七天学会

学习讲究的是学以致用和融会贯通。至此我们已经分别介绍了 NodeJS 的很多知识点，本章作为最后一章，将完整地介绍一个使用 NodeJS 开发 Web 服务器的示例。

需求

我们要开发的是一个简单的静态文件合并服务器，该服务器需要支持类似以下格式的 JS 或 CSS 文件合并请求。

http://assets.example.com/foo/??bar.js,baz.js

在以上 URL 中，??是一个分隔符，之前是需要合并的多个文件的 URL 的公共部分，之后是使用,分隔的差异部分。因此服务器处理这个 URL 时，返回的是以下两个文件按顺序合并后的内容。

/foo/bar.js

/foo/baz.js

另外，服务器也需要能支持类似以下格式的普通的 JS 或 CSS 文件请求。

http://assets.example.com/foo/bar.js

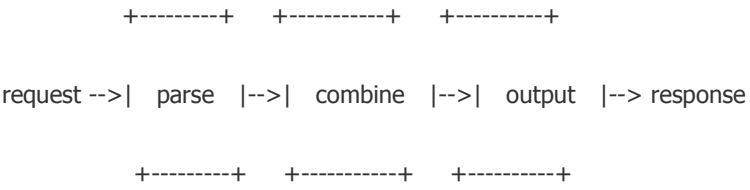
以上就是整个需求。

第一次迭代

快速迭代是一种不错的开发方式，因此我们在第一次迭代时先实现服务器的基本功能。

设计

简单分析了需求之后，我们大致会得到以下的设计方案。



也就是说，服务器会首先分析 URL，得到请求的文件的路径和类型（MIME）。然后，服务器会读取请求的文件，并按顺序合并文件内容。最后，服务器返回响应，完成对一次请求的处理。

另外，服务器在读取文件时需要有个根目录，并且服务器监听的 HTTP 端口最好也不要写死在代码里，因此服务器需要是可配置的。

实现

根据以上设计，我们写出了第一版代码如下。

```
var fs = require('fs'),
    path = require('path'),
    http = require('http');

var MIME = {
    '.css': 'text/css',
    '.js': 'application/javascript'
};

function combineFiles(pathnames, callback) {
    var output = [];

    (function next(i, len) {
        if (i < len) {
            fs.readFile(pathnames[i], function (err, data) {
                if (err) {
                    callback(err);
                } else {
                    output.push(data);
                    next(i + 1, len);
                }
            });
```

```
            } else {
                callback(null, Buffer.concat(output));
            }
        }(0, pathnames.length));
}

function main(argv) {
    var config = JSON.parse(fs.readFileSync(argv[0], 'utf-8')),
        root = config.root || '.',
        port = config.port || 80;

    http.createServer(function (request, response) {
        var urlInfo = parseURL(root, request.url);

        combineFiles(urlInfo.pathnames, function (err, data) {
            if (err) {
                response.writeHead(404);
                response.end(err.message);
            } else {
                response.writeHead(200, {
                    'Content-Type': urlInfo.mime
                });
                response.end(data);
            }
```

```
        });
    }).listen(port);
}

function parseURL(root, url) {
    var base, pathnames, parts;

    if (url.indexOf('??') === -1) {
        url = url.replace('/', '/??');
    }

    parts = url.split('??');
    base = parts[0];
    pathnames = parts[1].split(',').map(function (value) {
        return path.join(root, base, value);
    });

    return {
        mime: MIME[path.extname(pathnames[0])] || 'text/plain',
        pathnames: pathnames
    };
}

main(process.argv.slice(2));
```

以上代码完整实现了服务器所需的功能，并且有以下几点值得注意：

使用命令行参数传递 JSON 配置文件路径，入口函数负责读取配置并创建服务器。

入口函数完整描述了程序的运行逻辑，其中解析 URL 和合并文件的具体实现封装在其它两个函数里。

解析 URL 时先将普通 URL 转换为了文件合并 URL，使得两种 URL 的处理方式可以一致。

合并文件时使用异步 API 读取文件，避免服务器因等待磁盘 IO 而发生阻塞。

我们可以把以上代码保存为 server.js，之后就可以通过 node server.js config.json 命令启动程序，于是我们的第一版静态文件合并服务器就顺利完工了。

另外，以上代码存在一个不那么明显的逻辑缺陷。例如，使用以下 URL 请求服务器时会有惊喜。

```
http://assets.example.com/foo/bar.js,foo/baz.js
```

