/\*\*

\* The `http2` module provides an implementation of the [HTTP/2](https://tools.ietf.org/html/rfc7540) protocol. It

\* can be accessed using:

\*

\* ```js

\* const http2 = require('http2');

\* ```

\* @since v8.4.0

\* @see [source](https://github.com/nodejs/node/blob/v17.0.0/lib/http2.js)

\*/

declare module 'http2' {

import EventEmitter = require('node:events');

import \* as fs from 'node:fs';

import \* as net from 'node:net';

import \* as stream from 'node:stream';

import \* as tls from 'node:tls';

import \* as url from 'node:url';

import { IncomingHttpHeaders as Http1IncomingHttpHeaders, OutgoingHttpHeaders, IncomingMessage, ServerResponse } from 'node:http';

export { OutgoingHttpHeaders } from 'node:http';

export interface IncomingHttpStatusHeader {

':status'?: number | undefined;

}

export interface IncomingHttpHeaders extends Http1IncomingHttpHeaders {

':path'?: string | undefined;

':method'?: string | undefined;

':authority'?: string | undefined;

':scheme'?: string | undefined;

}

// Http2Stream

export interface StreamPriorityOptions {

exclusive?: boolean | undefined;

parent?: number | undefined;

weight?: number | undefined;

silent?: boolean | undefined;

}

export interface StreamState {

localWindowSize?: number | undefined;

state?: number | undefined;

localClose?: number | undefined;

remoteClose?: number | undefined;

sumDependencyWeight?: number | undefined;

weight?: number | undefined;

}

export interface ServerStreamResponseOptions {

endStream?: boolean | undefined;

waitForTrailers?: boolean | undefined;

}

export interface StatOptions {

offset: number;

length: number;

}

export interface ServerStreamFileResponseOptions {

statCheck?(stats: fs.Stats, headers: OutgoingHttpHeaders, statOptions: StatOptions): void | boolean;

waitForTrailers?: boolean | undefined;

offset?: number | undefined;

length?: number | undefined;

}

export interface ServerStreamFileResponseOptionsWithError extends ServerStreamFileResponseOptions {

onError?(err: NodeJS.ErrnoException): void;

}

export interface Http2Stream extends stream.Duplex {

/\*\*

\* Set to `true` if the `Http2Stream` instance was aborted abnormally. When set,

\* the `'aborted'` event will have been emitted.

\* @since v8.4.0

\*/

readonly aborted: boolean;

/\*\*

\* This property shows the number of characters currently buffered to be written.

\* See `net.Socket.bufferSize` for details.

\* @since v11.2.0, v10.16.0

\*/

readonly bufferSize: number;

/\*\*

\* Set to `true` if the `Http2Stream` instance has been closed.

\* @since v9.4.0

\*/

readonly closed: boolean;

/\*\*

\* Set to `true` if the `Http2Stream` instance has been destroyed and is no longer

\* usable.

\* @since v8.4.0

\*/

readonly destroyed: boolean;

/\*\*

\* Set the `true` if the `END\_STREAM` flag was set in the request or response

\* HEADERS frame received, indicating that no additional data should be received

\* and the readable side of the `Http2Stream` will be closed.

\* @since v10.11.0

\*/

readonly endAfterHeaders: boolean;

/\*\*

\* The numeric stream identifier of this `Http2Stream` instance. Set to `undefined`if the stream identifier has not yet been assigned.

\* @since v8.4.0

\*/

readonly id?: number | undefined;

/\*\*

\* Set to `true` if the `Http2Stream` instance has not yet been assigned a

\* numeric stream identifier.

\* @since v9.4.0

\*/

readonly pending: boolean;

/\*\*

\* Set to the `RST\_STREAM` `error code` reported when the `Http2Stream` is

\* destroyed after either receiving an `RST\_STREAM` frame from the connected peer,

\* calling `http2stream.close()`, or `http2stream.destroy()`. Will be`undefined` if the `Http2Stream` has not been closed.

\* @since v8.4.0

\*/

readonly rstCode: number;

/\*\*

\* An object containing the outbound headers sent for this `Http2Stream`.

\* @since v9.5.0

\*/

readonly sentHeaders: OutgoingHttpHeaders;

/\*\*

\* An array of objects containing the outbound informational (additional) headers

\* sent for this `Http2Stream`.

\* @since v9.5.0

\*/

readonly sentInfoHeaders?: OutgoingHttpHeaders[] | undefined;

/\*\*

\* An object containing the outbound trailers sent for this `HttpStream`.

\* @since v9.5.0

\*/

readonly sentTrailers?: OutgoingHttpHeaders | undefined;

/\*\*

\* A reference to the `Http2Session` instance that owns this `Http2Stream`. The

\* value will be `undefined` after the `Http2Stream` instance is destroyed.

\* @since v8.4.0

\*/

readonly session: Http2Session;

/\*\*

\* Provides miscellaneous information about the current state of the`Http2Stream`.

\*

\* A current state of this `Http2Stream`.

\* @since v8.4.0

\*/

readonly state: StreamState;

/\*\*

\* Closes the `Http2Stream` instance by sending an `RST\_STREAM` frame to the

\* connected HTTP/2 peer.

\* @since v8.4.0

\* @param [code=http2.constants.NGHTTP2\_NO\_ERROR] Unsigned 32-bit integer identifying the error code.

\* @param callback An optional function registered to listen for the `'close'` event.

\*/

close(code?: number, callback?: () => void): void;

/\*\*

\* Updates the priority for this `Http2Stream` instance.

\* @since v8.4.0

\*/

priority(options: StreamPriorityOptions): void;

/\*\*

\* ```js

\* const http2 = require('http2');

\* const client = http2.connect('http://example.org:8000');

\* const { NGHTTP2\_CANCEL } = http2.constants;

\* const req = client.request({ ':path': '/' });

\*

\* // Cancel the stream if there's no activity after 5 seconds

\* req.setTimeout(5000, () => req.close(NGHTTP2\_CANCEL));

\* ```

\* @since v8.4.0

\*/

setTimeout(msecs: number, callback?: () => void): void;

/\*\*

\* Sends a trailing `HEADERS` frame to the connected HTTP/2 peer. This method

\* will cause the `Http2Stream` to be immediately closed and must only be

\* called after the `'wantTrailers'` event has been emitted. When sending a

\* request or sending a response, the `options.waitForTrailers` option must be set

\* in order to keep the `Http2Stream` open after the final `DATA` frame so that

\* trailers can be sent.

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* stream.respond(undefined, { waitForTrailers: true });

\* stream.on('wantTrailers', () => {

\* stream.sendTrailers({ xyz: 'abc' });

\* });

\* stream.end('Hello World');

\* });

\* ```

\*

\* The HTTP/1 specification forbids trailers from containing HTTP/2 pseudo-header

\* fields (e.g. `':method'`, `':path'`, etc).

\* @since v10.0.0

\*/

sendTrailers(headers: OutgoingHttpHeaders): void;

addListener(event: 'aborted', listener: () => void): this;

addListener(event: 'close', listener: () => void): this;

addListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

addListener(event: 'drain', listener: () => void): this;

addListener(event: 'end', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'finish', listener: () => void): this;

addListener(event: 'frameError', listener: (frameType: number, errorCode: number) => void): this;

addListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

addListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

addListener(event: 'streamClosed', listener: (code: number) => void): this;

addListener(event: 'timeout', listener: () => void): this;

addListener(event: 'trailers', listener: (trailers: IncomingHttpHeaders, flags: number) => void): this;

addListener(event: 'wantTrailers', listener: () => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'aborted'): boolean;

emit(event: 'close'): boolean;

emit(event: 'data', chunk: Buffer | string): boolean;

emit(event: 'drain'): boolean;

emit(event: 'end'): boolean;

emit(event: 'error', err: Error): boolean;

emit(event: 'finish'): boolean;

emit(event: 'frameError', frameType: number, errorCode: number): boolean;

emit(event: 'pipe', src: stream.Readable): boolean;

emit(event: 'unpipe', src: stream.Readable): boolean;

emit(event: 'streamClosed', code: number): boolean;

emit(event: 'timeout'): boolean;

emit(event: 'trailers', trailers: IncomingHttpHeaders, flags: number): boolean;

emit(event: 'wantTrailers'): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'aborted', listener: () => void): this;

on(event: 'close', listener: () => void): this;

on(event: 'data', listener: (chunk: Buffer | string) => void): this;

on(event: 'drain', listener: () => void): this;

on(event: 'end', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'finish', listener: () => void): this;

on(event: 'frameError', listener: (frameType: number, errorCode: number) => void): this;

on(event: 'pipe', listener: (src: stream.Readable) => void): this;

on(event: 'unpipe', listener: (src: stream.Readable) => void): this;

on(event: 'streamClosed', listener: (code: number) => void): this;

on(event: 'timeout', listener: () => void): this;

on(event: 'trailers', listener: (trailers: IncomingHttpHeaders, flags: number) => void): this;

on(event: 'wantTrailers', listener: () => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'aborted', listener: () => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'data', listener: (chunk: Buffer | string) => void): this;

once(event: 'drain', listener: () => void): this;

once(event: 'end', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'finish', listener: () => void): this;

once(event: 'frameError', listener: (frameType: number, errorCode: number) => void): this;

once(event: 'pipe', listener: (src: stream.Readable) => void): this;

once(event: 'unpipe', listener: (src: stream.Readable) => void): this;

once(event: 'streamClosed', listener: (code: number) => void): this;

once(event: 'timeout', listener: () => void): this;

once(event: 'trailers', listener: (trailers: IncomingHttpHeaders, flags: number) => void): this;

once(event: 'wantTrailers', listener: () => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'aborted', listener: () => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

prependListener(event: 'drain', listener: () => void): this;

prependListener(event: 'end', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'finish', listener: () => void): this;

prependListener(event: 'frameError', listener: (frameType: number, errorCode: number) => void): this;

prependListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

prependListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

prependListener(event: 'streamClosed', listener: (code: number) => void): this;

prependListener(event: 'timeout', listener: () => void): this;

prependListener(event: 'trailers', listener: (trailers: IncomingHttpHeaders, flags: number) => void): this;

prependListener(event: 'wantTrailers', listener: () => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'aborted', listener: () => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

prependOnceListener(event: 'drain', listener: () => void): this;

prependOnceListener(event: 'end', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'finish', listener: () => void): this;

prependOnceListener(event: 'frameError', listener: (frameType: number, errorCode: number) => void): this;

prependOnceListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

prependOnceListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

prependOnceListener(event: 'streamClosed', listener: (code: number) => void): this;

prependOnceListener(event: 'timeout', listener: () => void): this;

prependOnceListener(event: 'trailers', listener: (trailers: IncomingHttpHeaders, flags: number) => void): this;

prependOnceListener(event: 'wantTrailers', listener: () => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

export interface ClientHttp2Stream extends Http2Stream {

addListener(event: 'continue', listener: () => {}): this;

addListener(event: 'headers', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

addListener(event: 'push', listener: (headers: IncomingHttpHeaders, flags: number) => void): this;

addListener(event: 'response', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'continue'): boolean;

emit(event: 'headers', headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number): boolean;

emit(event: 'push', headers: IncomingHttpHeaders, flags: number): boolean;

emit(event: 'response', headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'continue', listener: () => {}): this;

on(event: 'headers', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

on(event: 'push', listener: (headers: IncomingHttpHeaders, flags: number) => void): this;

on(event: 'response', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'continue', listener: () => {}): this;

once(event: 'headers', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

once(event: 'push', listener: (headers: IncomingHttpHeaders, flags: number) => void): this;

once(event: 'response', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'continue', listener: () => {}): this;

prependListener(event: 'headers', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

prependListener(event: 'push', listener: (headers: IncomingHttpHeaders, flags: number) => void): this;

prependListener(event: 'response', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'continue', listener: () => {}): this;

prependOnceListener(event: 'headers', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

prependOnceListener(event: 'push', listener: (headers: IncomingHttpHeaders, flags: number) => void): this;

prependOnceListener(event: 'response', listener: (headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

export interface ServerHttp2Stream extends Http2Stream {

/\*\*

\* True if headers were sent, false otherwise (read-only).

