

# Electron应用的打包和自动更新



**Alaso** 发布于 4 月 12 日

在上一篇文章中，我们介绍了electron的一些基础知识，[入门Electron](#)，[手把手教你编写完整实用案例](#)，在这里我们将基于这个项目继续介绍Electron的打包和自动更新。

## 生成图标

在打包应用之前，要为应用准备一个图标，作为安装包图标。不同的操作系统所需图标的格式不同，Mac对应的格式为`icns`，Windows对应的格式为`ico`。

图标的生成可以借助 `electron-icon-builder`。

- 首先，准备一张`1024*1024`的png图片，将图片放在项目文件夹中，我们这里选择放在`tasky/public`文件夹中。



- 安装 `electron-icon-builder`:

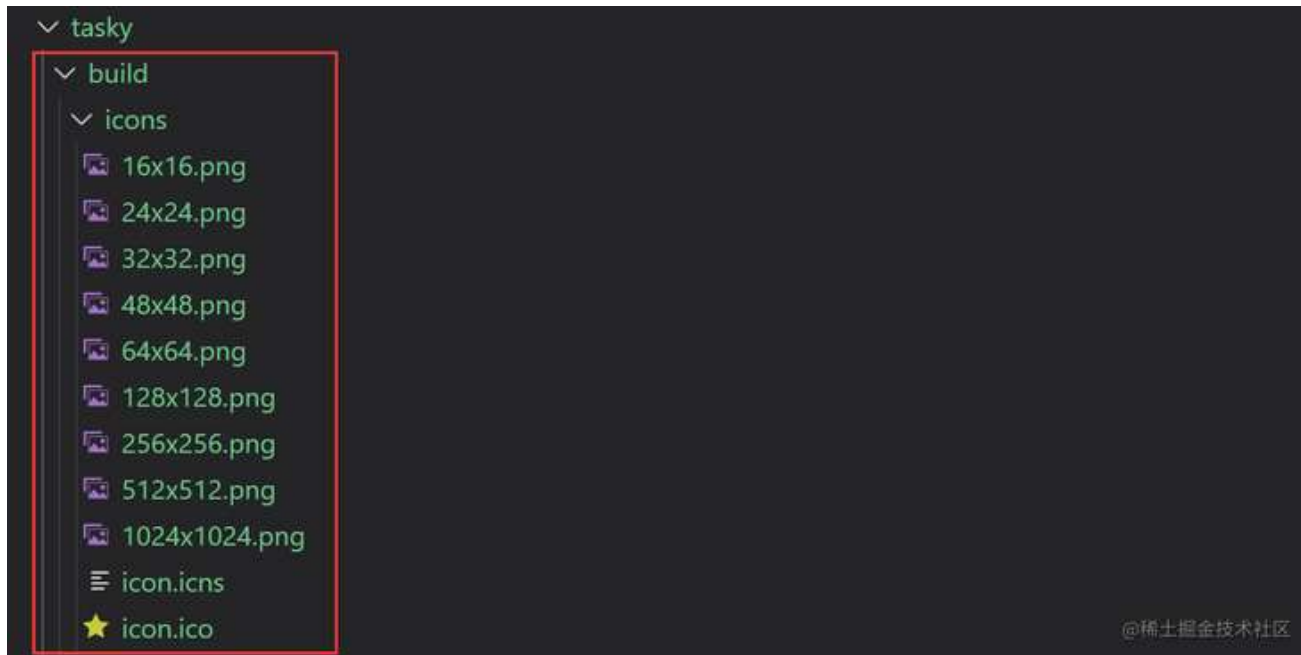
```
npm i electron-icon-builder --D
```

- 在`package.json`的`scripts`添加指令:

```
"build-icon": "electron-icon-builder --input=./public/icon.png --output=build --flatten"
```



- 运行`npm run build-icon`，就会在`build`文件夹中生成一系列打包所需的图标文件。



## 打包应用

Electron生态下常用的打包工具有两个：`electron-builder` 和 `electron-packager`。

`electron-builder`配置更灵活，使用也更广泛。下面，我们使用`electron-builder`来进行打包。

## 安装

```
npm i electron-builder --D
```

## 配置

使用`electron-builder`打包主要是各种配置，它支持两种配置方式：

1. 在`package.json`中添加`build`字段：

```
"build": {  
  "appId": "your.app.id"  
}
```

