1-го августа 2019

Эволюция шаблонных систем для JavaScript

Материал на этой странице устарел, поэтому скрыт из оглавления сайта.

Различных шаблонных систем – много.

Они постепенно эволюционировали и развивались.

В этой главе мы разберём, как шёл этот процесс, какие шаблонки «родились», какие бонусы нам даёт использование той или иной шаблонной системы.

Микрошаблоны

Микрошаблоны (англ. microtemplate) мы уже видели на примере `_.template` .

Это HTML со вставками переменных и произвольным JS.

Пример:

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

Шаблонная система компилирует этот код в JavaScript-функцию с минимальными модификациями, и она уже, запустившись с данными, генерирует результат.

Достоинства и недостатки такого подхода:

Недостатки

- Жёстко привязан к языку JavaScript.
- При ошибке в шаблоне приходится лезть внутрь «страшной» функции

Достоинства

- Простая и быстрая шаблонная система
- Внутри JS-функции доступен полноценный браузерный отладчик, функция хоть и страшна, но понятна.

Код в шаблоне

Включение произвольного JS-кода в шаблон, в теории, позволяет делать в нём всё, что угодно. Но обратная сторона медали – шаблон вместо внятного HTML может стать адским нагромождением разделителей вперемешку с вычислениями. Что рекомендуется делать в шаблонах, а что нет?

Можно разделить код на два типа с точки зрения шаблонизации:

- **Бизнес-логика** – код, *формирующий данные*, основной код приложения.
- **Презентационная логика** – код, описывающий, как *показываются данные*.

Например, код, получающий данные с сервера для вывода в таблице – бизнес-логика, а код, форматирующий даты для вывода – презентационная логика.

В шаблонах допустима лишь презентационная логика.

Кросс-платформенность

Зачастую, нужно использовать один и тот же шаблон и в браузере и на сервере.

Например, серверный код генерирует HTML со списком сообщений, а JavaScript на клиенте добавляет к нему новые по мере появления.

...Но как использовать на сервере шаблон с JavaScript, если его основной язык – PHP, Ruby, Java?

Эту проблему можно обойти. На сервер, использующем PHP, Ruby, Java или какой-то другой язык, дополнительно ставится виртуальная машина V8 и настраивается интеграция с ней. Почти все платформы это умеют.

После этого становится возможным запускать JavaScript-шаблоны и передавать им данные в виде объектов, массивов и так далее.

Этот подход может показаться искусственным, но на самом деле он вполне жизнеспособен и используется в ряде крупных проектов.

Прекомпиляция

Эта шаблонка и большинство других систем, которые мы рассмотрим далее, допускают *прекомпиляцию*.

То есть, можно заранее, до выкладывания сайта на «боевой сервер», обработать шаблоны, создать из них JS-функции, объединить их в единый файл и далее, в «боевом окружении» использовать уже их.

Современные системы сборки (brunch, grunt с плагинами и другие) позволяют делать это удобно, а также хранить шаблоны в разных файлах, каждый – в нужной директории с JS-кодом для виджета.

Хелперы и фильтры

JavaScript-вставки не всегда просты и элегантны. Иногда, чтобы что-то сделать, нужно написать порядочно кода.

Для того, чтобы сделать шаблоны компактнее и проще, в них стали добавлять фильтры и хелперы.

- **Хелпер** (англ. helper) – вспомогательная функция, которая доступна в шаблонах и используется для решения часто возникающих задач.

  В `_.template`, чтобы объявить хелпер, можно просто сделать глобальную функцию. Но это слишком грубо, так не делают. Гораздо лучше – использовать объект `_.templateSettings.imports`, в котором можно указать, какие функции добавлять в шаблоны, или опцию `imports` для `_.template`.

  Пример хелпера – функция `t(phrase)`, которая переводит `phrase` на текущий язык:

  ```
  1  _.templateSettings.imports.t = function(phrase) {
  2    // обычно функция перевода немного сложнее, но здесь это не важно
  3    if (phrase == "Hello") return "Привет";
  4  }
  5
  6  // в шаблоне используется хелпер t для перевода
  7  var compiled = _.template("<div><%=t('Hello')%></div>");
  8  alert( compiled() ); // <div>Привет</div>
  ```

  Такой хелпер очень полезен для мультиязычных сайтов, когда один шаблон нужно выводить на десяти языках. Нечто подобное используется почти во всех языках и платформах, не только в JavaScript.

