Глава 1

РАБОТА С ПОРОЖДАЮЩИМИ ПАТТЕРНАМИ ПРОЕКТИРОВАНИЯ

Паттерны проектирования JavaScript — это приемы, позволяющие писать более надежные, масштабируемые и расширяемые приложения на языке JavaScript. JavaScript очень популярный язык программирования, отчасти благодаря занятой нише предоставления интерактивной функциональности на веб-страницах. Другая причина популярности JavaScript — его легкость, динамичность, мультипарадигмальность. То есть паттерны проектирования из других экосистем могут быть адаптированы для использования сильных сторон этого языка. Сильные и слабые стороны JavaScript также могут стать основой для создания новых паттернов, характерных для него и контекстов его применения.

Порождающие паттерны проектирования задают структуру создания объектов, что позволяет разрабатывать системы и приложения, в которых различным модулям, классам и объектам не нужно знать, как создавать экземпляры друг друга. Будут рассмотрены паттерны проектирования, наиболее актуальные для JavaScript, — Прототип, Одиночка и Фабрика, а также ситуации, в которых они могут быть полезны, и способы их идиоматической реализации.

В этой главе мы поговорим на следующие темы:

- исчерпывающее определение паттернов порождающего проектирования и определения таких паттернов, как Прототип, Одиночка и Фабрика;
- несколько реализаций паттерна Прототип и примеры его использования;

- реализация паттерна проектирования Одиночка, немедленная и отложенная инициализации, случаи использования паттерна Одиночка и то, как он выглядит в современном JavaScript;
- как реализовать паттерн Фабрика с помощью классов, современная альтернатива JavaScript и примеры использования.

К концу главы вы сможете определять, в каких случаях полезен порождающий паттерн проектирования, и принимать взвешенное решение о том, какую из его многочисленных реализаций использовать — от более идиоматичной JavaScript-формы до классической.

Что такое порождающие паттерны проектирования

Созданием объектов занимаются порождающие паттерны проектирования. Они позволяют потребителю генерировать экземпляры объектов, не зная деталей инстанцирования объекта. Поскольку в объектно-ориентированных языках инстанцирование объектов ограничено конструктором класса, возможность создавать экземпляры объектов без вызова конструктора полезна для уменьшения шума и сильной связанности между потребителем и инстанцируемым классом.

В JavaScript существует двусмысленность, когда мы говорим о «создании объекта», поскольку мультипарадигмальная природа JavaScript означает, что объекты можно создавать без класса или конструктора. Например, создание объекта с помощью литерала объекта в JavaScript выглядит следующим образом: `const config = { forceUpdate: true }`. На самом деле современный идиоматический JavaScript больше склоняется к процедурной и функциональной парадигмам, чем к объектной ориентации. Это означает, что для полноценного использования порождающих паттернов проектирования в JavaScript может потребоваться их адаптация.

В целом такие паттерны полезны в объектно-ориентированном JavaScript, поскольку они скрывают подробности инстанцирования от потребителей, что сохраняет слабую связанность и позволяет лучше разделять модули.

В следующем разделе мы познакомимся с первым порождающим паттерном проектирования — паттерном Прототип.

Реализация паттерна Прототип в JavaScript

Паттерн проектирования Прототип позволяет создать экземпляр на основе другого существующего экземпляра (нашего прототипа).

Если говорить более формально, то класс `prototype` предоставляет метод `clone()`. Потребительский код вместо использования выражения `new SomeClass` выполнит вызов `new SomeClassPrototype(someClassInstance).clone()`. Этот метод вернет новый экземпляр `SomeClass` с копией всех значений `someClassInstance`.

Реализация

Представим себе сценарий создания шахматной доски. Существует два основных типа квадратов — белые и черные. В дополнение к этой информации каждый квадрат содержит сведения о строке, файле и о том, какая фигура находится на нем.

