

目 录

致谢

介绍

从零开始的 JSON 库教程（一）：启程

 从零开始的 JSON 库教程（一）：启程解答篇

从零开始的 JSON 库教程（二）：解析数字

 从零开始的 JSON 库教程（二）：解析数字解答篇

从零开始的 JSON 库教程（三）：解析字符串

 从零开始的 JSON 库教程（三）：解析字符串解答篇

从零开始的 JSON 库教程（四）：Unicode

 从零开始的 JSON 库教程（四）：Unicode 解答篇

从零开始的 JSON 库教程（五）：解析数组

 从零开始的 JSON 库教程（五）：解析数组解答篇

从零开始的 JSON 库教程（六）：解析对象

 从零开始的 JSON 库教程（六）：解析对象解答篇

从零开始的 JSON 库教程（七）：生成器

 从零开始的 JSON 库教程（七）：生成器解答篇

致谢

当前文档《从零开始的JSON库教程》由进击的皇虫使用书栈(BookStack.CN)进行构建,生成于2018-02-26。

书栈(BookStack.CN)仅提供文档编写、整理、归类等功能,以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理,书栈(BookStack.CN)难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候,发现文档内容有不恰当的地方,请向我们反馈,让我们共同携手,将知识准确、高效且有效地传递给每一个人。

同时,如果您在日常生活、工作和学习中遇到有价值有营养的知识文档,欢迎分享到书栈(BookStack.CN),为知识的传承献上您的一份力量!

如果当前文档生成时间太久,请到书栈(BookStack.CN)获取最新的文档,以跟上知识更新换代的步伐。

文档地址: <http://www.bookstack.cn/books/json-tutorial>

书栈官网: <http://www.bookstack.cn>

书栈开源: <https://github.com/TruthHun>

分享,让知识传承更久远!感谢知识的创造者,感谢知识的分享者,也感谢每一位阅读到此处的读者,因为我们都将成为知识的传承者。

介绍

- [从零开始的 JSON 库教程](#)
 - [对象与目标](#)
 - [教程大纲](#)
 - [关于作者](#)

从零开始的 JSON 库教程

- Milo Yip
- 2016/9/15

也许有很多同学上过 C/C++ 的课后，可以完成一些简单的编程练习，又能在一些网站刷题，但对于如何开发有实际用途的程序可能感到束手无策。本教程希望能以一个简单的项目开发形式，让同学能逐步理解如何从无到有去开发软件。

为什么选择 JSON？因为它足够简单，除基本编程外不需大量技术背景知识。JSON 有标准，可按照标准逐步实现。JSON 也是实际在许多应用上会使用的格式，所以才会有大量的开源库。

这是一个免费、开源的教程，如果你喜欢，也可以打赏鼓励。因为工作及家庭因素，不能保证每篇文章的首发时间，请各为见谅。

对象与目标

教程对象：学习过基本 C/C++ 编程的同学。

通过这个教程，同学可以了解如何从零开始写一个 JSON 库，其特性如下：

- 符合标准的 JSON 解析器和生成器

- 手写的递归下降解析器 (recursive descent parser)
- 使用标准 C 语言 (C89)
- 跨平台 / 编译器 (如 Windows / Linux / OS X, vc / gcc / clang)
- 仅支持 UTF-8 JSON 文本
- 仅支持以 `double` 存储 JSON number 类型
- 解析器和生成器的代码合共少于 500 行

除了围绕 JSON 作为例子, 希望能在教程中讲述一些课题:

- 测试驱动开发 (test driven development, TDD)
- C 语言编程风格
- 数据结构
- API 设计
- 断言
- Unicode
- 浮点数
- Github、CMake、valgrind、Doxygen 等工具

教程大纲

本教程预计分为 9 个单元, 第 1-8 个单元附带练习和解答。

1. [启程](#) (2016/9/15 完成): 编译环境、JSON 简介、测试驱动、解析器主要函数及各数据结构。练习 JSON 布尔类型的解析。 [启程解答篇](#) (2016/9/17 完成)。
2. [解析数字](#) (2016/9/18 完成): JSON number 的语法。练习 JSON number 类型的校验。 [解析数字解答篇](#) (2016/9/20 完成)。
3. [解析字符串](#) (2016/9/22 完成): 使用 union 存储

- variant、自动扩展的堆栈、JSON string 的语法、valgrind。练习最基本的 JSON string 类型的解析、内存释放。[解析字符串解答篇](#)（2016/9/27 完成）。
4. [Unicode](#)（2016/10/2 完成）：Unicode 和 UTF-8 的基本知识、JSON string 的 unicode 处理。练习完成 JSON string 类型的解析。[Unicode 解答篇](#)（2016/10/6 完成）。
 5. [解析数组](#)（2016/10/7 完成）：JSON array 的语法。练习完成 JSON array 类型的解析、相关内存释放。[解析数组解答篇](#)（2016/10/13 完成）。
 6. [解析对象](#)（2016/10/29 完成）：JSON object 的语法、重构 string 解析函数。练习完成 JSON object 的解析、相关内存释放。[解析对象解答篇](#)（2016/11/15 完成）。
 7. [生成器](#)（2016/12/20 完成）：JSON 生成过程、注意事项。练习完成 JSON 生成器。[生成器解答篇](#)（2017/1/5 完成）
 8. 访问：JSON array/object 的访问及修改。练习完成相关功能。
 9. 终点及新开始：加入 nativejson-benchmark 测试，与 RapidJSON 对比及展望。

关于作者

叶劲峰 (Milo Yip) 现任腾讯 T4 专家、互动娱乐事业群魔方工作室群前台技术总监。他获得香港大学认知科学学士 (BCogSc)、香港中文大学系统工程及工程管理哲学硕士 (MPhil)。他是《游戏引擎架构》译者、《C++ Primer 中文版 (第五版)》审校。他曾参与《天涯明月刀》、《斗战神》、《爱丽丝：疯狂回归》、《美食从天降》、《王子传奇》等游戏项目，以及多个游戏引擎及中间件的研发。他是开源项目 [RapidJSON](#) 的作者，开发 [nativejson-benchmark](#) 比较

41 个开源原生 JSON 库的标准符合程度及性能。他在 1990 年学习 C 语言，1995 年开始使用 C++ 于各种项目。

从零开始的 JSON 库教程（一）：启程

- 从零开始的 JSON 库教程（一）：启程
 - JSON 是什么
 - 搭建编译环境
 - 头文件与 API 设计
 - JSON 语法子集
 - 单元测试
 - 宏的编写技巧
 - 实现解析器
 - 关于断言
 - 总结与练习
 - 常见问题

从零开始的 JSON 库教程（一）：启程

- Milo Yip
- 2016/9/15

本文是《从零开始的 JSON 库教程》的第一个单元。教程练习源代码位于 [json-tutorial](#)。

本单元内容：

1. [JSON 是什么](#)
2. [搭建编译环境](#)
3. [头文件与 API 设计](#)
4. [JSON 语法子集](#)
5. [单元测试](#)
6. [宏的编写技巧](#)

- 7. [实现解析器](#)
- 8. [关于断言](#)
- 9. [总结与练习](#)
- 0. [常见问题](#)

JSON 是什么

JSON (JavaScript Object Notation) 是一个用于数据交换的文本格式，现时的标准为[ECMA-404](#)。

虽然 JSON 源至于 JavaScript 语言，但它只是一种数据格式，可用于任何编程语言。现时具类似功能的格式有 XML、YAML，当中以 JSON 的语法最为简单。

例如，一个动态网页想从服务器获得数据时，服务器从数据库查找数据，然后把数据转换成 JSON 文本格式：

```
1. {
2.     "title": "Design Patterns",
3.     "subtitle": "Elements of Reusable Object-Oriented Software",
4.     "author": [
5.         "Erich Gamma",
6.         "Richard Helm",
7.         "Ralph Johnson",
8.         "John Vlissides"
9.     ],
10.    "year": 2009,
11.    "weight": 1.8,
12.    "hardcover": true,
13.    "publisher": {
14.        "Company": "Pearson Education",
15.        "Country": "India"
16.    },
17.    "website": null
18. }
```

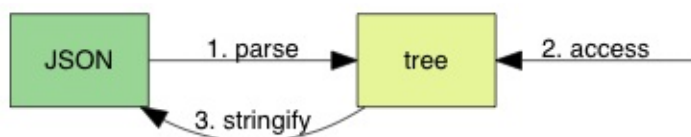
网页的脚本代码就可以把此 JSON 文本解析为内部的数据结构去使用。

从此例子可看出，JSON 是树状结构，而 JSON 只包含 6 种数据类型：

- null：表示为 null
- boolean：表示为 true 或 false
- number：一般的浮点数表示方式，在下一单元详细说明
- string：表示为 “...”
- array：表示为 [...]
- object：表示为 { ... }

我们要实现的 JSON 库，主要是完成 3 个需求：

1. 把 JSON 文本解析为一个树状数据结构（parse）。
2. 提供接口访问该数据结构（access）。
3. 把数据结构转换成 JSON 文本（stringify）。



我们会逐步实现这些需求。在本单元中，我们只实现最简单的 null 和 boolean 解析。

搭建编译环境

我们要做的库是跨平台、跨编译器的，同学可使用任意平台进行练习。

练习源代码位于 [json-tutorial](#)，当中 tutorial01 为本单元的练习代码。建议同学登记为 GitHub 用户，把项目 fork 一个自己

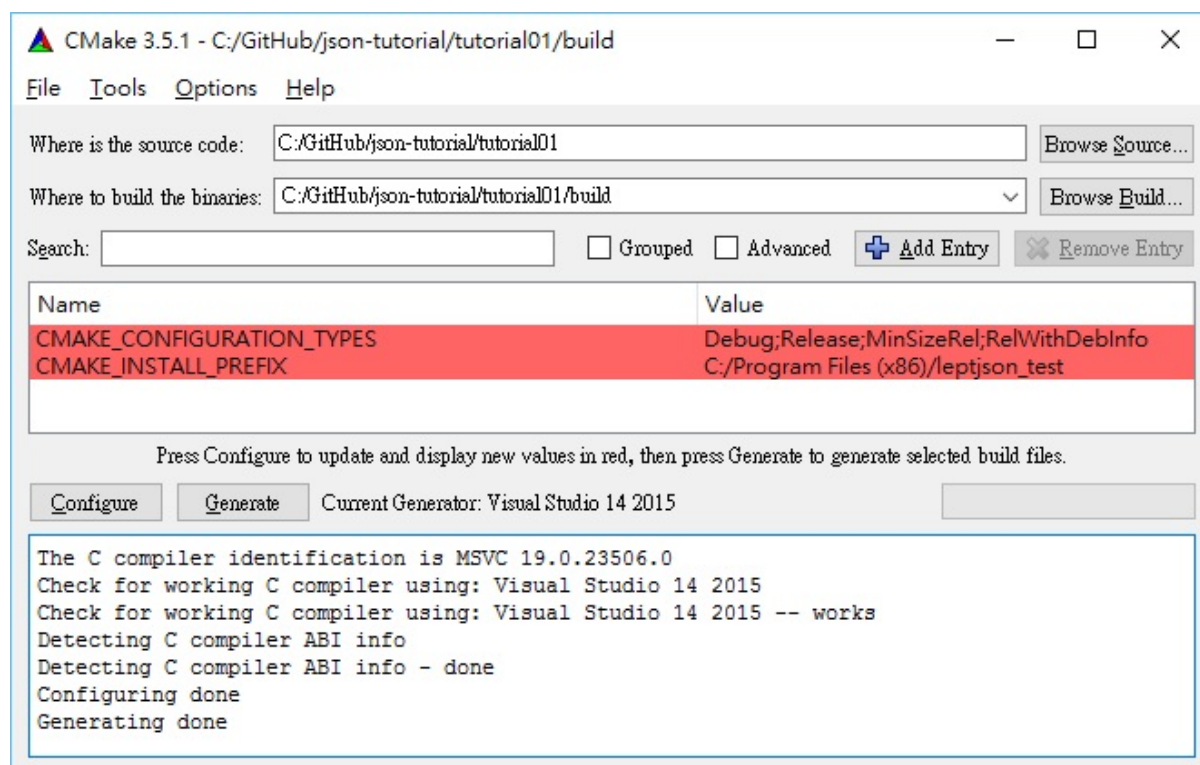
的版本，然后在上面进行修改。不了解版本管理的同学，也可以按右方「Clone or download」按钮，简单下载一个 zip 文件。

我们的 JSON 库名为 leptjson，代码文件只有 3 个：

1. `leptjson.h`：leptjson 的头文件 (header file)，含有对外的类型和 API 函数声明。
2. `leptjson.c`：leptjson 的实现文件 (implementation file)，含有内部的类型声明和函数实现。此文件会编译成库。
3. `test.c`：我们使用测试驱动开发 (test driven development, TDD)。此文件包含测试程序，需要链接 leptjson 库。

为了方便跨平台开发，我们会使用一个现时最流行的软件配置工具 [CMake](#)。

在 Windows 下，下载安装 CMake 后，可以使用其 cmake-gui 程序：



先在 “Where is the source code” 选择 json-tutorial/tutorial01, 再在 “Where to build the binary” 键入上一个目录加上 /build。

按 Configure, 选择编译器, 然后按 Generate 便会生成 Visual Studio 的 .sln 和 .vcproj 等文件。注意这个 build 目录都是生成的文件, 可以随时删除, 也不用上传至仓库。

在 OS X 下, 建议安装 [Homebrew](#), 然后在命令行键入:

```
1. $ brew install cmake
2. $ cd github/json-tutorial/tutorial01
3. $ mkdir build
4. $ cd build
5. $ cmake -DCMAKE_BUILD_TYPE=Debug ..
6. $ make
```

这样会使用 GNU make 来生成项目, 把 Debug 改成 Release 就会生成 Release 配置的 makefile。

若你喜欢的话, CMake 也可以生成 Xcode 项目:

```
1. $ cmake -G Xcode ..
2. $ open leptjson_test.xcodeproj
```

而在 Ubuntu 下, 可使用 `apt-get` 来安装:

```
1. $ apt-get install cmake
```

无论使用什么平台及编译环境, 编译运行后会出现:

```
1. $ ./leptjson_test
2. /Users/miloyip/github/json-tutorial/tutorial01/test.c:56: expect: 3
   actual: 0
3. 11/12 (91.67%) passed
```

若看到类似以上的结果，说明已成功搭建编译环境，我们可以去看看那几个代码文件的内容了。

头文件与 API 设计

C 语言有头文件的概念，需要使用 `#include` 去引入头文件中的类型声明和函数声明。但由于头文件也可以 `#include` 其他头文件，为避免重复声明，通常会利用宏加入 `include` 防范 (`include guard`)：

```
1. #ifndef LEPTJSON_H__
2. #define LEPTJSON_H__
3.
4. /* ... */
5.
6. #endif /* LEPTJSON_H__ */
```

宏的名字必须是唯一的，通常习惯以 `_H__` 作为后缀。由于 `leptjson` 只有一个头文件，可以简单命名为 `LEPTJSON_H__`。如果项目有多个文件或目录结构，可以用 `项目名称_目录_文件名称_H__` 这种命名方式。

如前所述，JSON 中有 6 种数据类型，如果把 `true` 和 `false` 当作两个类型就是 7 种，我们为此声明一个枚举类型 (`enumeration type`)：

```
1. typedef enum { LEPT_NULL, LEPT_FALSE, LEPT_TRUE, LEPT_NUMBER,
    LEPT_STRING, LEPT_ARRAY, LEPT_OBJECT } lept_type;
```

因为 C 语言没有 C++ 的命名空间 (`namespace`) 功能，一般会使用项目的简写作为标识符的前缀。通常枚举值用全大写（如

`LEPT_NULL`)，而类型及函数则用小写（如 `lept_type` ）。

接下来，我们声明 JSON 的数据结构。JSON 是一个树形结构，我们最终需要实现一个树的数据结构，每个节点使用 `lept_value` 结构体表示，我们会称它为一个 JSON 值 (JSON value)。

在此单元中，我们只需要实现 `null`, `true` 和 `false` 的解析，因此该结构体只需要存储一个 `lept_type`。之后的单元会逐步加入其他数据。

```
1. typedef struct {
2.     lept_type type;
3. }lept_value;
```

C 语言的结构体是以 `struct X {}` 形式声明的，定义变量时也要写成 `struct X x;`。为方便使用，上面的代码使用了 `typedef`。

然后，我们现在只需要两个 API 函数，一个是解析 JSON：

```
1. int lept_parse(lept_value* v, const char* json);
```

传入的 JSON 文本是一个 C 字符串（空结尾字符串 / null-terminated string），由于我们不应该改动这个输入字符串，所以使用 `const char*` 类型。

另一注意点是，传入的根节点指针 `v` 是由使用方负责分配的，所以一般用法是：

```
1. lept_value v;
2. const char json[] = ...;
3. int ret = lept_parse(&v, json);
```

返回值是以下这些枚举值，无错误会返回 `LEPT_PARSE_OK`，其他值在下节解释。

```

1. enum {
2.     LEPT_PARSE_OK = 0,
3.     LEPT_PARSE_EXPECT_VALUE,
4.     LEPT_PARSE_INVALID_VALUE,
5.     LEPT_PARSE_ROOT_NOT_SINGULAR
6. };

```

现时我们只需要一个访问结果的函数，就是获取其类型：

```

1. lept_type lept_get_type(const lept_value* v);

```

JSON 语法子集

下面是此单元的 JSON 语法子集，使用 [RFC7159](#) 中的 [ABNF](#) 表示：

```

1. JSON-text = ws value ws
2. ws = *(%x20 / %x09 / %x0A / %x0D)
3. value = null / false / true
4. null = "null"
5. false = "false"
6. true = "true"

```

当中 `%xhh` 表示以 16 进制表示的字符，`/` 是单选一，`*` 是零或多个，`()` 用于分组。

那么第一行的意思是，JSON 文本由 3 部分组成，首先是空白 (whitespace)，接着是一个值，最后是空白。

第二行告诉我们，所谓空白，是由零或多个空格符 (space U+0020)、制表符 (tab U+0009)、换行符 (LF U+000A)、回车符 (CR U+000D) 所组成。

第三行是说，我们现时的值只可以是 `null`、`false` 或 `true`，它们

分别有对应的字面值（literal）。

我们的解析器应能判断输入是否一个合法的JSON。如果输入的JSON不符合这个语法，我们要产生对应的错误码，方便使用者追查问题。

在这个JSON语法子集下，我们定义3种错误码：

- 若一个JSON只含有空白，传回 `LEPT_PARSE_EXPECT_VALUE`。
- 若一个值之后，在空白之后还有其他字符，传回 `LEPT_PARSE_ROOT_NOT_SINGULAR`。
- 若值不是那三种字面值，传回 `LEPT_PARSE_INVALID_VALUE`。

单元测试

许多同学在做练习题时，都是以 `printf` / `cout` 打印结果，再用肉眼对比结果是否合乎预期。但当软件项目越来越复杂，这个做法会越来越低效。一般我们会采用自动的测试方式，例如单元测试（unit testing）。单元测试也能确保其他人修改代码后，原来的功能维持正确（这称为回归测试 / regression testing）。

常用的单元测试框架有 xUnit 系列，如 C++ 的 [Google Test](#)、C# 的 [NUnit](#)。我们为了简单起见，会编写一个极简单的单元测试方式。

一般来说，软件开发是以周期进行的。例如，加入一个功能，再写关于该功能的单元测试。但也有另一种软件开发方法论，称为测试驱动开发（test-driven development, TDD），它的主要循环步骤是：

1. 加入一个测试。
2. 运行所有测试，新的测试应该会失败。
3. 编写实现代码。
4. 运行所有测试，若有测试失败回到3。

5. 重构代码。

6. 回到 1。

TDD 是先写测试，再实现功能。好处是实现只会刚好满足测试，而不会写了一些不需要的代码，或是没有被测试的代码。

但无论我们是采用 TDD，或是先实现后测试，都应尽量加入足够覆盖率的单元测试。

回到 leptjson 项目，`test.c` 包含了一个极简的单元测试框架：

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #include "leptjson.h"
5.
6. static int main_ret = 0;
7. static int test_count = 0;
8. static int test_pass = 0;
9.
10. #define EXPECT_EQ_BASE(equality, expect, actual, format) \
11.     do {\
12.         test_count++;\
13.         if (equality)\
14.             test_pass++;\
15.         else {\
16.             fprintf(stderr, "%s:%d: expect: " format " actual: "
format "\n", __FILE__, __LINE__, expect, actual);\
17.             main_ret = 1;\
18.         }\
19.     } while(0)
20.
21. #define EXPECT_EQ_INT(expect, actual) EXPECT_EQ_BASE((expect) ==
(actual), expect, actual, "%d")
22.
23. static void test_parse_null() {
24.     lept_value v;

```

```

25.     v.type = LEPT_TRUE;
26.     EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, "null"));
27.     EXPECT_EQ_INT(LEPT_NULL, lept_get_type(&v));
28. }
29.
30. /* ... */
31.
32. static void test_parse() {
33.     test_parse_null();
34.     /* ... */
35. }
36.
37. int main() {
38.     test_parse();
39.     printf("%d/%d (%3.2f%%) passed\n", test_pass, test_count,
40.           test_pass * 100.0 / test_count);
41.     return main_ret;
42. }

```

现时只提供了一个 `EXPECT_EQ_INT(expect, actual)` 的宏，每次使用这个宏时，如果 `expect != actual`（预期值不等于实际值），便会输出错误信息。

若按照 TDD 的步骤，我们先写一个测试，如上面的

`test_parse_null()`，而 `lept_parse()` 只返回 `LEPT_PARSE_OK`：

