云数据库 KeeWiDB

开发规范

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或95716。

文档目录

开发规范
与 Redis 的定位差异

最近更新时间：2026-04-24 10:57:51

KeeWiDB 和 Redis 都兼容 Redis 协议，但底层架构不同。Redis 将全部数据存储在内存中，而 KeeWiDB 以磁盘为主存储介质，仅将热数据缓存在内存中。这一架构差异使它们在性能特征、适用场景和成本结构上有所不同。本文档说明 KeeWiDB 的产品定位和性能边界，供开发者在选型阶段参考。

产品定位

KeeWiDB 是一款兼容 Redis 协议的磁盘型数据库。 架构设计上，全量数据持久化于磁盘，并同步在内存中维护热数据缓存以加速高频访问。冷热分层模式有效降低了大容量存储场景下的单 GB 成本，但也带来了性能曲线的阶梯式差异：请求命中内存时维持微秒级响应；一旦回落磁盘读取，执行耗时则上升至毫秒级。对于涉及大量元素遍历的复杂命令，受磁盘 I/O 累加影响，延迟现象将进一步放大。

KeeWiDB 与 Redis 的定位差异

选型时需充分评估磁盘 IO 对请求时延的影响，避免将 KeeWiDB 用于对时延稳定性要求极高的场景。两者的性能差异源于**单个操作耗时**的量级不同，对于一条涉及 N 个元素的命令，总耗时可按以下模型估算：

```
请求耗时 = 网络耗时 + 单个操作耗时 × N
```

在 Redis 中，单个操作在内存中完成，耗时为微秒级；而在 KeeWiDB 中，单个操作可能触发磁盘读取，耗时上升到毫秒级。当 N 较小时，两者差异不大；但当 N 增大到数百甚至数千时，KeeWiDB 的总耗时呈线性放大，高于 Redis。

数据库	网络耗时	单个操作耗时	O(N) 命令总体时延
Redis	毫秒级	微秒级（内存访问）	毫秒级，N 增大后仍可控
KeeWiDB	毫秒级	毫秒级（可能涉及磁盘 IO）	百毫秒级甚至秒级，随 N 线性增长

根据上述差异，可结合业务特征选择合适的产品：

推荐产品	适用场景特征	典型业务示例
KeeWiDB	数据量大（TB 级）、成本敏感、以简单读写为主、可接受毫秒级时延波动、数据需长期存储	**保单归档查询**：历史保单2TB+，日均查询量低，按保单号单条读取，存储成本优势明显

		操作日志存储：日志5TB+，保留180天，以写入和范围查询为主，成本敏感
Redis	数据量适中、对时延稳定性要求严格（如 P99 < 1ms）、存在大量复杂命令、数据频繁过期更新	**实时风控决策**：每笔交易需在5ms内完成风险画像查询和规则匹配，不能接受磁盘 IO 时延波动
		用户会话缓存：数据量50GB，会话30分钟过期，需高频读写，Redis 内存淘汰效率远高于磁盘扫描

二、架构差异对比（存储版 vs 极速版 vs Redis）

KeeWiDB 提供存储版和极速版两种架构，与 Redis 在内部实现上存在差异。以下对比供选型和性能调优时参考。

2.1 功能与效率对比

用法	KeeWiDB 存储版	KeeWiDB 极速版（已售罄）	Redis
SUBSCRIBE / PSUBSCRIBE	效率中。 Proxy 对每个分片仅建立1个连接，但仍受整体架构影响。	效率低。 分片内部再拆分为多个子分片，Proxy 需对每个分片建立 N 个连接，连接开销大、消息分发延迟高。	效率高。 Proxy 对每个分片仅建立1个连接，消息在内存中分发。
EVAL（Lua）	效率中。 每个工作线程各自维护一个 Lua 虚拟机实例。	效率低。 每个客户端连接独立维护一个 Lua 虚拟机，连接数越多，创建和销毁 Lua 虚拟机代价越大。	效率高。 全局仅一个 Lua 虚拟机，无需频繁创建和销毁 Lua 虚拟机。
Pipeline	效率低。 命令命中冷数据时需回落磁盘 IO 逐条处理，耗时累加导致整批请求延迟升高，易引发 Proxy 与后端节点的请求积压。	同存储版。	效率高。 全部数据在内存中，管道内命令连续执行，单次往返即完成批量操作。
过期淘汰	效率低。 清理过期数据需遍历磁盘，IO 开销高于内存扫描，大量 Key 集中过期时可能引发 IO 毛刺。	同存储版。	效率高。 过期扫描与数据清理全部在内存中完成，耗时稳定可控。

2.2 功能支持差异

以下为三个版本在关键特性上的支持情况。选型时需确认业务是否依赖多 DB 或事务功能：

特性	KeeWiDB 存储版	KeeWiDB 极速版	Redis
数据存储介质	磁盘 + 内存（热数据缓存）	磁盘 + 内存（热数据缓存）	纯内存（可选 RDB/AOF 持久化）
多 DB 支持	仅支持 DB 0，不支持 `SELECT` 切换	仅支持 DB 0，不支持 `SELECT` 切换	支持多 DB（默认16个）
事务支持	不支持（可用 Lua 脚本实现单节点原子操作）	不支持（可用 Lua 脚本实现单节点原子操作）	支持 `MULTI` / `EXEC` 事务

实例命名规范

最近更新时间：2026-04-24 10:57:51

当实例数量增长到数十甚至上百个、Key 总量达到千万级时，缺乏统一命名体系会导致问题排查效率降低、权限管理出错和资源归属不清。本文档从**实例命名**和 **Key 命名**两个维度定义标准化的命名规则，使团队成员可以通过名称直接识别资源所属的环境、业务和地域。

