Jeszcze wydajniejsze witryny internetowe. Przyspieszanie działania serwisów WWW

Autor: Steve Souders
Tłumaczenie: Leszek Sagalara
ISBN: 978-83-246-2579-6
Tytuł oryginału: Even Faster Web Sites: Performance Best Practices for Web Developers
Format: 168×237, stron: 240

Poznaj najlepsze techniki zwiększania wydajności aplikacji internetowych!

- Jak stosować technikę kodowania porcjami w celu szybszego kodowania stron?
- Jak pisać wydajny kod JavaScript?
- Jak rozdzielać zasoby na wiele domen?

Wydajność witryny stanowi jeden z podstawowych czynników jej sukcesu w sieci. Jednak bogactwo treści i popularność technologii Ajax w dzisiejszych aplikacjach internetowych wystawiają przeglądarki na ciężką próbę. W tej sytuacji potrzebujesz profesjonalnych informacji i skutecznych metod zwiększających wydajność Twojej strony WWW. Jeśli chcesz ją poprawić, powinieneś skorzystać z tej książki, ponieważ znajdziesz tu mnóstwo wartościowych technik, które pomogą Ci zoptymalizować działanie każdej aplikacji.

Książka „Jeszcze wydajniejsze witryny internetowe. Przyspieszanie działania serwisów WWW" zawiera najbardziej aktualne porady, dzięki którym Twoja witryna otrzyma nowy zastrzyk energii. Z tego podręcznika dowiesz się, w jaki sposób Ajax wpływa na interakcję przeglądarek i serwerów, oraz nauczysz się wykorzystywać tę relację w celu identyfikacji elementów służących do poprawy wydajności aplikacji. Poznasz metody łączenia kodu osadzonego ze skryptami asynchronicznymi oraz kilka specyficznych technik przyspieszania JavaScriptu. Dzięki tej książce będziesz wiedział, jak zaoszczędzić cenne sekundy przez skrócenie czasu wczytywania, a także sprawisz, że Twoja witryna będzie działać jeszcze szybciej.

- Tworzenie responsywnych aplikacji WWW
- Wczytywanie skryptów bez blokowania
- Łączenie skryptów asynchronicznych
- Pozycjonowanie skryptów osadzonych
- Pisanie wydajnego kodu JavaScript
- Skalowanie przy użyciu Comet
- Optymalizacja grafiki
- Rozdzielanie zasobów na wiele domen
- Upraszczanie selektorów CSS

Szybkość ma znaczenie – zwiększ wydajność swojej strony WWW

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 2.

Tworzenie responsywnych aplikacji WWW

Ben Galbraith i Dion Almaer

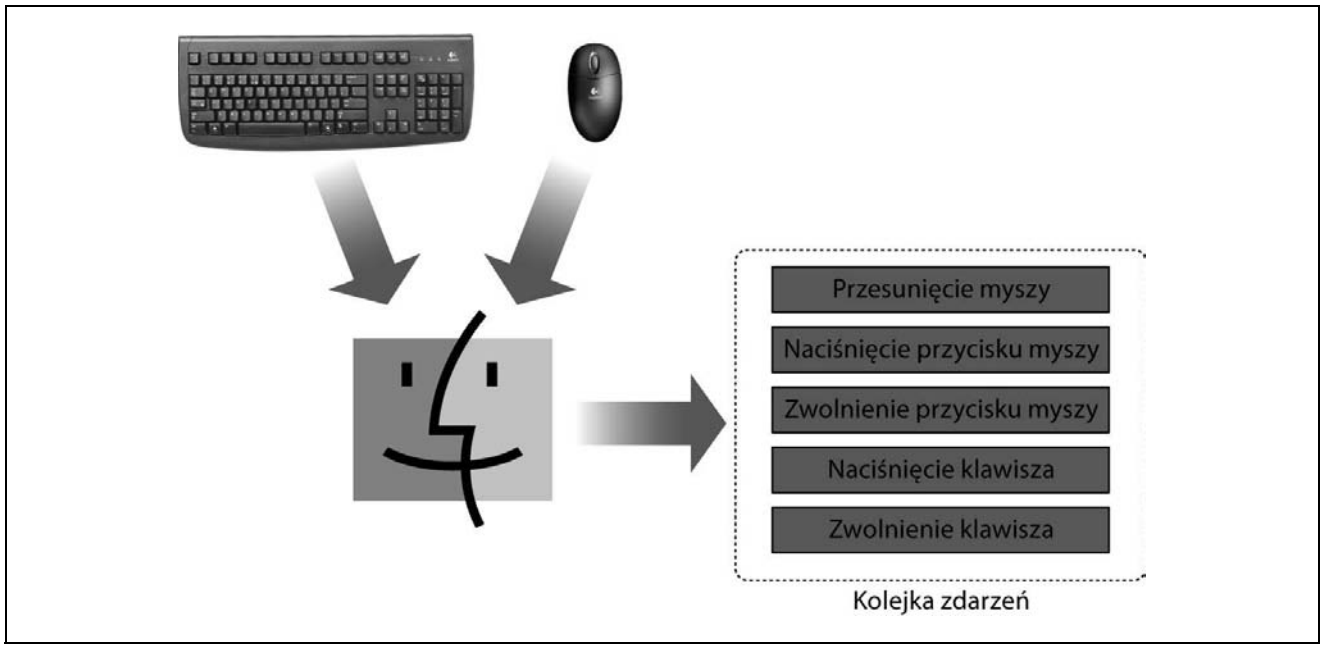
Technologia Ajax spowodowała, że wydajność witryn internetowych przestała być traktowana tylko w kategoriach szybkiego wczytywania stron. Coraz więcej witryn korzysta z JavaScriptu, aby po wczytaniu strony zmienić jej zawartość i wprowadzić w locie nową treść. Tego rodzaju witryny przypominają tradycyjne programy komputerowe, a optymalizacja ich wydajności wymaga użycia innego zestawu technik niż w przypadku tradycyjnych witryn internetowych.

