多平台编队控制

三车编队控制代码技术说明文档

1. 简介

本项目实现的是三车编队控制初步框架，包括：

- **主车（master）**：负责路径跟踪、生成队列参考信息，并通过网络下发给从车。
- **从车 1（slave）**
- **从车 2（slave1）**

三辆车共享一套 **非线性模型预测控制器（NMPC）**，在此基础上实现：

1. 主车对给定局部路径的跟踪控制；
2. 从车在主车（视作队列 leader）的参考轨迹与自身姿态基础上，实现指定相对位姿（dx, dy, yaw）的编队跟随。

代码主要由以下几个部分组成：

- `MasterRosNode` + `server_node.cpp`：主车 ROS 节点与 ZMQ 服务器。
- `SlaveRosNode` + `client_node.cpp`：从车 1 ROS 节点与 ZMQ 客户端。
- `SlaveRosNode1` + `client_node1.cpp`：从车 2 ROS 节点与 ZMQ 客户端。
- `SimpleFormationNMPCController`：公共 NMPC 控制器实现。

---

2. 系统总体架构

2.1 节点与进程结构

整体上分为三类进程（可以部署在同一台或不同主机上）：

1. **主车进程**
   - ROS 节点名：`zmq_server`（在 `server_node.cpp` 中 `ros::init`）
   - 核心类：`MasterRosNode`（`master_ros_node.h`）
   - 同时包含一个 ZMQ server，用于接收 / 发送与从车相关的数据。
2. **从车 1 进程**
   - ROS 节点名：`zmq_client`。

    - 核心类：`SlaveRosNode`（`slave_ros_node.h`），封装从车 1 的 ROS 话题与控制逻辑。
3. **从车 2 进程**
    - ROS 节点名：`zmq_client1`。
    - 核心类：`SlaveRosNode1`（`slave_ros_node1.h`），封装从车 2 的 ROS 话题与控制逻辑。

三者之间的通信方式：

- **ROS Topic**：节点内部与仿真 / 底层控制板通讯（里程计、激光、速度指令等）。
- **ZMQ**：主车与各从车之间传输编队参考信息、本地路径和部分传感器数据。

