/\*!

JSZip v3.2.1 - A JavaScript class for generating and reading zip files

<http://stuartk.com/jszip>

(c) 2009-2016 Stuart Knightley <stuart [at] stuartk.com>

Dual licenced under the MIT license or GPLv3. See https://raw.github.com/Stuk/jszip/master/LICENSE.markdown.

JSZip uses the library pako released under the MIT license :

https://github.com/nodeca/pako/blob/master/LICENSE

\*/

(function(f){if(typeof exports==="object"&&typeof module!=="undefined"){module.exports=f()}else if(typeof define==="function"&&define.amd){define([],f)}else{var g;if(typeof window!=="undefined"){g=window}else if(typeof global!=="undefined"){g=global}else if(typeof self!=="undefined"){g=self}else{g=this}g.JSZip = f()}})(function(){var define,module,exports;return (function e(t,n,r){function s(o,u){if(!n[o]){if(!t[o]){var a=typeof require=="function"&&require;if(!u&&a)return a(o,!0);if(i)return i(o,!0);var f=new Error("Cannot find module '"+o+"'");throw f.code="MODULE\_NOT\_FOUND",f}var l=n[o]={exports:{}};t[o][0].call(l.exports,function(e){var n=t[o][1][e];return s(n?n:e)},l,l.exports,e,t,n,r)}return n[o].exports}var i=typeof require=="function"&&require;for(var o=0;o<r.length;o++)s(r[o]);return s})({1:[function(require,module,exports){

'use strict';

var utils = require('./utils');

var support = require('./support');

// private property

var \_keyStr = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/=";

// public method for encoding

exports.encode = function(input) {

var output = [];

var chr1, chr2, chr3, enc1, enc2, enc3, enc4;

var i = 0, len = input.length, remainingBytes = len;

var isArray = utils.getTypeOf(input) !== "string";

while (i < input.length) {

remainingBytes = len - i;

if (!isArray) {

chr1 = input.charCodeAt(i++);

chr2 = i < len ? input.charCodeAt(i++) : 0;

chr3 = i < len ? input.charCodeAt(i++) : 0;

} else {

chr1 = input[i++];

chr2 = i < len ? input[i++] : 0;

chr3 = i < len ? input[i++] : 0;

}

enc1 = chr1 >> 2;

enc2 = ((chr1 & 3) << 4) | (chr2 >> 4);

enc3 = remainingBytes > 1 ? (((chr2 & 15) << 2) | (chr3 >> 6)) : 64;

enc4 = remainingBytes > 2 ? (chr3 & 63) : 64;

output.push(\_keyStr.charAt(enc1) + \_keyStr.charAt(enc2) + \_keyStr.charAt(enc3) + \_keyStr.charAt(enc4));

}

return output.join("");

};

// public method for decoding

exports.decode = function(input) {

var chr1, chr2, chr3;

var enc1, enc2, enc3, enc4;

var i = 0, resultIndex = 0;

var dataUrlPrefix = "data:";

if (input.substr(0, dataUrlPrefix.length) === dataUrlPrefix) {

// This is a common error: people give a data url

// (...) with a {base64: true} and

// wonders why things don't work.

// We can detect that the string input looks like a data url but we

// \*can't\* be sure it is one: removing everything up to the comma would

// be too dangerous.

throw new Error("Invalid base64 input, it looks like a data url.");

}

input = input.replace(/[^A-Za-z0-9\+\/\=]/g, "");

var totalLength = input.length \* 3 / 4;

if(input.charAt(input.length - 1) === \_keyStr.charAt(64)) {

totalLength--;

}

if(input.charAt(input.length - 2) === \_keyStr.charAt(64)) {

totalLength--;

}

if (totalLength % 1 !== 0) {

// totalLength is not an integer, the length does not match a valid

// base64 content. That can happen if:

// - the input is not a base64 content

// - the input is \*almost\* a base64 content, with a extra chars at the

// beginning or at the end

// - the input uses a base64 variant (base64url for example)

throw new Error("Invalid base64 input, bad content length.");

}

var output;

if (support.uint8array) {

output = new Uint8Array(totalLength|0);

} else {

output = new Array(totalLength|0);

}

while (i < input.length) {

enc1 = \_keyStr.indexOf(input.charAt(i++));

enc2 = \_keyStr.indexOf(input.charAt(i++));

enc3 = \_keyStr.indexOf(input.charAt(i++));

enc4 = \_keyStr.indexOf(input.charAt(i++));

chr1 = (enc1 << 2) | (enc2 >> 4);

chr2 = ((enc2 & 15) << 4) | (enc3 >> 2);

chr3 = ((enc3 & 3) << 6) | enc4;

output[resultIndex++] = chr1;

if (enc3 !== 64) {

output[resultIndex++] = chr2;

}

if (enc4 !== 64) {

output[resultIndex++] = chr3;

}

}

return output;

};

},{"./support":30,"./utils":32}],2:[function(require,module,exports){

'use strict';

var external = require("./external");

var DataWorker = require('./stream/DataWorker');

var DataLengthProbe = require('./stream/DataLengthProbe');

var Crc32Probe = require('./stream/Crc32Probe');

var DataLengthProbe = require('./stream/DataLengthProbe');

/\*\*

\* Represent a compressed object, with everything needed to decompress it.

\* @constructor

\* @param {number} compressedSize the size of the data compressed.

\* @param {number} uncompressedSize the size of the data after decompression.

\* @param {number} crc32 the crc32 of the decompressed file.

\* @param {object} compression the type of compression, see lib/compressions.js.

\* @param {String|ArrayBuffer|Uint8Array|Buffer} data the compressed data.

\*/

function CompressedObject(compressedSize, uncompressedSize, crc32, compression, data) {

this.compressedSize = compressedSize;

this.uncompressedSize = uncompressedSize;

this.crc32 = crc32;

this.compression = compression;

this.compressedContent = data;

}

CompressedObject.prototype = {

/\*\*

\* Create a worker to get the uncompressed content.

\* @return {GenericWorker} the worker.

\*/

getContentWorker : function () {

var worker = new DataWorker(external.Promise.resolve(this.compressedContent))

.pipe(this.compression.uncompressWorker())

.pipe(new DataLengthProbe("data\_length"));

var that = this;

worker.on("end", function () {

if(this.streamInfo['data\_length'] !== that.uncompressedSize) {

throw new Error("Bug : uncompressed data size mismatch");

}

});

return worker;

},

/\*\*

\* Create a worker to get the compressed content.

\* @return {GenericWorker} the worker.

\*/

getCompressedWorker : function () {

return new DataWorker(external.Promise.resolve(this.compressedContent))

.withStreamInfo("compressedSize", this.compressedSize)

.withStreamInfo("uncompressedSize", this.uncompressedSize)

.withStreamInfo("crc32", this.crc32)

.withStreamInfo("compression", this.compression)

;

}

};

/\*\*

\* Chain the given worker with other workers to compress the content with the

\* given compresion.

\* @param {GenericWorker} uncompressedWorker the worker to pipe.

\* @param {Object} compression the compression object.

\* @param {Object} compressionOptions the options to use when compressing.

\* @return {GenericWorker} the new worker compressing the content.

\*/

CompressedObject.createWorkerFrom = function (uncompressedWorker, compression, compressionOptions) {

return uncompressedWorker

.pipe(new Crc32Probe())

.pipe(new DataLengthProbe("uncompressedSize"))

.pipe(compression.compressWorker(compressionOptions))

.pipe(new DataLengthProbe("compressedSize"))

.withStreamInfo("compression", compression);

};

module.exports = CompressedObject;

},{"./external":6,"./stream/Crc32Probe":25,"./stream/DataLengthProbe":26,"./stream/DataWorker":27}],3:[function(require,module,exports){

'use strict';

var GenericWorker = require("./stream/GenericWorker");

exports.STORE = {

magic: "\x00\x00",

compressWorker : function (compressionOptions) {

return new GenericWorker("STORE compression");

},

uncompressWorker : function () {

return new GenericWorker("STORE decompression");

}

};

exports.DEFLATE = require('./flate');

},{"./flate":7,"./stream/GenericWorker":28}],4:[function(require,module,exports){

'use strict';

var utils = require('./utils');

/\*\*

\* The following functions come from pako, from pako/lib/zlib/crc32.js

\* released under the MIT license, see pako https://github.com/nodeca/pako/

\*/

// Use ordinary array, since untyped makes no boost here

function makeTable() {

var c, table = [];

for(var n =0; n < 256; n++){

c = n;

for(var k =0; k < 8; k++){

c = ((c&1) ? (0xEDB88320 ^ (c >>> 1)) : (c >>> 1));

}

table[n] = c;

}

return table;

}

// Create table on load. Just 255 signed longs. Not a problem.

var crcTable = makeTable();

function crc32(crc, buf, len, pos) {

var t = crcTable, end = pos + len;

crc = crc ^ (-1);

for (var i = pos; i < end; i++ ) {

crc = (crc >>> 8) ^ t[(crc ^ buf[i]) & 0xFF];

}

return (crc ^ (-1)); // >>> 0;

}

// That's all for the pako functions.

/\*\*

\* Compute the crc32 of a string.

\* This is almost the same as the function crc32, but for strings. Using the

\* same function for the two use cases leads to horrible performances.

\* @param {Number} crc the starting value of the crc.

\* @param {String} str the string to use.

\* @param {Number} len the length of the string.

\* @param {Number} pos the starting position for the crc32 computation.

\* @return {Number} the computed crc32.

\*/

function crc32str(crc, str, len, pos) {

var t = crcTable, end = pos + len;

crc = crc ^ (-1);

for (var i = pos; i < end; i++ ) {

crc = (crc >>> 8) ^ t[(crc ^ str.charCodeAt(i)) & 0xFF];

}

return (crc ^ (-1)); // >>> 0;

}

module.exports = function crc32wrapper(input, crc) {

if (typeof input === "undefined" || !input.length) {

return 0;

}

var isArray = utils.getTypeOf(input) !== "string";

if(isArray) {

return crc32(crc|0, input, input.length, 0);

} else {

return crc32str(crc|0, input, input.length, 0);

}

};

},{"./utils":32}],5:[function(require,module,exports){

'use strict';

exports.base64 = false;

exports.binary = false;

exports.dir = false;

exports.createFolders = true;

exports.date = null;

exports.compression = null;

exports.compressionOptions = null;

exports.comment = null;

exports.unixPermissions = null;

exports.dosPermissions = null;

},{}],6:[function(require,module,exports){

/\* global Promise \*/

'use strict';

// load the global object first:

// - it should be better integrated in the system (unhandledRejection in node)

// - the environment may have a custom Promise implementation (see zone.js)

var ES6Promise = null;

if (typeof Promise !== "undefined") {

ES6Promise = Promise;

} else {

ES6Promise = require("lie");

}

/\*\*

\* Let the user use/change some implementations.

\*/

module.exports = {

Promise: ES6Promise

};

},{"lie":37}],7:[function(require,module,exports){

'use strict';

var USE\_TYPEDARRAY = (typeof Uint8Array !== 'undefined') && (typeof Uint16Array !== 'undefined') && (typeof Uint32Array !== 'undefined');

var pako = require("pako");

var utils = require("./utils");

var GenericWorker = require("./stream/GenericWorker");

var ARRAY\_TYPE = USE\_TYPEDARRAY ? "uint8array" : "array";

exports.magic = "\x08\x00";

/\*\*

\* Create a worker that uses pako to inflate/deflate.

\* @constructor

\* @param {String} action the name of the pako function to call : either "Deflate" or "Inflate".

\* @param {Object} options the options to use when (de)compressing.

\*/

function FlateWorker(action, options) {

GenericWorker.call(this, "FlateWorker/" + action);

this.\_pako = null;

this.\_pakoAction = action;

this.\_pakoOptions = options;

// the `meta` object from the last chunk received

// this allow this worker to pass around metadata

this.meta = {};

}

utils.inherits(FlateWorker, GenericWorker);

/\*\*

\* @see GenericWorker.processChunk

\*/

FlateWorker.prototype.processChunk = function (chunk) {

this.meta = chunk.meta;

if (this.\_pako === null) {

this.\_createPako();

}

this.\_pako.push(utils.transformTo(ARRAY\_TYPE, chunk.data), false);

};

/\*\*

\* @see GenericWorker.flush

\*/

FlateWorker.prototype.flush = function () {

GenericWorker.prototype.flush.call(this);

if (this.\_pako === null) {

this.\_createPako();

}

this.\_pako.push([], true);

};

/\*\*

\* @see GenericWorker.cleanUp

\*/

FlateWorker.prototype.cleanUp = function () {

GenericWorker.prototype.cleanUp.call(this);

this.\_pako = null;

};

/\*\*

\* Create the \_pako object.

\* TODO: lazy-loading this object isn't the best solution but it's the

\* quickest. The best solution is to lazy-load the worker list. See also the

\* issue #446.

\*/

FlateWorker.prototype.\_createPako = function () {

this.\_pako = new pako[this.\_pakoAction]({

raw: true,

level: this.\_pakoOptions.level || -1 // default compression

});

var self = this;

this.\_pako.onData = function(data) {

self.push({

data : data,

meta : self.meta

});

};

};

exports.compressWorker = function (compressionOptions) {

return new FlateWorker("Deflate", compressionOptions);

};

exports.uncompressWorker = function () {

return new FlateWorker("Inflate", {});

};

},{"./stream/GenericWorker":28,"./utils":32,"pako":38}],8:[function(require,module,exports){

'use strict';

var utils = require('../utils');

var GenericWorker = require('../stream/GenericWorker');

var utf8 = require('../utf8');

var crc32 = require('../crc32');

var signature = require('../signature');

/\*\*

\* Transform an integer into a string in hexadecimal.

\* @private

\* @param {number} dec the number to convert.

\* @param {number} bytes the number of bytes to generate.

\* @returns {string} the result.

\*/

var decToHex = function(dec, bytes) {

var hex = "", i;

for (i = 0; i < bytes; i++) {

hex += String.fromCharCode(dec & 0xff);

dec = dec >>> 8;

}

return hex;

};

/\*\*

\* Generate the UNIX part of the external file attributes.

\* @param {Object} unixPermissions the unix permissions or null.

\* @param {Boolean} isDir true if the entry is a directory, false otherwise.

\* @return {Number} a 32 bit integer.

\*

\* adapted from http://unix.stackexchange.com/questions/14705/the-zip-formats-external-file-attribute :

\*

\* TTTTsstrwxrwxrwx0000000000ADVSHR

\* ^^^^\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ file type, see zipinfo.c (UNX\_\*)

\* ^^^\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ setuid, setgid, sticky

\* ^^^^^^^^^\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ permissions

\* ^^^^^^^^^^\_\_\_\_\_\_ not used ?

\* ^^^^^^ DOS attribute bits : Archive, Directory, Volume label, System file, Hidden, Read only

\*/

var generateUnixExternalFileAttr = function (unixPermissions, isDir) {

var result = unixPermissions;

if (!unixPermissions) {

// I can't use octal values in strict mode, hence the hexa.

// 040775 => 0x41fd

// 0100664 => 0x81b4

result = isDir ? 0x41fd : 0x81b4;

}

return (result & 0xFFFF) << 16;

};

/\*\*

\* Generate the DOS part of the external file attributes.

\* @param {Object} dosPermissions the dos permissions or null.

\* @param {Boolean} isDir true if the entry is a directory, false otherwise.

\* @return {Number} a 32 bit integer.

\*

\* Bit 0 Read-Only

\* Bit 1 Hidden

\* Bit 2 System

\* Bit 3 Volume Label

\* Bit 4 Directory

\* Bit 5 Archive

\*/

var generateDosExternalFileAttr = function (dosPermissions, isDir) {

// the dir flag is already set for compatibility

return (dosPermissions || 0) & 0x3F;

};

/\*\*

\* Generate the various parts used in the construction of the final zip file.

\* @param {Object} streamInfo the hash with informations about the compressed file.

\* @param {Boolean} streamedContent is the content streamed ?

\* @param {Boolean} streamingEnded is the stream finished ?

\* @param {number} offset the current offset from the start of the zip file.

\* @param {String} platform let's pretend we are this platform (change platform dependents fields)

\* @param {Function} encodeFileName the function to encode the file name / comment.

\* @return {Object} the zip parts.

\*/

var generateZipParts = function(streamInfo, streamedContent, streamingEnded, offset, platform, encodeFileName) {

var file = streamInfo['file'],

compression = streamInfo['compression'],

useCustomEncoding = encodeFileName !== utf8.utf8encode,

encodedFileName = utils.transformTo("string", encodeFileName(file.name)),

utfEncodedFileName = utils.transformTo("string", utf8.utf8encode(file.name)),

comment = file.comment,

encodedComment = utils.transformTo("string", encodeFileName(comment)),

utfEncodedComment = utils.transformTo("string", utf8.utf8encode(comment)),

useUTF8ForFileName = utfEncodedFileName.length !== file.name.length,

useUTF8ForComment = utfEncodedComment.length !== comment.length,

dosTime,

dosDate,

extraFields = "",

unicodePathExtraField = "",

unicodeCommentExtraField = "",

dir = file.dir,

date = file.date;

var dataInfo = {

crc32 : 0,

compressedSize : 0,

uncompressedSize : 0

};

// if the content is streamed, the sizes/crc32 are only available AFTER

// the end of the stream.

if (!streamedContent || streamingEnded) {

dataInfo.crc32 = streamInfo['crc32'];

dataInfo.compressedSize = streamInfo['compressedSize'];

dataInfo.uncompressedSize = streamInfo['uncompressedSize'];

}

var bitflag = 0;

if (streamedContent) {

// Bit 3: the sizes/crc32 are set to zero in the local header.

// The correct values are put in the data descriptor immediately

// following the compressed data.

bitflag |= 0x0008;

}

if (!useCustomEncoding && (useUTF8ForFileName || useUTF8ForComment)) {

// Bit 11: Language encoding flag (EFS).

bitflag |= 0x0800;

}

var extFileAttr = 0;

var versionMadeBy = 0;

if (dir) {

// dos or unix, we set the dos dir flag

extFileAttr |= 0x00010;

}

if(platform === "UNIX") {

versionMadeBy = 0x031E; // UNIX, version 3.0

extFileAttr |= generateUnixExternalFileAttr(file.unixPermissions, dir);

} else { // DOS or other, fallback to DOS

versionMadeBy = 0x0014; // DOS, version 2.0

extFileAttr |= generateDosExternalFileAttr(file.dosPermissions, dir);

}

// date

// @see http://www.delorie.com/djgpp/doc/rbinter/it/52/13.html

// @see http://www.delorie.com/djgpp/doc/rbinter/it/65/16.html

// @see http://www.delorie.com/djgpp/doc/rbinter/it/66/16.html

dosTime = date.getUTCHours();

dosTime = dosTime << 6;

dosTime = dosTime | date.getUTCMinutes();

dosTime = dosTime << 5;

dosTime = dosTime | date.getUTCSeconds() / 2;

dosDate = date.getUTCFullYear() - 1980;

dosDate = dosDate << 4;

dosDate = dosDate | (date.getUTCMonth() + 1);

dosDate = dosDate << 5;

dosDate = dosDate | date.getUTCDate();

if (useUTF8ForFileName) {

// set the unicode path extra field. unzip needs at least one extra

// field to correctly handle unicode path, so using the path is as good

// as any other information. This could improve the situation with

// other archive managers too.

// This field is usually used without the utf8 flag, with a non

// unicode path in the header (winrar, winzip). This helps (a bit)

// with the messy Windows' default compressed folders feature but

// breaks on p7zip which doesn't seek the unicode path extra field.

// So for now, UTF-8 everywhere !

unicodePathExtraField =

// Version

decToHex(1, 1) +

// NameCRC32

decToHex(crc32(encodedFileName), 4) +

// UnicodeName

utfEncodedFileName;

extraFields +=

// Info-ZIP Unicode Path Extra Field

"\x75\x70" +

// size

decToHex(unicodePathExtraField.length, 2) +

// content

unicodePathExtraField;

}

if(useUTF8ForComment) {

unicodeCommentExtraField =

// Version

decToHex(1, 1) +

// CommentCRC32

decToHex(crc32(encodedComment), 4) +

// UnicodeName

utfEncodedComment;

extraFields +=

// Info-ZIP Unicode Path Extra Field

"\x75\x63" +

// size

decToHex(unicodeCommentExtraField.length, 2) +

// content

unicodeCommentExtraField;

}

var header = "";

// version needed to extract

header += "\x0A\x00";

// general purpose bit flag

header += decToHex(bitflag, 2);

// compression method

header += compression.magic;

// last mod file time

header += decToHex(dosTime, 2);

// last mod file date

header += decToHex(dosDate, 2);

// crc-32

header += decToHex(dataInfo.crc32, 4);

// compressed size

header += decToHex(dataInfo.compressedSize, 4);

// uncompressed size

header += decToHex(dataInfo.uncompressedSize, 4);

// file name length

header += decToHex(encodedFileName.length, 2);

// extra field length

header += decToHex(extraFields.length, 2);

var fileRecord = signature.LOCAL\_FILE\_HEADER + header + encodedFileName + extraFields;

var dirRecord = signature.CENTRAL\_FILE\_HEADER +

// version made by (00: DOS)

decToHex(versionMadeBy, 2) +

// file header (common to file and central directory)

header +

// file comment length

decToHex(encodedComment.length, 2) +

// disk number start

"\x00\x00" +

// internal file attributes TODO

"\x00\x00" +

// external file attributes

decToHex(extFileAttr, 4) +

// relative offset of local header

decToHex(offset, 4) +

// file name

encodedFileName +

// extra field

extraFields +

// file comment

encodedComment;

return {

fileRecord: fileRecord,

dirRecord: dirRecord

};

};

/\*\*

\* Generate the EOCD record.

\* @param {Number} entriesCount the number of entries in the zip file.

\* @param {Number} centralDirLength the length (in bytes) of the central dir.

\* @param {Number} localDirLength the length (in bytes) of the local dir.

\* @param {String} comment the zip file comment as a binary string.

\* @param {Function} encodeFileName the function to encode the comment.

\* @return {String} the EOCD record.

\*/

var generateCentralDirectoryEnd = function (entriesCount, centralDirLength, localDirLength, comment, encodeFileName) {

var dirEnd = "";

var encodedComment = utils.transformTo("string", encodeFileName(comment));

// end of central dir signature

dirEnd = signature.CENTRAL\_DIRECTORY\_END +

// number of this disk

"\x00\x00" +

// number of the disk with the start of the central directory

"\x00\x00" +

// total number of entries in the central directory on this disk

decToHex(entriesCount, 2) +

// total number of entries in the central directory

decToHex(entriesCount, 2) +

// size of the central directory 4 bytes

decToHex(centralDirLength, 4) +

// offset of start of central directory with respect to the starting disk number

decToHex(localDirLength, 4) +

// .ZIP file comment length

decToHex(encodedComment.length, 2) +

// .ZIP file comment

encodedComment;

return dirEnd;

};

/\*\*

\* Generate data descriptors for a file entry.

\* @param {Object} streamInfo the hash generated by a worker, containing informations

\* on the file entry.

\* @return {String} the data descriptors.

\*/

var generateDataDescriptors = function (streamInfo) {

var descriptor = "";

descriptor = signature.DATA\_DESCRIPTOR +

// crc-32 4 bytes

decToHex(streamInfo['crc32'], 4) +

// compressed size 4 bytes

decToHex(streamInfo['compressedSize'], 4) +

// uncompressed size 4 bytes

decToHex(streamInfo['uncompressedSize'], 4);

return descriptor;

};

/\*\*

\* A worker to concatenate other workers to create a zip file.

\* @param {Boolean} streamFiles `true` to stream the content of the files,

\* `false` to accumulate it.

\* @param {String} comment the comment to use.

\* @param {String} platform the platform to use, "UNIX" or "DOS".

\* @param {Function} encodeFileName the function to encode file names and comments.

\*/

function ZipFileWorker(streamFiles, comment, platform, encodeFileName) {

GenericWorker.call(this, "ZipFileWorker");

// The number of bytes written so far. This doesn't count accumulated chunks.

this.bytesWritten = 0;

// The comment of the zip file

this.zipComment = comment;

// The platform "generating" the zip file.

this.zipPlatform = platform;

// the function to encode file names and comments.

this.encodeFileName = encodeFileName;

// Should we stream the content of the files ?

this.streamFiles = streamFiles;

// If `streamFiles` is false, we will need to accumulate the content of the

// files to calculate sizes / crc32 (and write them \*before\* the content).

// This boolean indicates if we are accumulating chunks (it will change a lot

// during the lifetime of this worker).

this.accumulate = false;

// The buffer receiving chunks when accumulating content.

this.contentBuffer = [];

// The list of generated directory records.

this.dirRecords = [];

// The offset (in bytes) from the beginning of the zip file for the current source.

this.currentSourceOffset = 0;

// The total number of entries in this zip file.

this.entriesCount = 0;

// the name of the file currently being added, null when handling the end of the zip file.

// Used for the emited metadata.

this.currentFile = null;

this.\_sources = [];

}

utils.inherits(ZipFileWorker, GenericWorker);

/\*\*

\* @see GenericWorker.push

\*/

ZipFileWorker.prototype.push = function (chunk) {

var currentFilePercent = chunk.meta.percent || 0;

var entriesCount = this.entriesCount;

var remainingFiles = this.\_sources.length;

if(this.accumulate) {

this.contentBuffer.push(chunk);

} else {

this.bytesWritten += chunk.data.length;

GenericWorker.prototype.push.call(this, {

data : chunk.data,

meta : {

currentFile : this.currentFile,

percent : entriesCount ? (currentFilePercent + 100 \* (entriesCount - remainingFiles - 1)) / entriesCount : 100

}

});

}

};

/\*\*

\* The worker started a new source (an other worker).

\* @param {Object} streamInfo the streamInfo object from the new source.

\*/

ZipFileWorker.prototype.openedSource = function (streamInfo) {

this.currentSourceOffset = this.bytesWritten;

this.currentFile = streamInfo['file'].name;

var streamedContent = this.streamFiles && !streamInfo['file'].dir;

// don't stream folders (because they don't have any content)

if(streamedContent) {

var record = generateZipParts(streamInfo, streamedContent, false, this.currentSourceOffset, this.zipPlatform, this.encodeFileName);

this.push({

data : record.fileRecord,

meta : {percent:0}

});

} else {

// we need to wait for the whole file before pushing anything

this.accumulate = true;

}

};

/\*\*

\* The worker finished a source (an other worker).

\* @param {Object} streamInfo the streamInfo object from the finished source.

\*/

ZipFileWorker.prototype.closedSource = function (streamInfo) {

this.accumulate = false;

var streamedContent = this.streamFiles && !streamInfo['file'].dir;

var record = generateZipParts(streamInfo, streamedContent, true, this.currentSourceOffset, this.zipPlatform, this.encodeFileName);

this.dirRecords.push(record.dirRecord);

if(streamedContent) {

// after the streamed file, we put data descriptors

this.push({

data : generateDataDescriptors(streamInfo),

meta : {percent:100}

});

} else {

// the content wasn't streamed, we need to push everything now

// first the file record, then the content

this.push({

data : record.fileRecord,

meta : {percent:0}

});

while(this.contentBuffer.length) {

this.push(this.contentBuffer.shift());

}

}

this.currentFile = null;

};

/\*\*

\* @see GenericWorker.flush

\*/

ZipFileWorker.prototype.flush = function () {

var localDirLength = this.bytesWritten;

for(var i = 0; i < this.dirRecords.length; i++) {

this.push({

data : this.dirRecords[i],

meta : {percent:100}

});

}

var centralDirLength = this.bytesWritten - localDirLength;

var dirEnd = generateCentralDirectoryEnd(this.dirRecords.length, centralDirLength, localDirLength, this.zipComment, this.encodeFileName);

this.push({

data : dirEnd,

meta : {percent:100}

});

};

/\*\*

\* Prepare the next source to be read.

\*/

ZipFileWorker.prototype.prepareNextSource = function () {

this.previous = this.\_sources.shift();

this.openedSource(this.previous.streamInfo);

if (this.isPaused) {

this.previous.pause();

} else {

this.previous.resume();

}

};

/\*\*

\* @see GenericWorker.registerPrevious

\*/

ZipFileWorker.prototype.registerPrevious = function (previous) {

this.\_sources.push(previous);

var self = this;

previous.on('data', function (chunk) {

self.processChunk(chunk);

});

previous.on('end', function () {

self.closedSource(self.previous.streamInfo);

if(self.\_sources.length) {

self.prepareNextSource();

} else {

self.end();

}

});

previous.on('error', function (e) {

self.error(e);

});

return this;

};

/\*\*

\* @see GenericWorker.resume

\*/

ZipFileWorker.prototype.resume = function () {

if(!GenericWorker.prototype.resume.call(this)) {

return false;

}

if (!this.previous && this.\_sources.length) {

this.prepareNextSource();

return true;

}

if (!this.previous && !this.\_sources.length && !this.generatedError) {

this.end();

return true;

}

};

/\*\*

\* @see GenericWorker.error

\*/

ZipFileWorker.prototype.error = function (e) {

var sources = this.\_sources;

if(!GenericWorker.prototype.error.call(this, e)) {

return false;

}

for(var i = 0; i < sources.length; i++) {

try {

sources[i].error(e);

} catch(e) {

// the `error` exploded, nothing to do

}

}

return true;

};

/\*\*

\* @see GenericWorker.lock

\*/

ZipFileWorker.prototype.lock = function () {

GenericWorker.prototype.lock.call(this);

var sources = this.\_sources;

for(var i = 0; i < sources.length; i++) {

sources[i].lock();

}

};

module.exports = ZipFileWorker;

},{"../crc32":4,"../signature":23,"../stream/GenericWorker":28,"../utf8":31,"../utils":32}],9:[function(require,module,exports){

'use strict';

var compressions = require('../compressions');

var ZipFileWorker = require('./ZipFileWorker');

/\*\*

\* Find the compression to use.

\* @param {String} fileCompression the compression defined at the file level, if any.

\* @param {String} zipCompression the compression defined at the load() level.

\* @return {Object} the compression object to use.

\*/

var getCompression = function (fileCompression, zipCompression) {

var compressionName = fileCompression || zipCompression;

var compression = compressions[compressionName];

if (!compression) {

throw new Error(compressionName + " is not a valid compression method !");

}

return compression;

};

/\*\*

\* Create a worker to generate a zip file.

\* @param {JSZip} zip the JSZip instance at the right root level.

\* @param {Object} options to generate the zip file.

\* @param {String} comment the comment to use.

\*/

exports.generateWorker = function (zip, options, comment) {

var zipFileWorker = new ZipFileWorker(options.streamFiles, comment, options.platform, options.encodeFileName);

var entriesCount = 0;

try {

zip.forEach(function (relativePath, file) {

entriesCount++;

var compression = getCompression(file.options.compression, options.compression);

var compressionOptions = file.options.compressionOptions || options.compressionOptions || {};

var dir = file.dir, date = file.date;

file.\_compressWorker(compression, compressionOptions)

.withStreamInfo("file", {

name : relativePath,

dir : dir,

date : date,

comment : file.comment || "",

unixPermissions : file.unixPermissions,

dosPermissions : file.dosPermissions

})

.pipe(zipFileWorker);

});

zipFileWorker.entriesCount = entriesCount;

} catch (e) {

zipFileWorker.error(e);

}

return zipFileWorker;

};

},{"../compressions":3,"./ZipFileWorker":8}],10:[function(require,module,exports){

'use strict';

/\*\*

\* Representation a of zip file in js

\* @constructor

\*/

function JSZip() {

// if this constructor is used without `new`, it adds `new` before itself:

if(!(this instanceof JSZip)) {

return new JSZip();

}

if(arguments.length) {

throw new Error("The constructor with parameters has been removed in JSZip 3.0, please check the upgrade guide.");

}

// object containing the files :

// {

// "folder/" : {...},

// "folder/data.txt" : {...}

// }

this.files = {};

this.comment = null;

// Where we are in the hierarchy

this.root = "";

this.clone = function() {

var newObj = new JSZip();

for (var i in this) {

if (typeof this[i] !== "function") {

newObj[i] = this[i];

}

}

return newObj;

};

}

JSZip.prototype = require('./object');

JSZip.prototype.loadAsync = require('./load');

JSZip.support = require('./support');

JSZip.defaults = require('./defaults');

// TODO find a better way to handle this version,

// a require('package.json').version doesn't work with webpack, see #327

JSZip.version = "3.2.0";

JSZip.loadAsync = function (content, options) {

return new JSZip().loadAsync(content, options);

};

JSZip.external = require("./external");

module.exports = JSZip;

},{"./defaults":5,"./external":6,"./load":11,"./object":15,"./support":30}],11:[function(require,module,exports){

'use strict';

var utils = require('./utils');

var external = require("./external");

var utf8 = require('./utf8');

var utils = require('./utils');

var ZipEntries = require('./zipEntries');

var Crc32Probe = require('./stream/Crc32Probe');

var nodejsUtils = require("./nodejsUtils");

/\*\*

\* Check the CRC32 of an entry.

\* @param {ZipEntry} zipEntry the zip entry to check.

\* @return {Promise} the result.

\*/

function checkEntryCRC32(zipEntry) {

return new external.Promise(function (resolve, reject) {

var worker = zipEntry.decompressed.getContentWorker().pipe(new Crc32Probe());

worker.on("error", function (e) {

reject(e);

})

.on("end", function () {

if (worker.streamInfo.crc32 !== zipEntry.decompressed.crc32) {

reject(new Error("Corrupted zip : CRC32 mismatch"));

} else {

resolve();

}

})

.resume();

});

}

module.exports = function(data, options) {

var zip = this;

options = utils.extend(options || {}, {

base64: false,

checkCRC32: false,

optimizedBinaryString: false,

createFolders: false,

decodeFileName: utf8.utf8decode

});

if (nodejsUtils.isNode && nodejsUtils.isStream(data)) {

return external.Promise.reject(new Error("JSZip can't accept a stream when loading a zip file."));

}

return utils.prepareContent("the loaded zip file", data, true, options.optimizedBinaryString, options.base64)

.then(function(data) {

var zipEntries = new ZipEntries(options);

zipEntries.load(data);

return zipEntries;

}).then(function checkCRC32(zipEntries) {

var promises = [external.Promise.resolve(zipEntries)];

var files = zipEntries.files;

if (options.checkCRC32) {

for (var i = 0; i < files.length; i++) {

promises.push(checkEntryCRC32(files[i]));

}

}

return external.Promise.all(promises);

}).then(function addFiles(results) {

var zipEntries = results.shift();

var files = zipEntries.files;

for (var i = 0; i < files.length; i++) {

var input = files[i];

zip.file(input.fileNameStr, input.decompressed, {

binary: true,

optimizedBinaryString: true,

date: input.date,

dir: input.dir,

comment : input.fileCommentStr.length ? input.fileCommentStr : null,

unixPermissions : input.unixPermissions,

dosPermissions : input.dosPermissions,

createFolders: options.createFolders

});

}

if (zipEntries.zipComment.length) {

zip.comment = zipEntries.zipComment;

}

return zip;

});

};

},{"./external":6,"./nodejsUtils":14,"./stream/Crc32Probe":25,"./utf8":31,"./utils":32,"./zipEntries":33}],12:[function(require,module,exports){

"use strict";

var utils = require('../utils');

var GenericWorker = require('../stream/GenericWorker');

/\*\*

\* A worker that use a nodejs stream as source.

\* @constructor

\* @param {String} filename the name of the file entry for this stream.

\* @param {Readable} stream the nodejs stream.

\*/

function NodejsStreamInputAdapter(filename, stream) {

GenericWorker.call(this, "Nodejs stream input adapter for " + filename);

this.\_upstreamEnded = false;

this.\_bindStream(stream);

}

utils.inherits(NodejsStreamInputAdapter, GenericWorker);

/\*\*

\* Prepare the stream and bind the callbacks on it.

\* Do this ASAP on node 0.10 ! A lazy binding doesn't always work.

\* @param {Stream} stream the nodejs stream to use.

\*/

NodejsStreamInputAdapter.prototype.\_bindStream = function (stream) {

var self = this;

this.\_stream = stream;

stream.pause();

stream

.on("data", function (chunk) {

self.push({

data: chunk,

meta : {

percent : 0

}

});

})

.on("error", function (e) {

if(self.isPaused) {

this.generatedError = e;

} else {

self.error(e);

}

})

.on("end", function () {

if(self.isPaused) {

self.\_upstreamEnded = true;

} else {

self.end();

}

});

};

NodejsStreamInputAdapter.prototype.pause = function () {

if(!GenericWorker.prototype.pause.call(this)) {

return false;

}

this.\_stream.pause();

return true;

};

NodejsStreamInputAdapter.prototype.resume = function () {

if(!GenericWorker.prototype.resume.call(this)) {

return false;

}

if(this.\_upstreamEnded) {

this.end();

} else {

this.\_stream.resume();

}

return true;

};

module.exports = NodejsStreamInputAdapter;

},{"../stream/GenericWorker":28,"../utils":32}],13:[function(require,module,exports){

'use strict';

var Readable = require('readable-stream').Readable;

var utils = require('../utils');

utils.inherits(NodejsStreamOutputAdapter, Readable);

/\*\*

\* A nodejs stream using a worker as source.

\* @see the SourceWrapper in http://nodejs.org/api/stream.html

\* @constructor

\* @param {StreamHelper} helper the helper wrapping the worker

\* @param {Object} options the nodejs stream options

\* @param {Function} updateCb the update callback.

\*/

function NodejsStreamOutputAdapter(helper, options, updateCb) {

Readable.call(this, options);

this.\_helper = helper;

var self = this;

helper.on("data", function (data, meta) {

if (!self.push(data)) {

self.\_helper.pause();

}

if(updateCb) {

updateCb(meta);

}

})

.on("error", function(e) {

self.emit('error', e);

})

.on("end", function () {

self.push(null);

});

}

NodejsStreamOutputAdapter.prototype.\_read = function() {

this.\_helper.resume();

};

module.exports = NodejsStreamOutputAdapter;

},{"../utils":32,"readable-stream":16}],14:[function(require,module,exports){

'use strict';

module.exports = {

/\*\*

\* True if this is running in Nodejs, will be undefined in a browser.

\* In a browser, browserify won't include this file and the whole module

\* will be resolved an empty object.

\*/

isNode : typeof Buffer !== "undefined",

/\*\*

\* Create a new nodejs Buffer from an existing content.

\* @param {Object} data the data to pass to the constructor.

\* @param {String} encoding the encoding to use.

\* @return {Buffer} a new Buffer.

\*/

newBufferFrom: function(data, encoding) {

if (Buffer.from && Buffer.from !== Uint8Array.from) {

return Buffer.from(data, encoding);

} else {

if (typeof data === "number") {

// Safeguard for old Node.js versions. On newer versions,

// Buffer.from(number) / Buffer(number, encoding) already throw.

throw new Error("The \"data\" argument must not be a number");

}

return new Buffer(data, encoding);

}

},

/\*\*

\* Create a new nodejs Buffer with the specified size.

\* @param {Integer} size the size of the buffer.

\* @return {Buffer} a new Buffer.

\*/

allocBuffer: function (size) {

if (Buffer.alloc) {

return Buffer.alloc(size);

} else {

var buf = new Buffer(size);

buf.fill(0);

return buf;

}

},

/\*\*

\* Find out if an object is a Buffer.

\* @param {Object} b the object to test.

\* @return {Boolean} true if the object is a Buffer, false otherwise.

\*/

isBuffer : function(b){

return Buffer.isBuffer(b);

},

isStream : function (obj) {

return obj &&

typeof obj.on === "function" &&

typeof obj.pause === "function" &&

typeof obj.resume === "function";

}

};

},{}],15:[function(require,module,exports){

'use strict';

var utf8 = require('./utf8');

var utils = require('./utils');

var GenericWorker = require('./stream/GenericWorker');

var StreamHelper = require('./stream/StreamHelper');

var defaults = require('./defaults');

var CompressedObject = require('./compressedObject');

var ZipObject = require('./zipObject');

var generate = require("./generate");

var nodejsUtils = require("./nodejsUtils");

var NodejsStreamInputAdapter = require("./nodejs/NodejsStreamInputAdapter");

/\*\*

\* Add a file in the current folder.

\* @private

\* @param {string} name the name of the file

\* @param {String|ArrayBuffer|Uint8Array|Buffer} data the data of the file

\* @param {Object} originalOptions the options of the file

\* @return {Object} the new file.

\*/

var fileAdd = function(name, data, originalOptions) {

// be sure sub folders exist

var dataType = utils.getTypeOf(data),

parent;

/\*

\* Correct options.

\*/

var o = utils.extend(originalOptions || {}, defaults);

o.date = o.date || new Date();

if (o.compression !== null) {

o.compression = o.compression.toUpperCase();

}

if (typeof o.unixPermissions === "string") {

o.unixPermissions = parseInt(o.unixPermissions, 8);

}

// UNX\_IFDIR 0040000 see zipinfo.c

if (o.unixPermissions && (o.unixPermissions & 0x4000)) {

o.dir = true;

}

// Bit 4 Directory

if (o.dosPermissions && (o.dosPermissions & 0x0010)) {

o.dir = true;

}

if (o.dir) {

name = forceTrailingSlash(name);

}

if (o.createFolders && (parent = parentFolder(name))) {

folderAdd.call(this, parent, true);

}

var isUnicodeString = dataType === "string" && o.binary === false && o.base64 === false;

if (!originalOptions || typeof originalOptions.binary === "undefined") {

o.binary = !isUnicodeString;

}

var isCompressedEmpty = (data instanceof CompressedObject) && data.uncompressedSize === 0;

if (isCompressedEmpty || o.dir || !data || data.length === 0) {

o.base64 = false;

o.binary = true;

data = "";

o.compression = "STORE";

dataType = "string";

}

/\*

\* Convert content to fit.

\*/

var zipObjectContent = null;

if (data instanceof CompressedObject || data instanceof GenericWorker) {

zipObjectContent = data;

} else if (nodejsUtils.isNode && nodejsUtils.isStream(data)) {

zipObjectContent = new NodejsStreamInputAdapter(name, data);

} else {

zipObjectContent = utils.prepareContent(name, data, o.binary, o.optimizedBinaryString, o.base64);

}

var object = new ZipObject(name, zipObjectContent, o);

this.files[name] = object;

/\*

TODO: we can't throw an exception because we have async promises

(we can have a promise of a Date() for example) but returning a

promise is useless because file(name, data) returns the JSZip

object for chaining. Should we break that to allow the user

to catch the error ?

return external.Promise.resolve(zipObjectContent)

.then(function () {

return object;

});

\*/

};

/\*\*

\* Find the parent folder of the path.

\* @private

\* @param {string} path the path to use

\* @return {string} the parent folder, or ""

\*/

var parentFolder = function (path) {

if (path.slice(-1) === '/') {

path = path.substring(0, path.length - 1);

}

var lastSlash = path.lastIndexOf('/');

return (lastSlash > 0) ? path.substring(0, lastSlash) : "";

};

/\*\*

\* Returns the path with a slash at the end.

