目前 Python 生态中，**没有单一的 JSON 序列化库能完美满足所有“开箱即用”和多类型支持的需求。** `orjson` **在性能和部分内置类型（如** `datetime` **、** `numpy` **数组）支持上表现优异，但处理自定义类、** `pandas` **等类型仍需结合自定义** `default` **函数。** `jsonpickle` **能处理复杂对象，但其默认输出包含额外类型信息，即使使用** `unpicklable=False` **，对** `numpy` **和** `pandas` **的支持也可能不理想。**自定义 `JSONEncoder` 提供了灵活性，但需要用户编写代码。最佳方案通常是根据具体需求，选择高性能库并辅以定制化序列化逻辑。

Python JSON 序列化器深度调研：开箱即用与多类型支持

1. 引言：Python JSON 序列化的挑战与需求

1.1. 用户对“开箱即用”和多类型支持的需求

用户对 Python 中的 JSON 序列化器提出了明确且具有挑战性的需求，核心在于**“开箱即用”和对“各种数据类型”的广泛支持**。具体而言，“开箱即用”期望库本身内置丰富的默认序列化规则，能够直接处理常见的 Python 数据类型，而无需用户进行大量初始配置或编写冗长的自定义代码。同时，用户也要求能够通过简单的配置或少量代码来扩展或自定义对特定数据类型的序列化行为，以适应项目中可能出现的特殊需求。这种平衡预设规则与灵活扩展的能力是“开箱即用”的关键。

在数据类型支持方面，用户的需求覆盖了广泛的 Python 生态系统。首先，**Python 内置的基本数据类型**，如 `int` （整数）、`str` （字符串）、`list` （列表）、`dict` （字典）、`tuple` （元组）、`set` （集合）以及 `datetime` （日期时间）等，是序列化器必须能够原生支持的。其次，对于科学计算和数据分析领域中广泛使用的第三方库，如 `numpy` **的数组和** `pandas` **的** `DataFrame` **或** `Series` **对象**，用户也希望序列化器能够提供便捷的序列化方案。更进一步，**用户自定义的类实例**，即通过 `class` 关键字定义的类所创建的对象，也需要能够被有效地序列化为 JSON。最后，对于一些特定类型的数据，如函数、生成器、文件对象等，用户也期望序列化器能够提供合理的处理方式，即使这意味着将它们序列化为某种占位符或元数据表示。这种对数据类型支持的广泛性要求，对序列化库的设计和实现提出了很高的标准。

1.2. 标准 `json` 模块的局限性

Python 标准库中的 `json` 模块虽然提供了基本的 JSON 编码和解码功能，但在满足用户提出的“开箱即用”和多类型支持需求方面存在显著的局限性。首先， `json` 模块主要针对 Python 的基本数据类型（如字典、列表、字符串、数字、布尔值和 `None` ）进行序列化 。当尝试序列化一些更复杂的 Python 内置类型，例如 `datetime.datetime` 对象时，`json.dumps()` 会直接抛出 `TypeError: Object of type datetime is not JSON`

`serializable` 的错误 。这是因为 JSON 标准本身并不直接支持日期时间类型， `json` 模块也没有内置的机制将其转换为 JSON 兼容的格式（如 ISO 8601 字符串）。

其次，对于第三方库数据类型，如 `numpy` 数组， `json` 模块同样无法直接处理。尝试序列化包含 `numpy.ndarray` 的字典时，会遇到 `TypeError: Object of type ndarray is not JSON serializable` 的错误 。 `numpy` 数组是科学计算中非常常用的数据结构，缺乏对它的原生支持极大地限制了 `json` 模块在这些领域的应用。类似地， `pandas` 的 `DataFrame` 和 `Series` 对象也无法直接被 `json` 模块序列化。

对于用户自定义的类实例， `json` 模块默认情况下也无法直接将其转换为 JSON。它会尝试使用对象的 `__dict__` 属性，但如果对象没有这个属性，或者 `__dict__` 中包含不可序列化的值，序列化就会失败 。虽然可以通过继承 `json.JSONEncoder` 并重写 `default` 方法来实现自定义类的序列化 ，但这需要用户编写额外的代码，不符合“开箱即用”中对默认规则丰富性的要求。

最后，对于一些特定类型，如函数、生成器、文件对象等， `json` 模块完全没有提供默认的序列化方案。这些类型的序列化通常需要用户根据具体需求实现复杂的转换逻辑。因此，尽管 `json` 模块是 Python 处理 JSON 的基础，但其在数据类型支持的广度和默认行为的丰富性上，与用户提出的“开箱即用”和多类型支持的需求存在较大差距。

2. 主流 Python JSON 库概览与比较

为了应对标准 `json` 模块的局限性，社区开发了多种第三方 JSON 库，它们在性能、功能以及对特定数据类型的支持上各有侧重。下表对几种主流库进行了比较：

表格 复制

特性/库	orjson	jsonpickle
主要目标	极致性能，标准兼容	复杂对象图序列
性能	**非常高** (Rust 实现)	相对较慢 (Pytho
内置多类型支持	datetime , numpy (需选项), UUID , dataclass	**几乎所有 Pytho**
自定义类型支持	通过 default 函数	自动处理，可配
输出纯净度	**标准 JSON** (RFC 8259)	默认含额外类型
numpy 支持	**原生支持** (OPT_SERIALIZE_NUMPY)	支持，但默认输
pandas 支持	需通过 default 函数 (如调用 to_dict())	支持，但默认输

特性/库	orjson	jsonpickle
自定义类支持	需通过 `default` 函数（如使用 `__dict__` 或自定义方法）	**原生支持**，自动
输出类型	`bytes`	`str`
安全性	良好	**警告：不应用于**
主要缺点	对部分类型仍需 `default` 函数，返回 `bytes`	默认输出非标准想

Table 1: 主流 Python JSON 库特性对比

2.1. `orjson`：高性能与部分内置类型支持

`orjson` 是一个用 Rust 编写的 Python JSON 库，以其**卓越的性能和对多种 Python 类型的原生支持**而闻名 。根据基准测试，`orjson` 在序列化和反序列化速度上显著优于 Python 标准库的 `json` 模块以及其他一些第三方库如 `ujson` 和 `rapidjson` 。例如，在一个测试中，`orjson` 的序列化时间仅为 `0.417962` 秒，而标准 `json` 库需要 `1.616786` 秒，`ujson` 需要 `1.413367` 秒 。这种性能优势使其成为对 I/O 密集型或大规模数据处理应用有吸引力的选择。

在数据类型支持方面，`orjson` 提供了对多种非标准 Python 类型的开箱即用序列化。它能够原生处理 `datetime.datetime` 、`datetime.date` 和 `datetime.time` 对象，将它们序列化为 RFC 3339 格式的字符串（ISO 8601 的子集）。例如，`datetime.datetime(2018, 12, 1, 2, 3, 4, 9, tzinfo=zoneinfo.ZoneInfo("Australia/Adelaide"))` 会被序列化为 `b'"2018-12-01T02:03:04.000009+10:30"'` 。此外，`orjson` 还支持 `numpy.ndarray` 的序列化（通过 `orjson.OPT_SERIALIZE_NUMPY` 选项启用）、`UUID` 实例以及 Python 的 `dataclass` 对象 。例如，一个包含 `numpy` 数组的字典 `{'nums': numpy.array([1, 2, 3])}` ，在使用 `orjson.OPT_SERIALIZE_NUMPY` 选项后，可以正确序列化为 `b'{"nums":[[1,2,3]]}'` 。

尽管 `orjson` 提供了对多种类型的默认支持，但对于一些更特殊的类型（如自定义类，如果未定义为 `dataclass` ）或 `pandas` 的 `DataFrame` ，它可能仍然需要用户通过 `default` 参数提供一个自定义的序列化函数 。例如，序列化 `decimal.Decimal` 类型时，如果不提供 `default` 函数，会抛出 `TypeError: Type is not JSON serializable: decimal.Decimal` 。因此，虽然 `orjson` 在“开箱即用”方面表现优异，尤其是在性能和常用第三方类型支持上，但对于用户自定义类的通用序列化，仍可能需要一定的配置。

