算法课期末大作业报告

实验目的

本实验涉及如何实现一个图的库，包括图的存储、读写、图结构挖掘算法的实现以及图的可视化。

实验内容

一个读入图的库，并赋有实现k-core分解，精确最密子图，近似最密子图，k-clique分解，k-core动态维护以及结果可视化的功能。

由于电脑性能的限制，三个测试集并不能在我的电脑上运行。所以我选择了fb-pages-food这个测试集，有620个点和2102条边

实验步骤

Kcore分解

用了peel算法，通过移除度数小于k的节点，逐步识别图中k-core子图的节点集合。

- 运行步骤

1. **初始化**：记录开始时间，初始化节点的度数和候选队列。
2. **主循环**：

   ```
   k = 0
   while candidates:
       k += 1
       while True:
           to_remove = [node for node in candidates if degrees[node] < k]
           if not to_remove:
               break

           for node in to_remove:
               k_core[node] = k - 1
               candidates.remove(node)
               for neighbor in self.G.neighbors(node):
                   if neighbor in candidates:
                       degrees[neighbor] -= 1
   ```

   逐步增加k的值，找出度数小于k的节点并移除，更新邻居节点的度数。
3. **处理剩余节点**：

```
for node in candidates:
    k_core[node] = k
```

将剩余节点的k-core值设为当前k值。

4. **记录时间并输出**：计算算法运行时间，并将结果写入输出文件。

精确最密子图

使用了Goldberg算法，用于精确求解图的最密子图问题。该算法通过构建网络流模型，结合二分搜索与最小割算法，找出密度最大的子图。

- 运行步骤

1. **构建网络**：

```
def construct_network(G, g, m):
    N = nx.DiGraph()
    s, t = 's', 't'
    for i in G.nodes():
        N.add_edge(s, i, capacity=m)
        N.add_edge(i, t, capacity=m + 2 * g - G.degree(i))
    for u, v in G.edges():
        N.add_edge(u, v, capacity=1)
        N.add_edge(v, u, capacity=1)
    return N, s, t
```

构建一个包含源点s和汇点t的有向图N。节点与源点和汇点之间的边容量根据密度g和节点度数确定，图中的边容量为1。

2. **初始化变量**：

```
n = len(self.G.nodes())
m = len(self.G.edges())
l, u = 0, m
V1 = set()
start_time = time.time()
```

记录图的节点数和边数，初始化密度下界l和上界u，以及开始时间。

3. **二分搜索与最小割**：

```
while u - l >= 1 / (n * (n - 1)):
    g = (u + l) / 2
    N, s, t = construct_network(self.G, g, m)
    cut_value, partition = nx.minimum_cut(N, s, t, flow_func=edmonds_karp)
    S, T = partition

    if S == {s}:
        u = g
    else:
        l = g
        V1 = S - {s}
```

使用二分搜索逐步逼近最密子图的密度。构建网络后，使用Edmonds-Karp算法计算最小割。如果割集S只包含源点，则更新密度上界；否则更新密度下界并记录当前子图节点集。

4. **计算密度与输出结果**：计算子图密度为E/V，记录算法运行时间，并将结果写入输出文件。

2-近似最密子图

实现了2-近似最密子图算法。该算法通过去皮退化排序（peeling degeneracy ordering）和逐步移除节点，近似求解图的最密子图问题。

- 运行步骤

1. **去皮退化排序**：

```
def peel_degeneracy_ordering(G):
    ordering = []
    degrees = {node: deg for node, deg in G.degree()}
    while degrees:
        min_node = min(degrees, key=degrees.get)
        ordering.append(min_node)
        del degrees[min_node]
        for neighbor in G.neighbors(min_node):
            if neighbor in degrees:
                degrees[neighbor] -= 1
    return ordering
```

初始化节点度数`degrees`。

逐步移除度数最小的节点，并更新其邻居的度数。

记录移除顺序`ordering`。

2. **初始化变量**：

```
degeneracy_order = peel_degeneracy_ordering(self.G)
densest_subgraph = None
highest_density = 0
S = set(self.G.nodes())
m_S = self.G.number_of_edges()
```

获取退化排序`degeneracy_order`，初始化变量`densest_subgraph`、`highest_density`、`S`（包含所有节点的集合）和`m_S`（图中边的数量）。