\* @private

\* @param {String} path the path to check.

\* @return {String} the path with a trailing slash.

\*/

var forceTrailingSlash = function(path) {

// Check the name ends with a /

if (path.slice(-1) !== "/") {

path += "/"; // IE doesn't like substr(-1)

}

return path;

};

/\*\*

\* Add a (sub) folder in the current folder.

\* @private

\* @param {string} name the folder's name

\* @param {boolean=} [createFolders] If true, automatically create sub

\* folders. Defaults to false.

\* @return {Object} the new folder.

\*/

var folderAdd = function(name, createFolders) {

createFolders = (typeof createFolders !== 'undefined') ? createFolders : defaults.createFolders;

name = forceTrailingSlash(name);

// Does this folder already exist?

if (!this.files[name]) {

fileAdd.call(this, name, null, {

dir: true,

createFolders: createFolders

});

}

return this.files[name];

};

/\*\*

\* Cross-window, cross-Node-context regular expression detection

\* @param {Object} object Anything

\* @return {Boolean} true if the object is a regular expression,

\* false otherwise

\*/

function isRegExp(object) {

return Object.prototype.toString.call(object) === "[object RegExp]";

}

// return the actual prototype of JSZip

var out = {

/\*\*

\* @see loadAsync

\*/

load: function() {

throw new Error("This method has been removed in JSZip 3.0, please check the upgrade guide.");

},

/\*\*

\* Call a callback function for each entry at this folder level.

\* @param {Function} cb the callback function:

\* function (relativePath, file) {...}

\* It takes 2 arguments : the relative path and the file.

\*/

forEach: function(cb) {

var filename, relativePath, file;

for (filename in this.files) {

if (!this.files.hasOwnProperty(filename)) {

continue;

}

file = this.files[filename];

relativePath = filename.slice(this.root.length, filename.length);

if (relativePath && filename.slice(0, this.root.length) === this.root) { // the file is in the current root

cb(relativePath, file); // TODO reverse the parameters ? need to be clean AND consistent with the filter search fn...

}

}

},

/\*\*

\* Filter nested files/folders with the specified function.

\* @param {Function} search the predicate to use :

\* function (relativePath, file) {...}

\* It takes 2 arguments : the relative path and the file.

\* @return {Array} An array of matching elements.

\*/

filter: function(search) {

var result = [];

this.forEach(function (relativePath, entry) {

if (search(relativePath, entry)) { // the file matches the function

result.push(entry);

}

});

return result;

},

/\*\*

\* Add a file to the zip file, or search a file.

\* @param {string|RegExp} name The name of the file to add (if data is defined),

\* the name of the file to find (if no data) or a regex to match files.

\* @param {String|ArrayBuffer|Uint8Array|Buffer} data The file data, either raw or base64 encoded

\* @param {Object} o File options

\* @return {JSZip|Object|Array} this JSZip object (when adding a file),

\* a file (when searching by string) or an array of files (when searching by regex).

\*/

file: function(name, data, o) {

if (arguments.length === 1) {

if (isRegExp(name)) {

var regexp = name;

return this.filter(function(relativePath, file) {

return !file.dir && regexp.test(relativePath);

});

}

else { // text

var obj = this.files[this.root + name];

if (obj && !obj.dir) {

return obj;

} else {

return null;

}

}

}

else { // more than one argument : we have data !

name = this.root + name;

fileAdd.call(this, name, data, o);

}

return this;

},

/\*\*

\* Add a directory to the zip file, or search.

\* @param {String|RegExp} arg The name of the directory to add, or a regex to search folders.

\* @return {JSZip} an object with the new directory as the root, or an array containing matching folders.

\*/

folder: function(arg) {

if (!arg) {

return this;

}

if (isRegExp(arg)) {

return this.filter(function(relativePath, file) {

return file.dir && arg.test(relativePath);

});

}

// else, name is a new folder

var name = this.root + arg;

var newFolder = folderAdd.call(this, name);

// Allow chaining by returning a new object with this folder as the root

var ret = this.clone();

ret.root = newFolder.name;

return ret;

},

/\*\*

\* Delete a file, or a directory and all sub-files, from the zip

\* @param {string} name the name of the file to delete

\* @return {JSZip} this JSZip object

\*/

remove: function(name) {

name = this.root + name;

var file = this.files[name];

if (!file) {

// Look for any folders

if (name.slice(-1) !== "/") {

name += "/";

}

file = this.files[name];

}

if (file && !file.dir) {

// file

delete this.files[name];

} else {

// maybe a folder, delete recursively

var kids = this.filter(function(relativePath, file) {

return file.name.slice(0, name.length) === name;

});

for (var i = 0; i < kids.length; i++) {

delete this.files[kids[i].name];

}

}

return this;

},

/\*\*

\* Generate the complete zip file

\* @param {Object} options the options to generate the zip file :

\* - compression, "STORE" by default.

\* - type, "base64" by default. Values are : string, base64, uint8array, arraybuffer, blob.

\* @return {String|Uint8Array|ArrayBuffer|Buffer|Blob} the zip file

\*/

generate: function(options) {

throw new Error("This method has been removed in JSZip 3.0, please check the upgrade guide.");

},

/\*\*

\* Generate the complete zip file as an internal stream.

\* @param {Object} options the options to generate the zip file :

\* - compression, "STORE" by default.

\* - type, "base64" by default. Values are : string, base64, uint8array, arraybuffer, blob.

\* @return {StreamHelper} the streamed zip file.

\*/

generateInternalStream: function(options) {

var worker, opts = {};

try {

opts = utils.extend(options || {}, {

streamFiles: false,

compression: "STORE",

compressionOptions : null,

type: "",

platform: "DOS",

comment: null,

mimeType: 'application/zip',

encodeFileName: utf8.utf8encode

});

opts.type = opts.type.toLowerCase();

opts.compression = opts.compression.toUpperCase();

// "binarystring" is prefered but the internals use "string".

if(opts.type === "binarystring") {

opts.type = "string";

}

if (!opts.type) {

throw new Error("No output type specified.");

}

utils.checkSupport(opts.type);

// accept nodejs `process.platform`

if(

opts.platform === 'darwin' ||

opts.platform === 'freebsd' ||

opts.platform === 'linux' ||

opts.platform === 'sunos'

) {

opts.platform = "UNIX";

}

if (opts.platform === 'win32') {

opts.platform = "DOS";

}

var comment = opts.comment || this.comment || "";

worker = generate.generateWorker(this, opts, comment);

} catch (e) {

worker = new GenericWorker("error");

worker.error(e);

}

return new StreamHelper(worker, opts.type || "string", opts.mimeType);

},

/\*\*

\* Generate the complete zip file asynchronously.

\* @see generateInternalStream

\*/

generateAsync: function(options, onUpdate) {

return this.generateInternalStream(options).accumulate(onUpdate);

},

/\*\*

\* Generate the complete zip file asynchronously.

\* @see generateInternalStream

\*/

generateNodeStream: function(options, onUpdate) {

options = options || {};

if (!options.type) {

options.type = "nodebuffer";

}

return this.generateInternalStream(options).toNodejsStream(onUpdate);

}

};

module.exports = out;

},{"./compressedObject":2,"./defaults":5,"./generate":9,"./nodejs/NodejsStreamInputAdapter":12,"./nodejsUtils":14,"./stream/GenericWorker":28,"./stream/StreamHelper":29,"./utf8":31,"./utils":32,"./zipObject":35}],16:[function(require,module,exports){

/\*

\* This file is used by module bundlers (browserify/webpack/etc) when

\* including a stream implementation. We use "readable-stream" to get a

\* consistent behavior between nodejs versions but bundlers often have a shim

\* for "stream". Using this shim greatly improve the compatibility and greatly

\* reduce the final size of the bundle (only one stream implementation, not

\* two).

\*/

module.exports = require("stream");

},{"stream":undefined}],17:[function(require,module,exports){

'use strict';

var DataReader = require('./DataReader');

var utils = require('../utils');

function ArrayReader(data) {

DataReader.call(this, data);

for(var i = 0; i < this.data.length; i++) {

data[i] = data[i] & 0xFF;

}

}

utils.inherits(ArrayReader, DataReader);

/\*\*

\* @see DataReader.byteAt

\*/

ArrayReader.prototype.byteAt = function(i) {

return this.data[this.zero + i];

};

/\*\*

\* @see DataReader.lastIndexOfSignature

\*/

ArrayReader.prototype.lastIndexOfSignature = function(sig) {

var sig0 = sig.charCodeAt(0),

sig1 = sig.charCodeAt(1),

sig2 = sig.charCodeAt(2),

sig3 = sig.charCodeAt(3);

for (var i = this.length - 4; i >= 0; --i) {

if (this.data[i] === sig0 && this.data[i + 1] === sig1 && this.data[i + 2] === sig2 && this.data[i + 3] === sig3) {

return i - this.zero;

}

}

return -1;

};

/\*\*

\* @see DataReader.readAndCheckSignature

\*/

ArrayReader.prototype.readAndCheckSignature = function (sig) {

var sig0 = sig.charCodeAt(0),

sig1 = sig.charCodeAt(1),

sig2 = sig.charCodeAt(2),

sig3 = sig.charCodeAt(3),

data = this.readData(4);

return sig0 === data[0] && sig1 === data[1] && sig2 === data[2] && sig3 === data[3];

};

/\*\*

\* @see DataReader.readData

\*/

ArrayReader.prototype.readData = function(size) {

this.checkOffset(size);

if(size === 0) {

return [];

}

var result = this.data.slice(this.zero + this.index, this.zero + this.index + size);

this.index += size;

return result;

};

module.exports = ArrayReader;

},{"../utils":32,"./DataReader":18}],18:[function(require,module,exports){

'use strict';

var utils = require('../utils');

function DataReader(data) {

this.data = data; // type : see implementation

this.length = data.length;

this.index = 0;

this.zero = 0;

}

DataReader.prototype = {

/\*\*

\* Check that the offset will not go too far.

\* @param {string} offset the additional offset to check.

\* @throws {Error} an Error if the offset is out of bounds.

\*/

checkOffset: function(offset) {

this.checkIndex(this.index + offset);

},

/\*\*

\* Check that the specified index will not be too far.

\* @param {string} newIndex the index to check.

\* @throws {Error} an Error if the index is out of bounds.

\*/

checkIndex: function(newIndex) {

if (this.length < this.zero + newIndex || newIndex < 0) {

throw new Error("End of data reached (data length = " + this.length + ", asked index = " + (newIndex) + "). Corrupted zip ?");

}

},

/\*\*

\* Change the index.

\* @param {number} newIndex The new index.

\* @throws {Error} if the new index is out of the data.

\*/

setIndex: function(newIndex) {

this.checkIndex(newIndex);

this.index = newIndex;

},

/\*\*

\* Skip the next n bytes.

\* @param {number} n the number of bytes to skip.

\* @throws {Error} if the new index is out of the data.

\*/

skip: function(n) {

this.setIndex(this.index + n);

},

/\*\*

\* Get the byte at the specified index.

\* @param {number} i the index to use.

\* @return {number} a byte.

\*/

byteAt: function(i) {

// see implementations

},

/\*\*

\* Get the next number with a given byte size.

\* @param {number} size the number of bytes to read.

\* @return {number} the corresponding number.

\*/

readInt: function(size) {

var result = 0,

i;

this.checkOffset(size);

for (i = this.index + size - 1; i >= this.index; i--) {

result = (result << 8) + this.byteAt(i);

}

this.index += size;

return result;

},

/\*\*

\* Get the next string with a given byte size.

\* @param {number} size the number of bytes to read.

\* @return {string} the corresponding string.

\*/

readString: function(size) {

return utils.transformTo("string", this.readData(size));

},

/\*\*

\* Get raw data without conversion, <size> bytes.

\* @param {number} size the number of bytes to read.

\* @return {Object} the raw data, implementation specific.

\*/

readData: function(size) {

// see implementations

},

/\*\*

\* Find the last occurence of a zip signature (4 bytes).

\* @param {string} sig the signature to find.

\* @return {number} the index of the last occurence, -1 if not found.

\*/

lastIndexOfSignature: function(sig) {

// see implementations

},

/\*\*

\* Read the signature (4 bytes) at the current position and compare it with sig.

\* @param {string} sig the expected signature

\* @return {boolean} true if the signature matches, false otherwise.

\*/

readAndCheckSignature: function(sig) {

// see implementations

},

/\*\*

\* Get the next date.

\* @return {Date} the date.

\*/

readDate: function() {

var dostime = this.readInt(4);

return new Date(Date.UTC(

((dostime >> 25) & 0x7f) + 1980, // year

((dostime >> 21) & 0x0f) - 1, // month

(dostime >> 16) & 0x1f, // day

(dostime >> 11) & 0x1f, // hour

(dostime >> 5) & 0x3f, // minute

(dostime & 0x1f) << 1)); // second

}

};

module.exports = DataReader;

},{"../utils":32}],19:[function(require,module,exports){

'use strict';

var Uint8ArrayReader = require('./Uint8ArrayReader');

var utils = require('../utils');

function NodeBufferReader(data) {

Uint8ArrayReader.call(this, data);

}

utils.inherits(NodeBufferReader, Uint8ArrayReader);

/\*\*

\* @see DataReader.readData

\*/

NodeBufferReader.prototype.readData = function(size) {

this.checkOffset(size);

var result = this.data.slice(this.zero + this.index, this.zero + this.index + size);

this.index += size;

return result;

};

module.exports = NodeBufferReader;

},{"../utils":32,"./Uint8ArrayReader":21}],20:[function(require,module,exports){

'use strict';

var DataReader = require('./DataReader');

var utils = require('../utils');

function StringReader(data) {

DataReader.call(this, data);

}

utils.inherits(StringReader, DataReader);

/\*\*

\* @see DataReader.byteAt

\*/

StringReader.prototype.byteAt = function(i) {

return this.data.charCodeAt(this.zero + i);

};

/\*\*

\* @see DataReader.lastIndexOfSignature

\*/

StringReader.prototype.lastIndexOfSignature = function(sig) {

return this.data.lastIndexOf(sig) - this.zero;

};

/\*\*

\* @see DataReader.readAndCheckSignature

\*/

StringReader.prototype.readAndCheckSignature = function (sig) {

var data = this.readData(4);

return sig === data;

};

/\*\*

\* @see DataReader.readData

\*/

StringReader.prototype.readData = function(size) {

this.checkOffset(size);

// this will work because the constructor applied the "& 0xff" mask.

var result = this.data.slice(this.zero + this.index, this.zero + this.index + size);

this.index += size;

return result;

};

module.exports = StringReader;

},{"../utils":32,"./DataReader":18}],21:[function(require,module,exports){

'use strict';

var ArrayReader = require('./ArrayReader');

var utils = require('../utils');

function Uint8ArrayReader(data) {

ArrayReader.call(this, data);

}

utils.inherits(Uint8ArrayReader, ArrayReader);

/\*\*

\* @see DataReader.readData

\*/

Uint8ArrayReader.prototype.readData = function(size) {

this.checkOffset(size);

if(size === 0) {

// in IE10, when using subarray(idx, idx), we get the array [0x00] instead of [].

return new Uint8Array(0);

}

var result = this.data.subarray(this.zero + this.index, this.zero + this.index + size);

this.index += size;

return result;

};

module.exports = Uint8ArrayReader;

},{"../utils":32,"./ArrayReader":17}],22:[function(require,module,exports){

'use strict';

var utils = require('../utils');

var support = require('../support');

var ArrayReader = require('./ArrayReader');

var StringReader = require('./StringReader');

var NodeBufferReader = require('./NodeBufferReader');

var Uint8ArrayReader = require('./Uint8ArrayReader');

/\*\*

\* Create a reader adapted to the data.

\* @param {String|ArrayBuffer|Uint8Array|Buffer} data the data to read.

\* @return {DataReader} the data reader.

\*/

module.exports = function (data) {

var type = utils.getTypeOf(data);

utils.checkSupport(type);

if (type === "string" && !support.uint8array) {

return new StringReader(data);

}

if (type === "nodebuffer") {

return new NodeBufferReader(data);

}

if (support.uint8array) {

return new Uint8ArrayReader(utils.transformTo("uint8array", data));

}

return new ArrayReader(utils.transformTo("array", data));

};

},{"../support":30,"../utils":32,"./ArrayReader":17,"./NodeBufferReader":19,"./StringReader":20,"./Uint8ArrayReader":21}],23:[function(require,module,exports){

'use strict';

exports.LOCAL\_FILE\_HEADER = "PK\x03\x04";

exports.CENTRAL\_FILE\_HEADER = "PK\x01\x02";

exports.CENTRAL\_DIRECTORY\_END = "PK\x05\x06";

exports.ZIP64\_CENTRAL\_DIRECTORY\_LOCATOR = "PK\x06\x07";

exports.ZIP64\_CENTRAL\_DIRECTORY\_END = "PK\x06\x06";

exports.DATA\_DESCRIPTOR = "PK\x07\x08";

},{}],24:[function(require,module,exports){

'use strict';

var GenericWorker = require('./GenericWorker');

var utils = require('../utils');

/\*\*

\* A worker which convert chunks to a specified type.

\* @constructor

\* @param {String} destType the destination type.

\*/

function ConvertWorker(destType) {

GenericWorker.call(this, "ConvertWorker to " + destType);

this.destType = destType;

}

utils.inherits(ConvertWorker, GenericWorker);

/\*\*

\* @see GenericWorker.processChunk

\*/

ConvertWorker.prototype.processChunk = function (chunk) {

this.push({

data : utils.transformTo(this.destType, chunk.data),

meta : chunk.meta

});

};

module.exports = ConvertWorker;

},{"../utils":32,"./GenericWorker":28}],25:[function(require,module,exports){

'use strict';

var GenericWorker = require('./GenericWorker');

var crc32 = require('../crc32');

var utils = require('../utils');

/\*\*

\* A worker which calculate the crc32 of the data flowing through.

\* @constructor

\*/

function Crc32Probe() {

GenericWorker.call(this, "Crc32Probe");

this.withStreamInfo("crc32", 0);

}

utils.inherits(Crc32Probe, GenericWorker);

/\*\*

\* @see GenericWorker.processChunk

\*/

Crc32Probe.prototype.processChunk = function (chunk) {

this.streamInfo.crc32 = crc32(chunk.data, this.streamInfo.crc32 || 0);

this.push(chunk);

};

module.exports = Crc32Probe;

},{"../crc32":4,"../utils":32,"./GenericWorker":28}],26:[function(require,module,exports){

'use strict';

var utils = require('../utils');

var GenericWorker = require('./GenericWorker');

/\*\*

\* A worker which calculate the total length of the data flowing through.

\* @constructor

\* @param {String} propName the name used to expose the length

\*/

function DataLengthProbe(propName) {

GenericWorker.call(this, "DataLengthProbe for " + propName);

this.propName = propName;

this.withStreamInfo(propName, 0);

}

utils.inherits(DataLengthProbe, GenericWorker);

/\*\*

\* @see GenericWorker.processChunk

\*/

DataLengthProbe.prototype.processChunk = function (chunk) {

if(chunk) {

var length = this.streamInfo[this.propName] || 0;

this.streamInfo[this.propName] = length + chunk.data.length;

}

GenericWorker.prototype.processChunk.call(this, chunk);

};

module.exports = DataLengthProbe;

},{"../utils":32,"./GenericWorker":28}],27:[function(require,module,exports){

'use strict';

var utils = require('../utils');

var GenericWorker = require('./GenericWorker');

// the size of the generated chunks

// TODO expose this as a public variable

var DEFAULT\_BLOCK\_SIZE = 16 \* 1024;

/\*\*

\* A worker that reads a content and emits chunks.

\* @constructor

\* @param {Promise} dataP the promise of the data to split

\*/

function DataWorker(dataP) {

GenericWorker.call(this, "DataWorker");

var self = this;

this.dataIsReady = false;

this.index = 0;

this.max = 0;

this.data = null;

this.type = "";

this.\_tickScheduled = false;

dataP.then(function (data) {

self.dataIsReady = true;

self.data = data;

self.max = data && data.length || 0;

self.type = utils.getTypeOf(data);

if(!self.isPaused) {

self.\_tickAndRepeat();

}

}, function (e) {

self.error(e);

});

}

utils.inherits(DataWorker, GenericWorker);

/\*\*

\* @see GenericWorker.cleanUp

\*/

DataWorker.prototype.cleanUp = function () {

GenericWorker.prototype.cleanUp.call(this);

this.data = null;

};

/\*\*

\* @see GenericWorker.resume

\*/

DataWorker.prototype.resume = function () {

if(!GenericWorker.prototype.resume.call(this)) {

return false;

}

if (!this.\_tickScheduled && this.dataIsReady) {

this.\_tickScheduled = true;

utils.delay(this.\_tickAndRepeat, [], this);

}

return true;

};

/\*\*

\* Trigger a tick a schedule an other call to this function.

\*/

DataWorker.prototype.\_tickAndRepeat = function() {

this.\_tickScheduled = false;

if(this.isPaused || this.isFinished) {

return;

}

this.\_tick();

if(!this.isFinished) {

utils.delay(this.\_tickAndRepeat, [], this);

this.\_tickScheduled = true;

}

};

/\*\*

\* Read and push a chunk.

\*/

DataWorker.prototype.\_tick = function() {

if(this.isPaused || this.isFinished) {

return false;

}

var size = DEFAULT\_BLOCK\_SIZE;

var data = null, nextIndex = Math.min(this.max, this.index + size);

if (this.index >= this.max) {

// EOF

return this.end();

} else {

switch(this.type) {

case "string":

data = this.data.substring(this.index, nextIndex);

break;

case "uint8array":

data = this.data.subarray(this.index, nextIndex);

break;

case "array":

case "nodebuffer":

data = this.data.slice(this.index, nextIndex);

break;

}

this.index = nextIndex;

return this.push({

data : data,

meta : {

percent : this.max ? this.index / this.max \* 100 : 0

}

});

}

};

module.exports = DataWorker;

},{"../utils":32,"./GenericWorker":28}],28:[function(require,module,exports){

'use strict';

/\*\*

\* A worker that does nothing but passing chunks to the next one. This is like

\* a nodejs stream but with some differences. On the good side :

\* - it works on IE 6-9 without any issue / polyfill

\* - it weights less than the full dependencies bundled with browserify

\* - it forwards errors (no need to declare an error handler EVERYWHERE)

\*

\* A chunk is an object with 2 attributes : `meta` and `data`. The former is an

\* object containing anything (`percent` for example), see each worker for more

\* details. The latter is the real data (String, Uint8Array, etc).

\*

\* @constructor

\* @param {String} name the name of the stream (mainly used for debugging purposes)

\*/

function GenericWorker(name) {

// the name of the worker

this.name = name || "default";

// an object containing metadata about the workers chain

this.streamInfo = {};

// an error which happened when the worker was paused

this.generatedError = null;

// an object containing metadata to be merged by this worker into the general metadata

this.extraStreamInfo = {};

// true if the stream is paused (and should not do anything), false otherwise

this.isPaused = true;

// true if the stream is finished (and should not do anything), false otherwise

this.isFinished = false;

// true if the stream is locked to prevent further structure updates (pipe), false otherwise

this.isLocked = false;

// the event listeners

this.\_listeners = {

'data':[],

'end':[],

'error':[]

};

// the previous worker, if any

this.previous = null;

}

GenericWorker.prototype = {

/\*\*

\* Push a chunk to the next workers.

\* @param {Object} chunk the chunk to push

\*/

push : function (chunk) {

this.emit("data", chunk);

},

/\*\*

\* End the stream.

\* @return {Boolean} true if this call ended the worker, false otherwise.

\*/

end : function () {

if (this.isFinished) {

return false;

}

this.flush();

try {

this.emit("end");

this.cleanUp();

this.isFinished = true;

} catch (e) {

this.emit("error", e);

}

return true;

},

/\*\*

\* End the stream with an error.

\* @param {Error} e the error which caused the premature end.

\* @return {Boolean} true if this call ended the worker with an error, false otherwise.

\*/

error : function (e) {

if (this.isFinished) {

return false;

}

if(this.isPaused) {

this.generatedError = e;

} else {

this.isFinished = true;

this.emit("error", e);

// in the workers chain exploded in the middle of the chain,

// the error event will go downward but we also need to notify

// workers upward that there has been an error.

if(this.previous) {

this.previous.error(e);

}

this.cleanUp();

}

return true;

},

/\*\*

\* Add a callback on an event.

\* @param {String} name the name of the event (data, end, error)

\* @param {Function} listener the function to call when the event is triggered

\* @return {GenericWorker} the current object for chainability

\*/

on : function (name, listener) {

this.\_listeners[name].push(listener);

return this;

},

/\*\*

\* Clean any references when a worker is ending.

\*/

cleanUp : function () {

this.streamInfo = this.generatedError = this.extraStreamInfo = null;

this.\_listeners = [];

},

/\*\*

\* Trigger an event. This will call registered callback with the provided arg.

\* @param {String} name the name of the event (data, end, error)

\* @param {Object} arg the argument to call the callback with.

\*/

emit : function (name, arg) {

if (this.\_listeners[name]) {

for(var i = 0; i < this.\_listeners[name].length; i++) {

this.\_listeners[name][i].call(this, arg);

}

}

},

/\*\*

\* Chain a worker with an other.

\* @param {Worker} next the worker receiving events from the current one.

\* @return {worker} the next worker for chainability

\*/

pipe : function (next) {

return next.registerPrevious(this);

},

/\*\*

\* Same as `pipe` in the other direction.

\* Using an API with `pipe(next)` is very easy.

\* Implementing the API with the point of view of the next one registering

\* a source is easier, see the ZipFileWorker.

\* @param {Worker} previous the previous worker, sending events to this one

\* @return {Worker} the current worker for chainability

\*/

registerPrevious : function (previous) {

if (this.isLocked) {

throw new Error("The stream '" + this + "' has already been used.");

}

// sharing the streamInfo...

this.streamInfo = previous.streamInfo;

// ... and adding our own bits

this.mergeStreamInfo();

this.previous = previous;

var self = this;

previous.on('data', function (chunk) {

self.processChunk(chunk);

});

previous.on('end', function () {

self.end();

});

previous.on('error', function (e) {

self.error(e);

});

return this;

},

/\*\*

\* Pause the stream so it doesn't send events anymore.

\* @return {Boolean} true if this call paused the worker, false otherwise.

\*/

pause : function () {

if(this.isPaused || this.isFinished) {

return false;

}

this.isPaused = true;

if(this.previous) {

this.previous.pause();

}

return true;

},

/\*\*

\* Resume a paused stream.

\* @return {Boolean} true if this call resumed the worker, false otherwise.

\*/

resume : function () {

if(!this.isPaused || this.isFinished) {

return false;

}

this.isPaused = false;

// if true, the worker tried to resume but failed

var withError = false;

if(this.generatedError) {

this.error(this.generatedError);

withError = true;

}

if(this.previous) {

this.previous.resume();

}

return !withError;

},

/\*\*

\* Flush any remaining bytes as the stream is ending.

\*/

flush : function () {},

/\*\*

\* Process a chunk. This is usually the method overridden.

\* @param {Object} chunk the chunk to process.

\*/

processChunk : function(chunk) {

this.push(chunk);

},

/\*\*

\* Add a key/value to be added in the workers chain streamInfo once activated.

\* @param {String} key the key to use

\* @param {Object} value the associated value

\* @return {Worker} the current worker for chainability

\*/

withStreamInfo : function (key, value) {

this.extraStreamInfo[key] = value;

this.mergeStreamInfo();

return this;

},

/\*\*

\* Merge this worker's streamInfo into the chain's streamInfo.

\*/

mergeStreamInfo : function () {

for(var key in this.extraStreamInfo) {

if (!this.extraStreamInfo.hasOwnProperty(key)) {

continue;

}

this.streamInfo[key] = this.extraStreamInfo[key];

}

},

/\*\*

\* Lock the stream to prevent further updates on the workers chain.

\* After calling this method, all calls to pipe will fail.

\*/

lock: function () {

if (this.isLocked) {

throw new Error("The stream '" + this + "' has already been used.");

}

this.isLocked = true;

if (this.previous) {

this.previous.lock();

}

},

/\*\*

\*

\* Pretty print the workers chain.

\*/

toString : function () {

var me = "Worker " + this.name;

if (this.previous) {

return this.previous + " -> " + me;

} else {

return me;

}

}

};

module.exports = GenericWorker;

},{}],29:[function(require,module,exports){

'use strict';

var utils = require('../utils');

var ConvertWorker = require('./ConvertWorker');

var GenericWorker = require('./GenericWorker');

var base64 = require('../base64');

var support = require("../support");

var external = require("../external");

var NodejsStreamOutputAdapter = null;

if (support.nodestream) {

try {

NodejsStreamOutputAdapter = require('../nodejs/NodejsStreamOutputAdapter');

} catch(e) {}

}

/\*\*

\* Apply the final transformation of the data. If the user wants a Blob for

\* example, it's easier to work with an U8intArray and finally do the

\* ArrayBuffer/Blob conversion.

\* @param {String} type the name of the final type

\* @param {String|Uint8Array|Buffer} content the content to transform

\* @param {String} mimeType the mime type of the content, if applicable.

\* @return {String|Uint8Array|ArrayBuffer|Buffer|Blob} the content in the right format.

\*/

function transformZipOutput(type, content, mimeType) {

switch(type) {

case "blob" :

return utils.newBlob(utils.transformTo("arraybuffer", content), mimeType);

case "base64" :

return base64.encode(content);

default :

return utils.transformTo(type, content);

}

}

/\*\*

\* Concatenate an array of data of the given type.

\* @param {String} type the type of the data in the given array.

\* @param {Array} dataArray the array containing the data chunks to concatenate

\* @return {String|Uint8Array|Buffer} the concatenated data

\* @throws Error if the asked type is unsupported

\*/

function concat (type, dataArray) {

var i, index = 0, res = null, totalLength = 0;

for(i = 0; i < dataArray.length; i++) {

totalLength += dataArray[i].length;

}

switch(type) {

case "string":

return dataArray.join("");

case "array":

return Array.prototype.concat.apply([], dataArray);

case "uint8array":

res = new Uint8Array(totalLength);

for(i = 0; i < dataArray.length; i++) {

res.set(dataArray[i], index);

index += dataArray[i].length;

}

return res;

case "nodebuffer":

return Buffer.concat(dataArray);

default:

throw new Error("concat : unsupported type '" + type + "'");

}

}

/\*\*

\* Listen a StreamHelper, accumulate its content and concatenate it into a

\* complete block.

\* @param {StreamHelper} helper the helper to use.

\* @param {Function} updateCallback a callback called on each update. Called

\* with one arg :

\* - the metadata linked to the update received.

\* @return Promise the promise for the accumulation.

\*/

function accumulate(helper, updateCallback) {

return new external.Promise(function (resolve, reject){

var dataArray = [];

var chunkType = helper.\_internalType,

resultType = helper.\_outputType,

mimeType = helper.\_mimeType;

helper

.on('data', function (data, meta) {

dataArray.push(data);

if(updateCallback) {

updateCallback(meta);

}

})

.on('error', function(err) {

dataArray = [];

reject(err);

})

.on('end', function (){

try {

var result = transformZipOutput(resultType, concat(chunkType, dataArray), mimeType);

resolve(result);

} catch (e) {

reject(e);

}

dataArray = [];

})

.resume();

});

}

/\*\*

\* An helper to easily use workers outside of JSZip.

\* @constructor

\* @param {Worker} worker the worker to wrap

\* @param {String} outputType the type of data expected by the use

\* @param {String} mimeType the mime type of the content, if applicable.

\*/

function StreamHelper(worker, outputType, mimeType) {

var internalType = outputType;

switch(outputType) {

case "blob":

case "arraybuffer":

internalType = "uint8array";

break;

case "base64":

internalType = "string";

break;

}

try {

// the type used internally

this.\_internalType = internalType;

// the type used to output results

this.\_outputType = outputType;

// the mime type

this.\_mimeType = mimeType;

utils.checkSupport(internalType);

this.\_worker = worker.pipe(new ConvertWorker(internalType));

// the last workers can be rewired without issues but we need to

// prevent any updates on previous workers.

worker.lock();

} catch(e) {

this.\_worker = new GenericWorker("error");

this.\_worker.error(e);

}

}

StreamHelper.prototype = {

/\*\*

\* Listen a StreamHelper, accumulate its content and concatenate it into a

\* complete block.

\* @param {Function} updateCb the update callback.

\* @return Promise the promise for the accumulation.

\*/

accumulate : function (updateCb) {

return accumulate(this, updateCb);

},

/\*\*

\* Add a listener on an event triggered on a stream.

\* @param {String} evt the name of the event

\* @param {Function} fn the listener

\* @return {StreamHelper} the current helper.

\*/

on : function (evt, fn) {

var self = this;

if(evt === "data") {

this.\_worker.on(evt, function (chunk) {

fn.call(self, chunk.data, chunk.meta);

});

} else {

this.\_worker.on(evt, function () {

utils.delay(fn, arguments, self);

});

}

return this;

},

/\*\*

\* Resume the flow of chunks.

\* @return {StreamHelper} the current helper.

\*/

resume : function () {

utils.delay(this.\_worker.resume, [], this.\_worker);

return this;

},

/\*\*

\* Pause the flow of chunks.

\* @return {StreamHelper} the current helper.

\*/

pause : function () {

this.\_worker.pause();

return this;

},

/\*\*

\* Return a nodejs stream for this helper.

\* @param {Function} updateCb the update callback.

\* @return {NodejsStreamOutputAdapter} the nodejs stream.

\*/

toNodejsStream : function (updateCb) {

utils.checkSupport("nodestream");

if (this.\_outputType !== "nodebuffer") {

// an object stream containing blob/arraybuffer/uint8array/string

// is strange and I don't know if it would be useful.

// I you find this comment and have a good usecase, please open a

// bug report !

throw new Error(this.\_outputType + " is not supported by this method");

}

return new NodejsStreamOutputAdapter(this, {

objectMode : this.\_outputType !== "nodebuffer"

}, updateCb);

}

};

module.exports = StreamHelper;

},{"../base64":1,"../external":6,"../nodejs/NodejsStreamOutputAdapter":13,"../support":30,"../utils":32,"./ConvertWorker":24,"./GenericWorker":28}],30:[function(require,module,exports){

'use strict';

exports.base64 = true;

exports.array = true;

exports.string = true;

exports.arraybuffer = typeof ArrayBuffer !== "undefined" && typeof Uint8Array !== "undefined";

exports.nodebuffer = typeof Buffer !== "undefined";

// contains true if JSZip can read/generate Uint8Array, false otherwise.

exports.uint8array = typeof Uint8Array !== "undefined";

if (typeof ArrayBuffer === "undefined") {

exports.blob = false;

}

else {

var buffer = new ArrayBuffer(0);

try {

exports.blob = new Blob([buffer], {

type: "application/zip"

}).size === 0;

}

catch (e) {

try {

var Builder = self.BlobBuilder || self.WebKitBlobBuilder || self.MozBlobBuilder || self.MSBlobBuilder;

var builder = new Builder();

builder.append(buffer);

exports.blob = builder.getBlob('application/zip').size === 0;

}

catch (e) {

exports.blob = false;

}

}

}

try {

exports.nodestream = !!require('readable-stream').Readable;

} catch(e) {

exports.nodestream = false;

}

},{"readable-stream":16}],31:[function(require,module,exports){

'use strict';

var utils = require('./utils');

var support = require('./support');

var nodejsUtils = require('./nodejsUtils');

var GenericWorker = require('./stream/GenericWorker');

/\*\*

\* The following functions come from pako, from pako/lib/utils/strings

\* released under the MIT license, see pako https://github.com/nodeca/pako/

\*/

// Table with utf8 lengths (calculated by first byte of sequence)

// Note, that 5 & 6-byte values and some 4-byte values can not be represented in JS,

// because max possible codepoint is 0x10ffff

var \_utf8len = new Array(256);

for (var i=0; i<256; i++) {

\_utf8len[i] = (i >= 252 ? 6 : i >= 248 ? 5 : i >= 240 ? 4 : i >= 224 ? 3 : i >= 192 ? 2 : 1);

}

\_utf8len[254]=\_utf8len[254]=1; // Invalid sequence start

// convert string to array (typed, when possible)

var string2buf = function (str) {

var buf, c, c2, m\_pos, i, str\_len = str.length, buf\_len = 0;

// count binary size

for (m\_pos = 0; m\_pos < str\_len; m\_pos++) {

c = str.charCodeAt(m\_pos);

if ((c & 0xfc00) === 0xd800 && (m\_pos+1 < str\_len)) {

c2 = str.charCodeAt(m\_pos+1);

if ((c2 & 0xfc00) === 0xdc00) {

c = 0x10000 + ((c - 0xd800) << 10) + (c2 - 0xdc00);

m\_pos++;

}

}

buf\_len += c < 0x80 ? 1 : c < 0x800 ? 2 : c < 0x10000 ? 3 : 4;

}

// allocate buffer

if (support.uint8array) {

buf = new Uint8Array(buf\_len);

} else {

buf = new Array(buf\_len);

}

// convert

for (i=0, m\_pos = 0; i < buf\_len; m\_pos++) {

c = str.charCodeAt(m\_pos);

if ((c & 0xfc00) === 0xd800 && (m\_pos+1 < str\_len)) {

c2 = str.charCodeAt(m\_pos+1);

if ((c2 & 0xfc00) === 0xdc00) {

c = 0x10000 + ((c - 0xd800) << 10) + (c2 - 0xdc00);

m\_pos++;

}

}

if (c < 0x80) {

/\* one byte \*/

buf[i++] = c;

} else if (c < 0x800) {

/\* two bytes \*/

buf[i++] = 0xC0 | (c >>> 6);

buf[i++] = 0x80 | (c & 0x3f);

} else if (c < 0x10000) {

/\* three bytes \*/

buf[i++] = 0xE0 | (c >>> 12);

buf[i++] = 0x80 | (c >>> 6 & 0x3f);

buf[i++] = 0x80 | (c & 0x3f);

} else {

/\* four bytes \*/

buf[i++] = 0xf0 | (c >>> 18);

buf[i++] = 0x80 | (c >>> 12 & 0x3f);

buf[i++] = 0x80 | (c >>> 6 & 0x3f);

buf[i++] = 0x80 | (c & 0x3f);

}

}

return buf;

};

// Calculate max possible position in utf8 buffer,

// that will not break sequence. If that's not possible

// - (very small limits) return max size as is.

//

// buf[] - utf8 bytes array

// max - length limit (mandatory);

var utf8border = function(buf, max) {

var pos;

max = max || buf.length;

if (max > buf.length) { max = buf.length; }

// go back from last position, until start of sequence found

pos = max-1;

while (pos >= 0 && (buf[pos] & 0xC0) === 0x80) { pos--; }

// Fuckup - very small and broken sequence,

// return max, because we should return something anyway.

if (pos < 0) { return max; }

// If we came to start of buffer - that means vuffer is too small,

// return max too.

if (pos === 0) { return max; }

return (pos + \_utf8len[buf[pos]] > max) ? pos : max;

};

// convert array to string

var buf2string = function (buf) {

var str, i, out, c, c\_len;

var len = buf.length;

// Reserve max possible length (2 words per char)

// NB: by unknown reasons, Array is significantly faster for

// String.fromCharCode.apply than Uint16Array.

var utf16buf = new Array(len\*2);

for (out=0, i=0; i<len;) {

c = buf[i++];

// quick process ascii

if (c < 0x80) { utf16buf[out++] = c; continue; }

c\_len = \_utf8len[c];

// skip 5 & 6 byte codes

if (c\_len > 4) { utf16buf[out++] = 0xfffd; i += c\_len-1; continue; }

// apply mask on first byte

c &= c\_len === 2 ? 0x1f : c\_len === 3 ? 0x0f : 0x07;

// join the rest

while (c\_len > 1 && i < len) {

c = (c << 6) | (buf[i++] & 0x3f);

c\_len--;

}

// terminated by end of string?

if (c\_len > 1) { utf16buf[out++] = 0xfffd; continue; }

if (c < 0x10000) {

utf16buf[out++] = c;

} else {

c -= 0x10000;

utf16buf[out++] = 0xd800 | ((c >> 10) & 0x3ff);

utf16buf[out++] = 0xdc00 | (c & 0x3ff);

}

}

// shrinkBuf(utf16buf, out)

if (utf16buf.length !== out) {

if(utf16buf.subarray) {

utf16buf = utf16buf.subarray(0, out);

} else {

utf16buf.length = out;

}

}

// return String.fromCharCode.apply(null, utf16buf);

return utils.applyFromCharCode(utf16buf);

};

// That's all for the pako functions.

/\*\*

\* Transform a javascript string into an array (typed if possible) of bytes,

\* UTF-8 encoded.

\* @param {String} str the string to encode

\* @return {Array|Uint8Array|Buffer} the UTF-8 encoded string.

\*/

exports.utf8encode = function utf8encode(str) {

if (support.nodebuffer) {

return nodejsUtils.newBufferFrom(str, "utf-8");

}

return string2buf(str);

};

/\*\*

\* Transform a bytes array (or a representation) representing an UTF-8 encoded

\* string into a javascript string.

\* @param {Array|Uint8Array|Buffer} buf the data de decode

\* @return {String} the decoded string.

\*/

exports.utf8decode = function utf8decode(buf) {

if (support.nodebuffer) {

return utils.transformTo("nodebuffer", buf).toString("utf-8");

}

buf = utils.transformTo(support.uint8array ? "uint8array" : "array", buf);

return buf2string(buf);

};

/\*\*

\* A worker to decode utf8 encoded binary chunks into string chunks.

\* @constructor

\*/

function Utf8DecodeWorker() {

GenericWorker.call(this, "utf-8 decode");

// the last bytes if a chunk didn't end with a complete codepoint.

this.leftOver = null;

}

utils.inherits(Utf8DecodeWorker, GenericWorker);

/\*\*

\* @see GenericWorker.processChunk

\*/

Utf8DecodeWorker.prototype.processChunk = function (chunk) {

var data = utils.transformTo(support.uint8array ? "uint8array" : "array", chunk.data);

// 1st step, re-use what's left of the previous chunk

if (this.leftOver && this.leftOver.length) {

if(support.uint8array) {

var previousData = data;

data = new Uint8Array(previousData.length + this.leftOver.length);

data.set(this.leftOver, 0);

data.set(previousData, this.leftOver.length);

} else {

data = this.leftOver.concat(data);

}

this.leftOver = null;

}

var nextBoundary = utf8border(data);

var usableData = data;

if (nextBoundary !== data.length) {

if (support.uint8array) {

usableData = data.subarray(0, nextBoundary);

this.leftOver = data.subarray(nextBoundary, data.length);

} else {

usableData = data.slice(0, nextBoundary);

this.leftOver = data.slice(nextBoundary, data.length);

}

}

this.push({

data : exports.utf8decode(usableData),

meta : chunk.meta

});

};

/\*\*

\* @see GenericWorker.flush

\*/

Utf8DecodeWorker.prototype.flush = function () {

if(this.leftOver && this.leftOver.length) {

this.push({

data : exports.utf8decode(this.leftOver),

meta : {}

});

this.leftOver = null;

}

};

exports.Utf8DecodeWorker = Utf8DecodeWorker;

/\*\*

\* A worker to endcode string chunks into utf8 encoded binary chunks.

