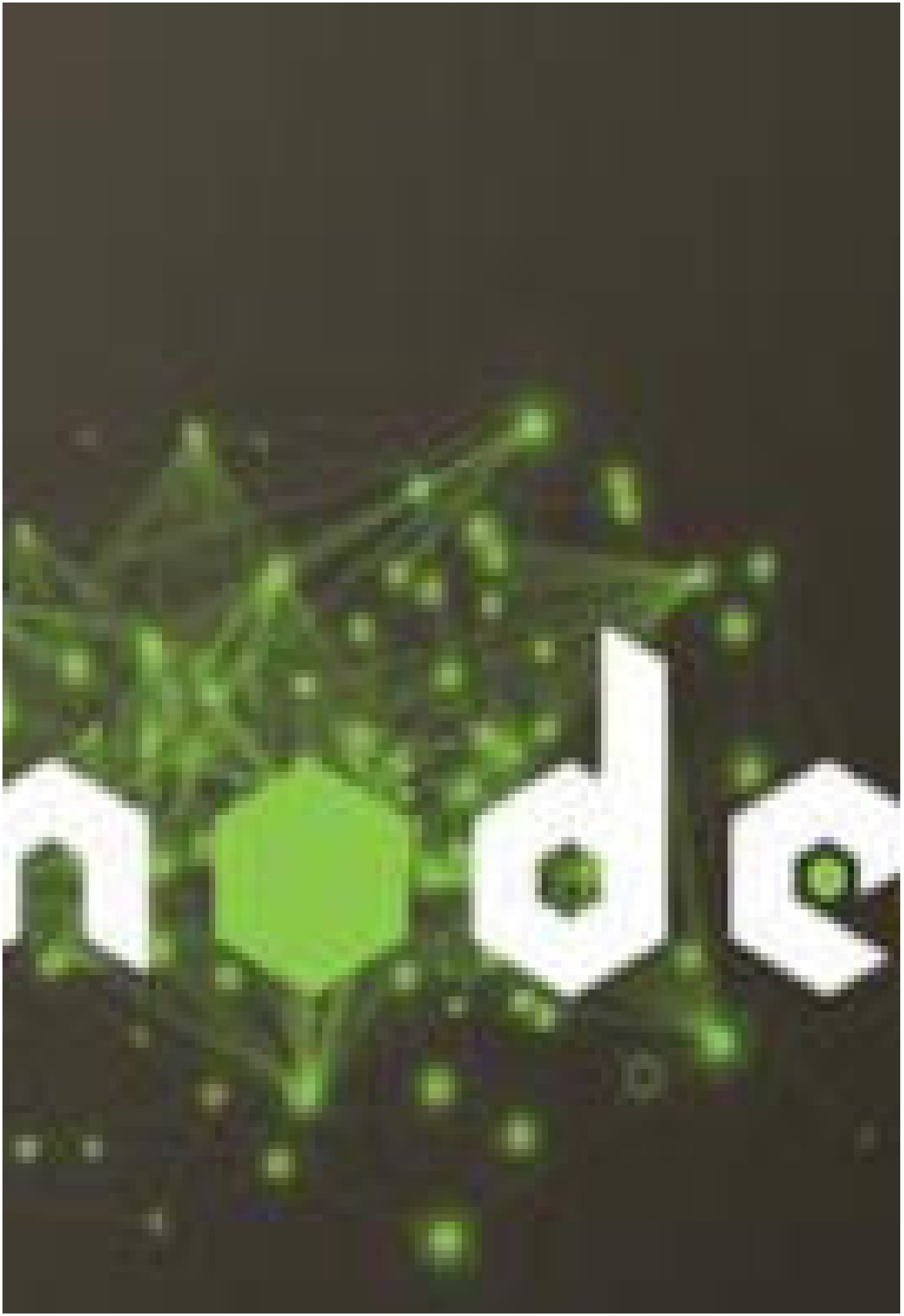

简介

> NodeJS的错误处理让人痛苦，在很长的一段时间里，大量的错误被放任不管。但是要想建立一个健壮的Node.js程序就必须正确的处理这些错误，而且这并不难学。如果你实在没有耐心，那就直接绕过长篇大论跳到“总结”部分吧。

原文

这篇文章会回答NodeJS初学者的若干问题：

- 我写的函数里什么时候该抛出异常，什么时候该传给callback, 什么时候触发`EventEmitter`等等。
- 我的函数对参数该做出怎样的假设？我应该检查更加具体的约束么？例如参数是否非空，是否大于零，是不是看起来像个IP地址，等等等。
- 我该如何处理那些不符合预期的参数？我是应该抛出一个异常，还是把错误传递给一个callback。
- 我该怎么在程序里区分不同的异常（比如“请求错误”和“服务不可用”）？
- 我怎么才能提供足够的信息让调用者知晓错误细节。
- 我该怎么处理未预料的出错？我是应该用`try/catch`，`domains`还是其它什么方式呢？

