CI passing Documentation passing code quality A release v3.4.0 issues 38 open license MIT

Inja is a template engine for modern C++, loosely inspired by jinja for python. It has an easy and yet powerful template syntax with all variables, loops, conditions, includes, callbacks, and comments you need, nested and combined as you like. Of course, everything is tested in CI on all relevant compilers. Here is what it looks like:

Inja 是一个现代 C++ 模板引擎，灵感来源于 Python 的 Jinja。它提供了一个简单而强大的模板语法，支持变量、循环、条件、包含、回调和注释等功能，这些功能可以按需嵌套和组合使用。所有的功能都在持续集成（CI）环境中进行了跨编译器测试。

- 一个为 C++ 设计的模板引擎，旨在提供类似于 Python 中 Jinja 模板引擎的功能和灵活性。它使得生成动态内容的文本（如 HTML、配置文件等）更加便捷和高效。
- Inja 使用了一种易于学习但功能强大的模板语法，支持多种编程常见的控制结构，如变量插值、循环遍历、条件判断等。这使得用户可以在模板中直接编写较为复杂的逻辑。
- Inja 支持诸如变量定义、循环语句、条件语句、文件包含、回调函数定义及模板注释等高级功能。用户可以根据需要将这些功能复合或嵌套使用，以实现更复杂的文本生成需求。
- Inja 的开发过程包括在多种编译器环境下的持续集成测试，确保其在不同的系统和编译器配置上都能稳定工作。这是现代软件开发中常见的质量保证手段，有助于及时发现和修复潜在的问题。

```
//  声明一个类型为json的变量
json data;
//  创建一个key-value(key = name, value = world)
data["name"] = "world";
//  使用inja中的render函数（返回为字符串类型）
inja::render("Hello {{ name }}!", data); // Returns "Hello world!"
```

Integration

Inja is a headers only library, which can be downloaded from the releases or directly from the `include/` or `single_include/` folder. Inja uses `nlohmann/json.hpp` (>= v3.8.0) as its single dependency, so make sure it can be included from `inja.hpp`. json can be downloaded here. Then integration is as easy as:

Inja 是一个仅头文件的 C++ 模板库，可以从其 GitHub 项目的发布页面或直接从项目目录中的 `include/` 或 `single_include/` 文件夹下载。Inja 的唯一依赖是 `nlohmann/json.hpp`（版本需不低于3.8.0），因此在使用 Inja 之前需要确保该 JSON 库可被 Inja 正确包含。可以从 nlohmann 的 JSON GitHub 页面下载这个库。集成 Inja 非常简单，只需确保正确包含相关头文件即可。

- Inja 作为一个仅头文件的库，意味着它不需要预编译成二进制形式，用户可以直接包含头文件到项目中使用，这简化了构建和链接过程。

- Inja 依赖于 `nlohmann/json.hpp` 库，这是一个广泛使用的现代 JSON 库，用于 C++。它提供了丰富的功能来处理 JSON 数据，例如解析和序列化。
- 集成 Inja 到项目中的过程非常简单，只需要下载 Inja 和它的依赖库 nlohmann/json，然后在项目中包含对应的头文件即可。
- 集成：在软件开发中，集成指的是将各个部分或者外部组件和系统结合在一起，形成一个可以协同工作的整体。对于库文件，这通常涉及将库文件包括到项目中，并确保项目可以识别并正确地使用这些库文件。

```
//  包含头文件，减少重复
#include <inja.hpp>

// Just for convenience ( 仅为了使用方便 )
using namespace inja;
```

If you are using the Meson Build System, then you can wrap this repository as a subproject.

If you are using Conan to manage your dependencies, have a look at this repository. Please file issues here if you experience problems with the packages.

You can also integrate inja in your project using Hunter, a package manager for C++.

If you are using vcpkg on your project for external dependencies, then you can use the inja package. Please see the vcpkg project for any issues regarding the packaging.

If you are using cget, you can install the latest development version with `cget install pantor/inja`. A specific version can be installed with `cget install pantor/inja@v2.1.0`.

On macOS, you can install inja via Homebrew and `brew install inja`.

