

YTM项目第4周上线

本周的任务是了解Mangodb数据库，同学完成之后可以自己去学习下Koa框架。

NPM项目构建

我们学习了NPM的基本用法，即安装和介绍。在一些大型项目中，例如Vue和其他开源前端模板，我们只需要执行两行命令：

```
git clone xxx
```

```
npm i
```

就可以快速安装和构建项目所需的依赖，并可以使用`npm run dev`等命令直接启动项目。

例如，使用`npm run dev`命令直接启动项目，这是如何实现的？

构建Node项目依赖于一个名为`package.json`的文件，`npm`执行的大部分任务与该文件有关。

描述 the-package-json-guide

`package-lock.json` 是在 `npm` 修改 `node_modules` 目录或 `package.json` 文件时自动生成的。它描述了生成的确切依赖树，使得后续的安装能够生成完全相同的依赖树，无论中间依赖的更新情况如何。

该文件的目的是将其提交到源代码库中，它有以下几个作用：

- 描述一种依赖树的唯一表示，以确保团队成员、部署和持续集成能够安装完全相同的依赖项。
- 提供一种功能，让用户可以在不提交整个 `node_modules` 目录的情况下“时光旅行”到先前的状态。
- 通过可读的源代码差异提高对依赖树更改的可见性。
- 通过允许 `npm` 跳过对先前安装的软件包重复解析元数据来优化安装过程。
- 自 `npm v7` 起，锁定文件包含足够的信息以获取完整的依赖树图，减少了读取 `package.json` 文件的需要，并实现了显著的性能改进。

package-lock.json vs npm-shrinkwrap.json

这两个文件具有相同的格式，并在项目的根目录中执行类似的功能。

区别在于 `package-lock.json` 不能被发布，如果在根目录以外的任何位置发现它，将被忽略。

相反，[`npm-shrinkwrap.json`](#) 允许发布，并从遇到的位置开始定义依赖树。除非部署 CLI 工具或以其他方式使用发布流程生成生产包，否则不建议使用它。

如果在项目的根目录中同时存在 `package-lock.json` 和 `npm-shrinkwrap.json`，`npm-shrinkwrap.json` 将优先，`package-lock.json` 将被忽略。

隐藏的锁定文件

为了避免重复处理 `node_modules` 文件夹，`npm` 从 v7 版开始使用位于 `node_modules/.package-lock.json` 中的“隐藏”锁定文件。它包含有关依赖树的信息，并在读取整个 `node_modules` 层次结构之前使用它，前提是满足以下条件：

- 所有它引用的软件包文件夹都存在于 `node_modules` 层次结构中。
- `node_modules` 层次结构中不存在任何不在锁定文件中列出的软件包文件夹。
- 文件的修改时间至少与它引用的所有软件包文件夹的修改时间相同。

也就是说，只有在它是与最近更新的软件包树的一部分创建的情况下，隐藏的锁定文件才是相关的。如果其他 CLI 在任何方面改变了树，则会检测到这一点，隐藏的锁定文件将被忽略。

请注意，可以手动更改软件包的 内容，而不影响软件包文件夹的修改时间。例如，如果您向 `node_modules/foo/lib/bar.js` 添加一个文件，则 `node_modules/foo` 的修改时间不会反映此更改。如果您要手动编辑 `node_modules` 中的文件，通常最好删除 `node_modules/.package-lock.json` 中的文件。

由于旧的 `npm` 版本会忽略隐藏的锁定文件，因此它不包含“常规”锁定文件中存在的向后兼容性功能。也就是说，它的 `lockfileVersion` 是 3，而不是 2。

处理旧的锁定文件

当 `npm` 在包安装过程中检测到来自 `npm v6` 或之前的锁定文件时，它会自动更新锁定文件，以从 `node_modules` 树或（在 `node_modules` 树为空或非常旧的锁定文件格式的情况下）`npm` 注册表中获取缺失的信息。

文件格式

name

这是该包的名称，与 `package.json` 中的名称相匹配。

version

这是该包的版本，与 `package.json` 中的版本相匹配。

lockfileVersion

一个整数版本，从 1 开始，表示生成此 `package-lock.json` 文件所使用的文档版本的语义。

请注意，文件格式在 `npm v7` 中发生了重大变化，以跟踪否则需要查看 `node_modules` 或 `npm` 注册表的信息。`npm v7` 生成的锁定文件将包含 `lockfileVersion: 2`。

- 没有提供版本：早于 `npm v5` 的版本的“古老”缩减包文件。
- 1: `npm v5` 和 `v6` 使用的锁定文件版本。
- 2: `npm v7` 和 `v8` 使用的锁定文件版本。向后兼容到 `v1` 锁定文件。
- 3: `npm v9` 使用的锁定文件版本。向后兼容到 `npm v7`。

`npm` 将始终尝试从锁定文件中获取任何可用的数据，即使它不是设计用于支持的版本。

packages

这是一个将软件包位置映射到包含该软件包信息的对象。

通常，根项目使用 `""` 作为键，所有其他软件包使用相对于根项目文件夹的路径作为键。

软件包描述符具有以下字段：

- **version**: 在 `package.json` 中找到的版本。
- **resolved**: 实际解析软件包的位置。对于从注册表获取的软件包，这将是一个指向 `tarball` 的 URL。对于 `git` 依赖项，这将是包含提交 SHA 的完整 `git` URL。对于链接依赖项，这将是链接目标的位置。`registry.npmjs.org` 是一个特殊的值，表示“当前配置的注册表”。
- **integrity**: 用于在此位置解压缩的软件包的 `sha512` 或 `sha1` [标准子资源完整性](#) 字符串。
- **link**: 一个标志，表示这是一个符号链接。如果存在此标志，则不指定其他字段，因为链接目标也将包含在锁定文件中。

