Webpack

- 简介

  可以看做一个模块化打包机，分析项目结构，处理模块化依赖，转换成为浏览器可运行的代码。

  - 代码转换: TypeScript 编译成 JavaScript、SCSS,LESS 编译成 CSS.
  - 文件优化：压缩 JavaScript、CSS、HTML 代码，压缩合并图片。
  - 代码分割：提取多个页面的公共代码、提取首屏不需要执行部分的代码让其异步加载。
  - 模块合并：在采用模块化的项目里会有很多个模块和文件，需要构建功能把模块分类合并成一个文件。
  - 自动刷新：监听本地源代码的变化，自动重新构建、刷新浏览器。

- 详解

  webpack需要在项目根目录下创建一个webpack.config.js来导出webpack的配置,配置多样化，可以自行定制，下面讲讲最基础的配置。

  ```
  module.exports = {
      entry: './src/index.js',
      output: {
          path: path.join(__dirname, './dist'),
          filename: 'main.js',
      }
  }
  ```

  - `entry`代表入口，webpack会找到该文件进行解析
  - `output`代表输入文件配置
  - `path`把最终输出的文件放在哪里
  - `filename`输出文件的名字

  有时候我们的项目并不是spa，需要生成多个js html，那么我们就需要配置多入口。

  ```
  module.exports = {
      entry: {
          pageA: './src/pageA.js',
          pageB: './src/pageB.js'
      },
      output: {
          path: path.join(__dirname, './dist'),
          filename: '[name].[hash:8].js',
      },
  }
  ```

  `entry`配置一个对象，key值就是`chunk`： 代码块，一个 Chunk 由多个模块组合而成，用于代码合并与分割。看看filename`[name]`: 这个name指的就是chunk的名字，我们配置的key值`pageA` `pageB`，这样打包出来的文件名是不同的，再来看看`[hash]`，这个是给输出文件一个hash值，避免缓存，那么`:8`是取前8位。

- HtmlWebpackPlugin

  该插件可以给每一个chunk生成html,指定一个`template`,可以接收参数，在模板里面使用，下面来看看如何使用：

```
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
    entry: {
        pageA: './src/pageA.js',
        pageB: './src/pageB.js'
    },
    output: {
        path: path.join(__dirname, './dist'),
        filename: '[name].[hash:8].js',
    },
    plugins: [
         new HtmlWebpackPlugin({
            template: './src/templet.html',
            filename: 'pageA.html',
            title: 'pageA',
            chunks: ['pageA'],
            hash: true,
            minify: {
                removeAttributeQuotes: true
            }
        }),
        new HtmlWebpackPlugin({
            template: './src/templet.html',
            filename: 'pageB.html',
            title: 'pageB',
            chunks: ['pageB'],
            hash: true,
            minify: {
                removeAttributeQuotes: true
            }
        }),
    ]
}
```

  - `template`: html模板的路径地址
  - `filename`: 生成的文件名
  - `title`: 传入的参数
  - `chunks`: 需要引入的chunk
  - `hash`: 在引入JS里面加入hash值 比如: `<script src='index.js?2f373be992fc073e2ef5'></script>`
  - `removeAttributeQuotes`: 去掉引号，减少文件大小`<script src=index.js></script>`

这样在dist目录下就生成了pageA.html和pageB.html并且通过配置chunks，让pageA.html里加上了script标签去引入pageA.js。那么现在还剩下css没有导入，css需要借助loader去做，所以现在要下载几个依赖，以scss为例，less同理

```
npm install css-loader style-loader sass-loader node-sass -D
```

- `css-loader`: 支持css中的import
- `style-loader`: 把css写入style内嵌标签
- `sass-loader`: scss转换为css
- `node-sass`: scss转换依赖

```
module.exports = {
    module: {
        rules: [
                {
                    test: /\.scss$/,
                    use: ['style-loader', 'css-loader', 'sass-
loader'],
                    exclude: /node_modules/
                }
        ]
    }
}
```

