手把手教你实现一个 JSON 解析器！ 转载

Java技术栈 2021-05-25 21:18:18 ©著作权

文章标签 JSON JSON学习 文章分类 其他 编程语言 阅读数 116

1. 背景

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。相对于另一种数据交换格式 XML，JSON 有着诸多优点。比如易读性更好，占用空间更少等。

在 web 应用开发领域内，得益于 JavaScript 对 JSON 提供的良好支持，JSON 要比 XML 更受开发人员青睐。所以作为开发人员，如果有兴趣的话，还是应该深入了解一下 JSON 相关的知识。

本着探究 JSON 原理的目的，我将会在这篇文章中详细向大家介绍一个简单的JSON解析器的解析流程和实现细节。

由于 JSON 本身比较简单，解析起来也并不复杂。所以如果大家感兴趣的话，在看完本文后，不妨自己动手实现一个 JSON 解析器。

好了，其他的话就不多说了，接下来让我们移步到重点章节吧。

2. JSON 解析器实现原理

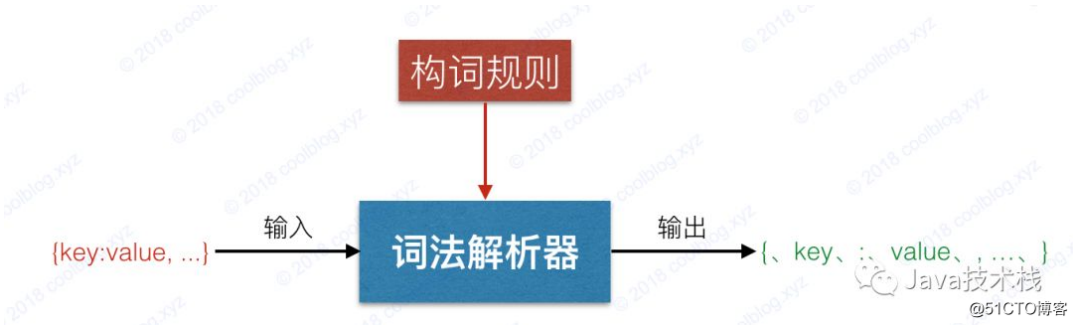
JSON 解析器从本质上来说就是根据 JSON 文法规则创建的状态机，输入是一个 JSON 字符串，输出是一个 JSON 对象。一般来说，解析过程包括词法分析和语法分析两个阶段。

词法分析阶段的目标是按照构词规则将 JSON 字符串解析成 Token 流，比如有如下的 JSON 字符串：

```
{
    "name" : "小明",
    "age": 18
}
```

结果词法分析后，得到一组 Token，如下：

```
{、 name、 :、 小明、 ,、 age、 :、 18、 }
```



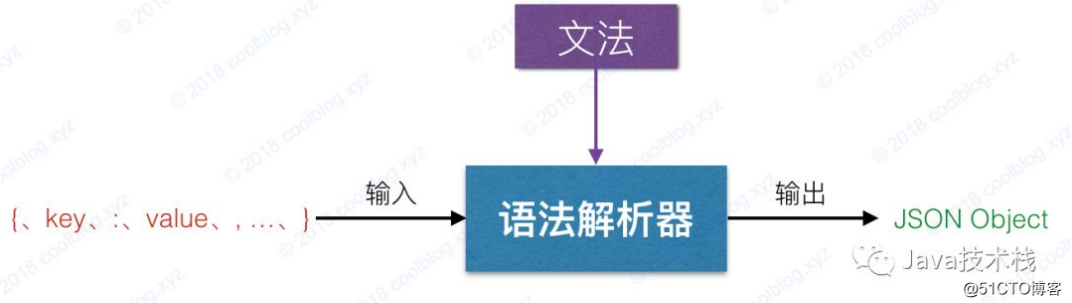
词法分析解析出 Token 序列后，接下来要进行语法分析。语法分析的目的是根据 JSON 文法检查上面 Token 序列所构成的 JSON 结构是否合法。

