云数据库 MongoDB

开发规范

腾讯云

文档目录

开发规范
连接与安全规范
连接方式决策指南

最近更新时间：2026-04-17 08:59:01

场景描述

在生产环境中，随着业务规模的增长，MongoDB 的部署架构通常会由副本集演进为分片集群。在管理不同架构的数据库接入与路由时，通常需要关注以下情况：

- 流量分配与负载均衡： 分片集群默认的负载均衡（LB）采用基于源 IP Hash 的策略。当客户端 IP 数量较少时（如少量应用服务器发起高并发请求），流量可能会集中路由至部分 Mongos 节点，导致节点间负载不均。
- 节点扩缩容与连接配置维护： 在对分片集群的 Mongos 节点进行扩缩容后，若应用端未及时同步更新连接串中的节点地址列表，新加入的节点将无法承接业务流量。
- 主从切换与拓扑感知： 当 Primary 节点发生切换时，若应用端未能及时感知拓扑架构的变化，持续向原主节点发送写请求，将导致写入操作失败。

连接方式总览

针对上述连接挑战，腾讯云 MongoDB 为分片集群分别提供了多种连接方式。不同方式在负载均衡策略、节点自动发现、运维复杂度等方面各有侧重，下表按推荐优先级进行汇总。

分片集群连接方式

分片集群因 Mongos 节点扩缩容较为常见，SRV 的自动发现能力优势突出，因此是分片集群的首选方案。LB 负载均衡地址在生产环境中客户端 IP 较多时表现良好，可满足绝大多数业务场景的需求。

优先级	连接方式	协议格式	核心优势
☆☆☆ 推荐	SRV 连接	`mongodb+srv://`	自动发现 Mongos 节点，扩缩容无需改连接串，驱动级负载均衡
☆☆ 次选	LB 负载均衡地址（默认）	`mongodb://`	接入简单，单 VIP 屏蔽后端拓扑，绝大多数生产场景均适用
☆ 补充	连接所有 Mongos	`mongodb://`	精准控制流量分配，适用于 LB 负载不均的特殊场景

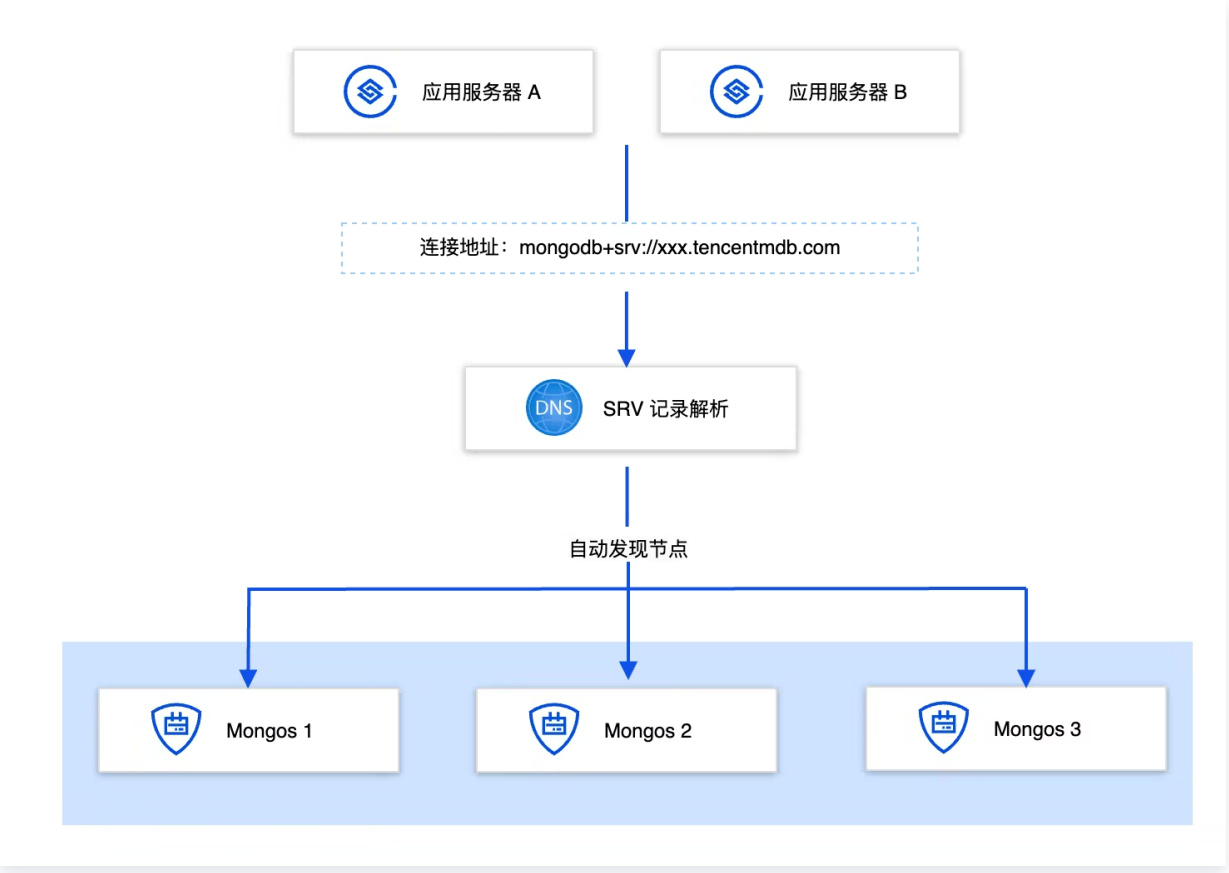
方式一：SRV 连接（分片集群首选）

SRV 连接通过 DNS 自动发现机制与 MongoDB 驱动的原生路由能力，实现灵活的节点接入与请求级负载均衡。在分片集群场景中，Mongos 节点扩缩容时驱动可自动感知变化，无需修改连接串、无需重启应用，在灵活性方面是分片集群的优选方案。副本集的节点拓扑一般保持稳定（Primary、Secondary、Hidden 节点数量固定），不涉及频繁的节点增减操作，使用控制台默认连接串即可满足需求。SRV 连接可作为副本集的可选方案，在连接串简洁性方面有一定优势。

连接原理

应用程序通过 mongodb+srv:// 协议连接时，驱动向 DNS 服务器发起 SRV 记录查询，动态获取当前集群中所有 Mongos 节点的地址列表。当集群进行扩缩容时，DNS 记录自动更新，驱动在下一次 DNS 刷新时（通常60秒内）即可感知并连接新节点，无需重启应用。

> **说明：**
> 驱动程序与所有存活的 Mongos 节点建立连接池，并在客户端按请求维度进行轮询分发，改善了传统网络 LB 基于源 IP Hash 带来的单点瓶颈。同时，MongoDB 官方驱动原生支持会话追踪，能够将 getMore 游标扫描和分布式事务的所有操作路由至同一个 Mongos 节点，在保证一致性的前提下实现负载均衡。



开通 SRV 连接模式

MongoDB 实例版本为4.0及以上，在 MongoDB 控制台 的实例详情中，定位至**网络配置**区域，单击**开通 SRV 连接模式**即可启用该连接方式。具体操作，请参见 开通 SRV 连接模式 。


SRV 访问地址

修改连接地址 关闭 SRV 连接模式

连接类型	访问地址（连接串）
访问读写主节点	mongodb+srv://mongouser:******@ .gz.tencentmdb.com/test?replicaSet=cmgo- _0&authSource=admin&ssl=false
优先读从节点	mongodb+srv://mongouser:******@r gz.tencentmdb.com/test?replicaSet=cmgo- _0&authSource=admin&ssl=false&readPreference=secondaryPreferred

业务应用

某游戏公司分片集群部署了8个 Mongos 节点，运营活动期间流量突增需紧急扩容至16个 Mongos。使用传统连接串的方式需要在应用配置中手动添加8个新节点地址，涉及20+微服务的配置变更和滚动重启，整个过程耗时2小时。改用 SRV 连接后，新增的 Mongos 节点被 DNS 自动解析，驱动在下一次 DNS 刷新时（通常60秒内）自动发现新节点，业务无感知、无需重启，扩容时间从2小时缩短至5分钟。

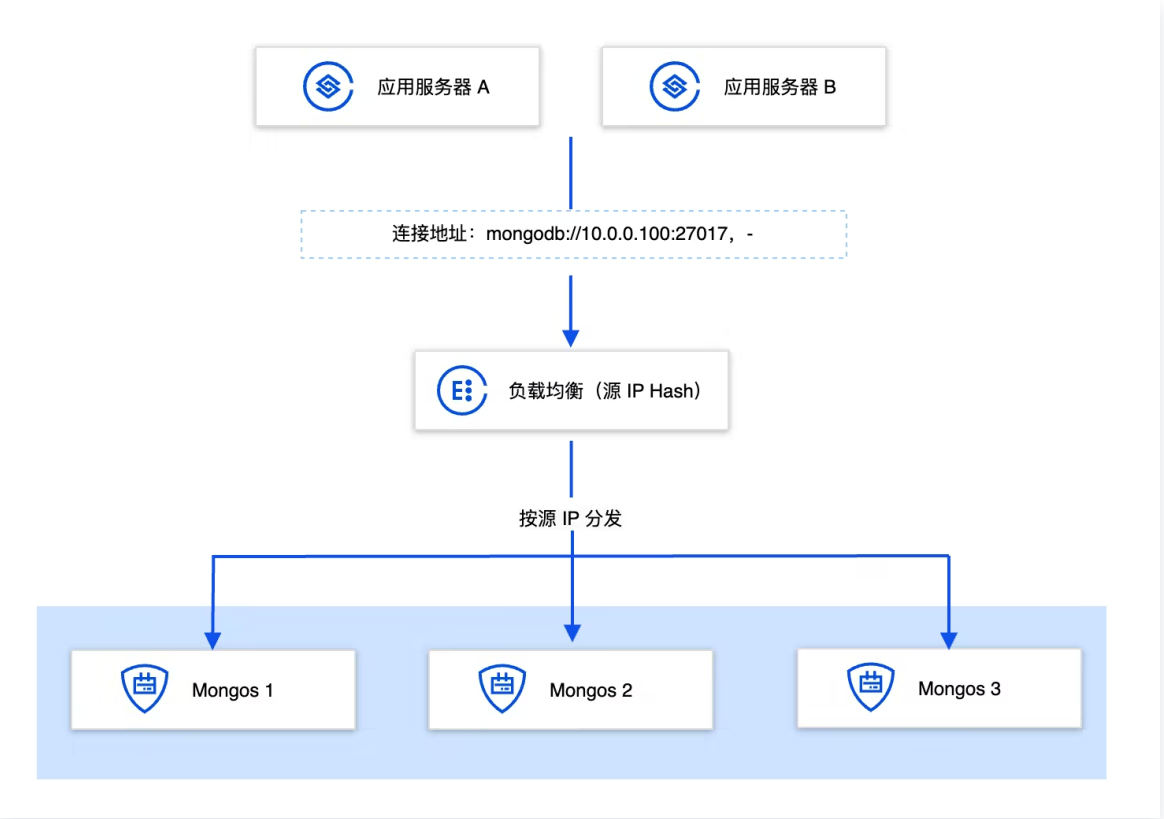
方式二：LB 负载均衡地址（分片集群默认连接）

腾讯云 MongoDB 分片集群实例默认提供负载均衡（LB）地址，通过单一 VIP 屏蔽后端多 Mongos 节点拓扑，接入简单、运维成本低，适合作为分片集群的默认连接方案。

连接原理

应用程序通过负载均衡服务提供的虚拟 IP（VIP）接入数据库，从而屏蔽了后端多个 Mongos 节点的真实内网地址（RSIP）。在请求分发上，负载均衡采用源 IP Hash 路由策略，确保来自同一客户端 IP 的请求始终被转发至固定的 Mongos 节点，以此满足 getMore 游标扫描及分布式事务对状态一致性的要求。同时，若后端 Mongos 节点发生变更（如故障替换），负载均衡将自动动态更新 VIP 与 RSIP 的映射关系。此时，应用层无需调整任何连接配置，即可实现底层节点切换对业务的完全无感知。

> **说明：**
> 负载均衡（LB）地址当前仅适用于分片集群架构。针对副本集架构，建议采用控制台提供的默认连接串（已包含全量节点地址）或 SRV 连接方式接入。此外，在客户端 IP 数量较少的场景下，若出现 Mongos 节点负载不均的现象，建议优先升级至 SRV 连接模式，或直接配置全量 Mongos 节点地址进行连接，以优化请求分发逻辑。



获取连接方式

实例创建成功之后，在 MongoDB 控制台 **实例详情**页面的**网络配置**区域，**访问地址**下便可以获取到 LB 方式的连接地址。

访问地址

连接类型	访问地址（连接串）
访问读写主节点	mongodb://mongouser:******@10.1.0.40:27017/test?authSource=admin
优先读从节点	mongodb://mongouser:******@10.1.0.40:27017/test?authSource=admin&readPreference=secondaryPreferred&readPreferenceTags=role-cmgo:primary-secondary-group
优先读从节点和只读节点	mongodb://mongouser:******@10.1.0.40:27017/test?authSource=admin&readPreference=secondaryPreferred

业务应用

以某 电商平台为例，因其底层业务系统的数据库驱动版本较旧，无法解析 mongodb+srv:// 协议，故不适用 SRV 连接模式。在此类场景下，采用负载均衡（LB）提供的 VIP 地址接入分片集群成为标准的替代方案。应用层仅需维护单一的 VIP 配置，即可实现对整个集群的透明访问；当 Mongos 节点进行变更或出现异常时，后端会自动完成故障节点的隔离与流量切换，无需应用端介入即可保障业务的连续性。

