ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 40 за 2015 г.

Бухштаб Ю.А., Воробьев А.А., Евтеева Н.Н.

Организация асинхронных процессов при реализации плеера, воспроизводящего потоковый медиа контент

Рекомендуемая форма библиографической ссылки: Бухштаб Ю.А., Воробьев А.А., Евтеева Н.Н. Организация асинхронных процессов при реализации плеера, воспроизводящего потоковый медиа контент // Препринты ИПМ им. М.В.Келдыша. 2015. № 40. 14 с. URL: http://library.keldysh.ru/preprint.asp?id=2015-40

О р д е н а Л е н и н а
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
имени М.В.Келдыша
Р о с с и й с к о й а к а д е м и и н а у к

Ю.А.Бухштаб, А.А.Воробьев, Н.Н.Евтеева

Организация асинхронных процессов при реализации плеера, воспроизводящего потоковый медиа контент

Москва — 2015

Бухштаб Ю.А., Воробьев А.А., Евтеева Н.Н..

Организация асинхронных процессов при реализации плеера, воспроизводящего потоковый медиа контент

Статья связана с проблематикой разработки программных средств, обеспечивающих полнофункциональное виртуальное редактирование мультимедийной информации, сформированной из распределенных потоковых ресурсов, и воспроизведение полученного как результат этого редактирования авторского медиа контента в режиме гипервидео. При функционировании приложений такого типа эффективная работа может осуществляться только при обеспечении взаимодействия большого количества процессов, выполняемых асинхронно. В статье рассматриваются использованные при реализации клиентской программы, поддерживающей воспроизведение медиа контента, способы программирования таких процессов, основанные как на функциях обратного вызова, так и на технологии использования понятия «обещания» («Promise»).

Ключевые слова: потоковое видео, асинхронное программирование, гипервидео

Yury Alexandrovich Bukhshtab, Andrey Arturovich Vorobiov, Natalia Nikolaevna Evteeva

Organization of asynchronous processes when implementing the player reproducing streaming media content

Article is concerned with the problems of software development, providing a full-featured multimedia virtual editing information formed from the distributed streaming resources and playback authoring media content received as a result of the editing in hypervideo mode. In operation, the effective application of this type of work can be carried out only in case of interaction of a large number of processes to be performed asynchronously. This article discusses used in the implementation of the client program that supports playback of media content, methods of programming these processes, based on both the callback function and the technology of using the concept of "promises» («Promise»).

Key words: streaming video, asynchronous programming, hypervideo

Работа выполнена при поддержке Российского фонда фундаментальных исследований, проект 15-07-00970-а.

В настоящее время основными форматами для трансляции потокового видео являются видео форматы Mpeg4, Webm, Flv и аудио форматы Mp3, Ogg. Для обеспечения возможности полнофункционального виртуального редактирования медиа контента, сформированного из распределенных потоковых ресурсов, и воспроизведения результата этого редактирования в режиме гипервидео авторами настоящей статьи ранее были разработаны структуры, задающие описания последовательности воспроизводимых фрагментов потоковых данных, представленных в этих форматах на языке XML [1]. Эти структуры определяют связи и временные соотношения между различными видео и аудио потоками и их фрагментами, а также определяют последовательность воспроизводимых фрагментов с указанием URL файлов, тайм-кодов начала и конца вставляемых в поток или удаляемых из него фрагментов. Эти XML структуры могут быть считаны и разобраны программами, реализованными на JavaScript и встраиваемыми в WEB страницу. Интерпретация таких структур позволяет виртуально создавать авторский медиа контент и воспроизводить его как единое целое. Просмотр медиа контента осуществляться с помощью разработанного авторами настоящей статьи оригинального плеера. В дальнейшем будем называть этот плеер – «Медиа-Гид» (демонстрационный пример его работы можно посмотреть на странице - http://www.keldysh.ru/hypervision/demo.html). Этот плеер способен функционировать в средах всех браузеров, поддерживающих HTML5, при этом используется HTTP стриминг без предварительной полной загрузки. Для создания авторского медиа контента могут использоваться видео файлы, размещенные на любых HTTP серверах. Могут использоваться и файлы с видеохостингов, использующих собственные API (например, видео с YouTube или Vimeo). Плеер способен функционировать и на мобильных устройствах.

