boost::typeIndex 的相关探究 https://zhuanlan.zhihu.com/p/165628422

Providence
腾讯 后台开发

+ 关注他

boost::typeIndex 的相关探究

Effective Modern C++ 的 Item 4: Know how to view deduced types. 中提到了 `Boost::typeindex` 的使用，但并没有讲到其实现原理。

1. typeid 操作符

typeid 是 C++ 中的一个操作符，可以用于获取类型的信息，常常用在必须知道多态对象的动态类型，或是识别静态类型的地方。

我们可以写一个简单的 demo 用于获取对象类型相关的信息，需要包含 `tepyinfo` 头文件：

```
#include <iostream>
#include <typeinfo>

using namespace std;

class Foo {};

int main()
{
    cout << "1: " << typeid(1).name() << endl;
    cout << "int: " << typeid(int).name() << endl; // 和 sizeof 操作符类似，typeid 也可以

    cout << "typeid: " << typeid(typeid(int)).name() << endl;
    cout << "typeid: " << typeid(const type_info &).name() << endl;

    const Foo *foo = new Foo();
    cout << "foo: " << typeid(foo).name() << endl;
    cout << "*foo: " << typeid(*foo).name() << endl;
    cout << "Foo: " << typeid(Foo).name() << endl;
}
[joelzychen@DevCloud ~/typeid]$ g++ -std=c++11 -otypeid_test typeid_test.cpp
[joelzychen@DevCloud ~/typeid]$ ./typeid_test
1: i
int: i
typeid: N10__cxxabiv123__fundamental_type_infoE
typeid: St9type_info
foo: PK3Foo
*foo: 3Foo
Foo: 3Foo
```

`std::type_info::name()` 函数返回的字符串中，在 GCC 和 Clang 的实现里一般 i 代表 int，P 代表 pointer，K 代表 const，数字用于标识其后跟随了几个字符；我们可以将这段代码使用微软的 MSVC 编译运行，得到更加直观的输出：

```
1: int
int: int
typeid: class type_info
typeid: class type_info
foo: class Foo const *
*foo: class Foo
Foo: class Foo
```

可以看到大多数结果都与我们的预期相符，但在调用 `typeid(const type_info &).name()` 返回的结果却不是我们所期望的 `const type_info &`，其中的 const 和 reference 特性并没有得到保留；再举一个简单的例子：

```
#include <iostream>
#include <typeinfo>

using namespace std;

template<typename T>
static void PrintType(const T &t)
{
    std::cout << "T: " << typeid(T).name() << std::endl;
    std::cout << "t: " << typeid(t).name() << std::endl;
}

int main()
{
    const int *p_i;
    PrintType(p_i);
}
[joelzychen@DevCloud ~/typeid]$ g++ -std=c++11 -otypeid_test typeid_test.cpp
[joelzychen@DevCloud ~/typeid]$ ./typeid_test
T: PKi
t: PKi
```

`PrintType` 这个模板接收到的 `T` 的确是 `PKi` （`const int*`）类型，但和之前的例子类似，`t` 的 `const reference` 特性并没有得到保留

2. 使用 boost::typeindex::type_id_with_cvr 代替 typeid

`boost` 库中有一个类似于 `typeid` 操作符的函数 `boost::typeindex::type_id_with_cvr` 可以用于获取对象类型，我们可以利用这个模板函数来获取更精确的类型：

```
#include <iostream>
#include <typeinfo>
#include <boost/type_index.hpp>

using namespace std;

template<typename T>
static void PrintType(const T &t)
{
    cout << "T: " << boost::typeindex::type_id_with_cvr<T>().pretty_name() << endl;
    cout << "t: " << boost::typeindex::type_id_with_cvr<decltype(t)>().pretty_name() <
    cout << "typeid: " << boost::typeindex::type_id_with_cvr<decltype(typeid(int))>().
}

int main()
{
    const int *p_i{ nullptr };
    PrintType(p_i);
}
[joelzychen@DevCloud ~/typeid]$ g++ -std=c++11 -otypeid_test typeid_test.cpp -I/usr/in
[joelzychen@DevCloud ~/typeid]$ ./typeid_test
T: int const*
t: int const* const&
typeid: std::type_info const&
```

