Шаблонизатор LoDash

В этой главе мы рассмотрим *шаблонизацию* – удобный способ генерации HTML по «шаблону» и данным.

Большинство виджетов, которые мы видели ранее, получают готовый HTML/DOM и «оживляют» его. Это типичный случай в сайтах, где JavaScript – на ролях «второго помощника». Разметка, CSS уже есть, от JavaScript, условно говоря, требуются лишь обработчики, чтобы менюшки заработали.

Но в сложных интерфейсах разметка изначально отсутствует на странице. Компоненты генерируют свой DOM сами, динамически, на основе данных, полученных с сервера или из других источников.

Зачем нужны шаблоны?

Ранее мы уже видели код `Menu`, который сам создаёт свой элемент:

```
function Menu(options) {
  // ... приведены только методы для генерации DOM ...

  function render() {
    elem = document.createElement('div');
    elem.className = "menu";

    var titleElem = document.createElement('span');
    elem.appendChild(titleElem);
    titleElem.className = "title";
    titleElem.textContent = options.title;

    elem.onmousedown = function() {
      return false;
    };

    elem.onclick = function(event) {
      if (event.target.closest('.title')) {
        toggle();
      }
    }

  }

  function renderItems() {
    var items = options.items || [];
    var list = document.createElement('ul');
    items.forEach(function(item) {
      var li = document.createElement('li');
      li.textContent = item;
      list.appendChild(li);
    });
    elem.appendChild(list);
  }
  // ...
}
```

Понятен ли этот код? Очевидно ли, какой HTML генерируют методы `render`, `renderItems`?

С первого взгляда – вряд ли. Нужно как минимум внимательно посмотреть и продумать код, чтобы разобраться, какая именно DOM-структура создаётся.

...А что, если нужно изменить создаваемый HTML? ...А что, если эта задача досталась не программисту, который написал этот код, а верстальщику, который с HTML/CSS проекта знаком отлично, но этот JS-код видит впервые? Вероятность ошибок при этом зашкаливает за все разумные пределы.

К счастью, генерацию HTML можно упростить. Для этого воспользуемся библиотекой шаблонизации.

Пример шаблона

Шаблон – это строка в специальном формате, которая путём подстановки значений (текст сообщения, цена и т.п.) и выполнения встроенных фрагментов кода превращается в DOM/HTML.

Пример шаблона для меню:

```
<div class="menu">
  <span class="title"><%-title%></span>
  <ul>
    <% items.forEach(function(item) { %>
    <li><%-item%></li>
    <% }); %>
  </ul>
</div>
```

Как видно, это обычный HTML, с вставками вида `<% ... %>`.

Для работы с таким шаблоном используется специальная функция `_.template`, которая предоставляется фреймворком LoDash, её синтаксис мы подробно посмотрим далее.

Пример использования `_.template` для генерации HTML с шаблоном выше:

```
// сгенерировать HTML, используя шаблон tmpl (см. выше)
// с данными title и items
var html = _.template(tmpl)({
  title: "Сладости",
  items: [
    "Торт",
    "Печенье",
    "Пирожное"
  ]
});
```

Значение `html` в результате:

```
<div class="menu">
  <span class="title">Сладости</span>
```

```
  <ul>
    <li>Торт</li>
    <li>Печенье</li>
    <li>Сладости</li>
  </ul>
</div>
```

Этот код гораздо проще, чем JS-код, не правда ли? Шаблон очень наглядно показывает, что в итоге должно получиться. В отличие от кода, в шаблоне первичен текст, а вставок кода обычно мало.

Давайте подробнее познакомимся с `_.template` и синтаксисом шаблонов.

> **Holy war detected!**
>
> Способов шаблонизации и, в особенности, синтаксисов шаблонов, примерно столько же, сколько способов поймать льва в пустыне. Иначе говоря... много.
>
> Эта глава – совершенно не место для священных войн на эту тему.
>
> Далее будет более полный обзор типов шаблонных систем, применяемых в JavaScript, но начнём мы с `_.template`, поскольку эта функция проста, быстра и демонстрирует приёмы, используемые в целом классе шаблонных систем, активно используемых в самых разных JS-проектах.

