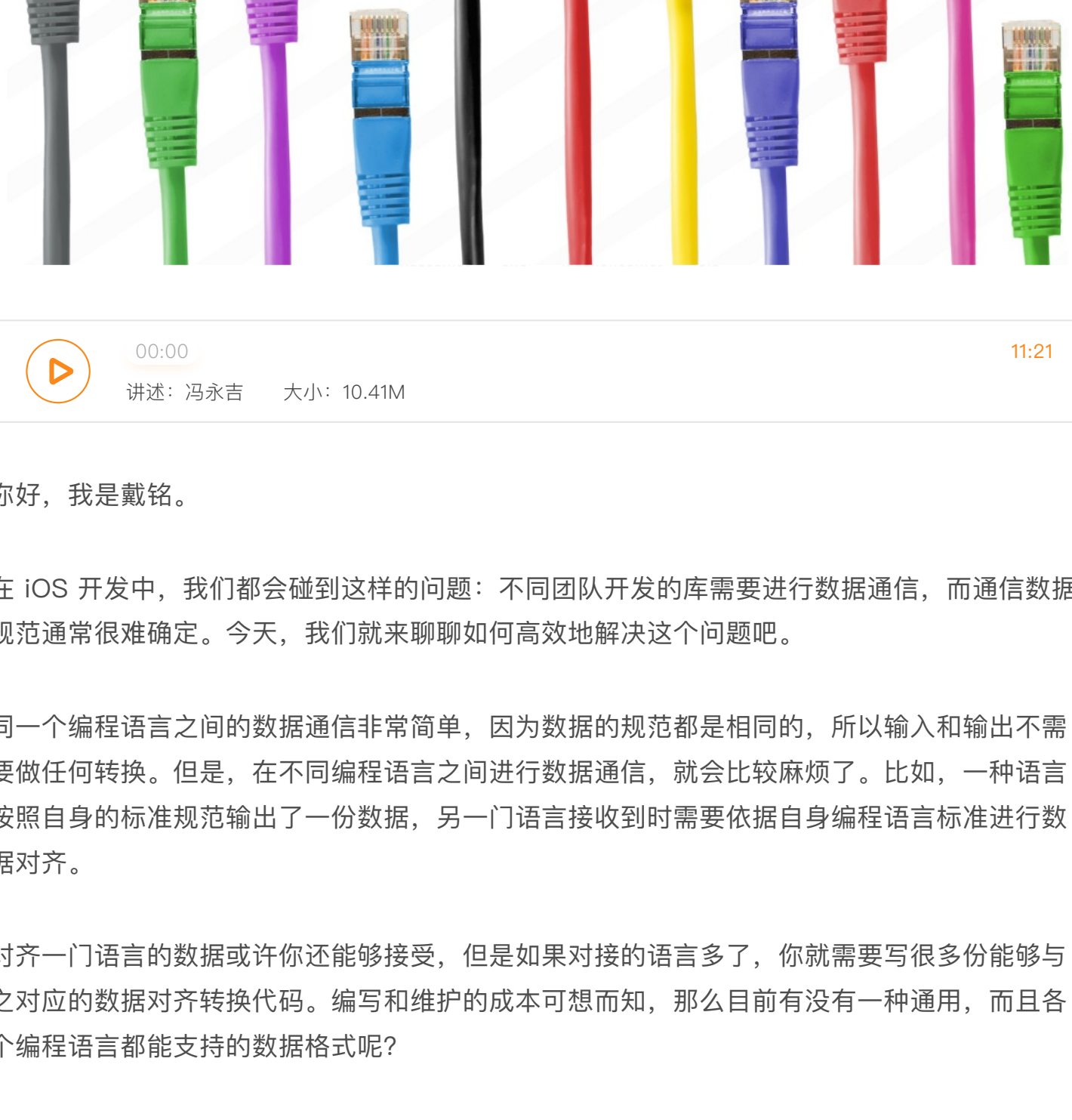
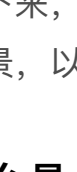



