# 目錄

# RapidJSON

release v1.1.0

## 高效的 C++ JSON 解析／生成器，提供 SAX 及 DOM 风格 API



- RapidJSON GitHub
- RapidJSON 文档
  - English
  - 简体中文
  - GitBook 可下载 PDF/EPUB/MOBI，但不含 API 参考手册。

## Build 状态

| Linux | Windows | Coveralls |
| --- | --- | --- |
| build passing | build pending | coverage 100% |

## 简介

RapidJSON 是一个 C++ 的 JSON 解析器及生成器。它的灵感来自 RapidXml。

- RapidJSON 小而全。它同时支持 SAX 和 DOM 风格的 API。SAX 解析器只有约 500 行代码。
- RapidJSON 快。它的性能可与 `strlen()` 相比。可支持 SSE2/SSE4.2 加速。
- RapidJSON 独立。它不依赖于 BOOST 等外部库。它甚至不依赖于 STL。
- RapidJSON 对内存友好。在大部分 32/64 位机器上，每个 JSON 值只占 16 字节（除字符串外）。它预设使用一个快速的内存分配器，令分析器可以紧凑地分配内存。
- RapidJSON 对 Unicode 友好。它支持 UTF-8、UTF-16、UTF-32 (大端序／小端序)，并内部支持这些编码的检测、校验及转码。例如，RapidJSON 可以在分析一个 UTF-8 文件至 DOM 时，把当中的 JSON 字符串转码至 UTF-16。它也支持代理对（surrogate pair）及 `"\u0000"`（空字符）。

在 这里 可读取更多特点。

JSON（JavaScript Object Notation）是一个轻量的数据交换格式。RapidJSON 应该完全遵从 RFC7159/ECMA-404，并支持可选的放宽语法。 关于 JSON 的更多信息可参考：

- Introducing JSON
- RFC7159: The JavaScript Object Notation (JSON) Data Interchange Format
- Standard ECMA-404: The JSON Data Interchange Format

## v1.1 中的亮点 (2016-8-25)

- 加入 JSON Pointer 功能，可更简单地访问及更改 DOM。
- 加入 JSON Schema 功能，可在解析或生成 JSON 时进行校验。
- 加入 放宽的 JSON 语法 （注释、尾随逗号、NaN/Infinity）
- 使用 C++11 范围 for 循环 去遍历 array 和 object。
- 在 x86-64 架构下，缩减每个 `Value` 的内存开销从 24 字节至 16 字节。

其他改动请参考 change log.

## 兼容性

RapidJSON 是跨平台的。以下是一些曾测试的平台／编译器组合：

- Visual C++ 2008/2010/2013 在 Windows (32/64-bit)
- GNU C++ 3.8.x 在 Cygwin
- Clang 3.4 在 Mac OS X (32/64-bit) 及 iOS
- Clang 3.4 在 Android NDK

用户也可以在他们的平台上生成及执行单元测试。

## 安装

RapidJSON 是只有头文件的 C++ 库。只需把 `include/rapidjson` 目录复制至系统或项目的 include 目录中。

RapidJSON 依赖于以下软件：

- CMake 作为通用生成工具
- (optional) Doxygen 用于生成文档
- (optional) googletest 用于单元及性能测试

生成测试及例子的步骤：

1. 执行 `git submodule update --init` 去获取 thirdparty submodules (google test)。
2. 在 rapidjson 目录下，建立一个 `build` 目录。
3. 在 `build` 目录下执行 `cmake ..` 命令以设置生成。Windows 用户可使用 cmake-gui 应用程序。
4. 在 Windows 下，编译生成在 build 目录中的 solution。在 Linux 下，于 build 目录运行 `make` 。

成功生成后，你会在 `bin` 的目录下找到编译后的测试及例子可执行文件。而生成的文档将位于 build 下的 `doc/html` 目录。要执行测试，请在 build 下执行 `make test` 或 `ctest` 。使用 `ctest -V` 命令可获取详细的输出。

我们也可以把程序库安装至全系统中，只要在具管理权限下从 build 目录执行 `make install` 命令。这样会按系统的偏好设置安装所有文件。当安装 RapidJSON 后，其他的 CMake 项目需要使用它时，可以通过在 `CMakeLists.txt` 加入一句 `find_package(RapidJSON)` 。

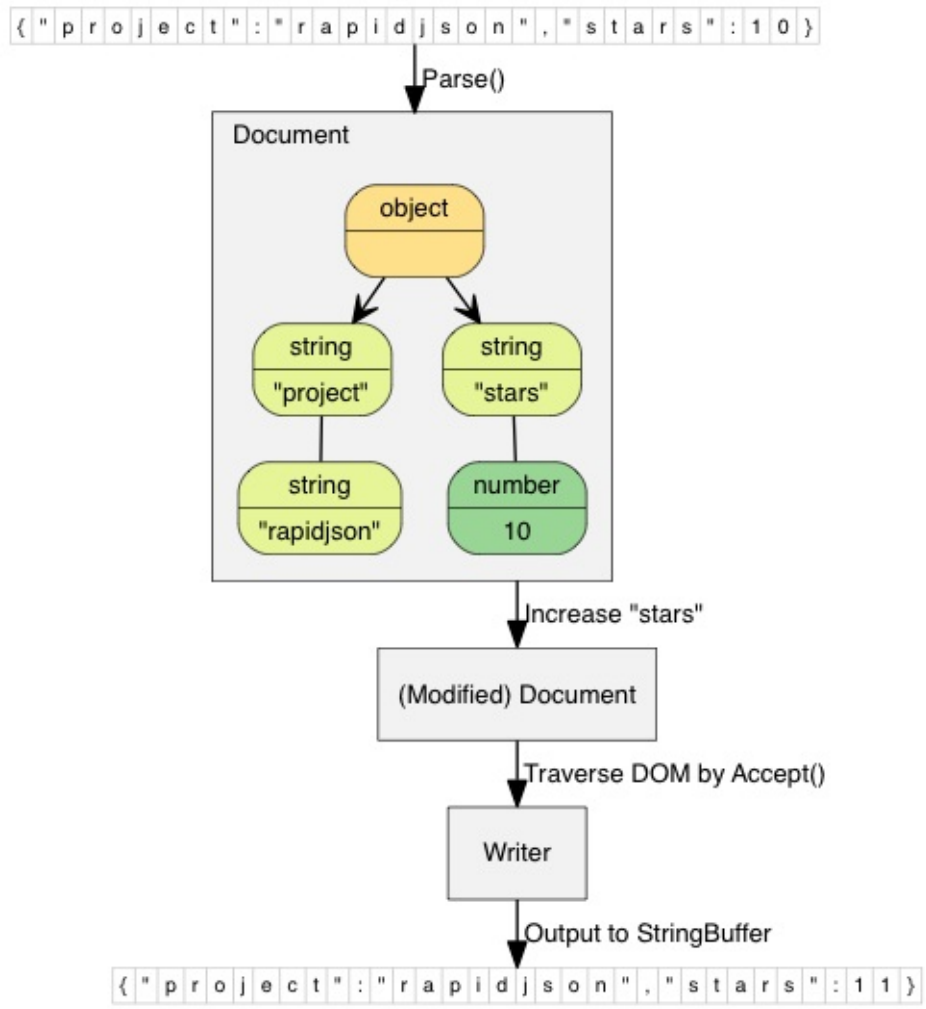
## 用法一览

此简单例子解析一个 JSON 字符串至一个 document (DOM)，对 DOM 作出简单修改，最终把 DOM 转换（stringify）至 JSON 字符串。

```
// rapidjson/example/simpledom/simpledom.cpp`
#include "rapidjson/document.h"
#include "rapidjson/writer.h"
#include "rapidjson/stringbuffer.h"
#include <iostream>

using namespace rapidjson;

int main() {
    // 1. 把 JSON 解析至 DOM。
    const char* json = "{\"project\":\"rapidjson\",\"stars\":10}";
    Document d;
    d.Parse(json);

    // 2. 利用 DOM 作出修改。
    Value& s = d["stars"];
    s.SetInt(s.GetInt() + 1);

    // 3. 把 DOM 转换（stringify）成 JSON。
    StringBuffer buffer;
    Writer<StringBuffer> writer(buffer);
    d.Accept(writer);

    // Output {"project":"rapidjson","stars":11}
    std::cout << buffer.GetString() << std::endl;
    return 0;
}
```

注意此例子并没有处理潜在错误。

下图展示执行过程。




还有许多 例子 可供参考：

- DOM API
  - tutorial: DOM API 的基本使用方法。
- SAX API
  - simplereader: 使用 `Reader` 解析 JSON 时，打印所有 SAX 事件。
  - condense: 移除 JSON 中所有空白符的命令行工具。
  - pretty: 为 JSON 加入缩进与换行的命令行工具，当中使用了 `PrettyWriter`。
  - capitalize: 把 JSON 中所有字符串改为大写的命令行工具。
  - messagereader: 使用 SAX API 去解析一个 JSON 报文。
  - serialize: 使用 SAX API 去序列化 C++ 对象，生成 JSON。
  - jsonx: 实现了一个 `JsonxWriter`，它能把 SAX 事件写成 JSONx（一种 XML）格式。这个例子是把 JSON 输入转换成 JSONx 格式的命令行工具。
- Schema API
  - schemavalidator: 使用 JSON Schema 去校验 JSON 的命令行工具。
- 进阶
  - prettyauto: pretty 的修改版本，可自动处理任何 UTF 编码的 JSON。
  - parsebyparts: 这例子中的 `AsyncDocumentParser` 类使用 C++ 线程来逐段解析 JSON。
  - filterkey: 移取使用者指定的键值的命令行工具。

- filterkeydom: 如上的工具，但展示如何使用生成器（generator）去填充一个 `Document` 。

# 特点

## 总体

- 跨平台
  - 编译器：Visual Studio、gcc、clang 等
  - 架构：x86、x64、ARM 等
  - 操作系统：Windows、Mac OS X、Linux、iOS、Android 等
- 容易安装
  - 只有头文件的库。只需把头文件复制至你的项目中。
- 独立、最小依赖
  - 不需依赖 STL、BOOST 等。
  - 只包含 `<cstdio>`，`<cstdlib>`，`<cstring>`，`<inttypes.h>`，`<new>`，`<stdint.h>`。
- 没使用 C++ 异常、RTTI
- 高性能
  - 使用模版及内联函数去降低函数调用开销。
  - 内部经优化的 Grisu2 及浮点数解析实现。
  - 可选的 SSE2/SSE4.2 支持。

## 符合标准

- RapidJSON 应完全符合 RFC4627/ECMA-404 标准。
- 支持 JSON Pointer (RFC6901).
- 支持 JSON Schema Draft v4.
- 支持 Unicode 代理对（surrogate pair）。
- 支持空字符（`"\u0000"`）。
  - 例如，可以优雅地解析及处理 `["Hello\u0000World"]`。含读写字符串长度的 API。
- 支持可选的放宽语法
  - 单行（`// ...`）及多行（`/* ... */`）注释（`kParseCommentsFlag`）。
  - 在对象和数组结束前含逗号（`kParseTrailingCommasFlag`）。
  - `NaN`、`Inf`、`Infinity`、`-Inf` 及 `-Infinity` 作为 `double` 值（`kParseNanAndInfFlag`）
- NPM 兼容.

# Unicode

- 支持 UTF-8、UTF-16、UTF-32 编码，包括小端序和大端序。
  - 这些编码用于输入输出流，以及内存中的表示。
- 支持从输入流自动检测编码。
- 内部支持编码的转换。
  - 例如，你可以读取一个 UTF-8 文件，让 RapidJSON 把 JSON 字符串转换至 UTF-16 的 DOM。
- 内部支持编码校验。
  - 例如，你可以读取一个 UTF-8 文件，让 RapidJSON 检查是否所有 JSON 字符串是合法的 UTF-8

字节序列。

- 支持自定义的字符类型。
  - 预设的字符类型是：UTF-8 为 `char`，UTF-16 为 `wchar_t`，UTF32 为 `uint32_t`。
- 支持自定义的编码。

## API 风格

- SAX（Simple API for XML）风格 API
  - 类似于 SAX, RapidJSON 提供一个事件循序访问的解析器 API（`rapidjson::GenericReader`）。RapidJSON 也提供一个生成器 API（`rapidjson::Writer`），可以处理相同的事件集合。
- DOM（Document Object Model）风格 API
  - 类似于 HTML／XML 的 DOM，RapidJSON 可把 JSON 解析至一个 DOM 表示方式（`rapidjson::GenericDocument`），以方便操作。如有需要，可把 DOM 转换（stringify）回 JSON。
  - DOM 风格 API（`rapidjson::GenericDocument`）实际上是由 SAX 风格 API（`rapidjson::GenericReader`）实现的。SAX 更快，但有时 DOM 更易用。用户可根据情况作出选择。

### 解析

- 递归式（预设）及迭代式解析器
  - 递归式解析器较快，但在极端情况下可出现堆栈溢出。
  - 迭代式解析器使用自定义的堆栈去维持解析状态。
- 支持原位（*in situ*）解析。
  - 把 JSON 字符串的值解析至原 JSON 之中，然后让 DOM 指向那些字符串。
  - 比常规分析更快：不需字符串的内存分配、不需复制（如字符串不含转义符）、缓存友好。
- 对于 JSON 数字类型，支持 32-bit/64-bit 的有号／无号整数，以及 `double`。
- 错误处理
  - 支持详尽的解析错误代号。
  - 支持本地化错误信息。

## DOM (Document)

- RapidJSON 在类型转换时会检查数值的范围。
- 字符串字面量的优化
  - 只储存指针，不作复制
- 优化“短”字符串
  - 在 `Value` 内储存短字符串，无需额外分配。
  - 对 UTF-8 字符串来说，32 位架构下可存储最多 11 字符，64 位下 21 字符（x86-64 下 13 字符）。
- 可选地支持 `std::string`（定义 `RAPIDJSON_HAS_STDSTRING=1`）

### 生成

- 支持 `rapidjson::PrettyWriter` 去加入换行及缩进。

## 输入输出流

- 支持 `rapidjson::GenericStringBuffer`，把输出的 JSON 储存于字符串内。
- 支持 `rapidjson::FileReadStream` 及 `rapidjson::FileWriteStream`，使用 `FILE` 对象作输入输出。
- 支持自定义输入输出流。

## 内存

- 最小化 DOM 的内存开销。
  - 对大部分 32／64 位机器而言，每个 JSON 值只占 16 或 20 字节（不包含字符串）。
- 支持快速的预设分配器。
  - 它是一个堆栈形式的分配器（顺序分配，不容许单独释放，适合解析过程之用）。
  - 使用者也可提供一个预分配的缓冲区。（有可能达至无需 CRT 分配就能解析多个 JSON）
- 支持标准 CRT（C-runtime）分配器。
- 支持自定义分配器。

## 其他

- 一些 C++11 的支持（可选）
  - 右值引用（rvalue reference）
  - `noexcept` 修饰符
  - 范围 for 循环

# 教程

本教程简介文件对象模型（Document Object Model, DOM）API。

如 用法一览 中所示，可以解析一个 JSON 至 DOM，然后就可以轻松查询及修改 DOM，并最终转换回 JSON。

[TOC]

## Value 及 Document

每个 JSON 值都储存为 `Value` 类，而 `Document` 类则表示整个 DOM，它存储了一个 DOM 树的根 `Value`。RapidJSON 的所有公开类型及函数都在 `rapidjson` 命名空间中。

## 查询 Value

在本节中，我们会使用到 `example/tutorial/tutorial.cpp` 中的代码片段。

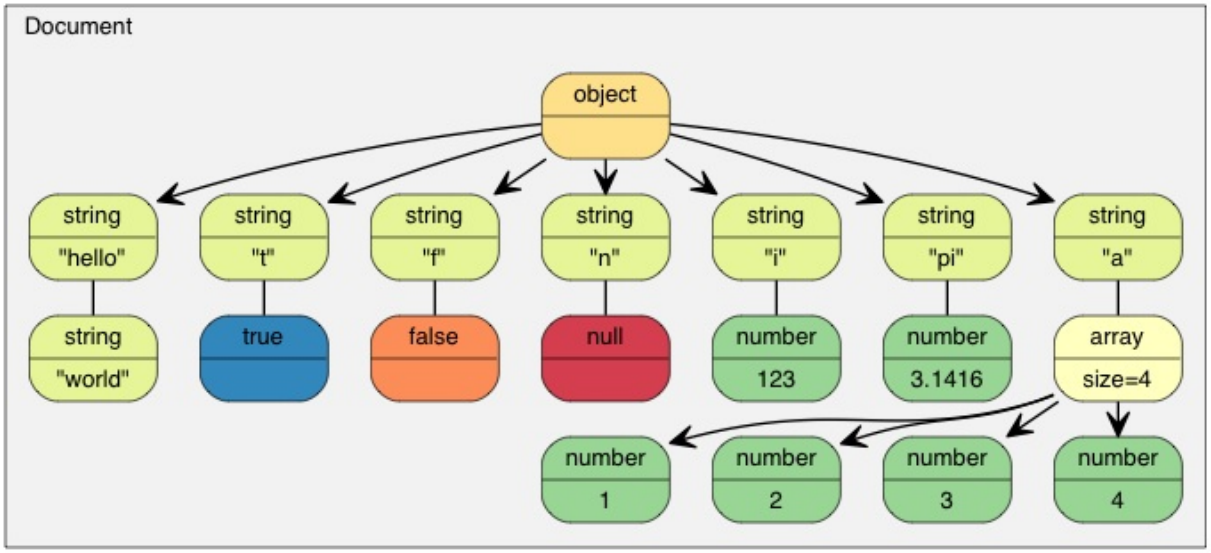
假设我们用 C 语言的字符串储存一个 JSON（`const char* json`）：

```
{
    "hello": "world",
    "t": true ,
    "f": false,
    "n": null,
    "i": 123,
    "pi": 3.1416,
    "a": [1, 2, 3, 4]
}
```

把它解析至一个 `Document`：

```
#include "rapidjson/document.h"

using namespace rapidjson;

// ...
Document document;
document.Parse(json);
```

那么现在该 JSON 就会被解析至 `document` 中，成为一棵 *DOM* 树：




自从 RFC 7159 作出更新，合法 JSON 文件的根可以是任何类型的 JSON 值。而在较早的 RFC 4627 中，根值只允许是 Object 或 Array。而在上述例子中，根是一个 Object。

```
assert(document.IsObject());
```

让我们查询一下根 Object 中有没有 `"hello"` 成员。由于一个 `Value` 可包含不同类型的值，我们可能需要验证它的类型，并使用合适的 API 去获取其值。在此例中，`"hello"` 成员关联到一个 JSON String。

```
assert(document.HasMember("hello"));
assert(document["hello"].IsString());
printf("hello = %s\n", document["hello"].GetString());
```

```
world
```

JSON True/False 值是以 `bool` 表示的。

```
assert(document["t"].IsBool());
printf("t = %s\n", document["t"].GetBool() ? "true" : "false");
```

```
true
```

JSON Null 值可用 `IsNull()` 查询。

```
printf("n = %s\n", document["n"].IsNull() ? "null" : "?");
```

```
null
```

JSON Number 类型表示所有数值。然而，C++ 需要使用更专门的类型。

```
assert(document["i"].IsNumber());

// 在此情况下，IsUint()/IsInt64()/IsUInt64() 也会返回 true
assert(document["i"].IsInt());
printf("i = %d\n", document["i"].GetInt());
// 另一种用法： (int)document["i"]

assert(document["pi"].IsNumber());
assert(document["pi"].IsDouble());
printf("pi = %g\n", document["pi"].GetDouble());
```

```
i = 123
pi = 3.1416
```

JSON Array 包含一些元素。

```
// 使用引用来连续访问，方便之余还更高效。
const Value& a = document["a"];
assert(a.IsArray());
for (SizeType i = 0; i < a.Size(); i++) // 使用 SizeType 而不是 size_t
        printf("a[%d] = %d\n", i, a[i].GetInt());
```

```
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
```

注意，RapidJSON 并不自动转换各种 JSON 类型。例如，对一个 String 的 Value 调用 `GetInt()` 是非法的。在调试模式下，它会被断言失败。在发布模式下，其行为是未定义的。

以下将会讨论有关查询各类型的细节。

## 查询 Array

缺省情况下，`SizeType` 是 `unsigned` 的 typedef。在多数系统中，Array 最多能存储 2^32-1 个元素。

你可以用整数字面量访问元素，如 `a[0]` 、 `a[1]` 、 `a[2]` 。

Array 与 `std::vector` 相似，除了使用索引，也可使用迭代器来访问所有元素。

```
for (Value::ConstValueIterator itr = a.Begin(); itr != a.End(); ++itr)
    printf("%d ", itr->GetInt());
```

