Spring Boot 异步请求和异步调用

转载

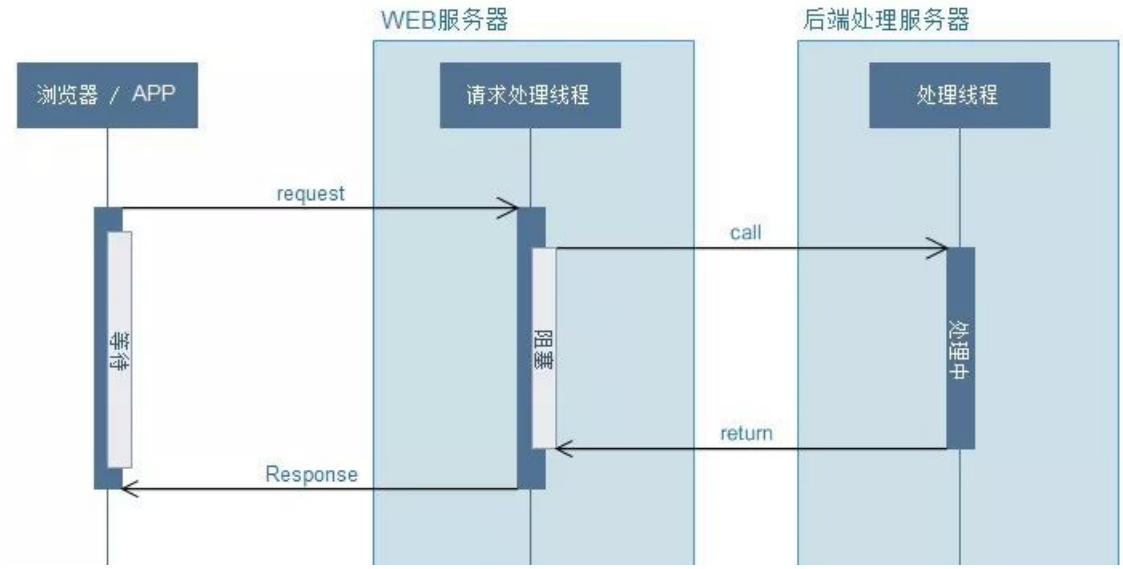
公众号:方志朋 于 2020-03-12 19:55:00 发布 18 收藏
文章标签： java spring 面试 aop 多线程

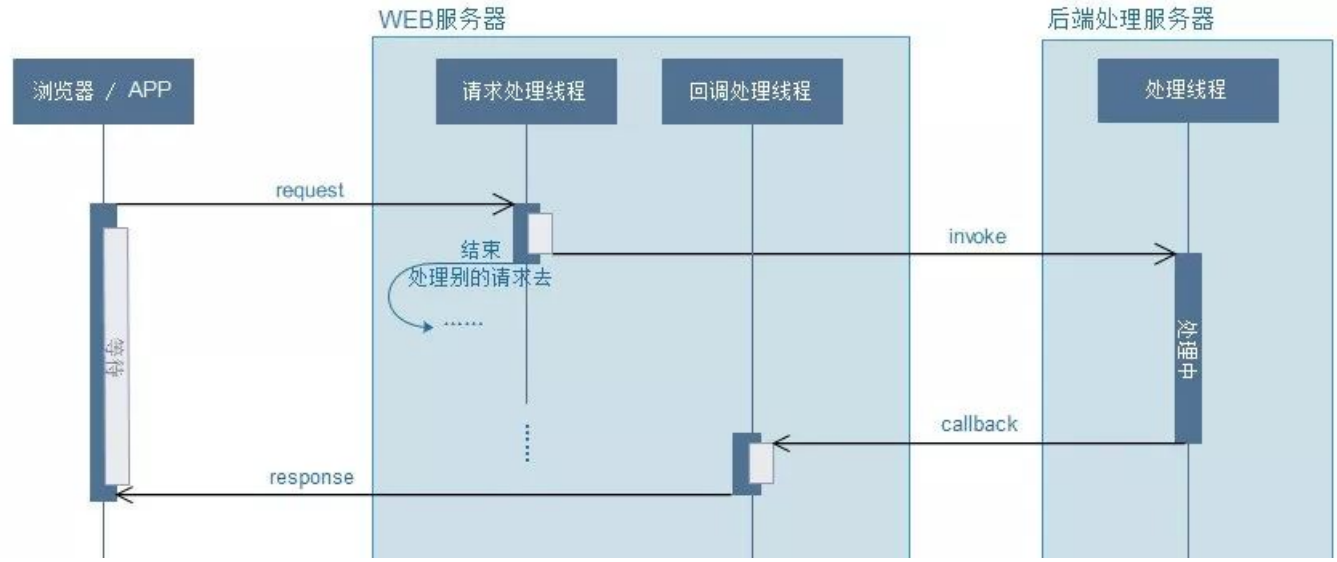
点击上方“方志朋”，选择“设为星标”

