7 июня 2022 г.

Сборка мусора

Управление памятью в JavaScript выполняется автоматически и незаметно. Мы создаём примитивы, объекты, функции… Всё это занимает память.

Но что происходит, когда что-то больше не нужно? Как движок JavaScript обнаруживает, что пора очищать память?

Достижимость

Основной концепцией управления памятью в JavaScript является принцип *достижимости*.

Если упростить, то «достижимые» значения – это те, которые доступны или используются. Они гарантированно находятся в памяти.

1. Существует базовое множество достижимых значений, которые не могут быть удалены.

   Например:

   - Выполняемая в данный момент функция, её локальные переменные и параметры.
   - Другие функции в текущей цепочке вложенных вызовов, их локальные переменные и параметры.
   - Глобальные переменные.
   - (некоторые другие внутренние значения)

   Эти значения мы будем называть *корнями*.

2. Любое другое значение считается достижимым, если оно доступно из корня по ссылке или по цепочке ссылок.

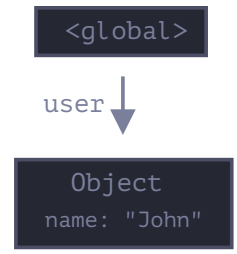
   Например, если в глобальной переменной есть объект, и он имеет свойство, в котором хранится ссылка на другой объект, то *этот* объект считается достижимым. И те, на которые он ссылается, тоже достижимы. Далее вы познакомитесь с подробными примерами на эту тему.

В движке JavaScript есть фоновый процесс, который называется сборщиком мусора. Он отслеживает все объекты и удаляет те, которые стали недоступными.

Простой пример

Вот самый простой пример:

```
1  // в user находится ссылка на объект
2  let user = {
3    name: "John"
4  };
```



Здесь стрелка обозначает ссылку на объект. Глобальная переменная `user` ссылается на объект `{name: "John"}` (мы будем называть его просто «John» для краткости). В свойстве `"name"` объекта John хранится примитив, поэтому оно нарисовано внутри объекта.

Если перезаписать значение `user`, то ссылка потеряется:

```
user = null;
```

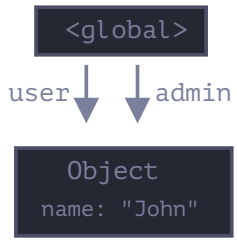
Теперь объект John становится недостижимым. К нему нет доступа, на него нет ссылок. Сборщик мусора удалит эти данные и освободит память.

Две ссылки

Представим, что мы скопировали ссылку из `user` в `admin`:

```
// в user находится ссылка на объект
let user = {
  name: "John"
};

let admin = user;
```



Теперь, если мы сделаем то же самое:

```
user = null;
```

…то объект John всё ещё достижим через глобальную переменную `admin`, поэтому он находится в памяти. Если бы мы также перезаписали `admin`, то John был бы удалён.

Взаимосвязанные объекты

Теперь более сложный пример. Семья:

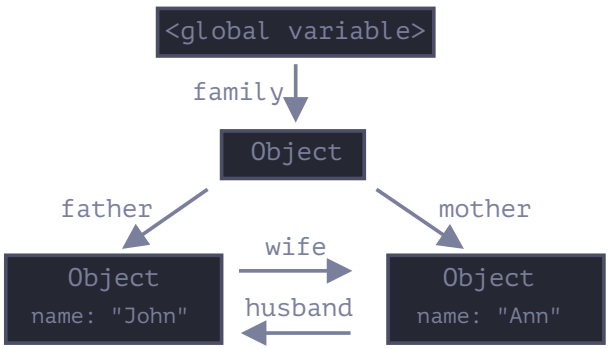
```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

  return {
    father: man,
    mother: woman
  }
}

let family = marry({
  name: "John"
}, {
  name: "Ann"
});
```

Функция `marry` «женит» два объекта, давая им ссылки друг на друга, и возвращает новый объект, содержащий ссылки на два предыдущих.

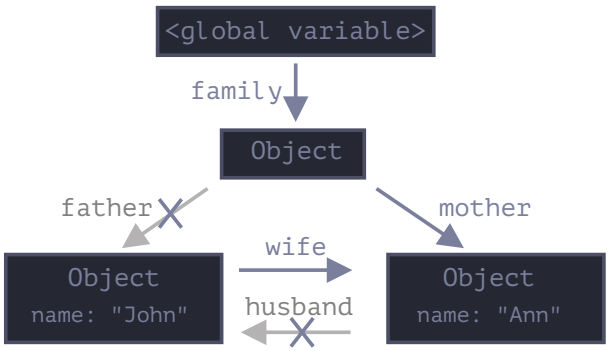
В результате получаем такую структуру памяти:



На данный момент все объекты достижимы.

