基于线性规划的数据中心电力-算力协同调度优化

摘要

本文针对数据中心电力-算力协同调度优化问题构建了基于混合整数线性规划（MILP）的多层动态调度框架。模型结合了超伽马分布模拟的动态绿色电能（绿电）数据、滚动优化模型和多优先级队列调度模型，系统性的优化了调度模型稳定性、经济性、环保性及鲁棒性的多目标平衡。

本文根据调度问题的应用场景复杂度与基本假设，递进的建立了三个模型。

部分 2.1的模型一为基础调度模型，根据计算任务处理的守时性和经济性构建了双目标 MILP，确保任务稳定按时完成同时最小化运营成本。复合目标函数的约束包括绿电供电量上限与电能即买即用、所有优先级计算任务全量完成及强制高优先级任务无延时处理。模型一通过附录 A.1和 A.2所示的 **Python** 的 `PuLP` 库的线性求解器精确求解目标函数的电能矩阵和任务分配矩阵，求得了 27875 元成本的最优解。

部分 3.1的模型二在模型一基础上考虑了调度环保性的绿电占比指标，通过动态平衡目标函数的成本函数、延时函数及绿电占比指标函数对应的权重系数得到了平衡经济性与环保性的 31830 元成本的全局最优解。模型二通过与模型一相似的附录 A.3代码求解，参数敏感性分析揭示了经济性与时效性权重接近时存在 “相变” 的临界现象，而经济性与环保性因绿电电价优势体现协同效应。二维参数空间扫描验证参数 $\alpha, \beta, \gamma \in (0.2, 0.9)^3$ 非极端参数区域时模型具有强鲁棒性。模型二通过时序优化算法实现绿电驱动的负载弹性迁移，在保障任务完成率的同时提升了资源利用率与环保性。

本文在以上建模过程后进一步通过部分 4的模型三，结合局部优化与全局优化增强模型在现实调度中的鲁棒性。信息不完全与绿电极端波动。部分 4.2的子模型一在模型二基础上考虑调度信息不完全，通过滚动优化和建立概率分布将全局调度分解为多时间窗口内的子问题并动态预测调度信息。然而受限于局部调度信息，子模型一在实时预测计算效率过低。相对的，部分 4.3的子模型二假设数据中心可获取全局调度信息而关注绿电电价与供电量的极端波动，采用 “鲁棒-队列协同优化框架”。其中分布鲁邦机会约束基于 KL 散度和概率分布不确定性集合，将绿电供电量的随机约束转化为最坏情况机会约束，通过拉格朗日对偶变换与二分搜索确定阈值 $\sum n_j(t) \cdot P_j(t)$ 从而确保绿电使用量无溢出。队列调度模型考虑了多优先级计算任务的等待队列与容忍延时。先进先出（FIFO）模型对高优先级计算任务严格按时处理。加权轮询调度（WRR）模型及滑动时间窗口约束动态分配中、低优先级计算任务。队列调度模型通过队列长度反馈实时动态调度数据中心算力，实现负载弹性迁移与资源利用率峰值调控。

由此，本文通过三个核心模型的逐步建立、求解、结果分析和模型灵敏度分析，结合迭代寻优过程与可视化参数等手段，得到了稳健且易扩展的数据中心电力-算力协同调度模型。

关键词：混合整数线性规划（MILP），超伽马分布，滚动优化控制，多优先级队列调度

目录

1 简介

数据中心是存储、传输和处理数据的最关键的设施之一，是云计算，大数据和人工智能等技术实现可能性的根基所在。进入 21 世纪，由于廉价服务器数量的增加，数据中心的电力消耗在 2000 年至 2005 年间翻了一番 [3]。到 2016 年，全球数据中心的电力消耗达到416 TW h，占全球电力消耗的约 3% [8, 10]。预计到 2030 年，数据中心的电力需求将继续增长，可能达到752 TW h [13]。

近年来，以大语言模型（LLM）为代表的 AI 行业发展尤其迅速，因此对数据中心计算能力的需求日益水涨船高。这在一方面当然推动了经济与资本市场的发展，但同时也带来了不可忽视的环境问题，其中电力消耗与可能的碳排放最为显著 [2]。

电能，尤其是风能和太阳能等 “绿色” 的再生电能，因材料等原因产电不稳定或存电困难 [4]，需要**即发即用**。因此数据中心对电力网络的合理调度不仅关系到用电成本，还关系到碳排放及按时完成计算任务。

1.1 问题重述

数据中心电力-算力协同调度如如图 1所示。优化此问题的核心在于应对动态的算力要求和电力供应，可分为以下几个方面：

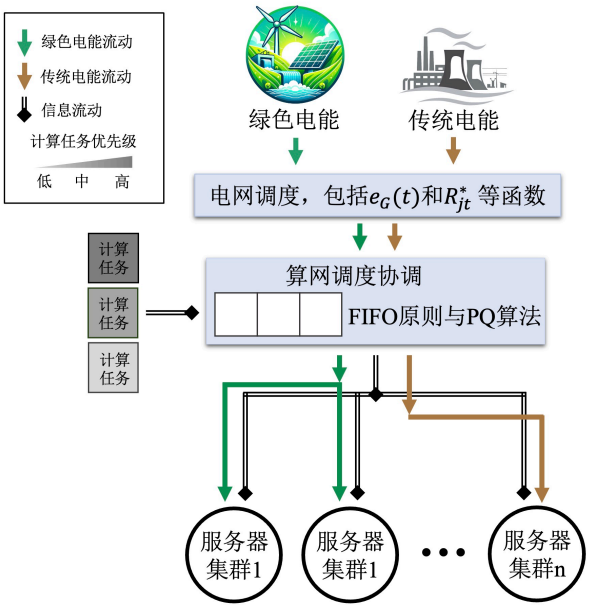

图 **1** 数据中心电网与算网协同调度示意图

1. **电力调度与算力分配**：任务的计算需求随时间变化，电力消耗与算力分配直接相关，需要通过调度避免算网中的空余算力浪费。
2. **任务优先级与调度**：不同任务具有不同的优先级，高优先级任务必须按时完成，低优先级任务可以适当延迟。调度策略需动态调整，以保证任务的按时完成。
3. **成本与减排优化**：根据实时电力供应合理调度恰好足够的电力，优先使用绿色电力，并只在其不足或无法完成任务时使用传统电力，从而最大化绿色电力的使用与降低成本。

4. **响应电力波动**：电力供应波动较大，需要设计高效的算法来实时调整计算资源，避免电力的过剩或短缺。

对于不同的优化问题，我们采用了 MILP、分布鲁棒优化、超伽马分布和队列调度等多个在实用性及准确度相互启发、递进的模型以平衡环保需求、计算负载和用电成本，为数据中心运行提供高效且环保的最优方案。图 2总结了我们建立模型的思路及模型间的关系：

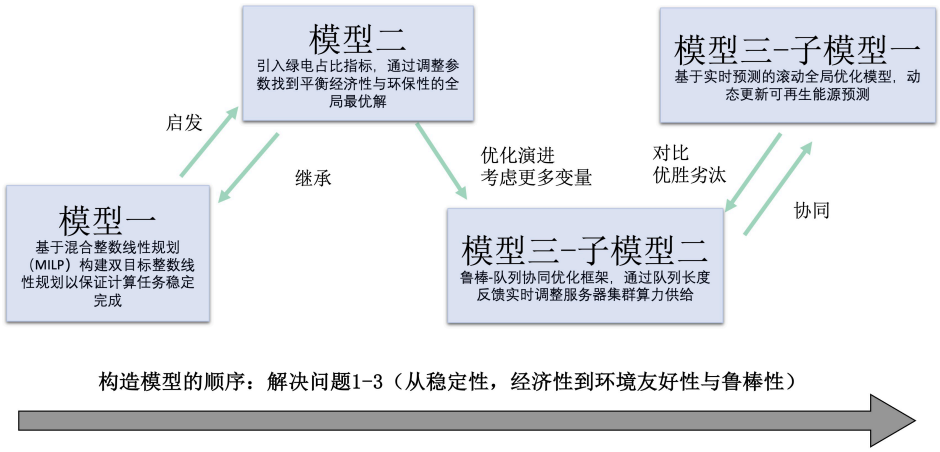

图 **2** 各模型的建立与关联流程图

1.2 文献综述

在经典的并行机调度（Parallel Machine Scheduling）问题中，任务分配与资源限制通常被视为一种**静态约束** [5]。然而，近些年来，研究者们逐渐意识到静态约束的局限性，并开始探索动态约束下的调度问题。文献 [20] 中探讨了在这个模型下如何利用一种延迟容忍的计算工作负载来优化数据中心的实时电力调度，他们通过分析短时可延迟工作负载、长期连续工作负载和长期可中断工作负载这三种工作负载，提出了一个基于时间转移的调度框架，并验证了其在多种场景下其对提高操作经济性和绿色能源利用效率的有效性。

然而，此类研究未充分考虑可再生能源本身的**强波动性**对调度方法**鲁棒性的影响**。文献 [9] 针对了风光互补供电的分布式数据中心场景，创新性地引入 KL 散度约束构建分布不确定性集合，结合分布鲁棒优化方法设计任务调度策略，证明其在电价波动与风光出力间歇场景下的经济性优势。值得注意的是，当任务调度需严格遵循时变可再生能源供给曲线时，问题复杂度显著提升。

文献 [15] 里首次从计算复杂性理论层面证明了此类受时变再生能源约束的并行机调度问题在属于非常强的 NP 困难问题，即便在单机场景下亦不存在多项式时间近似方案（PTAS），进一步了设计基于 KL 散度约束的分布鲁棒模型，通过双层分解策略将原问题退化为线性可解的调度子问题。这一理论标志着上述的传统精确算法的失效，促使学界转向启发式算法与分布鲁棒优化的融合

1.3 符号约定

我们根据后文公式中所需符号进行如下的符号约定。其中核心参数如表 1所示，核心函数如表 2所示。

表 **1** 核心参数的符号说明

符号	意义	单位
$T_I/\mathbf{T}$	计算任务 $T_I \in$ 任务集合 **T**，可分为绿电 T_G 或传统电能 T_T 供电	
$j/\mathbf{J}$	分为高、中、低三种的任务优先级类型索引 $j \in$ 类型集合 **J**	
$t/\mathbf{H}$	时刻 $t \in$24 h 时间集合 **H**，$\mathbf{H} = \{0, 1, \ldots, 23\}$	h
$w_i/\mathbf{W}$	第 i 个时间窗口 $w_i \in$ 时间窗口集合 **W**	h
D_I	处理每个任务 T_I 可容忍的延时	h
$c_i/\mathbf{c}$	第 i 个服务器集群 $c_i \in$ 服务器集群集合 **c**	
N_c	计算中心所有服务器集群总数量	个
ϵ	鲁棒性函数误差变量	
ξ	分布鲁棒约束的容忍阈值（允许的违约概率上限），$\xi = 0.05$	

表 **2** 核心函数的符号说明

符号	意义	单位
$N_j(t)$	收到的优先级 j 的计算任务数量关于 t 的函数	个
$n_j(t)$	可处理的优先级 j 的计算任务数量关于 t 的函数	个
$\epsilon_j(t)$	预测的优先级 j 的计算任务数量的扰动关于 t 的函数	
$\tau(T_I)$	计算任务处理时间关于 T_I 的函数	h
$p_{G/T}(t)$	绿电或传统电能的单位电价关于 t 的函数	元 $\cdot$ kW h^{-1}
$P_{I/j}(t)$	处理 j 优先级或某计算任务的单位能耗关于 t 的函数	kW h $\cdot$ 个$^{-1}$
$E_{G/T}(t)$	绿电或传统电能供电量关于 t 的函数	MW
$e_{G/T}(t)$	计算中心绿电或传统电能需求量关于 t 的函数	MW
$C/\tilde{C}$	**成本函数**，$\tilde{C}$ 为归一化后的成本函数	
$D/\tilde{D}$	**延时函数**，$\tilde{D}$ 为归一化后的延时函数	
$U(T_I)$	延时函数的权重关于 T_I 的函数	h^{-1}
$\delta x_j(T_I)$	延时函数的延时时长关于 T_I 的函数	h
$G/\tilde{G}$	**绿电占比指标函数**，$\tilde{G}$ 为归一化后的绿电占比指标函数	
R	**鲁棒性函数**	

表 **3** 数学推导的符号说明

符号	意义
$P(x)$	随机变量 x 发生的概率
$f(x)/F(x)$	连续随机变量 x 的**实际分布**的概率密度函数或概率分布函数
$g(x)/G(x)$	连续随机变量 x 的**参考分布**的概率密度函数或概率分布函数
$\mathbb{E}_f[x]$	某概率密度函数 f 关于随机变量 x 的期望
$g(x)$	连续随机变量 x 的**参考分布**的概率密度函数
$\mathbb{P}$	某概率密度函数的概率测度空间
$D_{KL}(f\|g)$	KL 散度（相对熵），衡量概率分布 f 与 g 的偏离程度
d	相对熵半径
U_r	不确定性集合
$\mathcal{L}(f, \alpha, \beta)$	关于函数 f 和参数 α 与 β 的拉格朗日函数
$h(x)$	示性函数。集合内有 x 时 $h(x) = 1$，无 x 时 $h(x) = 0$

1.4 基本假设

1. 通过超伽马分布生成的数据集是可信且可靠的。
   原因：数据驱动的调度方法依赖高保真数据集捕捉电力供需的随机性 [22]。超伽马分布能较稳定的描述电力供应波动和任务到达的非对称性 [1, 7]。

2. 模型中的 “绿电” 泛指所有新能源电力，等价于题目对新能源电力的任何描述（如 “New” 和 “Renewable”）。绿电没有价格和种类的区别。
   原因：绿电的环保性是核心差异，种类不影响调度逻辑。绿电定价可由政府补贴或市场机制统一，不同种类间价格差异可忽略。故统一绿电的定义和类型可简化供求关系 [17, 18] 和模型复杂度 [19]。

3. 服务器集群的功率为常数。
   原因：实际中服务器负载与功耗的线性关系已被广泛验证 [12, 16]，恒定功率的简化也便于计算总能耗与电力成本。

4. 每个服务器集群的算力状态依据于提供的电力，在高、中、低三个优先级的计算任务所需能耗对应的电力下都可以工作。
   原因：模型中服务器集群算力状态依赖供电状态体现电力对算力直接影响。离散的电力档位简化模型计算，实际中服务器可通过降频等方式适应不同供电量 [20, 21]。

5. 不同优先级的计算任务所需的处理时间相同。
   原因：实际中高优先级任务可能更紧急但未必更耗时，而任务优先级仅影响调度顺序。忽略计算任务复杂度差异可聚焦优先级调度，简化延迟分析 [14]。

6. 数据中心的任务调度可实时适应较大的电力波动。信息（如计算任务、电价等）传输没有延迟。
   原因：实时性是动态调度的核心需求，延迟会导致算力任务调度失效，可能导致电力短缺或过剩。故相关研究常通过假设或机会约束处理实时波动 [20]。现实中存在支持瞬时响应的数据同步技术（如智能电网通信协议）。

7. 计算任务收到的每个任务在24 h 内必须完成。调度系统需要在这个时间窗口内处理所有任务，并确保任务按其优先级完成。
   原因：符合 “关键任务不受影响” 要求，避免任务积压 [16]。24 h 窗口为成本优化提供利用低价时段等方法的灵活性，同时促成核心约束 “确保更高优先级任务按时或优先完成”。

