/\*\*

\* To use the HTTP server and client one must `require('http')`.

\*

\* The HTTP interfaces in Node.js are designed to support many features

\* of the protocol which have been traditionally difficult to use.

\* In particular, large, possibly chunk-encoded, messages. The interface is

\* careful to never buffer entire requests or responses, so the

\* user is able to stream data.

\*

\* HTTP message headers are represented by an object like this:

\*

\* ```js

\* { 'content-length': '123',

\* 'content-type': 'text/plain',

\* 'connection': 'keep-alive',

\* 'host': 'mysite.com',

\* 'accept': '\*' }

\* ```

\*

\* Keys are lowercased. Values are not modified.

\*

\* In order to support the full spectrum of possible HTTP applications, the Node.js

\* HTTP API is very low-level. It deals with stream handling and message

\* parsing only. It parses a message into headers and body but it does not

\* parse the actual headers or the body.

\*

\* See `message.headers` for details on how duplicate headers are handled.

\*

\* The raw headers as they were received are retained in the `rawHeaders`property, which is an array of `[key, value, key2, value2, ...]`. For

\* example, the previous message header object might have a `rawHeaders`list like the following:

\*

\* ```js

\* [ 'ConTent-Length', '123456',

\* 'content-LENGTH', '123',

\* 'content-type', 'text/plain',

\* 'CONNECTION', 'keep-alive',

\* 'Host', 'mysite.com',

\* 'accepT', '\*' ]

\* ```

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

\*/

declare module 'http' {

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

import { URL } from 'node:url';

import { TcpSocketConnectOpts, Socket, Server as NetServer, LookupFunction } from 'node:net';

// incoming headers will never contain number

interface IncomingHttpHeaders extends NodeJS.Dict<string | string[]> {

accept?: string | undefined;

'accept-language'?: string | undefined;

'accept-patch'?: string | undefined;

'accept-ranges'?: string | undefined;

'access-control-allow-credentials'?: string | undefined;

'access-control-allow-headers'?: string | undefined;

'access-control-allow-methods'?: string | undefined;

'access-control-allow-origin'?: string | undefined;

'access-control-expose-headers'?: string | undefined;

'access-control-max-age'?: string | undefined;

'access-control-request-headers'?: string | undefined;

'access-control-request-method'?: string | undefined;

age?: string | undefined;

allow?: string | undefined;

'alt-svc'?: string | undefined;

authorization?: string | undefined;

'cache-control'?: string | undefined;

connection?: string | undefined;

'content-disposition'?: string | undefined;

'content-encoding'?: string | undefined;

'content-language'?: string | undefined;

'content-length'?: string | undefined;

'content-location'?: string | undefined;

'content-range'?: string | undefined;

'content-type'?: string | undefined;

cookie?: string | undefined;

date?: string | undefined;

etag?: string | undefined;

expect?: string | undefined;

expires?: string | undefined;

forwarded?: string | undefined;

from?: string | undefined;

host?: string | undefined;

'if-match'?: string | undefined;

'if-modified-since'?: string | undefined;

'if-none-match'?: string | undefined;

'if-unmodified-since'?: string | undefined;

'last-modified'?: string | undefined;

location?: string | undefined;

origin?: string | undefined;

pragma?: string | undefined;

'proxy-authenticate'?: string | undefined;

'proxy-authorization'?: string | undefined;

'public-key-pins'?: string | undefined;

range?: string | undefined;

referer?: string | undefined;

'retry-after'?: string | undefined;

'sec-websocket-accept'?: string | undefined;

'sec-websocket-extensions'?: string | undefined;

'sec-websocket-key'?: string | undefined;

'sec-websocket-protocol'?: string | undefined;

'sec-websocket-version'?: string | undefined;

'set-cookie'?: string[] | undefined;

'strict-transport-security'?: string | undefined;

tk?: string | undefined;

trailer?: string | undefined;

'transfer-encoding'?: string | undefined;

upgrade?: string | undefined;

'user-agent'?: string | undefined;

vary?: string | undefined;

via?: string | undefined;

warning?: string | undefined;

'www-authenticate'?: string | undefined;

}

// outgoing headers allows numbers (as they are converted internally to strings)

type OutgoingHttpHeader = number | string | string[];

interface OutgoingHttpHeaders extends NodeJS.Dict<OutgoingHttpHeader> {}

interface ClientRequestArgs {

signal?: AbortSignal | undefined;

protocol?: string | null | undefined;

host?: string | null | undefined;

hostname?: string | null | undefined;

family?: number | undefined;

port?: number | string | null | undefined;

defaultPort?: number | string | undefined;

localAddress?: string | undefined;

socketPath?: string | undefined;

/\*\*

\* @default 8192

\*/

maxHeaderSize?: number | undefined;

method?: string | undefined;

path?: string | null | undefined;

headers?: OutgoingHttpHeaders | undefined;

auth?: string | null | undefined;

agent?: Agent | boolean | undefined;

\_defaultAgent?: Agent | undefined;

timeout?: number | undefined;

setHost?: boolean | undefined;

// https://github.com/nodejs/node/blob/master/lib/\_http\_client.js#L278

createConnection?: ((options: ClientRequestArgs, oncreate: (err: Error, socket: Socket) => void) => Socket) | undefined;

lookup?: LookupFunction | undefined;

}

interface ServerOptions {

IncomingMessage?: typeof IncomingMessage | undefined;

ServerResponse?: typeof ServerResponse | undefined;

/\*\*

\* Optionally overrides the value of

\* `--max-http-header-size` for requests received by this server, i.e.