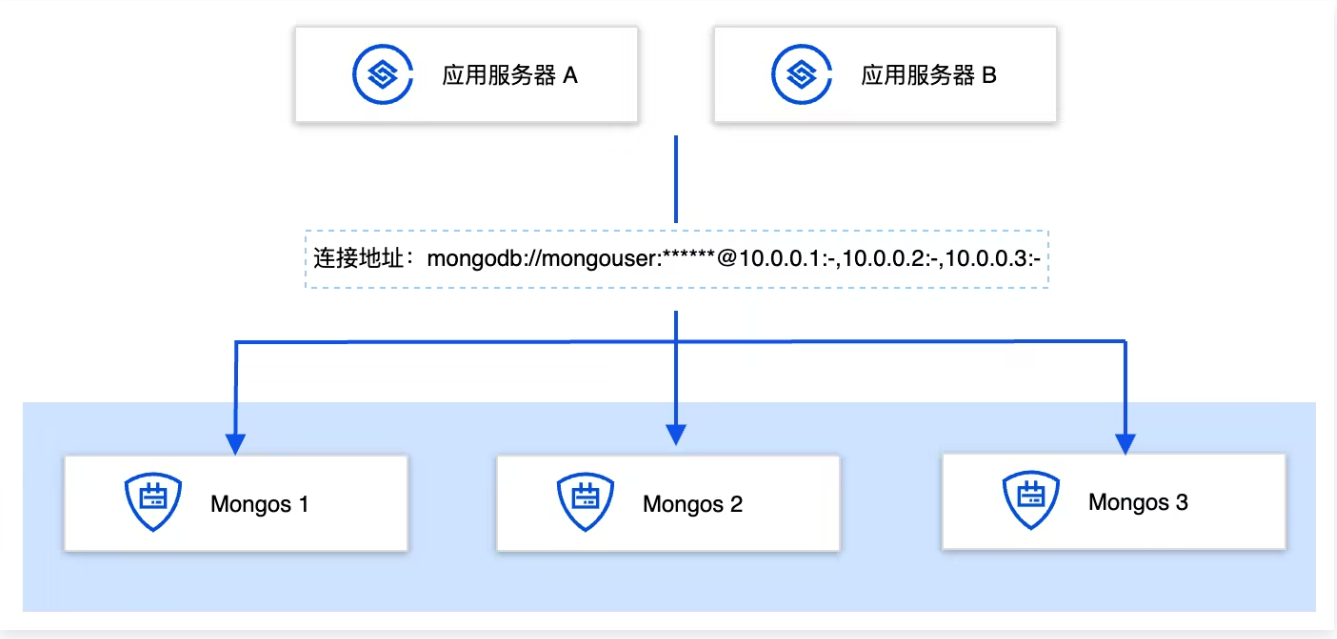
方式三：连接所有 Mongos 访问地址

分片集群支持开通 Mongos 访问，为每个 Mongos 节点分配一个由共享 VIP 与独立 VPORT 构成的固定地址。当某个 Mongos 节点发生故障时，系统将自动为其绑定新的 Mongos 进程，VIP 和 VPORT 保持不变，保障连接稳定性。

连接原理

系统为 Mongos 节点分配固定的虚拟 IP 与端口。底层机制实时探测节点状态，一旦发生故障，系统秒级自动将访问地址切换至健康节点。对外入口始终保持静态，上层应用无需修改配置即可实现透明自愈，保障了业务连接的持续稳定。

> **说明：**
> 如果您在连接串中配置了所有 Mongos 节点，当调整实例中的 Mongos 节点数后，需要在应用侧**同步**更新连接串。



开启 Mongos 访问地址

在 MongoDB 控制台 选择**分片实例**，进入实例详情 > **节点管理** > **Mongos 节点**页签，单击**开通 Mongos 访问地址**。具体操作，请参见 开通 Mongos 访问地址 。

连接方式对比与选择

腾讯云 MongoDB 针对分片集群各方式在接入复杂度、负载均衡效果、运维灵活性等方面各有侧重。以下通过对比表格，从多个决策维度进行横向比较，并给出对应的选择建议，帮助您快速确定适合当前业务场景的连接方案。

决策维度	SRV 连接	LB 地址	连接所有 Mongos
接入复杂度	低：一条短连接串	低：单 VIP	中：需列出所有节点
节点变更时	无需改连接串	无需改连接串	⚠ 必须更新连接串
均衡策略	驱动级均衡（连接级）	源 IP Hash	驱动级均衡（连接级）
客户端 IP 少时	✓ 均衡	⚠ 可能不均衡	✓ 均衡

连接数特征	连接数放大（连接所有 Mongos）	连接数不放大	连接数放大（连接所有 Mongos）

相关文档

- 腾讯云 MongoDB 连接实例
- 开通 SRV 连接模式
- Mongos 负载均衡与连接方案
- 开通 Mongos 访问地址
- 开启外网访问

连接配置规范

最近更新时间：2026-04-17 08:59:02

场景描述

当数据库的连接拓扑（网络路由）确立后，应用代码层面的连接池配置与安全防护将直接决定系统的稳定性与数据资产的安全。在日常开发与排障中，团队经常会因“配置不当”而遭遇以下高危场景：

- **连接数超限风险：** 在微服务架构下，若频繁使用短连接，或在多应用实例中将 maxPoolSize 设置过大，系统总并发连接数可能超出数据库的连接上限。这将导致新请求发生排队或超时，影响系统的整体可用性。
- **读写策略匹配不当：** 读写分离配置需与业务场景相匹配。若在核心交易中读取从库，可能因主从同步延迟导致数据读取不一致；而高频的分析与报表查询若未配置读写分离，则会显著增加主库（Primary）的性能负载。
- **数据安全与合规风险：** 在开发或测试过程中，部分业务可能会开启数据库的公网访问，且未配置复杂的鉴权密码。此类缺乏网络边界管控与权限最小化隔离的配置，不仅不符合安全合规要求，还会显著增加数据泄露、恶意篡改及外部勒索攻击的风险。

连接规范总览表

规范编号	规范名称	核心要求	适用范围	约束等级
规范一	正确配置连接串核心参数	• 必须使用高可用连接地址。 • 必须包含 `authSource=admin`。 • 副本集必须指定 `replicaSet`。 • 建议开启 `retryWrites` 和 `retryReads`。	副本集 / 分片集群	必须
规范二	建议使用连接池（长连接）	生产环境不建议使用短连接，建议使用连接池复用连接。	副本集 / 分片集群	必须
规范三	连接池参数合理配置	根据业务并发量计算 `maxPoolSize`，总连接数不超过实例上限的 80%	副本集 / 分片集群	必须
规范四	根据业务场景选择读偏好	核心强一致业务读 Primary，非敏感业务读 Secondary，读写分离建议 `secondaryPreferred`。	副本集 / 分片集群	建议
规范五	线上环境禁止关闭鉴权或随意暴露外网	• 必须开启鉴权，仅内网访问。	副本集 / 分片集群	必须

		• 外网必须通过 CLB 并配置安全组和 IP 限制。		

规范一：正确配置连接串核心参数

连接串（URI）参数的配置直接决定了数据库的可用性。请务必遵循以下四项核心准则：

1. 接入地址：严禁单点连接。
   - 指令：连接串必须包含集群内所有可用节点（Seed List）。
   - 原因：避免单点故障。仅指向单个节点将导致驱动程序无法在节点切换时自动发现新的 Primary，失去高可用能力。
2. 鉴权参数：明确 authSource。
   - 指令：必须显式指定鉴权数据库参数。
     - 通过控制台创建的用户：固定配置 authSource=admin。
     - 通过命令行创建的用户：配置为用户所属的逻辑数据库。
   - 原因：确保身份验证逻辑准确，避免因默认库不一致导致的连接认证失败。
3. 架构感知：必须携带 replicaSet。
   - 指令：副本集连接串必须包含 replicaSet=<副本集名称>。
   - 原因：只有显式指定该参数，驱动才能在故障发生时自动将流量重定向至新主节点，实现无感知切换。
4. 容灾增强：开启自动重试（推荐）。
   - 指令：建议追加参数 retryWrites=true 与 retryReads=true。
   - 原因：提升网络容错率。在发生瞬时网络抖动或主从选举期间，由驱动层自动发起重试，将故障影响降至最低。

连接串关键参数说明

参数	必填	说明	示例值
`authSource`	是	认证数据库，腾讯云统一为 admin	`admin`
`replicaSet`	副本集必填	副本集名称，从控制台获取	`cmgo-xxxxxxxx`
`readPreference`	建议	读偏好设置	`secondaryPreferred`
`maxPoolSize`	建议	连接池最大连接数	`150`
`retryWrites`	建议	自动重试写入	`true`
`retryReads`	建议	自动重试读取	`true`

`w`	核心业务必填	写确认级别	`majority`

业务应用

某金融系统使用副本集架构（1 Primary + 2 Secondary），连接串中未指定 replicaSet 参数。由于缺少该参数，MongoDB 驱动将连接视为 Standalone 模式，即直接连接到连接串中指定的单个节点，而不会感知副本集的整体拓扑结构。

- 日常运行期间未出现异常，但在一次计划内的运维操作中，Primary 节点发生主从切换，原 Primary 降级为 Secondary，另一个 Secondary 被选举为新 Primary。此时，驱动仍然持续向原节点（已降级为 Secondary）发送写请求，而 Secondary 节点默认拒绝写操作，导致业务端出现大量 not master 错误，写入失败持续约15分钟。
- 运维人员手动重启应用并更新连接地址后恢复。在连接串中添加 replicaSet=cmgo-xxxxxxxx 参数后，驱动以副本集模式运行，自动感知拓扑变化，在 Primary 切换后的数秒内自动将写请求路由至新 Primary，业务无需重启、无需修改连接地址，实现故障的自动恢复。

规范二：必须使用连接池（长连接）

生产环境建议使用连接池（长连接），不建议使用短连接模式。

- 短连接模式下，每次数据库请求都需要经历 TCP 三次握手、TLS 协商（如开启加密传输）和 SCRAM 鉴权等完整流程，单次连接建立耗时通常在10-50ms。
- 在高并发场景下，频繁的连接创建与销毁不仅大幅增加请求延迟，还会迅速耗尽实例的连接数上限，导致后续请求无法获取连接而超时失败。
- 连接池通过预先建立连接并在多次请求间复用，省去了重复的握手和鉴权开销，能够在高并发下保持连接数稳定可控、响应延迟在毫秒级。

短连接 vs 长连接对比

维度	短连接	长连接（连接池）
连接建立	每次请求新建	复用已建立连接
认证开销	每次都要认证（10-50ms）	仅首次认证
连接数	随 QPS 线性增长	稳定可控
适用场景	不适用于生产	所有生产环境

业务应用

某电商平台早期使用 PHP 短连接模式，每次数据库请求都新建连接，请求结束后立即销毁。日常流量（QPS 约 1000）下运行正常，但在大促期间 QPS 从 1000飙升至10000。由于每个请求都需要独占一条连接，连接建立速度远跟不上请求到达速度，MongoDB 实例的连接数在数分钟内从200飙升至3000（实例默认上限），随后所有新请求因无法获取连接而排队超时，接口平均响应时间从50ms恶化至5秒以上，大量订单提交失败。改用连接池

（maxPoolSize=200）后，应用启动时预先建立一批连接并在请求间复用，省去了重复的握手和认证开销。即使在次年大促 QPS 达到50000的情况下，实际连接数也稳定在500以内，接口响应时间始终保持在毫秒级。

> **说明：**
> 对于 Serverless、PHP-FPM 等无法在应用内存中维持长连接池的架构，建议在业务层与数据库之间引入中间件代理（Proxy）来实现连接池化。

规范三：连接池参数合理配置

连接池大小（maxPoolSize）并非越大越好。配置过大会导致全局连接数瞬间打满，新请求被数据库拒绝；配置过小则会在业务高峰期出现本地排队等待，徒增请求延迟。正确配置连接池，必须综合考量单次请求耗时、应用实例扩容边界以及网络接入拓扑。

不同连接方式的连接数计算

理论上的全局总连接数，不得超过 MongoDB 实例连接数上限的80%，以应对突发的业务扩容或主从重连。在使用分片集群时，底层驱动建立连接池的逻辑会因接入方式产生“乘数放大效应”。请在推算前明确以下三个关键变量：

- A (Application)：当前部署的业务应用总数。
- P (Pool Size)：单个应用配置的 maxPoolSize 值。
- M (Mongos)：通过 SRV 解析出或在连接串中指定的 Mongos 节点总数。

连接接入方式	驱动连接行为	理论总连接数计算公式
LB 负载均衡（内网 VIP 等）	驱动将 LB 视作单一节点，仅与 LB 建立一个连接池，由 LB 在底层分发请求。	A x P
SRV 记录连接	驱动会自动解析 SRV 记录，并与集群背后的每一个 Mongos 节点分别建立独立的连接池。	A x P x M
直连所有 Mongos	驱动会与连接串（URI）中显式配置的每一个 Mongos 节点分别建立独立的连接池。	A x P x M

maxPoolSize 参数推算建议

绝大多数连接数超限的故障，都源于开发者盲目使用默认值或将 QPS 错误等同于连接数。请按照以下步骤反推单个应用的合理 maxPoolSize：

步骤一：计算单个应用基础并发需量

并发连接数取决于请求量与处理速度。公式为：单个应用基础需量 = (单个应用预期峰值 QPS × 数据库平均响应耗时秒数) × 1.5（抗抖动系数）。例如，单实例峰值1000 QPS，平均每次查询耗时0.02秒（20ms），则基础并发需量为1000 × 0.02 × 1.5 = 30。

