RapidJson 사용 방법

C++에서 json을 처리하기 위해 library를 찾아보다가 rapidjson이 성능이 좋다고 들었다. 샘플과 구글링을 통해 필요한 기능을 여차저차 구현하였지만, 훗날 rapidjson을 다시 사용할 일이 생긴다면 사용법을 이해하는데 시간이 걸릴 것이다. 그래서 나중에 쉽게 참고할 수 있도록 이해한 것들을 기록으로 남겨두고자 한다.(문서 버전: v230223)

RapidJson 특징

- C++ 코드로 작성, header파일만 포함시켜서 빌드 가능, 별도 외부 library 종속성 없음
- JSON 처리 속도 우수, JSON 문법 호환성도 양호한 편
- 입력스트림의 문자열을 자동으로 감지하여 처리, 기본 문자형은 UTF8 인코딩
- MIT라이선스로 상업 목적으로 사용 가능, 제작자는 중국 Tencent 업체 직원

주의점

- 이 문서는 rapidjson v1.1.0을 기준으로 설명함
- json문자열내 멤버명와 문자열값은 모두 큰 따옴표("")로 감싸져야 한다.
  특히 파이썬 애플리케이션과 JSON데이터를 교환시 꼭 큰따옴표를 사용하자.
- json문자열내 주석(#)이 포함된 경우 파싱이 실패하게 된다.

```
{
# 2023-02-21 Generated by XX System
 "project": "랜섬웨어 대응 훈련",
...
```

- json문자열값에 역슬래쉬(Back-Slash)문자를 포함해야된다면 연속된 역슬래쉬 2개 문자로 표현해야 함.
  문자열값에 포함된 역슬래쉬는 Escape문자로 인식되어 파싱에 실패할 수 있다.

```
{"username": "dragon\user"} 인 경우, {"username": "dragon\\user"} 로 변환 사용
```

- rapidjson은 문자열을 유니코드로 처리하기 때문에 GetString()이 반환한 문자열 포인터를 strlen() 함수로 길이를 구해서는 안된다. 상황에 따라 정상적인 유니코드 문자가 NULL문자로 해석되어 정확한 길이를 반환하지 않는 경우가 존재하기 때문이다. 정확한 문자열 길이를 구한다면 GetStringLength() 함수를 이용해야 한다.

```
Value& v = doc["SomeMemberName"];
int len1 = strlen(v.GetString());
int len2 = v.GetStringLength();
// 문자열이 "a\u0000b" 이라면, len1은 1이 저장되고, len2는 3이 저장됨
```

- **(Move semantics)** Value객체를 통해 좌변으로 할당시 우변의 값은 없어지므로(Null 초기화된 객체), 할당후 재사용 금지.
  성능 향상을 위해 대입연산시 깊은 복사가 발생하지 않도록 Library가 설계됨

```
Value a(100);
Value b(200);
a = b;        // 우변인 b는 Null이 설정됨, a는 100값이 설정됨
```

"Move Semantics"가 되는 경우는 Value객체의 대입 연산(operator=) 외에도 AddMember(), PushBack() 함수 호출시에도 동일하게 적용된다. 한번 인자로 전달된 Value객체는 Null초기화되므로, 임시 Value객체를 만들어 전달하는 것이 편리하다. 이런 경우 임시 개체를 적절한 값 참조로 변환할 수 없으므로 Move() 함수를 통해 전달해야 한다고 한다.

> Value arr(kArrayType);
>
> arr.PushBack(Value().SetInt(42), d.GetAllocator()); // fluent API style
> **arr.PushBack(Value(43).Move(), d.GetAllocator()); // same as above**
> **arr.PushBack(Value("Hello").Move(), d.GetAllocator());**

- **(문자열 데이터 보관 전략)** 상수 문자열(const string)의 포인터를 Value객체에서 보관하는 방법과, 내부에 복사본을 만들어 보관하는 방법이 있으니 구현시 참고하자. 다음 예제 소스에서는 str2와 str3이 모두 outlive_string 을 참조하는데, str2는 상수 문자열로 취급하여 포인터만 보관하였고, str3은 내부에 복사본을 더이상 참조하지 않는 전략을 사용했다. 결과적으로 str3은 outlive_string에 변경이 발생하여도 내부 데이터에는 변경이 발생하지 않게 되었다.

```
void string_strategy()
{
    CHAR outlive_string[] = "hello world";

    Document doc(kObjectType);

    Value str1;
    Value str2;
    Value str3;

    //  문자열 보관 전략 1) 상수 문자열 포인터 보관
    str1.SetString("hello world");

    //  문자열 보관 전략 2) 상수 문자열은 아니지만,
    //  상수 문자열 포인터로 취급하여 보관
    str2.SetString(StringRef(outlive_string));

    //  문자열 보관 전략 3) Value객체에 문자열을 복사하여 보관
    str3.SetString(outlive_string, doc.GetAllocator());

    _ASSERTE( str1 == "hello world" );
    _ASSERTE( str2 == "hello world" );
    _ASSERTE( str3 == "hello world" );

    //  str2, str3이 참조한 outlive_string에 변조 발생!
    //  str2, str3에 담긴 데이터는 문제없을까?
    strncpy(outlive_string, "HELLO", 5);

    _ASSERTE( str2 != "hello world" );
    _ASSERTE( str3 == "hello world" );
}
```