2.2. `jsonpickle`：复杂对象序列化与额外信息问题

`jsonpickle` 是一个旨在**序列化和反序列化复杂 Python 对象图为 JSON** 的库 。与标准 `json` 模块或 `orjson` 等库不同， `jsonpickle` 的核心目标是能够处理几乎任何 Python 对象，包括自定义类的实例、循环引用等复杂结构，而不仅仅是那些有直接 JSON 等价物的基本类型 。它通过在生成的 JSON 中包含额外的元数据（如对象的类名和模块路径，存储在 `"py/object"` 键下）来实现这一点，从而使得在反序列化时能够重建原始对象 。例如，一个 `Thing` 类的实例 `obj = Thing('Awesome')` 经过 `jsonpickle.encode(obj)` 序列化后，可能会得到类似 `{"py/object": "__main__.Thing", "name": "Awesome"}` 的 JSON 字符串（具体格式可能因版本和配置而异）。

这种机制使得 `jsonpickle` 在需要完整保存和恢复 Python 对象状态的场景下非常强大，例如缓存、持久化或进程间通信（如果通信双方都是 Python 程序且能访问相同的类定义）。然而，这种在 JSON 中添加额外类型信息的做法也带来了与用户需求中“开箱即用”且适用于“各种数据类型”的一些潜在冲突。特别是当序列化后的 JSON 需要被非 Python 系统（如微信小程序、其他语言的微服务）消费时，这些额外的 `py/object` 等字段可能会成为不兼容的噪音数据，甚至导致解析错误 。虽然 `jsonpickle` 提供了 `unpicklable=False` 选项，可以在序列化时禁用这些额外信息的添加，使得输出更接近标准 JSON ，但这会牺牲反序列化回原始 Python 对象的能力。例如，使用 `jsonpickle.encode(obj, unpicklable=False)` 后， `Thing` 实例可能被序列化为 `{"name": "Awesome"}` ，丢失了类型信息 。

此外，尽管 `jsonpickle` 可以处理复杂的对象图，但它对某些第三方库类型的原生支持可能不如专门优化的库。例如，对于 `numpy` 数组或 `pandas` DataFrame， `jsonpickle` 可能会将它们作为通用 Python 对象处理，可能不会采用最有效或最简洁的 JSON 表示 。同时， `jsonpickle` 文档中也明确警告其不安全性，指出恶意构造的 pickle 数据可能执行任意代码，因此只应用于信任的数据源 。这与标准 `json` 模块在处理不受信任数据时相对更安全的特性形成对比。

2.3. `simplejson` / `ujson` ：标准库的替代与性能提升

`simplejson` 和 `ujson` (UltraJSON) 是两个流行的 Python JSON 库，它们通常被视为对 Python 标准库 `json` 模块的替代品，主要目标是提供更好的性能和/或更广泛的特性支持。

`simplejson` 是 `json` 模块在 PyPI 上的一个向后移植，并且通常会比标准库版本更新，包含一些错误修复和性能改进 。它在 API 上与标准 `json` 模块高度兼容，因此可以很容易地作为替代品使用。虽然具体的性能提升可能不如 `orjson` 显著，但在某些场景下，它可能比标准库版本更快。它也支持通过 `cls` 参数传递自定义编码器，以及通过 `default` 参数处理无法直接序列化的对象，这与标准库的行为一致。 `simplejson` 还支持 `Decimal` 类型的序列化与反序列化（通过 `use_decimal=True` 参数）。

ujson 是一个用 C 编写的 JSON 编码器和解码器，以其**极快的速度**而闻名 。它旨在成为 json 模块的直接替代品，提供相同的 API（如 dumps()， loads()， dump()， load()）。根据一些基准测试， ujson 在序列化方面通常比标准 json 模块快得多 。例如，在一个测试中， ujson 的序列化时间为 1.413367 秒，而标准 json 为 1.616786 秒 。然而，需要注意的是， ujson 在某些边缘情况或数据类型的处理上可能与标准 json 模块存在细微差异，并且其反序列化性能在某些测试中可能不如 orjson 甚至标准库 。例如，同一个测试显示 ujson 的反序列化时间为 1.853332 秒，而标准 json 为 1.616203 秒， orjson 为 1.272813 秒 。

在数据类型支持方面， simplejson 和 ujson 通常遵循与标准 json 模块相似的规则，即主要支持基本 Python 类型。它们本身可能不直接支持 datetime 对象或 numpy 数组等更复杂的类型，除非用户通过 default 函数或自定义编码器提供序列化逻辑 。因此，在“开箱即用”的多类型支持方面，它们可能不直接满足用户对所有指定类型（如 datetime， numpy， pandas，自定义类）的默认序列化需求，仍然需要用户进行一定的扩展配置。它们的主要优势在于对标准 JSON 操作的性能提升，而不是数据类型的广泛覆盖。

2.4. 其他库（如 nujson， marshmallow，FastAPI 的集成方案）

除了上述主流库外，还有一些其他的 Python 库或方案可以用于 JSON 序列化，它们在特定场景下可能具有一定的优势。

nujson 是由彩云天气团队开发的一个 JSON 库，其诞生背景是为了解决他们在处理大量包含 numpy 数据类型的 JSON 序列化时遇到的性能问题 。他们发现，虽然 orjson 在标准数据类型上性能优越，但在处理包含 numpy 类型的数据时，如果通过 default 函数进行类型转换，其性能可能会下降，甚至不如 Pandas 维护的 ujson 版本 。因此，他们开发了 nujson，旨在 C 语言层面实现对 numpy 类型的判断与转换，从而在不影响系统性能的前提下，提供对 numpy 数组的良好支持 。这表明 nujson 的核心优势在于对 numpy 的原生支持和高性能。然而，关于 nujson 是否“开箱即用”地支持其他数据类型（如 pandas DataFrame、自定义类等），以及其输出的 JSON 格式是否与标准 JSON 完全兼容，目前的信息还比较有限。

marshmallow 是一个流行的 Python 库，用于将复杂的数据类型（如对象、ORM 实例）与 Python 原生数据类型之间进行转换，这个过程通常被称为序列化（serialization）和反序列化（deserialization），或者更一般地称为“对象序列化”和“数据验证”。 marshmallow 的核心思想是定义模式（Schema），这些模式描述了数据的结构、类型以及如何进行转换和验证。通过为自定义类或复杂数据结构定义 marshmallow.Schema，用户可以精确控制对象如何被序列化为字典或 JSON 字符串，以及如何从字典或 JSON 字符串反序列化回对象。 marshmallow 提供了丰富的字段类型（如 String， Integer， DateTime， List， Nested 等）和验证规则，使得它非常灵活和强大。虽然 marshmallow 本身不是一个

JSON 库，但它通常与标准 `json` 模块或其他 JSON 库（如 `orjson`）结合使用，先使用 `marshmallow` 将对象转换为 Python 基本类型的字典，然后再使用 JSON 库将字典序列化为 JSON 字符串。这种方案提供了极高的灵活性和对序列化过程的控制，但代价是需要用户显式定义模式，这增加了代码量，可能不完全符合“开箱即用”中“简单的配置或几行代码”的要求。

FastAPI 是一个现代、快速（高性能）的 Web 框架，用于构建 API。FastAPI 内置了对 Pydantic 模型的支持，Pydantic 是一个数据验证和设置管理库，它使用 Python 类型注解来定义数据的结构和类型。在 FastAPI 中，当你从路径操作函数返回一个 Pydantic 模型时，FastAPI 会自动将其序列化为 JSON 响应。Pydantic 模型本身也提供了将实例转换为字典或 JSON 字符串的方法（如 `.dict()` 和 `.json()`）。Pydantic 支持多种 Python 类型，包括标准库类型、`datetime` 对象、`UUID` 对象，以及通过自定义编码器（通过 `json_encoders` 配置）支持其他类型。例如，可以为 `numpy` 数组或自定义类定义特定的编码方式。FastAPI 的这种集成方案，对于在其框架内构建 API 的场景来说，可以认为是“开箱即用”的，因为它简化了从 Python 对象到 JSON 响应的转换过程。然而，如果脱离 FastAPI 环境，直接使用 Pydantic 进行序列化，虽然也很强大，但用户仍然需要定义模型并可能配置编码器，这与其他自定义方案类似。

