原创

（6）Spring WebFlux性能测试——响应式Spring的道法术器

享学IT 2018-03-23 11:14:36 47290人阅读 14人评论

> 本系列文章索引《响应式Spring的道法术器》
> 前情提要 响应式流 | Reactor 3快速上手 | Spring WebFlux快速上手
> 本文源码

1.4 从负载测试看异步非阻塞的优势

前面总是“安利”异步非阻塞的好处，下面我们就实实在在感受一下响应式编程在高并发环境下的性能提升。异步非阻塞的优势体现在I/O操作方面，无论是文件I/O、网络I/O，还是数据库读写，都可能存在阻塞的情况。

我们的测试内容有三：

1. 首先分别创建基于WebMVC和WebFlux的Web服务，来对比观察异步非阻塞能带来多大的性能提升，我们模拟一个简单的带有延迟的场景，然后启动服务使用gatling进行测试，并进行分析；
2. 由于现在微服务架构应用越来越广泛，我们基于第一步的测试项目进一步观察调用存在延迟的服务的情况下的测试数据，其实主要是针对客户端的测试：阻塞的 `RestTemplate` 和非阻塞的 `WebClient` ；
3. 针对MongoDB的同步和异步数据库驱动进行性能测试和分析。

> 说明：本节进行的并非是严谨的基于性能调优的需求的，针对具体业务场景的负载测试。本节测试场景简单而直接，各位朋友GET到我的点即可。
> 此外：由于本节主要是进行横向对比测试，因此不需要特定的硬件资源配置，不过还是**建议在Linux环境下进行测试**，我最初是在Win10上跑的，当用户数上来之后出现了不少请求失败的情况，下边的测试数据是在一台系统为Deepin Linux（Debian系）的笔记本上跑出来的。

那么我们就开始搭建测试环境吧~（关于Spring WebFlux 不熟悉的话，请参考Spring WebFlux快速上手）。

1.4.1 带有延迟的负载测试分析

1）搭建待测试项目

我们分别基于WebMVC和WebFlux创建两个项目： `mvc-with-latency` 和 `WebFlux-with-latency` 。

为了模拟阻塞，我们分别在两个项目中各创建一个带有延迟的 `/hello/{latency}` 的API。比如 `/hello/100` 的响应会延迟100ms。

`mvc-with-latency` 中创建 `HelloController.java` ：

```
@RestController
public class HelloController {
    @GetMapping("/hello/{latency}")
    public String hello(@PathVariable long latency) {
        try {
            TimeUnit.MILLISECONDS.sleep(latency);   // 1
        } catch (InterruptedException e) {
            return "Error during thread sleep";
        }
        return "Welcome to reactive world ~";
    }
}
```

1. 利用sleep来模拟业务场景中发生阻塞的情况。

`WebFlux-with-latency` 中创建 `HelloController.java` ：

```
@RestController
public  class HelloController {
    @GetMapping("/hello/{latency}")
    public Mono<String> hello(@PathVariable int latency) {
        return  Mono.just("Welcome to reactive world ~")
                .delayElement(Duration.ofMillis(latency)); // 1
    }
}
```

1. 使用 `delayElement` 操作符来实现延迟。

然后各自在 `application.properties` 中配置端口号8091和8092：

```
server.port=8091
```

启动应用。

2）编写负载测试脚本

本节我们采用gatling来进行测试。创建测试项目 `gatling-scripts` 。

POM中添加gatling依赖和插件（目前gradle暂时还没有这个插件，所以只能是maven项目）：

```
<dependencies>
    <dependency>
        <groupId>io.gatling.highcharts</groupId>
        <artifactId>gatling-charts-highcharts</artifactId>
        <version>2.3.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>io.gatling</groupId>
            <artifactId>gatling-maven-plugin</artifactId>
            <version>2.2.4</version>
        </plugin>
    </plugins>
</build>
```