На языке XML кроме структур, определяющих состав и последовательность воспроизводимых видео и аудио фрагментов, также могут быть заданы структуры, интерпретация которых позволяет в заданные моменты времени выводить в определенных местах экрана дополнительные информационные элементы. И, что самое важное, плеер поддерживает технологию гипервидео, базирующуюся на понятии гиперссылки, имеющей (в отличие от традиционной гиперссылки в гипертексте) пространственные и временные характеристики. Использование технологии гипервидео позволяет объединить видео и дополнительные информационные элементы таким образом, что авторский медиа контент можно просматривать как линейно, в соответствии с заданными XML структурами порядком фрагментов, так и осуществлять навигацию в нелинейной порядке (то есть в режиме гипервидео).

В XML файле, который задает последовательность воспроизведения фрагментов видео и аудио файлов, могут содержаться описания вспомогательных видеопоследовательностей, воспроизводимых по запросу

пользователя, дополнительных текстовых и графических элементов, и определяться их активность. Элементы могут содержать произвольную информацию – текстовую, графическую или в виде HTML форм (с полями ввода, выбора, кнопками). Для визуального оформления элементов могут быть использованы все возможности HTML и CSS. Активность элемента может быть вызвана не только по клику мыши, но и другими событиями, например, перемещением мыши над элементом. В качестве реакции на действия пользователя возможно не только открытие другой WEB страницы, но и показ дополнительной информации на экране плеера или изменение информации на текущей странице. Кроме того, может поддерживаться возможность перехода на другой тайм-код в текущем клипе или показ другого клипа, с возможностью вернуться назад. В качестве реакции элементу может быть назначено несколько действий.

При наступлении события плеер может, например, выполнять следующие действия: перейти на заданный тайм-код внутри воспроизводимой видеопоследовательности, начать воспроизводить другую видеопоследовательность с возможностью вернуться в прерванное место исходной, показать на экране плеера (или странице браузера, в которую встроено окно плеера) дополнительную текстовую или графическую информацию, открыть дополнительное окно браузера, представляющее страницу любого сайта заданного URL-адресом и т. д.

Таким образом, плеер поддерживает объединение средств, обеспечивающих создание единого медиа контента, представляющего собой результат виртуального склеивания, удаления, выделения и переупорядочивания видео и аудио файлов и их фрагментов со средствами задания и воспроизведения целевого медиа контента в режиме многоуровневого гипервидео.

Такие возможности плеера «Медиа-Гид» практически могут быть реализованы при условии, что плеер поддерживает обработку многих событий, инициирующих вызовы обрабатывающих их функций, результаты выполнения которых становятся доступными в непредсказуемые моменты времени. При функционировании таких приложений эффективная работа может осуществляться только при обеспечении синхронизации между различными процессами.

Среда исполнения JavaScript программ в браузерах является однопоточной. Это означает, что любая функция, начав исполняться, не может быть прервана ничем, пока не закончится, то есть программы на JavaScript событийно исполнимые. Они состоят из функций, которые исполняются в ответ на возникновение какого либо события. Такими событиями могут быть действия пользователя: движение мыши, клик мышью по элементу на экране веб-страницы и т.д., или внутренними событиями, происходящими в браузере: окончание загрузки документа или его части, срабатывание установленного

таймера, возникновение ошибки. Функция, обрабатывающая событие, не прерывается никаким другим событием. Все события, которые происходят в этот момент, ставятся в очередь на обработку. Таким образом, все выполнение происходит последовательно в соответствии с порядком наступления событий.

Как уже отмечалось, бывают ситуации, когда функция, обрабатывающая событие, может работать слишком долго и, более того, невозможно предсказать, когда ее работа будет завершена. Например, необходимо вывести изображение заранее неизвестного размера в заданной области. Для этого необходимо загрузить оригинальное изображение, когда оно загрузится, узнать его ширину и высоту, отмасштабировать его и вывести. Но, начав загрузку изображения, нам придется дожидаться окончания его загрузки неопределенное время. Такая же ситуация с загрузкой XML документов: мы вынуждены ждать завершения загрузки документа, что бы начать его обрабатывать. Время загрузки зависит и от размера документа, и от скорости соединения с сервером. Другой пример. Нам необходимо узнать продолжительность видеофильма перед его воспроизведением. Мы задаем элементу <video> URL файла. Он отправляет запрос на сервер, чтобы считать метаданные из заголовка файла, но когда именно мы получим ответ, что бы иметь возможность узнать продолжительность видеофайла, нам неизвестно. Эти примеры показывают, что ситуации, связанные с необходимостью ожидать окончания какого либо процесса, встречаются довольно часто.

