РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

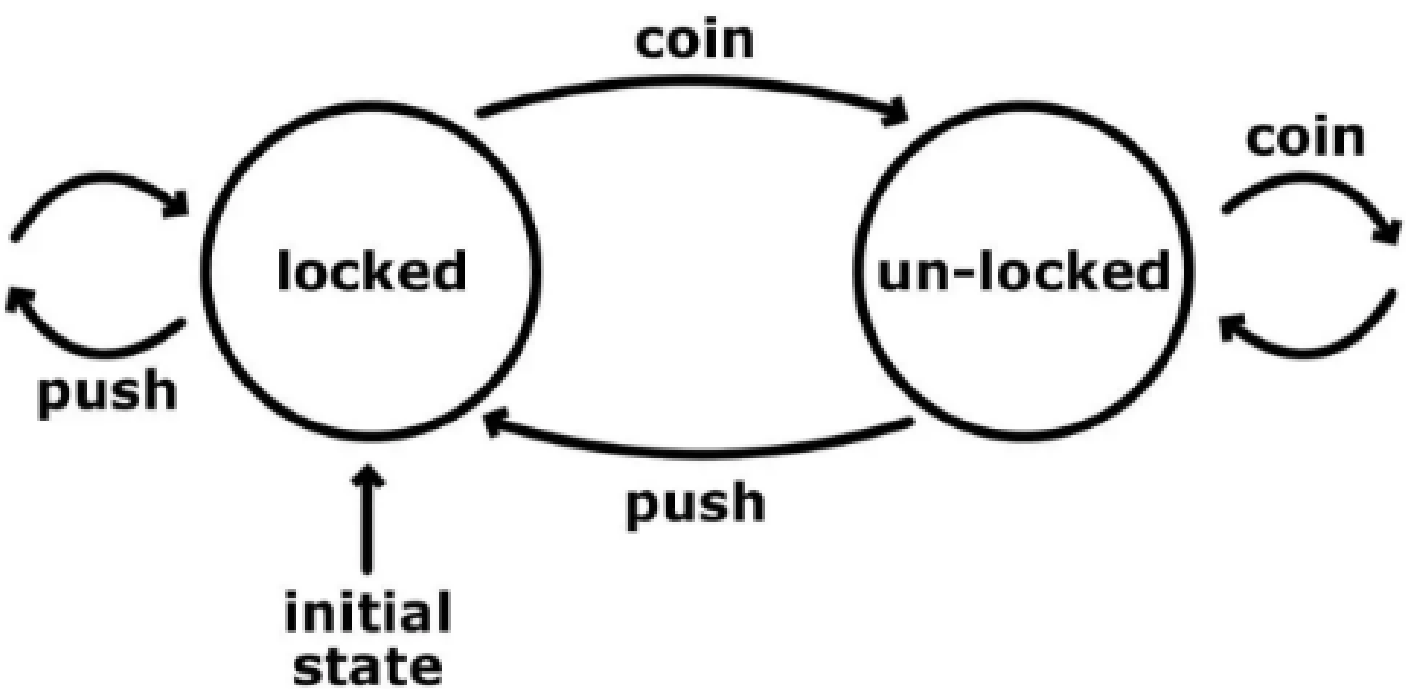
# Машины состояний и разработка веб-приложений

ПЕРЕВОД

ru_vds 18 января в 12:02 11k

Оригинал: Krasimir Tsonev

Настал 2018-й год, найдено множество замечательных способов создания приложений, но бесчисленные армии фронтенд-разработчиков всё ещё ведут борьбу за простоту и гибкость веб-проектов. Месяц за месяцем они проводят в попытках достигнуть заветной цели: найти программную архитектуру, свободную от ошибок, и помогающую им делать их работу быстро и качественно. Я — один из этих разработчиков. Мне удалось найти кое-что интересное, способное дать нам шанс на победу.



Инструменты вроде React и Redux позволили веб-разработке сделать большой шаг в правильном направлении. Однако, самих по себе их недостаточно для создания крупномасштабных приложений. Похоже, что ситуацию в разработке клиентских частей веб-приложений может значительно улучшить применение машин состояний. О них и пойдёт речь в этом материале. Кстати, возможно вы уже построили несколько таких машин, но пока ещё об этом не знаете.

