17 | Bit library（二）：如何利用新bit操作库释放编程生产力？

2023-03-01 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 11:56　大小 10.89M

你好，我是卢誉声。

在上一讲中，我们已经通过一些简单的编程示例，展示了 C++20 及其后续演进提供的位操作库的基本使用方法。

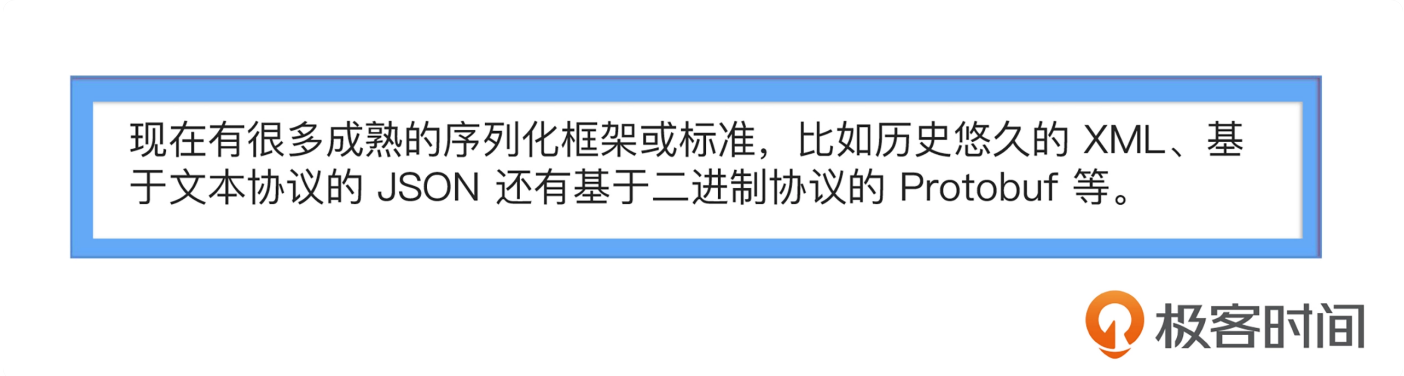
但是，简单的示例还无法体现位操作库的真正威力。所以，这一讲我会通过一个较为完整的工程代码，带你体会如何充分利用全新的位操作库，实现强大的序列化和反序列化功能以及位运算。

好，话不多说，就让我们开始吧（课程配套代码可以从这里获取）。

扩展数据流处理实战案例

在实际生产环境中，我们经常需要通过网络传输特定的数据，但是不同语言和不同平台的内存模型可能完全不同。这时，发送方需要将数据转换为符合特定标准的数据流，接收方将数据解析后转换为内部变量。

我们将变量转换为数据流的操作称为“序列化（Serialization）”，将数据流转换成变量的过程称为“反序列化（Deserialization）”。



我们曾在第 13 讲——Ranges 实战：数据序列函数式编程中，实现了一个获取三维模型数据并进行统计的程序。

今天，我们将继续对其进一步扩展，使用位操作库实现序列化和反序列化。

不知道你是否还记得，我在第 13 讲偷了一个懒，直接使用了硬编码的代码作为数据输入。我们会在这一讲改进一下，将本地的二进制文件读取到内存里，将其转换成内部变量。

与此同时，我们还新增了一项计算需求：在三维模型对象中，新增了 renderChannels 字段——用来表示某个对象支持哪些渲染通道。在统计过程中，需要确定某个三维模型对象是否只有一个渲染通道。增加这项计算需求的目的在于，演示如何使用位运算替代朴素实现，实现更高效的算数运算。

