Denobook02

by deno-ja

Denobook 02 English ver.

Author: kt3k, syumai, keroxp, sasurau4, Leko, hashrock

2019-09-22edition

Table of Contents

Chapter 1

Getting Started in Deno Programming

Author: @kt3k (Yoshiya Hinosawa)
Translator: @kt3k and @sasurau4

1.1 Introduction

This chapter explains for beginners of proramming how to start using Deno.

1.1.1 What's Deno?

Deno is a JavaScript engine. You can create various programs using JavaScript language and Deno's functions. The development of Deno has started in 2010s by Ryan Dahl, an American software engineer. The name "Deno" is an anagram of "Node". Node is another JavaScript engine which Deno was inspired by. Deno is pronounced similar to Dino (Dinosaur), and therefore its mascot character is the dinosaur.

1.1.2 What you can do with Deno

Because Deno is a general purpose JavaScript engine, you can program anything on the computer. For example, you can process files or download data from the Internet very easily using Deno. Deno is also good for running servers. If you use 'http' or 'ws' modules from Deno's standard library, you can craete the backend of web services very quickly and easily with small amount of source code.

1.1.3 Special features of Deno

The point of Deno is that it's a handy programming environment, but it also has some very special features.

TypeScript

Deno executes TypeScript as well as JavaScript. TypeScript is a very similar language to JavaScript, but it also handles the information about "types" of variables and functions. Because it checks such type information before executing the program and correct them if it finds mistakes, it reduces the bugs of software very much.

When you use TypeScript, you usually need many software installed on your computer such as node.js, typescript compiler, ts-node, ts-loader, etc. Deno includes all what you need to run TypeScript. So you can execute your TypeScript program without worrying about detailed setups.

Security System

Deno has an original security control layer. Deno executes the programs under the controlled persmissions which are given as the command line arguments. For example, it's appropriate to give only Read permission to the programs like Linter, which shouldn't need permissions other than reading of the files. If a linter needs to access the Internet, it probably tries to do something wrong or bad. In this case, you can run the program like:

```
deno --allow-read linter.js
```

By the command above, the program `linter.js` can only read the file, and it can't write files and can't access to the Internet.

If you want to allow the program to only access the network, the command is like:

```
deno --allow-net network.js
```

This permission system also has the whitelist feature. You can give more granular permissions to the programs using the whitelist.

```
deno --allow-net=https://example.com network.js
```

In the example above, the program `network.js` can only access to the pages under `https://example.com` and not other pages.

Deno has these security controlling features, and this prevents the attacks by the compromised 3rd party code.

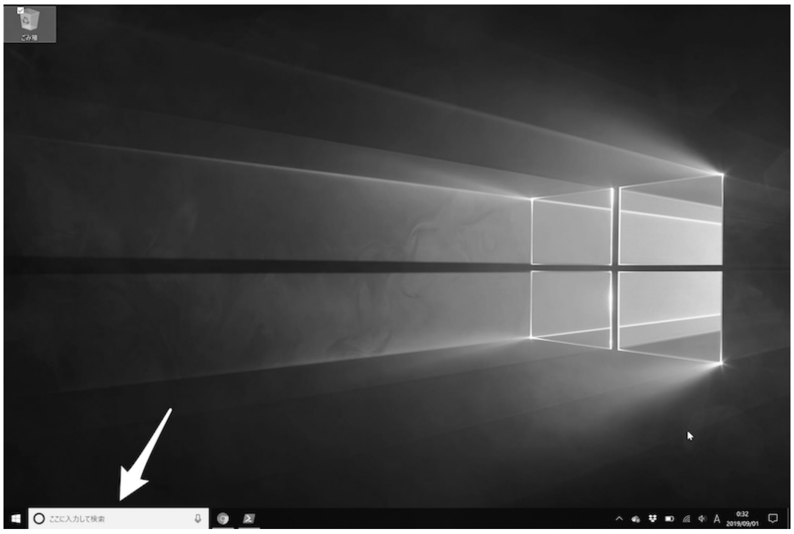
These days, more and more attacking code have been found in public module registries such as npm, Rubygems, etc. The compromised `eslint-scope` program stole authentication tokens of developers. The compromised `event-stream` program tried to stole cryptocurrencies of users. In these cases, typically attackers try to send the theft data to their own servers. If you set appropriate network permissions in these cases using Deno's feature, then these kind of attacks will never be successful.

1.2 Let’s set up Deno

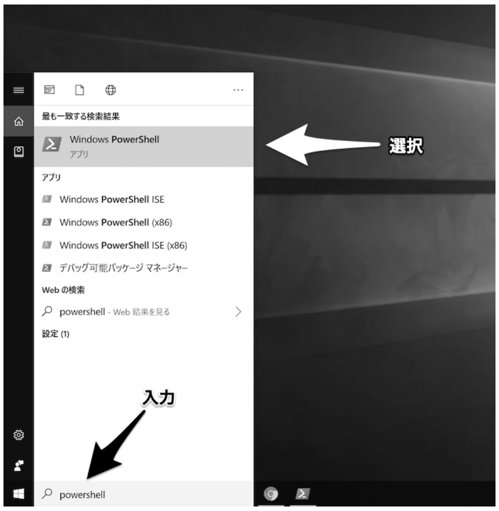
Then, let’s set up Deno on your computer!

1.2.1 Install Deno on windows

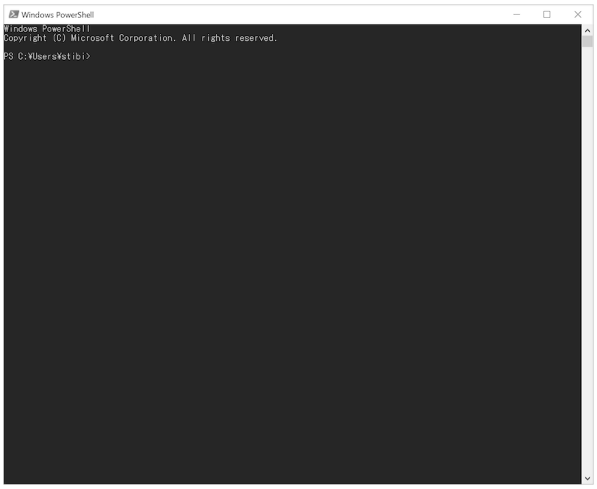
This chapter explains how to set up Deno on windows. (If you use Mac, then skip to 1.2.2.) First, find the search box at the left of the taskbar and input "powershell" in it.



When you input "powershell" in the search box, you’ll see powershell button appearing above. Then click it.



You’ll see the window of Powershell like the below.



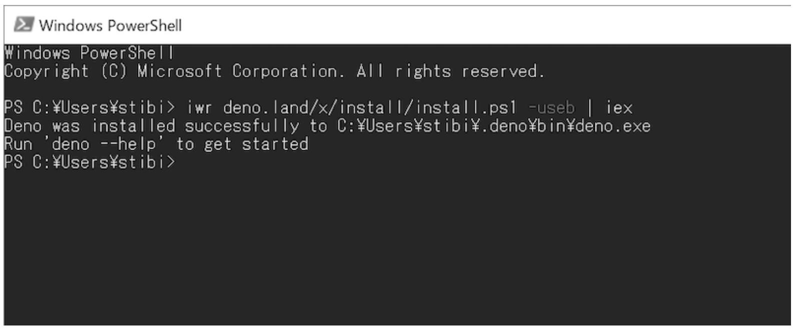
```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:¥Users¥stibi>
```

In Powershell, input the command below and download Deno.

```
iwr deno.land/x/install/install.ps1 -useb | iex
```

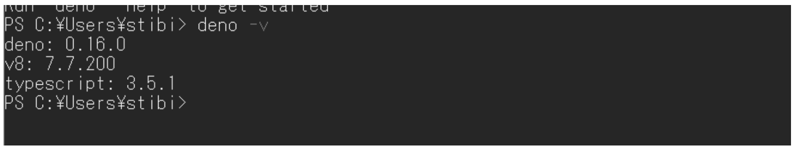
If you see the result like the below, then the download was successful.



```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:¥Users¥stibi> iwr deno.land/x/install/install.ps1 -useb | iex
Deno was installed successfully to C:¥Users¥stibi¥.deno¥bin¥deno.exe
Run 'deno --help' to get started
PS C:¥Users¥stibi>
```

After the download finished, input `deno -v`. You'll see the output like the below, and that's all of the installation!



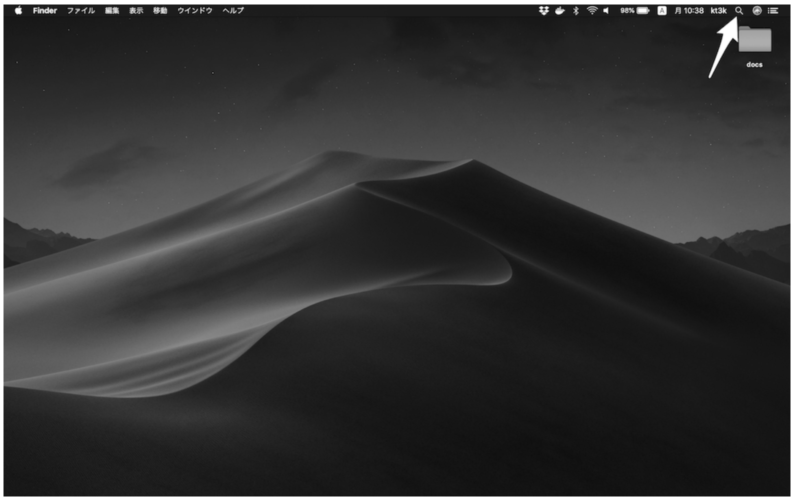
```
PS C:¥Users¥stibi> deno -v
deno: 0.16.0
v8: 7.7.200
typescript: 3.5.1
PS C:¥Users¥stibi>
```

Column: Install Deno with scoop

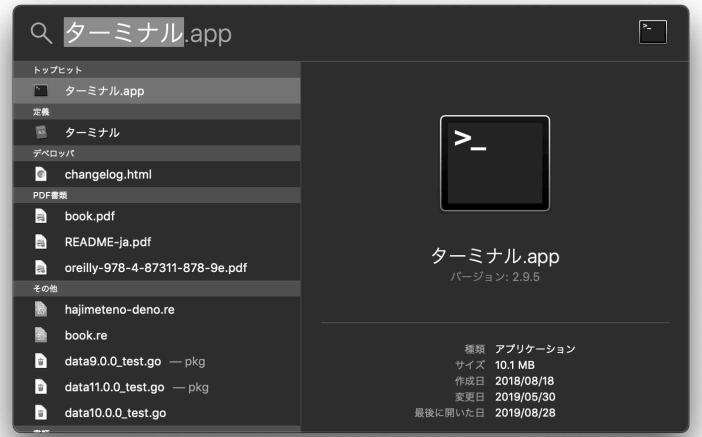
If you have `scoop` installed on your computer, you can install Deno with the command below.

```
scoop install deno
```

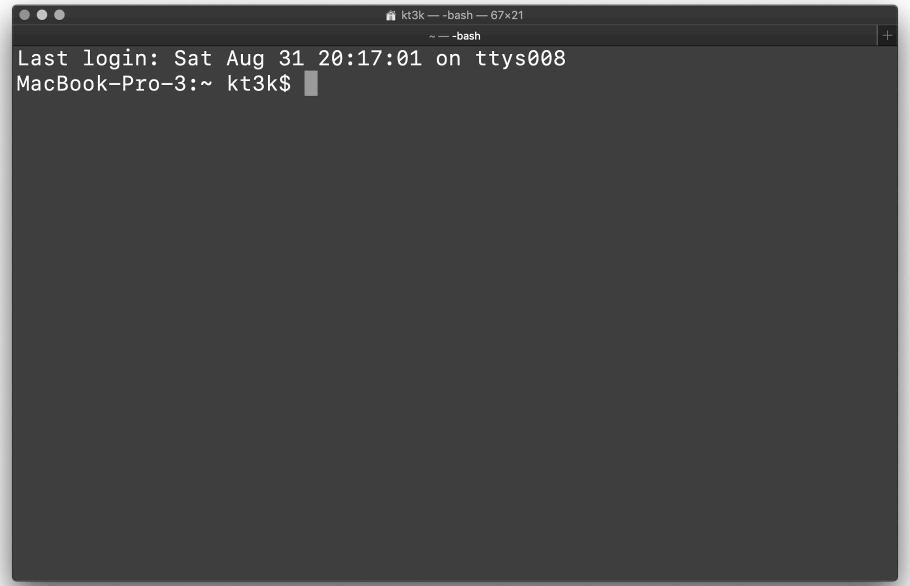
1.2.2 Install Deno on Mac



First, click the magnifier button at the top right corner of the screen, and open Spotlight search.



Input "terminal" in Spotlight search box, and click the Terminal.app from the search results.



```
Last login: Sat Aug 31 20:17:01 on ttys008
MacBook-Pro-3:~ kt3k$
```

You see the terminal window like the above. Then input the command below, and download Deno.

```
curl -L deno.land/x/install/install.sh | sh
```

After finishing the download, append the following line to `~/.bash_profile`.

```
export PATH=$HOME/.deno/bin:$PATH
```

And then, hit the command below to reload `~/.bash_profile`.

```
. ~/.bash_profile
```

That's the end of the installation! Then let's check it. Input `deno -v` in the terminal, and you'll see the version information of Deno like the below.

```
$ deno -v
deno: 0.16.0
v8: 7.7.200
typescript: 3.5.1
```

That's all!

Column: Install Deno with Homebrew

If you have Homebrew installed on your computer, you can install Deno with `brew` command. Input the command below in the terminal.

```
brew install deno
```

If you use this command to install Deno, you don't need to edit `~/.bash_profile`.

1.2.3 How to execute a Deno program

There are 2 ways to execute Deno programs. The one is to input the program from the interactive shell and execute it directly from there. The other one is to create a file of Deno program and execute it from the command line.

Execute a Deno program from the interactive shell

Let's execute the Deno program from the interactive shell! The interactive shell is a shell environment where you can input programs and execute them interactively from there. In this environment, the programs are directly evaluated by Deno and you can see the result instantly. This kind of environment is also called REPL, which stands for Read-Eval-Print Loop.

First, open the terminal (powershell) as explained in sections 1.2.1 (windows) or 1.2.2 (mac). Then input `deno` and press the Enter key.

```
deno ⏎
>
```

Then you'll see the symbol `>` as above. If you don't see this output, something's wrong about your installation. In that case, please go back to the previous sections and let's check the installation steps again.

When you see `>` symbol, then this is the interactive shell environment. You can input Deno program here. First, let's try to input the below program.

```
> console.log("hello world")
```

(console.log is the command for showing the text)

After finishing the input, press the Enter key, and then you'll see `hello world` output a line below.

```
> console.log("hello world") ⏎
hello world
```

If you see this output, it means you successfully executed the Deno program from the interactive shell.

Execute a Deno program file

Next, let’s execute the Deno program from a file. Open a file, write the same thing as above, and save it.

```
console.log("hello world")
```

After having saved it, open the terminal again, input `deno`, input a space, and then drag and drop the saved file into the terminal. Then you’ll see the path of the file in the terminal like the below.

```
deno /Users/kt3k/Desktop/hello.js
```

(The actual path varies depends on where you saved the file.)

Then press the Enter key and execute the file.

```
deno /Users/kt3k/Desktop/hello.js ⏎
hello world
```

If you see the output like the above, then the execution was successful.

1.3 Let’s start programming in Deno

In the last section, we learned how to run the Deno programs. From this section, let’s have a look at the basics of programming in Deno.

1.3.1 What’s Programming?

First of all, what’s "Programming"? In a daily situations, we usually use English to ask something to others. On the other hand, if you use English to ask computers to do something, the computers don’t understand you. You need to use something that the computers understand. The programming languages are such languages which the computers are able to understand.

The relationships between human languages and programming languages are as the following.

to Humans	to Computers
Languages	Programming Languages
To Write Sentences	To Program
Sentences	Programs

There are many kinds of programming languages such as JavaScript, Python, Go, etc, as there are many human languages such as English, Spanish, Chinese, etc.

1.3.2 Let’s calculate in Deno

All right, then let’s learn how to calculate numbers in Deno. Open the terminal, input `deno`, and press the Enter key to start the interative shell.

```
deno ⏎
>
```

You should see the symbol `>`. This is the interactive shell.

Addition, Subtraction

You can perform the addition by `+`, and the subtraction by `-`.

First, let’s calculate 1+1 in Deno. Input 1+1, and press the Enter key.

```
> 1+1 ⏎
2
```

Next, let’s do other calculations.

```
> 3+5
8
> 12+56
68
> 5-2
3
> 56-30
26
> 1729-3000
-1271
```

Multiplication, Division

You can use `*` for mutiplication, and `/` for division. Input the following to the interactive shell and check the results.

```
> 34 * 12
408
> 100 * 1.08
108
> 56 / 7
8
> 56 / 9
6.222222222222222
```

Operator Precedence

Now you know the usages of addtion, subtraction, multiplication, and division. These 4 operations are called the basic arithmetic operations. Next let's calculate the combination of these operations.

```
> 10 + 20 * 3 - 40
30
```

As you can see, when a expression has different operations, the multiplication is calculated first, and then other operations such as addition and substraction. This rule is the same as the operator precedence in mathematics. If you want to change this order of the calculation, you can use parenthesis `()` like the below. This rule is also the same as mathematics.

```
> (10 + 20) * 3 - 40
50
```

Modulo operator

There is another operator for calculation. You can use `%` for the modulo operation, which calculates the reminder of the division.

```
> 4 % 3
1
```

You can use the modulo operator when you want to handle the periodic situations such as doing something once every two, or once every four, etc.

Exponentiation

Exponentiation is the operation for doing multiplication repeatedly. For example, 4 to the 3 is the multiplication of three 4s, i.e. 64. The symbol for exponentiation is `**`.

```
> 4**3
64
> 9**3
729
> 12**3
1728
> 2**16
65536
> 1**100
1
```

Sum up

There are the following arithmetic operators in Deno.

Symbol	Usage	Meaning
`+`	`1 + 1`	Addition
`-`	`1 - 1`	Subtraction
`*`	`1 * 1`	Multiplication
`/`	`1 / 1`	Division
`%`	`1 % 1`	Modulo Calculation
`**`	`1 ** 1`	Exponentiation

1.3.3 Use data conveniently - Variables

In this section let’s have a look at "Variables". The variables are the feature for storing the data for some period of time. They are similar to the address book of a cell phone.

You usually don't memorize all of the phone numbers of your friends. You register the phone number of the friend to your phone and when you need it, you look it up by the name of the friend and call them. This is, in other words, that you saved the **data** (phone numbers), and stick the **labels** (the names) on it.

The mechanism of variables is similar to the address book. When you have data in your program, such as numbers or string, you can stick labels on them. You can use those data by those labels, and those labels are called variables.

So far we did various calculations in Deno. Now let's try to write programs using variables. The way to use variables in the program is like the following. You need to place = symbol between the variable and the value.

```
variable = value
```

This sets the value to the variable. In the words of address book example, you can use = symbol for sticking the labels on them.

In mathematics, the symbol = is used for asserting that the left side and the right side are equal as in `2 + 2 = 4`, but in Deno the symbol is used for setting the values to variables.

Now let's write the program using variables in the interactive shell.

```
> tax = 0.08
0.08
> price = 120
120
> john_telephone = "090-0123-4567"
090-0123-4567
```

On the first line, it sets the number `0.1` to the variable `tax`. On the second line, it sets the number `120` to the variable `tax`. On the third line, it sets the string `"090-0123-4567"` to the variable `john_telephone`. (We'll revisit the `string` later.)

Next let's check whether these variables are actually set. Input `tax` and press the Enter key. Then input `price` and `john_telephone` and check those values as well.

```
> tax
0.08
> price
120
> john_telephone
090-0123-4567
```

Then it shows the numbers and strings which you set before. This means that these values are stored in the variables.

Let’s do some calculations using these variables.

```
> price * tax
9.6
> 120 * 0.08
9.6
```

This shows that the result of the calculation of the variables is the same as the result of the calculation of the values.

The characters which can be used in variable names

You can give arbitrary names to the variables, but there are some exceptions. The names of variable have the following rules.

- You can’t use numbers as the initial character.
- You can’t use the reserved words.
- You can’t use symbols except `_`.

```
> value = 100
100
> value_ = 100
100
> 1value = 200
error: Uncaught SyntaxError: Invalid or unexpected token
|> <unknown>:1:1
    at evaluate (js/repl.ts:87:34)
    at replLoop (js/repl.ts:145:13)
```

There wasn't any problem with the variable names `value` and `value_`. However `1value` caused the error `error: Uncaught SyntaxError: Invalid or unexpected token`. This means there was an syntax error in the program. The syntax means the grammar of Deno, i.e. the rules of the language. By this we can see that we can't use numbers as the first character of the variables names.

Next it has the rule that it can't be a reseverd word. What is a reserved word? A reserved word is a word which is reserved for certain usages in Deno.

What happens if you use these reserved words for the variables names? Let's try it by using the reserved words `try` and `catch`.

```
> try = 1
error: Uncaught SyntaxError: Unexpected token '='
|> <unknown>:1:5
    at evaluate (js/repl.ts:87:34)
    at replLoop (js/repl.ts:145:13)
```

```
> catch = 1
error: Uncaught SyntaxError: Unexpected token 'catch'
|> <unknown>:1:1
    at evaluate (js/repl.ts:87:34)
    at replLoop (js/repl.ts:145:13)
```

Syntax errors happened as expected. Deno has the following reserved words. If you encounter the unexpected syntax errors, check this table and confirm whether you're not trying to name variables with reserved words.

Sum up

This section explained the variables.

Reserved words in Deno

break	case	catch	continue	debugger
default	delete	do	else	finally
for	function	if	in	instanceof
new	return	switch	this	throw
try	typeof	var	void	while
with	class	const	enum	export
extends	import	super	null	true
false				

1.3.4 Which is bigger? - Comparative Operator

In this section let’s have a look at comparative operators in Deno. Comparative operators, as the name suggests, is used for comparing values. In the daily situation, people compare things in many different aspect, like "I like it better than the other", "That is heavier than it", "It’s faster to go this way", etc. In Deno, we compare the data by their numbers.

