> **注意**　结构化克隆算法在对象比较复杂时会存在计算性消耗。因此，实践中要尽可能避免过大、过多的复制。

2. 可转移对象

使用**可转移对象**（transferable objects）可以把所有权从一个上下文转移到另一个上下文。在不太可能在上下文间复制大量数据的情况下，这个功能特别有用。只有如下几种对象是可转移对象：

- ArrayBuffer
- MessagePort
- ImageBitmap
- OffscreenCanvas

postMessage()方法的第二个可选参数是数组，它指定应该将哪些对象转移到目标上下文。在遍历消息负载对象时，浏览器根据转移对象数组检查对象引用，并对转移对象进行转移而不复制它们。这意味着被转移的对象可以通过消息负载发送，消息负载本身会被复制，比如对象或数组。

下面的例子演示了工作者线程对 ArrayBuffer 的常规结构化克隆。这里没有对象转移：

main.js

```
const worker = new Worker('./worker.js');

// 创建 32 位缓冲区
const arrayBuffer = new ArrayBuffer(32);

console.log(`page's buffer size: ${arrayBuffer.byteLength}`); // 32

worker.postMessage(arrayBuffer);

console.log(`page's buffer size: ${arrayBuffer.byteLength}`); // 32
```

worker.js

```
self.onmessage = ({data}) => {
  console.log(`worker's buffer size: ${data.byteLength}`);    // 32
};
```

如果把 ArrayBuffer 指定为可转移对象，那么对缓冲区内存的引用就会从父上下文中抹去，然后分配给工作者线程。下面的例子演示了这个操作，结果分配给 ArrayBuffer 的内存从父上下文转移到了工作者线程：

main.js

```
const worker = new Worker('./worker.js');

// 创建 32 位缓冲区
const arrayBuffer = new ArrayBuffer(32);

console.log(`page's buffer size: ${arrayBuffer.byteLength}`); // 32

worker.postMessage(arrayBuffer, [arrayBuffer]);

console.log(`page's buffer size: ${arrayBuffer.byteLength}`); // 0
```

worker.js

```
self.onmessage = ({data}) => {
  console.log(`worker's buffer size: ${data.byteLength}`);    // 32
};
```

在其他类型的对象中嵌套可转移对象也完全没有问题。包装对象会被复制，而嵌套的对象会被转移：

main.js

```
const worker = new Worker('./worker.js');

// 创建 32 位缓冲区
const arrayBuffer = new ArrayBuffer(32);

console.log(`page's buffer size: ${arrayBuffer.byteLength}`);      // 32

worker.postMessage({foo: {bar: arrayBuffer}}, [arrayBuffer]);

console.log(`page's buffer size: ${arrayBuffer.byteLength}`);      // 0
```

worker.js

```
self.onmessage = ({data}) => {
  console.log(`worker's buffer size: ${data.foo.bar.byteLength}`); // 32
};
```

3. SharedArrayBuffer

> **注意** 由于 Spectre 和 Meltdown 的漏洞，所有主流浏览器在 2018 年 1 月就禁用了 SharedArrayBuffer。从 2019 年开始，有些浏览器开始逐步重新启用这一特性。

既不克隆，也不转移，SharedArrayBuffer 作为 ArrayBuffer 能够在不同浏览器上下文间共享。在把 SharedArrayBuffer 传给 postMessage()时，浏览器只会传递原始缓冲区的引用。结果是，两个不同的 JavaScript 上下文会分别维护对同一个内存块的引用。每个上下文都可以随意修改这个缓冲区，就跟修改常规 ArrayBuffer 一样。来看下面的例子：

main.js

```
const worker = new Worker('./worker.js');

// 创建 1 字节缓冲区
const sharedArrayBuffer = new SharedArrayBuffer(1);

// 创建 1 字节缓冲区的视图
const view = new Uint8Array(sharedArrayBuffer);

// 父上下文赋值 1
view[0] = 1;

worker.onmessage = () => {
  console.log(`buffer value after worker modification: ${view[0]}`);
};

// 发送对 sharedArrayBuffer 的引用
worker.postMessage(sharedArrayBuffer);

// buffer value before worker modification: 1
// buffer value after worker modification: 2
```

worker.js

```
self.onmessage = ({data}) => {
  const view = new Uint8Array(data);
```

```
  console.log(`buffer value before worker modification: ${view[0]}`);

  // 工作者线程为共享缓冲区赋值
  view[0] += 1;

  // 发送空消息，通知赋值完成
  self.postMessage(null);
};
```

