回到顶部

1. webpack与grunt、gulp的不同?

三者都是前端构建工具，grunt 和 gulp 在早期比较流行，现在 webpack 相对来说比较主流，不过一些轻量化的任务还是会用 gulp 来处理，比如单独打包 CSS 文件等。

grunt 和 gulp 是基于任务和流（Task、Stream）的。类似 jQuery，找到一个（或一类）文件，对其做一系列链式操作，更新流上的数据， 整条链式操作构成了一个任务，多个任务就构成了整个 web 的构建流程。

webpack 是基于入口的。webpack 会自动地递归解析入口所需要加载的所有资源文件，然后用不同的 Loader 来处理不同的文件，用 Plugin 来扩展 webpack 功能。

所以总结一下：

从构建思路来说 gulp 和 grunt 需要开发者将整个前端构建过程拆分成多个`Task`，并合理控制所有`Task`的调用关系 webpack 需要开发者找到入口，并需要清楚对于不同的资源应该使用什么 Loader 做何种解析和加工

对于知识背景来说 gulp 更像后端开发者的思路，需要对于整个流程了如指掌 webpack 更倾向于前端开发者的思路

- 回到顶部

2.与webpack类似的工具还有哪些？谈谈你为什么最终选择（或放弃）使用webpack?

同样是基于入口的打包工具还有以下几个主流的：

- webpack
- rollup
- parcel

从应用场景上来看：

- webpack 适用于大型复杂的前端站点构建
- rollup 适用于基础库的打包，如 vue、react
- parcel 适用于简单的实验性项目，他可以满足低门槛的快速看到效果
- 由于 parcel 在打包过程中给出的调试信息十分有限，所以一旦打包出错难以调试，所以不建议复杂的项目使用 parcel
- 回到顶部

3.有哪些常见的Loader？他们是解决什么问题的？

- file-loader：把文件输出到一个文件夹中，在代码中通过相对 URL 去引用输出的文件
- url-loader：和 file-loader 类似，但是能在文件很小的情况下以 base64 的方式把文件内容注入到代码中去
- source-map-loader：加载额外的 Source Map 文件，以方便断点调试
- image-loader：加载并且压缩图片文件
- babel-loader：把 ES6 转换成 ES5
- css-loader：加载 CSS，支持模块化、压缩、文件导入等特性
- style-loader：把 CSS 代码注入到 JavaScript 中，通过 DOM 操作去加载 CSS。
- eslint-loader：通过 ESLint 检查 JavaScript 代码
- 回到顶部

.有哪些常见的Plugin？他们是解决什么问题的？

- define-plugin：定义环境变量
- commons-chunk-plugin：提取公共代码
- uglifyjs-webpack-plugin：通过UglifyES压缩ES6代码

- 回到顶部

Loader和Plugin的不同？

不同的作用

- Loader 直译为"加载器"。Webpack 将一切文件视为模块，但是 webpack 原生是只能解析 js 文件，如果想将其他文件也打包的话，就会用到 loader。 所以 Loader 的作用是让 webpack 拥有了加载和解析非 JavaScript 文件的能力。
- Plugin 直译为"插件"。Plugin 可以扩展 webpack 的功能，让 webpack 具有更多的灵活性。 在 Webpack 运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

不同的用法

- Loader 在 module.rules 中配置，也就是说他作为模块的解析规则而存在。 类型为数组，每一项都是一个 Object，里面描述了对于什么类型的文件（test），使用什么加载(loader)和使用的参数（options）
- Plugin 在 plugins 中单独配置。 类型为数组，每一项是一个 plugin 的实例，参数都通过构造函数传入。
- 回到顶部

6.webpack的构建流程是什么?从读取配置到输出文件这个过程尽量说全

Webpack 的运行流程是一个串行的过程，从启动到结束会依次执行以下流程：

1. 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
2. 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；

3. 确定入口：根据配置中的 entry 找出所有的入口文件；
4. 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
5. 完成模块编译：在经过第 4 步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；
6. 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
7. 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

在以上过程中，Webpack 会在特定的时间点广播出特定的事件，插件在监听到感 趣的事件后会执行特定的逻辑，并且插件可以调用 Webpack 提供的 API 改变
Webpack 的运行结果。

- 回到顶部

7.是否写过Loader和Plugin？描述一下编写loader或plugin的思路?