一、实例命名规则

规范的实例名称能够在控制台列表、监控面板和告警通知中直接传达实例的归属信息，帮助运维人员快速定位目标实例，降低误操作风险。建议按照**环境、组织、业务、位置**四个维度，以连字符 `-` 拼接的方式命名实例：

```
{环境}-{分公司}-{项目组编号}-{业务名称}-{地域}-{专区}-{序号}
```

其中各字段的含义如下：

字段	说明	示例
环境	区分生产与开发，避免误操作影响线上服务	`prd`（生产）、`dev`（开发）
分公司	所属分公司名称缩写，便于费用归属和权限划分	`cx`（财险）、`rx`（人险）
项目组编号	项目组唯一标识，方便按团队粒度统计资源用量	`p001`、`p002`
业务名称	该实例所服务的具体业务，使用有语义的英文缩写	`ordercache`（订单缓存）、`policysync`（保单同步）
地域	实例部署地域缩写，与腾讯云地域编码保持一致	`gz`（广州）、`sh`（上海）
专区	实例所属专区缩写	`gzcx01`、`shcx01`
序号	同一业务下的实例序号，用于区分多实例部署	`1`、`2`

- **正确示例：**

示例	说明

`prd-cx-p001-ordercache-gz-gzcx01-1`	生产环境，财险 P001 项目组的订单缓存实例，部署在广州财险01专区
`dev-cx-p001-ordercache-gz-gzcx01-1`	与上一条对应的开发环境实例
`prd-cx-p002-policysync-sh-shcx01-1`	生产环境，财险 P002 项目组的保单同步实例，部署在上海财险01专区

- **反面示例：**

示例	问题
`keewidb-01`	缺少环境、业务等关键信息，无法判断归属
`生产-财险-订单`	使用中文命名，控制台和脚本中容易引发编码问题
`prd_cx_p001`	使用下划线分隔，与规范的连字符风格不一致

二、Key 命名规则

建议 Key 命名采用业务名为前缀、冒号分层的结构。这种方式可以防止不同业务之间的 Key 冲突，同时使 Key 具备自描述能力——通过名称即可判断其所属的业务系统和数据来源。建议按以下格式命名：

```
业务名:数据库名称:数据库表名称:数据ID
```

其中各字段的含义如下：

字段	说明	示例
业务名	业务系统缩写，作为全局唯一的命名空间前缀	`cx`（财险）、`rx`（人险）
数据库名称	数据来源的数据库名，便于溯源	`orderdb`（订单库）、`userdb`（用户库）
数据库表名称	数据对应的表名，定位到具体的数据实体	`order`（订单表）、`user`（用户表）
数据 ID	数据表中的主键或唯一标识，精确定位到单条记录	`202401150001`、`100238`

- **正确示例：**

示例	说明
`cx:orderdb:order:202401150001`	财险订单库中的订单记录，订单号为202401150001
`cx:userdb:user:100238`	财险用户库中的用户记录，用户 ID 为100238
`rx:policydb:policy:PL20240001`	人险保单库中的保单记录，保单号为 PL20240001

- **反面示例：**

示例	问题
`000110011`	无前缀、无分层，无法判断业务归属和数据来源
`cx_cxdb_user_000110011`	使用下划线分隔层级，与冒号分层规范不一致
`cx:cxdb:cxdb_user_info:000110011`	表名中重复包含库名 "cxdb"，冗余且增加 Key 长度

Key 与 Value 设计规范

最近更新时间：2026-04-24 10:57:51

Key 的命名方式影响数据的可维护性，Value 的大小和结构影响查询效率与内存利用率。本文档从 Key 设计原则、大 Key 规避策略和数据类型选择三个维度，定义具体的设计准则，用于在项目初期建立合理的数据模型，减少慢查询和内存膨胀问题。

一、Key 设计原则

Key 的命名直接影响数据的可维护性、运维效率和内存利用率。以下六条原则从简洁性、字符集、语义结构、类型标识、长度控制和特殊字符六个维度，定义 Key 命名的标准化规范。

1. 简洁性

在保证语义清晰的前提下，尽量缩短 Key 长度。当 Key 总量达到百万级甚至千万级时，Key 本身占用的内存空间不可忽略。每个 Key 缩短10个字节，百万 Key 即可节省约10MB内存。

分类	示例	说明
正确	`cx:orderdb:order:202401150001:hash`	简洁明确，总长度40字节。
错误	`cx:orderdb:orderdb_order_detail_info:202401150001`	表名冗余重复库名"orderdb"，增加18字节无效长度。

2. 命名字符

以英文字母开头，只能包含大小写字母、数字、竖线（`|`）、下划线（`_`）、英文点号（`.`）和英文半角冒号（`:`）。统一字符集有助于避免编码歧义，同时确保 Key 在日志、监控和运维工具中能被正确解析。

分类	示例	说明
正确	`cx:userdb:session:U100238:string`	仅使用字母、数字和冒号，符合字符集规范。
错误	`user 100238`	包含空格，`SCAN` 命令无法正确匹配。
错误	`cx:orderdb:order:{202401150001}`	使用花括号，与 Hashtag 语法冲突。

3. 语义分割

使用英文半角冒号（ `:` ）分隔不同业务含义层级（如 `业务:模块:标识` ），同一层级内的单词之间使用英文半角点号（ `.` ）连接。清晰的分层结构可让运维人员在排查问题时快速定位 Key 所属的业务模块。

