Асинхронно изпълнение на JavaScript код

Функциите на JavaScript се изпълняват в рамките на един и същ контекст, по подразбиране асинхронно. Това означава, че не е възможно конкурентно изпълнение на техния код. Трябва да се прави разлика между *конкурентен* и *асинхронен* програмен код. *Конкурентен* е кодът, който позволява отделни части от приложението да се изпълняват в един и същ момент от време (паралелно или в режим на деление на ресурсите на микропроцесора). *Асинхронен* е кодът, който позволява отделни части от приложението да се изпълняват непоследователно във времето – не се изисква изчакване на един модул да завърши, за да се *подготви* за изпълнение друг модул. Конкурентният код по своята същност е асинхронен, но обратното не винаги е вярно.

При езиците от високо ниво, например Java, C++ и C#, конкурентността на кода се реализира най-често чрез използване на *програмни нишки* ($threads$). Програмната нишка е функция, която се „вижда" от операционната система като отделен процес и следователно тя има свой собствен контекст. Това позволява нишката да се изпълнява конкурентно на останалите процеси. Всяка нишка има свой собствен стек и програмен приоритет. При JavaScript, програмният код се изпълнява в рамките на *една* програмна нишка. Това е причината този език да не поддържа конкурентен код. Разбира се, трябва да има основателни причини JavaScript да е *еднонишков* (single-threaded) програмен език. При многонишковото програмиране са необходими много добри познания за съответния език, за да се пише безопасен (*thread-safe*) програмен код. В многонишкова среда трябва да се внимава с използването на *споделените* ресурси (променливи, обекти и функции, които се достъпват от множество нишки едновременно), както и възможността за получаване на така наречената мъртва хватка (dead lock) – взаимно блокиране на няколко нишки, работещи с общи споделени ресурси. Всичко това налага програмистът да има задълбочени знания в областта на синхронизиране на функционирането на множество програмни нишки в една многонишкова среда.

Всички тези проблеми не съществуват в JavaScript благодарение на това, че този език използва само една нишка. Така лесно се гарантира писането на *неблокиращ* програмен код и използване на по-малко ресурси. Един JavaScript модул може да започне да се изпълнява, ако няма друг, който преди него очаква изпълнение. Ако текущо изпълняваният програмен модул консумира дълго време ресурсите на процесора, това ще доведе до блокиране на потребителския интерфейс, а може и да наруши логиката на изпълнение на останалия код. Дори процесорът да е многоядрен, кодът от отделните JavaScript модули няма да могат да се изпълняват паралелно. Това е характерно за *синхронния* програмен код. Но JavaScript поддържа асинхронно изпълнение на кода, по точно *асинхронни функции*. Това най-често са функции, които трябва да се извикат при възникване