步骤二：结合拓扑架构计算最终值

- 如果是 LB 负载均衡模式：直接使用基础需量即可，即 maxPoolSize ≈ 30。
- 如果是 SRV 或直连多 Mongos 模式：由于请求会被驱动自动分散到 M 个节点的连接池中，为了防止全局总连接数按倍数超限，必须进行等比缩减。即 maxPoolSize ≈ 基础需量 ÷ M。

连接池参数配置参考

参数	说明	建议值	不当配置的后果
`maxPoolSize`	最大连接数	50–200	过大：超限；过小：排队
`minPoolSize`	最小连接数	5–20	过小：冷启动慢
`maxIdleTimeMS`	最大空闲时间	60000–300000	过小：频繁重建
`waitQueueTimeoutMS`	等待超时	5000–10000	过短：正常请求被拒
`connectTimeoutMS`	连接超时	10000–30000	过短：网络抖动时失败

业务应用

某内容平台部署了20个应用，通过 LB 负载均衡连接分片集群，每个应用的 maxPoolSize 配置为200（直接使用了驱动默认值）。理论总连接数为20 × 200 = 4000，实际上已经埋下了超过 MongoDB 实例连接数上限（3000）的隐患。

- 日常低峰期由于实际并发较低，连接池未被填满，运行正常。但在一次业务扩容中新增了5个应用，25个应用同时启动并初始化连接池，连接数瞬间接近 5000，触发实例连接数上限保护，后启动的5个应用全部报 `connection pool exhausted` 错误，无法连接数据库，新上线的功能模块完全不可用。
- 排障后，将 `maxPoolSize` 从200调整为100后，总连接数上限降为25 × 100 = 2500，占实例连接数上限的83%，并预留了后续扩容至30个应用的空间。同时设置 `minPoolSize=10` 确保冷启动时有足够的预热连接，解决了扩容时的连接雪崩问题。

规范四：根据业务场景选择读偏好

`readPreference`（读偏好）决定了 MongoDB 驱动将读请求发往哪个节点。不同业务场景对数据一致性的要求不同：核心交易（如转账、下单）需要读到最新数据，必须从 Primary 读取；而报表分析、历史查询等对实时性不敏感的场景，可以从 Secondary 读取以分担 Primary 压力。选择不当会导致两类问题：将强一致性要求的读请求发往 Secondary，业务可能读到过时数据引发逻辑错误；将所有读请求集中在 Primary，则 Secondary 节点资源闲置，Primary 在高并发下成为瓶颈。

读偏好选择决策表

> **说明：**
> 业务需要读写分离时建议使用 `secondaryPreferred`，可用性更高。如果有业务仅访问只读节点，建议配置两个或两个以上只读节点实现读请求负载均衡。只读节点连接串可直接在实例详情页面的网络配置中获取。

读偏好	一致性	适用场景
`primary`	强一致	转账后查余额、下单后查订单状态
`primaryPreferred`	强一致优先	一般业务，Primary 不可用时降级到 Secondary
`secondary`	最终一致	纯读场景、报表查询、数据分析
`secondaryPreferred`	最终一致优先	读多写少场景，分担 Primary 压力（推荐）
`nearest`	取决于节点	地理分布式部署，就近访问

业务应用

某银行的核心系统中，账户余额查询接口配置了 `readPreference=secondary`。用户在完成一笔转账（写入 Primary）后立即查询余额，由于主从同步存在 10-100ms 的延迟，Secondary 节点上的数据尚未更新，接口返回的仍是转账前的旧余额。用户看到转账成功但余额未变，频繁发起重复转账或拨打客服投诉。将该接口改为 `readPreference=primary` 后，余额查询直接从 Primary 读取，确保转账后立即返回准确余额，投诉量下降。而对于不要求实时性的历史账单查询和月度报表导出，仍保持 `readPreference=secondaryPreferred`，将约60%的读流量分流至 Secondary 节点，Primary 的 CPU 使用率从85%降至 50%，读写延迟均有所改善。

> **说明：**
> 启用 secondary 或 secondaryPreferred 时，业务代码必须能容忍因主从复制延迟带来的数据非强一致性（Stale Reads）风险。

规范五：线上环境禁止关闭鉴权或暴露外网

MongoDB 实例线上环境必须开启鉴权（Authentication），且仅允许通过内网（VPC）访问。未开启鉴权的实例相当于对所有能访问该 IP 的客户端完全开放读写权限，一旦被扫描发现将面临数据泄露或被勒索删库的风险。如业务确需外网访问，必须通过 CLB（负载均衡）配置外网访问服务，并同时满足以下条件：为 CLB 和 MongoDB 实例分别配置安全组，仅放通指定的客户端外网 IP；启用 SSL/TLS 加密传输，防止数据在公网传输过程中被截获；使用高强度密码（12 位以上，含大小写字母、数字和特殊字符），降低暴力破解风险。

安全配置要求

1. 网络边界隔离（VPC 与安全组）：默认仅限内网访问，公网访问必须经过严格的网络边界管控。
   - 默认内网直连：生产环境务必仅允许通过同一 VPC（私有网络）下的应用服务器进行内网访问。
   - 合规的公网访问架构：若业务（如跨地域调试、外部数据直连）确需公网访问，严禁为实例直接绑定公网 IP。必须通过 CLB（负载均衡）打通外网访问路径，并强制实行双层安全组管控：
     - CLB 安全组：仅放通特定客户端的外网 IP（如公司出口 IP）及监听器端口，实施第一道拦截。
     - MongoDB 安全组：仅放通指定的客户端外网 IP 以及27017端口，同时必须放通 CLB 实例的 VIP 地址（用于保障 CLB 节点对后端数据库的健康探测正常运行）。
2. 访问鉴权与最小化授权：禁止免密登录，应用账号与管理账号严格分离。
   - 强制开启鉴权：实例必须开启 Authentication 机制。
   - 高强度密码验证：数据库密码必须大于12位，且包含大小写字母、数字及特殊字符。杜绝使用 admin123、root 等弱口令，以防暴力破解。
   - 遵循最小权限原则（Least Privilege）：禁止业务应用直接使用 root 或拥有 dbAdminAnyDatabase 权限的超级账号连接数据库。必须为每个具体的业务模块创建专用的数据库账号，并仅授予对应业务库的 readWrite 权限。
3. 传输层加密（TLS/SSL）：凡是涉及公网链路的传输，必须加密。

   防抓包与中间人攻击：一旦启用了公网访问链路，必须在控制台开启 SSL/TLS 加密传输功能。并在应用端连接串中显式追加 tls=true，确保数据在不可信的公网传输过程中不会被明文截获或篡改。

安全配置自检清单

检查维度	检查项	合规标准
网络层	公网暴露面	仅允许内网访问；公网需求必须通过 CLB 代理并配置白名单。
网络层	安全组策略	严禁配置0.0.0.0/0全放通，仅允许明确的源 IP 访问。
认证层	身份鉴权	必须开启鉴权功能。
认证层	密码复杂度	长度 ⩾ 12 位，包含大小写字母、数字及特殊符号。
授权层	最小权限	应用账号仅具备目标库的读写权限，无管理权限。
传输层	SSL/TLS 加密	开启外网访问时，必须强制启用 SSL/TLS。
应用层	凭据安全存储	数据库账号密码禁止硬编码在代码中，应使用环境变量或 KMS 服务。

相关文档

- 腾讯云 MongoDB 连接实例

- 开通 SRV 连接模式
- Mongos 负载均衡与连接方案
- 开启外网访问

SDK 连接配置

最近更新时间：2026-04-17 08:59:02

正确的 SDK 连接配置是保障 MongoDB 服务稳定运行的基础。连接串中的参数直接决定了鉴权方式、故障切换行为、连接池效率和读写可靠性——任何一项配置缺失或不当，都可能在生产环境引发连接失败、性能下降或数据一致性问题。本文提供 Python、Java、Node.js、Go 四种主流 SDK 的连接配置示例，所有示例均已内置连接池、超时、重试、写关注（`w=majority`）和读偏好（`secondaryPreferred`）等生产环境推荐配置，可直接用于生产环境，也可根据实际业务需求调整参数值。连接串中的用户名、密码和地址请替换为实际实例信息（可在腾讯云控制台 > MongoDB > 实例详情页获取）。

Python（pymongo）

Python 生态中，`pymongo` 是 MongoDB 官方维护的驱动库，提供同步连接模式，适用于 Web 后端（如 Flask、Django）和数据处理脚本等场景。以下示例展示了四种连接方式的完整配置，包含连接池、超时、重试和读写策略等生产环境推荐参数。`pymongo` 的连接池由驱动内部自动管理，通过 `maxPoolSize` 和 `minPoolSize` 控制连接数量，无需手动创建或释放连接。

```
from pymongo import MongoClient
from pymongo.read_preferences import ReadPreference

# 分片集群：SRV 连接（自动发现 Mongos，扩缩容无需改连接串）
client = MongoClient(
    "mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin",
    authSource="admin",

    # 连接池配置
    maxPoolSize=150,
    minPoolSize=10,
    maxIdleTimeMS=120000,
    waitQueueTimeoutMS=5000,

    # 超时配置
    connectTimeoutMS=10000,
    serverSelectionTimeoutMS=5000,

    # 可靠性配置
    retryWrites=True,
    retryReads=True,
    w="majority",
```

```
    # 读偏好
    readPreference=ReadPreference.SECONDARY_PREFERRED
)

# 副本集：使用控制台默认连接串（包含所有节点地址）
client = MongoClient(

"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.101:27017,10.0.0.1
02:27017/admin",
    authSource="admin",
    replicaSet="cmgo-xxxxxxxx",
    maxPoolSize=150,
    retryWrites=True,
    w="majority",
    readPreference=ReadPreference.SECONDARY_PREFERRED
)

# 分片集群：LB 地址连接（默认方式，绝大多数场景适用）
client = MongoClient(
    "mongodb://mongouser:password@10.0.0.100:27017/admin",
    authSource="admin",
    maxPoolSize=150,
    retryWrites=True,
    w="majority",
    readPreference=ReadPreference.SECONDARY_PREFERRED
)

# 分片集群：连接所有 Mongos（需先开通 Mongos 访问地址）
client = MongoClient(

"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.100:27018,10.0.0.1
00:27019/admin",
    authSource="admin",
    maxPoolSize=150,
    retryWrites=True,
    w="majority",
    readPreference=ReadPreference.SECONDARY_PREFERRED
)
```

```
# 连接健康检查
def check_connection():
    try:
        client.admin.command('ping')
        print("MongoDB 连接正常")
        return True
    except Exception as e:
        print(f"MongoDB 连接失败: {e}")
        return False
```

Java（MongoDB Driver）

Java 生态中，`mongodb-driver-sync` 是 MongoDB 官方提供的同步驱动，广泛应用于 Spring Boot、微服务等企业级项目。Java 驱动通过 `MongoClientSettings` 构建器模式配置连接参数，将连接池、超时、写关注和读偏好等配置统一管理。以下示例将四种连接方式封装为独立方法，共享同一套 `MongoClientSettings` 配置，便于在不同架构间切换。连接池由驱动内部维护，应用只需调用对应方法获取 `MongoClient` 实例即可，无需手动管理连接生命周期。

```
import com.mongodb.ConnectionString;
import com.mongodb.MongoClientSettings;
import com.mongodb.ReadPreference;
import com.mongodb.WriteConcern;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoClients;
import java.util.concurrent.TimeUnit;

public class MongoDBConfig {

    // 分片集群: SRV 连接
    public static MongoClient createClientWithSRV() {
        String uri =
"mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin"
                    + "?authSource=admin";
        return createClientFromUri(uri);
    }

    // 副本集：控制台默认连接串（包含所有节点地址）
    public static MongoClient createClientWithReplicaSet() {
        String uri = "mongodb://mongouser:password"
```

```
                    +
"@10.0.0.100:27017,10.0.0.101:27017,10.0.0.102:27017/admin"
                    + "?authSource=admin&replicaSet=cmgo-xxxxxxxx";
        return createClientFromUri(uri);
    }

    // 分片集群：LB 地址连接（绝大多数场景适用）
    public static MongoClient createClientWithLB() {
        String uri =
"mongodb://mongouser:password@10.0.0.100:27017/admin"
                    + "?authSource=admin";
        return createClientFromUri(uri);
    }

    // 分片集群：连接所有 Mongos
    public static MongoClient createClientWithAllMongos() {
        String uri = "mongodb://mongouser:password"
                    +
"@10.0.0.100:27017,10.0.0.100:27018,10.0.0.100:27019/admin"
                    + "?authSource=admin";
        return createClientFromUri(uri);
    }

    private static MongoClient createClientFromUri(String uri) {
        MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(new ConnectionString(uri))

            // 连接池配置
            .applyToConnectionPoolSettings(builder -> builder
                .maxSize(150)
                .minSize(10)
                .maxConnectionIdleTime(120, TimeUnit.SECONDS)
                .maxWaitTime(5, TimeUnit.SECONDS))

            // 超时配置
            .applyToSocketSettings(builder -> builder
                .connectTimeout(10, TimeUnit.SECONDS)
                .readTimeout(30, TimeUnit.SECONDS))

            .applyToClusterSettings(builder -> builder
```

```
                .serverSelectionTimeout(5, TimeUnit.SECONDS))

            // 可靠性配置
            .retryWrites(true)
            .retryReads(true)
            .writeConcern(WriteConcern.MAJORITY)
            .readPreference(ReadPreference.secondaryPreferred())

            .build();

        return MongoClients.create(settings);
    }
}
```