\* @since v8.4.0

\*/

readonly headersSent: boolean;

/\*\*

\* Read-only property mapped to the `SETTINGS\_ENABLE\_PUSH` flag of the remote

\* client's most recent `SETTINGS` frame. Will be `true` if the remote peer

\* accepts push streams, `false` otherwise. Settings are the same for every`Http2Stream` in the same `Http2Session`.

\* @since v8.4.0

\*/

readonly pushAllowed: boolean;

/\*\*

\* Sends an additional informational `HEADERS` frame to the connected HTTP/2 peer.

\* @since v8.4.0

\*/

additionalHeaders(headers: OutgoingHttpHeaders): void;

/\*\*

\* Initiates a push stream. The callback is invoked with the new `Http2Stream`instance created for the push stream passed as the second argument, or an`Error` passed as the first argument.

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* stream.respond({ ':status': 200 });

\* stream.pushStream({ ':path': '/' }, (err, pushStream, headers) => {

\* if (err) throw err;

\* pushStream.respond({ ':status': 200 });

\* pushStream.end('some pushed data');

\* });

\* stream.end('some data');

\* });

\* ```

\*

\* Setting the weight of a push stream is not allowed in the `HEADERS` frame. Pass

\* a `weight` value to `http2stream.priority` with the `silent` option set to`true` to enable server-side bandwidth balancing between concurrent streams.

\*

\* Calling `http2stream.pushStream()` from within a pushed stream is not permitted

\* and will throw an error.

\* @since v8.4.0

\* @param callback Callback that is called once the push stream has been initiated.

\*/

pushStream(headers: OutgoingHttpHeaders, callback?: (err: Error | null, pushStream: ServerHttp2Stream, headers: OutgoingHttpHeaders) => void): void;

pushStream(headers: OutgoingHttpHeaders, options?: StreamPriorityOptions, callback?: (err: Error | null, pushStream: ServerHttp2Stream, headers: OutgoingHttpHeaders) => void): void;

/\*\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* stream.respond({ ':status': 200 });

\* stream.end('some data');

\* });

\* ```

\*

\* When the `options.waitForTrailers` option is set, the `'wantTrailers'` event

\* will be emitted immediately after queuing the last chunk of payload data to be

\* sent. The `http2stream.sendTrailers()` method can then be used to sent trailing

\* header fields to the peer.

\*

\* When `options.waitForTrailers` is set, the `Http2Stream` will not automatically

\* close when the final `DATA` frame is transmitted. User code must call either`http2stream.sendTrailers()` or `http2stream.close()` to close the`Http2Stream`.

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* stream.respond({ ':status': 200 }, { waitForTrailers: true });

\* stream.on('wantTrailers', () => {

\* stream.sendTrailers({ ABC: 'some value to send' });

\* });

\* stream.end('some data');

\* });

\* ```

\* @since v8.4.0

\*/

respond(headers?: OutgoingHttpHeaders, options?: ServerStreamResponseOptions): void;

/\*\*

\* Initiates a response whose data is read from the given file descriptor. No

\* validation is performed on the given file descriptor. If an error occurs while

\* attempting to read data using the file descriptor, the `Http2Stream` will be

\* closed using an `RST\_STREAM` frame using the standard `INTERNAL\_ERROR` code.

\*

\* When used, the `Http2Stream` object's `Duplex` interface will be closed

\* automatically.

\*

\* ```js

\* const http2 = require('http2');

\* const fs = require('fs');

\*

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* const fd = fs.openSync('/some/file', 'r');

\*

\* const stat = fs.fstatSync(fd);

\* const headers = {

\* 'content-length': stat.size,

\* 'last-modified': stat.mtime.toUTCString(),

\* 'content-type': 'text/plain; charset=utf-8'

\* };

\* stream.respondWithFD(fd, headers);

\* stream.on('close', () => fs.closeSync(fd));

\* });

\* ```

\*

\* The optional `options.statCheck` function may be specified to give user code

\* an opportunity to set additional content headers based on the `fs.Stat` details

\* of the given fd. If the `statCheck` function is provided, the`http2stream.respondWithFD()` method will perform an `fs.fstat()` call to

\* collect details on the provided file descriptor.

\*

\* The `offset` and `length` options may be used to limit the response to a

\* specific range subset. This can be used, for instance, to support HTTP Range

\* requests.

\*

\* The file descriptor or `FileHandle` is not closed when the stream is closed,

\* so it will need to be closed manually once it is no longer needed.

\* Using the same file descriptor concurrently for multiple streams

\* is not supported and may result in data loss. Re-using a file descriptor

\* after a stream has finished is supported.

\*

\* When the `options.waitForTrailers` option is set, the `'wantTrailers'` event

\* will be emitted immediately after queuing the last chunk of payload data to be

\* sent. The `http2stream.sendTrailers()` method can then be used to sent trailing

\* header fields to the peer.

\*

\* When `options.waitForTrailers` is set, the `Http2Stream` will not automatically

\* close when the final `DATA` frame is transmitted. User code \_must\_ call either`http2stream.sendTrailers()` or `http2stream.close()` to close the`Http2Stream`.

\*

\* ```js

\* const http2 = require('http2');

\* const fs = require('fs');

\*

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* const fd = fs.openSync('/some/file', 'r');

\*

\* const stat = fs.fstatSync(fd);

\* const headers = {

\* 'content-length': stat.size,

\* 'last-modified': stat.mtime.toUTCString(),

\* 'content-type': 'text/plain; charset=utf-8'

\* };

\* stream.respondWithFD(fd, headers, { waitForTrailers: true });

\* stream.on('wantTrailers', () => {

\* stream.sendTrailers({ ABC: 'some value to send' });

\* });

\*

\* stream.on('close', () => fs.closeSync(fd));

\* });

\* ```

\* @since v8.4.0

\* @param fd A readable file descriptor.

\*/

respondWithFD(fd: number | fs.promises.FileHandle, headers?: OutgoingHttpHeaders, options?: ServerStreamFileResponseOptions): void;

/\*\*

\* Sends a regular file as the response. The `path` must specify a regular file

\* or an `'error'` event will be emitted on the `Http2Stream` object.

\*

\* When used, the `Http2Stream` object's `Duplex` interface will be closed

\* automatically.

\*

\* The optional `options.statCheck` function may be specified to give user code

\* an opportunity to set additional content headers based on the `fs.Stat` details

\* of the given file:

\*

\* If an error occurs while attempting to read the file data, the `Http2Stream`will be closed using an `RST\_STREAM` frame using the standard `INTERNAL\_ERROR`code. If the `onError` callback is

\* defined, then it will be called. Otherwise

\* the stream will be destroyed.

\*

\* Example using a file path:

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* function statCheck(stat, headers) {

\* headers['last-modified'] = stat.mtime.toUTCString();

\* }

\*

\* function onError(err) {

\* // stream.respond() can throw if the stream has been destroyed by

\* // the other side.

\* try {

\* if (err.code === 'ENOENT') {

\* stream.respond({ ':status': 404 });

\* } else {

\* stream.respond({ ':status': 500 });

\* }

\* } catch (err) {

\* // Perform actual error handling.

\* console.log(err);

\* }

\* stream.end();

\* }

\*

\* stream.respondWithFile('/some/file',

\* { 'content-type': 'text/plain; charset=utf-8' },

\* { statCheck, onError });

\* });

\* ```

\*

\* The `options.statCheck` function may also be used to cancel the send operation

\* by returning `false`. For instance, a conditional request may check the stat

\* results to determine if the file has been modified to return an appropriate`304` response:

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* function statCheck(stat, headers) {

\* // Check the stat here...

\* stream.respond({ ':status': 304 });

\* return false; // Cancel the send operation

\* }

\* stream.respondWithFile('/some/file',

\* { 'content-type': 'text/plain; charset=utf-8' },

\* { statCheck });

\* });

\* ```

\*

\* The `content-length` header field will be automatically set.

\*

\* The `offset` and `length` options may be used to limit the response to a

\* specific range subset. This can be used, for instance, to support HTTP Range

\* requests.

\*

\* The `options.onError` function may also be used to handle all the errors

\* that could happen before the delivery of the file is initiated. The

\* default behavior is to destroy the stream.

\*

\* When the `options.waitForTrailers` option is set, the `'wantTrailers'` event

\* will be emitted immediately after queuing the last chunk of payload data to be

\* sent. The `http2stream.sendTrailers()` method can then be used to sent trailing

\* header fields to the peer.

\*

\* When `options.waitForTrailers` is set, the `Http2Stream` will not automatically

\* close when the final `DATA` frame is transmitted. User code must call either`http2stream.sendTrailers()` or `http2stream.close()` to close the`Http2Stream`.

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer();

\* server.on('stream', (stream) => {

\* stream.respondWithFile('/some/file',

\* { 'content-type': 'text/plain; charset=utf-8' },

\* { waitForTrailers: true });

\* stream.on('wantTrailers', () => {

\* stream.sendTrailers({ ABC: 'some value to send' });

\* });

\* });

\* ```

\* @since v8.4.0

\*/

respondWithFile(path: string, headers?: OutgoingHttpHeaders, options?: ServerStreamFileResponseOptionsWithError): void;

}

// Http2Session

export interface Settings {

headerTableSize?: number | undefined;

enablePush?: boolean | undefined;

initialWindowSize?: number | undefined;

maxFrameSize?: number | undefined;

maxConcurrentStreams?: number | undefined;

maxHeaderListSize?: number | undefined;

enableConnectProtocol?: boolean | undefined;

}

export interface ClientSessionRequestOptions {

endStream?: boolean | undefined;

exclusive?: boolean | undefined;

parent?: number | undefined;

weight?: number | undefined;

waitForTrailers?: boolean | undefined;

}

export interface SessionState {

effectiveLocalWindowSize?: number | undefined;

effectiveRecvDataLength?: number | undefined;

nextStreamID?: number | undefined;

localWindowSize?: number | undefined;

lastProcStreamID?: number | undefined;

remoteWindowSize?: number | undefined;

outboundQueueSize?: number | undefined;

deflateDynamicTableSize?: number | undefined;

inflateDynamicTableSize?: number | undefined;

}

export interface Http2Session extends EventEmitter {

/\*\*

\* Value will be `undefined` if the `Http2Session` is not yet connected to a

\* socket, `h2c` if the `Http2Session` is not connected to a `TLSSocket`, or

\* will return the value of the connected `TLSSocket`'s own `alpnProtocol`property.