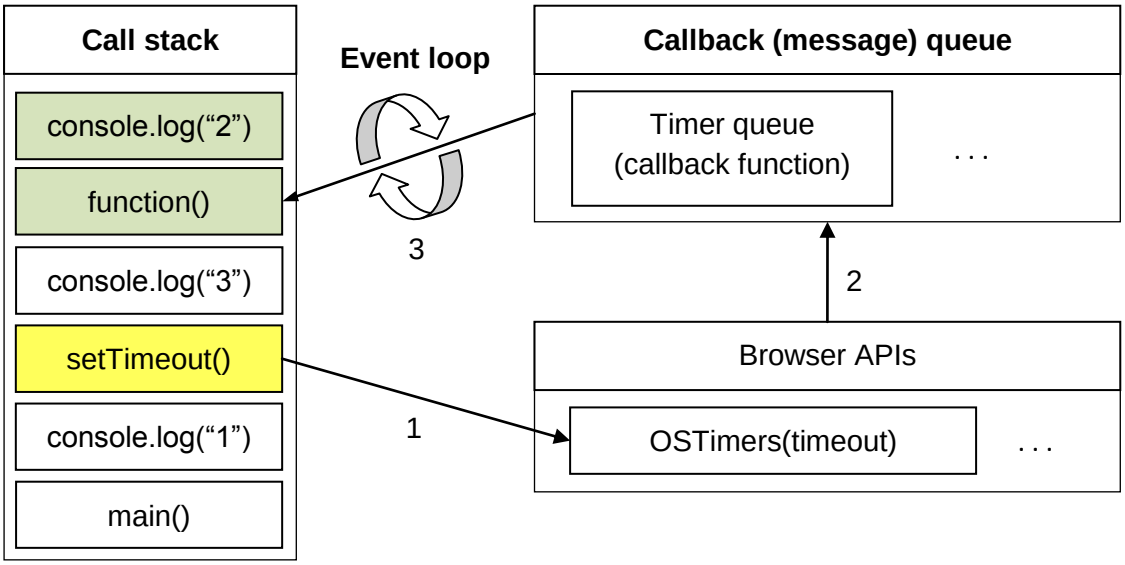
на определено събитие, например: изминал е зададен интервал от време, натиснат или отпуснат е бутон от клавиатурата или мишката, заредена в паметта е картинка или HTML страница, начало на въвеждане на информация в текстово поле, получаване или губене на фокус и много други. Функциите, които се стартират по условие, се наричат $callback$ функции. Тези функции се изпълняват асинхронно, но не конкурентно на кода от останалите функции. Типичен пример за асинхронни функции в JavaScript са тези, които използват таймери - $setTimeout$, $setInterval$ и $requestAnimationFrame$. Проблемът е, че основният JavaScript код и кодът от таймерите се изпълняват в един и същ контекст (на основната нишка) и следователно за истинско конкурентно изпълнение не може да се говори, но кодът на таймерите се изпълнява *асинхронно* на основния код.

```
1  var закъснение = 1000;
2  console.log("1");
3  setTimeout(function() {
4      console.log("2");
5      }, закъснение);
6  console.log("3");
```

Фиг. 1. Пример за използване на $callback$ функция

На Фиг.1 е показан програмен код, който използва функция $setTimeout$. Тя изисква два аргумента: 1) Асинхронна $callback$ функция (в случая анонимна) и 2) време закъснение, след което се вика кода на $callback$ функцията. В конкретният случай анонимната функция ще се извика след 1000 ms (1 секунда) след като се извика функция $setTimeout$. Ако JavaScript е синхронен програмен език, при изпълнение на този код, трябва на конзолата да се разпечата 1, да се изчака една секунда и да се разпечата 2 и веднага след това да се разпечата 3. На практика на конзолата се печата 1, веднага след това 3 и след една секунда 2. Това е доказателство, че JavaScript наистина е асинхронен език. Въпросът е точно кога се изпълнява анонимната $callback$ функция? Ако $callback$ функцията се вика *веднага*, след като измине зададения времеви интервал, то текущо изпълняваната функция трябва да се прекъсне от $callback$ функцията. Това обаче не е възможно при JavaScript, тъй като основната функция и $callback$ функцията се изпълняват от една нишка. За да докажем това ще зададем закъснение от 0 ms ($закъснение = 0$). В този случай би трябвало на конзолата да се разпечата 1, след това 2 (няма закъснение) и най-накрая – 3. На практика отново се печата първо 1, след това 3 и накрая 2. Това означава, че с по-висок приоритет е основната функция спрямо $callback$ функцията. Кодът на $callback$ функцията се вика, ако основната функция е завършила своето изпълнение.

За да се разбере как е възможно в една неконкурентна и еднонишкова среда да се поддържат асинхронни функции, трябва да разгледа по-подробно „конкурентния" модел на JavaScript. Той се базира на *call stack*, *message queue* и *event loop*.



Фиг. 2. Пример за работа на JavaScript при изпълнение на асинхронен код

На Фиг. 2 е показано как точно се реализира изпълнението на програмния код, показан на Фиг.1.

