# JSON Path to Path: Batch processing method for JSON

JSON (JavaScript Object Notation) is a lightweight data exchange format that has become one of the standard data formats in modern software development due to its concise and readable characteristics. Whether it is data interaction between front-end and back-end, or communication between microservices, JSON plays an important role.

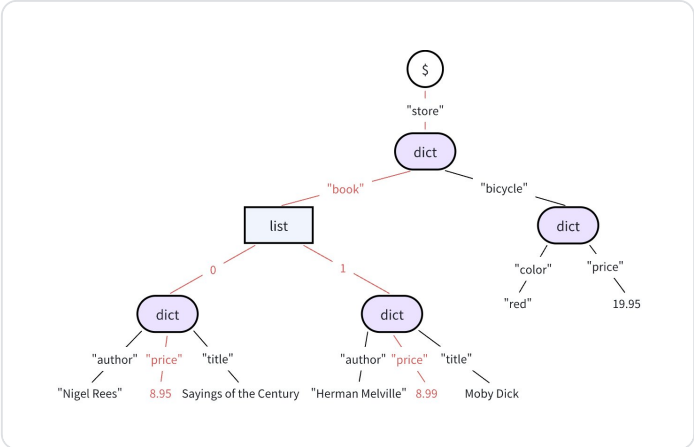
However, with the complexity of data structures, JSON processing has become increasingly challenging. Imagine facing a deeply nested JSON document and wanting to batch extract or modify certain data. Manual operations are not only time-consuming but also prone to errors.

Don't worry, here we will take you to explore an efficient JSON batch processing method, making it easy for you to handle complex JSON data operations.

## JSON Tree and JSON Path

```
{
    "store": {
        "book": [
            {"author": "Nigel Rees", "price": 8.95, "title": "Sayings of the Century"},
            {"author": "Herman Melville", "price": 8.99, "title": "Moby Dick"}
        ],
        "bicycle": [
            {
                "color": "red",
                "price": 19.95
            }
        ]
    }
}
```

- JSON Tree




- JSON Path

```
$.store.book[*].price=[8.95,8.99]
```

### What is JSON Tree?

Before we delve into the discussion, we need to understand the tree structure of JSON. JSON data can be seen as a tree, where each node represents a `dict`, `list`, `str`, `number`, `bool` or `null`. The edges of the tree are composed of dictionary keys or list indexes, and nodes are concrete values.

JSON Path can be seen as navigating through the JSON tree to locate the target node. It uses path expressions to describe the path from the root node to the target node.

### What is JSON Path?

JSON Path is a query language used to locate and extract data from JSON documents. Its inspiration comes from XPath, which is a language used to navigate and query data in XML documents. With concise syntax, JSON Path allows developers to quickly extract the required data from complex JSON structures without manually traversing the entire document.

### Basic syntax of JSON Path

The syntax of JSON Path is similar to the path expression of a file system. Here are some common examples of JSON Path expressions:

- `$.store.book[0].title` : Extract the `title` attribute of the first element of the `book` array under the `store` object in the JSON document.
- `$..Price` : Extracts the value of all `price` properties in the JSON document.
- `$store.book[*].author` : Extract the `author` property of all elements in the `book` array under the `store` object.

With these expressions, developers can easily extract the required data from complex JSON structures.

## Nodes and slots

Slot is a core concept of the batch processing method in this article. Slot can be understood as the position in the JSON tree where edges (keys or indexes) and nodes (values) are placed. Each slot can accommodate a node or an edge, and JSON operations essentially occupy and release slots.

### Node into slot modeling JSON operation

Under the JSON tree model, all JSON operations can be seen as the process of moving nodes (or edges) to slots. Whether adding, modifying, or deleting data, it is essentially adjusting the position of nodes or edges in the slots. For example:

- **Add data** : Place a node (and edge) into an empty slot.
- **Modify data** : move a node (and edge) into an occupied slot, overwriting the original content.

- **Delete data** : Move nodes and edges from the slot to the slot at infinity (graveyard slot), making the slot empty.

### Slot preemption mode

When a slot is already occupied, the new node or edge will overwrite the original content. This overwriting behavior is a common operation in JSON batch processing, for example:

- If a value is placed in an occupied slot, the original value will be replaced.
- If an object or array is placed in an occupied slot, the original data structure will be completely replaced.

