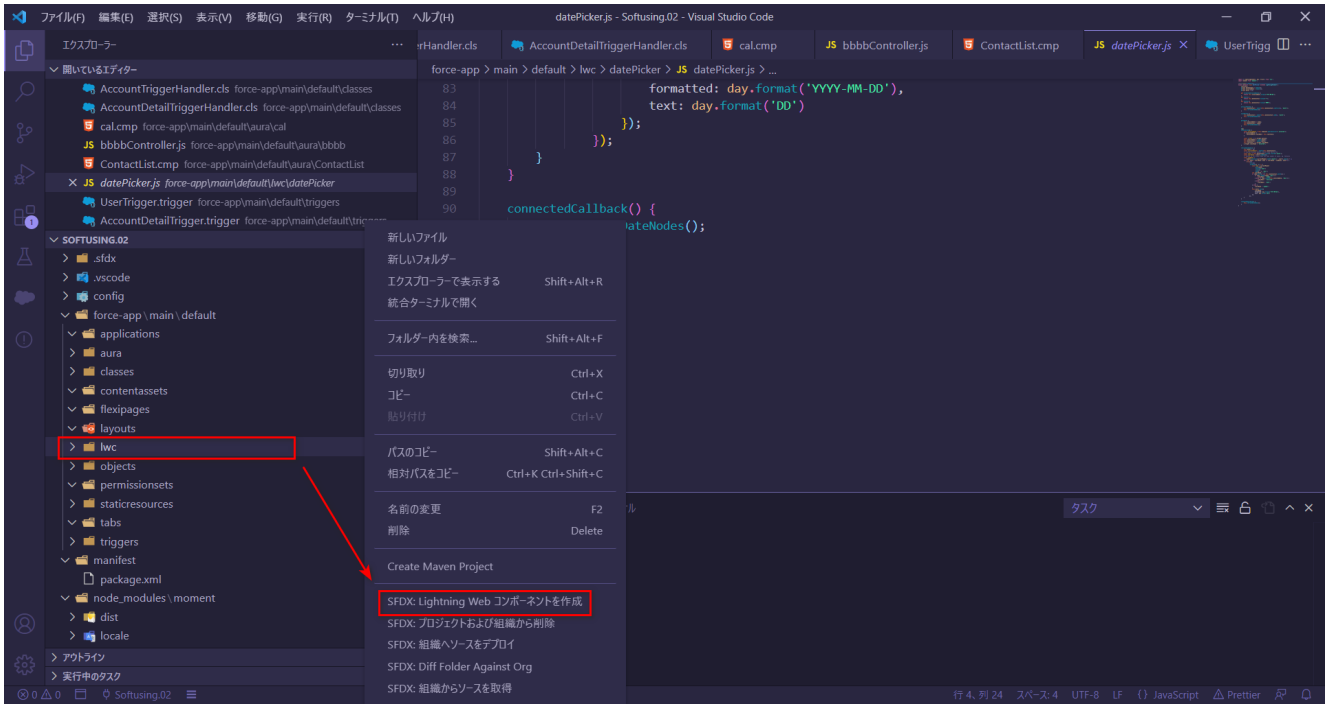
1.LWC是什么？

- LWC全称是Lightning Web Components，Salesforce最新的前端框架
- Salesforce前端发展：Visualforce ⇒ Aura(Lightning Components) ⇒ LWC

2.第一个lwc应用

:angel: 误解：
我在初学lwc的时候，看的教程都会先说sfdx，sfdx一堆的命令，然后还需要设置Devhub，然后还要创建Domain,
给我造成很多困扰，让我一度觉得lwc这东西很复杂。
我今天要告诉大家的是，不需要，只要有一个连接好开发环境的vscode，就可以完成lwc应用。

2.1 创建lwc



创建之后会生成3个文件 (html，js，xml)，并且开头字母会变成小写

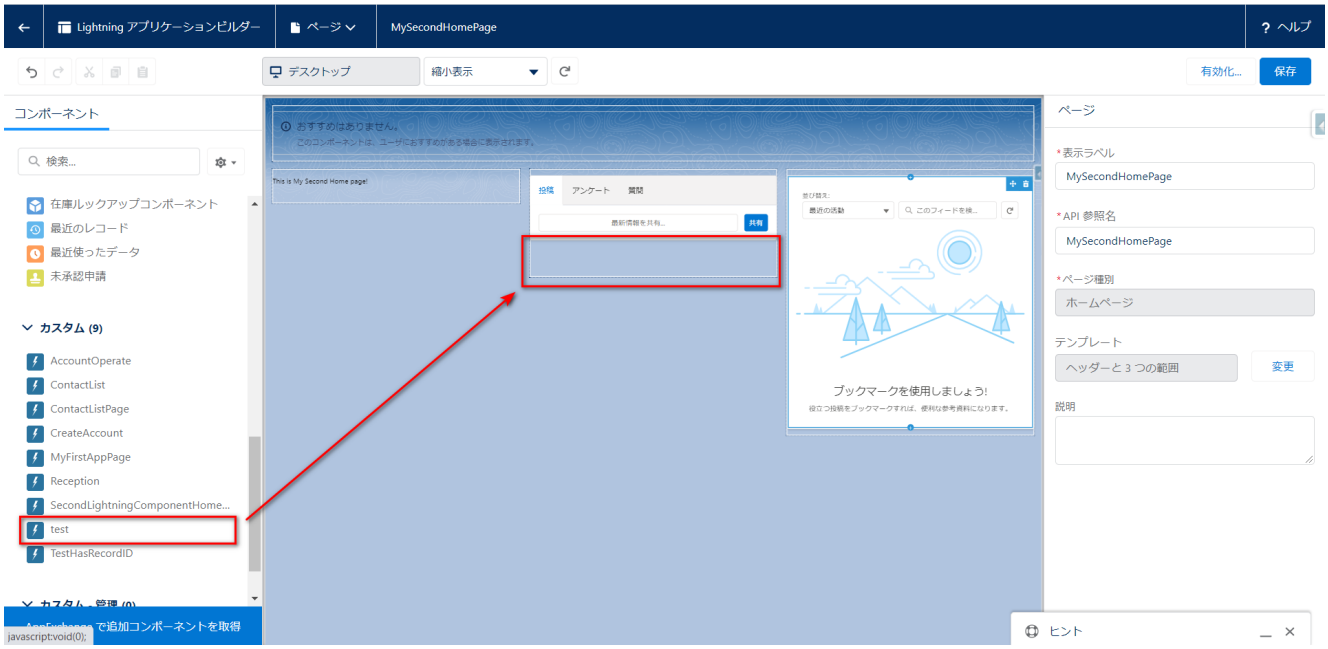
2.2 编写lwc

```
<template>
    <p>Hello LWC!</p>
</template>

import { LightningElement } from 'lwc';
export default class Test extends LightningElement {}
// import { LightningElement, api, wire, track } from 'lwc';
// export default class FAQInApplicationForm extends NavigationMixin(LightningElement) {
```

2.3 暴露lwc

```
<?xml version="1.0" encoding="UTF-8"?>
<LightningComponentBundle xmlns="http://soap.sforce.com/2006/04/metadata">
    <apiVersion>52.0</apiVersion>
    <isExposed>true</isExposed>
    <targets>
        <target>lightning__AppPage</target>
        <target>lightning__HomePage</target>
        <target>lightning__RecordPage</target>
    </targets>
</LightningComponentBundle>
```

2.4 設定lwc



2.5 删除lwc

首先要在页面编辑中删除LWC组件，然后才能在vscode中删除lwc应用。
:star: 所有的操作都要在vscode中完成，开发者console无法开发lwc应用

3.详解Import和Export

3.1 Salesforce中的导入方式

:scroll: Salesforce导入方式总结：

3.1.1 Visualforce中

- Visualforce中导入外部包：

```
<script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js"></script>
```

