nlohmann json 项目阅读报告

杨楠 2021010711

2022年6月

1 项目介绍

JSON的全称是JavaScript Object Notation，是一种轻量级的数据交换格式，易于阅读和编写，也易于机器解析和生成。

比如vscode就是用json为后缀（扩展名）的文件保存相关配置信息，比如编译任务的配置、调试环境的配置（tasks.json, launch.json等等）。

nlohmann json又名JSON for Modern C++，为C++解析json提供了简便的方案。

使用此库，可以很方便地建立json对象/数组，编辑json中的内容，读入并解析json文件，将内容输出到json文件等等。

此库也支持中文字符串的信息的保存、读入与输出，此时要将编码设置为utf-8。

1.1 补充内容：JSON语法

JSON的语法是JavaScript对象表示语法的子集。JSON建构于以下2种结构：对象和数组。

对象是name: value键值对的无序集合，用大括号{}保存，键值对之间用逗号隔开。

数组是value的有序集合，用中括号[]保存，value之间用逗号隔开（特别地，只有一个value时可以不用加中括号[]）。

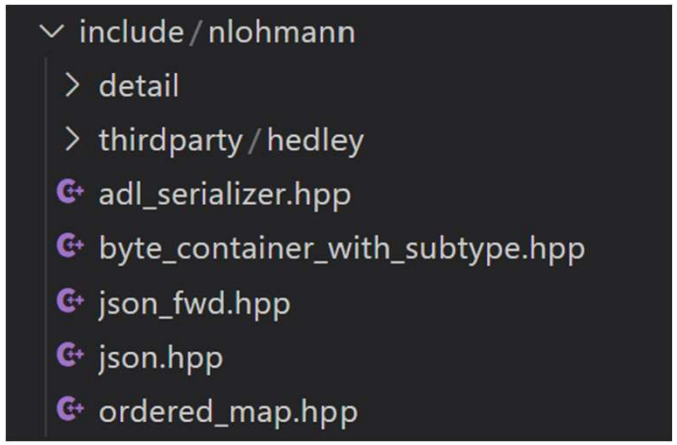
name只能是字符串，且要用双引号。

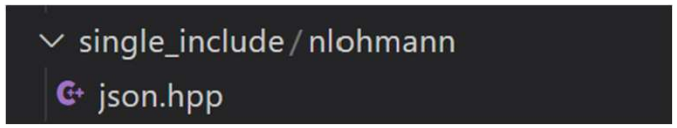
value只能是以下类型之一：数字（整数或浮点数），字符串（要用双引号），bool值（true或false，没有双引号），null（没有双引号），对象，数组。

一个json文件的内容只能表示一个对象或一个数组，从而其格式只能是以下2种类型：对象（最外层有且只有一对大括号）或者数组（最外层有且只有一对中括号）（特别地，只有一个value时可以不用加中括号）（具体示例可见tasks.json）。

https://www.json.org/json-zh.html

1.2 两种include





项目提供2类文件夹，代表2类使用方式（任选一种使用即可）：include和single_include

两者在使用时，将文件夹放到编译路径后，在C++文件内补充#include <nlohmann/json.hpp>

（具体配置方法见readme）

前者将核心代码（接口）写在json.hpp内，而将一些细节的处理安排在其他hpp文件（比如detail文件夹中的hpp）中，比较有条理，封装性比较好

后者则是将所有代码写在一个头文件json.hpp中，使用方便简单，乃至可以直接将此文件放在工程路径中

#include "json.hpp"

缺点是代码比较多（23762行，前者的json.hpp为5176行），相对而言条理性稍差，封装隐蔽性稍差

从方便性的角度，项目作者推荐使用后者方法

后续内容分析主要采用前者include

https://github.com/nlohmann/json

2 项目特点

项目采用C++11语言作为编写设计和语法依据

一、使用类模板basic_json作为容器来储存json对象

常用的json是basic_json的默认（缺省）实例化

这种处理方式类似于C++的STL库string等等，常用的string其实是类模板basic_string的char实例化，其他类型的实例化还有u16string，u32string等等

```
// 节选自STL库stringfwd.h
template<typename _CharT, typename _Traits =
char_traits<_CharT>,
         typename _Alloc = allocator<_CharT> >
   class basic_string;
 /// A string of @c char
 typedef basic_string<char>    string;
/// A string of @c char16_t
 typedef basic_string<char16_t> u16string;
 /// A string of @c char32_t
 typedef basic_string<char32_t> u32string;
```

```
// 节选自json_fwd.hpp
using json = basic_json<>;
```

