Web 服务组合执行引擎中服务异步调用机制研究*

李玲勇[1], 高春鸣[2], 文华南[1]
(1. 湖南师范大学 数学与计算机科学学院 计算机系, 长沙 410081; 2. 湖南大学 计算机与通信学院 计算机系, 长沙 410082)

摘 要: 研究了 BPEL4WS 执行引擎 WebJetFlow 对 Web 服务的异步调用机制,在引擎的服务调用代理中对 Web 服务统一采用非阻塞双传输异步调用,提高了调用线程的利用率。同时引入了 cache 机制并设计了相应的 cache 替换算法,保证了引擎对异步调用结果消息的匹配效率以及数据安全性, 通过实验验证引擎的性能有了明显的提高。

关键词: Web 服务; BPEL4WS 引擎; 异步调用; cache 机制
中图分类号: TP311 **文献标志码**: A **文章编号**: 1001-3695(2010)02-0558-05
doi:10.3969/j.issn.1001-3695.2010.02.043

Study of service asynchronous invocation mechanism in Web composite services execution engine

LI Ling-yong[1], GAO Chun-ming[2], WEN Hua-nan[1]
(1. *Dept. of Computer, College of Mathematics & Computer Science, Hunan Normal University, Changsha* 410081, *China*; 2. *Dept. of Computer, College of Computer & Communication, Hunan University, Changsha* 410082, *China*)

Abstract: This paper studied the Web services asynchronous invocation mechanism of the BPEL4WS engine WebJetFlow. It used the non-blocking-dual-channel asynchronous invocation to invoke the Web services in the service invocation proxy of the engine in order to improve the utilization ratio of the invocation threads. Also it imported the cache mechanism and designed an algorithm for cache substitution, thus it ensured the efficiency in matching the asynchronous results and data security. Finally, the experiment results demonstrate that the performance of the engine has been improved obviously.

Key words: Web services; BPEL4WS engine; asynchronous invocation; cache mechanism

0 引言

Web 服务是当前 SOA 实现的主流技术,越来越多的企业需要在 SOA 体系结构和 Web 服务技术框架下将企业已有的应用以 Web 服务的形式发布,并整合业务伙伴的 Web 服务以实现功能聚合,提供新的组合服务。多个 Web 服务按照工作流模型实现服务组合以满足用户需求,服务组合流程由多个成分 Web 服务构建并使用 BPEL4WS(business process execution language for Web services)[1] 等规范语言描述。服务组合流程被部署到 BPEL4WS 执行引擎上运行,由引擎按照事先编排的顺序调用各成分 Web 服务来实现对流程的执行。

Web 服务组合执行引擎的 QoS(服务质量)保证是服务组合实用化的关键影响因素[2]。在当前网络流量日益膨胀的情况下,一个服务组合流程很可能会被大量客户端同时调用[3],并且一个引擎上可能同时部署着多个服务组合流程,因此引擎可能成为服务组合性能的瓶颈。同时,由于服务实现中的各种原因,有些 Web 服务可能要花费相当长的时间才能响应请求。例如,如果某个 Web 服务需要人工介入或批处理,该 Web 服务可能要花数天时间才能获得结果,这些延迟还可能导致某些传输机制超时。因此如何将长生命周期的服务调用转换为异步调用也是 BPEL4WS 执行引擎必须要解决的问题。

WebJetFlow 是一个多线程的、解耦流程执行和成分 Web 服务调用的 BPEL4WS 执行引擎[4]。WebJetFlow 可以将对 Web 服务调用的同步模式转换为单纯的异步调用模式,减少了引擎等待 Web 服务调用返回结果的时间,极大地提高了流程的并发访问执行效率,明显改善了系统吞吐量和性能[5]。

本文讨论了 WebJetFlow 对 Web 服务的异步调用机制,以及如何通过在引擎内核中引入 cache 机制并设计一种 cache 替换算法来提高 cache 的命中率,以此提高服务异步调用的响应信息与原请求信息的匹配效率,并进而使引擎对用户提供 QoS 保障。