2. 指定配置文件，在其中写入配置项。默认是项目根目录下的`electron-builder.yml`。



在日常开发中，`package.json`这种配置方式比较常用，我们也以这种方式为主。

## 基础配置

```
"build": {
  "appId": "this.is.tasky",
  "productName": "Tasky",
  "copyright": "Copyright © 2021 Alaso",
  "directories": {
    "buildResources": "build",    //指定打包需要的静态资源，默认是build
    "output": "dist",           //打包生成的目录，默认是dist
  }
},
```

`build`文件夹放置的是，`electron-builder`默认的在打包过程中需要的静态文件，比如我们上面生成的图标文件；`dist`文件夹放置的是打包生成的各种文件。

3. 在`package.json`的`scripts`添加指令：`"pack": "electron-builder"`
4. 运行`npm run pack`

基于以上的配置，`electron-builder`会根据当前的操作系统打包出默认的文件。比如，在`windows`平台下，打包结果如下：



## 平台相关的配置

`electron-builder`会自动识别当前的操作系统，打出系统对应的安装包。这也意味着，如果要生成`exe\msi`，需要在`Windows`操作系统，如果是`dmg`，则需要在`Mac`操作系统。

`electron-builder`的配置选项中，有很多跟操作系统相关的配置，可以对不同平台的打包做一些定制效果。下面以`Windows`和`Mac`为例，介绍一些常用的平台相关的配置。

### 1. Windows

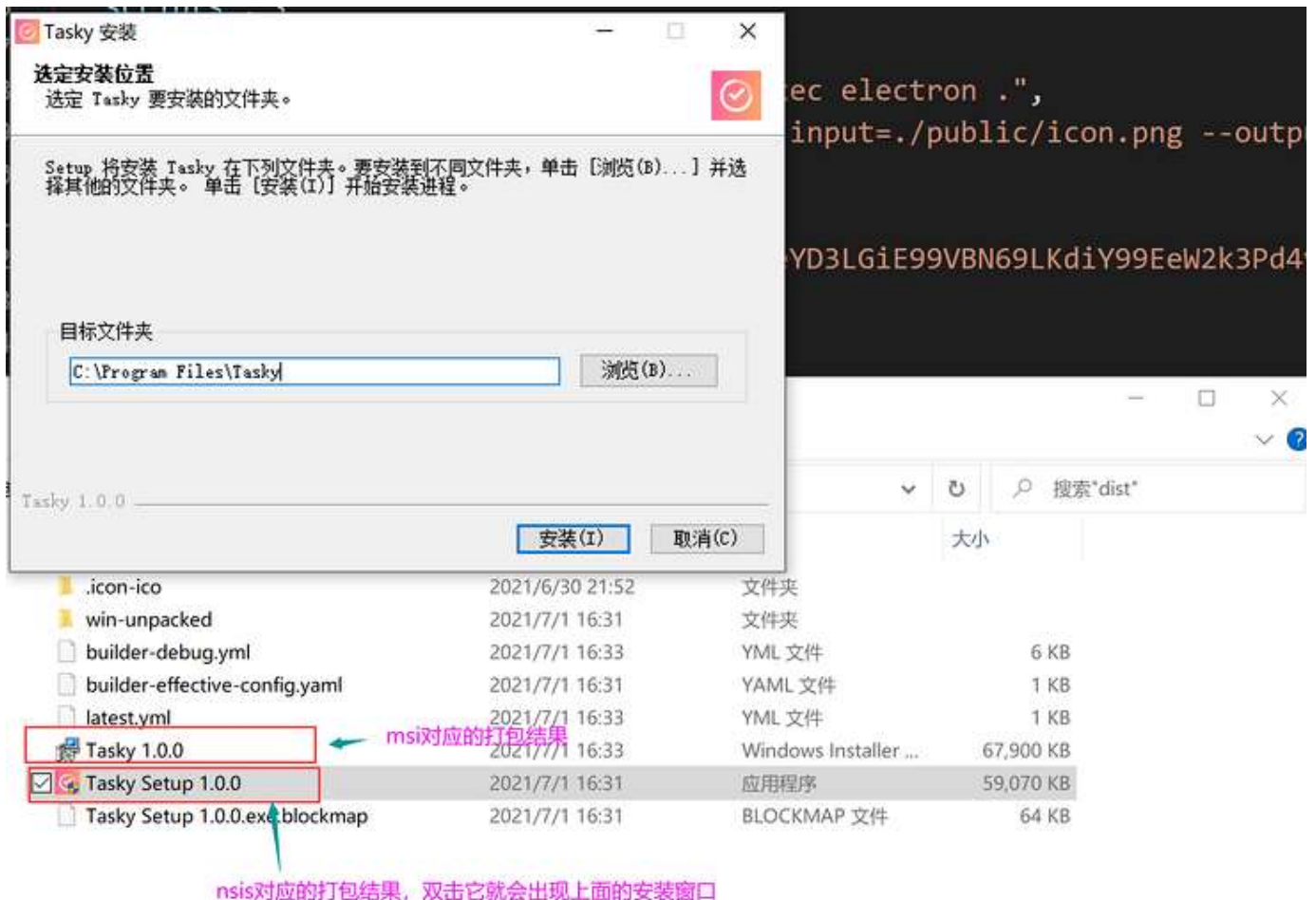


```

"build": {
  ...
  "win": {
    "target": ["msi", "nsis"],          //安装包的格式，默认是"nsis"
    "icon": "build/icons/icon.ico"    //安装包的图标
  },

  // "target"值"nsis"打包出来的就是exe文件
  // nsis是windows系统安装包的制作程序，它提供了安装、卸载、系统设置等功能
  // 关于"nsis"的一些配置
  "nsis": {
    "oneClick": false,                 //是否一键安装，默认为true
    "language": "2052",               //安装语言，2052对应中文
    "perMachine": true,               //为当前系统的所有用户安装该应用程序
    "allowToChangeInstallationDirectory": true //允许用户选择安装目录
  }
}

