

# StandardScene 项目开发指导文档

## 目录

- [1. 项目概述](#)
- [2. 系统架构](#)
- [3. 核心模块详解](#)
- [4. 开发 workflow](#)
- [5. 常见任务指南](#)
- [6. 调试技巧](#)
- [7. 最佳实践](#)

## 项目概述

### 什么是 StandardScene?

StandardScene 是一个基于 `.NET Framework 4.8` 的**自动化仓储物流管理系统**，主要用于管理和控制**自动导向车 (AGV)** 的调度、任务分配、路径规划和智能执行。该系统支持多种车型，包括 VDA5050 标准车、Kiva 搬运车、叉车等。

### 核心功能

- 车辆管理**：管理多种类型的 AGV 车辆
- 路线规划**：基于地图的最优路径计算
- 任务调度**：链式搬运任务的智能分配与执行
- 车路协调**：通过交通控制防止车辆碰撞和死锁
- 充电管理**：车辆能源管理和充电策略
- 通信协议**：支持 MQTT、HTTP 等多种通信协议

### 项目技术栈

| 技术  | 说明                     |
|-----|------------------------|
| 语言  | C# 13.0                |
| 框架  | .NET Framework 4.8     |
| 通信  | MQTT (MQTTnet) / HTTP  |
| 序列化 | JSON (Newtonsoft.Json) |
| UI  | Windows Forms          |
| 编译器 | SimpleCore (自有编译引擎)    |

# 系统架构

## 分层结构



## 核心类关系图



# 核心模块详解

## 1. 车辆模块 (CarTypes)

### 1.1 VDA5050Car

VDA5050 标准是欧洲机器人技术和自动化协会 (VDMA) 制定的 AGV 通信标准。

关键特性：

- 基于 MQTT 协议通信
- 支持分段路径规划（基础段 + 地平线段）
- 支持实时状态反馈

关键属性：

```
// 基础长度：主控制发送给AGV的基础路由总长度
public float BaseLength = 5000;

// 地平线长度：主控制发送给AGV的地平线路由总长度
public float HorizonLength = 5000;

// 内部路由缓存
internal VDA5050SegmentsCache RouteCache = new();

// 最后确认的路由状态
private List<sequenceItem> _receivedLastConfirmedRoute = [];
```

执行流程：

```
1. keepAlive() - 定期检查并处理待发送命令
  ↓
2. ProcessCacheAndSendOrderMessage() - 管理路由缓存和订单
  ├── 检查接收到的状态消息
  ├── 更新路由缓存状态
  ├── 生成新订单（如果需要）
  └── 通过 MQTT 发布订单
  ↓
3. UpdateState() - 接收来自AGV的状态反馈
  └── 更新 _receivedLastConfirmedRoute

4. actualSendScript() - 执行脚本
  ├── 创建 VDA5050Interface
  ├── 在独立线程中执行
  └── 返回完成信号
```

路由状态机：

```
waiting (等待中)
↓
BaseSending (基础段发送)
↓
BaseAcknowledged (基础段已确认)
↓
HorizonSending (地平线段发送)
↓
HorizonAcknowledged (地平线段已确认)
↓
Executed (已执行)
```

## 1.2 BasicFields

定义车辆、工位、轨道的基础字段：

```
// 车辆字段
public class BasicCarFields
{
    public float MagSlowSpeed = 0;    // 磁导航减速速度
    public float MagFullSpeed = 0;    // 磁导航满速
}

// 工位字段
public class BasicSiteFields
{
    public bool shelf = false;    // 是否为货架
    public float CarLength = -1;    // 车长
    public float Carwidth = -1;    // 车宽
    public float CarCenterX = 0;    // 车中心X偏移
    public float CarCenterY = 0;    // 车中心Y偏移
    public int tag = -1;    // 标签ID
    public int TagValue = -1;    // 标签值（二维码/RFID等）
}

// 轨道字段
public class BasicTrackFields
{
    public int IOArea = -1;    // 输入输出区域
    public float Speed = 0.2f;    // 行驶速度
    public bool Reverse = false;    // 是否反向
    public int ReverseDst = -1;    // 反向目标
    public bool SwitchBarrier = false;    // 障碍物开关
    public float CarDirectionBias = 0;    // 方向偏差
    public bool EnableCarAbsoluteDirection = false;    // 启用绝对方向
    public float SlowDistance = -1;    // 减速距离
    public float StopDistance = -1;    // 停止距离
}
```

