一、Git

1. git 和 svn 的区别

- git 和 svn 最大的区别在于 git 是分布式的，而 svn 是集中式的。因此我们不能再离线的情况下使用 svn。如果服务器出现问题，就没有办法使用 svn 来提交代码。
- svn 中的分支是整个版本库的复制的一份完整目录，而 git 的分支是指针指向某次提交，因此 git 的分支创建更加开销更小并且分支上的变化不会影响到其他人。svn 的分支变化会影响到所有的人。
- svn 的指令相对于 git 来说要简单一些，比 git 更容易上手。
- **GIT把内容按元数据方式存储，而SVN是按文件：**因为git目录是处于个人机器上的一个克隆版的版本库，它拥有中心版本库上所有的东西，例如标签，分支，版本记录等。
- **GIT分支和SVN的分支不同：**svn会发生分支遗漏的情况，而git可以同一个工作目录下快速的在几个分支间切换，很容易发现未被合并的分支，简单而快捷的合并这些文件。
- **GIT没有一个全局的版本号，而SVN有**
- **GIT的内容完整性要优于SVN：**GIT的内容存储使用的是SHA-1哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏

2. 经常使用的 git 命令?

```
git init                    // 新建 git 代码库
git add                     // 添加指定文件到暂存区
git rm                      // 删除工作区文件，并且将这次删除放入暂存区
git commit -m [message]     // 提交暂存区到仓库区
git branch                  // 列出所有分支
git checkout -b [branch]    // 新建一个分支，并切换到该分支
git status                  // 显示有变更文件的状态
```

3. git pull 和 git fetch 的区别

- git fetch 只是将远程仓库的变化下载下来，并没有和本地分支合并。
- git pull 会将远程仓库的变化下载下来，并和当前分支合并。

4. git rebase 和 git merge 的区别

git merge 和 git rebase 都是用于分支合并，关键**在 commit 记录的处理上不同**:

- git merge 会新建一个新的 commit 对象，然后两个分支以前的 commit 记录都指向这个新 commit 记录。这种方法会保留之前每个分支的 commit 历史。
- git rebase 会先找到两个分支的第一个共同的 commit 祖先记录，然后将提取当前分支这之后的所有 commit 记录，然后将这个 commit 记录添加到目标分支的最新提交后面。经过这个合并后，两个分支合并后的 commit 记录就变为了线性的记录了。

二、Webpack

1. **webpack**与**grunt**、**gulp**的不同?

Grunt、Gulp是基于任务运行的工具**： 它们会自动执行指定的任务，就像流水线，把资源放上去然后通过不同插件进行加工，它们包含活跃的社区，丰富的插件，能方便的打造各种工作流。

Webpack是基于模块化打包的工具: 自动化处理模块，webpack把一切当成模块，当 webpack 处理应用程序时，它会递归地构建一个依赖关系图 (dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。

因此这是完全不同的两类工具,而现在主流的方式是用npm script代替Grunt、Gulp，npm script同样可以打造任务流。

2. **webpack**、**rollup**、**parcel**优劣?

- webpack适用于大型复杂的前端站点构建: webpack有强大的loader和插件生态,打包后的文件实际上就是一个立即执行函数，这个立即执行函数接收一个参数，这个参数是模块对象，键为各个模块的路径，值为模块内容。立即执行函数内部则处理模块之间的引用，执行模块等,这种情况更适合文件依赖复杂的应用开发。
- rollup适用于基础库的打包，如vue、d3等: Rollup 就是将各个模块打包进一个文件中，并且通过 Tree-shaking 来删除无用的代码,可以最大程度上降低代码体积,但是rollup没有webpack如此多的的如代码分割、按需加载等高级功能，其更聚焦于库的打包，因此更适合库的开发。
- 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,但是生态差、报错信息不够全面都是他的硬伤，除了一些玩具项目或者实验项目不建议使用。

3. 有哪些常见的**Loader**?

- 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

注意：在Webpack中，loader的执行顺序是从右向左执行的。因为webpack选择了**compose这样的函数式编程方式**，这种方式的表达式执行是从右向左的。

4. 有哪些常见的**Plugin**?

- define-plugin：定义环境变量
- html-webpack-plugin：简化html文件创建
- uglifyjs-webpack-plugin：通过 UglifyES 压缩 ES6 代码
- webpack-parallel-uglify-plugin: 多核压缩，提高压缩速度
- webpack-bundle-analyzer: 可视化webpack输出文件的体积
- mini-css-extract-plugin: CSS提取到单独的文件中，支持按需加载

5. **bundle**，**chunk**，**module**是什么?

