/\*\*

\* > Stability: 2 - Stable

\*

\* The `net` module provides an asynchronous network API for creating stream-based

\* TCP or `IPC` servers ({@link createServer}) and clients

\* ({@link createConnection}).

\*

\* It can be accessed using:

\*

\* ```js

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

\* ```

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

\*/

declare module 'net' {

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

import { Abortable, EventEmitter } from 'node:events';

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

type LookupFunction = (hostname: string, options: dns.LookupOneOptions, callback: (err: NodeJS.ErrnoException | null, address: string, family: number) => void) => void;

interface AddressInfo {

address: string;

family: string;

port: number;

}

interface SocketConstructorOpts {

fd?: number | undefined;

allowHalfOpen?: boolean | undefined;

readable?: boolean | undefined;

writable?: boolean | undefined;

}

interface OnReadOpts {

buffer: Uint8Array | (() => Uint8Array);

/\*\*

\* This function is called for every chunk of incoming data.

\* Two arguments are passed to it: the number of bytes written to buffer and a reference to buffer.

\* Return false from this function to implicitly pause() the socket.

\*/

callback(bytesWritten: number, buf: Uint8Array): boolean;

}

interface ConnectOpts {

/\*\*

\* If specified, incoming data is stored in a single buffer and passed to the supplied callback when data arrives on the socket.

\* Note: this will cause the streaming functionality to not provide any data, however events like 'error', 'end', and 'close' will

\* still be emitted as normal and methods like pause() and resume() will also behave as expected.

\*/

onread?: OnReadOpts | undefined;

}

interface TcpSocketConnectOpts extends ConnectOpts {

port: number;

host?: string | undefined;

localAddress?: string | undefined;

localPort?: number | undefined;

hints?: number | undefined;

family?: number | undefined;

lookup?: LookupFunction | undefined;

}

interface IpcSocketConnectOpts extends ConnectOpts {

path: string;

}

type SocketConnectOpts = TcpSocketConnectOpts | IpcSocketConnectOpts;

/\*\*

\* This class is an abstraction of a TCP socket or a streaming `IPC` endpoint

\* (uses named pipes on Windows, and Unix domain sockets otherwise). It is also

\* an `EventEmitter`.

\*

\* A `net.Socket` can be created by the user and used directly to interact with

\* a server. For example, it is returned by {@link createConnection},

\* so the user can use it to talk to the server.

\*

\* It can also be created by Node.js and passed to the user when a connection

\* is received. For example, it is passed to the listeners of a `'connection'` event emitted on a {@link Server}, so the user can use

\* it to interact with the client.

\* @since v0.3.4

\*/

class Socket extends stream.Duplex {

constructor(options?: SocketConstructorOpts);

/\*\*

\* Sends data on the socket. The second parameter specifies the encoding in the

\* case of a string. It defaults to UTF8 encoding.

\*

\* 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 again free.

\*

\* The optional `callback` parameter will be executed when the data is finally

\* written out, which may not be immediately.

\*

\* See `Writable` stream `write()` method for more

\* information.

\* @since v0.1.90

\* @param [encoding='utf8'] Only used when data is `string`.

\*/

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

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

/\*\*

\* Initiate a connection on a given socket.

\*

\* Possible signatures:

\*

\* \* `socket.connect(options[, connectListener])`

\* \* `socket.connect(path[, connectListener])` for `IPC` connections.

\* \* `socket.connect(port[, host][, connectListener])` for TCP connections.

\* \* Returns: `net.Socket` The socket itself.

\*

\* This function is asynchronous. When the connection is established, the `'connect'` event will be emitted. If there is a problem connecting,

\* instead of a `'connect'` event, an `'error'` event will be emitted with

\* the error passed to the `'error'` listener.

\* The last parameter `connectListener`, if supplied, will be added as a listener

\* for the `'connect'` event \*\*once\*\*.

\*

\* This function should only be used for reconnecting a socket after`'close'` has been emitted or otherwise it may lead to undefined

\* behavior.

\*/

connect(options: SocketConnectOpts, connectionListener?: () => void): this;

connect(port: number, host: string, connectionListener?: () => void): this;

connect(port: number, connectionListener?: () => void): this;

connect(path: string, connectionListener?: () => void): this;

/\*\*

\* Set the encoding for the socket as a `Readable Stream`. See `readable.setEncoding()` for more information.

\* @since v0.1.90

\* @return The socket itself.