Для решения таких проблем в JavaScript существует механизм функций обратного вызова. Процессы, которые могут длиться неопределенное время, выполняются асинхронно. Их окончание (успешное или неуспешное) считаются событиями, и для обработки этих событий назначаются функции обратного вызова. Таким образом, эти функции будут вызваны средой исполнения при наступлении события - окончание процесса.

Так, загрузка изображения производится следующим образом:

```
var img = new Image();
img.onload = funload;
img.onerror = funerr;
img.src = “http://  ...url файла ... .jpg”;
```

где funload - функция, которая будет вызвана, когда изображение загрузится. Именно в ней мы сможем узнать реальные размеры изображения и провести необходимое масштабирование.

Присваивание свойству src строки, содержащей url файла, начинает асинхронный процесс загрузки.

Для объектов, являющихся HTML элементами страницы, назначение функций обратного вызова производится стандартными методами назначения обработчиков событий.

```
var video=document.getElementById (“video_box”);
var funmetaload = function () {
  len = video.duration;
};

var ferr = function () {
  // обработка ошибки
};

video.addEventLisenter(“metadata”, funmetaload);
video.addEventLisenter(“error”, ferr);
video.src = “http://  ...url файла ... .mp4”;
// дальнейшее выполнение
```

Здесь мы получаем ссылку на видеоэлемент с идентификатором «video_box» на HTML странице. С помощью метода addEventLisenter назначаем обработчики событий получения мета-данных и ошибки. Присваиванием свойству src url файла, начинаем соединение с сервером, чтобы получить мета-данные - асинхронный процесс. И продолжаем выполнение дальше. Когда данные будут получены (разумеется, после обработки всех событий, которые наступят до этого момента), среда браузера вызовет нашу функцию funmetaload, или функцию ferr, если что-то пойдет не так.

Несмотря на различные способы задания, суть одна - мы назначаем функцию обратного вызова, которая будет вызвана исполнительной средой JavaScript, когда закончится (успешно или неуспешно) запущенный асинхронный процесс. Необходимо обратить внимание, что назначение функций обратного вызова необходимо произвести до запуска процесса и ни в коем случае не после. Хотя мы и предполагаем, что процесс будет асинхронным и будет продолжаться какое-то время, но это не всегда так. Например, изображение может не загружаться с сервера, а быть взято из кэша браузера. В этом случае процесс выполнится синхронно и закончится до назначения функции обратного вызова, которая при назначении после окончания загрузки уже не будет исполняться.

При разработке и программировании плеера «Медиа-Гид» с подобными асинхронными процессами приходится иметь дело очень часто. Например:

- загрузка XML или JSON файлов, с заданием на воспроизведение;
- загрузка файлов YouTube API, если в задании присутствуют видеопоследовательности с сервиса YouTube;
- соединение со всеми видеофайлами, указанными в задании, чтобы убедиться, что они все доступны для воспроизведения и для определения их продолжительности;
- считывание файлов с изображениями для формирования дополнительных элементов.

Все эти процессы являются асинхронными. Функции обратного вызова позволят программировать подобные процессы. Но посмотрим, как это приходится делать. Рассмотрим начало работы плеера «Медиа-Гид» в укрупненных блоках. Необходимо совершить следующие действия:

- загрузить файл, определяемый заданным URL;
- выполнить разбор файла (парсинг), определить, сколько фрагментов и из каких видео файлов используются;
- попытаться соединиться со всеми файлами для определения их продолжительности;
- запустить результирующую видеопоследовательность на воспроизведение.