2.2 系统框图与工作流程

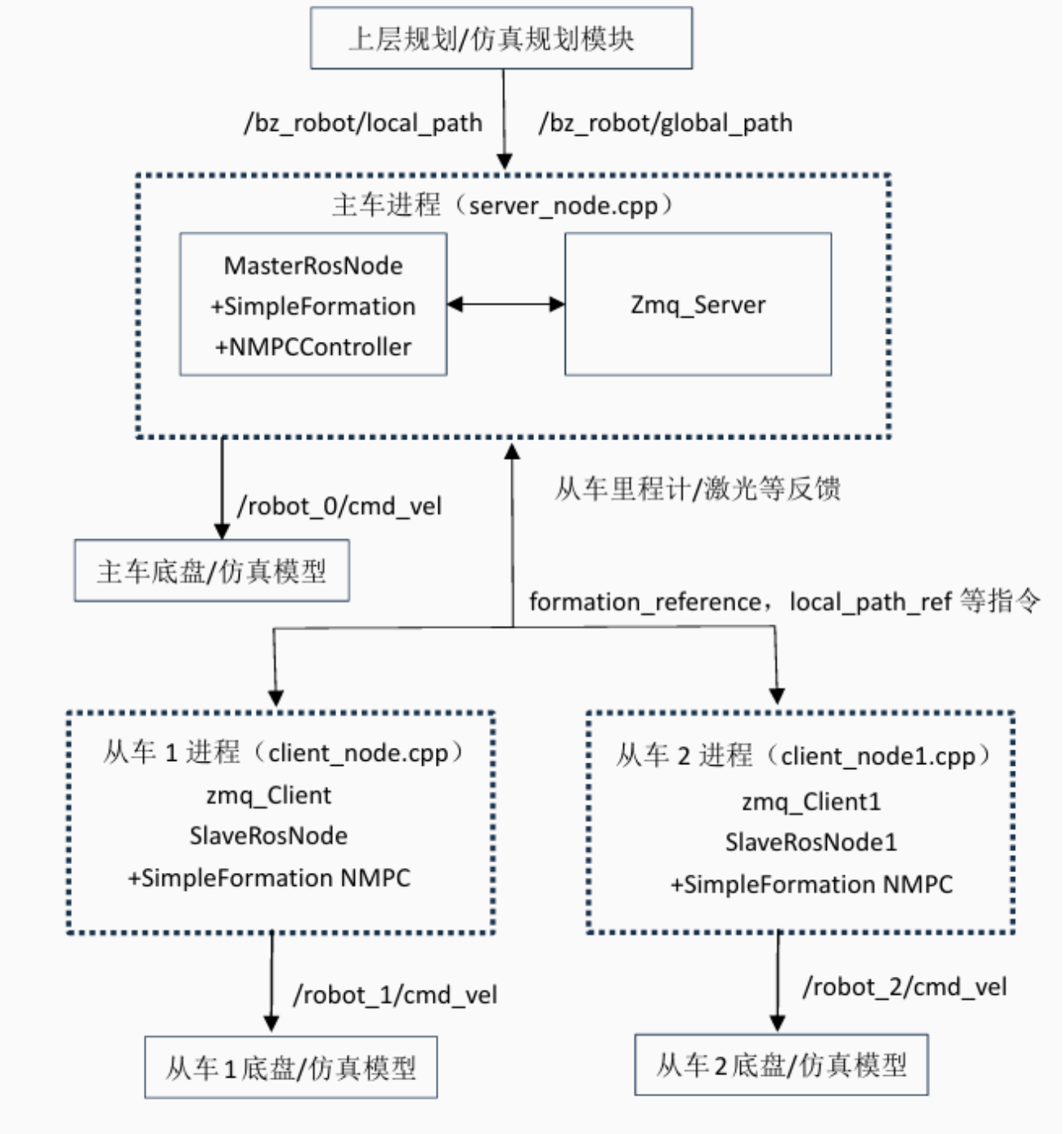

图 2-1 三车编队控制系统框图

---

3. 主车节点实现（Master）

3.1 主函数与 ZMQ 服务器

在 `server_node.cpp` 中，完成了主车进程的入口以及 ZMQ 服务器的初始化与回调注册：

- 初始化日志系统 `SimpleLogger`。
- 根据命令行参数或默认地址初始化 `ZmqServer`：

    - 默认监听：`tcp://*:5000`，并额外监听 `tcp://*:5001`（当前只监听，不支持 send）。
- 读取主车参数（轴距 `wheel_base`、轮距 `track_width`）来自 JSON 配置文件 `master_node.json`。
- 创建 `MasterRosNode` 实例，将 `ZmqServer` 指针和车辆参数传入。
- 在 ZMQ 侧，注册从车上传数据的回调，例如：
    - `"slave_motor_akm" -> HandleSlaveMotor`
    - `"slave_info" -> HandleSlaveInfo`
    - `"Slave_Odom_1/2" -> HandleSlaveOdom_1/2`
    - `"SlaveLaser_0/1" -> HandleSlaveLaser_0/1`
- 使用 `ros::SteadyTimer` 每 10ms 调用一次 `ZMQ_SERVER.Run()`，实现 ZMQ 收发调度。

这些回调会将从车上传的里程计、Info 等信息反序列化为 ROS 消息，再转交给 `MasterRosNode` 的接口（如 `SetSlaveOdom_1`）。

3.2 MasterRosNode 类结构

`MasterRosNode` 负责主车的所有 ROS 交互与控制逻辑，并持有一份 `SimpleFormationNMPCController` 作为主车的路径跟踪控制器：