```



## 2. Mac

```








"target": ["dmg", "zip"],          //安装包的格式，默认是"dmg"和"zip"

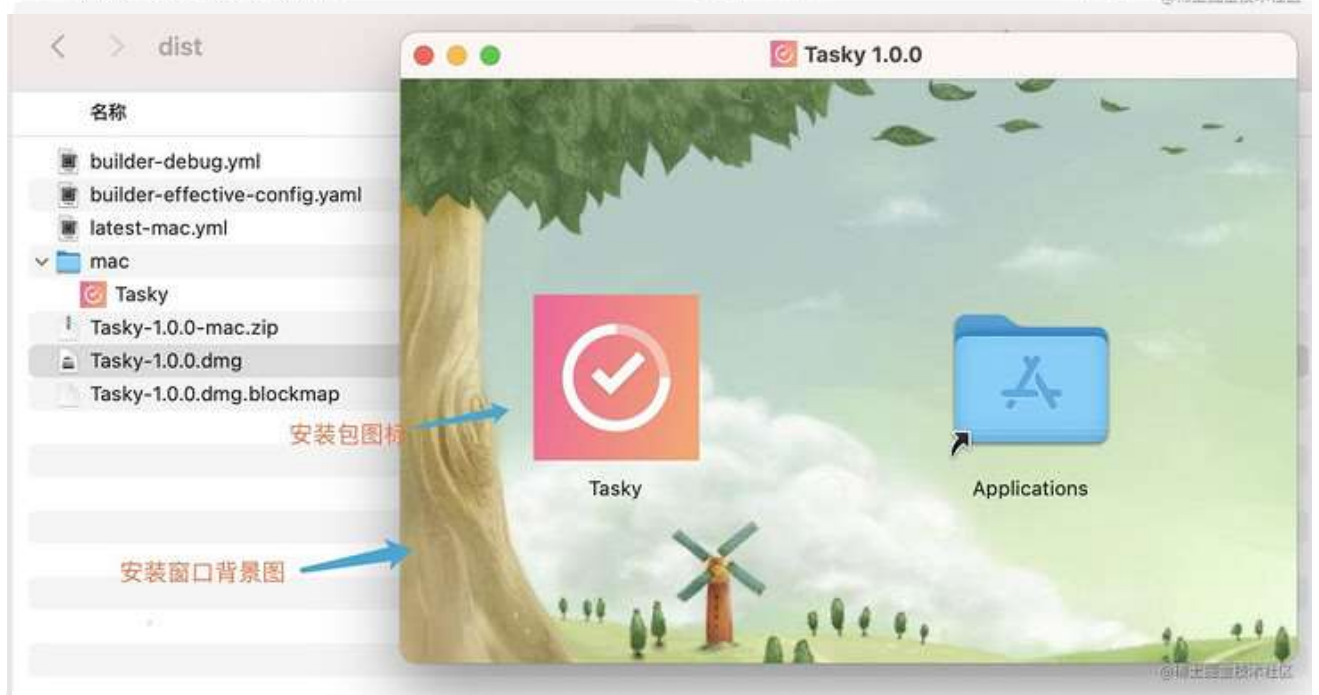
```

```

"background": "build/background.jfif",    //安装窗口背景图
"icon": "build/icons/icon.icns",         //安装图标
"iconSize": 100,                         //图标的尺寸
"contents": [                             //安装图标在安装窗口中的坐标信息
  {
    "x": 380,
    "y": 180,
    "type": "link",
    "path": "/Applications"
  },
  {
    "x": 130,
    "y": 180,
    "type": "file"
  }
],
"window": {                               //安装窗口的大小
  "width": 540,
  "height": 380
}
}
}

