

## 13 | Ranges实战：数据序列函数式编程

2023-02-20 卢誉声 来自北京

《现代C++20实战高手课》

课程介绍 >



讲述：卢誉声

时长 17:15 大小 15.76M



你好，我是卢誉声。

通过前面的学习，我们已经了解到，C++ Ranges 作为基础编程工具，可以大幅加强函数式编程的代码可读性和可维护性，解决了 C++ 传统函数式编程的困境。在 C++20 的加持下，我们终于可以优雅地处理大规模数据了。

在讲解完 Ranges 的概念和用法后，我们还是有必要通过实战来融会贯通 C++ Ranges。它的用法比较灵活，在熟练使用后，我相信你会在今后的代码实现中对它爱不释手。

在处理规模型数据时，函数式编程特别有用。为了让你建立更直观的感受，今天我为你准备了一个实战案例，设计一个简单的统计分析程序，用来分析三维视图中的对象。

好，话不多说，让我们从工程的基本介绍开始吧（课程完整代码，你可以从 [这里](#) 获取）。

在这个实战案例里，我们主要是展示 **Ranges** 的强大功能，而非数据本身的严谨性和正确性。因此，你可以重点关注处理数据的部分。

那么，**要分析统计的数据长什么样子呢？**我们假设一个三维模型包含多个视图，每个视图包含一定量的三维对象。某个三维对象中的三角面片就组成了逻辑上的三维对象。同时，三维模型会将视图分成高精度视图和低精度视图。

我造了一份简单的数据，一个三维模型的统计分析表是后面这样。

三维模型分析表						
视图类型：高精度						
序号	视图名称	三角面片数 (上限：50,000)	可用三角面片数	对象数	对象	对象面片数
1	Terminal	19,000	31,000	2	stair	8,000
					window	11,000
2	Side Road	8,765	41,235	2	curb	3,065
					arterial	5,700
3	Architecture	4,321	45,679	2	skeleton	3,320
					roof	1,001
视图类型：低精度						
序号	视图名称	三角面片数 (上限：50,000)	可用三角面片数	对象数	对象	对象面片数
1	Terminal	2,200	47,800	2	pool	201
					pinball arcade	1,999
2	Side Road	2,200	48,698	2	door	300
					wall	1,002
3	Architecture	310	49,690	2	table	300
					carpet	10



在这个案例中，我们先从系统接口获取数据，再基于这些数据生成表中的统计数据。系统接口的数据格式，都定义在了关键数据类型这个头文件（`include/Types.h`）中。代码是后面这样。

```
1 #pragma once
2
3 #include <stdint>
4 #include <string>
5 #include <chrono>
6 #include <vector>
7
8 namespace ca::types {
9     namespace chrono = std::chrono;
10
11     using Id = int32_t;
12     using ZonedTime = chrono::zoned_time<std::chrono::system_clock::duration, c
13
14     // 从API接口获取到的视图数据
15     struct ModelView {
16         // API接口中的三维对象数据
17         struct Object {
18             // 对象精度类型
19             enum class ResolutionType {
20                 High,
21                 Low
22             };
23
24             // 对象类型ID
25             Id objectTypeID;
26             // 对象名称
27             std::string name;
28             // 对象中各部件的面片数量（数组）
29             std::vector<int32_t> meshCounts;
30         };
31
32         // 视图ID
33         Id viewId = 0;
34         // 视图类型名称
35         std::string viewTypeName;
36         // 视图名称
37         std::string viewName;
38         // 创建时间
39         std::string createdAt;
40         // 三维对象列表
41         std::vector<Object> viewObjectList;
42
43         // 操作符重载，需要供算法使用
44         bool operator<(const ModelView& rhs) const {
45             if (createdAt < rhs.createdAt) {
46                 return true;
47             }
48
49             if (viewName > rhs.viewName) {
50                 return true;
51             }
52 }
```

```

53         return false;
54     }
55
56     bool operator>(const ModelView& rhs) const {
57         if (createdAt > rhs.createdAt) {
58             return true;
59         }
60
61         if (viewName < rhs.viewName) {
62             return true;
63         }
64
65         return false;
66     }
67
68     bool operator>=(const ModelView& rhs) const {
69         return *this == rhs || *this > rhs;
70     }
71
72     bool operator<=(const ModelView& rhs) const {
73         return *this == rhs || *this < rhs;
74     }
75
76     bool operator==(const ModelView& rhs) const {
77         return createdAt == rhs.createdAt && viewName == rhs.viewName;
78     }
79 };
80
81 // 统计后存储的对象数据
82 struct ModelObject {
83     // 视图序号
84     int32_t viewOrder = 0;
85     // 视图ID
86     Id viewId = 0;
87     // 视图类型名称
88     std::string viewTypeName;
89     // 视图名称
90     std::string viewName;
91     // 视图创建时间
92     ZonedTime createdAt;
93
94     // 对象类型ID
95     Id objectTypeID = 0;
96     // 对象名称
97     std::string objectName;
98     // 对象包含的三角面片数量
99     int32_t meshCount = 0;
100    // 对象在视图中的序号
101    int32_t viewObjectIndex = 0;
102    // 视图中剩余已用三角面片数量
103    int32_t viewUsedMeshCount = 0;
104    // 视图中可用三角面片数量上限

```

```

105     int32_t viewTotalMeshCount = 0;
106     // 视图中剩余可用三角面片数量
107     int32_t viewFreeMeshCount = 0;
108     // 视图中对象数量
109     size_t viewObjectCount = 0;
110
111     // 获取完整视图名称
112     std::string getCompleteViewName() const {
113         return viewTypeName + "/" + viewName;
114     }
115
116     // 获取对象Key
117     std::string getObjectKey() const {
118         return getObjectKey(objectTypeID, viewId);
119     }
120
121     // 根据objectTypeID和viewId获取对象Key
122     static std::string getObjectKey(Id objectTypeID, Id viewId) {
123         return std::to_string(objectTypeID) + "-" + std::to_string(viewId);
124     }
125 };
126
127 // 所有统计后的对象数据
128 struct ModelObjectTableData {
129     // 高精度对象
130     std::vector<ModelObject> highResolutionObjects;
131     // 低精度对象
132     std::vector<ModelObject> lowResolutionObjects;
133     // 三角面片数
134     int32_t meshCount;
135 };
136
137 // 被选择的某种精度的数据
138 struct ChoseModelObjectTableData {
139     // 当前对象数据
140     const ModelObjectTableData* objectTableData;
141     // 选择精度类型
142     ModelView::Object::ResolutionType resolutionType;
143
144     // 获取当前对象数据
145     const std::vector<ModelObject>& getCurrentModelObjects() const {
146         return resolutionType == ModelView::Object::ResolutionType::Low ?
147             objectTableData->lowResolutionObjects : objectTableData->highRe
148     }
149 };
150
151 // 从统计后的对象数据中选择某种精度的数据
152 inline ChoseModelObjectTableData chooseModelObjectTable(
153     const ModelObjectTableData& objectTableData,
154     ModelView::Object::ResolutionType resolutionType
155 ) {
156     return {