还有一些熟悉的查询函数：

- `SizeType Capacity() const`
- `bool Empty() const`

### 范围 for 循环 (v1.1.0 中的新功能)

当使用 C++11 功能时，你可使用范围 for 循环去访问 Array 内的所有元素。

```
for (auto& v : a.GetArray())
    printf("%d ", v.GetInt());
```

## 查询 Object

和 Array 相似，我们可以用迭代器去访问所有 Object 成员：

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

for (Value::ConstMemberIterator itr = document.MemberBegin();
    itr != document.MemberEnd(); ++itr)
{
    printf("Type of member %s is %s\n",
        itr->name.GetString(), kTypeNames[itr->value.GetType()]);
}
```

```
Type of member hello is String
Type of member t is True
Type of member f is False
Type of member n is Null
Type of member i is Number
Type of member pi is Number
Type of member a is Array
```

注意，当 `operator[](const char*)` 找不到成员，它会断言失败。

若我们不确定一个成员是否存在，便需要在调用 `operator[](const char*)` 前先调用 `HasMember()`。然而，这会导致两次查找。更好的做法是调用 `FindMember()`，它能同时检查成员是否存在并返回它的 Value：

```
Value::ConstMemberIterator itr = document.FindMember("hello");
if (itr != document.MemberEnd())
    printf("%s\n", itr->value.GetString());
```

### 范围 for 循环 (v1.1.0 中的新功能)

当使用 C++11 功能时，你可使用范围 for 循环去访问 Object 内的所有成员。

```
for (auto& m : document.GetObject())
    printf("Type of member %s is %s\n",
        m.name.GetString(), kTypeNames[m.value.GetType()]);
```

# 查询 Number

JSON 只提供一种数值类型——Number。数字可以是整数或实数。RFC 4627 规定数字的范围由解析器指定。

由于 C++ 提供多种整数及浮点数类型，DOM 尝试尽量提供最广的范围及良好性能。

当解析一个 Number 时, 它会被存储在 DOM 之中，成为下列其中一个类型：

| 类型 | 描述 |
| --- | --- |
| `unsigned` | 32 位无号整数 |
| `int` | 32 位有号整数 |
| `uint64_t` | 64 位无号整数 |
| `int64_t` | 64 位有号整数 |
| `double` | 64 位双精度浮点数 |

当查询一个 Number 时, 你可以检查该数字是否能以目标类型来提取：

| 查检 | 提取 |
| --- | --- |
| `bool IsNumber()` | 不适用 |
| `bool IsUint()` | `unsigned GetUint()` |
| `bool IsInt()` | `int GetInt()` |
| `bool IsUint64()` | `uint64_t GetUint64()` |
| `bool IsInt64()` | `int64_t GetInt64()` |
| `bool IsDouble()` | `double GetDouble()` |

注意，一个整数可能用几种类型来提取，而无需转换。例如，一个名为 `x` 的 Value 包含 123，那么 `x.IsInt() == x.IsUint() == x.IsInt64() == x.IsUint64() == true`。但如果一个名为 `y` 的 Value 包含 -3000000000，那么仅会令 `x.IsInt64() == true`。

当要提取 Number 类型，`GetDouble()` 是会把内部整数的表示转换成 `double`。注意 `int` 和 `unsigned` 可以安全地转换至 `double`，但 `int64_t` 及 `uint64_t` 可能会丧失精度（因为 `double` 的尾数只有 52 位）。

# 查询 String

除了 `GetString()`，`Value` 类也有一个 `GetStringLength()`。这里会解释个中原因。

根据 RFC 4627，JSON String 可包含 Unicode 字符 `U+0000`，在 JSON 中会表示为 `"\u0000"`。问题是，C/C++ 通常使用空字符结尾字符串（null-terminated string），这种字符串把 `` `\0' `` 作为结束符号。

为了符合 RFC 4627，RapidJSON 支持包含 `U+0000` 的 String。若你需要处理这些 String，便可使用 `GetStringLength()` 去获得正确的字符串长度。

例如，当解析以下的 JSON 至 `Document d` 之后：

```
{ "s" :  "a\u0000b" }
```

`"a\u0000b"` 值的正确长度应该是 3。但 `strlen()` 会返回 1。

`GetStringLength()` 也可以提高性能，因为用户可能需要调用 `strlen()` 去分配缓冲。

此外，`std::string` 也支持这个构造函数：

```
string(const char* s, size_t count);
```

此构造函数接受字符串长度作为参数。它支持在字符串中存储空字符，也应该会有更好的性能。

## 比较两个 Value

你可使用 `==` 及 `!=` 去比较两个 Value。当且仅当两个 Value 的类型及内容相同，它们才当作相等。你也可以比较 Value 和它的原生类型值。以下是一个例子。

```
if (document["hello"] == document["n"]) /*...*/;    // 比较两个值
if (document["hello"] == "world") /*...*/;          // 与字符串家面量作比较
if (document["i"] != 123) /*...*/;                  // 与整数作比较
if (document["pi"] != 3.14) /*...*/;                // 与 double 作比较
```

Array／Object 顺序以它们的元素／成员作比较。当且仅当它们的整个子树相等，它们才当作相等。

注意，现时若一个 Object 含有重复命名的成员，它与任何 Object 作比较都总会返回 `false`。

# 创建／修改值

有多种方法去创建值。 当一个 DOM 树被创建或修改后，可使用 `Writer` 再次存储为 JSON。

## 改变 Value 类型

当使用默认构造函数创建一个 Value 或 Document，它的类型便会是 Null。要改变其类型，需调用 `SetXXX()` 或赋值操作，例如：

```
Document d; // Null
d.SetObject();

Value v;    // Null
v.SetInt(10);
v = 10;     // 简写，和上面的相同
```

### 构造函数的各个重载

几个类型也有重载构造函数：

```
Value b(true);    // 调用 Value(bool)
Value i(-123);    // 调用 Value(int)
Value u(123u);    // 调用 Value(unsigned)
Value d(1.5);     // 调用 Value(double)
```

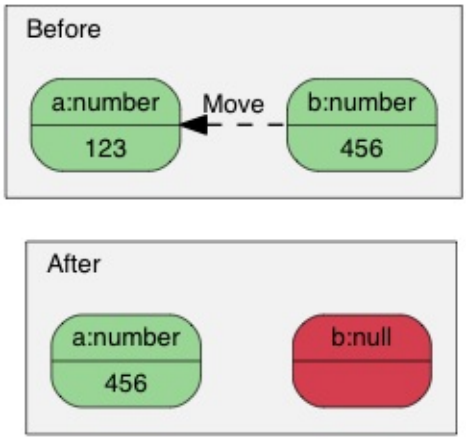
要重建空 Object 或 Array，可在默认构造函数后使用 `SetObject()` / `SetArray()`，或一次性使用 `Value(Type)`：

```
Value o(kObjectType);
Value a(kArrayType);
```

## 转移语义（Move Semantics）

在设计 RapidJSON 时有一个非常特别的决定，就是 Value 赋值并不是把来源 Value 复制至目的 Value，而是把把来源 Value 转移（move）至目的 Value。例如：

```
Value a(123);
Value b(456);
b = a;          // a 变成 Null，b 变成数字 123。
```

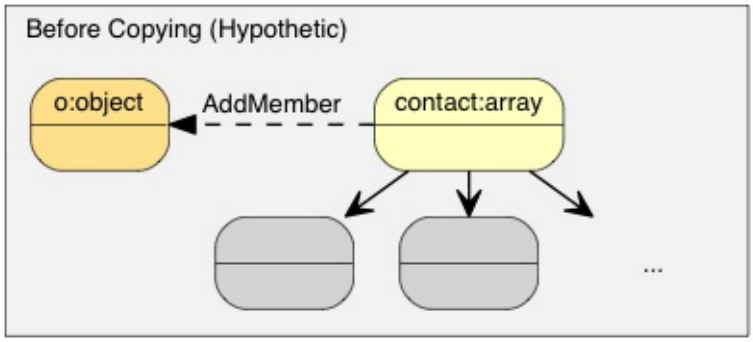


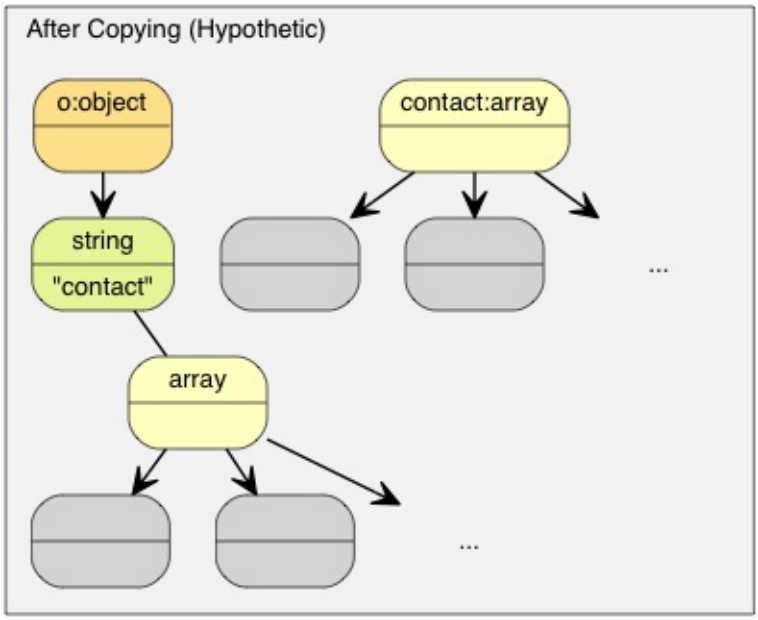

为什么？此语义有何优点？

最简单的答案就是性能。对于固定大小的 JSON 类型（Number、True、False、Null），复制它们是简单快捷。然而，对于可变大小的 JSON 类型（String、Array、Object），复制它们会产生大量开销，而且这些开销常常不被察觉。尤其是当我们需要创建临时 Object，把它复制至另一变量，然后再析构它。

例如，若使用正常 复制 语义：

```
Value o(kObjectType);
{
    Value contacts(kArrayType);
    // 把元素加进 contacts 数组。
    // ...
    o.AddMember("contacts", contacts, d.GetAllocator());  // 深度复制 contacts （可能有大量内存分配）
    // 析构 contacts。
}
```



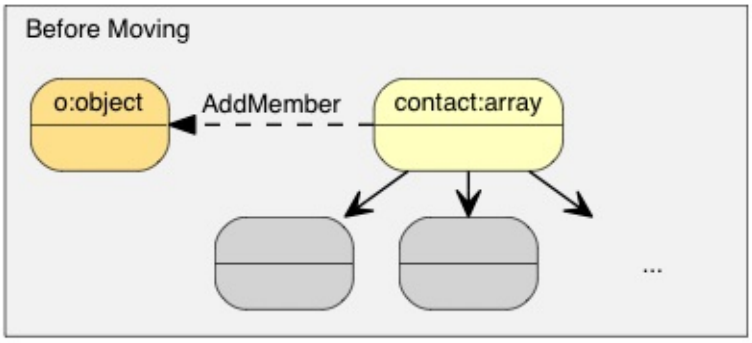

那个 `o` Object 需要分配一个和 contacts 相同大小的缓冲区，对 conacts 做深度复制，并最终要析构 contacts。这样会产生大量无必要的内存分配／释放，以及内存复制。

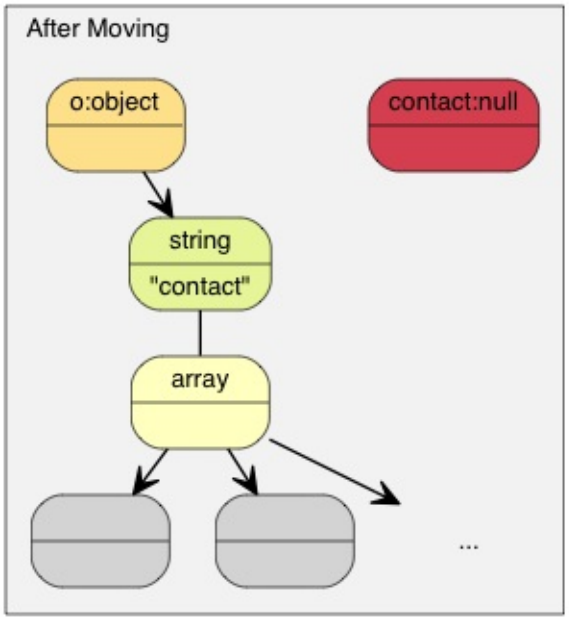
有一些方案可避免实质地复制这些数据，例如引用计数（reference counting）、垃圾回收（garbage collection, GC）。

为了使 RapidJSON 简单及快速，我们选择了对赋值采用 转移 语义。这方法与 `std::auto_ptr` 相似，都是在赋值时转移拥有权。转移快得多简单得多，只需要析构原来的 Value，把来源 `memcpy()` 至目标，最后把来源设置为 Null 类型。

因此，使用转移语义后，上面的例子变成：

```
Value o(kObjectType);
{
    Value contacts(kArrayType);
    // adding elements to contacts array.
    o.AddMember("contacts", contacts, d.GetAllocator());  // 只需 memcpy() contacts 本身至新成员的 Value（16 字节）
    // contacts 在这里变成 Null。它的析构是平凡的。
}
```




在 C++11 中这称为转移赋值操作（move assignment operator）。由于 RapidJSON 支持 C++03，它在赋值操作采用转移语义，其它修改型函数如 `AddMember()`，`PushBack()` 也采用转移语义。

## 转移语义及临时值

有时候，我们想直接构造一个 Value 并传递给一个“转移”函数（如 `PushBack()`、`AddMember()`）。由于临时对象是不能转换为正常的 Value 引用，我们加入了一个方便的 `Move()` 函数：

```
Value a(kArrayType);
Document::AllocatorType& allocator = document.GetAllocator();
// a.PushBack(Value(42), allocator);       // 不能通过编译
a.PushBack(Value().SetInt(42), allocator); // fluent API
a.PushBack(Value(42).Move(), allocator);   // 和上一行相同
```

## 创建 String

RapidJSON 提供两个 String 的存储策略。

1. copy-string: 分配缓冲区，然后把来源数据复制至它。
2. const-string: 简单地储存字符串的指针。

Copy-string 总是安全的，因为它拥有数据的克隆。Const-string 可用于存储字符串字面量，以及用于在 DOM 一节中将会提到的 in-situ 解析中。

为了让用户自定义内存分配方式，当一个操作可能需要内存分配时，RapidJSON 要求用户传递一个 allocator 实例作为 API 参数。此设计避免了在每个 Value 存储 allocator（或 document）的指针。

因此，当我们把一个 copy-string 赋值时，调用含有 allocator 的 `SetString()` 重载函数：

```
Document document;
Value author;
char buffer[10];
int len = sprintf(buffer, "%s %s", "Milo", "Yip"); // 动态创建的字符串。
author.SetString(buffer, len, document.GetAllocator());
memset(buffer, 0, sizeof(buffer));
// 清空 buffer 后 author.GetString() 仍然包含 "Milo Yip"
```

在此例子中，我们使用 `Document` 实例的 allocator。这是使用 RapidJSON 时常用的惯用法。但你也可以用其他 allocator 实例。

另外，上面的 `SetString()` 需要长度参数。这个 API 能处理含有空字符的字符串。另一个 `SetString()` 重载函数没有长度参数，它假设输入是空字符结尾的，并会调用类似 `strlen()` 的函数去获取长度。

最后，对于字符串字面量或有安全生命周期的字符串，可以使用 const-string 版本的 `SetString()`，它没有 allocator 参数。对于字符串家面量（或字符数组常量），只需简单地传递字面量，又安全又高效：

```
Value s;
s.SetString("rapidjson");    // 可包含空字符，长度在编译其推导
s = "rapidjson";             // 上行的缩写
```

对于字符指针，RapidJSON 需要作一个标记，代表它不复制也是安全的。可以使用 `StringRef` 函数：

```
const char * cstr = getenv("USER");
size_t cstr_len = ...;                  // 如果有长度
Value s;
// s.SetString(cstr);                   // 这不能通过编译
s.SetString(StringRef(cstr));           // 可以，假设它的生命周期安全，并且是以空字符结尾的
s = StringRef(cstr);                    // 上行的缩写
s.SetString(StringRef(cstr, cstr_len));// 更快，可处理空字符
s = StringRef(cstr, cstr_len);          // 上行的缩写
```

## 修改 Array

Array 类型的 Value 提供与 `std::vector` 相似的 API。

- `Clear()`
- `Reserve(SizeType, Allocator&)`
- `Value& PushBack(Value&, Allocator&)`
- `template <typename T> GenericValue& PushBack(T, Allocator&)`
- `Value& PopBack()`
- `ValueIterator Erase(ConstValueIterator pos)`
- `ValueIterator Erase(ConstValueIterator first, ConstValueIterator last)`

注意，`Reserve(...)` 及 `PushBack(...)` 可能会为数组元素分配内存，所以需要一个 allocator。

以下是 `PushBack()` 的例子：

```
Value a(kArrayType);
Document::AllocatorType& allocator = document.GetAllocator();

for (int i = 5; i <= 10; i++)
    a.PushBack(i, allocator);   // 可能需要调用 realloc() 所以需要 allocator

// 流畅接口（Fluent interface）
a.PushBack("Lua", allocator).PushBack("Mio", allocator);
```

与 STL 不一样的是，`PushBack()` / `PopBack()` 返回 Array 本身的引用。这称为流畅接口（*fluent interface*）。

如果你想在 Array 中加入一个非常量字符串，或是一个没有足够生命周期的字符串（见 Create String），你需要使用 copy-string API 去创建一个 String。为了避免加入中间变量，可以就地使用一个 临时值：

```
// 就地 Value 参数
contact.PushBack(Value("copy", document.GetAllocator()).Move(), // copy string
                 document.GetAllocator());

// 显式 Value 参数
Value val("key", document.GetAllocator()); // copy string
contact.PushBack(val, document.GetAllocator());
```

# 修改 Object

Object 是键值对的集合。每个键必须为 String。要修改 Object，方法是增加或移除成员。以下的 API 用来增加城员：

- `Value& AddMember(Value&, Value&, Allocator& allocator)`
- `Value& AddMember(StringRefType, Value&, Allocator&)`
- `template <typename T> Value& AddMember(StringRefType, T value, Allocator&)`

以下是一个例子。

```
Value contact(kObject);
contact.AddMember("name", "Milo", document.GetAllocator());
contact.AddMember("married", true, document.GetAllocator());
```

使用 `StringRefType` 作为 name 参数的重载版本与字符串的 `SetString` 的接口相似。这些重载是为了避免复制 `name` 字符串，因为 JSON object 中经常会使用常数键名。

如果你需要从非常数字符串或生命周期不足的字符串创建键名（见 创建 String），你需要使用 copy-string API。为了避免中间变量，可以就地使用 临时值：

```
// 就地 Value 参数
contact.AddMember(Value("copy", document.GetAllocator()).Move(), // copy string
                  Value().Move(),                                // null value
                  document.GetAllocator());

// 显式参数
Value key("key", document.GetAllocator()); // copy string name
Value val(42);                              // 某 Value
contact.AddMember(key, val, document.GetAllocator());
```

移除成员有几个选择：

- `bool RemoveMember(const Ch* name)`：使用键名来移除成员（线性时间复杂度）。
- `bool RemoveMember(const Value& name)`：除了 `name` 是一个 Value，和上一行相同。
- `MemberIterator RemoveMember(MemberIterator)`：使用迭代器移除成员（常数时间复杂度）。
- `MemberIterator EraseMember(MemberIterator)`：和上行相似但维持成员次序（线性时间复杂度）。
- `MemberIterator EraseMember(MemberIterator first, MemberIterator last)`：移除一个范围内的成员，维持次序（线性时间复杂度）。

`MemberIterator RemoveMember(MemberIterator)` 使用了“转移最后”手法来达成常数时间复杂度。基本上就是析构迭代器位置的成员，然后把最后的成员转移至迭代器位置。因此，成员的次序会被改变。

# 深复制 Value

若我们真的要复制一个 DOM 树，我们可使用两个 APIs 作深复制：含 allocator 的构造函数及 `CopyFrom()`。

```
Document d;
Document::AllocatorType& a = d.GetAllocator();
Value v1("foo");
// Value v2(v1); // 不容许

Value v2(v1, a);                        // 制造一个克隆
assert(v1.IsString());                  // v1 不变
d.SetArray().PushBack(v1, a).PushBack(v2, a);
assert(v1.IsNull() && v2.IsNull());     // 两个都转移动 d

v2.CopyFrom(d, a);                      // 把整个 document 复制至 v2
assert(d.IsArray() && d.Size() == 2); // d 不变
v1.SetObject().AddMember("array", v2, a);
d.PushBack(v1, a);
```

## 交换 Value

RapidJSON 也提供 `Swap()` 。