Синтаксис шаблона

Шаблон представляет собой строку со специальными разделителями, которых всего три:

`<% code %>` – код

Код между разделителями `<% ... %>` будет выполнен «как есть»

`<%= expr %>` – для вставки `expr` как HTML

Переменная или выражение внутри `<%= ... %>` будет вставлено «как есть». Например: `<%=title %>` вставит значение переменной `title`, а `<%=2+2%>` вставит `4`.

`<%- expr %>` – для вставки `expr` как текста

Переменная или выражение внутри `<%- ... %>` будет вставлено «как текст», то есть с заменой символов `< > & " '` на соответствующие HTML-entities.

Например, если `expr` содержит текст `<br>`, то при `<%-expr%>` в результат попадёт, в отличие от `<%=expr%>`, не HTML-тег `<br>`, а текст `<br>`.

Функция _.template

Для работы с шаблоном в библиотеке LoDash есть функция `_.template(tmpl, data, options)`.

Её аргументы:

`tmpl`

Шаблон.

`options`

Необязательные настройки, например можно поменять разделители.

Эта функция запускает «компиляцию» шаблона `tmpl` и возвращает результат в виде функции, которую далее можно запустить с данными и получить строку-результат.

Вот так:

```
// Шаблон
var tmpl = _.template('<span class="title"><%=title%></span>');

// Данные
var data = {
  title: "Заголовок"
};

// Результат подстановки
alert( tmpl(data) ); // <span class="title">Заголовок</span>
```

Пример выше похож на операцию «поиск-и-замена»: шаблон просто заменил `<%=title%>` на значение свойства `data.title`.

Но возможность вставки JS-кода делает шаблоны сильно мощнее.

Например, вот шаблон для генерации списка от `1` до `count`:

```
// используется \, чтобы объявить многострочную переменную-текст шаблона
var tmpl = '<ul>\
  <% for (var i=1; i<=count; i++) { %> \
    <li><%=i%></li> \
  <% } %>\
</ul>';
alert( _.template(tmpl)({count: 5}) ); // <ul><li>1</li><li>2</li>...</ul>
```

Здесь в результат попал сначала текст `<ul>`, потом выполнился код `for`, который последовательно сгенерировал элементы списка, и затем список был закрыт `</ul>`.

Хранение шаблона в документе

Шаблон – это многострочный HTML-текст. Записывать его прямо в скрипте – неудобно.

Один из альтернативных способов объявления шаблона – записать его в HTML, в тег `<script>` с нестандартным `type`, например `"text/template"`:

```
<script type="text/template" id="menu-template">
<div class="menu">
```

```
  <span class="title"><%-title%></span>
</div>
</script>
```

Если `type` не знаком браузеру, то содержимое такого скрипта игнорируется, однако оно доступно при помощи `innerHTML`:

```
var template = document.getElementById('menu-template').innerHTML;
```

В данном случае выбран `type="text/template"`, однако подошёл бы и любой другой нестандартный, например `text/html`. Главное, что браузер такой скрипт никак не обработает. То есть, это всего лишь способ передать строку шаблона в HTML.

Полный пример цикла с подключением библиотеки и шаблоном в HTML:

```
<!-- библиотека LoDash -->
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<!-- шаблон для списка от 1 до count -->
<script type="text/template" id="list-template">
<ul>
  <% for (var i=1; i<=count; i++) { %>
  <li><%=i%></li>
  <% } %>
</ul>
</script>

<script>
  var tmpl = _.template(document.getElementById('list-template').innerHTML);

  // ..а вот и результат
  var result = tmpl({count: 5});
  document.write( result );
</script>
```

Как работает функция _.template?

Понимание того, как работает `_.template`, очень важно для отладки ошибок в шаблонах.

Как обработка шаблонов устроена внутри? За счёт чего организована возможность перемежать с текстом произвольный JS-код?

Оказывается, очень просто.

Вызов `_.template(str)` разбивает строку `str` по разделителям и, при помощи `new Function` создаёт на её основе JavaScript-функцию. Тело этой функции создаётся таким образом, что код, который в шаблоне оформлен как `<% ... %>` – попадает в неё «как есть», а переменные и текст прибавляются к специальному временному «буферу», который в итоге возвращается.