\* the maximum length of request headers in bytes.

\* @default 8192

\*/

maxHeaderSize?: number | undefined;

/\*\*

\* Use an insecure HTTP parser that accepts invalid HTTP headers when true.

\* Using the insecure parser should be avoided.

\* See --insecure-http-parser for more information.

\* @default false

\*/

insecureHTTPParser?: boolean | undefined;

}

type RequestListener = (req: IncomingMessage, res: ServerResponse) => void;

/\*\*

\* @since v0.1.17

\*/

class Server extends NetServer {

constructor(requestListener?: RequestListener);

constructor(options: ServerOptions, requestListener?: RequestListener);

/\*\*

\* Sets the timeout value for sockets, and emits a `'timeout'` event on

\* the Server object, passing the socket as an argument, if a timeout

\* occurs.

\*

\* If there is a `'timeout'` event listener on the Server object, then it

\* will be called with the timed-out socket as an argument.

\*

\* By default, the Server does not timeout sockets. However, if a callback

\* is assigned to the Server's `'timeout'` event, timeouts must be handled

\* explicitly.

\* @since v0.9.12

\* @param [msecs=0 (no timeout)]

\*/

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

setTimeout(callback: () => void): this;

/\*\*

\* Limits maximum incoming headers count. If set to 0, no limit will be applied.

\* @since v0.7.0

\*/

maxHeadersCount: number | null;

/\*\*

\* The maximum number of requests socket can handle

\* before closing keep alive connection.

\*

\* A value of `0` will disable the limit.

\*

\* When the limit is reached it will set the `Connection` header value to `close`,

\* but will not actually close the connection, subsequent requests sent

\* after the limit is reached will get `503 Service Unavailable` as a response.

\* @since v16.10.0

\*/

maxRequestsPerSocket: number | null;

/\*\*

\* The number of milliseconds of inactivity before a socket is presumed

\* to have timed out.

\*

\* A value of `0` will disable the timeout behavior on incoming connections.

\*

\* The socket timeout logic is set up on connection, so changing this

\* value only affects new connections to the server, not any existing connections.

\* @since v0.9.12

\*/

timeout: number;

/\*\*

\* Limit the amount of time the parser will wait to receive the complete HTTP

\* headers.

\*

\* In case of inactivity, the rules defined in `server.timeout` apply. However,

\* that inactivity based timeout would still allow the connection to be kept open

\* if the headers are being sent very slowly (by default, up to a byte per 2

\* minutes). In order to prevent this, whenever header data arrives an additional

\* check is made that more than `server.headersTimeout` milliseconds has not

\* passed since the connection was established. If the check fails, a `'timeout'`event is emitted on the server object, and (by default) the socket is destroyed.

\* See `server.timeout` for more information on how timeout behavior can be

\* customized.

\* @since v11.3.0, v10.14.0

\*/

headersTimeout: number;

/\*\*

\* The number of milliseconds of inactivity a server needs to wait for additional

\* incoming data, after it has finished writing the last response, before a socket

\* will be destroyed. If the server receives new data before the keep-alive

\* timeout has fired, it will reset the regular inactivity timeout, i.e.,`server.timeout`.

\*

\* A value of `0` will disable the keep-alive timeout behavior on incoming

\* connections.

\* A value of `0` makes the http server behave similarly to Node.js versions prior

\* to 8.0.0, which did not have a keep-alive timeout.

\*

\* The socket timeout logic is set up on connection, so changing this value only

\* affects new connections to the server, not any existing connections.

\* @since v8.0.0

\*/

keepAliveTimeout: number;

/\*\*

\* Sets the timeout value in milliseconds for receiving the entire request from

\* the client.

\*

\* If the timeout expires, the server responds with status 408 without

\* forwarding the request to the request listener and then closes the connection.

\*

\* It must be set to a non-zero value (e.g. 120 seconds) to protect against

\* potential Denial-of-Service attacks in case the server is deployed without a

\* reverse proxy in front.

\* @since v14.11.0

\*/

requestTimeout: number;

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

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

addListener(event: 'connection', listener: (socket: Socket) => void): this;

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

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

addListener(event: 'checkContinue', listener: RequestListener): this;

addListener(event: 'checkExpectation', listener: RequestListener): this;

addListener(event: 'clientError', listener: (err: Error, socket: stream.Duplex) => void): this;

addListener(event: 'connect', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

addListener(event: 'request', listener: RequestListener): this;

addListener(event: 'upgrade', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

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

emit(event: 'close'): boolean;

emit(event: 'connection', socket: Socket): boolean;

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

emit(event: 'listening'): boolean;

emit(event: 'checkContinue', req: IncomingMessage, res: ServerResponse): boolean;

emit(event: 'checkExpectation', req: IncomingMessage, res: ServerResponse): boolean;

emit(event: 'clientError', err: Error, socket: stream.Duplex): boolean;

emit(event: 'connect', req: IncomingMessage, socket: stream.Duplex, head: Buffer): boolean;

emit(event: 'request', req: IncomingMessage, res: ServerResponse): boolean;

emit(event: 'upgrade', req: IncomingMessage, socket: stream.Duplex, head: Buffer): boolean;

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

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

on(event: 'connection', listener: (socket: Socket) => void): this;

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

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

on(event: 'checkContinue', listener: RequestListener): this;

on(event: 'checkExpectation', listener: RequestListener): this;

on(event: 'clientError', listener: (err: Error, socket: stream.Duplex) => void): this;

on(event: 'connect', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

on(event: 'request', listener: RequestListener): this;

on(event: 'upgrade', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

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

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

once(event: 'connection', listener: (socket: Socket) => void): this;