- Visualforce中导入其他组件：

```
<c:组件名 .....>
```

3.1.2 Lightning Components中

- aura中导入外部包 ：
  官方文档[1]
- aura中导入其他组件 ：

```
<c:组件名 .....>
```

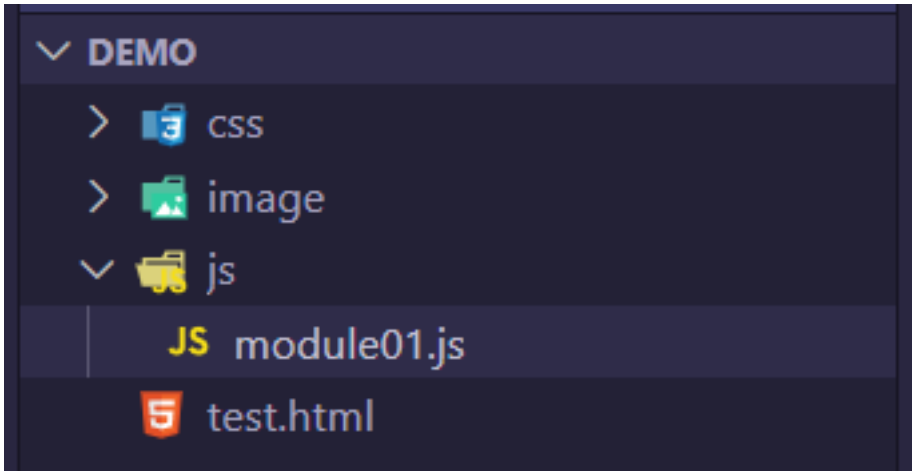
3.2 前端框架中的使用

当看到import和export 这两个关键字的时候，就会想到这个是ES6或者其他前端框架里面的模块化的概念。
要想把这两句话解释清楚，我们就先来看看，在ES6中的使用方式。

3.2.1 安装live server插件

- 无需刷新页面
- 直接右键「open with live server」

3.2.2 前端工程构建



3.2.3 创建js文件「module01」

```
module01.js

let x = 1;
let y = 2;
let z = 3;
export { x, y, z };
```

3.2.4 创建html文件「test」

:star2: 在test.html中输入「!」,就会出现提示。

```
test.html
```

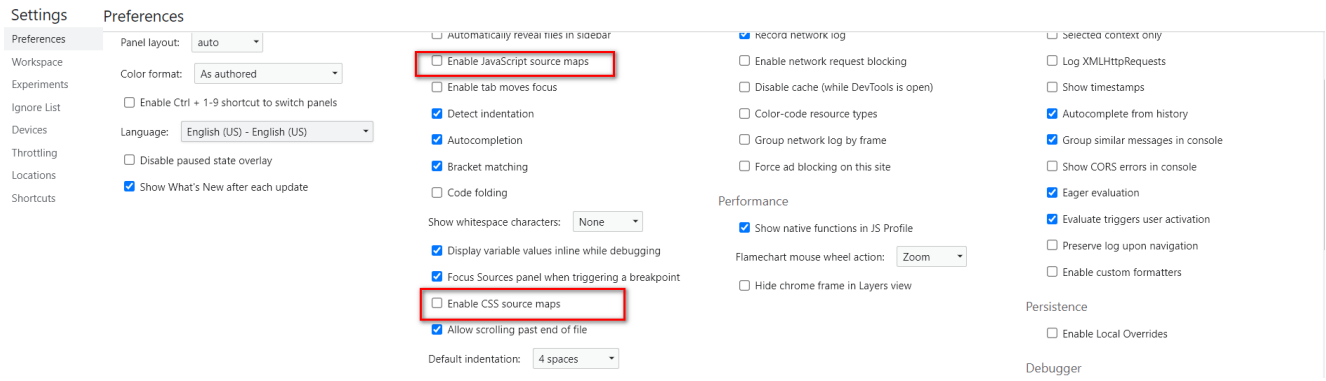
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script type="module">
        import {x, y, z} from './js/module01.js';

        console.log(x);
        console.log(y);
        console.log(z);
    </script>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script type="module">
        import {x} from './js/module01.js';
        import {y} from './js/module01.js';
        import {z} from './js/module01.js';
        console.log(x);
        console.log(y);
        console.log(z);
    </script>
</body>
</html>
```

:collision: 如果运行时出现下列警告：

```
DevTools failed to load SourceMap: Could not load content for…
```

首先，不影响正常运行，可以无视。
其次，如果看着不爽，去掉下面的勾选。



3.2.5 修改js文件「module01」

```
let x = 1;
let y = 2;
let z = 3;
function fn1() {
    console.log('fuction fn1');
}
let person = {
    name: 'zhang san',
    age: 20,
    say: function () {
        console.log(this.name + ':' + this.age);
    }
}
let pet = {
    name: 'xiao ming',
    age: 1,
    shout: function () {
        console.log(this.name + '-' + this.age);
    }
}
export { x, y, z, fn1, person, pet };
```

3.2.6 修改html文件「test」

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script type="module">
        import {x} from './js/module01.js';
        import {y} from './js/module01.js';
        import {z,fn1,person,pet} from './js/module01.js';
        console.log(x);
        console.log(y);
        console.log(z);
        fn1();
        person.say();
        pet.shout();

    </script>
</body>
</html>
```

3.2.7 导入多个文件

module02.js

```
let m = 100;
let n = 200;
export {m, n}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script type="module">
        import {x} from './js/module01.js';
        import {y} from './js/module01.js';
        import {z,fn1,person,pet} from './js/module01.js';
        import {n} from './js/module02.js';
        console.log(x);
        console.log(y);
        console.log(z);
        fn1();
        person.say();
        pet.shout();
        console.log(n);
    </script>
</body>
</html>
```

3.2.8 设定别名

- js文件中设定别名

```
let x = 100;
let n = 200;
function fn1() {
    console.log('fuction fn2');
}
export {x as x1, n, fn1 as fn2}

import {x1, fn2} from './js/module02.js';
console.log(x1);
fn2();
```

- html文件中设定别名

```
import {x as x1, fn1 as fn2} from './js/module02.js';
console.log(x1);
fn2();
```

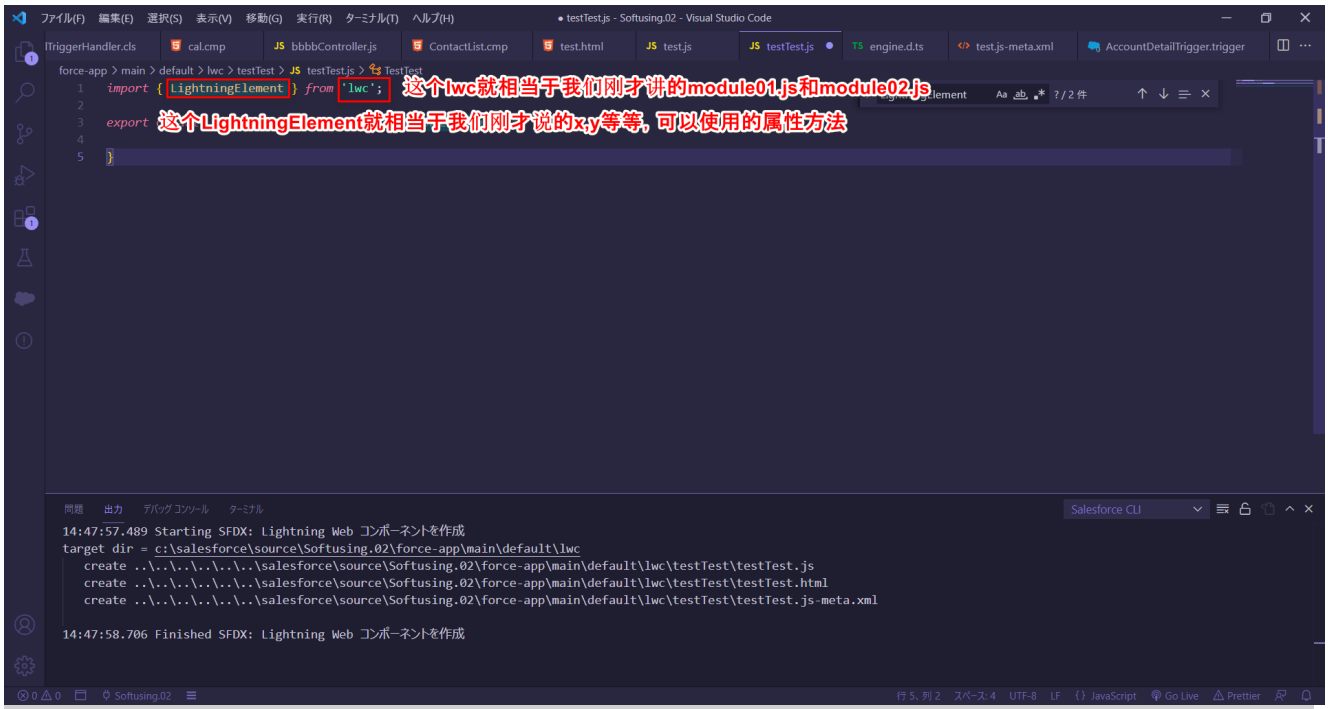
3.2.9 export default

```
export default pet;
export { x, y, z, fn1, person };
```

```
import pet from './js/module01.js';
pet.shout();

import aaa from './js/module01.js';
aaa .shout();
```