- `test`: 一个正则表达式，匹配文件名
- `use`: 一个数组，里面放需要执行的loader，倒序执行，从右至左。
- `exclude`: 取消匹配node_modules里面的文件

- ExtractTextPlugin：把css作为一个单独的文件

```
npm i extract-text-webpack-plugin@next -D
```

```
const ExtractTextPlugin = require('extract-text-webpack-plugin');
module.exports = {
    entry: './src/index.js',
    output: {
        path: path.join(__dirname, './dist'),
        filename: 'main.js',
    },
    module: {
        rules: [
            {
                test: /\.scss$/,
                use: ExtractTextPlugin.extract({
                    // style-loader 把css直接写入html中style标签
                    fallback: 'style-loader',
                    // css-loader css中import支持
```

```
                    // loader执行顺序 从右往左执行
                    use: ['css-loader', 'sass-loader']
                }),
                exclude: /node_modules/
            }
        ]
    },
    plugins: [
        new ExtractTextPlugin('[name].[contenthash:8].css'),
    ]
}
```

- 需要在plugins里加入插件`name`: chunk名字 `contenthash:8`: 根据内容生成hash值取前8位
- 修改loader配置下的`use`: `fallback`: 兼容方案

- `babel-loader`: 用babel转换代码
- `url-loader`: 依赖于`file-loader`,把图片转换成base64嵌入html,如果超出一定阈值则交给file-loader

```
rules: [
         // 处理js
         {
            test: /\.js?$/,
            exclude: /node_modules/,
            use: ['babel-loader']
        },
        // 处理图片
        {
            test: /\.(png|jpg|gif|ttf|eot|woff(2)?)(\?[=a-z0-9]+)?
$/,
            use: [{
                loader: 'url-loader',
                options: {
                    query: {
                        // 阈值 单位byte
                        limit: '8192',
                        name: 'images/[name]_[hash:7].[ext]',
                    }
                }
            }]
        },
    ]
```

babel的配置建议在根目录下新建一个.babelrc文件

```
{
    "presets": [
        "env",
        "stage-0",
        "react"
    ],
    "plugins": [
        "transform-runtime",
        "transform-decorators-legacy",
        "add-module-exports"
    ]
}
```

- `presets`: 预设, 一个预设包含多个插件 起到方便作用 不用引用多个插件
- `env`: 只转换新的句法，例如const let => ..等 不转换 Iterator、Generator、Set、Maps、Proxy、Reflect、Symbol、Promise、Object.assign。
- `stage-0`: es7提案转码规则 有 0 1 2 3 阶段 0包含 1 2 3里面的所有
- `react`: 转换react jsx语法
- `plugins`: 插件 可以自己开发插件 转换代码(依赖于ast抽象语法数)
- `transform-runtime`: 转换新语法，自动引入polyfill插件，另外可以避免污染全局变量
- `transform-decorators-legacy`: 支持装饰器
- `add-module-exports`: 转译export default {}; 添加上module.exports = exports.default 支持commonjs

因为我们在文件名中加入hash值，打包多次后dist目录变得非常多文件，没有删除或覆盖，这里可以引入一个插件，在打包前自动删除dist目录，保证dist目录下是当前打包后的文件:

```
plugins: [
    new CleanWebpackPlugin(
        // 需要删除的文件夹或文件
        [path.join(__dirname, './dist/*.*')],
        {
            // root目录
            root: path.join(__dirname, './')
        }
    ),
]
```

指定extension之后可以不用在require或是import的时候加文件扩展名,会依次尝试添加扩展名进行匹配:

```
resolve: {
    extensions: ['.js', '.jsx', '.scss', '.json'],
},
```

- 提出公共的JS文件

  webpack4中废弃了webpack.optimize.CommonsChunkPlugin插件，用新的配置项替代

```
module.exports = {
    entry: './src/index.js',
    output: {
        path: path.join(__dirname, './dist'),
        filename: 'main.js',
    },
    optimization: {
        splitChunks: {
            cacheGroups: {
                commons: {
                    chunks: 'initial',
                    minChunks: 2,
                    maxInitialRequests: 5,
                    minSize: 2,
                    name: 'common'
                }
            }
        }
    },
}
```