这篇文章可以划分成互相为基础的几个部分：

- **背景**：希望你所具备的知识。
- **操作失败和程序员的失误**：介绍两种基本的异常。
- **编写新函数的实践**：关于怎么让函数产生有用报错的基本原则。
- **编写新函数的具体推荐**：编写能产生有用报错的、健壮的函数需要的一个检查列表

- **例子**：以 `connect` 函数为例的文档和序言。
- **总结**：全文至此的观点总结。
- **附录：Error对象属性约定**：用标准方式提供一个属性列表，以提供更多信息。

背景

本文假设：

- 你已经熟悉了JavaScript、Java、 Python、 C++ 或者类似的语言中异常的概念，而且你知道抛出异常和捕获异常是什么意思。
- 你熟悉怎么用NodeJS编写代码。你使用异步操作的时候会很自在，并能用 `callback(err,result)` 模式去完成异步操作。你得知道下面的代码不能正确处理异常的原因是什么[脚注1]

```
function myApiFunc(callback)
{
/*
 * This pattern does NOT work!
 */
try {
  doSomeAsynchronousOperation(function (err) {
    if (err)
      throw (err);
    /* continue as normal */
  });
} catch (ex) {
  callback(ex);
}
}
```

你还要熟悉三种传递错误的方式: - 作为异常抛出。 - 把错误传给一个callback，这个函数正是为了处理异常和处理异步操作返回结果的。 - 在EventEmitter上触发一个Error事件。

接下来我们会详细讨论这几种方式。这篇文章不假设你知道任何关于domains的知识。

最后，你应该知道在JavaScript里，错误和异常是有区别的。错误是Error的一个实例。错误被创建并且直接传递给另一个函数或者被抛出。如果一个错误被抛出了那么它就变成了一个异常[脚注2]。举个例子：

```
throw new Error('something bad happened');
```

但是使用一个错误而不抛出也是可以的

```
callback(new Error('something bad happened'));
```

这种用法更常见，因为在NodeJS里，大部分的错误都是异步的。实际上，`try/catch`唯一常用的是在`JSON.parse`和类似验证用户输入的地方。接下来我们会看到，其实很少要捕获一个异步函数里的异常。这一点和Java，C++，以及其它严重依赖异常的语言很不一样。

操作失败和程序员的失误

把错误分成两大类很有用[脚注3]：

- **操作失败** 是正确编写的程序在运行时产生的错误。它并不是程序的Bug，反而经常是其它问题：系统本身（内存不足或者打开文件数过多），系统配置

（没有到达远程主机的路由），网络问题（端口挂起），远程服务（500错误，连接失败）。例子如下：

- 连接不到服务器
- 无法解析主机名
- 无效的用户输入
- 请求超时
- 服务器返回500
- 套接字被挂起
- 系统内存不足

- **程序员失误** 是程序里的Bug。这些错误往往可以通过修改代码避免。它们永远都没法被有效的处理。

- 读取 undefined 的一个属性

- 调用异步函数没有指定回调

- 该传对象的时候传了一个字符串

- 该传IP地址的时候传了一个对象

人们把操作失败和程序员的失误都称为“错误”，但其实它们很不一样。操作失败是所有正确的程序应该处理的错误情形，只要被妥善处理它们不一定会预示着Bug或是严重的问题。“文件找不到”是一个操作失败，但是它并不一定意味着哪里出错了。它可能只是代表着程序如果想用一个文件得事先创建它。

与之相反，程序员失误是彻彻底底的Bug。这些情形下你会犯错：忘记验证用户输入，敲错了变量名，诸如此类。这样的错误根本就没法被处理，如果可以，那就意味着你用处理错误的代码代替了出错的代码。

这样的区分很重要：操作失败是程序正常操作的一部分。而由程序员的失误则是Bug。

有的时候，你会在一个Root问题里同时遇到操作失败和程序员的失误。HTTP服

务器访问了未定义的变量时奔溃了，这是程序员的失误。当前连接着的客户端会在程序崩溃的同时看到一个 ECONNRESET 错误，在NodeJS里通常会被报成“Socket Hang-up”。对客户端来说，这是一个不相关的操作失败, 那是因为正确的客户端必须处理服务器宕机或者网络中断的情况。

类似的，如果不处理好操作失败, 这本身就是一个失误。举个例子，如果程序想要连接服务器，但是得到一个 ECONNREFUSED 错误，而这个程序没有监听套接字上的 error 事件,然后程序崩溃了，这是程序员的失误。连接断开是操作失败（因为这是任何一个正确的程序在系统的网络或者其它模块出问题时都会经历的），如果它不被正确处理，那它就是一个失误。

理解操作失败和程序员失误的不同, 是搞清怎么传递异常和处理异常的基础。明白了这点再继续往下读。

处理操作失败

就像性能和安全问题一样，错误处理并不是可以凭空加到一个没有任何错误处理的程序中的。你没有办法在一个集中的地方处理所有的异常，就像你不能在一个集中的地方解决所有的性能问题。你得考虑任何会导致失败的代码（比如打开文件，连接服务器，Fork子进程等）可能产生的结果。包括为什么出错，错误背后的原因。之后会提及，但是关键在于错误处理的粒度要细，因为哪里出错和为什么出错决定了影响大小和对策。