Let’s check it in the interactive shell. First input 35 > 24 and press the Enter key.

```
> 35 > 24
true
```

(The first `>` means that it’s the interactive shell. Don’t be confused with the comparative operators.)

By comparing 35 and 24 by `>`, it display the text `true`. This means that 35 > 24 is true! Then let’s try using the opposite comparative operator.

```
> 35 < 24
false
```

This time it displays `false`. This means $35 < 24$ is false, which means $35 < 24$ is not correct. In this way the program express the correct thing as `true`, and wrong thing as `false`. The following table shows all the comparative operators in Deno.

Comparative Operator	Example	Meaning
`>`	`x > y`	x is greater than y
`>=`	`x >= y`	x is greater than or equal to y
`<`	`x < y`	x is less than y
`<=`	`x <= y`	x is less than or equal to y
`==`	`x == y`	x is equal to y
`!=`	`x != y`	x is not equal to y
`===`	`x === y`	x is strictly equal to y
`!==`	`x !== y`	x is not strictly equal to y

Here you need to pay attention to == and ===. As explained in 1.3.3, = is used for setting values to the variables, and these operators can be confused with it. Be careful to use these properly.

Use = and === like the below.

```
> apple = 15
15
> apple === 15
true
```

On the first line it sets 15 to the variable `apple`. On the 2nd line it tests the equality of `apple` and 15, and it shows the result `true`.

Column: == vs ===

There are 2 similar operators == and ===. The both operators checks whether the values are equal, but these two have difference in the strictness of the comparison. == compares values **not strictly**, and === compares values **strictly**.

Then what is strict and what is not strict in this context? Strictness in this case means whether it checks the data types of the values or not. == compares 2 data without checking their types but only compares their contents. === compares 2 data in both their contents and their types.

Open the interactive shell, and let's compare the number 1 and the string '1' with the both operators.

```
> 1 == "1"
true
> 1 === "1"
false
```

As you can see, == considers the result true, and === considers the result false. This is because the number 1 and the string '1' are equal in the contents, but they have different types.

1.3.5 Data types

This section describes the various data types in Deno. There are many data types in Deno, but this section picks up the seven typical data types.

Numbers

Number is a data type for representing numbers such as 1, 2, 10, etc. You can perform various calculation on numbers as described in the section 1.3.1. Use Number type when you want to express some quantity in a program.

```
a = 12
```

The variable `a` is a Number.

String

String is a data type for handling data as a sequence of characters. Surrounding characters with single quote `'` or double quote `"` tells the program to treat them as strings. For example, if you set "hello" to the variable `message`, then it will be of type string.

```
> message = "hello"
hello
> message
hello
```

Useful methods of String

String type has various useful functions. Here are some of them.

toUpperCase() is a function that converts all characters in a string to capital letters.

```
> text = "hello"
> text.toUpperCase()
HELLO
```

The first line sets a string "hello" in lowercase letters to the variable `text`. On the second line, it writes toUpperCase() with a dot after the variable `text`. Then it show the result "HELLO" in all capital letters on the third line. On the other hand, you can use toLowerCase() to covert to lowercase letters.

There is another function called repeat() that repeats the string a specified number of times. Let's write the program.

```
> text = "hello"
> text.repeat(5)
hellohellohellohellohello
```

The first line sets a string "hello" to the variable `text`. On the second line, it writes .repeat(5) after the variable `text`. Since it entered 5 in (), it is a process of repeating it 5 times. Therefore the third line was displayed as the above.

The functions that these data types have are called methods.

Methods

Methods are generally functions that can be executed in the following format:

```
variable.method()
variable.method(parameter1)
variable.method(parameter1, parameter2)
variable.method(parameter1, parameter2, ...)
...
```

The number of parameters depends on the kind of method. The method .toUpperCase() has no paramters. The method .repeat() is an example with one parameter.

Methods that can be used are determined by the type of data.

Boolean

Boolean is a data type that represents `true` or `false`. The only two values `true` and `false` belong to this data type. As explained briefly in the previous section, a boolean value is a data that expresses whether an expression is correct or incorrect in the program.

```
> 1 < 2
true
> 1 > 2
false
```

null and undefined

Null is a data type that represents "no value". If there is no value as a result of some calculation, it can be expressed by null. The only value that has null type is the `null` value.

```
> a = null
null
> a
null
```

Undefined is a data type that represents being "not defined". It's similar to null, but used as a representation of being not defined at all. Similar to null type, undefined type has the only one value that is `undefined`.

```
> a = undefined
undefined
> a
undefined
```

Array

Array type is a little different from the previous ones. By using array type, you can group multiple data. If you actually write your own program, you may handle a lot of data, and you will feel the convenience of collecting the data. The following format is used when using an arraty type. Separate the data by comma `,` and surround the whole by the square brackets `[]`.

```
[37, "coffee", "tea"]
```

You can also set arrays to the variables just like other data types.

```
groupA = ["kazu", "goro"]
groupB = ["haru", "syun"]
```

Append to and Remove from Array

You can append items to the array or remove items from the array using the methods.

To append items, you can use .push() method. In the above example, `groupA` was the pair of kazu and goro. Let’s add the third memeber "tetsu" to this group.

```
> groupA.push("tetsu")
3
> groupA
[ "kazu", "goro", "tetsu" ]
```

In the first line, it writes .push("tetsu") after groupA. This means appending tetsu to this group. The second line displays 3. This is the number of members in the result array. On the third line, groupA is input, and the three members of the group are displayed on the fourth line. You can see that tetsu is the member here.

Next, let’s remove members from this group. Use .pop()[*1] method to do it.

```
> groupA.pop()
tetsu
> groupA
[ "kazu", "goro" ]
```

On the first line, it calls the .pop() method of groupA, and it deletes the last item of the group. On the second line, it shows tetsu. This means tetsu has been removed from the array. On the third line, it inputs groupA, and the result array is displayed

[*1] You can append the last item by .push() method, and remove the last item by .pop() method. If you want to add/remove to/from the first item, you can use .unshift() and .shift() methods respectively.

on the fourth line. You can see it has the only 2 members.

Sort Array

The array also provide the method for sorting its members. You can use .sort() method for sorting the contents of the array.

When the items in the array are strings, the method sort them alphabetically.

```
> groupA = ["kazu", "goro", "asuka"]
[ "kazu", "goro", "asuka" ]
> groupA.sort()
[ "asuka", "goro", "kazu" ]
```

If the items in the array are numbers, the method sort them in ascending order of the numbers.

```
> results = [88, 50, 55, 79, 46]
[ 88, 50, 55, 79, 46 ]
> results.sort()
[ 46, 50, 55, 79, 88 ]
```

Object

Object is a data type for handling data collectively like the array type. Array is the data type which stores the items in sequence, whereas the data in a object has no order. Instead, the object type manages its data by the labels on them. To create an object, make pairs of keys and values by placing `:` between them, separate those pairs by comma (`,`), and surround the whole by curly brackets `{}`.

```
{ "name": "John", "age": 10 }
```

If the characters in the labels don't include symbols like +, -, /, %, etc, you can omit the double quotations around the labels.

```
{ name: "John", age: 10 }
```

The pairs of the label and its value is called a **property**. The label is called the property name, and the value is called the property’s value.

You can get the property’s value of an object by using `.` operator on it as follows:

```
object.property
```

Let’s try in the interactive shell.

```
person = { name: "John", age: 10 }
```

The above sets the object `{ name: "John", age: 10 }` to the variable `person`. Let’s check if it’s set actually.

```
> person
{ name: "John", age: 10 }
```

You can see the object is acutally store in the variable `person`. Next, let’s retrieve the property’s value from it. Add a dot `.` and the property name after the variable name.

```
> person.name
John
```

You can see now the propery `name` of the object `person` is "John". Similarly, let’s check the age property.

```
> person.age
10
```

It displays 10. In this way, we were able to check the individual properties of the object.

Next, let's try changing the values of the properties. To do so, you can use the following format:

```
object.property = value
```

Let's try changing the age of the person to 11.

```
> person.age = 11
11
```

See the entire person now.

```
> person
{ name: "John", age: 11 }
```

You can see the age changed to 11. In this way, you can change the property of the object.

Sum up

Here we sum up how to define the various data types we learned so far.

Number

```
data = 1
data = 1.5
```

If you write only number, it becomes of number type. You can use the decimal point as well.

String

```
data_string = "str"
data_string = "str"
```

Texts enclosed in single or double quotes are of string type.

Boolean

```
data = true
data = false
```

True and false are of boolean type.

Array

```
data_array = ["a", "b", "c"]
data_array = [3, 15, 40]
```

Use [] to group items and make arrays.

Object

```
data_object = { name: "John", age: 30 }
data_object = { type: "animal", kind: "cat" }
```

Create pairs with : and group them together with `{}`.

1.4 The Basics of Programming

1.4.1 What do you do in this case? - If statement

In this section, let's learn how to do things conditionally. If you want to perform some process only on some situations, you can use `if statement` in the following format:

```
if (condition) {
  ... program ...
}
```

You should use boolean values for `condition` in the above format. In the curly brackets `{}`, enter the program that you want to execute only when the condition is true.

To show how this works, this section describes it by programming an practical example of the usage of if statement.

Consider a vending machine with only one product. Suppose that the product is 150 yen. Let's consider a program that determines whether the amount is sufficient to buy a product when money is put into this vending machine. The structure of the program is as follows.

```
if (the entered amount is greater than or equal to 150 yen) {
  You can buy the product
}
```

From here, we're going to write actual programs for the part written in English above. For the part `the entered amount is greater than or equal to 150 yen`, you can use the comparative operator described earlier in 1.3.4. Here suppose the amount of entered money is set in the variable `amount`. To express that you're able buy the product, let's just print `You can buy the product` by console.log function.

```
if (amount >= 150) {
  console.log("You can buy the product.")
}
```

Let's execute this program in the interactive shell. Open the console, input `deno` to start the shell, and input the following program.

```
> amount = 200
200
> if (amount >= 150) {
    console.log("You can buy the product.")
  }
You can buy the product.
undefined
```

It displayed "You can buy the product". (Undefined was displayed on the next line. This is because the entire expression above returned nothing, and in that case Deno displays `undefined`. So it doesn't have any significant meaning here, and therefore it's safe to ignore such occurence of undefined.)

Then next, let's change the amount to 100 yen.

```
> amount = 100
100
> if (amount >= 150) {
    console.log("You can buy the product.")
  }
undefined
```

In this case, nothing was displayed. (Likewise please ignore undefined at the last line.) By these executions, you can see that if amount >= 150 is true, then the programs in {} is executed and that if not, it isn't executed.

So far we've written the program of checking whether the amount of money is enough for buying the product in a vending machine. By the way, the program above didn't show any message when the amount is not enough. This is a little inconvenient because the users of the vending machine don't see any message when the money is not enough. Let's add the message for that situation.

You can use `else` keyword for executing the program only when the codition doesn't hold. You can use the following format.

```
if (condition) {
  The program when the condition is true - (A)
} else {
  The program when the condition is false - (B)
}
```

When the conditional expression is true, (A) is executed and (B) is not executed. Conversely, when the conditional expression is false, (B) is executed and (A) is not executed. By using this function, we can branch programs using conditional expressions.

In this format, an example of a vending machine is expressed as follows.

```
if (amount >= 150) {
  console.log("You can buy the product.")
} else {
  console.log("You cannot buy the product because of insufficient money to buy.")
}
```

First check whether the amount is 150 or more. If it is 150 or more, the program will end with "You can buy the product" on the second line. If it is less than 150, the program will end with the message "You cannot buy the product because of insufficient money to buy." on line 4.

Finally, let’s enter and run the following program: Set the amount to 100 for testing.

```
> amount = 100
100
> if (amount >= 150) {
    console.log("You can buy the product.")
  } else {
    console.log("You cannot buy the product because of insufficient money to buy.")
  }
You cannot buy the product because of insufficient money to buy.
undefined
```

Since amount was less than 150, only the line under else was executed. Try running this same program with an amount of 200. In that case, "You can buy the product" will be displayed.

Displays your change

So far, we have written a program that checks the input amount and branches depending on whether it has reached 150 yen. Here, let’s write a program that displays how much the change will be when the amount of input exceeds 150 yen, and displays “no change” if there is no change.

This time we use the keyword "else if". By using this "else if", conditional branching "if it was ~" can be performed multiple times.

```
if (condition 1) {
     Executed when condition 1 is true-(A)
} else if (condition 2) {
     Executes when condition 1 is false and condition 2 is true-(B)
} else {
    Executed when condition 1 and condition 2 are both false-(C)
}
```

When condition 1 is true (A) is executed. When condition 1 is false and condition 2 is true (B) is executed. (C) is executed when both condition 1 and 2 are false.

The program that displays the change using this format is as follows.

```
if (amount === 150) {
  console.log("You can buy the product. There is no change.")
} else if (amount > 150) {
  console.log("You can buy the product. " + (amount - 150) + " yen is your change.")
} else {
  console.log("You cannot buy the product because of insufficient money to buy.")
}
```

The first line determines whether amount is exactly 150. If this is the case, the message “You can buy the product. There is no change.” will be displayed.

If the amount is not exactly 150, the third line determines whether the amount exceeds 150. If it exceeds 150, the program will finish with the message “You can buy the product. ~ yen is your change.". The part of "~ yen" is (amount - 150). Now the amount is over 150, it is the amount of change.

If the above two conditions are not met, it comes to "else" on the 5th line, it displays "Insufficient money", and the program ends.

Start the interactive shell and enter the above program to try it out.

```
> amount = 180
180
> if (amount === 150) {
    console.log("You can buy the product. There is no change.")
  } else if (amount > 150) {
    console.log("You can buy the product. " + (amount - 150) + " yen is your change.")
  } else {
    console.log("You cannot buy the product because of insufficient money to buy.")
  }
You can buy the product. 30 yen is your change.
```

```
undefined
```

Did it behave as expected? Try the different values as amount and see the results.

Sum up

In this section, we learned how to execute programs conditionally using `if` and `else` statements.

1.4.2 Create a machine - Functions

There is a feature called function in programming. Functions are a feature that combine several processes into one so that they can be called later.

Functions can be created using the keyword "function". The basic format of the function is as follows:

```
function functionName() {
  processing1
  processing2
  ...
}
```

There is another way to define a function, and the way is to define it by setting the variable name to the format () => {}. This function is called the arrow function.

```
variableName = () => {
  processing1
  processing2
  ...
}
```

We don't need to distinguish arrow functions and normal functions at first. Use either one that is easy for you to write.

In the function processing, we can call another function such as console.log and use the if statement learned in the previous section.

How to use function

Let's write a program that actually works using the previous format. As an example, let's use the function described in the previous section to determine whether or not you can buy a product that price is 150 yen. The function name is checkAmount because it checks the amount.

Call this function after the function is defined. Calling is easy, just write () after the function name and execute. Let's run the following example:

```
> amount = 150
150
> function checkAmount() {
    if (amount === 150) {
      console.log("You can buy the product. There is no change.")
    } else if (amount > 150) {
      console.log("You can buy the product." + (amount - 150) + "yen is your change.")
    } else {
      console.log("You cannot buy the product because of insufficient money to buy.")
    }
  }
undefined
> checkAmount()
You can buy the product. There is no change.
undefined
```

When the function was called, it was confirmed that the amount judgment process, which is the process defined in the function, can be executed. A function is useful in that once a process is written in this way, a complex process can be executed any number of times just by calling it later.

Try changing the amount value and run checkAmount several times.

```
> amount = 100
> checkAmount()
You cannot bu the product because of insufficient money to buy.
undefined
> amount = 300
300
> checkAmount()
You can buy the product. 150 yen is your change.
undefined
```

We were able to confirm that the money amount judgment process was correctly performed by calling a function.

When you use a lot of processing like this, if you put it in a function, the number of lines in the program will be reduced, and the program will be cleaner and easier to read.

Use arguments (parameters)

In the above example, the process of determining whether the variable amount is 150 or more has been combined into a function, but this function may be inconvenient. That's when amount contains some other important data. In order to use the above function, it is necessary to put the amount to be judged in amount, so the data that has been in the amount will be overwritten and deleted.

A function has a mechanism called "argument (parameter)" that receives a variable that can be used only within the function. Here is an example of using amount in the above example as a function argument.

Try typing the following example into an interactive shell:

```
> function checkAmount(amount) {
    if (amount === 150) {
      console.log("You can buy the product. There is no change.")
    } else if (amount > 150) {
      console.log("You can buy the product." + (amount - 150) + "yen is your change.")
    } else {
      console.log("You cannot buy the product because of insufficient money to buy.")
    }
  }
undefined
```

The difference from the previous example is that the amount is inside () after checkAmount. This amount is an argument. To assign the amount to this amount variable, enter the number we want to assign in amount in parentheses when calling the function.

```
> checkAmount(80)
You cannot bu the product because of insufficient money to buy.
undefined
> checkAmount(150)
You can buy the product. There is no change.
undefined
> checkAmount(900)
You can buy the product. 750 yen is your change.
undefined
```

In this way, we can now determine the result to the amount of money we put in (). Now that the variable amount is only used inside the function, it doesn't matter how amount is used outside the function.

Function that returns data

a function has one more important functionality besides executing the process using the argument passed when it is called. It is a feature that returns data. Not only does it execute the process and display the results, it also returns (passes) the data. That being said, you may not be familiar with this concept, so I will explain it specifically in the program.

Let's create a function that calculates the area of a circle using the arguments we learned earlier. This function returns the area when the radius is passed as an argument. Recall that the formula for finding the area of a circle was radius * radius * π (approximately 3.14).

Enter the following in the interactive shell:

```
> area = (radius) => {
    const result = radius * radius * 3.14
    return result
  }
```

```
[Function: area]
```

The notation area = (radius) => {on the first line was another format (arrow function) for defining functions. Here, an arrow function that receives an argument called radius is set in a variable called area. Therefore, area is the function that we will define.

In the second line, we find the expression const result =. This sets the variable result to radius * radius * 3.14 (circle area). The expression const came out for the first time, this is an indication that this result variable is used only in the function. If const is not added, result will be a variable outside the function, but by adding const, result can be confined in the function.

The result is returned as the function result in the return result on the third line.

[Function: area] on the 4th line indicates that a function named area has been defined.

To do this, write:

```
> area(4)
50.24
```

It turns out that the area of the circle with radius 4 is 50.24. Let's change the arguments and check the area of the circle with other radii.

Abbreviated notation of {} in arrow function

There is a way to write an arrow function by omitting {} and return.

```
(parameters) => expression using parameters
```

Here, we can write only a simple expression that does not include a assignment of variables or an if statement in the "expression using parameters" section. If you write a function that returns the area of the upper circle using this format, it will be as follows.

```
> area = (radius) => radius * radius * 3.14
```

It became a lot easier. In the case of a function that can be easily calculated from parameters using one expression, as in this example, {} can be omitted and written easily.

Sum up

In this section, we learned about functions that organize program processing.

1.4.3 Iteration - for, while

Let's learn about "repetition", one of the basic concepts in programming.

If you want to repeat a certain number of times in Deno, use the for keyword to express it in the following format.

```
for (variableName of Array(number you want to repeat)) {
  processing you want to repeat
}
```

First, let's write a simple example program that displays hello and repeats 10 times.

```
> for (i of Array(10)) {
    console.log("hello")
  }
hello
hello
hello
hello
hello
```

```
hello
hello
hello
hello
hello
undefined
```

It was displayed 10 times with hello. We can also try changing the number after Array to 20 or 100.

If you want to repeat along existing array data instead of repeating a specific number of times, you can use the following format.

```
for (variableName of array) {
  processing you want to repeat
}
```

The variable written after for at this time is called a loop variable, and the data in the array is entered one by one in this variable. As an example, let's write a program that greets hello to each member when there is an array with a list of names.

この時の for の後ろに書いた変数はループ変数というもので、この変数に配列の中のデータが 1 つづつ入ってきます。例として、名前のリストが入った配列があった時に、それぞれの名前に対して hello と挨拶するプログラムを書いてみましょう。

```
> group = ["kazu", "goro", "tetsu"]
[ "kazu", "goro", "tetsu" ]
> for (member of group) {
    console.log("Hello " + member)
  }
Hello kazu
Hello goro
Hello tetsu
undefined
```

In the first line, we set an array containing the names of three people in a variable called group. The third line creates a loop of the array using a loop variable called member. In member, array elements kazu, goro, and tetsu are entered in turn for each loop. In the 4th line, "Hello" followed by "member" is displayed.

It turns out that a loop can be created for each element of the array in this way.

The repetition so far has been an example of a repetition in which the number of repetitions is known in advance. However, you sometimes want to repeat a program that you do not know in advance how many times it needs to be repeated. In such cases, it is used the syntax of repetition by conditional expressions. In order to repeat the process while a certain condition is met, it can be expressed in the following format using the while keyword.

```
while (conditional expression) {
  processing
}
```

At this time, while the conditional expression is true, the processing part is executed repeatedly.

As an example, consider a program that finds the largest circle whose radius is an integer and whose area does not exceed 1000.

Recall the area function that we learned in the previous section.

```
> area = (radius) => radius * radius * 3.14
[Function: area]
> i = 1
1
> while (area(i) < 1000) {
    console.log(i, area(i))
    i = i + 1
  }
1 3.14
2 12.56
3 28.26
(omitted)
15 706.5
16 803.84
17 907.46
18
```

The area function is defined in the first line. If you have forgotten the meaning of this function, see section 1.4.3.

The variable i is set to 1 in the third line. Find the circle we want by raising the value of this variable one by one in the loop.

The expression while (area (i) <1000) { on the fifth line means repeating the {} inside while the area (i) <1000 is true. If area (i) <1000 is true, area (i) is less than 1000, so if the condition of the desired circle is met, {} will continue to be executed.

The 6th line displays the current i, the radius of the circle, and area (i), the radius of the current circle.

In line 7, i is set to i + 1. This assignment of variables will increase the value of i by one. In other words, if i is 1, the value of i increases by 1 each time the loop goes, such as 2 if it is 1, and 3 if it is 2.

Let's look at the resulting output. The last line displaying the area is `17 907.46`. Since this is the last display through the loop, this is the radius and area of the circle we want.