- HappyPack

  在webpack运行在node中打包的时候是单线程去一件一件事情的做，HappyPack可以开启多个子进程去并发执行，子进程处理完后把结果交给主进程

```
npm i happypack -D
```

```
const HappyPack = require('happypack');
module.exports = {
    entry: './src/index.js',
    output: {
        path: path.join(__dirname, './dist'),
        filename: 'main.js',
    },
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                use: 'happypack/loader?id=babel',
            },
        ]
    },
    plugins: [
```

```
        new HappyPack({
            id: 'babel',
            threads: 4,
            loaders: ['babel-loader']
        }),
    ]
}
```

- `id`: id值，与loader配置项对应
- `threads`: 配置多少个子进程
- `loaders`: 用什么loader处理

- 作用域提升

  如果你的项目是用ES2015的模块语法，并且webpack3+，那么建议启用这一插件，把所有的模块放到一个函数里，减少了函数声明，文件体积变小，函数作用域变少。

```
module.exports = {
    entry: './src/index.js',
    output: {
        path: path.join(__dirname, './dist'),
        filename: 'main.js',
    },
    plugins: [
        new webpack.optimize.ModuleConcatenationPlugin(),
    ]
}
```

- 提取第三方库

  方便长期缓存第三方的库,新建一个入口，把第三方库作为一个chunk，生成vendor.js

```
module.exports = {
    entry: {
        main: './src/index.js',
        vendor: ['react', 'react-dom'],
    },
}
```

- DLL动态链接

  第三库不是经常更新，打包的时候希望分开打包，来提升打包速度。打包dll需要新建一个webpack配置文件，在打包dll的时候，webpack做一个索引，写在manifest文件中。然后打包项目文件时只需要读取manifest文件。

  `webpack.vendor.js`

```
const webpack = require('webpack');
const path = require('path');
```

```
module.exports = {
    entry: {
        vendor: ['react', 'react-dom'],
    },
    output: {
        path: path.join(__dirname, './dist'),
        filename: 'dll/[name]_dll.js',
        library: '_dll_[name]',
    },
    plugins: [
        new webpack.DllPlugin({
            path: path.join(__dirname, './dist/dll',
'manifest.json'),
            name: '_dll_[name]',
        }),
    ]
};
```

`path`: manifest文件的输出路径 `name`: dll暴露的对象名，要跟output.library保持一致
`context`: 解析包路径的上下文，这个要跟接下来配置的dll user一致

`webpack.config.js`

```
module.exports = {
    entry: {
        main: './src/index.js',
        vendor: ['react', 'react-dom'],
    },
    plugins: [
        new webpack.DllReferencePlugin({
            manifest: path.join(__dirname, './dist/dll',
'manifest.json')
        })
    ]
}
```

```
<script src="vendor_dll.js"></script>
```

- 线上和线下

  在生产环境和开发环境其实我们的配置是存在相同点，和不同点的，为了处理这个问题，会创建3个文件:

  - `webpack.base.js`: 共同的配置
  - `webpack.dev.js`: 在开发环境下的配置
  - `webpack.prod.js`: 在生产环境的配置

  通过webpack-merge去做配置的合并，比如:

  开发环境:

```
const path = require('path');
const webpack = require('webpack');
const merge = require('webpack-merge');
const base = require('./webpack.base');

const dev = {
    devServer: {
        contentBase: path.join(__dirname, '../dist'),
        port: 8080,
        host: 'localhost',
        overlay: true,
        compress: true,
        open:true,
        hot: true,
        inline: true,
        progress: true,
    },
    devtool: 'inline-source-map',
    plugins: [
        new webpack.HotModuleReplacementPlugin(),
        new webpack.NamedModulesPlugin(),
    ]
}
module.exports = merge(base, dev);
```

