GAMES104 现代游戏引擎：从入门到实践

作业 4 引擎工具链中反射系统的应用

一、前言：

经过两节的工具链课程学习后，同学们对游戏引擎工具链有了一定的了解，知道了工具链中一些工程架构、基本概念以及反射等知识。相信大家也非常希望熟悉反射这套在游戏引擎工具链中几乎无处不在的系统。这次作业我们会引导大家使用反射系统实现一些功能，相信同学们通过实践会进一步地了解反射系统。

本次作业作为开放性作业，我们会为同学们做好所有前期准备，搭好框架，提供功能的入口，让同学们实现具体的代码，希望带领同学们初探反射系统。而对于已经了解过反射系统的实现机制及用法的同学，也可以自行阅读理解 Piccolo 中的反射相关代码。

二、本次作业的具体内容：

1.掌握 Piccolo 中反射宏和反射标签的用法，关于反射宏和反射标签本文档第三部分会做详细说明。

2.同学们选择自己感兴趣的系统，给它管理的 Component 结构增加或修改一个属性，检查它及 Component 结构的反射宏和反射标签，确保它能够被正确的反射。(反射系统支持的原子类型可点此查看文章 B.1 节内容)

3.在 Piccolo 代码中找到 engine/source/editor/source/editor_ui.cpp，找到

createClassUI ()函数，详细阅读代码，检查通过反射系统创建 UI 面板的逻辑，确保新加的属性能被正确的显示到右侧 Components Details 面板对应的 component 中。（一般来说，基础功能不需要修改相应代码，如果想要实现更复杂的功能，同学们也可以自行对代码进行修改。）

4.基础的参数反射和控件生成完成后，想要继续挑战高难度的同学，可以尝试在对应系统中让新增或修改的属性按照自己预想的方式工作起来！比如接下来的例子中，可以让界面设置的值作为跳跃瞬间初速度控制角色跳跃高度。

例：现在的 Motor 系统中，我们通过 m_jump_height 来控制角色跳跃的高度，其定义位置在 engine/source/runtime/resource/res_type/components/motor.h 文件的 MotorComponentRes 类中。在 engine/source/runtime/function/framework/component/motor/ motor_component.cpp 中的 MotorCompont::calculatedDesiredVerticalMoveSpeed 方法内,使用该属性计算得到了跳跃瞬间初速度。假如我们想直接从界面设置跳跃瞬间初速度来控制角色的跳跃表现呢？

三、代码框架说明：

1.反射宏：

CLASS(class_name,tags…)

用以反射类型的申明，替换 class 字段,class_name 参数填写类型名称，需要继承基类则直接按继承写法写在该参数内，tags 参数填写传递给反射系统的反射

标签，多个标签需以‘,’分隔；

例：

```
REFLECTION_TYPE(GameObjectMaterialDesc)
STRUCT(GameObjectMaterialDesc, Fields)
{
    REFLECTION_BODY(GameObjectMaterialDesc)
    std::string m_base_color_texture_file;
    std::string m_metallic_roughness_texture_file;
    std::string m_normal_texture_file;
    std::string m_occlusion_texture_file;
    std::string m_emissive_texture_file;
    bool        m_with_texture {false};
};
```

```
REFLECTION_TYPE(MeshComponent)
CLASS(MeshComponent : public Component, WhiteListFields)
{
    REFLECTION_BODY(MeshComponent)
public:
    MeshComponent() {};

    void postLoadResource(std::weak_ptr<GObject> parent_object) override;

    const std::vector<GameObjectPartDesc>& getRawMeshes() const { return m_raw_meshes; }

    void tick(float delta_time) override;

private:
    META(Enable)
    MeshComponentRes m_mesh_res;

    std::vector<GameObjectPartDesc> m_raw_meshes;
};
```

图 1-CLASS 宏示例

STRUCT(struct_name,tags…)

用以反射结构体的申明，替换 struct 字段, struct_name 参数填写结构体名称，tags 参数填写传递给反射系统的反射标签，多个标签需以‘,’分隔；

例：

```
REFLECTION_TYPE(ComponentDefinitionRes)
CLASS(ComponentDefinitionRes, Fields)
{
    REFLECTION_BODY(ComponentDefinitionRes);

public:
    std::string m_type_name;
    META(Disable)
    std::string m_component;
};
```