- **dev**、**optional**、**devOptional**: 如果该软件包严格属于 **devDependencies** 树，则 **dev** 为 **true**。如果它严格属于 **optionalDependencies** 树，则 **optional** 为 **true**。如果它既是 **dev** 依赖项又是非 **dev** 依赖项的可选依赖项的传递依赖项，则 **devOptional** 为 **true**。（对于 **dev** 依赖项的 **optional** 依赖项，**dev** 和 **optional** 都为 **true**。）
- **inBundle**: 一个标志，表示该软件包是捆绑依赖项。
- **hasInstallScript**: 一个标志，表示该软件包具有 **preinstall**、**install** 或 **postinstall** 脚本。
- **hasShrinkwrap**: 一个标志，表示该软件包具有 **npm-shrinkwrap.json** 文件。
- **bin**、**license**、**engines**、**dependencies**、**optionalDependencies**: 来自 **package.json** 的字段。

dependencies

用于支持 **npm** 使用 **lockfileVersion: 1** 的旧版本的数据。这是一个将软件包名称映射到依赖对象的映射。由于对象结构严格是分层的，因此在某些情况下，表示符号链接依赖关系可能有一定的挑战性。

如果存在 **packages** 部分，**npm v7** 将完全忽略此部分，但会保持其最新状态，以支持在 **npm v6** 和 **npm v7** 之间切换。

依赖对象具有以下字段：

- **version**: 一个依赖包的规范，根据软件包的性质而变化，并且可用于获取它的新副本。
 - 捆绑依赖项：无论来源如何，都是纯粹用于信息目的的版本号。
 - 注册表源：这是一个版本号（例如 **1.2.3**）。
 - **git** 源：这是带有已解决提交的 **git** 规范（例如 **git+https://example.com/foo/bar#115311855adb0789a0466714ed48a1499ffea97e**）。
 - **HTTP tarball** 源：这是 **tarball** 的 URL（例如 **https://example.com/example-1.3.0.tgz**）。
 - 本地 **tarball** 源：这是 **tarball** 的文件 URL（例如 **file:///opt/storage/example-1.3.0.tgz**）。
 - 本地链接源：这是链接的文件 URL（例如 **file:libs/our-module**）。
- **integrity**: 解压缩在此位置的软件包的 **sha512** 或 **sha1** 标准子资源完整性 字符串。对于 **git** 依赖项，这是提交 **SHA**。

- **resolved**: 对于注册表源，这是 **tarball** 相对于注册表 URL 的路径。如果 **tarball** URL 不在与注册表 URL 相同的服务器上，则这是一个完整的 URL。**registry.npmjs.org** 是一个特殊的值，表示“当前配置的注册表”。
- **bundled**: 如果为 **true**，则表示这是捆绑依赖项，并且将由父模块安装。在安装过程中，将在提取阶段从父模块中提取此模块，而不是作为单独的依赖项安装。
- **dev**: 如果为 **true**，则该依赖项仅是顶级模块的开发依赖项，或者是一个非开发依赖项的传递依赖项。对于既是 **dev** 依赖项又是非开发依赖项的传递依赖项，此值为 **false**。（**optional** 依赖项的 **dev** 依赖项将同时设置 **dev** 和 **optional**。）
- **optional**: 如果为 **true**，则该依赖项仅是顶级模块的可选依赖项，或者是一个非可选依赖项的传递依赖项。对于既是 **optional** 依赖项又是非可选依赖项的传递依赖项，此值为 **false**。
- **requires**: 这是一个模块名称到版本的映射。这是我们无论将在哪里安装，都要求与之匹配的依赖关系的列表。版本应与我们的 **dependencies** 或高于我们的级别中的依赖关系通过正常匹配规则匹配。
- **dependencies**: 该依赖项的依赖关系，与顶级相同。

参见

- [npm shrinkwrap](#)
- [npm-shrinkwrap.json](#)
- [package.json](#)
- [npm install](#)

<http://nodejs.cn/learn/the-package-lock-json-file>

任务开始

创建一个名为YTM-KOA的项目，并临时引入**axios**、**koa**、**koa-jwt**、**music-metadata**、**jpeg-js**依赖，然后使用**npm start**命令运行脚本：

```
node main.js
```

在ytm-koa文件夹中，创建一个新文件**main.js**，并编写以下内容：

```
const Koa = require('koa');

const app = new Koa();

app.use(async ctx => {
  ctx.body = 'Hello World';
});

app.listen(3000);
```

