# Contents



# 自动驾驶数据闭环系统软件架构与详细设计文档

## 1. 引言

### 1.1 文档目的

本文档旨在详细描述自动驾驶数据闭环系统的软件架构和设计，涵盖系统的整体架构、模块划分、接口设计、数据流以及关键实现细节。该文档面向系统设计人员、开发人员和测试人员，为他们提供系统实现的详细指导。

### 1.2 系统概述

自动驾驶数据闭环系统是一个集成的数据采集、处理、分析和优化系统，旨在通过闭环反馈机制优化自动驾驶车辆的数据采集策略和路径规划算法。系统主要包括以下几个核心模块：

1. **数据采集模块** - 负责实际的数据采集工作
2. **导航规划模块** - 负责路径规划和导航决策
3. **状态机模块** - 协调各模块的工作流程
4. **数据处理与分析模块** - 分析采集的数据并优化系统参数

### 1.3 术语和缩写

| 术语 | 说明 |
|---|---|
| AD | Autonomous Driving (自动驾驶) |
| DCL | Data Closed Loop (数据闭环) |
| ROI | Region of Interest (感兴趣区域) |
| RL | Reinforcement Learning (强化学习) |

## 2. 系统架构设计

### 2.1 整体架构

系统采用模块化设计，主要分为以下几个层次：

1. **应用层** - 包含主控制器和状态机
2. **核心功能层** - 包含数据采集、导航规划等核心功能模块
3. **工具层** - 包含日志、配置管理等通用工具
4. **接口层** - 提供与外部系统的接口

```
graph TD
    A[主控制器] --> B(状态机)
    B --> C[数据采集模块]
    B --> D[导航规划模块]
    B --> E[数据处理模块]

    C --> F[数据存储]
    C --> G[数据上传]
    C --> H[触发管理]

    D --> I[成本地图]
    D --> J[路径规划器]
    D --> K[采样优化器]
    D --> L[语义地图]
    D --> M[约束检查器]

    E --> N[覆盖率度量]
    E --> O[奖励计算器]
    E --> P[数据分析器]
```

### 2.2 模块划分

#### 2.2.1 主要模块

1. **DataCollectionPlanner** - 数据采集规划器，作为系统的主控制器
2. **NavPlannerNode** - 导航规划节点，负责路径规划和导航决策
3. **StateMachine** - 状态机，协调各模块的工作流程
4. **DataStorage** - 数据存储模块
5. **DataUploader** - 数据上传模块
6. **TriggerManager** - 触发管理模块
7. **StrategyParser** - 策略解析模块

#### 2.2.2 辅助模块

1. **CostMap** - 成本地图管理
2. **RoutePlanner** - 路径规划器
3. **SamplingOptimizer** - 采样优化器
4. **SemanticMap** - 语义地图
5. **SemanticConstraintChecker** - 语义约束检查器
6. **CoverageMetric** - 覆盖率度量
7. **RewardCalculator** - 奖励计算器

