业时，可以使用 DROP\_JOB 删除一个指定的作业。

如下示例，删除了一个名为 job1 的作业。

```
call DBMS_SCHEDULER.DROP_JOB('job1', false, false, 'STOP_ON_FIRST_ERROR');
```

父主题：数据库作业调度

## 18.8 运行作业

通过调用 DBMS\_SCHEDULER.RUN\_JOB，您可以使用 RUN\_JOB 程序来测试作业或者在调度计划之外运行它。

RUN\_JOB 的 use\_current\_session 确定的是作业时异步还是同步，默认为 true，目前暂时不支持。

下列示例展示了，运行一个名为 job1 的任务。

```
call DBMS_SCHEDULER.RUN_JOB('job1', true);
```

父主题：数据库作业调度

## 18.9 禁用作业

您可以通过 DBMS\_SCHEDULER 的 DISABLE 禁用作业。

作业也可以通过其他方式失去功能效果，比如，删除作业指向的程序或者调度计划将形成失去功能，调度器尝试运行时将会报错。禁用作业意味着，尽管作业的元数据存在，但作业调度器不会对其进行处理。当作业被禁用时，作业在作业表中的状态为已禁用。

当一个正在运行的作业被禁用时，并且 force（第二个参数）为 false（默认值），将返回一个错误。目前 force 暂不支持 true。如果 commit\_semantics 被设置为 STOP\_ON\_FIRST\_ERROR（默认值），将在出现第一个错误时返回调用。（该参数暂不支持其他设置）

下列示例展示了禁用一个作业。

```
call DBMS_SCHEDULER.DISABLE('job1', false, 'STOP_ON_FIRST_ERROR');
```

父主题：[数据库作业调度](#)

## 18.10 启用作业

您可以通过 DBMS\_SCHEDULER 包的 ENABLE 启用作业。

该子程序的作用是，禁用状态的作业被调度器重新捡起来进行处理，默认情况下，创建的作业时禁用状态的，因此需要启用它们才能正常工作。当作业启动时，将执行有效性检查，如果检查不通过将不会启动作业。

启用作业后会立刻根据调度计划运行作业。如果 commit\_semantics 被设置为 STOP\_ON\_FIRST\_ERROR（默认值），将在出现第一个错误时返回调用。

下列示例展示了启用一个名为 job1 的计划任务。

```
call DBMS_SCHEDULER.ENABLE('job1', 'STOP_ON_FIRST_ERROR');
```

父主题：[数据库作业调度](#)

## 18.11 任务的日历表示法

创建的任务中会涉及使用日历表示法进行时间定义，日历表示法是用来定义重复计划的实现方法，能实现一些复杂的时间计划。

下列是一些可行的日历表示法示例：

示例：

每天早上 10 点

```
FREQ=DAILY;BYHOUR=10;BYMINUTE=0;BYSECOND=0;
```

每隔 7 天的早上 10 点

```
FREQ=DAILY;INTERVAL=7;BYHOUR=10;BYMINUTE=0;BYSECOND=0;
```

每年 10 月 1 号早上 10 点

```
FREQ=YEARLY;BYMONTH=10;BYMONTHDAY=1;BYHOUR=10;BYMINUTE=0;BYSECOND=0;
```

父主题：[数据库作业调度](#)

## 18.12 计算符合规则的下一个日期

您可以通过 DBMS\_SCHEDULER 的 EVALUATE\_CALENDAR\_STRING 计算符合规则的下一次计划运行时间。计算结果保存在参数 next\_run\_date 中。

下列例子展示了，计算 'freq=yearly;interval=5;bymonth=10;bymonthday=10;byhour=10;byminute=10;bysecond=10' 这个较复杂的日历规则下一次的计划运行时间，并将其打印显示出来。

示例：

```
declare
next_run_date timestamp;
begin DBMS_SCHEDULER.EVALUATE_CALENDAR_STRING('freq=yearly;interval=5;bymonth=10;
bymonthday=10;byhour=10;byminute=10;bysecond=10',' ',' ',next_run_date);
DBMS_OUTPUT.PUT_LINE(next_run_date);
end;
```

父主题：[数据库作业调度](#)

---

# 第 19 章 管理数据库调度器

---

您可以配置、管理、监控和排查数据库调度器故障。

---

**注意：** 本章描述了如何使用 DBMS\_SCHEDULER 包管理数据库调度器。

---

- 配置调度器

KingbaseES 通过 kdb\_schedule 插件提供了 DBMS\_JOB 和 DBMS\_SCHEDULER 包，其中定义了自动作业功能的相关函数。

- 监控和管理调度程序

您可以查看有关当前运行的作业的信息，监控和管理窗口和作业日志，并管理调度器安全性。

- 导入/导出计划任务

可以使用 sys\_dump 和 sys\_restore 导入/导出计划任务。

## 19.1 配置调度器

KingbaseES 通过 kdb\_schedule 插件提供了 DBMS\_JOB 和 DBMS\_SCHEDULER 包，其中定义了自动作业功能的相关函数。在使用 kdb\_schedule 之前，我们需要将他添加到 kingbase.conf 文件的 shared\_preload\_libraries 中，并重启数据库。具体的配置方法请参阅 KingbaseES 扩展插件参考手册中的插件 kdb\_schedule 使用方法。

父主题：管理数据库调度器

## 19.2 监控和管理调度程序

您可以查看有关当前运行的作业的信息，监控和管理作业日志。

- 查找当前运行的作业信息

可以通过查询 kdb\_schedule.DBA\_SCHEDULER\_JOBS 视图来检查作业的状态。

- 监控和管理作业日志

您可以在作业日志中查看作业运行、作业状态变化和作业失败的信息。

父主题：管理数据库调度器

## 19.2.1 查找当前运行的作业信息

可以通过查询 `kdb_schedule.DBA_SCHEDULER_JOBS` 视图来检查作业的状态。

例如，执行以下语句：

```
SELECT DISTINCT job_name, state FROM kdb_schedule.dba_scheduler_jobs
WHERE job_name = 'job1';
```

```

jobname  | jastatus
-----+-----
 job1    | failed
 job1    | success
(2 rows)

```

该例子中，可以使用 `enable` 子程序启用作业。作业状态值如下表所示：`running`, `success`, `failed`, `internal failure`, `aborted`

作业状态	描述
<code>running</code>	作业当前正在运行。
<code>aborted</code>	作业运行终止了。
<code>failed</code>	作业计划运行一次，但失败了。
<code>internal failure</code>	该作业至少失败过一次，并且已计划执行重试。
<code>success</code>	作业被调度运行一次并成功完成。

你可以通过执行以下语句，来了解某个作业的相关信息：

```
SELECT * FROM kdb_job WHERE jobname='job1';
```

父主题：监控和管理调度程序

## 19.2.2 监控和管理作业日志

您可以在作业日志中查看作业运行、作业状态变化和作业失败的信息。作业日志实现如下两个数据字典视图：

- `KDB_SCHEDULE.*_SCHEDULER_JOB_LOG`

- KDB\_SCHEDULE.\*\_SCHEDULER\_JOB\_RUN\_DETAILS

你可以通过执行以下语句，来了解某个作业的日志信息：

```
SELECT * FROM kdb_schedule.all_scheduler_job_log WHERE job_name = 'job1';
SELECT * FROM kdb_schedule.all_scheduler_job_run_details WHERE job_name = 'job1';
```

父主题：[监控和管理作业日志](#)

## 19.3 导入/导出计划任务

可以使用 `sys_dump` 和 `sys_restore` 导入/导出计划任务。

`sys_dump` 和 `sys_restore` 是一种用于导入/导出 KingbaseES 数据库的工具。即使数据库正在并发使用，也能执行导出计划任务。`sys_dump` 不阻塞其他用户访问数据库（读取或写入）。当一种归档文件格式与 `sys_restore` 结合使用时，`sys_dump` 提供了一种灵活的归档和传输机制。

`sys_dump` 可用于导出整个数据库，而 `sys_restore` 可用于检查归档并/或选择要恢复数据库的哪些部分。最灵活的输出文件格式是“自定义”格式（-Fc）和“目录”格式（-Fd）。它们允许选择和重排序所有已归档项，支持并行恢复，并且默认是压缩的。“目录”格式是唯一支持并行转储的格式。

另请参阅：

《KingbaseES 客户端应用参考手册》中关于 `sys_dump` 和 `ref:'sys_restore'` 工具详细介绍和示例的描述

父主题：[管理数据库调度器](#)

---

# 第 20 章 全文搜索

---

本章节包含以下内容：

- 介绍
- 表和索引
- 控制文本搜索
- 额外特性
- 解析器
- 词典
- 配置例子
- 中文分词
- 流版式文件内容抽取
- 测试和调试文本搜索
- GIN 和 GiST 索引类型
- RUM 索引类型
- ksql 支持
- 限制

## 20.1 介绍

全文搜索是指计算机索引程序通过扫描文档中的每一个词，对每一个词建立一个索引，并指明该词在文档中出现的次数和位置，当用户查询时，检索程序就根据事先建立的索引进行查找，并将查找的结果反馈给用户的检索方式。

全文搜索提供了满足查询一个自然语言文档的能力，并可以选择将它们按照与查询的相关度排序。最常用的搜索类型是找到所有包含给定查询词的文档并按照它们与查询的相似性顺序返回它们。查询和相似性的概念非常灵活并且依赖于特定的应用。最简单的搜索认为查询是一组词而相似性是查询词在文档中的频度。

文本搜索操作符已经在数据库中存在很多年了。KingbaseES 对文本数据类型提供了 `~`、`~*`、`LIKE` 和 `ILIKE` 操作符，但是它们缺少现代信息系统所要求的很多基本属性：

- 缺乏英语语言的支持。正则表达式是不够的，因为它们不易处理派生词，例如 `satisfies` 和 `satisfy`。因此可能会错过包含 `satisfies` 的文档，尽管你可能想要在对于 `satisfy` 的搜索中找到它们。可以使用 `OR` 来搜索多个派生形式，但此操作麻烦且容易出错（有些词可能有数千种派生）。
- 不提供对搜索结果的排序（排名），这使得在面对数以千计被找到的文档时变得无效。
- 因为没有索引支持它们很慢，因此必须为每次搜索处理所有的文档。

全文索引允许文档预处理，并且保存一个索引用于以后快速的搜索。预处理包括：

- 将文档解析成记号。标识出多种类型的记号是有所帮助的，例如数字、词、复杂的词、电子邮件地址，这样它们可以被以不同的方式处理。原则上记号分类取决于相关的应用，但是对于大部分目的都可以使用一套预定义的分类。KingbaseES 使用一个解析器来执行这个步骤。其中提供了一个标准的解析器，并且为特定的需要也可以创建定制的解析器。
- 将记号转换成词位。和一个记号一样，一个词位是一个字符串，但是它已经被正规化，这样同一个词的不同形式被变成一样。例如，正规化几乎总是包括将大写字母转换成小写形式，并且经常涉及移除后缀（例如英语中的 `s` 或 `es`）。这允许搜索找到同一个词的变体形式，而不需要冗长地输入所有可能的变体。此外，这个步骤通常会消除停用词，它们是那些太普通的词，它们对于搜索是无用的（简而言之，记号是文档文本的原始片段，而词位是那些被认为对索引和搜索有用的词）。KingbaseES 使用词典来执行这个步骤。已经提供了多种标准词典，并且为特定的需要也可以创建定制的词典。
- 为搜索优化存储预处理好的文档。例如，每一个文档可以被表示为正规化的词位的一个有序数组。与词位一起，通常还想要存储用于近似排名的位置信息，这样一个包含查询词更“密集”区域的文档要比那些包含分散的查询词的文档有更高的排名。

词典允许对记号如何被正规化进行细粒度的控制。使用合适的词典，你可以：

- 定义不应该被索引的停用词。
- 使用 `Ispell` 把同义词映射到一个单一词。
- 使用一个分类词典把短语映射到一个单一词。
- 使用一个 `Ispell` 词典把一个词的不同变体映射到一种规范的形式。
- 使用 `Snowball` 词干分析器规则将一个词的不同变体映射到一种规范的形式。

KingbaseES 提供了一种数据类型 `tsvector` 来存储预处理后的文档，还提供了一种类型 `tsquery` 来表示处理过的查询（文本搜索类型）。有很多函数和操作符可以用于这些数据类型（文本搜索函数和操作符），其中最重要的是匹配操作符 `@@`，它在基本文本匹配中介绍。全文搜索可以使用索引来加速（`GIN` 和 `GiST` 索引类型）。

### 20.1.1 什么是一个文档？

文档是在一个全文搜索系统中进行搜索的单元，例如，一篇杂志文章或电子邮件消息。文本搜索引擎必须能够解析文档并存储词位（关键词）与它们的父文档之间的关联。随后，这些关联会被用来搜索包含查询词的文档。



对于 KingbaseES 中的搜索，一个文档通常是一个数据库表中一行内的一个文本形式的域，或者可能是这类域的一个组合（连接），这些域可能存储在多个表或者是动态获取。换句话说，一个文档可能从用于索引的不同部分构建，并且它可能被作为一个整体存储在某个地方。例如：

```
SELECT title || ' ' || author || ' ' || abstract || ' ' || AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

**注意：**实际上在这些例子查询中，`coalesce` 应该被用来防止一个单一 `NULL` 属性导致整个文档的一个 `NULL` 结果。

另一种存储文档的可能性是作为文件系统中的简单文本文件。在这种情况下，数据库可以被用来存储全文索引并执行搜索，并且某些唯一标识符可以被用来从文件系统检索文档。但是，从数据库的外面检索文件要求超级用户权限或者特殊函数支持，因此这种方法通常不如把所有数据放在 KingbaseES 内部方便。另外，把所有东西放在数据库内部允许方便地访问文档元数据来协助索引和现实。

对于文本搜索目的，每一个文档必须被缩减成预处理后的 `tsvector` 格式。搜索和排名被整个在一个文档的 `tsvector` 表示上执行—只有当文档被选择来显示给用户时才需要检索原始文本。我们因此经常把 `tsvector` 说成是文档，但是当然它只是完整文档的一种紧凑表示。

## 20.1.2 基本文本匹配

KingbaseES 中的全文搜索基于匹配操作符 `@@`，它在一个 `tsvector`（文档）匹配一个 `tsquery`（查询）时返回 `true`。哪种数据类型写在前面没有影响：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'
::tsquery;
?column?
-----
t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'
::tsvector;
?column?
-----
f
```

正如以上例子所建议的，一个 `tsquery` 并不只是一个未经处理的文本，顶多一个 `tsvector` 是这样。一个 `tsquery` 包含搜索术语，它们必须是已经正规化的词位，并且可以使用 `AND`、`OR`、`NOT` 以及 `FOLLOWED BY` 操作符结合多个术语（语法详见 `tsquery`）。有几个函数 `to_tsquery`、`plainto_tsquery` 以及 `phraseto_tsquery`

可用于将用户书写的文本转换为正确的 `tsquery`，它们会主要采用正则化出现在文本中的词的方法。相似地，`to_tsvector` 被用来解析和正规化一个文档字符串。因此在实际上一个文本搜索匹配可能看起来更像：

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
?column?
-----
t
```

注意如果这个匹配被写成下面这样它将不会成功：

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
?column?
-----
f
```

因为这里不会发生词 `rats` 的正规化。一个 `tsvector` 的元素是词位，它被假定为已经正规化好，因此 `rats` 不匹配 `rat`。

`@@` 操作符也支持 `text` 输出，它允许在简单情况下跳过从文本字符串到 `tsvector` 或 `tsquery` 的显式转换。可用的变体是：

```
tsvector @@ tsquery
tsquery  @@ tsvector
text     @@ tsquery
text     @@ text
```

前两种我们已经见过。形式 `text @@ tsquery` 等价于 `to_tsvector(x) @@ y`。形式 `text @@ text` 等价于 `to_tsvector(x) @@ plainto_tsquery(y)`。

在 `tsquery` 中，`&` (AND) 操作符指定它的两个参数都必须出现在文档中才表示匹配。类似地，`|` (OR) 操作符指定至少一个参数必须出现，而 `!` (NOT) 操作符指定它的参数不出现才能匹配。例如，查询 `fat & ! rat` 匹配包含 `fat` 但不包含 `rat` 的文档。

在 `<->` (FOLLOWED BY) `tsquery` 操作符的帮助下搜索可能的短语，只有该操作符的参数的匹配是相邻的并且符合给定顺序时，该操作符才算是匹配。例如：

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
?column?
-----
t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
?column?
-----
f
```

FOLLOWED BY 操作符还有一种更一般的版本，形式是 `<N>`，其中“N”是一个表示匹配词位位置之间的差。`<1>`

和<-> 相同，而<2> 允许刚好一个其他词位出现在匹配之间，以此类推。当有些词是停用词时，`phraseto_tsquery` 函数利用这个操作符来构造一个能够匹配多词短语的 `tsquery`。例如：

```
SELECT phraseto_tsquery('cats ate rats');
       phraseto_tsquery
-----
'cat' <-> 'ate' <-> 'rat'

SELECT phraseto_tsquery('the cats ate the rats');
       phraseto_tsquery
-----
'cat' <-> 'ate' <2> 'rat'
```

某些时候存在特殊情况：<0> 可以被用来要求两个匹配同一个词的模式。

圆括号可以被用来控制 `tsquery` 操作符的嵌套。如果没有圆括号，| 的计算优先级最低，然后从低到高依次是 &、<->、!。

值得注意的是，当 AND/OR/NOT 操作符在一个 FOLLOWED BY 操作符的参数中时，它们表示与不在那些参数中时不同的含义，因为在 FOLLOWED BY 中匹配的准确位置是有意义的。例如，通常 `!x` 仅匹配在任何地方都不包含 `x` 的文档。但如果 `y` 不是紧接在一个 `x` 后面，`!x <-> y` 就会匹配那个 `y`，在文档中其他位置出现的 `x` 不会阻止匹配。另一个例子是，`x & y` 通常仅要求 `x` 和 `y` 均出现在文档中的某处，但是 `(x & y) <-> z` 要求 `x` 和 `y` 在紧挨着 `z` 之前的同一个位置匹配。因此这个查询的行为会不同于 `x <-> z & y <-> z`，它将匹配一个含有两个单独序列 `x z` 以及 `y z` 的文档（这个特定的查询一点用都没有，因为 `x` 和 `y` 不可能在同一个位置匹配，但是对于前缀匹配模式之类的更复杂的情况，这种形式的查询就会有用武之地）。

### 20.1.3 配置

前述的都是简单的文本搜索例子。正如前面所提到的，全文搜索功能包括做更多事情的能力：跳过索引特定词（停用词）、处理同义词并使用更高级的解析，例如基于空白之外的解析。这个功能由文本搜索配置控制。KingbaseES 中有多种语言的预定义配置，并且你可以很容易地创建你自己的配置（`ksql` 的 `\dF` 命令显示所有可用的配置）。

在安装期间一个合适的配置将被选择并且 `default_text_search_config` 也被相应地设置在 `kingbase.conf` 中。如果你正在对整个集簇使用相同的文本搜索配置，你可以使用在 `kingbase.conf` 中使用该值。要在集簇中使用不同的配置但是在任何一个数据库内部使用同一种配置，使用 `ALTER DATABASE ... SET`。否则，可以在每个会话中设置 `default_text_search_config`。

依赖一个配置的每一个文本搜索函数都有一个可选的 `regconfig` 参数，因此要使用的配置可以被显式指定。只有当这个参数被忽略时，`default_text_search_config` 才被使用。

为了让建立自定义文本搜索配置更容易，一个配置可以从更简单的数据库对象来建立。KingbaseES 的文本搜索功能提供了四类配置相关的数据库对象：

- 文本搜索解析器将文档拆分成记号并分类每个记号（例如，作为词或者数字）。
- 文本搜索词典将记号转变成正规化的形式并拒绝停用词。

- 文本搜索模板提供位于词典底层的函数（一个词典简单地指定一个模板和一组用于模板的参数）。
- 文本搜索配置选择一个解析器和一组用于将解析器产生的记号正规化的词典。

文本搜索解析器和模板是从低层 C 函数构建而来，因此它要求 C 编程能力来开发新的解析器和模板，并且还需要超级用户权限来把它们安装到一个数据库中（在 KingbaseES 发布的 contrib/区域中有一些附加的解析器和模板的例子）。由于词典和配置只是对底层解析器和模板的参数化和连接，不需要特殊的权限来创建一个新词典或配置。创建定制词典和配置的例子将在本章稍后的部分给出。

## 20.2 表和索引

在前一节中的例子演示了使用简单常数字符串进行全文匹配。本节展示如何搜索表数据，以及可选择地使用索引。

### 20.2.1 搜索一个表

可以在没有一个索引的情况下做一次全文搜索。一个简单的查询将打印每一个行的 `title`，这些行在其 ‘ ‘ 域中包含词 `friend`：

```
SELECT title
FROM kbweb
WHERE to_tsvector('english', ) @@ to_tsquery('english', 'friend');
```

这将会找到相关的词例如 `friends` 和 `friendly`，因为这些都约减到同一个正规化的词位。

以上的查询指定要使用 `english` 配置来解析和正规化字符串。我们也可以忽略配置参数：

