Shapefile转OpenDrive格式转换器V1.0

源代码

import numpy as np

from scipy.optimize import minimize

from scipy.interpolate import splprep, splev

from shapely.geometry import LineString, Point

from typing import List, Tuple, Dict, Optional, Any

import math

import logging

from scipy import interpolate

from scipy.optimize import minimize\_scalar

from abc import ABC, abstractmethod

logger = logging.getLogger(\_\_name\_\_)

class ConversionHandler(ABC):

def \_\_init\_\_(self):

self.\_next\_handler = None

def set\_next(self, handler: 'ConversionHandler') -> 'ConversionHandler':

self.\_next\_handler = handler

return handler

@abstractmethod

def handle(self, request: Dict[str, Any]) -> Dict[str, Any]:

pass

def \_handle\_next(self, request: Dict[str, Any]) -> Dict[str, Any]:

if self.\_next\_handler:

return self.\_next\_handler.handle(request)

return request

class GeometryConversionHandler(ConversionHandler):

def \_\_init\_\_(self, tolerance: float = 3.0, smooth\_curves: bool = True, preserve\_detail: bool = False,

curve\_fitting\_mode: str = "parampoly3", polynomial\_degree: int = 3, curve\_smoothness: float = 0.5,

coordinate\_precision: int = 3):

super().\_\_init\_\_()

self.geometry\_converter = GeometryConverter(tolerance, smooth\_curves, preserve\_detail,

curve\_fitting\_mode, polynomial\_degree, curve\_smoothness, coordinate\_precision)

def handle(self, request: Dict[str, Any]) -> Dict[str, Any]:

if request.get('stage') != 'geometry\_conversion':

return self.\_handle\_next(request)

roads\_data = request.get('roads\_data', [])

converted\_roads = []

for road\_data in roads\_data:

if road\_data.get('type') == 'lane\_based':

converted\_road = self.\_convert\_lane\_based\_geometry(road\_data)

else:

converted\_road = self.\_convert\_traditional\_geometry(road\_data)

if converted\_road:

converted\_roads.append(converted\_road)

request['converted\_roads'] = converted\_roads

request['stage'] = 'opendrive\_generation'

return self.\_handle\_next(request)

def \_convert\_lane\_based\_geometry(self, road\_data: Dict) -> Dict:

try:

lane\_surfaces = road\_data.get('lane\_surfaces', [])

if not lane\_surfaces:

return None

converted\_surfaces = self.geometry\_converter.convert\_lane\_surface\_geometry(lane\_surfaces)

if not converted\_surfaces:

return None

converted\_road = {

'id': road\_data['id'],

'type': 'lane\_based',

'lane\_surfaces': converted\_surfaces,

'attributes': road\_data.get('attributes', {})

}

return converted\_road

except Exception as e:

logger.error(f"Lane格式道路 {road\_data.get('id', 'unknown')} 几何转换失败: {e}")

return None

def \_convert\_traditional\_geometry(self, road\_data: Dict) -> Dict:

try:

coordinates = road\_data['coordinates']

segments = self.geometry\_converter.convert\_road\_geometry(coordinates)

if not segments:

return None

total\_length = self.geometry\_converter.calculate\_road\_length(segments)

converted\_road = {

'id': road\_data['id'],

'type': 'traditional',

'segments': segments,

'attributes': road\_data['attributes'],

'total\_length': total\_length

}

return converted\_road

except Exception as e:

logger.error(f"传统格式道路 {road\_data.get('id', 'unknown')} 几何转换失败: {e}")

return None

class GeometryConverter:

def \_\_init\_\_(self, tolerance: float = 3.0, smooth\_curves: bool = True, preserve\_detail: bool = False,

curve\_fitting\_mode: str = "parampoly3", polynomial\_degree: int = 3, curve\_smoothness: float = 0.5,

coordinate\_precision: int = 3):

self.tolerance = tolerance

self.smooth\_curves = smooth\_curves

self.preserve\_detail = preserve\_detail

self.curve\_fitting\_mode = curve\_fitting\_mode

self.polynomial\_degree = max(2, min(5, polynomial\_degree))

self.curve\_smoothness = max(0.0, min(1.0, curve\_smoothness))

self.coordinate\_precision = max(1, min(10, coordinate\_precision))

if preserve\_detail:

self.effective\_tolerance = tolerance \* 0.8

else:

self.effective\_tolerance = tolerance \* 1.5

self.road\_segments = []

self.max\_segments\_per\_road = 50

logger.info(f"几何转换器初始化，容差: {tolerance}m, 有效容差: {self.effective\_tolerance}m, 平滑曲线: {smooth\_curves}, 保留细节: {preserve\_detail}")

def convert\_road\_geometry(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

if len(coordinates) < 2:

logger.warning("坐标点数量不足，无法转换")

return []

if self.curve\_fitting\_mode == "polyline":

if len(coordinates) > 50:

logger.debug(f"检测到高密度坐标（{len(coordinates)}个点），使用保形转换")

return self.\_fit\_adaptive\_line\_segments(coordinates)

elif self.smooth\_curves and len(coordinates) >= 3:

return self.fit\_smooth\_curve\_segments(coordinates)

else:

return self.fit\_line\_segments(coordinates)

elif self.curve\_fitting\_mode == "polynomial":

logger.debug(f"使用多项式曲线拟合，阶数: {self.polynomial\_degree}, 平滑度: {self.curve\_smoothness}")

return self.\_fit\_polynomial\_curves(coordinates)

elif self.curve\_fitting\_mode == "spline":

logger.debug(f"使用样条曲线拟合，平滑度: {self.curve\_smoothness}")

return self.\_fit\_spline\_curves(coordinates)

elif self.curve\_fitting\_mode == "parampoly3":

logger.debug(f"使用ParamPoly3曲线拟合，阶数: {self.polynomial\_degree}, 平滑度: {self.curve\_smoothness}")

return self.\_fit\_polynomial\_curves(coordinates)

else:

logger.warning(f"未知的曲线拟合模式: {self.curve\_fitting\_mode}，使用默认折线拟合")

return self.fit\_line\_segments(coordinates)

def fit\_smooth\_curve\_segments(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

if len(coordinates) < 3:

return self.fit\_line\_segments(coordinates)

segments = []

current\_s = 0.0

if self.preserve\_detail:

simplified\_coords = self.\_adaptive\_simplify(coordinates)

else:

simplified\_coords = self.\_douglas\_peucker(coordinates, self.effective\_tolerance)

if self.smooth\_curves and len(simplified\_coords) >= 4:

smooth\_coords = self.\_spline\_interpolation(simplified\_coords)

segments = self.\_fit\_curve\_segments\_from\_smooth(smooth\_coords, current\_s)

else:

segments = self.\_fit\_adaptive\_line\_segments(simplified\_coords, current\_s)

return segments

def \_calculate\_arc\_lengths(self, coordinates: List[Tuple[float, float]]) -> np.ndarray:

arc\_lengths = np.zeros(len(coordinates))

for i in range(1, len(coordinates)):

dx = coordinates[i][0] - coordinates[i-1][0]

dy = coordinates[i][1] - coordinates[i-1][1]

arc\_lengths[i] = arc\_lengths[i-1] + np.sqrt(dx\*dx + dy\*dy)

return arc\_lengths

def \_calculate\_precise\_heading(self, coordinates: List[Tuple[float, float]]) -> float:

if len(coordinates) < 2:

return 0.0

if len(coordinates) == 2:

dx = coordinates[1][0] - coordinates[0][0]

dy = coordinates[1][1] - coordinates[0][1]

return math.atan2(dy, dx)

total\_dx = 0.0

total\_dy = 0.0

weight\_sum = 0.0

for i in range(1, min(len(coordinates), 4)):

dx = coordinates[i][0] - coordinates[0][0]

dy = coordinates[i][1] - coordinates[0][1]

distance = np.sqrt(dx\*dx + dy\*dy)

if distance > 1e-6:

weight = 1.0 / (i \* i)

total\_dx += dx \* weight

total\_dy += dy \* weight

weight\_sum += weight

if weight\_sum > 0:

avg\_dx = total\_dx / weight\_sum

avg\_dy = total\_dy / weight\_sum

return math.atan2(avg\_dy, avg\_dx)

return 0.0

def \_select\_optimal\_polynomial\_degree(self, t\_params: np.ndarray,

local\_u: np.ndarray,

local\_v: np.ndarray) -> int:

max\_degree = min(self.polynomial\_degree, len(t\_params) - 1, 3)

min\_degree = 1

if len(t\_params) <= 3:

return min\_degree

best\_degree = min\_degree

best\_score = float('inf')

for degree in range(min\_degree, max\_degree + 1):

try:

# 计算拟合误差

poly\_u = np.polyfit(t\_params, local\_u, degree)

poly\_v = np.polyfit(t\_params, local\_v, degree)

# 计算残差

u\_fitted = np.polyval(poly\_u, t\_params)

v\_fitted = np.polyval(poly\_v, t\_params)

u\_residuals = local\_u - u\_fitted

v\_residuals = local\_v - v\_fitted

# 计算均方根误差

rmse = np.sqrt(np.mean(u\_residuals\*\*2 + v\_residuals\*\*2))

# 添加复杂度惩罚（避免过拟合）

complexity\_penalty = degree \* 0.01

score = rmse + complexity\_penalty

if score < best\_score:

best\_score = score

best\_degree = degree

except Exception:

continue

return best\_degree

def \_calculate\_fitting\_weights(self, num\_points: int) -> np.ndarray:

weights = np.ones(num\_points)

weights[0] = 2.0

weights[-1] = 2.0

if num\_points > 4:

weights[1] = 1.5

weights[-2] = 1.5

return weights

def \_evaluate\_fitting\_quality(self, t\_params: np.ndarray,

local\_u: np.ndarray,

local\_v: np.ndarray,

poly\_u: np.ndarray,

poly\_v: np.ndarray) -> float:

try:

u\_fitted = np.polyval(poly\_u, t\_params)

v\_fitted = np.polyval(poly\_v, t\_params)

u\_residuals = local\_u - u\_fitted

v\_residuals = local\_v - v\_fitted

max\_error = np.max(np.sqrt(u\_residuals\*\*2 + v\_residuals\*\*2))

rmse = np.sqrt(np.mean(u\_residuals\*\*2 + v\_residuals\*\*2))

return 0.7 \* max\_error + 0.3 \* rmse

except Exception:

return float('inf')

def \_optimize\_boundary\_conditions(self, local\_u: np.ndarray, local\_v: np.ndarray,

au: float, bu: float, cu: float, du: float,

av: float, bv: float, cv: float, dv: float,

degree: int) -> Tuple[float, float, float, float, float, float, float, float]:

au = 0.0

av = 0.0

end\_u = local\_u[-1]

end\_v = local\_v[-1]

if degree == 1:

bu = end\_u

bv = end\_v

cu = du = cv = dv = 0.0

else:

current\_sum\_u = bu + cu + du

current\_sum\_v = bv + cv + dv

if abs(current\_sum\_u) < 1e-10:

bu = end\_u

else:

scale\_u = end\_u / current\_sum\_u

bu \*= scale\_u

cu \*= scale\_u

du \*= scale\_u

if abs(current\_sum\_v) < 1e-10:

bv = end\_v

else:

scale\_v = end\_v / current\_sum\_v

bv \*= scale\_v

cv \*= scale\_v

dv \*= scale\_v

return au, bu, cu, du, av, bv, cv, dv

def \_fit\_segmented\_polynomial\_curves(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

if len(coordinates) < 6:

return self.\_fit\_polynomial\_curves(coordinates)

segments = []

current\_s = 0.0

curvature\_changes = self.\_detect\_curvature\_changes(coordinates)

segment\_points = self.\_determine\_segment\_points(coordinates, curvature\_changes)

for i in range(len(segment\_points) - 1):

start\_idx = segment\_points[i]

end\_idx = segment\_points[i + 1] + 1 # 包含端点

segment\_coords = coordinates[start\_idx:end\_idx]

if len(segment\_coords) >= 3:

segment\_geometries = self.\_fit\_polynomial\_curves(segment\_coords)

for geom in segment\_geometries:

geom['s'] = current\_s

current\_s += geom['length']

segments.append(geom)

else:

line\_segments = self.fit\_line\_segments(segment\_coords)

for geom in line\_segments:

geom['s'] = current\_s

current\_s += geom['length']

segments.append(geom)

logger.debug(f"分段拟合完成，总段数: {len(segments)}, 原始点数: {len(coordinates)}")

return segments

def \_detect\_curvature\_changes(self, coordinates: List[Tuple[float, float]]) -> List[int]:

if len(coordinates) < 5:

return []

curvatures = []

for i in range(2, len(coordinates) - 2):

p1 = coordinates[i-2]

p2 = coordinates[i-1]

p3 = coordinates[i]

p4 = coordinates[i+1]

p5 = coordinates[i+2]

curvature = self.\_calculate\_point\_curvature([p1, p2, p3, p4, p5])

curvatures.append(curvature)

change\_points = []

threshold = np.std(curvatures) \* 1.5 # 动态阈值

for i in range(1, len(curvatures) - 1):

if abs(curvatures[i] - curvatures[i-1]) > threshold:

change\_points.append(i + 2) # 调整索引

return change\_points

def \_calculate\_point\_curvature(self, points: List[Tuple[float, float]]) -> float:

if len(points) < 3:

return 0.0

p1, p2, p3 = points[0], points[2], points[4]

v1 = (p2[0] - p1[0], p2[1] - p1[1])

v2 = (p3[0] - p2[0], p3[1] - p2[1])

cross = v1[0] \* v2[1] - v1[1] \* v2[0]

norm1 = np.sqrt(v1[0]\*\*2 + v1[1]\*\*2)

norm2 = np.sqrt(v2[0]\*\*2 + v2[1]\*\*2)

if norm1 \* norm2 > 1e-10:

return abs(cross) / (norm1 \* norm2)

return 0.0

def \_determine\_segment\_points(self, coordinates: List[Tuple[float, float]],

curvature\_changes: List[int]) -> List[int]:

segment\_points = [0]

max\_segment\_length = 15

min\_segment\_length = 5

current\_start = 0

for change\_point in curvature\_changes:

if (change\_point - current\_start >= min\_segment\_length and

change\_point - current\_start <= max\_segment\_length):

segment\_points.append(change\_point)

current\_start = change\_point

elif change\_point - current\_start > max\_segment\_length:

forced\_point = current\_start + max\_segment\_length

segment\_points.append(forced\_point)

current\_start = forced\_point

remaining\_length = len(coordinates) - 1 - current\_start

if remaining\_length > max\_segment\_length:

while current\_start + max\_segment\_length < len(coordinates) - 1:

current\_start += max\_segment\_length

segment\_points.append(current\_start)

if segment\_points[-1] != len(coordinates) - 1:

segment\_points.append(len(coordinates) - 1)

return segment\_points

def \_adaptive\_simplify(self, coordinates: List[Tuple[float, float]]) -> List[Tuple[float, float]]:

if len(coordinates) <= 3:

return coordinates

result = [coordinates[0]]

for i in range(1, len(coordinates) - 1):

curvature = self.\_calculate\_curvature(coordinates[i-1], coordinates[i], coordinates[i+1])

adaptive\_tolerance = self.effective\_tolerance

if curvature > 0.2:

adaptive\_tolerance \*= 0.7

elif curvature < 0.05:

adaptive\_tolerance \*= 4.0

else:

adaptive\_tolerance \*= 2.0

if len(result) >= 2:

distance = self.\_point\_to\_line\_distance(coordinates[i], result[-2], coordinates[-1])

if distance > adaptive\_tolerance:

result.append(coordinates[i])

else:

result.append(coordinates[i])

result.append(coordinates[-1])

return result

def \_calculate\_curvature(self, p1: Tuple[float, float], p2: Tuple[float, float], p3: Tuple[float, float]) -> float:

v1 = (p2[0] - p1[0], p2[1] - p1[1])

v2 = (p3[0] - p2[0], p3[1] - p2[1])

len1 = math.sqrt(v1[0]\*\*2 + v1[1]\*\*2)

len2 = math.sqrt(v2[0]\*\*2 + v2[1]\*\*2)

if len1 == 0 or len2 == 0:

return 0.0

dot\_product = v1[0] \* v2[0] + v1[1] \* v2[1]

cross\_product = v1[0] \* v2[1] - v1[1] \* v2[0]

angle = math.atan2(abs(cross\_product), dot\_product)

avg\_length = (len1 + len2) / 2

return angle / avg\_length if avg\_length > 0 else 0.0

def \_spline\_interpolation(self, coordinates: List[Tuple[float, float]], num\_points: int = None) -> List[Tuple[float, float]]:

if len(coordinates) < 4:

return coordinates

try:

x\_coords = [p[0] for p in coordinates]

y\_coords = [p[1] for p in coordinates]