当然，在两个并行线程中共享内存块有资源争用的风险。换句话说，SharedArrayBuffer 实例实际上会被当成易变（volatile）内存。下面的例子演示了这一点：

main.js

```
// 创建包含 4 个线程的线程池
const workers = [];
for (let i = 0; i < 4; ++i) {
  workers.push(new Worker('./worker.js'));
}

// 在最后一个工作者线程完成后打印最终值
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 发给每个线程
for (const worker of workers) {
  worker.postMessage(sharedArrayBuffer);
}

//  (期待结果为 4000001。实际输出类似于:)
// Final buffer value: 2145106
```

worker.js

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

  // 执行 100 万次加操作
  for (let i = 0; i < 1E6; ++i) {
    view[0] += 1;
  }

  self.postMessage(null);
};
```

这里，每个工作者线程都顺序执行了 100 万次加操作，每次都读取共享数组的索引、执行一次加操作，然后再把值写回数组索引。在所有工作者线程读/写操作交织的过程中就会发生资源争用。例如：

(1) 线程 A 读取到值 1；

(2) 线程 B 读取到值 1；

(3) 线程 A 加 1 并将 2 写回数组；

(4) 线程 B 仍然使用陈旧的数组值 1，同样把 2 写回数组。

为解决该问题，可以使用 Atomics 对象让一个工作者线程获得 SharedArrayBuffer 实例的锁，在执行完全部读/写/读操作后，再允许另一个工作者线程执行操作。把 Atomics.add()放到这个例子中就可以得到正确的最终值：

main.js

```
// 创建包含 4 个线程的线程池
const workers = [];
for (let i = 0; i < 4; ++i) {
  workers.push(new Worker('./worker.js'));
}

// 在最后一个工作者线程完成后打印最终值
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 发给每个线程
for (const worker of workers) {
  worker.postMessage(sharedArrayBuffer);
}

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

worker.js

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

  // 执行 100 万次加操作
  for (let i = 0; i < 1E6; ++i) {
    Atomics.add(view, 0, 1);
}

  self.postMessage(null);
};
```

> **注意** 第 20 章详细介绍了 SharedArrayBuffer 和 Atomics API。

27.2.11　线程池

因为启用工作者线程代价很大，所以某些情况下可以考虑始终保持固定数量的线程活动，需要时就把任务分派给它们。工作者线程在执行计算时，会被标记为忙碌状态。直到它通知线程池自己空闲了，才准备好接收新任务。这些活动线程就称为“线程池”或“工作者线程池”。

线程池中线程的数量多少合适并没有权威的答案，不过可以参考 navigator.hardware Concurrency 属性返回的系统可用的核心数量。因为不太可能知道每个核心的多线程能力，所以最好把这个数字作为线程池大小的上限。

一种使用线程池的策略是每个线程都执行同样的任务，但具体执行什么任务由几个参数来控制。通过使用特定于任务的线程池，可以分配固定数量的工作者线程，并根据需要为他们提供参数。工作者线程会接收这些参数，执行耗时的计算，并把结果返回给线程池。然后线程池可以再将其他工作分派给工作者线程去执行。接下来的例子将构建一个相对简单的线程池，但可以涵盖上述思路的所有基本要求。

首先是定义一个 TaskWorker 类，它可以扩展 Worker 类。TaskWorker 类负责两件事：跟踪线程是否正忙于工作，并管理进出线程的信息与事件。另外，传入给这个工作者线程的任务会封装到一个期约中，然后正确地解决和拒绝。这个类的定义如下：

```
class TaskWorker extends Worker {
  constructor(notifyAvailable, ...workerArgs) {
    super(...workerArgs);

