



Разбираем отладчик JavaScript (на основе Chrome DevTools)

Алексей Козятинский
ak239spb@gmail.com
twitter.com/ak_239



План

- ⇒ console.log
- ⇒ Асинхронные стеки
 - IO'18: Async Step Into
- ⇒ CPU profiler
- ⇒ Heap profiler
- ⇒ IO'18: Eager evaluation
- ⇒ IO'18: Step Into + Web Worker

IO'18: <https://youtu.be/mfuE53x4b3k>



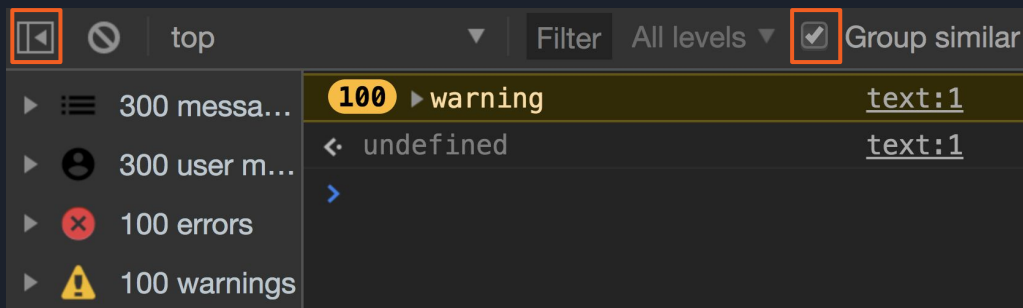
console.log

```
Object.defineProperty(window, 'a', {  
  get() {  
    console.log('DevTools are here!');  
  }  
});  
console.log(window);  
window.a = 42;
```

```
► Window {postMessage: f, blur: f, focus: f, close: f, frames: Window, ...}
```

console.log - ваш друг

- `console.log(window)` - медленный и бессмысленный
 - `console.log({a:1, b: 2})` лучше
- `console.log` игнорирует ваши геттеры
- `console.(log|warn|trace|...)` - у нас много разных методов и удобный сайдбар для их фильтрации



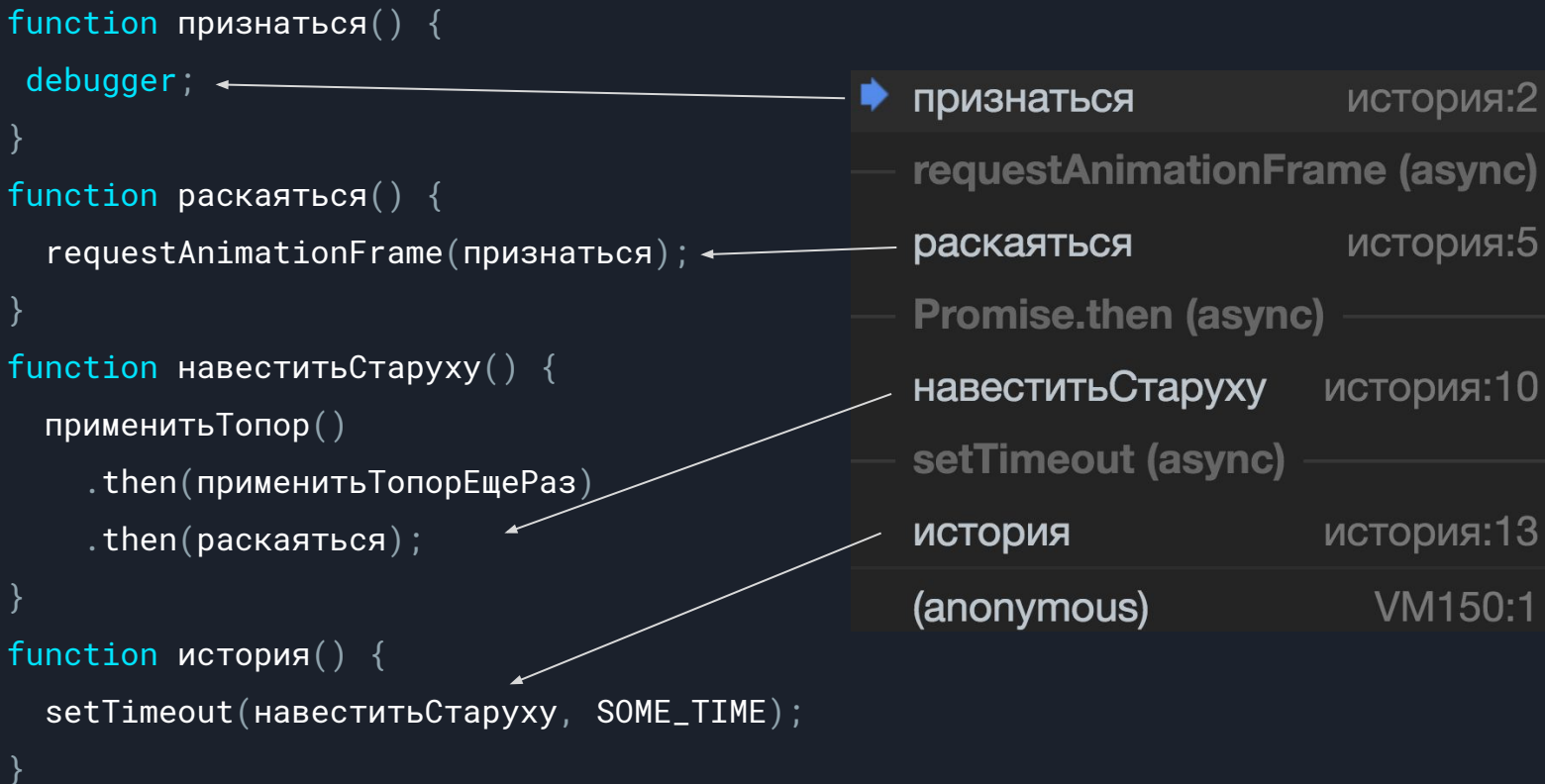
Асинхронный стек

```
function признаться() {  
  debugger;  
}
```

```
function раскаться() {  
  requestAnimationFrame(признаться);  
}
```

```
function навеститьСтаруху() {  
  применитьТопор()  
    .then(применитьТопорЕщеРаз)  
    .then(раскаться);  
}
```

```
function история() {  
  setTimeout(навеститьСтаруху, SOME_TIME);  
}
```



признаться	история:2
requestAnimationFrame (async)	
раскаться	история:5
Promise.then (async)	
навеститьСтаруху	история:10
setTimeout (async)	
история	история:13
(anonymous)	VM150:1



Асинхронный вызов

Асинхронный вызов - это любой вызов функции с использованием одного из встроенных примитивов (setTimeout, promise.then, ...), который вызывает вашу функцию в будущем.

```
setTimeout(yourFunction, 1000);  
promise.then(yourFunction);  
script.addEventListener('load', yourFunction);  
...
```

Синхронный стек - это список синхронных вызовов.

Асинхронный стек - это список синхронных и асинхронных вызовов.

Асинхронный вызов + StepInto = ❤️

```
1 function foo() {  
2   bar();  
3 }  
4 function bar() {  
5   return 42;  
6 }
```



```
1 function foo() {  
2   bar();  
3 }  
4 function bar() {  
5   return 42;  
6 }
```

```
1 function foo() {  
2   setTimeout(bar, 1000);  
3 }  
4 function bar() {  
5   return 42;  
6 }
```



```
1 function foo() {  
2   setTimeout(bar, 1000);  
3 }  
4 function bar() {  
5   return 42;  
6 }
```

Асинхронный стек: взгляд изнутри

```
function foo() {  
  setTimeout(bar, 1000);  
}
```

собираем и сохраняем текущий стек

foo async-step-into:2

```
function bar() {  
  return 42;  
}
```

используем собранный стек

bar async-step-into:5

setTimeout (async)
foo async-step-into:2

Асинхронный стек: сложность

```
function история() {  
  if (решился())  
    навеститьСтаруху();  
  setTimeout(история, ОДИН_ДЕНЬ);  
}  
setTimeout(выигратьВЛотерею, ПЯТНАДЦАТЬ_ЛЕТ);  
история();
```

➡ история	совсем-другая-история:1
— setTimeout (async)	
история	совсем-другая-история:4
— setTimeout (async)	
история	совсем-другая-история:4
— setTimeout (async)	
история	совсем-другая-история:4
— setTimeout (async)	
история	совсем-другая-история:4

...