分类	示例	说明
正确	`cx:userdb:profile:U100238:hash`	用冒号分层：业务 → 库 → 表 → ID → 类型。
正确	`cx:userdb:basic.info:U100238:string`	同层词汇"basic"和"info"用点号连接。
错误	`cx_userdb_profile_U100238`	使用下划线分隔层级，与冒号分层规范不一致。

4. 类型后缀

在 Key 末尾附加 Value 类型标识。当团队成员看到 Key 名称时，无需执行 `TYPE` 命令即可判断对应的数据结构，降低沟通成本。

分类	示例	说明
正确	`cx:orderdb:order:202401150001:hash`	末尾 `:hash` 表明 Value 为 Hash 类型。
正确	`cx:userdb:session:U100238:string`	末尾 `:string` 表明 Value 为 String 类型。
错误	`cx:orderdb:order:202401150001`	缺少类型后缀，无法判断 Value 是 String 还是 Hash。

5. 控制 Key 大小

Key 名称建议控制在128字节以内。过长的 Key 占用额外内存，同时增加比较和哈希计算的开销，在高并发场景下对 CPU 产生可测量的影响。

分类	示例	说明
正确	`cx:userdb:session:U100238:string`	总长度36字节，远低于128字节上限。
错误	`cx:orderdb:orderdb_order_detail_info_full_record:202401150001:hash`	总长度65字节，包含冗余描述，应精简。

6. 禁止特殊字符

禁止包含 `\`、`*`、`?`、`{}`、`[]`、`()`、空格、单双引号和转义字符。这些字符与 glob 通配符和 Shell 转义规则冲突，会导致 `SCAN`、`KEYS` 等模式匹配命令无法正确检索，同时在日志分析和脚本处理中引发解析异常。

分类	示例	说明
正确	`cx:orderdb:order:202401150001:hash`	不含任何特殊字符。
错误	`cx:orderdb:order:{202401150001}`	花括号与 Hashtag 语法冲突。
错误	`cx:orderdb:order:2024*:hash`	星号为 glob 通配符，`SCAN` 无法精确匹配。
错误	`user "test"`	包含空格和双引号，Shell 和日志解析会出错。

二、Value 设计原则

Value 的体积和集合元素数量需严格控制，避免产生大 Key。当某个 Key 的 Value 体积过大或集合元素过多时，读写该 Key 会占用较长的 CPU 时间片，阻塞其他请求的处理，表现为慢查询增多；同时，大 Value 在网络传输时占用较多网卡带宽，少量大 Key 的并发读取即可造成网络拥塞，影响同节点上其他业务的响应时间。本节定义大 Key 的判定阈值、日常开发建议阈值及标准化拆分策略。

1. 大 Key 判定阈值

当某个 Key 达到下表条件时，应排查并进行拆分或重构：

数据类型	大 Key 判定条件	风险说明
String	Value 值超过1MB。	单次读写占用过长 CPU 时间片，阻塞同线程其他请求。
Set	成员数量超过10000个。	全量遍历耗时线性增长，`SMEMBERS` 等命令可能触发慢查询。
List	成员数量超过10000个。	同上，`LRANGE 0 -1` 全量读取将阻塞服务端线程。
Hash	成员数量超过1000个，且所有字段的总 Value 大小超过 1000MB。	`HGETALL` 返回的数据量过大，网络传输耗时显著增加。

Sorted Set	成员数量超过10000个。	`ZRANGEBYSCORE` 大范围查询耗时线性增长。

2. 日常开发建议阈值

为了在问题发生之前将风险控制在安全范围内，建议在日常开发中遵循更严格的阈值标准：

数据类型	建议阈值	规范建议
String	≤ 10KB	超过10KB时按业务维度拆分为多个子 Key。
Hash / List / Set / ZSet	元素个数 ≤ 5000	超过5000时按字段范围、时间区间或业务分类拆分。

3. 大 Key 拆分策略

当 Value 超过上述阈值时，应根据数据类型选择对应的拆分方式。以下按数据类型分别说明拆分策略和代码实现规范。

String 类型拆分

当单个 String 的 Value 超过10KB时，按业务维度将数据拆分为多个子 Key。

- 反面示例（单 Key 超限）：将整个订单的 JSON 序列化后存入一个 String，体积达到50KB。

```
# 潜在风险：50KB 的 Value 在高并发读取时占用大量网卡带宽
SET cx:orderdb:order:202401150001:string
'{"status":"paid","amount":12580,"items":[...大量商品明
细...],"logistics":{...物流信息...}}'
```

- 正确示例（按业务维度拆分）：将订单拆分为基础信息、商品明细、物流信息三个子 Key，每个 Key 的 Value 控制在10KB以内。

```
# 拆分后每个子 Key 的 Value 体积可控，支持按需读取
SET cx:orderdb:order:202401150001:base:string
'{"status":"paid","amount":12580}'
SET cx:orderdb:order:202401150001:items:string '[...商品明细...]'
SET cx:orderdb:order:202401150001:logistics:string '{...物流信息...}'
```

Hash 类型拆分

当单个 Hash 的字段数超过5000个时，按字段范围或业务含义拆分为多个 Hash Key。

- 反面示例（单 Hash 超限）：将10000个用户属性全部存入一个 Hash。

```
# 潜在风险：HGETALL 返回 10000 个字段，网络传输和客户端解析耗时显著
HSET cx:userdb:user.attrs:hash field_0001 value_0001 field_0002
value_0002 ... field_10000 value_10000
```

- 正确示例（按字段范围拆分）：每个子 Hash 控制在5000个字段以内。

```
# 按字段范围拆分，HGETALL 单个子 Hash 的返回量可控
HSET cx:userdb:user.attrs:part1:hash field_0001 value_0001 ...
field_5000 value_5000
HSET cx:userdb:user.attrs:part2:hash field_5001 value_5001 ...
field_10000 value_10000
```