3.2.1 订阅与发布

在构造函数中，初始化了以下订阅器与发布器：

- 主车里程计：
    - `/robot_0/odom`（`nav_msgs::Odometry`）→ `MasterOdomCallback`
        - 提取 x, y, yaw；做简单有效性检查；更新 `master_odom_` 标志。
- 路径信息：
    - `/bz_robot/global_path`（`nav_msgs::Path`）→ `GlobalPathCallback`（当前空实现，预留）
    - `/bz_robot/local_path`（`nav_msgs::Path`）→ `LocalPathCallback`
        - 将 Path 转换为 `vector<RefPoint>`（x, y, yaw, v），设置到主车的 MPC 引用路径中。
        - 同时将该 `ref` 序列化 + Base64，经 ZMQ 以 `"local_path_ref"` 命令广播给两个从车，使三车共享同一局部路径。

发布：

- `/robot_0/cmd_vel`（`geometry_msgs::Twist`）：主车速度与角速度控制量，由 `PublishTwist` 发布。

3.2.2 控制定时器与主车控制流程

`MasterRosNode` 在构造函数中创建了一个控制定时器：周期为 0.1s（10Hz），回调 `CycleControlCallback`。

控制流程为：

1. 检查是否已经收到 `master_odom_`，否则跳过本周期。
2. 调用 `computeMasterPathFollowing()`：
   - 从 `master_odom_` 中提取 `x, y, yaw`，构造当前姿态 `pose`。
   - 调用 `master_mpc_.computeControl(pose)` 得到 `[v, ω]`。
   - 将结果封装为 `geometry_msgs::Twist`。
3. 发布主车控制指令到 `/robot_0/cmd_vel`。
4. 调用 `sendFormationReference(master_twist)`，将主车当前的 twist 与 odom、期望间距等信息打包通过 ZMQ 发送给两辆从车。
5. 调用 `logMasterControlStatus` 和 `logMasterPosition` 做日志记录（包括位置、实际速度等）。

3.2.3 编队参考下发（formation_reference）

`sendFormationReference` 是三车编队中的关键接口：

- 构造 JSON：
  - `master_twist`：主车线速度和角速度（Twist 消息经序列化 + Base64 编码）。
  - `master_odom`：主车里程计（同上）。
  - `send_time_ms`：当前时间戳（ms）。
  - `desired_gap`：默认 1.5 m，期望车间距离（目前主要用于日志和配置参考）。
  - `formation_type`：当前固定为 0，表示链式编队。
- 经 ZMQ 分别发送给：
  - `"client-reliable"`（从车 1）
  - `"client-reliable1"`（从车 2）