运行以下代码：

```
npm i

npm start
```

项目成功启动，访问localhost:3000即可看到Hello World。

```
● jingxuanwei@df0f0 Week4 % mkdir YTM-KOA
  cd YTM-KOA

● jingxuanwei@df0f0 YTM-KOA % npm init -y

Wrote to /Users/jingxuanwei/Desktop/Google/Week4/YTM-KOA/package.json:

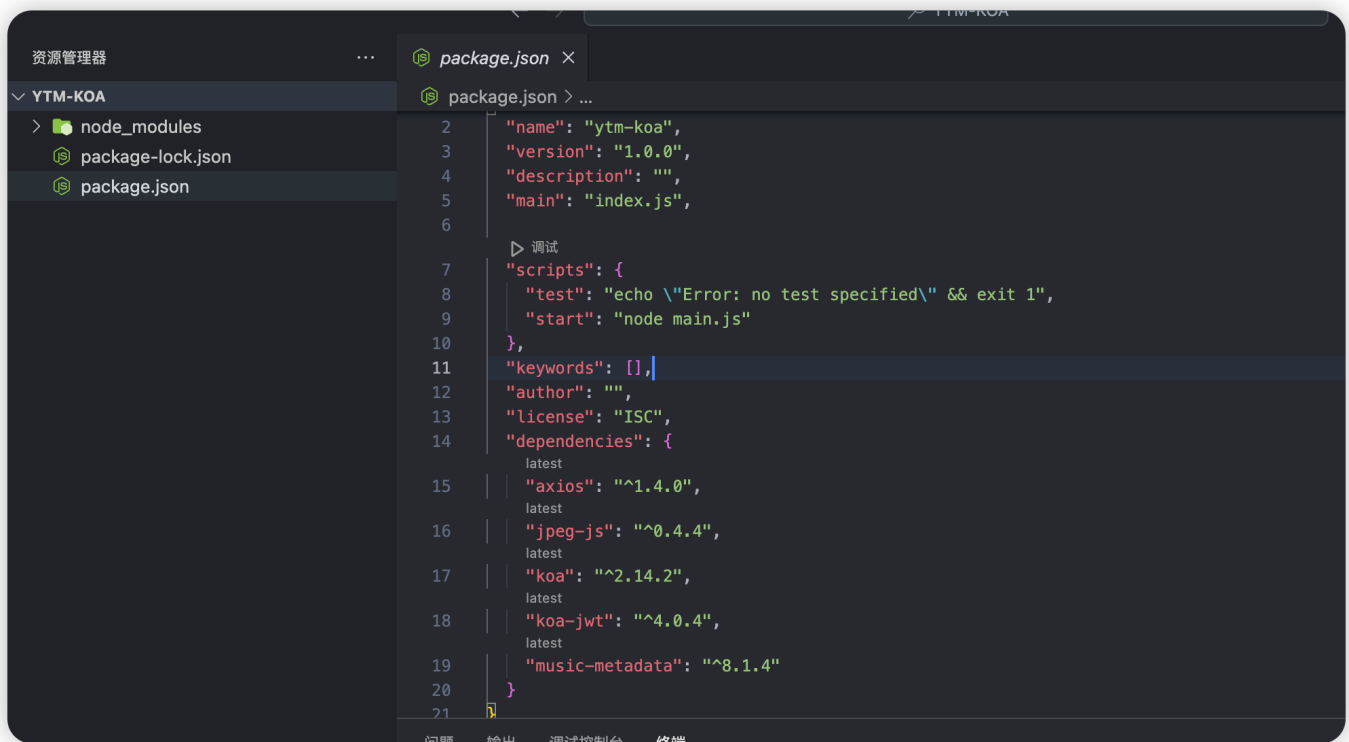
{
  "name": "ytm-koa",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

● jingxuanwei@df0f0 YTM-KOA % npm install axios koa koa-jwt music-metadata jpeg-js

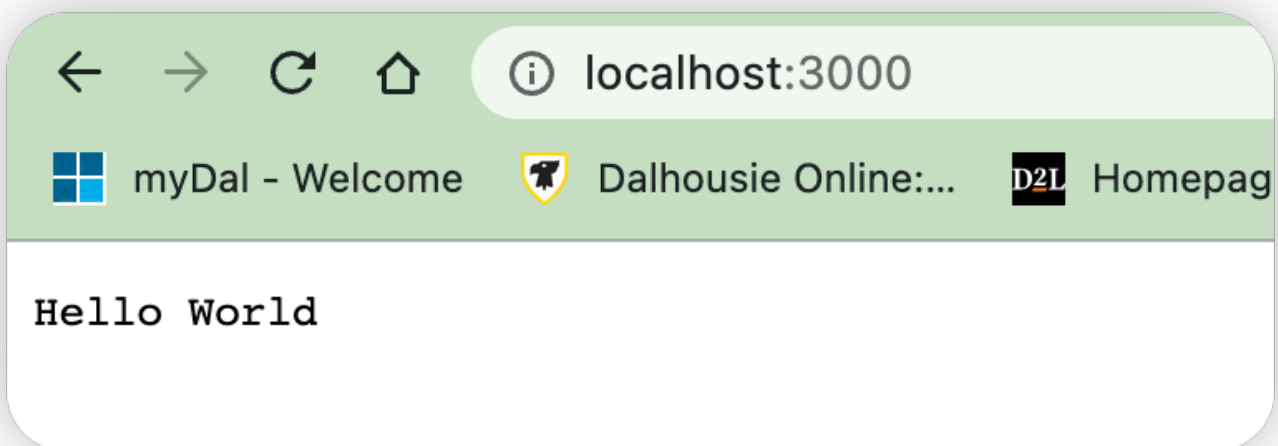
added 80 packages, and audited 81 packages in 18s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
○ jingxuanwei@df0f0 YTM-KOA %
```



```
2  "name": "ytm-koa",
3  "version": "1.0.0",
4  "description": "",
5  "main": "index.js",
6
7  "scripts": {
8    "test": "echo \\\"Error: no test specified\\\" && exit 1",
9    "start": "node main.js"
10 },
11 "keywords": [],
12 "author": "",
13 "license": "ISC",
14 "dependencies": {
15   "axios": "^1.4.0",
16   "jpeg-js": "^0.4.4",
17   "koa": "^2.14.2",
18   "koa-jwt": "^4.0.4",
19   "music-metadata": "^8.1.4"
20 }
21
```



本地模拟环境搭建

由于实习环境无法提供数据库，我们以本地方式模拟mongoDB数据库，mongo的基本格式实际上是Json。

要求实现一个libraryInit(path)方法，该函数在APP启动时读取/Library下的所有音频文件（用户需要提前放置一些MP3文件，大约100个），通过music-metadata库获取文件的标签信息，并生成index.json文件存储在/Library中，格式如下：


```
{
  "track_id": "",
  "title": "",
  "artist": ["", ""],
  "album": "",
  "album_id": "",
  "genre": "",
  "copyright": "",
  "length" : "",
  "track_number": 0,
  "quality": "STD",
  "file": "file path"
}
```

其中，`track_id`的生成采用连接`artist`、`title`和`album`三个字符串并使用MD5加密取前16个字符的方法。`track_id`是音乐库中曲目的唯一索引值，因此不同的歌曲不能重复。（即在数据库中，`track_id`并不是严格唯一的，后期同一首歌会有多个不同音质的文件对应）