Using the while statement in this way, we can write a program that uses a conditional expression to determine whether to exit a loop when you do not know how many times the loop statement will run.

Sum up

In this section, you learned how to iterate using for and while.

1.4.4 Errors

While you have been running your program with Deno so far, you may have seen errors due to accidental input mistakes or operation mistakes. When an error comes out, it will surprise and trouble you, but the error itself is never bad. This section explains how to handle errors.

An error is a phenomenon where a program stops unexpectedly. For example, when displaying a string, if you forget the single quote that closes the string, the following error will occur.

```
> console.log("hello)
error: Uncaught SyntaxError: Invalid or unexpected token
|> <unknown>:1:13
    at evaluate (js/repl.ts:98:34)
    at replLoop (js/repl.ts:156:13)
```

There are various reasons for errors. Here we will look at errors that occur in Deno programs.

Types of error

There are two main types of errors that occur in Deno programs. One is an error that occurs when the syntax is incorrect, and the other error occurs when Deno is unable to process the data successfully while it is running.

The first error is when there is a mistake, contrary to the grammar established for writing a Deno program as shown above. In the error message, “SyntaxError” is displayed. The translation of “Syntax” is “Koubun” in Japanese. When a SyntaxError is displayed, carefully review whether you have NOT forgotten to write something, such as "(double quotes) and () parentheses."

The second error occurs when data cannot be processed successfully during execution. The point is “while running”, and conversely, it means “no error if not executed”. Let’s check with the program.

If we mistakenly write console.log as console.lo in a program that displays hello with the console.log function, an error will naturally occur.

```
> console.lo("hello")
error: Uncaught TypeError: console.lo is not a function
|> <unknown>:1:9
    at <unknown>:1:9
    at evaluate (js/repl.ts:98:34)
    at replLoop (js/repl.ts:156:13)
```

The content of the error message is `TypeError: console.lo is not a function`. We interpret it as "I tried to execute a function called console.lo as written in the program, but the program cannot do anything because console.lo is not a function."

On the other hand, look at the following program.

```
> if (false) {
    console.lo("hello")
  }
undefined
```

As in the first example, we are trying to execute a function called console.lo, but I don't get any errors. This is because the result of the conditional statement written above console.lo is always false, so the program does not execute this console.lo line.

Unlike the previous SyntaxError, the absence of a function called console.lo means that the Deno program didn't know until it was actually run.

There are various patterns of errors. In each case the cause of the error is displayed at the top of the error message. If an error occurs, you may be surprised at how many lines of the message are displayed in English at one time. Let's make it a habit to calm down and read the contents of the top line of the error message.

Handle Errors

This section explains how to deal with errors. Some errors can be resolved by modifying the program, such as SyntaxError, but in some cases, the error itself cannot be avoided, such as a network error. In such a case, it is troubled if the program stops due to an error, so you may need to write a program to continue the process as if the error did not occur by stopping the error and recording the error or notifying the error to user.

Deno can deal with (handle) errors using the keywords try and catch as follows:

```
try {
  Actions that may cause error
} catch (e) {
  What todo when an error occurs
}
```

Let's write a program and confirm it.

```
> try {
    console.lo("hello")
  } catch (e) {
    console.log("An error has been caught.")
    #@# console.log("エラーをキャッチしました")
  }
An error has been caught.
undefined
```

I wrote a call to the console.lo function that would result in a TypeError under the try keyword. This mistake resulted in an error, but because it was written inside try statement, the error was caught and the line under catch statement was executed. As a result, the message “Caught an error” was displayed and the processing was completed, no error message was displayed, and no error occurred as a whole program. This means that you have dealed with (handled) the error.

Sum up

In this section, we learned about Deno errors. We also learned the syntax to handle errors with the keywords try and catch.

1.5 Asynchronous Operations

This section describes a process called asynchronous processing.

1.5.1 Synchronous and Asynchoronous Operations

Deno has a special process called asynchronous processing.

First, I explain the difference between synchronous and asynchronous. Synchronous means to perform each process in order. On the other hand, asynchronous means that each process is not performed in order, but postponed.

Think of a cash register at a convenience store. When you go to the cashier, the store clerk will do the following work and then check out.

- Read product barcode
- Check out goods
- Bag the goods
- Give the goods to the customer

After completing the above steps, the process of checking out for the next customer begins.

Let's consider a slightly different pattern where a procedure for warming a bento was included.

- Read barcode of goods
- Check out goods
- Bag the goods except the bento
- **Put the bento box in microwave and set timer**
- Give the goods except the bento box to the customer

At this stage, the bento is still being warmed up in the microwave, but the store clerk will begin checking out for the next customer. Then the following step is inserted when bento box has been done warmed.

- Give the bento box to the waiting customer

The store clerk then continues to check out for the next customer. In this story, there was one task that had different characteristic from the other tasks of the store clerk. It's a task of making the bento box warm with microwave oven. In other tasks, the store clerk did the task in order and did the next after one thing was done. But only in task of making the bento box warm, the clerk began next work before he finished it.

In this store clerk's example, the warming with the microwave oven is asynchronous processing, and the rest is synchronous processing. Like this example, in programing, you ofthen find a type of procedure that main program do a job, sub programs do some job in background and call the main program when the background job is done. Such processing is called asynchronous processing.

I introduce how to perform basic asynchronous processing in the following.

setTimeout

The setTimeout function is a basic function for asynchronous processing. This function is executed in the following format. It takes a function and time (milliseconds) as arguments and executes the specified function after the specified milliseconds.

```
setTimeout (a function you want to postpone, milliseconds)
```

Let's check the behavior using the interactive shell.

```
> hello = () => {
    console.log("hello")
  }
> setTimeout(hello, 1000)
1
> hello
```

We defined a function called hello that displays hello in the first line.

On the fourth line, we put the hello function defined above in the first argument of setTimeout and 1000 (milliseconds) in the second argument.

When you run the above program, you should see hello after 1 second (= 1000 milliseconds) executing the last setTimeout line. If you don't realize that the time is a bit off, try running the last line several times.

Next, change the the part of 1000 to 3000 (3 seconds) and try again.

```
> setTimeout(hello, 3000)
2
> hello
```

It should be much slower until hello appears. If you use setTimeout like this, you can execute the specified process after the specified number of milliseconds.

Next, let's execute another process right after the setTimeout call. Now try opening the file in an editor, not an interactive shell. Then try typing:

```
setTimeout(() => {
  console.log(2)
}, 1000)

console.log(1)
```

We use the format of () => {} immediately after setTimeout. This is an arrow function that is passed directly as the first argument of setTimeout.

Save this file as test.js and run it with deno.

```
$ deno test.js
1
2
```

1 should appear immediately and 2 should appear a little later. Why isn't 2 displayed first? In the above program, We `registered` the asynchronous process of `function () {console.log (2) }` to be executed after 1 second (1000 milliseconds). At that time, only registration is performed, the process proceeds as it is, `console.log(1)` is executed, and 1 is displayed. Then one second later, the process registered as an asynchronous process was executed and 2 was displayed.

Column: Why do asynchronous operations exist?

Why do we need something like asynchronous processing in the first place? The reason is making processing efficient. We use asynchronous processing for the process that takes time but does not require much computation such as network access and file access. If the program waits while doing such processing, the computer will be able to calculate, but will wait without calculating. In the case of a store clerk at a convenience store, it is a waste of time to wait for the task of making a bento box warm with a microwave to end without doing anything. To avoid wasting time, set a timer, do other work, and return back to the pending work from the middle when thee task making bento box warm is over. The way of asynchronous processing is the same also in Deno. The network access or file access will take time, but the computer will be idle during that time, so that such processing will be asynchronous processing and other computations will be executed instead.

Sum up

I explained that there are differences between synchronous processing and asynchronous processing in processing of Deno. I also explained synchronous processing is executed one by one in order from the front, asynchronous processing is not executed immediately, but is executed at an appropriate timing later.

The function setTimeout is a basic function for asynchronous processing. By using this function, processing could be executed asynchronously after a certain period of time.

1.5.2 Promise

Promise is a data type that applies asynchronous processing. In this section, I will explain what the promise is.

The name Promise means Yakusoku in Japanese. Why is such a name? Promise promises that some data can be prepared at some point in the future by applying the asynchronous processing described in the previous section. When the promised data is ready, the promise is fulfilled and the data can be retrieved. On the ohter hand, the promise may be broken, in which case the promise is rejected and the data is not retrieved and an error is returned instead.

The image of a promise is similar to a buzzer that is given when a product is purchased at a food court. With the buzzer, you can choose your seat, do another shopping, talk with people, and do whatever you like, but when the buzzer sounds, you will go to pick the product up. The buzzer itself is not a product, but when the buzzer sounds, it can be exchanged for a product.

Deno's promise is similar to this buzzer, and the promise itself has no value. It is possible to get the data from the promise when the the promise ready for tha data.

I will introduce some typical functions that return promises in the following.

fetch

Let's look at a concrete example using fetch, a representative function that returns promises. fetch is a function that takes the URL of a web page as an argument and returns information about that web page.

```
fetch(URL you want to fetch)
```

Let's try requesting the web page `https://example.com`.

```
> fetch("https://example.com")
Promise {}
```

We can see that fetch is indeed returning a promise. To retrieve a value from this promise, register a function in the promise's .then method. First, let's put the fetch result into a variable.

```
> res = fetch("https://example.com")
Promise {}
```

This sets promise of fetch to variable res. To retrieve the promise content from here, register a function to process the data in .then. Here, register console.log. You can display what data this promise has by registering console.log.

```
> res.then(console.log)
Promise {}
```

```
Response { status, url, statusText, type, bodyUsed, trailer, headers, body,
redirected }
```

Promise {} followed by the line `Response {status, url, statusText, type, bodyUsed, trailer, headers, body, redirected }`. This indicates that the promise returned by fetch will pass a Response object to console.log function.

This Response object contains various information about https://example.com. Let's display and examine some properties in the Response object

```
> res.then((r) => { console.log(r.status) })
Promise {}
200
```

`r.status` is the status of Response, which contains a number indicating whether the network request was successful. In the example above, the request was successful, so a success status code of 200 is displayed.

```
> res.then(r => { console.log(r.statusText) })
Promise {}
OK
```

`r.statusText` is the textual representation of Resnponse status. If successful, the expression of status is OK.

```
> res.then(r => { console.log(r.headers) })
Promise {}
Headers {}
```

`r.headers` is a Headers object. Various meta information is sent and received as headers in network requests (more precisely, HTTP requests), and the Headers object contains such meta information. Here, let's take a look at the value of Content-Type, which is a header indicating the data type of Response.

```
> res.then(r => { console.log(r.headers.get("Content-Type")) })
Promise {}
text/html; charset=UTF-8
```

As you can see, this response shows that the Content-Type header has the value `text/html; charset=UTF-8`. This means that the response data is written in html format and that the character code is UTF-8.

The text() method of the Response object is a function that returns a promise, and the data of promise becomes the contents of the website (HTML source code). To retrieve this data, write:

```
> res.then(r => r.text().then(console.log))
Promise {}
> <!doctype html>
<html>
<head>
    <title>Example Domain</title>
    ...(omitted code)
```

The HTML source code is displayed. This content is the HTML source code distributed from https://example.com. If you go to https://example.com in your browser and select View page source from the right-click menu, you should see the same HTML.

Sum up

In this section, we learned how to use the promise. We also learned that fetch is a function that returns a promise.

1.5.3 async and await syntax

The async await syntax is for more convenient use of promises. The basic usage of async await syntax is as follows.

```
async function () {
  processing using await
}
```

The usage of await alone is as follows

```
variable = await function that returns promise
```

If you write like this, the `await promise` part will stop there until the promise is resolved. Processing resumes at the timing when the promise is resolved, and the data resolved by the promise is assigned to the variable.

Rewriting the fetch example in the previous section with async and await syntax is as follows.

```
async function getSite() {
  const res = await fetch("https://example.com")
  const text = await res.text()
  console.log(text)
}
```

Since there are two promises, one isreturned by fetch() and the other is returned by res.text(), the number of awaits is also two.

If you use await like this, you can omit the registration of the callback to the promise then () and use the operator await to retrieve the contents.

1.6 Let's use Modules, create Modules

1.6.1 What's a module?

A module is like a toolbox for a program. There are various tools in the toolbox, and you can make or repair something depending on the use. You can write a program with less effort by selecting the most useful module for the work you want to do.

To use a tool, you need to take it out of the toolbox and hold it in your hand. In Deno, you select the function you want to use from the module and load it into the program.

To import a module, use the keyword import and use the following format.

```
import { Function1 you want to use, Function2, ... } from "Module URL"
```

fmt/colors

As an example, let's take a look at how to use the fmt/colors function that is in standard module to colorize text.

Enter the following and save the file as color.js.

```
import { red, green } from "https://deno.land/std/fmt/colors.ts"
console.log(red("Foo"))
console.log(green("Bar"))
```

The first line means taking in the functions red and green from the module `https://deno.land/std/fmt/colors.ts`. red and green are functions to make the string red and green, respectively.

In the second line, the red function is used to display the character string Foo. The third line uses the green function to display the string Bar in green.

Once saved, enter `deno color.js` in the terminal and run it. Then, it will be displayed as follows.

```
$ deno color.js
Download https://deno.land/std/fmt/colors.ts
Compile https://deno.land/std/fmt/colors.ts
Foo
Bar
```

The first line "Download ..." means that the fmt / colors module is being downloaded. In this way Deno will automatically download the necessary modules at runtime. The second line "Compile ..." means that you are compiling the downloaded module. Download and compilation are performed only for the first time, and the cache is used after the second time. In the third line, Foo should be displayed in red, and in the fourth line, Bar should be displayed in green.

encoding/csv

Next, let's use the standard module encoding/csv. This module reads csv (Comma Separated Values) format strings and files and converts them into arrays.

Enter the following and save it with the file name csv.js

```
import { parse } from "https://deno.land/std/encoding/csv.ts"
async function main() {
  const result = await parse("a,b,c\nd,e,f")
  console.log(result)
}
main()
```

In the first line, the function parse is imported from the encoding/csv module. parse is a function for converting csv strings to arrays.

The second line declares an asynchronous function named main async function main(){...}. Since parse is an asynchronous function (a function that returns a promise), declare it like this to use await.

In the third line, you assigns the result of parsing the string given as await parse(...)

to the variable named result. The 4th line shows result.

Let's run this file by typing `deno csv.js` from the terminal. Then it should be displayed as below.

```
$ deno csv.js
[ [ "a", "b", "c" ], [ "d", "e", "f" ] ]
```

You can see that the csv string `a,b,c d,e,f` has been converted to an array. csv is a format used for various software inputs and outputs. For example, a table created in Excel can be output in csv format. If you use encoding/csv module, you can convert table data created in Excel into array format and process it with Deno program.

Sum up

In this section, I described the basic usage of the module and examples of how to use the fmt/colors module and encoding/csv module. There are many other modules available for Deno. You can learn more about other standard modules on the standard module home page (https://github.com/denoland/deno_std).

In addition to the standard modules, there are external modules created by volunteers. External modules include many modules useful for practical programming, such as modules for database access, web frameworks, and template engines. You can find out more about external modules on the module registry (module registration site) page (https://deno.land/x/) that gathered external modules.

1.6.2 Try to make a module

We've looked at some ways to use modules. I will explain how to create modules in this section.

Creating a module is not so difficult. In fact, the module itself is just a file containing a program. If you specify the functions that can be used from outside the module with by adding the export keyword to the function and variable of a normal program, that completes the module.

The export keyword is used in the following format.

```
export const variableName = value
export let variableName = value
```

For functions, the format is as following.

```
export function functionName() {
  // The implementation of the function
}

export async function functionName() {
  // The implementation of the function
}
```

By using the above notation, variables and functions for which export is specified can be imported and used from outside the module.

Let's create a simple module for testing. Create the following file and save it with the file name hello.js.

```
export function hello() {
  console.log("hello")
}
```

hello is a function that outputs `hello`. The `function` keyword is preceded by the `export` keyword, so this hello function can be used from another program with import.

Create the following file and save it as main.js in the same folder as hello.js.

```
import { hello } from "./hello.js"
hello()
```

Let’s run this main.js

```
deno main.js
```

You see `hello`. If it doesn’t work, check whether the file is saved in the right place or whether the file name is correct.

You have created a module called hello.js that exports the hello function.

Publish your module

In the above, we introduced how to use the created module from another program in the same computer. So how can you use your module from another computer?

In order to use the created module from another computer, it is necessary to make the module public on the Internet and have a URL. Here, we will publish the module to the Internet using a service called gist. This example uses gist, but you can use any service that can upload files to the Internet.

Go to https://gist.github.com/ while logged in to GitHub. The screen for creating a new gist will appear. Enter the hello.js program above.

```
export function hello() {
  console.log("hello")
}
```

Then enter hello.js in the `file name` input box and press the Create Public Gist button. This will create a gist named hello.js. Press the Raw button to the right of the file name hello.js. Raw means uncooked, and it is a link to a URL that represents the contents of hello.js itself. If you press Raw to switch the page, you should have transitioned to a page with a long URL, such as `https://gist.githubusercontent.com/username/long symbol/hello.js`. This URL becomes the module URL. Copy this URL and rewrite main.js as follows.

```
import { hello } from "https://gist の URL/hello.js"
hello()
```

And do the same. You should now see hello as before. By placing modules on the URL in this way, modules created on one computer can be used on another computer.

1.6.3 Try to create a server using standard modules

Finally, let's create an HTTP server with Deno using the functions of the standard module. Before that, let's briefly explain what a server is.

What's a server?

The original meaning of a server is the person / thing / system that provides the service. A device in the office where drinking water comes out is called a water server. This is a device that provides a water as a service, so it is called a server.

In the computer world, a program that receives a request from another computer and responds to it is called a server. (We also sometimes call hardwares or computers with such programs "server".)

What's an HTTP server (Web server)?

A server is a program that responds to requests, and there are certain rules like grammar between requests and responses. Such rules are called protocols.

Among the protocols, the rules (protocols) for requesting / replying Internet web pages are called HTTP (Hypertext Transfer Protocol).

(If you want to know about the HTTP protocol itself in details, please refer to denobook01 Chapter 3 "Deno de HTTP Server" or "とほほの HTTP 入門 (http://www.tohoho-web.com/ex/http.htm)".

An HTTP server is a program that responds to HTTP protocol requests. An HTTP server is also called a web server because it is a server that displays web pages.

This section introduces how to create an HTTP server using Deno's standard modules.

The basic functionality for an HTTP server is summarized in the standard module `std/http/server.ts`. Create the following file and save it with the file name `server.js`.

```
import { serve } from "https://deno.land/std/http/server.ts";
window.onload = async () => {
  const body = new TextEncoder().encode("Hello World\n")
  console.log("Server started!")
  for await (const req of serve(":8000")) {
    req.respond({ body })
  }
}
```

In the first line, import {serve} from "..." takes in the function serve from the standard module std/http/server.ts. `window.onload = ...` sets a function to the onload property of the window object. window.onload is a special property, and the function set in this property is automatically executed when loading of the Deno program is completed.

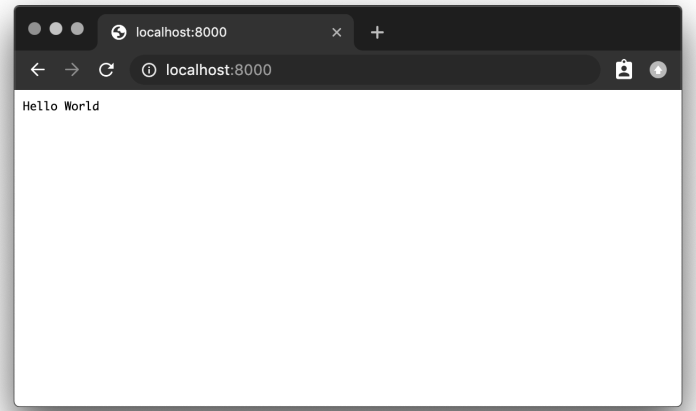
Lines 3-7 are the contents of the onload function. In line 3, the body variable is assigned to the value new TextEncoder().encode("Hello World\ n"). This is the string "Hello World\n" converted to raw data using TextEncoder's encode function. The fourth line outputs the string "Server started!"

On the 5th-7th lines, the block `for await (const req of serve(": 8000")) {... }` is the main loop as a server. First, a call of serve(": 8000") declares that this server runs on port 8000. Each time there is a request (access) to this server, the request information is entered in req and the contents of the loop are executed. The 6th line req.respond({body}) is a process that responds to the request that entered req with the variable body defined above. Since body contains "Hello World\n" data, this server always responds with "Hello World" for all requests.

Let's run this file as follows:

```
deno -A server.js
```

The standard module download should start automatically, and after a short wait you should see `Server started!`. When it is displayed, enter `localhost:8000` in your browser. You should see the following:



```
Hello World
```

We now have an HTTP server that responds to HTTP requests with `Hello World`.

1.7 Sum up

In this chapter, we learned the basics of Deno, the basic syntax of the language, and a little advanced topic such as asynchronous programming and modules.

References

- 確かな力が身につく Python「超」入門 鎌田正浩 ("Super" Introduction to Python by Masahiro Kamata) ISBN-13: 978-4797384406
- JavaScript Primer @azu, https://jsprimer.net

Chapter 2

Creating a command line tool with Deno

Author: @syumai

2.1 Introduction

Hi, Thank you for reading this article. I'm syumai. I started to touch Deno from the end of 2018. In the previous tech book fest 6, I wrote an article about Deno's I/O in Denobook 01. I made some tools in Deno, e.g. dinatra[*1], a small Sinatra-like web framework, dejs[*2], an ejs engine for Deno, dem [*3], a module version manager for Deno, etc. The tool named dem, the most recently created, is a small command line tool made with Deno. There are still few command line tools made with Deno, and I'm writing this article because I want everyone to feel free to make it.

