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

Элегантные паттерны современного JavaScript: RORO

ПЕРЕВОД

ru_vds 6 марта в 12:02 8,1k

Оригинал: Bill Sourour

Автор материала, перевод которого мы публикуем сегодня, Билл Соро, говорит, что написал первые строки кода на JavaScript вскоре после появления этого языка. По его словам, если тогда ему сказали бы, что однажды он выпустит серию статей об элегантных шаблонах проектирования в JavaScript, он умер бы со смеху. Тогда он воспринимал JS как странный маленький язык, писанину на котором можно было с большой натяжкой называть «программированием».

Но за 20 лет многое изменилось. Теперь Билл воспринимает JavaScript таким, каким видел его Дуглас Крокфорд, когда работал над книгой «JavaScript. Сильные стороны»: красивым, элегантным и выразительным динамическим языком программирования.

В этой статье Билл хочет рассказать о замечательном маленьком паттерне, которым он уже какое-то время с удовольствием пользуется. Он надеется, что этот шаблон проектирования пригодится и другим программистам. Билл говорит, что не считает себя первооткрывателем этого паттерна, скорее, речь идёт о том, что он увидел нечто подобное в чьём-то коде, а потом адаптировал это под свои нужды.

Паттерн RORO

RORO (Receive an object, return an object — получил объект, вернул объект), это паттерн, благодаря использованию которого большинство моих функций теперь принимают единственный параметр типа `object`, и многие из них также возвращают значение типа `object`, или разрешаются подобным значением.

Отчасти благодаря возможностям деструктурирования, которые появились в ES2015, я понял, что RORO — мощный и полезный паттерн. Я даже дал ему название — RORO, которое звучит несколько несерьёзно, наверное, для того, чтобы особым образом его выделить.

Хочу обратить ваше внимание на то, что деструктурирование — это одна из моих любимых возможностей в современном JavaScript. Мы поговорим о сильных сторонах этого механизма немного позже, поэтому, если вы с деструктурированием не знакомы, можете, чтобы дальше вам было читать интереснее, посмотреть это видео.

Вот несколько причин, по которым вам может понравиться паттерн RORO:

- В нём используются именованные параметры функций.
- Он позволяет более чётко выделять параметры функций, назначаемые по умолчанию.
- Он помогает возвращать из функций сложные наборы значений.
- Он упрощает композицию функций.

Рассмотрим каждую из этих возможностей RORO подробнее.

Именованные параметры

Представим, что у нас имеется функция, которая возвращает список пользователей (Users) с заданной ролью (Role), и предположим, что нам нужно предоставить этой функции аргумент,

который нужен для включения в выходные данные контактных сведений (Contact Info) каждого пользователя, и ещё один аргумент для включения в выдачу неактивных (Inactive) пользователей. При традиционном подходе объявление такой функции могло бы выглядеть так:

```
function findUsersByRole (
  role,
  withContactInfo,
  includeInactive
) {...}
```

Вызов этой функции выглядел бы так:

```
findUsersByRole(
  'admin',
  true,
  true
)
```

Обратите внимание на то, что при чтении кода роль последних двух аргументов в этом вызове совершенно непонятна. Что означают два значения `true`?

Что произойдёт в том случае, если приложению почти никогда не нужны контактные сведения пользователей, но почти всегда нужны данные по неактивным пользователям? Нам придётся всё время бороться с одним и тем же аргументом, даже хотя он практически никогда не меняется (позже мы поговорим об этом подробнее).

Если в двух словах, то этот традиционный подход даёт

потенциально непонятный, перегруженный ненужными деталями код, который сложно понять и непросто писать.

Взглянем, что произойдёт, если вместо обычного списка аргументов на входе функции ожидается единственный объект с аргументами:

```
function findUsersByRole ({
  role,
  withContactInfo,
  includeInactive
}) {...}
```

Обратите внимание на то, что объявление функции выглядит практически так же, как и в предыдущем примере, единственное различие заключается в том, что параметры теперь включены в фигурные скобки. Это указывает на то, что функция теперь, вместо получения трёх отдельных аргументов, ожидает один объект со свойствами `role`, `withContactInfo`, и `includeInactive`.

