declare module 'stream/web' {

// stub module, pending copy&paste from .d.ts or manual impl

// copy from lib.dom.d.ts

interface ReadableWritablePair<R = any, W = any> {

readable: ReadableStream<R>;

/\*\*

\* Provides a convenient, chainable way of piping this readable stream

\* through a transform stream (or any other { writable, readable }

\* pair). It simply pipes the stream into the writable side of the

\* supplied pair, and returns the readable side for further use.

\*

\* Piping a stream will lock it for the duration of the pipe, preventing

\* any other consumer from acquiring a reader.

\*/

writable: WritableStream<W>;

}

interface StreamPipeOptions {

preventAbort?: boolean;

preventCancel?: boolean;

/\*\*

\* Pipes this readable stream to a given writable stream destination.

\* The way in which the piping process behaves under various error

\* conditions can be customized with a number of passed options. It

\* returns a promise that fulfills when the piping process completes

\* successfully, or rejects if any errors were encountered.

\*

\* Piping a stream will lock it for the duration of the pipe, preventing

\* any other consumer from acquiring a reader.

\*

\* Errors and closures of the source and destination streams propagate

\* as follows:

\*

\* An error in this source readable stream will abort destination,

\* unless preventAbort is truthy. The returned promise will be rejected

\* with the source's error, or with any error that occurs during

\* aborting the destination.

\*

\* An error in destination will cancel this source readable stream,

\* unless preventCancel is truthy. The returned promise will be rejected

\* with the destination's error, or with any error that occurs during

\* canceling the source.

\*

\* When this source readable stream closes, destination will be closed,

\* unless preventClose is truthy. The returned promise will be fulfilled

\* once this process completes, unless an error is encountered while

\* closing the destination, in which case it will be rejected with that

\* error.

\*

\* If destination starts out closed or closing, this source readable

\* stream will be canceled, unless preventCancel is true. The returned

\* promise will be rejected with an error indicating piping to a closed

\* stream failed, or with any error that occurs during canceling the

\* source.

\*

\* The signal option can be set to an AbortSignal to allow aborting an

\* ongoing pipe operation via the corresponding AbortController. In this

\* case, this source readable stream will be canceled, and destination

\* aborted, unless the respective options preventCancel or preventAbort

\* are set.

\*/

preventClose?: boolean;

signal?: AbortSignal;

}

interface ReadableStreamGenericReader {

readonly closed: Promise<undefined>;

cancel(reason?: any): Promise<void>;

}

interface ReadableStreamDefaultReadValueResult<T> {

done: false;

value: T;

}

interface ReadableStreamDefaultReadDoneResult {

done: true;

value?: undefined;

}

type ReadableStreamController<T> = ReadableStreamDefaultController<T>;

type ReadableStreamDefaultReadResult<T> = ReadableStreamDefaultReadValueResult<T> | ReadableStreamDefaultReadDoneResult;

interface ReadableByteStreamControllerCallback {

(controller: ReadableByteStreamController): void | PromiseLike<void>;

}

interface UnderlyingSinkAbortCallback {

(reason?: any): void | PromiseLike<void>;

}

interface UnderlyingSinkCloseCallback {

(): void | PromiseLike<void>;

}

interface UnderlyingSinkStartCallback {

(controller: WritableStreamDefaultController): any;

}

interface UnderlyingSinkWriteCallback<W> {

(chunk: W, controller: WritableStreamDefaultController): void | PromiseLike<void>;

}

interface UnderlyingSourceCancelCallback {

(reason?: any): void | PromiseLike<void>;

}

interface UnderlyingSourcePullCallback<R> {

(controller: ReadableStreamController<R>): void | PromiseLike<void>;

}

interface UnderlyingSourceStartCallback<R> {

(controller: ReadableStreamController<R>): any;

}

interface TransformerFlushCallback<O> {

(controller: TransformStreamDefaultController<O>): void | PromiseLike<void>;

}

interface TransformerStartCallback<O> {

(controller: TransformStreamDefaultController<O>): any;

}

interface TransformerTransformCallback<I, O> {

(chunk: I, controller: TransformStreamDefaultController<O>): void | PromiseLike<void>;

