

03 | Modules（下）：实战写个多模块图像处理服务

2023-01-20 卢誉声 来自北京



天下无鱼

<https://shikey.com/>

《现代C++20实战高手课》

[课程介绍 >](#)



讲述：卢誉声

时长 17:07 大小 15.64M



你好，我是卢誉声。

通过前面的学习，我们掌握了模块的基本概念。这节课，我们会一起学习，怎样使用 C++ Modules 来组织实际的项目代码。相信你在动手实战后，就能进一步理解应该如何使用 C++ Modules 和 namespace 来解决现实问题。

掌握了基本概念和使用要点之后，我们也会站在语言设计者的角度，整体讨论一下 C++ Modules 能解决什么问题，不能解决什么问题。

好，话不多说，我们马上进入今天的学习。课程配套代码，点击 [🔗 这里](#) 获取。

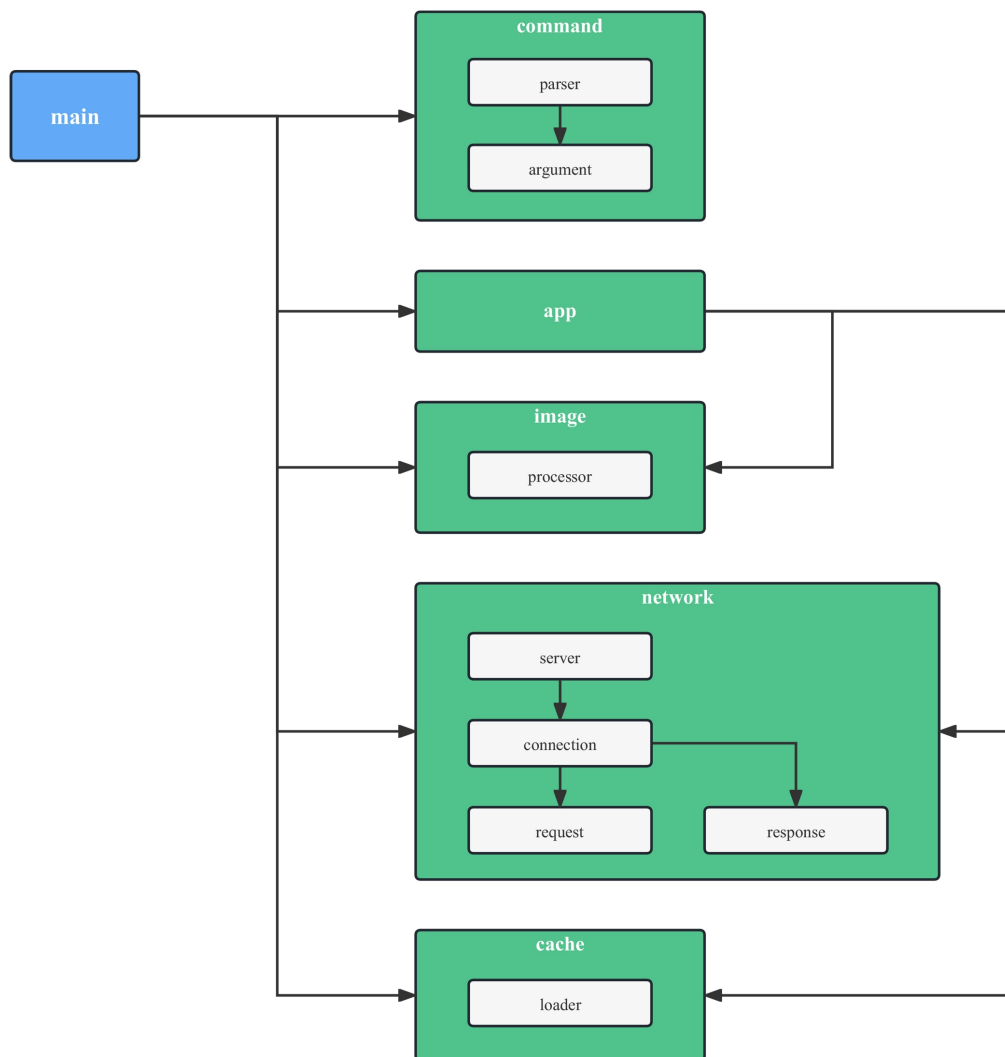
面向图像的对象存储系统

要写的实例是一个常见的面向图像的对象存储系统，核心功能是将图片存储在本地空间，用户通过 HTTP 请求获取相应的图片，而这个系统的特点是用户除了可以获取原始图片，还可以通过参数获取经过处理的图片，比如图像缩放、图像压缩等。



想实现这样的功能，需要哪些模块呢？

我们画一张系统的模块架构图，可以清晰地看到系统模块以及模块内部分区的依赖关系。



首先我们需要创建项目，项目包括 5 个子模块，分别是 `app`、`cache`、`command`、`image`、`network`，其中 `app` 是业务应用模块，`cache` 是本地缓存模块，`command` 是命令行解析模块，`image` 是图像处理模块，`network` 是网络服务模块，每个模块分别创建对应的目录存储模块内的源代码。