3. “开箱即用”的多类型序列化方案探讨

3.1. `orjson` + 自定义 `default` 函数

`orjson` 库本身已经提供了对多种 Python 内置类型和常用第三方库类型（如 `datetime`，`numpy` 数组，通过特定选项）的良好支持，这在一定程度上满足了“开箱即用”的需求 。然而，为了实现对更广泛数据类型的全面覆盖，特别是用户自定义类以及 `orjson` 未原生支持的其他第三方库类型（如 `pandas` DataFrame），结合使用自定义的 `default` 函数是一个强大且灵活的策略。`orjson.dumps()` 方法接受一个 `default` 参数，该参数可以是一个可调用对象，当 `orjson` 遇到无法直接序列化的对象时，会调用此函数，并将该对象作为参数传入 。这个 `default` 函数应该返回一个可以被 `orjson` 序列化的值（通常是 Python 的基本类型或 `orjson` 原生支持的类型），或者抛出一个 `TypeError` 。

通过精心设计的 `default` 函数，可以实现对各种数据类型的定制化序列化逻辑。例如，对于 `datetime` 对象，如果 `orjson` 的默认 RFC 3339 格式不满足需求，可以在 `default` 函数中将其转换为其他字符串格式。对于 `numpy` 标量类型（如 `np.int32`，`np.float64`），虽然 `orjson` 在启用 `OPT_SERIALIZE_NUMPY` 时可以处理数组，但对单独的标量可能仍需在 `default` 函数中将其转换为 Python 原生的 `int` 或 `float` 。对于 `pandas` 的 `Series` 或 `DataFrame`，可以在 `default` 函数中调用其 `to_dict()` 或 `to_json()` 方法，或者将其转换为记录列表。对于用户自定义的类实例，`default` 函数可以访问对象的 `__dict__` 属性，或者调用对象自身定义的序列化方法（如

to_json()）。对于 set 和 tuple 类型， default 函数可以将它们分别转换为 list 。

处理 datetime ， numpy 标量与数组， pandas Series/DataFrame, 自定义类， set ， tuple

orjson 对 datetime 对象提供了良好的原生支持，能够将其序列化为符合 RFC 3339 标准的字符串格式，例如 "1970-01-01T00:00:00+00:00" 。对于 numpy 数组，可以通过在 orjson.dumps() 调用中设置 option=orjson.OPT_SERIALIZE_NUMPY 来启用原生序列化支持，这将把 numpy.ndarray 实例高效地转换为 JSON 数组 。然而，对于 numpy 标量（如 np.int32 ， np.float64 等）， orjson 可能仍然需要 default 函数将其转换为 Python 的原生标量类型（如 int ， float ），尽管在某些情况下 orjson 可能已经能够处理它们。

对于 pandas 的 Series 和 DataFrame ， orjson 本身不直接支持。在自定义的 default 函数中，可以调用这些对象的 to_json() 方法（返回一个 JSON 字符串，可能需要进一步处理）或 to_dict() 方法（返回一个字典，通常更易于 orjson 处理）。例如， DataFrame.to_dict(orient='records') 可以将 DataFrame 转换为一个由记录（字典）组成的列表，这种格式在 JSON 中非常常见。如果直接使用 to_json() ，需要注意其输出的可能不是标准的 JSON 数组或对象，而是特定格式的字符串，可能需要根据 orient 参数进行调整 。

处理用户自定义类时，一种常见的方法是在 default 函数中检查对象是否是特定类的实例，然后返回其 __dict__ 属性，该属性包含了对象的所有实例变量。另一种更可控的方式是让自定义类实现一个特定的方法（例如 to_json() 或 serialize()），该方法负责将对象转换为可序列化的字典或列表，然后在 default 函数中调用这个方法。对于 dataclasses ， orjson 从版本 3 开始原生支持其序列化，无需 default 函数，除非需要自定义序列化行为，此时可以使用 option=orjson.OPT_PASSTHROUGH_DATACLASS 并结合 default 函数 。

Python 内置的 set 和 tuple 类型， orjson 本身可能不完全支持或可能以非预期的方式序列化（例如， set 不是 JSON 原生支持的类型）。在 default 函数中，可以将 set 转换为 list 进行序列化，将 tuple 也转换为 list 或保持为 tuple （如果目标系统能处理）。关键在于确保转换后的类型是 JSON 兼容的。

示例代码与配置要点

以下是一个示例性的 default 函数，展示了如何处理多种数据类型，并结合 orjson 进行序列化：

Python 复制

```python
import orjson
import datetime
import numpy as np
import pandas as pd
from decimal import Decimal
from typing import Any

class CustomClass:
    def __init__(self, name: str, value: Any):
        self.name = name
        self.value = value

    # 可选：为自定义类实现一个序列化方法
    def to_json_serializable(self):
        return {"type": "CustomClass", "name": self.name, "value":
self.value}

def custom_default(obj: Any) -> Any:
    # 处理 datetime 对象 (orjson 本身支持，但这里可以自定义格式)
    if isinstance(obj, (datetime.date, datetime.time)):
        return obj.isoformat()
    # 处理 numpy 标量
    if isinstance(obj, np.generic):
        return obj.item()  # 将 numpy 标量转换为 Python 原生类型
    # 处理 numpy 数组 (orjson 通过 OPT_SERIALIZE_NUMPY 处理)
    # 这里作为 fallback，如果未启用 OPT_SERIALIZE_NUMPY
    if isinstance(obj, np.ndarray):
        return obj.tolist()
    # 处理 pandas Series
    if isinstance(obj, pd.Series):
        return obj.to_dict()
    # 处理 pandas DataFrame
    if isinstance(obj, pd.DataFrame):
        # 可以选择不同的 orient，如 'records', 'split', 'index',
'columns'
        return obj.to_dict(orient='records')
    # 处理 Decimal 类型
    if isinstance(obj, Decimal):
        return str(obj)  # 或者 float(obj)，但可能丢失精度
    # 处理 set 类型
    if isinstance(obj, set):
        return list(obj)
    # 处理 tuple 类型
```

```
    if isinstance(obj, tuple):
        return list(obj)  # 或者保持 tuple，取决于接收方
    # 处理自定义类（使用 __dict__ 或自定义方法）
    if isinstance(obj, CustomClass):
        # return obj.__dict__  # 方法一：使用 __dict__
        return obj.to_json_serializable()  # 方法二：使用自定义方法
    # 如果类型无法处理，抛出 TypeError
    raise TypeError(f"Object of type {type(obj)} is not JSON
serializable")

# 示例数据
data = {
    "timestamp": datetime.datetime.now(),
    "numpy_array": np.array([[1, 2], [3, 4]]),
    "numpy_int": np.int64(42),
    "pandas_series": pd.Series([1, 2, 3], name='example'),
    "pandas_df": pd.DataFrame({'A': [1, 2], 'B': ['x', 'y']}),
    "decimal_value": Decimal('3.14159'),
    "custom_obj": CustomClass("test", 123),
    "set_data": {1, 2, 3},
    "tuple_data": (4, 5, 6)
}

# 序列化，启用 OPT_SERIALIZE_NUMPY 以原生支持 numpy 数组
serialized_data = orjson.dumps(
    data,
    default=custom_default,
    option=orjson.OPT_SERIALIZE_NUMPY | orjson.OPT_NAIVE_UTC  # 示例选
项
)

print(serialized_data.decode('utf-8'))
```

配置要点：

1. **default 函数**：这是核心。它必须接受一个参数（要序列化的对象）并返回一个 orjson 可以序列化的对象，或者在无法处理时抛出 TypeError 。
2. **option 参数**： orjson.dumps() 的 option 参数允许通过位掩码组合多个选项来定制序列化行为 。
    - orjson.OPT_SERIALIZE_NUMPY ：启用对 numpy.ndarray 的原生序列化 。
    - orjson.OPT_NAIVE_UTC ：将 datetime.datetime 对象视为 UTC 时间，即使它们是原生的（无时区信息）。

- `orjson.OPT_PASSTHROUGH_DATACLASS`：允许 `default` 函数处理 `dataclass` 实例，而不是 `orjson` 的默认序列化 。
- `orjson.OPT_PASSTHROUGH_DATETIME`：允许 `default` 函数处理 `datetime` 对象，而不是 `orjson` 的默认序列化 。
- `orjson.OPT_SORT_KEYS`：对输出的 JSON 对象按键名排序 。
- `orjson.OPT_INDENT_2`：对 JSON 输出进行缩进（2个空格），提高可读性，但会增加数据大小并可能影响性能 。