在 `src/test` 下创建测试类，gatling使用scala语言编写测试类：

```
import io.gatling.core.scenario.Simulation
import io.gatling.core.Predef._
import io.gatling.http.Predef._

import scala.concurrent.duration._

class LoadSimulation extends Simulation {

  // 从系统变量读取 baseUrl、path和模拟的用户数
  val baseUrl = System.getProperty("base.url")
  val testPath = System.getProperty("test.path")
  val sim_users = System.getProperty("sim.users").toInt

  val httpConf = http.baseURL(baseUrl)

  // 定义模拟的请求，重复30次
  val helloRequest = repeat(30) {
    // 自定义测试名称
    exec(http("hello-with-latency")
      // 执行get请求
      .get(testPath))
      // 模拟用户思考时间，随机1~2秒钟
      .pause(1 second, 2 seconds)
  }
```

```
  // 定义模拟的场景
  val scn = scenario("hello")
    // 该场景执行上边定义的请求
    .exec(helloRequest)

  // 配置并发用户的数量在30秒内均匀提高至sim_users指定的数量
  setUp(scn.inject(rampUsers(sim_users).over(30 seconds)).protocols(httpConf))
}
```

如上，这个测试的场景是：

- 指定的用户量是在30秒时间内匀速增加上来的；
- 每个用户重复请求30次指定的URL，中间会随机间隔1~2秒的思考时间。

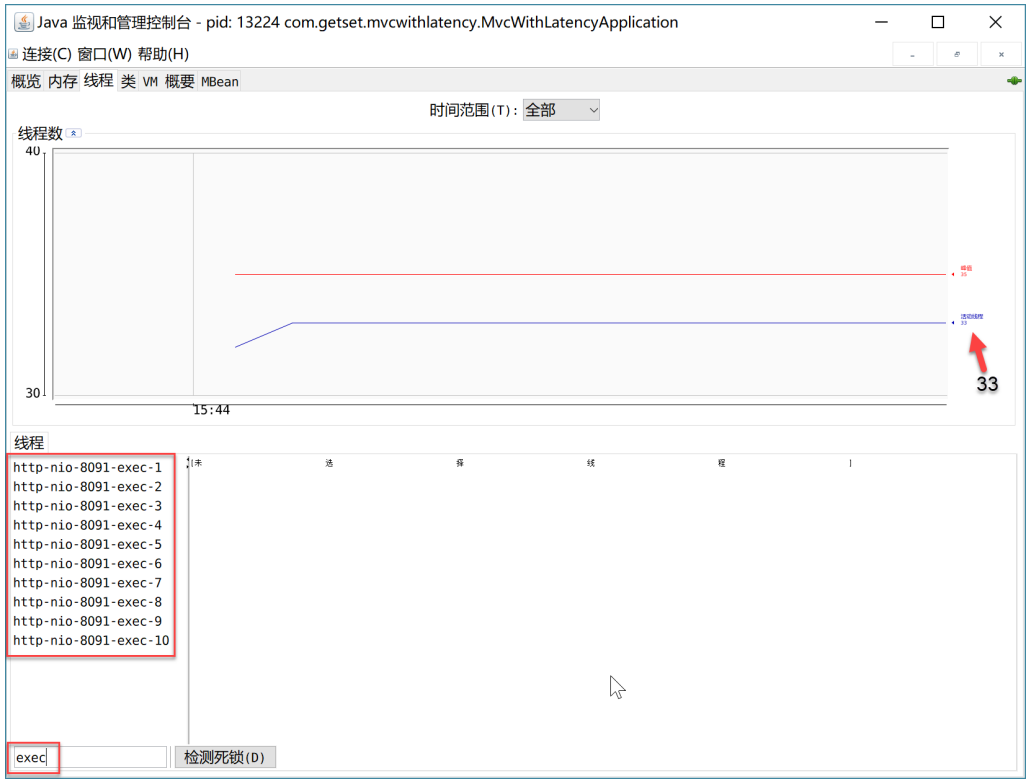
其中URL和用户量通过 `base.url` 、 `test.path` 、 `sim.users` 变量传入，借助maven插件，通过如下命令启动测试：

```
mvn gatling:test -Dgatling.simulationClass=test.load.sims.LoadSimulation -Dbase.url=http:/
```