3.2.10 lwc 框架分析一

```
import { LightningElement } from 'lwc';
export default class TestTest extends LightningElement {

}
```




```
export class LightningElement extends HTMLElementTheGoodPart {
    static get CustomElementConstructor(): typeof HTMLElement;
    connectedCallback(): void;
    disconnectedCallback(): void;
    renderedCallback(): void;
    errorCallback(error: Error, stack: string): void;
    readonly template: ShadowRootTheGoodPart;
    readonly shadowRoot: null;
}
```

所以说下面的方法是可以使用的，就是在这里定义的
* connectedCallback(): void;
* disconnectedCallback(): void;
* renderedCallback(): void;
* errorCallback(error: Error, stack: string): void;
在来看看其他的export内容（因为export的内容是你可以使用的东西）

```
/**
 * Decorator to mark public reactive properties
 */
export const api: PropertyDecorator;
/**
 * Decorator to mark private reactive properties
 */
export const track: PropertyDecorator;
/**
 * Decorator factory to wire a property or method to a wire adapter data source
 * @param getType imperative accessor for the data source
 * @param config configuration object for the accessor
 */
export function wire(getType: (config?: any) => any, config?: any): PropertyDecorator;
```

api，track，wire也是会用到的

3.2.11 lwc 框架分析二

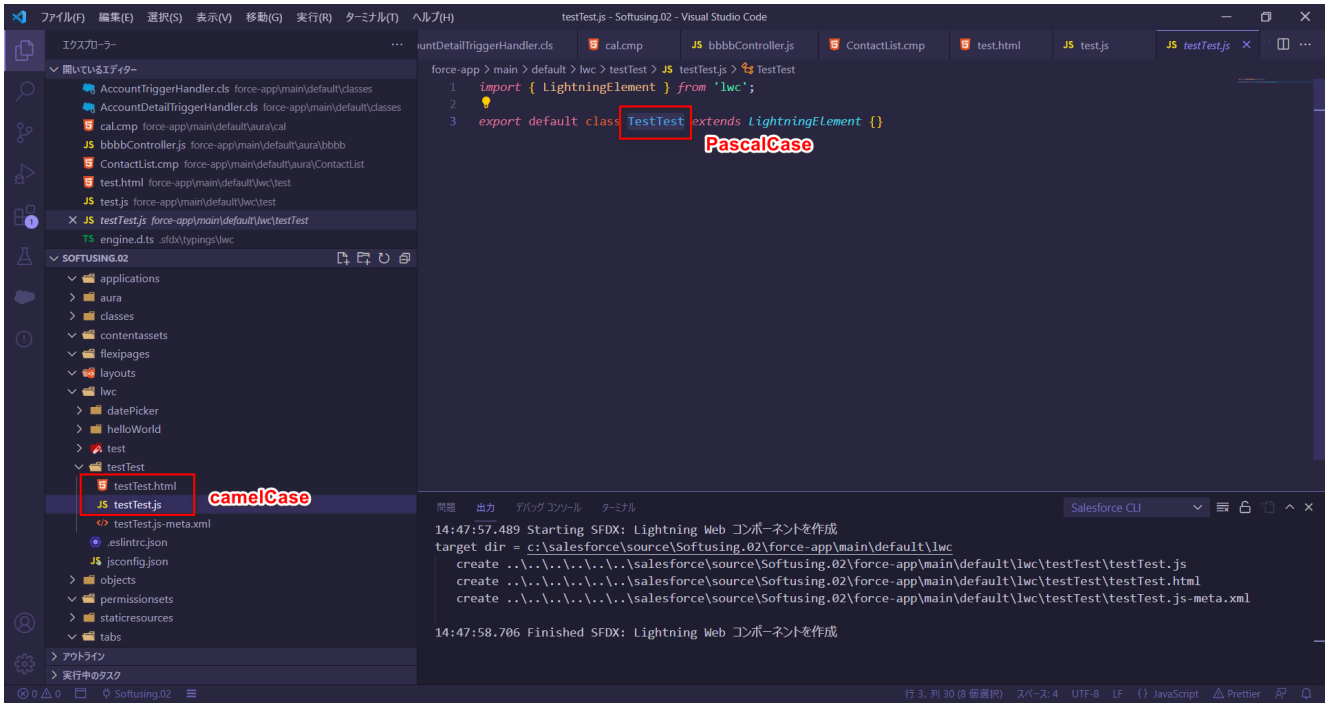
LWC 调用 Apex[2]
为什么是这种调用方式呢？大家有没有想过？
这个问题先放一下，咱们来看一下vscode里面的一个叫typings的文件夹下面有什么。
Typescript与typings
vscode的插件可以用Typescript和Javascript来写，但是大多是Typescript

```
typings的理解
一些js库扩展了JavaScript的特性和语法，但是TypeScript编译器并不识别，
通过typings.json配置可以辅助IDE，给出有智能的提示信息，以及重构的依据。
因此需要在typings.json文件中配置TypeScript类型定义文件（文件名后缀为.d.ts）
typings就方便TypeScript识别、编译、智能提示TypeScript无法识别的JS库的特性和语法
```

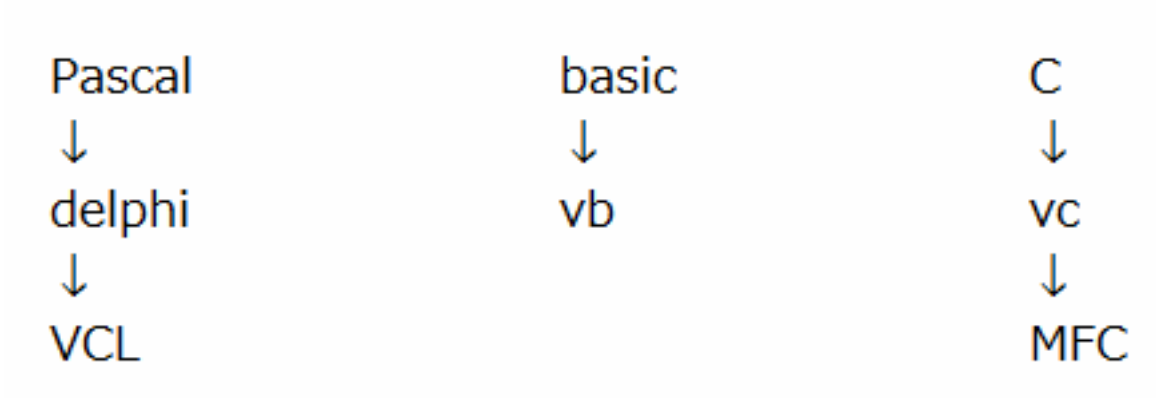
所以，salesforce的东西typescript不可能识别，所以他就借助了typings，来完成识别
所有会在vscode下面看到一个typings的文件夹，里面保存了很多.d.ts文件，会把salesforce的东西放在typings文件夹下面

4.命名規則

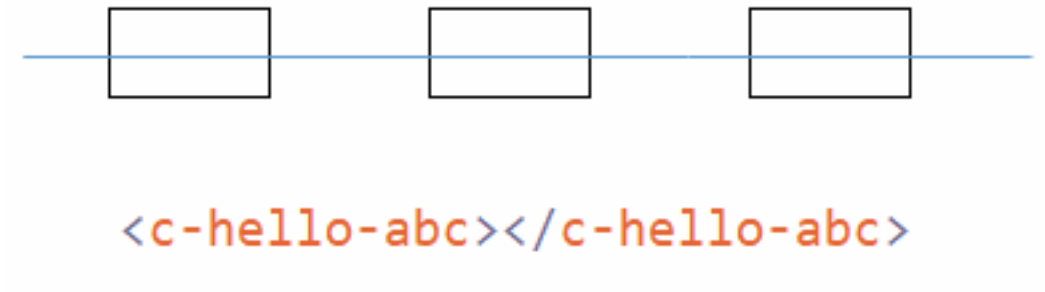
命名規則[3]
* PascalCase
* camelCase
* kebab-case



Pascal：



kebab翻译过来是烤肉串



:star2: 如果子组件是helloWorld的时候，在父组件中添加子组件的时候，要写成

```
<c-hello-world></c-hello-world>

<template>
    <c-hello-world></c-hello-world>
    <p>this is parent!</p>
</template>
```