3. **错误处理**：在 `default` 函数中，务必对所有无法处理的类型最终抛出 `TypeError`，这是 `orjson` 所期望的。
4. **性能考虑**：虽然 `orjson` 本身很快，但 `default` 函数是在 Python 层面执行的，复杂的 `default` 函数逻辑可能会成为性能瓶颈，特别是对于大型或嵌套很深的数据结构。应尽量保持 `default` 函数简洁高效。
5. **输出类型**：`orjson.dumps()` 返回的是 `bytes` 对象，而不是 `str` 。如果需要字符串，需要调用 `.decode('utf-8')` 。
6. **递归处理**：如果自定义对象内部包含其他自定义对象或复杂类型，`default` 函数需要能够递归地处理这些嵌套结构，或者确保它们本身也是可序列化的。

通过精心设计 `default` 函数并合理配置 `option` 参数，`orjson` 可以成为一个强大且灵活的 JSON 序列化工具，满足对性能和多种数据类型支持的需求。

3.2. `jsonpickle` 的 `unpicklable=False` 选项

`jsonpickle` 库的核心优势在于其能够序列化几乎任意的 Python 对象图，包括自定义类的实例、循环引用等复杂结构，这是通过在生成的 JSON 中包含对象的类型信息和状态来实现的 。然而，这种机制产生的 JSON 往往包含 `jsonpickle` 特有的元数据（如 `"py/object"` 键），这使得输出不再是标准的、简洁的 JSON，可能不适用于需要与其他系统（尤其是非 Python 系统）交互的场景 。为了解决这个问题，`jsonpickle` 提供了 `unpicklable=False` 选项。当在 `jsonpickle.encode(obj, unpicklable=False)` 中使用此选项时，库会尝试生成一个不包含这些额外类型信息的、更接近标准 JSON 的输出 。

使用 `unpicklable=False` 的主要优势在于，它允许 `jsonpickle` 利用其强大的对象图遍历能力，同时产生更干净的 JSON。对于许多内置类型和简单的自定义类（主要包含可序列化属性的类），这可以工作得很好。例如，一个包含基本数据类型的自定义类实例，在使用此选项后，其属性会被序列化为一个标准的 JSON 对象，而不带 `py/object` 标签。这使得序列化结果更容易被其他语言或工具解析。

然而， `unpicklable=False` 选项也存在一些局限性和需要注意的地方：

1. **反序列化能力丧失**：最显著的局限性是，当 `unpicklable=False` 时，序列化后的 JSON 数据将**无法**通过 `jsonpickle.decode()` 直接还原回原始的 Python 对象。类型信息被丢弃了，反序列化结果通常是 Python 字典或列表等基本结构 。如果目标是数据的持久化和完全恢复，这个选项就不适用。
2. **对复杂类型的处理可能不理想**：虽然 `jsonpickle` 会尝试序列化对象，但对于某些复杂的第三方库类型，如 `numpy` 数组或 `pandas` DataFrame，即使使用 `unpicklable=False` ，其默认的序列化方式可能仍然不是最优化或最符合特定需求的。例如， `numpy` 数组可能被序列化为一个嵌套列表，但可能丢失了其数据类型（如 `dtype` ）等信息。 `pandas` DataFrame 的默认序列化可能也不是最紧凑或最易读的形式 。
3. **自定义序列化逻辑的缺失**： `unpicklable=False` 主要影响的是是否添加类型元数据。它本身并不提供一种机制来为特定类型定义自定义的序列化格式（例如，将 `datetime` 对象格式化为特定的字符串表示）。如果需要对某些类型进行特殊的格式化，可能仍然需要结合其他方法，或者 `jsonpickle` 的 `make_refs=False` 等选项，但这会增加复杂性。
4. **并非所有对象都能完美处理**：对于一些非常复杂的对象，特别是那些其状态不完全由其 `__dict__` 属性决定的，或者包含大量计算属性的对象，仅使用 `unpicklable=False` 可能无法生成一个完整或有意义的 JSON 表示。

因此， `jsonpickle` 的 `unpicklable=False` 选项在需要将复杂 Python 对象快速转换为标准 JSON 格式，且不需要保留反序列化能力时，可以作为一种便捷的方案。它比完全依赖 `jsonpickle` 的默认行为（添加大量元数据）更适合于数据交换。但是，对于需要精确控制序列化格式、处理特定第三方库类型或保留反序列化能力的场景，它可能不是最佳选择，或者需要与其他技术结合使用。

3.3. 自定义 `JSONEncoder` 的实现

Python 标准库中的 `json` 模块允许通过继承 `json.JSONEncoder` 类并重写其 `default()` 方法来创建自定义的 JSON 编码器 。这种方法为实现“开箱即用”的多类型序列化提供了一种结构化的途径，允许开发者集中定义如何处理标准 `json` 模块无法直接序列化的各种数据类型。当 `json.dumps()` 或 `json.dump()` 函数遇到无法直接序列化的对象时，如果指定了自定义的 `JSONEncoder` （通过 `cls` 参数），则会调用该 `JSONEncoder` 的 `default()` 方法 。在 `default()` 方法内部，可以通过 `isinstance(obj, TargetType)` 来检查对象的类型，并返回一个可以被标准 JSON 编码器理解的 Python 对象（通常是字典、列表、字符串或数字）。如果 `default()` 方法也无法处理

该对象，则应调用父类的 `default()` 方法（即 `super().default(obj)`），这会引发一个 `TypeError`，表明该对象不可序列化 。

这种方法的优势在于其普适性和对标准库的集成。用户可以针对 `datetime` 对象、`Decimal` 对象、`numpy` 数组、`pandas` 的 `DataFrame` 或 `Series`，以及用户自定义的类实例等，编写统一的序列化逻辑。例如，可以将 `datetime` 对象转换为 ISO 8601 格式的字符串，将 `Decimal` 对象转换为字符串以保留精度，将 `numpy` 数组转换为其 `tolist()` 表示，将 `pandas` 的 `DataFrame` 转换为其 `to_dict(orient='records')` 的结果，或者将自定义类的实例转换为其 `__dict__` 属性（如果类的属性可以直接表示其状态）。

继承 `json.JSONEncoder` 并重写 `default` 方法

实现自定义 `JSONEncoder` 的核心步骤是创建一个继承自 `json.JSONEncoder` 的子类，并在这个子类中重写 `default(self, obj)` 方法。这个方法接收一个参数 `obj`，即需要被序列化的对象。在 `default` 方法内部，可以使用 `isinstance()` 函数来检查 `obj` 的类型，并根据类型执行相应的转换逻辑。如果 `obj` 的类型是 `JSONEncoder` 能够处理的，那么就返回转换后的 Python 基本数据类型。如果 `default` 方法无法处理该对象，它应该调用父类的 `default` 方法（即 `super().default(obj)`）或者直接抛出 `TypeError`，以遵循 `json` 模块的约定。

例如，一个简单的自定义编码器可能如下所示：

Python 复制

```python
import json
import numpy as np
import pandas as pd

class CustomJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        # 处理 numpy 数组
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        # 处理 numpy 标量
        if isinstance(obj, np.generic):
            return obj.item()
        # 处理 pandas DataFrame
        if isinstance(obj, pd.DataFrame):
            # 可以选择不同的 orient，如 'records', 'split', 'index',
'columns'
            return obj.to_dict(orient='records')
```

```
        # 处理 pandas Series
        if isinstance(obj, pd.Series):
            return obj.to_dict()
        # 对于其他类型，尝试调用其 to_json() 方法（如果存在）
        if hasattr(obj, 'to_json'):
            return obj.to_json()
        # 如果无法处理，调用父类的 default 方法，最终会抛出 TypeError
        return super().default(obj)

# 使用自定义编码器
data = {
    "numpy_array": np.array([1, 2, 3]),
    "pandas_df": pd.DataFrame({'col1': [1, 2], 'col2': ['a', 'b']})
}

json_string = json.dumps(data, cls=CustomJSONEncoder, indent=2)
print(json_string)
```

在这个例子中，`CustomJSONEncoder` 检查了 `numpy.ndarray` 、`numpy.generic` 、`pandas.DataFrame` 和 `pandas.Series` 类型，并将它们转换为 Python 列表或字典。它还提供了一个通用的回退机制，即如果对象拥有 `to_json()` 方法，则调用该方法。这种方式使得编码器具有一定的扩展性，可以处理其他定义了 `to_json()` 方法的自定义类型。