```

1. /Users/miloyip/github/json-tutorial/tutorial01/test.c:27: expect: 0
   actual: 1
2. 1/2 (50.00%) passed

```

第一个返回 `LEPT_PARSE_OK`，所以是通过的。第二个测试因为

`lept_parse()` 没有把 `v.type` 改成 `LEPT_NULL`，造成失败。我们再实现 `lept_parse()` 令到它能通过测试。

然而，完全按照 TDD 的步骤来开发，是会减慢开发进程。所以我个人会在这两种极端的工作方式取平衡。通常会在设计 API 后，先写部分

测试代码，再写满足那些测试的实现。

宏的编写技巧

有些同学可能不了解 `EXPECT_EQ_BASE` 宏的编写技巧，简单说明一下。反斜线代表该行未结束，会串接下一行。而如果宏里有多过一个语句（statement），就需要用 `do { /*...*/ } while(0)` 包裹成单个语句，否则会有如下的问题：

```
1. #define M() a(); b()
2. if (cond)
3.     M();
4. else
5.     c();
6.
7. /* 预处理后 */
8.
9. if (cond)
10.    a(); b(); /* b(); 在 if 之外 */
11. else
12.    c(); /* <- else 缺乏对应 if */
```

只用 `{ }` 也不行：

```
1. #define M() { a(); b(); }
2.
3. /* 预处理后 */
4.
5. if (cond)
6.    { a(); b(); }; /* 最后的分号代表 if 语句结束 */
7. else
8.    c(); /* else 缺乏对应 if */
```

用 `do while` 就行了：

```

1. #define M() do { a(); b(); } while(0)
2.
3. /* 预处理后 */
4.
5. if (cond)
6.     do { a(); b(); } while(0);
7. else
8.     c();

```

实现解析器

有了 API 的设计、单元测试，终于要实现解析器了。

首先为了减少解析函数之间传递多个参数，我们把这些数据都放进一个

`lept_context` 结构体：

```

1. typedef struct {
2.     const char* json;
3. }lept_context;
4.
5. /* ... */
6.
7. /* 提示：这里应该是 JSON-text = ws value ws */
8. /* 以下实现没处理最后的 ws 和 LEPT_PARSE_ROOT_NOT_SINGULAR */
9. int lept_parse(lept_value* v, const char* json) {
10.     lept_context c;
11.     assert(v != NULL);
12.     c.json = json;
13.     v->type = LEPT_NULL;
14.     lept_parse_whitespace(&c);
15.     return lept_parse_value(&c, v);
16. }

```

暂时我们只储存 `json` 字符串当前位置，之后的单元我们需要加入更多内容。

若 `lept_parse()` 失败，会把 `v` 设为 `null` 类型，所以这里先把它设为 `null`，让 `lept_parse_value()` 写入解析出来的根值。

`leptjson` 是一个手写的递归下降解析器（recursive descent parser）。由于 JSON 语法特别简单，我们不需要写分词器（tokenizer），只需检测下一个字符，便可以知道它是哪种类型的值，然后调用相关的分析函数。对于完整的 JSON 语法，跳过空白后，只需检测当前字符：

- `n` → `null`
- `t` → `true`
- `f` → `false`
- `"` → `string`
- `0-9/-` → `number`
- `[` → `array`
- `{` → `object`

所以，我们可以按照 JSON 语法一节的 EBNF 简单翻译成解析函数：

```

1. #define EXPECT(c, ch) do { assert(*c->json == (ch)); c->json++; }
   while(0)
2.
3. /* ws = *(%x20 / %x09 / %x0A / %x0D) */
4. static void lept_parse_whitespace(lept_context* c) {
5.     const char *p = c->json;
6.     while (*p == ' ' || *p == '\t' || *p == '\n' || *p == '\r')
7.         p++;
8.     c->json = p;
9. }
10.
11. /* null = "null" */
12. static int lept_parse_null(lept_context* c, lept_value* v) {
13.     EXPECT(c, 'n');
14.     if (c->json[0] != 'u' || c->json[1] != 'l' || c->json[2] !=

```

```

    'l')
15.         return LEPT_PARSE_INVALID_VALUE;
16.     c->json += 3;
17.     v->type = LEPT_NULL;
18.     return LEPT_PARSE_OK;
19. }
20.
21. /* value = null / false / true */
22. /* 提示：下面代码没处理 false / true, 将会是练习之一 */
23. static int lept_parse_value(lept_context* c, lept_value* v) {
24.     switch (*c->json) {
25.         case 'n': return lept_parse_null(c, v);
26.         case '\0': return LEPT_PARSE_EXPECT_VALUE;
27.         default: return LEPT_PARSE_INVALID_VALUE;
28.     }
29. }

```

由于 `lept_parse_whitespace()` 是不会出现错误的，返回类型为 `void`。其它的解析函数会返回错误码，传递至顶层。

关于断言

断言 (assertion) 是 C 语言中常用的防御式编程方式，减少编程错误。最常用的是在函数开始的地方，检测所有参数。有时候也可以在调用函数后，检查上下文是否正确。

C 语言的标准库含有 `assert()` 这个宏（需 `#include <assert.h>`），提供断言功能。当程序以 `release` 配置编译时（定义了 `NDEBUG` 宏），`assert()` 不会做检测；而当在 `debug` 配置时（没定义 `NDEBUG` 宏），则会在运行时检测 `assert(cond)` 中的条件是否为真（非 0），断言失败会直接令程序崩溃。

例如上面的 `lept_parse_null()` 开始时，当前字符应该是 `'n'`，所以我们使用一个宏 `EXPECT(c, ch)` 进行断言，并跳到下一字符。

初使用断言的同学，可能会错误地把含副作用的代码放在中：

```
assert()
```

```
1. assert(x++ == 0); /* 这是错误的! */
```

这样会导致 debug 和 release 版的行为不一样。

另一个问题是，初学者可能会难于分辨何时使用断言，何时处理运行时错误（如返回错误值或在 C++ 中抛出异常）。简单的答案是，如果那个错误是由于程序员错误编码所造成的（例如传入不合法的参数），那么应用断言；如果那个错误是程序员无法避免，而是由运行时的环境所造成的，就要处理运行时错误（例如开启文件失败）。

总结与练习

本文介绍了如何配置一个编程环境，单元测试的重要性，以至于一个 JSON 解析器的子集实现。如果你读到这里，还未动手，建议你快点试一下。以下是本单元的练习，很容易的，但我也会在稍后发出解答篇。

1. 修正关于 `LEPT_PARSE_ROOT_NOT_SINGULAR` 的单元测试，若 json 在一个值之后，空白之后还有其它字符，则要返回 `LEPT_PARSE_ROOT_NOT_SINGULAR`。
2. 参考 `test_parse_null()`，加入 `test_parse_true()`、`test_parse_false()` 单元测试。
3. 参考 `lept_parse_null()` 的实现和调用方，解析 true 和 false 值。

常见问答

1. 为什么把例子命名为 leptjson？

来自于标准模型中的轻子（lepton），意为很轻量的 JSON 库。另外，建议大家为项目命名时，先 google 一下是否够独特，有很多同名的话搜寻时难以命中。

2. 为什么使用宏而不用函数或内联函数？

因为这个测试框架使用了 `__LINE__` 这个编译器提供的宏，代表编译时该行的行号。如果用函数或内联函数，每次的行号便都会相同。另外，内联函数是 C99 的新增功能，本教程使用 C89。

其他常见问答将会从评论中整理。

从零开始的 JSON 库教程（一）：启程解答篇

- 从零开始的 JSON 库教程（一）：启程解答篇
 - 1. 修正 LEPT_PARSE_ROOT_NOT_SINGULAR
 - 2. true/false 单元测试
 - 3. true/false 解析
 - 4. 总结

从零开始的 JSON 库教程（一）：启程解答篇

- Milo Yip
- 2016/9/17

本文是《从零开始的 JSON 库教程》的第一个单元解答篇。解答代码位于 [json-tutorial/tutorial01_answer](https://github.com/miloyip/json-tutorial/blob/master/tutorial01_answer.c)。

1. 修正 LEPT_PARSE_ROOT_NOT_SINGULAR

单元测试失败的是这一行：

```
1. EXPECT_EQ_INT(LEPT_PARSE_ROOT_NOT_SINGULAR, lept_parse(&v, "null
x"));
```

我们从 JSON 语法发现，JSON 文本应该有 3 部分：

```
1. JSON-text = ws value ws
```

但原来的 `lept_parse()` 只处理了前两部分。我们只需要加入第三部分，解析空白，然后检查 JSON 文本是否完结：

```
1. int lept_parse(lept_value* v, const char* json) {
2.     lept_context c;
```

```

3.     int ret;
4.     assert(v != NULL);
5.     c.json = json;
6.     v->type = LEPT_NULL;
7.     lept_parse_whitespace(&c);
8.     if ((ret = lept_parse_value(&c, v)) == LEPT_PARSE_OK) {
9.         lept_parse_whitespace(&c);
10.        if (*c.json != '\0')
11.            ret = LEPT_PARSE_ROOT_NOT_SINGULAR;
12.    }
13.    return ret;
14. }

```

有一些 JSON 解析器完整解析一个值之后就会顺利返回，这是不符合标准的。但有时候也有另一种需求，文本中含多个 JSON 或其他文本串接在一起，希望当完整解析一个值之后就停下来。因此，有一些 JSON 解析器会提供这种选项，例如 RapidJSON 的

`kParseStopWhenDoneFlag`。

2. true/false 单元测试

此问题很简单，只需参考 `test_parse_null()` 加入两个测试函数：

```

1. static void test_parse_true() {
2.     lept_value v;
3.     v.type = LEPT_FALSE;
4.     EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, "true"));
5.     EXPECT_EQ_INT(LEPT_TRUE, lept_get_type(&v));
6. }
7.
8. static void test_parse_false() {
9.     lept_value v;
10.    v.type = LEPT_TRUE;
11.    EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, "false"));
12.    EXPECT_EQ_INT(LEPT_FALSE, lept_get_type(&v));
13. }

```

```

14.
15. static void test_parse() {
16.     test_parse_null();
17.     test_parse_true();
18.     test_parse_false();
19.     test_parse_expect_value();
20.     test_parse_invalid_value();
21.     test_parse_root_not_singular();
22. }

```

但要记得在上一级的测试函数 `test_parse()` 调用这函数，否则会不起作用。还好如果我们记得用 `static` 修饰这两个函数，编译器会发出告警：

```

1. test.c:30:13: warning: unused function 'test_parse_true' [-Wunused-function]
2. static void test_parse_true() {
3.     ^

```

因为 `static` 函数的意思是指，该函数只作用于编译单元中，那么没有被调用时，编译器是能发现的。

3. true/false 解析

这部分很简单，只要参考 `lept_parse_null()`，再写两个函数，然后在 `lept_parse_value` 按首字符分派。

```

1. static int lept_parse_true(lept_context* c, lept_value* v) {
2.     EXPECT(c, 't');
3.     if (c->json[0] != 'r' || c->json[1] != 'u' || c->json[2] != 'e')
4.         return LEPT_PARSE_INVALID_VALUE;
5.     c->json += 3;
6.     v->type = LEPT_TRUE;
7.     return LEPT_PARSE_OK;

```

```

8.  }
9.
10. static int lept_parse_false(lept_context* c, lept_value* v) {
11.     EXPECT(c, 'f');
12.     if (c->json[0] != 'a' || c->json[1] != 'l' || c->json[2] != 's'
        || c->json[3] != 'e')
13.         return LEPT_PARSE_INVALID_VALUE;
14.     c->json += 4;
15.     v->type = LEPT_FALSE;
16.     return LEPT_PARSE_OK;
17. }
18.
19. static int lept_parse_value(lept_context* c, lept_value* v) {
20.     switch (*c->json) {
21.         case 't': return lept_parse_true(c, v);
22.         case 'f': return lept_parse_false(c, v);
23.         case 'n': return lept_parse_null(c, v);
24.         case '\0': return LEPT_PARSE_EXPECT_VALUE;
25.         default: return LEPT_PARSE_INVALID_VALUE;
26.     }
27. }

```

其实这 3 种类型都是解析字面量，可以使用单一个函数实现，例如用这种方式调用：

```

1.         case 'n': return lept_parse_literal(c, v, "null",
        LEPT_NULL);

```

这样可以减少一些重复代码，不过可能有少许额外性能开销。

4. 总结

如果你能完成这个练习，恭喜你！我想你通过亲自动手，会对教程里所说的有更深入的理解。如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

下一单元是和数字类型相关，敬请期待。

从零开始的 JSON 库教程（二）：解析数字

- [从零开始的 JSON 库教程（二）：解析数字](#)
- [1. 初探重构](#)
- [2. JSON 数字语法](#)
- [3. 数字表示方式](#)
- [4. 单元测试](#)
- [5. 十进制转换至二进制](#)
- [6. 总结与练习](#)
- [7. 参考](#)
- [8. 常见问题](#)

从零开始的 JSON 库教程（二）：解析数字

- Milo Yip
- 2016/9/18

本文是《[从零开始的 JSON 库教程](#)》的第二个单元。本单元的源代码位于 [json-tutorial/tutorial02](#)。

本单元内容：

1. [初探重构](#)
2. [JSON 数字语法](#)
3. [数字表示方式](#)
4. [单元测试](#)
5. [十进制转换至二进制](#)
6. [总结与练习](#)
7. [参考](#)
8. [常见问题](#)

1. 初探重构

在讨论解析数字之前，我们再补充 TDD 中的一个步骤——重构（refactoring）。根据[1]，重构是一个这样的过程：

在不改变代码外在行为的情况下，对代码作出修改，以改进程序的内部结构。

在 TDD 的过程中，我们的目标是编写代码去通过测试。但由于这个目标的引导性太强，我们可能会忽略正确性以外的软件品质。在通过测试之后，代码的正确性得以保证，我们就应该审视现时的代码，看看有没有地方可以改进，而同时能维持测试顺利通过。我们可以安心地做各种修改，因为我们有单元测试，可以判断代码在修改后是否影响原来的行为。

那么，哪里要作出修改？Beck 和 Fowler（[1] 第 3 章）认为程序员要培养一种判断能力，找出程序中的坏味道。例如，在第一单元的练习中，可能大部分人都会复制 `lept_parse_null()` 的代码，作一些修改，成为 `lept_parse_true()` 和 `lept_parse_false()`。如果我们再审视这 3 个函数，它们非常相似。这违反编程中常说的 DRY（don't repeat yourself）原则。本单元的第一个练习题，就是尝试合并这 3 个函数。

另外，我们也可能发现，单元测试代码也有很重复的代码，例如 `test_parse_invalid_value()` 中我们每次测试一个不合法的 JSON 值，都有 4 行相似的代码。我们可以把它用宏的方式把它们简化：

```
1. #define TEST_ERROR(error, json)\
2.     do {\
3.         lept_value v;\
4.         v.type = LEPT_FALSE;\
5.         EXPECT_EQ_INT(error, lept_parse(&v, json));\
6.         EXPECT_EQ_INT(LEPT_NULL, lept_get_type(&v));\
7.     } while(0)
```

```

8.
9. static void test_parse_expect_value() {
10.     TEST_ERROR(LEPT_PARSE_EXPECT_VALUE, "");
11.     TEST_ERROR(LEPT_PARSE_EXPECT_VALUE, " ");
12. }

```

最后，我希望指出，软件的架构难以用单一标准评分，重构时要考虑平衡各种软件品质。例如上述把 3 个函数合并后，优点是减少重复的代码，维护较容易，但缺点可能是带来性能的少量影响。

2. JSON 数字语法

回归正题，本单元的重点在于解析 JSON number 类型。我们先看看它的语法：

```

1. number = [ "-" ] int [ frac ] [ exp ]
2. int = "0" / digit1-9 *digit
3. frac = "." 1*digit
4. exp = ("e" / "E") [ "-" / "+" ] 1*digit

```

number 是以十进制表示，它主要由 4 部分顺序组成：负号、整数、小数、指数。只有整数是必需部分。注意和直觉可能不同的是，正号是不合法的。

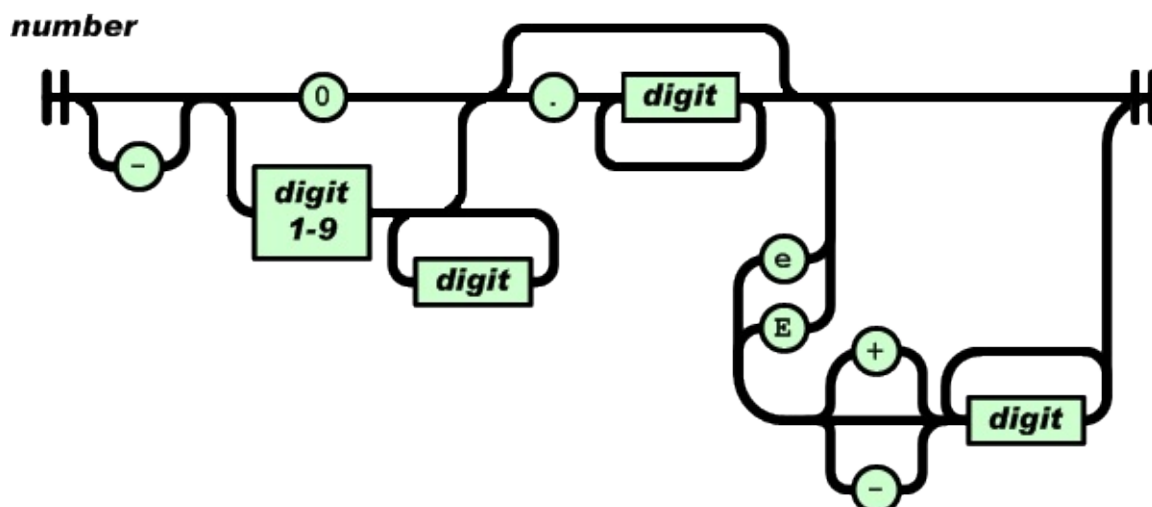
整数部分如果是 0 开始，只能是单个 0；而由 1-9 开始的话，可以加任意数量的数字（0-9）。也就是说，`0123` 不是一个合法的 JSON 数字。

小数部分比较直观，就是小数点后是一或多个数字（0-9）。

JSON 可使用科学记数法，指数部分由大写 E 或小写 e 开始，然后可有正负号，之后是一或多个数字（0-9）。

JSON 标准 [ECMA-404](#) 采用图的形式表示语法，也可以更直观地看到

解析时可能经过的路径：



上一单元的 `null`、`false`、`true` 在解析后，我们只需把它们存储为类型。但对于数字，我们要考虑怎么存储解析后的结果。

3. 数字表示方式

从 JSON 数字的语法，我们可能直观地会认为它应该表示为一个浮点数 (floating point number)，因为它带有小数和指数部分。然而，标准中并没有限制数字的范围或精度。为简单起见，`leptjson` 选择以双精度浮点数 (C 中的 `double` 类型) 来存储 JSON 数字。

我们为 `lept_value` 添加成员：

```
1. typedef struct {
2.     double n;
3.     lept_type type;
4. }lept_value;
```

仅当 `type == LEPT_NUMBER` 时，`n` 才表示 JSON 数字的数值。所以获取该值的 API 是这么实现的：

```
1. double lept_get_number(const lept_value* v) {
```

```

2.     assert(v != NULL && v->type == LEPT_NUMBER);
3.     return v->n;
4. }

```

使用者应确保类型正确，才调用此 API。我们继续使用断言来保证。

4. 单元测试

我们定义了 API 之后，按照 TDD，我们可以先写一些单元测试。这次我们使用多行的宏的减少重复代码：

```

1. #define TEST_NUMBER(expect, json)\
2.     do {\
3.         lept_value v;\
4.         EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, json));\
5.         EXPECT_EQ_INT(LEPT_NUMBER, lept_get_type(&v));\
6.         EXPECT_EQ_DOUBLE(expect, lept_get_number(&v));\
7.     } while(0)
8.
9. static void test_parse_number() {
10.     TEST_NUMBER(0.0, "0");
11.     TEST_NUMBER(0.0, "-0");
12.     TEST_NUMBER(0.0, "-0.0");
13.     TEST_NUMBER(1.0, "1");
14.     TEST_NUMBER(-1.0, "-1");
15.     TEST_NUMBER(1.5, "1.5");
16.     TEST_NUMBER(-1.5, "-1.5");
17.     TEST_NUMBER(3.1416, "3.1416");
18.     TEST_NUMBER(1E10, "1E10");
19.     TEST_NUMBER(1e10, "1e10");
20.     TEST_NUMBER(1E+10, "1E+10");
21.     TEST_NUMBER(1E-10, "1E-10");
22.     TEST_NUMBER(-1E10, "-1E10");
23.     TEST_NUMBER(-1e10, "-1e10");
24.     TEST_NUMBER(-1E+10, "-1E+10");
25.     TEST_NUMBER(-1E-10, "-1E-10");
26.     TEST_NUMBER(1.234E+10, "1.234E+10");

```

```

27.     TEST_NUMBER(1.234E-10, "1.234E-10");
28.     TEST_NUMBER(0.0, "1e-10000"); /* must underflow */
29. }

```

以上这些都是很基本的测试用例，也可供调试用。大部分情况下，测试案例不能穷举所有可能性。因此，除了加入一些典型的用例，我们也常会使用一些边界值，例如最大值等。练习中会让同学找一些边界值作为用例。

除了这些合法的 JSON，我们也要写一些不合语法的用例：