```
Value a(123);
Value b("Hello");
a.Swap(b);
assert(a.IsString());
assert(b.IsInt());
```

无论两棵 DOM 树有多复杂，交换是很快的（常数时间）。

## 下一部分

本教程展示了如何询查及修改 DOM 树。RapidJSON 还有一个重要概念：

1. 流 是读写 JSON 的通道。流可以是内存字符串、文件流等。用户也可以自定义流。
2. 编码 定义在流或内存中使用的字符编码。RapidJSON 也在内部提供 Unicode 转换及校验功能。
3. DOM 的基本功能已在本教程里介绍。还有更高级的功能，如原位（*in situ*）解析、其他解析选项及高级用法。
4. SAX 是 RapidJSON 解析／生成功能的基础。学习使用 `Reader` / `Writer` 去实现更高性能的应用程序。也可以使用 `PrettyWriter` 去格式化 JSON。
5. 性能 展示一些我们做的及第三方的性能测试。
6. 技术内幕 讲述一些 RapidJSON 内部的设计及技术。

你也可以参考 常见问题、API 文档、例子及单元测试。

# Pointer

（本功能于 v1.1.0 发布）

JSON Pointer 是一个标准化（RFC6901）的方式去选取一个 JSON Document（DOM）中的值。这类似于 XML 的 XPath。然而，JSON Pointer 简单得多，而且每个 JSON Pointer 仅指向单个值。

使用 RapidJSON 的 JSON Pointer 实现能简化一些 DOM 的操作。

[TOC]

# JSON Pointer

一个 JSON Pointer 由一串（零至多个）token 所组成，每个 token 都有 `/` 前缀。每个 token 可以是一个字符串或数字。例如，给定一个 JSON：

```
{
    "foo" : ["bar", "baz"],
    "pi" : 3.1416
}
```

以下的 JSON Pointer 解析为：

1. `"/foo"` → `[ "bar", "baz" ]`
2. `"/foo/0"` → `"bar"`
3. `"/foo/1"` → `"baz"`
4. `"/pi"` → `3.1416`

要注意，一个空 JSON Pointer `""` （零个 token）解析为整个 JSON。

## 基本使用方法

以下的代码范例不解自明。

```
#include "rapidjson/pointer.h"

// ...
Document d;

// 使用 Set() 创建 DOM
Pointer("/project").Set(d, "RapidJSON");
Pointer("/stars").Set(d, 10);

// { "project" : "RapidJSON", "stars" : 10 }

// 使用 Get() 访问 DOM。若该值不存在则返回 nullptr。
if (Value* stars = Pointer("/stars").Get(d))
    stars->SetInt(stars->GetInt() + 1);

// { "project" : "RapidJSON", "stars" : 11 }

// Set() 和 Create() 自动生成父值（如果它们不存在）。
Pointer("/a/b/0").Create(d);

// { "project" : "RapidJSON", "stars" : 11, "a" : { "b" : [ null ] } }

// GetWithDefault() 返回引用。若该值不存在则会深拷贝缺省值。
Value& hello = Pointer("/hello").GetWithDefault(d, "world");

// { "project" : "RapidJSON", "stars" : 11, "a" : { "b" : [ null ] }, "hello" : "world"
 }

// Swap() 和 Set() 相似
Value x("C++");
Pointer("/hello").Swap(d, x);

// { "project" : "RapidJSON", "stars" : 11, "a" : { "b" : [ null ] }, "hello" : "C++" }
// x 变成 "world"

// 删去一个成员或元素，若值存在返回 true
bool success = Pointer("/a").Erase(d);
assert(success);

// { "project" : "RapidJSON", "stars" : 10 }
```

## 辅助函数

由于面向对象的调用习惯可能不符直觉，RapidJSON 也提供了一些辅助函数，它们把成员函数包装成自由函数。

以下的例子与上面例子所做的事情完全相同。

```
Document d;

SetValueByPointer(d, "/project", "RapidJSON");
SetValueByPointer(d, "/stars", 10);

if (Value* stars = GetValueByPointer(d, "/stars"))
    stars->SetInt(stars->GetInt() + 1);

CreateValueByPointer(d, "/a/b/0");

Value& hello = GetValueByPointerWithDefault(d, "/hello", "world");

Value x("C++");
SwapValueByPointer(d, "/hello", x);

bool success = EraseValueByPointer(d, "/a");
assert(success);
```

以下对比 3 种调用方式：

1. `Pointer(source).<Method>(root, ...)`
2. `<Method>ValueByPointer(root, Pointer(source), ...)`
3. `<Method>ValueByPointer(root, source, ...)`

# 解析 Pointer

`Pointer::Get()` 或 `GetValueByPointer()` 函数并不修改 DOM。若那些 token 不能匹配 DOM 里的值，这些函数便返回 `nullptr`。使用者可利用这个方法来检查一个值是否存在。

注意，数值 token 可表示数组索引或成员名字。解析过程中会按值的类型来匹配。

```
{
    "0" : 123,
    "1" : [456]
}
```

1. `"/0"` → `123`
2. `"/1/0"` → `456`

Token `"0"` 在第一个 pointer 中被当作成员名字。它在第二个 pointer 中被当作成数组索引。

其他函数会改变 DOM，包括 `Create()`、`GetWithDefault()`、`Set()`、`Swap()`。这些函数总是成功的。若一些父值不存在，就会创建它们。若父值类型不匹配 token，也会强行改变其类型。改变类型也意味着完全移除其 DOM 子树的内容。

例如，把上面的 JSON 解译至 `d` 之后，

```
SetValueByPointer(d, "1/a", 789); // { "0" : 123, "1" : { "a" : 789 } }
```

## 解析负号 token

另外，RFC6901 定义了一个特殊 token `-`（单个负号），用于表示数组最后元素的下一个元素。`Get()` 只会把此 token 当作成员名字 "-"。而其他函数则会以此解析数组，等同于对数组调用 `Value::PushBack()`。

```
Document d;
d.Parse("{\"foo\":[123]}");
SetValueByPointer(d, "/foo/-", 456); // { "foo" : [123, 456] }
SetValueByPointer(d, "/-", 789);    // { "foo" : [123, 456], "-" : 789 }
```

## 解析 Document 及 Value

当使用 `p.Get(root)` 或 `GetValueByPointer(root, p)`，`root` 是一个（常数） `Value&`。这意味着，它也可以是 DOM 里的一个子树。

其他函数有两组签名。一组使用 `Document& document` 作为参数，另一组使用 `Value& root`。第一组使用 `document.GetAllocator()` 去创建值，而第二组则需要使用者提供一个 allocator，如同 DOM 里的函数。

以上例子都不需要 allocator 参数，因为它的第一个参数是 `Document&`。但如果你需要对一个子树进行解析，就需要如下面的例子般提供 allocator：

```
class Person {
public:
    Person() {
        document_ = new Document();
        // CreateValueByPointer() here no need allocator
        SetLocation(CreateValueByPointer(*document_, "/residence"), ...);
        SetLocation(CreateValueByPointer(*document_, "/office"), ...);
    };

private:
    void SetLocation(Value& location, const char* country, const char* addresses[2]) {
        Value::Allocator& a = document_->GetAllocator();
        // SetValueByPointer() here need allocator
        SetValueByPointer(location, "/country", country, a);
        SetValueByPointer(location, "/address/0", address[0], a);
        SetValueByPointer(location, "/address/1", address[1], a);
    }

    // ...

    Document* document_;
};
```

`Erase()` 或 `EraseValueByPointer()` 不需要 allocator。而且它们成功删除值之后会返回 `true`。

## 错误处理

`Pointer` 在其建构函数里会解译源字符串。若有解析错误，`Pointer::IsValid()` 返回 `false`。你可使用 `Pointer::GetParseErrorCode()` 和 `GetParseErrorOffset()` 去获取错信息。

要注意的是，所有解析函数都假设 pointer 是合法的。对一个非法 pointer 解析会做成断言失败。

# URI 片段表示方式

除了我们一直在使用的字符串方式表示 JSON pointer，RFC6901 也定义了一个 JSON Pointer 的 URI 片段（fragment）表示方式。URI 片段是定义于 RFC3986 "Uniform Resource Identifier (URI): Generic Syntax"。

URI 片段的主要分别是必然以 `#`（pound sign）开头，而一些字符也会以百分比编码成 UTF-8 序列。例如，以下的表展示了不同表示法下的 C/C++ 字符串常数。

| 字符串表示方式 | URI 片段表示方式 | Pointer Tokens（UTF-8） |
|---|---|---|
| `"/foo/0"` | `"#/foo/0"` | `{"foo", 0}` |
| `"/a~1b"` | `"#/a~1b"` | `{"a/b"}` |
| `"/m~0n"` | `"#/m~0n"` | `{"m~n"}` |
| `"/ "` | `"#/%20"` | `{" "}` |
| `"/\0"` | `"#/%00"` | `{"\0"}` |
| `"/€"` | `"#/%E2%82%AC"` | `{"€"}` |

RapidJSON 完全支持 URI 片段表示方式。它在解译时会自动检测 `#` 号。

## 字符串化

你也可以把一个 `Pointer` 字符串化，储存于字符串或其他输出流。例如：

```
Pointer p(...);
StringBuffer sb;
p.Stringify(sb);
std::cout << sb.GetString() << std::endl;
```

使用 `StringifyUriFragment()` 可以把 pointer 字符串化为 URI 片段表示法。

## 使用者提供的 tokens

若一个 pointer 会用于多次解析，它应该只被创建一次，然后再施于不同的 DOM，或在不同时间做解析。这样可以避免多次创键 `Pointer`，节省时间和内存分配。

我们甚至可以再更进一步，完全消去解析过程及动态内存分配。我们可以直接生成 token 数组：

```
#define NAME(s) { s, sizeof(s) / sizeof(s[0]) - 1, kPointerInvalidIndex }
#define INDEX(i) { #i, sizeof(#i) - 1, i }

static const Pointer::Token kTokens[] = { NAME("foo"), INDEX(123) };
static const Pointer p(kTokens, sizeof(kTokens) / sizeof(kTokens[0]));
// Equivalent to static const Pointer p("/foo/123");
```

这种做法可能适合内存受限的系统。

# 流

在 RapidJSON 中，`rapidjson::Stream` 是用於读写 JSON 的概念（概念是指 C++ 的 concept）。在这里我们先介绍如何使用 RapidJSON 提供的各种流。然后再看看如何自行定义流。

[TOC]

# 内存流

内存流把 JSON 存储在内存之中。

## StringStream（输入）

`StringStream` 是最基本的输入流，它表示一个完整的、只读的、存储于内存的 JSON。它在 `rapidjson/rapidjson.h` 中定义。

```
#include "rapidjson/document.h" // 会包含 "rapidjson/rapidjson.h"

using namespace rapidjson;

// ...
const char json[] = "[1, 2, 3, 4]";
StringStream s(json);

Document d;
d.ParseStream(s);
```

由于这是非常常用的用法，RapidJSON 提供 `Document::Parse(const char*)` 去做完全相同的事情：

```
// ...
const char json[] = "[1, 2, 3, 4]";
Document d;
d.Parse(json);
```

需要注意，`StringStream` 是 `GenericStringStream<UTF8<> >` 的 typedef，使用者可用其他编码类去代表流所使用的字符集。

## StringBuffer（输出）

`StringBuffer` 是一个简单的输出流。它分配一个内存缓冲区，供写入整个 JSON。可使用 `GetString()` 来获取该缓冲区。

```
#include "rapidjson/stringbuffer.h"

StringBuffer buffer;
Writer<StringBuffer> writer(buffer);
d.Accept(writer);

const char* output = buffer.GetString();
```

当缓冲区满溢，它将自动增加容量。缺省容量是 256 个字符（UTF8 是 256 字节，UTF16 是 512 字节等）。使用者能自行提供分配器及初始容量。

```
StringBuffer buffer1(0, 1024); // 使用它的分配器，初始大小 = 1024
StringBuffer buffer2(allocator, 1024);
```

如无设置分配器，`StringBuffer` 会自行实例化一个内部分配器。

相似地，`StringBuffer` 是 `GenericStringBuffer<UTF8<> >` 的 typedef。

# 文件流

当要从文件解析一个 JSON，你可以把整个 JSON 读入内存并使用上述的 `StringStream`。

然而，若 JSON 很大，或是内存有限，你可以改用 `FileReadStream`。它只会从文件读取一部分至缓冲区，然后让那部分被解析。若缓冲区的字符都被读完，它会再从文件读取下一部分。

## FileReadStream（输入）

`FileReadStream` 通过 `FILE` 指针读取文件。使用者需要提供一个缓冲区。

```
#include "rapidjson/filereadstream.h"
#include <cstdio>

using namespace rapidjson;

FILE* fp = fopen("big.json", "rb"); // 非 Windows 平台使用 "r"

char readBuffer[65536];
FileReadStream is(fp, readBuffer, sizeof(readBuffer));

Document d;
d.ParseStream(is);

fclose(fp);
```

与 `StringStreams` 不一样，`FileReadStream` 是一个字节流。它不处理编码。若文件并非 UTF-8 编码，可以把字节流用 `EncodedInputStream` 包装。我们很快会讨论这个问题。

除了读取文件，使用者也可以使用 `FileReadStream` 来读取 `stdin`。

## FileWriteStream（输出）

`FileWriteStream` 是一个含缓冲功能的输出流。它的用法与 `FileReadStream` 非常相似。

```
#include "rapidjson/filewritestream.h"
#include <cstdio>

using namespace rapidjson;

Document d;
d.Parse(json);
// ...

FILE* fp = fopen("output.json", "wb"); // 非 Windows 平台使用 "w"

char writeBuffer[65536];
FileWriteStream os(fp, writeBuffer, sizeof(writeBuffer));

Writer<FileWriteStream> writer(os);
d.Accept(writer);

fclose(fp);
```

它也可以把输出导向 `stdout`。

## iostream 包装类

基于用户的要求，RapidJSON 提供了正式的 `std::basic_istream` 和 `std::basic_ostream` 包装类。然而，请注意其性能会大大低于以上的其他流。

## IStreamWrapper

`IStreamWrapper` 把任何继承自 `std::istream` 的类（如 `std::istringstream`、`std::stringstream`、`std::ifstream`、`std::fstream`）包装成 RapidJSON 的输入流。

```
#include <rapidjson/document.h>
#include <rapidjson/istreamwrapper.h>
#include <fstream>

using namespace rapidjson;
using namespace std;

ifstream ifs("test.json");
IStreamWrapper isw(ifs);

Document d;
d.ParseStream(isw);
```

对于继承自 `std::wistream` 的类，则使用 `WIStreamWrapper`。

## OStreamWrapper

相似地，`OStreamWrapper` 把任何继承自 `std::ostream` 的类（如 `std::ostringstream`、`std::stringstream`、`std::ofstream`、`std::fstream`）包装成 RapidJSON 的输出流。

```
#include <rapidjson/document.h>
#include <rapidjson/ostreamwrapper.h>
#include <rapidjson/writer.h>
#include <fstream>

using namespace rapidjson;
using namespace std;

Document d;
d.Parse(json);

// ...

ofstream ofs("output.json");
OStreamWrapper osw(ofs);

Writer<OStreamWrapper> writer(osw);
d.Accept(writer);
```

对于继承自 `std::wistream` 的类，则使用 `WIStreamWrapper`。

## 编码流

编码流（encoded streams）本身不存储 JSON，它们是通过包装字节流来提供基本的编码／解码功能。

如上所述，我们可以直接读入 UTF-8 字节流。然而，UTF-16 及 UTF-32 有字节序（endian）问题。要正确地处理字节序，需要在读取时把字节转换成字符（如对 UTF-16 使用 `wchar_t`），以及在写入时把字符转换为字节。

除此以外，我们也需要处理 字节顺序标记（byte order mark, BOM）。当从一个字节流读取时，需要检测 BOM，或者仅仅是把存在的 BOM 消去。当把 JSON 写入字节流时，也可选择写入 BOM。

若一个流的编码在编译期已知，你可使用 `EncodedInputStream` 及 `EncodedOutputStream`。若一个流可能存储 UTF-8、UTF-16LE、UTF-16BE、UTF-32LE、UTF-32BE 的 JSON，并且编码只能在运行时得知，你便可以使用 `AutoUTFInputStream` 及 `AutoUTFOutputStream`。这些流定义在 `rapidjson/encodedstream.h`。

注意到，这些编码流可以施于文件以外的流。例如，你可以用编码流包装内存中的文件或自定义的字节流。

## EncodedInputStream

`EncodedInputStream` 含两个模板参数。第一个是 `Encoding` 类型，例如定义于 `rapidjson/encodings.h` 的 `UTF8` 、 `UTF16LE` 。第二个参数是被包装的流的类型。

```
#include "rapidjson/document.h"
#include "rapidjson/filereadstream.h"   // FileReadStream
#include "rapidjson/encodedstream.h"    // EncodedInputStream
#include <cstdio>

using namespace rapidjson;

FILE* fp = fopen("utf16le.json", "rb"); // 非 Windows 平台使用 "r"

char readBuffer[256];
FileReadStream bis(fp, readBuffer, sizeof(readBuffer));

EncodedInputStream<UTF16LE<>, FileReadStream> eis(bis);  // 用 eis 包装 bis

Document d; // Document 为 GenericDocument<UTF8<> >
d.ParseStream<0, UTF16LE<> >(eis);  // 把 UTF-16LE 文件解析至内存中的 UTF-8

fclose(fp);
```

## EncodedOutputStream

`EncodedOutputStream` 也是相似的，但它的构造函数有一个 `bool putBOM` 参数，用于控制是否在输出字节流写入 BOM。

```
#include "rapidjson/filewritestream.h"  // FileWriteStream
#include "rapidjson/encodedstream.h"    // EncodedOutputStream
#include <cstdio>

Document d;         // Document 为 GenericDocument<UTF8<> >
// ...