Node.js（MongoDB Driver）

Node.js 生态中，`mongodb` 是官方驱动包，基于异步 I/O 模型，适合高并发的 Web 服务（如 Express、Koa、NestJS）和 Serverless 函数等场景。以下示例将连接池、超时、重试和读写策略等通用参数提取为 `commonOptions` 对象，四种连接方式共享同一套配置，切换连接方式时只需修改 URI 即可。Node.js 驱动的连接池同样由驱动自动管理，每次调用 `client.connect()` 后即可复用连接池中的连接，无需手动创建或关闭单个连接。

```
const { MongoClient, ReadPreference } = require('mongodb');

// 通用连接配置
const commonOptions = {
    maxPoolSize: 150,
    minPoolSize: 10,
    maxIdleTimeMS: 120000,
    waitQueueTimeoutMS: 5000,
    connectTimeoutMS: 10000,
    serverSelectionTimeoutMS: 5000,
    retryWrites: true,
    retryReads: true,
    w: 'majority',
    readPreference: ReadPreference.SECONDARY_PREFERRED
};

// 分片集群：SRV 连接
async function connectWithSRV() {
```

```
    const uri =
"mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin?
authSource=admin";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    await client.db('admin').command({ ping: 1 });
    console.log('SRV 连接成功');
    return client;
}

// 副本集：控制台默认连接串
async function connectReplicaSet() {
    const uri =
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.101:27017,10.0.0.1
02:27017/admin?authSource=admin&replicaSet=cmgo-xxxxxxxx";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    console.log('副本集连接成功');
    return client;
}

// 分片集群：LB 地址连接（绝大多数场景适用）
async function connectWithLB() {
    const uri = "mongodb://mongouser:password@10.0.0.100:27017/admin?
authSource=admin";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    console.log('LB 连接成功');
    return client;
}

// 分片集群：连接所有 Mongos
async function connectWithAllMongos() {
    const uri =
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.100:27018,10.0.0.1
00:27019/admin?authSource=admin";
    const client = new MongoClient(uri, commonOptions);
    await client.connect();
    console.log('全 Mongos 连接成功');
    return client;
```

```
}
```

Go（mongo-driver）

Go 生态中，`mongo-driver` 是 MongoDB 官方维护的驱动库，适用于高性能后端服务和微服务架构。Go 驱动通过 `options.Client()` 链式调用配置连接参数，支持连接池、超时、重试和读写策略等完整配置项。以下示例将通用配置封装在 `createClient` 函数中，四种连接方式只需传入不同的 URI 即可复用同一套配置逻辑。Go 驱动要求在连接时传入 `context.Context` 以控制超时，建议在应用启动时创建一次 `mongo.Client` 实例并在整个生命周期中复用，避免重复建立连接。

```
package main

import (
    "context"
    "log"
    "time"

    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
    "go.mongodb.org/mongo-driver/mongo/readpref"
    "go.mongodb.org/mongo-driver/mongo/writeconcern"
)

// 片集群：SRV 连接
func createClientWithSRV() (*mongo.Client, error) {
    uri := "mongodb+srv://mongouser:password@xxx.tencentcdb.com/admin?
authSource=admin"
    return createClient(uri)
}

// 副本集：控制台默认连接串（包含所有节点地址）
func createClientWithReplicaSet() (*mongo.Client, error) {
    uri :=
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.101:27017,10.0.0.1
02:27017/admin" +
            "?authSource=admin&replicaSet=cmgo-xxxxxxxx"
    return createClient(uri)
}

// 分片集群：LB 地址连接（绝大多数场景适用）
```

```
func createClientWithLB() (*mongo.Client, error) {
    uri := "mongodb://mongouser:password@10.0.0.100:27017/admin?
authSource=admin"
    return createClient(uri)
}

// 分片集群：连接所有 Mongos
func createClientWithAllMongos() (*mongo.Client, error) {
    uri :=
"mongodb://mongouser:password@10.0.0.100:27017,10.0.0.100:27018,10.0.0.1
00:27019/admin" +
            "?authSource=admin"
    return createClient(uri)
}

func createClient(uri string) (*mongo.Client, error) {
    clientOpts := options.Client().ApplyURI(uri).
        // 连接池配置
        SetMaxPoolSize(150).
        SetMinPoolSize(10).
        SetMaxConnIdleTime(120 * time.Second).

        // 超时配置
        SetConnectTimeout(10 * time.Second).
        SetServerSelectionTimeout(5 * time.Second).

        // 可靠性配置
        SetRetryWrites(true).
        SetRetryReads(true).
        SetWriteConcern(writeconcern.Majority()).
        SetReadPreference(readpref.SecondaryPreferred())

    ctx, cancel := context.WithTimeout(context.Background(),
10*time.Second)
    defer cancel()

    client, err := mongo.Connect(ctx, clientOpts)
    if err != nil {
        return nil, err
    }
```

```
    // 连接健康检查
    if err := client.Ping(ctx, readpref.Primary()); err != nil {
        return nil, err
    }
    log.Println("MongoDB 连接正常")

    return client, nil
}
```

数据建模与 Schema 设计规范

最近更新时间：2026-04-17 14:53:31

操作场景

MongoDB 灵活的文档模型有助于业务快速迭代。但在实际应用中，这种灵活性需要配合合理的设计规范，才能保障系统的长期稳定。在过往的线上运营经验中，如果在初期缺乏一定的建模引导，以下几种常见的数据建模误区可能会在业务高速增长期带来性能挑战：

- 命名规范发散： 同一项目中 OrderDetail、order_detail、orderdetail 等命名风格共存，增加了后期代码维护和数据排查的沟通成本。
- 数组无界限增长： 习惯性地将持续产生的子数据（如用户的所有历史登录日志）不断 push 到同一个文档的数组中。如果缺乏截断机制，可能会触碰 MongoDB 单个 BSON 文档16MB的物理上限，导致部分写入请求受阻。
- 字段类型漂移： 同一个字段（如 age）混杂着数字与字符串类型，不仅增加了前端解析的复杂度，在特定情况下还可能影响底层索引的效率，导致查询结果不符合预期。
- 集合划分过细： 如果单个集群表过多(超过5000)，会加重底层 WiredTiger 存储引擎的元数据管理负担，可能会拉长实例的启动时间，并在极端情况下增加内存开销。
- 过度依赖拆表关联： 习惯性地沿用关系型数据库思维，将“订单主表”和“订单明细表”强行拆分为两个集合。这把原本可以通过单次内嵌读取（Embedding）完成的操作，退化为了多次网络 I/O 的往返调用。

本文档的目标：帮助您充分发挥 MongoDB 文档模型的优势，避免用关系型数据库的思维来设计 MongoDB。

命名规范

规范一：严禁使用系统库名

- **核心动作：**

  业务数据绝对禁止占用或混用 MongoDB 的系统保留库（包含但不限于 admin、local、config）。

规则	要求	正确示例	错误示例
红线	隔离业务数据与系统元数据	db_config（独立业务库）	admin、local、config

- **业务应用：**

  某团队为了图方便，直接在 admin 库中创建了业务集合来存储系统配置数据。由于 admin 库承担着用户鉴权与集群管理的核心职能，在执行管理类操作时可能需要获取高优先级的系统锁，与业务读写操作产生竞争，导致整个数据库响应急剧变慢。后续将配置数据迁移至独立的业务库后，性能才得以恢复。

规范二：数据库命名

- **核心动作：**

数据库名建议使用 db_ 前缀 + 小写字母 + 下划线。

- **命名规则与示例：**

规则	要求	正确示例	错误示例
前缀	建议 db_ 开头	db_order	Order
字符集	小写字母 + 下划线	db_user_center	db.user.center
长度	≤ 64 字节	db_payment	db-payment

- **业务应用：**

某研发团队创建了名为 UserCenter 的数据库，但在生产环境（Linux）时，部分微服务的连接字符串被误写为了 usercenter。由于 MongoDB 在 Linux 环境下对数据库名严格区分大小写，导致生产环境意外生成了一个全新的空库，核心业务查询大面积报空指针错误。此外，由于没有统一的前缀，DBA 在同一集群中清理历史废弃的临时库时，难以快速辨别哪些是核心业务库。全面实施 db_ 前缀加纯小写字母（如 db_user_center）的规范后，跨平台部署的库名大小写问题被根除，运维的安全边界也严密可控。

规范三：集合命名

- **核心动作：**

建议集合名使用 `t_` 前缀 + 小写字母 + 下划线，采用"模块_实体"格式。

- **命名规则与示例：**

规则	要求	正确示例	错误示例
前缀	建议 `t_` 开头	`t_order_detail`	`OrderDetail`
格式	模块_实体	`t_user_address`	`t_user-address`
禁用	不以 `system.` 开头	`t_system_config`	`system.config`
分表	时间后缀	`t_log_202403`	`t_log$202403`

- **业务应用：**

某项目使用 system.orders 作为订单集合名。由于 system. 是 MongoDB 保留前缀，某些底层管理操作会误将其识别为系统内部集合而直接跳过，最终导致自动备份数据不完整，改名为 t_orders 后问题解决。

规范四：字段命名

- **核心动作：**

建议统一采用 camelCase（小驼峰）或 snake_case（蛇形）命名域。字段设计需做到“自解释”，避免过度描述。严禁使用无意义的缩写。

- **字段命名对比：**

```
// 推荐：语义清晰、风格统一
{
    "_id": ObjectId("..."),
    "userName": "张三",              // camelCase 风格
    "createTime": ISODate("..."),
    "orderItems": [...],
    "totalAmount": 199.00
}

// 不推荐：命名混乱
{
    "_id": ObjectId("..."),
    "UN": "张三",                    // 缩写不清晰
    "Create_Time": ISODate("..."),  // 混合风格
    "oi": [...],                     // 含义不明
    "_total": 199.00                 // 业务字段以 _ 开头，易与系统字段冲突
}
```

- **业务应用：**

  某团队字段命名不统一，同一个集合中有 `createTime`、`Create_Time`、`create_time`、`CT` 四种写法表示创建时间。开发人员经常写错字段名导致查询结果为空，排查耗时从分钟变成小时。统一命名后，开发效率显著提升。

文档设计规范

规范五：控制单文档大小

- **核心动作：**单文档大小建议控制在100KB以内，不超过16MB限制。文档体积越大，对读写性能、内存占用和网络传输的影响越显著。
- **大文档处理策略：**

场景	解决方案	示例
数组元素过多	拆分为多个文档	用户动态：每条动态一个文档
存储大文件	使用 GridFS	图片、视频、大型日志
大文本内容	业务层压缩	HTML 内容压缩后存储
超大文件	对象存储 + URL 引用	文件存 COS，MongoDB 存 URL

- **检测文档大小的方法：**

```
// 查看单个文档大小
Object.bsonsize(db.collection.findOne({ _id: xxx }))

// 查看集合平均文档大小（单位：字节）
db.collection.stats().avgObjSize
```

- **业务应用：**

  某社交平台将用户的所有动态存储在用户文档的 `posts` 数组中。活跃用户发布数千条动态后，文档超过16MB，新动态无法写入，用户投诉发帖失败。改为每条动态独立文档 + 用户 ID 关联后，问题彻底解决。

规范六：控制嵌套层级

- **核心动作：**

  文档的嵌套层级建议控制在3-5层以内，避免过深的嵌套逻辑。

```
// 推荐：嵌套层级适中（2-3 层）
{
    "_id": ObjectId("..."),
    "orderId": "ORD202403001",
    "customer": {                    // 第 1 层
        "name": "张三",
        "contact": {                 // 第 2 层
            "phone": "13800138000",
            "email": "zhangsan@example.com"
        }
    },
    "items": [{ "productId": "P001", "quantity": 2 }]  // 第 1 层（数组）
}

// 不推荐：嵌套过深
{
    "level1": {
        "level2": {
            "level3": {
                "level4": {
                    "level5": {
                        "data": "太深了，无法维护"
                    }
```

```
                    }
                }
            }
        }
    }
```

- **业务应用：**

  某配置系统设计了8层嵌套的配置结构，修改最内层配置需要构造复杂的 `$set` 路径如 `"a.b.c.d.e.f.g.value"` 。开发人员频繁写错路径导致配置更新失败，且无法为深层字段建立有效索引。扁平化重构后，配置更新和查询都变得简单高效。

规范七：嵌入式 vs 引用式设计

- **核心动作：**

  优先考虑嵌入式设计，仅在数据量庞大或频繁独立更新等必要场景下使用引用式设计。

- **设计决策表：**

考量因素	优先选择【嵌入式】	优先选择【引用式】
读取模式	数据总是一起读取	数据经常需要单独读取
数据量	子数据量小且规模有限	子数据量大或呈无限增长趋势
更新频率	子数据很少发生独立更新	子数据频繁发生独立更新
关系类型	一对一、一对少量	一对多、多对多
共享性	子数据仅从属于一个父文档	子数据被多个文档高频共享

- **嵌入式设计示例：**

```
// 订单 + 订单项：嵌入式（总是一起查询，订单项不会独立存在）
{
    "_id": ObjectId("..."),
    "orderId": "ORD202403001",
    "customerId": "C001",
    "items": [
        { "productId": "P001", "name": "商品A", "quantity": 2,
"price": 99.00 },
        { "productId": "P002", "name": "商品B", "quantity": 1,
"price": 199.00 }
    ],
```

```
    "totalAmount": 397.00,
    "status": "paid",
    "createTime": ISODate("2024-03-15T10:30:00Z")
}
```

- **引用式设计示例：**

```
// 用户 + 文章：引用式（文章经常独立查询，且数量无限增长）
// 用户文档
{
    "_id": ObjectId("user_001"),
    "userName": "张三",
    "email": "zhangsan@example.com"
}

// 文章文档（通过 authorId 引用用户）
{
    "_id": ObjectId("article_001"),
    "title": "MongoDB 最佳实践",
    "authorId": ObjectId("user_001"),  // 引用用户
    "content": "...",
    "createTime": ISODate("...")
}
```