```

1. static void test_parse_invalid_value() {
2.     /* ... */
3.     /* invalid number */
4.     TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "+0");
5.     TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "+1");
6.     TEST_ERROR(LEPT_PARSE_INVALID_VALUE, ".123"); /* at least one
digit before '.' */
7.     TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "1."); /* at least one
digit after '.' */
8.     TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "INF");
9.     TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "inf");
10.    TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "NAN");
11.    TEST_ERROR(LEPT_PARSE_INVALID_VALUE, "nan");
12. }

```

5. 十进制转换至二进制

我们需要把十进制的数字转换成二进制的 `double`。这并不是容易的事情 [2]。为了简单起见，leptjson 将使用标准库的 `strtod()` 来进行转换。`strtod()` 可转换 JSON 所要求的格式，但问题是，一些 JSON 不容许的格式，`strtod()` 也可转换，所以我们需要自行做格式校验。

```

1. #include <stdlib.h> /* NULL, strtod() */
2.
3. static int lept_parse_number(lept_context* c, lept_value* v) {
4.     char* end;
5.     /* \TODO validate number */
6.     v->n = strtod(c->json, &end);
7.     if (c->json == end)
8.         return LEPT_PARSE_INVALID_VALUE;
9.     c->json = end;
10.    v->type = LEPT_NUMBER;
11.    return LEPT_PARSE_OK;
12. }

```

加入了 number 后, value 的语法变成:

```
1. value = null / false / true / number
```

记得在第一单元中, 我们说可以用一个字符就能得知 value 是什么类型, 有 11 个字符可判断 number:

- 0-9/- → number

但是, 由于我们在 `lept_parse_number()` 内部将会校验输入是否正确
的值, 我们可以简单地把余下的情况都交给 `lept_parse_number()` :

```

1. static int lept_parse_value(lept_context* c, lept_value* v) {
2.     switch (*c->json) {
3.         case 't': return lept_parse_true(c, v);
4.         case 'f': return lept_parse_false(c, v);
5.         case 'n': return lept_parse_null(c, v);
6.         default: return lept_parse_number(c, v);
7.         case '\0': return LEPT_PARSE_EXPECT_VALUE;
8.     }
9. }

```

6. 总结与练习

本单元讲述了 JSON 数字类型的语法，以及 leptjson 所采用的自行校验 + `strtod()` 转换为 `double` 的方案。实际上一些 JSON 库会采用更复杂的方案，例如支持 64 位带符号 / 无符号整数，自行实现转换。以我的个人经验，解析 / 生成数字类型可以说是 RapidJSON 中最难实现的部分，也是 RapidJSON 高效性能的原因，有机会再另外撰文解释。

此外我们谈及，重构与单元测试是互相依赖的软件开发技术，适当地运用可提升软件的品质。之后的单元还会有相关的话题。

1. 重构合并

`lept_parse_null()`、`lept_parse_false()`、`lept_parse_true` 为 `lept_parse_literal()`。

2. 加入 [维基百科双精度浮点数](#) 的一些边界值至单元测试，如 `min subnormal positive double`、`max double` 等。

3. 去掉 `test_parse_invalid_value()` 和 `test_parse_root_not_singular` 中的 `#if 0 ... #endif`，执行测试，证实测试失败。按 JSON number 的语法在 `lept_parse_number()` 校验，不符合标准的程况返回 `LEPT_PARSE_INVALID_VALUE` 错误码。

4. 去掉 `test_parse_number_too_big` 中的 `#if 0 ... #endif`，执行测试，证实测试失败。仔细阅读 `strtod()`，看看怎样从返回值得知数值是否过大，以返回 `LEPT_PARSE_NUMBER_TOO_BIG` 错误码。（提示：这里需要 `#include` 额外两个标准库头文件。）

以上最重要的是第 3 条题目，就是要校验 JSON 的数字语法。建议使用以下两个宏去简化一下代码：

```
1. #define ISDIGIT(ch)      ((ch) >= '0' && (ch) <= '9')
2. #define ISDIGIT1T09(ch) ((ch) >= '1' && (ch) <= '9')
```

另一提示，在校验成功以后，我们不再使用 `end` 指针去检测 `strtod()` 的正确性，第二个参数可传入 `NULL`。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

7. 参考

[1] Fowler, Martin. Refactoring: improving the design of existing code. Pearson Education India, 2009. 中译本：《重构：改善既有代码的设计》，熊节译，人民邮电出版社，2010年。

[2] Gay, David M. "Correctly rounded binary-decimal and decimal-binary conversions." Numerical Analysis Manuscript 90-10 (1990).

8. 常见问题

1. 为什么要把一些测试代码以 `#if 0 ... #endif` 禁用？

因为在做第 1 个练习题时，我希望能 100% 通过测试，方便做重构。另外，使用 `#if 0 ... #endif` 而不使用 `/* ... */`，是因为 C 的注释不支持嵌套（nested），而 `#if ... #endif` 是支持嵌套的。代码中已有注释时，用 `#if 0 ... #endif` 去禁用代码是一个常用技巧，而且可以把 `0` 改为 `1` 去恢复。

2. 科学计数法的指数部分没有对前导零作限制吗？`1E012` 也是合法的吗？

是的，这是合法的。JSON 源自于 JavaScript ([ECMA-262, 3rd edition](#))，数字语法取自 JavaScript 的十进位数字的语法 (§7.8.3 Numeric Literals)。整数不容许前导零 (leading zero)，是因为更久的 JavaScript 版本容许以前导零来表示八进位数字，如 `052 == 42`，这种八进位常数表示方式来自于 C 语言。禁止前导零避免了可能出现的歧义。但是在指数里就不会出现这个问题。多谢 @Smallay 提出及协助解答这个问题。

其他常见问答将会从评论中整理。

从零开始的 JSON 库教程（二）：解析数字解答篇

- [从零开始的 JSON 库教程（二）：解析数字解答篇](#)
 - [1. 重构合并](#)
 - [2. 边界值测试](#)
 - [3. 校验数字](#)
 - [4. 数字过大的处理](#)
 - [5. 总结](#)

从零开始的 JSON 库教程（二）：解析数字解答篇

- Milo Yip
- 2016/9/20

本文是《[从零开始的 JSON 库教程](#)》的第二个单元解答篇。解答代码位于 [json-tutorial/tutorial02_answer](#)。

1. 重构合并

由于 `true` / `false` / `null` 的字符数量不一样，这个答案以 `for` 循环作比较，直至 `'\0'`。

```
1. static int lept_parse_literal(lept_context* c, lept_value* v, const
   char* literal, lept_type type) {
2.     size_t i;
3.     EXPECT(c, literal[0]);
4.     for (i = 0; literal[i + 1]; i++)
5.         if (c->json[i] != literal[i + 1])
6.             return LEPT_PARSE_INVALID_VALUE;
7.     c->json += i;
8.     v->type = type;
```



```

9.     return LEPT_PARSE_OK;
10. }
11.
12. static int lept_parse_value(lept_context* c, lept_value* v) {
13.     switch (*c->json) {
14.         case 't': return lept_parse_literal(c, v, "true",
LEPT_TRUE);
15.         case 'f': return lept_parse_literal(c, v, "false",
LEPT_FALSE);
16.         case 'n': return lept_parse_literal(c, v, "null",
LEPT_NULL);
17.         /* ... */
18.     }
19. }

```

注意在 C 语言中，数组长度、索引值最好使用 `size_t` 类型，而不是 `int` 或 `unsigned`。

你也可以直接传送长度参数 4、5、4，只要能通过测试就行了。

2. 边界值测试

这问题其实涉及一些浮点数类型的细节，例如 IEEE-754 浮点数中，有所谓的 `normal` 和 `subnormal` 值，这里暂时不展开讨论了。以下是我加入的一些边界值，可能和同学的不完全一样。

```

1. TEST_NUMBER(1.00000000000000002, "1.00000000000000002"); /* the
smallest number > 1 */
2. TEST_NUMBER( 4.9406564584124654e-324, "4.9406564584124654e-324");
/* minimum denormal */
3. TEST_NUMBER(-4.9406564584124654e-324, "-4.9406564584124654e-324");
4. TEST_NUMBER( 2.2250738585072009e-308, "2.2250738585072009e-308");
/* Max subnormal double */
5. TEST_NUMBER(-2.2250738585072009e-308, "-2.2250738585072009e-308");
6. TEST_NUMBER( 2.2250738585072014e-308, "2.2250738585072014e-308");
/* Min normal positive double */

```

```

7. TEST_NUMBER(-2.2250738585072014e-308, "-2.2250738585072014e-308");
8. TEST_NUMBER( 1.7976931348623157e+308, "1.7976931348623157e+308");
   /* Max double */
9. TEST_NUMBER(-1.7976931348623157e+308, "-1.7976931348623157e+308");

```

另外，这些加入的测试用例，正常的 `strtod()` 都能通过。所以不能做到测试失败、修改实现、测试成功的 TDD 步骤。

有一些 JSON 解析器不使用 `strtod()` 而自行转换，例如在校验的同时，记录负号、尾数（整数和小数）和指数，然后 naive 地计算：

```

1. int negative = 0;
2. int64_t mantissa = 0;
3. int exp = 0;
4.
5. /* 解析... 并存储 negative, mantissa, exp */
6. v->n = (negative ? -mantissa : mantissa) * pow(10.0, exp);

```

这种做法会有精度问题。实现正确的答案是很复杂的，RapidJSON 的初期版本也是 naive 的，后来 RapidJSON 就内部实现了三种算法（使用 `kParseFullPrecision` 选项开启），最后一种算法用到了大整数（高精度计算）。有兴趣的同学也可以先尝试做一个 naive 版本，不使用 `strtod()`。之后可再参考 Google 的 [double-conversion](#) 开源项目及相关论文。

3. 校验数字

这条题目是本单元的重点，考验同学是否能把语法手写为校验规则。我详细说明。

首先，如同 `lept_parse_whitespace()`，我们使用一个指针 `p` 来表示当前的解析字符位置。这样做有两个好处，一是代码更简单，二是在某些编译器下性能更好（因为不能确定 `c` 会否被改变，从而每次更

改 `c->json` 都要做一次间接访问)。如果校验成功，才把 `p` 赋值至 `c->json`。

```

1. static int lept_parse_number(lept_context* c, lept_value* v) {
2.     const char* p = c->json;
3.     /* 负号 ... */
4.     /* 整数 ... */
5.     /* 小数 ... */
6.     /* 指数 ... */
7.     v->n = strtod(c->json, NULL);
8.     v->type = LEPT_NUMBER;
9.     c->json = p;
10.    return LEPT_PARSE_OK;
11. }
```

我们把语法再看一遍：

```

1. number = [ "-" ] int [ frac ] [ exp ]
2. int = "0" / digit1-9 *digit
3. frac = "." 1*digit
4. exp = ("e" / "E") ["-" / "+"] 1*digit
```

负号最简单，有的话跳过便行：

```

1.     if (*p == '-') p++;
```

整数部分有两种合法情况，一是单个 `0`，否则是一个 1-9 再加上任意数量的 `digit`。对于第一种情况，我们像负数般跳过便行。对于第二种情况，第一个字符必须为 1-9，如果否定的就是不合法的，可立即返回错误码。然后，有多少个 `digit` 就跳过多少个。

```

1.     if (*p == '0') p++;
2.     else {
3.         if (!ISDIGIT1T09(*p)) return LEPT_PARSE_INVALID_VALUE;
4.         for (p++; ISDIGIT(*p); p++);
```

```
5.      }
```

如果出现小数点，我们跳过该小数点，然后检查它至少应有一个 digit，不是 digit 就返回错误码。跳过首个 digit，我们再检查有没有 digit，有多少个跳过多少个。这里用了 for 循环技巧来做这件事。

```
1.      if (*p == '.') {
2.          p++;
3.          if (!ISDIGIT(*p)) return LEPT_PARSE_INVALID_VALUE;
4.          for (p++; ISDIGIT(*p); p++);
5.      }
```

最后，如果出现大小写 `e`，就表示有指数部分。跳过那个 `e` 之后，可以有一个正或负号，有的话就跳过。然后和小数的逻辑是一样的。

```
1.      if (*p == 'e' || *p == 'E') {
2.          p++;
3.          if (*p == '+' || *p == '-') p++;
4.          if (!ISDIGIT(*p)) return LEPT_PARSE_INVALID_VALUE;
5.          for (p++; ISDIGIT(*p); p++);
6.      }
```

这里用了 18 行代码去做这个校验。当中把一些 if 用一行来排版，而没用采用传统两行缩进风格，我个人认为在不影响阅读时可以这样弹性处理。当然那些 for 也可分拆成三行：

```
1.          p++;
2.          while (ISDIGIT(*p))
3.              p++;
```

4. 数字过大的处理

最后这题纯粹是阅读理解题。

```

1. #include <errno.h>    /* errno, ERANGE */
2. #include <math.h>     /* HUGE_VAL */
3.
4. static int lept_parse_number(lept_context* c, lept_value* v) {
5.     /* ... */
6.     errno = 0;
7.     v->n = strtod(c->json, NULL);
8.     if (errno == ERANGE && v->n == HUGE_VAL) return
        LEPT_PARSE_NUMBER_TOO_BIG;
9.     /* ... */
10. }
```

许多时候课本 / 书籍也不会把每个标准库功能说得很仔细，我想藉此提醒同学要好好看参考文档，学会读文档编程就简单得多！cppreference.com 是 C/C++ 程序员的宝库。

5. 总结

本单元的习题比上个单元较有挑战性一些，所以我花了多一些篇幅在解答篇。纯以语法来说，数字类型已经是 JSON 中最复杂的类型。如果同学能完成本单元的练习（特别是第 3 条），之后的字符串、数组和对象的语法一定难不到你。然而，接下来也会有一些新挑战，例如内存管理、数据结构、编码等，希望你能满载而归。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（三）：解析字符串

- [从零开始的 JSON 库教程（三）：解析字符串](#)
 - [1. JSON 字符串语法](#)
 - [2. 字符串表示](#)
 - [3. 内存管理](#)
 - [4. 缓冲区与堆栈](#)
 - [5. 解析字符串](#)
 - [6. 总结和练习](#)
 - [7. 参考](#)
- [8. 常见问题](#)

从零开始的 JSON 库教程（三）：解析字符串

- Milo Yip
- 2016/9/22

本文是《[从零开始的 JSON 库教程](#)》的第三个单元。本单元的练习源代码位于 [json-tutorial/tutorial03](#)。

本单元内容：

1. [JSON 字符串语法](#)
2. [字符串表示](#)
3. [内存管理](#)
4. [缓冲区与堆栈](#)
5. [解析字符串](#)
6. [总结和练习](#)
7. [参考](#)
8. [常见问题](#)

1. JSON 字符串语法

JSON 的字符串语法和 C 语言很相似，都是以双引号把字符括起来，如 `"Hello"`。但字符串采用了双引号作分隔，那么怎样可以在字符串中插入一个双引号？把 `a"b` 写成 `"a"b"` 肯定不行，都不知道那里是字符串的结束了。因此，我们需要引入转义字符（escape character），C 语言和 JSON 都使用 `\`（反斜线）作为转义字符，那么 `"` 在字符串中就表示为 `\"`，`a"b` 的 JSON 字符串则写成 `"a\"b"`。如以下的字符串语法所示，JSON 共支持 9 种转义序列：

```

1. string = quotation-mark *char quotation-mark
2. char = unescaped /
3.     escape (
4.         %x22 /           ; "    quotation mark    U+0022
5.         %x5C /           ; \    reverse solidus   U+005C
6.         %x2F /           ; /    solidus           U+002F
7.         %x62 /           ; b    backspace         U+0008
8.         %x66 /           ; f    form feed         U+000C
9.         %x6E /           ; n    line feed          U+000A
10.        %x72 /           ; r    carriage return    U+000D
11.        %x74 /           ; t    tab                 U+0009
12.        %x75 4HEXDIG ) ; uXXXX                    U+XXXX
13. escape = %x5C          ; \
14. quotation-mark = %x22 ; "
15. unescaped = %x20-21 / %x23-5B / %x5D-10FFFF

```

简单翻译一下，JSON 字符串是由前后两个双引号夹着零至多个字符。字符分为无转义字符或转义序列。转义序列有 9 种，都是以反斜线开始，如常见的 `\n` 代表换行符。比较特殊的是 `\uXXXX`，当中 XXXX 为 16 进位的 UTF-16 编码，本单元将不处理这种转义序列，留待下回分解。

无转义字符就是普通的字符，语法中列出了合法的码点范围（码点还是在下单元才介绍）。要注意的是，该范围不包括 0 至 31、双引号和反斜线，这些码点都必须使用转义方式表示。

2. 字符串表示

在 C 语言中，字符串一般表示为空结尾字符串（null-terminated string），即以空字符（`'\0'`）代表字符串的结束。然而，JSON 字符串是允许含有空字符的，例如这个 JSON `"Hello\u0000World"` 就是单个字符串，解析后为11个字符。如果纯粹使用空结尾字符来表示 JSON 解析后的结果，就没法处理空字符。

因此，我们可以分配内存来储存解析后的字符，以及记录字符的数目（即字符串长度）。由于大部分 C 程序都假设字符串是空结尾字符串，我们还是在最后加上一个空字符，那么不需处理 `\u0000` 这种字符的应用可以简单地把它当作是空结尾字符串。

了解需求后，我们考虑实现。`lept_value` 事实上是一种变体类型（variant type），我们通过 `type` 来决定它现时是哪种类型，而这也决定了哪些成员是有效的。首先我们简单地在这个结构中加入两个成员：

```
1. typedef struct {
2.     char* s;
3.     size_t len;
4.     double n;
5.     lept_type type;
6. }lept_value;
```

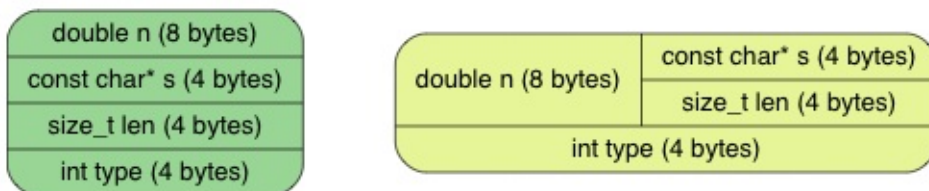
然而我们知道，一个值不可能同时为数字和字符串，因此我们可使用 C 语言的 `union` 来节省内存：


```

1. typedef struct {
2.     union {
3.         struct { char* s; size_t len; }s; /* string */
4.         double n; /* number */
5.     }u;
6.     lept_type type;
7. }lept_value;

```

这两种设计在 32 位平台时的内存布局如下，可看出右方使用 `union` 的能省下内存。



我们要把之前的 `v->n` 改成 `v->u.n`。而要访问字符串的数据，则使用 `v->u.s.s` 和 `v->u.s.len`。这种写法比较麻烦吧，其实 C11 新增了匿名 struct/union 语法，就可以采用 `v->n`、`v->s`、`v->len` 来作访问。

3. 内存管理

由于字符串的长度不是固定的，我们要动态分配内存。为简单起见，我们使用标准库 `<stdlib.h>` 中的 `malloc()`、`realloc()` 和 `free()` 来分配 / 释放内存。

当设置一个值为字符串时，我们需要把参数中的字符串复制一份：

```

1. void lept_set_string(lept_value* v, const char* s, size_t len) {
2.     assert(v != NULL && (s != NULL || len == 0));
3.     lept_free(v);
4.     v->u.s.s = (char*)malloc(len + 1);
5.     memcpy(v->u.s.s, s, len);

```

```

6.     v->u.s.s[len] = '\0';
7.     v->u.s.len = len;
8.     v->type = LEPT_STRING;
9. }

```

断言中的条件是，非空指针（有具体的字符串）或是零长度的字符串都是合法的。

注意，在设置这个 `v` 之前，我们需要先调用 `lept_free(v)` 去清空 `v` 可能分配到的内存。例如原来已有一字符串，我们要先把它释放。然后就是简单地用 `malloc()` 分配及用 `memcpy()` 复制，并补上结尾空字符。`malloc(len + 1)` 中的 1 是因为结尾空字符。

那么，再看看 `lept_free()`：

```

1. void lept_free(lept_value* v) {
2.     assert(v != NULL);
3.     if (v->type == LEPT_STRING)
4.         free(v->u.s.s);
5.     v->type = LEPT_NULL;
6. }

```

现时仅当值是字符串类型，我们才要处理，之后我们还要加上对数组及对象的释放。`lept_free(v)` 之后，会把它的类型变成 `null`。这个设计能避免重复释放。

但也由于我们会检查 `v` 的类型，在调用所有访问函数之前，我们必须初始化该类型。所以我们加入 `lept_init(v)`，因非常简单我们用宏实现：

```

1. #define lept_init(v) do { (v)->type = LEPT_NULL; } while(0)

```

用上 `do { ... } while(0)` 是为了把表达式转为语句，模仿无返回值的函数。

其实在前两个单元中，我们只提供读取值的 API，没有写入的 API，就是因为写入时我们还要考虑释放内存。我们在本单元中把它们补全：

```

1. #define lept_set_null(v) lept_free(v)
2.
3. int lept_get_boolean(const lept_value* v);
4. void lept_set_boolean(lept_value* v, int b);
5.
6. double lept_get_number(const lept_value* v);
7. void lept_set_number(lept_value* v, double n);
8.
9. const char* lept_get_string(const lept_value* v);
10. size_t lept_get_string_length(const lept_value* v);
11. void lept_set_string(lept_value* v, const char* s, size_t len);

```

由于 `lept_free()` 实际上也会把 `v` 变成 `null` 值，我们只用一个宏来提供 `lept_set_null()` 这个 API。

应用方的代码在调用 `lept_parse()` 之后，最终也应该调用 `lept_free()` 去释放内存。我们把之前的单元测试也加入此调用。

如果不使用 `lept_parse()`，我们需要初始化值，那么就像以下的单元测试，先 `lept_init()`，最后 `lept_free()`。