## 2. 任务调度模块 (Chained)

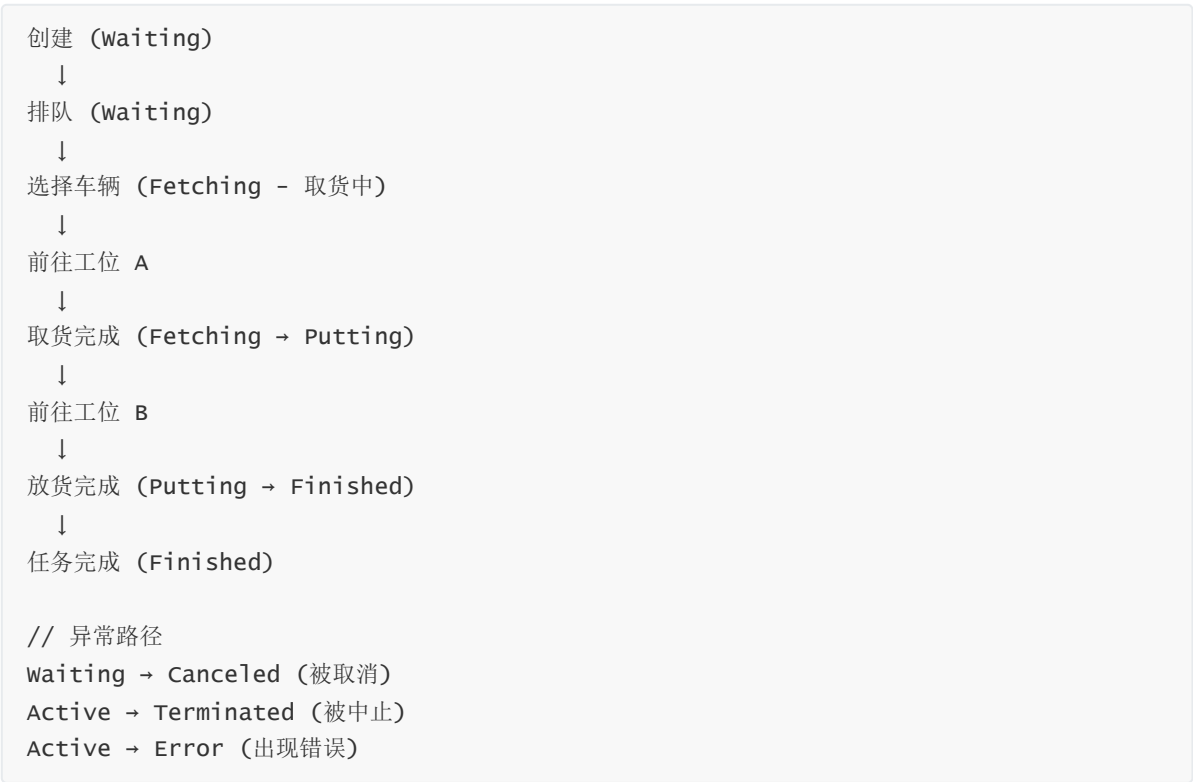
### 2.1 AbstractChainedDeliveryMission

链式搬运任务的核心管理类，负责多个搬运任务的智能调度。

关键概念：

| 概念       | 说明                             |
|----------|--------------------------------|
| Delivery | 单个搬运任务（从A工位取货到B工位放货）           |
| Chain    | 多个Delivery的链式执行（前序任务完成后执行后续任务） |
| Group    | 任务分组，用于车辆类型筛选                  |
| Priority | 任务优先级（数值越大优先级越高）               |