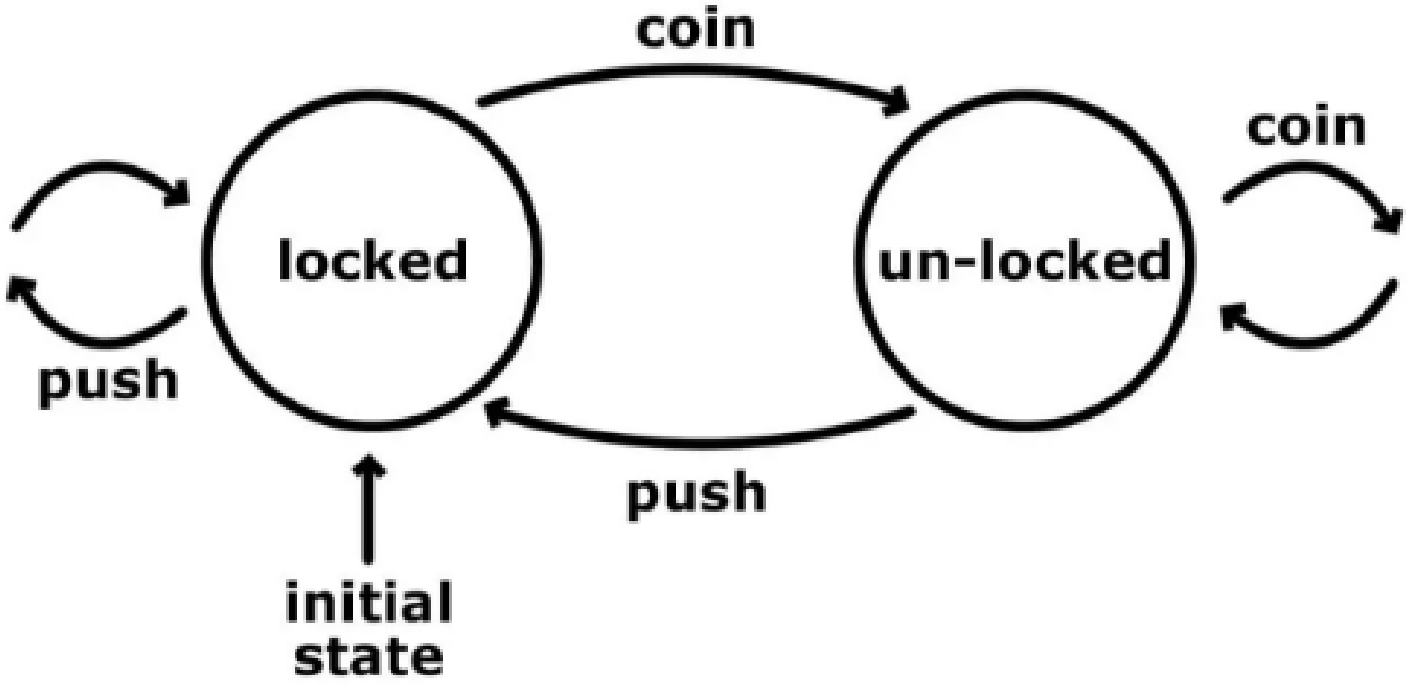
## Знакомство с машинами состояний

Машина состояний — это математическая модель вычислений. Это — абстрактная концепция, в соответствии с которой машина может иметь различные состояния, но, в некий момент времени, пребывать лишь в одном из них. Полагаю, самая известная машина состояний — это машина Тьюринга. Это — машина с неограниченным числом состояний, что означает, что состояний у неё может быть бесконечное количество. Машина Тьюринга не очень хорошо соответствует нуждам современной разработки интерфейсов, так как в большинстве случаев у нас имеется конечное число состояний. Именно поэтому нам лучше подходят машины с ограниченным числом состояний, или, как их часто называют, конечные автоматы. Это — автомат Мили и автомат Мура.

Разница между ними заключается в том, что автомат Мура меняет состояние, основываясь только на своём предыдущем состоянии. Однако, у нас имеется множество внешних факторов, таких, как

действия пользователя и процессы, происходящие в сети, что означает, что и автомат Мура нам не подойдёт. То, что мы ищем, очень похоже на автомат Мили. У этого конечного автомата имеется начальное состояние, после чего он переходит в новые состояния, основываясь на входных данных и на его текущем состоянии.

Один из самых простых способов проиллюстрировать работу машины состояний заключается в рассмотрении её через аналогию с турникетом. У него имеется ограниченный набор состояний: он может быть либо закрыт, либо открыт. Наш турникет преграждает вход в некую область. Пусть у него будет барабан с тремя планками и механизм для приёма денег. Пройти через закрытый турникет можно, опустив монету в монетоприёмник, что переводит устройство в открытое состояние, и толкнув планку для того, чтобы пройти через турникет. После того, как через турникет прошли, он снова закрывается. Вот простая схема, которая показывает нам эти состояния, а также возможные входные сигналы и переходы между состояниями.




Исходное состояние турникета — «закрыто» (locked). Неважно, сколько раз мы толкнём его планку, он останется закрытым. Однако, если мы опустим в него монетку, турникет перейдёт в состояние «открыто» (un-locked). В этот момент ещё одна монетка ничего не изменит, так как турникет всё ещё будет в открытом состоянии. С другой стороны, теперь толчок планки турникета имеет смысл, и мы сможем через него пройти. Это действие, кроме того, переведёт наш конечный автомат в исходное состояние «закрыто».

Если нужно реализовать единственную функцию, которая контролирует турникет, нам, вероятно, стоит остановиться на двух аргументах: это — текущее состояние и действие. Если вы используете Redux, возможно, вам это знакомо. Это похоже на широко известные функции-редьюсеры, где мы получаем текущее состояние, и, основываясь на полезной нагрузке действия, решаем, каким будет следующее состояние. Редьюсер — это переход в контексте машины состояний. На самом деле, любое

приложение, имеющее состояние, которое мы можем как-то менять, может называться машиной состояний. Дело просто в том, что всё это, снова и снова, реализуется вручную.