在从车侧，这些信息会被用于生成 leader 未来轨迹、计算编队误差并输入 NMPC。

3.3 主车节点内部结构框图

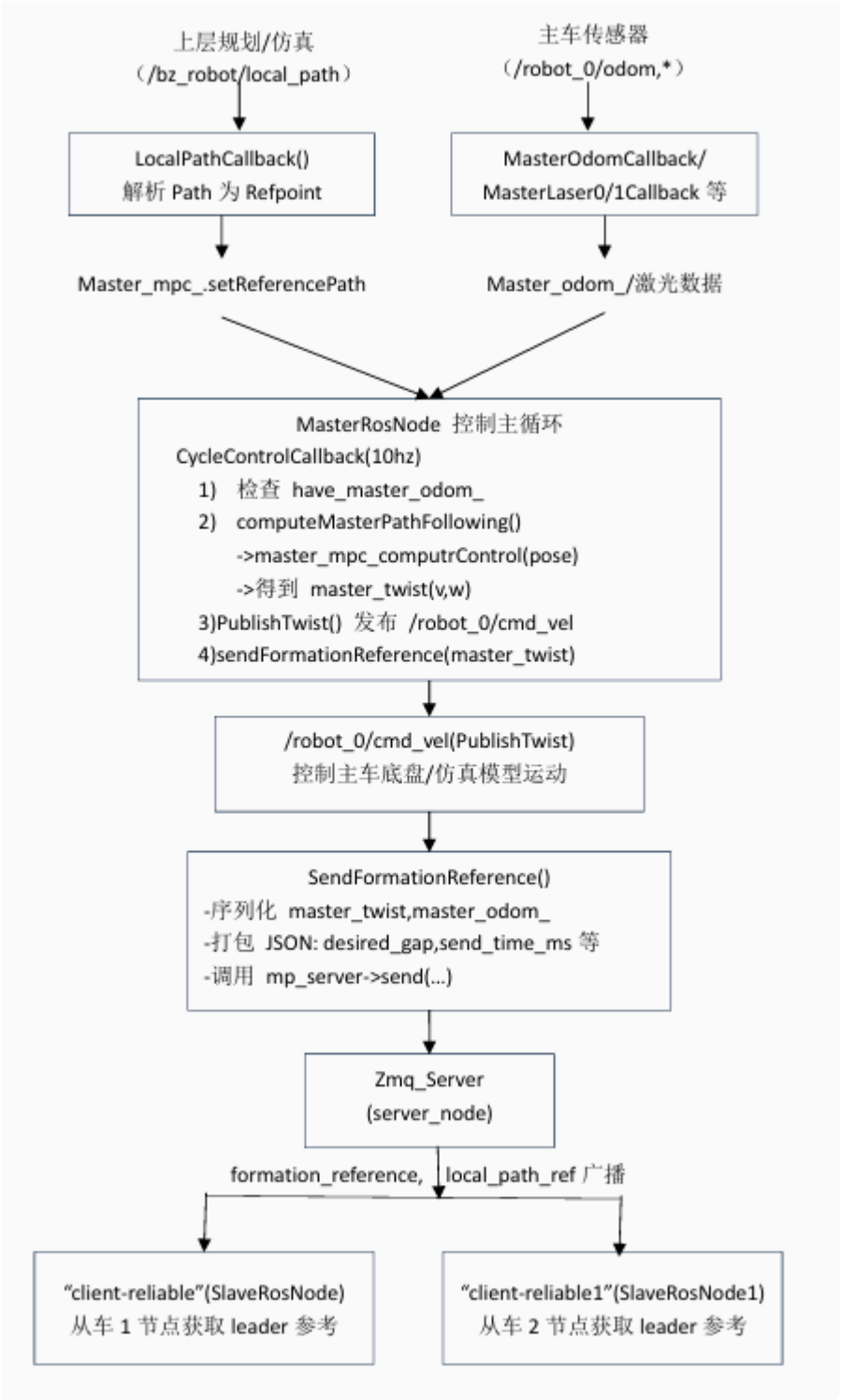

图 3-1 主车节点内部结构与数据流

---

4. 从车 1 节点实现（SlaveRosNode）

4.1 ZMQ 客户端与主函数

在 `client_node.cpp` 中，创建了一个 `ZmqClient` 实例 `ZMQ_CLIENT`，并将其与 `SlaveRosNode` 绑定：

- `ZMQ_CLIENT.Init("tcp://127.0.0.1:5000", "client-reliable")`：连接到主车的 ZMQ server。
- 注册 ZMQ 命令回调：
  - `"masterLaser0" -> HandleMasterLaser0`：接收主车激光 0（目前主要是计算延迟并缓存）。
  - `"masterLaser1" -> HandleMasterLaser1`：接收主车激光 1。
  - `"formation_reference" -> HandleFormationReference`：接收主车的编队参考信息。
  - `"local_path_ref" -> Receivelocalpath`：接收主车下发的参考路径。

上述回调会在处理完数据后调用 `SlaveRosNode` 的接口，例如 `SetFormationReference` 和 `Set_ref`。

4.2 从车 1 ROS 接口

在 `SlaveRosNode` 构造函数中，我初始化了若干订阅和发布接口：

订阅：

- 仿真环境中从车 1 的“真实”里程计与激光：
  - `/robot_1/odom` → `SlaveOdom_stage`（更新 `slave_odom_`，是编队控制使用的 odom）；
  - `/robot_1/base_scan_0`、`/robot_1/base_scan_1` → `SlaveLaser0Callback` / `SlaveLaser1Callback`（通过 ZMQ 发送回主车）。