二、以namespace nlohmann嵌套namespace detail的结构

采用上述2个命名空间，前者定义basic_json, json_pointer等主要的类，后者主要定义一些细节的函数或类，以类的公共函数作为接口，方便调用

上述命名空间的结构安排，还可以尽量减少命名冲突

在文件中，为了使用方便，可以加上

```
using json = nlohmann::json;
```

也可以写为

```
using nlohmann::json;
```

这样就方便声明json对象

```
using namespace nlohmann; //再或者这样
```

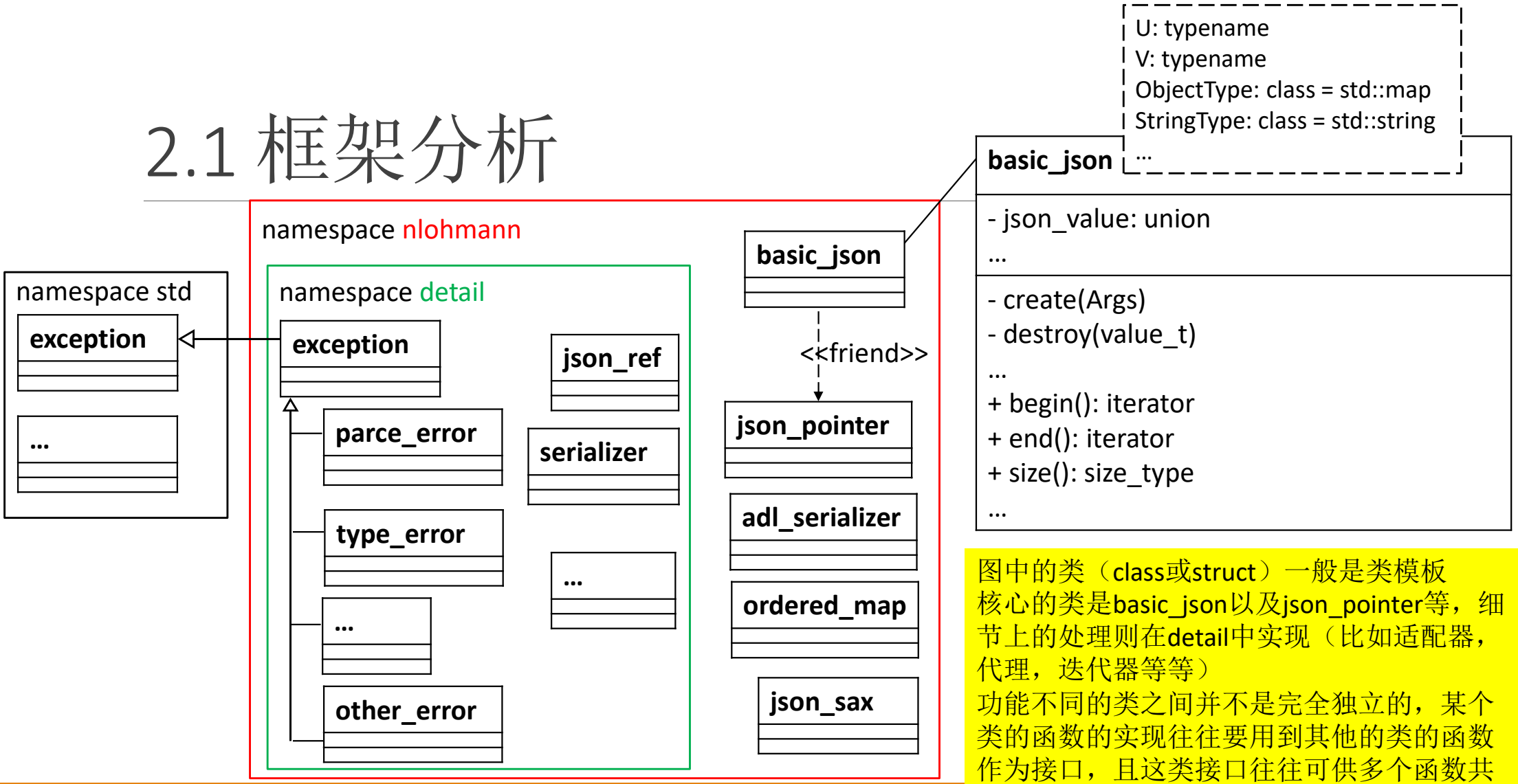
类模板basic_json的默认处理方式：
对象的处理是用map，数组的处理是用vector，字符串是string，整数是int64_t和uint64_t，浮点数是double，布尔值是bool，等等
实际构造时也可根据需要，自行设置类型
每个键值对的存储是用pair，根据函数的使用需求，还会涉及tuple，map，tree等STL库中的类型