\* @constructor

\*/

function Utf8EncodeWorker() {

GenericWorker.call(this, "utf-8 encode");

}

utils.inherits(Utf8EncodeWorker, GenericWorker);

/\*\*

\* @see GenericWorker.processChunk

\*/

Utf8EncodeWorker.prototype.processChunk = function (chunk) {

this.push({

data : exports.utf8encode(chunk.data),

meta : chunk.meta

});

};

exports.Utf8EncodeWorker = Utf8EncodeWorker;

},{"./nodejsUtils":14,"./stream/GenericWorker":28,"./support":30,"./utils":32}],32:[function(require,module,exports){

'use strict';

var support = require('./support');

var base64 = require('./base64');

var nodejsUtils = require('./nodejsUtils');

var setImmediate = require('set-immediate-shim');

var external = require("./external");

/\*\*

\* Convert a string that pass as a "binary string": it should represent a byte

\* array but may have > 255 char codes. Be sure to take only the first byte

\* and returns the byte array.

\* @param {String} str the string to transform.

\* @return {Array|Uint8Array} the string in a binary format.

\*/

function string2binary(str) {

var result = null;

if (support.uint8array) {

result = new Uint8Array(str.length);

} else {

result = new Array(str.length);

}

return stringToArrayLike(str, result);

}

/\*\*

\* Create a new blob with the given content and the given type.

\* @param {String|ArrayBuffer} part the content to put in the blob. DO NOT use

\* an Uint8Array because the stock browser of android 4 won't accept it (it

\* will be silently converted to a string, "[object Uint8Array]").

\*

\* Use only ONE part to build the blob to avoid a memory leak in IE11 / Edge:

\* when a large amount of Array is used to create the Blob, the amount of

\* memory consumed is nearly 100 times the original data amount.

\*

\* @param {String} type the mime type of the blob.

\* @return {Blob} the created blob.

\*/

exports.newBlob = function(part, type) {

exports.checkSupport("blob");

try {

// Blob constructor

return new Blob([part], {

type: type

});

}

catch (e) {

try {

// deprecated, browser only, old way

var Builder = self.BlobBuilder || self.WebKitBlobBuilder || self.MozBlobBuilder || self.MSBlobBuilder;

var builder = new Builder();

builder.append(part);

return builder.getBlob(type);

}

catch (e) {

// well, fuck ?!

throw new Error("Bug : can't construct the Blob.");

}

}

};

/\*\*

\* The identity function.

\* @param {Object} input the input.

\* @return {Object} the same input.

\*/

function identity(input) {

return input;

}

/\*\*

\* Fill in an array with a string.

\* @param {String} str the string to use.

\* @param {Array|ArrayBuffer|Uint8Array|Buffer} array the array to fill in (will be mutated).

\* @return {Array|ArrayBuffer|Uint8Array|Buffer} the updated array.

\*/

function stringToArrayLike(str, array) {

for (var i = 0; i < str.length; ++i) {

array[i] = str.charCodeAt(i) & 0xFF;

}

return array;

}

/\*\*

\* An helper for the function arrayLikeToString.

\* This contains static informations and functions that

\* can be optimized by the browser JIT compiler.

\*/

var arrayToStringHelper = {

/\*\*

\* Transform an array of int into a string, chunk by chunk.

\* See the performances notes on arrayLikeToString.

\* @param {Array|ArrayBuffer|Uint8Array|Buffer} array the array to transform.

\* @param {String} type the type of the array.

\* @param {Integer} chunk the chunk size.

\* @return {String} the resulting string.

\* @throws Error if the chunk is too big for the stack.

\*/

stringifyByChunk: function(array, type, chunk) {

var result = [], k = 0, len = array.length;

// shortcut

if (len <= chunk) {

return String.fromCharCode.apply(null, array);

}

while (k < len) {

if (type === "array" || type === "nodebuffer") {

result.push(String.fromCharCode.apply(null, array.slice(k, Math.min(k + chunk, len))));

}

else {

result.push(String.fromCharCode.apply(null, array.subarray(k, Math.min(k + chunk, len))));

}

k += chunk;

}

return result.join("");

},

/\*\*

\* Call String.fromCharCode on every item in the array.

\* This is the naive implementation, which generate A LOT of intermediate string.

\* This should be used when everything else fail.

\* @param {Array|ArrayBuffer|Uint8Array|Buffer} array the array to transform.

\* @return {String} the result.

\*/

stringifyByChar: function(array){

var resultStr = "";

for(var i = 0; i < array.length; i++) {

resultStr += String.fromCharCode(array[i]);

}

return resultStr;

},

applyCanBeUsed : {

/\*\*

\* true if the browser accepts to use String.fromCharCode on Uint8Array

\*/

uint8array : (function () {

try {

return support.uint8array && String.fromCharCode.apply(null, new Uint8Array(1)).length === 1;

} catch (e) {

return false;

}

})(),

/\*\*

\* true if the browser accepts to use String.fromCharCode on nodejs Buffer.

\*/

nodebuffer : (function () {

try {

return support.nodebuffer && String.fromCharCode.apply(null, nodejsUtils.allocBuffer(1)).length === 1;

} catch (e) {

return false;

}

})()

}

};

/\*\*

\* Transform an array-like object to a string.

\* @param {Array|ArrayBuffer|Uint8Array|Buffer} array the array to transform.

\* @return {String} the result.

\*/

function arrayLikeToString(array) {

// Performances notes :

// --------------------

// String.fromCharCode.apply(null, array) is the fastest, see

// see http://jsperf.com/converting-a-uint8array-to-a-string/2

// but the stack is limited (and we can get huge arrays !).

//

// result += String.fromCharCode(array[i]); generate too many strings !

//

// This code is inspired by http://jsperf.com/arraybuffer-to-string-apply-performance/2

// TODO : we now have workers that split the work. Do we still need that ?

var chunk = 65536,

type = exports.getTypeOf(array),

canUseApply = true;

if (type === "uint8array") {

canUseApply = arrayToStringHelper.applyCanBeUsed.uint8array;

} else if (type === "nodebuffer") {

canUseApply = arrayToStringHelper.applyCanBeUsed.nodebuffer;

}

if (canUseApply) {

while (chunk > 1) {

try {

return arrayToStringHelper.stringifyByChunk(array, type, chunk);

} catch (e) {

chunk = Math.floor(chunk / 2);

}

}

}

// no apply or chunk error : slow and painful algorithm

// default browser on android 4.\*

return arrayToStringHelper.stringifyByChar(array);

}

exports.applyFromCharCode = arrayLikeToString;

/\*\*

\* Copy the data from an array-like to an other array-like.

\* @param {Array|ArrayBuffer|Uint8Array|Buffer} arrayFrom the origin array.

\* @param {Array|ArrayBuffer|Uint8Array|Buffer} arrayTo the destination array which will be mutated.

\* @return {Array|ArrayBuffer|Uint8Array|Buffer} the updated destination array.

\*/

function arrayLikeToArrayLike(arrayFrom, arrayTo) {

for (var i = 0; i < arrayFrom.length; i++) {

arrayTo[i] = arrayFrom[i];

}

return arrayTo;

}

// a matrix containing functions to transform everything into everything.

var transform = {};

// string to ?

transform["string"] = {

"string": identity,

"array": function(input) {

return stringToArrayLike(input, new Array(input.length));

},

"arraybuffer": function(input) {

return transform["string"]["uint8array"](input).buffer;

},

"uint8array": function(input) {

return stringToArrayLike(input, new Uint8Array(input.length));

},

"nodebuffer": function(input) {

return stringToArrayLike(input, nodejsUtils.allocBuffer(input.length));

}

};

// array to ?

transform["array"] = {

"string": arrayLikeToString,

"array": identity,

"arraybuffer": function(input) {

return (new Uint8Array(input)).buffer;

},

"uint8array": function(input) {

return new Uint8Array(input);

},

"nodebuffer": function(input) {

return nodejsUtils.newBufferFrom(input);

}

};

// arraybuffer to ?

transform["arraybuffer"] = {

"string": function(input) {

return arrayLikeToString(new Uint8Array(input));

},

"array": function(input) {

return arrayLikeToArrayLike(new Uint8Array(input), new Array(input.byteLength));

},

"arraybuffer": identity,

"uint8array": function(input) {

return new Uint8Array(input);

},

"nodebuffer": function(input) {

return nodejsUtils.newBufferFrom(new Uint8Array(input));

}

};

// uint8array to ?

transform["uint8array"] = {

"string": arrayLikeToString,

"array": function(input) {

return arrayLikeToArrayLike(input, new Array(input.length));

},

"arraybuffer": function(input) {

return input.buffer;

},

"uint8array": identity,

"nodebuffer": function(input) {

return nodejsUtils.newBufferFrom(input);

}

};

// nodebuffer to ?

transform["nodebuffer"] = {

"string": arrayLikeToString,

"array": function(input) {

return arrayLikeToArrayLike(input, new Array(input.length));

},

"arraybuffer": function(input) {

return transform["nodebuffer"]["uint8array"](input).buffer;

},

"uint8array": function(input) {

return arrayLikeToArrayLike(input, new Uint8Array(input.length));

},

"nodebuffer": identity

};

/\*\*

\* Transform an input into any type.

\* The supported output type are : string, array, uint8array, arraybuffer, nodebuffer.

\* If no output type is specified, the unmodified input will be returned.

\* @param {String} outputType the output type.

\* @param {String|Array|ArrayBuffer|Uint8Array|Buffer} input the input to convert.

\* @throws {Error} an Error if the browser doesn't support the requested output type.

\*/

exports.transformTo = function(outputType, input) {

if (!input) {

// undefined, null, etc

// an empty string won't harm.

input = "";

}

if (!outputType) {

return input;

}

exports.checkSupport(outputType);

var inputType = exports.getTypeOf(input);

var result = transform[inputType][outputType](input);

return result;

};

/\*\*

\* Return the type of the input.

\* The type will be in a format valid for JSZip.utils.transformTo : string, array, uint8array, arraybuffer.

\* @param {Object} input the input to identify.

\* @return {String} the (lowercase) type of the input.

\*/

exports.getTypeOf = function(input) {

if (typeof input === "string") {

return "string";

}

if (Object.prototype.toString.call(input) === "[object Array]") {

return "array";

}

if (support.nodebuffer && nodejsUtils.isBuffer(input)) {

return "nodebuffer";

}

if (support.uint8array && input instanceof Uint8Array) {

return "uint8array";

}

if (support.arraybuffer && input instanceof ArrayBuffer) {

return "arraybuffer";

}

};

/\*\*

\* Throw an exception if the type is not supported.

\* @param {String} type the type to check.

\* @throws {Error} an Error if the browser doesn't support the requested type.

\*/

exports.checkSupport = function(type) {

var supported = support[type.toLowerCase()];

if (!supported) {

throw new Error(type + " is not supported by this platform");

}

};

exports.MAX\_VALUE\_16BITS = 65535;

exports.MAX\_VALUE\_32BITS = -1; // well, "\xFF\xFF\xFF\xFF\xFF\xFF\xFF\xFF" is parsed as -1

/\*\*

\* Prettify a string read as binary.

\* @param {string} str the string to prettify.

\* @return {string} a pretty string.

\*/

exports.pretty = function(str) {

var res = '',

code, i;

for (i = 0; i < (str || "").length; i++) {

code = str.charCodeAt(i);

res += '\\x' + (code < 16 ? "0" : "") + code.toString(16).toUpperCase();

}

return res;

};

/\*\*

\* Defer the call of a function.

\* @param {Function} callback the function to call asynchronously.

\* @param {Array} args the arguments to give to the callback.

\*/

exports.delay = function(callback, args, self) {

setImmediate(function () {

callback.apply(self || null, args || []);

});

};

/\*\*

\* Extends a prototype with an other, without calling a constructor with

\* side effects. Inspired by nodejs' `utils.inherits`

\* @param {Function} ctor the constructor to augment

\* @param {Function} superCtor the parent constructor to use

\*/

exports.inherits = function (ctor, superCtor) {

var Obj = function() {};

Obj.prototype = superCtor.prototype;

ctor.prototype = new Obj();

};

/\*\*

\* Merge the objects passed as parameters into a new one.

\* @private

\* @param {...Object} var\_args All objects to merge.

\* @return {Object} a new object with the data of the others.

\*/

exports.extend = function() {

var result = {}, i, attr;

for (i = 0; i < arguments.length; i++) { // arguments is not enumerable in some browsers

for (attr in arguments[i]) {

if (arguments[i].hasOwnProperty(attr) && typeof result[attr] === "undefined") {

result[attr] = arguments[i][attr];

}

}

}

return result;

};

/\*\*

\* Transform arbitrary content into a Promise.

\* @param {String} name a name for the content being processed.

\* @param {Object} inputData the content to process.

\* @param {Boolean} isBinary true if the content is not an unicode string

\* @param {Boolean} isOptimizedBinaryString true if the string content only has one byte per character.

\* @param {Boolean} isBase64 true if the string content is encoded with base64.

\* @return {Promise} a promise in a format usable by JSZip.

\*/

exports.prepareContent = function(name, inputData, isBinary, isOptimizedBinaryString, isBase64) {

// if inputData is already a promise, this flatten it.

var promise = external.Promise.resolve(inputData).then(function(data) {

var isBlob = support.blob && (data instanceof Blob || ['[object File]', '[object Blob]'].indexOf(Object.prototype.toString.call(data)) !== -1);

if (isBlob && typeof FileReader !== "undefined") {

return new external.Promise(function (resolve, reject) {

var reader = new FileReader();

reader.onload = function(e) {

resolve(e.target.result);

};

reader.onerror = function(e) {

reject(e.target.error);

};

reader.readAsArrayBuffer(data);

});

} else {

return data;

}

});

return promise.then(function(data) {

var dataType = exports.getTypeOf(data);

if (!dataType) {

return external.Promise.reject(

new Error("Can't read the data of '" + name + "'. Is it " +

"in a supported JavaScript type (String, Blob, ArrayBuffer, etc) ?")

);

}

// special case : it's way easier to work with Uint8Array than with ArrayBuffer

if (dataType === "arraybuffer") {

data = exports.transformTo("uint8array", data);

} else if (dataType === "string") {

if (isBase64) {

data = base64.decode(data);

}

else if (isBinary) {

// optimizedBinaryString === true means that the file has already been filtered with a 0xFF mask

if (isOptimizedBinaryString !== true) {

// this is a string, not in a base64 format.

// Be sure that this is a correct "binary string"

data = string2binary(data);

}

}

}

return data;

});

};

},{"./base64":1,"./external":6,"./nodejsUtils":14,"./support":30,"set-immediate-shim":54}],33:[function(require,module,exports){

'use strict';

var readerFor = require('./reader/readerFor');

var utils = require('./utils');

var sig = require('./signature');

var ZipEntry = require('./zipEntry');

var utf8 = require('./utf8');

var support = require('./support');

// class ZipEntries {{{

/\*\*

\* All the entries in the zip file.

\* @constructor

\* @param {Object} loadOptions Options for loading the stream.

\*/

function ZipEntries(loadOptions) {

this.files = [];

this.loadOptions = loadOptions;

}

ZipEntries.prototype = {

/\*\*

\* Check that the reader is on the specified signature.

\* @param {string} expectedSignature the expected signature.

\* @throws {Error} if it is an other signature.

\*/

checkSignature: function(expectedSignature) {

if (!this.reader.readAndCheckSignature(expectedSignature)) {

this.reader.index -= 4;

var signature = this.reader.readString(4);

throw new Error("Corrupted zip or bug: unexpected signature " + "(" + utils.pretty(signature) + ", expected " + utils.pretty(expectedSignature) + ")");

}

},

/\*\*

\* Check if the given signature is at the given index.

\* @param {number} askedIndex the index to check.

\* @param {string} expectedSignature the signature to expect.

\* @return {boolean} true if the signature is here, false otherwise.

\*/

isSignature: function(askedIndex, expectedSignature) {

var currentIndex = this.reader.index;

this.reader.setIndex(askedIndex);

var signature = this.reader.readString(4);

var result = signature === expectedSignature;

this.reader.setIndex(currentIndex);

return result;

},

/\*\*

\* Read the end of the central directory.

\*/

readBlockEndOfCentral: function() {

this.diskNumber = this.reader.readInt(2);

this.diskWithCentralDirStart = this.reader.readInt(2);

this.centralDirRecordsOnThisDisk = this.reader.readInt(2);

this.centralDirRecords = this.reader.readInt(2);

this.centralDirSize = this.reader.readInt(4);

this.centralDirOffset = this.reader.readInt(4);

this.zipCommentLength = this.reader.readInt(2);

// warning : the encoding depends of the system locale

// On a linux machine with LANG=en\_US.utf8, this field is utf8 encoded.

// On a windows machine, this field is encoded with the localized windows code page.

var zipComment = this.reader.readData(this.zipCommentLength);

var decodeParamType = support.uint8array ? "uint8array" : "array";

// To get consistent behavior with the generation part, we will assume that

// this is utf8 encoded unless specified otherwise.

var decodeContent = utils.transformTo(decodeParamType, zipComment);

this.zipComment = this.loadOptions.decodeFileName(decodeContent);

},

/\*\*

\* Read the end of the Zip 64 central directory.

\* Not merged with the method readEndOfCentral :

\* The end of central can coexist with its Zip64 brother,

\* I don't want to read the wrong number of bytes !

\*/

readBlockZip64EndOfCentral: function() {

this.zip64EndOfCentralSize = this.reader.readInt(8);

this.reader.skip(4);

// this.versionMadeBy = this.reader.readString(2);

// this.versionNeeded = this.reader.readInt(2);

this.diskNumber = this.reader.readInt(4);

this.diskWithCentralDirStart = this.reader.readInt(4);

this.centralDirRecordsOnThisDisk = this.reader.readInt(8);

this.centralDirRecords = this.reader.readInt(8);

this.centralDirSize = this.reader.readInt(8);

this.centralDirOffset = this.reader.readInt(8);

this.zip64ExtensibleData = {};

var extraDataSize = this.zip64EndOfCentralSize - 44,

index = 0,

extraFieldId,

extraFieldLength,

extraFieldValue;

while (index < extraDataSize) {

extraFieldId = this.reader.readInt(2);

extraFieldLength = this.reader.readInt(4);

extraFieldValue = this.reader.readData(extraFieldLength);

this.zip64ExtensibleData[extraFieldId] = {

id: extraFieldId,

length: extraFieldLength,

value: extraFieldValue

};

}

},

/\*\*

\* Read the end of the Zip 64 central directory locator.

\*/

readBlockZip64EndOfCentralLocator: function() {

this.diskWithZip64CentralDirStart = this.reader.readInt(4);

this.relativeOffsetEndOfZip64CentralDir = this.reader.readInt(8);

this.disksCount = this.reader.readInt(4);

if (this.disksCount > 1) {

throw new Error("Multi-volumes zip are not supported");

}

},

/\*\*

\* Read the local files, based on the offset read in the central part.

\*/

readLocalFiles: function() {

var i, file;

for (i = 0; i < this.files.length; i++) {

file = this.files[i];

this.reader.setIndex(file.localHeaderOffset);

this.checkSignature(sig.LOCAL\_FILE\_HEADER);

file.readLocalPart(this.reader);

file.handleUTF8();

file.processAttributes();

}

},

/\*\*

\* Read the central directory.

\*/

readCentralDir: function() {

var file;

this.reader.setIndex(this.centralDirOffset);

while (this.reader.readAndCheckSignature(sig.CENTRAL\_FILE\_HEADER)) {

file = new ZipEntry({

zip64: this.zip64

}, this.loadOptions);

file.readCentralPart(this.reader);

this.files.push(file);

}

if (this.centralDirRecords !== this.files.length) {

if (this.centralDirRecords !== 0 && this.files.length === 0) {

// We expected some records but couldn't find ANY.

// This is really suspicious, as if something went wrong.

throw new Error("Corrupted zip or bug: expected " + this.centralDirRecords + " records in central dir, got " + this.files.length);

} else {

// We found some records but not all.

// Something is wrong but we got something for the user: no error here.

// console.warn("expected", this.centralDirRecords, "records in central dir, got", this.files.length);

}

}

},

/\*\*

\* Read the end of central directory.

\*/

readEndOfCentral: function() {

var offset = this.reader.lastIndexOfSignature(sig.CENTRAL\_DIRECTORY\_END);

if (offset < 0) {

// Check if the content is a truncated zip or complete garbage.

// A "LOCAL\_FILE\_HEADER" is not required at the beginning (auto

// extractible zip for example) but it can give a good hint.

// If an ajax request was used without responseType, we will also

// get unreadable data.

var isGarbage = !this.isSignature(0, sig.LOCAL\_FILE\_HEADER);

if (isGarbage) {

throw new Error("Can't find end of central directory : is this a zip file ? " +

"If it is, see https://stuk.github.io/jszip/documentation/howto/read\_zip.html");

} else {

throw new Error("Corrupted zip: can't find end of central directory");

}

}

this.reader.setIndex(offset);

var endOfCentralDirOffset = offset;

this.checkSignature(sig.CENTRAL\_DIRECTORY\_END);

this.readBlockEndOfCentral();

/\* extract from the zip spec :

4) If one of the fields in the end of central directory

record is too small to hold required data, the field

should be set to -1 (0xFFFF or 0xFFFFFFFF) and the

ZIP64 format record should be created.

5) The end of central directory record and the

Zip64 end of central directory locator record must

reside on the same disk when splitting or spanning

an archive.

\*/

if (this.diskNumber === utils.MAX\_VALUE\_16BITS || this.diskWithCentralDirStart === utils.MAX\_VALUE\_16BITS || this.centralDirRecordsOnThisDisk === utils.MAX\_VALUE\_16BITS || this.centralDirRecords === utils.MAX\_VALUE\_16BITS || this.centralDirSize === utils.MAX\_VALUE\_32BITS || this.centralDirOffset === utils.MAX\_VALUE\_32BITS) {

this.zip64 = true;

/\*

Warning : the zip64 extension is supported, but ONLY if the 64bits integer read from

the zip file can fit into a 32bits integer. This cannot be solved : JavaScript represents

all numbers as 64-bit double precision IEEE 754 floating point numbers.

So, we have 53bits for integers and bitwise operations treat everything as 32bits.

see https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/Bitwise\_Operators

and http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf section 8.5

\*/

// should look for a zip64 EOCD locator

offset = this.reader.lastIndexOfSignature(sig.ZIP64\_CENTRAL\_DIRECTORY\_LOCATOR);

if (offset < 0) {

throw new Error("Corrupted zip: can't find the ZIP64 end of central directory locator");

}

this.reader.setIndex(offset);

this.checkSignature(sig.ZIP64\_CENTRAL\_DIRECTORY\_LOCATOR);

this.readBlockZip64EndOfCentralLocator();

// now the zip64 EOCD record

if (!this.isSignature(this.relativeOffsetEndOfZip64CentralDir, sig.ZIP64\_CENTRAL\_DIRECTORY\_END)) {

// console.warn("ZIP64 end of central directory not where expected.");

this.relativeOffsetEndOfZip64CentralDir = this.reader.lastIndexOfSignature(sig.ZIP64\_CENTRAL\_DIRECTORY\_END);

if (this.relativeOffsetEndOfZip64CentralDir < 0) {

throw new Error("Corrupted zip: can't find the ZIP64 end of central directory");

}

}

this.reader.setIndex(this.relativeOffsetEndOfZip64CentralDir);

this.checkSignature(sig.ZIP64\_CENTRAL\_DIRECTORY\_END);

this.readBlockZip64EndOfCentral();

}

var expectedEndOfCentralDirOffset = this.centralDirOffset + this.centralDirSize;

if (this.zip64) {

expectedEndOfCentralDirOffset += 20; // end of central dir 64 locator

expectedEndOfCentralDirOffset += 12 /\* should not include the leading 12 bytes \*/ + this.zip64EndOfCentralSize;

}

var extraBytes = endOfCentralDirOffset - expectedEndOfCentralDirOffset;

if (extraBytes > 0) {

// console.warn(extraBytes, "extra bytes at beginning or within zipfile");

if (this.isSignature(endOfCentralDirOffset, sig.CENTRAL\_FILE\_HEADER)) {

// The offsets seem wrong, but we have something at the specified offset.

// So… we keep it.

} else {

// the offset is wrong, update the "zero" of the reader

// this happens if data has been prepended (crx files for example)

this.reader.zero = extraBytes;

}

} else if (extraBytes < 0) {

throw new Error("Corrupted zip: missing " + Math.abs(extraBytes) + " bytes.");

}

},

prepareReader: function(data) {

this.reader = readerFor(data);

},

/\*\*

\* Read a zip file and create ZipEntries.

\* @param {String|ArrayBuffer|Uint8Array|Buffer} data the binary string representing a zip file.

\*/

load: function(data) {

this.prepareReader(data);

this.readEndOfCentral();

this.readCentralDir();

this.readLocalFiles();

}

};

// }}} end of ZipEntries

module.exports = ZipEntries;

},{"./reader/readerFor":22,"./signature":23,"./support":30,"./utf8":31,"./utils":32,"./zipEntry":34}],34:[function(require,module,exports){

'use strict';

var readerFor = require('./reader/readerFor');

var utils = require('./utils');

var CompressedObject = require('./compressedObject');

var crc32fn = require('./crc32');

var utf8 = require('./utf8');

var compressions = require('./compressions');

var support = require('./support');

var MADE\_BY\_DOS = 0x00;

var MADE\_BY\_UNIX = 0x03;

/\*\*

\* Find a compression registered in JSZip.

\* @param {string} compressionMethod the method magic to find.

\* @return {Object|null} the JSZip compression object, null if none found.

\*/

var findCompression = function(compressionMethod) {

for (var method in compressions) {

if (!compressions.hasOwnProperty(method)) {

continue;

}

if (compressions[method].magic === compressionMethod) {

return compressions[method];

}

}

return null;

};

// class ZipEntry {{{

/\*\*

\* An entry in the zip file.

\* @constructor

\* @param {Object} options Options of the current file.

\* @param {Object} loadOptions Options for loading the stream.

\*/

function ZipEntry(options, loadOptions) {

this.options = options;

this.loadOptions = loadOptions;

}

ZipEntry.prototype = {

/\*\*

\* say if the file is encrypted.

\* @return {boolean} true if the file is encrypted, false otherwise.

\*/

isEncrypted: function() {

// bit 1 is set

return (this.bitFlag & 0x0001) === 0x0001;

},

/\*\*

\* say if the file has utf-8 filename/comment.

\* @return {boolean} true if the filename/comment is in utf-8, false otherwise.

\*/

useUTF8: function() {

// bit 11 is set

return (this.bitFlag & 0x0800) === 0x0800;

},

/\*\*

\* Read the local part of a zip file and add the info in this object.

\* @param {DataReader} reader the reader to use.

\*/

readLocalPart: function(reader) {

var compression, localExtraFieldsLength;

// we already know everything from the central dir !

// If the central dir data are false, we are doomed.

// On the bright side, the local part is scary : zip64, data descriptors, both, etc.

// The less data we get here, the more reliable this should be.

// Let's skip the whole header and dash to the data !

reader.skip(22);

// in some zip created on windows, the filename stored in the central dir contains \ instead of /.

// Strangely, the filename here is OK.

// I would love to treat these zip files as corrupted (see http://www.info-zip.org/FAQ.html#backslashes

// or APPNOTE#4.4.17.1, "All slashes MUST be forward slashes '/'") but there are a lot of bad zip generators...

// Search "unzip mismatching "local" filename continuing with "central" filename version" on

// the internet.

//

// I think I see the logic here : the central directory is used to display

// content and the local directory is used to extract the files. Mixing / and \

// may be used to display \ to windows users and use / when extracting the files.

// Unfortunately, this lead also to some issues : http://seclists.org/fulldisclosure/2009/Sep/394

this.fileNameLength = reader.readInt(2);

localExtraFieldsLength = reader.readInt(2); // can't be sure this will be the same as the central dir

// the fileName is stored as binary data, the handleUTF8 method will take care of the encoding.

this.fileName = reader.readData(this.fileNameLength);

reader.skip(localExtraFieldsLength);

if (this.compressedSize === -1 || this.uncompressedSize === -1) {

throw new Error("Bug or corrupted zip : didn't get enough informations from the central directory " + "(compressedSize === -1 || uncompressedSize === -1)");

}

compression = findCompression(this.compressionMethod);

if (compression === null) { // no compression found

throw new Error("Corrupted zip : compression " + utils.pretty(this.compressionMethod) + " unknown (inner file : " + utils.transformTo("string", this.fileName) + ")");

}

this.decompressed = new CompressedObject(this.compressedSize, this.uncompressedSize, this.crc32, compression, reader.readData(this.compressedSize));

},

/\*\*

\* Read the central part of a zip file and add the info in this object.

\* @param {DataReader} reader the reader to use.

\*/

readCentralPart: function(reader) {

this.versionMadeBy = reader.readInt(2);

reader.skip(2);

// this.versionNeeded = reader.readInt(2);

this.bitFlag = reader.readInt(2);

this.compressionMethod = reader.readString(2);

this.date = reader.readDate();

this.crc32 = reader.readInt(4);

this.compressedSize = reader.readInt(4);

this.uncompressedSize = reader.readInt(4);

var fileNameLength = reader.readInt(2);

this.extraFieldsLength = reader.readInt(2);

this.fileCommentLength = reader.readInt(2);

this.diskNumberStart = reader.readInt(2);

this.internalFileAttributes = reader.readInt(2);

this.externalFileAttributes = reader.readInt(4);

this.localHeaderOffset = reader.readInt(4);

if (this.isEncrypted()) {

throw new Error("Encrypted zip are not supported");

}

// will be read in the local part, see the comments there

reader.skip(fileNameLength);

this.readExtraFields(reader);

this.parseZIP64ExtraField(reader);

this.fileComment = reader.readData(this.fileCommentLength);

},

/\*\*

\* Parse the external file attributes and get the unix/dos permissions.

\*/

processAttributes: function () {

this.unixPermissions = null;

this.dosPermissions = null;

var madeBy = this.versionMadeBy >> 8;

// Check if we have the DOS directory flag set.

// We look for it in the DOS and UNIX permissions

// but some unknown platform could set it as a compatibility flag.

this.dir = this.externalFileAttributes & 0x0010 ? true : false;

if(madeBy === MADE\_BY\_DOS) {

// first 6 bits (0 to 5)

this.dosPermissions = this.externalFileAttributes & 0x3F;

}

if(madeBy === MADE\_BY\_UNIX) {

this.unixPermissions = (this.externalFileAttributes >> 16) & 0xFFFF;

// the octal permissions are in (this.unixPermissions & 0x01FF).toString(8);

}

// fail safe : if the name ends with a / it probably means a folder

if (!this.dir && this.fileNameStr.slice(-1) === '/') {

this.dir = true;

}

},

/\*\*

\* Parse the ZIP64 extra field and merge the info in the current ZipEntry.

\* @param {DataReader} reader the reader to use.

\*/

parseZIP64ExtraField: function(reader) {

if (!this.extraFields[0x0001]) {

return;

}

// should be something, preparing the extra reader

var extraReader = readerFor(this.extraFields[0x0001].value);

// I really hope that these 64bits integer can fit in 32 bits integer, because js

// won't let us have more.

if (this.uncompressedSize === utils.MAX\_VALUE\_32BITS) {

this.uncompressedSize = extraReader.readInt(8);

}

if (this.compressedSize === utils.MAX\_VALUE\_32BITS) {

this.compressedSize = extraReader.readInt(8);

}

if (this.localHeaderOffset === utils.MAX\_VALUE\_32BITS) {

this.localHeaderOffset = extraReader.readInt(8);

}

if (this.diskNumberStart === utils.MAX\_VALUE\_32BITS) {

this.diskNumberStart = extraReader.readInt(4);

}

},

/\*\*

\* Read the central part of a zip file and add the info in this object.

\* @param {DataReader} reader the reader to use.

\*/

readExtraFields: function(reader) {

var end = reader.index + this.extraFieldsLength,

extraFieldId,

extraFieldLength,

extraFieldValue;

if (!this.extraFields) {

this.extraFields = {};

}

while (reader.index < end) {

extraFieldId = reader.readInt(2);

extraFieldLength = reader.readInt(2);

extraFieldValue = reader.readData(extraFieldLength);

this.extraFields[extraFieldId] = {

id: extraFieldId,

length: extraFieldLength,

value: extraFieldValue

};

}

},

/\*\*

\* Apply an UTF8 transformation if needed.

\*/

handleUTF8: function() {

var decodeParamType = support.uint8array ? "uint8array" : "array";

if (this.useUTF8()) {

this.fileNameStr = utf8.utf8decode(this.fileName);

this.fileCommentStr = utf8.utf8decode(this.fileComment);

} else {

var upath = this.findExtraFieldUnicodePath();

if (upath !== null) {

this.fileNameStr = upath;

} else {

// ASCII text or unsupported code page

var fileNameByteArray = utils.transformTo(decodeParamType, this.fileName);

this.fileNameStr = this.loadOptions.decodeFileName(fileNameByteArray);

}

var ucomment = this.findExtraFieldUnicodeComment();

if (ucomment !== null) {

this.fileCommentStr = ucomment;

} else {

// ASCII text or unsupported code page

var commentByteArray = utils.transformTo(decodeParamType, this.fileComment);

this.fileCommentStr = this.loadOptions.decodeFileName(commentByteArray);

}

}

},

/\*\*

\* Find the unicode path declared in the extra field, if any.

\* @return {String} the unicode path, null otherwise.

\*/

findExtraFieldUnicodePath: function() {

var upathField = this.extraFields[0x7075];

if (upathField) {

var extraReader = readerFor(upathField.value);

// wrong version

if (extraReader.readInt(1) !== 1) {

return null;

}

// the crc of the filename changed, this field is out of date.

if (crc32fn(this.fileName) !== extraReader.readInt(4)) {

return null;

}

return utf8.utf8decode(extraReader.readData(upathField.length - 5));

}

return null;

},

/\*\*

\* Find the unicode comment declared in the extra field, if any.

\* @return {String} the unicode comment, null otherwise.

\*/

findExtraFieldUnicodeComment: function() {

var ucommentField = this.extraFields[0x6375];

if (ucommentField) {

var extraReader = readerFor(ucommentField.value);

// wrong version

if (extraReader.readInt(1) !== 1) {

return null;

}

// the crc of the comment changed, this field is out of date.

if (crc32fn(this.fileComment) !== extraReader.readInt(4)) {

return null;

}

return utf8.utf8decode(extraReader.readData(ucommentField.length - 5));

}

return null;

}

};

module.exports = ZipEntry;

},{"./compressedObject":2,"./compressions":3,"./crc32":4,"./reader/readerFor":22,"./support":30,"./utf8":31,"./utils":32}],35:[function(require,module,exports){

'use strict';

var StreamHelper = require('./stream/StreamHelper');

var DataWorker = require('./stream/DataWorker');

var utf8 = require('./utf8');

var CompressedObject = require('./compressedObject');

var GenericWorker = require('./stream/GenericWorker');

/\*\*

\* A simple object representing a file in the zip file.

\* @constructor

\* @param {string} name the name of the file

\* @param {String|ArrayBuffer|Uint8Array|Buffer} data the data

\* @param {Object} options the options of the file

\*/

var ZipObject = function(name, data, options) {

this.name = name;

this.dir = options.dir;

this.date = options.date;

this.comment = options.comment;

this.unixPermissions = options.unixPermissions;

this.dosPermissions = options.dosPermissions;

this.\_data = data;

this.\_dataBinary = options.binary;

// keep only the compression

this.options = {

compression : options.compression,

compressionOptions : options.compressionOptions

};

};

ZipObject.prototype = {

/\*\*

\* Create an internal stream for the content of this object.

\* @param {String} type the type of each chunk.

\* @return StreamHelper the stream.

\*/

internalStream: function (type) {

var result = null, outputType = "string";

try {

if (!type) {

throw new Error("No output type specified.");

}

outputType = type.toLowerCase();

var askUnicodeString = outputType === "string" || outputType === "text";

if (outputType === "binarystring" || outputType === "text") {

outputType = "string";

}

result = this.\_decompressWorker();

var isUnicodeString = !this.\_dataBinary;

if (isUnicodeString && !askUnicodeString) {

result = result.pipe(new utf8.Utf8EncodeWorker());

}

if (!isUnicodeString && askUnicodeString) {

result = result.pipe(new utf8.Utf8DecodeWorker());

}

} catch (e) {

result = new GenericWorker("error");

result.error(e);

}

return new StreamHelper(result, outputType, "");

},

/\*\*

\* Prepare the content in the asked type.

\* @param {String} type the type of the result.

\* @param {Function} onUpdate a function to call on each internal update.

\* @return Promise the promise of the result.

\*/

async: function (type, onUpdate) {

return this.internalStream(type).accumulate(onUpdate);

},

/\*\*

\* Prepare the content as a nodejs stream.

\* @param {String} type the type of each chunk.

\* @param {Function} onUpdate a function to call on each internal update.

\* @return Stream the stream.

\*/

nodeStream: function (type, onUpdate) {

return this.internalStream(type || "nodebuffer").toNodejsStream(onUpdate);

},

/\*\*

\* Return a worker for the compressed content.

\* @private

\* @param {Object} compression the compression object to use.

\* @param {Object} compressionOptions the options to use when compressing.

\* @return Worker the worker.

\*/

\_compressWorker: function (compression, compressionOptions) {

if (

this.\_data instanceof CompressedObject &&

this.\_data.compression.magic === compression.magic

) {

return this.\_data.getCompressedWorker();

} else {

var result = this.\_decompressWorker();

if(!this.\_dataBinary) {

result = result.pipe(new utf8.Utf8EncodeWorker());

}

return CompressedObject.createWorkerFrom(result, compression, compressionOptions);

}

},

/\*\*

\* Return a worker for the decompressed content.

\* @private

\* @return Worker the worker.

