РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

Как работает JS: сервис-воркеры

ПЕРЕВОД

ru_vds 26 февраля в 11:40 8,9k

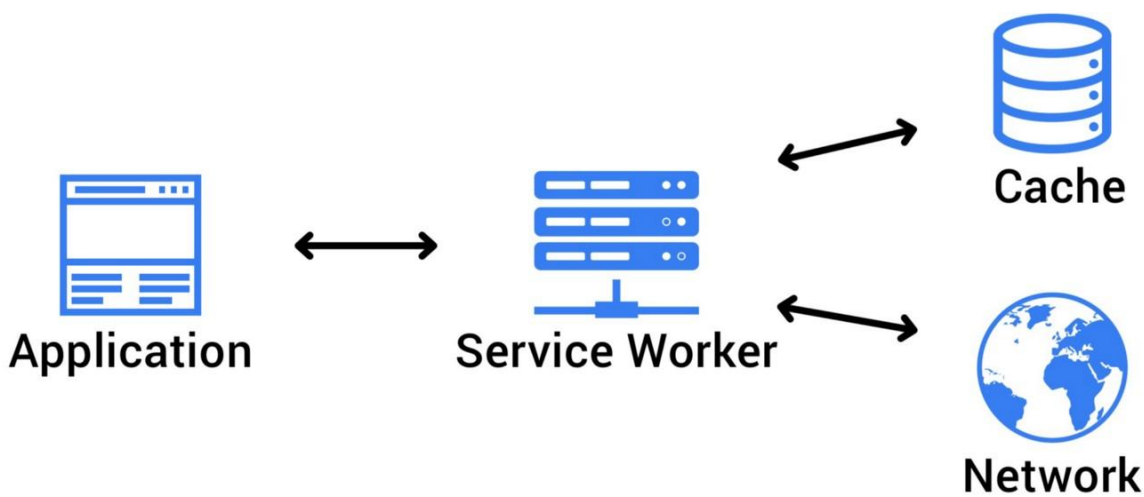
Оригинал: Alexander Zlatkov

Перед вами перевод восьмой части серии материалов об особенностях работы различных механизмов JavaScript. Сегодняшняя статья посвящена сервис-воркерам. Здесь мы рассмотрим их особенности, поговорим об их жизненном цикле, об их поддержке в браузерах, и о сценариях их использования.

Прогрессивные веб-приложения

Прогрессивные веб-приложения, по всей видимости, будут становиться всё популярнее и распространённее. Они нацелены на то, чтобы пользователь воспринимал их не как обычные веб-страницы, а как нечто вроде классических настольных приложений, которые нормально работают независимо от того, подключен компьютер к интернету или нет.

Отсюда исходит одно из основных требований к прогрессивным веб-приложениям, которое заключается в их надёжном функционировании при отсутствующем или нестабильном сетевом соединении. Сервис-воркеры являются важной технической деталью реализации подобного поведения приложений.



Приложение, сервис-воркер, кэш и сетевые ресурсы

Здесь вы можете видеть упрощённую схему взаимоотношений между приложением, сервис-воркером, кэшем, управлением

которым занимается сервис-воркер, и сетевыми ресурсами. В идеале, правильная организация взаимодействия приложения с сервис-воркером и кэшем позволит пользователю нормально работать с приложением даже без подключения к сети.

Особенности сервис-воркеров

Если вы хотите как следует изучить сервис-воркеры, вам стоит взглянуть на предыдущий материал этой серии, который посвящён веб-воркерам. В целом можно сказать, что сервис-воркеры — это разновидность веб-воркеров, а если точнее, то они похожи на разделяемые воркеры. В частности, можно выделить следующие важные особенности сервис-воркеров:

- Они выполняются в собственном глобальном контексте, `ServiceWorkerGlobalScope`.
- Они не привязаны к конкретной странице.
- Они не имеют доступа к DOM.

Особого внимания API сервис-воркеров заслуживает по той причине, что оно позволяет приложениям поддерживать оффлайновые сценарии работы, давая программисту полный контроль над тем, как приложение взаимодействует с внешними ресурсами.