AbstractDelivery 生命周期：



关键方法：

```
// 加入任务队列
public void Enqueue(AbstractDelivery d)

// 执行单个任务的关键逻辑
private bool ExecutesSingle(AbstractDelivery d, AbstractDelivery[] deliveries, ...)

// 避障与推挤逻辑
(bool escaped, int type) EscapeOther(AbstractCar c, int hd, int toAvoid, ...)

// 获取所有任务
public List<AbstractDelivery> GetDeliveries(...)
```

```
// 启动调度线程
public override void Execute()

// 停止调度
public void Stop()
```

调度算法核心：

```
// 1. 将所有待处理任务排序（按优先级和预抓取标记）
foreach (var delivery in checking.OrderByDescending(p => p.priority +
(p.fetchPlan != null ? 1 : 0)))
{
    // 2. 尝试执行单个任务
    if (!ExecutesSingle(delivery, checking, missionField: missionField))
    {
        // 如果失败，记录为淤积任务
        delivery.stuckTime += 1;
        q.Add(delivery);
    }
}

// 3. 更新待处理队列
pendings = new List<AbstractDelivery>(q.Concat(eq).ToHashSet());
```

## 2.2 避障与推挤逻辑

当任务的取货点或放货点被其他车辆占用时，系统会尝试：

1. **推挤任务**：让占用车执行其他待处理任务
2. **避让点绕行**：让占用车前往最近的避让点
3. **路权重新规划**：调整车辆的行驶顺序

```
// 关键参数
public bool enablePushAwayMission = false;           // 启用推挤任务
public double pushAwayMissionTriggerDistanceThreshold = 10000; // 推挤距离阈值

public bool enableBlockStandby = false;              // 允许打断待命车
public bool enableRightOfWayGrabbing = false;        // 允许抢夺路权
public double rightOfWayTriggerDistanceThreshold = 999999; // 抢路权触发距离
public double rightOfWayGiveUpDistanceThreshold = 10000;  // 放弃路权距离
```

## 3. 路径规划模块 (SegmentPlan)

SegmentPlan 负责从源工位到目标工位的最优路径规划。

核心方法：

```
// 查找路由
public float FindRoute(Site src, Site dst)

// 编译成可执行程序
public CarProgram Compile(string name, ...)

// 附加其他规划
public CarProgram Append(SegmentPlan otherPlan, ...)
```

字段配置:

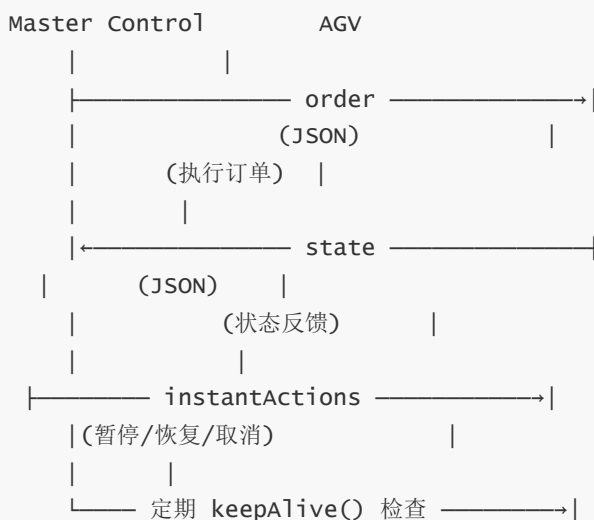
```
public Dictionary<string, string> fields = new()
{
    ["action"] = "fetch",          // 动作类型: fetch/put/move
    ["reverse"] = "false",         // 是否反向行驶
    ["allow_destination_on_route"] = "true", // 允许目标在路径上
    ["forbid_cross"] = "false",    // 禁止交叉
    ["CarLength"] = "-1",          // 车长
    ["Carwidth"] = "-1",          // 车宽
};
```