对于这样一个多模块的项目，我们的目的是学习如何灵活使用 `C++ Modules` 来组织程序中的几个模块，所以接下来我们不对这个项目做具体实现，主要看看如何编写接口。

`command/argument.cpp` 模块定义了 `Argument` 类，用于描述命令行参数。



- 第 1 行通过 `export module` 声明了这个文件属于 `ips.command` 模块的 `argument` 分区，并用 `export` 表明这是一个模块接口编译单元。
- 第 4 行定义了 `Argument` 类，并通过 `export` 将其标志为外部链接性，对其他模块可见。

小知识：嵌套命名空间

我们在第 3 行使用了 C++17 的新特性——嵌套命名空间（Nested Namespaced）。

在 C++17 之前，定义多层的 `namespace` 需要使用两个嵌套的 `namespace`，先通过 `namespace ips` 定义命名空间 `ips`，再通过 `namespace command` 定义 `ips` 中的命名空间 `command`，非常繁杂。

为了简化命名空间的定义，C++17 加入了嵌套命名空间特性，像在 Java 中定义包名一样，也和 C++ 引用一个命名空间中的符号一样，我们可以直接使用 `::` 连接多个命名空间名称直接定义一个嵌套的命名空间，简化了编写代码的过程，也更加鼓励我们使用合理的命名空间。

因此我们在代码中采用了嵌套命名空间来做命名空间定义，使代码更加清晰。

```
1 export module ips.command:argument;
2 import <string>;
3 namespace ips::command {
4     export class Argument {
5     public:
6         Argument(
7             const std::string& flag,
8             const std::string& name,
9             const std::string& helpMessage = "",
10            bool required = false
11        ) :
12            _flag(flag),
13            _name(name),
14            _helpMessage(helpMessage),
15            _required(required) {}
16        const std::string& getFlag() const {
17            return _flag;
18        }
```

```

19     void setFlag(const std::string& flag) {
20         _flag = flag;
21     }
22     const std::string& getHelpMessage() const {
23         return _helpMessage;
24     }
25     void setHelpMessage(const std::string& helpMessage) {
26         _helpMessage = helpMessage;
27     }
28     bool isRequired() const {
29         return _required;
30     }
31     void setRequired(bool required) {
32         _required = required;
33     }
34
35     private:
36         std::string _flag;
37         std::string _name;
38         std::string _helpMessage;
39         bool _required;
40 };
41 ~

```



天下无鱼

<https://shikey.com/>

command/parser.cpp

本模块定义了 Parser 类，用于解析命令行参数。

- 第 1 行通过 `export module` 声明了这个文件属于 `ips.command` 模块的 `parser` 分区，并用 `export` 表明这是一个模块接口编译单元。
- 第 11 行定义了 `Parser` 类，并通过 `export` 将其标志为外部链接性，对其他模块可见。

 复制代码

```

1  export module ips.command:parser;
2
3  import <string>;
4  import <map>;
5  import <vector>;
6  import <functional>;
7
8  import :argument;
9
10 namespace ips::command {
11     export class Parser {
12     public:
13         Parser& addArgument(const Argument& argument) {

```

```

14         _arguments.push_back(argument);
15
16         return *this;
17     }
18
19     std::map<std::string, std::string> parseArgs() {
20         return _parsedArgs;
21     }
22
23     std::string getNamedArgument(const std::string& name) {
24         std::string value = _parsedArgs[name];
25
26         value;
27     }
28
29     template <class T>
30     T getNamedArgument(const std::string& name, std::function<T(const std::
31         std::string value = _parsedArgs[name];
32
33         return converter(value);
34     }
35
36 private:
37     std::vector<Argument> _arguments;
38     std::map<std::string, std::string> _parsedArgs;
39 };
40 }

```



天下无鱼

<https://shikey.com/>

command/command.cpp 模块是整个 **ips.command** 模块的外部接口。在第 3 行和第 4 行通过 **export import** 导入并重新导出了 **:parser** 和 **:argument** 分区，这样我们可以将一个模块下的不同类分别在不同分区中实现，并在主模块接口单元中导入再导出，便于我们维护代码。

 复制代码

```

1 export module ips.command;
2
3 export import :parser;
4 export import :argument;

```

网络服务接口模块

network/request.cpp 模块定义了 **Request** 类，用于描述 HTTP 网络请求。

- 第 1 行通过 **export module** 声明了这个文件属于 **ips.network** 模块的 **request** 分区，并用 **export** 表明这是一个模块接口编译单元。