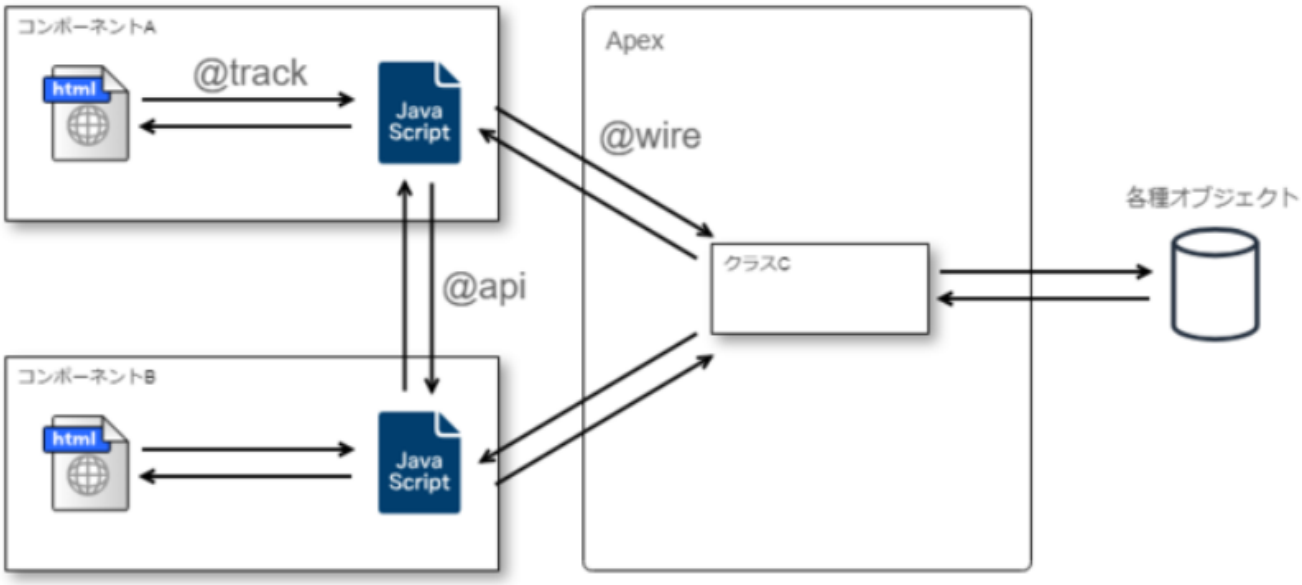
:angel: 思考题：如果子组件是ABCInApplicationForm的时候，父组件想导入子组件的时候应该怎么写？

```
<c-a-b-c-in-application-form123></c-a-b-c-in-application-form123>
```

5.Salesforce特有のデコレータ

デコレータ：Decorator

3つのデコレータとそれぞれの使い方[4]



5.1 @track

track 本身这个词就是追踪的意思，

private reactive properties

プロパティの値が変更されると、コンポーネントが再レンダリングされます。プロパティはプライベートなので、自コンポーネント内でのみアクセス可能です。（プライベートリアクティブプロパティ

如果一个属性的值被改变，该组件将被重新渲染。 由于该属性是私有的，它只能在自己的组件内被访问。

5.1.1 sample1

:clipboard: testTrack (用myName来追踪文本框的变化)

```
<template>
    hello {myName} !
    <lightning-input name="myName" label="enter the name:" onchange={changeName}></lightning-input>
</template>

import { LightningElement,track } from 'lwc';
export default class TestTrack extends LightningElement {
    @track myName;
    changeName(event) {
        this.myName = event.target.value;
    }
}
```

在这里有两点需要说一下：

1.变量直接用

2.用{}来绑定值

5.1.2 sample2

:clipboard: **testTrack (用selectedValue来追踪选择框的变化)**

```
<template>
    hello {myName} !
    <lightning-input name="myName" label="enter the name:" onchange={changeName}></lightning-input>
    <lightning-combobox name="Category" label="カテゴリ" placeholder="選択してください"
options={categoryOptions}
        onchange={handleCategoryChange} required>
    </lightning-combobox>
    選択項目は {selectedValue} です！
</template>

import { LightningElement,track } from 'lwc';
export default class TestTrack extends LightningElement {
    @track myName;
    @track selectedValue;
    categoryOptions = [
            { label: 'New', value: 'new' },
            { label: 'In Progress', value: 'inProgress' },
            { label: 'Finished', value: 'finished' },
    ];

    changeName(event) {
        this.myName = event.target.value;
    }
    handleCategoryChange(event) {
        this.selectedValue = event.target.value;
    }
}
```

5.1.3 js中的get，set

```
<template>
    hello {myName} !
    <lightning-input name="myName" label="enter the name:" onchange={changeName}></lightning-input>
    <lightning-combobox name="Category" label="カテゴリ" placeholder="選択してください"
options={categoryOptions}
        onchange={handleCategoryChange} required>
    </lightning-combobox>
    選択項目は {selectedValue} です！
    <lightning-combobox name="Category1" label="カテゴリ" placeholder="選択してください"
options={categoryOptions1}
        onchange={handleCategoryChange1} required>
    </lightning-combobox>
    選択項目は {selectedValue1} です！
</template>
```

```
import { LightningElement,track } from 'lwc';
export default class TestTrack extends LightningElement {
    @track myName;
    @track selectedValue;
    @track selectedValue1;
    categoryOptions = [
            { label: 'New', value: 'new' },
            { label: 'In Progress', value: 'inProgress' },
            { label: 'Finished', value: 'finished' },
    ];

    get categoryOptions1() {
        return [
            { label: 'New', value: 'new' },
            { label: 'In Progress', value: 'inProgress' },
            { label: 'Finished', value: 'finished' },
        ];
    }
    changeName(event) {
        this.myName = event.target.value;
    }
    handleCategoryChange(event) {
        this.selectedValue = event.target.value;
    }
    handleCategoryChange1(event) {
        this.selectedValue1 = event.target.value;
    }
}
```

如果遇到get，你就把get后面的东西想象成属性

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <script type="module">
        import {_name} from './js/module01.js';
        console.log(_name.fn);
        _name.fn = 'aaaaaaaaaa';
        console.log(_name.fn);
    </script>
</body>
</html>
```

```
let _name = {
    _x: 'test get()',
    get fn() {
        return this._x;
    },
    set fn(_y) {
        this._x = _y;
    }
}
export { _name};
```

5.1.4 @track问题说明

每次release之后官方都会提供release一览
SALESFORCE SPRING '20 リリースノート[5]
在不使用@track的情况下，无法跟踪person.name的变化

```
    @track person = {
        name: 'jin',
        age: 20,
    }
    changeName(event) {
        this.person.name = event.target.value;
    }

<template>
    <!--hello {myName} -->
    hello: {person.name}
</template>
```

5.1.5 使用扩展运算符(...)：copy

```
    person = {
        name: 'jin',
        age: 20,
    }
    changeName(event) {
        this.person = { ...this.person, 'name': event.target.value };
    }

<template>
    <!--hello {myName} -->
    hello: {person.name}
</template>
```

5.1.6 深拷贝还是浅拷贝

(...)的作用是复制了一份，但是一提到复制，就有一个绕不开的问题：深拷贝还是浅拷贝
对象的扩展运算符（...）深拷贝还是浅拷贝[6]

深拷贝：深拷贝是将一个对象从内存中完整的拷贝一份出来,从堆内存中开辟一个新的区域存放新对象,且修改新对象不会影响原对象。
浅拷贝：浅拷贝是创建一个新对象，这个对象有着原始对象属性值的一份精确拷贝。如果属性是基本类型，拷贝的就是基本类型的值，如果属性是引用类型，拷贝的就是内存地址 ，所以如果其中一个对象改变了这个地址，就会影响到另一个对象。

```
let person = {
    name: 'xiaojin',
    age: 100,
}
let person1 = {...aa};
person1.age = 22;
console.log(person.age);
```

:collision: 直接运用js代码安装【code runner】
安装之后，在右上角有一个可以运行的箭头

```
let person = {
    name: 'xiaojin',
    age: 100,
    address: {
        city:'beijing'
    }
}
let person1 = {...person};
person1.address.city = 'shanghai';
console.log(person.address.city);
```

:star2: 画图说明问题：

```
let person = {
    name: 'xiaojin',
    age: 100,
    address: {
        city:'beijing'
    }
}
let person1 = {...person, address: {...person.address}};
person1.address.city = 'shanghai';
console.log(person.address.city);
```


```
let person = {
    name: 'zhang san',
    age: 18,

    address: {
        city:'beijing'
    }
}
let person1 = { ...person };
person1.address.city = 'shanghai';
person1.age = 81;

console.log(person.address.city);

console.log(person1.address.city);

console.log(person.age);
console.log(person1.age);
```

5.1.7 get使用说明

:exclamation::exclamation: **get错误的例子**

