因为是使用 0 来初始化 SharedArrayBuffer，所以每个工作线程都会到达 Atomics.wait()并停止执行。在停止状态下，执行线程存在于一个**等待队列**中，在经过指定时间或在相应索引上调用 Atomics.notify()之前，一直保持暂停状态。1000 毫秒之后，顶部执行上下文会调用 Atomics.notify()释放其中一个等待的线程。这个线程执行完毕后会再次调用 Atomics.notify()释放另一个线程。这个过程会持续到所有线程都执行完毕并通过 postMessage()传出最终的值。

Atomics API 还提供了 Atomics.isLockFree()方法。不过我们基本上应该不会用到。这个方法在高性能算法中可以用来确定是否有必要获取锁。规范中的介绍如下：

> Atomics.isLockFree()是一个优化原语。基本上，如果一个原子原语（compareExchange、load、store、add、sub、and、or、xor 或 exchange）在 *n* 字节大小的数据上的原子步骤在不调用代理在组成数据的 *n* 字节之外获得锁的情况下可以执行，则 Atomics.isLockFree(n)会返回 true。高性能算法会使用 Atomics.isLockFree 确定是否在关键部分使用锁或原子操作。如果原子原语需要加锁，则算法提供自己的锁会更高效。
>
> Atomics.isLockFree(4)始终返回 true，因为在所有已知的相关硬件上都是支持的。能够如此假设通常可以简化程序。

20.2　跨上下文消息

跨文档消息，有时候也简称为 XDM（cross-document messaging），是一种在不同执行上下文（如不同工作线程或不同源的页面）间传递信息的能力。例如，www.wrox.com 上的页面想要与包含在内嵌窗格中的 p2p.wrox.com 上面的页面通信。在 XDM 之前，要以安全方式实现这种通信需要很多工作。XDM 以安全易用的方式规范化了这个功能。

> **注意**　跨上下文消息用于窗口之间通信或工作线程之间通信。本节主要介绍使用 postMessage()与其他窗口通信。关于工作线程之间通信、MessageChannel 和 BroadcastChannel，可以参考第 27 章。

XDM 的核心是 postMessage()方法。除了 XDM，这个方法名还在 HTML5 中很多地方用到过，但目的都一样，都是把数据传送到另一个位置。

postMessage()方法接收 3 个参数：消息、表示目标接收源的字符串和可选的可传输对象的数组（只与工作线程相关）。第二个参数对于安全非常重要，其可以限制浏览器交付数据的目标。下面来看一个例子：

```
let iframeWindow = document.getElementById("myframe").contentWindow;
iframeWindow.postMessage("A secret", "http://www.wrox.com");
```

最后一行代码尝试向内嵌窗格中发送一条消息，而且指定了源必须是"http://www.wrox.com"。如果源匹配，那么消息将会交付到内嵌窗格；否则，postMessage()什么也不做。这个限制可以保护信息不会因地址改变而泄露。如果不想限制接收目标，则可以给 postMessage()的第二个参数传"*"，但不推荐这么做。

接收到 XDM 消息后，window 对象上会触发 message 事件。这个事件是异步触发的，因此从消息发出到接收到消息（接收窗口触发 message 事件）可能有延迟。传给 onmessage 事件处理程序的 event

对象包含以下 3 方面重要信息。

- data：作为第一个参数传递给 postMessage()的字符串数据。
- origin：发送消息的文档源，例如"http://www.wrox.com"。
- source：发送消息的文档中 window 对象的代理。这个代理对象主要用于在发送上一条消息的窗口中执行 postMessage()方法。如果发送窗口有相同的源，那么这个对象应该就是 window 对象。

接收消息之后验证发送窗口的源是非常重要的。与 postMessage()的第二个参数可以保证数据不会意外传给未知页面一样，在 onmessage 事件处理程序中检查发送窗口的源可以保证数据来自正确的地方。基本的使用方式如下所示：

```
window.addEventListener("message", (event) => {
  // 确保来自预期发送者
  if (event.origin == "http://www.wrox.com") {

    // 对数据进行一些处理
    processMessage(event.data);
    // 可选：向来源窗口发送一条消息
    event.source.postMessage("Received!", "http://p2p.wrox.com");
  }
});
```

大多数情况下，event.source 是某个 window 对象的代理，而非实际的 window 对象。因此不能通过它访问所有窗口下的信息。最好只使用 postMessage()，这个方法永远存在而且可以调用。

