**Promise-Function（p-fun）**

|  |  |  |
| --- | --- | --- |
| **模块** | **简介** | **描述** |
| pify | Promisify a callback-style function | (async () => {  const data = await pify(fs.readFile)('package.json', 'utf8'); // Promisify a single function  console.log(JSON.parse(data).name); //=> 'pify'  const data2 = await pify(fs, {multiArgs: true}).readFile('package.json', 'utf8'); // Promisify all methods  console.log(JSON.parse(data2).name); //=> 'pify'  })(); |
| delay | Delay a promise a specified amount of time | (async () => {  bar();  await delay(100, {value: 'default\_return\_value'});  baz(); // Executed 100 milliseconds later  await delay(100, 2000); // random milliseconds between minimum and maximum  })(); |
| p-map | Map over promises concurrently, Useful when you need to run promise-returning & async functions multiple times with different inputs concurrently. This is different from Promise.all() in that you can control the concurrency and also decide whether or not to stop iterating when there's an error | const sites = [  getWebsiteFromUsername('https://sindresorhus'), //=> Promise  'https://ava.li', 'https://github.com'  ];  (async () => {  const mapper = async site => {  const {requestUrl} = await got.head(site);  return requestUrl;  };  const result = await pMap(sites, mapper, {concurrency: 2, stopOnError: false});  console.log(result); //=> ['https://sindresorhus.com/', 'https://ava.li/', 'https://github.com/']  })(); |
| p-all | Run promise-returning & async functions concurrently with optional limited concurrency, Similar to Promise.all(), but accepts functions instead of promises directly so you can limit the concurrency | (async () => {  const actions = [  () => got('https://sindresorhus.com'),  () => got('https://ava.li'), () => checkSomething(), () => doSomethingElse()  ];  console.log(await pAll(actions, {concurrency: 2, stopOnError: false}));  })(); |
| p-queue | Promise queue with concurrency control, Useful for rate-limiting async (or sync) operations. For example, when interacting with a REST API or when doing CPU/memory intensive tasks | const {default: PQueue} = require('p-queue');  const queue = new PQueue({concurrency: 1, timeout: 9000, autoStart: true, intervalCap: 1, interval: 1});  (async () => {  await queue.add(() => got('https://sindresorhus.com'));  console.log('Done: sindresorhus.com');  await queue.add(() => got('https://avajs.dev'), {priority: 1});  console.log('Done: avajs.dev');  const task = await getUnicornTask();  await queue.add(task);  console.log('Done: Unicorn task');  })(); |
| p-catch-if | Conditional promise catch handler, Useful for handling only some types of errors and let the rest pass through | getData().catch(pCatchIf(TimeoutError, () => retry(getData))); // Error constructor  getData().catch(pCatchIf([NetworkError, TimeoutError], () => retry(getData))); // Multiple error constructors  getData().catch(pCatchIf(isProduction, error => recordError(error))); // Boolean  const hasUnicornMessage = error => error.message.includes('unicorn'); // Function  getData().catch(pCatchIf(hasUnicornMessage, console.error));  const handler = error => sendError(error).then(checkResults); // Promise-returning function  getData().catch(pCatchIf(handler, console.error));  const validateMessage = error => error.message === 'Too many rainbows'; // Can also be nested  getData().catch(pCatchIf(UnicornError, pCatchIf(validateMessage, console.error))); |
| p-if | Conditional promise chains, | getData()  .then(pIf(process.env.NODE\_ENV !== 'production', addDebugInfo))  .then(data => {  console.log(data);  });  getList()  .then(pIf(shouldSort, pIf(sortDirection === 'ascending', sort.asc, sort.desc)))  .then(list => {  console.log(list);  }); |
| p-tap | Tap into a promise chain without affecting its value or state | Promise.resolve('unicorn')  .then(pTap(console.log)) // Logs `unicorn`  .then(value => {  // `value` is still `unicorn`  });  getUser()  .then(pTap(user => recordStatsAsync(user))) // Stats are saved about `user` async  .then(user => {  // `user` is the user from getUser(), not recordStatsAsync()  });  Promise.resolve(() => doSomething())  .catch(pTap.catch(console.error)) // prints any errors  .then(handleSuccess).catch(handleError); |