```
// 这里是错误的例子
<template>
    <p>hello LWC!</p>
    <div>{users[0]}</div>
    <div>{num1 * num2}</div>
</template>
-----------------------------------------------------------------------------------------------
import { LightningElement } from 'lwc';
export default class TestLwc extends LightningElement {
    users = ['jzy1', 'jzy2'];
    num1 = 10;
    num2 = 20;
}
```

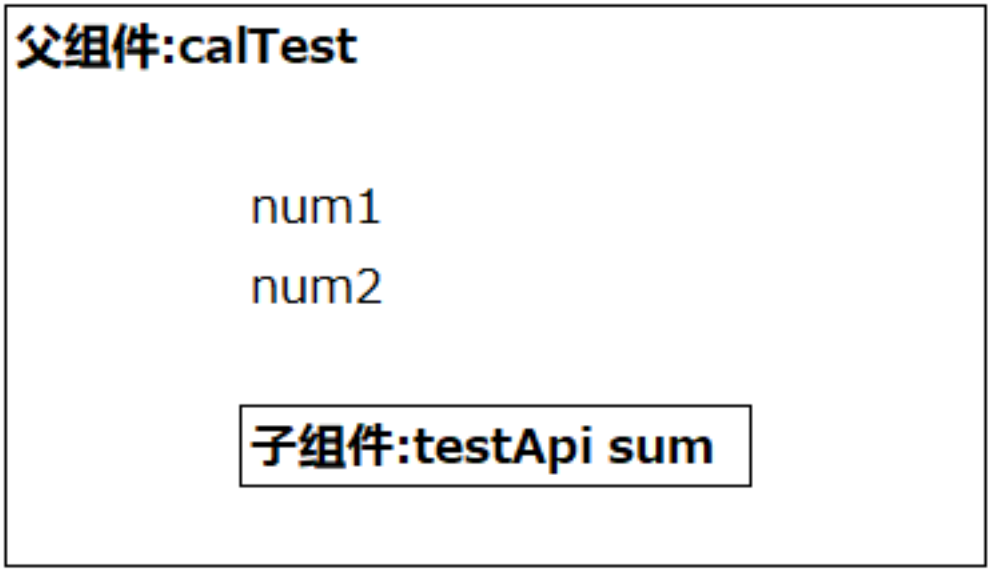
:exclamation::exclamation: **get正确的例子**

```
// 这里是正确的例子
<template>
    <p>hello LWC!</p>
    <div>{firstUserName}</div>
    <div>{sum}</div>
</template>
-------------------------------------------------------------
import { LightningElement } from 'lwc';
export default class TestLwc extends LightningElement {
    users = ['jzy1', 'jzy2'];
    get firstUserName() {
        return this.users[0];
    }
    num1 = 10;
    num2 = 20;
    get sum() {
        return this.num1 * this.num2;
    }
}
```

5.2 @api

プロパティの値が変更されると、コンポーネントが再レンダリングされます。プロパティはパブリックなので、自コンポーネント外からでもアクセス可能です。（パブリックリアクティブプロパティ）
如果一个属性的值被改变，该组件将被重新渲染。 由于该属性是公共的，它可以从组件外部被访问。

5.2.1 父的值传给子



父亲的组件:calTest

```
<template>
    <lightning-card>
        <lightning-input name="num1" label="first number" onchange={handleChange}></lightning-input>
        <lightning-input name="num2" label="second number" onchange={handleChange}></lightning-input>
        <c-test-api sumchild={sumparent}></c-test-api>
    </lightning-card>
</template>

import { LightningElement,track } from 'lwc';
export default class CalTest extends LightningElement {
    num1 = 0;
    num2 = 0;
    @track sumparent;
    handleChange(event) {
        if (event.target.name === 'num1') {
            this.num1 = event.target.value;
        }
        if (event.target.name === 'num2') {
            this.num2 = event.target.value;
        }
        this.sumparent = parseInt(this.num1) + parseInt(this.num2);
    }
}
```

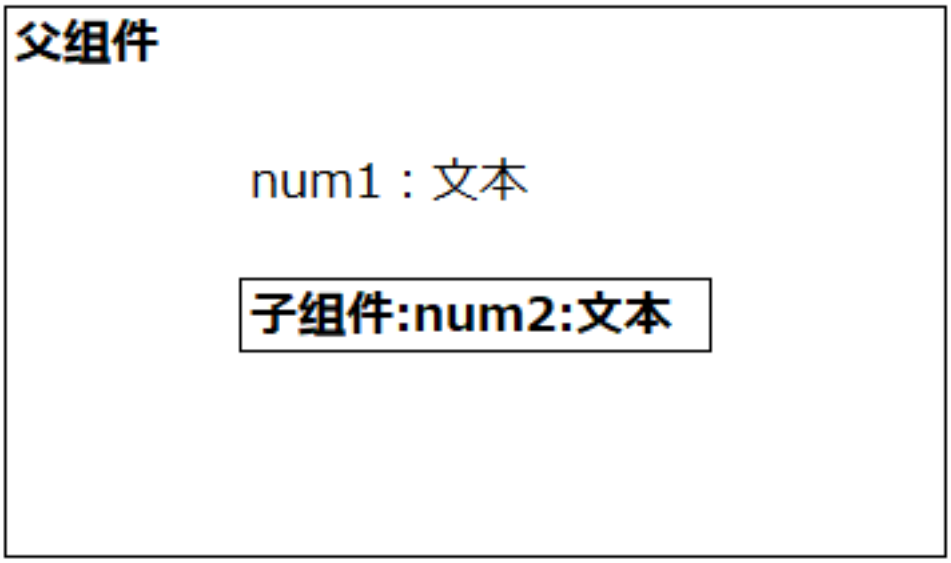
儿子的组件

```
<template>
    <div>
        result:{sumchild}
    </div>
</template>

import { LightningElement, api } from 'lwc';
export default class TestApi extends LightningElement {
    @api sumchild;
}
```

5.2.2 子的值传给父

大家理解了上面的操作之后，看看能不能完成 子的值传给父 的操作

※要求num1的值跟随着num2的变化而变化



5.2.3 无父子关系通信

Lightning Message Service
サンプル[7]

5.3 @wire

Lightning Data Serviceが提供する wireアダプター または Apexメソッド を利用し、Salesforceのデータまたはメタデータの操作ができます。wireサービス がデータをプロビジョニングすると、コンポーネントが再レンダリングされます。（リアクティブワイヤサービス）
Lightning数据服务提供的线程适配器或Apex方法可用于操作Salesforce数据或元数据。

5.3.1 @wire取得后台数据

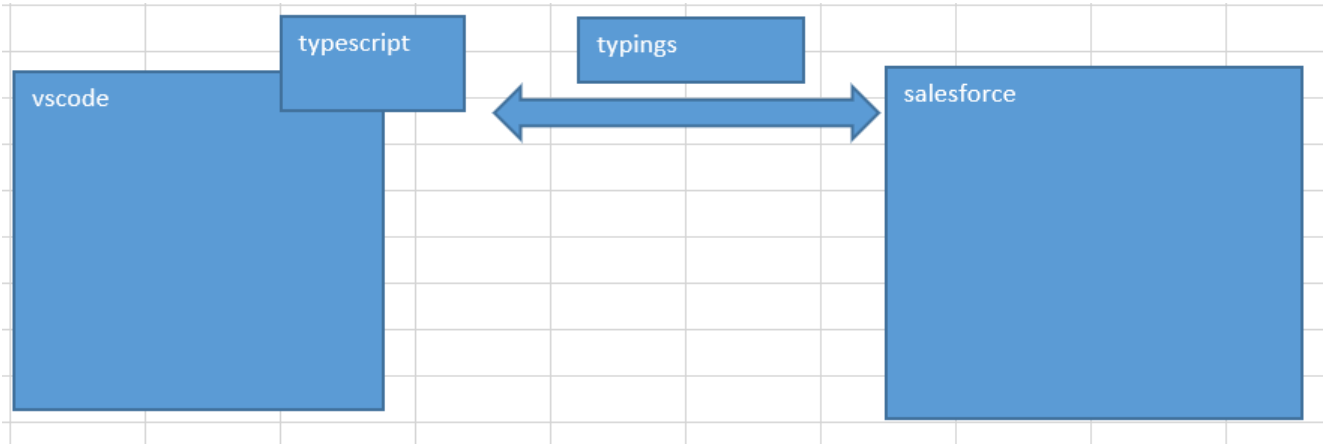
ContactController.cls

```
public with sharing class ContactController {
    public ContactController() {
    }
    @AuraEnabled(Cacheable=true)
    public static List<Contact> getContactName(){
        return [
            SELECT
             Id,
             Name
            FROM Contact
            order by Name
        ];
    }
}
```