经过分析之后我们会发现问题出在/被自动替换/??这个行为上，而这个问题我们可以到第二次迭代时再解决。

第二次迭代

在第一次迭代之后，我们已经有了一个可工作的版本，满足了功能需求。接下来我们需要从性能的角度出发，看看代码还有哪些改进余地。

设计

把 map 方法换成 for 循环或许会更快一些，但第一版代码最大的性能问题存在于从读取文件到输出响应的过程当中。我们以处理/??a.js,b.js,c.js 这个请求为例，看看整个处理过程中耗时在哪儿。

```
发送请求          等待服务端响应              接收响应

---------+---------------------+------------>

          --                                  解析请求

            ------                            读取 a.js

                  ------                      读取 b.js

                        ------                读取 c.js

                              --              合并数据

                                --            输出响应
```

可以看到，第一版代码依次把请求的文件读取到内存中之后，再合并数据和输出响应。这会导致以下两个问题：

当请求的文件比较多比较大时，串行读取文件会比较耗时，从而拉长了服务端响应等待时间。

由于每次响应输出的数据都需要先完整地缓存在内存里，当服务器请求并发数较大时，会有较大的内存开销。

对于第一个问题，很容易想到把读取文件的方式从串行改为并行。但是别这样做，因为对于机械磁盘 而言，因为只有一个磁头，尝试并行读取文件只会造成磁头频繁抖动，反而降低 IO 效率。而对于固态硬盘，虽然的确存在多个并行 IO 通道，但是对于服务器并行 处理的多个请求而言，硬盘已经在做并行 IO 了，对单个请求采用并行 IO 无异于拆东墙补西墙。因此，正确的做法不是改用并行 IO，而是一边读取文件一边输出 响应，把响应输出时机提前至读取第一个文件的时刻。这样调整后，整个请求处理过程变成下边这样。

发送请求 等待服务端响应 接收响应

---------+----+------------------------------>

--　　解析请求

--　　检查文件是否存在

--　　输出响应头

------　　读取和输出 a.js

------　　读取和输出 b.js

------　　读取和输出 c.js

按上述方式解决第一个问题后，因为服务器不需要完整地缓存每个请求的输出数据了，第二个问题也迎刃而解。

实现

根据以上设计，第二版代码按以下方式调整了部分函数。

```
function main(argv) {

    var config = JSON.parse(fs.readFileSync(argv[0], 'utf-8')),

        root = config.root || '.',

        port = config.port || 80;
```

```
    http.createServer(function (request, response) {
        var urlInfo = parseURL(root, request.url);

        validateFiles(urlInfo.pathnames, function (err, pathnames) {
            if (err) {
                response.writeHead(404);
                response.end(err.message);
            } else {
                response.writeHead(200, {
                    'Content-Type': urlInfo.mime
                });
                outputFiles(pathnames, response);
            }
        });
    }).listen(port);
}

function outputFiles(pathnames, writer) {
    (function next(i, len) {
        if (i < len) {
            var reader = fs.createReadStream(pathnames[i]);

            reader.pipe(writer, { end: false });
            reader.on('end', function() {
```

```
                next(i + 1, len);
            });
        } else {
            writer.end();
        }
    }(0, pathnames.length));
}

function validateFiles(pathnames, callback) {
    (function next(i, len) {
        if (i < len) {
            fs.stat(pathnames[i], function (err, stats) {
                if (err) {
                    callback(err);
                } else if (!stats.isFile()) {
                    callback(new Error());
                } else {
                    next(i + 1, len);
                }
            });
        } else {
            callback(null, pathnames);
        }
    }(0, pathnames.length));
```

}

可以看到，第二版代码在检查了请求的所有文件是否有效之后，立即就输出了响应头，并接着一边按顺序读取文件一边输出响应内容。并且，在读取文件时，第二版代码直接使用了只读数据流来简化代码。

第三次迭代

第二次迭代之后，服务器本身的功能和性能已经得到了初步满足。接下来我们需要从稳定性的角度重新审视一下代码，看看还需要做些什么。

设计

从工程角度上讲，没有绝对可靠的系统。即使第二次迭代的代码经过反复检查后能确保没有bug， 也很难说是否会因为 NodeJS 本身，或者是操作系统本身，甚至是硬件本身导致我们的服务器程序在某一天挂掉。因此一般生产环境下的服务器程序都配有一个 守护进程，在服务挂掉的时候立即重启服务。一般守护进程的代码会远比服务进程的代码简单，从概率上可以保证守护进程更难挂掉。如果再做得严谨一些，甚至守 护进程自身可以在自己挂掉时重启自己，从而实现双保险。