Если бы все происходило синхронно, то мы могли бы написать примерно такой код:

```
content = loadFile (url);          // загрузка файла
fragms = parse (content);          // разбор его содержимого
                                   // и определение фрагментов
connect (fragms);                  // соединение со всеми фрагментами
startPlay();                       // запуск на воспроизведение.
```

Но в реальности процессы являются асинхронными. Невозможно начать разбор, пока не завершится загрузка, нельзя производить соединение, пока не закончится разбор (это тоже может быть асинхронным процессом, так на этапе разбора определяется, нужно ли загружать файлы YouTube API, и производится их загрузка), нельзя начать воспроизведение, пока не произойдет соединение со всеми видео файлами, используемыми в данной видеопоследовательности.

Если воспользоваться функциями обратного вызова, то код получится примерно такой:

```
function onConnect() {             // функция вызывается
                                   // при завершении соединений
   startPlay();                    // запуск на воспроизведение
};
function onParse (fragms) {        // функция вызывается
                                   // при завершении разбора
   connect(fragms, onConnect)      // соединение со всеми фрагментами
};
function onLoad (content){         // функция вызывается
                                   // при завершении загрузки
   parse(content, onParse)         // разбор его содержимого
                                   // и определение фрагментов
};
loadFile(url, onLoad);             // загрузка файла
```

Последовательность кода (конечно, не самих действий) у нас стала обратной. И его понятность уменьшилась. Разумеется, JavaScript допускает

написание функций и в другой последовательности, лишь бы все они были определены до реального вызова, но ясность в последовательности действий все равно теряется. Обратные вызовы могут быть вложенными, что делают программы очень трудно отлаживаемыми. Кроме того, надо иметь в виду, что на каждом этапе могут возникнуть различные ошибки, для обработки которых требуется написание своих функций обратного вызова. Ошибочные ситуации могут встречаться в различных местах. Например, при загрузке XML или JSON файлов задания ошибка может произойти в следующих ситуациях:

- отсутствии Интернет соединения;
- недоступности указанного сервера;
- отсутствии заданного файла на сервере;
- синтаксической ошибке в файле.

Выявляются такие ошибки различными методами. Если ошибки, связанные с работой Интернета и наличием сервера и файла, можно обнаружить с помощью задания функции обратного вызова объекту XMLHttpRequest, который осуществляет считывание файла, то синтаксическую ошибку в JSON файле, подключаемом как элемент <script>, можно обнаружить только с помощью блока try - catch языка JavaScript. Это приводит к тому, что приходится использовать различные функции для обработки различных ошибок в одном логическом действии. Программирование самих этих укрупненных блоков так же связано с асинхронными процессами. Так, функция connect устанавливает множество асинхронных процессов соединения с медиа файлами и может считаться завершенной, только когда завершатся процессы соединения со всеми файлами. Асинхронные процессы возникают и при воспроизведении результирующей видеопоследовательности. Например, если фрагмент состоит из видео потока с наложенным на него аудио потоком, то для начала воспроизведения фрагмента необходимо установить каждый из потоков на требуемый тайм-код, дождаться окончания буферизации обоих потоков и только после этого запустить фрагмент на воспроизведение. Использование функций обратного вызова позволяет программировать асинхронные процессы, но код программы становится довольно запутанным, изобилует множеством функций. Часто бывает трудно отследить основную последовательность действий. Все это затрудняет отладку и дальнейшую модификацию программы.

В современных реализациях JavaScript появилась технология, стандартизирующая и облегчающая работу с асинхронными процессами. Эта технология использует понятие «обещания» (Promise). Совсем новой ее назвать нельзя. В различных модификациях она уже довольно давно использовалась в различных пакетах, таких как jQuery, Dojo и других. В некоторых пакетах технология такого типа называлась deferred. Но сейчас она введена в готовящийся к выходу в ближайшее время новый стандарт языка (ECMA-262 edition 6) и уже реализована в последних версиях некоторых браузеров (Firefox 29+, Chrome 32+, Safari 7.2+) в виде встроенных в библиотеку ядра нативных

функций [2]. Для других ситуаций существует эффективная реализация в виде функций на JavaScript. За основу в стандарте был взята спецификация Promises/A+ [3], которая может не полностью совпадать с другими, уже существующими программными реализациями.