2 模型一：守时性-经济性线性规划模型

2.1 模型建立

我们的协同算力与电力调度的优化模型设定了公式 1所示的**目标函数**。我们旨在通过最小化成本函数、绿电占比函数、稳定性函数和鲁棒性函数的线性组合优化数据中心运营的经济性、环保性、稳定性和鲁棒性。不过在对问题一的模型建立中我们只考虑成本函数和延时函数的影响，α 和 β 分别为归一化后的成本函数和延时函数的参数。

$$\min\left(\alpha\tilde{C}+\beta\tilde{D}\right) \tag{1}$$

我们为最小化数据中心整体运营成本以满足优化模型的经济性设定了公式 2所示的**成本函数**。有算力需求时购买相应电力以及没有算力需求时关闭服务器集群减少电力消耗可最小化运营成本。

$$C = \sum_{t\in\mathbf{H}} \sum_{j\in\mathbf{J}} \left(p_G(t)\cdot n_j(t)\cdot P_j(t) + p_T(t)\cdot n_j(t)\cdot P_j(t)\right) \tag{2}$$

我们设定了使用绿电或低估 t 时实际所需总功率导致完成计算任务延时的**延时函数**，如公式 3所示。其中 $\delta x_j(T_I) = t_{\text{完成}} - t_{\text{截止}} \geq 0$。模型通过减少完成计算任务延时增加数据中心可用电力和算力服务的稳定性和连续性。

$$D = \sum_{T_I\in\mathbf{T}} U(T_I)\cdot \delta x_j(T_I) \tag{3}$$

我们的限制包括公式 4所示的 “完成计算任务耗电即服务器集群全部耗电”，公式 5所示的 “完成所有优先级全部计算任务”，公式 6所示的 “绿电使用不超过上限”，公式 7所示的 “电力必须即买即用”，和公式 8所示的 “完成高优先级计算任务守时”:

$$\sum_{j\in\mathbf{J}} n_j(t)\cdot P_j(t) = e_G(t) + e_T(t),\ \forall t\in\mathbf{T} \tag{4}$$

$$\sum_{t\in\mathbf{H}} N_j(t) = \sum_{t\in\mathbf{H}} n_j(t),\ \forall j\in\mathbf{J} \tag{5}$$

$$e_G(t) \leq E_G(t),\ \forall t\in\mathbf{H} \tag{6}$$

$$e_G(t), e_T(t) \geq 0,\ \forall t\in\mathbf{H} \tag{7}$$

$$\delta x_j(T_I) = 0,\ \forall T_I \text{ where } U(T_I) = 3 \tag{8}$$

2.2 模型求解

我们采用线性规划方法对数据中心电力-算力调度优化问题求解并使用了 **Python** 的 `PuLP` 库作为工具，程序代码详见附录 A.2。`PuLP` 库能够有效处理具有多个决策变量和约束条件的大规模线性规划问题并具有良好的数值稳定性。算法的实现采用附录 A.1的框架。具体而言，我们考虑了每个时间窗口的 $n_j(t)$，然后将约束条件通过数学表达式转化为公式 9所示的线性约束:

$$\min\left(\alpha\tilde{C} + \beta\tilde{D}\right)$$

$$\text{s.t.}\begin{cases} n_j(t),\ \forall j\in\mathbf{J}, t\in\mathbf{H} \\ \sum_{j\in\mathbf{J}} n_j(t)\cdot P_j = e_G(t) + e_T(t),\ \forall t\in\mathbf{H} \\ \sum_{t\in\mathbf{H}} N_j(t) = \sum_{t\in\mathbf{H}} n_j(t),\ \forall j\in\mathbf{J} \\ e_G(t) \leq E_G(t),\ \forall t\in\mathbf{H} \\ e_G(t), e_T(t) \geq 0,\ \forall t\in\mathbf{H} \\ \delta x_j(T_I) = 0,\ \forall T_I \text{ where } U(T_I) = 3 \end{cases} \tag{9}$$

求解此线性规划问题会得到如下的 “能量矩阵”、表示每小时两种类型电价的 “电价矩阵” 及 “任务分配矩阵”:

$$\boldsymbol{e}(t)=\begin{bmatrix} e_G(0) & e_T(0) \\ e_G(1) & e_T(1) \\ \vdots & \vdots \\ e_G(23) & e_T(23) \end{bmatrix} \boldsymbol{p}(t)=\begin{bmatrix} p_G(0) & p_T(0) \\ p_G(1) & p_T(1) \\ \vdots & \vdots \\ p_G(23) & p_T(23) \end{bmatrix}^{\top} \boldsymbol{n}_j(t)=\begin{bmatrix} n_{\text{高}}(0) & n_{\text{中}}(0) & n_{\text{低}}(0) \\ n_{\text{高}}(1) & n_{\text{中}}(1) & n_{\text{低}}(1) \\ \vdots & \vdots & \vdots \\ n_{\text{高}}(23) & n_{\text{中}}(23) & n_{\text{低}}(23) \end{bmatrix}$$

进而可计算花销矩阵 $\mathbf{C}=\mathbf{e}\cdot\mathbf{p}$ 并显式地得到公式 10所示的第 t 小时花销和公式 11所示的总花销:

$$C(t)=e_G(t)\cdot p_G(t)+e_T(t)\cdot p_T(t),\ \forall t\in\{0,1,\ldots,23\} \tag{10}$$

$$C=\sum_{t=0}^{23}C(t)=\sum_{t=0}^{23}\left(e_G(t)\cdot p_G(t)+e_T(t)\cdot p_T(t)\right) \tag{11}$$

随后我们通过 PuLP 库的线性求解器得了公式 12所示的矩阵，并通过计算每小时能源使用量确定了能源使用方案的总耗能公式 $\sum_{j\in\mathbf{J}} n_j(t)\cdot P_j(t),\ \forall t\in\mathbf{T}$，进而计算出了如公式 13所示的最小化运营成本的总花销。

$$\boldsymbol{E}(t)=\begin{bmatrix} 0 & 35150 \\ 0 & 20600 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix} \boldsymbol{p}(t)=\begin{bmatrix} 0.6 & 0.5 \\ 0.6 & 0.5 \\ \vdots & \vdots \\ 0.6 & 0.6 \end{bmatrix}^{\top} \boldsymbol{n}_j(t)=\begin{bmatrix} 370 & 0 & 185 \\ 0 & 412 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \end{bmatrix} \tag{12}$$

$$C=\sum_{t=0}^{23}C(t)=\sum_{t=0}^{23}\left(e_G(t)\cdot p_G(t)+e_T(t)\cdot p_T(t)\right)=27875\ \text{元} \tag{13}$$

分析表明，该模型有效平衡了任务调度与能源利用，在保证没有延时的同时实现了成本最小化：在传统能源具有价格优势的半夜和绿色能源供应充足的时段，优先安排计算任务。这体现了守时性和经济性的双重考虑，满足了不同优先级任务的时序要求，保证了服务质量。

3 模型二：环保性线性规划模型

3.1 模型建立

我们通过改进模型一在协同算力与电力调度的基础上考虑了环保性，建立了如公式 14所示的**目标函数**。其中 α、β 和 γ 分别为归一化后函数的参数。限制条件在公式 9基础上新增了 $\alpha+\beta+\gamma=1$。

$$\min\left(\alpha\tilde{C}+\beta\tilde{D}+\gamma\tilde{G}\right) \tag{14}$$

其中**绿电占比指标函数** G 如公式 15所示，描述了绿色能源在总能源中占比。模型通过增大 G 使能源结构更环保。相对的，公式公式 16所示的绿电优先使用的约束 “要

求系统优先使用可用的绿电，不足时使用传统电力补充”，进而实现环保和成本优化的目的。

$$G = \frac{\sum_{t\in\mathbf{T}} P_G(t)}{\sum_{t\in\mathbf{T}}[P_G(t) + P_T(t)]} \tag{15}$$

$$e_G(t) = \min\left(E_G(t), \sum_{j\in\mathbf{J}} n_j(t)\cdot P_j(t)\right), \forall t \in H \tag{16}$$

3.2 模型求解

我们采用与部分 2.2一致的线性规划算法 A.1，通过 `Pulp` 库的线性求解器在确保一定守时性的同时成本最小化和绿色能源使用率最大化，得到了图 3所示的最优资源分配：100% 完成所有计算任务下优化总成本优至 31830 元、绿色电力占比至 70.3%。

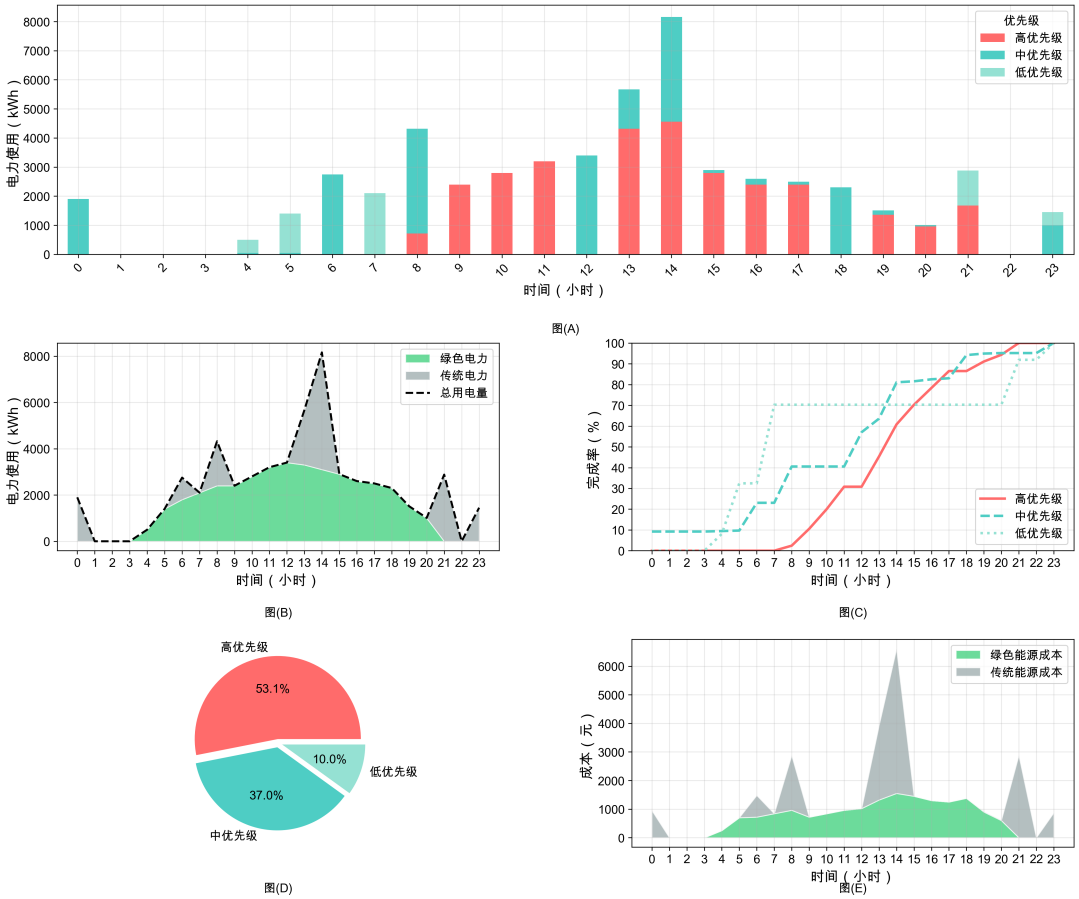

图 3 数据中心24 h 内计算任务调度与电力分析图。图（A）为计算任务电力分配图。图（B）为电力来源图。图（C）为累计计算任务完成率图。图（D）为计算任务电力按优先级分配图。图（E）为小时电力成本图

因此我们认为该模型有效平衡了任务调度、成本控制和绿色能源利用。如图（A）所示在保证没有计算任务延时下，在绿电供应充足的时段优先安排任务。这体现了经济性和环保性的双重考虑。

3.3 参数灵敏度分析

我们在调参过程中通过改变模型参数 α、β 和 γ 发现了它们的合作与博弈关系。三个参数代表的能耗成本，计算任务延时和绿电使用率的关系如图 4所示。由于 α 和 β 分别代表的 “经济性” 和 “守时性” 存在根本矛盾，可见图（A）和图（B）存在明显的 “分界线”。

若完全考虑成本控制，则调度程序将不会安排任何任务；若只考虑守时性，则调度程序将尽可能早的完成所有任务。α 和 β 大小相近时，经济性与守时性考量将达到类似 “受限平衡”（Limiting Equilibrium）的状态。当某参数略大于另一参数时，成本与守时的平衡被打破，出现类似 “相变” 的效果。当 β 较大时结果趋向守时，进入较高成本，较低延误 “相”；反之，当 α 较大时结果趋向成本，进入较低成本，较高延误 “相”。

同时图（A）和图（B）左上角的相似结构中存在的 “成本越高，绿电使用量越低” 关系也可看出 α 和 β 存在一种 “合作” 关系。这是由于绿电比传统电力存在大范围价格优势，保证绿电使用率一定程度上保证了调度经济性。

模型对三个参数的灵敏度由图 4所示参数空间的颜色梯度体现。除去 “任意一参数接近 1、其余参数接近 0” 的特殊极端情况，模型在空间中具有较高稳定性，能够在大多数参数下得到稳定的优化结果。

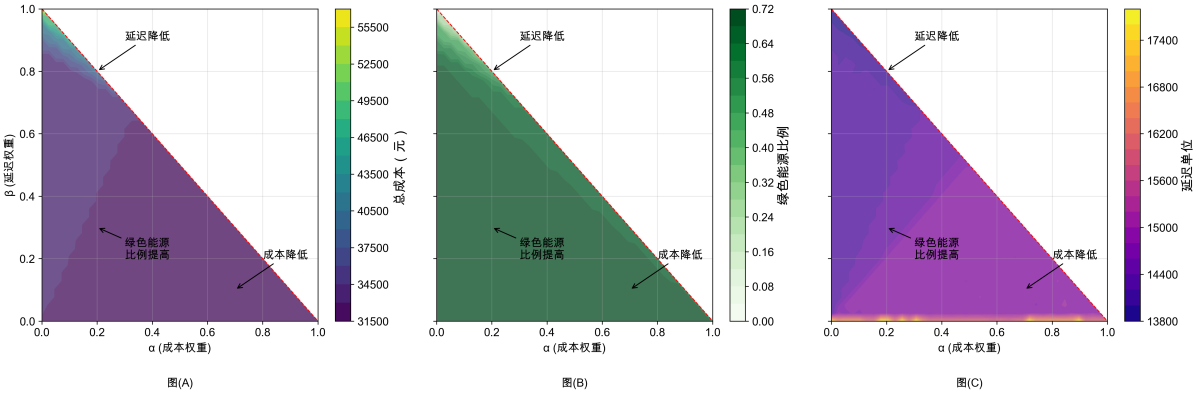

图 **4** 模型二参数敏感性分析图。图（A）为总成本敏感性分析图。图（B）为绿色能源比例敏感性分析图。图（C）为计算任务延时敏感性分析图（纵轴量级由于对延时判断并非以时间段为标准而严重过大，并非模型守时性表现糟糕。）

4 模型三

4.1 额外基本假设

1. 数据中心由不同的服务器集群组成，服务器集群有数量和算力的上限。数据中心的每个服务器集群的算力上限相同。
   原因：标准化算力上限便于量化任务分配，保证模型可扩展性。实际中数据中心常采用模块化设计，群组性能一致利于运维和调度。同构假设常见于分布式调度分析 [12, 14]。同构服务器集群简化资源分配模型，避免异构算力带来的非线性约束。