\* @since v9.4.0

\*/

readonly alpnProtocol?: string | undefined;

/\*\*

\* Will be `true` if this `Http2Session` instance has been closed, otherwise`false`.

\* @since v9.4.0

\*/

readonly closed: boolean;

/\*\*

\* Will be `true` if this `Http2Session` instance is still connecting, will be set

\* to `false` before emitting `connect` event and/or calling the `http2.connect`callback.

\* @since v10.0.0

\*/

readonly connecting: boolean;

/\*\*

\* Will be `true` if this `Http2Session` instance has been destroyed and must no

\* longer be used, otherwise `false`.

\* @since v8.4.0

\*/

readonly destroyed: boolean;

/\*\*

\* Value is `undefined` if the `Http2Session` session socket has not yet been

\* connected, `true` if the `Http2Session` is connected with a `TLSSocket`,

\* and `false` if the `Http2Session` is connected to any other kind of socket

\* or stream.

\* @since v9.4.0

\*/

readonly encrypted?: boolean | undefined;

/\*\*

\* A prototype-less object describing the current local settings of this`Http2Session`. The local settings are local to \_this\_`Http2Session` instance.

\* @since v8.4.0

\*/

readonly localSettings: Settings;

/\*\*

\* If the `Http2Session` is connected to a `TLSSocket`, the `originSet` property

\* will return an `Array` of origins for which the `Http2Session` may be

\* considered authoritative.

\*

\* The `originSet` property is only available when using a secure TLS connection.

\* @since v9.4.0

\*/

readonly originSet?: string[] | undefined;

/\*\*

\* Indicates whether the `Http2Session` is currently waiting for acknowledgment of

\* a sent `SETTINGS` frame. Will be `true` after calling the`http2session.settings()` method. Will be `false` once all sent `SETTINGS`frames have been acknowledged.

\* @since v8.4.0

\*/

readonly pendingSettingsAck: boolean;

/\*\*

\* A prototype-less object describing the current remote settings of this`Http2Session`. The remote settings are set by the \_connected\_ HTTP/2 peer.

\* @since v8.4.0

\*/

readonly remoteSettings: Settings;

/\*\*

\* Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but

\* limits available methods to ones safe to use with HTTP/2.

\*

\* `destroy`, `emit`, `end`, `pause`, `read`, `resume`, and `write` will throw

\* an error with code `ERR\_HTTP2\_NO\_SOCKET\_MANIPULATION`. See `Http2Session and Sockets` for more information.

\*

\* `setTimeout` method will be called on this `Http2Session`.

\*

\* All other interactions will be routed directly to the socket.

\* @since v8.4.0

\*/

readonly socket: net.Socket | tls.TLSSocket;

/\*\*

\* Provides miscellaneous information about the current state of the`Http2Session`.

\*

\* An object describing the current status of this `Http2Session`.

\* @since v8.4.0

\*/

readonly state: SessionState;

/\*\*

\* The `http2session.type` will be equal to`http2.constants.NGHTTP2\_SESSION\_SERVER` if this `Http2Session` instance is a

\* server, and `http2.constants.NGHTTP2\_SESSION\_CLIENT` if the instance is a

\* client.

\* @since v8.4.0

\*/

readonly type: number;

/\*\*

\* Gracefully closes the `Http2Session`, allowing any existing streams to

\* complete on their own and preventing new `Http2Stream` instances from being

\* created. Once closed, `http2session.destroy()`\_might\_ be called if there

\* are no open `Http2Stream` instances.

\*

\* If specified, the `callback` function is registered as a handler for the`'close'` event.

\* @since v9.4.0

\*/

close(callback?: () => void): void;

/\*\*

\* Immediately terminates the `Http2Session` and the associated `net.Socket` or`tls.TLSSocket`.

\*

\* Once destroyed, the `Http2Session` will emit the `'close'` event. If `error`is not undefined, an `'error'` event will be emitted immediately before the`'close'` event.

\*

\* If there are any remaining open `Http2Streams` associated with the`Http2Session`, those will also be destroyed.

\* @since v8.4.0

\* @param error An `Error` object if the `Http2Session` is being destroyed due to an error.

\* @param code The HTTP/2 error code to send in the final `GOAWAY` frame. If unspecified, and `error` is not undefined, the default is `INTERNAL\_ERROR`, otherwise defaults to `NO\_ERROR`.

\*/

destroy(error?: Error, code?: number): void;

/\*\*

\* Transmits a `GOAWAY` frame to the connected peer \_without\_ shutting down the`Http2Session`.

\* @since v9.4.0

\* @param code An HTTP/2 error code

\* @param lastStreamID The numeric ID of the last processed `Http2Stream`

\* @param opaqueData A `TypedArray` or `DataView` instance containing additional data to be carried within the `GOAWAY` frame.

\*/

goaway(code?: number, lastStreamID?: number, opaqueData?: NodeJS.ArrayBufferView): void;

/\*\*

\* Sends a `PING` frame to the connected HTTP/2 peer. A `callback` function must

\* be provided. The method will return `true` if the `PING` was sent, `false`otherwise.

\*

\* The maximum number of outstanding (unacknowledged) pings is determined by the`maxOutstandingPings` configuration option. The default maximum is 10.

\*

\* If provided, the `payload` must be a `Buffer`, `TypedArray`, or `DataView`containing 8 bytes of data that will be transmitted with the `PING` and

\* returned with the ping acknowledgment.

\*

\* The callback will be invoked with three arguments: an error argument that will

\* be `null` if the `PING` was successfully acknowledged, a `duration` argument

\* that reports the number of milliseconds elapsed since the ping was sent and the

\* acknowledgment was received, and a `Buffer` containing the 8-byte `PING`payload.

\*

\* ```js

\* session.ping(Buffer.from('abcdefgh'), (err, duration, payload) => {

\* if (!err) {

\* console.log(`Ping acknowledged in ${duration} milliseconds`);

\* console.log(`With payload '${payload.toString()}'`);

\* }

\* });

\* ```

\*

\* If the `payload` argument is not specified, the default payload will be the

\* 64-bit timestamp (little endian) marking the start of the `PING` duration.

\* @since v8.9.3

\* @param payload Optional ping payload.

\*/

ping(callback: (err: Error | null, duration: number, payload: Buffer) => void): boolean;

ping(payload: NodeJS.ArrayBufferView, callback: (err: Error | null, duration: number, payload: Buffer) => void): boolean;

/\*\*

\* Calls `ref()` on this `Http2Session`instance's underlying `net.Socket`.

\* @since v9.4.0

\*/

ref(): void;

/\*\*

\* Sets the local endpoint's window size.

\* The `windowSize` is the total window size to set, not

\* the delta.

\*

\* ```js

\* const http2 = require('http2');

\*

\* const server = http2.createServer();

\* const expectedWindowSize = 2 \*\* 20;

\* server.on('connect', (session) => {

\*

\* // Set local window size to be 2 \*\* 20

\* session.setLocalWindowSize(expectedWindowSize);

\* });

\* ```

\* @since v15.3.0, v14.18.0

\*/

setLocalWindowSize(windowSize: number): void;

/\*\*

\* Used to set a callback function that is called when there is no activity on

\* the `Http2Session` after `msecs` milliseconds. The given `callback` is

\* registered as a listener on the `'timeout'` event.

\* @since v8.4.0

\*/

setTimeout(msecs: number, callback?: () => void): void;

/\*\*

\* Updates the current local settings for this `Http2Session` and sends a new`SETTINGS` frame to the connected HTTP/2 peer.

\*

\* Once called, the `http2session.pendingSettingsAck` property will be `true`while the session is waiting for the remote peer to acknowledge the new

\* settings.

\*

\* The new settings will not become effective until the `SETTINGS` acknowledgment

\* is received and the `'localSettings'` event is emitted. It is possible to send

\* multiple `SETTINGS` frames while acknowledgment is still pending.

\* @since v8.4.0

\* @param callback Callback that is called once the session is connected or right away if the session is already connected.

\*/

settings(settings: Settings): void;

/\*\*

\* Calls `unref()` on this `Http2Session`instance's underlying `net.Socket`.

\* @since v9.4.0

\*/

unref(): void;

addListener(event: 'close', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: 'frameError', listener: (frameType: number, errorCode: number, streamID: number) => void): this;

addListener(event: 'goaway', listener: (errorCode: number, lastStreamID: number, opaqueData: Buffer) => void): this;

addListener(event: 'localSettings', listener: (settings: Settings) => void): this;

addListener(event: 'ping', listener: () => void): this;

addListener(event: 'remoteSettings', listener: (settings: Settings) => void): this;

addListener(event: 'timeout', listener: () => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'close'): boolean;

emit(event: 'error', err: Error): boolean;

emit(event: 'frameError', frameType: number, errorCode: number, streamID: number): boolean;

emit(event: 'goaway', errorCode: number, lastStreamID: number, opaqueData: Buffer): boolean;

emit(event: 'localSettings', settings: Settings): boolean;

emit(event: 'ping'): boolean;

emit(event: 'remoteSettings', settings: Settings): boolean;

emit(event: 'timeout'): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'close', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: 'frameError', listener: (frameType: number, errorCode: number, streamID: number) => void): this;

on(event: 'goaway', listener: (errorCode: number, lastStreamID: number, opaqueData: Buffer) => void): this;

on(event: 'localSettings', listener: (settings: Settings) => void): this;

on(event: 'ping', listener: () => void): this;

on(event: 'remoteSettings', listener: (settings: Settings) => void): this;

on(event: 'timeout', listener: () => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: 'frameError', listener: (frameType: number, errorCode: number, streamID: number) => void): this;

once(event: 'goaway', listener: (errorCode: number, lastStreamID: number, opaqueData: Buffer) => void): this;

once(event: 'localSettings', listener: (settings: Settings) => void): this;

once(event: 'ping', listener: () => void): this;

once(event: 'remoteSettings', listener: (settings: Settings) => void): this;

once(event: 'timeout', listener: () => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: 'frameError', listener: (frameType: number, errorCode: number, streamID: number) => void): this;

prependListener(event: 'goaway', listener: (errorCode: number, lastStreamID: number, opaqueData: Buffer) => void): this;

prependListener(event: 'localSettings', listener: (settings: Settings) => void): this;

prependListener(event: 'ping', listener: () => void): this;

prependListener(event: 'remoteSettings', listener: (settings: Settings) => void): this;

prependListener(event: 'timeout', listener: () => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: 'frameError', listener: (frameType: number, errorCode: number, streamID: number) => void): this;

prependOnceListener(event: 'goaway', listener: (errorCode: number, lastStreamID: number, opaqueData: Buffer) => void): this;

prependOnceListener(event: 'localSettings', listener: (settings: Settings) => void): this;

prependOnceListener(event: 'ping', listener: () => void): this;

prependOnceListener(event: 'remoteSettings', listener: (settings: Settings) => void): this;

prependOnceListener(event: 'timeout', listener: () => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

export interface ClientHttp2Session extends Http2Session {

/\*\*

\* For HTTP/2 Client `Http2Session` instances only, the `http2session.request()`creates and returns an `Http2Stream` instance that can be used to send an

\* HTTP/2 request to the connected server.

