C++反射的应用与实践

卜恪

银汉游戏 YINHAN GAMES

PureCPP

什么是反射？

- 程序对自身程序结构认知的能力
  - 能够使用数据结构来描述数据类型和其成员
  - 能够主动地遍历这些成员

反射可以做什么？

- 序列化与反序列化
- GUI库中对数据显示的自动化
- 对象关系映射(ORM)
- 与其他语言互操作
- 提供更强大的运行期特性

C++反射现状

- 没有原生的语言反射基础设施
- 只有一个在TS状态的反射提案
- 还有cppers造轮子的血泪史

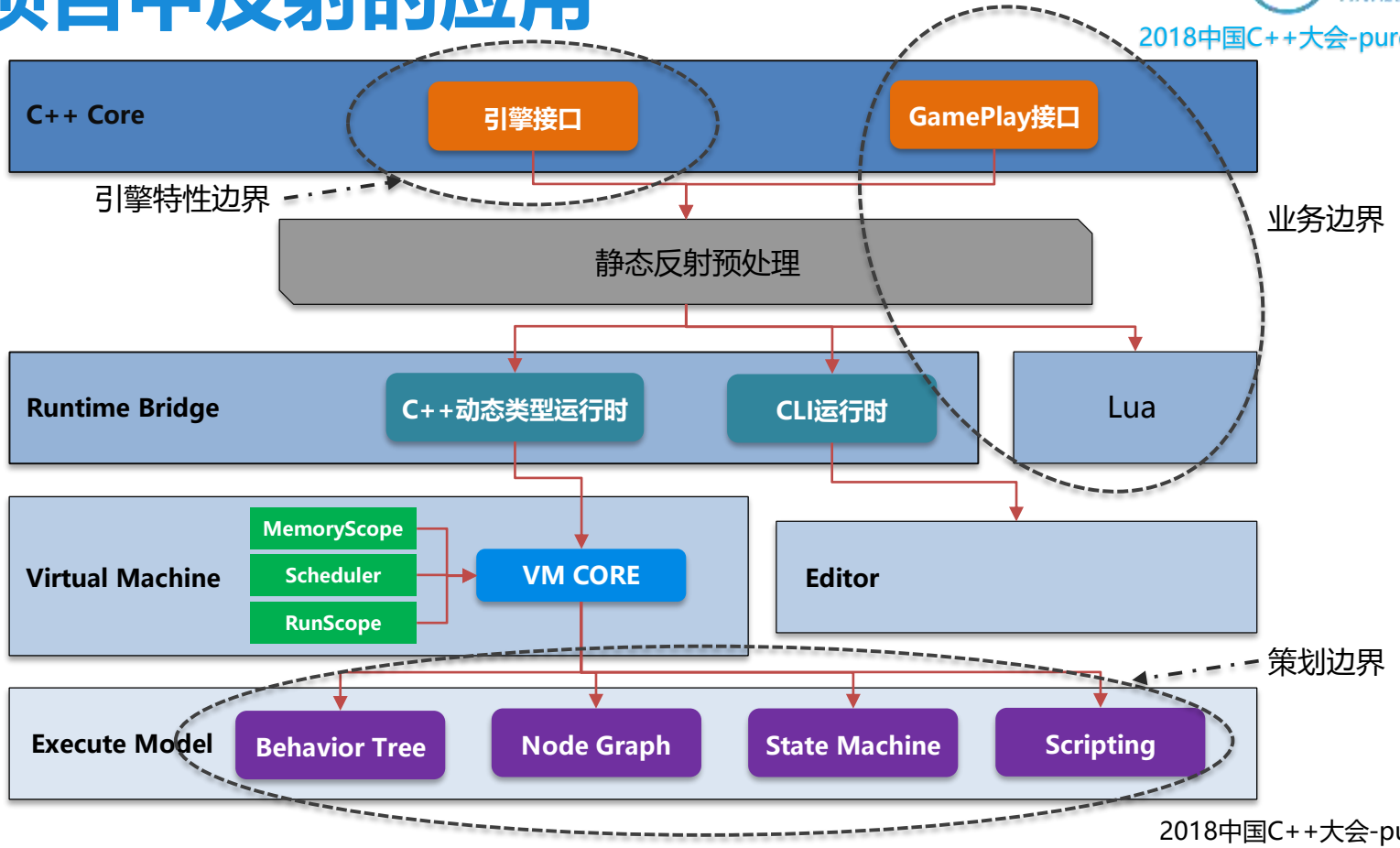
Struggle On Reflection

- 静态反射
  - 编译期反射 - 编译期的数据结构
  - 离线期反射 - 代码生成
- 动态反射 - 运行期的数据结构
- 野路子

反射元信息

- 类型的元信息
  - **成**员字段的元信息
    - 字段的参数元信息
  - **成**员方法的元信息
    - 方法名
    - 方法的形参参数元信息
    - 方法返回参数的元信息
  - **基**类的元信息
  - **参数元信息**
    - 参数名
    - 参数的类型元信息
  - **构造函数的元信息**
    - 形参参数元信息
  - More ...

项目中反射的应用


C++ Core
引擎接口
GamePlay接口
引擎特性边界
业务边界
静态反射预处理
Runtime Bridge
C++动态类型运行时
CLI运行时
Lua
Virtual Machine
