调用这个构造函数返回的 Notification 对象的 close()方法可以关闭显示的通知。下面的例子展示了显示通知后 1000 毫秒再关闭它：

```
const n = new Notification('I will close in 1000ms');
setTimeout(() => n.close(), 1000);
```

20.7.3　通知生命周期回调

通知并非只用于显示文本字符串，也可用于实现交互。Notifications API 提供了 4 个用于添加回调的生命周期方法：

- onshow 在通知显示时触发；
- onclick 在通知被点击时触发；
- onclose 在通知消失或通过 close()关闭时触发；
- onerror 在发生错误阻止通知显示时触发。

下面的代码将每个生命周期事件都通过日志打印了出来：

```
const n = new Notification('foo');

n.onshow = () => console.log('Notification was shown!');
n.onclick = () => console.log('Notification was clicked!');
n.onclose = () => console.log('Notification was closed!');
n.onerror = () => console.log('Notification experienced an error!');
```

20.8　Page Visibility API

Web 开发中一个常见的问题是开发者不知道用户什么时候真正在使用页面。如果页面被最小化或隐藏在其他标签页后面，那么轮询服务器或更新动画等功能可能就没有必要了。Page Visibility API 旨在为开发者提供页面对用户是否可见的信息。

这个 API 本身非常简单，由 3 部分构成。

- document.visibilityState 值，表示下面 4 种状态之一。
  - 页面在后台标签页或浏览器中最小化了。
  - 页面在前台标签页中。
  - 实际页面隐藏了，但对页面的预览是可见的（例如在 Windows 7 上，用户鼠标移到任务栏图标上会显示网页预览）。
  - 页面在屏外预渲染。
- visibilitychange 事件，该事件会在文档从隐藏变可见（或反之）时触发。
- document.hidden 布尔值，表示页面是否隐藏。这可能意味着页面在后台标签页或浏览器中被最小化了。这个值是为了向后兼容才继续被浏览器支持的，应该优先使用 document.visibilityState 检测页面可见性。

要想在页面从可见变为隐藏或从隐藏变为可见时得到通知，需要监听 visibilitychange 事件。document.visibilityState 的值是以下三个字符串之一：

- "hidden"
- "visible"
- "prerender"

20.9 Streams API

Streams API 是为了解决一个简单但又基础的问题而生的：Web 应用如何消费有序的小信息块而不是大块信息？这种能力主要有两种应用场景。

- 大块数据可能不会一次性都可用。网络请求的响应就是一个典型的例子。网络负载是以连续信息包形式交付的，而流式处理可以让应用在数据一到达就能使用，而不必等到所有数据都加载完毕。
- 大块数据可能需要分小部分处理。视频处理、数据压缩、图像编码和 JSON 解析都是可以分成小部分进行处理，而不必等到所有数据都在内存中时再处理的例子。

第 24 章在讨论网络请求和远程资源时会介绍 Streams API 在 `fetch()`中的应用，不过 Streams API 本身是通用的。实现 Observable 接口的 JavaScript 库共享了很多流的基础概念。

> **注意** 虽然 Fetch API 已经得到所有主流浏览器支持，但 Streams API 则没有那么快得到支持。

20.9.1 理解流

提到流，可以把数据想像成某种通过管道输送的液体。JavaScript 中的流借用了管道相关的概念，因为原理是相通的。根据规范，“这些 API 实际是为映射低级 I/O 原语而设计，包括适当时候对字节流的规范化”。Stream API 直接解决的问题是处理网络请求和读写磁盘。

Stream API 定义了三种流。

- **可读流**：可以通过某个公共接口读取数据块的流。数据在内部从底层源进入流，然后由**消费者**（consumer）进行处理。
- **可写流**：可以通过某个公共接口写入数据块的流。**生产者**（producer）将数据写入流，数据在内部传入底层数据槽（sink）。
- **转换流**：由两种流组成，可写流用于接收数据（可写端），可读流用于输出数据（可读端）。这两个流之间是**转换程序**（transformer），可以根据需要检查和修改流内容。

