Python 的 orjson 库

2025年4月22日　2 分钟阅读

#Python　#orjson

> *Python 的 orjson 库*

我们来详细探讨一下 `orjson` 库以及为什么它通常比 Python 标准库中的 `json` 模块性能更好。

首先，需要澄清一点：**`orjson` 并不是 Python 3.11 或任何 Python 版本的标准库组成部分。** 它是一个非常流行的、高性能的第三方 Python 库，需要单独安装（通常通过 `pip install orjson`）。Python 标准库中用于处理 JSON 的模块仍然是 `json`。

尽管如此，`orjson` 因其出色的性能而广受青睐，尤其是在需要高速处理大量 JSON 数据的场景中。它之所以更快，主要得益于以下几个关键因素：

底层实现语言 (Rust)

`orjson` 的核心部分是使用 **Rust** 语言编写的。Rust 是一种现代的、内存安全的系统编程语言，以其接近 C/C++ 的高性能而闻名。

相比之下，Python 标准库的 `json` 模块主要是用 Python 编写的，虽然它也包含一个 C 语言编写的加速器（`_json`），但在许多操作上，专门为性能优化的 Rust 代码通常能实现更高的执行效率。Rust 的编译时优化、对内存布局的精细控制以及避免 Python 解释器开销的能力，都对性能提升有显著贡献。

优化的序列化和反序列化算法

`orjson` 在将 Python 对象序列化为 JSON 字符串（`dumps`）和将 JSON 字符串反序列化为 Python 对象（`loads`）的过程中，采用了高度优化的算法。

它可能利用了更快的字符串处理技术、更高效的内存管理策略，并可能针对现代 CPU 架构（如 SIMD 指令）进行了优化，以加速解析和生成过程。

原生支持更多数据类型

标准 `json` 库仅原生支持基本的 Python 类型（如 `dict`, `list`, `str`, `int`, `float`, `bool`, `None`）。对于其他类型（如 `datetime`, `uuid`, `bytes`, `dataclasses`, `enums` 等），你需要提供自定义的 `default` 函数（用于序列化）或 `object_hook`（用于反序列化）。这些自定义函数的调用会增加额外的开销。

目录

文章信息

字数

阅读时间

发布时间

更新时间

标签

#Python　#orjson

`orjson` **内置了对多种常用但非 JSON 原生支持的 Python 类型的快速序列化支持**，包括 `datetime.datetime`, `uuid.UUID`, `int` (任意大小), `float` (包括 `NaN`, `Infinity`, `-Infinity`), `str` (正确处理 UTF-8), `bytes`, `bytearray`, `memoryview`, `dataclasses`, `enums`, `decimal.Decimal` 等。这种原生支持避免了 Python 层面的回调开销，直接在 Rust 代码中高效处理，从而显著提高了涉及这些类型的 JSON 操作速度。

严格遵循 JSON 规范 (RFC 8259)

`orjson` 严格遵守 JSON 规范。这意味着它在解析时可能比标准库更严格，不会接受一些格式略有问题的 JSON。虽然这在某些情况下可能需要你确保输入是完全标准的 JSON，但这种严格性也意味着它可以进行更直接、更少分支判断的解析，有助于提高速度。

它保证输出的 JSON 是符合 RFC 8259 规范的有效 UTF-8 编码字符串。

专注于性能和正确性

`orjson` 的设计目标就是提供一个既正确又快速的 JSON 库。它在实现上可能做出了一些有利于性能的权衡（例如，可能不如标准库那样灵活地处理某些边缘情况或非标准格式），并将性能优化置于非常高的优先级。

总结

总的来说，`orjson` 的高性能主要来源于：

使用 Rust 编写核心逻辑，利用了编译语言的性能优势和内存效率。

采用高度优化的序列化/反序列化算法。

内置对多种 Python 类型的原生快速支持，减少了 Python 回调开销。

严格遵循 JSON 规范，可能带来更直接高效的实现路径。

因此，如果你的应用对 JSON 处理的性能有较高要求（例如，Web 框架的请求/响应处理、数据处理管道、API 交互等），使用 `orjson` 替换标准 `json` 库通常能带来显著的速度提升。

如何使用 `orjson`？

安装:

```
pip install orjson
```

使用: 它的 API 设计与标准 `json` 库非常相似，通常可以轻松替换：

```python
import orjson
import datetime
import uuid
from dataclasses import dataclass

@dataclass
class User:
    id: int
    name: str
    registered_at: datetime.datetime
    api_key: uuid.UUID

user = User(
    id=1,
    name="Alice",

registered_at=datetime.datetime.now(datetime.timezone.utc),
    api_key=uuid.uuid4()
)

# 序列化 (dumps)
# 注意：orjson.dumps 返回的是 bytes，而不是 str
json_bytes = orjson.dumps(user, option=orjson.OPT_NAIVE_UTC
| orjson.OPT_PASSTHROUGH_DATACLASS |
orjson.OPT_SERIALIZE_UUID)
print(f"Serialized (bytes): {json_bytes}")

# 如果需要字符串，可以解码
json_string = json_bytes.decode('utf-8')
print(f"Serialized (string): {json_string}")

# 反序列化 (loads)
# orjson.loads 可以接受 bytes 或 str
data_from_bytes = orjson.loads(json_bytes)
print(f"Deserialized from bytes: {data_from_bytes}")

data_from_string = orjson.loads(json_string)
print(f"Deserialized from string: {data_from_string}")

# orjson 也能处理标准类型
basic_data = {"key": "value", "number": 123, "items": [1,
2, None]}
basic_json_bytes = orjson.dumps(basic_data)
print(f"Basic data serialized: {basic_json_bytes}")
basic_deserialized = orjson.loads(basic_json_bytes)
print(f"Basic data deserialized: {basic_deserialized}")
```

注意：`orjson.dumps()` 默认返回 `bytes` 对象，这通常更高效，因为避免了最终的 UTF-8 编码步骤。如果需要 `str`，需要自行解码 (`.decode('utf-8')`)。另外，对于 `datetime`、`dataclass`、`uuid` 等类型的序列化，可能需要通过 `option` 参数启用相应的选项。

这个详细的解释可以帮助理解 `orjson` 的高性能来源！

分享这篇文章