MemoryScope
Scheduler
RunScope
VM CORE
Editor
策划边界
Execute Model
Behavior Tree
Node Graph
State Machine
Scripting

目录

1. 静态反射
   a. 编译期与离线期的对比
   b. 离线期的实践
2. 动态反射与动态类型运行时
3. 编译期反射
   a. 现有设施下的实践
   b. 未来的新标准

静态反射 - 编译期

- 标记方式
  - 侵入式
    - 能够处理私有成员
    - 侵入导致对类型的修改
    - 处于class-scope
  - 非侵入式
    - 只能处理公有成员
    - 无需修改需要标记的用户类型
    - 处于namespace-scope

静态反射 - 编译期

- 数据访问
  - 等价tuple
    - 类型
    - 与反射类型大小及对其属性相同
  - 指向成员的指针（pointer to member data）的列表
    - 编译期常量
    - 无需考虑大小及对其属性

编译期反射 - 遍历

- 显式地遍历访问
  - static-for using std::index_sequence
- visitor模式
  - 类似visitor访问boost.variant

静态反射 - 离线期

- 在编译前解析程序的AST
  - 通常仅解析接口头文件
- 遍历AST生成结构化数据
  - AST的结构不利于数据的访问
  - 提取类型和模板的引用关系
- Code Generation