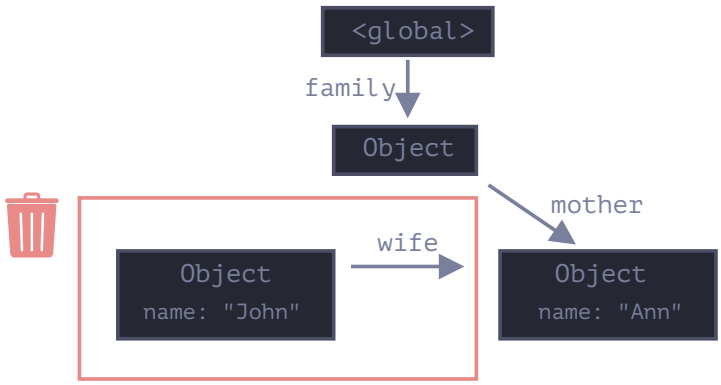
Теперь удалим две ссылки:

```
delete family.father;
delete family.mother.husband;
```



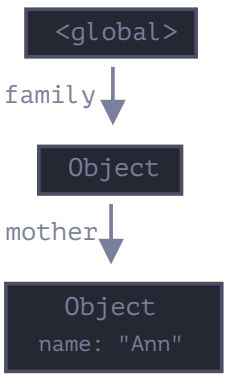
Недостаточно удалить только одну из этих двух ссылок, потому что все объекты останутся достижимыми.

Но если мы удалим обе, то увидим, что у объекта John больше нет входящих ссылок:



Исходящие ссылки не имеют значения. Только входящие ссылки могут сделать объект достижимым. Объект John теперь недостижим и будет удалён из памяти со всеми своими данными, которые также стали недоступны.

После сборки мусора:



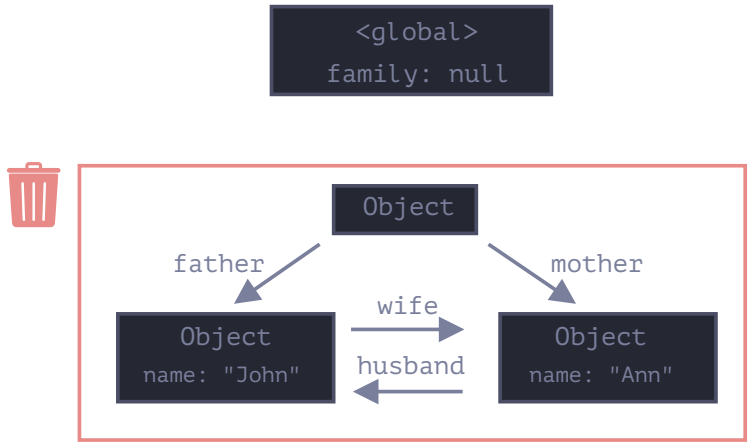
Недостижимый «остров»

Вполне возможна ситуация, при которой целый «остров» взаимосвязанных объектов может стать недостижимым и удалиться из памяти.

Возьмём объект `family` из примера выше. А затем:

```
family = null;
```

Структура в памяти теперь станет такой:



Этот пример демонстрирует, насколько важна концепция достижимости.

Объекты John и Ann всё ещё связаны, оба имеют входящие ссылки, но этого недостаточно.

Бывший объект `family` был отсоединён от корня, на него больше нет ссылки, поэтому весь «остров» становится недостижимым и будет удалён.

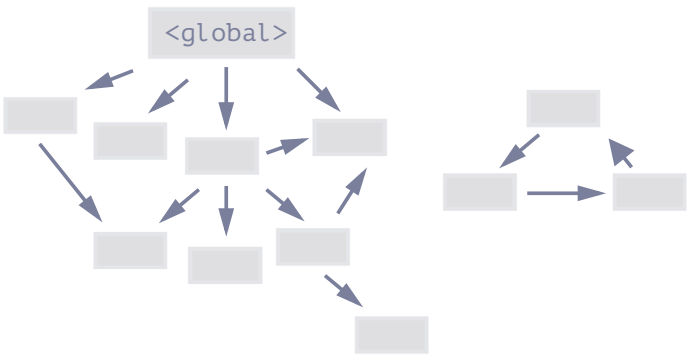
Внутренние алгоритмы

Основной алгоритм сборки мусора называется «алгоритм пометок» (от англ. «mark-and-sweep»).