Override rules allow JSON batch processing to update data flexibly, but also need to be careful to avoid accidental data loss.

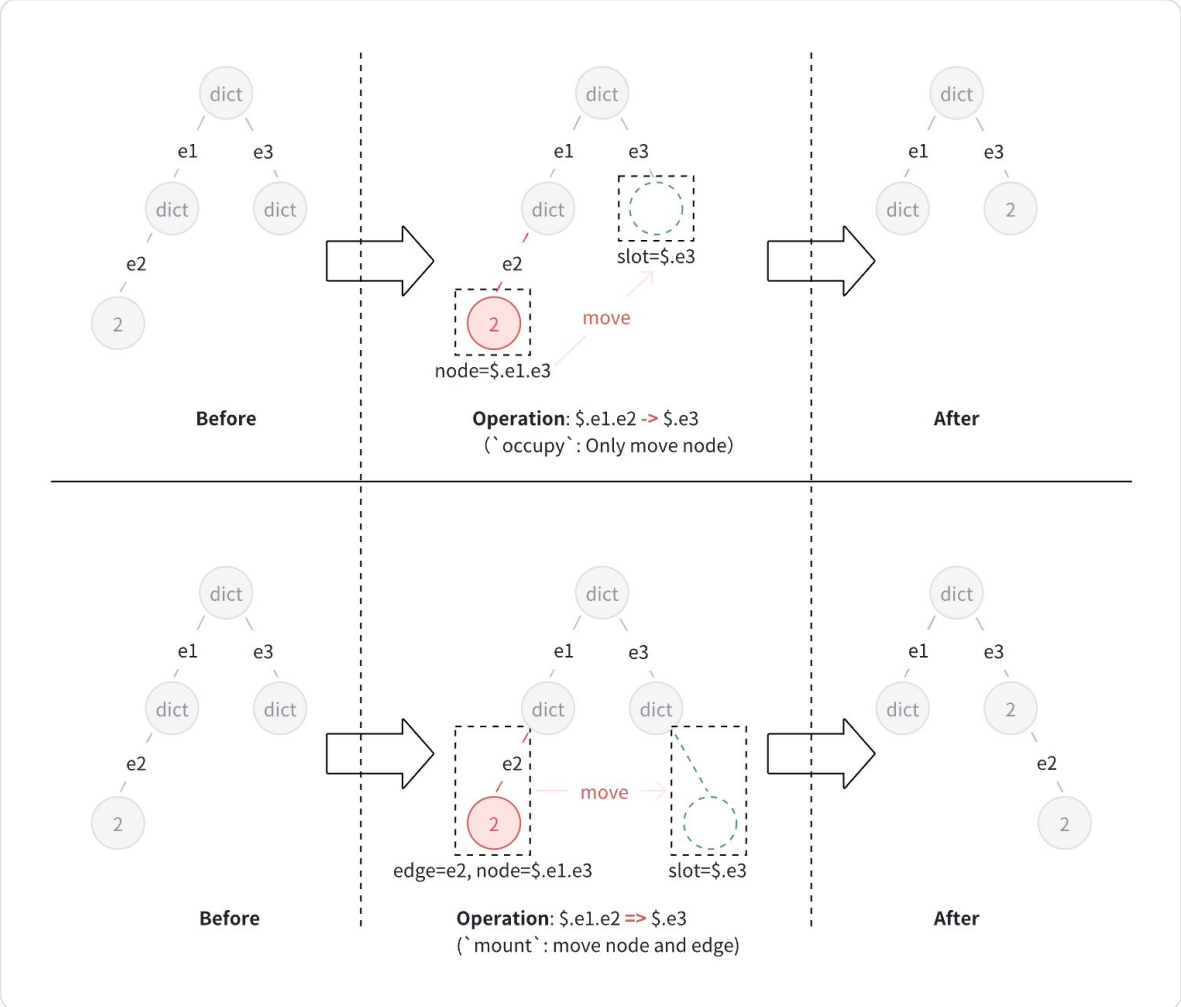
### Replacement vs Mount

In JSON batch processing, node movement and edge + node movement are two different operations.