XDM 有一些怪异之处。首先，postMessage()的第一个参数的最初实现始终是一个字符串。后来，第一个参数改为允许任何结构的数据传入，不过并非所有浏览器都实现了这个改变。为此，最好就是只通过 postMessage()发送字符串。如果需要传递结构化数据，那么最好先对该数据调用 JSON.stringify()，通过 postMessage()传过去之后，再在 onmessage 事件处理程序中调用 JSON.parse()。

在通过内嵌窗格加载不同域时，使用 XDM 是非常方便的。这种方法在混搭（mashup）和社交应用中非常常用。通过使用 XDM 与内嵌窗格中的网页通信，可以保证包含页面的安全。XDM 也可以用于同源页面之间通信。

20.3 Encoding API

Encoding API 主要用于实现字符串与定型数组之间的转换。规范新增了 4 个用于执行转换的全局类：TextEncoder、TextEncoderStream、TextDecoder 和 TextDecoderStream。

> **注意** 相比于批量（bulk）的编解码，对流（stream）编解码的支持很有限。

20.3.1 文本编码

Encoding API 提供了两种将字符串转换为定型数组二进制格式的方法：批量编码和流编码。把字符串转换为定型数组时，编码器始终使用 UTF-8。

1. 批量编码

所谓**批量**，指的是 JavaScript 引擎会同步编码整个字符串。对于非常长的字符串，可能会花较长时间。批量编码是通过 TextEncoder 的实例完成的：

```
const textEncoder = new TextEncoder();
```

这个实例上有一个 encode()方法，该方法接收一个字符串参数，并以 Uint8Array 格式返回每个字符的 UTF-8 编码：

```
const textEncoder = new TextEncoder();
const decodedText = 'foo';
const encodedText = textEncoder.encode(decodedText);

// f 的 UTF-8 编码是 0x66（即十进制 102）
// o 的 UTF-8 编码是 0x6F（即二进制 111）
console.log(encodedText); // Uint8Array(3) [102, 111, 111]
```

编码器是用于处理字符的，有些字符（如表情符号）在最终返回的数组中可能会占多个索引：

```
const textEncoder = new TextEncoder();
const decodedText = '☺';
const encodedText = textEncoder.encode(decodedText);

// ☺的 UTF-8 编码是 0xF0 0x9F 0x98 0x8A（即十进制 240、159、152、138）
console.log(encodedText); // Uint8Array(4) [240, 159, 152, 138]
```

编码器实例还有一个 encodeInto()方法，该方法接收一个字符串和目标 Unit8Array，返回一个字典，该字典包含 read 和 written 属性，分别表示成功从源字符串读取了多少字符和向目标数组写入了多少字符。如果定型数组的空间不够，编码就会提前终止，返回的字典会体现这个结果：

```
const textEncoder = new TextEncoder();
const fooArr = new Uint8Array(3);
const barArr = new Uint8Array(2);
const fooResult = textEncoder.encodeInto('foo', fooArr);
const barResult = textEncoder.encodeInto('bar', barArr);

console.log(fooArr);    // Uint8Array(3) [102, 111, 111]
console.log(fooResult); // { read: 3, written: 3 }

console.log(barArr);    // Uint8Array(2) [98, 97]
console.log(barResult); // { read: 2, written: 2 }
```

encode()要求分配一个新的 Unit8Array，encodeInto()则不需要。对于追求性能的应用，这个差别可能会带来显著不同。

> **注意**　文本编码会始终使用 UTF-8 格式，而且必须写入 Unit8Array 实例。使用其他类型数组会导致 encodeInto()抛出错误。

2. 流编码

TextEncoderStream 其实就是 TransformStream 形式的 TextEncoder。将解码后的文本流通过管道输入流编码器会得到编码后文本块的流：

```
async function* chars() {
  const decodedText = 'foo';
    for (let char of decodedText) {
      yield await new Promise((resolve) => setTimeout(resolve, 1000, char));
```

```
  }
}

const decodedTextStream = new ReadableStream({
  async start(controller) {
    for await (let chunk of chars()) {
      controller.enqueue(chunk);
    }

    controller.close();
  }
});

const encodedTextStream = decodedTextStream.pipeThrough(new TextEncoderStream());

const readableStreamDefaultReader = encodedTextStream.getReader();

(async function() {
  while(true) {
    const { done, value } = await readableStreamDefaultReader.read();

    if (done) {
      break;
    } else {
      console.log(value);
    }
  }
})();

// Uint8Array[102]
// Uint8Array[111]
// Uint8Array[111]
```