Call stack

Това е област от паметта с достъп Last Input First Output (LIFO), която JavaScript използва с цел да знае коя е текущо изпълняваната функция и коя ще е следващата. В стека се записват съобщения, които описват коя функция трябва да се стартира. За всяка функция, която се вика, първо се записва съобщение за нея във върха на стека. Един запис в стека се нарича рамка (frame). Интерпретаторът на JavaScript започва да изпълнява тази функция, която е записана в стека последна. Например, ако функция $f1$ вика $f2$, а $f2$ вика $f3$, в call stack първо се записва $f1$, след това – $f2$ и накрая - $f3$. Следователно, първо се изпълнява $f3$, след това - $f2$ и накрая се вика $f1$. Трябва да се има предвид, че този стек има краен размер. Например, при браузър Chrome размерът на call stack е 16000 рамки. Ако една функция вика себе си неограничен брой пъти, стекът ще се запълни и ще се генерира изключение от тип “Stack overflow.” Когато дадена функция върне от управление, съответното съобщение за нея се изтрива от стека. Стекът е празен, ако няма чакащи за обслужване функции. Ако JavaScript не поддържаше асинхронни функции, то този стек щеше да бъде достатъчен за функциониране на JavaScript приложенията.

Message (callback) queue

В тази структура в паметта от тип опашка с достъп First Input First Output (FIFO) се записват само съобщения (messages), които са свързани с изпълнението на $callback$ функции. Всяко ново съобщение се записва в края на опашката. Записът се реализира при детектиране на следено събитие, а изтриването – при

възможност за изпълнение на $callback$ функцията. Тази програмна опашка е прието да се нарича $message\ queue$ или $callback\ queue$. За да функционира тази идея е необходим и механизъм, наречен event loop.

Event loop

Основната задача на event loop е да изпълнява $callback$ функции в точният момент от време. Той проверява дали call stack е празен (не съдържа функции за изпълнение). Ако стекът не е празен се изчаква. Ако е празен – започва изпълнение на $callback$ функциите, които са в опашката.

Тъй като JavaScript код се изпълнява от тялото на функция, то първата функция, която попада в call stack е означена като $main$ – функцията, която съдържа вашия Javascript код. Тя вика функция log на която се предава аргумент „1”. Тъй като тази функция не вика друга функция, тя печата на конзолата цифра 1 и се премахва от стека. След това, в стека се записва функция $setTimeout$. Тази функция използва една от библиотеките на браузъра, за да стартира програмен таймер (1). От стека се изтрива функция $setTimeout$ и се записва функция log с аргумент „3”. След изтичане на зададения интервал от време, в callback queue се записва съобщение, което съдържа информация коя е $callback$ функцията която трябва да се активира (2). Модулът “event loop” проверява дали стекът е празен. Ако това не е изпълнено, $callback$ функцията от таймера изчаква функция log да разпечата на конзолата цифра 3. След това log функцията се изтрива от стека. В този момент стекът вече е празен и в него се зарежда съобщение за анонимната $callback$ функция (3). Тя вика функция log печата на екрана числото 2. Следва изтриване от стека на функция log, на анонимната функция и накрая на $main$ функцията.

Всички JavaScript функции са *неблокиращи*. За да бъде една функция необлокираща е достатъчно тя да регистрира своя $callback$ функция в message queue и веднага да бъде изтрита от call stack. По този начин, независимо колко процесорно време е необходимо за изпълнение на функцията, тя няма да блокира друга чакаща изпълнение функция. Следователно, ако желаете една ваша функция да се изпълни асинхронно, можете да я реализирате като $callback$ функция. За целта се използва функция $setTtimeout$ за която първи параметър е името на вашата функция, а втори параметър е нулево време закъснение. Когато закъснението е 0, браузърът веднага записва вашата $callback$ функцията в message queue. На Фиг. 3 е показан програмен код, който реализира тази идея.

```
1  console.log("Преди асинхронния код");
2  setTimeout(моятаФункция, 0);
3  console.log("След асинхронния код");
4  function моятаФункция() {
5      console.log("Асинхроннен код");
6  }
```

```
Преди асинхронния код
След асинхронния код
Асинхроннен код
```

Фиг. 3. Създаване на асинхронен код - моятаФункция

Нека да разгледаме програмния код от Фиг. 4. Той стартира в цикъл N на брой пъти функция $setTimeout$. За първи аргумент се предава анонимна функция, която печата на екрана номера на итерацията. Вторият аргумент задава нулево време закъснение.

```
1  var N = 10;
2  for (var i=0; i<N; i++) {
3      setTimeout(function() {
4      console.log("Итерация "+i);
5      }, 0);
6  }
```

Фиг. 4. Пример за асинхронен код в програмен цикъл

Изпълнението на този код води до разпечатване на конзолата 10 пъти на низа „Итерация 10”, въпреки, че се очаква да се разпечатат 10 низа, всеки от който съдържа думата „Итерация” и номера на итерацията от 0 до 9.