- **Node movement (occupy, represented by the symbol `->` )** : Only move the node (value) without changing its edge (key or index). For example, move a value from one slot to another, but keep its key or index unchanged.
- **Edge + Node Move (Mount, represented by the symbol `=>` )** : Move both the edge and the node. For example, move an Attribute - Value Pair from one object to another, or move an array element from one index to another.

> ⚠ `List` type out-of-bounds replacement/mounting, default to append mode, that is, if `idx > = len (list)` is appended to the `list` .
>
> **Reason** : Out of bounds will cause the value of some positions in the middle of the `list` to be empty, which is illegal and meaningless to some extent.

The difference between these two operations lies in whether to change the positional relationship of nodes in the JSON tree. Understanding this helps us choose the appropriate operation method in batch processing.




## From one-on-one to batch processing

The core of JSON batch processing is to decompose complex operations into one-to-one correspondence between nodes and slots. By mapping each operation to a simple task of "putting a node into a slot", we can transform the batch processing problem into the following steps:

1. **Get node list** : Match obtained by JSON Path, or directly generate a batch of new nodes;
2. **Get slot list** : Match the slot position by JSON Path;
3. **Implicit one-to-one correspondence** : the node list and the slot list must be one-to-one correspondence or can be achieved through some transformations;
4. **Perform the operation** : move the nodes one by one to the slot.

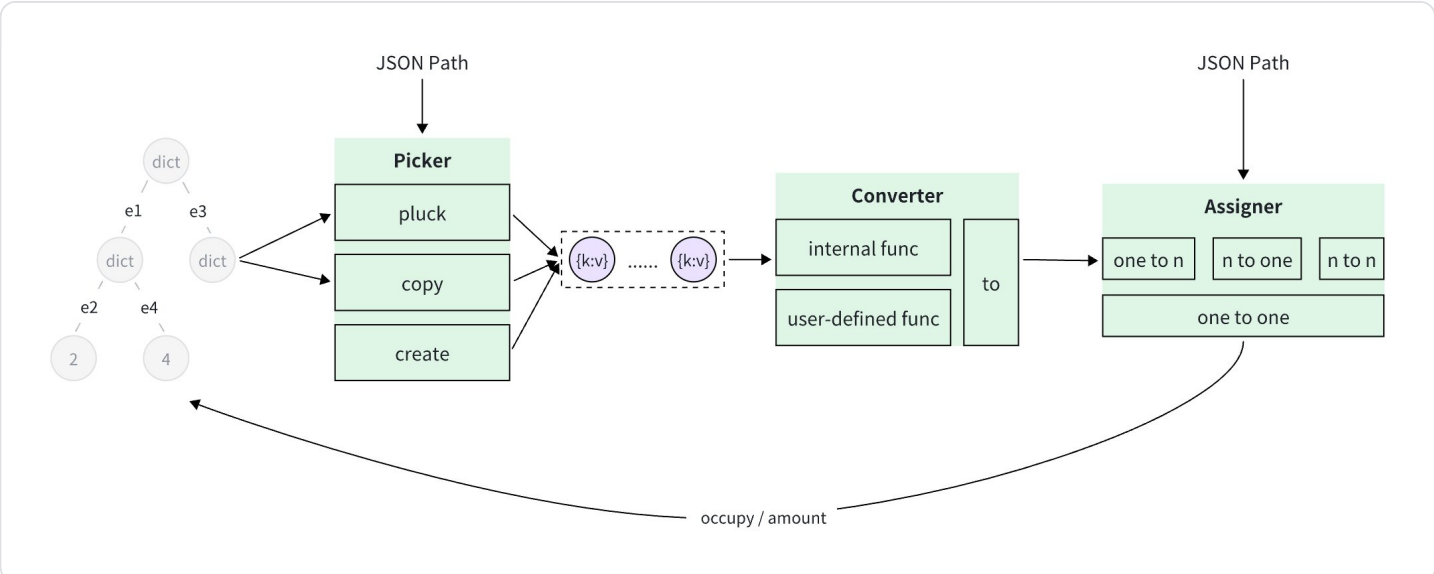
| Node: Slot | Conversion operation | Occupancy type | Example |
|---|---|---|---|
| `1 : 1` | Without conversion | Occupy ( `->` ) | Move a specified node to replace the target location value.<br>• `$.src->$.trg` |