编译期反射

- 语法和语义都完整
- 受限于C++编译器的特性
- 需要实现额外的编译期代码的中间层

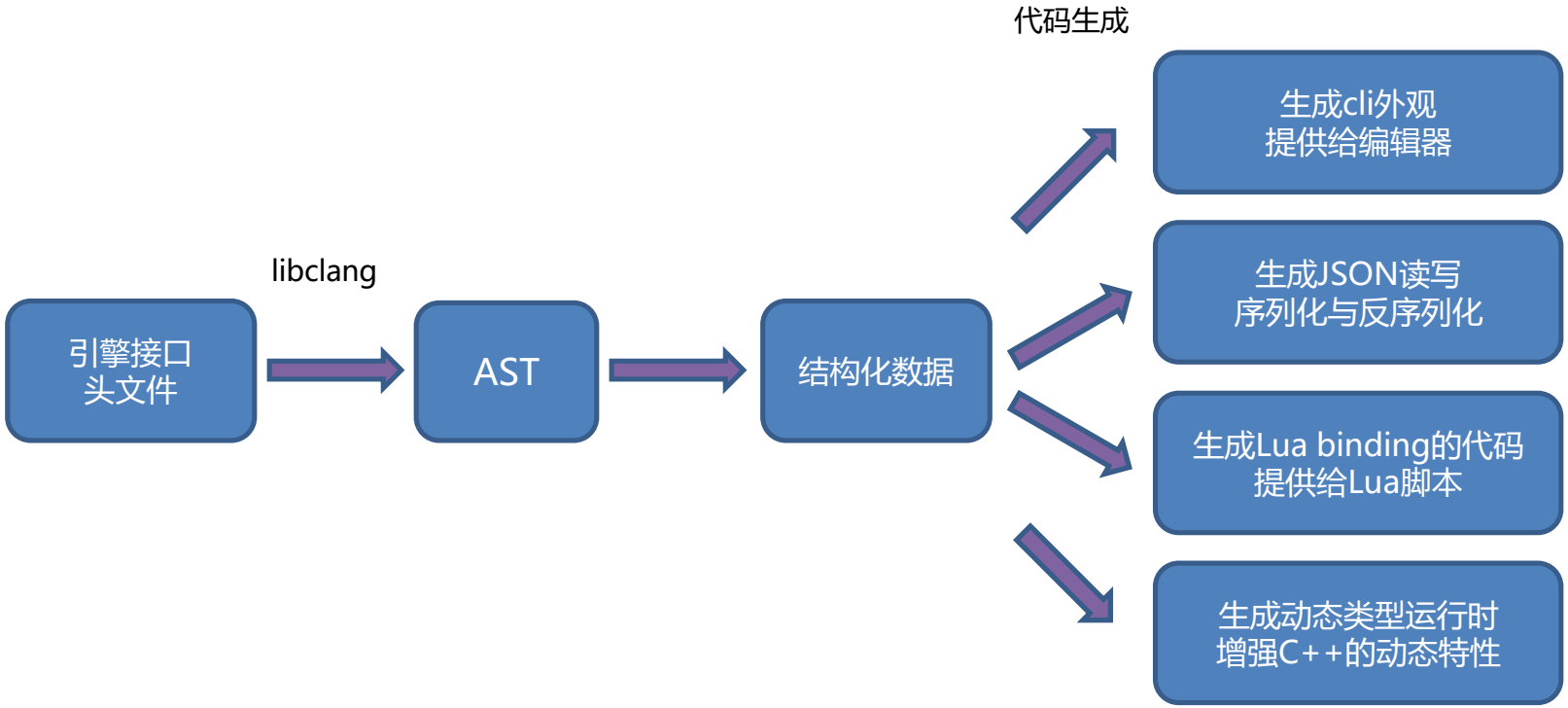
离线期反射

- 许多语法和语义丢失
- **独立之外的**预处理程序，**能**够实现复杂的需求
- **需要生成**额外的代码

静态反射 - 选型

- 静态反射承担引擎接口到业务逻辑层的导出工作
- 编译期反射由于语言设施还不够完善，有需求无法满足
- 编译期将引入大量的标记代码
- 编译期将引入大量而不可控的模板元代码
- 选择以离线期反射作为静态反射的方案

项目的离线期工具clangen



项目的离线期工具clangen

- Libclang解析接口头文件
  - 将Libclang导入到C#
  - ClangSharp自更新
- 遍历Libclang AST生成结构化数据
- 模板引擎DotLiquid作代码生成

Libclang的使用

- 设置系统头文件路径
- 妥善处理C++的类型
- Libclang的一些使用限制

设置头文件路径

- 确定VCTool Chain的版本
  - Microsoft.VisualStudio.Setup.Configuration寻找VisualStudio的安装目录和配置
  - $(YourVisualStudioPath)/VC/Auxiliary/Build/Microsoft.VCToolsVersion.default.txt
- 查询注册表确定Windows Kits的安装目录
- 确定libclang include命令行参数
  - 计算$(VC_IncludePath)与$(WindowsSDK_IncludePath)绝对路径
  - 绝对路径作为-I参数