因此在本次迭代时，我们先利用 NodeJS 的进程管理机制，将守护进程作为父进程，将服务器程序作为子进程，并让父进程监控子进程的运行状态，在其异常退出时重启子进程。

实现

根据以上设计，我们编写了守护进程需要的代码。

```
var cp = require('child_process');



var worker;



function spawn(server, config) {

    worker = cp.spawn('node', [ server, config ]);

    worker.on('exit', function (code) {

        if (code !== 0) {

            spawn(server, config);

        }
```

```
    });
}

function main(argv) {
    spawn('server.js', argv[0]);
    process.on('SIGTERM', function () {
        worker.kill();
        process.exit(0);
    });
}

main(process.argv.slice(2));
```

此外，服务器代码本身的入口函数也要做以下调整。

```
function main(argv) {
    var config = JSON.parse(fs.readFileSync(argv[0], 'utf-8')),
        root = config.root || '.',
        port = config.port || 80,
        server;

    server = http.createServer(function (request, response) {
        ...
    }).listen(port);

    process.on('SIGTERM', function () {
```

```
        server.close(function () {
            process.exit(0);
        });
    });
}
```

我们可以把守护进程的代码保存为 daemon.js，之后我们可以通过 node daemon.js config.json 启动服务，而守护进程会进一步启动和监控服务器进程。此外，为了能够正常终止服务，我们让守护进程在接收到 SIGTERM 信号时终止服务器进程。而在服务器进程这一端，同样在收到 SIGTERM 信号时先停掉 HTTP 服务再正常退出。至此，我们的服务器程序就靠谱很多了。

第四次迭代

在我们解决了服务器本身的功能、性能和可靠性的问题后，接着我们需要考虑一下代码部署的问题，以及服务器控制的问题。

设计

一般而言，程序在服务器上有一个固定的部署目录，每次程序有更新后，都重新发布到部署目录里。而一旦完成部署后，一般也可以通过固定的服务控制脚本启动和停止服务。因此我们的服务器程序部署目录可以做如下设计。

```
- deploy/
    - bin/
        startws.sh
        killws.sh
    + conf/
        config.json
    + lib/
        daemon.js
        server.js
```

在以上目录结构中，我们分类存放了服务控制脚本、配置文件和服务器代码。

实现

按以上目录结构分别存放对应的文件之后，接下来我们看看控制脚本怎么写。首先是 start.sh。

```
#!/bin/sh

if [ ! -f "pid" ]

then

    node ../lib/daemon.js ../conf/config.json &

    echo $! > pid

fi
```

然后是 killws.sh。

```
#!/bin/sh

if [ -f "pid" ]

then

    kill $(tr -d '\r\n' < pid)

    rm pid

fi
```

于是这样我们就有了一个简单的代码部署目录和服务控制脚本，我们的服务器程序就可以上线工作了。

后续迭代

我们的服务器程序正式上线工作后，我们接下来或许会发现还有很多可以改进的点。比如服务器程序在合并 JS 文件时可以自动在 JS 文件之间插入一个;来避免一些语法问题，比如服务器程序需要提供日志来统计访问量，比如服务器程序需要能充分利用多核 CPU，等等。而此时的你，在学习了这么久 NodeJS 之后，应该已经知道该怎么做了。

小结

本章将之前零散介绍的知识点串了起来，完整地演示了一个使用 NodeJS 开发程序的例子，至此我们的课程就全部结束了。以下是对新诞生的 NodeJSer 的一些建议。

要熟悉官方 API 文档。并不是说要熟悉到能记住每个 API 的名称和用法，而是要熟悉 NodeJS 提供了哪些功能，一旦需要时知道查询 API 文档的哪块地方。

要先设计再实现。在开发一个程序前首先要有一个全局的设计，不一定要很周全，但要足够能写出一些代码。

要实现后再设计。在写了一些代码，有了一些具体的东西后，一定会发现一些之前忽略掉的细节。这时再反过来改进之前的设计，为第二轮迭代做准备。

要充分利用三方包。NodeJS 有一个庞大的生态圈，在写代码之前先看看有没有现成的三方包能节省不少时间。

不要迷信三方包。任何事情做过头了就不好了，三方包也是一样。三方包是一个黑盒，每多使用一个三方包，就为程序增加了一份潜在风险。并且三方包很难恰好只提供程序需要的功能，每多使用一个三方包，就让程序更加臃肿一些。因此在决定使用某个三方包之前，最好三思而后行。