Асинхронный стек - ваш друг

- помогает разобраться, откуда текущий синхронный код вызван
- для экономии памяти - мы отбрасываем старые асинхронные стеки
- step-into одинаково хорош с синхронными и асинхронными вызовами



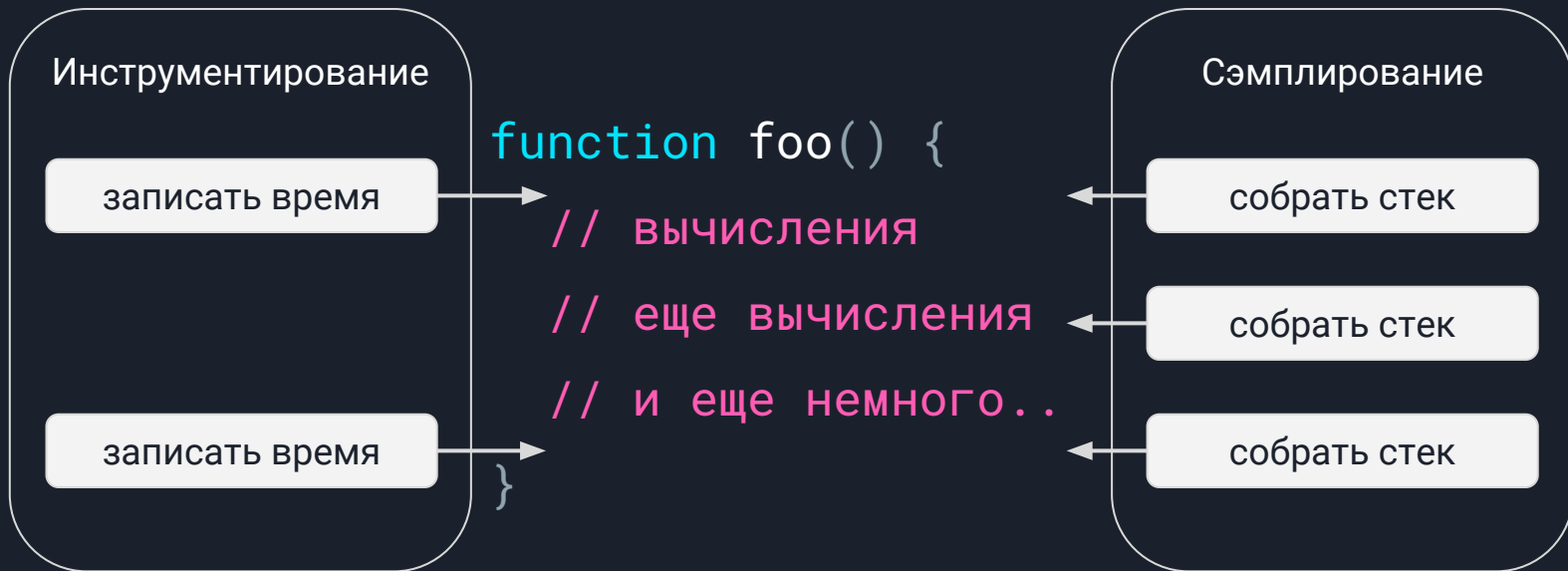
Промежуточные итоги

- `console.log` и стеки помогают разобраться, что именно происходит в вашей программе

Далее...

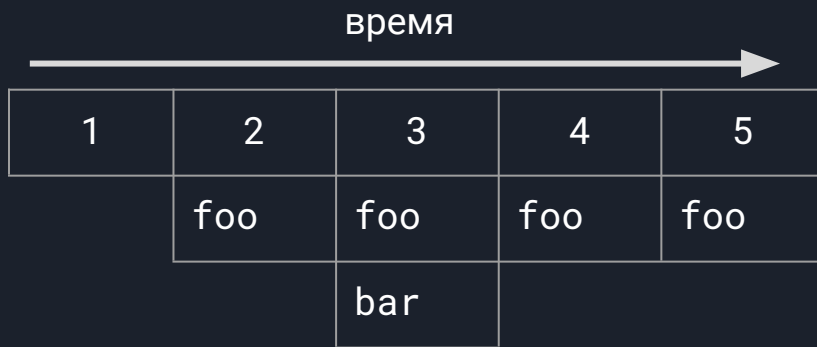
- CPU profiler - почему программа работает медленно
- Heap profiler - почему программа использует так много памяти

Профилирование



Сэмплирование

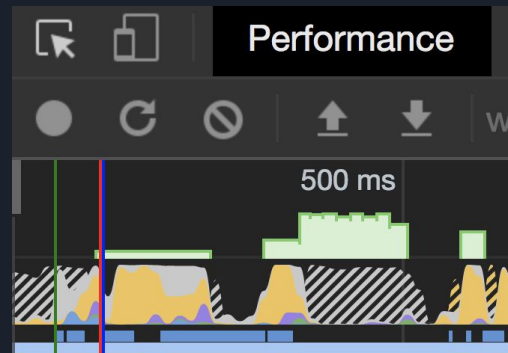
```
function bar() {  
    return 2 * 2;  
}  
  
function foo() {  
    const r = bar();  
    r *= 3;  
    return r;  
}  
  
foo()
```



	self	total
foo	3	4
bar	1	1

CPU profiler - ваш друг

- помогает найти медленные части вашего приложения
- в DevTools для JavaScript - сэмплирование - записанный профиль очень близок к реальной производительности



Неожиданное сэмплирование

```
function bar() {  
  new Int32Array(4);  
}  
  
function foo() {  
  const r = bar();  
  new Int32Array(4);  
  new Int32Array(4);  
}  
  
foo()
```



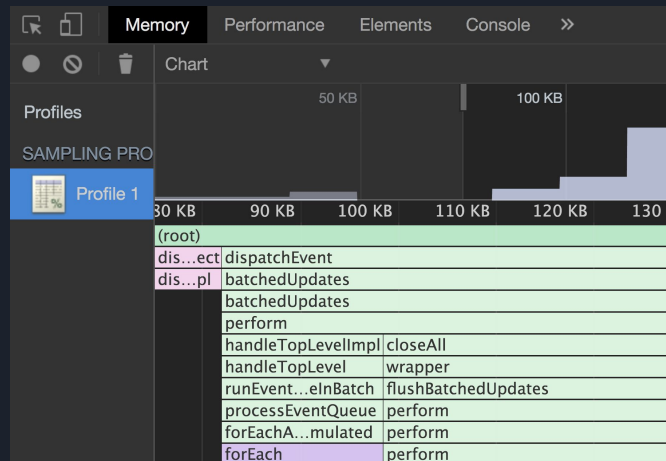
	self	total
foo	8	12
bar	4	4

Allocation sampling - ваш друг

- показывает, какая часть вашего приложения выделяет большую часть памяти
- выделение памяти - не катастрофа - пока мы ее вовремя освобождаем

Allocation sampling

Record memory allocations using sampling, which has low overhead and can be used for long periods. Allocations are broken down by JavaScript function.



Allocation instrumentation - ваш друг

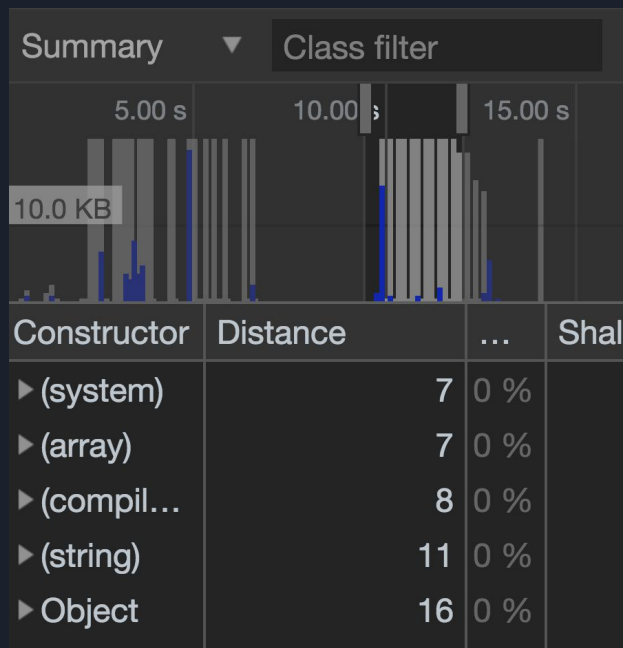
- позволяет изолировать утечки памяти

Показывает объекты
выделенные за интервал
времени и все еще живые к
окончанию записи.

☒ Allocation instrumentation on timeline

Allocation timelines show instrumented JavaScript memory allocations over time. Once profile is recorded you can select a time interval to see objects that were allocated within it and still alive by the end of recording. Use this profile type to isolate memory leaks.

☐ Record allocation stacks (extra performance overhead)





Промежуточные итоги

- CPU profiler показывает, сколько занимает вызов каждой функции в вашей программе
- Allocation sampling показывает, сколько памяти выделяет вызов каждой функции в вашей программе
- Allocation instrumentation показывает выделенные и все еще живые объекты

Далее:

- IO'18 Eager Evaluation
- IO'18 Step Into Worker

IO'18 eager evaluation

```
> [1n, 2n, 3n].map(v => v ** 1n)  
< (3) [1n, 2n, 3n]
```

▼

Filter

All levels ▼

☒ Group similar

1 hidden



☐ Hide network

☐ Log XMLHttpRequests

☐ Preserve log

☒ Eager evaluation

☐ Selected context only

☒ Autocomplete from history



IO'18 eager evaluation

```
> [1n,2n,3n].map(v => v ** 1n)
< (3) [1n, 2n, 3n]
```

- консоль показывает вам результат выражения на лету
- доступно в последнем Google Chrome Canary

```
while(true){} // очень долгий JS
window.location.href = '...' // очень опасный JS
```

Побочные эффекты: начало

```
function changeWindow() {  
    window.a = true;  
}
```

```
changeWindow();
```

перед вызовом любой функции

получить код функции

```
B(StackCheck),  
B(LdaGlobal),  
B(Star),  
B(LdaTrue),  
B(StaNamedProperty),  
B(LdaUndefined),  
B(Return),
```

если код содержит опасный байткод

Побочные эффекты: начало

```
function changeWindow() {  
    window.a = true;  
}
```

```
changeWindow();
```

перед вызовом любой функции

получить код функции

```
B(StackCheck),  
B(LdaGlobal),  
B(Star),  
B(LdaTrue),  
B(StaNamedProperty),  
B(LdaUndefined),  
B(Return),
```

если код содержит опасный байткод





Побочные эффекты: возвращение

```
function storeTime(object) {  
    object.time = Date.now();  
}  
  
storeTime(window);    // небезопасно  
storeTime({time: 0}); // ???
```



Побочные эффекты: возвращение

```
function changeObject(obj) {  
  obj.a = 239;  
}
```

```
changeObject({a: 42});
```

перед вызовом любой функции

получить код функции

```
B(StackCheck),  
B(LdaGlobal),  
B(Star),  
B(LdaTrue),  
B(StaNamedProperty),  
B(LdaUndefined),  
B(Return),
```