2. 一个服务器集群能够在给定时间（1 h）内处理所需功率不超过 $P_{\max}$（80 kW h）的计算任务，包括高、中、低三个优先级的任务。

原因：服务器集群存在算力上限符合现实情况。分优先级任务队列的模型可确保关键任务资源预留 [14, 21]，且可处理三种优先级计算任务的服务器集群满足适应动态负载的题目要求。

3. 文献中的计算任务电力供应的历史数据 [6, 19] 和超伽马分布生成的数据 [15] 可靠。调度模型可以根据这些数据较好的预测电力使用。
   原因：历史数据是时间序列预测如绿能发电周期、电价波动规律的基础 [21]，是动态调度的前提，模型需依赖数据拟合实现前瞻性规划。实际中数据中心的能源管理系统（EMS）已集成历史与预测数据接口。

4. 只要服务器集群开机，就至少会以最低功耗运行。
   原因:。在优化调度中计入最低功耗有助于更贴近实际情况，并可提高模型的适用性和现实意义。

5. 单个服务器集群能够运行多个计算任务，但功耗和不得超过服务器集群功耗上限 $P_{\max}$。
   原因：单个服务器集群能够运行多计算任务的调度模型能提高资源利用率，而功耗上限约束确保物理设备的安全性和电力供应的稳定性，符合实际服务器运行的硬件限制。这种限制在优化模型中可简化约束表达，便于计算。

6. 服务器集群开始运行某计算任务后必须不中断的运行完该任务才能切换其他任务。
   原因：计算任务的原子性符合数据中心调度的基本原则，有助于保证任务的完整性，避免因任务中断导致的数据丢失或延迟。同时，这一假设简化了调度模型，不需考虑任务的中途切换对时间和能耗的影响。

7. 同优先级的计算任务是完全相同的。
   原因：该假设减少调度算法复杂度，便于量化分析，且没有影响不同优先级计算任务的区分。

8. 计算任务间切换没有延时。
   原因：该假设可以显著减少模型复杂度同时满足大部分实际需求，尤其在计算任务量大或切换延时较小的情况下（如现代数据中心的快速切换技术）。

9. 发电量、电价、计算任务类型及计算任务数量等信息不再是静态的，而是随时间动态变化的，且在给定时间段结束前无法提前获取。
   原因：动态的信息更符合现实数据中心的情况，且符合 “…电价随着供需波动而变化” 的题目要求。模型依靠预测及队列处理的方法进行实时优化。

10. 电价实时波动为小时级变化，且电价不会被数据处理中心的需求波动影响。
    原因：数据处理中心能耗在电网中的占比较小。

4.2 子模型一：基于实时预测的滚动全局优化模型

本算法采用 “预测-优化-反馈” 的动态调度框架。通过实时更新可再生能源供应预测与任务需求预测，结合线性规划优化模型，实现电力与算力的协同调度。其核心流程分为以下三个阶段:

1. 预测模型：实时更新绿电供电量与收到的计算任务处理量的概率分布模型，为优化模型提供参数。

2. 优化模型：基于更新后的预测数据，生成当前窗口内的最优资源分配方案。

3. 反馈机制：将实际调度结果反馈至预测模块，修正后续预测误差。

为了更好的找到线性规划问题的最优解，我们引入了一系列误差变量 ϵ、计算效率 η 及新目标函数—**鲁棒性函数** R。我们对鲁棒性分析线性规划问题的数学表达如公式 18所示。

$$R = \alpha_1\epsilon_s + \alpha_2\epsilon_u + \alpha_3\epsilon_m + \alpha_4\epsilon_p - \alpha_5\eta + \alpha_6\epsilon_\Delta \tag{17}$$

$$\text{s.t.} \begin{cases} \epsilon_s = \sum \max(0, \Phi^d(t) - \Phi^a(t)) \\ \epsilon_u = \sum \max(0, \Phi^a(t) - \Phi^d(t)) \\ \epsilon_m = \sum \max(0, C(t) - D(t)) \\ \epsilon_p = \sum \max(0, C(t) - D(t)) \\ \eta = \dfrac{\sum \Phi_G(t)}{\sum \Phi_T(t)} \\ \epsilon_\Delta = \sum \left|\Delta\Phi^d(t) - \Delta\Phi^a(t)\right| \end{cases} \tag{18}$$

4.2.1 实时预测模型

我们受文献 [15] 启发通过贝叶斯方法每小时根据新的观测数据动态修正计算任务数量、绿电供电量及电能电价，并更新模型参数。

计算任务数量由滑动窗口加权平均与伽马分布预测，如公式 19所示。$\sum N_j(w_i)$ 与 $N_j(w_{k+1})$ 分别代表优先级 j 的计算任务数量的历史数据和当前时间窗口 w_{k+1} 的实时数据。预测的优先级 j 的计算任务数量的扰动项 $\epsilon_j(t) \sim \Gamma(10, 10)$。

$$N_j(w_{k+2}) = 0.9 \cdot \sum_i^k N_j(w_i) + 0.1 \cdot N_j(w_{k+1}) + \epsilon_j(t) \tag{19}$$

绿电供电量由伽马分布预测，如公式 20所示。$\mathbf{A}(t)$ 和 $\mathbf{B}(t)$ 均由历史数据均值和方差动态调整。

$$E_G(t) \sim \Gamma\left(\mathbf{A}(t), \mathbf{B}(t)\right) \tag{20}$$

绿电电价严格为正且右偏（如极端高价频发），故由对数正态分布预测以避免生成负电价，如公式 21所示。而传统电能电价短期波动近似关于 $\mu(t)$ 对称、突发性波动较少，故正态分布近似可有效简化模型计算，如公式 22所示。

$$p_G(t) \sim \ln\mathcal{N}(\mu(t), \sigma^2(t)),\ p_G(t) > 0 \tag{21}$$

$$p_T(t) \sim \mathcal{N}(\mu(t), \sigma^2(t)),\ \mu(t) \geq 3\sigma(t) \tag{22}$$

经检验，预测24 h 模拟数据的误差在 10% 以内。实现算法的 **Python** 代码详见附录 A.7。

4.2.2 滚动优化模型

我们采用了滚动优化的动态优化方法：在连续的时间窗口内反复进行决策调整。算法将时间窗口分为若干重叠或相邻的子窗口，每次仅针对当前窗口内的数据建模并求解最优方案，大致实现方式如附录 A.6的算法框架所示。窗口随时间推移向前 “滚动” 更新，利用局部最新信息重新优化后续决策。我们引入了鲁棒性惩罚函数、计算集群容量限制与电力需求限制的附加目标和约束以满足模型三在部分 23所示的额外基本假设。模型通过将时间窗口 w_k 的最优解作为当前窗口 w_{k+1} 的初始解的 “热启动策略” 降低迭代次数并加速结果收敛，并优化了计算效率。模型还通过允许中、低优先级计算任务在最大延迟约束 D_I 内弹性调度缓解了可能的瞬时电力供应压力，增强了鲁棒性。模型还通过全局优化根据实时供需变化调节目标函数的成本函数（经济性）与绿电占比指标函数（环保性），确保了不同场景下均能实现均衡优化。

$$\begin{cases} \sum\limits_{t\in\mathbf{H}} \max\left(0, \sum\limits_{j\in\mathbf{J}} n_j(t)\cdot P_j(t) - (E_G(t)+E_T(t))\right) \\ N_c\cdot\min\left(P_j(t)\right) \le e_G(t)+e_T(t) \le N_c\cdot\max\left(P_j(t)\right),\ \forall t\in\mathbf{H},\ \forall j\in\mathbf{J} \\ e_G(t)+e_T(t) \ge \sum\limits_{j\in\mathbf{J}} n_j(t)\cdot P_j(t),\ \forall t\in\mathbf{H} \end{cases} \tag{23}$$

4.2.3 模型结果分析及局限性

我们根据题目提供的数据，通过 **Python** 中的 `PuPL` 库的线性求解器动态求解该模型。我们得到了每小时的动态预测结果及优化方案。求得的绿电使用率始终超过 65%，总用电量约58 000 kW h，成本控制在 34502 元。模型的绿电使用率比非预测型模型降低 8%，成本上升 9%。由于部分 3.1的模型二可一次性获得24 h 内电能供电量，电力价格及计算任务数量的全部信息，我们认为本模型比非预测型模型模拟的绿电使用率下降低于 10% 展现了较好的稳定性和优化效果。

然而，模型的问题不可忽视。模型根据每小时最新数据修正后预测并优化牺牲了计算效率，导致其实用性低。模型根据题目24 h 数据滚动优化计算中心电力-算力协同调度优化问题时，**代码运行时长超过**2800 s。因此，我们不得不考虑算力中心可获得全局调度信息但如绿电电价等信息波动性强并建立新的模型。

4.3 子模型二：鲁棒-队列协同优化框架

4.3.1 分布鲁棒机会约束

由于数据中心的绿电供电量 $E_G(t)$ 具有强随机性，部分 4.2的子模型一的确定性约束 $\sum\limits_{j\in\mathbf{J}} n_j(t)\cdot P_j(t) \le E_G(t)+E_T(t)$ 在较大波动下频繁失效。例如绿电供电量低于预期时，高优先级计算任务可能因传统电能 $E_T(t)$ 无法及时弥补电力短缺而中断。因此，本模型通过要求绿电不足的概率不超过约定的容忍阈值 ξ 确保极少时间窗口的电力总需求不足支撑计算任务运行，引入公式 24所示的**分布鲁棒机会约束**。

$$P\left(\sum_{j\in\mathbf{J}} n_j(t)\cdot P_j(t) \ge E_G(t)+E_T(t)\right) \le \xi,\ \forall j\in\mathbf{J}, t\in\mathbf{H} \tag{24}$$

由于绿电供电量 $E_G(t)$ 的实际分布 f 未知，我们构建了公式 25所示的以参考分布 g 为中心、KL 散度（相对熵）半径为 d 的不确定性集合 U_r。其中 f, f^*, g 的自变量为 $E_G(t)+E_T(t)$。KL 散度 $D_{KL}(f\|g)=\mathbb{E}_f[\ln(f/g)]=\int f(i)\ln\frac{f(i)}{g(i)}\,\mathrm{d}x$，$d$ 为允许的最大偏离量。当 $d=0$ 时 f 与 g 严格一致。当 d 增大时 g 允许的偏离范围增大，模型鲁棒性增强，但保守程度也相应增强。公式 24所示的机会约束转化为最坏情况下 U_r 的概率上界 $\sup_{f\in U_r} P_f\left(\sum n_j(t)\cdot P_j(t)\geq E_G(t)+E_T(t)\right)\leq\xi$，要求电能供电量低于需求量的概率即使在最不利的真实分布 $f\in U_r$ 下仍不超过 ξ。

$$U_r=\left\{f\in\mathcal{P}\,\middle|\,\mathbb{E}_f\left[\ln\frac{f}{g}\right]\leq d\right\} \tag{25}$$

我们通过构造公式 26所示的拉格朗日函数 $\mathcal{L}$ 将鲁棒机会约束优化问题转化为对偶问题，并通过变分法求得公式 27所示的最坏实际分布的概率密度函数 f^* 及其对应的违约概率。$\mathcal{L}$ 包括违约概率期望项、KL 散度约束项及概率归一化约束项。f^* 是对 g 的指数修正，幅度由示性函数 h 和拉格朗日乘子 μ 和 λ 共同决定。x 代表 $\sum n_j(t)\cdot P_j(t)\geq E_G(t)+E_T(t)$ 的违约事件发生。$h(x)=1$ 时 f^* 相对于 g 被放大，反之被缩小，由此强化违约区域的概率密度并覆盖最不利情况。

$$\mathcal{L}(f,\mu,\lambda)=\mathbb{E}_f[h(x)]+\mu\left(d-\mathbb{E}_f\left[\ln\frac{f}{g}\right]\right)+\lambda\left(1-\mathbb{E}_f[1]\right) \tag{26}$$

$$f^*=g\cdot\exp\left(\frac{h(x)-\lambda}{\mu}-1\right) \tag{27}$$

公式 28结合了概率归一化约束（总概率积分为 1）和 KL 散度约束，进一步求解 μ 和 λ。其中 $(1+d)\mu+\lambda$ 为最坏实际分布的概率分布函数 F^*。$G(\sum n_j(t)\cdot P_j(t))=\int_0^{\sum n_j(t)\cdot P_j(t)} g(x)\,\mathrm{d}x$，为参考分布 g 的概率分布函数。由于 $\frac{\partial F^*}{\partial(\sum n_j(t)\cdot P_j(t))}\geq 0$，$F^*$ 随 $\sum n_j(t)\cdot P_j(t)$ 严格单调递增。这保证了存在唯一的阈值 $\sum n_j(t)\cdot P_j(t)$ 使 $F^*=\xi$。

$$\begin{cases} G\cdot e^{-\lambda/\mu}+(1-G)e^{(1-\lambda)/\mu}=1 \\ (1-G)e^{(1-\lambda)/\mu}=\lambda+\mu(1+d) \end{cases} \tag{28}$$

我们通过二分搜索法求得阈值，并将原约束转换为阈值对应的性线约束 $E_T(t)\geq\sum n_j(t)\cdot P_j(t)-E_G(t)$，要求传统电能需求量至少填补需求缺口从而间接限制绿电的最低需求量。将此约束带入模型约束后，我们得到了公式 29所示的线性规划问题:

$$\begin{aligned} &\min\sum_{t\in\mathbf{H}}\sum_{j\in\mathbf{J}}p_T(t)\cdot E_G(t) \\ &\text{s.t.}\begin{cases} \sum_{j\in\mathbf{J}} n_j(t)\cdot P_j=e_G(t)+e_T(t),\ \forall t\in\mathbf{T} \\ 0\leq e_G(t)\leq E_G(t),\ \forall t\in\mathbf{H} \\ E_T(t)\geq\sum_{j\in\mathbf{J}} n_j(t)\cdot P_j(t)-E_G(t),\ \forall t\in\mathbf{H} \\ \sum_{t\in\mathbf{H}} N_j(t)=\sum_{t\in\mathbf{H}} n_j(t),\ \forall j\in\mathbf{J} \\ \delta x_j(T_I)=0,\ \forall T_I \text{ where } U(T_I)=3 \end{cases} \end{aligned} \tag{29}$$

由此，我们通过鲁棒机会约束的阈值强制传统电能仅用于填补绿电供电量缺口进而**最大化了绿电使用率**。我们还通过 KL 散度定义的分布鲁棒方法考虑了绿电供电量波动的最坏情况，确保了**任何不确定性集合中的实际分布**的违约概率不超过容忍阈值。即使绿电供电量的实际分布与参考分布存在偏差，模型仍能可控的调度，避免了因绿电供电量波动导致的频繁电力短缺或过剩。

4.3.2 队列调度模型

队列调度模型通过队列积压的惩罚成本迫使模型对不同优先级计算任务调度，并按优先级独立更新队列。队列调度模型常采用 “先进先出”、“优先级队列调度” 和 “加权轮询调度” 三种优化方法 [11]，如图 5所示。我们在队列调度模型中主要考虑了加权轮询调度模型，并在建模过程中递进的考虑了三种方法。队列调度模型将24 h 分为绿电电价低谷期和高峰期，在绿电充足的低谷期分配更多计算任务来低成本清空队列。在绿电不足的高峰期，模型仅分配必须处理的高、中优先级任务，将低优先级任务延迟处理并依需求使用传统电力。具体而言，将计算任务优先级 j 与队列惩罚系数 u_j 关联量化了优先级对调度的影响。其中高优先级任务队列长度严格受限，可容忍延时 $D_I = 1\,\text{h}$ 且 $u_{\text{高}} = 0.1$，进而通过实时处理避免惩罚。中优先级任务允许短暂延迟，$D_I = 2\,\text{h}$ 且 $u_{\text{中}} = 0.05$，使其在绿电充足后被批量处理。低优先级任务允许最大程度的延迟，$D_I = 5\,\text{h}$ 且 $u_{\text{低}} = 0.01$，使其尽可能在队列短且电价低时被处理。