활용 팁

본 문서 설명에 사용된 샘플 JSON 문자열(UTF-8 인코딩된 문자열로 가정)은 다음과 같다.

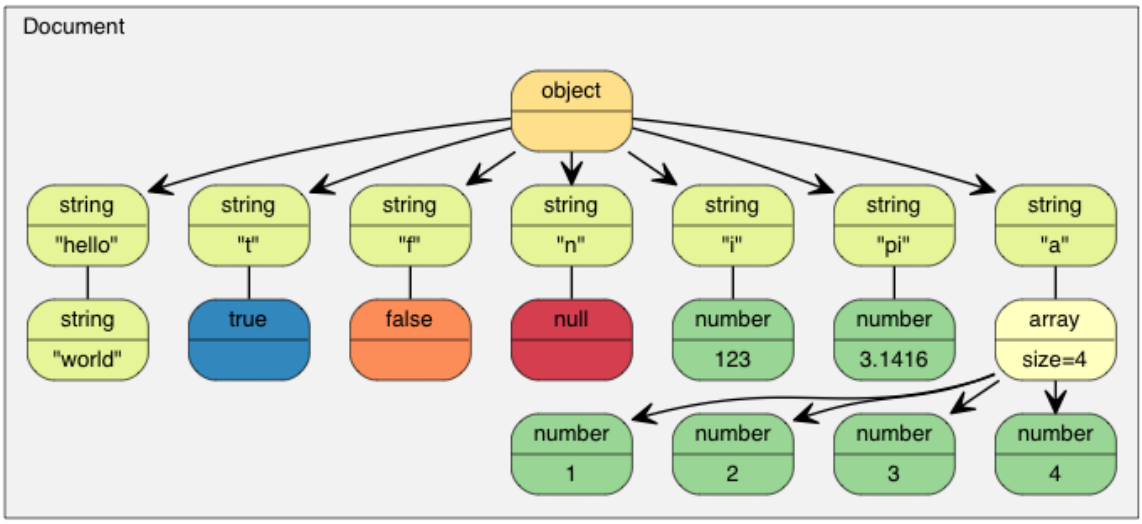
```
{
    "project": "랜섬웨어 대응 훈련",
    "localip": "192.168.11.17",
    "macaddr": "ab-cd-ef-hi-jk-lm",
    "osver": "Windows 10",
    "buildnum": 19044,
    "hostname": "dragon\\user",
    "username": "user"
}
```

1. rapidjson 헤더 포함 및 namespace 정의

```
#include "./rapidjson/document.h"
#include "./rapidjson/writer.h"
#include "./rapidjson/prettywriter.h"
#include "./rapidjson/stringbuffer.h"
#include "./rapidjson/pointer.h"
#include "./rapidjson/rapidjson.h"
using namespace rapidjson;
```

2. Value와 Document 객체 소개

rapidjosn은 파싱된 JSON문서 구조는 최상단 루트 Value가 자식 Value들을 포함하는 구조이다. 최상단 루트 Value는 사실 Document 클래스가 되며, Value클래스가 제공하는 멤버관리 기능을 동일하게 제공하면서 JSON문서를 파싱할 수 있는 기능도 제공한다.



Value클래스는 저장될 수 있는 데이터 타입은 다음과 같다. 이 중 Object타입과 Array타입은 또다른 멤버 또는 요소를 포함할 수 있는 컬렉션 타입이다.

```
static const char* kTypeNames[] =
{ "Null", "False", "True", "Object", "Array", "String", "Number" };
```

Value클래스는 데이터 타입을 확인할 수 있는 IsXXX() 함수들과, 데이터에 접근/설정할 수 있는 GetXXX(), SetXXX() 함수들을 제공한다(컬렉션은 멤버 및 요소를 다루는 함수가 별도 제공).

```
Value nullObj; // 타입을 지정하지 않으면 Null 타입으로 초기화 된다
Value obj(kObjectType);
Value arr(kArrayType);
Value numInt(1);
numInt.SetInt(2);
Value numDouble(3.14);

_ASSERTE(nullObj.IsNull() == TRUE);
_ASSERTE(obj.IsObject() == TRUE);
_ASSERTE(arr.IsArray() == TRUE);
_ASSERTE(numInt.GetInt() == 2);
_ASSERTE(numDouble.GetDouble() == 3.14);
```