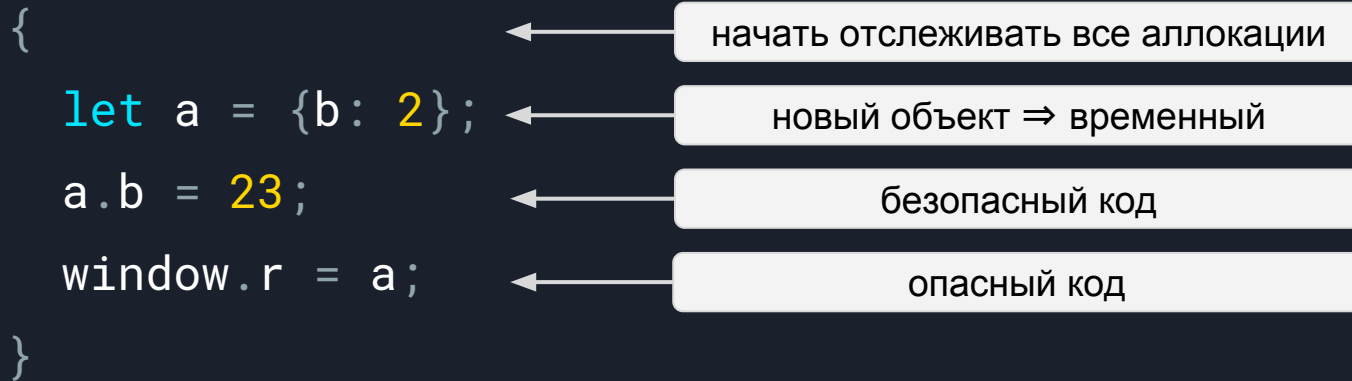
пропатчить опасный код

```
B(BoomIfNonTemporary),  
B(StaNamedProperty),
```

Побочные эффекты: недостающий кусок

Как мы помечаем временные объекты?

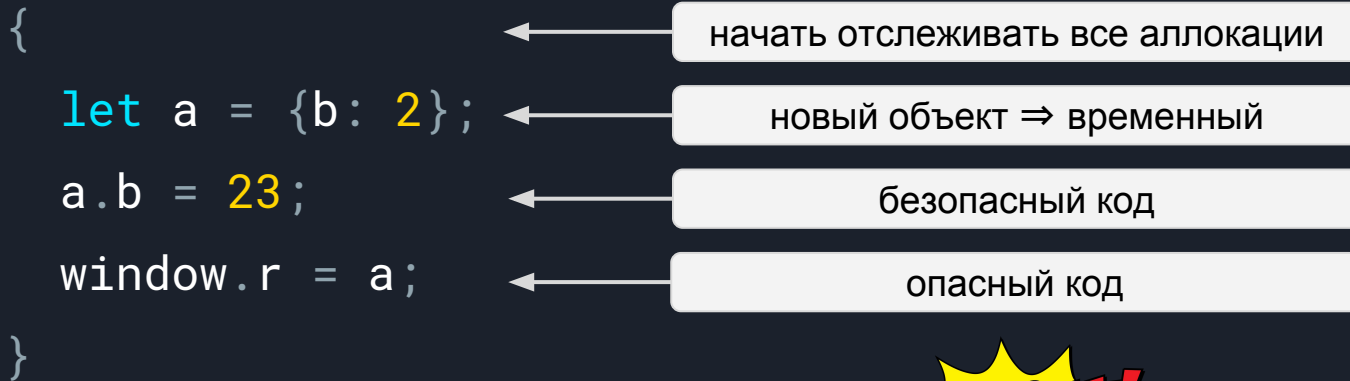
Временный объект - объект, созданный во время выполнения кода в консоли и нигде не сохраненный.



Побочные эффекты: недостающий кусок

Как мы помечаем временные объекты?

Временный объект - объект, созданный во время выполнения кода в консоли и нигде не сохраненный.





Побочные эффекты: итоги

- дополняем все опасные участки кода функции дополнительной проверкой
- проверка разрешает опасные операции на временных объектах
- не просто выполняем ваш код, а гарантируем, что это безопасно

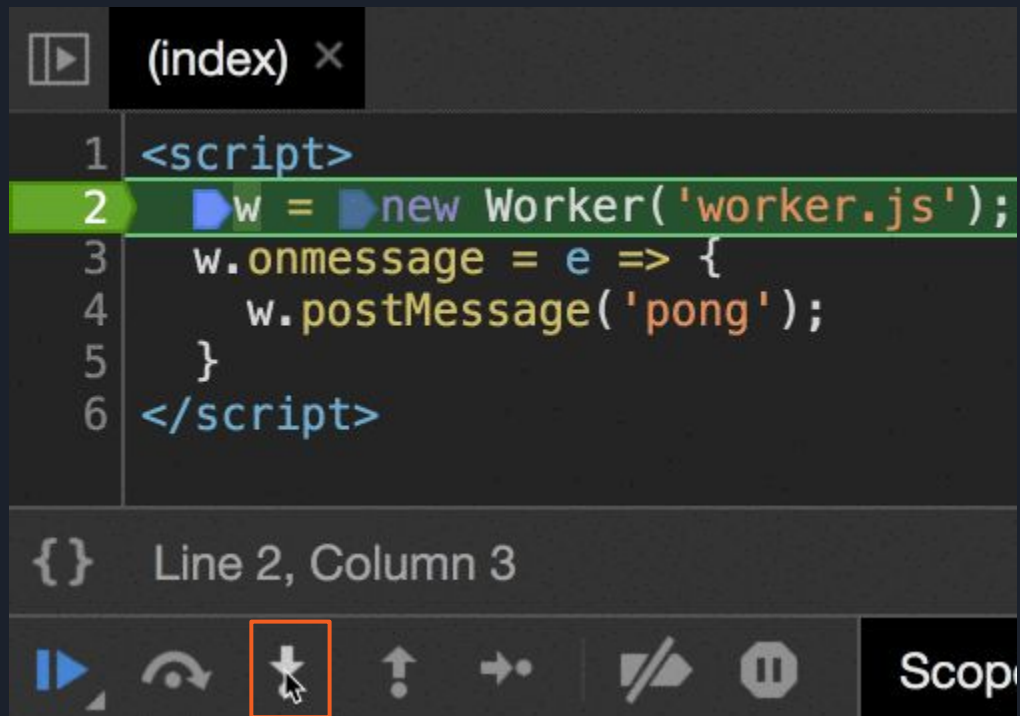
```
new Set([1, 2, 3]).delete(2)
```



Eager evaluation - ваш друг

- пробуйте разные новые языковые фичи, отлаживайте регулярные выражения, экспериментируйте в консоли
- новый режим работы V8 гарантирует отсутствие побочных эффектов

IO'18 step-into & workers = ❤️

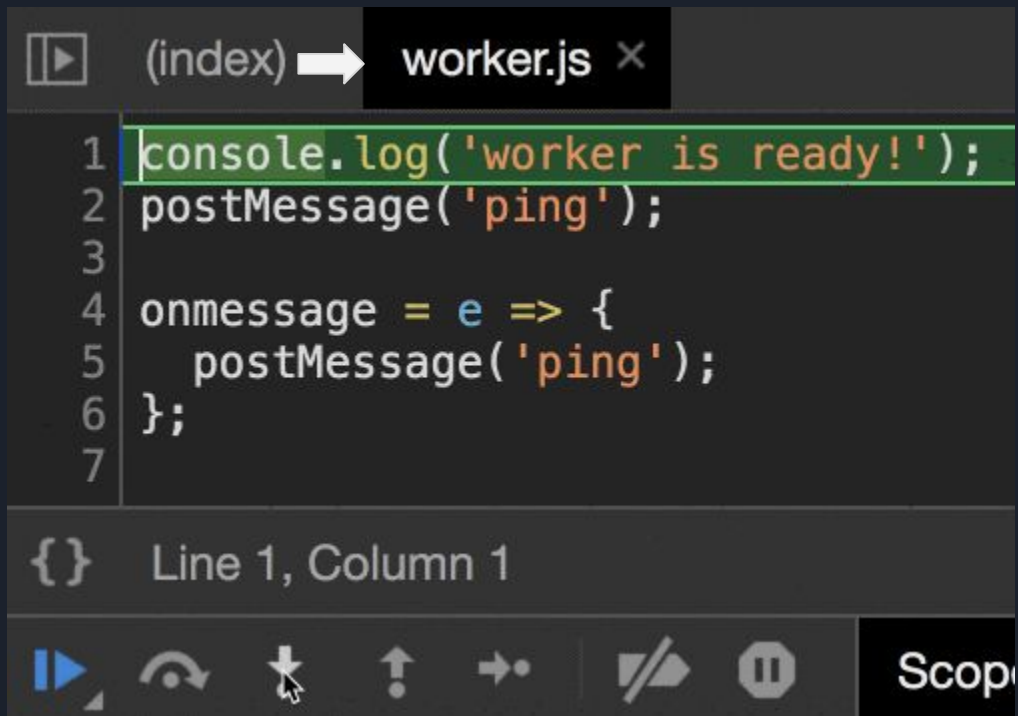


```
(index) x
1 <script>
2 w = new Worker('worker.js');
3   w.onmessage = e => {
4     w.postMessage('pong');
5   }
6 </script>
```

{ } Line 2, Column 3

Step into button circled in the toolbar.