\*/

\_decompressWorker : function () {

if (this.\_data instanceof CompressedObject) {

return this.\_data.getContentWorker();

} else if (this.\_data instanceof GenericWorker) {

return this.\_data;

} else {

return new DataWorker(this.\_data);

}

}

};

var removedMethods = ["asText", "asBinary", "asNodeBuffer", "asUint8Array", "asArrayBuffer"];

var removedFn = function () {

throw new Error("This method has been removed in JSZip 3.0, please check the upgrade guide.");

};

for(var i = 0; i < removedMethods.length; i++) {

ZipObject.prototype[removedMethods[i]] = removedFn;

}

module.exports = ZipObject;

},{"./compressedObject":2,"./stream/DataWorker":27,"./stream/GenericWorker":28,"./stream/StreamHelper":29,"./utf8":31}],36:[function(require,module,exports){

(function (global){

'use strict';

var Mutation = global.MutationObserver || global.WebKitMutationObserver;

var scheduleDrain;

{

if (Mutation) {

var called = 0;

var observer = new Mutation(nextTick);

var element = global.document.createTextNode('');

observer.observe(element, {

characterData: true

});

scheduleDrain = function () {

element.data = (called = ++called % 2);

};

} else if (!global.setImmediate && typeof global.MessageChannel !== 'undefined') {

var channel = new global.MessageChannel();

channel.port1.onmessage = nextTick;

scheduleDrain = function () {

channel.port2.postMessage(0);

};

} else if ('document' in global && 'onreadystatechange' in global.document.createElement('script')) {

scheduleDrain = function () {

// Create a <script> element; its readystatechange event will be fired asynchronously once it is inserted

// into the document. Do so, thus queuing up the task. Remember to clean up once it's been called.

var scriptEl = global.document.createElement('script');

scriptEl.onreadystatechange = function () {

nextTick();

scriptEl.onreadystatechange = null;

scriptEl.parentNode.removeChild(scriptEl);

scriptEl = null;

};

global.document.documentElement.appendChild(scriptEl);

};

} else {

scheduleDrain = function () {

setTimeout(nextTick, 0);

};

}

}

var draining;

var queue = [];

//named nextTick for less confusing stack traces

function nextTick() {

draining = true;

var i, oldQueue;

var len = queue.length;

while (len) {

oldQueue = queue;

queue = [];

i = -1;

while (++i < len) {

oldQueue[i]();

}

len = queue.length;

}

draining = false;

}

module.exports = immediate;

function immediate(task) {

if (queue.push(task) === 1 && !draining) {

scheduleDrain();

}

}

}).call(this,typeof global !== "undefined" ? global : typeof self !== "undefined" ? self : typeof window !== "undefined" ? window : {})

},{}],37:[function(require,module,exports){

'use strict';

var immediate = require('immediate');

/\* istanbul ignore next \*/

function INTERNAL() {}

var handlers = {};

var REJECTED = ['REJECTED'];

var FULFILLED = ['FULFILLED'];

var PENDING = ['PENDING'];

module.exports = Promise;

function Promise(resolver) {

if (typeof resolver !== 'function') {

throw new TypeError('resolver must be a function');

}

this.state = PENDING;

this.queue = [];

this.outcome = void 0;

if (resolver !== INTERNAL) {

safelyResolveThenable(this, resolver);

}

}

Promise.prototype["finally"] = function (callback) {

if (typeof callback !== 'function') {

return this;

}

var p = this.constructor;

return this.then(resolve, reject);

function resolve(value) {

function yes () {

return value;

}

return p.resolve(callback()).then(yes);

}

function reject(reason) {

function no () {

throw reason;

}

return p.resolve(callback()).then(no);

}

};

Promise.prototype["catch"] = function (onRejected) {

return this.then(null, onRejected);

};

Promise.prototype.then = function (onFulfilled, onRejected) {

if (typeof onFulfilled !== 'function' && this.state === FULFILLED ||

typeof onRejected !== 'function' && this.state === REJECTED) {

return this;

}

var promise = new this.constructor(INTERNAL);

if (this.state !== PENDING) {

var resolver = this.state === FULFILLED ? onFulfilled : onRejected;

unwrap(promise, resolver, this.outcome);

} else {

this.queue.push(new QueueItem(promise, onFulfilled, onRejected));

}

return promise;

};

function QueueItem(promise, onFulfilled, onRejected) {

this.promise = promise;

if (typeof onFulfilled === 'function') {

this.onFulfilled = onFulfilled;

this.callFulfilled = this.otherCallFulfilled;

}

if (typeof onRejected === 'function') {

this.onRejected = onRejected;

this.callRejected = this.otherCallRejected;

}

}

QueueItem.prototype.callFulfilled = function (value) {

handlers.resolve(this.promise, value);

};

QueueItem.prototype.otherCallFulfilled = function (value) {

unwrap(this.promise, this.onFulfilled, value);

};

QueueItem.prototype.callRejected = function (value) {

handlers.reject(this.promise, value);

};

QueueItem.prototype.otherCallRejected = function (value) {

unwrap(this.promise, this.onRejected, value);

};

function unwrap(promise, func, value) {

immediate(function () {

var returnValue;

try {

returnValue = func(value);

} catch (e) {

return handlers.reject(promise, e);

}

if (returnValue === promise) {

handlers.reject(promise, new TypeError('Cannot resolve promise with itself'));

} else {

handlers.resolve(promise, returnValue);

}

});

}

handlers.resolve = function (self, value) {

var result = tryCatch(getThen, value);

if (result.status === 'error') {

return handlers.reject(self, result.value);

}

var thenable = result.value;

if (thenable) {

safelyResolveThenable(self, thenable);

} else {

self.state = FULFILLED;

self.outcome = value;

var i = -1;

var len = self.queue.length;

while (++i < len) {

self.queue[i].callFulfilled(value);

}

}

return self;

};

handlers.reject = function (self, error) {

self.state = REJECTED;

self.outcome = error;

var i = -1;

var len = self.queue.length;

while (++i < len) {

self.queue[i].callRejected(error);

}

return self;

};

function getThen(obj) {

// Make sure we only access the accessor once as required by the spec

var then = obj && obj.then;

if (obj && (typeof obj === 'object' || typeof obj === 'function') && typeof then === 'function') {

return function appyThen() {

then.apply(obj, arguments);

};

}

}

function safelyResolveThenable(self, thenable) {

// Either fulfill, reject or reject with error

var called = false;

function onError(value) {

if (called) {

return;

}

called = true;

handlers.reject(self, value);

}

function onSuccess(value) {

if (called) {

return;

}

called = true;

handlers.resolve(self, value);

}

function tryToUnwrap() {

thenable(onSuccess, onError);

}

var result = tryCatch(tryToUnwrap);

if (result.status === 'error') {

onError(result.value);

}

}

function tryCatch(func, value) {

var out = {};

try {

out.value = func(value);

out.status = 'success';

} catch (e) {

out.status = 'error';

out.value = e;

}

return out;

}

Promise.resolve = resolve;

function resolve(value) {

if (value instanceof this) {

return value;

}

return handlers.resolve(new this(INTERNAL), value);

}

Promise.reject = reject;

function reject(reason) {

var promise = new this(INTERNAL);

return handlers.reject(promise, reason);

}

Promise.all = all;

function all(iterable) {

var self = this;

if (Object.prototype.toString.call(iterable) !== '[object Array]') {

return this.reject(new TypeError('must be an array'));

}

var len = iterable.length;

var called = false;

if (!len) {

return this.resolve([]);

}

var values = new Array(len);

var resolved = 0;

var i = -1;

var promise = new this(INTERNAL);

while (++i < len) {

allResolver(iterable[i], i);

}

return promise;

function allResolver(value, i) {

self.resolve(value).then(resolveFromAll, function (error) {

if (!called) {

called = true;

handlers.reject(promise, error);

}

});

function resolveFromAll(outValue) {

values[i] = outValue;

if (++resolved === len && !called) {

called = true;

handlers.resolve(promise, values);

}

}

}

}

Promise.race = race;

function race(iterable) {

var self = this;

if (Object.prototype.toString.call(iterable) !== '[object Array]') {

return this.reject(new TypeError('must be an array'));

}

var len = iterable.length;

var called = false;

if (!len) {

return this.resolve([]);

}

var i = -1;

var promise = new this(INTERNAL);

while (++i < len) {

resolver(iterable[i]);

}

return promise;

function resolver(value) {

self.resolve(value).then(function (response) {

if (!called) {

called = true;

handlers.resolve(promise, response);

}

}, function (error) {

if (!called) {

called = true;

handlers.reject(promise, error);

}

});

}

}

},{"immediate":36}],38:[function(require,module,exports){

// Top level file is just a mixin of submodules & constants

'use strict';

var assign = require('./lib/utils/common').assign;

var deflate = require('./lib/deflate');

var inflate = require('./lib/inflate');

var constants = require('./lib/zlib/constants');

var pako = {};

assign(pako, deflate, inflate, constants);

module.exports = pako;

},{"./lib/deflate":39,"./lib/inflate":40,"./lib/utils/common":41,"./lib/zlib/constants":44}],39:[function(require,module,exports){

'use strict';

var zlib\_deflate = require('./zlib/deflate');

var utils = require('./utils/common');

var strings = require('./utils/strings');

var msg = require('./zlib/messages');

var ZStream = require('./zlib/zstream');

var toString = Object.prototype.toString;

/\* Public constants ==========================================================\*/

/\* ===========================================================================\*/

var Z\_NO\_FLUSH = 0;

var Z\_FINISH = 4;

var Z\_OK = 0;

var Z\_STREAM\_END = 1;

var Z\_SYNC\_FLUSH = 2;

var Z\_DEFAULT\_COMPRESSION = -1;

var Z\_DEFAULT\_STRATEGY = 0;

var Z\_DEFLATED = 8;

/\* ===========================================================================\*/

/\*\*

\* class Deflate

\*

\* Generic JS-style wrapper for zlib calls. If you don't need

\* streaming behaviour - use more simple functions: [[deflate]],

\* [[deflateRaw]] and [[gzip]].

\*\*/

/\* internal

\* Deflate.chunks -> Array

\*

\* Chunks of output data, if [[Deflate#onData]] not overriden.

\*\*/

/\*\*

\* Deflate.result -> Uint8Array|Array

\*

\* Compressed result, generated by default [[Deflate#onData]]

\* and [[Deflate#onEnd]] handlers. Filled after you push last chunk

\* (call [[Deflate#push]] with `Z\_FINISH` / `true` param) or if you

\* push a chunk with explicit flush (call [[Deflate#push]] with

\* `Z\_SYNC\_FLUSH` param).

\*\*/

/\*\*

\* Deflate.err -> Number

\*

\* Error code after deflate finished. 0 (Z\_OK) on success.

\* You will not need it in real life, because deflate errors

\* are possible only on wrong options or bad `onData` / `onEnd`

\* custom handlers.

\*\*/

/\*\*

\* Deflate.msg -> String

\*

\* Error message, if [[Deflate.err]] != 0

\*\*/

/\*\*

\* new Deflate(options)

\* - options (Object): zlib deflate options.

\*

\* Creates new deflator instance with specified params. Throws exception

\* on bad params. Supported options:

\*

\* - `level`

\* - `windowBits`

\* - `memLevel`

\* - `strategy`

\* - `dictionary`

\*

\* [http://zlib.net/manual.html#Advanced](http://zlib.net/manual.html#Advanced)

\* for more information on these.

\*

\* Additional options, for internal needs:

\*

\* - `chunkSize` - size of generated data chunks (16K by default)

\* - `raw` (Boolean) - do raw deflate

\* - `gzip` (Boolean) - create gzip wrapper

\* - `to` (String) - if equal to 'string', then result will be "binary string"

\* (each char code [0..255])

\* - `header` (Object) - custom header for gzip

\* - `text` (Boolean) - true if compressed data believed to be text

\* - `time` (Number) - modification time, unix timestamp

\* - `os` (Number) - operation system code

\* - `extra` (Array) - array of bytes with extra data (max 65536)

\* - `name` (String) - file name (binary string)

\* - `comment` (String) - comment (binary string)

\* - `hcrc` (Boolean) - true if header crc should be added

\*

\* ##### Example:

\*

\* ```javascript

\* var pako = require('pako')

\* , chunk1 = Uint8Array([1,2,3,4,5,6,7,8,9])

\* , chunk2 = Uint8Array([10,11,12,13,14,15,16,17,18,19]);

\*

\* var deflate = new pako.Deflate({ level: 3});

\*

\* deflate.push(chunk1, false);

\* deflate.push(chunk2, true); // true -> last chunk

\*

\* if (deflate.err) { throw new Error(deflate.err); }

\*

\* console.log(deflate.result);

\* ```

\*\*/

function Deflate(options) {

if (!(this instanceof Deflate)) return new Deflate(options);

this.options = utils.assign({

level: Z\_DEFAULT\_COMPRESSION,

method: Z\_DEFLATED,

chunkSize: 16384,

windowBits: 15,

memLevel: 8,

strategy: Z\_DEFAULT\_STRATEGY,

to: ''

}, options || {});

var opt = this.options;

if (opt.raw && (opt.windowBits > 0)) {

opt.windowBits = -opt.windowBits;

}

else if (opt.gzip && (opt.windowBits > 0) && (opt.windowBits < 16)) {

opt.windowBits += 16;

}

this.err = 0; // error code, if happens (0 = Z\_OK)

this.msg = ''; // error message

this.ended = false; // used to avoid multiple onEnd() calls

this.chunks = []; // chunks of compressed data

this.strm = new ZStream();

this.strm.avail\_out = 0;

var status = zlib\_deflate.deflateInit2(

this.strm,

opt.level,

opt.method,

opt.windowBits,

opt.memLevel,

opt.strategy

);

if (status !== Z\_OK) {

throw new Error(msg[status]);

}

if (opt.header) {

zlib\_deflate.deflateSetHeader(this.strm, opt.header);

}

if (opt.dictionary) {

var dict;

// Convert data if needed

if (typeof opt.dictionary === 'string') {

// If we need to compress text, change encoding to utf8.

dict = strings.string2buf(opt.dictionary);

} else if (toString.call(opt.dictionary) === '[object ArrayBuffer]') {

dict = new Uint8Array(opt.dictionary);

} else {

dict = opt.dictionary;

}

status = zlib\_deflate.deflateSetDictionary(this.strm, dict);

if (status !== Z\_OK) {

throw new Error(msg[status]);

}

this.\_dict\_set = true;

}

}

/\*\*

\* Deflate#push(data[, mode]) -> Boolean

\* - data (Uint8Array|Array|ArrayBuffer|String): input data. Strings will be

\* converted to utf8 byte sequence.

\* - mode (Number|Boolean): 0..6 for corresponding Z\_NO\_FLUSH..Z\_TREE modes.

\* See constants. Skipped or `false` means Z\_NO\_FLUSH, `true` meansh Z\_FINISH.

\*

\* Sends input data to deflate pipe, generating [[Deflate#onData]] calls with

\* new compressed chunks. Returns `true` on success. The last data block must have

\* mode Z\_FINISH (or `true`). That will flush internal pending buffers and call

\* [[Deflate#onEnd]]. For interim explicit flushes (without ending the stream) you

\* can use mode Z\_SYNC\_FLUSH, keeping the compression context.

\*

\* On fail call [[Deflate#onEnd]] with error code and return false.

\*

\* We strongly recommend to use `Uint8Array` on input for best speed (output

\* array format is detected automatically). Also, don't skip last param and always

\* use the same type in your code (boolean or number). That will improve JS speed.

\*

\* For regular `Array`-s make sure all elements are [0..255].

\*

\* ##### Example

\*

\* ```javascript

\* push(chunk, false); // push one of data chunks

\* ...

\* push(chunk, true); // push last chunk

\* ```

\*\*/

Deflate.prototype.push = function (data, mode) {

var strm = this.strm;

var chunkSize = this.options.chunkSize;

var status, \_mode;

if (this.ended) { return false; }

\_mode = (mode === ~~mode) ? mode : ((mode === true) ? Z\_FINISH : Z\_NO\_FLUSH);

// Convert data if needed

if (typeof data === 'string') {

// If we need to compress text, change encoding to utf8.

strm.input = strings.string2buf(data);

} else if (toString.call(data) === '[object ArrayBuffer]') {

strm.input = new Uint8Array(data);

} else {

strm.input = data;

}

strm.next\_in = 0;

strm.avail\_in = strm.input.length;

do {

if (strm.avail\_out === 0) {

strm.output = new utils.Buf8(chunkSize);

strm.next\_out = 0;

strm.avail\_out = chunkSize;

}

status = zlib\_deflate.deflate(strm, \_mode); /\* no bad return value \*/

if (status !== Z\_STREAM\_END && status !== Z\_OK) {

this.onEnd(status);

this.ended = true;

return false;

}

if (strm.avail\_out === 0 || (strm.avail\_in === 0 && (\_mode === Z\_FINISH || \_mode === Z\_SYNC\_FLUSH))) {

if (this.options.to === 'string') {

this.onData(strings.buf2binstring(utils.shrinkBuf(strm.output, strm.next\_out)));

} else {

this.onData(utils.shrinkBuf(strm.output, strm.next\_out));

}

}

} while ((strm.avail\_in > 0 || strm.avail\_out === 0) && status !== Z\_STREAM\_END);

// Finalize on the last chunk.

if (\_mode === Z\_FINISH) {

status = zlib\_deflate.deflateEnd(this.strm);

this.onEnd(status);

this.ended = true;

return status === Z\_OK;

}

// callback interim results if Z\_SYNC\_FLUSH.

if (\_mode === Z\_SYNC\_FLUSH) {

this.onEnd(Z\_OK);

strm.avail\_out = 0;

return true;

}

return true;

};

/\*\*

\* Deflate#onData(chunk) -> Void

\* - chunk (Uint8Array|Array|String): ouput data. Type of array depends

\* on js engine support. When string output requested, each chunk

\* will be string.

\*

\* By default, stores data blocks in `chunks[]` property and glue

\* those in `onEnd`. Override this handler, if you need another behaviour.

\*\*/

Deflate.prototype.onData = function (chunk) {

this.chunks.push(chunk);

};

/\*\*

\* Deflate#onEnd(status) -> Void

\* - status (Number): deflate status. 0 (Z\_OK) on success,

\* other if not.

\*

\* Called once after you tell deflate that the input stream is

\* complete (Z\_FINISH) or should be flushed (Z\_SYNC\_FLUSH)

\* or if an error happened. By default - join collected chunks,

\* free memory and fill `results` / `err` properties.

\*\*/

Deflate.prototype.onEnd = function (status) {

// On success - join

if (status === Z\_OK) {

if (this.options.to === 'string') {

this.result = this.chunks.join('');

} else {

this.result = utils.flattenChunks(this.chunks);

}

}

this.chunks = [];

this.err = status;

this.msg = this.strm.msg;

};

/\*\*

\* deflate(data[, options]) -> Uint8Array|Array|String

\* - data (Uint8Array|Array|String): input data to compress.

\* - options (Object): zlib deflate options.

\*

\* Compress `data` with deflate algorithm and `options`.

\*

\* Supported options are:

\*

\* - level

\* - windowBits

\* - memLevel

\* - strategy

\* - dictionary

\*

\* [http://zlib.net/manual.html#Advanced](http://zlib.net/manual.html#Advanced)

\* for more information on these.

\*

\* Sugar (options):

\*

\* - `raw` (Boolean) - say that we work with raw stream, if you don't wish to specify

\* negative windowBits implicitly.

\* - `to` (String) - if equal to 'string', then result will be "binary string"

\* (each char code [0..255])

\*

\* ##### Example:

\*

\* ```javascript

\* var pako = require('pako')

\* , data = Uint8Array([1,2,3,4,5,6,7,8,9]);

\*

\* console.log(pako.deflate(data));

\* ```

\*\*/

function deflate(input, options) {

var deflator = new Deflate(options);

deflator.push(input, true);

// That will never happens, if you don't cheat with options :)

if (deflator.err) { throw deflator.msg || msg[deflator.err]; }

return deflator.result;

}

/\*\*

\* deflateRaw(data[, options]) -> Uint8Array|Array|String

\* - data (Uint8Array|Array|String): input data to compress.

\* - options (Object): zlib deflate options.

\*

\* The same as [[deflate]], but creates raw data, without wrapper

\* (header and adler32 crc).

\*\*/

function deflateRaw(input, options) {

options = options || {};

options.raw = true;

return deflate(input, options);

}

/\*\*

\* gzip(data[, options]) -> Uint8Array|Array|String

\* - data (Uint8Array|Array|String): input data to compress.

\* - options (Object): zlib deflate options.

\*

\* The same as [[deflate]], but create gzip wrapper instead of

\* deflate one.

\*\*/

function gzip(input, options) {

options = options || {};

options.gzip = true;

return deflate(input, options);

}

exports.Deflate = Deflate;

exports.deflate = deflate;

exports.deflateRaw = deflateRaw;

exports.gzip = gzip;

},{"./utils/common":41,"./utils/strings":42,"./zlib/deflate":46,"./zlib/messages":51,"./zlib/zstream":53}],40:[function(require,module,exports){

'use strict';

var zlib\_inflate = require('./zlib/inflate');

var utils = require('./utils/common');

var strings = require('./utils/strings');

var c = require('./zlib/constants');

var msg = require('./zlib/messages');

var ZStream = require('./zlib/zstream');

var GZheader = require('./zlib/gzheader');

var toString = Object.prototype.toString;

/\*\*

\* class Inflate

\*

\* Generic JS-style wrapper for zlib calls. If you don't need

\* streaming behaviour - use more simple functions: [[inflate]]

\* and [[inflateRaw]].

\*\*/

/\* internal

\* inflate.chunks -> Array

\*

\* Chunks of output data, if [[Inflate#onData]] not overriden.

\*\*/

/\*\*

\* Inflate.result -> Uint8Array|Array|String

\*

\* Uncompressed result, generated by default [[Inflate#onData]]

\* and [[Inflate#onEnd]] handlers. Filled after you push last chunk

\* (call [[Inflate#push]] with `Z\_FINISH` / `true` param) or if you

\* push a chunk with explicit flush (call [[Inflate#push]] with

\* `Z\_SYNC\_FLUSH` param).

\*\*/

/\*\*

\* Inflate.err -> Number

\*

\* Error code after inflate finished. 0 (Z\_OK) on success.

\* Should be checked if broken data possible.

\*\*/

/\*\*

\* Inflate.msg -> String

\*

\* Error message, if [[Inflate.err]] != 0

\*\*/

/\*\*

\* new Inflate(options)

\* - options (Object): zlib inflate options.

\*

\* Creates new inflator instance with specified params. Throws exception

\* on bad params. Supported options:

\*

\* - `windowBits`

\* - `dictionary`

\*

\* [http://zlib.net/manual.html#Advanced](http://zlib.net/manual.html#Advanced)

\* for more information on these.

\*

\* Additional options, for internal needs:

\*

\* - `chunkSize` - size of generated data chunks (16K by default)

\* - `raw` (Boolean) - do raw inflate

\* - `to` (String) - if equal to 'string', then result will be converted

\* from utf8 to utf16 (javascript) string. When string output requested,

\* chunk length can differ from `chunkSize`, depending on content.

\*

\* By default, when no options set, autodetect deflate/gzip data format via

\* wrapper header.

\*

\* ##### Example:

\*

\* ```javascript

\* var pako = require('pako')

\* , chunk1 = Uint8Array([1,2,3,4,5,6,7,8,9])

\* , chunk2 = Uint8Array([10,11,12,13,14,15,16,17,18,19]);

\*

\* var inflate = new pako.Inflate({ level: 3});

\*

\* inflate.push(chunk1, false);

\* inflate.push(chunk2, true); // true -> last chunk

\*

\* if (inflate.err) { throw new Error(inflate.err); }

\*

\* console.log(inflate.result);

\* ```

\*\*/

function Inflate(options) {

if (!(this instanceof Inflate)) return new Inflate(options);

this.options = utils.assign({

chunkSize: 16384,

windowBits: 0,

to: ''

}, options || {});

var opt = this.options;

// Force window size for `raw` data, if not set directly,

// because we have no header for autodetect.

if (opt.raw && (opt.windowBits >= 0) && (opt.windowBits < 16)) {

opt.windowBits = -opt.windowBits;

if (opt.windowBits === 0) { opt.windowBits = -15; }

}

// If `windowBits` not defined (and mode not raw) - set autodetect flag for gzip/deflate

if ((opt.windowBits >= 0) && (opt.windowBits < 16) &&

!(options && options.windowBits)) {

opt.windowBits += 32;

}

// Gzip header has no info about windows size, we can do autodetect only

// for deflate. So, if window size not set, force it to max when gzip possible

if ((opt.windowBits > 15) && (opt.windowBits < 48)) {

// bit 3 (16) -> gzipped data

// bit 4 (32) -> autodetect gzip/deflate

if ((opt.windowBits & 15) === 0) {

opt.windowBits |= 15;

}

}

this.err = 0; // error code, if happens (0 = Z\_OK)

this.msg = ''; // error message

this.ended = false; // used to avoid multiple onEnd() calls

this.chunks = []; // chunks of compressed data

this.strm = new ZStream();

this.strm.avail\_out = 0;

var status = zlib\_inflate.inflateInit2(

this.strm,

opt.windowBits

);

if (status !== c.Z\_OK) {

throw new Error(msg[status]);

}

this.header = new GZheader();

zlib\_inflate.inflateGetHeader(this.strm, this.header);

}

/\*\*

\* Inflate#push(data[, mode]) -> Boolean

\* - data (Uint8Array|Array|ArrayBuffer|String): input data

\* - mode (Number|Boolean): 0..6 for corresponding Z\_NO\_FLUSH..Z\_TREE modes.

\* See constants. Skipped or `false` means Z\_NO\_FLUSH, `true` meansh Z\_FINISH.

\*

\* Sends input data to inflate pipe, generating [[Inflate#onData]] calls with

\* new output chunks. Returns `true` on success. The last data block must have

\* mode Z\_FINISH (or `true`). That will flush internal pending buffers and call

\* [[Inflate#onEnd]]. For interim explicit flushes (without ending the stream) you

\* can use mode Z\_SYNC\_FLUSH, keeping the decompression context.

\*

\* On fail call [[Inflate#onEnd]] with error code and return false.

\*

\* We strongly recommend to use `Uint8Array` on input for best speed (output

\* format is detected automatically). Also, don't skip last param and always

\* use the same type in your code (boolean or number). That will improve JS speed.

\*

\* For regular `Array`-s make sure all elements are [0..255].

\*

\* ##### Example

\*

\* ```javascript

\* push(chunk, false); // push one of data chunks

\* ...

\* push(chunk, true); // push last chunk

\* ```

\*\*/

Inflate.prototype.push = function (data, mode) {

var strm = this.strm;

var chunkSize = this.options.chunkSize;

var dictionary = this.options.dictionary;

var status, \_mode;

var next\_out\_utf8, tail, utf8str;

var dict;

// Flag to properly process Z\_BUF\_ERROR on testing inflate call

// when we check that all output data was flushed.

var allowBufError = false;

if (this.ended) { return false; }

\_mode = (mode === ~~mode) ? mode : ((mode === true) ? c.Z\_FINISH : c.Z\_NO\_FLUSH);

// Convert data if needed

if (typeof data === 'string') {

// Only binary strings can be decompressed on practice

strm.input = strings.binstring2buf(data);

} else if (toString.call(data) === '[object ArrayBuffer]') {

strm.input = new Uint8Array(data);

} else {

strm.input = data;

}

strm.next\_in = 0;

strm.avail\_in = strm.input.length;

do {

if (strm.avail\_out === 0) {

strm.output = new utils.Buf8(chunkSize);

strm.next\_out = 0;

strm.avail\_out = chunkSize;

}

status = zlib\_inflate.inflate(strm, c.Z\_NO\_FLUSH); /\* no bad return value \*/

if (status === c.Z\_NEED\_DICT && dictionary) {

// Convert data if needed

if (typeof dictionary === 'string') {

dict = strings.string2buf(dictionary);

} else if (toString.call(dictionary) === '[object ArrayBuffer]') {

dict = new Uint8Array(dictionary);

} else {

dict = dictionary;

}

status = zlib\_inflate.inflateSetDictionary(this.strm, dict);

}

if (status === c.Z\_BUF\_ERROR && allowBufError === true) {

status = c.Z\_OK;

allowBufError = false;

}

if (status !== c.Z\_STREAM\_END && status !== c.Z\_OK) {

this.onEnd(status);

this.ended = true;

return false;

}

if (strm.next\_out) {

if (strm.avail\_out === 0 || status === c.Z\_STREAM\_END || (strm.avail\_in === 0 && (\_mode === c.Z\_FINISH || \_mode === c.Z\_SYNC\_FLUSH))) {

if (this.options.to === 'string') {

next\_out\_utf8 = strings.utf8border(strm.output, strm.next\_out);

tail = strm.next\_out - next\_out\_utf8;

utf8str = strings.buf2string(strm.output, next\_out\_utf8);

// move tail

strm.next\_out = tail;

strm.avail\_out = chunkSize - tail;

if (tail) { utils.arraySet(strm.output, strm.output, next\_out\_utf8, tail, 0); }

this.onData(utf8str);

} else {

this.onData(utils.shrinkBuf(strm.output, strm.next\_out));

}

}

}

// When no more input data, we should check that internal inflate buffers

// are flushed. The only way to do it when avail\_out = 0 - run one more

// inflate pass. But if output data not exists, inflate return Z\_BUF\_ERROR.

// Here we set flag to process this error properly.

//

// NOTE. Deflate does not return error in this case and does not needs such

// logic.

if (strm.avail\_in === 0 && strm.avail\_out === 0) {

allowBufError = true;

}

} while ((strm.avail\_in > 0 || strm.avail\_out === 0) && status !== c.Z\_STREAM\_END);

if (status === c.Z\_STREAM\_END) {

\_mode = c.Z\_FINISH;

}

// Finalize on the last chunk.

if (\_mode === c.Z\_FINISH) {

status = zlib\_inflate.inflateEnd(this.strm);

this.onEnd(status);

this.ended = true;

return status === c.Z\_OK;

}

// callback interim results if Z\_SYNC\_FLUSH.

if (\_mode === c.Z\_SYNC\_FLUSH) {

this.onEnd(c.Z\_OK);

strm.avail\_out = 0;

return true;

}

return true;

};

/\*\*

\* Inflate#onData(chunk) -> Void

\* - chunk (Uint8Array|Array|String): ouput data. Type of array depends

\* on js engine support. When string output requested, each chunk

\* will be string.

\*

\* By default, stores data blocks in `chunks[]` property and glue

\* those in `onEnd`. Override this handler, if you need another behaviour.

\*\*/

Inflate.prototype.onData = function (chunk) {

this.chunks.push(chunk);

};

/\*\*

\* Inflate#onEnd(status) -> Void

\* - status (Number): inflate status. 0 (Z\_OK) on success,

\* other if not.

\*

\* Called either after you tell inflate that the input stream is

\* complete (Z\_FINISH) or should be flushed (Z\_SYNC\_FLUSH)

\* or if an error happened. By default - join collected chunks,

\* free memory and fill `results` / `err` properties.

\*\*/

Inflate.prototype.onEnd = function (status) {

// On success - join

if (status === c.Z\_OK) {

if (this.options.to === 'string') {

// Glue & convert here, until we teach pako to send

// utf8 alligned strings to onData

this.result = this.chunks.join('');

} else {

this.result = utils.flattenChunks(this.chunks);

}

}

this.chunks = [];

this.err = status;

this.msg = this.strm.msg;

};

/\*\*

\* inflate(data[, options]) -> Uint8Array|Array|String

\* - data (Uint8Array|Array|String): input data to decompress.

\* - options (Object): zlib inflate options.

\*

\* Decompress `data` with inflate/ungzip and `options`. Autodetect

\* format via wrapper header by default. That's why we don't provide

\* separate `ungzip` method.

\*

\* Supported options are:

\*

\* - windowBits

\*

\* [http://zlib.net/manual.html#Advanced](http://zlib.net/manual.html#Advanced)

\* for more information.

\*

\* Sugar (options):

\*

\* - `raw` (Boolean) - say that we work with raw stream, if you don't wish to specify

\* negative windowBits implicitly.

\* - `to` (String) - if equal to 'string', then result will be converted

\* from utf8 to utf16 (javascript) string. When string output requested,

\* chunk length can differ from `chunkSize`, depending on content.

\*

\*

\* ##### Example:

\*

\* ```javascript

\* var pako = require('pako')

\* , input = pako.deflate([1,2,3,4,5,6,7,8,9])

\* , output;

\*

\* try {

\* output = pako.inflate(input);

\* } catch (err)

\* console.log(err);

\* }

\* ```

\*\*/

function inflate(input, options) {

var inflator = new Inflate(options);

inflator.push(input, true);

// That will never happens, if you don't cheat with options :)

if (inflator.err) { throw inflator.msg || msg[inflator.err]; }

return inflator.result;

}

/\*\*

\* inflateRaw(data[, options]) -> Uint8Array|Array|String

\* - data (Uint8Array|Array|String): input data to decompress.

\* - options (Object): zlib inflate options.

\*

\* The same as [[inflate]], but creates raw data, without wrapper

\* (header and adler32 crc).

\*\*/

function inflateRaw(input, options) {

options = options || {};

options.raw = true;

return inflate(input, options);

}

/\*\*

\* ungzip(data[, options]) -> Uint8Array|Array|String

\* - data (Uint8Array|Array|String): input data to decompress.

\* - options (Object): zlib inflate options.

\*

\* Just shortcut to [[inflate]], because it autodetects format

\* by header.content. Done for convenience.

\*\*/

exports.Inflate = Inflate;

exports.inflate = inflate;

exports.inflateRaw = inflateRaw;

exports.ungzip = inflate;

},{"./utils/common":41,"./utils/strings":42,"./zlib/constants":44,"./zlib/gzheader":47,"./zlib/inflate":49,"./zlib/messages":51,"./zlib/zstream":53}],41:[function(require,module,exports){

'use strict';

var TYPED\_OK = (typeof Uint8Array !== 'undefined') &&

(typeof Uint16Array !== 'undefined') &&

(typeof Int32Array !== 'undefined');

exports.assign = function (obj /\*from1, from2, from3, ...\*/) {

var sources = Array.prototype.slice.call(arguments, 1);

while (sources.length) {

var source = sources.shift();

if (!source) { continue; }

if (typeof source !== 'object') {

throw new TypeError(source + 'must be non-object');

}

for (var p in source) {

if (source.hasOwnProperty(p)) {

obj[p] = source[p];

}

}

}

return obj;

};

// reduce buffer size, avoiding mem copy

exports.shrinkBuf = function (buf, size) {

if (buf.length === size) { return buf; }

if (buf.subarray) { return buf.subarray(0, size); }

buf.length = size;

return buf;

};

var fnTyped = {

arraySet: function (dest, src, src\_offs, len, dest\_offs) {

if (src.subarray && dest.subarray) {

dest.set(src.subarray(src\_offs, src\_offs + len), dest\_offs);

return;

}

// Fallback to ordinary array

for (var i = 0; i < len; i++) {

dest[dest\_offs + i] = src[src\_offs + i];

}

},

// Join array of chunks to single array.

flattenChunks: function (chunks) {

var i, l, len, pos, chunk, result;

// calculate data length

len = 0;

for (i = 0, l = chunks.length; i < l; i++) {

len += chunks[i].length;

}

// join chunks

result = new Uint8Array(len);

pos = 0;

for (i = 0, l = chunks.length; i < l; i++) {

chunk = chunks[i];

result.set(chunk, pos);

pos += chunk.length;

}

return result;

}

};

var fnUntyped = {

arraySet: function (dest, src, src\_offs, len, dest\_offs) {

for (var i = 0; i < len; i++) {

dest[dest\_offs + i] = src[src\_offs + i];

}

},

// Join array of chunks to single array.

flattenChunks: function (chunks) {

return [].concat.apply([], chunks);

}

};

// Enable/Disable typed arrays use, for testing

//

exports.setTyped = function (on) {

if (on) {

exports.Buf8 = Uint8Array;

exports.Buf16 = Uint16Array;

exports.Buf32 = Int32Array;

exports.assign(exports, fnTyped);

} else {

exports.Buf8 = Array;

exports.Buf16 = Array;

exports.Buf32 = Array;

exports.assign(exports, fnUntyped);

}

};

exports.setTyped(TYPED\_OK);

},{}],42:[function(require,module,exports){

// String encode/decode helpers

'use strict';

var utils = require('./common');

// Quick check if we can use fast array to bin string conversion

//

// - apply(Array) can fail on Android 2.2

// - apply(Uint8Array) can fail on iOS 5.1 Safary

//

var STR\_APPLY\_OK = true;

var STR\_APPLY\_UIA\_OK = true;

try { String.fromCharCode.apply(null, [ 0 ]); } catch (\_\_) { STR\_APPLY\_OK = false; }

try { String.fromCharCode.apply(null, new Uint8Array(1)); } catch (\_\_) { STR\_APPLY\_UIA\_OK = false; }

// Table with utf8 lengths (calculated by first byte of sequence)

// Note, that 5 & 6-byte values and some 4-byte values can not be represented in JS,

// because max possible codepoint is 0x10ffff

var \_utf8len = new utils.Buf8(256);

for (var q = 0; q < 256; q++) {

\_utf8len[q] = (q >= 252 ? 6 : q >= 248 ? 5 : q >= 240 ? 4 : q >= 224 ? 3 : q >= 192 ? 2 : 1);

}

\_utf8len[254] = \_utf8len[254] = 1; // Invalid sequence start

// convert string to array (typed, when possible)

exports.string2buf = function (str) {

var buf, c, c2, m\_pos, i, str\_len = str.length, buf\_len = 0;

// count binary size

for (m\_pos = 0; m\_pos < str\_len; m\_pos++) {

c = str.charCodeAt(m\_pos);

if ((c & 0xfc00) === 0xd800 && (m\_pos + 1 < str\_len)) {

c2 = str.charCodeAt(m\_pos + 1);

if ((c2 & 0xfc00) === 0xdc00) {

c = 0x10000 + ((c - 0xd800) << 10) + (c2 - 0xdc00);

m\_pos++;

}

}

buf\_len += c < 0x80 ? 1 : c < 0x800 ? 2 : c < 0x10000 ? 3 : 4;

}

// allocate buffer

buf = new utils.Buf8(buf\_len);

// convert

for (i = 0, m\_pos = 0; i < buf\_len; m\_pos++) {

c = str.charCodeAt(m\_pos);

if ((c & 0xfc00) === 0xd800 && (m\_pos + 1 < str\_len)) {

c2 = str.charCodeAt(m\_pos + 1);

if ((c2 & 0xfc00) === 0xdc00) {

c = 0x10000 + ((c - 0xd800) << 10) + (c2 - 0xdc00);

m\_pos++;

}

}

if (c < 0x80) {

/\* one byte \*/

buf[i++] = c;

} else if (c < 0x800) {

/\* two bytes \*/

buf[i++] = 0xC0 | (c >>> 6);

buf[i++] = 0x80 | (c & 0x3f);

} else if (c < 0x10000) {

/\* three bytes \*/

buf[i++] = 0xE0 | (c >>> 12);

buf[i++] = 0x80 | (c >>> 6 & 0x3f);

buf[i++] = 0x80 | (c & 0x3f);

} else {

/\* four bytes \*/

buf[i++] = 0xf0 | (c >>> 18);

buf[i++] = 0x80 | (c >>> 12 & 0x3f);

buf[i++] = 0x80 | (c >>> 6 & 0x3f);

buf[i++] = 0x80 | (c & 0x3f);

}

}

return buf;

};

// Helper (used in 2 places)

function buf2binstring(buf, len) {

// use fallback for big arrays to avoid stack overflow

if (len < 65537) {

if ((buf.subarray && STR\_APPLY\_UIA\_OK) || (!buf.subarray && STR\_APPLY\_OK)) {

return String.fromCharCode.apply(null, utils.shrinkBuf(buf, len));

}

}

var result = '';

for (var i = 0; i < len; i++) {

result += String.fromCharCode(buf[i]);

}

return result;

}

// Convert byte array to binary string

exports.buf2binstring = function (buf) {

return buf2binstring(buf, buf.length);

};

// Convert binary string (typed, when possible)

exports.binstring2buf = function (str) {

var buf = new utils.Buf8(str.length);

for (var i = 0, len = buf.length; i < len; i++) {

buf[i] = str.charCodeAt(i);

}

return buf;

};

// convert array to string

exports.buf2string = function (buf, max) {

var i, out, c, c\_len;

var len = max || buf.length;

// Reserve max possible length (2 words per char)

// NB: by unknown reasons, Array is significantly faster for

// String.fromCharCode.apply than Uint16Array.

var utf16buf = new Array(len \* 2);

for (out = 0, i = 0; i < len;) {

c = buf[i++];

// quick process ascii

if (c < 0x80) { utf16buf[out++] = c; continue; }

c\_len = \_utf8len[c];

// skip 5 & 6 byte codes

if (c\_len > 4) { utf16buf[out++] = 0xfffd; i += c\_len - 1; continue; }

// apply mask on first byte

c &= c\_len === 2 ? 0x1f : c\_len === 3 ? 0x0f : 0x07;

// join the rest

while (c\_len > 1 && i < len) {

c = (c << 6) | (buf[i++] & 0x3f);

c\_len--;

}

// terminated by end of string?

if (c\_len > 1) { utf16buf[out++] = 0xfffd; continue; }

if (c < 0x10000) {

utf16buf[out++] = c;

} else {

c -= 0x10000;

utf16buf[out++] = 0xd800 | ((c >> 10) & 0x3ff);

utf16buf[out++] = 0xdc00 | (c & 0x3ff);

}

}

return buf2binstring(utf16buf, out);

};

// Calculate max possible position in utf8 buffer,

// that will not break sequence. If that's not possible

// - (very small limits) return max size as is.

//

// buf[] - utf8 bytes array

// max - length limit (mandatory);

exports.utf8border = function (buf, max) {

var pos;

max = max || buf.length;

if (max > buf.length) { max = buf.length; }

// go back from last position, until start of sequence found

pos = max - 1;

while (pos >= 0 && (buf[pos] & 0xC0) === 0x80) { pos--; }

// Fuckup - very small and broken sequence,

// return max, because we should return something anyway.

if (pos < 0) { return max; }

// If we came to start of buffer - that means vuffer is too small,

// return max too.

if (pos === 0) { return max; }

return (pos + \_utf8len[buf[pos]] > max) ? pos : max;

};

},{"./common":41}],43:[function(require,module,exports){

'use strict';

// Note: adler32 takes 12% for level 0 and 2% for level 6.

// It doesn't worth to make additional optimizationa as in original.

// Small size is preferable.

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

function adler32(adler, buf, len, pos) {

var s1 = (adler & 0xffff) |0,

s2 = ((adler >>> 16) & 0xffff) |0,

n = 0;

while (len !== 0) {

// Set limit ~ twice less than 5552, to keep

// s2 in 31-bits, because we force signed ints.

// in other case %= will fail.

n = len > 2000 ? 2000 : len;

len -= n;

do {

s1 = (s1 + buf[pos++]) |0;

s2 = (s2 + s1) |0;

} while (--n);

s1 %= 65521;

s2 %= 65521;

}

return (s1 | (s2 << 16)) |0;

}

module.exports = adler32;

},{}],44:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

module.exports = {

/\* Allowed flush values; see deflate() and inflate() below for details \*/

Z\_NO\_FLUSH: 0,

Z\_PARTIAL\_FLUSH: 1,

Z\_SYNC\_FLUSH: 2,

Z\_FULL\_FLUSH: 3,

Z\_FINISH: 4,

Z\_BLOCK: 5,

Z\_TREES: 6,

/\* Return codes for the compression/decompression functions. Negative values

\* are errors, positive values are used for special but normal events.

\*/

Z\_OK: 0,

Z\_STREAM\_END: 1,

Z\_NEED\_DICT: 2,

Z\_ERRNO: -1,

Z\_STREAM\_ERROR: -2,

Z\_DATA\_ERROR: -3,

//Z\_MEM\_ERROR: -4,

Z\_BUF\_ERROR: -5,

//Z\_VERSION\_ERROR: -6,

/\* compression levels \*/

Z\_NO\_COMPRESSION: 0,

Z\_BEST\_SPEED: 1,

Z\_BEST\_COMPRESSION: 9,

Z\_DEFAULT\_COMPRESSION: -1,

Z\_FILTERED: 1,

Z\_HUFFMAN\_ONLY: 2,

Z\_RLE: 3,

Z\_FIXED: 4,

Z\_DEFAULT\_STRATEGY: 0,

/\* Possible values of the data\_type field (though see inflate()) \*/

Z\_BINARY: 0,

Z\_TEXT: 1,

//Z\_ASCII: 1, // = Z\_TEXT (deprecated)

Z\_UNKNOWN: 2,

/\* The deflate compression method \*/

Z\_DEFLATED: 8

//Z\_NULL: null // Use -1 or null inline, depending on var type

};

},{}],45:[function(require,module,exports){

'use strict';

// Note: we can't get significant speed boost here.

// So write code to minimize size - no pregenerated tables

// and array tools dependencies.

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

// Use ordinary array, since untyped makes no boost here

function makeTable() {

var c, table = [];

for (var n = 0; n < 256; n++) {

c = n;

for (var k = 0; k < 8; k++) {

c = ((c & 1) ? (0xEDB88320 ^ (c >>> 1)) : (c >>> 1));

}

table[n] = c;

}

return table;

}

// Create table on load. Just 255 signed longs. Not a problem.

var crcTable = makeTable();

function crc32(crc, buf, len, pos) {

var t = crcTable,

end = pos + len;

crc ^= -1;

for (var i = pos; i < end; i++) {

crc = (crc >>> 8) ^ t[(crc ^ buf[i]) & 0xFF];

}

return (crc ^ (-1)); // >>> 0;

}

module.exports = crc32;

},{}],46:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

var utils = require('../utils/common');

var trees = require('./trees');

var adler32 = require('./adler32');

var crc32 = require('./crc32');

var msg = require('./messages');

/\* Public constants ==========================================================\*/

/\* ===========================================================================\*/

/\* Allowed flush values; see deflate() and inflate() below for details \*/

var Z\_NO\_FLUSH = 0;

var Z\_PARTIAL\_FLUSH = 1;

//var Z\_SYNC\_FLUSH = 2;

var Z\_FULL\_FLUSH = 3;

var Z\_FINISH = 4;

var Z\_BLOCK = 5;

//var Z\_TREES = 6;

/\* Return codes for the compression/decompression functions. Negative values

\* are errors, positive values are used for special but normal events.

