Future Of Streams Brainstorm

BlinkOn 9

Adam Rice, ricea@chromium.org

Coming Up

1. Recap
2. Coming soon
3. Under discussion
    a. Streaming HTML into DOM
    b. Streams ❤️ Workers: Transferable Streams
    c. Compressed Upload: GZIPStream
    d. Stream to element: srcObj
    e. Opaque Streams

Recap

Concepts

- Streaming data
- Chunks can be anything
  - Bytes (Uint8Array)
  - Text (string)
  - Objects
- Backpressure
- Piping
  - pipeTo and pipeThrough

ReadableStream

- Shipped in M43 (also in Edge & Safari)

```
const reader = response.body.getReader();
while (true) {
  const {value, done} = await reader.read();
  if (done) break;
  // do something with |value|.
}
```

WritableStream

- Shipped in M59 (also in Edge & Safari)

```
response.body.pipeTo(fileStream);
```

TransformStream

- Shipping in M67

```
stringStream.pipeThrough(splitStream('\n'))
            .pipeThrough(parseJSON())
            .pipeTo(dataSink);
```

Coming Soon

TextDecoderStream

- Converts chunks of bytes to strings
- Every time you stream text from fetch() you need this.

```
const response = await fetch('data');
response.body
    .pipeThrough(new TextDecoderStream())
    .pipeTo(dom);
```

- Also TextEncoderStream

Async iterator support

```
for await (const value of response.body) {
  // do something with |value|.
}
```

Under Discussion

Streaming HTML into DOM

Streaming HTML

```
await response.body
      .pipeThrough(new TextDecoderStream())
      .pipeTo(div.writable);
```

- Concrete design work starting this quarter
- A lot of interest

Streams  Workers: Transferable Streams

Transfer using postMessage

- Original stream becomes locked
- Destination receives a stream that proxies to the original
- Underlying sink, source, or transformer still execute in the original context
- Chunks are cloned or transferred automatically
- Like a generalisation of ServiceWorker respondWith()

In the page:

```
const worker = new Worker('transcode.js');
worker.onmessage = event => {
  const transcoder = event.data;
  await fetch('bunny.vp10')
      .pipeThrough(transcoder)
      .pipeTo(videoSink);
};
```

In the worker:

```
importScripts('vp10decode.js', 'mp4encode.js');
const transcoder = new TransformStream({
  transform(chunk, controller) {
    const decoded = vp10decode(chunk);
    controller.enqueue(mp4encode(decoded));
  }
});
postMessage(transcoder, [transcoder]);
```

In the page:

```
const worker = new Worker('transcode.js');
worker.onmessage = event => {
  const transcoder = event.data;
  await fetch('bunny.vp10')
      .pipeThrough(transcoder)
      .pipeTo(videoSink);
};
```

Many other ways of using it

For example, using a TransformStream to bridge a ServiceWorker returning a response and the Worker that generates the content:

https://gist.github.com/domenic/ea5ebedffcee27f552e103963cf8585c/

Open questions

- Can setting up an off-thread TransformStream be done with less boilerplate?
  - This is a convenient primitive to exploit parallelism, but the code we had to write to set it up was not convenient.

Open questions (cnt.)

- Given this chunk:

```
{
  timeCode: 23423,
  frame: <binary data>
}
```

Will “frame” be copied, or transferred?

Open questions (cnt.)

- Handling of invalid chunks
- Backpressure is weird
  - `size` function cannot be transferred
  - Optimal `highWaterMark` may depend on environmental factors
- Worker-friendly integration with Media Source Extensions

Compressed Upload: GZIPStream

GZIPStream

- “The” solution to the compressed upload problem.

```
const gz = file.pipeThrough(new GZIPStream());
fetch('/upload', {
  method: 'POST',
  body: gz,
  headers: {
    'Content-Type': 'application/gzip'
  }
});
```

Stream to element: srcObj

Like src, but for a stream

```
const img = new Image();
const response = new Response(streamingBody);
img.srcObj = response;
div.appendChild(img);
```

- Can be polyfilled with ServiceWorker.
- In many cases you will need to explicitly set Content-Type
- Any body type that the Response constructor accepts will work. For example, ArrayBuffer and Blob.

Opaque Streams

Opaque responses, generalised

- Stream pipes are designed to bypass JavaScript for performance
- But they can also bypass JavaScript for security
- A source can authenticate a sink, and error if it is not approved to receive data
- Authors can combine streams in novel ways without ever being able to see the data they contain
- Getting the security right is challenging

Strawman example

```
const response =
    await fetch('https://other/vid.mp4',
                {mode: 'no-cors'});
const transform = new VideoThumbnailExtractor();
const imgBody =
    response.body.pipeThrough(transform);
const img = new Image();
img.objsrc = new Reponse(imgBody);
```

Aftermatter

This presentation

https://goo.gl/1nbc4T

Links

Streams Standard: https://streams.spec.whatwg.org/

Demos: https://streams.spec.whatwg.org/demos/

Jake Archibald's guide to async iterators: https://jakearchibald.com/2017/async-iterators-and-generators/