// tslint:disable-next-line:dt-header

// Type definitions for inspector

// These definitions are auto-generated.

// Please see https://github.com/DefinitelyTyped/DefinitelyTyped/pull/19330

// for more information.

// tslint:disable:max-line-length

/\*\*

\* The `inspector` module provides an API for interacting with the V8 inspector.

\*

\* It can be accessed using:

\*

\* ```js

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

\* ```

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

\*/

declare module 'inspector' {

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

interface InspectorNotification<T> {

method: string;

params: T;

}

namespace Schema {

/\*\*

\* Description of the protocol domain.

\*/

interface Domain {

/\*\*

\* Domain name.

\*/

name: string;

/\*\*

\* Domain version.

\*/

version: string;

}

interface GetDomainsReturnType {

/\*\*

\* List of supported domains.

\*/

domains: Domain[];

}

}

namespace Runtime {

/\*\*

\* Unique script identifier.

\*/

type ScriptId = string;

/\*\*

\* Unique object identifier.

\*/

type RemoteObjectId = string;

/\*\*

\* Primitive value which cannot be JSON-stringified.

\*/

type UnserializableValue = string;

/\*\*

\* Mirror object referencing original JavaScript object.

\*/

interface RemoteObject {

/\*\*

\* Object type.

\*/

type: string;

/\*\*

\* Object subtype hint. Specified for <code>object</code> type values only.

\*/

subtype?: string | undefined;

/\*\*

\* Object class (constructor) name. Specified for <code>object</code> type values only.

\*/

className?: string | undefined;

/\*\*

\* Remote object value in case of primitive values or JSON values (if it was requested).

\*/

value?: any;

/\*\*

\* Primitive value which can not be JSON-stringified does not have <code>value</code>, but gets this property.

\*/

unserializableValue?: UnserializableValue | undefined;

/\*\*

\* String representation of the object.

\*/

description?: string | undefined;

/\*\*

\* Unique object identifier (for non-primitive values).

\*/

objectId?: RemoteObjectId | undefined;

/\*\*

\* Preview containing abbreviated property values. Specified for <code>object</code> type values only.

\* @experimental

\*/

preview?: ObjectPreview | undefined;

/\*\*

\* @experimental

\*/

customPreview?: CustomPreview | undefined;

}

/\*\*

\* @experimental

\*/

interface CustomPreview {

header: string;

hasBody: boolean;

formatterObjectId: RemoteObjectId;

bindRemoteObjectFunctionId: RemoteObjectId;

configObjectId?: RemoteObjectId | undefined;

}

/\*\*

\* Object containing abbreviated remote object value.

\* @experimental

\*/

interface ObjectPreview {

/\*\*

\* Object type.

\*/

type: string;

/\*\*

\* Object subtype hint. Specified for <code>object</code> type values only.

\*/

subtype?: string | undefined;

/\*\*

\* String representation of the object.

\*/

description?: string | undefined;

/\*\*

\* True iff some of the properties or entries of the original object did not fit.

\*/

overflow: boolean;

/\*\*

\* List of the properties.

\*/

properties: PropertyPreview[];

/\*\*

\* List of the entries. Specified for <code>map</code> and <code>set</code> subtype values only.

\*/

entries?: EntryPreview[] | undefined;

}

/\*\*

\* @experimental

\*/

interface PropertyPreview {

/\*\*

\* Property name.

\*/

name: string;

/\*\*

\* Object type. Accessor means that the property itself is an accessor property.

\*/

type: string;

/\*\*

\* User-friendly property value string.

\*/

value?: string | undefined;

/\*\*

\* Nested value preview.

\*/

valuePreview?: ObjectPreview | undefined;

/\*\*

\* Object subtype hint. Specified for <code>object</code> type values only.

\*/

subtype?: string | undefined;

}

/\*\*

\* @experimental

\*/

interface EntryPreview {

/\*\*

\* Preview of the key. Specified for map-like collection entries.

\*/

key?: ObjectPreview | undefined;

/\*\*

\* Preview of the value.

\*/

value: ObjectPreview;

}

/\*\*

\* Object property descriptor.

\*/

interface PropertyDescriptor {

/\*\*

\* Property name or symbol description.

\*/

name: string;

/\*\*

\* The value associated with the property.

\*/

value?: RemoteObject | undefined;

/\*\*

\* True if the value associated with the property may be changed (data descriptors only).

\*/

writable?: boolean | undefined;

/\*\*

\* A function which serves as a getter for the property, or <code>undefined</code> if there is no getter (accessor descriptors only).

\*/

get?: RemoteObject | undefined;

/\*\*

\* A function which serves as a setter for the property, or <code>undefined</code> if there is no setter (accessor descriptors only).

\*/

set?: RemoteObject | undefined;

/\*\*

\* True if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.

\*/

configurable: boolean;

/\*\*

\* True if this property shows up during enumeration of the properties on the corresponding object.

\*/

enumerable: boolean;

/\*\*

\* True if the result was thrown during the evaluation.

\*/

wasThrown?: boolean | undefined;

/\*\*

\* True if the property is owned for the object.

\*/

isOwn?: boolean | undefined;

/\*\*

\* Property symbol object, if the property is of the <code>symbol</code> type.

\*/

symbol?: RemoteObject | undefined;

}

/\*\*

\* Object internal property descriptor. This property isn't normally visible in JavaScript code.

\*/

interface InternalPropertyDescriptor {

/\*\*

\* Conventional property name.

\*/

name: string;

/\*\*

\* The value associated with the property.

\*/

value?: RemoteObject | undefined;

}

/\*\*

\* Represents function call argument. Either remote object id <code>objectId</code>, primitive <code>value</code>, unserializable primitive value or neither of (for undefined) them should be specified.

\*/

interface CallArgument {

/\*\*

\* Primitive value or serializable javascript object.

\*/

value?: any;

/\*\*

\* Primitive value which can not be JSON-stringified.

\*/

unserializableValue?: UnserializableValue | undefined;

/\*\*

\* Remote object handle.

\*/

objectId?: RemoteObjectId | undefined;

}

/\*\*

\* Id of an execution context.

\*/

type ExecutionContextId = number;

/\*\*

\* Description of an isolated world.

\*/

interface ExecutionContextDescription {

/\*\*

\* Unique id of the execution context. It can be used to specify in which execution context script evaluation should be performed.

\*/

id: ExecutionContextId;

/\*\*

\* Execution context origin.

\*/

origin: string;

/\*\*

\* Human readable name describing given context.

\*/

name: string;

/\*\*

\* Embedder-specific auxiliary data.

\*/

auxData?: {} | undefined;

}

/\*\*

\* Detailed information about exception (or error) that was thrown during script compilation or execution.

\*/

interface ExceptionDetails {

/\*\*

\* Exception id.

\*/

exceptionId: number;

/\*\*

\* Exception text, which should be used together with exception object when available.

\*/

text: string;

/\*\*

\* Line number of the exception location (0-based).

\*/

lineNumber: number;

/\*\*

\* Column number of the exception location (0-based).

\*/

columnNumber: number;

/\*\*

\* Script ID of the exception location.

\*/

scriptId?: ScriptId | undefined;

/\*\*

\* URL of the exception location, to be used when the script was not reported.

\*/

url?: string | undefined;

/\*\*

\* JavaScript stack trace if available.

\*/

stackTrace?: StackTrace | undefined;

/\*\*

\* Exception object if available.

\*/

exception?: RemoteObject | undefined;

/\*\*

\* Identifier of the context where exception happened.

\*/

executionContextId?: ExecutionContextId | undefined;

}

/\*\*

\* Number of milliseconds since epoch.

\*/

type Timestamp = number;

/\*\*

\* Stack entry for runtime errors and assertions.

\*/

interface CallFrame {

/\*\*

\* JavaScript function name.

\*/

functionName: string;

/\*\*

\* JavaScript script id.

\*/

scriptId: ScriptId;

/\*\*

\* JavaScript script name or url.

\*/

url: string;

/\*\*

\* JavaScript script line number (0-based).

\*/

lineNumber: number;

/\*\*

\* JavaScript script column number (0-based).

\*/

columnNumber: number;

}

/\*\*

\* Call frames for assertions or error messages.

\*/

interface StackTrace {

/\*\*

\* String label of this stack trace. For async traces this may be a name of the function that initiated the async call.

\*/

description?: string | undefined;

/\*\*

\* JavaScript function name.

\*/

callFrames: CallFrame[];

/\*\*

\* Asynchronous JavaScript stack trace that preceded this stack, if available.

\*/

parent?: StackTrace | undefined;

/\*\*

\* Asynchronous JavaScript stack trace that preceded this stack, if available.

\* @experimental

\*/

parentId?: StackTraceId | undefined;

}

/\*\*

\* Unique identifier of current debugger.

\* @experimental

\*/

type UniqueDebuggerId = string;

/\*\*

\* If <code>debuggerId</code> is set stack trace comes from another debugger and can be resolved there. This allows to track cross-debugger calls. See <code>Runtime.StackTrace</code> and <code>Debugger.paused</code> for usages.

\* @experimental

\*/

interface StackTraceId {

id: string;

debuggerId?: UniqueDebuggerId | undefined;

}

interface EvaluateParameterType {

/\*\*

\* Expression to evaluate.

\*/

expression: string;

/\*\*

\* Symbolic group name that can be used to release multiple objects.

\*/

objectGroup?: string | undefined;

/\*\*

\* Determines whether Command Line API should be available during the evaluation.

\*/

includeCommandLineAPI?: boolean | undefined;

/\*\*

\* In silent mode exceptions thrown during evaluation are not reported and do not pause execution. Overrides <code>setPauseOnException</code> state.

\*/

silent?: boolean | undefined;

/\*\*

\* Specifies in which execution context to perform evaluation. If the parameter is omitted the evaluation will be performed in the context of the inspected page.

\*/

contextId?: ExecutionContextId | undefined;

/\*\*

\* Whether the result is expected to be a JSON object that should be sent by value.

\*/

returnByValue?: boolean | undefined;

/\*\*

\* Whether preview should be generated for the result.

\* @experimental

\*/

generatePreview?: boolean | undefined;

/\*\*

\* Whether execution should be treated as initiated by user in the UI.

\*/

userGesture?: boolean | undefined;

/\*\*

\* Whether execution should <code>await</code> for resulting value and return once awaited promise is resolved.

\*/

awaitPromise?: boolean | undefined;

}

interface AwaitPromiseParameterType {

/\*\*

\* Identifier of the promise.

\*/

promiseObjectId: RemoteObjectId;

/\*\*

\* Whether the result is expected to be a JSON object that should be sent by value.

\*/

returnByValue?: boolean | undefined;

/\*\*

\* Whether preview should be generated for the result.

\*/

generatePreview?: boolean | undefined;

}

interface CallFunctionOnParameterType {

/\*\*

\* Declaration of the function to call.

\*/

functionDeclaration: string;

/\*\*

\* Identifier of the object to call function on. Either objectId or executionContextId should be specified.

\*/

objectId?: RemoteObjectId | undefined;

/\*\*

\* Call arguments. All call arguments must belong to the same JavaScript world as the target object.

\*/

arguments?: CallArgument[] | undefined;

/\*\*

\* In silent mode exceptions thrown during evaluation are not reported and do not pause execution. Overrides <code>setPauseOnException</code> state.

\*/

silent?: boolean | undefined;

/\*\*

\* Whether the result is expected to be a JSON object which should be sent by value.

\*/

returnByValue?: boolean | undefined;

/\*\*

\* Whether preview should be generated for the result.

\* @experimental

\*/

generatePreview?: boolean | undefined;

/\*\*

\* Whether execution should be treated as initiated by user in the UI.

\*/

userGesture?: boolean | undefined;

/\*\*

\* Whether execution should <code>await</code> for resulting value and return once awaited promise is resolved.

\*/

awaitPromise?: boolean | undefined;

/\*\*

\* Specifies execution context which global object will be used to call function on. Either executionContextId or objectId should be specified.

\*/

executionContextId?: ExecutionContextId | undefined;

/\*\*

\* Symbolic group name that can be used to release multiple objects. If objectGroup is not specified and objectId is, objectGroup will be inherited from object.

\*/

objectGroup?: string | undefined;

}

interface GetPropertiesParameterType {

/\*\*

\* Identifier of the object to return properties for.

\*/

objectId: RemoteObjectId;

/\*\*

\* If true, returns properties belonging only to the element itself, not to its prototype chain.

