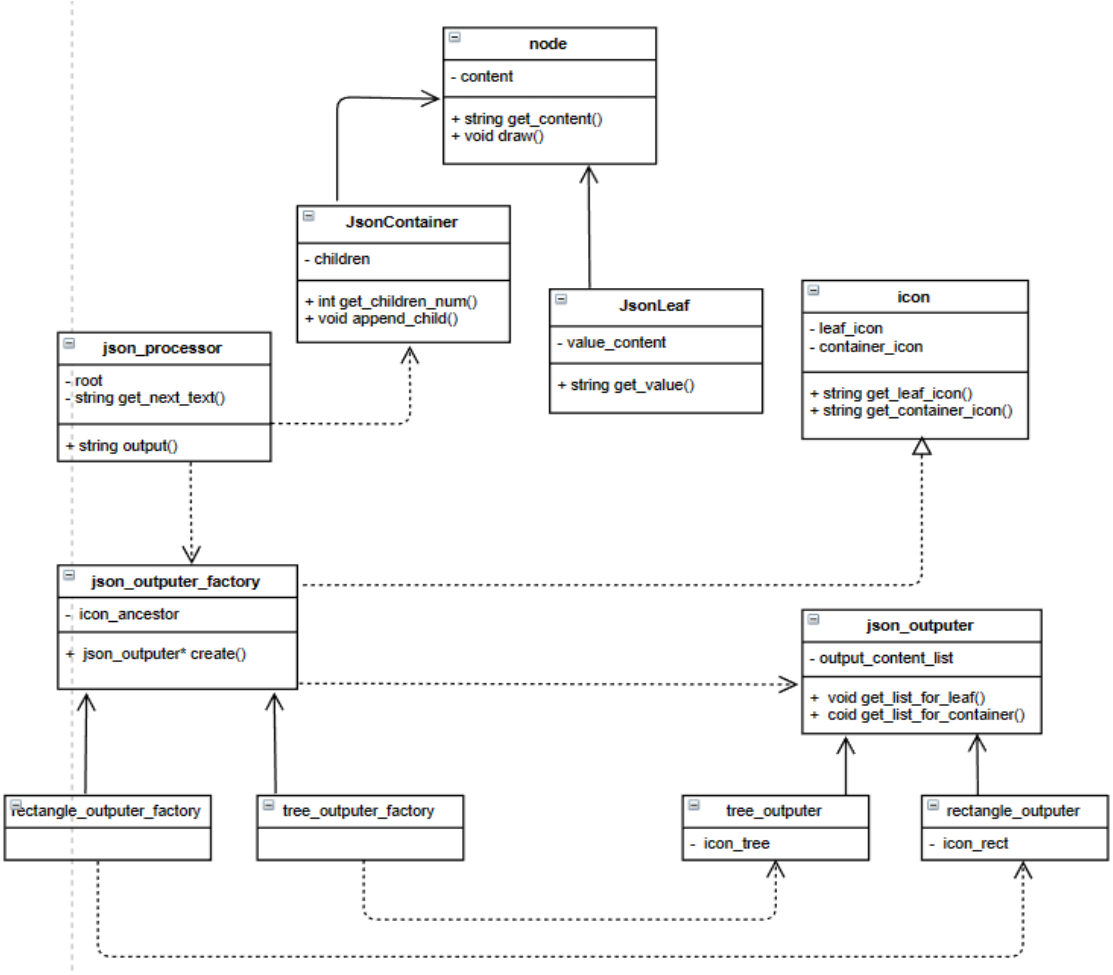
软件工程 design_pattern FJE

莫明昊 21307147

类图：



node 类

由 json 文件内容的结构，考虑编写 node 类，派生出 JsonContainer 类与 JsonLeaf 类，分别表示 json 中的非叶子节点和叶子节点。每一个 node 都有 content 属性，表示自己的节点内容（一段字符串）；JsonContainer 中加入子节点（children）属性和层级（layer_count）属性，JsonLeaf 中加入值（value_content）属性。

```
class node {
protected:
    string content;
public:
    node();
    virtual ~node();
    void set_content(const string& data);
    string get_content();
    virtual void draw(json_outputer* json_renderer, int level) = 0;
};

class JsonContainer :public node {
private:
    vector<shared_ptr<node>> children;
    int layer_count;
```

```
class JsonLeaf :public node {
private:
    string value_content;
public:
    JsonLeaf();
    virtual ~JsonLeaf();
```