Конструктор класса `BoardSquare` может выглядеть следующим образом:

```
class BoardSquare {
  constructor(color, row, file, startingPiece) {
    this.color = color;
    this.row = row;
    this.file = file;
  }
}
```

Полезными методами в `BoardSquare` будут `occupySquare` и `clearSquare`, как показано ниже:

```
class BoardSquare {
  // изменений в остальной части класса нет
  occupySquare(piece) {
    this.piece = piece;
  }
  clearSquare() {
    this.piece = null;
  }
}
```

Инстанцирование `BoardSquare` довольно громоздко из-за всех его свойств:

```
const whiteSquare = new BoardSquare('white');
const whiteSquareTwo = new BoardSquare('white');
// ...
const whiteSquareLast = new BoardSquare('white');
```

Обратите внимание на повторение аргументов, передаваемых в `new BoardSquare`, что вызовет проблемы, если потребуется изменить все квадраты доски на черные. Нам придется изменять параметр, передаваемый каждому вызову `BoardSquare`, по очереди для каждого вызова `new BoardSquare`. Это может привести к ошибкам: достаточно допустить одну трудноуловимую опечатку или неточность в значении аргумента `color`, чтобы вызвать сбой:

```
const blackSquare = new BoardSquare('black');
const blackSquareTwo = new BoardSquare('black');
// ...
const blackSquareLast = new BoardSquare('black');
```

Реализация нашей логики инстанцирования с помощью классического прототипа выглядит следующим образом. Нам потребуется класс `BoardSquarePrototype` — его конструктор принимает свойство `prototype`, сохраняемое в экземпляре. Класс `BoardSquarePrototype` предоставляет метод `clone()`, не принимающий никаких аргументов и возвращающий экземпляр `BoardSquare` с копией всех свойств `prototype`:

```
class BoardSquarePrototype {
  constructor(prototype) {
    this.prototype = prototype;
  }
  clone() {
    const boardSquare = new BoardSquare();
    boardSquare.color = this.prototype.color;
    boardSquare.row = this.prototype.row;
    boardSquare.file = this.prototype.file;
    return boardSquare;
  }
}
```

Использование класса `BoardSquarePrototype` потребует выполнения следующих шагов.

1. Сначала понадобится инициализация экземпляра `BoardSquare` — в данном случае со значением белого цвета — `'white'`. После этого оно будет передано в качестве свойства `prototype` во время вызова конструктора `BoardSquarePrototype`:

   ```
   const whiteSquare = new BoardSquare('white');
   const whiteSquarePrototype = new BoardSquarePrototype(whiteSquare);
   ```

2. Затем можно использовать `whiteSquarePrototype` с методом `.clone()` для создания копий экземпляра `whiteSquare`. Обратите внимание, что цвет

color копируется, но каждый вызов метода clone() возвращает новый экземпляр.

```
const whiteSquareTwo = whiteSquarePrototype.clone();
// ...
const whiteSquareLast = whiteSquarePrototype.clone();

console.assert(
  whiteSquare.color === whiteSquareTwo.color &&
    whiteSquareTwo.color === whiteSquareLast.color,
  'Prototype.clone()-ed instances have the same color as the prototype'
);
console.assert(
  whiteSquare !== whiteSquareTwo &&
    whiteSquare !== whiteSquareLast &&
    whiteSquareTwo !== whiteSquareLast,
  'each Prototype.clone() call outputs a different instances'
);
```

Согласно утверждениям в коде клонированные экземпляры содержат одно и то же значение для color, но представляют собой разные экземпляры объекта Square.

Вариант применения

Чтобы проиллюстрировать, что нужно сделать для превращения белого квадрата в черный, посмотрим на пример кода, в котором значение 'white' не упоминается в именах переменных:

```
const boardSquare = new BoardSquare('white');
const boardSquarePrototype = new BoardSquarePrototype(boardSquare);

const boardSquareTwo = boardSquarePrototype.clone();
// ...
const boardSquareLast = boardSquarePrototype.clone();

console.assert(
  boardSquareTwo.color === 'white' &&
    boardSquare.color === boardSquareTwo.color &&
    boardSquareTwo.color === boardSquareLast.color,
  'Prototype.clone()-ed instances have the same color as the prototype'
);
console.assert(
  boardSquare !== boardSquareTwo &&
    boardSquare !== boardSquareLast &&
    boardSquareTwo !== boardSquareLast,
  'each Prototype.clone() call outputs a different instances'
);
```

