链滴

Node 出现 uncaughtException 之后的优雅退出方案

作者：Vanessa

原文链接：https://ld246.com/article/1393367667890

来源网站：链滴

转自：http://www.infoq.com/cn/articles/quit-scheme-of-node-uncaughtexception-emerence

Node 的异步特性是它最大的魅力，但是在带来便利的同时也带来了不少麻烦和坑，错误捕获就一个。由于 Node 的异步特性，导致我们无法使用 try/catch 来捕获回调函数中的异常，例如:
<pre class="brush: js">try {
    console.log('进入 try/catch');
    require('fs').stat('SOME_FILE_DOES_NOT_EXIST',
function readCallback(err, content) {
        if (err) {
            throw err; // 抛出异常
        }
    });
} catch (e) {
    // 这里捕获不到 readCallback 函数中抛出的异常
} finally {
    console.log('离开 try/catch');
}</pre>
运行结果是:
<pre class="brush: js">进入 try/catch
离开 try/catch

test.js:7
throw err; // 抛出异常
^
Error: ENOENT, stat 'SOME_FILE_DOES_NOT_EXIST'</pre>

上面代码中由于 <code>fs.stat</code> 去查询一个不存在的文件的状态，导致 <code>readCallback</code> 抛出了一个异常。由于 <code>fs.read</code> 的异步特性，<code>readCallbac</code> 函数的调用发生在 <code>try/catch</code> 块结束之后，所以该异常不会被 try/catch 获。之后 Node 会触发 <code>uncaughtException</code> 事件，如果这个事件依然没有得到响，整个进程(<code>process</code>)就会 crash。
程序员永远无法保证代码中不出现 <code>uncaughtException</code>，即便是自己代码写足够小心，也不能保证用的第三方模块没有 bug，例如:
<pre class="brush: js">var deserialize = require('deserialize');
// 假设 deserialize 是一个带有 bug 的第三方模块

// app 是一个 express 服务对象
app.get('/users', function (req, res) {
mysql.query('SELECT * FROM user WHERE id=1', function (err, user) {
var config = deserialize(user.config);
// 假如这里触发了 deserialize 的 bug
res.send(config);
});
});</pre>

如果不幸触发了 <code>deserialize</code> 模块的 bug，这里就会抛出一个异常，最终结果整个服务 crash。

当这种情况发生在 Web 服务上时结果是灾难性的。<code>uncaughtException</code> 错会导致当前的所有的用户连接都被中断，甚至不能返回一个正常的 HTTP 错误码，用户只能等到浏览超时才能看到一个 <code>no data received</code> 错误。
这是一种非常野蛮粗暴的异常处理机制，任何线上服务都不应该因为 <code>uncaughtExceptin</code> 导致服务器崩溃。一个友好的错误处理机制应该满足三个条件:

对于引发异常的用户，返回 500 页面
其他用户不受影响，可以正常访问
不影响整个进程的正常运行

很遗憾的是，保证 <code>uncaughtException</code> 不影响整个进程的健康运转是不可能。当 Node 抛出 <code>uncaughtException</code> 异常时就会丢失当前环境的堆栈，导致 Node 不能正常进行内存回收。也就是说，每一次 <code>uncaughtException</code> 都有可能导致内泄露。
既然如此，退而求其次，我们可以在满足前两个条件的情况下退出进程以便重启服务。
<h2>用 domain 来捕获异步异常</h2>
普遍的思路是，如果可以通过某种方式来捕获回调函数中的异常，那么就不会有 <code>uncauhtException</code> 错误导致的崩溃。为了解决这个问题，Node 0.8 之后的版本新增了 <code>domain</code> 模块，它可以用来捕获回调函数中抛出的异常。
<code>domain</code> 主要的 API 有 <code>domain.run</code> 和 <code>error</code> 事件。简单的说，通过 <code>domain.run</code> 执行的函数中引发的异常都可以通过 <code>domain</code> 的 <code>error</code> 事件捕获，例如:

```
var domain = require('domain');
var d = domain.create();
d.run(function () {
    setTimeout(function () {
        throw new Error('async error'); // 抛出一个异步异常
    }, 1000);
});

d.on('error', function (err) {
console.log('catch err:', err); // 这里可以捕获异步异常
});
```

通过 <code>domain</code> 模块，以及 JavaScript 的词法作用域特性，可以很轻易的为引异常的用户返回 500 页面。以 express 为例:

```
var app = express();
var server = require('http').createServer(app);
var domain = require('domain');

app.use(function (req, res, next) {
var reqDomain = domain.create();
reqDomain.on('error', function (err) { // 下面抛出的异常在这里被捕获
res.send(500, err.stack); // 成功给用户返回了 500
});


reqDomain.run(next);

});
```

app.get('/', function () {
setTimeout(function () {
throw new Error('async exception'); // 抛出一个异步异常
}, 1000);
});</pre>