3. **逐步移除节点，计算密度**：

```
for u in degeneracy_order:
    density = m_S / len(S) if len(S) > 0 else 0
    if densest_subgraph is None or density > highest_density:
        highest_density = density
        densest_subgraph = S.copy()
    S.remove(u)
    for v in self.G.neighbors(u):
        if v in S:
            m_S -= 1
```

按退化排序依次移除节点u。

计算当前子图S的密度density，若密度更高则更新最密子图densest_subgraph。

更新集合S和边数量m_S。

4. **记录时间并输出**

k-clique分解

实现了Bron-Kerbosch算法用于求解k-clique分解。该算法用于查找图中的所有极大团，并选取大小为k的团。

- Bron-Kerbosch算法

```
def bron_kerbosch(self, R, P, X, cliques):
    if len(P) == 0 and len(X) == 0:
        cliques.append(R)
        return
    for v in list(P):
        new_R = R | {v}
        new_P = P & set(self.G.neighbors(v))
        new_X = X & set(self.G.neighbors(v))
        self.bron_kerbosch(new_R, new_P, new_X, cliques)
        P.remove(v)
        X.add(v)
```

  - **递归查找极大团**:
    - 如果候选节点P和排除节点X均为空，当前集合R是一个极大团，添加到结果cliques中。
    - 否则，对于P中的每个节点v，递归地扩展团R，更新候选节点和排除节点。
- 使用Bron-Kerbosch算法进行k-clique分解
  - 查找所有极大团之后选取大小为k的团并输出即可

动态维护K_core

用了《Efficient Core Maintenance in Large Dynamic Graphs》中提到的染色算法，具体如下图所示：

- **插入新边**

Algorithm 1 Insertion(G, u, v)

Input: Graph $G = (V, E)$ and an edge (u, v)
Output: the updated core number of the nodes

```
Initialize visited(w) ← 0 for all node w ∈ V;
Initialize color(w) ← 0 for all node w ∈ V;
V_c ← ∅;
if C_u > C_v then
    c ← C_v;
    Color(G, v, c);
    RecolorInsert(G, c);
    UpdateInsert(G, c);
else
    c ← C_u;
    Color(G, u, c);
    RecolorInsert(G, c);
    UpdateInsert(G, c);
```

Algorithm 2 void **Color**(G, u, c)

```
visited(u) ← 1;
if color(u) = 0 then
    V_c ← V_c ∪ {u};
    color(u) = 1;
for each node w ∈ N(u) do
    if visited(w) = 0 and C_w = c then
        Color(G, w, c);
```

Algorithm 3 void **RecolorInsert**(G, c)

```
flag ← 0;
for each node u ∈ V_c do
    if color(u) = 1 then
        X_u ← 0;
        for each node w ∈ N(u) do
            if (color(w) = 1) or (C_w > c) then
                X_u ← X_u + 1;
        if X_u ≤ c then
            color(u) ← 0;
            flag ← 1;
if flag = 1 then
    RecolorInsert(G, c);
```

Algorithm 4 void **UpdateInsert**(G, c)

```
for each node w ∈ V_c do
    if color(w) = 1 then
        C_w ← c + 1;
```

- **删除边**

Algorithm 5 Deletion(G, u, v)

Input: Graph $G = (V, E)$ and an edge (u, v)
Output: the updated core number of the nodes

Initialize visited(w) $\leftarrow 0$ for all node $w \in V$;
Initialize color(w) $\leftarrow 0$ for all node $w \in V$;
$V_c \leftarrow \emptyset$;
if $C_u > C_v$ **then**
$c \leftarrow C_v$;
Color(G, v, c);
RecolorDelete(G, c);
UpdateDelete(G, c);
if $C_u < C_v$ **then**
$c \leftarrow C_u$;
Color(G, u, c);
RecolorDelete(G, c);
UpdateDelete(G, c);
if $C_u = C_v$ **then**
$c \leftarrow C_u$;
Color(G, u, c);
if color(v) $= 0$ **then**
Initialize visited(w) $\leftarrow 0$ for all node $w \in V$;
Color(G, v, c);
RecolorDelete(G, c);
UpdateDelete(G, c);
else
RecolorDelete(G, c);
UpdateDelete(G, c);