考虑到json语法中对象里的键值对的无序性，json类是用map处理对象，里面的键值对是按照name的字典序进行储存的，如果在存储json时想保留原始的键值对的顺序，可以用ordered_json，则ObjectType为ordered_map
ordered_map是在namespace nlohmann里的一个struct，结构与std::map类似，但是保留顺序

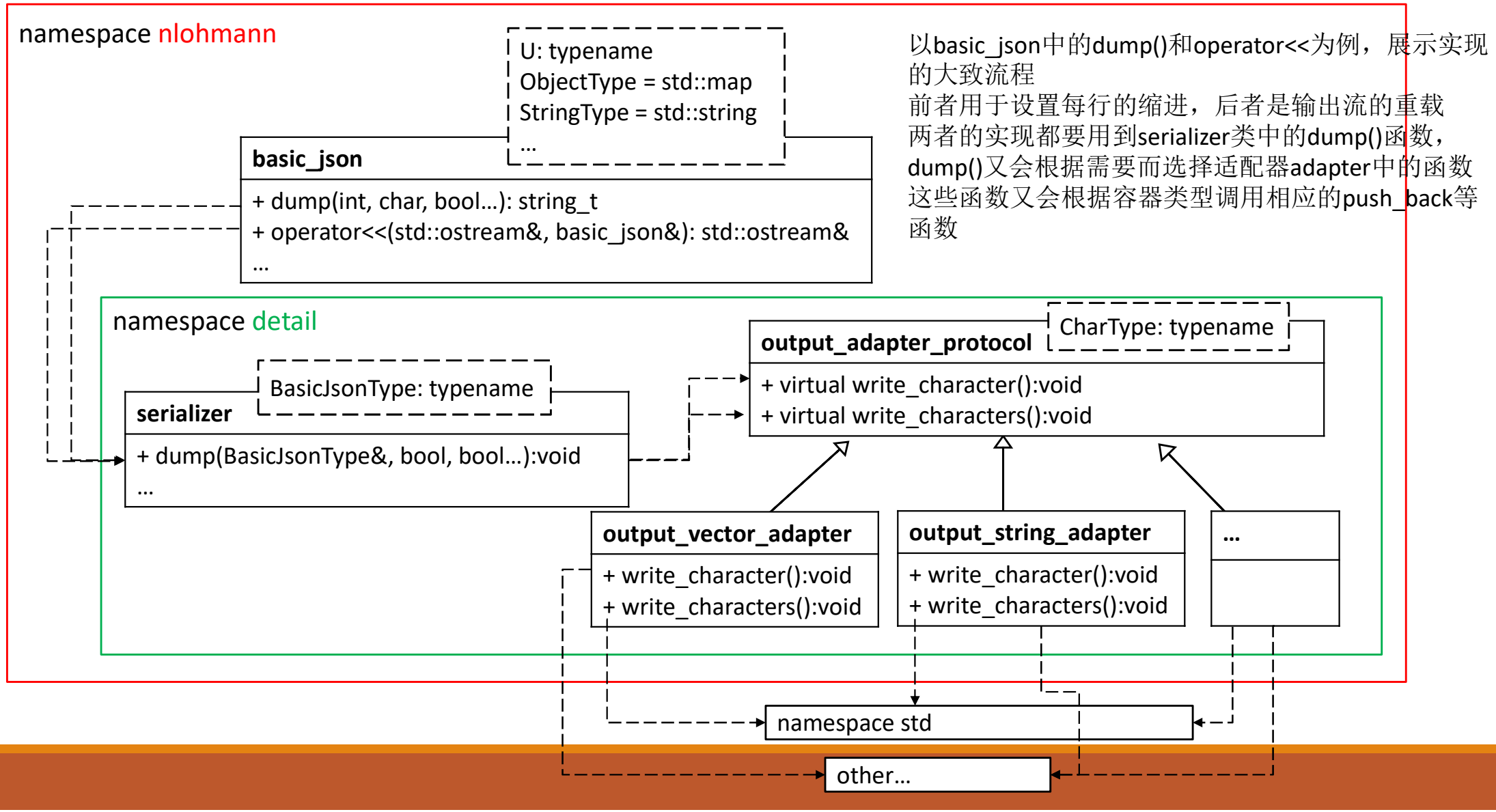
```
// 节选自json_fwd.hpp
template<template<typename U, typename V, typename... Args> class ObjectType =
        std::map,
        template<typename U, typename... Args> class ArrayType = std::vector,
        class StringType = std::string, class BooleanType = bool,
        class NumberIntegerType = std::int64_t,
        class NumberUnsignedType = std::uint64_t,
        class NumberFloatType = double,
        template<typename U> class AllocatorType = std::allocator,
        template<typename T, typename SFINAE = void> class JSONSerializer =
        adl_serializer,
        class BinaryType = std::vector<std::uint8_t>>
class basic_json;
```

```
// 节选自json_fwd.hpp
using json = basic_json<>;
using ordered_json = basic_json<nlohmann::ordered_map>;
```