```

```
157         .objectTableData = &objectTableData,  
158         .resolutionType = resolutionType,  
159     };  
160  
161
```

在这段代码中，定义了几个类，具体说明我用表格的形式做了整理。

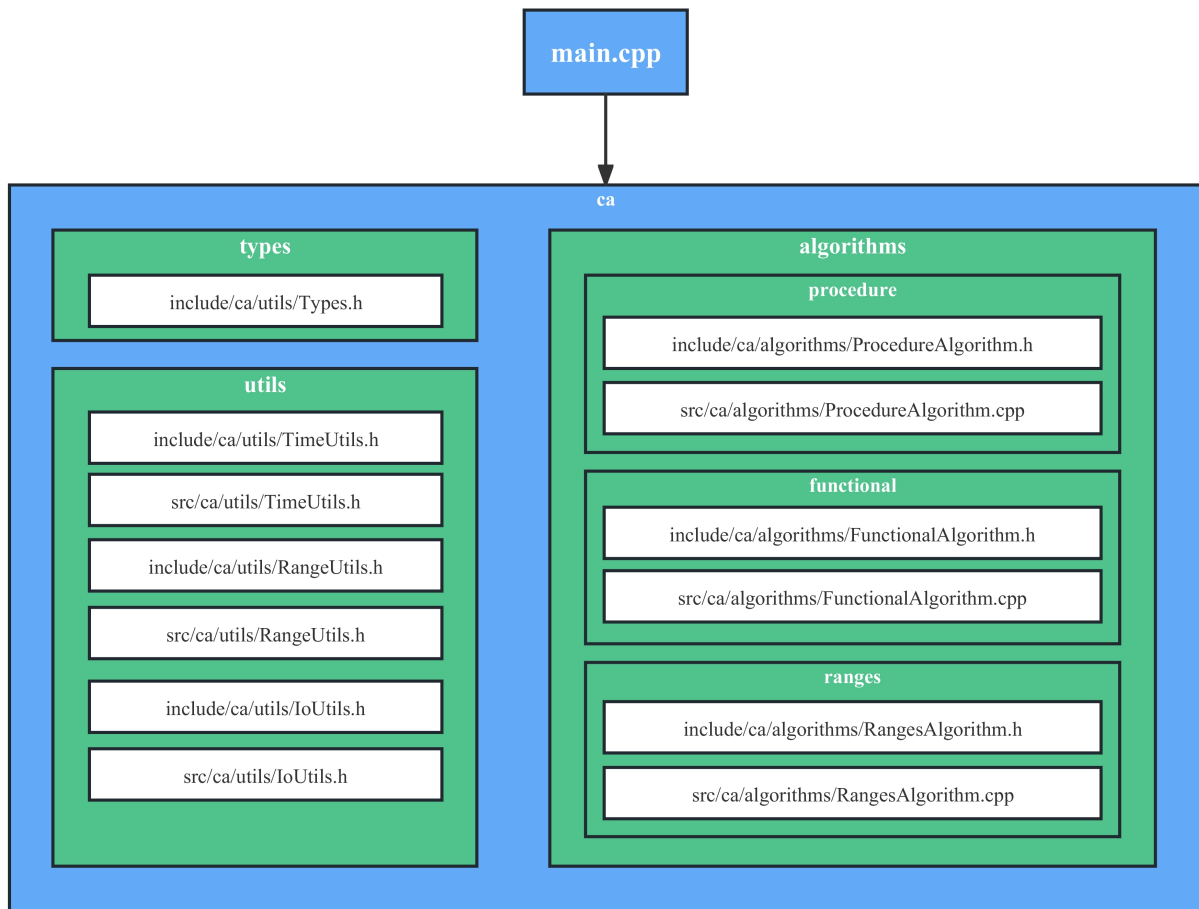
序号	类名	说明
1	ModelView	表示的是从系统接口获取的三维视图数据，它包含一个嵌套类 <b>Object</b> ，用来表示三维对象数据。 <b>Object</b> 中的枚举 <b>ResolutionType</b> 用于表示对象的精度类别。
2	ModelObject	表示的是统计分析后输出的对象数据。
3	ModelObjectTableData	用于存储完整的统计数据。
4	ChooseModelObjectTableData	表示选中的某种精度的数据，我们可以利用 <b>ChooseModelObjectTableData</b> 类提供的函数，来选中某种精度的数据。



至于类的数据成员，你可以对照文稿中的代码来具体了解。现在，我们有了关键数据类型定义，在这个基础上，我们就可以开始考虑如何设计程序的模块了。

这次，我们采用传统 **C++** 模块划分来实现整个工程。在学习的过程中，你可以思考一下，如何使用 **C++ Modules** 来改造代码的组织方式。

后面我画了示意图，方便你了解模块的结构和划分。



从图里可以看到，除了 `main.cpp` 作为程序入口以外，所有代码都放在 `ca`（`ca` 指的是 `correlations algorithm`，即统计算法）模块下。该模块包含了三个子模块。

- `types`：基础类型定义。
- `utls`：工具模块，包括时间工具库、输入输出工具库和 `Ranges` 工具库。
- `algorithms`：算法模块，包括 `procedure`、`functional`、`ranges` 单个子模块，分别对应过程化算法实现、函数式算法实现和基于 `Ranges` 的函数式算法实现。

因为我们关注的重点是跟 **Ranges** 库相关的逻辑，所以我们就沿着主模块（`main` 函数）和算法模块（`algorithms`）这条路线，利用 `Ranges` 来实现相关的统计分析功能。至于工具模块，只是提供了一些基本工具函数，不是我们学习的重点，你可以参考完整的 [工程代码](#)，了解其具体实现。