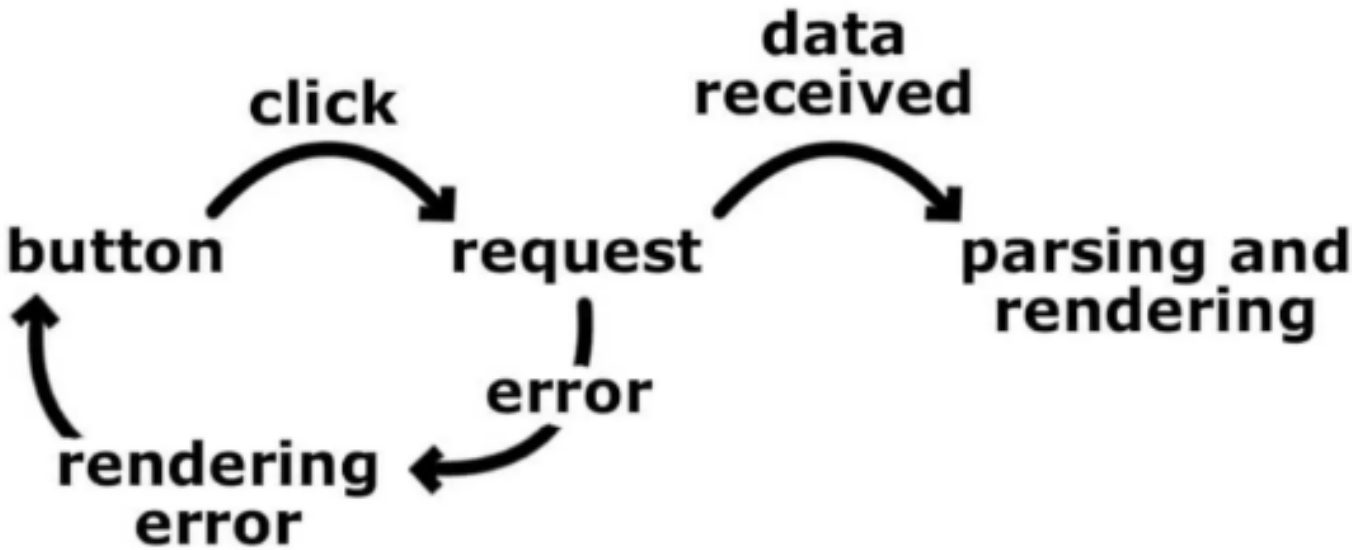
## Каковы сильные стороны машин состояний?

На работе мы используем Redux и он нас вполне устраивает. Однако я начал замечать кое-какие вещи, которые мне не нравятся. То, что мне что-то «не нравятся», не значит, что это не работает. Речь больше идёт о том, что всё это добавляет проекту сложности и принуждает меня писать больше кода. Как-то я занялся сторонним проектом, где у меня был простор для экспериментов, и я решил переосмыслить наши подходы к разработке на React и Redux. Я начал делать заметки о том, что меня беспокоит, и понял, что абстракция машины состояний, вероятнее всего, сможет решить некоторые из этих проблем. Примемся за дело и посмотрим, как реализовать машину состояний на JavaScript.

Мы будем решать простую задачу. Нам нужно получать данные из серверного API и показывать их пользователю. Самый первый шаг заключается в том, что понять, как, при осмыслении задачи, думать в терминах состояний, а не переходов. Прежде чем мы перейдём к машине состояний, хочу рассказать о том, как, на высоком уровне, выглядит то, что мы хотим создать:

- На страницу выводится кнопка `Загрузить данные`.
- Пользователь щёлкает по этой кнопке.
- Система выполняет запрос к серверу.

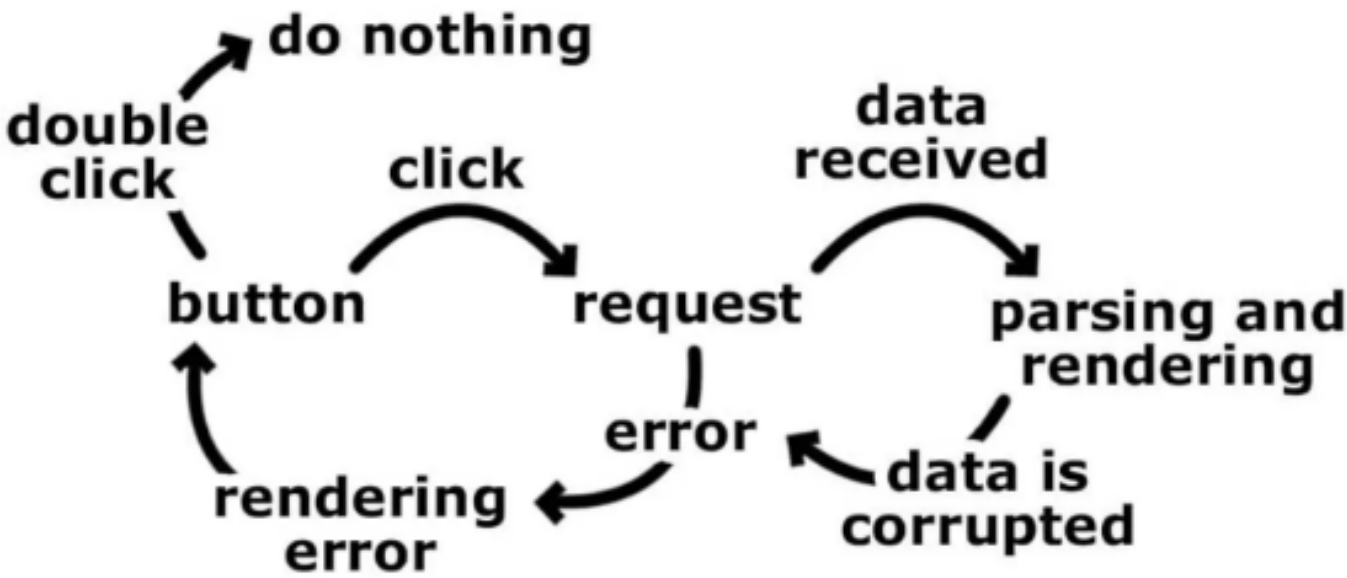
- Производится загрузка и разбор данных.
- Данные выводятся на страницу.
- Если произошла ошибка, выводится соответствующее сообщение и на странице снова появляется кнопка `Загрузить данные`, что даёт пользователю возможность снова запустить процесс получения данных с сервера.