## 4. 通信模块 (MasterMQTTCommunication)

### 4.1 MQTT 话题结构

```
vda5050/fr1dAGV/
├─ order    ← 订单下发
├─ state    ← 状态反馈
├─ visualization    ← 可视化信息
├─ connection ← 连接状态
├─ instantActions ← 即时动作
├─ factsheet    ← 事实表
└─ carFields    ← 车辆字段
```

### 4.2 消息流



## 5. 通用工具模块 (Commons)

Commons 提供项目全局使用的工具方法。

关键方法分类：

| 分类   | 方法                   | 说明        |
|------|----------------------|-----------|
| 标签操作 | AddOrUpdateTag       | 添加或更新标签   |
|      | DeleteTag            | 删除标签      |
|      | ClearTags            | 清空标签      |
| 字段操作 | AddOrUpdateCarField  | 添加/更新车辆字段 |
|      | DeleteCarField       | 删除车辆字段    |
|      | AddOrUpdateSiteField | 添加/更新工位字段 |
| 车辆选择 | SelectCar            | 根据条件筛选车辆  |
|      | GetOnlineCars        | 获取在线车辆    |
| 路径规划 | GetNearestPlan       | 获取最近的可达路径 |
|      | GenerateEscapePlan   | 生成逃逸路径    |
| 任务分配 | NearestTask          | 最近距离任务分配  |
|      | GoSite               | 让车前往指定工位  |

标签 (Tag) 系统：

标签用于标记车辆的当前状态：

```
车.tags.Add("occupied", "DeliverFetch123");    // 车忙碌中
车.tags.Add("deliver", "123");                  // 运输任务ID
车.tags.Add("dest", "5");                       // 目标工位
车.tags.Add("redirect", "124");                  // 重定向任务ID
车.tags.Add("pendingput", "123");                // 等待放货
车.tags.Add("shouldcharge", "yes");              // 需要充电
车.tags.Add("online", "true");// 在线状态
```

字段 (Field) 系统：

字段用于存储灵活的配置数据：

```
工位.fields["standby"] = "true";                // 待命点
工位.fields["giveway"] = "true";                 // 避让点
工位.fields["codeArrive"] = "agv.wait();...";    // 到达时执行代码

车.fields["group"] = "A";                         // 车辆分组
车.fields["CarType"] = "Kiva";                    // 车辆类型
车.fields["Soc"] = "80";                          // 电池电量
```

# 开发 workflow

## 1. 环境搭建

### 前置要求

- Visual Studio 2022 或更高版本
- .NET Framework 4.8
- NuGet 包管理器

### 关键依赖包

```
<!-- MQTT 通信 -->
<package id="MQTTnet" version="3.1.2" />

<!-- JSON 序列化 -->
<package id="Newtonsoft.Json" version="13.0.1" />

<!-- 自有框架 -->
<package id="SimpleCore" version="*" />
<package id="SimpleComposer" version="*" />
<package id="LessokajiWeaverUtilities" version="*" />
```

## 2. 添加新车型

### 步骤:

1. 在 `CarTypes` 文件夹创建新文件 `YourCarType.cs`
2. 继承 `Car` 基类
3. 实现必要方法:

```
[CarType(Name = "您的车型")]
public class YourCarType : Car
{
    [FieldMember] public string conf = "";

    // 车的初始化
    public static async Task<YourCarType> Create()
    {
        return new YourCarType()
        {
            status = "Not Connected",
            address = "127.0.0.1",
            name = "YourCar",
            haveCoordination = true,
        };
    }

    // 保活方法
    public override void keepAlive()
    {
        // 定期执行的维护逻辑
    }
}
```

```

// 脚本执行方法
public override async Task actualSendScript(string script)
{
    try
    {
        // 解析和执行脚本
        SelfEvaluating(yourInterface, script);
    }
    catch (Exception ex)
    {
        throw;
    }
}

// 绘制方法
protected override void draw(Graphics eGraphics)
{
    // 绘制车辆图形
    eGraphics.FillRectangle(Brushes.Gray, -160, -120, 320, 240);
}
}