3. Json문자열을 Document 객체로 변환후 멤버 접근하기

rapidjson에서 문자열은 기본적으로 UTF8 인코딩 처리하므로, Document객체의 Parse()함수 호출시 UTF8 인코딩된 json문자열을 전달해야 올바르게 파싱할 수 있다. 또는 프로젝트에 맞는 인코딩을 typedef문으로 정의하여 사용할 수 도 있다.

```
CStringA jsonStr = CT2A(strReport, CP_UTF8);

Document doc;
doc.Parse(jsonStr);
```

Document 객체에서 hostname, username 등 JSON 멤버에 대한 값을 구한다. 이때도 UTF8로 처리된 것을 현재 VC프로젝트 기본문자열 타입으로 변환시켜주어야 한다.

```
CString project = CA2T(doc["project"].GetString(), CP_UTF8);
CString hostname = CA2T(doc["hostname"].GetString(), CP_UTF8);
CString username = CA2T(doc["username"].GetString(), CP_UTF8);
_tprintf(_T("project: %s\n"), project);
_tprintf(_T("hostname: %s\n"), hostname);
_tprintf(_T("username: %s\n"), username);
```

위 예시는 간단하게 c++에서 json문자열을 로딩하여 멤버에 접근하는 방법을 설명하였지만, 개발시에는 다양한 상황을 고려하여 에러 체크를 강화해야할 것이다.

```
bool basic_json_string_load()
{
    CStringA jsonStr = CT2A(strReport, CP_UTF8);

    Document doc;
    ParseResult ok = doc.Parse(jsonStr);
    if (!ok)
    {
        _tprintf(_T("JSON parse error: %d (offset %u)\n"), ok.Code(),
ok.Offset());
        _tprintf(_T("%s\n"), strReport);
        return false;
    }
```

```
    if (!doc.IsObject())
        return false;
    //  혹은 상황에 따라 Document가 Array로 시작하는 경우를 체크함
    //if (!doc.IsArray())
    //  return false;

    //  멤버에 접근하기 위해 멤버 포함여부와 타입 체크
    _ASSERTE(doc.HasMember("project"));
    _ASSERTE(doc["project"].IsString() == TRUE);

    //  필요한 멤버에 접근
    CString project = CA2T(doc["project"].GetString(), CP_UTF8);
    _tprintf(_T("project: %s\n"), project);
    ... 생략 ...

    return true;
}
```

4. Document객체를 Json 문자열로 변환하기

Document객체를 전송(저장)하기 위해 Json문자열로 변환시 다음 코드를 참고하자. PrettyWriter대신 Writer 클래스를 사용하면, 포맷터 적용없이 Json 문자열로 변환된다.

```
void print_json_prettify(Document& doc)
{
    StringBuffer buffer;
    PrettyWriter<StringBuffer> writer(buffer);
    doc.Accept(writer);

    CString jsonReport = CA2T(buffer.GetString(), CP_UTF8);
                        _tprintf(_T("%s\n"), jsonReport);
}
```

5. Document 객체내 멤버에 Pointer로 손쉽게 추가/수정/삭제 하기

Document 객체안 멤버 관리는 Pointer사용하는 것이 가장 간편한 방법 같다. rapidjson 제작자도 이를 위해 헬퍼(Helper) 함수를 제공였다. CreateValueByPointer(), EraseValueByPointer(), GetValueByPointer(), SetValueByPointer() 함수를 활용해 멤버들을 관리할 수 있다.

다음 예제는 'project', 'buildnum' 멤버값을 변경하고, Number타입을 가지는 새로운 'userseq'멤버를 추가하였다. 이후 'macaddr' 멤버를 삭제하고 'hello/there' 오브젝트 멤버와 'hello/world' 배열 멤버를 새롭게 생성하였다.

```
    //  project 멤버값 변경
    //  SetValueByPointer() 구현이 인자로 전달된 문자열을 내부에 복사하여 보관 처리됨
    //  물론 Value("문자열").Move() 형태로도 전달 가능
    SetValueByPointer(doc, "/project", (LPCSTR)CT2A(_T("RapidJSON 테스트"),
CP_UTF8));

    if (Value* pBuildNum = GetValueByPointer(doc, "/buildnum"))
        pBuildNum->SetInt(pBuildNum->GetInt() + 1000);

    //SetValueByPointer(doc, "/userseq", 19214598);     // will not compile
```

```
    SetValueByPointer(doc, "/userseq", Value(19214598).Move());

    //  macaddr 멤버 삭제
    EraseValueByPointer(doc, "/macaddr");

    //  새로운 멤버 및 자식멤버까지 동시 생성 -> there멤버는 null로 최초 할당됨
    CreateValueByPointer(doc, "/hello/there");

    //  새로운 멤버 배열 생성 -> world멤버배열은 [null]로 할당됨
    CreateValueByPointer(doc, "/hello/world/0");
```