FILE* fp = fopen("output_utf32le.json", "wb"); // 非 Windows 平台使用 "w"

char writeBuffer[256];
FileWriteStream bos(fp, writeBuffer, sizeof(writeBuffer));

typedef EncodedOutputStream<UTF32LE<>, FileWriteStream> OutputStream;
OutputStream eos(bos, true);   // 写入 BOM

Writer<OutputStream, UTF32LE<>, UTF8<>> writer(eos);
d.Accept(writer);   // 这里从内存的 UTF-8 生成 UTF32-LE 文件

fclose(fp);
```

## AutoUTFInputStream

有时候，应用软件可能需要毋理所有可支持的 JSON 编码。`AutoUTFInputStream` 会先使用 BOM 来检测编码。若 BOM 不存在，它便会使用合法 JSON 的特性来检测。若两种方法都失败，它就会倒退至构造函数提供的 UTF 类型。

由于字符（编码单元／code unit）可能是 8 位、16 位或 32 位，`AutoUTFInputStream` 需要一个能至少储存 32 位的字符类型。我们可以使用 `unsigned` 作为模板参数：

```
#include "rapidjson/document.h"
#include "rapidjson/filereadstream.h"   // FileReadStream
#include "rapidjson/encodedstream.h"    // AutoUTFInputStream
#include <cstdio>

using namespace rapidjson;

FILE* fp = fopen("any.json", "rb"); // 非 Windows 平台使用 "r"

char readBuffer[256];
FileReadStream bis(fp, readBuffer, sizeof(readBuffer));

AutoUTFInputStream<unsigned, FileReadStream> eis(bis);  // 用 eis 包装 bis

Document d;         // Document 为 GenericDocument<UTF8<> >
d.ParseStream<0, AutoUTF<unsigned> >(eis); // 把任何 UTF 编码的文件解析至内存中的 UTF-8

fclose(fp);
```

当要指定流的编码，可使用上面例子中 `ParseStream()` 的参数 `AutoUTF<CharType>`。

你可以使用 `UTFType GetType()` 去获取 UTF 类型，并且用 `HasBOM()` 检测输入流是否含有 BOM。

## AutoUTFOutputStream

相似地，要在运行时选择输出的编码，我们可使用 `AutoUTFOutputStream`。这个类本身并非「自动」。你需要在运行时指定 UTF 类型，以及是否写入 BOM。

```
using namespace rapidjson;

void WriteJSONFile(FILE* fp, UTFType type, bool putBOM, const Document& d) {
    char writeBuffer[256];
    FileWriteStream bos(fp, writeBuffer, sizeof(writeBuffer));

    typedef AutoUTFOutputStream<unsigned, FileWriteStream> OutputStream;
    OutputStream eos(bos, type, putBOM);

    Writer<OutputStream, UTF8<>, AutoUTF<> > writer;
    d.Accept(writer);
}
```

`AutoUTFInputStream` / `AutoUTFOutputStream` 是比 `EncodedInputStream` / `EncodedOutputStream` 方便。但前者会产生一点运行期额外开销。

## 自定义流

除了内存／文件流，使用者可创建自行定义适配 RapidJSON API 的流类。例如，你可以创建网络流、从压缩文件读取的流等等。

RapidJSON 利用模板结合不同的类型。只要一个类包含所有所需的接口，就可以作为一个流。流的接合定义在 `rapidjson/rapidjson.h` 的注释里：

```
concept Stream {
    typename Ch;    //!< 流的字符类型

    //! 从流读取当前字符，不移动读取指针（read cursor）
    Ch Peek() const;

    //! 从流读取当前字符，移动读取指针至下一字符。
    Ch Take();

    //! 获取读取指针。
    //! \return 从开始以来所读过的字符数量。
    size_t Tell();

    //! 从当前读取指针开始写入操作。
    //! \return 返回开始写入的指针。
    Ch* PutBegin();

    //! 写入一个字符。
    void Put(Ch c);

    //! 清空缓冲区。
    void Flush();

    //! 完成写作操作。
    //! \param begin PutBegin() 返回的开始写入指针。
    //! \return 已写入的字符数量。
    size_t PutEnd(Ch* begin);
}
```

输入流必须实现 `Peek()`、`Take()` 及 `Tell()`。输出流必须实现 `Put()` 及 `Flush()`。`PutBegin()` 及 `PutEnd()` 是特殊的接口，仅用于原位（*in situ*）解析。一般的流不需实现它们。然而，即使接口不需用于某些流，仍然需要提供空实现，否则会产生编译错误。

## 例子：**istream** 的包装类

以下的简单例子是 `std::istream` 的包装类，它只需现 3 个函数。

```
class MyIStreamWrapper {
public:
    typedef char Ch;

    MyIStreamWrapper(std::istream& is) : is_(is) {
    }

    Ch Peek() const { // 1
        int c = is_.peek();
        return c == std::char_traits<char>::eof() ? '\0' : (Ch)c;
    }

    Ch Take() { // 2
        int c = is_.get();
        return c == std::char_traits<char>::eof() ? '\0' : (Ch)c;
    }

    size_t Tell() const { return (size_t)is_.tellg(); } // 3

    Ch* PutBegin() { assert(false); return 0; }
    void Put(Ch) { assert(false); }
    void Flush() { assert(false); }
    size_t PutEnd(Ch*) { assert(false); return 0; }

private:
    MyIStreamWrapper(const MyIStreamWrapper&);
    MyIStreamWrapper& operator=(const MyIStreamWrapper&);

    std::istream& is_;
};
```

使用者能用它来包装 `std::stringstream`、`std::ifstream` 的实例。

```
const char* json = "[1,2,3,4]";
std::stringstream ss(json);
MyIStreamWrapper is(ss);

Document d;
d.ParseStream(is);
```

但要注意，由于标准库的内部开销问，此实现的性能可能不如 RapidJSON 的内存／文件流。

## 例子：ostream 的包装类

以下的例子是 `std::istream` 的包装类，它只需实现 2 个函数。

```
class MyOStreamWrapper {
public:
    typedef char Ch;

    OStreamWrapper(std::ostream& os) : os_(os) {
    }

    Ch Peek() const { assert(false); return '\0'; }
    Ch Take() { assert(false); return '\0'; }
    size_t Tell() const {  }

    Ch* PutBegin() { assert(false); return 0; }
    void Put(Ch c) { os_.put(c); }                  // 1
    void Flush() { os_.flush(); }                   // 2
    size_t PutEnd(Ch*) { assert(false); return 0; }

private:
    MyOStreamWrapper(const MyOStreamWrapper&);
    MyOStreamWrapper& operator=(const MyOStreamWrapper&);

    std::ostream& os_;
};
```

使用者能用它来包装 `std::stringstream`、`std::ofstream` 的实例。

```
Document d;
// ...

std::stringstream ss;
MyOStreamWrapper os(ss);

Writer<MyOStreamWrapper> writer(os);
d.Accept(writer);
```

但要注意，由于标准库的内部开销问，此实现的性能可能不如 RapidJSON 的内存／文件流。

## 总结

本节描述了 RapidJSON 提供的各种流的类。内存流很简单。若 JSON 存储在文件中，文件流可减少 JSON 解析及生成所需的内存量。编码流在字节流和字符流之间作转换。最后，使用者可使用一个简单接口创建自定义的流。

# 编码

根据 ECMA-404：

> (in Introduction) JSON text is a sequence of Unicode code points.
>
> 翻译：JSON 文本是 Unicode 码点的序列。

较早的 RFC4627 申明：

> (in §3) JSON text SHALL be encoded in Unicode. The default encoding is UTF-8.
>
> 翻译：JSON 文本应该以 Unicode 编码。缺省的编码为 UTF-8。
>
> (in §6) JSON may be represented using UTF-8, UTF-16, or UTF-32. When JSON is written in UTF-8, JSON is 8bit compatible. When JSON is written in UTF-16 or UTF-32, the binary content-transfer-encoding must be used.
>
> 翻译：JSON 可使用 UTF-8、UTF-16 或 UTF-32 表示。当 JSON 以 UTF-8 写入，该 JSON 是 8 位兼容的。当 JSON 以 UTF-16 或 UTF-32 写入，就必须使用二进制的内容传送编码。

RapidJSON 支持多种编码。它也能检查 JSON 的编码，以及在不同编码中进行转码。所有这些功能都是在内部实现，无需使用外部的程序库（如 ICU）。

[TOC]

# Unicode

根据 Unicode 的官方网站：

> Unicode 给每个字符提供了一个唯一的数字， 不论是什么平台、 不论是什么程序、 不论是什么语言。

这些唯一数字称为码点（code point），其范围介乎 `0x0` 至 `0x10FFFF` 之间。

## Unicode 转换格式

存储 Unicode 码点有多种编码方式。这些称为 Unicode 转换格式（Unicode Transformation Format, UTF）。RapidJSON 支持最常用的 UTF，包括：

- UTF-8：8 位可变长度编码。它把一个码点映射至 1 至 4 个字节。
- UTF-16：16 位可变长度编码。它把一个码点映射至 1 至 2 个 16 位编码单元（即 2 至 4 个字节）。
- UTF-32：32 位固定长度编码。它直接把码点映射至单个 32 位编码单元（即 4 字节）。

对于 UTF-16 及 UTF-32 来说，字节序（endianness）是有影响的。在内存中，它们通常都是以该计算机的字节序来存储。然而，当要储存在文件中或在网上传输，我们需要指明字节序列的字节序，是小端（little endian, LE）还是大端（big-endian, BE）。

RapidJSON 通过 `rapidjson/encodings.h` 中的 struct 去提供各种编码：

```
namespace rapidjson {

template<typename CharType = char>
struct UTF8;

template<typename CharType = wchar_t>
struct UTF16;

template<typename CharType = wchar_t>
struct UTF16LE;

template<typename CharType = wchar_t>
struct UTF16BE;

template<typename CharType = unsigned>
struct UTF32;

template<typename CharType = unsigned>
struct UTF32LE;

template<typename CharType = unsigned>
struct UTF32BE;

} // namespace rapidjson
```

对于在内存中的文本，我们正常会使用 `UTF8` 、 `UTF16` 或 `UTF32` 。对于处理经过 I/O 的文本，我们可使用 `UTF8` 、 `UTF16LE` 、 `UTF16BE` 、 `UTF32LE` 或 `UTF32BE` 。

当使用 DOM 风格的 API， `GenericValue<Encoding>` 及 `GenericDocument<Encoding>` 里的 `Encoding` 模板参数是用于指明内存中存储的 JSON 字符串使用哪种编码。因此通常我们会在此参数中使用 `UTF8` 、 `UTF16` 或 `UTF32` 。如何选择，视乎应用软件所使用的操作系统及其他程序库。例如，Windows API 使用 UTF-16 表示 Unicode 字符，而多数的 Linux 发行版本及应用软件则更喜欢 UTF-8。

使用 UTF-16 的 DOM 声明例子：

```
typedef GenericDocument<UTF16<> > WDocument;
typedef GenericValue<UTF16<> > WValue;
```

可以在 DOM's Encoding 一节看到更详细的使用例子。

## 字符类型

从之前的声明中可以看到，每个编码都有一个 `CharType` 模板参数。这可能比较容易混淆，实际上，每个 `CharType` 存储一个编码单元，而不是一个字符（码点）。如之前所谈及，在 UTF-8 中一个码点可能会编码成 1 至 4 个编码单元。

对于 `UTF16(LE|BE)` 及 `UTF32(LE|BE)` 来说， `CharType` 必须分别是一个至少 2 及 4 字节的整数类型。

注意 C++11 新添了 `char16_t` 及 `char32_t` 类型，也可分别用于 `UTF16` 及 `UTF32` 。

## AutoUTF

上述所介绍的编码都是在编译期静态绑定的。换句话说，使用者必须知道内存或流之中使用了哪种编码。然而，有时候我们可能需要读写不同编码的文件，而且这些编码需要在运行时才能决定。

`AutoUTF` 是为此而设计的编码。它根据输入或输出流来选择使用哪种编码。目前它应该与 `EncodedInputStream` 及 `EncodedOutputStream` 结合使用。

## ASCII

虽然 JSON 标准并未提及 ASCII，有时候我们希望写入 7 位的 ASCII JSON，以供未能处理 UTF-8 的应用程序使用。由于任 JSON 都可以把 Unicode 字符表示为 `\uXXXX` 转义序列，JSON 总是可用 ASCII 来编码。

以下的例子把 UTF-8 的 DOM 写成 ASCII 的 JSON：

```
using namespace rapidjson;
Document d; // UTF8<>
// ...
StringBuffer buffer;
Writer<StringBuffer, Document::EncodingType, ASCII<> > writer(buffer);
d.Accept(writer);
std::cout << buffer.GetString();
```

ASCII 可用于输入流。当输入流包含大于 127 的字节，就会导致 `kParseErrorStringInvalidEncoding` 错误。

ASCII 不能用于内存（`Document` 的编码，或 `Reader` 的目标编码），因为它不能表示 Unicode 码点。

## 校验及转码

当 RapidJSON 解析一个 JSON 时，它能校验输入 JSON，判断它是否所标明编码的合法序列。要开启此选项，请把 `kParseValidateEncodingFlag` 加入 `parseFlags` 模板参数。

若输入编码和输出编码并不相同，`Reader` 及 `Writer` 会算把文本转码。在这种情况下，并不需要 `kParseValidateEncodingFlag`，因为它必须解码输入序列。若序列不能被解码，它必然是不合法的。

### 转码器

虽然 RapidJSON 的编码功能是为 JSON 解析／生成而设计，使用者也可以“滥用”它们来为非 JSON 字符串转码。

以下的例子把 UTF-8 字符串转码成 UTF-16：

```
#include "rapidjson/encodings.h"

using namespace rapidjson;

const char* s = "..."; // UTF-8 string
StringStream source(s);
GenericStringBuffer<UTF16<> > target;

bool hasError = false;
while (source.Peek() != '\0')
    if (!Transcoder<UTF8<>, UTF16<> >::Transcode(source, target)) {
        hasError = true;
        break;
    }

if (!hasError) {
    const wchar_t* t = target.GetString();
    // ...
}
```

你也可以用 `AutoUTF` 及对应的流来在运行时设置内源／目的之编码。

# DOM

文档对象模型（Document Object Model, DOM）是一种置于内存中的 JSON 表示方式，以供查询及操作。我们已于 教程 中介绍了 DOM 的基本用法，本节将讲述一些细节及高级用法。

[TOC]

## 模板

教程中使用了 `Value` 和 `Document` 类型。与 `std::string` 相似，这些类型其实是两个模板类的 `typedef`：

```
namespace rapidjson {

template <typename Encoding, typename Allocator = MemoryPoolAllocator<> >
class GenericValue {
    // ...
};

template <typename Encoding, typename Allocator = MemoryPoolAllocator<> >
class GenericDocument : public GenericValue<Encoding, Allocator> {
    // ...
};

typedef GenericValue<UTF8<> > Value;
typedef GenericDocument<UTF8<> > Document;

} // namespace rapidjson
```

使用者可以自定义这些模板参数。

## 编码

`Encoding` 参数指明在内存中的 JSON String 使用哪种编码。可行的选项有 `UTF8`、`UTF16`、`UTF32`。要注意这 3 个类型其实也是模板类。`UTF8<>` 等同 `UTF8<char>`，这代表它使用 `char` 来存储字符串。更多细节可以参考 编码。

这里是一个例子。假设一个 Windows 应用软件希望查询存储于 JSON 中的本地化字符串。Windows 中含 Unicode 的函数使用 UTF-16（宽字符）编码。无论 JSON 文件使用哪种编码，我们都可以把字符串以 UTF-16 形式存储在内存。

```
using namespace rapidjson;

typedef GenericDocument<UTF16<> > WDocument;
typedef GenericValue<UTF16<> > WValue;

FILE* fp = fopen("localization.json", "rb"); // 非 Windows 平台使用 "r"

char readBuffer[256];
FileReadStream bis(fp, readBuffer, sizeof(readBuffer));

AutoUTFInputStream<unsigned, FileReadStream> eis(bis);  // 包装 bis 成 eis

WDocument d;
d.ParseStream<0, AutoUTF<unsigned> >(eis);

const WValue locale(L"ja"); // Japanese

MessageBoxW(hWnd, d[locale].GetString(), L"Test", MB_OK);
```

## 分配器

`Allocator` 定义当 `Document` / `Value` 分配或释放内存时使用那个分配类。`Document` 拥有或引用到一个 `Allocator` 实例。而为了节省内存，`Value` 没有这么做。

`GenericDocument` 的缺省分配器是 `MemoryPoolAllocator`。此分配器实际上会顺序地分配内存，并且不能逐一释放。当要解析一个 JSON 并生成 DOM，这种分配器是非常合适的。

RapidJSON 还提供另一个分配器 `CrtAllocator`，当中 CRT 是 C 运行库（C RunTime library）的缩写。此分配器简单地读用标准的 `malloc()` / `realloc()` / `free()`。当我们需要许多增减操作，这种分配器会更为适合。然而这种分配器远远比 `MemoryPoolAllocator` 低效。

# 解析

`Document` 提供几个解析函数。以下的 (1) 是根本的函数，其他都是调用 (1) 的协助函数。

```
using namespace rapidjson;

// (1) 根本
template <unsigned parseFlags, typename SourceEncoding, typename InputStream>
GenericDocument& GenericDocument::ParseStream(InputStream& is);

// (2) 使用流的编码
template <unsigned parseFlags, typename InputStream>
GenericDocument& GenericDocument::ParseStream(InputStream& is);

// (3) 使用缺省标志
template <typename InputStream>
GenericDocument& GenericDocument::ParseStream(InputStream& is);

// (4) 原位解析
template <unsigned parseFlags>
GenericDocument& GenericDocument::ParseInsitu(Ch* str);

// (5) 原位解析，使用缺省标志
GenericDocument& GenericDocument::ParseInsitu(Ch* str);

// (6) 正常解析一个字符串
template <unsigned parseFlags, typename SourceEncoding>
GenericDocument& GenericDocument::Parse(const Ch* str);

// (7) 正常解析一个字符串，使用 Document 的编码
template <unsigned parseFlags>
GenericDocument& GenericDocument::Parse(const Ch* str);

// (8) 正常解析一个字符串，使用缺省标志
GenericDocument& GenericDocument::Parse(const Ch* str);
```

教程 中的例使用 (8) 去正常解析字符串。而 流 的例子使用前 3 个函数。我们将稍后介绍原位（*In situ*） 解析。

`parseFlags` 是以下位标置的组合：

| 解析位标志 | 意义 |
| --- | --- |
| `kParseNoFlags` | 没有任何标志。 |
| `kParseDefaultFlags` | 缺省的解析选项。它等于 `RAPIDJSON_PARSE_DEFAULT_FLAGS` 宏，此宏定义为 `kParseNoFlags` 。 |
| `kParseInsituFlag` | 原位（破坏性）解析。 |
| `kParseValidateEncodingFlag` | 校验 JSON 字符串的编码。 |
| `kParseIterativeFlag` | 迭代式（调用堆栈大小为常数复杂度）解析。 |
| `kParseStopWhenDoneFlag` | 当从流解析了一个完整的 JSON 根节点之后，停止继续处理余下的流。当使用了此标志，解析器便不会产生 `kParseErrorDocumentRootNotSingular` 错误。可使用本标志去解析同一个流里的多个 JSON。 |
| `kParseFullPrecisionFlag` | 使用完整的精确度去解析数字（较慢）。如不设置此标节，则会使用正常的精确度（较快）。正常精确度会有最多 3 个 ULP 的误差。 |
| `kParseCommentsFlag` | 容许单行 `// ...` 及多行 `/* ... */` 注释（放宽的 JSON 语法）。 |
| `kParseNumbersAsStringsFlag` | 把数字类型解析成字符串。 |
| `kParseTrailingCommasFlag` | 容许在对象和数组结束前含有逗号（放宽的 JSON 语法）。 |
| `kParseNanAndInfFlag` | 容许 `NaN` 、 `Inf` 、 `Infinity` 、 `-Inf` 及 `-Infinity` 作为 `double` 值（放宽的 JSON 语法）。 |

由于使用了非类型模板参数，而不是函数参数，C++ 编译器能为个别组合生成代码，以改善性能及减少代码尺寸（当只用单种特化）。缺点是需要在编译期决定标志。

`SourceEncoding` 参数定义流使用了什么编码。这与 `Document` 的 `Encoding` 不相同。细节可参考 转码和校验 一节。

此外 `InputStream` 是输入流的类型。

## 解析错误

当解析过程顺利完成， `Document` 便会含有解析结果。当过程出现错误，原来的 DOM 会维持不变。可使用 `bool HasParseError()` 、 `ParseErrorCode GetParseError()` 及 `size_t GetParseOffset()` 获取解析的错误状态。

| 解析错误代号 | 描述 |
| --- | --- |
| `kParseErrorNone` | 无错误。 |
| `kParseErrorDocumentEmpty` | 文档是空的。 |
| `kParseErrorDocumentRootNotSingular` | 文档的根后面不能有其它值。 |
| `kParseErrorValueInvalid` | 不合法的值。 |
| `kParseErrorObjectMissName` | Object 成员缺少名字。 |
| `kParseErrorObjectMissColon` | Object 成员名字后缺少冒号。 |
| `kParseErrorObjectMissCommaOrCurlyBracket` | Object 成员后缺少逗号或 `}`。 |
| `kParseErrorArrayMissCommaOrSquareBracket` | Array 元素后缺少逗号或 `]`。 |
| `kParseErrorStringUnicodeEscapeInvalidHex` | String 中的 `\\u` 转义符后含非十六进位数字。 |
| `kParseErrorStringUnicodeSurrogateInvalid` | String 中的代理对（surrogate pair）不合法。 |
| `kParseErrorStringEscapeInvalid` | String 含非法转义字符。 |
| `kParseErrorStringMissQuotationMark` | String 缺少关闭引号。 |
| `kParseErrorStringInvalidEncoding` | String 含非法编码。 |
| `kParseErrorNumberTooBig` | Number 的值太大，不能存储于 `double`。 |
| `kParseErrorNumberMissFraction` | Number 缺少了小数部分。 |
| `kParseErrorNumberMissExponent` | Number 缺少了指数。 |

错误的偏移量定义为从流开始至错误处的字符数量。目前 RapidJSON 不记录错误行号。

要取得错误讯息，RapidJSON 在 `rapidjson/error/en.h` 中提供了英文错误讯息。使用者可以修改它用于其他语言环境，或使用一个自定义的本地化系统。

以下是一个处理错误的例子。

```
#include "rapidjson/document.h"
#include "rapidjson/error/en.h"