就表示用户量为300的对 `http://localhost:8091/hello/100` 的测试。

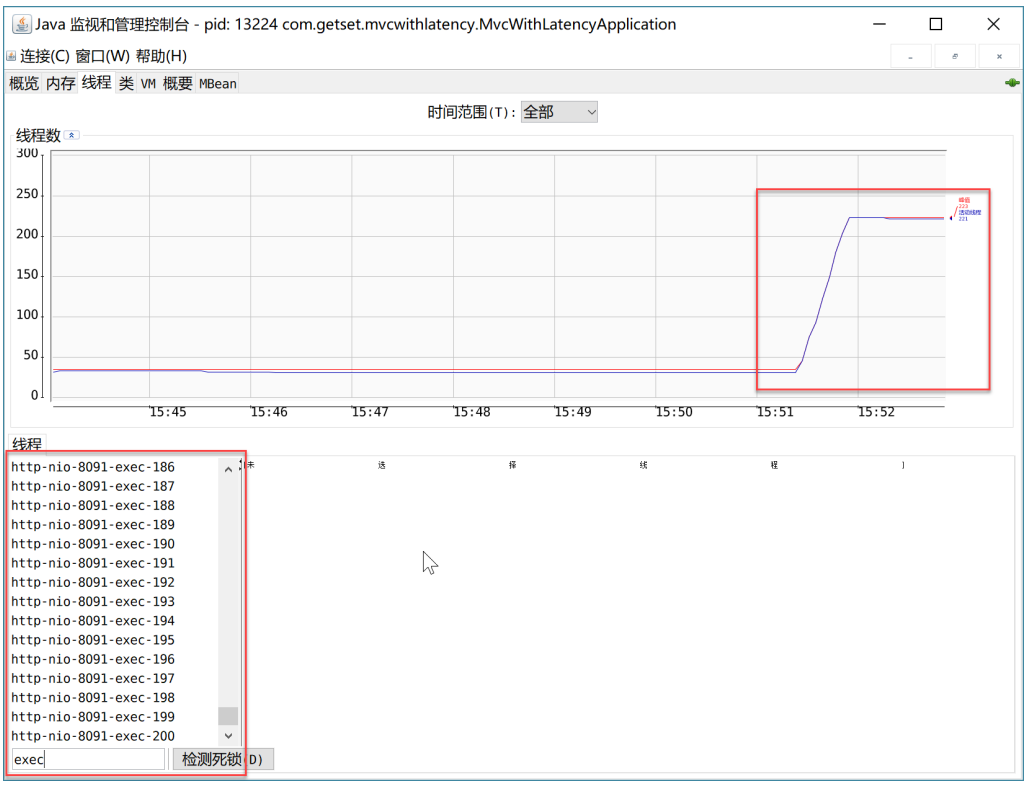
3）观察线程数量

测试之前，我们打开jconsole观察应用（连接MVCWithLatencyApplication）的线程变化情况：



如图（分辨率问题显示不太好）是刚启动无任何请求进来的时候，默认执行线程有10个，总的线程数31-33个。

比如，当进行用户数为2500个的测试时，执行线程增加到了200个，总的线程数峰值为223个，就是增加的这190个执行线程。如下：



由于在负载过去之后，执行线程数量会随机减少回10个，因此看最大线程编号估算线程个数的话并不靠谱，我们可以用“峰值线程数-23”得到测试过程中的执行线程个数。

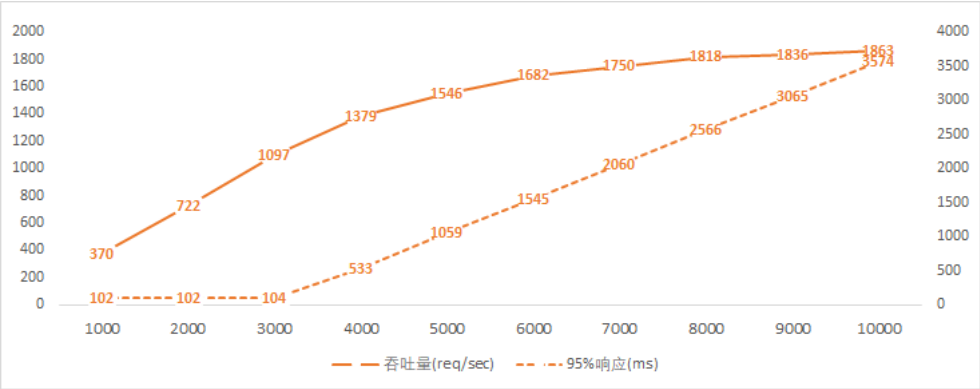
4）负载测试

首先我们测试 `mvc-with-latency` ：

- -Dbase.url=http://localhost:8091/；
- -Dtest.path=hello/100（延迟100ms）；
- -Dsim.users=1000/2000/3000/.../10000。