Жизненный цикл сервис-воркера

Жизненный цикл сервис-воркера не имеет ничего общего с жизненным циклом веб-страницы. Он включает в себя следующие

этапы:

- Загрузка
- Установка
- Активация

Загрузка

На этом этапе жизненного цикла веб-браузер загружает `.js`-файл, содержащий код сервис-воркера.

Установка

Для того чтобы установить сервис-воркер, сначала его нужно зарегистрировать. Это делается в JavaScript-коде. Когда сервис-воркер зарегистрирован, браузеру предлагается запустить установку в фоновом режиме.

Регистрируя сервис-воркер, вы сообщаете веб-браузеру о том, где находится его `.js`-файл. Взглянем на следующий код:

```
if ('serviceWorker' in navigator) {
  window.addEventListener('load', function() {
    navigator.serviceWorker.register('/sw.js').then(function(registr
ation) {
      // Успешная регистрация
      console.log('ServiceWorker registration successful');
    }, function(err) {
      // При регистрации произошла ошибка
      console.log('ServiceWorker registration failed: ', err);
    });
  });
}
```

Тут производится проверка того, поддерживается ли Service Worker API в текущем окружении. Если это API поддерживается, то регистрируется сервис-воркер `/sw.js`.

Метод `register()` можно без проблем вызывать каждый раз, когда загружается страница. Браузер самостоятельно выяснит, был ли уже зарегистрирован соответствующий сервис-воркер и правильно обработает повторный запрос на регистрацию.

Важная особенность в работе с методом `register()` заключается в расположении файла сервис-воркера. В данном случае вы можете видеть, что файл сервис-воркера расположен в корне домена. В результате областью видимости сервис-воркера будет весь домен. Другими словами, этот сервис-воркер будет получать события `fetch` (о которых мы поговорим ниже), генерируемые всеми страницами из этого домена. Аналогично, если зарегистрировать файл сервис-воркера, расположенный по адресу `/example/sw.js`, этот сервис-воркер будет видеть лишь события `fetch` со страниц, URL которых начинается с `/example/` (то есть, например, `/example/page1/`, `/example/page2/`).

В ходе процесса установки сервис-воркера рекомендуется загрузить и поместить в кэш статические ресурсы. После их кэширования установка веб-воркера будет успешно завершена. Если загрузка не удастся, автоматически будет сделана попытка повторной установки. В результате, после успешной установки веб-воркера, разработчик может быть уверен в том, что все необходимые статические материалы находятся в кэше.

Всё это иллюстрирует тот факт, что нельзя говорить о том, что совершенно необходимо то, чтобы регистрация веб-воркера произошла после события `load`, но рекомендуется поступать именно так.

Почему? Представим, что пользователь впервые открывает веб-приложение. В этот момент сервис-воркер для этого приложения пока не загружен, более того, браузер не может узнать заранее, будет ли приложение использовать сервис-воркер. Если сервис-воркер будет устанавливаться, браузеру понадобится потратить дополнительные системные ресурсы. Эти ресурсы в противном случае пошли бы на рендеринг веб-страницы. Как результат, запуск процесса установки сервис-воркера может отсрочить загрузку и вывод страницы. Обычно же разработчики стремятся к тому, чтобы как можно быстрее показать пользователю рабочую страницу приложения, но в нашем случае без сервис-воркера приложение не сможет нормально работать.

Обратите внимание на то, что вышеприведённые рассуждения имеют смысл только при разговоре о первом посещении страницы. Если, после установки сервис-воркера, пользователь снова посетит ту же страницу, повторная установка производиться не будет, а значит, не пострадает и скорость показа рабочей страницы. После первого посещения страницы приложения сервис-воркер будет активирован, в результате он сможет обрабатывать события загрузки и кэширования при последующих посещениях веб-приложения. Как результат, приложение будет

готово к работе в условиях ограниченного сетевого соединения.