支持 `numpy` ，`pandas` 等类型的通用编码器示例

一个更健壮的通用编码器会考虑更多细节，例如处理 `datetime` 对象、`Decimal` 类型、`set` 、`tuple` ，以及更复杂的嵌套结构。以下是一个更全面的示例，展示了如何构建一个支持多种常用类型的通用 `JSONEncoder` ：

Python 复制

```
import json
from datetime import date, datetime, time
from decimal import Decimal
import numpy as np
import pandas as pd

class ExtendedJSONEncoder(json.JSONEncoder):
    def default(self, obj):
        # 处理 datetime 对象
        if isinstance(obj, (datetime, date, time)):
            return obj.isoformat()
        # 处理 Decimal 类型
```

```
        if isinstance(obj, Decimal):
            return str(obj)  # 或者 float(obj)，但可能丢失精度
        # 处理 numpy 数组
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        # 处理 numpy 标量
        if isinstance(obj, np.generic):
            return obj.item()
        # 处理 pandas DataFrame
        if isinstance(obj, pd.DataFrame):
            # 选择适合的 orient，例如 'records' 生成列表 of 字典
            return obj.to_dict(orient='records')
        # 处理 pandas Series
        if isinstance(obj, pd.Series):
            return obj.to_dict()
        # 处理 set 类型
        if isinstance(obj, set):
            return list(obj)
        # 处理 tuple 类型
        if isinstance(obj, tuple):
            return list(obj)
        # 处理自定义对象（如果定义了 __dict__ 或 to_dict() 方法）
        if hasattr(obj, '__dict__'):
            return obj.__dict__
        if hasattr(obj, 'to_dict'):
            return obj.to_dict()
        # 如果无法处理，调用父类的 default 方法
        return super().default(obj)

# 示例数据
complex_data = {
    "timestamp": datetime.now(),
    "date_today": date.today(),
    "nested": {
        "numpy_matrix": np.random.rand(2, 2),
        "pandas_series": pd.Series([10, 20, 30], name='values')
    },
    "unique_numbers": {1, 2, 2, 3},
    "coordinates": (4.5, 2.3),
    "decimal_price": Decimal('19.99')
}

# 使用自定义编码器进行序列化
json_output = json.dumps(complex_data, cls=ExtendedJSONEncoder,
```

```
indent=4)
print(json_output)
```

这个 ExtendedJSONEncoder 类覆盖了更广泛的类型。它将 datetime 、 date 和 time 对象转换为 ISO 8601 格式的字符串。 Decimal 对象被转换为字符串以避免精度丢失。 numpy 数组和标量被转换为 Python 列表和原生数值类型。 pandas 的 DataFrame 和 Series 被转换为字典或列表的字典。 set 和 tuple 被转换为列表。对于自定义类，它尝试序列化其 __dict__ 属性或调用 to_dict() 方法。这种编码器可以显著提高标准 json 模块处理复杂 Python 对象的能力，使其更接近“开箱即用”的目标，同时保持了与标准库的良好集成。然而，需要注意的是，标准 json 模块的性能通常不如 orjson 或 ujson 等第三方库。

4. 特定数据类型序列化的处理策略

4.1. Python 内置类型（ int ， str ， list ， dict ， tuple ， set ）

Python 标准库中的 json 模块对大部分内置基本数据类型提供了原生的序列化支持。这些类型及其在 JSON 中的对应关系如下表所示：

表格 复制

Python 类型	JSON 类型
dict	object
list , tuple	array
str	string
int , float	number
True	true
False	false
None	null

这意味着，当使用 json.dumps() 或 json.dump() 函数时，Python 中的字典会被转换为 JSON 对象，列表和元组会被转换为 JSON 数组（注意 tuple 反序列化后通常变为 list ），字符串、整数、浮点数、布尔值和 None 也都有其直接的 JSON 对应表示。然而，对于 set 类型，标准库 json 模块无法直接序列化，因为它没有直接的 JSON 等价物。常见的处理策略是在序列化之前将 set 转换为 list 。例如，在自定义 default 函数或 JSONEncoder 中，可以添加如下逻辑：

Python 复制

```python
if isinstance(obj, set):
    return list(obj)
```

或者，如果使用 jsonpickle ，它会自动将 set 序列化为 JSON 数组（列表）。此外，JSON 规范要求对象键必须是字符串。如果 Python 字典的键不是字符串，标准库 json 模块会引发 TypeError 。一些第三方库（如 orjson 配合 OPT_NON_STR_KEYS 选项）可以处理非字符串键，但通常会将其转换为字符串。

4.2. datetime 对象

在JSON序列化过程中，处理 datetime 对象是一个常见的需求，因为JSON标准本身并不直接支持日期时间类型。**最广泛接受和推荐的格式是ISO 8601字符串格式**，例如 "YYYY-MM-DDTHH:MM:SS.ssssss" （例如 "2023-10-26T14:30:00.000123" ）或其变种，如 "YYYY-MM-DDTHH:MM:SSZ" （UTC时间）或带有时区偏移量的 "YYYY-MM-DDTHH:MM:SS+HH:MM" 。这种格式不仅人类可读，而且易于机器解析，并且在不同系统和编程语言之间具有良好的互操作性。 orjson 库在这方面提供了良好的支持，它能够自动将Python的 datetime.datetime 和 datetime.date 实例序列化为RFC 3339格式（ISO 8601的一个配置文件）的字符串 。例如，一个 datetime.datetime(2022, 6, 12) 对象会被 orjson 序列化为 b'"2022-06-12T00:00:00"' 。如果需要更精细的控制，例如省略微秒部分，可以通过 orjson.OPT_OMIT_MICROSECONDS 选项来实现 。如果需要对 datetime 对象进行自定义格式化，可以使用 orjson.OPT_PASSTHROUGH_DATETIME 选项，并配合 default 函数来实现，例如将 datetime 对象格式化为特定的HTTP日期头格式（如 "Thu, 01 Jan 1970 00:00:00 GMT" ）。Python标准库的 json 模块本身不直接支持 datetime 序列化，需要用户自定义 default 函数或继承 JSONEncoder 来实现，通常也是将其转换为ISO 8601字符串 。

4.3. numpy 数组与标量

numpy 是Python中用于科学计算的核心库，其 ndarray （多维数组）和标量类型在数据处理中非常常见。然而，标准的JSON格式并不直接支持这些类型。因此，在将包含 numpy 数据的Python对象序列化为JSON时，需要进行类型转换。对于 numpy 标量（如 np.int32 ， np.float64 ， np.bool_ 等），常用的方法是使用其 item() 方法，该方法会返回一个对应的Python内置类型的值（如 int ， float ， bool ），这些类型可以直接被JSON序列化器处理 。例如， np.float32(3.14).item() 会返回Python的 float 类型 3.14 。对于 numpy 数组（ numpy.ndarray ），最直接的方法是使用 tolist() 方法，该方法会将整个数组转换为一个嵌套的Python列表结构，而列表是JSON支持的数据类型 。例

如，一个二维的 numpy 数组 np.array([[1, 2], [3, 4]]) 调用 tolist() 后会变成 [[1, 2], [3, 4]] 。

orjson 库对 numpy 数组的序列化提供了原生支持，并且性能优越。通过设置 option=orjson.OPT_SERIALIZE_NUMPY ， orjson 可以直接将 numpy.ndarray 实例高效地转换为JSON数组，而无需手动调用 tolist() 。根据文档， orjson 序列化 numpy.ndarray 实例的速度比其他库快4–12倍，并且内存占用更少 。如果 numpy 数组包含非数值类型（例如对象数组），或者其形状非常特殊， orjson 的行为可能会有所不同，或者仍然需要 default 函数的辅助。对于 numpy 的 datetime64 类型，它通常会被 orjson （在启用 OPT_SERIALIZE_NUMPY 时）序列化为ISO 8601格式的字符串数组。如果需要对 numpy 的 datetime64 进行特殊处理，可能需要在 default 函数中将其转换为Python的 datetime 对象或字符串。需要注意的是，如果数组非常大，将其转换为列表可能会消耗大量内存并导致性能下降，此时可能需要考虑其他序列化方案，如HDF5，或者分块处理数据 。