比如 JSON 文法要求非空 JSON 对象以键值对的形式出现，形如 `object = {string : value}` 。如果传入了一个格式错误的字符串，比如：

```
{
    "name", "小明"
}
```

那么在语法分析阶段，语法分析器分析完 Token name后，认为它是一个符合规则的 Token，并且认为它是一个键。 请不要在 JDK 7+ 中使用这个 JSON 包了！ 这篇看下。

接下来，语法分析器读取下一个 Token，期望这个 Token 是 `:` 。但当它读取了这个 Token，发现这个 Token 是 `,` ，并非其期望的 `:` ，于是文法分析器就会报错误。



这里简单总结一下上面两个流程，词法分析是将字符串解析成一组 Token 序列，而语法分析则是检查输入的 Token 序列所构成的 JSON 格式是否合法。这里大家对 JSON 的解析流程有个印象就好，接下来我会详细分析每个流程。

2.1 词法分析

在本章开始，我说了词法解析的目的，即按照“构词规则”将 JSON 字符串解析成 Token 流。请注意双引号引起来词--构词规则，所谓构词规则是指词法分析模块在将字符串解析成 Token 时所参考的规则。
在 JSON 中，构词规则对应于几种数据类型，当词法解析器读入某个词，且这个词类型符合 JSON 所规定的数据类型时，词法分析器认为这个词符合构词规则，就会生成相应的 Token。
这里我们可以参考 http://www.json.org/ 对 JSON 的定义，罗列一下 JSON 所规定的数据类型：

- BEGIN_OBJECT（{）
- END_OBJECT（}）
- BEGIN_ARRAY（[）
- END_ARRAY（]）
- NULL（null）
- NUMBER（数字）
- STRING（字符串）
- BOOLEAN（true/false）
- SEP_COLON（:）
- SEP_COMMA（,）

当词法分析器读取的词是上面类型中的一种时，即可将其解析成一个 Token。我们可以定义一个枚举类来表示上面的数据类型，如下：

```
1.  public enum TokenType {
2.      BEGIN_OBJECT(1),
3.      END_OBJECT(2),
4.      BEGIN_ARRAY(4),
5.      END_ARRAY(8),
6.      NULL(16),
7.      NUMBER(32),
8.      STRING(64),
9.      BOOLEAN(128),
10.     SEP_COLON(256),
11.     SEP_COMMA(512),
12.     END_DOCUMENT(1024);
13.
14.     TokenType(int code) {
15.         this.code = code;
16.     }
17.
18.     private int code;
19.
20.     public int getTokenCode() {
21.         return code;
22.     }
23. }
```

在解析过程中，仅有 TokenType 类型还不行。我们除了要将某个词的类型保存起来，还需要保存这个词的字面量。所以，所以这里还需要定义一个 Token 类。用于封装词类型和字面量，如下：

```
1.  public class Token {
2.      private TokenType tokenType;
3.      private String value;
```

```
        // 省略不重要的代码
    }
```

定义好了 Token 类，接下来再来定义一个读取字符串的类。如下：

```
    public CharReader(Reader reader) {
            this.reader = reader;
            buffer = new char[BUFFER_SIZE];
        }

        /**
         * 返回 pos 下标处的字符，并返回
         * @return
         * @throws IOException
         */
        public char peek() throws IOException {
            if (pos - 1 >= size) {
                return (char) -1;
            }

            return buffer[Math.max(0, pos - 1)];
        }

        /**
         * 返回 pos 下标处的字符，并将 pos + 1，最后返回字符
         * @return
         * @throws IOException
         */
        public char next() throws IOException {
            if (!hasMore()) {
                return (char) -1;
            }

            return buffer[pos++];
        }

        public void back() {
            pos = Math.max(0, --pos);
        }

        public boolean hasMore() throws IOException {
            if (pos < size) {
                return true;
            }

            fillBuffer();
            return pos < size;
        }

        void fillBuffer() throws IOException {
            int n = reader.read(buffer);
            if (n == -1) {
                return;
            }

            pos = 0;
            size = n;
        }
    }
```

