还有与命令相关的其他一些方法。第一个方法是 queryCommandEnabled()，此方法用于确定对当前选中文本或光标所在位置是否可以执行相关命令。它只接收一个参数，即要检查的命令名。如果可编辑区可以执行该命令就返回 true，否则返回 false。来看下面的例子：

```
let result = frames["richedit"].document.queryCommandEnabled("bold");
```

以上代码在当前选区可以执行"bold"命令时返回 true。不过要注意，queryCommandEnabled() 返回 true 并不代表允许执行相关命令，只代表当前选区适合执行相关命令。在 Firefox 中，queryCommandEnabled("cut")即使默认不允许剪切也会返回 true。

另一个方法 queryCommandState()用于确定相关命令是否应用到了当前文本选区。例如，要确定当前选区的文本是否为粗体，可以这样：

```
let isBold = frames["richedit"].document.queryCommandState("bold");
```

如果之前给文本选区应用过"bold"命令，则以上代码返回 true。全功能富文本编辑器可以利用这个方法更新粗体、斜体等按钮。

最后一个方法是 queryCommandValue()，此方法可以返回执行命令时使用的值（即前面示例的 execCommand()中的第三个参数）。如果对一段选中文本应用了值为 7 的"fontsize"命令，则如下代码会返回 7：

```
let fontSize = frames["richedit"].document.queryCommandValue("fontsize");
```

这个方法可用于确定如何将命令应用于文本选区，从而进一步决定是否需要执行下一个命令。

19.5.3　富文件选择

在内嵌窗格中使用 getSelection()方法，可以获得富文本编辑器的选区。这个方法暴露在 document 和 window 对象上，返回表示当前选中文本的 Selection 对象。每个 Selection 对象都拥有以下属性。

- anchorNode：选区开始的节点。
- anchorOffset：在 anchorNode 中，从开头到选区开始跳过的字符数。
- focusNode：选区结束的节点。
- focusOffset：focusNode 中包含在选区内的字符数。
- isCollapsed：布尔值，表示选区起点和终点是否在同一个地方。
- rangeCount：选区中包含的 DOM 范围数量。

Selection 的属性并没有包含很多有用的信息。好在它的以下方法提供了更多信息，并允许操作选区。

- addRange(*range*)：把给定的 DOM 范围添加到选区。
- collapse(*node, offset*)：将选区折叠到给定节点中给定的文本偏移处。
- collapseToEnd()：将选区折叠到终点。
- collapseToStart()：将选区折叠到起点。
- containsNode(*node*)：确定给定节点是否包含在选区中。
- deleteFromDocument()：从文档中删除选区文本。与执行 execCommand("delete", false, null)命令结果相同。
- extend(*node, offset*)：通过将 focusNode 和 focusOffset 移动到指定值来扩展选区。

- ❑ getRangeAt(*index*)：返回选区中指定索引处的 DOM 范围。
- ❑ removeAllRanges()：从选区中移除所有 DOM 范围。这实际上会移除选区，因为选区中至少要包含一个范围。
- ❑ removeRange(*range*)：从选区中移除指定的 DOM 范围。
- ❑ selectAllChildren(*node*)：清除选区并选择给定节点的所有子节点。
- ❑ toString()：返回选区中的文本内容。

Selection 对象的这个方法极其强大，充分利用了 DOM 范围来管理选区。操纵 DOM 范围可以实现比 execCommand()更细粒度的控制，因为可以直接对选中文本的 DOM 内容进行操作。来看下面的例子：

```
let selection = frames["richedit"].getSelection();

// 取得选中的文本
let selectedText = selection.toString();

// 取得表示选区的范围
let range = selection.getRangeAt(0);

// 高亮选中的文本
let span = frames["richedit"].document.createElement("span");
span.style.backgroundColor = "yellow";
range.surroundContents(span);
```

以上代码会在富文本编辑器中给选中文本添加黄色高亮背景。实现方式是在默认选区使用 DOM 范围，用 surroundContents()方法给选中文本添加背景为黄色的标签。

getSelection()方法在 HTML5 中进行了标准化，IE9 以及 Firefox、Safari、Chrome 和 Opera 的所有现代版本中都实现了这个方法。