Основой данной технологии является объект обещания. Этот объект содержит или будет содержать некоторое значение, полученное в результате асинхронной операции. Этот объект может находиться в трех состояниях:

- pending - обещание еще не выполнено (асинхронная операция еще не завершилась);
- fulfilled - обещание выполнено, результат получен;
- reject - обещание отклонено (операция завершилась с ошибкой), в качестве результата есть причина ошибки.

Результатом может быть любое значение языка Javascript (число, строка, объект) Причиной тоже может быть любое значение, но чаще всего это встроенный объект Error.

Создание объекта обещания производится функцией конструктором Promise:

```
var myprom = new Promise (asyncfun);
```

Параметр asyncfun – это наша функция, которая выполняет какую-либо асинхронную операцию. Она должна иметь вид

```
function asyncfun (resolve, reject) {....
  resolve (value)
  ….
  reject (error)
  ….
}
```

Функция должна иметь два параметра, которые являются функциями обратного вызова, предоставляемыми функцией Promise. Функцию resolve мы должны вызвать, когда асинхронная операция закончилась, и мы можем предоставить ее результат для дальнейшей обработки. Функцию reject мы должны вызвать, когда произошла какая-либо ошибка, и мы передаем некоторое значение, описывающее ситуацию. Кроме того, любая ошибка, произошедшая во время выполнения нашей асинхронной функцией и не отловленная блоком try – catch, или выброс исключения с помощью оператора throw эквивалентно вызову функции reject со значением объекта ошибки.

После того, как объект обещания создан, нам надо использовать его результат. Для этого служит метод then объекта обещания. Он вызывается следующим образом:

```
myprom.then (fres, ferr);
```

где fres - функция обратного вызова, которая будет вызвана, когда и если обещание исполнено. Эта функция может иметь один параметр – результат обещания (который наша функция asyncfun передала, вызвав resolve).

ferr - функция обратного вызова, которая будет вызвана, когда и если обещание отклонено. Эта функция может иметь один параметр – причину (который наша функция asyncfun передала, вызвав reject). Параметр ferr не обязательный. Если он не указан, то причина отклонения не будет обработана в этом вызове then.

Методу then совершенно неважно, синхронно или асинхронно выполнялась функция. Завершилась ли уже асинхронная операция или нет. Он использует результат, если тот уже был получен или будет использовать результат, когда он будет получен.

Как представляется, разница с методами, использующими только функции обратного вызова, не очень большая. Но важно то, что метод then сам создает объект обещания. И функции fres (и ferr) сами могут быть асинхронными. Если функция fres асинхронная, то она должна вернуть свое обещание, которое будет использовано методом then. К этому обещанию так же можно применить метод then, который будет ждать исполнения этого обещания. Таким образом, методы then можно выстраивать в цепочки:

new Promise (async1) .then (async2) .then (async3) .then (finish).

Здесь мы запускаем первый асинхронный процесс async1. Когда он завершится, первый метод then запустит функцию async2. Эта функция должна вернуть свое обещание, тогда второй метод then будет ждать его исполнения, перед тем как запустить функцию async3, и т.д.

В этом примере не учитывается возможность ошибочных ситуаций. Если в методе then вызывается функция, заданная параметром ferr, то дальнейшее поведение зависит от того, какое значение она вернет. Если функция вернет встроенный объект Error или обещание, которое будет отклонено, то метод then создаст новое, отклоненное обещание и последующий метод then так же будет вызывать функцию, заданную своим параметром ferr. Если функция ferr вернет какое-либо другое значение, то считается, что она обработала и исправила ошибку, и дальнейший вызов then будет вызывать функцию fres с этим значением в качестве результата. Если у метода then не задана функция ferr, а обещание отклоняется, то then создаст отклоненное обещание и будет вызвана функция ferr следующего в цепочке метода then.

Кроме then, у объекта обещания есть метод catch, который позволяет задать только функцию обработки отклоненного обещания. Обычно этот метод используется в конце цепочек then и задает единый обработчик всех ошибок, на каком бы этапе они не произошли.

