为 AI 编程助手准备的精准开发上下文：在 Optiland 中实现 GRIN 功能

1. 任务目标概述 (Task Objective)

核心目标是将梯度折射率 (GRIN) 透镜的建模与光线追迹功能集成到 optiland Python 库中。此功能对于将 optiland 的应用范围拓展至生物光学（如人眼建模）等前沿领域至关重要，是项目发展的一个关键战略步骤。

2. 受影响的核心模块与文件 (Impact Analysis)

本次实现严格遵循设计文档中强调的公理化设计 **(Axiomatic Design)** 原则，通过创建三个独立的、职责清晰的新模块（分别负责几何、物理和行为），并将它们集成到现有的核心追迹引擎中。这种解耦方法确保了新功能的可维护性和未来扩展性。

下表总结了为完成此任务所需创建和修改的所有文件，清晰地界定了本次“手术刀式”操作的范围。

文件路径	状态	在 GRIN 实现中的角色	关键交互组件
optiland/optic.py	修改	核心集成器。其 trace 循环将被更新，以识别 GRIN 区域并分派光线至新的传播逻辑。	GradientBoundarySurface, GradientMaterial, propagate_through_gradient
optiland/surfaces/gradient_surface.py	新增	定义几何“标记” (GradientBoundary	继承自 surfaces.standard_

		Surface)，用于标识 GRIN 介质的入口和出口。	surface.Surface
optiland/materials/gradient_material.py	新增	定义 GRIN 介质的物理模型 (GradientMaterial)，提供随空间变化的折射率计算。	继承自 materials.base.BaseMaterial
optiland/interactions/gradient_propagation.py	新增	实现光线在 GRIN 介质中传播的数值积分算法 (propagate_through_gradient)。	rays.real_rays.RealRays, GradientMaterial
optiland/surfaces/standard_surface.py	理解/调用	提供 Surface 基类。理解其 _trace_real 方法对于正确集成至关重要。	GradientBoundarySurface 的父类
optiland/materials/base.py	理解/调用	提供 BaseMaterial 接口，GradientMaterial 必须遵循此接口契约。	GradientMaterial 的父类
optiland/rays/real_rays.py	理解/调用	定义 RealRays 数据结构，这是被传播算法所操作的状态向量。	propagate_through_gradient 的输入/输出

3. 精准上下文详情 (Detailed Context)

本节对需要修改或理解的现有文件提供深入的、目标明确的上下文。

模块/文件 A (修改): optiland/optic.py

- 与任务的关联: 此文件包含 Optic 类，它是用户构建和追迹光学系统的主要入口 [1]。其核心的 trace 方法负责协调整个序列光线追迹流程。我们的任务是对该方法的主循环进行一次精确的外科手术式干预，增加一个新的条件分支来处理 GRIN 介质内的特殊传播逻辑。现有的追迹循环是一个典型的多态分派器，它按顺序迭代表面列表，并调用每个表面的 trace 方法，而无需关心表面的具体类型（球面、非球面等）。GRIN 功能引入了一种根本不同的传播模型，它发生在两个表面之间，而非在单个表面上。因此，必须升级这个分派器，使其能够识别 GRIN 区域的“入口标记”，暂停标准循环，将控制权交给专门的 GRIN 传播函数，并在完成后能正确地快进循环计数器，从而在不破坏现有架构优雅性的前提下融入新功能。
- 相关代码片段 **(Existing Code)**:

```python
# In optiland/optic.py
# (假设 RealRays 已被导入)

class Optic:
    #... 其他方法，如 __init__, add_surface 等...

    def trace(self, Hx, Hy, wavelength, num_rays, distribution="hexapolar") -> "RealRays":
        """
        Traces a bundle of rays through the optical system.
        """
        #... 此处为光线生成逻辑...
        # rays = self._generate_rays(...)

        # 这是将被修改的核心追迹循环
        for i in range(1, self.surface_group.num_surfaces):
            surface = self.surface_group.surfaces[i]
            rays = surface.trace(rays)

            # 两个表面间的标准直线传播
            if i < self.surface_group.num_surfaces - 1:
                next_surface = self.surface_group.surfaces[i+1]
                thickness = next_surface.thickness
                rays.propagate(thickness, material=surface.material_post)

        return rays
```

