1. npm与包

1.1包管理配置文件

npm规定，在项目根目录里面，必须提供一个叫做`package.json`的包管理配置文件。用来记录一些与项目相关的配置信息，例如：

- 项目的名称，版本号，描述等
- 项目中都用到了哪些包
- 哪些包只在开发期间会用到
- 哪些包在开发和部署时都需要用到

1. 多人协作的问题

遇到的问题：第三方包的体积过大，不方便团队成员之间共享项目源代码

解决方案：共享时剔除`node_moudles`

2. 如何记录在项目中安装了哪些包

在项目目录中，创建一个叫做`package.json`的配置文件，即可用来记录项目中安装了哪些包，从而方便剔除`node_moudles`目录后，在团队成员之间共享项目的源代码

注意：今后的项目开发中，一定要把`node_moudles`文件夹，添加到`.gitignore`忽略文件中

3. 快速创建`package.json`

npm包管理工具提供了一个快捷命令，可以在执行命令时所处的目录中，快速创建`package.json`这个包管理配置文件：

```
npm init -y
```

注意：

- 上述命令只能在英文的目录下成功运行，所以，项目文件夹下的名称一定要使用英文名称，不要使用中文，不能出现空格
- 运行`npm install`命令安装包的时候，npm包管理工具会自动把包的名称和版本号，记录到`package.json`中

4. `dependencies`节点

`package.json`文件中，有一个`dependencies`节点，专门用来记录使用`npm install`命令安装了哪些包

5. 一次性的安装所有的包

当我们拿到一个剔除了`node_moudles`的项目之后，需要先把所有的包下载到项目中，才能将项目运行起来
否则会出现类似下面的错误：

```
Error:Cannot find moudle 'moment'
```

可以运行`npm install`命令（或者`npm i`）一次性安装所有的包：

```
npm install
```

6. 卸载包

可以运行`npm uninstall`命令，来卸载指定的包

```
// 使用 npm uninstall 具体的包名  来卸载包
npm uninstall moment
```

注意：`npm uninstall`命令执行成功之后，会把卸载的包，自动从`package.json`的`dependencies`中移除掉

7. `devDependencies`节点

如果某些包只在项目开发阶段会用到，在项目上线之后不会用到，则建议把这些包记录到`devDependencies`节点中。
与之对应的，如果某些包在开发和项目上线之后都需要用到，则建议把这些包记录到`dependencies`节点中

使用如下命令，将包记录到`devDependencies`节点中

```
//简略写法：
npm i 包名 -D

//完整写法：
npm install 包名 --save-dev
```

1.2. 解决下包速度慢的问题

1. 为什么下包速度慢

在使用npm下包的时候，默认从国外的服务器进行下载，因此下包速度会很慢

2. 淘宝npm镜像服务器

淘宝在国内搭建了一个服务器，专门把国外官方服务器上的包同步到国内的服务器，然后在国内提供下包的服务。从而极大地提高了下包的速度

镜像
镜像(Mirroring)是一种文件存储形式，一个磁盘上的数据在另一个磁盘上存在一个完全相同的副本即为镜像

3. 切换npm下包镜像源

下包的镜像源，指的就是下包的服务器地址

查看当前下包镜像源：

```
npm config get registry
```

将下包的镜像源切换为淘宝镜像源

```
npm config set registry=https://registry.npm.taobao.org/
```

切换回官方服务器：

```
npm config set registry=https://registry.npmjs.org/
```

4. nrm

为了更方便的切换下包的镜像源，我们可以安装`nrm`这个小工具，利用`nrm`提供的终端命令，可以快速查看和切换下包的镜像源

```
// 通过npm包管理器，将nrm安装为全局可用工具
npm i nrm -g

// 查看所有可用的镜像源
nrm ls

//将下包的镜像源切换为toabao镜像
nrm use taobao
```