发布：

- `/robot_1/cmd_vel`（`geometry_msgs::Twist`）：从车 1 实际控制量，最终由 `PublishTwist1` 发布。

4.3 编队信息接收与缓存

从车 1 通过 `SetFormationReference` 接收主车传来的编队参考信息：

- 存入 `formation_ref_` 结构，包括：
  - `master_twist`
  - `master_odom`
  - `desired_gap`
  - `formation_type`
  - `send_time_ms`
  - `received_time_ms`（本地接收时间）
- 计算通信延迟 `delay_ms` 并打印日志。

- 将 `have_formation_ref_` 标志设为 true。

在真正使用前，通过 `receiveFormationReference()` 完成一层安全检查：

- 若从未接收过，则使用默认值（v=0，ω=0，间距 1.5m）；
- 若数据时间超过 500 ms，则认为已过期，改用默认值；
- 这样保证在通信中断或延迟较大时，从车自动减速 / 停下。

4.4 从车 1 控制流程

`SlaveRosNode` 也有一个 0.1s 周期的控制定时器 `CycleControlCallbackSlave`：

完整流程：

1. 检查是否已经收到从车 odom（`have_slave_odom_`）。
2. 调用 `receiveFormationReference()`，获取最新的编队参考（可能是实际值或默认值）。
3. 调用 `SlaveCycleControlCallback(ref)` 计算控制量：
   - 从 `slave_odom_` 中获取从车当前 `(x, y, yaw, v)`。
   - 从编队参考中获取 leader（主车）的 `(x, y, yaw, v, ω)`。
   - 根据 MPC 设定的预测步数 `N = 10`、步长 `dt = 0.1`，对主车状态做简单前向积分，构造 leader 的预测轨迹 `leader_traj[N+1]`。
   - 调用 `slave_mpc_.setLeaderTrajectory(leader_traj)`。
   - 将当前从车姿态 `[x, y, yaw]` 输入 `slave_mpc_.computeControl`，得到速度 `[v, ω]`。
4. 得到的控制量封装为 `geometry_msgs::Twist`，通过 `PublishTwist1` 发布到 `/robot_1/cmd_vel`。
5. 定期通过 `logSlaveControlStatus` 输出从车控制状态。