测试数据如下（Tomcat最大线程数200，延迟100ms）：

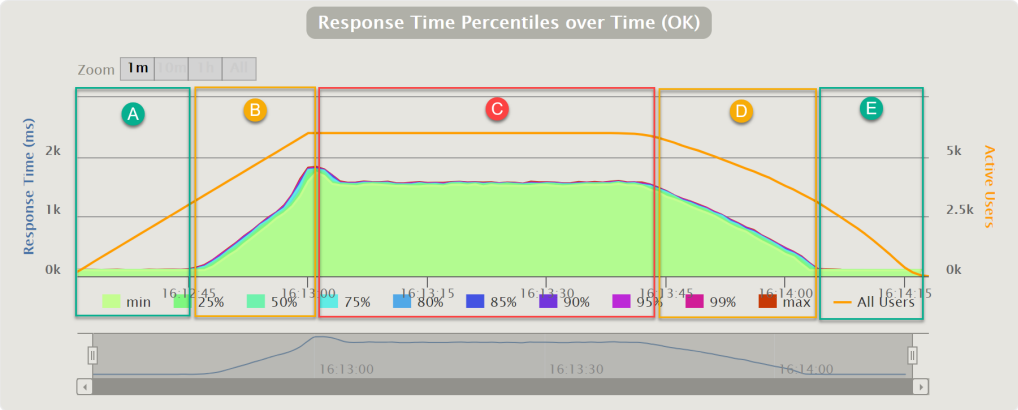
用户量	线程数	吞吐量(req/sec)	95%(ms)
1000	96	370	102
2000	173	722	102
3000	200	1097	104
4000	200	1379	533
5000	200	1546	1059
6000	200	1682	1545
7000	200	1750	2060
8000	200	1818	2566
9000	200	1836	3065
10000	200	1863	3574



由以上数据可知：

1. 用户量在接近3000的时候，线程数达到默认的最大值200；
2. 线程数达到200前，95%的请求响应时长是正常的（比100ms多一点点），之后呈直线上升的态势；
3. 线程数达到200后，吞吐量增幅逐渐放缓。

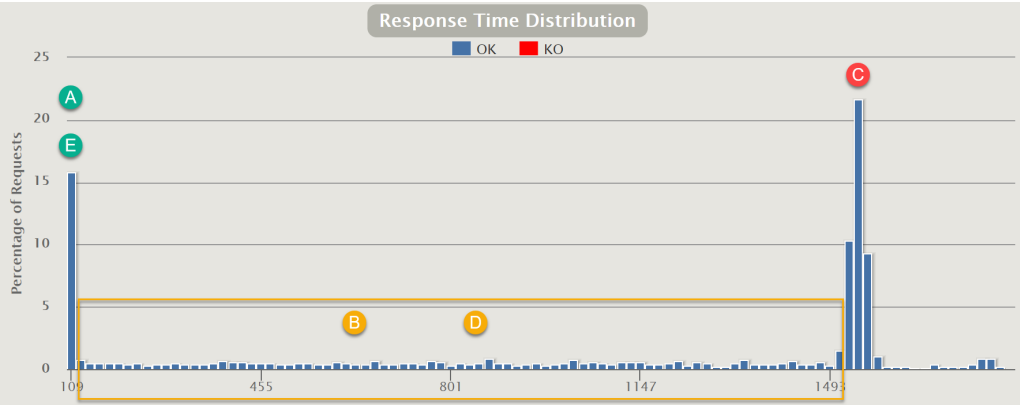
这里我们不难得出原因，那就是当所有可用线程都在阻塞状态的话，后续再进入的请求只能排队，从而当达到最大线程数之后，响应时长开始上升。我们以6000用户的报告为例：



这幅图是请求响应时长随时间变化的图，可以看到大致可以分为五个段：

- A. 有空闲线程可用，请求可以在100ms+时间返回；
- B. 线程已满，新来的请求开始排队，因为A和B阶段是用户量均匀上升的阶段，所以排队的请求越来越多；
- C. 每秒请求量稳定下来，但是由于排队，维持一段时间的高响应时长；
- D. 部分用户的请求完成，每秒请求量逐渐下降，排队情况逐渐缓解；
- E. 用户量降至线程满负荷且队列消化后，请求在正常时间返回；