Согласно этому алгоритму, сборщик мусора регулярно выполняет следующие шаги:

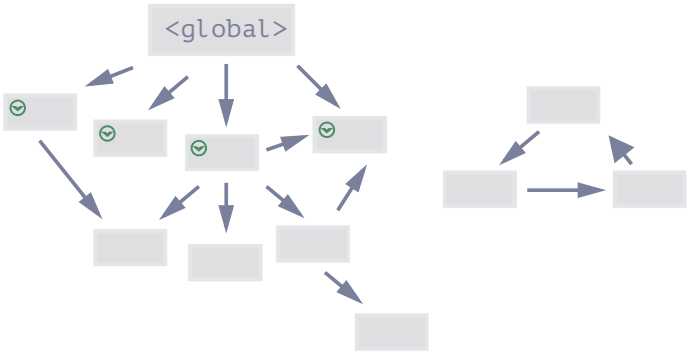
- Сборщик мусора «помечает» (запоминает) все корневые объекты.
- Затем он идёт по ним и «помечает» все ссылки из них.
- Затем он идёт по отмеченным объектам и отмечает их ссылки. Все посещённые объекты запоминаются, чтобы в будущем не посещать один и тот же объект дважды.
- …И так далее, пока не будут посещены все достижимые (из корней) ссылки.
- Все непомеченные объекты удаляются.

Например, пусть наша структура объектов выглядит так:

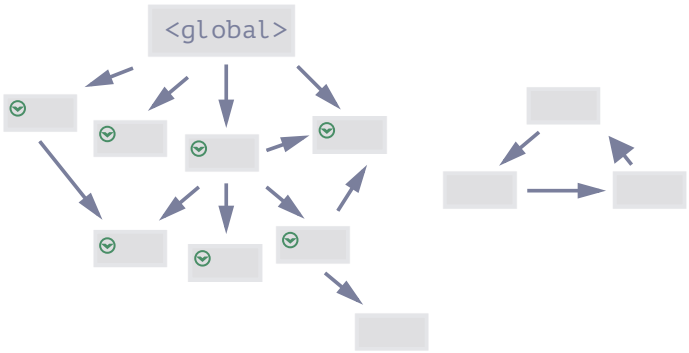


Мы ясно видим «недостижимый остров» справа. Теперь давайте посмотрим, как будет работать «алгоритм пометок» сборщика мусора.

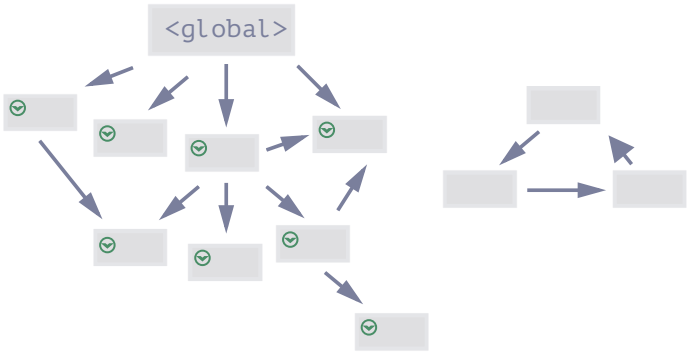
На первом шаге помечаются корни:



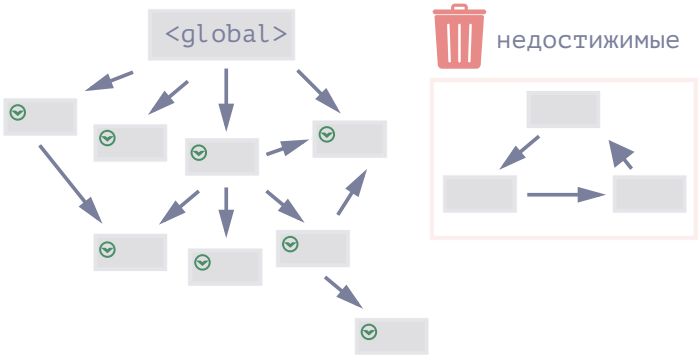
Затем помечаются объекты по их ссылкам:



...А затем объекты по их ссылкам и так далее, пока это возможно:



Теперь объекты, которые не удалось посетить в процессе, считаются недостижимыми и будут удалены:



Мы также можем представить себе этот процесс как выливание огромного ведра краски из корней, которая течёт по всем ссылкам и отмечает все достижимые объекты. Затем непомеченные удаляются.