\*/

var Z\_OK = 0;

var Z\_STREAM\_END = 1;

//var Z\_NEED\_DICT = 2;

//var Z\_ERRNO = -1;

var Z\_STREAM\_ERROR = -2;

var Z\_DATA\_ERROR = -3;

//var Z\_MEM\_ERROR = -4;

var Z\_BUF\_ERROR = -5;

//var Z\_VERSION\_ERROR = -6;

/\* compression levels \*/

//var Z\_NO\_COMPRESSION = 0;

//var Z\_BEST\_SPEED = 1;

//var Z\_BEST\_COMPRESSION = 9;

var Z\_DEFAULT\_COMPRESSION = -1;

var Z\_FILTERED = 1;

var Z\_HUFFMAN\_ONLY = 2;

var Z\_RLE = 3;

var Z\_FIXED = 4;

var Z\_DEFAULT\_STRATEGY = 0;

/\* Possible values of the data\_type field (though see inflate()) \*/

//var Z\_BINARY = 0;

//var Z\_TEXT = 1;

//var Z\_ASCII = 1; // = Z\_TEXT

var Z\_UNKNOWN = 2;

/\* The deflate compression method \*/

var Z\_DEFLATED = 8;

/\*============================================================================\*/

var MAX\_MEM\_LEVEL = 9;

/\* Maximum value for memLevel in deflateInit2 \*/

var MAX\_WBITS = 15;

/\* 32K LZ77 window \*/

var DEF\_MEM\_LEVEL = 8;

var LENGTH\_CODES = 29;

/\* number of length codes, not counting the special END\_BLOCK code \*/

var LITERALS = 256;

/\* number of literal bytes 0..255 \*/

var L\_CODES = LITERALS + 1 + LENGTH\_CODES;

/\* number of Literal or Length codes, including the END\_BLOCK code \*/

var D\_CODES = 30;

/\* number of distance codes \*/

var BL\_CODES = 19;

/\* number of codes used to transfer the bit lengths \*/

var HEAP\_SIZE = 2 \* L\_CODES + 1;

/\* maximum heap size \*/

var MAX\_BITS = 15;

/\* All codes must not exceed MAX\_BITS bits \*/

var MIN\_MATCH = 3;

var MAX\_MATCH = 258;

var MIN\_LOOKAHEAD = (MAX\_MATCH + MIN\_MATCH + 1);

var PRESET\_DICT = 0x20;

var INIT\_STATE = 42;

var EXTRA\_STATE = 69;

var NAME\_STATE = 73;

var COMMENT\_STATE = 91;

var HCRC\_STATE = 103;

var BUSY\_STATE = 113;

var FINISH\_STATE = 666;

var BS\_NEED\_MORE = 1; /\* block not completed, need more input or more output \*/

var BS\_BLOCK\_DONE = 2; /\* block flush performed \*/

var BS\_FINISH\_STARTED = 3; /\* finish started, need only more output at next deflate \*/

var BS\_FINISH\_DONE = 4; /\* finish done, accept no more input or output \*/

var OS\_CODE = 0x03; // Unix :) . Don't detect, use this default.

function err(strm, errorCode) {

strm.msg = msg[errorCode];

return errorCode;

}

function rank(f) {

return ((f) << 1) - ((f) > 4 ? 9 : 0);

}

function zero(buf) { var len = buf.length; while (--len >= 0) { buf[len] = 0; } }

/\* =========================================================================

\* Flush as much pending output as possible. All deflate() output goes

\* through this function so some applications may wish to modify it

\* to avoid allocating a large strm->output buffer and copying into it.

\* (See also read\_buf()).

\*/

function flush\_pending(strm) {

var s = strm.state;

//\_tr\_flush\_bits(s);

var len = s.pending;

if (len > strm.avail\_out) {

len = strm.avail\_out;

}

if (len === 0) { return; }

utils.arraySet(strm.output, s.pending\_buf, s.pending\_out, len, strm.next\_out);

strm.next\_out += len;

s.pending\_out += len;

strm.total\_out += len;

strm.avail\_out -= len;

s.pending -= len;

if (s.pending === 0) {

s.pending\_out = 0;

}

}

function flush\_block\_only(s, last) {

trees.\_tr\_flush\_block(s, (s.block\_start >= 0 ? s.block\_start : -1), s.strstart - s.block\_start, last);

s.block\_start = s.strstart;

flush\_pending(s.strm);

}

function put\_byte(s, b) {

s.pending\_buf[s.pending++] = b;

}

/\* =========================================================================

\* Put a short in the pending buffer. The 16-bit value is put in MSB order.

\* IN assertion: the stream state is correct and there is enough room in

\* pending\_buf.

\*/

function putShortMSB(s, b) {

// put\_byte(s, (Byte)(b >> 8));

// put\_byte(s, (Byte)(b & 0xff));

s.pending\_buf[s.pending++] = (b >>> 8) & 0xff;

s.pending\_buf[s.pending++] = b & 0xff;

}

/\* ===========================================================================

\* Read a new buffer from the current input stream, update the adler32

\* and total number of bytes read. All deflate() input goes through

\* this function so some applications may wish to modify it to avoid

\* allocating a large strm->input buffer and copying from it.

\* (See also flush\_pending()).

\*/

function read\_buf(strm, buf, start, size) {

var len = strm.avail\_in;

if (len > size) { len = size; }

if (len === 0) { return 0; }

strm.avail\_in -= len;

// zmemcpy(buf, strm->next\_in, len);

utils.arraySet(buf, strm.input, strm.next\_in, len, start);

if (strm.state.wrap === 1) {

strm.adler = adler32(strm.adler, buf, len, start);

}

else if (strm.state.wrap === 2) {

strm.adler = crc32(strm.adler, buf, len, start);

}

strm.next\_in += len;

strm.total\_in += len;

return len;

}

/\* ===========================================================================

\* Set match\_start to the longest match starting at the given string and

\* return its length. Matches shorter or equal to prev\_length are discarded,

\* in which case the result is equal to prev\_length and match\_start is

\* garbage.

\* IN assertions: cur\_match is the head of the hash chain for the current

\* string (strstart) and its distance is <= MAX\_DIST, and prev\_length >= 1

\* OUT assertion: the match length is not greater than s->lookahead.

\*/

function longest\_match(s, cur\_match) {

var chain\_length = s.max\_chain\_length; /\* max hash chain length \*/

var scan = s.strstart; /\* current string \*/

var match; /\* matched string \*/

var len; /\* length of current match \*/

var best\_len = s.prev\_length; /\* best match length so far \*/

var nice\_match = s.nice\_match; /\* stop if match long enough \*/

var limit = (s.strstart > (s.w\_size - MIN\_LOOKAHEAD)) ?

s.strstart - (s.w\_size - MIN\_LOOKAHEAD) : 0/\*NIL\*/;

var \_win = s.window; // shortcut

var wmask = s.w\_mask;

var prev = s.prev;

/\* Stop when cur\_match becomes <= limit. To simplify the code,

\* we prevent matches with the string of window index 0.

\*/

var strend = s.strstart + MAX\_MATCH;

var scan\_end1 = \_win[scan + best\_len - 1];

var scan\_end = \_win[scan + best\_len];

/\* The code is optimized for HASH\_BITS >= 8 and MAX\_MATCH-2 multiple of 16.

\* It is easy to get rid of this optimization if necessary.

\*/

// Assert(s->hash\_bits >= 8 && MAX\_MATCH == 258, "Code too clever");

/\* Do not waste too much time if we already have a good match: \*/

if (s.prev\_length >= s.good\_match) {

chain\_length >>= 2;

}

/\* Do not look for matches beyond the end of the input. This is necessary

\* to make deflate deterministic.

\*/

if (nice\_match > s.lookahead) { nice\_match = s.lookahead; }

// Assert((ulg)s->strstart <= s->window\_size-MIN\_LOOKAHEAD, "need lookahead");

do {

// Assert(cur\_match < s->strstart, "no future");

match = cur\_match;

/\* Skip to next match if the match length cannot increase

\* or if the match length is less than 2. Note that the checks below

\* for insufficient lookahead only occur occasionally for performance

\* reasons. Therefore uninitialized memory will be accessed, and

\* conditional jumps will be made that depend on those values.

\* However the length of the match is limited to the lookahead, so

\* the output of deflate is not affected by the uninitialized values.

\*/

if (\_win[match + best\_len] !== scan\_end ||

\_win[match + best\_len - 1] !== scan\_end1 ||

\_win[match] !== \_win[scan] ||

\_win[++match] !== \_win[scan + 1]) {

continue;

}

/\* The check at best\_len-1 can be removed because it will be made

\* again later. (This heuristic is not always a win.)

\* It is not necessary to compare scan[2] and match[2] since they

\* are always equal when the other bytes match, given that

\* the hash keys are equal and that HASH\_BITS >= 8.

\*/

scan += 2;

match++;

// Assert(\*scan == \*match, "match[2]?");

/\* We check for insufficient lookahead only every 8th comparison;

\* the 256th check will be made at strstart+258.

\*/

do {

/\*jshint noempty:false\*/

} while (\_win[++scan] === \_win[++match] && \_win[++scan] === \_win[++match] &&

\_win[++scan] === \_win[++match] && \_win[++scan] === \_win[++match] &&

\_win[++scan] === \_win[++match] && \_win[++scan] === \_win[++match] &&

\_win[++scan] === \_win[++match] && \_win[++scan] === \_win[++match] &&

scan < strend);

// Assert(scan <= s->window+(unsigned)(s->window\_size-1), "wild scan");

len = MAX\_MATCH - (strend - scan);

scan = strend - MAX\_MATCH;

if (len > best\_len) {

s.match\_start = cur\_match;

best\_len = len;

if (len >= nice\_match) {

break;

}

scan\_end1 = \_win[scan + best\_len - 1];

scan\_end = \_win[scan + best\_len];

}

} while ((cur\_match = prev[cur\_match & wmask]) > limit && --chain\_length !== 0);

if (best\_len <= s.lookahead) {

return best\_len;

}

return s.lookahead;

}

/\* ===========================================================================

\* Fill the window when the lookahead becomes insufficient.

\* Updates strstart and lookahead.

\*

\* IN assertion: lookahead < MIN\_LOOKAHEAD

\* OUT assertions: strstart <= window\_size-MIN\_LOOKAHEAD

\* At least one byte has been read, or avail\_in == 0; reads are

\* performed for at least two bytes (required for the zip translate\_eol

\* option -- not supported here).

\*/

function fill\_window(s) {

var \_w\_size = s.w\_size;

var p, n, m, more, str;

//Assert(s->lookahead < MIN\_LOOKAHEAD, "already enough lookahead");

do {

more = s.window\_size - s.lookahead - s.strstart;

// JS ints have 32 bit, block below not needed

/\* Deal with !@#$% 64K limit: \*/

//if (sizeof(int) <= 2) {

// if (more == 0 && s->strstart == 0 && s->lookahead == 0) {

// more = wsize;

//

// } else if (more == (unsigned)(-1)) {

// /\* Very unlikely, but possible on 16 bit machine if

// \* strstart == 0 && lookahead == 1 (input done a byte at time)

// \*/

// more--;

// }

//}

/\* If the window is almost full and there is insufficient lookahead,

\* move the upper half to the lower one to make room in the upper half.

\*/

if (s.strstart >= \_w\_size + (\_w\_size - MIN\_LOOKAHEAD)) {

utils.arraySet(s.window, s.window, \_w\_size, \_w\_size, 0);

s.match\_start -= \_w\_size;

s.strstart -= \_w\_size;

/\* we now have strstart >= MAX\_DIST \*/

s.block\_start -= \_w\_size;

/\* Slide the hash table (could be avoided with 32 bit values

at the expense of memory usage). We slide even when level == 0

to keep the hash table consistent if we switch back to level > 0

later. (Using level 0 permanently is not an optimal usage of

zlib, so we don't care about this pathological case.)

\*/

n = s.hash\_size;

p = n;

do {

m = s.head[--p];

s.head[p] = (m >= \_w\_size ? m - \_w\_size : 0);

} while (--n);

n = \_w\_size;

p = n;

do {

m = s.prev[--p];

s.prev[p] = (m >= \_w\_size ? m - \_w\_size : 0);

/\* If n is not on any hash chain, prev[n] is garbage but

\* its value will never be used.

\*/

} while (--n);

more += \_w\_size;

}

if (s.strm.avail\_in === 0) {

break;

}

/\* If there was no sliding:

\* strstart <= WSIZE+MAX\_DIST-1 && lookahead <= MIN\_LOOKAHEAD - 1 &&

\* more == window\_size - lookahead - strstart

\* => more >= window\_size - (MIN\_LOOKAHEAD-1 + WSIZE + MAX\_DIST-1)

\* => more >= window\_size - 2\*WSIZE + 2

\* In the BIG\_MEM or MMAP case (not yet supported),

\* window\_size == input\_size + MIN\_LOOKAHEAD &&

\* strstart + s->lookahead <= input\_size => more >= MIN\_LOOKAHEAD.

\* Otherwise, window\_size == 2\*WSIZE so more >= 2.

\* If there was sliding, more >= WSIZE. So in all cases, more >= 2.

\*/

//Assert(more >= 2, "more < 2");

n = read\_buf(s.strm, s.window, s.strstart + s.lookahead, more);

s.lookahead += n;

/\* Initialize the hash value now that we have some input: \*/

if (s.lookahead + s.insert >= MIN\_MATCH) {

str = s.strstart - s.insert;

s.ins\_h = s.window[str];

/\* UPDATE\_HASH(s, s->ins\_h, s->window[str + 1]); \*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[str + 1]) & s.hash\_mask;

//#if MIN\_MATCH != 3

// Call update\_hash() MIN\_MATCH-3 more times

//#endif

while (s.insert) {

/\* UPDATE\_HASH(s, s->ins\_h, s->window[str + MIN\_MATCH-1]); \*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[str + MIN\_MATCH - 1]) & s.hash\_mask;

s.prev[str & s.w\_mask] = s.head[s.ins\_h];

s.head[s.ins\_h] = str;

str++;

s.insert--;

if (s.lookahead + s.insert < MIN\_MATCH) {

break;

}

}

}

/\* If the whole input has less than MIN\_MATCH bytes, ins\_h is garbage,

\* but this is not important since only literal bytes will be emitted.

\*/

} while (s.lookahead < MIN\_LOOKAHEAD && s.strm.avail\_in !== 0);

/\* If the WIN\_INIT bytes after the end of the current data have never been

\* written, then zero those bytes in order to avoid memory check reports of

\* the use of uninitialized (or uninitialised as Julian writes) bytes by

\* the longest match routines. Update the high water mark for the next

\* time through here. WIN\_INIT is set to MAX\_MATCH since the longest match

\* routines allow scanning to strstart + MAX\_MATCH, ignoring lookahead.

\*/

// if (s.high\_water < s.window\_size) {

// var curr = s.strstart + s.lookahead;

// var init = 0;

//

// if (s.high\_water < curr) {

// /\* Previous high water mark below current data -- zero WIN\_INIT

// \* bytes or up to end of window, whichever is less.

// \*/

// init = s.window\_size - curr;

// if (init > WIN\_INIT)

// init = WIN\_INIT;

// zmemzero(s->window + curr, (unsigned)init);

// s->high\_water = curr + init;

// }

// else if (s->high\_water < (ulg)curr + WIN\_INIT) {

// /\* High water mark at or above current data, but below current data

// \* plus WIN\_INIT -- zero out to current data plus WIN\_INIT, or up

// \* to end of window, whichever is less.

// \*/

// init = (ulg)curr + WIN\_INIT - s->high\_water;

// if (init > s->window\_size - s->high\_water)

// init = s->window\_size - s->high\_water;

// zmemzero(s->window + s->high\_water, (unsigned)init);

// s->high\_water += init;

// }

// }

//

// Assert((ulg)s->strstart <= s->window\_size - MIN\_LOOKAHEAD,

// "not enough room for search");

}

/\* ===========================================================================

\* Copy without compression as much as possible from the input stream, return

\* the current block state.

\* This function does not insert new strings in the dictionary since

\* uncompressible data is probably not useful. This function is used

\* only for the level=0 compression option.

\* NOTE: this function should be optimized to avoid extra copying from

\* window to pending\_buf.

\*/

function deflate\_stored(s, flush) {

/\* Stored blocks are limited to 0xffff bytes, pending\_buf is limited

\* to pending\_buf\_size, and each stored block has a 5 byte header:

\*/

var max\_block\_size = 0xffff;

if (max\_block\_size > s.pending\_buf\_size - 5) {

max\_block\_size = s.pending\_buf\_size - 5;

}

/\* Copy as much as possible from input to output: \*/

for (;;) {

/\* Fill the window as much as possible: \*/

if (s.lookahead <= 1) {

//Assert(s->strstart < s->w\_size+MAX\_DIST(s) ||

// s->block\_start >= (long)s->w\_size, "slide too late");

// if (!(s.strstart < s.w\_size + (s.w\_size - MIN\_LOOKAHEAD) ||

// s.block\_start >= s.w\_size)) {

// throw new Error("slide too late");

// }

fill\_window(s);

if (s.lookahead === 0 && flush === Z\_NO\_FLUSH) {

return BS\_NEED\_MORE;

}

if (s.lookahead === 0) {

break;

}

/\* flush the current block \*/

}

//Assert(s->block\_start >= 0L, "block gone");

// if (s.block\_start < 0) throw new Error("block gone");

s.strstart += s.lookahead;

s.lookahead = 0;

/\* Emit a stored block if pending\_buf will be full: \*/

var max\_start = s.block\_start + max\_block\_size;

if (s.strstart === 0 || s.strstart >= max\_start) {

/\* strstart == 0 is possible when wraparound on 16-bit machine \*/

s.lookahead = s.strstart - max\_start;

s.strstart = max\_start;

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

/\* Flush if we may have to slide, otherwise block\_start may become

\* negative and the data will be gone:

\*/

if (s.strstart - s.block\_start >= (s.w\_size - MIN\_LOOKAHEAD)) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

}

s.insert = 0;

if (flush === Z\_FINISH) {

/\*\*\* FLUSH\_BLOCK(s, 1); \*\*\*/

flush\_block\_only(s, true);

if (s.strm.avail\_out === 0) {

return BS\_FINISH\_STARTED;

}

/\*\*\*/

return BS\_FINISH\_DONE;

}

if (s.strstart > s.block\_start) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

return BS\_NEED\_MORE;

}

/\* ===========================================================================

\* Compress as much as possible from the input stream, return the current

\* block state.

\* This function does not perform lazy evaluation of matches and inserts

\* new strings in the dictionary only for unmatched strings or for short

\* matches. It is used only for the fast compression options.

\*/

function deflate\_fast(s, flush) {

var hash\_head; /\* head of the hash chain \*/

var bflush; /\* set if current block must be flushed \*/

for (;;) {

/\* Make sure that we always have enough lookahead, except

\* at the end of the input file. We need MAX\_MATCH bytes

\* for the next match, plus MIN\_MATCH bytes to insert the

\* string following the next match.

\*/

if (s.lookahead < MIN\_LOOKAHEAD) {

fill\_window(s);

if (s.lookahead < MIN\_LOOKAHEAD && flush === Z\_NO\_FLUSH) {

return BS\_NEED\_MORE;

}

if (s.lookahead === 0) {

break; /\* flush the current block \*/

}

}

/\* Insert the string window[strstart .. strstart+2] in the

\* dictionary, and set hash\_head to the head of the hash chain:

\*/

hash\_head = 0/\*NIL\*/;

if (s.lookahead >= MIN\_MATCH) {

/\*\*\* INSERT\_STRING(s, s.strstart, hash\_head); \*\*\*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[s.strstart + MIN\_MATCH - 1]) & s.hash\_mask;

hash\_head = s.prev[s.strstart & s.w\_mask] = s.head[s.ins\_h];

s.head[s.ins\_h] = s.strstart;

/\*\*\*/

}

/\* Find the longest match, discarding those <= prev\_length.

\* At this point we have always match\_length < MIN\_MATCH

\*/

if (hash\_head !== 0/\*NIL\*/ && ((s.strstart - hash\_head) <= (s.w\_size - MIN\_LOOKAHEAD))) {

/\* To simplify the code, we prevent matches with the string

\* of window index 0 (in particular we have to avoid a match

\* of the string with itself at the start of the input file).

\*/

s.match\_length = longest\_match(s, hash\_head);

/\* longest\_match() sets match\_start \*/

}

if (s.match\_length >= MIN\_MATCH) {

// check\_match(s, s.strstart, s.match\_start, s.match\_length); // for debug only

/\*\*\* \_tr\_tally\_dist(s, s.strstart - s.match\_start,

s.match\_length - MIN\_MATCH, bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, s.strstart - s.match\_start, s.match\_length - MIN\_MATCH);

s.lookahead -= s.match\_length;

/\* Insert new strings in the hash table only if the match length

\* is not too large. This saves time but degrades compression.

\*/

if (s.match\_length <= s.max\_lazy\_match/\*max\_insert\_length\*/ && s.lookahead >= MIN\_MATCH) {

s.match\_length--; /\* string at strstart already in table \*/

do {

s.strstart++;

/\*\*\* INSERT\_STRING(s, s.strstart, hash\_head); \*\*\*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[s.strstart + MIN\_MATCH - 1]) & s.hash\_mask;

hash\_head = s.prev[s.strstart & s.w\_mask] = s.head[s.ins\_h];

s.head[s.ins\_h] = s.strstart;

/\*\*\*/

/\* strstart never exceeds WSIZE-MAX\_MATCH, so there are

\* always MIN\_MATCH bytes ahead.

\*/

} while (--s.match\_length !== 0);

s.strstart++;

} else

{

s.strstart += s.match\_length;

s.match\_length = 0;

s.ins\_h = s.window[s.strstart];

/\* UPDATE\_HASH(s, s.ins\_h, s.window[s.strstart+1]); \*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[s.strstart + 1]) & s.hash\_mask;

//#if MIN\_MATCH != 3

// Call UPDATE\_HASH() MIN\_MATCH-3 more times

//#endif

/\* If lookahead < MIN\_MATCH, ins\_h is garbage, but it does not

\* matter since it will be recomputed at next deflate call.

\*/

}

} else {

/\* No match, output a literal byte \*/

//Tracevv((stderr,"%c", s.window[s.strstart]));

/\*\*\* \_tr\_tally\_lit(s, s.window[s.strstart], bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, 0, s.window[s.strstart]);

s.lookahead--;

s.strstart++;

}

if (bflush) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

}

s.insert = ((s.strstart < (MIN\_MATCH - 1)) ? s.strstart : MIN\_MATCH - 1);

if (flush === Z\_FINISH) {

/\*\*\* FLUSH\_BLOCK(s, 1); \*\*\*/

flush\_block\_only(s, true);

if (s.strm.avail\_out === 0) {

return BS\_FINISH\_STARTED;

}

/\*\*\*/

return BS\_FINISH\_DONE;

}

if (s.last\_lit) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

return BS\_BLOCK\_DONE;

}

/\* ===========================================================================

\* Same as above, but achieves better compression. We use a lazy

\* evaluation for matches: a match is finally adopted only if there is

\* no better match at the next window position.

\*/

function deflate\_slow(s, flush) {

var hash\_head; /\* head of hash chain \*/

var bflush; /\* set if current block must be flushed \*/

var max\_insert;

/\* Process the input block. \*/

for (;;) {

/\* Make sure that we always have enough lookahead, except

\* at the end of the input file. We need MAX\_MATCH bytes

\* for the next match, plus MIN\_MATCH bytes to insert the

\* string following the next match.

\*/

if (s.lookahead < MIN\_LOOKAHEAD) {

fill\_window(s);

if (s.lookahead < MIN\_LOOKAHEAD && flush === Z\_NO\_FLUSH) {

return BS\_NEED\_MORE;

}

if (s.lookahead === 0) { break; } /\* flush the current block \*/

}

/\* Insert the string window[strstart .. strstart+2] in the

\* dictionary, and set hash\_head to the head of the hash chain:

\*/

hash\_head = 0/\*NIL\*/;

if (s.lookahead >= MIN\_MATCH) {

/\*\*\* INSERT\_STRING(s, s.strstart, hash\_head); \*\*\*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[s.strstart + MIN\_MATCH - 1]) & s.hash\_mask;

hash\_head = s.prev[s.strstart & s.w\_mask] = s.head[s.ins\_h];

s.head[s.ins\_h] = s.strstart;

/\*\*\*/

}

/\* Find the longest match, discarding those <= prev\_length.

\*/

s.prev\_length = s.match\_length;

s.prev\_match = s.match\_start;

s.match\_length = MIN\_MATCH - 1;

if (hash\_head !== 0/\*NIL\*/ && s.prev\_length < s.max\_lazy\_match &&

s.strstart - hash\_head <= (s.w\_size - MIN\_LOOKAHEAD)/\*MAX\_DIST(s)\*/) {

/\* To simplify the code, we prevent matches with the string

\* of window index 0 (in particular we have to avoid a match

\* of the string with itself at the start of the input file).

\*/

s.match\_length = longest\_match(s, hash\_head);

/\* longest\_match() sets match\_start \*/

if (s.match\_length <= 5 &&

(s.strategy === Z\_FILTERED || (s.match\_length === MIN\_MATCH && s.strstart - s.match\_start > 4096/\*TOO\_FAR\*/))) {

/\* If prev\_match is also MIN\_MATCH, match\_start is garbage

\* but we will ignore the current match anyway.

\*/

s.match\_length = MIN\_MATCH - 1;

}

}

/\* If there was a match at the previous step and the current

\* match is not better, output the previous match:

\*/

if (s.prev\_length >= MIN\_MATCH && s.match\_length <= s.prev\_length) {

max\_insert = s.strstart + s.lookahead - MIN\_MATCH;

/\* Do not insert strings in hash table beyond this. \*/

//check\_match(s, s.strstart-1, s.prev\_match, s.prev\_length);

/\*\*\*\_tr\_tally\_dist(s, s.strstart - 1 - s.prev\_match,

s.prev\_length - MIN\_MATCH, bflush);\*\*\*/

bflush = trees.\_tr\_tally(s, s.strstart - 1 - s.prev\_match, s.prev\_length - MIN\_MATCH);

/\* Insert in hash table all strings up to the end of the match.

\* strstart-1 and strstart are already inserted. If there is not

\* enough lookahead, the last two strings are not inserted in

\* the hash table.

\*/

s.lookahead -= s.prev\_length - 1;

s.prev\_length -= 2;

do {

if (++s.strstart <= max\_insert) {

/\*\*\* INSERT\_STRING(s, s.strstart, hash\_head); \*\*\*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[s.strstart + MIN\_MATCH - 1]) & s.hash\_mask;

hash\_head = s.prev[s.strstart & s.w\_mask] = s.head[s.ins\_h];

s.head[s.ins\_h] = s.strstart;

/\*\*\*/

}

} while (--s.prev\_length !== 0);

s.match\_available = 0;

s.match\_length = MIN\_MATCH - 1;

s.strstart++;

if (bflush) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

} else if (s.match\_available) {

/\* If there was no match at the previous position, output a

\* single literal. If there was a match but the current match

\* is longer, truncate the previous match to a single literal.

\*/

//Tracevv((stderr,"%c", s->window[s->strstart-1]));

/\*\*\* \_tr\_tally\_lit(s, s.window[s.strstart-1], bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, 0, s.window[s.strstart - 1]);

if (bflush) {

/\*\*\* FLUSH\_BLOCK\_ONLY(s, 0) \*\*\*/

flush\_block\_only(s, false);

/\*\*\*/

}

s.strstart++;

s.lookahead--;

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

} else {

/\* There is no previous match to compare with, wait for

\* the next step to decide.

\*/

s.match\_available = 1;

s.strstart++;

s.lookahead--;

}

}

//Assert (flush != Z\_NO\_FLUSH, "no flush?");

if (s.match\_available) {

//Tracevv((stderr,"%c", s->window[s->strstart-1]));

/\*\*\* \_tr\_tally\_lit(s, s.window[s.strstart-1], bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, 0, s.window[s.strstart - 1]);

s.match\_available = 0;

}

s.insert = s.strstart < MIN\_MATCH - 1 ? s.strstart : MIN\_MATCH - 1;

if (flush === Z\_FINISH) {

/\*\*\* FLUSH\_BLOCK(s, 1); \*\*\*/

flush\_block\_only(s, true);

if (s.strm.avail\_out === 0) {

return BS\_FINISH\_STARTED;

}

/\*\*\*/

return BS\_FINISH\_DONE;

}

if (s.last\_lit) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

return BS\_BLOCK\_DONE;

}

/\* ===========================================================================

\* For Z\_RLE, simply look for runs of bytes, generate matches only of distance

\* one. Do not maintain a hash table. (It will be regenerated if this run of

\* deflate switches away from Z\_RLE.)

\*/

function deflate\_rle(s, flush) {

var bflush; /\* set if current block must be flushed \*/

var prev; /\* byte at distance one to match \*/

var scan, strend; /\* scan goes up to strend for length of run \*/

var \_win = s.window;

for (;;) {

/\* Make sure that we always have enough lookahead, except

\* at the end of the input file. We need MAX\_MATCH bytes

\* for the longest run, plus one for the unrolled loop.

\*/

if (s.lookahead <= MAX\_MATCH) {

fill\_window(s);

if (s.lookahead <= MAX\_MATCH && flush === Z\_NO\_FLUSH) {

return BS\_NEED\_MORE;

}

if (s.lookahead === 0) { break; } /\* flush the current block \*/

}

/\* See how many times the previous byte repeats \*/

s.match\_length = 0;

if (s.lookahead >= MIN\_MATCH && s.strstart > 0) {

scan = s.strstart - 1;

prev = \_win[scan];

if (prev === \_win[++scan] && prev === \_win[++scan] && prev === \_win[++scan]) {

strend = s.strstart + MAX\_MATCH;

do {

/\*jshint noempty:false\*/

} while (prev === \_win[++scan] && prev === \_win[++scan] &&

prev === \_win[++scan] && prev === \_win[++scan] &&

prev === \_win[++scan] && prev === \_win[++scan] &&

prev === \_win[++scan] && prev === \_win[++scan] &&

scan < strend);

s.match\_length = MAX\_MATCH - (strend - scan);

if (s.match\_length > s.lookahead) {

s.match\_length = s.lookahead;

}

}

//Assert(scan <= s->window+(uInt)(s->window\_size-1), "wild scan");

}

/\* Emit match if have run of MIN\_MATCH or longer, else emit literal \*/

if (s.match\_length >= MIN\_MATCH) {

//check\_match(s, s.strstart, s.strstart - 1, s.match\_length);

/\*\*\* \_tr\_tally\_dist(s, 1, s.match\_length - MIN\_MATCH, bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, 1, s.match\_length - MIN\_MATCH);

s.lookahead -= s.match\_length;

s.strstart += s.match\_length;

s.match\_length = 0;

} else {

/\* No match, output a literal byte \*/

//Tracevv((stderr,"%c", s->window[s->strstart]));

/\*\*\* \_tr\_tally\_lit(s, s.window[s.strstart], bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, 0, s.window[s.strstart]);

s.lookahead--;

s.strstart++;

}

if (bflush) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

}

s.insert = 0;

if (flush === Z\_FINISH) {

/\*\*\* FLUSH\_BLOCK(s, 1); \*\*\*/

flush\_block\_only(s, true);

if (s.strm.avail\_out === 0) {

return BS\_FINISH\_STARTED;

}

/\*\*\*/

return BS\_FINISH\_DONE;

}

if (s.last\_lit) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

return BS\_BLOCK\_DONE;

}

/\* ===========================================================================

\* For Z\_HUFFMAN\_ONLY, do not look for matches. Do not maintain a hash table.

\* (It will be regenerated if this run of deflate switches away from Huffman.)

\*/

function deflate\_huff(s, flush) {

var bflush; /\* set if current block must be flushed \*/

for (;;) {

/\* Make sure that we have a literal to write. \*/

if (s.lookahead === 0) {

fill\_window(s);

if (s.lookahead === 0) {

if (flush === Z\_NO\_FLUSH) {

return BS\_NEED\_MORE;

}

break; /\* flush the current block \*/

}

}

/\* Output a literal byte \*/

s.match\_length = 0;

//Tracevv((stderr,"%c", s->window[s->strstart]));

/\*\*\* \_tr\_tally\_lit(s, s.window[s.strstart], bflush); \*\*\*/

bflush = trees.\_tr\_tally(s, 0, s.window[s.strstart]);

s.lookahead--;

s.strstart++;

if (bflush) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

}

s.insert = 0;

if (flush === Z\_FINISH) {

/\*\*\* FLUSH\_BLOCK(s, 1); \*\*\*/

flush\_block\_only(s, true);

if (s.strm.avail\_out === 0) {

return BS\_FINISH\_STARTED;

}

/\*\*\*/

return BS\_FINISH\_DONE;

}

if (s.last\_lit) {

/\*\*\* FLUSH\_BLOCK(s, 0); \*\*\*/

flush\_block\_only(s, false);

if (s.strm.avail\_out === 0) {

return BS\_NEED\_MORE;

}

/\*\*\*/

}

return BS\_BLOCK\_DONE;

}

/\* Values for max\_lazy\_match, good\_match and max\_chain\_length, depending on

\* the desired pack level (0..9). The values given below have been tuned to

\* exclude worst case performance for pathological files. Better values may be

\* found for specific files.

\*/

function Config(good\_length, max\_lazy, nice\_length, max\_chain, func) {

this.good\_length = good\_length;

this.max\_lazy = max\_lazy;

this.nice\_length = nice\_length;

this.max\_chain = max\_chain;

this.func = func;

}

var configuration\_table;

configuration\_table = [

/\* good lazy nice chain \*/

new Config(0, 0, 0, 0, deflate\_stored), /\* 0 store only \*/

new Config(4, 4, 8, 4, deflate\_fast), /\* 1 max speed, no lazy matches \*/

new Config(4, 5, 16, 8, deflate\_fast), /\* 2 \*/

new Config(4, 6, 32, 32, deflate\_fast), /\* 3 \*/

new Config(4, 4, 16, 16, deflate\_slow), /\* 4 lazy matches \*/

new Config(8, 16, 32, 32, deflate\_slow), /\* 5 \*/

new Config(8, 16, 128, 128, deflate\_slow), /\* 6 \*/

new Config(8, 32, 128, 256, deflate\_slow), /\* 7 \*/

new Config(32, 128, 258, 1024, deflate\_slow), /\* 8 \*/

new Config(32, 258, 258, 4096, deflate\_slow) /\* 9 max compression \*/

];

/\* ===========================================================================

\* Initialize the "longest match" routines for a new zlib stream

\*/

function lm\_init(s) {

s.window\_size = 2 \* s.w\_size;

/\*\*\* CLEAR\_HASH(s); \*\*\*/

zero(s.head); // Fill with NIL (= 0);

/\* Set the default configuration parameters:

\*/

s.max\_lazy\_match = configuration\_table[s.level].max\_lazy;

s.good\_match = configuration\_table[s.level].good\_length;

s.nice\_match = configuration\_table[s.level].nice\_length;

s.max\_chain\_length = configuration\_table[s.level].max\_chain;

s.strstart = 0;

s.block\_start = 0;

s.lookahead = 0;

s.insert = 0;

s.match\_length = s.prev\_length = MIN\_MATCH - 1;

s.match\_available = 0;

s.ins\_h = 0;

}

function DeflateState() {

this.strm = null; /\* pointer back to this zlib stream \*/

this.status = 0; /\* as the name implies \*/

this.pending\_buf = null; /\* output still pending \*/

this.pending\_buf\_size = 0; /\* size of pending\_buf \*/

this.pending\_out = 0; /\* next pending byte to output to the stream \*/

this.pending = 0; /\* nb of bytes in the pending buffer \*/

this.wrap = 0; /\* bit 0 true for zlib, bit 1 true for gzip \*/

this.gzhead = null; /\* gzip header information to write \*/

this.gzindex = 0; /\* where in extra, name, or comment \*/

this.method = Z\_DEFLATED; /\* can only be DEFLATED \*/

this.last\_flush = -1; /\* value of flush param for previous deflate call \*/

this.w\_size = 0; /\* LZ77 window size (32K by default) \*/

this.w\_bits = 0; /\* log2(w\_size) (8..16) \*/

this.w\_mask = 0; /\* w\_size - 1 \*/

this.window = null;

/\* Sliding window. Input bytes are read into the second half of the window,

\* and move to the first half later to keep a dictionary of at least wSize

\* bytes. With this organization, matches are limited to a distance of

\* wSize-MAX\_MATCH bytes, but this ensures that IO is always

\* performed with a length multiple of the block size.

\*/

this.window\_size = 0;

/\* Actual size of window: 2\*wSize, except when the user input buffer

\* is directly used as sliding window.

\*/

this.prev = null;

/\* Link to older string with same hash index. To limit the size of this

\* array to 64K, this link is maintained only for the last 32K strings.

\* An index in this array is thus a window index modulo 32K.

\*/

this.head = null; /\* Heads of the hash chains or NIL. \*/

this.ins\_h = 0; /\* hash index of string to be inserted \*/

this.hash\_size = 0; /\* number of elements in hash table \*/

this.hash\_bits = 0; /\* log2(hash\_size) \*/

this.hash\_mask = 0; /\* hash\_size-1 \*/

this.hash\_shift = 0;

/\* Number of bits by which ins\_h must be shifted at each input

\* step. It must be such that after MIN\_MATCH steps, the oldest

\* byte no longer takes part in the hash key, that is:

\* hash\_shift \* MIN\_MATCH >= hash\_bits

\*/

this.block\_start = 0;

/\* Window position at the beginning of the current output block. Gets

\* negative when the window is moved backwards.

\*/

this.match\_length = 0; /\* length of best match \*/

this.prev\_match = 0; /\* previous match \*/

this.match\_available = 0; /\* set if previous match exists \*/

this.strstart = 0; /\* start of string to insert \*/

this.match\_start = 0; /\* start of matching string \*/

this.lookahead = 0; /\* number of valid bytes ahead in window \*/

this.prev\_length = 0;

/\* Length of the best match at previous step. Matches not greater than this

\* are discarded. This is used in the lazy match evaluation.

\*/

this.max\_chain\_length = 0;

/\* To speed up deflation, hash chains are never searched beyond this

\* length. A higher limit improves compression ratio but degrades the

\* speed.

\*/

this.max\_lazy\_match = 0;

/\* Attempt to find a better match only when the current match is strictly

\* smaller than this value. This mechanism is used only for compression

\* levels >= 4.

\*/

// That's alias to max\_lazy\_match, don't use directly

//this.max\_insert\_length = 0;

/\* Insert new strings in the hash table only if the match length is not

\* greater than this length. This saves time but degrades compression.

\* max\_insert\_length is used only for compression levels <= 3.

\*/

this.level = 0; /\* compression level (1..9) \*/

this.strategy = 0; /\* favor or force Huffman coding\*/

this.good\_match = 0;

/\* Use a faster search when the previous match is longer than this \*/

this.nice\_match = 0; /\* Stop searching when current match exceeds this \*/

/\* used by trees.c: \*/

/\* Didn't use ct\_data typedef below to suppress compiler warning \*/

// struct ct\_data\_s dyn\_ltree[HEAP\_SIZE]; /\* literal and length tree \*/

// struct ct\_data\_s dyn\_dtree[2\*D\_CODES+1]; /\* distance tree \*/

// struct ct\_data\_s bl\_tree[2\*BL\_CODES+1]; /\* Huffman tree for bit lengths \*/

// Use flat array of DOUBLE size, with interleaved fata,

// because JS does not support effective

this.dyn\_ltree = new utils.Buf16(HEAP\_SIZE \* 2);

this.dyn\_dtree = new utils.Buf16((2 \* D\_CODES + 1) \* 2);

this.bl\_tree = new utils.Buf16((2 \* BL\_CODES + 1) \* 2);

zero(this.dyn\_ltree);

zero(this.dyn\_dtree);

zero(this.bl\_tree);

this.l\_desc = null; /\* desc. for literal tree \*/

this.d\_desc = null; /\* desc. for distance tree \*/

this.bl\_desc = null; /\* desc. for bit length tree \*/

//ush bl\_count[MAX\_BITS+1];

this.bl\_count = new utils.Buf16(MAX\_BITS + 1);

/\* number of codes at each bit length for an optimal tree \*/

//int heap[2\*L\_CODES+1]; /\* heap used to build the Huffman trees \*/

this.heap = new utils.Buf16(2 \* L\_CODES + 1); /\* heap used to build the Huffman trees \*/

zero(this.heap);

this.heap\_len = 0; /\* number of elements in the heap \*/

this.heap\_max = 0; /\* element of largest frequency \*/

/\* The sons of heap[n] are heap[2\*n] and heap[2\*n+1]. heap[0] is not used.

\* The same heap array is used to build all trees.

\*/

this.depth = new utils.Buf16(2 \* L\_CODES + 1); //uch depth[2\*L\_CODES+1];

zero(this.depth);

/\* Depth of each subtree used as tie breaker for trees of equal frequency

\*/

this.l\_buf = 0; /\* buffer index for literals or lengths \*/

this.lit\_bufsize = 0;

/\* Size of match buffer for literals/lengths. There are 4 reasons for

\* limiting lit\_bufsize to 64K:

\* - frequencies can be kept in 16 bit counters

\* - if compression is not successful for the first block, all input

\* data is still in the window so we can still emit a stored block even

\* when input comes from standard input. (This can also be done for

\* all blocks if lit\_bufsize is not greater than 32K.)

\* - if compression is not successful for a file smaller than 64K, we can

\* even emit a stored file instead of a stored block (saving 5 bytes).

\* This is applicable only for zip (not gzip or zlib).

\* - creating new Huffman trees less frequently may not provide fast

\* adaptation to changes in the input data statistics. (Take for

\* example a binary file with poorly compressible code followed by

\* a highly compressible string table.) Smaller buffer sizes give

\* fast adaptation but have of course the overhead of transmitting

\* trees more frequently.

\* - I can't count above 4

\*/

this.last\_lit = 0; /\* running index in l\_buf \*/

this.d\_buf = 0;

/\* Buffer index for distances. To simplify the code, d\_buf and l\_buf have

\* the same number of elements. To use different lengths, an extra flag

\* array would be necessary.

\*/

this.opt\_len = 0; /\* bit length of current block with optimal trees \*/

this.static\_len = 0; /\* bit length of current block with static trees \*/

this.matches = 0; /\* number of string matches in current block \*/

this.insert = 0; /\* bytes at end of window left to insert \*/

this.bi\_buf = 0;

/\* Output buffer. bits are inserted starting at the bottom (least

\* significant bits).

\*/

this.bi\_valid = 0;

/\* Number of valid bits in bi\_buf. All bits above the last valid bit

\* are always zero.

\*/

// Used for window memory init. We safely ignore it for JS. That makes

// sense only for pointers and memory check tools.

//this.high\_water = 0;

/\* High water mark offset in window for initialized bytes -- bytes above

\* this are set to zero in order to avoid memory check warnings when

\* longest match routines access bytes past the input. This is then

\* updated to the new high water mark.

\*/

}

function deflateResetKeep(strm) {

var s;

if (!strm || !strm.state) {

return err(strm, Z\_STREAM\_ERROR);

}

strm.total\_in = strm.total\_out = 0;

strm.data\_type = Z\_UNKNOWN;

s = strm.state;

s.pending = 0;

s.pending\_out = 0;

if (s.wrap < 0) {

s.wrap = -s.wrap;

/\* was made negative by deflate(..., Z\_FINISH); \*/

}

s.status = (s.wrap ? INIT\_STATE : BUSY\_STATE);

strm.adler = (s.wrap === 2) ?