20.3.2 文本解码

Encoding API 提供了两种将定型数组转换为字符串的方式：批量解码和流解码。与编码器类不同，在将定型数组转换为字符串时，解码器支持非常多的字符串编码，可以参考 Encoding Standard 规范的“Names and labels”一节。

默认字符编码格式是 UTF-8。

1. 批量解码

所谓**批量**，指的是 JavaScript 引擎会同步解码整个字符串。对于非常长的字符串，可能会花较长时间。批量解码是通过 `TextDecoder` 的实例完成的：

```
const textDecoder = new TextDecoder();
```

这个实例上有一个 `decode()`方法，该方法接收一个定型数组参数，返回解码后的字符串：

```
const textDecoder = new TextDecoder();

// f 的 UTF-8 编码是 0x66 (即十进制 102)
// o 的 UTF-8 编码是 0x6F (即二进制 111)
const encodedText = Uint8Array.of(102, 111, 111);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText); // foo
```

解码器不关心传入的是哪种定型数组，它只会专心解码整个二进制表示。在下面这个例子中，只包含 8 位字符的 32 位值被解码为 UTF-8 格式，解码得到的字符串中填充了空格：

```
const textDecoder = new TextDecoder();

// f 的 UTF-8 编码是 0x66（即十进制 102）
// o 的 UTF-8 编码是 0x6F（即二进制 111）
const encodedText = Uint32Array.of(102, 111, 111);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText);  // "f   o   o   "
```

解码器是用于处理定型数组中分散在多个索引上的字符的，包括表情符号：

```
const textDecoder = new TextDecoder();

// ☺的 UTF-8 编码是 0xF0 0x9F 0x98 0x8A（即十进制 240、159、152、138）
const encodedText = Uint8Array.of(240, 159, 152, 138);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText); // ☺
```

与 TextEncoder 不同，TextDecoder 可以兼容很多字符编码。比如下面的例子就使用了 UTF-16 而非默认的 UTF-8：

```
const textDecoder = new TextDecoder('utf-16');

// f 的 UTF-8 编码是 0x0066（即十进制 102）
// o 的 UTF-8 编码是 0x006F（即二进制 111）
const encodedText = Uint16Array.of(102, 111, 111);
const decodedText = textDecoder.decode(encodedText);

console.log(decodedText); // foo
```

2. 流解码

TextDecoderStream 其实就是 TransformStream 形式的 TextDecoder。将编码后的文本流通过管道输入流解码器会得到解码后文本块的流：

```
async function* chars() {
  // 每个块必须是一个定型数组
  const encodedText = [102, 111, 111].map((x) => Uint8Array.of(x));

  for (let char of encodedText) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, char));
  }
}

const encodedTextStream = new ReadableStream({
  async start(controller) {
    for await (let chunk of chars()) {
      controller.enqueue(chunk);
    }

    controller.close();
  }
});

const decodedTextStream = encodedTextStream.pipeThrough(new TextDecoderStream());
```

```
const readableStreamDefaultReader = decodedTextStream.getReader();

(async function() {
  while(true) {
    const { done, value } = await readableStreamDefaultReader.read();

    if (done) {
      break;
    } else {
      console.log(value);
    }
  }
})();

// f
// o
// o
```

文本解码器流能够识别可能分散在不同块上的代理对。解码器流会保持块片段直到取得完整的字符。比如在下面的例子中，流解码器在解码流并输出字符之前会等待传入 4 个块：

```
async function* chars() {
  // ☺的 UTF-8 编码是 0xF0 0x9F 0x98 0x8A（即十进制 240、159、152、138）
  const encodedText = [240, 159, 152, 138].map((x) => Uint8Array.of(x));

  for (let char of encodedText) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, char));
  }
}

const encodedTextStream = new ReadableStream({
  async start(controller) {
    for await (let chunk of chars()) {
      controller.enqueue(chunk);
    }
    controller.close();
  }
});

const decodedTextStream = encodedTextStream.pipeThrough(new TextDecoderStream());

const readableStreamDefaultReader = decodedTextStream.getReader();

(async function() {
  while(true) {
    const { done, value } = await readableStreamDefaultReader.read();

    if (done) {
      break;
    } else {
      console.log(value);
    }
  }
})();

// ☺
```