- 第 7 行定义了 Request 类，并通过 export 将其标志为外部链接性，对其他模块可见。



```
1 export module ips.network:request;
2
3 import <string>;
4 import <map>;
5
6 namespace ips::network {
7     export class Request {
8     public:
9         Request() {}
10
11         void setPath(const std::string& path) {
12             _path = path;
13         }
14
15         const std::string& getPath() {
16             return _path;
17         }
18
19         void setQuery(const std::map<std::string, std::string>& query) {
20             _query = query;
21         }
22
23         const std::map<std::string, std::string>& getQuery() {
24             return _query;
25         }
26
27         std::string&& getBody() {
28             return "";
29         }
30
31     private:
32         std::string _path;
33         std::map<std::string, std::string> _query;
34     };
35 }
```

network/response.cpp 模块定义了 Response 类，用于描述 HTTP 网络响应。

- 第 1 行通过 export module 声明了这个文件属于 ips.network 模块的 response 分区，并用 export 表明这是一个模块接口编译单元。
- 第 7 行定义了 Response 类，并通过 export 将其标志为外部链接性，对其他模块可见。

```
1 export module ips.network:response;
2
3 import <string>;
4 import <iostream>;
5
6 namespace ips::network {
7     export class Response {
8     public:
9         Response() {}
10
11         void send(const std::string& data) {
12             std::cout << "Sent data" << data.size() << std::endl;
13         }
14     };
15 }
16 }
```



network/connection.cpp 模块定义了 Connection 类，用于描述 HTTP 网络连接。

- 第 1 行通过 `export module` 声明了这个文件属于 `ips.network` 模块的 `connection` 分区，并用 `export` 表明这是一个模块接口编译单元。
- 第 11 行使用 `using` 定义了类型别名 `RequestPtr`，表示请求对象指针，并通过 `export` 将该符号导出。
- 第 12 行使用 `using` 定义了类型别名 `ResponsePtr`，表示响应对象指针，并通过 `export` 将该符号导出。
- 第 14 行使用 `using` 定义了类型别名 `OnRequestHandler`，表示请求处理函数，并通过 `export` 将该符号导出。
- 第 16 行定义了 `Connection` 类，并通过 `export` 将其标志为外部链接性，对其他模块可见。

```
1 export module ips.network:connection;
2
3 import <functional>;
4 import <memory>;
5 import <vector>;
6
7 import :request;
8 import :response;
9
10 namespace ips::network {
11     export using RequestPtr = std::shared_ptr<Request>;
```

```

12     export using ResponsePtr = std::shared_ptr<Response>;
13
14     export using OnRequestHandler = std::function<void(RequestPtr, ResponsePtr)>;
15
16     export class Connection {
17     public:
18         Connection() {}
19
20         void onRequest(OnRequestHandler requestHandler) {
21             _requestHandlers.push_back(requestHandler);
22         }
23
24     private:
25         std::vector<OnRequestHandler> _requestHandlers;
26     };
27 }

```



network/server.cpp 模块定义了 **Server** 类，用于实现 HTTP 服务器。

- 第 1 行通过 **export module** 声明了这个文件属于 **ips.network** 模块的 **server** 分区，并用 **export** 表明这是一个模块接口编译单元。
- 第 13 行使用 **using** 定义了类型别名 **ConnectionPtr**，表示连接对象指针，并通过 **export** 将该符号导出。
- 第 14 行使用 **using** 定义了类型别名 **OnConnectionHandler**，表示连接处理函数，并通过 **export** 将该符号导出。
- 第 16 行定义了 **Server** 类，并通过 **export** 将其标志为外部链接性，对其他模块可见。

 复制代码

```

1  export module ips.network:server;
2
3  import <string>;
4  import <cstdint>;
5  import <functional>;
6  import <vector>;
7  import <memory>;
8  import <iostream>;
9
10 import :connection;
11
12 namespace ips::network {
13     export using ConnectionPtr = std::shared_ptr<Connection>;
14     export using OnConnectionHandler = std::function<void(ConnectionPtr)>;
15
16     export class Server {

```



```

17     public:
18         Server(const std::string& host, int32_t port) :
19             _host(host), _port(port) {}
20
21         void setHost(const std::string& host) {
22             _host = host;
23         }
24
25         const std::string& getHost() const {
26             return _host;
27         }
28
29         void setPort(int32_t port) {
30             _port = port;
31         }
32
33         int32_t getPort() const {
34             return _port;
35         }
36
37         void startListen() {
38             std::cout << "Start listened at " << _host << ":" << _port << std::
39         }
40
41         void onConnection(OnConnectionHandler handler) {
42             _handlers.push_back(handler);
43         }
44
45     private:
46         std::string _host;
47         int32_t _port;
48         std::vector<OnConnectionHandler> _handlers;
49     };
50 }


```



天下无鱼
<https://shikey.com/>

network/network.cpp

本模块是整个 ips.network 模块的外部接口。导入并导出了 ips.network 下的所有分区。

 复制代码

```

1 export module ips.network;
2
3 export import :server;
4 export import :request;
5 export import :response;
6 export import :connection;