Разбира се, има начини този проблем да бъде решен. Възможно е използването на ключова дума let за дефиниране на променлива i (виж Фиг. 6, ред 2). В този случай се създава нов контекст за i при всяка итерация. Проблемът е, че този код може да се използва при ES6+.

```
1  var N = 10;
2  for (let i=0; i<N; i++) {
3      setTimeout(function() {
4      console.log("Итерация "+i);
5      }, 0);
6  }
```

Фиг. 5. Пример за асинхронен код в програмен цикъл и ключова дума let

Следва анализ на възможностите на JavaScript за създаване на приложения, използващи асинхронен код чрез: функции-генератори, promises и web workers.

Функции-генератори

При извикване на функция-генератор кодът от тялото й не се изпълнява веднага, а само се връщат обект *итератор*. В последствие (където и да е във вашия код), чрез метод $next$, приложен към обекта итератор, се изпълнява кода от тялото на функцията до достигане до израз, започващ с $yield$. Ключова дума $yield$ определя стойността, която $next$ връща. Тази стойност е обект, който има две свойства: 1) $value$ – съдържа текущо върнатия обект; и 2) $done$ – булев тип, показва дали това е последната стойност в итератора ($true$). Ако се опитате да извлечете стойност от итератора, след като сте обработили всички данни, ще получите стойност $undefined$.

Дефинирането на функция-генератор може да се реализира по два начина: 1) Използване на конструктора на клас $GeneratorFunction$; и 2) Използване на

символ звезда - $function*$. На практика по-често се използва вторият начин за дефиниране, защото е по-кратък. Синтаксисът на функциите-генератори е следния:

```
function* имеНаФункцията(параметри) {
  // тяло на функцията
}
```

Функциите генератори имат синтаксис, който съвпада със синтаксиса на нормалните функции. Броят на предаваните параметри е произволен. Може и да не се предават параметри.

Задача 1

Да се напише функция изброител на обекти, които се предават към функцията като масив.

Нека масивът да съдържа списък с имена на градове. Едно примерно решение е показано на Фиг. 7. Функцията-генератор е с име $изброител$ и е дефинирана на редове 1-5. В цикъл, чрез итерационна конструкция for, се извлича съдържанието на елемент от списъка (ред 2), който се връща чрез $yield$ (ред 3). Масивът с имената на градовете е дефиниран на ред 6. Създаването на обект итератор с име $градове$ се реализира на ред 7. След като той се създаде, можем да извличаме елементи от него по всяко време, като използваме метод $next$ (редове 9, 11 и 12).

```
1  function* изброител(списък) {
2      for (елемент of списък) {
3          yield елемент;
4      }
5  }
6  var градове_списък = ["Габрово","София","Бургас"];
7  var градове = изброител(градове_списък);
8
9  alert(градове.next().value);     // Габрово
10 // Друг JS Код
11 alert(градове.next().value);     // София
12 alert(градове.next().value);     // Бургас
```

Фиг. 6. Примерно решение на Задача 1

Задача 2

Да се промени логиката на функцията-генератор от Задача 1 така, че след като се достигне до зададен град, да започне изброяване на градове от друг списък.

Ще използваме възможността за активиране на функция-генератор чрез ключова дума $yield$, която завършва със символ *. Синтаксисът е следния:

```
yield* имеНаФункцияГенератор(параметри);
```

На Фиг. 7 е показано примерен код, който реализира Задача 2.

```
1  function* изброител(списък) {
2      for (елемент of списък) {
3          yield елемент;
4          if (елемент === име) {
5              yield* изброител(градове_списък_2);
6          }
7      }
8  }
9  var име = "София";
10 var градове_списък_1 = ["Габрово","София","Бургас"];
11 var градове_списък_2 = ["Лондон","Париж"];
12 var градове = изброител(градове_списък_1);
13
14 alert(градове.next().value);    // Габрово
15 alert(градове.next().value);    // София
16 alert(градове.next().value);    // Лондон
17 // Друг JS код
18 alert(градове.next().value);    // Париж
19 // Друг JS код
20 alert(градове.next().value);    // Бургас
21 alert(градове.next().value);    // undefined
```