В этом сценарии нам достаточно изменить значение свойства `color`, переданное в конструктор `BoardSquare`, чтобы изменить цвет всех экземпляров, клонированных из прототипа:

```
const boardSquare = new BoardSquare('black');
// изменений в остальной части кода нет
console.assert(
  boardSquareTwo.color === 'black' &&
    boardSquare.color === boardSquareTwo.color &&
    boardSquareTwo.color === boardSquareLast.color,
  'Prototype.clone()-ed instances have the same color as the prototype'
);
console.assert(
  boardSquare !== boardSquareTwo &&
    boardSquare !== boardSquareLast &&
    boardSquareTwo !== boardSquareLast,
  'each Prototype.clone() call outputs a different instances'
);
```

Паттерн Прототип полезен в ситуациях, когда необходимо создать «шаблон» для экземпляров объекта. Это хороший паттерн для создания «объекта по умолчанию», но с пользовательскими значениями. То есть данный способ позволяет быстрее и проще вносить изменения, поскольку они выполняются один раз для объекта-шаблона, но применяются ко всем клонированным экземплярам метода `clone()`.

Повышение устойчивости к изменениям в переменных экземпляра прототипа с помощью современного JavaScript

Можно внести некоторые улучшения в реализацию прототипа в JavaScript.

Первое будет касаться метода `clone()`. Чтобы сделать класс-прототип устойчивым к изменениям в конструкторе прототипа/переменных экземпляра, нельзя копировать свойства одно за другим.

Например, если добавить новый параметр `startingPiece`, который конструктор `BoardSquare` будет принимать и устанавливать переменную экземпляра `piece` в его значение, текущая реализация класса `BoardSquarePrototype` не сможет его скопировать, поскольку она копирует только параметры `color`, `row` и `file`:

```
class BoardSquare {
  constructor(color, row, file, startingPiece) {
    this.color = color;
    this.row = row;
    this.file = file;
```

```
    this.piece = startingPiece;
  }
  // оставшаяся часть класса остается без изменений
}

const boardSquare = new BoardSquare('white', 1, 'A', 'king');
const boardSquarePrototype = new BoardSquarePrototype(boardSquare);
const otherBoardSquare = boardSquarePrototype.clone();

console.assert(
  otherBoardSquare.piece === undefined,
  'prototype.piece was not copied over'
);
```

Примечание

Ссылка на метод Object.assign: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign.

Если изменить класс `BoardSquarePrototype`, используя метод `Object.assign(new BoardSquare(), this.prototype)`, он скопирует все перечисленные свойства прототипа:

```
class BoardSquarePrototype {
  constructor(prototype) {
    this.prototype = prototype;
  }
  clone() {
    return Object.assign(new BoardSquare(), this.prototype);
  }
}

const boardSquare = new BoardSquare('white', 1, 'A', 'king');
const boardSquarePrototype = new BoardSquarePrototype(boardSquare);
const otherBoardSquare = boardSquarePrototype.clone();

console.assert(
  otherBoardSquare.piece === 'king' && otherBoardSquare.piece === boardSquare.piece,
  'prototype.piece was copied over'
);
```

Паттерн Прототип без классов в JavaScript

По историческим причинам концепция прототипа глубоко укоренилась в JavaScript. На самом деле классы появились в стандарте ECMAScript гораздо позже, в версии ECMAScript 6, выпущенной в 2015 году (для справки, версия ECMAScript 1 вышла в 1997 году).

Именно поэтому многие разработчики на JavaScript полностью отказываются от классов. «Объект-прототип» в JavaScript можно использовать для того, чтобы объекты наследовали методы и переменные друг от друга.