0 // crc32(0, Z\_NULL, 0)

:

1; // adler32(0, Z\_NULL, 0)

s.last\_flush = Z\_NO\_FLUSH;

trees.\_tr\_init(s);

return Z\_OK;

}

function deflateReset(strm) {

var ret = deflateResetKeep(strm);

if (ret === Z\_OK) {

lm\_init(strm.state);

}

return ret;

}

function deflateSetHeader(strm, head) {

if (!strm || !strm.state) { return Z\_STREAM\_ERROR; }

if (strm.state.wrap !== 2) { return Z\_STREAM\_ERROR; }

strm.state.gzhead = head;

return Z\_OK;

}

function deflateInit2(strm, level, method, windowBits, memLevel, strategy) {

if (!strm) { // === Z\_NULL

return Z\_STREAM\_ERROR;

}

var wrap = 1;

if (level === Z\_DEFAULT\_COMPRESSION) {

level = 6;

}

if (windowBits < 0) { /\* suppress zlib wrapper \*/

wrap = 0;

windowBits = -windowBits;

}

else if (windowBits > 15) {

wrap = 2; /\* write gzip wrapper instead \*/

windowBits -= 16;

}

if (memLevel < 1 || memLevel > MAX\_MEM\_LEVEL || method !== Z\_DEFLATED ||

windowBits < 8 || windowBits > 15 || level < 0 || level > 9 ||

strategy < 0 || strategy > Z\_FIXED) {

return err(strm, Z\_STREAM\_ERROR);

}

if (windowBits === 8) {

windowBits = 9;

}

/\* until 256-byte window bug fixed \*/

var s = new DeflateState();

strm.state = s;

s.strm = strm;

s.wrap = wrap;

s.gzhead = null;

s.w\_bits = windowBits;

s.w\_size = 1 << s.w\_bits;

s.w\_mask = s.w\_size - 1;

s.hash\_bits = memLevel + 7;

s.hash\_size = 1 << s.hash\_bits;

s.hash\_mask = s.hash\_size - 1;

s.hash\_shift = ~~((s.hash\_bits + MIN\_MATCH - 1) / MIN\_MATCH);

s.window = new utils.Buf8(s.w\_size \* 2);

s.head = new utils.Buf16(s.hash\_size);

s.prev = new utils.Buf16(s.w\_size);

// Don't need mem init magic for JS.

//s.high\_water = 0; /\* nothing written to s->window yet \*/

s.lit\_bufsize = 1 << (memLevel + 6); /\* 16K elements by default \*/

s.pending\_buf\_size = s.lit\_bufsize \* 4;

//overlay = (ushf \*) ZALLOC(strm, s->lit\_bufsize, sizeof(ush)+2);

//s->pending\_buf = (uchf \*) overlay;

s.pending\_buf = new utils.Buf8(s.pending\_buf\_size);

// It is offset from `s.pending\_buf` (size is `s.lit\_bufsize \* 2`)

//s->d\_buf = overlay + s->lit\_bufsize/sizeof(ush);

s.d\_buf = 1 \* s.lit\_bufsize;

//s->l\_buf = s->pending\_buf + (1+sizeof(ush))\*s->lit\_bufsize;

s.l\_buf = (1 + 2) \* s.lit\_bufsize;

s.level = level;

s.strategy = strategy;

s.method = method;

return deflateReset(strm);

}

function deflateInit(strm, level) {

return deflateInit2(strm, level, Z\_DEFLATED, MAX\_WBITS, DEF\_MEM\_LEVEL, Z\_DEFAULT\_STRATEGY);

}

function deflate(strm, flush) {

var old\_flush, s;

var beg, val; // for gzip header write only

if (!strm || !strm.state ||

flush > Z\_BLOCK || flush < 0) {

return strm ? err(strm, Z\_STREAM\_ERROR) : Z\_STREAM\_ERROR;

}

s = strm.state;

if (!strm.output ||

(!strm.input && strm.avail\_in !== 0) ||

(s.status === FINISH\_STATE && flush !== Z\_FINISH)) {

return err(strm, (strm.avail\_out === 0) ? Z\_BUF\_ERROR : Z\_STREAM\_ERROR);

}

s.strm = strm; /\* just in case \*/

old\_flush = s.last\_flush;

s.last\_flush = flush;

/\* Write the header \*/

if (s.status === INIT\_STATE) {

if (s.wrap === 2) { // GZIP header

strm.adler = 0; //crc32(0L, Z\_NULL, 0);

put\_byte(s, 31);

put\_byte(s, 139);

put\_byte(s, 8);

if (!s.gzhead) { // s->gzhead == Z\_NULL

put\_byte(s, 0);

put\_byte(s, 0);

put\_byte(s, 0);

put\_byte(s, 0);

put\_byte(s, 0);

put\_byte(s, s.level === 9 ? 2 :

(s.strategy >= Z\_HUFFMAN\_ONLY || s.level < 2 ?

4 : 0));

put\_byte(s, OS\_CODE);

s.status = BUSY\_STATE;

}

else {

put\_byte(s, (s.gzhead.text ? 1 : 0) +

(s.gzhead.hcrc ? 2 : 0) +

(!s.gzhead.extra ? 0 : 4) +

(!s.gzhead.name ? 0 : 8) +

(!s.gzhead.comment ? 0 : 16)

);

put\_byte(s, s.gzhead.time & 0xff);

put\_byte(s, (s.gzhead.time >> 8) & 0xff);

put\_byte(s, (s.gzhead.time >> 16) & 0xff);

put\_byte(s, (s.gzhead.time >> 24) & 0xff);

put\_byte(s, s.level === 9 ? 2 :

(s.strategy >= Z\_HUFFMAN\_ONLY || s.level < 2 ?

4 : 0));

put\_byte(s, s.gzhead.os & 0xff);

if (s.gzhead.extra && s.gzhead.extra.length) {

put\_byte(s, s.gzhead.extra.length & 0xff);

put\_byte(s, (s.gzhead.extra.length >> 8) & 0xff);

}

if (s.gzhead.hcrc) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending, 0);

}

s.gzindex = 0;

s.status = EXTRA\_STATE;

}

}

else // DEFLATE header

{

var header = (Z\_DEFLATED + ((s.w\_bits - 8) << 4)) << 8;

var level\_flags = -1;

if (s.strategy >= Z\_HUFFMAN\_ONLY || s.level < 2) {

level\_flags = 0;

} else if (s.level < 6) {

level\_flags = 1;

} else if (s.level === 6) {

level\_flags = 2;

} else {

level\_flags = 3;

}

header |= (level\_flags << 6);

if (s.strstart !== 0) { header |= PRESET\_DICT; }

header += 31 - (header % 31);

s.status = BUSY\_STATE;

putShortMSB(s, header);

/\* Save the adler32 of the preset dictionary: \*/

if (s.strstart !== 0) {

putShortMSB(s, strm.adler >>> 16);

putShortMSB(s, strm.adler & 0xffff);

}

strm.adler = 1; // adler32(0L, Z\_NULL, 0);

}

}

//#ifdef GZIP

if (s.status === EXTRA\_STATE) {

if (s.gzhead.extra/\* != Z\_NULL\*/) {

beg = s.pending; /\* start of bytes to update crc \*/

while (s.gzindex < (s.gzhead.extra.length & 0xffff)) {

if (s.pending === s.pending\_buf\_size) {

if (s.gzhead.hcrc && s.pending > beg) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending - beg, beg);

}

flush\_pending(strm);

beg = s.pending;

if (s.pending === s.pending\_buf\_size) {

break;

}

}

put\_byte(s, s.gzhead.extra[s.gzindex] & 0xff);

s.gzindex++;

}

if (s.gzhead.hcrc && s.pending > beg) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending - beg, beg);

}

if (s.gzindex === s.gzhead.extra.length) {

s.gzindex = 0;

s.status = NAME\_STATE;

}

}

else {

s.status = NAME\_STATE;

}

}

if (s.status === NAME\_STATE) {

if (s.gzhead.name/\* != Z\_NULL\*/) {

beg = s.pending; /\* start of bytes to update crc \*/

//int val;

do {

if (s.pending === s.pending\_buf\_size) {

if (s.gzhead.hcrc && s.pending > beg) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending - beg, beg);

}

flush\_pending(strm);

beg = s.pending;

if (s.pending === s.pending\_buf\_size) {

val = 1;

break;

}

}

// JS specific: little magic to add zero terminator to end of string

if (s.gzindex < s.gzhead.name.length) {

val = s.gzhead.name.charCodeAt(s.gzindex++) & 0xff;

} else {

val = 0;

}

put\_byte(s, val);

} while (val !== 0);

if (s.gzhead.hcrc && s.pending > beg) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending - beg, beg);

}

if (val === 0) {

s.gzindex = 0;

s.status = COMMENT\_STATE;

}

}

else {

s.status = COMMENT\_STATE;

}

}

if (s.status === COMMENT\_STATE) {

if (s.gzhead.comment/\* != Z\_NULL\*/) {

beg = s.pending; /\* start of bytes to update crc \*/

//int val;

do {

if (s.pending === s.pending\_buf\_size) {

if (s.gzhead.hcrc && s.pending > beg) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending - beg, beg);

}

flush\_pending(strm);

beg = s.pending;

if (s.pending === s.pending\_buf\_size) {

val = 1;

break;

}

}

// JS specific: little magic to add zero terminator to end of string

if (s.gzindex < s.gzhead.comment.length) {

val = s.gzhead.comment.charCodeAt(s.gzindex++) & 0xff;

} else {

val = 0;

}

put\_byte(s, val);

} while (val !== 0);

if (s.gzhead.hcrc && s.pending > beg) {

strm.adler = crc32(strm.adler, s.pending\_buf, s.pending - beg, beg);

}

if (val === 0) {

s.status = HCRC\_STATE;

}

}

else {

s.status = HCRC\_STATE;

}

}

if (s.status === HCRC\_STATE) {

if (s.gzhead.hcrc) {

if (s.pending + 2 > s.pending\_buf\_size) {

flush\_pending(strm);

}

if (s.pending + 2 <= s.pending\_buf\_size) {

put\_byte(s, strm.adler & 0xff);

put\_byte(s, (strm.adler >> 8) & 0xff);

strm.adler = 0; //crc32(0L, Z\_NULL, 0);

s.status = BUSY\_STATE;

}

}

else {

s.status = BUSY\_STATE;

}

}

//#endif

/\* Flush as much pending output as possible \*/

if (s.pending !== 0) {

flush\_pending(strm);

if (strm.avail\_out === 0) {

/\* Since avail\_out is 0, deflate will be called again with

\* more output space, but possibly with both pending and

\* avail\_in equal to zero. There won't be anything to do,

\* but this is not an error situation so make sure we

\* return OK instead of BUF\_ERROR at next call of deflate:

\*/

s.last\_flush = -1;

return Z\_OK;

}

/\* Make sure there is something to do and avoid duplicate consecutive

\* flushes. For repeated and useless calls with Z\_FINISH, we keep

\* returning Z\_STREAM\_END instead of Z\_BUF\_ERROR.

\*/

} else if (strm.avail\_in === 0 && rank(flush) <= rank(old\_flush) &&

flush !== Z\_FINISH) {

return err(strm, Z\_BUF\_ERROR);

}

/\* User must not provide more input after the first FINISH: \*/

if (s.status === FINISH\_STATE && strm.avail\_in !== 0) {

return err(strm, Z\_BUF\_ERROR);

}

/\* Start a new block or continue the current one.

\*/

if (strm.avail\_in !== 0 || s.lookahead !== 0 ||

(flush !== Z\_NO\_FLUSH && s.status !== FINISH\_STATE)) {

var bstate = (s.strategy === Z\_HUFFMAN\_ONLY) ? deflate\_huff(s, flush) :

(s.strategy === Z\_RLE ? deflate\_rle(s, flush) :

configuration\_table[s.level].func(s, flush));

if (bstate === BS\_FINISH\_STARTED || bstate === BS\_FINISH\_DONE) {

s.status = FINISH\_STATE;

}

if (bstate === BS\_NEED\_MORE || bstate === BS\_FINISH\_STARTED) {

if (strm.avail\_out === 0) {

s.last\_flush = -1;

/\* avoid BUF\_ERROR next call, see above \*/

}

return Z\_OK;

/\* If flush != Z\_NO\_FLUSH && avail\_out == 0, the next call

\* of deflate should use the same flush parameter to make sure

\* that the flush is complete. So we don't have to output an

\* empty block here, this will be done at next call. This also

\* ensures that for a very small output buffer, we emit at most

\* one empty block.

\*/

}

if (bstate === BS\_BLOCK\_DONE) {

if (flush === Z\_PARTIAL\_FLUSH) {

trees.\_tr\_align(s);

}

else if (flush !== Z\_BLOCK) { /\* FULL\_FLUSH or SYNC\_FLUSH \*/

trees.\_tr\_stored\_block(s, 0, 0, false);

/\* For a full flush, this empty block will be recognized

\* as a special marker by inflate\_sync().

\*/

if (flush === Z\_FULL\_FLUSH) {

/\*\*\* CLEAR\_HASH(s); \*\*\*/ /\* forget history \*/

zero(s.head); // Fill with NIL (= 0);

if (s.lookahead === 0) {

s.strstart = 0;

s.block\_start = 0;

s.insert = 0;

}

}

}

flush\_pending(strm);

if (strm.avail\_out === 0) {

s.last\_flush = -1; /\* avoid BUF\_ERROR at next call, see above \*/

return Z\_OK;

}

}

}

//Assert(strm->avail\_out > 0, "bug2");

//if (strm.avail\_out <= 0) { throw new Error("bug2");}

if (flush !== Z\_FINISH) { return Z\_OK; }

if (s.wrap <= 0) { return Z\_STREAM\_END; }

/\* Write the trailer \*/

if (s.wrap === 2) {

put\_byte(s, strm.adler & 0xff);

put\_byte(s, (strm.adler >> 8) & 0xff);

put\_byte(s, (strm.adler >> 16) & 0xff);

put\_byte(s, (strm.adler >> 24) & 0xff);

put\_byte(s, strm.total\_in & 0xff);

put\_byte(s, (strm.total\_in >> 8) & 0xff);

put\_byte(s, (strm.total\_in >> 16) & 0xff);

put\_byte(s, (strm.total\_in >> 24) & 0xff);

}

else

{

putShortMSB(s, strm.adler >>> 16);

putShortMSB(s, strm.adler & 0xffff);

}

flush\_pending(strm);

/\* If avail\_out is zero, the application will call deflate again

\* to flush the rest.

\*/

if (s.wrap > 0) { s.wrap = -s.wrap; }

/\* write the trailer only once! \*/

return s.pending !== 0 ? Z\_OK : Z\_STREAM\_END;

}

function deflateEnd(strm) {

var status;

if (!strm/\*== Z\_NULL\*/ || !strm.state/\*== Z\_NULL\*/) {

return Z\_STREAM\_ERROR;

}

status = strm.state.status;

if (status !== INIT\_STATE &&

status !== EXTRA\_STATE &&

status !== NAME\_STATE &&

status !== COMMENT\_STATE &&

status !== HCRC\_STATE &&

status !== BUSY\_STATE &&

status !== FINISH\_STATE

) {

return err(strm, Z\_STREAM\_ERROR);

}

strm.state = null;

return status === BUSY\_STATE ? err(strm, Z\_DATA\_ERROR) : Z\_OK;

}

/\* =========================================================================

\* Initializes the compression dictionary from the given byte

\* sequence without producing any compressed output.

\*/

function deflateSetDictionary(strm, dictionary) {

var dictLength = dictionary.length;

var s;

var str, n;

var wrap;

var avail;

var next;

var input;

var tmpDict;

if (!strm/\*== Z\_NULL\*/ || !strm.state/\*== Z\_NULL\*/) {

return Z\_STREAM\_ERROR;

}

s = strm.state;

wrap = s.wrap;

if (wrap === 2 || (wrap === 1 && s.status !== INIT\_STATE) || s.lookahead) {

return Z\_STREAM\_ERROR;

}

/\* when using zlib wrappers, compute Adler-32 for provided dictionary \*/

if (wrap === 1) {

/\* adler32(strm->adler, dictionary, dictLength); \*/

strm.adler = adler32(strm.adler, dictionary, dictLength, 0);

}

s.wrap = 0; /\* avoid computing Adler-32 in read\_buf \*/

/\* if dictionary would fill window, just replace the history \*/

if (dictLength >= s.w\_size) {

if (wrap === 0) { /\* already empty otherwise \*/

/\*\*\* CLEAR\_HASH(s); \*\*\*/

zero(s.head); // Fill with NIL (= 0);

s.strstart = 0;

s.block\_start = 0;

s.insert = 0;

}

/\* use the tail \*/

// dictionary = dictionary.slice(dictLength - s.w\_size);

tmpDict = new utils.Buf8(s.w\_size);

utils.arraySet(tmpDict, dictionary, dictLength - s.w\_size, s.w\_size, 0);

dictionary = tmpDict;

dictLength = s.w\_size;

}

/\* insert dictionary into window and hash \*/

avail = strm.avail\_in;

next = strm.next\_in;

input = strm.input;

strm.avail\_in = dictLength;

strm.next\_in = 0;

strm.input = dictionary;

fill\_window(s);

while (s.lookahead >= MIN\_MATCH) {

str = s.strstart;

n = s.lookahead - (MIN\_MATCH - 1);

do {

/\* UPDATE\_HASH(s, s->ins\_h, s->window[str + MIN\_MATCH-1]); \*/

s.ins\_h = ((s.ins\_h << s.hash\_shift) ^ s.window[str + MIN\_MATCH - 1]) & s.hash\_mask;

s.prev[str & s.w\_mask] = s.head[s.ins\_h];

s.head[s.ins\_h] = str;

str++;

} while (--n);

s.strstart = str;

s.lookahead = MIN\_MATCH - 1;

fill\_window(s);

}

s.strstart += s.lookahead;

s.block\_start = s.strstart;

s.insert = s.lookahead;

s.lookahead = 0;

s.match\_length = s.prev\_length = MIN\_MATCH - 1;

s.match\_available = 0;

strm.next\_in = next;

strm.input = input;

strm.avail\_in = avail;

s.wrap = wrap;

return Z\_OK;

}

exports.deflateInit = deflateInit;

exports.deflateInit2 = deflateInit2;

exports.deflateReset = deflateReset;

exports.deflateResetKeep = deflateResetKeep;

exports.deflateSetHeader = deflateSetHeader;

exports.deflate = deflate;

exports.deflateEnd = deflateEnd;

exports.deflateSetDictionary = deflateSetDictionary;

exports.deflateInfo = 'pako deflate (from Nodeca project)';

/\* Not implemented

exports.deflateBound = deflateBound;

exports.deflateCopy = deflateCopy;

exports.deflateParams = deflateParams;

exports.deflatePending = deflatePending;

exports.deflatePrime = deflatePrime;

exports.deflateTune = deflateTune;

\*/

},{"../utils/common":41,"./adler32":43,"./crc32":45,"./messages":51,"./trees":52}],47:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

function GZheader() {

/\* true if compressed data believed to be text \*/

this.text = 0;

/\* modification time \*/

this.time = 0;

/\* extra flags (not used when writing a gzip file) \*/

this.xflags = 0;

/\* operating system \*/

this.os = 0;

/\* pointer to extra field or Z\_NULL if none \*/

this.extra = null;

/\* extra field length (valid if extra != Z\_NULL) \*/

this.extra\_len = 0; // Actually, we don't need it in JS,

// but leave for few code modifications

//

// Setup limits is not necessary because in js we should not preallocate memory

// for inflate use constant limit in 65536 bytes

//

/\* space at extra (only when reading header) \*/

// this.extra\_max = 0;

/\* pointer to zero-terminated file name or Z\_NULL \*/

this.name = '';

/\* space at name (only when reading header) \*/

// this.name\_max = 0;

/\* pointer to zero-terminated comment or Z\_NULL \*/

this.comment = '';

/\* space at comment (only when reading header) \*/

// this.comm\_max = 0;

/\* true if there was or will be a header crc \*/

this.hcrc = 0;

/\* true when done reading gzip header (not used when writing a gzip file) \*/

this.done = false;

}

module.exports = GZheader;

},{}],48:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

// See state defs from inflate.js

var BAD = 30; /\* got a data error -- remain here until reset \*/

var TYPE = 12; /\* i: waiting for type bits, including last-flag bit \*/

/\*

Decode literal, length, and distance codes and write out the resulting

literal and match bytes until either not enough input or output is

available, an end-of-block is encountered, or a data error is encountered.

When large enough input and output buffers are supplied to inflate(), for

example, a 16K input buffer and a 64K output buffer, more than 95% of the

inflate execution time is spent in this routine.

Entry assumptions:

state.mode === LEN

strm.avail\_in >= 6

strm.avail\_out >= 258

start >= strm.avail\_out

state.bits < 8

On return, state.mode is one of:

LEN -- ran out of enough output space or enough available input

TYPE -- reached end of block code, inflate() to interpret next block

BAD -- error in block data

Notes:

- The maximum input bits used by a length/distance pair is 15 bits for the

length code, 5 bits for the length extra, 15 bits for the distance code,

and 13 bits for the distance extra. This totals 48 bits, or six bytes.

Therefore if strm.avail\_in >= 6, then there is enough input to avoid

checking for available input while decoding.

- The maximum bytes that a single length/distance pair can output is 258

bytes, which is the maximum length that can be coded. inflate\_fast()

requires strm.avail\_out >= 258 for each loop to avoid checking for

output space.

\*/

module.exports = function inflate\_fast(strm, start) {

var state;

var \_in; /\* local strm.input \*/

var last; /\* have enough input while in < last \*/

var \_out; /\* local strm.output \*/

var beg; /\* inflate()'s initial strm.output \*/

var end; /\* while out < end, enough space available \*/

//#ifdef INFLATE\_STRICT

var dmax; /\* maximum distance from zlib header \*/

//#endif

var wsize; /\* window size or zero if not using window \*/

var whave; /\* valid bytes in the window \*/

var wnext; /\* window write index \*/

// Use `s\_window` instead `window`, avoid conflict with instrumentation tools

var s\_window; /\* allocated sliding window, if wsize != 0 \*/

var hold; /\* local strm.hold \*/

var bits; /\* local strm.bits \*/

var lcode; /\* local strm.lencode \*/

var dcode; /\* local strm.distcode \*/

var lmask; /\* mask for first level of length codes \*/

var dmask; /\* mask for first level of distance codes \*/

var here; /\* retrieved table entry \*/

var op; /\* code bits, operation, extra bits, or \*/

/\* window position, window bytes to copy \*/

var len; /\* match length, unused bytes \*/

var dist; /\* match distance \*/

var from; /\* where to copy match from \*/

var from\_source;

var input, output; // JS specific, because we have no pointers

/\* copy state to local variables \*/

state = strm.state;

//here = state.here;

\_in = strm.next\_in;

input = strm.input;

last = \_in + (strm.avail\_in - 5);

\_out = strm.next\_out;

output = strm.output;

beg = \_out - (start - strm.avail\_out);

end = \_out + (strm.avail\_out - 257);

//#ifdef INFLATE\_STRICT

dmax = state.dmax;

//#endif

wsize = state.wsize;

whave = state.whave;

wnext = state.wnext;

s\_window = state.window;

hold = state.hold;

bits = state.bits;

lcode = state.lencode;

dcode = state.distcode;

lmask = (1 << state.lenbits) - 1;

dmask = (1 << state.distbits) - 1;

/\* decode literals and length/distances until end-of-block or not enough

input data or output space \*/

top:

do {

if (bits < 15) {

hold += input[\_in++] << bits;

bits += 8;

hold += input[\_in++] << bits;

bits += 8;

}

here = lcode[hold & lmask];

dolen:

for (;;) { // Goto emulation

op = here >>> 24/\*here.bits\*/;

hold >>>= op;

bits -= op;

op = (here >>> 16) & 0xff/\*here.op\*/;

if (op === 0) { /\* literal \*/

//Tracevv((stderr, here.val >= 0x20 && here.val < 0x7f ?

// "inflate: literal '%c'\n" :

// "inflate: literal 0x%02x\n", here.val));

output[\_out++] = here & 0xffff/\*here.val\*/;

}

else if (op & 16) { /\* length base \*/

len = here & 0xffff/\*here.val\*/;

op &= 15; /\* number of extra bits \*/

if (op) {

if (bits < op) {

hold += input[\_in++] << bits;

bits += 8;

}

len += hold & ((1 << op) - 1);

hold >>>= op;

bits -= op;

}

//Tracevv((stderr, "inflate: length %u\n", len));

if (bits < 15) {

hold += input[\_in++] << bits;

bits += 8;

hold += input[\_in++] << bits;

bits += 8;

}

here = dcode[hold & dmask];

dodist:

for (;;) { // goto emulation

op = here >>> 24/\*here.bits\*/;

hold >>>= op;

bits -= op;

op = (here >>> 16) & 0xff/\*here.op\*/;

if (op & 16) { /\* distance base \*/

dist = here & 0xffff/\*here.val\*/;

op &= 15; /\* number of extra bits \*/

if (bits < op) {

hold += input[\_in++] << bits;

bits += 8;

if (bits < op) {

hold += input[\_in++] << bits;

bits += 8;

}

}

dist += hold & ((1 << op) - 1);

//#ifdef INFLATE\_STRICT

if (dist > dmax) {

strm.msg = 'invalid distance too far back';

state.mode = BAD;

break top;

}

//#endif

hold >>>= op;

bits -= op;

//Tracevv((stderr, "inflate: distance %u\n", dist));

op = \_out - beg; /\* max distance in output \*/

if (dist > op) { /\* see if copy from window \*/

op = dist - op; /\* distance back in window \*/

if (op > whave) {

if (state.sane) {

strm.msg = 'invalid distance too far back';

state.mode = BAD;

break top;

}

// (!) This block is disabled in zlib defailts,

// don't enable it for binary compatibility

//#ifdef INFLATE\_ALLOW\_INVALID\_DISTANCE\_TOOFAR\_ARRR

// if (len <= op - whave) {

// do {

// output[\_out++] = 0;

// } while (--len);

// continue top;

// }

// len -= op - whave;

// do {

// output[\_out++] = 0;

// } while (--op > whave);

// if (op === 0) {

// from = \_out - dist;

// do {

// output[\_out++] = output[from++];

// } while (--len);

// continue top;

// }

//#endif

}

from = 0; // window index

from\_source = s\_window;

if (wnext === 0) { /\* very common case \*/

from += wsize - op;

if (op < len) { /\* some from window \*/

len -= op;

do {

output[\_out++] = s\_window[from++];

} while (--op);

from = \_out - dist; /\* rest from output \*/

from\_source = output;

}

}

else if (wnext < op) { /\* wrap around window \*/

from += wsize + wnext - op;

op -= wnext;

if (op < len) { /\* some from end of window \*/

len -= op;

do {

output[\_out++] = s\_window[from++];

} while (--op);

from = 0;

if (wnext < len) { /\* some from start of window \*/

op = wnext;

len -= op;

do {

output[\_out++] = s\_window[from++];

} while (--op);

from = \_out - dist; /\* rest from output \*/

from\_source = output;

}

}

}

else { /\* contiguous in window \*/

from += wnext - op;

if (op < len) { /\* some from window \*/

len -= op;

do {

output[\_out++] = s\_window[from++];

} while (--op);

from = \_out - dist; /\* rest from output \*/

from\_source = output;

}

}

while (len > 2) {

output[\_out++] = from\_source[from++];

output[\_out++] = from\_source[from++];

output[\_out++] = from\_source[from++];

len -= 3;

}

if (len) {

output[\_out++] = from\_source[from++];

if (len > 1) {

output[\_out++] = from\_source[from++];

}

}

}

else {

from = \_out - dist; /\* copy direct from output \*/

do { /\* minimum length is three \*/

output[\_out++] = output[from++];

output[\_out++] = output[from++];

output[\_out++] = output[from++];

len -= 3;

} while (len > 2);

if (len) {

output[\_out++] = output[from++];

if (len > 1) {

output[\_out++] = output[from++];

}

}

}

}

else if ((op & 64) === 0) { /\* 2nd level distance code \*/

here = dcode[(here & 0xffff)/\*here.val\*/ + (hold & ((1 << op) - 1))];

continue dodist;

}

else {

strm.msg = 'invalid distance code';

state.mode = BAD;

break top;

}

break; // need to emulate goto via "continue"

}

}

else if ((op & 64) === 0) { /\* 2nd level length code \*/

here = lcode[(here & 0xffff)/\*here.val\*/ + (hold & ((1 << op) - 1))];

continue dolen;

}

else if (op & 32) { /\* end-of-block \*/

//Tracevv((stderr, "inflate: end of block\n"));

state.mode = TYPE;

break top;

}

else {

strm.msg = 'invalid literal/length code';

state.mode = BAD;

break top;

}

break; // need to emulate goto via "continue"

}

} while (\_in < last && \_out < end);

/\* return unused bytes (on entry, bits < 8, so in won't go too far back) \*/

len = bits >> 3;

\_in -= len;

bits -= len << 3;

hold &= (1 << bits) - 1;

/\* update state and return \*/

strm.next\_in = \_in;

strm.next\_out = \_out;

strm.avail\_in = (\_in < last ? 5 + (last - \_in) : 5 - (\_in - last));

strm.avail\_out = (\_out < end ? 257 + (end - \_out) : 257 - (\_out - end));

state.hold = hold;

state.bits = bits;

return;

};

},{}],49:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

var utils = require('../utils/common');

var adler32 = require('./adler32');

var crc32 = require('./crc32');

var inflate\_fast = require('./inffast');

var inflate\_table = require('./inftrees');

var CODES = 0;

var LENS = 1;

var DISTS = 2;

/\* Public constants ==========================================================\*/

/\* ===========================================================================\*/

/\* Allowed flush values; see deflate() and inflate() below for details \*/

//var Z\_NO\_FLUSH = 0;

//var Z\_PARTIAL\_FLUSH = 1;

//var Z\_SYNC\_FLUSH = 2;

//var Z\_FULL\_FLUSH = 3;

var Z\_FINISH = 4;

var Z\_BLOCK = 5;

var Z\_TREES = 6;

/\* Return codes for the compression/decompression functions. Negative values

\* are errors, positive values are used for special but normal events.

\*/

var Z\_OK = 0;

var Z\_STREAM\_END = 1;

var Z\_NEED\_DICT = 2;

//var Z\_ERRNO = -1;

var Z\_STREAM\_ERROR = -2;

var Z\_DATA\_ERROR = -3;

var Z\_MEM\_ERROR = -4;

var Z\_BUF\_ERROR = -5;

//var Z\_VERSION\_ERROR = -6;

/\* The deflate compression method \*/

var Z\_DEFLATED = 8;

/\* STATES ====================================================================\*/

/\* ===========================================================================\*/

var HEAD = 1; /\* i: waiting for magic header \*/

var FLAGS = 2; /\* i: waiting for method and flags (gzip) \*/

var TIME = 3; /\* i: waiting for modification time (gzip) \*/

var OS = 4; /\* i: waiting for extra flags and operating system (gzip) \*/

var EXLEN = 5; /\* i: waiting for extra length (gzip) \*/

var EXTRA = 6; /\* i: waiting for extra bytes (gzip) \*/

var NAME = 7; /\* i: waiting for end of file name (gzip) \*/

var COMMENT = 8; /\* i: waiting for end of comment (gzip) \*/

var HCRC = 9; /\* i: waiting for header crc (gzip) \*/

var DICTID = 10; /\* i: waiting for dictionary check value \*/

var DICT = 11; /\* waiting for inflateSetDictionary() call \*/

var TYPE = 12; /\* i: waiting for type bits, including last-flag bit \*/

var TYPEDO = 13; /\* i: same, but skip check to exit inflate on new block \*/

var STORED = 14; /\* i: waiting for stored size (length and complement) \*/

var COPY\_ = 15; /\* i/o: same as COPY below, but only first time in \*/

var COPY = 16; /\* i/o: waiting for input or output to copy stored block \*/

var TABLE = 17; /\* i: waiting for dynamic block table lengths \*/

var LENLENS = 18; /\* i: waiting for code length code lengths \*/

var CODELENS = 19; /\* i: waiting for length/lit and distance code lengths \*/

var LEN\_ = 20; /\* i: same as LEN below, but only first time in \*/

var LEN = 21; /\* i: waiting for length/lit/eob code \*/

var LENEXT = 22; /\* i: waiting for length extra bits \*/

var DIST = 23; /\* i: waiting for distance code \*/

var DISTEXT = 24; /\* i: waiting for distance extra bits \*/

var MATCH = 25; /\* o: waiting for output space to copy string \*/

var LIT = 26; /\* o: waiting for output space to write literal \*/

var CHECK = 27; /\* i: waiting for 32-bit check value \*/

var LENGTH = 28; /\* i: waiting for 32-bit length (gzip) \*/

var DONE = 29; /\* finished check, done -- remain here until reset \*/

var BAD = 30; /\* got a data error -- remain here until reset \*/

var MEM = 31; /\* got an inflate() memory error -- remain here until reset \*/

var SYNC = 32; /\* looking for synchronization bytes to restart inflate() \*/

/\* ===========================================================================\*/

var ENOUGH\_LENS = 852;

var ENOUGH\_DISTS = 592;

//var ENOUGH = (ENOUGH\_LENS+ENOUGH\_DISTS);

var MAX\_WBITS = 15;

/\* 32K LZ77 window \*/

var DEF\_WBITS = MAX\_WBITS;

function zswap32(q) {

return (((q >>> 24) & 0xff) +

((q >>> 8) & 0xff00) +

((q & 0xff00) << 8) +

((q & 0xff) << 24));

}

function InflateState() {

this.mode = 0; /\* current inflate mode \*/

this.last = false; /\* true if processing last block \*/

this.wrap = 0; /\* bit 0 true for zlib, bit 1 true for gzip \*/

this.havedict = false; /\* true if dictionary provided \*/

this.flags = 0; /\* gzip header method and flags (0 if zlib) \*/

this.dmax = 0; /\* zlib header max distance (INFLATE\_STRICT) \*/

this.check = 0; /\* protected copy of check value \*/

this.total = 0; /\* protected copy of output count \*/

// TODO: may be {}

this.head = null; /\* where to save gzip header information \*/

/\* sliding window \*/

this.wbits = 0; /\* log base 2 of requested window size \*/

this.wsize = 0; /\* window size or zero if not using window \*/

this.whave = 0; /\* valid bytes in the window \*/

this.wnext = 0; /\* window write index \*/

this.window = null; /\* allocated sliding window, if needed \*/

/\* bit accumulator \*/

this.hold = 0; /\* input bit accumulator \*/

this.bits = 0; /\* number of bits in "in" \*/

/\* for string and stored block copying \*/

this.length = 0; /\* literal or length of data to copy \*/

this.offset = 0; /\* distance back to copy string from \*/

/\* for table and code decoding \*/

this.extra = 0; /\* extra bits needed \*/

/\* fixed and dynamic code tables \*/

this.lencode = null; /\* starting table for length/literal codes \*/

this.distcode = null; /\* starting table for distance codes \*/

this.lenbits = 0; /\* index bits for lencode \*/

this.distbits = 0; /\* index bits for distcode \*/

/\* dynamic table building \*/

this.ncode = 0; /\* number of code length code lengths \*/

this.nlen = 0; /\* number of length code lengths \*/

this.ndist = 0; /\* number of distance code lengths \*/

this.have = 0; /\* number of code lengths in lens[] \*/

this.next = null; /\* next available space in codes[] \*/

this.lens = new utils.Buf16(320); /\* temporary storage for code lengths \*/

this.work = new utils.Buf16(288); /\* work area for code table building \*/

/\*

because we don't have pointers in js, we use lencode and distcode directly

as buffers so we don't need codes

\*/

//this.codes = new utils.Buf32(ENOUGH); /\* space for code tables \*/

this.lendyn = null; /\* dynamic table for length/literal codes (JS specific) \*/

this.distdyn = null; /\* dynamic table for distance codes (JS specific) \*/

this.sane = 0; /\* if false, allow invalid distance too far \*/

this.back = 0; /\* bits back of last unprocessed length/lit \*/

this.was = 0; /\* initial length of match \*/

}

function inflateResetKeep(strm) {

var state;

if (!strm || !strm.state) { return Z\_STREAM\_ERROR; }

state = strm.state;

strm.total\_in = strm.total\_out = state.total = 0;

strm.msg = ''; /\*Z\_NULL\*/

if (state.wrap) { /\* to support ill-conceived Java test suite \*/

strm.adler = state.wrap & 1;

}

state.mode = HEAD;

state.last = 0;

state.havedict = 0;

state.dmax = 32768;

state.head = null/\*Z\_NULL\*/;

state.hold = 0;

state.bits = 0;

//state.lencode = state.distcode = state.next = state.codes;

state.lencode = state.lendyn = new utils.Buf32(ENOUGH\_LENS);

state.distcode = state.distdyn = new utils.Buf32(ENOUGH\_DISTS);

state.sane = 1;

state.back = -1;

//Tracev((stderr, "inflate: reset\n"));

return Z\_OK;

}

function inflateReset(strm) {

var state;

if (!strm || !strm.state) { return Z\_STREAM\_ERROR; }

state = strm.state;

state.wsize = 0;

state.whave = 0;

state.wnext = 0;

return inflateResetKeep(strm);

}

function inflateReset2(strm, windowBits) {

var wrap;

var state;

/\* get the state \*/

if (!strm || !strm.state) { return Z\_STREAM\_ERROR; }

state = strm.state;

/\* extract wrap request from windowBits parameter \*/

if (windowBits < 0) {

wrap = 0;

windowBits = -windowBits;

}

else {

wrap = (windowBits >> 4) + 1;

if (windowBits < 48) {

windowBits &= 15;

}

}

/\* set number of window bits, free window if different \*/

if (windowBits && (windowBits < 8 || windowBits > 15)) {

return Z\_STREAM\_ERROR;

}

if (state.window !== null && state.wbits !== windowBits) {

state.window = null;

}

/\* update state and reset the rest of it \*/

state.wrap = wrap;

state.wbits = windowBits;

return inflateReset(strm);

}

function inflateInit2(strm, windowBits) {

var ret;

var state;

if (!strm) { return Z\_STREAM\_ERROR; }

//strm.msg = Z\_NULL; /\* in case we return an error \*/

state = new InflateState();

//if (state === Z\_NULL) return Z\_MEM\_ERROR;

//Tracev((stderr, "inflate: allocated\n"));

strm.state = state;

state.window = null/\*Z\_NULL\*/;

ret = inflateReset2(strm, windowBits);

if (ret !== Z\_OK) {

strm.state = null/\*Z\_NULL\*/;

}

return ret;

}

function inflateInit(strm) {

return inflateInit2(strm, DEF\_WBITS);

}

/\*

Return state with length and distance decoding tables and index sizes set to

fixed code decoding. Normally this returns fixed tables from inffixed.h.

If BUILDFIXED is defined, then instead this routine builds the tables the

first time it's called, and returns those tables the first time and

thereafter. This reduces the size of the code by about 2K bytes, in

exchange for a little execution time. However, BUILDFIXED should not be

used for threaded applications, since the rewriting of the tables and virgin

may not be thread-safe.

\*/

var virgin = true;

var lenfix, distfix; // We have no pointers in JS, so keep tables separate

function fixedtables(state) {

/\* build fixed huffman tables if first call (may not be thread safe) \*/

if (virgin) {

var sym;

lenfix = new utils.Buf32(512);

distfix = new utils.Buf32(32);

/\* literal/length table \*/

sym = 0;

while (sym < 144) { state.lens[sym++] = 8; }

while (sym < 256) { state.lens[sym++] = 9; }

while (sym < 280) { state.lens[sym++] = 7; }

while (sym < 288) { state.lens[sym++] = 8; }

inflate\_table(LENS, state.lens, 0, 288, lenfix, 0, state.work, { bits: 9 });

/\* distance table \*/

sym = 0;

while (sym < 32) { state.lens[sym++] = 5; }

inflate\_table(DISTS, state.lens, 0, 32, distfix, 0, state.work, { bits: 5 });

/\* do this just once \*/

virgin = false;

}

state.lencode = lenfix;

state.lenbits = 9;

state.distcode = distfix;

state.distbits = 5;

}

/\*

Update the window with the last wsize (normally 32K) bytes written before

returning. If window does not exist yet, create it. This is only called

when a window is already in use, or when output has been written during this

inflate call, but the end of the deflate stream has not been reached yet.

It is also called to create a window for dictionary data when a dictionary

is loaded.

Providing output buffers larger than 32K to inflate() should provide a speed

advantage, since only the last 32K of output is copied to the sliding window

upon return from inflate(), and since all distances after the first 32K of

output will fall in the output data, making match copies simpler and faster.

The advantage may be dependent on the size of the processor's data caches.