```
import { LightningElement,wire } from 'lwc';
import getContactName from '@salesforce/apex/ContactController.getContactName'
export default class TestWire extends LightningElement {
    @wire(getContactName)
    contacts;
}

<template>
    <lightning-card>
        <template for:each={contacts.data} for:item="contact">
            <li key={contact.Id}>
                {contact.Name}
            </li>
        </template>
    </lightning-card>
</template>
```

5.3.2 调用原理说明

调用原理说明：参照typings
:dizzy: **此处应该有图**



5.3.3 关于cacheable=true

官方文档[8]

実行時のパフォーマンスを改善するには、@AuraEnabled(cacheable=true) アノテーションを Apex メソッドに付加して、クライアントにメソッドの結果をキャッシュします。cacheable=true を設定するには、メソッドはデータの取得のみを行う必要があり、データを変更することはできません。
为了提高运行时的性能，你可以向Apex方法添加@AuraEnabled(cacheable=true)注解，以便为客户端缓存方法结果。 要设置cacheable=true，方法必须只检索数据，不修改数据。

5.3.4 参数传递

```
import { LightningElement,wire } from 'lwc';
import getContactName from '@salesforce/apex/ContactController.getContactName'
export default class TestWire extends LightningElement {
    @wire(getContactName, { name: 'AAA' })
    contacts;
}

public with sharing class ContactController {
    public ContactController() {
    }
    @AuraEnabled(Cacheable=true)
    public static List<Contact> getContactName(String name){
        return [
            SELECT
             Id,
             Name
            FROM Contact
            where name =: name
        ];
    }
}
```

5.3.5 处理重载方法

使用别名

首先看vscode怎么给我们做的映射

```
declare module "@salesforce/apex/ContactController.getContactName" {
  export default function getContactName(): Promise<any>;
}
declare module "@salesforce/apex/ContactController.getContactName" {
  export default function getContactName(param: {name: any}): Promise<any>;
}

import { LightningElement,wire } from 'lwc';
import getContactName from '@salesforce/apex/ContactController.getContactName'
import getContactName111 from '@salesforce/apex/ContactController.getContactName'
export default class TestWire extends LightningElement {
    // @wire(getContactName)
    @wire(getContactName111, { name: 'AAA' })
    contacts;
}
```

```
public with sharing class ContactController {
    public ContactController() {
    }
    @AuraEnabled(Cacheable=true)
    public static List<Contact> getContactName(){
        return [
            SELECT
             Id,
             Name
            FROM Contact
            order by Name
        ];
    }
    @AuraEnabled(Cacheable=true)
    public static List<Contact> getContactName(String name){
        return [
            SELECT
             Id,
             Name
            FROM Contact
            where name =: name
        ];
    }
}
```

5.3.6 返回值的处理方法

从Apex侧过来的值，通常不会直接返回给画面，需要在js侧进行处理。

```
import { LightningElement,wire } from 'lwc';
import getContactName from '@salesforce/apex/ContactController.getContactName'
import getContactName111 from '@salesforce/apex/ContactController.getContactName'
export default class TestWire extends LightningElement {
    data;
    // @wire(getContactName)
    @wire(getContactName111, { name: 'AAA' })
    contacts({ data, error }) {
        if (data) {
            this.data = data;
        } else if (error) {
            this.data = undefined;
        }
    }
}
```

```
<template>
    <lightning-card>
        <template for:each={data} for:item="contact">
            <li key={contact.Id}>
                {contact.Name}
            </li>
        </template>
    </lightning-card>
</template>
```

5.3.7 回调函数

1. 什么是回调函数
2. 回调函数怎么使用

通过 **nodejs**代码来掩饰回调函数。
比如说我要读取一个文件，或者说下载一个文件，需要考虑两个问题：
* 1是说我这个操作是同步的还是异步的呢？白话来说就是我这个文件没下载完，能不能做其他的事情呢？
* 2是如果一旦文件下载过程中出错了怎么办？比如说文件不存在
现做一个同步的操作，完全不考虑上述的情况

```
var fs = require('fs');
var data = fs.readFileSync('input.txt');
console.log(data.toString());
```

通常情况下，是不会用同步的方式来下载文件的，都是异步的操作，
那异步的情况下，就是碰到一个问题，如果出错终止了，或者下载成功结束了，怎么通知我呢
所以，有人就相出了一个方法，我可不可以事先些一个函数，等着程序执行完了，通知我呢，
这个函数就是回调函数，而且异步操作回调函数是必须的

```
var fs = require('fs');
fs.readFile('input.txt', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data.toString());
        console.log('success！');
    }
});
```

:angel: **回调函数虽然好用，简单的回调没什么问题，但是如果是多次的复杂的处理，会产生很多问题，通常称为回调地狱**
Promise就是为了解决这个问题而产生的。

5.3.8 Promise(普罗米斯)

6 template tag

6.1 If:true/If:false

```
<template if:true={表达式}>
    <div></div>
</template>
<template if:false={表达式}>
   <div></div>
 </template>
```

{表达式} ： 可以是一个变量，可以是一个方法。

6.1.1 {表达式} 是变量

```
<template>
   <div class="slds-m-top_small slds-m-bottom_medium">
      <h2 class="slds-text-heading_small slds-m-bottom_small">
         Click the buttons to activate the <code>onclick</code> handler and view the label of the clicked
button.
      </h2>
      <lightning-button
      variant="brand"
      label="Brand"
      title="Primary action"
      onclick={handleClick}
      class="slds-m-left_x-small">
      </lightning-button>

      <template if:true={isVisible}>
            <div>this is true!</div>
      </template>
      <template if:false={isVisible}>
         <div>this is false!</div>
      </template>
   </div>
</template>

import { LightningElement } from 'lwc';
export default class TestTag extends LightningElement {
   isVisible = false;
   handleClick(){
      this.isVisible = true;
   }
}
```

6.1.2 {表达式} 是方法

```
<template>
  <lightning-card title="Conditional rendering">
    <div class="slds-m-top_small slds-m-bottom_medium">
      <h2 class="slds-text-heading_small slds-m-bottom_small">
        Click the buttons to activate the <code>onclick</code> handler and view the label of the clicked
button.
      </h2>
      <lightning-button
      variant="brand"
      label="Brand"
      title="Primary action"
      onclick={handleClick}
      class="slds-m-left_x-small">
      </lightning-button>

      <template if:true={isVisible}>
          <div>this is true!</div>
      </template>
      <template if:false={isVisible}>
        <div>this is false!</div>
      </template>
      <lightning-input
      type="text"
      label="input hello to see the data" onkeyup={changeHandler}>
      </lightning-input>
      <template if:true={helloMethod}>
        <div>this is hello ! </div>
      </template>
    </div>
  </lightning-card>
</template>

import { LightningElement } from 'lwc';
export default class TestTag extends LightningElement {
  isVisible = false;
  name;

  handleClick(){
    this.isVisible = true;
  }
  changeHandler(event){
    this.name = event.target.value
  }
  get helloMethod(){
    return this.name === 'hello'
  }
}
```

6.2 loop

- for:each
- iterator

for:each, for:item, for:index を使用する。繰り返し要素には識別のためにkey属性を付ける必要がある。

```
<template for:each={items} for:item="item" for:index="idx">
    <p key={item.id}>{idx}:{item}</p>
</template>
1:要素1
2:要素2
3:要素3
・・・・
```

iteratorを使うやり方もある。この方法だと最初と最後の要素を識別することが可能。

```
<template iterator:it={contacts}>
  <li key={it.value.Id}>
    <div if:true={it.first} class="list-first"></div>
      {it.value.Name}, {it.value.Title}
    <div if:true={it.last} class="list-last"></div>
  </li>
</template>
//it.value: 値
//it.index: インデックス
//it.first: 最初の要素でtrue
//it.last:  最後の要素でtrue
```

6.2.1 for:each

- for:each
- for:item
- key

:star2: Sample1:

```
<template>
    <lightning-card title="test for:each ">
        <div class="slds-var-m-around_medium">
            <template for:each={numList} for:item="num">
                <li key={num}>
                    {num}
                </li>
            </template>
        </div>
    </lightning-card>
</template>
```

```
import { LightningElement } from 'lwc';
export default class TestLoop extends LightningElement {
    numList = ['991', '222', '333','222'];
}
```

:star2: Sample2:

```
<template>
    <lightning-card title="test for:each ">
        <div class="slds-var-m-around_medium">
            <template for:each={numList} for:item="num">
                <li key={num}>
                    {num}
                </li>
            </template>
        </div>
    </lightning-card>