Interfejsy użytkownika aplikacji WWW i tradycyjnych programów komputerowych mają wspólny cel: odpowiedzieć na działanie użytkownika tak szybko, jak to możliwe. W przypadku odpowiedzi na żądanie wczytania strony WWW większość pracy związanej z responsywnością przejmuje sama przeglądarka. Otwiera ona połączenie internetowe z wybraną witryną, analizuje kod HTML, wysyła żądania wczytania powiązanych zasobów itd. W oparciu o staranną analizę tego procesu możemy tak zoptymalizować nasze strony, aby były szybciej wyświetlane, ale ostatecznie to przeglądarka sprawuje kontrolę nad ich wczytywaniem i wyświetlaniem.

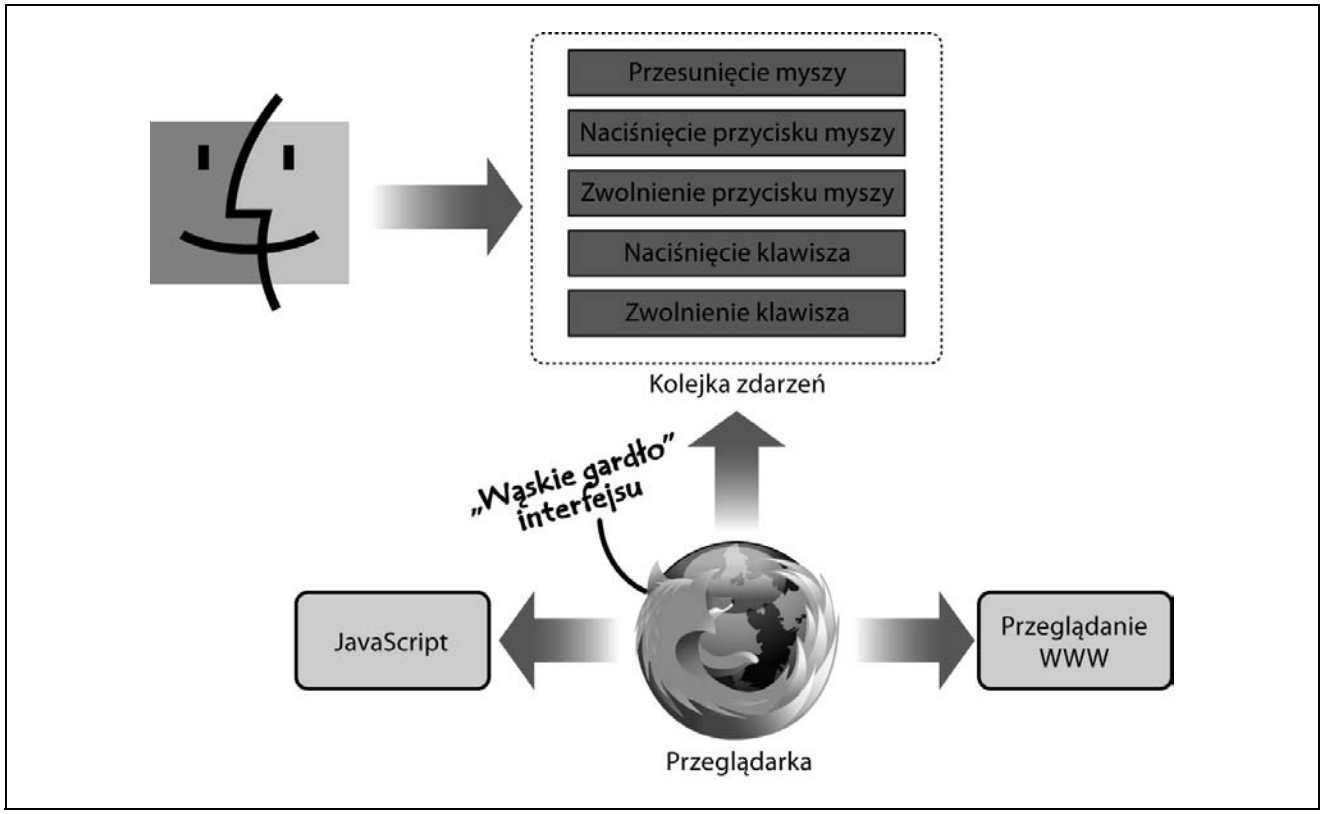
W przypadku działania użytkownika dotyczącego samej witryny (gdy nie powoduje ono załadowania nowej strony przez przeglądarkę) mamy nad tym kontrolę. Musimy zapewnić responsywność kodu JavaScript wykonywanego w wyniku tego działania. Aby lepiej zrozumieć, w jakim stopniu możemy kontrolować responsywność, niezbędne będzie wyjaśnienie sposobu działania interfejsu użytkownika w przeglądarce.

Jak widać na rysunku 2.1, gdy użytkownik pracuje z przeglądarką, system operacyjny otrzymuje sygnały wejściowe z różnych urządzeń podłączonych do komputera, takich jak klawiatura lub mysz. System rozdziela te sygnały na aplikacje, do których powinny one trafić, pakietuje je jako pojedyncze zdarzenia i umieszcza w kolejce dla danej aplikacji, zwanej *kolejką zdarzeń*.

Przeglądarka internetowa, podobnie jak każda inna aplikacja GUI, przetwarza następnie poszczególne zdarzenia umieszczone w swojej kolejce i na ich podstawie wykonuje na ogół jedną z dwóch rzeczy: sama obsługuje dane zdarzenie (np. przez wyświetlenie menu, przeglądanie WWW, wyświetlenie okna preferencji itp.) lub wykonuje kod JavaScript znajdujący się na stronie (np. kod JavaScript zawierający procedurę obsługi zdarzenia `onclick`), co przedstawia rysunek 2.2.



Rysunek 2.1. System operacyjny kieruje wszystkie sygnały wejściowe użytkownika do kolejki zdarzeń



Rysunek 2.2. Przeglądarka używa pojedynczego wątku do przetworzenia zdarzeń z kolejki i wykonania kodu użytkownika

Warto w tym miejscu wspomnieć, że jest to proces w zasadzie jednowątkowy, tzn. przeglądarka używa jednego wątku, aby pobrać zdarzenie z kolejki, i albo robi coś sama („Przeglądanie WWW" na rysunku 2.2), albo wykonuje kod JavaScript. Wskutek tego przeglądarka może wykonać tylko jedno z tych zadań naraz, a każde z nich może zapobiec wystąpieniu innego zdarzenia.

