

hdmap-ir 设计文档 (更新版)

Lanelet2 + OpenDRIVE 双格式接入 · 统一 IR · Tile 增量更新 · 高性能查询

文档状态: 草稿 (Draft)

版本: v0.3

日期: 2026-01-29

维护者: James

目录

一、核心组成部分

1. 文档概述
2. 项目概述
3. 系统架构设计
4. 模块/组件设计
5. 接口设计
6. 数据设计
7. 关键技术决策
8. 性能设计
9. 安全考虑
10. 测试策略
11. 部署与构建
12. 附录

二、C++ 特定考虑要点

三、文档编写建议

四、简洁模板（小型项目适用）

一、核心组成部分

1. 文档概述

本设计文档面向“没有高精地图/定位算法基础”的读者，目标是让你读完后能回答四个问题：

- 这个系统要解决什么问题（价值是什么）？
- 输入是什么、输出是什么、关键边界在哪里？
- 模块如何拆分、关键数据结构是什么、线程如何跑？
- 我下一步应该先写哪些代码、怎么验收？

1.1 文档状态（草稿/评审/发布）

当前状态：草稿（Draft）。建议在完成 MVP（能够加载数据 + 建索引 + 提供 NearestLane/Project/BBox）后进入评审（Review），发布（Release）版本与 v1.0 tag 对齐。

1.2 版本历史

版本	日期	变更摘要	作者
v0.1	2026-01-28	初稿：基础架构/模块/接口/性能/测试	James
v0.2	2026-01-29	更新：Tile 化增量更新、多版本共存、线性参考(LRS)查询、序列化与 mmap 方案、数据质量体系	James
v0.3	2026-01-29	统一项目命名为 hdmap-ir；同步更新库/二进制命名与示例。	James

1.3 文档维护者

- 维护者：James（负责设计变更同步、公共接口与 proto 版本管理）
- 建议流程：任何公共 API/数据格式/关键决策变更必须通过 PR 更新本设计文档，并在“决策记录”增加条目。

1.4 相关参考文档

建议在仓库 docs/references.md 维护参考链接。本文档不依赖外部链接即可理解实现路径。

2. 项目概述

2.1 项目目标：解决什么问题，提供什么价值

hdmap-ir 的目标是把两类高精地图数据源（Lanelet2 的 .osm 与 OpenDRIVE 的 .xodr）统一成一个可计算、可索引、可服务化的内存表示（IR），并对外提供高性能查询接口。

它为上层“定位/匹配/路由”服务提供三类底座能力：

- 候选检索：给定位置点，快速找到附近 TopK 车道候选（NearestLane）。
- 几何投影：把点投影到车道中心线，得到 s/l/距离等（ProjectToLane）。
- 拓扑与路网支持：提供车道连通关系与线性参考查询（successor/pred + LRS 查询），支撑路由与 route-aware map matching。

2.2 范围定义：包含/不包含的功能边界

为了在有限时间内做出“能面试的闭环版本（MVP）”，本项目按优先级划定边界：

包含（MVP 必须具备）：

- 双格式输入：Lanelet2(.osm) 与 OpenDRIVE(.xodr)。
- Lanelet2 双解析策略：Official（依赖 Lanelet2 库）与 OSM Lite（自研轻量解析子集）。
- 统一 IR（MapIR）：Lane（中心线）+ Segment（线段）+ 基本拓扑（successor/pred）+ 最小属性（如道路等级/限速，可缺省）。
- Tile 化地图组织：地图被切分为若干空间 tile；每个 tile 独立持有 TileIR + TileIndex。
- 多版本共存：至少支持 prod/canary 两版本同时在内存服务；按访问热度淘汰其他版本（LRU）。

- 查询接口：NearestLane、ProjectToLane、QueryBBox，以及线性参考(LRS)查询（ForwardAlongLane/SliceLaneByS）。
- 持久化：MapIR/索引序列化（支持快速重启加载），可选 mmap 读取。
- 数据质量：Validator + IssueReport + 少量 AutoFix（去重、断链修复等）。

