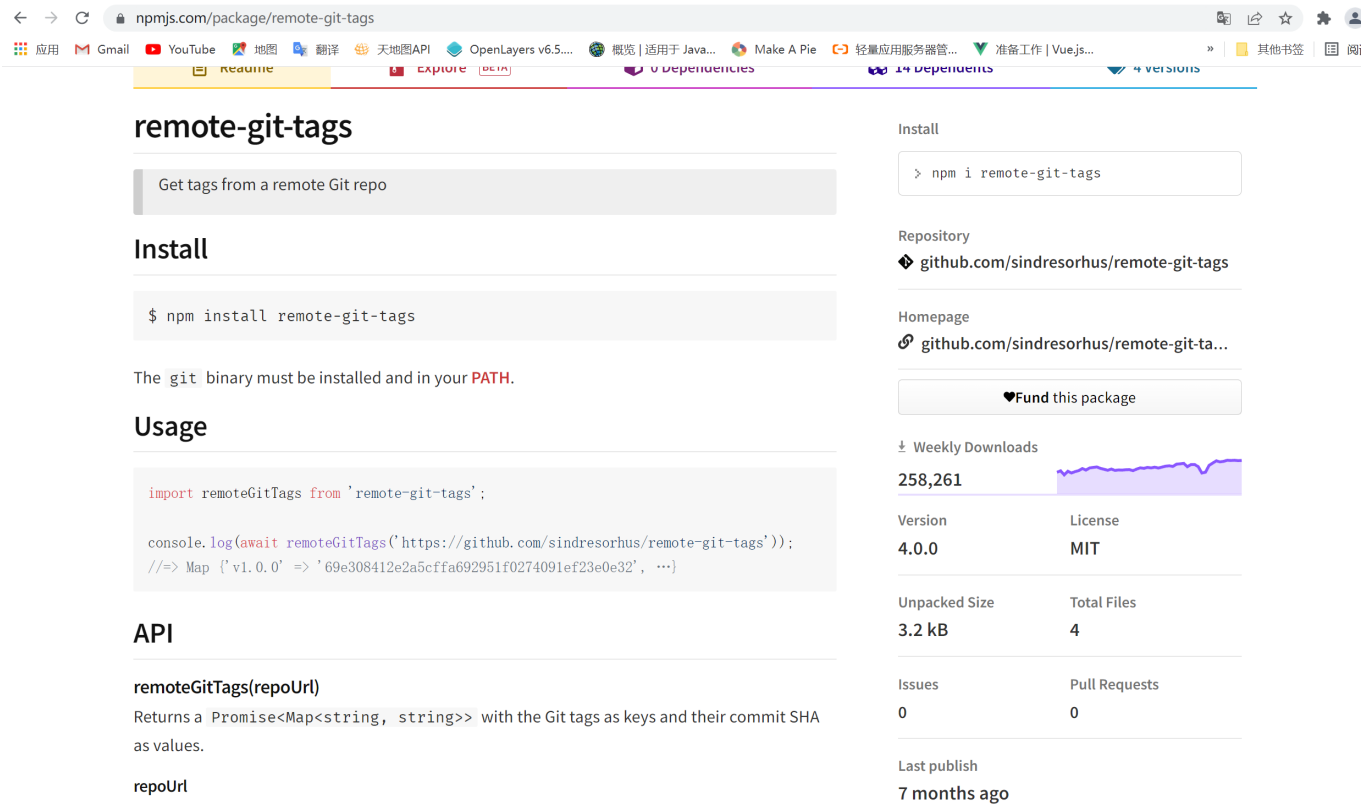
从remote-git-tags中学习promisify

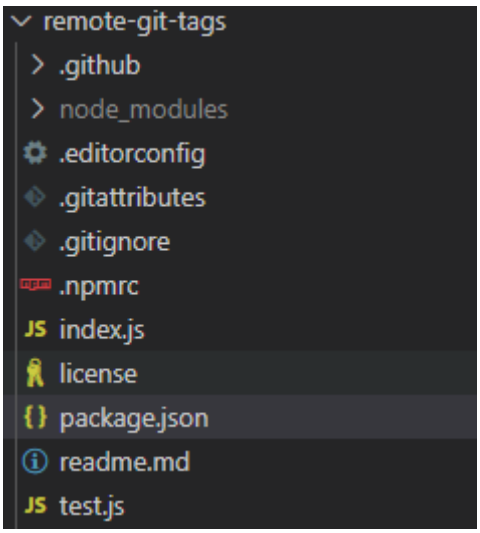
这一次学习的源码是remote-git-tags,先去npmjs看一看



使用非常简单，直接import后直接传参网址就行了。
从await中看出返回的应该是个promise

```
import remoteGitTags from 'remote-git-tags';

console.log(await remoteGitTags('https://github.com/sindresorhus/remote-git-tags'));
//=> Map {'v1.0.0' => '69e308412e2a5cffa692951f0274091ef23e0e32', …}
```

下载下来看一看



package.json

先看package.json

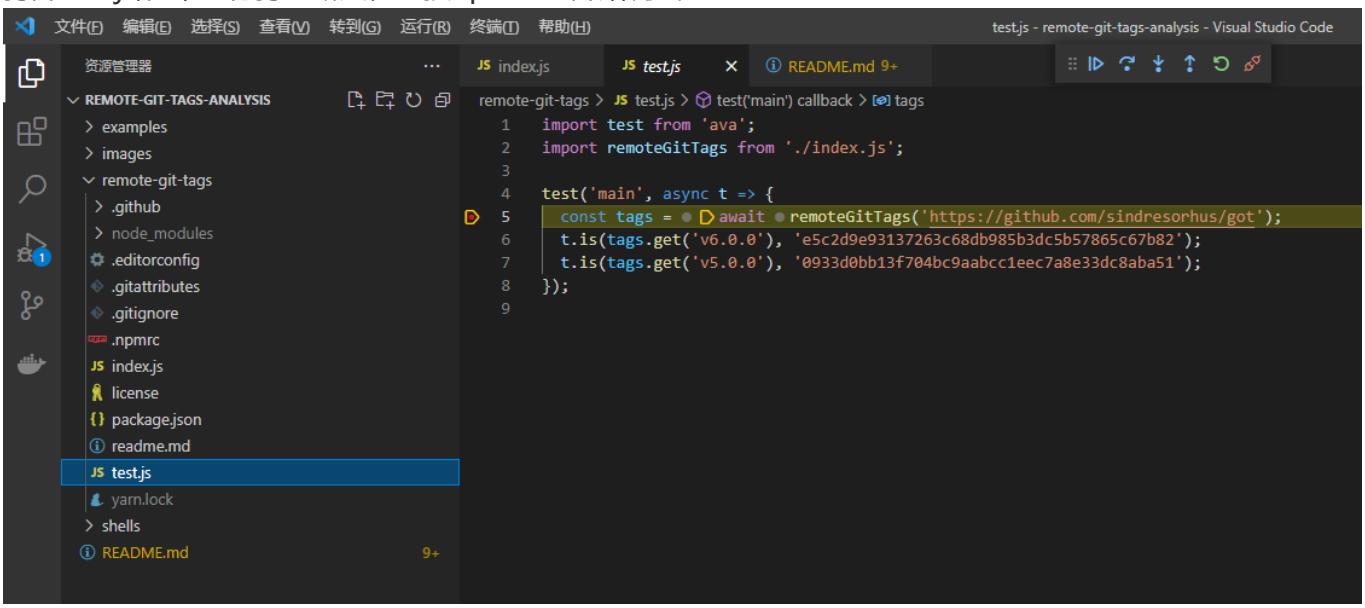

```
{
    "name": "remote-git-tags",
    "version": "4.0.0",
    "description": "Get tags from a remote Git repo",
    "license": "MIT",
    "repository": "sindresorhus/remote-git-tags",
    "funding": "https://github.com/sponsors/sindresorhus",
    "author": {
        "name": "Sindre Sorhus",
        "email": "sindresorhus@gmail.com",
        "url": "https://sindresorhus.com"
    },
    "type": "module",
    "exports": "./index.js",
    "engines": {
        "node": "^12.20.0 || ^14.13.1 || >=16.0.0"
    },
    "scripts": {
        "test": "xo && ava"
    },
    "files": [
        "index.js"
    ],
    "keywords": [
        "git",
        "tags",
        "tag",
        "remote",
        "ls-remote",
        "ls",
        "repo",
        "repository",
        "commit",
        "sha",
        "url"
    ],
    "devDependencies": {
        "ava": "^3.15.0",
        "xo": "^0.44.0"
    }
}
```