```

1. static void test_access_string() {
2.     lept_value v;
3.     lept_init(&v);
4.     lept_set_string(&v, "", 0);
5.     EXPECT_EQ_STRING("", lept_get_string(&v),
6. lept_get_string_length(&v));
7.     lept_set_string(&v, "Hello", 5);
8.     EXPECT_EQ_STRING("Hello", lept_get_string(&v),
9. lept_get_string_length(&v));
10.    lept_free(&v);
11. }

```

4. 缓冲区与堆栈

我们解析字符串（以及之后的数组、对象）时，需要把解析的结果先储存在一个临时的缓冲区，最后再用 `lept_set_string()` 把缓冲区的结果设进值之中。在完成解析一个字符串之前，这个缓冲区的大小是不能预知的。因此，我们可以采用动态数组（dynamic array）这种数据结构，即数组空间不足时，能自动扩展。C++ 标准库的 `std::vector` 也是一种动态数组。

如果每次解析字符串时，都重新建一个动态数组，那么是比较耗时的。我们可以重用这个动态数组，每次解析 JSON 时就只需要创建一个。而且我们将会发现，无论是解析字符串、数组或对象，我们也只需要以先进后出的方式访问这个动态数组。换句话说，我们需要一个动态的堆栈数据结构。

我们把一个动态堆栈的数据放进 `lept_context` 里：

```
1. typedef struct {
2.     const char* json;
3.     char* stack;
4.     size_t size, top;
5. }lept_context;
```

当中 `size` 是当前的堆栈容量，`top` 是栈顶的位置（由于我们会扩展 `stack`，所以不要把 `top` 用指针形式存储）。

然后，我们在创建 `lept_context` 的时候初始化 `stack` 并最终释放内存：

```
1. int lept_parse(lept_value* v, const char* json) {
2.     lept_context c;
3.     int ret;
4.     assert(v != NULL);
```

```

5.     c.json = json;
6.     c.stack = NULL;          /* <- */
7.     c.size = c.top = 0;      /* <- */
8.     lept_init(v);
9.     lept_parse_whitespace(&c);
10.    if ((ret = lept_parse_value(&c, v)) == LEPT_PARSE_OK) {
11.        /* ... */
12.    }
13.    assert(c.top == 0);        /* <- */
14.    free(c.stack);            /* <- */
15.    return ret;
16. }

```

在释放时，加入了断言确保所有数据都被弹出。

然后，我们实现堆栈的压入及弹出操作。和普通的堆栈不一样，我们这个堆栈是以字节储存的。每次可要求压入任意大小的数据，它会返回数据起始的指针（会 C++ 的同学可再参考[1]）：

```

1.  #ifndef LEPT_PARSE_STACK_INIT_SIZE
2.  #define LEPT_PARSE_STACK_INIT_SIZE 256
3.  #endif
4.
5.  static void* lept_context_push(lept_context* c, size_t size) {
6.      void* ret;
7.      assert(size > 0);
8.      if (c->top + size >= c->size) {
9.          if (c->size == 0)
10.             c->size = LEPT_PARSE_STACK_INIT_SIZE;
11.          while (c->top + size >= c->size)
12.             c->size += c->size >> 1; /* c->size * 1.5 */
13.          c->stack = (char*)realloc(c->stack, c->size);
14.      }
15.      ret = c->stack + c->top;
16.      c->top += size;
17.      return ret;
18. }

```

```

19.
20. static void* lept_context_pop(lept_context* c, size_t size) {
21.     assert(c->top >= size);
22.     return c->stack + (c->top -= size);
23. }

```

压入时若空间不足，便回以 1.5 倍大小扩展。为什么是 1.5 倍而不是两倍？可参考我在 [STL 的 vector 有哪些封装上的技巧？](#) 的答案。

注意到这里使用了 `realloc()` 来重新分配内存，`c->stack` 在初始化为 `NULL`，`realloc(NULL, size)` 的行为是等价于 `malloc(size)` 的，所以我们不需要为第一次分配内存作特别处理。

另外，我们把初始大小以宏 `LEPT_PARSE_STACK_INIT_SIZE` 的形式定义，使用 `#ifndef X #define X ... #endif` 方式的好处是，使用者可在编译选项中自行设置宏，没设置的话就用缺省值。

5. 解析字符串

有了以上的工具，解析字符串的任务就变得很简单。我们只需要先备份栈顶，然后把解析到的字符压栈，最后计算出长度并一次性把所有字符弹出，再设置至值里便可以。以下是部分实现，没有处理转义和一些不合法字符的校验。

```

1. #define PUTC(c, ch) do { *(char*)lept_context_push(c, sizeof(char))
   = (ch); } while(0)
2.
3. static int lept_parse_string(lept_context* c, lept_value* v) {
4.     size_t head = c->top, len;
5.     const char* p;
6.     EXPECT(c, '"');
7.     p = c->json;
8.     for (;;) {

```

```

9.         char ch = *p++;
10.        switch (ch) {
11.            case '\\':
12.                len = c->top - head;
13.                lept_set_string(v, (const char*)lept_context_pop(c,
len), len);
14.                c->json = p;
15.                return LEPT_PARSE_OK;
16.            case '\\0':
17.                c->top = head;
18.                return LEPT_PARSE_MISS_QUOTATION_MARK;
19.            default:
20.                PUTC(c, ch);
21.        }
22.    }
23. }

```

6. 总结和练习

之前的单元都是固定长度的数据类型（fixed length data type），而字符串类型是可变长度的数据类型（variable length data type），因此本单元花了较多篇幅讲述内存管理和数据结构的设计和实现。字符串的解析相对数字简单，以下的习题难度不高，同学们应该可轻松完成。

1. 编写 `lept_get_boolean()` 等访问函数的单元测试，然后实现。
2. 实现除了 `\u` 以外的转义序列解析，令 `test_parse_string()` 中所有测试通过。
3. 解决 `test_parse_invalid_string_escape()` 和 `test_parse_invalid_string_char()` 中的失败测试。
4. 思考如何优化 `test_parse_string()` 的性能，那些优化方法有没有缺点。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

7. 参考

[1] [RapidJSON 代码剖析（一）：混合任意类型的堆栈](#)

8. 常见问题

其他常见问答将会从评论中整理。

从零开始的 JSON 库教程（三）：解析字符串解答篇

- [从零开始的 JSON 库教程（三）：解析字符串解答篇](#)
 - [1. 访问的单元测试](#)
 - [1A. Windows 下的内存泄漏检测方法](#)
 - [1B. Linux/OSX 下的内存泄漏检测方法](#)
 - [2. 转义序列的解析](#)
 - [3. 不合法的字符串](#)
 - [4. 性能优化的思考](#)
 - [5. 总结](#)

从零开始的 JSON 库教程（三）：解析字符串解答篇

- Milo Yip
- 2016/9/27

本文是《[从零开始的 JSON 库教程](#)》的第三个单元解答编。解答代码位于 [json-tutorial/tutorial03_answer](#)。

1. 访问的单元测试

在编写单元测试时，我们故意先把值设为字符串，那么做可以测试设置其他类型时，有没有调用 `lept_free()` 去释放内存。

```
1. static void test_access_boolean() {  
2.     lept_value v;  
3.     lept_init(&v);  
4.     lept_set_string(&v, "a", 1);  
5.     lept_set_boolean(&v, 1);
```

```

6.     EXPECT_TRUE(lept_get_boolean(&v));
7.     lept_set_boolean(&v, 0);
8.     EXPECT_FALSE(lept_get_boolean(&v));
9.     lept_free(&v);
10. }
11.
12. static void test_access_number() {
13.     lept_value v;
14.     lept_init(&v);
15.     lept_set_string(&v, "a", 1);
16.     lept_set_number(&v, 1234.5);
17.     EXPECT_EQ_DOUBLE(1234.5, lept_get_number(&v));
18.     lept_free(&v);
19. }

```

以下是访问函数的实现：

```

1. int lept_get_boolean(const lept_value* v) {
2.     assert(v != NULL && (v->type == LEPT_TRUE || v->type ==
   LEPT_FALSE));
3.     return v->type == LEPT_TRUE;
4. }
5.
6. void lept_set_boolean(lept_value* v, int b) {
7.     lept_free(v);
8.     v->type = b ? LEPT_TRUE : LEPT_FALSE;
9. }
10.
11. double lept_get_number(const lept_value* v) {
12.     assert(v != NULL && v->type == LEPT_NUMBER);
13.     return v->u.n;
14. }
15.
16. void lept_set_number(lept_value* v, double n) {
17.     lept_free(v);
18.     v->u.n = n;
19.     v->type = LEPT_NUMBER;
20. }

```

那问题是，如果我们没有调用 `lept_free()`，怎样能发现这些内存泄漏？

1A. Windows 下的内存泄漏检测方法

在 Windows 下，可使用 Visual C++ 的 [C Runtime Library \(CRT\)](#) 检测内存泄漏。

首先，我们在两个 `.c` 文件首行插入这一段代码：

```
1. #ifdef _WINDOWS
2. #define _CRTDBG_MAP_ALLOC
3. #include <crtdbg.h>
4. #endif
```

并在 `main()` 开始位置插入：

```
1. int main() {
2. #ifdef _WINDOWS
3.     _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
4. #endif
```

在 Debug 配置下按 F5 生成、开始调试程序，没有任何异样。

然后，我们删去 `lept_set_boolean()` 中的 `lept_free(v)`：

```
1. void lept_set_boolean(lept_value* v, int b) {
2.     /* lept_free(v); */
3.     v->type = b ? LEPT_TRUE : LEPT_FALSE;
4. }
```

再次按 F5 生成、开始调试程序，在输出会看到内存泄漏信息：

```
1. Detected memory leaks!
```

```

2. Dumping objects ->
3. C:\GitHub\json-tutorial\tutorial03_answer\leptjson.c(212) : {79}
   normal block at 0x013D9868, 2 bytes long.
4. Data: <a > 61 00
5. Object dump complete.

```

这正是我们在单元测试中，先设置字符串，然后设布尔值时没释放字符串所分配的内存。比较麻烦的是，它没有显示调用堆栈。从输出信息中

```
... {79} ...
```

我们知道是第 79 次分配的内存做成问题，我们可以加上 `_CrtSetBreakAlloc(79);` 来调试，那么它便会在第 79 次时中断于分配调用的位置，那时候就能从调用堆栈去找出来龙去脉。

1B. Linux/OSX 下的内存泄漏检测方法

在 Linux、OS X 下，我们可以使用 `valgrind` 工具（用 `apt-get install valgrind`、`brew install valgrind`）。我们完全不用修改代码，只要在命令行执行：

```

1. $ valgrind --leak-check=full ./leptjson_test
2. ==22078== Memcheck, a memory error detector
3. ==22078== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward
   et al.
4. ==22078== Using Valgrind-3.11.0 and LibVEX; rerun with -h for
   copyright info
5. ==22078== Command: ./leptjson_test
6. ==22078==
7. --22078-- run: /usr/bin/dsymutil "./leptjson_test"
8. 160/160 (100.00%) passed
9. ==22078==
10. ==22078== HEAP SUMMARY:
11. ==22078==      in use at exit: 27,728 bytes in 209 blocks
12. ==22078==    total heap usage: 301 allocs, 92 frees, 34,966 bytes
   allocated
13. ==22078==
14. ==22078== 2 bytes in 1 blocks are definitely lost in loss record 1

```

```

of 79
15. ==22078==      at 0x100012EBB: malloc (in
    /usr/local/Cellar/valgrind/3.11.0/lib/valgrind/vgpreload_memcheck-
    amd64-darwin.so)
16. ==22078==      by 0x100008F36: lept_set_string (leptjson.c:208)
17. ==22078==      by 0x100008415: test_access_boolean (test.c:187)
18. ==22078==      by 0x100001849: test_parse (test.c:229)
19. ==22078==      by 0x1000017A3: main (test.c:235)
20. ==22078==
21. ...

```

它发现了在 `test_access_boolean()` 中，由 `lept_set_string()` 分配的 2 个字节（`"a"`）泄漏了。

Valgrind 还有很多功能，例如可以发现未初始化变量。我们若在应用程序或测试程序中，忘了调用 `lept_init(&v)`，那么 `v.type` 的值没被初始化，其值是不确定的（indeterministic），一些函数如果读取那个值就会出现问题：

```

1. static void test_access_boolean() {
2.     lept_value v;
3.     /* lept_init(&v); */
4.     lept_set_string(&v, "a", 1);
5.     ...
6. }

```

这种错误有时候测试时能正确运行（刚好 `v.type` 被设为 `0`），使我们误以为程序正确，而在发布后一些机器上却可能崩溃。这种误以为正确的假像是很危险的，我们可利用 valgrind 能自动测出来：

```

1. $ valgrind --leak-check=full ./leptjson_test
2. ...
3. ==22174== Conditional jump or move depends on uninitialised
    value(s)
4. ==22174==      at 0x100008B5D: lept_free (leptjson.c:164)

```

```

5. ==22174== by 0x100008F26: lept_set_string (leptjson.c:207)
6. ==22174== by 0x1000083FE: test_access_boolean (test.c:187)
7. ==22174== by 0x100001839: test_parse (test.c:229)
8. ==22174== by 0x100001793: main (test.c:235)
9. ==22174==

```

它发现 `lept_free()` 中依靠了一个未初始化的值来跳转，就是 `v.type`，而错误是沿自 `test_access_boolean()`。

编写单元测试时，应考虑哪些执行次序会有机会出错，例如内存相关的错误。然后我们可以利用 TDD 的步骤，先令测试失败（以内存工具检测），修正代码，再确认测试是否成功。

2. 转义序列的解析

转义序列的解析很直观，对其他不合法的字符返回

`LEPT_PARSE_INVALID_STRING_ESCAPE`：

```

1. static int lept_parse_string(lept_context* c, lept_value* v) {
2.     /* ... */
3.     for (;;) {
4.         char ch = *p++;
5.         switch (ch) {
6.             /* ... */
7.             case '\\':
8.                 switch (*p++) {
9.                     case '\"': PUTC(c, '\"'); break;
10.                    case '\\': PUTC(c, '\\'); break;
11.                    case '/': PUTC(c, '/') ; break;
12.                    case 'b': PUTC(c, '\b'); break;
13.                    case 'f': PUTC(c, '\f'); break;
14.                    case 'n': PUTC(c, '\n'); break;
15.                    case 'r': PUTC(c, '\r'); break;
16.                    case 't': PUTC(c, '\t'); break;
17.                    default:
18.                        c->top = head;

```

```

19.                                     return LEPT_PARSE_INVALID_STRING_ESCAPE;
20.                                     }
21.                                     break;
22.                                /* ... */
23.                                }
24.    }
25. }

```

3. 不合法的字符串

上面已解决不合法转义，余下部分的唯一难度，是要从语法中知道哪些是不合法字符：

```
1. unescaped = %x20-21 / %x23-5B / %x5D-10FFFF
```

当中空缺的 %x22 是双引号，%x5C 是反斜线，都已经处理。所以不合法的字符是 %x00 至 %x1F。我们简单地在 default 里处理：

```

1.    /* ... */
2.    default:
3.        if ((unsigned char)ch < 0x20) {
4.            c->top = head;
5.            return LEPT_PARSE_INVALID_STRING_CHAR;
6.        }
7.        PUTC(c, ch);
8.    /* ... */

```

注意到 `char` 带不带符号，是实现定义的。如果编译器定义 `char` 为带符号的话，`(unsigned char)ch >= 0x80` 的字符，都会变成负数，并产生 `LEPT_PARSE_INVALID_STRING_CHAR` 错误。我们现时还没有测试 ASCII 以外的字符，所以有没有转型至不带符号都不影响，但下一单元开始处理 Unicode 的时候就要考虑了。

4. 性能优化的思考

这是本教程第一次的开放式问题，没有标准答案。以下列出一些我想到的。

1. 如果整个字符串都没有转义符，我们不就是把字符复制了两次？第一次是从 `json` 到 `stack`，第二次是从 `stack` 到 `v->u.s.s`。我们可以在 `json` 扫描 `'\0'`、`'\"'` 和 `'\\'` 3 个字符（`ch < 0x20` 还是要检查），直至它们其中一个出现，才开始用现在的解析方法。这样做的话，前半没转义的部分可以只复制一次。缺点是，代码变得复杂一些，我们也不能使用 `lept_set_string()`。
2. 对于扫描没转义部分，我们可考虑用 SIMD 加速，如 [RapidJSON 代码剖析（二）：使用 SSE4.2 优化字符串扫描](#) 的做法。这类底层优化的缺点是不跨平台，需要设置编译选项等。
3. 在 gcc/clang 上使用 `__builtin_expect()` 指令来处理低概率事件，例如需要对每个字符做 `LEPT_PARSE_INVALID_STRING_CHAR` 检测，我们可以假设出现不合法字符是低概率事件，然后用这个指令告之编译器，那么编译器可能生成较快的代码。然而，这类做法明显是不跨编译器，甚至是某个版本后的 gcc 才支持。

5. 总结

本解答篇除了给出一些建议方案，也介绍了内存泄漏的检测方法。

JSON 字符串本身的语法并不复杂，但它需要相关的内存分配与数据结构的设计，还好这些设计都能用于之后的数组和对象类型。下一单元专门针对 Unicode，这部分也是许多 JSON 库没有妥善处理的地方。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（四）：Unicode

- 从零开始的 JSON 库教程（四）：Unicode
 - 1. Unicode
 - 2. 需求
 - 3. UTF-8 编码
 - 4. 实现 `\uXXXX` 解析
 - 5. 总结与练习

从零开始的 JSON 库教程（四）：Unicode

- Milo Yip
- 2016/10/2

本文是《从零开始的 JSON 库教程》的第四个单元。代码位于 [json-tutorial/tutorial04](https://github.com/miloyip/json-tutorial/tutorial04)。

本单元内容：

1. Unicode
2. 需求
3. UTF-8 编码
4. 实现 `\uXXXX` 解析
5. 总结与练习

1. Unicode

在上一个单元，我们已经能解析「一般」的 JSON 字符串，仅仅没有处理 `\uXXXX` 这种转义序列。为了解析这种序列，我们必须了解有关 Unicode 的基本概念。

读者应该知道 ASCII，它是一种字符编码，把 128 个字符映射至整数 0 ~ 127。例如，`1` → 49，`A` → 65，`B` → 66 等等。这种 7-bit 字符编码系统非常简单，在计算机中以一个字节存储一个字符。然而，它仅适合美国英语，甚至一些英语中常用的标点符号、重音符号都不能表示，无法表示各国语言，特别是中日韩语等表意文字。

在 Unicode 出现之前，各地区制定了不同的编码系统，如中文主要用 GB 2312 和大五码、日文主要用 JIS 等。这样会造成很多不便，例如一个文本信息很难混合各种语言的文字。

因此，在上世纪80年代末，Xerox、Apple 等公司开始研究，是否能制定一套多语言的统一编码系统。后来，多个机构成立了 Unicode 联盟，在 1991 年释出 Unicode 1.0，收录了 24 种语言共 7161 个字符。在四分之一个世纪后的 2016年，Unicode 已释出 9.0 版本，收录 135 种语言共 128237 个字符。

这些字符被收录为统一字符集 (Universal Coded Character Set, UCS)，每个字符映射至一个整数码点 (code point)，码点的范围是 0 至 0x10FFFF，码点又通常记作 U+XXXX，当中 XXXX 为 16 进位数字。例如 `劲` → U+52B2、`峰` → U+5CF0。很明显，UCS 中的字符无法像 ASCII 般以一个字节存储。

因此，Unicode 还制定了各种储存码点的方式，这些方式称为 Unicode 转换格式 (Uniform Transformation Format, UTF)。现时流行的 UTF 为 UTF-8、UTF-16 和 UTF-32。每种 UTF 会把一个码点储存为一至多个编码单元 (code unit)。例如 UTF-8 的编码单元是 8 位的字节、UTF-16 为 16 位、UTF-32 为 32 位。除 UTF-32 外，UTF-8 和 UTF-16 都是可变长度编码。

UTF-8 成为现时互联网上最流行的格式，有几个原因：

1. 它采用字节为编码单元，不会有字节序（endianness）的问题。
2. 每个 ASCII 字符只需一个字节去储存。
3. 如果程序原来是以字节方式储存字符，理论上不需要特别改动就能处理 UTF-8 的数据。

2. 需求

由于 UTF-8 的普及性，大部分的 JSON 也通常会以 UTF-8 存储。我们的 JSON 库也会只支持 UTF-8。（RapidJSON 同时支持 UTF-8、UTF-16LE/BE、UTF-32LE/BE、ASCII。）

C 标准库没有关于 Unicode 的处理功能（C++11 有），我们会实现 JSON 库所需的字符编码处理功能。

对于非转义（unescaped）的字符，只要它们不少于 32（0 ~ 31 是不合法的编码单元），我们可以直接复制至结果，这一点我们稍后再说明。我们假设输入是以合法 UTF-8 编码。

而对于 JSON 字符串中的 `\uXXXX` 是以 16 进制表示码点 U+0000 至 U+FFFF，我们需要：

1. 解析 4 位十六进制整数为码点；
2. 由于字符串是以 UTF-8 存储，我们要把这个码点编码成 UTF-8。

同学可能会发现，4 位的 16 进制数字只能表示 0 至 0xFFFF，但之前我们说 UCS 的码点是从 0 至 0x10FFFF，那怎么能表示多出来的码点？

其实，U+0000 至 U+FFFF 这组 Unicode 字符称为基本多文种平面（basic multilingual plane, BMP），还有另外 16 个平面。那么 BMP 以外的字符，JSON 会使用代理对（surrogate

pair) 表示 `\uXXXX\uYYYY`。在 BMP 中，保留了 2048 个代理码点。如果第一个码点是 U+D800 至 U+DBFF，我们便知道它的代码对的高代理项 (high surrogate)，之后应该伴随一个 U+DC00 至 U+DFFF 的低代理项 (low surrogate)。然后，我们用下列公式把代理对 (H, L) 变换成真实的码点：