## 26 | 如何提高 JSON 解析的性能？

戴铭 2019-05-09



00:00

讲述：冯永吉    大小：10.41M

11:21

你好，我是戴铭。

在 iOS 开发中，我们都会碰到这样的问题：不同团队开发的库需要进行数据通信，而通信数据规范通常很难确定。今天，我们就来聊聊如何高效地解决这个问题吧。

同一个编程语言之间的数据通信非常简单，因为数据的规范都是相同的，所以输入和输出不需要做任何转换。但是，在不同编程语言之间进行数据通信，就会比较麻烦了。比如，一种语言按照自身的标准规范输出了一份数据，另一门语言接收到时需要依据自身编程语言标准进行数据对齐。

对齐一门语言的数据或许你还能够接受，但是如果对接的语言多了，你就需要写很多份能够与之对应的数据对齐转换代码。编写和维护的成本可想而知，那么目前有没有一种通用，而且各个编程语言都能支持的数据格式呢？

答案是有的。这个数据格式，就是我今天要跟你聊的 JSON。

接下来，在今天这篇文章中，我会先和你聊聊什么是 JSON；然后，再和你说说 JSON 的使用场景，以及 iOS 里是如何解析 JSON 的；最后，再和你分析如何提高 JSON 的解析性能。

### 什么是 JSON？

JSON，是 JavaScript Object Notation 的缩写。其实，JSON 最初是被设计为 JavaScript 语言的一个子集，但最终因为和编程语言无关，所以成为了一种开放标准的常见数据格式。

虽然 JSON 源于 JavaScript，但到目前很多编程语言都有了 JSON 解析的库，包括 C、C++、Java、Perl、Python 等等。除此之外，还有很多编程语言内置了 JSON 生成和解析的方法，比如 PHP 在 5.2 版本开始内置了 json\_encode() 方法，可以将 PHP 里的 Array 直接转换成 JSON。转换代码如下：

```
1 $arr = array(array(7,11,21));
2 echo json_encode($arr)."<br>";
3
4 $dic = array('name1' => 'val1', 'name2' => 'val2');
5 echo json_encode($dic);
6
```

输出结果如下：

```
1 [[7,11,21]]
2 {"name1":"val1","name2":"val2"}
3
```

如上所示，生成了两个 JSON 对象，第一个解析完后就是一个二维数组，第二个解析完后就是一个字典。**有了编程语言内置方法解析和生成 JSON 的支持，JSON 成为了理想的数据交换格式。**

通过上面生成的 JSON 可以看出，JSON 这种文本数据交换格式易读，且结构简单。

JSON 基于两种结构：

名字 / 值对集合：这种结构在其他编程语言里被实现为对象、字典、Hash 表、结构体或者关联数组。

有序值列表：这种结构在其他编程语言里被实现为数组、向量、列表或序列。

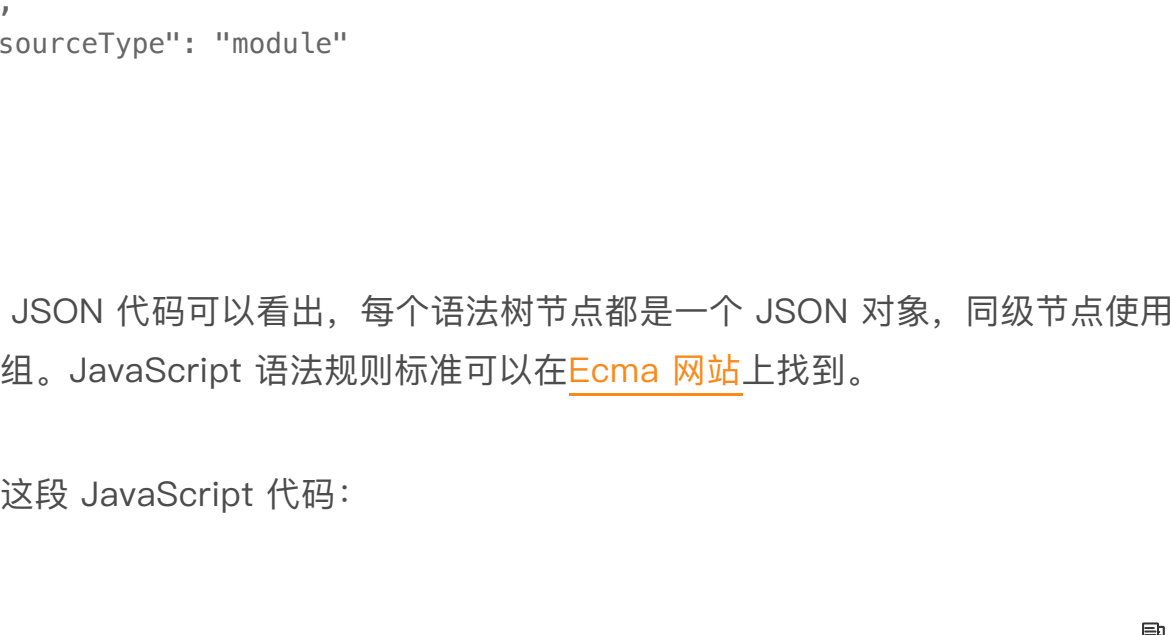
各种编程语言都以某种形式支持着这两种结构。比如，PHP 的 Array 既支持名字 / 值对集合又支持有序值列表；在 Swift 里键值集合就是字典，有序值列表就是数组。**名字 / 值对集合**在 JSON 和 JavaScript 里都被称为对象。JSON 语法图以及说明，你可以在 [JSON 官网](#) 查看。在这里，我只列出了几个用的比较多的语法图。