```
SELECT title
FROM kbweb
WHERE to_tsvector() @@ to_tsquery('friend');
```

这个查询将使用由 `default_text_search_config` 设置的配置。

一个更复杂的例子是选择 10 个最近的文档，要求它们在 `title` 或 ‘ ‘ 中包含 `create` 和 `table`：

```
SELECT title
FROM kbweb
WHERE to_tsvector(title || ' ' || ) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

为了清晰，我们忽略 `coalesce` 函数调用，它可能需要被用来查找在这两个域之中包含 `NULL` 的行。

尽管这些查询可以在没有索引的情况下工作，大部分应用会发现这种方法太慢了，除了偶尔的临时搜索。实际使用文本搜索通常要求创建一个索引。

## 20.2.2 创建索引

可以创建一个 GIN 索引（GIN 和 GiST 索引类型）来加速文本搜索：

```
CREATE INDEX kbweb_idx ON kbweb USING GIN(to_tsvector('english', ));
```

注意这里使用了 `to_tsvector` 的双参数版本。只有指定了一个配置名称的文本搜索函数可以被用在表达式索引（表达式索引）中。这是因为索引内容必须是没有被 `default_text_search_config` 影响的。如果它们被影响，索引内容可能会不一致因为不同的项可能包含被使用不同文本搜索配置创建的 `tsvector`，并且没有办法猜测哪个是哪个。也没有可能正确地转储和恢复这样的一个索引。

由于 `to_tsvector` 的双参数版本被使用在上述的索引中，只有一个使用了带有相同配置名的双参数版 `to_tsvector` 的查询引用才能使用该索引。即，`WHERE to_tsvector('english', ) @@ 'a & b'` 可以使用该索引，但 `WHERE to_tsvector() @@ 'a & b'` 不能。这保证一个索引只能和创建索引项时所用的相同配置一起使用。可以建立更复杂的表达式索引，在其中配置名被另一个列指定，例如：

```
CREATE INDEX kbweb_idx ON kbweb USING GIN(to_tsvector(config_name, ));
```

这里 `config_name` 是 `kbweb` 表中的一个列。这允许在同一个索引中有混合配置，同时记录哪个配置被用于每一个索引项。例如，如果文档集合包含不同语言的文档，这就可能会有用。同样，要使用索引的查询必须被措辞成匹配，例如 `WHERE to_tsvector(config_name, ) @@ 'a & b'`。索引甚至可以连接列：

```
CREATE INDEX kbweb_idx ON kbweb USING GIN(to_tsvector('english', title
|| ' ' || )));
```

另一种方法是创建一个单独的 `tsvector` 列来保存 `to_tsvector` 的输出。若要使此列自动与源数据保持最新，请使用已存储的生成列。这个例子是 `title` 和“”的连接，使用 `coalesce` 来保证当其他域为 `NULL` 时一个域仍然能留在索引中：

```
ALTER TABLE kbweb
  ADD COLUMN textsearchable_index_col tsvector
      GENERATED ALWAYS AS (to_tsvector('english', coalesce(title,
      '')) || ' ' || coalesce(, ''))) STORED;
, ''))) STORED;
```

然后创建一个 GIN 索引来加速搜索：

```
CREATE INDEX textsearch_idx ON kbweb USING GIN(textsearchable_index_col);
```

执行一个快速的全文搜索了：

```
SELECT title
FROM kbweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
```

```
ORDER BY last_mod_date DESC
LIMIT 10;
```

在使用一个单独的列来存储 `tsvector` 表示时，有必要创建一个触发器在 `title` 或“`改变时保证 tsvector 列为当前值。用于自动更新的触发器`”解释了怎样去做。单独列方法相对于表达式索引的一个优势在于，它不必为了利用索引而在查询中显式地指定文本搜索配置。如上述例子所示，查询可以依赖 `default_text_search_config`。另一个优势是搜索将会更快，因为它不必重做 `to_tsvector` 调用来验证索引匹配（在使用 `GiST` 索引时这一点比使用 `GIN` 索引时更重要；见 [GIN 和 GiST 索引类型](#)）。表达式索引方法更容易建立，但是它要求更少的磁盘空间，因为 `tsvector` 表示没有被显式地存储下来。

## 20.3 控制文本搜索

要实现全文搜索必须要有一个从文档创建 `tsvector` 以及从用户查询创建 `tsquery` 的函数。而且我们需要一种有用的顺序返回结果，因此我们需要一个函数能够根据文档与查询的相关性比较文档。还有一点重要的是要能够很好地显示结果。KingbaseES 对所有这些函数都提供了支持。

### 20.3.1 解析文档

KingbaseES 提供了函数 `to_tsvector` 将一个文档转换成 `tsvector` 数据类型。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

`to_tsvector` 把一个文本文档解析成记号，把记号缩减成词位，并且返回一个 `tsvector`，它列出了词位以及词位在文档中的位置。文档将根据指定的或默认的文本搜索配置来处理。下面是一个简单例子：

```
SELECT to_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');
      to_tsvector
-----
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

在上面这个例子中我们看到，作为结果的 `tsvector` 不包含词 `a`、`on` 或 `it`，词 `rats` 变成了 `rat`，并且标点符号-被忽略了。

`to_tsvector` 函数在内部调用了一个解析器，它把文档文本分解成记号并且为每一种记号分配一个类型。对于每一个记号，会去查询一个词典列表（词典），该列表会根据记号的类型而变化。第一个识别记号的词典产生一个或多个正规化的词位来表示该记号。例如，`rats` 变成 `rat` 是因为一个词典识别到该词 `rats` 是 `rat` 的复数形式。一些词会被识别为停用词（停用词），这将导致它们被忽略，因为它们出现得太频繁以至于在搜索中起不到作用。在我们的例子中有 `a`、`on` 和 `it` 是停用词。如果在列表中没有词典能识别该记号，那它也将被忽略。在这个例子中标点符号-就属于这种情况，因为事实上没有词典会给它分配记号类型（空间符号），即空间记号不会被索引。对于解析器、词典以及要索引哪些记号类型是由所选择的文本搜索配置（[配置例子](#)）决定的。可以在同一个数据库中有多种不同的配置，并且有用于很多种语言的预定义配置。在我们的例子中，我们使用用于英语的默认配置 `english`。

函数 `setweight` 可以被用来对 `tsvector` 中的项标注一个给定的权重，这里一个权重可以是四个字母之一：A、B、C 或 D。这通常被用来标记来自文档不同部分的项，例如标题对正文。稍后，这种信息可以被用来排名搜索结果。

因为 `to_tsvector(NULL)` 将返回 `NULL`，不论何时一个域可能为空时，我们推荐使用 `coalesce`。下面是我们推荐的从一个结构化文档创建一个 `tsvector` 的方法：

```
UPDATE tt SET ti =
  setweight(to_tsvector(coalesce(title,'')), 'A') ||
  setweight(to_tsvector(coalesce(keyword,'')), 'B') ||
  setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
  setweight(to_tsvector(coalesce(''), 'D');
```

这里我们已经使用了 `setweight` 在完成的 `tsvector` 标注每一个词位的来源，并且接着将标注过的 `tsvector` 值用 `tsvector` 连接操作符 `||` 合并在一起（[操纵文档](#)给出了关于这些操作符的细节）。

## 20.3.2 解析查询

KingbaseES 提供了函数 `to_tsquery`、`plainto_tsquery`、`phraseto_tsquery` 以及 `websearch_to_tsquery` 用来把一个查询转换成 `tsquery` 数据类型。`to_tsquery` 提供了比 `plainto_tsquery` 和 `phraseto_tsquery` 更多的特性，但是它对其输入要求更加严格。`websearch_to_tsquery` 是 `to_tsquery` 的一个简化版本，它使用一种可选择的语法，类似于 Web 搜索引擎使用的语法。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`to_tsquery` 从 “querytext” 创建一个 `tsquery` 值，该值由被 `tsquery` 操作符 `&` (AND)、`|` (OR)、`!` (NOT) 和 `<->` (FOLLOWED BY) 分隔的单个记号组成。这些操作符可以使用圆括号分组。换句话说，`to_tsquery` 的输入必须已经遵循 `tsquery` 输入的一般规则，如 `tsquery` 所述。区别在于基本的 `tsquery` 输入把记号当作表面值，而 `to_tsquery` 会使用指定的或者默认的配置把每一个记号正规化成一个词位，并且丢弃掉任何根据配置是停用词的记号。例如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
   to_tsquery
-----
'fat' & 'rat'
```

和在基本 `tsquery` 输入中一样，权重可以被附加到每一个词位来限制它只匹配属于那些权重的 `tsvector` 词位。例如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
   to_tsquery
-----
'fat' | 'rat':AB
```

同样，`*` 可以被附加到一个词位来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
       to_tsquery
-----
'supern':*A & 'star':*AB
```

这样一个词位将匹配一个 `tsvector` 中的任意以给定字符串开头的词。

`to_tsquery` 也能够接受单引号短语。当配置包括一个会在这种短语上触发的分类词典时就是它的主要用处。在下面的例子中，一个分类词典含规则 `supernovae stars : sn:`

```
SELECT to_tsquery('supernovae stars' & !'crab');
       to_tsquery
-----
'sn' & !'crab'
```

在没有引号时，`to_tsquery` 将为那些没有被 AND、OR 或者 FOLLOWED BY 操作符分隔的记号产生一个语法错误。

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`plainto_tsquery` 将未格式化的文本 “querytext” 转换成一个 `tsquery` 值。该文本被解析并被正规化，很像 `to_tsvector`，然后 `&` (AND) 布尔操作符被插入到留下来的词之间。

例子：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
       plainto_tsquery
-----
'fat' & 'rat'
```

注意 `plainto_tsquery` 不会识其输入中的 `tsquery` 操作符、权重标签或前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
       plainto_tsquery
-----
'fat' & 'rat' & 'c'
```

这里，所有输入的标点都被作为空间符号并且丢弃。

```
phraseto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`phraseto_tsquery` 的行为很像 `plainto_tsquery`，不过前者会在留下来的词之间插入 `<->` (FOLLOWED BY) 操作符而不是 `&` (AND) 操作符。还有，停用词也不是简单地丢弃掉，而是通过插入 `<N>` 操作符（而不是 `<->` 操作符）来解释。在搜索准确的词位序列时这个函数很有用，因为 FOLLOWED BY 操作符不只是检查所有词位的存在性，还会检查词位的顺序。

例子：



```
SELECT phraseto_tsquery('english', 'The Fat Rats');
      phraseto_tsquery
-----
'fat' <-> 'rat'
```

和 `plainto_tsquery` 相似, `phraseto_tsquery` 函数不会识别其输入中的 `tsquery` 操作符、权重标签或者前缀匹配标签:

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
      phraseto_tsquery
-----
'fat' <-> 'rat' <-> 'c'
```

```
websearch_to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`websearch_to_tsquery` 使用一种可供选择的语法从“querytext”创建一个 `tsquery` 值, 这种语法中简单的未格式化文本是一个有效的查询。和 `plainto_tsquery` 以及 `phraseto_tsquery` 不同, 它还识别特定的操作符。此外, 这个函数绝不会报出语法错误, 这就可以把原始的用户提供的输入用于搜索。支持下列语法:

- 无引号文本: 不在引号中的文本将被转换成由 `&` 操作符分隔的词, 就像被 `plainto_tsquery` 处理过那样。
- " 引号文本": 在引号中的文本将被转换成由 `<->` 操作符分隔的词, 就像被 `phraseto_tsquery` 处理过那样。
- OR: 逻辑或将被转换成 `|` 操作符。
- -: 逻辑非操作符, 被转换成 `!` 操作符。

示例:

```
SELECT websearch_to_tsquery('english', 'The fat rats');
      websearch_to_tsquery
-----
'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
      websearch_to_tsquery
-----
'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
      websearch_to_tsquery
-----
'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)
```

```

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
       websearch_to_tsquery
-----
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', '""')( dummy \ query <->');
       websearch_to_tsquery
-----
 'dummi' & 'queri'
(1 row)

```

### 20.3.3 排名搜索结果

排名处理尝试度量文档和一个特定查询的接近程度，这样当有很多匹配时最相关的那些可以被先显示。KingbaseES 提供了两种预定义的排名函数，它们考虑词法、临近性和结构信息；即，它们考虑查询词在文档中出现得有多频繁，文档中的词有多接近，以及词出现的文档部分有多重要。不过，相关性的概念是模糊的并且与应用非常相关。不同的应用可能要求额外的信息用于排名，例如，文档修改时间。内建的排名函数只是例子。你可以编写你自己的排名函数和/或把它们的结果与附加因素整合在一起来适应你的特定需求。

目前可用的两种排名函数是：

```

ts_rank([weights float4[],]*vector* tsvector, *query* tsquery[, normalization integer])
returns float4

```

基于向量的匹配词位的频率来排名向量。

```

ts_rank_cd([weights float4[],]*vector* tsvector, *query* tsquery[, normalization integer])
returns float4

```

这个函数为给定文档向量和查询计算覆盖密度排名，该方法在 Clarke、Cormack 和 Tudhope 于 1999 年在期刊“Information Processing and Management”上的文章“Relevance Ranking for One to Three Term Queries”文章中有描述。覆盖密度类似于 `ts_rank` 排名，不过它会考虑匹配词位相互之间的接近度。

这个函数要求词位的位置信息来执行其计算。因此它会忽略 `tsvector` 中任何“被剥离的”词位。如果在输入中有未被剥离的词位，结果将会是零（`strip` 函数和 `tsvector` 中的位置信息的更多内容请见[操纵文档](#)）。

对这两个函数，可选的“权重”参数提供了为词实例赋予更多或更少权重的能力，这种能力是依据它们被标注的情况的。权重数组指定每一类词应该得到多重的权重，按照如下的顺序：

```
{D-权重, C-权重, B-权重, A-权重}
```

如果没有提供“权重”，那么将使用这些默认值：

```
{0.1, 0.2, 0.4, 1.0}
```

通常权重被用来标记来自文档特别区域的词，如标题或一个初始的摘要，这样它们可以被认为比来自文档正文的词更重要或更不重要。

由于一个较长的文档有更多的机会包含一个查询术语，因此考虑文档的尺寸是合理的，例如一个一百个词的文档中有一个搜索词的五个实例而零一个一千个词的文档中有该搜索词的五个实例，则前者比后者更相关。两种排名函数都采用一个整数“正规化”选项，它指定文档长度是否影响其排名以及如何影响。该整数选项控制多个行为，因此它是一个位掩码：你可以使用 | 指定一个或多个行为（例如，2|4）。

- 0（默认值）忽略文档长度
- 1 用  $1 + \text{文档长度}$  的对数除排名
- 2 用文档长度除排名
- 4 用长度之间的平均调和距离除排名（只被 `ts_rank_cd` 实现）
- 8 用文档中唯一词的数量除排名
- 16 用  $1 + \text{文档中唯一词数量}$  的对数除排名
- 32 用排名 + 1 除排名

如果多于一个标志位被指定，转换将根据列出的顺序被应用。

值得注意的是排名函数并不使用任何全局信息，因此它不可能按照某些时候期望地产生一个公平的正规化，从 1% 或 100%。正规化选项 32 ( $\text{rank}/(\text{rank}+1)$ ) 可以被应用来缩放所有的排名到范围零到一，但是当然这只是一个外观上的改变；它不会影响搜索结果的顺序。

这里是一个例子，它只选择十个最高排名的匹配：

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

title	rank
Neutrinos in the Sun	3.1
The Sudbury Neutrino Detector	2.4
A MACHO View of Galactic Dark Matter	2.01317
Hot Gas and Dark Matter	1.91171
The Virgo Cluster: Hot Plasma and Dark Matter	1.90953
Rafting for Solar Neutrinos	1.9
NGC 4650A: Strange Galaxy and Dark Matter	1.85774
Hot Gas and Dark Matter	1.6123
Ice Fishing for Cosmic Neutrinos	1.6
Weak Lensing Distorts the Universe	0.818218

这是相同的例子使用正规化的排名：

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
```

title	rank
Neutrinos in the Sun	0.756097569485493
The Sudbury Neutrino Detector	0.705882361190954
A MACHO View of Galactic Dark Matter	0.668123210574724
Hot Gas and Dark Matter	0.65655958650282
The Virgo Cluster: Hot Plasma and Dark Matter	0.656301290640973
Rafting for Solar Neutrinos	0.655172410958162
NGC 4650A: Strange Galaxy and Dark Matter	0.650072921219637
Hot Gas and Dark Matter	0.617195790024749
Ice Fishing for Cosmic Neutrinos	0.615384618911517
Weak Lensing Distorts the Universe	0.450010798361481

排名可能会非常昂贵，因为它要求查询每一个匹配文档的 `tsvector`，这可能会涉及很多 I/O 因而很慢。不幸的是，这几乎不可避免，因为实际查询常常导致巨大数目的匹配。

## 20.3.4 加亮结果

要表示搜索结果，理想的方式是显示每一个文档的一个部分并且显示它是怎样与查询相关的。通常，搜索引擎显示文档片段时会对其中的搜索术语进行标记。KingbaseES 提供了一个函数 `ts_headline` 来实现这个功能。

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

`ts_headline` 接受一个文档和一个查询，并且从该文档返回一个引用，在其中来自查询的术语会被加亮。被用来解析该文档的配置可以用“`config`”指定；如果“`config`”被忽略，将会使用 `default_text_search_config` 配置。

如果一个“`options`”字符串被指定，它必须由一个逗号分隔的列表组成，列表中是一个或多个“`option`”=“`value`”对。可用的选项是：

- **StartSel**、**StopSel**：用来定界文档中出现的查询词的字符串，这用来把它们与其他被引用的词区分开。如果这些字符串包含空格或逗号，你必须把它们加上双引号。
- **MaxWords**、**MinWords**：这些数字决定要输出的最长和最短 headline。
- **ShortWord**：长度小于等于这个值的词将被从一个 headline 的开头或结尾处丢掉。默认值三消除普通英语文章。
- **HighlightAll**：布尔标志，如果为 `true` 整个文档将被用作 headline，并忽略前面的三个参数。
- **MaxFragments**：要显示的文本引用或片段的最大数量。默认值零选择一种非片段倾向的 headline 生成方法。一

个大于零的值选择基于片段的 headline 生成。这种方法找到有尽可能多查询词的文本片段并且展开查询词周围的那些片段。结果是查询词会靠近每个片段的中间并且在两侧都有词。每一个片段将是最多 MaxWords 并且长度小于等于 ShortWord 的词被从每个片段的开头或结尾丢弃。如果不是所有的查询词都在该文档中找到，文档中第一个 MinWords 的单一片段将被显示。

- **FragmentDelimiter**: 当多于一个片段被显示时，片段将被这个字符串所分隔。

这些选项名称不区分大小写。任何未指定的选项将收到这些默认值：

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

例如：

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
           ts_headline
-----
containing given <b>query</b> terms
and return them in order of their <b>similarity</b> to the
<b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
           ts_headline
-----
containing given <query> terms
and return them in order of their <similarity> to the
<query>.
```

ts\_headline 使用原始文档，而不是一个 tsvector 摘要，因此它可能很慢并且应该被小心使用。

## 20.4 额外特性

这一节描述在文本搜索中的一些额外的函数和操作符。

## 20.4.1 操纵文档

**解析文档** 展示了未经处理的文本文档如何被转换成 `tsvector` 值。KingbaseES 也提供了用于操纵已经为 `tsvector` 形式的文档的函数和操作符。

`tsvector || tsvector tsvector` 连接操作符返回一个向量，它结合了作为参数给出的两个向量的词位和位置信息。位置和权重标签在连接期间被保留。出现在右手向量中的位置被使用左手向量中提到的最大位置进行偏移，这样结果几乎等于在两个原始文档字符串的连接上执行 `to_tsvector` 的结果（这种等价不是完全的，因为从左手参数的尾端移除的任何停用词将会影响结果，而如果文本连接被使用它们就影响了右手参数中的词位位置）。

使用向量形式的连接而不是在应用 `to_tsvector` 之前连接文本的一个优点是你可以使用不同配置来解析文档的不同小节。此外，因为 `setweight` 函数按照相同的方式标记给定向量的所有词位，如果你想把文档的不同部分标注不同的权重，你就有必要解析文本并且在连接之前做 `setweight`。

`setweight(vector tsvector, weight "char") returns tsvector setweight` 返回输入向量的一个拷贝，其中每一个位置都被标注为给定的“权重”：A、B、C 或 D（D 是新向量的默认值并且并不会被显示在输出上）。向量被连接时会保留这些标签，允许来自文档的不同部分的词被排名函数给予不同的权重。

注意权重标签是应用到位置而不是词位。如果输入向量已经被剥离了位置，则 `setweight` 什么也不会做。

`length(vector tsvector) returns integer` 返回存储在向量中的词位数。

`strip(vector tsvector) returns tsvector` 返回一个向量，其中列出了和给定向量相同的词位，不过没有任何位置或者权重信息。其结果通常比未被剥离的向量小很多，但是用处也小很多。和未被剥离的向量一样，相关度排名在已剥离的向量上也不起作用。此外，`<->`（FOLLOWED BY）`tsquery` 操作符不会匹配已剥离的输入，因为它无法确定词位之间的距离。

## 20.4.2 操纵查询

**解析查询** 展示了未经处理的文本形式的查询如何被转换成 `tsquery` 值。KingbaseES 也提供了用于操纵已经是 `tsquery` 形式的查询的函数和操作符。

`tsquery && tsquery` 返回用 AND 结合的两个给定查询。

`tsquery || tsquery` 返回用 OR 结合的两个给定查询。

`!! tsquery` 返回一个给定查询的反（NOT）。

`tsquery <-> tsquery` 返回一个查询，它用 `<->`（FOLLOWED BY）`tsquery` 操作符搜索两个紧跟的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
       ?column?
-----
'fat' <-> 'cat' | 'fat' <-> 'rat'
```