```
1. codepoint = 0x10000 + (H - 0xD800) × 0x400 + (L - 0xDC00)
```

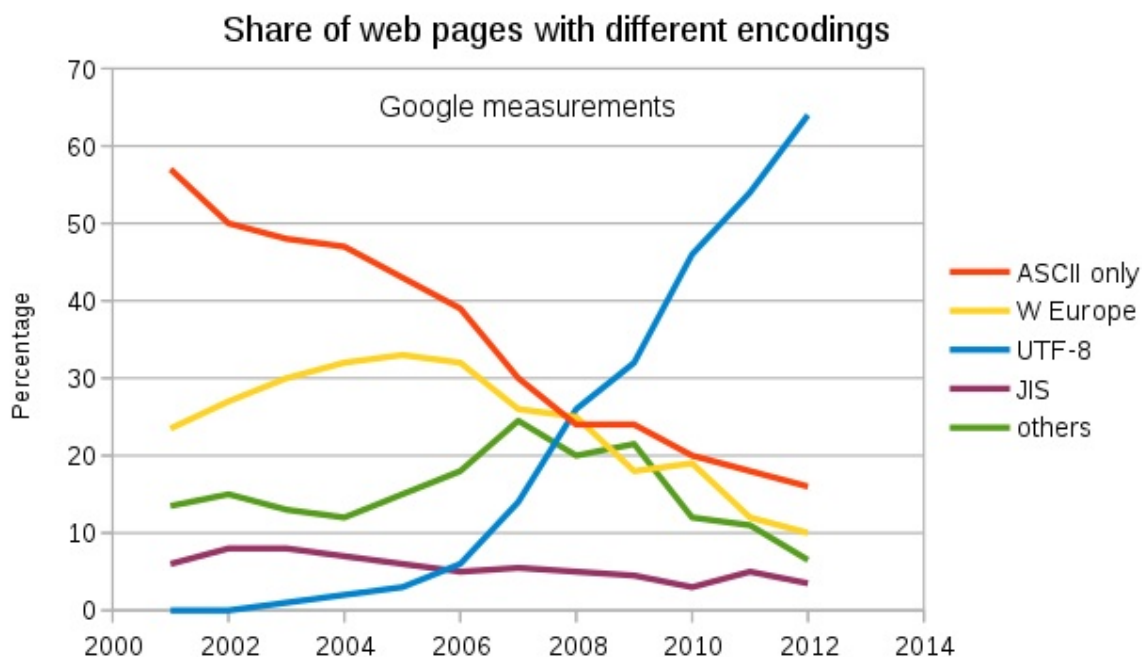
举个例子，高音谱号字符 `?` → U+1D11E 不是 BMP 之内的字符。在 JSON 中可写成转义序列 `\uD834\uDD1E`，我们解析第一个 `\uD834` 得到码点 U+D834，我们发现它是 U+D800 至 U+DBFF 内的码点，所以它是高代理项。然后我们解析下一个转义序列 `\uDD1E` 得到码点 U+DD1E，它在 U+DC00 至 U+DFFF 之内，是合法的低代理项。我们计算其码点：

```
1. H = 0xD834, L = 0xDD1E
2. codepoint = 0x10000 + (H - 0xD800) × 0x400 + (L - 0xDC00)
3.           = 0x10000 + (0xD834 - 0xD800) × 0x400 + (0xDD1E - 0xDC00)
4.           = 0x10000 + 0x34 × 0x400 + 0x11E
5.           = 0x10000 + 0xD000 + 0x11E
6.           = 0x1D11E
```

这样就得出这转义序列的码点，然后我们再把它编码成 UTF-8。如果只有高代理项而欠缺低代理项，或是低代理项不在合法码点范围，我们都返回 `LEPT_PARSE_INVALID_UNICODE_SURROGATE` 错误。如果 `\u` 后不是 4 位十六进位数字，则返回 `LEPT_PARSE_INVALID_UNICODE_HEX` 错误。

3. UTF-8 编码

UTF-8 在网页上的使用率势无可挡：



（图片来自 [Wikipedia Common](#)，数据来自 Google 对网页字符编码的统计。）


由于我们的 JSON 库也只支持 UTF-8，我们需要把码点编码成 UTF-8。这里简单介绍一下 UTF-8 的编码方式。

UTF-8 的编码单元是 8 位字节，每个码点编码成 1 至 4 个字节。它的编码方式很简单，按照码点的范围，把码点的二进位分拆成 1 至最多 4 个字节：

码点范围	码点位数	字节1	字节2	字节3	字节4
U+0000 ~ U+007F	7	0xxxxxxx			
U+0080 ~ U+07FF	11	110xxxxx	10xxxxxx		
U+0800 ~ U+FFFF	16	1110xxxx	10xxxxxx	10xxxxxx	
U+10000 ~ U+10FFFF	21	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

这个编码方法的好处之一是，码点范围 U+0000 ~ U+007F 编码为一个字节，与 ASCII 编码兼容。这范围的 Unicode 码点也是和

ASCII 字符相同的。因此，一个 ASCII 文本也是一个 UTF-8 文本。

我们举一个例子解析多字节的情况，欧元符号  → U+20AC：

1. U+20AC 在 U+0800 ~ U+FFFF 的范围内，应编码成 3 个字节。
2. U+20AC 的二进位为 10000010101100
3. 3 个字节的情况我们要 16 位的码点，所以在前面补两个 0，成为 0010000010101100
4. 按上表把二进位分成 3 组：0010, 000010, 101100
5. 加上每个字节的前缀：11100010, 10000010, 10101100
6. 用十六进位表示即：0xE2, 0x82, 0xAC

对于这例子的范围，对应的 C 代码是这样的：

```
1. if (u >= 0x0800 && u <= 0xFFFF) {
2.     OutputByte(0xE0 | ((u >> 12) & 0xFF)); /* 0xE0 = 11100000 */
3.     OutputByte(0x80 | ((u >> 6) & 0x3F)); /* 0x80 = 10000000 */
4.     OutputByte(0x80 | (u & 0x3F)); /* 0x3F = 00111111 */
5. }
```

UTF-8 的解码稍复杂一点，但我们的 JSON 库不会校验 JSON 文本是否符合 UTF-8，所以这里也不展开了。

4. 实现 解析

我们只需要在其它转义符的处理中加入对  的处理：

```
1. static int lept_parse_string(lept_context* c, lept_value* v) {
2.     unsigned u;
3.     /* ... */
4.     for (;;) {
```

```

5.         char ch = *p++;
6.         switch (ch) {
7.             /* ... */
8.             case '\\':
9.                 switch (*p++) {
10.                    /* ... */
11.                    case 'u':
12.                        if (!(p = lept_parse_hex4(p, &u)))
13.                            STRING_ERROR(LEPT_PARSE_INVALID_UNICODE_HEX);
14.                        /* \TODO surrogate handling */
15.                        lept_encode_utf8(c, u);
16.                        break;
17.                    /* ... */
18.                }
19.            /* ... */
20.        }
21.    }
22. }

```

上面代码的过程很简单，遇到 `\u` 转义时，调用

`lept_parse_hex4()` 解析 4 位十六进数字，存储为码点 `u`。这个函数在成功时返回解析后的文本指针，失败返回 `NULL`。如果失败，就返回 `LEPT_PARSE_INVALID_UNICODE_HEX` 错误。最后，把码点编码成 UTF-8，写进缓冲区。这里没有处理代理对，留作练习。

顺带一提，我为 `lept_parse_string()` 做了个简单的重构，把返回错误的处理抽取为宏：

```

1. #define STRING_ERROR(ret) do { c->top = head; return ret; }
   while(0)

```

5. 总结与练习

本单元介绍了 Unicode 的基本知识，同学应该了解到一些常用的

Unicode 术语，如码点、编码单元、UTF-8、代理对等。这次的练习代码只有个空壳，要由同学填充。完成后应该能通过所有单元测试，届时我们的 JSON 字符串解析就完全符合标准了。

1. 实现 `lept_parse_hex4()`，不合法的十六进位数返回

`LEPT_PARSE_INVALID_UNICODE_HEX`。

2. 按第 3 节谈到的 UTF-8 编码原理，实现 `lept_encode_utf8()`。这函数假设码点在正确范围 `U+0000 ~ U+10FFFF`（用断言检测）。

3. 加入对代理对的处理，不正确的代理对范围要返回

`LEPT_PARSE_INVALID_UNICODE_SURROGATE` 错误。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（四）：Unicode 解答篇

- 从零开始的 JSON 库教程（四）：Unicode 解答篇

- 1. 实现 `lept_parse_hex4()`
- 2. 实现 `lept_encode_utf8()`
- 3. 代理对的处理
- 4. 总结

从零开始的 JSON 库教程（四）：Unicode 解答篇

- Milo Yip
- 2016/10/6

本文是《从零开始的 JSON 库教程》的第四个单元解答篇。解答代码位于 [json-tutorial/tutorial04_answer](https://github.com/miloyip/json-tutorial/blob/master/tutorial04_answer.c)。

1. 实现 `lept_parse_hex4()`

这个函数只是读 4 位 16 进制数字，可以简单地自行实现：

```
1. static const char* lept_parse_hex4(const char* p, unsigned* u) {
2.     int i;
3.     *u = 0;
4.     for (i = 0; i < 4; i++) {
5.         char ch = *p++;
6.         *u <<= 4;
7.         if (ch >= '0' && ch <= '9') *u |= ch - '0';
8.         else if (ch >= 'A' && ch <= 'F') *u |= ch - ('A' - 10);
9.         else if (ch >= 'a' && ch <= 'f') *u |= ch - ('a' - 10);
10.        else return NULL;
11.    }
12.    return p;
```

```
13. }
```

可能有同学想到用标准库的 `strtoul()`，因为它也能解析 16 进制数字，那么可以简短的写成：

```
1. static const char* lept_parse_hex4(const char* p, unsigned* u) {
2.     char* end;
3.     *u = (unsigned)strtoul(p, &end, 16);
4.     return end == p + 4 ? end : NULL;
5. }
```

但这个实现会错误地接受 `"\u 123"` 这种不合法的 JSON，因为 `strtoul()` 会跳过开始的空白。要解决的话，还需要检测第一个字符是否 `[0-9A-Fa-f]`，或者 `!isspace(*p)`。但为了 `strtoul()` 做多余的检测，而且自行实现也很简单，我个人会选择首个方案。（前两个单元用 `strtod()` 就没办法，因为它的实现要复杂得多。）

2. 实现 `lept_encode_utf8()`

这个函数只需要根据那个 UTF-8 编码表就可以实现：

```
1. static void lept_encode_utf8(lept_context* c, unsigned u) {
2.     if (u <= 0x7F)
3.         PUTC(c, u & 0xFF);
4.     else if (u <= 0x7FF) {
5.         PUTC(c, 0xC0 | ((u >> 6) & 0xFF));
6.         PUTC(c, 0x80 | (u & 0x3F));
7.     }
8.     else if (u <= 0xFFFF) {
9.         PUTC(c, 0xE0 | ((u >> 12) & 0xFF));
10.        PUTC(c, 0x80 | ((u >> 6) & 0x3F));
11.        PUTC(c, 0x80 | (u & 0x3F));
12.    }
13.    else {
14.        assert(u <= 0x10FFFF);
```

```

15.         PUTC(c, 0xF0 | ((u >> 18) & 0xFF));
16.         PUTC(c, 0x80 | ((u >> 12) & 0x3F));
17.         PUTC(c, 0x80 | ((u >> 6) & 0x3F));
18.         PUTC(c, 0x80 | (u & 0x3F));
19.     }
20. }

```

有同学可能觉得奇怪，最终也是写进一个 `char`，为什么要做 `x & 0xFF` 这种操作呢？这是因为 `u` 是 `unsigned` 类型，一些编译器可能会警告这个转型可能会截断数据。但实际上，配合了范围的检测然后右移之后，可以保证写入的是 0~255 内的值。为了避免一些编译器的警告误判，我们加上 `x & 0xFF`。一般来说，编译器在优化之后，这与操作是会被消去的，不会影响性能。

其实超过 1 个字符输出时，可以只调用 1 次

`lept_context_push()`。这里全用 `PUTC()` 只是为了代码看上去简单一点。

3. 代理对的处理

遇到高代理项，就需要把低代理项 `\uxxxx` 也解析进来，然后用这两个项去计算出码点：

```

1. case 'u':
2.     if (!(p = lept_parse_hex4(p, &u)))
3.         STRING_ERROR(LEPT_PARSE_INVALID_UNICODE_HEX);
4.     if (u >= 0xD800 && u <= 0xDBFF) { /* surrogate pair */
5.         if (*p++ != '\\')
6.             STRING_ERROR(LEPT_PARSE_INVALID_UNICODE_SURROGATE);
7.         if (*p++ != 'u')
8.             STRING_ERROR(LEPT_PARSE_INVALID_UNICODE_SURROGATE);
9.         if (!(p = lept_parse_hex4(p, &u2)))
10.            STRING_ERROR(LEPT_PARSE_INVALID_UNICODE_HEX);
11.         if (u2 < 0xDC00 || u2 > 0xDFFF)

```

```
12.         STRING_ERROR(LEPT_PARSE_INVALID_UNICODE_SURROGATE);
13.         u = (((u - 0xD800) << 10) | (u2 - 0xDC00)) + 0x10000;
14.     }
15.     lept_encode_utf8(c, u);
16.     break;
```

4. 总结

JSON 的字符串解析终于完成了。但更重要的是，同学通过教程和练习后，应该对于 Unicode 和 UTF-8 编码有基本了解。使用 Unicode 标准去处理文本数据已是世界潮流。虽然 C11/C++11 引入了 Unicode 字符串字面量及少量函数，但仍然有很多不足，一般需要借助第三方库。

我们在稍后的单元还要处理生成时的 Unicode 问题，接下来我们要继续讨论数组和对象的解析。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（五）：解析数组

- [从零开始的 JSON 库教程（五）：解析数组](#)
 - [1. JSON 数组](#)
 - [2. 数据结构](#)
 - [3. 解析过程](#)
 - [4. 实现](#)
 - [5. 总结与练习](#)

从零开始的 JSON 库教程（五）：解析数组

- Milo Yip
- 2016/10/7

本文是《[从零开始的 JSON 库教程](#)》的第五个单元。代码位于 [json-tutorial/tutorial05](#)。

本单元内容：

- [1. JSON 数组](#)
- [2. 数据结构](#)
- [3. 解析过程](#)
- [4. 实现](#)
- [5. 总结与练习](#)

1. JSON 数组

从零到这第五单元，我们终于要解析一个 JSON 的复合数据类型了。一个 JSON 数组可以包含零至多个元素，而这些元素也可以是数组类型。换句话说，我们可以表示嵌套（nested）的数据结构。先来看看

JSON 数组的语法：

```
1. array = %x5B ws [ value *( ws %x2C ws value ) ] ws %x5D
```

当中，`%x5B` 是左中括号 `[`，`%x2C` 是逗号 `,`，`%x5D` 是右中括号 `]`，`ws` 是空白字符。一个数组可以包含零至多个值，以逗号分隔，例如 `[]`、`[1,2,true]`、`[[1,2],[3,4],"abc"]` 都是合法的数组。但注意 JSON 不接受末端额外的逗号，例如 `[1,2,]` 是不合法的（许多编程语言如 C/C++、Javascript、Java、C# 都容许数组初始值包含末端逗号）。

JSON 数组的语法很简单，实现的难点不在语法上，而是怎样管理内存。

2. 数据结构

首先，我们需要设计存储 JSON 数组类型的数据结构。

JSON 数组存储零至多个元素，最简单就是使用 C 语言的数组。数组最大的好处是能以 $O(1)$ 用索引访问任意元素，次要好处是内存布局紧凑，省内存之余还有高缓存一致性（cache coherence）。但数组的缺点是不能快速插入元素，而且我们在解析 JSON 数组的时候，还不知道应该分配多大的数组才合适。

另一个选择是链表（linked list），它的最大优点是可快速地插入元素（开端、末端或中间），但需要以 $O(n)$ 时间去经索引取得内容。如果我们只需顺序遍历，那么是没有问题的。还有一个小缺点，就是相对数组而言，链表在存储每个元素时有额外内存开销（存储下一节点的指针），而且遍历时元素所在的内存可能不连续，令缓存不命中（cache miss）的机会上升。

我见过一些 JSON 库选择了链表，而这里则选择了数组。我们将会通过之前在解析字符串时实现的堆栈，来解决解析 JSON 数组时未知数组大小的问题。

决定之后，我们在 `lept_value` 的 `union` 中加入数组的结构：

```
1. typedef struct lept_value lept_value;
2.
3. struct lept_value {
4.     union {
5.         struct { lept_value* e; size_t size; }a; /* array */
6.         struct { char* s; size_t len; }s;
7.         double n;
8.     }u;
9.     lept_type type;
10. };
```

由于 `lept_value` 内使用了自身类型的指针，我们必须前向声明（forward declare）此类型。

另外，注意这里 `size` 是元素的个数，不是字节单位。我们增加两个 API 去访问 JSON 数组类型的值：

```
1. size_t lept_get_array_size(const lept_value* v) {
2.     assert(v != NULL && v->type == LEPT_ARRAY);
3.     return v->u.a.size;
4. }
5.
6. lept_value* lept_get_array_element(const lept_value* v, size_t
    index) {
7.     assert(v != NULL && v->type == LEPT_ARRAY);
8.     assert(index < v->u.a.size);
9.     return &v->u.a.e[index];
10. }
```

暂时我们不考虑增删数组元素，这些功能留待第八单元讨论。

然后，我们写一个单元测试去试用这些 API（练习需要更多测试）。