基于 C++20 位操作库实现

针对这些变化，我们来看一看如何基于 C++20 位操作库，来进行编程实现。

数据结构

根据需求，我们首先要更新基础类型定义，修改内容在 Types.h 中。更新后的 ModelView 类型的定义，代码是后面这样。

复制代码

```
struct ModelView {
    // API接口中的视图对象数据
    struct Object {
        // 对象精度类型
        enum class ResolutionType {
            High,
            Low
        };

        // 渲染通道集合，每个渲染通道占用一个bit，通过位运算设置获取特定的渲染通道
        using RenderChannelBits = uint8_t;

        // 渲染通道枚举类，每个枚举都通过移位生成，使得枚举量独自占用一个bit
        struct RenderChannel {
            enum {
                Buffer = 0b1u,
                Window = 0b1u << 1,
                Image = 0b1u << 2,
                Printer = 0b1u << 3
            };
        };

        // 对象类型ID
        Id objectTypeID;
        // 对象名称
        std::string name;
        // 对象中各部件的面片数量（数组）
        std::vector<int32_t> meshCounts;
        // 对象支持的渲染通道
        RenderChannelBits renderChannels;
    };

    // 视图ID
    Id viewId = 0;
    // 视图类型名称
    std::string viewTypeName;
    // 视图名称
    std::string viewName;
    // 创建时间
    std::string createdAt;
    // 视图对象列表
    std::vector<Object> viewObjectList;
};
```

没有变化的成员函数这里就不写出来了。我们把关注点放在更新的部分。

首先，我们新增了 RenderChannelBits 类型与 RenderChannel 枚举。

RenderChannel 枚举的每个枚举量都对应一个数字，每个数字只有 1 位是 1，其他都是 0，并且不同枚举量占用的位置必须不同。这样一来，就可以通过位运算设置一个对象有哪些渲染通道，也可以通过位运算获取一个对象是否具有某个渲染通道，本质就是一个通过位运算实现的“集合类型”。

代码中将该类型定义为 RenderChannelBits，其优点是存储空间小、计算快，早些年计算机存储容量与计算性能有限的时候，这种技术运用非常广泛。

代码中定义了 renderChannels 成员变量，其类型就是 RenderChannelBits，是对象所有渲染通道的集合。

接下来还要修改 ModelObject 的定义，让我们对照代码往下看。

复制代码

```
// 统计后存储的模型对象数据
struct ModelObject {
    // 视图序号
    int32_t viewOrder = 0;
    // 视图ID
    Id viewId = 0;
    // 视图类型名称
    std::string viewTypeName;
    // 视图名称
    std::string viewName;
    // 视图创建时间
    ZonedTime createdAt;

    // 对象类型ID
    Id objectTypeID = 0;
    // 对象名称
    std::string objectName;
    // 对象包含的三角面片数量
    int32_t meshCount = 0;
    // 对象在视图中的序号
    int32_t viewObjectIndex = 0;
    // 视图中剩余已用三角面片数量
    int32_t viewUsedMeshCount = 0;
    // 视图中可用三角面片数量上限
    int32_t viewTotalMeshCount = 0;
    // 视图中剩余可用三角面片数量
    int32_t viewFreeMeshCount = 0;
    // 视图中对象数量
    size_t viewObjectCount = 0;
    // 对象支持的渲染通道
    ModelView::Object::RenderChannelBits renderChannels;
```

```
    // 是否只有一个渲染通道
    bool onlyOneRenderChannel = 0;

    // 获取完整视图名称
    std::string getCompleteViewName() const {
        return viewTypeName + "/" + viewName;
    }

    // 获取对象Key
    std::string getObjectKey() const {
        return getObjectKey(objectTypeID, viewId);
    }

    // 根据objectTypeID和viewId获取对象Key
    static std::string getObjectKey(Id objectTypeID, Id viewId) {
        return std::to_string(objectTypeID) + "-" + std::to_string(viewId);
    }
};
```

可以看到，代码中新增了 renderChannels 和 onlyOneRenderChannel。其中 onlyOneRenderChannel 需要通过对 renderChannels 统计计算得出。

修改类型定义后，我们修改一下获取数据的函数，代码在 src/data.cpp 中，这部分代码为每个 ModelView::Object 对象都添加了 renderChannel。这个新的成员变量是为了给后续位运算的代码使用的。