```

`images/processor.cpp` 模块定义了 `Processor` 类，用户实现图像处理。



- 第 1 行通过 `export module` 声明了这个文件属于 `ips.image` 模块的 `processor` 分区，并用 `export` 表明这是一个模块接口编译单元。
- 第 7 行定义了 `Processor` 类，并通过 `export` 将其标志为外部链接性，对其他模块可见。

 复制代码

```
1 export module ips.image:processor;
2
3 import <string>;
4 import <cstdint>;
5
6 namespace ips::image {
7     export class Processor {
8     public:
9         void setWidth(int32_t width) {
10             _width = width;
11         }
12
13         int32_t getWidth() const {
14             return _width;
15         }
16
17         void setHeight(int32_t height) {
18             _height = height;
19         }
20
21         int32_t getHeight() const {
22             return _height;
23         }
24
25         void setQuality(int32_t quality) {
26             _quality = quality;
27         }
28
29         int32_t getQuality() const {
30             return _quality;
31         }
32
33         void setMode(const std::string& mode) {
34             _mode = mode;
35         }
36
37         const std::string& getMode() const {
38             return _mode;
39         }
40     }
```

```

40         std::string&& processImage(const std::string& data) {
41             return "";
42         }
43     }
44
45     private:
46         int32_t _width;
47         int32_t _height;
48         int32_t _quality;
49         std::string _mode;
50     };
51 }

```



images/image.cpp

本模块是整个 ips.image 模块的外部接口。导入并导出了 ips.image 下的所有分区。

复制代码

```

1  export module ips.image;
2
3  export import :processor;

```

本地缓存模块

cache/loader.cpp 模块定义了 Loader 类，用户实现缓存加载。

- 第 1 行通过 `export module` 声明了这个文件属于 `ips.cache` 模块的 `loader` 分区，并用 `export` 表明这是一个模块接口编译单元。
- 第 6 行定义了 `CacheLoader` 类，并通过 `export` 将其标志为外部链接性，对其他模块可见。

复制代码

```

1  export module ips.cache:loader;
2
3  import <string>;
4
5  namespace ips::cache {
6      export class CacheLoader {
7      public:
8          CacheLoader(const std::string& basePath) :
9              _basePath(basePath) {}
10

```

```

11         bool loadCacheFile(const std::string& key, std::string* cacheFileData)
12             return false;
13     }
14
15     private:
16         std::string _basePath;
17     };
18 }

```



天下无鱼

<https://shikey.com/>

cache/cache.cpp 模块是整个 ips.cache 模块的外部接口。导入并导出了 ips.cache 下的所有分区。

复制代码

```

1 export module ips.cache;
2
3 export import :loader;

```

业务应用模块

app/app.cpp 这个模块是整个 ips.app 模块的外部接口。由于比较简单，只定义了一个 processRequest 函数，因此没有定义其他的分区。

- 第 1 行通过 export module 声明了这个文件为 ips.app 模块，并用 export 表明这是一个模块接口编译单元。
- 第 23 行定义了 processRequest 类，并通过 export 将其标志为外部链接性，对其他模块可见。

复制代码

```

1 export module ips.app;
2
3 import <string>;
4 import <map>;
5
6 import ips.network;
7 import ips.image;
8 import ips.cache;
9
10 namespace ips::app {
11     export void processRequest(
12         ips::cache::CacheLoader* cacheLoader,
13         ips::network::RequestPtr request,
14         ips::network::ResponsePtr response

```

```

15     ) {
16         const std::string& path = request->getPath();
17         const std::map<std::string, std::string>& query = request->getQuery();
18         std::string data = request->getBody();
19         ips::image::Processor imageProcessor;
20         std::string cacheKey = path;
21
22         auto widthIterator = query.find("width");
23         if (widthIterator != query.cend()) {
24             imageProcessor.setWidth(std::stoi(widthIterator->second));
25             cacheKey += "&width=" + widthIterator->second;
26         }
27
28         auto heighIterator = query.find("height");
29         if (heighIterator != query.cend()) {
30             imageProcessor.setHeight(std::stoi(heighIterator->second));
31             cacheKey += "&height=" + heighIterator->second;
32         }
33
34         auto qualityIterator = query.find("quality");
35         if (qualityIterator != query.cend()) {
36             imageProcessor.setQuality(std::stoi(qualityIterator->second));
37             cacheKey += "&quality=" + qualityIterator->second;
38         }
39
40         auto modeIterator = query.find("mode");
41         if (modeIterator != query.cend()) {
42             imageProcessor.setMode(modeIterator->second);
43             cacheKey += "&mode=" + modeIterator->second;
44         }
45
46         std::string processedImageData;
47         bool hasCache = cacheLoader->loadCacheFile(cacheKey, &processedImageDat
48         if (hasCache) {
49             response->send(processedImageData);
50
51             return;
52         }
53
54         processedImageData = imageProcessor.processImage(data);
55         response->send(processedImageData);
56     }
57 }