Każda chwila spędzona przez przeglądarkę na wykonywaniu kodu JavaScript to okres, w ciągu którego nie może ona odpowiadać na inne zdarzenia użytkownika. Dlatego też niezwykle istotne jest, aby kod JavaScript umieszczony na stronie był wykonywany jak najszybciej. W przeciwnym razie strona WWW i sama przeglądarka mogą reagować z opóźnieniem lub całkiem zaprzestać działania.

Trzeba tu zaznaczyć, że cały ten wykład o działaniu przeglądarki i systemu operacyjnego pod względem obsługi sygnałów wejściowych i zdarzeń jest tylko uogólnieniem obejmującym szeroki zakres przypadków, które w szczegółach mogą się różnić. Jednak niezależnie od różnic wszystkie przeglądarki wykonują cały kod JavaScript w jednym wątku (chyba że użyjemy Web Workers, co zostanie omówione w dalszej części tego rozdziału), co sprawia, że z powodzeniem można zastosować techniki zalecane w tym rozdziale.

Co to znaczy „wystarczająco szybko"?

Łatwo powiedzieć, że kod powinien być wykonywany „tak szybko, jak to możliwe", ale czasami trzeba przeprowadzić operacje, które zajmują trochę czasu. Algorytmy szyfrujące, złożone generowanie grafiki i operacje na obrazach — to przykłady obliczeń, które są czasochłonne niezależnie od tego, ile wysiłku włożymy w to, aby były one „tak szybkie, jak to możliwe".

Jednakże, o czym wspomniał Douglas w rozdziale 1., programiści szukający sposobu na tworzenie responsywnych, wysoko wydajnych witryn internetowych nie mogą — i nie powinni — dążyć do tego celu przez optymalizowanie każdego napisanego przez siebie fragmentu kodu. Prawda wygląda inaczej: należy optymalizować tylko to, co nie działa wystarczająco szybko.

Dlatego też istotne jest zdefiniowanie, co jest „wystarczająco szybkie" w tym kontekście. Na szczęście ktoś już to za nas zrobił.

Jacob Nielsen, znany i poważany ekspert w dziedzinie użyteczności aplikacji WWW, w poniższym cytacie[1] kwestię „wystarczającej szybkości" przedstawia następująco:

> Wytyczne dotyczące czasu reakcji dla aplikacji WWW są takie same jak dla wszystkich innych aplikacji. Pozostają one niezmienne już od 37 lat i najprawdopodobniej nie zmienią się bez względu na to, jaka technologia zostanie zaimplementowana w przyszłości.
>
> **0,1 sekundy**: Ograniczenie pozwalające użytkownikom odnieść wrażenie, że bezpośrednio operują obiektami w graficznym interfejsie użytkownika. Dla przykładu jest to czas od zaznaczenia przez użytkownika kolumny w tabeli do podświetlenia kolumny lub zasygnalizowania w inny sposób, że została ona wybrana. Byłoby idealnie, gdyby był to również czas reakcji na sortowanie kolumny — w takim przypadku użytkownik miałby uczucie bezpośredniego sortowania tabeli.
>
> **1 sekunda**: Ograniczenie pozwalające użytkownikom odnieść wrażenie swobodnej nawigacji w przestrzeni poleceń bez konieczności nadmiernego oczekiwania na odpowiedź komputera. Opóźnienie od 0,2 do 1 sekundy oznacza, że użytkownicy je zauważą i będą mieć wrażenie, że komputer „pracuje" nad poleceniem, w przeciwieństwie do sytuacji, gdy wykonanie polecenia jest bezpośrednim efektem działania użytkownika. Przykład: jeśli nie można posortować tabeli zgodnie z wybraną kolumną w czasie krótszym niż 0,1 sekundy, trzeba to zrobić w ciągu 1 sekundy, w przeciwnym razie użytkownicy odniosą wrażenie, że interfejs działa ociężale, i stracą

[1] *http://www.useit.com/papers/responsetime.html*

poczucie płynności w wykonywaniu swojej pracy. W przypadku opóźnień większych niż 1 sekunda należy zasygnalizować użytkownikowi, że komputer pracuje nad problemem, np. przez zmianę kształtu kursora.

10 sekund: Ograniczenie dla użytkowników skupiających swą uwagę na zadaniu. Wszystko, co trwa dłużej niż 10 sekund, wymaga procentowego wskaźnika, a także wyraźnie wskazanego sposobu przerwania operacji. Trzeba założyć, że wracając do interfejsu po opóźnieniu dłuższym niż 10 sekund, użytkownicy stracą płynność wykonywania zadań i będą musieli się „przestawić". Opóźnienia dłuższe niż 10 sekund są dopuszczalne tylko podczas naturalnych przerw w pracy użytkownika, np. przy przełączaniu zadań.

Innymi słowy, jeśli wykonywanie Twojego kodu JavaScript trwa dłużej niż 0,1 sekundy, Twoja strona nie wywoła wrażenia płynności i szybkości; jeśli zajmie to dłużej niż 1 sekundę, aplikacja będzie się wydawać ociężała; przy opóźnieniu przekraczającym 10 sekund użytkownik będzie już niezmiernie poirytowany. Takie są ostateczne wskazówki, jeśli chodzi o definicję „wystarczającej szybkości".

Pomiar opóźnienia

Skoro znasz już progi opóźnień, następnym etapem będzie poznanie sposobów pomiaru szybkości wykonywania kodu JavaScript, co pozwoli określić, czy nie przekracza ona wspomnianych wcześniej zakresów (sam musisz określić, jak szybko ma działać Twoja strona; naszym celem jest utrzymanie wszystkich opóźnień interfejsu poniżej granicy 0,1 sekundy).

Najłatwiejszym, najprostszym i prawdopodobnie najmniej precyzyjnym sposobem pomiaru opóźnienia jest obserwacja przez człowieka; po prostu użyj aplikacji na swojej docelowej platformie i sprawdź, czy ma wystarczającą wydajność. W końcu zapewnienie odpowiedniej wydajności interfejsu ma służyć człowiekowi, więc jest to właściwy sposób przeprowadzenia takich pomiarów (oczywiście tylko nieliczni będą w stanie podać czas opóźnienia w sekundach lub ułamkach sekund; większość posłuży się bardziej ogólnymi określeniami, np. „szybki", „powolny", „zadowalający" itp.).