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

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

once(event: 'checkContinue', listener: RequestListener): this;

once(event: 'checkExpectation', listener: RequestListener): this;

once(event: 'clientError', listener: (err: Error, socket: stream.Duplex) => void): this;

once(event: 'connect', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

once(event: 'request', listener: RequestListener): this;

once(event: 'upgrade', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

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

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

prependListener(event: 'connection', listener: (socket: Socket) => void): this;

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

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

prependListener(event: 'checkContinue', listener: RequestListener): this;

prependListener(event: 'checkExpectation', listener: RequestListener): this;

prependListener(event: 'clientError', listener: (err: Error, socket: stream.Duplex) => void): this;

prependListener(event: 'connect', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

prependListener(event: 'request', listener: RequestListener): this;

prependListener(event: 'upgrade', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

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

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

prependOnceListener(event: 'connection', listener: (socket: Socket) => void): this;

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

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

prependOnceListener(event: 'checkContinue', listener: RequestListener): this;

prependOnceListener(event: 'checkExpectation', listener: RequestListener): this;

prependOnceListener(event: 'clientError', listener: (err: Error, socket: stream.Duplex) => void): this;

prependOnceListener(event: 'connect', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

prependOnceListener(event: 'request', listener: RequestListener): this;

prependOnceListener(event: 'upgrade', listener: (req: IncomingMessage, socket: stream.Duplex, head: Buffer) => void): this;

}

/\*\*

\* This class serves as the parent class of {@link ClientRequest} and {@link ServerResponse}. It is an abstract of outgoing message from

\* the perspective of the participants of HTTP transaction.

\* @since v0.1.17

\*/

class OutgoingMessage extends stream.Writable {

readonly req: IncomingMessage;

chunkedEncoding: boolean;

shouldKeepAlive: boolean;

useChunkedEncodingByDefault: boolean;

sendDate: boolean;

/\*\*

\* @deprecated Use `writableEnded` instead.

\*/

finished: boolean;

/\*\*

\* Read-only. `true` if the headers were sent, otherwise `false`.

\* @since v0.9.3

\*/

readonly headersSent: boolean;

/\*\*

\* Aliases of `outgoingMessage.socket`

\* @since v0.3.0

\* @deprecated Since v15.12.0,v14.17.1 - Use `socket` instead.

\*/

readonly connection: Socket | null;

/\*\*

\* Reference to the underlying socket. Usually, users will not want to access

\* this property.

\*

\* After calling `outgoingMessage.end()`, this property will be nulled.

\* @since v0.3.0

\*/

readonly socket: Socket | null;

constructor();

/\*\*

\* Once a socket is associated with the message and is connected,`socket.setTimeout()` will be called with `msecs` as the first parameter.

\* @since v0.9.12

\* @param callback Optional function to be called when a timeout occurs. Same as binding to the `timeout` event.

\*/

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

/\*\*

\* Sets a single header value for the header object.

\* @since v0.4.0

\* @param name Header name

\* @param value Header value

\*/

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

/\*\*

\* Gets the value of HTTP header with the given name. If such a name doesn't

\* exist in message, it will be `undefined`.

\* @since v0.4.0

\* @param name Name of header

\*/

getHeader(name: string): number | string | string[] | undefined;

/\*\*

\* 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 `outgoingMessage.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

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

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

\*

\* const headers = outgoingMessage.getHeaders();

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

\* ```

\* @since v8.0.0

\*/

getHeaders(): OutgoingHttpHeaders;

/\*\*

\* Returns an array of names of headers of the outgoing outgoingMessage. All

\* names are lowercase.

\* @since v8.0.0

\*/

getHeaderNames(): string[];

/\*\*

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

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

\*

\* ```js

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

\* ```

\* @since v8.0.0

\*/

hasHeader(name: string): boolean;

/\*\*

\* Removes a header that is queued for implicit sending.

\*

\* ```js

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

\* ```

\* @since v0.4.0

\*/

removeHeader(name: string): void;

/\*\*

\* Adds HTTP trailers (headers but at the end of the message) to the message.

\*

\* Trailers are \*\*only\*\* be emitted if the message is chunked encoded. If not,

\* the trailer will be silently discarded.

\*

\* HTTP requires the `Trailer` header to be sent to emit trailers,

\* with a list of header fields in its value, e.g.

\*

\* ```js

\* message.writeHead(200, { 'Content-Type': 'text/plain',

\* 'Trailer': 'Content-MD5' });

\* message.write(fileData);

\* message.addTrailers({ 'Content-MD5': '7895bf4b8828b55ceaf47747b4bca667' });

\* message.end();

\* ```

\*

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

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

\* @since v0.3.0

\*/

addTrailers(headers: OutgoingHttpHeaders | ReadonlyArray<[string, string]>): void;

/\*\*

\* Compulsorily flushes the message headers

\*

\* For efficiency reason, Node.js normally buffers the message headers

\* until `outgoingMessage.end()` is called or the first chunk of message data

\* is written. It then tries to pack the headers and data into a single TCP

\* packet.

\*

\* It is usually desired (it saves a TCP round-trip), but not when the first

\* data is not sent until possibly much later. `outgoingMessage.flushHeaders()`bypasses the optimization and kickstarts the request.

\* @since v1.6.0

\*/

flushHeaders(): void;

}

/\*\*

\* 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 v0.1.17

\*/

class ServerResponse extends OutgoingMessage {

/\*\*

\* 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 v0.4.0

\*/

statusCode: number;

/\*\*

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

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

\* the headers get flushed. If this is left as `undefined` then the standard

\* message for the status code will be used.