| p-log | Log the value/error of a promise | Promise.resolve('unicorn')  .then(pLog()) // Logs `unicorn`  .then(value => {  // `value` is still `unicorn`  });  Promise.resolve()  .then(() => { throw new Error('pony');})  .catch(pLog.catch()) // Logs `Error: pony`  .catch(error => {  // `error` is still `Error: pony`  }); |
| p-event | Promisify an event by waiting for it to be emitted, Useful when you need only one event emission and want to use it with promises or await it in an async function | (async () => {  try {  const result = await pEvent(emitter, 'finish', {multiArgs: true});  console.log(result); // `emitter` emitted a `finish` event  } catch (error) {  console.error(error); // `emitter` emitted an `error` event  }  })();  (async () => {  const asyncIterator = pEvent.iterator(emitter, 'data', {  resolutionEvents: ['finish']  });  for await (const event of asyncIterator) {  console.log(event);  }  })(); |
| p-debounce | Debounce promise-returning & async functions | const expensiveCall = async input => input;  const debouncedFn = pDebounce(expensiveCall, 200, {leading: false});  for (const i of [1, 2, 3]) {  debouncedFn(i).then(console.log);  }  //=> 3, //=> 3, //=> 3 |
| p-throttle | Throttle promise-returning & async functions, Useful for rate limiting calls to an external API | const now = Date.now();  const throttled = pThrottle(index => {  const secDiff = ((Date.now() - now) / 1000).toFixed();  return Promise.resolve(`${index}: ${secDiff}s`);  }, 2, 1000); // limit:2, interval: 1000  for (let i = 1; i <= 6; i++) {  throttled(i).then(console.log);  }  //=> 1: 0s //=> 2: 0s //=> 3: 1s //=> 4: 1s //=> 5: 2s //=> 6: 2s |
| p-timeout | Timeout a promise after a specified amount of time | const delayedPromise = delay(200);  pTimeout(delayedPromise, 50).then(() => 'foo');  //=> [TimeoutError: Promise timed out after 50 milliseconds]  const delayedPromise = () => delay(200);  pTimeout(delayedPromise(), 50, () => {  return pTimeout(delayedPromise(), 300);  }); |
| p-finally | Promise#finally() ponyfill - Invoked when the promise is settled regardless of outcome, Useful for cleanup | const directory = createTempDirectory();  (async () => {  await pFinally(write(directory), () => {  cleanup(directory);  });  }); |
| p-retry | Retry a promise-returning or async function, It does exponential backoff and supports custom retry strategies for failed operations | const run = async (args) => {  const response = await fetch('https://sindresorhus.com/unicorn');  // Abort retrying if the resource doesn't exist  if (response.status === 404) {  throw new pRetry.AbortError(response.statusText);  }  return response.blob();  };  (async () => {  // console.log(await pRetry(run, {retries: 5}));  console.log(await pRetry(() => run(‘args’), {retries: 5}));  })(); |
| p-any | Wait for any promise to be fulfilled, Useful when you need the fastest promise | (async () => {  const first = await pAny([  got.head('https://github.com').then(() => 'github'),  got.head('https://google.com').then(() => 'google'),  got.head('https://twitter.com').then(() => 'twitter'),  ]);  console.log(first); //=> 'google'  })(); |
| p-some | Wait for a specified number of promises to be fulfilled, Useful when you need the fastest of multiple promises | (async () => {  const input = [  got.head('github.com').then(() => 'github'),  got.head('google.com').then(() => 'google'),  got.head('twitter.com').then(() => 'twitter'),  got.head('medium.com').then(() => 'medium')  ];  const [first, second] = await pSome(input, {count: 2});  console.log(first, second); //=> 'google twitter'  })(); |
| p-locate | Get the first fulfilled promise that satisfies the provided testing function, Think of it like an async version of Array#find | // Here we find the first file that exists on disk, in array order.  const pathExists = require('path-exists');  const files = [  'unicorn.png',  'rainbow.png', // Only this one actually exists on disk  'pony.png'  ];  (async () => {  const foundPath = await pLocate(files, file => pathExists(file), {concurrency: 1, preserveOrder: true});  console.log(foundPath); //=> 'rainbow'  })(); |
