PBRTv4-系列3-PBRT的整体理解

Dezeming Family

2023年9月7日

DezemingFamily系列文章和电子文档**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了DezemingFamily的系列文章，可以从我们的网站[https://dezeming.top/]找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

2023-09-12：完成第一版。

目录

一　前言

TaggedPointer利用的机制就是：因为现代CPU寻址是64位，而实际指针分配最多是57个位，剩下还有7个最高位都是0。这7个位可以用来指定类型。

因此，在构造TaggedPointer模板对象时使用一个64位的bits变量来存放信息，57个位用于表示实际的指针地址，然后剩下7个位标记是模板参数类们中的第几个类。

二　Tagged指针

TaggedPointer类是pbrt如何处理多态类型的核心。它将指针指向已知类型的对象，并在指针中使用多余的位来编码对象的实际类型（即标记(tag)它）。当需要动态调度或其他类型特定的操作时，可以从指针中提取对象的类型。这个类的实现在util/taggedptr.h文件中。

TaggedPointer是一个模板类，它要求在编译时提供它可能表示的所有类型，注意，这种方法因此排除了新类型的额外类定义的运行时加载，就像基于虚函数的多态性的通常方法一样。

下面的结构记录类型，可以统计指针记录的类型数：

```
using Types = TypePack<Ts...>;
```

现代处理器普遍使用64位指针，可以寻址2^{64}字节的内存。几十到几百GB的内存大小现在很常见，这与64位指针所能寻址的数十亿GB相去甚远。因此，处理器以较小的位数指定其可寻址存储器空间的大小。直到最近，48位地址空间在CPU上还是很常见的，尽管最近已经增加到57位。虽然单个系统拥有2^{57}字节的RAM仍然是不可想象的，但大地址空间对于集群计算非常有用，在集群计算中，许多机器都有一个统一的地址空间，或者用于将指针映射到离线存储中的数据。

TaggedPointer获取指针的高位（寻址RAM以外的剩下的7个位），以便对类型进行编码。即使有57位的地址空间，仍然有7位，这允许2^7种类型，远远超过pbrt的需求。

```
static constexpr int tagShift = 57;
static constexpr int tagBits = 64 - tagShift;
```

tagMask是提取类型标记位的位掩码（最高的7位都是1，其他位都是0），ptrMask提取原始指针。

```
static constexpr uint64_t tagMask = ((1ull << tagBits) - 1) << tagShift;
static constexpr uint64_t ptrMask = ~tagMask;
```

我们现在可以实现主TaggedPointer构造函数。给定已知类型T的指针，它使用TypeIndex()方法为其类型获取整数索引。位成员是通过将原始指针与整数类型组合来设置的，向上移位到指针值的未使用位中。

```
template <typename T>
PBRT_CPU_GPU TaggedPointer(T *ptr) {
    uint64_t iptr = reinterpret_cast<uint64_t>(ptr);
    DCHECK_EQ(iptr & ptrMask, iptr);
    constexpr unsigned int type = TypeIndex<T>();
    bits = iptr | ((uint64_t)type << tagShift);
}
```

bits就是这64个位中为1的位有那些。注意最高的7个位其实就构成了type值。

TypeIndex()方法的大部分工作都是由IndexOf结构完成的。然而，还需要一个索引来表示一个空指针，因此使用了一个0的索引，其余的索引都加1。

```
template <typename T>
static constexpr unsigned int TypeIndex() {
```

```
    using Tp = typename std::remove_cv_t<T>;
    if constexpr (std::is_same_v<Tp, std::nullptr_t>) return 0;
    else return 1 + pbrt::IndexOf<Tp, Types>::count;
}
```

std::remove_cv模板用于获取类型T，且类型不带const和volatile限定。

Tag()通过提取相关位来返回TaggedPointer的标记。反过来，Is()方法在运行时检查TaggedPointer是否表示特定类型：

```
unsigned int Tag() const { return ((bits & tagMask) >> tagShift); }
template <typename T>
bool Is() const {
    return Tag() == TypeIndex<T>();
}
```

标记的最大值等于表示的类型数。

指定类型的指针由CastOrNullptr()返回，顾名思义，如果TaggedPointer实际上不包含T类型的对象，它将返回nullptr。除了此方法之外，TaggedPPointer还提供了一个返回常量T*的常量变量，以及总是返回给定类型指针的不安全的Cast()方法。只有当TaggedPointer所持有的底层类型没有问题时，才应该使用这些类型。

```
template <typename T>
const T *CastOrNullptr() const {
    if (Is<T>())
        return reinterpret_cast<const T *>(ptr());
    else
        return nullptr;
}
```

对于需要原始指针但void指针就足够的情况，ptr()方法是可用的。

```
void *ptr() { return reinterpret_cast<void *>(bits & ptrMask); }
```