// ...
Document d;
if (d.Parse(json).HasParseError()) {
    fprintf(stderr, "\nError(offset %u): %s\n",
        (unsigned)d.GetErrorOffset(),
        GetParseError_En(d.GetParseErrorCode()));
    // ...
}
```

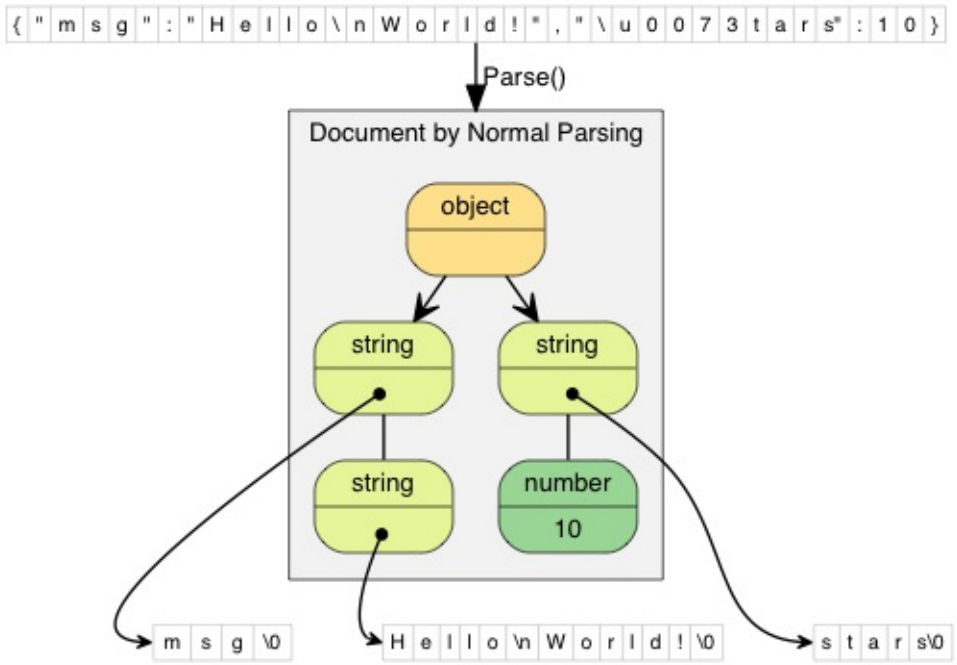
## 原位解析

根据 维基百科:

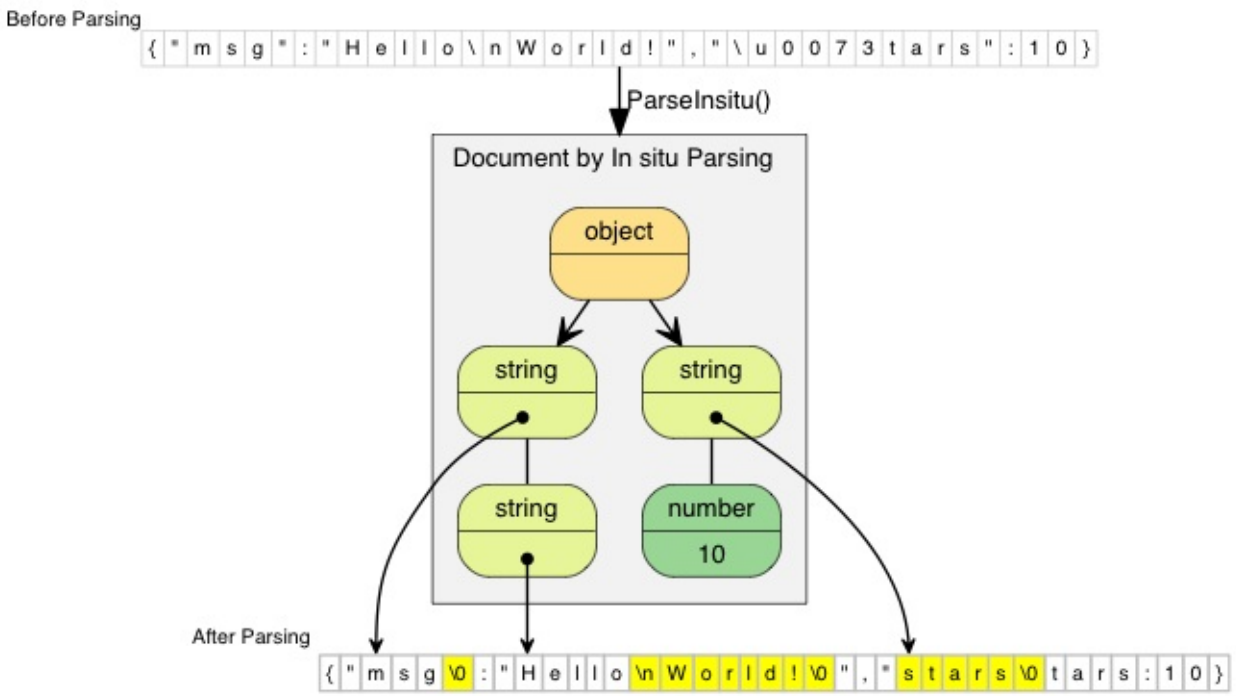
> *In situ* ... is a Latin phrase that translates literally to "on site" or "in position". It means "locally", "on site", "on the premises" or "in place" to describe an event where it takes place, and is used in many different contexts. ... (In computer science) An algorithm is said to be an in situ algorithm, or in-place algorithm, if the extra amount of memory required to execute the algorithm is O(1), that is, does not exceed a constant no matter how large the input. For example, heapsort is an in situ sorting algorithm.
>
> 翻译：*In situ*……是一个拉丁文片语，字面上的意思是指「现场」、「在位置」。在许多不同语境中，它描述一个事件发生的位置，意指「本地」、「现场」、「在处所」、「就位」。……（在计算机科学中）一个算法若称为原位算法，或在位算法，是指执行该算法所需的额外内存空间是 O(1) 的，换句话说，无论输入大小都只需要常数空间。例如，堆排序是一个原位排序算法。

在正常的解析过程中，对 JSON string 解码并复制至其他缓冲区是一个很大的开销。原位解析（*in situ* parsing）把这些 JSON string 直接解码于它原来存储的地方。由于解码后的 string 长度总是短于或等于原来储存于 JSON 的 string，所以这是可行的。在这个语境下，对 JSON string 进行解码是指处理转义符，如 `"\n"`、`"\u1234"` 等，以及在 string 末端加入空终止符号 ( `'\0'` )。

以下的图比较正常及原位解析。JSON string 值包含指向解码后的字符串。




在正常解析中，解码后的字符串被复制至全新分配的缓冲区中。`"\\n"`（2 个字符）被解码成 `"\n"`（1 个字符）。`"\\u0073"`（6 个字符）被解码成 `"s"`（1 个字符）。




原位解析直接修改了原来的 JSON。图中高亮了被更新的字符。若 JSON string 不含转义符，例如 `"msg"`，那么解析过程仅仅是以空字符代替结束双引号。

由于原位解析修改了输入，其解析 API 需要 `char*` 而非 `const char*`。

```
// 把整个文件读入 buffer
FILE* fp = fopen("test.json", "r");
fseek(fp, 0, SEEK_END);
size_t filesize = (size_t)ftell(fp);
fseek(fp, 0, SEEK_SET);
char* buffer = (char*)malloc(filesize + 1);
size_t readLength = fread(buffer, 1, filesize, fp);
buffer[readLength] = '\0';
fclose(fp);

// 原位解析 buffer 至 d，buffer 内容会被修改。
Document d;
d.ParseInsitu(buffer);

// 在此查询、修改 DOM……

free(buffer);
// 注意：在这个位置，d 可能含有指向已被释放的 buffer 的悬空指针
```

JSON string 会被打上 const-string 的标志。但它们可能并非真正的「常数」。它的生命周期取决于存储 JSON 的缓冲区。

原位解析把分配开销及内存复制减至最小。通常这样做能改善缓存一致性，而这对现代计算机来说是一个重要的性能因素。

原位解析有以下限制：

1. 整个 JSON 须存储在内存之中。

2. 流的来源缓码与文档的目标编码必须相同。
3. 需要保留缓冲区，直至文档不再被使用。
4. 若 DOM 需要在解析后被长期使用，而 DOM 内只有很少 JSON string，保留缓冲区可能造成内存浪费。

原位解析最适合用于短期的、用完即弃的 JSON。实际应用中，这些场合是非常普遍的，例如反序列化 JSON 至 C++ 对象、处理以 JSON 表示的 web 请求等。

## 转码与校验

RapidJSON 内部支持不同 Unicode 格式（正式的术语是 UCS 变换格式）间的转换。在 DOM 解析时，流的来源编码与 DOM 的编码可以不同。例如，来源流可能含有 UTF-8 的 JSON，而 DOM 则使用 UTF-16 编码。在 EncodedInputStream 一节里有一个例子。

当从 DOM 输出一个 JSON 至输出流之时，也可以使用转码功能。在 EncodedOutputStream 一节里有一个例子。

在转码过程中，会把来源 string 解码成 Unicode 码点，然后把码点编码成目标格式。在解码时，它会校验来源 string 的字节序列是否合法。若遇上非合法序列，解析器会停止并返回 `kParseErrorStringInvalidEncoding` 错误。

当来源编码与 DOM 的编码相同，解析器缺省地 不会 校验序列。使用者可开启 `kParseValidateEncodingFlag` 去强制校验。

# 技巧

这里讨论一些 DOM API 的使用技巧。

## 把 DOM 作为 SAX 事件发表者

在 RapidJSON 中，利用 `Writer` 把 DOM 生成 JSON 的做法，看来有点奇怪。

```
// ...
Writer<StringBuffer> writer(buffer);
d.Accept(writer);
```

实际上，`Value::Accept()` 是负责发布该值相关的 SAX 事件至处理器的。通过这个设计，`Value` 及 `Writer` 解除了偶合。`Value` 可生成 SAX 事件，而 `Writer` 则可以处理这些事件。

使用者可以创建自定义的处理器，去把 DOM 转换成其它格式。例如，一个把 DOM 转换成 XML 的处理器。

要知道更多关于 SAX 事件与处理器，可参阅 SAX。

## 使用者缓冲区

许多应用软件可能需要尽量减少内存分配。

`MemoryPoolAllocator` 可以帮助这方面，它容许使用者提供一个缓冲区。该缓冲区可能置于程序堆栈，或是一个静态分配的「草稿缓冲区（scratch buffer）」（一个静态／全局的数组），用于储存临时数据。

`MemoryPoolAllocator` 会先用使用者缓冲区去解决分配请求。当使用者缓冲区用完，就会从基础分配器（缺省为 `CrtAllocator`）分配一块内存。

以下是使用堆栈内存的例子，第一个分配器用于存储值，第二个用于解析时的临时缓冲。

```
typedef GenericDocument<UTF8<>, MemoryPoolAllocator<>, MemoryPoolAllocator<>> DocumentT
ype;
char valueBuffer[4096];
char parseBuffer[1024];
MemoryPoolAllocator<> valueAllocator(valueBuffer, sizeof(valueBuffer));
MemoryPoolAllocator<> parseAllocator(parseBuffer, sizeof(parseBuffer));
DocumentType d(&valueAllocator, sizeof(parseBuffer), &parseAllocator);
d.Parse(json);
```

若解析时分配总量少于 4096+1024 字节时，这段代码不会造成任何堆内存分配（经 `new` 或 `malloc()`）。

使用者可以通过 `MemoryPoolAllocator::Size()` 查询当前已分的内存大小。那么使用者可以拟定使用者缓冲区的合适大小。

# SAX

"SAX" 此术语源于 Simple API for XML。我们借了此术语去套用在 JSON 的解析及生成。

在 RapidJSON 中，`Reader`（`GenericReader<...>` 的 typedef）是 JSON 的 SAX 风格解析器，而 `Writer`（`GenericWriter<...>` 的 typedef）则是 JSON 的 SAX 风格生成器。

[TOC]

# Reader

`Reader` 从输入流解析一个 JSON。当它从流中读取字符时，它会基于 JSON 的语法去分析字符，并向处理器发送事件。

例如，以下是一个 JSON。

```
{
    "hello": "world",
    "t": true ,
    "f": false,
    "n": null,
    "i": 123,
    "pi": 3.1416,
    "a": [1, 2, 3, 4]
}
```

当一个 `Reader` 解析此 JSON 时，它会顺序地向处理器发送以下的事件：

```
StartObject()
Key("hello", 5, true)
String("world", 5, true)
Key("t", 1, true)
Bool(true)
Key("f", 1, true)
Bool(false)
Key("n", 1, true)
Null()
Key("i")
UInt(123)
Key("pi")
Double(3.1416)
Key("a")
StartArray()
Uint(1)
Uint(2)
Uint(3)
Uint(4)
EndArray(4)
EndObject(7)
```

除了一些事件参数需要再作解释，这些事件可以轻松地与 JSON 对上。我们可以看看 `simplereader` 例子怎样产生和以上完全相同的结果：

```
#include "rapidjson/reader.h"
#include <iostream>

using namespace rapidjson;
using namespace std;

struct MyHandler : public BaseReaderHandler<UTF8<>, MyHandler> {
    bool Null() { cout << "Null()" << endl; return true; }
    bool Bool(bool b) { cout << "Bool(" << boolalpha << b << ")" << endl; return true;
}
    bool Int(int i) { cout << "Int(" << i << ")" << endl; return true; }
    bool Uint(unsigned u) { cout << "Uint(" << u << ")" << endl; return true; }
    bool Int64(int64_t i) { cout << "Int64(" << i << ")" << endl; return true; }
    bool Uint64(uint64_t u) { cout << "Uint64(" << u << ")" << endl; return true; }
    bool Double(double d) { cout << "Double(" << d << ")" << endl; return true; }
    bool String(const char* str, SizeType length, bool copy) {
        cout << "String(" << str << ", " << length << ", " << boolalpha << copy << ")"
<< endl;
        return true;
    }
    bool StartObject() { cout << "StartObject()" << endl; return true; }
    bool Key(const char* str, SizeType length, bool copy) {
        cout << "Key(" << str << ", " << length << ", " << boolalpha << copy << ")" <<
endl;
        return true;
    }
    bool EndObject(SizeType memberCount) { cout << "EndObject(" << memberCount << ")" <
< endl; return true; }
    bool StartArray() { cout << "StartArray()" << endl; return true; }
    bool EndArray(SizeType elementCount) { cout << "EndArray(" << elementCount << ")" <
< endl; return true; }
};

void main() {
    const char json[] = " { \"hello\" : \"world\", \"t\" : true , \"f\" : false, \"n\":
 null, \"i\":123, \"pi\": 3.1416, \"a\":[1, 2, 3, 4] } ";

    MyHandler handler;
    Reader reader;
    StringStream ss(json);
    reader.Parse(ss, handler);
}
```

注意 RapidJSON 使用模板去静态绑定 `Reader` 类型及处理器的类形，而不是使用含虚函数的类。这个范式可以通过把函数内联而改善性能。

## 处理器

如前例所示，使用者需要实现一个处理器（handler），用于处理来自 `Reader` 的事件（函数调用）。处理器必须包含以下的成员函数。

```
class Handler {
    bool Null();
    bool Bool(bool b);
    bool Int(int i);
    bool Uint(unsigned i);
    bool Int64(int64_t i);
    bool Uint64(uint64_t i);
    bool Double(double d);
    bool RawNumber(const Ch* str, SizeType length, bool copy);
    bool String(const Ch* str, SizeType length, bool copy);
    bool StartObject();
    bool Key(const Ch* str, SizeType length, bool copy);
    bool EndObject(SizeType memberCount);
    bool StartArray();
    bool EndArray(SizeType elementCount);
};
```

当 `Reader` 遇到 JSON null 值时会调用 `Null()`。

当 `Reader` 遇到 JSON true 或 false 值时会调用 `Bool(bool)`。

当 `Reader` 遇到 JSON number，它会选择一个合适的 C++ 类型映射，然后调用 `Int(int)`、`Uint(unsigned)`、`Int64(int64_t)`、`Uint64(uint64_t)` 及 `Double(double)` 的其中之一个。若开启了 `kParseNumbersAsStrings` 选项，`Reader` 便会改为调用 `RawNumber()`。

当 `Reader` 遇到 JSON string，它会调用 `String(const char* str, SizeType length, bool copy)`。第一个参数是字符串的指针。第二个参数是字符串的长度（不包含空终止符号）。注意 RapidJSON 支持字串中含有空字符 `\0`。若出现这种情况，便会有 `strlen(str) < length`。最后的 `copy` 参数表示处理器是否需要复制该字符串。在正常解析时，`copy = true`。仅当使用原位解析时，`copy = false`。此外，还要注意字符的类型与目标编码相关，我们稍后会再谈这一点。

当 `Reader` 遇到 JSON object 的开始之时，它会调用 `StartObject()`。JSON 的 object 是一个键值对（成员）的集合。若 object 包含成员，它会先为成员的名字调用 `Key()`，然后再按值的类型调用函数。它不断调用这些键值对，直至最终调用 `EndObject(SizeType memberCount)`。注意 `memberCount` 参数对处理器来说只是协助性质，使用者可能不需要此参数。

JSON array 与 object 相似，但更简单。在 array 开始时，`Reader` 会调用 `BeginArary()`。若 array 含有元素，它会按元素的类型来读用函数。相似地，最后它会调用 `EndArray(SizeType elementCount)`，其中 `elementCount` 参数对处理器来说只是协助性质。

每个处理器函数都返回一个 `bool`。正常它们应返回 `true`。若处理器遇到错误，它可以返回 `false` 去通知事件发送方停止继续处理。

例如，当我们用 `Reader` 解析一个 JSON 时，处理器检测到该 JSON 并不符合所需的 schema，那么处理器可以返回 `false`，令 `Reader` 停止之后的解析工作。而 `Reader` 会进入一个错误状态，并以 `kParseErrorTermination` 错误码标识。

## GenericReader

前面提及，`Reader` 是 `GenericReader` 模板类的 typedef：

```
namespace rapidjson {

template <typename SourceEncoding, typename TargetEncoding, typename Allocator = Memory
PoolAllocator<> >
class GenericReader {
    // ...
};

typedef GenericReader<UTF8<>, UTF8<> > Reader;

} // namespace rapidjson
```

`Reader` 使用 UTF-8 作为来源及目标编码。来源编码是指 JSON 流的编码。目标编码是指 `String()` 的 `str` 参数所用的编码。例如，要解析一个 UTF-8 流并输出至 UTF-16 string 事件，你需要这么定义一个 reader：

```
GenericReader<UTF8<>, UTF16<> > reader;
```

注意到 `UTF16` 的缺省类型是 `wchar_t`。因此这个 `reader` 需要调用处理器的 `String(const wchar_t*, SizeType, bool)`。

第三个模板参数 `Allocator` 是内部数据结构（实际上是一个堆栈）的分配器类型。

### 解析

`Reader` 的唯一功能就是解析 JSON。

```
template <unsigned parseFlags, typename InputStream, typename Handler>
bool Parse(InputStream& is, Handler& handler);

// 使用 parseFlags = kDefaultParseFlags
template <typename InputStream, typename Handler>
bool Parse(InputStream& is, Handler& handler);
```

若在解析中出现错误，它会返回 `false`。使用者可调用 `bool HasParseEror()`，`ParseErrorCode GetParseErrorCode()` 及 `size_t GetErrorOffset()` 获取错误状态。实际上 `Document` 使用这些 `Reader` 函数去获取解析错误。请参考 DOM 去了解有关解析错误的细节。

## Writer

`Reader` 把 JSON 转换（解析）成为事件。`Writer` 做完全相反的事情。它把事件转换成 JSON。

`Writer` 是非常容易使用的。若你的应用程序只需把一些数据转换成 JSON，可能直接使用 `Writer`，会比建立一个 `Document` 然后用 `Writer` 把它转换成 JSON 更加方便。

在 `simplewriter` 例子里，我们做 `simplereader` 完全相反的事情。

```
#include "rapidjson/writer.h"
#include "rapidjson/stringbuffer.h"
#include <iostream>

using namespace rapidjson;
using namespace std;

void main() {
    StringBuffer s;
    Writer<StringBuffer> writer(s);

    writer.StartObject();
    writer.Key("hello");
    writer.String("world");
    writer.Key("t");
    writer.Bool(true);
    writer.Key("f");
    writer.Bool(false);
    writer.Key("n");
    writer.Null();
    writer.Key("i");
    writer.Uint(123);
    writer.Key("pi");
    writer.Double(3.1416);
    writer.Key("a");
    writer.StartArray();
    for (unsigned i = 0; i < 4; i++)
        writer.Uint(i);
    writer.EndArray();
    writer.EndObject();

    cout << s.GetString() << endl;
}
```

```
{"hello":"world","t":true,"f":false,"n":null,"i":123,"pi":3.1416,"a":[0,1,2,3]}
```

`String()` 及 `Key()` 各有两个重载。一个是如处理器 concept 般，有 3 个参数。它能处理含空字符的字符串。另一个是如上中使用的较简单版本。

注意到，例子代码中的 `EndArray()` 及 `EndObject()` 并没有参数。可以传递一个 `SizeType` 的参数，但它会被 `Writer` 忽略。

你可能会怀疑，为什么不使用 `sprintf()` 或 `std::stringstream` 去建立一个 JSON？

这有几个原因：

1. `Writer` 必然会输出一个结构良好（well-formed）的 JSON。若然有错误的事件次序（如 `Int()` 紧随 `StartObject()` 出现），它会在调试模式中产生断言失败。
2. `Writer::String()` 可处理字符串转义（如把码点 `U+000A` 转换成 `\n`）及进行 Unicode 转码。
3. `Writer` 一致地处理 number 的输出。

4. `Writer` 实现了事件处理器 concept。可用于处理来自 `Reader`、`Document` 或其他事件发生器。
5. `Writer` 可对不同平台进行优化。

无论如何，使用 `Writer` API 去生成 JSON 甚至乎比这些临时方法更简单。

## 模板

`Writer` 与 `Reader` 有少许设计区别。`Writer` 是一个模板类，而不是一个 typedef。并没有 `GenericWriter`。以下是 `Writer` 的声明。