你可能会发现在栈的某几层不断地处理相同的错误。这是因为底层除了向上层传递错误，上层再向它的上层传递错误以外，底层没有做任何有意义的事情。通常，只有顶层的调用者知道正确的应对是什么，是重试操作，报告给用户还是其它。但是那并不意味着，你应该把所有的错误全都丢给顶层的回调函数。因为，顶层的回调函数不知道发生错误的上下文，不知道哪些操作已经成功执行，哪些操作实际上失败了。

我们来更具体一些。对于一个给定的错误，你可以做这些事情:

- **直接处理**。有的时候该做什么很清楚。如果你在尝试打开日志文件的时候得

到了一个 ENOENT 错误，很有可能你是第一次打开这个文件，你要做的就是首先创建它。更有意思的例子是，你维护着到服务器（比如数据库）的持久连接，然后遇到了一个“socket hang-up”的异常。这通常意味着要么远端要么本地的网络失败了。很多时候这种错误是暂时的，所以大部分情况下你得重新连接来解决问题。（这和接下来的重试不大一样，因为在你得到这个错误的时候不一定有操作正在进行）

- **把出错扩散到客户端。**如果你不知道怎么处理这个异常，最简单的方式就是放弃你正在执行的操作，清理所有开始的，然后把错误传递给客户端。（怎么传递异常是另外一回事了，接下来会讨论）。这种方式适合错误短时间内无法解决的情形。比如，用户提交了不正确的JSON，你再解析一次是没什么帮助的。

- **重试操作。**对于那些来自网络和远程服务的错误，有的时候重试操作就可以解决问题。比如，远程服务返回了503（服务不可用错误），你可能会在几秒种后重试。**如果确定要重试，你应该清晰的用文档记录下将会多次重试，重试多少次直到失败,以及两次重试的间隔。** 另外，不要每次都假设需要重试。如果在栈中很深的地方（比如，被一个客户端调用，而那个客户端被另外一个由用户操作的客户端控制），这种情形下快速失败让客户端去重试会更好。如果栈中的每一层都觉得需要重试，用户最终会等待更长的时间，因为每一层都没有意识到下层同时也在尝试。

- **直接崩溃。**对于那些本不可能发生的错误，或者由程序员失误导致的错误（比如无法连接到同一程序里的本地套接字），可以记录一个错误日志然后直接崩溃。其它的比如内存不足这种错误，是JavaScript这样的脚本语言无法处理的，崩溃是十分合理的。（即便如此，在 child_process.exec 这样的分离的操作里，得到 ENOMEM 错误，或者那些你可以合理处理的错误时，你应该考虑这么做）。在你无计可施需要让管理员做修复的时候，你也可以直接崩溃。如果你用光了所有的文件描述符或者没有访问配置文件的权限，这种情况下你���么都做不了，只能等某个用户登录系统把东西修好。

- **记录错误，其他什么都不做。**有的时候你什么都做不了，没有操作可以重试

或者放弃，没有任何理由崩溃掉应用程序。举个例子吧，你用DNS跟踪了一组远程服务，结果有一个DNS失败了。除了记录一条日志并且继续使用剩下的服务以外，你什么都做不了。但是，你至少得记录点什么（凡事都有例外。如果这种情况每秒发生几千次，而你又没法处理，那每次发生都记录可能就不值得了，但是要周期性的记录）。

（没有办法）处理程序员的失误

对于程序员的失误没有什么好做的。从定义上看，一段本该工作的代码坏掉了（比如变量名敲错），你不能用更多的代码再去修复它。一旦你这样做了，你就使用错误处理的代码代替了出错的代码。

有些人赞成从程序员的失误中恢复，也就是让当前的操作失败，但是继续处理请求。这种做法不推荐。考虑这样的情况：原始代码里有一个失误是没考虑到某种特殊情况。你怎么确定这个问题不会影响其他请求呢？如果其它的请求共享了某个状态（服务器，套接字，数据库连接池等），有极大的可能其他请求会不正常。

典型的例子是REST服务器（比如用Restify搭的），如果有一个请求处理函数抛出了一个`ReferenceError`（比如，变量名打错）。继续运行下去很有肯能会导致严重的Bug，而且极其难发现。例如：

1. 一些请求间共享的状态可能会被变成`null`，`undefined`或者其它无效值，结果就是下一个请求也失败了。
2. 数据库（或其它）连接可能会被泄露，降低了能够并行处理的请求数量。最后只剩下几个可用连接会很坏，将导致请求由并行变成串行被处理。
3. 更糟的是， postgres 连接会被留在打开的请求事务里。这会导致 postgres “持有”表中某一行的旧值，因为它对这个事务可见。这个问题会存在好几周，造成表无限制的增长，后续的请求全都被拖慢了，从几毫秒到几分钟[脚注4]。虽然这个问题和 postgres 紧密相关，但是它很好的说明了程序员一个简单的失误会让应用程序陷入一种非常可怕的状态。
4. 连接会停留在已认证的状态，并且被后续的连接使用。结果就是在请求里搞