- **混合模式**（Hybrid）——将高频访问的子数据冗余嵌入，同时保留引用关系。

```
// ✅ 混合模式：订单中冗余商品名称和价格（快照），同时保留 productId 引用
{
    "orderId": "ORD001",
    "items": [
        {
            "productId": ObjectId("..."),  // 引用（用于关联最新商品信息）
            "name": "商品A",                // 冗余（下单时快照，避免商品改
名影响历史订单）
            "price": NumberDecimal("99.00") // 冗余（下单时价格快照）
        }
    ]
}
```

- **业务应用：**

  某电商系统将订单和订单项分为两个集合（关系型思维），查询订单详情需要先查订单、再查订单项、最后应用层拼接。大促期间接口延迟从50ms飙升至 800ms。改为嵌入式设计（订单项嵌入订单文档）后，单次查询即可返回完整数据，延迟降至30ms，代码也大幅简化。

规范八：数组设计注意事项

- **核心动作：**

  单个数组元素数量建议不超过1000，严禁设计无限增长的数组。

- **无限增长数组 vs 独立文档关联示例：**

```
// 不推荐：无边界、无限增长的数组

{
    "userId": "user_10001",
    "orders": [
        { "orderId": "ORD_001", "amount": 99.00, "date":
ISODate("...") },
        { "orderId": "ORD_002", "amount": 158.00, "date":
ISODate("...") },
        // ... 活跃用户可能累积数万条订单，触发 16MB 限制
    ]
}

// 推荐：将数组元素拆分为独立文档，通过外键关联
// 用户文档
{
    "userId": "user_10001",
    "name": "张三",
    "orderCount": 1024
}

// 订单文档（独立集合）
{
    "orderId": "ORD_001",
    "userId": "user_10001",    // 外键关联
    "amount": 99.00,
    "date": ISODate("2024-03-15T10:30:00Z")
}
```

- **业务应用：**

  某 IoT 平台将传感器的所有读数存储在设备文档的 `readings` 数组中。运行一年后，活跃设备的数组包含数十万条读数，文档超过16MB无法写入新数据。改用"分桶模式"（每小时一个文档）后，单文档大小稳定在100KB以内。

> **说明：**
> - 对于只需保留最近 N 条记录的场景（例如只保留最近10次登录记录），建议在写入时使用 $push 结合 $slice 修饰符，在数据库层面自动维护数组的固定长度上限，避免应用层多次读取与截断。
> - IoT 或监控等时序数据，MongoDB 5.0+已经推出了原生 Time Series Collections，优先选择 MongoDB 原生的 Time Series 集合，而非手动实现分桶模式，可以获得更高的压缩率和查询性能。

数据类型规范

规范九：选择正确的数据类型

- **核心动作：**

  日期用 Date 类型，金额用 Decimal128，ID 用 ObjectId 或递增 Long。
- **数据类型选择表：**

场景	推荐类型	不推荐类型	潜在问题
日期时间	Date	字符串	无法使用原生的日期运算和范围查询优化
金融金额	Decimal128	Double	浮点精度丢失，导致账务对账出现差异
文档主键	ObjectId（默认）	随机字符串	非递增的随机 ID 会导致频繁的页分裂，严重拖慢写入性能。ObjectId 的前4字节为秒级时间戳，具备大致递增特性，使得 B-Tree 索引的写入集中在尾部，避免了随机插入导致的页分裂
大整数 ID	NumberLong	字符串	无法进行数值比较和范围排序
状态标志	String（枚举值）	数字	魔术数字（Magic Number）含义不明确，后期维护困难

- **数据类型使用示例：**

```
// 正确的类型使用
{
    "_id": ObjectId("65f3a2b8c1d2e3f4a5b6c7d8"),  // ObjectId
```

```
    "orderId": NumberLong("20240315000001"),        // 大整数
    "amount": NumberDecimal("199.99"),              // 金额用 Decimal128
    "createTime": ISODate("2024-03-15T10:30:00Z"), // 日期用 Date
    "status": "paid"                                // 状态用字符串枚举
}

// 错误的类型使用
{
    "_id": "random-uuid-string",           // 随机字符串影响性能
    "orderId": "20240315000001",           // 字符串无法数值排序
    "amount": 199.99,                      // Double 有精度问题
    "createTime": "2024-03-15 10:30:00",   // 字符串无法日期运算
    "status": 1                            // 数字含义不明
}
```

- **业务应用：**

  某金融系统用 Double 类型存储金额，计算 `0.1 + 0.2` 得到 `0.30000000000000004`，累计计算后与银行对账差异达数百元。改用 Decimal128 后，计算精确到分，对账完全准确。

规范十：_id 字段使用规范

- **核心动作：**

  除非有特殊需求，首选默认的 ObjectId 作为 _id；若业务需要自定义 _id，建议具备递增特性。

- **使用示例：**

```
// 推荐：使用默认 ObjectId
{ "_id": ObjectId("65f3a2b8c1d2e3f4a5b6c7d8") }

// 可以：自定义递增 ID（需确保递增特性）
{ "_id": NumberLong("20240315000001") }

// 禁止：随机字符串（影响写入性能）
{ "_id": "550e8400-e29b-41d4-a716-446655440000" }
```

- **业务应用：**

  某系统使用随机 UUID 作为主键，随数据量增长，索引树的随机写入导致磁盘 I/O 剧增并触发频繁的页分裂，写入 QPS 从10,000锐减至3,000。改为单调递增的 ObjectId 后，利用顺序追加特性消除了 I/O 瓶颈，性能恢复正常。

Schema 验证

规范十一：为核心集合配置 Schema 验证

- **核心动作：**

  通过 MongoDB 提供的 JSON Schema 验证功能，在数据库引擎层面确保数据写入的类型与格式一致性。

  > **说明：**
  > - `validationLevel: "moderate"` 模式：只验证新写入和被更新的文档，不验证已有文档（适合存量数据迁移场景）。
  > - Schema 验证对**写入性能有一定影响**（通常 < 5%），高频写入集合需评估。
  > - 修改已有集合的验证规则用 `collMod` 命令。

- **Schema 验证示例：**

```
// 创建带验证规则的集合
db.createCollection("t_users", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["userName", "email", "createTime"],
            properties: {
                userName: {
                    bsonType: "string",
                    minLength: 2,
                    maxLength: 50,
                    description: "用户名，必填，2-50 个字符"
                },
                email: {
                    bsonType: "string",
                    pattern: "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-
zA-Z]{2,}$",
                    description: "邮箱，必填，需符合邮箱格式"
                },
                age: {
                    bsonType: "int",
                    minimum: 0,
                    maximum: 150,
                    description: "年龄，可选，0-150 的整数"
                },
                status: {
```

```
                    enum: ["active", "inactive", "deleted"],
                    description: "状态，枚举值"
                },
                createTime: {
                    bsonType: "date",
                    description: "创建时间，必填"
                }
            }
        }
    },
    validationLevel: "strict",     // strict: 所有写入都验证
    validationAction: "error"      // error: 验证失败则拒绝写入
});
```

- **业务应用：**

  某电商系统的 `price` 字段没有类型约束，有的存数字 `99.00`，有的存字符串 `"99.00"`，甚至有存对象 `{value: 99}`。价格排序和比较完全混乱，促销活动的"满100减20"逻辑失效。添加 Schema 验证后，脏数据被拒绝写入，存量数据清洗后功能恢复正常。

容量规划

规范十二：控制集合数量

- **核心动作：**

  单个实例的总集合数量建议不超过5000个。

- **集合数量过多的影响：**

影响	说明
实例启动时间长	引擎启动时需要逐一加载所有集合的元数据信息
内存占用高	每个集合的元数据都会常驻缓存，挤占业务内存
文件句柄消耗	每个集合对应多个底层数据文件，易触碰系统上限
运维复杂	状态监控、数据迁移、大版本升级等日常操作的耗时将呈指数级成倍增加
备份超时或失败	遍历海量元数据极易导致物理备份任务严重超时，甚至可能完全无法生成物理备份

- **业务应用：**

  某物联网平台初期采用“一设备一集合”模式，随着业务增长，单库产生超过50,000个集合。海量元数据导致内存开销激增，实例重启时间从秒级恶化至数十分钟。备份任务因需遍历庞大元数据而严重超时，甚至直接中

断，无法完成全量备份。废弃按设备分表，将数据合并至单一的时间序列集合（Time Series Collection），通过 deviceId + timestamp 建立复合索引。优化后集合数降至个位数，启动与备份均恢复正常。

数据建模检查清单

为确保规范落地，建议在项目上线评审阶段逐项对照以下清单进行检查：

检查项	验证手段	通过标准
1. 命名规范统一	Review 所有涉及的库/集合/字段命名代码	完全符合本文档的命名及前缀规则
2. 文档规模可控	抽样执行 Object.bsonsize(doc)	核心文档大小建议控制在1MB以内
3. 嵌套层级合理	审查核心文档的 JSON 结构树	最大嵌套深度 ⩽ 3-5 层
4. 规避无限数组	深入审查数据写入与追加逻辑	数组具备明确的业务上限，或已采用分桶模式/$slice 截断机制
5. 数据类型严谨	Review 实体类的字段类型定义	日期建议用 Date，金额建议用 Decimal128
6. 开启 Schema 验证	检查建表脚本或 validator 配置	核心集合的必填项、字段类型、格式校验已全部配置约束
7. 单个实例总集合数量建议不超过5000	预估并执行 show collections 统计	单库内业务集合数量预估/实际值 ⩽ 100 个

索引设计与优化

最近更新时间：2026-05-08 10:04:01

操作场景

在 MongoDB 中，合理设计索引是保障查询性能与系统稳定性的关键。在实际生产环境中，由于索引使用不当，往往会引发严重的可用性问题，例如：

- 查询性能骤降：核心查询未命中索引导致全表扫描，响应时间从毫秒级上升至秒级，引发接口超时。
- 内存溢出报错：千万级数据集无索引排序，突破了 MongoDB 默认的内存限制（4.4之前版本为32MB，4.4+版本为100MB），导致查询直接中断。
- 阻塞读写请求：在业务高峰期采用前台（Foreground）方式创建大表索引，导致集合被锁，业务读写被迫中断。
- 写入延迟激增：单个集合存在大量冗余索引（如20个以上），导致每次写入操作均需同步维护多个索引树，严重消耗计算资源。

本文档旨在：提供标准的索引设计与运维规范，确保核心查询精准命中索引，并保障线上环境的索引变更操作安全可控。

索引设计基本准则

准则一：ESR 准则（复合索引字段排序）

- **核心动作**：构建复合索引时，务必按照 Equality（等值）、Sort（排序）、Range（范围）的顺序排列字段。该原则基于 B-Tree 索引的遍历机制，旨在最大化索引利用率并消除内存排序。

顺序	类型	说明	查询示例
E	Equality（等值）	索引最左位置，精确匹配可快速缩小扫描范围	`{ status: "paid" }`
S	Sort（排序）	紧跟 E 之后，利用索引有序性，避免内存排序	`.sort({ createTime: -1 })`
R	Range（范围）	索引最右位置，范围扫描后索引有序性中断，后续字段退化为逐条过滤	`{ createTime: { $gte: ISODate("2024-01-01") } }`

- **业务应用**：以订单查询接口为例，需求为"查找特定用户的已支付订单，按金额倒序，且只需最近一个月的数据"。
  - **问题表现**：索引设计为 {userId: 1, status: 1, createTime: 1, amount: -1}，将 Range（createTime）放在了 Sort（amount）之前。索引在时间字段上发生发散，导致后续的金额排序无法

走索引，触发内存排序，查询延迟达秒级且存在 OOM 风险。

- **优化操作**：调整为 {userId: 1, status: 1, amount: -1, createTime: 1}，严格遵循 ESR 顺序。等值字段精准过滤后，直接利用索引的天然顺序输出排序结果，最后通过时间字段进行边界截断。查询延迟从秒级降至 10ms，性能提升100倍以上。

- **ESR 索引设计示例**：以下示例对比 ESR 正确顺序与错误顺序的索引设计，演示 Range 字段位置对排序性能的影响。

```
// 业务查询语句
db.t_orders.find({
    userId: 10086,                                  // E: 等值
    status: "paid",                                 // E: 等值
    createTime: { $gte: ISODate("2024-02-01") }   // R: 范围
}).sort({ amount: -1 })                             // S: 排序

// 错误：Range 在 Sort 之前，索引在 createTime 发散后无法支撑 amount 排序，触发内存排序
db.t_orders.createIndex(
    { userId: 1, status: 1, createTime: 1, amount: -1 },
    { background: true }
)

// 正确：ESR 顺序，等值过滤 → 索引排序 → 范围截断
db.t_orders.createIndex(
    { userId: 1, status: 1, amount: -1, createTime: 1 },
    { background: true }
)
```

准则二：最左匹配准则

- **核心动作**：查询条件必须显式包含复合索引的最左侧字段，否则索引无效。若业务场景中存在跳过最左字段的独立查询需求，必须为该字段单独构建索引。

  复合索引的 B-Tree 结构具有严格的字段先后依赖关系：先按第一字段进行全局排序；在第一字段相同的区间内，再按第二字段进行局部排序。若查询缺失了“最左字段”这一全局搜索基准，数据库便无法利用索引树进行路径导航，查询将被迫退化为全表扫描（COLLSCAN）。
  假设集合已建立复合索引 { a: 1, b: 1, c: 1 }，不同查询条件的索引利用路径如下表所示：