json_processor 类

为了读取 json 文件中的内容，做法是将其中的文本拼接成一个长字符串存储。编写 json_processor 类，其中存储一个 JsonContainer 的智能指针 root，并提供 load（）函数与 output（）函数。

```
class json_processor {
private:
    shared_ptr<JsonContainer> root;

public:
    json_processor();
    ~json_processor();
    string output(json_outputer_factory* factory);
    void load(const string& path);
```

```
void json_processor::load(const string& path) {
    ifstream file(path);
    if (!file.is_open()) {
        printf("open error!\n");
        return;
    }

    stringstream content_stream;
    content_stream << file.rdbuf(); // 将文件内容读入 stringstream

    // 关闭文件，避免资源泄漏
    file.close();

    string content = content_stream.str();

    size_t now_pos = 0;
    root = process_container(content, 0, now_pos);
}
```

Load（）函数中，首先打开文件，读入内容。由于根节点是一个 container 节点，调用函数 process_container（）来进行递归处理。处理过程是：先读取到“{”，然后读取到键，然后读取到“:”，然后分两种情况：

1.再次读取到“{”，说明接下来又是一个 container，递归调用 process_container（）。

2.读取到的不是“{”，说明接下来是一个叶子节点，读取其值。

```
// Value is a container, recursively call this function
if (data[index] == '{') {
    shared_ptr<JsonContainer> child = process_container(data, level + 1, index);
    if (!child) {
        return nullptr;
    }
    child->set_content(key);
    container->append_child(child);
}
// Value is a leaf
else {
    shared_ptr<JsonLeaf> child = process_leaf(data, index);
    if (!child) {
        return nullptr;
    }
    child->set_content(key);
    container->append_child(child);
}
```

无论以上哪种情况，都需要创建 node 类的子类对象，来向本节点（JsonContainer）的子节点列表（children）中添加子节点。

json_processor 类中再提供 output 函数，通过传参（一个抽象工厂对象）来产生相应风格的 json_outputer 对象，并调用 json_outputer 对象中的 get_outputer_content()来得到最终输出结果。此函数在 main 函数中被调用。

```
public:
    json_processor();
    ~json_processor();
    string output(json_outputer_factory* factory);
```

json_outputer 类

接下来编写 json_outputer 类，它有一个名为 outputer_content_list 的向量列表，用于存储最终输出的文本内容。并且它还有两个需要在子类中重写的函数：get_list_for_leaf()和 get_list_for_container()，分别处理得到叶子节点和非叶子节点的输出文本，添加到 outputer_content_list 中。

```
class json_outputer
{
protected:

    vector<string> output_content_list;
public:
    json_outputer();
    virtual void get_list_for_leaf(JsonLeaf* leaf, int level) = 0;
    virtual void get_list_for_container(JsonContainer* container, int level) = 0;
```

json_outputer 类派生出两个子类：tree_outputer 和 rectangle_outputer，分别对应树形和矩形两种风格。这两个类都通过 get_list_for_leaf()函数和 get_list_for_container()函数来向 outputer_content_list 属性赋值。为了实现这两个函数，两个子类分别编写了各自的辅助函数，作为私有的成员函数。

```
class tree_outputer :public json_outputer {
private:
    icon* icon_tree;
    vector<string> before_text;
    int if_final_child;
public:
    tree_outputer(icon* input_icon = nullptr);
    virtual void get_list_for_leaf(JsonLeaf* leaf, int level) ;
    virtual void get_list_for_container(JsonContainer* container, int level) ;
    virtual ~tree_outputer();

private:
    void adjust_before_text(int level, int child_count);
    void update_branch_prefix(int level, int currentIndex, int child_count);
    void append_to_output();
};
```

```
class rectangle_outputer :public json_outputer
{
private:
    icon* icon_rect;
    size_t len;
    vector<string> before_text;

public:
    rectangle_outputer(icon* icon = nullptr);
    virtual void get_list_for_leaf(JsonLeaf* jleaf, int layer) ;
    virtual void get_list_for_container(JsonContainer* jcontainer, int layer) ;
    virtual ~rectangle_outputer();
private:
    size_t char_num(const string& str);
    void update_before_text(int depth, int index);
    string concatenate_before_text();
    void finalize_output();
};
```