Jeśli jednak pragniesz uzyskać bardziej precyzyjne wyniki, masz dwie możliwości: ręczne (**logowanie**) lub zautomatyzowane (**profilowanie**) oprzyrządowanie kodu.

Ręczne oprzyrządowanie kodu jest naprawdę proste. Powiedzmy, że mamy na stronie funkcję obsługi zdarzenia, np.:

```
<div onclick="myJavaScriptFunction()"> ... </div>
```

Prostym sposobem dodania oprzyrządowania ręcznego będzie zlokalizowanie definicji funkcji `myJavaScriptFunction()` i dodanie do niej pomiaru czasu:

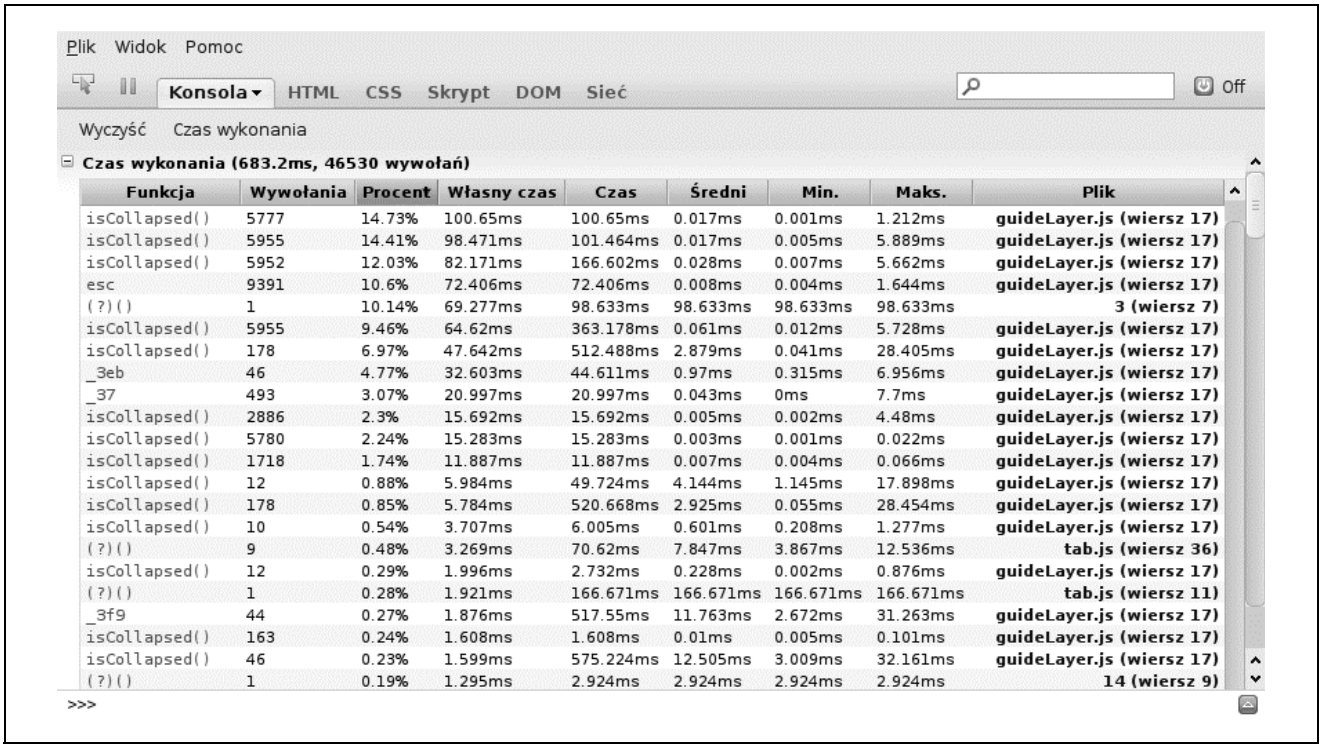
```
function myJavaScriptFunction() {
    var start = new Date().getMilliseconds();
    // tutaj jakiś skomplikowany kod
    var stop = new Date().getMilliseconds();
    var executionTime = stop - start;
    alert("Wykonywanie funkcji myJavaScriptFunction() trwało " + executionTime + "
    ↪millisekund");
}
```

Powyższy kod spowoduje otwarcie okna wyświetlającego czas wykonywania; milisekunda to $^{1}/_{1000}$ sekundy, więc 100 milisekund odpowiada wspomnianemu wcześniej progowi 0,1 sekundy.

Wiele przeglądarek jest wyposażonych we wbudowaną konsolę zapewniającą funkcję `log()` (Firefox udostępnia ją w popularnym dodatku Firebug); osobiście bardziej wolimy `console.log()` niż `alert()`.

Istnieją narzędzia przeprowadzające automatyczny pomiar czasu wykonywania kodu, ale zazwyczaj stosuje się je do innych celów. Zamiast do pomiaru dokładnego czasu wykonywania funkcji narzędzia takie — zwane profilerami — są zwykle używane do określenia względnej ilości czasu, jaki zajmuje wykonanie zestawu funkcji; inaczej mówiąc, służą one do wykrywania „wąskich gardeł" lub wolniej wykonywanych fragmentów kodu.

Firebug (*http://getfirebug.com/*), popularny dodatek do przeglądarki Firefox, zawiera profiler kodu JavaScript generujący komunikaty wyjściowe podobne do przedstawionych na rysunku 2.3.



Rysunek 2.3. Profiler w dodatku Firebug

Kolumna *Czas* przedstawia czas, jaki interpreter JavaScriptu poświęcił ogółem na wykonanie danej funkcji w okresie profilowania. Często zdarza się, że jakaś funkcja wywołuje inne funkcje; kolumna *Własny czas* przedstawia ilość czasu poświęconego na wykonywanie tylko określonej funkcji, bez innych funkcji, które mogły być wywoływane.