块、内部队列和反压

流的基本单位是**块**（chunk）。块可是任意数据类型，但通常是定型数组。每个块都是离散的流片段，可以作为一个整体来处理。更重要的是，块不是固定大小的，也不一定按固定间隔到达。在理想的流当中，块的大小通常近似相同，到达间隔也近似相等。不过好的流实现需要考虑边界情况。

前面提到的各种类型的流都有入口和出口的概念。有时候，由于数据进出速率不同，可能会出现不匹配的情况。为此流平衡可能出现如下三种情形。

- 流出口处理数据的速度比入口提供数据的速度快。流出口经常空闲（可能意味着流入口效率较低），但只会浪费一点内存或计算资源，因此这种流的不平衡是可以接受的。
- 流入和流出均衡。这是理想状态。
- 流入口提供数据的速度比出口处理数据的速度快。这种流不平衡是固有的问题。此时一定会在某个地方出现数据积压，流必须相应做出处理。

流不平衡是常见问题，但流也提供了解决这个问题的工具。所有流都会为已进入流但尚未离开流的块提供一个内部队列。对于均衡流，这个内部队列中会有零个或少量排队的块，因为流出口块出列的速

度与流入口块入列的速度近似相等。这种流的内部队列所占用的内存相对比较小。

如果块入列速度快于出列速度，则内部队列会不断增大。流不能允许其内部队列无限增大，因此它会使用**反压**（backpressure）通知流入口停止发送数据，直到队列大小降到某个既定的阈值之下。这个阈值由排列策略决定，这个策略定义了内部队列可以占用的最大内存，即**高水位线**（high water mark）。

20.9.2　可读流

可读流是对底层数据源的封装。底层数据源可以将数据填充到流中，允许消费者通过流的公共接口读取数据。

1. ReadableStreamDefaultController

来看下面的生成器，它每 1000 毫秒就会生成一个递增的整数：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}
```

这个生成器的值可以通过可读流的控制器传入可读流。访问这个控制器最简单的方式就是创建 `ReadableStream` 的一个实例，并在这个构造函数的 `underlyingSource` 参数（第一个参数）中定义 `start()`方法，然后在这个方法中使用作为参数传入的 `controller`。默认情况下，这个控制器参数是 `ReadableStreamDefaultController` 的一个实例：

```
const readableStream = new ReadableStream({
  start(controller) {
    console.log(controller); // ReadableStreamDefaultController {}
  }
});
```

调用控制器的 `enqueue()`方法可以把值传入控制器。所有值都传完之后，调用 `close()`关闭流：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

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

    controller.close();
  }
});
```

2. ReadableStreamDefaultReader

前面的例子把 5 个值加入了流的队列，但没有把它们从队列中读出来。为此，需要一个 `ReadableStreamDefaultReader` 的实例，该实例可以通过流的 `getReader()`方法获取。调用这个方法会获得流的锁，保证只有这个读取器可以从流中读取值：

```
  async function* ints() {
    // 每1000毫秒生成一个递增的整数
    for (let i = 0; i < 5; ++i) {
      yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
    }
  }

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

      controller.close();
    }
});

console.log(readableStream.locked); // false
const readableStreamDefaultReader = readableStream.getReader();
console.log(readableStream.locked); // true
```

消费者使用这个读取器实例的`read()`方法可以读出值：

```
async function* ints() {
  // 每1000毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

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

    controller.close();
  }
});
console.log(readableStream.locked); // false
const readableStreamDefaultReader = readableStream.getReader();
console.log(readableStream.locked); // true

// 消费者
(async function() {
  while(true) {
    const { done, value } = await readableStreamDefaultReader.read();
      if (done) {
        break;
      } else {
        console.log(value);
      }
  }
})();

// 0
// 1
// 2
// 3
// 4
```

20.9.3　可写流

可写流是底层数据槽的封装。底层数据槽处理通过流的公共接口写入的数据。

1. 创建 WritableStream

来看下面的生成器，它每 1000 毫秒就会生成一个递增的整数：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}
```

这些值通过可写流的公共接口可以写入流。在传给 WritableStream 构造函数的 underlyingSink 参数中，通过实现 write()方法可以获得写入的数据：