distances = [0]

for i in range(1, len(coordinates)):

dist = math.sqrt((x\_coords[i] - x\_coords[i-1])\*\*2 + (y\_coords[i] - y\_coords[i-1])\*\*2)

distances.append(distances[-1] + dist)

if num\_points is None:

num\_points = len(coordinates) \* 2

t\_new = np.linspace(0, distances[-1], num\_points)

if len(coordinates) >= 4:

spline\_x = interpolate.interp1d(distances, x\_coords, kind='cubic', bounds\_error=False, fill\_value='extrapolate')

spline\_y = interpolate.interp1d(distances, y\_coords, kind='cubic', bounds\_error=False, fill\_value='extrapolate')

else:

spline\_x = interpolate.interp1d(distances, x\_coords, kind='linear')

spline\_y = interpolate.interp1d(distances, y\_coords, kind='linear')

x\_smooth = spline\_x(t\_new)

y\_smooth = spline\_y(t\_new)

return list(zip(x\_smooth, y\_smooth))

except Exception as e:

logger.warning(f"样条插值失败，使用原始坐标: {e}")

return coordinates

def \_fit\_curve\_segments\_from\_smooth(self, smooth\_coords: List[Tuple[float, float]], start\_s: float = 0.0) -> List[Dict]:

segments = []

current\_s = start\_s

i = 0

while i < len(smooth\_coords) - 1:

curve\_end = self.\_detect\_smooth\_curve\_segment(smooth\_coords, i)

if curve\_end > i + 2: # 找到曲线段

curve\_coords = smooth\_coords[i:curve\_end + 1]

arc\_segment = self.\_fit\_smooth\_arc(curve\_coords, current\_s)

if arc\_segment:

segments.append(arc\_segment)

current\_s += arc\_segment['length']

i = curve\_end

else:

line\_segment = self.\_create\_line\_segment(smooth\_coords[i], smooth\_coords[i + 1], current\_s, i == 0)

segments.append(line\_segment)

current\_s += line\_segment['length']

i += 1

else:

line\_segment = self.\_create\_line\_segment(smooth\_coords[i], smooth\_coords[i + 1], current\_s, i == 0)

segments.append(line\_segment)

current\_s += line\_segment['length']

i += 1

return segments

def \_detect\_smooth\_curve\_segment(self, coordinates: List[Tuple[float, float]], start\_idx: int) -> int:

if start\_idx >= len(coordinates) - 2:

return start\_idx + 1

curve\_threshold = 0.05 # 曲率阈值

min\_curve\_points = 3

max\_curve\_points = min(20, len(coordinates) - start\_idx)

curve\_points = 0

for i in range(start\_idx + 1, min(start\_idx + max\_curve\_points, len(coordinates) - 1)):

if i + 1 < len(coordinates):

curvature = self.\_calculate\_curvature(coordinates[i-1], coordinates[i], coordinates[i+1])

if curvature > curve\_threshold:

curve\_points += 1

else:

if curve\_points >= min\_curve\_points:

return i

curve\_points = 0

if curve\_points >= min\_curve\_points:

return min(start\_idx + max\_curve\_points - 1, len(coordinates) - 1)

return start\_idx + 1

def \_fit\_smooth\_arc(self, coordinates: List[Tuple[float, float]], start\_s: float) -> Optional[Dict]:

if len(coordinates) < 3:

return None

try:

center, radius = self.\_fit\_circle(coordinates)

if radius < 10 or radius > 10000:

return None

start\_point = coordinates[0]

end\_point = coordinates[-1]

start\_angle = math.atan2(start\_point[1] - center[1], start\_point[0] - center[0])

end\_angle = math.atan2(end\_point[1] - center[1], end\_point[0] - center[0])

angle\_diff = end\_angle - start\_angle

while angle\_diff > math.pi:

angle\_diff -= 2 \* math.pi

while angle\_diff < -math.pi:

angle\_diff += 2 \* math.pi

arc\_length = abs(angle\_diff) \* radius

curvature = angle\_diff / arc\_length if arc\_length > 0 else 0

dx = coordinates[1][0] - coordinates[0][0]

dy = coordinates[1][1] - coordinates[0][1]

heading = math.atan2(dy, dx)

segment = {

'type': 'arc',

's': start\_s,

'hdg': heading,

'length': arc\_length,

'curvature': curvature

}

if start\_s == 0:

segment['x'] = start\_point[0]

segment['y'] = start\_point[1]

return segment

except Exception as e:

logger.debug(f"圆弧拟合失败: {e}")

return None

def \_fit\_adaptive\_line\_segments(self, coordinates: List[Tuple[float, float]], start\_s: float = 0.0) -> List[Dict]:

segments = []

current\_s = start\_s

for i in range(len(coordinates) - 1):

segment = self.\_create\_line\_segment(coordinates[i], coordinates[i + 1], current\_s, i == 0)

segments.append(segment)

current\_s += segment['length']

return segments

def \_create\_line\_segment(self, start\_point: Tuple[float, float], end\_point: Tuple[float, float],

s\_coord: float, include\_xy: bool = False) -> Dict:

dx = end\_point[0] - start\_point[0]

dy = end\_point[1] - start\_point[1]

length = math.sqrt(dx\*\*2 + dy\*\*2)

heading = math.atan2(dy, dx)

segment = {

'type': 'line',

's': s\_coord,

'hdg': heading,

'length': length

}

if include\_xy:

segment['x'] = start\_point[0]

segment['y'] = start\_point[1]

return segment

def fit\_line\_segments(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

segments = []

current\_s = 0.0

simplified\_coords = self.\_douglas\_peucker(coordinates, self.tolerance)

if len(simplified\_coords) > self.max\_segments\_per\_road:

logger.warning(f"简化后仍有{len(simplified\_coords)}个点，超过限制{self.max\_segments\_per\_road}，进一步简化")

simplified\_coords = self.\_limit\_segments(simplified\_coords, self.max\_segments\_per\_road)

for i in range(len(simplified\_coords) - 1):

start = simplified\_coords[i]

end = simplified\_coords[i + 1]

dx = end[0] - start[0]

dy = end[1] - start[1]

length = math.sqrt(dx\*\*2 + dy\*\*2)

heading = math.atan2(dy, dx)

segment = {

'type': 'line',

's': current\_s,

'hdg': heading,

'length': length

}

if len(segments) == 0:

segment['x'] = start[0]

segment['y'] = start[1]

segments.append(segment)

current\_s += length

return segments

def fit\_arc\_segments(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

segments = []

current\_s = 0.0

i = 0

while i < len(coordinates) - 1:

curve\_end = self.\_detect\_curve\_segment(coordinates, i)

if curve\_end > i + 1: # 找到弯曲段

curve\_coords = coordinates[i:curve\_end + 1]

arc\_segment = self.\_fit\_single\_arc(curve\_coords, current\_s)

if arc\_segment:

segments.append(arc\_segment)

current\_s += arc\_segment['length']

i = curve\_end

else: # 直线段

start = coordinates[i]

end = coordinates[i + 1]

dx = end[0] - start[0]

dy = end[1] - start[1]

length = math.sqrt(dx\*\*2 + dy\*\*2)

heading = math.atan2(dy, dx)

segment = {

'type': 'line',

's': current\_s,

'hdg': heading,

'length': length

}

if len(segments) == 0:

segment['x'] = start[0]

segment['y'] = start[1]

segments.append(segment)

current\_s += length

i += 1

return segments

def \_douglas\_peucker(self, coordinates: List[Tuple[float, float]], tolerance: float) -> List[Tuple[float, float]]:

if len(coordinates) <= 2:

return coordinates

start = coordinates[0]

end = coordinates[-1]

max\_distance = 0

max\_index = 0

for i in range(1, len(coordinates) - 1):

distance = self.\_point\_to\_line\_distance(coordinates[i], start, end)

if distance > max\_distance:

max\_distance = distance

max\_index = i

if max\_distance < tolerance:

return [start, end]

left\_part = self.\_douglas\_peucker(coordinates[:max\_index + 1], tolerance)

right\_part = self.\_douglas\_peucker(coordinates[max\_index:], tolerance)

return left\_part[:-1] + right\_part

def \_point\_to\_line\_distance(self, point: Tuple[float, float],

line\_start: Tuple[float, float],

line\_end: Tuple[float, float]) -> float:

x0, y0 = point

x1, y1 = line\_start

x2, y2 = line\_end

line\_length = math.sqrt((x2 - x1)\*\*2 + (y2 - y1)\*\*2)

if line\_length == 0:

return math.sqrt((x0 - x1)\*\*2 + (y0 - y1)\*\*2)

distance = abs((y2 - y1) \* x0 - (x2 - x1) \* y0 + x2 \* y1 - y2 \* x1) / line\_length

return distance

def \_limit\_segments(self, coordinates: List[Tuple[float, float]], max\_segments: int) -> List[Tuple[float, float]]:

if len(coordinates) <= max\_segments:

return coordinates

result = [coordinates[0]]

step = (len(coordinates) - 1) / (max\_segments - 1)

for i in range(1, max\_segments - 1):

idx = int(round(i \* step))

if idx < len(coordinates) and coordinates[idx] not in result:

result.append(coordinates[idx])

result.append(coordinates[-1])

logger.info(f"几何段数量从{len(coordinates)}限制到{len(result)}")

return result

def \_detect\_curve\_segment(self, coordinates: List[Tuple[float, float]], start\_idx: int) -> int:

if start\_idx >= len(coordinates) - 2:

return start\_idx + 1

angle\_threshold = math.radians(10) # 10度阈值

for i in range(start\_idx + 2, len(coordinates)):

if i >= len(coordinates) - 1:

break

p1 = coordinates[i - 2]

p2 = coordinates[i - 1]

p3 = coordinates[i]

angle1 = math.atan2(p2[1] - p1[1], p2[0] - p1[0])

angle2 = math.atan2(p3[1] - p2[1], p3[0] - p2[0])

angle\_diff = abs(angle2 - angle1)

if angle\_diff > math.pi:

angle\_diff = 2 \* math.pi - angle\_diff

if angle\_diff < angle\_threshold:

return i - 1

return len(coordinates) - 1

def \_fit\_single\_arc(self, coordinates: List[Tuple[float, float]], start\_s: float) -> Optional[Dict]:

if len(coordinates) < 3:

return None

try:

center, radius = self.\_fit\_circle(coordinates)

if radius is None or radius < 1.0: # 半径太小，当作直线处理

return None

start\_point = coordinates[0]

end\_point = coordinates[-1]

start\_angle = math.atan2(start\_point[1] - center[1], start\_point[0] - center[0])

end\_angle = math.atan2(end\_point[1] - center[1], end\_point[0] - center[0])

angle\_diff = end\_angle - start\_angle

if angle\_diff > math.pi:

angle\_diff -= 2 \* math.pi

elif angle\_diff < -math.pi:

angle\_diff += 2 \* math.pi

arc\_length = abs(angle\_diff \* radius)

curvature = 1.0 / radius if radius > 0 else 0

dx = coordinates[1][0] - coordinates[0][0]

dy = coordinates[1][1] - coordinates[0][1]

heading = math.atan2(dy, dx)

return {

'type': 'arc',

's': start\_s,

'x': start\_point[0],

'y': start\_point[1],

'hdg': heading,

'length': arc\_length,

'curvature': curvature if angle\_diff > 0 else -curvature

}

except Exception as e:

logger.warning(f"圆弧拟合失败: {e}")

return None

def \_fit\_circle(self, coordinates: List[Tuple[float, float]]) -> Tuple[Tuple[float, float], float]:

points = np.array(coordinates)

if len(points) >= 3:

x1, y1 = points[0]

x2, y2 = points[len(points)//2]

x3, y3 = points[-1]

d = 2 \* (x1 \* (y2 - y3) + x2 \* (y3 - y1) + x3 \* (y1 - y2))

if abs(d) < 1e-10: # 三点共线

return (0, 0), None

ux = ((x1\*\*2 + y1\*\*2) \* (y2 - y3) + (x2\*\*2 + y2\*\*2) \* (y3 - y1) + (x3\*\*2 + y3\*\*2) \* (y1 - y2)) / d

uy = ((x1\*\*2 + y1\*\*2) \* (x3 - x2) + (x2\*\*2 + y2\*\*2) \* (x1 - x3) + (x3\*\*2 + y3\*\*2) \* (x2 - x1)) / d

center = (ux, uy)

radius = math.sqrt((x1 - ux)\*\*2 + (y1 - uy)\*\*2)

return center, radius

return (0, 0), None

def calculate\_road\_length(self, segments: List[Dict]) -> float:

return sum(segment['length'] for segment in segments)

def validate\_geometry\_continuity(self, segments: List[Dict]) -> bool:

if len(segments) < 2:

return True

tolerance = 0.1 # 1cm容差

current\_x = segments[0].get('x', 0.0)

current\_y = segments[0].get('y', 0.0)

for i in range(len(segments) - 1):

current = segments[i]

if current['type'] == 'line':

end\_x = current\_x + current['length'] \* math.cos(current['hdg'])

end\_y = current\_y + current['length'] \* math.sin(current['hdg'])

else: # arc

end\_x = current\_x + current['length'] \* math.cos(current['hdg'])

end\_y = current\_y + current['length'] \* math.sin(current['hdg'])

next\_start\_x = end\_x

next\_start\_y = end\_y

current\_x = next\_start\_x

current\_y = next\_start\_y

return True

def convert\_lane\_surface\_geometry(self, lane\_surfaces: List[Dict]) -> List[Dict]:

converted\_surfaces = []

for surface in lane\_surfaces:

try:

left\_coords = surface['left\_boundary']['coordinates']

right\_coords = surface['right\_boundary']['coordinates']

center\_coords, width\_data = self.\_calculate\_center\_line(left\_coords, right\_coords)

center\_segments = self.convert\_road\_geometry(center\_coords)

width\_profile = self.\_calculate\_width\_profile(left\_coords, right\_coords, center\_segments)

surface\_data = {

'surface\_id': surface['surface\_id'],

'center\_segments': center\_segments,

'width\_profile': width\_profile,

'left\_boundary': surface['left\_boundary'],

'right\_boundary': surface['right\_boundary']

}

converted\_surfaces.append(surface\_data)

except Exception as e:

logger.error(f"车道面 {surface.get('surface\_id', 'unknown')} 几何转换失败: {e}")

continue

logger.info(f"成功转换 {len(converted\_surfaces)} 个车道面的几何")

return converted\_surfaces

def \_calculate\_center\_line(self, left\_coords: List[Tuple[float, float]],

right\_coords: List[Tuple[float, float]]) -> Tuple[List[Tuple[float, float]], List[Dict]]:

max\_points = max(len(left\_coords), len(right\_coords))

if max\_points <= 10:

target\_points = max(max\_points \* 10, 50) # 少点数时增加10倍采样

else:

target\_points = max(max\_points \* 2, 100) # 多点数时增加2倍采样

logger.debug(f"车道面中心线计算：原始点数 {max\_points}，目标点数 {target\_points}")

left\_interpolated = self.\_interpolate\_coordinates(left\_coords, target\_points)

right\_interpolated = self.\_interpolate\_coordinates(right\_coords, target\_points)

center\_coords = []

width\_data = []

current\_s = 0.0

for i, (left\_pt, right\_pt) in enumerate(zip(left\_interpolated, right\_interpolated)):

center\_x = (left\_pt[0] + right\_pt[0]) / 2

center\_y = (left\_pt[1] + right\_pt[1]) / 2

center\_coords.append((center\_x, center\_y))

width = math.sqrt((left\_pt[0] - right\_pt[0])\*\*2 + (left\_pt[1] - right\_pt[1])\*\*2)

width = round(width, self.coordinate\_precision)

width\_info = {

's': current\_s,

'width': width,

'left\_point': left\_pt,

'right\_point': right\_pt,

'center\_point': (center\_x, center\_y)