Można by sądzić, że ta i inne kolumny podające czas przedstawiają precyzyjny pomiar czasu wykonywania funkcji. Okazuje się jednak, że profilery wywołują coś zbliżonego do efektu obserwatora w fizyce: czynność obserwacji działania kodu wpływa na działanie kodu.

Profilery mogą stosować jedną z dwóch strategii: albo będą ingerować w kod będący przedmiotem pomiarów przez dodanie specjalnego kodu zbierającego statystyki wydajności (chodzi o proste zautomatyzowanie tworzenia kodu przedstawionego na poprzednim listingu), albo pasywnie monitorować czas uruchamiania, sprawdzając, jaki fragment kodu wykonywany jest w danym momencie. Ostatnie z tych dwóch podejść w mniejszym stopniu wpływa na wykonywanie profilowanego kodu, jednak odbywa się to kosztem zebrania danych o niższej jakości.

W dodatku Firebug wyniki ulegają dalszym zniekształceniom, ponieważ jego profiler wywołuje własny proces wewnątrz procesu Firebuga, co obniża wydajność kodu będącego przedmiotem pomiarów.

Kolumna *Procent* w oknie dodatku Firebug pokazuje względny czas wykonania: na interfejsie swojej strony możesz przeprowadzić jakieś wysokopoziomowe zadanie (np. kliknąć przycisk *Wyślij*), a następnie za pomocą profilera dodatku Firebug sprawdzić, które funkcje zabierają najwięcej czasu, i skupić swoje wysiłki na ich optymalizacji.

Gdy opóźnienia są zbyt duże

Okazuje się, że gdy Twój kod JavaScript zatrzymuje wątek przeglądarki na szczególnie długi czas, większość przeglądarek zainterweniuje i pozwoli użytkownikowi przerwać wykonywanie kodu. Nie ma żadnego standardu mówiącego o tym, w jaki sposób przeglądarki mają określać, czy dać użytkownikowi taką możliwość (szczegółowe informacje dotyczące zachowania poszczególnych przeglądarek znajdują się na stronie *http://www.nczonline.net/blog/2009/01/05/what-determines-that-a-script-is-long-running/*).

Wniosek jest prosty: nie wprowadzaj na swoją stronę WWW źle napisanego kodu, którego wykonywanie może trwać potencjalnie bardzo długo.

Wątkowanie

Po zidentyfikowaniu kodu, który zachowuje się nieodpowiednio, następnym etapem będzie jego optymalizacja. Czasem jednak zadanie wykonywane przez taki kod może wiązać się z dużym kosztem i nie da się go w magiczny sposób zoptymalizować w celu jego przyspieszenia. Czy taki scenariusz musi się wiązać ze spowolnieniem działania interfejsu? Czy nie ma żadnego sposobu, aby zadowolić naszych użytkowników?

Tradycyjnym rozwiązaniem w takich przypadkach jest skorzystanie z **wątków**, aby odciążyć wątek używany do interakcji z użytkownikiem od wykonywania kodu powodującego duży koszt. W naszym scenariuszu umożliwi to przeglądarce kontynuowanie przetwarzania zdarzeń z kolejki i zachowanie responsywności interfejsu, podczas gdy długo działający kod będzie wykonywany w innym wątku (a system operacyjny zatroszczy się o to, aby zarówno wątek interfejsu użytkownika przeglądarki, jak i wątek wykonywany w tle sprawiedliwie korzystały z zasobów komputera).

JavaScript nie zawiera jednak obsługi wątków, dlatego w przypadku kodu JavaScript nie ma możliwości utworzenia wątku wykonującego w tle kod o dużym koszcie. Co więcej, nie wygląda na to, aby ta sytuacja miała ulec zmianie.

Brendan Eich, twórca JavaScriptu i dyrektor techniczny fundacji Mozilla, wyraził się dość jasno w tej kwestii[2]:

> Żeby pracować z systemami wątkowania, musisz być [wielki jak gracz NBA], a to oznacza, że większość programistów powinna uciec z płaczem. Ale tak nie zrobią. Zamiast tego, podobnie jak to jest w przypadku większości innych ostrych narzędzi, będzie ich kusić, żeby pokazać, jacy

[2] *http://weblogs.mozillazine.org/roadmap/archives/2007/02/threads_suck.html*

są wielcy. Wezmą najbliższy jednowątkowy kod i wepchną go w środowisko wielowątkowe lub w inny sposób doprowadzą do sytuacji wyścigu. Sporadycznie efekty będą tragiczne, ale zbyt często skończy się to tak, że mimo pokaleczenia sobie palców nikt nie wyniesie z tego nauki.

Wątki wprowadzają cały szereg zakłóceń, głównie przez tworzenie sytuacji wyścigu, ryzyko zakleszczenia i pesymistyczną blokadę obciążenia. I wciąż nie są na tyle skalowalne, aby mogły w przyszłości obsłużyć te wszystkie megardzenie i teraflopy.

Dlatego moja standardowa odpowiedź na takie pytania, jak: „Kiedy dodasz wątki do JavaScriptu?", brzmi: „Po twoim trupie!".

Biorąc pod uwagę wpływ Brendana na branżę i na przyszłość JavaScriptu (który jest znaczny) oraz fakt, że wiele osób w dużym stopniu podziela jego poglądy, można bezpiecznie przyjąć, że w JavaScripcie nieprędko pojawią się wątki.

Istnieją jednak inne rozwiązania. Podstawowy problem z wątkami polega na tym, że różne wątki mogą mieć dostęp do tych samych zmiennych i mogą je modyfikować. To powoduje cały szereg problemów, np. gdy wątek A modyfikuje zmienne, które są aktywnie modyfikowane przez wątek B itp. Można by sądzić, że przyzwoity programista poradzi sobie z tego rodzaju kwestiami, ale jak się okazuje, Brendan twierdzi, że nawet najlepsi z nas popełniają okropne pomyłki w tej dziedzinie.

Zapewnienie responsywności

To, czego potrzebujemy, to odnoszenie korzyści z wątków — równoległego wykonywania zadań — bez ryzyka, że będą one sobie nawzajem wchodzić w paradę. Firma Google w swoim popularnym dodatku do przeglądarek o nazwie Gears zaimplementowała właśnie takie API: WorkerPool. W zasadzie umożliwia ono głównemu wątkowi JavaScript przeglądarki tworzenie działających w tle wątków roboczych (ang. *workers*), które rozpoczynając pracę, otrzymują pewne proste komunikaty (np. stan autonomiczny, bez odniesień do wspólnych zmiennych) z wątku przeglądarki i zwracają komunikat po jej zakończeniu.