```
namespace rapidjson {

template<typename OutputStream, typename SourceEncoding = UTF8<>, typename TargetEncodi
ng = UTF8<>, typename Allocator = CrtAllocator<> >
class Writer {
public:
    Writer(OutputStream& os, Allocator* allocator = 0, size_t levelDepth = kDefaultLeve
lDepth)
// ...
};

} // namespace rapidjson
```

`OutputStream` 模板参数是输出流的类型。它的类型不可以被自动推断，必须由使用者提供。

`SourceEncoding` 模板参数指定了 `String(const Ch*, ...)` 的编码。

`TargetEncoding` 模板参数指定输出流的编码。

`Allocator` 是分配器的类型，用于分配内部数据结构（一个堆栈）。

`writeFlags` 是以下位标志的组合：

| 写入位标志 | 意义 |
| --- | --- |
| `kWriteNoFlags` | 没有任何标志。 |
| `kWriteDefaultFlags` | 缺省的解析选项。它等于 `RAPIDJSON_WRITE_DEFAULT_FLAGS` 宏，此宏定义为 `kWriteNoFlags`。 |
| `kWriteValidateEncodingFlag` | 校验 JSON 字符串的编码。 |
| `kWriteNanAndInfFlag` | 容许写入 `Infinity`, `-Infinity` 及 `NaN`。 |

此外，`Writer` 的构造函数有一 `levelDepth` 参数。存储每层阶信息的初始内存分配量受此参数影响。

# PrettyWriter

`Writer` 所输出的是没有空格字符的最紧凑 JSON，适合网络传输或储存，但不适合人类阅读。

因此，RapidJSON 提供了一个 `PrettyWriter`，它在输出中加入缩进及换行。

`PrettyWriter` 的用法与 `Writer` 几乎一样，不同之处是 `PrettyWriter` 提供了一个 `SetIndent(Ch indentChar, unsigned indentCharCount)` 函数。缺省的缩进是 4 个空格。

## 完整性及重置

一个 `Writer` 只可输出单个 JSON，其根节点可以是任何 JSON 类型。当处理完单个根节点事件（如 `String()`），或匹配的最后 `EndObject()` 或 `EndArray()` 事件，输出的 JSON 是结构完整（well-formed）及完整的。使用者可调用 `Writer::IsComplete()` 去检测完整性。

当 JSON 完整时，`Writer` 不能再接受新的事件。不然其输出便会是不合法的（例如有超过一个根节点）。为了重新利用 `Writer` 对象，使用者可调用 `Writer::Reset(OutputStream& os)` 去重置其所有内部状态及设置新的输出流。

# 技巧

## 解析 JSON 至自定义结构

`Document` 的解析功能完全依靠 `Reader`。实际上 `Document` 是一个处理器，在解析 JSON 时接收事件去建立一个 DOM。

使用者可以直接使用 `Reader` 去建立其他数据结构。这消除了建立 DOM 的步骤，从而减少了内存开销并改善性能。

在以下的 `messagereader` 例子中，`ParseMessages()` 解析一个 JSON，该 JSON 应该是一个含键值对的 object。

```
#include "rapidjson/reader.h"
#include "rapidjson/error/en.h"
#include <iostream>
#include <string>
#include <map>

using namespace std;
using namespace rapidjson;

typedef map<string, string> MessageMap;

struct MessageHandler
    : public BaseReaderHandler<UTF8<>, MessageHandler> {
    MessageHandler() : state_(kExpectObjectStart) {
    }

    bool StartObject() {
        switch (state_) {
        case kExpectObjectStart:
            state_ = kExpectNameOrObjectEnd;
            return true;
        default:
            return false;
```

```
        }
    }

    bool String(const char* str, SizeType length, bool) {
        switch (state_) {
        case kExpectNameOrObjectEnd:
            name_ = string(str, length);
            state_ = kExpectValue;
            return true;
        case kExpectValue:
            messages_.insert(MessageMap::value_type(name_, string(str, length)));
            state_ = kExpectNameOrObjectEnd;
            return true;
        default:
            return false;
        }
    }

    bool EndObject(SizeType) { return state_ == kExpectNameOrObjectEnd; }

    bool Default() { return false; } // All other events are invalid.

    MessageMap messages_;
    enum State {
        kExpectObjectStart,
        kExpectNameOrObjectEnd,
        kExpectValue,
    }state_;
    std::string name_;
};

void ParseMessages(const char* json, MessageMap& messages) {
    Reader reader;
    MessageHandler handler;
    StringStream ss(json);
    if (reader.Parse(ss, handler))
        messages.swap(handler.messages_);   // Only change it if success.
    else {
        ParseErrorCode e = reader.GetParseErrorCode();
        size_t o = reader.GetErrorOffset();
        cout << "Error: " << GetParseError_En(e) << endl;;
        cout << " at offset " << o << " near '" << string(json).substr(o, 10) << "...'"
 << endl;
    }
}

int main() {
    MessageMap messages;

    const char* json1 = "{ \"greeting\" : \"Hello!\", \"farewell\" : \"bye-bye!\" }";
    cout << json1 << endl;
    ParseMessages(json1, messages);
```

```
    for (MessageMap::const_iterator itr = messages.begin(); itr != messages.end(); ++it
r)
        cout << itr->first << ": " << itr->second << endl;

    cout << endl << "Parse a JSON with invalid schema." << endl;
    const char* json2 = "{ \"greeting\" : \"Hello!\", \"farewell\" : \"bye-bye!\", \"fo
o\" : {} }";
    cout << json2 << endl;
    ParseMessages(json2, messages);

    return 0;
}
```

```
{ "greeting" : "Hello!", "farewell" : "bye-bye!" }
farewell: bye-bye!
greeting: Hello!

Parse a JSON with invalid schema.
{ "greeting" : "Hello!", "farewell" : "bye-bye!", "foo" : {} }
Error: Terminate parsing due to Handler error.
 at offset 59 near '} }...'
```

第一个 JSON（`json1`）被成功地解析至 `MessageMap`。由于 `MessageMap` 是一个 `std::map`，打印次序按键值排序。此次序与 JSON 中的次序不同。

在第二个 JSON（`json2`）中，`foo` 的值是一个空 object。由于它是一个 object，`MessageHandler::StartObject()` 会被调用。然而，在 `state_ = kExpectValue` 的情况下，该函数会返回 `false`，并导致解析过程终止。错误代码是 `kParseErrorTermination`。

## 过滤 JSON

如前面提及过，`Writer` 可处理 `Reader` 发出的事件。`example/condense/condense.cpp` 例子简单地设置 `Writer` 作为一个 `Reader` 的处理器，因此它能移除 JSON 中的所有空白字符。`example/pretty/pretty.cpp` 例子使用同样的关系，只是以 `PrettyWriter` 取代 `Writer`。因此 `pretty` 能够重新格式化 JSON，加入缩进及换行。

实际上，我们可以使用 SAX 风格 API 去加入（多个）中间层去过滤 JSON 的内容。例如 `capitalize` 例子可以把所有 JSON string 改为大写。

```
#include "rapidjson/reader.h"
#include "rapidjson/writer.h"
#include "rapidjson/filereadstream.h"
#include "rapidjson/filewritestream.h"
#include "rapidjson/error/en.h"
#include <vector>
#include <cctype>

using namespace rapidjson;
```

```
template<typename OutputHandler>
struct CapitalizeFilter {
    CapitalizeFilter(OutputHandler& out) : out_(out), buffer_() {
    }

    bool Null() { return out_.Null(); }
    bool Bool(bool b) { return out_.Bool(b); }
    bool Int(int i) { return out_.Int(i); }
    bool Uint(unsigned u) { return out_.Uint(u); }
    bool Int64(int64_t i) { return out_.Int64(i); }
    bool Uint64(uint64_t u) { return out_.Uint64(u); }
    bool Double(double d) { return out_.Double(d); }
    bool RawNumber(const char* str, SizeType length, bool copy) { return out_.RawNumber
(str, length, copy); }
    bool String(const char* str, SizeType length, bool) {
        buffer_.clear();
        for (SizeType i = 0; i < length; i++)
            buffer_.push_back(std::toupper(str[i]));
        return out_.String(&buffer_.front(), length, true); // true = output handler ne
ed to copy the string
    }
    bool StartObject() { return out_.StartObject(); }
    bool Key(const char* str, SizeType length, bool copy) { return String(str, length,
copy); }
    bool EndObject(SizeType memberCount) { return out_.EndObject(memberCount); }
    bool StartArray() { return out_.StartArray(); }
    bool EndArray(SizeType elementCount) { return out_.EndArray(elementCount); }

    OutputHandler& out_;
    std::vector<char> buffer_;
};

int main(int, char*[]) {
    // Prepare JSON reader and input stream.
    Reader reader;
    char readBuffer[65536];
    FileReadStream is(stdin, readBuffer, sizeof(readBuffer));

    // Prepare JSON writer and output stream.
    char writeBuffer[65536];
    FileWriteStream os(stdout, writeBuffer, sizeof(writeBuffer));
    Writer<FileWriteStream> writer(os);

    // JSON reader parse from the input stream and let writer generate the output.
    CapitalizeFilter<Writer<FileWriteStream> > filter(writer);
    if (!reader.Parse(is, filter)) {
        fprintf(stderr, "\nError(%u): %s\n", (unsigned)reader.GetErrorOffset(), GetPars
eError_En(reader.GetParseErrorCode()));
        return 1;
    }

    return 0;
}
```

注意到，不可简单地把 JSON 当作字符串去改为大写。例如：

```
["Hello\nWorld"]
```

简单地把整个 JSON 转为大写的话会产生错误的转义符：

```
["HELLO\NWORLD"]
```

而 `capitalize` 就会产生正确的结果：

```
["HELLO\nWORLD"]
```

我们还可以开发更复杂的过滤器。然而，由于 SAX 风格 API 在某一时间点只能提供单一事件的信息，使用者需要自行记录一些上下文信息（例如从根节点起的路径、储存其他相关值）。对于处理某些情况，用 DOM 会比 SAX 更容易实现。

# Schema

（本功能于 v1.1.0 发布）

JSON Schema 是描述 JSON 格式的一个标准草案。一个 schema 本身也是一个 JSON。使用 JSON Schema 去校验 JSON，可以让你的代码安全地访问 DOM，而无须检查类型或键值是否存在等。这也能确保输出的 JSON 是符合指定的 schema。

RapidJSON 实现了一个 JSON Schema Draft v4 的校验器。若你不熟悉 JSON Schema，可以参考 Understanding JSON Schema。

[TOC]

## 基本用法

首先，你要把 JSON Schema 解析成 `Document`，再把它编译成一个 `SchemaDocument`。

然后，利用该 `SchemaDocument` 创建一个 `SchemaValidator`。它与 `Writer` 相似，都是能够处理 SAX 事件的。因此，你可以用 `document.Accept(validator)` 去校验一个 JSON，然后再获取校验结果。

```
#include "rapidjson/schema.h"

// ...

Document sd;
if (sd.Parse(schemaJson).HasParseError()) {
    // 此 schema 不是合法的 JSON
    // ...
}
SchemaDocument schema(sd); // 把一个 Document 编译至 SchemaDocument
// 之后不再需要 sd

Document d;
if (d.Parse(inputJson).HasParseError()) {
    // 输入不是一个合法的 JSON
    // ...
}

SchemaValidator validator(schema);
if (!d.Accept(validator)) {
    // 输入的 JSON 不合乎 schema
    // 打印诊断信息
    StringBuffer sb;
    validator.GetInvalidSchemaPointer().StringifyUriFragment(sb);
    printf("Invalid schema: %s\n", sb.GetString());
    printf("Invalid keyword: %s\n", validator.GetInvalidSchemaKeyword());
    sb.Clear();
    validator.GetInvalidDocumentPointer().StringifyUriFragment(sb);
    printf("Invalid document: %s\n", sb.GetString());
}
```

一些注意点：

- 一个 `SchemaDocment` 能被多个 `SchemaValidator` 引用。它不会被 `SchemaValidator` 修改。
- 可以重复使用一个 `SchemaValidator` 来校验多个文件。在校验其他文件前，须先调用 `validator.Reset()`。

## 在解析／生成时进行校验

与大部分 JSON Schema 校验器有所不同，RapidJSON 提供了一个基于 SAX 的 schema 校验器实现。因此，你可以在输入流解析 JSON 的同时进行校验。若校验器遇到一个与 schema 不符的值，就会立即终止解析。这设计对于解析大型 JSON 文件时特别有用。

### DOM 解析

在使用 DOM 进行解析时，`Document` 除了接收 SAX 事件外，还需做一些准备及结束工作，因此，为了连接 `Reader`、`SchemaValidator` 和 `Document` 要做多一点事情。`SchemaValidatingReader` 是一个辅助类去做那些工作。

```
#include "rapidjson/filereadstream.h"

// ...
SchemaDocument schema(sd); // 把一个 Document 编译至 SchemaDocument

// 使用 reader 解析 JSON
FILE* fp = fopen("big.json", "r");
FileReadStream is(fp, buffer, sizeof(buffer));

// 用 reader 解析 JSON，校验它的 SAX 事件，并存储至 d
Document d;
SchemaValidatingReader<kParseDefaultFlags, FileReadStream, UTF8<> > reader(is, schema);
d.Populate(reader);

if (!reader.GetParseResult()) {
    // 不是一个合法的 JSON
    // 当 reader.GetParseResult().Code() == kParseErrorTermination,
    // 它可能是被以下原因中止：
    // (1) 校验器发现 JSON 不合乎 schema；或
    // (2) 输入流有 I/O 错误。

    // 检查校验结果
    if (!reader.IsValid()) {
        // 输入的 JSON 不合乎 schema
        // 打印诊断信息
        StringBuffer sb;
        reader.GetInvalidSchemaPointer().StringifyUriFragment(sb);
        printf("Invalid schema: %s\n", sb.GetString());
        printf("Invalid keyword: %s\n", reader.GetInvalidSchemaKeyword());
        sb.Clear();
        reader.GetInvalidDocumentPointer().StringifyUriFragment(sb);
        printf("Invalid document: %s\n", sb.GetString());
    }
}
```

## SAX 解析

使用 SAX 解析时，情况就简单得多。若只需要校验 JSON 而无需进一步处理，那么仅需要：

```
SchemaValidator validator(schema);
Reader reader;
if (!reader.Parse(stream, validator)) {
    if (!validator.IsValid()) {
        // ...
    }
}
```

这种方式和 schemavalidator 例子完全相同。这带来的独特优势是，无论 JSON 多巨大，永远维持低内存用量（内存用量只与 Schema 的复杂度相关）。

若你需要进一步处理 SAX 事件，便可使用模板类 `GenericSchemaValidator` 去设置校验器的输出 `Handler`：

```
MyHandler handler;
GenericSchemaValidator<SchemaDocument, MyHandler> validator(schema, handler);
Reader reader;
if (!reader.Parse(ss, validator)) {
    if (!validator.IsValid()) {
        // ...
    }
}
```

## 生成

我们也可以在生成（serialization）的时候进行校验。这能确保输出的 JSON 符合一个 JSON Schema。

```
StringBuffer sb;
Writer<StringBuffer> writer(sb);
GenericSchemaValidator<SchemaDocument, Writer<StringBuffer> > validator(s, writer);
if (!d.Accept(validator)) {
    // Some problem during Accept(), it may be validation or encoding issues.
    if (!validator.IsValid()) {
        // ...
    }
}
```

当然，如果你的应用仅需要 SAX 风格的生成，那么只需要把 SAX 事件由原来发送到 `Writer`，改为发送到 `SchemaValidator`。

# 远程 Schema

JSON Schema 支持 `$ref` 关键字，它是一个 JSON pointer 引用至一个本地（local）或远程（remote）schema。本地指针的首字符是 `#`，而远程指针是一个相对或绝对 URI。例如：

```
{ "$ref": "definitions.json#/address" }
```

由于 `SchemaDocument` 并不知道如何处理那些 URI，它需要使用者提供一个 `IRemoteSchemaDocumentProvider` 的实例去处理。

```
class MyRemoteSchemaDocumentProvider : public IRemoteSchemaDocumentProvider {
public:
    virtual const SchemaDocument* GetRemoteDocument(const char* uri, SizeType length) {
        // Resolve the uri and returns a pointer to that schema.
    }
};

// ...