错了用户。

5. 套接字会一直打开着。一般情况下 NodeJS 会在一个空闲的套接字上应用两分钟的超时，但这个值可以覆盖，这将会泄露一个文件描述符。如果这种情况不断发生，程序会因为用光了所有的文件描述符而强退。即使不覆盖这个超时时间，客户端会挂两分钟直到 “hang-up” 错误的发生。这两分钟的延迟会让问题难于处理和调试。
6. 很多内存引用会被遗留。这会导致泄露，进而导致内存耗尽，GC需要的时间增加，最后性能急剧下降。这点非常难调试，而且很需要技巧与导致造成泄露的失误联系起来。

最好的从失误恢复的方法是立刻崩溃。你应该用一个restarter 来启动你的程序，在奔溃的时候自动重启。如果restarter 准备就绪，崩溃是失误来临时最快的恢复可靠服务的方法。

奔溃应用程序唯一的负面影响是相连的客户端临时被扰乱，但是记住：

- 从定义上看，这些错误属于Bug。我们并不是在讨论正常的系统或是网络错误，而是程序里实际存在的Bug。它们应该在线上很罕见，并且是调试和修复的最高优先级。
- 上面讨论的种种情形里，请求没有必要一定得成功完成。请求可能成功完成，可能让服务器再次崩溃，可能以某种明显的方式不正确的完成，或者以一种很难调试的方式错误的结束了。
- 在一个完备的分布式系统里，客户端必须能够通过重连和重试来处理服务端的错误。不管 NodeJS 应用程序是否被允许崩溃，网络和系统的失败已经是一个事实了。
- 如果你的线上代码如此频繁地崩溃让连接断开变成了问题，那么正真的问题是你的服务器Bug太多了，而不是因为你选择出错就崩溃。

如果出现服务器经常崩溃导致客户端频繁掉线的问题，你应该把经历集中在造成服务器崩溃的Bug上，把它们变成可捕获的异常，而不是在代码明显有问题的情况下尽可能地避免崩溃。调试这类问题最好的方法是，把 NodeJS 配置成出现未捕获异常时把内核文件打印出来。在 GNU/Linux 或者 基于 illumos 的系统

上使用这些内核文件，你不仅查看应用崩溃时的堆栈记录，还可以看到传递给函数的参数和其它的 JavaScript 对象，甚至是那些在闭包里引用的变量。即使没有配置 code dumps，你也可以用堆栈信息和日志来开始处理问题。

最后，记住程序员在服务器端的失误会造成客户端的操作失败，还有客户端必须处理好服务器端的奔溃和网络中断。这不只是理论，而是实际发生在线上环境里。

编写函数的实践

我们已经讨论了如何处理异常，那么当你在编写新的函数的时候，怎么才能向调用者传递错误呢？

最最重要的一点是为你的函数写好文档，包括它接受的参数（附上类型和其它约束），返回值，可能发生的错误，以及这些错误意味着什么。 **如果你不知道会导致什么错误或者不了解错误的含义，那你的应用程序正常工作就是一个巧合。** 所以，当你编写新的函数的时候，一定要告诉调用者可能发生哪些错误和错误的含义。

Throw， Callback 还是 EventEmitter

函数有三种基本的传递错误的模式。

- `throw` 以同步的方式传递异常--也就是在函数被调用处的相同的上下文。如果调用者（或者调用者的调用者）用了 `try/catch` ，则异常可以捕获。如果所有的调用者都没有用，那么程序通常情况下会崩溃（异常也可能会被 `domains` 或者进程级的 `uncaughtException` 捕捉到，详见下文）。
- Callback 是最基础的异步传递事件的一种方式。用户传进来一个函数（callback），之后当某个异步操作完成后调用这个 callback。通常 callback 会以 `callback(err,result)` 的形式被调用，这种情况下， err和

result必然有一个是非空的，取决于操作是成功还是失败。

- 更复杂的情形是，函数没有用 Callback 而是返回一个 EventEmitter 对象，调用者需要监听这个对象的 error事件。这种方式在两种情况下很有用。

- 当你在做一个可能会产生多个错误或多个结果的复杂操作的时候。比如，有一个请求一边从数据库取数据一边把数据发送回客户端，而不是等待所有的结果一起到达。在这个例子里，没有用 callback，而是返回了一个 EventEmitter，每个结果会触发一个 `row` 事件，当所有结果发送完毕后会触发 `end` 事件，出现错误时会触发一个 `error` 事件。