查询条件	索引利用率	执行说明
{ a: 1 }	完全利用	命中最左前缀 a。

{ a: 1, b: 1 }	完全利用	命中最左前缀 a, b。
{ a: 1, b: 1, c: 1 }	完全利用	完整匹配全部索引字段 a, b, c。
{ a: 1, c: 1 }	部分利用	仅能利用最左字段 a 进行索引分支定位，因缺失中间字段 b，导致 c 字段的索引匹配被阻断。
{ b: 1 }	无法使用	缺失最左字段 a，失去搜索基准，触发全表扫描。
{ b: 1, c: 1 }	无法使用	缺失最左字段 a，触发全表扫描。

- **业务应用**：以某业务系统为例，原集合已建立复合索引 {userId: 1, status: 1, createTime: 1}。
  - **问题表现**：运营后台发起独立查询 db.orders.find({status: "pending"})，因缺失最左前缀 userId，复合索引完全无法使用，触发全表扫描。高峰期接口超时率飙升至30%。
  - **优化操作**：识别到跳过最左字段的查询需求后，为 status 字段单独建立单键索引 {status: 1}，恢复系统可用性。
- **最左匹配索引示例**：以下示例演示缺失最左字段导致索引失效的场景及对应解决方案。

```
 // 已有复合索引
db.t_orders.createIndex(
    { userId: 1, status: 1, createTime: 1 },
    { background: true }
)

// 正确：包含最左字段 userId，索引完全命中
db.t_orders.find({ userId: 10086, status: "paid" })

// 部分命中：包含最左字段 userId，但跳过 status，仅利用 userId 定位
db.t_orders.find({ userId: 10086, createTime: { $gte: ISODate("2024-
01-01") } })

// 错误：缺失最左字段 userId，触发全表扫描
db.t_orders.find({ status: "pending" })

// 解决方案：为独立查询字段单独建立索引
db.t_orders.createIndex(
    { status: 1 },
    { background: true }
```

```
)
```

法则三：高基数字段前置准则

- **核心动作**：在构建包含多个等值匹配（Equality）条件的复合索引时，务必将区分度高（高基数）的字段排在最前面，区分度低（低基数）的字段排在后面。
  - 将**高基数（区分度高）**字段前置，可利用其显著的“选择性”在查询初期迅速缩小搜索范围，确保数据库在 B+ 树索引搜索阶段排除绝大多数无关记录，减少磁盘 I/O 吞吐和数据页扫描量。
  - 根据**最左匹配原则**，高基数前置的索引可同时支撑“组合查询”与“单字段前缀查询”，从而实现索引的高效复用。通过这种策略可以减少冗余索引的建立，在保证查询速度的同时，有效降低存储空间占用及频繁写操作（增删改）带来的索引维护开销。

区分度级别	数据特征	字段示例	索引排位策略
高（High）	具有唯一性或极少重复	用户 ID、订单号	应优先作为复合索引的最前置字段。
中（Medium）	存在一定重复，但分类较多	城市、商品分类	有效。通常排在复合索引的中段，需联合使用。
低（Low）	值域极小，海量重复	性别、布尔值	差。若必须纳入复合索引，应严格置于最末端。

- **业务应用**：高基数前置的核心价值是高基数字段作为索引前缀时，能被更多查询模式复用，用更少的索引覆盖更多场景。

  以某订单系统为例，原集合已建立复合索引 `{userId, status}`：随着业务的需求变化，产生以下两种查询需求：
  - 场景 A： `db.orders.find({ userId: "U123" })`

    效果： 直接命中复合索引的最左前缀，由于 userId 区分度高，检索性能很好。
  - 场景 B： `db.orders.find({ status: "completed" })`

    分析： 该查询虽无法利用现有索引，但是由于 status 字段本身区分度很低，即使通过 `{status, userId}` 索引匹配最左边的 status 字段，其性能也会很差。

索引创建规范

规范一：后台建索引原则（Background Indexing）

- **核心动作**：在生产环境中执行创建索引操作时，必须显式指定 background: true 参数（注：针对 MongoDB 4.2 之前的版本）。

  MongoDB 在执行默认的前台（Foreground）建索引操作时，会获取集合甚至数据库级别的排他锁（Exclusive Lock）。在此构建期间，目标集合上的所有读写请求均会被完全阻塞。通过启用 background:

true 模式，数据库会在后台交替处理建索引任务与业务读写请求，从而消除全局锁阻塞等待，保障系统的高可用性。

> **说明：**
> 自 MongoDB 4.2 起，所有索引构建均默认采用优化的非阻塞机制，background 参数虽被弃用但仍可被解析兼容。

- **业务应用**：某电商 DBA 在白天业务期对千万级订单表执行单键索引创建。
  - **问题表现**：执行 db.t_orders.createIndex({ customerId: 1 }) 时未指定 background 参数。MongoDB 4.0默认前台建索引，获取排他锁导致整个订单库的读写操作全部阻塞，持续40分钟，直接引发大促期间严重故障。
  - **优化操作**：使用 background: true 后台模式建索引，构建期间业务正常运行，仅有轻微性能下降。多个索引可合并为一次批量构建操作，进一步降低 I/O 损耗。
- **后台建索引示例**：以下示例演示前台建索引的阻塞风险及后台模式的正确用法。

```
// 错误：未指定 background，可能阻塞所有操作
db.t_orders.createIndex({ customerId: 1 })

// 单个创建：开启 background 模式，避免阻塞业务
db.t_orders.createIndex(
    { customerId: 1, createTime: -1 },
    { background: true }
)
```

规范二：控制索引数量

- **核心动作**：限制单个集合的索引规模：常规建议数量 ⩽ 10 个，绝对红线不超过20个。
  当集合中存在过多索引时，系统将面临以下维度的性能衰退：

资源维度	性能损耗说明
写入延迟（Write I/O）	每次执行 Insert、Update 或 Delete 时，数据库必须同步遍历并更新所有相关索引的 B-Tree 结构。索引越多，单次写入的 I/O 开销呈线性增长。
存储占用（Disk）	每一个索引均是一棵独立的数据树，会占用大量的物理磁盘空间。

- **业务应用**：某电商订单表为了满足各类查询，累积创建了高达25个索引。
  - **问题表现**：每产生1笔新订单，数据库需同步更新25棵索引树，导致该表的索引总容量（120GB）竟然反超了数据本身（50GB），单条订单的写入延迟从5ms持续上升至50ms。

- **优化动作**：经排查，直接删除了15个调用次数为0的历史报表索引和前缀重复索引。写入延迟瞬间回落至5ms的正常水平，同时直接释放了约70GB的磁盘与内存空间。

- **确认索引数量示例**：以下示例演示如何排查冗余索引并进行清理。

```
// 第一步：查看集合当前所有索引
db.t_orders.getIndexes()

// 第二步：通过 $indexStats 排查每个索引的实际使用情况
db.t_orders.aggregate([{ $indexStats: {} }]).forEach(function(idx) {
    print(
        "索引名称:", idx.name,
        "| 调用次数:", idx.accesses.ops,
        "| 最近使用:", idx.accesses.since
    )
})

// 第三步：查看索引与数据的存储占比
var stats = db.t_orders.stats()
print("数据大小:", (stats.size / 1024 / 1024).toFixed(2), "MB")
print("索引大小:", (stats.totalIndexSize / 1024 / 1024).toFixed(2),
"MB")
print("索引/数据比:", (stats.totalIndexSize / stats.size *
100).toFixed(1), "%")

// 第四步：删除调用次数为 0 的冗余索引
// 操作前务必确认该索引确实无业务依赖
db.t_orders.dropIndex("idx_legacy_report_1")
db.t_orders.dropIndex("idx_legacy_report_2")

// 注意：禁止删除 _id 默认索引
// db.t_orders.dropIndex("_id")  // 此操作会报错，_id 索引不可删除
```

规范三：冗余索引的安全清理

- **核心动作**：建立定期巡检机制，使用 `$indexStats` 管道操作符识别零调用的未使用的索引，并结合隐藏索引（Hidden Index）特性执行安全下线。

  $indexStats 可以精准捕获索引的真实调用频次，再结合 MongoDB 4.4+引入的 hideIndex 特性，可以对目标索引进行“逻辑屏蔽”（优化器不再使用，但后台仍维护更新），在确认无业务影响后再执行“物理删除”，从而实现零风险的资源回收。

- **业务应用**：某物流平台的订单集合 t_orders 运行2年后累积了23个索引，其中多个是早期需求迭代遗留的废弃索引和功能重叠的冗余索引。每次写入操作需同步更新所有索引，导致写入延迟从初期的5ms逐步上升至15ms，且索引占用存储空间已超过数据本身。
  - **问题表现**：通过 $indexStats 聚合分析发现，23个索引中有14个近90天访问次数为0，属于完全无用索引；另有多组索引存在前缀重叠（如 {userId: 1} 与 {userId: 1, status: 1} 并存），低选择性索引可被复合索引覆盖。
  - **优化操作**：对14个零访问索引逐一执行 hideIndex 隐藏，观察7天确认无业务影响后物理删除；合并前缀重叠索引，最终索引数从23个缩减至9个。清理后写入延迟从15ms降至5ms（提升约30%），索引存储空间从18GB降至7GB（节省约40%）。
- **安全清理代码示例**：以下示例演示冗余索引的完整清理流程——先通过 $indexStats 识别零访问索引，再用 hideIndex 逻辑隐藏并观察，确认无业务影响后再物理删除。

```
// 第一步：执行统计聚合，识别无用索引
db.t_orders.aggregate([{ $indexStats: {} }])
/* 输出示例分析：
{
    "name": "idx_old_field",
    "accesses": {
        "ops": NumberLong(0),     // 关键指标：调用次数为 0
        "since": ISODate("2024-01-01T00:00:00Z")
    }
}
*/

// 第二步：逻辑隐藏（支持快速回滚，MongoDB 4.4+）
db.t_orders.hideIndex("idx_old_field")

// 观察期：持续监控 1-7 天，若业务受损可立即执行
db.t_orders.unhideIndex("idx_old_field") 恢复。

// 第三步：物理删除（确认无损后执行）
db.t_orders.dropIndex("idx_old_field")
```

特殊索引使用规范

规范四：TTL 索引自动清理过期数据

- **核心动作**：对于日志、会话等有过期时间的数据，建议使用 TTL (Time-To-Live) 索引实现自动过期清理，避免通过应用层定时脚本执行手动的海量批量删除。

在使用 TTL 索引时，需严格遵循以下机制与限制：

特性 / 限制	规范说明
单键约束	TTL 索引必须是单键索引（Single Field Index）。若 TTL 字段被放入复合索引中，TTL 自动回收特性将直接失效。
数据类型	目标字段的数据类型必须是 Date 类型（或包含 Date 类型的数组），否则引擎不会执行清理。
清理延迟	并非到期精确秒级删除。默认情况下，TTLMonitor 线程每60秒轮询一次，因此数据的物理删除存在分钟级的延迟。
引擎参数控制	线上若存在性能波动，DBA 可通过动态调整 ttlMonitorSleepSecs（控制轮询休眠间隔）和 ttlDeleteBatch（控制单批次删除量）来进一步平抑 I/O 资源消耗。具体操作，请参见 参数配置 。

- **业务应用**：某日志中台每天凌晨2点通过定时任务，下发 db.logs.remove({createTime: {$lt: 某时间戳}}) 命令，集中删除30天前的数亿条历史日志。集中式的海量删除引发严重的锁等待，数据库 CPU 瞬间飙升至100%。同时，产生的巨量 Oplog 导致集群主从延迟高达数分钟，严重影响了依赖该集群的其他在线业务。改用 TTL 索引后，MongoDB 后台任务自动、平滑地淘汰过期数据，对业务完全无感知。
- **代码示例**：根据业务灵活度，通常有两种 TTL 索引设计模式。固定时长淘汰适用于统一过期策略的场景，精准时间戳淘汰适用于需要按文档粒度动态控制存活时间的场景。

```
// 模式一：固定时长淘汰（例如：按 createTime 字段，30 天后过期）
db.t_sessions.createIndex(
    { createTime: 1 },
    { expireAfterSeconds: 2592000, background: true }
)

// 模式二：精准时间戳淘汰（推荐，由业务层动态决定存活时间）
// 将 expireAfterSeconds 设为 0，引擎将在 expireAt 时间点到达时进行清理
db.t_sessions.createIndex(
    { expireAt: 1 },
    { expireAfterSeconds: 0, background: true }
)

// 业务层插入时，直接指定具体的死亡时间
db.t_sessions.insertOne({
    sessionId: "sess_001",
    userId: "user_001",
    expireAt: new Date(Date.now() + 3600000)  // 动态指定 1 小时后过期
```

```
})
```

规范五：时序索引高效存储与查询时间序列数据

- **核心动作**：对于 IoT 传感器、系统监控、日志采集等按时间持续写入的数据，建议使用 MongoDB 5.0+提供的时序集合（Time Series Collection）替代普通集合，通过自动按时间分桶和桶内字段列式聚合获得更高压缩率与范围查询效率，避免在普通集合上手动维护时间戳索引和数据归档逻辑。

  在使用时序集合时，需严格遵循以下机制与限制：

特性 / 限制	规范说明
必选参数 `timeField`	创建时序集合时必须指定 `timeField`，该字段的值必须是 Date 类型，用于标识每条测量值的时间戳。MongoDB 据此自动分桶存储。
推荐参数 `metaField`	`metaField` 用于标识数据源（如设备 ID、传感器编号），MongoDB 据此对数据进行分区。建议选择**很少变化**的标识符，避免使用数组类型。
粒度设置 `granularity`	建议根据同一数据源相邻测量值的时间间隔选择粒度：`seconds`（桶跨度1小时）、`minutes`（桶跨度24小时）、`hours`（桶跨度30天）。粒度过粗导致单桶数据量过大查询变慢，粒度过细导致桶数量激增浪费存储。
自动索引	MongoDB 6.0+自动在 `timeField` 和 `metaField` 上创建二级索引，低版本需手动创建。
不支持的操作	时序集合不支持 `distinct()` 高效执行（建议用 `$group` 聚合替代）；8.0以下版本不支持区域分片（Zone Sharding）；`metaField` 一旦定义不可更换为其他字段。