Algorithm 6 void **RecolorDelete**(G, c)

```
flag ← 0;
for each node u ∈ V_c do
    if color(u) = 1 then
        X_u ← 0;
        for each node w ∈ N(u) do
            if (color(w) = 1) or (C_w > c) then
                X_u ← X_u + 1;
        if X_u < c then
            color(u) ← 0;
            flag ← 1;
if flag = 1 then
    RecolorDelete(G, c);
```

Algorithm 7 void **UpdateDelete**(G, c)

```
for each node w ∈ V_c do
    if color(w) = 0 then
        C_w ← c − 1;
```

具体来讲，有以下5个辅助函数：

Color：递归着色函数，标记访问节点，并将符合条件的节点加入集合V_c。

RecolorInsert：重新着色函数，用于在插入边时更新节点的颜色状态。

UpdateInsert：更新插入操作后节点的核心数。

RecolorDelete：重新着色函数，用于在删除边时更新节点的颜色状态。

UpdateDelete：更新删除操作后节点的核心数。

和两个核心函数：

Insertion：处理插入边的操作，根据节点的核心数进行着色、重新着色和更新操作。

Deletion：处理删除边的操作，根据节点的核心数进行着色、重新着色和更新操作。

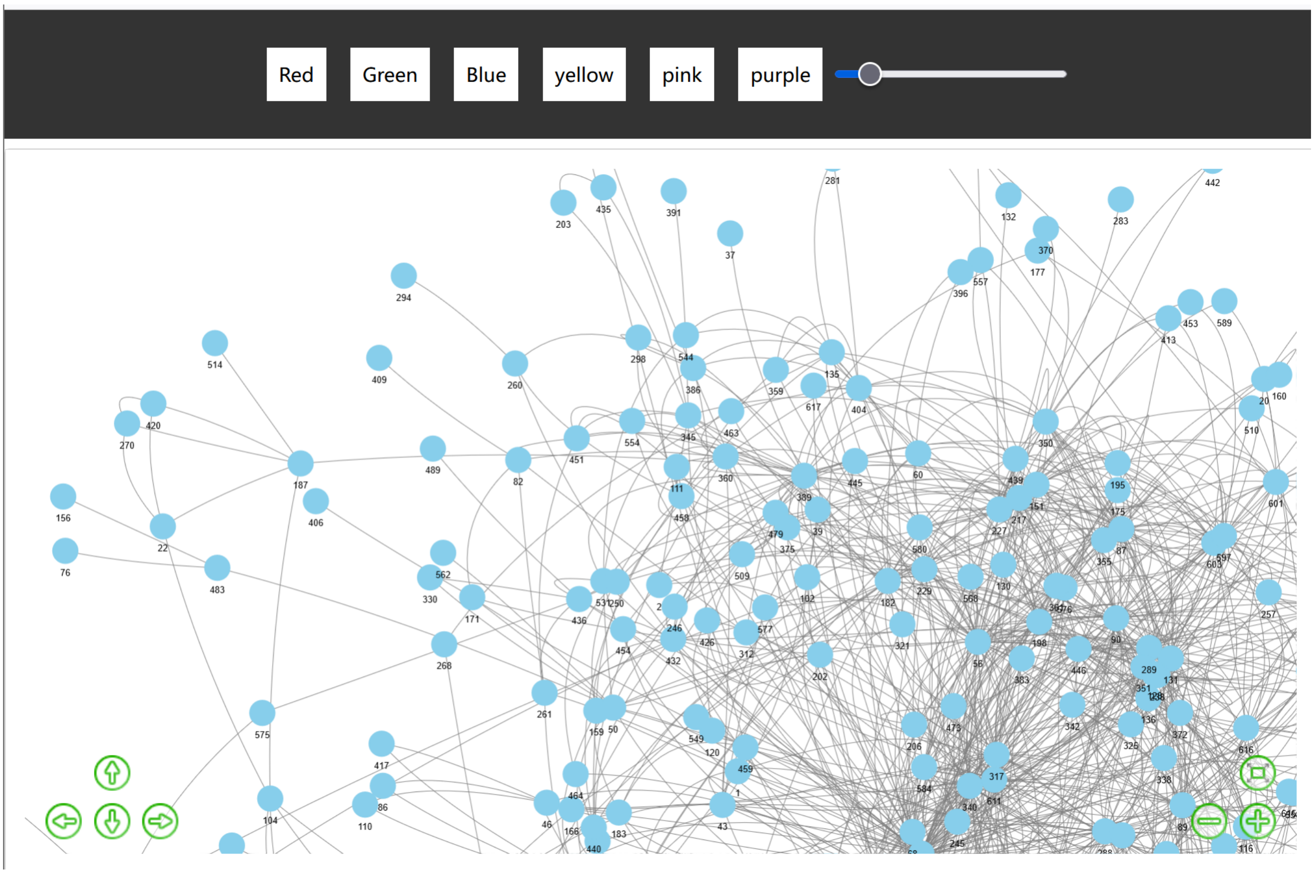
在主函数Dy_kcore中，只需要

- 根据边是否已经存在图中
- 帮**Insertion**和**Deletion**做初始化（这两个初始化是一样的，即visit，color和Vc集合）
- 判断调用**Insertion**还是**Deletion**
- 输出结果

实验结果

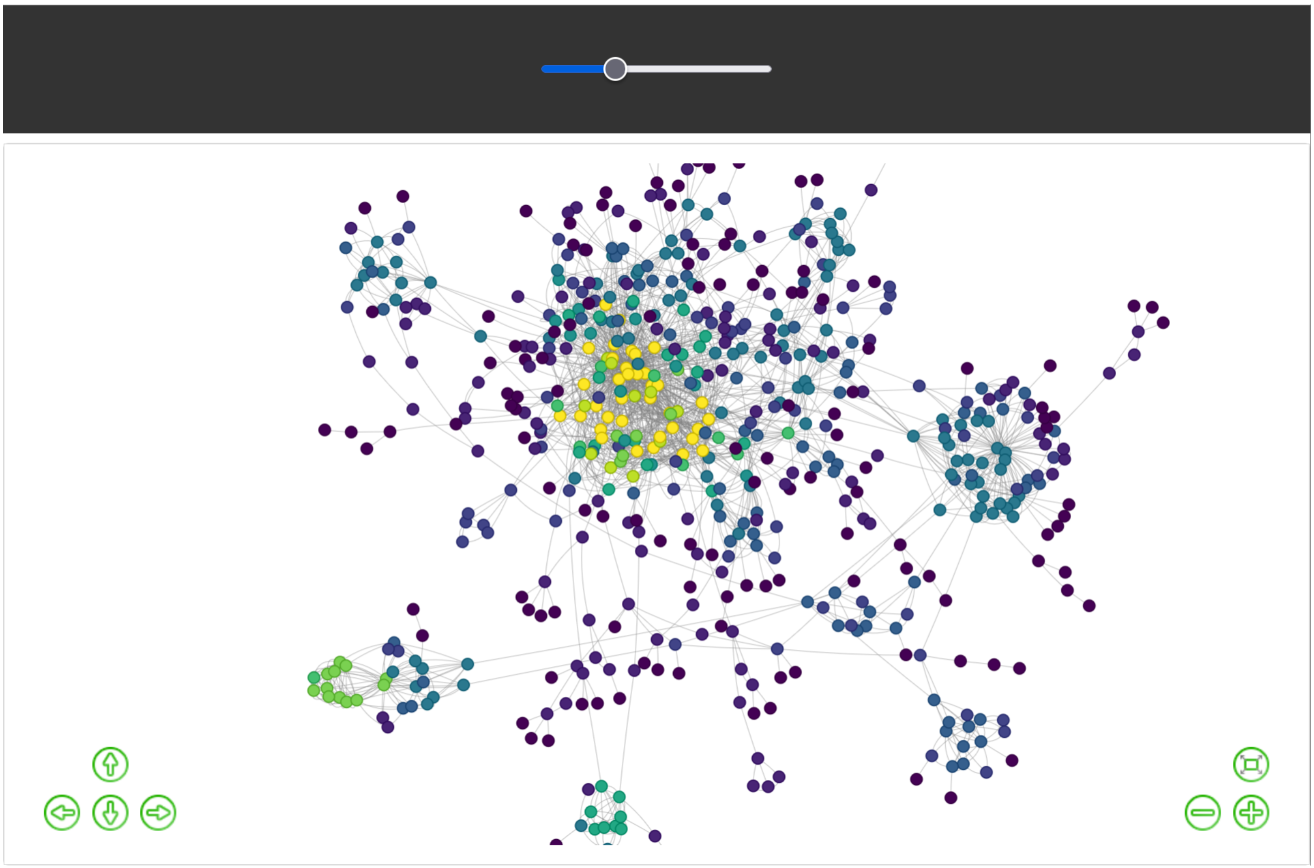
实验结果输出至output文件夹中，包括6个txt文件和6个html（可视化图）