- 交互与依赖:
    - Optic.trace 方法是整个追迹过程的“指挥官”。它从 self.surface_group 中获取表面对象

，并依次调用它们的 trace 方法。
    - 标准传播模型非常简单：光线与 surface[i] 相互作用后，在一个均匀介质 (surface.material_post) 中沿直线传播 (rays.propagate) 一段距离，该距离由下一个表面 surface[i+1] 的 thickness 属性定义。
    - 新的 GRIN 逻辑必须在此循环中被触发，临时接管光线传播的控制权，并在完成 GRIN 区域的追迹后，将光线状态无缝交还给主循环。

模块/文件 B (理解/调用): optiland/surfaces/standard_surface.py

- 与任务的关联: 新的 GradientBoundarySurface 将继承自此文件中定义的 Surface 类 [2]。理解基类的 _trace_real 和 _interact 方法至关重要，因为 GRIN 入口表面在将控制权交给数值积分器之前，仍需执行一次标准的折射计算。这确保了光线以正确的初始角度进入 GRIN 介质。
- 相关代码片段 **(Existing Code)**:

```python
# In optiland/surfaces/standard_surface.py
# (假设 RealRays, BaseGeometry, BaseMaterial 等已被导入)

class Surface:
    #... __init__ 等方法...

    def _trace_real(self, rays: "RealRays") -> "RealRays":
        """
        Traces real rays through the surface.
        """
        #...
        self.geometry.localize(rays)
        t = self.geometry.distance(rays)
        rays.propagate(t, self.material_pre)
        rays.opd = rays.opd + be.abs(t * self.material_pre.n(rays.w))

        # 关键：调用 _interact 来处理物理现象（折射/反射）
        rays = self._interact(rays)

        self.geometry.globalize(rays)
        #...
        return rays

    def _interact(self, rays: "RealRays") -> "RealRays":
        """
        Interacts the rays with the surface by either reflecting or refracting.
```

```
    """
    nx, ny, nz = self.geometry.surface_normal(rays)

    if self.is_reflective:
        rays.reflect(nx, ny, nz)
    else:
        # 此处应用斯涅尔定律
        n1 = self.material_pre.n(rays.w)
        n2 = self.material_post.n(rays.w) # 对于 GRIN 入口，这里将调用 GradientMaterial.n()
        rays.refract(nx, ny, nz, n1, n2)

    #... (处理涂层和散射的逻辑)
    return rays
```

- 交互与依赖:
  - _trace_real 方法严格遵循一个操作序列：1) 计算交点，2) 传播至交点，3) 更新光程差 (OPD)，4) 执行物理交互（折射），5) 坐标系反变换。
  - _interact 方法是应用斯涅尔定律的地方。根据设计文档（第3节第3项）的要求，对于 GradientBoundarySurface，n1 是前一个介质的折射率，而 n2 是 GradientMaterial 在交点处的折射率。

模块/文件 C (理解/调用): optiland/materials/base.py

- 与任务的关联: 新的 GradientMaterial 必须继承自 BaseMaterial。尽管无法直接获取 BaseMaterial 的源码 [3]，但可以通过分析其子类（如 AbbeMaterial）来推断其接口契约 [3]。现有系统（特别是 Surface._interact）期望所有材料对象都提供一个仅依赖于波长的 n(wavelength) 方法。然而，GradientMaterial 的折射率是空间坐标的函数。这种接口不匹配必须得到解决，以确保新模块能被现有代码无缝调用。解决方案是让 GradientMaterial 实现一个 n(wavelength) 方法，该方法返回与位置无关的基准折射率 n_0。这使得在 GRIN 边界上的初始折射计算可以正确进行，同时保留了内部更复杂的空间依赖模型。
- 相关代码片段 **(Inferred Interface)**:
  Python
  ```
  # Inferred from optiland/materials/base.py and its subclasses
  from abc import ABC, abstractmethod

  class BaseMaterial(ABC):

      @abstractmethod
      def n(self, wavelength: float) -> float:
  ```

```
        """
        Returns the refractive index of the material.
        For standard materials, this is only a function of wavelength.
        """
        pass

    #... 其他方法，如 to_dict, from_dict...
```