    // 初始化为不可用状态
    this.available = false;
    this.resolve = null;
    this.reject = null;

    // 线程池会传递回调
    // 以便工作者线程发出它需要新任务的信号
    this.notifyAvailable = notifyAvailable;

    // 线程脚本在完全初始化之后
    // 会发送一条"ready"消息
    this.onmessage = () => this.setAvailable();
  }

  // 由线程池调用，以分派新任务
  dispatch({ resolve, reject, postMessageArgs }) {
    this.available = false;

    this.onmessage = ({ data }) => {
      resolve(data);
      this.setAvailable();
    };

    this.onerror = (e) => {
      reject(e);
      this.setAvailable();
    };

    this.postMessage(...postMessageArgs);
  }

  setAvailable() {
```

```
    this.available = true;
    this.resolve = null;
    this.reject = null;
    this.notifyAvailable();
  }
}
```

然后是定义使用 TaskWorker 类的 WorkerPool 类。它还必须维护尚未分派给工作者线程的任务队列。两个事件可以表明应该分派一个新任务：新任务被添加到队列中，或者工作者线程完成了一个任务，应该再发送另一个任务。WorkerPool 类定义如下：

```
class WorkerPool {
  constructor(poolSize, ...workerArgs) {
    this.taskQueue = [];
    this.workers = [];

    // 初始化线程池
    for (let i = 0; i < poolSize; ++i) {
      this.workers.push(
        new TaskWorker(() => this.dispatchIfAvailable(), ...workerArgs));
    }
  }

  // 把任务推入队列
  enqueue(...postMessageArgs) {
    return new Promise((resolve, reject) => {
      this.taskQueue.push({ resolve, reject, postMessageArgs });

      this.dispatchIfAvailable();
    });
  }

  // 把任务发送给下一个空闲的线程（如果有的话）
  dispatchIfAvailable() {
    if (!this.taskQueue.length) {
      return;
    }
    for (const worker of this.workers) {
      if (worker.available) {
        let a = this.taskQueue.shift();
        worker.dispatch(a);
        break;
      }
    }
  }

  // 终止所有工作者线程
  close() {
    for (const worker of this.workers) {
      worker.terminate();
    }
  }
}
```


定义了这两个类之后，现在可以把任务分派到线程池，并在工作者线程可用时执行它们。在这个例子中，假设我们想计算 1000 万个浮点值之和。为节省转移成本，我们使用 SharedArrayBuffer。工作者线程的脚本（worker.js）大致如下：

```
self.onmessage = ({data}) => {
  let sum = 0;
  let view = new Float32Array(data.arrayBuffer)

  // 求和
  for (let i = data.startIdx; i < data.endIdx; ++i) {
    // 不需要原子操作，因为只需要读
    sum += view[i];
  }

  // 把结果发送给工作者线程
  self.postMessage(sum);
};

// 发送消息给 TaskWorker
// 通知工作者线程准备好接收任务了
self.postMessage('ready');
```

有了以上代码，利用线程池分派任务的代码可以这样写：

```
Class TaskWorker {
  ...
]

Class WorkerPool {
  ...
}

const totalFloats = 1E8;
const numTasks = 20;
const floatsPerTask = totalFloats / numTasks;
const numWorkers = 4;

// 创建线程池
const pool = new WorkerPool(numWorkers, './worker.js');

// 填充浮点值数组
let arrayBuffer = new SharedArrayBuffer(4 * totalFloats);
let view = new Float32Array(arrayBuffer);
for (let i = 0; i < totalFloats; ++i) {
  view[i] = Math.random();
}

let partialSumPromises = [];
for (let i = 0; i < totalFloats; i += floatsPerTask) {
  partialSumPromises.push(
    pool.enqueue({
      startIdx: i,
      endIdx: i + floatsPerTask,
      arrayBuffer: arrayBuffer
    })
  );
}

// 等待所有期约完成，然后求和
Promise.all(partialSumPromises)
  .then((partialSums) => partialSums.reduce((x, y) => x + y))
  .then(console.log);