\*

\* ```js

\* response.statusMessage = 'Not found';

\* ```

\*

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

\* status message which was sent out.

\* @since v0.11.8

\*/

statusMessage: string;

constructor(req: IncomingMessage);

assignSocket(socket: Socket): void;

detachSocket(socket: Socket): void;

/\*\*

\* Sends a HTTP/1.1 100 Continue message to the client, indicating that

\* the request body should be sent. See the `'checkContinue'` event on`Server`.

\* @since v0.3.0

\*/

writeContinue(callback?: () => void): 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.

\* Optionally one can give a human-readable `statusMessage` as the second

\* argument.

\*

\* `headers` may be an `Array` where 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. The array is in the same

\* format as `request.rawHeaders`.

\*

\* Returns a reference to the `ServerResponse`, so that calls can be chained.

\*

\* ```js

\* const body = 'hello world';

\* response

\* .writeHead(200, {

\* 'Content-Length': Buffer.byteLength(body),

\* 'Content-Type': 'text/plain'

\* })

\* .end(body);

\* ```

\*

\* This method must only be called once on a message and it must

\* be called 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.

\*

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

\* it will directly write the supplied header values onto the network channel

\* without caching internally, and the `response.getHeader()` on the header

\* will not yield the expected result. If progressive population of headers is

\* desired with potential future retrieval and modification, use `response.setHeader()` instead.

\*

\* ```js

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

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

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

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

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

\* res.end('ok');

\* });

\* ```

\*

\* `Content-Length` is given in bytes, not characters. Use `Buffer.byteLength()` to determine the length of the body in bytes. Node.js

\* does not check whether `Content-Length` and the length of the body which has

\* been transmitted are equal or not.

\*

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

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

\* @since v0.1.30

\*/

writeHead(statusCode: number, statusMessage?: string, headers?: OutgoingHttpHeaders | OutgoingHttpHeader[]): this;

writeHead(statusCode: number, headers?: OutgoingHttpHeaders | OutgoingHttpHeader[]): this;

/\*\*

\* Sends a HTTP/1.1 102 Processing message to the client, indicating that

\* the request body should be sent.

\* @since v10.0.0

\*/

writeProcessing(): void;

}

interface InformationEvent {

statusCode: number;

statusMessage: string;

httpVersion: string;

httpVersionMajor: number;

httpVersionMinor: number;

headers: IncomingHttpHeaders;

rawHeaders: string[];

}

/\*\*

\* This object is created internally and returned from {@link request}. It

\* represents an \_in-progress\_ request whose header has already been queued. The

\* header is still mutable using the `setHeader(name, value)`,`getHeader(name)`, `removeHeader(name)` API. The actual header will

\* be sent along with the first data chunk or when calling `request.end()`.

\*

\* To get the response, add a listener for `'response'` to the request object.`'response'` will be emitted from the request object when the response

\* headers have been received. The `'response'` event is executed with one

\* argument which is an instance of {@link IncomingMessage}.

\*

\* During the `'response'` event, one can add listeners to the

\* response object; particularly to listen for the `'data'` event.

\*

\* If no `'response'` handler is added, then the response will be

\* entirely discarded. However, if a `'response'` event handler is added,

\* then the data from the response object \*\*must\*\* be consumed, either by

\* calling `response.read()` whenever there is a `'readable'` event, or

\* by adding a `'data'` handler, or by calling the `.resume()` method.

\* Until the data is consumed, the `'end'` event will not fire. Also, until

\* the data is read it will consume memory that can eventually lead to a

\* 'process out of memory' error.

\*

\* For backward compatibility, `res` will only emit `'error'` if there is an`'error'` listener registered.

\*

\* Node.js does not check whether Content-Length and the length of the

\* body which has been transmitted are equal or not.

\* @since v0.1.17

\*/

class ClientRequest extends OutgoingMessage {

/\*\*

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

\* been aborted.

\* @since v0.11.14

\* @deprecated Since v17.0.0 - Check `destroyed` instead.

\*/

aborted: boolean;

/\*\*

\* The request host.

\* @since v14.5.0, v12.19.0

\*/

host: string;

/\*\*

\* The request protocol.

\* @since v14.5.0, v12.19.0

\*/

protocol: string;

constructor(url: string | URL | ClientRequestArgs, cb?: (res: IncomingMessage) => void);

/\*\*

\* The request method.

\* @since v0.1.97

\*/

method: string;

/\*\*

\* The request path.

\* @since v0.4.0

\*/

path: string;

/\*\*

\* Marks the request as aborting. Calling this will cause remaining data

\* in the response to be dropped and the socket to be destroyed.

\* @since v0.3.8

\* @deprecated Since v14.1.0,v13.14.0 - Use `destroy` instead.

\*/

abort(): void;

onSocket(socket: Socket): void;

/\*\*

\* Once a socket is assigned to this request and is connected `socket.setTimeout()` will be called.

\* @since v0.5.9

\* @param timeout Milliseconds before a request times out.

\* @param callback Optional function to be called when a timeout occurs. Same as binding to the `'timeout'` event.

\*/

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

/\*\*

\* Once a socket is assigned to this request and is connected `socket.setNoDelay()` will be called.

\* @since v0.5.9

\*/

setNoDelay(noDelay?: boolean): void;

/\*\*

\* Once a socket is assigned to this request and is connected `socket.setKeepAlive()` will be called.

\* @since v0.5.9

\*/

setSocketKeepAlive(enable?: boolean, initialDelay?: number): void;

/\*\*

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

\* headers. Header names are returned with their exact casing being set.