如上面语法图所示，对象是以左大括号开头和右大括号结尾，名字后面跟冒号，名字 / 值对用逗号分隔。比如：

```
1 {"name1":"val1","name2":"val2"}
2
```

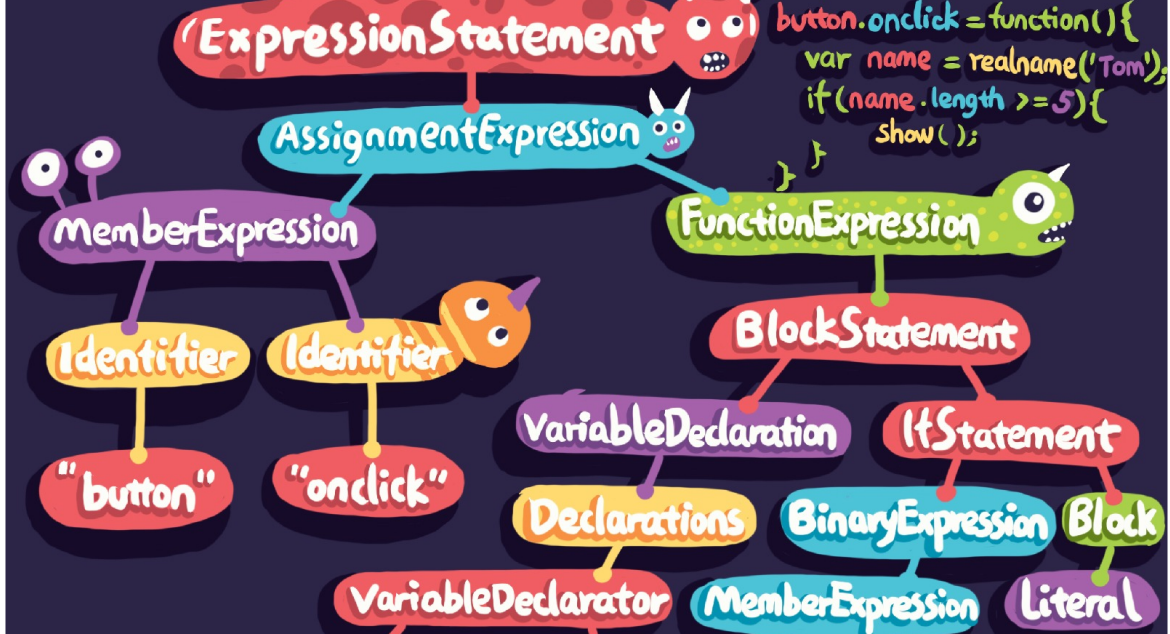
**有序值列表**在 JSON 和 JavaScript 里都叫数组，其语法图如下：



可以看出数组是以左中括号开头，以右中括号结尾，值以逗号分隔。数组代码如下所示：

```
1 [[7,11,21]]
2
```

**语法图中值**的语法图如下：



可以看出，值可以是字符串、数字、对象、数组、布尔值 true、布尔值 false、空值。根据这个语法，JSON 可以通过实现对象和数组的嵌套来描述更为复杂的数据结构。