图 2-STRUCT 宏示例

META(tags…)

用以给类或和结构体的属性增加反射信息标签，写在属性定义前，tags 参数填写传递给反射系统的反射标签，多个标签需以‘,’分隔；

例：

```
REFLECTION_TYPE(ComponentDefinitionRes)
CLASS(ComponentDefinitionRes, Fields)
{
    REFLECTION_BODY(ComponentDefinitionRes);

public:
    std::string m_type_name;
    META(Disable)
    std::string m_component;
};
```

图 3-META 宏示例

REFLECTION_TYPE (class_name)

用以提供反射访问器的声明，配合 CLASS 使用，写在类型定义前，class_name 填写类型名称

例：

```
REFLECTION_TYPE(ComponentDefinitionRes)
CLASS(ComponentDefinitionRes, Fields)
{
    REFLECTION_BODY(ComponentDefinitionRes);

public:
    std::string m_type_name;
    META(Disable)
    std::string m_component;
};
```

图 4- REFLECTION_TYPE 宏示例

REFLECTION_BODY(class_name)

对访问器进行友元声明，允许访问器访问其私有属性，写在类型定义内第一行，class_name 填写类型名称，需配合 REFLECTION_TYPE，CLASS 使用

例：

```
REFLECTION_TYPE(ComponentDefinitionRes)
CLASS(ComponentDefinitionRes, Fields)
{
    REFLECTION_BODY(ComponentDefinitionRes);

public:
    std::string m_type_name;
    META(Disable)
    std::string m_component;
};
```

图 5- REFLECTION_BODY 宏示例

2.反射标签:

Fields

表明类型或结构体中除特殊标记的属性外(参见 Disable)，所有属性均需反射，该标签需在 CLASS 或 STRUCT 中声明。如图 6 所示，MotorComponentRes 类型标记了 Fields 标签，因此其所有属性都支持反射。

```
REFLECTION_TYPE(MotorComponentRes)
CLASS(MotorComponentRes, Fields)
{
    REFLECTION_BODY(MotorComponentRes);

public:
    MotorComponentRes() = default;
    ~MotorComponentRes();

    float m_move_speed { 0.f};
    float m_jump_height {0.f};
    float m_max_move_speed_ratio { 0.f};
    float m_max_sprint_speed_ratio { 0.f};
    float m_move_acceleration {0.f};
    float m_sprint_acceleration { 0.f};

    Reflection::ReflectionPtr<ControllerConfig> m_controller_config;
};
```

图 6- Fields 标签示例

WhiteListFields

表明类型或结构体中只有标记过的属性才允许反射(参见 Enable)，该标签需在 CLASS 或 STRUCT 中声明。如图 7 所示，MeshComponent 类型标记了 WhiteListFields 标签，因此只有 m_mesh_res 才允许反射，

```
REFLECTION_TYPE(MeshComponent)
CLASS(MeshComponent : public Component, WhiteListFields)
{
    REFLECTION_BODY(MeshComponent)
public:
    MeshComponent() {};

    void postLoadResource(std::weak_ptr<GObject> parent_object) override;

    const std::vector<GameObjectPartDesc>& getRawMeshes() const { return m_raw_meshes; }

    void tick(float delta_time) override;

private:
    META(Enable)
    MeshComponentRes m_mesh_res;

    std::vector<GameObjectPartDesc> m_raw_meshes;
};
```

图 7- WhiteListFields 及 Enable 标签示例

Disable

表明该属性无需反射，该标签需要在属性前以 META 参数的形式声明,且只有在 CLASS 或 STRUCT 中声明为 Fields 时有效。如图 8 所示，MotorComponentRes 类型标记了 Fields 标签，m_controller_config 标记了 Disable，因此它不会被反射。

```
REFLECTION_TYPE(MotorComponentRes)
CLASS(MotorComponentRes, Fields)
{
    REFLECTION_BODY(MotorComponentRes);

public:
    MotorComponentRes() = default;
    ~MotorComponentRes();

    float m_move_speed { 0.f};
    float m_jump_height {0.f};
    float m_max_move_speed_ratio { 0.f};
    float m_max_sprint_speed_ratio { 0.f};
    float m_move_acceleration {0.f};
    float m_sprint_acceleration { 0.f};
    META(Disable)
    Reflection::ReflectionPtr<ControllerConfig> m_controller_config;
};
```