Сейчас, анализируя задачу, мы мыслим линейно, и, собственно говоря, пытаемся охватить все возможные пути к итоговому результату. Один шаг ведёт к другому, следующий ведёт ещё к одному, и так далее. В коде это может выражаться в виде операторов ветвления. Проведём мысленный эксперимент с программой, которая построена на основе действий пользователя и системы.

Как насчёт ситуации, в которой пользователь щёлкнет кнопку два раза? Что произойдёт, если пользователь щёлкнет по кнопке, пока мы ожидаем ответа от сервера? Как поведёт себя система, если запрос выполнен успешно, но данные оказались повреждёнными?

Для обработки подобных ситуаций нам, вероятно, понадобятся различные флаги, которые сигнализируют о происходящем. Наличие флагов означает увеличение числа операторов `if`, и, в более сложных приложениях, рост числа конфликтов.

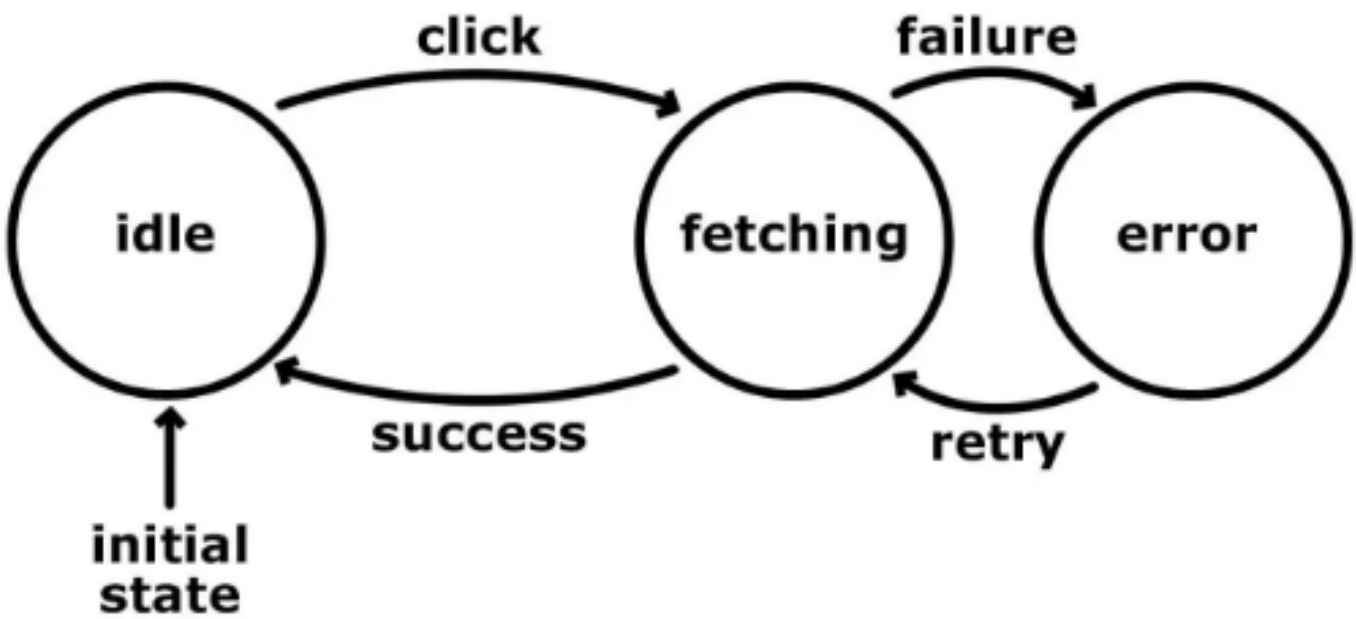



Происходит так из-за того, что мы мыслим переходами. Мы сосредоточены на том, как именно происходят изменения в программе, и на том, в каком порядке они происходят. Если, вместо этого, сосредоточиться на различных состояниях приложения, всё значительно упростится. Сколько состояний у нас есть? Каковы их входные данные? Используем тот же пример:

- Состояние `idle` (простой). В этом состоянии мы выводим кнопку `Загрузить данные` и ждём действий пользователя. Вот возможные действия:
- Состояние `fetching` (получение данных). Запрос отправлен на сервер и мы ожидаем его завершения. Вот возможные действия:

- Состояние `error` (ошибка). Мы выводим сообщение об ошибке и показываем кнопку `Загрузить данные`. Это состояние принимает одно действие:

Тут мы описали примерно тот же самый процесс, но теперь уже пользуясь состояниями и входными данными.




Это упростило логику и сделало её более предсказуемой. Кроме того, это решило некоторые из упомянутых выше проблем. Обратите внимание на то, что когда машина пребывает в состоянии `fetching`, мы не принимаем событий, связанных со щелчком мышью по кнопке. Поэтому даже если пользователь щёлкнет по кнопке, ничего не произойдёт, так как машина не настроена на реакцию на данное действие, когда она находится в состоянии `fetching`. Этот подход позволяет автоматически устранить непредвиденное ветвление логики кода.

Это означает, что нам придётся покрыть меньше кода при

тестирования. Кроме того, некоторые типы тестирования, такие, как интеграционное тестирование, могут быть автоматизированы. Подумайте о том, что при таком подходе у нас было бы по-настоящему чёткое понимание того, что делает наше приложение, и мы могли бы создать скрипт, который проходится по предопределённым состояниями и переходам и генерирует утверждения. Эти утверждения могут доказать, что мы достигли каждого из возможных состояний или осуществили конкретную последовательность переходов.