\*/

ownProperties?: boolean | undefined;

/\*\*

\* If true, returns accessor properties (with getter/setter) only; internal properties are not returned either.

\* @experimental

\*/

accessorPropertiesOnly?: boolean | undefined;

/\*\*

\* Whether preview should be generated for the results.

\* @experimental

\*/

generatePreview?: boolean | undefined;

}

interface ReleaseObjectParameterType {

/\*\*

\* Identifier of the object to release.

\*/

objectId: RemoteObjectId;

}

interface ReleaseObjectGroupParameterType {

/\*\*

\* Symbolic object group name.

\*/

objectGroup: string;

}

interface SetCustomObjectFormatterEnabledParameterType {

enabled: boolean;

}

interface CompileScriptParameterType {

/\*\*

\* Expression to compile.

\*/

expression: string;

/\*\*

\* Source url to be set for the script.

\*/

sourceURL: string;

/\*\*

\* Specifies whether the compiled script should be persisted.

\*/

persistScript: boolean;

/\*\*

\* Specifies in which execution context to perform script run. If the parameter is omitted the evaluation will be performed in the context of the inspected page.

\*/

executionContextId?: ExecutionContextId | undefined;

}

interface RunScriptParameterType {

/\*\*

\* Id of the script to run.

\*/

scriptId: ScriptId;

/\*\*

\* Specifies in which execution context to perform script run. If the parameter is omitted the evaluation will be performed in the context of the inspected page.

\*/

executionContextId?: ExecutionContextId | undefined;

/\*\*

\* Symbolic group name that can be used to release multiple objects.

\*/

objectGroup?: string | undefined;

/\*\*

\* In silent mode exceptions thrown during evaluation are not reported and do not pause execution. Overrides <code>setPauseOnException</code> state.

\*/

silent?: boolean | undefined;

/\*\*

\* Determines whether Command Line API should be available during the evaluation.

\*/

includeCommandLineAPI?: boolean | undefined;

/\*\*

\* Whether the result is expected to be a JSON object which should be sent by value.

\*/

returnByValue?: boolean | undefined;

/\*\*

\* Whether preview should be generated for the result.

\*/

generatePreview?: boolean | undefined;

/\*\*

\* Whether execution should <code>await</code> for resulting value and return once awaited promise is resolved.

\*/

awaitPromise?: boolean | undefined;

}

interface QueryObjectsParameterType {

/\*\*

\* Identifier of the prototype to return objects for.

\*/

prototypeObjectId: RemoteObjectId;

}

interface GlobalLexicalScopeNamesParameterType {

/\*\*

\* Specifies in which execution context to lookup global scope variables.

\*/

executionContextId?: ExecutionContextId | undefined;

}

interface EvaluateReturnType {

/\*\*

\* Evaluation result.

\*/

result: RemoteObject;

/\*\*

\* Exception details.

\*/

exceptionDetails?: ExceptionDetails | undefined;

}

interface AwaitPromiseReturnType {

/\*\*

\* Promise result. Will contain rejected value if promise was rejected.

\*/

result: RemoteObject;

/\*\*

\* Exception details if stack strace is available.

\*/

exceptionDetails?: ExceptionDetails | undefined;

}

interface CallFunctionOnReturnType {

/\*\*

\* Call result.

\*/

result: RemoteObject;

/\*\*

\* Exception details.

\*/

exceptionDetails?: ExceptionDetails | undefined;

}

interface GetPropertiesReturnType {

/\*\*

\* Object properties.

\*/

result: PropertyDescriptor[];

/\*\*

\* Internal object properties (only of the element itself).

\*/

internalProperties?: InternalPropertyDescriptor[] | undefined;

/\*\*

\* Exception details.

\*/

exceptionDetails?: ExceptionDetails | undefined;

}

interface CompileScriptReturnType {

/\*\*

\* Id of the script.

\*/

scriptId?: ScriptId | undefined;

/\*\*

\* Exception details.

\*/

exceptionDetails?: ExceptionDetails | undefined;

}

interface RunScriptReturnType {

/\*\*

\* Run result.

\*/

result: RemoteObject;

/\*\*

\* Exception details.

\*/

exceptionDetails?: ExceptionDetails | undefined;

}

interface QueryObjectsReturnType {

/\*\*

\* Array with objects.

\*/

objects: RemoteObject;

}

interface GlobalLexicalScopeNamesReturnType {

names: string[];

}

interface ExecutionContextCreatedEventDataType {

/\*\*

\* A newly created execution context.

\*/

context: ExecutionContextDescription;

}

interface ExecutionContextDestroyedEventDataType {

/\*\*

\* Id of the destroyed context

\*/

executionContextId: ExecutionContextId;

}

interface ExceptionThrownEventDataType {

/\*\*

\* Timestamp of the exception.

\*/

timestamp: Timestamp;

exceptionDetails: ExceptionDetails;

}

interface ExceptionRevokedEventDataType {

/\*\*

\* Reason describing why exception was revoked.

\*/

reason: string;

/\*\*

\* The id of revoked exception, as reported in <code>exceptionThrown</code>.

\*/

exceptionId: number;

}

interface ConsoleAPICalledEventDataType {

/\*\*

\* Type of the call.

\*/

type: string;

/\*\*

\* Call arguments.

\*/

args: RemoteObject[];

/\*\*

\* Identifier of the context where the call was made.

\*/

executionContextId: ExecutionContextId;

/\*\*

\* Call timestamp.

\*/

timestamp: Timestamp;

/\*\*

\* Stack trace captured when the call was made.

\*/

stackTrace?: StackTrace | undefined;

/\*\*

\* Console context descriptor for calls on non-default console context (not console.\*): 'anonymous#unique-logger-id' for call on unnamed context, 'name#unique-logger-id' for call on named context.

\* @experimental

\*/

context?: string | undefined;

}

interface InspectRequestedEventDataType {

object: RemoteObject;

hints: {};

}

}

namespace Debugger {

/\*\*

\* Breakpoint identifier.

\*/

type BreakpointId = string;

/\*\*

\* Call frame identifier.

\*/

type CallFrameId = string;

/\*\*

\* Location in the source code.

\*/

interface Location {

/\*\*

\* Script identifier as reported in the <code>Debugger.scriptParsed</code>.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* Line number in the script (0-based).

\*/

lineNumber: number;

/\*\*

\* Column number in the script (0-based).

\*/

columnNumber?: number | undefined;

}

/\*\*

\* Location in the source code.

\* @experimental

\*/

interface ScriptPosition {

lineNumber: number;

columnNumber: number;

}

/\*\*

\* JavaScript call frame. Array of call frames form the call stack.

\*/

interface CallFrame {

/\*\*

\* Call frame identifier. This identifier is only valid while the virtual machine is paused.

\*/

callFrameId: CallFrameId;

/\*\*

\* Name of the JavaScript function called on this call frame.

\*/

functionName: string;

/\*\*

\* Location in the source code.

\*/

functionLocation?: Location | undefined;

/\*\*

\* Location in the source code.

\*/

location: Location;

/\*\*

\* JavaScript script name or url.

\*/

url: string;

/\*\*

\* Scope chain for this call frame.

\*/

scopeChain: Scope[];

/\*\*

\* <code>this</code> object for this call frame.

\*/

this: Runtime.RemoteObject;

/\*\*

\* The value being returned, if the function is at return point.

\*/

returnValue?: Runtime.RemoteObject | undefined;

}

/\*\*

\* Scope description.

\*/

interface Scope {

/\*\*

\* Scope type.

\*/

type: string;

/\*\*

\* Object representing the scope. For <code>global</code> and <code>with</code> scopes it represents the actual object; for the rest of the scopes, it is artificial transient object enumerating scope variables as its properties.

\*/

object: Runtime.RemoteObject;

name?: string | undefined;

/\*\*

\* Location in the source code where scope starts

\*/

startLocation?: Location | undefined;

/\*\*

\* Location in the source code where scope ends

\*/

endLocation?: Location | undefined;

}

/\*\*

\* Search match for resource.

\*/

interface SearchMatch {

/\*\*

\* Line number in resource content.

\*/

lineNumber: number;

/\*\*

\* Line with match content.

\*/

lineContent: string;

}

interface BreakLocation {

/\*\*

\* Script identifier as reported in the <code>Debugger.scriptParsed</code>.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* Line number in the script (0-based).

\*/

lineNumber: number;

/\*\*

\* Column number in the script (0-based).

\*/

columnNumber?: number | undefined;

type?: string | undefined;

}

interface SetBreakpointsActiveParameterType {

/\*\*

\* New value for breakpoints active state.

\*/

active: boolean;

}

interface SetSkipAllPausesParameterType {

/\*\*

\* New value for skip pauses state.

\*/

skip: boolean;

}

interface SetBreakpointByUrlParameterType {

/\*\*

\* Line number to set breakpoint at.

\*/

lineNumber: number;

/\*\*

\* URL of the resources to set breakpoint on.

\*/

url?: string | undefined;

/\*\*

\* Regex pattern for the URLs of the resources to set breakpoints on. Either <code>url</code> or <code>urlRegex</code> must be specified.

\*/

urlRegex?: string | undefined;

/\*\*

\* Script hash of the resources to set breakpoint on.

\*/

scriptHash?: string | undefined;

/\*\*

\* Offset in the line to set breakpoint at.

\*/

columnNumber?: number | undefined;

/\*\*

\* Expression to use as a breakpoint condition. When specified, debugger will only stop on the breakpoint if this expression evaluates to true.

\*/

condition?: string | undefined;

}

interface SetBreakpointParameterType {

/\*\*

\* Location to set breakpoint in.

\*/

location: Location;

/\*\*

\* Expression to use as a breakpoint condition. When specified, debugger will only stop on the breakpoint if this expression evaluates to true.

\*/

condition?: string | undefined;

}

interface RemoveBreakpointParameterType {

breakpointId: BreakpointId;

}

interface GetPossibleBreakpointsParameterType {

/\*\*

\* Start of range to search possible breakpoint locations in.

\*/

start: Location;

/\*\*

\* End of range to search possible breakpoint locations in (excluding). When not specified, end of scripts is used as end of range.

\*/

end?: Location | undefined;

/\*\*

\* Only consider locations which are in the same (non-nested) function as start.

\*/

restrictToFunction?: boolean | undefined;

}

interface ContinueToLocationParameterType {

/\*\*

\* Location to continue to.

\*/

location: Location;

targetCallFrames?: string | undefined;

}

interface PauseOnAsyncCallParameterType {

/\*\*

\* Debugger will pause when async call with given stack trace is started.

\*/

parentStackTraceId: Runtime.StackTraceId;

}

interface StepIntoParameterType {

/\*\*

\* Debugger will issue additional Debugger.paused notification if any async task is scheduled before next pause.

\* @experimental

\*/

breakOnAsyncCall?: boolean | undefined;

}

interface GetStackTraceParameterType {

stackTraceId: Runtime.StackTraceId;

}

interface SearchInContentParameterType {

/\*\*

\* Id of the script to search in.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* String to search for.

\*/

query: string;

/\*\*

\* If true, search is case sensitive.

\*/

caseSensitive?: boolean | undefined;

/\*\*

\* If true, treats string parameter as regex.

\*/

isRegex?: boolean | undefined;

}

interface SetScriptSourceParameterType {

/\*\*

\* Id of the script to edit.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* New content of the script.

\*/

scriptSource: string;

/\*\*

\* If true the change will not actually be applied. Dry run may be used to get result description without actually modifying the code.

\*/

dryRun?: boolean | undefined;

}

interface RestartFrameParameterType {

/\*\*

\* Call frame identifier to evaluate on.

\*/

callFrameId: CallFrameId;

}

interface GetScriptSourceParameterType {

/\*\*

\* Id of the script to get source for.

\*/

scriptId: Runtime.ScriptId;

}

interface SetPauseOnExceptionsParameterType {

/\*\*

\* Pause on exceptions mode.

\*/

state: string;

}

interface EvaluateOnCallFrameParameterType {

/\*\*

\* Call frame identifier to evaluate on.

\*/

callFrameId: CallFrameId;

/\*\*

\* Expression to evaluate.

\*/

expression: string;

/\*\*

\* String object group name to put result into (allows rapid releasing resulting object handles using <code>releaseObjectGroup</code>).

\*/

objectGroup?: string | undefined;

/\*\*

\* Specifies whether command line API should be available to the evaluated expression, defaults to false.