If you are using conda, you can install the latest version from conda-forge with `conda install -c conda-forge inja`.

上述内容介绍了多种集成 Inja 库到 C++ 项目中的方法，包括使用不同的构建系统和包管理器。这些方法涵盖了 Meson Build System、Conan、Hunter、vcpkg、cget、Homebrew 和 conda 等多种流行工具，展示了 Inja 库在不同开发环境中的应用灵活性和可访问性。

- **Meson Build System**：可以将 Inja 作为一个子项目包含在使用 Meson 构建系统的项目中。
- **Conan**：通过 Conan 这个依赖管理工具，可以从一个特定的 Conan 仓库获取 Inja 库。如果在使用 Conan 包时遇到问题，用户被引导到相关的 GitHub 仓库提交问题。
- **Hunter**：Hunter 是专为 C++ 设计的包管理器，可以用来集成 Inja。
- **vcpkg**：vcpkg 是 Microsoft 推出的 C++ 库管理器，可以通过它来管理和安装 Inja 库。
- **cget**：cget 是一个基于 CMake 的包管理器，支持安装 Inja 的最新开发版本或指定的特定版本。
- **Homebrew**：macOS 用户可以通过 Homebrew 安装 Inja。
- **Conda**：使用 conda 包管理器的用户可以从 conda-forge 渠道安装 Inja 的最新版本。

解释上述提到的概念

- **Meson Build System**：一个开源的构建系统，旨在提供极速的构建性能和简洁的构建脚本语法。
- **Conan**：一个开源的 C/C++ 包管理器，用于自动下载、构建和链接依赖库。

- **Hunter**：Hunter 是一个基于 CMake 的开源 C++ 包管理器，专为 C++ 社区设计。
- **vcpkg**：一个 Microsoft 开发的 C++ 库管理器，用于 Windows, Linux 和 macOS 平台的库管理。
- **cget**：一个基于 CMake 的包管理器，允许用户从源码直接安装 C++ 库。
- **Homebrew**：macOS 和 Linux 下的包管理器，用户通过简单命令就可以安装软件。
- **Conda**：一个开源包和环境管理系统，支持多种语言，常用于科学计算领域。

Tutorial

This tutorial will give you an idea how to use inja. It will explain the most important concepts and give practical advices using examples and executable code. Beside this tutorial, you may check out the documentation.

这个教程将会教你如何使用inja。这个教程将会使用例子和可执行代码解释最重要的概念和给出实用的建议。除了这个教程，你可能需要查看相关的文档。

Template Rendering

The basic template rendering takes a template as a `std::string` and a `json` object for all data. It returns the rendered template as an `std::string`.

基本模板渲染将模板作为 `std::string` 和所有数据的 `json` 对象。它将渲染的模板作为 `std::string` 返回。

1. **模板 (`std::string`)**：一个包含特定标记或占位符的字符串，这些标记在渲染过程中会被实际数据替换。
2. **数据 (`json` 对象)**：一个 JSON 格式的对象，包含用于替换模板中占位符的具体数据。

返回值：函数通过处理输入的模板和数据，生成一个新的字符串，其中所有的模板标记已经被相应的数据替换。

```
//  创建一个类型为json的变量
json data;
//  向json变量中写入key-value（key = name, value = world）
data["name"] = "world";
//  使用inja中的渲染函数将对应的json函数进行渲染，返回字符串
render("Hello {{ name }}!", data); // Returns std::string "Hello world!"
//  将渲染后的字符串输出到特定文件中（everything is file）
render_to(std::cout, "Hello {{ name }}!", data); // Writes "Hello world!" to
stream
```

For more advanced usage, an environment is recommended.