2.2 deno install

The other day, the `deno install` command has been added to make it easier to install command line tools created with Deno. Usage is simple. For example, to install dem:

[*1] https://github.com/syumai/dinatra/
[*2] https://github.com/syumai/dejs/
[*3] https://github.com/syumai/dem/

```
$ deno install dem https://deno.land/x/dem/cmd.ts --allow-write --allow-read
```

When executed, the executable file is stored in the `.deno/bin` directory under the home directory. I recommend you to add this directory into your PATH environment in your shell config, such as .bashrc.

```
export PATH="$HOME/.deno/bin:$PATH"
```

`deno install` is free to name the command to install. The name of the command can be set directly after the install command, such as `deno install ${name}`. This time it was named `dem`. The name cannot be omitted.

`--allow-write --allow-read` after the script URL specifies the permissions that the command to install can access. Granting only write / read permissions does not allow access to environment variables (env) or network connections (net), thus preventing commands from gaining unauthorized privileges and operating in a safe manner.

The installed command can be used easily like a general command line tool.

```
$ dem init
```

2.3 Creating a command line tool

I've introduced how to use `deno install`, but how can we write an installable script? In fact, you don't have to do anything special, just write a Deno script (TypeScript / JavaScript) as usual.

Let's create a command line tool that only displays `Hello, world!`. First, create `hello.ts`.

```
$ echo 'console.log ("Hello, world!");'> hello.ts
```

Let's verify that this script can be run in Deno.

```
$ deno hello.ts
Hello, world!
```

Install this script using `deno install` and run it.

```
$ deno install hello ./hello.ts
$ hello
Hello, world!
```

Isn't it easy to install and run?

2.4 Creating more advanced tools

In order to create more advanced command line tools, it is necessary to use APIs provided by Deno and external modules provided by deno_std. For example, there are many scenes where you want the following:

- Receive command line arguments
- Flag parsing
- Using environment variables

I will introduce how to handle these with Deno one by one.

2.4.1 Receiving command line arguments

Use `Deno.args` to receive command line arguments. `Deno.args` is an array of strings that contains the values passed as command line arguments. Since args includes the path of the executed script itself, the value after `args [1]` is the actual value we want. Let's modify hello.ts to display the received command line arguments.

```
$ echo "console.log(Deno.args);"> hello.ts
$ hello world of deno!
["/home/syumai/src/deno/hello/hello.ts", "world", "of", "deno!"]
```

It was confirmed that the hello.ts path and the three arguments passed were displayed. Also, you can see that the latest contents are reflected without re-installing the script that deno install with relative path like `deno install hello ./hello.ts`. [*4]

You can use `Array.prototype.slice()` to get an array with args removed.

```
$ echo "console.log(Deno.args.slice(1));"> hello.ts
$ hello world of deno!
["world", "of", "deno!"]
```

2.4.2 Parsing flags

For more advanced usage, if you want to parse received command line arguments, you can use `flags` module of `deno_std`[*5]. Create `flags.ts` with the following content:

```
import * as flags from "https://deno.land/std@v0.17.0/flags/mod.ts";

console.log(flags.parse(Deno.args));
```

Let's execute the following:

[*4] This is because deno install does not save a copy of the target script, but an alias.
[*5] https://github.com/denoland/deno_std

```
$ deno flags.ts -a -b bbbbb --c=ccccc d e
```

Then, the following execution result is obtained. (The following is the formatted result, which is normally displayed on a single line.)

```
{
   _: [
      "flags.ts",
      "d",
      "e"
   ],
    a: true,
    b: "bbbbb",
    c: "ccccc"
}
```

From this result, you can see that flags.parse can be used as follows.

- `-a`: Can be used to specify a flag and stores `true` as a value
- `-b bbbbb`: If an argument is added at the end, the value is stored in the key of the corresponding identifier as a character string
- `--c=ccccc`: If a value is given using =, the value is stored as a character string in the key of the corresponding identifier
- `d e`: If an argument is given without a flag, the value is stored as an array in the key of `_`

Let's make the `hello` command correspond to `-h`, `-help`. Edit `hello.ts` to the following contents.

```
import * as flags from "https://deno.land/std@v0.17.0/flags/mod.ts";

type Flags = {
  h?: boolean;
  help?: boolean;
};
```

```
function main() {
  const { h, help } = flags.parse(Deno.args) as Flags;

  if (h || help) {
    console.log("Usage: hello");
    return;
  }

  console.log("Hello, world!");
}

main();
```

Now you can use `hello -h` or `hello --help`. [*6]

2.4.3 Using environment variables

Next, I'll explain how to handle environment variables. Environment variables are necessary if you want to create an application that depends on the user's environment settings. For example, if you want to read the path of a user's home directory on Mac or Linux, you will want to access `$HOME`. (In Windows, it will be `$HOMEDRIVE + $HOMEPATH`) Here, we introduce how to write a program that reads and manipulates the environment variables set in the shell from the beginning, as well as the environment variables set by you.

Reading environment variables

First, let's read and display the environment variable of $HOME. In Deno, environment variables can be acquired as Object by `Deno.env()`. The contents of Object that can be obtained are simple key / value pairs. To try this out easily, let's use the `deno eval` command. The `deno eval` command interprets the passed string as JavaScript and can be executed on the spot. Please read the `HOME` key of the Object obtained by `Deno.env()` as shown below.

[*6] I tried to add support for `--version` at the time of writing, but it doesn't work now because `--version` in the `deno` command is called.

```
$ deno eval 'console.log(Deno.env()["HOME"]);'
/home/syumai
```

The home directory path will be displayed as shown above. (My environment is Ubuntu) For Windows, replace the above with `HOMEPATH` and try. You can also read environment variables that you define yourself in the same way.

If you do not use `deno eval` and create a script file and execute the above contents, you need to give permission to read the environment variable.

```
$ deno script.ts --allow-env
```

Switching program behavior by reading environment variables

Let's add a function to read the configuration file path from the environment variable and apply it to the hello command as a practical part. The environment variable name is `HELLO_CONFIG_PATH`, and the setting item `isCapital` is read from the JSON read from this path. If it is set, it will be capitalized as `HELLO, WORLD`. First, let's create a configuration file with the file name `helloConfig.json`.

```
{
   "isCapital": true
}
```

The program of `hello.ts` is changed as follows. For the sake of space, the help command created earlier has been deleted.

Here, deno_std's path module was used to get the configuration file path. The path module absorbs the differences between OSs and generates file paths, so it is useful when you want to create multi-platform tools.

```
import * as path from "https://deno.land/std@v0.17.0/fs/path.ts";

type Config = {
  isCapital?: boolean;
};

const dec = new TextDecoder();

function main() {
  const config: Config = {
    isCapital: false, // Use false as default value of isCapital
  };
   // Get configPath from environment variable
  const configPath = Deno.env()["HELLO_CONFIG_PATH"];
  if (configPath) {
    const configStr = dec.decode(Deno.readFileSync(path.resolve(configPath)));
    let configJSON: Config;
    try {
      configJSON = JSON.parse(configStr) as Config;
    } catch(e) {
      console.error(e);
      return;
    }
       // If settings can be read, reflect them in config
    Object.assign(config, configJSON);
  }
  if (config.isCapital) {
    console.log("HELLO, WORLD!");
    return;
  }
  console.log("Hello, world!");
}

main();
```

Once you have done this, run the hello command as follows:

```
$ hello
Hello, world!

$ export HELLO_CONFIG_PATH=config.json
$ hello
HELLO, WORLD!
```

Although not shown on the page, environment variables and permission to read the file are required during execution. When you execute it, you can confirm that the command is executed by reading the configuration file specified in the environment variable. Since it is troublesome to ask for permission one by one, let’s finally reinstall the hello command with permissions.

```
$ deno install hello ./hello.ts --allow-env --allow-read
```

Now you can use the `hello` command without stress.

2.5 Distributing the created tool

Finally, I’ll introduce how to distribute tools on GitHub. Since it is the most common, we use GitHub, but Deno scripts can be imported from anywhere, so it only needs to be placed somewhere accessible via HTTP. However, considering version control, it is more convenient to use GitHub.

First, create a repository for this tool. Set the repository name `hello` and change the main file name from`hello.ts` to `mod.ts`. This is because it follows the Deno style guide[*7], and the module’s entry point file should be named `mod.ts`. You can actually choose freely[*8], but here it follows.

The repository URL looks like `https://github.com/syumai/hello`.

As a command to install, the following contents should be described in README.md.

```
deno install hello https://raw.githubusercontent.com/syumai/hello/master/mod.ts
  --allow-env --allow-read
```

[*7] https://deno.land/style_guide.html

[*8] syumai/dem has mod.ts as an entry point, so the command line tool is named cmd.ts.

`raw.githubusercontent.com...` shown here is the URL for accessing the `mod.ts` file directly on GitHub.

To change this to a cleaner URL, the transfer service `denopkg.com`[*9] provided by egoist[*10] or the officially provided `denoland/registry`[*11] should be used. Using denopkg.com, you can rewrite the above URL as follows:

```
https://denopkg.com/syumai/hello@master/mod.ts
```

If you create a release tag on GitHub, you can also write `@master` as `@v0.1.0`. denopkg.com simply forwards denopkg.com to raw.githubusercontent.com with certain rules, so no special work is required. However, since it is a personal service, there is no guarantee of how long it can be used. There is a denopkg.com repository that can be deployed to zeit now[*12], so if you're worried, you can host it yourself.

If you add a module to denoland/registry, you can use the following URL.

```
https://deno.land/x/hello@master/mod.ts
```

Usage is the same as denopkg.com. However, adding modules to the registry is first-come-first-served basis. Therefore, someone may have already registered a module with the same name. Since denoland/registry is a repository that serves as a central registry in Deno, if you create interesting commands or modules, please send a pull request to register the module.

2.6 Finally

If you use the contents described so far, you will be able to create a simple command line tool and publish it on GitHub. If you have an interesting tool, you may add it

[*9] https://github.com/denopkg/denopkg.com
[*10] https://github.com/egoist
[*11] https://github.com/denoland/registry
[*12] https://github.com/zeit/now

to awesome-deno [*13] or share it with Deno's Gitter [*14]. Deno doesn't necessarily need a package registry service such as npm, and I think it's the easiest and most free environment to create and publish a small command line tool using TypeScript. Please make a fun command line tool with Deno!

[*13] https://github.com/denolib/awesome-deno

[*14] https://gitter.im/denolife/Lobby Deno's official Gitter chat room and the center of interaction of the Deno community.

Chapter 3

Bundler for Deno

Author: @keroxp

Translator: @sasurau4, @keroxp

3.1 Introduction

In this chapter, I will talk about the history of the JavaScript's module system and the mechanism of the JS bundler.

I'm exhausted every day because it is too difficult for me to use recent JS bundlers such as webpack. So, I got to want to know the mechanism of JS Bundler by making my own JS Bundler from scratch. As a result, I've pretty understood what JS bundler does, and how it works. And that is why I decided writing this chapter.

You'll realize that the bundler doesn't do much complicated tasks if you know a basic structure and processes about bundling.

For those who write JS for the browsers, bundlers are an inevitable and bothering tool, but if you know how they work, you might not be afraid.

The most basic definition of JS bundler is "Traversing JS files recursively, building dependency graph of modules, and combine those files into a single file.".*1

Why is that necessary in web front end development? In Ruby, Python, Java, PHP, or any other language, we need not to combine multiple source files into single file.

The module system of programming language is closely related to its design. There is an exceptional reason why JS needs a unique way

*1 Modern complex bundlers (e.g. webpack) have many other secondary features. Many people seem to consider it usual. But the core functionality of bundler is only this.

3.2 The Module Eve (~2009)

JavaScript was originally developed to enable dynamic web pages on the browser. In order to execute JS on the web page even now, only way is adding script tag in html that refers source scripts. (Or writing code directly into tags).

JS appeared in the 90s, there were few functions for universal programing until the end of the 2000s, and the role required of Web pages was much simple. Most JS scripts were developed with single file, and no one had any trouble.

The specification of HTML5 was formulated in the latter half of the 2000s, and as the things that could be done with JS gradually increased, the scale of JS development inevitably increased. If the program exceeds 1000 lines, it will be difficult to continue developing with one file. Then, separating files is a natural idea for a modern program paradigm.

However, JS at that time had no way to rely on external files. The browser could only execute scripts that described in script tags serially, so one JS file could only depend on the contents of that file.

Therefore, the technique of **Global module definition** has been devised as a way to realize a module system in JS. In JS, variables declared without `var` are defined as global variables and can be referenced from all scripts thereafter.

Script tags are executed serially in the order described in html. So variables defined as a global can be referred in scripts loaded later.

```
<script src="./module.js"></script>
<script src="./index.js"></script>
```

Definition of global module(module.js)

```
MyObj = { value: function () { return 1; }};
```

Loading of global module(index.js)

```
var value = MyObj.value();
```

Though it was a hack-like method using a dynamic and slightly rough global space mechanism, it became the most common method in the JS module system because it was easy to use anyway. After JS got the official module system, it is still the most used **for various reasons**[*2].

3.3 Node.js and CommonJS (2009 ~)

JS was still idyllic like that, but the turning point was come in 2009. It's Node.js[*3].

Node.js is a project started by Ryan Dahl [*4], the founder of Deno. Node.js was originally started for efficient IO on the server side. By selecting JS for its runtime, Node.js made JS a useful script language on the server side.

Before the advent of Node.js, JavaScript had no place for running other than the browser, and there was very little that could be done within that browser, so it seemed only like a toy of web engineers.[*5]

Node.js was breakthrough in two ways for JS. One was the event-driven server-side runtime.[*6]

[*2] It's very sad.

[*3] https://nodejs.org/

[*4] https://en.wikipedia.org/wiki/Ryan_Dahl

[*5] It may be overstated. But in fact it felt like that. well-experienced programmers used C, C++ and Java for building servers. On the other hand, JavaScript seemed secondary language so that PHPer used JS in one hand and graphic designers struggled to use JS. Also, in the consumer app market that gained momentum after the appearance of the iPhone, Objective-C(iOS)/Java(Android) became popular. On server side, Ruby on Rails that was a framework famous in good and bad reputation was also dominant. So there was no place of using JavaScript.

[*6] Node.js was not the first server-side JS runtime. However, it was breakthrough that you could start the web server with ease without unfamiliar configuration file of Apache. Further, it was also breakthrough that asynchronous processing by single thread that was different from thread-based that was the mainstream at that time achieved high efficiency of processing IO without resource contention. The bad culture that controlling threads meant controlling the program was driven away by the advent of Node.js. In HTTP, which is a TCP-based protocol, it is very difficult to perform asynchronous processing of a single TCP socket on a thread base programming. If you build an HTTP server with Ruby Rack without any thoughts, you will be surprising at that there is only one HTTP connection that can be processed at the same time. On that point, Node.js is versatile enough to solve the C10K problem even with servers

Another one was **inventing original module system** called require/exports.

Loading and defining of Node.js module(module.js)

```
var fs = require("fs")
exports.value = 1;
```

Loading of Node.js module(index.js)

```
var module = require("./module");
console.log(module.value); // 1
```

Node.js invented its own module system when JS had no official module system. It created a practical module system without any changes to the existing JS grammar by adding 2 kind of objects called `require` and `exports`.

The Node.js module system is very simple,

> "require function returns exports object defined by other modules"

That was all. The Node.js module definition method itself was similar to the module system that had been realized using the global space(window variable). Thanks to the introduction of the require function, it has been reborn as a full-fledged module system.

This method was later called CommonJS and laid the foundation for the JavaScript module system.[*7]

3.3.1 require.js and AMD (2010-)

JavaScript users were shocked by the appearance of Node.js, but it was just a server side story. The browser JS were still using GMD.

jQuery was the most popular JS library on browsers. Until then, the DOM API

made by copying a sample code.

[*7] CommonJS itself is not a module specification, but it is a project that started in 2009 to make a different JS API specification from ECMAScript other than browsers, including the server side. But the project has been stopped. As a result, Node.js module system has been memetically called CommonJS.

provided by browsers was awkward [*8], which required some degree of specialized programming knowledge to create rich content.

Also, at that time, JS API implementations were different with each browser, and it was natural that behaviors differed by browsers when same api called. Internet Explorer's implementation delays and bugs were especially severe, which supported the stability and superiority of Flash Player.

At that time, Flash was more popular on the Web than JS, and Flash was said to be an essential technology for creating dynamic Web pages. However, since the HTML5's specification has been finalized and the iPhone has appeared, the convenience of browser vendors such as Apple and Google had become given priority on the Web. Flash also suffered from slow response to security-related issues. It ended completely in terms of the web content platform.

jQuery provided an easy-to-use API for creating rich content with JS which was thought to be difficult. It became popular with Web designers who used Flash, and the potential of JS began to become a hot topic.

jQuery is also defined as the global module whose reference name was `$`. That was very useful for users because they got to be able to manipulate DOM easily with loading jQuery from script tag.

However, GMD had several problems. The most notable problem was that when using jQuery, other libraries that created references in the global space with the name `$` could not be used.

```
<script src="./module.js"></script>
<script src="./jquery.js"></script>
<script src="./index.js"></script>
```

module.js

```
$ = {value: 1}
```

index.js

[*8] At that time there was no document.querySelector, let alone VDom. The DOM API specification has definitely been improved by jQuery.

```
// $ defined in module.js is overwritten by jQuery and disappears
console.log($); // jQuery
```

Thus, GMD had a critical reference namespace problem. In fact, a library called prototype.js which it was as popular as jQuery also used $ as the reference name. So we couldn't use them at the same time.[*9]

There was a problem that multiple jQuery versions could not coexist when using libraries other than jQuery or plugins of jQuery. Under these circumstances, the limitation of GMD began to appear. It began to recognize as a problem that there was no module in JS.

require.js[*10] appeared shortly after Node.js. require.js was a library inspired by `require` of Node.js and trying to create a module system like that on the browser.

Loading module of require.js format

```
<script data-main="./index.js" src="./require.js"></script>
```

Definition of module with require.js format(module.js)

```
define(function() {
  return {
    value: 1
  }
})
```

Calling module of require.js format(index.js)

```
define(function (require) {
  var $ = require("./jquery.js");
```

[*9] is a little wrong, and there was a mysterious method called jQuery.noConflict(). This is somehow managed to deal with the reference conflict with prototype.js. What was being done inside the method is in the dark.

[*10] https://requirejs.org/

```
  var module = require("./module.js");
})
```

A unique feature of require.js was that there was only one script tag to be described in html. require.js adopted a mechanism to load dependent scripts asynchronously from the URL script set `data-main` as an entry

All module files are defined as functions passed to the function `define`, rather than exposing references to the global scope, each module's export was defined as a function's return value. The require function of require.js is different from that of Node.js, the script was loaded with ajax based the web page url if the module had not been loaded.

This define/require module would later be called **AMD(Asynchronous Module Definition)**. [*11]

3.4 Browserify and Browser Side CommonJS(2011~)

Well, it was require.js that appeared in such a way, but **It didn't get popular at all.** The reason for this is not clear, but I think 2 reasons. One is the module definition method was clumsy and the other is specifications was difficult to understand.[*12]

Also, the problem was the startup becomes slow because the module resolution mechanism was asynchronous that loading modules with ajax after web page loaded.

Additionally there was a problem with the management of complicated external modules. In require.js, all dependent libraries had to be downloaded once, not via script tags because of avoiding GMD.

This led to the creation of a package manager for the browser **bower**[*13]. But it also didn't get popular same as **require.js**.

The main reason why both of require.js and bower didn't became popular was the presence of **npm**. `npm` is the official package manager for Node.js. It is designed simply, inspired by Ruby's **bundler**[*14], so even beginners can use it without thinking

[*11] The feature of AMD was in the Asynchronous = asynchronous part, but it seems that the format of define/require was later called AMD.

[*12] I couldn't figure out any good points of it when I used it at the first time. I felt that it would still be easier to do with GMD.

[*13] https://bower.io/

[*14] Ruby's non-standard package manager. It laid the foundation of package management that

about difficult things.

npm manages to store all dependent modules in a folder called @ <code> {node_modules} for each project. Its way is a bit excessive but this resolved the problem that multiple versions of the same library in project, also machine. That was a difficult point of package management in a lot of programming languages.[*15]

Ryan, developer of Node.js module system, seems to regret that this was the biggest failure of Node.js. [*16]

However, npm was just a convenient mechanism for developers. It was a great development experience that just writing `require` led us be able to use external modules.

Browserify[*17] was developed to use npm and CommonJS modules also in browser side development.

Browserify had devised a completely different mechanism from require.js as a module system. It was developed with the opposite idea of **"Let's run a program written in Node.js format in a browser"** unlike require.js, which tried (and failed) to create a module system within the browser's JS mechanism.

But of course, programs written in Node.js format (CommonJS) don't work in the browser.

```
// Since there is no require function in the browser, it does not work.
var module = require ("./ module")
```

Node.js require is executed synchronously, but such code cannot be ran in the browser. There are several reasons

1. Relative paths cannot be resolved as each scrip doesn' know its location being executed is unknown 2. Libraries installed from npm can't be imoported 3. No way to fetch and execute remote files synchronously

manages whole packages used in the project with two types of declaration files, Gemfile and Gmefile.lock

[*15] Up until then, package managers such as Bundler (Ruby) and maven (Java) had to fix versions of libraries they depend on in one project. Therefore, if two libraries depends on different versions of the same library, neither can't be used.

[*16] I don't think so. The module system of npm is certainly complicated, but it achieved solving a number of problems with its flexibility and simplicity like designed by beginners.

[*17] http://browserify.org/

In other words, it's natural that "Node.js programs can't run in non Node.js environment". However, Browserify solved those problems by interesting way.

That is, as the subject of this chapter, **bundling**[*18].

Bundling is the idea of analyzing a JS written in the CommonJS format and combining all the dependent files into one file that can be executed by the browser.

This invention was epoch-making. It is completely different from every single module system that is designed for browsers than ever before. Browserify was the first successful project of transforming CommonJS program into browser compatible.

We could use jQuery via npm, not via script tag. We could also use multiple versions together thanks to the npm mechanism.

```
npm i jquery
```

JS for browser written with CommonJS format(module.js)

```
exports.value = 1;
```

JS for browser written with CommonJS format(index.js)

```
var module = require("./module");
var $ = require("jquery")
```

Browserify analyzes files to be bundled from entry file, searching for all require function calls and property assignments to exports object recursively.