因为代码中数据定义很多，这里只截取了部分代码。你可以重点看看，这段代码里是怎样通过位或（|）生成渲染通道集合的。

复制代码

```
#include "data.h"

ModelObjectsInfo getModelObjectsInfo() {
    using RenderChannel = ca::types::ModelView::Object::RenderChannel;

    return {
        .modelViews = {
            {
                .viewId = 1,
                .viewTypeName = "Building",
                .viewName = "Terminal",
                .createdAt = "2020-09-01T08:00:00+0800",
                .viewObjectList = {
                    {
                        .objectTypeID = 1,
                        .name = "stair",
```

```
                        .meshCounts = { 2000, 3000, 3000 },
                        .renderChannels = RenderChannel::Buffer | RenderChannel
                    },
                    {
                        .objectTypeID = 2,
                        .name = "window",
                        .meshCounts = { 3000, 4000, 4000 },
                        .renderChannels = RenderChannel::Buffer
                    },
                    {
                        .objectTypeID = 3,
                        .name = "pool",
                        .meshCounts = { 100, 101 },
                        .renderChannels = RenderChannel::Buffer | RenderChannel
                    },
                    {
                        .objectTypeID = 4,
                        .name = "pinball arcade",
                        .meshCounts = { 1000, 999 },
                        .renderChannels = RenderChannel::Buffer | RenderChannel
                    },
                },
            },
            ……
}
```

序列化与反序列化

接下来，我们需要考虑如何实现数据的序列化和反序列化。为了演示二进制操作，我这里并没有使用成熟的序列化框架，而是自己实现了一个简单的二进制序列化与反序列化框架。

二进制序列化框架首先需要确定数据字节序，在我们的框架中，以大端作为标准字节序，因此就需要实现字节序的检测与转换，具体实现在 BitUtils.h 中。这段代码是基于第 16 讲“字节序处理”这个部分中的代码修改而来的。

复制代码

```
#pragma once

#include "ca/BitCompact.h"

namespace ca::utils {
    // 如果字节序为小端，并且类型size不为1，需要转换字节序
    template <typename T, std::endian ByteOrder = std::endian::native>
        requires (ByteOrder == std::endian::little && sizeof(T) != 1)
    T consumeBigEndian(T src) {
        return std::byteswap(src);
```

```
11      }
12
13      // 如果字节序为小端，但size为1，不需要转换字节序
14      template <typename T, std::endian ByteOrder = std::endian::native>
15          requires (ByteOrder == std::endian::little && sizeof(T) == 1)
16      T consumeBigEndian(T src) {
17          return src;
18      }
19
20      // 如果字节序为大端，但size为1，不需要转换字节序
21      template <typename T, std::endian ByteOrder = std::endian::native>
22          requires (ByteOrder == std::endian::big)
23      T consumeBigEndian(T src) {
24          return src;
25      }
26
27      // 如果字节序为小端，并且类型size不为1，需要转换字节序
28      template <typename T, std::endian ByteOrder = std::endian::native>
29          requires (ByteOrder == std::endian::little && sizeof(T) != 1)
30      T produceBigEndian(T src) {
31          return std::byteswap(src);
32      }
33
34      // 如果字节序为小端，但size为1，不需要转换字节序
35      template <typename T, std::endian ByteOrder = std::endian::native>
36          requires (ByteOrder == std::endian::little && sizeof(T) == 1)
37      T produceBigEndian(T src) {
38          return src;
39      }
40
41      // 如果字节序为大端，但size为1，不需要转换字节序
42      template <typename T, std::endian ByteOrder = std::endian::native>
43          requires (ByteOrder == std::endian::big)
44      T produceBigEndian(T src) {
45          return src;
46      }
47  }
```

在这段代码中，有两个地方值得注意。

首先，顶部头文件 include/BitCompact.h 的设计，是为了让不支持 C++20 位操作的编译器，能够支持 C++20 位操作。我们将在下个部分具体讨论一下该头文件的实现，这里先不过多扩展了。

其次，代码第 7-25 行，对 consumeBigEndian 的实现做了扩展，针对输入参数只有 1 个字节的情况做了优化，直接返回原始数据。这样做，可以提升运行时的实际性能——这是通过 requires 实现的。

接下来，我们基于 BitUtils 实现序列化和反序列化框架，具体实现在 SerializerUtils.h 和 SerializerUtils.cpp 中。首先我们看一下 SerializerUtils.h。