\*/

function updatewindow(strm, src, end, copy) {

var dist;

var state = strm.state;

/\* if it hasn't been done already, allocate space for the window \*/

if (state.window === null) {

state.wsize = 1 << state.wbits;

state.wnext = 0;

state.whave = 0;

state.window = new utils.Buf8(state.wsize);

}

/\* copy state->wsize or less output bytes into the circular window \*/

if (copy >= state.wsize) {

utils.arraySet(state.window, src, end - state.wsize, state.wsize, 0);

state.wnext = 0;

state.whave = state.wsize;

}

else {

dist = state.wsize - state.wnext;

if (dist > copy) {

dist = copy;

}

//zmemcpy(state->window + state->wnext, end - copy, dist);

utils.arraySet(state.window, src, end - copy, dist, state.wnext);

copy -= dist;

if (copy) {

//zmemcpy(state->window, end - copy, copy);

utils.arraySet(state.window, src, end - copy, copy, 0);

state.wnext = copy;

state.whave = state.wsize;

}

else {

state.wnext += dist;

if (state.wnext === state.wsize) { state.wnext = 0; }

if (state.whave < state.wsize) { state.whave += dist; }

}

}

return 0;

}

function inflate(strm, flush) {

var state;

var input, output; // input/output buffers

var next; /\* next input INDEX \*/

var put; /\* next output INDEX \*/

var have, left; /\* available input and output \*/

var hold; /\* bit buffer \*/

var bits; /\* bits in bit buffer \*/

var \_in, \_out; /\* save starting available input and output \*/

var copy; /\* number of stored or match bytes to copy \*/

var from; /\* where to copy match bytes from \*/

var from\_source;

var here = 0; /\* current decoding table entry \*/

var here\_bits, here\_op, here\_val; // paked "here" denormalized (JS specific)

//var last; /\* parent table entry \*/

var last\_bits, last\_op, last\_val; // paked "last" denormalized (JS specific)

var len; /\* length to copy for repeats, bits to drop \*/

var ret; /\* return code \*/

var hbuf = new utils.Buf8(4); /\* buffer for gzip header crc calculation \*/

var opts;

var n; // temporary var for NEED\_BITS

var order = /\* permutation of code lengths \*/

[ 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15 ];

if (!strm || !strm.state || !strm.output ||

(!strm.input && strm.avail\_in !== 0)) {

return Z\_STREAM\_ERROR;

}

state = strm.state;

if (state.mode === TYPE) { state.mode = TYPEDO; } /\* skip check \*/

//--- LOAD() ---

put = strm.next\_out;

output = strm.output;

left = strm.avail\_out;

next = strm.next\_in;

input = strm.input;

have = strm.avail\_in;

hold = state.hold;

bits = state.bits;

//---

\_in = have;

\_out = left;

ret = Z\_OK;

inf\_leave: // goto emulation

for (;;) {

switch (state.mode) {

case HEAD:

if (state.wrap === 0) {

state.mode = TYPEDO;

break;

}

//=== NEEDBITS(16);

while (bits < 16) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

if ((state.wrap & 2) && hold === 0x8b1f) { /\* gzip header \*/

state.check = 0/\*crc32(0L, Z\_NULL, 0)\*/;

//=== CRC2(state.check, hold);

hbuf[0] = hold & 0xff;

hbuf[1] = (hold >>> 8) & 0xff;

state.check = crc32(state.check, hbuf, 2, 0);

//===//

//=== INITBITS();

hold = 0;

bits = 0;

//===//

state.mode = FLAGS;

break;

}

state.flags = 0; /\* expect zlib header \*/

if (state.head) {

state.head.done = false;

}

if (!(state.wrap & 1) || /\* check if zlib header allowed \*/

(((hold & 0xff)/\*BITS(8)\*/ << 8) + (hold >> 8)) % 31) {

strm.msg = 'incorrect header check';

state.mode = BAD;

break;

}

if ((hold & 0x0f)/\*BITS(4)\*/ !== Z\_DEFLATED) {

strm.msg = 'unknown compression method';

state.mode = BAD;

break;

}

//--- DROPBITS(4) ---//

hold >>>= 4;

bits -= 4;

//---//

len = (hold & 0x0f)/\*BITS(4)\*/ + 8;

if (state.wbits === 0) {

state.wbits = len;

}

else if (len > state.wbits) {

strm.msg = 'invalid window size';

state.mode = BAD;

break;

}

state.dmax = 1 << len;

//Tracev((stderr, "inflate: zlib header ok\n"));

strm.adler = state.check = 1/\*adler32(0L, Z\_NULL, 0)\*/;

state.mode = hold & 0x200 ? DICTID : TYPE;

//=== INITBITS();

hold = 0;

bits = 0;

//===//

break;

case FLAGS:

//=== NEEDBITS(16); \*/

while (bits < 16) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.flags = hold;

if ((state.flags & 0xff) !== Z\_DEFLATED) {

strm.msg = 'unknown compression method';

state.mode = BAD;

break;

}

if (state.flags & 0xe000) {

strm.msg = 'unknown header flags set';

state.mode = BAD;

break;

}

if (state.head) {

state.head.text = ((hold >> 8) & 1);

}

if (state.flags & 0x0200) {

//=== CRC2(state.check, hold);

hbuf[0] = hold & 0xff;

hbuf[1] = (hold >>> 8) & 0xff;

state.check = crc32(state.check, hbuf, 2, 0);

//===//

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

state.mode = TIME;

/\* falls through \*/

case TIME:

//=== NEEDBITS(32); \*/

while (bits < 32) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

if (state.head) {

state.head.time = hold;

}

if (state.flags & 0x0200) {

//=== CRC4(state.check, hold)

hbuf[0] = hold & 0xff;

hbuf[1] = (hold >>> 8) & 0xff;

hbuf[2] = (hold >>> 16) & 0xff;

hbuf[3] = (hold >>> 24) & 0xff;

state.check = crc32(state.check, hbuf, 4, 0);

//===

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

state.mode = OS;

/\* falls through \*/

case OS:

//=== NEEDBITS(16); \*/

while (bits < 16) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

if (state.head) {

state.head.xflags = (hold & 0xff);

state.head.os = (hold >> 8);

}

if (state.flags & 0x0200) {

//=== CRC2(state.check, hold);

hbuf[0] = hold & 0xff;

hbuf[1] = (hold >>> 8) & 0xff;

state.check = crc32(state.check, hbuf, 2, 0);

//===//

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

state.mode = EXLEN;

/\* falls through \*/

case EXLEN:

if (state.flags & 0x0400) {

//=== NEEDBITS(16); \*/

while (bits < 16) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.length = hold;

if (state.head) {

state.head.extra\_len = hold;

}

if (state.flags & 0x0200) {

//=== CRC2(state.check, hold);

hbuf[0] = hold & 0xff;

hbuf[1] = (hold >>> 8) & 0xff;

state.check = crc32(state.check, hbuf, 2, 0);

//===//

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

}

else if (state.head) {

state.head.extra = null/\*Z\_NULL\*/;

}

state.mode = EXTRA;

/\* falls through \*/

case EXTRA:

if (state.flags & 0x0400) {

copy = state.length;

if (copy > have) { copy = have; }

if (copy) {

if (state.head) {

len = state.head.extra\_len - state.length;

if (!state.head.extra) {

// Use untyped array for more conveniend processing later

state.head.extra = new Array(state.head.extra\_len);

}

utils.arraySet(

state.head.extra,

input,

next,

// extra field is limited to 65536 bytes

// - no need for additional size check

copy,

/\*len + copy > state.head.extra\_max - len ? state.head.extra\_max : copy,\*/

len

);

//zmemcpy(state.head.extra + len, next,

// len + copy > state.head.extra\_max ?

// state.head.extra\_max - len : copy);

}

if (state.flags & 0x0200) {

state.check = crc32(state.check, input, copy, next);

}

have -= copy;

next += copy;

state.length -= copy;

}

if (state.length) { break inf\_leave; }

}

state.length = 0;

state.mode = NAME;

/\* falls through \*/

case NAME:

if (state.flags & 0x0800) {

if (have === 0) { break inf\_leave; }

copy = 0;

do {

// TODO: 2 or 1 bytes?

len = input[next + copy++];

/\* use constant limit because in js we should not preallocate memory \*/

if (state.head && len &&

(state.length < 65536 /\*state.head.name\_max\*/)) {

state.head.name += String.fromCharCode(len);

}

} while (len && copy < have);

if (state.flags & 0x0200) {

state.check = crc32(state.check, input, copy, next);

}

have -= copy;

next += copy;

if (len) { break inf\_leave; }

}

else if (state.head) {

state.head.name = null;

}

state.length = 0;

state.mode = COMMENT;

/\* falls through \*/

case COMMENT:

if (state.flags & 0x1000) {

if (have === 0) { break inf\_leave; }

copy = 0;

do {

len = input[next + copy++];

/\* use constant limit because in js we should not preallocate memory \*/

if (state.head && len &&

(state.length < 65536 /\*state.head.comm\_max\*/)) {

state.head.comment += String.fromCharCode(len);

}

} while (len && copy < have);

if (state.flags & 0x0200) {

state.check = crc32(state.check, input, copy, next);

}

have -= copy;

next += copy;

if (len) { break inf\_leave; }

}

else if (state.head) {

state.head.comment = null;

}

state.mode = HCRC;

/\* falls through \*/

case HCRC:

if (state.flags & 0x0200) {

//=== NEEDBITS(16); \*/

while (bits < 16) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

if (hold !== (state.check & 0xffff)) {

strm.msg = 'header crc mismatch';

state.mode = BAD;

break;

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

}

if (state.head) {

state.head.hcrc = ((state.flags >> 9) & 1);

state.head.done = true;

}

strm.adler = state.check = 0;

state.mode = TYPE;

break;

case DICTID:

//=== NEEDBITS(32); \*/

while (bits < 32) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

strm.adler = state.check = zswap32(hold);

//=== INITBITS();

hold = 0;

bits = 0;

//===//

state.mode = DICT;

/\* falls through \*/

case DICT:

if (state.havedict === 0) {

//--- RESTORE() ---

strm.next\_out = put;

strm.avail\_out = left;

strm.next\_in = next;

strm.avail\_in = have;

state.hold = hold;

state.bits = bits;

//---

return Z\_NEED\_DICT;

}

strm.adler = state.check = 1/\*adler32(0L, Z\_NULL, 0)\*/;

state.mode = TYPE;

/\* falls through \*/

case TYPE:

if (flush === Z\_BLOCK || flush === Z\_TREES) { break inf\_leave; }

/\* falls through \*/

case TYPEDO:

if (state.last) {

//--- BYTEBITS() ---//

hold >>>= bits & 7;

bits -= bits & 7;

//---//

state.mode = CHECK;

break;

}

//=== NEEDBITS(3); \*/

while (bits < 3) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.last = (hold & 0x01)/\*BITS(1)\*/;

//--- DROPBITS(1) ---//

hold >>>= 1;

bits -= 1;

//---//

switch ((hold & 0x03)/\*BITS(2)\*/) {

case 0: /\* stored block \*/

//Tracev((stderr, "inflate: stored block%s\n",

// state.last ? " (last)" : ""));

state.mode = STORED;

break;

case 1: /\* fixed block \*/

fixedtables(state);

//Tracev((stderr, "inflate: fixed codes block%s\n",

// state.last ? " (last)" : ""));

state.mode = LEN\_; /\* decode codes \*/

if (flush === Z\_TREES) {

//--- DROPBITS(2) ---//

hold >>>= 2;

bits -= 2;

//---//

break inf\_leave;

}

break;

case 2: /\* dynamic block \*/

//Tracev((stderr, "inflate: dynamic codes block%s\n",

// state.last ? " (last)" : ""));

state.mode = TABLE;

break;

case 3:

strm.msg = 'invalid block type';

state.mode = BAD;

}

//--- DROPBITS(2) ---//

hold >>>= 2;

bits -= 2;

//---//

break;

case STORED:

//--- BYTEBITS() ---// /\* go to byte boundary \*/

hold >>>= bits & 7;

bits -= bits & 7;

//---//

//=== NEEDBITS(32); \*/

while (bits < 32) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

if ((hold & 0xffff) !== ((hold >>> 16) ^ 0xffff)) {

strm.msg = 'invalid stored block lengths';

state.mode = BAD;

break;

}

state.length = hold & 0xffff;

//Tracev((stderr, "inflate: stored length %u\n",

// state.length));

//=== INITBITS();

hold = 0;

bits = 0;

//===//

state.mode = COPY\_;

if (flush === Z\_TREES) { break inf\_leave; }

/\* falls through \*/

case COPY\_:

state.mode = COPY;

/\* falls through \*/

case COPY:

copy = state.length;

if (copy) {

if (copy > have) { copy = have; }

if (copy > left) { copy = left; }

if (copy === 0) { break inf\_leave; }

//--- zmemcpy(put, next, copy); ---

utils.arraySet(output, input, next, copy, put);

//---//

have -= copy;

next += copy;

left -= copy;

put += copy;

state.length -= copy;

break;

}

//Tracev((stderr, "inflate: stored end\n"));

state.mode = TYPE;

break;

case TABLE:

//=== NEEDBITS(14); \*/

while (bits < 14) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.nlen = (hold & 0x1f)/\*BITS(5)\*/ + 257;

//--- DROPBITS(5) ---//

hold >>>= 5;

bits -= 5;

//---//

state.ndist = (hold & 0x1f)/\*BITS(5)\*/ + 1;

//--- DROPBITS(5) ---//

hold >>>= 5;

bits -= 5;

//---//

state.ncode = (hold & 0x0f)/\*BITS(4)\*/ + 4;

//--- DROPBITS(4) ---//

hold >>>= 4;

bits -= 4;

//---//

//#ifndef PKZIP\_BUG\_WORKAROUND

if (state.nlen > 286 || state.ndist > 30) {

strm.msg = 'too many length or distance symbols';

state.mode = BAD;

break;

}

//#endif

//Tracev((stderr, "inflate: table sizes ok\n"));

state.have = 0;

state.mode = LENLENS;

/\* falls through \*/

case LENLENS:

while (state.have < state.ncode) {

//=== NEEDBITS(3);

while (bits < 3) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.lens[order[state.have++]] = (hold & 0x07);//BITS(3);

//--- DROPBITS(3) ---//

hold >>>= 3;

bits -= 3;

//---//

}

while (state.have < 19) {

state.lens[order[state.have++]] = 0;

}

// We have separate tables & no pointers. 2 commented lines below not needed.

//state.next = state.codes;

//state.lencode = state.next;

// Switch to use dynamic table

state.lencode = state.lendyn;

state.lenbits = 7;

opts = { bits: state.lenbits };

ret = inflate\_table(CODES, state.lens, 0, 19, state.lencode, 0, state.work, opts);

state.lenbits = opts.bits;

if (ret) {

strm.msg = 'invalid code lengths set';

state.mode = BAD;

break;

}

//Tracev((stderr, "inflate: code lengths ok\n"));

state.have = 0;

state.mode = CODELENS;

/\* falls through \*/

case CODELENS:

while (state.have < state.nlen + state.ndist) {

for (;;) {

here = state.lencode[hold & ((1 << state.lenbits) - 1)];/\*BITS(state.lenbits)\*/

here\_bits = here >>> 24;

here\_op = (here >>> 16) & 0xff;

here\_val = here & 0xffff;

if ((here\_bits) <= bits) { break; }

//--- PULLBYTE() ---//

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

//---//

}

if (here\_val < 16) {

//--- DROPBITS(here.bits) ---//

hold >>>= here\_bits;

bits -= here\_bits;

//---//

state.lens[state.have++] = here\_val;

}

else {

if (here\_val === 16) {

//=== NEEDBITS(here.bits + 2);

n = here\_bits + 2;

while (bits < n) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

//--- DROPBITS(here.bits) ---//

hold >>>= here\_bits;

bits -= here\_bits;

//---//

if (state.have === 0) {

strm.msg = 'invalid bit length repeat';

state.mode = BAD;

break;

}

len = state.lens[state.have - 1];

copy = 3 + (hold & 0x03);//BITS(2);

//--- DROPBITS(2) ---//

hold >>>= 2;

bits -= 2;

//---//

}

else if (here\_val === 17) {

//=== NEEDBITS(here.bits + 3);

n = here\_bits + 3;

while (bits < n) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

//--- DROPBITS(here.bits) ---//

hold >>>= here\_bits;

bits -= here\_bits;

//---//

len = 0;

copy = 3 + (hold & 0x07);//BITS(3);

//--- DROPBITS(3) ---//

hold >>>= 3;

bits -= 3;

//---//

}

else {

//=== NEEDBITS(here.bits + 7);

n = here\_bits + 7;

while (bits < n) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

//--- DROPBITS(here.bits) ---//

hold >>>= here\_bits;

bits -= here\_bits;

//---//

len = 0;

copy = 11 + (hold & 0x7f);//BITS(7);

//--- DROPBITS(7) ---//

hold >>>= 7;

bits -= 7;

//---//

}

if (state.have + copy > state.nlen + state.ndist) {

strm.msg = 'invalid bit length repeat';

state.mode = BAD;

break;

}

while (copy--) {

state.lens[state.have++] = len;

}

}

}

/\* handle error breaks in while \*/

if (state.mode === BAD) { break; }

/\* check for end-of-block code (better have one) \*/

if (state.lens[256] === 0) {

strm.msg = 'invalid code -- missing end-of-block';

state.mode = BAD;

break;

}

/\* build code tables -- note: do not change the lenbits or distbits

values here (9 and 6) without reading the comments in inftrees.h

concerning the ENOUGH constants, which depend on those values \*/

state.lenbits = 9;

opts = { bits: state.lenbits };

ret = inflate\_table(LENS, state.lens, 0, state.nlen, state.lencode, 0, state.work, opts);

// We have separate tables & no pointers. 2 commented lines below not needed.

// state.next\_index = opts.table\_index;

state.lenbits = opts.bits;

// state.lencode = state.next;

if (ret) {

strm.msg = 'invalid literal/lengths set';

state.mode = BAD;

break;

}

state.distbits = 6;

//state.distcode.copy(state.codes);

// Switch to use dynamic table

state.distcode = state.distdyn;

opts = { bits: state.distbits };

ret = inflate\_table(DISTS, state.lens, state.nlen, state.ndist, state.distcode, 0, state.work, opts);

// We have separate tables & no pointers. 2 commented lines below not needed.

// state.next\_index = opts.table\_index;

state.distbits = opts.bits;

// state.distcode = state.next;

if (ret) {

strm.msg = 'invalid distances set';

state.mode = BAD;

break;

}

//Tracev((stderr, 'inflate: codes ok\n'));

state.mode = LEN\_;

if (flush === Z\_TREES) { break inf\_leave; }

/\* falls through \*/

case LEN\_:

state.mode = LEN;

/\* falls through \*/

case LEN:

if (have >= 6 && left >= 258) {

//--- RESTORE() ---

strm.next\_out = put;

strm.avail\_out = left;

strm.next\_in = next;

strm.avail\_in = have;

state.hold = hold;

state.bits = bits;

//---

inflate\_fast(strm, \_out);

//--- LOAD() ---

put = strm.next\_out;

output = strm.output;

left = strm.avail\_out;

next = strm.next\_in;

input = strm.input;

have = strm.avail\_in;

hold = state.hold;

bits = state.bits;

//---

if (state.mode === TYPE) {

state.back = -1;

}

break;

}

state.back = 0;

for (;;) {

here = state.lencode[hold & ((1 << state.lenbits) - 1)]; /\*BITS(state.lenbits)\*/

here\_bits = here >>> 24;

here\_op = (here >>> 16) & 0xff;

here\_val = here & 0xffff;

if (here\_bits <= bits) { break; }

//--- PULLBYTE() ---//

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

//---//

}

if (here\_op && (here\_op & 0xf0) === 0) {

last\_bits = here\_bits;

last\_op = here\_op;

last\_val = here\_val;

for (;;) {

here = state.lencode[last\_val +

((hold & ((1 << (last\_bits + last\_op)) - 1))/\*BITS(last.bits + last.op)\*/ >> last\_bits)];

here\_bits = here >>> 24;

here\_op = (here >>> 16) & 0xff;

here\_val = here & 0xffff;

if ((last\_bits + here\_bits) <= bits) { break; }

//--- PULLBYTE() ---//

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

//---//

}

//--- DROPBITS(last.bits) ---//

hold >>>= last\_bits;

bits -= last\_bits;

//---//

state.back += last\_bits;

}

//--- DROPBITS(here.bits) ---//

hold >>>= here\_bits;

bits -= here\_bits;

//---//

state.back += here\_bits;

state.length = here\_val;

if (here\_op === 0) {

//Tracevv((stderr, here.val >= 0x20 && here.val < 0x7f ?

// "inflate: literal '%c'\n" :

// "inflate: literal 0x%02x\n", here.val));

state.mode = LIT;

break;

}

if (here\_op & 32) {

//Tracevv((stderr, "inflate: end of block\n"));

state.back = -1;

state.mode = TYPE;

break;

}

if (here\_op & 64) {

strm.msg = 'invalid literal/length code';

state.mode = BAD;

break;

}

state.extra = here\_op & 15;

state.mode = LENEXT;

/\* falls through \*/

case LENEXT:

if (state.extra) {

//=== NEEDBITS(state.extra);

n = state.extra;

while (bits < n) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.length += hold & ((1 << state.extra) - 1)/\*BITS(state.extra)\*/;

//--- DROPBITS(state.extra) ---//

hold >>>= state.extra;

bits -= state.extra;

//---//

state.back += state.extra;

}

//Tracevv((stderr, "inflate: length %u\n", state.length));

state.was = state.length;

state.mode = DIST;

/\* falls through \*/

case DIST:

for (;;) {

here = state.distcode[hold & ((1 << state.distbits) - 1)];/\*BITS(state.distbits)\*/

here\_bits = here >>> 24;

here\_op = (here >>> 16) & 0xff;

here\_val = here & 0xffff;

if ((here\_bits) <= bits) { break; }

//--- PULLBYTE() ---//

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

//---//

}

if ((here\_op & 0xf0) === 0) {

last\_bits = here\_bits;

last\_op = here\_op;

last\_val = here\_val;

for (;;) {

here = state.distcode[last\_val +

((hold & ((1 << (last\_bits + last\_op)) - 1))/\*BITS(last.bits + last.op)\*/ >> last\_bits)];

here\_bits = here >>> 24;

here\_op = (here >>> 16) & 0xff;

here\_val = here & 0xffff;

if ((last\_bits + here\_bits) <= bits) { break; }

//--- PULLBYTE() ---//

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

//---//

}

//--- DROPBITS(last.bits) ---//

hold >>>= last\_bits;

bits -= last\_bits;

//---//

state.back += last\_bits;

}

//--- DROPBITS(here.bits) ---//

hold >>>= here\_bits;

bits -= here\_bits;

//---//

state.back += here\_bits;

if (here\_op & 64) {

strm.msg = 'invalid distance code';

state.mode = BAD;

break;

}

state.offset = here\_val;

state.extra = (here\_op) & 15;

state.mode = DISTEXT;

/\* falls through \*/

case DISTEXT:

if (state.extra) {

//=== NEEDBITS(state.extra);

n = state.extra;

while (bits < n) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

state.offset += hold & ((1 << state.extra) - 1)/\*BITS(state.extra)\*/;

//--- DROPBITS(state.extra) ---//

hold >>>= state.extra;

bits -= state.extra;

//---//

state.back += state.extra;

}

//#ifdef INFLATE\_STRICT

if (state.offset > state.dmax) {

strm.msg = 'invalid distance too far back';

state.mode = BAD;

break;

}

//#endif

//Tracevv((stderr, "inflate: distance %u\n", state.offset));

state.mode = MATCH;

/\* falls through \*/

case MATCH:

if (left === 0) { break inf\_leave; }

copy = \_out - left;

if (state.offset > copy) { /\* copy from window \*/

copy = state.offset - copy;

if (copy > state.whave) {

if (state.sane) {

strm.msg = 'invalid distance too far back';

state.mode = BAD;

break;

}

// (!) This block is disabled in zlib defailts,

// don't enable it for binary compatibility

//#ifdef INFLATE\_ALLOW\_INVALID\_DISTANCE\_TOOFAR\_ARRR

// Trace((stderr, "inflate.c too far\n"));

// copy -= state.whave;

// if (copy > state.length) { copy = state.length; }

// if (copy > left) { copy = left; }

// left -= copy;

// state.length -= copy;

// do {

// output[put++] = 0;

// } while (--copy);

// if (state.length === 0) { state.mode = LEN; }

// break;

//#endif

}

if (copy > state.wnext) {

copy -= state.wnext;

from = state.wsize - copy;

}

else {

from = state.wnext - copy;

}

if (copy > state.length) { copy = state.length; }

from\_source = state.window;

}

else { /\* copy from output \*/

from\_source = output;

from = put - state.offset;

copy = state.length;

}

if (copy > left) { copy = left; }

left -= copy;

state.length -= copy;

do {

output[put++] = from\_source[from++];

} while (--copy);

if (state.length === 0) { state.mode = LEN; }

break;

case LIT:

if (left === 0) { break inf\_leave; }

output[put++] = state.length;

left--;

state.mode = LEN;

break;

case CHECK:

if (state.wrap) {

//=== NEEDBITS(32);

while (bits < 32) {

if (have === 0) { break inf\_leave; }

have--;

// Use '|' insdead of '+' to make sure that result is signed

hold |= input[next++] << bits;

bits += 8;

}

//===//

\_out -= left;

strm.total\_out += \_out;

state.total += \_out;

if (\_out) {

strm.adler = state.check =

/\*UPDATE(state.check, put - \_out, \_out);\*/

(state.flags ? crc32(state.check, output, \_out, put - \_out) : adler32(state.check, output, \_out, put - \_out));

}

\_out = left;

// NB: crc32 stored as signed 32-bit int, zswap32 returns signed too

if ((state.flags ? hold : zswap32(hold)) !== state.check) {

strm.msg = 'incorrect data check';

state.mode = BAD;

break;

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

//Tracev((stderr, "inflate: check matches trailer\n"));

}

state.mode = LENGTH;

/\* falls through \*/

case LENGTH:

if (state.wrap && state.flags) {

//=== NEEDBITS(32);

while (bits < 32) {

if (have === 0) { break inf\_leave; }

have--;

hold += input[next++] << bits;

bits += 8;

}

//===//

if (hold !== (state.total & 0xffffffff)) {

strm.msg = 'incorrect length check';

state.mode = BAD;

break;

}

//=== INITBITS();

hold = 0;

bits = 0;

//===//

//Tracev((stderr, "inflate: length matches trailer\n"));

}

state.mode = DONE;

/\* falls through \*/

case DONE:

ret = Z\_STREAM\_END;

break inf\_leave;

case BAD:

ret = Z\_DATA\_ERROR;

break inf\_leave;

case MEM:

return Z\_MEM\_ERROR;

case SYNC:

/\* falls through \*/

default:

return Z\_STREAM\_ERROR;

}

}

// inf\_leave <- here is real place for "goto inf\_leave", emulated via "break inf\_leave"

/\*

Return from inflate(), updating the total counts and the check value.

If there was no progress during the inflate() call, return a buffer

error. Call updatewindow() to create and/or update the window state.

Note: a memory error from inflate() is non-recoverable.

\*/

//--- RESTORE() ---

strm.next\_out = put;

strm.avail\_out = left;

strm.next\_in = next;

strm.avail\_in = have;

state.hold = hold;

state.bits = bits;

//---

if (state.wsize || (\_out !== strm.avail\_out && state.mode < BAD &&

(state.mode < CHECK || flush !== Z\_FINISH))) {

if (updatewindow(strm, strm.output, strm.next\_out, \_out - strm.avail\_out)) {

state.mode = MEM;

return Z\_MEM\_ERROR;

}

}

\_in -= strm.avail\_in;

\_out -= strm.avail\_out;

strm.total\_in += \_in;

strm.total\_out += \_out;

state.total += \_out;

if (state.wrap && \_out) {

strm.adler = state.check = /\*UPDATE(state.check, strm.next\_out - \_out, \_out);\*/

(state.flags ? crc32(state.check, output, \_out, strm.next\_out - \_out) : adler32(state.check, output, \_out, strm.next\_out - \_out));

}

strm.data\_type = state.bits + (state.last ? 64 : 0) +

(state.mode === TYPE ? 128 : 0) +

(state.mode === LEN\_ || state.mode === COPY\_ ? 256 : 0);

if (((\_in === 0 && \_out === 0) || flush === Z\_FINISH) && ret === Z\_OK) {

ret = Z\_BUF\_ERROR;

}

return ret;

}

function inflateEnd(strm) {

if (!strm || !strm.state /\*|| strm->zfree == (free\_func)0\*/) {

return Z\_STREAM\_ERROR;

}

var state = strm.state;

if (state.window) {

state.window = null;

}

strm.state = null;

return Z\_OK;

}

function inflateGetHeader(strm, head) {

var state;

/\* check state \*/

if (!strm || !strm.state) { return Z\_STREAM\_ERROR; }

state = strm.state;

if ((state.wrap & 2) === 0) { return Z\_STREAM\_ERROR; }

/\* save header structure \*/

state.head = head;

head.done = false;

return Z\_OK;

}

function inflateSetDictionary(strm, dictionary) {

var dictLength = dictionary.length;

var state;

var dictid;

var ret;

/\* check state \*/

if (!strm /\* == Z\_NULL \*/ || !strm.state /\* == Z\_NULL \*/) { return Z\_STREAM\_ERROR; }

state = strm.state;

if (state.wrap !== 0 && state.mode !== DICT) {

return Z\_STREAM\_ERROR;

}

/\* check for correct dictionary identifier \*/

if (state.mode === DICT) {

dictid = 1; /\* adler32(0, null, 0)\*/

/\* dictid = adler32(dictid, dictionary, dictLength); \*/

dictid = adler32(dictid, dictionary, dictLength, 0);

if (dictid !== state.check) {

return Z\_DATA\_ERROR;

}

}

/\* copy dictionary to window using updatewindow(), which will amend the

existing dictionary if appropriate \*/

ret = updatewindow(strm, dictionary, dictLength, dictLength);

if (ret) {

state.mode = MEM;

return Z\_MEM\_ERROR;

}

state.havedict = 1;

// Tracev((stderr, "inflate: dictionary set\n"));

return Z\_OK;

}

exports.inflateReset = inflateReset;

exports.inflateReset2 = inflateReset2;

exports.inflateResetKeep = inflateResetKeep;

exports.inflateInit = inflateInit;

exports.inflateInit2 = inflateInit2;

exports.inflate = inflate;

exports.inflateEnd = inflateEnd;

exports.inflateGetHeader = inflateGetHeader;

exports.inflateSetDictionary = inflateSetDictionary;

exports.inflateInfo = 'pako inflate (from Nodeca project)';

/\* Not implemented

exports.inflateCopy = inflateCopy;

exports.inflateGetDictionary = inflateGetDictionary;

exports.inflateMark = inflateMark;

exports.inflatePrime = inflatePrime;

exports.inflateSync = inflateSync;

exports.inflateSyncPoint = inflateSyncPoint;

exports.inflateUndermine = inflateUndermine;

\*/

},{"../utils/common":41,"./adler32":43,"./crc32":45,"./inffast":48,"./inftrees":50}],50:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

var utils = require('../utils/common');

var MAXBITS = 15;

var ENOUGH\_LENS = 852;

var ENOUGH\_DISTS = 592;

//var ENOUGH = (ENOUGH\_LENS+ENOUGH\_DISTS);

var CODES = 0;

var LENS = 1;

var DISTS = 2;

var lbase = [ /\* Length codes 257..285 base \*/

3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15, 17, 19, 23, 27, 31,

35, 43, 51, 59, 67, 83, 99, 115, 131, 163, 195, 227, 258, 0, 0

];

var lext = [ /\* Length codes 257..285 extra \*/

16, 16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 18, 18, 18, 18,

19, 19, 19, 19, 20, 20, 20, 20, 21, 21, 21, 21, 16, 72, 78

];

var dbase = [ /\* Distance codes 0..29 base \*/

1, 2, 3, 4, 5, 7, 9, 13, 17, 25, 33, 49, 65, 97, 129, 193,

257, 385, 513, 769, 1025, 1537, 2049, 3073, 4097, 6145,

8193, 12289, 16385, 24577, 0, 0

];

var dext = [ /\* Distance codes 0..29 extra \*/

16, 16, 16, 16, 17, 17, 18, 18, 19, 19, 20, 20, 21, 21, 22, 22,

23, 23, 24, 24, 25, 25, 26, 26, 27, 27,

28, 28, 29, 29, 64, 64

];

module.exports = function inflate\_table(type, lens, lens\_index, codes, table, table\_index, work, opts)

{

var bits = opts.bits;

//here = opts.here; /\* table entry for duplication \*/

var len = 0; /\* a code's length in bits \*/

var sym = 0; /\* index of code symbols \*/

var min = 0, max = 0; /\* minimum and maximum code lengths \*/

var root = 0; /\* number of index bits for root table \*/

var curr = 0; /\* number of index bits for current table \*/

var drop = 0; /\* code bits to drop for sub-table \*/

var left = 0; /\* number of prefix codes available \*/

var used = 0; /\* code entries in table used \*/

var huff = 0; /\* Huffman code \*/

var incr; /\* for incrementing code, index \*/

var fill; /\* index for replicating entries \*/

var low; /\* low bits for current root entry \*/

var mask; /\* mask for low root bits \*/

var next; /\* next available space in table \*/

var base = null; /\* base value table to use \*/

var base\_index = 0;

// var shoextra; /\* extra bits table to use \*/

var end; /\* use base and extra for symbol > end \*/

var count = new utils.Buf16(MAXBITS + 1); //[MAXBITS+1]; /\* number of codes of each length \*/

var offs = new utils.Buf16(MAXBITS + 1); //[MAXBITS+1]; /\* offsets in table for each length \*/

var extra = null;

var extra\_index = 0;

var here\_bits, here\_op, here\_val;

/\*

Process a set of code lengths to create a canonical Huffman code. The

code lengths are lens[0..codes-1]. Each length corresponds to the

symbols 0..codes-1. The Huffman code is generated by first sorting the

symbols by length from short to long, and retaining the symbol order

for codes with equal lengths. Then the code starts with all zero bits

for the first code of the shortest length, and the codes are integer

increments for the same length, and zeros are appended as the length

increases. For the deflate format, these bits are stored backwards

from their more natural integer increment ordering, and so when the

decoding tables are built in the large loop below, the integer codes

are incremented backwards.

This routine assumes, but does not check, that all of the entries in

lens[] are in the range 0..MAXBITS. The caller must assure this.

1..MAXBITS is interpreted as that code length. zero means that that

symbol does not occur in this code.

The codes are sorted by computing a count of codes for each length,

creating from that a table of starting indices for each length in the

sorted table, and then entering the symbols in order in the sorted

table. The sorted table is work[], with that space being provided by

the caller.

The length counts are used for other purposes as well, i.e. finding

the minimum and maximum length codes, determining if there are any

codes at all, checking for a valid set of lengths, and looking ahead

at length counts to determine sub-table sizes when building the

decoding tables.

\*/

/\* accumulate lengths for codes (assumes lens[] all in 0..MAXBITS) \*/

for (len = 0; len <= MAXBITS; len++) {

count[len] = 0;

}

for (sym = 0; sym < codes; sym++) {

count[lens[lens\_index + sym]]++;

}

/\* bound code lengths, force root to be within code lengths \*/

root = bits;

for (max = MAXBITS; max >= 1; max--) {

if (count[max] !== 0) { break; }

}

if (root > max) {

root = max;

}

if (max === 0) { /\* no symbols to code at all \*/

//table.op[opts.table\_index] = 64; //here.op = (var char)64; /\* invalid code marker \*/

//table.bits[opts.table\_index] = 1; //here.bits = (var char)1;

//table.val[opts.table\_index++] = 0; //here.val = (var short)0;

table[table\_index++] = (1 << 24) | (64 << 16) | 0;

//table.op[opts.table\_index] = 64;

//table.bits[opts.table\_index] = 1;

//table.val[opts.table\_index++] = 0;

table[table\_index++] = (1 << 24) | (64 << 16) | 0;

opts.bits = 1;

return 0; /\* no symbols, but wait for decoding to report error \*/

}

for (min = 1; min < max; min++) {

if (count[min] !== 0) { break; }

}

if (root < min) {

root = min;

}

/\* check for an over-subscribed or incomplete set of lengths \*/

left = 1;

for (len = 1; len <= MAXBITS; len++) {

left <<= 1;

left -= count[len];

if (left < 0) {

return -1;

} /\* over-subscribed \*/

}

if (left > 0 && (type === CODES || max !== 1)) {

return -1; /\* incomplete set \*/

}

/\* generate offsets into symbol table for each length for sorting \*/

offs[1] = 0;

for (len = 1; len < MAXBITS; len++) {

offs[len + 1] = offs[len] + count[len];

}

/\* sort symbols by length, by symbol order within each length \*/

for (sym = 0; sym < codes; sym++) {

if (lens[lens\_index + sym] !== 0) {

work[offs[lens[lens\_index + sym]]++] = sym;

}

}

/\*

Create and fill in decoding tables. In this loop, the table being

filled is at next and has curr index bits. The code being used is huff

with length len. That code is converted to an index by dropping drop

bits off of the bottom. For codes where len is less than drop + curr,

those top drop + curr - len bits are incremented through all values to

fill the table with replicated entries.

root is the number of index bits for the root table. When len exceeds

root, sub-tables are created pointed to by the root entry with an index

of the low root bits of huff. This is saved in low to check for when a

new sub-table should be started. drop is zero when the root table is

being filled, and drop is root when sub-tables are being filled.

When a new sub-table is needed, it is necessary to look ahead in the

code lengths to determine what size sub-table is needed. The length

counts are used for this, and so count[] is decremented as codes are

entered in the tables.

used keeps track of how many table entries have been allocated from the

provided \*table space. It is checked for LENS and DIST tables against

the constants ENOUGH\_LENS and ENOUGH\_DISTS to guard against changes in

the initial root table size constants. See the comments in inftrees.h

for more information.

sym increments through all symbols, and the loop terminates when

all codes of length max, i.e. all codes, have been processed. This

routine permits incomplete codes, so another loop after this one fills

in the rest of the decoding tables with invalid code markers.

\*/

/\* set up for code type \*/

// poor man optimization - use if-else instead of switch,

// to avoid deopts in old v8

if (type === CODES) {

base = extra = work; /\* dummy value--not used \*/

end = 19;

} else if (type === LENS) {

base = lbase;

base\_index -= 257;

extra = lext;

extra\_index -= 257;

end = 256;

} else { /\* DISTS \*/

base = dbase;

extra = dext;

end = -1;

}

/\* initialize opts for loop \*/

huff = 0; /\* starting code \*/

sym = 0; /\* starting code symbol \*/

len = min; /\* starting code length \*/

next = table\_index; /\* current table to fill in \*/

curr = root; /\* current table index bits \*/

drop = 0; /\* current bits to drop from code for index \*/

low = -1; /\* trigger new sub-table when len > root \*/

used = 1 << root; /\* use root table entries \*/

mask = used - 1; /\* mask for comparing low \*/

/\* check available table space \*/

if ((type === LENS && used > ENOUGH\_LENS) ||

(type === DISTS && used > ENOUGH\_DISTS)) {

return 1;

}

/\* process all codes and make table entries \*/

for (;;) {

/\* create table entry \*/

here\_bits = len - drop;

if (work[sym] < end) {

here\_op = 0;

here\_val = work[sym];

}

else if (work[sym] > end) {

here\_op = extra[extra\_index + work[sym]];

here\_val = base[base\_index + work[sym]];

}

else {

here\_op = 32 + 64; /\* end of block \*/

here\_val = 0;

}

/\* replicate for those indices with low len bits equal to huff \*/

incr = 1 << (len - drop);

fill = 1 << curr;

min = fill; /\* save offset to next table \*/

do {

fill -= incr;

table[next + (huff >> drop) + fill] = (here\_bits << 24) | (here\_op << 16) | here\_val |0;

} while (fill !== 0);

/\* backwards increment the len-bit code huff \*/

incr = 1 << (len - 1);

while (huff & incr) {

incr >>= 1;

}

if (incr !== 0) {

huff &= incr - 1;

huff += incr;

} else {

huff = 0;

}

/\* go to next symbol, update count, len \*/

sym++;

if (--count[len] === 0) {

if (len === max) { break; }

len = lens[lens\_index + work[sym]];

}

/\* create new sub-table if needed \*/

if (len > root && (huff & mask) !== low) {

/\* if first time, transition to sub-tables \*/

if (drop === 0) {

drop = root;

}

/\* increment past last table \*/

next += min; /\* here min is 1 << curr \*/

/\* determine length of next table \*/

curr = len - drop;

left = 1 << curr;

while (curr + drop < max) {

left -= count[curr + drop];

if (left <= 0) { break; }

curr++;

left <<= 1;

}

/\* check for enough space \*/

used += 1 << curr;

if ((type === LENS && used > ENOUGH\_LENS) ||

(type === DISTS && used > ENOUGH\_DISTS)) {

return 1;

}

/\* point entry in root table to sub-table \*/

low = huff & mask;

/\*table.op[low] = curr;

table.bits[low] = root;

table.val[low] = next - opts.table\_index;\*/

table[low] = (root << 24) | (curr << 16) | (next - table\_index) |0;

}

}

/\* fill in remaining table entry if code is incomplete (guaranteed to have

at most one remaining entry, since if the code is incomplete, the

maximum code length that was allowed to get this far is one bit) \*/

if (huff !== 0) {

//table.op[next + huff] = 64; /\* invalid code marker \*/

//table.bits[next + huff] = len - drop;

//table.val[next + huff] = 0;

table[next + huff] = ((len - drop) << 24) | (64 << 16) |0;

}

/\* set return parameters \*/

//opts.table\_index += used;

opts.bits = root;

return 0;

};

},{"../utils/common":41}],51:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

module.exports = {

2: 'need dictionary', /\* Z\_NEED\_DICT 2 \*/

1: 'stream end', /\* Z\_STREAM\_END 1 \*/

0: '', /\* Z\_OK 0 \*/

'-1': 'file error', /\* Z\_ERRNO (-1) \*/

'-2': 'stream error', /\* Z\_STREAM\_ERROR (-2) \*/

'-3': 'data error', /\* Z\_DATA\_ERROR (-3) \*/

'-4': 'insufficient memory', /\* Z\_MEM\_ERROR (-4) \*/

'-5': 'buffer error', /\* Z\_BUF\_ERROR (-5) \*/

'-6': 'incompatible version' /\* Z\_VERSION\_ERROR (-6) \*/

};

},{}],52:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

var utils = require('../utils/common');

/\* Public constants ==========================================================\*/

/\* ===========================================================================\*/

//var Z\_FILTERED = 1;

//var Z\_HUFFMAN\_ONLY = 2;

//var Z\_RLE = 3;

var Z\_FIXED = 4;

//var Z\_DEFAULT\_STRATEGY = 0;

/\* Possible values of the data\_type field (though see inflate()) \*/

var Z\_BINARY = 0;

var Z\_TEXT = 1;

//var Z\_ASCII = 1; // = Z\_TEXT

var Z\_UNKNOWN = 2;

/\*============================================================================\*/

function zero(buf) { var len = buf.length; while (--len >= 0) { buf[len] = 0; } }

// From zutil.h

var STORED\_BLOCK = 0;

var STATIC\_TREES = 1;

var DYN\_TREES = 2;

/\* The three kinds of block type \*/

var MIN\_MATCH = 3;

var MAX\_MATCH = 258;

/\* The minimum and maximum match lengths \*/

// From deflate.h

/\* ===========================================================================

\* Internal compression state.

\*/

var LENGTH\_CODES = 29;

/\* number of length codes, not counting the special END\_BLOCK code \*/

var LITERALS = 256;

/\* number of literal bytes 0..255 \*/

var L\_CODES = LITERALS + 1 + LENGTH\_CODES;

/\* number of Literal or Length codes, including the END\_BLOCK code \*/

var D\_CODES = 30;

/\* number of distance codes \*/

var BL\_CODES = 19;

/\* number of codes used to transfer the bit lengths \*/

var HEAP\_SIZE = 2 \* L\_CODES + 1;

/\* maximum heap size \*/

var MAX\_BITS = 15;

/\* All codes must not exceed MAX\_BITS bits \*/

var Buf\_size = 16;

/\* size of bit buffer in bi\_buf \*/

/\* ===========================================================================

\* Constants

\*/

var MAX\_BL\_BITS = 7;

/\* Bit length codes must not exceed MAX\_BL\_BITS bits \*/

var END\_BLOCK = 256;

/\* end of block literal code \*/

var REP\_3\_6 = 16;

/\* repeat previous bit length 3-6 times (2 bits of repeat count) \*/

var REPZ\_3\_10 = 17;

/\* repeat a zero length 3-10 times (3 bits of repeat count) \*/

var REPZ\_11\_138 = 18;

/\* repeat a zero length 11-138 times (7 bits of repeat count) \*/

/\* eslint-disable comma-spacing,array-bracket-spacing \*/

var extra\_lbits = /\* extra bits for each length code \*/

[0,0,0,0,0,0,0,0,1,1,1,1,2,2,2,2,3,3,3,3,4,4,4,4,5,5,5,5,0];

var extra\_dbits = /\* extra bits for each distance code \*/

[0,0,0,0,1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8,9,9,10,10,11,11,12,12,13,13];

var extra\_blbits = /\* extra bits for each bit length code \*/

[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,3,7];

var bl\_order =

[16,17,18,0,8,7,9,6,10,5,11,4,12,3,13,2,14,1,15];

/\* eslint-enable comma-spacing,array-bracket-spacing \*/

/\* The lengths of the bit length codes are sent in order of decreasing

\* probability, to avoid transmitting the lengths for unused bit length codes.

