SpringBoot异步接口实现：提高系统的吞吐量

前言

Servlet 3.0之前：每一次Http请求都由一个线程从头到尾处理。

Servlet 3.0之后，提供了异步处理请求：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，从而增加服务的吞吐量。

在springboot应用中，可以有4种方式实现异步接口（至于ResponseBodyEmitter、SseEmitter、StreamingResponseBody，不在本文介绍内，之后新写文章介绍）：

* AsyncContext
* Callable
* WebAsyncTask
* DeferredResult

第一中AsyncContext是Servlet层级的，比较原生的方式，本文不对此介绍（一般都不使用它，太麻烦了）。本文着重介绍后面三种方式。

特别说明：服务端的异步或同步对于客户端而言是不可见的。不会因为服务端使用了异步，接口的结果就和同步不一样了。另外，对于单个请求而言，使用异步接口会导致响应时间比同步大，但不特别明显。具体后文分析。

基于Callable实现

Controller中，返回一个java.util.concurrent.Callable包装的任何值，都表示该接口是一个异步接口：

```
@GetMapping(”/testCallAble”)
public Callable<String> testCallAble() {
```

```
    return () -> {
        Thread.sleep(40000);
        return "hello";
    };
}
```

服务器端的异步处理对客户端来说是不可见的。例如，上述接口，最终返回的客户端的是一个String，和同步接口中，直接返回String的效果是一样的。

Callable 处理过程如下：

控制器返回一个 Callable 。

* Spring MVC 调用 request.startAsync() 并将 Callable 提交给 AsyncTaskExecutor 以在单独的线程中进行处理。
* 同时， DispatcherServlet 和所有过滤器退出 Servlet 容器线程，但 response保持打开状态。
* 最终 Callable 产生结果，Spring MVC将请求分派回Servlet容器以完成处理。
* 再次调用 DispatcherServlet ，并使用 Callable 异步生成的返回值继续处理。

Callable默认使用SimpleAsyncTaskExecutor类来执行，这个类非常简单而且没有重用线程。在实践中，需要使用AsyncTaskExecutor类来对线程进行配置。

基于WebAsyncTask实现

Spring提供的WebAsyncTask是对Callable的包装，提供了更强大的功能，比如：处理超时回调、错误回调、完成回调等。本质上，和Callable区别不大，但是由于它额外封装了一些事件的回调，所有，通常都使用WebAsyncTask而不是Callable：

```
@GetMapping("/webAsyncTask")
public WebAsyncTask<String> webAsyncTask() {
    WebAsyncTask<String> result = new WebAsyncTask<>(30003, () -> {
        return "success";
```

```
    });
    result.onTimeout(() -> {
        log.info("timeout callback");
        return "timeout callback";
    });
    result.onCompletion(() -> log.info("finish callback"));
    return result;
}
```

这里额外提一下，WebAsyncTask可以配置一个超时时间，这里配置的超时时间比全局配置的超时时间优先级都高（会覆盖全局配置的超时时间）。

基于DeferredResult实现

DeferredResult使用方式与Callable类似，但在返回结果时不一样，它返回的时实际结果可能没有生成，实际的结果可能会在另外的线程里面设置到DeferredResult中去。

```
//定义一个全局的变量，用来存储DeferredResult对象
private Map<String, DeferredResult<String>> deferredResultMap = new ConcurrentHashMap<>();

@GetMapping("/testDeferredResult")
public DeferredResult<String> testDeferredResult(){
    DeferredResult<String> deferredResult = new DeferredResult<>();
    deferredResultMap.put("test", deferredResult);
    return deferredResult;
}
```

如果调用以上接口，会发现客户端的请求一直是在pending状态——等待后端响应。这里，我简单的将该接口返回的DeferredResult对象存放在了一个Map集合中，实际应用中可以设计一个对象管理器来统一管理这些个对象，注意：要考虑定时轮询（或其他方式）这些对象，将已经处理过或无效的DeferredResult对象清理掉（DeferredResult.isSetOrExpired方法可以判断是否还有效），避免内存泄露。

这里我又写了一个接口，模拟

```
@GetMapping("/testSetDeferredResult")
public String testSetDeferredResult() throws InterruptedException {
    DeferredResult<String> deferredResult =
deferredResultMap.get("test");
    boolean flag = deferredResult.setResult("testSetDeferredResult");
    if(!flag){
        log.info("结果已经被处理，此次操作无效");
    }
    return "ok";
}
```

其他线程修改DeferredResult的值：首先是从之前存放DeferredResult的map中拿到DeferredResult的值，然后设置它的返回值。当执行deferredResult.setResult之后，可以看到之前pending状态的接口完成了响应，得到的结果，就是这里设置的值。