IE8 及更早版本不支持 DOM 范围，不过它们允许通过专有的 selection 对象操作选中的文本。如本章前面所讨论的，这个 selection 对象是 document 的属性。要取得富文本编辑器中选中的文本，必须先创建一个文本范围，然后再访问其 text 属性：

```
let range = frames["richedit"].document.selection.createRange();
let selectedText = range.text;
```

使用 IE 文本范围执行 HTML 操作不像使用 DOM 范围那么可靠，不过也是可以做到的。要实现与使用 DOM 范围一样的高亮效果，可以组合使用 htmlText 属性和 pasteHTML()方法：

```
let range = frames["richedit"].document.selection.createRange();
range.pasteHTML(
    '<span style="background-color:yellow">${range.htmlText}</span>');
```

以上代码使用 htmlText 取得了当前选区的 HTML，然后用一个标签将其包围起来并通过 pasteHTML()再把它插入选区中。

19.5.4 通过表单提交富文本

因为富文本编辑是在内嵌窗格中或通过为元素指定 contenteditable 属性实现的，而不是在表单控件中实现，所以富文本编辑器技术上与表单没有关系。这意味着要把富文本编辑的结果提交给服务器，必须手工提取 HTML 并自己提交。通常的解决方案是在表单中添加一个隐藏字段，使用内嵌窗格或 contenteditable 元素的 HTML 更新它的值。在表单提交之前，从内嵌窗格或 contenteditable 元

素中提取出 HTML 并插入隐藏字段中。例如，以下代码在使用内嵌窗格实现富文本编辑时，可以用在表单的 onsubmit 事件处理程序中：

```
form.addEventListener("submit", (event) => {
  let target = event.target;

  target.elements["comments"].value =
      frames["richedit"].document.body.innerHTML;
});
```

这里，代码使用文档主体的 innerHTML 属性取得了内嵌窗格的 HTML，然后将其插入名为"comments"的表单字段中。这样做可以确保在提交表单之前给表单字段赋值。如果使用 submit()方法手工提交表单，那么要注意在提交前先执行上述操作。对于 contenteditable 元素，执行这一操作的代码是类似的：

```
form.addEventListener("submit", (event) => {
  let target = event.target;

  target.elements["comments"].value =
      document.getElementById("richedit").innerHTML;
});
```

19.6　小结

尽管 HTML 和 Web 应用自诞生以来已经发生了天翻地覆的变化，但 Web 表单几乎从来没有变过。JavaScript 可以增加现有的表单字段以提供新功能或增强易用性。为此，表单字段也暴露了属性、方法和事件供 JavaScript 使用。以下是本章介绍的一些概念。

- 可以使用标准或非标准的方法全部或部分选择文本框中的文本。
- 所有浏览器都采用了 Firefox 操作文本选区的方式，使其成为真正的标准。
- 可以通过监听键盘事件并检测要插入的字符来控制文本框接受或不接受某些字符。

所有浏览器都支持剪贴板相关的事件，包括 copy、cut 和 paste。剪贴板事件在不同浏览器中的实现有很大差异。

在文本框只限某些字符时，可以利用剪贴板事件屏幕粘贴事件。

选择框也是经常使用 JavaScript 来控制的一种表单控件。借助 DOM，操作选择框比以前方便了很多。使用标准的 DOM 技术，可以为选择框添加或移除选项，也可以将选项从一个选择框移动到另一个选择框，或者重排选项。

富文本编辑通常以使用包含空白 HTML 文档的内嵌窗格来处理。通过将文档的 designMode 属性设置为"on"，可以让整个页面变成编辑区，就像文字处理软件一样。另外，给元素添加 contenteditable 属性也可以将元素转换为可编辑区。默认情况下，可以切换文本的粗体、斜体样式，也可以使用剪贴板功能。JavaScript 通过 execCommand()方法可以执行一些富文本编辑功能，通过 queryCommandEnabled()、queryCommandState()和 queryCommandValue()方法则可以获取有关文本选区的信息。由于富文本编辑区不涉及表单字段，因此要将富文本内容提交到服务器，必须把 HTML 从 iframe 或 contenteditable 元素中复制到一个表单字段。

第20章

JavaScript API

本章内容

- Atomics 与 SharedArrayBuffer
- 跨上下文消息
- Encoding API
- File API 与 Blob API
- 拖放
- Notifications API
- Page Visibility API
- Streams API
- 计时 API
- Web 组件
- Web Cryptography API