}

width\_data.append(width\_info)

if i < len(left\_interpolated) - 1:

next\_left, next\_right = left\_interpolated[i + 1], right\_interpolated[i + 1]

next\_center\_x = (next\_left[0] + next\_right[0]) / 2

next\_center\_y = (next\_left[1] + next\_right[1]) / 2

segment\_length = math.sqrt((next\_center\_x - center\_x)\*\*2 + (next\_center\_y - center\_y)\*\*2)

current\_s += segment\_length

if self.preserve\_detail and len(center\_coords) > 10:

original\_center\_coords = center\_coords.copy()

simplified\_coords = self.\_adaptive\_simplify(center\_coords)

logger.debug(f"中心线计算：原始点数 {len(center\_coords)}，简化后 {len(simplified\_coords)}")

simplified\_width\_data = self.\_simplify\_width\_data(width\_data, original\_center\_coords, simplified\_coords)

return simplified\_coords, simplified\_width\_data

return center\_coords, width\_data

def \_simplify\_width\_data(self, width\_data: List[Dict],

original\_coords: List[Tuple[float, float]],

simplified\_coords: List[Tuple[float, float]]) -> List[Dict]:

if not width\_data or not simplified\_coords:

return width\_data

def \_smooth\_width\_profile\_bezier(self, width\_profile: List[Dict]) -> List[Dict]:

if len(width\_profile) < 4:

return width\_profile

try:

s\_values = [item['s'] for item in width\_profile]

width\_values = [item['width'] for item in width\_profile]

smoothed\_widths = self.\_bezier\_smooth(s\_values, width\_values)

for i, item in enumerate(width\_profile):

if i < len(smoothed\_widths):

item['width'] = max(0.1, smoothed\_widths[i]) # 确保宽度不为负

return width\_profile

except Exception as e:

logger.warning(f"贝塞尔曲线平滑失败: {e}，使用原始数据")

return width\_profile

def \_calculate\_cubic\_polynomial\_coefficients(self, width\_profile: List[Dict]) -> List[Dict]:

if len(width\_profile) < 2:

return []

simplified\_profile = self.\_simplify\_width\_profile(width\_profile)

polynomial\_segments = []

for i in range(len(simplified\_profile) - 1):

current\_point = simplified\_profile[i]

next\_point = simplified\_profile[i + 1]

s0 = current\_point['s']

s1 = next\_point['s']

w0 = current\_point['width']

w1 = next\_point['width']

ds = s1 - s0

if ds <= 0:

continue

if i == 0:

if len(simplified\_profile) > 2:

dw0 = (simplified\_profile[i + 1]['width'] - w0) / (simplified\_profile[i + 1]['s'] - s0)

else:

dw0 = 0

else:

dw0 = (w1 - simplified\_profile[i - 1]['width']) / (s1 - simplified\_profile[i - 1]['s'])

if i == len(simplified\_profile) - 2:

if len(simplified\_profile) > 2:

dw1 = (w1 - simplified\_profile[i]['width']) / (s1 - simplified\_profile[i]['s'])

else:

dw1 = 0

else:

dw1 = (simplified\_profile[i + 2]['width'] - w0) / (simplified\_profile[i + 2]['s'] - s0)

a = w0

b = dw0

c = (3 \* (w1 - w0) / ds - 2 \* dw0 - dw1) / ds

d = (2 \* (w0 - w1) / ds + dw0 + dw1) / (ds \* ds)

segment = {

's': s0,

'length': ds,

'a': a,

'b': b,

'c': c,

'd': d,

'end\_s': s1,

'start\_width': w0,

'end\_width': w1

}

polynomial\_segments.append(segment)

return polynomial\_segments

def \_bezier\_smooth(self, s\_values: List[float], width\_values: List[float]) -> List[float]:

if len(width\_values) < 4:

return width\_values

smoothed = []

for i in range(len(width\_values)):

if i == 0:

smoothed.append(width\_values[i])

elif i == len(width\_values) - 1:

smoothed.append(width\_values[i])

else:

p0\_idx = max(0, i - 1)

p1\_idx = i

p2\_idx = min(len(width\_values) - 1, i + 1)

p3\_idx = min(len(width\_values) - 1, i + 2)

p0 = width\_values[p0\_idx]

p1 = width\_values[p1\_idx]

p2 = width\_values[p2\_idx]

p3 = width\_values[p3\_idx]

tension = 0.5 # 张力参数，控制平滑程度

if i > 0 and i < len(width\_values) - 1:

# 前向差分和后向差分的平均

tangent\_in = (p2 - p0) \* tension

tangent\_out = (p3 - p1) \* tension

t = 0.5 # 插值参数

h1 = 2\*t\*\*3 - 3\*t\*\*2 + 1

h2 = -2\*t\*\*3 + 3\*t\*\*2

h3 = t\*\*3 - 2\*t\*\*2 + t

h4 = t\*\*3 - t\*\*2

smoothed\_value = h1\*p1 + h2\*p2 + h3\*tangent\_in + h4\*tangent\_out

min\_val = min(p0, p1, p2, p3) \* 0.8

max\_val = max(p0, p1, p2, p3) \* 1.2

smoothed\_value = max(min\_val, min(max\_val, smoothed\_value))

smoothed.append(smoothed\_value)

else:

smoothed.append(p1)

return smoothed

def \_simplify\_width\_data(self, width\_data: List[Dict], original\_coords: List[Tuple[float, float]],

simplified\_coords: List[Tuple[float, float]]) -> List[Dict]:

if not width\_data or not simplified\_coords:

return width\_data

simplified\_width\_data = []

for simplified\_pt in simplified\_coords:

min\_distance = float('inf')

closest\_index = 0

for i, original\_pt in enumerate(original\_coords):

distance = math.sqrt((simplified\_pt[0] - original\_pt[0])\*\*2 +

(simplified\_pt[1] - original\_pt[1])\*\*2)

if distance < min\_distance:

min\_distance = distance

closest\_index = i

if closest\_index < len(width\_data):

width\_info = width\_data[closest\_index].copy()

width\_info['center\_point'] = tuple(simplified\_pt) if not isinstance(simplified\_pt, tuple) else simplified\_pt

simplified\_width\_data.append(width\_info)

# 重新计算s坐标以保持连续性

current\_s = 0.0

for i, width\_info in enumerate(simplified\_width\_data):

width\_info['s'] = current\_s

if i < len(simplified\_width\_data) - 1:

current\_pt = width\_info['center\_point']

next\_pt = simplified\_width\_data[i + 1]['center\_point']

if isinstance(current\_pt, (list, tuple)) and isinstance(next\_pt, (list, tuple)):

segment\_length = math.sqrt((next\_pt[0] - current\_pt[0])\*\*2 +

(next\_pt[1] - current\_pt[1])\*\*2)

current\_s += segment\_length

logger.debug(f"宽度数据简化：原始 {len(width\_data)} 个点，简化后 {len(simplified\_width\_data)} 个点")

return simplified\_width\_data

def \_interpolate\_coordinates(self, coords: List[Tuple[float, float]],

target\_points: int) -> List[Tuple[float, float]]:

if len(coords) == target\_points:

return coords

distances = [0]

for i in range(1, len(coords)):

dist = math.sqrt((coords[i][0] - coords[i-1][0])\*\*2 +

(coords[i][1] - coords[i-1][1])\*\*2)

distances.append(distances[-1] + dist)

total\_length = distances[-1]

interpolated\_coords = []

for i in range(target\_points):

target\_dist = (i / (target\_points - 1)) \* total\_length

for j in range(len(distances) - 1):

if distances[j] <= target\_dist <= distances[j + 1]:

ratio = (target\_dist - distances[j]) / (distances[j + 1] - distances[j])

x = coords[j][0] + ratio \* (coords[j + 1][0] - coords[j][0])

y = coords[j][1] + ratio \* (coords[j + 1][1] - coords[j][1])

interpolated\_coords.append((x, y))

break

return interpolated\_coords

def \_calculate\_width\_profile(self, left\_coords: List[Tuple[float, float]],

right\_coords: List[Tuple[float, float]],

center\_segments: List[Dict]) -> List[Dict]:

width\_profile = []

if len(left\_coords) != len(right\_coords):

target\_points = max(len(left\_coords), len(right\_coords))

left\_coords = self.\_interpolate\_coordinates(left\_coords, target\_points)

right\_coords = self.\_interpolate\_coordinates(right\_coords, target\_points)

reference\_line = self.\_reconstruct\_reference\_line(center\_segments)

if not reference\_line:

logger.warning("无法从几何段重建参考线，使用简化计算")

return self.\_calculate\_width\_profile\_simple(left\_coords, right\_coords)

total\_length = sum(segment['length'] for segment in center\_segments)

base\_samples = min(len(left\_coords), len(right\_coords)) # 使用较少的边界点数作为基准

if total\_length <= 50: # 短道路：每20-25米一个采样点

samples\_by\_length = max(int(total\_length / 25), 3)

elif total\_length <= 200: # 中等道路：每30-40米一个采样点

samples\_by\_length = max(int(total\_length / 35), 6)

else: # 长道路：每50米一个采样点

samples\_by\_length = max(int(total\_length / 50), 8)

max\_samples = min(20, base\_samples) # 最多20个采样点

num\_samples = min(max(samples\_by\_length, 3), max\_samples) # 至少3个点，最多max\_samples个点

sample\_interval = total\_length / (num\_samples - 1) if num\_samples > 1 else 0

for i in range(num\_samples):

current\_s = i \* sample\_interval

ref\_point, ref\_heading = self.\_get\_reference\_point\_at\_s(center\_segments, current\_s)

if ref\_point is None:

logger.warning(f"无法在s={current\_s:.2f}处找到参考点")

continue

left\_pts = self.\_find\_closest\_two\_points(left\_coords, ref\_point)

right\_pts = self.\_find\_closest\_two\_points(right\_coords, ref\_point)

width = self.\_calculate\_line\_intersection\_width(left\_pts, right\_pts, ref\_point, ref\_heading)

logger.debug(f"宽度计算 - s={current\_s:.2f}: 左边界{left\_pts}, 右边界{right\_pts}, 参考点{ref\_point}, 宽度={width:.3f}")

if width <= 0.001: # 小于1mm认为是异常

logger.warning(f"检测到异常宽度 - s={current\_s:.2f}: 宽度={width:.6f}, 左边界{left\_pts}, 右边界{right\_pts}, 参考点{ref\_point}, 航向角={ref\_heading:.3f}")

width\_data = {

's': current\_s,

'width': width,

'left\_point': left\_pts[0] if left\_pts else ref\_point,

'right\_point': right\_pts[0] if right\_pts else ref\_point,

'reference\_point': ref\_point,

'reference\_heading': ref\_heading

}

width\_profile.append(width\_data)

if len(width\_profile) > 3:

width\_profile = self.\_smooth\_width\_profile\_bezier(width\_profile)

polynomial\_segments = self.\_calculate\_cubic\_polynomial\_coefficients(width\_profile)

for i, segment in enumerate(polynomial\_segments):

if i < len(width\_profile):

width\_profile[i]['polynomial'] = {

'a': segment['a'],

'b': segment['b'],

'c': segment['c'],

'd': segment['d'],

'length': segment['length']

}

logger.info(f"计算车道宽度变化：{len(width\_profile)}个采样点，{len(polynomial\_segments)}个多项式段，总长度{total\_length:.2f}m")

return width\_profile

def \_simplify\_width\_profile(self, width\_profile: List[Dict], width\_threshold: float = 0.02) -> List[Dict]:

if len(width\_profile) <= 2:

return width\_profile

simplified = [width\_profile[0]] # 保留第一个点

for i in range(1, len(width\_profile) - 1):

current\_width = width\_profile[i]['width']

prev\_width = simplified[-1]['width']

next\_width = width\_profile[i + 1]['width']

width\_change\_prev = abs(current\_width - prev\_width)

width\_change\_next = abs(next\_width - current\_width)

if (width\_change\_prev > width\_threshold or

width\_change\_next > width\_threshold or

self.\_is\_local\_extremum(width\_profile, i)):

simplified.append(width\_profile[i])

simplified.append(width\_profile[-1]) # 保留最后一个点

if len(simplified) < len(width\_profile):

logger.info(f"宽度数据简化: {len(width\_profile)} -> {len(simplified)} 点 (阈值: {width\_threshold}m)")

return simplified

def \_is\_local\_extremum(self, width\_profile: List[Dict], index: int) -> bool:

if index <= 0 or index >= len(width\_profile) - 1:

return False

current\_width = width\_profile[index]['width']

prev\_width = width\_profile[index - 1]['width']

next\_width = width\_profile[index + 1]['width']

is\_max = current\_width > prev\_width and current\_width > next\_width

is\_min = current\_width < prev\_width and current\_width < next\_width

return is\_max or is\_min

def \_calculate\_width\_profile\_simple(self, left\_coords: List[Tuple[float, float]],

right\_coords: List[Tuple[float, float]]) -> List[Dict]:

width\_profile = []

current\_s = 0.0

for i, (left\_pt, right\_pt) in enumerate(zip(left\_coords, right\_coords)):

width = math.sqrt((left\_pt[0] - right\_pt[0])\*\*2 + (left\_pt[1] - right\_pt[1])\*\*2)

width = round(width, self.coordinate\_precision)

width\_data = {

's': current\_s,

'width': width,

'left\_point': left\_pt,

'right\_point': right\_pt

}

width\_profile.append(width\_data)

if i < len(left\_coords) - 1:

center\_current = ((left\_pt[0] + right\_pt[0]) / 2, (left\_pt[1] + right\_pt[1]) / 2)

left\_next, right\_next = left\_coords[i + 1], right\_coords[i + 1]

center\_next = ((left\_next[0] + right\_next[0]) / 2, (left\_next[1] + right\_next[1]) / 2)

segment\_length = math.sqrt((center\_next[0] - center\_current[0])\*\*2 +

(center\_next[1] - center\_current[1])\*\*2)

current\_s += segment\_length

return width\_profile

def \_reconstruct\_reference\_line(self, center\_segments: List[Dict]) -> List[Tuple[float, float]]:

reference\_line = []

current\_x = 0.0

current\_y = 0.0

current\_hdg = 0.0

for segment in center\_segments:

if 'x' in segment and 'y' in segment:

current\_x = segment['x']

current\_y = segment['y']

if 'hdg' in segment:

current\_hdg = segment['hdg']

reference\_line.append((current\_x, current\_y))

if segment['type'] == 'line':

length = segment['length']

end\_x = current\_x + length \* math.cos(current\_hdg)

end\_y = current\_y + length \* math.sin(current\_hdg)

reference\_line.append((end\_x, end\_y))

current\_x = end\_x

current\_y = end\_y

elif segment['type'] == 'arc':

length = segment['length']

curvature = segment.get('curvature', 0.0)

if abs(curvature) > 1e-10: # 避免除零

radius = 1.0 / curvature

num\_points = max(int(length / 2.0), 5) # 每2米一个点，最少5个点

for i in range(1, num\_points + 1):

s = (i / num\_points) \* length

angle = s / radius

if curvature > 0: # 左转

center\_x = current\_x - radius \* math.sin(current\_hdg)

center\_y = current\_y + radius \* math.cos(current\_hdg)

point\_angle = current\_hdg - math.pi/2 + angle

else: # 右转

center\_x = current\_x + radius \* math.sin(current\_hdg)

center\_y = current\_y - radius \* math.cos(current\_hdg)

point\_angle = current\_hdg + math.pi/2 - angle

point\_x = center\_x + abs(radius) \* math.cos(point\_angle)

point\_y = center\_y + abs(radius) \* math.sin(point\_angle)

reference\_line.append((point\_x, point\_y))

current\_x = reference\_line[-1][0]

current\_y = reference\_line[-1][1]

current\_hdg += length \* curvature

else:

end\_x = current\_x + length \* math.cos(current\_hdg)

end\_y = current\_y + length \* math.sin(current\_hdg)

reference\_line.append((end\_x, end\_y))

current\_x = end\_x

current\_y = end\_y

return reference\_line

def \_get\_reference\_point\_at\_s(self, center\_segments: List[Dict], s: float) -> Tuple[Tuple[float, float], float]:

current\_s = 0.0

current\_x = 0.0

current\_y = 0.0

current\_hdg = 0.0

for segment in center\_segments:

if 'x' in segment and 'y' in segment:

current\_x = segment['x']

current\_y = segment['y']

if 'hdg' in segment:

current\_hdg = segment['hdg']

segment\_length = segment['length']

if current\_s + segment\_length >= s:

local\_s = s - current\_s

if segment['type'] == 'line':

point\_x = current\_x + local\_s \* math.cos(current\_hdg)

point\_y = current\_y + local\_s \* math.sin(current\_hdg)

heading = current\_hdg

elif segment['type'] == 'arc':

curvature = segment.get('curvature', 0.0)

if abs(curvature) > 1e-10:

radius = 1.0 / curvature

angle = local\_s / radius

if curvature > 0: # 左转

center\_x = current\_x - radius \* math.sin(current\_hdg)

center\_y = current\_y + radius \* math.cos(current\_hdg)

point\_angle = current\_hdg - math.pi/2 + angle

else: # 右转

center\_x = current\_x + radius \* math.sin(current\_hdg)

center\_y = current\_y - radius \* math.cos(current\_hdg)

point\_angle = current\_hdg + math.pi/2 - angle

point\_x = center\_x + abs(radius) \* math.cos(point\_angle)

point\_y = center\_y + abs(radius) \* math.sin(point\_angle)

heading = current\_hdg + local\_s \* curvature

else:

point\_x = current\_x + local\_s \* math.cos(current\_hdg)

point\_y = current\_y + local\_s \* math.sin(current\_hdg)

heading = current\_hdg

else:

point\_x = current\_x + local\_s \* math.cos(current\_hdg)