Loader 像一个"翻译官"把读到的源文件内容转义成新的文件内容，并且每个 Loader 通过链式操作，将源文件一步步翻译成想要的样子。

编写 Loader 时要遵循单一原则，每个 Loader 只做一种"转义"工作。 每个 Loader 的拿到的是源文件内容（source），可以通过返回值的方式将处理后的内容输出，也可以调用 this.callback()方法，将内容返回给 webpack。 还可以通过 this.async()生成一个 callback 函数，再用这个 callback 将处理后的内容输出出去。 此外 webpack 还为开发者准备了开发 loader 的工具函数集——loader-utils。

相对于 Loader 而言，Plugin 的编写就灵活了许多。 webpack 在运行的生命周期中会广播出许多事件，Plugin 可以监听这些事件，在合适的时机通过 Webpack 提供的 API 改变输出结果。

- 回到顶部

8.webpack的热更新是如何做到的？说明其原理?

webpack 的热更新又称热替换（Hot Module Replacement），缩写为 HMR。 这个机制可以做到不用刷新浏览器而将新变更的模块替换掉旧的模块。

原理：

首先要知道 server 端和 client 端都做了处理工作

1. 第一步，在 webpack 的 watch 模式下，文件系统中某一个文件发生修改，webpack 监听到文件变化，根据配置文件对模块重新编译打包，并将打包后的代码通过简单的 JavaScript 对象保存在内存中。
2. 第二步是 webpack-dev-server 和 webpack 之间的接口交互，而在这一步，主要是 dev-server 的中间件 webpack-dev-middleware 和 webpack 之间的交互，webpack-dev-middleware 调用 webpack 暴露的 API 对代码变化进行监控，并且告诉 webpack，将代码打包到内存中。
3. 第三步是 webpack-dev-server 对文件变化的一个监控，这一步不同于第一步，并不是监控代码变化重新打包。当我们在配置文件中配置了 devServer.watchContentBase 为 true 的时候，Server 会监听这些配置文件夹中静态文件的变化，变化后会通知浏览器端对应用进行 live reload。注意，这儿是浏览器刷新，和 HMR 是两个概念。
4. 第四步也是 webpack-dev-server 代码的工作，该步骤主要是通过 sockjs（webpack-dev-server 的依赖）在浏览器端和服务端之间建立一个 websocket 长连接，将 webpack 编译打包的各个阶段的状态信息告知浏览器端，同时也包括第三步中 Server 监听静态文件变化的信息。浏览器端根据这些 socket 消息进行不同的操作。当然服务端传递的最主要信息还是新模块的 hash 值，后面的步骤根据这一 hash 值来进行模块热替换。
5. webpack-dev-server/client 端并不能够请求更新的代码，也不会执行热更模块操作，而把这些工作又交回给了 webpack，webpack/hot/dev-server 的工作就是根据 webpack-dev-server/client

传给它的信息以及 dev-server 的配置决定是刷新浏览器呢还是进行模块热更新。当然如果仅仅是刷新浏览器，也就没有后面那些步骤了。

6. HotModuleReplacement.runtime 是客户端 HMR 的中枢，它接收到上一步传递给他的新模块的 hash 值，它通过 JsonpMainTemplate.runtime 向 server 端发送 Ajax 请求，服务端返回一个 json，该 json 包含了所有要更新的模块的 hash 值，获取到更新列表后，该模块再次通过 jsonp 请求，获取到最新的模块代码。这就是上图中 7、8、9 步骤。
7. 而第 10 步是决定 HMR 成功与否的关键步骤，在该步骤中，HotModulePlugin 将会对新旧模块进行对比，决定是否更新模块，在决定更新模块后，检查模块之间的依赖关系，更新模块的同时更新模块间的依赖引用。
8. 最后一步，当 HMR 失败后，回退到 live reload 操作，也就是进行浏览器刷新来获取最新打包代码。

- 回到顶部

9.如何利用webpack来优化前端性能？（提高性能和体验）

用webpack优化前端性能是指优化webpack的输出结果，让打包的最终结果在浏览器运行快速高效。

- 压缩代码。删除多余的代码、注释、简化代码的写法等等方式。可以利用 webpack 的 UglifyJsPlugin 和 ParallelUglifyPlugin 来压缩 JS 文件， 利用 cssnano（css-loader?minimize）来压缩 css
- 利用 CDN 加速。在构建过程中，将引用的静态资源路径修改为 CDN 上对应的路径。可以利用 webpack 对于 output 参数和各 loader 的 publicPath 参数来修改资源路径
- 删除死代码（Tree Shaking）。将代码中永远不会走到的片段删除掉。可以通过在启动 webpack 时追加参数--optimize-minimize 来实现
- 提取公共代码。
- 回到顶部

10.如何提高webpack的构建速度?