Активация

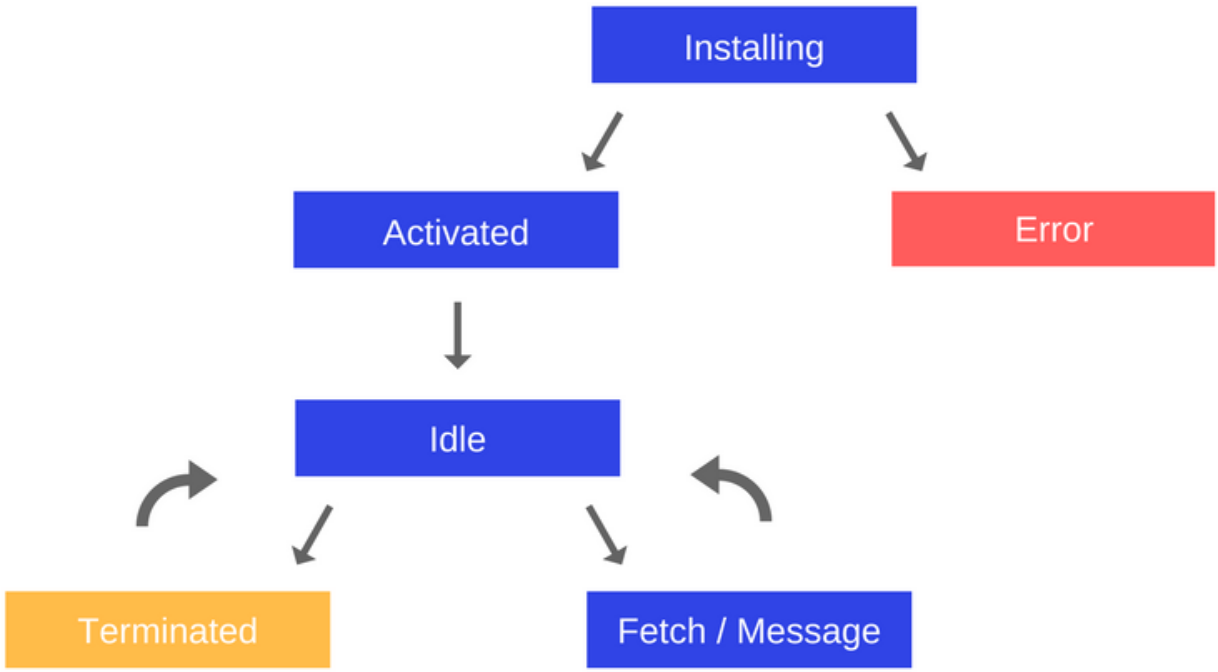
После установки сервис-воркера мы переходим к следующему этапу его жизненного цикла — к активации. На этом шаге у разработчика появляется возможность поработать с данными, которые были кэшированы ранее.

После активации сервис-воркер сможет управлять всеми страницами, которые попадают в его область видимости. Тут стоит отметить, что механизмы сервис-воркера не будут действовать на ту страницу, которая его зарегистрировала, до тех пор, пока эта страница не будет перезагружена.

После того, как сервис-воркер получит управление, он может оказаться в одном из следующих состояний:

- Обработка событий. Сервис-воркер ожидает поступления событий `fetch` и `message`, которые возникают, когда страницы выполняет сетевые запросы или отправляют сообщения. При поступлении события сервис-воркер его обрабатывает.
- Остановка. Система останавливает сервис-воркер для экономии ресурсов.

Вот как выглядит жизненный цикл сервис-воркера:



Жизненный цикл сервис-воркера

Обработка процесса установки внутри сервис-воркера

Сервис-воркер, после того, как был запущен процесс его регистрации, способен воздействовать на происходящее. В частности, речь идёт об обработчике события `install` в коде сервис-воркера.

Вот что нужно сделать сервис-воркеру при обработке события `install`:

- Открыть кэш.
- Поместить в кэш необходимые материалы.
- Подтвердить кэширование всех необходимых материалов.