```

 builder-debug.yml	今天 下午 8:16	653 字节	YAML
 builder-effective-config.yml	今天 下午 8:15	697 字节	YAML
 latest-mac.yml	今天 下午 8:16	503 字节	YAML
>  mac	今天 下午 8:15	--	文件夹
 Tasky-1.0.0-mac.zip	今天 下午 8:16	77.6 MB	ZIP 归档
 Tasky-1.0.0.dmg	今天 下午 8:15	80.3 MB	磁盘映像
 Tasky-1.0.0.dmg.blockmap	今天 下午 8:15	86 KB	文稿



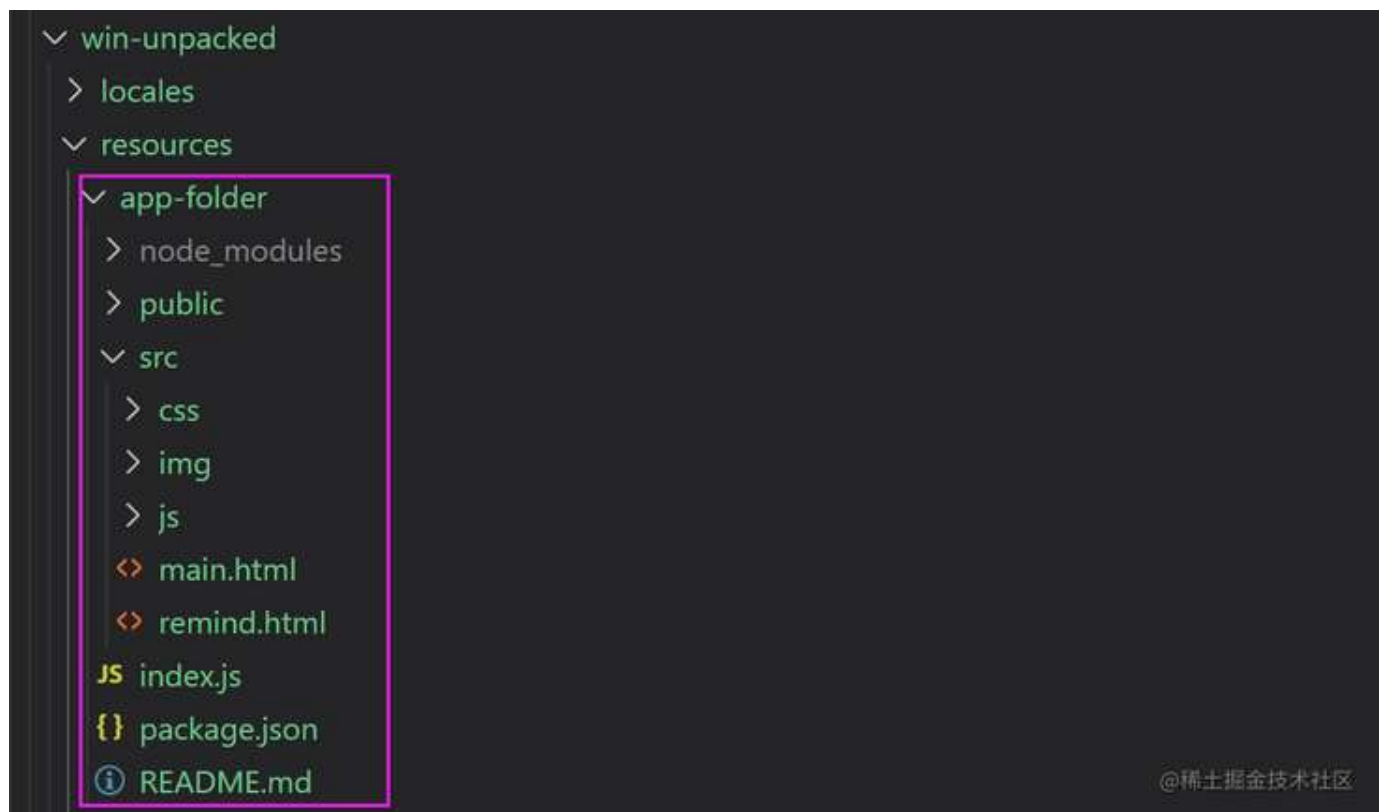
## 会将哪些文件pack到安装包

在打包生成的文件夹中，会有一个`app.asar`，它是Electron应用程序的主业务文件压缩包，要知道项目中哪些文件被pack到安装包，可以通过解压`app.asar`进行查看。