处理C++的类型

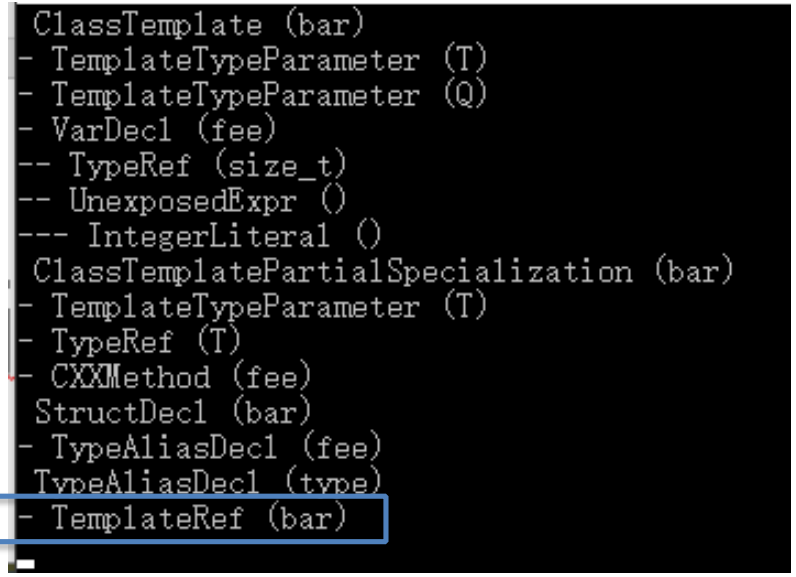
- 内建类型和抽象数据类型（ADT）
- 类型修饰符带来的衍生类型
  - const和volatile修饰符
  - 指针类型
  - 左值与右值引用
  - 数组，尤其是大小未明确的数组类型（int a[]）
  - 指向成员的函数及指向成员的数据类型
- Type define & type alias
  - namespace-scope & class-scope
  - 从基类继承而来的
  - 依赖模板参数的类型

使用限制

- Libclang可以且仅能看到源码的AST
  - 合成的函数和操作符对AST不可见
  - 类模板实例化后的实体类型的结构对AST不可见
- 模板的类型参数坍塌
  - typedef和type alias会丢失
  - std::string到basic_string<char, char_traits<char>, allocator<char>>
- 无法准确地索引到实例化类型的模板特化
  - 全特化是普通类，可以正确处理
  - 偏特化只能索引到主模板

使用限制


```cpp
template <typename T, typename Q>
struct bar {
    constexpr static size_t fee = 1;
};

template <typename T>
struct bar<T, int> {
    void fee() {}
};

template <>
struct bar<int, int> {
    using fee = void;
};

using type = bar<void, int>;
```



项目的离线期工具clangen

- 对模板的支持
  - <type_traits> on Clang AST
  - 处理的是实例化的类型，而不是处理模板
  - 处理了标准库的容器及迭代器
  - 统一处理字符串

一点小建议

- 原则上避免在模块级别的接口中使用模板
  - 为libclang的使用招致诸多麻烦
  - ABI问题和ODR问题
- 可以避免接口头文件包含系统头文件

编译期的C++

运行期的C++

运行期设施简陋

- typeof operator 和 std::type_info
- virtual table: 动态分派
- dynamic_cast: 动态类型转换

动态反射 - 需要完善的设施

- 运行时获取类型的元信息
- 通过类型的元信息操作实体对象
  - 成员变量的访问和设置
  - 成员函数的调用
  - 运行期的类型检查与类型转换
- 与静态代码解耦，消除
  - 显式的表达式
  - 显式的字面量
  - 显式的函数调用
- C++动态类型运行时