Until then, it was the best practice to read and place dependent libraries individually with script tags. The idea of combining all the dependent libraries into one js file is very innovative and bundled JS has following advantages:

1. Only one JS file will be loaded via script tag, making loading faster <fn>{script-async}

[*18] It's not clear who invented bundling. The idea itself is similar to the C compiler's linker and it may not worth to identify who devised it.

2. Nice supports from editors for Node.js
3. Many libraries developed for Node.js became to run in browsers

Although JS on the browser side and JS on Node.js were considered different even though the same language at that time, the third influence changed the situation. The interoperability of them had been progressed because major browser(Google Chrome) and Node.js were using same JavaScript engine[*19]

The browser libraries, that were previously read with script tags and used in GMD, were gradually changed to providing distribution of CommonJS format and releasing on npm

The Node.js module system is called CommonJS today.[*20]

Several similar bundlers developed after the advent of Browserify. webpack[*21], FuseBox[*22], rollup.js[*23], etc. provide not only bundling feature same as browserify but also more complex program transformations functionality.

As a result, it seems that people who do not know the bundler can't start front-end development by JS today.[*24] This situation is exactly putting the cart before the horse.

3.4.1 ESModule(2015~)

As the practicality and popularity of JavaScript increased, the opinion that module systems should be introduced to JS also increased. So ECMAScript6 (later renamed ECMAScript2015), the next generation JavaScript, appeared in 2015. In ES2015, various advanced syntaxes and functions can be used, and among them, a standard module system has appeared in JS. That's **ESModule**.[*25]

ESModule was inspired by Node.js CommonJS, but introduced a new syntax instead of a require function.

Definition of module with ESModule format(module.js)

[*19] V8 is a JavaScript engine that was developed for Google Chrome, it has been by far the best in execution speed ever and that's why it was chosen for Node.js.

[*20] The method of providing modules with switching GMD and CommonJS depending on the execution environment were later called UMD(Universal Module Definition).

[*21] https://webpack.js.org/

[*22] https://fuse-box.org/

[*23] https://rollupjs.org/guide/en/

[*24] so-called webpack.config.js phobia

[*25] ESModule is not an official name. It is called for convenience to distinguish it from some existing JS module systems. In ECMAScript Standard, it is simply called Module.

```
export const value = 1;
```

Loading of module with ESModule format(index.js)

```
import * as module from "./module"
import * as jquery from "jquery";
console.log(module.value); // 1
```

In addition, changes have been made to the script tag, and JS files loaded with `type="module"` are interpreted as ESModules, making it possible to use import/export syntax internally.

script tag of enabling ESModule

```
<script type="module" src="./index.js"></script>
```

The biggest difference between ESModule and CommonJS is whether variables can be used in module identifiers or not.*26

In contrast to CommonJS, where the require function dynamically resolves module files at runtime, ESModule was a static module system that explicitly declared dependencies on external modules at the time of writing. *27

However, even after the official module system was provided, Bundler and CommonJS format programs are still mainstream in front-end development. The one reason is that JavaScript written in Pure ESModule cannot solve CommonJS format modules.

Node.js require can use modules that npm install under `node_modules` with the form of `require("jquery")`. On the other hand ESModule in browser can't resolve

*26 Standardization is progressing with a mechanism called dynamic import. It will be officially introduced at ES2020.

*27 I add that this is still controversial. CommonJS's dynamic module system is sometimes criticized for its ambiguity, but there are many advantages where require/exports is a pure JavaScript feature. In particular, the ease of mocking modules in tests can only be realized with CommonJS. Popular test runners such as Jest implement such mocking by replacing require/exports objects with fake ones during test execution.

the module identifier `"jquery"` because it's not Node.js. Also, ESModule syntax is a static grammar, and it cannot change its behavior itself at runtime.

Furthermore, module resolution via script tags resolves import declarations recursively when the web page is loaded, so the startup time gets slow. There is nothing good right now, unfortunately.

3.5 Deno and URL import(2018~)

Meanwhile, Deno appeared in 2018. Deno started as a TypeScript runtime, but later became able to execute JavaScript.

The Deno's module system is implemented on a pure ESModule. It has two big and new features. One is readable both JS / TS directly without any configuration and the other is available to use the URL as a module identifier.

Definition of module with Deno format(module.ts)

```
export const value = 1;
```

Loading of module with Deno format(index.ts)

```
import { serve } from "https://deno.land/std/http/server.ts"
import { value } from "./module.ts"
console.log(value); // 1
```

In Browser and Node.js, TypeScript cannot be directly loaded or executed as a module, but Deno can resolve JS/TS modules that depend on other modules via the built-in TypeScript compiler. There is also an advanced attempt to make HTTP URLs available for module identifiers.[*28]

Ryan regretted that the behavior of Node.js require had become ambiguous and complicated[*29], and created a module system that supports only "with

[*28] Actually, the browser can import via URL. But it's not practical because of the problem that it can only be loaded at runtime as I described in the above. This functionality is a reimportation from the browser.

[*29] https://www.youtube.com/watch?v=M3BM9TB-8yA

extension/relative path or complete URL" as a module identifier.[*30]

These Deno module systems seem to be influenced by Go's solid module system rather than Node.js.[*31]

3.6 Development of bundler in practice

In the first half, we reviewed the history of the JS module system up to Deno. In the second half, I will explain the mechanism of the Deno-style TS/JS bundler **tsb**[*32] developed by me.

Before that, one point must be noted that all the JS programs described in this section are Node.js programs. You may think that "Why using Node.js even though it's a Deno book?" Unfortunately, it's not a joke. Because there is no established method for using the TypeScript API published on npm from Deno. So it is a book of Deno, but I hope you read belows with thoughts upside down.[*33]

The Deno format module has the following characteristics unlike the Node.js CommonJS format.

3.6.1 Resolve only local files with extension and relative path

```
import * as some from "./some.ts" // OK
import * as other from "./other" // NG
import * as foo from "foo.js" // NG
import * as val from "/some/src/val.js" // NG
```

[*30] I add that this is a little debatable. The module system of Node.js certainly behaves strangely, but it is convenient and easy-to-use mechanism for actual users. The version resolution between modules using lock files after the appearance of yarn has made considerable progress. So, I think npm is the one of easiest tool for package management. On the other hand, the fact that npm relies on the centralized registry hosting by npm, inc might be problematic. However, GitHub has recently begun to provide npm registry functionality and GitHub is already playing a central role as a registry not only for programs just like Go but also all other languages. I feel that just company can play the role of providing a stable registry even though it may be centralized.

[*31] Deno was heavily influenced by Go in not only a module system but also its API design. The comment that a certain interface has been entirely copied from Go indicate that briefly.

[*32] https://github.com/keroxp/tsb

[*33] Deno-> Node

In Deno, the dependence on local files always starts with a relative path (`"./"`, `"../"`) and the extension cannot be omitted. This is designed from Ryan's criticism that there was an ambiguity that the correspondence between module identifiers and module files was unknown until runtime because the CommonJS require function was able to resolve modules dynamically.

This design imposes unambiguous dependencies on both the modular system design side and the use side. We can grasp he dependency between modules in Deno format just by parsing the files unlike CommonJS, which can only be understood by running a program. It is a great advantage for users making their own module bundler.[*34]

Also, it may be very intuitive for beginners to simply use relative paths when they want to use what is written in the adjacent file.[*35]

3.6.2 CommonJS format is not supported

```
import * as some from "./some.ts" // OK
const some = require("./some.ts") // NG: require is not defined
import * as React from "react" // NG: The module identifier "react" cannot be resolved because
```

Deno can only use the ESModule mechanism as a dependency on external modules. We cannot use require/exports in Deno unlike we can use a module installed by npm with using require from its name in CommonJS.

[*34] As anyone who has written C or C ++ should know, “dependence on external modules” is often too complex for what you want to do. What I want to do is always to “read a file from somewhere”, but it seems like an overcomplicated mechanism is created in the process of generalizing the mechanism. Personally, I don't like module systems that are searched from search paths such as GCC and ruby. There are too much trouble that are completely unrelated to programming. I think the one of the reasons why npm/CommonJS became popular is that it was easy for users to manage modules.

[*35] "Using the file next to it" seems very easy but often difficult in other languages. Describing the path and URL for import every time has been avoided in the programming paradigm so far, but now there is no bothered at all thanks to the rich IDE support. CommonJS/ESModule is the most advanced and easy to use module system in programming languages.

3.6.3 Both TypeScript and JavaScript can be imported

```
import * someTs from "./some.ts"
import * someJs from "./some.js"
```

Deno can load and execute raw TypeScript directly. This is because Deno compiles TypeScript to JS before running and then runs. JS can also be loaded and executed without type checking.

3.6.4 HTTP URL can be specified for module identifier

```
import { serve } from "https://deno.land/std/http/server.ts"
import React from "https://dev.jspm.io/react"
```

A major feature of Deno is that URLs can be used as module identifiers. Deno recursively downloads uncached URL modules before execution and switches references to local files during module resolution.[*36]

3.7 The structure of bundled files

Before you know what the bundler does, let's take a look at what the bundler generates. The following two files introduced in the previous section are output to a file like following when bundled with Browserify.

module.js

```
exports.value = 1;
```

[*36] TypeScript compiler API has an option to change the correspondence between module identifiers and files.

index.js

```
var module = requier("./module");
var $ = require("jquery")
```

Browserify bundled code (excerpt)

```
(function() {
   // Omitted because of code generated by Browserify
})()(
  {
    1: [
      function(require, module, exports) {
        var module = require("./module");
        var $ = require("jqeury");
      },
      { "./module": 2, "jquery": 3 }
    ],
    2: [
      function(require, module, exports) {
        exports.value = 1;
      },
      {}
    ],
    3: [
      function(require,module,exports) {
        // The code of jQuery
      },
      {}
    ]
  },  {}, [1]);
```

Here is the basic mechanism of the JS bundler. The JavaScript bundled with Browserify is roughly executed in this way.

The structure of bundled file with Browserify

```
(function() {
  / ** Startup process of module system and the entry module ** /
})(
  /** Map of functionalized modules **/,
  /** IDs of entry modules **/
)
```

Each module recursively resolved from the entry file is transformed to be a function with an ID, and an argument of an object with the ID as a key. Each module is contained as is in a function with three arguments. The second part of the array is a map of the module identifier that the module uses for require and the module ID after bundling.

The structure of individual modules bundled by Browserify

```
[function(require, module, exports) {
  // This is the contents of the module as it is
  var module = require("./module");
  var $ = require("jqeury");
}, {
  // Map of module identifiers resolved by require and module IDs
  "./module.js": 2,
  "jquery": 3,
}]
```

From the module system point of view, the difference between Node.js and browser is only whether require and exports objects exist or not, so if you define require and exports that behave in the same way in a bundle file, JavaScript in CommonJS format also be executable on the browser.

The inline module system provided by Bundler is only the difference whether the module resolution destination is a file system or a program expanded inline. What JS Bundler is doing is a imitation what file system is doing.

3.8 Process of bundling

There are three main processes for bundling. That is below.

1. Parsing: Parse JS file to identify modules to be included in the final bundle file 2. Syntax transfomation: Transform import/export descriptions 3. Output: Combine deformed module files and generate bundle file

These three are done in order and never go back and forth.

3.9 Step1: Parsing

The first step of the bundler is to understand how modules that will be bundled depend on each other. A bundler always needs one file as an entry. It takes one entry file as input and outputs one bundle file.

In this step, bundler constructs the graph structure by recursively tracing the modules that the entry file depends on and all of their dependencies. The reason for identifying all the dependent module files from the entry file is to identify the modules to be included in the final bundle file.

index.ts

```
import { value } from "./other.ts"
function add(a: number, b:number): number {
  return a + b
}
const v = add(value, 1);
console.log(v)
```

For example, we are only interested in following part when we bundle the above file as an entry in the first step.

```
import { value } from "./other.ts"
```

From this import declaration, we can see that the module `index.ts` depends on the module `./other.ts`. From here you can see that the bundle file must also include `./other.ts`, so next you need to analyze the dependency module for `./other.ts`.

In this way, in the first step, we parse the JS files with the program to find the part where the reference to the external module is declared in them.

According to the ESModule syntax, there are three following grammars to describe the dependency between modules.

```
// import statement in general
import { other } from "./other.ts"
// dynamic import
import("./some.ts")
// export assignment
export { another } from "./another.ts"
```

The part enclosed in "" is the external module that the JS file depends on, and this is called **Module Specifier**. In the parsing process, we will write a program that finds these three types of syntax and collects the module identifiers from the JS files.

3.9.1 TypeScript Compiler API

The easiest way to parse TypeScript/JavaScript is to use the TypeScript Compiler API.*[37]
This is a library used by TS compiler tsc and can be used from Node.js programs.

```
npm i typescript
```

Following is a simple sample of parsing with TypeScript compiler API.

```
import * as ts from "typescript"
const visit = (node: ts.Node) => {
    console.log(node);
    // 3: Scan the node children recursively
    ts.forEachChild(node, visit);
}
const text = ‘
    import { other } from "./other.ts"
    import("./some.ts").then(some => { })
    export { another } from "./another.ts"
‘;
// 1: Create a source file
const source = ts.createSourceFile("./entry.ts", text, ts.ScriptTarget.ESNext);
```

*[37] TS is a strict superset of JS, so it is also possible to parse JS.

```
// 2: Scan elements in the file
ts.forEachChild(source, visit);
```

At first, we create `SourceFile` from thee TypeScript string to parse TypeScript. `SourceFile` is a set of file name, text, and ECMAScript version to compile. If you don't create it from an actual file, the file name is anything you like.

Next, pass the source file created and the callback function `visit` to `ts.forEachChild`. The programming language is represented by a tree-structured node **AST (Abstract Syntax Tree)**. It's not specific to TypeScript.

Parsing is an operation that scans the AST in order and continues until the target syntax is found.

3.9.2 Find module identifiers from abstract syntax tree

```
  import { other } from "./other.ts"
  import("./some.ts").then(some => { })
  export { another } from "./another.ts"
```

The above script, which is a sample script, is interpreted as the following AST in TypeScript syntax.*[38]

```
ImportDeclaration
 ImportClause
  NamedImports
   ImportSpecifier
    Identifier
 StringLiteral
ExpressionStatement
 CallExpression
  ImportKeyword
  StringLiteral
```

*[38] Syntax node names are mostly derived from ECMAScript. TypeScript without TypeScript-specific grammar can be analyzed as pure ECMAScript

```
ExportDeclaration
 NamedExports
  ExportSpecifier
   Identifier
 StringLiteral
EndOfFileToken
```

You might think "what's this?", because you see unfamiliar words. but the purpose here is to find the three types of module identifier expressions listed above. Among these nodes, the node corresponding to the module identifier is as follows.

```
- ImportDeclaration
  ...
  - StringLiteral <- this ("./other.ts")
- CallExpression
  - ImportKeyword
  - StringLiteral <- this ("./some.ts")
- ExportDeclaration
  ...
  - StringLiteral <- this ("./another.ts")
```

We make the program being able to find these nodes with adding process to the previous `visit` function.

```
const visit = (node: ts.Node) => {
  if (ts.isImportDeclaration(node)) {
    const module = (node.moduleSpecifier as ts.StringLiteral).text;
    console.log(module);
  } else if (ts.isCallExpression(node)
    && node.expression.kind === ts.SyntaxKind.ImportKeyword) {
    const [module] = node.arguments;
    if (ts.isStringLiteral(module)) {
      // You can pass anything other than a string literal to dynamic import
      console.log(module.text);
    }
  } else if (ts.isExportDeclaration(node)) {
    const module = (node.moduleSpecifier as ts.StringLiteral).text;
    console.log(module);
  }
  ts.forEachChild(node, visit);
}
```

If we want to find out what kind of node passed from `visit` is, we use **Custom Type Guard**[*39] named isXXXX in the ts namespace.

If you run this, you should get this output.

```
./other.ts
./some.ts
./another.ts
```

Now we have extracted the dependencies from the file. Although we can pass anything other than a string to dynamic import, we cannot know the value at the time of bundling, which is a static analysis, so it cannot realistically bundle all dynamic import destinations. This will be described later.

3.9.3 Normalization of module identifier

It was relatively easy to extract dependent module identifiers from a single file. However, this module identifier is a troublesome because it is not unique in the true sense even though being called an identifier.

For example, suppose a file in `home/deno/src/main.ts` depends on a module `./other.ts`. In this case, `./other.ts` points to `/home/deno/src/other.ts`. But if the file in `/home/deno/src/sub/sub.ts` depends on the module `../../src/other.ts`, it points to the same file `/home/deno/src/other.ts` on the file system.

In other words, it is not possible to specify the file on the file system only with the module identifier of the relative path. When designing an ESModule module system, not just a module bundler, the uniqueness of a module is determined by two combinations of "file path" + "module identifier".

[*39] This is a self defined type guard for TS that does not have type information in runtime to resolve types such as type alias and interface at compile time defined

```
/home/deno/src/main.ts + ./other.ts => /home/deno/src/other.ts
/home/deno/src/sub/sub.ts + ../../src/other.ts => /home/deno/src/other.ts
```

It may sound difficult, but in this case, "path that resolved the relative path of module identifier from the file path" is the unique module ID for bundling. If you do not normalize module identifiers, you cannot build a dependency graph.

The way of assigning the module ID varies depending on the bundler. I recommend using the "relative path from the entry file" as the module ID.

For example,

* Entry file is `/home/deno/src/main.ts` * `/home/deno/src/main.ts` depends on `./other.ts`

In such a case, the module ID of `/home/deno/src/other.ts` will be `./other.ts`.

Similarly, if `/home/deno/src/sub/sub.ts` depends on `../foo/foo.ts`, the module ID of `.../foo/foo.ts` is `./foo/foo.ts`

In the parsing process, we get the following graph structure as output when the following JS file is an input as an entry.

```
import one from "./one.ts"
import React from "https://dev.jspm.io/react"
import("./two.ts")
export * from "./three.ts"
```

```
{
  "./entry.ts" => [
    "./one.ts",
    "./two.ts",
    "./three.ts",
    "https://dev.jspm.io/react"
  ],
  "./one.ts" => [ ... ],
  "./two.ts" => [ ... ],
  "./three.ts" => [ ... ],
  "https://dev.jspm.io/react" => [ ... ]
  ...
}
```

3.10 Step2: Syntax transfomation

Unlike CommonJS, TS/JS written in ESModule cannot combine the contents of files as they are. This is because import / export declarations are different from require/exports and can only be described in the top level context of a module.

In order to redefine a module as a function like Browserify, it is necessary to transform the part described in ESModule from a raw file into a program that can be described as the contents of the function.

Module with CommonJS format (before bundling)

```
const ts = require("typescript");
exports.value = 1;
```

Module with CommonJS format (after bundling)

```
function (require, exports, module) {
  // OK: no need to to transform content, only reference require/export are replaced
  const ts = require("typescript");
  exports.value = 1;
}
```

Module with ESModule format (before bundling)

```
import * as ts from "typescript";
export const value = 1;
```

Module with ESModule format (after bundling)

```
function (require, exports, module) {
  // NG: According to ES specifications, import/export declarations cannot be written in funct
  import * as ts from "typescript";
  export const value = 1;
}
```

Currently, when writing JS that runs on a browser, we write CommonJS modules with ESModule notation in general. On the other hand since Deno supports only TSModule (a superset of ESModule), all we have to transform is TSModule.[*40]

If you only support ESModule, this transformation policy is very simple,

* Replace import declarations and dynamic imports with function calls * Replace export declaration with property assignment

There are only two. As an example, it is transformed like this.

Module with ESModule format (before bundling)

```
import * as other from "./other.ts"
import("./some.ts").then(v => {})
export const value = 1;
export { serve } from "https://deno.land/std/http/server.ts";
```

Module with ESModule format (after bundling)

```
// Bind an object with a function equivalent to import, dynamic import, export to a scope vari
function ({require, exports, requireDynamic}) {
  var other = require("./other.ts");
  requireDynamic("./some.ts").then(v => {})
  exports.value = 1;
  exports.serve = require("https://deno.land/http/server.ts").serve;
}
```

How to transform this will be explained in the next section, but this transformation rule seems quite simple. Since ESModule import/export declarations are designed

[*40] Bundlers currently in use are basically based on CommonJS. The bundler that only supports ESModule has no presence because there is no demand.

with a syntax similar to that of JS variable declarations, it is possible to make this kind of transformation.

3.10.1 How to use TypeScript transform API

TypeScript Compiler API has a function `ts.transform ()`, which allows syntax transformation with code similar to that used during parsing. This is an API that can rewrite only a specific node while accessing a syntax node like `ts.forEachChild()`.