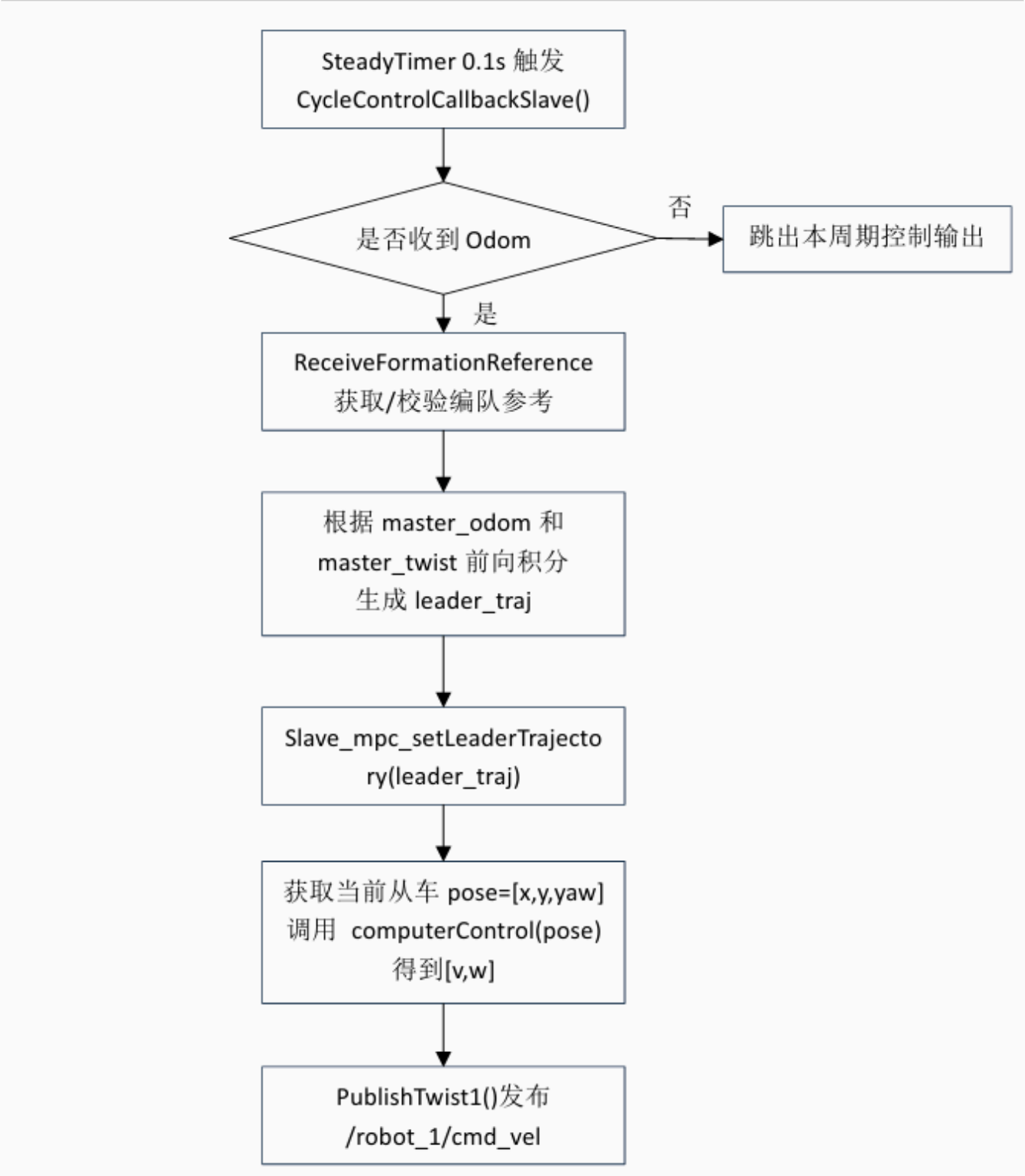

图 4-1 从车 1 控制流程框图

4.5 MPC 参数与编队配置（从车 1）

在 `SlaveInit()` 中我为从车 1 配置了 NMPC 参数和编队期望位姿：

- MPC 基本参数：
  - 预测步长 `horizon = 10`，周期 `dt = 0.1 s`；
  - 路径误差权重：`q_pos = 3.0`，`q_yaw = 1.5`；

    - 终端误差权重：`q_terminal = 8.0`；
    - 控制增量权重：`r_v = r_omega = 0.1`；
    - 速度限制：`max_v = 0.8 m/s`，`max_omega = 2.5 rad/s`。
- 编队配置（面向主车）：
    - `dx_des = -2.0`：期望从车在 leader 车体坐标系下，沿 x 轴落后 2 m；
    - `dy_des = 0.0`：保持在同一车道中心；
    - `dyaw_des = 0.0`：期望航向与 leader 相同；
    - 权重：`q_pos = 10.0`，`q_yaw = 5.0`。

这些参数通过 `slave_mpc_.setParameters`、`setLimits`、`setFormationConfig` 完成配置。

---

5. 从车 2 节点实现（SlaveRosNode1）

从车 2 整体结构与从车 1 非常类似，主要差异在：

1. 订阅和发布的话题使用不同的机器人命名空间 `/robot_2`。
2. 编队期望间距更大（落后距离为 4m）。
3. 在 ZMQ 层单独使用 `client-reliable1` 标识，与主车进行区分。