对于更加高级的用法，一个environment是被推荐的。

```
//  创建一个类型为Environment的变量
Environment env;

// Render a string with json data（使用一个json数据来对一个字符串进行渲染）
std::string result = env.render("Hello {{ name }}!", data); // "Hello world!"
```

```
// Or directly read a template file (或者直接使用一个template file进行渲染)
Template temp = env.parse_template("./templates/greeting.txt");
std::string result = env.render(temp, data); // "Hello world!"

data["name"] = "Inja";
std::string result = env.render(temp, data); // "Hello Inja!"

// Or read the template file (and/or the json file) directly from the environment
result = env.render_file("./templates/greeting.txt", data);
result = env.render_file_with_json_file("./templates/greeting.txt",
"./data.json");

// Or write a rendered template file
env.write(temp, data, "./result.txt");
env.write_with_json_file("./templates/greeting.txt", "./data.json",
"./result.txt");
```

The environment class can be configured to your needs.

environment类是可以根据你的需求进行配置的。

```
// With default settings (使用environment类中的默认设置)
Environment env_default;

// With global path to template files and where files will be saved (包含模板文件的
全局路径以及文件保存位置)
Environment env_1 {"../path/templates/"};

// With separate input and output path (包含分开的输入和输出路径)
Environment env_2 {"../path/templates/", "../path/results/"};

// With other opening and closing strings (here the defaults)
env.set_expression("{{", "}}"); // Expressions
env.set_comment("{#", "#}"); // Comments
env.set_statement("{%", "%}"); // Statements {% %} for many things, see below
env.set_line_statement("##"); // Line statements ## (just an opener)
```

Variables

Variables are rendered within the `{{ ... }}` expressions.

```
json data;
data["neighbour"] = "Peter";
data["guests"] = {"Jeff", "Tom", "Patrick"};
data["time"]["start"] = 16;
data["time"]["end"] = 22;

// Indexing in array
render("{{ guests.1 }}", data); // "Tom"
```

```
// Objects
render("{{ time.start }} to {{ time.end + 1 }}pm", data); // "16 to 23pm"
```

If no variable is found, valid JSON is printed directly, otherwise an `inja::RenderError` is thrown.（如果没有找到变量，则直接打印有效的 JSON，否则会抛出“inja::RenderError”。）

Statements

Statements can be written either with the `{% ... %}` syntax or the `##` syntax for entire lines. Note that `##` needs to start the line without indentation. The most important statements are loops, conditions and file includes. All statements can be nested.

语句可以采用 `{% ... %}` 语法或整行 `##` 语法编写。请注意，`##` 需要以不缩进的方式开始行。最重要的语句是循环、条件和文件包含。所有语句都可以嵌套。

Loops

```
// Combining loops and line statements
render(R"(Guest List:
## for guest in guests
    {{ loop.index1 }}: {{ guest }}
## endfor )", data)

/* Guest List:
    1: Jeff
    2: Tom
    3: Patrick */
```

In a loop, the special variables `loop.index (number)`, `loop.index1 (number)`, `loop.is_first (boolean)` and `loop.is_last (boolean)` are defined. In nested loops, the parent loop variables are available e.g. via `loop.parent.index`. You can also iterate over objects like `{% for key, value in time %}`.

在循环中，定义了特殊变量 `loop.index (number)`、`loop.index1 (number)`、`loop.is_first (boolean)` 和 `loop.is_last (boolean)`。在嵌套循环中，父循环变量可通过`loop.parent.index` 等方式获取。您还可以迭代对象，例如 `{% for key, value in time %}`。

Conditions

Conditions support the typical if, else if and else statements. Following conditions are for example possible:

条件支持典型的 if、else if 和 else 语句。例如，以下条件是可能的：

```
// Standard comparisons with a variable
render("{% if time.hour >= 20 %}Serve{% else if time.hour >= 18 %}Make{% endif %}
dinner.", data); // Serve dinner.

// Variable in list
```

```
render("{% if neighbour in guests %}Turn up the music!{% endif %}", data); // Turn
up the music!

// Logical operations
render("{% if guest_count < (3+2) and all_tired %}Sleepy...{% else %}Keep going...
{% endif %}", data); // Sleepy...

// Negations
render("{% if not guest_count %}The End{% endif %}", data); // The End
```

Includes

You can either include other in-memory templates or from the file system.

```
// To include in-memory templates, add them to the environment first
inja::Template content_template = env.parse("Hello {{ neighbour }}!");
env.include_template("content", content_template);
env.render("Content: {% include \"content\" %}", data); // "Content: Hello Peter!"

// Other template files are included relative from the current file location
render("{% include \"footer.html\" %}", data);
```