## 主模块

在主模块中，我定义了统计分析的接口，代码实现在 `src/main.cpp` 中。

```
1 #include "data.h"
2
3 #include "ca/IoUtils.h"
4 #include "ca/algorithms/ProcedureAlgorithm.h"
5 #include "ca/algorithms/FunctionalAlgorithm.h"
6 #include "ca/algorithms/RangesAlgorithm.h"
7
8 #include <iostream>
9 #include <set>
10
11 int main() {
12     using ca::types::ModelObjectTableData;
13     using ResolutionType = ca::types::ModelView::Object::ResolutionType;
14
15     // 获取对象信息
16     auto modelObjectsInfo = getModelObjectsInfo();
17
18     auto& highResolutionObjectSet = modelObjectsInfo.highResolutionObjectSet;
19     int32_t meshCount = modelObjectsInfo.meshCount;
20     auto& modelViews = modelObjectsInfo.modelViews;
21
22     // 过程化算法实现
23     auto procedureObjectTable = ca::algorithms::procedure::parseModelObjectTable(
24         modelViews,
25         highResolutionObjectSet,
26         meshCount
27     );
28     std::cout << ca::types::chooseModelObjectTable(procedureObjectTable, ResolutionType::PROCEDURE);
29     std::cout << ca::types::chooseModelObjectTable(procedureObjectTable, ResolutionType::FUNCTIONAL);
30
31     // 函数式算法实现（传统STL）
32     ModelObjectTableData functionalObjectTable = {
33         .meshCount = meshCount
34     };
35     ca::algorithms::functional::parseModelObjectTableData(
36         modelViews.begin(),
37         modelViews.end(),
38         highResolutionObjectSet,
39         meshCount,
40         std::back_inserter(functionalObjectTable.lowResolutionObjects),
41         std::back_inserter(functionalObjectTable.highResolutionObjects)
42     );
43     std::cout << ca::types::chooseModelObjectTable(functionalObjectTable, ResolutionType::PROCEDURE);
44     std::cout << ca::types::chooseModelObjectTable(functionalObjectTable, ResolutionType::FUNCTIONAL);
45
46     // 函数式算法实现（Ranges）
47     auto rangesObjectTable = ca::algorithms::ranges::parseModelObjectTableData(
48         std::cout << ca::types::chooseModelObjectTable(rangesObjectTable, ResolutionType::PROCEDURE);
49         std::cout << ca::types::chooseModelObjectTable(rangesObjectTable, ResolutionType::FUNCTIONAL);
50
51     return 0;
```



从实现代码可以看到，我们首先调用了 `getModelObjectsInfo` 函数，通过系统接口获取视图数据。视图数据是一个结构体，包括后面这些数据。

- `modelViews`: 视图数据列表。
- `highResolutionObjectSet`: 高精度的对象集合。
- `meshCount`: 每个视图可以使用的三角面片的数量上限。

接着，分别调用了以下 3 个不同的算法接口，使用不同范式实现了对相同数据的处理。

1. 过程式接口: `ca::algorithms::procedure::parseModelObjectTableData`。
2. 传统函数式接口: `ca::algorithms::functional::parseModelObjectTableData`。
3. 基于 `Ranges` 的函数式接口: `ca::algorithms::ranges::parseModelObjectTableData`。

传统函数式算法接口的输入参数比较特殊，由于传统的 STL 算法在数据的输入输出上，都使用迭代器而非容器，因此我们也模仿这种风格。最后，我们还需要调用 `chooseModelObjectTable` 来选择高精度或低精度对象，并将对象通过 `cout` 输出到标准输出中。

现在，我们来看一下刚才提到的 `getModelObjectsInfo` 函数，通过系统接口获取视图数据。该函数声明在 `include/data.h` 中。

 复制代码

```

1  #pragma once
2
3  #include "ca/Types.h"
4  #include <set>
5  #include <string>
6  #include <stdint>
7
8  struct ModelObjectsInfo {
9      // 视图数据列表
10     std::vector<ca::types::ModelView> modelViews;
11     // 高精度对象集合
12     std::set<std::string> highResolutionObjectSet;

```

```

13 // 3D模型可用三角面片总数
14 int32_t meshCount;
15 };
16
17 ModelObjectsInfo getModelObjectsInfo();

```

函数的实现代码在 `src/data.cpp` 中。

 复制代码

```

1 #include "data.h"
2
3 ModelObjectsInfo getModelObjectsInfo() {
4     return {
5         .modelViews = {
6             {
7                 .viewId = 1,
8                 .viewTypeName = "Building",
9                 .viewName = "Terminal",
10                .createdAt = "2020-09-01T08:00:00+0800",
11                .viewObjectList = {
12                    {
13                        .objectTypeID = 1,
14                        .name = "stair",
15                        .meshCounts = { 2000, 3000, 3000 },
16                    },
17                    {
18                        .objectTypeID = 2,
19                        .name = "window",
20                        .meshCounts = { 3000, 4000, 4000 },
21                    },
22                    {
23                        .objectTypeID = 3,
24                        .name = "pool",
25                        .meshCounts = { 100, 101 },
26                    },
27                    {
28                        .objectTypeID = 4,
29                        .name = "pinball arcade",
30                        .meshCounts = { 1000, 999 },
31                    },
32                },
33            },
34            {
35                .viewId = 2,
36                .viewTypeName = "Building",
37                .viewName = "Side Road",
38                .createdAt = "2020-09-01T08:00:00+0800",
39                .viewObjectList = {
40                    {

```

```

41         .objectTypeID = 5,
42         .name = "curb",
43         .meshCounts = { 1000, 1000, 1000, 65 },
44     },
45     {
46         .objectTypeID = 6,
47         .name = "arterial",
48         .meshCounts = { 1000, 2000, 2700 },
49     },
50     {
51         .objectTypeID = 7,
52         .name = "door",
53         .meshCounts = { 60, 40, 200 },
54     },
55     {
56         .objectTypeID = 8,
57         .name = "wall",
58         .meshCounts = { 200, 500, 302 },
59     },
60 },
61 },
62 {
63     .viewId = 3,
64     .viewTypeName = "Building",
65     .viewName = "Architecture",
66     .createdAt = "2020-09-01T08:00:00+0800",
67     .viewObjectList = {
68         {
69             .objectTypeID = 9,
70             .name = "skeleton",
71             .meshCounts = { 1000, 1000, 1000, 320 },
72         },
73         {
74             .objectTypeID = 10,
75             .name = "roof",
76             .meshCounts = { 500, 501 },
77         },
78         {
79             .objectTypeID = 11,
80             .name = "table",
81             .meshCounts = { 50, 50, 100, 100 },
82         },
83         {
84             .objectTypeID = 12,
85             .name = "carpet",
86             .meshCounts = { 2, 2, 2, 1, 3 },
87         },
88     },
89 },
90 },
91 .highResolutionObjectSet = {
92     "1-1",