关于信息对应的标签键的更多信息，请参考<https://www.npmjs.com/package/music-metadata>。

另外，如果有专辑封面图像，则将标签中的`imageBuffer`存储在`/Library/cover/<album_id>.jpg`中。注意：有时`imageBuffer`是PNG格式，因此需要使用`jpeg-js`库进行编码和保存，质量设置为100。

该JSON文件是顺序存储的，采用单行结构，即每行为上述对象的一部分。

每个文件的数据库创建应该是异步并行的，使用各种方法在有限数量内控制并行线程。具体的软件行为是，在通过`walk`等方法构建MP3文件列表之后，在`n`个线程的线程池中对列表中的每个文件执行类似于`indexCreate(filePath)`的异步方法。该函数负责读取文件的`id3Tag`，获取上述信息，编写JSON变量，将`imageBuffer`转储到文件中，然后阻止写入`/Library/index.json`（即如果`index.json`正在被其他线程使用，该函数应该等待，直到它可以访问该文件）。在创建每个文件后，在控制台中记录以下结构：

```
Index Created: <track_id> <file>
```

线程池中的线程数`n`应该通过使用`os`模块获取有关CPU的信息来确定。同时编写一个`libraryLoad(filePath)`方法，直接读取`index.json`并将其加载到内存中，返回对象。编写一个`libraryUpdate(lib, filePath)`方法来更新`index.json`文件，其中包含：

1. 比较并添加未索引的MP3文件
2. 删除在本地搜索中找不到对应文件的条目

注：这个库是针对用户的索引，以`track_id`为唯一标识。然而，在文件级别上，文件路径是唯一的索引值。上述函数名称是自定义的。

`libraryLoad(filePath)`和`libraryUpdate(path)`应该占用文件直到运行过程结束，然后释放控制。所以现在逻辑非常清晰：在'`npm start`'启动项目后，首先尝试'`libraryLoad(filePath)`'，如果文件不存在，则运行'`libraryInit(path)`'。如果文件存在，则挂载`.then((lib) => { libraryUpdate(lib, filePath) })`来完成索引库的加载和刷新。

```
npm install music-metadata md5 fs path os jpeg-js async glob crypto
```

Library.js

```
const fs = require('fs');
const path = require('path');
const glob = require('glob');
const md5 = require('md5');
const os = require('os');
const async = require('async');
const { encode } = require('jpeg-js');

const libraryPath = path.join(__dirname, 'Library');
const coverPath = path.join(libraryPath, 'cover');
const indexPath = path.join(libraryPath, 'index.json');
const cpuCount = os.cpus().length;

// 初始化音乐库
async function libraryInit() {
  // 获取音乐文件列表
  const files = glob.sync('**/*.mp3', { cwd: libraryPath });
  const index = []
  // 初始化音乐照片文件夹
  const albumCovers = {};

  // 创建音乐照片文件夹
  await fs.promises.mkdir(coverPath, { recursive: true });

  // 处理单个文件
  const processFile = async (file, callback) => {
    if (typeof callback !== 'function') {
      // 默认回调函数为空函数
      callback = () => {};
    }
  }
```

```

// 获取文件路径
const filePath = path.join(libraryPath, file);
try {
  // 解析音乐文件的元数据, 调用 `music-metadata` 模块
  const parseFile = await import('music-metadata').then((module) =>
module.parseFile);
  const metadata = await parseFile(filePath);

  let {
    artist,
    title,
    album,
    genre,
    track,
    picture
  } = metadata.common;
  // 如果 `artist` 不是一个数组, 则将其转换为一个包含一个字符串的数组
  if (!Array.isArray(artist)) {
    artist = [artist];
  }

  // 返回报错
  if (!artist || !title || !album) {
    console.error(`Invalid metadata for file: ${file}`);
    callback();
    return;
  }

  // 获取音乐文件的 ID, 等信息
  const trackId = md5(artist.join('') + title + album).substring(0,
16);

  const trackNumber = track.no || 0;
  const quality = 'STD';
  // json file的数据
  const fileData = {
    track_id: trackId,
    title,
    artist,
    album,
    album_id: md5(album),
    genre: genre ? genre[0] : '',
    length: metadata.format.duration,
    track_number: trackNumber,

```

```

        quality,
        file: file,
    };

    // 保存专辑封面图像
    if (picture && picture.length > 0) {
        const albumCoverPath = path.join(coverPath,
`${fileData.album_id}.jpg`);
        const imageData = picture[0].data;
        const imageBuffer = Buffer.from(imageData);
        await fs.promises.writeFile(albumCoverPath, encode({ data:
imageBuffer, width: 0, height: 0 }, 100).data);
    }

    // 将文件数据添加到索引数组, 按照格式要求换行
    index.push(JSON.stringify(fileData, null, 2));
    console.log(`Index Created: ${trackId} ${file}`);
}
catch (error) {
    console.error(`Error processing file: ${file}`);
    console.error(error);
}

callback();
};

// 并发处理文件
await new Promise((resolve) => {
    async.eachLimit(files, cpuCount, processFile, () => {
        resolve();
    });
});

// 将索引数据写入文件, 按照格式要求识别整体json
await fs.promises.writeFile(indexPath, JSON.stringify(index));
}

// 加载音乐库索引
async function libraryLoad() {
    try {
        // 读取索引文件
        const data = await fs.promises.readFile(indexPath, 'utf-8');
        // 使用换行符分割索引文件
        const lines = data.split('\n');
    }
}