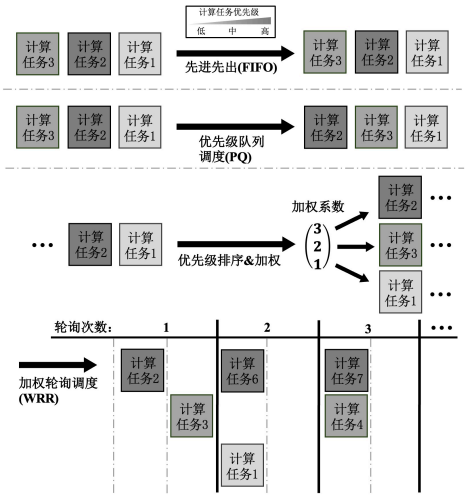

图 **5** 队列优化原则示意图。上、中、下图分别为先进先出（FIFO）、优先级队列调度（PQ）及加权轮询调度（WRR）模型示意图。其中收到及处理计算任务队列按从左往右顺序。

模型中 “队列长度约束” 如公式 30所示。每个优先级 j 计算任务在 t 时排队的队列不能积压负任务且长度不能超过 D_I 决定的最大值。因此，队列长度 $Q_j(t)$ 满足下限 $L_j^{\min}(t) = 0$ 与上限 $Q_j^{\max}(t)$。

$$L_j^{\min}(t) \le Q_j(t) \le Q_j^{\max}(t),\ \forall\, j \in \mathbf{J} \tag{30}$$

模型中优先级 j 计算任务 “等待的队列上限” 如公式 31所示，等于过去 i 个时间窗口内收到的此优先级计算任务的总数。$Q_j^{\max}(t)$ 随 t 滑动，确当前保计算任务在**可容忍的延时内被处理**。例如 w_1 时计算任务 T_I 等待的队列中仅允许保留近1 h 内的计算任务。

当 $t > \delta x_j(T_I)$ 时，$Q_j^{\max}(t) = \sum_{k=t-\delta x_j(T_I)}^{t} N_j(k)$；当 $t \le \delta x_j(T_I)$ 时，求和从非负索引的时间窗口 w_0 开始。

$$Q_j^{\max}(t) = \sum_{k=\max(1,t-\delta x_j(T_I))}^{t} N_j(k), \forall t \in \mathbf{H}, \forall j \in \mathbf{J} \tag{31}$$

$Q_j(t)$ 通过 $Q_j(t-1) + N_j(t) - n_j(t)$ 动态更新，由上一时间窗口队列长度 $Q_j(t-1)$ 加本时间窗口内收到的计算任务数量 $N_j(t)$ 减去可处理的任务数量 $n_j(t)$ 得到。考虑 w_i 前的所有时间窗口后可得公式 32所示的“累计队列长度”。

$$\sum_{t\in\mathbf{H}} Q_j(t) = \sum_{t\in\mathbf{H}} (Q_j(t-1) + N_j(t)) - n_j(t), \forall t \in \mathbf{H}, \forall j \in \mathbf{J} \tag{32}$$

先进先出（FIFO）模型中任务仅按抵达队列的时序被处理。由于不同优先级任务的 D_I 决定队列长度上限，模型在极端情况如 D_I 较短的高优先级任务累计时可导致队列长度溢出。因此，FIFO 模型通过定义滑动时间窗口的 $Q_j^{\max}(t)$ 仅保留最近 $\delta x_j(T_I)$ 内的任务队列。

优先级队列调度（PQ）模型中计算任务严格按其被划分的高、中、低三个优先级被依次处理，考虑到优先级越高 D_I 越短且队列惩罚系数 $u_j = 0.1$ 越高。因此，PQ 模型的成本函数将优先级映射为处理紧迫性并相应调度算力、电力资源，如图 6所示。

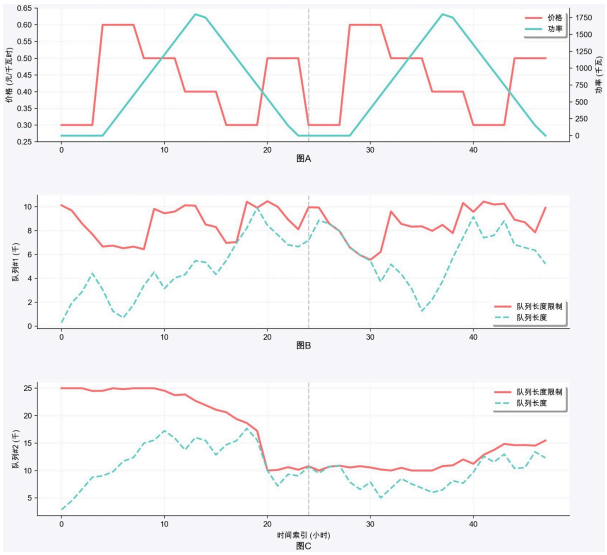

图 **6** 优先级队列调度（PQ）模型效率图。图（A）为实时电价与功率需求图。图（B）为高优先级任务队列状态图。图（C）为低优先级任务队列状态图。

加权轮询调度（WRR）模型对算力、电力的调度在 FIFO 和 PQ 模型的基础上还考虑了绿电供电量，将任务优先级对应到可容忍的延时与惩罚系数。绿电供应充足的用电量低谷期，如午间光伏供电超发时，WRR 模型通过公式 31动态计算 $Q_j^{\max}(t)$ 并根据 w_i 的低优先级任务累计量增加 $\sum n_j(t)$。因此，WRR 模型通过均衡三种优先级任务的处理权重利用过剩的低价绿电批量处理任务并快速清空队列。同时，与电价变化同步 $Q_j^{\max}(t)$ 确保了计算任务不超期和电力不浪费。绿电供应受限的高峰期 WRR 模型通过

限制 $n_j(t)$ 减少算力中心总处理量。惩罚系数 u_j 倾向优先处理 $Q_j^{\max}(t)$ 更小的高优先级任务而将低优先级任务延至电力充裕时段。因此，WRR 模型进一步避免了如处理过多任务导致电力不足等 FIFO 和 PQ 模型中已讨论过的隐患。

由此，我们得到子模型二“鲁棒-队列协同优化框架”三层结构的调度模型。外层考虑电网在 t 时的绿电和传统电能的电价和供电量，动态调整算力负载 $n_j(t)$。中层考虑任务优先级分配收到的任务量 $N_j(t)$ 以最小化高优先级任务的队列长度。内层通过 FIFO 模型和滑动窗口约束确保每个任务被按时处理且用电量无溢出。

4.3.3 模型有效性及灵敏度评估

“鲁棒-队列协同优化框架”的调度模型在24 h 内对电力的需求如图 7所示。其中绿电需求量在绝大多数时间都超过了传统电能需求量，而20 h 后由发电机制导致的高峰期限制短暂的低于传统电能需求。电力使用大多集中在 0 h − 10 h，总体峰值在 6 h − 7 h。其中可再生能源的峰值延后 1 h − 2 h 至 8 h − 10 h 左右达峰值。传统电能整体更平滑，没有明显的峰值。在10 h 后，数据中心电力负载呈整体下降趋势。

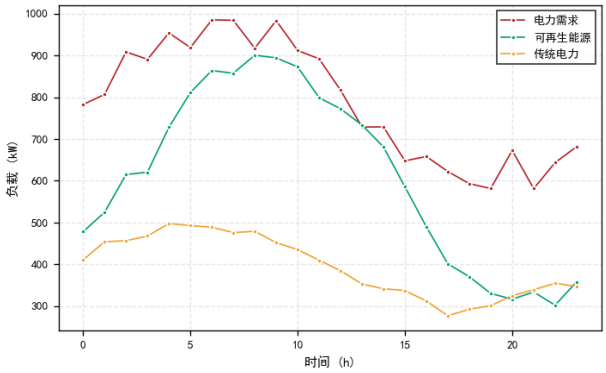

图 7 算力中心24 h 电力负载变化图

从图 7和 8可以看出，容忍的延时 D_I 变化对调度总成本影响有限。当 D_I 在 $0.1-0.3$ 内变化时总成本在 110000 元附近震荡且波动幅度较小，可见模型的成本控制具有较强鲁棒性。同时，违约概率在 $D_I = 0.1$ 时存在最小值 0.01 并随 D_I 增大略增大。但违约概率最大值仍低于 0.02，可见 D_I 对违约概率影响也有限，进一步说明了模型能适度容忍计算任务延时完成的同时灵活控制风险。

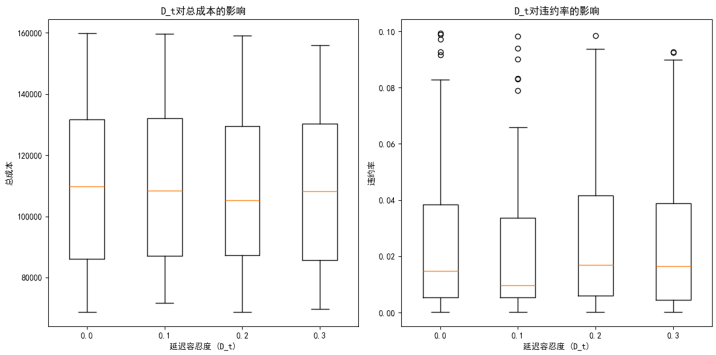

图 8 D_I 对总成本和违约概率的影响图。

子模型二在预测信息和滚动优化时具有显著效率优势，附录 A.8所示的代码运行时间绝大部分情况下小于600 s，在复杂场景测试下的稳定优化效果与部分 4.2的子模型一有显著区别。总成本低波动与违约概率最优阈值 $D_I = 0.1$ 共同体现了模型在无频繁优化参数时可稳定且高效的优化调度。然而，模型仍可面临如长期电力紧张时低优先级任务积压导致队列长度溢出等问题。因此，进一步优化调度可在子模型二“鲁棒-队列协同优化框架”的基础上根据更具体的应用场景与基本假设改进。

5 结论

我们递进的构建了兼顾经济性、环保性与鲁棒性的“鲁棒-队列协同优化框架”三层结构的数据中心电力-算力协同调度模型。部分 2.1的模型一实现了高优先级任务无延时且动态互补绿电与传统电能最小化运营成本的数据中心调度。部分 3.1的模型二进一步兼顾了增加绿电占比提升调度环保性。部分 4进一步考虑了现实中数据中心调度的不确定性。部分 4.2的子模型一考虑调度信息存在滞后性和较大波动的局部最优解，然而存在不可忽视的效率问题。因此，我们转向了基于分布鲁棒约束和队列调度策略的部分 4.3的子模型二，确保极端情况下计算任务按时完成且模型稳健。

子模型二为数据中心能源管理提供了实用参考。例如，电价波动较大时数据中心可通过实时优化预测其供需并在低价环保的电能充足时优先调度算力要求较大的高负载计算任务，从而最小化运营成本。我们在递进的模型建立和分析中所体现的弹性调度模式不仅契合智能电网的动态电价机制，还可助力绿色、减碳的环保性需求，为以“双碳”为目标的高耗能行业转型提供成熟的理论参考。

子模型二的算法效率可被进一步优化。例如，“鲁棒-队列协同优化框架”可结合深度学习更快速的优化现实中的大规模调度决策。同时，我们希望通过优化算法求解和调度信息预测精准度解决子模型一尚存的滚动全局优化问题。我们希望结合子模型一的实时局部预测和子模型二的高波动性全局优化实现兼顾两种情况的泛用性数据中心电力-算力协同调度模型。

参考文献

[1] E Suzuki. “Hyper gamma distribution and its fitting to rainfall data”. In: *Pap. Meteor. Geophys* 15 (1964), pp. 31–51.

[2] B. Davidson et al. “Large-scale electrical energy storage”. In: 127 (1980), pp. 345–385. DOI: 10.1049/IP-A-1:19800054.

[3] J. Koomey. “Worldwide electricity used in data centers”. In: *Environmental Research Letters* 3 (2008). DOI: 10.1088/1748-9326/3/3/034008.

[4] M. Whittingham. “Materials Challenges Facing Electrical Energy Storage”. In: *MRS Bulletin* 33 (2008), pp. 411–419. DOI: 10.1557/MRS2008.82.

[5] Tim Nieberg. “Scheduling Parallel Machine Scheduling”. In: *Colombia University, Spring* (2010). URL: https://www.or.uni-bonn.de/lectures/ss10/scheduling_data/sched10_4.pdf.

[6] Mario Mureddu et al. “Green Power Grids: How Energy from Renewable Sources Affects Networks and Markets”. In: *PLOS ONE* 10.9 (Sept. 2015), pp. 1–15. DOI: 10.1371/journal.pone.0135312.

[7] Mustafa bahşi and Süleyman Solak. “On the hyper-gamma function”. In: *Mathematical Sciences and Applications E-Notes* 5 (Apr. 2017), pp. 64–69. DOI: 10.36753/mathenot.421700.

[8] Bruce Bugbee et al. “Prediction and characterization of application power use in a high-performance computing environment”. In: *Statistical Analysis and Data Mining: The ASA Data Science Journal* 10.3 (2017), pp. 155–165.

[9] Yiwen Lu et al. “Energy-efficient task scheduling for data centers with unstable renewable energy: A robust optimization approach”. In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE. 2018, pp. 455–462.

[10] Indrajit Purkayastha. “Optimizing data center energy consumption”. In: 2018. URL: https://api.semanticscholar.org/CorpusID:227659409.

[11] 熊平. 中短期电价预测模型简介. Licensed under CC BY-NC-SA 4.0. XP’s Blog. May 2019. URL: https://blog.xpgreat.com/p/epf_models/.

[12] Leila Ismail and Huned Materwala. “Computing server power modeling in a data center: Survey, taxonomy, and performance evaluation”. In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–34.

[13] E. Masanet et al. “Recalibrating global data center energy-use estimates”. In: *Science* 367 (2020), pp. 984–986. DOI: 10.1126/science.aba3758.

[14] Reem Alshahrani and Hassan Peyravi. “Modeling and analysis of distributed schedulers in data center cluster networks”. In: *Cluster Computing* 24.4 (2021), pp. 3351–3366.

[15] Ayham Kassab et al. “Green power aware approaches for scheduling independent tasks on a multi-core machine”. In: *Sustainable Computing: Informatics and Systems* 31 (2021). ISSN: 2210-5379. DOI: 10.1016/j.suscom.2021.100590. URL: https://www.sciencedirect.com/science/article/pii/S2210537921000792.

[16] Bin Zou et al. “Optimal Power Scheduling of Data Centers with Deferrable Computation Requests”. In: *2021 IEEE 5th Conference on Energy Internet and Energy System Integration (EI2)*. IEEE. 2021, pp. 721–726.

[17] 曹雨洁 et al. “能源互联网背景下数据中心与电力系统协同优化 (二): 机遇与挑战”. In: 中国电机工程学报 42.10 (2021), pp. 3512–3526.

[18] 丁肇豪 et al. “能源互联网背景下数据中心与电力系统协同优化 (一): 数据中心能耗模型”. In: 中国电机工程学报 42.9 (2022), pp. 3161–3177.

[19] 陈敏 et al. “互联网数据中心负荷时空可转移特性建模与协同优化: 驱动力与研究架构”. In: 中国电机工程学报 42.19 (2022), pp. 6945–6957.

[20] Luyao Liu et al. “Optimal Energy Management of Data Center Micro-Grid Considering Computing Workloads Shift”. In: *IEEE Access* 12 (2024), pp. 102061–102075. DOI: 10.1109/ACCESS.2024.3432120.