Добавим в предыдущий пример обработку ошибок

```
new Promise (async1)
.then (async2)
.then (async3, ferr1)
.then (finish)
.catch (msgerr).
```

В этом примере, если ошибка произойдет на этапе выполнения функций async1 или async2, то обработка ошибки будет производиться функцией ferr1. Эта функция может попытаться «исправить» ошибку и вернуть нормальный результат, тогда дальше будет выполнена функция finish. Если ferr1 вернет состояние ошибки или ошибка произойдет при выполнении async3 или finish, то ее обработка будет производиться функцией msgerr.

Вернемся теперь к работе плеера «Медиа-Гид». Если функции, которые загружают файл, производят его разбор и соединения с фрагментами, переписать так, что бы они возвращали не результат, а обещание результата, то основная последовательность действий может быть записана так

```
loadFile (url)          // загрузка файла
.then (parse)           // разбор его содержимого и определение фрагментов
.then (connect)         // соединение со всеми фрагментами
.then (startPlay)       // запуск на воспроизведение
.catch (msgErr);        // сообщение об ошибке
```

Каждая асинхронная функция должна возвращать не результат своего действия, а создать объект обещания, выполнить асинхронную часть внутри этого объекта и вернуть полученный результат с помощью функций обратного вызова. Этот результат будет передан в качестве параметра следующей функции в методе then.

Функция Promise имеет ряд полезных «статических» методов. Это методы не объекта, созданного Promise, а самой функции.

Метод resolve создает изначально выполненное обещание:

```
var prom = Promise.resolve (value);     //создаем выполненное обещание
                                        // с результатом value
prom.then (fres,  ferr) ;               // всегда будет вызвана fres
                                        // с параметром value
```

Метод reject создает изначально отклоненное обещание.

```
var prom = Promise. reject (value);     // создаем отклоненное обещание
                                        // с результатом value
prom.then (fres,  ferr) ;     // всегда будет вызвана ferr с параметром value
```

Метод all создает обещание из массива обещаний, которое будет выполнено, когда и если будут выполнены все обещания из переданного массива.

```
var promarr = [ ];
promarr[0] =  new Promise (async1);
promarr[1]  =  new Promise (async2);
promarr[2] = new Promise (async3);
var myprom = Promise.all (promarr);
myprom.then (fres, ferr);
```

Создаем массив promarr из трех обещаний. Каждый элемент массива содержит обещание результата асинхронной операции. Обещание myprom будет выполнено и будет вызвана функция fres, только когда выполнятся обещания всех асинхронных операций (async1, async2 и async3). В качестве параметра функция fres получит массив из результатов всех обещаний. Функция ferr будет вызвана, если хотя бы одно из обещаний будет отклонено.

Этот метод удобно применять, когда у нас есть множество асинхронных процессов, которые могут исполняться параллельно и независимо друг от друга, а нам необходимо дождаться, когда они все закончатся. Например, функция connect получает на вход массив фрагментов, из которых состоит видеопоследовательность. Она должна установить соединение с ними, если это возможно. Соединение с каждым фрагментом - независимый процесс, и все они могут выполняться параллельно. Функция connect может быть записана следующим образом

```
function connect (fragms) {
  var proms = [ ],
  for (var i = 0; i < fragms.length; i++) {
          proms.push ( connectFragm (fragms[i]) );
  }
  return Promise.all (proms);
}
```

Здесь мы предполагаем, что у нас имеется функция connectFragm, которая возвращает обещание соединения с одним фрагментом. Мы создаем массив и заполняем его обещаниями от каждого из фрагментов. И возвращаем обещание соединения со всеми фрагментами.

Конечно, использование технологии обещаний не избавляет нас от необходимости использовать функции обратного вызова, собственно в ней самой они используются постоянно. Но она позволяет переносить технические детали на более низкий уровень. И на каждом уровне код программы становится более прямым и логичным. Каждая функция, где используется асинхронная операция, оформляется единым образом, единообразно строятся ожидания окончания этих операций, что существенно облегчает отладку и дальнейшее сопровождение.

Технология обещаний позволяет описывать последовательность асинхронных действий, не думая об их асинхронности. Код программы выглядит так, как будто эти действия выполняются синхронно и последовательно. Но довольно часто возникает обратная задача: необходимо асинхронно выполнить действия, которые таковыми не являются.