Doświadczenia z działania tego API w dodatku Gears sprawiły, że w wielu przeglądarkach (np. Safari 4, Firefox 3.1[3]) wprowadzono bezpośrednią obsługę wątków roboczych w oparciu o wspólne API zdefiniowane w specyfikacji HTML 5. Opcja ta jest znana pod nazwą Web Workers.

Web Workers

Rozważmy, w jaki sposób wykorzystać API Web Workers do odszyfrowania wartości. Poniższy listing przedstawia sposób tworzenia i zainicjowania wątku roboczego:

```
// utworzenie i rozpoczęcie wykonywania wątku roboczego
var worker = new Worker("js/decrypt.js");
// zarejestrowanie procedury obsługi zdarzenia, która zostanie wykonana,
// gdy wątek roboczy wyśle komunikat do wątku głównego
worker.onmessage = function(e) {
    alert("Odszyfrowana wartość to " + e.data);
}
// wysłanie komunikatu do wątku roboczego, w tym przypadku będzie to wartość do odszyfrowania
worker.postMessage(getValueToDecrypt());
```

[3] Przeglądarka Firefox nosiła numer 3.1 w wersji beta, ostatecznie jednak została wydana jako Firefox 3.5 — *przyp. tłum.*

Przyjrzyjmy się teraz hipotetycznej zawartości skryptu *js/decrypt.js*:

```
// zarejestrowanie funkcji obsługi komunikatów przekazywanych przez wątek główny
    onmessage = function(e) {
    // pobranie otrzymanych danych
    var valueToDecrypt = e.data;

    // Do zrobienia: zaimplementować w tym miejscu funkcję deszyfrującą

    // zwrócenie wartości do wątku głównego
    postMessage(decryptedValue);
}
```

Wszystkie potencjalnie kosztowne (tj. o długim czasie działania) operacje JavaScriptu wykonywane przez Twoją stronę powinny być przekazywane do wątków roboczych. Dzięki temu Twoja aplikacja będzie działać z pełną szybkością.

Gears

Jeśli znajdziesz się w sytuacji, że Twoja aplikacja ma działać w przeglądarce, która nie obsługuje API Web Workers, istnieje kilka innych rozwiązań. W poprzednim rozdziale wspomnieliśmy o dodatku Gears firmy Google; dzięki niemu można uzyskać coś bardzo podobnego do Web Workers w przeglądarce Internet Explorer, w starszych wersjach Firefoksa i starszych wersjach Safari.

API wątków roboczych w Gears jest podobne do API Web Workers, choć nie identyczne. Poniżej przedstawione zostały dwa poprzednie listingi z kodem skonwertowanym na API Gears, począwszy od kodu wykonywanego w głównym wątku w celu zainicjowania wątku roboczego:

```
// utworzenie puli wątków, która zainicjuje wątki robocze
var workerPool = google.gears.factory.create('beta.workerpool');

// zarejestrowanie funkcji obsługi zdarzeń, która będzie odbierać komunikaty z wątków roboczych
workerPool.onmessage = function(ignore1, ignore2, e) {
    alert("Odszyfrowana wartość to + " e.body);
}

// utworzenie wątku roboczego
var workerId = workerPool.createWorkerFromUrl("js/decrypt.js");

// wysłanie komunikatu do wątku roboczego
workerPool.sendMessage(getValueToDecrypt(), workerId);
```

A tak wygląda zawartość skryptu *js/decrypt.js* w wersji Gears:

```
var workerPool = google.gears.workerPool;
workerPool.onmessage = function(ignore1, ignore2, e) {
    // pobranie przysłanych danych
    var valueToDecrypt = e.body;

    // Do zrobienia: zaimplementować w tym miejscu funkcję deszyfrującą

    // zwrócenie wartości do wątku głównego
    workerPool.sendMessage(decryptedValue, e.sender);
}
```

Więcej na temat Gears

Warto poznać kilka faktów z historii powstania wątków roboczych Gears, ponieważ zostały one wymyślone z bardzo praktycznych względów. Dodatek Gears został utworzony przez zespół Google, który starał się zmusić przeglądarkę do wykonywania zadań, jakich nie była wtedy w stanie wykonać (działo się to przed powstaniem Google Chrome — ale nawet mając Chrome, Google chce, aby możliwie jak największa liczba użytkowników mogła wykonywać złożone zadania za pomocą ich aplikacji WWW).

Wyobraź sobie, że chciałbyś zbudować Gmail Offline — czego być potrzebował? Po pierwsze, musiałbyś znaleźć sposób na lokalne buforowanie dokumentów i ich przechwytywanie, aby — gdy przeglądarka spróbuje uzyskać dostęp do strony *http://mail.google.com/* — wyświetliła się żądana strona, a nie komunikat o braku połączenia. Po drugie, wymagałoby to jakiejś metody przechowywania wiadomości e-mail, zarówno tych nowych, jak i starych. Można by to zrobić na wiele sposobów, ale skoro dobrze znany SQLite jest obecny w większości nowych przeglądarek i dołączany do wielu systemów operacyjnych, dlaczego tego nie wykorzystać? Tutaj właśnie tkwi problem.

Omawialiśmy kwestie związane z jednowątkową przeglądarką. Wyobraź sobie teraz takie operacje, jak zapisywanie nowych wiadomości e-mail do bazy danych lub wykonywanie długich zapytań. Możemy zablokować interfejs użytkownika w czasie, gdy baza danych wykonuje swoją pracę — opóźnienia mogą być ogromne! Zespół pracujący nad Gears musiał sobie jakoś z tym poradzić. Dodatek Gears może robić, co chce, więc z łatwością można było obejść brak wątków w JavaScripcie. Skoro jednak współbieżność stanowi ogólny problem, czemu nie udostępnić tej możliwości na zewnątrz? Tak powstało API WorkerPool, które doprowadziło do powstania Web Workers w standardzie HTML 5.