[21] Yehan Wang et al. “Temporal Forecasting for IT Power Demand of Data Center”. In: *2024 IEEE/IAS Industrial and Commercial Power System Asia (I&CPS Asia)*. IEEE. 2024, pp. 754–759.

[22] 文淅宇 et al. “基于时空协同的多数据中心虚拟电厂低碳经济调度策略”. In: 电力系统自动化 48.18 (2024), pp. 56–65. ISSN: 1000-1026.

A 附录代码

A.1 模型一算法框架

部分 2.2的模型一的线性规划问题的算法框架伪码：

Algorithm 1: 数据中心电力-算力调度优化算法

Require: 任务集合 **J** 包含 N 个计算任务
时间窗口集合 **H** 包含 T 个调度时段
单位算力成本 c_j 任务 j 的算力成本
功率限制 $P_t^{\max}$ 时段 t 的最大允许功率
算力容量 $C^{\max}$ 服务器最大计算资源
任务需求 D_j 任务 j 总计算量
截止时间 t_j^{deadline} 任务 j 最晚完成时段

Ensure: 最优调度方案 $\{n_j(t)^*\}$.

初始化线性规划模型
创建问题实例：model $\leftarrow$ lpProblem("DCScheduling", Minimize);
定义变量：$n_j(t) \leftarrow$ lpVariable("$n_j(t)$", $0, C^{\max}$).
构建目标函数
最小化总成本：$\min \sum_{j\in\mathbf{J}} \sum_{t\in\mathbf{H}} c_j n_j(t)$.
添加约束条件
forall $j \in$ **J do**
  任务需求：$\sum_{t\le t_j^{\text{deadline}}} n_j(t) \ge D_j$. ▷ 确保任务完成
forall $t \in$ **H do**
  算力容量：$\sum_{j\in\mathbf{J}} n_j(t) \le C^{\max}$;
  功率限制：$\sum_{j\in\mathbf{J}} \epsilon_j n_j(t) \le P_t^{\max}$. ▷ 资源限制与电力约束
求解与验证
选择求解器：solver $\leftarrow$ COIN_CMD(threads $= 4$, timeLimit $= 3600$);
执行求解：status $\leftarrow$ model.solve(solver).
if *status* $\neq$ *Optimal* **then**
  抛出异常 ("无可行解")
返回结果
提取解：$\{n_j(t)^*\} \leftarrow$ getSolution(model);
return $\{n_j(t)^*\}$.

A.2 模型一代码

部分 2.2的模型一通过 PuLP 库求解线性规划问题的 **Python** 核心代码：

```python
import pulp


# Constants
HOURS = list(range(24))
URGENCY = {"HIGH": 3, "MEDIUM": 2, "LOW": 1}
```

```
6
7               # Task data structure with [count, power_per_task]
8               TASK_DATA = {
9                  "0-6": {"HIGH": [0, 80], "MEDIUM": [40, 50], "LOW": [60, 30]},
10                 "6-8": {"HIGH": [0, 80], "MEDIUM": [55, 50], "LOW": [70, 30]},
11                 "8-12": {"HIGH": [114, 80], "MEDIUM": [72, 50], "LOW": [0,
                       30]},
12                 "12-14": {"HIGH": [54, 80], "MEDIUM": [95, 50], "LOW": [0,
                       30]},
13                 "14-18": {"HIGH": [152, 80], "MEDIUM": [80, 50], "LOW": [0,
                       30]},
14                 "18-22": {"HIGH": [50, 80], "MEDIUM": [50, 50], "LOW": [40,
                       30]},
15                 "22-24": {"HIGH": [0, 80], "MEDIUM": [20, 50], "LOW": [15,
                       30]},
16              }
17
18              # Energy prices
19              GREEN_PRICES = [0.6, 0.6, 0.6, 0.6, 0.5, 0.5, 0.4, 0.4, 0.4,
                    0.3, 0.3, 0.3, 0.3, 0.4, 0.5, 0.5, 0.5, 0.5, 0.6, 0.6, 0.6,
                    0.6, 0.6, 0.6]
20              TRADITIONAL_PRICES = [0.5, 0.5, 0.5, 0.5, 0.6, 0.7, 0.8, 0.9,
                    1.0, 1.2, 1.3, 1.3, 1.2, 1.1, 1.0, 1.0, 1.1, 1.2, 1.3, 1.2,
                    1.1, 1.0, 0.8, 0.6]
21              GREEN_SUPPLY = [0, 0, 0, 0, 0.5, 1.4, 1.8, 2.1, 2.4, 2.4, 2.8,
                    3.2, 3.4, 3.3, 3.1, 2.9, 2.6, 2.5, 2.3, 1.5, 1.0, 0, 0, 0]
22
23              # Total tasks to be distributed across hours
24              n_h = sum([TASK_DATA[slot]["HIGH"][0] for slot in TASK_DATA])
25              n_m = sum([TASK_DATA[slot]["MEDIUM"][0] for slot in TASK_DATA])
26              n_l = sum([TASK_DATA[slot]["LOW"][0] for slot in TASK_DATA])
27
28              # Define LP problem
29              prob = pulp.LpProblem("Task Allocation Optimization",
                    pulp.LpMinimize)
30
31              # Decision variables: number of tasks to allocate to each hour
                    and priority
32              n_hi = pulp.LpVariable.dicts("HighTask", HOURS, lowBound=0,
```

```
            cat="Integer")
n_mi = pulp.LpVariable.dicts("MediumTask", HOURS, lowBound=0,
            cat="Integer")
n_li = pulp.LpVariable.dicts("LowTask", HOURS, lowBound=0,
            cat="Integer")

# Function to get the time slot based on the hour
def get_time_slot(hour):
    if hour < 6:
        return "0-6"
    if hour < 8:
        return "6-8"
    if hour < 12:
        return "8-12"
    if hour < 14:
        return "12-14"
    if hour < 18:
        return "14-18"
    if hour < 22:
        return "18-22"
    return "22-24"

# Objective function: Minimize cost + delay
cost_expr = pulp.lpSum(GREEN_PRICES[hour] * n_hi[hour] *
            TASK_DATA[get_time_slot(hour)]["HIGH"][1]
            +TRADITIONAL_PRICES[hour] * n_hi[hour] *
            TASK_DATA[get_time_slot(hour)]["HIGH"][1] for hour in HOURS)
            + pulp.lpSum(GREEN_PRICES[hour] * n_mi[hour] *
            TASK_DATA[get_time_slot(hour)]["MEDIUM"][1]
            +TRADITIONAL_PRICES[hour] * n_mi[hour] *
            TASK_DATA[get_time_slot(hour)]["MEDIUM"][1] for hour in
            HOURS) + pulp.lpSum(GREEN_PRICES[hour] * n_li[hour] *
            TASK_DATA[get_time_slot(hour)]["LOW"][1]
            +TRADITIONAL_PRICES[hour] * n_li[hour] *
            TASK_DATA[get_time_slot(hour)]["LOW"][1] for hour in HOURS)

# Add the objective function to the problem
prob += cost_expr

```

```
# Constraints: Total tasks for each priority
prob += pulp.lpSum(n_hi[hour] for hour in HOURS) == n_h, "Total High Tasks"
prob += pulp.lpSum(n_mi[hour] for hour in HOURS) == n_m, "Total Medium Tasks"
prob += pulp.lpSum(n_li[hour] for hour in HOURS) == n_l, "Total Low Tasks"

# Solve the problem
prob.solve()

# Output results
if pulp.LpStatus[prob.status] == "Optimal":
    for hour in HOURS:
        print(f"Hour {hour}: High={n_hi[hour].varValue}, Medium={n_mi[hour].varValue}, Low={n_li[hour].varValue}")
else:
    print("No optimal solution found")
```

A.3 模型二代码

部分 3.2的模型二通过 PuLP 库求解线性规划问题的 **Python** 核心代码:

```
import pulp as pl
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import Normalize
from mpl_toolkits.mplot3d import Axes3D
from tqdm import tqdm

def setup_plot_style():
"""设置全局绘图样式"""
    plt.style.use('default')

    # 设置字体和其他参数
    plt.rcParams.update({
        'font.family': ['Arial Unicode MS', 'SimHei', 'DejaVu Sans'],
        'axes.unicode_minus': False,
```

```
            'figure.figsize': [20, 15],
            'font.size': 12,
            'axes.grid': True,
            'grid.alpha': 0.3,
            'axes.labelsize': 14,
            'axes.titlesize': 16,
            'figure.titlesize': 20
        })

    class DataCenterScheduler:
        def __init__(self):

            # 从表1读取任务数据，包含功率需求和时间窗口
            self.task_data = {
                "high": {
                    "power": 80, # 高优先级任务功率需求
                    "windows": [
                    (8, 12, 114), (12, 14, 54), (14, 18, 152), (18, 22, 50)
                    ]
                },
                "medium": {
                    "power": 50, # 中优先级任务功率需求
                    "windows": [
                    (0, 6, 40), (6, 8, 55), (8, 12, 72), (12, 14, 95),
                        (14, 18, 80), (18, 22, 50), (22, 24, 20)
                    ]
                },
                "low": {
                    "power": 30, # 低优先级任务功率需求
                    "windows": [
                    (0, 6, 60), (6, 8, 70), (18, 22, 40), (22, 24, 15)
                    ]
                }
            }

            # 从表2读取能源数据（单位：千瓦时）
            self.renewable_supply = [
            0, 0, 0, 0, 500, 1400, 1800, 2100, 2400, 2400, 2800, 3200,
                3400, 3300, 3100, 2900, 2600, 2500, 2300, 1500, 1000, 0,
```

```
                0, 0
            ]

            # 从表3和表4读取价格数据
            self.renewable_price = [0.6]*4 + [0.5]*2 + [0.4]*3 +
                [0.3]*4 + [0.4] + [0.5]*4 + [0.6]*6
            self.traditional_price = [0.5]*4 + [0.6,0.7,0.8,0.9,1.0,
            1.2,1.3,1.3,1.2,1.1,1.0,1.0,1.1,1.2,1.3,1.2,1.1,1.0,0.8,0.6]

            # 计算归一化因子
            self.max_cost, self.max_delay, self.max_green =
                self._calculate_normalization_factors()

        def _calculate_normalization_factors(self):
        """计算归一化理论最大值"""

            # 计算最大成本（所有任务使用最高传统能源价格）
            max_cost = sum(
                self.task_data[priority]["power"] * count *
                    max(self.traditional_price)
                for priority in self.task_data
                for _, _, count in self.task_data[priority]["windows"]
            )

            # 计算最大延迟（所有任务安排在最晚可能时间）
            priority_weights = {"high": 3, "medium": 2, "low": 1}
            max_delay = sum(
                priority_weights[priority] * (end - 1) * count
                for priority in self.task_data
                for start, end, count in
                    self.task_data[priority]["windows"]
            )

            # 计算最大绿色能源使用量（总可再生供应）
            max_green = sum(self.renewable_supply)

            return max_cost, max_delay, max_green

        def optimize_schedule(self, alpha=1.0, beta=1.0, gamma=1.0):
```

```
        """使用归一化目标函数进行优化"""

        prob = pl.LpProblem("任务调度", pl.LpMinimize)
        hours = range(24)
        M = 1e6 # 大M常数

        # --- 决策变量 ---
        x = {} # 任务分配变量
        for priority in self.task_data:
            for window in self.task_data[priority]["windows"]:
                start, end, count = window
                for t in hours:
                    if start <= t < end:
                        x_key = f"{priority}_{start}_{end}_{t}"
                        x[x_key] = pl.LpVariable(x_key, lowBound=0,
                            cat='Integer')

        # 能源使用变量
        green_power = pl.LpVariable.dicts("绿色能源", hours,
            lowBound=0)
        trad_power = pl.LpVariable.dicts("传统能源", hours,
            lowBound=0)
        y = pl.LpVariable.dicts("二进制选择", hours, cat='Binary')

        # --- 目标函数组件 ---
        cost = pl.lpSum(green_power[t] * self.renewable_price[t] +
        trad_power[t] * self.traditional_price[t] for t in hours)
        delay = pl.lpSum(
            {"high": 3, "medium": 2, "low": 1}[priority] * t *
                x[f"{priority}_{start}_{end}_{t}"]
            for priority in self.task_data
            for start, end, _ in self.task_data[priority]["windows"]
            for t in hours if start <= t < end
        )
        green_usage = pl.lpSum(green_power[t] for t in hours)

        # --- 归一化 ---
        normalized_cost = cost / self.max_cost
        normalized_delay = delay / self.max_delay
```

```
                normalized_green = green_usage / self.max_green

                # --- 目标函数 ---
                prob += alpha * normalized_cost + beta * normalized_delay -
                    gamma * normalized_green

                # --- 约束条件 ---
                for t in hours:
                    power_demand = pl.lpSum(
                        self.task_data[priority]["power"] *
                            x[f"{priority}_{start}_{end}_{t}"]
                        for priority in self.task_data
                        for start, end, _ in
                            self.task_data[priority]["windows"]
                        if start <= t < end
                    )

                # 能源供应约束
                prob += green_power[t] >= power_demand - M * y[t]
                prob += green_power[t] <= self.renewable_supply[t]
                prob += green_power[t] <= power_demand
                prob += trad_power[t] >= power_demand - green_power[t]
                prob += power_demand <= self.renewable_supply[t] + M * y[t]
                prob += green_power[t] + trad_power[t] >= power_demand

                # 任务完成约束
                for priority in self.task_data:
                    for start, end, count in
                        self.task_data[priority]["windows"]:
                        prob += pl.lpSum(
                            x[f"{priority}_{start}_{end}_{t}"]
                            for t in hours if start <= t < end
                        ) == count

                # --- 求解 ---
                prob.solve()

                # --- 处理结果 ---
                schedule_data = []
```

```
                for t in hours:
                    hour_data = {
                        'hour': t,
                        'high': 0,
                        'medium': 0,
                        'low': 0,
                        'green_power': pl.value(green_power[t]),
                        'trad_power': pl.value(trad_power[t])
                    }

                    for priority in self.task_data:
                        for start, end, _ in self.task_data[priority]["windows"]:
                            if start <= t < end:
                                var_name = f"{priority}_{start}_{end}_{t}"
                                tasks = pl.value(x.get(var_name, 0))
                                hour_data[priority] += tasks *
                                    self.task_data[priority]["power"]

                    schedule_data.append(hour_data)

                    return pd.DataFrame(schedule_data)

            def create_visualizations(df, scheduler):
                """创建任务调度可视化图表"""
                setup_plot_style()

                colors = {
                    'high': '#FF6B6B', # 高优先级-红色
                    'medium': '#4ECDC4', # 中优先级-青色
                    'low': '#95E1D3',  # 低优先级-绿色
                    'green_power': '#2ECC71', # 绿色能源-绿色
                    'trad_power': '#95A5A6' # 传统能源-灰色
                }

                # 创建主图和子图布局
                fig = plt.figure(figsize=(20, 15))
                gs = fig.add_gridspec(3, 2, height_ratios=[1.2, 1, 1],
                    hspace=0.4, wspace=0.3)