List / Set / Sorted Set 类型拆分

当集合元素超过5000个时，按时间范围、ID 区间或业务分类拆分为多个子 Key。

- 反面示例（单集合超限）：将全年的用户操作日志存入一个 List，元素数达到数万。

```
# 潜在风险：LRANGE 0 -1 全量读取数万条记录，阻塞服务端线程数秒
LPUSH cx:userdb:user.oplog:U100238:list "2024-01-01 login" "2024-01-01
query" ... "2024-12-31 logout"
```

- 正确示例（按月份拆分）：每个 List 仅存储当月日志，单个 List 元素数可控。

```
# 按月份拆分后，单个 List 的元素数控制在合理范围内
LPUSH cx:userdb:user.oplog:U100238:202401:list "2024-01-01 login"
"2024-01-01 query" ...
LPUSH cx:userdb:user.oplog:U100238:202402:list "2024-02-01 login" ...
```

三、数据类型选择指南

KeeWiDB 兼容 Redis 协议，提供五种基础数据类型。不同类型在内部编码和操作复杂度上存在差异，选择合适的类型可以在相同硬件条件下提升读写效率、降低内存占用。以下按数据类型分别说明适用场景、典型用法及代码示例。

1. String（字符串）

存储单一值，如状态标记、序列化对象、二进制数据等。适用于分布式锁、计数器、会话缓存、配置信息等场景。

```
# 会话缓存：存储用户登录态，设置 30 分钟过期
```

```
SET cx:userdb:session:U100238:string '{"token":"abc123","role":"admin"}'
EX 1800

# 计数器：文章阅读量自增
INCR cx:articledb:article:A20240001:views:string
```

2. Hash（哈希表）

存储具有多个字段的结构化数据，支持对单个字段的独立读写，无需整体序列化。适用于用户画像（姓名、年龄、地域）、商品属性、设备信息等场景。

```
# 用户画像：聚合存储多个属性
HMSET cx:userdb:profile:U100238:hash name "张三" age 28 region "广州"

# 按需读取单个字段
HGET cx:userdb:profile:U100238:hash name

# 批量读取部分字段
HMGET cx:userdb:profile:U100238:hash name age
```

3. List（列表）

维护有序的元素序列，支持从两端执行 O(1) 复杂度的插入和弹出操作。适用于消息时间线、最近浏览记录、任务队列等场景。

```
# 最近浏览记录：从左侧插入最新记录，保留最近 50 条
LPUSH cx:userdb:recent.views:U100238:list "商品A_2024-01-15"
LTRIM cx:userdb:recent.views:U100238:list 0 49
```

4. Set（集合）

存储无序且不重复的元素集合，支持交集、并集、差集运算。适用于标签系统、共同好友计算、去重集合等场景。

```
# 标签系统：为用户添加兴趣标签
SADD cx:userdb:tags:U100238:set "保险" "理财" "汽车"

# 共同好友：计算两个用户的共同标签
SINTER cx:userdb:tags:U100238:set cx:userdb:tags:U200456:set
```

5. Sorted Set（有序集合）

每个元素关联一个分值（score），按分值排序存储，支持范围查询和排名查询。适用于实时排行榜、延迟任务调度、带权重的投票系统等场景。

```
# 实时排行榜：以销售额作为分值
ZADD cx:salesdb:rank:202401:zset 125800 "华南区" 98500 "华东区" 76200 "华北区"

# 查询 Top 3
ZREVRANGE cx:salesdb:rank:202401:zset 0 2 WITHSCORES
```

> **说明：**
> KeeWiDB 极速版在底层使用了 ziplist（压缩列表）等紧凑编码方式来优化小型数据结构的内存占用。ziplist 将元素存储在一段连续的内存块中，省去了指针开销，但由于缺少索引，查找、插入和删除操作均需线性扫描。因此，当元素数量较少（通常不超过128个）且单个元素体积较小时，ziplist 可以减少内存占用；当元素规模增长后，应让系统自动转换为标准数据结构以维持查询效率。

6. 多属性存储：用 Hash 替代多个 String Key

当一个实体拥有多个属性时（如用户的姓名、年龄、爱好），建议使用一个 Hash Key 来聚合存储，而非为每个属性创建独立的 String Key。这样做有三个好处：一是减少 Key 的总数量，降低数据库的键空间占用；二是可以通过 `HGET`、`HMGET` 按需读取部分字段，避免不必要的数据传输；三是在逻辑上保持数据的内聚性，便于后续维护。

- 反面示例（多 String Key 分散存储）：为订单的每个属性创建单独的 String Key，5个属性即产生5个Key，百万订单将占用500万个键空间。

```
# 潜在风险：每个属性一个 Key，键空间膨胀 5 倍，内存开销显著增加
SET cx:orderdb:order:202401150001:status paid
SET cx:orderdb:order:202401150001:amount 12580
SET cx:orderdb:order:202401150001:product car_insurance
SET cx:orderdb:order:202401150001:customer C100238
SET cx:orderdb:order:202401150001:create_time 2024-01-15T10:30:00
```

- 正确示例（Hash 聚合存储）：使用一个 Hash Key 聚合全部属性，百万订单仅占100万个键空间，且支持按需读取单个字段。

```
# 一个 Hash Key 聚合全部属性，减少键空间占用
```

```
HMSET cx:orderdb:order:202401150001:hash status paid amount 12580
product car_insurance customer C100238 create_time 2024-01-15T10:30:00

# 按需读取单个字段，无需加载整个订单
HGET cx:orderdb:order:202401150001:hash status

# 批量读取部分字段
HMGET cx:orderdb:order:202401150001:hash status amount customer
```