文本解码器流经常与 `fetch()`一起使用，因为响应体可以作为 `ReadableStream` 来处理。比如：

```
const response = await fetch(url);
const stream = response.body.pipeThrough(new TextDecoderStream());
const decodedStream = stream.getReader()

for await (let decodedChunk of decodedStream) {
  console.log(decodedChunk);
}
```

20.4　File API 与 Blob API

Web 应用程序的一个主要的痛点是无法操作用户计算机上的文件。2000 年之前，处理文件的唯一方式是把<input type="file">放到一个表单里，仅此而已。File API 与 Blob API 是为了让 Web 开发者能以安全的方式访问客户端机器上的文件，从而更好地与这些文件交互而设计的。

20.4.1　File 类型

File API 仍然以表单中的文件输入字段为基础，但是增加了直接访问文件信息的能力。HTML5 在 DOM 上为文件输入元素添加了 files 集合。当用户在文件字段中选择一个或多个文件时，这个 files 集合中会包含一组 File 对象，表示被选中的文件。每个 File 对象都有一些只读属性。

- name：本地系统中的文件名。
- size：以字节计的文件大小。
- type：包含文件 MIME 类型的字符串。
- lastModifiedDate：表示文件最后修改时间的字符串。这个属性只有 Chome 实现了。

例如，通过监听 change 事件然后遍历 files 集合可以取得每个选中文件的信息：

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
  let files = event.target.files,
      i = 0,
      len = files.length;

  while (i < len) {
    const f = files[i];
    console.log(`${f.name} (${f.type}, ${f.size} bytes)`);
    i++;
  }
});
```

这个例子简单地在控制台输出了每个文件的信息。仅就这个能力而言，已经可以说是 Web 应用向前迈进的一大步了。不过，File API 还提供了 FileReader 类型，让我们可以实际从文件中读取数据。

20.4.2　FileReader 类型

FileReader 类型表示一种异步文件读取机制。可以把 FileReader 想象成类似于 XMLHttpRequest，只不过是用于从文件系统读取文件，而不是从服务器读取数据。FileReader 类型提供了几个读取文件数据的方法。

- readAsText(file, encoding)：从文件中读取纯文本内容并保存在 result 属性中。第二个参数表示编码，是可选的。

- readAsDataURL(file)：读取文件并将内容的数据 URI 保存在 result 属性中。
- readAsBinaryString(file)：读取文件并将每个字符的二进制数据保存在 result 属性中。
- readAsArrayBuffer(file)：读取文件并将文件内容以 ArrayBuffer 形式保存在 result 属性。

这些读取数据的方法为处理文件数据提供了极大的灵活性。例如，为了向用户显示图片，可以将图片读取为数据 URI，而为了解析文件内容，可以将文件读取为文本。

因为这些读取方法是异步的，所以每个 FileReader 会发布几个事件，其中 3 个最有用的事件是 progress、error 和 load，分别表示还有更多数据、发生了错误和读取完成。

progress 事件每 50 毫秒就会触发一次，其与 XHR 的 progress 事件具有相同的信息：lengthComputable、loaded 和 total。此外，在 progress 事件中可以读取 FileReader 的 result 属性，即使其中尚未包含全部数据。

error 事件会在由于某种原因无法读取文件时触发。触发 error 事件时，FileReader 的 error 属性会包含错误信息。这个属性是一个对象，只包含一个属性：code。这个错误码的值可能是 1（未找到文件）、2（安全错误）、3（读取被中断）、4（文件不可读）或 5（编码错误）。

load 事件会在文件成功加载后触发。如果 error 事件被触发，则不会再触发 load 事件。下面的例子演示了所有这 3 个事件：

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
  let info = "",
      output = document.getElementById("output"),
      progress = document.getElementById("progress"),
      files = event.target.files,
      type = "default",
      reader = new FileReader();

  if (/image/.test(files[0].type)) {
    reader.readAsDataURL(files[0]);
    type = "image";
  } else {
    reader.readAsText(files[0]);
    type = "text";
  }

  reader.onerror = function() {
    output.innerHTML = "Could not read file, error code is " +
        reader.error.code;
  };

  reader.onprogress = function(event) {
    if (event.lengthComputable) {
      progress.innerHTML = `${event.loaded}/${event.total}`;
    }
  };

  reader.onload = function() {
    let html = "";

    switch(type) {
      case "image":
        html = `<img src="${reader.result}">`;
        break;
```

```
        case "text":
          html = reader.result;
          break;
      }
      output.innerHTML = html;
    };
});
```