所有请求的响应时长分布如下图所示：



A/E段与C段的时长只差就是平均的排队等待时间。在持续的高并发情况下，大部分请求是处在C段的。而且等待时长随请求量的提高而线性增长。

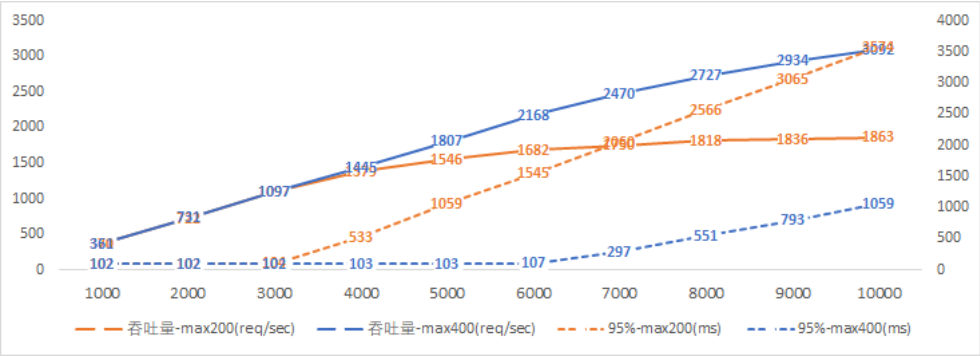
增加Servlet容器处理请求的线程数量可以缓解这一问题，就像上边把最大线程数量从默认的200增加的400。

最高200的线程数是Tomcat的默认设置，我们将其设置为400再次测试。在 `application.properties` 中增加：

```
server.tomcat.max-threads=400
```

测试数据如下：

用户量	max-thread-200			max-thread-400		
	线程数	吞吐量(req/sec)	95%(ms)	线程数	吞吐量(req/sec)	95%(ms)
1000	96	370	102	95	361	102
2000	173	722	102	169	731	102
3000	200	1097	104	260	1097	102
4000	200	1379	533	379	1445	103
5000	200	1546	1059	400	1807	103
6000	200	1682	1545	400	2168	107
7000	200	1750	2060	400	2470	297
8000	200	1818	2566	400	2727	551
9000	200	1836	3065	400	2934	793
10000	200	1863	3574	400	3092	1059



由于工作线程数扩大一倍，因此请求排队的情况缓解一半，具体可以对比一下数据：

1. “最大线程数200用户5000”的“95%响应时长”恰好与“最大线程数400用户10000”完全一致，我对天发誓，这绝对绝对是真实数据，更加巧合的是，吞吐量也恰好是1:2的关系！有此巧合也是因为测试场景太简单粗暴，哈哈；
2. “95%响应时长”的曲线斜率也是两倍的关系。

这也再次印证了我们上边的分析。增加线程数确实可以一定程度下提高吞吐量，降低因阻塞造成的响应延时，但此时我们需要权衡一些因素：

- 增加线程是有成本的，JVM中默认情况下在创建新线程时会分配大小为1M的线程栈，所以更多的线程异味着更多的内存；
- 更多的线程会带来更多的线程上下文切换成本。

我们再来看一下对于 `WebFlux-with-latency` 的测试数据：

用户量	max-thread-200			max-thread-400			用户量	webflux	
	线程数	吞吐量(req/sec)	95%(ms)	线程数	吞吐量(req/sec)	95%(ms)		吞吐量(req/sec)	95%(ms)
1000	96	370	102	95	361	102	1000	370	102
2000	173	722	102	169	731	102	2000	772	102
3000	200	1097	104	260	1097	102	3000	1084	103
4000	200	1379	533	379	1445	103	4000	1428	103
5000	200	1546	1059	400	1807	103	5000	1829	103
6000	200	1682	1545	400	2168	107	6000	2168	104
7000	200	1750	2060	400	2470	297	7000	2560	104
8000	200	1818	2566	400	2727	551	8000	2926	104
9000	200	1836	3065	400	2934	793	9000	3253	105
10000	200	1863	3574	400	3092	1059	10000	3614	105