命令使用规范

最近更新时间：2026-04-24 10:57:51

KeeWiDB 兼容 Redis 协议，但在架构上属于磁盘型数据库，数据访问涉及磁盘 IO，其延迟特性与内存型 Redis 存在显著差异。由于磁盘操作耗时远高于内存，命令复杂度对响应时间的影响具有放大效应。为确保数据库响应时间及系统整体稳定性，本文档遵循"先规避风险、再规范用法"的原则，定义以下使用准则：

1. **禁用命令**：明确禁止在生产环境执行的高危命令。
2. **O(N) 命令管控**：识别高复杂度命令的潜在风险，提供标准化替代方案。
3. **批量操作规范**：定义单批次处理的数量阈值，防止磁盘 IO 累加导致请求超时。
4. **功能限制**：界定存储架构层面的约束，规避非兼容特性调用。
5. **Lua 脚本使用规范**：约束脚本执行逻辑，确保其在分布式集群环境下的正确性。
6. **Hashtag 使用规范**：规范 Key 路由逻辑，防止数据分布倾斜。

一、禁用命令

受限于磁盘 IO 独占风险或数据安全考虑，以下命令不建议在生产环境的业务逻辑中直接调用。为确保系统吞吐量及响应延迟的稳定，本节提供了对应的标准化替代路径，建议在开发设计阶段优先采用。

1. 默认禁用命令清单

命令	禁用原因	替代方案
`KEYS`	对整个键空间执行全量扫描，复杂度 O(N)。在百万级 Key 的实例上，单次执行可能阻塞数据库数秒，影响所有并发请求	使用 `SCAN` 命令分批遍历
`FLUSHALL` / `FLUSHDB`	清空全部或当前库的数据，产生大量磁盘 IO 和 CPU 开销，且操作不可逆	通过控制台的**清空实例**功能执行，该操作具备二次确认和审计日志
`CLIENT LIST`	当连接数较高时（如数千并发），遍历并序列化所有连接信息会增加 CPU 和内存开销	通过提交工单获取连接信息

2. 自定义禁用

如需根据业务需求禁用其他命令，可通过调整参数 `disable-command-list` 进行配置。

二、O(N) 命令管控

KeeWiDB 的磁盘存储特性决定了 O(N) 命令的每个元素访问均可能涉及物理磁盘 IO。当 N 规模较大时，累计耗时将从毫秒级线性上升至秒级，不仅会导致当前请求超时，还会长时间占用后端执行线程，引发同节点其他请求的排队堆积。本节界定高危命令的识别标准，并提供标准化替代路径，用于指导开发阶段的方案选型。

1. 高危命令识别

以下命令在处理大规模数据集合时具有较高的执行风险，需严格评估其元素规模：

命令	风险说明	规范建议
`HGETALL`	一次性返回 Hash 全部字段。字段数达到数千级时，单次调用涉及数 MB 数据的磁盘读取与网络传输。	字段数 > 500 时禁用，改用 HGET/HMGET/HSCAN。
`LRANGE`	返回 List 指定范围元素。若执行 `0 -1` 等全量遍历操作，将引发持续的磁盘扫描。	禁止对未知长度的 List 执行全量查询，改为分段读取。
`SMEMBERS`	返回 Set 全部成员，将引发持续的磁盘扫描。	建议替换为 `SSCAN`。
`ZRANGE`	返回有序集合成员。耗时随范围扩大呈线性增长，且涉及较多磁盘读。	控制 Range 区间，或改用 `ZSCAN`。
`SINTER/SUNION/ZINTERSTORE/ZINTERSTORE`	多集合交集或并集运算。复杂度取决于参与集合的成员总数，在集合规模不对称时性能波动剧烈。	在业务层执行逻辑交集计算，减轻服务端压力。

2. 标准替代方案：游标式扫描（SCAN 类命令）

为规避大规模数据访问带来的阻塞风险，应采用 `HSCAN`、`SSCAN`、`ZSCAN` 等游标式命令。通过分批迭代，将单次请求的 IO 耗时和传输数据量锁定在可控范围内。

- 反面示例（全量读取）：当集合包含万级元素时，下述操作将导致数秒的独占阻塞。

```
# 潜在风险：字段数达 5000+ 时，触发秒级延迟
HGETALL cx:orderdb:order_index:hash

# 潜在风险：成员数达 10000+ 时，导致磁盘 IO 饱和
SMEMBERS cx:userdb:active_users:set
```

- 正确示例（分批迭代）：通过设置合理的 COUNT 参数，平衡执行效率与系统稳定性。

```
# 分段获取，建议单次 COUNT 值在 100-500 之间
HSCAN cx:orderdb:order_index:hash 0 COUNT 200

# 配合模式匹配进行分批扫描（不指定COUNT值）
SSCAN cx:userd b:active users:set 0 MATCH U10*
```

3. COUNT 参数使用建议

使用场景	COUNT 参数建议	原因
未指定 `MATCH` 参数	指定合适的 COUNT 值（建议 ≤ 1000）	控制每轮扫描返回的元素上限，避免单次返回数据量过大
已指定 `MATCH` 参数	使用数据库默认值，不额外指定 COUNT	`MATCH` 过滤在扫描之后执行。若 COUNT 过大，可能出现"扫描了大量数据但匹配结果极少"的低效情况