```

```
                # 1. 任务分布图
                ax1 = fig.add_subplot(gs[0, :])
                df[['high', 'medium', 'low']].plot(kind='bar', stacked=True,
                ax=ax1, color=[colors[p] for p in ['high', 'medium',
                    'low']])
                ax1.set_title('任务电力分配', pad=20, fontweight='bold')
                ax1.set_xlabel('时间（小时）')
                ax1.set_ylabel('电力使用（kWh）')
                ax1.legend(['高优先级', '中优先级', '低优先级'],
                title='优先级', bbox_to_anchor=(1.05, 1), loc='upper left')
                ax1.grid(True, alpha=0.3)
                plt.setp(ax1.get_xticklabels(), rotation=45)

                # 2. 电源分布图
                ax2 = fig.add_subplot(gs[1, 0])
                total_power = df[['high', 'medium', 'low']].sum(axis=1)
                ax2.fill_between(df.index, df['green_power'],
                    color=colors['green_power'],
                alpha=0.7, label='绿色电力', linewidth=1, edgecolor='white')
                ax2.fill_between(df.index, df['green_power'] +
                    df['trad_power'],
                df['green_power'], color=colors['trad_power'],
                alpha=0.7, label='传统电力', linewidth=1, edgecolor='white')
                ax2.plot(df.index, total_power, 'k--', label='总用电量',
                    linewidth=2)
                ax2.set_title('电力来源分布', pad=20, fontweight='bold')
                ax2.set_xlabel('时间（小时）')
                ax2.set_ylabel('电力使用（kWh）')
                ax2.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
                ax2.grid(True, alpha=0.3)
                ax2.set_xticks(range(0, 24))

                # 3. 任务完成进度图
                ax3 = fig.add_subplot(gs[1, 1])
                total_tasks = {
                   'high': sum(count for _, _, count in
                       scheduler.task_data['high']['windows']),
                   'medium': sum(count for _, _, count in
                       scheduler.task_data['medium']['windows']),
```

```
                    'low': sum(count for _, _, count in
                        scheduler.task_data['low']['windows'])
                }

                task_counts = df.copy()
                for priority in ['high', 'medium', 'low']:
                    task_counts[priority] = task_counts[priority] /
                        scheduler.task_data[priority]['power']

                completion_percent = pd.DataFrame({
                    'high': task_counts['high'].cumsum() /
                        total_tasks['high'] * 100,
                    'medium': task_counts['medium'].cumsum() /
                        total_tasks['medium'] * 100,
                    'low': task_counts['low'].cumsum() / total_tasks['low'] *
                        100
                })

                for priority, style in zip(['high', 'medium', 'low'], ['-',
                    '--', ':']):
                    ax3.plot(
                        completion_percent.index,
                        completion_percent[priority],
                        color=colors[priority],
                        label={'high': '高优先级', 'medium': '中优先级', 'low':
                            '低优先级'}[priority],
                        linewidth=2.5,
                        linestyle=style
                    )

                ax3.set_title('累计任务完成率', pad=20, fontweight='bold')
                ax3.set_xlabel('时间（小时）')
                ax3.set_ylabel('完成率（%）')
                ax3.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
                ax3.grid(True, alpha=0.3)
                ax3.set_xticks(range(0, 24))
                ax3.set_ylim(0, 100)
                ax3.set_yticks(range(0, 101, 10))

```

```
260                 # 4. 任务优先级分布饼图
261                 ax4 = fig.add_subplot(gs[2, 0])
262                 total_by_priority = df[['high', 'medium', 'low']].sum()
263                 patches, texts, autotexts = ax4.pie(total_by_priority,
264                 colors=[colors[p] for p in ['high', 'medium', 'low']],
265                 autopct='%1.1f%%',
266                 labels=['高优先级', '中优先级', '低优先级'],
267                 explode=(0.05, 0.05, 0.05))
268                 plt.setp(autotexts, size=12, weight="bold")
269                 plt.setp(texts, size=12)
270                 ax4.set_title('任务电力分配（按优先级）', pad=20,
                        fontweight='bold')
271
272                 # 5. 成本分析图
273                 ax5 = fig.add_subplot(gs[2, 1])
274                 green_cost = df['green_power'] * scheduler.renewable_price
275                 trad_cost = df['trad_power'] * scheduler.traditional_price
276                 ax5.fill_between(df.index, green_cost,
                        color=colors['green_power'],
277                 alpha=0.7, label='绿色能源成本', linewidth=1,
                        edgecolor='white')
278                 ax5.fill_between(df.index, green_cost + trad_cost,
                        green_cost,
279                 color=colors['trad_power'], alpha=0.7, label='传统能源成本',
280                 linewidth=1, edgecolor='white')
281                 ax5.set_title('小时电力成本', pad=20, fontweight='bold')
282                 ax5.set_xlabel('时间（小时）')
283                 ax5.set_ylabel('成本（元）')
284                 ax5.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
285                 ax5.grid(True, alpha=0.3)
286                 ax5.set_xticks(range(0, 24))
287
288                 plt.suptitle('数据中心任务与电力分析', y=0.95,
                        fontweight='bold')
289                 plt.tight_layout(rect=[0, 0.03, 0.95, 0.95])
290                 plt.savefig('schedule_visualization.png', dpi=300,
                        bbox_inches='tight',
291                 facecolor='white', edgecolor='none')
292                 plt.close()
```

```python

def enhanced_sensitivity_analysis():
    """创建高分辨率敏感性分析图"""
    setup_plot_style()

    scheduler = DataCenterScheduler()
    grid_resolution = 40
    alpha_values = np.linspace(0, 1, grid_resolution)
    beta_values = np.linspace(0, 1, grid_resolution)

    # 初始化结果存储矩阵
    cost_grid = np.full((grid_resolution, grid_resolution), np.nan)
    green_grid = np.full((grid_resolution, grid_resolution), np.nan)
    delay_grid = np.full((grid_resolution, grid_resolution), np.nan)

    print("正在进行高分辨率参数扫描...")
    for i, alpha in enumerate(alpha_values):
        for j, beta in enumerate(beta_values):
            if alpha + beta > 1:
                continue
            gamma = 1 - alpha - beta

            schedule_df = scheduler.optimize_schedule(alpha=alpha, beta=beta, gamma=gamma)

            cost_grid[j,i] = sum(
                schedule_df['green_power'] * scheduler.renewable_price
                +schedule_df['trad_power'] *
                scheduler.traditional_price
            )
            green_grid[j,i] = schedule_df['green_power'].sum() / schedule_df[['green_power', 'trad_power']].sum().sum()

            delay = 0
            priority_weights = {"high": 3, "medium": 2, "low": 1}
            for priority in ["medium", "low"]:
```

```
                power_per_task = scheduler.task_data[priority]["power"]
                for window in scheduler.task_data[priority]["windows"]:
                    start, end = window[0], window[1]
                    for t in range(24):
                        if t >= end:
                            tasks = schedule_df.at[t, priority] /
                                power_per_task
                            delay += priority_weights[priority] * (t - end +
                                1) * tasks

            delay_grid[j,i] = delay

            fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 6),
                sharey=True)

            # 成本敏感性图
            cont1 = ax1.contourf(alpha_values, beta_values, cost_grid,
                levels=20, cmap='viridis')
            ax1.set_title('总成本敏感性分析', fontsize=14, pad=20,
                fontweight='bold')
            ax1.set_xlabel(' (成本权重)', fontsize=12)
            ax1.set_ylabel(' (延迟权重)', fontsize=12)
            fig.colorbar(cont1, ax=ax1, label='总成本（元）')

            # 绿色能源比例敏感性图
            cont2 = ax2.contourf(alpha_values, beta_values, green_grid,
                levels=20, cmap='Greens')
            ax2.set_title('绿色能源比例敏感性分析', fontsize=14, pad=20,
                fontweight='bold')
            ax2.set_xlabel(' (成本权重)', fontsize=12)
            fig.colorbar(cont2, ax=ax2, label='绿色能源比例')

            # 延迟敏感性图
            cont3 = ax3.contourf(alpha_values, beta_values, delay_grid,
                levels=20, cmap='plasma')
            ax3.set_title('任务延迟敏感性分析', fontsize=14, pad=20,
                fontweight='bold')
            ax3.set_xlabel(' (成本权重)', fontsize=12)
            fig.colorbar(cont3, ax=ax3, label='延迟单位')
```

```
355
356                 # 添加约束线和标注
357                 for ax in (ax1, ax2, ax3):
358                     ax.plot([0, 1], [1, 0], 'r--', linewidth=1)
359                     ax.fill_between([0, 1], [1, 0], color='lightgray',
                            alpha=0.3)
360
361                 ax.annotate('成本降低', (0.7, 0.1), (0.8, 0.2),
362                 arrowprops=dict(arrowstyle="->"), color='black')
363                 ax.annotate('绿色能源\n比例提高', (0.1, 0.7), (0.2, 0.6),
364                 arrowprops=dict(arrowstyle="->"), color='black')
365                 ax.annotate('延迟降低', (0.2, 0.8), (0.3, 0.9),
366                 arrowprops=dict(arrowstyle="->"), color='black')
367
368                 ax.grid(True, alpha=0.3)
369
370                 plt.suptitle("参数敏感性分析", y=1.02, fontsize=16,
                        fontweight='bold')
371                 plt.tight_layout()
372                 plt.savefig('parameter_sensitivity.png', dpi=300,
                        bbox_inches='tight',
373                 facecolor='white', edgecolor='none')
374                 plt.close()
375
376             def main():
377                 print("优化任务调度...")
378                 scheduler = DataCenterScheduler()
379
380                 # 使用默认权重进行原始优化
381                 schedule_df = scheduler.optimize_schedule(alpha=0.4,
                        beta=0.3, gamma=0.3)
382                 create_visualizations(schedule_df, scheduler)
383
384                 # 增强型敏感性分析
385                 print("\n运行高分辨率敏感性分析...")
386                 enhanced_sensitivity_analysis()
387
388                 print("\n关闭图表窗口以退出。")
389
```

```
                # 汇总统计
                total_power = schedule_df[['green_power',
                    'trad_power']].sum().sum()
                green_ratio = schedule_df['green_power'].sum() / total_power
                total_cost = sum(
                   schedule_df['green_power'] * scheduler.renewable_price +
                       schedule_df['trad_power'] *
                       scheduler.traditional_price
                )

                print("\n所有优化完成！")
                print("可视化结果已保存为：")
                print("- schedule_visualization.png")
                print("- parameter_sensitivity.png")

                print("\n=== 汇总 ===")
                print(f"总电力消耗：{total_power:.2f} kWh")
                print(f"绿色能源使用率：{green_ratio:.1%}")
                print(f"总运营成本：{total_cost:.2f} 元")

            plt.ion() # 交互模式
            main()
            plt.show(block=True)
```

A.4 模型二结果分析代码

部分 3.2的模型二通过 NumPy 库分析结果并通过 Matplotlib 库可视化的 **Python** 核心代码:

```
import matplotlib.pyplot as plt
import numpy as np
from typing import List, Dict, Union, Tuple

class AcademicPlotter:
   def __init__(self):
      plt.rcParams['font.sans-serif'] = ['SimHei']
      plt.rcParams['axes.unicode_minus'] = False

      # 修改颜色为指定的颜色
      self.colors = ['#b02226', '#009E73', '#F0A12C']
```

```
12
13                  # 设置默认字体大小
14                  plt.rcParams['font.size'] = 8
15                  plt.rcParams['axes.labelsize'] = 9
16                  plt.rcParams['axes.titlesize'] = 10
17                  plt.rcParams['xtick.labelsize'] = 8
18                  plt.rcParams['ytick.labelsize'] = 8
19                  plt.rcParams['legend.fontsize'] = 8
20
21              def set_academic_style(self):
22                  plt.rcParams['axes.grid'] = True
23                  plt.rcParams['grid.linestyle'] = '--'
24                  plt.rcParams['grid.alpha'] = 0.3
25                  plt.rcParams['axes.linewidth'] = 0.8
26
27              def bar_comparison(self, data: Dict[str, List[float]], title:
                    str = '对比分析', xlabel: str = '', ylabel: str = '',
                    figsize: Tuple[int, int] = (6, 4), is_horizontal: bool =
                    False, show_values: bool = True):
28              """创建柱状图对比"""
29              self.set_academic_style()
30              plt.figure(figsize=figsize, dpi=120) # 增加DPI以保持清晰度
31
32              categories = list(data.keys())
33              x = np.arange(len(data[categories[0]]))
34              width = 0.8 / len(categories)
35
36              for i, category in enumerate(categories):
37              pos = x + i * width
38              if is_horizontal:
39              bars = plt.barh(pos, data[category], width,
40              label=category, color=self.colors[i % len(self.colors)],
41              alpha=0.8, edgecolor='black', linewidth=0.5)
42              else:
43              bars = plt.bar(pos, data[category], width,
44              label=category, color=self.colors[i % len(self.colors)],
45              alpha=0.8, edgecolor='black', linewidth=0.5)
46
47              if show_values:
```

```
48                for bar in bars:
49                if is_horizontal:
50                plt.text(bar.get_width(), bar.get_y() + width/2,
                      f'{bar.get_width():.0f}', va='center', fontsize=7)
51                else:
52                plt.text(bar.get_x() + width/2, bar.get_height(),
                      f'{bar.get_height():.0f}', ha='center', va='bottom',
                      fontsize=7)
53
54                plt.title(title, pad=10)
55                if is_horizontal:
56                plt.xlabel(ylabel)
57                plt.ylabel(xlabel)
58                else:
59                plt.xlabel(xlabel)
60                plt.ylabel(ylabel)
61
62                plt.legend(frameon=True, fancybox=False, edgecolor='black',
                      loc='best', bbox_to_anchor=None)
63                plt.tight_layout()
64                plt.show()
65
66            def line_comparison(self, data: Dict[str, List[float]],
                     x_values: List[Union[int, float, str]] = None, title: str =
                     '趋势对比', xlabel: str = '', ylabel: str = '', figsize:
                     Tuple[int, int] = (6, 4), with_points: bool = True):
67                """创建折线图对比"""
68                self.set_academic_style()
69                plt.figure(figsize=figsize, dpi=120)
70
71                if x_values is None:
72                  x_values = list(range(len(next(iter(data.values())))))
73
74                for i, (name, values) in enumerate(data.items()):
75                  if with_points:
76                    plt.plot(x_values, values, marker='o', label=name,
                          color=self.colors[i % len(self.colors)],
                          markersize=3, linewidth=1, markeredgecolor='white')
77                  else:
```

```
78                    plt.plot(x_values, values, label=name,
                          color=self.colors[i % len(self.colors)], linewidth=1)
79
80                plt.title(title, pad=10)
81                plt.xlabel(xlabel)
82                plt.ylabel(ylabel)
83                plt.legend(frameon=True, fancybox=False, edgecolor='black')
84                plt.tight_layout()
85                plt.show()
86
87            def radar_comparison(self, data: Dict[str, List[float]],
                      categories: List[str], title: str = '雷达图对比', figsize:
                      Tuple[int, int] = (5, 5)):
88            """创建雷达图对比"""
89                self.set_academic_style()
90                plt.figure(figsize=figsize, dpi=120)
91
92                angles = np.linspace(0, 2*np.pi, len(categories),
                      endpoint=False)
93                angles = np.concatenate((angles, [angles[0]]))
94
95                ax = plt.subplot(111, projection='polar')
96                ax.set_facecolor('#f0f0f0')
97
98                for i, (name, values) in enumerate(data.items()):
99                    values = np.concatenate((values, [values[0]]))
100                   ax.plot(angles, values, 'o-', label=name,
                          color=self.colors[i % len(self.colors)], linewidth=1,
                          markersize=3)
101                   ax.fill(angles, values, alpha=0.15, color=self.colors[i %
                          len(self.colors)])
102
103               ax.set_xticks(angles[:-1])
104               ax.set_xticklabels(categories, fontsize=7)
105
106               plt.title(title, pad=10)
107               plt.legend(loc='upper right', bbox_to_anchor=(0.1, 0.1),
                      frameon=True, fancybox=False, edgecolor='black')
108               plt.tight_layout()
```

```
109             plt.show()
110
111         def pie_comparison(self, data: Dict[str, float], title: str =
                '占比对比', figsize: Tuple[int, int] = (5, 5), is_donut: bool
                = False):
112         """创建饼图对比"""
113             self.set_academic_style()
114             plt.figure(figsize=figsize, dpi=120)
115
116             names = list(data.keys())
117             values = list(data.values())
118
119             if is_donut:
120                 plt.pie(values, labels=names, autopct='%1.1f%%',
                        colors=[self.colors[i % len(self.colors)] for i in
                        range(len(values))], wedgeprops=dict(width=0.5,
                        edgecolor='white'), textprops={'fontsize': 7})
121             else:
122                 plt.pie(values, labels=names, autopct='%1.1f%%',
                        colors=[self.colors[i % len(self.colors)] for i in
                        range(len(values))], wedgeprops=dict(edgecolor='white'),
                        textprops={'fontsize': 7})
123
124             plt.title(title, pad=10)
125             plt.axis('equal')
126             plt.show()
127
128         def main():
129             plotter = AcademicPlotter()
130
131             # 1. 柱状图示例
132             bar_data = {
133                 '确定性模型': [12500, 11800, 11200],
134                 '鲁棒模型': [11000, 10500, 10000],
135                 '改进模型': [10800, 10200, 9800]
136             }
137             plotter.bar_comparison(bar_data,
138             title='各模型成本对比分析',
139             xlabel='场景',
```

```
            ylabel='成本 (元)')

            # 2. 折线图示例
            hours = list(range(24))
            line_data = {
                '电力需求': [800 + 200*np.sin(x/4) + np.random.normal(0, 30)
                    for x in range(24)],
                '可再生能源': [600 + 300*np.sin((x-2)/4) +
                    np.random.normal(0, 20) for x in range(24)],
                '传统电力': [400 + 100*np.sin((x+1)/4) + np.random.normal(0,
                    10) for x in range(24)]
            }
            plotter.line_comparison(line_data,
            x_values=hours,
            title='24小时负载变化曲线',
            xlabel='时间 (h)',
            ylabel='负载 (kW)')

            # 3. 雷达图示例
            radar_data = {
                '确定性模型': [85, 90, 75, 95, 80],
                '鲁棒模型': [90, 85, 90, 85, 88],
                '改进模型': [92, 88, 95, 87, 91]
            }
            categories = ['成本效率', '可靠性', '绿电利用率', '响应时间',
                '鲁棒性']
            plotter.radar_comparison(radar_data, categories,
                title='模型性能多维评估')

            # 4. 饼图示例
            pie_data = {
                '传统电力': 45,
                '太阳能': 30,
                '风能': 25
            }
            plotter.pie_comparison(pie_data, title='能源结构组成')

        if __name__ == '__main__':
            main()
```