以上代码从表单字段中读取一个文件，并将其内容显示在了网页上。如果文件的 MIME 类型表示它是一个图片，那么就将其读取后保存为数据 URI，在 load 事件触发时将数据 URI 作为图片插入页面中。如果文件不是图片，则读取后将其保存为文本并原样输出到网页上。progress 事件用于跟踪和显示读取文件的进度，而 error 事件用于监控错误。

如果想提前结束文件读取，则可以在过程中调用 abort()方法，从而触发 abort 事件。在 load、error 和 abort 事件触发后，还会触发 loadend 事件。loadend 事件表示在上述 3 种情况下，所有读取操作都已经结束。readAsText()和 readAsDataURL()方法已经得到了所有主流浏览器支持。

20.4.3　FileReaderSync 类型

顾名思义，FileReaderSync 类型就是 FileReader 的**同步**版本。这个类型拥有与 FileReader 相同的方法，只有在整个文件都加载到内存之后才会继续执行。FileReaderSync 只在工作线程中可用，因为如果读取整个文件耗时太长则会影响全局。

假设通过 postMessage()向工作线程发送了一个 File 对象。以下代码会让工作线程同步将文件读取到内存中，然后将文件的数据 URL 发回来：

```
// worker.js

self.omessage = (messageEvent) => {
  const syncReader = new FileReaderSync();
  console.log(syncReader); // FileReaderSync {}

  // 读取文件时阻塞工作线程
  const result = syncReader.readAsDataUrl(messageEvent.data);

  // PDF 文件的示例响应
  console.log(result); // data:application/pdf;base64,JVBERi0xLjQK...

  // 把 URL 发回去
  self.postMessage(result);
};
```

20.4.4　Blob 与部分读取

某些情况下，可能需要读取部分文件而不是整个文件。为此，File 对象提供了一个名为 slice()的方法。slice()方法接收两个参数：起始字节和要读取的字节数。这个方法返回一个 Blob 的实例，而 Blob 实际上是 File 的超类。

blob 表示**二进制大对象**（binary larget object），是 JavaScript 对不可修改二进制数据的封装类型。包含字符串的数组、ArrayBuffers、ArrayBufferViews，甚至其他 Blob 都可以用来创建 blob。Blob 构造函数可以接收一个 options 参数，并在其中指定 MIME 类型：

```
console.log(new Blob(['foo']));
// Blob {size: 3, type: ""}

console.log(new Blob(['{"a": "b"}'], { type: 'application/json' }));
// {size: 10, type: "application/json"}

console.log(new Blob(['<p>Foo</p>', '<p>Bar</p>'], { type: 'text/html' }));
// {size: 20, type: "text/html"}
```

Blob 对象有一个 size 属性和一个 type 属性，还有一个 slice()方法用于进一步切分数据。另外也可以使用 FileReader 从 Blob 中读取数据。下面的例子只会读取文件的前 32 字节：

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
  let info = "",
    output = document.getElementById("output"),
    progress = document.getElementById("progress"),
    files = event.target.files,
    reader = new FileReader(),
    blob = blobSlice(files[0], 0, 32);
  if (blob) {
    reader.readAsText(blob);

    reader.onerror = function() {
      output.innerHTML = "Could not read file, error code is " +
                 reader.error.code;
    };
    reader.onload = function() {
      output.innerHTML = reader.result;
    };
  } else {
    console.log("Your browser doesn't support slice().");
  }
});
```

只读取部分文件可以节省时间，特别是在只需要数据特定部分比如文件头的时候。

20.4.5 对象 URL 与 Blob

对象 URL 有时候也称作 Blob URL，是指引用存储在 File 或 Blob 中数据的 URL。对象 URL 的优点是不用把文件内容读取到 JavaScript 也可以使用文件。只要在适当位置提供对象 URL 即可。要创建对象 URL，可以使用 window.URL.createObjectURL()方法并传入 File 或 Blob 对象。这个函数返回的值是一个指向内存中地址的字符串。因为这个字符串是 URL，所以可以在 DOM 中直接使用。例如，以下代码使用对象 URL 在页面中显示了一张图片：

```
let filesList = document.getElementById("files-list");
filesList.addEventListener("change", (event) => {
  let info = "",
    output = document.getElementById("output"),
    progress = document.getElementById("progress"),
    files = event.target.files,
    reader = new FileReader(),
    url = window.URL.createObjectURL(files[0]);
  if (url) {
    if (/image/.test(files[0].type)) {
      output.innerHTML = `<img src="${url}">`;
```