Фиг. 7. Примерно решение на Задача 2

На ред 4 се реализира проверка дали е достигнато до зададен град – в конкретният случай София (ред 9). Ако това условие върне $true$ се изпълнява израза от ред 5 с което се получава нов итератор, който започва да връща обекти от списъка, дефиниран на ред 11. Последователността на имената на градовете, които итераторите ще върнат, е показана на редове 14-20. Ако продължим да извличаме данни, след достигане до последния елемент от итератора, ще получим стойност $undefined$ (ред 21).

Promises

Вече знаем, че JavaScript runtime използва *една нишка* при изпълнение на програмен код. Това означава, че системният и потребителският JavaScript код споделят една програмна нишка. За да не се блокират системните функции на браузъра (например обработка на събития от потребителския интерфейс), Web APIs използват множество нишки, когато е необходимо. Когато заявено събитие се сбъдне, event loop записва информация в message queue коя $callback$ функция трябва да се изпълни.

Задача 3

Нека от тялото на цикъл да разпечатаме на конзолата N на брой пъти дадено съобщение. Печатът на отделните съобщения трябва да се реализира през интервал от T милисекунди.

Следва програмен код - опит за решаване на задачата:

```
1   var T = 1000, N=10;
2   for (let i=0; i<N; i++) {
3       setTimeout(() => {
4           console.log("Съобщение "+i);
5       }, T);
6   }
```

Използва се функция $setTimeout$ (ред 3), за да получи необходимото време закъснение от T милисекунди, и програмен цикъл (редове 2-6), за да извикаме $callback$ (ламбда) функция N на брой пъти. В тялото на тази функция е асинхронният код, който печата на конзолата (ред 4). Това, което се *очаква* да се разпечата на конзолата, е следното:

Съобщение 0
Съобщение 1
…
Съобщение N-1

Програмният език JavaScript предлага начин за създаване на асинхронно работещи модули, както и възможност за синхронизиране на съвместната работа на множество такива модули. За целта се използва интерфейс $Promise$. Чрез обект, създаден чрез този интерфейс, е възможно да дефинираме асинхронно работещ код, както и да получим информация в момента, когато кодът завърши без или с грешка. Създаването на този обект е показано на Фиг. 8.

```
1   var promise = new Promise(function(resolve, reject) {
2       // асинхронен код, изпълнението на който
3       // завършва с уставяване на флаг статус
4       if (статус === "ок") {
5           resolve(обект);
6       }
7       else {
8           reject(обект);
9       }
10  });
```

Фиг. 8. Създаване на обект $promise$

Конструкторът (ред 1) изисква един аргумент – анонимна функция с два аргумента - $resolve$ и $reject$. Това са функции, които се викат при изпълнение на асинхронния код без грешка ($resolve$) или с грешка ($reject$). Тези функции изпълняват ролята на $callback$ функции. Обектите, които те предават, са достъпни чрез функция *then*. Тя изисква два атрибута. Първият атрибут е функция, която се вика при изпълнение на кода без грешка. Вторият атрибут е функция, която се вика при изпълнение на кода с грешка:

```
promise.then(function(обект) {
  // успешно изпълнен код (resolve)
}, function(обект) {
  // неуспешно изпълнен код (reject)
});
```

Фиг. 9. Анализ как е завършил асинхронния код чрез функция $then$

Обработката при грешки може да се реализира и чрез функция $catch$. Логиката е същата, както при предходния пример, но този синтаксис е по-разбираем:

```
promise.then(function(обект) {
  // успешно изпълнен код (resolve)
}).catch(function(обект) {
  // неуспешно изпълнен код
})
```

Фиг. 10. Обработка на грешките чрез функция $catch$

Когато трябва да се синхронизира работата на няколко блока от асинхронен код, за всеки от тях се създава обект $promise$. След това се използва функция all. Към така създадения чрез функция all promise обект може да се приложат функции $then$ и/или $catch$:

```
Promise.all(array_of_promises).then(function() {
  // всички модули са завършили успешно
}, function() {
  // един или повече модули не са
  // завършили успешно
});
```