// (在这个例子中，和应该约等于 1E8/2)
// 49997075.47203197
```

> **注意** 草率地采用并行计算不一定是最好的办法。线程池的调优策略会因计算任务不同和系统硬件不同而不同。

27.3 共享工作者线程

共享工作者线程或**共享线程**与专用工作者线程类似，但可以被多个可信任的执行上下文访问。例如，同源的两个标签页可以访问同一个共享工作者线程。`SharedWorker` 与 `Worker` 的消息接口稍有不同，包括外部和内部。

共享线程适合开发者希望通过在多个上下文间共享线程减少计算性消耗的情形。比如，可以用一个共享线程管理多个同源页面 WebSocket 消息的发送与接收。共享线程也可以用在同源上下文希望通过一个线程通信的情形。

27.3.1 共享工作者线程简介

从行为上讲，共享工作者线程可以看作是专用工作者线程的一个扩展。线程创建、线程选项、安全限制和 `importScripts()` 的行为都是相同的。与专用工作者线程一样，共享工作者线程也在独立执行上下文中运行，也只能与其他上下文异步通信。

1. 创建共享工作者线程

与专用工作者线程一样，创建共享工作者线程非常常用的方式是通过加载 JavaScript 文件创建。此时，需要给 `SharedWorker` 构造函数传入文件路径，该构造函数在后台异步加载脚本并实例化共享工作者线程。

下面的例子演示了如何基于绝对路径创建空共享工作者线程：

emptySharedWorker.js

```
// 空的 JavaScript 线程文件
```

main.js

```
console.log(location.href); // "https://example.com/"
const sharedWorker = new SharedWorker(
    location.href + 'emptySharedWorker.js');
console.log(sharedWorker);  // SharedWorker {}
```

前面的例子可以修改为使用相对路径，不过这需要 main.js 和 emptySharedWorker.js 在同一个目录下：

```
const worker = new Worker('./emptyWorker.js');
console.log(worker); // Worker {}
```

也可以在行内脚本中创建共享工作者线程，但这样做没什么意义。因为每个基于行内脚本字符串创建的 `Blob` 都会被赋予自己唯一的浏览器内部 URL，所以行内脚本中创建的共享工作者线程始终是唯一的。这里的原因将在下一节介绍。

2. `SharedWorker` 标识与独占

共享工作者线程与专用工作者线程的一个重要区别在于，虽然 `Worker()` 构造函数始终会创建新实例，而 `SharedWorker()` 则只会在相同的标识不存在的情况下才创建新实例。如果**的确**存在与标识匹配的共享工作者线程，则只会与已有共享者线程建立新的连接。

共享工作者线程标识源自解析后的脚本 URL、工作者线程名称和文档源。例如，下面的脚本将实例

化一个共享工作者线程并添加两个连接：

```
// 实例化一个共享工作者线程
//  - 全部基于同源调用构造函数
//  - 所有脚本解析为相同的 URL
//  - 所有线程都有相同的名称
new SharedWorker('./sharedWorker.js');
new SharedWorker('./sharedWorker.js');
new SharedWorker('./sharedWorker.js');
```

类似地，因为下面三个脚本字符串都解析到相同的 URL，所以也只会创建一个共享工作者线程：

```
// 实例化一个共享工作者线程
//  - 全部基于同源调用构造函数
//  - 所有脚本解析为相同的 URL
//  - 所有线程都有相同的名称
new SharedWorker('./sharedWorker.js');
new SharedWorker('sharedWorker.js');
new SharedWorker('https://www.example.com/sharedWorker.js');
```

因为可选的工作者线程名称也是共享工作者线程标识的一部分，所以不同的线程名称会强制浏览器创建多个共享工作者线程。对下面的例子而言，一个名为'foo'，另一个名为'bar'，尽管它们同源且脚本 URL 相同：