不包含（明确延后，可写入 Future Work）：

- 全国尺度 TB 级地图的完整磁盘索引与按需加载（这里我们做 tile + 序列化 + 预留接口）。
- OpenDRIVE / Lanelet2 的全部语义规则（信号灯、交通规则全量、可变车道全量）。
- 真正的“地图差分发布链路”（线上数据生产与分发），本项目仅提供 Delta 应用机制（概念与工程骨架）。
- 更复杂索引结构（R-tree 变种、分层网格自适应等）作为对照实现可以做，但不阻塞 MVP。

2.3 目标用户：最终用户和技术用户

- 最终用户：定位/匹配/路由等上层服务（项目 2）或任何需要地图候选检索的模块。
- 技术用户：C++ 地图基础设施开发者；需要读懂数据结构、调参、压测与定位性能问题。

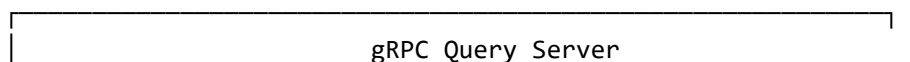
2.4 技术约束：C++ 标准、平台、性能要求

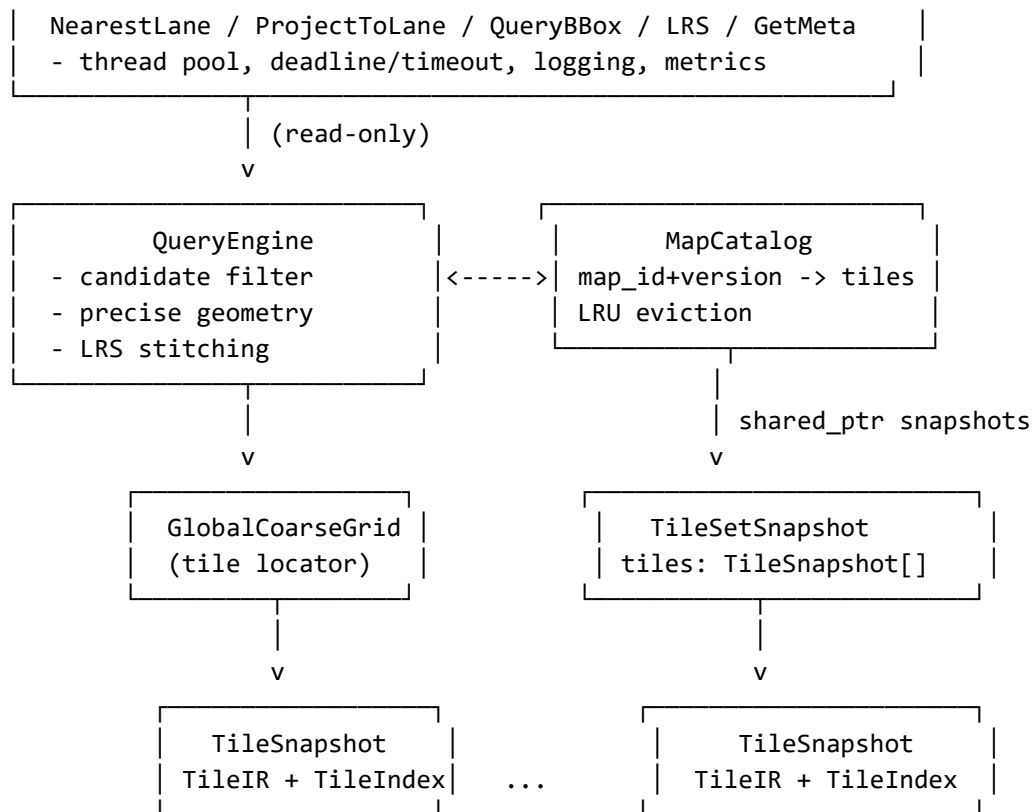
- C++ 标准：C++17（兼顾生态与现代特性，适配 gRPC 等依赖）。
- 平台：MVP 以 Linux x86_64 为主；开发兼容 macOS。
- 性能目标（建议写进 README 并持续更新）：
 - - NearestLane：小/中等地图 P95 < 10ms；并发下 P95 波动可控。
 - - ProjectToLane：P95 < 5ms（候选已缩小）。
 - - QueryBBox：P95 < 20ms（bbox 范围受限）。
 - - Reload/Delta 应用：局部 tile 更新不阻塞查询；swap 时间在毫秒级。