Один из способов клонировать объекты с их методами — задействовать метод `Object.create`. Он создан на основе системы прототипов JavaScript:

```
const square = {
  color: 'white',
  occupySquare(piece) {
    this.piece = piece;
  },
  clearSquare() {
    this.piece = null;
  },
};
const otherSquare = Object.create(square);
```

Тонкость данного приема в том, что `Object.create` на самом деле ничего не копирует — он просто создает новый объект и устанавливает объект `square` в качестве прототипа. Это означает, что, если объекты не найдены в экземпляре `otherSquare`, они доступны в `square`:

```
console.assert(otherSquare.__proto__ === square, 'uses JS prototype');

console.assert(
  otherSquare.occupySquare === square.occupySquare &&
    otherSquare.clearSquare === square.clearSquare,
  "methods are not copied, they're 'inherited' using the prototype"
);

delete otherSquare.color;
console.assert(
  otherSquare.color === 'white' && otherSquare.color === square.color,
  'data fields are also inherited'
);
```

Еще один момент, связанный с прототипом JavaScript и его существованием до того, как классы стали частью JavaScript: подклассы в JavaScript представляют собой другой синтаксис для указания прототипа объекта. Взгляните на следующий пример использования ключевого слова `extends`.

Выражение `BlackSquare extends Square` задает свойство `prototype.__proto__` объекта `BlackSquare` прототипу `Square.prototype`:

```
class Square {
  constructor() {}
  occupySquare(piece) {
    this.piece = piece;
```

```
  }
  clearSquare() {
    this.piece = null;
  }
}

class BlackSquare extends Square {
  constructor() {
    super();
    this.color = 'black';
  }
}

console.assert(
  BlackSquare.prototype.__proto__ === Square.prototype,
  'subclass prototype has prototype of superclass'
);
```

В этом разделе мы узнали, как реализовать паттерн Прототип с помощью класса прототипа, предоставляющего метод `clone()`. Мы выяснили, в каких ситуациях могут быть полезны паттерны Прототип и как еще улучшить нашу реализацию прототипа с помощью современных возможностей JavaScript. Мы также рассказали о «прототипе» JavaScript, о том, почему он существует и как связан с паттерном Прототип.

Далее мы рассмотрим еще один порождающий паттерн проектирования — паттерн Одиночка, а также некоторые подходы к его реализации в JavaScript и примеры использования.

Паттерн Одиночка с немедленной и отложенной инициализацией в JavaScript

Для начала определим, что такое *паттерн проектирования Одиночка* (Синглтон).

Паттерн Одиночка позволяет инстанцировать объект только один раз, предоставляет этот единственный экземпляр потребителям и контролирует инстанцирование единственного экземпляра.

Одиночка предоставляет еще один способ получить доступ к экземпляру объекта без использования конструктора, хотя для этого необходимо, чтобы объект был спроектирован в качестве одиночного.

Реализация

Классическим примером является журнал. Редко возникает необходимость инстанцировать несколько журналов в приложении. Использование паттерна Одиночка означает, что место инициализации контролируется, а конфигурация журнала будет согласованной во всем приложении. Например, независимо от того, из какой части приложения мы вызываем журнал, его уровень не поменяется.

Простой журнал выглядит следующим образом: конструктор принимает параметры `logLevel` и `transport`, а также приватный метод `isLevelEnabled`, что позволяет отбрасывать записи, которые журнал не настроен хранить (например, когда уровень принимает значение `warn`, сообщения типа `info` отбрасываются). Наконец, журнал реализует методы `info`, `warn` и `error`. Они ведут себя, как описано ранее: вызывают соответствующий метод `transport`, только если уровень «включен» (то есть «выше» настроенного уровня журнала).