```
// 实例化一个共享工作者线程
//  - 全部基于同源调用构造函数
//  - 所有脚本解析为相同的 URL
//  - 一个线程名称为'foo'，一个线程名称为'bar'
new SharedWorker('./sharedWorker.js', {name: 'foo'});
new SharedWorker('./sharedWorker.js', {name: 'foo'});
new SharedWorker('./sharedWorker.js', {name: 'bar'});
```

共享线程，顾名思义，可以在不同标签页、不同窗口、不同内嵌框架或同源的其他工作者线程之间共享。因此，下面的脚本如果在多个标签页运行，只会在第一次执行时创建一个共享工作者线程，后续执行会连接到该线程：

```
// 实例化一个共享工作者线程
//  - 全部基于同源调用构造函数
//  - 所有脚本解析为相同的 URL
//  - 所有线程都有相同的名称
new SharedWorker('./sharedWorker.js');
```

初始化共享线程的脚本只会限制 URL，因此下面的代码会创建两个共享工作者线程，尽管加载了相同的脚本：

```
// 实例化一个共享工作者线程
//  - 全部基于同源调用构造函数
//  - '?'导致了两个不同的 URL
//  - 所有线程都有相同的名称
new SharedWorker('./sharedWorker.js');
new SharedWorker('./sharedWorker.js?');
```

如果该脚本在两个不同的标签页中运行，同样也只会创建两个共享工作者线程。每个构造函数都会检查匹配的共享工作者线程，然后连接到已存在的那个。

3. 使用 SharedWorker 对象

SharedWorker()构造函数返回的 SharedWorker 对象被用作与新创建的共享工作者线程通信的连接点。它可以用来通过 MessagePort 在共享工作者线程和父上下文间传递信息，也可以用来捕获共享线程中发出的错误事件。

SharedWorker 对象支持以下属性。

- onerror：在共享线程中发生 ErrorEvent 类型的错误事件时会调用指定给该属性的处理程序。
  - 此事件会在共享线程抛出错误时发生。
  - 此事件也可以通过使用 sharedWorker.addEventListener('error', handler)处理。
- port：专门用来跟共享线程通信的 MessagePort。

4. **SharedWorkerGlobalScope**

在共享线程内部，全局作用域是 SharedWorkerGlobalScope 的实例。SharedWorkerGlobalScope 继承自 WorkerGlobalScope，因此包括它所有的属性和方法。与专用工作者线程一样，共享工作者线程也可以通过 self 关键字访问该全局上下文。

SharedWorkerGlobalScope 通过以下属性和方法扩展了 WorkerGlobalScope。

- name：可选的字符串标识符，可以传给 SharedWorker 构造函数。
- importScripts()：用于向工作者线程中导入任意数量的脚本。
- close()：与 worker.terminate()对应，用于立即终止工作者线程。没有给工作者线程提供终止前清理的机会；脚本会突然停止。
- onconnect：与共享线程建立新连接时，应将其设置为处理程序。connect 事件包括 MessagePort 实例的 ports 数组，可用于把消息发送回父上下文。
  - 在通过 worker.port.onmessage 或 worker.port.start()与共享线程建立连接时都会触发 connect 事件。
  - connect 事件也可以通过使用 sharedWorker.addEventListener('connect', handler)处理。

> **注意** 根据浏览器实现，在 SharedWorker 中把日志打印到控制台不一定能在浏览器默认的控制台中看到。

27.3.2 理解共享工作者线程的生命周期

共享工作者线程的生命周期具有与专用工作者线程相同的阶段的特性。不同之处在于，专用工作者线程只跟一个页面绑定，而共享工作者线程只要还有一个上下文连接就会持续存在。

比如下面的脚本，每次调用它都会创建一个专用工作者线程：

```
new Worker('./worker.js');
```

下表详细列出了当三个包含此脚本的标签页按顺序打开和关闭时会发生什么。

事　件	结　果	事件发生后的线程数
标签页 1 执行 main.js	创建专用线程 1	1
标签页 2 执行 main.js	创建专用线程 2	2
标签页 3 执行 main.js	创建专用线程 3	3
标签页 1 关闭	专用线程 1 终止	2
标签页 2 关闭	专用线程 2 终止	1
标签页 3 关闭	专用线程 3 终止	0