\*/

setEncoding(encoding?: BufferEncoding): this;

/\*\*

\* Pauses the reading of data. That is, `'data'` events will not be emitted.

\* Useful to throttle back an upload.

\* @return The socket itself.

\*/

pause(): this;

/\*\*

\* Resumes reading after a call to `socket.pause()`.

\* @return The socket itself.

\*/

resume(): this;

/\*\*

\* Sets the socket to timeout after `timeout` milliseconds of inactivity on

\* the socket. By default `net.Socket` do not have a timeout.

\*

\* When an idle timeout is triggered the socket will receive a `'timeout'` event but the connection will not be severed. The user must manually call `socket.end()` or `socket.destroy()` to

\* end the connection.

\*

\* ```js

\* socket.setTimeout(3000);

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

\* console.log('socket timeout');

\* socket.end();

\* });

\* ```

\*

\* If `timeout` is 0, then the existing idle timeout is disabled.

\*

\* The optional `callback` parameter will be added as a one-time listener for the `'timeout'` event.

\* @since v0.1.90

\* @return The socket itself.

\*/

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

/\*\*

\* Enable/disable the use of Nagle's algorithm.

\*

\* When a TCP connection is created, it will have Nagle's algorithm enabled.

\*

\* Nagle's algorithm delays data before it is sent via the network. It attempts

\* to optimize throughput at the expense of latency.

\*

\* Passing `true` for `noDelay` or not passing an argument will disable Nagle's

\* algorithm for the socket. Passing `false` for `noDelay` will enable Nagle's

\* algorithm.

\* @since v0.1.90

\* @param [noDelay=true]

\* @return The socket itself.

\*/

setNoDelay(noDelay?: boolean): this;

/\*\*

\* Enable/disable keep-alive functionality, and optionally set the initial

\* delay before the first keepalive probe is sent on an idle socket.

\*

\* Set `initialDelay` (in milliseconds) to set the delay between the last

\* data packet received and the first keepalive probe. Setting `0` for`initialDelay` will leave the value unchanged from the default

\* (or previous) setting.

\*

\* Enabling the keep-alive functionality will set the following socket options:

\*

\* \* `SO\_KEEPALIVE=1`

\* \* `TCP\_KEEPIDLE=initialDelay`

\* \* `TCP\_KEEPCNT=10`

\* \* `TCP\_KEEPINTVL=1`

\* @since v0.1.92

\* @param [enable=false]

\* @param [initialDelay=0]

\* @return The socket itself.

\*/

setKeepAlive(enable?: boolean, initialDelay?: number): this;

/\*\*

\* Returns the bound `address`, the address `family` name and `port` of the

\* socket as reported by the operating system:`{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`

\* @since v0.1.90

\*/

address(): AddressInfo | {};

/\*\*

\* Calling `unref()` on a socket will allow the program to exit if this is the only

\* active socket in the event system. If the socket is already `unref`ed calling`unref()` again will have no effect.

\* @since v0.9.1

\* @return The socket itself.

\*/

unref(): this;

/\*\*

\* Opposite of `unref()`, calling `ref()` on a previously `unref`ed socket will\_not\_ let the program exit if it's the only socket left (the default behavior).

\* If the socket is `ref`ed calling `ref` again will have no effect.

\* @since v0.9.1

\* @return The socket itself.

\*/

ref(): this;

/\*\*

\* This property shows the number of characters buffered for writing. The buffer

\* may contain strings whose length after encoding is not yet known. So this number

\* is only an approximation of the number of bytes in the buffer.

\*

\* `net.Socket` has the property that `socket.write()` always works. This is to

\* help users get up and running quickly. The computer cannot always keep up

\* with the amount of data that is written to a socket. The network connection

\* simply might be too slow. Node.js will internally queue up the data written to a

\* socket and send it out over the wire when it is possible.

\*

\* The consequence of this internal buffering is that memory may grow.

\* Users who experience large or growing `bufferSize` should attempt to

\* "throttle" the data flows in their program with `socket.pause()` and `socket.resume()`.

\* @since v0.3.8

\* @deprecated Since v14.6.0 - Use `writableLength` instead.

\*/

readonly bufferSize: number;

/\*\*

\* The amount of received bytes.

\* @since v0.5.3

\*/

readonly bytesRead: number;

/\*\*

\* The amount of bytes sent.