1.3. 包的分类

使用`npm`包管理工具下载的包，可以分为两大类，分别是：

- 项目包
- 全局包

1. 项目包

> 那些被安装到项目的`node_modules`目录中的包，都是项目包

项目包又可以分为2类：

- 开发依赖包（被记录到`devDependencies`节点的包，只会在开发时用到
- 核心依赖包（被记录到`dependencies`节点中的包，在开发期间和项目上线后都会用到）

2. 全局包

> 在执行`npm install`命令时，如果提供了`-g`参数，则会把包安装为全局包

全局包默认会被安装到`C:\Users\用户目录\AppData\Roaming\npm\node_modules`下

```
npm i 包名 -g #全局安装指定的包
npm uninstall 包名 -g  #卸载全局安装的包
```

注意：

- 只有**工具性质**的包，才有全局安装的必要性，因为它们提供了好用的终端命令
- 判断某个包是否需要全局安装后才能使用，可以参考官方提供的使用说明即可

3. i5ting_toc

`i5ting_toc`是一个把md文档转换为HTML页面的小工具，使用步骤如下：

```
//将i5ting_toc安装为全局包
npm install -g i5ting_toc

//调用i5ting_toc 轻松实现md转HTML功能
i5ting_toc -f 要转换的md文件路径 -o
```

1.4. 规范的包结构

在清楚了包的概念，以及如何下载和使用包之后，深入了解一下包的内部结构

一个规范的包，它的组成结构，必须符合以下3点要求：

- 包必须以单独的目录而存在
- 包的顶级目录下必须要包含`package.json`这个包管理配置文件
- `package.json`中必须要包含`name`，`version`，`main`这三个属性，分别代表包的名字、版本号、包的入口

1.5. 开发属于自己的包

1. 要实现的功能

- 格式化日期
- 转义HTML中的特殊字符
- 还原HTML的特殊字符

2. 初始化包的基本结构

- 新建`itheima-tools`文件夹，作为包的根目录
- 在`itheima-tools`文件夹中新建三个文件：
  - `package.json`(包管理配置文件)
  - `index.js`（包的入口文件）
  - `README.md`(包的说明文档)

3. 初始化 `package.json`

```
{
    "name": "itheima-tools-wjq",
    "veersion": "1.0.0",
    "main": "index.js",
    "description": "提供了格式化时间、HTMLEscape相关的功能",
    "keywords": ["itheima", "dateFormat", "escape"],
    "license": "ISC"
}
```

4. 在`index.js`中定义格式化时间的方法

```
// 这是包的入口文件

// 定义格式化时间的函数
function dateFormat(dateStr) {
    const dt = new Date(dateStr)

    const y = dt.getFullYear()
    const m = padZero(dt.getMonth() + 1)
    const d = padZero(dt.getDate())

    const hh = padZero(dt.getHours())
    const mm = padZero(dt.getMinutes())
    const ss = padZero(dt.getSeconds())


    return `${y}-${m}-${d} ${hh}:${mm}:${ss}`
}

// 定义一个补零的函数
function padZero(n) {
    return n > 9 ? n : '0' + n
}

// 向外暴露需要的成员
module.exports = {
    dateFormat
}
```

5. 在`index.js`中定义转义HTML的方法

```
// 定义转义HTML字符的函数
function htmlEscape(htmlstr) {
    return htmlstr.replace(/<|>|"|&/g, (match) => {
        switch (match) {
            case '<':
                return "&lt;"
            case '>':
                return '&gt;'
            case '"':
                return '&quot;'
            case '&':
                return '&amp;'
        }
    })
}

// 向外暴露需要的成员
module.exports = {
    dateFormat,
    htmlEscape
}
```

6. 在`index.js`中定义还原HTML的方法

```
// 定义还原HTML字符串的函数
function htmlUnEscape(str){
    return str.replace(/&lt;|&gt;|&quot;|&amp;/g,(match)=>{
        switch(match){
            case '&lt;':
                return '<'
            case '&gt;':
                return '>'
            case '&quot;':
                return '"'
            case '&amp;':
                return '&'
        }
    })
}

// 向外暴露需要的成员
module.exports = {
    dateFormat,
    htmlEscape,
    htmlUnEscape
}
```