这里也额外说下：在返回DeferredResult时也可以设置超时时间，这个时间的优先级也是大于全局设置的。另外，判断DeferredResult是否有效，只是一个简单的判断，实际中判断有效的并不一定是有效的（比如：客户端取消了请求，服务端是不知道的），但是一般判断为无效的，那肯定是无效了。

DeferredResult 处理过程如下：

* 控制器返回一个 DeferredResult 并将其保存在可以访问的内存队列或列表中。
* Spring MVC 调用 request.startAsync() 。
* 同时，DispatcherServlet 和所有配置的过滤器退出请求处理线程，但响应保持打开状态。
* 应用程序从某个线程设置 DeferredResult ，Spring MVC 将请求分派回 Servlet 容器。
* 再次调用 DispatcherServlet ，并使用异步生成的返回值继续处理。
*

提供一个线程池

异步请求，不会一直占用请求的主线程（tomcat容器中处理请求的线程），而是通过一个其他的线程来处理异步任务。也正是如此，在相同的最大请求数配

置下，异步请求由于迅速的释放了主线程，所以才能提高吞吐量。

这里提到一个其他线程，那么这个其他线程我们一般都不适用默认的，都是根据自身情况提供一个线程池供异步请求使用：（我给的参数都是测试用的，实际中不可照搬）

```
    @Bean(”mvcAsyncTaskExecutor”)
    public AsyncTaskExecutor asyncTaskExecutor() {
        ThreadPoolTaskExecutor executor = new
ThreadPoolTaskExecutor();
        // 线程池维护线程的最少数量
        //
asyncServiceExecutor.setCorePoolSize(Runtime.getRuntime().availablePr
ocessors() + 1);
        executor.setCorePoolSize(5);
        // 线程池维护线程的最大数量
        executor.setMaxPoolSize(10);
        // 线程池所使用的缓冲队列
        executor.setQueueCapacity(10);
        //    asyncServiceExecutor.prefersShortLivedTasks();
        executor.setThreadNamePrefix(”fyk-mvcAsyncTask-Thread-”);
//       asyncServiceExecutor.setBeanName(”TaskId” + taskId);
        //   asyncServiceExecutor.setKeepAliveSeconds(20);
        //调用者执行
        //    asyncServiceExecutor.setRejectedExecutionHandler(new
ThreadPoolExecutor.CallerRunsPolicy());
        executor.setRejectedExecutionHandler(new
ThreadPoolExecutor.DiscardPolicy());
        // 线程全部结束才关闭线程池
        executor.setWaitForTasksToCompleteOnShutdown(true);
        // 如果超过60s还没有销毁就强制销毁，以确保应用最后能够被关闭，而
不是阻塞住
        executor.setAwaitTerminationSeconds(30);
        executor.initialize();

        return executor;
    }
```

把这个线程池配置设置到异步请求配置中：

```
@Configuration
public class FykWebMvcConfigurer implements WebMvcConfigurer {

    @Autowired
    @Qualifier("mvcAsyncTaskExecutor")
    private AsyncTaskExecutor asyncTaskExecutor;

    @Override
    public void configureAsyncSupport(AsyncSupportConfigurer
configurer) {
        //异步操作的超时时间，值为0或者更小，表示永不超时
        configurer.setDefaultTimeout(60001);
        configurer.setTaskExecutor(asyncTaskExecutor);
    }
}
```

什么时候使用异步请求

异步请求能提高吞吐量，这个是建立在相同配置（这里的配置指的是：最大连接数、最大工作线程数）的情况下。因此并不是说任何接口都可以使用异步请求。比如：一个请求是进行大量的计算（总之就是在处理这个请求的业务方法时CPU是没有休息的），这种情况使用异步请求就没有多大意义了，因为这时的异步请求只是把一个任务从tomcat的工作线程搬到了另一个线程罢了。直接调大最大工作线程数配置也能到达要求。所以，真正使用异步请求的场景应该是该请求的业务代码中，大量的时间CPU是休息的（比如：在业务代码中请求其他系统的接口，在其他系统响应之前，CPU是阻塞等待的），这个时候使用异步请求，就可以释放tomcat的工作线程，让释放的工作线程可以处理其他的请求，从而提高吞吐量。

由于异步请求增加了更多的线程切换（同步请求是同一个工作线程一直处理），所以理论上会增加接口的耗时。但，这个耗时很短很短。

最后

看到这里，如果还是不太懂，或者想更多的了解下相关的内容，可以看下我的另外文章，或许有更多的帮助：

关于springboot内置tomcat最大请求数配置的一些问题

关于SpringBoot MVC接口超时时间的分析

原文链接: https://juejin.cn/post/7367186272849788962