Это концепция того, как работает сборка мусора. Движки JavaScript применяют множество оптимизаций, чтобы она работала быстрее и не задерживала выполнение кода.

Вот некоторые из оптимизаций:

- **Сборка по поколениям (Generational collection)** – объекты делятся на два набора: «новые» и «старые». В типичном коде многие объекты имеют короткую жизнь: они появляются, выполняют свою работу и быстро умирают, так что имеет смысл отслеживать новые объекты и, если это так, быстро очищать от них память. Те, которые выживают достаточно долго, становятся «старыми» и проверяются реже.
- **Инкрементальная сборка (Incremental collection)** – если объектов много, и мы пытаемся обойти и пометить весь набор объектов сразу, это может занять некоторое время и привести к видимым задержкам в выполнения скрипта. Так что движок делит всё множество объектов на части, и далее очищает их одну за другой. Получается несколько небольших сборок мусора вместо одной всеобщей. Это требует дополнительного учёта для отслеживания изменений между частями, но зато получается много крошечных задержек вместо одной большой.
- **Сборка в свободное время (Idle-time collection)** – чтобы уменьшить возможное влияние на производительность, сборщик мусора старается работать только во время простоя процессора.

Существуют и другие способы оптимизации и разновидности алгоритмов сборки мусора. Но как бы мне ни хотелось описать их здесь, я должен воздержаться, потому что разные движки реализуют разные хитрости и методы. И, что ещё более важно, все меняется по мере развития движков, поэтому изучать тему глубоко «заранее», без реальной необходимости, вероятно, не стоит. Если, конечно, это не вопрос чистого интереса, тогда для вас будет несколько ссылок ниже.

Итого

Главное, что нужно знать:

- Сборка мусора выполняется автоматически. Мы не можем ускорить или предотвратить её.
- Объекты сохраняются в памяти, пока они достижимы.
- Если на объект есть ссылка – вовсе не факт, что он является достижимым (из корня): набор взаимосвязанных объектов может стать недоступен в целом, как мы видели в примере выше.

Современные движки реализуют разные продвинутые алгоритмы сборки мусора.

О многих из них рассказано в прекрасной книге о сборке мусора «The Garbage Collection Handbook: The Art of Automatic Memory Management» (R. Jones и др.).

Если вы знакомы с низкоуровневым программированием, то более подробная информация о сборщике мусора V8 находится в статье A tour of V8: Garbage Collection.

Также в блоге V8 время от времени публикуются статьи об изменениях в управлении памятью. Разумеется, чтобы изучить сборку мусора, вам лучше подготовиться, узнав о том как устроен движок V8 внутри в целом и почитав блог Вячеслава Егорова, одного из инженеров, разрабатывавших V8. Я говорю про «V8», потому что он лучше всего освещается в статьях в Интернете. Для других движков многие подходы схожи, но сборка мусора отличается во многих аспектах.

Глубокое понимание работы движков полезно, когда вам нужна низкоуровневая оптимизация. Было бы разумно запланировать их изучение как следующий шаг после того, как вы познакомитесь с языком.

Предыдущий урок | Следующий урок

Поделиться

Карта учебника

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen…)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

Anatoliyrnd Netu
4 дня назад edited

А вот интересно после удаления из DOM при помощи innerHTML='', динамически вставленных полей input с обработчиками, "сборщик мусора" освободит память?

```
<table>
    <tr>
      <td id="td1"><button id="generateBut">add</button></td>
      <td id="td2"><button id="clearBut">clear</button></td>
    </tr>
    <tr>
      <td id="td3">
        <div id="insert"></div>
      </td>
      <td id="td4"></td>
    </tr>
  </table>
  <script>
    const ins = document.getElementById('insert')
    const txt = document.getElementById('td4')
```

показать больше

Эльдар Юсупжанов
2 месяца назад

какой же это офигенный справочник! Все объясняется очень понятным и информативным языком. Огромное спасибо автору!

хлорка
3 месяца назад

13.12.2022 19:48

Alexander Mandrov

0 9 Ответить • Поделиться ›

Akari Amano
6 месяцев назад

Зачем мне вообще знать о том, как именно работает сборка мусора и ее оптимизации? JS на то и высокоуровневый, чтобы я не задумывался о таких вещах...

1 0 Ответить • Поделиться ›

GROL COON → Akari Amano
5 месяцев назад edited

> Объекты сохраняются в памяти, пока они достижимы.