解压`app.asar`需要借助`asar`工具，首先来安装：`npm i asar -g`。

然后切换到`app.asar`所在目录，执行：`asar extract app.asar ./app-folder`。

以windows为例，`app.asar`位于`tasky\dist\win-unpacked\resources`目录中，解压后，可以看到`app-folder`中的内容如下：



可以看到，基本上就是项目所有文件了（除了`package-lock.json`、`.gitignore`、`build`文件夹），并且还有`node_modules`。

对于`node_modules`，并不是所有`node_modules`中的内容都会被打包进安装包，只有`package.json`中`dependencies`字段中的依赖会被打包，`devDependencies`字段中的依赖则不会。这是唯一规则，跟项目实际是否使用依赖没有关系。

所以，为了减小安装包体积，建议在渲染进程中使用的外部包，都安装在`devDependencies`中，然后使用`webpack`将外部包的代码和业务代码打包到一起，在后面的文章中会详细介绍。





当然，可以通过配置files字段，来指定将哪些内容进行打包。

## files

Array<String | FileSet> | String | FileSet

A [glob patterns](#) relative to the [app directory](#), which specifies which files to include when copying files to create the package.

Defaults to:

```
[
  "**/*",
  "!**/node_modules/*/ {CHANGELOG.md, README.md, README, readme.md, readme}",
  "!**/node_modules/*/ {test, __tests__, tests, powered-test, example, examples}",
  "!**/node_modules/*.d.ts",
  "!**/node_modules/.bin",
  "!**/*. {iml, o, hprof, orig, pyc, pyo, rbc, swp, csproj, sln, xproj}",
  "!*.editorconfig",
  "!**/._*",
  "!**/ { .DS_Store, .git, .hg, .svn, CVS, RCS, SCCS, .gitignore, .gitattributes }",
  "!**/ { __pycache__, thumbs.db, .flowconfig, .idea, .vs, .nyc_output }",
  "!**/ { appveyor.yml, .travis.yml, circle.yml }",
  "!**/ { npm-debug.log, yarn.lock, .yarn-integrity, .yarn-metadata.json }"
]
```