1. 多入口情况下，使用 CommonsChunkPlugin 来提取公共代码
2. 通过 externals 配置来提取常用库
3. 利用 DllPlugin 和 DllReferencePlugin 预编译资源模块 通过 DllPlugin 来对那些我们引用但是绝对不会修改的 npm 包来进行预编译，再通过 DllReferencePlugin 将预编译的模块加载进来。
4. 使用 Happypack 实现多线程加速编译
5. 使用 webpack-uglify-parallel 来提升 uglifyPlugin 的压缩速度。 原理上 webpack-uglify-parallel 采用了多核并行压缩来提升压缩速度
6. 使用 Tree-shaking 和 Scope Hoisting 来剔除多余代码

- 回到顶部

11.怎么配置单页应用？怎么配置多页应用?

单页应用可以理解为 webpack 的标准模式，直接在 entry 中指定单页应用的入口即可，这里不再赘述

多页应用的话，可以使用 webpack 的 AutoWebPlugin 来完成简单自动化的构建，但是前提是项目的目录结构必须遵守他预设的规范。 多页应用中要注意的是：

- 每个页面都有公共的代码，可以将这些代码抽离出来，避免重复的加载。比如，每个页面都引用了同一套 css 样式表
- 随着业务的不断扩展，页面可能会不断的追加，所以一定要让入口的配置足够灵活，避免每次添加新页面还需要修改构建配置
- 回到顶部

12.npm打包时需要注意哪些？如何利用webpack来更好的构建?

Npm 是目前最大的 JavaScript 模块仓库，里面有来自全世界开发者上传的可复用模块。你可能只是 JS 模块的使用者，但是有些情况你也会去选择上传自己开发的模块。 关于 NPM 模块上传的方法可以去官网上进行学习，这里只讲解如何利用 webpack 来构建。

NPM 模块需要注意以下问题：

1. 要支持 CommonJS 模块化规范，所以要求打包后的最后结果也遵守该规则。 Npm 模块使用者的环境是不确定的，很有可能并不支持 ES6，所以打包的最后结果应该是采用 ES5 编写的。并且如果 ES5 是经过转换的，请最好连同 SourceMap 一同上传。
2. Npm 包大小应该是尽量小（有些仓库会限制包大小） 发布的模块不能将依赖的模块也一同打包，应该让用户选择性的去自行安装。这样可以避免模块应用者再次打包时出现底层模块被重复打包的情况。
3. UI 组件类的模块应该将依赖的其它资源文件，例如.css 文件也需要包含在发布的模块里。
4. 基于以上需要注意的问题，我们可以对于 webpack 配置做以下扩展和优化：

CommonJS 模块化规范的解决方案：

1. 设置 output.libraryTarget='commonjs2'使输出的代码符合 CommonJS2 模块化规范，以供给其它模块导入使用
2. 输出 ES5 代码的解决方案：使用 babel-loader 把 ES6 代码转换成 ES5 的代码。再通过开启 devtool: 'source-map'输出 SourceMap 以发布调试。
3. Npm 包大小尽量小的解决方案：Babel 在把 ES6 代码转换成 ES5 代码时会注入一些辅助函数，最终导致每个输出的文件中都包含这段辅助函数的代码，造成了代码的冗余。解决方法是修改.babelrc 文件，为其加入 transform-runtime 插件
4. 不能将依赖模块打包到 NPM 模块中的解决方案：使用 externals 配置项来告诉 webpack 哪些模块不需要打包。
5. 对于依赖的资源文件打包的解决方案：通过 css-loader 和 extract-text-webpack-plugin 来实现，配置如下：

- 回到顶部

13.如何在vue项目中实现按需加载?

