七天学会 NodeJS #6

NodeJS 最大的卖点——事件机制和异步 **IO**，对开发者并不是透明的。开发者需要按异步方式 编写代码才用得上这个卖点，而这一点也遭到了一些 **NodeJS** 反对者的抨击。但不管怎样，异步编程确实是 **NodeJS** 最大的特点，没有掌握异步编程就不能 说是真正学会了 **NodeJS**。本章将介绍与异步编程相关的各种知识。

回调

在代码中，异步编程的直接体现就是回调。异步编程依托于回调来实现，但不能说使用了回调后程序就异步化了。我们首先可以看看以下代码。

```
function heavyCompute(n, callback) {

    var count = 0,

        i, j;



    for (i = n; i > 0; --i) {

        for (j = n; j > 0; --j) {

            count += 1;

        }

    }



    callback(count);

}



heavyCompute(10000, function (count) {

    console.log(count);

});
```

```
console.log('hello');
```

```
-- Console ----------------------------

100000000

hello
```

可以看到，以上代码中的回调函数仍然先于后续代码执行。JS 本身是单线程运行的，不可能在一段代码还未结束运行时去运行别的代码，因此也就不存在异步执行的概念。

但是，如果某个函数做的事情是创建一个别的线程或进程，并与 JS 主线程并行地做一些事情，并在事情做完后通知 JS 主线程，那情况又不一样了。我们接着看看以下代码。

```
setTimeout(function () {
    console.log('world');
}, 1000);

console.log('hello');
```

```
-- Console ----------------------------

hello

world
```

这次可以看到，回调函数后于后续代码执行了。如同上边所说，JS 本身是单线程的，无法异步执行，因此我们可以认为 `setTimeout` 这类 JS 规范之外的由运行环境提供的特殊函数做的事情是创建一个平行线程后立即返回，让 JS 主进程可以接着执行后续代码，并在收到平行进程的通知后再执行回调函数。除了 `setTimeout`、`setInterval` 这些常见的，这类函数还包括 NodeJS 提供的诸如 `fs.readFile` 之类的异步 API。

另外，我们仍然回到 JS 是单线程运行的这个事实上，这决定了 JS 在执行完一段代码之前无法执行包括回调函数在内的别的代码。也就是说，即使平行线程完成工作了，通知 JS 主线程执行回调函数了，回调函数也要等到 JS 主线程空闲时才能开始执行。以下就是这么一个例子。

```
function heavyCompute(n) {
```

```
    var count = 0,
        i, j;

    for (i = n; i > 0; --i) {
        for (j = n; j > 0; --j) {
            count += 1;
        }
    }
}

var t = new Date();

setTimeout(function () {
    console.log(new Date() - t);
}, 1000);

heavyCompute(50000);

-- Console ---------------------------
8520
```

可以看到，本来应该在**1**秒后被调用的回调函数因为 **JS** 主线程忙于运行其它代码，实际执行时间被大幅延迟。

代码设计模式

异步编程有很多特有的代码设计模式，为了实现同样的功能，使用同步方式和异步方式编写

的代码会有很大差异。以下分别介绍一些常见的模式。

函数返回值

使用一个函数的输出作为另一个函数的输入是很常见的需求，在同步方式下一般按以下方式编写代码：

```
var output = fn1(fn2('input'));

// Do something.
```

而在异步方式下，由于函数执行结果不是通过返回值，而是通过回调函数传递，因此一般按以下方式编写代码：

```
fn2('input', function (output2) {

    fn1(output2, function (output1) {

        // Do something.

    });

});
```

可以看到，这种方式就是一个回调函数套一个回调函多，套得太多了很容易写出>形状的代码。

遍历数组

在遍历数组时，使用某个函数依次对数据成员做一些处理也是常见的需求。如果函数是同步执行的，一般就会写出以下代码：

```
var len = arr.length,

    i = 0;



for (; i < len; ++i) {

    arr[i] = sync(arr[i]);

}



// All array items have processed.
```

如果函数是异步执行的，以上代码就无法保证循环结束后所有数组成员都处理完毕了。如果数组成员必须一个接一个串行处理，则一般按照以下方式编写异步代码：

```
(function next(i, len, callback) {
    if (i < len) {
        async(arr[i], function (value) {
            arr[i] = value;
            next(i + 1, len, callback);
        });
    } else {
        callback();
    }
}(0, arr.length, function () {
    // All array items have processed.
}));
```