- bundle：是由webpack打包出来的文件；
- chunk：代码块，一个chunk由多个模块组合而成，用于代码的合并和分割；

- module：是开发中的单个模块，在webpack的世界，一切皆模块，一个模块对应一个文件，webpack会从配置的 entry中递归开始找出所有依赖的模块。

6. **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。

7. **webpack**的构建流程****?**

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，插件在监听到感兴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。

8. 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。

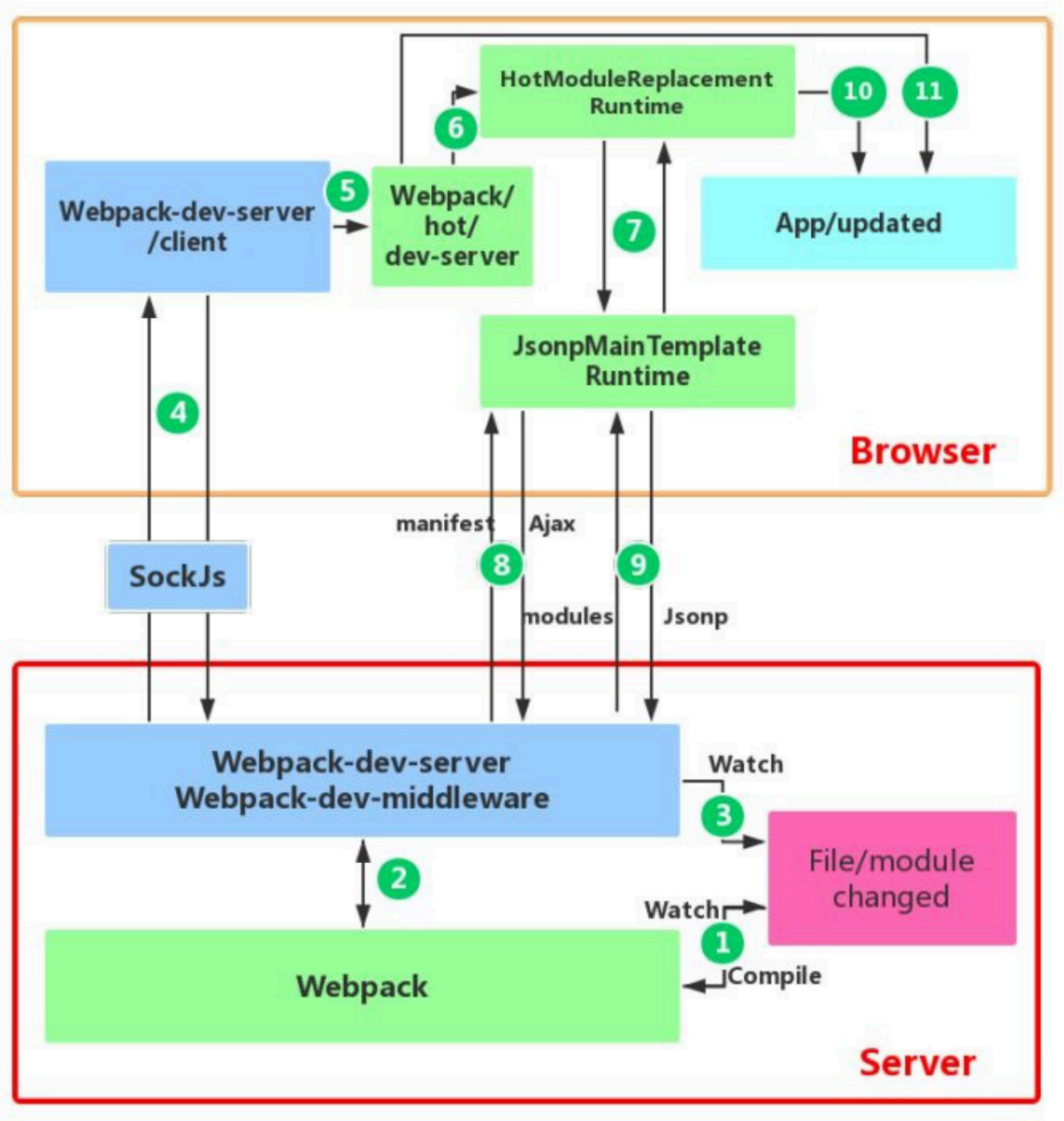
编写Loader时要遵循单一原则，每个Loader只做一种"转义"工作。 每个Loader的拿到的是源文件内容（source），可以通过返回值的方式将处理后的内容输出，也可以调用 this.callback() 方法，将内容返回给webpack。 还可以通过this.async() 生成一个 callback 函数，再用这个callback将处理后的内容输出出去。 此外webpack 还为开发者准备了开发loader的工具函数集——loader-utils 。

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。

9. **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。

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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。
7. 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。

10. 如何用**webpack**来优化前端性能?