有了 TokenType、Token 和 CharReader 这三个辅助类，接下来我们就可以实现词法解析器了。

```
    public class Tokenizer {
        private CharReader charReader;
        private TokenList tokens;

        public TokenList tokenize(CharReader charReader) throws IOException {
            this.charReader = charReader;
            tokens = new TokenList();
            tokenize();
```

```
9.
10.         return tokens;
11.     }
12.
13.     private void tokenize() throws IOException {
14.         // 使用do-while处理空文件
15.         Token token;
16.         do {
17.             token = start();
18.             tokens.add(token);
19.         } while (token.getTokenType() != TokenType.END_DOCUMENT);
20.     }
21.
22.     private Token start() throws IOException {
23.         char ch;
24.         for(;;) {
25.             if (!charReader.hasMore()) {
26.                 return new Token(TokenType.END_DOCUMENT, null);
27.             }
28.
29.             ch = charReader.next();
30.             if (!isWhiteSpace(ch)) {
31.                 break;
32.             }
33.         }
34.
35.         switch (ch) {
36.             case '{':
37.                 return new Token(TokenType.BEGIN_OBJECT, String.valueOf(ch));
38.             case '}':
39.                 return new Token(TokenType.END_OBJECT, String.valueOf(ch));
40.             case '[':
41.                 return new Token(TokenType.BEGIN_ARRAY, String.valueOf(ch));
42.             case ']':
43.                 return new Token(TokenType.END_ARRAY, String.valueOf(ch));
44.             case ',':
45.                 return new Token(TokenType.SEP_COMMA, String.valueOf(ch));
46.             case ':':
47.                 return new Token(TokenType.SEP_COLON, String.valueOf(ch));
48.             case 'n':
49.                 return readNull();
50.             case 't':
51.             case 'f':
52.                 return readBoolean();
53.             case '"':
54.                 return readString();
55.             case '-':
56.                 return readNumber();
57.         }
58.
59.         if (isDigit(ch)) {
60.             return readNumber();
61.         }
62.
63.         throw new JsonParseException("Illegal character");
64.     }
65.
66.     private Token readNull() {...}
67.     private Token readBoolean() {...}
68.     private Token readString() {...}
69.     private Token readNumber() {...}
70. }
```

上面的代码是词法分析器的实现，部分代码这里没有贴出来，后面具体分析的时候再贴。

先来看看词法分析器的核心方法 start，这个方法代码量不多，并不复杂。其通过一个死循环不停的读取字符，然后再根据字符的类型，执行不同的解析逻辑。

上面说过，JSON 的解析过程比较简单。原因在于，在解析时，只需通过每个词第一个字符即可判断出这个词的 Token Type。比如：

- 第一个字符是 { 、 } 、 [、] 、 , 、 : ，直接封装成相应的 Token 返回即可
- 第一个字符是 n ，期望这个词是 null ， Token 类型是 NULL
- 第一个字符是 t 或 f ，期望这个词是 true 或者 false ， Token 类型是 BOOLEAN
- 第一个字符是 " ，期望这个词是字符串，Token 类型为 String
- 第一个字符是 0~9 或 - ，期望这个词是数字，类型为 NUMBER

正如上面所说，词法分析器只需要根据每个词的第一个字符，即可知道接下来它所期望读取的到的内容是什么样的。如果满足期望了，则返回 Token，否则返回错误。
下面就来看看词法解析器在碰到第一个字符是n和"时的处理过程。先看碰到字符n的处理过程：

```
1.  private Token readNull() throws IOException {
2.      if (!(charReader.next() == 'u' && charReader.next() == 'l' && charReader.next() == 'l
3.          throw new JsonParseException("Invalid json string");
4.      }
5.
6.      return new Token(TokenType.NULL, "null");
7.  }
```