\* @since v0.5.3

\*/

readonly bytesWritten: number;

/\*\*

\* If `true`,`socket.connect(options[, connectListener])` was

\* called and has not yet finished. It will stay `true` until the socket becomes

\* connected, then it is set to `false` and the `'connect'` event is emitted. Note

\* that the `socket.connect(options[, connectListener])` callback is a listener for the `'connect'` event.

\* @since v6.1.0

\*/

readonly connecting: boolean;

/\*\*

\* See `writable.destroyed` for further details.

\*/

readonly destroyed: boolean;

/\*\*

\* The string representation of the local IP address the remote client is

\* connecting on. For example, in a server listening on `'0.0.0.0'`, if a client

\* connects on `'192.168.1.1'`, the value of `socket.localAddress` would be`'192.168.1.1'`.

\* @since v0.9.6

\*/

readonly localAddress?: string;

/\*\*

\* The numeric representation of the local port. For example, `80` or `21`.

\* @since v0.9.6

\*/

readonly localPort?: number;

/\*\*

\* The string representation of the remote IP address. For example,`'74.125.127.100'` or `'2001:4860:a005::68'`. Value may be `undefined` if

\* the socket is destroyed (for example, if the client disconnected).

\* @since v0.5.10

\*/

readonly remoteAddress?: string | undefined;

/\*\*

\* The string representation of the remote IP family. `'IPv4'` or `'IPv6'`.

\* @since v0.11.14

\*/

readonly remoteFamily?: string | undefined;

/\*\*

\* The numeric representation of the remote port. For example, `80` or `21`.

\* @since v0.5.10

\*/

readonly remotePort?: number | undefined;

/\*\*

\* Half-closes the socket. i.e., it sends a FIN packet. It is possible the

\* server will still send some data.

\*

\* See `writable.end()` for further details.

\* @since v0.1.90

\* @param [encoding='utf8'] Only used when data is `string`.

\* @param callback Optional callback for when the socket is finished.

\* @return The socket itself.

\*/

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

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

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

/\*\*

\* events.EventEmitter

\* 1. close

\* 2. connect

\* 3. data

\* 4. drain

\* 5. end

\* 6. error

\* 7. lookup

\* 8. timeout

\*/

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

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

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

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

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

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

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

addListener(event: 'lookup', listener: (err: Error, address: string, family: string | number, host: string) => void): this;

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

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

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

emit(event: 'close', hadError: boolean): boolean;

emit(event: 'connect'): boolean;

emit(event: 'data', data: Buffer): boolean;

emit(event: 'drain'): boolean;

emit(event: 'end'): boolean;

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

emit(event: 'lookup', err: Error, address: string, family: string | number, host: string): boolean;

emit(event: 'ready'): boolean;

emit(event: 'timeout'): boolean;

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

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

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

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

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

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

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

on(event: 'lookup', listener: (err: Error, address: string, family: string | number, host: string) => void): this;

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

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

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

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

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

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

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

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

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

once(event: 'lookup', listener: (err: Error, address: string, family: string | number, host: string) => void): this;

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

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

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

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

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

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

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

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

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

prependListener(event: 'lookup', listener: (err: Error, address: string, family: string | number, host: string) => void): this;

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

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

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

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

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

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

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

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

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

prependOnceListener(event: 'lookup', listener: (err: Error, address: string, family: string | number, host: string) => void): this;

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

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

}

interface ListenOptions extends Abortable {

port?: number | undefined;

host?: string | undefined;

backlog?: number | undefined;

path?: string | undefined;

exclusive?: boolean | undefined;

readableAll?: boolean | undefined;

writableAll?: boolean | undefined;

/\*\*

\* @default false

\*/

ipv6Only?: boolean | undefined;

}

interface ServerOpts {

/\*\*

\* Indicates whether half-opened TCP connections are allowed.

\* @default false

\*/

allowHalfOpen?: boolean | undefined;

/\*\*

\* Indicates whether the socket should be paused on incoming connections.

\* @default false

\*/

pauseOnConnect?: boolean | undefined;

}

/\*\*

\* This class is used to create a TCP or `IPC` server.

\* @since v0.1.90

\*/

class Server extends EventEmitter {

constructor(connectionListener?: (socket: Socket) => void);

constructor(options?: ServerOpts, connectionListener?: (socket: Socket) => void);

/\*\*

\* Start a server listening for connections. A `net.Server` can be a TCP or

\* an `IPC` server depending on what it listens to.

