JsonOOLib 介绍

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。它使得人们很容易的进行阅读和编写，同时也方便了机器进行解析和生成。JsonOOLib 是一个 C++语言编写的、基于面向对象思想的 json 解析生成库。本文将主要介绍如何使用 JsonOOLib。

如果要使用 JsonOOLib，只需要包含头文件”JsonOOLib.h”,这个头文件将所需要的其他头文件包含进来了。JsonOOLib 包含多个类，都定义在命名空间 json 中，这意味着当你需要使用某个类（例如 String）时，需要以 json::String 的方式访问，或者提前包含 using 声明语句。

表 1、JsonOOLib 类	
String	表示一个 json **string** 对象
Number	表示一个 json **number** 对象
Object	表示一个 json **object** 对象
Array	表示一个 json **array** 对象
True	表示一个 json **true** 对象
False	表示一个 json **false** 对象
Null	表示一个 json **null** 对象

JsonOOLib 定义了多个类，分别对应 json 语言中的相应结构。每个类都包含 Parse/Serialize/Format 操作。其中，Parse 支持从一个 json 字符串中解析出 json 对象，Serialize 支持从 json 对象生成 json 字符串，Format 生成可读性更好的 json 字符串。

表 2、JsonOOLib 类通用操作	
Parse	从 json 字符串解析出 json 对象
Serialize	从 json 对象生成 json 字符串
Format	从 json 对象生成可读性更好的 json 字符串

示例 1：

```
typedef std::string JsonString;

JsonString js1 = "\"Hello\\tworld!\"";
String str = String::Parse(js1);
JsonString ret1 = str.Serialize();
JsonString fmt1= str.Format();

JsonString js2 = "3.1415926";
Number num = Number::Parse(js2);
JsonString ret2 = num.Serialize();
JsonString fmt2= num.Format();

JsonString js3 = "{\"a\": 1, \"b\": 2}";
Object obj = Object::Parse(js3);
JsonString ret3 = obj.Serialize();
JosnString fmt3= obj.Format();
……
```

解析一个 json 字符串时需要了解其结构，比如已知这是一个 Object 还是一个 Array，才能调用相应的函数来进行解析。例如，不能使用一个 Number::Parse 去解析 js1（见示例 1）。如果解析的字符串不符合 json 格式，则在运行时标准库会抛出一个类型为 JsonError 的异常。类似标准异常类型，JsonError 有一个 What 操作来描述发生了什么错误，以及一个名为 Code 的成员，用来返回某个错误类型对应的数值编码。

表 3、JsonError 错误类型

类型	Code	含义
error_empty	0	待解析字符串为空
error_escape	1	无效的转义字符
error_quote	2	不匹配的双引号
error_brack	3	不匹配的方括号
error_brace	5	不匹配的花括号
error_mismatch	6	不匹配的双引号/方括号/花括号
error_comma	7	不合法的逗号（位置）

error_pair	8	不合法的键值对
error_badnum	9	不合法的数字字符串
error_literal	10	错误的字面值（true/false/null）
deref_nullptr	11	试图使用一个空 Value 对象的数据
json_bad_cast	12	试图将 Value 解释成一个不匹配的 Json 类

JsonOOLib 类介绍

对于上述 JsonOOLib 类对象来说，它们除了包含 Parse/Serialize/Format 通用操作之外，每个类还包含针对各自类型的非通用操作，以及丰富多彩的初始化和赋值方式，用以操作不同的数据。

String

Stirng 可以从一个字符串字面值、以'\0'结尾的字符数组、string 对象进行初始化和赋值。需要注意的是，此处用来初始化和赋值的字符串均是普通的字符串，而不是 josn 字符串。在 Serialize/Format 操作中，String 会由这些保存的普通字符串生成符合 json 格式的字符串。String 还支持下表中的其他特定操作。

示例：

```
String str1 = "Hello,\tworld!"; // 初始化
String str2 = string(10, 'c');
str1 = "Hello,\tworld!";         // 赋值
str2 = string(10, 'c');
```

表 4、String 的特定操作	
void clear();	清空
std::string to_string() const;	返回 String 保存的普通字符串

Number

Number 可以从一个整数或者浮点数进行初始化和赋值，并将其保存为相应的类型，其

中小整数类型会被提升（bool、char、 short 等类型被提升为 int 或 unsigned int）。Number 还可以通过 to_开头的函数获取其保存的值。如果一个 to_函数的目标类型与 Number 保存的实际类型不一致，会进行算术类型转换。

示例：

```
Number n1 = 42, n2 = 42U, n3 = 42L, n4 = 42UL,       // 初始化
       n5 = 42LL, n6 = 42ULL, n7 = 3.14f, n8 = 3.14, n9 = 3.14L,
       n10(3.1415926f, 10), n11(3.14, 10), n12(90.8L, 2);
n1 = 42, n1 = 42U, n1 = 42L, n1 = 42UL, n1 = 42LL,  // 赋值
n1 = 42ULL, n1 = 3.14f, n1 = 3.14, n1 = 3.14L;
```