\*

\* This method is only available if `http2session.type` is equal to`http2.constants.NGHTTP2\_SESSION\_CLIENT`.

\*

\* ```js

\* const http2 = require('http2');

\* const clientSession = http2.connect('https://localhost:1234');

\* const {

\* HTTP2\_HEADER\_PATH,

\* HTTP2\_HEADER\_STATUS

\* } = http2.constants;

\*

\* const req = clientSession.request({ [HTTP2\_HEADER\_PATH]: '/' });

\* req.on('response', (headers) => {

\* console.log(headers[HTTP2\_HEADER\_STATUS]);

\* req.on('data', (chunk) => { // .. });

\* req.on('end', () => { // .. });

\* });

\* ```

\*

\* When the `options.waitForTrailers` option is set, the `'wantTrailers'` event

\* is emitted immediately after queuing the last chunk of payload data to be sent.

\* The `http2stream.sendTrailers()` method can then be called to send trailing

\* headers to the peer.

\*

\* When `options.waitForTrailers` is set, the `Http2Stream` will not automatically

\* close when the final `DATA` frame is transmitted. User code must call either`http2stream.sendTrailers()` or `http2stream.close()` to close the`Http2Stream`.

\*

\* When `options.signal` is set with an `AbortSignal` and then `abort` on the

\* corresponding `AbortController` is called, the request will emit an `'error'`event with an `AbortError` error.

\*

\* The `:method` and `:path` pseudo-headers are not specified within `headers`,

\* they respectively default to:

\*

\* \* `:method` \= `'GET'`

\* \* `:path` \= `/`

\* @since v8.4.0

\*/

request(headers?: OutgoingHttpHeaders, options?: ClientSessionRequestOptions): ClientHttp2Stream;

addListener(event: 'altsvc', listener: (alt: string, origin: string, stream: number) => void): this;

addListener(event: 'origin', listener: (origins: string[]) => void): this;

addListener(event: 'connect', listener: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

addListener(event: 'stream', listener: (stream: ClientHttp2Stream, headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'altsvc', alt: string, origin: string, stream: number): boolean;

emit(event: 'origin', origins: ReadonlyArray<string>): boolean;

emit(event: 'connect', session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket): boolean;

emit(event: 'stream', stream: ClientHttp2Stream, headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'altsvc', listener: (alt: string, origin: string, stream: number) => void): this;

on(event: 'origin', listener: (origins: string[]) => void): this;

on(event: 'connect', listener: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

on(event: 'stream', listener: (stream: ClientHttp2Stream, headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'altsvc', listener: (alt: string, origin: string, stream: number) => void): this;

once(event: 'origin', listener: (origins: string[]) => void): this;

once(event: 'connect', listener: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

once(event: 'stream', listener: (stream: ClientHttp2Stream, headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'altsvc', listener: (alt: string, origin: string, stream: number) => void): this;

prependListener(event: 'origin', listener: (origins: string[]) => void): this;

prependListener(event: 'connect', listener: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

prependListener(event: 'stream', listener: (stream: ClientHttp2Stream, headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'altsvc', listener: (alt: string, origin: string, stream: number) => void): this;

prependOnceListener(event: 'origin', listener: (origins: string[]) => void): this;

prependOnceListener(event: 'connect', listener: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

prependOnceListener(event: 'stream', listener: (stream: ClientHttp2Stream, headers: IncomingHttpHeaders & IncomingHttpStatusHeader, flags: number) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

export interface AlternativeServiceOptions {

origin: number | string | url.URL;

}

export interface ServerHttp2Session extends Http2Session {

readonly server: Http2Server | Http2SecureServer;

/\*\*

\* Submits an `ALTSVC` frame (as defined by [RFC 7838](https://tools.ietf.org/html/rfc7838)) to the connected client.

\*

\* ```js

\* const http2 = require('http2');

\*

\* const server = http2.createServer();

\* server.on('session', (session) => {

\* // Set altsvc for origin https://example.org:80

\* session.altsvc('h2=":8000"', 'https://example.org:80');

\* });

\*

\* server.on('stream', (stream) => {

\* // Set altsvc for a specific stream

\* stream.session.altsvc('h2=":8000"', stream.id);

\* });

\* ```

\*

\* Sending an `ALTSVC` frame with a specific stream ID indicates that the alternate

\* service is associated with the origin of the given `Http2Stream`.

\*

\* The `alt` and origin string \_must\_ contain only ASCII bytes and are

\* strictly interpreted as a sequence of ASCII bytes. The special value `'clear'`may be passed to clear any previously set alternative service for a given

\* domain.

\*

\* When a string is passed for the `originOrStream` argument, it will be parsed as

\* a URL and the origin will be derived. For instance, the origin for the

\* HTTP URL `'https://example.org/foo/bar'` is the ASCII string`'https://example.org'`. An error will be thrown if either the given string

\* cannot be parsed as a URL or if a valid origin cannot be derived.

\*

\* A `URL` object, or any object with an `origin` property, may be passed as`originOrStream`, in which case the value of the `origin` property will be

\* used. The value of the `origin` property \_must\_ be a properly serialized

\* ASCII origin.

\* @since v9.4.0

\* @param alt A description of the alternative service configuration as defined by `RFC 7838`.

\* @param originOrStream Either a URL string specifying the origin (or an `Object` with an `origin` property) or the numeric identifier of an active `Http2Stream` as given by the

\* `http2stream.id` property.

\*/

altsvc(alt: string, originOrStream: number | string | url.URL | AlternativeServiceOptions): void;

/\*\*

\* Submits an `ORIGIN` frame (as defined by [RFC 8336](https://tools.ietf.org/html/rfc8336)) to the connected client

\* to advertise the set of origins for which the server is capable of providing

\* authoritative responses.

\*

\* ```js

\* const http2 = require('http2');

\* const options = getSecureOptionsSomehow();

\* const server = http2.createSecureServer(options);

\* server.on('stream', (stream) => {

\* stream.respond();

\* stream.end('ok');

\* });

\* server.on('session', (session) => {

\* session.origin('https://example.com', 'https://example.org');

\* });

\* ```

\*

\* When a string is passed as an `origin`, it will be parsed as a URL and the

\* origin will be derived. For instance, the origin for the HTTP URL`'https://example.org/foo/bar'` is the ASCII string`'https://example.org'`. An error will be thrown if either the given

\* string

\* cannot be parsed as a URL or if a valid origin cannot be derived.

\*

\* A `URL` object, or any object with an `origin` property, may be passed as

\* an `origin`, in which case the value of the `origin` property will be

\* used. The value of the `origin` property \_must\_ be a properly serialized

\* ASCII origin.

\*

\* Alternatively, the `origins` option may be used when creating a new HTTP/2

\* server using the `http2.createSecureServer()` method:

\*

\* ```js

\* const http2 = require('http2');

\* const options = getSecureOptionsSomehow();

\* options.origins = ['https://example.com', 'https://example.org'];

\* const server = http2.createSecureServer(options);

\* server.on('stream', (stream) => {

\* stream.respond();

\* stream.end('ok');

\* });

\* ```

\* @since v10.12.0

\* @param origins One or more URL Strings passed as separate arguments.

\*/

origin(

...origins: Array<

| string

| url.URL

| {

origin: string;

}

>

): void;

addListener(event: 'connect', listener: (session: ServerHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

addListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'connect', session: ServerHttp2Session, socket: net.Socket | tls.TLSSocket): boolean;

emit(event: 'stream', stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'connect', listener: (session: ServerHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

on(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'connect', listener: (session: ServerHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

once(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'connect', listener: (session: ServerHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

prependListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'connect', listener: (session: ServerHttp2Session, socket: net.Socket | tls.TLSSocket) => void): this;

prependOnceListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

// Http2Server

export interface SessionOptions {

maxDeflateDynamicTableSize?: number | undefined;

maxSessionMemory?: number | undefined;

maxHeaderListPairs?: number | undefined;

maxOutstandingPings?: number | undefined;

maxSendHeaderBlockLength?: number | undefined;

paddingStrategy?: number | undefined;

peerMaxConcurrentStreams?: number | undefined;

settings?: Settings | undefined;

/\*\*

\* Specifies a timeout in milliseconds that

\* a server should wait when an [`'unknownProtocol'`][] is emitted. If the

\* socket has not been destroyed by that time the server will destroy it.

\* @default 100000

\*/

unknownProtocolTimeout?: number | undefined;

selectPadding?(frameLen: number, maxFrameLen: number): number;

createConnection?(authority: url.URL, option: SessionOptions): stream.Duplex;

}

export interface ClientSessionOptions extends SessionOptions {

maxReservedRemoteStreams?: number | undefined;

createConnection?: ((authority: url.URL, option: SessionOptions) => stream.Duplex) | undefined;

protocol?: 'http:' | 'https:' | undefined;

}

export interface ServerSessionOptions extends SessionOptions {

Http1IncomingMessage?: typeof IncomingMessage | undefined;

Http1ServerResponse?: typeof ServerResponse | undefined;

Http2ServerRequest?: typeof Http2ServerRequest | undefined;

Http2ServerResponse?: typeof Http2ServerResponse | undefined;

}

export interface SecureClientSessionOptions extends ClientSessionOptions, tls.ConnectionOptions {}

export interface SecureServerSessionOptions extends ServerSessionOptions, tls.TlsOptions {}

export interface ServerOptions extends ServerSessionOptions {}

export interface SecureServerOptions extends SecureServerSessionOptions {

allowHTTP1?: boolean | undefined;

origins?: string[] | undefined;

}

interface HTTP2ServerCommon {

setTimeout(msec?: number, callback?: () => void): this;

/\*\*

\* Throws ERR\_HTTP2\_INVALID\_SETTING\_VALUE for invalid settings values.

\* Throws ERR\_INVALID\_ARG\_TYPE for invalid settings argument.