}

interface UnderlyingByteSource {

autoAllocateChunkSize?: number;

cancel?: ReadableStreamErrorCallback;

pull?: ReadableByteStreamControllerCallback;

start?: ReadableByteStreamControllerCallback;

type: 'bytes';

}

interface UnderlyingSource<R = any> {

cancel?: UnderlyingSourceCancelCallback;

pull?: UnderlyingSourcePullCallback<R>;

start?: UnderlyingSourceStartCallback<R>;

type?: undefined;

}

interface UnderlyingSink<W = any> {

abort?: UnderlyingSinkAbortCallback;

close?: UnderlyingSinkCloseCallback;

start?: UnderlyingSinkStartCallback;

type?: undefined;

write?: UnderlyingSinkWriteCallback<W>;

}

interface ReadableStreamErrorCallback {

(reason: any): void | PromiseLike<void>;

}

/\*\* This Streams API interface represents a readable stream of byte data. \*/

interface ReadableStream<R = any> {

readonly locked: boolean;

cancel(reason?: any): Promise<void>;

getReader(): ReadableStreamDefaultReader<R>;

pipeThrough<T>(transform: ReadableWritablePair<T, R>, options?: StreamPipeOptions): ReadableStream<T>;

pipeTo(destination: WritableStream<R>, options?: StreamPipeOptions): Promise<void>;

tee(): [ReadableStream<R>, ReadableStream<R>];

[Symbol.asyncIterator](options?: { preventCancel?: boolean }): AsyncIterableIterator<R>;

}

const ReadableStream: {

prototype: ReadableStream;

new (underlyingSource: UnderlyingByteSource, strategy?: QueuingStrategy<Uint8Array>): ReadableStream<Uint8Array>;

new <R = any>(underlyingSource?: UnderlyingSource<R>, strategy?: QueuingStrategy<R>): ReadableStream<R>;

};

interface ReadableStreamDefaultReader<R = any> extends ReadableStreamGenericReader {

read(): Promise<ReadableStreamDefaultReadResult<R>>;

releaseLock(): void;

}

const ReadableStreamDefaultReader: {

prototype: ReadableStreamDefaultReader;

new <R = any>(stream: ReadableStream<R>): ReadableStreamDefaultReader<R>;

};

const ReadableStreamBYOBReader: any;

const ReadableStreamBYOBRequest: any;

interface ReadableByteStreamController {

readonly byobRequest: undefined;

readonly desiredSize: number | null;

close(): void;

enqueue(chunk: ArrayBufferView): void;

error(error?: any): void;

}

const ReadableByteStreamController: {

prototype: ReadableByteStreamController;

new (): ReadableByteStreamController;

};

interface ReadableStreamDefaultController<R = any> {

readonly desiredSize: number | null;

close(): void;

enqueue(chunk?: R): void;

error(e?: any): void;

}

const ReadableStreamDefaultController: {

prototype: ReadableStreamDefaultController;

new (): ReadableStreamDefaultController;

};

interface Transformer<I = any, O = any> {

flush?: TransformerFlushCallback<O>;

readableType?: undefined;

start?: TransformerStartCallback<O>;

transform?: TransformerTransformCallback<I, O>;

writableType?: undefined;

}

interface TransformStream<I = any, O = any> {

readonly readable: ReadableStream<O>;

readonly writable: WritableStream<I>;

}

const TransformStream: {

prototype: TransformStream;

new <I = any, O = any>(transformer?: Transformer<I, O>, writableStrategy?: QueuingStrategy<I>, readableStrategy?: QueuingStrategy<O>): TransformStream<I, O>;

};

interface TransformStreamDefaultController<O = any> {

readonly desiredSize: number | null;

enqueue(chunk?: O): void;

error(reason?: any): void;

terminate(): void;

}

const TransformStreamDefaultController: {

prototype: TransformStreamDefaultController;

new (): TransformStreamDefaultController;

};

/\*\*

\* This Streams API interface provides a standard abstraction for writing

\* streaming data to a destination, known as a sink. This object comes with

\* built-in back pressure and queuing.