\*

\* Possible signatures:

\*

\* \* `server.listen(handle[, backlog][, callback])`

\* \* `server.listen(options[, callback])`

\* \* `server.listen(path[, backlog][, callback])` for `IPC` servers

\* \* `server.listen([port[, host[, backlog]]][, callback])` for TCP servers

\*

\* This function is asynchronous. When the server starts listening, the `'listening'` event will be emitted. The last parameter `callback`will be added as a listener for the `'listening'`

\* event.

\*

\* All `listen()` methods can take a `backlog` parameter to specify the maximum

\* length of the queue of pending connections. The actual length will be determined

\* by the OS through sysctl settings such as `tcp\_max\_syn\_backlog` and `somaxconn`on Linux. The default value of this parameter is 511 (not 512).

\*

\* All {@link Socket} are set to `SO\_REUSEADDR` (see [`socket(7)`](https://man7.org/linux/man-pages/man7/socket.7.html) for

\* details).

\*

\* The `server.listen()` method can be called again if and only if there was an

\* error during the first `server.listen()` call or `server.close()` has been

\* called. Otherwise, an `ERR\_SERVER\_ALREADY\_LISTEN` error will be thrown.

\*

\* One of the most common errors raised when listening is `EADDRINUSE`.

\* This happens when another server is already listening on the requested`port`/`path`/`handle`. One way to handle this would be to retry

\* after a certain amount of time:

\*

\* ```js

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

\* if (e.code === 'EADDRINUSE') {

\* console.log('Address in use, retrying...');

\* setTimeout(() => {

\* server.close();

\* server.listen(PORT, HOST);

\* }, 1000);

\* }

\* });

\* ```

\*/

listen(port?: number, hostname?: string, backlog?: number, listeningListener?: () => void): this;

listen(port?: number, hostname?: string, listeningListener?: () => void): this;

listen(port?: number, backlog?: number, listeningListener?: () => void): this;

listen(port?: number, listeningListener?: () => void): this;

listen(path: string, backlog?: number, listeningListener?: () => void): this;

listen(path: string, listeningListener?: () => void): this;

listen(options: ListenOptions, listeningListener?: () => void): this;

listen(handle: any, backlog?: number, listeningListener?: () => void): this;

listen(handle: any, listeningListener?: () => void): this;

/\*\*

\* Stops the server from accepting new connections and keeps existing

\* connections. This function is asynchronous, the server is finally closed

\* when all connections are ended and the server emits a `'close'` event.

\* The optional `callback` will be called once the `'close'` event occurs. Unlike

\* that event, it will be called with an `Error` as its only argument if the server

\* was not open when it was closed.

\* @since v0.1.90

\* @param callback Called when the server is closed.

\*/

close(callback?: (err?: Error) => void): this;

/\*\*

\* Returns the bound `address`, the address `family` name, and `port` of the server

\* as reported by the operating system if listening on an IP socket

\* (useful to find which port was assigned when getting an OS-assigned address):`{ port: 12346, family: 'IPv4', address: '127.0.0.1' }`.

\*

\* For a server listening on a pipe or Unix domain socket, the name is returned

\* as a string.

\*

\* ```js

\* const server = net.createServer((socket) => {

\* socket.end('goodbye\n');

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

\* // Handle errors here.

\* throw err;

\* });

\*

\* // Grab an arbitrary unused port.

\* server.listen(() => {

\* console.log('opened server on', server.address());

\* });

\* ```

\*

\* `server.address()` returns `null` before the `'listening'` event has been

\* emitted or after calling `server.close()`.

\* @since v0.1.90

\*/

address(): AddressInfo | string | null;

/\*\*

\* Asynchronously get the number of concurrent connections on the server. Works

\* when sockets were sent to forks.

\*

\* Callback should take two arguments `err` and `count`.

\* @since v0.9.7

\*/

getConnections(cb: (error: Error | null, count: number) => void): void;

/\*\*

\* Opposite of `unref()`, calling `ref()` on a previously `unref`ed server will\_not\_ let the program exit if it's the only server left (the default behavior).

\* If the server is `ref`ed calling `ref()` again will have no effect.

\* @since v0.9.1

\*/

ref(): this;

/\*\*

\* Calling `unref()` on a server will allow the program to exit if this is the only

\* active server in the event system. If the server is already `unref`ed calling`unref()` again will have no effect.

\* @since v0.9.1

\*/

unref(): this;

/\*\*

\* Set this property to reject connections when the server's connection count gets

\* high.

\*

\* It is not recommended to use this option once a socket has been sent to a child

\* with `child\_process.fork()`.