```

### 3. 添加新任务类型

步骤:

1. 在 Chained 文件夹创建新文件 YourMission.cs
2. 继承 AbstractChainedDeliveryMission
3. 实现必要方法:

```

[Mission(Name = "您的任务")]
public class YourMission : AbstractChainedDeliveryMission
{
    public class YourDelivery : AbstractDelivery
    {
        // 自定义字段
        public int customField = 0;
    }

    public override void Execute()
    {
        base.Execute();

        // 自定义逻辑
        var myThread = new Thread(() =>
        {
            while (status.active)
            {
                // 处理逻辑
                Thread.Sleep(1000);
            }
        });
        myThread.Start();
    }
}

```

```

    }

    public override void ChangePriority()
    {
        // 动态调整优先级
    }
}

```

## 4. 添加新的状态同步

监听车辆状态变化：

```

// 在 VDA5050Car 中
public override void keepAlive()
{
    if (!Listened)
    {
        SetupListeners(); // 设置监听器
        Listened = true;
    }

    lock(RouteCache)
    {
        ProcessCacheAndSendOrderMessage(); // 处理待发订单
    }
}

// 实现监听处理方法
public void UpdateState(StateMessage msg)
{
    lock(_receivedLastConfirmedRoute)
    {
        // 更新本地缓存
        _receivedOrderId = msg.orderId;
        _receivedLastNodeSequenceId = (int)msg.lastNodeSequenceId;
        _receivedLastConfirmedRoute = segments;
    }
}

```

## 常见任务指南

### 任务 1: 让车辆执行简单移动

```

// 获取车辆和工位
var car = SimpleLib.GetCar(carId) as Car;
var srcSite = SimpleLib.GetSite(srcSiteId);
var dstSite = SimpleLib.GetSite(dstSiteId);

// 创建路径规划
var plan = new SegmentPlan() { usingCar = car };
plan.FindRoute(srcSite, dstSite);

// 编译并执行

```

```
var program = plan.Compile("move");
var task = program.Queue();
await task;

// 更新车辆位置
car.siteID = dstSite.id;
```

## 任务 2: 创建和分配搬运任务

```
// 创建任务
var delivery = new YourDelivery()
{
    src = 1, // 取货工位ID
    dst = 5, // 放货工位ID
    priority = 10,
    group = "A",
    startCondition = () => DateTime.Now.Hour > 9, // 条件
};

// 添加回调
delivery.onStart = (d) => Console.WriteLine($"开始任务 {d.id}");
delivery.doneFetch = (d) => Console.WriteLine($"取货完成 {d.id}");
delivery.donePut = (d) => Console.WriteLine($"放货完成 {d.id}");
delivery.doneMission = (d) => Console.WriteLine($"任务完成 {d.id}");

// 加入队列
mission.Enqueue(delivery);
```

## 任务 3: 监听车辆状态

```
// 通过标签监听
car.tags.Add("Online", "true");
car.tags.TryGetValue("occupy", out var occupiedBy);

// 通过字段存储配置
Commons.AddOrUpdateCarField(car, "ChargeThreshold", "30");
Commons.AddOrUpdateCarField(car, "group", "A");

// 通过状态对象获取信息
var pendingLocks = car.status.pendingLocks; // 待锁定工位
var holdingLocks = car.status.holdingLocks; // 持有工位
var currentSite = car.GetLastSite(); // 当前工位ID
```

## 任务 4: 避障与避让

```
// 查找避让点
var givewayPlan = Commons.GetNearestPlan(
    car,
    site => site.fields.ContainsKey("giveway") // 搜索条件
);

if (givewayPlan != null)
{
```

```

    var program = giveawayPlan.Compile("giveaway");
    await program.Queue();
}

// 设置避让点字段
Commons.AddOrUpdateSiteField(site, "giveaway", "true");