saved.txt & graph.html



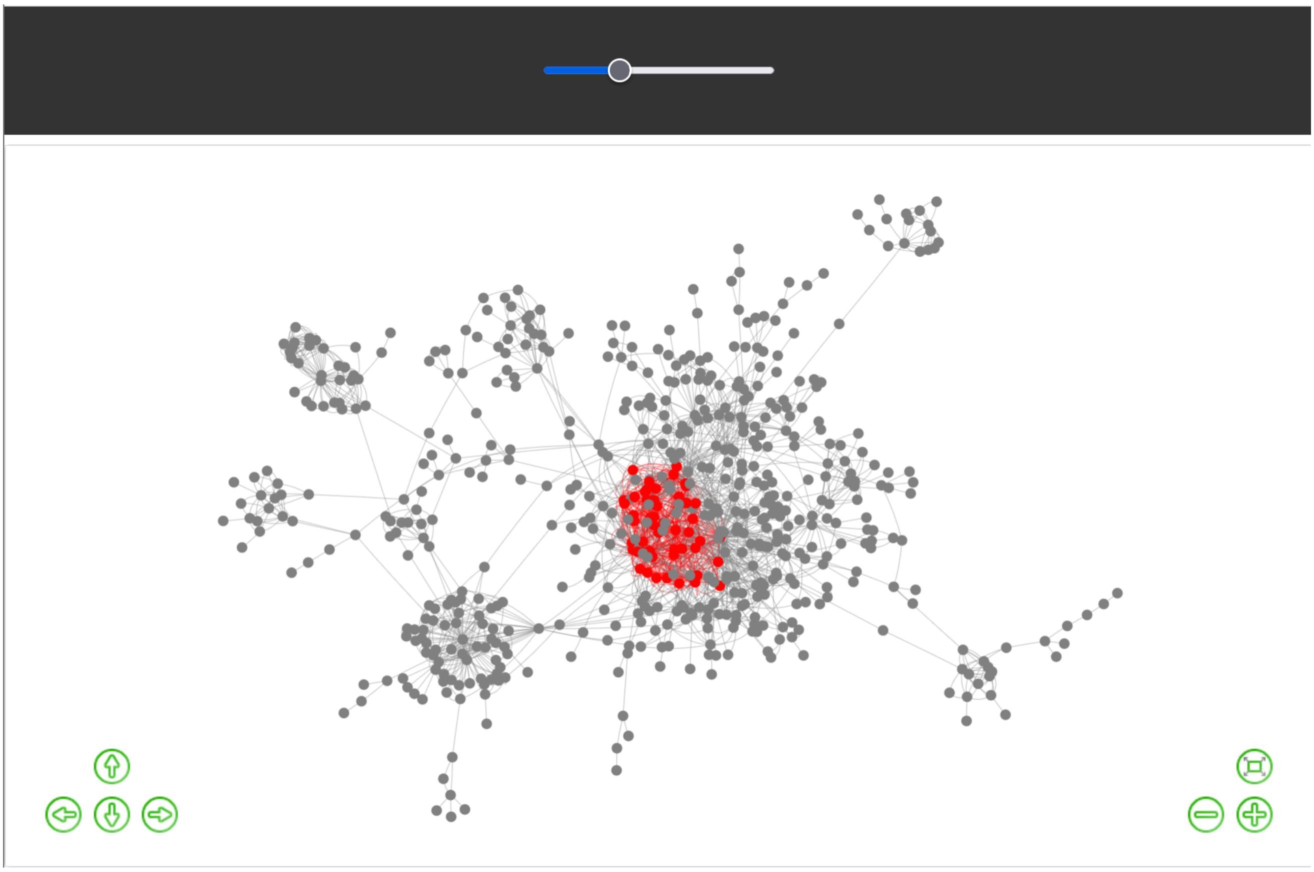
k-core.txt & kcore_visualization_before.html