```



主程序调用

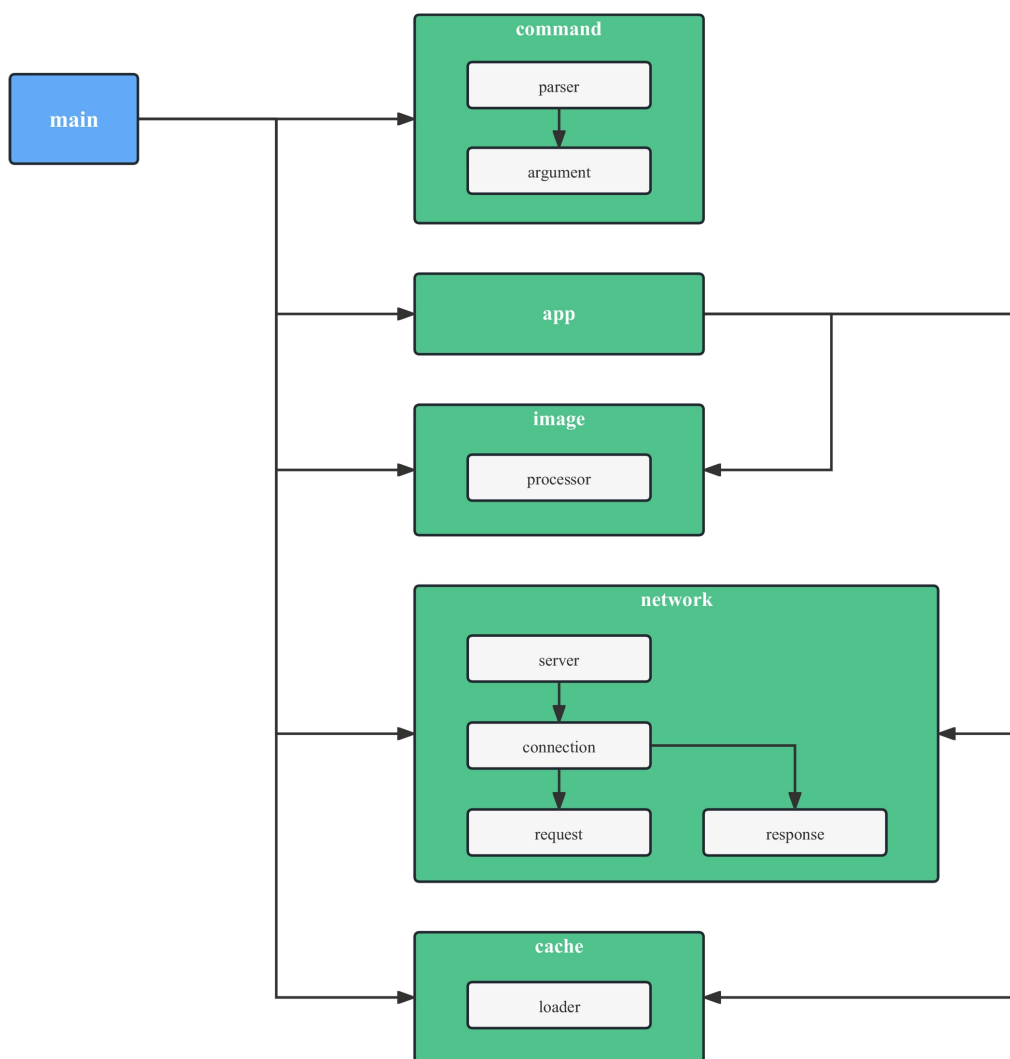
main.cpp 是整个程序的调用的模块，首先创建命令行解析器，对命令行进行解析，接着创建 HTTP 服务器和缓存加载器，最后注册连接处理函数和请求处理函数，并启动监听，进入事件循环。

- 第 1 到 3 行，通过 `import` 导入 C++ 标准库的头文件。
- 第 5-9 行，通过 `import` 导入项目内部的各个模块，后面就能使用这些模块内的符号了。

 复制代码

```
1 import <iostream>;
2 import <string>;
3 import <functional>;
4
5 import ips.command;
6 import ips.network;
7 import ips.image;
8 import ips.app;
9 import ips.cache;
10
11 int main() {
12     std::cout << "Image Processor" << std::endl;
13
14     ips::command::Parser parser;
15     parser.addArgument(ips::command::Argument("--host", "host"));
16     parser.addArgument(ips::command::Argument("--port", "port"));
17     parser.addArgument(ips::command::Argument("--cache", "cachePath"));
18     parser.parseArgs();
19
20     std::string cachePath = parser.getNamedArgument("cachePath");
21     ips::cache::CacheLoader cacheLoader(cachePath);
22
23     std::string host = parser.getNamedArgument("host");
24     int port = parser.getNamedArgument<int32_t>("port", [] (const std::string& v
25         return std::stoi(value);
26     });
27     ips::network::Server server(host, port);
28
29     server.onConnection([&cacheLoader] (ips::network::ConnectionPtr connection)
30         connection->onRequest(std::bind(
31             ips::app::processRequest,
32             &cacheLoader,
33             std::placeholders::_1,
34             std::placeholders::_2
35         ));
36     });
37
38     server.startListen();
39
40     return 0;
41 }
```