\*

\* ```js

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

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

\*

\* const headerNames = request.getRawHeaderNames();

\* // headerNames === ['Foo', 'Set-Cookie']

\* ```

\* @since v15.13.0, v14.17.0

\*/

getRawHeaderNames(): string[];

/\*\*

\* @deprecated

\*/

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

addListener(event: 'connect', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

addListener(event: 'information', listener: (info: InformationEvent) => void): this;

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

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

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

addListener(event: 'upgrade', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

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

addListener(event: 'error', listener: (err: 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;

/\*\*

\* @deprecated

\*/

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

on(event: 'connect', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

on(event: 'information', listener: (info: InformationEvent) => void): this;

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

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

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

on(event: 'upgrade', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

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

on(event: 'error', listener: (err: 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;

/\*\*

\* @deprecated

\*/

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

once(event: 'connect', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

once(event: 'information', listener: (info: InformationEvent) => void): this;

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

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

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

once(event: 'upgrade', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

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

once(event: 'error', listener: (err: 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;

/\*\*

\* @deprecated

\*/

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

prependListener(event: 'connect', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

prependListener(event: 'information', listener: (info: InformationEvent) => void): this;

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

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

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

prependListener(event: 'upgrade', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

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

prependListener(event: 'error', listener: (err: 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;

/\*\*

\* @deprecated

\*/

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

prependOnceListener(event: 'connect', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

prependOnceListener(event: 'information', listener: (info: InformationEvent) => void): this;

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

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

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

prependOnceListener(event: 'upgrade', listener: (response: IncomingMessage, socket: Socket, head: Buffer) => void): this;

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

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

prependOnceListener(event: 'error', listener: (err: 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;

}

/\*\*

\* An `IncomingMessage` object is created by {@link Server} or {@link ClientRequest} and passed as the first argument to the `'request'` and `'response'` event respectively. It may be used to

\* access response

\* status, headers and data.

\*

\* Different from its `socket` value which is a subclass of `stream.Duplex`, the`IncomingMessage` itself extends `stream.Readable` and is created separately to

\* parse and emit the incoming HTTP headers and payload, as the underlying socket

\* may be reused multiple times in case of keep-alive.

\* @since v0.1.17

\*/

class IncomingMessage extends stream.Readable {

constructor(socket: Socket);

/\*\*

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

\* been aborted.

\* @since v10.1.0

\* @deprecated Since v17.0.0 - Check `message.destroyed` from [stream.Readable](https://nodejs.org/dist/latest-v17.x/docs/api/stream.html#class-streamreadable).

\*/

aborted: boolean;

/\*\*

\* 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.

\* Probably either `'1.1'` or `'1.0'`.

\*

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

\* @since v0.1.1

\*/

httpVersion: string;

httpVersionMajor: number;

httpVersionMinor: number;

/\*\*

\* The `message.complete` property will be `true` if a complete HTTP message has

\* been received and successfully parsed.

\*

\* This property is particularly useful as a means of determining if a client or

\* server fully transmitted a message before a connection was terminated:

\*

\* ```js

\* const req = http.request({

\* host: '127.0.0.1',

\* port: 8080,

\* method: 'POST'

\* }, (res) => {

\* res.resume();

\* res.on('end', () => {

\* if (!res.complete)

\* console.error(

\* 'The connection was terminated while the message was still being sent');

\* });

\* });

\* ```

\* @since v0.3.0

\*/

complete: boolean;

/\*\*

\* Alias for `message.socket`.

\* @since v0.1.90

\* @deprecated Since v16.0.0 - Use `socket`.

\*/

connection: Socket;

/\*\*

\* The `net.Socket` object associated with the connection.

\*

\* With HTTPS support, use `request.socket.getPeerCertificate()` to obtain the

\* client's authentication details.

\*

\* This property is guaranteed to be an instance of the `net.Socket` class,

\* a subclass of `stream.Duplex`, unless the user specified a socket

\* type other than `net.Socket`.

\* @since v0.3.0

\*/

socket: Socket;

/\*\*

\* 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);

\* ```

\*

\* Duplicates in raw headers are handled in the following ways, depending on the

\* header name:

\*

\* \* Duplicates of `age`, `authorization`, `content-length`, `content-type`,`etag`, `expires`, `from`, `host`, `if-modified-since`, `if-unmodified-since`,`last-modified`, `location`,

\* `max-forwards`, `proxy-authorization`, `referer`,`retry-after`, `server`, or `user-agent` are discarded.

\* \* `set-cookie` is always an array. Duplicates are added to the array.

\* \* For duplicate `cookie` headers, the values are joined together with '; '.

\* \* For all other headers, the values are joined together with ', '.

\* @since v0.1.5

\*/

headers: IncomingHttpHeaders;

/\*\*

\* 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 v0.11.6

\*/

rawHeaders: string[];

/\*\*

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

\* @since v0.3.0

\*/

trailers: NodeJS.Dict<string>;

/\*\*

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

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

\* @since v0.11.6

\*/

rawTrailers: string[];

/\*\*

\* Calls `message.socket.setTimeout(msecs, callback)`.