@稀土掘金技术社区

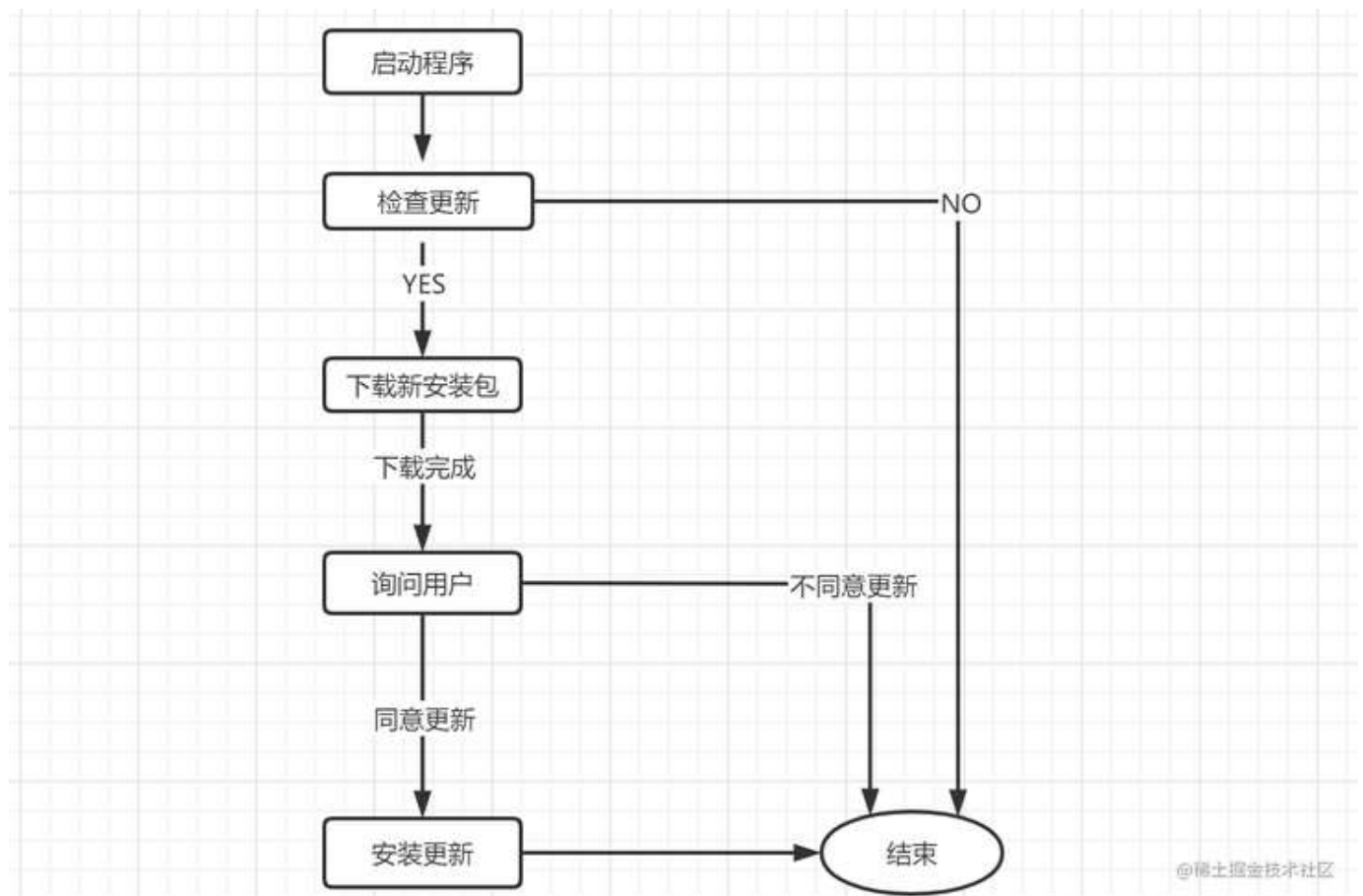
比如，我们只打包src文件夹、index.js和package.json，可以这样配置：

```
"build": {
  "files": [
    "package.json",
    "index.js",
    "src/**/*"
  ]
}
```

## 自动更新

要自动更新，应用程序的安装包应该存放在互联网的某台服务器上，每次打开应用的时候，进行自动检测，根据当前应用程序的version和线上版本进行匹配，当发现有新的version的时候，就自动下载，下载完成后，询问用户是否安装新版本。





## 打包不同版本

在`package.json`中, 有个`"version"`字段, 用于决定当前版本。

- step1: 设置`"version": "1.0.0"`, 运行`npm run pack`
- step2: 设置`"version": "1.0.1"`, 运行`npm run pack`

虽然, 我们没有改变应用程序的内容, 但是会被识别成`"1.0.0"`和`"1.0.1"`两个版本。

## 搭建一个服务器放安装包

我们在本地启动一个服务器, 放最新版本的安装包资源。

- 1、初始化

```
mkdir tasky-server
cd tasky-server
npm init -y
npm install koa koa-static --save
```





```
const Koa = require('koa')
const app = new Koa()

const static = require('koa-static')
const path = require('path')

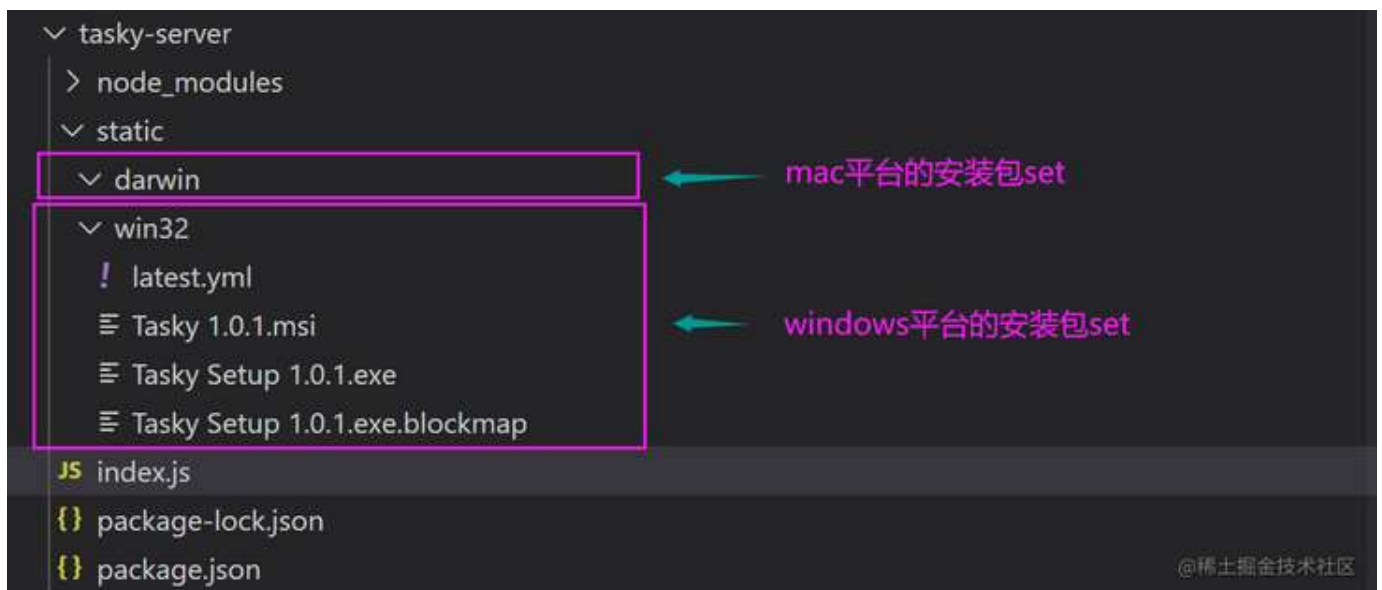
app.use(static(path.join(__dirname, './static')));

app.listen(9005)
```