```

```

93         "2-1",
94         "5-2",
95         "6-2",
96         "9-3",
97         "10-3",
98     },
99     .meshCount = 50000,
100 };
101

```

在这里，我选择直接返回了硬编码的数据，这种方式充分利用了 **C++20** 的初始化表达式扩展，让初始化代码变得和直接编写 **JSON** 一样简单。数据本身就是需求中表格的详细数据。

我在这里简化了数据初始化的过程，你也可以考虑使用外部数据作为程序的输入。

接下来，我们重点看看算法模块，其中包含了关键的 **Ranges** 用法，所以该模块是整个工程实战的重头戏。

## 算法模块

算法模块分别包含传统过程式实现，函数式实现（基于传统 **STL**）和采用 **Ranges** 的函数式实现的分析算法。

为什么要展示不同方案呢？这是为了让你有个对比，也能突出使用 **Ranges** 版本的优势。

## 过程化实现

先来看过程化算法，我们先编写头文件 `include/ca/algorithms/ProcedureAlgorithm.h`。

 复制代码

```

1  #pragma once
2  #include "ca/Types.h"
3  #include <set>
4
5  namespace ca::algorithms::procedure {
6      ca::types::ModelObjectTableData parseModelObjectTableData(
7          // 视图数据
8          std::vector<ca::types::ModelView> modelViews,
9          // 高精度对象集合
10         const std::set<std::string>& highResolutionObjectSet,
11         // 三角面片数上限
12         int32_t meshCount
13     );

```

这里声明了几个简单的数据成员变量，对照代码和注释，你就能知道它们的用途。

接下来，我们来看具体的算法实现。

[复制代码](#)

```

1  #include "ca/algorithms/ProcedureAlgorithm.h"
2  #include "ca/TimeUtils.h"
3  #include <iostream>
4  #include <map>
5  #include <stdint>
6  #include <iostream>
7  #include <algorithm>
8  #include <numeric>
9
10 namespace ca::algorithms::procedure {
11     using ca::types::ModelObjectTableData;
12     using ca::types::ModelView;
13     using ca::types::ModelObject;
14     using ca::utils::timePointFromString;
15
16     ModelObjectTableData parseModelObjectTableData(
17         std::vector<ModelView> modelViews,
18         const std::set<std::string>& highResolutionObjectSet,
19         int32_t meshCount
20     ) {
21         std::cout << "[PROCEDURE] Parse model objects table data" << std::endl;
22
23         // 对视图数组进行排序
24         std::sort(modelViews.begin(), modelViews.end());
25
26         // 低精度对象
27         std::vector<ModelObject> highResolutionObjects;
28         // 高精度对象
29         std::vector<ModelObject> lowResolutionObjects;
30
31         int32_t viewOrder = 0;
32         // 遍历视图
33         for (const auto& modelView : modelViews) {
34             auto viewId = modelView.viewId;
35             auto& viewTypeName = modelView.viewTypeName;
36             auto& viewName = modelView.viewName;
37             auto& viewObjectList = modelView.viewObjectList;
38             auto& createdAt = modelView.createdAt;
39             viewOrder++;
40
41             // 本视图的低精度对象
42             std::vector<ModelObject> lowResolutionModelObjects;

```

```

43 // 本视图的低精度对象面片数总和
44 int32_t lowResolutionMeshCounts = 0;
45 // 本视图的高精度对象
46 std::vector<ModelObject> highResolutionModelObjects;
47 // 本视图的高精度对象面片数总和
48 int32_t doubleResolutionMeshCounts = 0;
49
50 // 遍历视图的对象信息，将对象的数据按高低精度添加到各自的数组并更改统计数据
51 for (const auto& viewObject : modelView.viewObjectList) {
52     // 遍历meshCounts计算对象的总面片数
53     int32_t objectMeshCount = 0;
54     for (int meshCount : viewObject.meshCounts) {
55         objectMeshCount += meshCount;
56     }
57
58     ModelObject modelObject = {
59         .viewOrder = viewOrder,
60         .viewId = viewId,
61         .viewTypeName = viewTypeName,
62         .viewName = viewName,
63         .createdAt = timePointFromString(createdAt),
64         .objectTypeID = viewObject.objectTypeID,
65         .objectName = viewObject.name,
66         .meshCount = objectMeshCount
67     };
68
69     // 确定对象是否是高精度对象
70     if (highResolutionObjectSet.find(modelObject.getObjectKey()) ==
71         // 低精度对象
72         lowResolutionMeshCounts += objectMeshCount;
73         lowResolutionModelObjects.push_back(modelObject);
74     }
75     else {
76         // 高精度对象
77         doubleResolutionMeshCounts += objectMeshCount;
78         highResolutionModelObjects.push_back(modelObject);
79     }
80 }
81
82 // 计算低精度视图统计信息，原地修改低精度对象信息
83 for (auto& modelObject : lowResolutionModelObjects) {
84     modelObject.viewUsedMeshCount = lowResolutionMeshCounts;
85     modelObject.viewTotalMeshCount = meshCount;
86     modelObject.viewFreeMeshCount = meshCount - lowResolutionMeshCounts;
87     modelObject.viewObjectCount = lowResolutionModelObjects.size();
88
89     lowResolutionObjects.push_back(modelObject);
90 }
91
92 // 计算高精度视图统计信息，原地修改高精度对象信息
93 for (auto& modelObject : highResolutionModelObjects) {
94     modelObject.viewUsedMeshCount = doubleResolutionMeshCounts;

```

```

95         modelObject.viewTotalMeshCount = meshCount;
96         modelObject.viewFreeMeshCount = meshCount - doubleResolutionMes
97         modelObject.viewObjectCount = highResolutionModelObjects.size()
98
99         highResolutionObjects.push_back(modelObject);
100     }
101 }
102
103 // 返回数据
104 return ModelObjectTableData{
105     .highResolutionObjects = highResolutionObjects,
106     .lowResolutionObjects = lowResolutionObjects,
107     .meshCount = meshCount,
108 };
109 }
110 }

