**1 介绍**

Express.js无疑是当前Node.js中最流行的Web应用程序框架。它几乎成为了大多数Node.js web应用程序的基本的依赖，甚至一些例如Sails.js这样的流行的框架也是基于Express.js。然而你还有一些其他框架的选择，可以给你带来“sinatra”一样的感觉（译注：sinatra是一个简单的Ruby的Web框架，可以参考[这篇博文](http://pinghackers.com/2014/11/27/sinatra-introduction/)）。另外两个最流行的框架分别是Koa和Hapi。

这篇文章不是打算说服你哪个框架比另外一个更好，而是只是打算让你更好地理解每个框架能做什么，什么情况下一个框架可以秒杀另外一个。

**2 框架的背景**

我们将要探讨的两个框架看起来都非常相似。每一个都能够用几行代码来构建一个服务器，并都可以非常轻易地构建REST API。我们先瞧瞧这几个框架是怎么诞生的。

**2.1 Express**

2009年6月26日，TJ Holowaychuk提交了Express的第一次commit，接下来在2010年1月2日，有660次commits的Express 0.0.1版本正式发布。TJ和Ciaron Jessup是当时最主要的两个代码贡献者。在第一个版本发布的时候，根据github上的readme.md，这个框架被描述成

疯一般快速（而简洁）的服务端JavaScript Web开发框架，基于Node.js和V8 JavaScript引擎。

差不多5年的时间过去了，Express拥有了4,925次commit，现在Express的最新版本是4.10.1，由StrongLoop维护，因为TJ现在已经跑去玩Go了。

**2.2 Koa**

大概在差不多一年前的2013年8月17日，TJ Holowaychuk（又是他！）只身一人提交了Koa的第一次commit。他描述Koa为“表现力强劲的Node.js中间件，通过[co](https://github.com/tj/co)使用generators使得编写web应用程序和REST API更加丝般顺滑”。Koa被标榜为只占用约400行源码空间的框架。Koa的目前最新版本为0.13.0，拥有583次commits。

**2.3 Hapi**

2011年8月5日，WalmartLabs的一位成员Eran Hammer提交了Hapi的第一次commit。Hapi原本是Postmile的一部分，并且最开始是基于Express构建的。后来它发展成自己自己的框架，正如Eran在他的博客里面所说的：

Hapi基于这么一个想法：配置优于编码，业务逻辑必须和传输层进行分离..

Hapi最新版本为7.2.0，拥有3,816次commits，并且仍然由Eran Hammer维护。

最后我们一起来看看关于这几个流行的框架的社区数据：

**Metric** Express.js Koa.js Hapi.js

Github Stars 16,158 4,846 3,283

Contributors 163 49 95

Packages that depend on: 3,828 99 102

StackOverflow Questions 11,419 72 82

**3 创建一个服务器**

所有开发者要开发Node.js web应用程序的第一步就是构建一个基本的服务器。所以我们来看看用这几个框架构建一个服务器的时候有什么异同。

**3.1 Express**

**var** express = **require**('express');

**var** app = express();

**var** server = app.listen(3000, **function**() {

**console**.log('Express is listening to http://localhost:3000');

});

对于所有的node开发者来说，这看起来相当的自然。我们把express require进来，然后初始化一个实例并且赋值给一个为app的变量。接下来这个实例初始化一个server监听特定的端口，3000端口。app.listen()函数实际上包装了node原生的http.createServer()函数。

**3.2 Koa**

**var** koa = **require**('koa');

**var** app = koa();

**var** server = app.listen(3000, **function**() {

**console**.log('Koa is listening to http://localhost:3000');

});

你马上发现Koa和Express是很相似的。其实差别只是你把require那部分换成koa而不是express而已。app.listen()也是和Express一模一样的对原生代码的封装函数。

**3.3 Hapi**

**var** Hapi = **require**('hapi');

**var** server = **new** Hapi.Server(3000);

server.start(**function**() {

**console**.log('Hapi is listening to http://localhost:3000');

});

Hapi是三者中最独特的一个。和其他两者一样，hapi被require进来了但是没有初始化一个hapi app而是构建了一个server并且指定了端口。在Express和Koa中我们得到的是一个回调函数而在hapi中我们得到的是一个新的server对象。一旦我们调用了server.start()我们就开启了端口为3000的服务器，并且返回一个回调函数。这个server.start()函数和Koa、Express不一样，它并不是一个http.CreateServer()的包装函数，它的逻辑是由自己构建的。

**4 路由控制**

现在一起来搞搞一下服务器最重要的特定之一，路由控制。我们先用每个框架分别构建一个老掉渣的“Hello world”应用程序，然后我们再探索一下一些更有用的东东，REST API。

**4.1 Hello world**

**4.1.1 Express**

**var** express = **require**('express');

**var** app = express();

app.get('/', **function**(req, res) {

res.send('Hello world');

});

**var** server = app.listen(3000, **function**() {

**console**.log('Express is listening to http://localhost:3000');

});