- 用在那些具有复杂状态机的对象上，这些对象往往伴随着大量的异步事件。例如，一个套接字是一个EventEmitter，它可能会触发“connect“，” end “，” timeout “，” drain “，” close “事件。这样，很自然地可以把” error “作为另外一种可以被触发的事件。在这种情况下，清楚知道” error “还有其它事件何时被触发很重要，同时被触发的还有什么事件（例如” close “），触发的顺序，还有套接字是否在结束的时候处于关闭状态。

在大多数情况下，我们会把 callback 和 event emitter 归到同一个“异步错误传递”篮子里。如果你有传递异步错误的需要，你通常只要用其中的一种而不是同时使用。

那么，什么时候用 `throw` ，什么时候用callback，什么时候又用 EventEmitter 呢？这取决于两件事：

- 这是操作失败还是程序员的失误？
- 这个函数本身是同步的还是异步的。

直到目前，最常见的例子是在异步函数里发生了操作失败。在大多数情况下，你需要写一个以回调函数作为参数的函数，然后你会把异常传递给这个回调函数。这种方式工作的很好，并且被广泛使用。例子可参照 NodeJS 的 `fs` 模块。如果你的场景比上面这个还复杂，那么你可能就得换用 EventEmitter 了，不过你也

还是在用异步方式传递这个错误。

其次常见的一个例子是像 `JSON.parse` 这样的函数同步产生了一个异常。对这些函数而言，如果遇到操作失败（比如无效输入），你得用同步的方式传递它。你可以抛出（更加常见）或者返回它。

对于给定的函数，如果有一个异步传递的异常，那么所有的异常都应该被异步传递。可能有这样的情况，请求一到来你就知道它会失败，并且知道不是因为程序员的失误。可能的情形是你缓存了返回给最近请求的错误。虽然你知道请求一定失败，但是你还是应该用异步的方式传递它。

通用的准则就是 **你即可以同步传递错误（抛出），也可以异步传递错误（通过传给一个回调函数或者触发EventEmitter的 `error` 事件），但是不用同时使用。**以这种方式，用户处理异常的时候可以选择用回调函数还是用 `try/catch` ，但是不需要两种都用。具体用哪一个取决于异常是怎么传递的，这点得在文档里说明清楚。

差点忘了程序员的失误。回忆一下，它们其实是Bug。在函数开头通过检查参数的类型（或是其它约束）就可以被立即发现。一个退化的例子是，某人调用了一个异步的函数，但是没有传回调函数。你应该立刻把这个错抛出，因为程序已经出错而在这个点上最好的调试的机会就是得到一个堆栈信息，如果有内核信息就更好了。

因为程序员的失误永远不应该被处理，上面提到的调用者只能用 `try/catch` 或者回调函数（或者 EventEmitter）其中一种处理异常的准则并没有因为这条意见而改变。如果你想知道更多，请见上面的 （不要）处理程序员的失误。

下表以 NodeJS 核心模块的常见函数为例，做了一个总结，大致按照每种问题出现的频率来排列：

函数	类型	错误	错误类型	传递方式	调用者
	异		操作失		

函数类型	错误类型	操作失败	传递方式	调用者处理
fs.stat	异步	file not found	失误	callback	handle
JSON.parse	同步	bad user input	操作失败	throw	try/catch
fs.stat	异步	null for filename	失误	throw	none (crash)

异步函数里出现操作错误的例子（第一行）是最常见的。在同步函数里发生操作失败（第二行）比较少见，除非是验证用户输入。程序员失误（第三行）除非是在开发环境下，否则永远都不应该出现。

吐槽：程序员失误还是操作失败？

你怎么知道是程序员的失误还是操作失败呢？很简单，你自己来定义并且记在文档里，包括允许什么类型的函数，怎样打断它的执行。如果你得到的异常不是文档里能接受的，那就是一个程序员失误。如果在文档里写明接受但是暂时处理不了的，那就是一个操作失败。

你得用你的判断力去决定你想做到多严格，但是我们会给你一定的意见。具体一些，想象有个函数叫做“connect”，它接受一个IP地址和一个回调函数作为参数，这个回调函数会在成功或者失败的时候被调用。现在假设用户传进来一个明显不是IP地址的参数，比如`“bob”`，这个时候你有几种选择：

- 在文档里写清楚只接受有效的IPV4的地址，当用户传进来`“bob”`的时候抛出一个异常。强烈推荐这种做法。
- 在文档里写上接受任何string类型的参数。如果用户传的是`“bob”`，触发一个异步错误指明无法连接到`“bob”`这个IP地址。

这两种方式和我们上面提到的关于操作失败和程序员失误的指导原则是一致的。你决定了这样的输入算是程序员的失误还是操作失败。通常，用户输入的校验是很松的，为了证明这点，可以看`Date.parse`这个例子，它接受很多类型的输入。但是对于大多数其它函数，我们强烈建议你偏向更严格而不是更松。你的程序越是猜测用户的本意（使用隐式的转换，无论是JavaScript语言本身这么做还