`tsquery_phrase(query1 tsquery, query2 tsquery [, `` ``distance integer]) returns tsquery` 返回一个查询，它使用 `<N>` `tsquery` 操作符搜索两个距离为“distance”个词位的匹配，第一个匹配符合第一个给定的查询

而第二个匹配符合第二个给定的查询。例如：

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
   tsquery_phrase
-----
'fat' <10> 'cat'
```

`numnode(query tsquery ``)` returns “integer 返回一个 `tsquery` 中的结点数（词位外加操作符）。要确定“查询是否有意义或者是否只包含停用词时，这个函数有用，在前一种情况它返回  $> 0$ ，后一种情况返回 0。例子：

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE: query contains only stopword(s) or doesn't contain lexeme(s),
ignored
   numnode
-----
          0

SELECT numnode('foo & bar'::tsquery);
   numnode
-----
          3
```

`querytree(query tsquery ``)` returns “text 返回一个 `tsquery` 中可以被用来搜索一个索引的部分。这个函数用来检测不可被索引的查询，例如那些只包含停用词或者只有否定术语的查询。例如：

```
SELECT querytree(to_tsquery('!defined'));
   querytree
-----
```

### 20.4.2.1 查询重写

`ts_rewrite` 函数族在一个给定的 `tsquery` 中搜索一个目标子查询的出现，并且将每一次出现替换成一个替补子查询。本质上这个操作就是一个 `tsquery` 版本的子串替换。一个目标和替补的组合可以被看成是一个查询重写规则。一个这类重写规则的集合可以是一个强大的搜索助手。例如，你可以使用同义词扩展搜索（如，`new york`、`big apple`、`nyc`、`gotham`），或者收缩搜索来将用户导向某些特点主题。在这个特性和分类词典（分类词典）有些功能重叠。但是，你可以随时修改一组重写规则而无需重索引，而更新一个分类词典则要求进行重索引才能生效。

`ts_rewrite (query tsquery, target tsquery, substitute tsquery)` returns `tsquery` 这种形式的 `ts_rewrite` 简单地应用一个单一重写规则：不管“target“出现在“query“中的那个地方，它都被“substitute“替代。例如：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
   ts_rewrite
-----
'b' & 'c'
```

`ts_rewrite (query tsquery, select text ) returns tsquery` 这种形式的 `ts_rewrite` 接受一个开始 “query” 和一个 SQL “select” 命令，它们以一个文本字符串的形式给出。“select” 必须得到 `tsquery` 类型的两列。对于 “select” 结果的每一行，在当前 “query” 值中出现的第一列值（目标）被第二列值（替补）所替换。例如：

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
      ts_rewrite
-----
'b' & 'c'
```

注意当多个重写规则被以这种方式应用时，应用的顺序很重要；因此在实际中你会要求源查询按某些排序键 `ORDER BY`。

考虑一个现实的天文学例子。将使用表驱动的重写规则扩展查询 `supernovae`：

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
      ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' )
```

可以通过只更新表来改变重写规则：

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
      ts_rewrite
-----
'crab' & ( 'supernova' | 'sn' & '!nebulae' )
```

当有很多重写规则时，重写可能会很慢，因为它要为每一个可能的匹配检查每一条规则。要过滤掉明显不符合的规则，可为 `tsquery` 类型使用包含操作符。在下面的例子中，只选择那些可能匹配原始查询的规则：

```
SELECT ts_rewrite('a & b'::tsquery,
                'SELECT t,s FROM aliases WHERE 'a & b'::tsquery @> t');
      ts_rewrite
-----
'b' & 'c'
```





的不同—就需要编写一个自定义触发器。下面是用 PL/SQL 作为触发器语言的一个例子：

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
    new.tsv :=
        setweight(to_tsvector('sys_catalog.english', coalesce(new.title,'')), 'A') ||
        setweight(to_tsvector('sys_catalog.english', coalesce(new.,'')), 'D');
    return new;
end
$$ LANGUAGE plsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

记住当在触发器内创建 `tsvector` 值时，显式地指定配置名非常重要，这样列的内容才不会因为 `default_text_search_config` 的改变所影响。如果不这样做很可能导致问题，例如在转储并重新载入之后搜索结果改变。

## 20.4.4 收集文档统计数据

`ts_stat` 被用于检查你的配置以及寻找候选的停用词。

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

`sqlquery` 是一个文本值，它包含一个必须返回单一 `tsvector` 列的 SQL 查询。`ts_stat` 执行该查询并返回有关包含在该 `tsvector` 数据中的每一个可区分词位（词）的统计数据。返回的列是：

- `word text` — 一个词位的值
- `ndoc integer` — 词出现过的文档（`tsvector`）的数量
- `nentry integer` — 词出现的总次数

如果提供了权重，只有具有其中之一权重的出现才被计算在内。

例如，要在一个文档集合中查找十个最频繁的词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同样的要求，但是只计算以权重 A 或 B 出现的次数：

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

## 20.5 解析器

文本搜索解析器负责把未处理的文档文本划分成记号并且标识每一个记号的类型，而可能的类型集合由解析器本身定义。注意一个解析器完全不会修改文本—它简单地标识看似有理的词边界。因为这种有限的视野，对于应用相关的自定义解析器的需求就没有自定义字典那么强烈。目前 KingbaseES 只提供了一种内建解析器，它已经被证实对很多种应用都适用。

内建解析器被称为 `sys_catalog.default`。它识别 23 种记号类型，如下表所示。

表 20.5.1: 默认解析器的记号类型

别名	描述	例子
asciiword	单词, 所有 ASCII 字母	elephant
word	单词, 所有字母	ma?ana
numword	单词, 字母和数字	beta1
asciihword	带连字符的单词, 所有 ASCII	up-to-date
hword	带连字符的单词, 所有字母	lógico-matemática
numhword	带连字符的单词, 字母和数字	kingbase-beta1
hword_asciipart	带连字符的单词部分, 所有 ASCII	kingbase in the context kingbase-beta1
hword_part	带连字符的单词部分, 所有字母	lógico or matemática in the context lógico-matemática
hword_numpart	带连字符的单词部分, 字母和数字	beta1 in the context kingbase-beta1
email	Email 地址	foo@example.com
protocol	协议头部	http://
url	URL	example.com/stuff/index.html
host	主机	example.com
url_path	URL 路径	/stuff/index.html , in the context of a URL
file	文件或路径名	/usr/local/foo.txt, if not within a URL
sfloat	科学记数法	-1.234e56
float	十进制记数法	-1.234
int	有符号整数	-1234
uint	无符号整数	1234
version	版本号	8.3.0
tag	XML 标签	
entity	XML 实体	&amp;
blank	空格符号	(其他不识别的任意空白或标点符号)

**注意：**解析器的一个“字母”的概念由数据库的区域设置决定，具体是 `lc_ctype`。只包含基本 ASCII 字母的词被报告为一个单独的记号类型，因为有时可以用来区别它们。在大部分欧洲语言中，记号类型 `word` 和 `asciiword` 应该被同样对待。

`email` 不支持 RFC 5322 定义的所有合法 email 字符。具体来说，对 email 用户名被支持的非字母数字字符只有句点、破折号和下划线。

解析器有可能从同一份文本得出相互覆盖的记号。例如，一个带连字符的词可能会被报告为一整个词或者多个部分：

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
 alias      | description                               | token
-----+-----+-----
 numhword   | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_ascii | Hyphenated word part, all ASCII          | foo
 blank      | Space symbols                            | -
 hword_ascii | Hyphenated word part, all ASCII          | bar
 blank      | Space symbols                            | -
 hword_num  | Hyphenated word part, letters and digits | beta1
```

这种行为是值得要的，因为它允许对整个复合词和每个部分进行搜索。这里是另一个例子：

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/
index.html');
 alias      | description | token
-----+-----+-----
 protocol  | Protocol head | http://
 url       | URL          | example.com/stuff/index.html
 host      | Host         | example.com
 url_path  | URL path     | /stuff/index.html
```

## 20.6 词典

词典被用来消除不被搜索考虑的词（stop words）、并被用来正规化词这样同一个词的不同派生形式将会匹配。一个被成功地正规化的词被称为一个词位。除了提高搜索质量，正规化和移除停用词减小了文档的 `tsvector` 表示的尺寸，因而提高了性能。正规化不会总是有语言上的意义并且通常依赖于应用的语义。

一些正规化的例子：

- 语言学的 - Ispell 词典尝试将输入词缩减为一种正规化的形式；词干分析器词典移除词的结尾
- URL 位置可以被规范化来得到等效的 URL 匹配：

– <http://www.kingbase.ru/db/mw/index.html>

- <http://www.kingbase.ru/db/mw/>
- <http://www.kingbase.ru/db/./db/mw/index.html>

- 颜色名可以被它们的十六进制值替换，例如 `red`, `green`, `blue`, `magenta` -> `FF0000`, `00FF00`, `0000FF`, `FF00FF`
- 如果索引数字，可以移除某些小数位来缩减可能的数字的范围，因此如果只保留小数点后两位，例如 `3.14159265359`、`3.1415926`、`3.14` 在正规化后会变得相同。

一个词典是一个程序，它接受一个记号作为输入，并返回：

- 如果输入的记号对词典是已知的，则返回一个词位数组（注意一个记号可能产生多于一个词位）
- 一个 `TSL_FILTER` 标志被设置的单一词位，用一个新记号来替换要被传递给后续字典的原始记号（做这件事的一个字典被称为一个过滤字典）
- 如果字典知道该记号但它是一个停用词，则返回一个空数组
- 如果字典不识别该输入记号，则返回 `NULL`

KingbaseES 为许多语言提供了预定义的字典。也有多种预定义模板可以被用于创建带自定义参数的新词典。每一种预定义词典模板在下面描述。如果没有合适的现有模板，可以创建新的；例子见 KingbaseES 发布的 `contrib/` 区域。

一个文本搜索配置把一个解析器和一组处理解析器输出记号的词典绑定在一起。对于每一中解析器能返回的记号类型，配置都指定了一个单独的词典列表。当该类型的一个记号被解析器找到时，每一个词典都被按照顺序查询，知道某个词典将其识别为一个已知词。如果它被标识为一个停用词或者没有一个词典识别它，它将被丢弃并且不会被索引和用于搜索。通常，第一个返回非 `NULL` 输出的词典决定结果，并且任何剩下的词典都不会被查找；但是一个过滤词典可以将给定词替换为一个被修改的词，它再被传递给后续的词典。

配置一个词典列表的通用规则是将最狭窄、最特定的词典放在第一位，然后是更加通用的词典，以一个非常通用的词典结尾，像一个 `Snowball` 词干分析器或什么都识别的 `simple`。例如，对于一个天文学相关的搜索（`astro_en` 配置）我们可以把记号类型 `asciiword`（ASCII 词）绑定到一个天文学术语的分类词典、一个通用英语词典和一个 `Snowball` 英语词干分析器：

```
ALTER TEXT SEARCH CONFIGURATION astro_en
ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

一个过滤词典可以被放置在列表中的任意位置，除了在最后，因为过滤词典放在最后就等于无用。过滤词典可用于部分正规化词来简化后续词典的工作。例如，一个过滤词典可以被用来从音标字母中移除重音符号。

## 20.6.1 停用词

停用词是非常常用、在几乎每一个文档中出现并且没有任何区分度的词。因此，在全文搜索的环境中它们可以被忽略。例如，每一段英语文本都包含 `a` 和 `the` 等次，因此把它们存储在一个索引中是没有用处的。但是，停用词确实会影响在 `tsvector` 中的位置，这进而会影响排名：

```
SELECT to_tsvector('english','in the list of stop words');
       to_tsvector
-----
'list':3 'stop':5 'word':6
```

缺失的位置 1、2、4 是因为停用词。文档的排名计算在使用和不使用停用词的情况下是很不同的：

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'),
to_tsquery('list & stop'));
       ts_rank_cd
-----
          0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'),
to_tsquery('list & stop'));
       ts_rank_cd
-----
          0.1
```

如何对待停用词是由指定词典决定的。例如，`ispell` 词典首先正规化词并且查看停用词列表，而 `Snowball` 词干分析器首先检查停用词的列表。这种不同行为的原因是一冲降低噪声的尝试。

## 20.6.2 简单词典

`simple` 词典模板的操作是将输入记号转换为小写形式并且根据一个停用词文件检查它。如果该记号在该文件中被找到，则返回一个空数组，导致该记号被丢弃。否则，该词的小写形式被返回作为正规化的词位。作为一种选择，该词典可以被配置为将非停用词报告为未识别，允许它们被传递给列表中的下一个词典。

下面是一个使用 `simple` 模板的词典定义的例子：

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = sys_catalog.simple,
    STOPWORDS = english
);
```

这里，`english` 是一个停用词文件的基本名称。该文件的全名将是 `$$SHAREDIR/tsearch_data/english.stop`，其中 `$$SHAREDIR` 表示 KingbaseES 安装的共享数据目录，通常是 `/usr/local/share/kingbase`（如果不确定，使用 `sys_config --sharedir`）。该文件格式是一个词的列表，每行一个。空行和尾部的空格都被忽略，并且大写也被折叠成小写，但是没有其他对该文件内容的处理。

测试词典：

```
SELECT ts_lexize('public.simple_dict','YeS');
       ts_lexize
-----
```

```
{yes}

SELECT ts_lexize('public.simple_dict','The');
   ts_lexize
-----
   {}
```

如果没有在停用词文件中找到，可以选择返回 `NULL` 而不是小写形式的词。这种行为可以通过设置词典的 `Accept` 参数为 `false` 来选择。继续该例子：

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict','Yes');
   ts_lexize
-----

SELECT ts_lexize('public.simple_dict','The');
   ts_lexize
-----
   {}
```

在使用默认值 `Accept = true`，只有把一个 `simple` 词典放在词典列表的尾部才有用，因为它将不会传递任何记号给后续的词典。相反，`Accept = false` 只有当至少有一个后续词典的情况下才有用。

**Caution:** 大部分类型的词典依赖于配置文件，例如停用词文件。这些文件必须被存储为 UTF-8 编码。当它们被读入服务器时，如果存在不同，它们将被翻译成真实的数据库编码。

**Caution:** 通常，当一个词典配置文件第一次在数据库会话中使用时，数据库会话将只读取它一次。如果你修改了一个配置文件并且想强迫现有的会话取得新内容，可以在该词典上发出一个 `ALTER TEXT SEARCH DICTIONARY` 命令。这可以是一次“假”更新，它并不实际修改任何参数值。

### 20.6.3 同义词词典

这个词典模板被用来创建用于同义词替换的词典。不支持短语（使用分类词典模板（[分类词典](#)）可以支持）。一个同义词词典可以被用来解决语言学问题，例如，阻止一个英语词干分析器词典把词“Paris”缩减成“pari”。在同义词词典中有一行 `Paris paris` 并把它放在 `english_stem` 词典之前就足够了。例如：

```
SELECT * FROM ts_debug('english', 'Paris');
   alias | description | token | dictionaries | dictionary | lexemes
```



```

-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
alias      | description | token|          dictionaries          | dictionary|lexemes
-----+-----+-----+-----+-----+-----
asciiword|Word,all ASCII|Paris|{my_synonym,english_stem}|my_synonym|{paris}

```

synonym 模板要求的唯一参数是 SYNONYMS，它是其配置文件的基本名——上例中的 my\_synonyms。该文件的完整名称将是 \$SHAREDIR/tsearch\_data/my\_synonyms.syn（其中 \$SHAREDIR 表示 KingbaseES 安装的共享数据目录）。该文件格式是每行一个要被替换的词，后面跟着它的同义词，用空白分隔。空行和结尾的空格会被忽略。

synonym 模板还有一个可选的参数 CaseSensitive，其默认值为 false。当 CaseSensitive 为 false 时，同义词文件中的词被折叠成小写，这和输入记号一样。当它为 true 时，词和记号将不会被折叠成小写，但是比较时就好像被折叠过一样。

一个星号 (\*) 可以被放置在配置文件中一个同义词的末尾。这表示该同义词是一个前缀。当项被用在 to\_tsvector() 中时，星号会被忽略；当它被用在 to\_tsquery() 中时，结果将是一个带有前缀匹配标记器（见解析查询）的查询项。例如，假设在 \$SHAREDIR/tsearch\_data/synonym\_sample.syn 中有这些项：

```

kingbase      kingbase
kingbase      kingbase
kingbase kingbase
google        googl
indices index*

```

那么将得到这些结果：

```

mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms=
'synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);

```

```

mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-----
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
 to_tsquery
-----
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
          tsvector
-----
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
 ?column?
-----
 t
(1 row)

```

## 20.6.4 分类词典

一个分类词典（有时被简写成 TZ）是一个词的集合，其中包括了词与短语之间的联系，即广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。

基本上一个分类词典会用一个首选词替换所有非首选词，并且也可选择地保留原始术语用于索引。KingbaseES 的分类词典的当前实现是同义词词典的一个扩展，并增加了短语支持。一个分类词典要求一个下列格式的配置文件：

```

# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...