2.1 框架分析



图中的类（class或struct）一般是类模板
核心的类是basic_json以及json_pointer等，细节上的处理则在detail中实现（比如适配器，代理，迭代器等等）
功能不同的类之间并不是完全独立的，某个类的函数的实现往往要用到其他的类的函数作为接口，且这类接口往往可供多个函数共用（调用），从而提高使用效率



以basic_json中的dump()和operator<<为例，展示实现的大致流程

前者用于设置每行的缩进，后者是输出流的重载

两者的实现都要用到serializer类中的dump()函数，dump()又会根据需要而选择适配器adapter中的函数

这些函数又会根据容器类型调用相应的push_back等函数

2.2 使用例子

JSON语法里，创建对象有2类方式
方式一：
var obj = new Object();
obj.name = "yang";
obj.age = 19;
方式二：
var obj = { "name" : "yang", "age": 19};
//这里name和age也可以不用加双引号
访问及修改value有2类方式：点号或中括号
obj["age"] = 20;
obj.sex = "boy";
//这2种方式既可以修改已有的name对应的value，也可用于新建新的键值对

https://www.runoob.com/js/js-tutorial.html
https://www.runoob.com/json/json-tutorial.html

在C++中使用本项目，也有类似的创建、访问和修改方式
//节选自demo1.cpp

```
9     json j1 = { {"name", "yang"}, {"age", 19} };
10    json j2 = R"(
11        {
12            "name": "yang",
13            "age": 19
14        }
15    )"_json;  //raw string 双引号后紧跟着_json是用于识别读取
```

此外，还可以将C++类型的容器转为json，以及通过文件流读取json文件，等等，在此不详述
访问和修改也有与JSON语法类似的2种方式：.at()或中括号
//节选自demo1.cpp

```
25    j1["name"] = "yangnan";
26    j1.at("age") = 20;
27    cout << j1.at("name") << endl; //"yangnan"
28    cout << j1["age"] << endl; //20
```