Vue UI 组件库的按需加载 为了快速开发前端项目，经常会引入现成的 UI 组件库如 ElementUI、iView 等，但是他们的体积和他们所提供的功能一样，是很庞大的。 而通常情况下，我们仅仅需要少量的几个组件就足够了，但是我们却将庞大的组件库打包到我们的源码中，造成了不必要的开销。

不过很多组件库已经提供了现成的解决方案，如 Element 出品的 babel-plugin-component 和 AntDesign 出品的 babel-plugin-import 安装以上插件后，在.babelrc 配置中或 babel-loader 的参数中进行设置，即可实现组件按需加载了。

单页应用的按需加载 现在很多前端项目都是通过单页应用的方式开发的，但是随着业务的不断扩展，会面临一个严峻的问题——首次加载的代码量会越来越多，影响用户的体验。

通过 import()*语句来控制加载时机，webpack 内置了对于 import()*的解析，会将 import()*中引入的模块作为一个新的入口在生成一个 chunk。 当代码执行到 import()*语句时，会去加载 Chunk 对应生成的文件。import()会返回一个 Promise 对象，所以为了让浏览器支持，需要事先注入 Promise polyfill

- 回到顶部

webpack 中如何使用 vue :

安装 vue 的包： cnpm i vue -S 由于 在 webpack 中，推荐使用 .vue 这个组件模板文件定义组件，所以，需要安装 能解析这种文件的 loader cnpm i vue-loader vue-template-complier -D 在 main.js 中，导入 vue 模块 import Vue from ‘vue’ 定义一个 .vue 结尾的组件，其中，组件有三部分组成：template script style 使用 import login from ‘./login.vue’ 导入这个组件 创建 vm 的实例 var vm =

new Vue({ el: '#app', render: c => c(login) }) new VueLoaderPlugin() 引入这个插件，必须的 在页面中创建一个 id 为 app 的 div 元素，作为我们 vm 实例要控制的区域；

- 回到顶部

Feach与axios的区别

fetch

优点

- 更加底层，提供的API丰富（request, response）
- 脱离了XHR，是ES规范里新的实现方式

缺点

- fetch 只对网络请求报错，对 400，500 都当做成功的请求，需要封装去处理
- fetch 默认不会带 cookie，需要添加配置项
- fetch 不支持 abort，不支持超时控制，使用 setTimeout 及 Promise.reject 的实现的超时控制并不能阻止请求过程继续在后台运行，造成了流量的浪费
- fetch 没有办法原生监测请求的进度，而 XHR 可以

axios

特点

- 从浏览器中创建 XMLHttpRequest
- 从 node.js 发出 http 请求
- 支持 Promise API
- 拦截请求和响应
- 转换请求和响应数据
- 取消请求
- 自动转换 JSON 数据
- 客户端支持防止 CSRF/XSRF
- 回到顶部

redux如何处理网络请求

1. redux-thunk 使用方法详见 readme。这个中间件的作用实际是给 redux 中的 dispatch 方法做了拓展，dispatch 方法本身是接收一个对象的，通过 redux-thunk，dispatch 方法可以接收一个函数并调用。这个中间件的好处在于我们我们可以在 action-creator 中去发送请求，而不用写在生命周期函数里面，还有很重要的一点就是方便进行单元测试。但是如果在请求过后需要对 response 进行复杂的数据处理，写在 action-creator 中会显得比较臃肿。第二种中间件就是为了处理这种状况的。
2. redux-saga 使用方法详见 readme。这个中间件实际是将所有异步请求放在一个文件中进行处理，如果有对请求数据进行操作的需求，建议放到这里，但是。。。saga 的入门门槛好像有点高，因为他使用到了 ES6 中的 generator 函数（说实话我在接触 saga 之前完全没有用过 generator 函数），另外就是在 error handle 方面要用 try-catch。以下可供参考：

```
function* getReq(){ let res = yield axios.get(url); let action =
initAction(res.data); yield put(action); }
```


- 回到顶部

es6里面的class和function

相同点

- 不管是 class 还是 function，constructor 属性默认不可枚举
- 都可通过实例的**proto**属性向原型添加方法
- 推荐使用 Object.getPrototypeOf()获取实例原型后再添加方法
- 所有原型方法属性都可用 Object.getOwnPropertyNames(Point.prototype)访问到