Этот механизм работает благодаря деструктурированию — возможности, которая появилась в ES2015.

Теперь мы можем вызывать функцию так:

```
findUsersByRole({
  role: 'admin',
  withContactInfo: true,
  includeInactive: true
})
```

Так в коде оказывается куда меньше неоднозначностей, теперь его легче читать, он получился гораздо понятнее. Кроме того, пропуск аргументов или изменение их порядка больше не приводит к возникновению проблем, так как параметры теперь представлены именованными свойствами объекта.

Например, мы можем вызвать функцию так:

```
findUsersByRole({
  withContactInfo: true,
  role: 'admin',
  includeInactive: true
})
```

А можем и так:

```
findUsersByRole({
  role: 'admin',
  includeInactive: true
})
```

Кроме того, новые параметры можно добавлять так, что их появление не поломает код, написанный ранее.

Тут стоит отметить одну важную вещь, которая связана с возможностью вызова функций без аргументов, то есть, с тем, чтобы все параметры функции могли бы быть необязательными. Вызов подобной функции выглядит так:

```
findUsersByRole()
```

Для того, чтобы нашу функцию можно было бы вызывать без аргументов, нужно задать значения аргументов по умолчанию:

```
function findUsersByRole ({
  role,
  withContactInfo,
  includeInactive
} = {}) {...}
```

Дополнительная польза от применения деструктурирования для объекта с параметрами заключается в том, что такой подход способствует иммутабельности. Когда мы деструктурируем объект при поступлении его в функцию, мы присваиваем свойства объекта новым переменным. Изменение значений этих переменных не приведёт к изменению исходного объекта.

Рассмотрим следующий пример:

```
const options = {
  role: 'Admin',
  includeInactive: true
}
findUsersByRole(options)
function findUsersByRole ({
  role,
  withContactInfo,
  includeInactive
} = {}) {
  role = role.toLowerCase()
  console.log(role) // 'admin'
  ...
}
console.log(options.role) // 'Admin'
```

Тут, несмотря на то, что мы изменили значение переменной `role`, значение свойства объекта `options.role` осталось неизменным.

Стоит отметить, что деструктурирование делает мелкую копию объекта, в результате, если любое из свойств объекта с аргументами является объектом (например, имеет тип `array` или `object`), то изменение такого свойства, даже назначенного отдельной переменной, приведёт к изменению исходного объекта. Выражаю благодарностью Юрию Хомякову за то, что обратил на это внимание.

Стандартные значения параметров, задаваемые по умолчанию

Благодаря новшествам ES2015 при объявлении JS-функций теперь можно задавать значения параметров, назначаемые им по умолчанию. На самом деле, мы уже демонстрировали тут использование параметров по умолчанию, добавив в объявление функции `={}` после описания объекта с параметрами.

При использовании традиционных параметров добавление значений параметров по умолчанию может выглядеть так:

```
function findUsersByRole (
  role,
  withContactInfo = true,
  includeInactive
) {...}
```

Если нам надо установить параметр `includeInactive` в `true`, требуется явным образом передать `undefined` как значение для `withContactInfo` для того, чтобы сохранить значение, заданное по умолчанию:

```
findUsersByRole(
  'Admin',
  undefined,
  true
)
```

Всё это выглядит весьма таинственно.

Сравните это с использованием в объявлении функции объекта с параметрами:

```
function findUsersByRole ({
  role,
  withContactInfo = true,
  includeInactive
} = {}) {...}
```

Теперь эту функцию можно вызывать так:

```
findUsersByRole({
  role: ‘Admin’,
  includeInactive: true
})
```

При этом значение по умолчанию, заданное для

`withContactInfo`, никуда не девается.