随着 Web 浏览器能力的增加，其复杂性也在迅速增加。从很多方面看，现代 Web 浏览器已经成为构建于诸多规范之上、集不同 API 于一身的“瑞士军刀”。浏览器规范的生态在某种程度上是混乱而无序的。一些规范如 HTML5，定义了一批增强已有标准的 API 和浏览器特性。而另一些规范如 Web Cryptography API 和 Notifications API，只为一个特性定义了一个 API。不同浏览器实现这些新 API 的情况也不同，有的会实现其中一部分，有的则干脆尚未实现。

最终，是否使用这些比较新的 API 还要看项目是支持更多浏览器，还是要采用更多现代特性。有些 API 可以通过腻子脚本来模拟，但腻子脚本通常会带来性能问题，此外也会增加网站 JavaScript 代码的体积。

> **注意** Web API 的数量之多令人难以置信（参见 MDN 文档的 Web APIs 词条）。本章要介绍的 API 仅限于与大多数开发者有关、已经得到多个浏览器支持，且本书其他章节没有涵盖的部分。

20.1 Atomics 与 SharedArrayBuffer

多个上下文访问 SharedArrayBuffer 时，如果同时对缓冲区执行操作，就可能出现资源争用问题。Atomics API 通过强制同一时刻只能对缓冲区执行一个操作，可以让多个上下文安全地读写一个 SharedArrayBuffer。Atomics API 是 ES2017 中定义的。

仔细研究会发现 Atomics API 非常像一个简化版的指令集架构（ISA），这并非意外。原子操作的本

质会排斥操作系统或计算机硬件通常会自动执行的优化（比如指令重新排序）。原子操作也让并发访问内存变得不可能，如果应用不当就可能导致程序执行变慢。为此，Atomics API 的设计初衷是在最少但很稳定的原子行为基础之上，构建复杂的多线程 JavaScript 程序。

20.1.1 SharedArrayBuffer

SharedArrayBuffer 与 ArrayBuffer 具有同样的 API。二者的主要区别是 ArrayBuffer 必须在不同执行上下文间切换，SharedArrayBuffer 则可以被任意多个执行上下文同时使用。

在多个执行上下文间共享内存意味着并发线程操作成为了可能。传统 JavaScript 操作对于并发内存访问导致的资源争用没有提供保护。下面的例子演示了 4 个专用工作线程访问同一个 SharedArrayBuffer 导致的资源争用问题：

```
const workerScript = `
self.onmessage = ({data}) => {
  const view = new Uint32Array(data);

  // 执行 1 000 000 次加操作
  for (let i = 0; i < 1E6; ++i) {
    // 线程不安全加操作会导致资源争用
    view[0] += 1;
  }

  self.postMessage(null);
};
`;

const workerScriptBlobUrl = URL.createObjectURL(new Blob([workerScript]));

// 创建容量为 4 的工作线程池
const workers = [];
for (let i = 0; i < 4; ++i) {
  workers.push(new Worker(workerScriptBlobUrl));
}

// 在最后一个工作线程完成后打印出最终值
let responseCount = 0;
for (const worker of workers) {
  worker.onmessage = () => {
    if (++responseCount == workers.length) {
      console.log(`Final buffer value: ${view[0]}`);
    }
  };
}

// 初始化 SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);
view[0] = 1;

// 把 SharedArrayBuffer 发送到每个工作线程
for (const worker of workers) {
  worker.postMessage(sharedArrayBuffer);
}

// (期待结果为 4000001。实际输出可能类似这样:)
// Final buffer value: 2145106
```

为解决这个问题，Atomics API 应运而生。Atomics API 可以保证 SharedArrayBuffer 上的 JavaScript 操作是线程安全的。

> **注意** SharedArrayBuffer API 等同于 ArrayBuffer API，后者在第 6 章介绍过。关于如何在多个上下文中使用 SharedArrayBuffer，可以参考第 27 章。

20.1.2 原子操作基础

任何全局上下文中都有 Atomics 对象，这个对象上暴露了用于执行线程安全操作的一套静态方法，其中多数方法以一个 TypedArray 实例（一个 SharedArrayBuffer 的引用）作为第一个参数，以相关操作数作为后续参数。

1. 算术及位操作方法

Atomics API 提供了一套简单的方法用以执行就地修改操作。在 ECMA 规范中，这些方法被定义为 AtomicReadModifyWrite 操作。在底层，这些方法都会从 SharedArrayBuffer 中某个位置读取值，然后执行算术或位操作，最后再把计算结果写回相同的位置。这些操作的原子本质意味着上述读取、修改、写回操作会按照顺序执行，不会被其他线程中断。