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
- **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
- **Code Splitting:** 将代码按路由维度或者组件分块(chunk),这样做到按需加载,同时可以充分利用浏览器缓存
- **提取公共第三方库**: SplitChunksPlugin插件来进行公共模块抽取,利用浏览器缓存可以长期缓存这些无需频繁变动的公共代码

11. 如何提高**webpack**的打包速度**?**

- happypack: 利用进程并行编译loader,利用缓存来使得 rebuild 更快,遗憾的是作者表示已经不会继续开发此项目,类似的替代者是thread-loader
- 外部扩展(externals): 将不怎么需要更新的第三方库脱离webpack打包，不被打入bundle中，从而减少打包时间，比如jQuery用script标签引入

- dll: 采用webpack的 DllPlugin 和 DllReferencePlugin 引入dll，让一些基本不会改动的代码先打包成静态资源，避免反复编译浪费时间
- 利用缓存: webpack.cache 、babel-loader.cacheDirectory、 HappyPack.cache 都可以利用缓存提高rebuild效率缩小文件搜索范围: 比如babel-loader插件,如果你的文件仅存在于src中,那么可以 include: path.resolve(__dirname,'src') ,当然绝大多数情况下这种操作的提升有限，除非不小心build了node_modules文件

12. 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

13. 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:

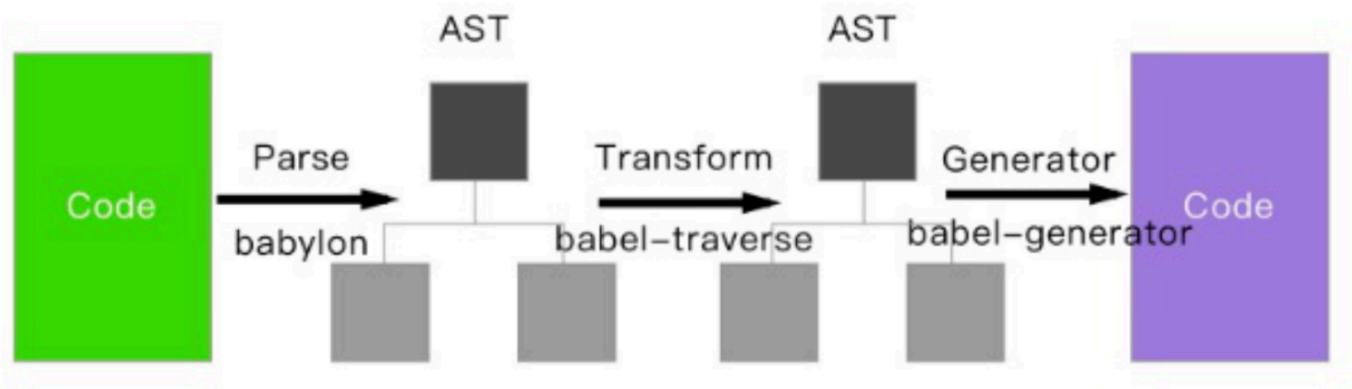
- 每个页面都有公共的代码，可以将这些代码抽离出来，避免重复的加载。比如，每个页面都引用了同一套css样式表
- 随着业务的不断扩展，页面可能会不断的追加，所以一定要让入口的配置足够灵活，避免每次添加新页面还需要修改构建配置

三、其他

1. Babel的原理是什么**?**

babel 的转译过程也分为三个阶段，这三步具体是:

- **解析 Parse**: 将代码解析生成抽象语法树（AST），即词法分析与语法分析的过程;
- **转换 Transform**: 对于 AST 进行变换一系列的操作，babel 接受得到 AST 并通过 babel-traverse 对其进行遍历，在此过程中进行添加、更新及移除等操作;
- **生成 Generate**: 将变换后的 AST 再转换为 JS 代码, 使用到的模块是 babel-generator。


AST
AST
Code
Parse
babylon
Transform
babel-traverse
Generator
babel-generator
Code