"type": "module" //指定Node用什么模块加载，默认CommonJS
写node.js时候使用import导入的时候报错SyntaxError: Cannot use import statement outside a module，就是使用了CommonJS加载的原因啦，这时候只要把type改成module就不会报错了。

.mjs文件，Node始终会用ES6模块加载
.cjs文件，Node始终会用CommonJS模块加载
.js 默认用CommonJS加载，如果同级目录有package.json文件，且type为module则使用ES6加载

调试

打开test.js,在第五行打上断点，直接npm test开始调试

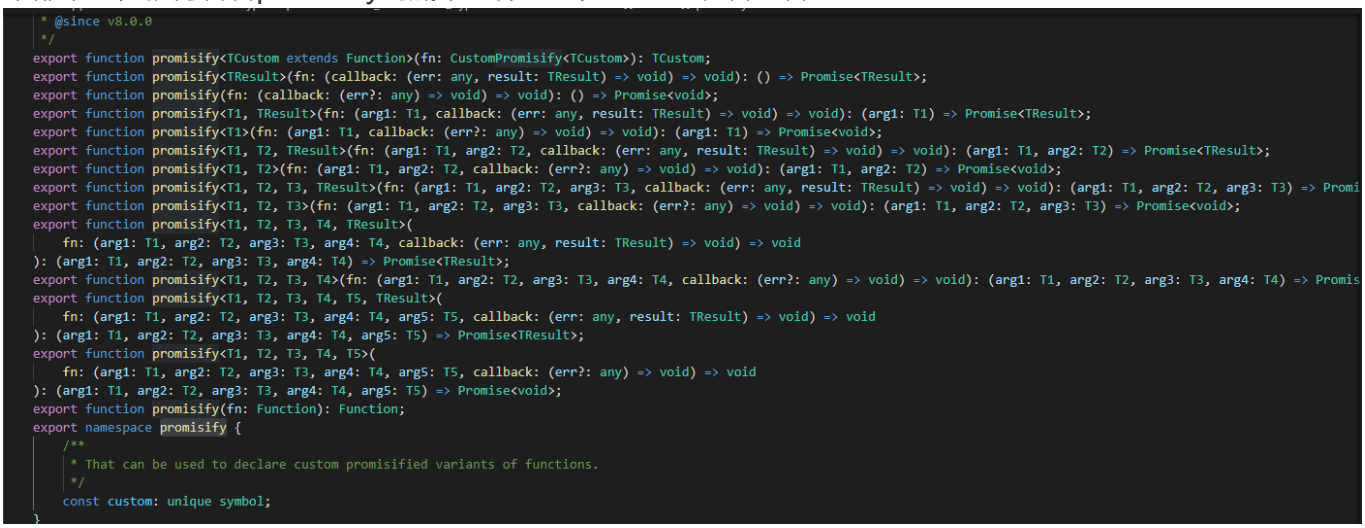


F11,单步调试，进入remoteGitTags，发现index.js只有22行代码！！！

直接把代码发出来。

```
import {promisify} from 'node:util';
import childProcess from 'node:child_process';

const execFile = promisify(childProcess.execFile);

export default async function remoteGitTags(repoUrl) {
    // execFile应该是执行 git ls-remote --tags repoUrl
    // repoUrl就是我们传入的URL
    // 解构赋值stdout，标准输出
    const {stdout} = await execFile('git', ['ls-remote', '--tags', repoUrl]);
    const tags = new Map();

    // 首位去除空格，然后以换行为间隔，切割字符串得到每一行，应该是每一行的tag
    for (const line of stdout.trim().split('\n')) {
        // 以\t为间隔，切割字符串
        // \t 的意思是 横向跳到下一制表符位置
        // 将hash 跟 tag 存入tags Map对象中
        const [hash, tagReference] = line.split('\t');

        // Strip off the indicator of dereferenced tags so we can override the
        // previous entry which points at the tag hash and not the commit hash
        // `refs/tags/v9.6.0^{}` → `v9.6.0`
        const tagName = tagReference.replace(/^refs\/tags\//, '').replace(/\^{}$/, '');

        tags.set(tagName, hash);
    }

    return tags;
}
```