3. 系统架构设计

3.1 总体架构图：高层次组件关系（框架图）

系统采用“写路径构建 + 读路径查询”的分离架构。写路径生成不可变快照，读路径在并发环境下只读访问。





Write path (build/update):

Lanelet2/OpenDRIVE -> Normalize -> Validate -> Build Tiles -> Build Index -> Publish (atomic swap per tile / per version)

3.2 架构模式：分层架构 + 快照 (Snapshot) + Tile 化增量更新

架构采用分层设计：IO/Parser -> Normalize/Validate -> IR -> Index -> Query -> Service。

为了支持局部更新与多版本共存，核心对象从“单一 snapshot”升级为：

- MapCatalog：管理多个地图版本 (prod/canary/实验版本)。
- TileSetSnapshot：一个版本对应一组 tile；每个 tile 是一个 TileSnapshot。
- TileSnapshot (不可变)：TileIR + TileIndex 的只读组合，通过 shared_ptr 共享。
- 更新策略：仅重建受影响 tile，然后原子替换对应 tile 指针 (Copy-on-Write 思路)。

3.3 部署视图：跨平台部署与两种集成方式

部署形态建议：

- 形态 A (推荐)：独立 gRPC 服务 hdmapi_server，供项目 2 或其他服务调用。

- 形态 B：作为静态库链接到业务服务（延迟更低，依赖更少）。

跨平台要点：

- 构建系统使用 CMake + vcpkg/conan，尽量避免平台特定 API；必要时用 `#ifdef` 封装在 util 层。
- 性能工具：Linux 以 perf 为主，macOS 使用 Instruments（文档记录方法即可）。

4. 模块/组件设计

4.1 模块划分：职责与边界（面向新手的理解方式）

你可以把系统拆成三条线：输入线（加载地图）、底座线（IR+索引）、服务线（对外接口）。

- io/*：把不同格式文件读出来，得到“原始 lane 几何与拓扑”。
- normalize/*：把坐标、采样、属性统一，生成 MapIR/TileIR。
- validate/*：检查错误并输出报告；可做少量修复。
- index/*：把大量 segments 放进索引，支持快速候选检索。
- query/*：把索引返回的候选做精确几何计算，输出业务可用结果。
- catalog/*：管理多个版本、多个 tile 的快照生命周期。
- service/*：gRPC server，处理并发、超时、日志、指标。

4.2 类设计：关键类（文字版 UML）

MapCatalog

- Get(map_id, version) -> shared_ptr<const TileSetSnapshot>
- Put(map_id, version, snapshot)
- EvictLRU()

TileSetSnapshot (immutable)

- map_id, version, meta
- shared_ptr<const GlobalCoarseGrid> locator
- vector<shared_ptr<const TileSnapshot>> tiles

TileSnapshot (immutable)

- TileId id, bounds
- shared_ptr<const TileIR> ir
- shared_ptr<const TileIndex> index

QueryEngine

- shared_ptr<const MapCatalog> catalog
- NearestLane(req) / ProjectToLane(req) / QueryBBox(req)
- LRS: ForwardAlongLane(lane_id, s, dist)

TileIndex (interface)

- QueryRadius(local_xy, r) -> vector<SegId>
- QueryBBox(aabb) -> vector<SegId>

GridIndex : TileIndex

- cell_size
- cells -> seg list (可选 compact CSR)

Validator

- Validate(TileIR/MapIR) -> IssueReport
- AutoFix(policy) -> fixed_ir

4.3 内存管理策略：智能指针、对象生命周期、可选内存池

核心原则：发布后只读（immutable），共享用 shared_ptr，构建期可变。

- TileSnapshot/TileSetSnapshot: shared_ptr<const ...>, 读路径不修改，避免锁。
- 构建期：使用 std::vector 并 reserve，减少 realloc；对象尽量连续存储提升 cache 命中。
- 可选优化（后置）：对 grid cells 的大量小分配引入 pool allocator（必须以 profiling 为依据）。

4.4 并发模型：查询不中断、更新不阻塞（关键点讲清楚）

并发模型分两部分：

- 读路径（查询）：多线程并发访问 TileSnapshot，只读，无锁。
- 写路径（更新）：后台线程解析 delta/重建 tile，完成后对 tile 指针做 atomic swap（旧 tile 仍可被正在执行的查询安全访问）。