Эти знания тебе пригодятся, когда ты будешь изучать любимый вопрос на всех собеседованиях: "замыкание".)

4 0 Ответить • Поделиться ›

John → GROL COON
5 месяцев назад

Любимый вопрос "замыкание"?)) Я уже раза 3 изучал замыкания, вылетает из головы как назло

6 0 Ответить • Поделиться ›

Sokrat → Akari Amano
6 месяцев назад

На собеседовании как минимум спрашивают

1 0 Ответить • Поделиться ›

John → Sokrat
5 месяцев назад

Ну это конечно бред, я вот искал как можно этот процесс контролировать. Типа какой-нибудь код написать, типа "сиеста скрипты!, даем дорогу сборщикам мусора")

0 0 Ответить • Поделиться ›

Al. Shvets → John
2 месяца назад

John, в статье написано что ты не достижим.

5 0 Ответить • Поделиться ›

Akylbek → Al. Shvets
месяц назад

axax)

0 0 Ответить • Поделиться ›

Max_Khaletskiy

6 месяцев назад

let family = {

father: {
name: "John"
wife: {
name: "Ann"
}
}
mother: {
name: "Ann"
husband: {
name: "John"
}
}

0 0 Ответить • Поделиться ›

Lina V

7 месяцев назад

Вообще не понимаю этот момент:
" let user = {
name: "John"
};
let admin = user;
user = null;
…то объект John всё ещё достижим через глобальную переменную admin, поэтому он находится в памяти. Если бы мы также перезаписали admin, то John был бы удалён"

Вроде бы сто раз сказали, что 2 одинаковых переменных это просто 2 ссылки на один и тот же объект. И если внутри объекта произошли изменения, то они будут видны по любой из этих ссылок.
Так почему же в этом случае John остается в admin?

0 0 Ответить • Поделиться ›

Ivan → Lina V

2 месяца назад

Непонятно зачем, но решил добавить код для визуализации вопроса:

```
let car = { color: 'red', maker: 'Tesla' }
let car2 = car;
let car3 = car;

car2.color = 'blue';
car3 = null;

console.log(car); // {color: 'blue', maker: 'Tesla'}
console.log(car2); // {color: 'blue', maker: 'Tesla'}
```

```
console.log(car2); // {color: "blue", maker: "Tesla"}
console.log(car3); // null
```

0 0 Ответить • Поделиться ›

Сергей М. Lina V

4 месяца назад

Так внутри объекта ничего и не менялось

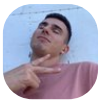
0 0 Ответить • Поделиться ›

Дима Кононов Lina V

6 месяцев назад

Пока есть ссылки на обьект--обьект живет. как только ссылок не станет -обьект будет удален. Это и есть сборщик мусора. Зачем хранить обьект в памяти ,если на него нет ссылок....

1 0 Ответить • Поделиться ›



kay Lina V

7 месяцев назад

У тебя есть сейф(объект с данными) и 2 ключа от него(переменные которые хранят в себе ссылки на один и тот же объект), если ты теряешь один ключ, то ты все еще можешь открыть сейф другим ключом, потеряешь оба ключа, потеряешь доступ к сейфу)

9 0 Ответить • Поделиться ›

Alex kay

11 дней назад

Потеряешь ключи от дома - весь остров будет удален, а место очищено бульдозером под новый объект

0 0 Ответить • Поделиться ›

Wufora Lina V

7 месяцев назад edited

Вот переменная юзер сошлась с объектом, при помощи "=" и присвоила себе только **ссылку**. Потом админ скопировал значение переменной, а именно ссылку на объект у юзера. То есть он только присвоил ссылку, но не присвоил ссылку на переменную. Потом когда юзер очистился, то админ по прежнему ссылается на объект.
В JS переменные, которые присвоили себе значение других переменных, взяли только их значение, но они не ссылаются как в Пайтоне

1 0 Ответить • Поделиться ›

L

Lina V Wufora

7 месяцев назад edited

Спасибо. В общем, я поняла, что каким-то образом операция с переменной user не затронула сам объект, а только ссылку на него. Но это все равно немного сбивает с толку. Разве присваивание null не обнуляет саму переменную/объект? Или это в принципе невозможно сделать таким образом? А если user'y undefined присвоить, это затронет объект? Или тут токо вариант вместо "Джона" поставить "Кевина" или "Макса" какого-нибудь?)

0 0 Ответить • Поделиться ›

Загрузить ещё комментарии