三、批量操作规范

KeeWiDB 的数据主要存储在磁盘上，磁盘寻址耗时与网络往返耗时处于同一量级（均为毫秒级）。将多个命令打包发送所节省的网络往返时间，会被磁盘 IO 的线性累加所抵消——当单批次包含的 Key 数量超过阈值时，整批请求的总耗时将突破 Proxy 超时上限，引发请求积压与客户端报错。本节定义单批次操作的数量阈值与拆分策略，用于指导批量接口的设计与调用。

1. 数量限制

操作方式	数量限制	风险说明	规范建议
`MGET` / `MSET`	单次 ≤ 20 个元素	每个 Key 的磁盘读取耗时线性累加，超限后整批耗时突破超时阈值。	超过20个 Key 时，拆分为多批顺序发送。
Pipeline	单次 ≤ 20 条命令	打包命令在服务端串行执行，命令数过多将导致后端线程长时间占用。	超过20条命令时，拆分为多个 Pipeline 分批执行。

2. 代码实现规范

- 反面示例（单批超限）：单次批量100个 Key，磁盘 IO 累加导致整批超时。

```
# 潜在风险：100 个 Key 的磁盘随机读累加，耗时可达数秒
MGET key1 key2 key3 ... key100
```

- 正确示例（分批发送）：拆分为5批，每批20个，逐批发送。

```
# 第 1 批：20 个 Key，单批耗时可控
MGET key1 key2 key3 ... key20
```

```
# 第 2 批：20 个 Key
MGET key21 key22 key23 ... key40
# ...依次类推，共 5 批完成
```

四、功能限制

以下限制源于 KeeWiDB 的磁盘存储引擎与集群路由架构，属于引擎层面的固有约束而非配置可调项。若在开发阶段未识别这些限制，将在运行时触发命令报错或数据不一致。本节逐项界定限制范围、根因说明及标准化替代路径，用于指导选型与方案设计阶段的兼容性评估。

1. 仅支持 DB 0

KeeWiDB 不支持多 DB 切换，`SELECT` 命令将返回错误。

隔离需求	限制说明	替代方案
业务级隔离	多 DB 切换在磁盘引擎下涉及文件句柄与索引切换，架构不予支持。	通过 Key 前缀命名区分不同业务（如 `cx:` 和 `rx:`）。
环境级隔离	同上。	部署独立实例（如生产实例和开发实例）。

2. 不支持事务

不支持事务相关命令：`MULTI`、`EXEC`、`WATCH`、`UNWATCH`、`DISCARD`。

原子性需求	限制说明	替代方案
单节点范围内的原子操作	集群架构下事务无法跨节点保证一致性，引擎层面不予支持。	使用 Lua 脚本实现（详见第五章），脚本内所有 Key 须路由到同一节点。

3. 禁止作为消息队列

KeeWiDB 的存储模型面向持久化键值访问，不具备消息队列所需的投递保障与消费确认机制。

机制	风险说明	规范建议
Pub/Sub	消息不做持久化，客户端断连后消息丢失，无法满足可靠投递要求。	谨慎使用。
List 模拟队列	缺乏 ACK 确认机制，消费失败后消息不可回溯，且磁盘 IO 下吞吐量远低于专业消息中间件。	不建议在生产环境使用。

如有消息队列需求，请使用专业消息中间件（如 CKafka、TDMQ），获取可靠投递、消费确认和消息回溯能力。

五、Lua 脚本使用规范

Lua 脚本可在 KeeWiDB 服务端原子执行多条命令，适用于需要原子性保障的业务场景（如库存扣减、状态流转）。但在集群版架构下，请求由 Proxy 根据 Key 路由到不同后端节点，脚本的 Key 引用方式和路由逻辑需满足特定约束——违反这些约束将导致 Proxy 路由失败、跨节点执行异常或脚本加载性能劣化。本节定义三条核心规则，确保脚本在分布式集群环境下的正确性与执行效率。

规则1：Key 必须通过 KEYS 数组传递

在 `redis.call` / `pcall` 中引用的所有 Key，必须通过 `KEYS` 数组传递，禁止硬编码在脚本中。Proxy 依赖 `KEYS` 数组判断路由目标节点；硬编码 Key 将导致 Proxy 无法解析路由信息，请求直接失败。

- 反面示例（Key 硬编码）：Proxy 无法从脚本文本中提取路由信息，请求将返回路由错误。

```
# 潜在风险：Key 硬编码在脚本中，Proxy 无法识别路由目标
EVAL "redis.call('SET', 'cx:orderdb:order:202401150001:hash', 'paid')"
0
```

- 正确示例（KEYS 数组传递）：Proxy 从 KEYS 数组提取路由信息，正确分发到目标节点。

```
# Key 通过 KEYS[1] 传递，Proxy 可正确计算哈希槽并路由
EVAL "redis.call('SET', KEYS[1], ARGV[1])" 1
cx:orderdb:order:202401150001:hash paid
```

规则2：单脚本所有 Key 必须位于同一节点

单个 Lua 脚本操作的所有 Key 必须路由到同一个节点。若 Key 分布在不同节点，脚本将执行失败。可通过 Hashtag（如 `{order:202401150001}`）确保相关 Key 落在同一哈希槽。如下示例使用 Hashtag 将同一订单的多个属性 Key 绑定到同一哈希槽。

```
# 两个 Key 共享 Hashtag {order:202401150001}，确保路由到同一节点
EVAL "redis.call('SET', KEYS[1], ARGV[1]); redis.call('SET', KEYS[2],
ARGV[2])" 2 \
  {order:202401150001}:status {order:202401150001}:amount paid 12580
```

规则3：使用 EVALSHA 替代 EVAL

首次通过 `SCRIPT LOAD` 加载脚本后，后续调用应使用 `EVALSHA` 传入脚本的 SHA1 摘要。每次使用 `EVAL` 都会传输完整脚本文本并触发重新编译，在高频调用场景下将产生显著的网络开销与 CPU 消耗。建议部署阶段使用