复制代码

```cpp
#pragma once

#include "ca/Types.h"
#include "ca/BitUtils.h"
#include <ostream>
#include <concepts>
#include <string>
#include <iostream>
#include <vector>
#include <set>

namespace ca::utils {
    // 序列化类，序列化输出到特定输出流中
    class Serializer {
    public:
        // 构造函数，绑定特定的输出流
        Serializer(std::ostream& os) : _os(os) {}

        // 禁止拷贝
        Serializer(const Serializer& rhs) = delete;
        // 禁止赋值
        Serializer& operator=(const Serializer& rhs) = delete;
        // 允许移动
        Serializer(Serializer&& rhs) noexcept : _os(rhs._os) {
        }

        // 将特定的类型的数据转换为大端并输出
        // 一般用于标准数据类型（整数、浮点数）
        template <typename T>
        Serializer& dumpBE(T value) {
            T bigEndianValue = produceBigEndian(value);
            _os.write(reinterpret_cast<char*>(&bigEndianValue), sizeof(bigEndia

            return *this;
        }

        // 输入特定长度的字节数组，不进行字节序转换
        // 一般用于字符串或二进制串等自定义类型
        Serializer& dump(const char* data, std::size_t size) {
            _os.write(data, size);

            return *this;
        }

    private:
        // 输出流引用
```

```
47          std::ostream& _os;
48      };
49
50      // 反序列化类，从特定输入流中反序列化
51      class Deserializer {
52      public:
53          // 构造函数，绑定特定的输入流
54          Deserializer(std::istream& is) : _is(is) {}
55
56          // 禁止拷贝
57          Deserializer(const Deserializer& rhs) = delete;
58          // 禁止赋值
59          Deserializer& operator=(const Deserializer& rhs) = delete;
60          // 允许移动
61          Deserializer(Deserializer&& rhs) noexcept : _is(rhs._is) {
62          }
63
64          // 输入大端数据，并将数据转换为本地字节序
65          // 一般用于标准数据类型（整数、浮点数）
66          template <typename T>
67          Deserializer& loadBE(T& value) {
68              T originalValue = T();
69              _is.read(reinterpret_cast<char*>(&originalValue), sizeof(originalVa
70              value = consumeBigEndian(originalValue);
71
72              return *this;
73          }
74
75          // 输入特定长度的字节数组，不进行字节序转换
76          // 一般用于字符串或二进制串等自定义类型
77          Deserializer& load(char* data, std::size_t size) {
78              _is.read(data, size);
79
80              return *this;
81          }
82
83      private:
84          // 输入流引用
85          std::istream& _is;
86      };
87  }
88
89  // Concept，用于确定类型是否为数值（整数或浮点数）
90  template <typename T>
91  concept Number = std::integral<T> || std::floating_point<T>;
92
93  // 针对数值类型序列化的<<操作符重载，这样可以像C++流类型那样按照流的风格使用Serializer类型
94  template <Number T>
95  ca::utils::Serializer& operator<<(ca::utils::Serializer& ss, T value) {
96      return ss.dumpBE(value);
97  }
98
```

```cpp
// 针对字符串类型序列化的<<操作重载
ca::utils::Serializer& operator<<(ca::utils::Serializer& ss, const std::string&

// 针对数值类型反序列化的>>操作重载，这样可以像C++流类型那样按照流的风格使用Deserializer类型
template <Number T>
ca::utils::Deserializer& operator>>(ca::utils::Deserializer& ss, T& value) {
    return ss.loadBE(value);
}

// 针对字符串类型反序列化的>>操作重载
ca::utils::Deserializer& operator>>(ca::utils::Deserializer& ss, std::string& v

// 针对std::vector序列化的<<操作重载，会递归调用每个元素的序列化实现
template <typename T>
ca::utils::Serializer& operator<<(ca::utils::Serializer& ss, const std::vector<
    // 序列化vector的长度
    ss.dumpBE(value.size());

    // 将元素逐个序列化
    for (const auto& element : value) {
        ss << element;
    }

    return ss;
}

// 针对std::vector反序列化的>>操作重载，会递归调用每个元素的反序列化实现
template <typename T>
ca::utils::Deserializer& operator>>(ca::utils::Deserializer& ds, std::vector<T>
    // 反序列化vector的长度
    std::size_t vectorSize = 0;
    ds.loadBE(vectorSize);
    // 调整vector长度
    value.resize(vectorSize);

    // 逐个反序列数组中的元素
    for (auto& element : value) {
        ds >> element;
    }

    return ds;
}

// 针对std::set序列化的<<操作重载，会递归调用每个元素的序列化实现
template <typename T>
ca::utils::Serializer& operator<<(ca::utils::Serializer& ss, const std::set<T>&
    ss.dumpBE(value.size());

    for (const auto& element : value) {
        ss << element;
    }