If a corresponding template could not be found in the file system, the *include callback* is called:

```
// The callback takes the current path and the wanted include name and returns a
template
env.set_include_callback([&env](const std::string& path, const std::string&
template_name) {
  return env.parse("Hello {{ neighbour }} from " + template_name);
});

// You can disable to search for templates in the file system via
env.set_search_included_templates_in_files(false);
```

Inja will throw an `inja::RenderError` if an included file is not found and no callback is specified. To disable this error, you can call `env.set_throw_at_missing_includes(false)`.

Assignments

Variables can also be defined within the template using the set statment.

```
render("{% set new_hour=23 %}{{ new_hour }}pm", data); // "23pm"
render("{% set time.start=18 %}{{ time.start }}pm", data); // using json pointers
```

Assignments only set the value within the rendering context; they do not modify the json object passed into the `render` call.

Functions

A few functions are implemented within the inja template syntax. They can be called with

```
// Upper and lower function, for string cases
render("Hello {{ upper(neighbour) }}!", data); // "Hello PETER!"
render("Hello {{ lower(neighbour) }}!", data); // "Hello peter!"

// Range function, useful for loops
render("{% for i in range(4) %}{{ loop.index1 }}{% endfor %}", data); // "1234"
render("{% for i in range(3) %}{{ at(guests, i) }} {% endfor %}", data); // "Jeff
Tom Patrick "

// Length function (please don't combine with range, use list directly...)
render("I count {{ length(guests) }} guests.", data); // "I count 3 guests."

// Get first and last element in a list
render("{{ first(guests) }} was first.", data); // "Jeff was first."
render("{{ last(guests) }} was last.", data); // "Patir was last."

// Sort a list
render("{{ sort([3,2,1]) }}", data); // "[1,2,3]"
render("{{ sort(guests) }}", data); // "[\"Jeff\", \"Patrick\", \"Tom\"]"

// Join a list with a separator
render("{{ join([1,2,3], \" + \") }}", data); // "1 + 2 + 3"
render("{{ join(guests, \", \") }}", data); // "Jeff, Patrick, Tom"

// Round numbers to a given precision
render("{{ round(3.1415, 0) }}", data); // 3
render("{{ round(3.1415, 3) }}", data); // 3.142

// Check if a value is odd, even or divisible by a number
render("{{ odd(42) }}", data); // false
render("{{ even(42) }}", data); // true
render("{{ divisibleBy(42, 7) }}", data); // true

// Maximum and minimum values from a list
render("{{ max([1, 2, 3]) }}", data); // 3
render("{{ min([-2.4, -1.2, 4.5]) }}", data); // -2.4

// Convert strings to numbers
render("{{ int(\"2\") == 2 }}", data); // true
render("{{ float(\"1.8\") > 2 }}", data); // false

// Set default values if variables are not defined
render("Hello {{ default(neighbour, \"my friend\") }}!", data); // "Hello Peter!"
render("Hello {{ default(colleague, \"my friend\") }}!", data); // "Hello my
friend!"
```

```
// Access an objects value dynamically
render("{{ at(time, \"start\") }} to {{ time.end }}", data); // "16 to 22"

// Check if a key exists in an object
render("{{ exists(\"guests\") }}", data); // "true"
render("{{ exists(\"city\") }}", data); // "false"
render("{{ existsIn(time, \"start\") }}", data); // "true"
render("{{ existsIn(time, neighbour) }}", data); // "false"

// Check if a key is a specific type
render("{{ isString(neighbour) }}", data); // "true"
render("{{ isArray(guests) }}", data); // "true"
// Implemented type checks: isArray, isBoolean, isFloat, isInteger, isNumber,
isObject, isString,
```