\* @since v0.2.0

\*/

maxConnections: number;

connections: number;

/\*\*

\* Indicates whether or not the server is listening for connections.

\* @since v5.7.0

\*/

listening: boolean;

/\*\*

\* events.EventEmitter

\* 1. close

\* 2. connection

\* 3. error

\* 4. listening

\*/

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;

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

emit(event: 'close'): boolean;

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

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

emit(event: 'listening'): 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;

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;

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;

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;

}

type IPVersion = 'ipv4' | 'ipv6';

/\*\*

\* The `BlockList` object can be used with some network APIs to specify rules for

\* disabling inbound or outbound access to specific IP addresses, IP ranges, or

\* IP subnets.

\* @since v15.0.0, v14.18.0

\*/

class BlockList {

/\*\*

\* Adds a rule to block the given IP address.

\* @since v15.0.0, v14.18.0

\* @param address An IPv4 or IPv6 address.

\* @param [type='ipv4'] Either `'ipv4'` or `'ipv6'`.

\*/

addAddress(address: string, type?: IPVersion): void;

addAddress(address: SocketAddress): void;

/\*\*

\* Adds a rule to block a range of IP addresses from `start` (inclusive) to`end` (inclusive).

\* @since v15.0.0, v14.18.0

\* @param start The starting IPv4 or IPv6 address in the range.

\* @param end The ending IPv4 or IPv6 address in the range.

\* @param [type='ipv4'] Either `'ipv4'` or `'ipv6'`.

\*/

addRange(start: string, end: string, type?: IPVersion): void;

addRange(start: SocketAddress, end: SocketAddress): void;

/\*\*

\* Adds a rule to block a range of IP addresses specified as a subnet mask.

\* @since v15.0.0, v14.18.0

\* @param net The network IPv4 or IPv6 address.

\* @param prefix The number of CIDR prefix bits. For IPv4, this must be a value between `0` and `32`. For IPv6, this must be between `0` and `128`.

\* @param [type='ipv4'] Either `'ipv4'` or `'ipv6'`.

\*/

addSubnet(net: SocketAddress, prefix: number): void;

addSubnet(net: string, prefix: number, type?: IPVersion): void;

/\*\*

\* Returns `true` if the given IP address matches any of the rules added to the`BlockList`.

\*

\* ```js

\* const blockList = new net.BlockList();

\* blockList.addAddress('123.123.123.123');

\* blockList.addRange('10.0.0.1', '10.0.0.10');

\* blockList.addSubnet('8592:757c:efae:4e45::', 64, 'ipv6');

\*

\* console.log(blockList.check('123.123.123.123')); // Prints: true

\* console.log(blockList.check('10.0.0.3')); // Prints: true

\* console.log(blockList.check('222.111.111.222')); // Prints: false

\*

\* // IPv6 notation for IPv4 addresses works:

\* console.log(blockList.check('::ffff:7b7b:7b7b', 'ipv6')); // Prints: true

\* console.log(blockList.check('::ffff:123.123.123.123', 'ipv6')); // Prints: true

\* ```

\* @since v15.0.0, v14.18.0

\* @param address The IP address to check

\* @param [type='ipv4'] Either `'ipv4'` or `'ipv6'`.

\*/

check(address: SocketAddress): boolean;

check(address: string, type?: IPVersion): boolean;

}

interface TcpNetConnectOpts extends TcpSocketConnectOpts, SocketConstructorOpts {

timeout?: number | undefined;

}

interface IpcNetConnectOpts extends IpcSocketConnectOpts, SocketConstructorOpts {

timeout?: number | undefined;

}

type NetConnectOpts = TcpNetConnectOpts | IpcNetConnectOpts;

/\*\*

\* Creates a new TCP or `IPC` server.

\*

\* If `allowHalfOpen` is set to `true`, when the other end of the socket

\* signals the end of transmission, the server will only send back the end of

\* transmission when `socket.end()` is explicitly called. For example, in the

\* context of TCP, when a FIN packed is received, a FIN packed is sent

\* back only when `socket.end()` is explicitly called. Until then the

\* connection is half-closed (non-readable but still writable). See `'end'` event and [RFC 1122](https://tools.ietf.org/html/rfc1122) (section 4.2.2.13) for more information.

\*

\* If `pauseOnConnect` is set to `true`, then the socket associated with each

\* incoming connection will be paused, and no data will be read from its handle.

\* This allows connections to be passed between processes without any data being

\* read by the original process. To begin reading data from a paused socket, call `socket.resume()`.