```

其中冒号（:）符号扮演了一个短语及其替换之间的定界符。

一个分类词典使用一个子词典（在词典的配置中指定）在检查短语匹配之前正规化输入文本。只能选择一个子词典。如果子词典无法识别一个词，将报告一个错误。在这种情况下，你应该移除该词的使用或者让子词典学会这个词。你可以在一个被索引词的开头放上一个星号（\*）来跳过在其上应用子词典，但是所有采样词必须被子词典知道。

如果有多个短语匹配输入，则分类词典选择最长的那一个，并且使用最后的定义打破连结。

由于子词典识别的特定停用词不能够被指定；改用? 标记任何可以出现停用词的地方。例如，假定根据子词典 `a` 和 `the` 是停用词：

```
? one ? two : swsw
```

匹配 `a one the two` 和 `the one a two`；两者都将被 `swsw` 替换。

由于一个分类词典具有识别短语的能力，它必须记住它的状态并与解析器交互。一个分类词典使用这些任务来检查它是否应当处理下一个词或者停止累积。分类词典必须被小心地配置。例如，如果分类词典被分配只处理 `asciiword` 记号，则一个形如 `one 7` 的分类词典定义将不会工作，因为记号类型 `uint` 没有被分配给该分类词典。

**Caution:** 在索引期间要用到分类词典，因此分类词典参数中的任何变化都要求重索引。对于大多数其他索引类型，例如增加或移除停用词等小改动都不会强制重索引。

### 20.6.4.1 分类词典配置

要定义一个新的分类词典，可使用 `thesaurus` 模板。例如：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
  TEMPLATE = thesaurus,
  DictFile = mythesaurus,
  Dictionary = sys_catalog.english_stem
);
```

这里：

- `thesaurus_simple` 是新词典的名称
- `mythesaurus` 是分类词典配置文件的基础名称（它的全名将是 `$SHAREDIR/tsearch_data/mythesaurus.ths`，其中 `$SHAREDIR` 表示安装的共享数据目录）。
- `sys_catalog.english_stem` 是要用于分类词典正规化的子词典（这里是一个 Snowball 英语词干分析器）。注意子词典将拥有它自己的配置（例如停用词），但这里没有展示。

现在可以在配置中把分类词典 `thesaurus_simple` 绑定到想要的记号类型上，例如：

```
ALTER TEXT SEARCH CONFIGURATION russian
  ALTER MAPPING FOR asciiword, asciihword, hword_asciiword
  WITH thesaurus_simple;
```

### 20.6.4.2 分类词典例子

考虑一个简单的天文学分类词典 `thesaurus_astro`，它包含一些天文学词组合：

```
supernovae stars : sn
crab nebulae : crab
```

创建一个词典并绑定一些记号类型到一个天文学分类词典以及英语词干分析器：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

`ts_lexize` 对于测试一个分类词典用处不大，因为它将输入看成是一个单一记号。可以用 `plainto_tsquery` 和 `to_tsvector`，它们将把其输入字符串打断成多个记号：

```
SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----
'sn'

SELECT to_tsvector('supernova star');
to_tsvector
-----
'sn':1
```

原则上，如果对参数加了引号，可以使用 `to_tsquery`：

```
SELECT to_tsquery(''supernova star'');
to_tsquery
-----
'sn'
```

注意在 `thesaurus_astro` 中 `supernova star` 匹配 `supernovae stars`，因为在分类词典定义中指定了 `english_stem` 词干分析器。该词干分析器移除了 `e` 和 `s`。

要和替补一样也索引原始短语，只要将它包含在定义的右手部分中：

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
plainto_tsquery
-----
'sn' & 'supernova' & 'star'
```

## 20.6.5 Ispell 词典

Ispell 词典模板支持词法词典，它可以把一个词的很多不同语言学的形式正规化成相同的词位。例如，一个英语 Ispell 词典可以匹配搜索词 `bank` 的词尾变化和词形变化，例如 `banking`、`banked`、`banks`、`banks'` 和 `bank's`。

标准的 KingbaseES 发布不包括任何 Ispell 配置文件。用于很多种语言的词典可以从 Ispell 得到。此外，也支持一些更现代的词典文件格式 `MySpell` (OO < 2.0.1) 和 `Hunspell` (OO >= 2.0.2)。词典列表可以从 [OpenOffice Wiki](#) 上获取。

创建一个 Ispell 词典，执行三步：

- 下载词典配置文件。OpenOffice 扩展文件的扩展名是 `.oxf`。有必要抽取 `.aff` 和 `.dic` 文件，把扩展改为 `.affix` 和 `.dict`。对于某些词典文件，还需要使用下面的命令把字符转换成 UTF-8 编码（例如挪威语词典）：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- 拷贝文件到 `$SHAREDIR/tsearch_data` 目录
- 用下面的命令把文件载入到 KingbaseES：

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

`DictFile`、`AffFile` 和 `StopWords` 指定词典、词缀和停用词文件的基础名称。停用词文件的格式和前面解释的 `simple` 词典类型相同。其他文件的格式在这里没有指定，但是也可以从上面提到的网站获得。

Ispell 词典通常识别一个有限集合的词，这样它们后面应该跟着另一个更广义的词典；例如，一个 `Snowball` 词典，它可以识别所有东西。

Ispell 的 `.affix` 文件具有下面的结构：

```
prefixes
flag *A:
    .      > RE      # As in enter > reenter
suffixes
flag T:
    E      > ST      # As in late > latest
    [^AEIOU]Y > -Y, IEST # As in dirty > dirtiest
    [AEIOU]Y > EST    # As in gray > grayest
    [^EY]  > EST    # As in small > smallest
```

`.dict` 文件具有下面的结构：

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

.dict 文件的格式是:

```
basic_form/affix_class_name
```

在 .affix 文件中, 每一个词缀标志以下面的格式描述:

```
condition > [-stripping_letters,] adding_affix
```

这里的条件具有和正则表达式相似的格式。它可以使用分组 [...] 和 [^...]。例如, [AEIOU]Y 表示词的最后一个字母是 "y" 并且倒数第二个字母是 "a"、"e"、"i"、"o" 或者 "u"。[^EY] 表示最后一个字母既不是 "e" 也不是 "y"。

IsPELL 词典支持划复合词, 这是一个有用的特性。注意词缀文件应该用 compoundwords controlled 语句指定一个特殊标志, 它标记可以参与到复合格式中的词典词:

```
compoundwords controlled z
```

下面是挪威语的一些例子:

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkemesterassistent');
  {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
  {sjokoladefabrikk,sjokolade,fabrikk}
```

MySpell 格式是 Hunspell 格式的一个子集。Hunspell 的 .affix 文件具有下面的结构:

```
PFX A Y 1
PFX A 0 re .
SFX T N 4
SFX T 0 st e
SFX T y iest [^aeiouy
SFX T 0 est [aeiouy
SFX T 0 est [^ey]
```

一个词缀类的第一行是头部。头部后面列出了词缀规则的域:

- 参数名 (PFX 或者 SFX)
- 标志 (词缀类的名称)
- 从该词的开始 (前缀) 或者结尾 (后缀) 剥离字符
- 增加词缀

- 和正则表达式格式类似的条件。

.dict 文件看起来和 Ispell 的 .dict 文件相似:

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

**注意:** MySpell 不支持复合词。Hunspell 则对复合词有更好的支持。当前, KingbaseES 只实现了 Hunspell 中基本的复合词操作。

## 20.6.6 Snowball 词典

Snowball 词典模板基于 Martin Porter 的一个项目, 他是流行的英语 Porter 词干分析算法的发明者。Snowball 现在对许多语言提供词干分析算法 (详见 [Snowball 站点](#))。每一个算法懂得按照其语言中的拼写, 如何缩减词的常见变体形式为一个基础或词干。一个 Snowball 词典要求一个 `language` 参数来标识要用哪种词干分析器, 并且可以选择地指定一个 `stopword` 文件名来给出一个要被消除的词列表 (KingbaseES 的标准停用词列表也是由 Snowball 项目提供的)。例如, 有一个内建的定义等效于

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

停用词文件格式和已经解释的一样。

一个 Snowball 词典识别所有的东西, 不管它能不能简化该词, 因此它应当被放置在词典列表的最后。把它放在任何其他词典前面是没有用处的, 因为一个记号永远不会穿过它而进入到下一个词典。

## 20.7 配置例子

一个文本搜索配置指定了将一个文档转换成一个 `tsvector` 所需的所有选项: 用于把文本分解成记号的解析器, 以及用于将每一个记号转换成词位的词典。每一次 `to_tsvector` 或 `to_tsquery` 的调用都需要一个文本搜索配置来执行其处理。配置参数 `default_text_search_config` 指定了默认配置的名称, 如果忽略了显式的配置参数, 文本搜索函数将会使用它。它可以在 `kingbase.conf` 中设置, 或者使用 `SET` 命令为一个单独的会话设置。

有一些预定义的文本搜索配置可用, 并且你可以容易地创建自定义的配置。为了便于管理文本搜索对象, 可以使用一组 SQL 命令, 并且有多个 `ksql` 命令可以显示有关文本搜索对象 (`ksql 支持`) 的信息。

作为一个例子, 将创建一个配置 `pg`, 从复制内建的 `english` 配置开始:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = sys_catalog.english );
```

将使用一个 KingbaseES 相关的同义词列表，并将它存储在 `$SHAREDIR/tsearch_data/sys_dict.syn` 中。文件内容看起来像：

```
kingbase    pg
kingbase    pg
kingbase    pg
```

定义同义词词典如下：

```
CREATE TEXT SEARCH DICTIONARY sys_dict (
    TEMPLATE = synonym,
    SYNONYMS = sys_dict
);
```

接下来注册 Ispell 词典 `english_ispell`，它有其自己的配置文件：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

在配置 `pg` 中建立词的映射：

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_ascii,
                    word, hword, hword_part
    WITH sys_dict, english_ispell, english_stem;
```

选择不索引或搜索某些内建配置确实处理的记号类型：

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

测试配置：

```
SELECT * FROM ts_debug('public.pg', '
KingbaseES, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

下一个步骤是设置会话让它使用新配置，它被创建在 `public` 模式中：



```

=> \dF
    List of text search configurations
 Schema | Name | Description
-----+-----+-----
 public | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
-----
 public.pg

```

## 20.8 中文分词

### 20.8.1 zhparser 中文分词插件支持 utf8 和 gbk 字符集

zhparser 中文分词插件使用说明:

```

test=# create extension zhparser;
CREATE EXTENSION
test=# CREATE TEXT SEARCH CONFIGURATION testzhcfg (PARSER = zhparser);
CREATE TEXT SEARCH CONFIGURATION
test=#
test=# ALTER TEXT SEARCH CONFIGURATION testzhcfg ADD MAPPING FOR n,v,a,i,e,l WITH simple;
ALTER TEXT SEARCH CONFIGURATION
test=# SELECT * FROM ts_parse('zhparser', 'KingbaseES 中文分词插件测试');
 tokid | token
-----+-----
    101 | KingbaseES
    110 | 中文
    118 | 分词
    110 | 插件
    118 | 测试
(5 rows)

test=# SELECT to_tsvector('testzhcfg', 'KingbaseES 中文分词插件测试');
          to_tsvector
-----
'kingbasees':1 '中文':2 '分词':3 '插件':4 '测试':5
(1 row)

```

```
test=# SELECT to_tsquery('testzhcfg', '分词');
      to_tsquery
-----
 '分词'
(1 row)
```

## 20.8.2 sys\_jieba 中文分词插件支持 utf8 字符集

sys\_jieba 中文分词插件使用说明:

```
test=# CREATE EXTENSION sys_jieba;
CREATE EXTENSION
test=# SELECT * FROM ts_parse('jieba', 'KingbaseES 中文分词插件测试');
 tokid | token
-----+-----
      2 | KingbaseES
      2 | 分词
      2 | 插件
      2 | 测试
(5 rows)

test=# SELECT to_tsvector('jiebacfg', 'KingbaseES 中文分词插件测试');
      to_tsvector
-----
 'kingbasees':1 '中文':2 '分词':3 '插件':4 '测试':5
(1 row)

test=# SELECT to_tsquery('jiebacfg', '分词');
      to_tsquery
-----
 '分词'
(1 row)
```

## 20.9 流版式文件内容抽取

### 20.9.1 使用 ftutilx 插件抽取文件文本内容

ftutilx 插件提供了 extracttext 函数用于抽取存储在 blob 类型字段内的文件内容。extracttext 函数接受一个代表文件内容的 blob 类型参数，返回抽取的 text 类型文本内容。其中 blob 类型字段内容可以包括 pdf、doc、docx、wps、xls、xlsx、ppt 和 pptx 格式文件。ftutilx 插件不支持加密文件格式。

```
CREATE EXTENSION ftutilx;
CREATE TABLE tab (title text, body blob);
INSERT INTO tab VALUES ('test.doc', blob_import('/home/test/data.doc'));
SELECT title, length(extracttext(body)) FROM tab;
```

ftutilx 插件支持以下参数:

- 1) ftutilx.max\_string\_length : 抽取结果最大长度, 默认值: 128M, 该参数设置后立即生效。
- 2) ftutilx.jvm\_option\_string : JVM 初始化参数, 默认值: "-Xmx1024m,-Xms1024m,-Xmn256m,-XX:MetaspaceSize=64m,-XX:MaxMetaspaceSize=128m,-XX:CompressedClassSpaceSize=256m", 该参数只在会话进程中首次调用 extracttext 函数创建 JVM 时生效, 再次设置该参数不再有效。

在数据库默认扩展加载机制下, 在一个会话中创建扩展以后, 在新的会话开始后并不是立即加载扩展动态库, 而是直到首次调用扩展中的接口时才会加载扩展动态库, 导致在这之前在新会话中设置扩展参数无效。解决方法是: 修改数据库配置文件中的 shared\_preload\_libraries 或者 session\_preload\_libraries 两个参数之一, 使参数值包括 ftutilx, 即可在新会话开始后立即加载 ftutilx 扩展动态库, 并设置扩展参数。

## 20.9.2 使用 ftutilx 全文检索的联合使用方案

由于电子文档内容抽取速度较慢, 为提高全文检索性能, 可以在表中添加存储列, 用于存储内容抽取结果或者词位列表。

方案一:

```
CREATE EXTENSION zhparser;
CREATE TEXT SEARCH CONFIGURATION zhparsercfg (PARSER = zhparser);
ALTER TEXT SEARCH CONFIGURATION zhparsercfg ADD MAPPING FOR n,v,a,i,e,l WITH simple;

CREATE EXTENSION ftutilx;
CREATE TABLE tab (title text, body blob);

ALTER TABLE tab ADD COLUMN content text GENERATED ALWAYS AS (extracttext(body)) STORED;
CREATE INDEX tab_idx ON tab USING GIN (to_tsvector('zhparsercfg', content));

INSERT INTO tab VALUES ('test.doc', blob_import('/home/test/data.doc'));

SELECT title FROM tab WHERE to_tsvector('zhparsercfg', content) @@ to_tsquery('日志');
```

方案二:

```
CREATE EXTENSION zhparser;
CREATE TEXT SEARCH CONFIGURATION zhparsercfg (PARSER = zhparser);
ALTER TEXT SEARCH CONFIGURATION zhparsercfg ADD MAPPING FOR n,v,a,i,e,l WITH simple;

CREATE EXTENSION ftutilx;
CREATE TABLE tab (title text, body blob);
```

```
ALTER TABLE tab ADD COLUMN tab_idx_col tsvector GENERATED ALWAYS AS (to_tsvector('zhparsercfg',
extracttext(body))) STORED;
CREATE INDEX tab_idx ON tab USING GIN (tab_idx_col);

INSERT INTO tab VALUES ('test.doc', blob_import('/home/test/data.doc'));

SELECT title FROM tab WHERE tab_idx_col @@ to_tsquery('日志');
```

### 20.9.3 使用 ftutilx 插件的注意事项

1) ftutilx 需要依赖于 jre-1.8.0 运行时环境，部署后需要设置 LD\_LIBRARY\_PATH 系统环境变量包含 jre-1.8.0 的 libjvm.so 路径。

2) ftutilx.max\_string\_length 参数用于配置抽取结果的最大长度，但由于 tsvector 目前最大支持 (1M-1)，所以 extracttext 结合 to\_tsvector 使用时，分词结果大小不能超过 (1M-1)。

3) ftutilx 需要创建 JVM，JVM 会占用较多内存。虽然调整 ftutilx.jvm\_option\_string 的-Xmx 可以限制 JVM 的内存占用，但过小的-Xmx 值会导致大文件解析时 JVM 发生内存不足异常。

4) 基于前面的全文检索联合使用方案，在系统内存较少的环境中，需要限制并行插入数据的会话进程数量，以免系统内存被耗尽。

## 20.10 测试和调试文本搜索

一个自定义文本搜索配置的行为很容易变得混乱。本节中描述的函数对于测试文本搜索对象有用。你可以测试一个完整的配置，或者独立测试解析器和词典。

### 20.10.1 配置测试

函数 ts\_debug 允许简单地测试一个文本搜索配置。

```
ts_debug([ config regconfig, ] document text,
        OUT alias text,
        OUT description text,
        OUT token text,
        OUT dictionaries regdictionary[],
        OUT dictionary regdictionary,
        OUT lexemes text[])
returns setof record
```

`ts_debug` 显示 `document` 的每一个记号的信息，记号由解析器产生并由配置的词典处理过。该函数使用由 “`config`” 指定的配置，如果该参数被忽略则使用 `default_text_search_config` 指定的配置。

`ts_debug` 为解析器在文本中标识的每一个记号返回一行。被返回的列是：

- `alias text` — 记号类型的短名称
- `description text` — 记号类型的描述
- `token text` — 记号的文本
- `dictionaries regdictionary []` — 配置为这种记号类型选择的词典
- `dictionary regdictionary` — 识别该记号的词典，如果没有词典能识别则为 `NULL`
- `lexemes text []` — 识别该记号的词典产生的词位，如果没有词典能识别则为 `NULL`；一个空数组 (`{}`) 表示该记号被识别为一个停用词

这里是一个简单的例子：

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat rats');
  alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | cat | {english_stem} | english_stem | {cat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | sat | {english_stem} | english_stem | {sat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | on | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | mat | {english_stem} | english_stem | {mat}
blank | Space symbols | | {} | | 
blank | Space symbols | - | {} | | 
asciiword | Word, all ASCII | it | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | ate | {english_stem} | english_stem | {ate}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | a | {english_stem} | english_stem | {}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | fat | {english_stem} | english_stem | {fat}
blank | Space symbols | | {} | | 
asciiword | Word, all ASCII | rats | {english_stem} | english_stem | {rat}
```

为了一个更广泛的示范，为英语语言创建一个 `public.english` 配置和 `Ispell` 词典：

```

CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = sys_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
    ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;

```

```

SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
  alias | description | token | dictionaries | dictionary | lexemes
-----+-----+-----+-----+-----+-----
asciiword|Word, all ASCII|The |{english_ispell,|english_ispell| {}
          |          |      |              |english_stem}
blank    |Space symbols |      |{}            |          |
asciiword|Word, all ASCII|Brightest |{english_ispell,|english_ispell|{bright}
          |          |      |              |english_stem}
blank    |Space symbols |      |{}            |          |
asciiword|Word, all ASCII|supernovaes|{english_ispell,|english_stem |{supernova}
          |          |      |              |english_stem}

```

在这个例子中，词 `Brightest` 被解析器识别为一个 ASCII 词（别名 `asciiword`）。对于这种记号类型，词典列表是 `english_ispell` 和 `english_stem`。该词被 `english_ispell` 识别，这个词典将它缩减为名词 `bright`。词 `supernovaes` 对于 `english_ispell` 词典是未知的，因此它被传递给下一个词典，并且幸运地是，它被识别了（实际上，`english_stem` 是一个 Snowball 词典，它识别所有的东西；这也是为什么它被放置在词典列表的尾部）。

词 `The` 被 `english_ispell` 词典识别为一个停用词（停用词）并且将不会被索引。空格也被丢弃，因为该配置没有为它们提供词典。

可以通过显式地指定想看哪些列来缩减输出的宽度：

```

SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');
  alias | token | dictionary | lexemes
-----+-----+-----+-----
asciiword | The | english_ispell | {}
blank | | | |
asciiword | Brightest | english_ispell | {bright}
blank | | | |
asciiword | supernovaes | english_stem | {supernova}

```

## 20.10.2 解析器测试

下列函数允许直接测试一个文本搜索解析器。

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

`ts_parse` 解析给定的 “document” 并返回一系列记录，每一个记录对应一个由解析产生的记号。每一个记录包括一个 `tokid` 展示分配给记号的类型以及一个 `token` 展示记号的文本。例如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-----+-----
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

`ts_token_type` 返回一个表，该表描述指定解析器能够识别的每一种记号类型。对于每一种记号类型，该表给出了解析器用来标注该类型记号的整数 `tokid`，还给出了在配置命令中命名该记号类型的 `alias`，以及一个简短的 `description`。例如：

```
SELECT * FROM ts_token_type('default');
 tokid | alias | description
-----+-----+-----
     1 | asciiword | Word, all ASCII
     2 | word | Word, all letters
     3 | numword | Word, letters and digits
     4 | email | Email address
     5 | url | URL
     6 | host | Host
     7 | sfloat | Scientific notation
     8 | version | Version number
     9 | hword_numpart | Hyphenated word part, letters and digits
    10 | hword_part | Hyphenated word part, all letters
    11 | hword_asciipart | Hyphenated word part, all ASCII
```

12	blank	Space symbols
13	tag	XML tag
14	protocol	Protocol head
15	numhword	Hyphenated word, letters and digits
16	asciihword	Hyphenated word, all ASCII
17	hword	Hyphenated word, all letters
18	url_path	URL path
19	file	File or path name
20	float	Decimal notation
21	int	Signed integer
22	uint	Unsigned integer
23	entity	XML entity

### 20.10.3 词典测试

`ts_lexize` 函数帮助词典测试。

```
ts_lexize(dict regdictionary, token text) returns text[]
```

如果输入的“token”是该词典已知的，则 `ts_lexize` 返回一个词位数组；如果记号是词典已知的但是它是一个停用词，则返回一个空数组；或者如果它对词典是未知词，则返回 `NULL`。

例子：

```
SELECT ts_lexize('english_stem', 'stars');
ts_lexize
-----
{star}

SELECT ts_lexize('english_stem', 'a');
ts_lexize
-----
{}
```

注意: `ts_lexize` 函数期望一个单一记号而不是文本。下面的情况会让它搞混：

```
SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
?column?
-----
t
```

分类词典 `thesaurus_astro` 确实知道短语 `supernovae stars`，但是 `ts_lexize` 会失败，因为它无法解析输入文本而把它当做一个单一记号。可以使用 `plainto_tsquery` 或 `to_tsvector` 来测试分类词典，例如：