上面的代码将 domain 作为一个中间件来使用，保证之后 express 所有的中间件都在 <code>domain.run</code> 函数内部执行。这些中间件内的异常都可以通过 <code>error</code> 事件来获。
尽管借助于闭包，我们可以正常的给用户返回 500 错误，但是 <code>domain</code> 捕获错误时依然会丢失堆栈信息，此时已经无法保证程序的健康运行，必须退出。Node http server 提了 <code>close</code> 方法，该方法在调用时会停止 server 接收新的请求，但不会断开当前已建立的连接。
<pre class="brush: js">reqDomain.on('error', function () {
    try {
        // 强制退出机制
        var killTimer = setTimeout(function () {
            process.exit(1);
        }, 30000);
        killTimer.unref(); // 非常重要

    // 自动退出机制，停止接收新链接，等待当前已建立连接的关闭
    server.close(function () {
   // 此时所有连接均已关闭，此时 Node 会自动退出，不需要再调用

process.exit(1) 来结束进程

});

} catch(e) {

console.log('err', e.stack);

}

});</pre>

这个例子来自 Node 的文档。其中有几个关键点：

 Node 有个非常好的特性，所有连接都被释放后进程会自动结束，所以不需要再 <code>server.close</code> 方法的回调函数中退出进程
 强制退出机制: 因为用户连接有可能因为某些原因无法释放，在这种情况下应该强制退出整个进。
 <code>killTimer.unref()</code>: 如果不使用 <code>unref</code> 方法，那么即使 server 的所有连接都关闭，Node 也会保持运行直到 <code>killTimer</code> 的回调函数被调用。<code>unref</code> 可以创建一个"不保持程序运行"的计时器。
 处理异常时要小心的把异常处理逻辑用 try/catch 包住，避免处理异常时抛出新的异常

通过 <code>domain</code> 似乎就已经解决了我们的需求: 给触发异常的用户一个 500，停接收新请求，提供正常的服务给已经建立连接的用户，直到所有请求都已结束，退出进程。但是，理很丰满，现实很骨感，<code>domain</code> 有个最大的问题，它不能捕获所有的异步异常！。也就是说，即使用了 <code>domain</code>，程序依然有因为 <code>uncaughtException</code> crash 的可能。
所幸的是我们可以监听 <code>uncaughtException</code> 事件。

<h2><code>uncaughtException</code> 事件</h2>
<code>uncaughtException</code> 是一个非常古老的事件。当 Node 发现一个未捕获的异时，会触发这个事件。并且如果这个事件存在回调函数，Node 就不会强制结束进程。这个特性，可用来弥补 <code>domain</code> 的不足:
<pre class="brush: js">process.on('uncaughtException', function (err) {
    console.log(err);

try {
    var killTimer = setTimeout(function () {
        process.exit(1);
    }, 30000);
    killTimer.unref();

    server.close();
} catch (e) {
    console.log('error when exit', e.stack);
}

});</pre>

<code>uncaughtException</code> 事件的缺点在于无法为抛出异常的用户请求返回一个 500 错误，这是由于 <code>uncaughtException</code> 丢失了当前环境的上下文，比如下面的例子是它做不到的:
<pre class="brush: js">javascript
app.get('/', function (req, res) {
    setTimeout(function () {
        throw new Error('async error');
// uncaughtException, 导致 req 的引用丢失
        res.send(200);
    }, 1000);
});

process.on('uncaughtException', function (err) {

res.send(500); // 做不到，拿不到当前请求的 res 对象

});</pre>

最终出错的用户只能等待浏览器超时。
<h2><code>domain</code> + <code>uncaughtException</code></h2>
所以，我们可以结合两种异常捕获机制，用 <code>domain</code> 来捕获大部分的异常，且提供友好的 500 页面以及优雅退出。对于剩下的异常，通过 <code>uncaughtException</code> 事件来避免服务器直接 crash。
代码如下:
<pre class="brush: js">var app = express();
var server = require('http').create(app);
var domain = require('domain');

// 使用 domain 来捕获大部分异常

app.use(function (req, res, next) {

var reqDomain = domain.create();

reqDomain.on('error', function () {

```
try {
var killTimer = setTimeout(function () {
process.exit(1);
}, 30000);
killTimer.unref();


        server.close();

        res.send(500);
    } catch (e) {
        console.log('error when exit', e.stack);
    }
});

reqDomain.run(next);

});

// uncaughtException 避免程序崩溃
process.on('uncaughtException', function (err) {
console.log(err);


try {
    var killTimer = setTimeout(function () {
        process.exit(1);
    }, 30000);
    killTimer.unref();

    server.close();
} catch (e) {
    console.log('error when exit', e.stack);
}

});</pre>

<h2>其他的一些问题</h2>
<h3><code>express</code> 中异常的处理</h3>
<p>使用 <code>express</code> 时记住一定不要在 controller 的异步回调中抛出异常，例如:</
>
<pre class="brush: js">app.get('/', function (req, res, next) { // 总是接收 next 参数
    mysql.query('SELECT * FROM users', function (err, results) {
        // 不要这样做
        if (err) throw err;


    // 应该将 err 传递给 errorHandler 处理
    if (err) return next(err);
});

});
```

app.use(function (err, req, res, next) {
// 带有四个参数的 middleware 专门用来处理异常
res.render(500, err.stack);
});</pre>