本文分析了当前的 Web 服务异步调用机制,给出了 BPEL4WS 执行引擎 WebJetFlow 的异步调用的设计,阐述了如何通过 cache 技术来提高服务异步调用的效率,给出了测试结果与分析。

1 Web 服务异步调用机制分析

由 SOAP 定义的服务调用本质上是同步的,而且 WSDL 也

收稿日期: 2009-06-27; **修回日期**: 2009-07-26 **基金项目**: 湖南省研究生科研创新资助项目(125000-4026);湖南省重大科技攻关项目(2006GK4034)

作者简介:李玲勇(1983-),男,湖南郴州人,硕士研究生,主要研究方向为 Web services 与中间件等(liet2008@126.com);高春鸣(1957-),男,教授,博士,主要研究方向为软件体系结构;文华南(1981-),男,硕士研究生,主要研究方向为数据挖掘与电子商务.

只支持同步的交互模型,但并不是所有的 Web 服务都会同步工作,有时候对 Web 服务请求的响应并不是立即提供的,而是在最初的请求事务完成后的某个时候提供,即需要服务之间的异步通信。

目前对 Web 服务的异步调用主要通过回调机制来实现。回调是一种双向调用模式,也就是说,被调用方在接口被调用时也会调用对方的接口。为实现自动回调机制,请求者在发送请求消息前必须进行回调注册,即注册相应的标志信息和相应的结果处理组件。发送请求后即可释放请求线程,处理其他事务。当服务器处理完请求消息后,通过回调函数,把结果消息发送到服务请求者注册的回调地址。请求者在接收到结果消息后会自动匹配注册的回调信息,触发相应的组件来处理结果。

Web 服务的异步调用使得客户端可以不必等待 Web 服务的返回结果而继续执行,调用结果在客户端注册的回调函数(callback)中返回。因此对于客户端的多个调用不会随着调用数目的增多而发生显著的速度变化,同时即使对某个 Web 服务的调用出现异常也不会影响到其他服务的调用。

为支持 Web 服务的异步调用,必须解决一些同步调用时无须考虑的问题:

a)必须定义回调函数以及接收服务响应结果的回调地址,并确保向服务提供方通知了这个地址。

b)客户端发送的服务请求和服务端返回的响应结果应该分别使用不同的传输通道,因为如果使用相同的传输通道,一旦服务端对请求的处理所花的时间较长,可能造成客户端的传输通道超时。

c)在不同的传输通道中,客户端和服务端都要能够正确地把服务的请求与响应关联起来。

当前的很多平台都对 Web 服务的异步调用提供了良好的支持。Apache 软件基金组织推出的开源 SOAP 引擎 Apache Axis2 [6] 就为 Web 服务的异步调用和实现提供了一流的支持。

Axis2 可以同时在客户端和服务端对 Web 服务的异步调用提供支持,它既支持单向传输的 SMTP,又支持双向传输的 HTTP。因此,为了实现一次 Web 服务的请求—响应交互,在引擎端可以使用两个 SMTP 传输通道或者两个发送即关闭的 HTTP 传输通道(在发送请求消息之后马上关闭该 HTTP 通道)来实现传输级别的异步。在引擎中可以使用 useSeparateListener(true)标志来通知 Axis2 对此调用使用两个不同的传输通道。这种传输级别的异步需要消息相关性机制的支持,也就是如何才能在两个独立的传输通道中正确地把服务的请求与响应关联起来,引擎 WebJetFlow 和 Axis2 都支持 WS-Addressing[7],通过使用〈wsa:MessageID〉和〈wsa:RelatesTo〉SOAP Header 来实现消息相关性,以此来支持对模块寻址。

同时,在 Axis2 中可以使用 Non-Blocking API 来异步调用 Web 服务,引擎可以通过一个 Non-Blocking API 把请求传递给 Axis2,而引擎继续去做其他工作。引擎将 Axis2 中的 AxisCallback 类包装成一个回调对象,在该回调对象中,定义了接收服务响应结果的回调地址,当 Axis2 从服务器端收到响应时,就会通知这个回调对象。

至此,在引擎中实现对 Web 服务的异步调用的三个关键问题已经迎刃而解。下面给出 BPEL4WS 执行引擎 WebJetFlow 对 Web 服务异步调用的详细设计。