是有意为之），就越是容易猜错。本意是想让开发者在使用的时候不用更加具体，结果却耗费了人家好几个小时在Debug上。再说了，如果你觉得这是个好主意，你也可以在未来的版本里让函数不那么严格，但是如果你发现由于猜测用户的意图导致了很多恼人的bug，要修复它的时候想保持兼容性就不大可能了。

所以如果一个值怎么都不可能是有效的（本该是string却得到一个`undefined`，本该是string类型的IP但明显不是），你应该在文档里写明是这不允许的并且立刻抛出一个异常。只要你在文档里写的清清楚楚，那这就是一个程序员的失误而不是操作失败。立即抛出可以把Bug带来的损失降到最小，并且保存了开发者可以用来调试这个问题的信息（例如，调用堆栈，如果用内核文件还可以得到参数和内存分布）。

那么`domains`和`process.on('uncaughtException')`呢？

操作失败总是可以被显示的机制所处理的：捕获一个异常，在回调里处理错误，或者处理EventEmitter的“error”事件等等。`Domains`以及进程级别的`‘uncaughtException’`主要是用来从未料到的程序错误恢复的。由于上面我们所讨论的原因，这两种方式都不鼓励。

编写新函数的具体建议

我们已经谈论了很多指导原则，现在让我们具体一些。

1. 你的函数做什么得很清楚。

这点非常重要。每个接口函数的文档都要很清晰的说明： - 预期参数 - 参数的类型 - 参数的额外约束（例如，必须是有效的IP地址）

如果其中有一点不正确或者缺少，那就是一个程序员的失误，你应该立刻抛出来。

此外，你还要记录：

- 调用者可能会遇到的操作失败（以及它们的`name`）
- 怎么处理操作失败（例如是抛出，传给回调函数，还是被 EventEmitter 发出）
- 返回值

1. 使用 Error 对象或它的子类，并且实现 Error 的协议。

你的所有错误要么使用 Error 类要么使用它的子类。你应该提供`name`和`message`属性，`stack`也是（注意准确）。

1. 在程序里通过 Error 的`name`属性区分不同的错误。

当你想要知道错误是何种类型的时候，用name属性。 JavaScript内置的供你重用的名字包括“RangeError”（参数超出有效范围）和“TypeError”（参数类型错误）。而HTTP异常，通常会用RFC指定的名字，比如“BadRequestError”或者“ServiceUnavailableError”。

不要想着给每个东西都取一个新的名字。如果你可以只用一个简单的InvalidArgumentError，就不要分成InvalidHostnameError，InvalidIpAddressError，InvalidDnsError等等，你要做的是通过增加属性来说明那里出了问题（下面会讲到）。

1. 用详细的属性来增强 Error 对象。

举个例子，如果遇到无效参数，把`propertyName`设成参数的名字，把`propertyValue`设成传进来的值。如果无法连到服务器，用`remoteIp`属性指明尝试连接到的 IP。如果发生一个系统错误，在`syscal`属性里设置是哪个系统调用，并把错误代码放到`errno`属性里。具体你可以查看附录，看有哪些样例属性可以用。

至少需要这些属性：

`name`：用于在程序里区分众多的错误类型（例如参数非法和连接失败）

`message`：一个供人类阅读的错误消息。对可能读到这条消息的人来说这应该已经足够完整。如果你从更底层的地方传递了一个错误，你应该加上一些信息来说明你在做什么。怎么包装异常请往下看。

`stack`：一般来讲不要随意扰乱堆栈信息。甚至不要增强它。V8引擎只有在这个属性被读取的时候才会真的去运算，以此大幅提高处理异常时候的性能。如果你读完再去增强它，结果就会多付出代价，哪怕调用者并不需要堆栈信息。

你还应该在错误信息里提供足够的消息，这样调用者不用分析你的错误就可以新建自己的错误。它们可能会本地化这个错误信息，也可能想要把大量的错误聚集到一起，再或者用不同的方式显示错误信息（比如在网页上的一个表格里，或者高亮显示用户错误输入的字段）。

1. 若果你传递一个底层的错误给调用者，考虑先包装一下。

经常会发现一个异步函数 `funcA` 调用另外一个异步函数 `funcB`，如果 `funcB` 抛出了一个错误，希望 `funcA` 也抛出一模一样的错误。（请注意，第二部分并不总是跟在第一部分之后。有的时候 `funcA` 会重新尝试。有的时候又希望 `funcA` 忽略错误因为无事可做。但在这里，我们只讨论 `funcA` 直接返回 `funcB` 错误的情况）

在这个例子里，可以考虑包装这个错误而不是直接返回它。包装的意思是继续抛出一个包含底层信息的新的异常，并且带上当前层的上下文。用 `verror` 这个包可以很简单的做到这点。

举个例子，假设有一个函数叫做 `fetchConfig`，这个函数会到一个远程的数据库取得服务器的配置。你可能会在服务器启动的时候调用这个函数。整个流程看起来是这样的：