开发环境中我们可以启动一个devServer静态文件服务器，预览我们的项目，引入base配置文件，用merge去合并配置。

- `contentBase`: 静态文件地址
- `port`: 端口号
- `host`: 主机
- `overlay`: 如果出错，则在浏览器中显示出错误
- `compress`: 服务器返回浏览器的时候是否启动gzip压缩
- `open`: 打包完成自动打开浏览器
- `hot`: 模块热替换 需要`webpack.HotModuleReplacementPlugin`插件
- `inline`: 实时构建
- `progress`: 显示打包进度
- `devtool`: 生成代码映射，查看编译前代码，利于找bug
- `webpack.NamedModulesPlugin`: 显示模块的相对路径

- 生产环境

```
const path = require('path');
const merge = require('webpack-merge');
const WebpackParallelUglifyPlugin = require('webpack-parallel-
uglify-plugin');
const base = require('./webpack.base');

const prod = {
```

```
    plugins: [
        // 文档: https://github.com/gdborton/webpack-parallel-
uglify-plugin
        new WebpackParallelUglifyPlugin(
            {
                uglifyJS: {
                    mangle: false,
                    output: {
                        beautify: false,
                        comments: false
                    },
                    compress: {
                        warnings: false,
                        drop_console: true,
                        collapse_vars: true,
                        reduce_vars: true
                    }
                }
            }
        ),
    ]
}
module.exports = merge(base, prod);
```

`webpack-parallel-uglify-plugin`可以并行压缩代码，提升打包效率

uglifyJS配置:

- `mangle`: 是否混淆代码
- `output.beautify`: 代码压缩成一行 true为不压缩 false压缩
- `output.comments`: 去掉注释
- `compress.warnings`: 在删除没用到代码时 不输出警告
- `compress.drop_console`: 删除console
- `compress.collapse_vars`: 把定义一次的变量，直接使用，取消定义变量
- `compress.reduce_vars`: 合并多次用到的值，定义成变量

- 总结
  - loader
    - `this.context`: 当前处理文件的所在目录，假如当前 Loader 处理的文件是 /src/main.js，则 this.context 就等于 /src。
    - `this.resource`: 当前处理文件的完整请求路径，包括 querystring，例如 /src/main.js?name=1。
    - `this.resourcePath`: 当前处理文件的路径，例如 /src/main.js。
    - `this.resourceQuery`: 当前处理文件的 querystring。
    - `this.target`: 等于 Webpack 配置中的 Target
    - `this.loadModule`: 但 Loader 在处理一个文件时，如果依赖其它文件的处理结果才能得出当前文件的结果时， 就可以通过 - - - this.loadModule(request: string, callback: function(err, source, sourceMap, module)) 去获得 request 对应文件的处

理结果。
    - `this.resolve`: 像 require 语句一样获得指定文件的完整路径，使用方法为 resolve(context: string, request: string, callback: function(err, result: string))。
    - `this.addDependency`: 给当前处理文件添加其依赖的文件，以便再其依赖的文件发生变化时，会重新调用 Loader 处理该文件。使用方法为 addDependency(file: string)。
    - `this.addContextDependency`: 和 addDependency 类似，但 addContextDependency 是把整个目录加入到当前正在处理文件的依赖中。使用方法为 addContextDependency(directory: string)。
    - `this.clearDependencies`: 清除当前正在处理文件的所有依赖，使用方法为 clearDependencies()。
    - `this.emitFile`: 输出一个文件，使用方法为 emitFile(name: string, content: Buffer|string, sourceMap: {...})。
    - `this.async`: 返回一个回调函数，用于异步执行。

  - plugin

    webpack整个构建流程有许多钩子，开发者可以在指定的阶段加入自己的行为到webpack构建流程中。插件由以下构成:

    - 一个 JavaScript 命名函数。
    - 在插件函数的 prototype 上定义一个 apply 方法。
    - 指定一个绑定到 webpack 自身的事件钩子。
    - 处理 webpack 内部实例的特定数据。
    - 功能完成后调用 webpack 提供的回调。

    整个webpack流程由compiler和compilation构成,compiler只会创建一次，compilation如果开起了watch文件变化，那么会多次生成compilation. 那么这2个类下面生成了需要事件钩子