\*/

includeCommandLineAPI?: boolean | undefined;

/\*\*

\* In silent mode exceptions thrown during evaluation are not reported and do not pause execution. Overrides <code>setPauseOnException</code> state.

\*/

silent?: boolean | undefined;

/\*\*

\* Whether the result is expected to be a JSON object that should be sent by value.

\*/

returnByValue?: boolean | undefined;

/\*\*

\* Whether preview should be generated for the result.

\* @experimental

\*/

generatePreview?: boolean | undefined;

/\*\*

\* Whether to throw an exception if side effect cannot be ruled out during evaluation.

\*/

throwOnSideEffect?: boolean | undefined;

}

interface SetVariableValueParameterType {

/\*\*

\* 0-based number of scope as was listed in scope chain. Only 'local', 'closure' and 'catch' scope types are allowed. Other scopes could be manipulated manually.

\*/

scopeNumber: number;

/\*\*

\* Variable name.

\*/

variableName: string;

/\*\*

\* New variable value.

\*/

newValue: Runtime.CallArgument;

/\*\*

\* Id of callframe that holds variable.

\*/

callFrameId: CallFrameId;

}

interface SetReturnValueParameterType {

/\*\*

\* New return value.

\*/

newValue: Runtime.CallArgument;

}

interface SetAsyncCallStackDepthParameterType {

/\*\*

\* Maximum depth of async call stacks. Setting to <code>0</code> will effectively disable collecting async call stacks (default).

\*/

maxDepth: number;

}

interface SetBlackboxPatternsParameterType {

/\*\*

\* Array of regexps that will be used to check script url for blackbox state.

\*/

patterns: string[];

}

interface SetBlackboxedRangesParameterType {

/\*\*

\* Id of the script.

\*/

scriptId: Runtime.ScriptId;

positions: ScriptPosition[];

}

interface EnableReturnType {

/\*\*

\* Unique identifier of the debugger.

\* @experimental

\*/

debuggerId: Runtime.UniqueDebuggerId;

}

interface SetBreakpointByUrlReturnType {

/\*\*

\* Id of the created breakpoint for further reference.

\*/

breakpointId: BreakpointId;

/\*\*

\* List of the locations this breakpoint resolved into upon addition.

\*/

locations: Location[];

}

interface SetBreakpointReturnType {

/\*\*

\* Id of the created breakpoint for further reference.

\*/

breakpointId: BreakpointId;

/\*\*

\* Location this breakpoint resolved into.

\*/

actualLocation: Location;

}

interface GetPossibleBreakpointsReturnType {

/\*\*

\* List of the possible breakpoint locations.

\*/

locations: BreakLocation[];

}

interface GetStackTraceReturnType {

stackTrace: Runtime.StackTrace;

}

interface SearchInContentReturnType {

/\*\*

\* List of search matches.

\*/

result: SearchMatch[];

}

interface SetScriptSourceReturnType {

/\*\*

\* New stack trace in case editing has happened while VM was stopped.

\*/

callFrames?: CallFrame[] | undefined;

/\*\*

\* Whether current call stack was modified after applying the changes.

\*/

stackChanged?: boolean | undefined;

/\*\*

\* Async stack trace, if any.

\*/

asyncStackTrace?: Runtime.StackTrace | undefined;

/\*\*

\* Async stack trace, if any.

\* @experimental

\*/

asyncStackTraceId?: Runtime.StackTraceId | undefined;

/\*\*

\* Exception details if any.

\*/

exceptionDetails?: Runtime.ExceptionDetails | undefined;

}

interface RestartFrameReturnType {

/\*\*

\* New stack trace.

\*/

callFrames: CallFrame[];

/\*\*

\* Async stack trace, if any.

\*/

asyncStackTrace?: Runtime.StackTrace | undefined;

/\*\*

\* Async stack trace, if any.

\* @experimental

\*/

asyncStackTraceId?: Runtime.StackTraceId | undefined;

}

interface GetScriptSourceReturnType {

/\*\*

\* Script source.

\*/

scriptSource: string;

}

interface EvaluateOnCallFrameReturnType {

/\*\*

\* Object wrapper for the evaluation result.

\*/

result: Runtime.RemoteObject;

/\*\*

\* Exception details.

\*/

exceptionDetails?: Runtime.ExceptionDetails | undefined;

}

interface ScriptParsedEventDataType {

/\*\*

\* Identifier of the script parsed.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* URL or name of the script parsed (if any).

\*/

url: string;

/\*\*

\* Line offset of the script within the resource with given URL (for script tags).

\*/

startLine: number;

/\*\*

\* Column offset of the script within the resource with given URL.

\*/

startColumn: number;

/\*\*

\* Last line of the script.

\*/

endLine: number;

/\*\*

\* Length of the last line of the script.

\*/

endColumn: number;

/\*\*

\* Specifies script creation context.

\*/

executionContextId: Runtime.ExecutionContextId;

/\*\*

\* Content hash of the script.

\*/

hash: string;

/\*\*

\* Embedder-specific auxiliary data.

\*/

executionContextAuxData?: {} | undefined;

/\*\*

\* True, if this script is generated as a result of the live edit operation.

\* @experimental

\*/

isLiveEdit?: boolean | undefined;

/\*\*

\* URL of source map associated with script (if any).

\*/

sourceMapURL?: string | undefined;

/\*\*

\* True, if this script has sourceURL.

\*/

hasSourceURL?: boolean | undefined;

/\*\*

\* True, if this script is ES6 module.

\*/

isModule?: boolean | undefined;

/\*\*

\* This script length.

\*/

length?: number | undefined;

/\*\*

\* JavaScript top stack frame of where the script parsed event was triggered if available.

\* @experimental

\*/

stackTrace?: Runtime.StackTrace | undefined;

}

interface ScriptFailedToParseEventDataType {

/\*\*

\* Identifier of the script parsed.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* URL or name of the script parsed (if any).

\*/

url: string;

/\*\*

\* Line offset of the script within the resource with given URL (for script tags).

\*/

startLine: number;

/\*\*

\* Column offset of the script within the resource with given URL.

\*/

startColumn: number;

/\*\*

\* Last line of the script.

\*/

endLine: number;

/\*\*

\* Length of the last line of the script.

\*/

endColumn: number;

/\*\*

\* Specifies script creation context.

\*/

executionContextId: Runtime.ExecutionContextId;

/\*\*

\* Content hash of the script.

\*/

hash: string;

/\*\*

\* Embedder-specific auxiliary data.

\*/

executionContextAuxData?: {} | undefined;

/\*\*

\* URL of source map associated with script (if any).

\*/

sourceMapURL?: string | undefined;

/\*\*

\* True, if this script has sourceURL.

\*/

hasSourceURL?: boolean | undefined;

/\*\*

\* True, if this script is ES6 module.

\*/

isModule?: boolean | undefined;

/\*\*

\* This script length.

\*/

length?: number | undefined;

/\*\*

\* JavaScript top stack frame of where the script parsed event was triggered if available.

\* @experimental

\*/

stackTrace?: Runtime.StackTrace | undefined;

}

interface BreakpointResolvedEventDataType {

/\*\*

\* Breakpoint unique identifier.

\*/

breakpointId: BreakpointId;

/\*\*

\* Actual breakpoint location.

\*/

location: Location;

}

interface PausedEventDataType {

/\*\*

\* Call stack the virtual machine stopped on.

\*/

callFrames: CallFrame[];

/\*\*

\* Pause reason.

\*/

reason: string;

/\*\*

\* Object containing break-specific auxiliary properties.

\*/

data?: {} | undefined;

/\*\*

\* Hit breakpoints IDs

\*/

hitBreakpoints?: string[] | undefined;

/\*\*

\* Async stack trace, if any.

\*/

asyncStackTrace?: Runtime.StackTrace | undefined;

/\*\*

\* Async stack trace, if any.

\* @experimental

\*/

asyncStackTraceId?: Runtime.StackTraceId | undefined;

/\*\*

\* Just scheduled async call will have this stack trace as parent stack during async execution. This field is available only after <code>Debugger.stepInto</code> call with <code>breakOnAsynCall</code> flag.

\* @experimental

\*/

asyncCallStackTraceId?: Runtime.StackTraceId | undefined;

}

}

namespace Console {

/\*\*

\* Console message.

\*/

interface ConsoleMessage {

/\*\*

\* Message source.

\*/

source: string;

/\*\*

\* Message severity.

\*/

level: string;

/\*\*

\* Message text.

\*/

text: string;

/\*\*

\* URL of the message origin.

\*/

url?: string | undefined;

/\*\*

\* Line number in the resource that generated this message (1-based).

\*/

line?: number | undefined;

/\*\*

\* Column number in the resource that generated this message (1-based).

\*/

column?: number | undefined;

}

interface MessageAddedEventDataType {

/\*\*

\* Console message that has been added.

\*/

message: ConsoleMessage;

}

}

namespace Profiler {

/\*\*

\* Profile node. Holds callsite information, execution statistics and child nodes.

\*/

interface ProfileNode {

/\*\*

\* Unique id of the node.

\*/

id: number;

/\*\*

\* Function location.

\*/

callFrame: Runtime.CallFrame;

/\*\*

\* Number of samples where this node was on top of the call stack.

\*/

hitCount?: number | undefined;

/\*\*

\* Child node ids.

\*/

children?: number[] | undefined;

/\*\*

\* The reason of being not optimized. The function may be deoptimized or marked as don't optimize.

\*/

deoptReason?: string | undefined;

/\*\*

\* An array of source position ticks.

\*/

positionTicks?: PositionTickInfo[] | undefined;

}

/\*\*

\* Profile.

\*/

interface Profile {

/\*\*

\* The list of profile nodes. First item is the root node.

\*/

nodes: ProfileNode[];

/\*\*

\* Profiling start timestamp in microseconds.

\*/

startTime: number;

/\*\*

\* Profiling end timestamp in microseconds.

\*/

endTime: number;

/\*\*

\* Ids of samples top nodes.

\*/

samples?: number[] | undefined;

/\*\*

\* Time intervals between adjacent samples in microseconds. The first delta is relative to the profile startTime.

\*/

timeDeltas?: number[] | undefined;

}

/\*\*

\* Specifies a number of samples attributed to a certain source position.

\*/

interface PositionTickInfo {

/\*\*

\* Source line number (1-based).

\*/

line: number;

/\*\*

\* Number of samples attributed to the source line.

\*/

ticks: number;

}

/\*\*

\* Coverage data for a source range.

\*/

interface CoverageRange {

/\*\*

\* JavaScript script source offset for the range start.

\*/

startOffset: number;

/\*\*

\* JavaScript script source offset for the range end.

\*/

endOffset: number;

/\*\*

\* Collected execution count of the source range.

\*/

count: number;

}

/\*\*

\* Coverage data for a JavaScript function.

\*/

interface FunctionCoverage {

/\*\*

\* JavaScript function name.

\*/

functionName: string;

/\*\*

\* Source ranges inside the function with coverage data.

\*/

ranges: CoverageRange[];

/\*\*

\* Whether coverage data for this function has block granularity.

\*/

isBlockCoverage: boolean;

}

/\*\*

\* Coverage data for a JavaScript script.

\*/

interface ScriptCoverage {

/\*\*

\* JavaScript script id.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* JavaScript script name or url.

\*/

url: string;

/\*\*

\* Functions contained in the script that has coverage data.

\*/

functions: FunctionCoverage[];

}

/\*\*

\* Describes a type collected during runtime.

\* @experimental

\*/

interface TypeObject {

/\*\*

\* Name of a type collected with type profiling.

\*/

name: string;

}

/\*\*

\* Source offset and types for a parameter or return value.

\* @experimental

\*/

interface TypeProfileEntry {

/\*\*

\* Source offset of the parameter or end of function for return values.

\*/

offset: number;

/\*\*

\* The types for this parameter or return value.

\*/

types: TypeObject[];

}

/\*\*

\* Type profile data collected during runtime for a JavaScript script.

\* @experimental

\*/

interface ScriptTypeProfile {

/\*\*

\* JavaScript script id.

\*/

scriptId: Runtime.ScriptId;

/\*\*

\* JavaScript script name or url.

\*/

url: string;

/\*\*

\* Type profile entries for parameters and return values of the functions in the script.

\*/

entries: TypeProfileEntry[];

}

interface SetSamplingIntervalParameterType {

/\*\*

\* New sampling interval in microseconds.