\*/

updateSettings(settings: Settings): void;

}

export interface Http2Server extends net.Server, HTTP2ServerCommon {

addListener(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

addListener(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

addListener(event: 'session', listener: (session: ServerHttp2Session) => void): this;

addListener(event: 'sessionError', listener: (err: Error) => void): this;

addListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

addListener(event: 'timeout', listener: () => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'checkContinue', request: Http2ServerRequest, response: Http2ServerResponse): boolean;

emit(event: 'request', request: Http2ServerRequest, response: Http2ServerResponse): boolean;

emit(event: 'session', session: ServerHttp2Session): boolean;

emit(event: 'sessionError', err: Error): boolean;

emit(event: 'stream', stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number): boolean;

emit(event: 'timeout'): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

on(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

on(event: 'session', listener: (session: ServerHttp2Session) => void): this;

on(event: 'sessionError', listener: (err: Error) => void): this;

on(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

on(event: 'timeout', listener: () => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

once(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

once(event: 'session', listener: (session: ServerHttp2Session) => void): this;

once(event: 'sessionError', listener: (err: Error) => void): this;

once(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

once(event: 'timeout', listener: () => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependListener(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependListener(event: 'session', listener: (session: ServerHttp2Session) => void): this;

prependListener(event: 'sessionError', listener: (err: Error) => void): this;

prependListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

prependListener(event: 'timeout', listener: () => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependOnceListener(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependOnceListener(event: 'session', listener: (session: ServerHttp2Session) => void): this;

prependOnceListener(event: 'sessionError', listener: (err: Error) => void): this;

prependOnceListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

prependOnceListener(event: 'timeout', listener: () => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

export interface Http2SecureServer extends tls.Server, HTTP2ServerCommon {

addListener(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

addListener(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

addListener(event: 'session', listener: (session: ServerHttp2Session) => void): this;

addListener(event: 'sessionError', listener: (err: Error) => void): this;

addListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

addListener(event: 'timeout', listener: () => void): this;

addListener(event: 'unknownProtocol', listener: (socket: tls.TLSSocket) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'checkContinue', request: Http2ServerRequest, response: Http2ServerResponse): boolean;

emit(event: 'request', request: Http2ServerRequest, response: Http2ServerResponse): boolean;

emit(event: 'session', session: ServerHttp2Session): boolean;

emit(event: 'sessionError', err: Error): boolean;

emit(event: 'stream', stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number): boolean;

emit(event: 'timeout'): boolean;

emit(event: 'unknownProtocol', socket: tls.TLSSocket): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

on(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

on(event: 'session', listener: (session: ServerHttp2Session) => void): this;

on(event: 'sessionError', listener: (err: Error) => void): this;

on(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

on(event: 'timeout', listener: () => void): this;

on(event: 'unknownProtocol', listener: (socket: tls.TLSSocket) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

once(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

once(event: 'session', listener: (session: ServerHttp2Session) => void): this;

once(event: 'sessionError', listener: (err: Error) => void): this;

once(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

once(event: 'timeout', listener: () => void): this;

once(event: 'unknownProtocol', listener: (socket: tls.TLSSocket) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependListener(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependListener(event: 'session', listener: (session: ServerHttp2Session) => void): this;

prependListener(event: 'sessionError', listener: (err: Error) => void): this;

prependListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

prependListener(event: 'timeout', listener: () => void): this;

prependListener(event: 'unknownProtocol', listener: (socket: tls.TLSSocket) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'checkContinue', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependOnceListener(event: 'request', listener: (request: Http2ServerRequest, response: Http2ServerResponse) => void): this;

prependOnceListener(event: 'session', listener: (session: ServerHttp2Session) => void): this;

prependOnceListener(event: 'sessionError', listener: (err: Error) => void): this;

prependOnceListener(event: 'stream', listener: (stream: ServerHttp2Stream, headers: IncomingHttpHeaders, flags: number) => void): this;

prependOnceListener(event: 'timeout', listener: () => void): this;

prependOnceListener(event: 'unknownProtocol', listener: (socket: tls.TLSSocket) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

/\*\*

\* A `Http2ServerRequest` object is created by {@link Server} or {@link SecureServer} and passed as the first argument to the `'request'` event. It may be used to access a request status,

\* headers, and

\* data.

\* @since v8.4.0

\*/

export class Http2ServerRequest extends stream.Readable {

constructor(stream: ServerHttp2Stream, headers: IncomingHttpHeaders, options: stream.ReadableOptions, rawHeaders: ReadonlyArray<string>);

/\*\*

\* The `request.aborted` property will be `true` if the request has

\* been aborted.

\* @since v10.1.0

\*/

readonly aborted: boolean;

/\*\*

\* The request authority pseudo header field. Because HTTP/2 allows requests

\* to set either `:authority` or `host`, this value is derived from`req.headers[':authority']` if present. Otherwise, it is derived from`req.headers['host']`.

\* @since v8.4.0

\*/

readonly authority: string;

/\*\*

\* See `request.socket`.

\* @since v8.4.0

\* @deprecated Since v13.0.0 - Use `socket`.

\*/

readonly connection: net.Socket | tls.TLSSocket;

/\*\*

\* The `request.complete` property will be `true` if the request has

\* been completed, aborted, or destroyed.

\* @since v12.10.0

\*/

readonly complete: boolean;

/\*\*

\* The request/response headers object.

\*

\* Key-value pairs of header names and values. Header names are lower-cased.

\*

\* ```js

\* // Prints something like:

\* //

\* // { 'user-agent': 'curl/7.22.0',

\* // host: '127.0.0.1:8000',

\* // accept: '\*' }

\* console.log(request.headers);

\* ```

\*

\* See `HTTP/2 Headers Object`.

\*

\* In HTTP/2, the request path, host name, protocol, and method are represented as

\* special headers prefixed with the `:` character (e.g. `':path'`). These special

\* headers will be included in the `request.headers` object. Care must be taken not

\* to inadvertently modify these special headers or errors may occur. For instance,

\* removing all headers from the request will cause errors to occur:

\*

\* ```js

\* removeAllHeaders(request.headers);

\* assert(request.url); // Fails because the :path header has been removed

\* ```

\* @since v8.4.0

\*/

readonly headers: IncomingHttpHeaders;

/\*\*

\* In case of server request, the HTTP version sent by the client. In the case of

\* client response, the HTTP version of the connected-to server. Returns`'2.0'`.

\*

\* Also `message.httpVersionMajor` is the first integer and`message.httpVersionMinor` is the second.

\* @since v8.4.0

\*/

readonly httpVersion: string;

readonly httpVersionMinor: number;

readonly httpVersionMajor: number;

/\*\*

\* The request method as a string. Read-only. Examples: `'GET'`, `'DELETE'`.

\* @since v8.4.0

\*/

readonly method: string;

/\*\*

\* The raw request/response headers list exactly as they were received.

\*

\* The keys and values are in the same list. It is \_not\_ a

\* list of tuples. So, the even-numbered offsets are key values, and the

\* odd-numbered offsets are the associated values.

\*

\* Header names are not lowercased, and duplicates are not merged.

\*

\* ```js

\* // Prints something like:

\* //

\* // [ 'user-agent',

\* // 'this is invalid because there can be only one',

\* // 'User-Agent',

\* // 'curl/7.22.0',

\* // 'Host',

\* // '127.0.0.1:8000',

\* // 'ACCEPT',

\* // '\*' ]

\* console.log(request.rawHeaders);

\* ```

\* @since v8.4.0

\*/

readonly rawHeaders: string[];

/\*\*

\* The raw request/response trailer keys and values exactly as they were

\* received. Only populated at the `'end'` event.

\* @since v8.4.0

\*/

readonly rawTrailers: string[];

/\*\*

\* The request scheme pseudo header field indicating the scheme

\* portion of the target URL.

\* @since v8.4.0

\*/

readonly scheme: string;

/\*\*

\* Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but

\* applies getters, setters, and methods based on HTTP/2 logic.

\*

\* `destroyed`, `readable`, and `writable` properties will be retrieved from and

\* set on `request.stream`.

\*

\* `destroy`, `emit`, `end`, `on` and `once` methods will be called on`request.stream`.

\*

\* `setTimeout` method will be called on `request.stream.session`.

\*

\* `pause`, `read`, `resume`, and `write` will throw an error with code`ERR\_HTTP2\_NO\_SOCKET\_MANIPULATION`. See `Http2Session and Sockets` for

\* more information.

\*

\* All other interactions will be routed directly to the socket. With TLS support,

\* use `request.socket.getPeerCertificate()` to obtain the client's

\* authentication details.

\* @since v8.4.0

\*/

readonly socket: net.Socket | tls.TLSSocket;

/\*\*

\* The `Http2Stream` object backing the request.

\* @since v8.4.0

\*/

readonly stream: ServerHttp2Stream;

/\*\*

\* The request/response trailers object. Only populated at the `'end'` event.

\* @since v8.4.0

\*/

readonly trailers: IncomingHttpHeaders;

/\*\*

\* Request URL string. This contains only the URL that is present in the actual

\* HTTP request. If the request is:

\*

\* ```http

\* GET /status?name=ryan HTTP/1.1

\* Accept: text/plain

\* ```

\*

\* Then `request.url` will be:

\*

\* ```js

\* '/status?name=ryan'

\* ```

\*

\* To parse the url into its parts, `new URL()` can be used:

\*

\* ```console

\* $ node

\* > new URL('/status?name=ryan', 'http://example.com')

\* URL {

\* href: 'http://example.com/status?name=ryan',

\* origin: 'http://example.com',

\* protocol: 'http:',

\* username: '',

\* password: '',

\* host: 'example.com',

\* hostname: 'example.com',

\* port: '',

\* pathname: '/status',

\* search: '?name=ryan',

\* searchParams: URLSearchParams { 'name' => 'ryan' },

\* hash: ''

\* }

\* ```

\* @since v8.4.0

\*/

url: string;

/\*\*

\* Sets the `Http2Stream`'s timeout value to `msecs`. If a callback is

\* provided, then it is added as a listener on the `'timeout'` event on

\* the response object.

\*

\* If no `'timeout'` listener is added to the request, the response, or

\* the server, then `Http2Stream` s are destroyed when they time out. If a

\* handler is assigned to the request, the response, or the server's `'timeout'`events, timed out sockets must be handled explicitly.

\* @since v8.4.0

\*/

setTimeout(msecs: number, callback?: () => void): void;

read(size?: number): Buffer | string | null;

addListener(event: 'aborted', listener: (hadError: boolean, code: number) => void): this;

addListener(event: 'close', listener: () => void): this;

addListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

addListener(event: 'end', listener: () => void): this;

addListener(event: 'readable', listener: () => void): this;

addListener(event: 'error', listener: (err: Error) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'aborted', hadError: boolean, code: number): boolean;

emit(event: 'close'): boolean;

emit(event: 'data', chunk: Buffer | string): boolean;

emit(event: 'end'): boolean;

emit(event: 'readable'): boolean;

emit(event: 'error', err: Error): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'aborted', listener: (hadError: boolean, code: number) => void): this;

on(event: 'close', listener: () => void): this;

on(event: 'data', listener: (chunk: Buffer | string) => void): this;

on(event: 'end', listener: () => void): this;

on(event: 'readable', listener: () => void): this;

on(event: 'error', listener: (err: Error) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'aborted', listener: (hadError: boolean, code: number) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'data', listener: (chunk: Buffer | string) => void): this;

once(event: 'end', listener: () => void): this;

once(event: 'readable', listener: () => void): this;

once(event: 'error', listener: (err: Error) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'aborted', listener: (hadError: boolean, code: number) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

prependListener(event: 'end', listener: () => void): this;

prependListener(event: 'readable', listener: () => void): this;

prependListener(event: 'error', listener: (err: Error) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'aborted', listener: (hadError: boolean, code: number) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'data', listener: (chunk: Buffer | string) => void): this;

prependOnceListener(event: 'end', listener: () => void): this;

prependOnceListener(event: 'readable', listener: () => void): this;

prependOnceListener(event: 'error', listener: (err: Error) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

/\*\*

\* This object is created internally by an HTTP server, not by the user. It is

\* passed as the second parameter to the `'request'` event.