Te dwa API różnią się nieznacznie, ale wynika to stąd, że Web Workers jest czymś w rodzaju wersji 2.0 pionierskiego API Gears; niebawem Gears powinien zostać wyposażony w obsługę standardowego API. Powstały już biblioteki pośredniczące, będące czymś w rodzaju pomostu między istniejącym API Gears a standardowym API Web Workers, których można używać nawet bez Gears lub Web Workers (stosując `setTimeout()`, co zostanie omówione w dalszej części tego rozdziału).

Timery

Inne podejście, popularne przed powstaniem Gears i Web Workers, polegało na podzieleniu długotrwałych operacji na mniejsze części i kontrolowaniu ich działania przy użyciu timerów JavaScriptu. Np.:

```
var functionState = {};

function expensiveOperation() {
    var startTime = new Date().getMilliseconds();
    while ((new Date().getMilliseconds() - startTime) < 100) {
        // Do zrobienia: zaimplementować długotrwałą operację w taki sposób,
        // aby cała praca była wykonywana w mniejszych fragmentach,
        // trwających krócej niż 100 ms, z przekazywaniem stanu do "functionState"
        // poza tą funkcją; powodzenia ;-)
    }

    if (!functionState.isFinished) {
        // ponownie wprowadzić długotrwałą operację 10 ms po wyjściu;
        // warto poeksperymentować z większymi wartościami, aby zachować
        // właściwe proporcje między responsywnością a wydajnością
```

```
        setTimeout(expensiveOperation(), 10);
        }
    }
```

Dzieląc operację w przedstawiony sposób, zachowamy responsywność interfejsu, ale — jak wskazuje komentarz w listingu — skonstruowanie takiej operacji może być trudne (a nawet niewykonalne). Więcej informacji na temat takiego użycia metody `setTimeout()` zawiera punkt „Wprowadzanie przerw przy użyciu timerów" na stronie 116.

Z podejściem tym wiąże się inna podstawowa kwestia. Większość nowoczesnych komputerów wyposażona jest w procesory o kilku rdzeniach, co oznacza, że mogą pracować naprawdę wielowątkowo (podczas gdy wcześniejsze procesory jedynie emulowały wielowątkowość przez szybkie przełączanie się między zadaniami). Ręczne przełączanie zadań zaimplementowane za pomocą Javascriptu, jakie miało miejsce w przedstawionym listingu, nie może wykorzystywać takich architektur; a więc nie użyjemy całej mocy procesora, zmuszając jeden z rdzeni do wykonania całej pracy.

A zatem możliwe jest przeprowadzanie długotrwałych operacji w głównym wątku przeglądarki i zachowanie responsywności interfejsu, ale łatwiejsze i wydajniejsze będzie użycie wątków roboczych.

XMLHttpRequest

Cała ta dyskusja na temat wątkowania nie byłaby kompletna, gdybyśmy nie wspomnieli o `XMLHttpRequest` (w skrócie XHR), który umożliwił rewolucję, jaką był Ajax. Używając XHR, strona WWW może wysłać komunikat i otrzymać odpowiedź w całości w środowisku JavaScript, co pozwala uzyskać złożoną interaktywność bez konieczności wczytywania nowych stron.

XHR ma dwa proste tryby działania: synchroniczny i asynchroniczny. W trybie asynchronicznym XHR jest w zasadzie wątkiem Web Worker, tyle że posiada wyspecjalizowane API; w istocie w połączeniu z innymi funkcjami powstającej specyfikacji HTML 5 można odtworzyć działanie XHR za pomocą wątków roboczych. W trybie synchronicznym XHR działa w taki sposób, że całą swą pracę wykonuje w głównym wątku przeglądarki, co sprawia, że opóźnienia w interfejsie użytkownika trwają tyle czasu, ile potrzeba XHR na wysłanie jego żądania i przeanalizowanie odpowiedzi z serwera. Dlatego też nigdy nie używaj XHR w trybie synchronicznym, gdyż może to doprowadzić do nieprzewidzianych opóźnień w funkcjonowaniu interfejsu użytkownika, znacznie wykraczających poza dopuszczalne granice.

Wpływ zużycia pamięci na czas odpowiedzi

W tworzeniu responsywnych stron WWW występuje jeszcze inny kluczowy aspekt: zarządzanie pamięcią. Podobnie jak w wielu nowoczesnych językach wysokiego poziomu, które zawierają elementy niskopoziomowego zarządzania pamięcią, tak i w większości środowisk uruchomieniowych JavaScriptu występuje mechanizm automatycznego oczyszczania pamięci (ang. *garbage collection*, w skrócie GC). Może to być cudowne rozwiązanie, uwalniające programistów od nużących detali kojarzących się bardziej z księgowością niż z programowaniem.

Automatyczne zarządzanie pamięcią wiąże się jednak z pewnym kosztem. Wszystkie z wyjątkiem najbardziej wyrafinowanych implementacji GC „zatrzymują cały świat", przeprowadzając oczyszczanie pamięci, tzn. „zamrażają" całe środowisko (włącznie z tym, co nazwaliśmy

głównym wątkiem JavaScript przeglądarki), podczas gdy same wykonują całą masę czynności związanych z tworzeniem obiektów oraz wyszukiwaniem tych, które nie są już używane i które można zutylizować do nieużywanych obszarów pamięci.

W przypadku większości aplikacji proces GC jest całkowicie przezroczysty; okresy zablokowania środowiska uruchomieniowego są na tyle krótkie, że całkowicie umyka to uwadze użytkownika. Jednak w miarę zajmowania przez aplikację coraz większych obszarów pamięci wydłuża się czas niezbędny do wyszukania nieużywanych obiektów i ostatecznie może osiągnąć poziom, który będzie zauważalny dla użytkownika.

Gdy to nastąpi, aplikacja w dość regularnych odstępach czasu będzie działać mniej lub bardziej ociężale; w miarę narastania problemu może dojść nawet do zablokowania przeglądarki. Oba te efekty mogą doprowadzić użytkownika do frustracji.

