JSON (JavaScript Object Notation) 指南

1. JSON简介

JSON是一种轻量级的数据交换格式，易于人阅读和编写，同时也易于机器解析和生成。它基于JavaScript编程语言的一个子集，但是它的使用不限于JavaScript。

2. JSON的基本数据类型

JSON支持以下数据类型：

- 字符串（String）
- 数字（Number）
- 布尔值（Boolean）
- 空值（null）
- 对象（Object）
- 数组（Array）

3. JSON语法规则

3.1 基本语法

- 数据以键值对的形式存在
- 数据由逗号分隔
- 大括号保存对象
- 方括号保存数组

3.2 具体示例

字符串

```
{
    "name": "张三",
    "description": "这是一个字符串"
}
```

数字

```
{
    "age": 25,
    "price": 99.99,
    "temperature": -10
}
```

布尔值

```
{
    "isStudent": true,
    "isWorking": false
}
```

空值

```
{
    "middleName": null
}
```

对象

```
{
    "person": {
        "name": "李四",
        "age": 30,
        "address": {
            "city": "北京",
            "street": "朝阳区"
        }
    }
}
```

数组

```
{
    "fruits": ["苹果", "香蕉", "橙子"],
    "numbers": [1, 2, 3, 4, 5],
    "mixed": [1, "hello", true, null]
}
```

4. JSON的常见用途

1. 配置文件
2. API数据交换
3. 数据存储
4. 前后端通信

5. JSON的注意事项

1. JSON中的字符串必须使用双引号（"）而不是单引号（'）
2. JSON不支持注释
3. JSON不支持尾随逗号
4. JSON不支持JavaScript中的undefined
5. JSON中的键名必须是字符串

6. JSON的验证

可以使用在线工具或编程语言的JSON解析器来验证JSON格式是否正确。常见的验证工具包括：

- JSONLint
- JSON Schema Validator
- 各种编程语言的内置JSON解析器

7. JSON与XML的比较

优点：

- 更简洁的语法
- 更小的数据体积
- 更容易解析
- 更好的可读性

缺点：

- 不支持注释
- 不支持命名空间
- 不支持属性

8. 实际应用示例

8.1 用户信息

```
{
    "user": {
        "id": 1001,
        "username": "zhangsan",
        "email": "zhangsan@example.com",
        "isActive": true,
        "roles": ["user", "editor"],
        "profile": {
            "firstName": "张",
            "lastName": "三",
            "age": 28,
            "address": {
                "city": "上海",
                "zipCode": "200000"
            }
        }
```

```
    }
}
```

8.2 配置信息

```
{
    "app": {
        "name": "MyApp",
        "version": "1.0.0",
        "settings": {
            "theme": "dark",
            "language": "zh-CN",
            "notifications": true,
            "maxConnections": 100
        }
    }
}
```

9. 编程语言中的JSON处理

大多数现代编程语言都提供了JSON的解析和生成功能：

Python示例

```
import json

# 将Python对象转换为JSON
data = {
    "name": "张三",
    "age": 25
}
json_str = json.dumps(data)

# 将JSON转换为Python对象
json_str = '{"name": "张三", "age": 25}'
data = json.loads(json_str)
```

JavaScript示例

```
// 将JavaScript对象转换为JSON
const data = {
    name: "张三",
    age: 25
};
const jsonStr = JSON.stringify(data);
```

```
// 将JSON转换为JavaScript对象
const jsonStr = '{"name": "张三", "age": 25}';
const data = JSON.parse(jsonStr);
```

10. C++中的JSON处理

在C++中处理JSON，最常用的是nlohmann/json库，这是一个现代C++的JSON解析库，使用简单且功能强大。

10.1 安装和配置

```
// 使用vcpkg安装
vcpkg install nlohmann-json

// 使用CMake配置
find_package(nlohmann_json REQUIRED)
target_link_libraries(your_target PRIVATE nlohmann_json::nlohmann_json)
```

10.2 基本用法示例

创建JSON对象

```
#include <nlohmann/json.hpp>
using json = nlohmann::json;

// 创建JSON对象
json j = {
    {"name", "张三"},
    {"age", 25},
    {"is_student", true},
    {"scores", {90, 85, 95}},
    {"address", {
        {"city", "北京"},
        {"street", "朝阳区"}
    }}
};
```

访问JSON数据

```
// 访问基本类型
std::string name = j["name"];  // 获取字符串
int age = j["age"];            // 获取数字
bool is_student = j["is_student"];  // 获取布尔值

// 访问数组
int first_score = j["scores"][0];  // 获取数组第一个元素
```

```
// 访问嵌套对象
std::string city = j["address"]["city"];  // 获取嵌套对象的值
```

修改JSON数据

```
// 修改现有值
j["age"] = 26;
j["scores"][0] = 95;

// 添加新值
j["email"] = "zhangsan@example.com";
j["hobbies"] = {"读书", "运动", "音乐"};
```

JSON序列化和反序列化

```
// 将JSON对象转换为字符串
std::string json_str = j.dump();  // 紧凑格式
std::string pretty_str = j.dump(4);  // 美化格式，缩进4个空格

// 从字符串解析JSON
json j2 = json::parse(json_str);

// 从文件读取JSON
std::ifstream file("data.json");
json j3;
file >> j3;

// 将JSON写入文件
std::ofstream out("output.json");
out << std::setw(4) << j << std::endl;
```

10.3 高级特性

类型检查

```
// 检查JSON值的类型
if (j["age"].is_number()) {
    // 处理数字类型
}
if (j["name"].is_string()) {
    // 处理字符串类型
}
if (j["scores"].is_array()) {
    // 处理数组类型
}
if (j["address"].is_object()) {
```

```
    // 处理对象类型
}
```

异常处理

```
try {
    json j = json::parse(json_str);
} catch (json::parse_error& e) {
    std::cout << "解析错误: " << e.what() << std::endl;
}

try {
    int age = j["age"];
} catch (json::type_error& e) {
    std::cout << "类型错误: " << e.what() << std::endl;
}
```

自定义类型转换

```
// 定义自定义类型
struct Person {
    std::string name;
    int age;
};

// 实现JSON转换
void to_json(json& j, const Person& p) {
    j = json{
        {"name", p.name},
        {"age", p.age}
    };
}

void from_json(const json& j, Person& p) {
    j.at("name").get_to(p.name);
    j.at("age").get_to(p.age);
}

// 使用示例
Person person{"张三", 25};
json j = person;  // 自动转换为JSON
Person p2 = j.get<Person>();  // 从JSON转换回Person对象
```

10.4 性能优化建议

1. 使用引用避免复制

```
const json& j_ref = j;  // 使用引用访问JSON对象
```

2. 预分配空间

```
json j;
j["array"] = json::array();
j["array"].reserve(1000);  // 预分配空间
```

3. 使用移动语义

```
json j1 = std::move(j2);  // 使用移动构造
```

10.5 其他C++ JSON库

除了nlohmann/json，还有其他可选的JSON库：

1. RapidJSON
    - 高性能
    - 支持DOM和SAX风格的API
    - 内存效率高
2. JsonCpp
    - 成熟稳定
    - 支持C++98
    - 文档完善
3. Boost.JSON
    - Boost库的一部分
    - 与Boost生态系统集成
    - 性能优秀

选择建议：

- 如果需要现代C++特性和易用性，推荐使用nlohmann/json
- 如果追求极致性能，可以考虑RapidJSON
- 如果项目已经使用Boost，可以考虑Boost.JSON
- 如果需要支持旧版C++，可以考虑JsonCpp