更新过程中查询不中断的原因（新手要点）：shared_ptr 确保旧对象在没有引用时才释放，所以即使我们把 catalog 里的指针换成新的，正在用旧指针的线程也不会崩溃。

5. 接口设计

5.1 API 设计：公共头文件与最小稳定接口

公共接口建议保持“少而稳定”，内部实现可以频繁迭代。对外提供两套接口：C++ library API 与 gRPC API。

C++ Library API（示例）：

```
// loader
StatusOr<std::shared_ptr<const TileSetSnapshot>>
LoadMapVersion(const LoadOptions& opt);

// catalog
Status PutVersion(const std::string& map_id, const std::string& version,
                  std::shared_ptr<const TileSetSnapshot> snap);
StatusOr<std::shared_ptr<const TileSetSnapshot>>
GetVersion(const std::string& map_id, const std::string& version);

// query
NearestLaneResp NearestLane(const NearestLaneReq&);
ProjectResp      ProjectToLane(const ProjectReq&);
BBoxResp         QueryBBox(const BBoxReq&);
LrsResp          ForwardAlongLane(const LrsReq&);
```

5.2 接口约定：错误码、异常规范、资源所有权

约定：库 API 不抛异常，使用 Status/StatusOr。服务层把错误码映射到 gRPC status 或业务 status 字段。

错误码	含义	典型场景
INVALID_ARGUMENT	参数不合法	radius<=0、bbox 反转、 version 为空
NOT_FOUND	未找到结果	附近无候选车道、lane_id 不存在
PARSE_ERROR	解析失败	OSM/XODR XML 不合法或缺 关键字段

UNSUPPORTED	不支持的特性	遇到未实现的 OpenDRIVE 元素
RESOURCE_EXHAUSTED	资源不足/超时	候选过多、deadline 到期
INTERNAL	内部错误	不变量被破坏、未知异常

资源所有权：所有 snapshot 为只读共享，调用方拿到 shared_ptr<const ...> 不允许修改内部数据；返回结果采用值语义（struct/vector）。

5.3 ABI 兼容性：二进制兼容考虑

MVP 以源码集成（静态库/编译时集成）为主。若未来需要发布二进制：建议对外 API 使用 PImpl 或提供 C API 外壳，并对数据格式做版本化（mapir.bin header 包含版本号）。

6. 数据设计

6.1 数据结构：核心定义（新手友好解释）

理解数据结构的关键：我们只需要能做“距离与拓扑”的计算。中心线 polyline 是最核心的几何。

```

Wgs84Point { double lat, lon; }
LocalPoint { float x, y; }    // meters in local ENU or local x/y

Polyline = vector<LocalPoint>

Lane {
    LaneId id;
    Polyline centerline;
    vector<LaneId> successors;
    vector<LaneId> predecessors;    // optional
    Attrs attrs;                   // speed_limit, road_class ...
    CumS cum_s;                    // cumulative arc-length for LRS queries
}

Segment {
    LaneId lane_id;
    LocalPoint a, b;
    AABB bbox;
    float s0, s1;                  // arc-length range on the lane
}

```

6.2 数据流：从文件到可查询快照

Lanelet2/OpenDRIVE file

- > Parse (source-specific)
- > Normalize
 - coordinate to local meters
 - resample centerline (max step, e.g. 1.0m)
 - build cum_s for LRS
 - segmentize (segments with s0/s1 and bbox)
- > Validate + AutoFix
- > Tile split (assign lanes/segments to tiles)
- > Build TileIndex (GridIndex)
- > Publish TileSetSnapshot into MapCatalog

6.3 持久化方案：mapir.bin + index.bin + delta.json（最小可用）

为了支持快速重启与多节点分发，建议实现二进制序列化。MVP 先做到可读可写即可，后续再优化到 mmap。

- mapir.bin: TileSetSnapshot 的核心数据 (lanes/points/topology/attrs/cum_s)。
- index.bin: TileIndex (可选，若不存则启动时重建)。
- delta.json: 增量更新输入 (合成数据即可)，描述新增/删除/修改 lanes 或属性。