| p-limit | Run multiple promise-returning & async functions with limited concurrency | const limit = pLimit(1); // Only one promise is run at once  const input = [  limit(() => fetchSomething('foo')),  limit(() => fetchSomething('bar')),  limit(() => doSomething())  ];  (async () => {  const result = await Promise.all(input);  console.log(result);  })(); |
| p-series | Run promise-returning & async functions in series, If you're doing the same work in each function, use p-each-series instead, See p-all for a concurrent counterpart | (async () => {  const tasks = [  () => got('https://sindresorhus.com'),  () => checkSomething(),  () => doSomethingElse()  ];  console.log(await pSeries(tasks));  })(); |
| p-memoize | Memoize promise-returning & async functions, Useful for speeding up consecutive function calls by caching the result of calls with identical input | const memGot = pMemoize(got, {maxAge: 1000});  (async () => {  memGot('https://sindresorhus.com');  // This call is cached  memGot('https://sindresorhus.com');  setTimeout(() => {  // This call is not cached as the cache has expired  memGot('https://sindresorhus.com');  }, 2000);  })(); |
| p-pipe | Compose promise-returning & async functions into a reusable pipeline | const addUnicorn = async string => `${string} Unicorn`;  const addRainbow = async string => `${string} Rainbow`;  const pipeline = pPipe(addUnicorn, addRainbow);  (async () => {  console.log(await pipeline('❤️')); //=> '❤️ Unicorn Rainbow'  })(); |
| p-props | Like Promise.all() but for Map and Object, Useful when you need to run multiple promises concurrently and keep track of the fulfilled values by name | (async () => {  const fetch = async url => {  const {body} = await got(url);  return body;  };  const sites = {  ava: fetch('ava.li'), todomvc: fetch('todomvc.com'), github: fetch('github.com'), foo: 'bar'  };  console.log(await pProps(sites));  // {ava: '<!doctype …', todomvc: '<!doctype …', github: '<!doctype …', foo: 'bar'}  })(); |
| p-waterfall | Run promise-returning & async functions in series, each passing its result to the next | (async () => {  const tasks = [  initialValue => getEmoji(initialValue),  previousValue => `I ❤️ ${previousValue}`  ];  console.log(await pWaterfall(tasks, 'unicorn')); //=> 'I ❤️ 🦄'  })(); |
| p-cancelable | Create a promise that can be canceled, Useful for animation, loading resources, long-running async computations, async iteration | const cancelablePromise = new PCancelable((resolve, reject, onCancel) => {  const worker = new SomeLongRunningOperation();  onCancel(() => {  worker.close();  });  worker.on('finish', resolve);  worker.on('error', reject);  });  (async () => {  try {  console.log('Operation finished successfully:', await cancelablePromise);  } catch (error) {  if (cancelablePromise.isCanceled) {  console.log('Operation was canceled'); // Handle the cancelation here  return;  }  throw error;  }  })();  setTimeout(() => { // Cancel the operation after 10 seconds  cancelablePromise.cancel('Unicorn has changed its color');  }, 10000); |
| p-progress | Create a promise that reports progress, Useful for reporting progress to the user during long-running async operations | const progressPromise = new PProgress((resolve, reject, progress) => {  const job = new Job();  job.on('data', data => {  progress(data.length / job.totalSize);  });  job.on('finish', resolve);  job.on('error', reject);  });  (async () => {  progressPromise.onProgress(progress => {  console.log(`${progress \* 100}%`);  //=> 9%, => 23%, //=> 59%, //=> 75%, //=> 100%  });  await progressPromise;  })(); |
| p-reflect | Make a promise always fulfill with its actual fulfillment value or rejection reason, Useful when you want a promise to fulfill no matter what and would rather handle the actual state afterwards | (async () => {  const promises = [getPromise(), etPromiseThatRejects(), etPromise()];  const results = await Promise.all(promises.map(pReflect));  console.log(results);  /\*[{isFulfilled: true, isRejected: false, value: '🦄'}, {isFulfilled: false,isRejected: true,  reason: [Error: 👹]},{isFulfilled: true,isRejected: false,value: '🐴'}]\*/  const resolvedString = results.filter(result => result.isFulfilled).map(result => result.value).join('');  console.log(resolvedString); //=> '🦄🐴'  })(); |