- **Фильтр** – это функция, которая трансформирует данные, например, форматирует дату, сортирует элементы массива и так далее.

  Обычно для фильтров предусмотрен специальный «особо простой и короткий» синтаксис.

  Например, в системе шаблонизации EJS, которая по сути такая же, но мощнее, чем `_.template`, фильтры задаются через символ `|`, внутри разделителя `<%=: ... %>`.

  Чтобы вывести `item` с большой буквы, можно вместо `<%=item%>` написать `<%=: item | capitalize %>`. Чтобы выводить отсортированный массив, можно использовать `<%=: items | sort %>` и так далее.

Свой язык

Для того, чтобы сделать шаблон ещё короче, а также с целью «отвязать» их от JavaScript, ряд шаблонных систем предлагают свой язык.

Например:

- Mustache
- Handlebars
- Closure Templates
- …тысячи их…

Шаблон для меню в Handlerbars, к примеру, будет выглядеть так:

```
<div class="menu">
  <span class="title">{{title}}</span>
  <ul>
    {{#each items}}
    <li>{{item}}</li>
    {{/each}}
  </ul>
<div>
```

Как видно, *вместо* JavaScript-конструкций здесь используются хелперы. В примере выше `{{#each}} ... {{/each}` – «блочный» хелпер: он показывает своё содержимое для каждого элемента `items` и является альтернативой `forEach`.

Есть и другие встроенные в шаблонизатор хелперы, можно легко делать свои.

Использование такого шаблона:

```
// текст шаблона должен быть в переменной tmpl
var compiled = Handlebars.compile(tmpl);

var result = compiled({
  title: "Сладости",
  items: ["Торт", "Пирожное", "Пончик"]
});
```

Библиотека шаблонизации Handlebars «понимает» этот язык. Вызов `Handlebars.compile` принимает строку шаблона, разбивает по разделителям и, аналогично предыдущему виду шаблонов, делает JavaScript-функцию, которая затем по данным выдаёт строку-результат.

Запрет на встроенный JS

Если «свой язык шаблонизатора» очень прост, то библиотеку для его поддержки можно легко написать под PHP, Ruby, Java и других языках, которые тем самым научатся понимать такие шаблоны.

Если шаблонка действительна нацелена на кросс-платформенность, то явные JS-вызовы в ней запрещены. Всё делается через хелперы.

Если же нужна какая-то логика, то она либо выносится во внешний код, либо делается через новый хелпер – он отдельно пишется на JavaScript (для клиента) и для сервера (на его языке). Получается полная совместимость.

Это создаёт определённые сложности. Например, в Handlebars есть хелпер `{{#if cond}} ... {{/if}}`, который выводит содержимое, если истинно условие `cond`. При этом вместо `cond` нельзя поставить, к примеру, `a > b` или вызов `str.toUpperCase()`, будет ошибка. Все вычисления должны быть сделаны на этапе передачи данных в шаблон.

Так сделано как раз для переносимости шаблонной системы на другие языки, но на практике не очень-то удобно.

Продвинутые кросс-платформенные шаблонизаторы, в частности, Closure Templates, обладают более мощным языком и умеют самостоятельно разбирать и компилировать многие выражения.

Шаблонизация компонент

До этого мы говорили о шаблонных системах «общего назначения». По большому счёту, это всего лишь механизмы для преобразования одной строки в другую. Но при описании шаблона для компоненты мы хотим сгенерировать не просто строку, а DOM-элемент, и не просто генерировать, а в дальнейшем – с ним работать.

Современные шаблонные системы «заточены» на это. Они умеют создавать по шаблону DOM-элементы и автоматически выполнять после этого разные полезные действия.

Например:

- Можно сохранить важные подэлементы в свойства компоненты, чтобы было проще к ним обращаться из JavaScript.
- Можно автоматически назначать обработчики из методов компонента.
- Можно запомнить, какие данные относятся к каким элементам и в дальнейшем, при изменении данных автоматически обновлять DOM («привязка данных» – англ. data binding).

Одной из первых систем шаблонизации, которая поддерживает подобные возможности была Knockout.JS.