\* @since v0.5.9

\*/

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

/\*\*

\* \*\*Only valid for request obtained from {@link Server}.\*\*

\*

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

\* @since v0.1.1

\*/

method?: string | undefined;

/\*\*

\* \*\*Only valid for request obtained from {@link Server}.\*\*

\*

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

\* HTTP request. Take the following request:

\*

\* ```http

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

\* Accept: text/plain

\* ```

\*

\* To parse the URL into its parts:

\*

\* ```js

\* new URL(request.url, `http://${request.headers.host}`);

\* ```

\*

\* When `request.url` is `'/status?name=ryan'` and`request.headers.host` is `'localhost:3000'`:

\*

\* ```console

\* $ node

\* > new URL(request.url, `http://${request.headers.host}`)

\* URL {

\* href: 'http://localhost:3000/status?name=ryan',

\* origin: 'http://localhost:3000',

\* protocol: 'http:',

\* username: '',

\* password: '',

\* host: 'localhost:3000',

\* hostname: 'localhost',

\* port: '3000',

\* pathname: '/status',

\* search: '?name=ryan',

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

\* hash: ''

\* }

\* ```

\* @since v0.1.90

\*/

url?: string | undefined;

/\*\*

\* \*\*Only valid for response obtained from {@link ClientRequest}.\*\*

\*

\* The 3-digit HTTP response status code. E.G. `404`.

\* @since v0.1.1

\*/

statusCode?: number | undefined;

/\*\*

\* \*\*Only valid for response obtained from {@link ClientRequest}.\*\*

\*

\* The HTTP response status message (reason phrase). E.G. `OK` or `Internal Server Error`.

\* @since v0.11.10

\*/

statusMessage?: string | undefined;

/\*\*

\* Calls `destroy()` on the socket that received the `IncomingMessage`. If `error`is provided, an `'error'` event is emitted on the socket and `error` is passed

\* as an argument to any listeners on the event.

\* @since v0.3.0

\*/

destroy(error?: Error): void;

}

interface AgentOptions extends Partial<TcpSocketConnectOpts> {

/\*\*

\* Keep sockets around in a pool to be used by other requests in the future. Default = false

\*/

keepAlive?: boolean | undefined;

/\*\*

\* When using HTTP KeepAlive, how often to send TCP KeepAlive packets over sockets being kept alive. Default = 1000.

\* Only relevant if keepAlive is set to true.

\*/

keepAliveMsecs?: number | undefined;

/\*\*

\* Maximum number of sockets to allow per host. Default for Node 0.10 is 5, default for Node 0.12 is Infinity

\*/

maxSockets?: number | undefined;

/\*\*

\* Maximum number of sockets allowed for all hosts in total. Each request will use a new socket until the maximum is reached. Default: Infinity.

\*/

maxTotalSockets?: number | undefined;

/\*\*

\* Maximum number of sockets to leave open in a free state. Only relevant if keepAlive is set to true. Default = 256.

\*/

maxFreeSockets?: number | undefined;

/\*\*

\* Socket timeout in milliseconds. This will set the timeout after the socket is connected.

\*/

timeout?: number | undefined;

/\*\*

\* Scheduling strategy to apply when picking the next free socket to use.

\* @default `lifo`

\*/

scheduling?: 'fifo' | 'lifo' | undefined;

}

/\*\*

\* An `Agent` is responsible for managing connection persistence

\* and reuse for HTTP clients. It maintains a queue of pending requests

\* for a given host and port, reusing a single socket connection for each

\* until the queue is empty, at which time the socket is either destroyed

\* or put into a pool where it is kept to be used again for requests to the

\* same host and port. Whether it is destroyed or pooled depends on the`keepAlive` `option`.

\*

\* Pooled connections have TCP Keep-Alive enabled for them, but servers may

\* still close idle connections, in which case they will be removed from the

\* pool and a new connection will be made when a new HTTP request is made for

\* that host and port. Servers may also refuse to allow multiple requests

\* over the same connection, in which case the connection will have to be

\* remade for every request and cannot be pooled. The `Agent` will still make

\* the requests to that server, but each one will occur over a new connection.

\*

\* When a connection is closed by the client or the server, it is removed

\* from the pool. Any unused sockets in the pool will be unrefed so as not

\* to keep the Node.js process running when there are no outstanding requests.

\* (see `socket.unref()`).

\*

\* It is good practice, to `destroy()` an `Agent` instance when it is no

\* longer in use, because unused sockets consume OS resources.

\*

\* Sockets are removed from an agent when the socket emits either

\* a `'close'` event or an `'agentRemove'` event. When intending to keep one

\* HTTP request open for a long time without keeping it in the agent, something

\* like the following may be done:

\*

\* ```js

\* http.get(options, (res) => {

\* // Do stuff

\* }).on('socket', (socket) => {

\* socket.emit('agentRemove');

\* });

\* ```

\*

\* An agent may also be used for an individual request. By providing`{agent: false}` as an option to the `http.get()` or `http.request()`functions, a one-time use `Agent` with default options

\* will be used

\* for the client connection.

\*

\* `agent:false`:

\*

\* ```js

\* http.get({

\* hostname: 'localhost',

\* port: 80,

\* path: '/',

\* agent: false // Create a new agent just for this one request

\* }, (res) => {

\* // Do stuff with response

\* });

\* ```

\* @since v0.3.4

\*/

class Agent {

/\*\*

\* By default set to 256\. For agents with `keepAlive` enabled, this

\* sets the maximum number of sockets that will be left open in the free

\* state.

\* @since v0.11.7

\*/

maxFreeSockets: number;

/\*\*

\* By default set to `Infinity`. Determines how many concurrent sockets the agent

\* can have open per origin. Origin is the returned value of `agent.getName()`.

\* @since v0.3.6

\*/

maxSockets: number;

/\*\*

\* By default set to `Infinity`. Determines how many concurrent sockets the agent

\* can have open. Unlike `maxSockets`, this parameter applies across all origins.