Using the TypeScript transform API is very similar to `Array.map`. As a sample, I made a transformer that changes all string literals in the code to `"deno"`.

```
import * as ts from "typescript"
function transformers() {
  const transformStringLiterals = <T extends ts.Node>(
    context: ts.TransformationContext
  ) => (rootNode: T) => {
    // Using visitor pattern same as forEachChild.
    const visit = (node: ts.Node): ts.VisitResult<ts.Node> => {
      node = ts.visitEachChild(node, visit, context);
      // If node is a string literal, transform it to a string literal "deno".
      if (ts.isStringLiteral(node)) {
        return ts.createStringLiteral("deno");
      }
      return node;
    };
    return ts.visitNode(rootNode, visit);
  };
  return [transformStringLiterals];
}
const text = ‘
  import { other } from "./other.ts"
  import("./some.ts");
  export { another } from "./another.ts"
‘;
const source = ts.createSourceFile("./entry.ts", text, ts.ScriptTarget.ESNext);
const result = ts.transform(source, transformers(), undefined);
// For outputting ts.Node as a string.
const printer = ts.createPrinter();
console.log(printer.printFile(result.transformed[0] as ts.SourceFile));
```

I think it will be output like following

```
import { other } from "deno"
import("deno");
export { another } from "deno"
```

3.10.2 Transformation of import declaration

According to MDN, following formats are valid statement for the import declaration of ESModule.[*41]

```
// 1 default
import defaultExport from "module-name";
// 2 namespace
import * as name from "module-name";
// 3 named
import { export } from "module-name";
// 4 binding
import { export as alias } from "module-name";
// 5 named list
import { export1 , export2 } from "module-name";
// 6 named + binding list
import { export1 , export2 as alias2 , [...] } from "module-name";
// 7 default + named or bindings
import defaultExport, { export [ , [...] ] } from "module-name";
// 8 default + namespace
import defaultExport, * as name from "module-name";
// 9 import only
import "module-name";
```

In conclusion, these need to be transformed in following way.

```
// 1 default
var defaultExport = require("module-name").default;
// 2 namespace
var name = require("module-name");
```

[*41] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import

```
// 3 named
var { export } = require("module-name");
// 4 binding
var { export: alias } = require("module-name");
// 5 named list
var { export1 , export2 } = require("module-name");
// 6 named + binding list
var { export1 , export2: alias2 , [...] } = require("module-name");
// 7 default + named or bindings
var {default: defaultExport, export [ , [...] ] } = require("module-name");
// 8 default + namespace
var defaultExport = require("module-name").default;
var [name, defaultExport] = [require("module-name"), require("module-name").default];
// 9 import only
require("module-name");
```

3.10.3 Transformation of export declaration

Similarly, following formats are valid export descriptions in ESModule specifications.[*42] In addition to this, TypeScript can also export its own type`Enum`, which must be taken into consideration.

```
// 1-1 variable declaration
export let name1, name2, …, nameN; // var, const も
// 1-2 variable declaration with initializer
export let name1 = …, name2 = …, …, nameN; // var, const も
// 1-3 function declaration
export function functionName(){...}
// 1-4 class declaration
export class ClassName {...}
// 1-5 enum declaration ※ TypeScript のみ
export enum Enum { ... }
// 2-1 export list
export { name1, name2, …, nameN };
// 2-2 export list with alias
export { variable1 as name1, variable2 as name2, …, nameN };
// 3 destructuring export with alias
export const { name1, name2: bar } = o;
// 4-1 default export
```

[*42] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

```
export default expression;
// 4-2 default function
export default function (…) { … } // class, function*, enum are also available
export default function name1(…) { … } // class, function*, enum are also available
// 4-3 default with alias
export { name1 as default, … };
// 5-1 export aggregation
export * from …;
// 5-2 export list aggregation
export { name1, name2, …, nameN } from …;
// 5-3 export list aggregation with alias
export { import1 as name1, import2 as name2, …, nameN } from …;
// 5-4 default export aggregation
export { default } from …;
```

These need to be transformed in following way.

```
// 1-1 variable declaration
let name1;
exports.__defineGetter__("name1", function() { return name1; })
// If we find const variables, throw an exception.
exports.__defineSetter__("name1", function(value) { name1 = value; })
// 1-2 variable declaration with initializer
exports.name1 = ..., exports.name1 = ...,
// 1-3 function declaration
function functionName() { ... }
exports.functionName = functionName;
// 1-4 class declaration
class ClassName {...}
exports.ClassName = ClassName
```

Unlike import, export is difficult to have a one-to-one correspondence with variable declaration syntax. So I recommend you to inject the following helper function.

```
// Assigning all properties of "o" to exports
function __export(o) {
  for (const p in o) {
    if (!exports.hasOwnProperty(p)) {
      exports[p] = o[p];
    }
  }
}
```

```
// 2-1 export list
__export({name1, name2, ...nameN})
// 2-2 export list with alias
__export({ name1: variable1, name2: variable2, ⋯, nameN })
// 3 destructuring export with alias
const { name1, name2: bar } = o;
__export({name1, bar})
// 4-1 default export
exports.default = expression
// 4-2 default function
exports.default = function (⋯) { ⋯ }
exports.default = function name1(⋯) { ⋯ }
// 4-3 default with alias
exports.default = name1
// 5-1 export aggregation
__export(require(...))
// 5-2 export list aggregation
const { name1, name2, ..., nameN } = require(...);
__export({name1, name2, ..., nameN})
// 5-3 export list aggregation with alias
const { import1: name1, import2: name2, ⋯, nameN } = require(...);
__export( {name1, name2, ... nameN})
// 5-4 default export aggregation
exports.default = require(...).default
```

3.10.4 Transformation of module identifier

We are ready for bundling by transforming the import and export declarations into function calls and property assignments, respectively. However, as explained in the normalization of module identifier section, module identifiers that exist in modules are not unique in the bundled program. Therefore, a mechanism is required for the require function to return an appropriate module for the module identifier even in the file after bundling.

The structure of bundled file with Browserify was as follows.

```
(function(modules, entryId) {
  // Here is module system of Browserify
})()({
  0: [
    // ./entry.js
    function(require, exports, module) {
      const {value} = require("./module")
    },
    {
      "./module": 1
    }
  ],
  1: [
    // ./module.js
    function(require, exports, module) {
      exports.value = 1;
    },
    {}
  ]
}, 0)
```

Browserify is characterized by numerically normalizing module identifiers, and each module passed to the argument of the whole function is in an array format of `[function, module ID map]`. A map object that contains the correspondence between the module identifier in the module and the module ID after bundling is passed to the second part of the array. This is to ensure that the contents of the first function are correctly handled with require.

However, code modification is required for ESModule bundling, so if you change this module identifier to a normalized module ID at the same time as the import/export declaration modification, the process will be easier.

./sub/sub.js (before transfomation)

```
import * as module "../module.js"
import { serve } from "https://deno.land/std/http/server.ts"
export { some } from "./some.js"
```

./sub/sub.js (after transfomation)

```
function ({require, exporjs, requireDynamic}) {
  // If the entry file is ./entry.js,
  // the module id of ../module.js is ./module.js from the view of ./sub/sub.js
  var module = require("./module.js");
  // No need of transforming URL import
  var { serve } = require("https://deno.land/std/http/server.ts");
  // The module id of ./some.js is ./sub/some.js from the view of ./sub/sub.js
  exports.value = require("./sub/some.js");
}
```

By transforming the module identifier in this way, the amount of description in the final output file described in the next section can be reduced.

3.11 Step3: Output

Now that all the preparations are complete, generate the final bundle file. Recall the code that Browserify generates at the beginning. The bundled code will be running in the following way.

* The whole code is defined as a function * Individual modules are defined as functions * Call the entire function by passing modules as arguments

At the final stage of bundling, we combine the contents of modules that we have transpiled to JS through parsing into a single JS file.

3.11.1 How to use TypeScript transpile API

It is necessary to transpile TypeScript when we combine TS/JS files because the bundled output file is JS. TypeScript Compiler API has an function for transpiling TS files to JS. The process of transforming and transpiling the module at the same time is as follows after normalizing the module identifiers.

```
function transformModule(moduleId: string, text: string) {
  const src = ts.createSourceFile(moduleId, text, ts.ScriptTarget.ESNext);
  const printer = ts.createPrinter();
  const transformer = (context: ts.TransformationContext) => {
    return (node: ts.Node) => {
      // transformation process here
      return node;
    }
  }
  const result = ts.transform(src, [transformer]);
  const transformed = printer.printFile(result.transformed[0] as ts.SourceFile);
  // Outputting the JS file with transpiling ASTs of TypeScript that are transformed
  return ts.transpile(transformed, {
    target: ts.ScriptTarget.ESNext
  });
}
```

3.11.2 Template of bundle file

When all modules have been transformed and transpiled, we combine them with the module system. The easy way is to prepare a template and embed modules there. I prepared a template of bundle file as follows for sample.

```
type ModuleDefinition = ({
  require,
  requireDynamic,
  exports
}: Module) => unknown;
type Module = {
  require(module: string): any;
  requireDynamic(module: string): Promise<any>;
  exports: any;
};
// 1-1: The entire function
(function(modules: { [key: string]: ModuleDefinition }, entryId: string) {
  const installedModules = new Map<string, any>();
  // 2-1: Function to which import declaration is converted
  function require(moduleId: string): any {
    let exports = installedModules.get(moduleId);
    if (exports) {
      return exports;
    }
    // If the module is not installed, execute the function and create exports
    exports = Object.create(null);
    installedModules.set(moduleId, exports);
    const module = modules[moduleId];
    module.call(this, { require, requireDynamic, exports });
    return exports;
  }
  // 2-2: Function to which dynamic import is converted
  async function requireDynamic(moduleId: string): Promise<any> {
    if (modules[moduleId]) {
      return require(moduleId);
    } else {
      return import(moduleId);
    }
  }
  return require(entryId);
})(
  // 1-2: Modules defined as functions with normalized module IDs as keys
  // We embed the transpiled modules here.
  {
    "./entry.ts": function({ require, exports, requireDynamic }: Module) {
      // ...
    },
    "./module.ts": function({ require, exports, requireDynamic }: Module) {
      // ...
    }
  },
  // The ID of the first module loaded at startup
  "./entry.ts"
);
```

3.12 Afterword

I'll end this chapter that is introducing the outline of JS bundler.

I've proceeded explanation with omitting many things due to the number of pages, but I hope that you can understand how the modern ECMAScript bundler works.

Although I wrote the above was a bundler for Deno, the bundler described here is incomplete. In particular, I couldn't explain how to resolve URL import.

We need to download modules locally that are referenced via URL locally. But I omitted description the steps because it's not directly related a bundler works.

Also, it's a little regret that I described only simple samples, not chunks of the code that actually worked. When I started writing this chapter, I wanted to describe the entire code of the bundler. But I quit it because the code was too much than I expected. I only explained necessary parts of TypeScript Compiler API, but there are many other functions in it. To be honest, I have used it without enough understanding.

The practical code for TypeScript Compiler API is in the source code of tsb, which is a Deno bundler made by me. If you really want to make a bundler yourself, see there.

The sample code for this book is also available at `https://scrapbox.io/deno-ja/Denobook_02`.

See you next time.

Chapter 4

Tokio and Deno

Author: @sasurau4

4.1 First of All

It has been a little over a year since Mr. Ryan Dahl, the creator of Node.js, announced Deno at JSconf.EU2018[*1] on June 2 and 3, 2018.

Deno wasn't famous and most of people don't know much about it. The issue for v1[*2] was opened and started counting down to GA. Of course it'll take a few weeks or months, Deno will be ready for production soon.

I've never heard about using Deno for production. That's right because Deno is very much under development. The other hand, deno-ja that is the community of Deno is becoming more active in Japan.

I personally think we, Node.js users, will get confused by migration to zero install world[*3] from node_modules centralarized npm ecosystem. The world is oriented by 2 dominant JavaScript package managers. One is npm that npm v8 will be integrating Tink[*4] and the other is berry that is the project name of yarn v2. So, it's a chance that Deno will be spreading explosively because Deno resolves all dependencies when executing the code, doesn't need any package managers and can run TypeScript code without any dependencies.

[*1] https://yosuke-furukawa.hatenablog.com/entry/2018/06/07/080335
[*2] https://github.com/denoland/deno/issues/2473
[*3] https://yosuke-furukawa.hatenablog.com/entry/2019/06/17/191720
[*4] https://blog.npmjs.org/post/186983646370/npm-cli-roadmap-summer-2019

The prediction is only my opinion and the god only knows whether the future prediction is true or not.

I had interest to Deno because I wanted to use Rust rather than I was sure to the prediction became true.

So let's show Deno's internal details forcused on how to implement asynchronous operations!

The keyword of Deno's asynchronous runtime is Tokio.

Notes: I use `Rust v1.37.0` and `deno v0.16.0` in this chapter. If you try some code on your local or refer to the corresponding code on GitHub, please keep in mind that.

4.2 What is Tokio?

Tokio is the asynchronous run-time for the Rust programiing language. More specifically, Tokio is an event driven, non-blocking I/O platform for writing asynchronous applications with Rust.[*5]

Rust doesn't support asynchronous programming in language specification itself.[*6] That is different from Go or JavaScript. For a long time ago, Rust runtime had libuv that was the library for asynchrono programming used for Node.js.[*7] But it was removed from runtime at 2014.[*8]

The description of using threads in Rust official learning book[*9] implies this historical background.

So Tokio is the essential tool for async programming with Rust.

By the way I recommend you to search words `Tokio rust` when you want to know something about Tokio. Because if you search word only `Tokio`, your search result is definitely filled by infomartion about the group they are very famout for rock band and idol activities in Japan.

[*5] https://tokio.rs/docs/overview/

[*6] This explanation is not accurate, and the situation has changed in Rust v1.36 as described later

[*7] https://github.com/joyent/libuv/wiki/Projects-that-use-libuv

[*8] https://github.com/rust-lang/rust/pull/17673

[*9] https://doc.rust-lang.org/book/ch16-01-threads.html

4.3 How does Tokio work?

Since Tokio's mechanism relies on Future trait, let's take a look at its definition for now.

```
trait Future {
    type Item;

    type Error;

    fn poll(&mut self) -> Result<Async<Self::Item>, Self::Error>;
}
```

`Item` is return type of Future, `Error` is return type of when Future fail and `poll` is function of polling from Tokio's runtime.

You have Tokio do the work `HelloWorld` when you implement some tasks as follows.

```
struct HelloWorld;

impl Future for HelloWorld {
    type Item = String;
    type Error = ();

    fn poll(&mut self) -> Poll<Self::Item, Self::Error> {
        Ok(Async::Ready("hello world".to_string()))
    }
}
```

Below is a sample code that writes `hello world` to `TcpStream` already implemented in Tokio.

main.rs

```
 1: extern crate tokio;
 2:
 3: use tokio::io;
```

```
use tokio::net::TcpStream;
use tokio::prelude::*;


fn main() {
    let addr = "127.0.0.1:6142".parse().unwrap();
    let client = TcpStream::connect(&addr).and_then(|stream| {
        println!("created stream");

        io::write_all(stream, "hello world\n").then(|result| {
            println!("wrote to stream; success={:?}", result.is_ok());
            Ok(())
        })

    }).map_err(|err| {
        println!("connection error = {:?}", err);
    });

    println!("About to create the stream and write to it...");
    tokio::run(client);
    println!("Stream has been created and written to.");
}
```

Enter `nc -l -p 6142` on your terminal and enter `cargo run` on other terminal, then `hello world` is output on the terminal entering nc command.

The code is below.

https://github.com/sasurau4/tech-book-fes-2019-fall/tree/master/hello-world

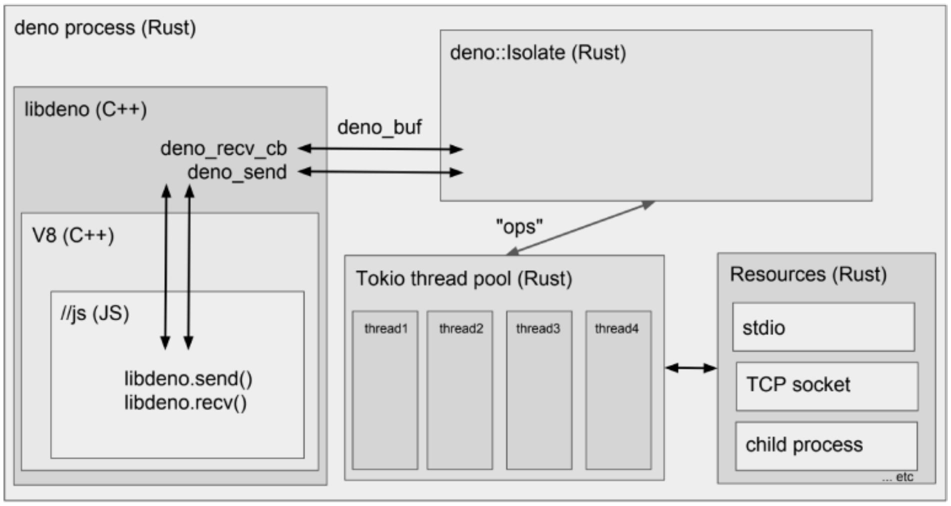
4.4 Where is Tokio in Deno?

So far we've shown the getting started guide of Tokio briefly. Next, let we see where is Deno using Tokio?

I quote the internal details table and figure in the Deno manual as they are very easy to understand.[*10]

[*10] https://deno.land/manual.html#internaldetails

Linux	Deno
Processes	Web Workers
Syscalls	Ops
File descriptors (fd)	Resource ids (rid)
Scheduler	Tokio
Userland: libc++ / glib / boost	deno_std
/proc/\$\$/stat	Deno.metrics()
man pages	deno types



This table and figure describes everything, Tokio is same as a Scheduler in Linux. Let’s take a quick look at how Tokio is used by Deno.

4.4.1 Going Deeper

The entry point of entering Deno’s excutable is `cli/main.rs` as described below.
https://blog.leko.jp/post/code-reading-of-deno-boot-process/#srcmainrs-の-main-関数のアウトライン

There is code to run Tokio in the run-script function.

cli/main.rs

```
372: fn run_script(flags: DenoFlags, argv: Vec<String>) {
373:   let use_current_thread = flags.current_thread;
374:   let (mut worker, state) = create_worker_and_state(flags, argv);
375:
376:   let main_module = state.main_module().unwrap();
377:   // Normal situation of executing a module.
378:   let main_future = lazy(move || {
379:     // Setup runtime.
380:     js_check(worker.execute("denoMain()"));
381:     debug!("main_module {}", main_module);
382:
383:     worker
384:       .execute_mod_async(&main_module, false)
385:       .and_then(move |()| {
386:         js_check(worker.execute("window.dispatchEvent(new Event('load'))"));
387:         worker.then(|result| {
388:           js_check(result);
389:           Ok(())
390:         })
391:       })
392:       .map_err(print_err_and_exit)
393:   });
394:
395:   if use_current_thread {
396:     tokio_util::run_on_current_thread(main_future);
397:   } else {
398:     tokio_util::run(main_future);
399:   }
400: }
```

The first worker created is Future as you can see from the last run. I will leave the explanation of worker once.

State is a State that is accessed from Tokio's thread other than the main V8 thread with ThreadSafeState type as commented in `cli/state.rs#L57-L59`. The contents include @ <tt> {main_module} and @ <tt> {permission} that are executed.

Then, we go to see the definition of Worker, we are at `cli/worker.rs#L17-L23`.

cli/worker.rs

```
17: /// Wraps deno::Isolate to provide source maps, ops for the CLI, and
18: /// high-level module loading
19: #[derive(Clone)]
```

```
20: pub struct Worker {
21:   isolate: Arc<Mutex<deno::Isolate>>,
22:   pub state: ThreadSafeState,
23: }
```

Worker looks like the wrapper of `deno::Isolate`. It implement Future at `cli/worker.rs#L112-L120`, but it just calls poll of isolate.

So let's go to the definition of Isolate at `core/isolate.rs#L162-L182`.

core/worker.rs

```
162: /// A single execution context of JavaScript. Corresponds roughly to the
    "Web
163: /// Worker" concept in the DOM. An Isolate is a Future that can be used
    "with
164: /// Tokio.  The Isolate future complete when there is an error or when all
165: /// pending ops have completed.
166: ///
167: /// Ops are created in JavaScript by calling Deno.core.dispatch(), and in
    "Rust
168: /// by implementing deno::Dispatch::dispatch. An async Op corresponds
    "exactly to
169: /// a Promise in JavaScript.
170: pub struct Isolate {
171:   libdeno_isolate: *const libdeno::isolate,
172:   shared_libdeno_isolate: Arc<Mutex<Option<*const libdeno::isolate>>>,
173:   dispatch: Option<Arc<CoreDispatchFn>>,
174:   dyn_import: Option<Arc<DynImportFn>>,
175:   js_error_create: Arc<JSErrorCreateFn>,
176:   needs_init: bool,
177:   shared: SharedQueue,
178:   pending_ops: FuturesUnordered<PendingOpFuture>,
179:   pending_dyn_imports: FuturesUnordered<StreamFuture<DynImport>>,
180:   have_unpolled_ops: bool,
181:   startup_script: Option<OwnedScript>,
182: }
```

As described in the comments, Isolate is a struct that represents the execution context of just one JS equivalent to a Web Worker.

We find the name of libdeno that is in charge of the connection with V8. It means Isolate have a role of the bridge between JS(V8) and Tokio.

Let's take a look at isolate poll function that is the main content of this chapter.

core/isolate.rs

```
fn poll(&mut self) -> Poll<(), ErrBox> {
  self.shared_init();

  let mut overflow_response: Option<(OpId, Buf)> = None;

  loop {
    // If there are any pending dyn_import futures, do those first.
    self.poll_dyn_imports()?;

    // Now handle actual ops.
    self.have_unpolled_ops = false;
    #[allow(clippy::match_wild_err_arm)]
    match self.pending_ops.poll() {
      Err(_) => panic!("unexpected op error"),
      Ok(Ready(None)) => break,
      Ok(NotReady) => break,
      Ok(Ready(Some((op_id, buf)))) => {
        let successful_push = self.shared.push(op_id, &buf);
        if !successful_push {
          // If we couldn't push the response to the shared queue, because
          // there wasn't enough size, we will return the buffer via the
          // legacy route, using the argument of deno_respond.
          overflow_response = Some((op_id, buf));
          break;
        }
      }
    }
  }

  if self.shared.size() > 0 {
    // Lock the current thread for V8.
    let locker = LockerScope::new(self.libdeno_isolate);
    self.respond(None)?;
    // The other side should have shifted off all the messages.
    assert_eq!(self.shared.size(), 0);
    drop(locker);
  }

  if overflow_response.is_some() {
    // Lock the current thread for V8.
    let locker = LockerScope::new(self.libdeno_isolate);
    let (op_id, buf) = overflow_response.take().unwrap();
    self.respond(Some((op_id, &buf)))?;
    drop(locker);
  }
```

```
    self.check_promise_errors();
    self.check_last_exception()?;

    // We're idle if pending_ops is empty.
    if self.pending_ops.is_empty() && self.pending_dyn_imports.is_empty() {
      Ok(futures::Async::Ready(()))
    } else {
      if self.have_unpolled_ops {
        task::current().notify();
      }
      Ok(futures::Async::NotReady)
    }
  }
```

I will omit a little on the way, but it seems that it is processing in the following order in general

1. If necessary, initialize sharedQueue 2. Dynamic import solution 3. Poll the pending ops in order, push to sharedQueue until it fails because of overflow or the ops disappear 4. If the size of sharedQueue is larger than 0, lock V8 Thread, return response to JS side, and remove lock 5. If overflow_response contains a value, extract the value, lock the V8 Thread, return response to the JS side, and remove the lock 6. If there are no pending ops and there are no pending dynamic imports, it return `Ready`, then it will notify the Tokio runtime if there are unpolled ops, otherwise it will return `NotReady`

If isolate returns `Ready` in 6, then the future of isolate is successfully completed.