5.1 ZMQ 客户端与回调

在 `client_node1.cpp` 中，我初始化了另一个 `ZmqClient1`：

- 默认连接 `tcp://127.0.0.1:5000`，身份为 `"client-reliable1"`。
- 注册：
    - `"formation_reference" -> HandleFormationReference1`
    - `"local_path_ref" -> Receivelocalpath1`

5.2 ROS 接口与控制流程

`SlaveRosNode1` 构造函数中：

- 订阅：
    - `/robot_2/odom` → `SlaveOdom_stage`，获取从车 2 的实际姿态；
    - `/robot_2/base_scan_0`、`/robot_2/base_scan_1` → 预留回调 `SlaveLaserCallback_0/1`（当前为空实现）。
- 发布：
    - `/robot_2/cmd_vel` → `PublishTwist2`，对从车 2 施加控制。

控制定时器 `CycleControlCallbackSlave1` 与从车 1 类似：

1. 校验是否有 odom；
2. 通过 `receiveFormationReference()` 获取编队参考（带过期检查）；
3. 调用 `SlaveCycleControlCallback1`：
   - 根据 leader 的 `master_odom`、`master_twist` 构造 `leader_traj`；
   - 设置到 `slave1_mpc_`，并传入当前从车 2 姿态；
   - 求解得到 `[v, ω]`，封装为 `Twist` 发布。

5.3 MPC 与编队参数（从车 2）

在 `Slave1Init` 中配置从车 2 的 MPC：

- MPC 参数基本与从车 1 相同（horizon=10, dt=0.1, q_pos=3.0, q_yaw=1.5, q_terminal=8.0 等）。
- 编队配置区别：
  - `dx_des = -4.0`：从车 2 相对 leader 落后 4 m。
  - `dy_des = 0.0`，`dyaw_des = 0.0`，权重同从车 1。

因此，在同一个主车速度轨迹下，从车 2 总是预期比从车 1 更靠后，形成三车单列链式编队。

---

6. NMPC 控制器实现（公共部分）

三辆车都使用同一个类 `SimpleFormationNMPCController` 来计算控制量，只是配置不同：

6.1 状态与控制变量

- 车辆状态（简化单轨模型）：
  - `x, y, yaw`
- 控制量：
  - `v`：前向线速度
  - `omega`：角速度

6.2 参考信息

控制器内部维护三类参考：

1. **路径参考** `ref_path_`
   - 类型：`std::vector<RefPoint>`，其中 `RefPoint` 实际为 `jarvis_msgs::RefPoint`（包含 `x, y, yaw, v`）。

- 主车和从车都通过 `setReferencePath` 设置（来自 `LocalPathCallback` 或 ZMQ）。

2. **leader 轨迹** `leader_traj_`
   - 类型：`std::vector<LeaderPoint>`，包含 `(x, y, yaw, v, omega)`。
   - 仅从车使用，通过 `setLeaderTrajectory` 设置（由从车端根据主车当前状态前向积分得到）。
3. **编队配置** `formation_cfg_`
   - 期望相对位姿 `(dx_des, dy_des, dyaw_des)`；
   - 编队误差权重 `(q_pos, q_yaw)`。
   - `formation_enabled_` 标志由 `q_pos` 和 `q_yaw` 是否大于 0 决定，从而区分 master / slave 场景。