```
SELECT plainto_tsquery('supernovae stars');
plainto_tsquery
-----
'sn'
```

## 20.11 GIN 和 GiST 索引类型

有两种索引可以被用来加速全文搜索。注意全文搜索并非一定需要索引，但是在一个定期会被搜索的列上，通常需要一个索引。

**CREATE INDEX name ON table USING GIN( column );** 创建一个基于 GIN（通用倒排索引）的索引。“column”必须是 `tsvector` 类型。

**CREATE INDEX name ON table USING GIST( column );** 创建一个基于 GiST（通用搜索树）的索引。“column”可以是 `tsvector` 或 `tsquery` 类型。

GIN 索引是更好的文本搜索索引类型。作为倒排索引，每个词（词位）在其中都有一个索引项，其中有压缩过的匹配位置的列表。多词搜索可以找到第一个匹配，然后使用该索引移除缺少额外词的行。GIN 索引只存储 `tsvector` 值的词（词位），并且不存储它们的权重标签。因此，在使用涉及权重的查询时需要一次在表行上的重新检查。

一个 GiST 索引是有损的，这表示索引可能产生假匹配，并且有必要检查真实的表行来消除这种假匹配（KingbaseES 在需要时会自动做这一步）。GiST 索引之所以是有损的，是因为每一个文档在索引中被表示为一个定长的签名。该签名通过哈希每一个词到一个 `n` 位串中的一个单一一位来产生，通过将所有这些位 OR 在一起产生一个 `n` 位的文档签名。当两个词哈希到同一个位位置时就会产生假匹配。如果查询中所有词都有匹配（真或假），则必须检索表行查看匹配是否正确。

GiST 索引可以覆盖，例如，使用 `INCLUDE` 子句。包含的列可以有数据类型，而不需要任何 GiST 操作符类。包含的属性将不压缩存储。

有损性导致的性能下降归因于不必要的表记录（即被证实为假匹配的记录）获取。因为表记录的随机访问是较慢的，这限制了 GiST 索引的可用性。假匹配的可能性取决于几个因素，特别是唯一词的数量，因此推荐使用词典来缩减这个数量。

注意 GIN 索引的构件时间常常可以通过增加 `maintenance_work_mem` 来改进，而 GiST 索引的构建时间则与该参数无关。

对大集合分区并正确使用 GIN 和 GiST 索引允许实现带在线更新的快速搜索。分区可以在数据库层面上使用表继承来完成，或者是通过将文档分布在服务器上并收集外部的搜索结果，例如通过 外部数据访问。后者是可能的，因为排名函数只使用本地信息。

## 20.12 RUM 索引类型

RUM 索引作为一个全文索引类型，是 GIN 全文索引的扩展。与内建的 GIN 和 GiST 全文索引的区别是它以插件包的形式存在。在使用前需要通过 `CREATE EXTENSION` 语句来加载插件包。

```
CREATE EXTENSION RUM;
```

RUM 索引虽然基于 GIN 索引实现，但是它能够更快地执行全文索引。

GIN 索引可以基于 `tsvector` 和 `tsquery` 类型执行快速的全文搜索，但是它也存在以下的问题：

- 排序慢。需要有关词素的位置信息才能进行排序。GIN 索引不存储词素的位置信息，因此，在索引扫描之后，我们需要额外的堆扫描以检索词素位置。
- 短语搜索慢。该问题与上一个问题是一个原因，它需要位置信息来执行短语搜索。
- 时间戳排序慢。GIN 索引无法在带有词素的索引中存储相关信息，因此，要执行额外的堆扫描。

RUM 索引解决了上述问题，但是它也存在缺点，因为它要存储额外的信息和需要使用 WAL 日志，导致它比 GIN 索引构建和插入时更慢。

### 20.12.1 常用操作符和函数

`rum` 插件提供了如下的操作符

操作符	返回值	描述
<code>tsvector &lt;=&gt; tsquery</code>	<code>float4</code>	返回 <code>tsvector</code> 和 <code>tsquery</code> 的间距
<code>timestamp &lt;=&gt; timestamp</code>	<code>float8</code>	返回两个 <code>timestamp</code> 的间距
<code>timestamp &lt;=  timestamp</code>	<code>float8</code>	仅返回左边 <code>timestamps</code> 的间距
<code>timestamp  =&gt; timestamp</code>	<code>float8</code>	仅返回右边 <code>timestamps</code> 的间距

后面的三个操作符同样适用于 `timestamptz`, `int2`, `int4`, `int8`, `float4`, `float8`, `money` 和 `oid` 类型。

### 20.12.2 操作符类

`rum` 插件提供了如下的操作符类

#### 20.12.2.1 rum\_tsvector\_ops

适用类型: `tsvector`

这个操作符类存储 tsvector 词素的位置信息，同时支持使用操作符 <=> 进行排序及前缀搜索。如下例所示：

```
CREATE TABLE test_rum(t text, a tsvector);

CREATE TRIGGER tsvectorupdate
BEFORE UPDATE OR INSERT ON test_rum
FOR EACH ROW EXECUTE PROCEDURE tsvector_update_trigger('a', 'sys_catalog.english', 't');

INSERT INTO test_rum(t) VALUES ('The situation is most beautiful');
INSERT INTO test_rum(t) VALUES ('It is a beautiful');
INSERT INTO test_rum(t) VALUES ('It looks like a beautiful place');
```

要使用 rum 索引，需要先创建 rum 插件：

```
CREATE EXTENSION rum;
```

然后我们创建 rum 索引：

```
CREATE INDEX rumidx ON test_rum USING rum (a rum_tsvector_ops);
```

执行如下的 SQL 语句：

```
SELECT t, a <=> to_tsquery('english', 'beautiful | place') AS rank
FROM test_rum
WHERE a @@ to_tsquery('english', 'beautiful | place')
ORDER BY a <=> to_tsquery('english', 'beautiful | place');
      t                | rank
-----+-----
It looks like a beautiful place | 8.22467
The situation is most beautiful | 16.4493
It is a beautiful                | 16.4493
(3 rows)

SELECT t, a <=> to_tsquery('english', 'place | situation') AS rank
FROM test_rum
WHERE a @@ to_tsquery('english', 'place | situation')
ORDER BY a <=> to_tsquery('english', 'place | situation');
      t                | rank
-----+-----
The situation is most beautiful | 16.4493
It looks like a beautiful place | 16.4493
(2 rows)
```

### 20.12.2.2 rum\_tsvector\_hash\_ops

适用类型: tsvector

这个操作符类存储了 tsvector 词素位置的哈希值，支持操作符 <=> 排序，但是不支持前缀搜索。

### 20.12.2.3 rum\_TYPE\_ops

TYPE 对应类型: int2, int4, int8, float4, float8, money, oid, time, timetz, date, interval, macaddr, inet, cidr, text, varchar, char, bytea, bit, varbit, numeric, timestamp, timestamptz

支持的操作符: <, <=, =, >=, > 适用上述所有类型; <=>, <=| 和 |=> 适用 int2, int4, int8, float4, float8, money, oid,timestamp 和 timestamptz 类型。

支持使用 <=>, <=| 和 |=> 操作符进行排序，同时也可以使用 rum\_tsvector\_addon\_ops, rum\_tsvector\_hash\_addon\_ops 和 rum\_anyarray\_addon\_ops 操作符类。

### 20.12.2.4 rum\_tsvector\_addon\_ops

适用类型: tsvector

这个操作符支持以附加的形式保存 tsvector 词素。

```
CREATE TABLE tstst (id int, t tsvector, d timestamp);

\copy tstst from 'rum/data/tstst.data'

CREATE INDEX tstst_idx ON tstst USING rum (t rum_tsvector_addon_ops, d)
WITH (attach = 'd', to = 't');
```

执行如下 SQL 语句:

```
EXPLAIN (costs off)
  SELECT id, d, d <=> '2016-05-16 14:21:25' FROM tstst WHERE t @@ 'wr&qh' ORDER BY d <=> '2016-05-16 14:21:
25' LIMIT 5;

          QUERY PLAN
-----
Limit
->  Index Scan using tstst_idx on tstst
      Index Cond: (t @@ ''wr'' & ''qh''::tsquery)
      Order By: (d <=> 'Mon May 16 14:21:25 2016'::timestamp without time zone)
(4 rows)

SELECT id, d, d <=> '2016-05-16 14:21:25' FROM tstst WHERE t @@ 'wr&qh' ORDER BY d <=> '2016-05-16 14:21:25'
LIMIT 5;
 id |          d          | ?column?
```

```

-----+-----
355 | Mon May 16 14:21:22.326724 2016 |      2.673276
354 | Mon May 16 13:21:22.326724 2016 |    3602.673276
371 | Tue May 17 06:21:22.326724 2016 |   57597.326724
406 | Wed May 18 17:21:22.326724 2016 |  183597.326724
415 | Thu May 19 02:21:22.326724 2016 | 215997.326724
(5 rows)

```

### 20.12.2.5 rum\_tsvector\_hash\_addon\_ops

适用类型: tsvector

这个操作符以附加的信息保存 tsvector 词素的哈希值。它不支持前缀搜索。

### 20.12.2.6 rum\_tsquery\_ops

适用类型: tsquery

在附加信息里保存查询树的分支:

```

CREATE TABLE query (q tsquery, tag text);

INSERT INTO query VALUES ('supernova & star', 'sn'),
    ('black', 'color'),
    ('big & bang & black & hole', 'bang'),
    ('spiral & galaxy', 'shape'),
    ('black & hole', 'color');

CREATE INDEX query_idx ON query USING rum(q);

```

执行如下 SQL 语句:

```

SELECT * FROM query
    WHERE to_tsvector('black holes never exists before we think about them') @@ q;
      q      | tag
-----+-----
'black'      | color
'black' & 'hole' | color
(2 rows)

```

### 20.12.2.7 rum\_anyarray\_ops

适用类型: anyarray

这个操作符类保存 anyarray 元素的长度，支持 &&, @>, <@, =, % 操作符，支持使用 <=> 操作进行排序。

```
CREATE TABLE test_array (i int2[]);

INSERT INTO test_array VALUES ('{1}'), ('{0}'), ('{1,2,3,4}'), ('{1,2,3}'), ('{1,2}'), ('{1}');

CREATE INDEX idx_array ON test_array USING rum (i rum_anyarray_ops);
```

执行如下 SQL 语句：

```
SET enable_seqscan TO off;

EXPLAIN (COSTS OFF) SELECT * FROM test_array WHERE i && '{1}' ORDER BY i <=> '{1}' ASC;
      QUERY PLAN
-----
Index Scan using idx_array on test_array
  Index Cond: (i && '{1}'::smallint[])
  Order By: (i <=> '{1}'::smallint[])
(3 rows)

SELECT * FROM test_array WHERE i && '{1}' ORDER BY i <=> '{1}' ASC;
      i
-----
 {1}
 {1,2}
 {1,2,3}
 {1,2,3,4}
(4 rows)
```

### 20.12.2.8 rum\_anyarray\_addon\_ops

使用类型: anyarray

这个操作符类支持在任何支持附加属性的地方保存 anyarray 元素。

## 20.13 ksql 支持

关于文本搜索配置对象的信息可以在 ksql 中使用一组命令获得：

```
\dF{d,p,t}[+] [PATTERN]
```

可选的 + 能产生更多细节。

可选参数“PATTERN”可以是一个文本搜索对象的名称，可以是模式限定的。如果“PATTERN”被忽略，则所有可见对象的信息都将被显示。“PATTERN”可以是一个正则表达式并且可以为模式和对象名称提供独立的模式。下面的例子展示了这些特性：

```
=> \dF *fulltext*
      List of text search configurations
Schema | Name          | Description
-----+-----+-----
public | fulltext_cfg |
```

```
=> \dF *.fulltext*
      List of text search configurations
Schema | Name          | Description
-----+-----+-----
fulltext | fulltext_cfg |
public   | fulltext_cfg |
```

可用的命令是：

- \dF[+] [PATTERN] 列出文本搜索配置（加上 + 得到更多细节）。
- \dFd[+] [PATTERN] 列出文本搜索词典（加上 + 得到更多细节）。
- \dFp[+] [PATTERN] 列出文本搜索解析器（加上 + 得到更多细节）。
- \dFt[+] [PATTERN] 列出文本搜索模板（加上 + 得到更多细节）。

## 20.14 限制

KingbaseES 的文本搜索特性的当前限制是：

- 每一个词位的长度必须小于 2K 字节
- 一个 `tsvector`（词位 + 位置）的长度必须小于 1 兆字节
- 词位的数量必须小于  $2^{64}$
- `tsvector` 中的位置值必须大于 0 并且小于 16,383
- `<N>`（FOLLOWED BY）`tsquery` 操作符中的匹配距离不能超过 16,384
- 每个词位不超过 256 个位置
- 一个 `tsquery` 中结点（词位 + 操作符）的个数必须小于 32,768

为了对比，KingbaseES V7 的文档包含 10,441 个唯一词，总数 335,420 个词，并且最频繁的词“kingbase”在 655 个文档中被提到 6,127 次。

另一个例子—KingbaseES 的邮件列表归档在 461,020 条消息的 57,491,343 个词位中包含 910,989 个唯一词。

---

# 第 21 章 规则系统

---

本文档讨论 KingbaseES 中的规则系统。

规则系统（查询重写规则系统）与存储过程和触发器完全不同。它将查询修改为需要考虑规则，并且把修改过的查询传递给查询规划器进行规划和执行。规则系统非常强大，可以被应用于查询语言过程、视图和版本等多领域。

本章节包含以下内容：

- 查询树
- 视图和规则系统
- 物化视图
- INSERT、UPDATE 和 DELETE 上的规则
- 规则和权限
- 规则和命令状态
- 规则 vs 触发器

## 21.1 查询树

了解规则系统是如何工作的，必须知道它在什么时候被调用以及其输入和结果是什么。

规则系统位于解析器和规划器之间。它采用解析器的输出（即一个查询树）和用户定义的重写规则（也是查询树，不过带有一些额外信息），并且常见零个或者更多个查询树作为结果。因此它的输入和输出总是那些规划器自身就能产生的东西，并且因此它看到的任何东西都可以被表示成一个 SQL 语句。

那么什么是一个查询树？它是一个 SQL 语句的一种内部表示，其中用于创建它的每一个单独的部分都被独立存储。如果你设置了配置参数 `debug_print_parse`、`debug_print_rewritten` 或 `debug_print_plan`，这些查询树可以被显示在服务器日志中。规则动作也被做为查询树存储在系统目录 `sys_rewrite` 中。它们没有被格式化为日志输出的形式，但是它们包含完全相同的信息。

阅读一棵未加工的查询树需要一些经验。但是由于查询树的 SQL 表示形式足以用来理解规则系统，本章将不会教授如何阅读查询树。



在阅读本章中查询树的 SQL 表现形式时，读者需要能够知道语句被分解成了哪些部分并且能在查询树结构中标识它们。一棵查询树的部分有：

- 命令类型

这是一个简单的值来说明是哪一种命令（SELECT、INSERT、UPDATE、DELETE）产生了该查询树。

- 范围表

范围表是被使用在该查询中的关系的列表。在一个 SELECT 语句中，范围表是在关键词 FROM 后面给出的关系。

每一个范围表项标识一个表或视图，并且说明在该查询的其他部分要以哪个名称调用它。在查询树中，范围表项被使用编号而不是名称来引用，因此在一个 SQL 语句中出现重复的名字也没有关系。在规则的范围表被合并以后可能会发生这种情况。本章中的例子将不会有这种情况。

- 结果关系

这是一个指向范围表的索引，它标识了该查询的结果应该去哪个关系。

SELECT 查询没有结果关系（特殊情况 SELECT INTO 几乎等于 CREATE TABLE 后面跟上 INSERT ... SELECT，并且不在此单独讨论）。

对于 INSERT、UPDATE 和 DELETE 命令，结果关系是修改要进行的表（或视图！）。

- 目标列表

目标列表是一个表达式的列表，它定义了查询的结果。在一个 SELECT 的情况下，这些表达式会构建出该查询最终的输出。它们对应于关键字 SELECT 和 FROM 之间的表达式（\* 是一个关系所有列名的缩写。解析器会把它扩展成独立的列，因此规则系统永远见不到它）。

DELETE 命令不需要一个目标列表，因为它们不产生任何结果。相反，规划器会向空的目标列表中加入一个特殊的 CTID 项来允许执行器找到要被删除的行（当结果关系是一个普通表时才加入 CTID。如果结果关系是一个视图，则会被规则系统加入一个整行变量，如更新一个视图 所述）。

对于 INSERT 命令，目标列表描述了将要进入到结果关系中的新行。它由 VALUES 子句中的表达式或来自 INSERT ... SELECT 中 SELECT 子句的表达式构成。重写处理的第一步会为那些没有被原始命令赋值但有默认值的列增加目标列表项。任何剩余的列（既没有给定值也没有默认值）将被规划器用一个常量空值表达式填充。

对于 UPDATE 命令，目标列表描述要替换旧行的新行。在规则系统中，它只包含来自命令的 SET column = expression 部分的表达式。规划器将处理缺失的列，做法是为它们插入表达式，这种表达式会把旧行的值复制到新行。正如 DELETE 一样，会增加一个 CTID 或整行变量，这样执行器能够标识要被更新的旧行。

目标列表中的每一个项所包含的表达式可以是一个常量值、一个指向范围表中关系的列的变量、一个参数或一个由函数调用、常量、变量、操作符等构成的表达式树。

- 条件

查询的条件是一个表达式，它很像包含在目标列表项中的表达式。这个表达式的结果值是一个布尔值，它说明对最终结果行的操作（INSERT、UPDATE、DELETE 或 SELECT）是否应该被执行。它对应于一个 SQL 语句的 WHERE 子句。

- 连接树

查询的连接树展示了 FROM 子句的结构。对于一个 SELECT ... FROM a, b, c 这样的简单查询，连接树就是 FROM 项的一个列表，因为我们被允许以任何顺序连接它们。但是当 JOIN 表达式（特别是外连接）被使用时，我们必须按照连接显示的顺序来连接。在这种情况下，连接树展示了 JOIN 表达式的结构。与特定 JOIN 子句（来自 ON 或 USING）相关的限制被存储为附加到那些连接树节点的条件表达式。我们发现把顶层 WHERE 表达式存储为附加到顶层连接树项的一个条件也很方便。这样实际上连接树表达了一个 SELECT 的 FROM 和 WHERE 子句。

- 其他

查询树的其他部分（如 ORDER BY 子句）在这里并不受到关注。规则系统在设计时会替换这里的某些项，但是这些与规则系统的基础没有什么关系。

## 21.2 视图和规则系统

KingbaseES 中的视图是通过规则系统来实现的。事实上，下面的命令

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

与下面两个命令相比没有不同：

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
SELECT * FROM mytab;
```

因为这就是 CREATE VIEW 命令在内部所作的。这样做有一些副作用。其中之一就是在 KingbaseES 系统目录中的视图信息与表的信息完全一样。所以对于解析器来说，表和视图之间完全没有区别。它们是同样的事物：关系。

### 21.2.1 SELECT 规则如何工作

规则 ON SELECT 被应用于所有查询作为最后一步，即使给出的是一条 INSERT、UPDATE 或 DELETE 命令。而且它们与其他命令类型上的规则有着不同的语义，它们会就地修改查询树而不是创建一个新的查询树。因此我们首先描述 SELECT 规则。

目前，一个 ON SELECT 规则中只能有一个动作，而且它必须是一个无条件的 INSTEAD 的 SELECT 动作。这个限制是为了令规则足够安全，以便普通用户也可以打开它们，并且它限制 ON SELECT 规则使之行为类似视图。

本章的例子是两个连接视图，它们做一些运算并且某些更多视图会轮流使用它们。最前面的两个视图之一后面将利用对 INSERT、UPDATE 和 DELETE 操作增加规则的方法被自定义，这样最终结果将是一个视图，它表现得像一个具有魔力的真正的表。这个例子不适合于作为简单易懂的例子，它可能会让本章更难懂。但是用一个覆盖所有关键点的例子来一步一步讨论要比举很多例子搞乱思维好。

例如，我们需要一个小巧的 min 函数用于返回两个整数值中较小的那个。我们这样创建它：

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS $$
    SELECT CASE WHEN $1 < $2 THEN $1 ELSE $2 END
$$ LANGUAGE SQL STRICT;
```

在前两个规则系统描述中我们需要真实表是：

```
CREATE TABLE shoe_data (
    shoename    text,          -- 主键
    sh_avail    integer,      -- 可用的双数
    slcolor     text,          -- 首选的鞋带颜色
    slminlen    real,          -- 最小鞋带长度
    slmaxlen    real,          -- 最大鞋带长度
    slunit      text          -- 长度单位
);

CREATE TABLE shoelace_data (
    sl_name     text,          -- 主键
    sl_avail    integer,      -- 可用的双数
    sl_color    text,          -- 鞋带颜色
    sl_len      real,          -- 鞋带长度
    sl_unit     text          -- 长度单位
);

CREATE TABLE unit (
    un_name     text,          -- 主键
    un_fact     real          -- 转换到厘米的参数
);
```

如你所见，它们表示鞋店的数据。

视图被创建为：

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
    FROM shoe_data sh, unit un
    WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
```

```

        s.sl_avail,
        s.sl_color,
        s.sl_len,
        s.sl_unit,
        s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
    WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
    FROM shoe rsh, shoelace rsl
    WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm;

```

创建 `shoelace` 视图的 `CREATE VIEW` 命令（也是最简单的一个）将创建一个 `shoelace` 关系和一个 `sys_rewrite` 项，这个 `sys_rewrite` 项说明有一个重写规则，只要一个查询的范围表中引用了关系 `shoelace`，就必须应用它。该规则没有规则条件（稍后和非 `SELECT` 规则一起讨论，因为目前的 `SELECT` 规则不能有规则条件）并且它是 `INSTEAD` 规则。要注意规则条件与查询条件不一样。我们的规则的动作有一个查询条件。该规则的动作是一个查询树，这个查询是视图创建命令中的 `SELECT` 语句的一个拷贝。

**注意：** 你在 `sys_rewrite` 项中看到的两个额外的用于 `NEW` 和 `OLD` 的范围表项不是 `SELECT` 规则感兴趣的东西。

现在我们填充 `unit`、`shoe_data` 和 `shoelace_data`，并且在视图上运行一个简单的查询：

```

INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh1', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh2', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh3', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0, 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0, 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9, 'm');