可以看到 `typeid` 真正的返回值类型是 `std::type_info const&`，`boost::typeindex::type_id_with_cvr` 通过某种机制保留了其 `const` 和 `reference` 的特性并通过 pretty_name() 函数将结果转换成了字符串进行输出；和 `typeid` 操作符不同的是，`type_id_with_cvr` 函数只能接收模板参数类型或通过 `decltype` 推导出的类型，而不能接收一个变量。

3 type_id_with_cvr() 的实现

`type_id_with_cvr` 这个模板函数定义在 `boost/type_index.hpp` 中，它实际上是调用了 `stl_type_index` 类的静态模板函数 `type_id_with_cvr`：

```
// boost/type_index.hpp
namespace boost { namespace typeindex {
template <class T>
inline type_index type_id_with_cvr() BOOST_NOEXCEPT {
    return type_index::type_id_with_cvr<T>();
}
}

// boost/type_index/stl_type_index.hpp
namespace boost {
class stl_type_index : public type_index_facade<stl_type_index, std::type_info> // 省略
{
public:
    typedef std::type_info type_info_t; // 省略了 BOOST_NO_STD_TYPEINFO 宏的判断
private:
    const type_info_t* data_;
public:
    inline stl_type_index(const type_info_t& data) BOOST_NOEXCEPT
        : data_(&data) // 利用 typeid 操作符返回的 const type_info_t& 对象进行构造
    {}

    template <class T>
    inline static stl_type_index type_id_with_cvr() BOOST_NOEXCEPT;
}
}
```

`boost::typeindex::type_id_with_cvr` 函数将其第二个模板参数 `detail::cvr_saver<T>` 作为实参调用了 `typeid` 操作符，并利用返回的 `const type_info_t&` 对象构造了 `stl_type_index` 对象；

`detail::cvr_saver` 是一个空的模板类，只带有模板参数 `<class T>` 的信息，可以利用 `typeid` 来获取这个特例化模板类的 `type_info`。

```
// boost/type_index/stl_type_index.hpp
namespace boost {
template <class T>
inline stl_type_index stl_type_index::type_id_with_cvr() BOOST_NOEXCEPT {
    typedef BOOST_DEDUCED_TYPENAME boost::conditional<
        boost::is_reference<T>::value ||  boost::is_const<T>::value || boost::is_volat
        detail::cvr_saver<T>,
        T
    >::type type; // 等价于 using type = boost::conditional<...>

    return typeid(type);
}
}

// boost/type_traits/conditional.hpp
namespace detail {
    template <class T> class cvr_saver{};
}

namespace boost {
template <bool b, class T, class U> struct conditional { typedef T type; };
}
```

4 class stl_type_facade

`class type_index_facade` 是 `class stl_type_index` 的基类，其源码在 `type_index_facade.hpp` 文件中，使用了门面模式的设计：

```
// boost/type_index/stl_type_index.hpp
// 将派生类 Derived 作为模板参数
template <class Derived, class TypeInfo>
class type_index_facade {
public:
    typedef TypeInfo                                type_info_t;

    // 调用子类的 raw_name()，没有使用虚函数的方式，而是利用模板实现了静态多态
    inline const char* name() const BOOST_NOEXCEPT {
        return derived().raw_name();
    }

    // 返回 human-readable 的字符串，调用子类的 name()
    inline std::string pretty_name() const {
        return derived().name();
    }

    // 比较派生类的 raw_name()，需要派生类实现 raw_name() 函数
    inline bool equal(const Derived& rhs) const BOOST_NOEXCEPT {
        const char* const left = derived().raw_name();
        const char* const right = rhs.raw_name();
        return left == right || !std::strcmp(left, right);
    }

    // 比较派生类的 raw_name()，需要派生类实现 raw_name() 函数
    inline bool before(const Derived& rhs) const BOOST_NOEXCEPT {
        const char* const left = derived().raw_name();
        const char* const right = rhs.raw_name();
        return left != right && std::strcmp(left, right) < 0;
    }

    // 获取一个类型的哈希值，默认对派生类的 raw_name() 进行哈希
    inline std::size_t hash_code() const BOOST_NOEXCEPT {
        const char* const name_raw = derived().raw_name();
        return boost::hash_range(name_raw, name_raw + std::strlen(name_raw));
    }
}
```