\*/

interval: number;

}

interface StartPreciseCoverageParameterType {

/\*\*

\* Collect accurate call counts beyond simple 'covered' or 'not covered'.

\*/

callCount?: boolean | undefined;

/\*\*

\* Collect block-based coverage.

\*/

detailed?: boolean | undefined;

}

interface StopReturnType {

/\*\*

\* Recorded profile.

\*/

profile: Profile;

}

interface TakePreciseCoverageReturnType {

/\*\*

\* Coverage data for the current isolate.

\*/

result: ScriptCoverage[];

}

interface GetBestEffortCoverageReturnType {

/\*\*

\* Coverage data for the current isolate.

\*/

result: ScriptCoverage[];

}

interface TakeTypeProfileReturnType {

/\*\*

\* Type profile for all scripts since startTypeProfile() was turned on.

\*/

result: ScriptTypeProfile[];

}

interface ConsoleProfileStartedEventDataType {

id: string;

/\*\*

\* Location of console.profile().

\*/

location: Debugger.Location;

/\*\*

\* Profile title passed as an argument to console.profile().

\*/

title?: string | undefined;

}

interface ConsoleProfileFinishedEventDataType {

id: string;

/\*\*

\* Location of console.profileEnd().

\*/

location: Debugger.Location;

profile: Profile;

/\*\*

\* Profile title passed as an argument to console.profile().

\*/

title?: string | undefined;

}

}

namespace HeapProfiler {

/\*\*

\* Heap snapshot object id.

\*/

type HeapSnapshotObjectId = string;

/\*\*

\* Sampling Heap Profile node. Holds callsite information, allocation statistics and child nodes.

\*/

interface SamplingHeapProfileNode {

/\*\*

\* Function location.

\*/

callFrame: Runtime.CallFrame;

/\*\*

\* Allocations size in bytes for the node excluding children.

\*/

selfSize: number;

/\*\*

\* Child nodes.

\*/

children: SamplingHeapProfileNode[];

}

/\*\*

\* Profile.

\*/

interface SamplingHeapProfile {

head: SamplingHeapProfileNode;

}

interface StartTrackingHeapObjectsParameterType {

trackAllocations?: boolean | undefined;

}

interface StopTrackingHeapObjectsParameterType {

/\*\*

\* If true 'reportHeapSnapshotProgress' events will be generated while snapshot is being taken when the tracking is stopped.

\*/

reportProgress?: boolean | undefined;

}

interface TakeHeapSnapshotParameterType {

/\*\*

\* If true 'reportHeapSnapshotProgress' events will be generated while snapshot is being taken.

\*/

reportProgress?: boolean | undefined;

}

interface GetObjectByHeapObjectIdParameterType {

objectId: HeapSnapshotObjectId;

/\*\*

\* Symbolic group name that can be used to release multiple objects.

\*/

objectGroup?: string | undefined;

}

interface AddInspectedHeapObjectParameterType {

/\*\*

\* Heap snapshot object id to be accessible by means of $x command line API.

\*/

heapObjectId: HeapSnapshotObjectId;

}

interface GetHeapObjectIdParameterType {

/\*\*

\* Identifier of the object to get heap object id for.

\*/

objectId: Runtime.RemoteObjectId;

}

interface StartSamplingParameterType {

/\*\*

\* Average sample interval in bytes. Poisson distribution is used for the intervals. The default value is 32768 bytes.

\*/

samplingInterval?: number | undefined;

}

interface GetObjectByHeapObjectIdReturnType {

/\*\*

\* Evaluation result.

\*/

result: Runtime.RemoteObject;

}

interface GetHeapObjectIdReturnType {

/\*\*

\* Id of the heap snapshot object corresponding to the passed remote object id.

\*/

heapSnapshotObjectId: HeapSnapshotObjectId;

}

interface StopSamplingReturnType {

/\*\*

\* Recorded sampling heap profile.

\*/

profile: SamplingHeapProfile;

}

interface GetSamplingProfileReturnType {

/\*\*

\* Return the sampling profile being collected.

\*/

profile: SamplingHeapProfile;

}

interface AddHeapSnapshotChunkEventDataType {

chunk: string;

}

interface ReportHeapSnapshotProgressEventDataType {

done: number;

total: number;

finished?: boolean | undefined;

}

interface LastSeenObjectIdEventDataType {

lastSeenObjectId: number;

timestamp: number;

}

interface HeapStatsUpdateEventDataType {

/\*\*

\* An array of triplets. Each triplet describes a fragment. The first integer is the fragment index, the second integer is a total count of objects for the fragment, the third integer is a total size of the objects for the fragment.

\*/

statsUpdate: number[];

}

}

namespace NodeTracing {

interface TraceConfig {

/\*\*

\* Controls how the trace buffer stores data.

\*/

recordMode?: string;

/\*\*

\* Included category filters.

\*/

includedCategories: string[];

}

interface StartParameterType {

traceConfig: TraceConfig;

}

interface GetCategoriesReturnType {

/\*\*

\* A list of supported tracing categories.

\*/

categories: string[];

}

interface DataCollectedEventDataType {

value: Array<{}>;

}

}

namespace NodeWorker {

type WorkerID = string;

/\*\*

\* Unique identifier of attached debugging session.

\*/

type SessionID = string;

interface WorkerInfo {

workerId: WorkerID;

type: string;

title: string;

url: string;

}

interface SendMessageToWorkerParameterType {

message: string;

/\*\*

\* Identifier of the session.

\*/

sessionId: SessionID;

}

interface EnableParameterType {

/\*\*

\* Whether to new workers should be paused until the frontend sends `Runtime.runIfWaitingForDebugger`

\* message to run them.

\*/

waitForDebuggerOnStart: boolean;

}

interface DetachParameterType {

sessionId: SessionID;

}

interface AttachedToWorkerEventDataType {

/\*\*

\* Identifier assigned to the session used to send/receive messages.

\*/

sessionId: SessionID;

workerInfo: WorkerInfo;

waitingForDebugger: boolean;

}

interface DetachedFromWorkerEventDataType {

/\*\*

\* Detached session identifier.

\*/

sessionId: SessionID;

}

interface ReceivedMessageFromWorkerEventDataType {

/\*\*

\* Identifier of a session which sends a message.

\*/

sessionId: SessionID;

message: string;

}

}

namespace NodeRuntime {

interface NotifyWhenWaitingForDisconnectParameterType {

enabled: boolean;

}

}

/\*\*

\* The `inspector.Session` is used for dispatching messages to the V8 inspector

\* back-end and receiving message responses and notifications.

\*/

class Session extends EventEmitter {

/\*\*

\* Create a new instance of the inspector.Session class.

\* The inspector session needs to be connected through session.connect() before the messages can be dispatched to the inspector backend.

\*/

constructor();

/\*\*

\* Connects a session to the inspector back-end.

\* @since v8.0.0

\*/

connect(): void;

/\*\*

\* Connects a session to the main thread inspector back-end. An exception will

\* be thrown if this API was not called on a Worker thread.

\* @since v12.11.0

\*/

connectToMainThread(): void;

/\*\*

\* Immediately close the session. All pending message callbacks will be called

\* with an error. `session.connect()` will need to be called to be able to send

\* messages again. Reconnected session will lose all inspector state, such as

\* enabled agents or configured breakpoints.

\* @since v8.0.0

\*/

disconnect(): void;

/\*\*

\* Posts a message to the inspector back-end. `callback` will be notified when

\* a response is received. `callback` is a function that accepts two optional

\* arguments: error and message-specific result.

\*

\* ```js

\* session.post('Runtime.evaluate', { expression: '2 + 2' },

\* (error, { result }) => console.log(result));

\* // Output: { type: 'number', value: 4, description: '4' }

\* ```

\*

\* The latest version of the V8 inspector protocol is published on the [Chrome DevTools Protocol Viewer](https://chromedevtools.github.io/devtools-protocol/v8/).

\*

\* Node.js inspector supports all the Chrome DevTools Protocol domains declared

\* by V8\. Chrome DevTools Protocol domain provides an interface for interacting

\* with one of the runtime agents used to inspect the application state and listen

\* to the run-time events.

\*

\* ## Example usage

\*

\* Apart from the debugger, various V8 Profilers are available through the DevTools

\* protocol.

\* @since v8.0.0

\*/

post(method: string, params?: {}, callback?: (err: Error | null, params?: {}) => void): void;

post(method: string, callback?: (err: Error | null, params?: {}) => void): void;

/\*\*

\* Returns supported domains.

\*/

post(method: 'Schema.getDomains', callback?: (err: Error | null, params: Schema.GetDomainsReturnType) => void): void;

/\*\*

\* Evaluates expression on global object.

\*/

post(method: 'Runtime.evaluate', params?: Runtime.EvaluateParameterType, callback?: (err: Error | null, params: Runtime.EvaluateReturnType) => void): void;

post(method: 'Runtime.evaluate', callback?: (err: Error | null, params: Runtime.EvaluateReturnType) => void): void;

/\*\*

\* Add handler to promise with given promise object id.

\*/

post(method: 'Runtime.awaitPromise', params?: Runtime.AwaitPromiseParameterType, callback?: (err: Error | null, params: Runtime.AwaitPromiseReturnType) => void): void;

post(method: 'Runtime.awaitPromise', callback?: (err: Error | null, params: Runtime.AwaitPromiseReturnType) => void): void;

/\*\*

\* Calls function with given declaration on the given object. Object group of the result is inherited from the target object.

\*/

post(method: 'Runtime.callFunctionOn', params?: Runtime.CallFunctionOnParameterType, callback?: (err: Error | null, params: Runtime.CallFunctionOnReturnType) => void): void;

post(method: 'Runtime.callFunctionOn', callback?: (err: Error | null, params: Runtime.CallFunctionOnReturnType) => void): void;

/\*\*

\* Returns properties of a given object. Object group of the result is inherited from the target object.

\*/

post(method: 'Runtime.getProperties', params?: Runtime.GetPropertiesParameterType, callback?: (err: Error | null, params: Runtime.GetPropertiesReturnType) => void): void;

post(method: 'Runtime.getProperties', callback?: (err: Error | null, params: Runtime.GetPropertiesReturnType) => void): void;

/\*\*

\* Releases remote object with given id.

\*/

post(method: 'Runtime.releaseObject', params?: Runtime.ReleaseObjectParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Runtime.releaseObject', callback?: (err: Error | null) => void): void;

/\*\*

\* Releases all remote objects that belong to a given group.

\*/

post(method: 'Runtime.releaseObjectGroup', params?: Runtime.ReleaseObjectGroupParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Runtime.releaseObjectGroup', callback?: (err: Error | null) => void): void;

/\*\*

\* Tells inspected instance to run if it was waiting for debugger to attach.

\*/

post(method: 'Runtime.runIfWaitingForDebugger', callback?: (err: Error | null) => void): void;

/\*\*

\* Enables reporting of execution contexts creation by means of <code>executionContextCreated</code> event. When the reporting gets enabled the event will be sent immediately for each existing execution context.

\*/

post(method: 'Runtime.enable', callback?: (err: Error | null) => void): void;

/\*\*

\* Disables reporting of execution contexts creation.

\*/

post(method: 'Runtime.disable', callback?: (err: Error | null) => void): void;

/\*\*

\* Discards collected exceptions and console API calls.

\*/

post(method: 'Runtime.discardConsoleEntries', callback?: (err: Error | null) => void): void;

/\*\*

\* @experimental

\*/

post(method: 'Runtime.setCustomObjectFormatterEnabled', params?: Runtime.SetCustomObjectFormatterEnabledParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Runtime.setCustomObjectFormatterEnabled', callback?: (err: Error | null) => void): void;

/\*\*

\* Compiles expression.

\*/

post(method: 'Runtime.compileScript', params?: Runtime.CompileScriptParameterType, callback?: (err: Error | null, params: Runtime.CompileScriptReturnType) => void): void;

post(method: 'Runtime.compileScript', callback?: (err: Error | null, params: Runtime.CompileScriptReturnType) => void): void;

/\*\*

\* Runs script with given id in a given context.

\*/

post(method: 'Runtime.runScript', params?: Runtime.RunScriptParameterType, callback?: (err: Error | null, params: Runtime.RunScriptReturnType) => void): void;

post(method: 'Runtime.runScript', callback?: (err: Error | null, params: Runtime.RunScriptReturnType) => void): void;

post(method: 'Runtime.queryObjects', params?: Runtime.QueryObjectsParameterType, callback?: (err: Error | null, params: Runtime.QueryObjectsReturnType) => void): void;

post(method: 'Runtime.queryObjects', callback?: (err: Error | null, params: Runtime.QueryObjectsReturnType) => void): void;

/\*\*

\* Returns all let, const and class variables from global scope.