IO'18 step-into & workers = ❤️

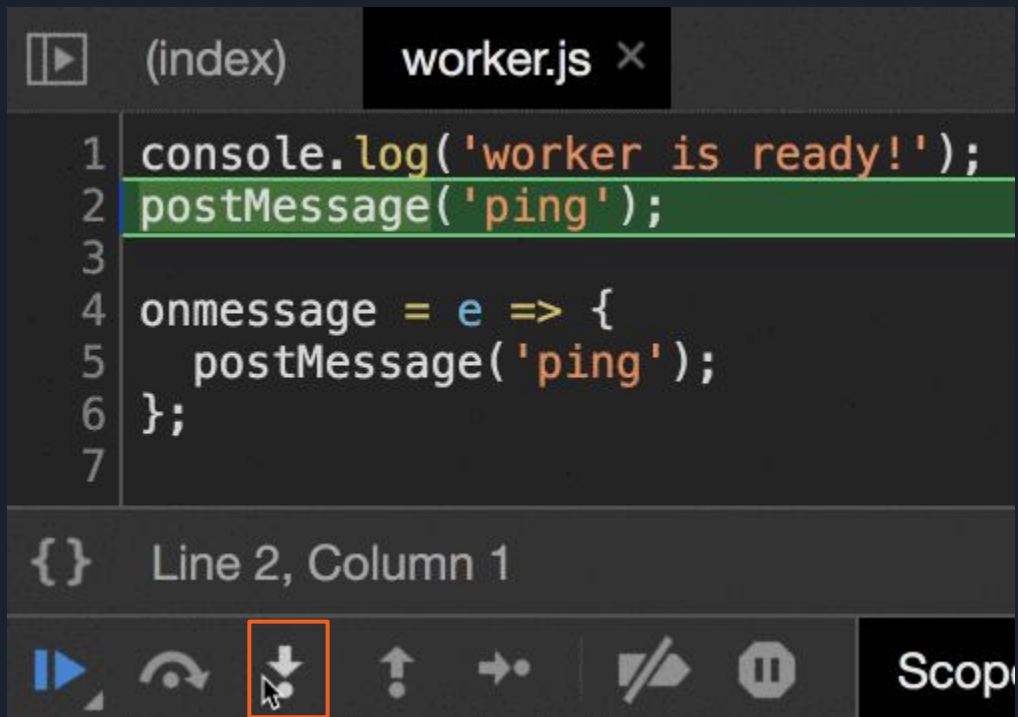


The screenshot shows a web browser's developer console with a worker script loaded. The tab is labeled '(index)' with a right-pointing arrow and 'worker.js' with a close button. The script content is as follows:

```
1 console.log('worker is ready!');
2 postMessage('ping');
3
4 onmessage = e => {
5   postMessage('ping');
6 };
7
```

The first line is highlighted in green. The status bar at the bottom indicates '{ } Line 1, Column 1'. The bottom toolbar contains icons for stepping through code (next, previous, step into, step out, step over) and a 'Scope' panel.

IO'18 step-into & workers = ❤️

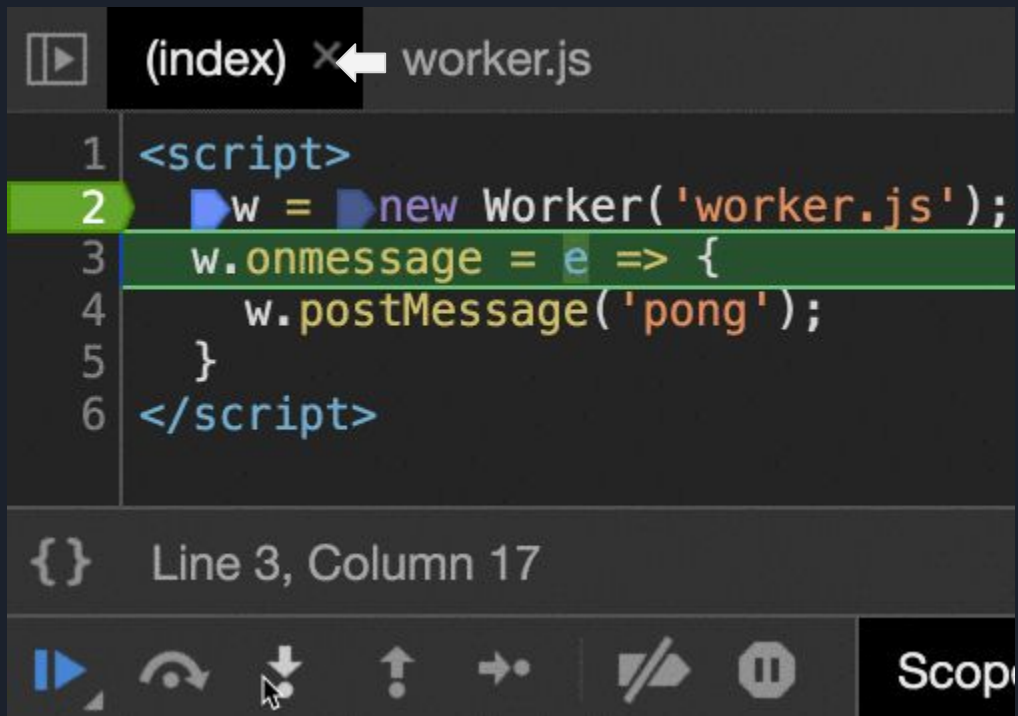


```
(index) worker.js x
1 console.log('worker is ready!');
2 postMessage('ping');
3
4 onmessage = e => {
5   postMessage('ping');
6 };
7
```

{ } Line 2, Column 1

Step into icon highlighted in red box

IO'18 step-into & workers = ❤️

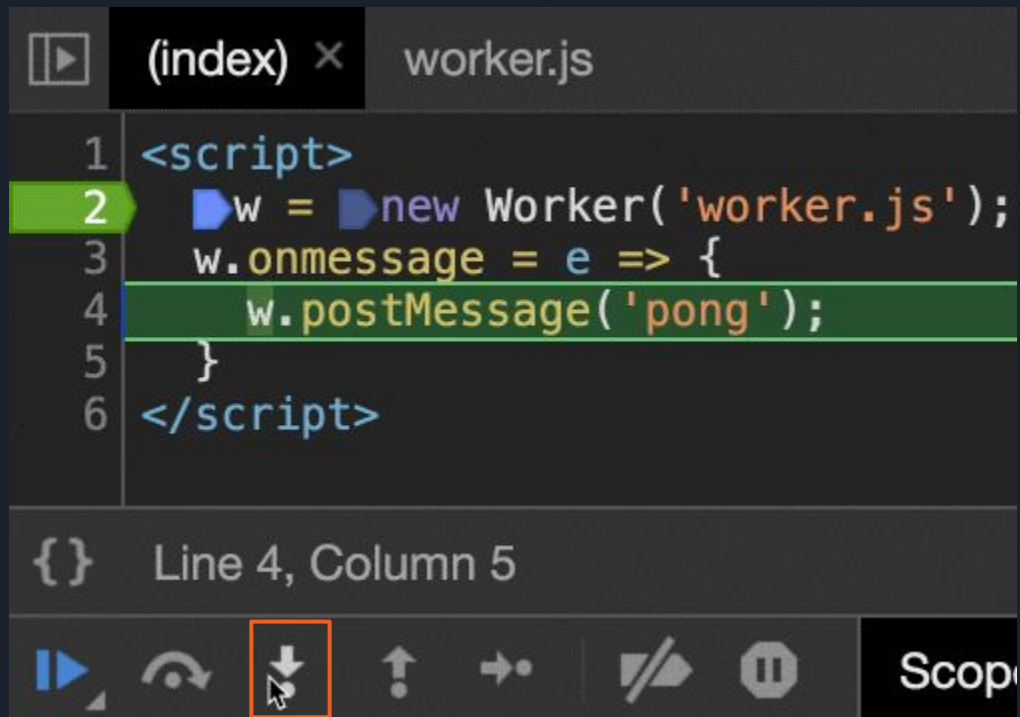


The screenshot shows a web browser's developer console with a tab for 'worker.js'. The code is as follows:

```
1 <script>
2 w = new Worker('worker.js');
3 w.onmessage = e => {
4   w.postMessage('pong');
5 }
6 </script>
```

The cursor is positioned at the end of line 3, column 17. The status bar at the bottom indicates 'Line 3, Column 17'. The bottom toolbar includes icons for stepping through code (next, previous, first, last, step into, step out) and a 'Scope' panel.

IO'18 step-into & workers = ❤️

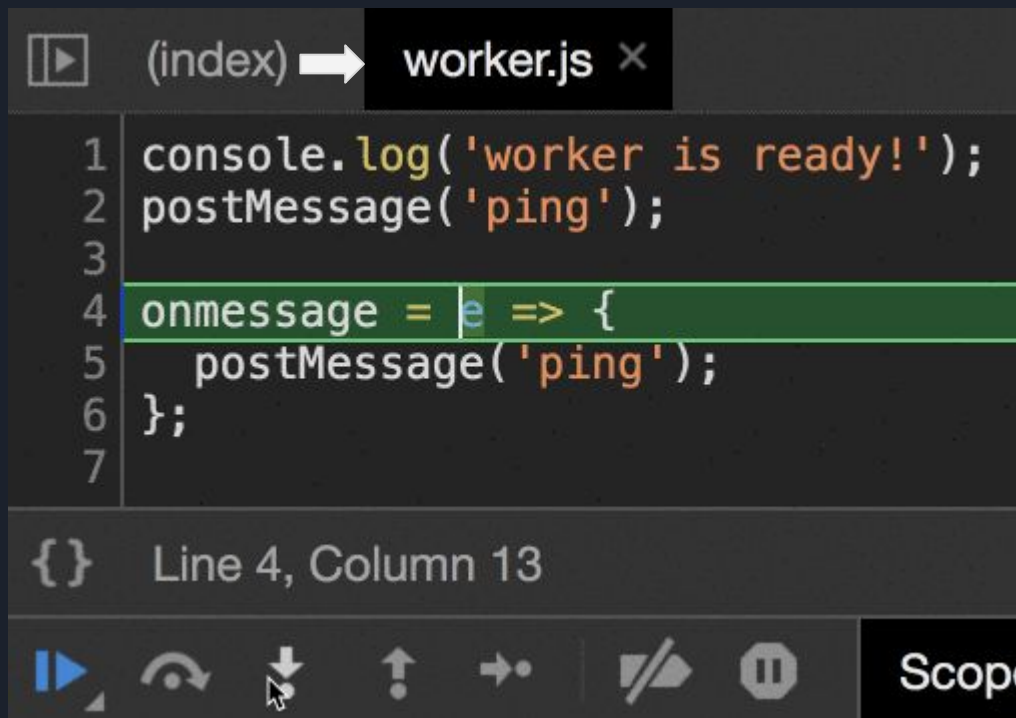


```
(index) × worker.js
1 <script>
2 w = new Worker('worker.js');
3 w.onmessage = e => {
4   w.postMessage('pong');
5 }
6 </script>
```

{ } Line 4, Column 5

Step Into

IO'18 step-into & workers = ❤️



The screenshot shows a web browser's developer console with a worker script named 'worker.js' open. The script contains the following code:

```
1 console.log('worker is ready!');
2 postMessage('ping');
3
4 onmessage = e => {
5   postMessage('ping');
6 };
7
```

The line `onmessage = e => {` is highlighted in green. The console status bar at the bottom indicates the cursor is at 'Line 4, Column 13'. The bottom of the console shows various debugging icons (run, step-into, step-over, step-out, etc.) and a 'Scope' panel.