相对于传统的方法，我们不需要关心头文件和符号实现的各种细节，C++ Modules 规定我们将接口和实现都组织在通过模块关联的代码文件中，虽然灵活性相对较低，但在一般的工程实践中，这样的代码组织更加合理，也能降低模块开发者和使用者的心智负担。



深入理解 C++ Modules

掌握了 C++ Modules 的基础概念，也通过实例体会了 C++ Modules 的用法和好处，我们再回过头来，站在语言设计者的角度，讨论一下 C++ Modules 中一些深层次问题，C++ Modules 核心语言特性变更到底能为我们带来什么？它能解决什么，同时又不能解决什么问题？

Modules 能解决什么

首先需要理解 Modules 到底帮助我们解决了什么问题？在 C++ Modules 的基本概念介绍中，我们说过了 Modules 解决的是符号可见性问题。

在传统的 C++ 解决方案中，处理符号可见性，需要我们充分理解 C++ 的“编译 - 链接”原理，甚至很多的实现技术细节。



由于 C++ 中的各个编译单元需要独立编译，同时在链接中通过检索符号填补缺失的符号，我们不仅要在实现符号的编译单元中编写实现，还要在引用的编译单元中，通过书写符号声明，来告知编译器这些符号会在链接过程中存在。所以，我们需要通过头文件来为模块调用的编译单元提供这些必要的前置符号声明。

这就出现了一个问题：**模块之间的符号引用，因为这种“编译 - 链接”机制被硬生生拆分成两个阶段**。哪怕能通过编译，也可能在链接时产生错误，而这种错误也很难被编译器和 IDE 在编译阶段提前侦测到，更多的问题将链接时才暴露出来。

只有经验丰富的 C++ 工程师，在了解基本的“编译 - 链接”原理后，才能熟练排查这些因为两阶段的不一致性导致的链接问题，并找到方法尝试解决。因此传统的符号可见性解决方案，对 C++ 初学者不友好。

新的 C++ Modules 方法，本质上抛弃了“头文件”这种 C/C++ 中的重要组成部分，将头文件转换成了模块接口文件，也为 C++ 提供了一种在编译期检测声明实现不一致的方法，也为 IDE 的智能提示提供了新的可靠方法。

另外，C++ Modules 也部分抛弃了 C/C++ 原本通过简单的文本处理为编译单元引入声明的方式，使得编译器可以为模块编译单元生成二进制的编译缓存，为加快编译过程提供了一个新的契机。

所以简单来说，C++ Modules 给我们带来了一种更为现代化的，更简单的符号可见性控制方案，同时又能加快编译速度。

Modules 不能解决什么

那么，Modules 不能解决什么呢？

第一，Modules 不能解决**符号命名冲突**的问题。

在实例中，你会发现我们在代码中同时使用了命名空间和 Modules，通过 Modules 来控制符号可见性，然后使用命名空间来避免符号命名冲突。

符号命名冲突，可能因为两个不同的模块使用了相同名称的函数、类、全局变量等，并将其 `export` 出来，如果这两个模块同时 `import` 到同一个编译单元中，就会出现冲突，因为编译器并不知道我们使用的是哪个模块中的符号。因此在不同的模块中，我们仍习惯使用不同的命名空间，确保一个编译单元导入两个模块的时候不会出现模块冲突。

这就是我们一直所说的，模块只解决符号可见性问题，而命名冲突问题依然需要通过 `namespace` 解决，这就是 `Modules` 和 `namespace` 是保持正交设计的。

第二，目前 `Modules` 不能用来解决**二进制库分发**的问题。

现阶段，编译器在编译模块编译单元的过程中，会为每个模块编译单元生成对应的二进制缓存，无论是模块接口单元还是模块实现单元都会生成，甚至通过 `import` 导入 `iostream` 这种标准库，也会为 `iostream` 生成二进制缓存。