При обработке событий часто встречается ситуация, когда необходимо выполнить какое то «тяжеловесное» действие, которое может занять довольно продолжительное время. Но это нежелательно делать во время обработки этого события, так как другие события в это время будут стоять в очереди на

обработку. К тому же результат выполнения этого действия нас в данный момент не интересует – его просто надо выполнить.

Хотелось бы просто указать, что эти действия нужно выполнить как можно скорее, но потом, когда будет обработано текущее событие и другие, уже стоящие в очереди. То есть, как при асинхронном событии, инициировать действие, но, не ожидая его окончания, идти дальше. Если нам не нужен результат этого действия, то этим можно и ограничиться. Если это действие может вернуть какой-то результат, который нам может понадобиться позже, то можно использовать для него обещание, как для обычной асинхронной операции.

Для решения подобной проблемы в JavaScript имеется функция setImmediate [4]. Пока она реализована только в браузере Internet Explorer 10+, и даже не включена в стандарты HTML5 и JavaScript 6, но, в силу ее важности, существует большое количество программных реализаций для других браузеров. Функция setImmediate очень похожа на функцию setTimeout, которая позволяет установить таймер на определенное время и по срабатыванию таймера вызвать указанную функцию. Только в setImmediate не указывается время срабатывания таймера. Вызов функции осуществляется как можно быстрее, когда освободится очередь обработки событий. Функция имеет вид:

setImmediate (fun [, arg1, …, argN]);

Здесь fun – функция, которую необходимо будет вызвать, необязательные параметры arg1 … argN – список параметров, которые будут переданы функции при ее вызове.

В старых версиях браузеров для подобных целей использовалась функция setTimeout с нулевой задержкой. Однако таймеры браузера реально не могут устанавливать интервалы меньше 4 мс (16 мс для некоторых версий). Это может оказаться слишком большим временем. В современных браузерах для программной реализации setImmediate используется посылка окном браузера сообщения самому себе, используя postMessage. Задержка в этом случае практически нулевая. Тесты показывают, что программная реализация setImmediate работает в несколько десятков раз быстрее, чем с использованием setTimeout.

При реализации плеера «Медиа-Гид» такая техника асинхронного вызова используется часто. Например, когда гипермедиа плеер начинает воспроизведение очередного фрагмента, желательно подготовить к воспроизведению следующий фрагмент, чтобы время переключения между ними было минимальным. Это довольно «тяжеловесная» операция. Однако нам требуется как можно быстрее запустить текущий фрагмент, чтобы получать и обрабатывать события от таймера, позволяющие в нужные моменты высвечивать на экране дополнительные элементы, а если в текущем фрагменте заданы видео и аудио потоки, то следить за их синхронизацией. Рассмотрим еще следующие примеры.

В плеере имеется основной временной цикл. При воспроизведении видеопоследовательности каждые 20 мс срабатывает таймер и вызывается функция, которая проверяет, не закончился ли текущий фрагмент, не надо ли показать одни дополнительные элементы и погасить другие, передает информацию контролеру (часть программы, которая отвечает за работу панели управления), чтобы он визуализировал информацию о текущем состоянии. Некоторые из этих операций могут занять продолжительное время (например, поиск среди дополнительных элементов тех, которые надо высветить, и тех, которые необходимо погасить). Чтобы не занимать очередь обработки событий, подобные операции удобнее выполнить асинхронно.

Выполнение клика на дополнительном элементе может привести к множеству действий, связанных с этим событием для данного элемента. Может потребоваться высветить или убрать другие элементы, внести изменения на странице, в окружении которой работает плеер. Эти действия синхронны, но их совершенно не обязательно выполнять непосредственно при обработке данного события.

Литература

1. Бухштаб Ю.А., Воробьев А.А., Евтеева Н.Н. Интерактивные возможности управления потоковым видео в среде HTML5 // Препринты ИПМ им. М.В.Келдыша. 2013. № 36. 19 с.
URL: http://library.keldysh.ru/preprint.asp?id=2013-36

2. Mozilla Developer Network (MDN). Promise. – URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/

3. Promises/A+ organization. – URL: https://promisesaplus.com/

4. Internet Explorer Dev Center. – URL: https://msdn.microsoft.com/en-us/library/ie/hh773176%28v=vs.85%29.aspx