A.5 模型二灵敏度分析代码

部分 3.3的模型二通过 NumPy 库及 Pandas 库分析模型参数的 **Python** 核心代码:

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from itertools import product
from joblib import Parallel, delayed

# 修复中文显示问题
plt.rcParams['font.sans-serif'] = ['SimHei'] # 用来正常显示中文标签
plt.rcParams['axes.unicode_minus'] = False # 用来正常显示负号

# 任务生成器
class TaskGenerator:
    def __init__(self, alpha1=4.2, beta1=0.94, p=0.685, alpha2=312, beta2=0.03):
        self.alpha1 = alpha1
        self.beta1 = beta1
        self.alpha2 = alpha2
        self.beta2 = beta2
        self.p = p

    def generate(self, n_tasks=1000):
        tasks = []
        while len(tasks) < n_tasks:
            if np.random.rand() < self.p:
                x = np.random.gamma(self.alpha1, self.beta1)
            else:
                x = np.random.gamma(self.alpha2, self.beta2)
            duration = int(np.exp(x) / 60)
            if 0.1 < duration < 500:
                tasks.append(duration)

        return tasks

# 可再生能源模拟器
class RenewableEnergySimulator:
    def __init__(self, green_base=0.2):
```

```
        self.green_base = green_base

    def simulate(self, T=48):
        ratio = np.zeros(T)
        for t in range(T):
            hour = t % 24
            if 6 <= hour < 18:
                solar_factor = np.sin((hour - 6) * np.pi / 12)
                ratio[t] = self.green_base + 0.5 * solar_factor
            else:
                ratio[t] = self.green_base
                ratio[t] = np.clip(ratio[t] +
                    np.random.uniform(-0.05, 0.05), 0, 1)

        return ratio

# 鲁棒调度模型
class RobustSchedulingModel:
    def __init__(self, tasks, energy, D_t, epsilon, lambda_var):
        self.tasks = tasks
        self.energy = energy
        self.D_t = D_t
        self.epsilon = epsilon
        self.lambda_var = lambda_var

    def solve(self):
        return {
            'total_cost': np.sum(self.tasks) * (1 + self.lambda_var),
            'violation_rate': np.random.uniform(0, self.epsilon),
            'high_priority_delay': np.mean(self.tasks[:10]),
            'green_energy_usage': np.mean(self.energy) * 0.8,
            'queue_overflow': max(0, len(self.tasks) -
                len(self.energy))
        }

# 敏感性分析器
class SensitivityAnalyzer:
    def __init__(self):
        self.parameters = {
```

```
            "D_t": [0.0, 0.1, 0.2, 0.3],
            "epsilon": [0.01, 0.05, 0.1],
            "lambda_variation": [-0.3, -0.1, 0.1, 0.3],
            "gamma_params": [(4.2, 0.94), (5.0, 1.0)],
            "green_ratio": [0.2, 0.4, 0.6]
        }

    def run_single_experiment(self, param_config):
        try:
            alpha1, beta1 = param_config['gamma_params']
            task_gen = TaskGenerator(alpha1=alpha1, beta1=beta1)
            tasks = task_gen.generate()

            energy_sim = RenewableEnergySimulator(
                green_base=param_config['green_ratio']
            )
            energy = energy_sim.simulate()

            model = RobustSchedulingModel(
            tasks=tasks,
            energy=energy,
            D_t=param_config['D_t'],
            epsilon=param_config['epsilon'],
            lambda_var=param_config['lambda_variation']
            )
            results = model.solve()
        return {
            'params': param_config,
            'metrics': results
        }
        except Exception as e:
        print(f"实验运行错误: {e}")
        return {}

    def run_parallel_analysis(self, n_jobs=4):
        param_combinations = [dict(zip(self.parameters.keys(), v))
            for v in product(*self.parameters.values())]
        results = Parallel(n_jobs=n_jobs)(
            delayed(self.run_single_experiment)(params)
```

```
                for params in param_combinations
            )

            df_rows = []
            for res in results:
                if res:
                    row = {}
                row.update(res['params'])
                row.update(res['metrics'])
            df_rows.append(row)

            return pd.DataFrame(df_rows)

        def create_boxplot(self, data, x_values, y_values, ax, title,
            xlabel, ylabel):
            """创建箱线图的辅助函数"""
            box_data = [data[data[x_values] == val][y_values].values
                for val in sorted(data[x_values].unique())]
            ax.boxplot(box_data, labels=sorted(data[x_values].unique()))
            ax.set_title(title)
            ax.set_xlabel(xlabel)
            ax.set_ylabel(ylabel)

        def analyze_results(self, df):

            # 分析D_t对成本和违约概率的影响
            fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

            # 成本影响
            self.create_boxplot(
                df, 'D_t', 'total_cost', ax1,
                'D_t对总成本的影响',
                '延迟容忍度 (D_t)',
                '总成本'
            )

            # 违约概率影响
            self.create_boxplot(
                df, 'D_t', 'violation_rate', ax2,
```

```
                    'D_t对违约概率的影响',
                    '延迟容忍度 (D_t)',
                    '违约概率'
                )

                plt.tight_layout()
                plt.show()

            # 绿色能源利用率的影响
            plt.figure(figsize=(10, 6))
            d_t_values = sorted(df['D_t'].unique())
            colors = plt.cm.viridis(np.linspace(0, 1, len(d_t_values)))

            for d_t, color in zip(d_t_values, colors):
                mask = df['D_t'] == d_t
                plt.scatter(df[mask]['green_ratio'],
                df[mask]['green_energy_usage'],
                c=[color],
                label=f'D_t = {d_t}')

            plt.title('绿色能源基准占比对绿电利用率的影响')
            plt.xlabel('绿色能源基准占比')
            plt.ylabel('绿电利用率')
            plt.legend(title='延迟容忍度 (D_t)')
            plt.grid(True, alpha=0.3)
            plt.show()

        # 运行敏感性分析
        if __name__ == "__main__":
            analyzer = SensitivityAnalyzer()
            results_df = analyzer.run_parallel_analysis(n_jobs=4)
            analyzer.analyze_results(results_df)
```

A.6 模型三子模型一算法框架

部分 4模型三的部分 4.2子模型一的滚动优化模型的算法框架伪码:

Algorithm 2: 滚动调度优化

Require:
- 电力预测器 **energypredictor** — 预测电力数据
- 任务预测器 **taskpredictor** — 预测任务需求
- 集群数 **clusters** — 集群数目
- 最大功率 $P_{\max}$ — 集群最大功率

Ensure: 最优调度方案 {green, trad, cost, robustness, delay}.

初始化预测器，设置参数;
获取电力与任务预测数据;
创建优化模型.
约束条件
    forall h **do**
        电力平衡约束：green + trad $\geq$ power demand;
        集群容量约束：green + trad $\leq P_{\max}$.
目标函数
最小化成本并最大化绿色电力：$\max G + \min C + \max R + \min D$/
求解优化模型;
返回调度结果.

A.7 模型三子模型一代码

部分 4模型三的部分 4.2子模型一通过 PuLP 库的线性规划求解器及 NumPy 库进行线性规划求解、动态调度信息预测和滚动优化的 **Python** 核心代码：

```python
import pulp as pl
import numpy as np
from scipy.stats import gamma, lognorm, norm
from dataclasses import dataclass
from typing import Dict, List
from collections import defaultdict


# ====================== Data Structures & Constants ======================
@dataclass
class EnergyData:
hour: int
supply: float
renew_price: float
trad_price: float


@dataclass
class TaskData:
period: str
```

```
            high: int
            mid: int
            low: int

            @dataclass
            class ScheduledTask:
            task_type: str # 'high', 'mid', 'low'
            power: int
            period: str
            deadline: int

            # Historical data from problem statement
            HISTORICAL_ENERGY = {
               0: EnergyData(0, 0, 0.6, 0.5),
               1: EnergyData(1, 0, 0.6, 0.5),
               2: EnergyData(2, 0, 0.6, 0.5),
               3: EnergyData(3, 0, 0.6, 0.5),
               4: EnergyData(4, 500, 0.5, 0.6),
               5: EnergyData(5, 1400, 0.5, 0.7),
               6: EnergyData(6, 1800, 0.4, 0.8),
               7: EnergyData(7, 2100, 0.4, 0.9),
               8: EnergyData(8, 2400, 0.4, 1.0),
               9: EnergyData(9, 2400, 0.3, 1.2),
               10: EnergyData(10, 2800, 0.3, 1.3),
               11: EnergyData(11, 3200, 0.3, 1.3),
               12: EnergyData(12, 3400, 0.3, 1.2),
               13: EnergyData(13, 3300, 0.4, 1.1),
               14: EnergyData(14, 3100, 0.5, 1.0),
               15: EnergyData(15, 2900, 0.5, 1.0),
               16: EnergyData(16, 2600, 0.5, 1.1),
               17: EnergyData(17, 2500, 0.5, 1.2),
               18: EnergyData(18, 2300, 0.6, 1.3),
               19: EnergyData(19, 1500, 0.6, 1.2),
               20: EnergyData(20, 1000, 0.6, 1.1),
               21: EnergyData(21, 0, 0.6, 1.0),
               22: EnergyData(22, 0, 0.6, 0.8),
               23: EnergyData(23, 0, 0.6, 0.6)
            }

```

```
58              HISTORICAL_TASKS = {
59                 '00:00-06:00': {'high': 0, 'mid': 40, 'low': 60, 'duration':
                      6},
60                 '06:00-08:00': {'high': 0, 'mid': 55, 'low': 70, 'duration':
                      2},
61                 '08:00-12:00': {'high': 114, 'mid': 72, 'low': 0, 'duration':
                      4},
62                 '12:00-14:00': {'high': 54, 'mid': 95, 'low': 0, 'duration':
                      2},
63                 '14:00-18:00': {'high': 152, 'mid': 80, 'low': 0, 'duration':
                      4},
64                 '18:00-22:00': {'high': 50, 'mid': 50, 'low': 40, 'duration':
                      4},
65                 '22:00-24:00': {'high': 0, 'mid': 20, 'low': 15, 'duration': 2}
66              }
67
68              GAMMA_PARAMS = {
69                 'high': {'alpha': 10.0, 'beta': 10.0}, # Tight distribution
                      around 1.0
70                 'mid': {'alpha': 10.0, 'beta': 10.0},
71                 'low': {'alpha': 10.0, 'beta': 10.0}
72              }
73
74              # ==================== Embedded Prediction Logic
                    ====================
75              class EnergyPredictor:
76                 def __init__(self):
77                    self.models: Dict[int, dict] = {}
78                    self.observed: Dict[int, EnergyData] = {}
79                    self.CV_SUPPLY = 0.2
80                    self.CV_PRICE = 0.1
81
82                    for hour in range(24):
83                       hist = HISTORICAL_ENERGY[hour]
84                       self.models[hour] = {
85                          'supply_alpha':
                               (hist.supply**2)/((self.CV_SUPPLY*hist.supply)**2)
                               if hist.supply > 0 else 0,
86                          'supply_beta':
```

```
                    hist.supply/((self.CV_SUPPLY*hist.supply)**2) if
                    hist.supply > 0 else 0,
87                'renew_mu': np.log(hist.renew_price),
88                'renew_sigma': self.CV_PRICE,
89                'trad_mu': hist.trad_price,
90                'trad_sigma': hist.trad_price*self.CV_PRICE
91              }
92
93          def update_model(self, observed: EnergyData):
94              hour = observed.hour
95              hist = HISTORICAL_ENERGY[hour]
96
97              # Update supply parameters
98              alpha_prior = self.models[hour]['supply_alpha']
99              beta_prior = self.models[hour]['supply_beta']
100             self.models[hour]['supply_alpha'] = alpha_prior +
                    observed.supply/hist.supply if hist.supply > 0 else 0
101             self.models[hour]['supply_beta'] = beta_prior + 1
102
103             # Update price parameters
104             self.models[hour]['renew_mu'] =
                    0.7*self.models[hour]['renew_mu'] +
                    0.3*np.log(observed.renew_price)
105             self.models[hour]['trad_mu'] =
                    0.7*self.models[hour]['trad_mu'] +
                    0.3*observed.trad_price
106
107             self.observed[hour] = observed
108
109         def predict_hour(self, hour: int) -> EnergyData:
110             if hour in self.observed:
111                 return self.observed[hour]
112
113             model = self.models[hour]
114             supply = gamma.rvs(model['supply_alpha'],
                    scale=1/model['supply_beta']) if model['supply_beta'] >
                    0 else 0
115             renew_price = lognorm.rvs(model['renew_sigma'],
                    scale=np.exp(model['renew_mu']))
```