We saw what kind of task the variable of `main_future` in <@tt>{main_future} is. Then, let's take a look at `tokio_util::run` where Tokio actually run Future.

cli/tokio_util.rs

```
pub fn create_threadpool_runtime() -> tokio::runtime::Runtime {
  runtime::Builder::new()
    .panic_handler(|err| std::panic::resume_unwind(err))
    .build()
    .unwrap()
}

pub fn run<F>(future: F)
where
  F: Future<Item = (), Error = ()> + Send + 'static,
{
  // tokio::runtime::current_thread::run(future)
  let rt = create_threadpool_runtime();
  rt.block_on_all(future).unwrap();
}
```

The implementation of `run` is a function that creates a Tokio runtime and waits for the resolution of all tasks generated from the given Future and returns it.

In other words, it waits for all Future tasks generated by Isolate and finishes when they are finished.

For now, we've seen how the main Future is executed, so let's keep it around this time.

4.5 What is future of Tokio, Rust and Deno?

So far, we've seen how Deno is using Tokio.

Now it's ready for when Deno becomes GA!

However, it is not true. Because Rust plans breaking changes related to future.

Rust is going to incorporate futures crate which is theee key to asynchronous programming as a language standard library to archieve the async/await syntax.

The following articles and tweets are very detailed about this change, so please check them out.

- The future of Rust, so-called the future of future[*11]
  - https://tech-blog.optim.co.jp/entry/2019/07/05/173000
- The prospect of breaking changes in Rust's Futures[*12]
  - https://twitter.com/qnighy/status/1002210554419671040

The language-standard future has just been stabilized in Rust v1.36.0, released in July 2019. The next phase is each librariy will follow the change.

Tokio is also on this wave, and at the time of writing this manuscript as of the end of August 2019, v0.2.0-alpha.3 has been released, replacing the external crate Future

[*11] Additional note for English ver: This article describes the history and the future of Rust's future in Japanese

[*12] Additional note for English ver: This tweets describes why the breaking chenage occurs and the effect to each library in Japanese

with the Future of standard.

It is imporant thing that the futures crate v0.1 `Future` and Rust v1.36 `Future` are incompatible.

If Tokio changes, Deno should follow it. So Deno will also change after Tokio's transition has settled.

I'm getting sad because it means that most of the content described in this chapter will change after it will come to Tokio v2.

Furthermore, as mentioned in both of the above articles, Rust plans to introducing the async/await syntax. So Rust, Tokio and Deno are likely to continue to change significantly for some time.

Not just sad, but it's also a good news because the library breaking changes means a lot of contribute chance to Deno. We'll do it!

4.6 Afterword

In fact, this is the first Tech Book Fest[*13][*14] participation as a member of a circle.

Deno is a very fast project and the big changes go in and out. For example, the command system changed to go run format[*15], Flatbuffer used for communicate with TypeScript and Rust was removed[*16]. It's a lot of fun to watch destruction and creation of the project that users can't see in other OSS projects that are well established and run stably.

I don't know if Deno will become mainstream in the future. However, it's very interesting to know how much effort JavaScript / TypeScript, which is intended to work on a web browser and used by all front-end engineers casually, will become runable on the server side. So I recommend you to jump in and try to contribute.

Also, as we can see from the quoted code, Deno has mostly comments on important parts. TODO comments are often left, so it is recommended to look for and contribute.

By the way, it's very difficult for me to understand the code need knowledge of the OS like syscall or like that. I wrote this chapter about Tokio because I wanted understanding about it even a little. Still the chapter has become just trace the Deno

[*13] Additional note for English ver: https://techbookfest.org

[*14] Additional note for English ver: Tech book fest is the festival where a lot of people bring their technical book that they are written

[*15] https://github.com/denoland/deno/issues/2189

[*16] https://github.com/denoland/deno/pull/2818

code lightly as a whole because I’m not enough understand about Rust and Tokio. So, I will read and write code to learn much for the next opportunity.

The responsibility for this chapter is at @sasura4, so if you find any mistakes, please reply to the following.

https://twitter.com/sasurau4

Have a good Deno Life!

Chapter 5

Deep dive into Deno compiler

Author: @Leko
Translator: @sasurau4

* This chapter is based on my blog post "Dive into Deno: プロセス起動から TypeScript が実行されるまで"[*1]. I revised it to follow up the latest state of Deno and to write this chapter.

5.1 Introduction

This chapter explains what happens when you run the `deno https://deno.land/welcome.ts` command described in the official document Example [*2]. While introducing the code related to Deno's Compiler, I will explain the sequence of steps from when the Deno process starts up until the TypeScript code is executed.

This chapter is not for Deno users, but for advanced users who want to know how to make Deno itself. I hope this chapter will help those who are interested in the Deno itself.

5.1.1 The significance for JSer reading Deno

Deno is a relatively new language runtime built as a secure TypeScript runtime. It has a lot of very powerful functions, but I think end users who consider the Deno runtime is blackbox have great difficulty of debugging, when Deno come to v1 and

[*1] https://blog.leko.jp/post/code-reading-of-deno-boot-process/
[*2] https://deno.land/#example

become popular and when they get into trouble in development with Deno. There is a big difference in the possibility of finding the bug when debugging between when the layers below the processing system are all black boxes and when they are understood about including the runtime. In short, How Deno is works is **a TypeScript runtime written mainly in Rust, C ++ and transpiling TypeScript into JavaScript and executing it in V8**. I also describe Node.js in the same way to compare with Deno. It is **a JavaScript runtime mainly written in C++ and executing JavaScript in V8**. The internal structure is complex enough to perceive with a brief comparison. I think that productivity will improve if we resolve bugs systematically by understanding of an internal structure a little and excluding errors that can not occur theoritically, even though we can not grasp all the process.

5.2 Before code reading

The code introduced in this chapter may differ from the latest code because Deno is very much under development. I recommend you to read whole structure and flow not each line in detail. Because Deno is actively developed, the code introduced in this chapter may differ from the latest code. I think that it is suitable to read the whole structure and flow without being too particular about the deal of each line.

In addition, I will briefly explain the technologies that are not unique to Deno so as not to depart from the main part when reading Deno. Even if you don't have these knowledge, you can get a lot of information in Japanese if you google, so there is no problem to read through this chapter and learn them after reading.

5.2.1 Rust, C++, TypeScript

Since explanations of the syntax, language specifications, and knowledge of built-in modules in these languages are quite a lot, I'll omit them due to space limitations. The main roles of each language are as follows.

* Rust: Deno is mainly written in Rust * C ++: V8 is written in C ++, so the processing required for V8 operations is written in C ++ * TypeScript: used to define standard modules to be used at run time and to transpile TypeScript at run time

5.2.2 V8

V8 is an open source JavaScript and WebAssembly execution engine developed by Google. V8 is also used internally by Google Chrome and Node.js. Deno has also transferred the execution of JavaScript code to V8 and often appears when reading internal code.

However, the V8 code base is very large. It is not realistic to cover all specifications. You should know that JavaScript will be executed when you pass JavaScript code as a C ++ string to V8. The knowledge enables you to understand the intent of many codes when reading Deno's code. If we need about V8 in more details, I'll use footnotes to supplement any further explanation.

5.2.3 Check current location and goal path

Before we start reading the code, I paste a part of the results of running welcome.ts with debug logs. The debug log is very useful as a map showing the course of code reading. If you get lost while following the code or are worried about whether you are following the code correctly or not, use the output of the `debug` macro as a map.

```
$ deno --log-level=debug --reload https://deno.land/welcome.ts
DEBUG RS - deno_bin::startup_data:26 - Deno isolate init with
snapshots.
DEBUG RS - deno::shared_queue:57 - rust:shared_queue:reset
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=339
DEBUG RS - deno_bin:380 - main_module https://deno.land/welcome.ts
DEBUG RS - deno_bin::file_fetcher:121 - fetch_source_file.
specifier https://deno.land/welcome.ts
Download https://deno.land/welcome.ts
...省略...
DEBUG RS - deno_bin::compilers::ts:352 - >>>>> compile_sync START
DEBUG RS - deno_bin::compilers::ts:355 - Running rust part of compi
le_sync, module specifier: https://deno.land/welcome.ts
DEBUG RS - deno_bin::startup_data:52 - Deno isolate init with
snapshots.
DEBUG RS - deno::shared_queue:57 - rust:shared_queue:reset
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=339
Compile https://deno.land/welcome.ts
```

```
DEBUG RS - deno_bin::ops::workers:47 - op_worker_get_message
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=249
DEBUG RS - deno::shared_queue:167 - rust:shared_queue:pre-push:
op=38, off=812, end=1064, len=252
DEBUG RS - deno::shared_queue:186 - rust:shared_queue:push:
num_records=1, num_shifted_off=0, head=1064
DEBUG RS - deno_bin::file_fetcher:121 - fetch_source_file.
specifier https://deno.land/welcome.ts
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=207
...省略...
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=26
DEBUG RS - deno_bin::fs:53 - set file perm to 438
DEBUG RS - deno_bin::file_fetcher:121 - fetch_source_file.
specifier https://deno.land/welcome.ts
DEBUG RS - deno_bin::fs:53 - set file perm to 438
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=26
DEBUG RS - deno_bin::ops::dispatch_json:41 - JSON response
pre-align, len=26
DEBUG RS - deno_bin::compilers::ts:378 - Sent message to worker
DEBUG RS - deno_bin::compilers::ts:388 - Received message from
worker
DEBUG RS - deno_bin::compilers::ts:392 - Message:
{"emitSkipped":false}
DEBUG RS - deno_bin::compilers::ts:475 - compiled filename: "/home/
leko/.cache/deno/gen/https/deno.land/welcome.ts.js"
DEBUG RS - deno_bin::compilers::ts:418 - >>>>> compile_sync END
DEBUG RS - deno::modules:203 - Event::Fetch
DEBUG RS - deno::modules:465 - register_complete https://deno.land/
welcome.ts
DEBUG RS - deno::modules:209 - Event::Fetch
Welcome to Deno
```

5.2.4 My code reading environment

You can follow the code from the GitHub, but it is more convenient to read it on your local machine. If you don't have your favorite code reading environment, I recommend you the following environtment to read source code. I cloned deno and used Visual Studio Code with the extension Rust (rls) [*3] for the source code reading. Rust LanguageServer enables me to read code easily with the code jump to definitions I cloned deno and used Visual Studio Code with the VSCode extension Rust (rls) [*4] for the source code reading. Rust LanguageServer enables me to read code easily with the function of go to definition.

5.2.5 Repository of Deno

The source code of Deno is available on the GitHub denoland/deno[*5] repository. I'll explain the source code based on the master branch's HEAD (`595b4daa`) at the time of writing.

5.3 /cli/main.rs

The first thing to do when reading code is to find where to start reading code. Where is the file (entry point) that is executed first when the `deno` command is executed? It's `/cli/main.rs`. There is a function called main at the end of the file, which is the entry point.

```
fn main() {
  // ...
  let args: Vec<String> = env::args().collect();
  let (flags, subcommand, argv) = flags::flags_from_vec(args);
  // ...
  match subcommand {
    // ...
    DenoSubcommand::Run => run_script(flags, argv),
    // ...
  }
}
```

The main function processing can be broadly divided into two blocks: command line argument parsing and subcommand execution.

[*3] https://marketplace.visualstudio.com/items?itemName=rust-lang.rust
[*4] https://marketplace.visualstudio.com/items?itemName=rust-lang.rust
[*5] https://github.com/denoland/deno

5.3.1 Parsing command line arguments

Parsing command line arguments is done with the `flags::flags_from_vec` function (`/cli/flags.rs`). The library called clap [*6] is used for the parsing process. Options and subcommand parsers are defined by the `create_cli_app` function called from the `flags_from_vec` function. When the command `deno https://deno.land/welcome.ts` is executed, the corresponding subcommand is `[SCRIPT]`. clap allows us to define an argument fallback that is not one of the predefined subcommands (`bundle`, `run`, etc.) and matches it. Since the code is quite long, I only extract it relevant to processing.

```
pub fn create_cli_app<'a, 'b>() -> App<'a, 'b> {
  add_run_args(App::new("deno"))
    .bin_name("deno")
    // ...
    .settings(&[AppSettings::AllowExternalSubcommands])
  // ...
  ).subcommand(
      // ...
      SubCommand::with_name("[SCRIPT]").about("Script to run"),
    )
}
```

Next, read the process of formatting the result of parsing the command line arguments. The parsing result is pattern-matched, and the necessary formatting processing is performed for each matched subcommand. `(script, Some (script_match)) => The process of {` corresponds to the subcommand corresponding to `[SCRIPT]`.

```
pub fn flags_from_vec(
  args: Vec<String>,
) -> (DenoFlags, DenoSubcommand, Vec<String>) {
  let cli_app = create_cli_app();
  let matches = cli_app.get_matches_from(args);
  // ...
```

[*6] https://github.com/clap-rs/clap

```
  let subcommand = match matches.subcommand() {
    // ...
    (script, Some(script_match)) => {
      // ...
      DenoSubcommand::Run
    }
    // ...
  };

  (flags, subcommand, argv)
}
```

Because `DenoSubcommand::Run` is evaluated at the end of the block, the value of the variable `subcommand` is `DenoSubcommand::Run`.

5.3.2 Subcommand execution

We read the command line argument parsing by the `flags_from_vec` function, so I'll return the focus to the main function. When the command line argument is parsed, the parsed V8 flag is set and the function corresponding to the subcommand is called. The value of `subcommand` is `DenoSubcommand::Run` from the pattern matching result of `flags_from_vec` that I read earlier, so the next function to be called is `run_script`.

```
fn run_script(flags: DenoFlags, argv: Vec<String>) {
  // ...
  let (mut worker, state) = create_worker_and_state(flags, argv);

  let main_module = state.main_module().unwrap();
  // Normal situation of executing a module.
  let main_future = lazy(move || {
    // Setup runtime.
    js_check(worker.execute("denoMain()"));
    debug!("main_module {}", main_module);

    worker
      .execute_mod_async(&main_module, false)
      .and_then(move |()| {
        // ...
      })
      .map_err(print_err_and_exit)
  });

  // ...
  } else {
    tokio_util::run(main_future);
  }
}
```

If you understand what is happening in this function, you can understand the process from the Deno process starts to TypeScript can be executed. In other words, the rest of this chapter is almost entirely about the inside of `run_script`. Let's look at them in turn.

First, create worker and state with `create_worker_and_state` function. Next, excute a process with `tokio_util::run` asynchronously. The process is described "Execute JavaScript code (`denoMain()`) for the worker, and when setup is complete, execute TypeScript code at `worker.execute_mod_async`. If there is no error, the `load` event is issued to the window object, and the process ends. If there is an error, the process is terminated with an error output". We found a code that ends the process. Next, we will understand the contents of each code. We will focus on the three important parts: worker + state generation, denoMain function execution, and `main_module` execution.

Column: The tips for code reading is a clear purpose and completing it all the way.

If you read the code so far, you may have noticed that the code reading in this chapter is only a small part of deno. For example, there were 10 sub-commands (`Bundle`, `Completions`, `Eval`, `Fetch`, `Info`, `Install`, `Repl`, `Types`, `Version`, `Xeval`) other than run. There was also a considerable amount of code for parsing command line arguments.

It is my own idea that it's not effective that you read code shallowlly and widely or just see the code from top to bottom without any purpose. I think that the understanding of the code will increase by having some purpose, repeatedly selecting and concentrating code so as not to read code that does not meet the purpose, and structurally grasps the flow of the code. The purpose of this chapter is to "follow the flow from when the `deno https://deno.land/welcome.ts` command is executed until the Deno process starts and the TypeScript code is executed" . so, I don't explain other processing.

5.4 create_worker_and_state

We continue reading about the `create_worker_and_state` function that creates the worker and state. The `create_worker_and_state` function is defined in cli/main.rs. The rough process creates an instance of `ThreadSafeState` and uses it to create an instance of `Worker`.

```
fn create_worker_and_state(
  flags: DenoFlags,
  argv: Vec<String>,
) -> (Worker, ThreadSafeState) {
  // ...
  let state = ThreadSafeState::new(flags, argv, progress, true)
    .map_err(print_err_and_exit)
    .unwrap();
  let worker = Worker::new(
    "main".to_string(),
    startup_data::deno_isolate_init(),
    state.clone(),
  );

  (worker, state)
}
```

For ThreadSafeState, we skip now. Since it will appear in the process of loading the worker process, I will explain there. Before reading the implementation of `Worker`, let's read in detail about `startup_data::deno_isolate_init()` passed to the argument of `Worker::new`. `deno_isolate_init` is defined in `cli/startup_data.rs`. `#[cfg(...)]` attribute[*7] is used to devide which function definition is used when transpiling.

```
#[cfg(feature = "no-snapshot-init")]
pub fn deno_isolate_init() -> StartupData<'static> {
  debug!("Deno isolate init without snapshots.");
  // ...
```

[*7] https://doc.rust-lang.org/1.0.0/book/conditional-compilation.html

```
}

#[cfg(not(feature = "no-snapshot-init"))]
pub fn deno_isolate_init() -> StartupData<'static> {
  debug!("Deno isolate init with snapshots.");
  // ...
}
```

Which definition is used? This answer is quick to read from the debug log. We compare the log with the code, there is a log left where the second function is executed. In the debug log described at the beginning, `DEBUG RS-deno_bin::startup_data:24 - Deno isolate init with snapshots.` was output. It would be nice to read `deno_isolate_init` latter. Since we have decided which function should be read, we will follow the process inside. The main function of this function is to generate `StartupData::Snapshot`.

```
use deno::StartupData;
use deno_cli_snapshots::CLI_SNAPSHOT;
// ...
pub fn deno_isolate_init() -> StartupData<'static> {
  // ...
  let data = CLI_SNAPSHOT;
  // ...

  StartupData::Snapshot(data)
}
```

`StartupData` is an enum and is defined in `core/isolate.rs`. So what is @<tt {CLI_SNAPSHOT}? This constant is defined in `cli_snapshots/lib.rs`. `include_bytes` Macro[*8] is used to hold the contents of the file `CLI_SNAPSHOT.bin` as a byte array.

```
pub static CLI_SNAPSHOT: &[u8] =
  include_bytes!(concat!(env!("OUT_DIR"), "/CLI_SNAPSHOT.bin"));
```

[*8] https://doc.rust-lang.org/std/macro.include_bytes.html

It is replaced with the value of the environment variable `OUT_DIR` that is set when deno is transpiled using the `env` macro[*9]. In my environment, the files were placed under `target/rls/debug/build/deno_cli_snapshots-e4bac17f9b4c4252/out/`.

We read about `startup_data::deno_isolate_init()` passed to `Worker::new` up to here. So we go back to the original process and return to read through the implementation of `Worker::new`.

`Worker` is defined in <tt>{cli/worker.rs}.

```
impl Worker {
  pub fn new(
    _name: String,
    startup_data: StartupData,
    state: ThreadSafeState,
  ) -> Worker {
    let isolate = Arc::new(Mutex::new(deno::Isolate::new(startup_data, false)));
    {
      let mut i = isolate.lock().unwrap();

      let state_ = state.clone();
      i.set_dispatch(move |op_id, control_buf, zero_copy_buf| {
        state_.dispatch(op_id, control_buf, zero_copy_buf)
      });

      // ...
    }
    Self { isolate, state }
  }
```

The main process is to acquire an Isolate, call methods such as `set_dispatch`, register callbacks, and initialize the structure using the generated Isolate and the state passed as an argument.

First, let's see `Arc::new(Mutex::new(deno::Isolate::new(startup_data, false)))`. `Arc` and `Mutex`[*10] are used as thread-safe value management mechanisms in Rust multithreading. By combining Arc and Mutex, it is a container type that

[*9] https://doc.rust-lang.org/std/macro.env.html
[*10] https://doc.rust-jp.rs/the-rust-programming-language-ja/1.6/book/concurrency.html

can guarantee that the value can be changed only within one thread at any given time. See the footnote link for an explanation of Rust's parallelism. What is `deno::Isolate` passed to `Arc` and `Mutex`? In short, Isolate is an isolated execution context of JavaScript. See the Doc comment for the Isolate structure defined in `core/isolate.rs` for more details. Let's read the definition of `Isolate::New`.

```
impl Isolate {
  // ...
  pub fn new(startup_data: StartupData, will_snapshot: bool) -> Self {
    DENO_INIT.call_once(|| {
      unsafe { libdeno::deno_init() };
    });
    // ...
    let mut libdeno_config = libdeno::deno_config {
      will_snapshot: will_snapshot.into(),
      // ...
    };
    // ...
    match startup_data {
      // ...
      StartupData::Snapshot(d) => {
        libdeno_config.load_snapshot = d.into();
      }
      // ...
    };

    let libdeno_isolate = unsafe { libdeno::deno_new(libdeno_config) };

    Self {
      libdeno_isolate,
      // ...
    }
  }
```

`libdeno::deno_init` and `libdeno::deno_new` enclosed in unsafe blocks are called FFI (other language function interface[*11]). They are process of Rust that calls the function of C++. Other language functions are considered unsafe[*12] in Rust, so the calling code is enclosed in an unsafe block. Since there is too much text to explain everything, I will not explain the mechanism of FFI and the related things to the

[*11] https://doc.rust-jp.rs/the-rust-programming-language-ja/1.6/book/ffi.html
[*12] https://doc.rust-jp.rs/the-rust-programming-language-ja/1.6/book/unsafe.html

build. If you are interested, see the code about the build. `libdeno::deno_init`, `libdeno::deno_new` and other C++ function definitions that deno calls in FFI are basically defined in `core/libdeno/api.cc`. With respect to `libdeno::deno_init`, I omitted the code because it just initializes each platform at runtime and V8 itself. `libdeno::deno_new` has important process, so I quoted the C++ code below.

```
Deno* deno_new(deno_config config) {
  // ...
  deno::DenoIsolate* d = new deno::DenoIsolate(config);
  v8::Isolate::CreateParams params;
  // ...
  if (config.load_snapshot.data_ptr) {
    params.snapshot_blob = &d->snapshot_;
  }
  v8::Isolate* isolate = v8::Isolate::New(params);
  d->AddIsolate(isolate);
  // ...
}
```

The main process is to create a V8 Isolate and set it to Deno Isolate. When V8 Isolate option `params.snapshot_blob` is set, it functions as a V8 snapshot. V8 snapshot is a technology that saves the state of memory expanded in the heap as a result of executing JavaScript on V8 as binary, reads the binary directly and expands it directly to the heap. In addition to reading JavaScript built-in objects such as Date and Math, it is also possible to snapshot your own defined JavaScript code. Using V8 snapshot can greatly reduce the initial startup overhead such as syntax analysis. The binary data passed to this snapshot is passed the value of `CLI_SNAPSHOT` read earlier. I will explain what is in this binary later.