可以看到，以上代码在异步函数执行一次并返回执行结果后才传入下一个数组成员并开始下一轮执行，直到所有数组成员处理完毕后，通过回调的方式触发后续代码的执行。

如果数组成员可以并行处理，但后续代码仍然需要所有数组成员处理完毕后才能执行的话，则异步代码会调整成以下形式：

```
(function (i, len, count, callback) {
    for (; i < len; ++i) {
        (function (i) {
            async(arr[i], function (value) {
                arr[i] = value;
                if (++count === len) {
                    callback();
```

```
                }

            });

        }(i));

    }

}(0, arr.length, 0, function () {

    // All array items have processed.

}));
```

可以看到，与异步串行遍历的版本相比，以上代码并行处理所有数组成员，并通过计数器变量来判断什么时候所有数组成员都处理完毕了。

异常处理

JS 自身提供的异常捕获和处理机制——try..catch..，只能用于同步执行的代码。以下是一个例子。

```
function sync(fn) {

    return fn();

}



try {

    sync(null);

    // Do something.

} catch (err) {

    console.log('Error: %s', err.message);

}



-- Console ---------------------------

Error: object is not a function
```

可以看到，异常会沿着代码执行路径一直冒泡，直到遇到第一个 try 语句时被捕获住。但由于异步函数会打断代码执行路径，异步函数执行过程中以及执行之后产生的异常冒泡到执行路径被打断的位置时，如果一直没有遇到 try 语句，就作为一个全局异常抛出。以下是一个例子。

```
function async(fn, callback) {
    // Code execution path breaks here.
    setTimeout(function () {
        callback(fn());
    }, 0);
}

try {
    async(null, function (data) {
        // Do something.
    });
} catch (err) {
    console.log('Error: %s', err.message);
}

-- Console ----------------------------
/home/user/test.js:4
        callback(fn());
                 ^
TypeError: object is not a function
    at null._onTimeout (/home/user/test.js:4:13)
```

```
    at Timer.listOnTimeout [as ontimeout] (timers.js:110:15)
```

因为代码执行路径被打断了，我们就需要在异常冒泡到断点之前用 `try` 语句把异常捕获住，并通过回调函数传递被捕获的异常。于是我们可以像下边这样改造上边的例子。

```
function async(fn, callback) {
    // Code execution path breaks here.
    setTimeout(function () {
        try {
            callback(null, fn());
        } catch (err) {
            callback(err);
        }
    }, 0);
}

async(null, function (err, data) {
    if (err) {
        console.log('Error: %s', err.message);
    } else {
        // Do something.
    }
});

-- Console ------------------------------

Error: object is not a function
```

可以看到，异常再次被捕获住了。在 NodeJS 中，几乎所有异步 API 都按照以上方式设计，回调函数中第一个参数都是 err。因此我们在编写自己的异步函数时，也可以按照这种方式来处理异常，与 NodeJS 的设计风格保持一致。

有了异常处理方式后，我们接着可以想一想一般我们是怎么写代码的。基本上，我们的代码都是做一些事情，然后调用一个函数，然后再做一些事情，然后再调用一个函数，如此循环。如果我们写的是同步代码，只需要在代码入口点写一个 try 语句就能捕获所有冒泡上来的异常，示例如下。

```
function main() {
    // Do something.
    syncA();
    // Do something.
    syncB();
    // Do something.
    syncC();
}


try {
    main();
} catch (err) {
    // Deal with exception.
}
```

但是，如果我们写的是异步代码，就只有呵呵了。由于每次异步函数调用都会打断代码执行路径，只能通过回调函数来传递异常，于是我们就需要在每个回调函数里判断是否有异常发生，于是只用三次异步函数调用，就会产生下边这种代码。

```
function main(callback) {
    // Do something.
    asyncA(function (err, data) {
```

```
        if (err) {
            callback(err);
        } else {
            // Do something
            asyncB(function (err, data) {
                if (err) {
                    callback(err);
                } else {
                    // Do something
                    asyncC(function (err, data) {
                        if (err) {
                            callback(err);
                        } else {
                            // Do something
                            callback(null);
                        }
                    });
                }
            });
        }
    });
}

main(function (err) {
```

```
    if (err) {

        // Deal with exception.

    }

});
```

可以看到，回调函数已经让代码变得复杂了，而异步方式下对异常的处理更加剧了代码的复杂度。如果 NodeJS 的最大卖点最后变成这个样子，那就没人愿意用 NodeJS 了，因此接下来会介绍 NodeJS 提供的一些解决方案。