IO'18 step-into & workers: new Worker

страничка

```
new Worker('worker.js');
```

сообщить **токен**
остановиться...

DevTools

worker

IO'18 step-into & workers: new Worker

страничка

```
new Worker('worker.js');
```

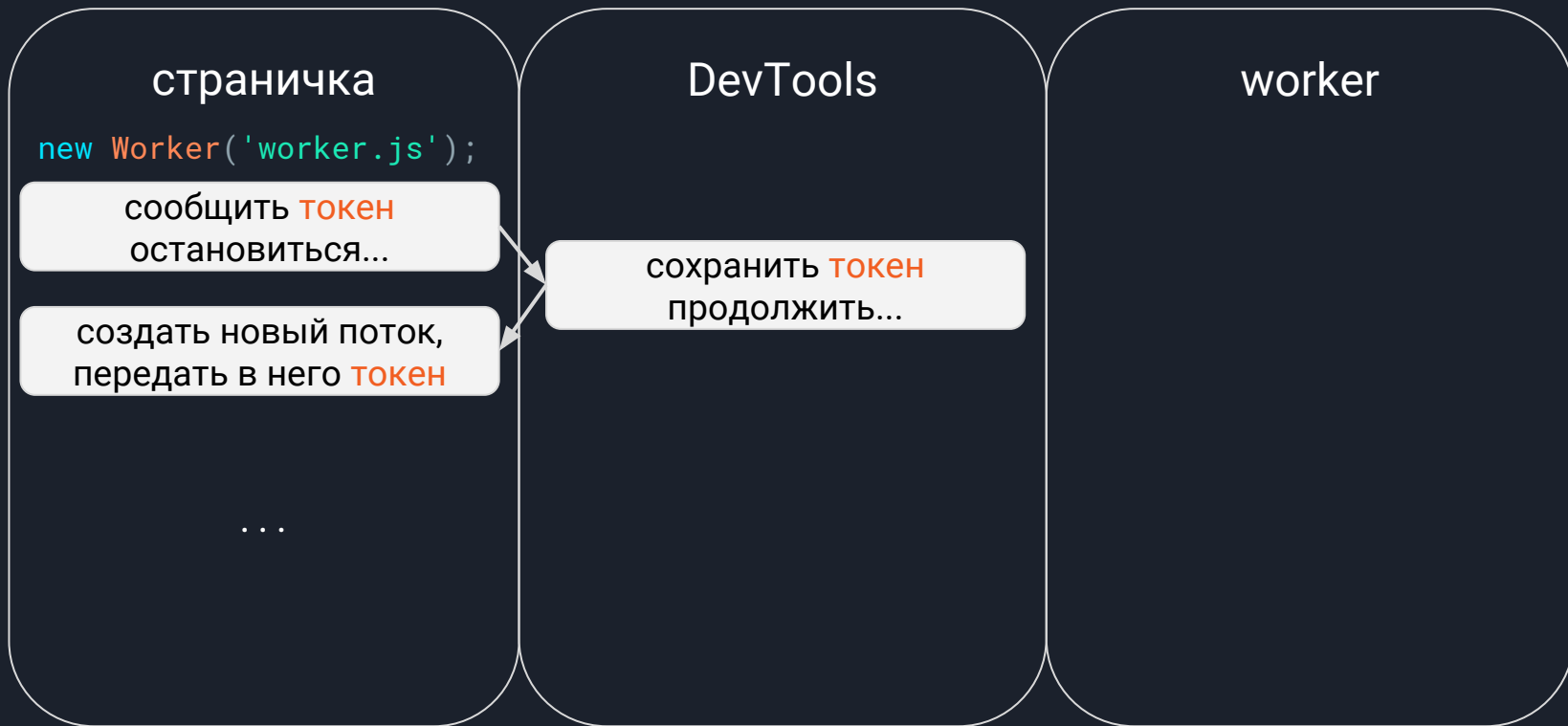
сообщить **токен**
остановиться...

DevTools

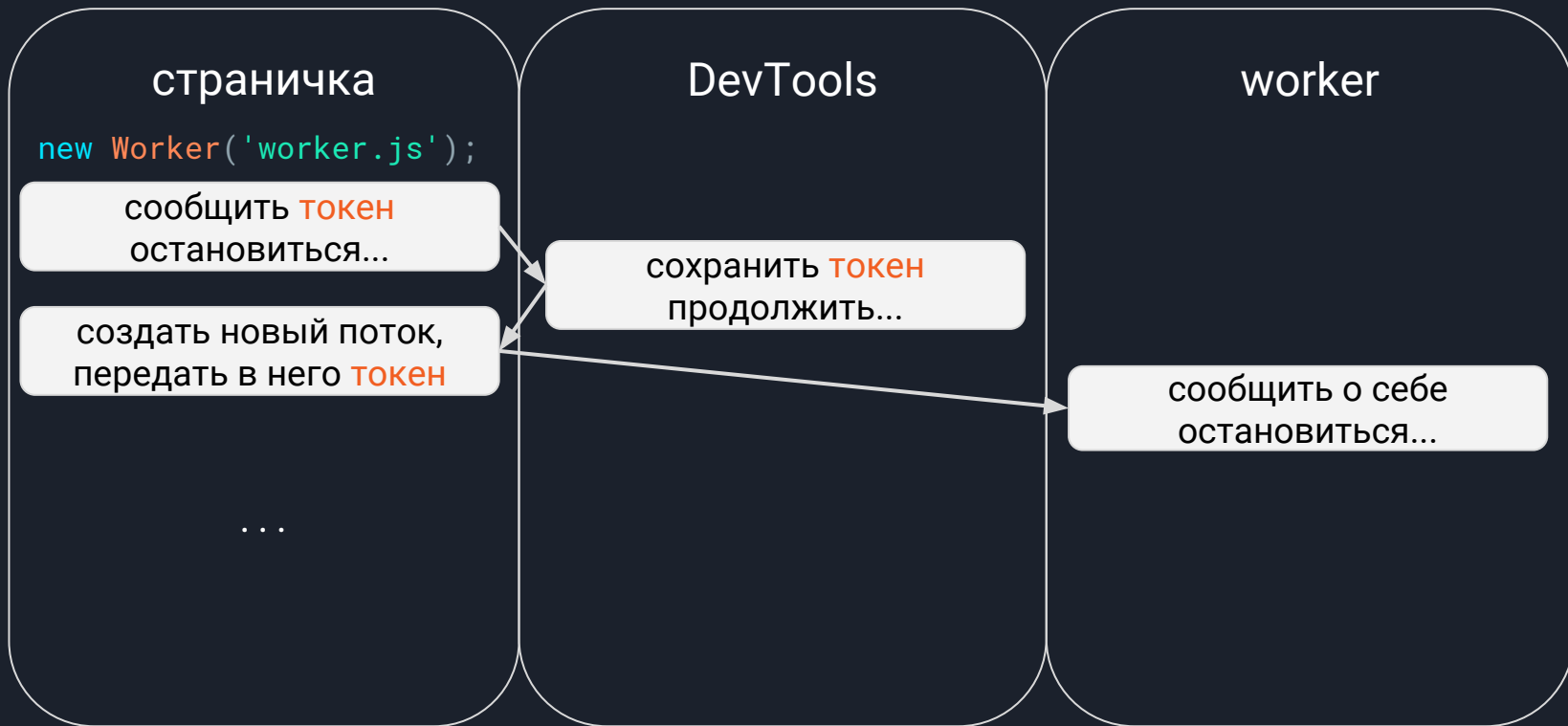
сохранить **токен**
продолжить...