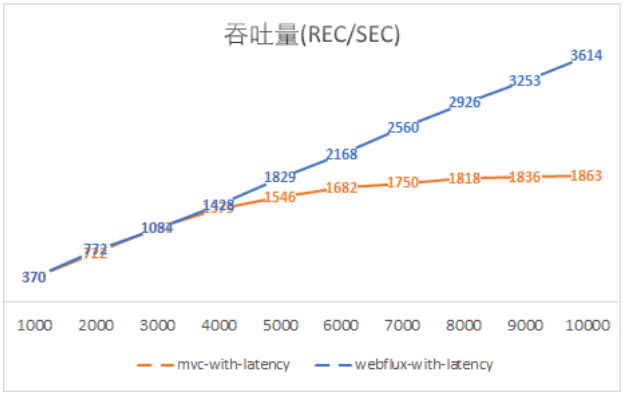
- 这里没有统计线程数量，因为对于运行在异步IO的Netty之上的WebFlux应用来说，其工作线程数量始终维持在一个固定的数量上，通常这个固定的数量等于CPU核数（通过jconsole可以看到有名为 `reactor-http-nio-X` 和 `parallel-X` 的线程，我这是四核八线程的i7，所以 X 从1-8），因为异步非阻塞条件下，程序逻辑是由事件驱动的，并不需要多线程并发；

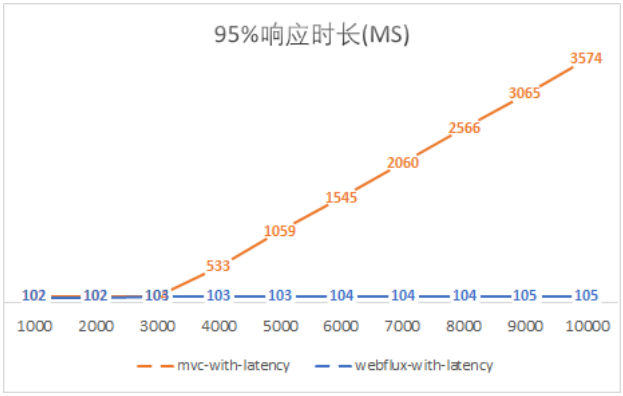
- 随着用户数的增多，吞吐量基本呈线性增多的趋势；
- 95%的响应都在100ms+的可控范围内返回了，并未出现延时的情况。

可见，非阻塞的处理方式规避了线程排队等待的情况，从而可以用少量而固定的线程处理应对大量请求的处理。

> 除此之外，我又一步到位直接测试了一下20000用户的情况：
> 1. 对 `mvc-with-latency` 的测试由于出现了许多的请求fail而以失败告终；
> 2. 而 `WebFlux-with-latency` 应对20000用户已然面不改色心不慌，吞吐量达到7228 req/sec（我擦，正好是10000用户下的两倍，太巧了今天怎么了，绝对是真实数据！），95%响应时长仅117ms。

最后，再给出两个吞吐量和响应时长的图，更加直观地感受异步非阻塞的WebFlux是如何一骑绝尘的吧：





综上来说，结论就是相对于Servlet多线程的处理方式来说，**Spring WebFlux在应对高并发的请求时，借助于异步IO，能够以少量而稳定的线程处理更高吞吐量的请求**，尤其是当请求处理过程如果因为业务复杂或IO阻塞等导致处理时长较长时，对比更加显著。

> 本文模拟的延迟时间较长，达到了100ms，虽然有些夸张，但是不能否认IO阻塞的严重性，还记得“1.2.1.1 IO有多慢？”中的比喻吗？如果CPU执行一条指令的时间是1秒，那么内存寻址就需要4分20秒，SSD寻址需要4.5天，磁盘寻址需要1个月。。。异步IO能够将CPU从“漫长”的等待中解放出来，不再需要堆砌大量的线程来提高CPU利用率。这也是Spring WebFlux能够以少量线程处理更高吞吐量的原因。

此时，我们更加理解了Nodejs的骄傲，不过我们大Java语言也有了Vert.x和现在的Spring WebFlux。

本节我们进行服务器端的性能测试，下一节继续分析Spring WebFlux的客户端工具WebClient的性能表现，它会对微服务架构的系统带来不小的性能提升呢！

响应式编程 Spring WebFlux 【打怪升级】响应式Spring

1 收藏 分享

上一篇：（5）Spring WebFlu... 下一篇：（7）Spring WebCli...