4.4. pandas Series 与 DataFrame

pandas 是Python中用于数据处理和分析的强大库，其核心数据结构 Series 和 DataFrame 在Web应用和API交互中也经常需要被序列化为JSON。与 numpy 类似，JSON标准并不直接支持这些 pandas 类型。 pandas 自身提供了 to_json() 方法，可以直接将 Series 或 DataFrame 对象序列化为JSON字符串 。这个方法非常灵活，通过 orient 参数可以控制输出的JSON格式。例如，对于 DataFrame ，常见的 orient 选项包括：

- 'split' ：输出一个包含 index 、 columns 和 data 三个键的字典，如 {"index": [...], "columns": [...], "data": [[...], ...]} 。
- 'records' ：输出一个字典列表，每个字典代表一行数据，如 [{"col1": val1, "col2": val2}, ...] 。这种格式非常适合在Web API中返回表格数据。
- 'index' ：输出一个字典，其键是索引标签，值是包含列数据的字典，如 {index1: {"col1": val1, "col2": val2}, ...} 。
- 'columns' ：输出一个字典，其键是列名，值是包含索引数据的字典，如 {"col1": {index1: val1, ...}, ...} 。
- 'values' ：仅输出值的二维数组。
- 'table' ：输出一个包含 schema 和 data 的字典，遵循Table Schema规范， data 部分的格式与 'records' 类似 。

选择合适的 orient 取决于数据的使用场景和消费端的需求。例如，如果前端需要直接渲染表格， 'records' 可能比较方便；如果需要保留索引和列名信息以便重建 DataFrame ， 'split' 或 'table' 可能更合适。

当使用通用的JSON库（如 orjson 或标准 json 模块）序列化包含 pandas 对象的Python数据结构时，如果这些库没有内置对 pandas 类型的支持，就需要在自定义的 default 函数中处理它们。一种常见的方法是在 default 函数中判断对象类型，如果是 pd.Series ，则调用其 to_dict() 方法将其转换为Python字典（通常是 {index: value} 的映射）；如果是 pd.DataFrame ，则根据需求调用其 to_dict(orient=...) 方法，例如 to_dict(orient='records') 将其转换为记录列表 。例如，在之前讨论的 custom_default 函数中，对 pd.Series 使用了 obj.to_dict() ，对 pd.DataFrame 使用了 obj.to_dict(orient='records') 。需要注意的是， pandas 的 to_json() 方法直接返回JSON字符串，而 to_dict() 方法返回Python数据结构，后者可以进一步被其他JSON库序列化。如果 DataFrame 中包含 datetime 对象或 numpy 类型， pandas 的 to_json() 方法通常能很好地处理它们，但自定义 default 函数时仍需确保这些嵌套类型也能被正确处理。

4.5. 用户自定义类实例

序列化用户自定义的类实例是JSON序列化中一个常见的挑战，因为JSON编码器默认并不知道如何将这些复杂的Python对象转换为JSON兼容的格式。有几种常用的策略来处理这种情况。最简单直接的方法是如果自定义类的实例有一个 __dict__ 属性（这通常是用户定义的类的默认行为，除非特殊定义了 __slots__ ），那么 __dict__ 属性会包含该实例的所有属性和对应的值，形成一个字典。这个字典可以直接被JSON序列化器处理。在自定义的 default 函数中，可以通过检查对象是否具有 hasattr(obj, '__dict__') 来判断，并返回 obj.__dict__ 。例如，一个类 Person 的实例 person = Person(name="Alice", age=30) ，其 person.__dict__ 可能是 {'name': 'Alice', 'age': 30} 。

另一种更可控和灵活的方式是在自定义类中实现一个特定的方法，例如 to_dict() 或 to_json() ，该方法负责将对象的内部状态转换为一个字典或可以直接序列化的JSON字符串。然后在通用的 default 函数中，检查对象是否拥有这个方法（例如 hasattr(obj, 'to_dict') ），如果存在则调用它（例如 return obj.to_dict() ）。这种方式允许类的设计者精确控制哪些属性应该被序列化以及它们的格式。例如，一个 User 类可能包含密码哈希等敏感信息，在 to_dict() 方法中可以明确排除这些字段。

对于使用 dataclasses 模块定义的类（Python 3.7+）， orjson 从版本3开始原生支持序列化 dataclasses.dataclass 实例，无需额外配置 。 dataclasses.asdict(obj) 函数也可以用来将 dataclass 实例转换为字典。如果需要对 dataclass 的序列化进行自定义（例如，排除某些字段或转换字段值），可以使用 orjson.OPT_PASSTHROUGH_DATACLASS 选项，

并配合 `default` 函数来实现 。例如，可以定义一个 `default` 函数，检查对象是否是 `dataclass` 实例（ `if dataclasses.is_dataclass(obj):` ），然后使用 `dataclasses.asdict(obj, dict_factory=...)` 或手动构建字典。

如果自定义类继承自一些复杂的第三方库的类（例如SQLAlchemy的ORM模型），情况可能会更复杂。SQLAlchemy模型实例通常不会直接将其所有列和关系暴露在 `__dict__` 中，或者 `__dict__` 可能包含SQLAlchemy内部状态。在这种情况下，通常需要更特定的序列化逻辑。例如，可以定义一个方法遍历模型的 `__table__.columns` 来获取列名和值，或者使用像 `marshmallow-sqlalchemy` 这样的库来定义模式并进行序列化。Django的ORM模型也有类似的考虑，可以使用Django内置的序列化模块，或者手动构建字典 。关键在于将复杂的对象图展平为JSON能够表示的简单数据结构（如字典、列表、字符串、数字等）。

4.6. 其他特定类型（如函数、生成器、文件对象等）

对于函数、生成器、文件对象等特定类型的序列化，通常没有一种通用的、标准化的JSON表示方法，因为它们的语义和行为与JSON的简单数据结构（对象、数组、值）有较大差异。因此，处理这些类型时需要根据具体应用场景进行考量。

函数 (Function)：序列化函数本身（即其代码）通常不是JSON序列化的目标。如果目的是传递可执行逻辑，可以考虑序列化函数的名称（字符串）或某种标识符，然后在接收端根据这个标识符查找或重建函数。如果确实需要序列化函数对象， `jsonpickle` 可能会尝试序列化其字节码或引用，但这通常不是跨平台或安全的做法，并且反序列化后可能无法正确执行。

生成器 (Generator)：生成器是状态性的迭代器。直接序列化生成器对象本身通常没有意义，因为它代表的是一个计算过程而非静态数据。更常见的做法是序列化生成器产生的数据序列。可以将生成器产生的所有值收集到一个列表或元组中，然后序列化这个容器。例如： `json.dumps(list(my_generator()))` 。如果生成器产生无限序列或数据量非常大，直接序列化其所有输出可能不现实，此时可能需要重新设计数据流或分块处理。

文件对象 (File Object)：文件对象代表一个打开的文件流，它包含操作系统级别的资源（如文件描述符）和缓冲区状态，这些都无法直接映射到JSON。序列化文件对象本身通常没有意义。如果需要序列化文件内容，应该读取文件内容（例如，作为字节串或字符串），然后序列化这些内容。如果需要序列化文件的元数据（如文件名、路径、大小、修改时间等），可以提取这些信息构建一个字典，然后序列化这个字典。例如，一个文件对象 `f = open('example.txt', 'r')` ，可以序列化其内容 `f.read()` （如果是文本文件）或 `f.name` （文件名）。

对于这些特殊类型，通用的策略是：

1. **转换为基本类型**：尽可能将对象转换为JSON支持的基本类型（字符串、数字、布尔值、列表、字典）。例如，函数可以转换为其名称字符串，生成器的输出可以转换为列表。
2. **序列化为元数据/占位符**：如果不能或不希望序列化完整状态，可以序列化一个包含对象类型和关键元数据的字典。例如，对于文件对象，可以序列化 `{"type": "file", "name": "example.txt", "mode": "r"}`。
3. **忽略或引发错误**：在某些情况下，如果特定类型不应该被序列化，或者没有合理的JSON表示，可以在自定义编码器或 `default` 函数中明确忽略它们（例如，返回 `None` 或一个特定的标记），或者抛出一个 `TypeError` 表示该类型不可序列化。
4. **使用特定库**：如 `jsonpickle` 可能会尝试序列化这些对象，但其输出可能包含Python特定的信息，并且反序列化可能有限制或安全隐患。