```

```

INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----+-----+-----+-----+-----+-----
sl1     |         5 | black    | 80     | cm      | 80
sl2     |         6 | black    | 100    | cm      | 100
sl7     |         7 | brown    | 60     | cm      | 60
sl3     |         0 | black    | 35     | inch    | 88.9
sl4     |         8 | black    | 40     | inch    | 101.6
sl8     |         1 | brown    | 40     | inch    | 101.6
sl5     |         4 | brown    | 1      | m       | 100
sl6     |         0 | brown    | 0.9    | m       | 90
(8 rows)

```

这是你可以在我们的视图上做的最简单的 `SELECT`，所以我们用这次机会来解释视图规则的基本要素。`SELECT * FROM shoelace` 会被解析器解释并生成下面的查询树：

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM shoelace shoelace;

```

然后这将被交给规则系统。规则系统遍历范围表，检查有没有可用于任何关系的规则。在为 `shoelace`（到目前为止的唯一一个）处理范围表时，它会发现查询树里有 `_RETURN` 规则：

```

SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
FROM shoelace old, shoelace new,
     shoelace_data s, unit u
WHERE s.sl_unit = u.un_name;

```

要扩展该视图，重写器简单地创建一个子查询范围表项，它包含规则的动作的查询树，然后用这个范围表记录取代原来引用视图的那个。作为结果的重写后的查询树几乎与你键入的那个一样：

```

SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
FROM (SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,

```

```

        s.sl_unit,
        s.sl_len * u.un_fact AS sl_len_cm
    FROM shoelace_data s, unit u
    WHERE s.sl_unit = u.un_name) shoelace;

```

不过有一个区别：子查询的范围表有两个额外的项 `shoelace old` 和 `shoelace new`。这些项并不直接参与到查询中，因为它们没有被子查询的连接树或者目标列表引用。重写器用它们存储最初出现在引用视图的范围表项中表达的访问权限检查信息。以这种方式，执行器仍然会检查该用户是否有访问视图的正确权限，尽管在重写后的查询中没有对视图的直接使用。

这是被应用的第一个规则。规则系统将检查顶层查询里剩下的范围表项（本例中没有了），并且它将递归的检查增加的子查询中的范围表项，看看其中有没有引用视图的（不过这样不会扩展 `old` 或 `new` 一否则我们会得到无限递归！）。在这个例子中，没有用于 `shoelace_data` 或 `unit` 的重写规则，所以重写结束并且上面得到的就是给规划器的最终结果。

现在我们想写一个查询，它找出目前在店里哪些鞋子有匹配的（颜色和长度）鞋带并且完全匹配的鞋带双数大于等于二。

```

SELECT * FROM shoe_ready WHERE total_avail >= 2;

```

shoename	sh_avail	sl_name	sl_avail	total_avail
sh1	2	sl1	5	2
sh2	4	sl7	7	4

(2 rows)

这个词解析器的输出是查询树：

```

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM shoe_ready shoe_ready
WHERE shoe_ready.total_avail >= 2;

```

第一个被应用的规则将是用于 `shoe_ready` 的规则并且它会导致查询树：

```

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
      WHERE rsl.sl_color = rsh.slcolor

```

```

        AND rsl.sl_len_cm >= rsh.slminlen_cm
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail >= 2;

```

相似地，用于 shoe 和 shoelace 的规则被替换到子查询的范围表中，得到一个三层的最终查询树：

```

SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
FROM (SELECT rsh.shoename,
            rsh.sh_avail,
            rsl.sl_name,
            rsl.sl_avail,
            min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM (SELECT sh.shoename,
                  sh.sh_avail,
                  sh.slcolor,
                  sh.slminlen,
                  sh.slminlen * un.un_fact AS slminlen_cm,
                  sh.slmaxlen,
                  sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                  sh.slunit
            FROM shoe_data sh, unit un
            WHERE sh.slunit = un.un_name) rsh,
          (SELECT s.sl_name,
                  s.sl_avail,
                  s.sl_color,
                  s.sl_len,
                  s.sl_unit,
                  s.sl_len * u.un_fact AS sl_len_cm
            FROM shoelace_data s, unit u
            WHERE s.sl_unit = u.un_name) rsl
      WHERE rsl.sl_color = rsh.slcolor
        AND rsl.sl_len_cm >= rsh.slminlen_cm
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
WHERE shoe_ready.total_avail > 2;

```

最后规划器会把这个树折叠成一个两层查询树：最下层的 SELECT 命令将被“提升”到中间的 SELECT 中，因为没有必要分别处理它们。但是中间的 SELECT 仍然和顶层的分开，因为它包含聚集函数。如果我们把它们也提升，它将改变顶层 SELECT 的行为，这不是我们想要的。不过，折叠查询树是一种优化，重写系统不需要关心它。

## 21.2.2 非 SELECT 语句中的视图规则

有两个查询树的细节在上面的视图规则的描述中没有涉及。它们是命令类型和结果关系。实际上，视图规则不需要命令类型，但是结果关系可能会影响查询重写器工作的方式，因为如果结果关系是一个视图，我们需要采取特殊的措施。

一个 `SELECT` 的查询树和其它命令的查询树之间很少的几处不同。显然，它们有不同的命令类型并且对于 `SELECT` 之外的命令，结果关系指向结果将进入的范围表项。其它所有东西都完全相同。所以如果有两个表 `t1` 和 `t2` 分别有列 `a` 和 `b`，下面两个语句的查询树：

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

几乎是一样的。特别是：

- 范围表包含表 `t1` 和 `t2` 的项。
- 目标列表包含一个变量，该变量指向表 `t2` 的范围表项的列 `b`。
- 条件表达式比较两个范围表项的列 `a` 以寻找相等。
- 连接树展示了 `t1` 和 `t2` 之间的一次简单连接。

结果是，两个查询树生成相似的执行计划：它们都是两个表的连接。对于 `UPDATE` 语句，规划器把 `t1` 缺失的列加到目标列并且最终查询树读起来是：

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

因此在连接上运行的执行器将产生完全相同的结果集：

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

但是在 `UPDATE` 中有个小问题：执行器计划中执行连接的部分不关心连接的结果的含义。它只是产生一个行的结果集。一个是 `SELECT` 命令而另一个是由执行器中的更高层处理的 `UPDATE` 命令，在那里执行器知道这是一个 `UPDATE`，并且它知道这个结果应该进入表 `t1`。但是这里的哪些行必须被新行替换呢？

要解决这个问题，在 `UPDATE` 和 `DELETE` 语句的目标列表里面增加了另外一个项：当前元组 ID (CTID)。这是一个系统列，它包含行所在的文件块编号和在块中的位置。在已知表的情况下，CTID 可以被用来检索要被更新的 `t1` 的原始行。在添加 CTID 到目标列之后，该查询实际看起来像：

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

现在，另一个 KingbaseES 的细节进入到这个阶段了。表中的旧行还没有被覆盖，这就是为什么 `ROLLBACK` 很快的原因。在一个 `UPDATE` 中，新的结果行被插入到表中（在剥除 CTID 之后），并且把 CTID 指向的旧行的行头部中的 `cmx` 和 `xmx` 项设置为当前命令计数器和当前事务 ID。这样旧的行就被隐藏起来，并且在事务提交之后 `vacuum` 清理器就可以最终移除死亡的行。

知道了所有这些，我们就可以用完全相同的方式简单地把视图规则应用到任意命令中。没有任何区别。



### 21.2.3 KingbaseES 中视图的能力

上文演示了规则系统如何把视图定义整合到原始的查询树中。在第二个例子中，一个来自于一个视图的简单 SELECT 创建了一个四表连接（unit 以不同的名字被用了两次）的最终查询树。

用规则系统实现视图的好处是，规划器拥有关于哪些表必须被扫描、这些表之间的联系、来自于视图的限制性条件、一个单一查询树中原始查询的条件等所有信息。当原始查询已经是一个视图上的连接时仍然是这样。规划器必须决定执行查询的最优路径，而且规划器拥有越多信息，该决定就越好。并且 KingbaseES 中实现的规则系统保证这些信息是此时能获得的有关该查询的所有信息。

### 21.2.4 更新一个视图

如果视图是 INSERT、UPDATE 或 DELETE 的目标关系会怎样？使用上文所述的替换将给出一个查询树，其中的结果关系指向一个子查询范围表项，这样无法工作。不过，KingbaseES 中有几种方法来支持更新视图。

如果子查询从一个单一基本关系选择并且该关系足够简单，重写器会自动地把该子查询替换成底层的基本关系，这样 INSERT、UPDATE 或 DELETE 会被以适当的方式应用到该基本关系。其中“足够简单”的视图被称为自动可更新。有关这种可以被自动更新的视图类别的详细信息，请见 CREATE VIEW。

或者，该操作可以被定义在视图上的一个用户提供的 INSTEAD OF 触发器处理。在这种情况下重写工作有一点点不同。对于 INSERT，重写器对视图什么也不做，让它作为查询的结果关系。对于 UPDATE 和 DELETE，仍有必要扩展该视图查询来产生命令将尝试更新或删除的“旧”行。因此该视图被按照通常的方式扩展，但是另一个未被扩展的范围表项会被增加到查询来表示该视图会尽其所能作为结果关系。

现在出现的问题是如何标识在视图中要被更新的行。回忆一下，当结果关系是一个表时，一个特殊的 CTID 项会被加入到目标列表来标识要被更新的行的物理位置。如果结果关系是一个视图这就行不通，因为一个视图根本就没有 CTID，它的行没有实际的物理位置。对于一个 UPDATE 或 DELETE 操作，一个特殊的 wholerow 项会被增加到目标列表中，它会扩展来包括来自该视图的所有列。执行器使用这个值来提供“旧”行给 INSTEAD OF 触发器。现在就轮到触发器来基于新旧行值来找出要更新什么了。

另外一种可能性是让用户定义 INSTEAD 规则，这种规则指定对视图上的 INSERT\UPDATE 和 DELETE 命令的替代动作。这些规则将重写该命令，通常是重写成一个更新一个或多个表（而不是视图）的命令。这是 INSERT、UPDATE 和 DELETE 上的规则的主题。

注意规则会首先被计算，然后在原始查询被规划和执行之前重写它。因此，如果一个视图上同时有 INSTEAD OF 触发器和 INSERT、UPDATE 或 DELETE 规则，那么首先会计算规则，然后根据其结果决定是否执行触发器，触发器可能完全都不会被使用。

在一个简单视图上的 INSERT、UPDATE 或 DELETE 查询的自动重写总是在最后尝试。因此，如果一个视图有规则或触发器，它们将重载自动可更新视图的默认行为。

如果对该视图没有 INSTEAD 规则或 INSTEAD OF 触发器，并且重写器不能自动地把该查询重写成一个底层基本关系上的更新，将会抛出一个错误，因为执行器不能更新一个这样的视图。

## 21.3 物化视图

KingbaseES 中的物化视图像视图一样使用了规则系统，但是以一种类表的形式保留了结果。在物化视图：

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

和视图：

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

之间的主要区别是物化视图不能被更新，并且用于创建物化视图的查询的存储方式和视图查询的存储方式完全相同，因此要为物化视图生成新鲜的数据：

```
REFRESH MATERIALIZED VIEW mymatview;
```

有关一个 KingbaseES 系统目录中的物化视图的信息和一个表或视图的信息完全相同。因此对于解析器，一个物化视图就是一个关系，就像一个表或一个视图。当一个物化视图被一个查询引用时，数据直接从物化视图中返回，如同表一样；规则只被用来填充物化视图。

虽然对物化视图中存储的数据的访问常常要快于直接访问底层表或通过一个视图访问，但是数据并不总是最新的；但是某些时候并不需要当前数据。考虑一个记录销售情况的表：

```
CREATE TABLE invoice (
    invoice_no    integer          PRIMARY KEY,
    seller_no     integer,         -- 销售员的 ID
    invoice_date  date,           -- 销售日期
    invoice_amt   numeric(13,2)   -- 销售量
);
```

如果人们想快速绘制历史销售数据，他们可能希望汇总，并且他们可能并不关心当前日期的不完整数据：

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT
    seller_no,
    invoice_date,
    sum(invoice_amt)::numeric(13,2) as sales_amt
FROM invoice
WHERE invoice_date < CURRENT_DATE
GROUP BY
    seller_no,
    invoice_date
ORDER BY
    seller_no,
    invoice_date;
```

```
CREATE UNIQUE INDEX sales_summary_seller
ON sales_summary (seller_no, invoice_date);
```

这个物化视图可能对在为销售员创建的控制面板上显示一个图表非常有用。可以用一个计划任务在每晚使用这个 SQL 语句更新该统计信息：

```
REFRESH MATERIALIZED VIEW sales_summary;
```

物化视图的另一种使用是允许通过一个外部数据包装器对来自一个远程系统的数据进行更快的访问。下面有一个使用 `file_fdw` 的简单例子，但是由于本地系统上可以使用高速缓存，因此比起访问一个远程系统的性能差异可能会比这里所展示的更大。注意鉴于 `file_fdw` 不支持索引，我们也使用这种能力来在物化视图上放置索引。这种优势可能不适用于其他种类的外部数据访问。

建立：

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
CREATE FOREIGN TABLE words (word text NOT NULL)
SERVER local_file
OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION sys_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

现在让我们对一个词进行拼写检查。直接使用 `file_fdw`：

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

count
-----
      0
(1 row)
```

通过 `EXPLAIN ANALYZE`，我们可以看到：

```
Aggregate (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181
rows=1 loops=1)
-> Foreign Scan on words (cost=0.00..21761.41 rows=1032 width=0) (actual
time=188.177..188.177 rows=0 loops=1)
    Filter: (word = 'caterpiler'::text)
    Rows Removed by Filter: 479829
    Foreign File: /usr/share/dict/words
    Foreign File Size: 4953699
Planning time: 0.118 ms
```

```
Execution time: 188.273 ms
```

如果使用物化视图，该查询会快很多：

```
Aggregate (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1
loops=1)
  -> Index Only Scan using wrd_word on wrd (cost=0.42..4.44 rows=1 width=0)
      (actual time=0.039..0.039 rows=0 loops=1)
          Index Cond: (word = 'caterpiler'::text)
          Heap Fetches: 0
Planning time: 0.164 ms
Execution time: 0.117 ms
```

不管哪种方式，单词都是被拼错的，因此让我们看看什么是我们可能想要的。再次使用 `file_fdw`：

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;
```

```
   word
-----
cater
caterpillar
Caterpillar
caterpillars
caterpillar's
Caterpillar's
caterer
caterer's
caters
catered
(10 rows)
```

```
Limit (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594
rows=10 loops=1)
  -> Sort (cost=11583.61..11804.76 rows=88459 width=32) (actual
      time=1431.589..1431.591 rows=10 loops=1)
          Sort Key: ((word <-> 'caterpiler'::text))
          Sort Method: top-N heapsort  Memory: 25kB
          -> Foreign Scan on words (cost=0.00..9672.05 rows=88459 width=32)
              (actual time=0.057..1286.455 rows=479829 loops=1)
                  Foreign File: /usr/share/dict/words
                  Foreign File Size: 4953699
Planning time: 0.128 ms
Execution time: 1431.679 ms
```

使用物化视图：

```

Limit (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257
rows=10 loops=1)
  -> Index Scan using wrd_trgm on wrd (cost=0.29..37020.87 rows=479829
width=10) (actual time=187.219..188.252 rows=10 loops=1)
    Order By: (word <-> 'caterpiler'::text)
Planning time: 0.196 ms
Execution time: 198.640 ms

```

如果你能够忍受定期把远程数据更新到本地数据库，其性能收益可能是巨大的。

## 21.4 INSERT、UPDATE 和 DELETE 上的规则

定义在 INSERT、UPDATE 和 DELETE 上的规则与前一节描述的视图规则有明显的不同。

首先，它们的 CREATE RULE 命令允许更多：

- 它们可以没有动作。
- 它们可以有多个动作。
- 它们可以是 INSTEAD 或 ALSO（缺省）。
- 伪关系 NEW 和 OLD 变得有用了。
- 它们可以有规则条件。

第二，它们不是就地修改查询树，而是创建零个或多个新查询树并且可能把原始的那个查询树扔掉。

Caution: 在很多情况下，由 INSERT/UPDATE/DELETE 上的规则执行的任务用触发器能做得更好。触发器在记法上要更复杂些，但是它们的语义理解起来更简单些。当原始查询包含不稳定函数时，规则容易产生令人惊讶的结果：在执行规则的过程中不稳定函数的执行次数可能比语气中的更多。

还有，有些情况根本无法用这些类型的规则支持，典型的是在原始查询中包括 WITH 子句以及在 UPDATE 查询的 SET 列表中包括多个赋值的子 SELECT。这是因为把这些结构复制到一个规则查询中可能导致子查询的多次计算，这与查询作者表达的意图相悖。

### 21.4.1 更新规则如何工作

记住以下语法：

```

CREATE [ OR REPLACE ] RULE name AS ON event
TO table [ WHERE condition ]
DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }

```

在随后的内容中，更新规则表示定义在 INSERT、UPDATE 或 DELETE 上的规则。

如果查询树的结果关系和命令类型等于 CREATE RULE 命令中给出的对象和事件，规则系统就会应用更新规则。对于更新规则，规则系统会创建一个查询树列表。一开始该查询树列表是空的。更新规则中可以有零个（NOTHING 关键字）、一个或多个动作。为简单起见，我们先看一个只有一个动作的规则。这个规则可以有条件或者没有条件，并且它可以是 INSTEAD 或 ALSO（缺省）。

什么是规则条件？它是一个限制，告诉规则动作什么时候做、什么时候不做。这个条件只能引用 NEW 和/或 OLD 伪关系，它们基本上代表作为对象给定的关系（但是有着特殊含义）。

所以，对这个单动作的规则生成下面的查询树，我们有三种情况。

- 没有条件，有 ALSO 或 INSTEAD

来自规则动作的查询树，在其上增加原始查询树的条件

- 给出了条件，有 ALSO

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件

- 给出了条件，有 INSTEAD

来自规则动作的查询树，在其上加入规则条件和原始查询树的条件；以及带有反规则条件的原始查询树

最后，如果规则是 ALSO，那么未修改的原始查询树也被加入到列表。因为只有合格的 INSTEAD 规则已经被加入到原始查询树中，对于单动作的规则，我们将结束于一个或两个输出查询树。

对于 ON INSERT 规则，原始查询（如果没有被 INSTEAD 取代）是在任何规则增加的动作之前完成的。这样就允许动作看到被插入的行。但是对 ON UPDATE 和 ON DELETE 规则，原始查询是在规则增加的动作之后完成的。这样就确保动作可以看到将要更新或者将要删除的行；否则，动作可能什么也不做，因为它们无法发现符合它们要求的行。

从规则动作生成的查询树会被再次丢给重写系统，并且可能有更多规则被应用而得到更多或更少的查询树。所以，一个规则的动作必须有一种不同的命令类型或者和规则所在的关系不同的另一个结果关系。否则这样的递归处理就会没完没了（规则的递归展开会被检测到，并当作一个错误报告）。

在 sys\_rewrite 系统目录中的动作中的查询树只是模板。因为它们可以引用 NEW 和 OLD 的范围表项，在使用它们之前必须做一些替换。对于任何 NEW 的引用，都要先在原始查询的目标列表中搜索对应的项。如果找到，该项的表达式将会替换该引用。否则 NEW 和 OLD 的含义一样（对于 UPDATE）或者被替换成一个空值（对于 INSERT）。任何对 OLD 的引用都用结果关系的范围表项的引用替换。

在系统完成应用更新规则后，它再应用视图规则到生成的查询树上。视图无法插入新的更新动作，所以没有必要向视图重写的输出应用更新规则。

### 21.4.1.1 第一个规则循序渐进

假设我们想要跟踪 shoelace\_data 关系中的 sl\_avail 列。所以我们建立一个日志表和一条规则，这条规则每次在 shoelace\_data 上执行 UPDATE 时有条件地写入一个日志项。

```
CREATE TABLE shoelace_log (
  sl_name    text,      -- 改变的鞋带
  sl_avail   integer,   -- 新的可用值
```

```

log_who    text,          -- 谁做的
log_when   timestamp      -- 何时做的
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
WHERE NEW.sl_avail <> OLD.sl_avail
DO INSERT INTO shoelace_log VALUES (
        NEW.sl_name,
        NEW.sl_avail,
        current_user,
        current_timestamp
);

```

现在有人做:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

然后看看日志表:

```
SELECT * FROM shoelace_log;
```

sl_name	sl_avail	log_who	log_when
sl7	6	Al	Tue Oct 20 16:14:45 1998 MET DST

(1 row)

这就是我们所期望的。在后台发生的的事情如下。解析器创建查询树:

```
UPDATE shoelace_data SET sl_avail = 6
FROM shoelace_data shoelace_data
WHERE shoelace_data.sl_name = 'sl7';
```