Web基础

[前端开发面试题]

- CSS选择符
    1. id选择器（ #myid）
    2. 类选择器（.myclassname）
    3. 标签选择器（div, h1, p）
    4. 相邻选择器（h1 + p）
    5. 子选择器（ul > li）
    6. 后代选择器（li a）
    7. 通配符选择器（ * ）
    8. 属性选择器（a[rel = "external"]）
    9. 伪类选择器（a:hover, li:nth-child）
- CSS3新增伪类

```
::after          在元素之前添加内容,也可以用来做清除浮动。
::before              在元素之后添加内容
:enabled
:disabled        控制表单控件的禁用状态。
:checked         单选框或复选框被选中。
```

- 如何居中div
  - 水平居中：给div设置一个宽度，然后添加margin:0 auto属性

```
div{
    width:200px;
    margin:0 auto;
  }
```

- 让绝对定位的div居中

```
div {
    position: absolute;
    width: 300px;
    height: 300px;
    margin: auto;
    top: 0;
    left: 0;
    bottom: 0;
    right: 0;
    background-color: pink; /* 方便看效果 */
 }
```

- 水平垂直居中

```
确定容器的宽高 宽500 高 300 的层
设置层的外边距
div {
    position: relative;      /* 相对定位或绝对定位均可 */
    width:500px;
    height:300px;
    top: 50%;
    left: 50%;
    margin: -150px 0 0 -250px;       /* 外边距为自身宽高的一半 */
    background-color: pink;      /* 方便看效果 */
  }

未知容器的宽高，利用 `transform` 属性
div {
    position: absolute;      /* 相对定位或绝对定位均可 */
    width:500px;
    height:300px;
```

```
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    background-color: pink;      /* 方便看效果 */
 }

利用 flex 布局
实际使用时应考虑兼容性
.container {
    display: flex;
    align-items: center;         /* 垂直居中 */
    justify-content: center;     /* 水平居中 */

 }
 .container div {
    width: 100px;
    height: 100px;
    background-color: pink;      /* 方便看效果 */
 }
```

- display有哪些值？说明他们的作用。

```
block           块类型。默认宽度为父元素宽度，可设置宽高，换行显示。
none            缺省值。象行内元素类型一样显示。
inline          行内元素类型。默认宽度为内容宽度，不可设置宽高，同行显示。
inline-block    默认宽度为内容宽度，可以设置宽高，同行显示。
list-item       象块类型元素一样显示，并添加样式列表标记。
table           此元素会作为块级表格来显示。
inherit         规定应该从父元素继承 display 属性的值。
```

- position的值relative和absolute定位原点是

```
absolute
生成绝对定位的元素，相对于值不为 static的第一个父元素进行定位。
fixed （老IE不支持）
生成绝对定位的元素，相对于浏览器窗口进行定位。
relative
生成相对定位的元素，相对于其正常位置进行定位。
static
默认值。没有定位，元素出现在正常的流中（忽略 top, bottom, left, right z-index 声明）。
inherit
规定从父元素继承 position 属性的值。
```

- CSS3有哪些新特性

```
新增各种CSS选择器    (: not(.input)：所有 class 不是“input”的节点)
圆角              (border-radius:8px)
多列布局           (multi-column layout)
阴影和反射      (Shadow\Reflect)
文字特效           (text-shadow、)
文字渲染           (Text-decoration)
线性渐变           (gradient)
旋转              (transform)
缩放,定位,倾斜,动画,多背景
例如:transform:\scale(0.85,0.90)\ translate(0px,-30px)\
skew(-9deg,0deg)\Animation:
```