О необходимых параметрах функции

Тут мы немного отклонимся от нашей основной темы для того, чтобы рассмотреть один полезный приём, касающийся указания при объявлении функции параметров, которые совершенно необходимы для её правильной работы.

Часто ли вам приходилось писать нечто подобное коду, приведённому ниже?

```
function findUsersByRole ({
  role,
  withContactInfo,
  includeInactive
} = {}) {
  if (role == null) {
    throw Error(...)
  }
  ...
}
```

Обратите внимание на то, что тут используется двойной знак равенства, `==`, для того, чтобы проверить значение одновременно и на `null`, и на `undefined`.

Что если я скажу вам, что вместо всего этого можно использовать параметры по умолчанию, благодаря которым реализуется проверка поступления в функцию необходимых данных?

Для того чтобы выполнить подобную проверку, сначала нужно

объявить функцию `requiredParam()`, которая возвращает ошибку:

```
function requiredParam (param) {
  const requiredParamError = new Error(
   `Required parameter, "${param}" is missing.`
  )
  // сохраним исходный стек-трейс
  if (typeof Error.captureStackTrace === ‘function’) {
    Error.captureStackTrace(
      requiredParamError,
      requiredParam
    )
  }
  throw requiredParamError
}
```

Кстати, я знаю, что эта функция не использует RORO, но поэтому в самом начале я и не говорил о том, что абсолютно все мои функции используют этот паттерн.

Теперь мы можем, в качестве значения по умолчанию для `role`, установить вызов функции `requiredParam`:

```
function findUsersByRole ({
  role = requiredParam('role'),
  withContactInfo,
  includeInactive
} = {}) {...}
```

В результате получается, что если кто-нибудь вызовет функцию `findUserByRole`, не указав `role`, он увидит сообщение об ошибке, которое указывает на то, что он забыл передать в функцию необходимый параметр `role`, что в нашем случае будет

выглядеть как `Required parameter, "role" is missing`.

С технической точки зрения мы можем использовать этот подход и с обычными параметрами по умолчанию. Объект здесь — дело добровольное. Однако, это очень уж удобно, поэтому я тут об этом и рассказал.

Возврат из функций сложных наборов значений

Функции в JavaScript могут возвращать лишь одно значение. Если это значение имеет тип `object`, оно может содержать в себе очень много всего интересного.

Представим себе функцию, которая сохраняет объекты `User` в базу данных. Когда эта функция возвращает объект, она может передать туда, откуда она была вызвана, гораздо больше информации, чем без использования такого подхода.

Например, обычно при сохранении информации в базу данных в соответствующих функциях используется подход, который заключается в том, что в таблицу вставляются новые строки, если их пока не существует, если же соответствующие строки существуют, они обновляются.

В подобных случаях было бы полезно знать, какую именно операцию с базой данных выполнила наша функция сохранения информации — `INSERT` или `UPDATE`. Хорошо было бы и получить точное описание того, что именно было сохранено в базе данных, не помешали бы и подробности о результате операции. Она,

например, может завершиться успешно, может быть объединена с более крупной транзакцией и поставлена в очередь, попытка записи может оказаться неудачной, так как истёк тайм-аут на выполнение операции.

Возвращая из функции объект, очень легко поместить в него все необходимые сведения:

```
async saveUser({
  upsert = true,
  transaction,
  ...userInfo
}) {
  // сохранение данных
  return {
    operation, // например 'INSERT'
    status, // например 'Success'
    saved: userInfo
  }
}
```

Технически следующая конструкция возвращает объект `Promise`, который разрешается обычным объектом типа `object`, но, полагаю, этот пример хорошо иллюстрирует рассматриваемую здесь идею.

Упрощение композиции функций

Композиция функций — это процесс комбинирования двух или большего количества функций для создания новой функции. Композиция функций похожа на сборку системы из нескольких труб, через которые планируется пропустить наши данные.
Эрик Эллиотт

Заниматься композицией функций можно с помощью специальной функции `pipe`, объявление которой выглядит так:

```
function pipe(...fns) {
  return param => fns.reduce(
    (result, fn) => fn(result),
    param
  )
}
```

Эта функция принимает список функций и возвращает одну функцию, которая может выполнить эти функции слева направо, передав первой из этих функций аргументы, переданные в `pipe`, второй — то, что вернёт первая функция, и так далее.