5.5 Execution of denoMain function

We read about `create_worker_and_state`, so next I will read about `js_check(worker.execute("de`
One thing to keep in mind before reading the code is that the code `denoMain()` is JavaScript, not TypeScript. Workers and Isolates don't have the ability to run TypeScript and can only run JavaScript. The process of transpiling TypeScript will come later. First we skip reading the denoMain function, we read the mechanism that executes JavaScript. After that, we will read the denoMain function.

First, list the functions and methods that are mainly appeared in `worker.execute` unntil JavaScript is executed. I will explain along this flow.

1. Worker#execute (cli/worker.rs L61)
2. Worker#execute2 (cli/worker.rs L69)
3. Isolate#execute (core/isolate.rs L373)
4. libdeno::deno_execute (core/libdeno.rs L276)
5. Call a C++ function with FFI
6. deno_execute (core/libdeno/api.cc L144)
7. Execute (core/libdeno/binding.cc L333)

`Worker#execute` and `Worker#execute2` only transfer JavaScript execution to `Isolate#execute`, and there are only 2-3 lines of processing each. So I will omit it and we read the definition of `Isolate#execute`.

```
pub fn execute(
  &mut self,
  js_filename: &str,
  js_source: &str,
) -> Result<(), ErrBox> {
  self.shared_init();
  let filename = CString::new(js_filename).unwrap();
  let source = CString::new(js_source).unwrap();
  unsafe {
    libdeno::deno_execute(
      self.libdeno_isolate,
      self.as_raw_ptr(),
      filename.as_ptr(),
      source.as_ptr(),
    )
  };
  self.check_last_exception()
}
```

The main process calls `libdeno::deno_execute`. FFI also appeared again. `CString`[*13] is a structure for generating a C-compatible string.

Next, we read the C ++ layer code such as `deno_execute` and `Execute` but they are not specific to Deno. If I explained them in detail, this chapter seemed how to use

[*13] https://doc.rust-lang.org/std/ffi/struct.CString.html

V8. So, I describes them roughly as "Run JavaScript in V8 + Doing things necessary for it" and treat it as a black box in this chapter.

5.6 Generate V8 snapshot

We read the mechanism that JavaScript code `denoMain()` can be executed. Let's read what happens when the denoMain function is called and where the denoMain function came from in the first place. There is no JavaScript code other than `denoMain()`, including parts omitted so far. In other words, JavaScript that defines the denoMain function is not executed. However, denoMain() that is JavaScript actually works. It is snapshot (byte array stored in constant `CLI_SNAPSHOT`) passed when initializing V8::Isolate that realizes this strange behavior. The denoMain function can be defined by saving and loading a snapshot of Isolate that has been executed JavaScript in advance. So we can understand how the denoMain function was defined by figuring out what is stored in the constant `CLI_SNAPSHOT`.

Since creating snapshots is very easy to get lost, I write a guide. The process to read from now on is **the process that transform js, which was transpiled from js/main.ts to amd module with tsc, to the snapshot**. In addition, it will be less confusing if you keep in mind that this is happening in the build phase and happens in a different process than the one you've been reading. First, we list the main functions that appear in order.

1. `main` (`cli_snapshots/build.rs` L5)
2. `compile_bundle` (`deno_typescript/lib.rs` L84)
3. `TSIsolate::compile` (`deno_typescript/lib.rs` L74)
4. `deno_typescript::mksnapshot_bundle` (`deno_typescript/lib.rs` L141)
5. `write_snapshot` (`deno_typescript/lib.rs` L182)

First, the process of generating a snapshot from `cli_snapshots/build.rs` starts, creates a bundle called `CLI_SNAPSHOT.js`, and `write_snapshot` function create a file {CLI_SNAPSHOT.bin} and the contents of this file are stored in the constant `CLI_SNAPSHOT`. Let's look at them in order.

```
fn main() {
  // ...
  let c = PathBuf::from(env::var_os("CARGO_MANIFEST_DIR").unwrap());
  let o = PathBuf::from(env::var_os("OUT_DIR").unwrap());
  let js_dir = c.join("../js");

  let root_names = vec![js_dir.join("main.ts")];
  let bundle = o.join("CLI_SNAPSHOT.js");
  let state = deno_typescript::compile_bundle(&bundle, root_names).unwrap();
  assert!(bundle.exists());
  deno_typescript::mksnapshot_bundle(&bundle, state).unwrap();
  // ...
}
```

The most important of these are `deno_typescript::compile_bundle` and `deno_typescript::mksnapshot_bundle`. For `compile_bundle`, specify the path for generating bundled files and the list of TS files that are entry points. As in the main function, `js/main.ts` is the entry point. `denoMain` function is defined in `js/main.ts` and is exposed globally as `window["denoMain"] = denoMain`. From this, we can understand that the JavaScript `denoMain()` can be executed after loading this JS.

The variable `config_json` defined inside `compile_bundle` has a value equivalent to tsconfig.json. Of particular note is module = amd. In other words, JavaScript generated when transpiling with this setting will be in AMD format. Then we will follow the definition of `TSIsolate::compile` corresponding to `ts_isolate.compile` that creates the bundle.

```
fn compile(
  mut self,
  config_json: &serde_json::Value,
  root_names: Vec<String>,
) -> Result<Arc<Mutex<TSState>>, ErrBox> {
  let root_names_json = serde_json::json!(root_names).to_string();
  let source =
    &format!("main({:?}, {})", config_json.to_string(), root_names_json);
  self.isolate.execute("<anon>", source)?;
  Ok(self.state.clone())
}
```

Here `self.isolate` is another structure called TSIsolate, not Isolate, which we read earlier. JavaScript code `main(tsc settings, ["js/main.ts"])` is executed for TSIsolate. Where did the main function come from? Here is a look back at the initialization process of `TSIsolate`.

```
static TYPESCRIPT_CODE: &str =
  include_str!("../third_party/node_modules/typescript/lib/typescript.js");
static COMPILER_CODE: &str = include_str!("compiler_main.js");
// ...
impl TSIsolate {
  fn new(bundle: bool) -> TSIsolate {
    let mut isolate = Isolate::new(StartupData::None, false);
    js_check(isolate.execute("assets/typescript.js", TYPESCRIPT_CODE));
    js_check(isolate.execute("compiler_main.js", COMPILER_CODE));
    // ...
  }
```

When initializing the structure, create Isolate without snapshot and execute TypeScript and `compiler_main.js`. **Note that this Isolate is not for generating a snapshot, but for running a TypeScript transpile to generate JavaScript for creating a snapshot.** Please be careful because it is very confusing. The `main` function is defined in `compiler_main.js`. In this file, TypeScript transpile is executed to create AMD bundle.

Now that we have a bundle for taking snapshots, return to the main function (in Rust) and read the `deno_typescript::mksnapshot_bundle` function. The variable `runtime_isolate` is an Isolate for generating a snapshot. A clean Isolate is generated, `amd_runtime.js` and the generated bundle are executed, and the result of executing the `require` statement to call the bundle is made into a snapshot.

`write_snapshot` that is called in `deno_typescript::mksnapshot_bundle` is a function that actually retrieves and saves a snapshot. The function acquire a snapshot from V8 Isolate and save in a file called `CLI_SNAPSHOT.bin`.

From the above, `OUT_DIR/CLI_SNAPSHOT.bin` was generated, and it was found that the denoMain function can be called by restoring the snapshot.

5.7 execute_mod_async

We have read a long long pre-processing. The topic has returned from the build-time process to the run-time process, and we start to read the process of executing TypeScript code given by the user. Finally, we will read about `worker.execute_mod_async(& main_module, false)`. First, what is the value of `main_module`? Let’s read back the debug log.

```
DEBUG RS - deno_bin:379 - main_module https://deno.land/welcome.ts
```

As shown in the debug log output, the URL `https://deno.land/welcome.ts` is stored in `main_module`. More precisely, it stores Deno’s own structure that extends the Url `ModuleSpecifier`.

```
pub fn execute_mod_async(
  &mut self,
  module_specifier: &ModuleSpecifier,
  is_prefetch: bool,
) -> impl Future<Item = (), Error = ErrBox> {
  let worker = self.clone();
  let loader = self.state.clone();
  let isolate = self.isolate.clone();
  let modules = self.state.modules.clone();
  let recursive_load =
    RecursiveLoad::main(&module_specifier.to_string(), loader, modules)
      .get_future(isolate);
  recursive_load.and_then(move |id| -> Result<(), ErrBox> {
    worker.state.progress.done();
    if is_prefetch {
      // ...
    } else {
      let mut isolate = worker.isolate.lock().unwrap();
      isolate.mod_evaluate(id)
    }
  })
}
```

`RecursiveLoad::main(...). get_future(isolate)` is the dependency resolution process, and executed with specified the module ID (`id`) generated after the dependency resolution is finished. Since false was passed to the provisinal argument `is_prefetch`, `isolate.mod_evaluate` is executed. First read `RecursiveLoad::main(...).Get_future(isolate)` for dependency resolution. The definition is in `core/modules.rs`. `RecursiveLoad::main` is like a factory method for calling new, new just initializes the structure.

```
pub fn main(
  specifier: &str,
  loader: L,
  modules: Arc<Mutex<Modules>>,
) -> Self {
  let kind = Kind::Main;
  let state = State::ResolveMain(specifier.to_owned());
  Self::new(kind, state, loader, modules)
}
```

Then, we read `get_future`.

```
pub fn get_future(
  self,
  isolate: Arc<Mutex<Isolate>>,
) -> impl Future<Item = deno_mod, Error = ErrBox> {
  loop_fn(self, move |load| {
    let isolate = isolate.clone();
    load.into_future().map_err(|(e, _)| e).and_then(
      move |(event, mut load)| {
        Ok(match event.unwrap() {
          Event::Fetch(info) => {
            let mut isolate = isolate.lock().unwrap();
            load.register(info, &mut isolate)?;
            Loop::Continue(load)
          }
          Event::Instantiate(id) => Loop::Break(id),
        })
      },
    )
  })
}
```

This process requires an understanding of Rust and Future. Even if I read only this code, I could not read the code flow at all. The key to read is the `futures::stream::Stream` interface implemented in the same file, and a method called poll has been created. This is the key. Asynchronous processing such as Tokio and Future is covered in detail in the other chapter “Tokio and Deno” , so please read it.

```
impl<L: Loader> Stream for RecursiveLoad<L> {
  type Item = Event;
  type Error = ErrBox;

  fn poll(&mut self) -> Poll<Option<Self::Item>, Self::Error> {
    Ok(match self.state {
      State::ResolveMain(..) | State::ResolveImport(..) => {
        self.add_root()?;
        self.poll()?
      }
      State::LoadingRoot | State::LoadingImports(..) => {
        match self.pending.poll()? {
          Ready(None) => unreachable!(),
          Ready(Some(info)) => Ready(Some(Event::Fetch(info))),
          NotReady => NotReady,
        }
      }
      State::Instantiated(id) => Ready(Some(Event::Instantiate(id))),
    })
  }
}
```

The process is complicated and mixeddd with polling and event handling. It looks like following flow when I arrange in order. In this process, the process from getting TypeScript code from URL (`https://deno.land/welcome.ts`) to transpiling to JavaScript is described.

1. state = `State::ResolveMain` * `RecursiveLoad::add_root` is called and polled again 1. State = `State::LoadingRoot` * Poll until `Ready(Some(info))` is returned * Trigger `Event::Fetch(info)` event 1. event = `Event::Fetch(info)` * `RecursiveLoad::register` is called to resume polling 1. state = `State::Instantiated(id)` * `Event::Fetch(info)` event triggered 1. event = `Event::Instantiate(id)` * End the loop

Since the state when initializing the structure with the main function was `State::ResolveMain`, it matches state = `State::ResolveMain` and `add_root` is called, state changes to `State::LoadingRoot`. State changes are expressed by events and polling as in "...". First, we read the definition of `add_root`.

```
fn add_root(&mut self) -> Result<(), ErrBox> {
  let module_specifier = match self.state {
    State::ResolveMain(ref specifier) => self.loader.resolve(
      specifier,
      ".",
      true,
      self.dyn_import_id().is_some(),
    )?,
    // ...
  };
  // ...
  self
    .pending
    .push(Box::new(self.loader.load(&module_specifier)));
  self.state = State::LoadingRoot;

  Ok(())
}
```

`self.loader.load` corresponds to `ThreadSafeState::load` in `cli/state.rs`. `ThreadSafeState::load` internally calls `fetch_compiled_module`.

```
pub fn fetch_compiled_module(
  self: &Self,
  module_specifier: &ModuleSpecifier,
) -> impl Future<Item = CompiledModule, Error = ErrBox> {
  let state_ = self.clone();

  self
    .file_fetcher
    .fetch_source_file_async(&module_specifier)
    .and_then(move |out| match out.media_type {
      msg::MediaType::Unknown => {
        // ...
      }
      msg::MediaType::Json => {
        // ...
      }
      msg::MediaType::TypeScript => {
        state_.ts_compiler.compile_async(state_.clone(), &out)
      }
      msg::MediaType::JavaScript => {
        // ...
      }
    })
}
```

The pattern match between `self.file_fetcher.fetch_source_file_async` and its return value seems the main. TypeScript transpiling is being performed when the file is retrieved and the media type is `msg::MediaType::TypeScript`. During `fetch_source_file_async`, if the URL protocol is `file://`, call `fetch_local_file`, otherwise (`https?://`) `fetch_remote_source_async` is called to retrieve the contents of the file. Media type resolution is determined from the HTTP response header Content-Type and the extension at the end of the URL (file path). I omitted this area because they are not complicated and there are large amount of code. In this chapter, we read the process after the contents of the file corresponding to `https://deno.land/welcome.ts` are acquired and the media type is determined to be `msg::MediaType::TypeScript`.

The next step is to transpile TypeScript and convert it to JavaScript. `state_.ts_compiler.compile_async`. It is defined in `cli/compilers /ts.rs`.

```
pub fn compile_async(
  self: &Self,
  state: ThreadSafeState,
  source_file: &SourceFile,
) -> Box<CompiledModuleFuture> {
  // ...
  debug!(">>>>> compile_sync START");
  // ...
  let root_names = vec![module_url.to_string()];
  let req_msg = req(root_names, self.config.clone(), None);
  // ...
  let worker = TsCompiler::setup_worker(state.clone());
  let resource = worker.state.resource.clone();
  let compiler_rid = resource.rid;
  let first_msg_fut =
    resources::post_message_to_worker(compiler_rid, req_msg)
      .then(move |_| worker)
      .then(move |result| {
        // ...
        debug!("Sent message to worker");
        let stream_future =
          resources::get_message_stream_from_worker(compiler_rid)
            .into_future();
        stream_future.map(|(f, _rest)| f).map_err(|(f, _rest)| f)
      });
  // ...
  Box::new(fut)
}
```

Create a message to be sent to the worker with `req` function, initialize Isolate for transpiling with `TsCompiler::setup_worker`, and send the message to that worker. The definition of `TsCompiler::setup_worker` is like following. As with `CLI_SNAPSHOT`, this Isolate initialization uses Isolate snapshot `COMPILER_SNAPSHOT.bin` to perform TypeScript transpiling. The process of creating this snapshot is omitted in this chapter, but it is written in a set with the process of generating a snapshot at runtime. As a outline, it becomes a JavaScript snapshot bundled in AMD format with `js/compiler.ts` as the entry point. Therefore, three JavaScript functions of `denoMain()`, `workerMain()`, `compilerMain()` can be executed.

```
fn setup_worker(state: ThreadSafeState) -> Worker {
  // Count how many times we start the compiler worker.
  state.metrics.compiler_starts.fetch_add(1, Ordering::SeqCst);

  let mut worker = Worker::new(
    "TS".to_string(),
    startup_data::compiler_isolate_init(),
    // TODO(ry) Maybe we should use a separate state for the compiler.
    // as was done previously.
    state.clone(),
  );
  worker.execute("denoMain()").unwrap();
  worker.execute("workerMain()").unwrap();
  worker.execute("compilerMain()").unwrap();
  worker
}
```

`post_message_to_worker` uses `WorkerSender#send` to send and receive messages from Rust to C ++ (V8) and TypeScript. For details about this, please refer to my blog post before the revision. The name exposed on the js side has been changed from `libdeno` to `Deno.core`, but the mechanism is the same.

> send はここに定義されています。src/isolate.rs の中で deno_new 関数の中で渡している deno_config の recv_cb が DenoIsolate に渡っており、結果的に pre}dispatch という Rust の関数が呼び出されます。
> Translator note: send is defined here. The deno_config recv_cb passed in the deno_new function in src / isolate.rs is passed to DenoIsolate, and as a result, the Rust function pre_dispatch is called.
> `https://blog.leko.jp/post/code-reading-of-deno-boot-process/#補足-libdeno オブジェクト`

The contents of the `compilerMain` function in `js/compiler.ts` are transpiled to js with a TypeScript transpiler for Deno that implements the TypeScript `LanguageServerHost` interface. The `LanguageServerHost` interface is cleanly decoupled so that it works in both browsers and Node.js, so it's interesting we can re-implement it as an interface that works with the same JavaScript runtime, Deno.

5.7.1 `isolate.mod_evaluate`

We manage to reach the final processing of the `execute_mod_async` function. We will read `isolate.mod_evaluate`.

```
pub fn mod_evaluate(&mut self, id: deno_mod) -> Result<(), ErrBox> {
  // ...
  unsafe {
    libdeno::deno_mod_evaluate(self.libdeno_isolate, self.as_raw_ptr(), id)
  };
  // ...
}
```

The FFI appears again. Corresponding C ++ functions are defined in `core/libdeno/modules.cc`. In short, running transpiled JavaScript against Isolate.

```
extern "C" {
// ...
void deno_mod_evaluate(Deno* d_, void* user_data, deno_mod id) {
  auto* d = unwrap(d_);
  // ...
  auto* info = d->GetModuleInfo(id);
  auto module = info->handle.Get(isolate);
  auto status = module->GetStatus();

  if (status == Module::kInstantiated) {
    bool ok = !module->Evaluate(context).IsEmpty();
    // ...
  }
  // ...
}
}
```

The last one is very simple. The above is the process until Deno transpiles TypeScript and executes it as JavaScript.

5.8 Afterword

In this chapter, I introduced the map and way for reading code deeply. Now, I look back at this chapter and paste the result of running deno's welcome.ts.

```
$ ./target/debug/deno https://deno.land/welcome.ts
Download https://deno.land/welcome.ts
Compile https://deno.land/welcome.ts
Welcome to Deno
```

The output is only 3 lines. But if you have a concrete image of what the process is doing while the process is from starting to ending, it's to say that you got something from this chapter. welcome.ts is a very simple file and is very suitable for explaining the minimum mechanism. But real-world programs are more complex. For example, what if you import another file? How to resolve URL when it was imported dynamically? How does the security mechanism work when performing file I/O? There are a lot of important features of deno that I have not been introduced this time. I hope you would be more fun to read the parts that were not mentioned in this chapter.

Appendix

About the authors

Chapter 1 Getting Started in Deno Programming / kt3k (Yoshiya Hinosawa)

A freelance web engineer. Working recently at a startup, SEQSENSE, which creates the security robot SQ-2. A co-maintainer of C3.js, and a contributor of Deno.

`Twitter: @kt3k / GitHub: @kt3k`

Chapter 2 Creating a command line tool with Deno / syumai

I was a former front-end engineer and developed mainly React Native / Vue.js. Currently, I only write Go at work, so I write a TypeScript at the time of my hobby.

`Twitter: @__syumai / GitHub: @syumai`

Chapter 3 Bundler for Deno / keroxp

I bought two of Spitz[*14]'s best albums at night when I couldn't sleep, TS Bandler was made. I also write the blog https://scrapbox.io/keroxp.

`Twitter: @keroxp / GitHub: @keroxp`

Chapter 4 Tokio and Deno / sasurau4

I'm Web engineer of writing React, React Native, TypeScript with Vim keybinded VSCode. Since I wanted to write Rust, I was attracted to Deno.

`Twitter: @sasurau4 / GitHub: @sasurau4`

[*14] Additional note for English ver: Spitz is very famous rock band group in Japan. https://spitz-web.com

Chapter 5 Deep dive into Deno compiler / Leko

At work, I make medical devices using React, React Native, Node.js and TypeScript. I like reading OSS code. I'm a collaborator of Node.js, textlint and Gatsby in GitHub.
`Twitter: @L_e_k_o / GitHub: @Leko / Blog: https://blog.leko.jp`

Cover illustration / hashrock

I'm programmer who loves SVG and Vue.js very much. My hoby is drawing illustration, and I was also in charge of the cover this time again. I've made strange web app such as "anydown" and "mangadown" that convert Markdown into diagram and other things.
`Twitter: @hashedrock / GitHub: @hashrock`

Links

The example source codes for this book

https://scrapbox.io/deno-ja/Denobook_02

deno-ja Scrapbox

https://scrapbox.io/deno-ja/

deno-ja Slack

https://deno-ja-slackin.herokuapp.com/

The official website of Deno

https://deno.land/

deno_std (The standard modules of deno)

https://github.com/denoland/deno_std/

awesome-deno

https://github.com/denolib/awesome-deno/

Denobook 02 English ver.

published by Sep. 22, 2019 first edition1 impression

Author kt3k, syumai, keroxp, sasurau4, Leko, hashrock

(C) 2019 deno-ja