8. **DataCollectionAnalyzer** - 数据分析器

### 2.3 数据流设计

系统数据流主要包括以下几个阶段：

1. **初始化阶段** - 各模块初始化和配置加载
2. **任务规划阶段** - 根据任务区域规划最优路径
3. **执行采集阶段** - 沿路径执行数据采集
4. **反馈优化阶段** - 分析采集数据并优化系统参数
5. **数据上传阶段** - 将采集的数据上传至云端

```
sequenceDiagram
    participant M as Main
    participant DCP as DataCollectionPlanner
    participant NP as NavPlannerNode
    participant SO as SamplingOptimizer
    participant CM as CostMap
    participant TM as TriggerManager
    participant DS as DataStorage
    participant DU as DataUploader
    participant DA as DataCollectionAnalyzer

    M->>DCP: initialize()
    DCP->>NP: initialize()
    NP->>NP: loadConfiguration()
    NP->>CM: setParameters(sparse_threshold, exploration_bonus, redundancy_penalty)
    DCP->>TM: initialize()
    DCP->>DS: initialize()
    DCP->>DU: initialize()

    M->>DCP: setMissionArea(mission)
    DCP->>NP: setGoalPosition(center)

    M->>DCP: planDataCollectionMission()
    DCP->>NP: planGlobalPath()
    NP->>NP: computeAStarPath(costmap, current_position, goal_position)
    NP-->>DCP: collection_path

    DCP->>NP: getCurrentPosition()
    NP-->>DCP: current_pos

    loop For each path segment
        DCP->>TM: shouldTrigger(waypoint)
        TM-->>DCP: true/false
        DCP->>NP: optimizeNextWaypoint()
```

```
    NP->>SO: optimizeNextSample(costmap, current_position)
    SO->>CM: getDataDensity(x, y)
    SO->>CM: getCellCost(x, y)
    SO-->>NP: optimal_point
    NP-->>DCP: optimal_point
    DCP->>NP: setCurrentPosition(optimal_point)
end

DCP->>NP: setCurrentPosition(current_pos)
DCP-->>M: optimized_waypoints

M->>DCP: executeDataCollection(path)
loop For each waypoint
    DCP->>NP: setCurrentPosition(waypoint)
    DCP->>TM: shouldTrigger(waypoint)
    TM-->>DCP: true

    DCP->>DS: collectSensorData(waypoint)
    DS-->>DCP: sensor_data

    DCP->>DS: storeData(data_point)
    DCP->>TM: isInSparseArea(waypoint)
    TM-->>DCP: true/false
    DCP->>TM: getDistanceToNearestSparseArea(waypoint)
    TM-->>DCP: distance

    DCP->>NP: computeStateReward(prev_state, current_state)
    NP->>NP: RewardCalculator::computeReward(prev_state, current_state)
    NP-->>DCP: reward
end

DCP->>DCP: updateWithNewData(new_data)
loop For each data point
    DCP->>NP: addDataPoint(position)
    DCP->>collected_data_: push_back(data_point)
end

DCP->>NP: updateCostmapWithStatistics()
NP->>CM: updateWithDataStatistics(collected_data_points)
NP->>CM: adjustCostsBasedOnDensity()

DCP->>NP: updateCoverageMetrics(visited_cells)
NP->>NP: coverage_metric_->updateCoverage(costmap, visited_cells)

M->>DCP: reportCoverageMetrics()
```

```
DCP->>NP: getCoverageMetric()
NP-->>DCP: coverage_metric
DCP->>DCP: coverage_metric->getCoverageRatio()
DCP->>DCP: coverage_metric->getSparseCoverageRatio()

M->>DCP: analyzeAndExportWeights()
DCP->>DA: computeDensityMap(collected_data)
DA-->>DCP: heatmap

DCP->>DA: detectSparseRegions(heatmap)
DA-->>DCP: sparse_zones

DCP->>DA: adjustCostWeights(sparse_zones, current_weights)
DA-->>DCP: adjusted_weights

DCP->>DA: saveToPlannerConfig(adjusted_weights, config_path)

DCP->>NP: reloadConfiguration()
NP->>NP: loadConfiguration()
NP->>NP: initialize() [reinit with new params]

M->>DCP: uploadCollectedData()
DCP->>DU: uploadData(collected_data_)
DU-->>DCP: true/false
DCP->>collected_data_: clear()
```

## 3. 详细设计

### 3.1 状态机设计

状态机是系统的核心协调组件，负责管理系统的不同状态和状态转换。

#### 3.1.1 状态定义

1. **INITIALIZING** - 系统初始化中
2. **IDLE** - 空闲状态，等待任务指令
3. **PLANNING** - 路径规划中
4. **NAVIGATING** - 导航中，移动到下一个路径点
5. **DATA_COLLECTION** - 数据采集中
6. **UPLOADING** - 数据上传中
7. **ERROR** - 错误状态
8. **SHUTTING_DOWN** - 系统关闭中

#### 3.1.2 事件定义

1. **INIT_COMPLETE** - 初始化完成

2. **PLAN_REQUEST** - 请求路径规划
3. **PLAN_COMPLETE** - 路径规划完成
4. **NAVIGATION_START** - 开始导航
5. **WAYPOINT_REACHED** - 到达路径点
6. **TRIGGER_CONDITION** - 触发条件满足
7. **DATA_COLLECTED** - 数据采集完成
8. **UPLOAD_REQUEST** - 请求数据上传
9. **UPLOAD_COMPLETE** - 数据上传完成
10. **ERROR_OCCURRED** - 发生错误
11. **RECOVERY_REQUEST** - 请求恢复
12. **SHUTDOWN_REQUEST** - 请求关闭