| p-filter | Filter promises concurrently, Useful when you need to run promise-returning & async functions multiple times with different inputs concurrently and get a filtered down result | const places = [  getCapital('Norway').then(info => info.name), 'Bangkok, Thailand', 'Berlin, Germany', 'Tokyo, Japan'  ];  const filterer = async place => {  const weather = await getWeather(place);  return weather.temperature > 30;  };  (async () => {  const result = await pFilter(places, filterer, {concurrency: 1});  console.log(result); //=> ['Bangkok, Thailand']  })(); |
| p-reduce | Reduce a list of values using promises into a promise for a value, Useful when you need to calculate some accumulated value based on async resources | (async () => {  const names = [  getUser('sindresorhus').then(info => info.name), 'Addy Osmani',  'Pascal Hartig', 'Stephen Sawchuk'];  const totalAge = await pReduce(names, async (total, name) => {  const info = await humanInfo(name);  return total + info.age;  }, 0);  console.log(totalAge); //=> 125  })(); |
| p-settle | Settle promises concurrently and get their fulfillment value or rejection reason | (async () => {  const files = ['a.txt', 'b.txt' // Doesn't exist  ].map(fileName => fs.readFile(fileName, 'utf8'));  console.log(await pSettle(files, {concurrency: 1}));  // [{isFulfilled: true, isRejected: false,value: '🦄'},{isFulfilled: false, isRejected: true,  reason: [Error: ENOENT: no such file or directory, open 'b.txt']}]  })(); |
| p-every | Test whether all promises passes a testing function, Like Array.every for promises | const places = [  getCapital('Norway').then(info => info.name), 'Bangkok,Thailand','Berlin, Germany','Tokyo, Japan'];  const testFunction = async place => {  const continent = await getContinent(place);  return continent === 'europe';  }  (async () => {  const result = await pEvery(places, testFunction, {concurrency: 1});  console.log(result); //=> false  })(); |
| p-one | Test whether some promise passes a testing function, Like Array.some for promises | const places = [  getCapital('Norway').then(info => info.name), 'Bangkok,Thailand','Berlin, Germany','Tokyo, Japan'];  const testFunction = async place => {  const continent = await getContinent(place);  return continent === 'europe';  }  (async () => {  const result = await pOne(places, testFunction, {concurrency: 1})  console.log(result); //=> true  })(); |
| p-map-series | Map over promises serially, Useful as a side-effect mapper. Use p-map if you don't need side-effects, as it's concurrent | const keywords = [getTopKeyword(), 'rainbow', 'pony'];  let scores = [];  const mapper = async keyword => {  const score = await fetchScore(keyword);  scores.push(score);  return {keyword, score};  });  (async () => {  console.log(await pMapSeries(keywords, mapper));  // [{keyword: 'unicorn',score: 99},{keyword: 'rainbow',score: 70},{keyword: 'pony',score: 79}]  })(); |
| p-each-series | Iterate over promises serially, Useful as a side-effect iterator. Prefer p-map if you don't need side-effects, as it's concurrent | const keywords = [getTopKeyword(), 'rainbow', 'pony'];  const iterator = async element => saveToDiskPromise(element);  (async () => {  console.log(await pEachSeries(keywords, iterator)); //=> ['unicorn', 'rainbow', 'pony']  })(); |
| p-times | Run promise-returning & async functions a specific number of times concurrently | (async () => {  const result = await pTimes(5, i => createFixture(`🦄-${i + 1}`), {concurrency:1, stopOnError: true});  console.log(`Created fixtures: ${result.join(' ')}`);  //=> 'Created fixtures: 🦄-1 🦄-2 🦄-3 🦄-4 🦄-5'  })(); |
| p-lazy | Create a lazy promise that defers execution until .then() or .catch() is called, Useful if you're doing some heavy operations and would like to only do it when the promise is actually used | const lazyPromise = new PLazy(resolve => {  someHeavyOperation(resolve);  });  // `someHeavyOperation` is not yet called  (async () => {  await doSomethingFun; // `someHeavyOperation` is called  console.log(await lazyPromise);  })(); |