```

```

    // 转换成一个数组
    return lines.map(line => JSON.parse(line));
  }
  catch (error) {
    console.error('Error loading library index');
    console.error(error);
    return [];
  }
}

// 更新音乐库
async function libraryUpdate(lib) {
  const existingFiles = lib.map((item) => item.file);
  const files = glob.sync('*/*.mp3', { cwd: libraryPath });
  // 解析文件
  const removeItems = lib.filter((item) =>
!existingFiles.includes(item.file));
  const newFiles = files.filter((file) =>
!existingFiles.includes(file));

  if (removeItems.length > 0) {
    console.log('Removing items:');
    console.log(removeItems);
    const newLib = lib.filter((item) => !removeItems.includes(item));
    await fs.promises.writeFile(indexPath, newLib.map((item) =>
JSON.stringify(item)).join('\n'));
  }

  if (newFiles.length > 0) {
    console.log('Adding new files:');
    console.log(newFiles);
    await libraryInit();
  }
}

// 启动应用程序
async function start() {
  try {
    const indexExists = await fs.promises.access(indexPath)
      .then(() => true)
      .catch(() => false);
    if (indexExists) {
      // 加载已有的音乐库索引

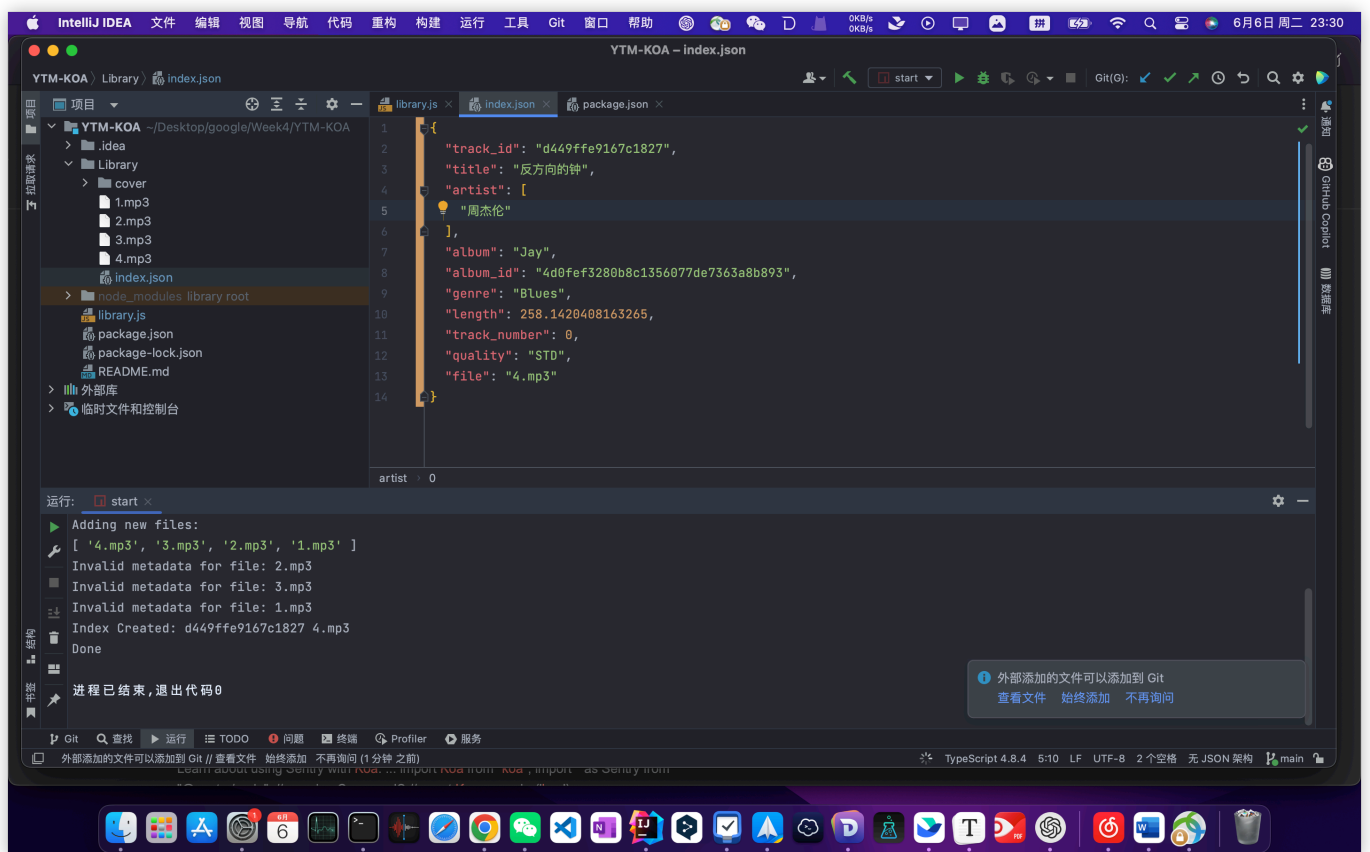
```

```

    const lib = await libraryLoad();
    // 更新音乐库
    await libraryUpdate(lib);
  }
  else {
    // 初始化音乐库
    await libraryInit();
  }
} catch (error) {
  console.error('Error starting the app');
  console.error(error);
}

start().then(() => console.log('Done'));

```



"云端"上的数据

###

在这个项目中，我们使用非关系型数据库mongoDB构建整个系统的数据库系统。

mongo的基本概念和介绍:

<https://www.runoob.com/mongodb/mongodb-intro.html>

什么是MongoDB?