    <lightning-card title="test for:each ">
        <div class="slds-var-m-around_medium">
            <template for:each={persons} for:item="person">
                <li key={person.id}>
                    {person.name} : {person.age}
                </li>
            </template>
        </div>
    </lightning-card>
</template>

import { LightningElement } from 'lwc';
export default class TestLoop extends LightningElement {
    numList = ['991', '222', '333', '222'];

    persons = [
        {
            id:'AAAAA',
            name: 'A1',
            age:20,
        },
        {
            id:'AAAAB',
            name: 'A2',
            age:18
        }
    ]
}
```

6.2.2 iterator

- <template iterator:**person**={persons}>

- {person.**value**.name}

:star: **Sample1：**

```
    <lightning-card title="test iterator ">
        <div class="slds-var-m-around_medium">
            <template iterator:person={persons}>
                <li key={person.value.id}>
                    {person.value.name} : {person.value.age}
                </li>
            </template>
        </div>
    </lightning-card>

import { LightningElement } from 'lwc';
export default class TestLoop extends LightningElement {
    numList = ['991', '222', '333', '222'];

    persons = [
        {
            id:'AAAAA',
            name: 'A1',
            age:20,
        },
        {
            id:'AAAAB',
            name: 'A2',
            age:18
        }
    ]
}
```

:star: **Sample2：** first，last

```
    <lightning-card title="test iterator ">
        <div class="slds-var-m-around_medium">
            <template iterator:person={persons}>
                <li key={person.value.id}>
                    <template if:true={person.first}>
                        this first<br/>
                    </template>
                    {person.value.name} : {person.value.age}
                    <template if:true={person.last}>
                        <br/>this last
                    </template>
                </li>
            </template>
        </div>
    </lightning-card>
```

```
import { LightningElement } from 'lwc';
export default class TestLoop extends LightningElement {
    numList = ['991', '222', '333', '222'];

    persons = [
        {
            id:'AAAAA',
            name: 'A1',
            age:20,
        },
        {
            id:'AAAAB',
            name: 'A2',
            age:18
        }
    ]
}
```

7. Promise理解和使用

7.1 Promise基础

Promise是JS中异步编程的解决方案
异步编程的使用场景
* fs 文件操作
* 数据库操作
* AJAX调用
* 定时器
promise是一个构造函数
用来 **封装** 一个异步操作，并可以获取其成功和失败的结果。

```
const p = new Promise(.....)
```

我们接着上节课的代码，我提到一个匿名函数，就是没有名字，只有参数和方法体，
但是写了一个关键字「function」,其实这个关键字也是可以省略的，有另一种写法，叫做箭头函数

```
fs.readFile('input.txt', (err, data)=> {
    if (err) {
        console.log(err);
        // 送信
    } else {
        console.log(data.toString());
        console.log('success!');
        //
    }
});
```

到了普罗米斯，也是不需要的function

7.2 then关键字

```
let p = new Promise((resolve, reject) => {
    fs.readFile('input1.txt', (err, data) => {
        if (err) reject(err);
        resolve(data);
    });
});
p.then(value => {
    console.log(value.toString());
}, reason => {
    console.log(reason);
});
```

// resolve 函数类型 成功的时候调用
// reject 函数类型 失败的时候调用
通常是使用then方法来处理具体的操作，
「value =>」、「reason =>」只有一个参数，所有就不用些小括号了
而且value和reason都可以随便起名字

```
p.then(value11 => {
    console.log(value11.toString());
}, reason11 => {
    console.log(reason11);
});
```

7.3 catch关键字

```
let p = new Promise((resolve, reject) => {
    fs.readFile('input1.txt', (err, data) => {
        if (err) reject(err);
        resolve(data);
    });
});
p.then(value => {
    console.log(value.toString());
}).catch(reason => {
    console.log(reason);
})
```

7.4 finally关键字

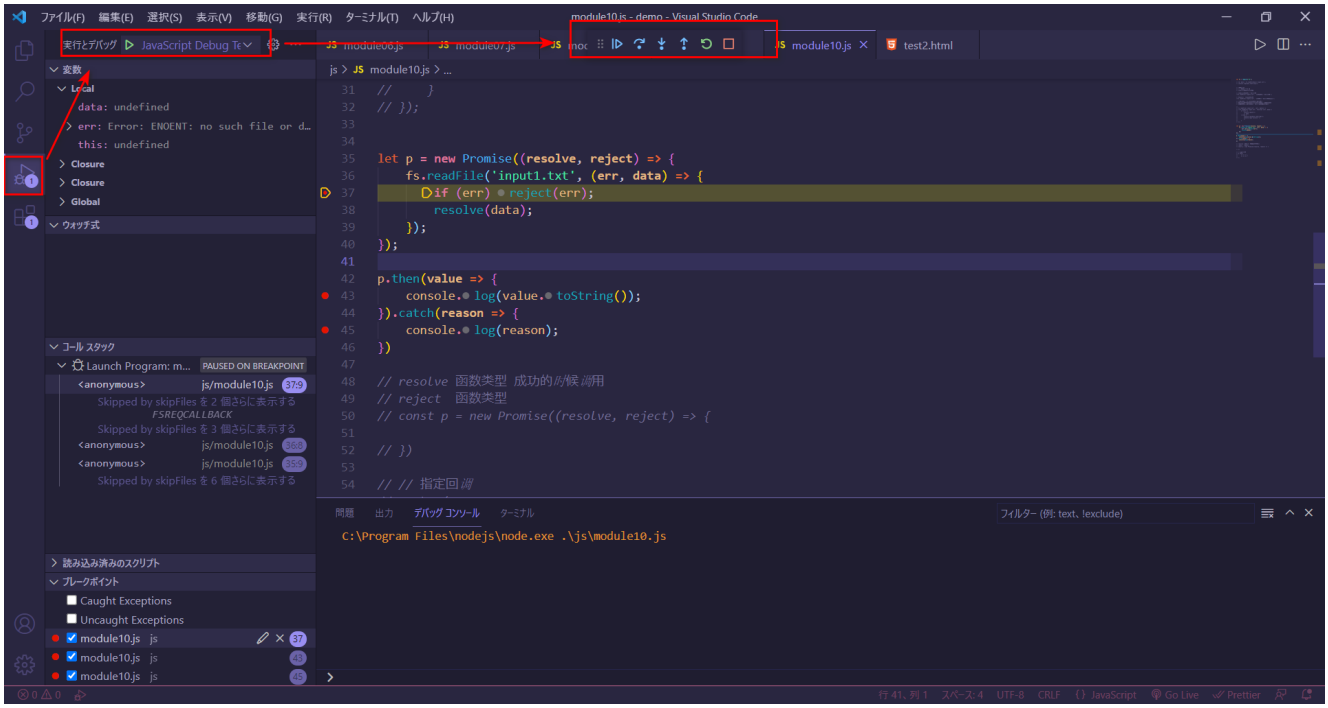
```
    data;
    connectedCallback() {
        getContact().then(result => { this.data = result }).catch(error => { alert(error) }).finally(() => { });
    }
```

```
// });

let p = new Promise((resolve, reject) => {
    fs.readFile('input.txt', (err, data) => {
        if (err) reject(err);
        resolve(data);
    });
});

p.then(value => {
    console.log(value.toString());
}).catch(reason => {
    console.log(reason);
})
```

7.5 在lwc中利用promise