- 3、在创建一个`static`文件夹，放入最新版本的安装包set。具体包含哪些文件呢？假如最新版本是"1.0.1"。

**Mac平台：** latest-mac.yml、 Tasky-1.0.1-mac.zip、 Tasky-1.0.1.dmg、 Tasky-1.0.1.dmg.blockmap

**Windows平台：** latest.yml、 Tasky 1.0.1.msi、 Tasky Setup 1.0.1.exe、 Tasky Setup 1.0.1.exe.blockmap



- 4、启动服务器。 `node index.js`

## 检测更新

检测更新可以借助`electron-updater`来实现。它结合`electron-builder`，实现起来非常简单。直接上代码。

- 第一步、在build中配置"publish"字段：



```
"build": {  
  ...  
  "publish": [  
    {  
      "provider": "generic",  
      "url": "http://127.0.0.1:9005/"  
    }  
  ]  
}
```

第二步、在应用程序主进程中调用`electron-updater`模块检测更新。

```
const { autoUpdater } = require('electron-updater')  
function checkUpdate(){  
  if(process.platform == 'darwin'){  
  
    //我们使用koa-static将静态目录设置成了static文件夹，  
    //所以访问http://127.0.0.1:9005/darwin，就相当于访问了static/darwin文件夹，win32同  
    autoUpdater.setFeedURL('http://127.0.0.1:9005/darwin') //设置要检测更新的路径  
  
  }else{  
    autoUpdater.setFeedURL('http://127.0.0.1:9005/win32')  
  }  
  
  //检测更新  
  autoUpdater.checkForUpdates()  
  
  //监听'error'事件  
  autoUpdater.on('error', (err) => {  
    console.log(err)  
  })  
  
  //监听'update-available'事件，发现有新版本时触发  
  autoUpdater.on('update-available', () => {  
    console.log('found new version')  
  })  
}
```

[注册登录](#)



## 是否需要更新是根据什么判断的呢?

electron-updater会根据上面setFeedURL指定路径下的latest.yml中的version来判断是否需要更新, 大于当前版本的version则需要更新, 否则不更新。 .yml也是一种配置文件, 有点类似于我们常用的.json配置文件, 两者写法不一样。

```
tasky-server > static > win32 > ! latest.yml
1  version: 1.0.1
2  files:
3    - url: Tasky Setup 1.0.1.exe
4      sha512: omk89eEFuV1Y7/AkyEacuMp1NkQuJaQYeZhT3V8NZIvpiX6dMaB+nlyr5D0fzE7aPmZZsUud5rz8tk
5      size: 60487406
6  path: Tasky Setup 1.0.1.exe
7  sha512: omk89eEFuV1Y7/AkyEacuMp1NkQuJaQYeZhT3V8NZIvpiX6dMaB+nlyr5D0fzE7aPmZZsUud5rz8tkCKFc
8  releaseDate: '2021-06-30T03:09:37.060Z'
```