point\_y = current\_y + local\_s \* math.sin(current\_hdg)

heading = current\_hdg

return (point\_x, point\_y), heading

current\_s += segment\_length

if segment['type'] == 'line':

current\_x += segment\_length \* math.cos(current\_hdg)

current\_y += segment\_length \* math.sin(current\_hdg)

elif segment['type'] == 'arc':

curvature = segment.get('curvature', 0.0)

if abs(curvature) > 1e-10:

current\_hdg += segment\_length \* curvature

current\_x += segment\_length \* math.cos(current\_hdg)

current\_y += segment\_length \* math.sin(current\_hdg)

return (current\_x, current\_y), current\_hdg

def \_find\_closest\_point(self, coords: List[Tuple[float, float]],

target: Tuple[float, float]) -> Tuple[float, float]:

if not coords:

return target

min\_dist = float('inf')

closest\_point = coords[0]

for point in coords:

dist = math.sqrt((point[0] - target[0])\*\*2 + (point[1] - target[1])\*\*2)

if dist < min\_dist:

min\_dist = dist

closest\_point = point

return closest\_point

def \_find\_closest\_two\_points(self, coords: List[Tuple[float, float]],

target: Tuple[float, float]) -> List[Tuple[float, float]]:

if not coords:

return [target, target]

if len(coords) == 1:

return [coords[0], coords[0]]

distances = []

for i, point in enumerate(coords):

dist = math.sqrt((point[0] - target[0])\*\*2 + (point[1] - target[1])\*\*2)

distances.append((dist, i, point))

distances.sort(key=lambda x: x[0])

return [distances[0][2], distances[1][2]]

def \_calculate\_perpendicular\_width(self, left\_pt: Tuple[float, float],

right\_pt: Tuple[float, float],

ref\_pt: Tuple[float, float],

ref\_heading: float) -> float:

perp\_x = -math.sin(ref\_heading)

perp\_y = math.cos(ref\_heading)

left\_vec\_x = left\_pt[0] - ref\_pt[0]

left\_vec\_y = left\_pt[1] - ref\_pt[1]

right\_vec\_x = right\_pt[0] - ref\_pt[0]

right\_vec\_y = right\_pt[1] - ref\_pt[1]

left\_proj = left\_vec\_x \* perp\_x + left\_vec\_y \* perp\_y

right\_proj = right\_vec\_x \* perp\_x + right\_vec\_y \* perp\_y

width = abs(left\_proj - right\_proj)

logger.debug(f"垂直宽度计算详情: 航向角={ref\_heading:.3f}, 垂直向量=({perp\_x:.3f},{perp\_y:.3f}), "

f"左投影={left\_proj:.3f}, 右投影={right\_proj:.3f}, 原始宽度={width:.6f}")

if width <= 0.001:

direct\_width = math.sqrt((left\_pt[0] - right\_pt[0])\*\*2 + (left\_pt[1] - right\_pt[1])\*\*2)

logger.warning(f"垂直宽度异常小({width:.6f})，直线距离={direct\_width:.3f}，"

f"左边界{left\_pt}, 右边界{right\_pt}, 参考点{ref\_pt}")

if direct\_width > 0.1: # 直线距离大于10cm

logger.info(f"使用直线距离替代垂直宽度: {direct\_width:.3f}")

width = direct\_width

return round(width, self.coordinate\_precision)

def \_calculate\_line\_intersection\_width(self, left\_pts: List[Tuple[float, float]],

right\_pts: List[Tuple[float, float]],

ref\_pt: Tuple[float, float],

ref\_heading: float) -> float:

if len(left\_pts) < 2 or len(right\_pts) < 2:

left\_pt = left\_pts[0] if left\_pts else ref\_pt

right\_pt = right\_pts[0] if right\_pts else ref\_pt

return self.\_calculate\_perpendicular\_width(left\_pt, right\_pt, ref\_pt, ref\_heading)

perp\_x = -math.sin(ref\_heading)

perp\_y = math.cos(ref\_heading)

left\_intersection = self.\_line\_intersection(

left\_pts[0], left\_pts[1], # 左边界直线的两个点

ref\_pt, (ref\_pt[0] + perp\_x, ref\_pt[1] + perp\_y) # 垂线的两个点

)

right\_intersection = self.\_line\_intersection(

right\_pts[0], right\_pts[1], # 右边界直线的两个点

ref\_pt, (ref\_pt[0] + perp\_x, ref\_pt[1] + perp\_y) # 垂线的两个点

)

if left\_intersection is None or right\_intersection is None:

logger.warning(f"无法计算直线交点，回退到投影方法")

left\_pt = left\_pts[0]

right\_pt = right\_pts[0]

return self.\_calculate\_perpendicular\_width(left\_pt, right\_pt, ref\_pt, ref\_heading)

width = math.sqrt((left\_intersection[0] - right\_intersection[0])\*\*2 +

(left\_intersection[1] - right\_intersection[1])\*\*2)

logger.debug(f"直线交点宽度计算: 左交点{left\_intersection}, 右交点{right\_intersection}, 宽度={width:.6f}")

if width <= 0.001:

logger.warning(f"交点宽度异常小({width:.6f})，回退到投影方法")

left\_pt = left\_pts[0]

right\_pt = right\_pts[0]

return self.\_calculate\_perpendicular\_width(left\_pt, right\_pt, ref\_pt, ref\_heading)

return round(width, self.coordinate\_precision)

def \_line\_intersection(self, p1: Tuple[float, float], p2: Tuple[float, float],

p3: Tuple[float, float], p4: Tuple[float, float]) -> Optional[Tuple[float, float]]:

x1, y1 = p1

x2, y2 = p2

x3, y3 = p3

x4, y4 = p4

denom = (x1 - x2) \* (y3 - y4) - (y1 - y2) \* (x3 - x4)

if abs(denom) < 1e-10:

return None

t = ((x1 - x3) \* (y3 - y4) - (y1 - y3) \* (x3 - x4)) / denom

intersection\_x = x1 + t \* (x2 - x1)

intersection\_y = y1 + t \* (y2 - y1)

return (intersection\_x, intersection\_y)

def \_fit\_polynomial\_curves(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

if len(coordinates) < 3:

return self.fit\_line\_segments(coordinates)

if len(coordinates) > 20: # 对于复杂曲线使用分段拟合

return self.\_fit\_segmented\_polynomial\_curves(coordinates)

segments = []

current\_s = 0.0

coords\_array = np.array(coordinates)

x\_coords = coords\_array[:, 0]

y\_coords = coords\_array[:, 1]

arc\_lengths = self.\_calculate\_arc\_lengths(coordinates)

total\_length = arc\_lengths[-1]

if total\_length <= 0:

return self.fit\_line\_segments(coordinates)

t\_params = arc\_lengths / total\_length

try:

start\_heading = self.\_calculate\_precise\_heading(coordinates[:min(3, len(coordinates))])

start\_x, start\_y = x\_coords[0], y\_coords[0]

cos\_hdg = math.cos(start\_heading)

sin\_hdg = math.sin(start\_heading)

local\_u = np.zeros(len(coordinates))

local\_v = np.zeros(len(coordinates))

for i in range(len(coordinates)):

dx = x\_coords[i] - start\_x

dy = y\_coords[i] - start\_y

local\_u[i] = dx \* cos\_hdg + dy \* sin\_hdg

local\_v[i] = -dx \* sin\_hdg + dy \* cos\_hdg

optimal\_degree = self.\_select\_optimal\_polynomial\_degree(t\_params, local\_u, local\_v)

weights = self.\_calculate\_fitting\_weights(len(coordinates))

poly\_u = np.polyfit(t\_params, local\_u, optimal\_degree, w=weights)

poly\_v = np.polyfit(t\_params, local\_v, optimal\_degree, w=weights)

fitting\_error = self.\_evaluate\_fitting\_quality(t\_params, local\_u, local\_v, poly\_u, poly\_v)

if fitting\_error > self.tolerance and optimal\_degree > 2:

logger.debug(f"拟合误差过大({fitting\_error:.3f}m)，降低多项式阶数重新拟合")

optimal\_degree = max(2, optimal\_degree - 1)

poly\_u = np.polyfit(t\_params, local\_u, optimal\_degree, w=weights)

poly\_v = np.polyfit(t\_params, local\_v, optimal\_degree, w=weights)

fitting\_error = self.\_evaluate\_fitting\_quality(t\_params, local\_u, local\_v, poly\_u, poly\_v)

poly\_u\_padded = np.pad(poly\_u[::-1], (0, max(0, 4 - len(poly\_u))), 'constant')

poly\_v\_padded = np.pad(poly\_v[::-1], (0, max(0, 4 - len(poly\_v))), 'constant')

au = float(poly\_u\_padded[0]) if len(poly\_u\_padded) > 0 else 0.0

bu = float(poly\_u\_padded[1]) if len(poly\_u\_padded) > 1 else 0.0

cu = float(poly\_u\_padded[2]) if len(poly\_u\_padded) > 2 else 0.0

du = float(poly\_u\_padded[3]) if len(poly\_u\_padded) > 3 else 0.0

av = float(poly\_v\_padded[0]) if len(poly\_v\_padded) > 0 else 0.0

bv = float(poly\_v\_padded[1]) if len(poly\_v\_padded) > 1 else 0.0

cv = float(poly\_v\_padded[2]) if len(poly\_v\_padded) > 2 else 0.0

dv = float(poly\_v\_padded[3]) if len(poly\_v\_padded) > 3 else 0.0

au, bu, cu, du, av, bv, cv, dv = self.\_optimize\_boundary\_conditions(

local\_u, local\_v, au, bu, cu, du, av, bv, cv, dv, optimal\_degree

)

segment = {

'type': 'parampoly3',

's': current\_s,

'x': float(x\_coords[0]),

'y': float(y\_coords[0]),

'hdg': start\_heading,

'length': total\_length,

'au': au,

'bu': bu,

'cu': cu,

'du': du,

'av': av,

'bv': bv,

'cv': cv,

'dv': dv,

'fitting\_error': fitting\_error,

'polynomial\_degree': optimal\_degree

}

segments.append(segment)

logger.debug(f"高精度ParamPoly3拟合完成，点数: {len(coordinates)}, 长度: {total\_length:.2f}m, 阶数: {optimal\_degree}, 误差: {fitting\_error:.4f}m")

logger.debug(f"多项式系数 - au:{au:.8f}, bu:{bu:.8f}, cu:{cu:.8f}, du:{du:.8f}")

logger.debug(f"多项式系数 - av:{av:.8f}, bv:{bv:.8f}, cv:{cv:.8f}, dv:{dv:.8f}")

except Exception as e:

logger.warning(f"高精度ParamPoly3拟合失败: {e}，回退到直线拟合")

segments = self.fit\_line\_segments(coordinates)

return segments

def \_fit\_spline\_curves(self, coordinates: List[Tuple[float, float]]) -> List[Dict]:

if len(coordinates) < 4:

return self.fit\_line\_segments(coordinates)

segments = []

current\_s = 0.0

try:

coords\_array = np.array(coordinates)

smoothing\_factor = self.curve\_smoothness \* len(coordinates) \* self.tolerance

tck, u = splprep([coords\_array[:, 0], coords\_array[:, 1]],

s=smoothing\_factor, k=min(3, len(coordinates)-1))

num\_points = max(len(coordinates), int(len(coordinates) \* (2.0 - self.curve\_smoothness)))

u\_new = np.linspace(0, 1, num\_points)

spline\_coords = splev(u\_new, tck)

spline\_x = spline\_coords[0]

spline\_y = spline\_coords[1]

spline\_x[0] = coords\_array[0, 0] # 固定起点X坐标

spline\_y[0] = coords\_array[0, 1] # 固定起点Y坐标

spline\_x[-1] = coords\_array[-1, 0] # 固定终点X坐标

spline\_y[-1] = coords\_array[-1, 1] # 固定终点Y坐标

fitted\_coords = list(zip(spline\_x, spline\_y))

segments = self.\_fit\_adaptive\_line\_segments(fitted\_coords, current\_s)

logger.debug(f"样条拟合完成，原始点数: {len(coordinates)}, 拟合点数: {num\_points}, 几何段数: {len(segments)}")

except Exception as e:

logger.warning(f"样条拟合失败: {e}，回退到直线拟合")

segments = self.fit\_line\_segments(coordinates)

return segments

from scenariogeneration import xodr

from scenariogeneration.xodr.geometry import Line, Arc, ParamPoly3, PlanView

from scenariogeneration.xodr.lane import Lane, LaneSection, Lanes

from scenariogeneration.xodr.enumerations import LaneType

import numpy as np

from typing import List, Dict, Optional, Tuple, Any

import logging

import os

from pathlib import Path

logger = logging.getLogger(\_\_name\_\_)

try:

from geometry\_converter import ConversionHandler

except ImportError:

from abc import ABC, abstractmethod

class ConversionHandler(ABC):

def \_\_init\_\_(self):

self.\_next\_handler = None

def set\_next(self, handler: 'ConversionHandler') -> 'ConversionHandler':

self.\_next\_handler = handler

return handler

@abstractmethod

def handle(self, request: Dict[str, Any]) -> Dict[str, Any]:

pass

def \_handle\_next(self, request: Dict[str, Any]) -> Dict[str, Any]:

if self.\_next\_handler:

return self.\_next\_handler.handle(request)

return request

class OpenDriveGenerationHandler(ConversionHandler):

def \_\_init\_\_(self, name: str = "ConvertedRoad", curve\_fitting\_mode: str = "polyline",

polynomial\_degree: int = 3, curve\_smoothness: float = 0.5):

super().\_\_init\_\_()

self.opendrive\_generator = OpenDriveGenerator(name, curve\_fitting\_mode, polynomial\_degree, curve\_smoothness)

def handle(self, request: Dict[str, Any]) -> Dict[str, Any]:

if request.get('stage') != 'opendrive\_generation':

return self.\_handle\_next(request)

converted\_roads = request.get('converted\_roads', [])

output\_path = request.get('output\_path', '')

road\_ids = []

for road\_data in converted\_roads:

if road\_data.get('type') == 'lane\_based':

road\_id = self.opendrive\_generator.create\_road\_from\_lane\_surfaces(

road\_data.get('lane\_surfaces', []),

road\_data.get('attributes', {})

)

else:

road\_id = self.opendrive\_generator.create\_road\_from\_segments(

road\_data.get('segments', []),

road\_data.get('attributes', {})

)

if road\_id > 0:

road\_ids.append(road\_id)

if road\_ids and output\_path:

success = self.opendrive\_generator.generate\_file(output\_path)

request['success'] = success

request['road\_ids'] = road\_ids

else:

request['success'] = False

request['stage'] = 'completed'

return self.\_handle\_next(request)

class OpenDriveGenerator:

def \_\_init\_\_(self, name: str = "ConvertedRoad", curve\_fitting\_mode: str = "polyline",

polynomial\_degree: int = 3, curve\_smoothness: float = 0.5):

self.name = name

self.curve\_fitting\_mode = curve\_fitting\_mode

self.polynomial\_degree = polynomial\_degree

self.curve\_smoothness = curve\_smoothness

self.odr = xodr.OpenDrive(self.name)

self.odr.revMajor = 1

self.odr.revMinor = 7

self.roads = []

self.road\_id\_counter = 1

def create\_road\_from\_segments(self, segments: List[Dict],

road\_attributes: Dict = None) -> int:

if not segments:

logger.error("几何段列表为空")

return -1

if road\_attributes is None:

road\_attributes = {

'lane\_width': 3.5, # 默认车道宽度3.5米

'num\_lanes': 1, # 默认单车道

'speed\_limit': 50 # 默认限速50km/h

}

road\_id = self.road\_id\_counter

self.road\_id\_counter += 1

try:

planview = self.\_create\_planview\_from\_segments(segments)

lanes = self.\_create\_lane\_section(road\_attributes)

road = xodr.Road(road\_id, planview, lanes)

self.odr.add\_road(road)

self.roads.append(road)

logger.info(f"成功创建道路 ID: {road\_id}，包含 {len(segments)} 个几何段")

return road\_id

except Exception as e:

logger.error(f"创建道路失败: {e}")

return -1

def create\_road\_from\_lane\_surfaces(self, lane\_surfaces: List[Dict],

road\_attributes: Dict = None) -> int:

if not lane\_surfaces:

logger.error("车道面数据为空")

return -1

segments = self.\_calculate\_road\_reference\_line(lane\_surfaces)

if not segments:

logger.error("无法计算道路参考线")

return -1

road\_id = self.road\_id\_counter

self.road\_id\_counter += 1

try:

planview = self.\_create\_planview\_from\_segments(segments)

lanes = self.\_create\_lane\_section\_from\_surfaces(lane\_surfaces, road\_attributes, road\_id)

road = xodr.Road(road\_id, planview, lanes)

self.odr.add\_road(road)

self.roads.append(road)

logger.info(f"成功创建基于车道面的道路 ID: {road\_id}，包含 {len(lane\_surfaces)} 个车道面")

return road\_id

except Exception as e:

logger.error(f"创建基于车道面的道路失败: {e}")

return -1

def \_calculate\_road\_reference\_line(self, lane\_surfaces: List[Dict]) -> List[Dict]:

try:

if not lane\_surfaces:

return []

reference\_boundary = None

reference\_coords = None

logger.info("查找index为'0'的边界线作为planview参考线")

for surface in lane\_surfaces:

if ('left\_boundary' in surface and

surface['left\_boundary'].get('index') == '0'):

reference\_boundary = surface['left\_boundary']

reference\_coords = surface['left\_boundary']['coordinates']

logger.info(f"找到index为'0'的左边界线，坐标点数: {len(reference\_coords)}")

break

if ('right\_boundary' in surface and

surface['right\_boundary'].get('index') == '0'):

reference\_boundary = surface['right\_boundary']

reference\_coords = surface['right\_boundary']['coordinates']

logger.info(f"找到index为'0'的右边界线，坐标点数: {len(reference\_coords)}")

break

if not reference\_coords:

logger.warning("未找到index为'0'的边界线，回退到使用第一个车道面的中心线")

reference\_surface = lane\_surfaces[0]

if 'center\_line' in reference\_surface:

reference\_coords = reference\_surface['center\_line']

elif ('left\_boundary' in reference\_surface and

'right\_boundary' in reference\_surface):

left\_coords = reference\_surface['left\_boundary']['coordinates']

right\_coords = reference\_surface['right\_boundary']['coordinates']

reference\_coords = self.\_calculate\_center\_line\_coords(left\_coords, right\_coords)

if not reference\_coords:

logger.error("无法获取有效的参考线坐标")

return []

from geometry\_converter import GeometryConverter

converter = GeometryConverter(

tolerance=0.1,

smooth\_curves=False,

preserve\_detail=True,

curve\_fitting\_mode=getattr(self, 'curve\_fitting\_mode', 'polyline'),

polynomial\_degree=getattr(self, 'polynomial\_degree', 3),

curve\_smoothness=getattr(self, 'curve\_smoothness', 0.5),

coordinate\_precision=getattr(self, 'coordinate\_precision', 3)