```

```
151     return ss;
152 }
153
154 // 针对std::set反序列化的>>操作重载，会递归调用每个元素的反序列化实现
155 template <typename T>
156 ca::utils::Deserializer& operator>>(ca::utils::Deserializer& ds, std::set<T>& v
157     std::size_t setSize = 0;
158     ds.loadBE(setSize);
159
160     for (std::size_t elementIndex = 0; elementIndex != setSize; ++elementIndex)
161         T element = T();
162         ds >> element;
163
164         value.insert(element);
165     }
166
167
168
```

又是一段有点长的代码，不过只要你简单浏览一下，应该还是挺好理解的。这段代码分为三个部分，你可以参考一下表格来具体了解。

序号	行数	代码块	说明
1	第 14 - 48 行	Serializer 类	定义了标准的序列化器包装类，该类型需要绑定特定的输出流，并将数据的序列化结果输出到绑定的输出流中。该类提供了 2 个成员函数用于序列化数据，dumpBE 用于将基础类型的数据转换大小端后写入到输出流中，而 dump 则用于将不需要大小端转换的字节数组写入到输出流中。这两个成员函数都会返回 *this，这样就可以链式调用序列化成员函数，让代码更加优雅。
2	第 51 - 86 行	Deserializer 类	标准的反序列化包装类，标准的反序列化包装类，需要绑定特定输入流，会从输入流中读取数据并反序列到特定变量中。该类提供了 loadBE 和 load 进行反序列化，具体实现与 dumpBE/dump 类似，只不过是逆向过程。
3	第 90 - 168 行	输出/输入操作符重载	针对特定类型实现了 operator<</operator>>，支持像 C++ 的输入输出流那样通过 << 和 >> 进行序列化输出与反序列化输入，同样也便于开发者针对自定义类型进行序列化和反序列化扩展。每个函数也会返回序列化/反序列化流本身，便于像标准流对象那样流式调用。

针对 std::string 类型的序列化、反序列实现在 src/ca/SerializerUtils.cpp 中。如果感兴趣的话，你可以参考完整代码。

自定义类型序列化、反序列化

我们设计的框架，是支持针对自定义类型的序列化和反序列化扩展的。

比如说，如果要序列化或反序列化 ca::types::ModelView、ca::types::ModelView::Object，可以在 include/ca/IoUtils.h 中添加后面这些声明。

复制代码

```
// 序列化ca::types::ModelView
ca::utils::Serializer& operator<<(
    ca::utils::Serializer& serializer,
    const ca::types::ModelView& modelView
);

// 序列化ca::types::ModelView::Object
ca::utils::Serializer& operator<<(
    ca::utils::Serializer& serializer,
    const ca::types::ModelView::Object& object
);

// 反序列化ca::types::ModelView
ca::utils::Deserializer& operator>>(
    ca::utils::Deserializer& deserializer,
    ca::types::ModelView& modelView
);

// 反序列化ca::types::ModelView::Object
ca::utils::Deserializer& operator>>(
    ca::utils::Deserializer& deserializer,
    ca::types::ModelView::Object& object
);

// 然后在src/ca/src/IoUtils.cpp中添加相应实现：
// 序列化ca::types::ModelView
ca::utils::Serializer& operator<<(
    ca::utils::Serializer& serializer,
    const ca::types::ModelView& modelView
) {
    return serializer
        << modelView.viewId
        << modelView.viewTypeName
        << modelView.viewName
        << modelView.createdAt
        << modelView.viewObjectList;
}

// 序列化ca::types::ModelView::Object
ca::utils::Serializer& operator<<(
    ca::utils::Serializer& serializer,
```

```
    const ca::types::ModelView::Object& object
) {
    return serializer
        << object.objectTypeID
        << object.name
        << object.meshCounts
        << object.renderChannels;
}

// 反序列化ca::types::ModelView
ca::utils::Deserializer& operator>>(
    ca::utils::Deserializer& deserializer,
    ca::types::ModelView& modelView
) {
    return deserializer
        >> modelView.viewId
        >> modelView.viewTypeName
        >> modelView.viewName
        >> modelView.createdAt
        >> modelView.viewObjectList;
}

// 反序列化ca::types::ModelView::Object
ca::utils::Deserializer& operator>>(
    ca::utils::Deserializer& deserializer,
    ca::types::ModelView::Object& object
) {
    return deserializer
        >> object.objectTypeID
        >> object.name
        >> object.meshCounts
        >> object.renderChannels;
}
```