\*

\* The server can be a TCP server or an `IPC` server, depending on what it `listen()` to.

\*

\* Here is an example of an TCP echo server which listens for connections

\* on port 8124:

\*

\* ```js

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

\* const server = net.createServer((c) => {

\* // 'connection' listener.

\* console.log('client connected');

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

\* console.log('client disconnected');

\* });

\* c.write('hello\r\n');

\* c.pipe(c);

\* });

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

\* throw err;

\* });

\* server.listen(8124, () => {

\* console.log('server bound');

\* });

\* ```

\*

\* Test this by using `telnet`:

\*

\* ```console

\* $ telnet localhost 8124

\* ```

\*

\* To listen on the socket `/tmp/echo.sock`:

\*

\* ```js

\* server.listen('/tmp/echo.sock', () => {

\* console.log('server bound');

\* });

\* ```

\*

\* Use `nc` to connect to a Unix domain socket server:

\*

\* ```console

\* $ nc -U /tmp/echo.sock

\* ```

\* @since v0.5.0

\* @param connectionListener Automatically set as a listener for the {@link 'connection'} event.

\*/

function createServer(connectionListener?: (socket: Socket) => void): Server;

function createServer(options?: ServerOpts, connectionListener?: (socket: Socket) => void): Server;

/\*\*

\* Aliases to {@link createConnection}.

\*

\* Possible signatures:

\*

\* \* {@link connect}

\* \* {@link connect} for `IPC` connections.

\* \* {@link connect} for TCP connections.

\*/

function connect(options: NetConnectOpts, connectionListener?: () => void): Socket;

function connect(port: number, host?: string, connectionListener?: () => void): Socket;

function connect(path: string, connectionListener?: () => void): Socket;

/\*\*

\* A factory function, which creates a new {@link Socket},

\* immediately initiates connection with `socket.connect()`,

\* then returns the `net.Socket` that starts the connection.

\*

\* When the connection is established, a `'connect'` event will be emitted

\* on the returned socket. The last parameter `connectListener`, if supplied,

\* will be added as a listener for the `'connect'` event \*\*once\*\*.

\*

\* Possible signatures:

\*

\* \* {@link createConnection}

\* \* {@link createConnection} for `IPC` connections.

\* \* {@link createConnection} for TCP connections.

\*

\* The {@link connect} function is an alias to this function.

\*/

function createConnection(options: NetConnectOpts, connectionListener?: () => void): Socket;

function createConnection(port: number, host?: string, connectionListener?: () => void): Socket;

function createConnection(path: string, connectionListener?: () => void): Socket;

/\*\*

\* Tests if input is an IP address. Returns `0` for invalid strings,

\* returns `4` for IP version 4 addresses, and returns `6` for IP version 6

\* addresses.

\* @since v0.3.0

\*/

function isIP(input: string): number;

/\*\*

\* Returns `true` if input is a version 4 IP address, otherwise returns `false`.

\* @since v0.3.0

\*/

function isIPv4(input: string): boolean;

/\*\*

\* Returns `true` if input is a version 6 IP address, otherwise returns `false`.

\* @since v0.3.0

\*/

function isIPv6(input: string): boolean;

interface SocketAddressInitOptions {

/\*\*

\* The network address as either an IPv4 or IPv6 string.

\* @default 127.0.0.1

\*/

address?: string | undefined;

/\*\*

\* @default `'ipv4'`

\*/

family?: IPVersion | undefined;

/\*\*

\* An IPv6 flow-label used only if `family` is `'ipv6'`.

\* @default 0

\*/

flowlabel?: number | undefined;

/\*\*

\* An IP port.

\* @default 0

\*/

port?: number | undefined;

}

/\*\*

\* @since v15.14.0, v14.18.0

\*/

class SocketAddress {

constructor(options: SocketAddressInitOptions);

/\*\*

\* Either \`'ipv4'\` or \`'ipv6'\`.

\* @since v15.14.0, v14.18.0

\*/

readonly address: string;

/\*\*

\* Either \`'ipv4'\` or \`'ipv6'\`.

\* @since v15.14.0, v14.18.0

\*/

readonly family: IPVersion;

/\*\*

\* @since v15.14.0, v14.18.0

\*/

readonly port: number;

/\*\*

\* @since v15.14.0, v14.18.0

\*/

readonly flowlabel: number;

}

}

declare module 'node:net' {

export \* from 'net';

}