Callbacks

You can create your own and more complex functions with callbacks. These are implemented with `std::function`, so you can for example use C++ lambdas. Inja `Arguments` are a vector of json pointers.

```
Environment env;

/*
 * Callbacks are defined by its:
 * - name,
 * - (optional) number of arguments,
 * - callback function.
 */
env.add_callback("double", 1, [](Arguments& args) {
    int number = args.at(0)->get<int>(); // Adapt the index and type of the
argument
    return 2 * number;
});

// You can then use a callback like a regular function
env.render("{{ double(16) }}", data); // "32"

// Inja falls back to variadic callbacks if the number of expected arguments is
omitted.
env.add_callback("argmax", [](Arguments& args) {
  auto result = std::max_element(args.begin(), args.end(), [](const json* a, const
json* b) { return *a < *b;});
  return std::distance(args.begin(), result);
});
env.render("{{ argmax(4, 2, 6) }}", data); // "2"
env.render("{{ argmax(0, 2, 6, 8, 3) }}", data); // "3"

// A callback without argument can be used like a dynamic variable:
std::string greet = "Hello";
env.add_callback("double-greetings", 0, [greet](Arguments args) {
```

```
    return greet + " " + greet + "!";
});
env.render("{{ double-greetings }}", data); // "Hello Hello!"
```

You can also add a void callback without return variable, e.g. for debugging:

```
env.add_void_callback("log", 1, [greet](Arguments args) {
    std::cout << "logging: " << args[0] << std::endl;
});
env.render("{{ log(neighbour) }}", data); // Prints nothing to result, only to
cout...
```

Template Inheritance

Template inheritance allows you to build a base *skeleton* template that contains all the common elements and defines blocks that child templates can override. Lets show an example: The base template

```
<!DOCTYPE html>
<html>
<head>
  {% block head %}
  <link rel="stylesheet" href="style.css" />
  <title>{% block title %}{% endblock %} - My Webpage</title>
  {% endblock %}
</head>
<body>
  <div id="content">{% block content %}{% endblock %}</div>
</body>
</html>
```

contains three `blocks` that child templates can fill in. The child template

```
{% extends "base.html" %}
{% block title %}Index{% endblock %}
{% block head %}
  {{ super() }}
  <style type="text/css">
    .important { color: #336699; }
  </style>
{% endblock %}
{% block content %}
  <h1>Index</h1>
  <p class="important">
    Welcome to my blog!
  </p>
{% endblock %}
```

calls a parent template with the `extends` keyword; it should be the first element in the template. It is possible to render the contents of the parent block by calling `super()`. In the case of multiple levels of `{% extends %}`, super references may be called with an argument (e.g. `super(2)`) to skip levels in the inheritance tree.

Whitespace Control

In the default configuration, no whitespace is removed while rendering the file. To support a more readable template style, you can configure the environment to control whitespaces before and after a statement automatically. While enabling `set_trim_blocks` removes the first newline after a statement, `set_lstrip_blocks` strips tabs and spaces from the beginning of a line to the start of a block.

```
Environment env;
env.set_trim_blocks(true);
env.set_lstrip_blocks(true);
```

With both `trim_blocks` and `lstrip_blocks` enabled, you can put statements on their own lines. Furthermore, you can also strip whitespaces for both statements and expressions by hand. If you add a minus sign (`-`) to the start or end, the whitespaces before or after that block will be removed:

```
render("Hello       {{- name -}}     !", data); // "Hello Inja!"
render("{% if neighbour in guests -%}   I was there{% endif -%}   !", data); //
Renders without any whitespaces
```

Stripping behind a statement or expression also removes any newlines.

Comments

Comments can be written with the `{# ... #}` syntax.

```
render("Hello{# Todo #}!", data); // "Hello!"
```

Exceptions

Inja uses exceptions to handle ill-formed template input. However, exceptions can be switched off with either using the compiler flag `-fno-exceptions` or by defining the symbol `INJA_NOEXCEPTION`. In this case, exceptions are replaced by `abort()` calls.

Supported compilers

Inja uses the `string_view` feature of the C++17 STL. Currently, the following compilers are tested:

- GCC 8 - 11 (and possibly later)
- Clang 5 - 12 (and possibly later)
- Microsoft Visual C++ 2017 15.0 - 2022 (and possibly later)

A list of supported compiler / os versions can be found in the CI definition.