\*/

post(

method: 'Runtime.globalLexicalScopeNames',

params?: Runtime.GlobalLexicalScopeNamesParameterType,

callback?: (err: Error | null, params: Runtime.GlobalLexicalScopeNamesReturnType) => void

): void;

post(method: 'Runtime.globalLexicalScopeNames', callback?: (err: Error | null, params: Runtime.GlobalLexicalScopeNamesReturnType) => void): void;

/\*\*

\* Enables debugger for the given page. Clients should not assume that the debugging has been enabled until the result for this command is received.

\*/

post(method: 'Debugger.enable', callback?: (err: Error | null, params: Debugger.EnableReturnType) => void): void;

/\*\*

\* Disables debugger for given page.

\*/

post(method: 'Debugger.disable', callback?: (err: Error | null) => void): void;

/\*\*

\* Activates / deactivates all breakpoints on the page.

\*/

post(method: 'Debugger.setBreakpointsActive', params?: Debugger.SetBreakpointsActiveParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setBreakpointsActive', callback?: (err: Error | null) => void): void;

/\*\*

\* Makes page not interrupt on any pauses (breakpoint, exception, dom exception etc).

\*/

post(method: 'Debugger.setSkipAllPauses', params?: Debugger.SetSkipAllPausesParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setSkipAllPauses', callback?: (err: Error | null) => void): void;

/\*\*

\* Sets JavaScript breakpoint at given location specified either by URL or URL regex. Once this command is issued, all existing parsed scripts will have breakpoints resolved and returned in <code>locations</code> property. Further matching script parsing will result in subsequent <code>breakpointResolved</code> events issued. This logical breakpoint will survive page reloads.

\*/

post(method: 'Debugger.setBreakpointByUrl', params?: Debugger.SetBreakpointByUrlParameterType, callback?: (err: Error | null, params: Debugger.SetBreakpointByUrlReturnType) => void): void;

post(method: 'Debugger.setBreakpointByUrl', callback?: (err: Error | null, params: Debugger.SetBreakpointByUrlReturnType) => void): void;

/\*\*

\* Sets JavaScript breakpoint at a given location.

\*/

post(method: 'Debugger.setBreakpoint', params?: Debugger.SetBreakpointParameterType, callback?: (err: Error | null, params: Debugger.SetBreakpointReturnType) => void): void;

post(method: 'Debugger.setBreakpoint', callback?: (err: Error | null, params: Debugger.SetBreakpointReturnType) => void): void;

/\*\*

\* Removes JavaScript breakpoint.

\*/

post(method: 'Debugger.removeBreakpoint', params?: Debugger.RemoveBreakpointParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.removeBreakpoint', callback?: (err: Error | null) => void): void;

/\*\*

\* Returns possible locations for breakpoint. scriptId in start and end range locations should be the same.

\*/

post(

method: 'Debugger.getPossibleBreakpoints',

params?: Debugger.GetPossibleBreakpointsParameterType,

callback?: (err: Error | null, params: Debugger.GetPossibleBreakpointsReturnType) => void

): void;

post(method: 'Debugger.getPossibleBreakpoints', callback?: (err: Error | null, params: Debugger.GetPossibleBreakpointsReturnType) => void): void;

/\*\*

\* Continues execution until specific location is reached.

\*/

post(method: 'Debugger.continueToLocation', params?: Debugger.ContinueToLocationParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.continueToLocation', callback?: (err: Error | null) => void): void;

/\*\*

\* @experimental

\*/

post(method: 'Debugger.pauseOnAsyncCall', params?: Debugger.PauseOnAsyncCallParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.pauseOnAsyncCall', callback?: (err: Error | null) => void): void;

/\*\*

\* Steps over the statement.

\*/

post(method: 'Debugger.stepOver', callback?: (err: Error | null) => void): void;

/\*\*

\* Steps into the function call.

\*/

post(method: 'Debugger.stepInto', params?: Debugger.StepIntoParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.stepInto', callback?: (err: Error | null) => void): void;

/\*\*

\* Steps out of the function call.

\*/

post(method: 'Debugger.stepOut', callback?: (err: Error | null) => void): void;

/\*\*

\* Stops on the next JavaScript statement.

\*/

post(method: 'Debugger.pause', callback?: (err: Error | null) => void): void;

/\*\*

\* This method is deprecated - use Debugger.stepInto with breakOnAsyncCall and Debugger.pauseOnAsyncTask instead. Steps into next scheduled async task if any is scheduled before next pause. Returns success when async task is actually scheduled, returns error if no task were scheduled or another scheduleStepIntoAsync was called.

\* @experimental

\*/

post(method: 'Debugger.scheduleStepIntoAsync', callback?: (err: Error | null) => void): void;

/\*\*

\* Resumes JavaScript execution.

\*/

post(method: 'Debugger.resume', callback?: (err: Error | null) => void): void;

/\*\*

\* Returns stack trace with given <code>stackTraceId</code>.

\* @experimental

\*/

post(method: 'Debugger.getStackTrace', params?: Debugger.GetStackTraceParameterType, callback?: (err: Error | null, params: Debugger.GetStackTraceReturnType) => void): void;

post(method: 'Debugger.getStackTrace', callback?: (err: Error | null, params: Debugger.GetStackTraceReturnType) => void): void;

/\*\*

\* Searches for given string in script content.

\*/

post(method: 'Debugger.searchInContent', params?: Debugger.SearchInContentParameterType, callback?: (err: Error | null, params: Debugger.SearchInContentReturnType) => void): void;

post(method: 'Debugger.searchInContent', callback?: (err: Error | null, params: Debugger.SearchInContentReturnType) => void): void;

/\*\*

\* Edits JavaScript source live.

\*/

post(method: 'Debugger.setScriptSource', params?: Debugger.SetScriptSourceParameterType, callback?: (err: Error | null, params: Debugger.SetScriptSourceReturnType) => void): void;

post(method: 'Debugger.setScriptSource', callback?: (err: Error | null, params: Debugger.SetScriptSourceReturnType) => void): void;

/\*\*

\* Restarts particular call frame from the beginning.

\*/

post(method: 'Debugger.restartFrame', params?: Debugger.RestartFrameParameterType, callback?: (err: Error | null, params: Debugger.RestartFrameReturnType) => void): void;

post(method: 'Debugger.restartFrame', callback?: (err: Error | null, params: Debugger.RestartFrameReturnType) => void): void;

/\*\*

\* Returns source for the script with given id.

\*/

post(method: 'Debugger.getScriptSource', params?: Debugger.GetScriptSourceParameterType, callback?: (err: Error | null, params: Debugger.GetScriptSourceReturnType) => void): void;

post(method: 'Debugger.getScriptSource', callback?: (err: Error | null, params: Debugger.GetScriptSourceReturnType) => void): void;

/\*\*

\* Defines pause on exceptions state. Can be set to stop on all exceptions, uncaught exceptions or no exceptions. Initial pause on exceptions state is <code>none</code>.

\*/

post(method: 'Debugger.setPauseOnExceptions', params?: Debugger.SetPauseOnExceptionsParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setPauseOnExceptions', callback?: (err: Error | null) => void): void;

/\*\*

\* Evaluates expression on a given call frame.

\*/

post(method: 'Debugger.evaluateOnCallFrame', params?: Debugger.EvaluateOnCallFrameParameterType, callback?: (err: Error | null, params: Debugger.EvaluateOnCallFrameReturnType) => void): void;

post(method: 'Debugger.evaluateOnCallFrame', callback?: (err: Error | null, params: Debugger.EvaluateOnCallFrameReturnType) => void): void;

/\*\*

\* Changes value of variable in a callframe. Object-based scopes are not supported and must be mutated manually.

\*/

post(method: 'Debugger.setVariableValue', params?: Debugger.SetVariableValueParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setVariableValue', callback?: (err: Error | null) => void): void;

/\*\*

\* Changes return value in top frame. Available only at return break position.

\* @experimental

\*/

post(method: 'Debugger.setReturnValue', params?: Debugger.SetReturnValueParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setReturnValue', callback?: (err: Error | null) => void): void;

/\*\*

\* Enables or disables async call stacks tracking.

\*/

post(method: 'Debugger.setAsyncCallStackDepth', params?: Debugger.SetAsyncCallStackDepthParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setAsyncCallStackDepth', callback?: (err: Error | null) => void): void;

/\*\*

\* Replace previous blackbox patterns with passed ones. Forces backend to skip stepping/pausing in scripts with url matching one of the patterns. VM will try to leave blackboxed script by performing 'step in' several times, finally resorting to 'step out' if unsuccessful.

\* @experimental

\*/

post(method: 'Debugger.setBlackboxPatterns', params?: Debugger.SetBlackboxPatternsParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setBlackboxPatterns', callback?: (err: Error | null) => void): void;

/\*\*

\* Makes backend skip steps in the script in blackboxed ranges. VM will try leave blacklisted scripts by performing 'step in' several times, finally resorting to 'step out' if unsuccessful. Positions array contains positions where blackbox state is changed. First interval isn't blackboxed. Array should be sorted.

\* @experimental

\*/

post(method: 'Debugger.setBlackboxedRanges', params?: Debugger.SetBlackboxedRangesParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Debugger.setBlackboxedRanges', callback?: (err: Error | null) => void): void;

/\*\*

\* Enables console domain, sends the messages collected so far to the client by means of the <code>messageAdded</code> notification.

\*/

post(method: 'Console.enable', callback?: (err: Error | null) => void): void;

/\*\*

\* Disables console domain, prevents further console messages from being reported to the client.

\*/

post(method: 'Console.disable', callback?: (err: Error | null) => void): void;

/\*\*

\* Does nothing.

\*/

post(method: 'Console.clearMessages', callback?: (err: Error | null) => void): void;

post(method: 'Profiler.enable', callback?: (err: Error | null) => void): void;

post(method: 'Profiler.disable', callback?: (err: Error | null) => void): void;

/\*\*

\* Changes CPU profiler sampling interval. Must be called before CPU profiles recording started.

\*/

post(method: 'Profiler.setSamplingInterval', params?: Profiler.SetSamplingIntervalParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Profiler.setSamplingInterval', callback?: (err: Error | null) => void): void;

post(method: 'Profiler.start', callback?: (err: Error | null) => void): void;

post(method: 'Profiler.stop', callback?: (err: Error | null, params: Profiler.StopReturnType) => void): void;

/\*\*

\* Enable precise code coverage. Coverage data for JavaScript executed before enabling precise code coverage may be incomplete. Enabling prevents running optimized code and resets execution counters.

\*/

post(method: 'Profiler.startPreciseCoverage', params?: Profiler.StartPreciseCoverageParameterType, callback?: (err: Error | null) => void): void;

post(method: 'Profiler.startPreciseCoverage', callback?: (err: Error | null) => void): void;

/\*\*

\* Disable precise code coverage. Disabling releases unnecessary execution count records and allows executing optimized code.

\*/

post(method: 'Profiler.stopPreciseCoverage', callback?: (err: Error | null) => void): void;

/\*\*

\* Collect coverage data for the current isolate, and resets execution counters. Precise code coverage needs to have started.

\*/

post(method: 'Profiler.takePreciseCoverage', callback?: (err: Error | null, params: Profiler.TakePreciseCoverageReturnType) => void): void;

/\*\*

\* Collect coverage data for the current isolate. The coverage data may be incomplete due to garbage collection.

\*/

post(method: 'Profiler.getBestEffortCoverage', callback?: (err: Error | null, params: Profiler.GetBestEffortCoverageReturnType) => void): void;

/\*\*

\* Enable type profile.

\* @experimental

\*/

post(method: 'Profiler.startTypeProfile', callback?: (err: Error | null) => void): void;

/\*\*

\* Disable type profile. Disabling releases type profile data collected so far.

\* @experimental

\*/

post(method: 'Profiler.stopTypeProfile', callback?: (err: Error | null) => void): void;

/\*\*

\* Collect type profile.