表 5、Number 的特定操作	
int to_int() const;	返回 Number 保存的 int
unsinged int to_uint() const;	返回 Number 保存的 unsigned int
long to_long() const;	返回 Number 保存的 long
unsigned long to_ulong() const;	返回 Number 保存的 unsigned long
long long to_longlong() const;	返回 Number 保存的 long long
unsigned long long to_ulonglong() const;	返回 Number 保存的 unsigned long long
float to_float() const;	返回 Number 保存的 float
double to_double() const;	返回 Number 保存的 double
long double to_longdouble() const;	返回 Number 保存的 long double

Object

Object 可以从花括号列表包围的元素，或一对迭代器进行初始化，也可以由花括号列表包围的元素进行赋值。由于 Object 的内部数据结构实现是一个 map，所以支持几乎所有 map 操作，并且意义与 map 操作一致。

示例：

```
Object obj1 = {{"a",1}, {"b", 2.0}, {"c", true}}, // 初始化
       obj2(obj1.begin(), obj1.end());
obj1 = {{"a",1}, {"b", 2.0}, {"c", true}};         // 赋值
```

Array

Array 可以从花括号列表包围的元素、一对迭代器、或一个整数值和一个可选的 Value（后面介绍）对象进行初始化，也可以由花括号列表包围的元素进行赋值。同样，由于 Array 的内部数据结构实现是一个 vector，所以支持几乎所有 vector 操作，并且意义与 vector 操作一致。

示例：

```
Array arr1 = {1, "a", 2.0}, // 初始化
      arr2(arr1.begin(), arr1.end()), arr3(10), arr4(10, "abc");
arr3 = {1, "a", 2.0};       // 赋值
```

此外，Array 和 Object 还支持更复杂的嵌套的列表初始化/赋值方法：例如：

```
Array arr = {{1,2,3}, {{"a", {1, 2}}, {"b", 3.14}}, 1};
```

元素 1 是一个 Array； 元素 2 是一个 Object，内部还嵌套了 Array 结构； 元素 3 是一个 Number。此外，嵌套层数没有限制。

True/False/Null

True/False/Null 对应 json 语言中的字面值 true/false/null 结构，只能通过默认构造函数进行初始化。

Value 类介绍

前面介绍的 Json00Lib 看似已经能满足对 json 字符串的操作，其实还有一个问题没有解决。在 json 语言中，array 结构中的元素和 object 结构中的值是一个 value，该 value 可以是任何 json 结构。类似的，Json00Lib 的 Array 类的元素以及 Object 类的值也应该可以是任何一个 Json00Lib 类的对象。库定义了 Value 类来支持这种特性，Value 对象可以表示任意一个 Json00Lib 类的对象。下面将详细介绍 Value 类。

Value 类的 Parse/Serialize/Format

由于 Value 类可以表示任何一个 Json00Lib 类，所以 Value 的 Parse 支持从任何 json

字符串解析，也可以通过 Serialize/Format 生成 json 字符串。

Value 类初始化和赋值

Value 可以由各种类型的数据进行初始化，并根据初始化数据的类型创建表示不同 JsonOOLib 对象的 Value。具体见表 6。

表 6、初始化一个 Value	
Value()	默认初始化，生成一个包含 Null 对象的 Value
Value(bool)	生成一个包含 True 或 False 对象的 Value
Value(int)	生成一个包含 Number 对象的 Value
Value(unsigned int)	生成一个包含 Number 对象的 Value
Value(long)	生成一个包含 Number 对象的 Value
Value(unsigned long)	生成一个包含 Number 对象的 Value
Value(unsigned long long)	生成一个包含 Number 对象的 Value
Value(float)	生成一个包含 Number 对象的 Value
Value(double)	生成一个包含 Number 对象的 Value
Value(long double)	生成一个包含 Number 对象的 Value
Value(float, int)	生成一个包含指定最大有效数字个数的 Number 对象的 Value
Value(double, int)	生成一个包含指定最大有效数字个数的 Number 对象的 Value
Value(long double, int)	生成一个包含指定最大有效数字个数的 Number 对象的 Value
Value(const string &)	生成一个包含 String 对象的 Value
Value(string &&)	生成一个包含 String 对象的 Value
Value(const char *)	生成一个包含 String 对象的 Value
Value("std::initialzier_list")	生成一个包含 Object 或 Array 对象的 Value
Value("JsonOOLib")	生成一个包含相应 JsonOOLib 对象的 Value