\* @since v14.5.0, v12.19.0

\*/

maxTotalSockets: number;

/\*\*

\* An object which contains arrays of sockets currently awaiting use by

\* the agent when `keepAlive` is enabled. Do not modify.

\*

\* Sockets in the `freeSockets` list will be automatically destroyed and

\* removed from the array on `'timeout'`.

\* @since v0.11.4

\*/

readonly freeSockets: NodeJS.ReadOnlyDict<Socket[]>;

/\*\*

\* An object which contains arrays of sockets currently in use by the

\* agent. Do not modify.

\* @since v0.3.6

\*/

readonly sockets: NodeJS.ReadOnlyDict<Socket[]>;

/\*\*

\* An object which contains queues of requests that have not yet been assigned to

\* sockets. Do not modify.

\* @since v0.5.9

\*/

readonly requests: NodeJS.ReadOnlyDict<IncomingMessage[]>;

constructor(opts?: AgentOptions);

/\*\*

\* Destroy any sockets that are currently in use by the agent.

\*

\* It is usually not necessary to do this. However, if using an

\* agent with `keepAlive` enabled, then it is best to explicitly shut down

\* the agent when it is no longer needed. Otherwise,

\* sockets might stay open for quite a long time before the server

\* terminates them.

\* @since v0.11.4

\*/

destroy(): void;

}

const METHODS: string[];

const STATUS\_CODES: {

[errorCode: number]: string | undefined;

[errorCode: string]: string | undefined;

};

/\*\*

\* Returns a new instance of {@link Server}.

\*

\* The `requestListener` is a function which is automatically

\* added to the `'request'` event.

\* @since v0.1.13

\*/

function createServer(requestListener?: RequestListener): Server;

function createServer(options: ServerOptions, requestListener?: RequestListener): Server;

// although RequestOptions are passed as ClientRequestArgs to ClientRequest directly,

// create interface RequestOptions would make the naming more clear to developers

interface RequestOptions extends ClientRequestArgs {}

/\*\*

\* Node.js maintains several connections per server to make HTTP requests.

\* This function allows one to transparently issue requests.

\*

\* `url` can be a string or a `URL` object. If `url` is a

\* string, it is automatically parsed with `new URL()`. If it is a `URL` object, it will be automatically converted to an ordinary `options` object.

\*

\* If both `url` and `options` are specified, the objects are merged, with the`options` properties taking precedence.

\*

\* The optional `callback` parameter will be added as a one-time listener for

\* the `'response'` event.

\*

\* `http.request()` returns an instance of the {@link ClientRequest} class. The `ClientRequest` instance is a writable stream. If one needs to

\* upload a file with a POST request, then write to the `ClientRequest` object.

\*

\* ```js

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

\*

\* const postData = JSON.stringify({

\* 'msg': 'Hello World!'

\* });

\*

\* const options = {

\* hostname: 'www.google.com',

\* port: 80,

\* path: '/upload',

\* method: 'POST',

\* headers: {

\* 'Content-Type': 'application/json',

\* 'Content-Length': Buffer.byteLength(postData)

\* }

\* };

\*

\* const req = http.request(options, (res) => {

\* console.log(`STATUS: ${res.statusCode}`);

\* console.log(`HEADERS: ${JSON.stringify(res.headers)}`);

\* res.setEncoding('utf8');

\* res.on('data', (chunk) => {

\* console.log(`BODY: ${chunk}`);

\* });

\* res.on('end', () => {

\* console.log('No more data in response.');

\* });

\* });

\*

\* req.on('error', (e) => {

\* console.error(`problem with request: ${e.message}`);

\* });

\*

\* // Write data to request body

\* req.write(postData);

\* req.end();

\* ```

\*

\* In the example `req.end()` was called. With `http.request()` one

\* must always call `req.end()` to signify the end of the request -

\* even if there is no data being written to the request body.

\*

\* If any error is encountered during the request (be that with DNS resolution,

\* TCP level errors, or actual HTTP parse errors) an `'error'` event is emitted

\* on the returned request object. As with all `'error'` events, if no listeners

\* are registered the error will be thrown.

\*

\* There are a few special headers that should be noted.

\*

\* \* Sending a 'Connection: keep-alive' will notify Node.js that the connection to

\* the server should be persisted until the next request.

\* \* Sending a 'Content-Length' header will disable the default chunked encoding.

\* \* Sending an 'Expect' header will immediately send the request headers.

\* Usually, when sending 'Expect: 100-continue', both a timeout and a listener

\* for the `'continue'` event should be set. See RFC 2616 Section 8.2.3 for more

\* information.

\* \* Sending an Authorization header will override using the `auth` option

\* to compute basic authentication.

\*

\* Example using a `URL` as `options`:

\*

\* ```js

\* const options = new URL('http://abc:xyz@example.com');

\*

\* const req = http.request(options, (res) => {

\* // ...

\* });

\* ```

\*

\* In a successful request, the following events will be emitted in the following

\* order:

\*

\* \* `'socket'`

\* \* `'response'`

\* \* `'data'` any number of times, on the `res` object

\* (`'data'` will not be emitted at all if the response body is empty, for

\* instance, in most redirects)

\* \* `'end'` on the `res` object

\* \* `'close'`

\*

\* In the case of a connection error, the following events will be emitted:

\*

\* \* `'socket'`

\* \* `'error'`

\* \* `'close'`

\*

\* In the case of a premature connection close before the response is received,

\* the following events will be emitted in the following order:

\*

\* \* `'socket'`

\* \* `'error'` with an error with message `'Error: socket hang up'` and code`'ECONNRESET'`

\* \* `'close'`

\*

\* In the case of a premature connection close after the response is received,

\* the following events will be emitted in the following order:

\*

\* \* `'socket'`

\* \* `'response'`

\* \* `'data'` any number of times, on the `res` object

\* \* (connection closed here)

\* \* `'aborted'` on the `res` object

\* \* `'error'` on the `res` object with an error with message`'Error: aborted'` and code `'ECONNRESET'`.