总之，处理这些特定类型需要仔细考虑序列化的目的和接收方的期望，通常需要定制化的逻辑。

5. 与微信小程序等外部系统的 JSON 兼容性

5.1. 避免非标准 JSON 的额外类型信息

在与微信小程序等外部系统进行数据交互时，确保 JSON 数据的兼容性是至关重要的。这些系统通常期望接收符合标准 JSON 规范的数据，而不包含任何特定于序列化库的额外类型信息。许多 Python JSON 序列化库，如 `jsonpickle`，在默认情况下为了能够在 Python 环境中完整地反序列化对象，会在生成的 JSON 中添加额外的字段。例如，`jsonpickle` 默认会添加 `"py/object"` 这样的键来标识原始 Python 对象的类型 。虽然这对于 Python 内部的序列化和反序列化非常有用，但它会导致生成的 JSON 字符串不再是严格意义上的标准 JSON，从而可能被外部系统视为无效或无法解析。例如，一个包含自定义类 `Person` 实例的对象，在使用 `jsonpickle` 默认序列化后可能看起来像 `{"py/object": "__main__.Person", "name": "Alice", "age": 30}`。这种包含 `"py/object"` 字段的 JSON 对于不期望此类信息的微信小程序来说是不兼容的。

为了解决这个问题，关键在于选择能够生成纯净 JSON 的序列化方法或配置。对于 `jsonpickle`，可以通过设置 `unpicklable=False` 参数来避免添加这些额外的类型信息 。当使用 `jsonpickle.encode(obj, unpicklable=False)` 时，生成的 JSON 将只包含对象的数据属性，而不会包含 `jsonpickle` 特有的元数据。例如，上述 `Person` 对象在 `unpicklable=False` 的情况下会被序列化为 `{"name": "Alice", "age": 30}`，这是一个标准的 JSON 对象，可以被微信小程序等外部系统正确解析。同样，如果使用 `orjson` 这样的库，它本身设计用于生成高度优化的标准 JSON，但处理自定义类型时通常需要用户提供一个 `default` 函数来指定如何将这些类型转换为 JSON 兼容的类型。在这个 `default` 函数

中，用户也需要确保返回的是标准的 JSON 类型（如字典、列表、字符串、数字等），而不是包含额外类型信息的复杂对象。

因此，为了确保与微信小程序等外部系统的 JSON 兼容性，开发者需要仔细选择序列化库并正确配置其选项，以避免在 JSON 输出中包含非标准的额外类型信息。这意味着可能需要放弃一些库提供的“智能”反序列化功能，以换取数据的通用性和互操作性。在评估一个序列化方案时，不仅要考虑其序列化 Python 对象的能力，还要仔细检查其生成的 JSON 输出是否符合目标系统的期望。对于 `jsonpickle`，这意味着几乎总是需要使用 `unpicklable=False`。对于其他库，则可能需要自定义序列化逻辑或确保其默认行为不引入非标准元素。最终目标是生成一个简洁、标准、不包含任何可能引起解析歧义或错误的额外信息的 JSON 字符串。

5.2. 生成简洁、标准的 JSON 输出

生成简洁且符合标准的 JSON 输出是确保与微信小程序等外部系统顺利通信的关键。标准 JSON (RFC 8259) 定义了一组有限的数据类型：对象（字典）、数组（列表）、字符串、数字、布尔值（`true` / `false`）和 `null`。任何超出这些类型的表示都可能导致解析错误或数据 misinterpretation。许多 Python 对象（如 `datetime` 对象、`numpy` 数组、自定义类实例）没有直接的 JSON 等效项，因此序列化库需要提供一种机制将它们转换为标准 JSON 类型。理想情况下，这种转换应该产生尽可能简洁的 JSON，避免不必要的嵌套或冗余信息，以减少数据传输量并提高解析效率。

使用 `jsonpickle` 时，通过设置 `unpicklable=False` 参数，可以有效地去除其默认添加的 `"py/object"` 等类型信息，从而生成更接近标准 JSON 的输出。例如，一个包含 `datetime` 对象的 Python 字典 `{"event": "meeting", "time": datetime.datetime(2023, 10, 26, 15, 30)}`，在使用 `jsonpickle` 并配置了相应的处理器（或依赖其内置的 `DatetimeHandler`）且 `unpicklable=False` 时，可能会被序列化为 `{"event": "meeting", "time": "2023-10-26T15:30:00"}`。这里，`datetime` 对象被转换为了 ISO 8601 格式的字符串，这是一种被广泛接受的标准日期时间表示形式，也是 JSON 兼容的。同样，对于自定义类，`jsonpickle` 在 `unpicklable=False` 模式下，通常会将其属性序列化为一个 JSON 对象，例如 `MyClass(name="Test", value=42)` 可能变成 `{"name": "Test", "value": 42}`，这完全符合 JSON 标准。

然而，仅仅设置 `unpicklable=False` 可能不足以应对所有情况，特别是对于 `numpy` 和 `pandas` 等库的复杂数据结构。`jsonpickle` 的文档提到，为了确保 `numpy` 数组的输出不包含其 `dtype` 信息，可能需要注册一个特定的处理器（`UnpicklableNumpyGenericHandler`）。对于 `pandas` DataFrame，虽然 `jsonpickle` 提供了支持，但用户仍需确保其序列化结果（例如通过 DataFrame 的 `to_json()` 方法或 `jsonpickle` 的内部处理）是简洁且标准的。例如，一个 `pandas` DataFrame 可以被序列

化为一个包含记录列表的 JSON 数组，或者一个以列为键、以值列表为值的 JSON 对象，具体取决于所选的 `orient` 参数（如果通过 `to_json()` 处理）。目标应该是选择一个能够被微信小程序轻松解析和使用的结构。例如， `df.to_json(orient="records")` 会生成一个形如 `[{"col1": "val1", "col2": "val2"}, {...}]` 的 JSON 数组，其中每个元素是一个代表行的 JSON 对象，这种格式通常易于处理。因此，开发者需要仔细考虑每种数据类型的序列化策略，以确保最终的 JSON 输出既简洁又完全符合标准，从而最大限度地提高与外部系统的兼容性。

6. 性能考量与最佳实践

6.1. `orjson` 的性能优势

`orjson` 库因其 Rust 实现而展现出**显著的性能优势**，通常远超 Python 标准库的 `json` 模块以及其他主流第三方 JSON 库如 `ujson` 和 `rapidjson` 。根据官方文档和多项基准测试， `orjson` 在序列化和反序列化速度上可以达到其他库的数倍甚至数十倍，尤其是在处理大规模或复杂数据结构时 。例如，序列化 `dataclass` 实例时， `orjson` 比其他库快40–50倍；序列化 `numpy` 数组时快4–12倍，内存使用仅为其他库的0.3倍；美化输出比标准库快10–20倍；序列化浮点数比其他库快10倍，反序列化快2倍 。这种性能提升对于 I/O 密集型应用、高并发 API 服务以及大规模数据处理场景至关重要，能够有效降低延迟并提高吞吐量。 `orjson` 直接序列化为 `bytes` 对象而非 `str` 对象，这避免了不必要的编码转换，进一步提升了效率 。此外， `orjson` 严格遵守 JSON 标准（RFC 8259），并且在正确性方面也表现出色，能够正确处理各种边缘情况和测试用例 。

6.2. 自定义 `default` 函数对性能的影响

虽然 `orjson` 等高性能库本身速度极快，但当需要通过自定义 `default` 函数来处理其原生不支持的数据类型时，该函数的执行效率会成为影响整体序列化性能的关键因素。 **`default` 函数是在 Python 层面执行的，而 `orjson` 的核心逻辑是用 Rust 编写的**。如果 `default` 函数逻辑复杂、包含大量 Python 代码或频繁的类型检查，它可能会成为性能瓶颈，尤其是在序列化大量对象或嵌套层级较深的数据结构时 。彩云天气团队的测试表明，当 `default` 函数需要处理的类型数量达到 3 个时， `orjson` 的性能可能不如优化过的 `ujson` 。因此，在实现自定义 `default` 函数时，应遵循以下最佳实践以最小化性能开销：

1. **保持简洁高效**：尽量使 `default` 函数中的逻辑简单直接，避免不必要的计算或耗时操作。
2. **优先使用 `isinstance`** ：使用 `isinstance(obj, TargetType)` 进行类型检查通常比 `type(obj) is TargetType` 更灵活且高效，尤其是在处理继承关系时。
3. **避免重复计算**：如果某些计算在 `default` 函数中可能被多次执行，考虑是否可以预先计算或缓存结果。