6. 将不同的功能进行模块化拆分

- 将格式化时间的功能，拆分到`src->dateFormat.js`中
- 将处理HTML字符串的功能，拆分到`src->htmlEscape.js`中
- 在`index.js`中，导入两个模块，得到需要向外共享的方法
- 在`index.js`中，使用`module.exports`把对应的方法共享出去

```
// 这是包的入口文件
const date = require('./src/dateFormat')
const escape = require('./src/htmlEscape')

// 向外暴露需要的成员
module.exports = {
    ...date,
    ...escape
}
```

8. 编写包的说明文档

> README.md：包的说明文档

README.md会包含以下六项内容：

- 安装方式
- 导入方式
- 格式化时间用法
- 转义HTML特殊字符的用法
- 还原HTML特殊字符的用法
- 开源协议

1.5. 发布包

1. 注册npm账号

在https://www.npmjs.com/上注册账号，需要邮箱验证

2. 登陆npm账号

npm注册完成后，可以在终端执行`npm login`命令，依次输入用户名、密码、邮箱后，即可登陆成功

注意：在运行`npm login`命令之前，必须先把下包的服务器地址切换为npm官方服务器，否则会发布包失败

3. 把包发布到npm上

将终端切换到包的根目录之后，运行`npm publish`命令，即可将包发布到npm上（注意：包名不能相同）

删除已发布的包

运行`npm unpublish 包名 --force`命令，即可从npm删除已发布的包

注意：

- npm unpublish 命令只能删除72小时内发布的包
- npm unpublish删除的包，在24小时内不许重复发布

2. 模块的加载机制

2.1. 优先从缓存中加载

模块在第一次加载后会被缓存，这也意味着多次调用`require()`不会导致模块的代码被执行多次

注意：无论是内置模块，用户自定义模块还是第三方模块，他们都会优先从缓存中加载，从而提高模块的加载效率

2.2. 内置模块的加载机制

> 内置模块是由`Node.js`官方提供的模块，内置模块的加载优先级最高

例如：`require('fs')`始终返回fs模块，即使在`node_modules`目录下有名字相同的包也叫js

2.3. 自定义模块的加载机制

使用`require()`加载自定义模块时，必须指定以`./`或者`../`开头的**路径标识符**。在加载自定义模块时，如果没有指定`./`或者`../`这样的路径标识符，则node会把他当做内置模块或者第三方模块加载

同时，在使用`require()`导入自定义模块时，如果省略了文件的扩展名，则Node.js会按照顺序分别尝试加载以下文件：

- 按照确切的文件名进行加载
- 补全.js扩展名进行加载
- 补全.json扩展名进行加载
- 补全.node扩展名进行加载
- 加载失败，终端报错

2.4. 第三方模块的加载机制

如果传递给`require()`的模块标识符不是一个内置模块，也没有以`./`或者`../`开头，则Node.js会从当前模块的父目录开始，尝试从`/node_moudles`文件夹中加载第三方模块
如果没有找到对应的模块，则移动到再上一层父目录中，进行加载，知道文件系统的根目录

2.5. 目录作为模块

当把目录作为模块标识符，传递给`require()`进行加载的时候，有三种加载模式：

- 在被加载的目录中查找一个叫做`package.json`的文件，并寻找`main`属性，作为`require()`加载的入口
- 如果目录里面没有`package.json`文件，或者`main`入口不存在或者无法解析，则Node.js会视图加载目录下的`index.js`文件
- 如果以上两步都失败了，则Node.js会在终端打印错误消息，报告模块的缺失：`Error:Cannot find module 'xxx'`