```

1. #if defined(_MSC_VER)
2. #define EXPECT_EQ_SIZE_T(expect, actual) EXPECT_EQ_BASE((expect) ==
   (actual), (size_t)expect, (size_t)actual, "%Iu")
3. #else
4. #define EXPECT_EQ_SIZE_T(expect, actual) EXPECT_EQ_BASE((expect) ==
   (actual), (size_t)expect, (size_t)actual, "%zu")
5. #endif
6.
7. static void test_parse_array() {
8.     lept_value v;
9.
10.    lept_init(&v);
11.    EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, "[ ]"));
12.    EXPECT_EQ_INT(LEPT_ARRAY, lept_get_type(&v));
13.    EXPECT_EQ_SIZE_T(0, lept_get_array_size(&v));
14.    lept_free(&v);
15. }
```

在之前的单元中，作者已多次重申，C 语言的数组大小应该使用

`size_t` 类型。因为我们要验证 `lept_get_array_size()` 返回值是否

正确，所以再为单元测试框架添加一个宏 `EXPECT_EQ_SIZE_T`。麻烦之

处在于，ANSI C (C89) 并没有的 `size_t` 打印方法，在 C99 则

加入了 `"%zu"`，但 VS2015 中才有，之前的 VC 版本使用非标准的

`"%Iu"`。因此，上面的代码使用条件编译去区分 VC 和其他编译器。

虽然这部分不跨平台也不是 ANSI C 标准，但它只在测试程序中，不太影响程序库的跨平台性。

3. 解析过程

我们在解析 JSON 字符串时，因为在开始时不能知道字符串的长度，而又需要进行转义，所以需要有一个临时缓冲区去存储解析后的结果。我

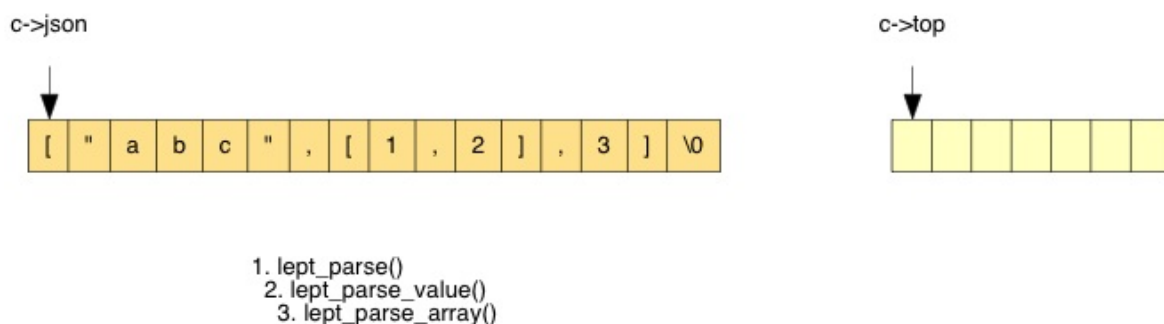
们为此实现了一个动态增长的堆栈，可以不断压入字符，最后一次性把整个字符串弹出，复制至新分配的内存之中。

对于 JSON 数组，我们也可以用相同的方法，而且，我们可以用同一个堆栈！我们只需要把每个解析好的元素压入堆栈，解析到数组结束时，再一次性把所有元素弹出，复制至新分配的内存之中。

但和字符串有点不一样，如果把 JSON 当作一棵树的数据结构，JSON 字符串是叶节点，而 JSON 数组是中间节点。在叶节点的解析函数中，我们怎样使用那个堆栈也可以，只要最后还原就好了。但对于数组这样的中间节点，共用这个堆栈没问题么？

答案是：只要在解析函数结束时还原堆栈的状态，就没有问题。为了直观地了解这个解析过程，我们用连环图去展示 `["abc", [1, 2], 3]` 的解析过程。

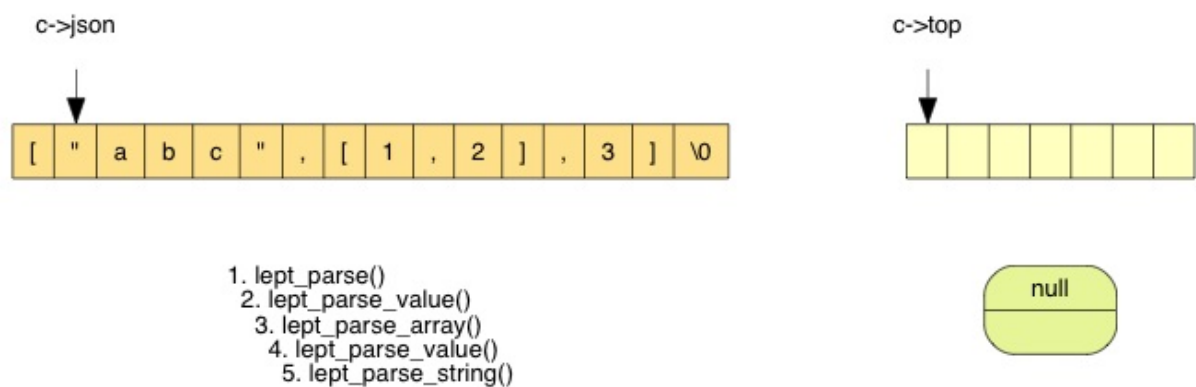
首先，我们遇到 `[`，进入 `lept_parse_array()`：



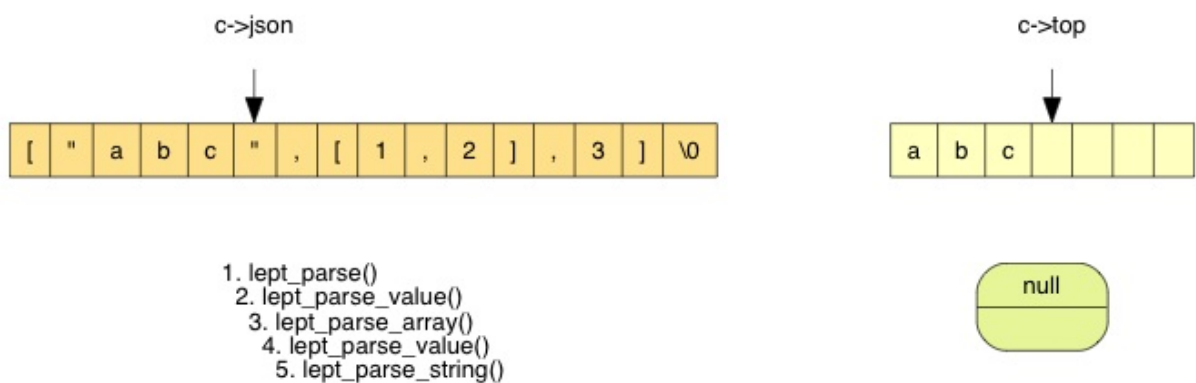
生成一个临时的 `lept_value`，用于存储之后的元素。我们再调用

`lept_parse_value()` 去解析这个元素值，因为遇到 `"` 进入

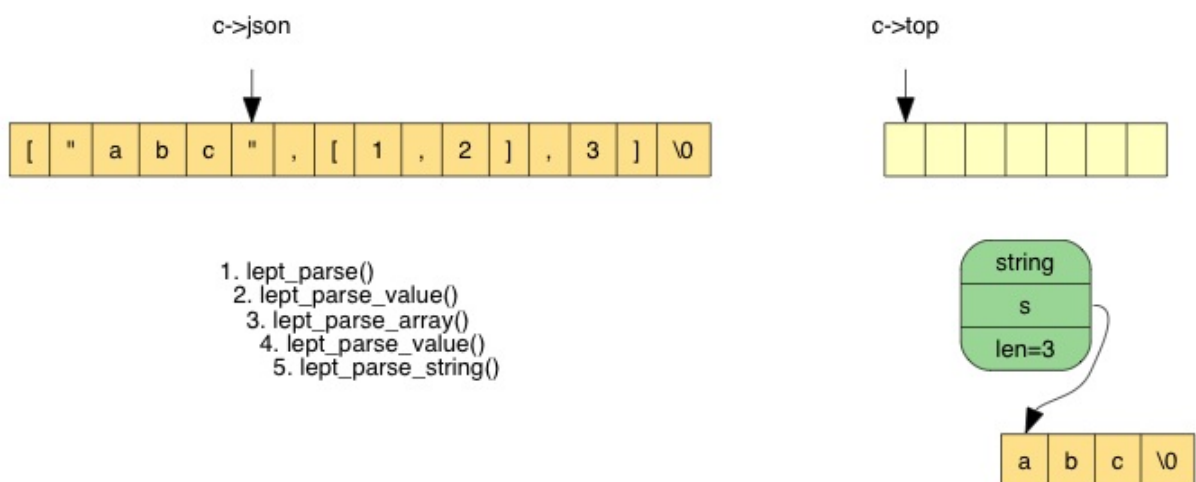
`lept_parse_string()`：



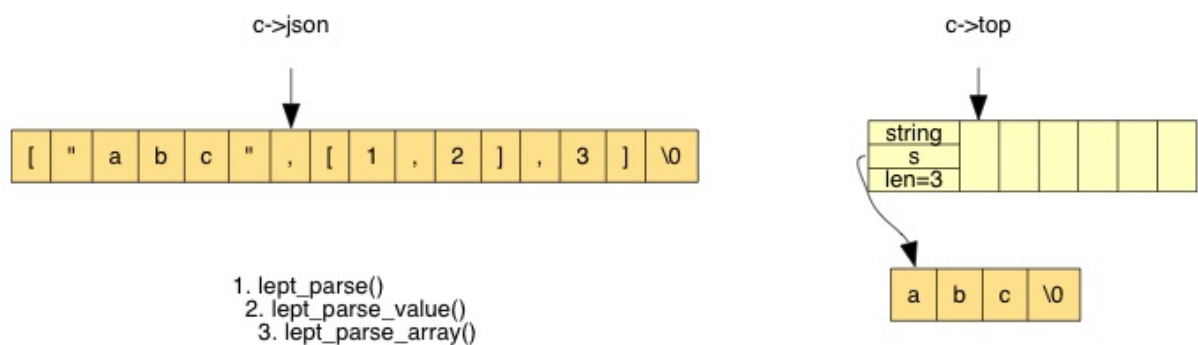
在 `lept_parse_string()` 中，不断解析字符直至遇到 `"`，过程中把每个字符压栈：



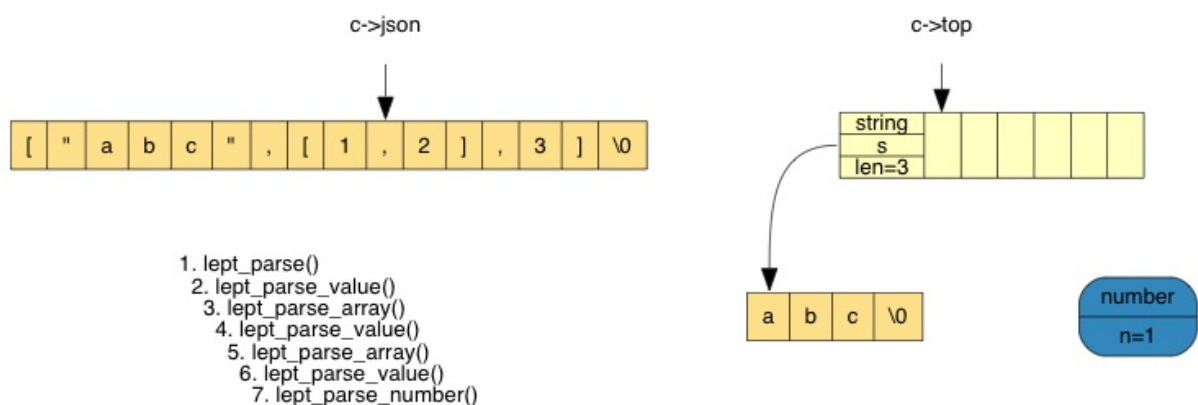
最后在 `lept_parse_string()` 中，把栈上 3 个字符弹出，分配内存，生成字符串值：



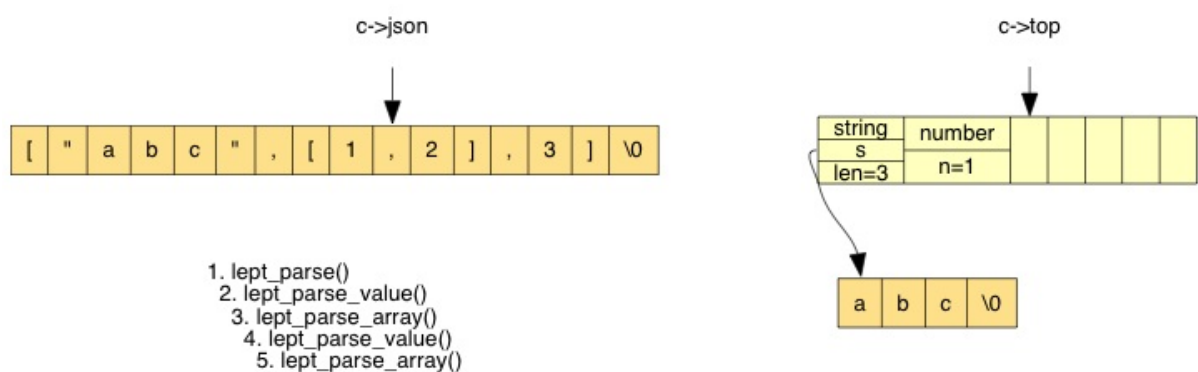
返回上一层 `lept_parse_array()`，把临时元素压栈：



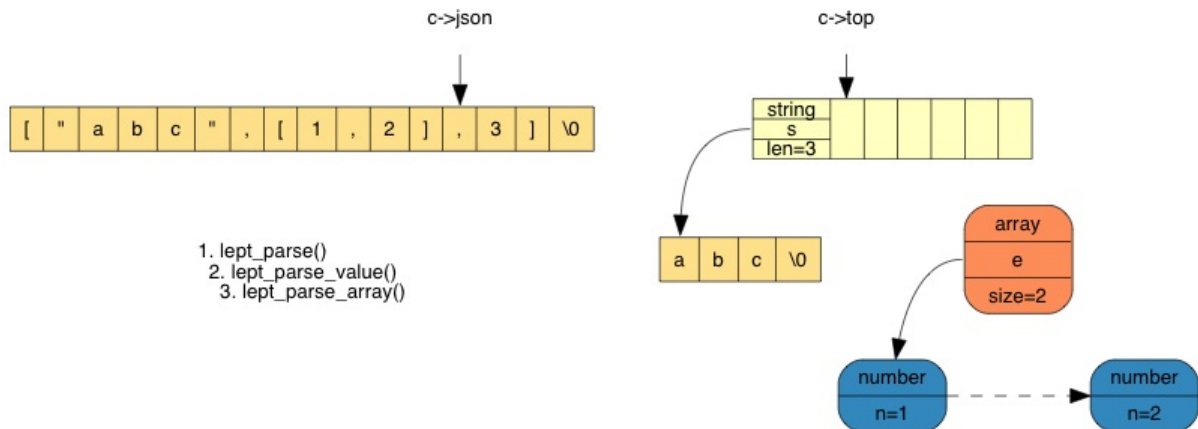
然后我们再遇到 `[`，进入另一个 `lept_parse_array()`。它发现第一个元素是数字类型，所以调用 `lept_parse_number()`，生成一个临时的元素值：



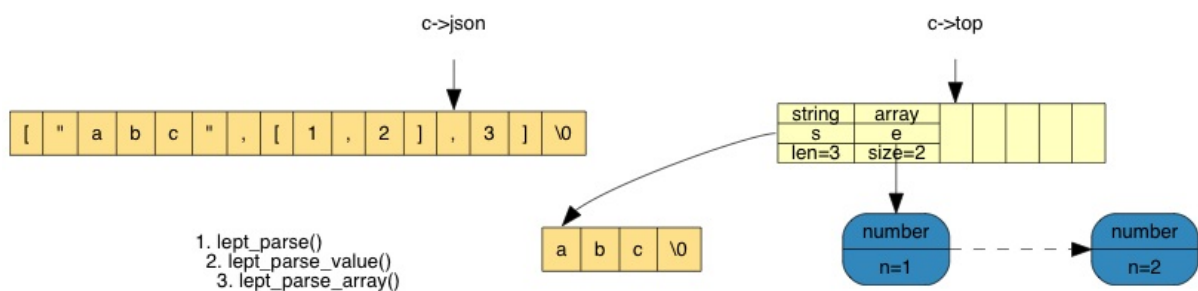
之后把该临时的元素值压栈：



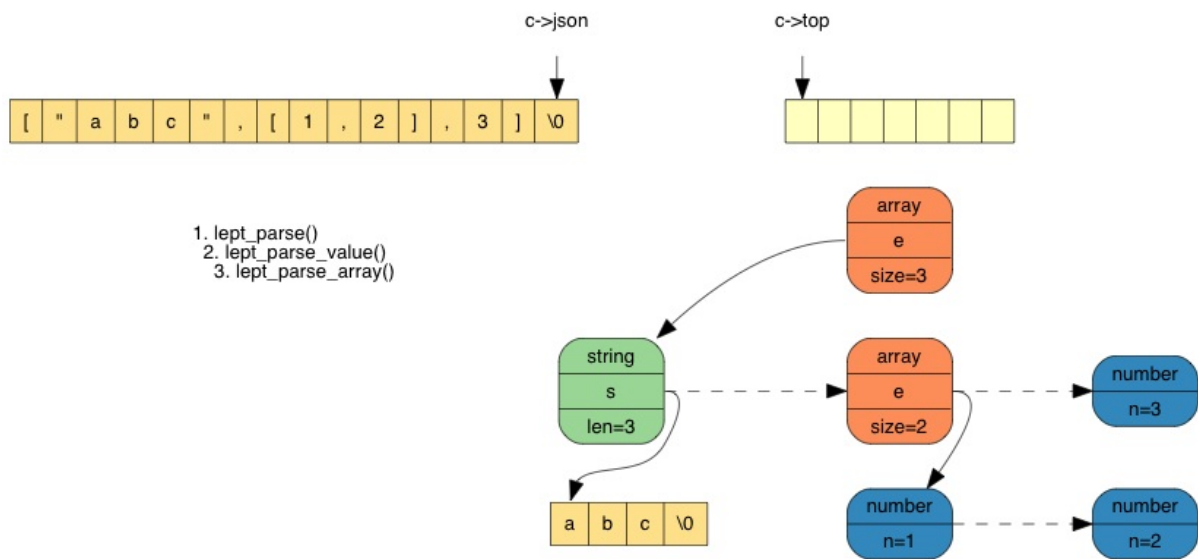
接着再解析第二个元素。我们遇到了 `]`，从栈上弹出 2 个元素，分配内存，生成数组（虚线代表是连续的内存）：



那个数组是上层数组的元素，我们把它压栈。现时栈内已有两个元素，我们再继续解析下一个元素：



最后，遇到了 `]`，可以弹出栈内 3 个元素，分配内存，生成数组：



4. 实现

经过这个详细的图解，实现 `lept_parse_array()` 应该没有难度。以下是半制成品：

```

1. static int lept_parse_value(lept_context* c, lept_value* v); /* 前向
   声明 */
2.
3. static int lept_parse_array(lept_context* c, lept_value* v) {
4.     size_t size = 0;
5.     int ret;
6.     EXPECT(c, '[');
7.     if (*c->json == ']') {
8.         c->json++;
9.         v->type = LEPT_ARRAY;
10.        v->u.a.size = 0;
11.        v->u.a.e = NULL;
12.        return LEPT_PARSE_OK;
13.    }
14.    for (;;) {
15.        lept_value e;
16.        lept_init(&e);
17.        if ((ret = lept_parse_value(c, &e)) != LEPT_PARSE_OK)

```

```

18.         return ret;
19.         memcpy(lept_context_push(c, sizeof(lept_value)), &e,
    sizeof(lept_value));
20.         size++;
21.         if (*c->json == ',')
22.             c->json++;
23.         else if (*c->json == ']') {
24.             c->json++;
25.             v->type = LEPT_ARRAY;
26.             v->u.a.size = size;
27.             size *= sizeof(lept_value);
28.             memcpy(v->u.a.e = (lept_value*)malloc(size),
    lept_context_pop(c, size), size);
29.             return LEPT_PARSE_OK;
30.         }
31.         else
32.             return LEPT_PARSE_MISS_COMMA_OR_SQUARE_BRACKET;
33.     }
34. }
35.
36. static int lept_parse_value(lept_context* c, lept_value* v) {
37.     switch (*c->json) {
38.         /* ... */
39.         case '[': return lept_parse_array(c, v);
40.     }
41. }

```

简单说明的话，就是在循环中建立一个临时值（`lept_value e`），然后调用 `lept_parse_value()` 去把元素解析至这个临时值，完成后把临时值压栈。当遇到 `]`，把栈内的元素弹出，分配内存，生成数组值。

注意到，`lept_parse_value()` 会调用 `lept_parse_array()`，而 `lept_parse_array()` 又会调用 `lept_parse_value()`，这是互相引用，所以必须要加入函数前向声明。

最后，我想告诉同学，实现这个函数时，我曾经制造一个不明显的 bug。这个函数有两个 `memcpy()`，第一个「似乎」是可以避免的，先压栈取得元素的指针，给 `lept_parse_value`：

```

1.     for (;;) {
2.         /* bug! */
3.         lept_value* e = lept_context_push(c, sizeof(lept_value));
4.         lept_init(e);
5.         size++;
6.         if ((ret = lept_parse_value(c, e)) != LEPT_PARSE_OK)
7.             return ret;
8.         /* ... */
9.     }

```

这种写法为什么会有 bug？这是第 5 条练习题。

5. 总结与练习

1. 编写 `test_parse_array()` 单元测试，解析以下 2 个 JSON。由于数组是复合的类型，不能使用一个宏去测试结果，请使用各个 API 检查解析后的内容。

```

1. [ null , false , true , 123 , "abc" ]
2. [ [ ] , [ 0 ] , [ 0 , 1 ] , [ 0 , 1 , 2 ] ]

```

1. 现时的测试结果应该是失败的，因为 `lept_parse_array()` 里没有处理空白字符，加进合适的 `lept_parse_whitespace()` 令测试通过。
2. 使用[第三单元解答篇](#)介绍的检测内存泄漏工具，会发现测试中有内存泄漏。很明显在 `lept_parse_array()` 中使用到 `malloc()` 分配内存，但却没有对应的 `free()`。应该在哪里释放内存？修改代码，使工具不再检测到相关的内存泄漏。

3. 开启 `test.c` 中两处被 `#if 0 ... #endif` 关闭的测试，本来 `test_parse_array()` 已经能处理这些测试。然而，运行时会发现 `Assertion failed: (c.top == 0)` 断言失败。这是由于，当错误发生时，仍然有一些临时值在堆栈里，既没有放进数组，也没有被释放。修改 `test_parse_array()`，当遇到错误时，从堆栈中弹出并释放那些临时值，然后才返回错误码。

4. 第 4 节那段代码为什么会有 bug？

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（五）：解析数组解答篇

- 从零开始的 JSON 库教程（五）：解析数组解答篇
 - 1. 编写 `test_parse_array()` 单元测试
 - 2. 解析空白字符
 - 3. 内存泄漏
 - 4. 解析错误时的内存处理
 - 5. bug 的解释
 - 6. 总结

从零开始的 JSON 库教程（五）：解析数组解答篇

- Milo Yip
- 2016/10/13

本文是《从零开始的 JSON 库教程》的第五个单元解答篇。解答代码位于 [json-tutorial/tutorial05_answer](https://github.com/miloyip/json-tutorial/blob/master/tutorial05_answer.c)。

1. 编写 `test_parse_array()` 单元测试

这个练习纯粹为了熟习数组的访问 API。新增的第一个 JSON 只需平凡的检测。第二个 JSON 有特定模式，第 i 个子数组的长度为 i ，每个子数组的第 j 个元素是数字值 j ，所以可用两层 for 循环测试。

```
1. static void test_parse_array() {
2.     size_t i, j;
3.     lept_value v;
4.
5.     /* ... */
```