域（Domain）

官方文档： http://nodejs.org/api/domain.html

NodeJS 提供了 domain 模块，可以简化异步代码的异常处理。在介绍该模块之前，我们需要首先理解"域"的概念。简单的讲，一个域就是一个 JS 运行环境，在一个运行环境中，如果一个异常没有被捕获，将作为一个全局异常被抛出。NodeJS 通过 process 对象提供了捕获全局异常的方法，示例代码如下

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

    console.log('Error: %s', err.message);

});



setTimeout(function (fn) {

    fn();

});



-- Console ------------------------

Error: undefined is not a function
```

虽然全局异常有个地方可以捕获了，但是对于大多数异常，我们希望尽早捕获，并根据结果决定代码的执行路径。我们用以下 HTTP 服务器代码作为例子：

```
function async(request, callback) {

    // Do something.
```

```
    asyncA(request, function (err, data) {
        if (err) {
            callback(err);
        } else {
            // Do something
            asyncB(request, function (err, data) {
                if (err) {
                    callback(err);
                } else {
                    // Do something
                    asyncC(request, function (err, data) {
                        if (err) {
                            callback(err);
                        } else {
                            // Do something
                            callback(null, data);
                        }
                    });
                }
            });
        }
    });
}
```

```
http.createServer(function (request, response) {
    async(request, function (err, data) {
        if (err) {
            response.writeHead(500);
            response.end();
        } else {
            response.writeHead(200);
            response.end(data);
        }
    });
});
```

以上代码将请求对象交给异步函数处理后，再根据处理结果返回响应。这里采用了使用回调函数传递异常的方案，因此 async 函数内部如果再多几个异步函数调用的话，代码就变成上边这副鬼样子了。为了让代码好看点，我们可以在每处理一个请求时，使用 domain 模块创建一个子域（JS 子运行环境）。在子域内运行的代码可以随意抛出异常，而这些异常可以通过子域对象的 error 事件统一捕获。于是以上代码可以做如下改造：

```
function async(request, callback) {
    // Do something.
    asyncA(request, function (data) {
        // Do something
        asyncB(request, function (data) {
            // Do something
            asyncC(request, function (data) {
                // Do something
                callback(data);
            });
```

```
        });

    });

}


http.createServer(function (request, response) {

    var d = domain.create();


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

        response.writeHead(500);

        response.end();

    });


    d.run(function () {

        async(request, function (data) {

            response.writeHead(200);

            response.end(data);

        });

    });

});
```

可以看到，我们使用.create方法创建了一个子域对象，并通过.run方法进入需要在子域中运行的代码的入口点。而位于子域中的异步函数回调函数由于不再需要捕获异常，代码一下子瘦身很多。

陷阱

无论是通过 process 对象的 uncaughtException 事件捕获到全局异常，还是通过子域对象的 error 事件捕获到了子域异常，在 NodeJS 官方文档里都强烈建议处理完异常后立即重启程序，而不是让程序继续运行。按照官方文档的说法，发生异常后的程序处于一个不确定的运行状态，

如果不立即退出的话，程序可能会发生严重内存泄漏，也可能表现得很奇怪。

但这里需要澄清一些事实。**JS** 本身的 `throw..try..catch` 异 常处理机制并不会导致内存泄漏，也不会让程序的执行结果出乎意料，但 **NodeJS** 并不是存粹的 **JS**。**NodeJS** 里大量的 **API** 内部是用 **C/C++**实现 的，因此 **NodeJS** 程序的运行过程中，代码执行路径穿梭于 **JS** 引擎内部和外部，而 **JS** 的异常抛出机制可能会打断正常的代码执行流程，导致 **C/C++**部分 的代码表现异常，进而导致内存泄漏等问题。

因此，使用 `uncaughtException` 或 `domain` 捕获异常，代码执行路径里涉及到了 **C/C++**部分的代码时，如果不能确定是否会导致内存泄漏等问题，最好在处理完异常后重启程序比较妥当。而使用 `try` 语句捕获异常时一般捕获到的都是 **JS** 本身的异常，不用担心上诉问题。

小结

本章介绍了 **JS** 异步编程相关的知识，总结起来有以下几点：

不掌握异步编程就不算学会 **NodeJS**。

异步编程依托于回调来实现，而使用回调不一定就是异步编程。

异步编程下的函数间数据传递、数组遍历和异常处理与同步编程有很大差别。

使用 `domain` 模块简化异步代码的异常处理，并小心陷阱。