Фиг. 11. Синхронизация на множество програмни модули чрез функция all

Нека да реализираме Задача 3 като използваме възможностите на интерфейс $Promise$. Първо, ще създадем функция, която връща обект $promise$ (виж Фиг. 12). Програмният код от тялото на анонимната функция стартира таймер (ред 3). След T милисекунди се вика функция $resolve$ (ред 4). Приема се, че няма вероятност за грешка при създаване на таймера и затова не се използва функция $reject$.

```
var timeout = function () {
    return new Promise(function(resolve) {
        setTimeout(function() {
            resolve("Съобщение ");
        }, T);
    });
};
```

Фиг. 12. Функция $timeout$ за време закъснение, която връща $promise$ обект

Печатането на съобщението от всеки таймер се реализира чрез използване на функция *then*. По този начин ще получим информация кога таймера е сработил. На Фиг.13 е показан програмният код на функция *print*, която в цикъл вика функции

timeout и *then* (ред 3). Ако изпълним функция $print$, след T милисекунди всички таймери едновременно ще върнат своето съобщение.

```
1  function print() {
2      for (let i=0; i<N; i++) {
3          timeout().then(function(data) {
4              console.log(data+i);
5          });
6      }
7  }
```

Фиг. 13. Функция за разпечатване на съобщенията – Вариант 1

При JavaScript е възможно да се укаже, че една функция съдържа код, който трябва да се изпълни асинхронно. За целта се използва ключова дума $async$. Чрез нея се дефинира асинхронна функция, която връща обект от тип $AsyncFunction$. Когато една асинхронна функция се извика тя връща $promise$ обект. Когато функцията завърши се вика функция $resolve$ с аргумент – върнатата стойност. Асинхронните функции могат да съдържат $await$ изрази. Това са изрази, които започват с ключова дума $await$. Тази ключова дума прекъсва временно изпълнението на асинхронната функция, докато асинхронния код от $promise$ обекта не завърши ($resolve$). Следователно, коректният програмен код на функция $print$ е следният:

```
1  async function print() {
2      for (let i=0; i<N; i++) {
3          await timeout().then(function(data) {
4              console.log(data+i);
5          });
6      }
7  }
```

Фиг. 14. Функция за разпечатване на съобщенията – Вариант 2

Използването на интерфейс $Promise$ решава проблема със синхронизацията на множество събития, но с малко повече програмен код. Това обаче не е проблем, тъй като предимствата на този код са по-важни: 1) Кодът е лесно разбираем; 2) Бързо и лесно може да се промени с цел решаване на подобни задачи (преизползваем код) и 3) Добра поддръжка на обработка на грешки (изключения).

Web workers

При JavaScript се предоставя възможност за изпълнение на програмен код във фонов режим под формата на модули, наричани *web workers*. Те частично симулират действието на програмните нишки. Всички по-нови версии на браузърите поддържат технология web workers. Програмният код на web worker е отделен JavaScript файл. Той функционира в *свой глобален контекст*, различен от този на текущия обект $window$. Всеки web worker код има свой *собствен* стек, хийп и message queue. Съществуват два вида web workers според контекста –

несподелени ($dedicated$) и *споделени* ($shared$). Контекстът на несподелените web worker се описва чрез обект $DedicatedWorkerGlobalScope$, а контекстът на споделените web worker - чрез обект $SharedWorkerGlobalScope$. В първият случай се създава web worker, функциалността на който е достъпна от един скрипт файл. Споделеният web worker е достъпен от множество скриптове, посредством обект с име $port$. Част от браузърите поддържат *subworkers*. Те се получават като от тялото на web worker се създава един или няколко обекта чрез конструктора $Worker$. Задължително трябва да проверите дали можете да създадете този обект, защото браузъри като Chrome и Safari не поддържат subworkers.