## 基于github的方案

如果你不想搭建自己的服务器, 也可以借助github。使用github自动发布, 不用每次手动上传最新安装包资源。

### 自动发布



```
"build": {
  ...
  "publish": ['github']
}
```

第二步、在"scripts"中配置新的指令，由于github权限控制，需要GH\_TOKEN，可以在<https://github.com/settings/t...> 中生成GH\_TOKEN。

```
"scripts": {
  ...
  "release": "cross-env GH_TOKEN=ghp_KmVD3.....W2k3Pd4vV electron-builder"
}
```

第三步、`npm run release`，就会在打包后，将资源上传到github，生成release draft，你在github项目中，找到这个draft，publish release就可以了。

```
> npm run release
> tasky@1.0.1 release
> cross-env GH_TOKEN=ghp_KmVD3.....W2k3Pd4vV electron-builder

• electron-builder version=22.11.7 os=10.0.19042
• loaded configuration file=package.json ("build" field)
• writing effective config file=dist\builder-effective-config.yaml
• packaging platform=win32 arch=x64 electron=13.1.2 appOutDir=dist\win-unpacked
• building target=MSI arch=x64 file=dist\Tasky 1.0.1.msi
• building target=nsis file=dist\Tasky Setup 1.0.1.exe archs=x64 oneClick=false perMachine=true
• building block map blockMapFile=dist\Tasky Setup 1.0.1.exe.blockmap • publishing publisher=Github (owner: alasolala, project: tasky, version: 1.0.1)
• uploading file=Tasky-Setup-1.0.1.exe.blockmap provider=Github
• uploading file=Tasky-Setup-1.0.1.exe provider=Github
• creating Github release reason=release doesn't exist tag=v1.0.1 version=1.0.1
[ ] 1% 11328.8s | Tasky-Setup-1.0.1.exe to Github • overwrite published file file=Tasky-Setup-1.0.1.exe reason=already exists on Github
[ ] 4% 1918.7s | Tasky-Setup-1.0.1.exe to Github • uploading file=Tasky-1.0.1.msi provider=Github
[ ] 8% 1866.3s | Tasky-Setup-1.0.1.exe to Github
[ ] 7% 206.4s | Tasky-1.0.1.msi to Github
```

Latest release

## 1.0.1

alasolala released this 3 days ago

v1.0.1 test

Assets 6

latest.yml	338 Bytes
Tasky-1.0.1.msi	66.1 MB
Tasky-Setup-1.0.1.exe	57.4 MB
Tasky-Setup-1.0.1.exe.blockmap	62.4 KB
Source code (zip)	

## 检测更新

和上面类似，以Windows为例，代码如下。

```
const { autoUpdater } = require('electron-updater')
function checkUpdate(){
  //检测更新
  autoUpdater.checkForUpdates()

  //监听 'error' 事件
  autoUpdater.on('error', (err) => {
    console.log(err)
  })

  //监听 'update-available' 事件，发现有新版本时触发
  autoUpdater.on('update-available', () => {
    console.log('found new version')
  })

  //默认会自动下载新版本，如果不想自动下载，设置autoUpdater.autoDownload = false

  //监听 'update-downloaded' 事件，新版本下载完成时触发
  autoUpdater.on('update-downloaded', () => {
    dialog.showMessageBox({
      type: 'info',
      title: '应用更新',
      message: '发现新版本，是否更新？',
      buttons: ['是', '否']
    }).then((buttonIndex) => {
      if(buttonIndex.response == 0) { //选择是，则退出程序，安装新版本
```

## 结语

我们上面的例子中，是将页面的web资源都打包到了安装包，还有一种情况就是，web资源和“app壳子”分离，web资源放在服务器，每次都通过网络动态加载，像下面这样：

```
mainWindow.loadURL('https://juejin.cn')
```

在业务需要频繁更新的场景中，可以使用这种方式，快速无障碍地实现更新。在这种情况下，我们可以按照上述方式打包和更新“壳子”，也就是主进程相关；而页面资源的打包和普通的前端项目打包无异，这里不再赘述。



感谢你的阅读，如果觉得还不错，欢迎点赞哦❤️❤️！

更多技术交流欢迎关注我的公众号：Alasolala

前端 javascript electron npm node.js

阅读 2.4k • 更新于 6 月 18 日

👍 赞

🔖 收藏

🔗 分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



Alaso

6 声望 4 粉丝

关注作者

0 条评论

得票 最新



撰写评论 ...

提交评论

评论支持部分 Markdown 语法： **\*\*粗体\*\*** *\_斜体\_* [链接](http://example.com) `代码` - 列表 > 引用。你还可以使用 @ 来通知其他用户。

继续阅读

Electron快速入门，手把手教你编写完整实用案例

👍

🔖

💬

🔗