| p-whilst | While a condition returns true, calls a function repeatedly, and then resolves the promise, Think async version of the while statement | (async () => {  let count = 0;  await pWhilst( () => count < 5, () => count++ );  console.log(count); //=> 5  })(); |
| p-do-whilst | Calls a function repeatedly while a condition returns true and then resolves the promise, Think async version of the do…while statement | (async () => {  let count = 0;  await pDoWhilst( () => count++, () => count < 5 );  console.log(count); //=> 5  })(); |
| p-forever | Run promise-returning & async functions repeatedly until you end it, Think of it like an async version of while (true) {} | pForever(async i => {  i++;  if (i > 100) {  return pForever.end;  }  await createFixture(i);  return i;  }, 0); |
| p-wait-for | Wait for a condition to be true, Can be useful for polling | const pathExists = require('path-exists');  (async () => {  await pWaitFor(() => pathExists('unicorn.png'), { interval: 200, timeout: 5000});  console.log('Yay! The file now exists.');  })(); |
| p-min-delay | Delay a promise a minimum amount of time, While the delay module delays the promise a specified amount of time and then resolves it, this module ensures the promise resolves after the specified amount of time | (async () => {  const value = await pMinDelay(somePromise, 1000);  // Executed after minimum 1 second even if `somePromise` fulfills before that  })(); |
| p-try | Promise.try() ponyfill - Starts a promise chain | (async () => {  try {  const value = await pTry(() => {  return synchronousFunctionThatMightThrow();  });  console.log(value);  } catch (error) {  console.error(error);  }  })(); |
| p-race | A better Promise.race(),This fixes the silly behavior of Promise.race() returning a forever pending promise when supplied an empty iterable, which could create some really hard to debug problems | Promise.race([]);  // Returns a forever pending promise…  pRace([]);  //=> [RangeError: Expected the input to contain at least one item] |
| p-immediate | Returns a promise resolved in the next event loop - think setImmediate(),Promises are by default resolved in a microtask (current event loop) | (async () => {  await pImmediate();  // Executed in the next event loop  console.log('🦄');  })(); |
| p-time | Measure the time a promise takes to resolve | (async () => {  const promise = pTime(execa)('sleep', ['1']);  await promise;  console.log(promise.time); //=> 1016  })(); |
| p-defer | Create a deferred promise, Don't use this unless you know what you're doing. Prefer the Promise constructor | function delay(ms) {  const deferred = pDefer();  setTimeout(deferred.resolve, ms, '🦄');  return deferred.promise;  }  (async () => {  console.log(await delay(100)); //=> '🦄'  })(); |
| p-break | Break out of a promise chain, See "How do I break out of a promise chain?" for a better way. | // Here the onlyRunConditional promises are skipped if conditional is false:  alwaysRun1()  .then(() => alwaysRun2())  .then(conditional => conditional || pBreak('🦄'))  .then(() => onlyRunConditional1())  .then(() => onlyRunConditional2())  .then(() => onlyRunConditional3())  .then(() => onlyRunConditional4())  .catch(pBreak.end)  .then(console.log);  //=> '🦄' |
| p-is-promise | Check if something is a promise, Can be useful if you need to create a fast path for a synchronous operation | const Bluebird = require('bluebird');  pIsPromise(Promise.resolve('🦄')); //=> true  pIsPromise(Bluebird.resolve('🦄')); //=> true  pIsPromise('🦄'); //=> false |
| loud-rejection | Make unhandled promise rejections fail loudly instead of the default silent fail, By default, promises fail silently if you don't attach a .catch() handler to them | const loudRejection = require('loud-rejection');  const promiseFunction = require('promise-fn');  loudRejection(); // Install the `unhandledRejection` listeners  promiseFunction(); |
| hard-rejection | Make unhandled promise rejections fail hard right away instead of the default silent fail | const hardRejection = require('hard-rejection');  const promiseFunction = require('some-promise-fn');  hardRejection(); // Install the handler  promiseFunction(); |