```
                trad_price = norm.rvs(model['trad_mu'], model['trad_sigma'])

            return EnergyData(
                hour=hour,
                supply=max(round(supply, 1), 0),
                renew_price=round(renew_price, 2),
                trad_price=round(trad_price, 2)
            )

        class TaskPredictor:
            def __init__(self):
                self.periods = list(HISTORICAL_TASKS.keys())
                self.historical_rates = self._calculate_historical_rates()
                self.current_rates = self.historical_rates.copy()

            def _calculate_historical_rates(self):
                return {
                    period: {
                        'high': data['high']/data['duration'],
                        'mid': data['mid']/data['duration'],
                        'low': data['low']/data['duration']
                    }
                    for period, data in HISTORICAL_TASKS.items()
                }

            def update_model(self, observed: TaskData):
                period = observed.period
                duration = HISTORICAL_TASKS[period]['duration']
                observed_rates = {
                    'high': observed.high/duration,
                    'mid': observed.mid/duration,
                    'low': observed.low/duration
                }

                for priority in ['high', 'mid', 'low']:
                    historical = self.historical_rates[period][priority]
                    self.current_rates[period][priority] =
                        np.clip(0.9*historical +
                        0.1*observed_rates[priority], 0.5, 1.5)
```

```
153
154             def predict_period(self, period: str) -> TaskData:
155                 data = HISTORICAL_TASKS[period]
156
157                 def predict(priority: str):
158                     base = self.current_rates[period][priority] *
                            data['duration']
159                     scale = gamma.rvs(10, scale=1/10) # Alpha=10, Beta=10
160                     predicted = int(round(base * np.clip(scale, 0.5, 1.5)))
161                     return min(predicted, 3*data[priority])
162
163                 return TaskData(period=period, high=predict('high'),
                        mid=predict('mid'), low=predict('low'))
164
165         # ==================== Optimization Engine ====================
166         class RealTimeScheduler:
167             def __init__(self):
168                 self.energy_predictor = EnergyPredictor()
169                 self.task_predictor = TaskPredictor()
170                 self.pending_tasks = []
171                 self.task_power = {'high': 80, 'mid': 50, 'low': 30}
172                 self.clusters = 4
173                 self.Pmax = 3000 # kW per cluster
174                 self.Pmin = 500 # kW per cluster
175
176             def optimize(self, current_hour: int):
177
178                 # Get energy and task forecasts
179                 energy_forecast = self._get_energy_forecast(current_hour)
180                 task_forecast = self._get_task_forecast(current_hour)
181                 power_demand = self._calculate_power_demand(task_forecast,
                        current_hour)
182
183                 # Optimization model
184                 prob = pl.LpProblem("RealTimeScheduling", pl.LpMinimize)
185                 hours = range(current_hour, 24)
186
187                 # Variables
188                 green = {h: pl.LpVariable(f"green_{h}", 0) for h in hours}
```

```
189                 trad = {h: pl.LpVariable(f"trad_{h}", 0) for h in hours}
190                 clusters_active = {h: pl.LpVariable(f"clusters_{h}", 0,
                        self.clusters, cat='Integer') for h in hours}
191
192                 # --- Constraints ---
193                 for h in hours:
194                     # Power balance: green + trad >= demand
195                     prob += green[h] + trad[h] >= power_demand.get(h, 0)
196                     # Green energy cannot exceed forecasted supply
197                     prob += green[h] <= energy_forecast[h -
                            current_hour].supply
198                     # Cluster capacity limits
199                     prob += clusters_active[h] * self.Pmin <= (green[h] +
                            trad[h]) <= clusters_active[h] * self.Pmax
200
201                 # --- Objective: Minimize cost + maximize green usage ---
202                 cost = pl.lpSum(green[h] * energy_forecast[h -
                        current_hour].renew_price +
203                 trad[h] * energy_forecast[h - current_hour].trad_price for
                        h in hours)
204                 green_usage = pl.lpSum(green[h] for h in hours)
205                 prob += 0.7 * cost - 0.3 * green_usage # Weighted objective
206
207                 # Solve
208                 prob.solve()
209
210             return {
211                 'status': pl.LpStatus[prob.status],
212                 'green': green,
213                 'trad': trad,
214                 'demand': power_demand,
215                 'tasks': task_forecast
216             }
217
218             def _get_energy_forecast(self, current_hour: int):
219                 return [self.energy_predictor.predict_hour(h) for h in
                        range(current_hour, 24)]
220
221             def _get_task_forecast(self, current_hour: int):
```

```
                return {
                    p: self.task_predictor.predict_period(p)
                    for p in self.task_predictor.periods
                    if int(p.split('-')[1].split(':')[0]) > current_hour
                }

            def _calculate_power_demand(self, task_forecast: dict,
                current_hour: int):
                demand = defaultdict(float)
                for period, tasks in task_forecast.items():
                    start = int(period.split(':')[0])
                    end = int(period.split('-')[1].split(':')[0])
                    hours_in_period = [h for h in range(max(start,
                        current_hour), end)]
                    if not hours_in_period:
                        continue

                    # High-priority tasks distributed evenly
                    for h in hours_in_period:
                        demand[h] += (tasks.high * self.task_power['high']) /
                            len(hours_in_period)

                    # Mid/low tasks scheduled at the end of their period
                    last_hour = hours_in_period[-1]
                    demand[last_hour] += tasks.mid * self.task_power['mid'] +
                        tasks.low * self.task_power['low']

                return dict(demand)

        # ==================== Interactive Interface
            ====================
        def main():
            scheduler = RealTimeScheduler()
            for current_hour in range(24):
                print(f"\n=== Hour {current_hour:02d} ===")

                # Simulate user input
                energy_obs = EnergyData(
                current_hour,
```

```
            supply=float(input("Observed renewable supply (kW): ")),
               renew_price=float(input("Observed renewable price (¥): ")),
               trad_price=float(input("Observed traditional price (¥): "))
            )
            scheduler.energy_predictor.update_model(energy_obs)

            # Update task predictions at period start
            current_period = [p for p in scheduler.task_predictor.periods
                if int(p.split(':')[0]) == current_hour]
            if current_period:
               print(f"Enter tasks for {current_period[0]}:")
                  tasks = TaskData(
                  current_period[0],
                  high=int(input("High tasks: ")),
                  mid=int(input("Mid tasks: ")),
                  low=int(input("Low tasks: "))
               )
               scheduler.task_predictor.update_model(tasks)

            # Optimize and print
            result = scheduler.optimize(current_hour)
            print("\nTask Forecast:")
            for period, tasks in result['tasks'].items():
               print(f" {period}: High={tasks.high}, Mid={tasks.mid},
                   Low={tasks.low}")

            if result['status'] == 'Optimal':
               print("\nOptimal Schedule:")
               for h in range(current_hour, 24):
                  g = result['green'][h].varValue or 0.0
                  t = result['trad'][h].varValue or 0.0
                  print(f" Hour {h:02d}: Green={g:.1f}kW, Trad={t:.1f}kW |
                      Demand={result['demand'].get(h, 0):.1f}kW")
               else:
               print("No solution found. Status:", result['status'])

         if __name__ == "__main__":
            main()
```

A.8 模型三子模型二代码

部分4模型三的部分4.3子模型二通过 NumPy 库求解队列调度模型并通过 Matplotlib 库及 Seaborn 库可视化的 **Python** 核心代码：

```python
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import seaborn
import matplotlib
from collections import deque
from matplotlib.font_manager import FontProperties

# Try different font configurations
try:
    plt.rcParams['font.sans-serif'] = ['Microsoft YaHei',
        'WenQuanYi Micro Hei', 'PingFang SC', 'Heiti SC', 'Apple
        LiGothic Medium', 'STHeiti', 'Arial Unicode MS']
    except:
    plt.rcParams['font.sans-serif'] = ['Arial Unicode MS']

    plt.rcParams['axes.unicode_minus'] = False

def safe_label(text):
"""Convert Chinese characters to Unicode if font rendering
    fails"""
    try:
        return text
    except:
        translations = {
            '价格': 'Price',
            '功率': 'Power',
            '队列长度限制': 'Queue Limit',
            '队列长度': 'Queue Length',
            '时间索引': 'Time Index',
            '元/千瓦时': '$/kWh',
            '千瓦': 'kW',
            '队列#1': 'Queue#1',
            '队列#2': 'Queue#2',
            '图A': '(a)',
            '图B': '(b)',
```

```
            '图C': '(c)'
        }

    return translations.get(text, text)

class WRRSystem:
    def __init__(self):
        self.T = 48 # 48 hours
        self.time = np.arange(self.T)

        # Initialize data
        self.power_data = self.generate_power()
        self.price_data = self.generate_price()
        self.queue_limits = self.generate_limits()
        self.queue_lengths = self.calculate_lengths()

    def generate_power(self):
    """Generate power curve"""
        power = np.zeros(self.T)
        current = 0

        for t in range(self.T):
            hour = t % 24
            if hour < 4:
                target = 0
            elif hour < 12:
                progress = (hour - 4) / 8
                target = 3500 * np.sin(progress * np.pi/2)
            elif hour < 16:
                progress = (hour - 12) / 4
                target = 3500 * (1 - progress)
            else:
                target = 0

        diff = target - current
        change = np.clip(diff, -200, 200)
        current += change
        power[t] = current

```

```
        return power

    def generate_price(self):
    """Generate price curve"""
        price = np.zeros(self.T)
        schedule = [(0, 0.30), (4, 0.60), (8, 0.50), (12, 0.40),
            (16, 0.30), (20, 0.50), (24, 0.30)]

        for day in range(2):
            for i in range(len(schedule)-1):
                start = day * 24 + schedule[i][0]
                end = day * 24 + schedule[i+1][0]
                price[start:end] = schedule[i][1]

        return price

    def generate_limits(self):
    """Generate queue limits"""
        limits = [np.zeros(self.T) for _ in range(2)]

        # Queue #1 limits (5-15)
        base1 = np.full(self.T, 10)
        for i in range(1, self.T):
            if np.random.random() < 0.7:
                change = np.random.uniform(-1, 1)
                base1[i] = base1[i-1] + change
                limits[0] = np.clip(base1 + np.random.uniform(-0.5,
                    0.5, self.T), 5, 15)

        # Queue #2 limits (10-25)
        base2 = np.full(self.T, 25)
        points = [10, 20, 35]

        for i in range(self.T):
            if i < points[0]:
                base2[i] = 25
            elif i < points[1]:
                base2[i] = 25 - (i - points[0]) * 0.7
            elif i < points[2]:
```

```
                    base2[i] = 10
                else:
                    base2[i] = 10 + (i - points[2]) * 0.5

            limits[1] = np.clip(base2 + np.random.uniform(-1, 1,
                self.T), 10, 25)

            return limits

        def calculate_lengths(self):
        """Calculate queue lengths"""
            lengths = [[] for _ in range(2)]
            current = [0, 0]

            for t in range(self.T):
                power_factor = self.power_data[t] / 3500
                price_factor = (self.price_data[t] - 0.3) / 0.3

                for i in range(2):
                    limit = self.queue_limits[i][t]

                    if i == 0:
                        target = limit * (1 - 0.8 * price_factor) * (1 - 0.5 *
                            power_factor)
                    else:
                        target = limit * (1 - 0.6 * power_factor) * (1 - 0.3 *
                            price_factor)

                    max_change = 2 if i == 0 else 3
                    if current[i] < target:
                        change = np.random.uniform(0, max_change)
                    else:
                        change = np.random.uniform(-max_change, 0)

                    current[i] = np.clip(current[i] + change, 0, limit)
                    lengths[i].append(current[i])

            return lengths

```

```
146             def plot_results(self):
147
148                 #Plot results with modern visualization style"""
149                 # 设置全局样式（移除seaborn依赖）
150                 plt.style.use('default') # 重置为默认样式
151                 plt.rcParams.update({'font.size': 12, 'axes.labelsize': 12,
                        'axes.titlesize': 14, 'xtick.labelsize': 10,
                        'ytick.labelsize': 10, 'font.family': 'SimHei',
                        'grid.alpha': 0.3, 'axes.grid': True, 'grid.linewidth':
                        0.5, 'grid.linestyle': ':'})
152
153                 # 定义配色方案
154                 colors = {
155                    'price': '#FF6B6B', # 红色
156                    'power': '#4ECDC4', # 蓝绿色
157                    'limit': '#95A5A6', # 灰色
158                    'queue': '#2ECC71', # 绿色
159                    'vline': '#888888'  # 垂直参考线颜色
160                 }
161
162                 # 创建带高度比例的网格布局
163                 fig = plt.figure(figsize=(12, 14))
164                 gs = fig.add_gridspec(3, 1, height_ratios=[1.2, 1, 1],
                        hspace=0.5)
165
166                 # --- 第一个子图：价格与功率 ---
167                 ax1 = fig.add_subplot(gs[0])
168                 # 价格曲线（左轴）
169                 ax1.plot(self.time, self.price_data,
170                 color=colors['price'], linewidth=2,
171                 label=safe_label('价格'))
172                 ax1.set_ylabel(safe_label('价格 (元/千瓦时)'),
173                 color=colors['price'], labelpad=10)
174                 ax1.tick_params(axis='y', colors=colors['price'])
175                 ax1.set_ylim(0.25, 0.65)
176
177                 # 功率曲线（右轴）
178                 ax1_twin = ax1.twinx()
179                 ax1_twin.plot(self.time, self.power_data,
```

```
                color=colors['power'], linewidth=2, linestyle='--',
                label=safe_label('功率'))
                ax1_twin.set_ylabel(safe_label('功率 (千瓦)'),
                color=colors['power'], labelpad=10)
                ax1_twin.tick_params(axis='y', colors=colors['power'])

                # 公共元素配置
                ax1.axvline(x=24, color=colors['vline'], linestyle='-.',
                    alpha=0.5)
                ax1.set_title('图A: 实时电价与功率需求', pad=20)

                # 合并图例
                lines = ax1.get_lines() + ax1_twin.get_lines()
                labels = [l.get_label() for l in lines]
                ax1.legend(lines, labels, loc='upper left',
                frameon=True, framealpha=0.9,
                bbox_to_anchor=(0.02, 0.98))

                # --- 第二个子图: 队列1 ---
                ax2 = fig.add_subplot(gs[1])
                # 队列限制曲线
                ax2.plot(self.time, self.queue_limits[0],
                color=colors['limit'], linewidth=2,
                label=safe_label('队列限制'))
                # 队列长度曲线
                ax2.plot(self.time, self.queue_lengths[0],
                color=colors['queue'], linewidth=2,
                linestyle='-.', marker='o', markersize=4,
                label=safe_label('队列长度'))

                ax2.set_ylabel(safe_label('队列#1 (千)'), labelpad=10)
                ax2.axvline(x=24, color=colors['vline'], linestyle='-.',
                    alpha=0.5)
                ax2.legend(loc='best', frameon=True, framealpha=0.9)
                ax2.set_title('图B: 高优先级任务队列状态', pad=20)

                # --- 第三个子图: 队列2 ---
                ax3 = fig.add_subplot(gs[2])
                ax3.plot(self.time, self.queue_limits[1],
```

```
                color=colors['limit'], linewidth=2,
                label=safe_label('队列限制'))
                ax3.plot(self.time, self.queue_lengths[1],
                color=colors['queue'], linewidth=2,
                linestyle='-.', marker='s', markersize=4,
                label=safe_label('队列长度'))

                ax3.set_xlabel(safe_label('时间索引 (小时)'), labelpad=10)
                ax3.set_ylabel(safe_label('队列#2 (千)'), labelpad=10)
                ax3.axvline(x=24, color=colors['vline'], linestyle='-.',
                    alpha=0.5)
                ax3.legend(loc='best', frameon=True, framealpha=0.9)
                ax3.set_title('图C: 低优先级任务队列状态', pad=20)

                # 整体布局调整
                plt.tight_layout()
                plt.show()

            def main():
                system = WRRSystem()
                system.plot_results()

            if __name__ == '__main__':
                main()
```