```
const readableStream = new ReadableStream({
  write(value) {
    console.log(value);
  }
});
```

2. WritableStreamDefaultWriter

要把获得的数据写入流，可以通过流的 getWriter()方法获取 WritableStreamDefaultWriter 的实例。这样会获得流的锁，确保只有一个写入器可以向流中写入数据：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

const writableStream = new WritableStream({
  write(value) {
    console.log(value);
  }
});

console.log(writableStream.locked); // false
const writableStreamDefaultWriter = writableStream.getWriter();
console.log(writableStream.locked); // true
```

在向流中写入数据前，生产者必须确保写入器可以接收值。writableStreamDefaultWriter.ready 返回一个期约，此期约会在能够向流中写入数据时解决。然后，就可以把值传给 writableStream-DefaultWriter.write()方法。写入数据之后，调用 writableStreamDefaultWriter.close()将流关闭：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

const writableStream = new WritableStream({
```

```
  write(value) {
    console.log(value);
  }
});

console.log(writableStream.locked); // false
const writableStreamDefaultWriter = writableStream.getWriter();
console.log(writableStream.locked); // true

// 生产者
(async function() {
  for await (let chunk of ints()) {
    await writableStreamDefaultWriter.ready;
    writableStreamDefaultWriter.write(chunk);
  }

  writableStreamDefaultWriter.close();
})();
```

20.9.4 转换流

转换流用于组合可读流和可写流。数据块在两个流之间的转换是通过 `transform()`方法完成的。来看下面的生成器，它每 1000 毫秒就会生成一个递增的整数：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}
```

下面的代码创建了一个 `TransformStream` 的实例，通过 `transform()`方法将每个值翻倍：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

const { writable, readable } = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk * 2);
  }
});
```

向转换流的组件流（可读流和可写流）传入数据和从中获取数据，与本章前面介绍的方法相同：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

const { writable, readable } = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk * 2);
```

```
  }
});

const readableStreamDefaultReader = readable.getReader();
const writableStreamDefaultWriter = writable.getWriter();

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

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

// 生产者
(async function() {
  for await (let chunk of ints()) {
    await writableStreamDefaultWriter.ready;
    writableStreamDefaultWriter.write(chunk);
  }

  writableStreamDefaultWriter.close();
})();
```

20.9.5　通过管道连接流

流可以通过管道连接成一串。最常见的用例是使用 pipeThrough()方法把 ReadableStream 接入 TransformStream。从内部看，ReadableStream 先把自己的值传给 TransformStream 内部的 WritableStream，然后执行转换，接着转换后的值又在新的 ReadableStream 上出现。下面的例子将一个整数的 ReadableStream 传入 TransformStream，TransformStream 对每个值做加倍处理：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

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

    controller.close();
  }
});

const doublingStream = new TransformStream({
  transform(chunk, controller) {
    controller.enqueue(chunk * 2);
```

```
  }
});

// 通过管道连接流
const pipedStream = integerStream.pipeThrough(doublingStream);

// 从连接流的输出获得读取器
const pipedStreamDefaultReader = pipedStream.getReader();

// 消费者
(async function() {
  while(true) {
    const { done, value } = await pipedStreamDefaultReader.read();

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

// 0
// 2
// 4
// 6
// 8
```


另外，使用 pipeTo()方法也可以将 ReadableStream 连接到 WritableStream。整个过程与使用 pipeThrough()类似：

```
async function* ints() {
  // 每 1000 毫秒生成一个递增的整数
  for (let i = 0; i < 5; ++i) {
    yield await new Promise((resolve) => setTimeout(resolve, 1000, i));
  }
}

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

    controller.close();
  }
});

const writableStream = new WritableStream({
  write(value) {
    console.log(value);
  }
});

const pipedStream = integerStream.pipeTo(writableStream);