```

6.
7.     lept_init(&v);
8.     EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, "[ null , false ,
true , 123 , \"abc\" ]"));
9.     EXPECT_EQ_INT(LEPT_ARRAY, lept_get_type(&v));
10.    EXPECT_EQ_SIZE_T(5, lept_get_array_size(&v));
11.    EXPECT_EQ_INT(LEPT_NULL,
lept_get_type(lept_get_array_element(&v, 0)));
12.    EXPECT_EQ_INT(LEPT_FALSE,
lept_get_type(lept_get_array_element(&v, 1)));
13.    EXPECT_EQ_INT(LEPT_TRUE,
lept_get_type(lept_get_array_element(&v, 2)));
14.    EXPECT_EQ_INT(LEPT_NUMBER,
lept_get_type(lept_get_array_element(&v, 3)));
15.    EXPECT_EQ_INT(LEPT_STRING,
lept_get_type(lept_get_array_element(&v, 4)));
16.    EXPECT_EQ_DOUBLE(123.0,
lept_get_number(lept_get_array_element(&v, 3)));
17.    EXPECT_EQ_STRING("abc",
lept_get_string(lept_get_array_element(&v, 4)),
lept_get_string_length(lept_get_array_element(&v, 4)));
18.    lept_free(&v);
19.
20.    lept_init(&v);
21.    EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, "[ [ ] , [ 0 ] , [
0 , 1 ] , [ 0 , 1 , 2 ] ]"));
22.    EXPECT_EQ_INT(LEPT_ARRAY, lept_get_type(&v));
23.    EXPECT_EQ_SIZE_T(4, lept_get_array_size(&v));
24.    for (i = 0; i < 4; i++) {
25.        lept_value* a = lept_get_array_element(&v, i);
26.        EXPECT_EQ_INT(LEPT_ARRAY, lept_get_type(a));
27.        EXPECT_EQ_SIZE_T(i, lept_get_array_size(a));
28.        for (j = 0; j < i; j++) {
29.            lept_value* e = lept_get_array_element(a, j);
30.            EXPECT_EQ_INT(LEPT_NUMBER, lept_get_type(e));
31.            EXPECT_EQ_DOUBLE((double)j, lept_get_number(e));
32.        }
33.    }

```

```

34.     lept_free(&v);
35. }

```

2. 解析空白字符

按现时的 `lept_parse_array()` 的编写方式，需要加入 3 个 `lept_parse_whitespace()` 调用，分别是解析 `[` 之后，元素之后，以及 `,` 之后：

```

1. static int lept_parse_array(lept_context* c, lept_value* v) {
2.     /* ... */
3.     EXPECT(c, '[');
4.     lept_parse_whitespace(c);
5.     /* ... */
6.     for (;;) {
7.         /* ... */
8.         if ((ret = lept_parse_value(c, &e)) != LEPT_PARSE_OK)
9.             return ret;
10.        /* ... */
11.        lept_parse_whitespace(c);
12.        if (*c->json == ',') {
13.            c->json++;
14.            lept_parse_whitespace(c);
15.        }
16.        /* ... */
17.    }
18. }

```

3. 内存泄漏

成功测试那 3 个 JSON 后，使用内存泄漏检测工具会发现

`lept_parse_array()` 用 `malloc()` 分配的内存没有被释放：

```

1. ==154== 124 (120 direct, 4 indirect) bytes in 1 blocks are
   definitely lost in loss record 2 of 4

```

```

2. ==154==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
3. ==154==    by 0x409D82: lept_parse_array (in /json-
tutorial/tutorial05/build/leptjson_test)
4. ==154==    by 0x409E91: lept_parse_value (in /json-
tutorial/tutorial05/build/leptjson_test)
5. ==154==    by 0x409F14: lept_parse (in /json-
tutorial/tutorial05/build/leptjson_test)
6. ==154==    by 0x405261: test_parse_array (in /json-
tutorial/tutorial05/build/leptjson_test)
7. ==154==    by 0x408C72: test_parse (in /json-
tutorial/tutorial05/build/leptjson_test)
8. ==154==    by 0x40916A: main (in /json-
tutorial/tutorial05/build/leptjson_test)
9. ==154==
10. ==154== 240 (96 direct, 144 indirect) bytes in 1 blocks are
definitely lost in loss record 4 of 4
11. ==154==    at 0x4C28C20: malloc (vg_replace_malloc.c:296)
12. ==154==    by 0x409D82: lept_parse_array (in /json-
tutorial/tutorial05/build/leptjson_test)
13. ==154==    by 0x409E91: lept_parse_value (in /json-
tutorial/tutorial05/build/leptjson_test)
14. ==154==    by 0x409F14: lept_parse (in /json-
tutorial/tutorial05/build/leptjson_test)
15. ==154==    by 0x40582C: test_parse_array (in /json-
tutorial/tutorial05/build/leptjson_test)
16. ==154==    by 0x408C72: test_parse (in /json-
tutorial/tutorial05/build/leptjson_test)
17. ==154==    by 0x40916A: main (in /json-
tutorial/tutorial05/build/leptjson_test)

```

很明显，有 `malloc()` 就要有对应的 `free()`。正确的释放位置应该放置在 `lept_free()`，当值被释放时，该值拥有的内存也在那里释放。之前字符串的释放也是放在这里：

```

1. void lept_free(lept_value* v) {
2.     assert(v != NULL);
3.     if (v->type == LEPT_STRING)

```

```

4.         free(v->u.s.s);
5.         v->type = LEPT_NULL;
6.     }

```

但对于数组，我们应该先把数组内的元素通过递归调用
释放，然后才释放本身的

`lept_free()`

`v->u.a.e` :

```

1. void lept_free(lept_value* v) {
2.     size_t i;
3.     assert(v != NULL);
4.     switch (v->type) {
5.         case LEPT_STRING:
6.             free(v->u.s.s);
7.             break;
8.         case LEPT_ARRAY:
9.             for (i = 0; i < v->u.a.size; i++)
10.                 lept_free(&v->u.a.e[i]);
11.             free(v->u.a.e);
12.             break;
13.         default: break;
14.     }
15.     v->type = LEPT_NULL;
16. }

```

修改之后，再运行内存泄漏检测工具，确保问题已被修正。

4. 解析错误时的内存处理

遇到解析错误时，我们可能在之前已压入了一些值在自定义堆栈上。如果没有处理，最后会在 `lept_parse()` 中发现堆栈上还有一些值，做成断言失败。所以，遇到解析错误时，我们必须弹出并释放那些值。

在 `lept_parse_array` 中，原本遇到解析失败时，会直接返回错误码。

我们把它改为 `break` 离开循环，在循环结束后的地方用

`lept_free()` 释放从堆栈弹出的值，然后才返回错误码：

```

1. static int lept_parse_array(lept_context* c, lept_value* v) {
2.     /* ... */
3.     for (;;) {
4.         /* ... */
5.         if ((ret = lept_parse_value(c, &e)) != LEPT_PARSE_OK)
6.             break;
7.         /* ... */
8.         if (*c->json == ',') {
9.             /* ... */
10.        }
11.        else if (*c->json == ']') {
12.            /* ... */
13.        }
14.        else {
15.            ret = LEPT_PARSE_MISS_COMMA_OR_SQUARE_BRACKET;
16.            break;
17.        }
18.    }
19.    /* Pop and free values on the stack */
20.    for (i = 0; i < size; i++)
21.        lept_free((lept_value*)lept_context_pop(c,
22.            sizeof(lept_value)));
23.    return ret;
24. }

```

5. bug 的解释

这个 bug 源于压栈时，会获得一个指针 `e`，指向从堆栈分配到的空间：

```

1.     for (;;) {
2.         /* bug! */
3.         lept_value* e = lept_context_push(c, sizeof(lept_value));
4.         lept_init(e);
5.         size++;
6.         if ((ret = lept_parse_value(c, e)) != LEPT_PARSE_OK)

```

```

7.         return ret;
8.         /* ... */
9.     }

```

然后，我们把这个指针调用 `lept_parse_value(c, e)`，这里会出现问题，因为 `lept_parse_value()` 及之下的函数都需要调用 `lept_context_push()`，而 `lept_context_push()` 在发现栈满了的时候会用 `realloc()` 扩容。这时候，我们上层的 `e` 就会失效，变成一个悬挂指针（dangling pointer），而且 `lept_parse_value(c, e)` 会通过这个指针写入解析结果，造成非法访问。

在使用 C++ 容器时，也会遇到类似的问题。从容器中取得的迭代器（iterator）后，如果改动容器内容，之前的迭代器会失效。这里的悬挂指针问题也是相同的。

但这种 bug 有时可能在简单测试中不能自动发现，因为问题只有堆栈满了才会出现。从测试的角度看，我们需要一些压力测试（stress test），测试更大更复杂的数据。但从编程的角度看，我们要谨慎考虑变量的生命周期，尽量从编程阶段避免出现问题。例如把

`lept_context_push()` 的 API 改为：

```

1. static void lept_context_push(lept_context* c, const void* data,
    size_t size);

```

这样就确把数据压入栈内，避免了返回指针的生命周期问题。但我们之后会发现，原来的 API 设计在一些情况会更方便一些，例如在把字符串值转化（stringify）为 JSON 时，我们可以预先在堆栈分配字符串所需的最大空间，而当时是未有数据填充进去的。

无论如何，我们编程时都要考虑清楚变量的生命周期，特别是指针的生命周期。

6. 总结

经过对数组的解析，我们也了解到如何利用递归处理复合型的数据类型解析。与一些用链表或自动扩展的动态数组的实现比较，我们利用了自定义堆栈作为缓冲区，能分配最紧凑的数组作存储之用，会比其他实现更省内存。我们完成了数组类型后，只余下对象类型了。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（六）：解析对象

- 从零开始的 JSON 库教程（六）：解析对象
 - 1. JSON 对象
 - 2. 数据结构
 - 3. 重构字符串解析
 - 4. 实现
 - 5. 总结与练习

从零开始的 JSON 库教程（六）：解析对象

- Milo Yip
- 2016/10/29

本文是《从零开始的 JSON 库教程》的第六个单元。代码位于 [json-tutorial/tutorial06](https://github.com/miloyp/json-tutorial/tutorial06)。

本单元内容：

1. JSON 对象
2. 数据结构
3. 重构字符串解析
4. 实现
5. 总结与练习

1. JSON 对象

此单元是本教程最后一个关于 JSON 解析器的部分。JSON 对象和 JSON 数组非常相似，区别包括 JSON 对象以花括号

`{ }`（`U+007B`、`U+007D`）包裹表示，另外 JSON 对象由对象成员

(member) 组成，而 JSON 数组由 JSON 值组成。所谓对象成员，就是键值对，键必须为 JSON 字符串，然后值是什么 JSON 值，中间以冒号 `:` (`U+003A`) 分隔。完整语法如下：

```
1. member = string ws %x3A ws value
2. object = %x7B ws [ member *( ws %x2C ws member ) ] ws %x7D
```

2. 数据结构

要表示键值对的集合，有很多数据结构可供选择，例如：

- 动态数组 (dynamic array)：可扩展容量的数组，如 C++ 的 `std::vector`。
- 有序动态数组 (sorted dynamic array)：和动态数组相同，但保证元素已排序，可用二分搜寻查询成员。
- 平衡树 (balanced tree)：平衡二叉树可有序地遍历成员，如红黑树和 C++ 的 `std::map` (`std::multi_map` 支持重复键)。
- 哈希表 (hash table)：通过哈希函数能实现平均 $O(1)$ 查询，如 C++11 的 `std::unordered_map` (`unordered_multimap` 支持重复键)。

设一个对象有 n 个成员，数据结构的容量是 m ， $n \leq m$ ，那么一些常用操作的时间 / 空间复杂度如下：

	动态数组	有序动态数组	平衡树	哈希表
有序	否	是	是	否
自定成员次序	可	否	否	否
初始化 n 个成员	$O(n)$	$O(n \log n)$	$O(n \log n)$	平均 $O(n)$ 、最坏 $O(n^2)$
加入成员	分摊 $O(1)$	$O(n)$	$O(\log n)$	平均 $O(1)$ 、最坏 $O(n)$
移除成员	$O(n)$	$O(n)$	$O(\log n)$	平均 $O(1)$ 、最坏 $O(n)$

查询成员	$O(n)$	$O(\log n)$	$O(\log n)$	平均 $O(1)$ 、最坏 $O(n)$
遍历成员	$O(n)$	$O(n)$	$O(n)$	$O(m)$
检测对象相等	$O(n^2)$	$O(n)$	$O(n)$	平均 $O(n)$ 、最坏 $O(n^2)$
空间	$O(m)$	$O(m)$	$O(n)$	$O(m)$

在 ECMA-404 标准中，并没有规定对象中每个成员的键一定要唯一的，也没有规定是否需要维持成员的次序。

为了简单起见，我们的 leptjson 选择用动态数组的方案。我们将在单元八才加入动态功能，所以这单元中，每个对象仅仅是成员的数组。那么它跟上一单元的数组非常接近：

```

1. typedef struct lept_value lept_value;
2. typedef struct lept_member lept_member;
3.
4. struct lept_value {
5.     union {
6.         struct { lept_member* m; size_t size; }o;
7.         struct { lept_value* e; size_t size; }a;
8.         struct { char* s; size_t len; }s;
9.         double n;
10.    }u;
11.    lept_type type;
12. };
13.
14. struct lept_member {
15.     char* k; size_t klen; /* member key string, key string length
16.                          */
17.     lept_value v; /* member value */
18. };

```

成员结构 `lept_member` 是一个 `lept_value` 加上键的字符串。如同 JSON 字符串的值，我们也需要同时保留字符串的长度，因为字符串本身可能包含空字符 `\u0000`。

在这单元中，我们仅添加了最基本的访问函数，用于撰写单元测试：

```
1. size_t lept_get_object_size(const lept_value* v);
2. const char* lept_get_object_key(const lept_value* v, size_t index);
3. size_t lept_get_object_key_length(const lept_value* v, size_t
   index);
4. lept_value* lept_get_object_value(const lept_value* v, size_t
   index);
```

在软件开发过程中，许多时候，选择合适的数据结构后已等于完成一半工作。没有完美的数据结构，所以最好考虑多一些应用的场合，看看时间 / 空间复杂度以至相关系数是否合适。

接下来，我们就可以着手实现。

3. 重构字符串解析

在软件工程中，**代码重构**（code refactoring）是指在不改变软件外在行为时，修改代码以改进结构。代码重构十分依赖于单元测试，因为我们是通过单元测试去维护代码的正确性。有了足够的单元测试，我们可以放胆去重构，尝试并评估不同的改进方式，找到合乎心意而且能通过单元测试的改动，我们才提交它。

我们知道，成员的键也是一个 JSON 字符串，然而，我们不使用

`lept_value` 存储键，因为这样会浪费了当中 `type` 这个无用的字段。由于 `lept_parse_string()` 是直接地把解析的结果写进一个 `lept_value`，所以我们先用「提取方法（extract method，见下注）」的重构方式，把解析 JSON 字符串及写入 `lept_value` 分拆成两部分：

```
1. /* 解析 JSON 字符串，把结果写入 str 和 len */
2. /* str 指向 c->stack 中的元素，需要在 c->stack */
3. static int lept_parse_string_raw(lept_context* c, char** str,
```

```

    size_t* len) {
4.     /* \todo */
5. }
6.
7. static int lept_parse_string(lept_context* c, lept_value* v) {
8.     int ret;
9.     char* s;
10.    size_t len;
11.    if ((ret = lept_parse_string_raw(c, &s, &len)) ==
        LEPT_PARSE_OK)
12.        lept_set_string(v, s, len);
13.    return ret;
14. }

```

这样的话，我们实现对象的解析时，就可以使用

`lept_parse_string_raw()` 来解析 JSON 字符串，然后把结果复制至 `lept_member` 的 `k` 和 `klen` 字段。

注：在 Fowler 的经典著作 [1] 中，把各种重构方式分门别类，每个方式都有详细的步骤说明。由于书中以 Java 为例子，所以方式的名称使用了 Java 的术语，例如方法 (method)。在 C 语言中，「提取方法」其实应该称为「提取函数」。

[1] Fowler, Martin. Refactoring: improving the design of existing code. Pearson Education India, 2009. 中译本：熊节译，《重构——改善既有代码的设计》，人民邮电出版社，2010年。

4. 实现

解析对象与解析数组非常相似，所以我留空了几段作为练习。在解析数组时，我们把当前的元素以 `lept_value` 压入栈中，而在这里，我们则是以 `lept_member` 压入：

```

1. static int lept_parse_object(lept_context* c, lept_value* v) {
2.     size_t size;
3.     lept_member m;
4.     int ret;
5.     EXPECT(c, '{');
6.     lept_parse_whitespace(c);
7.     if (*c->json == '}') {
8.         c->json++;
9.         v->type = LEPT_OBJECT;
10.        v->u.o.m = 0;
11.        v->u.o.size = 0;
12.        return LEPT_PARSE_OK;
13.    }
14.    m.k = NULL;
15.    size = 0;
16.    for (;;) {
17.        lept_init(&m.v);
18.        /* \todo parse key to m.k, m.klen */
19.        /* \todo parse ws colon ws */
20.        /* parse value */
21.        if ((ret = lept_parse_value(c, &m.v)) != LEPT_PARSE_OK)
22.            break;
23.        memcpy(lept_context_push(c, sizeof(lept_member)), &m,
24.            sizeof(lept_member));
25.        size++;
26.        m.k = NULL; /* ownership is transferred to member on stack */
27.        /* \todo parse ws [comma | right-curly-brace] ws */
28.    }
29.    /* \todo Pop and free members on the stack */
30.    return ret;
}

```

要注意的是，我们要为 `m.k` 分配内存去存储键的字符串，若在整个对象解析时发生错误，也要记得释放栈中的 `lept_member` 的 `k`。

我们为解析对象定义了几个新的错误码：

```

1. enum {
2.     /* ... */
3.     LEPT_PARSE_MISS_KEY,
4.     LEPT_PARSE_MISS_COLON,
5.     LEPT_PARSE_MISS_COMMA_OR_CURLY_BRACKET
6. };

```

在此不再赘述它们的意义了，可从以下的单元测试看到例子：

```

1. static void test_parse_miss_key() {
2.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{:1,");
3.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{1:1,");
4.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{true:1,");
5.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{false:1,");
6.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{null:1,");
7.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{[:1,");
8.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{}:1,");
9.     TEST_ERROR(LEPT_PARSE_MISS_KEY, "{\a":1,");
10. }
11.
12. static void test_parse_miss_colon() {
13.     TEST_ERROR(LEPT_PARSE_MISS_COLON, "{\a}");
14.     TEST_ERROR(LEPT_PARSE_MISS_COLON, "{\a\", \"b\"}");
15. }
16.
17. static void test_parse_miss_comma_or_curly_bracket() {
18.     TEST_ERROR(LEPT_PARSE_MISS_COMMA_OR_CURLY_BRACKET, "{\a":1");
19.     TEST_ERROR(LEPT_PARSE_MISS_COMMA_OR_CURLY_BRACKET, "
20.         {\a":1]");
21.     TEST_ERROR(LEPT_PARSE_MISS_COMMA_OR_CURLY_BRACKET, "{\a":1
22.         \b\"");
23.     TEST_ERROR(LEPT_PARSE_MISS_COMMA_OR_CURLY_BRACKET, "{\a":
24.         {}");
25. }

```

5. 总结与练习

在本单元中，除了谈及 JSON 对象的语法、可选的数据结构、实现方式，我们也轻轻谈及了重构的概念。有赖于测试驱动开发（TDD），我们可以不断重塑软件的内部结构。

完成这次练习之后，恭喜你，你已经完整地实现了一个符合标准的 JSON 解析器了。之后我们会完成更简单的生成器及其他访问功能。

由于对象和数组的相似性，此单元留空了较多实现部分作为练习：

1. 依第 3 节所述，重构 `lept_parse_string()`。重构前运行单元测试，重构后确保单元测试仍保持通过。
2. 打开 `test.c` 中两个 `#if 0`，运行单元测试，证实单元测试不通过。然后实现 `lept_parse_object()` 中的 `\todo` 部分。验证实现能通过单元测试。
3. 使用工具检测内存泄漏，解决它们。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（六）：解析对象解答篇

- 从零开始的 JSON 库教程（六）：解析对象解答篇

- 1. 重构 `lept_parse_string()`
- 2. 实现 `lept_parse_object()`
- 3. 释放内存
- 4. 总结

从零开始的 JSON 库教程（六）：解析对象解答篇

- Milo Yip
- 2016/11/15

本文是《从零开始的 JSON 库教程》的第六个单元解答篇。解答代码位于 [json-tutorial/tutorial06_answer](https://github.com/miloyip/json-tutorial/blob/master/tutorial06_answer.c)。

1. 重构 `lept_parse_string()`

这个「提取方法」重构练习很简单，只需要把原来调用

`lept_set_string` 的地方，改为写入参数变量。因此，原来的 `lept_parse_string()` 和 答案中的 `lept_parse_string_raw()` 的 diff 仅是两处：

```
1. 130,131c130,131
2. < static int lept_parse_string(lept_context* c, lept_value* v) {
3. <     size_t head = c->top, len;
4. ---
5. > static int lept_parse_string_raw(lept_context* c, char** str,
6. >     size_t* len) {
7. 140,141c140,141
```

```

8. <          len = c->top - head;
9. <          lept_set_string(v, (const
    char*)lept_context_pop(c, len), len);
10. ---
11. >          *len = c->top - head;
12. >          *str = lept_context_pop(c, *len);

```

以 TDD 方式开发软件，因为有单元测试确保软件的正确性，面对新需求可以安心重构，改善软件架构。

2. 实现 lept_parse_object()

有了 lept_parse_array() 的经验，实现 lept_parse_object() 几乎是一样的，分别只是每个迭代要多处理一个键和冒号。我们把这个实现过程分为 5 步曲。

第 1 步是利用刚才重构出来的 lept_parse_string_raw() 去解析键的字符串。由于 lept_parse_string_raw() 假设第一个字符为 "，我们要先作校检，失败则要返回 LEPT_PARSE_MISS_KEY 错误。若字符串解析成功，它会把结果存储在我们的栈之中，需要把结果写入临时 lept_member 的 k 和 klen 字段中：

```

1. static int lept_parse_object(lept_context* c, lept_value* v) {
2.     size_t i, size;
3.     lept_member m;
4.     int ret;
5.     /* ... */
6.     m.k = NULL;
7.     size = 0;
8.     for (;;) {
9.         char* str;
10.         lept_init(&m.v);
11.         /* 1. parse key */
12.         if (*c->json != '"') {

```