На самом деле, выписать все возможные состояния легче, чем выписать все возможные переходы, так как нам известно, какие состояния нам нужны, или какие состояния у нас есть. Между прочим, в большинстве случаев, состояния описывали бы логику функционирования нашего приложения. А вот если говорить о переходах, их смысл очень часто в начале работы неизвестен. Ошибки в программах являются результатам того, что действия выполнены тогда, когда приложение находится в состоянии, не рассчитанном на эти действия. Кроме того, даже если приложение находится в подходящем состоянии, действие может быть выполнено в неподходящее время. Подобные действия приводят наше приложение в состояние, о котором мы не знаем, и это выводит программу из строя или приводит к тому, что она ведёт себя неправильно. Конечно, нам это ни к чему. Машины состояний — это хорошие средства защиты от подобных проблем. Они защищают нас от достижения неизвестных состояний, так как мы устанавливаем границы для того, что и когда может случиться, не указывая явно — как это может случиться. Концепция машины состояний отлично сочетается с однонаправленным потоком

данных. Вместе они уменьшают сложность кода и дают чёткие ответы на вопросы о том, как система попала в то или иное состояние.

## Создание машины состояний средствами JavaScript

Хватит разговоров — пришло время программировать. Использовать будем тот же самый пример. Основываясь на вышеприведённом списке, начнём со следующего кода:

```
const machine = {
  'idle': {
    click: function () { ... }
  },
  'fetching': {
    success: function () { ... },
    failure: function () { ... }
  },
  'error': {
    'retry': function () { ... }
  }
}
```

Состояния представлены объектами, возможные входные сигналы состояний — методами объектов. Однако исходного состояния тут нет. Изменим вышеприведённый код, приведя его к следующему виду:

```
const machine = {
  state: 'idle',
  transitions: {
    'idle': {
      click: function() { ... }
    },
    'fetching': {
      success: function() { ... },
```

```
      failure: function() { ... }
    },
    'error': {
      'retry': function() { ... }
    }
  }
}
```

После того, как мы определили все состояния, имеющие смысл, мы готовы отправлять им входные сигналы и менять состояние системы. Делать это будем, используя два следующих вспомогательных метода:

```
const machine = {
  dispatch(actionName, ...payload) {
    const actions = this.transitions[this.state];
    const action = this.transitions[this.state][actionName];

    if (action) {
      action.apply(machine, ...payload);
    }
  },
  changeStateTo(newState) {
    this.state = newState;
  },
  ...
}
```

Функция `dispatch` проверяет, имеется ли действие с заданным именем среди переходов текущего состояния. Если это так, она вызывает это действие, передавая ему переданные ей при вызове данные. Кроме того, обработчик `action` вызывается с `machine` в качестве контекста, поэтому мы можем диспетчеризировать другое действие с помощью `this.dispatch(<action>)` или изменить состояние с помощью `this.changeStateTo(<new state>)`.

Следуя пути пользователя из нашего примера, первое действие, которое нам надо диспетчеризировать — это `click`. Вот как выглядит обработчик этого действия:

```
transitions: {
  'idle': {
    click: function () {
      this.changeStateTo('fetching');
      service.getData().then(
        data => {
          try {
            this.dispatch('success', JSON.parse(data));
          } catch (error) {
            this.dispatch('failure', error)
          }
        },
        error => this.dispatch('failure', error)
      );
    }
  },
  ...
}

machine.dispatch('click');
```

Сначала мы изменяем состояние машины на `fetching`. Затем выполним запрос к серверу. Предположим, что у нас имеется служба с методом `getData`, который возвращает промис. После того, как этот промис будет разрешён и данные будут успешно разобраны, мы диспетчеризируем событие `succes`, иначе — `failure`.

Пока всё идёт как надо. Далее нам нужно реализовать действия `success` и `failure` и описать входные данные состояния `fetching`:

```
transitions: {
  'idle': { ... },
  'fetching': {
    success: function (data) {
      // вывод данных
      this.changeStateTo('idle');
    },
    failure: function (error) {
      this.changeStateTo('error');
    }
  },
  ...
}
```

Обратите внимание на то, как мы избавили себя от необходимости думать о предыдущем процессе. Мы не заботимся о щелчках пользователя по кнопке, или о том, что происходит с HTTP-запросом. Мы знаем, что приложение находится в состоянии `fetching`, и мы ожидаем появления лишь этих двух действий. Это немного похоже на создание новых механизмов приложения, которые работают в изоляции.

Последнее, с чем нам осталось разобраться — это состояние `error`. Будет очень хорошо, если мы создадим тут код для реализации повторной попытки, в результате приложение сможет восстановиться после возникновения ошибки.

```
transitions: {
  'error': {
    retry: function () {
      this.changeStateTo('idle');
      this.dispatch('click');
    }
  }
}
```

Здесь придётся копировать тот код, который уже написан в обработчике `click`. Для того чтобы этого избежать, нам либо нужно объявить обработчик как функцию, доступную обоим действиям, либо сначала перейти в состояние `idle`, а затем диспетчеризировать действие `click` самостоятельно.

Полный пример работающей машины состояний можно найти в моём CodePen-проекте.