- **业务应用**：某 IoT 平台每秒接收数万条传感器数据，使用普通集合存储后，3个月数据量突破10亿条。按时间范围查询最近24小时的温度趋势需要扫描数千万条文档，P99延迟超过5s，且存储空间持续膨胀。迁移至时序集合后，MongoDB 自动按 `metaField`（设备 ID）分区、按 `timeField` 分桶压缩，存储空间减少约70%，相同查询延迟降至200ms以内，同时无需额外维护数据归档脚本。
- **代码示例**：根据数据采集频率选择合适的粒度参数，是时序集合性能优化的关键。

```
// 推荐：创建时序集合（传感器每 5 分钟上报一次）
db.createCollection("t_sensor_data", {
    timeseries: {
        timeField: "timestamp",          // 必选：时间戳字段
        metaField: "sensorId",           // 推荐：数据源标识
        granularity: "minutes"           // 粒度匹配采集频率（5 分钟 → minutes）
    }
```

```
})

// 推荐：创建时序集合（系统监控每秒采集）
db.createCollection("t_metrics", {
    timeseries: {
        timeField: "ts",
        metaField: "metadata",          // metadata 可以是对象，如 {
host: "web01", region: "bj" }
        granularity: "seconds"          // 粒度匹配采集频率（秒级 →
seconds）
    }
})

// 插入时序数据
db.t_sensor_data.insertMany([
    { sensorId: "sensor_001", timestamp: new Date(), temperature:
23.5, humidity: 65 },
    { sensorId: "sensor_001", timestamp: new Date(), temperature:
23.6, humidity: 64 },
    { sensorId: "sensor_002", timestamp: new Date(), temperature:
18.2, humidity: 72 }
], { ordered: false })  // ordered: false 提升批量写入性能

// 错误：在普通集合上手动管理时序数据
db.createCollection("t_sensor_data_old")
db.t_sensor_data_old.createIndex({ sensorId: 1, timestamp: -1 })
// 需要自行处理数据归档、压缩、清理，维护成本高

// 查询最近 24 小时某设备的数据（自动利用时序索引）
db.t_sensor_data.find({
    sensorId: "sensor_001",
    timestamp: { $gte: new Date(Date.now() - 86400000) }
}).sort({ timestamp: -1 })

// 用 $group 聚合替代 distinct()（时序集合不支持高效 distinct）
db.t_sensor_data.aggregate([
    { $match: { timestamp: { $gte: new Date(Date.now() - 86400000) } }
},
    { $group: { _id: "$sensorId" } }
```

```
])
```

排序与内存限制

规范六：排序字段建议加入索引

- **核心动作**：所有排序操作的字段必须包含在索引中，禁止内存排序。

  MongoDB 执行排序有两种方式——索引排序和内存排序。当排序字段已被索引覆盖时，数据库直接按索引的有序结构返回结果，无需额外排序计算；当排序字段未建索引时，数据库必须将所有匹配文档加载到内存中排序（即 SORT 阶段），消耗大量内存资源。
  - 内存限制：内存排序的数据量超过版本限制（4.2及以前为32MB，4.4+为100MB）时，查询直接报错失败。
  - 索引设计：在复合查询场景中，排序字段应与查询条件字段组合为复合索引，且排序字段的方向（升序/降序）必须与索引定义一致，从而确保数据库利用索引同时完成过滤和排序。

MongoDB 版本	默认内存限制	超限后果
4.2及以前	32MB	报错，查询失败
4.4+	100MB	报错，查询失败

- **业务应用**：某电商平台运营后台的"按成交额倒序"报表，底层查询为 `db.t_orders.find({ status: "paid" }).sort({ amount: -1 })`，涉及千万级订单数据。
  - **问题表现**：订单量突破800万条后，排序数据量超过32MB（MongoDB 4.2限制），查询报错 `Sort operation used more than the maximum 33554432 bytes of RAM`，报表功能不可用，影响运营决策2天。
  - **优化操作**：通过 `explain("executionStats")` 确认执行计划存在 SORT 阶段，排序字段 amount 缺少索引，触发内存排序。建立 `{ status: 1, amount: -1 }` 复合索引，status 用于过滤、amount 利用索引有序性完成排序，SORT 阶段消除，响应时间从超时降至200ms。
- **代码示例**：以下示例演示如何为排序字段建立复合索引，并通过 explain("executionStats") 验证执行计划中是否消除了 SORT 阶段。

```
// 错误：排序字段未建索引，可能触发内存排序
db.t_orders.find({ status: "paid" }).sort({ createTime: -1 })

// 正确：排序字段包含在索引中
db.t_orders.createIndex({ status: 1, createTime: -1 }, { background:
true })
db.t_orders.find({ status: "paid" }).sort({ createTime: -1 })  // 索引
排序，无内存限制
```

```
// 验证是否使用索引排序
db.t_orders.find({ status: "paid" }).sort({ createTime: -1
}).explain("executionStats")
// 检查是否有 SORT 阶段，无 SORT 表示使用了索引排序
```

避免低效操作符

规范七：避免使用无法利用索引的操作符

- **核心动作**：避免使用 `$ne` 、 `$nin` 、无前缀 `$regex` 、 `$where` 等低效操作符。
  - **否定操作符的代价**：$ne 和 $nin 属于否定条件，数据库无法通过索引直接定位目标文档，而是需要扫描索引中所有不匹配的条目再逐一排除，本质上退化为全索引扫描甚至全表扫描。
  - **正则与脚本的代价**：无前缀锚定的 $regex 无法利用索引的有序性进行范围查找，数据量越大性能衰减越严重。
  - **优化原则**：将否定条件转换为正向条件（$ne → $in），将模糊匹配改为前缀锚定，将 $where 改写为标准查询操作符。

低效操作符及优化方案：下表列出常见的无法有效利用索引的操作符及其对应的优化替代方案。

操作符	问题	优化方案
`$ne`	需扫描所有非匹配值	改用 `$in` 列出有效值
`$nin`	需扫描所有不在列表中的值	改用 `$in` 正向匹配
`$not`	通常无法利用索引	改用正向条件
`$regex`（无前缀）	前缀无锚定无法利用索引	使用前缀锚定 `/^prefix/`
`$where`	JavaScript 执行，极慢	改用标准查询操作符
`$exists: false`	需扫描所有文档	使用稀疏索引或重新设计

- **业务应用**：某会员管理系统查询"非 VIP 用户"列表，底层查询为 db.t_users.find({ level: { $ne: "VIP" } })，用户总量约500万。
  - **问题表现**：$ne 操作符无法有效利用 level 字段索引，每次查询触发全表扫描，数据库 CPU 持续高位运行，P99延迟超过2s，页面加载缓慢。
  - **优化操作**：将否定条件改为正向枚举 { level: { $in: ["普通", "银卡", "金卡"] } }，查询直接命中索引，P99延迟从2s降至40ms。
- **代码示例**：以下示例对比否定操作符和正向操作符的写法差异，以及正则表达式的前缀锚定优化。

```
// 低效：$ne 无法有效利用索引
db.t_orders.find({ status: { $ne: "cancelled" } })

// 优化：$in 列出所有有效状态
db.t_orders.find({ status: { $in: ["pending", "paid", "shipped",
"completed"] } })

// 低效：无前缀正则，全表扫描
db.t_users.find({ name: /张/ })

// 优化：前缀锚定正则，可利用索引
db.t_users.find({ name: /^张/ })
```

索引设计检查清单

上线前必查

检查项	验证方法	通过标准
所有查询都命中索引	`explain("executionStats")`	`stage` 为 `IXSCAN`，无 `COLLSCAN`
无内存排序	`explain("executionStats")`	无 `SORT` 阶段
扫描效率合理	`totalDocsExamined / nReturned`	比值接近1
复合索引遵循 ESR	审查索引字段顺序	Equality → Sort → Range
索引数量可控	`db.collection.getIndexes()`	≤ 10个
建索引使用 background	审查建索引命令	包含 `background: true`

定期检查

检查项	验证方法	操作建议
无用索引清理	`$indexStats`	`ops: 0` 的索引评估后删除
索引使用率	`$indexStats`	低使用率索引考虑删除

索引大小	`db.collection.stats().indexSizes`	异常大的索引需要分析

索引优化建议

当集合的索引策略难以人工判断时，建议通过**数据库智能管家（DBbrain）**的**索引推荐**功能进行辅助决策。DBbrain 基于实际慢查询日志和查询模式，自动分析并推荐最优索引方案，帮助识别缺失索引和冗余索引。具体操作，请参见 索引推荐 。

常见问题

Q1：索引建好了，查询还是慢?

1. 确认查询是否命中索引。

```
db.t_orders.find({ status: "paid" }).explain("executionStats")
```

2. 检查扫描效率：通过 explain() 方法分析查询的执行计划，在返回结果中检查索引命中的效率。

```
// 关键指标，接近 1：索引高效，扫描即命中
totalDocsExamined / nReturned ≈ 1    // 理想值
```

3. 检查是否存在内存排序：在 explain("executionStats") 的返回结果中，沿着 executionStages 逐层查看 stage 字段：

```
// 在 explain 结果中定位排序阶段
executionStats.executionStages.stage
// 或嵌套在 inputStage / inputStages 中
```

stage 值	含义	是否需要优化
SORT	内存排序，未利用索引有序性	需优化
SORT_KEY_GENERATOR	正在提取排序键，配合 SORT 出现	需优化
无 SORT 阶段	排序由索引天然有序性完成	无需优化

4. 检查索引字段顺序是否符合 ESR 原则。

```
// 不符合 ESR：范围字段在前，排序字段在后
db.t_orders.createIndex({ createTime: 1, status: 1 })
```

```
db.t_orders.find({ createTime: { $gte: ISODate("2024-01-01") },
status: "paid" }).sort({ amount: -1 })

// 符合 ESR：等值 → 排序 → 范围
db.t_orders.createIndex({ status: 1, amount: -1, createTime: 1 })
```

Q2：复合索引和多个单字段索引，哪个好?

绝大多数场景下，复合索引优于多个单字段索引。

对比维度	多个单字段索引	复合索引
索引方案	{ status: 1 } + { userId: 1 } + { createTime: 1 }	{ status: 1, userId: 1, createTime: −1 }
查询过程	查询一般只有一个字段走索引	单次 B−Tree 查找直接定位
排序	索引无法覆盖排序，可能触发内存 SORT	索引天然有序，无需内存排序
覆盖查询	无法实现，必须回表取完整文档	若查询字段全部包含在索引中，可直接返回结果，无需回表
内存开销	整体内存开销更大	无额外开销

```
// 多个单字段索引：MongoDB 尝试索引交集，效率不稳定
db.t_orders.createIndex({ status: 1 })
db.t_orders.createIndex({ userId: 1 })
db.t_orders.createIndex({ createTime: 1 })

// 复合索引：一个索引覆盖查询 + 排序，遵循 ESR 原则
db.t_orders.createIndex({ status: 1, userId: 1, createTime: -1 })

// 当查询模式不固定、字段组合多变时，单字段索引更灵活：
// 场景：运营后台的动态筛选，用户可能按任意字段组合查询，此时为每种组合建复合索引不
现实，单字段索引 + 索引交集是合适的选择
db.t_orders.find({ status: "paid" })                    // 只按状态
db.t_orders.find({ userId: "u_10001" })                  // 只按用户
```

```
db.t_orders.find({ createTime: { $gte: ISODate("...") } }) // 只按时间
db.t_orders.find({ status: "paid", userId: "u_10001" })  // 状态 + 用户
```

Q3：大集合如何安全建索引?