这段代码实现非常简单直接，序列化过程就是将对象的成员变量逐个通过 serializer，序列化输出。反序列化过程，则是逐个通过 deserializer 反序列化输入到成员变量中。

我们无法从数据流中知道输入的数据与成员变量的对应关系。因此，这就要求输入输出的顺序必须完全一致。

最后，我们修改一下 src/main.cpp，首先将对象序列化到文件中。然后模拟从外部获取数据流的过程，读取文件反序列化，并基于反序列化的新对象进行统计分析，修改的代码是后面这样。

```cpp
// 序列化ModelObjectsInfo
void serializeModelObjectsInfo() {
    using ca::utils::Serializer;

    // 获取模型对象信息
    auto modelObjectsInfo = getModelObjectsInfo();

    // 序列化
    std::ofstream outFile("ca.dat", std::ios::binary);
    Serializer ss(outFile);
    ss
        << modelObjectsInfo.modelViews
        << modelObjectsInfo.highResolutionObjectSet
        << modelObjectsInfo.meshCount;
}

// 反序列化ModelObjectsInfo
ModelObjectsInfo deserializeModelObjectInfo() {
    using ca::utils::Deserializer;

    ModelObjectsInfo modelObjectsInfo;

    std::ifstream inputFile("ca.dat", std::ios::binary);
    // 反序列化
    Deserializer ds(inputFile);
    ds
        >> modelObjectsInfo.modelViews
        >> modelObjectsInfo.highResolutionObjectSet
        >> modelObjectsInfo.meshCount;

    return modelObjectsInfo;
}

int main() {
    using ca::types::ModelObjectTableData;
    using ResolutionType = ca::types::ModelView::Object::ResolutionType;

    // 生成二进制数据流
    serializeModelObjectsInfo();
    // 获取模型对象信息（从二进制数据流解析）
    auto modelObjectsInfo = deserializeModelObjectInfo();
    ……
    return 0;
}
```

这样一来，我们就实现了最简单的二进制序列化和反序列化。

比较复杂的二进制序列化 / 反序列化框架一般还会包括数据压缩、模式描述（一些标准支持将数据类的结构定义描述在数据流中），感兴趣的话你可以课后自行探索。

使用位运算进行计算

还记得么？我们还有一个需求，就是基于 renderChannels，计算出对象“是否只有一个渲染通道”。

跟上步伐，不要溜号。我们以 Ranges 的统计算法实现为例，看看这个需求该如何实现。我将代码放在了 src/ca/algorithms/RangesAlgorithm.cpp 中。