不同

- class 中定义的方法不可用 Object.keys(Point.prototype)枚举到
- function 构造器原型方法可被 Object.keys(Point.prototype)枚举到，除过 constructor
- function 会覆盖之前定义的方法
- class 会报错
- 在 function 定义的构造函数中，其 prototype.constructor 属性指向构造器自身
- 在 class 定义的类中，constructor 其实也相当于定义在 prototype 属性上
- class 没有变量提升
- class 定义的类没有私有方法和私有属性
- class 用类似于解构的方式获取原型上的方法

class 静态方法与静态属性

- class 定义的静态方法前加 static 关键字
- 只能通过类名调用
- 不能通过实例调用
- 可与实例方法重名
- 静态方法中的 this 指向类而非实例
- 静态方法可被继承
- 在子类中可通过 super 方法调用父类的静态方法 class 内部没有静态属性，只能在外面通过类名定义
- ES6 中当函数用 new 关键词的时候，增加了 new.target 属性来判断当前调用的构造函数。这个有什么用处呢？他可以限制函数的调用，比如一定要用 new 命令来调用，或者不能直接被实例化需要调用它的子类
- 回到顶部

canvas和svg的区别

canvas

canvas 是 html5 提供的新元素

- Canvas 通过 JavaScript 来绘制 2D 图形。 Canvas 是逐像素进行渲染的。
- 在 canvas 中，一旦图形被绘制完成，它就不会继续得到浏览器的关注。
- 如 果其位置发生变化，那么整个场景也需要重新绘制，包括任何或许已被图形覆盖的对 象。

特点

- 依赖分辨率
- 不支持事件处理器
- 弱的文本渲染能力
- 能够以 .png 或 .jpg 格式保存结果图像

- 最适合图像密集型的游戏，其中的许多对象会被频繁重绘

SVG

svg，所绘制的图形为矢量图，所以其用法上受到了限制。因为只能绘制矢量 图，所以 svg 中不能引入普通的图片，因为矢量图的不会失真的效果，在项目中我们 会用来做一些动态的小图标。但是由于其本质为矢量图，可以被无限放大而不会失 真，这很适合被用来做地图，而百度地图就是用 svg 技术做出来的.　　而 svg 里面的图 形可以被引擎抓取，支持事件的绑定。

svg 更多的是通过标签来来实现，如在 svg 中绘制正矩形形就要 用，这里我们不能用属性 style="width:XXX;height:XXX;”来定义

- SVG 是一种使用 XML 描述 2D 图形的语言。
- SVG 基于 XML，这意味着 SVG DOM 中的每个元素都是可用的。您可以为 某个元素附加 JavaScript 事件处理器。
- 在 SVG 中，每个被绘制的图形均被视为对象。如果 SVG 对象的属性发生变 化，那么浏览器能够自动重现图形。 特点
- 不依赖分辨率
- 支持事件处理器
- 最适合带有大型渲染区域的应用程序（比如谷歌地图）
- 复杂度高会减慢渲染速度（任何过度使用 DOM 的应用都不快）
- 不适合游戏应用

- 回到顶部

promise

Promise 必须为以下三种状态之一：等待态（Pending）、执行态（Fulfilled）和拒绝态（Rejected）。一旦 Promise 被 resolve 或 reject，不能再迁移至其他任何状态（即状态 immutable）

基本过程：

1. 初始化 Promise 状态（pending）
2. 执行 then(..) 注册回调处理数组（then 方法可被同一个 promise 调用多次）
3. 立即执行 Promise 中传入的 fn 函数，将 Promise 内部 resolve、reject 函数作为参数传递给 fn ，按事件机制时机处理
4. Promise 里的关键是要保证，then 方法传入的参数 onFulfilled 和 onRejected，必须在 then 方法被调用的那一轮事件循环之后的新执行栈中执行。

链式调用

这里最关键的点就是在 then 中新创建的 Promise，它的状态变为 fulfilled 的节点是在上一个 Promise 的回调执行完毕的时候。也就是说当一个 Promise 的状态被 fulfilled 之后，会执行其回调函数，而回调函数返回的结果会被当作 value，返回给下一个 Promise(也就是then 中产生的 Promise)，同时下一个 Promise的状态也会被改变(执行 resolve 或 reject)，然后再去执行其回调,以此类推下去...链式调用的效应就出来了