```

这里，我们按照标准的过程化编程思路进行编程，这是一种非常平凡的方法——按部就班地处理数据。但是，有必要提一下这段代码的两个特性。

1. 除了 `sort` 函数以外，统计分析基本都通过 `for` 循环完成。
2. 在数据处理时，存在大量的原地修改。

这几特性在过程化编程中很常见。不过，这种原地修改数据的行为，不利于数据的处理和计算。

虽然原地修改数据可能在简单的程序中很实用，但在复杂的程序中，理论上每个变量应该都只有唯一用途，这么做会让一个变量在程序的不同位置上具备不同的用途。在并行程序中，这种行为会引发数据竞争，反而可能降低并行程序处理性能。所以，在很多时候，对于一个复杂数据处理程序，我们应该避免原地修改数据。

## 传统 STL 函数式实现

那么，如果用函数式编程的方法实现，会有什么变化呢？接下来，我们就看看基于传统 STL 的、函数式算法的实现版本。

先看头文件的定义，具体代码放在了 `include/ca/algorithms/FunctionalAlgorithm.h` 下面。

```

2 #pragma once
3 #include "ca/Types.h"
4 #include <set>
5 #include <vector>
6
7 namespace ca::algorithms::functional {
8     using ModelObjectOutputIterator = std::back_inserter<std::vector<ca:
9     void parseModelObjectTableData(
10         // 视图begin迭代器
11         std::vector<ca::types::ModelView>::const_iterator modelViewsBegin,
12         // 视图end迭代器
13         std::vector<ca::types::ModelView>::const_iterator modelViewsEnd,
14         // 高精度对象集合
15         const std::set<std::string>& highResolutionObjectSet,
16         // 三角面片数上限
17         int32_t meshCount,
18         // 低精度对象输出迭代器，用于输出插入低精度对象
19         ModelObjectOutputIterator lowResolutionObjectOutputIterator,
20         // 高精度对象输出迭代器，用于输出插入高精度对象
21         ModelObjectOutputIterator highResolutionObjectOutputIterator
22     );
23 }

```

接下来，是这种方法的具体实现，代码在 `src/ca/algorithms/FunctionalAlgorithm.cpp` 中。建议你大致浏览一下代码，想想和前面过程化实现有什么不同，然后我们再进行进一步探讨。

 复制代码

```

1 #include "ca/algorithms/FunctionalAlgorithm.h"
2 #include "ca/TimeUtils.h"
3 #include <iostream>
4 #include <map>
5 #include <tuple>
6 #include <cstdint>
7 #include <iostream>
8 #include <algorithm>
9 #include <numeric>
10
11 using ca::types::ModelObjectTableData;
12 using ca::types::ModelView;
13 using ca::types::ModelObject;
14 using ca::utils::timePointFromString;
15
16 namespace ca::algorithms::functional {
17     static void extractHighOrLowResolutionObjects(
18         // 排序后的视图数组
19         const std::vector<ca::types::ModelView>& modelViews,
20         // 高精度对象集合
21         const std::set<std::string>& highResolutionObjectSet,
22         // 三角面片数上限

```



```

23     int32_t meshCount,
24     // 是否要提取高精度数据, 参数为true时提取高精度数据, 参数为false时提取低精度数据
25     bool isHigh,
26     // 对象输出迭代器, 用于输出插入对象数据 (高精度或双精度)
27     ModelObjectOutputIterator outputIterator
28 );
29
30 void parseModelObjectTableData(
31     std::vector<ModelView>::const_iterator modelViewsBegin,
32     std::vector<ModelView>::const_iterator modelViewsEnd,
33     const std::set<std::string>& highResolutionObjectSet,
34     int32_t meshCount,
35     ModelObjectOutputIterator lowResolutionObjectOutputIterator,
36     ModelObjectOutputIterator highResolutionObjectOutputIterator
37 ) {
38     std::cout << "[FUNCTIONAL] Parse model object data" << std::endl;
39
40     // 对视图数组进行排序
41     std::vector<ModelView> sortedModelViews;
42     std::copy(modelViewsBegin, modelViewsEnd, std::back_inserter(sortedModelViews));
43     std::sort(sortedModelViews.begin(), sortedModelViews.end());
44
45     // 提取低精度对象数据, 通过lowResolutionObjectOutputIterator输出数据
46     extractHighOrLowResolutionObjects(
47         sortedModelViews,
48         highResolutionObjectSet,
49         meshCount,
50         false,
51         lowResolutionObjectOutputIterator
52     );
53
54     // 提取高精度对象数据, 通过highResolutionObjectOutputIterator输出数据
55     extractHighOrLowResolutionObjects(
56         sortedModelViews,
57         highResolutionObjectSet,
58         meshCount,
59         true,
60         highResolutionObjectOutputIterator
61     );
62 }
63
64 static void extractHighOrLowResolutionObjects(
65     const std::vector<ca::types::ModelView>& modelViews,
66     const std::set<std::string>& highResolutionObjectSet,
67     int32_t meshCount,
68     bool isHigh,
69     ModelObjectOutputIterator outputIterator
70 ) {
71     // 生成对象数据 (高精度或双精度)
72     // 将模型三维对象数组转换成一个新数组, 数组元素是每个视图的对象数组 (返回的是二维数组)
73     std::vector<std::vector<ModelObject>> objectsOfViews;
74     auto modelViewsData = modelViews.data();