除此之外，`class type_index_facade` 基类还重载了各类对比操作符，输出流操作符和类的哈希值算法：

```
// boost/type_index/stl_type_index.hpp
// 省略了其它类型的对比操作符
template <class Derived, class TypeInfo>
inline bool operator == (const TypeInfo& lhs, const type_index_facade<Derived, TypeInf
    return Derived(lhs) == rhs; // 需要派生类实现以 const TypeInfo& 作为参数的构造函数
}

// 重载输出流操作符
template <class CharT, class TriatT, class Derived, class TypeInfo>
inline std::basic_ostream<CharT, TriatT>& operator<<(
    std::basic_ostream<CharT, TriatT>& ostr,
    const type_index_facade<Derived, TypeInfo>& ind)
{
    ostr << static_cast<Derived const&>(ind).pretty_name();
    return ostr;
}

// 类的哈希值算法
template <class Derived, class TypeInfo>
inline std::size_t hash_value(const type_index_facade<Derived, TypeInfo>& lhs) BOOST_N
    return static_cast<Derived const&>(lhs).hash_code();
}
```

如果想要进行 `class type_index_facade` 基类的全部操作，还需要派生类至少实现至少以下两个函数：

1. `raw_name()`，基类的很多函数都依赖于派生类的这个函数
2. `Derived(const TypeInfo&)`，即以 `const TypeInfo&` 作为参数的构造函数，用于与 `TypeInfo` 对象进行对比

5 class type_type_index

`stl_type_index` 是 `stl_type_facade` 的派生类，它的私有成员变量的类型 `type_info_t` 是通过 `typedef` 定义出来的，`BOOST_NO_STD_TYPEINFO` 是意义是编译器的 `namespace std` 下没有 `type_info` 这个类型，这时会将全局命名空间的 `type_info` 定义为 `type_info_t`。

```
public:
#ifdef BOOST_NO_STD_TYPEINFO
    typedef type_info type_info_t;
#else
    typedef std::type_info type_info_t;
#endif

private:
    const type_info_t* data_;
```

为了方便理解暂时省略 `BOOST_NO_STD_TYPEINFO` 宏的定义；派生类 `stl_type_index` 的声明大致如下：

```
class stl_type_index : public type_index_facade<stl_type_index, std::type_info>
{
public:
    typedef std::type_info type_info_t;

private:
    const type_info_t* data_; // 唯一的私有成员 const type_info_t*

public:
    inline stl_type_index() BOOST_NOEXCEPT
        : data_(&typeid(void))
    {}

    inline stl_type_index(const type_info_t& data) BOOST_NOEXCEPT
        : data_(&data) // 以 const TypeInfo& 作为参数的构造函数，对比操作符和 type_id_with
    {}

    inline const type_info_t&  type_info() const BOOST_NOEXCEPT; // 获取私有成员数据

    inline const char*  raw_name() const BOOST_NOEXCEPT; // raw_name() 函数
    inline const char*  name() const BOOST_NOEXCEPT;
    inline std::string  pretty_name() const;

    inline std::size_t  hash_code() const BOOST_NOEXCEPT;
    inline bool         equal(const stl_type_index& rhs) const BOOST_NOEXCEPT;
    inline bool         before(const stl_type_index& rhs) const BOOST_NOEXCEPT;

    template <class T>
    inline static stl_type_index type_id() BOOST_NOEXCEPT;

    template <class T>
    inline static stl_type_index type_id_with_cvr() BOOST_NOEXCEPT;

    template <class T>
    inline static stl_type_index type_id_runtime(const T& value) BOOST_NOEXCEPT;
};
```