复制代码

```
static std::vector<ModelObject> extractHighOrLowResolutionObjects(
    const std::vector<ca::types::ModelView>& modelViews,
    const std::set<std::string>& highResolutionObjectSet,
    int32_t meshCount,
    bool isHigh
) {
    auto modelViewsData = modelViews.data();

    // 生成模型对象数据（高精度或双精度）
    // 将模型视图对象数组转换成一个新数组，数组元素是每个模型视图的模型对象数组（返回的是二维数
    return modelViews |
        views::transform([modelViewsData, &highResolutionObjectSet, meshCount,
        // 通过模型视图指针地址计算模型视图序号
            int32_t viewOrder = static_cast<int32_t>(&modelView - modelViewsDat
            const std::vector<ModelView::Object>& viewObjectList = modelView.vi

            auto filteredModelObjects = viewObjectList |
                // 筛选满足要求的对象（高精度或低精度）
                views::filter([&modelView, &highResolutionObjectSet, isHigh](co
                    auto viewId = modelView.viewId;
                    auto objectTypeID = viewObject.objectTypeID;
                    auto objectKey = ModelObject::getObjectKey(objectTypeID, vi

                    return highResolutionObjectSet.contains(objectKey) == isHig
                }) |
                // 计算各模型对象总面片数，生成模型对象数组
                views::transform([&modelView, &highResolutionObjectSet, viewOrd
                    auto viewId = modelView.viewId;
                    auto& viewTypeName = modelView.viewTypeName;
                    auto& viewName = modelView.viewName;
                    auto& viewObjectList = modelView.viewObjectList;
                    auto& createdAt = modelView.createdAt;
                    auto objectTypeID = viewObject.objectTypeID;

                    const auto& meshCounts = viewObject.meshCounts;
                    auto objectMeshCount = std::accumulate(meshCounts.begin(),

                    return ModelObject{
                        .viewOrder = viewOrder,
```

```
                        .viewId = viewId,
                        .viewTypeName = viewTypeName,
                        .viewName = viewName,
                        .createdAt = timePointFromString(createdAt),
                        .objectTypeID = viewObject.objectTypeID,
                        .objectName = viewObject.name,
                        .meshCount = objectMeshCount,
                        .renderChannels = viewObject.renderChannels,
                        .onlyOneRenderChannel = std::has_single_bit(viewObject.
                    };
                });

                // 计算模型视图已占用面片数
                auto viewUsedMeshCount = std::accumulate(
                    filteredModelObjects.begin(),
                    filteredModelObjects.end(),
                    0,
                    [](int32_t prev, const auto& modelObject) { return prev + m
                );

                // 计算模型视图中的对象数量
                size_t viewObjectCount = sizeOfRange(filteredModelObjects);

                // 生成包含统计信息的模型对象数据
                return filteredModelObjects |
                    views::transform(
                        [viewUsedMeshCount, meshCount, viewObjectCount](const a
                            return ModelObject{
                                .viewOrder = incomingModelObject.viewOrder,
                                .viewId = incomingModelObject.viewId,
                                .viewTypeName = incomingModelObject.viewTypeNam
                                .viewName = incomingModelObject.viewName,
                                .createdAt = incomingModelObject.createdAt,
                                .objectTypeID = incomingModelObject.objectTypeI
                                .objectName = incomingModelObject.objectName,
                                .meshCount = incomingModelObject.meshCount,
                                .viewUsedMeshCount = viewUsedMeshCount,
                                .viewTotalMeshCount = meshCount,
                                .viewFreeMeshCount = meshCount - viewUsedMeshCo
                                .viewObjectCount = viewObjectCount,
                                .onlyOneRenderChannel = incomingModelObject.onl
                            };
                        }
                    ) |
                    to<std::vector<ModelObject>>();
            }) |
            to<std::vector<std::vector<ModelObject>>>() |
            views::join |
            to<std::vector<ModelObject>>();
l
```

代码第 48 行，我们调用了 std::has_single_bit，判断 renderChannels 是否只有一位为 1。

如果只有 1 位就说明只有一个通道，所以如果想知道一个对象包含几个通道，只需要计算有多少位 1 即可。

相比于通过集合数据结构实现集合，针对这种集合长度与元素已知而且集合元素较少的场景，使用位操作不仅节省空间，而且速度也更快。

基于传统位处理操作符实现

你有没有注意到，我在前面使用了 C++20 的 endian，has_single_bit 以及 C++23 才支持的 byteswap。

因此，我们在工程实战时，必须考虑一个问题，**那就是如果编译器不支持相关特性要怎么办？**

很简单，只要自己实现这些特性就行了。为此，我们实现了 BitCompact.h，讨论一下 C++20 中如何检测编译器特性，同时也看一下如果没有这些现成的函数，我们要如何实现。