#### 3.1.3 状态转换图

```
stateDiagram-v2
    [*] --> INITIALIZING
    INITIALIZING --> IDLE: INIT_COMPLETE
    IDLE --> PLANNING: PLAN_REQUEST
    PLANNING --> NAVIGATING: PLAN_COMPLETE
    NAVIGATING --> DATA_COLLECTION: TRIGGER_CONDITION
    DATA_COLLECTION --> NAVIGATING: DATA_COLLECTED
    NAVIGATING --> UPLOADING: 任务完成
    UPLOADING --> IDLE: UPLOAD_COMPLETE
    IDLE --> UPLOADING: UPLOAD_REQUEST
    IDLE --> SHUTTING_DOWN: SHUTDOWN_REQUEST
    PLANNING --> ERROR: ERROR_OCCURRED
    NAVIGATING --> ERROR: ERROR_OCCURRED
    DATA_COLLECTION --> ERROR: ERROR_OCCURRED
    UPLOADING --> ERROR: ERROR_OCCURRED
    ERROR --> IDLE: RECOVERY_REQUEST
    ERROR --> SHUTTING_DOWN: SHUTDOWN_REQUEST
    SHUTTING_DOWN --> [*]
```

### 3.2 数据采集模块设计

**3.2.1 DataCollectionPlanner** 作为系统的主控制器，DataCollectionPlanner 负责协调导航规划和数据采集工作。

**主要功能**：1. 初始化各子模块 2. 设置任务区域 3. 规划数据采集任务 4. 执行数据采集 5. 更新系统状态 6. 报告覆盖率指标 7. 分析并导出权重 8. 上传采集的数据

**接口设计**：

```cpp
class DataCollectionPlanner {
private:
    std::unique_ptr<NavPlannerNode> nav_planner_;
    std::unique_ptr<DataStorage> data_storage_;
```

```
    std::unique_ptr<DataUploader> data_uploader_;
    std::unique_ptr<TriggerManager> trigger_manager_;
    std::unique_ptr<StrategyParser> strategy_parser_;

    std::vector<DataPoint> collected_data_;
    MissionArea mission_area_;

public:
    DataCollectionPlanner(const std::string& config_file =
                          "/workspaces/ad_data_closed_loop/infra/navigation_planner/confi

    ~DataCollectionPlanner() = default;

    /**
     * @brief Initialize the data collection planner
     * @return true if initialization successful, false otherwise
     */
    bool initialize();

    /**
     * @brief Set the mission area for data collection
     */
    void setMissionArea(const MissionArea& area);

    /**
     * @brief Plan an optimal path for data collection mission
     * @return Path that optimizes data coverage
     */
    std::vector<Point> planDataCollectionMission();

    /**
     * @brief Execute data collection along a path using real data collection modules
     * @param path Path to follow for data collection
     */
    void executeDataCollection(const std::vector<Point>& path);

    /**
     * @brief Update planner with newly collected data
     * @param new_data Newly collected data points
     */
    void updateWithNewData(const std::vector<DataPoint>& new_data);

    /**
     * @brief Get coverage metrics for reporting
     */
```

```
    void reportCoverageMetrics();

    /**
     * @brief Analyze collected data and update planner weights
     */
    void analyzeAndExportWeights();

    /**
     * @brief Get collected data
     */
    const std::vector<DataPoint>& getCollectedData() const { return collected_data_; }

    /**
     * @brief Upload collected data to cloud
     */
    void uploadCollectedData();
};
```