Вот простой пример обработки события `install` в сервис-воркере:

```
var CACHE_NAME = 'my-web-app-cache';
var urlsToCache = [
  '/',
  '/styles/main.css',
  '/scripts/app.js',
  '/scripts/lib.js'
];

self.addEventListener('install', function(event) {
  // event.waitUntil принимает промис для того, чтобы узнать,
  // сколько времени займёт установка, и успешно
  // или нет она завершилась.
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('Opened cache');
        return cache.addAll(urlsToCache);
      })
  );
});
```

Если все материалы успешно кэшированы, это означает и успешную установку сервис-воркера. Если что-нибудь загрузить не удастся, тогда установка будет признана несостоявшейся. Поэтому следует обращать особое внимание на то, какие данные требуется поместить в кэш.

Тут надо отметить, что обработка события `install` внутри сервис-воркера необязательна.

Работа с кэшем в процессе выполнения приложения

Здесь начинается самое интересное. Именно тут мы разберём механизмы перехвата запросов, возврата кэшированных данных и кэширования новых материалов.

После того, как сервис-воркер будет установлен и пользователь перейдёт на другую страницу приложения или обновит страницу, на которой он находится, сервис-воркер начнёт получать события `fetch`. Вот пример, который показывает, как возвращать кэшированные материалы или выполнять новые запросы, а затем кэшировать результат:

```
self.addEventListener('fetch', function(event) {
  event.respondWith(
    // Этот метод анализирует запрос и
    // ищет кэшированные результаты для этого запроса в любом из
    // созданных сервис-воркером кэшей.
    caches.match(event.request)
      .then(function(response) {
        // если в кэше найдено то, что нужно, мы можем тут же
вернуть ответ.
        if (response) {
          return response;
        }

        // Клонируем запрос. Так как объект запроса - это поток,
        // обратиться к нему можно лишь один раз.
        // При этом один раз мы обрабатываем его для нужд
кэширования,
        // ещё один раз он обрабатывается браузером, для запроса
ресурсов,
        // поэтому объект запроса нужно клонировать.
        var fetchRequest = event.request.clone();

        // В кэше ничего не нашлось, поэтому нужно выполнить
загрузку материалов,
        // что заключается в выполнении сетевого запроса и в
возврате данных, если
        // то, что нужно, может быть получено из сети.
        return fetch(fetchRequest).then(
```

```
          function(response) {
            // Проверка того, получили ли мы правильный ответ
            if(!response || response.status !== 200 || response.type
!== 'basic') {
              return response;
            }

            // Клонирование объекта ответа, так как он тоже является
потоком.
            // Так как нам надо, чтобы ответ был обработан
браузером,
            // а так же кэшем, его нужно клонировать,
            // поэтому в итоге у нас будет два потока.
            var responseToCache = response.clone();

            caches.open(CACHE_NAME)
              .then(function(cache) {
                // Добавляем ответ в кэш для последующего
использования.
                cache.put(event.request, responseToCache);
              });

            return response;
          }
        );
      })
    );
});
```

Вот что тут, в общих чертах, происходит:

- Конструкция `event.respondWith()` определяет то, как мы будем реагировать на событие `fetch`. Мы передаём из `caches.match()` промис, который анализирует запрос и выясняет, имеются ли какие-либо кэшированные ответы на подобный запрос, сохранённые в любом из созданных кэшей.
- Если в кэше найдено то, что нужно, из него извлекается ответ.
- Если в кэше не найдено совпадений — выполняется операция `fetch`.

- Проверяется статус ответа (нам нужен статус 200). Кроме того, мы проверяем тип ответа, который должен равняться `basic`, что указывает на то, что это запрос из нашего домена. Запросы к материалам из сторонних источников в этом случае кэшированы не будут.
- Ответ добавляется в кэш.

Объекты запросов и ответов необходимо клонировать, так как они являются потоками. Поток можно обработать лишь один раз. Однако с этими потоками нужно работать и сервис-воркеру, и браузеру.

Обновление сервис-воркера

Когда пользователь посещает веб-приложение, браузер пытается выполнить повторную загрузку `.js`-файла, который содержит код сервис-воркера. Этот процесс выполняется в фоне.