)

segments = converter.convert\_road\_geometry(reference\_coords)

logger.info(f"使用高精度参数转换边界线：容差=0.1m, 保留细节=True, 原始坐标点数={len(reference\_coords)}")

if segments:

logger.info(f"道路参考线计算完成，使用index为'0'的边界线，包含 {len(segments)} 个几何段")

return segments

else:

logger.error("边界线坐标转换为几何段失败")

return []

except Exception as e:

logger.error(f"计算道路参考线失败: {e}")

return []

def \_extract\_coordinates\_from\_segments(self, segments: List[Dict]) -> List[Tuple[float, float]]:

from geometry\_converter import GeometryConverter

converter = GeometryConverter(coordinate\_precision=getattr(self, 'coordinate\_precision', 3))

return converter.\_reconstruct\_reference\_line(segments)

def \_calculate\_center\_line\_coords(self, left\_coords: List[Tuple[float, float]],

right\_coords: List[Tuple[float, float]]) -> List[Tuple[float, float]]:

if not left\_coords or not right\_coords:

return []

min\_points = min(len(left\_coords), len(right\_coords))

center\_coords = []

for i in range(min\_points):

left\_x, left\_y = left\_coords[i]

right\_x, right\_y = right\_coords[i]

center\_x = (left\_x + right\_x) / 2

center\_y = (left\_y + right\_y) / 2

center\_coords.append((center\_x, center\_y))

return center\_coords

def \_calculate\_average\_center\_line(self, all\_center\_coords: List[List[Tuple[float, float]]]) -> List[Tuple[float, float]]:

if not all\_center\_coords:

return []

max\_length = max(len(coords) for coords in all\_center\_coords)

if max\_length == 0:

return []

interpolated\_coords = []

for coords in all\_center\_coords:

if len(coords) == max\_length:

interpolated\_coords.append(coords)

else:

interpolated = self.\_interpolate\_to\_target\_length(coords, max\_length)

interpolated\_coords.append(interpolated)

avg\_coords = []

for i in range(max\_length):

avg\_x = sum(coords[i][0] for coords in interpolated\_coords) / len(interpolated\_coords)

avg\_y = sum(coords[i][1] for coords in interpolated\_coords) / len(interpolated\_coords)

avg\_coords.append((avg\_x, avg\_y))

return avg\_coords

def \_interpolate\_to\_target\_length(self, coords: List[Tuple[float, float]], target\_length: int) -> List[Tuple[float, float]]:

if len(coords) <= 1 or target\_length <= 1:

return coords

import numpy as np

from scipy import interpolate

distances = [0]

for i in range(1, len(coords)):

dist = ((coords[i][0] - coords[i-1][0])\*\*2 + (coords[i][1] - coords[i-1][1])\*\*2)\*\*0.5

distances.append(distances[-1] + dist)

total\_distance = distances[-1]

target\_distances = np.linspace(0, total\_distance, target\_length)

x\_coords = [coord[0] for coord in coords]

y\_coords = [coord[1] for coord in coords]

f\_x = interpolate.interp1d(distances, x\_coords, kind='linear', bounds\_error=False, fill\_value='extrapolate')

f\_y = interpolate.interp1d(distances, y\_coords, kind='linear', bounds\_error=False, fill\_value='extrapolate')

interpolated\_x = f\_x(target\_distances)

interpolated\_y = f\_y(target\_distances)

return list(zip(interpolated\_x, interpolated\_y))

def \_create\_planview\_from\_segments(self, segments: List[Dict]) -> xodr.PlanView:

start\_x = segments[0].get('x', 0) if segments else 0

start\_y = segments[0].get('y', 0) if segments else 0

start\_heading = segments[0].get('hdg', 0) if segments else 0

planview = xodr.PlanView(start\_x, start\_y, start\_heading)

for segment in segments:

if segment['type'] == 'line':

geometry = Line(length=segment['length'])

planview.add\_geometry(geometry, heading=segment['hdg'])

elif segment['type'] == 'arc':

geometry = Arc(curvature=segment['curvature'], length=segment['length'])

planview.add\_geometry(geometry, heading=segment['hdg'])

elif segment['type'] == 'parampoly3':

geometry = ParamPoly3(

au=segment['au'],

bu=segment['bu'],

cu=segment['cu'],

du=segment['du'],

av=segment['av'],

bv=segment['bv'],

cv=segment['cv'],

dv=segment['dv'],

prange='normalized',

length=segment['length']

)

planview.add\_geometry(geometry, heading=segment['hdg'])

else:

logger.warning(f"未知的几何类型: {segment['type']}")

return planview

def \_create\_lane\_section(self, attributes: Dict) -> xodr.Lanes:

lanes = xodr.Lanes()

lane\_section = xodr.LaneSection(0, xodr.standard\_lane())

num\_lanes = attributes.get('num\_lanes', 1)

lane\_width = attributes.get('lane\_width', 3.5)

for i in range(1, num\_lanes + 1):

lane = xodr.Lane(lane\_type=xodr.LaneType.driving)

lane.add\_lane\_width(a=lane\_width, soffset=0)

road\_mark = xodr.RoadMark(

xodr.RoadMarkType.solid,

xodr.RoadMarkWeight.standard,

xodr.RoadMarkColor.white

)

lane.add\_roadmark(road\_mark)

lane\_section.add\_right\_lane(lane)

if attributes.get('bidirectional', False):

for i in range(1, num\_lanes + 1):

lane = xodr.Lane(lane\_type=xodr.LaneType.driving)

lane.add\_lane\_width(a=lane\_width, soffset=0)

lane\_section.add\_left\_lane(lane)

lanes.add\_lanesection(lane\_section)

return lanes

def \_create\_lane\_section\_from\_surfaces(self, lane\_surfaces: List[Dict],

attributes: Dict = None, road\_id: int = None) -> xodr.Lanes:

lanes = xodr.Lanes()

lane\_section = xodr.LaneSection(0, xodr.standard\_lane())

logger.info(f"创建车道剖面 (Road ID: {road\_id})：总共{len(lane\_surfaces)}个车道面，全部创建为右侧车道")

for i, surface in enumerate(lane\_surfaces):

lane = xodr.Lane(lane\_type=xodr.LaneType.driving)

lane\_id = -(i + 1)

width\_profile = surface.get('width\_profile', [])

surface\_id = surface.get('surface\_id', f'surface\_{i}')

logger.info(f"处理车道 (Road ID: {road\_id}, Lane ID: {lane\_id}, Surface ID: {surface\_id})")

if width\_profile:

widths = [wp['width'] for wp in width\_profile]

min\_width = min(widths)

max\_width = max(widths)

width\_change = max\_width - min\_width

logger.info(f" 宽度数据: {len(width\_profile)}个采样点, 最小={min\_width:.3f}m, 最大={max\_width:.3f}m, 变化={width\_change:.3f}m")

zero\_widths = [wp for wp in width\_profile if wp['width'] <= 0.001]

if zero\_widths:

logger.warning(f" 发现{len(zero\_widths)}个异常宽度点 (Road ID: {road\_id}, Lane ID: {lane\_id}):")

for zw in zero\_widths[:5]:

logger.warning(f" s={zw['s']:.2f}m, width={zw['width']:.6f}")

if min\_width > 0:

width\_change\_ratio = width\_change / min\_width

else:

avg\_width = sum(wp['width'] for wp in width\_profile) / len(width\_profile) if width\_profile else 0

width\_change\_ratio = width\_change / avg\_width if avg\_width > 0 else float('inf')

if width\_change > 0.1 and width\_change\_ratio > 0.03:

logger.info(f" 检测到变宽车道: 变化率={width\_change\_ratio:.1%}, 原始宽度点数={len(width\_profile)}")

filtered\_profile = self.\_filter\_significant\_width\_changes(width\_profile)

logger.info(f" 过滤后宽度点数: {len(filtered\_profile)} (Road ID: {road\_id}, Lane ID: {lane\_id})")

max\_segments = min(8, len(filtered\_profile))

if len(filtered\_profile) > max\_segments:

logger.warning(f" 宽度变化段过多，将从{len(filtered\_profile)}段简化为{max\_segments}段")

segment\_count = 0

for j, wp in enumerate(filtered\_profile):

if segment\_count >= max\_segments:

break

if 'polynomial' in wp:

poly = wp['polynomial']

lane.add\_lane\_width(

a=poly['a'],

b=poly['b'],

c=poly['c'],

d=poly['d'],

soffset=wp['s']

)

segment\_count += 1

if j == 0:

logger.info(f" 起始多项式段: s={wp['s']:.2f}m, a={poly['a']:.3f}, b={poly['b']:.3f}, c={poly['c']:.3f}, d={poly['d']:.3f}")

else:

lane.add\_lane\_width(a=wp['width'], soffset=wp['s'])

segment\_count += 1

if j == len(filtered\_profile) - 1:

logger.info(f" 结束宽度: s={wp['s']:.2f}m, width={wp['width']:.2f}m")

logger.info(f" 实际使用{segment\_count}个多项式段（原始{len(width\_profile)}个，过滤后{len(filtered\_profile)}个，限制{max\_segments}个）")

else:

lane\_width = width\_profile[0]['width']

lane.add\_lane\_width(a=lane\_width, soffset=0)

logger.info(f" 等宽车道: 宽度={lane\_width:.3f}m")

if lane\_width <= 0.001:

logger.error(f" 异常！车道宽度为0 (Road ID: {road\_id}, Lane ID: {lane\_id}, Surface ID: {surface\_id})")

logger.error(f" 宽度数据详情: {width\_profile[:3]}...") # 显示前3个数据点

valid\_widths = [wp['width'] for wp in width\_profile if wp['width'] > 0.001]

if valid\_widths:

lane\_width = min(valid\_widths) # 使用最小的有效宽度

lane.add\_lane\_width(a=lane\_width, soffset=0)

logger.warning(f" 已修正为最小有效宽度: {lane\_width:.3f}m")

else:

logger.error(f" 所有宽度数据都为0，跳过此车道")

continue

else:

lane\_width = attributes.get('lane\_width', 3.5) if attributes else 3.5

lane.add\_lane\_width(a=lane\_width, soffset=0)

logger.warning(f" 车道面缺少宽度信息，使用默认宽度{lane\_width:.2f}m (Road ID: {road\_id}, Lane ID: {lane\_id})")

road\_mark = xodr.RoadMark(

xodr.RoadMarkType.solid,

xodr.RoadMarkWeight.standard,

xodr.RoadMarkColor.white

)

lane.add\_roadmark(road\_mark)

lane\_section.add\_right\_lane(lane)

logger.info(f" 车道已添加为右侧车道 (Road ID: {road\_id}, Lane ID: {lane\_id}, Surface ID: {surface\_id})")

lanes.add\_lanesection(lane\_section)

logger.info(f"车道剖面创建完成 (Road ID: {road\_id})，包含{len(lane\_surfaces)}个右侧车道")

return lanes

def create\_multiple\_roads(self, roads\_data: List[Dict]) -> List[int]:

road\_ids = []

for road\_data in roads\_data:

segments = road\_data.get('segments', [])

attributes = road\_data.get('attributes', {})

road\_id = self.create\_road\_from\_segments(segments, attributes)

if road\_id > 0:

road\_ids.append(road\_id)

logger.info(f"成功创建 {len(road\_ids)} 条道路")

return road\_ids

def \_filter\_significant\_width\_changes(self, width\_profile: List[Dict],

change\_threshold: float = 0.05) -> List[Dict]:

if len(width\_profile) <= 2:

return width\_profile

filtered = [width\_profile[0]] # 保留第一个点

for i in range(1, len(width\_profile) - 1):

current\_width = width\_profile[i]['width']

prev\_width = filtered[-1]['width']

next\_width = width\_profile[i + 1]['width']

width\_change\_prev = abs(current\_width - prev\_width)

width\_change\_next = abs(next\_width - current\_width)

is\_turning\_point = ((current\_width > prev\_width and current\_width > next\_width) or

(current\_width < prev\_width and current\_width < next\_width))

if (width\_change\_prev > change\_threshold or

width\_change\_next > change\_threshold or

is\_turning\_point):

filtered.append(width\_profile[i])

filtered.append(width\_profile[-1]) # 保留最后一个点

return filtered

def add\_road\_connections(self, connections: List[Dict]):

for connection in connections:

try:

road1\_id = connection['road1\_id']

road2\_id = connection['road2\_id']

contact\_point = connection.get('contact\_point', 'end')

logger.info(f"添加道路连接: {road1\_id} -> {road2\_id}")

except Exception as e:

logger.error(f"添加道路连接失败: {e}")

def add\_road\_objects(self, road\_id: int, objects: List[Dict]):

try:

road = next((r for r in self.roads if r.id == road\_id), None)

if not road:

logger.error(f"未找到道路 ID: {road\_id}")

return

for obj in objects:

road\_object = xodr.Object(

s=obj.get('s', 0),

t=obj.get('t', 0),

object\_id=obj.get('id', 0),

object\_type=obj.get('type', 'pole'),

name=obj.get('name', ''),

zOffset=obj.get('z\_offset', 0),

hdg=obj.get('heading', 0)

)

road.add\_object(road\_object)

logger.info(f"为道路 {road\_id} 添加了 {len(objects)} 个对象")

except Exception as e:

logger.error(f"添加道路对象失败: {e}")

def set\_road\_elevation(self, road\_id: int, elevation\_data: List[Dict]):

try:

road = next((r for r in self.roads if r.id == road\_id), None)

if not road:

logger.error(f"未找到道路 ID: {road\_id}")

return

elevation\_profile = xodr.ElevationProfile()

for elev in elevation\_data:

elevation = xodr.Elevation(

s=elev.get('s', 0),

a=elev.get('a', 0), # 高程值

b=elev.get('b', 0), # 坡度

c=elev.get('c', 0), # 曲率变化

d=elev.get('d', 0) # 曲率变化率

)

elevation\_profile.add\_elevation(elevation)

road.add\_elevation\_profile(elevation\_profile)

logger.info(f"为道路 {road\_id} 设置了高程剖面")

except Exception as e:

logger.error(f"设置道路高程失败: {e}")

def generate\_file(self, output\_path: str) -> bool:

try:

# 创建输出目录

output\_dir = os.path.dirname(output\_path)

if output\_dir and not os.path.exists(output\_dir):

os.makedirs(output\_dir)

self.odr.adjust\_roads\_and\_lanes()

self.odr.write\_xml(output\_path)

logger.info(f"OpenDrive文件已生成: {output\_path}")

return True

except Exception as e:

logger.error(f"生成OpenDrive文件失败: {e}")

return False

def validate\_opendrive(self) -> Dict[str, any]:

validation\_result = {

'valid': True,

'errors': [],

'warnings': [],

'road\_count': len(self.roads),

'total\_length': 0

}

try:

if len(self.roads) == 0:

validation\_result['errors'].append("没有道路数据")

validation\_result['valid'] = False

for road in self.roads:

road\_length = 0

if hasattr(road, 'planview') and road.planview:

road\_length = 100.0 # 设置默认长度用于验证

validation\_result['total\_length'] += road\_length

if road\_length < 1.0:

validation\_result['warnings'].append(f"道路 {road.id} 长度过短: {road\_length:.2f}m")

if not hasattr(road, 'lanes') or not road.lanes:

validation\_result['errors'].append(f"道路 {road.id} 缺少车道定义")

validation\_result['valid'] = False

logger.info(f"验证完成: {validation\_result['road\_count']} 条道路，总长度: {validation\_result['total\_length']:.2f}m")

except Exception as e:

validation\_result['errors'].append(f"验证过程出错: {e}")

validation\_result['valid'] = False

return validation\_result

def get\_statistics(self) -> Dict[str, any]:

stats = {

'road\_count': len(self.roads),

'total\_length': 0,

'geometry\_types': {},

'lane\_count': 0

}

for road in self.roads:

if hasattr(road, 'planview') and road.planview:

stats['total\_length'] += 100.0 # 默认长度

stats['geometry\_types']['Line'] = stats['geometry\_types'].get('Line', 0) + 1

if hasattr(road, 'lanes') and road.lanes:

stats['lane\_count'] += 2 # 假设每条道路有2条车道

return stats

import geopandas as gpd

import pandas as pd

from shapely.geometry import LineString, Point, MultiLineString

from typing import Dict, List, Tuple, Optional, Any

import logging

import os

from abc import ABC, abstractmethod

log\_dir = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'logs')

os.makedirs(log\_dir, exist\_ok=True)

log\_file = os.path.join(log\_dir, 'shp\_to\_opendrive.log')

if not logging.getLogger().handlers:

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler(log\_file, encoding='utf-8'),

]

)

logger = logging.getLogger(\_\_name\_\_)

try:

from geometry\_converter import ConversionHandler

except ImportError:

class ConversionHandler(ABC):

def \_\_init\_\_(self):

self.\_next\_handler = None

def set\_next(self, handler: 'ConversionHandler') -> 'ConversionHandler':

self.\_next\_handler = handler

return handler

@abstractmethod

def handle(self, request: Dict[str, Any]) -> Dict[str, Any]:

pass

def \_handle\_next(self, request: Dict[str, Any]) -> Dict[str, Any]:

if self.\_next\_handler:

return self.\_next\_handler.handle(request)

return request

class ShapefileReadingHandler(ConversionHandler):

def \_\_init\_\_(self, coordinate\_precision: int = 3):

super().\_\_init\_\_()

self.coordinate\_precision = coordinate\_precision

self.shp\_reader = None

def handle(self, request: Dict[str, Any]) -> Dict[str, Any]:

if request.get('stage') != 'shapefile\_loading':

return self.\_handle\_next(request)

shapefile\_path = request.get('shapefile\_path', '')

attribute\_mapping = request.get('attribute\_mapping', {})

config = request.get('config', {})

if not shapefile\_path:

request['success'] = False

request['error'] = '未提供shapefile路径'

return self.\_handle\_next(request)

self.shp\_reader = ShapefileReader(shapefile\_path, self.coordinate\_precision)

if not self.shp\_reader.load\_shapefile():

request['success'] = False

request['error'] = 'shapefile加载失败'

return self.\_handle\_next(request)

if not self.shp\_reader.convert\_to\_utm():

logger.warning("坐标系转换失败，继续使用原坐标系")

if not self.shp\_reader.convert\_to\_local\_coordinates():

logger.warning("局部坐标系转换失败，继续使用当前坐标系")

filtered\_count = self.shp\_reader.filter\_roads\_by\_length(

config.get('min\_road\_length', 1.0)

)

if filtered\_count == 0:

request['success'] = False

request['error'] = '过滤后没有剩余道路'

return self.\_handle\_next(request)

roads\_geometries = self.shp\_reader.extract\_road\_geometries()

if not roads\_geometries:

request['success'] = False

request['error'] = '提取道路数据失败'

return self.\_handle\_next(request)

if self.\_is\_lane\_format(roads\_geometries):

roads\_data = self.\_process\_lane\_data(roads\_geometries, attribute\_mapping)

else:

roads\_data = self.\_process\_traditional\_data(roads\_geometries, attribute\_mapping)

request['roads\_data'] = roads\_data

request['stage'] = 'geometry\_conversion'

return self.\_handle\_next(request)

def \_is\_lane\_format(self, roads\_geometries: List[Dict]) -> bool:

if not roads\_geometries:

return False

first\_road = roads\_geometries[0]

return all(key in first\_road for key in ['road\_id', 'lanes', 'lane\_surfaces'])

def \_process\_lane\_data(self, roads\_geometries: List[Dict], attribute\_mapping: Dict = None) -> List[Dict]:

roads\_data = []

for road\_geom in roads\_geometries:

road\_data = {

'id': road\_geom['road\_id'],

'type': 'lane\_based',

'lane\_surfaces': road\_geom.get('lane\_surfaces', []),

'attributes': self.\_extract\_lane\_attributes(road\_geom.get('lanes', []), attribute\_mapping)

}

roads\_data.append(road\_data)

return roads\_data

def \_process\_traditional\_data(self, roads\_geometries: List[Dict], attribute\_mapping: Dict = None) -> List[Dict]:

roads\_data = []

for road\_geom in roads\_geometries:

road\_data = {

'id': road\_geom['id'],

'type': 'traditional',

'coordinates': road\_geom['coordinates'],

'attributes': self.\_map\_attributes(road\_geom.get('attributes', {}), attribute\_mapping or {})

}

roads\_data.append(road\_data)

return roads\_data

def \_extract\_lane\_attributes(self, lanes: List[Dict], attribute\_mapping: Dict = None) -> Dict:

if not lanes:

return {}

first\_lane = lanes[0]

return self.\_map\_attributes(first\_lane.get('attributes', {}), attribute\_mapping or {})

def \_map\_attributes(self, original\_attrs: Dict, mapping: Dict) -> Dict:

mapped\_attrs = {}

for key, value in original\_attrs.items():

mapped\_key = mapping.get(key, key)

mapped\_attrs[mapped\_key] = value

return mapped\_attrs

class ShapefileReader:

def \_\_init\_\_(self, shapefile\_path: str, coordinate\_precision: int = 3):

self.shapefile\_path = shapefile\_path

self.coordinate\_precision = max(1, min(10, coordinate\_precision)) # 限制在1-10之间

self.gdf = None

self.roads\_data = []

self.lane\_data = {} # 存储按RoadID分组的车道数据

self.coordinate\_offset = {'x': 0.0, 'y': 0.0} # 存储坐标偏移量

def load\_shapefile(self) -> bool:

try:

self.gdf = gpd.read\_file(self.shapefile\_path)

logger.info(f"成功加载shapefile: {self.shapefile\_path}")

logger.info(f"包含 {len(self.gdf)} 条道路记录")

logger.info(f"坐标系统: {self.gdf.crs}")

return True

except Exception as e:

logger.error(f"加载shapefile失败: {e}")

return False

def get\_coordinate\_offset(self) -> Dict:

return self.coordinate\_offset.copy()

def get\_road\_info(self) -> Dict:

if self.gdf is None:

return {}

return {

'road\_count': len(self.gdf),

'crs': str(self.gdf.crs),

'bounds': self.gdf.total\_bounds.tolist(),

'columns': self.gdf.columns.tolist(),

'geometry\_types': self.gdf.geometry.geom\_type.unique().tolist()

}

def extract\_road\_geometries(self) -> List[Dict]:

if self.gdf is None:

logger.error("请先加载shapefile")

return []

if self.\_is\_lane\_shapefile():

return self.extract\_lane\_geometries()

roads = []

for idx, row in self.gdf.iterrows():

geometry = row.geometry

if not isinstance(geometry, LineString):

logger.warning(f"跳过非线性几何 (索引: {idx})")

continue

coords = [(coord[0], coord[1]) for coord in geometry.coords]

road\_info = {

'id': idx,

'geometry': geometry,

'coordinates': coords,

'length': geometry.length,

'start\_point': coords[0],

'end\_point': coords[-1],

'attributes': {}

}

for col in self.gdf.columns:

if col != 'geometry':

road\_info['attributes'][col] = row[col]

roads.append(road\_info)

self.roads\_data = roads

logger.info(f"提取了 {len(roads)} 条有效道路")

return roads

def \_is\_lane\_shapefile(self) -> bool:

if self.gdf is None:

return False

columns = [col.upper() for col in self.gdf.columns]

return 'ROADID' in columns and 'INDEX' in columns

def extract\_lane\_geometries(self) -> List[Dict]:

if self.gdf is None:

logger.error("请先加载shapefile")

return []

logger.info("检测到Lane.shp格式，开始提取车道数据")

grouped = self.gdf.groupby('RoadID')

roads = []

logger.info(f"开始处理 {len(grouped)} 个RoadID分组")

for road\_id, group in grouped:

logger.info(f"\n=== 处理RoadID: {road\_id} ===")

logger.info(f"该RoadID包含 {len(group)} 条边界线记录")

original\_indices = group['Index'].tolist()

logger.info(f"原始Index值: {original\_indices}")

try:

group\_sorted = group.sort\_values('Index', key=lambda x: x.astype(int))

except (ValueError, TypeError):

logger.warning(f"RoadID {road\_id} 的Index无法转换为整数，使用字符串排序")

group\_sorted = group.sort\_values('Index')

sorted\_indices = group\_sorted['Index'].tolist()

logger.info(f"排序后Index值: {sorted\_indices}")

boundary\_lines = []

for idx, row in group\_sorted.iterrows():

geometry = row.geometry

if isinstance(geometry, LineString):

coords = [(coord[0], coord[1]) for coord in geometry.coords]

boundary\_info = {

'index': str(row['Index']), # 保持为字符串，如"01", "12", "23"

'geometry': geometry,

'coordinates': coords,

'length': geometry.length,

'start\_point': coords[0],

'end\_point': coords[-1],

'attributes': {}

}

for col in self.gdf.columns:

if col != 'geometry':

boundary\_info['attributes'][col] = row[col]

boundary\_lines.append(boundary\_info)

logger.info(f" 添加边界线 Index={row['Index']}, 长度={geometry.length:.2f}m, 坐标点数={len(coords)}")

else:

logger.warning(f"跳过非线性几何 (RoadID: {road\_id}, Index: {row['Index']})")

logger.info(f"RoadID {road\_id} 共处理了 {len(boundary\_lines)} 条边界线")

lanes = self.\_build\_lanes\_from\_boundaries(boundary\_lines)

logger.info(f"RoadID {road\_id} 构建了 {len(lanes)} 个车道面")

road\_info = {

'road\_id': str(road\_id),

'lanes': lanes,

'lane\_count': len(lanes),

'lane\_surfaces': self.\_build\_lane\_surfaces(lanes)

}

roads.append(road\_info)

self.roads\_data = roads

logger.info(f"提取了 {len(roads)} 条道路，共 {sum(len(road['lanes']) for road in roads)} 条车道")

return roads

def \_build\_lanes\_from\_boundaries(self, boundary\_lines: List[Dict]) -> List[Dict]:

lanes = []

logger.info(f" 开始构建车道面，输入边界线数量: {len(boundary\_lines)}")

boundaries\_sorted = sorted(boundary\_lines, key=lambda x: x['index'])

sorted\_indices = [b['index'] for b in boundaries\_sorted]

logger.info(f" 边界线排序后Index顺序: {sorted\_indices}")

for i in range(len(boundaries\_sorted) - 1):

left\_boundary = boundaries\_sorted[i]

right\_boundary = boundaries\_sorted[i + 1]

surface\_id = f"{left\_boundary['index']}\_{right\_boundary['index']}"

logger.info(f" 构建车道面 {surface\_id}: 左边界Index={left\_boundary['index']}, 右边界Index={right\_boundary['index']}")

lane\_surface = {

'surface\_id': surface\_id,

'left\_boundary': {

'index': left\_boundary['index'],

'coordinates': left\_boundary['coordinates'],

'geometry': left\_boundary['geometry']

},

'right\_boundary': {

'index': right\_boundary['index'],

'coordinates': right\_boundary['coordinates'],

'geometry': right\_boundary['geometry']

},

'center\_line': self.\_calculate\_center\_line(

left\_boundary['coordinates'],

right\_boundary['coordinates']

),

'width\_profile': self.\_calculate\_width\_profile(

left\_boundary['coordinates'],

right\_boundary['coordinates']

),

'attributes': self.\_merge\_boundary\_attributes(

left\_boundary['attributes'],

right\_boundary['attributes']

)

}

lanes.append(lane\_surface)

logger.info(f" 车道面 {surface\_id} 构建完成，中心线点数={len(lane\_surface['center\_line'])}，宽度变化点数={len(lane\_surface['width\_profile'])}")

return lanes

def \_build\_lane\_surfaces(self, lanes: List[Dict]) -> List[Dict]:

return lanes

def get\_road\_attributes\_mapping(self) -> Dict[str, str]:

if self.gdf is None:

return {}

mapping\_suggestions = {}

columns = [col.lower() for col in self.gdf.columns if col != 'geometry']

for col in columns:

if any(keyword in col for keyword in ['type', 'class', 'category']):

mapping\_suggestions[col] = 'road\_type'

elif any(keyword in col for keyword in ['width', 'lane']):

mapping\_suggestions[col] = 'lane\_width'

elif any(keyword in col for keyword in ['speed', 'limit']):

mapping\_suggestions[col] = 'speed\_limit'

elif any(keyword in col for keyword in ['name', 'id']):

mapping\_suggestions[col] = 'road\_name'

return mapping\_suggestions

def \_calculate\_center\_line(self, left\_coords: List[tuple], right\_coords: List[tuple]) -> List[tuple]:

if len(left\_coords) != len(right\_coords):

min\_len = min(len(left\_coords), len(right\_coords))

left\_coords = left\_coords[:min\_len]

right\_coords = right\_coords[:min\_len]

center\_coords = []

for left\_pt, right\_pt in zip(left\_coords, right\_coords):

center\_x = (left\_pt[0] + right\_pt[0]) / 2

center\_y = (left\_pt[1] + right\_pt[1]) / 2

center\_coords.append((center\_x, center\_y))

return center\_coords

def \_calculate\_width\_profile(self, left\_coords: List[tuple], right\_coords: List[tuple]) -> List[float]:

import math

if len(left\_coords) != len(right\_coords):

min\_len = min(len(left\_coords), len(right\_coords))

left\_coords = left\_coords[:min\_len]

right\_coords = right\_coords[:min\_len]

widths = []

for left\_pt, right\_pt in zip(left\_coords, right\_coords):

width = math.sqrt((left\_pt[0] - right\_pt[0])\*\*2 + (left\_pt[1] - right\_pt[1])\*\*2)

width = round(width, self.coordinate\_precision)

widths.append(width)

return widths

def \_merge\_boundary\_attributes(self, left\_attrs: Dict, right\_attrs: Dict) -> Dict:

merged = {}

merged.update(left\_attrs)

for key, value in right\_attrs.items():

if key not in merged:

merged[key] = value

return merged

def convert\_to\_utm(self) -> bool:

if self.gdf is None:

return False

try:

if self.gdf.crs.is\_geographic:

bounds = self.gdf.total\_bounds

center\_lon = (bounds[0] + bounds[2]) / 2

utm\_zone = int((center\_lon + 180) / 6) + 1

center\_lat = (bounds[1] + bounds[3]) / 2

hemisphere = 'north' if center\_lat >= 0 else 'south'

utm\_crs = f"EPSG:{32600 + utm\_zone if hemisphere == 'north' else 32700 + utm\_zone}"

original\_crs = self.gdf.crs

logger.info(f"原始坐标系: {original\_crs}")

self.gdf = self.gdf.to\_crs(utm\_crs)

logger.info(f"坐标系已转换为: {utm\_crs}")

new\_bounds = self.gdf.total\_bounds

logger.info(f"转换后坐标范围: X[{new\_bounds[0]:.2f}, {new\_bounds[2]:.2f}], Y[{new\_bounds[1]:.2f}, {new\_bounds[3]:.2f}]")

else:

logger.info(f"当前坐标系已是投影坐标系: {self.gdf.crs}")

return True

except Exception as e:

logger.error(f"坐标系转换失败: {e}")

return False

def convert\_to\_local\_coordinates(self) -> bool:

if self.gdf is None:

return False

try:

bounds = self.gdf.total\_bounds

min\_x, min\_y = bounds[0], bounds[1]

self.coordinate\_offset = {'x': min\_x, 'y': min\_y}

logger.info(f"原点设置为: ({min\_x:.2f}, {min\_y:.2f})")

def translate\_geometry(geom):

if geom.geom\_type == 'LineString':

coords = [(x - min\_x, y - min\_y) for x, y in geom.coords]

return LineString(coords)

elif geom.geom\_type == 'MultiLineString':

lines = []

for line in geom.geoms:

coords = [(x - min\_x, y - min\_y) for x, y in line.coords]

lines.append(LineString(coords))

return MultiLineString(lines)

else:

return geom

self.gdf['geometry'] = self.gdf['geometry'].apply(translate\_geometry)

new\_bounds = self.gdf.total\_bounds

logger.info(f"局部坐标系范围: X[{new\_bounds[0]:.2f}, {new\_bounds[2]:.2f}], Y[{new\_bounds[1]:.2f}, {new\_bounds[3]:.2f}]")

return True

except Exception as e:

logger.error(f"局部坐标系转换失败: {e}")

return False

def filter\_roads\_by\_length(self, min\_length: float = 1.0) -> int:

if self.gdf is None:

return 0

original\_count = len(self.gdf)

self.gdf = self.gdf[self.gdf.geometry.length >= min\_length]

filtered\_count = len(self.gdf)

logger.info(f"长度过滤: {original\_count} -> {filtered\_count} 条道路")

return filtered\_count

def extract\_roads\_data(self) -> List[Dict]:

return self.extract\_road\_geometries()

def get\_sample\_data(self, n: int = 5) -> List[Dict]:

if not self.roads\_data:

self.extract\_road\_geometries()

return self.roads\_data[:n]

def read\_features(self) -> List[Dict]:

if not self.load\_shapefile():

logger.error("无法加载shapefile文件")

return []

self.convert\_to\_utm()

roads = self.extract\_road\_geometries()

logger.info(f"成功读取 {len(roads)} 个道路特征")

return roads

def get\_bounds(self) -> Dict[str, float]:

if self.gdf is None:

if not self.load\_shapefile():

return {'minX': 0, 'minY': 0, 'maxX': 0, 'maxY': 0}

bounds = self.gdf.total\_bounds

return {

'minX': float(bounds[0]),

'minY': float(bounds[1]),

'maxX': float(bounds[2]),

'maxY': float(bounds[3])