\* \* `'close'`

\* \* `'close'` on the `res` object

\*

\* If `req.destroy()` is called before a socket is assigned, the following

\* events will be emitted in the following order:

\*

\* \* (`req.destroy()` called here)

\* \* `'error'` with an error with message `'Error: socket hang up'` and code`'ECONNRESET'`

\* \* `'close'`

\*

\* If `req.destroy()` is called before the connection succeeds, the following

\* events will be emitted in the following order:

\*

\* \* `'socket'`

\* \* (`req.destroy()` called here)

\* \* `'error'` with an error with message `'Error: socket hang up'` and code`'ECONNRESET'`

\* \* `'close'`

\*

\* If `req.destroy()` is called after the response is received, the following

\* events will be emitted in the following order:

\*

\* \* `'socket'`

\* \* `'response'`

\* \* `'data'` any number of times, on the `res` object

\* \* (`req.destroy()` called here)

\* \* `'aborted'` on the `res` object

\* \* `'error'` on the `res` object with an error with message`'Error: aborted'` and code `'ECONNRESET'`.

\* \* `'close'`

\* \* `'close'` on the `res` object

\*

\* If `req.abort()` is called before a socket is assigned, the following

\* events will be emitted in the following order:

\*

\* \* (`req.abort()` called here)

\* \* `'abort'`

\* \* `'close'`

\*

\* If `req.abort()` is called before the connection succeeds, the following

\* events will be emitted in the following order:

\*

\* \* `'socket'`

\* \* (`req.abort()` called here)

\* \* `'abort'`

\* \* `'error'` with an error with message `'Error: socket hang up'` and code`'ECONNRESET'`

\* \* `'close'`

\*

\* If `req.abort()` is called after the response is received, the following

\* events will be emitted in the following order:

\*

\* \* `'socket'`

\* \* `'response'`

\* \* `'data'` any number of times, on the `res` object

\* \* (`req.abort()` called here)

\* \* `'abort'`

\* \* `'aborted'` on the `res` object

\* \* `'error'` on the `res` object with an error with message`'Error: aborted'` and code `'ECONNRESET'`.

\* \* `'close'`

\* \* `'close'` on the `res` object

\*

\* Setting the `timeout` option or using the `setTimeout()` function will

\* not abort the request or do anything besides add a `'timeout'` event.

\*

\* Passing an `AbortSignal` and then calling `abort` on the corresponding`AbortController` will behave the same way as calling `.destroy()` on the

\* request itself.

\* @since v0.3.6

\*/

function request(options: RequestOptions | string | URL, callback?: (res: IncomingMessage) => void): ClientRequest;

function request(url: string | URL, options: RequestOptions, callback?: (res: IncomingMessage) => void): ClientRequest;

/\*\*

\* Since most requests are GET requests without bodies, Node.js provides this

\* convenience method. The only difference between this method and {@link request} is that it sets the method to GET and calls `req.end()`automatically. The callback must take care to consume the

\* response

\* data for reasons stated in {@link ClientRequest} section.

\*

\* The `callback` is invoked with a single argument that is an instance of {@link IncomingMessage}.

\*

\* JSON fetching example:

\*

\* ```js

\* http.get('http://localhost:8000/', (res) => {

\* const { statusCode } = res;

\* const contentType = res.headers['content-type'];

\*

\* let error;

\* // Any 2xx status code signals a successful response but

\* // here we're only checking for 200.

\* if (statusCode !== 200) {

\* error = new Error('Request Failed.\n' +

\* `Status Code: ${statusCode}`);

\* } else if (!/^application\/json/.test(contentType)) {

\* error = new Error('Invalid content-type.\n' +

\* `Expected application/json but received ${contentType}`);

\* }

\* if (error) {

\* console.error(error.message);

\* // Consume response data to free up memory

\* res.resume();

\* return;

\* }

\*

\* res.setEncoding('utf8');

\* let rawData = '';

\* res.on('data', (chunk) => { rawData += chunk; });

\* res.on('end', () => {

\* try {

\* const parsedData = JSON.parse(rawData);

\* console.log(parsedData);

\* } catch (e) {

\* console.error(e.message);

\* }

\* });

\* }).on('error', (e) => {

\* console.error(`Got error: ${e.message}`);

\* });

\*

\* // Create a local server to receive data from

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

\* res.writeHead(200, { 'Content-Type': 'application/json' });

\* res.end(JSON.stringify({

\* data: 'Hello World!'

\* }));

\* });

\*

\* server.listen(8000);

\* ```

\* @since v0.3.6

\* @param options Accepts the same `options` as {@link request}, with the `method` always set to `GET`. Properties that are inherited from the prototype are ignored.

\*/

function get(options: RequestOptions | string | URL, callback?: (res: IncomingMessage) => void): ClientRequest;

function get(url: string | URL, options: RequestOptions, callback?: (res: IncomingMessage) => void): ClientRequest;

let globalAgent: Agent;

/\*\*

\* Read-only property specifying the maximum allowed size of HTTP headers in bytes.

\* Defaults to 16KB. Configurable using the `--max-http-header-size` CLI option.

\*/

const maxHeaderSize: number;

}

declare module 'node:http' {

export \* from 'http';

}