- 交互与依赖:
  - Surface._interact 方法会调用 self.material_post.n(rays.w) 来获取折射后的介质折射率。如果 GradientMaterial 不提供此方法，程序将在运行时崩溃。
  - 因此，必须为 GradientMaterial 实现一个符合此接口的 n 方法，即使它忽略 wavelength 参数并简单地返回其基准折射率 n_0。

模块/文件 D (理解/调用): optiland/rays/real_rays.py

- 与任务的关联: RealRays 类是光线状态的权威表示 [4]。它封装了光线的位置 (x, y, z)、方向余弦 (L, M, N) 以及光程差 (OPD) 等所有关键属性。新的 propagate_through_gradient 函数将接收一个 RealRays 对象作为输入，并根据光线微分方程，通过 RK4 数值积分方法迭代更新这些状态变量。
- 相关代码片段 **(Existing Code)**:

```python
# In optiland/rays/real_rays.py

class RealRays(BaseRays):
    def __init__(self, x, y, z, L, M, N, intensity, wavelength):
        self.x = be.as_array_1d(x)  # 位置矢量分量
        self.y = be.as_array_1d(y)
        self.z = be.as_array_1d(z)
        self.L = be.as_array_1d(L)  # 方向余弦矢量分量
        self.M = be.as_array_1d(M)
        self.N = be.as_array_1d(N)
        self.opd = be.zeros_like(self.x) # 光程差
        #...

    @property
    def position(self) -> "be.ndarray":
        """Returns the position vectors of the rays."""
        return be.stack([self.x, self.y, self.z], axis=-1)

    @property
```

```
    def direction(self) -> "be.ndarray":
        """Returns the direction vectors of the rays."""
        return be.stack([self.L, self.M, self.N], axis=-1)

    #... 其他方法，如 propagate, refract, reflect...
```

- 交互与依赖:
  - RealRays 对象是整个追迹过程中的核心数据载体。
  - 设计文档中 propagate_through_gradient 函数的实现依赖于对 ray_in.position 和 ray_in.direction 的访问，RealRays 类通过属性提供了这些便捷的接口。
  - 同样，该函数需要累积光程差，并更新 ray.opd 属性，RealRays 类已具备此属性。

4. 实现建议 (Implementation Guidance)

以下是为 AI 程序员提供的高层次、分步骤的实现指南，旨在清晰、无歧义地指导编码工作。

1. 第一步：创建新模块文件
   - 在指定的目录路径下创建三个新文件：
     - optiland/surfaces/gradient_surface.py
     - optiland/materials/gradient_material.py
     - optiland/interactions/gradient_propagation.py
   - 将【新功能/修订设计文档】中为这三个模块提供的最终代码定义完整地粘贴到对应的新文件中。
   - 关键补充：为了解决第 3.3 节中分析的接口兼容性问题，请在 optiland/materials/gradient_material.py 的 GradientMaterial 类中添加以下方法：
     Python
     ```
     # 将此方法添加到 GradientMaterial 类中
     def n(self, wavelength: float) -> float:
         """
         为与 BaseMaterial 接口兼容，返回基准折射率 n0。
         在此简化模型中，波长参数被忽略。
         """
         return self.n0
     ```

2. 第二步：修改核心追迹逻辑 **(optic.py)**
   - 这是任务中最复杂的部分。用以下经过充分注释的 while 循环逻辑完全替换 Optic.trace 方法中现有的 for 循环。使用 while 循环是必要的，因为它允许我们手动控制循环计数器 i，从而在处理完一个 GRIN 区域后能够“跳过”该区域内的所有表面。
   - 首先，在 optiland/optic.py 文件顶部添加必要的导入语句：
     Python

```
from optiland.surfaces.gradient_surface import GradientBoundarySurface
from optiland.materials.gradient_material import GradientMaterial
from optiland.interactions.gradient_propagation import propagate_through_gradient
```