\* @experimental

\*/

post(method: 'Profiler.takeTypeProfile', callback?: (err: Error | null, params: Profiler.TakeTypeProfileReturnType) => void): void;

post(method: 'HeapProfiler.enable', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.disable', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.startTrackingHeapObjects', params?: HeapProfiler.StartTrackingHeapObjectsParameterType, callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.startTrackingHeapObjects', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.stopTrackingHeapObjects', params?: HeapProfiler.StopTrackingHeapObjectsParameterType, callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.stopTrackingHeapObjects', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.takeHeapSnapshot', params?: HeapProfiler.TakeHeapSnapshotParameterType, callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.takeHeapSnapshot', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.collectGarbage', callback?: (err: Error | null) => void): void;

post(

method: 'HeapProfiler.getObjectByHeapObjectId',

params?: HeapProfiler.GetObjectByHeapObjectIdParameterType,

callback?: (err: Error | null, params: HeapProfiler.GetObjectByHeapObjectIdReturnType) => void

): void;

post(method: 'HeapProfiler.getObjectByHeapObjectId', callback?: (err: Error | null, params: HeapProfiler.GetObjectByHeapObjectIdReturnType) => void): void;

/\*\*

\* Enables console to refer to the node with given id via $x (see Command Line API for more details $x functions).

\*/

post(method: 'HeapProfiler.addInspectedHeapObject', params?: HeapProfiler.AddInspectedHeapObjectParameterType, callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.addInspectedHeapObject', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.getHeapObjectId', params?: HeapProfiler.GetHeapObjectIdParameterType, callback?: (err: Error | null, params: HeapProfiler.GetHeapObjectIdReturnType) => void): void;

post(method: 'HeapProfiler.getHeapObjectId', callback?: (err: Error | null, params: HeapProfiler.GetHeapObjectIdReturnType) => void): void;

post(method: 'HeapProfiler.startSampling', params?: HeapProfiler.StartSamplingParameterType, callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.startSampling', callback?: (err: Error | null) => void): void;

post(method: 'HeapProfiler.stopSampling', callback?: (err: Error | null, params: HeapProfiler.StopSamplingReturnType) => void): void;

post(method: 'HeapProfiler.getSamplingProfile', callback?: (err: Error | null, params: HeapProfiler.GetSamplingProfileReturnType) => void): void;

/\*\*

\* Gets supported tracing categories.

\*/

post(method: 'NodeTracing.getCategories', callback?: (err: Error | null, params: NodeTracing.GetCategoriesReturnType) => void): void;

/\*\*

\* Start trace events collection.

\*/

post(method: 'NodeTracing.start', params?: NodeTracing.StartParameterType, callback?: (err: Error | null) => void): void;

post(method: 'NodeTracing.start', callback?: (err: Error | null) => void): void;

/\*\*

\* Stop trace events collection. Remaining collected events will be sent as a sequence of

\* dataCollected events followed by tracingComplete event.

\*/

post(method: 'NodeTracing.stop', callback?: (err: Error | null) => void): void;

/\*\*

\* Sends protocol message over session with given id.

\*/

post(method: 'NodeWorker.sendMessageToWorker', params?: NodeWorker.SendMessageToWorkerParameterType, callback?: (err: Error | null) => void): void;

post(method: 'NodeWorker.sendMessageToWorker', callback?: (err: Error | null) => void): void;

/\*\*

\* Instructs the inspector to attach to running workers. Will also attach to new workers

\* as they start

\*/

post(method: 'NodeWorker.enable', params?: NodeWorker.EnableParameterType, callback?: (err: Error | null) => void): void;

post(method: 'NodeWorker.enable', callback?: (err: Error | null) => void): void;

/\*\*

\* Detaches from all running workers and disables attaching to new workers as they are started.

\*/

post(method: 'NodeWorker.disable', callback?: (err: Error | null) => void): void;

/\*\*

\* Detached from the worker with given sessionId.

\*/

post(method: 'NodeWorker.detach', params?: NodeWorker.DetachParameterType, callback?: (err: Error | null) => void): void;

post(method: 'NodeWorker.detach', callback?: (err: Error | null) => void): void;

/\*\*

\* Enable the `NodeRuntime.waitingForDisconnect`.

\*/

post(method: 'NodeRuntime.notifyWhenWaitingForDisconnect', params?: NodeRuntime.NotifyWhenWaitingForDisconnectParameterType, callback?: (err: Error | null) => void): void;

post(method: 'NodeRuntime.notifyWhenWaitingForDisconnect', callback?: (err: Error | null) => void): void;

// Events

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

/\*\*

\* Emitted when any notification from the V8 Inspector is received.

\*/

addListener(event: 'inspectorNotification', listener: (message: InspectorNotification<{}>) => void): this;

/\*\*

\* Issued when new execution context is created.

\*/

addListener(event: 'Runtime.executionContextCreated', listener: (message: InspectorNotification<Runtime.ExecutionContextCreatedEventDataType>) => void): this;

/\*\*

\* Issued when execution context is destroyed.

\*/

addListener(event: 'Runtime.executionContextDestroyed', listener: (message: InspectorNotification<Runtime.ExecutionContextDestroyedEventDataType>) => void): this;

/\*\*

\* Issued when all executionContexts were cleared in browser

\*/

addListener(event: 'Runtime.executionContextsCleared', listener: () => void): this;

/\*\*

\* Issued when exception was thrown and unhandled.

\*/

addListener(event: 'Runtime.exceptionThrown', listener: (message: InspectorNotification<Runtime.ExceptionThrownEventDataType>) => void): this;

/\*\*

\* Issued when unhandled exception was revoked.

\*/

addListener(event: 'Runtime.exceptionRevoked', listener: (message: InspectorNotification<Runtime.ExceptionRevokedEventDataType>) => void): this;

/\*\*

\* Issued when console API was called.

\*/

addListener(event: 'Runtime.consoleAPICalled', listener: (message: InspectorNotification<Runtime.ConsoleAPICalledEventDataType>) => void): this;

/\*\*

\* Issued when object should be inspected (for example, as a result of inspect() command line API call).

\*/

addListener(event: 'Runtime.inspectRequested', listener: (message: InspectorNotification<Runtime.InspectRequestedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

\*/

addListener(event: 'Debugger.scriptParsed', listener: (message: InspectorNotification<Debugger.ScriptParsedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine fails to parse the script.

\*/

addListener(event: 'Debugger.scriptFailedToParse', listener: (message: InspectorNotification<Debugger.ScriptFailedToParseEventDataType>) => void): this;

/\*\*

\* Fired when breakpoint is resolved to an actual script and location.

\*/

addListener(event: 'Debugger.breakpointResolved', listener: (message: InspectorNotification<Debugger.BreakpointResolvedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

\*/

addListener(event: 'Debugger.paused', listener: (message: InspectorNotification<Debugger.PausedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine resumed execution.

\*/

addListener(event: 'Debugger.resumed', listener: () => void): this;

/\*\*

\* Issued when new console message is added.

\*/

addListener(event: 'Console.messageAdded', listener: (message: InspectorNotification<Console.MessageAddedEventDataType>) => void): this;

/\*\*

\* Sent when new profile recording is started using console.profile() call.

\*/

addListener(event: 'Profiler.consoleProfileStarted', listener: (message: InspectorNotification<Profiler.ConsoleProfileStartedEventDataType>) => void): this;

addListener(event: 'Profiler.consoleProfileFinished', listener: (message: InspectorNotification<Profiler.ConsoleProfileFinishedEventDataType>) => void): this;

addListener(event: 'HeapProfiler.addHeapSnapshotChunk', listener: (message: InspectorNotification<HeapProfiler.AddHeapSnapshotChunkEventDataType>) => void): this;

addListener(event: 'HeapProfiler.resetProfiles', listener: () => void): this;

addListener(event: 'HeapProfiler.reportHeapSnapshotProgress', listener: (message: InspectorNotification<HeapProfiler.ReportHeapSnapshotProgressEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend regularly sends a current value for last seen object id and corresponding timestamp. If the were changes in the heap since last event then one or more heapStatsUpdate events will be sent before a new lastSeenObjectId event.

\*/

addListener(event: 'HeapProfiler.lastSeenObjectId', listener: (message: InspectorNotification<HeapProfiler.LastSeenObjectIdEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend may send update for one or more fragments

\*/

addListener(event: 'HeapProfiler.heapStatsUpdate', listener: (message: InspectorNotification<HeapProfiler.HeapStatsUpdateEventDataType>) => void): this;

/\*\*

\* Contains an bucket of collected trace events.

\*/

addListener(event: 'NodeTracing.dataCollected', listener: (message: InspectorNotification<NodeTracing.DataCollectedEventDataType>) => void): this;

/\*\*

\* Signals that tracing is stopped and there is no trace buffers pending flush, all data were

\* delivered via dataCollected events.

\*/

addListener(event: 'NodeTracing.tracingComplete', listener: () => void): this;

/\*\*

\* Issued when attached to a worker.

\*/

addListener(event: 'NodeWorker.attachedToWorker', listener: (message: InspectorNotification<NodeWorker.AttachedToWorkerEventDataType>) => void): this;

/\*\*

\* Issued when detached from the worker.

\*/

addListener(event: 'NodeWorker.detachedFromWorker', listener: (message: InspectorNotification<NodeWorker.DetachedFromWorkerEventDataType>) => void): this;

/\*\*

\* Notifies about a new protocol message received from the session

\* (session ID is provided in attachedToWorker notification).

\*/

addListener(event: 'NodeWorker.receivedMessageFromWorker', listener: (message: InspectorNotification<NodeWorker.ReceivedMessageFromWorkerEventDataType>) => void): this;

/\*\*

\* This event is fired instead of `Runtime.executionContextDestroyed` when

\* enabled.

\* It is fired when the Node process finished all code execution and is

\* waiting for all frontends to disconnect.

\*/

addListener(event: 'NodeRuntime.waitingForDisconnect', listener: () => void): this;

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

emit(event: 'inspectorNotification', message: InspectorNotification<{}>): boolean;

emit(event: 'Runtime.executionContextCreated', message: InspectorNotification<Runtime.ExecutionContextCreatedEventDataType>): boolean;

emit(event: 'Runtime.executionContextDestroyed', message: InspectorNotification<Runtime.ExecutionContextDestroyedEventDataType>): boolean;

emit(event: 'Runtime.executionContextsCleared'): boolean;

emit(event: 'Runtime.exceptionThrown', message: InspectorNotification<Runtime.ExceptionThrownEventDataType>): boolean;

emit(event: 'Runtime.exceptionRevoked', message: InspectorNotification<Runtime.ExceptionRevokedEventDataType>): boolean;

emit(event: 'Runtime.consoleAPICalled', message: InspectorNotification<Runtime.ConsoleAPICalledEventDataType>): boolean;

emit(event: 'Runtime.inspectRequested', message: InspectorNotification<Runtime.InspectRequestedEventDataType>): boolean;

emit(event: 'Debugger.scriptParsed', message: InspectorNotification<Debugger.ScriptParsedEventDataType>): boolean;

emit(event: 'Debugger.scriptFailedToParse', message: InspectorNotification<Debugger.ScriptFailedToParseEventDataType>): boolean;

emit(event: 'Debugger.breakpointResolved', message: InspectorNotification<Debugger.BreakpointResolvedEventDataType>): boolean;

emit(event: 'Debugger.paused', message: InspectorNotification<Debugger.PausedEventDataType>): boolean;

emit(event: 'Debugger.resumed'): boolean;

emit(event: 'Console.messageAdded', message: InspectorNotification<Console.MessageAddedEventDataType>): boolean;

emit(event: 'Profiler.consoleProfileStarted', message: InspectorNotification<Profiler.ConsoleProfileStartedEventDataType>): boolean;

emit(event: 'Profiler.consoleProfileFinished', message: InspectorNotification<Profiler.ConsoleProfileFinishedEventDataType>): boolean;

emit(event: 'HeapProfiler.addHeapSnapshotChunk', message: InspectorNotification<HeapProfiler.AddHeapSnapshotChunkEventDataType>): boolean;

emit(event: 'HeapProfiler.resetProfiles'): boolean;

emit(event: 'HeapProfiler.reportHeapSnapshotProgress', message: InspectorNotification<HeapProfiler.ReportHeapSnapshotProgressEventDataType>): boolean;

emit(event: 'HeapProfiler.lastSeenObjectId', message: InspectorNotification<HeapProfiler.LastSeenObjectIdEventDataType>): boolean;

emit(event: 'HeapProfiler.heapStatsUpdate', message: InspectorNotification<HeapProfiler.HeapStatsUpdateEventDataType>): boolean;

emit(event: 'NodeTracing.dataCollected', message: InspectorNotification<NodeTracing.DataCollectedEventDataType>): boolean;

emit(event: 'NodeTracing.tracingComplete'): boolean;

emit(event: 'NodeWorker.attachedToWorker', message: InspectorNotification<NodeWorker.AttachedToWorkerEventDataType>): boolean;

emit(event: 'NodeWorker.detachedFromWorker', message: InspectorNotification<NodeWorker.DetachedFromWorkerEventDataType>): boolean;

emit(event: 'NodeWorker.receivedMessageFromWorker', message: InspectorNotification<NodeWorker.ReceivedMessageFromWorkerEventDataType>): boolean;

emit(event: 'NodeRuntime.waitingForDisconnect'): boolean;

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

/\*\*

\* Emitted when any notification from the V8 Inspector is received.