`SCRIPT LOAD` 将脚本文本上传至服务端并获取 SHA1 摘要；运行阶段使用 `EVALSHA` 仅传输20字节的摘要值即可调用已缓存的脚本，避免每次请求都传输完整脚本文本。

```
# 部署阶段（执行一次）：将脚本文本上传至服务端，服务端返回 SHA1 摘要
SCRIPT LOAD "redis.call('SET', KEYS[1], ARGV[1])"
# 返回："a1b2c3d4e5f6..."

# 运行阶段（反复调用）：仅传输 SHA1 摘要，服务端直接执行已缓存的脚本
EVALSHA a1b2c3d4e5f6... 1 cx:orderdb:order:202401150001:hash paid
```

六、Hashtag 使用规范

Hashtag 是集群版 KeeWiDB 提供的路由控制机制，通过在 Key 中使用花括号（如 `{user}:name`）指定哈希计算的子串，将多个 Key 强制分配到同一哈希槽。该机制在 Lua 脚本和多键命令中用于保证多 Key 路由到同一节点，但**滥用将引发数据倾斜和请求倾斜，且倾斜一旦形成，无法通过集群扩容缓解**。本节界定 Hashtag 的三大风险、使用原则及代码实现规范，用于指导架构设计阶段的路由策略评估。

1. 三大风险

Hashtag 将多个 Key 绑定到同一哈希槽，可能引发以下风险：

风险类型	风险说明	规范建议
数据倾斜	大量 Key 共用同一 Hashtag 时，数据集中在同一节点，该节点内存使用率偏高，可能提前触发告警甚至写满。	禁止使用宽泛 Hashtag（如 `{global}`、`{order}`），按实体粒度拆分。
请求倾斜	与数据倾斜伴生——所有读写请求集中在同一节点，该节点延迟升高、CPU 使用率上升，拖累集群整体响应时间。	确保 Hashtag 值在整体上均匀分布。
不可扩容	哈希槽分配由 Hashtag 值固定决定，新增节点无法分散倾斜数据。区别于普通热点 Key 问题——扩容无法缓解。	设计阶段评估 Hashtag 基数，确保与节点数匹配。

2. 使用原则

原则	说明	规范建议
仅在必要时使用	仅在 Lua 脚本操作多 Key 或使用 `MGET`、`SINTER` 等多键命令时才引入 Hashtag。	不涉及跨 Key 操作的场景，保持自然哈希分布，禁止引入 Hashtag。

使用细粒度 Hashtag	按具体实体拆分（如 `{order:202401150001}`），使不同实体的 Key 分散到不同节点。	禁止使用宽泛 Hashtag（如 `{global}`、`{order}`）。
保证均匀分布	确保 Hashtag 值在整体上均匀分布。	若原始 ID 分布不均匀，可在 ID 后添加后缀（如 `{user_id:1}`）进行人工分片。

3. 代码实现规范

- 反面示例（宽泛 Hashtag）：全部订单共用 `{order}`，百万订单 Key 集中在同一哈希槽，节点内存写满且扩容无法缓解。

```
# 潜在风险：所有订单 Key 共用 {order}，全部落在同一哈希槽
{order}:202401150001:status
{order}:202401150001:amount
{order}:202401150002:status
{order}:202401150002:amount
# 百万订单集中在一个节点，扩容也无法分散
```

- 正确示例（细粒度 Hashtag）：按订单号使用独立 Hashtag，仅同一订单的属性 Key 落在同一节点，不同订单自然分散。

```
# 订单 1 的属性 Key 共享 {order:202401150001}，落在同一节点
{order:202401150001}:status
{order:202401150001}:amount
# 订单 2 的属性 Key 共享 {order:202401150002}，自然分散到其他节点
{order:202401150002}:status
{order:202401150002}:amount
# 每个订单独立 Hashtag，百万订单均匀分布到不同节点
```

客户端设计规范

最近更新时间：2026-04-24 10:57:51

客户端负责应用与 KeeWiDB 之间的通信。连接池配置不当可能导致连接耗尽或资源浪费，缺少熔断保护会使单节点故障扩散为服务整体不可用，弱密码则增加被暴力破解的风险。本文档从连接池配置、熔断机制和密码安全三个方面定义设计准则，用于构建可用性和安全性符合生产要求的客户端程序。

一、连接池配置

建议使用带有连接池的客户端访问 KeeWiDB。

1. 连接池的作用

每次建立 TCP 连接都需要经历三次握手和身份认证，在高并发场景下频繁创建和销毁连接会增加延迟开销和系统资源消耗。连接池通过预先创建并复用一组持久连接，将"每次请求建连"变为"从池中借用连接"，从而降低连接延迟、控制并发连接总数，并避免连接数超出数据库承载上限。

2. 核心参数

连接池通过以下三个参数控制连接的创建、保持与回收：

参数	说明
最大连接数	连接池中允许同时存在的连接上限。达到上限后，新请求将排队等待空闲连接释放，防止突发流量下连接数暴涨、耗尽服务端连接配额
最大空闲连接数	允许保持空闲状态的连接上限。超出部分将被主动关闭，避免低峰期占用不必要的系统资源（如文件描述符和内存）
最小空闲连接数	必须始终保持的空闲连接数。低于此阈值时，连接池在后台预创建新连接进行补充，确保流量突增时有可用连接，减少排队等待