| | | | |
|---|---|---|---|
| | | | • Equivalent to `json[trg]=json[src], del(json,trg))` |
| | | Mount ( `=>` ) | Move a specified node and mount it to the target location.<br>• `$.src=>$.trg`<br>• Equivalent to `json[trg][src]=json[src], del(json,src)` |
| `N : N` | Correspond one-to-one according to the list index | Occupy ( `->` ) | Move N designated nodes one by one to replace the values of N target locations.<br>• `$.src[0:n]->$.trg[0:n]`<br>• Equivalent to `json[trg][0]=json[src][0],..., json[trg][n-1]=json[src][n-1], del(json[src], 0, n)` ) |
| | | Mount ( `=>` ) | Move N designated nodes one-to-one and mount them to N target locations.<br>• `$.src[0:2].a=>$.trg[0:2]`<br>• Equivalent to `json[trg][0][a]=json[src][0][a], json[trg][1][a]=json[src][1][a], del(json,src,0,2)` |
| `1 : N` | Copy the node to `N` copies of one-to-one correspondence | Occupy ( `->` ) | Copy a node and replace N target location values in a diffuse manner.<br>• `$.src->$.trg[0:n]`<br>• Equivalent to `json[trg][0]=json[src],..., json[trg][n-1]=json[src], del(json, src)` |
| | | Mount ( `=>` ) | Copy a node and mount it diffusely to N target locations.<br>• `$.src=>$.trg[0:n]`<br>• Equivalent to `json[trg][0][src]=json[src],..., json[trg][n-1][src]=json[src], del(json, src)` |
| `N : 1` | Only support mounting operations, corresponding operations for target location type: | Mount ( `=>` ) | Mount N nodes to a target location<br>• `$.src[0:n]=>$.trg`<br>• Equivalent to `json[trg].append(json[trg][0]),..., json[trg].append(json[trg][n-1])` |

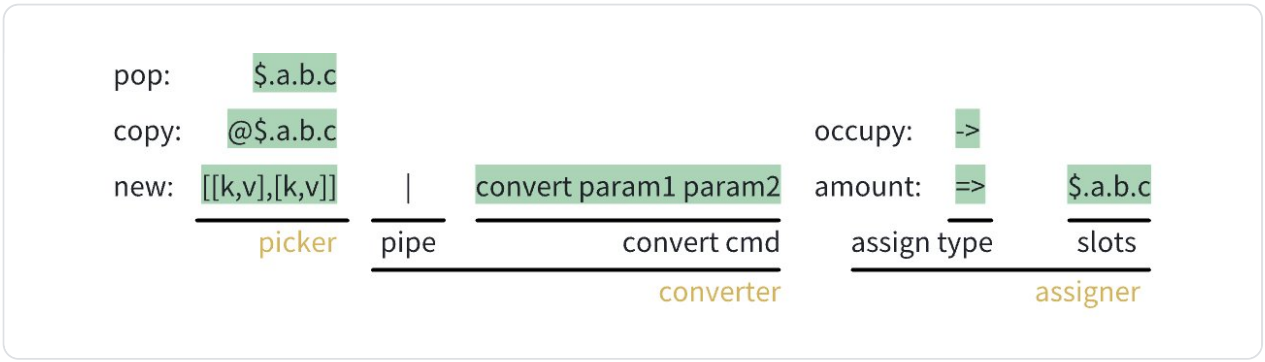
| | | | |
|---|---|---|---|
| | • `List` : Append `N` nodes to the list<br>• `Dict` : Mount the corresponding `N` Attribute - Value Pair positions, requiring `N` keys to be different | | |
| `M : N` | Group `M` and `N` according to a certain condition, so that the groups correspond one-to-one, and each group pair is the above four cases (complex, not supported) | - | - |

## Commands, with Python libraries

### Architecture (picker-converter-assigner)




### Command usage


pop: $.a.b.c
copy: @$.a.b.c
new: [[k,v],[k,v]] | convert param1 param2
occupy: ->
amount: => $.a.b.c
picker
pipe
convert cmd
converter
assign type
slots
assigner