Взглянем на пример:

```
var compiled = _.template("<h1><%=title%></h1>");

alert( compiled );
```

Функция `compiled`, которую вернул вызов `_template` из этого примера, выглядит примерно так:

```
function(obj) {
  obj || (obj = {});
  var __t, __p = '', __e = _.escape;
  with(obj) { \
    __p += '<h1>' +
      ((__t = (title)) == null ? '' : __t) +
      '</h1>';
  }
  return __p
}
```

Она является результатом вызова `new Function("obj", "код")`, где `код` динамическим образом генерируется на основе шаблона:

1. Вначале в коде идёт «шапка» – стандартное начало функции, в котором объявляется переменная `__p`. В неё будет записываться результат.
2. Затем добавляется блок `with(obj) { ... }`, внутри которого в `__p` добавляются фрагменты HTML из шаблона, а также переменные из выражений `<%=...%>`. Код из `<%...%>` копируется в функцию «как есть».
3. Затем функция завершается, и `return __p` возвращает результат.

При вызове этой функции, например `compiled({title: "Заголовок"})`, она получает объект данных как `obj`, здесь это `{title: "Заголовок"}`, и если внутри `with(obj) { .. }` обратиться к `title`, то по правилам конструкции with это свойство будет получено из объекта.

Можно и без `with`

Конструкция `with` является устаревшей, но в данном случае она полезна.

Так как функция создаётся через `new Function("obj", "код")` то:

- Она работает в глобальной области видимости, не имеет доступа к внешним локальным переменным.
- Внешний `use strict` на такую функцию не влияет, то есть даже в строгом режиме шаблон продолжит работать.

Если мы всё же не хотим использовать `with` – нужно поставить второй параметр – `options`, указав параметр `variable` (название переменной с данными).

Например:

```
alert( _.template("<h1><%=menu.title%></h1>", {variable: "menu"}) );
```

Результат:

```
function(menu) {
  var __t, __p = '';
  __p += '<h1>' +
    ((__t = (menu.title)) == null ? '' : __t) +
    '</h1>';
  return __p
}
```

При таком подходе переменная `title` уже не будет искаться в объекте данных автоматически, поэтому нужно будет обращаться к ней как `<%=menu.title%>`.

Кеширование скомпилированных шаблонов

Чтобы не компилировать один и тот же шаблон много раз, результаты обычно кешируют.

Например, глобальная функция `getTemplate("menu-template")` может доставать шаблон из HTML, компилировать, результат запоминать и сразу отдавать при последующих обращениях к тому же шаблону.

Меню на шаблонах

Рассмотрим для наглядности полный пример меню на шаблонах.

HTML (шаблоны):

```
<script type="text/template" id="menu-template">
<div class="menu">
  <span class="title"><%-title%></span>
</div>
</script>

<script type="text/template" id="menu-list-template">
<ul>
  <% items.forEach(function(item) { %>
  <li><%-item%></li>
  <% }); %>
</ul>
</script>
```

JS для создания меню:

```
var menu = new Menu({
  title: "Сладости",
  // передаём также шаблоны
  template: _.template(document.getElementById('menu-template').innerHTML),
  listTemplate: _.template(document.getElementById('menu-list-template').innerHTML),
  items: [
    "Торт",
    "Пончик",
    "Пирожное",
    "Шоколадка",
    "Мороженое"
  ]
});

document.body.appendChild(menu.getElem());
```

JS код Menu :

```
function Menu(options) {
  var elem;

  function getElem() {
    if (!elem) render();
    return elem;
  }

  function render() {
    var html = options.template({
      title: options.title
    });

    elem = document.createElement('div');
    elem.innerHTML = html;
    elem = elem.firstElementChild;

```

```
    elem.onmousedown = function() {
      return false;
    }

    elem.onclick = function(event) {
      if (event.target.closest('.title')) {
        toggle();
      }
    }
  }

  function renderItems() {
    if (elem.querySelector('ul')) return;

    var listHtml = options.listTemplate({
      items: options.items
    });
    elem.insertAdjacentHTML("beforeEnd", listHtml);
  }

  function open() {
    renderItems();
    elem.classList.add('open');
  };

  function close() {
    elem.classList.remove('open');
  };

  function toggle() {
    if (elem.classList.contains('open')) close();
    else open();
  };

  this.getElem = getElem;
  this.toggle = toggle;
  this.close = close;
  this.open = open;
}
```