建议的 delta.json 结构 (示例)：

```
{
  "map_id": "city_demo",
  "base_version": "prod_2026w01",
  "target_version": "canary_2026w01_patch1",
  "tiles": [
    {
      "tile_id": "x12_y34",
      "ops": [
        {"op": "update_attr", "lane_id": 10021, "speed_limit": 8.33},
        {"op": "add_lane", "lane": {"id": 20001, "centerline_local":
[[0,0],[10,0]], "successors":[20002]}}
      ]
    }
  ]
}
```

解释：你不需要真的对接施工数据；只要做出“局部 tile 更新生效且不中断查询”的机制，就已经非常生产化。

7. 关键技术决策

7.1 第三方库选择（按必要性分级）

- 必须：gRPC/protobuf（服务接口）、GoogleTest（测试）、tinyclang（解析 OpenDRIVE 与 OSM Lite）。
- 可选：Lanelet2 官方库（作为 Official 解析策略）；spdlog（结构化日志）。
- 后置：Prometheus exporter（指标）、benchmark 框架（Google Benchmark）。

7.2 构建系统与依赖管理

- 构建系统：CMake（跨平台、生态成熟、易集成 gRPC）。
- 依赖管理：vcpkg（推荐）或 conan（二选一，建议先 vcpkg）。
- 工程建议：提供 CMake Presets（dev/release/asan/tsan），便于新手一键构建。

7.3 平台相关代码处理

- 所有平台相关逻辑封装在 util/platform_* 中，通过 #ifdef 分发。
- 业务逻辑（IR/Index/Query）不允许出现平台 ifdef。

7.4 模板与编译期技巧（克制使用）

MVP 不追求模板炫技，优先可读可测。允许的模板用途：StrongId、Span/View、简单泛型算法。

8. 性能设计

8.1 性能指标：延迟、吞吐、内存、构建时间

建议每次 milestone 更新以下表格，并在 README 展示。

指标	目标（MVP）	测量方法	备注
NearestLane P95	< 10ms	固定 10k queries，统计 P95	并发下观察抖动
Project P95	< 5ms	候选已缩小后投影	几何热点
QueryBBox P95	< 20ms	bbox 限制大小	可分页

Index build time	可接受 (记录)	parse/normalize/index 分段	支持 tile 增量更新
RSS 内存	可控 (记录)	/proc/self/statm 或工具	对比序列化/压缩前后

8.2 优化策略：以 profiling 驱动的优先级

- 优先级 1：减少候选数量 (tile coarse grid + tile grid 参数调优)。
- 优先级 2：减少分配与提升 cache locality (points 连续存储、vector reserve)。
- 优先级 3：读路径无锁 (immutable snapshot + atomic swap)。
- 优先级 4：缓存 (tile 级/查询级 LRU)，必须用命中率与 P95 变化证明价值。

8.3 Profiling 方案：怎么定位慢在哪里 (新手可执行)

- 基准：先写 bench 程序，固定地图 + 固定随机种子生成 queries。
- CPU：Linux 用 perf record/report，找热点函数 (通常在投影、候选遍历、cell 去重)。
- 内存：用 heaptrack 或简单统计 (对象数量、vector 容量)，观察是否存在大量小分配。
- 结论必须写进 docs/perf_report.md：热点、原因、改动、前后数字。

9. 安全考虑

9.1 内存安全：缓冲区溢出防护

- Debug preset 开启 ASAN/UBSAN；CI 至少跑一次 sanitizer 构建。
- 解析输入设置上限 (最大节点数、最大文件大小)，避免恶意输入导致 OOM。
- 避免裸指针所有权不清：对外只暴露 shared_ptr<const ...> 或值语义结果。

9.2 异常安全：异常保证级别

- 库 API 不抛异常 (Status/StatusOr)。
- 内部如使用异常，必须在模块边界捕获并转为 Status，避免异常穿透到调用方。
- 关键路径尽量标注 noexcept (尤其是析构与小工具函数)。

9.3 代码安全：静态分析工具集成

- clang-tidy (推荐) 集成到 CI。
- cppcheck (可选)。

- 若时间允许：对 OSM Lite / OpenDRIVE parser 做 fuzz（加分项）。

10. 测试策略

10.1 单元测试：GoogleTest

- geo：点到线段距离、投影、退化段、精度边界。
- io：解析错误、缺字段、空文件、重复 id。
- index：grid vs brute 结果一致性（固定随机种子）。
- query：Nearest/Project 的 golden case（固定输入与期望输出）。