\*/

/\* ===========================================================================

\* Local data. These are initialized only once.

\*/

// We pre-fill arrays with 0 to avoid uninitialized gaps

var DIST\_CODE\_LEN = 512; /\* see definition of array dist\_code below \*/

// !!!! Use flat array insdead of structure, Freq = i\*2, Len = i\*2+1

var static\_ltree = new Array((L\_CODES + 2) \* 2);

zero(static\_ltree);

/\* The static literal tree. Since the bit lengths are imposed, there is no

\* need for the L\_CODES extra codes used during heap construction. However

\* The codes 286 and 287 are needed to build a canonical tree (see \_tr\_init

\* below).

\*/

var static\_dtree = new Array(D\_CODES \* 2);

zero(static\_dtree);

/\* The static distance tree. (Actually a trivial tree since all codes use

\* 5 bits.)

\*/

var \_dist\_code = new Array(DIST\_CODE\_LEN);

zero(\_dist\_code);

/\* Distance codes. The first 256 values correspond to the distances

\* 3 .. 258, the last 256 values correspond to the top 8 bits of

\* the 15 bit distances.

\*/

var \_length\_code = new Array(MAX\_MATCH - MIN\_MATCH + 1);

zero(\_length\_code);

/\* length code for each normalized match length (0 == MIN\_MATCH) \*/

var base\_length = new Array(LENGTH\_CODES);

zero(base\_length);

/\* First normalized length for each code (0 = MIN\_MATCH) \*/

var base\_dist = new Array(D\_CODES);

zero(base\_dist);

/\* First normalized distance for each code (0 = distance of 1) \*/

function StaticTreeDesc(static\_tree, extra\_bits, extra\_base, elems, max\_length) {

this.static\_tree = static\_tree; /\* static tree or NULL \*/

this.extra\_bits = extra\_bits; /\* extra bits for each code or NULL \*/

this.extra\_base = extra\_base; /\* base index for extra\_bits \*/

this.elems = elems; /\* max number of elements in the tree \*/

this.max\_length = max\_length; /\* max bit length for the codes \*/

// show if `static\_tree` has data or dummy - needed for monomorphic objects

this.has\_stree = static\_tree && static\_tree.length;

}

var static\_l\_desc;

var static\_d\_desc;

var static\_bl\_desc;

function TreeDesc(dyn\_tree, stat\_desc) {

this.dyn\_tree = dyn\_tree; /\* the dynamic tree \*/

this.max\_code = 0; /\* largest code with non zero frequency \*/

this.stat\_desc = stat\_desc; /\* the corresponding static tree \*/

}

function d\_code(dist) {

return dist < 256 ? \_dist\_code[dist] : \_dist\_code[256 + (dist >>> 7)];

}

/\* ===========================================================================

\* Output a short LSB first on the stream.

\* IN assertion: there is enough room in pendingBuf.

\*/

function put\_short(s, w) {

// put\_byte(s, (uch)((w) & 0xff));

// put\_byte(s, (uch)((ush)(w) >> 8));

s.pending\_buf[s.pending++] = (w) & 0xff;

s.pending\_buf[s.pending++] = (w >>> 8) & 0xff;

}

/\* ===========================================================================

\* Send a value on a given number of bits.

\* IN assertion: length <= 16 and value fits in length bits.

\*/

function send\_bits(s, value, length) {

if (s.bi\_valid > (Buf\_size - length)) {

s.bi\_buf |= (value << s.bi\_valid) & 0xffff;

put\_short(s, s.bi\_buf);

s.bi\_buf = value >> (Buf\_size - s.bi\_valid);

s.bi\_valid += length - Buf\_size;

} else {

s.bi\_buf |= (value << s.bi\_valid) & 0xffff;

s.bi\_valid += length;

}

}

function send\_code(s, c, tree) {

send\_bits(s, tree[c \* 2]/\*.Code\*/, tree[c \* 2 + 1]/\*.Len\*/);

}

/\* ===========================================================================

\* Reverse the first len bits of a code, using straightforward code (a faster

\* method would use a table)

\* IN assertion: 1 <= len <= 15

\*/

function bi\_reverse(code, len) {

var res = 0;

do {

res |= code & 1;

code >>>= 1;

res <<= 1;

} while (--len > 0);

return res >>> 1;

}

/\* ===========================================================================

\* Flush the bit buffer, keeping at most 7 bits in it.

\*/

function bi\_flush(s) {

if (s.bi\_valid === 16) {

put\_short(s, s.bi\_buf);

s.bi\_buf = 0;

s.bi\_valid = 0;

} else if (s.bi\_valid >= 8) {

s.pending\_buf[s.pending++] = s.bi\_buf & 0xff;

s.bi\_buf >>= 8;

s.bi\_valid -= 8;

}

}

/\* ===========================================================================

\* Compute the optimal bit lengths for a tree and update the total bit length

\* for the current block.

\* IN assertion: the fields freq and dad are set, heap[heap\_max] and

\* above are the tree nodes sorted by increasing frequency.

\* OUT assertions: the field len is set to the optimal bit length, the

\* array bl\_count contains the frequencies for each bit length.

\* The length opt\_len is updated; static\_len is also updated if stree is

\* not null.

\*/

function gen\_bitlen(s, desc)

// deflate\_state \*s;

// tree\_desc \*desc; /\* the tree descriptor \*/

{

var tree = desc.dyn\_tree;

var max\_code = desc.max\_code;

var stree = desc.stat\_desc.static\_tree;

var has\_stree = desc.stat\_desc.has\_stree;

var extra = desc.stat\_desc.extra\_bits;

var base = desc.stat\_desc.extra\_base;

var max\_length = desc.stat\_desc.max\_length;

var h; /\* heap index \*/

var n, m; /\* iterate over the tree elements \*/

var bits; /\* bit length \*/

var xbits; /\* extra bits \*/

var f; /\* frequency \*/

var overflow = 0; /\* number of elements with bit length too large \*/

for (bits = 0; bits <= MAX\_BITS; bits++) {

s.bl\_count[bits] = 0;

}

/\* In a first pass, compute the optimal bit lengths (which may

\* overflow in the case of the bit length tree).

\*/

tree[s.heap[s.heap\_max] \* 2 + 1]/\*.Len\*/ = 0; /\* root of the heap \*/

for (h = s.heap\_max + 1; h < HEAP\_SIZE; h++) {

n = s.heap[h];

bits = tree[tree[n \* 2 + 1]/\*.Dad\*/ \* 2 + 1]/\*.Len\*/ + 1;

if (bits > max\_length) {

bits = max\_length;

overflow++;

}

tree[n \* 2 + 1]/\*.Len\*/ = bits;

/\* We overwrite tree[n].Dad which is no longer needed \*/

if (n > max\_code) { continue; } /\* not a leaf node \*/

s.bl\_count[bits]++;

xbits = 0;

if (n >= base) {

xbits = extra[n - base];

}

f = tree[n \* 2]/\*.Freq\*/;

s.opt\_len += f \* (bits + xbits);

if (has\_stree) {

s.static\_len += f \* (stree[n \* 2 + 1]/\*.Len\*/ + xbits);

}

}

if (overflow === 0) { return; }

// Trace((stderr,"\nbit length overflow\n"));

/\* This happens for example on obj2 and pic of the Calgary corpus \*/

/\* Find the first bit length which could increase: \*/

do {

bits = max\_length - 1;

while (s.bl\_count[bits] === 0) { bits--; }

s.bl\_count[bits]--; /\* move one leaf down the tree \*/

s.bl\_count[bits + 1] += 2; /\* move one overflow item as its brother \*/

s.bl\_count[max\_length]--;

/\* The brother of the overflow item also moves one step up,

\* but this does not affect bl\_count[max\_length]

\*/

overflow -= 2;

} while (overflow > 0);

/\* Now recompute all bit lengths, scanning in increasing frequency.

\* h is still equal to HEAP\_SIZE. (It is simpler to reconstruct all

\* lengths instead of fixing only the wrong ones. This idea is taken

\* from 'ar' written by Haruhiko Okumura.)

\*/

for (bits = max\_length; bits !== 0; bits--) {

n = s.bl\_count[bits];

while (n !== 0) {

m = s.heap[--h];

if (m > max\_code) { continue; }

if (tree[m \* 2 + 1]/\*.Len\*/ !== bits) {

// Trace((stderr,"code %d bits %d->%d\n", m, tree[m].Len, bits));

s.opt\_len += (bits - tree[m \* 2 + 1]/\*.Len\*/) \* tree[m \* 2]/\*.Freq\*/;

tree[m \* 2 + 1]/\*.Len\*/ = bits;

}

n--;

}

}

}

/\* ===========================================================================

\* Generate the codes for a given tree and bit counts (which need not be

\* optimal).

\* IN assertion: the array bl\_count contains the bit length statistics for

\* the given tree and the field len is set for all tree elements.

\* OUT assertion: the field code is set for all tree elements of non

\* zero code length.

\*/

function gen\_codes(tree, max\_code, bl\_count)

// ct\_data \*tree; /\* the tree to decorate \*/

// int max\_code; /\* largest code with non zero frequency \*/

// ushf \*bl\_count; /\* number of codes at each bit length \*/

{

var next\_code = new Array(MAX\_BITS + 1); /\* next code value for each bit length \*/

var code = 0; /\* running code value \*/

var bits; /\* bit index \*/

var n; /\* code index \*/

/\* The distribution counts are first used to generate the code values

\* without bit reversal.

\*/

for (bits = 1; bits <= MAX\_BITS; bits++) {

next\_code[bits] = code = (code + bl\_count[bits - 1]) << 1;

}

/\* Check that the bit counts in bl\_count are consistent. The last code

\* must be all ones.

\*/

//Assert (code + bl\_count[MAX\_BITS]-1 == (1<<MAX\_BITS)-1,

// "inconsistent bit counts");

//Tracev((stderr,"\ngen\_codes: max\_code %d ", max\_code));

for (n = 0; n <= max\_code; n++) {

var len = tree[n \* 2 + 1]/\*.Len\*/;

if (len === 0) { continue; }

/\* Now reverse the bits \*/

tree[n \* 2]/\*.Code\*/ = bi\_reverse(next\_code[len]++, len);

//Tracecv(tree != static\_ltree, (stderr,"\nn %3d %c l %2d c %4x (%x) ",

// n, (isgraph(n) ? n : ' '), len, tree[n].Code, next\_code[len]-1));

}

}

/\* ===========================================================================

\* Initialize the various 'constant' tables.

\*/

function tr\_static\_init() {

var n; /\* iterates over tree elements \*/

var bits; /\* bit counter \*/

var length; /\* length value \*/

var code; /\* code value \*/

var dist; /\* distance index \*/

var bl\_count = new Array(MAX\_BITS + 1);

/\* number of codes at each bit length for an optimal tree \*/

// do check in \_tr\_init()

//if (static\_init\_done) return;

/\* For some embedded targets, global variables are not initialized: \*/

/\*#ifdef NO\_INIT\_GLOBAL\_POINTERS

static\_l\_desc.static\_tree = static\_ltree;

static\_l\_desc.extra\_bits = extra\_lbits;

static\_d\_desc.static\_tree = static\_dtree;

static\_d\_desc.extra\_bits = extra\_dbits;

static\_bl\_desc.extra\_bits = extra\_blbits;

#endif\*/

/\* Initialize the mapping length (0..255) -> length code (0..28) \*/

length = 0;

for (code = 0; code < LENGTH\_CODES - 1; code++) {

base\_length[code] = length;

for (n = 0; n < (1 << extra\_lbits[code]); n++) {

\_length\_code[length++] = code;

}

}

//Assert (length == 256, "tr\_static\_init: length != 256");

/\* Note that the length 255 (match length 258) can be represented

\* in two different ways: code 284 + 5 bits or code 285, so we

\* overwrite length\_code[255] to use the best encoding:

\*/

\_length\_code[length - 1] = code;

/\* Initialize the mapping dist (0..32K) -> dist code (0..29) \*/

dist = 0;

for (code = 0; code < 16; code++) {

base\_dist[code] = dist;

for (n = 0; n < (1 << extra\_dbits[code]); n++) {

\_dist\_code[dist++] = code;

}

}

//Assert (dist == 256, "tr\_static\_init: dist != 256");

dist >>= 7; /\* from now on, all distances are divided by 128 \*/

for (; code < D\_CODES; code++) {

base\_dist[code] = dist << 7;

for (n = 0; n < (1 << (extra\_dbits[code] - 7)); n++) {

\_dist\_code[256 + dist++] = code;

}

}

//Assert (dist == 256, "tr\_static\_init: 256+dist != 512");

/\* Construct the codes of the static literal tree \*/

for (bits = 0; bits <= MAX\_BITS; bits++) {

bl\_count[bits] = 0;

}

n = 0;

while (n <= 143) {

static\_ltree[n \* 2 + 1]/\*.Len\*/ = 8;

n++;

bl\_count[8]++;

}

while (n <= 255) {

static\_ltree[n \* 2 + 1]/\*.Len\*/ = 9;

n++;

bl\_count[9]++;

}

while (n <= 279) {

static\_ltree[n \* 2 + 1]/\*.Len\*/ = 7;

n++;

bl\_count[7]++;

}

while (n <= 287) {

static\_ltree[n \* 2 + 1]/\*.Len\*/ = 8;

n++;

bl\_count[8]++;

}

/\* Codes 286 and 287 do not exist, but we must include them in the

\* tree construction to get a canonical Huffman tree (longest code

\* all ones)

\*/

gen\_codes(static\_ltree, L\_CODES + 1, bl\_count);

/\* The static distance tree is trivial: \*/

for (n = 0; n < D\_CODES; n++) {

static\_dtree[n \* 2 + 1]/\*.Len\*/ = 5;

static\_dtree[n \* 2]/\*.Code\*/ = bi\_reverse(n, 5);

}

// Now data ready and we can init static trees

static\_l\_desc = new StaticTreeDesc(static\_ltree, extra\_lbits, LITERALS + 1, L\_CODES, MAX\_BITS);

static\_d\_desc = new StaticTreeDesc(static\_dtree, extra\_dbits, 0, D\_CODES, MAX\_BITS);

static\_bl\_desc = new StaticTreeDesc(new Array(0), extra\_blbits, 0, BL\_CODES, MAX\_BL\_BITS);

//static\_init\_done = true;

}

/\* ===========================================================================

\* Initialize a new block.

\*/

function init\_block(s) {

var n; /\* iterates over tree elements \*/

/\* Initialize the trees. \*/

for (n = 0; n < L\_CODES; n++) { s.dyn\_ltree[n \* 2]/\*.Freq\*/ = 0; }

for (n = 0; n < D\_CODES; n++) { s.dyn\_dtree[n \* 2]/\*.Freq\*/ = 0; }

for (n = 0; n < BL\_CODES; n++) { s.bl\_tree[n \* 2]/\*.Freq\*/ = 0; }

s.dyn\_ltree[END\_BLOCK \* 2]/\*.Freq\*/ = 1;

s.opt\_len = s.static\_len = 0;

s.last\_lit = s.matches = 0;

}

/\* ===========================================================================

\* Flush the bit buffer and align the output on a byte boundary

\*/

function bi\_windup(s)

{

if (s.bi\_valid > 8) {

put\_short(s, s.bi\_buf);

} else if (s.bi\_valid > 0) {

//put\_byte(s, (Byte)s->bi\_buf);

s.pending\_buf[s.pending++] = s.bi\_buf;

}

s.bi\_buf = 0;

s.bi\_valid = 0;

}

/\* ===========================================================================

\* Copy a stored block, storing first the length and its

\* one's complement if requested.

\*/

function copy\_block(s, buf, len, header)

//DeflateState \*s;

//charf \*buf; /\* the input data \*/

//unsigned len; /\* its length \*/

//int header; /\* true if block header must be written \*/

{

bi\_windup(s); /\* align on byte boundary \*/

if (header) {

put\_short(s, len);

put\_short(s, ~len);

}

// while (len--) {

// put\_byte(s, \*buf++);

// }

utils.arraySet(s.pending\_buf, s.window, buf, len, s.pending);

s.pending += len;

}

/\* ===========================================================================

\* Compares to subtrees, using the tree depth as tie breaker when

\* the subtrees have equal frequency. This minimizes the worst case length.

\*/

function smaller(tree, n, m, depth) {

var \_n2 = n \* 2;

var \_m2 = m \* 2;

return (tree[\_n2]/\*.Freq\*/ < tree[\_m2]/\*.Freq\*/ ||

(tree[\_n2]/\*.Freq\*/ === tree[\_m2]/\*.Freq\*/ && depth[n] <= depth[m]));

}

/\* ===========================================================================

\* Restore the heap property by moving down the tree starting at node k,

\* exchanging a node with the smallest of its two sons if necessary, stopping

\* when the heap property is re-established (each father smaller than its

\* two sons).

\*/

function pqdownheap(s, tree, k)

// deflate\_state \*s;

// ct\_data \*tree; /\* the tree to restore \*/

// int k; /\* node to move down \*/

{

var v = s.heap[k];

var j = k << 1; /\* left son of k \*/

while (j <= s.heap\_len) {

/\* Set j to the smallest of the two sons: \*/

if (j < s.heap\_len &&

smaller(tree, s.heap[j + 1], s.heap[j], s.depth)) {

j++;

}

/\* Exit if v is smaller than both sons \*/

if (smaller(tree, v, s.heap[j], s.depth)) { break; }

/\* Exchange v with the smallest son \*/

s.heap[k] = s.heap[j];

k = j;

/\* And continue down the tree, setting j to the left son of k \*/

j <<= 1;

}

s.heap[k] = v;

}

// inlined manually

// var SMALLEST = 1;

/\* ===========================================================================

\* Send the block data compressed using the given Huffman trees

\*/

function compress\_block(s, ltree, dtree)

// deflate\_state \*s;

// const ct\_data \*ltree; /\* literal tree \*/

// const ct\_data \*dtree; /\* distance tree \*/

{

var dist; /\* distance of matched string \*/

var lc; /\* match length or unmatched char (if dist == 0) \*/

var lx = 0; /\* running index in l\_buf \*/

var code; /\* the code to send \*/

var extra; /\* number of extra bits to send \*/

if (s.last\_lit !== 0) {

do {

dist = (s.pending\_buf[s.d\_buf + lx \* 2] << 8) | (s.pending\_buf[s.d\_buf + lx \* 2 + 1]);

lc = s.pending\_buf[s.l\_buf + lx];

lx++;

if (dist === 0) {

send\_code(s, lc, ltree); /\* send a literal byte \*/

//Tracecv(isgraph(lc), (stderr," '%c' ", lc));

} else {

/\* Here, lc is the match length - MIN\_MATCH \*/

code = \_length\_code[lc];

send\_code(s, code + LITERALS + 1, ltree); /\* send the length code \*/

extra = extra\_lbits[code];

if (extra !== 0) {

lc -= base\_length[code];

send\_bits(s, lc, extra); /\* send the extra length bits \*/

}

dist--; /\* dist is now the match distance - 1 \*/

code = d\_code(dist);

//Assert (code < D\_CODES, "bad d\_code");

send\_code(s, code, dtree); /\* send the distance code \*/

extra = extra\_dbits[code];

if (extra !== 0) {

dist -= base\_dist[code];

send\_bits(s, dist, extra); /\* send the extra distance bits \*/

}

} /\* literal or match pair ? \*/

/\* Check that the overlay between pending\_buf and d\_buf+l\_buf is ok: \*/

//Assert((uInt)(s->pending) < s->lit\_bufsize + 2\*lx,

// "pendingBuf overflow");

} while (lx < s.last\_lit);

}

send\_code(s, END\_BLOCK, ltree);

}

/\* ===========================================================================

\* Construct one Huffman tree and assigns the code bit strings and lengths.

\* Update the total bit length for the current block.

\* IN assertion: the field freq is set for all tree elements.

\* OUT assertions: the fields len and code are set to the optimal bit length

\* and corresponding code. The length opt\_len is updated; static\_len is

\* also updated if stree is not null. The field max\_code is set.

\*/

function build\_tree(s, desc)

// deflate\_state \*s;

// tree\_desc \*desc; /\* the tree descriptor \*/

{

var tree = desc.dyn\_tree;

var stree = desc.stat\_desc.static\_tree;

var has\_stree = desc.stat\_desc.has\_stree;

var elems = desc.stat\_desc.elems;

var n, m; /\* iterate over heap elements \*/

var max\_code = -1; /\* largest code with non zero frequency \*/

var node; /\* new node being created \*/

/\* Construct the initial heap, with least frequent element in

\* heap[SMALLEST]. The sons of heap[n] are heap[2\*n] and heap[2\*n+1].

\* heap[0] is not used.

\*/

s.heap\_len = 0;

s.heap\_max = HEAP\_SIZE;

for (n = 0; n < elems; n++) {

if (tree[n \* 2]/\*.Freq\*/ !== 0) {

s.heap[++s.heap\_len] = max\_code = n;

s.depth[n] = 0;

} else {

tree[n \* 2 + 1]/\*.Len\*/ = 0;

}

}

/\* The pkzip format requires that at least one distance code exists,

\* and that at least one bit should be sent even if there is only one

\* possible code. So to avoid special checks later on we force at least

\* two codes of non zero frequency.

\*/

while (s.heap\_len < 2) {

node = s.heap[++s.heap\_len] = (max\_code < 2 ? ++max\_code : 0);

tree[node \* 2]/\*.Freq\*/ = 1;

s.depth[node] = 0;

s.opt\_len--;

if (has\_stree) {

s.static\_len -= stree[node \* 2 + 1]/\*.Len\*/;

}

/\* node is 0 or 1 so it does not have extra bits \*/

}

desc.max\_code = max\_code;

/\* The elements heap[heap\_len/2+1 .. heap\_len] are leaves of the tree,

\* establish sub-heaps of increasing lengths:

\*/

for (n = (s.heap\_len >> 1/\*int /2\*/); n >= 1; n--) { pqdownheap(s, tree, n); }

/\* Construct the Huffman tree by repeatedly combining the least two

\* frequent nodes.

\*/

node = elems; /\* next internal node of the tree \*/

do {

//pqremove(s, tree, n); /\* n = node of least frequency \*/

/\*\*\* pqremove \*\*\*/

n = s.heap[1/\*SMALLEST\*/];

s.heap[1/\*SMALLEST\*/] = s.heap[s.heap\_len--];

pqdownheap(s, tree, 1/\*SMALLEST\*/);

/\*\*\*/

m = s.heap[1/\*SMALLEST\*/]; /\* m = node of next least frequency \*/

s.heap[--s.heap\_max] = n; /\* keep the nodes sorted by frequency \*/

s.heap[--s.heap\_max] = m;

/\* Create a new node father of n and m \*/

tree[node \* 2]/\*.Freq\*/ = tree[n \* 2]/\*.Freq\*/ + tree[m \* 2]/\*.Freq\*/;

s.depth[node] = (s.depth[n] >= s.depth[m] ? s.depth[n] : s.depth[m]) + 1;

tree[n \* 2 + 1]/\*.Dad\*/ = tree[m \* 2 + 1]/\*.Dad\*/ = node;

/\* and insert the new node in the heap \*/

s.heap[1/\*SMALLEST\*/] = node++;

pqdownheap(s, tree, 1/\*SMALLEST\*/);

} while (s.heap\_len >= 2);

s.heap[--s.heap\_max] = s.heap[1/\*SMALLEST\*/];

/\* At this point, the fields freq and dad are set. We can now

\* generate the bit lengths.

\*/

gen\_bitlen(s, desc);

/\* The field len is now set, we can generate the bit codes \*/

gen\_codes(tree, max\_code, s.bl\_count);

}

/\* ===========================================================================

\* Scan a literal or distance tree to determine the frequencies of the codes

\* in the bit length tree.

\*/

function scan\_tree(s, tree, max\_code)

// deflate\_state \*s;

// ct\_data \*tree; /\* the tree to be scanned \*/

// int max\_code; /\* and its largest code of non zero frequency \*/

{

var n; /\* iterates over all tree elements \*/

var prevlen = -1; /\* last emitted length \*/

var curlen; /\* length of current code \*/

var nextlen = tree[0 \* 2 + 1]/\*.Len\*/; /\* length of next code \*/

var count = 0; /\* repeat count of the current code \*/

var max\_count = 7; /\* max repeat count \*/

var min\_count = 4; /\* min repeat count \*/

if (nextlen === 0) {

max\_count = 138;

min\_count = 3;

}

tree[(max\_code + 1) \* 2 + 1]/\*.Len\*/ = 0xffff; /\* guard \*/

for (n = 0; n <= max\_code; n++) {

curlen = nextlen;

nextlen = tree[(n + 1) \* 2 + 1]/\*.Len\*/;

if (++count < max\_count && curlen === nextlen) {

continue;

} else if (count < min\_count) {

s.bl\_tree[curlen \* 2]/\*.Freq\*/ += count;

} else if (curlen !== 0) {

if (curlen !== prevlen) { s.bl\_tree[curlen \* 2]/\*.Freq\*/++; }

s.bl\_tree[REP\_3\_6 \* 2]/\*.Freq\*/++;

} else if (count <= 10) {

s.bl\_tree[REPZ\_3\_10 \* 2]/\*.Freq\*/++;

} else {

s.bl\_tree[REPZ\_11\_138 \* 2]/\*.Freq\*/++;

}

count = 0;

prevlen = curlen;

if (nextlen === 0) {

max\_count = 138;

min\_count = 3;

} else if (curlen === nextlen) {

max\_count = 6;

min\_count = 3;

} else {

max\_count = 7;

min\_count = 4;

}

}

}

/\* ===========================================================================

\* Send a literal or distance tree in compressed form, using the codes in

\* bl\_tree.

\*/

function send\_tree(s, tree, max\_code)

// deflate\_state \*s;

// ct\_data \*tree; /\* the tree to be scanned \*/

// int max\_code; /\* and its largest code of non zero frequency \*/

{

var n; /\* iterates over all tree elements \*/

var prevlen = -1; /\* last emitted length \*/

var curlen; /\* length of current code \*/

var nextlen = tree[0 \* 2 + 1]/\*.Len\*/; /\* length of next code \*/

var count = 0; /\* repeat count of the current code \*/

var max\_count = 7; /\* max repeat count \*/

var min\_count = 4; /\* min repeat count \*/

/\* tree[max\_code+1].Len = -1; \*/ /\* guard already set \*/

if (nextlen === 0) {

max\_count = 138;

min\_count = 3;

}

for (n = 0; n <= max\_code; n++) {

curlen = nextlen;

nextlen = tree[(n + 1) \* 2 + 1]/\*.Len\*/;

if (++count < max\_count && curlen === nextlen) {

continue;

} else if (count < min\_count) {

do { send\_code(s, curlen, s.bl\_tree); } while (--count !== 0);

} else if (curlen !== 0) {

if (curlen !== prevlen) {

send\_code(s, curlen, s.bl\_tree);

count--;

}

//Assert(count >= 3 && count <= 6, " 3\_6?");

send\_code(s, REP\_3\_6, s.bl\_tree);

send\_bits(s, count - 3, 2);

} else if (count <= 10) {

send\_code(s, REPZ\_3\_10, s.bl\_tree);

send\_bits(s, count - 3, 3);

} else {

send\_code(s, REPZ\_11\_138, s.bl\_tree);

send\_bits(s, count - 11, 7);

}

count = 0;

prevlen = curlen;

if (nextlen === 0) {

max\_count = 138;

min\_count = 3;

} else if (curlen === nextlen) {

max\_count = 6;

min\_count = 3;

} else {

max\_count = 7;

min\_count = 4;

}

}

}

/\* ===========================================================================

\* Construct the Huffman tree for the bit lengths and return the index in

\* bl\_order of the last bit length code to send.

\*/

function build\_bl\_tree(s) {

var max\_blindex; /\* index of last bit length code of non zero freq \*/

/\* Determine the bit length frequencies for literal and distance trees \*/

scan\_tree(s, s.dyn\_ltree, s.l\_desc.max\_code);

scan\_tree(s, s.dyn\_dtree, s.d\_desc.max\_code);

/\* Build the bit length tree: \*/

build\_tree(s, s.bl\_desc);

/\* opt\_len now includes the length of the tree representations, except

\* the lengths of the bit lengths codes and the 5+5+4 bits for the counts.

\*/

/\* Determine the number of bit length codes to send. The pkzip format

\* requires that at least 4 bit length codes be sent. (appnote.txt says

\* 3 but the actual value used is 4.)

\*/

for (max\_blindex = BL\_CODES - 1; max\_blindex >= 3; max\_blindex--) {

if (s.bl\_tree[bl\_order[max\_blindex] \* 2 + 1]/\*.Len\*/ !== 0) {

break;

}

}

/\* Update opt\_len to include the bit length tree and counts \*/

s.opt\_len += 3 \* (max\_blindex + 1) + 5 + 5 + 4;

//Tracev((stderr, "\ndyn trees: dyn %ld, stat %ld",

// s->opt\_len, s->static\_len));

return max\_blindex;

}

/\* ===========================================================================

\* Send the header for a block using dynamic Huffman trees: the counts, the

\* lengths of the bit length codes, the literal tree and the distance tree.

\* IN assertion: lcodes >= 257, dcodes >= 1, blcodes >= 4.

\*/

function send\_all\_trees(s, lcodes, dcodes, blcodes)

// deflate\_state \*s;

// int lcodes, dcodes, blcodes; /\* number of codes for each tree \*/

{

var rank; /\* index in bl\_order \*/

//Assert (lcodes >= 257 && dcodes >= 1 && blcodes >= 4, "not enough codes");

//Assert (lcodes <= L\_CODES && dcodes <= D\_CODES && blcodes <= BL\_CODES,

// "too many codes");

//Tracev((stderr, "\nbl counts: "));

send\_bits(s, lcodes - 257, 5); /\* not +255 as stated in appnote.txt \*/

send\_bits(s, dcodes - 1, 5);

send\_bits(s, blcodes - 4, 4); /\* not -3 as stated in appnote.txt \*/

for (rank = 0; rank < blcodes; rank++) {

//Tracev((stderr, "\nbl code %2d ", bl\_order[rank]));

send\_bits(s, s.bl\_tree[bl\_order[rank] \* 2 + 1]/\*.Len\*/, 3);

}

//Tracev((stderr, "\nbl tree: sent %ld", s->bits\_sent));

send\_tree(s, s.dyn\_ltree, lcodes - 1); /\* literal tree \*/

//Tracev((stderr, "\nlit tree: sent %ld", s->bits\_sent));

send\_tree(s, s.dyn\_dtree, dcodes - 1); /\* distance tree \*/

//Tracev((stderr, "\ndist tree: sent %ld", s->bits\_sent));

}

/\* ===========================================================================

\* Check if the data type is TEXT or BINARY, using the following algorithm:

\* - TEXT if the two conditions below are satisfied:

\* a) There are no non-portable control characters belonging to the

\* "black list" (0..6, 14..25, 28..31).

\* b) There is at least one printable character belonging to the

\* "white list" (9 {TAB}, 10 {LF}, 13 {CR}, 32..255).

\* - BINARY otherwise.

\* - The following partially-portable control characters form a

\* "gray list" that is ignored in this detection algorithm:

\* (7 {BEL}, 8 {BS}, 11 {VT}, 12 {FF}, 26 {SUB}, 27 {ESC}).

\* IN assertion: the fields Freq of dyn\_ltree are set.

\*/

function detect\_data\_type(s) {

/\* black\_mask is the bit mask of black-listed bytes

\* set bits 0..6, 14..25, and 28..31

\* 0xf3ffc07f = binary 11110011111111111100000001111111

\*/

var black\_mask = 0xf3ffc07f;

var n;

/\* Check for non-textual ("black-listed") bytes. \*/

for (n = 0; n <= 31; n++, black\_mask >>>= 1) {

if ((black\_mask & 1) && (s.dyn\_ltree[n \* 2]/\*.Freq\*/ !== 0)) {

return Z\_BINARY;

}

}

/\* Check for textual ("white-listed") bytes. \*/

if (s.dyn\_ltree[9 \* 2]/\*.Freq\*/ !== 0 || s.dyn\_ltree[10 \* 2]/\*.Freq\*/ !== 0 ||

s.dyn\_ltree[13 \* 2]/\*.Freq\*/ !== 0) {

return Z\_TEXT;

}

for (n = 32; n < LITERALS; n++) {

if (s.dyn\_ltree[n \* 2]/\*.Freq\*/ !== 0) {

return Z\_TEXT;

}

}

/\* There are no "black-listed" or "white-listed" bytes:

\* this stream either is empty or has tolerated ("gray-listed") bytes only.

\*/

return Z\_BINARY;

}

var static\_init\_done = false;

/\* ===========================================================================

\* Initialize the tree data structures for a new zlib stream.

\*/

function \_tr\_init(s)

{

if (!static\_init\_done) {

tr\_static\_init();

static\_init\_done = true;

}

s.l\_desc = new TreeDesc(s.dyn\_ltree, static\_l\_desc);

s.d\_desc = new TreeDesc(s.dyn\_dtree, static\_d\_desc);

s.bl\_desc = new TreeDesc(s.bl\_tree, static\_bl\_desc);

s.bi\_buf = 0;

s.bi\_valid = 0;

/\* Initialize the first block of the first file: \*/

init\_block(s);

}

/\* ===========================================================================

\* Send a stored block

\*/

function \_tr\_stored\_block(s, buf, stored\_len, last)

//DeflateState \*s;

//charf \*buf; /\* input block \*/

//ulg stored\_len; /\* length of input block \*/

//int last; /\* one if this is the last block for a file \*/

{

send\_bits(s, (STORED\_BLOCK << 1) + (last ? 1 : 0), 3); /\* send block type \*/

copy\_block(s, buf, stored\_len, true); /\* with header \*/

}

/\* ===========================================================================

\* Send one empty static block to give enough lookahead for inflate.

\* This takes 10 bits, of which 7 may remain in the bit buffer.

\*/

function \_tr\_align(s) {

send\_bits(s, STATIC\_TREES << 1, 3);

send\_code(s, END\_BLOCK, static\_ltree);

bi\_flush(s);

}

/\* ===========================================================================

\* Determine the best encoding for the current block: dynamic trees, static

\* trees or store, and output the encoded block to the zip file.

\*/

function \_tr\_flush\_block(s, buf, stored\_len, last)

//DeflateState \*s;

//charf \*buf; /\* input block, or NULL if too old \*/

//ulg stored\_len; /\* length of input block \*/

//int last; /\* one if this is the last block for a file \*/

{

var opt\_lenb, static\_lenb; /\* opt\_len and static\_len in bytes \*/

var max\_blindex = 0; /\* index of last bit length code of non zero freq \*/

/\* Build the Huffman trees unless a stored block is forced \*/

if (s.level > 0) {

/\* Check if the file is binary or text \*/

if (s.strm.data\_type === Z\_UNKNOWN) {

s.strm.data\_type = detect\_data\_type(s);

}

/\* Construct the literal and distance trees \*/

build\_tree(s, s.l\_desc);

// Tracev((stderr, "\nlit data: dyn %ld, stat %ld", s->opt\_len,

// s->static\_len));

build\_tree(s, s.d\_desc);

// Tracev((stderr, "\ndist data: dyn %ld, stat %ld", s->opt\_len,

// s->static\_len));

/\* At this point, opt\_len and static\_len are the total bit lengths of

\* the compressed block data, excluding the tree representations.

\*/

/\* Build the bit length tree for the above two trees, and get the index

\* in bl\_order of the last bit length code to send.

\*/

max\_blindex = build\_bl\_tree(s);

/\* Determine the best encoding. Compute the block lengths in bytes. \*/

opt\_lenb = (s.opt\_len + 3 + 7) >>> 3;

static\_lenb = (s.static\_len + 3 + 7) >>> 3;

// Tracev((stderr, "\nopt %lu(%lu) stat %lu(%lu) stored %lu lit %u ",

// opt\_lenb, s->opt\_len, static\_lenb, s->static\_len, stored\_len,

// s->last\_lit));

if (static\_lenb <= opt\_lenb) { opt\_lenb = static\_lenb; }

} else {

// Assert(buf != (char\*)0, "lost buf");

opt\_lenb = static\_lenb = stored\_len + 5; /\* force a stored block \*/

}

if ((stored\_len + 4 <= opt\_lenb) && (buf !== -1)) {

/\* 4: two words for the lengths \*/

/\* The test buf != NULL is only necessary if LIT\_BUFSIZE > WSIZE.

\* Otherwise we can't have processed more than WSIZE input bytes since

\* the last block flush, because compression would have been

\* successful. If LIT\_BUFSIZE <= WSIZE, it is never too late to

\* transform a block into a stored block.

\*/

\_tr\_stored\_block(s, buf, stored\_len, last);

} else if (s.strategy === Z\_FIXED || static\_lenb === opt\_lenb) {

send\_bits(s, (STATIC\_TREES << 1) + (last ? 1 : 0), 3);

compress\_block(s, static\_ltree, static\_dtree);

} else {

send\_bits(s, (DYN\_TREES << 1) + (last ? 1 : 0), 3);

send\_all\_trees(s, s.l\_desc.max\_code + 1, s.d\_desc.max\_code + 1, max\_blindex + 1);

compress\_block(s, s.dyn\_ltree, s.dyn\_dtree);

}

// Assert (s->compressed\_len == s->bits\_sent, "bad compressed size");

/\* The above check is made mod 2^32, for files larger than 512 MB

\* and uLong implemented on 32 bits.

\*/

init\_block(s);

if (last) {

bi\_windup(s);

}

// Tracev((stderr,"\ncomprlen %lu(%lu) ", s->compressed\_len>>3,

// s->compressed\_len-7\*last));

}

/\* ===========================================================================

\* Save the match info and tally the frequency counts. Return true if

\* the current block must be flushed.

\*/

function \_tr\_tally(s, dist, lc)

// deflate\_state \*s;

// unsigned dist; /\* distance of matched string \*/

// unsigned lc; /\* match length-MIN\_MATCH or unmatched char (if dist==0) \*/

{

//var out\_length, in\_length, dcode;

s.pending\_buf[s.d\_buf + s.last\_lit \* 2] = (dist >>> 8) & 0xff;

s.pending\_buf[s.d\_buf + s.last\_lit \* 2 + 1] = dist & 0xff;

s.pending\_buf[s.l\_buf + s.last\_lit] = lc & 0xff;

s.last\_lit++;

if (dist === 0) {

/\* lc is the unmatched char \*/

s.dyn\_ltree[lc \* 2]/\*.Freq\*/++;

} else {

s.matches++;

/\* Here, lc is the match length - MIN\_MATCH \*/

dist--; /\* dist = match distance - 1 \*/

//Assert((ush)dist < (ush)MAX\_DIST(s) &&

// (ush)lc <= (ush)(MAX\_MATCH-MIN\_MATCH) &&

// (ush)d\_code(dist) < (ush)D\_CODES, "\_tr\_tally: bad match");

s.dyn\_ltree[(\_length\_code[lc] + LITERALS + 1) \* 2]/\*.Freq\*/++;

s.dyn\_dtree[d\_code(dist) \* 2]/\*.Freq\*/++;

}

// (!) This block is disabled in zlib defailts,

// don't enable it for binary compatibility

//#ifdef TRUNCATE\_BLOCK

// /\* Try to guess if it is profitable to stop the current block here \*/

// if ((s.last\_lit & 0x1fff) === 0 && s.level > 2) {

// /\* Compute an upper bound for the compressed length \*/

// out\_length = s.last\_lit\*8;

// in\_length = s.strstart - s.block\_start;

//

// for (dcode = 0; dcode < D\_CODES; dcode++) {

// out\_length += s.dyn\_dtree[dcode\*2]/\*.Freq\*/ \* (5 + extra\_dbits[dcode]);

// }

// out\_length >>>= 3;

// //Tracev((stderr,"\nlast\_lit %u, in %ld, out ~%ld(%ld%%) ",

// // s->last\_lit, in\_length, out\_length,

// // 100L - out\_length\*100L/in\_length));

// if (s.matches < (s.last\_lit>>1)/\*int /2\*/ && out\_length < (in\_length>>1)/\*int /2\*/) {

// return true;

// }

// }

//#endif

return (s.last\_lit === s.lit\_bufsize - 1);

/\* We avoid equality with lit\_bufsize because of wraparound at 64K

\* on 16 bit machines and because stored blocks are restricted to

\* 64K-1 bytes.

\*/

}

exports.\_tr\_init = \_tr\_init;

exports.\_tr\_stored\_block = \_tr\_stored\_block;

exports.\_tr\_flush\_block = \_tr\_flush\_block;

exports.\_tr\_tally = \_tr\_tally;

exports.\_tr\_align = \_tr\_align;

},{"../utils/common":41}],53:[function(require,module,exports){

'use strict';

// (C) 1995-2013 Jean-loup Gailly and Mark Adler

// (C) 2014-2017 Vitaly Puzrin and Andrey Tupitsin

//

// This software is provided 'as-is', without any express or implied

// warranty. In no event will the authors be held liable for any damages

// arising from the use of this software.

//

// Permission is granted to anyone to use this software for any purpose,

// including commercial applications, and to alter it and redistribute it

// freely, subject to the following restrictions:

//

// 1. The origin of this software must not be misrepresented; you must not

// claim that you wrote the original software. If you use this software

// in a product, an acknowledgment in the product documentation would be

// appreciated but is not required.

// 2. Altered source versions must be plainly marked as such, and must not be

// misrepresented as being the original software.

// 3. This notice may not be removed or altered from any source distribution.

function ZStream() {

/\* next input byte \*/

this.input = null; // JS specific, because we have no pointers

this.next\_in = 0;

/\* number of bytes available at input \*/

this.avail\_in = 0;

/\* total number of input bytes read so far \*/

this.total\_in = 0;

/\* next output byte should be put there \*/

this.output = null; // JS specific, because we have no pointers

this.next\_out = 0;

/\* remaining free space at output \*/

this.avail\_out = 0;

/\* total number of bytes output so far \*/

this.total\_out = 0;

/\* last error message, NULL if no error \*/

this.msg = ''/\*Z\_NULL\*/;

/\* not visible by applications \*/

this.state = null;

/\* best guess about the data type: binary or text \*/

this.data\_type = 2/\*Z\_UNKNOWN\*/;

/\* adler32 value of the uncompressed data \*/

this.adler = 0;

}

module.exports = ZStream;

},{}],54:[function(require,module,exports){

'use strict';

module.exports = typeof setImmediate === 'function' ? setImmediate :

function setImmediate() {

var args = [].slice.apply(arguments);

args.splice(1, 0, 0);

setTimeout.apply(null, args);

};

},{}]},{},[10])(10)

});