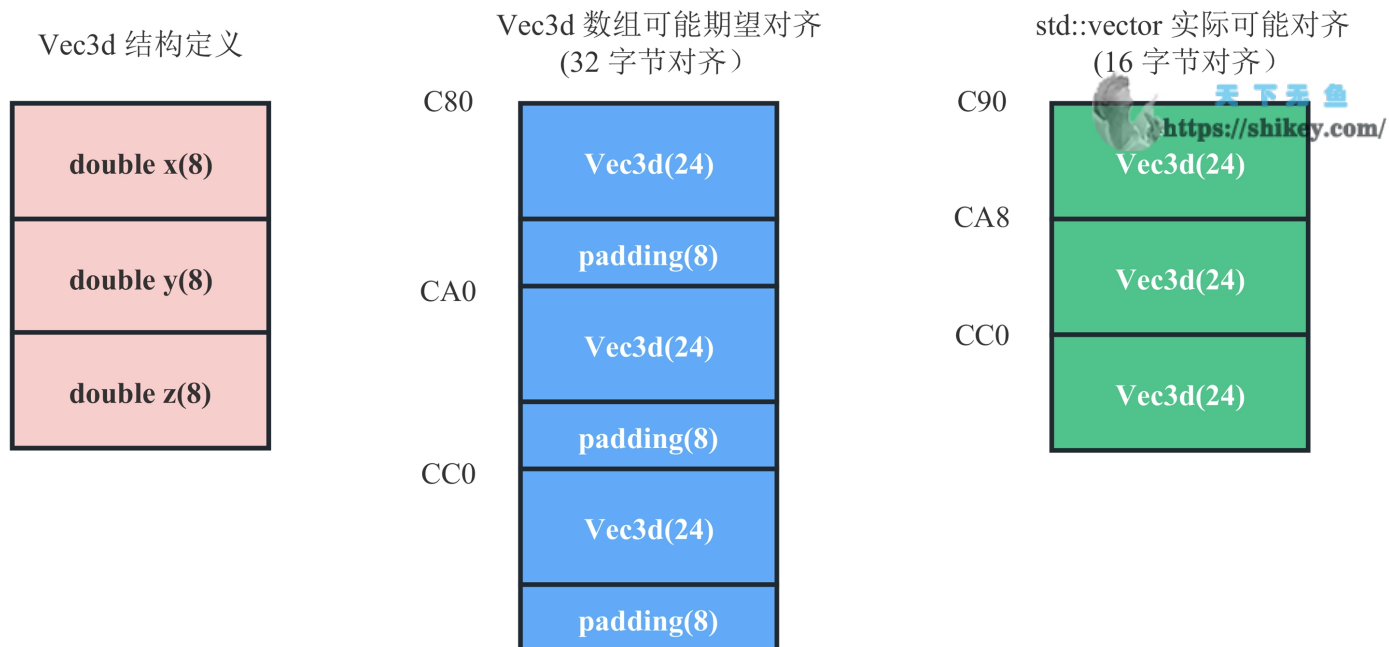
这些二进制缓存，不仅包括编译后生成的中间码、机器码，还包括源代码之类的 `meta` 数据，这样，其他编译模块在通过 `import` 导入模块的时候，编译器将会直接读取二进制缓存，不需要在预处理阶段做文本替换，再在各个编译单元的编译过程中进行编译，可以加快编译速度。

但我们要注意的是，在生成的静态链接库或者动态链接库中，标准并没有定义需将这些缓存中的 `meta` 数据加入到库中。

因此，目前通过 `Modules` 编写的代码，在进行二进制分发时，会面临很多问题，只有 `Visual C++`（自 `Visual Studio 2022` 起）通过标头单元来实现通过 `import` 导入（可以读取编译器自动生成的二进制缓存 `ifc` 文件，`ifc` 文件是 `VC` 编译单元生成的标头单元二进制缓存文件格式），其他的编译器只支持通过源代码分发的方式来使用 `import`。

第三，**STL 内存布局**问题。

在使用 `STL` 的过程中，我们会遇到 `ABI` 与内存布局的很多问题。比如一些 `SIMD` 的计算场景，需要调用 `CPU` 的加速指令，而这些加速指令对数据的内存地址对齐都有严格要求，因此我们可能需要可以预期的内存对齐结果。但是，实际上内存对齐会受到编译器和体系结构影响，如下图。



自己管理内存，可以产生我们预期的内存对齐效果，但如果使用 **STL**，则需要依赖编译器和体系结构，可能无法产生我们所预期的内存对齐。这只是 **STL** 内存布局问题的冰山一角。

现阶段的 **Modules** 暂时无法解决**各编译器之间 ABI**，尤其是使用模板后的问题。

目前，编译和链接还是会依赖编译器和体系结构定义的 **ABI**，所以，如果 **A** 编译器生成的二进制符号格式，不同于 **B** 编译器的二进制符号格式，那么 **B** 编译器也就无法使用 **A** 编译器生成的库（无论是动态链接库还是静态链接库），更不用说不同编译器生成的二进制缓存文件了。

我们了解 **C++ Modules** 能做什么，不能做什么，就知道该在什么场景如何使用 **C++ Modules** 了。总的来说，目前 **C++ Modules** 的支持还不够完善，不同的 **C++** 编译器，对现代 **C++** 新标准的支持情况各不相同，这里也给出当下主流编译器对新特性的支持情况。