\* @since v8.4.0

\*/

export class Http2ServerResponse extends stream.Writable {

constructor(stream: ServerHttp2Stream);

/\*\*

\* See `response.socket`.

\* @since v8.4.0

\* @deprecated Since v13.0.0 - Use `socket`.

\*/

readonly connection: net.Socket | tls.TLSSocket;

/\*\*

\* Boolean value that indicates whether the response has completed. Starts

\* as `false`. After `response.end()` executes, the value will be `true`.

\* @since v8.4.0

\* @deprecated Since v13.4.0,v12.16.0 - Use `writableEnded`.

\*/

readonly finished: boolean;

/\*\*

\* True if headers were sent, false otherwise (read-only).

\* @since v8.4.0

\*/

readonly headersSent: boolean;

/\*\*

\* A reference to the original HTTP2 request object.

\* @since v15.7.0

\*/

readonly req: Http2ServerRequest;

/\*\*

\* Returns a `Proxy` object that acts as a `net.Socket` (or `tls.TLSSocket`) but

\* applies getters, setters, and methods based on HTTP/2 logic.

\*

\* `destroyed`, `readable`, and `writable` properties will be retrieved from and

\* set on `response.stream`.

\*

\* `destroy`, `emit`, `end`, `on` and `once` methods will be called on`response.stream`.

\*

\* `setTimeout` method will be called on `response.stream.session`.

\*

\* `pause`, `read`, `resume`, and `write` will throw an error with code`ERR\_HTTP2\_NO\_SOCKET\_MANIPULATION`. See `Http2Session and Sockets` for

\* more information.

\*

\* All other interactions will be routed directly to the socket.

\*

\* ```js

\* const http2 = require('http2');

\* const server = http2.createServer((req, res) => {

\* const ip = req.socket.remoteAddress;

\* const port = req.socket.remotePort;

\* res.end(`Your IP address is ${ip} and your source port is ${port}.`);

\* }).listen(3000);

\* ```

\* @since v8.4.0

\*/

readonly socket: net.Socket | tls.TLSSocket;

/\*\*

\* The `Http2Stream` object backing the response.

\* @since v8.4.0

\*/

readonly stream: ServerHttp2Stream;

/\*\*

\* When true, the Date header will be automatically generated and sent in

\* the response if it is not already present in the headers. Defaults to true.

\*

\* This should only be disabled for testing; HTTP requires the Date header

\* in responses.

\* @since v8.4.0

\*/

sendDate: boolean;

/\*\*

\* When using implicit headers (not calling `response.writeHead()` explicitly),

\* this property controls the status code that will be sent to the client when

\* the headers get flushed.

\*

\* ```js

\* response.statusCode = 404;

\* ```

\*

\* After response header was sent to the client, this property indicates the

\* status code which was sent out.

\* @since v8.4.0

\*/

statusCode: number;

/\*\*

\* Status message is not supported by HTTP/2 (RFC 7540 8.1.2.4). It returns

\* an empty string.

\* @since v8.4.0

\*/

statusMessage: '';

/\*\*

\* This method adds HTTP trailing headers (a header but at the end of the

\* message) to the response.

\*

\* Attempting to set a header field name or value that contains invalid characters

\* will result in a `TypeError` being thrown.

\* @since v8.4.0

\*/

addTrailers(trailers: OutgoingHttpHeaders): void;

/\*\*

\* This method signals to the server that all of the response headers and body

\* have been sent; that server should consider this message complete.

\* The method, `response.end()`, MUST be called on each response.

\*

\* If `data` is specified, it is equivalent to calling `response.write(data, encoding)` followed by `response.end(callback)`.

\*

\* If `callback` is specified, it will be called when the response stream

\* is finished.

\* @since v8.4.0

\*/

end(callback?: () => void): void;

end(data: string | Uint8Array, callback?: () => void): void;

end(data: string | Uint8Array, encoding: BufferEncoding, callback?: () => void): void;

/\*\*

\* Reads out a header that has already been queued but not sent to the client.

\* The name is case-insensitive.

\*

\* ```js

\* const contentType = response.getHeader('content-type');

\* ```

\* @since v8.4.0

\*/

getHeader(name: string): string;

/\*\*

\* Returns an array containing the unique names of the current outgoing headers.

\* All header names are lowercase.

\*

\* ```js

\* response.setHeader('Foo', 'bar');

\* response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

\*

\* const headerNames = response.getHeaderNames();

\* // headerNames === ['foo', 'set-cookie']

\* ```

\* @since v8.4.0

\*/

getHeaderNames(): string[];

/\*\*

\* Returns a shallow copy of the current outgoing headers. Since a shallow copy

\* is used, array values may be mutated without additional calls to various

\* header-related http module methods. The keys of the returned object are the

\* header names and the values are the respective header values. All header names

\* are lowercase.

\*

\* The object returned by the `response.getHeaders()` method \_does not\_prototypically inherit from the JavaScript `Object`. This means that typical`Object` methods such as `obj.toString()`,

\* `obj.hasOwnProperty()`, and others

\* are not defined and \_will not work\_.

\*

\* ```js

\* response.setHeader('Foo', 'bar');

\* response.setHeader('Set-Cookie', ['foo=bar', 'bar=baz']);

\*

\* const headers = response.getHeaders();

\* // headers === { foo: 'bar', 'set-cookie': ['foo=bar', 'bar=baz'] }

\* ```

\* @since v8.4.0

\*/

getHeaders(): OutgoingHttpHeaders;

/\*\*

\* Returns `true` if the header identified by `name` is currently set in the

\* outgoing headers. The header name matching is case-insensitive.

\*

\* ```js

\* const hasContentType = response.hasHeader('content-type');

\* ```

\* @since v8.4.0

\*/

hasHeader(name: string): boolean;

/\*\*

\* Removes a header that has been queued for implicit sending.

\*

\* ```js

\* response.removeHeader('Content-Encoding');

\* ```

\* @since v8.4.0

\*/

removeHeader(name: string): void;

/\*\*

\* Sets a single header value for implicit headers. If this header already exists

\* in the to-be-sent headers, its value will be replaced. Use an array of strings

\* here to send multiple headers with the same name.

\*

\* ```js

\* response.setHeader('Content-Type', 'text/html; charset=utf-8');

\* ```

\*

\* or

\*

\* ```js

\* response.setHeader('Set-Cookie', ['type=ninja', 'language=javascript']);

\* ```

\*

\* Attempting to set a header field name or value that contains invalid characters

\* will result in a `TypeError` being thrown.

\*

\* When headers have been set with `response.setHeader()`, they will be merged

\* with any headers passed to `response.writeHead()`, with the headers passed

\* to `response.writeHead()` given precedence.

\*

\* ```js

\* // Returns content-type = text/plain

\* const server = http2.createServer((req, res) => {

\* res.setHeader('Content-Type', 'text/html; charset=utf-8');

\* res.setHeader('X-Foo', 'bar');

\* res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });

\* res.end('ok');

\* });

\* ```

\* @since v8.4.0

\*/

setHeader(name: string, value: number | string | ReadonlyArray<string>): void;

/\*\*

\* Sets the `Http2Stream`'s timeout value to `msecs`. If a callback is

\* provided, then it is added as a listener on the `'timeout'` event on

\* the response object.

\*

\* If no `'timeout'` listener is added to the request, the response, or

\* the server, then `Http2Stream` s are destroyed when they time out. If a

\* handler is assigned to the request, the response, or the server's `'timeout'`events, timed out sockets must be handled explicitly.

\* @since v8.4.0

\*/

setTimeout(msecs: number, callback?: () => void): void;

/\*\*

\* If this method is called and `response.writeHead()` has not been called,

\* it will switch to implicit header mode and flush the implicit headers.

\*

\* This sends a chunk of the response body. This method may

\* be called multiple times to provide successive parts of the body.

\*

\* In the `http` module, the response body is omitted when the

\* request is a HEAD request. Similarly, the `204` and `304` responses\_must not\_ include a message body.

\*

\* `chunk` can be a string or a buffer. If `chunk` is a string,

\* the second parameter specifies how to encode it into a byte stream.

\* By default the `encoding` is `'utf8'`. `callback` will be called when this chunk

\* of data is flushed.

\*

\* This is the raw HTTP body and has nothing to do with higher-level multi-part

\* body encodings that may be used.

\*

\* The first time `response.write()` is called, it will send the buffered

\* header information and the first chunk of the body to the client. The second

\* time `response.write()` is called, Node.js assumes data will be streamed,

\* and sends the new data separately. That is, the response is buffered up to the

\* first chunk of the body.

\*

\* Returns `true` if the entire data was flushed successfully to the kernel

\* buffer. Returns `false` if all or part of the data was queued in user memory.`'drain'` will be emitted when the buffer is free again.

\* @since v8.4.0

\*/

write(chunk: string | Uint8Array, callback?: (err: Error) => void): boolean;

write(chunk: string | Uint8Array, encoding: BufferEncoding, callback?: (err: Error) => void): boolean;

/\*\*

\* Sends a status `100 Continue` to the client, indicating that the request body

\* should be sent. See the `'checkContinue'` event on `Http2Server` and`Http2SecureServer`.

\* @since v8.4.0

\*/

writeContinue(): void;

/\*\*

\* Sends a response header to the request. The status code is a 3-digit HTTP

\* status code, like `404`. The last argument, `headers`, are the response headers.

\*

\* Returns a reference to the `Http2ServerResponse`, so that calls can be chained.

\*

\* For compatibility with `HTTP/1`, a human-readable `statusMessage` may be

\* passed as the second argument. However, because the `statusMessage` has no

\* meaning within HTTP/2, the argument will have no effect and a process warning

\* will be emitted.

\*

\* ```js

\* const body = 'hello world';

\* response.writeHead(200, {

\* 'Content-Length': Buffer.byteLength(body),

\* 'Content-Type': 'text/plain; charset=utf-8',

\* });

\* ```

\*

\* `Content-Length` is given in bytes not characters. The`Buffer.byteLength()` API may be used to determine the number of bytes in a

\* given encoding. On outbound messages, Node.js does not check if Content-Length

\* and the length of the body being transmitted are equal or not. However, when

\* receiving messages, Node.js will automatically reject messages when the`Content-Length` does not match the actual payload size.

\*

\* This method may be called at most one time on a message before `response.end()` is called.

\*

\* If `response.write()` or `response.end()` are called before calling

\* this, the implicit/mutable headers will be calculated and call this function.

\*

\* When headers have been set with `response.setHeader()`, they will be merged

\* with any headers passed to `response.writeHead()`, with the headers passed

\* to `response.writeHead()` given precedence.

\*

\* ```js

\* // Returns content-type = text/plain

\* const server = http2.createServer((req, res) => {

\* res.setHeader('Content-Type', 'text/html; charset=utf-8');

\* res.setHeader('X-Foo', 'bar');

\* res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });

\* res.end('ok');

\* });

\* ```

\*

\* Attempting to set a header field name or value that contains invalid characters

\* will result in a `TypeError` being thrown.