\*/

on(event: 'inspectorNotification', listener: (message: InspectorNotification<{}>) => void): this;

/\*\*

\* Issued when new execution context is created.

\*/

on(event: 'Runtime.executionContextCreated', listener: (message: InspectorNotification<Runtime.ExecutionContextCreatedEventDataType>) => void): this;

/\*\*

\* Issued when execution context is destroyed.

\*/

on(event: 'Runtime.executionContextDestroyed', listener: (message: InspectorNotification<Runtime.ExecutionContextDestroyedEventDataType>) => void): this;

/\*\*

\* Issued when all executionContexts were cleared in browser

\*/

on(event: 'Runtime.executionContextsCleared', listener: () => void): this;

/\*\*

\* Issued when exception was thrown and unhandled.

\*/

on(event: 'Runtime.exceptionThrown', listener: (message: InspectorNotification<Runtime.ExceptionThrownEventDataType>) => void): this;

/\*\*

\* Issued when unhandled exception was revoked.

\*/

on(event: 'Runtime.exceptionRevoked', listener: (message: InspectorNotification<Runtime.ExceptionRevokedEventDataType>) => void): this;

/\*\*

\* Issued when console API was called.

\*/

on(event: 'Runtime.consoleAPICalled', listener: (message: InspectorNotification<Runtime.ConsoleAPICalledEventDataType>) => void): this;

/\*\*

\* Issued when object should be inspected (for example, as a result of inspect() command line API call).

\*/

on(event: 'Runtime.inspectRequested', listener: (message: InspectorNotification<Runtime.InspectRequestedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

\*/

on(event: 'Debugger.scriptParsed', listener: (message: InspectorNotification<Debugger.ScriptParsedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine fails to parse the script.

\*/

on(event: 'Debugger.scriptFailedToParse', listener: (message: InspectorNotification<Debugger.ScriptFailedToParseEventDataType>) => void): this;

/\*\*

\* Fired when breakpoint is resolved to an actual script and location.

\*/

on(event: 'Debugger.breakpointResolved', listener: (message: InspectorNotification<Debugger.BreakpointResolvedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

\*/

on(event: 'Debugger.paused', listener: (message: InspectorNotification<Debugger.PausedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine resumed execution.

\*/

on(event: 'Debugger.resumed', listener: () => void): this;

/\*\*

\* Issued when new console message is added.

\*/

on(event: 'Console.messageAdded', listener: (message: InspectorNotification<Console.MessageAddedEventDataType>) => void): this;

/\*\*

\* Sent when new profile recording is started using console.profile() call.

\*/

on(event: 'Profiler.consoleProfileStarted', listener: (message: InspectorNotification<Profiler.ConsoleProfileStartedEventDataType>) => void): this;

on(event: 'Profiler.consoleProfileFinished', listener: (message: InspectorNotification<Profiler.ConsoleProfileFinishedEventDataType>) => void): this;

on(event: 'HeapProfiler.addHeapSnapshotChunk', listener: (message: InspectorNotification<HeapProfiler.AddHeapSnapshotChunkEventDataType>) => void): this;

on(event: 'HeapProfiler.resetProfiles', listener: () => void): this;

on(event: 'HeapProfiler.reportHeapSnapshotProgress', listener: (message: InspectorNotification<HeapProfiler.ReportHeapSnapshotProgressEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend regularly sends a current value for last seen object id and corresponding timestamp. If the were changes in the heap since last event then one or more heapStatsUpdate events will be sent before a new lastSeenObjectId event.

\*/

on(event: 'HeapProfiler.lastSeenObjectId', listener: (message: InspectorNotification<HeapProfiler.LastSeenObjectIdEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend may send update for one or more fragments

\*/

on(event: 'HeapProfiler.heapStatsUpdate', listener: (message: InspectorNotification<HeapProfiler.HeapStatsUpdateEventDataType>) => void): this;

/\*\*

\* Contains an bucket of collected trace events.

\*/

on(event: 'NodeTracing.dataCollected', listener: (message: InspectorNotification<NodeTracing.DataCollectedEventDataType>) => void): this;

/\*\*

\* Signals that tracing is stopped and there is no trace buffers pending flush, all data were

\* delivered via dataCollected events.

\*/

on(event: 'NodeTracing.tracingComplete', listener: () => void): this;

/\*\*

\* Issued when attached to a worker.

\*/

on(event: 'NodeWorker.attachedToWorker', listener: (message: InspectorNotification<NodeWorker.AttachedToWorkerEventDataType>) => void): this;

/\*\*

\* Issued when detached from the worker.

\*/

on(event: 'NodeWorker.detachedFromWorker', listener: (message: InspectorNotification<NodeWorker.DetachedFromWorkerEventDataType>) => void): this;

/\*\*

\* Notifies about a new protocol message received from the session

\* (session ID is provided in attachedToWorker notification).

\*/

on(event: 'NodeWorker.receivedMessageFromWorker', listener: (message: InspectorNotification<NodeWorker.ReceivedMessageFromWorkerEventDataType>) => void): this;

/\*\*

\* This event is fired instead of `Runtime.executionContextDestroyed` when

\* enabled.

\* It is fired when the Node process finished all code execution and is

\* waiting for all frontends to disconnect.

\*/

on(event: 'NodeRuntime.waitingForDisconnect', listener: () => void): this;

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

/\*\*

\* Emitted when any notification from the V8 Inspector is received.

\*/

once(event: 'inspectorNotification', listener: (message: InspectorNotification<{}>) => void): this;

/\*\*

\* Issued when new execution context is created.

\*/

once(event: 'Runtime.executionContextCreated', listener: (message: InspectorNotification<Runtime.ExecutionContextCreatedEventDataType>) => void): this;

/\*\*

\* Issued when execution context is destroyed.

\*/

once(event: 'Runtime.executionContextDestroyed', listener: (message: InspectorNotification<Runtime.ExecutionContextDestroyedEventDataType>) => void): this;

/\*\*

\* Issued when all executionContexts were cleared in browser

\*/

once(event: 'Runtime.executionContextsCleared', listener: () => void): this;

/\*\*

\* Issued when exception was thrown and unhandled.

\*/

once(event: 'Runtime.exceptionThrown', listener: (message: InspectorNotification<Runtime.ExceptionThrownEventDataType>) => void): this;

/\*\*

\* Issued when unhandled exception was revoked.

\*/

once(event: 'Runtime.exceptionRevoked', listener: (message: InspectorNotification<Runtime.ExceptionRevokedEventDataType>) => void): this;

/\*\*

\* Issued when console API was called.

\*/

once(event: 'Runtime.consoleAPICalled', listener: (message: InspectorNotification<Runtime.ConsoleAPICalledEventDataType>) => void): this;

/\*\*

\* Issued when object should be inspected (for example, as a result of inspect() command line API call).

\*/

once(event: 'Runtime.inspectRequested', listener: (message: InspectorNotification<Runtime.InspectRequestedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

\*/

once(event: 'Debugger.scriptParsed', listener: (message: InspectorNotification<Debugger.ScriptParsedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine fails to parse the script.

\*/

once(event: 'Debugger.scriptFailedToParse', listener: (message: InspectorNotification<Debugger.ScriptFailedToParseEventDataType>) => void): this;

/\*\*

\* Fired when breakpoint is resolved to an actual script and location.

\*/

once(event: 'Debugger.breakpointResolved', listener: (message: InspectorNotification<Debugger.BreakpointResolvedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

\*/

once(event: 'Debugger.paused', listener: (message: InspectorNotification<Debugger.PausedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine resumed execution.

\*/

once(event: 'Debugger.resumed', listener: () => void): this;

/\*\*

\* Issued when new console message is added.

\*/

once(event: 'Console.messageAdded', listener: (message: InspectorNotification<Console.MessageAddedEventDataType>) => void): this;

/\*\*

\* Sent when new profile recording is started using console.profile() call.

\*/

once(event: 'Profiler.consoleProfileStarted', listener: (message: InspectorNotification<Profiler.ConsoleProfileStartedEventDataType>) => void): this;

once(event: 'Profiler.consoleProfileFinished', listener: (message: InspectorNotification<Profiler.ConsoleProfileFinishedEventDataType>) => void): this;

once(event: 'HeapProfiler.addHeapSnapshotChunk', listener: (message: InspectorNotification<HeapProfiler.AddHeapSnapshotChunkEventDataType>) => void): this;

once(event: 'HeapProfiler.resetProfiles', listener: () => void): this;

once(event: 'HeapProfiler.reportHeapSnapshotProgress', listener: (message: InspectorNotification<HeapProfiler.ReportHeapSnapshotProgressEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend regularly sends a current value for last seen object id and corresponding timestamp. If the were changes in the heap since last event then one or more heapStatsUpdate events will be sent before a new lastSeenObjectId event.

\*/

once(event: 'HeapProfiler.lastSeenObjectId', listener: (message: InspectorNotification<HeapProfiler.LastSeenObjectIdEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend may send update for one or more fragments

\*/

once(event: 'HeapProfiler.heapStatsUpdate', listener: (message: InspectorNotification<HeapProfiler.HeapStatsUpdateEventDataType>) => void): this;

/\*\*

\* Contains an bucket of collected trace events.

\*/

once(event: 'NodeTracing.dataCollected', listener: (message: InspectorNotification<NodeTracing.DataCollectedEventDataType>) => void): this;

/\*\*

\* Signals that tracing is stopped and there is no trace buffers pending flush, all data were

\* delivered via dataCollected events.

\*/

once(event: 'NodeTracing.tracingComplete', listener: () => void): this;

/\*\*

\* Issued when attached to a worker.

\*/

once(event: 'NodeWorker.attachedToWorker', listener: (message: InspectorNotification<NodeWorker.AttachedToWorkerEventDataType>) => void): this;

/\*\*

\* Issued when detached from the worker.

\*/

once(event: 'NodeWorker.detachedFromWorker', listener: (message: InspectorNotification<NodeWorker.DetachedFromWorkerEventDataType>) => void): this;

/\*\*

\* Notifies about a new protocol message received from the session

\* (session ID is provided in attachedToWorker notification).

\*/

once(event: 'NodeWorker.receivedMessageFromWorker', listener: (message: InspectorNotification<NodeWorker.ReceivedMessageFromWorkerEventDataType>) => void): this;

/\*\*

\* This event is fired instead of `Runtime.executionContextDestroyed` when

\* enabled.

\* It is fired when the Node process finished all code execution and is

\* waiting for all frontends to disconnect.

\*/

once(event: 'NodeRuntime.waitingForDisconnect', listener: () => void): this;

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

/\*\*

\* Emitted when any notification from the V8 Inspector is received.

\*/

prependListener(event: 'inspectorNotification', listener: (message: InspectorNotification<{}>) => void): this;

/\*\*

\* Issued when new execution context is created.

\*/

prependListener(event: 'Runtime.executionContextCreated', listener: (message: InspectorNotification<Runtime.ExecutionContextCreatedEventDataType>) => void): this;

/\*\*

\* Issued when execution context is destroyed.

\*/

prependListener(event: 'Runtime.executionContextDestroyed', listener: (message: InspectorNotification<Runtime.ExecutionContextDestroyedEventDataType>) => void): this;

/\*\*

\* Issued when all executionContexts were cleared in browser

\*/

prependListener(event: 'Runtime.executionContextsCleared', listener: () => void): this;

/\*\*

\* Issued when exception was thrown and unhandled.

\*/

prependListener(event: 'Runtime.exceptionThrown', listener: (message: InspectorNotification<Runtime.ExceptionThrownEventDataType>) => void): this;

/\*\*

\* Issued when unhandled exception was revoked.