对于一个初始化列表，Value 会尽可能将其解释为一个 Object，如果不能，Value 才会将其解释为一个 Array。例如 Value = {{“a”, 1}, {“b”, 2}}; 将得到一个 Object。而 Value = {1, 2}; 将得到一个 Array。有一点需要注意的是：对于 Value v = {}这种初始化方式（花括号为空），C++将其解释为**值初始化**，所以 Value 中将包含一个 Null 对象。这是本库中唯一一个与直觉不符的地方。

Value 同样支持通过上述类型的数据进行赋值。赋值会销毁 Value 中的已有对象，并保存一个新的对象。所以，赋值前后 Value 中的 JsonOOLib 对象类型可能不同。

Value 的类型转换

Value 可以表示任何一个 JsonOOLib 对象，所以不能确定 Value 中保存的对象的实际类型。所以 Value 仅支持初始化、赋值、Parse/Serialize/Format 等少数几个操作。如果需要针对特定 JsonOOLib 类型的操作,可以通过类型转换函数,先将其转换至具体的 JsonOOlib 类对象，然后再进行更多特定的操作。

表 7、Value 中类型转换函数	
String **to_String**() const	转换成一个 String
Number **to_Number**() const	转换成一个 Number
Object **to_Objecvt**() const	转换成一个 Object
Array **to_Array**() const	转换成一个 Array
True **to_True**() const	转换成一个 True
False **to_False**() const	转换成一个 False
Null **to_Null**() const	转换成一个 Null

例如，如果 Value 中保存的是一个 String 对象，可以使用 to_String()将其转换为一个 String，而不能转换成其他类型的对象。如果 Value 保存的对象类型与转换的目标类型不同，则抛出一个 JsonError(json_bad_cast)异常。如果不确定 Value 的类型，可以使用 is_开头的函数进行判断。

表 8、判断 Value 的类型	
bool **is_String**() const	Value 是否是一个 String
bool **is_Number**() const	Value 是否是一个 Number
bool **is_Objecvt**() const	Value 是否是一个 Object
bool **is_Array**() const	Value 是否是一个 Array
bool **is_True**() const	Value 是否是一个 True
bool **is_False**() const	Value 是否是一个 False
bool **is_Null**() const	Value 是否是一个 Null

如果你更喜欢用 switch 语句，JsonOOLib 还提供了 Type()函数进行类型判断，Type()返回枚举值： string_type、number_type、object_type、array_type、true_type、false_type、null_type 中的一种。

示例：

现有一个存放学生成绩的 json 格式文本文件，类似如下内容：

{“Zhang 3”: 90, “Li 4”:89, “Wang 5”: null}

json 格式是一个学生姓名到成绩的映射，如果学生有成绩，则是一个 0-100 的整数，如果没有成绩，则是一个 null。

现在已经将 json 文件内容读取到字符串 file_content 中。设计一个函数，接受一个表示学生成绩的 json 字符串和学生姓名，返回该学生成绩。如果学生成绩不存在，则返回-1。如果学生不存在，则返回-2。

```
int find_score(const string &file_content, const string &name)
{
    try{
        auto scores = Object::Parse(file_content);
        auto iter = scores.find(name);
        if(iter != scores.cend())
            if(iter->second.is_Null())
                return -1;
            else
                return iter->second.to_Number().to_int();（1）
    } catch(JsonError e){
```

```
            cout << e.What() << endl;
        }
        return -2;
    }
```

从 Value 中获取一个整型值，可以这样实现：

```
auto number = iter->second.to_Number();
return number.to_int();
```

如果只是偶尔需要这样做，可以像（1）处写在一条表达式中。Value 也提供了简便的写法，直接调用 to_int 来达到上述功能。例如（1）处也可以这样实现：

```
return iter->second.to_int();
```

表 9、从 Value 中获取一个值	
std::string to_string() const	从包含 String 的 Value 中获取其中的字符串
int to_int() const	从包含 Number 的 Value 中获取其中的数值
unsigned int to_uint() const	从包含 Number 的 Value 中获取其中的数值
long to_long() const	从包含 Number 的 Value 中获取其中的数值
unsigned long to_ulong() const	从包含 Number 的 Value 中获取其中的数值
long long to_longlong() const	从包含 Number 的 Value 中获取其中的数值
unsigned long long to_ulonglong() const	从包含 Number 的 Value 中获取其中的数值
float to_float() const	从包含 Number 的 Value 中获取其中的数值
double to_double() const	从包含 Number 的 Value 中获取其中的数值
long double to_longdouble() const	从包含 Number 的 Value 中获取其中的数值

表 9 中的函数使得从 Value 中获取其保存的对象的值更加的便捷，不过如果 Value 中包含的类型不支持该函数，则会抛出一个 JsonError(json_bad_cast)的异常。例如对一个包含 String 的 Value 对象调用 to_int 将会抛出异常。