```

```

75     std::transform(
76         modelViews.begin(), modelViews.end(), std::back_inserter(objectsOfV
77         [modelViewsData, &highResolutionObjectSet, meshCount, isHigh](const
78             // 通过视图指针地址计算视图序号
79             int32_t viewOrder = static_cast<int32_t>(&modelView - modelView
80
81             // 筛选满足要求的对象（高精度或低精度）
82             const std::vector<ModelView::Object>& viewObjectList = modelVie
83             std::vector<ModelView::Object> filteredViewObjectList;
84             std::copy_if(
85                 viewObjectList.begin(), viewObjectList.end(),
86                 std::back_inserter(filteredViewObjectList),
87                 [&modelView, &highResolutionObjectSet, isHigh](const ModelV
88                     auto viewId = modelView.viewId;
89                     auto objectTypeID = viewObject.objectTypeID;
90                     // 通过ModelObject::getObjectKey获取对象的key（格式为object
91                     auto objectKey = ModelObject::getObjectKey(objectTypeID
92
93                     // 如果高精度对象集合中存在该对象返回true，可以筛选出高精度对象
94                     // 如果不存在则返回false，可以筛选出低精度对象
95                     return highResolutionObjectSet.contains(objectKey) == i
96             }
97         );
98
99         // 计算各对象总面片数，生成对象数组
100         std::vector<ModelObject> highResolutionObjects;
101         std::transform(
102             filteredViewObjectList.begin(), filteredViewObjectList.end(
103             [&modelView, &highResolutionObjectSet, &filteredViewObjectL
104                 auto viewId = modelView.viewId;
105                 auto& viewTypeName = modelView.viewTypeName;
106                 auto& viewName = modelView.viewName;
107                 auto& viewObjectList = modelView.viewObjectList;
108                 auto& createdAt = modelView.createdAt;
109                 auto objectTypeID = viewObject.objectTypeID;
110
111                 // 求对象总面片数
112                 const auto& meshCounts = viewObject.meshCounts;
113                 auto objectMeshCount = std::accumulate(meshCounts.begin
114
115                 return ModelObject{
116                     .viewOrder = viewOrder,
117                     .viewId = viewId,
118                     .viewTypeName = viewTypeName,
119                     .viewName = viewName,
120                     .createdAt = timePointFromString(createdAt),
121                     .objectTypeID = viewObject.objectTypeID,
122                     .objectName = viewObject.name,
123                     .meshCount = objectMeshCount
124                 };
125             }
126         );

```

```

127 // 求视图已占用面片数
128 auto viewUsedMeshCount = std::accumulate(
129     highResolutionObjects.begin(), highResolutionObjects.end(),
130     [](int32_t prev, const auto& modelObject) {
131         return prev + modelObject.meshCount;
132     }
133 );
134
135 // 生成完整的对象数据
136 std::vector<ModelObject> resultModelObjects;
137 auto viewObjectCount = highResolutionObjects.size();
138 std::transform(
139     highResolutionObjects.begin(), highResolutionObjects.end(),
140     [viewUsedMeshCount, meshCount, viewObjectCount](const auto&
141 // 返回全新的ModelObject对象，不原地修改数据
142     return ModelObject{
143         .viewOrder = incomingModelObject.viewOrder,
144         .viewId = incomingModelObject.viewId,
145         .viewTypeName = incomingModelObject.viewTypeName,
146         .viewName = incomingModelObject.viewName,
147         .createdAt = incomingModelObject.createdAt,
148         .objectTypeID = incomingModelObject.objectTypeID,
149         .objectName = incomingModelObject.objectName,
150         .meshCount = incomingModelObject.meshCount,
151         .viewUsedMeshCount = viewUsedMeshCount,
152         .viewTotalMeshCount = meshCount,
153         .viewFreeMeshCount = meshCount - viewUsedMeshCount,
154         .viewObjectCount = viewObjectCount,
155     };
156     }
157 );
158
159 // 返回完整的对象数据
160 return resultModelObjects;
161 }
162 );
163
164 // 展平二维数组
165 std::for_each(
166     objectsOfViews.begin(), objectsOfViews.end(),
167     [&outputIterator](const auto& modelObjects) {
168         outputIterator = std::copy(modelObjects.begin(), modelObjects.e
169     }
170 );
171 }
172 }
173 }

```

174

看完这段代码实现，有没有什么感想或发现？

没错，代码更复杂了？！

这比纯粹的过程式代码还要长，函数 `parseModelObjectTableData` 的计算过程，总共分为三步。

第一步，采用 `sort` 算法函数排序视图数组。第二步，调用 `extractHighOrLowResolutionObjects` 提取低精度对象数据，通过 `lowResolutionObjectOutputIterator` 输出数据。第三步，还是调用上一步的函数，通过 `highResolutionObjectOutputIterator` 输出数据。

在这个函数中，我们调用了 `transform` 算法，将对象的数组转换成一个新数组。同时，为了把二维数组展平成一维数组，我们通过 `for_each` 将二维数组中的子数组的数据，都拷贝到了最终的输出迭代器中。

现在，我们分析一下这种传统函数式编程方案有哪些特点。

**首先，它基于 STL 实现，所有的任务都转换成了函数。**

包括将循环都转换成了 `transform`（映射）、`copy_if`（筛选）和 `accumulate`（聚合计算）。其中，`transform` 算法将一个数组中的元素映射到另一个数组中，类似于 Python 和 JavaScript 中的 `map` 函数。

在代码的第 85 行，`copy_if` 算法遍历了迭代器中的数据，将符合 `copy_if` 中条件函数的元素写入到输出迭代器中，本质类似于 Python 和 JavaScript 中 `filter` 的效果。

**其次，数据处理过程中尽量避免出现副作用。**

比如说，在原地排序前，我们先复制数据。在 `transform` 和 `copy_if` 时，都创建了一个新的空数组，然后通过 `back_inserter` 获取插入迭代器，然后将输入插入到新数组中。同时，避免在 `transform` 的过程中修改输入参数（输入数组的元素）。

**最后，在复杂的计算过程中，将类似的任务提取出来，然后分段处理数组。**

比如说，在第 46 行和 55 行，调用了两次 `extractHighOrLowResolutionObjects` 函数，也就是需要遍历两次视图，所以相对于过程化版本需要多一次循环。但是，从时间复杂度上看，没有

本质区别。

接下来，我们思考这样一个问题——**如果比较传统 STL 函数式编程和过程实现的方案，哪种方案编程效率更高？**

从表面上看，这种编码方案比过程式方案复杂。

但是，考虑到并行化处理问题，比如将高精度、低精度的计算任务分别放在两个线程上执行。那么，由于这种函数式处理方式不会产生对数据的副作用，在处理大规模数据时，无需担心数据竞争和加锁的问题。因此，这种编程方案，执行效率反而更高。

不过，这段代码看起来还是过于冗长了，而且编码效率低下。所以，我们还是得靠 **Ranges** 库来改善函数式编程的开发效率。

## 基于 Ranges 的函数式实现

现在，让我们聚焦在 **Ranges** 库上，看看基于 **Ranges** 的函数式算法是怎么实现的。

头文件定义在 `include/ca/algorithms/RangesAlgorithm.h` 中。

 复制代码

```
1 #pragma once
2 #include "ca/Types.h"
3 #include <set>
4 namespace ca::algorithms::ranges {
5     ca::types::ModelObjectTableData parseModelObjectTableData(
6         // 视图数据
7         std::vector<ca::types::ModelView> modelViews,
8         // 高精度对象集合
9         const std::set<std::string>& highResolutionObjectSet,
10        // 三角面片数上限
11        int32_t meshCount
12    );
13 }
```