1. 选择低峰期：在业务低峰期执行索引构建，降低对线上读写的影响。
2. 分片集群关闭 Balancer：MongoDB 4.4以下版本需在建索引前关闭 Balancer，避免数据迁移与索引构建并发冲突。

```
sh.stopBalancer()
```

3. 执行后台建索引：添加 { background: true } 选项（仅适用于 MongoDB 4.2及以下版本，4.2+版本索引构建默认不阻塞读写操作）。

```
db.orders.createIndex({ userId: 1, createTime: -1 }, { background:
true })
```

4. 监控索引构建进度与资源：持续观察 CPU、内存、磁盘 IO 使用情况，必要时终止构建。

> ⚠ **注意：**
> 严禁在索引构建期间删除同集合索引：MongoDB 4.4以下版本采用串行索引构建机制，若从节点尚未完成索引构建，此时对同一集合触发删除索引操作（如 dropIndex），可能导致从节点复制中断甚至不可用。建索引期间应冻结该集合的所有索引变更操作。等待 currentOp 返回为空，确认构建完成后再操作。

```
// 查看当前正在执行的索引构建任务
db.currentOp({ "command.createIndexes": { $exists: true } })
```

5. 预估时间：大集合索引构建可能需要数小时，提前评估并预留维护窗口。

通用开发规范

最近更新时间：2026-04-17 08:59:02

数据库设计规范

禁止类

禁止数据库中创建过多的表

在 WiredTiger 引擎中，每个集合都需要创建多个文件来保存元数据、数据及索引，磁盘上过多的小文件会导致性能下降。建议单个数据库表个数控制在100个以内，整个数据库实例表数量控制在2000个以内。

建议类

- 建议数据库名以 db 开头，不能包含除 _ 以外的特殊字符，所有字母全部小写，数据库名不超过64个字符。
- 建议集合名以 t_开头，不能包含除 _ 以外的特殊字符、集合名不超过120个字符。

索引设计规范

禁止类

- **禁止线上库不带 background 参数建索引**

  MongoDB 4.2之前的版本，createIndex() 命令默认是 foreground 模式，这种模式下创建索引会阻塞数据库的所有操作，造成业务中断，线上业务执行 createIndex() 务必添加 background 参数。

  > ⚠ **注意：**
  > background 需要在 createIndex() 命令的 options 参数中，示例：`db.test.createIndex({a: 1}, {unique:true, background: true})`，切勿将不同的参数分开，示例：`db.test.createIndex({a: 1}, {unique:true}, {background: true})` 。

- **排序字段需要放到索引中，避免业务大量在内存中排序造成数据库 OOM（Out Of Memory）**
  - MongoDB 4.2及之前的版本，一条 sql 默认只允许使用32MB内存进行排序，如果超出会提示 `Sort operation used more than the maximum 33554432 bytes of RAM` 错误，此时可以通过执行 `db.runCommand({ getParameter : 1, "internalQueryExecMaxBlockingSortBytes" : 1 })` 调整排序内存。
    但是线上业务禁止调整，因为这样会让数据库 OOM 的概率增大。建议将排序字段加到索引中，通过索引排序实现排序能力。
  - MongoDB 4.4版本，虽然提供了磁盘排序的选项以避免排序消耗大量内存，但是建议最好使用索引排序。
- **禁止在索引同步过程中触发同命名空间的索引删除操作**

  在 MongoDB 4.4 以下版本中，索引创建是异步的。从节点通过回放 oplog 同步索引时，若对其所在集合执行 dropIndexes 操作，将引发锁冲突。此冲突会阻塞 oplog 回放线程，导致后续操作（包括新连接）排队等

待。持续的阻塞将迅速耗尽连接资源，最终使从节点无法响应新请求，服务不可用。
  - 临时方案：索引创建后，需确认所有分片的主从节点均构建完成，方可进行后续操作。
  - 根本方案：升级至 MongoDB 4.4及以上版本，其并发索引构建机制可彻底规避此问题。
- **禁止一个表内创建过多的索引**

  MongoDB 插入每条数据的时候同时需要写索引。索引越多，写入数据时就要花费更多的代价。因此禁止对索引的滥用，如对每个字段都建立索引，哪怕不会根据该字段进行查询。建议单表索引最多不超过10个。

建议类

- **建议定期清理无用索引**

  索引在写操作时会带来额外的资源消耗，因此需要尽量精简索引。
  MongoDB 4.4之后的版本，建议使用 hidden index 先隐藏无用的索引，隐藏后业务确认正常再删除索引。
- **按照最左匹配原则，如果单列索引已经被复合索引包含，建议删除**

  额外的索引会造成写操作时性能浪费。
- **建议尽量避免使用 $ne/$nin 等操作**
  和其他数据库（如 MySQL）一样，不等于 not in 类的操作无法有效利用索引，尽量应该避免。
- **建议在区分度较大的字段上建立索引**

  如果索引字段区分度较小，查询扫描的行数依然会比较多，查询效率较低，对数据库负载影响较大。因此建立索引的字段尽量应该有较大的区分度。
- **索引构建期间保持对各分片实例状态的监控，发现重启事件后立即检查并补齐索引**

  MongoDB 4.4 以下版本缺乏索引构建的持久化和跨节点协调机制，各分片独立执行索引构建，进度状态仅保存在本地内存中。若主节点在构建过程中重启，内存状态丢失且无法自动恢复续建，导致索引处于不完整状态，需手动在该分片上重新执行索引构建。
- **大集合创建索引构建期间建议关闭 Balancer，等加完再打开 Balancer**

  MongoDB 4.4 以下版本中，索引构建的批量键插入阶段会持有意向排他锁（IX 锁），而 chunk 迁移（moveChunk）内部操作同样需要获取相关锁资源。两者在并发执行时产生锁冲突，索引构建会阻塞 chunk 迁移进度，chunk 迁移也会延缓索引构建，导致分片集群在索引维护期间出现迁移停滞、请求延迟增加等问题。

数据库操作规范

禁止类

- **禁止上线未经过 explain() 确认执行计划的 sql**
  上线 sql 前需要 explain() 确认执行计划是否符合预期，否则上线可能会引起故障。
- **线上环境禁止关闭鉴权，特别是开放外网访问的数据库**
  关闭鉴权会将数据库暴露给所有人，特别是数据库服务器开通了外网。建议线上环境打开鉴权。如果一定要关闭鉴权，务必设置防火墙规则或 IP 白名单。
- **禁止在 admin 库、local 中存储业务数据**

- admin 库读写时会加 db 锁，影响性能；local 库只会保存到本地，不会复制到从节点，如果发生主从切换会丢失数据。因此禁止使用 admin 库和 local 库。
- **分片集群禁止执行 db.dropDatabase() 命令后再创建同名的 db**
  - MongoDB 4.0及之前的版本，官方文件要求删除 db 并创建同名 db 后，业务读写数据前需要在所有 mongos 节点上执行重启或 flushRouterConfig 命令。
  - MongoDB 4.2版本，官方文件要求删除 db 命令执行完以后，再执行一次删除 db 命令，并在所有 mongos 节点上执行重启或 flushRouterConfig 命令。MongoDB 4.4版本，官方文件要求删除 db 命令执行完以后，再执行一次删除 db 命令，方可重建同名的 db。
  - MongoDB 5.0及以上版本，执行一次删除 db 命令即可。

  因此禁止在业务代码中直接进行 db.dropDatabase() 命令后再创建同名的 db。运维人员在做该操作时，请务必按照官方文档要求进行所有必要的操作。
- **高并发高性能场景，禁止过度使用 in 和 or**
  in 或者 or 条件语句在数据库底层需要转换成多次查询，过多的 in 和 or 操作在高并发高性能场景，会严重影响请求的响应时延及数据库负载。
- **高并发高性能场景，禁止将复杂的运算操作交给数据库进行**
  MongoDB 提供了强大的计算能力（如 MapReduce 等），这些特性对开发人员非常友好，极大减轻了业务逻辑。但是这些运算不可避免是需要资源的，如果将复杂运算下沉到数据库层，高并发场景势必会给数据库造成极大的负担，数据库一旦故障会造成整个系统雪崩。
  建议在高并发高性能的场景下，数据库操作保持简单，复杂的运算交给服务器并适当在数据库前端增加缓存。
- **线上业务禁止直接进行批量数据 remove**
  remove 命令到数据库后会先查询符合删除条件记录的 _id，之后一条条按照 _id 进行删除，并记录到 oplog 中（删除每条记录都会写一条 oplog）。
  当满足 remove 条件的数据较多时对数据库压力较大，且极容易引起主从延迟突然增大。
  线上业务建议直接用 drop 集合或用脚本一条条删除并控制删除速度。或尽量使用 ttl 索引。
- **业务禁止自定义 _id 字段**
  _id 是 MongoDB 内部的默认主键，默认这是一个自增的序列。如果自定义 _id 并且业务无法保证 _id 递增，每次插入数据后，_id 索引不可避免需要对 B 树索引进行调整，这将对数据库带来额外的负担。
- **副本集直连 mongod 节点的场景，禁止只在连接串中配置单个 IP；分片集群禁止只连接单个 mongos 地址**
  线上业务如果只连接副本集主节点，一旦数据库发生 HA 会造成写入中断；如果只连接单个 mongos，这个 mongos 故障后会造成业务中断。
- **线上业务禁止设置 Write Concern j:false**
  Write Concern 默认一般为 j:true，表示服务端会写入 journal log 完成后再向 client 端返回。一般请勿设置 j:false，否则进程突然故障重启后，可能会造成数据丢失。
- **update 语句中禁止不带条件的更新**
  推荐保持 multi 为默认值（false），避免程序 bug（如由于某些异常造成 query 参数传了{}）造成全表数据更新。
- **禁止更新数组内部分元素时，将数组全部拿出来更新后再写回去**
  推荐使用 arrayFilters 仅对需要的元素进行修改。

建议类

- **建议局部读写而不是全读全写**
  查询语句中应尽量使用 $projection 运算符投影出需要的字段；在 update 命令中如果只是修改某个字段，建议使用 $set，请勿将文档全部读出来修改后再全量写进去。
- **线上环境慎重使用 db.collection.renameCollection() 命令**
  renameCollection() 在4.0及之前的版本会阻塞 db 的所有操作；在4.2及其之后版本会阻塞当前表及目标表的操作。而且 renameCollection() 执行期间会造成游标失效、changeStream 失效及带 --oplog 命令的 mongodump 失败等问题。线上环境禁止高峰期直接操作。
- **建议核心业务配置 WriteConcern 为 {w: "majority"} 参数**
  默认情况下，一般驱动的 WriteConcern 配置为 {w:1}，即在主节点写入完成后认为请求成功。如果机器突然发生故障并且写入的数据还未复制到从节点，这样的配置会导致数据丢失。因此对于线上的核心业务，建议配置 {w: "majority"}，这样的配置会等数据同步大多数节点后再返回客户端。当然可靠性和性能不能兼顾，选择了 {w: “majority” } 配置后请求的延迟也会相应的增加。
- **建议在低峰期执行 dropDatabase 操作**
  MongoDB 4.0 中 dropDatabase 命令需获取数据库级别排他锁（X 锁），必须等待所有读写锁释放后才能执行。当存在长查询、写入操作、未关闭游标或后台索引构建时会产生阻塞，而该版本新增的多文档事务可能长时间持有锁，进一步加剧了 dropDatabase 的阻塞风险。
- **建议设置事务超时时间**
  MongoDB 4.0 中事务操作与 oplog 读取共享同一读取 tickets 池。极端情况可能因为事务持有 tickets 并等待 oplog 同步，而 oplog 读取又因等待 tickets 被阻塞，两者形成循环等待，导致死锁。
- **建议使用 MongoDB 官方推荐的事务回调 API 编写事务逻辑**
  事务回调 API 内部自动处理了事务重试逻辑（例如遇到 TransientTransactionError 等错误），能在发生网络波动或短暂锁冲突时安全地重试事务，显著提升事务成功率并减少因手动重试逻辑缺失造成的事务挂起风险。

不建议类

- **除非必要，不要在高性能场景大量使用多文档事务**
  MongoDB 4.0及之后的版本，MongoDB 提供了多文档事务。但是多文档事务只是 MongoDB 数据库能力的补充，在高并发高性能场景下，大规模使用多文档事务需要进行充分的压测。
  一般来说，多文档事务提交前需要在内存中保留快照，这可能消耗大量的 cache 从而导致性能下降。
- **不建议使用短连接**
  MongoDB 的认证逻辑是一个比较复杂的运算过程，而且默认 MongoDB 会为每个连接创建一个线程。大量短连接会对数据库产生较大的负担，特别是没有 mongos 的副本集集群。建议使用长连接，详细参考 MongoDB URL 中 Connection Pool 参数。

分片集群设计规范

禁止类

- **如果使用 _id 字段作为片键，禁止使用范围分片**
  id 默认是一个递增的序列，随着数据量的增加会一直增大。如果 _id 作为片键并使用范围分片，集群随着数据的插入不断的进行 balance。
- **分片集群禁止直连 mongod 节点写数据**
  分片集群应该通过 mongos 写数据，直接通过 mongod 写入的数据无路由信息，会导致访问不到。
- **线上环境禁止长时间关闭 balancer 和 autoSplit 配置**
  关闭 balance 会导致片之间数据不均衡，关闭 autoSplit 可能会产生 jumbo chunk。
- **分片表尽量避免不带片键的查询**
  分片表不带片键进行查询，需要扫描所有分片后在 mongos 聚合结果，比较消耗性能，不推荐使用。
- **线上环境务必设置 balancer 窗口，避免 balance 对业务造成影响**
  balance 过程会明显对数据库造成较大的压力，建议放在业务低峰期进行。

建议类

- **建议使用区分度较大的字段作为片键，最理想的情况是使用唯一主键作为片键**
  假设我们有一个存储人口信息的集合，其使用性别作为片键，这个片键认为区分度较低，因为集合中性别字段相同的数据理论上会有一半之多。
  如果片键区分度不大，可能导致大量的记录集中在某些片上，而这种不均衡也无法添加分片进行扩展。因此建议使用区分度较大的字段作为片键。
- **如果使用 hash 分片，建议进行预分配，特别是表比较大且经常需要大量插入数据**
  shardCollection() 命令默认每个分片只会创建2个 chunk，随着数据量的越来越大，MongoDB 需要不断的 balance 和 splitChunk，这将对数据库带来较大的负担。
  因此在对于大集合，建议提前进行预分片（shardCollection 命令指定 numInitialChunks 参数，每个分片最大支持8192个），特别是向大集合中批量导数据。

不建议类

- **没有按片键顺序扫描的强需求，不建议使用 range 分片，推荐 hash 分片**
  range 分片容易引起不均衡和数据热点，而且因为无法预分片所以随着数据的写入 balance 不可避免，因此不建议使用，除非有特殊的按片键范围查询需求。

  > ⚠ **注意：**
  > 在 shardCollection 的时候，sh.shardCollection("records.people", { zipcode: 1 }) 命令中1表示范围分片，sh.shardCollection("records.people", { zipcode: "hashed" }) 命令中"hashed"表示 hash 分片。需注意不要用错。

- **分片集群中不建议使用非分片表**
  MongoDB 的分片集群如果未执行 shardCollection 命令，默认数据只存储在主分片上。大量未分片的表会造成分片和分片间的数据量不一致。集群长时间运行下去，可能会造成某些片数据量特别多甚至会打满磁盘，运维在这种情况下不得不使用 movePrimary 手动进行数据搬迁，从而增加了运维复杂度。