将coreness相同的点用相同颜色和相同大小表示：



exact_densest.txt & exact_densest_visualization.html

最密子图用特殊颜色标注

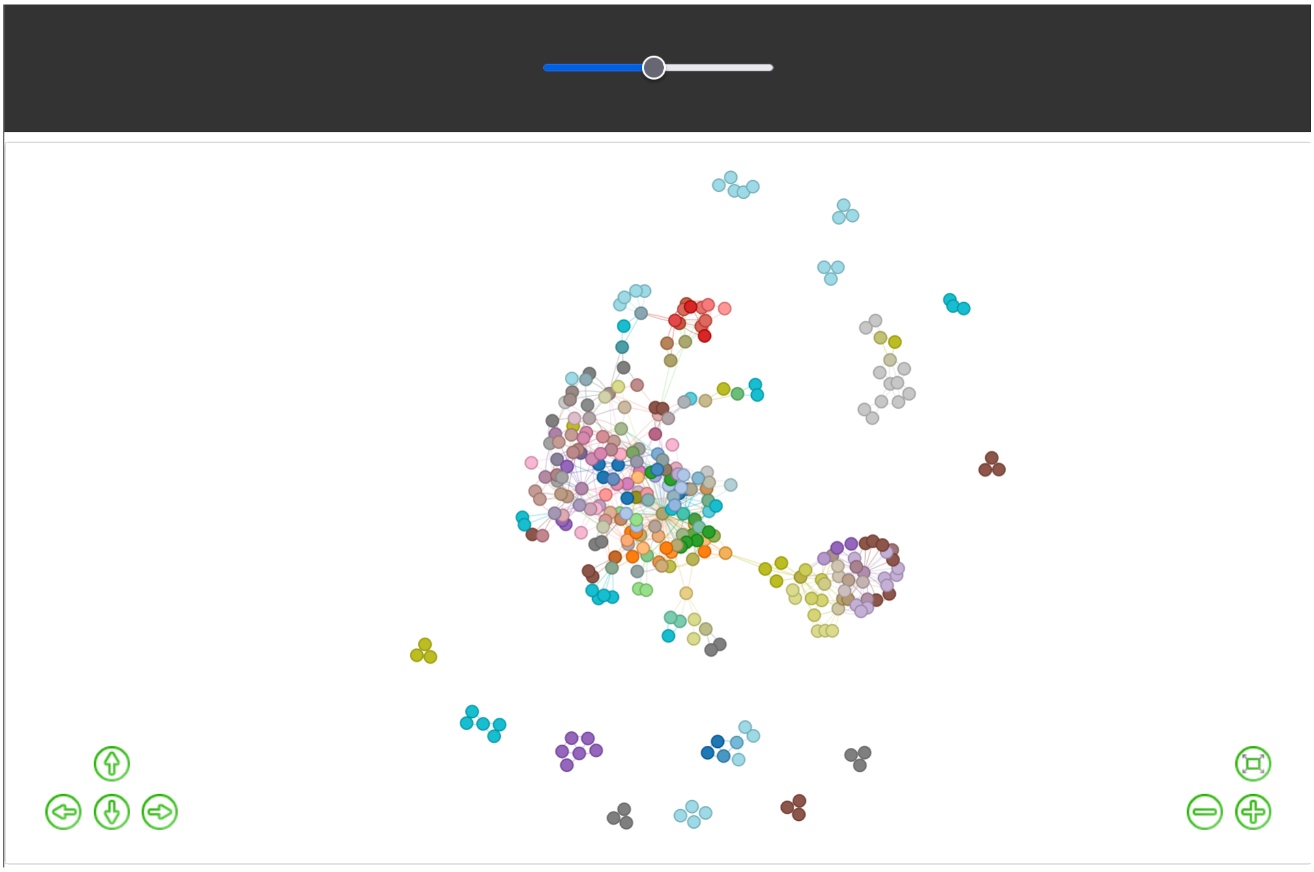


approximate_densest.txt & approximate_densest_visualization.html

和精确算法结果的表现形式一样，不赘述

k-clique.txt & cliques_visualization.html

将相同极大团的点用相同颜色表示，如一个点在不同极大团中，则使用极大团颜色的混合。（以下是k=3的结果）



dynamic_Kcore.txt & kcore_visualization_after.html

和k-core.txt & kcore_visualization_before.html的表现一样，不赘述。