<h3>和 cluster 一起使用</h3>
cluster 是 node 自带的负载均衡模块，使用 cluster 模块可以方便的建立起一套 master/slave
务。在使用 cluster 模块时，需要注意不仅需要调用 <code>server.close()</code> 来关闭连接，
时还需要调用 <code>cluster.worker.disconnect()</code> 通知 master 进程已停止服务:
<pre class="brush: js">var cluster = require('cluster');

process.on('uncaughtException', function (err) {
console.log(err);

try {
    var killTimer = setTimeout(function () {
        process.exit(1);
    }, 30000);
    killTimer.unref();

    server.close();

    if (cluster.worker) {
        cluster.worker.disconnect();
    }
} catch (e) {
    console.log('error when exit', e.stack);
}

});</pre>

<h3>不要通过 <code>uncaughtException</code> 来忽略错误</h3>
当 <code>uncaughtException</code> 事件有一个以上的 <code>listener</code> 时，会
止 Node 结束进程。因此就有一个广泛流传的做法是监听 <code>process</code> 的 <code>unc
ughtException</code> 事件来阻止进程退出，这种做法有内存泄露的风险，所以千万不要这么做:<
p>
<pre>javascript
process.on('uncaughtException', function (err) { // 不要这么做
    console.log(err);
});</pre>
<h3>pm2 对于 <code>uncaughtException</code> 的额外处理</h3>
如果你在用 pm2 0.7.1 之前的版本，那么要当心。pm2 有一个 bug，如果进程抛出了 <code>
ncaughtException</code>，无论代码中是否捕获了这个事件，进程都会被 pm2 杀死。0.7.2 之后
 pm2 解决了这个问题。
<h3>要小心 worker.disconnect()</h3>
如果你在退出进程时希望可以发消息给监控服务器，并且还使用了 cluster，那么这个时候要特
小心，比如下面的代码:
<pre class="brush: js">var udpLog = dgram.createSocket('udp4');
var cluster = require('cluster');

process.on('uncaughtException', function (err) {

```
udpLog.send('process ' + process.pid + ' down',
/* ... 一些发送 udp 消息的参数 ...*/);


server.close();
cluster.worker.disconnect();

});</pre>

<p>这份代码就不能正常的将消息发送出去。因为 <code>udpLog.send</code> 是一个异步方法真正发消息的操作发生在下一个事件循环中。而在真正的发送消息之前 <code>cluster.worker.disconnect()</code> 就已经执行了。<code>worker.disconnect()</code> 会在当前进程没有任何链接后，杀掉整个进程，这种情况有可能发生在发送 log 数据之前，导致 log 数据发不出去。</p>
<p>一个解决方法是在 <code>udpLog.send</code> 方法发送完数据后再调用 <code>worker.disconnect</code>:</p>
<pre class="brush: js">var udpLog = dgram.createSocket('udp4');
var cluster = require('cluster');

process.on('uncaughtException', function (err) {
udpLog.send('process ' + process.pid + ' down', /* ...
一些发送 udp 消息的参数 ...*/, function () {
cluster.worker.disconnect();
});


server.close();

// 保证 worker.disconnect 不会拖太久..
setTimeout(function () {
    cluster.worker.disconnect();
}, 100).unref();

});</pre>

<h2>小节</h2>
<p>说了这么多，结论是，目前为止(Node 0.10.25)，依然没有一个完美的方案来解决任意异常的优退出问题。用 <code>domain</code> 来捕获大部分异常，并且通过 <code>uncaughtException</code> 避免程序 crash 是目前来说最理想的方案。回调异常的退出问题在遇到 cluster 以后会更加杂，特别是对于连接关闭的处理要格外小心。</p>
<h2>参考文章</h2>
<ul>
 <li><a href="http://nodejs.org/api/domain.html#domain_warning_don_t_ignore_errors">Don't ignore errors</a></li>
 <li><a href="http://nodejs.org/api/domain.html">Node API: domain</a></li>
 <li><a href="http://nodejs.org/api/process.html">Node API: process</a></li>
 <li><a href="http://cnodejs.org/topic/516b64596d38277306407936">Node.js 异步异常的处与domain模块解析</a></li>
 <li><a href="http://www.alloyteam.com/2013/12/node-js-series-exception-caught/">Node.j异常捕获的一些实践</a></li>
</ul>
<hr />
<p>感谢<a href="http://www.infoq.com/cn/author/%E7%94%B0%E6%B0%B8%E5%BC%BA"
```

田永强对本文的审校。
给InfoQ中文站投稿或者参与内容翻译工作，请邮件至editors@cn.infoq.com。也欢迎大家通过新浪微博（@InfoQ）或者腾讯微博（@InfoQ）关注我们，并与我们的编辑和其他读者朋友交流。