最有趣的TaggedPointer方法是Dispatch()，它是pbrt多态类型动态调度机制的核心。它的任务是确定TaggedPointer指向哪种类型的对象，然后调用所提供的函数，将对象的指针传递给它，并强制转换为正确的类型。（Spectrum::operator()方法调用TaggedPointer::Dispatch()；提供给Dispatch()的函数的操作细节将与其实现在后面一起讨论。）

大部分工作都是由在 detail 命名空间中定义的独立Dispatch()函数完成的，这意味着尽管它们是在头文件中定义的，但它们不应该被头之外的代码使用。这些函数需要所提供函数的返回类型，该类型由ReturnType帮助器模板(TaggedPointer Helper Templates)确定。我们不会在此处包含ReturnType的实现；当用TaggedPointer可以容纳的每个类型调用时，它使用C++模板包扩展来查找func的返回类型，如果它们不完全相同，则会发出编译时错误，并通过其类型定义提供返回类型。

```
template <typename F>
PBRT_CPU_GPU decltype(auto) Dispatch(F &&func) {
    DCHECK(ptr());
    using R = typename detail::ReturnType<F, Ts...>::type;
    return detail::Dispatch<F, R, Ts...>(func, ptr(), Tag() - 1);
}
```

detail::Dispatch()可以使用任意数量的类型进行调用，具体取决于TaggedPointer管理的类型数量。这是通过为不同数量的此类类型提供大量模板专门化来处理的。

在这个版本的pbrt开发的早期，我们实现了一种应用二进制搜索的调度机制，根据类型索引进行一系列递归函数调用，直到找到相应的类型。这与这里实现的方法具有同等的性能，并且需要更少的代码行。然而，我们发现它扰乱了调用堆栈，这在调试时很麻烦。在当前的方法中，动态调度只强制执行单个函数调用。

作为Dispatch()函数的一个示例，这里是处理三种类型的函数的实现；它由回调函数F的类型及其返回类型R参数化。它只需要一个switch语句，根据从TaggedPointer::Dispatch()传入的索引，用适当的指针类型调用函数。

```
template <typename F, typename R, typename T0, typename T1, typename T2>
PBRT_CPU_GPU R Dispatch(F &&func, void *ptr, int index) {
    switch (index) {
    case 0: return func((T0 *)ptr);
    case 1: return func((T1 *)ptr);
    default: return func((T2 *)ptr);
    }
}
```

Dispatch和DispatchCPU可以接受任意数量的模板参数，当类型模板参数少于或等于8个（即T0到T7）时，函数都是一个一个写的，大于8个则是递归形态。这些函数可以在文件中找到，占据了很多行代码。

TaggedPointer还包括一个const限定的分派方法以及DispatchCPU()，这对于只能在CPU上运行的方法来说是必要的。（默认的Dispatch()方法要求该方法可以从CPU或GPU代码中调用，这是pbrt中最常见的用例。）这两者在detail命名空间中都有相应的dispatch函数。

三　一个例子：光谱类

C++中的典型实现是由Spectrum中的纯虚拟方法指定这样的接口，并且Spectrum实现从Spectrum继承并实现这些方法。

而pbrt v4有些不同，与其他基于TaggedPointer的类一样，Spectrum定义了一个必须由所有频谱表示实现的接口。使用TaggedPointer方法，接口是隐式指定的：对于接口中的每个方法，Spectrum中都有一个方法将调用分派到适当类型的实现。

```
class Spectrum : public TaggedPointer<
    ConstantSpectrum, DenselySampledSpectrum,
    PiecewiseLinearSpectrum, RGBAlbedoSpectrum,
    RGBUnboundedSpectrum, RGBIlluminantSpectrum,
    BlackbodySpectrum> {
    ...
}
```

我们将在这里讨论这如何适用于单个方法的细节，但对于其他Spectrum方法和其他接口类，我们将省略它们，因为它们都遵循相同的模板。Spectrum定义的最重要的方法是operator()，它采用单个波长λ并返回该波长的光谱分布值。

```
Float operator()(Float lambda) const;
```

对TaggedPointer::Dispatch()的调用开始分派方法调用的过程。TaggedPointer类存储一个整数标记以及对其类型进行编码的对象指针；反过来，Dispatch()能够在运行时确定指针的特定类型。然后，它调用提供给它的回调函数，该函数带有指向对象的指针，并强制转换为指向其实际类型的指针。