回复”666“获取新整理的面试文章

一、Spring Boot中异步请求的使用

1、异步请求与同步请求





特点：

可以先释放容器分配给请求的线程与相关资源，减轻系统负担，释放了容器所分配线程的请求，其响应将被延后，可以在耗时处理完成（例如长时间的运算）时再对客户端进行响应。

一句话：增加了服务器对客户端请求的吞吐量（实际生产上我们用的比较少，如果并发请求量很大的情况下，我们会通过nginx把请求负载到集群服务的各个节点上来分摊请求压力，当然还可以通过消息队列来做请求的缓冲）。

2、异步请求的实现

方式一：Servlet方式实现异步请求

```
@RequestMapping(value = "/email/servletReq", method = GET)
  public void servletReq (HttpServletRequest request, HttpServletResponse response) {
      AsyncContext asyncContext = request.startAsync();
      //设置监听器:可设置其开始、完成、异常、超时等事件的回调处理
      asyncContext.addListener(new AsyncListener() {
          @Override
          public void onTimeout(AsyncEvent event) throws IOException {
              System.out.println("超时了...");
              //做一些超时后的相关操作...
          }
          @Override
          public void onStartAsync(AsyncEvent event) throws IOException {
              System.out.println("线程开始");
          }
          @Override
          public void onError(AsyncEvent event) throws IOException {
              System.out.println("发生错误：+event.getThrowable());
          }
          @Override
          public void onComplete(AsyncEvent event) throws IOException {
              System.out.println("执行完成");
              //这里可以做一些清理资源的操作...
          }
      });
      //设置超时时间
      asyncContext.setTimeout(20000);
      asyncContext.start(new Runnable() {
          @Override
          public void run() {
              try {
                  Thread.sleep(10000);
                  System.out.println("内部线程：" + Thread.currentThread().getName());
                  asyncContext.getResponse().setCharacterEncoding("utf-8");
                  asyncContext.getResponse().setContentType("text/html;charset=UTF-8");
                  asyncContext.getResponse().getWriter().println("这是异步的请求返回");
              } catch (Exception e) {
                  System.out.println("异常：+e);
              }
              //异步请求完成通知
              //此时整个请求才完成
              asyncContext.complete();
          }
      });
      //此时之类 request的线程连接已经释放了
      System.out.println("主线程：" + Thread.currentThread().getName());
  }
```

方式二：使用很简单，直接返回的参数包裹一层callable即可，可以继承WebMvcConfigurerAdapter类来设置默认线程池和超时处理

```
@RequestMapping(value = "/email/callableReq", method = GET)
  @ResponseBody
  public Callable<String> callableReq () {
      System.out.println("外部线程：" + Thread.currentThread().getName());

      return new Callable<String>() {

          @Override
          public String call() throws Exception {
              Thread.sleep(10000);
              System.out.println("内部线程：" + Thread.currentThread().getName());
              return "callable!";
          }
      };
  }

  @Configuration
  public class RequestAsyncPoolConfig extends WebMvcConfigurerAdapter {

  @Resource
  private ThreadPoolTaskExecutor myThreadPoolTaskExecutor;

  @Override
  public void configureAsyncSupport(final AsyncSupportConfigurer configurer) {
      //处理 callable超时
      configurer.setDefaultTimeout(60*1000);
      configurer.setTaskExecutor(myThreadPoolTaskExecutor);
      configurer.registerCallableInterceptors(timeoutCallableProcessingInterceptor());
  }

  @Bean
  public TimeoutCallableProcessingInterceptor timeoutCallableProcessingInterceptor() {
      return new TimeoutCallableProcessingInterceptor();
  }
}
```