worker

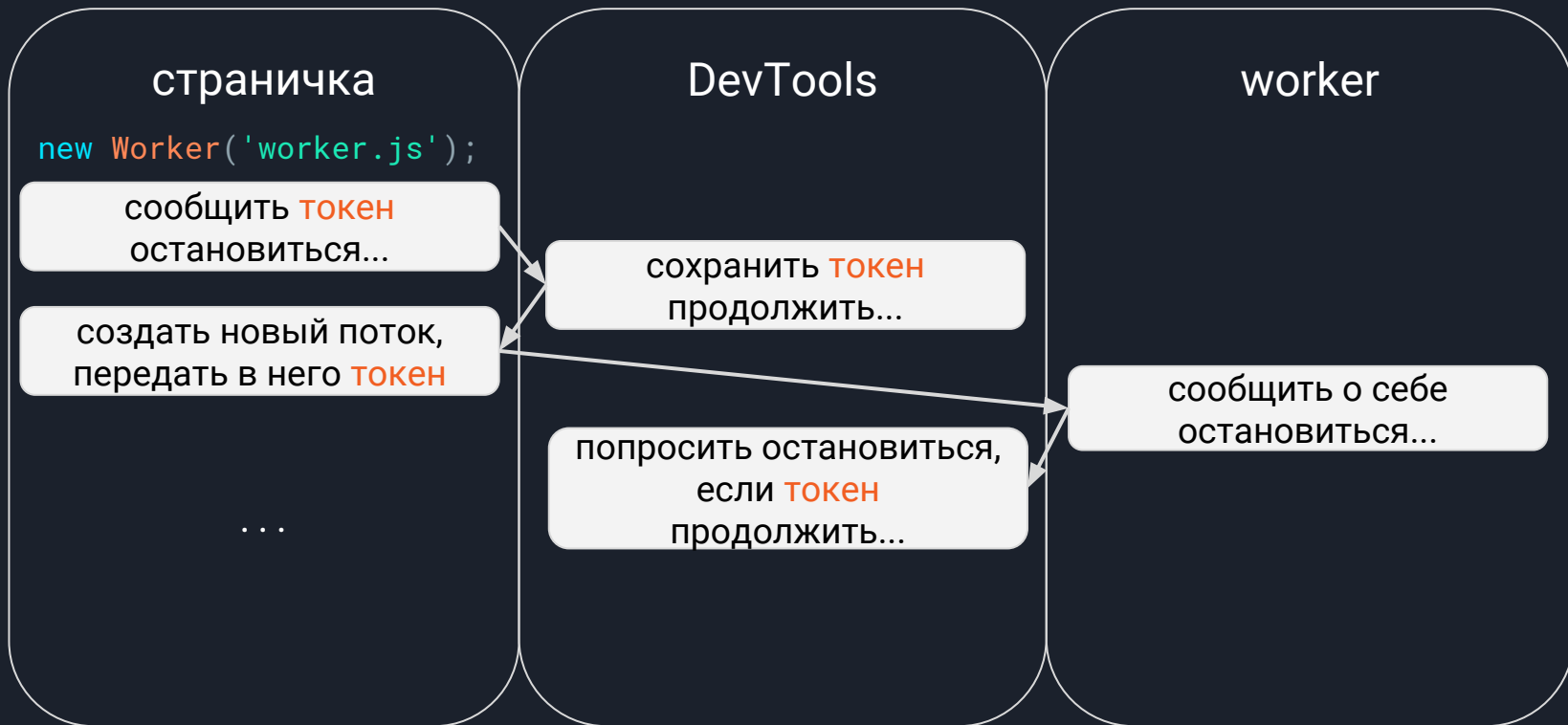
IO'18 step-into & workers: new Worker



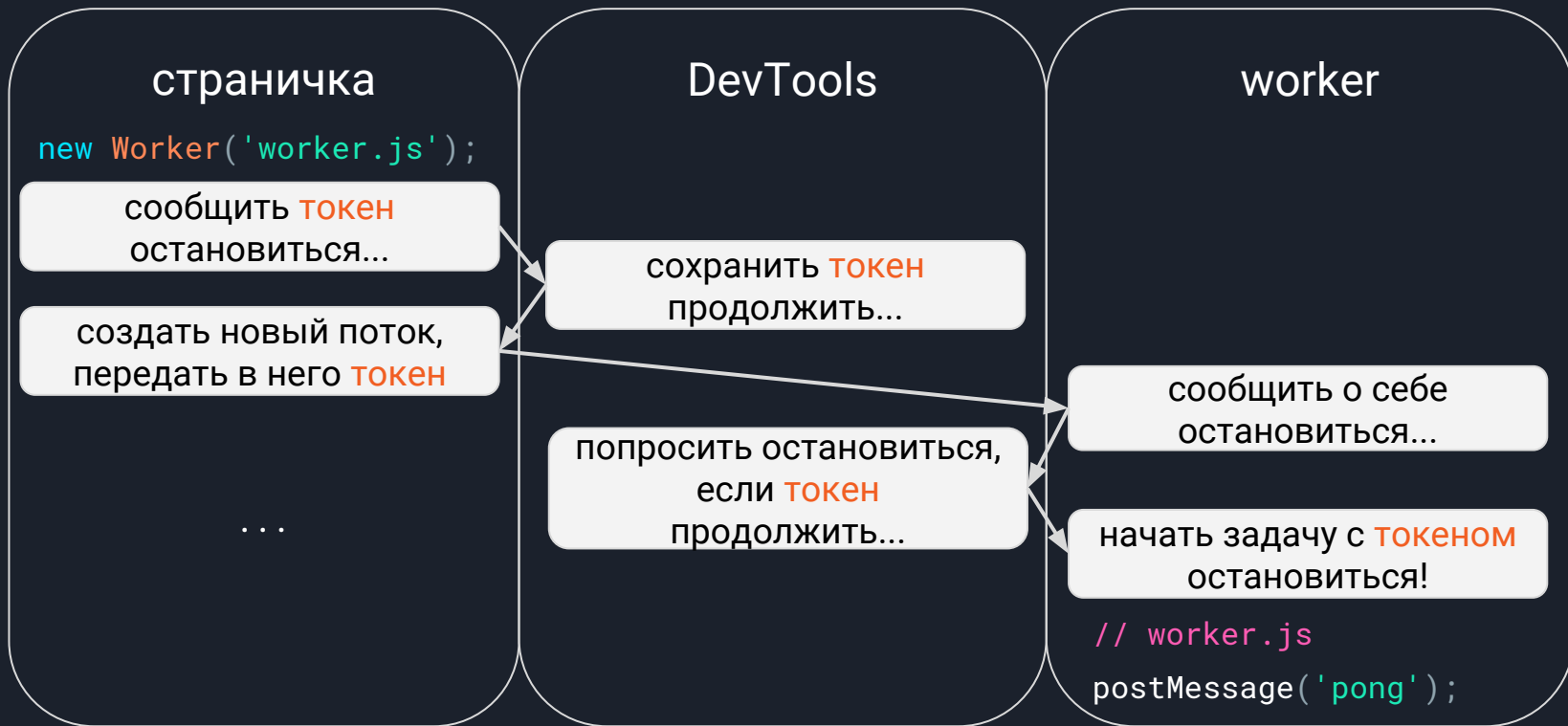
IO'18 step-into & workers: new Worker



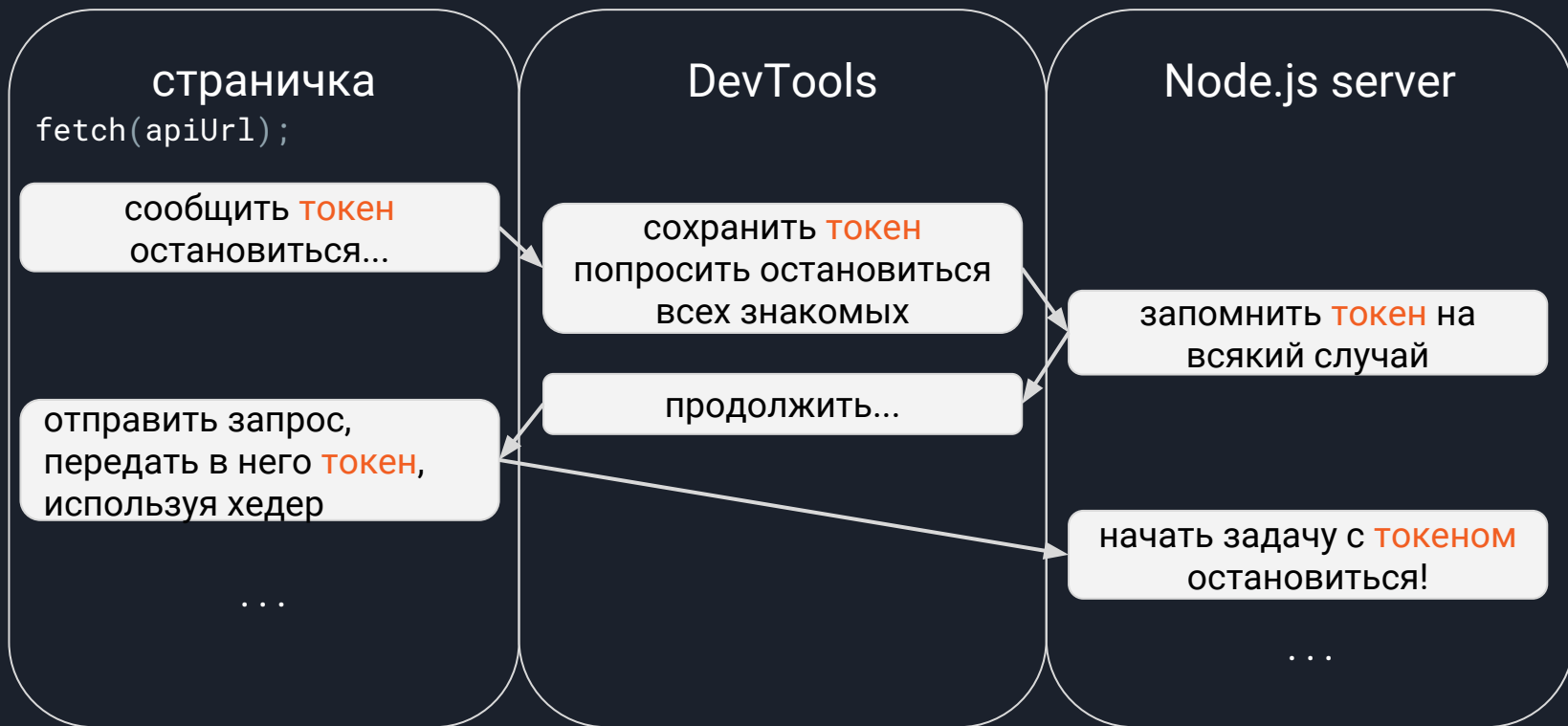
IO'18 step-into & workers: new Worker



IO'18 step-into & workers: new Worker



IO'18 step-into: универсальная схема





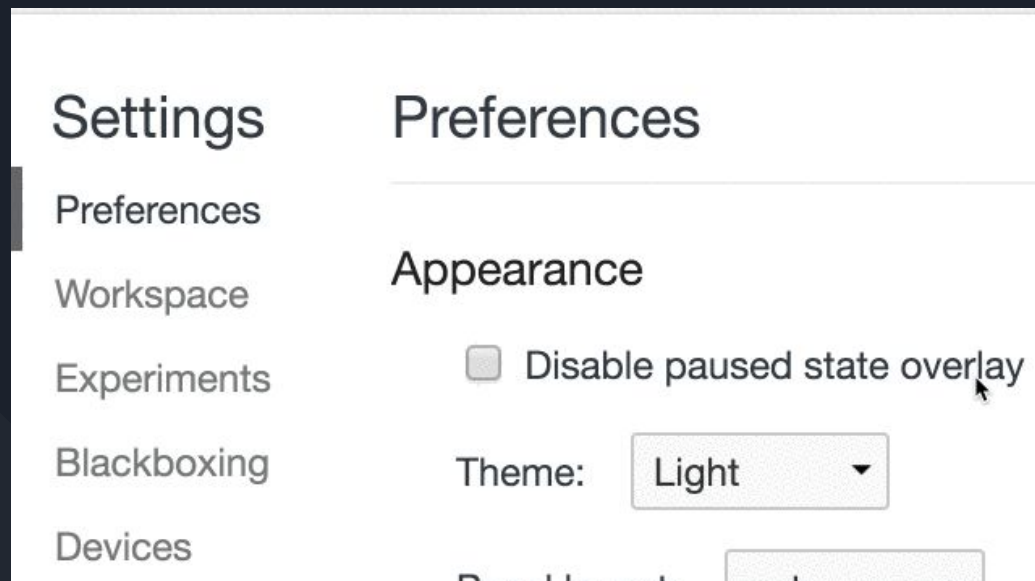
IO'18 step-into worker - ваш друг

- используйте step-into для перехода из воркера в страничку и обратно
- используйте step-into для перехода в скрипт воркера из конструктора
- используйте step-into для асинхронных вызовов



Всем спасибо!

Сцена после титров





console.assert(true, ...)

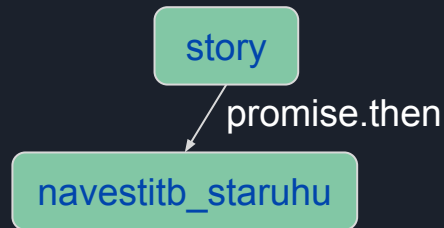
```
function division(a,b) {  
    console.assert(b !== 0, 'на ноль нельзя');  
    return a / b;  
}  
  
// ...  
  
var assert = console.assert;  
console.assert = function(a, b, c, d) {  
    if (a) return;  
    assert(a, b, c, d);  
}
```

Асинхронные стеки

```