## Управление машинами состояний с помощью библиотеки

Шаблон конечного автомата работает вне зависимости от того, используем ли мы React, Vue или Angular. Как мы видели в предыдущем разделе, реализовать машину состояний на чистом JS можно без особых сложностей. Однако, если доверить это специализированной библиотеке, это способно добавить проекту больше гибкости. Среди примеров хороших библиотек для реализации машин состояний можно отметить Machina.js и XState. В этой статье, однако, мы поговорим о Stent — моей Redux-подобной библиотеке, в которой реализована концепция конечных автоматов.

Stent — это реализация контейнера машин состояний. Эта библиотека следует некоторым идеям проектов Redux и Redux-Saga, но даёт, по моему мнению, более простые в использовании и менее скованные шаблонами возможности. Она разработана с использованием подхода, основанного на том, что сначала пишут

документацию к проекту, а потом — код. Следуя этому подходу, я потратил недели только на проектирование API. Так как я самостоятельно писал библиотеку, у меня был шанс исправить проблемы, с которыми я столкнулся, используя архитектуры Redux и Flux.

## Создание машин состояний в Stent

В большинстве случаев приложение выполняет множество функций. В результате лишь одной машиной нам не обойтись. Поэтому Stent позволяет создавать столько машин, сколько нужно:

```
import { Machine } from 'stent';

const machineA = Machine.create('A', {
  state: ...,
  transitions: ...
});
const machineB = Machine.create('B', {
  state: ...,
  transitions: ...
});
```

Позже мы можем получить доступ к этим машинам, используя метод `Machine.get`:

```
const machineA = Machine.get('A');
const machineB = Machine.get('B');
```

## Подключение машин к логике рендеринга

Рендеринг в моём случае выполняется средствами React, но мы

можем использовать любую другую библиотеку. Всё это сводится к вызову коллбэка, в котором мы инициируем рендеринг. Одной из первых возможностей библиотеки, которую я создал, была функция `connect`:

```
import { connect } from 'stent/lib/helpers';

Machine.create('MachineA', ...);
Machine.create('MachineB', ...);

connect()
  .with('MachineA', 'MachineB')
  .map((MachineA, MachineB) => {
    ... функцию рендеринга будем вызывать здесь
  });
```

Мы сообщаем системе о том, с какими машинами хотим работать, указывая их имена. Коллбэк, который мы передаём методу `map`, сразу же вызывается, делается это один раз. После этого он вызывается каждый раз, когда состояние какой-то из машин меняется. Именно здесь мы вызываем функцию рендеринга. В этом месте у нас есть прямой доступ к подключённым машинам, поэтому мы можем получить текущее состояние машин и их методы. В библиотеке, кроме того, имеется метод `mapOnce`, применяемый для работы с коллбэками, которые нужно вызывать лишь один раз, и `mapSilent` — для того, чтобы пропустить это первоначальное однократное выполнение коллбэка.

Для удобства вспомогательные функции экспортированы в расчёте на интеграцию с React. Это очень сильно похоже на конструкцию connect(mapStateToProps) Redux.

```
import React from 'react';
import { connect } from 'stent/lib/react';

class TodoList extends React.Component {
  render() {
    const { isIdle, todos } = this.props;
    ...
  }
}

// MachineA и MachineB - это машины, определённые
// с помощью функции Machine.create
export default connect(TodoList)
  .with('MachineA', 'MachineB')
  .map((MachineA, MachineB) => {
    isIdle: MachineA.isIdle,
    todos: MachineB.state.todos
  });
```

Stent выполняет коллбэк и ожидает получить объект. А именно — объект, который отправлен как `props` компоненту React.

## Что такое состояние в контексте Stent?

До сих пор состояния были простыми строками. К сожалению, в реальном мире приходится хранить в состоянии нечто большее, чем обычная строка. Именно поэтому состояние Stent — это объект, внутри которого имеются свойства. Единственным зарезервированным свойством является `name`. Всё остальное — это данные, специфичные для приложения. Например:

```
{ name: 'idle' }
{ name: 'fetching', todos: [] }
{ name: 'forward', speed: 120, gear: 4 }
```

Мой опыт работы со Stent показывает, что если объект состояния

становится слишком большим, то нам, возможно, нужна ещё одна машина состояний, которая могла бы обрабатывать эти дополнительные свойства. Идентификация различных состояний занимает некоторое время, но я полагаю, что это — большой шаг вперёд в деле написания приложений, которыми легче управлять. Это нечто вроде попытки заблаговременного планирования поведения системы и подготовки пространства для будущих действий.

## Работа с машиной состояний

Практически так же, как в примере, приведённом в начале материала, нам, при работе со Stent, нужно задать возможные (конечные) состояния машины и описать возможные входные сигналы:

```
import { Machine } from 'stent';

const machine = Machine.create('sprinter', {
  state: { name: 'idle' }, // начальное состояние
  transitions: {
    'idle': {
      'run please': function () {
        return { name: 'running' };
      }
    },
    'running': {
      'stop now': function () {
        return { name: 'idle' };
      }
    }
  }
});
```

Тут есть начальное состояние, `idle`, которое принимает действие

`run`. После того, как машина оказывается в состоянии `running`, можно запустить действие `stop`, которое переведёт машину обратно в состояние `idle`.

Вы, возможно, помните вспомогательные функции `dispatch` и `changeStateTo` из примера, приведённого выше. Stent предоставляет те же возможности, но они скрыты внутри библиотеки, в результате, нам не приходится заниматься ими самостоятельно. Для удобства, основываясь на свойстве `transitions`, Stent генерирует следующее:

- Вспомогательные методы для проверки того, пребывает ли машина в некоем состоянии. Так, наличие состояния `idle` приводит к созданию метода `isIdle()`, а наличие состояния `running` — к созданию метода `isRunning()`.
- Вспомогательные методы для диспетчеризации событий: `runPlease()` и `stopNow()`.