JSON 是没有注释的，水平制表符、换行符、回车符都会被当做空格。字符串由双引号括起来，里面可以使零到多个 Unicode 字符序列，使用反斜杠来进行转义。

### JSON 的使用场景

JSON 的数据结构和任何一门编程语言的语法结构比起来都要简单得多，但它能干的事情却一点儿也不少，甚至可以完整地描述出一门编程语言的代码逻辑。比如，下面的这段 JavaScript 代码：

```
1 if (hour < 18) {
2   greeting = "Good day";
3 }
4
```

这段 JavaScript 代码的逻辑是，当 hour 变量小于 18 时，greeting 设置为 Good day 字符串，根据 JavaScript 的语法规则，完整逻辑的语法树结构可以通过 JSON 描述出来。对应的 JSON，如下：

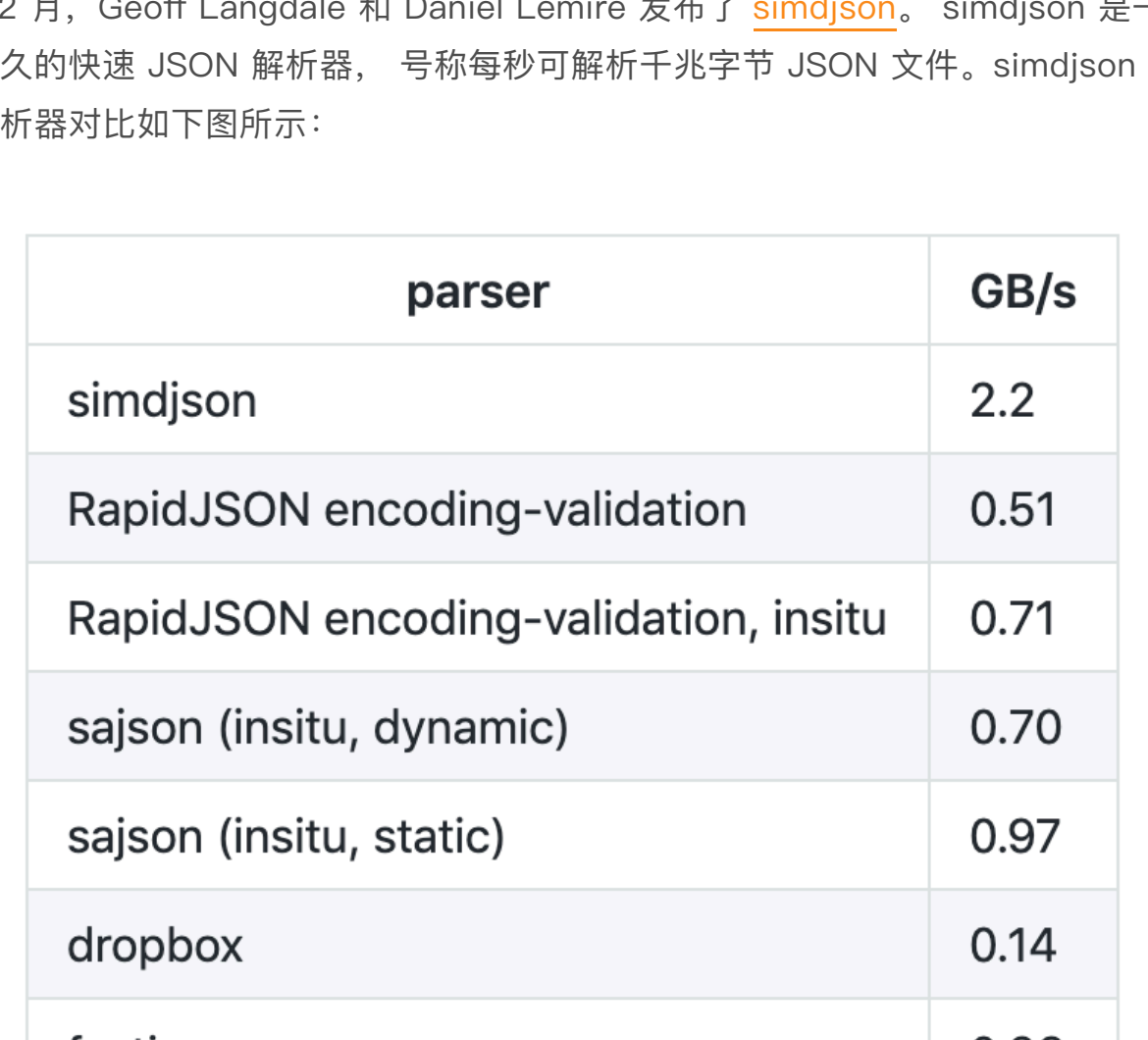
```
1 {
2   "type": "Program",
3   "body": [
4     {
5       "type": "IfStatement",
6       "test": {
7         "type": "BinaryExpression",
8         "left": {
9           "type": "Identifier",
10          "name": "hour"
11        },
12        "operator": "<",
13        "right": {
14          "type": "Literal",
15          "value": 18,
16          "raw": "18"
17        }
18      },
19      "consequent": {
20        "type": "BlockStatement",
21        "body": [
22          {
23            "type": "ExpressionStatement",
24            "expression": {
25              "type": "AssignmentExpression",
26              "operator": "=",
27              "left": {
28                "type": "Identifier",
29                "name": "greeting"
30              },
31              "right": {
32                "type": "Literal",
33                "value": "Good day",
34                "raw": "\"Good day\""
35              }
36            }
37          }
38        ]
39      },
40      "alternate": null
41    }
42  ],
43  "sourceType": "module"
44 }
45
```