注意：中括号还可用于新建一个原先不存在的name，如果cout一个中括号下标加原先不存在的name则会输出null
而at只能访问或修改原有的name的value，否则会在输出中告知错误信息

2.2 使用例子

demo2.cpp演示了json文件读取，以及数组访问等内容，在此不便详述

输出整个json对象/数组

json数据中的空格和换行等内容不影响读取与解析，默认输出时会以字符串的形式不带空格地一整行输出

而dump()函数可以用于设置每行的缩进以及缩进所用的字符（对于对象与数组嵌套的情况则依次按照层次进行缩进）

//节选自demo1.cpp

```
16      cout << j1 << endl; // {"age":19,"name":"yang"}
17      cout << j2.dump(4) << endl;
18      /*
19  {
20      "age": 19,
21      "name": "yang"
22  }
23      */
```

dump()函数的参数及默认值：

indent表示每行的每层缩进值

默认为-1，表示不换行

indent>=0则换行

indent_char表示缩进字符，默认为空格

可以自行设置为其他ASCII字符

函数实现的大致流程可参考ppt第8页的UML图

//节选自json.hpp

```
1293    string_t dump(const int indent = -1,
1294                  const char indent_char = ' ',
1295                  const bool ensure_ascii = false,
1296                  const error_handler_t error_handler = error_handler_t::strict) const
```

3 功能测试

以basic_json中的push_back()函数进行测试分析

//test1.cpp

```cpp
#include <iostream>
#include <nlohmann/json.hpp>

using nlohmann::json;
using namespace std;
int main()
{
    json j = {{"name","yang"}};
    j.push_back({"sex","male"});
    cout << j << endl; //{"name":"yang","sex":"male"}
    j += {"age", 20};
    cout << j << endl; //{"age":20,"name":"yang","sex":"male"}
    return 0;
}
```

输出结果为：

```
{"name":"yang","sex":"male"}
{"age":20,"name":"yang","sex":"male"}
```

json.hpp中提供了push_back()函数的4个重载，而+=运算符的重载也是通过push_back()来实现的，相应地也有4个重载。

```
// (1)
void push_back(basic_json&& val);
void push_back(const basic_json& val);

// (2)
void push_back(const typename object_t::value_type& val);

// (3)
void push_back(initializer_list_t init);
```

（1）中的2句重载，对应移动和拷贝，用于往json数组中添加对象
（2）用于往json对象中添加对象（若干键值对），上述test1.cpp中调用的就是（2）
关于（3）中的initializer_list_t，可见下列定义：

//节选自json.hpp

```
/// helper type for initializer lists of basic_json values
using initializer_list_t = std::initializer_list<detail::json_ref<basic_json>>;
```

initializer_list是C++11中提供的新类型，用于表示某种特定类型的值的数组，若要向此形参中传递一个值的序列，需要把序列写在大括号{}内
initializer_list可用于STL的容器的初始化，比如
vector<int> v = {1, 3, 5};

nlohmann json的早期版本中并没有重载（3），导致在某些编译器下，类似于ppt第11页（也就是上述测试中的操作）出现编译报错为ambiguous overload
问题在于{"key", "value"}可以理解为object_t::value_type或者std::initializer_list<basic_json>
为解决此问题，作者增加了重载（3）

https://json.nlohmann.me/api/basic_json/push_back/
https://blog.csdn.net/fengxinlinux/article/details/72614874
https://github.com/nlohmann/json/issues/235

重载（3）中，如果：原先的json是对象（而不是数组）（is_object()），列表init中的元素个数是2（init.size()==2），第一个元素符合原先json的string类型（is_string()），那么调用重载（2），否则调用重载（1）

```
3157      /// @brief add an object to an object
3158      /// @sa https://json.nlohmann.me/api/basic_json/push_back/
3159      void push_back(initializer_list_t init)
3160      {
3161          if (is_object() && init.size() == 2 && (*init.begin())->is_string())
3162          {
3163              basic_json&& key = init.begin()->moved_or_copied();
3164              push_back(typename object_t::value_type(
3165                            std::move(key.get_ref<string_t&>()), (init.begin() + 1)->moved_or_copied()));
3166          }
3167          else
3168          {
3169              push_back(basic_json(init));
3170          }
3171      }
```