\* @since v8.4.0

\*/

writeHead(statusCode: number, headers?: OutgoingHttpHeaders): this;

writeHead(statusCode: number, statusMessage: string, headers?: OutgoingHttpHeaders): this;

/\*\*

\* Call `http2stream.pushStream()` with the given headers, and wrap the

\* given `Http2Stream` on a newly created `Http2ServerResponse` as the callback

\* parameter if successful. When `Http2ServerRequest` is closed, the callback is

\* called with an error `ERR\_HTTP2\_INVALID\_STREAM`.

\* @since v8.4.0

\* @param headers An object describing the headers

\* @param callback Called once `http2stream.pushStream()` is finished, or either when the attempt to create the pushed `Http2Stream` has failed or has been rejected, or the state of

\* `Http2ServerRequest` is closed prior to calling the `http2stream.pushStream()` method

\*/

createPushResponse(headers: OutgoingHttpHeaders, callback: (err: Error | null, res: Http2ServerResponse) => void): void;

addListener(event: 'close', listener: () => void): this;

addListener(event: 'drain', listener: () => void): this;

addListener(event: 'error', listener: (error: Error) => void): this;

addListener(event: 'finish', listener: () => void): this;

addListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

addListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

addListener(event: string | symbol, listener: (...args: any[]) => void): this;

emit(event: 'close'): boolean;

emit(event: 'drain'): boolean;

emit(event: 'error', error: Error): boolean;

emit(event: 'finish'): boolean;

emit(event: 'pipe', src: stream.Readable): boolean;

emit(event: 'unpipe', src: stream.Readable): boolean;

emit(event: string | symbol, ...args: any[]): boolean;

on(event: 'close', listener: () => void): this;

on(event: 'drain', listener: () => void): this;

on(event: 'error', listener: (error: Error) => void): this;

on(event: 'finish', listener: () => void): this;

on(event: 'pipe', listener: (src: stream.Readable) => void): this;

on(event: 'unpipe', listener: (src: stream.Readable) => void): this;

on(event: string | symbol, listener: (...args: any[]) => void): this;

once(event: 'close', listener: () => void): this;

once(event: 'drain', listener: () => void): this;

once(event: 'error', listener: (error: Error) => void): this;

once(event: 'finish', listener: () => void): this;

once(event: 'pipe', listener: (src: stream.Readable) => void): this;

once(event: 'unpipe', listener: (src: stream.Readable) => void): this;

once(event: string | symbol, listener: (...args: any[]) => void): this;

prependListener(event: 'close', listener: () => void): this;

prependListener(event: 'drain', listener: () => void): this;

prependListener(event: 'error', listener: (error: Error) => void): this;

prependListener(event: 'finish', listener: () => void): this;

prependListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

prependListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

prependListener(event: string | symbol, listener: (...args: any[]) => void): this;

prependOnceListener(event: 'close', listener: () => void): this;

prependOnceListener(event: 'drain', listener: () => void): this;

prependOnceListener(event: 'error', listener: (error: Error) => void): this;

prependOnceListener(event: 'finish', listener: () => void): this;

prependOnceListener(event: 'pipe', listener: (src: stream.Readable) => void): this;

prependOnceListener(event: 'unpipe', listener: (src: stream.Readable) => void): this;

prependOnceListener(event: string | symbol, listener: (...args: any[]) => void): this;

}

export namespace constants {

const NGHTTP2\_SESSION\_SERVER: number;

const NGHTTP2\_SESSION\_CLIENT: number;

const NGHTTP2\_STREAM\_STATE\_IDLE: number;

const NGHTTP2\_STREAM\_STATE\_OPEN: number;

const NGHTTP2\_STREAM\_STATE\_RESERVED\_LOCAL: number;

const NGHTTP2\_STREAM\_STATE\_RESERVED\_REMOTE: number;

const NGHTTP2\_STREAM\_STATE\_HALF\_CLOSED\_LOCAL: number;

const NGHTTP2\_STREAM\_STATE\_HALF\_CLOSED\_REMOTE: number;

const NGHTTP2\_STREAM\_STATE\_CLOSED: number;

const NGHTTP2\_NO\_ERROR: number;

const NGHTTP2\_PROTOCOL\_ERROR: number;

const NGHTTP2\_INTERNAL\_ERROR: number;

const NGHTTP2\_FLOW\_CONTROL\_ERROR: number;

const NGHTTP2\_SETTINGS\_TIMEOUT: number;

const NGHTTP2\_STREAM\_CLOSED: number;

const NGHTTP2\_FRAME\_SIZE\_ERROR: number;

const NGHTTP2\_REFUSED\_STREAM: number;

const NGHTTP2\_CANCEL: number;

const NGHTTP2\_COMPRESSION\_ERROR: number;

const NGHTTP2\_CONNECT\_ERROR: number;

const NGHTTP2\_ENHANCE\_YOUR\_CALM: number;

const NGHTTP2\_INADEQUATE\_SECURITY: number;

const NGHTTP2\_HTTP\_1\_1\_REQUIRED: number;

const NGHTTP2\_ERR\_FRAME\_SIZE\_ERROR: number;

const NGHTTP2\_FLAG\_NONE: number;

const NGHTTP2\_FLAG\_END\_STREAM: number;

const NGHTTP2\_FLAG\_END\_HEADERS: number;

const NGHTTP2\_FLAG\_ACK: number;

const NGHTTP2\_FLAG\_PADDED: number;

const NGHTTP2\_FLAG\_PRIORITY: number;

const DEFAULT\_SETTINGS\_HEADER\_TABLE\_SIZE: number;

const DEFAULT\_SETTINGS\_ENABLE\_PUSH: number;

const DEFAULT\_SETTINGS\_INITIAL\_WINDOW\_SIZE: number;

const DEFAULT\_SETTINGS\_MAX\_FRAME\_SIZE: number;

const MAX\_MAX\_FRAME\_SIZE: number;

const MIN\_MAX\_FRAME\_SIZE: number;

const MAX\_INITIAL\_WINDOW\_SIZE: number;

const NGHTTP2\_DEFAULT\_WEIGHT: number;

const NGHTTP2\_SETTINGS\_HEADER\_TABLE\_SIZE: number;

const NGHTTP2\_SETTINGS\_ENABLE\_PUSH: number;

const NGHTTP2\_SETTINGS\_MAX\_CONCURRENT\_STREAMS: number;

const NGHTTP2\_SETTINGS\_INITIAL\_WINDOW\_SIZE: number;

const NGHTTP2\_SETTINGS\_MAX\_FRAME\_SIZE: number;

const NGHTTP2\_SETTINGS\_MAX\_HEADER\_LIST\_SIZE: number;

const PADDING\_STRATEGY\_NONE: number;

const PADDING\_STRATEGY\_MAX: number;

const PADDING\_STRATEGY\_CALLBACK: number;

const HTTP2\_HEADER\_STATUS: string;

const HTTP2\_HEADER\_METHOD: string;

const HTTP2\_HEADER\_AUTHORITY: string;

const HTTP2\_HEADER\_SCHEME: string;

const HTTP2\_HEADER\_PATH: string;

const HTTP2\_HEADER\_ACCEPT\_CHARSET: string;

const HTTP2\_HEADER\_ACCEPT\_ENCODING: string;

const HTTP2\_HEADER\_ACCEPT\_LANGUAGE: string;

const HTTP2\_HEADER\_ACCEPT\_RANGES: string;

const HTTP2\_HEADER\_ACCEPT: string;

const HTTP2\_HEADER\_ACCESS\_CONTROL\_ALLOW\_ORIGIN: string;

const HTTP2\_HEADER\_AGE: string;

const HTTP2\_HEADER\_ALLOW: string;

const HTTP2\_HEADER\_AUTHORIZATION: string;

const HTTP2\_HEADER\_CACHE\_CONTROL: string;

const HTTP2\_HEADER\_CONNECTION: string;

const HTTP2\_HEADER\_CONTENT\_DISPOSITION: string;

const HTTP2\_HEADER\_CONTENT\_ENCODING: string;

const HTTP2\_HEADER\_CONTENT\_LANGUAGE: string;

const HTTP2\_HEADER\_CONTENT\_LENGTH: string;

const HTTP2\_HEADER\_CONTENT\_LOCATION: string;

const HTTP2\_HEADER\_CONTENT\_MD5: string;

const HTTP2\_HEADER\_CONTENT\_RANGE: string;

const HTTP2\_HEADER\_CONTENT\_TYPE: string;

const HTTP2\_HEADER\_COOKIE: string;

const HTTP2\_HEADER\_DATE: string;

const HTTP2\_HEADER\_ETAG: string;

const HTTP2\_HEADER\_EXPECT: string;

const HTTP2\_HEADER\_EXPIRES: string;

const HTTP2\_HEADER\_FROM: string;

const HTTP2\_HEADER\_HOST: string;

const HTTP2\_HEADER\_IF\_MATCH: string;

const HTTP2\_HEADER\_IF\_MODIFIED\_SINCE: string;

const HTTP2\_HEADER\_IF\_NONE\_MATCH: string;

const HTTP2\_HEADER\_IF\_RANGE: string;

const HTTP2\_HEADER\_IF\_UNMODIFIED\_SINCE: string;

const HTTP2\_HEADER\_LAST\_MODIFIED: string;

const HTTP2\_HEADER\_LINK: string;

const HTTP2\_HEADER\_LOCATION: string;

const HTTP2\_HEADER\_MAX\_FORWARDS: string;

const HTTP2\_HEADER\_PREFER: string;

const HTTP2\_HEADER\_PROXY\_AUTHENTICATE: string;

const HTTP2\_HEADER\_PROXY\_AUTHORIZATION: string;

const HTTP2\_HEADER\_RANGE: string;

const HTTP2\_HEADER\_REFERER: string;

const HTTP2\_HEADER\_REFRESH: string;

const HTTP2\_HEADER\_RETRY\_AFTER: string;

const HTTP2\_HEADER\_SERVER: string;

const HTTP2\_HEADER\_SET\_COOKIE: string;

const HTTP2\_HEADER\_STRICT\_TRANSPORT\_SECURITY: string;

const HTTP2\_HEADER\_TRANSFER\_ENCODING: string;

const HTTP2\_HEADER\_TE: string;

const HTTP2\_HEADER\_UPGRADE: string;

const HTTP2\_HEADER\_USER\_AGENT: string;

const HTTP2\_HEADER\_VARY: string;

const HTTP2\_HEADER\_VIA: string;

const HTTP2\_HEADER\_WWW\_AUTHENTICATE: string;

const HTTP2\_HEADER\_HTTP2\_SETTINGS: string;

const HTTP2\_HEADER\_KEEP\_ALIVE: string;

const HTTP2\_HEADER\_PROXY\_CONNECTION: string;

const HTTP2\_METHOD\_ACL: string;

const HTTP2\_METHOD\_BASELINE\_CONTROL: string;

const HTTP2\_METHOD\_BIND: string;

const HTTP2\_METHOD\_CHECKIN: string;

const HTTP2\_METHOD\_CHECKOUT: string;

const HTTP2\_METHOD\_CONNECT: string;