以下代码演示了所有算术方法：

```
// 创建大小为 1 的缓冲区
let sharedArrayBuffer = new SharedArrayBuffer(1);

// 基于缓冲创建 Uint8Array
let typedArray = new Uint8Array(sharedArrayBuffer);

// 所有 ArrayBuffer 全部初始化为 0
console.log(typedArray); // Uint8Array[0]

const index = 0;
const increment = 5;

// 对索引 0 处的值执行原子加 5
Atomics.add(typedArray, index, increment);

console.log(typedArray); // Uint8Array[5]

// 对索引 0 处的值执行原子减 5
Atomics.sub(typedArray, index, increment);

console.log(typedArray); // Uint8Array[0]
```

以下代码演示了所有位方法：

```
// 创建大小为 1 的缓冲区
let sharedArrayBuffer = new SharedArrayBuffer(1);

// 基于缓冲创建 Uint8Array
let typedArray = new Uint8Array(sharedArrayBuffer);

// 所有 ArrayBuffer 全部初始化为 0
console.log(typedArray); // Uint8Array[0]
```

```
const index = 0;

// 对索引 0 处的值执行原子或 0b1111
Atomics.or(typedArray, index, 0b1111);

console.log(typedArray); // Uint8Array[15]

// 对索引 0 处的值执行原子与 0b1111
Atomics.and(typedArray, index, 0b1100);

console.log(typedArray); // Uint8Array[12]

// 对索引 0 处的值执行原子异或 0b1111
Atomics.xor(typedArray, index, 0b1111);

console.log(typedArray); // Uint8Array[3]
```

前面线程不安全的例子可以改写为下面这样：

```
  const workerScript = `
  self.onmessage = ({data}) => {

  const view = new Uint32Array(data);

  // 执行 1 000 000 次加操作
  for (let i = 0; i < 1E6; ++i) {
    // 线程安全的加操作
    Atomics.add(view, 0, 1);
  }

  self.postMessage(null);
};
`;

const workerScriptBlobUrl = URL.createObjectURL(new Blob([workerScript]));

// 创建容量为 4 的工作线程池
const workers = [];
for (let i = 0; i < 4; ++i) {
  workers.push(new Worker(workerScriptBlobUrl));
}

// 在最后一个工作线程完成后打印出最终值
let responseCount = 0;
for (const worker of workers) {
  worker.onmessage = () => {
    if (++responseCount == workers.length) {
      console.log(`Final buffer value: ${view[0]}`);
    }
  };
}

// 初始化 SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);
view[0] = 1;

// 把 SharedArrayBuffer 发送到每个工作线程
for (const worker of workers) {
```

```
  worker.postMessage(sharedArrayBuffer);
}

// (期待结果为 4000001)
// Final buffer value: 4000001
```

2. 原子读和写

浏览器的 JavaScript 编译器和 CPU 架构本身都有权限重排指令以提升程序执行效率。正常情况下，JavaScript 的单线程环境是可以随时进行这种优化的。但多线程下的指令重排可能导致资源争用，而且极难排错。

Atomics API 通过两种主要方式解决了这个问题。

- 所有原子指令相互之间的顺序永远不会重排。
- 使用原子读或原子写保证所有指令（包括原子和非原子指令）都不会相对原子读/写重新排序。这意味着位于原子读/写之前的所有指令会在原子读/写发生前完成，而位于原子读/写之后的所有指令会在原子读/写完成后才会开始。

除了读写缓冲区的值，`Atomics.load()`和`Atomics.store()`还可以构建“代码围栏”。JavaScript 引擎保证非原子指令可以相对于`load()`或`store()`**本地**重排，但这个重排不会侵犯原子读/写的边界。以下代码演示了这种行为：

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// 执行非原子写
view[0] = 1;

// 非原子写可以保证在这个读操作之前完成，因此这里一定会读到 1
console.log(Atomics.load(view, 0)); // 1

// 执行原子写
Atomics.store(view, 0, 2);