```
data;
connectedCallback() {
    getContact().then(result => { this.data = result }).catch(error => {alert(error)});
}
```

7.6 nodejs中debug



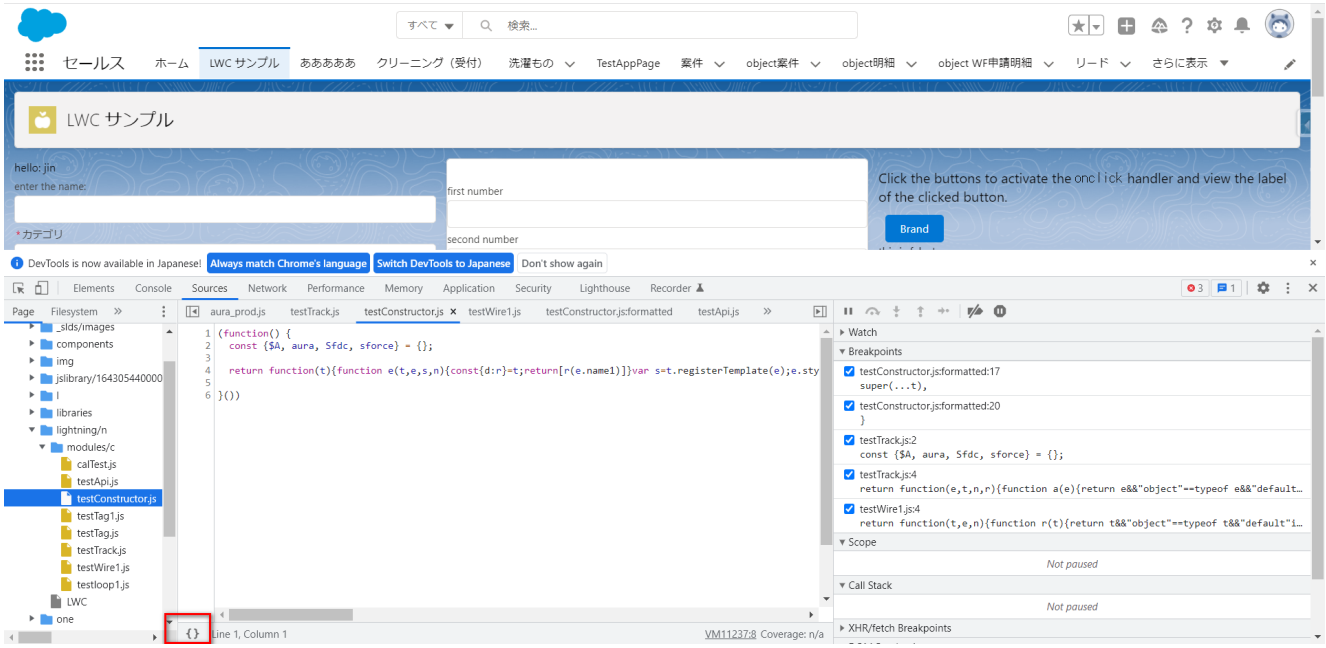
8.組件生命周期

- constructor()
- connectedCallback()
- renderedCallback()

8.1 constructor

```
import { LightningElement } from 'lwc';
export default class TestConstructor extends LightningElement {
    name1;
    constructor() {
        super();
        this.name1 = 'call constructor';
    }
}
```

:collision: 花括号的小技巧



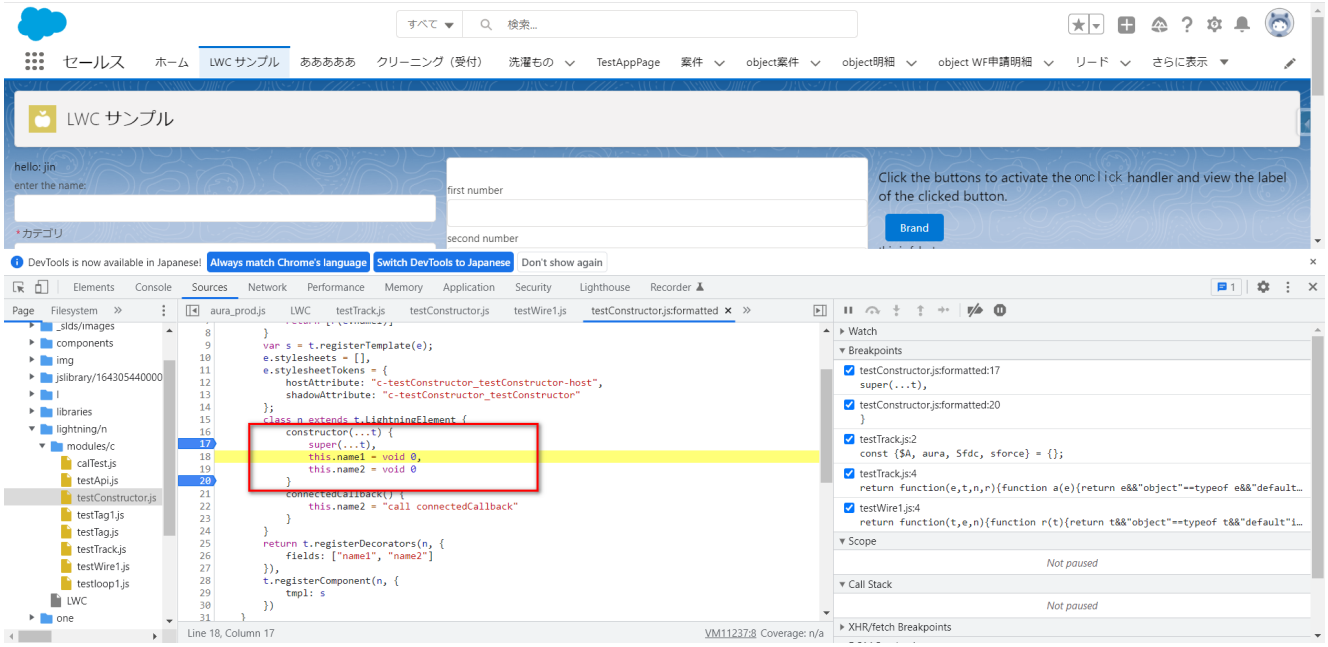
开源的js库，一般会同时提供两个文件，
比如说jquery，开发的时候，我们会用开发版，部署之后，会用压缩版
开发版本：jquery-3.6.0.js
部署版本：jquery-3.6.0.min.js

.

8.2 super()

为什么要有super()呢，super()做了什么操作呢。

:question: 思考：程序没有明显的调用constructor，this为什么会好用呢



操作流程：

1.注册模板

2.调用构造方法

8.3 connectedCallback

```
import { LightningElement } from 'lwc';
export default class TestConstructor extends LightningElement {
    name1;
    name2;
    connectedCallback() {
        this.name2 = 'call connectedCallback';
    }
}
```

8.4 render

一定要返回一个模板文件

引出一个小问题，lwc中如何引入自定义的资源（和组件名字不同的资源）

```
import test01 from "./test01.html"

import { LightningElement } from 'lwc';
import test01 from "./test01.html"
export default class TestConstructor extends LightningElement {
    name1;
    name2;
    constructor() {
        super();
        this.name1 = 'call constructor';
    }
    connectedCallback() {
        this.name2 = 'call connectedCallback';
    }
    render() {
        this.name2 = 'call render';
        return test01;
    }
}
```

```
import { LightningElement } from 'lwc';
import test01 from "./test01.html"
import test02 from "./test02.html"
export default class TestConstructor extends LightningElement {
    name1;
    name2;
    tp = 'test02';
    constructor() {
        super();
        this.name1 = 'call constructor';
        console.log('call constructor');
    }
    connectedCallback() {
        this.name2 = 'call connectedCallback';
        console.log('call connectedCallback');
    }
    render() {
        this.name2 = 'call render';
        console.log('call render');
        if (this.tp === 'test01') {
            return test01;
        } else {
            return test02;
        }
    }
}
```

8.5 renderedCallback

在render之后调用

8.6 disconnectedCallback

https://developer.salesforce.com/docs/component-library/documentation/ja-jp/lwc/lwc.create_lifecycle_hooks_dom

当离开当前画面的时候，这个方法触发。

```
    disconnectedCallback() {
        console.log('call disconnectedCallback');
    }
```

8.7 errorCallback

用的不多

8.8 父子组件调用问题

如果涉及到父子组件调用的时候，生命周期如何处理

- parent constructor

- parent connectedCallback
- parent render
- child constructor
- child connectedCallback
- child render
- child renderedCallback
- parent renderedCallback

9.Lightning Message Service

Lightning Message Service简称LMS，用于在 VF Page, Aura Component, lwc之间进行跨DOM 通讯

リンク

[1] 官方文档, https://developer.salesforce.com/blogs/developer-relations/2015/05/loading-external-js-css-libraries-lightning-components
[2] LWC 调用 Apex, https://trailhead.salesforce.com/ja/content/learn/modules/lightning-web-components-and-salesforce-data/use-apex-to-work-with-data
[3] 命名規則, https://developer.salesforce.com/docs/component-library/documentation/ja-jp/lwc/lwc.create_components_folder
[4] 3つのデコレータとそれぞれの使い方, https://keneloper.com/three-decorators-and-how-to-use-each/
[5] SALESFORCE SPRING '20 リリースノート, https://help.salesforce.com/s/articleView?id=release-notes.rn_lwc_track.htm&type=5&release=224
[6] 对象的扩展运算符（...）深拷贝还是浅拷贝, https://segmentfault.com/a/1190000039296602
[7] サンプル, https://qiita.com/TaaaZyyy/items/ec82c0f048440e904d5e#親子関係でないコンポーネントへの通信
[8] 官方文档, https://developer.salesforce.com/docs/component-library/documentation/ja-jp/lwc/apex