Если имеется хотя бы мельчайшее различие между файлом сервис-воркера, который был загружен, и текущим файлом, браузер решит, что в коде воркера произошли изменения. Это означает, что в приложении должен использоваться новый сервис-воркер.

Браузер запустит этот новый сервис-воркер и вызовет событие `install`. Однако в этот момент за взаимодействие приложения с внешним миром всё ещё отвечает старый воркер. Поэтому новый сервис-воркер окажется в состоянии ожидания.

После того, как будет закрыта текущая открытая страница веб-приложения, старый сервис-воркер будет остановлен браузером, а только что установленный сервис-воркер получит полный контроль над происходящим. В этот момент будет вызвано его событие `activate`.

Зачем всё это нужно? Для того чтобы избежать проблемы наличия двух версий веб-приложения, выполняющихся одновременно, в разных вкладках браузера. Подобное, на самом деле, встречается весьма часто, и ситуация, в которой разные вкладки находятся под контролем разных веб-воркеров, способна привести к серьёзным ошибкам (например — к использованию различных схем данных при локальном хранении информации).

Удаление данных из кэша

При обработке события `activate` новой версии сервис-воркера, обычно занимаются работой с кэшем. В частности, тут удаляют старые кэши. Если сделать это раньше, на этапе установки нового воркера, старый сервис-воркер не сможет нормально работать.