MongoDB 是由C++语言编写的，是一个基于分布式文件存储的开源数据库系统。

在高负载的情况下，添加更多的节点，可以保证服务器性能。

MongoDB 旨在为WEB应用提供可扩展的高性能数据存储解决方案。

MongoDB 将数据存储为一个文档，数据结构由键值(key=>value)对组成。MongoDB 文档类似于 JSON 对象。字段值可以包含其他文档，数组及文档数组。

```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```



The diagram illustrates the structure of a MongoDB document. It shows a JSON-like object with four fields: 'name', 'age', 'status', and 'groups'. Each field is followed by a colon and its value. To the right of each field-value pair, there is a horizontal arrow pointing left towards the field name, and the text 'field: value' is written to the right of the arrow. This visualizes the key-value pair structure of the document.

主要特点

- MongoDB 是一个面向文档存储的数据库，操作起来比较简单和容易。
- 你可以在MongoDB记录中设置任何属性的索引(如: FirstName="Sameer",Address="8 Gandhi Road")来实现更快的排序。
- 你可以通过本地或者网络创建数据镜像，这使得MongoDB有更强的扩展性。
- 如果负载的增加（需要更多的存储空间和更强的处理能力），它可以分布在计算机网络中的其他节点上这就是所谓的分片。
- Mongo支持丰富的查询表达式。查询指令使用JSON形式的标记，可轻易查询文档中内嵌的对象及数组。
- MongoDB 使用update()命令可以实现替换完成的文档（数据）或者一些指定的数据字段。
- Mongodb中的Map/reduce主要是用来对数据进行批量处理和聚合操作。
- Map和Reduce。Map函数调用emit(key,value)遍历集合中所有的记录，将key与value传给Reduce函数进行处理。

- Map函数和Reduce函数是使用Javascript编写的，并可以通过db.runCommand或mapreduce命令来执行MapReduce操作。
 - GridFS是MongoDB中的一个内置功能，可以用于存放大量小文件。
 - MongoDB允许在服务端执行脚本，可以用Javascript编写某个函数，直接在服务端执行，也可以把函数的定义存储在服务端，下次直接调用即可。
 - MongoDB支持各种编程语言:RUBY, PYTHON, JAVA, C++, PHP, C#等多种语言。
 - MongoDB安装简单。
-

历史

- 2007年10月，MongoDB由10gen团队所发展。2009年2月首度推出。
 - 2012年05月23日，MongoDB 2.1 开发分支发布了! 该版本采用全新架构，包含诸多增强。
 - 2012年06月06日，MongoDB 2.0.6 发布，分布式文档数据库。
 - 2013年04月23日，MongoDB 2.4.3 发布，此版本包括了一些性能优化，功能增强以及bug修复。
 - 2013年08月20日，MongoDB 2.4.6 发布。
 - 2013年11月01日，MongoDB 2.4.8 发布。
 -
-

MongoDB 下载

你可以在mongodb官网下载该安装包，地址为：<https://www.mongodb.com/download-center#community>。MongoDB支持以下平台：

- OS X 32-bit
 - OS X 64-bit
 - Linux 32-bit
 - Linux 64-bit
 - Windows 32-bit
 - Windows 64-bit
 - Solaris i86pc
 - Solaris 64
-

语言支持

MongoDB有官方的驱动如下：

- [C](#)
 - [C++](#)
 - [C# / .NET](#)
 - [Erlang](#)
 - [Haskell](#)
 - [Java](#)
 - [JavaScript](#)
 - [Lisp](#)
 - [node.JS](#)
 - [Perl](#)
 - [PHP](#)
 - [Python](#)
 - [Ruby](#)
 - [Scala](#)
 - [Go](#)
-

MongoDB 工具

有几种可用于MongoDB的管理工具。

监控

MongoDB提供了网络和系统监控工具Munin，它作为一个插件应用于MongoDB中。

Gangila是MongoDB高性能的系统监视的工具，它作为一个插件应用于MongoDB中。

基于图形界面的开源工具 Cacti, 用于查看CPU负载, 网络带宽利用率, 它也提供了一个应用于监控 MongoDB 的插件。

GUI

- Fang of Mongo – 网页式, 由Django和jQuery所构成。
- Futon4Mongo – 一个CouchDB Futon web的mongodb山寨版。
- Mongo3 – Ruby写成。

- MongoHub – 适用于OSX的应用程序。
 - Opricot – 一个基于浏览器的MongoDB控制台, 由PHP撰写而成。
 - Database Master — Windows的mongodb管理工具
 - RockMongo — 最好的PHP语言的MongoDB管理工具, 轻量级, 支持多国语言.
-

MongoDB 应用案例

下面列举一些公司MongoDB的实际应用:

- Craigslist上使用MongoDB的存档数十亿条记录。
- FourSquare, 基于位置的社交网站, 在Amazon EC2的服务器上使用MongoDB分享数据。
- Shutterfly, 以互联网为基础的社会和个人出版服务, 使用MongoDB的各种持久性数据存储的要求。
- bit.ly, 一个基于Web的网址缩短服务, 使用MongoDB的存储自己的数据。
- spike.com, 一个MTV网络的联营公司, spike.com使用MongoDB的。
- Intuit公司, 一个为小企业和个人的软件和服务提供商, 为小型企业使用MongoDB的跟踪用户的数据。
- sourceforge.net, 资源网站查找, 创建和发布开源软件免费, 使用MongoDB的后端存储。
- etsy.com, 一个购买和出售手工制作物品网站, 使用MongoDB。
- 纽约时报, 领先的在线新闻门户网站之一, 使用MongoDB。
- CERN, 著名的粒子物理研究所, 欧洲核子研究中心大型强子对撞机的数据使用MongoDB。

安装mongodb

Mac OSX 平台安装 MongoDB

包安装

MongoDB 提供了 OSX 平台上 64 位的安装包, 你可以在官网下载安装包。

下载地址: <https://www.mongodb.com/try/download/community>

Version

4.0.9 (current release) ▼

OS

macOS 64-bit x64 ▼

Package

TGZ ▼

Download

https://fastdl.mongodb.org/osx/mongodb-osx-ssl-x86_64-4.0.9.tgz

从 MongoDB 3.0 版本开始只支持 OS X 10.7 (Lion) 版本及更新版本的系统。

接下来我们使用 `curl` 命令来下载安装：