图 8- REFLECTION_BODY 宏示例

Enable

表明该属性需要反射，该标签需要在属性前以 META 参数的形式声明,且只有在 CLASS 或 STRUCT 中声明为 WhiteListFields 时有效。如图 6。

3.反射功能类:

FieldAccessor:

属性访问器类，提供属性名称，属性类型名称以及 set/get 方法。

主要方法:

const char* getFieldName()：获取属性名称;

const char* getFieldTypeName()：获取属性的类型名称;

bool getTypeMeta(TypeMeta& filed_type)：获取属性的类型元信息(参见 TypeMeta)，filed_type 为返回的类型元信息，方法本身返回一个布尔值，只有当属性类型也支持反射时，才会为 true;

void* get(void* instance)：instance 为属性所属类型的实例指针，方法返回属性的指针;

void set(void* instance, void* value): instance 为属性所属类型的实例指针，value 为要设置的值的指针

TypeMeta

类型元信息，提供类型的名称、属性访问器列表等。

主要方法:

TypeMeta newMetaFromName(std::string type_name)：静态方法，可通过类

型名称直接创建 TypeMeta;

std::string getTypeName()：返回类型名称;

int getFieldsList(FieldAccessor*& out_list)：out_list 为 FieldAccessor 数组头指针，该方法会通过 out_list 返回类型所有反射属性的访问器数组，方法本身返回数组大小;

int getBaseClassReflectionInstanceList(ReflectionInstance*& out_list,void* instance)：instance 为类型实例指针，out_list 为 ReflectionInstance 数组头指针，该方法会通过 out_list 返回类型所有基类的 ReflectionInstance 数组，方法本身返回数组大小。

ReflectionInstance:

反射实例，包含了类型的元信息及实例指针。

属性:

m_meta：类型为 TypeMeta，表示类型元信息;

m_instance：类型为 void*，表示实例的指针;

4.反射生成面板过程

利用反射动态生成属性面板过程

Piccolo 中 Components Details 面板通过反射系统自动生成，显示场景中选中物体的所有 component 信息，如下图所示:

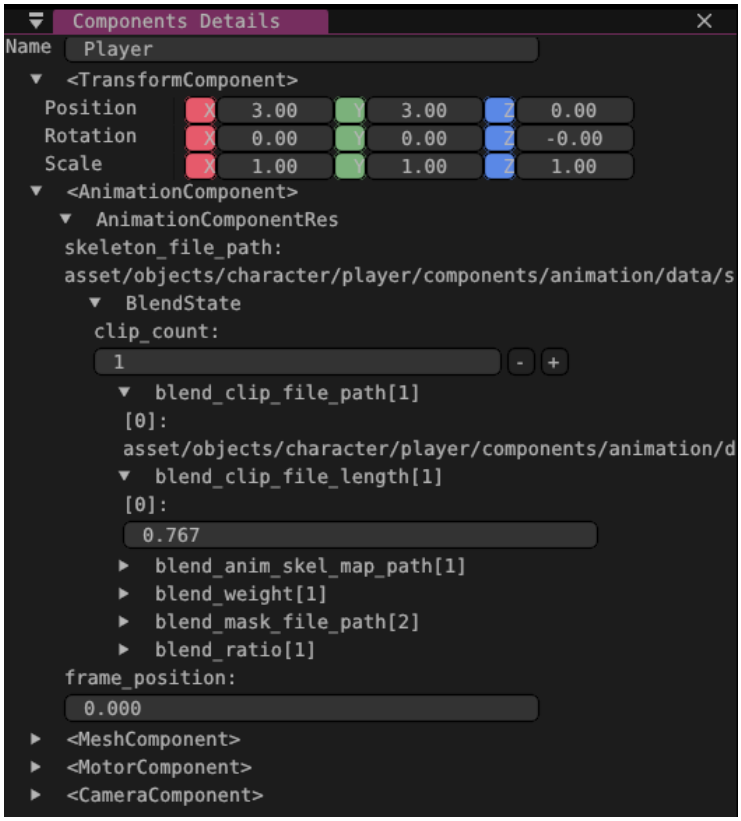

图 9- Components Details 面板

实现思路：

使用反射系统，将复杂类型拆解为 UI 控件可直接表示的原子类型，生成对应的 UI 控件，形成原子类型控件树，展示复杂类型信息。

实现过程：

1. 如图 10 所示，在 editor_ui.cpp 文件的 showEditorDetailWindow 方法中，获取选中物体的所有 component 对象**（图 10 绿框部分）**。

```
std::vector<Reflection::ReflectionPtr<Component>>&&  selected_object_components = selected_object->getComponents();
for (auto component_ptr : selected_object_components)
{
    m_editor_ui_creator["TreeNodePush"](("<" + component_ptr.getTypeName() + ">").c_str(), nullptr);
    Reflection::ReflectionInstance object_instance(
        Piccolo::Reflection::TypeMeta::newMetaFromName(component_ptr.getTypeName().c_str()),
        component_ptr.operator->());
    createClassUI(object_instance);
    m_editor_ui_creator["TreeNodePop"](("<" + component_ptr.getTypeName() + ">").c_str(), nullptr);
}
```

图 10-editor_ui.cpp-showEditorDetailWindow 方法

2. 为每个 component 对象构造反射实例 **(图 10 红框部分)**，并依次调用

createClassUI 对反射实例进行类型拆解。

3. 如图 11 所示，在 editor_ui.cpp 文件的 createClassUI 方法中，先获取该类型的所有基类反射实例，并递归调用该方法生成基类的 UI，然后调用 creatLeafNodeUI 方法生成该类型属性 UI。

```
void EditorUI::createClassUI(Reflection::ReflectionInstance& instance)
{
    Reflection::ReflectionInstance* reflection_instance;
    int count = instance.m_meta.getBaseClassReflectionInstanceList(reflection_instance, instance.m_instance);
    for (int index = 0; index < count; index++)
    {
        createClassUI(reflection_instance[index]);
    }
    createLeafNodeUI(instance);

    if (count > 0)
        delete[] reflection_instance;
}
```

图 11-editor_ui.cpp-createClassUI 方法

4. 如图 12 所示，在 editor_ui.cpp 文件的 createLeafNodeUI 方法中，获取到所有属性访问器。

```
Reflection::FieldAccessor* fields;
int                        fields_count = instance.m_meta.getFieldsList(fields);
```

图 12-属性访问器获取

5. 如图 13 所示，遍历所有属性访问器，使用属性类型名称查找对应的构建方法，传入属性名称和属性地址，进行最后的 UI 生成。

```
for (size_t index = 0; index < fields_count; index++)
{
    Reflection::FieldAccessor field = fields[index];
    //...
    m_editor_ui_creator[field.getFieldTypeName()](field.getFieldName(),
        field.get(instance.m_instance));

}
```

图 13-遍历访问器生成 UI

6. UI 生成过程中，需要对 UI 支持的原子类型进行数据和 UI 控件的绑定，因此需要建立原子类型的 UI 构建方法表，即 5 中用到的 m_editor_ui_creator。

构建方法如图 14 所示，m_editor_ui_creator 是一个 map <std::string,std::function<const std::string&, void*>>,对于每一种 UI 支持的原子类类型，都需要注册其属性名称及处理方法。

```
m_editor_ui_creator["std::string"] = [this, &asset_folder](const std::string& name, void* value_ptr) -> void {
    if (g_node_depth == -1)
    {
        std::string label = "##" + name;
        ImGui::Text("%s", name.c_str());
        ImGui::SameLine();
        ImGui::Text("%s", (*static_cast<std::string*>(value_ptr)).c_str());
    }
    else
    {
        if (g_editor_node_state_array[g_node_depth].second)
        {
            std::string full_label = "##" + getLeafUINodeParentLabel() + name;
            ImGui::Text("%s", (name + ":").c_str());
            std::string value_str = *static_cast<std::string*>(value_ptr);
            if (value_str.find_first_of('/') != std::string::npos)
            {
                std::filesystem::path value_path(value_str);
                if (value_path.is_absolute())
                {
                    value_path = Path::getRelativePath(asset_folder, value_path);
                }
                value_str = value_path.generic_string();
                if (value_str.size() >= 2 && value_str[0] == '.' && value_str[1] == '.')
                {
                    value_str.clear();
                }
            }
            ImGui::Text("%s", value_str.c_str());
        }
    }
};
```