Вот пример того, как можно удалить из кэша файлы, которые не были помещены в белый список (в данном случае белый список представлен записями `page-1` и `page-2`):

```
self.addEventListener('activate', function(event) {

  var cacheWhitelist = ['page-1', 'page-2'];

  event.waitUntil(
    // Получение всех ключей из кэша.
```

```
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        // Прохождение по всем кэшированным файлам.
        cacheNames.map(function(cacheName) {
          // Если файл из кэша не находится в белом списке,
          // его следует удалить.
          if (cacheWhitelist.indexOf(cacheName) === -1) {
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

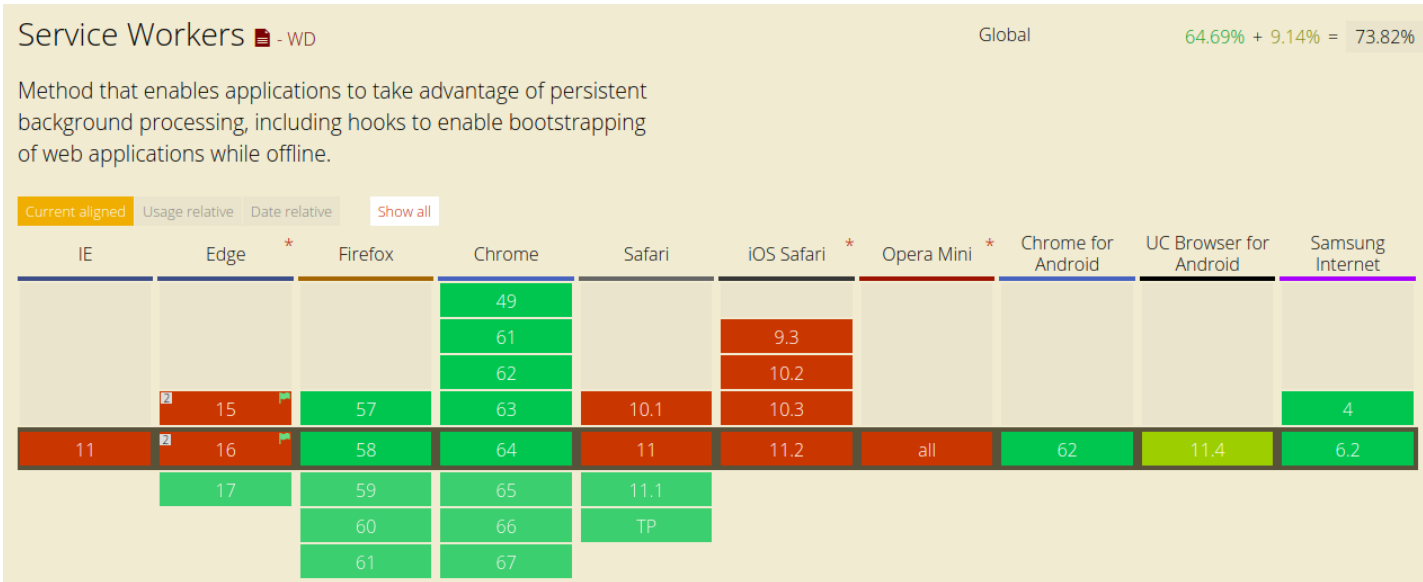
Использование HTTPS

В ходе разработки веб-приложения сервис-воркеры будут нормально работать на `localhost`, но после выпуска приложения в продакшн нужно будет использовать HTTPS (пожалуй, если вы ещё не пользуетесь HTTPS, это — весьма веская причина улучшить ситуацию).

Сервис-воркер, который не защищён HTTPS, подвержен атакам посредника, так как злоумышленник сможет перехватывать соединения и создавать фальшивые ответы на запросы приложения. Именно поэтому, для того, чтобы сделать систему безопаснее, разработчик должен регистрировать сервис-воркеры на страницах, которые обслуживаются по HTTPS. В частности, это даёт уверенность в том, что сервис-воркер, загружаемый в браузер, не был модифицирован во время передачи его кода по сети.

Поддержка в браузерах

Пока ситуация с поддержкой сервис-воркеров браузерами не идеальна, но она улучшается:



Поддержка сервис-воркеров в браузерах

Тут можно наблюдать за процессом внедрения поддержки API сервис-воркеров в браузеры.

Сценарии использования

Сервис-воркеры открывают дорогу для замечательных возможностей веб-приложений. Вот некоторые из них:

- Push-уведомления. Они позволяют пользователям настраивать периодические уведомления, поступающие из веб-приложений.
- Фоновая синхронизация. Этот механизм даёт возможность откладывать выполнение неких действий до тех пор, пока у пользователя не будет стабильного соединения с интернетом. При использовании системы фоновой синхронизации

разработчик может быть уверен в том, что если пользователь, скажем, хочет сохранить изменения документа, отредактированного в веб-приложении без доступа к сети, эти изменения не пропадут.
- Периодическая синхронизация (ожидаемая возможность). Это API, которое предоставляет функционал для управления периодической фоновой синхронизацией.
- Работа с геозонами (ожидаемая возможность). Данная возможность позволяет приложению предоставлять пользователю полезный функционал на базе его географического положения, и, в частности, основываясь на событиях попадания пользователя в заранее заданную область.

Итоги

Сервис-воркеры — перспективная технология, являющаяся базой прогрессивных веб-приложений. Эту технологию, а так же её отдельные возможности, поддерживают пока не все браузеры, но у нас есть все основания ожидать улучшения ситуации. Поэтому вполне возможно, что в обозримом будущем таланты сервис-воркеров смогут полноценно раскрыться во всех ведущих браузерах, давая разработчикам новые инструменты и идеи.

Предыдущие части цикла статей:

Часть 1: Как работает JS: обзор движка, механизмов времени выполнения, стека вызовов
Часть 2: Как работает JS: о внутреннем устройстве V8 и оптимизации кода

Часть 3: Как работает JS: управление памятью, четыре вида утечек памяти и борьба с ними
Часть 4: Как работает JS: цикл событий, асинхронность и пять способов улучшения кода с помощью async / await
Часть 5: Как работает JS: WebSocket и HTTP/2+SSE. Что выбрать?
Часть 6: Как работает JS: особенности и сфера применения WebAssembly
Часть 7: Как работает JS: веб-воркеры и пять сценариев их использования

Уважаемые читатели! Какие сценарии использования сервис-воркеров в ваших проектах кажутся вам наиболее интересными?

JavaScript-прокси: и красиво, и полезно

13

ПЕРЕВОД

ru_vds • 29 января в 12:23

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript и ужасы мутаций

48

ПЕРЕВОД

ru_vds • 19 января в 11:42

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации 2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю 0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни 33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018 70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ 40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству? 8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32