核心语言功能	备注	GCC 支持	Clang 支持	VS 支持
C++ Modules (P1103R3)	C++ Modules 核心提案	11	Clang 15	VS 2019 16.8
P1766R1	修正部分 Modules 小问题	否	Clang 11	VS 2019 16.8
P1811R0	放松符号重定义限制 (提升重新导出符号的健壮性)	11	否	VS 2019 16.8
P1703R1	解决通过 import 导入模块之前需要进行完整预处理的问题	11	否 (被 P1857 取代)	VS 2019 16.5
P1874R1	解决模块的非局部变量动态初始化顺序问题	11	Clang 15	VS 2019 16.8
P1979R0	针对 US086 的解决方案, 避免全局模块片段导致的间接 import 问题	11	否	VS 2019 16.6
P1779R3	成员函数的 ABI 隔离问题	11	Clang 15	否
P1857R3	完善模块的依赖扫描	11	否	否
P2115R0	解决匿名无作用域枚举量的重复定义合并问题	11	Clang 15 (部分支持)	否
P1815R2	编译单元内部符号定义处理问题	否	Clang 15 (部分支持)	否

(2022年12月)

极客时间

随着编译器支持越来越成熟, 相信会带来更多的编译性能提升, 就像编译器对头文件支持的性能提升一样。

总结

自 C++20 标准开始, C++ Modules 给我们带来了一种更为现代化的、更简单的符号可见性控制方案, 同时又能加快编译速度。

总体上看, C++ Modules 很好地提供了解决符号可见性问题的方案。在传统的 C++ 解决方案中, 处理符号可见性, 需要在充分理解 C++ 的“编译 - 链接”原理甚至很多实现技术细节, 而现在, 我们可以更简单地掌控符号的可见性, 并在不牺牲编译性能的情况下使用 C++ 进行编码。

但是目前 C++ Modules 并不是完美的。

1. 不能解决符号命名冲突的问题。
2. 不能用来解决二进制库分发的问题。
3. 现阶段的 Modules 暂时无法解决各编译器之间 ABI, 尤其是使用模板之后带来的问题。

随着现代 C++ 标准化进程的稳步推进，我们期待着这些问题能够在未来得到标准和编译器的统一支持。C++ Modules 已经逐渐成为解决编译性能和符号隔离的银弹，但我们让这枚子弹“再飞一会儿”。




课后思考

这节课，我们了解到，C++ Modules 带来了极大的便利性，不过当前也仍然存在一些功能限制，你能否举出在日常使用 C++ 过程中碰到的有关于符号的编译、链接问题，并给出你的解决方法。

欢迎留言和我分享你的想法，我们一同交流！

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 2  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。

上一篇 02 | Modules（中）：解决编译性能和符号隔离的银弹

下一篇 04 | Concepts背景：C++泛型编程之殇

精选留言 (4)

 写留言



wilby

2023-01-20 来自瑞典

怎么编译这个项目呢？在macOS下没试出来怎么编译

作者回复: 目前来说，对C++20后续演进标准的支持，Visual C++ 支持最好。macOS默认使用 Xcode 提供的 clang（llvm），此外，gcc、clang对新标准的支持存在一定差异。你可以稍后参考我分享的代码仓库来使用CMake尝试编译（即将提供）。

也欢迎提供反馈，甚至是PR

**浩浩**

2023-01-31 来自广东

**天下无鱼**<https://shikey.com/>

请教老师：

- 1) 文中：“现阶段的 **Modules** 暂时无法解决各编译器之间 **ABI**”，**C/C++** 现今有解决这个问题的方案吗？
- 2) 文中：“二进制库分发的问题”，具体是指二进制库在不同体系结构设备上无法通用吗？

作者回复: 1) **C/C++** 现在还是没有解决这个问题。2) 指的是二进制库甚至在相同体系结构但采用**ABI**不同的编译器之间都无法通用。**Coding Fatty**

2023-01-25 来自辽宁

ips 表示什么含义？

作者回复: image processor solution

**peter**

2023-01-20 来自北京

请教老师几个问题：

Q1: **cpp**文件中怎么会有变量声明？命令行模块的代码中，**argument.cpp**中怎么声明了一些字段？：`private: std::string _flag; std::string _name;` 这些不是应该在**.h**文件中声明吗？**Cpp**文件是类的实现文件啊。**Q2:** 用了**module**以后就不再有头文件了吗？文中有一句“新的 **C++ Modules** 方法，本质上抛弃了“头文件”这种 **C/C++** 中的重要组成部分”。采用**module**以后，不再是“**.h + .cpp**”这种方式，而时只有**.cpp**一个文件，对吗？**Q3:** **C80**、**CA0**表示什么？文中“**STL 内存布局问题**”部分，图中有“**C80**、**CA0**”等内容，是表示**CPU**类型吗？作者回复: **Q1:** 我们现在用的是**C++ Modules**，所以所有文件都是**cpp**（也就是模块接口或实现），不再有**.h**的概念了**Q2:** 这点就是对应**Q1**，除非引用标准库和特殊用途（比如配置），一般情况下完全不需要有头文件这个概念了，只需要用**.cpp**即可**Q3:** **C80**和**CA0**指的是内存地址