#### 3.2.2 DataStorage　负责本地数据存储和采集触发。

#### 3.2.3 DataUploader　负责将采集的数据上传至云端。

#### 3.2.4 TriggerManager　管理采集触发策略。

#### 3.2.5 StrategyParser　解析采集策略配置。

### 3.3 导航规划模块设计

#### 3.3.1 NavPlannerNode　作为导航规划的核心节点，NavPlannerNode 集成所有导航规划组件。

**主要功能**：1. 初始化各子模块 2. 加载配置 3. 更新成本地图统计数据 4. 规划全局路径 5. 规划局部路径 6. 优化下一个航路点 7. 验证路径 8. 更新覆盖率指标 9. 计算状态奖励 10. 重新加载配置

**接口设计**：

```
class NavPlannerNode {
private:
    // Core components
    std::unique_ptr<CostMap> costmap_;
    std::unique_ptr<RoutePlanner> route_planner_;
    std::unique_ptr<SamplingOptimizer> sampling_optimizer_;
    std::unique_ptr<SemanticMap> semantic_map_;
    std::unique_ptr<SemanticConstraintChecker> constraint_checker_;
    std::unique_ptr<SemanticFilter> semantic_filter_;
    std::unique_ptr<CoverageMetric> coverage_metric_;
```

```
    // Configuration
    std::string config_file_path_;
    std::map<std::string, double> planner_parameters_;

    // State variables
    Point current_position_;
    Point goal_position_;
    std::vector<Point> global_path_;
    std::vector<Point> local_path_;

    // Data collection statistics
    std::vector<Point> collected_data_points_;

public:
    NavPlannerNode(const std::string& config_file =
                   "/workspaces/ad_data_closed_loop/infra/navigation_planner/config/plan

    ~NavPlannerNode() = default;

    /**
     * @brief Initialize the navigation planner node
     * @return true if initialization successful, false otherwise
     */
    bool initialize();

    /**
     * @brief Load configuration parameters from YAML file
     * @return true if loading successful, false otherwise
     */
    bool loadConfiguration();

    /**
     * @brief Update the costmap with new data statistics
     */
    void updateCostmapWithStatistics();

    /**
     * @brief Plan global path from current position to goal
     * @return Planned path
     */
    std::vector<Point> planGlobalPath();

    /**
     * @brief Plan local path/replan as needed
     * @return Local path
```

```
 */
std::vector<Point> planLocalPath();

/**
 * @brief Optimize next waypoint for data collection
 * @return Next optimal waypoint
 */
Point optimizeNextWaypoint();

/**
 * @brief Check if planned path satisfies all constraints
 * @param path Path to validate
 * @return true if path is valid, false otherwise
 */
bool validatePath(const std::vector<Point>& path);

/**
 * @brief Update coverage metrics based on visited locations
 * @param visited_cells List of visited cell coordinates
 */
void updateCoverageMetrics(const std::vector<std::pair<int, int>>& visited_cells);

/**
 * @brief Compute reward for current state transition
 * @param prev_state_info Information about previous state
 * @param new_state_info Information about new state
 * @return Computed reward
 */
double computeStateReward(const StateInfo& prev_state_info,
                          const StateInfo& new_state_info);

/**
 * @brief Reload planner configuration (e.g., when weights file is updated)
 */
void reloadConfiguration();

/**
 * @brief Add a new data point to collected data
 */
void addDataPoint(const Point& point);

/**
 * @brief Get current coverage metrics
 */
const CoverageMetric& getCoverageMetric() const { return *coverage_metric_; }
```

```
    /**
     * @brief Get current position
     */
    const Point& getCurrentPosition() const { return current_position_; }

    /**
     * @brief Set current position
     */
    void setCurrentPosition(const Point& position) { current_position_ = position; }

    /**
     * @brief Get goal position
     */
    const Point& getGoalPosition() const { return goal_position_; }

    /**
     * @brief Set goal position
     */
    void setGoalPosition(const Point& position) { goal_position_ = position; }

    /**
     * @brief Get global path
     */
    const std::vector<Point>& getGlobalPath() const { return global_path_; }

    /**
     * @brief Get local path
     */
    const std::vector<Point>& getLocalPath() const { return local_path_; }
};
```

**3.3.2 CostMap** 管理成本地图与数据密度信息。

**3.3.3 RoutePlanner** 执行 $A^*$ 等路径规划算法。

**3.3.4 SamplingOptimizer** 优化数据采集点选择。

**3.3.5 SemanticMap** 维护环境语义对象（道路、交通标志等）。

**3.3.6 SemanticConstraintChecker** 检查路径合规性与数据采集约束。

**3.3.7 CoverageMetric** 计算区域覆盖率与稀疏区覆盖情况。

#### 3.3.8 RewardCalculator 基于状态变化计算奖励值。

### 3.4 数据分析模块设计

#### 3.4.1 DataCollectionAnalyzer 分析采集数据分布，动态调整规划参数。

**主要功能**：1. 计算密度图 2. 检测稀疏区域 3. 调整成本权重 4. 保存规划器配置

**接口设计**：

```
class DataCollectionAnalyzer {
public:
    struct Heatmap {
        std::vector<std::vector<double>> density_values;
        int width, height;
        double resolution;

        Heatmap(int w = 0, int h = 0, double res = 1.0)
            : width(w), height(h), resolution(res) {
            density_values.resize(h, std::vector<double>(w, 0.0));
        }
    };

    struct Region {
        Point center;
        double radius;
        bool is_sparse;

        Region(const Point& c = Point(), double r = 0.0, bool sparse = false)
            : center(c), radius(r), is_sparse(sparse) {}
    };

    struct PlannerWeights {
        double sparse_threshold;
        double exploration_bonus;
        double redundancy_penalty;
        double grid_resolution;

        PlannerWeights(double threshold = 0.2, double bonus = 0.5,
                       double penalty = 0.4, double resolution = 1.0)
            : sparse_threshold(threshold), exploration_bonus(bonus),
              redundancy_penalty(penalty), grid_resolution(resolution) {}
    };

    /**
     * @brief Compute density map from collected data
```

```
     */
    static Heatmap computeDensityMap(const std::vector<DataPoint>& data_points,
                                     int grid_width = 100, int grid_height = 100,
                                     double resolution = 1.0);

    /**
     * @brief Detect sparse regions in heatmap
     */
    static std::vector<Region> detectSparseRegions(const Heatmap& heatmap,
                                                   double sparse_threshold = 0.2);

    /**
     * @brief Adjust cost weights based on sparse zones
     */
    static PlannerWeights adjustCostWeights(const std::vector<Region>& sparse_zones,
                                            const PlannerWeights& current_weights);

    /**
     * @brief Save weights to planner configuration file
     */
    static void saveToPlannerConfig(const PlannerWeights& weights,
                                    const std::string& config_path);
};
```