- 请解释一下CSS3的Flexbox（弹性盒布局模型）,以及适用场景?

```
一个用于页面布局的全新CSS3功能，Flexbox可以把列表放在同一个方向（从上到下排列，从左到右），并让列表能延伸到占用可用的空间。
    较为复杂的布局还可以通过嵌套一个伸缩容器（flex container）来实现。
    采用Flex布局的元素，称为Flex容器（flex container），简称"容器"。
    它的所有子元素自动成为容器成员，称为Flex项目（flex item），简称"项目"。
    常规布局是基于块和内联流方向，而Flex布局是基于flex-flow流可以很方便的用来做局中，能对不同屏幕大小自适应。
    在布局上有了比以前更加灵活的空间。

    具体：http://www.w3cplus.com/css3/flexbox-basics.html
```

- 请解释一下为什么需要清除浮动？清除浮动的方式

清除浮动是为了清除使用浮动元素产生的影响。浮动的元素，高度会塌陷，而高度的塌陷使我们页面后面的布局不能正常显示。

```
1、父级div定义height;
2、父级div 也一起浮动;
3、常规的使用一个class;
  .clearfix::before, .clearfix::after {
      content: " ";
      display: table;
  }
  .clearfix::after {
      clear: both;
  }
  .clearfix {
      *zoom: 1;
  }

4、SASS编译的时候，浮动元素的父级div定义伪类:after
  &::after,&::before{
      content: " ";
        visibility: hidden;
        display: block;
```

```
        height: 0;
        clear: both;
    }
```

解析原理：
1）display:block 使生成的元素以块级元素显示,占满剩余空间;
2）height:0 避免生成内容破坏原有布局的高度。
3）visibility:hidden 使生成的内容不可见，并允许可能被生成内容盖住的内容可以进行点击和交互;
4）通过 content:"."生成内容作为最后一个元素，至于content里面是点还是其他都是可以的，例如oocss里面就有经典的 content:".",有些版本可能content 里面内容为空,一丝冰凉是不推荐这样做的,firefox直到7.0 content:”" 仍然会产生额外的空隙;
5）zoom：1 触发IE hasLayout。

通过分析发现，除了clear：both用来闭合浮动的，其他代码无非都是为了隐藏掉content生成的内容，这也就是其他版本的闭合浮动为什么会有font-size：0，line-height：0。

- JavaScript原型，原型链？有什么特点?

每个对象都会在其内部初始化一个属性，就是prototype(原型)，当我们访问一个对象的属性时，
如果这个对象内部不存在这个属性，那么他就会去prototype里找这个属性，这个prototype又会有自己的prototype，
于是就这样一直找下去，也就是我们平时所说的原型链的概念。
关系：instance.constructor.prototype = instance.__proto__

特点：
JavaScript对象是通过引用来传递的，我们创建的每个新对象实体中并没有一份属于自己的原型副本。当我们修改原型时，与之相关的对象也会继承这一改变。

当我们需要一个属性的时，Javascript引擎会先看当前对象中是否有这个属性， 如果没有的话，就会查找他的Prototype对象是否有这个属性，如此递推下去，一直检索到 Object 内建对象。

function Func(){} Func.prototype.name = "Sean"; Func.prototype.getInfo = function() { return this.name; } var person = new Func();//现在可以参考var person = Object.create(oldObject); console.log(person.getInfo());//它拥有了Func的属性和方法 //"Sean" console.log(Func.prototype); // Func { name="Sean", getInfo=function()}

- Javascript如何实现继承?

```javascript
1、构造继承
2、原型继承
3、实例继承
4、拷贝继承
```

原型prototype机制或apply和call方法去实现较简单，建议使用构造函数与原型混合方式。

```
function Parent(){
    this.name = 'wang';
}

function Child(){
    this.age = 28;
}
Child.prototype = new Parent();//继承了Parent，通过原型

var demo = new Child();
alert(demo.age);
alert(demo.name);//得到被继承的属性
```

- javascript创建对象的几种方式?

javascript创建对象简单的说,无非就是使用内置对象或各种自定义对象，当然还可以用JSON；但写法有很多种，也能混合使用。

1、对象字面量的方式

```
person={firstname:"Mark",lastname:"Yun",age:25,eyecolor:"black"};
```