В результате в нашем примере доступны следующие конструкции:

```
machine.isIdle(); // логическое значение
machine.isRunning(); // логическое значение
machine.runPlease(); // запускает действие
machine.stopNow(); // запускает действие
```

Комбинируя автоматически сгенерированные методы с сервисной функцией `connect`, мы можем выйти на готовое решение. Пользователь влияет на входные данные машины и на действия, которые ведут к изменению состояния. Из-за изменения состояния

вызывается функция маппинга, переданная `connect`, и мы получаем уведомление об изменении состояния. Затем производится вывод данных на экран.

## Входные данные и обработчики действий

Возможно, самое важное в этом примере — обработчики действий. Это — место, где мы пишем большую часть логики приложения, так как здесь описывается реакция системы на входные данные и на изменённые состояния. Порой мне кажется, что самые удачные архитектурные решения, принятые при проектировании Redux — это иммутабельность и простота функций-редьюсеров. В сущности, обработчики действий Stent — это то же самое. Обработчик получает текущее состояние и данные, связанные с действием, после чего он должен вернуть новое состояние. Если обработчик не возвращает ничего (`undefined`), тогда состояние машины остаётся неизменным.

```
transitions: {
  'fetching': {
    'success': function (state, payload) {
      const todos = [ ...state.todos, payload ];

      return { name: 'idle', todos };
    }
  }
}
```

Предположим, что требуется загрузить данные с удалённого сервера. Для этого мы выполняем запрос и переводим машину в состояние `fetching`. Как только данные приходят с сервера,

вызывается событие `success`:

```
machine.success({ label: '...' });
```

Затем мы возвращаемся в состояние `idle` и сохраняем некие данные в виде массива `todos`. Здесь есть пара вариантов, касающихся настройки обработчиков действий. Первый и самый простой случай — это работа с простой строкой, которая становится новым состоянием.

```
transitions: {
  'idle': {
    'run': 'running'
  }
}
```

Здесь показан переход из состояния `{ name: 'idle' }` в состояние `{ name: 'running' }` с использованием действия `run()`. Этот подход полезен, когда используются синхронные переходы между состояниями, и при этом у состояний нет дополнительных данных. Поэтому, если в объекте состояния хранят ещё что-нибудь, подобный переход между состояниями уничтожит такие дополнительные данные. Похожим образом можно передать объект состояния напрямую:

```
transitions: {
  'editing': {
    'delete all todos': { name: 'idle', todos: [] }
  }
}
```

Здесь можно наблюдать переход из состояния `editing` в `idle` с использованием действия `deleteAllTodos`.

Мы уже видели функцию-обработчик, и последний вариант обработчика действия — это функция-генератор. Идейным вдохновителем этого механизма стал проект Redux-Saga, выглядит это так:

```
import { call } from 'stent/lib/helpers';

Machine.create('app', {
  'idle': {
    'fetch data': function * (state, payload) {
      yield { name: 'fetching' }

      try {
        const data = yield call(requestToBackend, '/api/todos/',
'POST');

        return { name: 'idle', data };
      } catch (error) {
        return { name: 'error', error };
      }
    }
  }
});
```

Если у вас нет опыта работы с генераторами, вышеприведённый фрагмент кода может выглядеть немного таинственно. Однако, генераторы в JavaScript — это мощный инструмент. С их помощью можно приостановить обработчик действия, изменять состояние несколько раз и обрабатывать асинхронные механизмы.

## Эксперименты с генераторами

Когда я впервые познакомился с Redux-Saga, я решил, что там представлен чрезмерно усложнённый способ поддержки асинхронных операций. На самом же деле это — весьма остроумная реализация шаблона command (команда). Главное преимущество этого шаблона заключается в том, что он разделяет вызов некоего механизма и его реальную реализацию.

Другими словами — мы сообщаем системе о том, чего мы хотим, но не говорим о том, как это должно произойти. В этом всём мне помогла разобраться серия материалов Мэтта Хикса, рекомендую с ней ознакомиться. Те же самые идеи я принёс и в Stent. Мы передаём системе управление, сообщая ей о том, что нам нужно, но на самом деле этого не делая. Как только действие выполняется, мы получаем управление обратно.

Вот что можно передавать системе в настоящий момент:

- Объект состояния (или строку) для изменения состояния машины.
- Вспомогательную функцию `call` (она принимает синхронную функцию, которая является функцией, возвращающей промис или другую функцию-генератор). Передавая такую функцию, мы, по существу, сообщаем системе: «Запусти эту функцию, и если она является асинхронной, подожди. Как только дело будет сделано — передай нам результат».
- Вспомогательную функции `wait` (она принимает строку, представляющую другое действие). При использовании этой

сервисной функции мы приостанавливаем обработчик и ждём диспетчеризации другого действия.

Вот функция, которая иллюстрирует рассмотренные выше примеры:

```
const fireHTTPRequest = function () {
  return new Promise((resolve, reject) => {
    // ...
  });
}

...
transitions: {
  'idle': {
    'fetch data': function * () {
      yield 'fetching'; // устанавливает состояние в { name:
'fetching' }
      yield { name: 'fetching' }; // то же самое, что и выше

      // ожидание диспетчеризации
      // действий getTheData и checkForErrors
      const [ data, isError ] = yield wait('get the data', 'check
for errors');

      // ожидание разрешения промиса,
      // возвращённого fireHTTPRequest
      const result = yield call(fireHTTPRequest, '/api/data/users');

      return { name: 'finish', users: result };
    }
  }
}
```

Этот код выглядит как синхронный, но, на самом деле, таковым он не является. Здесь мы видим, как Stent выполняет рутинные операции по ожиданию разрешения промиса и по работе с другим генератором.