```
# 进入 /usr/local
cd /usr/local

# 下载
sudo curl -O https://fastdl.mongodb.org/osx/mongodb-osx-ssl-x86_64-4.0.9.tgz

# 解压
sudo tar -zxvf mongodb-osx-ssl-x86_64-4.0.9.tgz

# 重命名为 mongodb 目录

sudo mv mongodb-osx-x86_64-4.0.9/ mongodb
```

安装完成后，我们可以把 MongoDB 的二进制命令文件目录（安装目录/bin）添加到 PATH 路径中：

```
export PATH=/usr/local/mongodb/bin:$PATH
```

创建日志及数据存放的目录：

- 数据存放路径：

```
sudo mkdir -p /usr/local/var/mongodb
```

- 日志文件路径:

```
sudo mkdir -p /usr/local/var/log/mongodb
```

接下来要确保当前用户对以上两个目录有读写的权限:

```
sudo chown jingxuanwei /usr/local/var/mongodb  
sudo chown jingxuanwei /usr/local/var/log/mongodb
```

以上 **runoob** 是我电脑上的用户，你这边需要根据你当前对用户名来修改。

接下来我们使用以下命令在后台启动 **mongodb**:

```
mongod --dbpath /usr/local/var/mongodb --logpath  
/usr/local/var/log/mongodb/mongo.log --fork
```

- **--dbpath** 设置数据存放目录
- **--logpath** 设置日志存放目录
- **--fork** 在后台运行

如果不想在后端运行，而是在控制台上查看运行过程可以直接设置配置文件启动:

```
mongod --config /usr/local/etc/mongod.conf
```

查看 **mongod** 服务是否启动:

```
ps aux | grep -v grep | grep mongod
```

使用以上命令如果看到有 **mongod** 的记录表示运行成功。

启动后我们可以使用 **mongo** 命令打开一个终端:

```
$ cd /usr/local/mongodb/bin
$ ./mongo
MongoDB shell version v4.0.9
connecting to: mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("3c12bf4f-695c-48b2-b160-8420110ccdcf") }
MongoDB server version: 4.0.9

.....
> 1 + 1
2
>
```

使用 brew 安装

此外你还可以使用 OSX 的 brew 来安装 mongodb:

```
brew tap mongodb/brew
brew install mongodb-community@4.4
```

@ 符号后面的 4.4 是最新版本号。

安装信息:

- 配置文件: **/usr/local/etc/mongod.conf**
- 日志文件路径: **/usr/local/var/log/mongodb**
- 数据存放路径: **/usr/local/var/mongodb**

运行 MongoDB

我们可以使用 brew 命令或 mongod 命令来启动服务。

brew 启动:

```
brew services start mongodb-community@4.4
```

brew 停止:

```
brew services stop mongodb-community@4.4
```

mongod 命令后台进程方式:

```
mongod --config /usr/local/etc/mongod.conf --fork
```

这种方式启动要关闭可以进入 mongo shell 控制台来实现：

```
> db.adminCommand({ "shutdown" : 1 })
```

任务介绍

在Node中，我们使用mongoose库处理Node与MongoDB的交互。实际上，Node本身集成了对MongoDB.js的原生支持，但是mongoose中间件更容易使用。Mongoose中文手册：

<http://mongoosejs.net/docs/index.html>

在本地安装MongoDB库之后，我们需要4个数据库：users、library、playlists、history，用于后续项目。

其中，library的数据结构之前已经指出，总集合命名为index，用户的私人库集合命名为u__，结构如下：

```
{
  "type": "track / album / playlist",
  "id": "id",
  "added_date":
}
```

users的数据结构如下，不需要管理集合：

```
{
  "uid": "",
  "name": "",
  "secret": "",
  "subscribe": "Premium",
  "subscribe_expired": ,
  "last_login": ,
  "playing": "machine_id"
}
```

playlists分为总索引集合和单个列表集合，结构如下：

总索引集合命名为index：

```
{
  "pid": "id",
  "author": "uid",
}
```

```
"name": "",
"description": "",
"added": 0,
"liked": 0,
"shared": 0,
"played": 0,
"public": true,
"image": "path",
"type": "playlist / album",
"last_update":
}
```

单个列表集合为:

```
{
  "tid": "track_id",
  "order": 0 // order
}
```

同时，将之前在前一节中构建的library中与文件处理相关的所有函数替换为使用Mongoose与Mongoose数据库操作的方式，并将index.json迁移到MongoDB，但保留cover中的内容，仍以本地文件路径索引的形式存储。

在将数据写入MongoDB等数据库时，可以忽略阻塞问题，中间件和数据库引擎将自行处理并发问题。

```
const fs = require('fs');
const path = require('path');
const glob = require('glob');
const md5 = require('md5');
const os = require('os');
const mongoose = require('mongoose');
const libraryPath = path.join(__dirname, 'Library');
const coverPath = path.join(libraryPath, 'cover');

// 链接数据库
const connectionString = 'mongodb://localhost:27017/YTM';
mongoose.connect(connectionString, { useNewUrlParser: true,
useUnifiedTopology: true });

// 创建音乐库集合模型
const librarySchema = new mongoose.Schema({
  track_id: String,
```



```
    title: String,
    artist: [String],
    album: String,
    album_id: String,
    genre: String,
    length: Number,
    track_number: Number,
    quality: String,
    file: String,
    added_date: { type: Date, default: Date.now }
  });

const Library = mongoose.model('library', librarySchema);

// 创建用户集合模型
const userSchema = new mongoose.Schema({
  uid: String,
  name: String,
  secret: String,
  subscribe: String,
  subscribe_expired: Date,
  last_login: Date,
  playing: String,
});

const User = mongoose.model('users', userSchema);

// 创建播放列表模型
const playlistSchema = new mongoose.Schema({
  pid: String,
  author: String,
  name: String,
  description: String,
  added: Number,
  liked: Number,
  shared: Number,
  played: Number,
  public: Boolean,
  image: String,
  type: String,
  last_update: Date,
});

const Playlist = mongoose.model('Playlist', playlistSchema);
```

```
const playlistItemSchema = new mongoose.Schema({
  tid: String,
  order: Number,
});

const PlaylistItem = mongoose.model('<pid>', playlistItemSchema);