Здесь два шаблона. Первый мы уже разобрали, посмотрим теперь на список `ul/li` :

```
<ul>
  <% items.forEach(function(item) { %>
  <li><%-item%></li>
  <% }); %>
</ul>
```

Если разбить шаблон для списка элементов по разделителям, то он будет таким:

- `<ul>` – текст
- `<% items.forEach(function(item) { %>` – код

- `<li>` – текст
- `<%-item%>` – вставить значение `item` с экранированием
- `</li>` – текст
- `<% }); %>` – код
- `</ul>` – текст

Вот функция, которую возвратит `_.template(tmpl)` для этого шаблона:

```
function(obj) {
  obj || (obj = {});
  var __t, __p = '', __e = _.escape;
  with(obj) {
    __p += '\n<ul>\n  ';
    items.forEach(function(item) {
      __p += '\n  <li>' +
        __e(item) + // <%-item%> экранирование функцией _.escape
        '</li>\n  ';
    });
    __p += '\n</ul>\n';
  }
  return __p
}
```

Как видно, она один-в-один повторяет код и вставляет текст в переменную `__p` . При этом выражение в `<%-...%>` обёрнуто в вызов _.escape, который заменяет спецсимволы HTML на их текстовые варианты.

Отладка шаблонов

Что, если в шаблоне ошибка? Например, синтаксическая. Конечно, ошибки будут возникать, куда же без них.

Шаблон компилируется в функцию, ошибка будет либо при компиляции, либо позже, в процессе её выполнения. В различных шаблонных системах есть свои средства отладки, `_.template` тут не блистает.

Но и здесь можно кое-что отладить. При ошибке, если она не синтаксическая, отладчик при этом останавливается где-то посередине «страшной» функции.

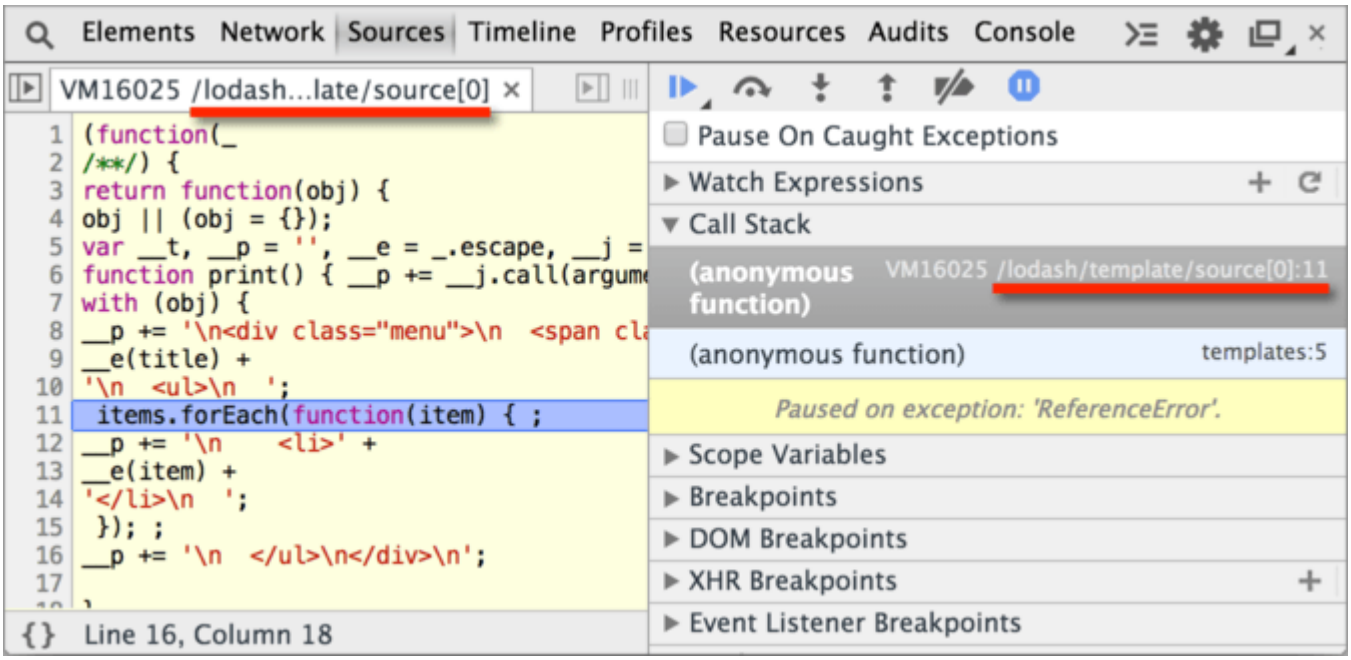
Попробуйте сами запустить пример с открытыми инструментами разработчика и *включённой* опцией «остановка при ошибке»:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.3.0/lodash.js"></script>