}

def get\_center(self) -> Dict[str, float]:

bounds = self.get\_bounds()

return {

'x': (bounds['minX'] + bounds['maxX']) / 2,

'y': (bounds['minY'] + bounds['maxY']) / 2

}

import tkinter as tk

from tkinter import ttk, filedialog, messagebox, scrolledtext

import os

import threading

import json

from pathlib import Path

import sys

sys.path.append(os.path.dirname(os.path.abspath(\_\_file\_\_)))

from shp2xodr import ChainOfResponsibilityConverter

class ShpToOpenDriveGUI:

def \_\_init\_\_(self, root):

self.root = root

self.setup\_window()

self.create\_widgets()

self.converter = None

def setup\_window(self):

self.root.title("Shapefile转OpenDrive格式转换器")

self.root.geometry("800x700")

self.root.resizable(True, True)

self.colors = {

'bg': '#f5f5f7', # 浅灰背景

'card\_bg': '#ffffff', # 白色卡片背景

'border': '#d2d2d7', # 边框颜色

'text': '#1d1d1f', # 深色文字

'secondary\_text': '#86868b', # 次要文字

'accent': '#007aff', # 蓝色强调色

'success': '#34c759', # 成功绿色

'error': '#ff3b30' # 错误红色

}

self.root.configure(bg=self.colors['bg'])

def create\_widgets(self):

main\_frame = ttk.Frame(self.root, padding="20")

main\_frame.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

self.root.columnconfigure(0, weight=1)

self.root.rowconfigure(0, weight=1)

main\_frame.columnconfigure(0, weight=1)

title\_label = ttk.Label(main\_frame, text="Shapefile转OpenDrive格式转换器",

font=('SF Pro Display', 24, 'bold'))

title\_label.grid(row=0, column=0, pady=(0, 30), sticky=tk.W)

self.create\_file\_selection\_frame(main\_frame, 1)

self.create\_config\_frame(main\_frame, 2)

self.create\_control\_frame(main\_frame, 3)

self.create\_log\_frame(main\_frame, 4)

main\_frame.rowconfigure(4, weight=1)

def create\_file\_selection\_frame(self, parent, row):

file\_frame = ttk.LabelFrame(parent, text="文件选择", padding="15")

file\_frame.grid(row=row, column=0, sticky=(tk.W, tk.E), pady=(0, 20))

file\_frame.columnconfigure(1, weight=1)

ttk.Label(file\_frame, text="输入Shapefile:").grid(row=0, column=0, sticky=tk.W, pady=(0, 10))

self.input\_var = tk.StringVar()

input\_entry = ttk.Entry(file\_frame, textvariable=self.input\_var, width=50)

input\_entry.grid(row=0, column=1, sticky=(tk.W, tk.E), padx=(10, 10), pady=(0, 10))

ttk.Button(file\_frame, text="浏览", command=self.browse\_input\_file).grid(row=0, column=2, pady=(0, 10))

ttk.Label(file\_frame, text="输出文件:").grid(row=1, column=0, sticky=tk.W)

self.output\_var = tk.StringVar()

output\_entry = ttk.Entry(file\_frame, textvariable=self.output\_var, width=50)

output\_entry.grid(row=1, column=1, sticky=(tk.W, tk.E), padx=(10, 10))

ttk.Button(file\_frame, text="浏览", command=self.browse\_output\_file).grid(row=1, column=2)

def create\_config\_frame(self, parent, row):

config\_frame = ttk.LabelFrame(parent, text="转换参数", padding="15")

config\_frame.grid(row=row, column=0, sticky=(tk.W, tk.E), pady=(0, 20))

config\_frame.columnconfigure(1, weight=1)

ttk.Label(config\_frame, text="几何拟合容差 (米):").grid(row=0, column=0, sticky=tk.W, pady=(0, 10))

self.tolerance\_var = tk.DoubleVar(value=1.0)

tolerance\_spin = ttk.Spinbox(config\_frame, from\_=0.1, to=10.0, increment=0.1,

textvariable=self.tolerance\_var, width=10)

tolerance\_spin.grid(row=0, column=1, sticky=tk.W, padx=(10, 0), pady=(0, 10))

ttk.Label(config\_frame, text="最小道路长度 (米):").grid(row=1, column=0, sticky=tk.W, pady=(0, 10))

self.min\_length\_var = tk.DoubleVar(value=1.0)

min\_length\_spin = ttk.Spinbox(config\_frame, from\_=0.1, to=100.0, increment=0.1,

textvariable=self.min\_length\_var, width=10)

min\_length\_spin.grid(row=1, column=1, sticky=tk.W, padx=(10, 0), pady=(0, 10))

self.use\_arcs\_var = tk.BooleanVar()

arcs\_check = ttk.Checkbutton(config\_frame, text="使用圆弧拟合", variable=self.use\_arcs\_var)

arcs\_check.grid(row=2, column=0, columnspan=2, sticky=tk.W, pady=(0, 10))

ttk.Label(config\_frame, text="配置文件 (可选):").grid(row=3, column=0, sticky=tk.W)

self.config\_var = tk.StringVar()

config\_entry = ttk.Entry(config\_frame, textvariable=self.config\_var, width=40)

config\_entry.grid(row=3, column=1, sticky=(tk.W, tk.E), padx=(10, 10))

ttk.Button(config\_frame, text="浏览", command=self.browse\_config\_file).grid(row=3, column=2)

def create\_control\_frame(self, parent, row):

control\_frame = ttk.Frame(parent)

control\_frame.grid(row=row, column=0, sticky=(tk.W, tk.E), pady=(0, 20))

control\_frame.columnconfigure(0, weight=1)

self.convert\_button = ttk.Button(control\_frame, text="开始转换",

command=self.start\_conversion, style='Accent.TButton')

self.convert\_button.grid(row=0, column=0, pady=10)

self.progress\_var = tk.StringVar(value="准备就绪")

progress\_label = ttk.Label(control\_frame, textvariable=self.progress\_var)

progress\_label.grid(row=1, column=0, pady=(0, 10))

self.progress\_bar = ttk.Progressbar(control\_frame, mode='indeterminate')

self.progress\_bar.grid(row=2, column=0, sticky=(tk.W, tk.E), pady=(0, 10))

def create\_log\_frame(self, parent, row):

log\_frame = ttk.LabelFrame(parent, text="转换日志", padding="15")

log\_frame.grid(row=row, column=0, sticky=(tk.W, tk.E, tk.N, tk.S), pady=(0, 0))

log\_frame.columnconfigure(0, weight=1)

log\_frame.rowconfigure(0, weight=1)

self.log\_text = scrolledtext.ScrolledText(log\_frame, height=15, width=80,

wrap=tk.WORD, state=tk.DISABLED)

self.log\_text.grid(row=0, column=0, sticky=(tk.W, tk.E, tk.N, tk.S))

ttk.Button(log\_frame, text="清除日志", command=self.clear\_log).grid(row=1, column=0, pady=(10, 0))

def browse\_input\_file(self):

filename = filedialog.askopenfilename(

title="选择输入Shapefile",

filetypes=[("Shapefile", "\*.shp"), ("所有文件", "\*.\*")]

)

if filename:

self.input\_var.set(filename)

if not self.output\_var.get():

output\_dir = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(filename))), "output")

os.makedirs(output\_dir, exist\_ok=True)

output\_file = os.path.join(output\_dir,

os.path.splitext(os.path.basename(filename))[0] + ".xodr")

self.output\_var.set(output\_file)

def browse\_output\_file(self):

filename = filedialog.asksaveasfilename(

title="选择输出文件",

defaultextension=".xodr",

filetypes=[("OpenDrive文件", "\*.xodr"), ("所有文件", "\*.\*")]

)

if filename:

self.output\_var.set(filename)

def browse\_config\_file(self):

filename = filedialog.askopenfilename(

title="选择配置文件",

filetypes=[("JSON文件", "\*.json"), ("所有文件", "\*.\*")]

)

if filename:

self.config\_var.set(filename)

def log\_message(self, message, level="INFO"):

self.log\_text.config(state=tk.NORMAL)

timestamp = tk.datetime.datetime.now().strftime("%H:%M:%S")

formatted\_message = f"[{timestamp}] {level}: {message}\n"

self.log\_text.insert(tk.END, formatted\_message)

self.log\_text.see(tk.END)

self.log\_text.config(state=tk.DISABLED)

self.root.update\_idletasks()

def clear\_log(self):

self.log\_text.config(state=tk.NORMAL)

self.log\_text.delete(1.0, tk.END)

self.log\_text.config(state=tk.DISABLED)

def validate\_inputs(self):

if not self.input\_var.get():

messagebox.showerror("错误", "请选择输入Shapefile文件")

return False

if not os.path.exists(self.input\_var.get()):

messagebox.showerror("错误", "输入文件不存在")

return False

if not self.output\_var.get():

messagebox.showerror("错误", "请指定输出文件路径")

return False

output\_dir = os.path.dirname(self.output\_var.get())

if output\_dir and not os.path.exists(output\_dir):

try:

os.makedirs(output\_dir, exist\_ok=True)

except Exception as e:

messagebox.showerror("错误", f"无法创建输出目录: {e}")

return False

return True

def start\_conversion(self):

if not self.validate\_inputs():

return

self.convert\_button.config(state=tk.DISABLED)

self.progress\_bar.start()

self.progress\_var.set("转换中...")

conversion\_thread = threading.Thread(target=self.run\_conversion)

conversion\_thread.daemon = True

conversion\_thread.start()

def run\_conversion(self):

try:

self.log\_message("开始转换过程")

config = {

'geometry\_tolerance': self.tolerance\_var.get(),

'min\_road\_length': self.min\_length\_var.get(),

'use\_arc\_fitting': self.use\_arcs\_var.get()

}

if self.config\_var.get() and os.path.exists(self.config\_var.get()):

try:

with open(self.config\_var.get(), 'r', encoding='utf-8') as f:

file\_config = json.load(f)

config.update(file\_config)

self.log\_message(f"已加载配置文件: {self.config\_var.get()}")

except Exception as e:

self.log\_message(f"配置文件加载失败: {e}", "WARNING")

self.converter = ChainOfResponsibilityConverter(config)

success = self.converter.convert(

self.input\_var.get(),

self.output\_var.get()

)

self.root.after(0, self.conversion\_completed, success)

except Exception as e:

self.root.after(0, self.conversion\_error, str(e))

def conversion\_completed(self, success):

self.progress\_bar.stop()

self.convert\_button.config(state=tk.NORMAL)

if success:

self.progress\_var.set("转换成功完成")

self.log\_message("转换成功完成！", "SUCCESS")

if self.converter and hasattr(self.converter, 'conversion\_stats'):

stats = self.converter.conversion\_stats

self.log\_message(f"输入道路数: {stats.get('input\_roads', 0)}")

self.log\_message(f"输出道路数: {stats.get('output\_roads', 0)}")

self.log\_message(f"总长度: {stats.get('total\_length', 0):.2f} 米")

self.log\_message(f"转换时间: {stats.get('conversion\_time', 0):.2f} 秒")

messagebox.showinfo("成功", "转换成功完成！")

else:

self.progress\_var.set("转换失败")

self.log\_message("转换失败！", "ERROR")

messagebox.showerror("失败", "转换失败，请检查日志获取详细信息")

def conversion\_error(self, error\_message):

self.progress\_bar.stop()

self.convert\_button.config(state=tk.NORMAL)

self.progress\_var.set("转换出错")

self.log\_message(f"转换出错: {error\_message}", "ERROR")

messagebox.showerror("错误", f"转换过程出错: {error\_message}")

def main():

import datetime

tk.datetime = datetime

root = tk.Tk()

app = ShpToOpenDriveGUI(root)

root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

main()

import os

import logging

from typing import Dict, List, Optional

import json

from pathlib import Path

from shp\_reader import ShapefileReader, ShapefileReadingHandler

from geometry\_converter import GeometryConverter, GeometryConversionHandler

from opendrive\_generator import OpenDriveGenerator, OpenDriveGenerationHandler

import time

log\_dir = os.path.join(os.path.dirname(os.path.dirname(\_\_file\_\_)), 'logs')

os.makedirs(log\_dir, exist\_ok=True)

log\_file = os.path.join(log\_dir, 'shp\_to\_opendrive.log')

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler(log\_file, encoding='utf-8'),

logging.StreamHandler()

]

)

logger = logging.getLogger(\_\_name\_\_)

class ChainOfResponsibilityConverter:

def \_\_init\_\_(self, config: Dict = None):

self.config = config or {}

self.conversion\_stats = {

'input\_roads': 0,

'output\_roads': 0,

'conversion\_time': 0.0,

'errors': [],

'warnings': []

}

self.\_setup\_chain()

def \_setup\_chain(self):

self.shapefile\_handler = ShapefileReadingHandler(

coordinate\_precision=self.config.get('coordinate\_precision', 3)

)

self.geometry\_handler = GeometryConversionHandler(

tolerance=self.config.get('geometry\_tolerance', 1.0),

smooth\_curves=self.config.get('use\_smooth\_curves', True),

preserve\_detail=self.config.get('preserve\_detail', True),

curve\_fitting\_mode=self.config.get('curve\_fitting\_mode', 'parampoly3'),

polynomial\_degree=self.config.get('polynomial\_degree', 3),

curve\_smoothness=self.config.get('curve\_smoothness', 0.5),

coordinate\_precision=self.config.get('coordinate\_precision', 3)

)

self.opendrive\_handler = OpenDriveGenerationHandler(

name=self.config.get('name', 'ConvertedRoad'),

curve\_fitting\_mode=self.config.get('curve\_fitting\_mode', 'polyline'),

polynomial\_degree=self.config.get('polynomial\_degree', 3),

curve\_smoothness=self.config.get('curve\_smoothness', 0.5)

)

self.shapefile\_handler.set\_next(self.geometry\_handler).set\_next(self.opendrive\_handler)

def convert(self, shapefile\_path: str, output\_path: str, attribute\_mapping: Dict = None) -> bool:

try:

start\_time = time.time()

request = {

'stage': 'shapefile\_loading',

'shapefile\_path': shapefile\_path,

'output\_path': output\_path,

'attribute\_mapping': attribute\_mapping or {},

'config': self.config

}

result = self.shapefile\_handler.handle(request)

self.conversion\_stats['conversion\_time'] = time.time() - start\_time

if result.get('success', False):

self.conversion\_stats['output\_roads'] = len(result.get('road\_ids', []))

logger.info("转换完成！")

return True

else:

error\_msg = result.get('error', '未知错误')

logger.error(f"转换失败: {error\_msg}")

self.conversion\_stats['errors'].append(error\_msg)

return False

except Exception as e:

logger.error(f"转换过程出错: {e}")

self.conversion\_stats['errors'].append(str(e))

return False

def get\_conversion\_stats(self) -> Dict:

return self.conversion\_stats.copy()

def save\_conversion\_report(self, report\_path: str):

try:

report = {

'config': self.config,

'statistics': self.conversion\_stats,

'timestamp': time.strftime('%Y-%m-%d %H:%M:%S')