对头文件对应的具体实现是这样。

 复制代码

```
1 #include "ca/algorithms/RangesAlgorithm.h"
```

```

2 #include "ca/TimeUtils.h"
3 #include "ca/RangeUtils.h"
4 #include <iostream>
5 #include <ranges>
6 #include <algorithm>
7 #include <numeric>
8
9 using ca::types::ModelObjectTableData;
10 using ca::types::ModelView;
11 using ca::types::ModelObject;
12 using ca::utils::timePointFromString;
13
14 namespace ca::algorithms::ranges {
15     namespace ranges = std::ranges;
16     namespace views = std::views;
17
18     using ca::utils::sizeOfRange;
19     using ca::utils::views::to;
20
21     // 提取对象
22     static std::vector<ModelObject> extractHighOrLowResolutionObjects(
23         // 排序后的视图数组
24         const std::vector<ca::types::ModelView>& modelViews,
25         // 高精度对象集合
26         const std::set<std::string>& highResolutionObjectSet,
27         // 三角面片数上限
28         int32_t meshCount,
29         // 是否要提取高精度数据，参数为true时提取高精度数据，参数为false时提取低精度数据
30         bool isDouble
31     );
32
33     ca::types::ModelObjectTableData parseModelObjectTableData(
34         std::vector<ca::types::ModelView> modelViews,
35         const std::set<std::string>& highResolutionObjectSet,
36         int32_t meshCount
37     ) {
38         std::cout << "[RANGES] Parse model objects table data" << std::endl;
39
40         // 对视图数组进行排序
41         ranges::sort(modelViews);
42
43         return ca::types::ModelObjectTableData{
44             // 提取低精度对象数据
45             .highResolutionObjects = extractHighOrLowResolutionObjects(
46                 modelViews,
47                 highResolutionObjectSet,
48                 meshCount,
49                 true
50             ),
51             // 提取高精度对象数据
52             .lowResolutionObjects = extractHighOrLowResolutionObjects(
53                 modelViews,

```

```

54         highResolutionObjectSet,
55         meshCount,
56         false
57     ),
58     .meshCount = meshCount,
59 };
60 }
61
62 static std::vector<ModelObject> extractHighOrLowResolutionObjects(
63     const std::vector<ca::types::ModelView>& modelViews,
64     const std::set<std::string>& highResolutionObjectSet,
65     int32_t meshCount,
66     bool isHigh
67 ) {
68     auto modelViewsData = modelViews.data();
69
70     // 生成对象数据（高精度或双精度）
71     // 将模型三维对象数组转换成一个新数组，数组元素是每个视图的对象数组（返回的是二维数组）
72     return modelViews |
73         views::transform([modelViewsData, &highResolutionObjectSet, meshCou
74         // 通过视图指针地址计算视图序号
75         int32_t viewOrder = static_cast<int32_t>(&modelView - modelView
76         const std::vector<ModelView::Object>& viewObjectList = modelVie
77
78         auto filteredModelObjects = viewObjectList |
79         // 筛选满足要求的对象（高精度或低精度）
80         views::filter([&modelView, &highResolutionObjectSet, isHigh
81             auto viewId = modelView.viewId;
82             auto objectTypeID = viewObject.objectTypeID;
83             auto objectKey = ModelObject::getObjectKey(objectTypeID
84
85             return highResolutionObjectSet.contains(objectKey) == i
86         }) |
87         // 计算各对象总面片数，生成对象数组
88         views::transform([&modelView, &highResolutionObjectSet, vie
89             auto viewId = modelView.viewId;
90             auto& viewTypeName = modelView.viewTypeName;
91             auto& viewName = modelView.viewName;
92             auto& viewObjectList = modelView.viewObjectList;
93             auto& createdAt = modelView.createdAt;
94             auto objectTypeID = viewObject.objectTypeID;
95
96             const auto& meshCounts = viewObject.meshCounts;
97             auto objectMeshCount = std::accumulate(meshCounts.begin
98
99             return ModelObject{
100                 .viewOrder = viewOrder,
101                 .viewId = viewId,
102                 .viewTypeName = viewTypeName,
103                 .viewName = viewName,
104                 .createdAt = timePointFromString(createdAt),
105                 .objectTypeID = viewObject.objectTypeID,

```

```

106         .objectName = viewObject.name,
107         .meshCount = objectMeshCount
108     };
109 });
110
111 // 计算视图已占用面片数
112 auto viewUsedMeshCount = std::accumulate(
113     filteredModelObjects.begin(),
114     filteredModelObjects.end(),
115     0,
116     [](int32_t prev, const auto& modelObject) { return prev
117 });
118
119 // 计算视图中的对象数量
120 size_t viewObjectCount = sizeofRange(filteredModelObjects);
121
122 // 生成包含统计信息的对象数据
123 return filteredModelObjects |
124     views::transform(
125         [viewUsedMeshCount, meshCount, viewObjectCount](const
126             return ModelObject{
127                 .viewOrder = incomingModelObject.viewOrder,
128                 .viewId = incomingModelObject.viewId,
129                 .viewTypeName = incomingModelObject.viewType
130                 .viewName = incomingModelObject.viewName,
131                 .createdAt = incomingModelObject.createdAt,
132                 .objectTypeID = incomingModelObject.objectT
133                 .objectName = incomingModelObject.objectNar
134                 .meshCount = incomingModelObject.meshCount,
135                 .viewUsedMeshCount = viewUsedMeshCount,
136                 .viewTotalMeshCount = meshCount,
137                 .viewFreeMeshCount = meshCount - viewUsedMe
138                 .viewObjectCount = viewObjectCount,
139             };
140         }
141     ) |
142     to<std::vector<ModelObject>>>();
143 }) |
144 to<std::vector<std::vector<ModelObject>>>>() |
145 views::join |
146 to<std::vector<ModelObject>>>();
147
148

```

你应该感觉到了，这段代码的基本结构和传统的 STL 版本一样，但是明显简洁不少——最后的代码形式更像函数式编程语言。

这都得益于 Ranges 带来的以下两个关键变化。



**第一，采用视图替换原本的算法。**比如说，用 `views::transform` 替换 `std::transform`、`views::filter` 替换 `std::copy_if`，并使用 `views::join` 替代 `std::for_each`，让二维数组展平为一维数组。这样一来，我们就可以直接通过 `range` 对象作为输入参数，而不再需要手动获取迭代器，代码更加清晰明了。