2、用function来模拟无参的构造函数

```
function Person(){}
var person=new Person();//定义一个function，如果使用new"实例化",该function可以看作是一个Class
person.name="Mark";
person.age="25";
person.work=function(){
alert(person.name+" hello...");
}
person.work();
```

3、用function来模拟参构造函数来实现（用this关键字定义构造的上下文属性）

```
function Pet(name,age,hobby){
    this.name=name;//this作用域：当前对象
    this.age=age;
    this.hobby=hobby;
    this.eat=function(){
        alert("我叫"+this.name+",我喜欢"+this.hobby+",是个程序员");
    }
}
var maidou =new Pet("麦兜",25,"coding");//实例化、创建对象
maidou.eat();//调用eat方法
```

4、用工厂方式来创建（内置对象） var wcDog =new Object(); wcDog.name="旺财"; wcDog.age=3; wcDog.work=function(){ alert("我是"+wcDog.name+",汪汪汪……"); } wcDog.work();

5、用原型方式来创建 function Dog(){

} Dog.prototype.name="旺财"; Dog.prototype.eat=function(){ alert(this.name+"是个吃货"); } var wangcai =new Dog(); wangcai.eat();

6、用混合方式来创建 function Car(name,price){ this.name=name; this.price=price; } Car.prototype.sell=function(){ alert("我是"+this.name+"，我现在卖"+this.price+"万元"); } var camry =new Car("凯美瑞",27); camry.sell();

- 什么是闭包（closure），为什么要用它?

  ```javascript
    闭包是指有权访问另一个函数作用域中变量的函数，创建闭包的最常见的方式就是在一个函数内创建另一个函数，通过另一个函数访问这个函数的局部变量,利用闭包可以突破作用链域，将函数内部的变量和方法传递到外部。

    闭包的特性:
    1.函数内再嵌套函数
    2.内部函数可以引用外层的参数和变量
    3.参数和变量不会被垃圾回收机制回收

    //li节点的onclick事件都能正确的弹出当前被点击的li索引
     <ul id="testUL">
        <li> index = 0</li>
        <li> index = 1</li>
        <li> index = 2</li>
        <li> index = 3</li>
    </ul>
    <script type="text/javascript">
         var nodes = document.getElementsByTagName("li");
     for(i = 0;i<nodes.length;i+= 1){
         nodes[i].onclick = (function(i){
                   return function() {
                      console.log(i);
                   } //不用闭包的话，值每次都是4
                 })(i);
     }
    </script>

    执行say667()后,say667()闭包内部变量会存在,而闭包内部函数的内部变量不会存在
    使得Javascript的垃圾回收机制GC不会收回say667()所占用的资源
    因为say667()的内部函数的执行需要依赖say667()中的变量
    这是对闭包作用的非常直白的描述

      function say667() {
     // Local variable that ends up within closure
     var num = 666;
     var sayAlert = function() {
         alert(num);
  ```

```
    }
    num++;
    return sayAlert;
  }

  var sayAlert = say667();
  sayAlert()//执行结果应该弹出的667
```

- DOM操作——怎样添加、移除、移动、复制、创建和查找节点?

```
    (1) 创建新节点
     createDocumentFragment()    //创建一个DOM片段
     createElement()   //创建一个具体的元素
     createTextNode()   //创建一个文本节点
    (2) 添加、移除、替换、插入
     appendChild()
     removeChild()
     replaceChild()
     insertBefore() //在已有的子节点前插入一个新的子节点
    (3) 查找
     getElementsByTagName()    //通过标签名称
     getElementsByName()    //通过元素的Name属性的值(IE容错能力较强，会得到一个数组，其中包括id等于name值的)
     getElementById()    //通过元素Id，唯一性
```

- .call() 和 .apply() 的区别?

```
    例子中用 add 来替换 sub，add.call(sub,3,1) == add(3,1) ，所以运行结果为：alert(4);

    注意：js 中的函数其实是对象，函数名是对 Function 对象的引用。

     function add(a,b)
     {
         alert(a+b);
     }

     function sub(a,b)
     {
         alert(a-b);
     }

     add.call(sub,3,1);
```