HUAWEI

21天微服务实战营-Day10

华为云DevCloud & ServiceStage服务联合出品

Day10 CSE实战之微服务线程模型和性能统计

大纲

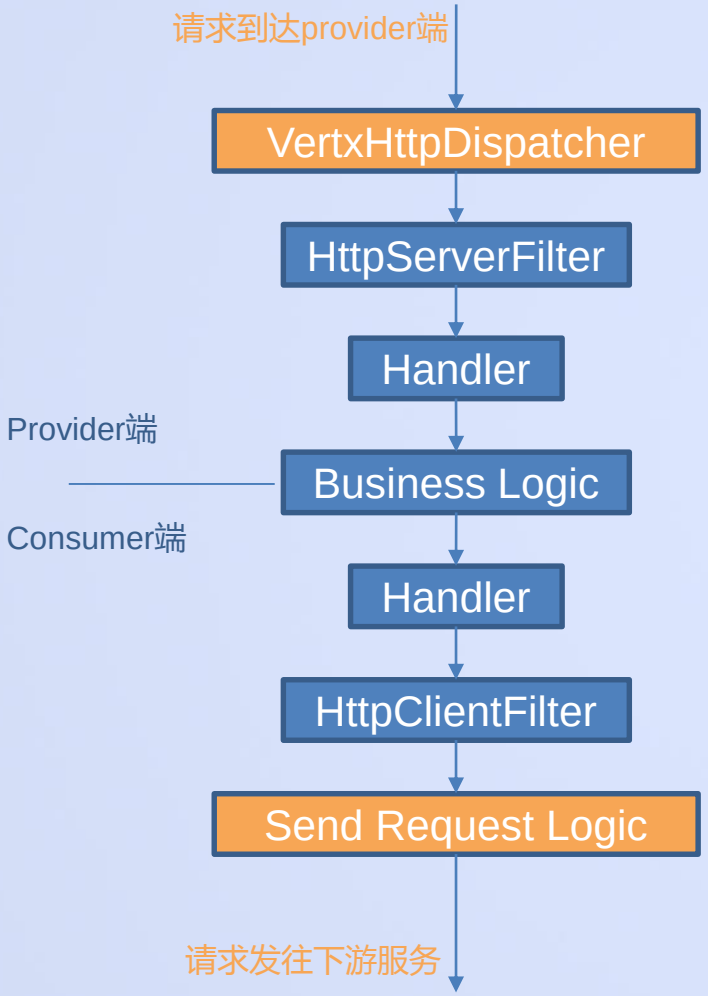
- 线程模型简介
- 性能统计（Metrics）

线程模型简介

ServiceComb（CSEJavaSDK）是基于Vert.x开发的。

Vert.x是一个依赖Netty，具有异步非阻塞特点的框架，它是CSEJavaSDK高性能的基础，但也让CSEJavaSDK的线程模型看上去与传统的服务框架有所不同。CSEJavaSDK线程模型的说明可以参考开源文档，使用CSEJavaSDK原生的默认开发方式时，其传输方式为Rest over Vertx传输方式。

线程模型简介——同步模型



原生的CSEJavaSDK框架开发的微服务默认工作于同步模式，传输方式为Rest over Vertx模式，基于Vert.x进行网络通信。
在此模式下，左图中橙色的部分是在网络线程中处理的，该部分逻辑代码是异步的，以避免阻塞网络线程；蓝色的部分是在业务线程池中处理的，可以是同步代码。

服务端方面，当请求到达微服务实例时，首先是网络线程从网络连接中接收到请求，经过一些处理后切换到业务线程运行provider端handler链、HttpServerFilter、用户的业务逻辑。切换到业务线程后，网络线程就可以去处理下一个请求了。这样可以使网络线程一直处于处理请求的状态，**开发者要避免做阻塞网络线程的操作**，如访问数据库、以同步逻辑代码发送REST请求等。

客户端方面，业务线程发送请求时，首先会在业务线程中对请求做一些处理（包括consumer端handler链、HttpClientFilter），然后转移到网络线程中进行发送。在等待应答的过程中，业务线程会一直处于阻塞状态。等到网络线程返回应答后，会通知业务线程继续运行后面的逻辑。

线程模型简介——reactive模型

除了同步模式，CSEJavaSDK也支持Reactive模式，该模式下所有的处理逻辑都运行在网络线程中。为避免阻塞网络线程，provider端服务接口代码和consumer端调用代码都需要是异步风格的。

Reactive模式的开发较为复杂，用户有兴趣的话可以查阅ServiceComb-Java-Chassis的资料了解相关信息。

Reactive在性能方面有着巨大的优势，但是却并非完美无缺的。它最大的问题就是要求整个项目的代码都运行于异步非阻塞的状态。一旦有一些第三方系统只有同步接口，比如某些数据库驱动三方件，那么这些地方的调用就不能直接放在业务逻辑中，否则会造成网络线程阻塞，性能打折扣。而即使使用线程池将其隔离，也会因为线程上下文的切换而带来额外开销。同时，异步风格的代码有违一般开发人员的习惯，写出来的代码可能不如传统的同步风格代码那么容易理解、调试和定位问题。

因此，是否使用Reactive模式进行开发，需要设计和开发人员结合实际情况进行取舍。

线程模型简介——reactive模型

同步风格

```java
@Path("/hello")
@GET
public String sayHello(@QueryParam("name") String name) {
    // RPC 调用方式体验与本地调用相同
    if (helloDelay.get() > 0) {
        try {
            Thread.sleep(helloDelay.get());
        } catch (InterruptedException e) {
            LOGGER.error("sayHello sleeping is interrupted!", e);
        }
    }
    return helloService.sayHello(name);
}
```