<script type="text/template" id="menu-template">
<div class="menu">
  <span class="title"><%-title%></span>
  <ul>
  <% items.forEach(function(item) { %>
    <li><%-item%></li>
  <% }); %>
  </ul>
```

```
</div>
</script>

<script>
  var tmpl = _.template(document.getElementById('menu-template').innerHTML);

  var result = tmpl({ title: "Заголовок" });

  document.write(result);
</script>
```

В шаблоне допущена ошибка, поэтому отладчик остановит выполнение.

В Chrome картина будет примерно такой:



Библиотека LoDash пытается нам помочь, подсказать, в каком именно шаблоне произошла ошибка. Ведь из функции это может быть неочевидно.

Для этого она добавляет к шаблонам специальный идентификатор sourceURL, который служит аналогом «имени файла». На картинке он отмечен красным.

По умолчанию `sourceURL` имеет вид `/lodash/template/source[N]`, где `N` – постоянно увеличивающийся номер шаблона. В данном случае мы можем понять, что эта функция получена при самой первой компиляции.

Это, конечно, лучше чем ничего, но, как правило, его имеет смысл заменить `sourceURL` на свой, указав при компиляции дополнительный параметр `sourceURL`:

```
...
var compiled = _.template(tmpl, {sourceURL: '/template/menu-template'});
...
```

Попробуйте запустить исправленный пример и вы увидите в качестве имени файла `/template/menu-template`.

Не определена переменная – ошибка

Кстати говоря, а в чём же здесь ошибка?

...А в том, что переменная `items` не передана в шаблон. При доступе к неизвестной переменной JavaScript генерирует ошибку.

Самый простой способ это обойти – обращаться к необязательным переменным через `obj`, например `<%=obj.items%>`. Тогда в случае `undefined` просто ничего не будет выведено. Но в данном случае реакция совершенно адекватна, так как для меню список опций `items` является обязательным.

Итого

Шаблоны полезны для того, чтобы отделить HTML от кода. Это упрощает разработку и поддержку.

В этой главе подробно разобрана система шаблонизации из библиотеки LoDash:

- Шаблон – это строка со специальными вставками кода `<% ... %>` или переменных `<%- expr ->`, `<%= expr ->`.
- Вызов `_.template(tmpl)` превращает шаблон `tmpl` в функцию, которой в дальнейшем передаются данные – и она генерирует HTML с ними.

В этой главе мы рассмотрели хранение шаблонов в документе, при помощи `<script>` с нестандартным `type`. Конечно, есть и другие способы, можно хранить шаблоны и в отдельном файле, если шаблонная система или система сборки проектов это позволяют.

Шаблонных систем много. Многие основаны на схожем принципе – генерации функции из строки, например:

- EJS
- Jade
- Handlebars

Есть и альтернативный подход – шаблонная система получает «образец» DOM-узла и клонирует его вызовом `cloneNode(true)`, каждый раз изменяя что-то внутри. В отличие от подхода, описанного выше, это будет работать не с произвольной строкой текста, а только и именно с DOM-узлами. Но в некоторых ситуациях у него есть преимущество.

Такой подход используется во фреймворках:

- AngularJS
- Knockout.JS