## Проблемы Redux и их решение с помощью Stent

### Избавление от чрезмерного количества шаблонного кода

Архитектура Redux (как и Flux) основана на действиях, которые циркулируют в системе. По мере роста приложения, как правило, в нём появляется множество констант и создателей действий. Эти сущности обычно оказываются в разных папках, в результате анализ кода при его выполнении иногда требует дополнительного времени. Кроме того, при добавлении в приложение новой возможности, всегда приходится иметь дело с полным набором действий, что означает определение большего количества имён действий и создателей действий.

В Stent нет имён действий. Библиотека генерирует создателей действий автоматически:

```
const machine = Machine.create('todo-app', {
  state: { name: 'idle', todos: [] },
  transitions: {
    'idle': {
      'add todo': function (state, todo) {
        ...
      }
    }
  }
});

machine.addTodo({ title: 'Fix that bug' });
```

Тут есть создатель действия `machine.addTodo`, определённый как метод машины. Этот подход также решает другую проблему, с

которой я столкнулся: поиск редьюсера, который реагирует на конкретное действие. Обычно в компонентах React можно видеть имена создателей действий как нечто вроде `addToDo`. Однако в редьюсерах мы работаем с типом действия, который является константой. Иногда мне приходится переходить к коду создателя действия потому что только так я могу увидеть точный тип. Здесь вообще нет типов.

## Устранение непредсказуемых изменений состояния

В целом, Redux замечательно реализует иммутабельный подход к управлению состоянием приложения. Проблема заключается не в самом Redux, а в том, что разработчику позволено диспетчеризировать любое действие в любое время. Если предположить, что у нас есть действие, которое включает свет, нормально ли будет выполнить это действие два раза подряд? Если нет — тогда как решить эту проблему средствами Redux? Например, мы, возможно, поместим какой-то код в редьюсер. Этот код будет защищать логику приложения, проверяя при попытке включения света, не включен ли он ранее. Вероятно, это будет что-то вроде условного оператора, который проверяет текущее состояние.

Теперь вопрос заключается в том, не находится ли это действие за пределами сферы ответственности редьюсера? Следует ли редьюсеру знать о подобных пограничных случаях?

Чего мне не хватает в Redux, так этот способа остановки диспетчеризации действия, основанного на текущем состоянии

приложения без загрязнения редьюсера условной логикой. И мне совершенно не хочется принимать подобное решение в том слое, где был вызван создатель действия. С помощью Stent фильтрация бессмысленных действий происходит автоматически, так как машина не реагирует на действия, которые не объявлены в текущем состоянии. Например:

```
const machine = Machine.create('app', {
  state: { name: 'idle' },
  transitions: {
    'idle': {
      'run': 'running',
      'jump': 'jumping'
    },
    'running': {
      'stop': 'idle'
    }
  }
});

// это сработает
machine.run();
// А здесь не произойдёт ничего,
// так как машина находится в состоянии
// running и в нём имеется лишь действие stop.
machine.jump();
```

Факт того, что машина принимает лишь специфические входные данные в конкретное время защищает нас от странных ошибок и делает наше приложение более предсказуемым.

## Состояния как основа архитектуры приложений

Redux, как и Flux, заставляют нас мыслить в терминах переходов. Модель мышления, характерная для Redux-разработки, в основном, базируется на действиях, и на том, как эти действия

трансформируют состояние в редьюсерах. Это не так уж и плохо, но я обнаружил, что гораздо выгоднее, во всех смыслах, оперировать не переходами между состояниями, а самими состояниями. При таком подходе разработчика будут занимать вопросы о том, какими могут быть состояния приложения, и о том, как эти состояния представляют требования логики функционирования проекта.

## Итоги

Концепция машин состояний в программировании, особенно — в разработке пользовательских интерфейсов, стала для меня настоящим открытием. Я начал видеть машины состояний повсюду, у меня появилось желание везде, где это возможно, ими пользоваться. Я чётко вижу преимущества наличия в проекте строго определённых состояний и переходов между ними. Мне, в ходе работы, всегда хочется сделать мои приложения как можно более простыми, а их код — как можно более понятным. Я уверен, что машины состояний — это шаг в правильном направлении. Это — простая, и в то же время — мощная концепция. Её практическое использование вполне способно помочь сделать веб-приложения стабильнее и устранить множество ошибок, характерных для других подходов к разработке.

**Уважаемые читатели!** Пользуетесь ли вы машинами состояний при работе над своими проектами?

Проголосовать:

+15

Поделиться:

Сохранить:

Комментарии (8)

## Похожие публикации

### Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

### JavaScript и ужасы мутаций

48

ПЕРЕВОД

ru_vds • 19 января в 11:42

### Возможности JavaScript, о существовании которых я не знал

149

ПЕРЕВОД

ru_vds • 12 января в 12:14

## Популярное за сутки

### Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

### Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

### Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

### Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

### Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

## Лучшее на Geektimes

## Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни

33

HostingManager • вчера в 13:49

## Обзор рынка моноколес 2018

70

lozga • вчера в 06:58

## «Битва за Telegram»: 35 пользователей подали в суд на ФСБ

40

alizar • вчера в 15:14

## Стивен Хокинг и его работа — что дал ученый человечеству?

8

marks • вчера в 14:46

## Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32