3. 配置建议

将上述三项参数设为相同数值，可使连接池维持恒定规模。保持固定连接数旨在确保行为可预测性：既能消除频繁建立与销毁连接的性能损耗，又能规避空闲回收后因突发流量触发的连接延迟。具体数值应结合业务并发量与 KeeWiDB 实例的最大连接数限制综合评估。

- **正确示例（Go 语言 Redigo）：**

```
pool := &redis.Pool{
    MaxIdle:     50,     // 最大空闲连接数
    MaxActive:   50,     // 最大连接数，与 MaxIdle 一致
    IdleTimeout: 240 * time.Second,
```

```
    Dial: func() (redis.Conn, error) {
        return redis.Dial("tcp", "keewidb-instance:6379",
            redis.DialPassword("your_password"))
    },
}
```

- **正确示例（Java Jedis）：**

```
JedisPoolConfig config = new JedisPoolConfig();
config.setMaxTotal(50);       // 最大连接数
config.setMaxIdle(50);        // 最大空闲连接数，与 MaxTotal 一致
config.setMinIdle(50);        // 最小空闲连接数，与 MaxTotal 一致
config.setTestOnBorrow(true);

JedisPool pool = new JedisPool(config, "keewidb-instance", 6379, 2000,
"your_password");
```

- **反面示例：** 三个参数差异过大，低峰期连接被大量回收，高峰期需重新建连，造成延迟毛刺。

```
config.setMaxTotal(200);
config.setMaxIdle(100);
config.setMinIdle(10);
```

更多参数详情请参见：

- **Go 语言 Redigo：** Redigo 官方文档 。
- **Java Jedis：** Jedis 官方文档 。

二、熔断机制

在高并发场景下，建议客户端集成熔断保护机制（如 Netflix Hystrix、Sentinel 等）。

1. 熔断的作用

在分布式架构中，当 KeeWiDB 集群的某个节点因磁盘 IO 异常、网络抖动或负载过高而响应变慢时，如果客户端持续向该节点发送请求，大量请求将因超时而堆积在线程池中，最终耗尽客户端的线程和连接资源，导致故障从单节点扩散至整个服务链路。熔断机制通过自动隔离故障节点，将影响限制在单点范围内，维持服务整体可用性。

2. 工作流程

熔断器在以下三个状态之间自动切换：

状态	触发条件	行为
关闭（正常）	错误率低于阈值	所有请求正常放行，持续统计错误率和超时率
打开（熔断）	错误率或超时率超过预设阈值	自动停止向故障节点发送请求，直接返回失败或将流量转向健康节点
半开（探测）	熔断持续时间超过冷却期	放行少量探测请求。若探测成功，切回关闭状态恢复正常流量；若仍失败，重新进入打开状态

3. 核心参数

参数	说明	参考值
错误率阈值	在统计窗口内，请求失败比例达到此值时触发熔断	50%
统计窗口	计算错误率的时间窗口大小	10s
最小请求数	统计窗口内请求数低于此值时不触发熔断，避免少量请求偶发失败导致误熔断	20 次
冷却期	熔断打开后等待多久进入半开状态	5s ~ 10s
半开探测数	半开状态下放行的探测请求数量	5 ~ 10次

说明：
以上参考值适用于一般业务场景。对延迟敏感的业务可缩短冷却期以加快恢复；对稳定性要求高的业务可降低错误率阈值以更早触发熔断。

4. 配置示例

Java Spring Cloud CircuitBreaker（Resilience4j）：

```
CircuitBreakerConfig config = CircuitBreakerConfig.custom()
    .failureRateThreshold(50)                    // 错误率阈值 50%
    .slidingWindowSize(10)                       // 统计窗口：最近 10 次请求
    .minimumNumberOfCalls(20)                    // 最小请求数 20 次
    .waitDurationInOpenState(Duration.ofSeconds(5))   // 冷却期 5 秒
    .permittedNumberOfCallsInHalfOpenState(5)         // 半开探测数 5 次
    .build();
```

```
CircuitBreaker breaker = CircuitBreaker.of("keewidb", config);
```

Go 语言 gobreaker：

```
cb := gobreaker.NewCircuitBreaker(gobreaker.Settings{
    Name:        "keewidb",
    MaxRequests: 5,                    // 半开探测数 5 次
    Interval:    10 * time.Second,     // 统计窗口 10 秒
    Timeout:     5 * time.Second,      // 冷却期 5 秒
    ReadyToTrip: func(counts gobreaker.Counts) bool {
        failureRatio := float64(counts.TotalFailures) /
float64(counts.Requests)
        return counts.Requests >= 20 && failureRatio >= 0.5  // 最小 20
次请求，错误率 50%
    },
})
```

三、密码安全

数据库访问密码用于控制数据库的访问权限。弱密码（如纯数字、常见单词）容易被暴力破解工具在短时间内攻破。密码泄露后，攻击者可访问或清空数据库中的数据。因此，密码必须满足以下复杂度要求：

要求	说明
长度	8 ~ 30个字符
复杂度	至少包含小写字母、大写字母、数字和特殊字符（`` `() ~!@#$%^&*-+=_
格式限制	不能以 `/` 开头

- **反面示例：** 以下密码过于简单，容易被暴力破解。

```
12345678          # 纯数字，暴力破解工具可在数秒内攻破
password          # 常见英文单词，在字典攻击列表中排名前列
keewidb           # 与产品名相关，极易被猜测
/Admin@2024       # 以 "/" 开头，不符合格式要求
```

- **正确示例：** 包含多种字符类型，长度充足，无规律可循。

```
Cx_order#2024Db    # 大小写字母 + 数字 + 特殊字符，16 位长度
Kw!9mP$3xR7n       # 随机组合，12 位长度，包含 4 种字符类型
```