SCRIPTING！

动态反射 - PONDER


```cpp
class Person
{
public:
    // constructor
    Person(const std::string& name)
            : m_name(name)
    {}

    // accessors for private members
    std::string name() const { return m_name; }
    void setName(const std::string& name) { m_name = name; }

    // public members
    float height;
    unsigned int shoeSize;

    // member function
    bool hasBigFeet() const { return shoeSize > 10; } // U.K.!

private:
    std::string m_name;
};
```

```cpp
static void declare()
{
    using namespace ponder;
    Class::declare<Person>("Person")
        .constructor<std::string>()
        .property("name", &Person::name, &Person::setName)
        .property("height", &Person::height)
        .property("shoeSize", &Person::shoeSize)
        .function("hasBigFeet", &Person::hasBigFeet);

    // get meta info
    auto& metaclass = classByType<Person>();
    // create default instance
    UserObject person = runtime::create(metaclass, "Bozo");
    // set attributes
    person.set("height", 1.62f);
    // retrieve a function we would like to call
    const auto& func = metaclass.function("hasBigFeet");
    // call the function and get the result
    const bool bigFeet = runtime::call(func, person).to<bool>();
    runtime::destroy(person);
}
```

动态反射 - PONDER

- 注册类型的元信息到runtime type system中
- 通过元信息创建默认实例
- 对字段的Get和Set
- 通过成员方法的元信息调用方法

PONDER的局限性


```
auto& metaclass = classByType<Person>();
```

```
runtime::call(func, person).to<bool>();
```

- 依赖模板参数也就意味着依赖编译期，类型硬编码

```
person.set("height", 1.62f);
```

- 1.62f字面量表达式也属于编译期

引入类型擦除

类型擦除

- 引入公共基类
  - 额外的数据类型的枚举类型开销
  - 额外的数据实际存储开销
    - 内建数值类型用值存储
    - 字符串和ADT引用存储

```
struct object_t {
   enum class value_type {
      boolean,
      integer,
      floating_point,
      string,
      object_value,
      user_begin,
   };

   using storage_type =  union {
      bool          bool_val;
      int         int_val;
      double      float_val;
      object_t*   object_val;
   };

   value_type         type_;
   storage_type       storage_;
};
```

类型擦除

- 引入公共基类
- 成员方法类型擦除
  - data stack擦除形参列表
  - data stack擦除返回值
  - int返回函数调用结果

```
struct data_stack {
    using stack_storage_t =
        std::vector<object_t*>;
    stack_storage_t stack_storage_;
};

using method_function =
    std::function<int(data_stack*)>;

auto method = meta_bind(
    &Person::hasBigFeet);
```

类型擦除

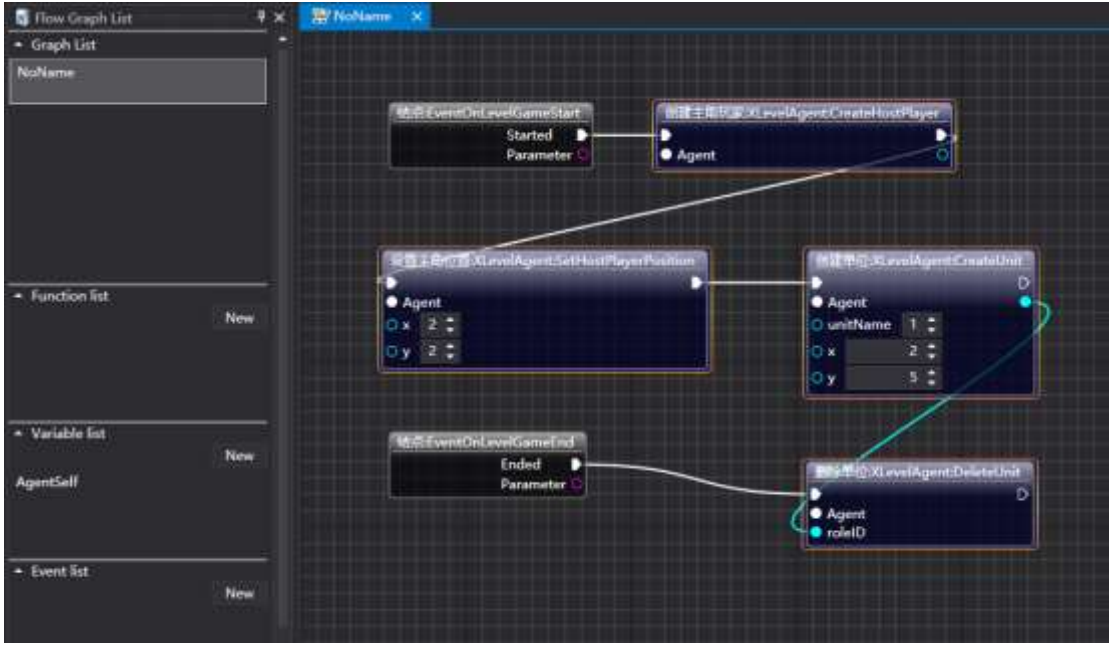
- 引入公共基类
- 成员方法类型擦除
- 成员字段Get Set类型擦除
  - 从Get和Set函数
  - 从指向成员变量的指针

```
using get_function =
   std::function<object_t*()>;

using set_funciton =
   std::function<int(object_t*)>;

auto name_set = meta_set_bind(
    &Person::setName);
auto name_get = meta_get_bind(
    &Person::name);

auto height_set = meta_set_bind(
    &Person::height);
auto height_get = meta_get_bind(
    &Person::height);
```