function navestit_staruhu() {
  primenitb_topor();
  setTimeout(arest_Mikoly, SOME_TIME1);
  rodion_raskaylsya.then(priznatbsya);
}

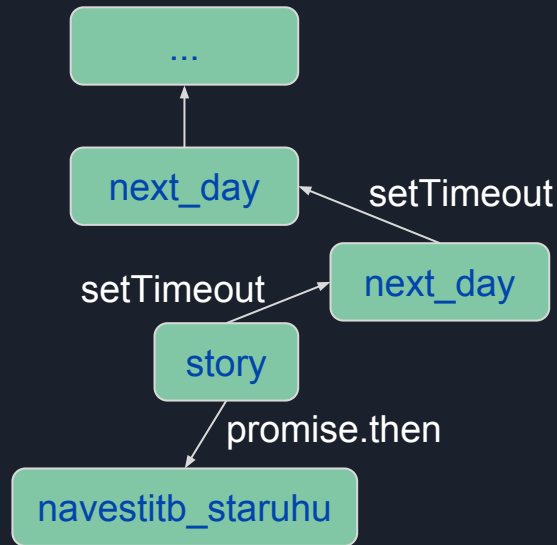
function priznatbsya() {
  setTimeout(uehatb_v_Sibire, SOME_TIME2);
}

function story() {
  rodion_poluchil_pismo.then(navestitbStaruhu);
  function next_day() {
    update_world_state();
    setTimeout(next_day, ONE_DAY);
  }
  next_day();
}
```



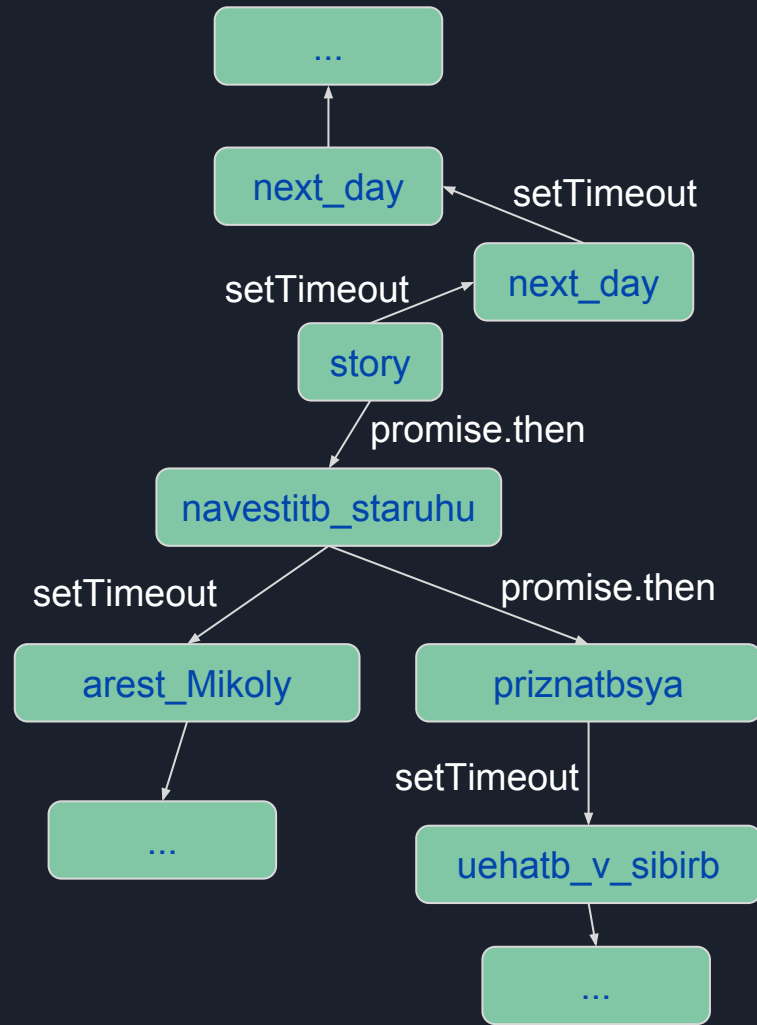
Асинхронные стеки

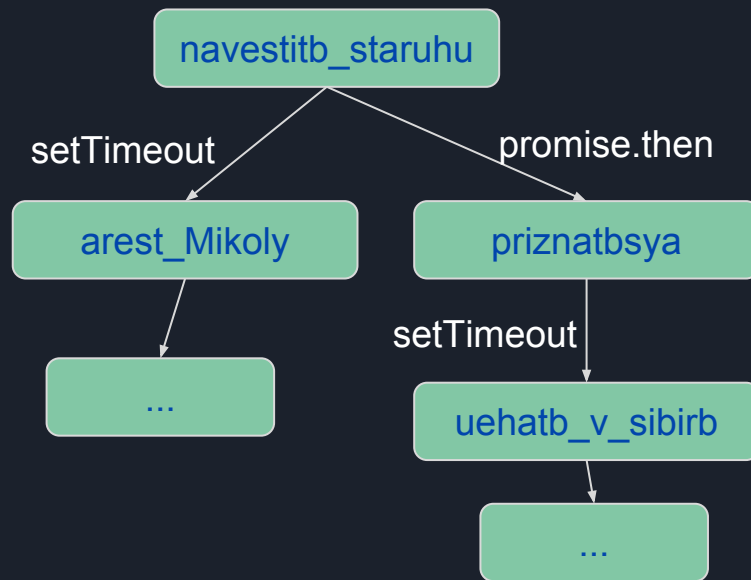
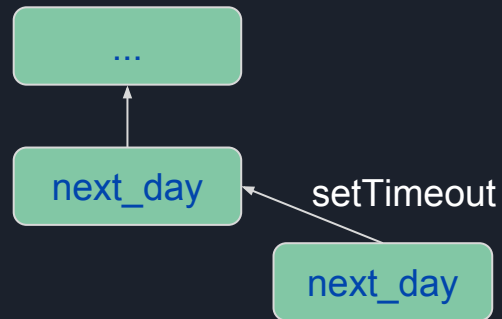
```
function navestit_staruhu() {  
    primenitb_topor();  
    setTimeout(arest_Mikoly, SOME_TIME1);  
    rodion_raskaylsya.then(priznatbsya);  
}  
  
function priznatbsya() {  
    setTimeout(uehatb_v_Sibire, SOME_TIME2);  
}  
  
function story() {  
    rodion_poluchil_pismo.then(navestitbStaruhu);  
    function next_day() {  
        update_world_state();  
        setTimeout(next_day, ONE_DAY);  
    }  
    next_day();  
}
```