上面的代码很简单，词法分析器在读取字符n后，期望后面的三个字符分别是u,l,l，与 n 组成词 null。如果满足期望，则返回类型为 NULL 的 Token，否则报异常。readNull 方法逻辑很简单，不多说了。
接下来看看 string 类型的数据处理过程：

```
1.  private Token readString() throws IOException {
2.      StringBuilder sb = new StringBuilder();
3.      for (;;) {
4.          char ch = charReader.next();
5.          // 处理转义字符
6.          if (ch == '\\') {
7.              if (!isEscape()) {
8.                  throw new JsonParseException("Invalid escape character");
9.              }
10.             sb.append('\\');
11.             ch = charReader.peek();
12.             sb.append(ch);
13.             // 处理 Unicode 编码，形如 \u4e2d。且只支持 \u0000 ~ \uFFFF 范围内的编码
14.             if (ch == 'u') {
15.                 for (int i = 0; i < 4; i++) {
16.                     ch = charReader.next();
17.                     if (isHex(ch)) {
18.                         sb.append(ch);
19.                     } else {
20.                         throw new JsonParseException("Invalid character");
21.                     }
22.                 }
23.             }
24.         } else if (ch == '"') { // 碰到另一个双引号，则认为字符串解析结束，返回 Token
25.             return new Token(TokenType.STRING, sb.toString());
26.         } else if (ch == '\r' || ch == '\n') { // 传入的 JSON 字符串不允许换行
27.             throw new JsonParseException("Invalid character");
28.         } else {
29.             sb.append(ch);
30.         }
31.     }
32. }
33.
34. private boolean isEscape() throws IOException {
35.     char ch = charReader.next();
36.     return (ch == '"' || ch == '\\' || ch == 'u' || ch == 'r'
37.                 || ch == 'n' || ch == 'b' || ch == 't' || ch == 'f');
38. }
39.
40. private boolean isHex(char ch) {
41.     return ((ch >= '0' && ch <= '9') || ('a' <= ch && ch <= 'f')
42.             || ('A' <= ch && ch <= 'F'));
43. }
```

string 类型的数据解析起来要稍微复杂一些，主要是需要处理一些特殊类型的字符。JSON 所允许的特殊类型的字符如下：

```
\"
\
\b
\f
\n
\r
\t
\u four-hex-digits
\/
```

最后一种特殊字符 \/ 代码中未做处理，其他字符均做了判断，判断逻辑在 isEscape 方法中。在传入 JSON 字符串中，仅允许字符串包含上面所列的转义字符。如果乱传转义字符，解析时会报错。
对于 STRING 类型的词，解析过程始于字符 " ，也终于 " 。所以在解析的过程中，当再次遇到字符 " ，readString 方法会认为本次的字符串解析过程结束，并返回相应类型的 Token。
上面说了 null 类型和 string 类型的数据解析过程，过程并不复杂，理解起来应该不难。至于 boolean 和 number 类型的数据解析过程，大家有兴趣的话可以自己看源码，这里就不在说了。

2.2 语法分析

当词法分析结束后，且分析过程中没有抛出错误，那么接下来就可以进行语法分析了。语法分析过程以词法分析阶段解析出的 Token 序列作为输入，输出 JSON Object 或 JSON Array。
语法分析器的实现的文法如下：

```
object = {} | { members }
members = pair | pair , members
pair = string : value
array = [] | [ elements ]
elements = value | value , elements
value = string | number | object | array | true | false | null
```