从上面的 JSON 代码可以看出，每个语法树节点都是一个 JSON 对象，同级节点使用的是 JSON 数组。JavaScript 语法规则标准可以在 [Ecma 网站](#) 上找到。

比如下面这段 JavaScript 代码：

```
1 button.onclick = function() {
2   var name = realName('Tom');
3   if(name.length >= 5) {
4     show();
5   }
6 }
7
```

上面这段 JavaScript 代码对应的语法树如下图所示：



JavaScript 编程语言的语法树能够使用 JSON 来描述，其他编程语言同样也可以，比如 Objective-C 或 Swift，都能够生成自己的语法树结构，转成 JSON 后能够在运行期被动态地识别。因此，**App 的业务逻辑动态化就不仅限于使用 JavaScript 这一门语言来编写，而是可以选择使用其他你熟悉的语言。**

JSON 不仅可以描述业务数据使得业务数据能够动态更新，还可以用来描述业务逻辑，以实现业务逻辑的动态化，除此之外还可以用来描述页面布局。比如，我以前就做过这么一件事儿：解析一个 H5 页面编辑器生成的 JSON，将 JSON 对应生成 iOS 原生界面布局代码。我当时是用 Swift 语言来编写这个项目的，完整代码在[这里](#)。

在这个项目中，对 JSON 的解析使用的是系统自带的 JSONDecoder 的 decode 方法，具体代码如下：

```
1 let jsonData = jsonString.data(using: .utf8)!
2 let decoder = JSONDecoder()
3 let jsonModel = try! decoder.decode(H5Editor.self, from: jsonData)
4
```

上面代码中的，H5Editor 是一个结构体，能够记录 JSON 解析后的字典和数组。H5Editor 结构体完整定义，请点击[这里的链接](#)。

那么，JSONDecoder 的 decode 方法到底是怎么解析 JSON 的呢？在我看来，了解这一过程的最好方式，就是直接看看它在 Swift 源码里是怎么实现的。

### JSONDecoder 如何解析 JSON？

JSONDecoder 的代码，你可以在[在 Swift 的官方 GitHub 上](#)查看。

接下来，我先跟你说下解析 JSON 的入口，JSONDecoder 的 decode 方法。下面是 decode 方法的定义代码：

```
1 open func decode<T : Decodable>(_ type: T.Type, from data: Data) throws -> T {
2   let topLevel: Any
3   do {
4     topLevel = try JSONSerialization.jsonObject(with: data)
5   } catch {
6     throw DecodingError.dataCorrupted(DecodingError.Context(codingPath: [],
7   }
8   // JSONDecoder 的初始化
9   let decoder = JSONDecoder(referencing: topLevel, options: self.options)
10  // 从顶层开始解析
11  guard let value = try decoder.unbox(topLevel, as: type) else {
12    throw DecodingError.valueNotFound(type, DecodingError.Context(codingPath:
13  }
14
15  return value
16 }
17
```