这是一个带有规则条件表达式的 ON UPDATE 规则 log\_shoelace, 条件是:

```
NEW.sl_avail <> OLD.sl_avail
```

它的动作是:

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old;
```

(这看起来有点奇怪, 因为你通常不能写 INSERT ... VALUES ... FROM。这里的 FROM 子句只是表示查询树里有用于 new 和 old 的范围表项。这些东西是必需的, 这样它们就可以被 INSERT 命令的查询树中的变量引用)。

该规则是一个有条件的 ALSO 规则，所以规则系统必须返回两个查询树：更改过的规则动作和原始查询树。在第 1 步里，原始查询的范围表被集成到规则动作的查询树中。得到：

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data;
```

第 2 步把规则条件增加进去，所以结果集被限制为 sl\_avail 改变了的行：

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail;
```

（这看起来更奇怪，因为 INSERT ... VALUES 也没有 WHERE 子句，但是规划器和执行器处理它没有任何难度。不管怎样，它们需要为 INSERT ... SELECT 支持这种相同功能）。

第 3 步把原始查询树的条件加进去，把结果集进一步限制成只有被初始查询树改变的行：

```
INSERT INTO shoelace_log VALUES (
    new.sl_name, new.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE new.sl_avail <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

第 4 步把 NEW 引用替换为来自原始查询树的目标列表项或来自结果关系的相匹配的变量引用：

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
    shoelace_data shoelace_data
WHERE 6 <> old.sl_avail
    AND shoelace_data.sl_name = 'sl7';
```

第 5 步，用结果关系引用把 OLD 引用替换掉：

```
INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data new, shoelace_data old,
```



```

shoelace_data shoelace_data
WHERE 6 <> shoelace_data.sl_avail
AND shoelace_data.sl_name = 's17';

```

这就完成了。因为规则是 ALSO，我们还要输出原始查询树。简而言之，从规则系统输出的是一个包含两个查询树的列表，它们与下面语句相对应：

```

INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, 6,
    current_user, current_timestamp )
FROM shoelace_data
WHERE 6 <> shoelace_data.sl_avail
AND shoelace_data.sl_name = 's17';

UPDATE shoelace_data SET sl_avail = 6
WHERE sl_name = 's17';

```

这些会按照这个顺序被执行，并且这也正是规则要做的事情。

做的替换和追加的条件用于确保对于下面这样的原始查询不会有日志记录被写入：

```

UPDATE shoelace_data SET sl_color = 'green'
WHERE sl_name = 's17';

```

在这种情况下，原始查询树不包含 sl\_avail 的目标列表项，因此 NEW.sl\_avail 将被 shoelace\_data.sl\_avail 代替。所以，规则生成的额外命令是：

```

INSERT INTO shoelace_log VALUES (
    shoelace_data.sl_name, shoelace_data.sl_avail,
    current_user, current_timestamp )
FROM shoelace_data
WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
AND shoelace_data.sl_name = 's17';

```

并且条件将永远不可能为真。

如果原始查询修改多个行，这也能争产工作。所以如果某人发出命令：

```

UPDATE shoelace_data SET sl_avail = 0
WHERE sl_color = 'black';

```

实际上有四行（s11、s12、s13 和 s14）被更新。但 s13 已经是 sl\_avail = 0。在这种情况下，原始查询树的条件不同并且导致规则产生额外的查询树：

```

INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
    current_user, current_timestamp

```

```

FROM shoelace_data
WHERE 0 <> shoelace_data.sl_avail
AND shoelace_data.sl_color = 'black';

```

这个查询树将肯定插入三个新的日志项。这也是完全正确的。

到这里我们就能明白为什么原始查询树最后执行非常重要。如果 UPDATE 先被执行，则所有的行都已经被设为零，所以记日志的 INSERT 将无法找到任何符合 0 <> shoelace\_data.sl\_avail 的行。

## 21.4.2 与视图合作

要保护一个视图关系不被 INSERT、UPDATE 或 DELETE，一种简单的方法是让那些查询树被丢掉。因此我们可以创建规则：

```

CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
DO INSTEAD NOTHING;

```

如果现在某人尝试对视图关系 shoe 做任何这些操作，规则系统将应用这些规则。因为这些规则没有动作而且是 INSTEAD，作为的查询树列表将是空的并且整个查询将变得什么也不做，因为经过规则系统处理后没有什么东西剩下下来被优化或执行了。

一个更好的使用规则系统的方法是创建一些规则，这些规则把查询树重写成一个在真实表上进行正确的操作的查询树。要在视图 shoelace 上做这件事，我们创建下列规则：

```

CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
DO INSTEAD
UPDATE shoelace_data
SET sl_name = NEW.sl_name,
    sl_avail = NEW.sl_avail,
    sl_color = NEW.sl_color,
    sl_len = NEW.sl_len,

```

```

        sl_unit = NEW.sl_unit
    WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
DO INSTEAD
DELETE FROM shoelace_data
WHERE sl_name = OLD.sl_name;

```

如果你要在视图上支持 RETURNING 查询，你需要让规则包含 RETURNING 子句来计算视图行。这对于基于单个表的视图来说通常非常简单，但是对于连接视图（如 shoelace）就有点冗长了。对于插入的一个例子：

```

CREATE RULE shoelace_ins AS ON INSERT TO shoelace
DO INSTEAD
INSERT INTO shoelace_data VALUES (
    NEW.sl_name,
    NEW.sl_avail,
    NEW.sl_color,
    NEW.sl_len,
    NEW.sl_unit
)
RETURNING
    shoelace_data.*,
    (SELECT shoelace_data.sl_len * u.un_fact
     FROM unit u WHERE shoelace_data.sl_unit = u.un_name);

```

注意，这个规则同时支持该视图上的 INSERT 和 INSERT RETURNING 查询—对于 INSERT 会简单地忽略 RETURNING 子句。

现在假设有时一包鞋带抵达了商店，并且随着它有一个大的清单。但是你不希望每次都手工更新 shoelace 视图。取而代之的是我们建立两个小表：一个用来从清单向其中插入东西，另一个则用了一个特殊的技巧。这些东西的创建命令如下：

```

CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
DO INSTEAD
UPDATE shoelace
SET sl_avail = sl_avail + NEW.ok_quant

```

```
WHERE sl_name = NEW.ok_name;
```

现在你可以用来自清单的数据填充表 `shoelace_arrive`:

```
SELECT * FROM shoelace_arrive;
```

arr_name	arr_quant
s13	10
s16	20
s18	20

(3 rows)

快速地看一看当前的数据:

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s13	0	black	35	inch	88.9
s14	8	black	40	inch	101.6
s18	1	brown	40	inch	101.6
s15	4	brown	1	m	100
s16	0	brown	0.9	m	90

(8 rows)

现在把到的货鞋带移到:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

并检查结果:

```
SELECT * FROM shoelace ORDER BY sl_name;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s14	8	black	40	inch	101.6
s13	10	black	35	inch	88.9
s18	21	brown	40	inch	101.6
s15	4	brown	1	m	100

```
sl6      |      20 | brown   |      0.9 | m      |      90
(8 rows)
```

```
SELECT * FROM shoelace_log;
```

```
sl_name | sl_avail | log_who | log_when
-----+-----+-----+-----
sl7     |      6  | A1     | Tue Oct 20 19:14:45 1998 MET DST
sl3     |     10  | A1     | Tue Oct 20 19:25:16 1998 MET DST
sl6     |     20  | A1     | Tue Oct 20 19:25:16 1998 MET DST
sl8     |     21  | A1     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

从一个 INSERT ... SELECT 到这些结果经过了很长的过程。并且该查询树转换的描述将出现在本章的最后。首先，这里是解析器的输出：

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

现在应用第一条规则 shoelace\_ok\_ins 被应用并且把这个输出转换成：

```
UPDATE shoelace
SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace
WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

并且丢掉 shoelace\_ok 上的 INSERT。这个被重写后的查询被再次传递给规则系统，并且第二个被应用的规则 shoelace\_upd 会产生：

```
UPDATE shoelace_data
SET sl_name = shoelace.sl_name,
sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
sl_color = shoelace.sl_color,
sl_len = shoelace.sl_len,
sl_unit = shoelace.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
shoelace_ok old, shoelace_ok new,
shoelace shoelace, shoelace old,
shoelace new, shoelace_data shoelace_data
WHERE shoelace.sl_name = shoelace_arrive.arr_name
AND shoelace_data.sl_name = shoelace.sl_name;
```

同样这是一个 INSTEAD 规则并且前一个查询树会被丢弃掉。注意这个查询仍然使用视图 shoelace。但是规则系

统还没有完成这一步，所以它会继续并在其上应用 `_RETURN` 规则，并且我们得到：

```
UPDATE shoelace_data
  SET sl_name = s.sl_name,
      sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
      sl_color = s.sl_color,
      sl_len = s.sl_len,
      sl_unit = s.sl_unit
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
     shoelace_ok old, shoelace_ok new,
     shoelace shoelace, shoelace old,
     shoelace new, shoelace_data shoelace_data,
     shoelace old, shoelace new,
     shoelace_data s, unit u
WHERE s.sl_name = shoelace_arrive.arr_name
      AND shoelace_data.sl_name = s.sl_name;
```

最后，规则 `log_shoelace` 被应用，生成额外的查询树：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
     shoelace_ok old, shoelace_ok new,
     shoelace shoelace, shoelace old,
     shoelace new, shoelace_data shoelace_data,
     shoelace old, shoelace new,
     shoelace_data s, unit u,
     shoelace_data old, shoelace_data new
     shoelace_log shoelace_log
WHERE s.sl_name = shoelace_arrive.arr_name
      AND shoelace_data.sl_name = s.sl_name
      AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

完成这些之后，规则系统用完了所有的规则并且返回生成的查询树。

所以我们结束于两个最终查询树，它们等效于 SQL 语句：

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
     shoelace_data s
```

```

WHERE s.sl_name = shoelace_arrive.arr_name
      AND shoelace_data.sl_name = s.sl_name
      AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;

UPDATE shoelace_data
      SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
FROM shoelace_arrive shoelace_arrive,
      shoelace_data shoelace_data,
      shoelace_data s
WHERE s.sl_name = shoelace_arrive.sl_name
      AND shoelace_data.sl_name = s.sl_name;

```

结果是从一个关系来的数据插入了到另一个中，改变成第三个上的更新，改变成更新第四个外加做日志，在第五个中的最后更新缩减为两个查询。

有一个小细节有点丑陋。看看那两个查询，我们会发现 `shoelace_data` 关系在范围表中出现了两次而实际上绝对可以缩为出现一次。规划器不会处理它，因此 `INSERT` 的规则系统输出的执行规划会是

```

Nested Loop
-> Merge Join
    -> Seq Scan
        -> Sort
            -> Seq Scan on s
    -> Seq Scan
        -> Sort
            -> Seq Scan on shoelace_arrive
-> Seq Scan on shoelace_data

```

在省略额外的范围表项后会得到

```

Merge Join
-> Seq Scan
    -> Sort
        -> Seq Scan on s
-> Seq Scan
    -> Sort
        -> Seq Scan on shoelace_arrive

```

这在日志表中生成完全一样的项。因此，规则系统导致了 `shoelace_data` 表上的一次绝对不必要的扫描。并且同样的冗余扫描会在 `UPDATE` 中进行。但是要把这些全部实现实在是一项很困难的工作。

现在我们对 KingbaseES 规则系统及其能力做最后一个演示。假设你向你的数据库中添加一些有特别颜色的鞋带：

```

INSERT INTO shoelace VALUES ('s19', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('s110', 1000, 'magenta', 40.0, 'inch', 0.0);

```

我们想要建立一个视图来检查哪些 shoelace 项在颜色上不配任何鞋子。适用的视图是：

```
CREATE VIEW shoelace_mismatch AS
SELECT * FROM shoelace WHERE NOT EXISTS
    (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

它的输出是：

```
SELECT * FROM shoelace_mismatch;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s19	0	pink	35	inch	88.9
s110	1000	magenta	40	inch	101.6

现在我们要建立它，这样没有库存的不匹配的鞋带都会被从数据库中删除。为了对 KingbaseES 有点难度，我们不直接删除它们。而是我们再创建一个视图：

```
CREATE VIEW shoelace_can_delete AS
SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

然后用下面方法：

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
     WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;
```

sl_name	sl_avail	sl_color	sl_len	sl_unit	sl_len_cm
s11	5	black	80	cm	80
s12	6	black	100	cm	100
s17	6	brown	60	cm	60
s14	8	black	40	inch	101.6
s13	10	black	35	inch	88.9
s18	21	brown	40	inch	101.6
s110	1000	magenta	40	inch	101.6
s15	4	brown	1	m	100
s16	20	brown	0.9	m	90

(9 rows)

对一个视图上的 DELETE，这个命令带有一个总共使用了四个嵌套/连接视图的子查询条件，这四个视图之一本身有一个包含一个视图的子查询条件，该条件计算使用的视图列；这个命令被重写成了一个查询树，该查询树从一个真正的表里面把需要删除的数据删除。



在现实世界里只有很少的情况需要上面的这样的构造。但这些东西能运转肯定让你感觉不错。

## 21.5 规则和权限

由于 KingbaseES 规则系统对查询的重写，会访问没有在原始查询中指定的表/视图。使用更新规则时，这可能包括对表的写权限。

重写规则并不拥有一个独立的所有者。关系（表或视图）的所有者自动成为为其所定义的重写规则的所有者。KingbaseES 规则系统改变了默认的访问控制系统的行为。由于规则被使用的关系会按照规则所有者的权限来检查，而不是调用规则的用户。这表示用户只需要在其查询中显式指定的表/视图上的所需权限。

例如：某用户有一个电话号码列表，其中一些是私人的，另外的一些是办公室助理需要的。该用户可以构建下面的东西：

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO assistant;
```

除了该用户以外（还有数据库超级用户）没有人可以访问 `phone_data` 表。但因为 `GRANT` 的原因，助理可以在 `phone_number` 视图上运行 `SELECT`。规则系统将把 `phone_number` 上的 `SELECT` 重写为 `phone_data` 上的 `SELECT`。因为该用户是 `phone_number` 的所有者，因此也是规则的所有者，对 `phone_data` 的读访问现在被根据该用户的权限检查，并且该查询被允许。同时也要检查访问 `phone_number` 的权限，但这是针对调用用户进行的，所以除了用户自己和助理外没有人可以使用它。

权限检查是按规则逐条进行的。所以此时助理是唯一的一个可以看到公共电话号码的人。但助理可以建立另一个视图并且赋予该视图公共权限。这样，任何人都可以通过助理的视图看到 `phone_number` 数据。助理不能做的事情是创建一个直接访问 `phone_data` 的视图（实际上助理是可以的，但没有任何作用，因为每次访问都会因通不过权限检查而被否定）。而且该用户一旦注意到助理开放了他的 `phone_number` 视图，该用户还可以收回助理的访问权限。立刻，所有对助理视图的访问将会失败。

有人可能会认为这种逐条规则的检查是一个安全漏洞，但事实上不是。如果这样做不能奏效，助理将必须建立一个与 `phone_number` 有相同列的表并且每天拷贝一次数据进去。那么这是助理自己的数据因而助理可以为每一个想要访问的人授权。一个 `GRANT` 意味着“我信任你”。如果某个你信任的人做了上面的事情，那么是时候认为信任已经结束并且要使用 `REVOKE`。

需要注意的是，虽然视图可以用前文展示的技术来隐藏特定列的内容，它们不能可靠地在不可见行上隐藏数据，除非标志被设置。例如，下面的视图是不安全的：

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

这个视图看起来是安全的，因为规则系统会把任何 `phone_number` 上的 `SELECT` 重写成 `phone_data` 上的 `SELECT`，并且增加限制使得只有 `phone` 不以 412 开头的项才被处理。但是如果用户可以创建自己的函数，那就不难

让规划器在 NOT LIKE 表达式之前先执行用户自定义函数。例如：

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END
$$ LANGUAGE plsql COST 0.000000000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

phone\_data 表中的每一个人和电话号码会被打印成一个 NOTICE，因为规划器会选择在执行 NOT LIKE 之前先执行 tricky，因为前者的开销大。即使禁止用户自定义一个新函数，内置函数也可以用在类似的攻击中（例如，大部分造型函数会在它们产生的错误信息中包含它们的输入值）。

类似的考虑应用于更新规则。在前一节例子中，例子数据库中表的所有者可以把 shoelace 视图上的 SELECT、INSERT、UPDATE 和 DELETE 权限授予其他人，但对 shoelace\_log 只有 SELECT 权限。写日志项的规则动作将仍然可以被成功地执行，并且其它用户可以看到日志项。但他们不能创建伪造的项，并且他们也不能操纵或删除现有的项。在这种情况下，不可能通过让规划器改变操作的顺序来推翻规则，因为引用 shoelace\_log 的唯一规则是无限制的 INSERT。在更复杂的情景中，这可能不正确。

当需要对一个视图提供行级安全时，security\_barrier 属性应该被应用到该视图。这会阻止恶意选择的函数和操作符通过行被传递，直到视图完成其工作。例如，如果前文所示的视图被创建成这样，它就是安全的：

```
CREATE VIEW phone_number WITH (security_barrier) AS
SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

使用 security\_barrier 创建的视图的性能会远差于没有使用该选项的视图。通常，没有办法来避免这种现状：如果最快的候选计划可能在安全性上折衷，它就必须被拒绝。出于该原因，这个选项在默认情况下是没有启用的。

当处理没有副作用的函数时，查询规划器有更多的灵活性。这类函数被称为 LEAKPROOF，并且包括很多简单常用的操作符，例如很多等于操作符。查询规划器可以安全地允许这类函数在查询执行过程中的任何点被计算，因为在用户不可见的行上调用它们将不会泄露关于不可见行的任何信息。更进一步，不接收参数或者不从安全屏障视图得到任何参数的函数不必被标记为 LEAKPROOF 以便被下推，因为它们从来不会从该视图接收数据。相反，一个可能会基于接收到的参数值抛出错误的函数（例如在溢出或被零除事件中抛出错误的函数）不是防泄漏的，并且如果它被应用在安全性视图的行过滤器之前，它可能会提供有关不可见行的有效信息。

有一点很重要是理解：即使一个视图使用 security\_barrier 选项创建，它也只在不可见元组不会被传递给可能不安全的函数的前提下才是安全的。用户可能也有其他方式来推断不可见数据；例如，他们可以使用 EXPLAIN 看到查询计划，或者针对视图来测量查询的运行时间。一个恶意攻击者可能有能力推断有关不可见数据的总量，或者甚至得到有关数据分布的某些信息或最常用值（因为这些东西可以影响计划的运行时间；或者甚至计划的选择，因为它们也被反映在优化器的统计数据中）。如果这类“隐通道”攻击很重要，那么授予任何到该数据的访问都可能是不明智的。

## 21.6 规则和命令状态

KingbaseES 服务器为它收到的每个命令返回一个命令状态字符串，例如 `INSERT 149592 1`。没有涉及规则时这很简单，但是查询被规则重写时会发生什么呢？

规则对命令状态的影响如下：

- 如果没有查询的无条件 `INSTEAD` 规则，那么原始给出的查询将会被执行，并且它的命令状态将像平常一样被返回（但是请注意如果存在任何有条件 `INSTEAD` 规则，那么它们的反条件将被加到原始查询中。这样可能会减少它处理的行数，并且报告的状态将受影响）。
- 如果有查询的任何无条件 `INSTEAD` 规则，那么原始查询将完全不被执行。在这种情况下，服务器将返回由服务器将返回由 `INSTEAD` 规则（有条件的或无条件的）插入的最后一条和原始查询命令类型（`INSERT`、`UPDATE` 或 `DELETE`）相同的查询的命令状态。如果任何规则添加的查询都不符合这些要求，那么返回的命令状态显示原始查询类型并且行计数和 `OID` 域为零。

通过为任何想要的 `INSTEAD` 规则指定在活动规则中排名最后的规则名，程序员可以确保该规则都是在第二种情况里设置命令状态的规则，因为它会被最后一个应用。

## 21.7 规则 VS 触发器

许多触发器可以干的事情同样也可以用 KingbaseES 规则系统来实现。目前不能用规则来实现的东西之一是某些约束，特别是外键。可以放置一个合格的规则在一列上，这个规则在列的值没有出现在另一个表中时把命令重写成 `NOTHING`。但是这样做数据就会被不声不响地丢弃，因此也不是一个好主意。如果要求检查值的有效性，并且在出现无效值的情况下应该生成一个错误消息，这种需求就必须要用触发器来完成。

在本章中，我们关注于使用规则来更新视图。本章中所有的更新规则的例子都可以使用视图上的 `INSTEAD OF` 触发器来实现。编写这类触发器通常比编写规则要容易，特别是在要求使用复杂逻辑来执行更新的情况下。

对于两者都可实现的情况，哪个更好取决于对数据库的使用。触发器为每一个受影响的行都执行一次。规则修改查询树或生成一个额外的查询。所以如果在一个语句中影响到很多行，一个发出额外查询的规则通常可能会比一个触发器快，因为触发器对每一个行都要被调用，并且每次被调用时都需要重新判断要做什么样的操作。不过，触发器方法从概念上要远比规则方法简单，并且很容易让新人上手。

下面我们展示一个例子，该例子说明了在同种情况下两种选择的比较。这里有两个表：