// 0
// 1
```

```
// 2
// 3
// 4
```

注意，这里的管道连接操作隐式从 ReadableStream 获得了一个读取器，并把产生的值填充到 WritableStream。

20.10　计时 API

页面性能始终是 Web 开发者关心的话题。Performance 接口通过 JavaScript API 暴露了浏览器内部的度量指标，允许开发者直接访问这些信息并基于这些信息实现自己想要的功能。这个接口暴露在 window.performance 对象上。所有与页面相关的指标，包括已经定义和将来会定义的，都会存在于这个对象上。

Performance 接口由多个 API 构成：

- High Resolution Time API
- Performance Timeline API
- Navigation Timing API
- User Timing API
- Resource Timing API
- Paint Timing API

有关这些规范的更多信息以及新增的性能相关规范，可以关注 W3C 性能工作组的 GitHub 项目页面。

> **注意**　浏览器通常支持被废弃的 Level 1 和作为替代的 Level 2。本节尽量介绍 Level 2 级规范。

20.10.1　High Resolution Time API

Date.now()方法只适用于日期时间相关操作，而且是不要求计时精度的操作。在下面的例子中，函数 foo()调用前后分别记录了一个时间戳：

```
const t0 = Date.now();
foo();
const t1 = Date.now();

const duration = t1 - t0;

console.log(duration);
```

考虑如下 duration 会包含意外值的情况。

- **duration 是 0**。Date.now()只有毫秒级精度，如果 foo()执行足够快，则两个时间戳的值会相等。
- **duration 是负值或极大值**。如果在 foo()执行时，系统时钟被向后或向前调整了（如切换到夏令时），则捕获的时间戳不会考虑这种情况，因此时间差中会包含这些调整。

为此，必须使用不同的计时 API 来精确且准确地度量时间的流逝。High Resolution Time API 定义了 window.performance.now()，这个方法返回一个微秒精度的浮点值。因此，使用这个方法先后捕获

的时间戳更不可能出现相等的情况。而且这个方法可以保证时间戳单调增长。

```
const t0 = performance.now();
const t1 = performance.now();

console.log(t0);       // 1768.625000026077
console.log(t1);       // 1768.6300000059418

const duration = t1 - t0;

console.log(duration); // 0.004999979864805937
```

performance.now()计时器采用**相对**度量。这个计时器在执行上下文创建时从 0 开始计时。例如，打开页面或创建工作线程时，performance.now()就会从 0 开始计时。由于这个计时器在不同上下文中初始化时可能存在时间差，因此不同上下文之间如果没有共享参照点则不可能直接比较 performance.now()。performance.timeOrigin 属性返回计时器初始化时全局系统时钟的值。

```
const relativeTimestamp = performance.now();

const absoluteTimestamp = performance.timeOrigin + relativeTimestamp;

console.log(relativeTimestamp); // 244.43500000052154
console.log(absoluteTimestamp); // 1561926208892.4001
```

> **注意** 通过使用 performance.now()测量 L1 缓存与主内存的延迟差，幽灵漏洞（Spectre）可以执行缓存推断攻击。为弥补这个安全漏洞，所有的主流浏览器有的选择降低 performance.now()的精度，有的选择在时间戳里混入一些随机性。WebKit 博客上有一篇相关主题的不错的文章“What Spectre and Meltdown Mean For WebKit”，作者是 Filip Pizlo。

20.10.2 Performance Timeline API

Performance Timeline API 使用一套用于度量客户端延迟的工具扩展了 Performance 接口。性能度量将会采用计算结束与开始时间差的形式。这些开始和结束时间会被记录为 DOMHighResTimeStamp 值，而封装这个时间戳的对象是 PerformanceEntry 的实例。

浏览器会自动记录各种 PerformanceEntry 对象，而使用 performance.mark()也可以记录自定义的 PerformanceEntry 对象。在一个执行上下文中被记录的所有性能条目可以通过 performance.getEntries()获取：

```
console.log(performance.getEntries());

// [PerformanceNavigationTiming, PerformanceResourceTiming, ... ]
```

这个返回的集合代表浏览器的**性能时间线**（performance timeline）。每个 PerformanceEntry 对象都有 name、entryType、startTime 和 duration 属性：

```
const entry = performance.getEntries()[0];

console.log(entry.name);      // "https://foo.com"
console.log(entry.entryType); // navigation
console.log(entry.startTime); // 0
console.log(entry.duration);  // 182.36500001512468
```