1.加载配置 1.1 连接数据库 1.1.1 解析数据库服务器的DNS主机名 1.1.2 建立一个到数据库服务器的TCP连接 1.1.3 向数据库服务器认证 1.2 发送DB请求 1.3 解析返回结果 1.4 加载配置 2 开始处理请求

假设在运行时出了一个问题连接不到数据库服务器。如果连接在 1.1.2 的时候因为没有到主机的路由而失败了，每个层都不加处理地都把异常向上抛出给调用者。你可能会看到这样的异常信息：

> myserver: Error: connect ECONNREFUSED

这显然没什么大用。

另一方面，如果每一层都把下一层返回的异常包装一下，你可以得到更多的信息：

> myserver: failed to start up: failed to load configuration: failed to connect to database server: failed to connect to 127.0.0.1 port 1234: connect ECONNREFUSED。

你可能会想跳过其中几层的封装来得到一条不那么充满学究气息的消息：

> myserver: failed to load configuration: connection refused from database at 127.0.0.1 port 1234.

不过话又说回来，报错的时候详细一点总比信息不够要好。

如果你决定封装一个异常了，有几件事情要考虑：

- 保持原有的异常完整不变，保证当调用者想要直接用的时候底层的异常还可用。
- 要么用原有的名字，要么显示地选择一个更有意义的名字。例如，最底层是NodeJS 报的一个简单的Error，但在步骤1中可以是个 IntializationError 。（但是如果程序可以通过其它的属性区分，不要觉得有责任取一个新的名

字）

- 保留原错误的所有属性。在合适的情况下增强`message`属性（但是不要在原始的异常上修改）。浅拷贝其它的像是`syscall`，`errno`这类的属性。最好是直接拷贝除了`name`，`message`和`stack`以外的所有属性，而不是硬编码等待拷贝的属性列表。不要理会`stack`，因为即使是读取它也是相对昂贵的。如果调用者想要一个合并后的堆栈，它应该遍历错误原因并打印每一个错误的堆栈。

在Joyent，我们使用`verror`这个模块来封装错误，因为它的语法简洁。写这篇文章的时候，它还不能支持上面的所有功能，但是会被扩���以期支持。

例子

考虑有这样的一个函数，这个函数会异步地连接到一个IPv4地址的TCP端口。我们通过例子来看文档怎么写：

```
/*
* Make a TCP connection to the given IPv4 address.  Arguments:
*
*    ip4addr       a string representing a valid IPv4 address
*
*    tcpPort       a positive integer representing a valid TCP port
*
*    timeout       a positive integer denoting the number of millisecon
ds
*                  to wait for a response from the remote server before
*                  considering the connection to have failed.
*
*    callback      invoked when the connection succeeds or fails.  Upon
*                  success, callback is invoked as callback(null, socket),
*                  where `socket` is a Node net.Socket object.  Upon failure
```

```
,
*                callback is invoked as callback(err) instead.
*
* This function may fail for several reasons:
*
*    SystemError    For "connection refused" and "host unreachable" a
nd other
*                errors returned by the connect(2) system call.  For these
*                errors, err.errno will be set to the actual errno symbolic
*                name.
*
*    TimeoutError   Emitted if "timeout" milliseconds elapse without
*                successfully completing the connection.
*
* All errors will have the conventional "remoteIp" and "remotePort" p
roperties.
* After any error, any socket that was created will be closed.
*/
function connect(ip4addr, tcpPort, timeout, callback)
{
assert.equal(typeof (ip4addr), 'string',
    "argument 'ip4addr' must be a string");
assert.ok(net.isIPv4(ip4addr),
    "argument 'ip4addr' must be a valid IPv4 address");
assert.equal(typeof (tcpPort), 'number',
    "argument 'tcpPort' must be a number");
assert.ok(!isNaN(tcpPort) && tcpPort > 0 && tcpPort < 65536,
    "argument 'tcpPort' must be a positive integer between 1 and 655
35");
assert.equal(typeof (timeout), 'number',
    "argument 'timeout' must be a number");
assert.ok(!isNaN(timeout) && timeout > 0,
    "argument 'timeout' must be a positive integer");
assert.equal(typeof (callback), 'function');

/* do work */
```

```
  // do work
}
```

这个例子在概念上很简单，但是展示了上面我们所谈论的一些建议：

- 参数，类型以及其它一些约束被清晰的文档化。
- 这个函数对于接受的参数是非常严格的，并且会在得到错误参数的时候抛出异常（程序员的失误）。
- 可能出现的操作失败集合被记录了。通过不同的”name“值可以区分不同的异常，而”errno“被用来获得系统错误的详细信息。
- 异常被传递的方式也被记录了（通过失败时调用回调函数）。
- 返回的错误有”remoteIp“和”remotePort“字段，这样用户就可以定义自己的错误了（比如，一个HTTP客户端的端口号是隐含的）。
- 虽然很明显，但是连接失败后的状态也被清晰的记录了：所有被打开的套接字此时已经被关闭。