我们用get()函数来捕获“GET /”请求然后调用一个回调函数，这个回调函数会被传入req和res两个对象。这个例子当中我们只利用了res的res.send()来返回整个页面的字符串。Express有很多内置的方法可以用来进行路由控制。get,post, put, head, delete等等这些方法都是Express支持的最常用的方法（这只是一部分而已，并不是全部）。

**4.1.2 Koa**

**var** koa = **require**('koa');

**var** app = koa();

app.use(**function** \*() {

**this**.body = 'Hello world';

});

**var** server = app.listen(3000, **function**() {

**console**.log('Koa is listening to http://localhost:3000');

});

Koa和Express稍微有点儿不同，它用了ES6的generators。所有带有\*前缀的函数都表示这个函数会返回一个generator对象。根本上来说，generator会同步地yield出数据（译注：如果对Python比较熟悉的话，应该对ES6的generator不陌生，这里的yield其实和Python的yield语句差不多一个意思），这个超出本文所探索的内容，不详述。在app.use()函数中，generator函数设置响应体。在Koa中，this这个上下文其实就是对node的request和response对象的封装。this.body是KoaResponse对象的一个属性。this.body可以设置为字符串, buffer, stream, 对象, 或者null也行。上面的例子中我们使用了Koa为数不多的中间件的其中一个。这个中间件捕获了所有的路由并且响应同一个字符串。

**4.1.3 Hapi**

**var** Hapi = **require**('hapi');

**var** server = **new** Hapi.Server(3000);

server.route({

method: 'GET',

path: '/',

handler: **function**(request, reply) {

reply('Hello world');

}

});

server.start(**function**() {

**console**.log('Hapi is listening to http://localhost:3000');

});

这里使用了server对象给我们提供的server.route内置的方法，这个方法接受配置参数：path（必须），method（必须），vhost，和handler（必须）。HTTP方法可以处理典型的例如GET、PUT、POST、DELETE的请求，\*通配符可以匹配所有的路由。handler函数被传入一个request对象的引用，它必须调用reply函数包含需要返回的数据。数据可以是字符串、buffer、可序列化对象、或者stream。

**4.2 REST API**

Hello world除了给我们展示了如何让一个应用程序运行起来以外几乎啥都没干。在所有的重数据的应用程序当中，REST API几乎是一个必须的设计，并且能让我们更好地理解这些框架是可以如何使用的。现在让我们看看这些框架是怎么处理REST API的。

**4.2.1 Express**

**var** express = **require**('express');

**var** app = express();

**var** router = express.Router();

// REST API

router.route('/items')

.get(**function**(req, res, next) {

res.send('Get');

})

.post(**function**(req, res, next) {

res.send('Post');

});

router.route('/items/:id')

.get(**function**(req, res, next) {

res.send('Get id: ' + req.params.id);

})

.put(**function**(req, res, next) {

res.send('Put id: ' + req.params.id);

})

.delete(**function**(req, res, next) {

res.send('Delete id: ' + req.params.id);

});

app.use('/api', router);

// index

app.get('/', **function**(req, res) {

res.send('Hello world');

});

**var** server = app.listen(3000, **function**() {

**console**.log('Express is listening to http://localhost:3000');

});

我们为已有的Hello World应用程序添加REST API。Express提供一些处理路由的便捷的方式。这是Express 4.x的语法，除了你不需要express.Router()和不能用app.user('/api', router)以外，其实上是和Express 3.x本质上是一样的。在Express 3.x中，你需要用app.route()替换router.route()并且需要加上/api前缀。Express 4.x的这种语法可以减少开发者编码错误并且你只需要修改少量代码就可以修改HTTP方法规则。

**4.2.2 Koa**

**var** koa = **require**('koa');

**var** route = **require**('koa-route');

**var** app = koa();

// REST API

app.use(route.get('/api/items', **function**\*() {

**this**.body = 'Get';

}));

app.use(route.get('/api/items/:id', **function**\*(id) {

**this**.body = 'Get id: ' + id;

}));

app.use(route.post('/api/items', **function**\*() {

**this**.body = 'Post';

}));

app.use(route.put('/api/items/:id', **function**\*(id) {

**this**.body = 'Put id: ' + id;

}));

app.use(route.delete('/api/items/:id', **function**\*(id) {

**this**.body = 'Delete id: ' + id;

}));

// all other routes

app.use(**function** \*() {

**this**.body = 'Hello world';

});

**var** server = app.listen(3000, **function**() {

**console**.log('Koa is listening to http://localhost:3000');

});

很明显，Koa并没有类似Express这样的可以减少编码重复路由规则的能力。它需要额外的中间件来处理路由控制。我选择使用koa-route因为它是由Koa团队维护的，但是还有很多由其他开发者维护的可用的中间件。Koa的路由和Express一样使用类似的关键词来定义它们的方法，.get(), .put(), .post(), 和 .delete()。Koa在处理路由的时候有一个好处就是，它使用ES6的generators函数来减少对回调函数的处理。