Większość nowoczesnych platform dostarcza wyrafinowanych narzędzi umożliwiających monitorowanie wydajności procesu GC w środowisku uruchomieniowym i podgląd bieżącego zestawu analizowanych obiektów w celu zdiagnozowania problemów związanych z oczyszczaniem pamięci. Niestety, środowiska uruchomieniowe JavaScriptu nie mieszczą się w tej kategorii. Co gorsza, nie istnieją żadne narzędzia, które mogłyby poinformować programistę o tym, kiedy następuje oczyszczanie pamięci lub ile czasu trwa jego przeprowadzenie. Wykorzystując je, można by sprawdzić, czy zaobserwowane opóźnienia mają związek z procesem oczyszczania pamięci.

Brak takich narzędzi jest poważną niedogodnością dla twórców złożonych aplikacji JavaScript działających w oparciu o przeglądarki. Na razie programiści mogą jedynie zgadywać, czy za opóźnienia interfejsu użytkownika odpowiada mechanizm automatycznego oczyszczania pamięci.

Pamięć wirtualna

Istnieje jeszcze inne niebezpieczeństwo związane z pamięcią: stronicowanie. W systemach operacyjnych mamy dwa rodzaje pamięci udostępnianej aplikacjom: fizyczna i wirtualna. **Pamięć fizyczna** to niezwykle szybkie moduły RAM zamontowane w komputerze; **pamięć wirtualna** znajduje się na znacznie wolniejszych urządzeniach pamięci masowej (np. na dysku twardym), które swoją stosunkową powolność nadrabiają znacznie większą dostępną przestrzenią pamięci.

Jeśli wymagania pamięciowe Twojej strony WWW wzrosną w znacznym stopniu, system operacyjny może być zmuszony do rozpoczęcia **stronicowania**, niezwykle powolnego procesu, przez co inne procesy będą musiały zwolnić zajmowaną przez siebie pamięć fizyczną, aby udostępnić miejsce na rosnący apetyt przeglądarki. Proces ten nazywa się **stronicowaniem**, ponieważ wszystkie nowoczesne systemy operacyjne dzielą pamięć na pojedyncze **strony**, tj. najmniejsze jednostki pamięci, które są odwzorowywane do pamięci fizycznej lub wirtualnej. W trakcie stronicowania strony te są przenoszone z pamięci fizycznej do pamięci wirtualnej (tj. z pamięci RAM na dysk twardy) lub odwrotnie.

Spadek wydajności spowodowany stronicowaniem różni się od przerw wywoływanych procesem oczyszczania pamięci; stronicowanie powoduje ogólne, wszechobecne spowolnienie, natomiast opóźnienia w procesie GC przejawiają się zwykle w formie dyskretnych, pojedynczych przerw występujących w określonych przedziałach czasowych — choć długość tych

przerw rośnie wraz z upływem czasu. Niezależnie od tych różnic każdy z tych problemów stanowi znaczące zagrożenie dla osiągnięcia Twojego celu, jakim jest utworzenie responsywnego interfejsu użytkownika.

Rozwiązywanie problemów związanych z pamięcią

Jak wspomnieliśmy wcześniej, nie znamy żadnych dobrych narzędzi do diagnostyki pamięci przeznaczonych dla aplikacji JavaScript uruchamianych w przeglądarce. Póki co należy obserwować zajętość pamięci procesu przeglądarki (szczegółowe informacje na temat pomiaru zajętości pamięci przez procesy w systemach Windows i OS X zawiera sekcja „Measuring Memory Use" pod adresem *http://blog.pavlov.net/2008/03/11/firefox-3-memory-usage/*). Jeśli podczas działania aplikacji zajętość pamięci wzrasta znacznie powyżej akceptowanych wartości, sprawdź, czy w kodzie Twojej aplikacji są jakieś możliwości optymalizacji zużycia pamięci.

Jeśli okaże się, że masz problem związany z pamięcią, powinieneś poszukać możliwości jej oczyszczenia, jeśli jeszcze tego nie zrobiłeś. Możesz to zrobić przez:

- użycie słowa kluczowego `delete` w celu usunięcia z pamięci zbędnych obiektów JavaScript,
- usunięcie zbędnych węzłów z modelu DOM strony WWW.

Poniższy listing przedstawia sposób przeprowadzenia obu tych zadań:

```
var page = { address: "http://jakis/adres/url" };

page.contents = getContents(page.address);

...

// później zawartość nie będzie już dłużej potrzebna
delete page.contents;

...

var nodeToDelete = document.getElementById("redundant");

// usuniecie węzła z modelu DOM (co można zrobić jedynie
// przez wywołanie do removeChild() z węzła nadrzędnego)
// i równoczesne usunięcie węzła z pamięci
delete nodeToDelete.parent.removeChild(nodeToDelete);
```

Oczywiście jest jeszcze wiele możliwości ulepszeń w zakresie optymalizacji zużycia pamięci przez strony WWW. W fundacji Mozilla zajmujemy się obecnie tworzeniem narzędzi przeznaczonych do rozwiązywania tego rodzaju problemów. W chwili gdy czytasz tę książkę, co najmniej jedno z takich narzędzi powinno być już dostępne pod adresem *http://labs.mozilla.com*.

Podsumowanie

Ajax zapoczątkował nową erę stron WWW działających w oparciu o JavaScript. Są one w rzeczywistości aplikacjami działającymi w przeglądarce i jeśli chodzi o interfejs użytkownika, podlegają takim samym wymogom jak każda inna aplikacja. Istotne jest to, aby zapewniały one responsywność interfejsu przez zminimalizowanie operacji wykonywanych w głównym wątku aplikacji.

Web Workers to potężne nowe narzędzie, które można wykorzystać do przerzucenia skomplikowanych operacji zagrażających responsywności interfejsu użytkownika. Jeśli nie można użyć Web Workers, możemy wykorzystać dodatek Gears i timery JavaScriptu.

Złe zarządzanie pamięcią może doprowadzić do powstania problemów związanych z wydajnością interfejsu użytkownika. Mimo braku dobrych narzędzi do rozwiązywania problemów z pamięcią programiści mogą obserwować wykorzystanie pamięci przez przeglądarkę i w razie wystąpienia problemów podjąć odpowiednie kroki w celu zminimalizowania zajętości pamięci przez aplikację. Dobra wiadomość jest taka, że trwają już prace nad utworzeniem narzędzi do rozwiązywania problemów z pamięcią.