派生类 `stl_type_index` 的 `equal`，`before`，`hash_code` 的实现都和基类类似，操作的对象都是 `raw_name()`：

```
inline std::size_t stl_type_index::hash_code() const BOOST_NOEXCEPT {
#ifdef BOOST_TYPE_INDEX_STD_TYPE_INDEX_HAS_HASH_CODE
    return data_->hash_code();
#else
    return boost::hash_range(raw_name(), raw_name() + std::strlen(raw_name()));
#endif
}

inline bool stl_type_index::equal(const stl_type_index& rhs) const BOOST_NOEXCEPT {
#ifdef BOOST_TYPE_INDEX_CLASSINFO_COMPARE_BY_NAMES
    return raw_name() == rhs.raw_name() || !std::strcmp(raw_name(), rhs.raw_name());
#else
    return !!(*data_ == *rhs.data_);
#endif
}

inline bool stl_type_index::before(const stl_type_index& rhs) const BOOST_NOEXCEPT {
#ifdef BOOST_TYPE_INDEX_CLASSINFO_COMPARE_BY_NAMES
    return raw_name() != rhs.raw_name() && std::strcmp(raw_name(), rhs.raw_name()) < 0
#else
    return !!data_->before(*rhs.data_);
#endif
}
```

`name()` 和 `raw_name()` 都调用了私有成员的 `name()` 函数，即 `std::type_info::name()`：

```
inline const char* stl_type_index::raw_name() const BOOST_NOEXCEPT {
#ifdef _MSC_VER // 不同编译器对 typeid 的实现不同，因此 boost 库在进行封装时同时实现了 raw_n
    return data_->raw_name();
#else
    return data_->name();
#endif
}

inline const char* stl_type_index::name() const BOOST_NOEXCEPT {
    return data_->name();
}
```

在第 2 部分中调用的 `pretty_name()` 函数原型如下：

```
inline std::string stl_type_index::pretty_name() const {
    static const char cvr_saver_name[] = "boost::typeindex::detail::cvr_saver<";
    static BOOST_CONSTEXPR_OR_CONST std::string::size_type cvr_saver_name_len = sizeof

    // 对于 GCC 和 CLang，demangled_name 函数会去执行解码操作；而对于 MSVC，因为通过 std::ty
    const boost::core::scoped_demangled_name demangled_name(data_->name());

    // begin 是通过 demangled_name.get() 获取到的 svr_saver 类型对象的全文，用 GDB 断点到此
    // (gdb) p begin
    // $1 = 0x605010 "boost::typeindex::detail::cvr_saver<int const> ()"
    const char* begin = demangled_name.get();
    if (!begin) {
        boost::throw_exception(std::runtime_error("Type name demangling failed"));
    }

    const std::string::size_type len = std::strlen(begin);
    const char* end = begin + len;

    // 字符串对比，裁剪两边多余的字符
    if (len > cvr_saver_name_len) {
        const char* b = std::strstr(begin, cvr_saver_name);
        if (b) {
            b += cvr_saver_name_len;

            // Trim leading spaces
            while (*b == ' ') {         // the string is zero terminated, we won't exc
                ++ b;
            }

            // Skip the closing angle bracket
            const char* e = end - 1;
            while (e > b && *e != '>') {
                -- e;
            }

            // Trim trailing spaces
            while (e > b && *(e - 1) == ' ') {
                -- e;
            }

            if (b < e) {
                // Parsing seems to have succeeded, the type name is not empty
                begin = b;
                end = e;
            }
        }
    }

    return std::string(begin, end);
}
```

至此就了解了除了 `demangled_name` 函数以外的所有实现细节了，不难理解其实 `stl_type_index` 这个类就是对 `std::type_info` 类的封装，`type_id_with_cvr` 和 `pretty_name` 两个函数分别细化了 `typeid` 操作符和 `std::type_info::name()` 函数。

发布于 2020-07-31 20:36

C++ Boost（C++ 库） C / C++