\*/

interface WritableStream<W = any> {

readonly locked: boolean;

abort(reason?: any): Promise<void>;

close(): Promise<void>;

getWriter(): WritableStreamDefaultWriter<W>;

}

const WritableStream: {

prototype: WritableStream;

new <W = any>(underlyingSink?: UnderlyingSink<W>, strategy?: QueuingStrategy<W>): WritableStream<W>;

};

/\*\*

\* This Streams API interface is the object returned by

\* WritableStream.getWriter() and once created locks the < writer to the

\* WritableStream ensuring that no other streams can write to the underlying

\* sink.

\*/

interface WritableStreamDefaultWriter<W = any> {

readonly closed: Promise<undefined>;

readonly desiredSize: number | null;

readonly ready: Promise<undefined>;

abort(reason?: any): Promise<void>;

close(): Promise<void>;

releaseLock(): void;

write(chunk?: W): Promise<void>;

}

const WritableStreamDefaultWriter: {

prototype: WritableStreamDefaultWriter;

new <W = any>(stream: WritableStream<W>): WritableStreamDefaultWriter<W>;

};

/\*\*

\* This Streams API interface represents a controller allowing control of a

\* WritableStream's state. When constructing a WritableStream, the

\* underlying sink is given a corresponding WritableStreamDefaultController

\* instance to manipulate.

\*/

interface WritableStreamDefaultController {

error(e?: any): void;

}

const WritableStreamDefaultController: {

prototype: WritableStreamDefaultController;

new (): WritableStreamDefaultController;

};

interface QueuingStrategy<T = any> {

highWaterMark?: number;

size?: QueuingStrategySize<T>;

}

interface QueuingStrategySize<T = any> {

(chunk?: T): number;

}

interface QueuingStrategyInit {

/\*\*

\* Creates a new ByteLengthQueuingStrategy with the provided high water

\* mark.

\*

\* Note that the provided high water mark will not be validated ahead of

\* time. Instead, if it is negative, NaN, or not a number, the resulting

\* ByteLengthQueuingStrategy will cause the corresponding stream

\* constructor to throw.

\*/

highWaterMark: number;

}

/\*\*

\* This Streams API interface provides a built-in byte length queuing

\* strategy that can be used when constructing streams.

\*/

interface ByteLengthQueuingStrategy extends QueuingStrategy<ArrayBufferView> {

readonly highWaterMark: number;

readonly size: QueuingStrategySize<ArrayBufferView>;

}

const ByteLengthQueuingStrategy: {

prototype: ByteLengthQueuingStrategy;

new (init: QueuingStrategyInit): ByteLengthQueuingStrategy;

};

/\*\*

\* This Streams API interface provides a built-in byte length queuing

\* strategy that can be used when constructing streams.

\*/

interface CountQueuingStrategy extends QueuingStrategy {

readonly highWaterMark: number;

readonly size: QueuingStrategySize;

}

const CountQueuingStrategy: {

prototype: CountQueuingStrategy;

new (init: QueuingStrategyInit): CountQueuingStrategy;

};

interface TextEncoderStream {

/\*\* Returns "utf-8". \*/

readonly encoding: 'utf-8';

readonly readable: ReadableStream<Uint8Array>;

readonly writable: WritableStream<string>;

readonly [Symbol.toStringTag]: string;

}

const TextEncoderStream: {

prototype: TextEncoderStream;

new (): TextEncoderStream;

};

interface TextDecoderOptions {

fatal?: boolean;

ignoreBOM?: boolean;

}

type BufferSource = ArrayBufferView | ArrayBuffer;

interface TextDecoderStream {

/\*\* Returns encoding's name, lower cased. \*/

readonly encoding: string;

/\*\* Returns `true` if error mode is "fatal", and `false` otherwise. \*/

readonly fatal: boolean;

/\*\* Returns `true` if ignore BOM flag is set, and `false` otherwise. \*/

readonly ignoreBOM: boolean;

readonly readable: ReadableStream<string>;

readonly writable: WritableStream<BufferSource>;

readonly [Symbol.toStringTag]: string;

}

const TextDecoderStream: {

prototype: TextDecoderStream;

new (label?: string, options?: TextDecoderOptions): TextDecoderStream;

};

}

declare module 'node:stream/web' {

export \* from 'stream/web';

}