```

## 任务 5: 实现充电逻辑

```

// 检查电量
if (Commons.CarValue(car, "Soc") < 30)
{
    // 标记需要充电
    Commons.AddOrUpdateTag(car.tags, "shouldcharge", "yes");

    // 前往充电站
    var chargeSite = SimpleLib.GetAllSites()
        .FirstOrDefault(s => s.name.Contains("charge"));

    if (chargeSite != null)
    {
        await Commons.GoSite(car, chargeSite, 0);
    }
}

```

## 调试技巧

### 1. 启用诊断日志

```

// 发送诊断日志
Diagnosis.Post("消息内容", "分类标签", isImportant: true);
Diagnosis.Log("详细日志", "script", true);

// 查看诊断输出窗口
G.pushStatus("状态栏消息");
AppendDebug("车辆调试信息");

```

### 2. 查看路由缓存

```

// VDA5050Car 中
[MethodMember(Name = "显示RouteCache")]
public void DisplayRouteCache()
{
    RouteCacheViewer = new TextViewer();
    RouteCacheViewer.Show();
}

// 缓存显示格式
// sequenceId | nodeId/edgeId | 类型 | 状态
// 0          | 1          | node | waiting
// 1          | A->B      | edge | BaseSending

```

### 3. 监听 MQTT 消息

```
// 在 VDA5050Car 中
[MethodMember(Name = "显示上报消息")]
public void DisplayLatestMessage()
{
    StateMessageViewer = new TextViewer();
    StateMessageViewer.Show();
}

// 查看消息内容：
// 节点状态、边状态、错误信息、动作状态
```

### 4. 打断点调试

关键打断点位置：

| 位置                                | 用途       |
|-----------------------------------|----------|
| ExecutesSingle()                  | 调试任务调度逻辑 |
| ProcessCacheAndSendOrderMessage() | 调试订单发送   |
| UpdateState()                     | 调试状态同步   |
| EscapeOther()                     | 调试避障逻辑   |
| FindRoute()                       | 调试路径规划   |

### 5. 输出调试信息示例

```
// 车辆调试
car.AppendDebug($"当前状态: {car.lstatus}");
car.AppendDebug($"当前工位: {car.siteID}");
car.AppendDebug($"持有工位: {string.Join(",", car.status.holdingLocks)}");
car.AppendDebug($"标签: {string.Join(",", car.tags.Select(t => $"{t.Key}={t.Value}"))}");

// 任务调试
Console.WriteLine($"任务 {d.id}: {d.src} -> {d.dst}, 优先级: {d.priority}");
Console.WriteLine($"状态: {d.GetStatus()}, 堵塞原因: {d.stuckReason}");

// 路径调试
Diagnosis.Log($"路径: {srcSite.id} -> {dstSite.id}, 距离: {distance}", "route", true);
Diagnosis.Post($"冲突车辆: {string.Join(",", conflictedCars.Select(c => c.id))}", "conflict", true);
```

# 最佳实践

---

## 1. 线程安全

使用 lock 保护共享资源：

```
// ? 正确做法
lock (RouteCache)
{
    if (RouteCache.Count > 0)
    {
        var lastSegment = RouteCache.Last();
    }
}

// ? 错误做法 - 竞态条件
if (RouteCache.Count > 0)
{
    var lastSegment = RouteCache.Last(); // 可能被其他线程修改
}
```

## 2. 异步任务处理

优先使用异步方法：

```
// ? 正确做法 - 异步
public override async Task actualSendScript(string script)
{
    var tcs = new TaskCompletionSource<int>();
    new Thread(() => {
        try
        {
            SelfEvaluating(currentAgv, script);
            tcs.SetResult(1);
        }
        catch (Exception ex)
        {
            tcs.SetException(ex);
        }
    }).Start();

    await tcs.Task;
}

// ? 不推荐 - 阻塞等待
public void SendScriptBlocking(string script)
{
    SelfEvaluating(currentAgv, script);
    Thread.Sleep(5000); // 阻塞主线程
}
```