来都来了，就再看看promisify到底干了什么吧，ctrl+左键看一看



```
 * @since v8.0.0
 */
export function promisify<TCustom extends Function>(fn: CustomPromisify<TCustom>): TCustom;
export function promisify<TResult>(fn: (callback: (err: any, result: TResult) => void) => void): () => Promise<TResult>;
export function promisify(fn: (callback: (err?: any) => void) => void): () => Promise<void>;
export function promisify<T1, TResult>(fn: (arg1: T1, callback: (err: any, result: TResult) => void) => void): (arg1: T1) => Promise<TResult>;
export function promisify<T1>(fn: (arg1: T1, callback: (err?: any) => void) => void): (arg1: T1) => Promise<void>;
export function promisify<T1, T2, TResult>(fn: (arg1: T1, arg2: T2, callback: (err: any, result: TResult) => void) => void): (arg1: T1, arg2: T2) => Promise<TResult>;
export function promisify<T1, T2>(fn: (arg1: T1, arg2: T2, callback: (err?: any) => void) => void): (arg1: T1, arg2: T2) => Promise<void>;
export function promisify<T1, T2, T3, TResult>(fn: (arg1: T1, arg2: T2, arg3: T3, callback: (err: any, result: TResult) => void) => void): (arg1: T1, arg2: T2, arg3: T3) => Promi
export function promisify<T1, T2, T3>(fn: (arg1: T1, arg2: T2, arg3: T3, callback: (err?: any) => void) => void): (arg1: T1, arg2: T2, arg3: T3) => Promise<void>;
export function promisify<T1, T2, T3, T4, TResult>(
    fn: (arg1: T1, arg2: T2, arg3: T3, arg4: T4, callback: (err: any, result: TResult) => void) => void
): (arg1: T1, arg2: T2, arg3: T3, arg4: T4) => Promise<TResult>;
export function promisify<T1, T2, T3, T4>(fn: (arg1: T1, arg2: T2, arg3: T3, arg4: T4, callback: (err?: any) => void) => void): (arg1: T1, arg2: T2, arg3: T3, arg4: T4) => Promis
export function promisify<T1, T2, T3, T4, T5, TResult>(
    fn: (arg1: T1, arg2: T2, arg3: T3, arg4: T4, arg5: T5, callback: (err: any, result: TResult) => void) => void
): (arg1: T1, arg2: T2, arg3: T3, arg4: T4, arg5: T5) => Promise<TResult>;
export function promisify<T1, T2, T3, T4, T5>(
    fn: (arg1: T1, arg2: T2, arg3: T3, arg4: T4, arg5: T5, callback: (err?: any) => void) => void
): (arg1: T1, arg2: T2, arg3: T3, arg4: T4, arg5: T5) => Promise<void>;
export function promisify(fn: Function): Function;
export namespace promisify {
    /**
     * That can be used to declare custom promisified variants of functions.
     */
    const custom: unique symbol;
}
```

从d.ts中差不多可以发现，promisify大概是把callback变成Promise的一个函数。

去官方文档看一看。

util.promisify(original)