类型擦除

- 引入公共基类
- 成员方法类型擦除
- 成员字段Get Set类型擦除
- 字面量表达式的擦除
  - object_t序列化和反序列化的支持

动态特性用例 - NodeGraph



NodeGraph

- 用Graph组织的顺序结点执行模型
- Running on Virtual Machine
- 通过运行期类型元信息的方法信息调用C++接口
- 静态反射为编辑器自动生成C++接口的UI结点

NodeGraph - 调用


```
class ng_node_t {
public:
    // ...
    virtual int on_active(vm_coroutine_t* coroutine,
        ng_instruction_slot_t* slot) = 0;
};

class ng_meta_node_t : public ng_node_t {
public:
    // ...
    int on_active(vm_coroutine_t* coroutine,
        ng_instruction_slot_t* slot) override {
        auto* memory_scope = coroutine->get_memory_scope();
        auto* data_stack = coroutine->get_data_stack();
        for(auto& var_slot : this->input_var_slots()) {
            data_stack->push(memory_scope->get_object(var_slot->id()));
        }
        auto return_count = method_->invoke(data_stack);
        // scheduling codes ...
        return return_count;
    }
private:
    meta_method_t* method_;
};
```

类型擦除的元函数调用：

1. 遍历输入数据结点
2. 依次从局部内存中取值入栈
3. 调用函数并将结果入栈
4. 返回调用返回值参数的个数

项目反射方案小结

- 离线期静态反射
  - 为C++ Core的接口生成桥接层代码
- 动态反射
  - 支撑C++动态类型运行时

编译期反射实践 - IGUANA


```
IGUANA_REFLECT(
    test::base,
    IGUANA_CTOR((), (int), (int, int)),
    IGUANA_SDATA(sa, sb),
    IGUANA_MDATA(a, b),
    IGUANA_SFUNC(
        (bar, int(int)),
        (bar, int(float, int)),
        bee
    ),
    IGUANA_MFUNC(
        fee,
        (foo, int(int)),
        (foo, int(int, double) const)
    )
);
```

- 使用PMD序列
- 使用非侵入式
- 增加的构造函数的标记支持
- 增加了基类的标记支持
- 增加了对成员函数的支持
  - 可以处理函数重载
- 增加了对静态变量的支持
  - One Define Rule需要用户保证

IGUANA IN FEATHER

- JSON配置文件的自动解析
- 网络协议的序列化和反序列化
- C++对象与数据库的数据映射关系

编译期反射 - IGUANA

- iguana的局限性
  - 无法实现语言设施之外的功能：is_virtual, is_public, get_param_name
  - 只能处理类型而不能处理模板
  - 无法处理私有成员变量和方法
  - 复杂而庞大的标记工作，使用上不便利

能否不用显示地标记？

等待新的语言标准

编译期反射 - magic_get


```cpp
#include <boost/pfr.hpp>
#include <string>
#include <iostream>

struct foo
{
    std::string     name;
    double          power;
};

int main(void)
{
    using boost::pfr::get;
    foo f = { "kakaroto", 100000000.0 };
    std::cout << get<0, foo>(f) << std::endl;
    std::cout << get<1, foo>(f) << std::endl;
    return 0;
}

// or
```