// 处理单个文件
const processFile = async (file) => {
  const filePath = path.join(libraryPath, file);
  try {
    const parseFile = await import('music-metadata').then((module)
=> module.parseFile);
    const metadata = await parseFile(filePath);
    let { artist, title, album, genre, track, picture } =
metadata.common;

    if (!Array.isArray(artist)) {
      artist = [artist];
    }

    if (!artist || !title || !album) {
      console.error(`Invalid metadata for file: ${file}`);
      return;
    }

    const trackId = md5(artist.join('') + title +
album).substring(0, 16);
    const trackNumber = track.no || 0;
    const quality = 'STD';
    const fileData = {
      track_id: trackId,
      title,
      artist,
      album,
      album_id: md5(album),
      genre: genre ? genre[0] : '',
      length: metadata.format.duration,
      track_number: trackNumber,
      quality,
      file: file,
    };
  }
};
```

```

    await Library.create(fileData);

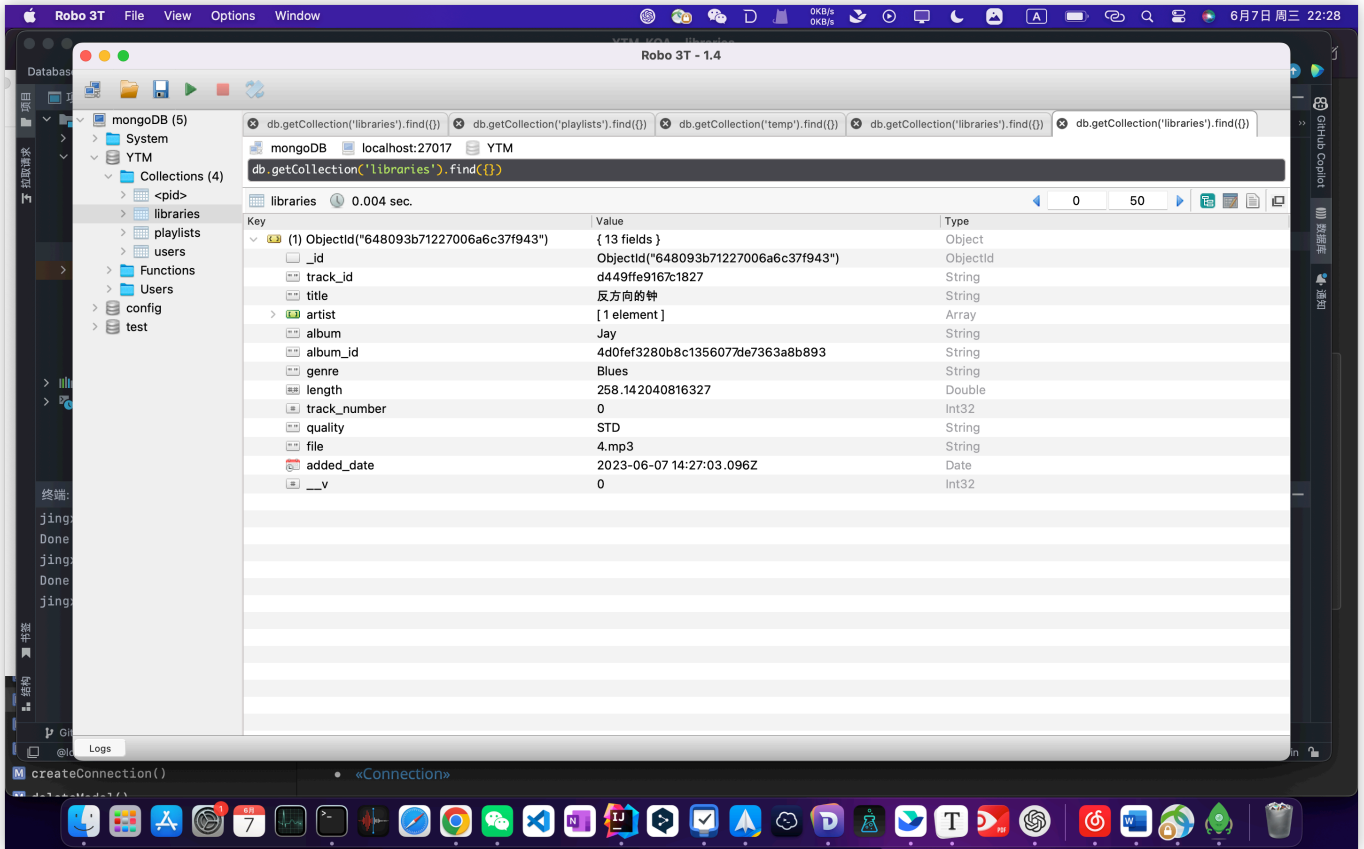
    if (picture && picture.length > 0) {
        const albumCoverPath = path.join(coverPath,
`${fileData.album_id}.jpg`);
        const imageData = picture[0].data;
        const imageBuffer = Buffer.from(imageData);
        await fs.promises.writeFile(albumCoverPath, imageBuffer);
    }
} catch (error) {
    console.error(`Error processing file: ${file}`);
    console.error(error);
}
};

// 初始化音乐库
async function libraryInit() {
    const files = glob.sync('**/*.mp3', { cwd: libraryPath });
    await Promise.all(files.map(file => processFile(file)));
}

// 启动应用程序
async function start() {
    try {
        await libraryInit();
        // 在此处启动你的应用程序
    } catch (error) {
        console.error('Error starting the app');
        console.error(error);
    }
    // 关闭与 MongoDB 的连接
    await mongoose.connection.close();
}

start().then(() => console.log('Done'));

```



添加了更多歌曲后