get_list_for_container()函数的基本逻辑都是：先向 outputer_content_list 尾部添加图标，再添加本节点的内容，然后遍历每一个子节点，进行相应的前缀处理，调用子节点的 draw（）函数。

get_list_for_leaf()函数的基本逻辑都是：先向 outputer_content_list 尾部添加图标，再添加本节点的内容，再判断是否有值，有则添加。

需要注意，由于矩形风格需要将输出结果“围成矩形”，所以 rectangle_outputer 类有一个 len 属性，用于记录最长的一行的输出长度，以便最终输出的每一行都用横线补齐至此长度。

设计模式

工厂方法

为 json_outputer/ tree_outputer/ rectangle_outputer 编写其对应的工厂类：

json_outputer_factory/ tree_outputer_factory/ rectangle_outputer_factory，前者是后两者的父类。

```
class json_outputer_factory {
protected:
    icon* icon_ancestor;
public:
    json_outputer_factory(const string& icon_path);
    virtual json_outputer* create() = 0;
    ~json_outputer_factory();
};
```

父类 json_outputer_factory 中维护一个 icon 对象 icon_ancestor，并有一个需要被子类重写的 create（）函数。构造函数中，根据输入的图标配置文件路径来创建 icon 对象 icon_ancestor：

```
json_outputer_factory::json_outputer_factory(const string& icon_path = ""){
    icon_ancestor = nullptr;
    if (icon_path != ""){
        icon_ancestor = new icon(icon_path);
    }
}
```

tree_outputer_factory/ rectangle_outputer_factory 两个子类相似，都通过各自的 create（）函数来创建对应风格的 json_outputer 对象并返回此对象。

```
class rectangle_outputer_factory :public json_outputer_factory {
public:
    rectangle_outputer_factory(const string& icon_path = "");
    virtual ~rectangle_outputer_factory();
    virtual json_outputer* create() ;
};
```

```
json_outputer* rectangle_outputer_factory::create(){
    rectangle_outputer* ans=new rectangle_outputer(icon_ancestor);
    return ans;
}
```

这样，这两个工厂子类的对象就能以 json_outputer_factory 对象的形式传入到 json_processor 对象的 output 函数中，在 output 函数里调用相应子类的具体 create（）函数:

```
if(style == "tree"){
    tree_outputer_factory tree_factory(icon_file_name);
    content = processor.output(&tree_factory);
    printf("%s\n", content.c_str());
}
else if(style == "rectangle"){
    rectangle_outputer_factory rect_factory(icon_file_name);
    content = processor.output(&rect_factory);
    printf("%s\n", content.c_str());
}
else {
    printf("style error\n");
    return 0;
}
```

组合模式

node 类派生出 JsonContainer 类与 JsonLeaf 类，而 JsonContainer 中维护一个指向自己子节点的只能指针向量，指针的类型是 shared_ptr<node>。这样中间节点的子节点就可以是中间节点、叶子节点，也可以是未来可能扩展出的其他 node 类的子类。

```
class JsonContainer :public node {
private:
    vector<shared_ptr<node>> children;
    int layer_count;
```

运行结果展示：

运行细节见根目录下的 readme.txt。

src/main.cpp 即为主程序入口，也是 Funny_JSON_Explorer 领域模型中的 FunnyJSONExplorer。

运行 make_build.bat 后，进入到 run/bin 文件夹，进入命令行，执行命令如：

main fje -f test1.json -s tree -i icon1.config

指定 json 文件名、风格、图标族文件名。

运行截图如下：（四张截图存放在根目录下的 screenshot 目录下）