**使用**

Below is quick example how to use mustache.js:

var view = {

title: "Joe",

calc: function () {

return 2 + 4;

}

};

var output = Mustache.render("{{title}} spends {{calc}}", view);

In this example, the Mustache.render function takes two parameters: 1) the [mustache](http://mustache.github.com/) template and 2) a view object that contains the data and code needed to render the template.

**模板（变量、片段、注释）**

A [mustache](http://mustache.github.com/) template is a string that contains any number of mustache tags. Tags are indicated by the double mustaches that surround them. {{person}} is a tag, as is {{#person}}. In both examples we refer to person as the tag's key.

在mustache.js中有好几种tags可用。

**变量**

最基本的tag类型是一个简单的变量。一个{{name}} tag将显示当前上下文中的name键。如果没有这个键，将什么都不显示。所有的变量都会默认做HTML编码处理。如果你不想值作为HTML编码处理，那么可以使用{{{name}}}。或者使用{{&name}}也可。

视图:

{

"name": "Chris",

"company": "<b>GitHub</b>"

}

模板:

\* {{name}}

\* {{age}}

\* {{company}}

\* {{{company}}}

\* {{&company}}

输出:

\* Chris

\*

\* &lt;b&gt;GitHub&lt;/b&gt;

\* <b>GitHub</b>

\* <b>GitHub</b>

也可以使用JavaScript的点号运算符来访问视图中对象的属性。

视图:

{

"name": {

"first": "Michael",

"last": "Jackson"

},

"age": "RIP"

}

模板:

\* {{name.first}} {{name.last}}

\* {{age}}

输出:

\* Michael Jackson

\* RIP

**片段**

片段根据当前上下文中key的值，将一段文本重复显示几遍。

一个片段以#还是并以/结束。如{{#person}}开始一个person片段，{{/person}}结束person片段。

片段的行为取决于模板中key的值。

**False值或空Lists**

如果person键不存在，或存在但是值是null，undefined或False，或是个空列表，那么整个片段不显示

视图:

{

"person": false

}

模板:

Shown.

{{#person}}

Never shown!

{{/person}}

输出:

Shown.

**非空列表**

整个片段将显示一次或多次。片段中的上下文指向当前显示的列表元素。

视图:

{

"stooges": [

{ "name": "Moe" },

{ "name": "Larry" },

{ "name": "Curly" }

]

}

模板:

{{#stooges}}

<b>{{name}}</b>

{{/stooges}}

输出:

<b>Moe</b>

<b>Larry</b>

<b>Curly</b>

如果视图是一个字符串的数组，那么可以使用“.”来显示当前的字符串元素。

视图:

{

"musketeers": ["Athos", "Aramis", "Porthos", "D'Artagnan"]

}

模板:

{{#musketeers}}

\* {{.}}

{{/musketeers}}

输出:

\* Athos

\* Aramis

\* Porthos

\* D'Artagnan

如果视图元素是一个函数，那么函数中的this指向当前上下文。

视图:

{

"beatles": [

{ "firstName": "John", "lastName": "Lennon" },

{ "firstName": "Paul", "lastName": "McCartney" },

{ "firstName": "George", "lastName": "Harrison" },

{ "firstName": "Ringo", "lastName": "Starr" }

],

"name": function () {

return this.firstName + " " + this.lastName;

}

}

视图:

{{#beatles}}

\* {{name}}

{{/beatles}}

输出:

\* John Lennon

\* Paul McCartney

\* George Harrison

\* Ringo Starr

**函数**

如果片段是一个函数，那么它的第一个参数是片段的模板文本（没有嵌入变量值的）。第二个参数是一个特殊的呈现函数（该函数使用当前视图作为view参数）。函数中this指向当前的视图

视图:

{

"name": "Tater",

"bold": function () {

return function (text, render) {

return "<b>" + render(text) + "</b>";

}

}

}

模板:

{{#bold}}Hi {{name}}.{{/bold}}

输出:

<b>Hi Tater.</b>

**反向片段**

以^开始，并以/结束。如{{^person}} {{/person}}，这种情况下，重要当person是null，undefined，false或空列表时才显示。

视图:

{

"repos": []

}

模板:

{{#repos}}<b>{{name}}</b>{{/repos}}

{{^repos}}No repos :({{/repos}}

输出:

No repos :(

**注释**

用感叹号标识，如{{!person}}

模板:

<h1>Today{{! ignore me }}.</h1>

将如下显示:

<h1>Today.</h1>

注释可以包含换行符

**局部模板**

局部变量使用大于号“>”标识，如{{>box}}

局部模板在运行时呈现，所以可能有递归局部模板。只要避免死循环即可。

They also inherit the calling context. Whereas in ERB you may have this:

<%= partial :next\_more, :start => start, :size => size %>

mustache只需要这个:

{{> next\_more}}

Why? Because the next\_more.mustache file will inherit the size and start variables from the calling context. In this way you may want to think of partials as includes, or template expansion, even though it's not literally true.

For example, this template and partial:

base.mustache:

<h2>Names</h2>

{{#names}}

{{> user}}

{{/names}}

user.mustache:

<strong>{{name}}</strong>

Can be thought of as a single, expanded template:

<h2>Names</h2>

{{#names}}

<strong>{{name}}</strong>

{{/names}}

In mustache.js an object of partials may be passed as the third argument to Mustache.render. The object should be keyed by the name of the partial, and its value should be the partial text.

**Set Delimiter**

Set Delimiter tags start with an equals sign and change the tag delimiters from {{ and }} to custom strings.

Consider the following contrived example:

\* {{ default\_tags }}

{{=<% %>=}}

\* <% erb\_style\_tags %>

<%={{ }}=%>

\* {{ default\_tags\_again }}

Here we have a list with three items. The first item uses the default tag style, the second uses ERB style as defined by the Set Delimiter tag, and the third returns to the default style after yet another Set Delimiter declaration.

According to [ctemplates](http://google-ctemplate.googlecode.com/svn/trunk/doc/howto.html), this "is useful for languages like TeX, where double-braces may occur in the text and are awkward to use for markup."

Custom delimiters may not contain whitespace or the equals sign.

**Compiled Templates**

Mustache templates can be compiled into JavaScript functions using Mustache.compile for improved rendering performance.

If you have template views that are rendered multiple times, compiling your template into a JavaScript function will minimise the amount of work required for each re-render.

Pre-compiled templates can also be generated server-side, for delivery to the browser as ready to use JavaScript functions, further reducing the amount of client side processing required for initialising templates.

**Mustache.compile**

Use Mustache.compile to compile standard Mustache string templates into reusable Mustache template functions.

var compiledTemplate = Mustache.compile(stringTemplate);

The function returned from Mustache.compile can then be called directly, passing in the template data as an argument (with an object of partials as an optional second parameter), to generate the final output.

var templateOutput = compiledTemplate(templateData);

**Mustache.compilePartial**

Template partials can also be compiled using the Mustache.compilePartial function. The first parameter of this function, is the name of the partial as it appears within parent templates.

Mustache.compilePartial('partial-name', stringTemplate);

Compiled partials are then available to both Mustache.render and Mustache.compile.

**Plugins for JavaScript Libraries**

mustache.js may be built specifically for several different client libraries, including the following:

* [jQuery](http://jquery.com/)
* [MooTools](http://mootools.net/)
* [Dojo](http://www.dojotoolkit.org/)
* [YUI](http://developer.yahoo.com/yui/)
* [qooxdoo](http://qooxdoo.org/)

These may be built using [Rake](http://rake.rubyforge.org/) and one of the following commands:

$ rake jquery

$ rake mootools

$ rake dojo

$ rake yui3

$ rake qooxdoo

**Testing**

The mustache.js test suite uses the [mocha](http://visionmedia.github.com/mocha/) testing framework. In order to run the tests you'll need to install [node](http://nodejs.org/). Once that's done you can install mocha using [npm](http://npmjs.org/).

$ npm install -g mocha

Then run the tests.

$ mocha test

The test suite consists of both unit and integration tests. If a template isn't rendering correctly for you, you can make a test for it by doing the following:

1. Create a template file named mytest.mustache in the test/\_files directory. Replace mytest with the name of your test.
2. Create a corresponding view file named mytest.js in the same directory. This file should contain a JavaScript object literal enclosed in parentheses. See any of the other view files for an example.
3. Create a file with the expected output in mytest.txt in the same directory.

Then, you can run the test with:

$ TEST=mytest mocha test/render\_test.js