方式三：和方式二差不多，在Callable外包一层，给WebAsyncTask设置一个超时回调，即可实现超时处理

```
@RequestMapping(value = "/email/webAsyncReq", method = GET)
    @ResponseBody
    public WebAsyncTask<String> webAsyncReq () {
        System.out.println("外部线程：" + Thread.currentThread().getName());
        Callable<String> result = () -> {
            System.out.println("内部线程开始：" + Thread.currentThread().getName());
            try {
                TimeUnit.SECONDS.sleep(4);
            } catch (Exception e) {
                // TODO: handle exception
            }
            logger.info("副线程返回");
            System.out.println("内部线程返回：" + Thread.currentThread().getName());
            return "success";
        };
        WebAsyncTask<String> wat = new WebAsyncTask<String>(3000L, result);
        wat.onTimeout(new Callable<String>() {

            @Override
            public String call() throws Exception {
                // TODO Auto-generated method stub
                return "超时";
            }
        });
        return wat;
    }
```

方式四：DeferredResult可以处理一些相对复杂一些的业务逻辑，最主要还是可以在另一个线程里面进行业务处理及返回，即可在两个完全不相干的线程间的通信。

```
@RequestMapping(value = "/email/deferredResultReq", method = GET)
    @ResponseBody
    public DeferredResult<String> deferredResultReq () {
        System.out.println("外部线程：" + Thread.currentThread().getName());
        //设置超时时间
        DeferredResult<String> result = new DeferredResult<String>(60*1000L);
        //处理超时事件 采用委托机制
        result.onTimeout(new Runnable() {

            @Override
            public void run() {
                System.out.println("DeferredResult超时");
                result.setResult("超时了!");
            }
        });
        result.onCompletion(new Runnable() {

            @Override
            public void run() {
                //完成后
                System.out.println("调用完成");
            }
        });
        myThreadPoolTaskExecutor.execute(new Runnable() {

            @Override
            public void run() {
                //处理业务逻辑
                System.out.println("内部线程：" + Thread.currentThread().getName());
                //返回结果
                result.setResult("DeferredResult!!");
            }
        });
       return result;
    }
```

二、Spring Boot中异步调用的使用

1、介绍

异步请求的处理。除了异步请求，一般上我们用的比较多的应该是异步调用。通常在开发过程中，会遇到一个方法是和实际业务无关的，没有紧密性的。比如记录日志信息等业务。这个时候正常就是启一个新线程去做一些业务处理，让主线程异步的执行其他业务。

小插曲：更多Spring Boot 相关文章可以本公众号（Java后端）回复 技术博文，获取~

2、使用方式（基于spring下）

需要在启动类加入@EnableAsync使异步调用@Async注解生效

在需要异步执行的方法上加入此注解即可@Async("threadPool"),threadPool为自定义线程池。

代码略。。。就俩标签，自己试一把就可以了

3、注意事项

在默认情况下，未设置TaskExecutor时，默认是使用SimpleAsyncTaskExecutor这个线程池，但此线程不是真正意义上的线程池，因为线程不重用，每次调用都会创建一个新的线程。可通过控制台日志输出可以看出，每次输出线程名都是递增的。所以最好我们来自定义一个线程池。

调用的异步方法，不能为同一个类的方法（包括同一个类的内部类），简单来说，因为Spring在启动扫描时会为其创建一个代理类，而同类调用时，还是调用本身的代理类的，所以和平常调用是一样的。

其他的注解如@Cache等也是一样的道理，说白了，就是Spring的代理机制造成的。所以在开发中，最好把异步服务单独抽出一个类来管理。下面会重点讲述。。

4、什么情况下会导致@Async异步方法会失效？

调用同一个类下注有@Async异步方法：

在spring中像@Async和@Transactional、cache等注解本质使用的是动态代理，其实Spring容器在初始化的时候Spring容器会将含有AOP注解的类对象“替换”为代理对象（简单这么理解），那么注解失效的原因就很明显了，就是因为调用方法的是对象本身而不是代理对象，因为没有经过Spring容器，那么解决方法也会沿着这个思路来解决。

调用的是静态(static)方法

调用(private)私有化方法

5、解决4中问题1的方式（其它2,3两个问题自己注意下就可以了）

将要异步执行的方法单独抽取成一个类，原理就是当你把执行异步的方法单独抽取成一个类的时候，这个类肯定是被Spring管理的，其他Spring组件需要调用的时候肯定会注入进去，这时候实际上注入进去的就是代理类了。