重载（2）中，要求原先的json是对象或者空（而不是数组），否则报错
如果原先是null，那么先让其转变为对象类型
之后通过调用原json中处理对象用的STL容器（默认值一般是map）的insert函数，完成push_back操作

```
3126      /// @brief add an object to an object
3127      /// @sa https://json.nlohmann.me/api/basic_json/push_back/
3128      void push_back(const typename object_t::value_type& val)
3129      {
3130          // push_back only works for null objects or objects
3131          if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_object())))
3132          {
3133              JSON_THROW(type_error::create(308, detail::concat("cannot use push_back() with ", type_name()), this));
3134          }
3135
3136          // transform null object into an object
3137          if (is_null())
3138          {
3139              m_type = value_t::object;
3140              m_value = value_t::object;
3141              assert_invariant();
3142          }
3143
3144          // add element to object
3145          auto res = m_value.object->insert(val);
3146          set_parent(res.first->second);
3147      }
```

重载（1）中的两句重载函数（左值和右值引用）的实现思路类似，区别仅在于，第2句重载是将第3081行push_back括号中的std::move(val)改为val

重载（1）要求原先的json是数组类型（而不是对象）或者空，否则报错
如果原先是null，那么先让其转变为数组类型
之后通过调用原先json中处理数组用的STL容器（默认值一般是vector）的push_back函数，完成push_back操作

```
3061    /// @brief add an object to an array
3062    /// @sa https://json.nlohmann.me/api/basic_json/push_back/
3063    void push_back(basic_json&& val)
3064    {
3065        // push_back only works for null objects or arrays
3066        if (JSON_HEDLEY_UNLIKELY(!(is_null() || is_array())))
3067        {
3068            JSON_THROW(type_error::create(308, detail::concat("cannot use push_back() with ", type_name()), this));
3069        }
3070
3071        // transform null object into an array
3072        if (is_null())
3073        {
3074            m_type = value_t::array;
3075            m_value = value_t::array;
3076            assert_invariant();
3077        }
3078
3079        // add element to array (move semantics)
3080        const auto old_capacity = m_value.array->capacity();
3081        m_value.array->push_back(std::move(val));
3082        set_parent(m_value.array->back(), old_capacity);
3083        // if val is moved from, basic_json move constructor marks it null, so we do not call the destructor
3084    }
```

Complexity

1. Amortized constant.
2. Logarithmic in the size of the container, O(log(`size()`)).
3. Linear in the size of the initializer list `init`.

总体而言，新版本加上重载（3）之后，解决了编译的模糊重载问题，其他重载的思路清晰，格式简洁，3类重载的复杂度（左图）与相应的重载涉及的STL库的类型相关

https://json.nlohmann.me/api/basic_json/push_back/

4 项目评价

本项目主要考虑的是各种功能的整合，在完备性与使用的方便性上比较突出；相对而言，关于项目的内存效率和运行速度，用项目作者的话说，“对我们（团队）来说并非特别重要（not so important to us）”。尽管有其他一些运行速度更快的json解析库，但是在功能性上，本项目依然具有优势，且在json解析量并不是很大的情况下，本库的运行速度和效率在总体而言还是位于中等偏上的水平的。

对于json注释（comment）的处理，尽管json文件一般不允许在键值对的后面加//注释，但是在语言模式为JSON with comments的情况下，允许在其他行的后面加//注释，此时用文件流读取则会在输出中报错。解决方法是将原来的json文件中的注释删除。如果可以在读取部分加上对于注释的处理或忽略的步骤的话，那么此种环境下的json文件也可读取。

https://github.com/miloyip/nativejson-benchmark

谢谢！

参考资料

https://www.json.org/json-zh.html

https://www.runoob.com/js/js-tutorial.html

https://www.runoob.com/json/json-tutorial.html

https://github.com/nlohmann/json

https://github.com/miloyip/nativejson-benchmark

https://json.nlohmann.me/