享学IT
52篇文章，187W+人气，0粉丝

提问和评论都可以，用心的回复会被更多人看到和认可

Ctrl+Enter 发布 取消 发布

14条评论 按时间正序 | 按时间倒序

memmsc
1楼 2018-06-08 14:40:01 2

您好，为什么我测试下来的结果都一样，我用的JMeter

> wx5b8bcf0fcbdc0:@memmsc
> 我也用JMeter测试 200并发，controller层延迟1000ms，然后都是webmvc的结果吊打 webflux的。再把并发提高到250以上，webflux就出现很多请求失败，报 'Connection refused: connect' 异常。难道是测试工具的原因？
>
> 2018-09-02 20:00:24 回复
>
> 作者 享学IT:@wx5b8bcf0fcbdc0
> 不好意思，才看到有通知。 webflux出现请求失败的情况我之前在测试的时候也有出现过，当时是在windows上起的，请求数量达到7000/s以上出现过，不过在linux上就没有这个问题了。所以本文也提到是建议用linux。不知道你那边是什么环境。如果webmvc性能没有问题的话，可以进jconsole看一下线程个数有没有打满，以及线程内是不是以nio的方式跑的，包括底层的web server，如果已经自动进行了优化，以异步IO的方式运行的话，那么两者结果估计就没有太大差别了。
>
> 2018-10-15 14:25:35 回复

memmsc
2楼 2018-06-08 14:40:26 1

博主可否提供测试代码下载地址

> 作者 享学IT:@memmsc
> 你好，测试的代码文章开头有给出：https://github.com/get-set/get-reactive/tree/master/gatling
>
> 2018-10-15 14:18:46 回复

memmsc
3楼 2018-06-08 14:58:05 2

server.tomcat.max-threads=1800 压测下来的效果高于 webflux

> 作者 享学IT:@memmsc
> 嗯，谢谢反馈，如果1800个线程没有打满的话，servlet同步IO的方式会比基于eventloop的异步IO的方式略快一点，因为响应式架构本身会有一定的成本，但是假设同步阻塞是1秒，但是1秒内接收到2000个请求，那么如果是servlet多线程同步处理的方式，1800个线程会打满，会有200个请求处于阻塞状态，这200个请求的延迟就会变成2s。所以如果压测发现效果高，可以看一下线程数有没有打满，有没有工作于异步IO方式下。
>
> 2018-10-15 14:30:37 回复
>
> 作者 享学IT:@memmsc
> 另外，你说得压测效果指的是吞吐量还是响应时长？
>
> 2018-10-15 14:36:33 回复

在线客服

wx5b9cfcdc4e47b
4楼 2018-09-15 20:39:25 1

我用的jmeter测试结果，也是springmvc比较优秀，同样50w请求，mvc吞吐6396.07025
webflux只有5230.45379

> 作者 享学IT:@wx5b9cfcdc4e47b
> 你好，请看一下高并发情况下，请求的相应时长，如果对于请求的处理存在阻塞的话，二者的平均响应时长会有较大的差距。
>
> 2018-10-15 14:36:11 回复

仰泳的双鱼
5楼 2018-11-08 21:04:38

你好博主，我也做了几轮测试，确实没有得到您当时的结果，不知道是否是我的测试过程有问题https://www.jianshu.com/p/b2d53667e7e2 ，不知道你当时测试的整体环境是什么样

kylekangkang
6楼 2018-11-29 17:17:32 1 1

你好，请问一下你这个吞吐量表格是在哪里生产的，gatling里面好像没有

> siyuedeyingzi:@kylekangkang
> 同问
>
> 2019-03-29 07:17:34 回复

Jason20Ming
7楼 2019-03-23 00:22:00 1

博主忽略了实际情况下阻塞是从哪里来的。
大多都是从DB的连接池访问里来的，由于DB存在交互协议复杂的原因，到现时为止还没有一款真正可用的Reactive化的驱动，所以大多数DB访问仍然是需要IO线程的同步阻塞等待的。
而很多其它NoSQL，MQ，Rpc类的调用都能很好Reactive化（简单的HTTP/TCP通讯模型只要系统支持是很简单的事），这些调用才能真正实现无额外线程开销的链路Reactive化。