10.2 集成测试：组件间集成方案

- end-to-end：加载地图 -> 构建 tiles/index -> 执行 1k queries -> 校验结果数量与一致性。
- server：启动 gRPC server -> client 发请求 -> 校验 response 与错误码映射。
- delta 更新：应用 delta -> 仅受影响 tile 更新 -> 查询结果变化符合预期。

10.3 性能测试：基准测试设计

- NearestLane：固定 10k queries，统计 P50/P95/QPS。
- IndexBuild：统计 parse/normalize/index_build 三段耗时。
- 并发：线程数 1/2/4/8/16 下 P95 与 QPS 曲线。

10.4 Mock 策略：依赖隔离方案

- QueryEngine 可 mock TileIndex（返回固定候选集合），专注测试几何与过滤逻辑。
- Service 层可 mock QueryEngine，专注测试 timeout、并发、错误码。

11. 部署与构建

11.1 依赖管理：vcpkg/conan 或源码集成

- 推荐 vcpkg：使用 manifest + lockfile 保证可复现。
- 依赖：grpc, protobuf, gtest, tinyxml2, spdlog（可选）。

11.2 交叉编译：多平台构建配置

- 提供 CMake toolchain 示例（aarch64 可选）。
- 避免在核心算法中使用平台特定指令；若使用需封装并提供 fallback。

11.3 安装打包：安装程序或包设计

- 输出：libhdmapi + public headers + hdmapi_server（可选）。
- Docker：multi-stage（builder + runtime），runtime 只包含二进制与配置，地图数据通过 volume 挂载。

12. 附录

12.1 术语表

- IR：Intermediate Representation，统一地图中间表示。
- Snapshot：不可变快照对象，支持并发读与原子替换。
- Tile：空间分块单元（如 1km x 1km），用于增量更新与分层索引。
- LRS：Linear Referencing System，沿中心线弧长 s 的线性参考查询。

12.2 参考资料

建议维护在 docs/references.md（Lanelet2/OpenDRIVE/gRPC/性能工具等）。

12.3 决策记录：重要技术决策的论证过程

示例记录：

- 为何采用 Tile + atomic swap：在读多写少场景下，能够实现局部增量更新且查询不中断；实现复杂度可控。
- 为何先用 Uniform Grid：简单、易调参、可做 benchmark 量化；未来可加 R-tree 对照作为实验与取舍亮点。
- 为何必须做 LRS：规划/匹配常见需求（前向/后向距离查询），实现成本低但价值高。

二、C++ 特定考虑要点

RAII 原则应用：资源管理设计模式

- 线程池、gRPC server、文件句柄、计时器都封装成 RAII 类，确保异常/早退不泄漏。
- snapshot 共享用 `shared_ptr<const T>`，生命周期自动管理。

移动语义：大对象传递优化

- 构建期生成的大 vector 在发布 snapshot 时 move，避免深拷贝。
- 返回结果尽量使用值语义 + NRVO；必要时显式 `std::move`。

模板特化：泛型设计的具体实现

- 克制使用模板，优先可读可测。
- 允许：StrongId、Span/View、少量泛型算法。

编译期计算：constexpr/constexpr 使用

- constexpr 用于默认参数、错误码映射等小常量。
- 不为炫技使用 constexpr；若升级到 C++20 再评估。

模块化：C++20 Modules 的使用（如适用）

本项目以 C++17 为基线，暂不引入 Modules，避免工具链与依赖复杂度。

三、文档编写建议

- 保持简洁：重点描述设计决策与取舍，不复制实现细节代码。
- 图文并茂：至少维护总体架构图与关键流程图；任何关键决策都要配图或数据表。
- 版本控制：设计文档与代码同仓库同 PR；公共接口与数据格式变更必须更新版本历史。
- 持续更新：每次 milestone 更新性能指标、数据质量统计、已知限制与未来工作。

四、简洁模板（小型项目适用）

若未来将某个子模块拆成小项目，可简化为以下结构：

- 概述（目标、范围）
- 核心类设计（数据结构、所有权）
- 关键算法/流程（输入-处理-输出）
- 接口定义（错误码、线程安全）
- 依赖和构建说明（CMake、测试、bench）