**4.2.3 Hapi**

**var** Hapi = **require**('hapi');

**var** server = **new** Hapi.Server(3000);

server.route([

{

method: 'GET',

path: '/api/items',

handler: **function**(request, reply) {

reply('Get item id');

}

},

{

method: 'GET',

path: '/api/items/{id}',

handler: **function**(request, reply) {

reply('Get item id: ' + request.params.id);

}

},

{

method: 'POST',

path: '/api/items',

handler: **function**(request, reply) {

reply('Post item');

}

},

{

method: 'PUT',

path: '/api/items/{id}',

handler: **function**(request, reply) {

reply('Put item id: ' + request.params.id);

}

},

{

method: 'DELETE',

path: '/api/items/{id}',

handler: **function**(request, reply) {

reply('Delete item id: ' + request.params.id);

}

},

{

method: 'GET',

path: '/',

handler: **function**(request, reply) {

reply('Hello world');

}

}

]);

server.start(**function**() {

**console**.log('Hapi is listening to http://localhost:3000');

});

对于Hapi路由处理的第一印象就是，相对于其它两个框架，这货是多么的清爽，可读性是多么的棒！即使是那些必须的method，path，handler和reply配置参数都是那么的赏心悦目（译注：作者高潮了）。类似于Koa，Hapi很多重复的代码会导致更大的出错多可能性。然而这是Hapi的有意之为，Hapi更关注配置并且希望使得代码更加清晰和让团队开发使用起来更加简便。Hapi希望可以不需要开发者进行编码的情况下对错误处理进行优化。如果你尝试去访问一个没有被定义的REST API，它会返回一个包含状态码和错误的描述的JSON对象。

**5 优缺点比较**

**5.1 Express**

**5.1.1 优点**

Express拥有的社区不仅仅是上面三者当中最大的，并且是所有Node.js web应用程序框架当中最大的。在经过其背后差不多5年的发展和在StrongLoop的掌管下，它是三者当中最成熟的框架。它为服务器启动和运行提供了简单的方式，并且通过内置的路由提高了代码的复用性。

**5.1.2 缺点**

使用Express需要手动处理很多单调乏味的任务。它没有内置的错误处理。当你需要解决某个特定的问题的时候，你会容易迷失在众多可以添加的中间件中，在Express中，你有太多方式去解决同一个问题。Express自诩为高度可配置，这有好处也有坏处，对于准备使用Express的刚入门的开发者来说，这不是一件好的事情。并且对比起其他框架来说，Express体积更大。

**5.2 Koa**

**5.2.1 优点**

Koa有着傲人的身材（体积小），它表现力更强；对比起其他框架，它使得中间件的编写变的更加容易。Koa基本上就是一个只有骨架的框架，你可以选择（或者自己写一个）中间件，而不用妥协于Express或者Hapi它们自带的中间件。它也是唯一一个采用ES6的框架，例如它使用了ES6的generators。

**5.2.2 缺点**

Koa不稳定，仍处于活跃的开发完善阶段。使用ES6还是有点太超前了，例如只有0.11.9+的Node.js版本才能运行Koa，而现在最新的Node.js稳定版本是0.10.33。和Express一样有好也有坏的一点就是，在多种中间件的选择还是自己写中间件。就像我们之前所用的router那样，有太多类似的router中间件可供我们选择。

**5.3 Hapi**

**5.3.1 优点**

Hapi自豪地宣称它自己是基于配置优于编码的概念，并且很多开发者认为这是一件好事。在团队项目开发中，可以很容易地增强一致性和可复用性。作为有着大名鼎鼎的WalmartLabs支持的框架和其他响当当的企业在实际生产中使用Hapi，它已经经过了实际战场的洗礼，企业们可以没有担忧地基于Hopi运行自己的应用程序。所有的迹象都表明Hapi向着成为的伟大的框架的方向持续成熟。

**5.3.2 缺点**

Hapi绝逼适合用来开发更大更复杂的应用。但对于一个简单的web app来说，它的可能有点儿堆砌太多样板代码了。而且Hapi的可供参考样例太少了，或者说开源的使用Hapi的应用程序太少了。所以选择它对开发者的要求更高一点，而不是所使用的中间件。

**6 总结**

我们已经看过三个框架一些棒棒的而且很实际的例子了。Express毫无疑问是三个当中最流行和最出名的框架。当你要开发一个新的应用程序的时候，使用Express来构建一个服务器可能已经成为了你的条件反射了；但希望现在你在做选择的时候会多一些思考，可以考虑选择Koa或者Hapi。Koa通过超前拥抱ES6和Web component的思想，显示了Web开发社区正在进步的对未来的承诺。对于比较大的团队和比较大的项目来说，Hapi应该成为首要选择。它所推崇的配置优于编码，对团队和对团队一直追求的可复用性都大有裨益。现在赶紧行动起来尝试使用一个新的框架，可能你会喜欢或者讨厌它，但没到最后你总不会知道结果是怎么样的，有一点无容置疑的是，它会让你成为一个更好的开发者。