Возможно, всё это выглядит несколько запутанным, поэтому сейчас мы рассмотрим пример, который расставит всё по местам. Единственное ограничение такого подхода заключается в том, что каждая функция в списке должна получать лишь один параметр. К счастью, когда мы пользуемся паттерном RORO, это — не проблема.

Вот наш пример. Тут имеется функция `saveUser`, которая пропускает объект `userInfo` через три отдельные функции, которые, соответственно, проверяют, нормализуют и сохраняют данные.

```
function saveUser(userInfo) {
  return pipe(
    validate,
```

```
    normalize,
    persist
  )(userInfo)
}
```

Мы можем использовать так называемые rest-параметры в функциях `validate`, `normalize`, и `persist` для деструктурирования только тех значений, которые нужны каждой из функций, передавая всё остальное дальше.

Вот, чтобы было понятнее, немного кода:

```
function validate(
  id,
  firstName,
  lastName,
  email = requiredParam(),
  username = requiredParam(),
  pass = requiredParam(),
  address,
  ...rest
) {
  // выполняем какие-то проверки
  return {
    id,
    firstName,
    lastName,
    email,
    username,
    pass,
    address,
    ...rest
  }
}
function normalize(
  email,
  username,
  ...rest
) {
  // нормализуем данные
  return {
    email,
```

```
    username,
    ...rest
  }
}
async function persist({
  upsert = true,
  ...info
}) {
  // сохраняем userInfo в базе данных
  return {
    operation,
    status,
    saved: info
  }
}
```

RO иль не RO — вот в чём вопрос

В самом начале я говорил о том, что большинство моих функций принимают объекты, и многие из них также возвращают объекты. Многие, но не все. Как и любой паттерн, RORO стоит рассматривать лишь как один из инструментов в арсенале разработчика. Мы используем его там, где он полезен, делая список параметров функции более понятным и гибким, а возвращаемое функцией значение — более выразительным.

Если вы пишете функцию, которой вполне достаточно передать один аргумент, то передавать ей объект — это уже перебор. Аналогично, если вы пишете функцию, которая может передать в место вызова простое значение, которое чётко и понятно выражает всё, что нужно, то ясно, что такая функция вовсе не должна возвращать значение типа `object`.

Например, я практически никогда не использую RORO в функциях проверки утверждений. Предположим, имеется функция

`isPositiveInteger`, проверяющая переданное ей значение на то, является ли оно неотрицательным целым числом. При написании подобной функции, весьма вероятно то, что от паттерна RORO никакого толку не будет. Однако, при разработке JavaScript-приложений этот паттерн может пригодиться во многих других ситуациях.

Уважаемые читатели! Планируете ли вы использовать паттерн RORO в своём коде? А может быть, вы уже пользуетесь чем-то подобным?

Комментарии (13)

Похожие публикации

JavaScript и ужасы мутаций

ПЕРЕВОД

48

ru_vds • 19 января в 11:42

Машины состояний и разработка веб-приложений

8

ПЕРЕВОД

ru_vds • 18 января в 12:02

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

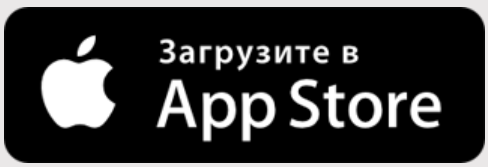
marks • вчера в 14:46

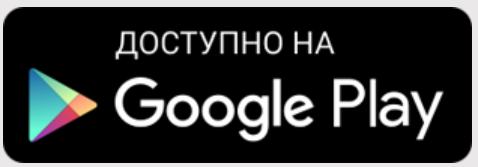
Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32

Мобильное приложение





Полная версия