语法分析器的实现需要借助两个辅助类，也就是语法分析器的输出类，分别是 JsonObject 和 JsonArray。Java常用的几个Json库，性能强势对比！这篇推荐看下。
代码如下：

```
public class JsonObject {

    private Map<String, Object> map = new HashMap<String, Object>();

    public void put(String key, Object value) {
        map.put(key, value);
    }

    public Object get(String key) {
        return map.get(key);
    }

    public List<Map.Entry<String, Object>> getAllKeyValue() {
        return new ArrayList<>(map.entrySet());
    }

    public JsonObject getJsonObject(String key) {
        if (!map.containsKey(key)) {
            throw new IllegalArgumentException("Invalid key");
        }

        Object obj = map.get(key);
        if (!(obj instanceof JsonObject)) {
            throw new JsonTypeException("Type of value is not JsonObject");
        }

        return (JsonObject) obj;
    }

```

```
    public JsonArray getJsonArray(String key) {
        if (!map.containsKey(key)) {
            throw new IllegalArgumentException("Invalid key");
        }

        Object obj = map.get(key);
        if (!(obj instanceof JsonArray)) {
            throw new JsonTypeException("Type of value is not JsonArray");
        }

        return (JsonArray) obj;
    }

    @Override
    public String toString() {
        return BeautifyJsonUtils.beautify(this);
    }
}

public class JsonArray implements Iterable {

    private List list = new ArrayList();

    public void add(Object obj) {
        list.add(obj);
    }

    public Object get(int index) {
        return list.get(index);
    }

    public int size() {
        return list.size();
    }

    public JsonObject getJsonObject(int index) {
        Object obj = list.get(index);
        if (!(obj instanceof JsonObject)) {
            throw new JsonTypeException("Type of value is not JsonObject");
        }

        return (JsonObject) obj;
    }

    public JsonArray getJsonArray(int index) {
        Object obj = list.get(index);
        if (!(obj instanceof JsonArray)) {
            throw new JsonTypeException("Type of value is not JsonArray");
        }

        return (JsonArray) obj;
    }

    @Override
    public String toString() {
        return BeautifyJsonUtils.beautify(this);
    }

    public Iterator iterator() {
        return list.iterator();
    }
}
```

语法解析器的核心逻辑封装在了 parseJsonObject 和 parseJsonArray 两个方法中，接下来我会详细分析 parseJsonObject 方法，parseJsonArray 方法大家自己分析吧。