### 3. 错误处理

使用 try-catch 和诊断日志:

```
try
{
    // 业务逻辑
    var plan = mplan.FindRoute(src, dst);
}
catch (BasicHeuristics.PlanningConflictException dce)
{
    // 特定异常处理
    G.pushStatus($"规划冲突, 冲突车辆: {string.Join(", ", dce.conflictedCars.Select(c => c.id))}");

    // 尝试恢复
    foreach (var conflictCar in dce.conflictedCars)
    {
        // 避障逻辑
    }
}
catch (Exception ex)
{
    // 通用异常处理
    Diagnosis.Log($"异常: {ExceptionFormatter.FormatEx(ex)}", "error", true);
    throw;
}
```

### 4. 配置管理

使用字段存储灵活配置:

```
// 定义配置字段
[FieldMember] public string conf = "";

// 解析配置
private T GetConf<T>(string name, T defVal)
{
    if (conf.Contains(name))
    {
        var values = conf.Split(',').Where(ss => ss.Contains(name)).ToList();
        if (values.Count > 0)
        {
            var value = values[0].Split(':')[1];
            return
            (T)TypeDescriptor.GetConverter(typeof(T)).ConvertFromString(value);
        }
    }
    return defVal;
}

// 使用
var timeout = GetConf("timeout", 5);
var port = GetConf("cport", 8008);
```

## 5. 代码组织

按功能分组代码：

```
public class VDA5050Car : Car
{
    // 1. 字段定义
    private string _connectionStatus = "Offline";
    private List<(string Content, DateTime SendTime)> _orderList = new();

    // 2. 属性
    public string ConnectionStatus { get; set; }

    // 3. 初始化方法
    public static async Task<VDA5050Car> Create() { ... }

    // 4. 核心方法
    public override void keepAlive() { ... }
    public override async Task actualSendScript(string script) { ... }

    // 5. 事件处理
    public void UpdateState(stateMessage msg) { ... }
    public void UpdatePosition(visualizationMessage msg) { ... }

    // 6. 辅助方法
    private void ProcessCacheAndSendOrderMessage() { ... }

    // 7. 工具方法
    private HttpClient2 GetHC() { ... }
}
```

## 6. 任务链式调用

正确的链式任务写法：

```
// ? 正确 - 利用前序任务完成回调
delivery.doneFetch = (d) =>
{
    // 取货完成后的逻辑
    d.putting = true;
};

// ? 正确 - 设置前序任务依赖
delivery2.former = delivery1;
delivery1.doneMission = (d) =>
{
    // delivery1 完成时自动触发 delivery2
};

// ? 错误 - 直接等待（可能死锁）
delivery1.doneMission = (d) =>
{
    while (!delivery2.IsFinished())
    {

```

```
Thread.Sleep(100); // 忙轮询，浪费资源
}
};
```

## 7. 注释和文档

编写清晰的注释：

```
/// <summary>
/// 路由状态存储在主控制中。
///
/// 此缓存跟踪 AGV 应执行的所有节点和边，并维护每个段的状态：
/// - waiting: 等待发送
/// - BaseSending: 作为基础段发送
/// - BaseAcknowledged: 基础段已确认
/// - HorizonSending: 作为地平线段发送
/// - HorizonAcknowledged: 地平线段已确认
/// - Executed: 已执行完毕
///
/// 当AGV报告完成某段时，此缓存会相应更新。
/// </summary>
internal VDA5050SegmentsCache RouteCache = new();

// 内联注释
if (_receivedOrderId == OrderId)
{
    // 如果是同一订单，从最后确认的节点开始
    _toTraverseSequenceId = (int)_receivedLastConfirmedRoute[0].sequenceId;
}
```

## 常见问题与解决方案

### Q1: 车辆被标记为 "Offline"，但实际上已连接

原因：MQTT 连接状态未同步

解决：

```
// 检查 MQTT 连接
var isConnected = _mqttCommunication._client.IsConnected;

// 手动重启连接
await _mqttCommunication.RestartClient();

// 重新订阅话题
SetupListeners();
```