// 非原子读可以保证在原子写完成后发生，因此这里一定会读到 2
console.log(view[0]); // 2
```

3. 原子交换

为了保证连续、不间断的先读后写，Atomics API 提供了两种方法：`exchange()`和`compareExchange()`。`Atomics.exchange()`执行简单的交换，以保证其他线程不会中断值的交换：

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// 在索引 0 处写入 3
Atomics.store(view, 0, 3);

// 从索引 0 处读取值，然后在索引 0 处写入 4
console.log(Atomics.exchange(view, 0, 4));  // 3

// 从索引 0 处读取值
console.log(Atomics.load(view, 0));         // 4
```

在多线程程序中，一个线程可能**只希望**在上次读取某个值之后没有其他线程修改该值的情况下才对共享缓冲区执行写操作。如果这个值没有被修改，这个线程就可以安全地写入更新后的值；如果这个值

被修改了，那么执行写操作将会破坏其他线程计算的值。对于这种任务，Atomics API 提供了 compareExchange()方法。这个方法只在目标索引处的值与预期值匹配时才会执行写操作。来看下面这个例子：

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// 在索引 0 处写入 5
Atomics.store(view, 0, 5);
// 从缓冲区读取值
let initial = Atomics.load(view, 0);

// 对这个值执行非原子操作
let result = initial ** 2;

// 只在缓冲区未被修改的情况下才会向缓冲区写入新值
Atomics.compareExchange(view, 0, initial, result);

// 检查写入成功
console.log(Atomics.load(view, 0)); // 25
```

如果值不匹配，compareExchange()调用则什么也不做：

```
const sharedArrayBuffer = new SharedArrayBuffer(4);
const view = new Uint32Array(sharedArrayBuffer);

// 在索引 0 处写入 5
Atomics.store(view, 0, 5);
// 从缓冲区读取值
let initial = Atomics.load(view, 0);

// 对这个值执行非原子操作
let result = initial ** 2;

// 只在缓冲区未被修改的情况下才会向缓冲区写入新值
Atomics.compareExchange(view, 0, -1, result);

// 检查写入失败
console.log(Atomics.load(view, 0)); // 5
```

4. 原子 Futex 操作与加锁

如果没有某种锁机制，多线程程序就无法支持复杂需求。为此，Atomics API 提供了模仿 Linux Futex（**快速用户空间互斥量**，fast user-space mutex）的方法。这些方法本身虽然非常简单，但可以作为更复杂锁机制的基本组件。

> **注意**　所有原子 Futex 操作只能用于 Int32Array 视图。而且，也只能用在工作线程内部。

Atomics.wait()和 Atomics.notify()通过示例很容易理解。下面这个简单的例子创建了 4 个工作线程，用于对长度为 1 的 Int32Array 进行操作。这些工作线程会依次取得锁并执行自己的加操作：

```
const workerScript = `
self.onmessage = ({data}) => {
  const view = new Int32Array(data);

  console.log('Waiting to obtain lock');

  // 遇到初始值则停止，10 000 毫秒超时
```

```
  Atomics.wait(view, 0, 0, 1E5);

  console.log('Obtained lock');

  // 在索引 0 处加 1
  Atomics.add(view, 0, 1);

  console.log('Releasing lock');

  // 只允许 1 个工作线程继续执行
  Atomics.notify(view, 0, 1);

  self.postMessage(null);
};
`;

const workerScriptBlobUrl = URL.createObjectURL(new Blob([workerScript]));

const workers = [];
for (let i = 0; i < 4; ++i) {
  workers.push(new Worker(workerScriptBlobUrl));
}

// 在最后一个工作线程完成后打印出最终值
let responseCount = 0;
for (const worker of workers) {
  worker.onmessage = () => {
    if (++responseCount == workers.length) {
      console.log(`Final buffer value: ${view[0]}`);
    }
  };
}

// 初始化 SharedArrayBuffer
const sharedArrayBuffer = new SharedArrayBuffer(8);
const view = new Int32Array(sharedArrayBuffer);

// 把 SharedArrayBuffer 发送到每个工作线程
for (const worker of workers) {
  worker.postMessage(sharedArrayBuffer);
}

// 1000 毫秒后释放第一个锁
setTimeout(() => Atomics.notify(view, 0, 1), 1000);

// Waiting to obtain lock
// Waiting to obtain lock
// Waiting to obtain lock
// Waiting to obtain lock
// Obtained lock
// Releasing lock
// Obtained lock
// Releasing lock
// Obtained lock
// Releasing lock
// Obtained lock
// Releasing lock
// Final buffer value: 4
```