- 然后，在 Optic.trace 方法内部，用以下代码块替换原有的 for 循环：

```
Python
#... 在 trace 方法内部，光线生成之后...
i = 1
while i < self.surface_group.num_surfaces:
    surface = self.surface_group.surfaces[i]

    # 检查当前表面是否为 GRIN 介质的入口标记
    if isinstance(surface, GradientBoundarySurface) and not surface.is_reflective:

        # 1. 验证其后的材料必须是 GradientMaterial
        grin_material = surface.material_post
        if not isinstance(grin_material, GradientMaterial):
            raise TypeError("GradientBoundarySurface 必须后跟一个 GradientMaterial。")

        # 2. 寻找配对的出口表面（也必须是 GradientBoundarySurface）
        #    这实现了设计文档中推荐的“成对标记”方案
        exit_surface_index = -1
        for j in range(i + 1, self.surface_group.num_surfaces):
            if isinstance(self.surface_group.surfaces[j], GradientBoundarySurface):
                exit_surface_index = j
                break

        if exit_surface_index == -1:
            raise ValueError("GRIN 区域已开始，但未找到匹配的 GradientBoundarySurface 作为出口。")

        exit_surface = self.surface_group.surfaces[exit_surface_index]

        # 3. 在入口表面执行一次标准追迹。这会处理从外部介质
        #    到 GRIN 介质基准折射率的初始折射。
        rays = surface.trace(rays)

        # 4. 将控制权交给专门的 GRIN 传播函数。
        #    该函数将通过数值积分计算光线轨迹，直到与出口表面相交。
        rays = propagate_through_gradient(
            ray_in=rays,
            grin_material=grin_material,
            exit_surface=exit_surface.geometry, # 传递几何对象用于相交检测
```

```
                step_size=0.1, # 采用设计文档中的默认步长
                max_steps=10000
            )

            # 5. 此时光线已位于出口表面上。我们必须执行离开 GRIN 介质的
            #    最后一次折射。这通过直接调用出口表面的 _interact 方法完成。
            #    这一步至关重要，确保了光线正确地进入下一个均匀介质。
            rays = exit_surface._interact(rays)

            # 6. 关键步骤：将循环计数器快进到出口表面的索引。
            #    下一次循环将从 GRIN 区域之后的第一个表面开始。
            i = exit_surface_index

        else:
            # 对于所有非 GRIN 的标准表面，执行常规追迹
            rays = surface.trace(rays)

        # 两个表面间的标准直线传播（如果未到达系统末尾）
        # 如果下一个表面是 GRIN 区域的开始，则跳过此步骤，
        # 因为传播已由上面的 GRIN 逻辑处理。
        if i < self.surface_group.num_surfaces - 1:
            next_surface = self.surface_group.surfaces[i+1]
            # 检查下一个表面是否为 GRIN 入口，如果是，则不进行标准传播
            if not isinstance(next_surface, GradientBoundarySurface):
                thickness = next_surface.thickness
                rays.propagate(thickness, material=surface.material_post)

        i += 1

    return rays
```

5. 测试与集成上下文 (Testing & Integration Context)

为确保新功能的代码质量、一致性和可验证性，请遵循以下上下文信息。

- 测试模式:
  - 项目使用 pytest 作为测试框架。
  - 测试文件与源码文件平行存放于 tests/ 目录下，并遵循 test_*.py 的命名约定。例如，针对 optic.py 中新逻辑的测试应添加到 tests/optics/test_optic.py 中，或创建一个新的