6. 오브젝트(Object) 관리하기

오브젝트 자료형은 키-값 쌍으로 구성된 컬렉션으로, 오브젝트내 각 멤버의 키는 문자열형을 사용해야 한다. 멤버를 추가할 수 있는 AddMember() 함수와 삭제할 수 있는 RemoveMember(), EraseMember()를 사용할 수 있다.

```
void work_with_json_object(Document& doc)
{
    Value o(kObjectType);

    //  오브젝트에 멤버 추가
    o.AddMember("string", "hello world", doc.GetAllocator());
    o.AddMember("message", "rapidjson is powerful", doc.GetAllocator());
    o.AddMember("int", 123, doc.GetAllocator());
    o.AddMember("double", 3.14, doc.GetAllocator());
    o.AddMember("arr", Value(kArrayType).Move(), doc.GetAllocator());

    //  멤버 삭제
    o.RemoveMember("message");

    static const char* kTypeNames[] =
    { "Null", "False", "True", "Object", "Array", "String", "Number" };

    //  반복자를 통해서 Object에 포함된 모든 멤버에 접근
    for (Value::ConstMemberIterator itr = o.MemberBegin();
        itr != o.MemberEnd(); ++itr)
    {
        printf("Type of member %s is %s\n",
            itr->name.GetString(), kTypeNames[itr->value.GetType()]);
    }

    //  Document(doc)객체에 Value(o)객체를 추가
    doc.AddMember("new_json_object", o.Move(), doc.GetAllocator());
    //  Value o 객체의 데이터는 doc객체로 Move Semantics되므로, Null초기화 됨
    _ASSERTE(o.IsNull() == TRUE);

    print_json_prettify(doc);
}
```

7. 배열(Array) 요소 관리하기

배열 관리 방식은 std::vector 클래스를 사용하는 방식과 비슷한 API를 제공하므로 어렵지 않게 사용할 수 있다.

```
void work_with_json_array(Document& doc)
{
    Value& arr = doc["hello"]["world"];

    //  배열 초기화
    arr.Clear();

    //  새로운 값 추가
    arr.PushBack("First Element", doc.GetAllocator());

    for (SizeType i = 1; i <= 3; ++i)
        arr.PushBack(i, doc.GetAllocator());

    arr.PushBack(Value("Last Element").Move(), doc.GetAllocator());

    //  첫번째 요소 삭제
    arr.Erase(arr.Begin());

    _tprintf(_T("====== 배열인덱스를 통해 배열 접근 =======\n"));

    //  0부터 arr배열의 크기-1까지 i값을 증가시키면서 배열 인덱스를 통해 접근
    for (SizeType i = 0; i < arr.Size(); ++i)
    {
        const Value& v = arr[i];

        if (v.IsString())
        {
            _tprintf(_T("%d: \"%s\"\n"), i, CA2T(v.GetString(), CP_UTF8));
        }
        else if (v.IsInt())
        {
            _tprintf(_T("%d: %d\n"), i, v.GetInt());
        }
    }

    _tprintf(_T("\n====== 반복자를 통해 배열 접근 =======\n"));

    //  std::vector 사용하는 것처럼, iterator를 통해 접근하는 방법
    for (Value::ConstValueIterator itr = arr.Begin(); itr != arr.End(); ++itr)
    {
        if (itr->IsString())
        {
            _tprintf(_T("\"%s\"\n"), CA2T(itr->GetString(), CP_UTF8));
        }
        else if (itr->IsInt())
        {
            _tprintf(_T("%d\n"), itr->GetInt());
        }
    }
}
```

마치며

이 문서는 가장 기초적인 rapidjson 사용 방법에 대해서만 설명하였지만, 심도깊은 내용(유니코드 문자열 인코딩 처리, SAX-style API 처리방법 등)은 공식 사이트 강좌(Tutorial)를 참고하기 바란다.

참고 자료

- rapidjson 공식 사이트
- RapidJson NEXT 김명찬
- RapidJSONのメモ
- When should you use CrtAllocator vs MemoryPoolAllocator in RapidJSON?