За да се изпълнява кода от web workers *паралелно* е необходимо процесорът да е многоядрен. Броят на ядрата (логически) можете да получите чрез следния код:

$$var\ \text{бройЯдра} = window.navigator.hardwareConcurrency;$$

Следва описание на начина на използване на *несподелени* web workers. Програмно може да установите дали браузътът поддържа web workers като проверите дали се поддържа тип $Worker$:

```
if ( typeof (Worker) === "function") {
    // поддържа се
}
else {
    // не се поддържа
}
```

Обект web worker се създава чрез програмния интерфейс $Worker$. На конструктора се предава пътя и името на файла с кода на web worker:

```
var worker = new Worker("js/worker.js");
```

Кодът на web worker е изолиран от останалия JavaScript код и се изпълнява подобно на отделна програмна нишка със свой собствен контекст. Достъпът до кода на web worker се реализира чрез разширяване на стандартните възможностите на браузърите за генериране и обработка на събития. За да получавате информация от web worker е необходимо да се прехване събитие $message$ и да се зададе име на $callback$ функцията, която ще се активира при наличие на данни от web worker:

```
worker.addEventListener("message", информацияОтWorker, false);
```

В този случай, всеки път когато web worker изпрати данни, ще се извика функция $\text{информацияОт}Worker$:

```
1  function информацияОтWorker(данни) {
2      var data = данни.data;
3      // извличане на данни ...
4      // логика ...
5  }
```

Тази функция получава данни – обект с име данни. Достъпът до тези данни (най-често JSON обект) се реализира на ред 2.

Можете да изпратите информация към web worker чрез метод $postMessage$ от интерфейс $Worker$. Методът предава тази информация към контекста на web worker като обект, който се *клонира*. Използва се *структурно клониране* - обектът се сериализира преди предаване и десериализира при приемане. Това гарантира изолация на кода на web worker от кода на останалите JavaScript модули. Можете да създадете свой собствен команден език и формат на отговорите. Следва пример за изпращане на команда $start$ с един параметър $time$. Тази информацията се изпраща като JSON обект:

```
worker.postMessage({command: "start", time: 100});
```

Командата може да се интерпретира по следния начин – активирай кода на web worker за време от 100 ms.

За да може web worker да получава информация, той трябва по аналогичен начин да прехване събитие $message$:

```
this.addEventListener("message", function(данни) {
    var data = данни.data;
    switch (data.command) {
        case "start":
            var time = data.time;
            // логика ...
            break;
        // други команди ...
    }
}
```

Наименованието и броят на командите, като и останалите параметри от JSON обекта, зависят изцяло от логиката на работа на web worker и е задача, която се решава от програмиста.

Когато web worker изпълни логиката, която му е заложена, неговия програмен код трябва да бъде премахнат от паметта. Това може да се реализира по два основни начина:

1. Изпращане чрез $postMessage$ на команда за унищожаване. При разпознаване на тази команда web worker трябва да се самоунищожи чрез следния код:

```
self.close();
```

2. Унищожаване на web worker с код извън неговия контекст. Това се реализира чрез метод $terminate$ от интерфейс $Worker$:

```
worker.terminate();
```

Кой метод ще използвате зависи от конкретната разработка и логика на функциониране на web worker, но за предпочитане е първият.

От тялото на web worker *нямате достъп до всички обекти*, които са създадени от браузъра с цел сигурност на кода. Имате достъп до следните по-важни обекти:

$Navigator$, $Array$, $Date$, $String$, $Math$, $XMLHttpRequest$, $WindowTimers$ (методи $setTimeout$, $setInterval$, $clearTimeout$ и $clearInterval$), $requestAnimationFrame$.

Както вече знаем кодът на един web worker трябва да е в отделен JavaScript файл. Ако Вашата цел е кодът на web worker и на модула, който го създава, да бъдат в един файл е необходимо да използвате *inline web worker*. Той се създава като на конструктора $Worker$ се подава $Blob$ (Binary Large Object) обект. Чрез него се създава URL манипулатор към кода на web worker:

```
var blob = new Blob(["onmessage = function(e) { "+
    "console.log(e.data);"+
    "postMessage('Информация от web worker'); }"]);

var blobHandle = window.URL.createObjectURL(blob);
var worker = new Worker(blobHandle);
worker.onmessage = function(event) {
    console.log(event.data);
};
worker.postMessage("Информация за web worker");
```