- 不需要显式地标记成员
- 只能标记非静态公有成员变量
- 只支持aggregate类型

magic_get – More Details

- Github Repository
- Cppcon Representation
- The C++ Type Loophole (C++14)
- C++14实现编译期反射--剖析magic_get中的magic
- magic_get - A reflection techniques using modern C++

Reflection TS


```
// get the compile-time data structure for meta information
using meta_info_t = meta_info<decltype(client::foo_t)>;
```

新的关键字：reflexpr

```
using meta_info_t = reflexpr(client::foo_t);
```

Reflection TS

C++语法的全面支持

```
// alias namespace
namespace reflect = std::experimental::reflect;

// get meta info for base type
using base_meta_t = reflect::get_base_class_t<meta_info_t>;

// check if base type is virtual inherited
reflect::is_virtual_v<base_meta_t>;

// check if base type if public inherited
reflect::is_public_v<base_meta_t>;

// check if this type is final class
reflect::is_final_v<meta_info_t>;
```

Reflection TS – 访问成员

可以访问函数和数据成员，包括私有和公有

```
// get private data member
using private_data_members_t = reflect::get_private_data_members_t<meta_info_t>;

// get member functions
using member_functions_t = reflect::get_member_functions_t<meta_info_t>;

// get constructors
using ctors_t = reflect::get_constructors_t<meta_info_t>;
```

Reflection TS – 遍历

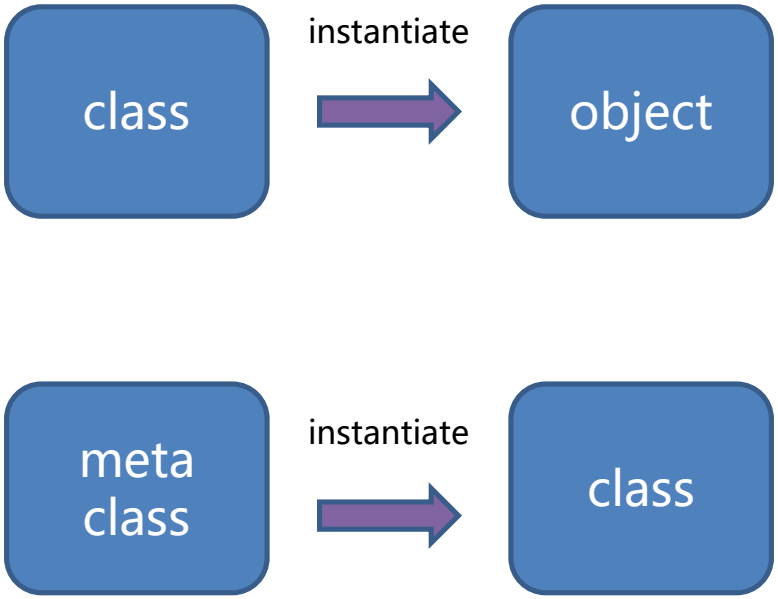
只提供access和unpack接口
遍历还需要自行实现

```
template <ObjectSequence Members, size_t ... Is>
inline void for_each_impl(std::index_sequence<Is...>)
{
    std::cout << ... <<
        reflect::get_name_v<reflect::get_element_t<Is, Members>>;
}

template <ObjectSequence Members>
inline void for_each()
{
    for_each_impl(std::make_index_sequence<reflect::get_size_v<Members>>{});
}
```

Reflection TS – More Details

- Concepts for meta-object types
- 反射对Callable Concepts的全面支持
- 反射对命名空间的支持
- 源码文件位置的支持
  - get_source_line
  - get_source_column
  - get_source_file_name

Metaclass 提案



- 编译期class实例化为运行期object
- 编译期metaclass实例化为class
- 基于Reflection提案

Q & A

谢谢

银汉游戏
YINHAN GAMES