MyRemoteSchemaDocumentProvider provider;
SchemaDocument schema(sd, &provider);
```

## 标准的符合程度

RapidJSON 通过了 JSON Schema Test Suite (Json Schema draft 4) 中 263 个测试的 262 个。

没通过的测试是 `refRemote.json` 中的 "change resolution scope" - "changed scope ref invalid"。这是由于未实现 `id` schema 关键字及 URI 合并功能。

除此以外，关于字符串类型的 `format` schema 关键字也会被忽略，因为标准中并没需求必须实现。

## 正则表达式

`pattern` 及 `patternProperties` 这两个 schema 关键字使用了正则表达式去匹配所需的模式。

RapidJSON 实现了一个简单的 NFA 正则表达式引擎，并预设使用。它支持以下语法。

| 语法 | 描述 |
| --- | --- |
| `ab` | 串联 |
| `a\|b` | 交替 |
| `a?` | 零或一次 |
| `a*` | 零或多次 |
| `a+` | 一或多次 |
| `a{3}` | 刚好 3 次 |
| `a{3,}` | 至少 3 次 |
| `a{3,5}` | 3 至 5 次 |
| `(ab)` | 分组 |
| `^a` | 在开始处 |
| `a$` | 在结束处 |
| `.` | 任何字符 |
| `[abc]` | 字符组 |
| `[a-c]` | 字符组范围 |
| `[a-z0-9_]` | 字符组组合 |
| `[^abc]` | 字符组取反 |
| `[^a-c]` | 字符组范围取反 |
| `[\b]` | 退格符 (U+0008) |
| `\\|` , `\\` , ... | 转义字符 |
| `\f` | 馈页 (U+000C) |
| `\n` | 馈行 (U+000A) |
| `\r` | 回车 (U+000D) |
| `\t` | 制表 (U+0009) |
| `\v` | 垂直制表 (U+000B) |

对于使用 C++11 编译器的使用者，也可使用 `std::regex` ，只需定义 `RAPIDJSON_SCHEMA_USE_INTERNALREGEX=0` 及 `RAPIDJSON_SCHEMA_USE_STDREGEX=1` 。若你的 schema 无需使用 `pattern` 或 `patternProperties` ，可以把两个宏都设为零，以禁用此功能，这样做可节省一些代码体积。

## 性能

大部分 C++ JSON 库都未支持 JSON Schema。因此我们尝试按照 json-schema-benchmark 去评估 RapidJSON 的 JSON Schema 校验器。该评测测试了 11 个运行在 node.js 上的 JavaScript 库。

该评测校验 JSON Schema Test Suite 中的测试，当中排除了一些测试套件及个别测试。我们在 `schematest.cpp` 实现了相同的评测。

在 MacBook Pro (2.8 GHz Intel Core i7) 上收集到以下结果。

| 校验器 | 相对速度 | 每秒执行的测试数目 |
| --- | --- | --- |
| RapidJSON | 155% | 30682 |
| `ajv` | 100% | 19770 (± 1.31%) |
| `is-my-json-valid` | 70% | 13835 (± 2.84%) |
| `jsen` | 57.7% | 11411 (± 1.27%) |
| `schemasaurus` | 26% | 5145 (± 1.62%) |
| `themis` | 19.9% | 3935 (± 2.69%) |
| `z-schema` | 7% | 1388 (± 0.84%) |
| `jsck` | 3.1% | 606 (± 2.84%) |
| `jsonschema` | 0.9% | 185 (± 1.01%) |
| `skeemas` | 0.8% | 154 (± 0.79%) |
| tv4 | 0.5% | 93 (± 0.94%) |
| `jayschema` | 0.1% | 21 (± 1.14%) |

换言之，RapidJSON 比最快的 JavaScript 库（ajv）快约 1.5x。比最慢的快 1400x。

# 性能

有一个 native JSON benchmark collection 项目，能评估 37 个 JSON 库在不同操作下的速度、内存用量及代码大小。

RapidJSON 0.1 版本的性能测试文章位于 这里.

此外，你也可以参考以下这些第三方的评测。

## 第三方评测

- Basic benchmarks for miscellaneous C++ JSON parsers and generators by Mateusz Loskot (Jun 2013)
    - casablanca
    - json_spirit
    - jsoncpp
    - libjson
    - rapidjson
    - QJsonDocument
- JSON Parser Benchmarking by Chad Austin (Jan 2013)
    - sajson
    - rapidjson
    - vjson
    - YAJL
    - Jansson

# 内部架构

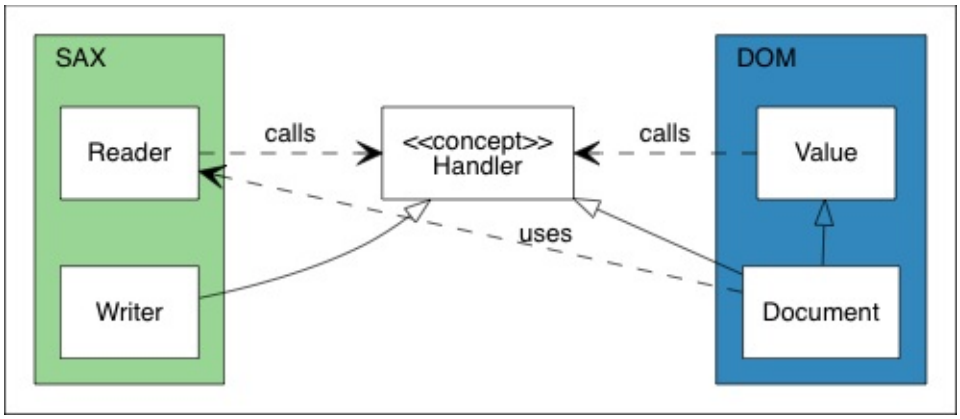
本部分记录了一些设计和实现细节。

[TOC]

# 架构

## SAX 和 DOM

下面的 UML 图显示了 SAX 和 DOM 的基本关系。

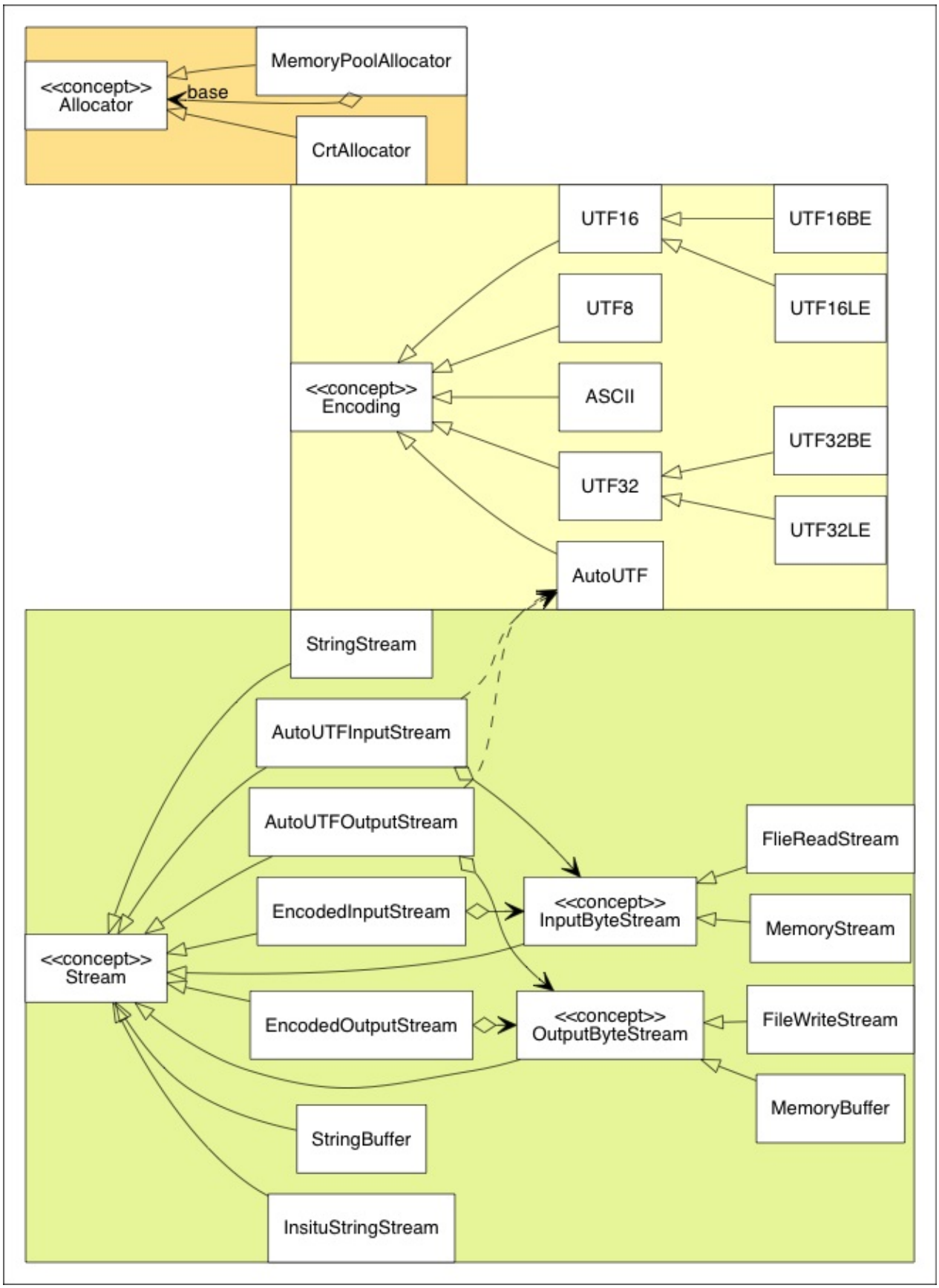



关系的核心是 `Handler` 概念。在 SAX 一边，`Reader` 从流解析 JSON 并将事件发送到 `Handler`。`Writer` 实现了 `Handler` 概念，用于处理相同的事件。在 DOM 一边，`Document` 实现了 `Handler` 概念，用于通过这些时间来构建 DOM。`Value` 支持了 `Value::Accept(Handler&)` 函数，它可以将 DOM 转换为事件进行发送。

在这个设计，SAX 是不依赖于 DOM 的。甚至 `Reader` 和 `Writer` 之间也没有依赖。这提供了连接事件发送器和处理器的灵活性。除此之外，`Value` 也是不依赖于 SAX 的。所以，除了将 DOM 序列化为 JSON 之外，用户也可以将其序列化为 XML，或者做任何其他事情。

# 工具类

SAX 和 DOM API 都依赖于3个额外的概念：`Allocator`、`Encoding` 和 `Stream`。它们的继承层次结构如下图所示。




# 值（**Value**）

`Value` （实际上被定义为 `GenericValue<UTF8<>>` ）是 DOM API 的核心。本部分描述了它的设计。

# 数据布局

`Value` 是可变类型。在 RapidJSON 的上下文中，一个 `Value` 的实例可以包含6种 JSON 数据类型之一。通过使用 `union` ，这是可能实现的。每一个 `Value` 包含两个成员： `union Data data_` 和 `unsigned flags_` 。 `flags_` 表明了 JSON 类型，以及附加的信息。

下表显示了所有类型的数据布局。32位/64位列表明了字段所占用的字节数。

| Null | | 32位 | 64位 |
|---|---|---|---|
| （未使用） | | 4 | 8 |
| （未使用） | | 4 | 4 |
| （未使用） | | 4 | 4 |
| `unsigned flags_` | `kNullType kNullFlag` | 4 | 4 |

| Bool | | 32位 | 64位 |
|---|---|---|---|
| （未使用） | | 4 | 8 |
| （未使用） | | 4 | 4 |
| （未使用） | | 4 | 4 |
| `unsigned flags_` | `kBoolType` (either `kTrueFlag` or `kFalseFlag` ) | 4 | 4 |

| String | | 32位 | 64位 |
|---|---|---|---|
| `Ch* str` | 指向字符串的指针（可能拥有所有权） | 4 | 8 |
| `SizeType length` | 字符串长度 | 4 | 4 |
| （未使用） | | 4 | 4 |
| `unsigned flags_` | `kStringType kStringFlag ...` | 4 | 4 |

| Object | | 32位 | 64位 |
|---|---|---|---|
| `Member* members` | 指向成员数组的指针（拥有所有权） | 4 | 8 |
| `SizeType size` | 成员数量 | 4 | 4 |
| `SizeType capacity` | 成员容量 | 4 | 4 |
| `unsigned flags_` | `kObjectType kObjectFlag` | 4 | 4 |

| Array | | 32位 | 64位 |
|---|---|---|---|
| `Value* values` | 指向值数组的指针（拥有所有权） | 4 | 8 |
| `SizeType size` | 值数量 | 4 | 4 |
| `SizeType capacity` | 值容量 | 4 | 4 |
| `unsigned flags_` | `kArrayType kArrayFlag` | 4 | 4 |

| Number (Int) | | 32位 | 64位 |
|---|---|---|---|
| `int i` | 32位有符号整数 | 4 | 4 |
| （零填充） | 0 | 4 | 4 |
| （未使用） | | 4 | 8 |
| `unsigned flags_` | `kNumberType kNumberFlag kIntFlag kInt64Flag ...` | 4 | 4 |

| Number (UInt) | | 32位 | 64位 |
|---|---|---|---|
| `unsigned u` | 32位无符号整数 | 4 | 4 |
| （零填充） | 0 | 4 | 4 |
| （未使用） | | 4 | 8 |
| `unsigned flags_` | `kNumberType kNumberFlag kUIntFlag kUInt64Flag ...` | 4 | 4 |

| Number (Int64) | | 32位 | 64位 |
|---|---|---|---|
| `int64_t i64` | 64位有符号整数 | 8 | 8 |
| （未使用） | | 4 | 8 |
| `unsigned flags_` | `kNumberType kNumberFlag kInt64Flag ...` | 4 | 4 |

| Number (Uint64) | | 32位 | 64位 |
|---|---|---|---|
| `uint64_t i64` | 64位无符号整数 | 8 | 8 |
| （未使用） | | 4 | 8 |
| `unsigned flags_` | `kNumberType kNumberFlag kInt64Flag ...` | 4 | 4 |

| Number (Double) | | 32位 | 64位 |
|---|---|---|---|
| `uint64_t i64` | 双精度浮点数 | 8 | 8 |
| （未使用） | | 4 | 8 |
| `unsigned flags_` | `kNumberType kNumberFlag kDoubleFlag` | 4 | 4 |

这里有一些需要注意的地方：

- 为了减少在64位架构上的内存消耗，`SizeType` 被定义为 `unsigned` 而不是 `size_t`。
- 32位整数的零填充可能被放在实际类型的前面或后面，这依赖于字节序。这使得它可以将32位整数不经过任何转换就可以解释为64位整数。
- `Int` 永远是 `Int64`，反之不然。

## 标志

32位的 `flags_` 包含了 JSON 类型和其他信息。如前文中的表所述，每一种 JSON 类型包含了冗余的 `kXXXType` 和 `kXXXFlag`。这个设计是为了优化测试位标志（`IsNumber()`）和获取每一种类型的序列号（`GetType()`）。

字符串有两个可选的标志。`kCopyFlag` 表明这个字符串拥有字符串拷贝的所有权。而 `kInlineStrFlag` 意味着使用了短字符串优化。

数字更加复杂一些。对于普通的整数值，它可以包含 `kIntFlag` 、 `kUintFlag` 、 `kInt64Flag` 和/或 `kUint64Flag` ，这由整数的范围决定。带有小数或者超过64位所能表达的范围的整数的数字会被存储为带有 `kDoubleFlag` 的 `double` 。

## 短字符串优化

Kosta 提供了很棒的短字符串优化。这个优化的xxx如下所述。除去 `flags_` ， `Value` 有12或16字节（对于32位或64位）来存储实际的数据。这为在其内部直接存储短字符串而不是存储字符串的指针创造了可能。对于1字节的字符类型（例如 `char` ），它可以在 `Value` 类型内部存储至多11或15个字符的字符串。

| ShortString (Ch=char) | | 32位 | 64位 |
|---|---|---|---|
| `Ch str[MaxChars]` | 字符串缓冲区 | 11 | 15 |
| `Ch invLength` | MaxChars - Length | 1 | 1 |
| `unsigned flags_` | `kStringType kStringFlag ...` | 4 | 4 |

这里使用了一项特殊的技术。它存储了 (MaxChars - length) 而不直接存储字符串的长度。这使得存储11个字符并且带有后缀 `\0` 成为可能。

这个优化可以减少字符串拷贝内存占用。它也改善了缓存一致性，并进一步提高了运行时性能。

# 分配器（Allocator）

`Allocator` 是 RapidJSON 中的概念：

```
concept Allocator {
    static const bool kNeedFree;    //!< 表明这个分配器是否需要调用 Free()。

    // 申请内存块。
    // \param size 内存块的大小，以字节记。
    // \returns 指向内存块的指针。
    void* Malloc(size_t size);

    // 调整内存块的大小。
    // \param originalPtr 当前内存块的指针。空指针是被允许的。
    // \param originalSize 当前大小，以字节记。（设计问题：因为有些分配器可能不会记录它，显示的传递它可以节约内存。）
    // \param newSize 新大小，以字节记。
    void* Realloc(void* originalPtr, size_t originalSize, size_t newSize);

    // 释放内存块。
    // \param ptr 指向内存块的指针。空指针是被允许的。
    static void Free(void *ptr);
};
```

需要注意的是 `Malloc()` 和 `Realloc()` 是成员函数而 `Free()` 是静态成员函数。

## MemoryPoolAllocator

`MemoryPoolAllocator` 是 DOM 的默认内存分配器。它只申请内存而不释放内存。这对于构建 DOM 树非常合适。

在它的内部，它从基础的内存分配器申请内存块（默认为 `CrtAllocator` ）并将这些内存块存储为单向链表。当用户请求申请内存，它会遵循下列步骤来申请内存：

1. 如果可用，使用用户提供的缓冲区。（见 User Buffer section in DOM）
2. 如果用户提供的缓冲区已满，使用当前内存块。
3. 如果当前内存块已满，申请新的内存块。

# 解析优化

## 使用 SIMD 跳过空格

当从流中解析 JSON 时，解析器需要跳过4种空格字符：

1. 空格 ( `U+0020` )
2. 制表符 ( `U+000B` )
3. 换行 ( `U+000A` )
4. 回车 ( `U+000D` )

这是一份简单的实现：

```
void SkipWhitespace(InputStream& s) {
    while (s.Peek() == ' ' || s.Peek() == '\n' || s.Peek() == '\r' || s.Peek() == '\t')
        s.Take();
}
```

但是，这需要对每个字符进行4次比较以及一些分支。这被发现是一个热点。

为了加速这一处理，RapidJSON 使用 SIMD 来在一次迭代中比较16个字符和4个空格。目前 RapidJSON 支持 SSE2 ， SSE4.2 和 ARM Neon 指令。同时它也只会对 UTF-8 内存流启用，包括字符串流或 原位 解析。

你可以通过在包含 `rapidjson.h` 之前定义 `RAPIDJSON_SSE2` ， `RAPIDJSON_SSE42` 或 `RAPIDJSON_NEON` 来启用这个优化。一些编译器可以检测这个设置，如 `perftest.h` ：

```
// __SSE2__ 和 __SSE4_2__ 可被 gcc、clang 和 Intel 编译器识别：
// 如果支持的话，我们在 gmake 中使用了 -march=native 来启用 -msse2 和 -msse4.2
// 同样的， __ARM_NEON 被用于识别Neon
#if defined(__SSE4_2__)
#  define RAPIDJSON_SSE42
#elif defined(__SSE2__)
#  define RAPIDJSON_SSE2
#elif defined(__ARM_NEON)
#  define RAPIDJSON_NEON
#endif
```

需要注意的是，这是编译期的设置。在不支持这些指令的机器上运行可执行文件会使它崩溃。

## 页面对齐问题

在 RapidJSON 的早期版本中，被报告了一个问题：`SkipWhitespace_SIMD()` 会罕见地导致崩溃（约五十万分之一的几率）。在调查之后，怀疑是 `_mm_loadu_si128()` 访问了 `'\0'` 之后的内存，并越过被保护的页面边界。

在 Intel® 64 and IA-32 Architectures Optimization Reference Manual 中，章节 10.2.1：

> 为了支持需要费对齐的128位 SIMD 内存访问的算法，调用者的内存缓冲区申请应当考虑添加一些填充空间，这样被调用的函数可以安全地将地址指针用于未对齐的128位 SIMD 内存操作。 在结合非对齐的 SIMD 内存操作中，最小的对齐大小应该等于 SIMD 寄存器的大小。

对于 RapidJSON 来说，这显然是不可行的，因为 RapidJSON 不应当强迫用户进行内存对齐。

为了修复这个问题，当前的代码会先按字节处理直到下一个对齐的地址。在这之后，使用对齐读取来进行 SIMD 处理。见 #85。

# 局部流拷贝

在优化的过程中，我们发现一些编译器不能将访问流的一些成员数据放入局部变量或者寄存器中。测试结果显示，对于一些流类型，创建流的拷贝并将其用于内层循环中可以改善性能。例如，实际（非 SIMD）的 `SkipWhitespace()` 被实现为：

```
template<typename InputStream>
void SkipWhitespace(InputStream& is) {
    internal::StreamLocalCopy<InputStream> copy(is);
    InputStream& s(copy.s);

    while (s.Peek() == ' ' || s.Peek() == '\n' || s.Peek() == '\r' || s.Peek() == '\t')
        s.Take();
}
```

基于流的特征，`StreamLocalCopy` 会创建（或不创建）流对象的拷贝，在局部使用它并将流的状态拷贝回原来的流。

### 解析为双精度浮点数

将字符串解析为 `double` 并不简单。标准库函数 `strtod()` 可以胜任这项工作，但它比较缓慢。默认情况下，解析器使用默认的精度设置。这最多有 3ULP 的误差，并实现在 `internal::StrtodNormalPrecision()` 中。

当使用 `kParseFullPrecisionFlag` 时，编译器会改为调用 `internal::StrtodFullPrecision()` ，这个函数会自动调用三个版本的转换。

1. Fast-Path。
2. double-conversion 中的自定义 DIY-FP 实现。
3. （Clinger, William D. How to read floating point numbers accurately. Vol. 25. No. 6. ACM, 1990） 中的大整数算法。

如果第一个转换方法失败，则尝试使用第二种方法，以此类推。

## 生成优化

### 整数到字符串的转换

整数到字符串转换的朴素算法需要对每一个十进制位进行一次处罚。我们实现了若干版本并在 itoa-benchmark 中对它们进行了评估。

虽然 SSE2 版本是最快的，但它和第二快的 `branchlut` 差距不大。而且 `branchlut` 是纯C++实现，所以我们在 RapidJSON 中使用了 `branchlut` 。

### 双精度浮点数到字符串的转换

原来 RapidJSON 使用 `snprintf(..., ..., "%g")` 来进行双精度浮点数到字符串的转换。这是不准确的，因为默认的精度是6。随后我们发现它很缓慢，而且有其它的替代品。

Google 的 V8 double-conversion 实现了更新的、快速的被称为 Grisu3 的算法（Loitsch, Florian. "Printing floating-point numbers quickly and accurately with integers." ACM Sigplan Notices 45.6 (2010): 233-243.）。

然而，这个实现不是仅头文件的，所以我们实现了一个仅头文件的 Grisu2 版本。这个算法保证了结果永远精确。而且在大多数情况下，它会生成最短的（可选）字符串表示。

这个仅头文件的转换函数在 dtoa-benchmark 中进行评估。

## 解析器

### 迭代解析

迭代解析器是一个以非递归方式实现的递归下降的 LL(1) 解析器。

### 语法

解析器使用的语法是基于严格 JSON 语法的：