const HTTP2\_METHOD\_COPY: string;

const HTTP2\_METHOD\_DELETE: string;

const HTTP2\_METHOD\_GET: string;

const HTTP2\_METHOD\_HEAD: string;

const HTTP2\_METHOD\_LABEL: string;

const HTTP2\_METHOD\_LINK: string;

const HTTP2\_METHOD\_LOCK: string;

const HTTP2\_METHOD\_MERGE: string;

const HTTP2\_METHOD\_MKACTIVITY: string;

const HTTP2\_METHOD\_MKCALENDAR: string;

const HTTP2\_METHOD\_MKCOL: string;

const HTTP2\_METHOD\_MKREDIRECTREF: string;

const HTTP2\_METHOD\_MKWORKSPACE: string;

const HTTP2\_METHOD\_MOVE: string;

const HTTP2\_METHOD\_OPTIONS: string;

const HTTP2\_METHOD\_ORDERPATCH: string;

const HTTP2\_METHOD\_PATCH: string;

const HTTP2\_METHOD\_POST: string;

const HTTP2\_METHOD\_PRI: string;

const HTTP2\_METHOD\_PROPFIND: string;

const HTTP2\_METHOD\_PROPPATCH: string;

const HTTP2\_METHOD\_PUT: string;

const HTTP2\_METHOD\_REBIND: string;

const HTTP2\_METHOD\_REPORT: string;

const HTTP2\_METHOD\_SEARCH: string;

const HTTP2\_METHOD\_TRACE: string;

const HTTP2\_METHOD\_UNBIND: string;

const HTTP2\_METHOD\_UNCHECKOUT: string;

const HTTP2\_METHOD\_UNLINK: string;

const HTTP2\_METHOD\_UNLOCK: string;

const HTTP2\_METHOD\_UPDATE: string;

const HTTP2\_METHOD\_UPDATEREDIRECTREF: string;

const HTTP2\_METHOD\_VERSION\_CONTROL: string;

const HTTP\_STATUS\_CONTINUE: number;

const HTTP\_STATUS\_SWITCHING\_PROTOCOLS: number;

const HTTP\_STATUS\_PROCESSING: number;

const HTTP\_STATUS\_OK: number;

const HTTP\_STATUS\_CREATED: number;

const HTTP\_STATUS\_ACCEPTED: number;

const HTTP\_STATUS\_NON\_AUTHORITATIVE\_INFORMATION: number;

const HTTP\_STATUS\_NO\_CONTENT: number;

const HTTP\_STATUS\_RESET\_CONTENT: number;

const HTTP\_STATUS\_PARTIAL\_CONTENT: number;

const HTTP\_STATUS\_MULTI\_STATUS: number;

const HTTP\_STATUS\_ALREADY\_REPORTED: number;

const HTTP\_STATUS\_IM\_USED: number;

const HTTP\_STATUS\_MULTIPLE\_CHOICES: number;

const HTTP\_STATUS\_MOVED\_PERMANENTLY: number;

const HTTP\_STATUS\_FOUND: number;

const HTTP\_STATUS\_SEE\_OTHER: number;

const HTTP\_STATUS\_NOT\_MODIFIED: number;

const HTTP\_STATUS\_USE\_PROXY: number;

const HTTP\_STATUS\_TEMPORARY\_REDIRECT: number;

const HTTP\_STATUS\_PERMANENT\_REDIRECT: number;

const HTTP\_STATUS\_BAD\_REQUEST: number;

const HTTP\_STATUS\_UNAUTHORIZED: number;

const HTTP\_STATUS\_PAYMENT\_REQUIRED: number;

const HTTP\_STATUS\_FORBIDDEN: number;

const HTTP\_STATUS\_NOT\_FOUND: number;

const HTTP\_STATUS\_METHOD\_NOT\_ALLOWED: number;

const HTTP\_STATUS\_NOT\_ACCEPTABLE: number;

const HTTP\_STATUS\_PROXY\_AUTHENTICATION\_REQUIRED: number;

const HTTP\_STATUS\_REQUEST\_TIMEOUT: number;

const HTTP\_STATUS\_CONFLICT: number;

const HTTP\_STATUS\_GONE: number;

const HTTP\_STATUS\_LENGTH\_REQUIRED: number;

const HTTP\_STATUS\_PRECONDITION\_FAILED: number;

const HTTP\_STATUS\_PAYLOAD\_TOO\_LARGE: number;

const HTTP\_STATUS\_URI\_TOO\_LONG: number;

const HTTP\_STATUS\_UNSUPPORTED\_MEDIA\_TYPE: number;

const HTTP\_STATUS\_RANGE\_NOT\_SATISFIABLE: number;

const HTTP\_STATUS\_EXPECTATION\_FAILED: number;

const HTTP\_STATUS\_TEAPOT: number;

const HTTP\_STATUS\_MISDIRECTED\_REQUEST: number;

const HTTP\_STATUS\_UNPROCESSABLE\_ENTITY: number;

const HTTP\_STATUS\_LOCKED: number;

const HTTP\_STATUS\_FAILED\_DEPENDENCY: number;

const HTTP\_STATUS\_UNORDERED\_COLLECTION: number;

const HTTP\_STATUS\_UPGRADE\_REQUIRED: number;

const HTTP\_STATUS\_PRECONDITION\_REQUIRED: number;

const HTTP\_STATUS\_TOO\_MANY\_REQUESTS: number;

const HTTP\_STATUS\_REQUEST\_HEADER\_FIELDS\_TOO\_LARGE: number;

const HTTP\_STATUS\_UNAVAILABLE\_FOR\_LEGAL\_REASONS: number;

const HTTP\_STATUS\_INTERNAL\_SERVER\_ERROR: number;

const HTTP\_STATUS\_NOT\_IMPLEMENTED: number;

const HTTP\_STATUS\_BAD\_GATEWAY: number;

const HTTP\_STATUS\_SERVICE\_UNAVAILABLE: number;

const HTTP\_STATUS\_GATEWAY\_TIMEOUT: number;

const HTTP\_STATUS\_HTTP\_VERSION\_NOT\_SUPPORTED: number;

const HTTP\_STATUS\_VARIANT\_ALSO\_NEGOTIATES: number;

const HTTP\_STATUS\_INSUFFICIENT\_STORAGE: number;

const HTTP\_STATUS\_LOOP\_DETECTED: number;

const HTTP\_STATUS\_BANDWIDTH\_LIMIT\_EXCEEDED: number;

const HTTP\_STATUS\_NOT\_EXTENDED: number;

const HTTP\_STATUS\_NETWORK\_AUTHENTICATION\_REQUIRED: number;

}

/\*\*

\* This symbol can be set as a property on the HTTP/2 headers object with

\* an array value in order to provide a list of headers considered sensitive.

\*/

export const sensitiveHeaders: symbol;

/\*\*

\* Returns an object containing the default settings for an `Http2Session`instance. This method returns a new object instance every time it is called

\* so instances returned may be safely modified for use.

\* @since v8.4.0

\*/

export function getDefaultSettings(): Settings;

/\*\*

\* Returns a `Buffer` instance containing serialized representation of the given

\* HTTP/2 settings as specified in the [HTTP/2](https://tools.ietf.org/html/rfc7540) specification. This is intended

\* for use with the `HTTP2-Settings` header field.

\*

\* ```js

\* const http2 = require('http2');

\*

\* const packed = http2.getPackedSettings({ enablePush: false });

\*

\* console.log(packed.toString('base64'));

\* // Prints: AAIAAAAA

\* ```

\* @since v8.4.0

\*/

export function getPackedSettings(settings: Settings): Buffer;

/\*\*

\* Returns a `HTTP/2 Settings Object` containing the deserialized settings from

\* the given `Buffer` as generated by `http2.getPackedSettings()`.

\* @since v8.4.0

\* @param buf The packed settings.

\*/

export function getUnpackedSettings(buf: Uint8Array): Settings;

/\*\*

\* Returns a `net.Server` instance that creates and manages `Http2Session`instances.

\*

\* Since there are no browsers known that support [unencrypted HTTP/2](https://http2.github.io/faq/#does-http2-require-encryption), the use of {@link createSecureServer} is necessary when

\* communicating

\* with browser clients.

\*

\* ```js

\* const http2 = require('http2');

\*

\* // Create an unencrypted HTTP/2 server.

\* // Since there are no browsers known that support

\* // unencrypted HTTP/2, the use of `http2.createSecureServer()`

\* // is necessary when communicating with browser clients.

\* const server = http2.createServer();

\*

\* server.on('stream', (stream, headers) => {

\* stream.respond({

\* 'content-type': 'text/html; charset=utf-8',

\* ':status': 200

\* });

\* stream.end('<h1>Hello World</h1>');

\* });

\*

\* server.listen(80);

\* ```

\* @since v8.4.0

\* @param onRequestHandler See `Compatibility API`

\*/

export function createServer(onRequestHandler?: (request: Http2ServerRequest, response: Http2ServerResponse) => void): Http2Server;

export function createServer(options: ServerOptions, onRequestHandler?: (request: Http2ServerRequest, response: Http2ServerResponse) => void): Http2Server;

/\*\*

\* Returns a `tls.Server` instance that creates and manages `Http2Session`instances.

\*

\* ```js

\* const http2 = require('http2');

\* const fs = require('fs');

\*

\* const options = {

\* key: fs.readFileSync('server-key.pem'),

\* cert: fs.readFileSync('server-cert.pem')

\* };

\*

\* // Create a secure HTTP/2 server

\* const server = http2.createSecureServer(options);

\*

\* server.on('stream', (stream, headers) => {

\* stream.respond({

\* 'content-type': 'text/html; charset=utf-8',

\* ':status': 200

\* });

\* stream.end('<h1>Hello World</h1>');

\* });

\*

\* server.listen(80);

\* ```

\* @since v8.4.0

\* @param onRequestHandler See `Compatibility API`

\*/

export function createSecureServer(onRequestHandler?: (request: Http2ServerRequest, response: Http2ServerResponse) => void): Http2SecureServer;

export function createSecureServer(options: SecureServerOptions, onRequestHandler?: (request: Http2ServerRequest, response: Http2ServerResponse) => void): Http2SecureServer;

/\*\*

\* Returns a `ClientHttp2Session` instance.

\*

\* ```js

\* const http2 = require('http2');

\* const client = http2.connect('https://localhost:1234');

\*

\* // Use the client

\*

\* client.close();

\* ```

\* @since v8.4.0

\* @param authority The remote HTTP/2 server to connect to. This must be in the form of a minimal, valid URL with the `http://` or `https://` prefix, host name, and IP port (if a non-default port

\* is used). Userinfo (user ID and password), path, querystring, and fragment details in the URL will be ignored.

\* @param listener Will be registered as a one-time listener of the {@link 'connect'} event.

\*/

export function connect(authority: string | url.URL, listener: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void): ClientHttp2Session;

export function connect(

authority: string | url.URL,

options?: ClientSessionOptions | SecureClientSessionOptions,

listener?: (session: ClientHttp2Session, socket: net.Socket | tls.TLSSocket) => void

): ClientHttp2Session;

}

declare module 'node:http2' {

export \* from 'http2';

}