4. **利用库原生方法**：对于 `numpy` 或 `pandas` 等库的类型，尽量使用它们自身提供的高效转换方法（如 `tolist()`，`item()`，`to_dict()`），而不是手动迭代或构建数据结构。
5. **减少 Python 对象创建**：频繁创建和销毁大量小型 Python 对象（如临时列表或字典）会产生垃圾回收开销。如果可能，尝试重用对象或使用更高效的数据结构。
6. **性能分析**：对于性能关键的代码路径，使用 Python 的性能分析工具（如 `cProfile`）来识别 `default` 函数中的热点，并进行针对性优化。

通过精心设计和优化 `default` 函数，可以在享受 `orjson` 等库带来的高性能的同时，灵活处理多种数据类型。

6.3. 数据预处理与类型检查的重要性

在 JSON 序列化过程中，**数据预处理和严格的类型检查是确保数据质量和避免运行时错误的关键环节**。在将数据传递给序列化器之前，进行适当的数据清洗、转换和验证，可以显著提高序列化过程的稳定性和可靠性。例如，确保所有字典的键都是字符串类型（如果目标 JSON 库严格要求），将非 JSON 兼容的类型预先转换为兼容类型，或者处理可能存在的 `None` 值。对于用户自定义类，如果其内部状态复杂或包含不可序列化的属性，预处理步骤可以将其转换为一个更简单的、只包含可序列化数据的字典或数据结构。

类型检查同样重要。在自定义的 `default` 函数或 `JSONEncoder` 中，使用 `isinstance()` 来明确处理预期的数据类型，并为无法处理的类型提供清晰的错误提示或备选方案。这有助于及早发现数据不匹配的问题，而不是在序列化过程中因意外的 `TypeError` 而失败。例如，如果期望一个属性是 `datetime` 对象，但在实际数据中它可能是一个字符串或 `None`，类型检查可以帮助识别并处理这些不一致。对于从外部来源（如数据库、API 响应）获取的数据，进行严格的类型检查和转换尤为重要，因为这些数据可能不完全符合预期格式。通过在前端或数据准备阶段进行充分的预处理和类型校验，可以减少序列化逻辑的复杂性，并确保生成的 JSON 数据是干净、一致且符合预期的。

7. 结论与推荐

7.1. 当前“开箱即用”程度的评估

综合来看，**目前 Python 生态中尚无一个 JSON 序列化库能够完美满足所有“开箱即用”和多类型支持的需求**。标准库 `json` 模块在类型支持上较为基础，需要大量自定义工作。第三方库各有侧重：

- `orjson` 在性能和部分内置类型（如 `datetime`、`numpy` 数组）的原生支持上表现突出，但对于自定义类、`pandas` 等仍需 `default` 函数，这在一定程度上偏离了“无需配置”的理想状态。

- `jsonpickle` 确实能“开箱即用”地处理几乎所有 Python 对象，但其默认输出包含非标准类型信息。即使使用 `unpicklable=False`，对于 `numpy` 和 `pandas` 等特定类型的序列化结果可能仍不够理想或需要额外配置。
- `simplejson` / `ujson` 主要提升了标准库的性能，但在广泛的类型支持上改进有限，仍需自定义。
- `marshmallow` / Pydantic 等数据验证和序列化框架提供了强大的控制力和灵活性，但需要显式定义模式/模型，这与“简单的配置或几行代码”的期望有一定距离。

因此，**当前的“开箱即用”程度是相对的**。对于常见的内置类型和少数第三方库类型，`orjson` 等库提供了较好的默认支持。但对于更广泛的“各种数据类型”，特别是用户自定义类和复杂的第三方库对象，通常仍需要用户进行一定程度的编码或配置。

7.2. 针对不同需求的推荐方案

根据不同的应用场景和需求，推荐以下 JSON 序列化方案：

表格 复制

需求场景	推荐方案
追求极致性能，且主要处理标准类型或 `orjson` 原生支持类型	`orjson`
需要处理多种自定义类型和复杂对象图，且对输出纯净度要求不高（如 Python 内部通信）	`jsonpickle`
需要生成标准 JSON，且愿意编写一些自定义逻辑来处理 `orjson` 不原生支持的类型	`orjson` + 自定义
需要严格的数据验证和复杂的序列化/反序列化规则，且代码量不是首要考虑	`marshmallow` 或
对性能有较高要求，且数据中包含大量 `numpy` 类型	`nujson`（如果其通用 `default` 函数
仅需处理基本 Python 类型，且对性能要求不高	Python 标准库 `json`

Table 2: 针对不同需求的 JSON 序列化方案推荐

- **追求极致性能与灵活配置：`orjson` + 自定义 `default`**
  对于需要高性能且能够灵活处理多种数据类型的场景，**`orjson` 配合自定义 `default` 函数是目前较为理想的组合。** `orjson` 提供了卓越的序列化速度和内存效率，以及对

`datetime` 、 `numpy` 数组等类型的原生支持 。通过精心设计的 `default` 函数，可以扩展其对 `pandas` DataFrame/Series、用户自定义类、 `set` 、 `tuple` 等类型的支持，确保生成的 JSON 是标准且符合外部系统要求的。虽然这需要用户编写和维护 `default` 函数，但它提供了对序列化过程的精细控制，并能充分利用 `orjson` 的性能优势。配置 `orjson` 的 `option` 参数（如 `OPT_SERIALIZE_NUMPY` ， `OPT_NAIVE_UTC` ）可以进一步优化其行为 。

- **快速序列化复杂对象（不严格依赖标准JSON，或可接受 `unpicklable=False` 的限制）： `jsonpickle` （注意 `unpicklable=False` ）**
  如果应用场景主要是在 Python 环境内部进行对象的持久化或通信，并且需要方便地序列化和反序列化几乎任意的 Python 对象图（包括自定义类、函数等）， **`jsonpickle` 是一个强大的选择** 。它能够自动处理这些复杂对象，而无需为每个类编写特定的序列化逻辑。然而，当需要与外部系统（如微信小程序）交换数据时，其默认生成的包含额外类型信息的 JSON 可能不兼容。此时，可以使用 `unpicklable=False` 选项来生成更接近标准 JSON 的输出 。但需注意，即使在此模式下，对于 `numpy` 和 `pandas` 等特定类型的处理可能仍需谨慎评估，其输出可能不是最简洁或最理想的格式，且反序列化能力会受限 。

7.3. 对未来更完善解决方案的展望

展望未来，理想的 Python JSON 序列化库应朝着**更高程度的“开箱即用”和更广泛的类型支持**方向发展，同时保持高性能和安全性。这可能包括：

1. **更智能的默认序列化规则**：库能够自动推断更多常见 Python 类型（包括流行的第三方库类型和用户自定义类）的合理 JSON 表示，而无需或仅需极少的配置。例如，自动将自定义类的 `__dict__` 或 `__slots__` 序列化为 JSON 对象，或者为 `pandas` 和 `numpy` 类型提供更优化的默认输出选项。
2. **模块化和可扩展的类型处理**：提供一个清晰的插件系统或注册机制，允许用户轻松地为特定类型注册自定义的序列化和反序列化逻辑，这些逻辑可以像库内置的一样被调用。这可以平衡默认行为的丰富性和用户自定义的灵活性。
3. **性能与安全并重**：在追求极致性能的同时，确保库能够安全地处理不受信任的输入，避免类似 `pickle` 模块的安全隐患。
4. **更好的标准兼容性与配置选项**：提供丰富的选项来控制 JSON 输出的格式（如缩进、键排序、日期时间格式、NaN/Infinity 处理等），同时确保默认行为符合最新的 JSON 标准。
5. **更友好的错误提示和调试信息**：当序列化或反序列化失败时，提供清晰、具体的错误信息，帮助开发者快速定位问题。

社区驱动的库（如 `orjson` ）已经在这些方面取得了显著进展。未来的发展可能会看到更多库借鉴这些优点，并进一步探索如何更智能、更便捷地处理 Python 丰富多样的数据类型到 JSON 的转换。同时，Python 语言本身的发展（如类型注解的普及）也可能为序列化库提供新的思路和优化方向。