**第二，采用视图管道替换原本的过程化衔接。**比如说，筛选视图中的高精度 / 低精度对象（`ModelView::Object`）以及生成对象（`ModelObject`）的过程，被改写成了视图管道的连接。二维数组的生成与展平过程也通过 `views::join` 和视图管道实现。

我们通过 `Ranges` 大幅减少了临时变量的定义。否则，在复杂的函数式编码过程中，给这么多临时变量起名，几乎是一个不可能完成的任务。

不过受限于 C++20（不考虑 C++20 的后续演进标准）提供的支持，现在的代码还有几点不足。

第一，算法函数 `accumulate` 其实是在头文件 `<numeric>` 中，因此并没有提供针对视图的实现，这里我们还是用了迭代器作为输入，希望 C++ 之后能提供其 `range` 版本。

第二，视图 `views::join` 可以将诸如 `T<T<E>>` 这种类型转换为 `T<E>`，也就是将子数组的元素都“join”到一起。但是，这个函数无法直接连接嵌套视图，因此，我们需要一个视图适配器，将视图转换为具体容器类型，然后通过 `join` 进行转换。

第三，将视图转换为容器需要通过模板函数 `to<>` 实现。但这个视图是 C++23 标准中的，C++20 中并没有提供该视图，也就意味着正常情况这段代码无法在 C++20 中通过编译！

标准不支持，那我们自己实现不就可以了，为此我们在 `Ranges` 工具库中实现了模板函数 `to<>`，具体定义在 `Ranges` 工具库头文件 `include/ca/RangeUtils.h` 中。

 复制代码

```
1 #pragma once
2
3 #include <ranges>
4 #include <algorithm>
5 #include <numeric>
6
7 namespace ca::utils {
8     template <std::ranges::range T>
9     size_t sizeOfRange(
```

```

10     T& range
11     ) {
12         return static_cast<size_t>(std::accumulate(
13             range.begin(),
14             range.end(),
15             0,
16             [](int32_t prev, const auto& value) { return prev + 1; }
17         ));
18     }
19
20     namespace views {
21         template <class Container>
22         struct ToFn {
23         };
24
25         template <class Container>
26         ToFn<Container> to() {
27             return ToFn<Container>();
28         }
29
30         template <class Container, std::ranges::viewable_range Range>
31         Container operator | (Range range, const ToFn<Container>& fn) {
32             Container container{};
33             std::ranges::copy(range, std::back_inserter(container));
34
35             return container;
36         }
37     }
38 }

```

由于 C++20 并不支持自定义视图类型的 `range` 适配器闭包对象，这里我们再了解一下实现视图管道的变通方案。

首先回顾一下视图管道。视图管道是一种语法糖：假定 `C` 是一个视图适配器闭包对象，`R` 是一个 `Range` 对象，编译器可以自动将以下代码中的第 1 行，转换成第 2 行的形式。

```

1 R | C
2 C(R)

```

 复制代码

这里的 `|` 就是视图管道的操作符。只不过这种语法糖要求视图适配器闭包对象，要按照特定要求实现 `operator()` 操作符重载。但在 C++20 中，自定义的视图类型是无法通过 `operator()` 操作符重载获得视图管道支持的。

虽然编译器无法给予语法糖支持，但开发者可以通过 C++ 标准的操作符重载实现相同的效果。假设 **R** 是 **Range** 对象的类型，**P** 是自定义适配器闭包对象的类型，如果我们实现下列代码中的函数，就可以实现与视图管道相同的效果了。

 复制代码

```
1 auto operator | (R r, P p) {  
2     return p(r);  
3 }
```

其实，这么做就是实现了 **operator |** 的操作符重载，可以针对特定类型的 **R** 与 **P**，将 **R | P** 这种表达式转换为 **P(R)**。

如果你希望支持所有的 **Range** 视图，那么可以使用 **concept** 将该函数定义成后面的形式。

 复制代码

```
1 template <std::ranges::viewable_range R>  
2 auto operator | (R r, P p) {  
3     return p(r);  
4 }
```

看，这不就实现了跟 **range** 适配器闭包对象一样的效果了嘛。

只不过，这种方式要求我们定义一个类型 **P**，作为操作符重载的“占位符”。因此，在这里我们又总结出一个新的概念——将这种不是视图，但为了模拟 **range** 适配器闭包对象，而创建的类似于 **range** 适配器闭包的类型，称之为“仿 **range** 适配器闭包”。

这是受到了 C++ 的仿函数的启发，也是通过 **()** 操作符重载来模仿函数行为。虽然需要一些技巧，不过在编译器支持 C++23 之前，在现阶段我们还需要这种变通方式，希望你能掌握这种编程技巧。

## 总结

这一讲，我们结合一个简单的统计分析程序案例。为了方便你对比，在算法模块我还给出了过程化实现方案和传统 **STL** 函数式的实现方案。

通过这一讲的学习，相信你已经直观地感受到了 **Ranges** 的强大功能。**Ranges** 可以大幅提高 **C++** 中函数式编程的代码可读性，降低代码复杂度，提高函数式编程效率。我们可以通过 **Ranges** 库中的视图来简化数据处理过程，并利用管道来替换原有的过程化衔接。

在处理大规模数据的时候，利用 **Ranges** 库我们几乎可以避免声明临时变量。现在，**C++** 中的函数式编程变得更加现代，也跟其他支持函数式编程语言一样，实现了类似的编程范式。其实 **Ranges** 还有更多使用场景，期待你在日常开发中多多探索。

## 课后思考

我们在一讲中使用 **Ranges** 库实现了对数据集合的处理。那么，你能否结合 **Ranges** 库实现以下功能？

1. 输入一组数据，数据由一系列的字符串组成，每个字符串是一个句子。比如：


- “C++ 20 is much more powerful than ever before ”
- “I am learning C++ 20, C++ 23 and C++ 26 ”


2. 剔除每个字符串中的数字，返回新的数组。

3. 处理的过程中，结合 **C++ Coroutines** 来实现异步处理（你可以尝试复用前面课程的代码）。

欢迎把你的代码贴出来，与大家一起分享。我们一同交流。下一讲见！

分享给需要的人，Ta购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言

[写留言](#)

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。