3. 初识Express

3.1. 什么是Express

官方给出的概念：`Express`是基于Node.js平台，快速、开放、极简的`web开发框架`

通俗的理解：`Express`的作用和Node.js内置的`http`模块类似，是专门用来创建web服务器的

Express的本质：就是一个npm上的第三方的包，提供了快速创建web服务器的便捷方法

使用`Express`，我们可以方便、快速的创建web服务器或者API接口服务器

3.2. Express的基本使用

1. 安装

在项目所处的目录中，运行如下的终端命令，即可将express安装到项目中使用

```
npm i express@4.17.1
```

2. 创建基本的web服务器

```
// 1. 导入express
const express=require('express')

// 2. 创建web服务器
const app=express()

// 3. 启动web服务器
app.listen(80,()=>{
    console.log('express server running at http://127.0.0.1');
})
```

3. 监听get请求

通过`app.get()`方法，可以监听客户端的get请求，具体的语法格式如下：

```
app.get('/user',(req,res)=>{
    // 调用express提供的res.send()方法，向客户端响应一个JSON对象
    res.send({name:'海绵宝宝',age:3,gender:'male'})
})
```

4. 监听post请求

通过`app.post()`方法，可以监听客户端的get请求，具体的语法格式如下：

```
app.post('/user',(req,res)=>{
    // 调用express提供的res.send()方法，向客户端响应一个文本字符串
    res.send('request successed')
})
```

5. 把内容响应给客户端

通过`res.send()`方法，可以把处理好的内容，发送给客户端：

```
app.get('/user',(req,res)=>{
    // 调用express提供的res.send()方法，向客户端响应一个JSON对象
    res.send({name:'海绵宝宝',age:3,gender:'male'})
})

app.post('/user',(req,res)=>{
    // 调用express提供的res.send()方法，向客户端响应一个文本字符串
    res.send('request successed')
})
```

6. 获取URL中携带的查询参数

通过`req.query`对象，可以访问到客户端通过查询字符串的方式，发送到服务器的参数：

```
app.get('/',(req,res)=>{
    // 通过req.query可以获取到客户端发送过来的查询参数
    console.log(req.query)
    res.send(req.query)
})
```

7. 获取URL中的动态参数

通过`req.params`对象，可以访问到URL中，通过`:`匹配到的动态参数：

```
// 注意：这里的：id是一个动态的参数
app.get('/user/:id',(req,res)=>{
    // req.params是动态匹配到的URL参数，默认是一个空对象
    console.log(req.params);
    res.send(req.params)
})
```

3.3. 托管静态资源

1. express.static()

express提供了一个非常好用的函数，叫做`express.static()`，通过它，我们可以非常方便的创建一个静态资源服务器，
例如：通过如下代码就可以将public目录下的图片，css文件，JavaScript文件对外开放访问了：

```
app.use(express.static('public'))
```

现在就可以访问public目录下所有的文件了

注意：Express在指定的静态目录中查找文件，并对外提供资源的访问路径。因此，存放静态资源的目录名下不会出现在URL中

```
const express=require('express')

const app=express()

// 在这里，调用express.static()方法，快速的对外提供静态资源
app.use(express.static('./clock'))

app.listen(80,()=>{
    console.log('express server running at http://127.0.0.1');
})
```

2. 托管多个静态资源目录

如果要托管多个静态资源目录，请多次调用`express.static()`函数：

```
app.use(express.static('public'))
app.use(express.static('files'))
```

访问静态资源文件时，`express.static()`函数会根据目录的添加顺序查找所需的文件

3. 挂载路径的前缀

如果希望在托管的静态资源访问路径之前，挂载路径前缀，则可以使用如下的方式：

```
app.use('/public',express.static('public'))
```

3.4. nodemon

1. 为什么要使用nodemon

在编写调试NOde.js时，如果修改了项目的代码，则需要频繁手动的close掉，然后再重新启动，非常繁琐
现在，我们可以使用`nodemon`这个工具，它能够监听项目文件的变动，当代码被修改后，`nodemon`会**自动帮我们重启项目**，极大地方便了开发和调试

2. 安装nodemon

在终端中，运行如下命令，即可将`nodemon`安装为全局可用工具

```
npm install -g nodemon
```

3. 使用nodemon

使用nodemon命令代替node命令，使用`nodemon app.js`来启动项目。这样做的好处是：代码被修改后，会被nodemon监听到，从而实现自动重启项目的效果