parseJsonObject 方法实现如下：

```
private JsonObject parseJsonObject() {
    JsonObject jsonObject = new JsonObject();
```

```
        int expectToken = STRING_TOKEN | END_OBJECT_TOKEN;
        String key = null;
        Object value = null;
        while (tokens.hasMore()) {
            Token token = tokens.next();
            TokenType tokenType = token.getTokenType();
            String tokenValue = token.getValue();
            switch (tokenType) {
            case BEGIN_OBJECT:
                checkExpectToken(tokenType, expectToken);
                jsonObject.put(key, parseJsonObject()); // 递归解析 json object
                expectToken = SEP_COMMA_TOKEN | END_OBJECT_TOKEN;
                break;
            case END_OBJECT:
                checkExpectToken(tokenType, expectToken);
                return jsonObject;
            case BEGIN_ARRAY: // 解析 json array
                checkExpectToken(tokenType, expectToken);
                jsonObject.put(key, parseJsonArray());
                expectToken = SEP_COMMA_TOKEN | END_OBJECT_TOKEN;
                break;
            case NULL:
                checkExpectToken(tokenType, expectToken);
                jsonObject.put(key, null);
                expectToken = SEP_COMMA_TOKEN | END_OBJECT_TOKEN;
                break;
            case NUMBER:
                checkExpectToken(tokenType, expectToken);
                if (tokenValue.contains(".") || tokenValue.contains("e") || tokenValue.contai
                    jsonObject.put(key, Double.valueOf(tokenValue));
                } else {
                    Long num = Long.valueOf(tokenValue);
                    if (num > Integer.MAX_VALUE || num < Integer.MIN_VALUE) {
                        jsonObject.put(key, num);
                    } else {
                        jsonObject.put(key, num.intValue());
                    }
                }
                expectToken = SEP_COMMA_TOKEN | END_OBJECT_TOKEN;
                break;
            case BOOLEAN:
                checkExpectToken(tokenType, expectToken);
                jsonObject.put(key, Boolean.valueOf(token.getValue()));
                expectToken = SEP_COMMA_TOKEN | END_OBJECT_TOKEN;
                break;
            case STRING:
                checkExpectToken(tokenType, expectToken);
                Token preToken = tokens.peekPrevious();
                /*
                 * 在 JSON 中，字符串既可以作为键，也可作为值。
                 * 作为键时，只期待下一个 Token 类型为 SEP_COLON。
                 * 作为值时，期待下一个 Token 类型为 SEP_COMMA 或 END_OBJECT
                 */
                if (preToken.getTokenType() == TokenType.SEP_COLON) {
                    value = token.getValue();
                    jsonObject.put(key, value);
                    expectToken = SEP_COMMA_TOKEN | END_OBJECT_TOKEN;
                } else {
                    key = token.getValue();
                    expectToken = SEP_COLON_TOKEN;
                }
                break;
            case SEP_COLON:
                checkExpectToken(tokenType, expectToken);
                expectToken = NULL_TOKEN | NUMBER_TOKEN | BOOLEAN_TOKEN | STRING_TOKEN
                        | BEGIN_OBJECT_TOKEN | BEGIN_ARRAY_TOKEN;
                break;
            case SEP_COMMA:
                checkExpectToken(tokenType, expectToken);
                expectToken = STRING_TOKEN;
```

```
            break;
        case END_DOCUMENT:
            checkExpectToken(tokenType, expectToken);
            return jsonObject;
        default:
            throw new JsonParseException("Unexpected Token.");
        }
    }

    throw new JsonParseException("Parse error, invalid Token.");
}

private void checkExpectToken(TokenType tokenType, int expectToken) {
    if ((tokenType.getTokenCode() & expectToken) == 0) {
        throw new JsonParseException("Parse error, invalid Token.");
    }
}
```

parseJsonObject 方法解析流程大致如下：

1. 读取一个 Token，检查这个 Token 是否是其所期望的类型
2. 如果是，更新期望的 Token 类型。否则，抛出异常，并退出
3. 重复步骤1和2，直至所有的 Token 都解析完，或出现异常

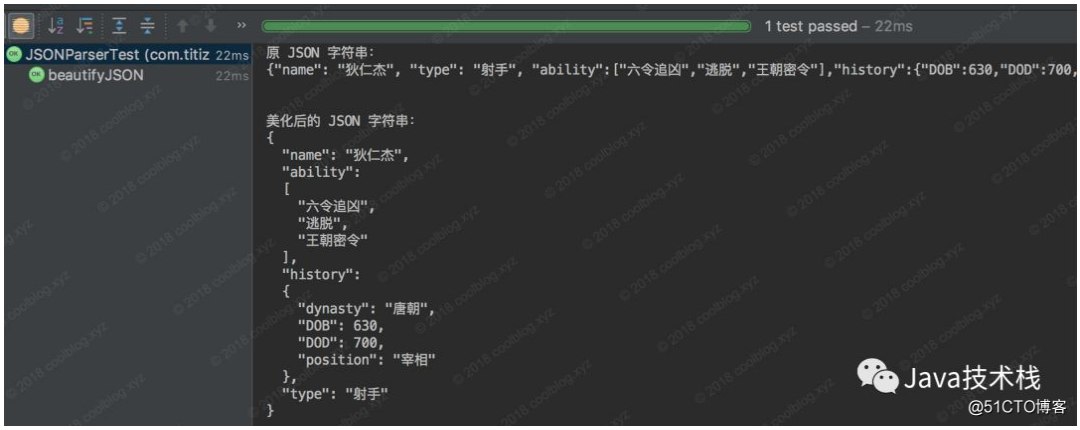
上面的步骤并不复杂，但有可能不好理解。这里举个例子说明一下，有如下的 Token 序列：

```
{、 id、 :、 1、 }
```