6.3 求解流程概要

`computeControl(current_pose)` 的主要步骤：

1. 若 `ref_path_` 长度小于 2，则直接输出零控制（停车）。
2. 根据当前位姿 `(x, y)` 在 `ref_path_` 中寻找最近点索引 `nearest_idx`。
3. 从最近点开始，取出 `horizon_ + 1` 个点构造 `horizon_ref_`（若不足则重复最后一点）。
4. 若编队启用，并且存在 `leader_traj_`，则取前 `horizon_ + 1` 个点构造 `leader_horizon_`，不足同样用最后一个点补齐。
5. 构造初始状态 `x0` = 当前 `(x, y, yaw)`。
6. 初始化控制序列 `u_seq[0..N-1]`（使用上一次解作为 warm-start，若无则全部置 0）。
7. 迭代若干次（`max_iterations_`），执行基于有限差分的梯度下降：
   - 对每个时刻 k 和每个控制分量 (v, omega)，通过 `J_plus` 和 `J_minus` 计算数值梯度；
   - 按一定步长 `step_size_` 更新 `u_seq`，并强制投影到速度约束区间内。
8. 控制序列求解完成后，取第一步控制 `u_seq[0]` 作为当前输出。

6.4 代价函数

总代价 `J` = 路径跟踪误差 + 控制量惩罚 + 终端误差 + （可选）编队误差：

- 路径误差（每步）：
  - 位置误差：`q_pos * (ex^2 + ey^2)`
  - 航向误差：`q_yaw * eyaw^2`
- 控制量惩罚：
  - `r_v * v^2 + r_omega * omega^2`

- 终端误差：
  - 对最后一个状态与 `horizon_ref_[N]` 之间的偏差施加更大的 `q_terminal`。
- 编队误差（仅在 formation_enabled_ 时生效）：
  - 对每一步，将当前车辆位置转换到 leader 车体坐标系，计算与期望 `(dx_des, dy_des, dyaw_des)` 的偏差；
  - 使用 `formation_cfg_.q_pos` 与 `q_yaw` 加权。

---

7. 关键参数与配置点汇总

1. **车辆参数（主车）**：
   - 从 `master_node.json` 中读取 `wheel_base`、`track_width`，用于构造 `MasterRosNode`。
2. **MPC 通用参数**（在各节点中通过 `setParameters` / `setLimits` 设置）：
   - 预测步长 `horizon`，周期 `dt`
   - 路径误差权重 `q_pos`，`q_yaw`，`q_terminal`
   - 控制权重 `r_v`，`r_omega`
   - 速度约束 `max_v`，`max_omega`
   - 优化器参数 `max_iterations`，`step_size`，`finite_diff_eps`（目前使用默认）
3. **编队配置**：
   - 从车 1：`dx_des = -2.0`，从车 2：`dx_des = -4.0`；两者 `dy_des = 0.0`，`dyaw_des = 0.0`。
4. **通信相关参数**：
   - ZMQ 地址（主车 Listen / 从车 Connect）。
   - 话题名：`/robot_0/1/2/odom`、`/robot_0/1/2/cmd_vel`、`/bz_robot/local_path` 等。
5. **超时与安全策略**：
   - 编队参考信息超过 500 ms 未更新，则从车使用默认参考（速度 0），避免盲目跟随。

---

8. 运行时序概览

大致时序可以总结为：

1. 上位规划模块发布 `/bz_robot/local_path`。主车接收到后：

- 将 Path 转换为 `RefPoint` 列表，设置到主车 MPC；
- 通过 ZMQ 下发同样的路径给两辆从车。

2. 主车周期性（10Hz）执行：
   - 读取自身 odom（`/robot_0/odom`）；
   - 调用 MPC 计算 `[v, ω]`，发布到 `/robot_0/cmd_vel`；
   - 打包当前 `Twist + Odom + desired_gap` 为 `formation_reference`，通过 ZMQ 发送给从车。
3. 从车 1 & 从车 2：
   - 通过 ZMQ 接收 `local_path_ref`，设置 MPC 路径参考；
   - 通过 ZMQ 接收 `formation_reference`，缓存 leader 状态和期望间距；
   - 本地从仿真 / 硬件获取各自 `/robot_1/2/odom`；
   - 周期性调用 MPC 计算自身控制量 `[v, ω]`，分别发布到 `/robot_1/cmd_vel` 与 `/robot_2/cmd_vel`。
4. 在通信正常情况下，三车沿同一局部路径行驶，保持指定纵向间距（2m、4m），形成稳定的链式编队。