}

with open(report\_path, 'w', encoding='utf-8') as f:

json.dump(report, f, indent=2, ensure\_ascii=False)

logger.info(f"转换报告已保存: {report\_path}")

except Exception as e:

logger.error(f"保存转换报告失败: {e}")

class ShpToOpenDriveConverter:

def \_\_init\_\_(self, config: Dict = None):

self.config = {

'geometry\_tolerance': 1.0, # 几何拟合容差（米）

'min\_road\_length': 1.0, # 最小道路长度（米）

'default\_lane\_width': 3.5, # 默认车道宽度（米）

'default\_num\_lanes': 1, # 默认车道数

'default\_speed\_limit': 50, # 默认限速（km/h）

'use\_arc\_fitting': False, # 是否使用圆弧拟合

'coordinate\_precision': 3, # 坐标精度（小数位数）

'use\_smooth\_curves': True, # 是否使用平滑曲线拟合

'preserve\_detail': True, # 是否保留更多细节

'curve\_fitting\_mode': 'parampoly3', # 曲线拟合模式

'polynomial\_degree': 3, # 多项式拟合阶数

'curve\_smoothness': 0.5, # 曲线平滑度

'lane\_format\_settings': {

'enabled': True,

'road\_id\_field': 'RoadID',

'index\_field': 'Index',

'auto\_detect\_lane\_format': True,

'lane\_surface\_generation': {

'enabled': True,

'width\_interpolation': 'linear',

'center\_line\_calculation': 'midpoint',

'width\_sampling\_points': 50

},

'validation': {

'check\_index\_continuity': True,

'min\_lanes\_per\_road': 2,

'max\_width\_variation': 10.0

}

}

}

if config:

self.config.update(config)

self.shp\_reader = None

geometry\_tolerance = self.config['geometry\_tolerance']

if self.config.get('lane\_format\_settings', {}).get('enabled', False):

geometry\_tolerance = min(geometry\_tolerance, 0.3)

self.geometry\_converter = GeometryConverter(

tolerance=geometry\_tolerance,

smooth\_curves=self.config.get('use\_smooth\_curves', True),

preserve\_detail=self.config.get('preserve\_detail', True),

curve\_fitting\_mode=self.config.get('curve\_fitting\_mode', 'parampoly3'),

polynomial\_degree=self.config.get('polynomial\_degree', 3),

curve\_smoothness=self.config.get('curve\_smoothness', 0.5),

coordinate\_precision=self.config.get('coordinate\_precision', 3)

)

self.opendrive\_generator = None

self.conversion\_stats = {

'input\_roads': 0,

'output\_roads': 0,

'total\_length': 0,

'conversion\_time': 0,

'errors': [],

'warnings': []

}

def convert(self, shapefile\_path: str, output\_path: str,

attribute\_mapping: Dict = None) -> bool:

import time

start\_time = time.time()

try:

logger.info("开始Shapefile到OpenDrive转换")

logger.info(f"输入文件: {shapefile\_path}")

logger.info(f"输出文件: {output\_path}")

if not self.\_load\_shapefile(shapefile\_path):

return False

roads\_data = self.\_extract\_roads\_data(attribute\_mapping)

if not roads\_data:

logger.error("没有提取到有效的道路数据")

return False

converted\_roads = self.\_convert\_geometries(roads\_data)

if not converted\_roads:

logger.error("几何转换失败")

return False

if not self.\_generate\_opendrive(converted\_roads, output\_path):

return False

self.conversion\_stats['conversion\_time'] = time.time() - start\_time

self.\_log\_conversion\_stats()

logger.info("转换完成！")

return True

except Exception as e:

logger.error(f"转换过程出错: {e}")

self.conversion\_stats['errors'].append(str(e))

return False

def \_load\_shapefile(self, shapefile\_path: str) -> bool:

try:

self.shp\_reader = ShapefileReader(shapefile\_path, self.config.get('coordinate\_precision', 3))

if not self.shp\_reader.load\_shapefile():

return False

if not self.shp\_reader.convert\_to\_utm():

logger.warning("坐标系转换失败，继续使用原坐标系")

if not self.shp\_reader.convert\_to\_local\_coordinates():

logger.warning("局部坐标系转换失败，继续使用当前坐标系")

filtered\_count = self.shp\_reader.filter\_roads\_by\_length(

self.config['min\_road\_length']

)

if filtered\_count == 0:

logger.error("过滤后没有剩余道路")

return False

road\_info = self.shp\_reader.get\_road\_info()

self.conversion\_stats['input\_roads'] = road\_info['road\_count']

logger.info(f"成功加载 {road\_info['road\_count']} 条道路")

return True

except Exception as e:

logger.error(f"加载shapefile失败: {e}")

return False

def \_extract\_roads\_data(self, attribute\_mapping: Dict = None) -> List[Dict]:

try:

roads\_geometries = self.shp\_reader.extract\_road\_geometries()

if not roads\_geometries:

return []

if self.\_is\_lane\_format(roads\_geometries):

return self.\_process\_lane\_data(roads\_geometries, attribute\_mapping)

else:

return self.\_process\_traditional\_data(roads\_geometries, attribute\_mapping)

except Exception as e:

logger.error(f"提取道路数据失败: {e}")

return []

def \_is\_lane\_format(self, roads\_geometries: List[Dict]) -> bool:

if not roads\_geometries:

return False

first\_road = roads\_geometries[0]

return all(key in first\_road for key in ['road\_id', 'lanes', 'lane\_surfaces'])

def \_process\_lane\_data(self, roads\_geometries: List[Dict],

attribute\_mapping: Dict = None) -> List[Dict]:

logger.info("处理Lane.shp格式数据")

roads\_data = []

for road\_geom in roads\_geometries:

road\_id = road\_geom['road\_id']

lanes = road\_geom['lanes'] # 现在lanes就是车道面

lane\_surfaces = road\_geom['lane\_surfaces'] # 与lanes相同

total\_length = 0

if lanes:

center\_coords = lanes[0].get('center\_line', [])

if center\_coords:

total\_length = self.\_calculate\_line\_length(center\_coords)

road\_data = {

'id': road\_id,

'type': 'lane\_based', # 标识为基于车道的道路

'lanes': lanes,

'lane\_surfaces': lane\_surfaces,

'lane\_count': len(lanes),

'length': total\_length,

'attributes': self.\_extract\_lane\_attributes(lanes, attribute\_mapping)

}

roads\_data.append(road\_data)

logger.info(f"处理了 {len(roads\_data)} 条基于车道的道路")

return roads\_data

def \_calculate\_line\_length(self, coords: List[tuple]) -> float:

import math

if len(coords) < 2:

return 0.0

total\_length = 0.0

for i in range(len(coords) - 1):

x1, y1 = coords[i]

x2, y2 = coords[i + 1]

segment\_length = math.sqrt((x2 - x1)\*\*2 + (y2 - y1)\*\*2)

total\_length += segment\_length

return total\_length

def \_process\_traditional\_data(self, roads\_geometries: List[Dict],

attribute\_mapping: Dict = None) -> List[Dict]:

logger.info("处理传统格式道路数据")

if attribute\_mapping is None:

attribute\_mapping = self.shp\_reader.get\_road\_attributes\_mapping()

roads\_data = []

for road\_geom in roads\_geometries:

road\_data = {

'id': road\_geom['id'],

'type': 'traditional', # 标识为传统道路

'coordinates': road\_geom['coordinates'],

'length': road\_geom['length'],

'attributes': self.\_map\_attributes(

road\_geom['attributes'],

attribute\_mapping

)

}

roads\_data.append(road\_data)

logger.info(f"处理了 {len(roads\_data)} 条传统道路")

return roads\_data

def \_extract\_lane\_attributes(self, lanes: List[Dict],

attribute\_mapping: Dict = None) -> Dict:

if not lanes:

return {}

first\_lane\_attrs = lanes[0].get('attributes', {})

if attribute\_mapping:

return self.\_map\_attributes(first\_lane\_attrs, attribute\_mapping)

else:

return first\_lane\_attrs

def \_map\_attributes(self, original\_attrs: Dict, mapping: Dict) -> Dict:

mapped\_attrs = {

'lane\_width': self.config['default\_lane\_width'],

'num\_lanes': self.config['default\_num\_lanes'],

'speed\_limit': self.config['default\_speed\_limit'],

'road\_type': 'urban',

'bidirectional': False

}

for original\_key, mapped\_key in mapping.items():

if original\_key in original\_attrs:

value = original\_attrs[original\_key]

if mapped\_key == 'lane\_width':

try:

mapped\_attrs['lane\_width'] = float(value)

except (ValueError, TypeError):

pass

elif mapped\_key == 'num\_lanes':

try:

mapped\_attrs['num\_lanes'] = int(value)

except (ValueError, TypeError):

pass

elif mapped\_key == 'speed\_limit':

try:

mapped\_attrs['speed\_limit'] = float(value)

except (ValueError, TypeError):

pass

else:

mapped\_attrs[mapped\_key] = value

return mapped\_attrs

def \_convert\_geometries(self, roads\_data: List[Dict]) -> List[Dict]:

try:

converted\_roads = []

for road\_data in roads\_data:

if road\_data.get('type') == 'lane\_based':

converted\_road = self.\_convert\_lane\_based\_geometry(road\_data)

else:

converted\_road = self.\_convert\_traditional\_geometry(road\_data)

if converted\_road:

converted\_roads.append(converted\_road)

self.conversion\_stats['total\_length'] += converted\_road['total\_length']

logger.info(f"成功转换 {len(converted\_roads)} 条道路的几何")

return converted\_roads

except Exception as e:

logger.error(f"几何转换失败: {e}")

return []

def \_convert\_lane\_based\_geometry(self, road\_data: Dict) -> Dict:

try:

road\_id = road\_data['id']

lane\_surfaces = road\_data['lane\_surfaces']

converted\_surfaces = self.geometry\_converter.convert\_lane\_surface\_geometry(lane\_surfaces)

if not converted\_surfaces:

logger.warning(f"道路 {road\_id} 车道面几何转换失败")

return None

total\_length = 0

if converted\_surfaces:

first\_surface = converted\_surfaces[0]

total\_length = self.geometry\_converter.calculate\_road\_length(

first\_surface['center\_segments']

)

converted\_road = {

'id': road\_id,

'type': 'lane\_based',

'lane\_surfaces': converted\_surfaces,

'lanes': road\_data['lanes'],

'lane\_count': road\_data['lane\_count'],

'attributes': road\_data['attributes'],

'total\_length': total\_length

}

logger.info(f"成功转换基于车道的道路 {road\_id}，包含 {len(converted\_surfaces)} 个车道面")

return converted\_road

except Exception as e:

logger.error(f"Lane格式道路 {road\_data.get('id', 'unknown')} 几何转换失败: {e}")

return None

def \_convert\_traditional\_geometry(self, road\_data: Dict) -> Dict:

try:

coordinates = road\_data['coordinates']

if self.config.get('use\_smooth\_curves', True):

segments = self.geometry\_converter.convert\_road\_geometry(coordinates)

elif self.config['use\_arc\_fitting']:

segments = self.geometry\_converter.fit\_arc\_segments(coordinates)

else:

segments = self.geometry\_converter.fit\_line\_segments(coordinates)

if not segments:

logger.warning(f"道路 {road\_data['id']} 几何转换失败")

return None

if not self.geometry\_converter.validate\_geometry\_continuity(segments):

logger.warning(f"道路 {road\_data['id']} 几何不连续")

self.conversion\_stats['warnings'].append(

f"道路 {road\_data['id']} 几何不连续"

)

total\_length = self.geometry\_converter.calculate\_road\_length(segments)

converted\_road = {

'id': road\_data['id'],

'type': 'traditional',

'segments': segments,

'attributes': road\_data['attributes'],

'total\_length': total\_length

}

return converted\_road

except Exception as e:

logger.error(f"传统格式道路 {road\_data.get('id', 'unknown')} 几何转换失败: {e}")

return None

def \_extract\_segments\_from\_lane\_surfaces(self, lane\_surfaces: List[Dict]) -> List[Dict]:

if not lane\_surfaces:

return []

first\_surface = lane\_surfaces[0]

if 'center\_segments' in first\_surface:

return first\_surface['center\_segments']

logger.warning("车道面缺少center\_segments，尝试从边界生成")

return []

def \_generate\_opendrive(self, converted\_roads: List[Dict], output\_path: str) -> bool:

try:

road\_network\_name = Path(output\_path).stem

self.opendrive\_generator = OpenDriveGenerator(

name=road\_network\_name,

curve\_fitting\_mode=self.config.get('curve\_fitting\_mode', 'polyline'),

polynomial\_degree=self.config.get('polynomial\_degree', 3),

curve\_smoothness=self.config.get('curve\_smoothness', 0.5)

)

road\_ids = []

for road\_data in converted\_roads:

if road\_data['type'] == 'lane\_based':

road\_id = self.opendrive\_generator.create\_road\_from\_lane\_surfaces(

road\_data['lane\_surfaces'],

road\_data['attributes']

)

else:

segments = road\_data['segments']

road\_id = self.opendrive\_generator.create\_road\_from\_segments(

segments,

road\_data['attributes']

)

if road\_id > 0:

road\_ids.append(road\_id)

if not road\_ids:

logger.error("没有成功创建任何道路")

return False

validation\_result = self.opendrive\_generator.validate\_opendrive()

if not validation\_result['valid']:

logger.error(f"OpenDrive验证失败: {validation\_result['errors']}")

return False

if not self.opendrive\_generator.generate\_file(output\_path):

return False

self.conversion\_stats['output\_roads'] = len(road\_ids)

stats = self.opendrive\_generator.get\_statistics()

logger.info(f"OpenDrive统计: {stats}")

return True

except Exception as e:

logger.error(f"生成OpenDrive文件失败: {e}")

return False

def \_log\_conversion\_stats(self):

stats = self.conversion\_stats

logger.info("=== 转换统计 ===")

logger.info(f"输入道路数: {stats['input\_roads']}")

logger.info(f"输出道路数: {stats['output\_roads']}")

logger.info(f"总长度: {stats['total\_length']:.2f} 米")

logger.info(f"转换时间: {stats['conversion\_time']:.2f} 秒")

if stats['warnings']:

logger.info(f"警告数: {len(stats['warnings'])}")

for warning in stats['warnings'][:5]: # 只显示前5个警告

logger.warning(warning)

if stats['errors']:

logger.info(f"错误数: {len(stats['errors'])}")

for error in stats['errors']:

logger.error(error)

def save\_conversion\_report(self, report\_path: str):

try:

report = {

'config': self.config,

'statistics': self.conversion\_stats,

'timestamp': time.strftime('%Y-%m-%d %H:%M:%S')

}

with open(report\_path, 'w', encoding='utf-8') as f:

json.dump(report, f, indent=2, ensure\_ascii=False)

logger.info(f"转换报告已保存: {report\_path}")

except Exception as e:

logger.error(f"保存转换报告失败: {e}")

def get\_conversion\_stats(self) -> Dict:

return self.conversion\_stats.copy()

def main():

import argparse

parser = argparse.ArgumentParser(description='将Shapefile转换为OpenDrive格式')

parser.add\_argument('input', help='输入shapefile路径')

parser.add\_argument('output', help='输出OpenDrive文件路径')

parser.add\_argument('--config', help='配置文件路径（JSON格式）')

parser.add\_argument('--tolerance', type=float, default=1.0, help='几何拟合容差（米）')

parser.add\_argument('--min-length', type=float, default=1.0, help='最小道路长度（米）')

parser.add\_argument('--use-arcs', action='store\_true', help='使用圆弧拟合')

parser.add\_argument('--report', help='转换报告输出路径')

args = parser.parse\_args()

config = {}

if args.config and os.path.exists(args.config):

with open(args.config, 'r', encoding='utf-8') as f:

config = json.load(f)

config.update({

'geometry\_tolerance': args.tolerance,

'min\_road\_length': args.min\_length,

'use\_arc\_fitting': args.use\_arcs

})

converter = ChainOfResponsibilityConverter(config)

success = converter.convert(args.input, args.output)

if args.report:

converter.save\_conversion\_report(args.report)

if success:

print("转换成功完成！")

return 0

else:

print("转换失败！")

return 1

if \_\_name\_\_ == '\_\_main\_\_':

import sys

sys.exit(main())