## 4. 系统工作流程

### 4.1 初始化流程

1. 系统启动后进入 INITIALIZING 状态
2. 初始化 DataCollectionPlanner
3. 初始化 NavPlannerNode
4. 初始化各子模块（DataStorage、DataUploader、TriggerManager、StrategyParser）
5. 加载配置文件
6. 初始化完成后进入 IDLE 状态

### 4.2 任务规划流程

1. 在 IDLE 状态下接收 PLAN_REQUEST 事件
2. 进入 PLANNING 状态
3. 设置任务区域
4. 规划全局路径
5. 优化数据采集点
6. 规划完成后进入 NAVIGATING 状态

### 4.3 数据采集流程

1. 在 NAVIGATING 状态下接收 WAYPOINT_REACHED 事件
2. 检查是否满足触发条件
3. 如果满足触发条件，进入 DATA_COLLECTION 状态
4. 执行数据采集
5. 数据采集完成后返回 NAVIGATING 状态
6. 继续导航到下一个路径点

### 4.4 数据分析与优化流程

1. 任务完成后进入 UPLOADING 状态
2. 上传采集的数据
3. 分析采集的数据
4. 计算新的规划权重
5. 更新配置文件
6. 重新加载配置
7. 上传完成后回到 IDLE 状态

## 5. 关键技术实现

### 5.1 成本地图动态调整

系统根据数据密度统计动态调整成本地图：- 对数据稀疏区域（低于 sparse_threshold）给予 exploration_bonus 降低通行成本，鼓励探索 - 对数据密集区域施加 redundancy_penalty 增加成本，避免重复采集 - 使用 grid_resolution 控制密度计算粒度

### 5.2 强化学习奖励机制

奖励函数设计：- 访问新稀疏区域：+10.0（主要正向激励）- 成功触发数据采集：+0.5 - 碰撞/障碍物：-1.0（强惩罚）- 每步操作：-0.01（路径长度惩罚）- 可选的形状奖励引导向稀疏区域移动

### 5.3 语义感知规划

语义对象类型包括道路、交通标志、数据采集区等，语义约束检查应用于路径规划，语义信息转化为成本地图修正项。

## 6. 性能优化

### 6.1 内存优化

1. 使用智能指针管理内存
2. 及时释放不需要的资源
3. 优化数据结构减少内存占用

### 6.2 计算优化

1. 使用高效的路径规划算法

2. 优化成本地图更新逻辑
3. 减少不必要的计算

### 6.3 并发优化

1. 合理使用多线程
2. 避免竞态条件
3. 优化锁的使用

## 7. 错误处理与恢复

### 7.1 错误检测

1. 初始化失败检测
2. 路径规划失败检测
3. 数据采集失败检测
4. 数据上传失败检测

### 7.2 错误恢复

1. 进入 ERROR 状态
2. 记录错误日志
3. 尝试恢复操作
4. 恢复成功后回到 IDLE 状态

## 8. 测试策略

### 8.1 单元测试

1. 各模块功能测试
2. 接口测试
3. 边界条件测试

### 8.2 集成测试

1. 模块间接口测试
2. 完整工作流程测试
3. 异常处理测试

### 8.3 性能测试

1. 响应时间测试
2. 资源占用测试
3. 并发性能测试

# 9. 部署与运维

## 9.1 部署环境

1. Linux 操作系统
2. C++14 或更高版本编译器
3. 相关依赖库

## 9.2 配置管理

1. 使用 YAML 配置文件
2. 支持运行时动态加载
3. 配置参数验证

## 9.3 监控与日志

1. 状态转换日志
2. 错误日志
3. 性能日志

# 10. 未来扩展

## 10.1 功能扩展

1. 支持更多传感器类型
2. 增加更多语义类别
3. 支持更复杂的路径规划算法

## 10.2 性能优化

1. 进一步优化算法性能
2. 支持分布式部署
3. 增加缓存机制

## 10.3 可靠性提升

1. 增强错误处理机制
2. 支持热备份
3. 增加自检功能