接下来，我们通过上面的代码一起来看看 decode 方法是如何解析 JSON 的。

上面 decode 方法入参 T.type 的 T 是一个泛型，具体到解析 H5 页面编辑器生成的 JSON 的例子，就是 H5Editor 结构体；入参 data 就是 JSON 字符串转成的 Data 数据。

decode 方法在解析完后会将解析到的数据保存到传入的结构体中，然后返回。在 decode 方法里可以看到，对于传入的 Data 数据会首先通过 JSONSerialization 方法转化成 topLevel 原生对象，然后 topLevel 原生对象通过 JSONDecoder 初始化成一个 JSONDecoder 对象，最后使用 JSONDecoder 的 unbox 方法将数据和传入的结构体对应上，并保存在结构体里进行返回。

可以看出，目前 JSONSerialization 已经能够很好地解析 JSON，JSONDecoder 将其包装以后，通过 unbox 方法使得 JSON 解析后能很方便地匹配 JSON 数据结构和 Swift 原生结构体。

试想一下，如果要将 JSON 应用到更大的场景时，比如对编程语言的描述或者界面布局的描述，其生成的 JSON 文件可能会很大，并且对这种大 JSON 文件解析性能的要求也会更高。那么，有比 JSONSerialization 性能更好的解析 JSON 的方法吗？

### 提高 JSON 解析性能

2019 年 2 月，Geoff Langdale 和 Daniel Lemire 发布了 [simdjson](#)。simdjson 是一款他们研究了很久的快速 JSON 解析器，号称每秒可解析千兆字节 JSON 文件。simdjson 和其他 JSON 解析器对比如下图所示：

parser	GB/s
simdjson	2.2
RapidJSON encoding-validation	0.51
RapidJSON encoding-validation, insitu	0.71
sajson (insitu, dynamic)	0.70
sajson (insitu, static)	0.97
dropbox	0.14
fastjson	0.26
gason	0.85
ultrajson	0.42
jsmn	0.28
cJSON	0.34

可以看出，只有 simdjson 能够达到每秒千兆字节级别，并且远远高于其他 JSON 解析器。那么，simdjson 是怎么做到的呢？接下来，我通过 simdjson 解析 JSON 的两个阶段来跟你说明下这个问题。

**第一个阶段**，使用 simdjson 去发现需要 JSON 里重要的字符集，比如大括号、中括号、逗号、冒号等，还有类似 true、false、null、数字这样的原子字符集。第一个阶段是没有分支处理的，这个阶段与词法分析非常类似。

**第二个阶段**，simdjson 也没有做分支处理，而是采用的堆栈结构，嵌套关系使用 goto 的方式进行导航。simdjson 通过索引可以处理所有输入的 JSON 内容而无需使用分支，这都归功于聪明的条件移动操作，使得遍历过程变得高效了很多。

通过 simdjson 解析 JSON 的两个阶段可以看出，simdjson 的主要思路是尽可能地以最高效的方式将 JSON 这种可读性高的数据格式转换为计算机能更快理解的数据格式。

为了达到快速解析的目的，simdjson 在第一个阶段一次性使用了 64 字节输入进行大规模的数据操作，检查字符和符号类时以及当获得掩码应用变换时以 64 位进行位操作。对于大的 JSON 数据解析性能提升是非常明显的。

如果你想更详细地了解这两个阶段的解析思路，可以查看这篇论文[“Parsing Gigabytes of JSON per Second”](#)。其实，simdjson 就是对这篇论文的实现，你可以在[GitHub](#)上查看具体的实现代码。在我看来，一边看论文，一边看对应的代码实现，不失为一种高效的学习方式。

而如果你想要在工程中使用 simdjson 的话，直接使用它提供的的一个简单接口即可。具体的使用代码如下：

```
1 #include "simdjson/jsonparser.h"
2
3 //...
4
5 const char * filename = ... // JSON 文件
6 std::string_view p = get_corpus(filename);
7 ParsedJson pj = build_parsed_json(p); // 解析方法
8 // you no longer need p at this point, can do aligned_free((void*)p.data())
9 if( ! pj.isValid() ) {
10   // 出错处理
11 }
```