\*/

prependListener(event: 'Runtime.exceptionRevoked', listener: (message: InspectorNotification<Runtime.ExceptionRevokedEventDataType>) => void): this;

/\*\*

\* Issued when console API was called.

\*/

prependListener(event: 'Runtime.consoleAPICalled', listener: (message: InspectorNotification<Runtime.ConsoleAPICalledEventDataType>) => void): this;

/\*\*

\* Issued when object should be inspected (for example, as a result of inspect() command line API call).

\*/

prependListener(event: 'Runtime.inspectRequested', listener: (message: InspectorNotification<Runtime.InspectRequestedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

\*/

prependListener(event: 'Debugger.scriptParsed', listener: (message: InspectorNotification<Debugger.ScriptParsedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine fails to parse the script.

\*/

prependListener(event: 'Debugger.scriptFailedToParse', listener: (message: InspectorNotification<Debugger.ScriptFailedToParseEventDataType>) => void): this;

/\*\*

\* Fired when breakpoint is resolved to an actual script and location.

\*/

prependListener(event: 'Debugger.breakpointResolved', listener: (message: InspectorNotification<Debugger.BreakpointResolvedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

\*/

prependListener(event: 'Debugger.paused', listener: (message: InspectorNotification<Debugger.PausedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine resumed execution.

\*/

prependListener(event: 'Debugger.resumed', listener: () => void): this;

/\*\*

\* Issued when new console message is added.

\*/

prependListener(event: 'Console.messageAdded', listener: (message: InspectorNotification<Console.MessageAddedEventDataType>) => void): this;

/\*\*

\* Sent when new profile recording is started using console.profile() call.

\*/

prependListener(event: 'Profiler.consoleProfileStarted', listener: (message: InspectorNotification<Profiler.ConsoleProfileStartedEventDataType>) => void): this;

prependListener(event: 'Profiler.consoleProfileFinished', listener: (message: InspectorNotification<Profiler.ConsoleProfileFinishedEventDataType>) => void): this;

prependListener(event: 'HeapProfiler.addHeapSnapshotChunk', listener: (message: InspectorNotification<HeapProfiler.AddHeapSnapshotChunkEventDataType>) => void): this;

prependListener(event: 'HeapProfiler.resetProfiles', listener: () => void): this;

prependListener(event: 'HeapProfiler.reportHeapSnapshotProgress', listener: (message: InspectorNotification<HeapProfiler.ReportHeapSnapshotProgressEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend regularly sends a current value for last seen object id and corresponding timestamp. If the were changes in the heap since last event then one or more heapStatsUpdate events will be sent before a new lastSeenObjectId event.

\*/

prependListener(event: 'HeapProfiler.lastSeenObjectId', listener: (message: InspectorNotification<HeapProfiler.LastSeenObjectIdEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend may send update for one or more fragments

\*/

prependListener(event: 'HeapProfiler.heapStatsUpdate', listener: (message: InspectorNotification<HeapProfiler.HeapStatsUpdateEventDataType>) => void): this;

/\*\*

\* Contains an bucket of collected trace events.

\*/

prependListener(event: 'NodeTracing.dataCollected', listener: (message: InspectorNotification<NodeTracing.DataCollectedEventDataType>) => void): this;

/\*\*

\* Signals that tracing is stopped and there is no trace buffers pending flush, all data were

\* delivered via dataCollected events.

\*/

prependListener(event: 'NodeTracing.tracingComplete', listener: () => void): this;

/\*\*

\* Issued when attached to a worker.

\*/

prependListener(event: 'NodeWorker.attachedToWorker', listener: (message: InspectorNotification<NodeWorker.AttachedToWorkerEventDataType>) => void): this;

/\*\*

\* Issued when detached from the worker.

\*/

prependListener(event: 'NodeWorker.detachedFromWorker', listener: (message: InspectorNotification<NodeWorker.DetachedFromWorkerEventDataType>) => void): this;

/\*\*

\* Notifies about a new protocol message received from the session

\* (session ID is provided in attachedToWorker notification).

\*/

prependListener(event: 'NodeWorker.receivedMessageFromWorker', listener: (message: InspectorNotification<NodeWorker.ReceivedMessageFromWorkerEventDataType>) => void): this;

/\*\*

\* This event is fired instead of `Runtime.executionContextDestroyed` when

\* enabled.

\* It is fired when the Node process finished all code execution and is

\* waiting for all frontends to disconnect.

\*/

prependListener(event: 'NodeRuntime.waitingForDisconnect', listener: () => void): this;

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

/\*\*

\* Emitted when any notification from the V8 Inspector is received.

\*/

prependOnceListener(event: 'inspectorNotification', listener: (message: InspectorNotification<{}>) => void): this;

/\*\*

\* Issued when new execution context is created.

\*/

prependOnceListener(event: 'Runtime.executionContextCreated', listener: (message: InspectorNotification<Runtime.ExecutionContextCreatedEventDataType>) => void): this;

/\*\*

\* Issued when execution context is destroyed.

\*/

prependOnceListener(event: 'Runtime.executionContextDestroyed', listener: (message: InspectorNotification<Runtime.ExecutionContextDestroyedEventDataType>) => void): this;

/\*\*

\* Issued when all executionContexts were cleared in browser

\*/

prependOnceListener(event: 'Runtime.executionContextsCleared', listener: () => void): this;

/\*\*

\* Issued when exception was thrown and unhandled.

\*/

prependOnceListener(event: 'Runtime.exceptionThrown', listener: (message: InspectorNotification<Runtime.ExceptionThrownEventDataType>) => void): this;

/\*\*

\* Issued when unhandled exception was revoked.

\*/

prependOnceListener(event: 'Runtime.exceptionRevoked', listener: (message: InspectorNotification<Runtime.ExceptionRevokedEventDataType>) => void): this;

/\*\*

\* Issued when console API was called.

\*/

prependOnceListener(event: 'Runtime.consoleAPICalled', listener: (message: InspectorNotification<Runtime.ConsoleAPICalledEventDataType>) => void): this;

/\*\*

\* Issued when object should be inspected (for example, as a result of inspect() command line API call).

\*/

prependOnceListener(event: 'Runtime.inspectRequested', listener: (message: InspectorNotification<Runtime.InspectRequestedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine parses script. This event is also fired for all known and uncollected scripts upon enabling debugger.

\*/

prependOnceListener(event: 'Debugger.scriptParsed', listener: (message: InspectorNotification<Debugger.ScriptParsedEventDataType>) => void): this;

/\*\*

\* Fired when virtual machine fails to parse the script.

\*/

prependOnceListener(event: 'Debugger.scriptFailedToParse', listener: (message: InspectorNotification<Debugger.ScriptFailedToParseEventDataType>) => void): this;

/\*\*

\* Fired when breakpoint is resolved to an actual script and location.

\*/

prependOnceListener(event: 'Debugger.breakpointResolved', listener: (message: InspectorNotification<Debugger.BreakpointResolvedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine stopped on breakpoint or exception or any other stop criteria.

\*/

prependOnceListener(event: 'Debugger.paused', listener: (message: InspectorNotification<Debugger.PausedEventDataType>) => void): this;

/\*\*

\* Fired when the virtual machine resumed execution.

\*/

prependOnceListener(event: 'Debugger.resumed', listener: () => void): this;

/\*\*

\* Issued when new console message is added.

\*/

prependOnceListener(event: 'Console.messageAdded', listener: (message: InspectorNotification<Console.MessageAddedEventDataType>) => void): this;

/\*\*

\* Sent when new profile recording is started using console.profile() call.

\*/

prependOnceListener(event: 'Profiler.consoleProfileStarted', listener: (message: InspectorNotification<Profiler.ConsoleProfileStartedEventDataType>) => void): this;

prependOnceListener(event: 'Profiler.consoleProfileFinished', listener: (message: InspectorNotification<Profiler.ConsoleProfileFinishedEventDataType>) => void): this;

prependOnceListener(event: 'HeapProfiler.addHeapSnapshotChunk', listener: (message: InspectorNotification<HeapProfiler.AddHeapSnapshotChunkEventDataType>) => void): this;

prependOnceListener(event: 'HeapProfiler.resetProfiles', listener: () => void): this;

prependOnceListener(event: 'HeapProfiler.reportHeapSnapshotProgress', listener: (message: InspectorNotification<HeapProfiler.ReportHeapSnapshotProgressEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend regularly sends a current value for last seen object id and corresponding timestamp. If the were changes in the heap since last event then one or more heapStatsUpdate events will be sent before a new lastSeenObjectId event.

\*/

prependOnceListener(event: 'HeapProfiler.lastSeenObjectId', listener: (message: InspectorNotification<HeapProfiler.LastSeenObjectIdEventDataType>) => void): this;

/\*\*

\* If heap objects tracking has been started then backend may send update for one or more fragments

\*/

prependOnceListener(event: 'HeapProfiler.heapStatsUpdate', listener: (message: InspectorNotification<HeapProfiler.HeapStatsUpdateEventDataType>) => void): this;

/\*\*

\* Contains an bucket of collected trace events.

\*/

prependOnceListener(event: 'NodeTracing.dataCollected', listener: (message: InspectorNotification<NodeTracing.DataCollectedEventDataType>) => void): this;

/\*\*

\* Signals that tracing is stopped and there is no trace buffers pending flush, all data were

\* delivered via dataCollected events.

\*/

prependOnceListener(event: 'NodeTracing.tracingComplete', listener: () => void): this;

/\*\*

\* Issued when attached to a worker.

\*/

prependOnceListener(event: 'NodeWorker.attachedToWorker', listener: (message: InspectorNotification<NodeWorker.AttachedToWorkerEventDataType>) => void): this;

/\*\*

\* Issued when detached from the worker.

\*/

prependOnceListener(event: 'NodeWorker.detachedFromWorker', listener: (message: InspectorNotification<NodeWorker.DetachedFromWorkerEventDataType>) => void): this;

/\*\*

\* Notifies about a new protocol message received from the session

\* (session ID is provided in attachedToWorker notification).

\*/

prependOnceListener(event: 'NodeWorker.receivedMessageFromWorker', listener: (message: InspectorNotification<NodeWorker.ReceivedMessageFromWorkerEventDataType>) => void): this;

/\*\*

\* This event is fired instead of `Runtime.executionContextDestroyed` when

\* enabled.

\* It is fired when the Node process finished all code execution and is

\* waiting for all frontends to disconnect.

\*/

prependOnceListener(event: 'NodeRuntime.waitingForDisconnect', listener: () => void): this;

}

/\*\*

\* Activate inspector on host and port. Equivalent to `node --inspect=[[host:]port]`, but can be done programmatically after node has

\* started.

\*

\* If wait is `true`, will block until a client has connected to the inspect port

\* and flow control has been passed to the debugger client.

\*

\* See the `security warning` regarding the `host`parameter usage.

\* @param [port='what was specified on the CLI'] Port to listen on for inspector connections. Optional.

\* @param [host='what was specified on the CLI'] Host to listen on for inspector connections. Optional.

\* @param [wait=false] Block until a client has connected. Optional.

\*/

function open(port?: number, host?: string, wait?: boolean): void;

/\*\*

\* Deactivate the inspector. Blocks until there are no active connections.

\*/

function close(): void;

/\*\*

\* Return the URL of the active inspector, or `undefined` if there is none.

\*

\* ```console

\* $ node --inspect -p 'inspector.url()'

\* Debugger listening on ws://127.0.0.1:9229/166e272e-7a30-4d09-97ce-f1c012b43c34

\* For help see https://nodejs.org/en/docs/inspector

\* ws://127.0.0.1:9229/166e272e-7a30-4d09-97ce-f1c012b43c34

\*

\* $ node --inspect=localhost:3000 -p 'inspector.url()'

\* Debugger listening on ws://localhost:3000/51cf8d0e-3c36-4c59-8efd-54519839e56a

\* For help see https://nodejs.org/en/docs/inspector

\* ws://localhost:3000/51cf8d0e-3c36-4c59-8efd-54519839e56a

\*

\* $ node -p 'inspector.url()'

\* undefined

\* ```

\*/

function url(): string | undefined;

/\*\*

\* Blocks until a client (existing or connected later) has sent`Runtime.runIfWaitingForDebugger` command.

\*

\* An exception will be thrown if there is no active inspector.

\* @since v12.7.0

\*/

function waitForDebugger(): void;

}

declare module 'node:inspector' {

import EventEmitter = require('inspector');

export = EventEmitter;

}