Возможные значения `logLevel`, влияющие на `isLevelEnabled`, хранятся как статическое поле в классе `Logger`:

```
class Logger {
  static logLevels = ['info', 'warn', 'error'];
  constructor(logLevel = 'info', transport = console) {
    if (Logger.#loggerInstance) {
      throw new TypeError(
        'Logger is not constructable, use getInstance() instead'
      );
    }
    this.logLevel = logLevel;
    this.transport = transport;
  }
  isLevelEnabled(targetLevel) {
    return (
      Logger.logLevels.indexOf(targetLevel) >=
      Logger.logLevels.indexOf(this.logLevel)
    );
  }
  info(message) {
    if (this.isLevelEnabled('info')) {
      return this.transport.info(message);
    }
  }
  warn(message) {
    if (this.isLevelEnabled('warn')) {
      this.transport.warn(message);
    }
  }
```

```
  error(message) {
    if (this.isLevelEnabled('error')) {
      this.transport.error(message);
    }
  }
}
```

Чтобы сделать экземпляр `Logger` синглтоном, необходимо реализовать статический метод `getInstance`, возвращающий кэшированный экземпляр. Для этого мы будем использовать статический метод `loggerInstance` для класса `Logger`. Метод `getInstance` проверит, существует ли `Logger.loggerInstance`, и вернет его при наличии. В противном случае создаст новый экземпляр класса `Logger`, установит его в качестве `loggerInstance` и вернет его:

```
class Logger {
  static loggerInstance = null;
  // оставшаяся часть класса
  static getInstance() {
    if (!Logger.loggerInstance) {
      Logger.loggerInstance = new Logger('warn', console);
    }
    return Logger.loggerInstance;
  }
}
```

Использовать это в другом модуле так же просто, как вызвать метод `Logger.getInstance()`. Все вызовы `getInstance` вернут один и тот же экземпляр класса `Logger`:

```
const a = Logger.getInstance();
const b = Logger.getInstance();
console.assert(a === b, 'Logger.getInstance() returns the same reference');
```

Мы реализовали одиночку с отложенной (ее еще называют «ленивой») инициализацией. Инициализация происходит при первом вызове метода `getInstance`. В следующем разделе мы рассмотрим, как можно расширить код, чтобы «немедленно» инициализировать `loggerInstance`, когда экземпляр `loggerInstance` будет инициализироваться при оценке кода `Logger`.

Обеспечение создания только одного экземпляра одиночки

Характерной чертой синглтона является концепция «единственного экземпляра». Мы хотим «заставить» потребителей использовать метод `getInstance`.

Для этого можно проверить существование экземпляра `loggerInstance` при вызове конструктора:

```
class Logger {
  // оставшаяся часть класса
  constructor(logLevel = 'info', transport = console) {
    if (Logger.loggerInstance) {
      throw new TypeError(
        'Logger is not constructable, use getInstance() instead'
      );
    }
    this.logLevel = logLevel;
    this.transport = transport;
  }
  // оставшаяся часть класса
}
```

В случае вызова метода `getInstance` (и, следовательно, `Logger.loggerInstance` заполняется) конструктор будет выдавать ошибку:

```
Logger.getInstance();
new Logger('info', console); // new TypeError('Logger is not constructable,
use getInstance() instead');
```

Такое поведение полезно для того, чтобы потребители не инстанцировали свой собственный журнал, а использовали метод `getInstance`. Все потребители, задействующие метод `getInstance`, подразумевают, что конфигурация для установки регистратора инкапсулируется классом `Logger`.

В реализации все еще есть пробел, поскольку вызов конструктора `new Logger()` перед любым вызовом метода `getInstance()` будет успешным, как показано в следующем примере:

```
new Logger('info', console); // Logger { logLevel: 'info', transport: ... }
new Logger('info', console); // Logger { logLevel: 'info', transport: ... }
Logger.getInstance();
new Logger('info', console); // new TypeError('Logger is not constructable,
use getInstance() instead');
```

В многопоточных языках наша реализация также будет иметь потенциальное состояние гонки — одновременный вызов нескольких потребителей `Logger.getInstance()` может привести к существованию нескольких экземпляров. Однако, поскольку популярные среды выполнения JavaScript являются однопоточными, нам не придется беспокоиться об этом. Поскольку `getInstance` — синхронный метод, несколько его вызовов будут интерпретироваться один за другим. Для справки, Node.js, Deno и основные браузеры Chrome, Safari, Edge и Firefox предоставляют однопоточную среду выполнения JavaScript.