头文件 include/ca/BitCompact.h 的实现是这样的。

复制代码

```
#pragma once

#if __has_include(<bit>)
#include <bit>
#endif // __has_include(<bit>)

#include <concepts>
#include <cstdint>

namespace std {
#if !(__cpp_lib_byteswap == 202110L)
    // 如果没有std::byteswap，使用自己实现的版本
    template <typename T>
    T byteswap(T src) {
        T dest = 0;

        for (uint8_t* pSrcByte = reinterpret_cast<uint8_t*>(&src),
            *pDestByte = reinterpret_cast<uint8_t*>(&dest) + sizeof(T) - 1;
            pSrcByte != reinterpret_cast<uint8_t*>(&src) + sizeof(T);
            ++pSrcByte, --pDestByte) {
            *pDestByte = *pSrcByte;
```

```
        }

        return dest;
    }
#endif // __cpp_lib_byteswap

#if !(__cpp_lib_endian == 201907L)
    // 如果没有std::endian，使用自己实现的版本
    enum class endian {
        little,
        big,
        // 需要根据目标体系结构判断，这里简单设置为little
        native = little
    };
#endif // __cpp_lib_endian

#if !(__cpp_lib_int_pow2 == 202002L)
    // 如果没有std::has_single_bit，使用自己实现的版本
    template <std::unsigned_integral T>
        requires (!std::same_as<T, bool> && !std::same_as<T, char> &&
        !std::same_as<T, char8_t> && !std::same_as<T, char16_t> &&
        !std::same_as<T, char32_t> && !std::same_as<T, wchar_t>)
        constexpr bool has_single_bit(T x) noexcept
    {
        return x != 0 && (x & (x - 1)) == 0;
    }
#endif // __cpp_lib_int_pow2
}
```

其实，这段代码对标准库特性兼容性非常重要。

由于 C++23 是“更好的 C++20”，但是考虑到编译器对新标准的支持进度，我们往往需要一些编程技巧来提高代码的兼容性。

为了在现代 C++ 编程环境下兼容不同版本的 C++，就要用到前面的这些预处理指令。建议你反复品味一下，说不定以后就能在自己的工作里用上。

__has_include 是 C++17 中引入的预处理指令，可以在预处理指令中判断头文件的检索路径是否存在特定的头文件。

C++20 还引入了针对语言特性和库特性的检测宏，代码中的 **__cpp_lib_byteswap**、**__cpp_lib_endian** 和 **__cpp_lib_int_pow2** 分别用于检测编译器是否支持特定的标准库函数。我们可以利用这些特性确定是否使用自定义版本的函数。

枚举类型 endian 的定义比较简单，我们将 native 强制设置为 little，实际情况下需要根据不同编译器和目标体系结构的定义来设置这个值。

函数 has_single_bit 的实现比较巧妙。这个函数用于判断二进制位串是否只有一个 1，也就是判断数字是否为 2 的幂。当 x 为 0 时返回 false 容易理解，实现的关键在于 x & (x - 1)。

这里的原理是：如果 x 只有一个 1，那么 x-1 相当于按位取反，x & (x - 1) 必定为 0。你自己尝试计算看一下，就非常容易理解了。

总结

C++20 及其后续演进标准提供的位操作库，显著改善了我们的编程效率，特别是针对序列化、反序列化以及位计算这些领域。但是，我们也看到位操作库仍然处在一个持续演进的过程（当然了），我们在实战中就使用到了 C++23 才提供的工具。

因此，如何实现兼容的 C++ 位操作库封装，其实是非常重要的。今天，在实现头文件 include/ca/BitCompact.h 的时候，我用到了一些新的预处理指令和编程技巧，实现了这种兼容性。你可以再好好回味一下最后的代码，相信这能对你的日常编程工作，添上一份力。

课后思考

我们在 include/ca/BitCompact.h 的实现里定义了枚举类型 endian，用来兼容不同版本的 C++ 和编译器。但是 native 这一枚举值需要在编译时计算得出。

那么，你能否给出这段发生在“编译时”的代码，自动计算出不同编译器和目标体系结构的值？

欢迎把你的方案贴出来，与大家一起分享。我们一同交流。下一讲见！

上一篇 16丨Bit library（一）：如何利用新bit操作库释放编程生产力？

下一篇 18丨其他重要标准库特性：还有哪些库变更值得关注？

精选留言 (1)

写留言

peter

2023-03-02 来自北京

请问：C++20的位操作比以前版本的性能有多少提升？

作者回复: 位操作性能上没有提升，只是不需要自己实现了。