tests/optics/test_grin_integration.py 文件。

- 相关测试示例:
  - 为了帮助 AI 编写风格一致且有效的测试，以下提供一个端到端的集成测试用例。该测试构建一个具有解析解的简单抛物线 GRIN 透镜（Wood lens），并验证其是否能正确聚焦准直光线。这不仅测试了 optic.py 中的集成逻辑，也同时验证了 propagate_through_gradient 的正确性。

```python
# 在新文件 tests/optics/test_grin_integration.py 中
import pytest
import numpy as np
from optiland import Optic
from optiland.surfaces.gradient_surface import GradientBoundarySurface
from optiland.materials.gradient_material import GradientMaterial
from optiland.rays import Ray

def test_parabolic_grin_lens_focusing():
    """
    测试一个简单的抛物线 GRIN 透镜（Wood lens）是否能正确聚焦准直光线。
    对于折射率 n(r) = n0 - A * r^2 的透镜，其焦距为 f = 1 / (2 * n0 * sqrt(2A))。
    为简化，我们使用 n(r^2) = n0 + nr2 * r^2 的形式，其焦距 f 约为 1 / (2 * n0 * alpha)，
    其中 alpha = sqrt(-2 * nr2 / n0)。
    这里我们直接验证光线是否在预期点汇聚。
    """
    n0 = 1.5
    nr2 = -0.005
    thickness = 10.0

    # 对于近轴光线，预期焦点位置 z (从透镜后表面算起)
    # p = sqrt(-2 * nr2 / n0)
    # expected_focal_length = 1 / (n0 * p * np.sin(p * thickness))
    # 为简单起见，我们直接追迹并验证其在轴上汇聚

    # 构建 GRIN 透镜系统
    grin_lens = Optic()
    grin_lens.add_surface(
        surface_type='GradientBoundarySurface',
        thickness=thickness,
        material_post=GradientMaterial(n0=n0, nr2=nr2)
    )
    grin_lens.add_surface(
        surface_type='GradientBoundarySurface',
        thickness=100.0 # 足够长的追迹距离
    )
    grin_lens.add_field(y=0.0) # 轴上视场
    grin_lens.add_wavelength(value=0.55)

    # 追迹一条离轴的准直光线 (Hx=0, Hy=0, Px=0, Py=0.1)
```

```
    # 注意：Optic.trace 尚不支持直接传递 Px, Py，这里用一个示例光线代替
    # 实际测试中可能需要构建 Ray 对象或扩展 trace 方法
    # 为了提供一个可工作的示例，我们假设可以追迹单条光线
    # 这是一个概念性测试，AI 需要根据实际 API 进行调整

    # 追迹一组光线
    rays = grin_lens.trace(Hx=0, Hy=0, wavelength=0.55, num_rays=1,
distribution="uniform")

    # 为了验证聚焦，我们需要一个离轴光线
    # 此处仅为示例，实际测试可能需要更复杂的设置
    # 假设我们能追迹一条 y=0.1, L=0, M=0, N=1 的光线
    # 最终，我们期望光线穿过光轴，即 y 坐标接近 0
    # 此处断言光线追迹没有失败
    assert rays is not None
    # 更具体的断言需要一个已知结果的精确模型
    # 例如，对于一个半周期 GRIN 棒，平行光应在出射面中心聚焦
    # assert np.isclose(rays.y, 0.0, atol=1e-4)
```

- 用户**API**示例:
  - 新功能的集成不应改变用户与 Optic 类的核心交互方式。复杂性应被完全封装。以下代码片段展示了用户将如何定义一个 GRIN 系统，并强调 trace 方法的调用方式保持不变，从而确保了 API 的一致性和向后兼容性。

```
Python
from optiland import Optic
from optiland.surfaces.gradient_surface import GradientBoundarySurface
from optiland.materials.gradient_material import GradientMaterial

# 用户如何定义一个包含 GRIN 透镜的系统：
my_grin_system = Optic()

# 第一个表面（空气）
my_grin_system.add_surface(thickness=10.0)

# GRIN 透镜入口
my_grin_system.add_surface(
    surface_type='GradientBoundarySurface',
    radius_of_curvature=100.0,
    thickness=5.0, # GRIN 介质的厚度
    material_post=GradientMaterial(n0=1.6, nr2=-0.01)
)

# GRIN 透镜出口
my_grin_system.add_surface(
    surface_type='GradientBoundarySurface',
    radius_of_curvature=-100.0,
```

```
    thickness=50.0 # 到下一个表面的距离
)

# 像面
my_grin_system.add_surface()

# 设置系统参数
my_grin_system.add_field(y=1.0)
my_grin_system.add_wavelength(value=0.55)

# 核心：trace 调用方式与标准系统完全相同，所有复杂性都被隐藏。
final_rays = my_grin_system.trace(Hx=0.0, Hy=0.5, wavelength=0.55, num_rays=128)

# 用户可以像往常一样分析 final_rays
```

引用的著作

1. 4. Getting Starting with the Codebase — Optiland 0.5.6 documentation, 访问时间为 十月 3, 2025, https://optiland.readthedocs.io/en/latest/developers_guide/getting_started.html
2. surfaces.standard_surface — Optiland 0.5.6 documentation, 访问时间为 十月 3, 2025, https://optiland.readthedocs.io/en/stable/_modules/surfaces/standard_surface.html
3. materials.abbe — Optiland 0.5.5 documentation, 访问时间为 十月 3, 2025, https://optiland.readthedocs.io/en/latest/_modules/materials/abbe.html
4. Source code for rays.real_rays - Optiland's documentation!, 访问时间为 十月 3, 2025, https://optiland.readthedocs.io/en/latest/_modules/rays/real_rays.html