这里调用的lambda函数op为其参数获取一个带有自动类型说明符的指针。在C++17中，这样的lambda函数充当模板化函数；使用具体类型对其进行的调用充当采用该类型的lambda的实例化。因此，lambda主体中的调用(*ptr)(lambda)最终成为对适当方法的直接调用。

```
inline Float Spectrum::operator()(Float lambda) const {
    auto op = [&](auto ptr) { return (*ptr)(lambda); };
    return Dispatch(op);
}
```

这里的Dispatch就是TaggedPointer::Dispatch()：

```
template <typename F>
PBRT_CPU_GPU decltype(auto) Dispatch(F &&func) const {
    using R = typename detail::ReturnType<F, Ts...>::type;
    return detail::Dispatch<F, R, Ts...>(func, ptr(), Tag() - 1);
}
```

&&是右值引用，主要是为了提高参数传递和赋值的效率。注意Ts就是TaggedPointer模板的参数列表。因为模板参数有7个，所以：

```
template <typename F, typename R,
typename T0, typename T1, typename T2, typename T3,
typename T4, typename T5, typename T6, typename T7>
PBRT_CPU_GPU R Dispatch(F &&func, const void *ptr, int index) {
    switch (index) {
    case 0:
        return func((const T0 *)ptr);
    case 1:
        return func((const T1 *)ptr);
    case 2:
        return func((const T2 *)ptr);
    case 3:
        return func((const T3 *)ptr);
    case 4:
        return func((const T4 *)ptr);
    case 5:
        return func((const T5 *)ptr);
    case 6:
        return func((const T6 *)ptr);
    default:
        return func((const T7 *)ptr);
    }
}
```

可以理解为，内部实现机制就是给出一个函数类型，然后基于Dispatch来选择执行该函数的类。所有TaggedPointer模板的参数列表中的类都应该有下面的函数的实现，否则就会报错：

```
Float operator()(Float lambda) const
```

四　使用分派指针

随便找一个使用的例子：

```
Spectrum s = alloc.new_object<RGBAlbedoSpectrum>(colorspace, rgb);
```

该赋值会调用下面的构造函数：

```
template <typename T>
PBRT_CPU_GPU TaggedPointer(T *ptr) {
    uint64_t iptr = reinterpret_cast<uint64_t>(ptr);
    DCHECK_EQ(iptr & ptrMask, iptr);
    constexpr unsigned int type = TypeIndex<T>();
    bits = iptr | ((uint64_t)type << tagShift);
}
```

TaggedPointer之间的赋值方法其实就是把保存对象指针以及类型的bits进行赋值：

```
PBRT_CPU_GPU
TaggedPointer &operator=(const TaggedPointer &t) {
    bits = t.bits;
    return *this;
}
```

4 1　一个小例子

我们写一个小程序来验证一下。首先定义一个模板（这个模板不实现TaggedPointer的具体功能，而是测试构造函数和赋值）：

```
#include <iostream>
template <typename... Ts>
class Tag {
public:
    Tag() {
        std::cout << "Tag()" << std::endl;
    }
    template <typename T>
    Tag(T * ptr) {
        uint64_t iptr = reinterpret_cast<uint64_t>(ptr);
        bits = iptr;
        std::cout << "Tag(T*ptr)" << std::endl;
    }
    Tag& operator=(const Tag& t) {
        bits = t.bits;
        std::cout << "operator=(const Tag&t)" << std::endl;
        return *this;
    }
    uint64_t bits;
};
```

定义两个类：

```
class A {
public:
    A() = default;
};
class B {
public:
    B() = default;
};
```

定义有这两个类的模板参数的Tag类：

```
class Spectrum : public Tag<A, B> {
public:
    using Tag::Tag;
};
```

注意using这行代码是必须要有的，这句代码的意义是明确指出派生类要继承基类的构造函数。

主函数调用：

```
int main() {
    A* a = new A();

    Spectrum ap1 = a;

    Spectrum ap2;
    ap1 = ap2;

    std::cout << "Hello World!\n";
}
```

打印结果如下：

```
Tag(T * ptr)
Tag()
operator=(const Tag& t)
Hello World!
```

五　小结

本文简单介绍了TaggedPointer类的实现，以及一些小例子加深理解。TaggedPointer类是PBRT V4中一个非常重要的概念。PBRT V1和Mitsuba V1版本都是基于插件开发的；PBRT V2和V3都是基于抽象类继承实现的多态性；而PBRT V4使用的则是分派指针类型。

等后面有时间，我们可能还会在本文中补充一些关于SameType/ReturnType以及Dispatch等相关的内容。

参考文献

[1] https://github.com/mmp/pbrt-v4

[2] https://pbrt.org/

[3] https://pbrt.org/resources

[4] Pharr, Matt, Wenzel Jakob, and Greg Humphreys. Physically based rendering: From theory to implementation. MIT Press, 2023.

[5] https://www.cnblogs.com/pointer-smq/p/7522416.html