```
CREATE TABLE computer (
    hostname      text,    -- 被索引
    manufacturer  text    -- 被索引
);

CREATE TABLE software (
    software      text,    -- 被索引
    hostname      text    -- 被索引
);
```

两个表都有数千行，并且在 `hostname` 上的索引是唯一的。规则或触发器应该实现一个约束，该约束从 `software` 中删除引用已删除计算机的行。触发器可以用下面这条命令：

```
DELETE FROM software WHERE hostname = $1;
```

因为触发器会为每一个从 `computer` 中删除的独立行调用一次，那么它可以准备并且保存这个命令的规划，把 `hostname` 作为参数传入。规则应该被写为：

```
CREATE RULE computer_del AS ON DELETE TO computer
DO DELETE FROM software WHERE hostname = OLD.hostname;
```

现在看看不同类型的删除。在这种情况下：

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

表 `computer` 被使用索引（快速）扫描，并且由触发器发出的命令也将使用一个索引扫描（同样快速）。来自规则的额外查询应该是：

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
AND software.hostname = computer.hostname;
```

由于已经建立了合适的索引，规划器将创建一个规划

```
Nestloop
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

所以在触发器和规则的实现之间没有太多的速度差别。

在接下来的删除中，我们想要去掉所有 2000 个 `hostname` 以 `old` 开头的计算机。有两个命令可以来做这件事。一个是：

```
DELETE FROM computer WHERE hostname >= 'old'
AND hostname < 'ole'
```

被规则增加的命令将是：

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname
< 'ole'
AND software.hostname = computer.hostname;
```

计划是：

```
Hash Join
-> Seq Scan on software
-> Hash
-> Index Scan using comp_hostidx on computer
```

另一个可能的命令是：

```
DELETE FROM computer WHERE hostname ~ '^old';
```

它会为规划增加的命令产生下面的执行计划：

```
Nestloop
-> Index Scan using comp_hostidx on computer
-> Index Scan using soft_hostidx on software
```

这表明，当有多个条件表达式被使用 AND 组合在一起时，规划器不能认识到表 `computer` 中 `hostname` 上的条件也可以被用于一个 `software` 上的索引扫描，而在该命令的正则表达式版本中正是这样做的。触发器将为要被删除的 2000 个旧计算机中的每一个调用，并且会导致在 `computer` 上的一次索引扫描和 `software` 上的 2000 次索引扫描。采用规则的实现将会使用两个使用索引的命令来完成。并且在顺序扫描情况下规则是否仍将更快是取决于 `software` 表的总体大小的。即使所有的索引块都将很快地进入高速缓存，通过 SPI 管理器执行来自触发器的 2000 个命令也要花不少时间。

我们要看的最后一个命令是：

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

同样，这也会导致很多行被从 `computer` 中删除。所以触发器同样会通过执行器运行很多命令。规则生成的命令将会是：

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
AND software.hostname = computer.hostname;
```

这个命令的计划又将是在两个索引扫描上的嵌套循环，只不过使用了 `computer` 上的另一个索引：

```
Nestloop
-> Index Scan using comp_manufidx on computer
-> Index Scan using soft_hostidx on software
```

在任何这些情况之一，来自规则系统的额外命令都或多或少与命令中影响的行数无关。

概括来说，规则只有在其动作导致了大而且糟糕的条件连接时才会明显地慢于触发器，这种情况下规划器将没有什么办法。

# 第 22 章 管理表/索引膨胀

本章节包含以下内容:

- 识别表/索引膨胀
- 处理表/索引膨胀

## 22.1 识别表/索引膨胀

### 22.1.1 查询表的更新量

动态视图 `sys_stat_all_tables` 中包括了表的修改量、行数; 被自动、手动清理的时间。可以通过这个视图评估系统中更新量大的表用于制定更新计划。例如获取系统中修改量最多的表信息:

```
SELECT schemaname, relname, n_tup_upd, n_tup_del, n_tup_hot_upd, n_live_tup, n_dead_tup, last_vacuum, last_
autovacuum FROM sys_stat_all_tables ORDER BY (n_tup_upd + n_tup_del) DESC;
```

视图具体含义参考《KingbaseES 数据库参考手册》中 动态性能视图一节。

### 22.1.2 查询表/索引的膨胀率

可以通过 `sys_stat_all_tables` 视图获取表中旧版本的比例:

```
SELECT schemaname || '.' || relname as table_name,
sys_size_pretty(sys_relation_size(schemaname || '.' || relname)) as table_size,
n_dead_tup,
n_live_tup,
round(n_dead_tup::numeric * 100 / (n_live_tup + n_dead_tup),2) as dead_tup_ratio,
round((n_live_tup::numeric + n_dead_tup) / n_live_tup ,2) as tblratio
FROM
sys_stat_all_tables
```

```
WHERE n_dead_tup >= 1000 AND n_live_tup !=0
ORDER BY dead_tup_ratio DESC;
```

注意：当数据库重启等操作后，表的统计信息此时不准确，此时 `n_dead_tup`、`n_live_tup` 等值默认为 0，此时需要执行 `analyze` 或者对表有读写业务。

在某些表、索引的容量异常增加时可以进一步通过以下函数确认空间情况：

```
SELECT * FROM kbstattuple_approx('表名'::regclass);
SELECT * FROM kbstatindex('索引名'::regclass);
```

函数返回值具体含义参考《KingbaseES 插件参考手册》中 `kbstattuple` 一节。

## 22.2 处理表/索引膨胀

### 22.2.1 清理

清理是防止表、索引膨胀的首选方式。

清理可以回收旧版本、一定条件下回收磁盘空间。注意，清理同时可以更新统计信息、`vm`、避免事务号，这部分的相关配置将不在本文档中描述。

清理方式包括通过 `autovacuum` 后台进程发起的自动清理和手动/通过各类脚本或定时任务执行 `VACUUM` 的命令。

清理的目标是使需要清理的表/索引都能得到及时清理，后续的修改可以重用空闲空间，从而把容量维持在一定大小。手动或定时的执行 `VACUUM` 不容易应对业务预期外的波动，所以在规划清理配置时优先选择通过 `autovacuum` 方式达成目标，辅助 `VACUUM` 命令处理配置不好覆盖的部分表/索引。

#### 22.2.1.1 Autovacuum

以下描述中涉及自动清理相关的参数配置，各个参数含义参考手册《KingbaseES 数据库参考手册》- 服务器配置参数 - 自动清理。

##### 22.2.1.1.1 参数配置需要考虑的因素

`autovacuum` 可能无法达成预期清理目标的原因如下，设计配置时应考虑这些因素：

1. 目标对象清理没有按预期启动
  - 1) 错误的 `threshold` 相关参数配置

2) 清理速度慢, autovacuum worker 被占用 a. 没有充分利用系统资源造成的清理速度慢 b. 系统资源瓶颈造成的清理速度慢

## 2. 清理没达到预期效果

1) 元组无法回收 a. 存在长事务 b. 清理时页面被并发访问 pin

2) 热点表业务压力过大

### 22.2.1.1.2 推荐配置

配置和问题的对应关系表

问题/配置设计	配置 1	配置 2	配置 3	配置 4	配置 5	配置 6
1-1)	√	√				
1-2)-a			√			
1-2)-b			√			
2-1)-a						
2-1)-b					√	
2-2)				√		
通用						√

#### 1. 开启 autovacuum, 保证自动清理频率

```
autovacuum = on
track_counts = on
autovacuum_naptime = 10
```

#### 2. 避免因为过大的 threshold 导致目标对象没有启动清理

```
autovacuum_vacuum_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
```

以上的 threshold 参数有些情况下不适用, 例如:

1. 容量大的表中大部分是冷数据, 少量更新频繁的热数据
2. 容量特别大的表, 希望保持更少的增量

这种情况下需要为这类表和对应的 toast 表单独配置表级 autovacuum 参数。

需要修改的参数 (希望在修改比例超过 1/1000 时就启动清理)



```
ALTER TABLE TABLE_1 SET
(autovacuum_vacuum_scale_factor = 0.001, toast.autovacuum_vacuum_scale_factor = 0.001);
```

### 3. 保证 autovacuum 的资源使用

- a) 在方案或评估测试中加入并发清理对 I/O 资源的需求考虑，降低并发清理时对正常业务的影响。
- b) 配置并发清理进程数，需要根据清理目标制定并发清理进程数，例如：确认系统需要 autovacuum 的表，制定清理目标。例如：系统中更新量大的表有 10 个，容量为 100GB，每小时更新的热数据容量为 20GB，希望维持在 120GB 左右的容量。假设系统在正常运行业务的情况下清理表需要 30 分钟，则这些表都需要在清理后 30 分钟内启动清理，并发清理进程至少应该有 5 个。autovacuum\_max\_workers = 5
- c) 结合并发清理进程数配置记录清理元组的内存，注意每个清理进程都会最多使用配置的内存，上限 1GB。autovacuum\_max\_workers × autovacuum\_work\_mem 不超过物理内存 10% 的情况下尽可能配置更大的 autovacuum\_work\_mem。
- d) 默认不限制 autovacuum 的 I/O 使用，优先保证清理的及时，代价是资源不足时可能和正常业务争抢资源。

```
autovacuum_vacuum_cost_delay = 0
autovacuum_vacuum_cost_limit = 0
```

### 4. 调整物理组织或调整业务实现。

目标对象的修改并发量超过清理速度时，在资源充足的情况下对目标做分区，以支持并发的清理，增加清理速度直到达到和修改速度的平衡。另外可以考虑一些常见的实现优化手段：使用序列而不是对一行数据更新获得递增值；利用分区 +truncate 代替 delete 操作；符合使用场景的情况下使用本地更新。

### 5. 避免长事务的影响

长事务除了影响清理，也会影响系统的可用性。维护过程中应监控并及时处理长事务。

长事务的监控和处理方式参考《KingbaseES 数据库运维手册》中相关内容。

需要确认 vacuum\_defer\_cleanup\_age = 0，尽可能通过 hot\_standby\_feedback 避免读写分离集群备机的查询冲突。

### 6. 开启自动清理日志

```
log_autovacuum_min_duration = 600000
```

根据清理目标调整值，使得超出预期的清理可以有日志记录。

注意清理日志也可以做表级调整，对应 SQL 示例：

```
ALTER TABLE TABLE_1 SET
(log_autovacuum_min_duration = 600000, toast.log_autovacuum_min_duration = 600000);
```

### 22.2.1.1.3 监控自动清理

监控表/索引的膨胀率，通过清理日志确认清理不达到预期的原因，调整配置或者使用 VACUUM 命令处理自动清理没有达到预期的对象。

## 22.2.1.2 VACUUM 命令

### 参数说明

部分表/索引的自动清理可能由于长事务、配置错误等原因没有达到预期，使用 VACUUM 手动清理这些表/索引。

也可能全局配置无法保证部分表/索引的清理目标，例如不希望增加 `autovacuum_max_workers` 数量但需要保证一些容量不大但更新频繁的表的清理频率，可以配合定时任务对这些表/索引做 VACUUM。

VACUUM 命令应该使用 VERBOSE 选项用于诊断结果，除非有必要，不对 VACUUM 做限流：

```
vacuum_cost_delay = 0
vacuum_cost_limit = 0
```

## 22.2.2 重建

清理操作在部分情况下无法实现目标，例如少见的业务波峰造成的额外空间膨胀、递增值的索引无法利用回收的空间等等。重建操作可以重组物理空间消除膨胀。

### 22.2.2.1 sys\_squeeze

`sys_squeeze` 支持对目标对象的重建，和 VACUUM FULL 不同，`sys_squeeze` 不会对目标对象全程加排他锁

#### 22.2.2.1.1 工作方式

1. 创建 replication slot，构造 replication decoding context。记录 snapshot。
2. 创建一个新表包含旧表中所有的行，在该表加 access exclusive lock。
3. 在新表上建立索引。
4. 强制 flush xlog，随后将增量数据写入新表，该过程在原始表上加 access exclusive lock。
5. 修改 `sys_class` 替换目标表，包括索引和 Toast 表，该过程在 `sys_class` 表的相对应行加 row exclusive lock。
6. 删除原始表。
7. 上述 4、5 步使用 access exclusive lock 和 row exclusive lock，该过程内对应表不可访问。

## 22.2.2.1.2 使用方式

参见手册《KingbaseES 插件参考手册》中 sys\_squeeze 。

## 注意:

1. sys\_squeeze 不支持对分区表主表直接重建, 需要对子表执行重建
2. sys\_squeeze 建议不要同 vacuum full 同时使用, 此时可能造成 sys\_squeeze 失败, 带来额外的 IO 开销
3. sys\_squeeze 建议在非业务高峰期执行
4. 建议尽量避免对大表使用

sys\_squeeze 完整执行流程如下:

```
test=# select squeeze.squeeze_table ('public','tb1',null,null,null);
```

注意: Now begin to squeeze the table.

注意: Trying to setup logical decoding.

注意: This step needs acquire lock and may be block if there is long time not ending transaction,if this step is not done in a long time(e.g. 1min) please cancel the session and try again when the transaction is end.

注意: Setup logical decoding done.

注意: Now create transient table.

注意: New table 'tb1' locates at [base/15368/16709]

注意: New toast table 'sys\_toast\_16709' locates at [base/15368/16712]

注意: New toast index 'sys\_toast\_16709\_index' locates at [base/15368/16714]

注意: Now cp the data to new created table.

注意: Now create index on the new created table.

注意: New index 'tb1\_pkey' locates at [base/15368/16715].

注意: Now process the concurrent changes via logical decoding.

注意: The data has been moved to new table, now release the replication slot.

注意: Now swap the filenode.

注意: Delete the old table.

注意: The squeeze process is done.

```
squeeze_table
```

```
-----
```

(1 行记录)

## 22.2.2.1.3 可能的开销

sys\_squeeze 执行过程中带来的开销如下:

1. 重建新表文件, 数据会新写入一遍到新表
2. 日志开销, 相关写操作记录 wal 日志

3. 逻辑解码过程中的计算和 IO 开销
4. 会记录复制槽，可能造成期间日志无法归档
5. 中途中断或执行失败后可能残留临时表文件

#### 22.2.2.1.4 失败处理

`sys_squeeze` 在处理表膨胀过程中可能会遇到存储空间不足、进程被误杀等情况，在这些特殊极端情况下 `sys_squeeze` 会异常崩溃退出，无法完成正常的表膨胀处理工作（不影响原表的使用），会在 `data` 目录残留一些无效的临时文件，可通过 `squeeze_table()` 的输出获知这些临时文件的路径，手动删除这些临时文件即可。

#### 注意:

1. 只有在明确知道 `squeeze_table()` 失败的情况下，才可以手动删除这些文件
2. `squeeze_table()` 成功后不能删除这些文件
3. `squeeze_table()` 卡住无输出时，不能删除这些文件，可手动 `Ctrl+C` 中断 `squeeze_table()` 函数后删除这些文件

### 22.2.2.2 REINDEX

REINDEX 用于重建索引解决索引膨胀。REINDEX CONCURRENTLY 支持在不阻塞 HEAP 表的写入和查询情况下重建索引，在需要在线处理膨胀时可以使用。

#### 22.2.2.2.1 工作方式

REINDEX 会重建目标索引。

使用 CONCURRENTLY 的过程和创建索引的过程有些差异：

1. 开启事务，创建目标索引元信息，`isready` 和 `isvalid` 都是 `false`，对表加 `share update exclusive lock`，提交。
2. 开启新事务，等待修改目标表的事务结束，获取快照，对快照可见的数据创建索引，标记 `isready` 为 `true`，提交。后续的修改都会维护索引。
3. 开启新事务，等待修改目标表的事务结束，获取快照 X，为快照可见但索引中不包括的数据创建索引，提交。
4. 开启新事务，等待持有比 X 的 `xmin` 更老的快照的事务结束，提交。
5. 开启新事务，交换新旧索引，提交。
6. 开启新事务，等待访问目标表的事务结束，标记旧索引 `dead`，提交。
7. 开启新事务，等待访问目标表的事务结束，删除旧索引，提交。放表锁。

#### 22.2.2.2.2 使用方式

参见手册《KingbaseES SQL 语言参考手册》- REINDEX 。

使用时调整 `maintenance_work_mem` 获得更好的性能并降低 I/O 影响。

#### 22.2.2.2.3 可能的开销

参见手册《KingbaseES SQL 语言参考手册》- REINDEX 。注意使用 `CONCURRENTLY` 时会带来额外开销。

#### 22.2.2.2.4 失败处理

命令失败：使用 `CONCURRENTLY` 时失败会残留 `invalid` 的索引，处理方式参见手册《KingbaseES SQL 语言参考手册》- REINDEX 。

过程中数据库 crash：可能性很小，如果发生可能残留存储文件。

### 22.2.2.3 VACUUM FULL

`VACUUM FULL` 和 `VACUUM` 不同，实际是对目标对象的重建。在目标对象膨胀严重且清理无法有效回收空间时可以使用。由于重建过程中需要在目标对象上加排他锁，建议在 `sys_squeeze` 不适用的情况下使用。

#### 22.2.2.3.1 工作方式

`VACUUM FULL` 等同于重建目标对象。

#### 22.2.2.3.2 使用方式

使用方式和相关参数参见手册《KingbaseES SQL 语言参考手册》- `VACUUM` 。

#### 22.2.2.3.3 可能的开销

参见手册《KingbaseES SQL 语言参考手册》- `VACUUM` 中对选项 `FULL` 的描述，主要包括新建对象额外的空间和过程中目标对象上的排他锁。

#### 22.2.2.3.4 失败处理

过程中数据库 crash：可能性很小，如果发生可能残留存储文件。

### 22.2.3 查看清理/重建进度

可以通过以下视图查看容量大的对象清理、重建的进度，根据进度估算完成时间。

```
VACUUM: sys_stat_progress_vacuum  
REINDEX: sys_stat_progress_create_index  
VACUUM FULL: sys_stat_progress_cluster
```

视图具体含义参考《KingbaseES 数据库参考手册》中 动态性能视图一节。

# 版权声明

---

北京人大金仓信息技术股份有限公司（简称：人大金仓）版权所有，并保留对本手册及本声明的一切权利。

未得到人大金仓的书面许可，任何人不得以任何方式或形式对本手册内的任何部分进行复制、摘录、备份、修改、传播、翻译成其他语言、将其全部或部分用于商业用途。

## 免责声明

本手册内容依据现有信息制作，由于产品版本升级或其他原因，其内容有可能变更。人大金仓保留在没有任何通知或者提示的情况下对手册内容进行修改的权利。

本手册仅作为使用指导，人大金仓在编写本手册时已尽力保证其内容准确可靠，但并不确保手册内容完全没有错误或遗漏，本手册中的所有信息也不构成任何明示或暗示的担保。

## 技术支持

- 人大金仓官方网站：<http://www.kingbase.com.cn/>
- 人大金仓文档中心：<http://help.kingbase.com.cn/>
- 全国服务热线：400-601-1188
- 人大金仓技术支持与反馈信箱：[support@kingbase.com.cn](mailto:support@kingbase.com.cn)

# 服务周期承诺

由于市场需求在不断变化，技术创新和发展的进程不断加剧，产品的版本更迭不可避免。人大金仓对于产品版本生命周期的有效管理，有助于您提前规划项目，更好地从产品服务终止上过渡。

表 1: KingbaseES 产品生命周期里程碑

关键里程碑点	定义
产品发布日期	产品正式发布版本，即 GA (general availability) 版本的发布日期。
停止销售日期	正式停止销售的日期，版本停止接受订单日。该日之后，产品将不再销售。
停止功能升级日期	在该日期之后，不再提供新特性和新硬件支持。但依旧提供错误修复、安全修复、功能维护等服务。
停止功能维护日期	在该日期之后，不再维护功能，修复问题。但依旧提供安全修复等服务
停止安全维护日期	在该日期之后，不再发布补丁版本修复中高风险漏洞，仅提供有限的支持。
产品服务终止日期	停止提供产品服务和支持的日期。包括软件维护版本，缺陷修复，以及针对该产品的所有服务支持（包括服务热线和远程/现场支持）。

## 服务周期策略

金仓数据库管理系统 KingbaseES 产品确保以下的服务周期：

- 1) 产品自发布之日起至产品停止功能升级（包含新特性、新硬件支持）之日不少于 5 年。
- 2) 产品停止功能升级之日起至产品停止功能维护（主要包括问题修复）之日不少于 4 年。
- 3) 产品功能维护停止之日起至产品停止安全维护（包括中高风险漏洞修复）之日不少于 2 年。

## 服务终止策略

金仓数据库管理系统 KingbaseES 产品确保在销售后，至少提供 6 年的服务支持。

**注意：** 人大金仓将会综合各方因素来确定产品服务终止日期。并将在实际产品服务终止日期之前至少 90 天，通过公



开方式宣布产品服务终止日期。

---