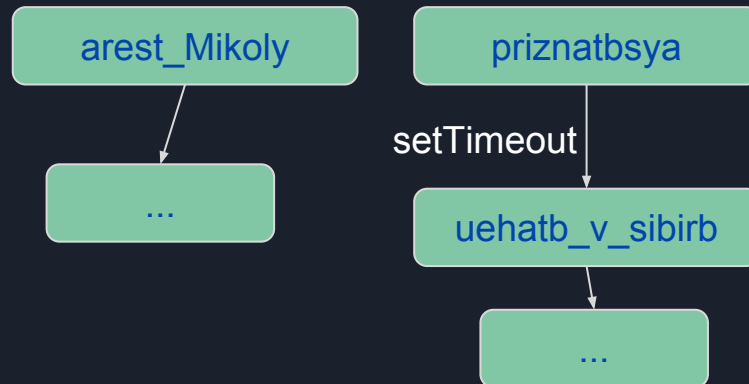
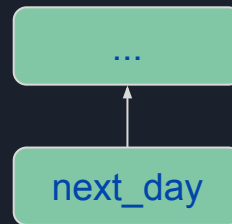


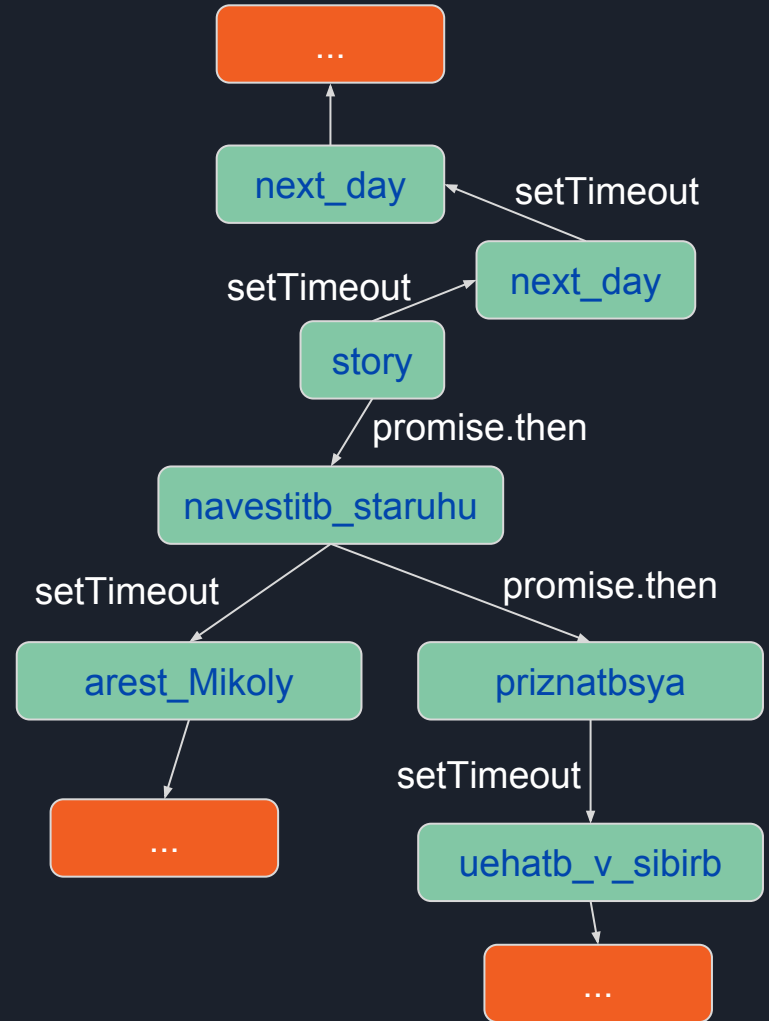
Асинхронные стеки

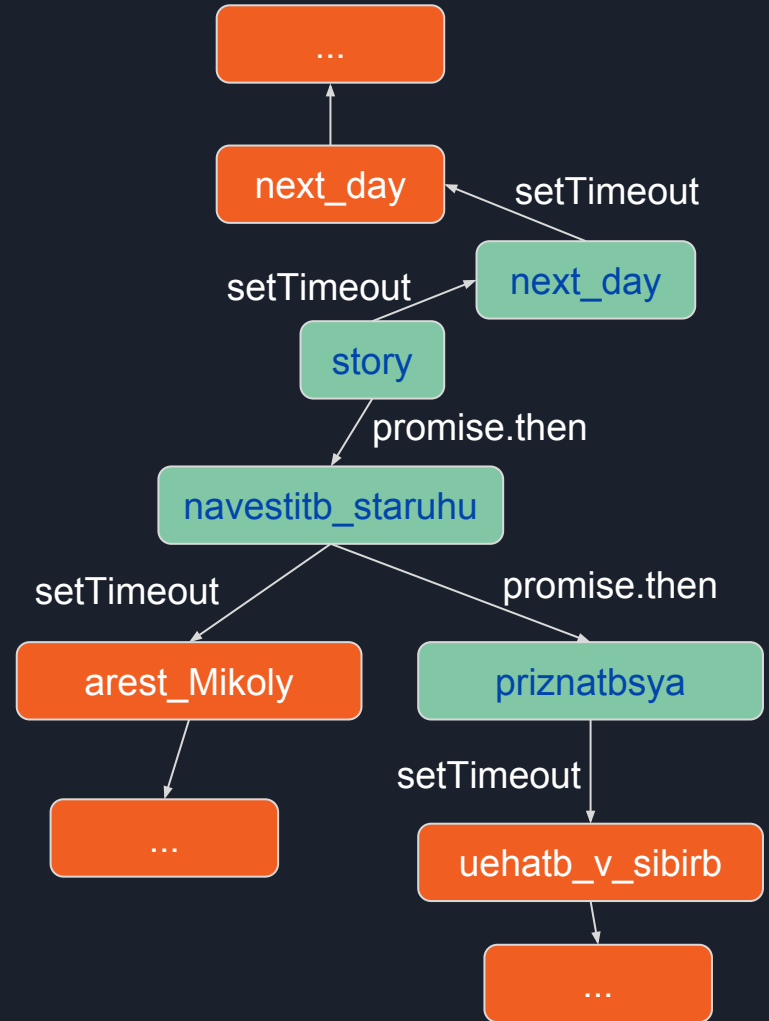
```
function navestit_staruhu() {  
    primenitb_topor();  
    setTimeout(arest_Mikoly, SOME_TIME1);  
    rodion_raskaylsya.then(priznatbsya);  
}  
  
function priznatbsya() {  
    setTimeout(uehatb_v_Sibire, SOME_TIME2);  
}  
  
function story() {  
    rodion_poluchil_pismo.then(navestitbStaruhu);  
    function next_day() {  
        update_world_state();  
        setTimeout(next_day, ONE_DAY);  
    }  
    next_day();  
}
```

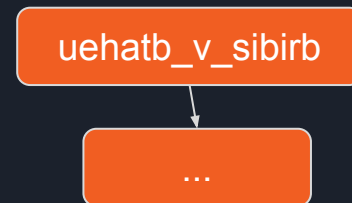
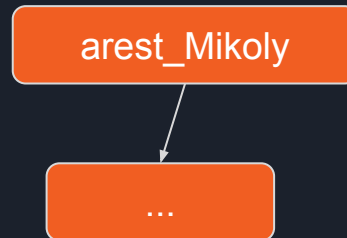














Инструментирование VS сэмплирование

- инструментирование
 - всегда включает все функции и содержит очень точную информацию о том, какие функции вызывались
 - неравномерно искажает результат
 - сложнее в реализации
- сэмплирование
 - равномерно искажает результат
 - точность ограничена частотой сэмплирования



```
function leakA() {  
  window.leak = new A();  
}  
leakA();
```

СНИМОК ХИПА ДО ВЫЗОВА ФУНКЦИИ



window

СНИМОК ХИПА ПОСЛЕ ВЫЗОВА ФУНКЦИИ



window

A



Heap snapshot

Heap snapshot profiles show me


```
function getValue() {  
  if (!window.cache)  
    window.cache = new A();  
  return window.cache;  
}
```

...

```
getValue();
```

...

```
delete window.cache;
```

СНИМОК ХИПА ДО ВЫЗОВА ФУНКЦИИ

window

СНИМОК ХИПА ПОСЛЕ ВЫЗОВА ФУНКЦИИ

window

A

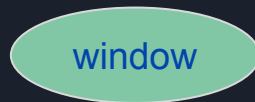
СНИМОК ХИПА ПОСЛЕ ОЧИСТКИ КЭША

window

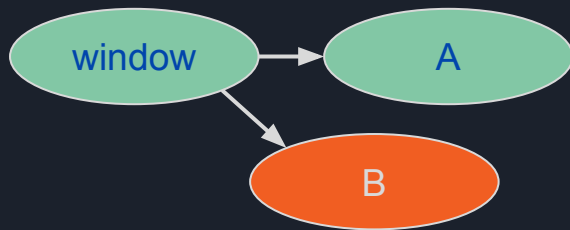
Summary ▼ Class filter All objects ▼

```
function getValue() {  
  if (!window.cache)  
    window.cache = new A();  
  window.leak = new B();  
  return window.cache;  
}  
...  
getValue();  
...  
delete window.cache;
```

снимок хипа до вызова функции



снимок хипа после вызова функции



снимок хипа после очистки кэша



снимок хипа до вызова функции

window

снимок хипа после вызова функции

window

A

B

снимок хипа после очистки кэша

window

B

разница до и после

B

A

снимок хипа до вызова функции

window

снимок хипа после вызова функции

window

A

B

снимок хипа после очистки кэша

window

B

разница до и после

B

A

утекшая память

B