中英对照

新增于: v8.0.0

- `original` <Function>
- 返回: <Function>

采用遵循常见的错误优先的回调风格的函数（也就是将 `(err, value) => ...` 回调作为最后一个参数），并返回一个返回 promise 的版本

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // 使用 `stats` 做些事情
}).catch((error) => {
  // 处理错误。
});
```

或者，等效地使用 `async function`：

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

如果存在 `original[util.promisify.custom]` 属性，则 `promisify` 将返回其值，请参阅自定义的 promise 化函数。

`promisify()` 假设 `original` 是在所有情况下都将回调作为其最后一个参数的函数。 如果 `original` 不是函数，则 `promisify()` 将抛出错误。 如果 `original` 是函数，但其最后一个参数不是错误优先的回调，则它仍然会被传入错误优先的回调作为其最后一个参数。

除非经过特殊处理，否则在类方法或其他使用 `this` 的方法上使用 `promisify()` 可能无法按预期工作：

```
const util = require('util');

class Foo {
  constructor() {
    this.a = 42;
  }

  bar(callback) {
    callback(null, this.a);
  }
}

const foo = new Foo();

const naiveBar = util.promisify(foo.bar);
// TypeError: Cannot read property 'a' of undefined
// naiveBar().then(a => console.log(a));

naiveBar.call(foo).then((a) => console.log(a)); // '42'

const bindBar = naiveBar.bind(foo);
bindBar().then((a) => console.log(a)); // '42'
```

自定义的 promise 化函数

中英对照

使用 `util.promisify.custom` 符号可以覆盖 `util.promisify()` 的返回值：

```
const util = require('util');

function doSomething(foo, callback) {
  // ...
}

doSomething[util.promisify.custom] = (foo) => {
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// 打印 'true'
```

这对于原始函数不遵循将错误优先的回调作为最后一个参数的标准格式的情况很有用。

例如，对于接受 `(foo, onSuccessCallback, onErrorCallback)` 的函数：

```
doSomething[util.promisify.custom] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
  });
};
```

如果 `promisify.custom` 已定义但不是函数，则 `promisify()` 将抛出错误。

util.promisify.custom

中英对照

▶ 版本历史

- `<symbol>` 可用于声明自定义的 promise 化的函数变体，参阅 自定义的 promise 化函数。

除了可以通过 `util.promisify.custom` 访问之外，此符号是 全局地注册，可以在任何环境中作为 `Symbol.for('nodejs.util.promisify.custom')` 访问。

例如，对于接受 `(foo, onSuccessCallback, onErrorCallback)` 的函数：

```
const kCustomPromisifiedSymbol = Symbol.for('nodejs.util.promisify.custom');

doSomething[kCustomPromisifiedSymbol] = (foo) => {
  return new Promise((resolve, reject) => {
    doSomething(foo, resolve, reject);
  });
};
```

文档详细介绍了promisify的功能，参考文档
util.promisify(original)
original <Function>
返回: <Function>
这都明示是闭包了，那就自己写一个简版的promisify(original)吧
采用遵循常见的错误优先的回调风格的函数（也就是将 (err, value) => ... 回调作为最后一个参数），并返回一个返回 promise 的版本。

```
function promisify(original) {
    return (...args) => {
        return new Promise((resolve,reject)=>{
           // 错误优先的回调风格函数
           args.push((err,value)=>{
               if (err) {
                   return reject(err)
               }
               resolve(value)
           })
           original.apply(this,args)
        })
    }
}
```

写下来有个地方还是蛮纠结的: original.apply第一个参数应该传什么?
一开始认为是null，但是看了源码之后发现用的是this，查阅资料后，如果传null/undefined的话，会默认指向window，那就错大发了!
对比川哥的代码，对于回调函数的参数川哥用了...values，我受了文档的束缚，想也没想直接就写了value，仔细斟酌一下确实应该使用...values，查看源码用的也是...values。
那么到了这里，promisify也就成功完成了简化版，源码中还有对于返回值细分，以及自定义promise化函数。

总结

1.川哥的循序渐进的自己模拟出源码的过程值得学习!
2.以后针对一些自己认为简单的功能，可以尝试自己先实现（可以参考文档，可以参考d.ts文件），然后再去

阅读源码，可以发现自己与大佬们的差距。这些差距正是自己所欠缺的！这样比单纯的阅读源码收获来得大得多！

TODO: 源码用了Reflect.apply,静态方法。调用函数，同时可以对函数进行类似调用Function.prototype.apply()。 两者之间有没有啥区别？