异常处理

异常通常是指在执行成功/失败回调时代码出错产生的错误，对于这类异常，我们使用 try-catch 来捕获错误，并将 Promise 设为 rejected 状态即可。

Promise.all

Promise.all([p1, p2, p3])用于将多个promise实例，包装成一个新的Promise实例，返回的实例就是普通的promise

- 它接收一个数组作为参数

- 数组里可以是 Promise 对象，也可以是别的值，只有 Promise 会等待状态改变
- 当所有的子 Promise 都完成，该 Promise 完成，返回值是全部值得数组
- 有任何一个失败，该 Promise 失败，返回值是第一个失败的子 Promise 结果

Promise.race

Promise.race() 类似于Promise.all() ，区别在于它有任意一个完成就算完成

Promise.then

1. 接收两个函数作为参数，分别代表 fulfilled（成功）和 rejected（失败）
2. .then()返回一个新的 Promise 实例，所以它可以链式调用
3. 当前面的 Promise 状态改变时，.then()根据其最终状态，选择特定的状态响应函数执行
4. 状态响应函数可以返回新的 promise，或其他值，不返回值也可以我们可以认为它返回了一个 null；
5. 如果返回新的 promise，那么下一级.then()会在新的 promise 状态改变之后执行
6. 如果返回其他任何值，则会立即执行下一级.then()

Promise.catch

- catch也会返回一个promise实例，并且是resolved状态

错误处理两种做法：
第一种：reject('错误信息').then(() => {}, () => {错误处理逻辑})
第二种：throw new Error('错误信息').catch(() => {错误处理逻辑}) 推荐使用第二种方式，更加清晰好读，并且可以捕获前面所有的错误（可以捕获 N 个 then 回调错误）

- 回到顶部

Generator函数

Generator 函数是 ES6 提供的一种异步编程解决方案，形式上也是一个普通函数，但有几个显著的特征.

1. function 关键字与函数名之间有一个星号 "*"（推荐紧挨着 function 关键字）
2. 函数体内使用 yield 表达式，定义不同的内部状态 （可以有多个 yield）
3. 直接调用 Generator 函数并不会执行，也不会返回运行结果，而是返回一个遍历器对象（Iterator Object）
4. 依次调用遍历器对象的 next 方法，遍历 Generator 函数内部的每一个状态
5. Generator 函数则没有执行而是返回一个 Iterator 对象，并通过调用 Iterator 对象的 next 方法来遍历，函数体内的执行看起来更像是“被人踢一脚才动一下”的感觉
6. Generator 函数，其中包含两个 yield 表达式和一个 return 语句（即产生了三个状态） 每次调用 Iterator 对象的 next 方法时，内部的指针就会从函数的头部或上一次停下来的地方开始执行，直到遇到下一个 yield 表达式或 return 语句暂停。换句话说，Generator 函数是分段执行的，yield 表达式是暂停执行的标记，而 next 方法可以恢复执行

yield 表达式

- yield 表达式只能用在 Generator 函数里面，用在其它地方都会报错
- yield 表达式如果用在另一个表达式中，必须放在圆括号里面
- yield 表达式用作参数或放在赋值表达式的右边，可以不加括号
- yield 表达式和 return 语句的区别
  相似：都能返回紧跟在语句后面的那个表达式的值

yield 与 return 区别

- 每次遇到 yield，函数就暂停执行，下一次再从该位置继续向后执行；而 return 语句不具备记忆位置的功能

- 一个函数只能执行一次 return 语句，而在 Generator 函数中可以有任意多个 yield

yield* 表达式

- 如果在 Generator 函数里面调用另一个 Generator 函数，默认情况下是没有效果的
- yield* 表达式用来在一个 Generator 函数里面 执行 另一个 Generator 函数

next() 方法的参数

- yield 表达式本身没有返回值，或者说总是返回 undefined。next 方法可以带一个参数，该参数就会被当作上一个 yield 表达式的返回值
- 这个当然是无效的，next 方法的参数表示上一个 yield 表达式的返回值，所以在第一次使用 next 方法时，传递参数是无效的。
- 从语义上讲，第一个 next 方法用来启动遍历器对象，所以不用带有参数。
- Generator 函数从暂停状态到恢复运行，它的上下文状态（context）是不变的。通过 next 方法的参数，就有办法在 Generator 函数开始运行之后，继续向函数体内部注入值。也就是说，可以在 Generator 函数运行的不同阶段，从外部向内部注入不同的值，从而调整函数行为。