parseJsonObject 解析完 `{` Token 后，接下来它将期待 STRING 类型的 Token 或者 END_OBJECT 类型的 Token 出现。于是 parseJsonObject 读取了一个新的 Token，发现这个 Token 的类型是 STRING 类型，满足期望。
于是 parseJsonObject 更新期望Token 类型为 SEL_COLON，即 `:` 。如此循环下去，直至 Token 序列解析结束或者抛出异常退出。
上面的解析流程虽然不是很复杂，但在具体实现的过程中，还是需要注意一些细节问题。比如：
在 JSON 中，字符串既可以作为键，也可以作为值。作为键时，语法分析器期待下一个 Token 类型为 SEP_COLON。而作为值时，则期待下一个 Token 类型为 SEP_COMMA 或 END_OBJECT。
所以这里要判断该字符串是作为键还是作为值，判断方法也比较简单，即判断上一个 Token 的类型即可。如果上一个 Token 是 SEP_COLON，即 `:` ，那么此处的字符串只能作为值了。否则，则只能做为键。
对于整数类型的 Token 进行解析时，简单点处理，可以直接将该整数解析成 Long 类型。但考虑到空间占用问题，对于 `[Integer.MIN_VALUE, Integer.MAX_VALUE]` 范围内的整数来说，解析成 Integer 更为合适，所以解析的过程中也需要注意一下。

3. 测试及效果展示

为了验证代码的正确性，这里对代码进行了简单的测试。测试数据来自网易音乐，大约有4.5W个字符。为了避免每次下载数据，因数据发生变化而导致测试不通过的问题。
我将某一次下载的数据保存在了 music.json 文件中，后面每次测试都会从文件中读取数据。
关于测试部分，这里就不贴代码和截图了。大家有兴趣的话，可以自己下载源码测试玩玩。
测试就不多说了，接下来看看 JSON 美化效果展示。这里随便模拟点数据，就模拟王者荣耀里的狄仁杰英雄信息吧（对，这个英雄我经常用）。如下图：



关于 JSON 美化的代码这里也不讲解了，并非重点，只算一个彩蛋吧。

4. 写作最后

到此，本文差不多要结束了。本文对应的代码已经放到了 github 上，需要的话，大家可自行下载：https://github.com/code4wt/JSONParser。

这里需要声明一下，本文对应的代码实现了一个比较简陋的 JSON 解析器，实现的目的是探究 JSON 的解析原理。JSONParser 只算是一个练习性质的项目，代码实现的并不优美，而且缺乏充足的测试。

同时，限于本人的能力（编译原理基础基本可以忽略），我并无法保证本文以及对应的代码中不出现错误。如果大家在阅读代码的过程中，发现了一些错误，或者写的不好的地方，可以提出来，我来修改。如果这些错误对你造成了困扰，这里先说一声很抱歉。

最后，本文及实现主要参考了一起写一个JSON解析器和如何编写一个JSON解析器两篇文章及两篇文章对应的实现代码，在这里向着两篇博文的作者表示感谢。好了，本文到此结束，祝大家生生活愉快！再见。

作者：田小波

www.cnblogs.com/nullllun/p/8358146.html

参考

> 一起写一个JSON解析器
>
> 如何编写一个JSON解析器
> https://www.liaoxuefeng.com/article/994977272296736
> 介绍JSON
> http://json.org/json-zh.html
> 写一个 JSON、XML 或 YAML 的 Parser 的思路是什么？www.zhihu.com/question/24640264/answer/80500016

推荐阅读

1. Java JVM、集合、多线程、新特性系列教程
2. Spring MVC、Spring Boot、Spring Cloud 系列教程
3. Maven、Git、Eclipse、Intellij IDEA 系列工具教程
4. Java、后端、架构、阿里巴巴等大厂最新面试题