这看起来像是给一个很容易理解的函数写了超过大部分人会写的的超长注释，但大部分函数实际上没有这么容易理解。所有建议都应该被有选择的吸收，如果事情很简单，你应该自己做出判断，但是记住：用十分钟把预计发生的记录下来可能之后会为你或其他人节省数个小时。

总结

- 学习了怎么区分操作失败，即那些可以被预测的哪怕在正确的程序里也无法避免的错误（例如，无法连接到服务器）；而程序的Bug则是程序员失误。
- 操作失败可以被处理,也应当被处理。程序员的失误无法被处理或可靠地恢复

（本不应该这么做），尝试这么做只会让问题更难调试。

- 一个给定的函数，它处理异常的方式要么是同步（用throw方式）要么是异步的（用callback或者EventEmitter），不会两者兼具。用户可以在回调函数里处理错误，也可以使用 `try/catch` 捕获异常 ，但是不能一起用。实际上，使用throw并且期望调用者使用 `try/catch` 是很罕见的，因为 NodeJS 里的同步函数通常不会产生运行失败（主要的例外是类似于 `JSON.parse` 的用户输入验证函数）。

- 在写新函数的时候，用文档清楚地记录函数预期的参数，包括它们的类型、是否有其它约束（例如必须是有效的IP地址），可能会发生的合理的操作失败（例如无法解析主机名，连接服务器失败，所有的服务器端错误），错误是怎么传递给调用者的（同步，用 `throw` ，还是异步，用 callback 和 EventEmitter）。

- 缺少参数或者参数无效是程序员的失误，一旦发生总是应该抛出异常。函数的作者认为的可接受的参数可能会有一个灰色地带，但是如果传递的是一个文档里写明接收的参数以外的东西，那就是一个程序员失误。

- 传递错误的时候用标准的 Error 类和它标准的属性。尽可能把额外的有用信息放在对应的属性里。如果有可能，用约定的属性名（如下）。

附录：Error 对象属性命名约定

强烈建议你在发生错误的时候用这些名字来保持和Node核心以及Node插件的一致。这些大部分不会和某个给定的异常对应，但是出现疑问的时候，你应该包含任何看起来有用的信息，即从编程上也从自定义的错误消息上。【表】。

Property name	Intended use
localHostname	the local DNS hostname (e.g., that you're

Property name	Intended use
	accepting connections at)
localIp	the local IP address (e.g., that you're accepting connections at)
localPort	the local TCP port (e.g., that you're accepting connections at)
remoteHostname	the DNS hostname of some other service (e.g., that you tried to connect to)
remoteIp	the IP address of some other service (e.g., that you tried to connect to)
remotePort	the port of some other service (e.g., that you tried to connect to)
path	the name of a file, directory, or Unix Domain Socket (e.g., that you tried to open)
srcpath	the name of a path used as a source (e.g., for a rename or copy)
dstpath	the name of a path used as a destination (e.g., for a rename or copy)
hostname	a DNS hostname (e.g., that you tried to resolve)
ip	an IP address (e.g., that you tried to reverse-resolve)
propertyName	an object property name, or an argument name (e.g., for a validation error)
propertyValue	an object property value (e.g., for a validation error)
syscall	the name of a system call that failed
	the symbolic value of errno (e.g., "ENOENT"). Do

Property name	Intended use
errno	not use this for errors that don't actually set the C value of errno.Use "name" to distinguish between types of errors.

脚注

1. 人们有的时候会这么写代码，他们想要在出现异步错误的时候调用 callback 并把错误作为参数传递。他们错误地认为在自己的回调函数（传递给 `doSomeAsynchronousOperation` 的函数）里 `throw` 一个异常，会被外面的catch代码块捕获。`try/catch` 和异步函数不是这么工作的。回忆一下，异步函数的意义就在于被调用的时候 `myApiFunc` 函数已经返回了。这意味着try代码块已经退出了。这个回调函数是由Node直接调用的，外面并没有try的代码块。如果你用这个反模式，结果就是抛出异常的时候，程序崩溃了。

2. 在JavaScript里，抛出一个不属于Error的参数从技术上是可行的，但是应该被避免。这样的结果使获得调用堆栈没有可能，代码也无法检查”name“属性，或者其它任何能够说明哪里有问题的属性。

3. 操作失败和程序员的失误这一概念早在NodeJS之前就已经存在存在了。不严格地对应者Java里的checked和unchecked异常，虽然操作失败被认为是无法避免的，比如 OutOfMemeoryError，被归为uncheked异常。在C语言里有对应的概念，普通异常处理和使用断言。维基百科上关于断言的的文章也有关于什么时候用断言什么时候用普通的错误处理的类似的解释。

4. 如果这看起来非常具体，那是因为我们在产品环境中遇到这样过这样的问题。这真的很可怕。

本文由OneAPM工程师编译并整理 ，想阅读更多技术文章，请访问OneAPM官方技术博客。