FAQ

**1、How can I run 100 async/promise-returning functions with only 5 running at once?**

This is a good use-case for p-map. You might ask why you can't just specify an array of promises. Promises represent values of a computation and not the computation itself - they are eager. So by the time p-map starts reading the array, all the actions creating those promises have already started running. p-map works by executing a promise-returning function in a mapper function. This way the promises are created lazily and can be concurrency limited. Check out p-all instead if you're using different functions to get each promise.

const pMap = require('p-map');

const urls = ['sindresorhus.com','ava.li','github.com',…];

const mapper = url => {return fetchStats(url);};

pMap(urls, mapper, {concurrency: 5}).then(result => {

console.log(result); //=> [{url: 'sindresorhus.com', stats: {…}}, …]

});

**2、How can I reduce nesting?**

Let's say you want to fetch some data, process it, and return both the data and the processed data. The common approach would be to nest the promises:

const getData = id => Storage.find(id).then(data => {

return process(data).then(result => {

return prepare(data, result);

});

});

But we can take advantage of Promise.all:

const getData = id => Storage.find(id).then(data => Promise.all([data, process(data)])

.then(([data, result]) => prepare(data, result));

And even simpler with async functions: (Requires Babel or Node.js 8)

const getData = async id => { const data = await Storage.find(id);

return prepare(data, await process(data));

};

**3、What about something like Bluebird#spread()?**

Promise.resolve([1, 2]).spread((one, two) => {

console.log(one, two); //=> 1 2

});

Instead of Bluebird, use destructuring:

Promise.resolve([1, 2]).then(([one, two]) => {

console.log(one, two); //=> 1 2

});

**4、What about something like Bluebird.join()?**

Promise.join(p1, p2, p3, (r1, r2, r3) => {

// …

});

Instead of Bluebird, use an async function and destructuring:

const [r1, r2, r3] = await Promise.all([p1, p2, p3]);

// …

**5、How do I break out of a promise chain?**

You might think you want to break out ("return early") when doing conditional logic in promise chains. Here you would like to only run the onlyRunConditional promises if conditional is truthy.

alwaysRun1()

.then(() => alwaysRun2())

.then(conditional => conditional || somehowBreakTheChain())

.then(() => onlyRunConditional1())

.then(() => onlyRunConditional2())

.then(() => onlyRunConditional3())

.then(() => onlyRunConditional4())

.then(() => alwaysRun3());

You could implement the above by abusing the promise rejection mechanism. However, it would be better to branch out the chain instead. Promises can not only be chained, but also nested and unnested.

const runConditional = () =>

onlyRunConditional1()

.then(() => onlyRunConditional2())

.then(() => onlyRunConditional3())

.then(() => onlyRunConditional4());

alwaysRun1()

.then(() => alwaysRun2())

.then(conditional => conditional && runConditional())

.then(() => alwaysRun3());