Попробуйте поменять значение `<input>` в примере ниже и вы увидите двухстороннюю привязку данных в действии:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.3.0/knockout-m...

Поменяйте имя: <input data-bind="value: name, valueUpdate: 'input'">
<hr>
<h1>Привет, <span data-bind="text: name"></span>!</h1>

<script>
var user = {
  name: ko.observable("Вася")
};

ko.applyBindings(user, document.body);
</script>
```

Поменяйте имя: Вася

Привет, Вася!

Библиотека Knockout.JS создаёт объект `ko`, который и содержит все её возможности.

В этом примере работу начинает вызов `ko.applyBindings(user, document.body)`.

Его аргументы:

- `user` – объект с данными.
- `document.body` – DOM-элемент, который будет использован в качестве шаблона.

Он пробегает по всем подэлементам `document.body` и, если видит атрибут `data-bind`, то читает его и выполняет привязку данных.

Значение `<input data-bind="value: name, ...">` означает, что нужно привязать `input.value` к свойству `name` объекта данных.

Привязка осуществляется в две стороны:

1. Во-первых, библиотека ставит на `input` свой обработчик `oninput` (можно выбрать другие события, см. документацию), который будет обновлять `user.name`. То есть, изменение `input` автоматически меняет `user.name`
2. Во-вторых, свойство `user.name` создано как `ko.observable(...)`. Технически, `ko.observable(value)` – это функция-обёртка вокруг значения: геттер-сеттер, который умеет рассылать события при изменении.

   Например:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/knockout/3.3.0/knockout

<script>
var user = ko.observable("Вася");

// вызов user() возвращает значение
alert( user() ); // Вася

// вызов user.subscribe(func) ставит обработчик на изменение значения
user.subscribe(function(newValue) {
  alert("Новое значение: " + newValue);
});

```

```
// вызов user(newValue) меняет значение
user("Петя"); // сработает обработчик, назначенный выше
</script>
```

Библиотека Knockout.JS ставит свой обработчик на изменение значения и при этом обновляет все привязки. Так что при изменении `user.name` меняется и `input.value`.

Далее в том же примере находится `<span data-bind="text: name">` – здесь атрибут означает привязку текста к `name`. Так как `<span>` по своей инициативе меняться не может, то привязка односторонняя, но если бы мог, то можно сделать и двухстороннюю, это несложно.

Вызов `ko.applyBindings` можно делать внутри компоненты, и таким образом устанавливать соответствия между её объектом и DOM.

Библиотека также поддерживает хранение шаблонов в `<script type="text/template">` – см. документацию template-binding, можно организовать прекомпиляцию, добавлять свои привязки и так далее.

Другие библиотеки

Есть другие библиотеки «продвинутой шаблонизации», которые добавляют свои возможности по работе с DOM, например:

- Ractive.JS
- Rivets.JS

Подобная шаблонная система является частью многих фреймворков, например:

- React.JS
- Angular.JS
- Ember.JS

Все эти фреймворки разные:

- Ember использует надстройку над Handlebars.
- React использует JSX (JavaScript XML syntax transform) – свой особый способ вставки разметки в JS-код, который нужно обязательно прекомпилировать перед запуском.
- Angular вместо работы со строками использует клонирование DOM-узлов.

При разработке современного веб-приложения имеет смысл выбрать продвинутую шаблонную систему или даже один из этих архитектурных фреймворков.

Итого

Системы шаблонизации, в порядке развития и усложнения:

- Микрошаблонизация – строка с JS-вставками, которая компилируется в функцию – самый простой вариант, минимальная работа для шаблонизатора.
- Собственный язык шаблонов – «особо простой» синтаксис для частых операций, с запретом на JS в случае, если нужна кросс-платформенность.
- Шаблонизация для компонентов – современные системы, которые умеют не только генерировать DOM, но и помогать с дальнейшей работой с ним.

Для того, чтобы использовать одни и те же шаблоны на клиенте и сервере, применяют либо кросс-платформенную систему шаблонизации, либо, чаще – интегрируют серверную часть с V8 и, возможно, с сервером Node.JS.

В главе было много ссылок на шаблонные системы. Все они достаточно современные, поддерживаемые и используются во многих проектах. Среди них вы наверняка найдёте нужную вам.

Комментарии

 |