```

13.         ret = LEPT_PARSE_MISS_KEY;
14.         break;
15.     }
16.     if ((ret = lept_parse_string_raw(c, &str, &m.klen)) !=
        LEPT_PARSE_OK)
17.         break;
18.     memcpy(m.k = (char*)malloc(m.klen + 1), str, m.klen);
19.     m.k[m.klen] = '\0';
20.     /* 2. parse ws colon ws */
21.     /* ... */
22. }
23. /* 5. Pop and free members on the stack */
24. /* ... */
25. }

```

第 2 步是解析冒号，冒号前后可有空白字符：

```

1.     /* 2. parse ws colon ws */
2.     lept_parse_whitespace(c);
3.     if (*c->json != ':') {
4.         ret = LEPT_PARSE_MISS_COLON;
5.         break;
6.     }
7.     c->json++;
8.     lept_parse_whitespace(c);

```

第 3 步是解析任意的 JSON 值。这部分与解析数组一样，递归调用 `lept_parse_value()`，把结果写入临时 `lept_member` 的 `v` 字段，然后把整个 `lept_member` 压入栈：

```

1.     /* 3. parse value */
2.     if ((ret = lept_parse_value(c, &m.v)) != LEPT_PARSE_OK)
3.         break;
4.     memcpy(lept_context_push(c, sizeof(lept_member)), &m,
        sizeof(lept_member));
5.     size++;
6.     m.k = NULL; /* ownership is transferred to member on stack

```

```
*/
```

但有一点要注意，如果之前缺乏冒号，或是这里解析值失败，在函数返回前我们要释放 `m.k`。如果我们成功地解析整个成员，那么就要把 `m.k` 设为空指针，其意义是说明该键的字符串的拥有权已转移至栈，之后如遇到错误，我们不会重覆释放栈里成员的键和这个临时成员的键。

第 4 步，解析逗号或右花括号。遇上右花括号的话，当前的 JSON 对象就解析完结了，我们把栈上的成员复制至结果，并直接返回：

```
1.      /* 4. parse ws [comma | right-curly-brace] ws */
2.      lept_parse_whitespace(c);
3.      if (*c->json == ',') {
4.          c->json++;
5.          lept_parse_whitespace(c);
6.      }
7.      else if (*c->json == '}') {
8.          size_t s = sizeof(lept_member) * size;
9.          c->json++;
10.         v->type = LEPT_OBJECT;
11.         v->u.o.size = size;
12.         memcpy(v->u.o.m = (lept_member*)malloc(s),
lept_context_pop(c, s), s);
13.         return LEPT_PARSE_OK;
14.     }
15.     else {
16.         ret = LEPT_PARSE_MISS_COMMA_OR_CURLY_BRACKET;
17.         break;
18.     }
```

最后，当 `for (;;)` 中遇到任何错误便会到达这第 5 步，要释放临时的 key 字符串及栈上的成员：

```
1.      /* 5. Pop and free members on the stack */
```

```

2.     free(m.k);
3.     for (i = 0; i < size; i++) {
4.         lept_member* m = (lept_member*)lept_context_pop(c,
sizeof(lept_member));
5.         free(m->k);
6.         lept_free(&m->v);
7.     }
8.     v->type = LEPT_NULL;
9.     return ret;

```

注意我们不需要先检查 `m.k != NULL`，因为 `free(NULL)` 是完全合法的。

3. 释放内存

使用工具检测内存泄漏时，我们会发现以下这行代码造成内存泄漏：

```

1. memcpy(v->u.o.m = (lept_member*)malloc(s), lept_context_pop(c, s),
s);

```

类似数组，我们也需要在 `lept_free()` 释放 JSON 对象的成员（包括键及值）：

```

1. void lept_free(lept_value* v) {
2.     size_t i;
3.     assert(v != NULL);
4.     switch (v->type) {
5.         /* ... */
6.         case LEPT_OBJECT:
7.             for (i = 0; i < v->u.o.size; i++) {
8.                 free(v->u.o.m[i].k);
9.                 lept_free(&v->u.o.m[i].v);
10.            }
11.            free(v->u.o.m);
12.            break;
13.            default: break;

```

```
14.     }  
15.     v->type = LEPT_NULL;  
16. }
```

4. 总结

至此，你已实现一个完整的 JSON 解析器，可解析任何合法的 JSON。统计一下，不计算空行及注释，现时 `leptjson.c` 只有 405 行代码，`lept_json.h` 54 行，`test.c` 309 行。

另一方面，一些程序也需要生成 JSON。也许最初读者会以为生成 JSON 只需直接 `sprintf()/fprintf()` 就可以，但深入了解 JSON 的语法之后，我们应该知道 JSON 语法还是需做一些处理，例如字符串的转义、数字的格式等。然而，实现生成器还是要比解析器容易得多。而且，假设我们有一个正确的解析器，可以简单使用 `roundtrip` 方式实现测试。请期待下回分解。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（七）：生成器

- 从零开始的 JSON 库教程（七）：生成器
 - 1. JSON 生成器
 - 2. 再利用 lept_context 做动态数组
 - 3. 生成 null、false 和 true
 - 4. 生成数字
 - 5. 总结与练习

从零开始的 JSON 库教程（七）：生成器

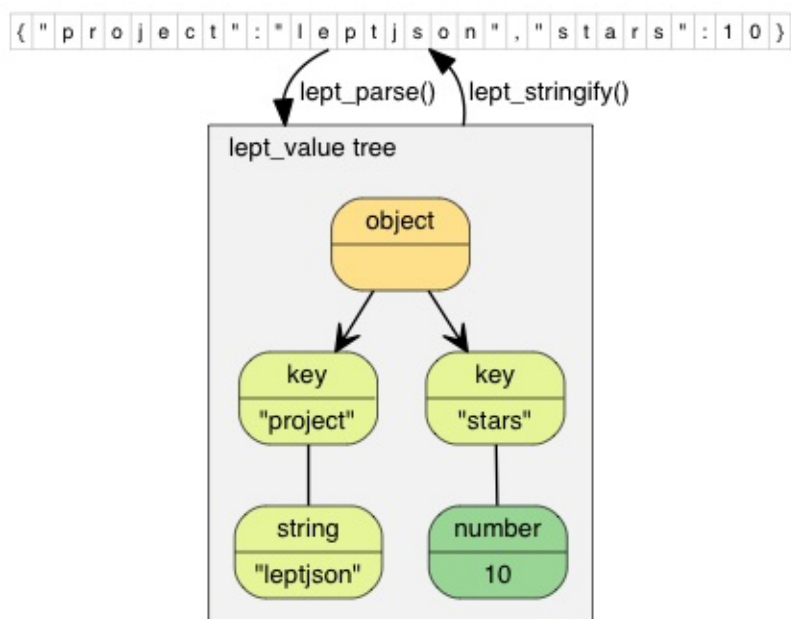
- Milo Yip
- 2016/12/20

本文是《从零开始的 JSON 库教程》的第七个单元。代码位于 [json-tutorial/tutorial07](https://github.com/miloyip/json-tutorial/tutorial07)。

1. JSON 生成器

我们在前 6 个单元实现了一个合乎标准的 JSON 解析器，它把 JSON 文本解析成一个树形数据结构，整个结构以 `lept_value` 的节点组成。

JSON 生成器 (generator) 负责相反的事情，就是把树形数据结构转换成 JSON 文本。这个过程又称为「字符串化 (stringify)」。



相对于解析器，通常生成器更容易实现，而且生成器几乎不会造成运行时错误。因此，生成器的 API 设计为以下形式，直接返回 JSON 的字符串：

```
1. char* lept_stringify(const lept_value* v, size_t* length);
```

`length` 参数是可选的，它会存储 JSON 的长度，传入 `NULL` 可忽略此参数。使用方需负责用 `free()` 释放内存。

为了简单起见，我们不做换行、缩进等美化（`prettify`）处理，因此它生成的 JSON 会是单行、无空白字符的最紧凑形式。

2. 再利用 `lept_context` 做动态数组

在实现 JSON 解析时，我们加入了一个动态变长的堆栈，用于存储临时的解析结果。而现在，我们也需要存储生成的结果，所以最简单是再利用该数据结构，作为输出缓冲区。

```
1. #ifndef LEPT_PARSE_STRINGIFY_INIT_SIZE
```

```

2.  #define LEPT_PARSE_STRINGIFY_INIT_SIZE 256
3.  #endif
4.
5.  int lept_stringify(const lept_value* v, char** json, size_t*
    length) {
6.      lept_context c;
7.      int ret;
8.      assert(v != NULL);
9.      assert(json != NULL);
10.     c.stack = (char*)malloc(c.size =
        LEPT_PARSE_STRINGIFY_INIT_SIZE);
11.     c.top = 0;
12.     if ((ret = lept_stringify_value(&c, v)) != LEPT_STRINGIFY_OK) {
13.         free(c.stack);
14.         *json = NULL;
15.         return ret;
16.     }
17.     if (length)
18.         *length = c.top;
19.     PUTC(&c, '\0');
20.     *json = c.stack;
21.     return LEPT_STRINGIFY_OK;
22. }

```

生成根节点的值之后，我需还需要加入一个空字符作结尾。

如前所述，此 API 还提供了 `length` 可选参数，当传入非空指针时，就能获得生成 JSON 的长度。或许读者会疑问，为什么需要获得长度，我们不是可以用 `strlen()` 获得么？是的，因为 JSON 不会含有空字符（若 JSON 字符串中含空字符，必须转义为 `\u0000`），用 `strlen()` 是没有问题的。但这样做会带来不必要的性能消耗，理想地是避免调用方有额外消耗。

3. 生成 null、false 和 true

接下来，我们生成最简单的 JSON 类型，就是 3 种 JSON 字面值。为贯彻 TDD，先写测试：

```

1. #define TEST_ROUNDTRIP(json)\
2.     do {\
3.         lept_value v;\
4.         char* json2;\
5.         size_t length;\
6.         lept_init(&v);\
7.         EXPECT_EQ_INT(LEPT_PARSE_OK, lept_parse(&v, json));\
8.         EXPECT_EQ_INT(LEPT_STRINGIFY_OK, lept_stringify(&v, &json2,\
           &length));\
9.         EXPECT_EQ_STRING(json, json2, length);\
10.        lept_free(&v);\
11.        free(json2);\
12.    } while(0)
13.
14. static void test_stringify() {
15.     TEST_ROUNDTRIP("null");
16.     TEST_ROUNDTRIP("false");
17.     TEST_ROUNDTRIP("true");
18.     /* ... */
19. }

```

这里我们采用一个最简单的测试方式，把一个 JSON 解析，然后再生成另一 JSON，逐字符比较两个 JSON 是否一模一样。这种测试可称为往返（roundtrip）测试。但需要注意，同一个 JSON 的内容可以有多种不同的表示方式，例如可以插入不定数量的空白字符，数字

`1.0` 和 `1` 也是等价的。所以另一种测试方式，是比较两次解析的结果（`lept_value` 的树）是否相同，此功能将会在下一单元讲解。

然后，我们实现 `lept_stringify_value`，加入一个 `PUTS()` 宏去输出字符串：

```

1. #define PUTS(c, s, len)    memcpy(lept_context_push(c, len), s,

```

```

len)
2.
3. static int lept_stringify_value(lept_context* c, const lept_value*
   v) {
4.     size_t i;
5.     int ret;
6.     switch (v->type) {
7.         case LEPT_NULL:    PUTS(c, "null", 4); break;
8.         case LEPT_FALSE:   PUTS(c, "false", 5); break;
9.         case LEPT_TRUE:    PUTS(c, "true", 4); break;
10.        /* ... */
11.    }
12.    return LEPT_STRINGIFY_OK;
13. }

```

4. 生成数字

为了简单起见，我们使用 `sprintf("%.17g", ...)` 来把浮点数转换成文本。`"%.17g"` 是足够把双精度浮点转换成可还原的文本。

最简单的实现方式可能是这样的：

```

1.     case LEPT_NUMBER:
2.     {
3.         char buffer[32];
4.         int length = sprintf(buffer, "%.17g", v->u.n);
5.         PUTS(c, buffer, length);
6.     }
7.     break;

```

但这样需要在 `PUTS()` 中做一次 `memcpy()`，实际上我们可以避免这次复制，只需要生成的时候直接写进 `c` 里的推栈，然后再按实际长度调查 `c->top`：

```

1.     case LEPT_NUMBER:

```

```

2.         {
3.             char* buffer = lept_context_push(c, 32);
4.             int length = sprintf(buffer, "%.17g", v->u.n);
5.             c->top -= 32 - length;
6.         }
7.         break;

```

因每个临时变量只用了一次，我们可以把代码压缩成一行：

```

1.         case LEPT_NUMBER:
2.             c->top -= 32 - sprintf(lept_context_push(c, 32),
3.             "%.17g", v->u.n);
4.             break;

```

5. 总结与练习

我们在此单元中简介了 JSON 的生成功能和 leptjson 中的实现方式。

leptjson 重复利用了 `lept_context` 中的数据结构作为输出缓冲，可以节省代码量。

生成通常比解析简单（一个例外是 RapidJSON 自行实现了浮点数至字符串的算法），余下的 3 种 JSON 类型就当作练习吧：

1. 由于有两个地方需要生成字符串（JSON 字符串和对象类型），所以先实现 `lept_stringify_string()`。注意，字符串的语法比较复杂，一些字符必须转义，其他少于 `0x20` 的字符需要转义为 `\u00xx` 形式。
2. 直接在 `lept_stringify_value()` 的 `switch` 内实现 JSON 数组和对象类型的生成。这些实现里都会递归调用 `lept_stringify_value()`。

3. 在你的 `lept_stringify_string()` 是否使用了多次 `putc()`？如果是，它每次输出一个字符时，都要检测缓冲区是否有足够空间（不够时需扩展）。能否优化这部分的性能？这种优化有什么代价么？

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。

从零开始的 JSON 库教程（七）：生成器解答篇

- 从零开始的 JSON 库教程（七）：生成器解答篇
 - 1. 生成字符串
 - 2. 生成数组和对象
 - 3. 优化 `lept_stringify_string()`
 - 4. 总结

从零开始的 JSON 库教程（七）：生成器解答篇

- Milo Yip
- 2017/1/5

本文是《从零开始的 JSON 库教程》的第七个单元解答篇。解答代码位于 [json-tutorial/tutorial07_answer](https://github.com/miloyip/json-tutorial/blob/master/tutorial07_answer.c)。

1. 生成字符串

我们需要对一些字符进行转义，最简单的实现如下：

```
1. static void lept_stringify_string(lept_context* c, const char* s,
   size_t len) {
2.     size_t i;
3.     assert(s != NULL);
4.     PUTC(c, '"');
5.     for (i = 0; i < len; i++) {
6.         unsigned char ch = (unsigned char)s[i];
7.         switch (ch) {
8.             case '\\': PUTS(c, "\\\"", 2); break;
9.             case '\\": PUTS(c, "\\\\", 2); break;
10.            case '\b': PUTS(c, "\\b", 2); break;
11.            case '\f': PUTS(c, "\\f", 2); break;
12.            case '\n': PUTS(c, "\\n", 2); break;
```

```

13.         case '\\r': PUTS(c, "\\r", 2); break;
14.         case '\\t': PUTS(c, "\\t", 2); break;
15.         default:
16.             if (ch < 0x20) {
17.                 char buffer[7];
18.                 sprintf(buffer, "\\u%04X", ch);
19.                 PUTS(c, buffer, 6);
20.             }
21.             else
22.                 PUTC(c, s[i]);
23.         }
24.     }
25.     PUTC(c, '"');
26. }
27.
28. static void lept_stringify_value(lept_context* c, const lept_value*
    v) {
29.     switch (v->type) {
30.         /* ... */
31.         case LEPT_STRING: lept_stringify_string(c, v->u.s.s, v-
    >u.s.len); break;
32.         /* ... */
33.     }
34. }

```

注意到，十六进制输出的字母可以用大写或小写，我们这里选择了大写，所以 `roundstrip` 测试时也用大写。但这个并不是必然的，输出小写（用 `"\\u%04x"`）也可以。

2. 生成数组和对象

生成数组也是非常简单，只要输出 `[` 和 `]`，中间对逐个子值递归调用 `lept_stringify_value()`。只要注意在第一个元素后才加入 `,`。而对象也仅是多了一个键和 `:`。

```
1. static void lept_stringify_value(lept_context* c, const lept_value*
```



```

v) {
2.     size_t i;
3.     switch (v->type) {
4.         /* ... */
5.         case LEPT_ARRAY:
6.             PUTC(c, '[');
7.             for (i = 0; i < v->u.a.size; i++) {
8.                 if (i > 0)
9.                     PUTC(c, ',');
10.                lept_stringify_value(c, &v->u.a.e[i]);
11.            }
12.            PUTC(c, ']');
13.            break;
14.        case LEPT_OBJECT:
15.            PUTC(c, '{');
16.            for (i = 0; i < v->u.o.size; i++) {
17.                if (i > 0)
18.                    PUTC(c, ',');
19.                lept_stringify_string(c, v->u.o.m[i].k, v->u.o.m[i].klen);
20.                PUTC(c, ':');
21.                lept_stringify_value(c, &v->u.o.m[i].v);
22.            }
23.            PUTC(c, '}');
24.            break;
25.        /* ... */
26.    }
27. }

```

3. 优化 lept_stringify_string()

最后，我们讨论一下优化。上面的 lept_stringify_string() 实现中，每次输出一个字符 / 字符串，都要调用 lept_context_push()。如果我们使用一些性能剖测工具，也可能会发现这个函数消耗较多 CPU。

```
1. static void* lept_context_push(lept_context* c, size_t size) {
```

```

2.     void* ret;
3.     assert(size > 0);
4.     if (c->top + size >= c->size) { // (1)
5.         if (c->size == 0)
6.             c->size = LEPT_PARSE_STACK_INIT_SIZE;
7.         while (c->top + size >= c->size)
8.             c->size += c->size >> 1; /* c->size * 1.5 */
9.         c->stack = (char*)realloc(c->stack, c->size);
10.    }
11.    ret = c->stack + c->top;          // (2)
12.    c->top += size;                  // (3)
13.    return ret;                     // (4)
14. }

```

中间最花费时间的，应该会是 (1)，需要计算而且作分支检查。即使使用 C99 的 `inline` 关键字（或使用宏）去减少函数调用的开销，这个分支也无法避免。

所以，一个优化的点子是，预先分配足够的内存，每次加入字符就不用做这个检查了。但多大的内存才足够呢？我们可以看到，每个字符可生成最长的形式是 `\u00XX`，占 6 个字符，再加上前后两个双引号，也就是共 `len * 6 + 2` 个输出字符。那么，使用 `char* p = lept_context_push()` 作一次分配后，便可以用 `*p++ = c` 去输出字符了。最后，再按实际输出量调整堆栈指针。

另一个小优化点，是自行编写十六进位输出，避免了 `printf()` 内解析格式的开销。

```

1. static void lept_stringify_string(lept_context* c, const char* s,
   size_t len) {
2.     static const char hex_digits[] = { '0', '1', '2', '3', '4',
   '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
3.     size_t i, size;
4.     char* head, *p;
5.     assert(s != NULL);

```

```

6.     p = head = lept_context_push(c, size = len * 6 + 2); /*
      "\u00xx..." */
7.     *p++ = '"';
8.     for (i = 0; i < len; i++) {
9.         unsigned char ch = (unsigned char)s[i];
10.        switch (ch) {
11.            case '\\': *p++ = '\\'; *p++ = '\\'; break;
12.            case '\': *p++ = '\\'; *p++ = '\\'; break;
13.            case '\b': *p++ = '\\'; *p++ = 'b'; break;
14.            case '\f': *p++ = '\\'; *p++ = 'f'; break;
15.            case '\n': *p++ = '\\'; *p++ = 'n'; break;
16.            case '\r': *p++ = '\\'; *p++ = 'r'; break;
17.            case '\t': *p++ = '\\'; *p++ = 't'; break;
18.            default:
19.                if (ch < 0x20) {
20.                    *p++ = '\\'; *p++ = 'u'; *p++ = '0'; *p++ =
21.                    '0';
22.                    *p++ = hex_digits[ch >> 4];
23.                    *p++ = hex_digits[ch & 15];
24.                }
25.                else
26.                    *p++ = s[i];
27.            }
28.        *p++ = '"';
29.        c->top -= size - (p - head);
30.    }

```

要注意的是，很多优化都是有代价的。第一个优化采取空间换时间的策略，对于只含一个字符串的 JSON，很可能会分配多 6 倍内存；但对于正常含多个值的 JSON，多分配的内存可在之后的值所利用，不会造成太多浪费。

而第二个优化的缺点，就是有稍增加了一点程序体积。也许有人会问，为什么 `hex_digits` 不用字符串字面量 `"0123456789ABCDEF"`？其实是可以的，但这会多浪费 1 个字节（实际因数据对齐可能会浪费 4

个或更多）。

4. 总结

我们用 80 行左右的代码就实现了 JSON 生成器，并尝试了做一些简单的优化。除了这种最简单的功能，有一些 JSON 库还会提供一些美化功能，即加入缩进及换行。另外，有一些应用可能需要大量输出数字，那么就可能需要优化数字的输出。这方面可考虑 C++ 开源库 [double-conversion](#)，以及参考本人另一篇文章《[RapidJSON 代码剖析（四）：优化 Grisu](#)》。

现时数组和对象类型只有最基本的访问、修改函数，我们会在下一篇补完。

如果你遇到问题，有不理解的地方，或是有建议，都欢迎在评论或 [issue](#) 中提出，让所有人一起讨论。