```
S -> array | object
array -> [ values ]
object -> { members }
values -> non-empty-values | ε
non-empty-values -> value addition-values
addition-values -> ε | , non-empty-values
members -> non-empty-members | ε
non-empty-members -> member addition-members
addition-members -> ε | , non-empty-members
member -> STRING : value
value -> STRING | NUMBER | NULL | BOOLEAN | object | array
```

注意到左因子被加入了非终结符的 `values` 和 `members` 来保证语法是 LL(1) 的。

## 解析表

基于这份语法，我们可以构造 FIRST 和 FOLLOW 集合。

非终结符的 FIRST 集合如下所示：

| NON-TERMINAL | FIRST |
|---|---|
| array | [ |
| object | { |
| values | ε STRING NUMBER NULL BOOLEAN { [ |
| addition-values | ε COMMA |
| members | ε STRING |
| addition-members | ε COMMA |
| member | STRING |
| value | STRING NUMBER NULL BOOLEAN { [ |
| S | [ { |
| non-empty-members | STRING |
| non-empty-values | STRING NUMBER NULL BOOLEAN { [ |

FOLLOW 集合如下所示：

| NON-TERMINAL | FOLLOW |
|---|---|
| S | $ |
| array | , $ } ] |
| object | , $ } ] |
| values | ] |
| non-empty-values | ] |
| addition-values | ] |
| members | } |
| non-empty-members | } |
| addition-members | } |
| member | , } |
| value | , } ] |

最终可以从 FIRST 和 FOLLOW 集合生成解析表：

| NON-TERMINAL | [ | { | , | : | ] | } | STRING | NUMBER | N |
|---|---|---|---|---|---|---|---|---|---|
| S | array | object | | | | | | | |
| array | [ values ] | | | | | | | | |
| object | | { members } | | | | | | | |
| values | non-empty-values | non-empty-values | | | ε | | non-empty-values | non-empty-values | e v |
| non-empty-values | value addition-values | value addition-values | | | | | value addition-values | value addition-values | v ad v |
| addition-values | | | , non-empty-values | | ε | | | | |
| members | | | | | | ε | non-empty-members | | |
| non-empty-members | | | | | | | member addition-members | | |
| addition-members | | | , non-empty-members | | | ε | | | |
| member | | | | | | | STRING : value | | |
| value | array | object | | | | | STRING | NUMBER | N |

对于上面的语法分析，这里有一个很棒的工具。

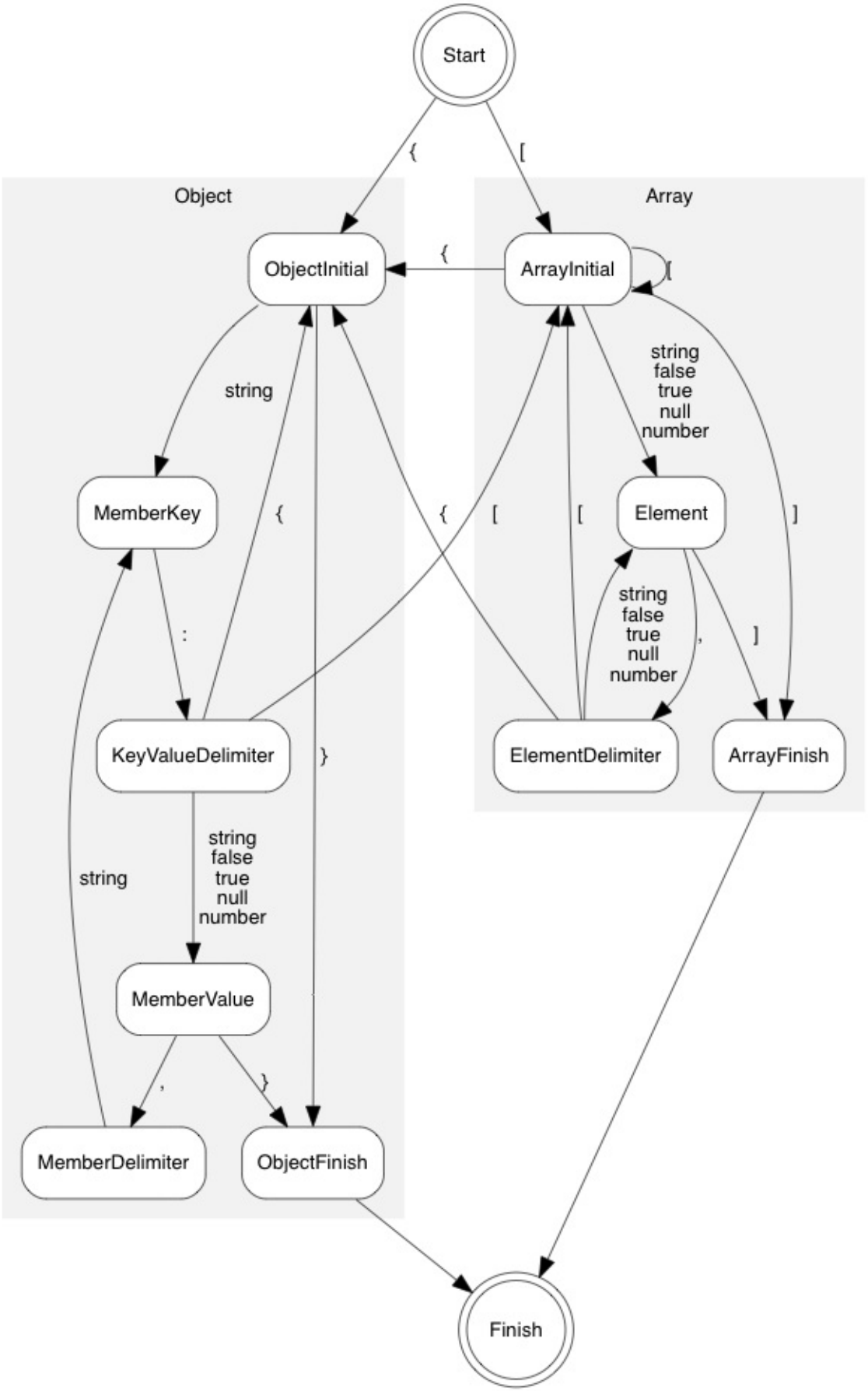
## 实现

基于这份解析表，一个直接的（常规的）将规则反向入栈的实现可以正常工作。

在 RapidJSON 中，对直接的实现进行了一些修改：

首先，在 RapidJSON 中，这份解析表被编码为状态机。 规则由头部和主体组成。 状态转换由规则构造。 除此之外，额外的状态被添加到与 `array` 和 `object` 有关的规则。 通过这种方式，生成数组值或对象成员可以只用一次状态转移便可完成， 而不需要在直接的实现中的多次出栈/入栈操作。 这也使得估计栈的大小更加容易。

状态图如如下所示：




第二，迭代解析器也在内部栈保存了数组的值个数和对象成员的数量，这也与传统的实现不同。

# 常见问题

[TOC]

## 一般问题

1. RapidJSON 是什么？

   RapidJSON 是一个 C++ 库，用于解析及生成 JSON。读者可参考它的所有 特点。

2. 为什么称作 RapidJSON？

   它的灵感来自于 RapidXML，RapidXML 是一个高速的 XML DOM 解析器。

3. RapidJSON 与 RapidXML 相似么？

   RapidJSON 借镜了 RapidXML 的一些设计，包括原位（*in situ*）解析、只有头文件的库。但两者的 API 是完全不同的。此外 RapidJSON 也提供许多 RapidXML 没有的特点。

4. RapidJSON 是免费的么？

   是的，它在 MIT 特許條款下免费。它可用于商业软件。详情请参看 license.txt。

5. RapidJSON 很小么？它有何依赖？

   是的。在 Windows 上，一个解析 JSON 并打印出统计的可执行文件少于 30KB。

   RapidJSON 仅依赖于 C++ 标准库。

6. 怎样安装 RapidJSON？

   见 安装一节。

7. RapidJSON 能否运行于我的平台？

   社区已在多个操作系统／编译器／CPU 架构的组合上测试 RapidJSON。但我们无法确保它能运行于你特定的平台上。只需要生成及执行单元测试便能获取答案。

8. RapidJSON 支持 C++03 么？C++11 呢？

   RapidJSON 开始时在 C++03 上实现。后来加入了可选的 C++11 特性支持（如转移构造函数、`noexcept`）。RapidJSON 应该兼容所有遵从 C++03 或 C++11 的编译器。

9. RapidJSON 是否真的用于实际应用？

   是的。它被配置于前台及后台的真实应用中。一个社区成员说 RapidJSON 在他们的系统中每日解析 5 千万个 JSON。

10. RapidJSON 是如何被测试的？

    RapidJSON 包含一组单元测试去执行自动测试。Travis（供 Linux 平台）及 AppVeyor（供 Windows 平台）会对所有修改进行编译及执行单元测试。在 Linux 下还会使用 Valgrind 去检测内存泄漏。

11. RapidJSON 是否有完整的文档？

RapidJSON 提供了使用手册及 API 说明文档。

12. 有没有其他替代品？

    有许多替代品。例如 nativejson-benchmark 列出了一些开源的 C/C++ JSON 库。json.org 也有一个列表。

# JSON

1. 什么是 JSON？

   JSON (JavaScript Object Notation) 是一个轻量的数据交换格式。它使用人类可读的文本格式。更多关于 JSON 的细节可考 RFC7159 及 ECMA-404。

2. JSON 有什么应用场合？

   JSON 常用于网页应用程序，以传送结构化数据。它也可作为文件格式用于数据持久化。

3. RapidJSON 是否符合 JSON 标准？

   是。RapidJSON 完全符合 RFC7159 及 ECMA-404。它能处理一些特殊情况，例如支持 JSON 字符串中含有空字符及代理对（surrogate pair）。

4. RapidJSON 是否支持宽松的语法？

   现时不支持。RapidJSON 只支持严格的标准格式。宽松语法现时在这 issue 中进行讨论。

# DOM 与 SAX

1. 什么是 DOM 风格 API？

   Document Object Model（DOM）是一个储存于内存的 JSON 表示方式，用于查询及修改 JSON。

2. 什么是 SAX 风格 API?

   SAX 是一个事件驱动的 API，用于解析及生成 JSON。

3. 我应用 DOM 还是 SAX？

   DOM 易于查询及修改。SAX 则是非常快及省内存的，但通常较难使用。

4. 什么是原位（*in situ*）解析？

   原位解析会把 JSON 字符串直接解码至输入的 JSON 中。这是一个优化，可减少内存消耗及提升性能，但输入的 JSON 会被更改。进一步细节请参考 原位解析 。

5. 什么时候会产生解析错误？

   当输入的 JSON 包含非法语法，或不能表示一个值（如 Number 太大），或解析器的处理器中断解析过程，解析器都会产生一个错误。详情请参考 解析错误。

6. 有什么错误信息？

   错误信息存储在 `ParseResult` ，它包含错误代号及偏移值（从 JSON 开始至错误处的字符数目）。可以把错误代号翻译为人类可读的错误讯息。

7. 为何不只使用 `double` 去表示 JSON number？

   一些应用需要使用 64 位无号／有号整数。这些整数不能无损地转换成 `double`。因此解析器会检测一个 JSON number 是否能转换至各种整数类型及 `double`。

8. 如何清空并最小化 `document` 或 `value` 的容量？

   调用 `SetXXX()` 方法 - 这些方法会调用析构函数，并重建空的 Object 或 Array:

   ```
   Document d;
   ...
   d.SetObject();  // clear and minimize
   ```

   另外，也可以参考在 C++ swap with temporary idiom 中的一种等价的方法:

   ```
   Value(kObjectType).Swap(d);
   ```

   或者，使用这个稍微长一点的代码也能完成同样的事情:

   ```
   d.Swap(Value(kObjectType).Move());
   ```

9. 如何将一个 `document` 节点插入到另一个 `document` 中？

   比如有以下两个 document(DOM):

   ```
   Document person;
   person.Parse("{\"person\":{\"name\":{\"first\":\"Adam\",\"last\":\"Thomas\"}}}");

   Document address;
   address.Parse("{\"address\":{\"city\":\"Moscow\",\"street\":\"Quiet\"}}");
   ```

   假设我们希望将整个 `address` 插入到 `person` 中，作为其的一个子节点:

   ```
   { "person": {
      "name": { "first": "Adam", "last": "Thomas" },
      "address": { "city": "Moscow", "street": "Quiet" }
      }
   }
   ```

   在插入节点的过程中需要注意 `document` 和 `value` 的生命周期并且正确地使用 allocator 进行内存分配和管理。

   一个简单有效的方法就是修改上述 `address` 变量的定义，让其使用 `person` 的 allocator 初始化，然后将其添加到根节点。

   ```
   Documnet address(person.GetAllocator());
   ...
   person["person"].AddMember("address", address["address"], person.GetAllocator());
   ```

当然，如果你不想通过显式地写出 `address` 的 key 来得到其值，可以使用迭代器来实现:

```
auto addressRoot = address.MemberBegin();
person["person"].AddMember(addressRoot->name, addressRoot->value, person.GetAllocat
or());
```

此外，还可以通过深拷贝 address document 来实现:

```
Value addressValue = Value(address["address"], person.GetAllocator());
person["person"].AddMember("address", addressValue, person.GetAllocator());
```

# Document/Value (DOM)

1. 什么是转移语义？为什么？

   `Value` 不用复制语义，而使用了转移语义。这是指，当把来源值赋值于目标值时，来源值的所有权会转移至目标值。

   由于转移快于复制，此设计决定强迫使用者注意到复制的消耗。

2. 怎样去复制一个值？

   有两个 API 可用：含 allocator 的构造函数，以及 `CopyFrom()`。可参考 深复制 Value 里的用例。

3. 为什么我需要提供字符串的长度？

   由于 C 字符串是空字符结尾的，需要使用 `strlen()` 去计算其长度，这是线性复杂度的操作。若使用者已知字符串的长度，对很多操作来说会造成不必要的消耗。

   此外，RapidJSON 可处理含有 `\u0000`（空字符）的字符串。若一个字符串含有空字符，`strlen()` 便不能返回真正的字符串长度。在这种情况下使用者必须明确地提供字符串长度。

4. 为什么在许多 DOM 操作 API 中要提供分配器作为参数？

   由于这些 API 是 `Value` 的成员函数，我们不希望为每个 `Value` 储存一个分配器指针。

5. 它会转换各种数值类型么？

   当使用 `GetInt()`、`GetUint()` 等 API 时，可能会发生转换。对于整数至整数转换，仅当保证转换安全才会转换（否则会断言失败）。然而，当把一个 64 位有号／无号整数转换至 double 时，它会转换，但有可能会损失精度。含有小数的数字、或大于 64 位的整数，都只能使用 `GetDouble()` 获取其值。

# Reader/Writer (SAX)

1. 为什么不仅仅用 `printf` 输出一个 JSON？为什么需要 `Writer`？

   最重要的是，`Writer` 能确保输出的 JSON 是格式正确的。错误地调用 SAX 事件（如 `StartObject()` 错配 `EndArray()`）会造成断言失败。此外，`Writer` 会把字符串进行转义（如 `\n`）。最后，`printf()` 的数值输出可能并不是一个合法的 JSON number，特别是某些 locale 会

有数字分隔符。而且 `Writer` 的数值字符串转换是使用非常快的算法来实现的，胜过 `printf()` 及 `iostream` 。

2. 我能否暂停解析过程，并在稍后继续？

   基于性能考虑，目前版本并不直接支持此功能。然而，若执行环境支持多线程，使用者可以在另一线程解析 JSON，并通过阻塞输入流去暂停。

## Unicode

1. 它是否支持 UTF-8、UTF-16 及其他格式？

   是。它完全支持 UTF-8、UTF-16（大端／小端）、UTF-32（大端／小端）及 ASCII。

2. 它能否检测编码的合法性？

   能。只需把 `kParseValidateEncodingFlag` 参考传给 `Parse()` 。若发现在输入流中有非法的编码，它就会产生 `kParseErrorStringInvalidEncoding` 错误。

3. 什么是代理对（surrogate pair)？RapidJSON 是否支持？

   JSON 使用 UTF-16 编码去转义 Unicode 字符，例如 `\u5927` 表示中文字“大”。要处理基本多文种平面（basic multilingual plane，BMP）以外的字符时，UTF-16 会把那些字符编码成两个 16 位值，这称为 UTF-16 代理对。例如，绘文字字符 U+1F602 在 JSON 中可被编码成 `\uD83D\uDE02` 。

   RapidJSON 完全支持解析及生成 UTF-16 代理对。

4. 它能否处理 JSON 字符串中的 `\u0000` （空字符）？

   能。RapidJSON 完全支持 JSON 字符串中的空字符。然而，使用者需要注意到这件事，并使用 `GetStringLength()` 及相关 API 去取得字符串真正长度。

5. 能否对所有非 ASCII 字符输出成 `\uxxxx` 形式？

   可以。只要在 `Writer` 中使用 `ASCII<>` 作为输出编码参数，就可以强逼转义那些字符。

## 流

1. 我有一个很大的 JSON 文件。我应否把它整个载入内存中？

   使用者可使用 `FileReadStream` 去逐块读入文件。但若使用于原位解析，必须载入整个文件。

2. 我能否解析一个从网络上串流进来的 JSON？

   可以。使用者可根据 `FileReadStream` 的实现，去实现一个自定义的流。

3. 我不知道一些 JSON 将会使用哪种编码。怎样处理它们？

   你可以使用 `AutoUTFInputStream` ，它能自动检测输入流的编码。然而，它会带来一些性能开销。

4. 什么是 BOM？RapidJSON 怎样处理它？

   字节顺序标记（byte order mark, BOM） 有时会出现于文件／流的开始，以表示其 UTF 编码类型。

RapidJSON 的 `EncodedInputStream` 可检测／跳过 BOM。`EncodedOutputStream` 可选择是否写入 BOM。可参考 编码流 中的例子。

5. 为什么会涉及大端／小端？

   流的大端／小端是 UTF-16 及 UTF-32 流要处理的问题，而 UTF-8 不需要处理。

## 性能

1. RapidJSON 是否真的快？

   是。它可能是最快的开源 JSON 库。有一个 评测 评估 C/C++ JSON 库的性能。

2. 为什么它会快？

   RapidJSON 的许多设计是针对时间／空间性能来设计的，这些决定可能会影响 API 的易用性。此外，它也使用了许多底层优化（内部函数／intrinsic、SIMD）及特别的算法（自定义的 double 至字符串转换、字符串至 double 的转换）。

3. 什是是 SIMD？它如何用于 RapidJSON？

   SIMD 指令可以在现代 CPU 中执行并行运算。RapidJSON 支持使用 Intel 的 SSE2/SSE4.2 和 ARM 的 Neon 来加速对空白符、制表符、回车符和换行符的过滤处理。在解析含缩进的 JSON 时，这能提升性能。只要定义名为 `RAPIDJSON_SSE2` ，`RAPIDJSON_SSE42` 或 `RAPIDJSON_NEON` 的宏，就能启动这个功能。然而，若在不支持这些指令集的机器上执行这些可执行文件，会导致崩溃。

4. 它会消耗许多内存么？

   RapidJSON 的设计目标是减低内存占用。

   在 SAX API 中，`Reader` 消耗的内存与 JSON 树深度加上最长 JSON 字符成正比。

   在 DOM API 中，每个 `Value` 在 32/64 位架构下分别消耗 16/24 字节。RapidJSON 也使用一个特殊的内存分配器去减少分配的额外开销。

5. 高性能的意义何在？

   有些应用程序需要处理非常大的 JSON 文件。而有些后台应用程序需要处理大量的 JSON。达到高性能同时改善延时及吞吐量。更广义来说，这也可以节省能源。

## 八挂

1. 谁是 RapidJSON 的开发者？

   叶劲峰（Milo Yip，miloyip）是 RapidJSON 的原作者。全世界许多贡献者一直在改善 RapidJSON。Philipp A. Hartmann（pah）实现了许多改进，也设置了自动化测试，而且还参与许多社区讨论。丁欧南（Don Ding，thebusytypist）实现了迭代式解析器。Andrii Senkovych（jollyroger）完成了向 CMake 的迁移。Kosta（Kosta-Github）提供了一个非常灵巧的短字符串优化。也需要感谢其他献者及社区成员。

2. 为何你要开发 RapidJSON？

在 2011 年开始这项目是，它仅一个兴趣项目。Milo Yip 是一个游戏程序员，他在那时候认识到 JSON 并希望在未来的项目中使用。由于 JSON 好像很简单，他希望写一个仅有头文件并且快速的程序库。

3. 为什么开发中段有一段长期空档？

   主要是个人因素，例如加入新家庭成员。另外，Milo Yip 也花了许多业馀时间去翻译 Jason Gregory 的《Game Engine Architecture》至中文版《游戏引擎架构》。

4. 为什么这个项目从 Google Code 搬到 GitHub？

   这是大势所趋，而且 GitHub 更为强大及方便。