图 14-m_editor_ui_creator 注册过程

7. 如图 15 所示，除了原子类型外，对于复杂类型，需要继续调用 createClassUI 方法进行拆解。

```
auto ui_creator_iterator = m_editor_ui_creator.find(field.getFieldTypeName());
if (ui_creator_iterator == m_editor_ui_creator.end())
{
    Reflection::TypeMeta field_meta =
        Reflection::TypeMeta::newMetaFromName(field.getFieldTypeName());
    if (field.getTypeMeta(field_meta))
    {
        Reflection::ReflectionInstance child_instance =
            Reflection::ReflectionInstance(field_meta, field.get(instance.m_instance));
        m_editor_ui_creator["TreeNodePush"](field_meta.getTypeName(), nullptr);
        createClassUI(child_instance);
        m_editor_ui_creator["TreeNodePop"](field_meta.getTypeName(), nullptr);
    }
    else
    {
        if (ui_creator_iterator == m_editor_ui_creator.end())
        {
            continue;
        }
        m_editor_ui_creator[field.getFieldTypeName()](field.getFieldName(),
            field.get(instance.m_instance));
    }
}
```

图 15-非原子类型处理

8. 如图 16 所示，对于容器类型，如 array，需要遍历 array，对于每个元素进行类型判断，原子类型则直接生成 UI，非原子类型继续调用 createClassUI 进行类型拆解。

```
if (field.isArrayType())
{
    Reflection::ArrayAccessor array_accessor;
    if (Reflection::TypeMeta::newArrayAccessorFromName(field.getFieldTypeName(), array_accessor))
    {
        void* field_instance = field.get(instance.m_instance);
        int   array_count = array_accessor.getSize(field_instance);
        m_editor_ui_creator["TreeNodePush"](
            std::string(field.getFieldName()) + "[" + std::to_string(array_count) + "]", nullptr);
        Reflection::TypeMeta item_type_meta_item =
            Reflection::TypeMeta::newMetaFromName(array_accessor.getElementTypeName());
        auto item_ui_creator_iterator = m_editor_ui_creator.find(item_type_meta_item.getTypeName());
        for (int index = 0; index < array_count; index++)
        {
            if (item_ui_creator_iterator == m_editor_ui_creator.end())
            {
                m_editor_ui_creator["TreeNodePush"]("[" + std::to_string(index) + "]", nullptr);
                Reflection::ReflectionInstance object_instance(
                    Piccolo::Reflection::TypeMeta::newMetaFromName(item_type_meta_item.getTypeName().c_str()),
                    array_accessor.get(index, field_instance));
                createClassUI(object_instance);
                m_editor_ui_creator["TreeNodePop"]("[" + std::to_string(index) + "]", nullptr);
            }
            else
            {
                if (item_ui_creator_iterator == m_editor_ui_creator.end())
                {
                    continue;
                }
                m_editor_ui_creator[item_type_meta_item.getTypeName()](
                    "[" + std::to_string(index) + "]", array_accessor.get(index, field_instance));
            }
        }
        m_editor_ui_creator["TreeNodePop"](field.getFieldName(), nullptr);
    }
}
```

图 16-容器方法处理

四、作业提交说明

作业的评分分为基础与提高两部分。想要完成作业同学们需达成基础评分条件，这样便视为通过作业。有能力有兴趣的同学还可以尝试达成提高项以获取更多分数。

作业提交格式

提交作业为一个压缩文件，命名为 Games104_homework4.zip 或 Games104_homework4.rar，其中内容为：

1. 一个文件夹 runtime，里面是 Piccolo Runtime 部分的代码；
2. 一个文件夹 editor，里面是 Piccolo Editor 部分的代码；
3. Games104_homework4_report.doc 或 Games104_homework4_report.docx 或 Games104_homework4_report.pdf 格式的报告文档。 报告内容包含以下几点（若只完成部分题目，可以只写对应部分的报告）：
    - 新增或修改属性的定义及意义说明
    - 新增或修改属性在 Components Details 面板上显示的截图
    - 使该属性在其系统内生效的代码说明，包括代码解释及实现思路说明
4. Games104_homework4_report.MP4，展示新增或修改属性在其系统内生效的效果。

打分细则

1. 基础：[40 分] 掌握反射宏及反射标签用法，正确完成反射信息标注。

2. 基础：[40 分] 掌握反射访问器的用法，读懂 UI 生成代码，能够在 Detail 面板正确显示新增或修改属性。

3. 提高：[20 分] 使新增或修改属性在其系统内按照预期生效。