- Provider端业务接口直接返回响应消息
- Consumer端业务代码所使用的RPC代理接口也是直接返回响应消息的

reactive风格

```java
@Path("/hello")
@GET
public CompletableFuture<String> sayHello(@QueryParam("name") String name) {
    if (helloDelay.get() > 0) {
        try {
            Thread.sleep(helloDelay.get());
        } catch (InterruptedException e) {
            LOGGER.error("sayHello sleeping is interrupted!", e);
        }
    }

    CompletableFuture<String> response = new CompletableFuture<>();
    helloService.sayHello(name)
            .whenComplete((result, throwable) -> {
                if (null != throwable) {
                    LOGGER.error("invoke sayHello failed!", throwable);
                    response.completeExceptionally(throwable);
                    return;
                }

                LOGGER.info("get response: {}", result);
                response.complete(result);
            });
    return response;
}
```

- Provider端业务接口返回CompletableFuture
- Consumer端业务代码使用的RPC代理接口返回的也是CompletableFuture。如果要处理应答的话，在应答异步返回的时候处理

线程模型简介——reactive模型

普通微服务默认都是工作于同步模式的，但是EdgeService网关服务默认是工作于Reactive模式的。因此，在EdgeService网关进行handler和filter扩展定制时需要注意不能阻塞线程，如进行数据库查询、文件IO、同步网络调用等。如果有需要，可以考虑将这部分工作移到其他线程池中处理，网络调用可以改为reactive调用模式，或者直接将EdgeService网关服务的默认线程池设为普通服务所使用的同步线程池，在microservice.yaml文件中进行如下配置：

```
cse:
  executors:
    default: servicecomb.executor.groupThreadPool
```

线程模型简介

判断一个微服务的工作模式是否是Reactive模式，最直接的办法是在本地以Debug模式启动该服务，在Filter/Handler扩展类或者业务代码里打断点，观察线程名。例如在edge和consumer服务的HttpServerFilter扩展类里打断点，可以看到filter逻辑的执行线程名的命名格式不同：

edge服务，reactive模式

consumer服务，同步模式

TIPS：在同一个微服务中，同步模式和Reactive模式可以并存，一部分REST接口方法以同步模式处理请求，另一部分以Reactive模式处理请求

性能统计

CSEJavaSDK自带了一个简单好用的性能统计模块，只需要在maven依赖中引入metrics-core即可使用：

```
<dependency>
  <groupId>org.apache.servicecomb</groupId>
  <artifactId>metrics-core</artifactId>
</dependency>
```

为开启metrics日志打印功能，还需要在microservice.yaml文件中配置：

```
cse:
  metrics:
    publisher:
      defaultLog:
        enabled: true  # 是否在默认的日志中打印metrics日志
    window_time: 10000  # metrics日志打印周期，单位ms
```

TIPS：由于引入metrics-core模块会增加两个契约healthEndpoint和metricsEndpoint，分别描述的是健康检查和性能数据的REST接口，因此需要删除服务中心里的旧服务记录以更新契约。

性能统计

我们在consumer服务中开启metrics日志，重启并连续调用consumer服务，可以在consumer的日志中看到如下内容：

```
vertx:
  instances:
    name                eventLoopContext-created
    registry            2
    transport           10
    monitor-center      2
    config-center       2
  transport:
    client.endpoints:
      remote            connectCount    disconnectCount connections     send(Bps)       receive(Bps)
      (summary)         0               0               1               0               795
    server.endpoints:
      listen            connectCount    disconnectCount rejectByLimit   connections     send(Bps)       receive(Bps)
      0.0.0.0:9090      0               0               0               1               795             0
      (summary)         0               0               0               1               795             0
threadPool:
  corePoolSize maxThreads poolSize currentThreadsBusy queueSize taskCount completedTaskCount name
  8            8          0        0                  0         0.0       0.0                cse.executor.groupThreadPool-group1
  8            8          8        0                  0         30.4      30.4               cse.executor.groupThreadPool-group0
consumer:
  simple:
    status          tps     latency             operation
    rest.200        30      2.459/5.911         provider.hello.sayHello
                    30      2.459/5.911         (summary)
  details:
    rest.200:
      provider.hello.sayHello:
        prepare     : 0.010/0.018       handlersReq : 0.533/3.978       cFiltersReq: 0.017/0.084        sendReq     : 0.390/0.564
        getConnect  : 0.216/0.316       writeBuf    : 0.174/0.290       waitResp   : 1.263/2.355        wakeConsumer: 0.045/0.285
        cFiltersResp: 0.109/0.198       handlersResp: 0.090/0.285
producer:
  simple:
    status          tps     latency             operation
    rest.200        30      3.045/6.470         consumer.helloConsumer.sayHello
                    30      3.045/6.470         (summary)
  details:
    rest.200:
      consumer.helloConsumer.sayHello:
        prepare: 0.048/0.082        queue       : 0.133/0.217       filtersReq : 0.037/0.091        handlersReq: 0.113/0.220
        execute: 2.557/6.002        handlersResp: 0.019/0.046       filtersResp: 0.083/0.156        sendResp   : 0.048/0.081
```

Consumer服务既作为服务端接受edge服务的请求，也作为客户端调用provider服务，所以它的metrics日志会打印consumer/provider两方面的内容。

这里详细打印了连接建立、线程池工作状态、吞吐量、请求在内部各阶段的平均处理时间、最大处理时间等数据。

进行压测和性能调优时，可以打开metrics日志作为判断依据。

Thank You