其实我们的注入对象都是从Spring容器中给当前Spring组件进行成员变量的赋值，由于某些类使用了AOP注解，那么实际上在Spring容器中实际存在的是它的代理对象。那么我们就可以**通过上下文获取自己的代理对象调用异步方法。**

```
@Controller
@RequestMapping("/app")
public class EmailController {

    //获取ApplicationContext对象方式有多种,这种最简单,其它的大家自行了解一下
    @Autowired
    private ApplicationContext applicationContext;

    @RequestMapping(value = "/email/asyncCall", method = GET)
    @ResponseBody
    public Map<String, Object> asyncCall () {
        Map<String, Object> resMap = new HashMap<String, Object>();
        try{
            //这样调用同类下的异步方法是不起作用的
            //this.testAsyncTask();
            //通过上下文获取自己的代理对象调用异步方法
            EmailController emailController = (EmailController)applicationContext.getBean(EmailController.c
            emailController.testAsyncTask();
            resMap.put("code",200);
        }catch (Exception e) {
            resMap.put("code",400);
            logger.error("error!",e);
        }
        return resMap;
    }

    //注意一定是public,且是非static方法
    @Async
    public void testAsyncTask() throws InterruptedException {
        Thread.sleep(10000);
        System.out.println("异步任务执行完成！");
    }

}
```

开启cglib代理，手动获取Spring代理类,从而调用同类下的异步方法。首先，在启动类上加上@EnableAspectJAutoProxy(exposeProxy = true)注解。代码实现，如下：

```
@Service
@Transactional(value = "transactionManager", readOnly = false, propagation = Propagation.REQUIRED, rollback
public class EmailService {

    @Autowired
    private ApplicationContext applicationContext;

    @Async
    public void testSyncTask() throws InterruptedException {
        Thread.sleep(10000);
        System.out.println("异步任务执行完成！");
    }


    public void asyncCallTwo() throws InterruptedException {
        //this.testSyncTask();
// EmailService emailService = (EmailService)applicationContext.getBean(EmailService.class);
// emailService.testSyncTask();
        boolean isAop = AopUtils.isAopProxy(EmailController.class);//是否是代理对象；
        boolean isCglib = AopUtils.isCglibProxy(EmailController.class); //是否是CGLIB方式的代理对象；
        boolean isJdk = AopUtils.isJdkDynamicProxy(EmailController.class); //是否是JDK动态代理方式的代理对象；
        //以下才是重点!!!
        EmailService emailService = (EmailService)applicationContext.getBean(EmailService.class);
        EmailService proxy = (EmailService) AopContext.currentProxy();
        System.out.println(emailService == proxy ? true : false);
        proxy.testSyncTask();
        System.out.println("end!!!");
    }
}
```

三、异步请求与异步调用的区别

两者的使用场景不同，异步请求用来解决并发请求对服务器造成的压力，从而提高对请求的吞吐量；而异步调用是用来做一些非主线流程且不需要实时计算和响应的任务，比如同步日志到kafka中做日志分析等。

异步请求是会一直等待response相应的，需要返回结果给客户端的；而异步调用我们往往会马上返回给客户端响应，完成这次整个的请求，至于异步调用的任务后台自己慢慢跑就行，客户端不会关心。

四、总结

异步请求和异步调用的使用到这里基本就差不多了，有问题还希望大家多多指出。这边文章提到了动态代理，而spring中Aop的实现原理就是动态代理，后续会对动态代理做详细解读，还望多多支持哈。

作者 | 会炼钢的小白龙

链接 | cnblogs.com/baixianlong/p/10661591.html

热门内容：一个基于Spring Boot的API、RESTful API项目骨架
你能说出多线程中 sleep、yield、join 的用法及 sleep与wait区别吗？
试试 IntelliJ IDEA 自带的高能神器！我去，你写的 switch 语句也太老土了吧硬核干货：一位码农的架构师封神之路！
阿里问题定位神器 Arthas 的骚操作，定位线上BUG，超给力
用好idea这几款插件，可以帮你少写30%的代码！

最近面试BAT，整理一份面试资料《Java面试BAT通关手册》，覆盖了Java核心技术、JVM、Java并发、SSM、微服务、数据库、数据
获取方式：点“在看”，关注公众号并回复 666 领取，更多内容陆续奉上。
明天见(。·ω·。