Generator.prototype.return()
Generator 函数返回的遍历器对象，还有一个 return 方法，可以返回给定的值(若没有提供参数，则返回值的 value 属性为 undefined)，并且**终结**遍历 Generator 函数

与 Iterator 接口的关系

- ES6 规定，默认的 Iterator 接口部署在数据结构的 Symbol.iterator 属性，或者说，一个数据结构只要具有 Symbol.iterator 属性，就可以认为是“可遍历的”（iterable）。
- Symbol.iterator 属性本身是一个函数，就是当前数据结构默认的遍历器生成函数。执行这个函数，就会返回一个遍历器。
- 由于执行 Generator 函数实际返回的是一个遍历器，因此可以把 Generator 赋值给对象的 Symbol.iterator 属性，从而使得该对象具有 Iterator 接口。
- 回到顶部

babel

核心包

- babel-core：babel 转译器本身，提供了 babel 的转译 API，如 babel.transform 等，用于对代码进行转译。像 webpack 的 babel-loader 就是调用这些 API 来完成转译过程的。
- babylon：js 的词法解析器
- babel-traverse：用于对 AST（抽象语法树，想了解的请自行查询编译原理）的遍历，主要给 plugin 用
- babel-generator：根据 AST 生成代码

功能包

- babel-types：用于检验、构建和改变 AST 树的节点
- babel-template：辅助函数，用于从字符串形式的代码来构建 AST 树节点
- babel-helpers：一系列预制的 babel-template 函数，用于提供给一些 plugins 使用
- babel-code-frames：用于生成错误信息，打印出错误点源代码帧以及指出出错位置
- babel-plugin-xxx：babel 转译过程中使用到的插件，其中 babel-plugin-transform-xxx 是 transform 步骤使用的
- babel-preset-xxx：transform 阶段使用到的一系列的 plugin
- babel-polyfill：JS 标准新增的原生对象和 API 的 shim，实现上仅仅是 core-js 和 regenerator-runtime 两个包的封装
- babel-runtime：功能类似 babel-polyfill，一般用于 library 或 plugin 中，因为它不会污染全局作用域

工具包

- babel-cli：babel 的命令行工具，通过命令行对 js 代码进行转译
- babel-register：通过绑定 node.js 的 require 来自动转译 require 引用的 js 代码文件

babel的配置

使用形式，如果是以命令行方式使用 babel，那么 babel 的设置就以命令行参数的形式带过去；
还可以在 package.json 里在 babel 字段添加设置； 但是建议还是使用一个单独的.babelrc 文件，把 babel 的设置都放置在这里，
所有 babel API 的 options（除了回调函数之外）都能够支持，具体的 options 见 babel 的 API options 文档

常用options字段说明

- env：指定在不同环境下使用的配置。比如 production 和 development 两个环境使用不同的配置，就可以通过这个字段来配置。
  env 字段的从 process.env.BABEL_ENV 获取，如果 BABEL_ENV 不存在，则从 process.env.NODE_ENV 获取，如果 NODE_ENV 还是不存在，则取默认值"development"
- plugins：要加载和使用的插件列表，插件名前的 babel-plugin-可省略；plugin 列表按从头到尾的顺序运行
- presets：要加载和使用的 preset 列表，preset 名前的 babel-preset-可省略；presets 列表的 preset 按从尾到头的逆序运行（为了兼容用户使用习惯）
  同时设置了 presets 和 plugins，那么 plugins 的先运行；每个 preset 和 plugin 都可以再配置自己的 option

babel的工作原理

babel 是一个转译器，感觉相对于编译器 compiler，叫转译器 transpiler 更准确，因为它只是把同种语言的高版本规则翻译成低版本规则，而不像编译器那样，输出的是另一种更低级的语言代码。 但是和编译器类似，babel 的转译过程也分为三个阶段：parsing、transforming、generating，以 ES6 代码转译为 ES5 代码为例，babel 转译的具体过程如下：

```
ES6代码输入 ==》 babylon进行解析 ==》 得到AST==》 plugin用babel-traverse对AST树进行遍历转译 ==》
得到新的AST树==》 用babel-generator通过AST树生成ES5代码
```

- 回到顶部