### Q2: 任务堵塞在 "取货点有车阻拦"

原因：避障逻辑未能成功将阻拦车推开

解决：

```
// 启用推挤任务
mission.enablePushAwayMission = true;
mission.pushAwayMissionTriggerDistanceThreshold = 5000;

// 或启用避让点绕行
Commons.AddOrUpdateSiteField(site, "giveWay", "true");

// 检查是否有可用的避让点
var hasGiveWay = SimpleLib.GetAllSites().Any(s =>
    s.fields.Containskey("giveWay"));
```

## Q3: 路径规划失败 "no route found"

**原因：**地图连接问题或目标工位无法到达

**解决：**

```
// 检查地图连接
var sites = SimpleLib.GetAllSites();
var tracks = SimpleLib.GetAllTracks();
Console.WriteLine($"工位数: {sites.Count}, 轨道数: {tracks.Count}");

// 尝试双向规划
try
{
    plan.FindRoute(src, dst);
}
catch (Exception ex)
{
    Console.WriteLine($"A→B失败: {ex.Message}");
    // 尝试反向规划
    plan.FindRoute(dst, src);
}

// 启用允许目标在路径上选项
plan.fields["allow_destination_on_route"] = "true";
```

## Q4: 车辆脚本执行超时

**原因：**脚本执行时间过长或车辆无响应

**解决：**

```
// 增加超时时间
public int GetTimeout()
{
    var timeout = 5;
    if (conf.Contains("timeout"))
        timeout = Convert.ToInt32(
            conf.Substring(conf.IndexOf("timeout") + 8)
                .Split(', ')[0]
        );
    return timeout;
}
```

```
// 添加进度监控
async void MonitorScript()
{
    var startTime = DateTime.Now;
    while ((DateTime.Now - startTime).TotalSeconds < 60)
    {
        if (status.programs.now == null) break;
        if (status.programs.now.status.state == CarProgram.StatusEnum.Completed)
            break;

        Console.WriteLine($"执行中: {(DateTime.Now -
            startTime).TotalSeconds:0.0}s");
        await Task.Delay(1000);
    }
}
```

## Q5: 死锁检测和恢复

**原因：**多个车辆互相等待资源

**解决：**

```
// 启用死锁检测
TrafficControl.OnDeadLock = (loopingCars) =>
{
    var carIds = string.Join(",", loopingCars.Select(c => c.id));
    Diagnosis.Post($"检测到死锁: {carIds}", "deadlock", true);

    // 手动解除某个车的锁定
    var firstCar = loopingCars.First();
    firstCar.Reset(SimpleLib.GetSite(firstCar.GetLastSite()), true, false);
};

// 或使用 Escape 机制
var escapePlan = Commons.GenerateEscapePlan(
    currentPlan,
    site => site.fields.ContainsKey("giveWay")
);
```

## 参考资料

### 相关标准

- [VDA5050 标准规范](#)
- MQTTv3.1.1 协议规范

### 内部接口

- `SimpleCore`：核心库（路径规划、交通管制）
- `SimpleComposer`：任务编程框架
- `SimpleLib`：全局库访问接口

## 文件导航

| 文件                                        | 用途           |
|-------------------------------------------|--------------|
| Commons.cs                                | 全局工具方法       |
| CarTypes/VDA5050Car.cs                    | VDA5050 车型实现 |
| Chained/AbstractChainedDeliveryMission.cs | 链式任务调度       |
| Charge/StandardChargeMission.cs           | 充电任务         |
| InterLock/TrafficInterlockMission.cs      | 交通控制         |
| WebApi.cs                                 | HTTP 接口      |

## 后续学习

建议按以下顺序深入学习：

- 基础概念** → 理解 AGV、MQTT、路径规划
- 简单任务** → 实现单个搬运任务
- 高级特性** → 链式任务、避障、死锁恢复
- 优化调优** → 性能优化、算法改进
- 实战项目** → 自定义车型、自定义任务

文档版本：1.0

最后更新：2024年

维护者：StandardScene 开发团队