2　WebJetFlow 的异步调用机制分析

2.1　流程执行机制

高效的流程执行机制是提高引擎性能的关键,WebJetFlow 是一个采用双反馈控制环结构来保障服务请求的服务响应时间的 BPEL4WS 执行引擎[4]。对引擎内核的流程执行过程设计如图 1 所示。

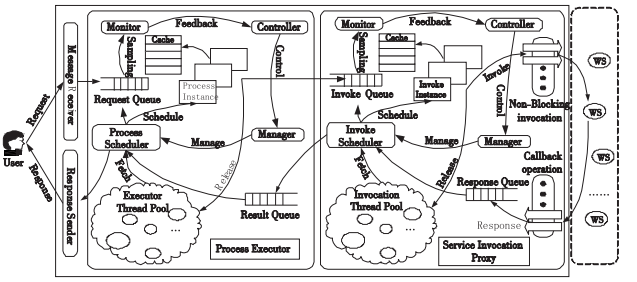

图1　带有双反馈控制环结构的引擎消息处理流程图

引擎内核主要由流程执行器和服务调用代理两部分组成。在引擎内核中创建两个独立的线程池,即执行线程池和调用线程池,分别管理多线程模型中的执行线程和调用线程。引擎启动的时候初始化四个 JMS 消息队列:request 和 invoke 队列,分别用来缓存流程请求信息以及服务调用信息;response 和 result 队列,分别用来缓存服务调用响应信息以及 invoke 活动结果信息。

每一个流程请求消息到达之后都会被放入 request 队列进行缓存。由 process scheduler 调度该队列,逐个提取流程请求信息并交由从执行器线程池取出的流程执行线程处理。执行线程根据请求消息中的 partnerLink、portType、operation 三元组在数据库中查找是否已经存在与该消息匹配的流程实例。如果没有与该消息匹配的流程实例,将根据部署的服务组合流程信息创建一个实例并启动该实例,然后处理 BPEL4WS 流程中的所有内部数据流和控制流。

当流程实例执行到一个 invoke 活动时,将服务调用信息发送到服务调用代理中的 invoke 队列,并对该流程实例进行状态持久化,然后该执行线程继续去处理其他的请求信息;同时在服务调用代理中,由 invoke scheduler 调度 invoke 队列,逐个提取 invoke 信息并交由服务调用线程处理。服务调用线程将 invoke 信息包装成 OMElement 格式之后,通过 Non-Blocking API 执行真正的 Web 服务异步调用,引擎端异步调用 Web 服务代码如下(以能实现最大程度的异步执行的非阻塞双传输模式为例):

```
// 解析 invoke 信息并包装成 OMElement 格式
    OMElement invoke = parseInvoke();
    ServiceClient serviceClient = new ServiceClient();
    Options options = new Options();
// 绑定需要异步调用的 Web 服务地址
    options.setTo(new EndpointReference("http://wjf.cslab211.
com/SampleService"));
// 通知 Axis2 使用双通道传输
    options.setUseSeparateListener(true);
    serviceClient.setOptions(options);
// 创建 AxisCallback 回调对象 callback 用于接收异步响应的结果
    AxisCallback callback = new MyCallback();
// 非阻塞异步调用,结果回调给 callback
    serviceClient.sendReceiveNonBlocking(payload, callback);
```

```
// 执行其他操作,无须阻塞等待响应结果
……
```

如果流程中有 flow 分支活动,服务调用代理从调用线程池中取出多个线程同时执行 Web 服务调用,从而有效实现分支活动的并发执行。

引擎对外发布了一个 response receiver 服务,是一个标准的 Web 服务,作为回调地址专门负责接收引擎发出的 Web 服务调用异步返回的结果;然后将返回结果放入 response 队列,由 invoke 调度器调度该队列,将返回结果和 invoke 消息实例匹配之后,得到流程实例 ID,然后组装成 invoke 活动结果信息,将其送入 result 队列排队,由流程调度器调度该队列,根据 invoke 活动调用结果消息找到之前已经被持久化的流程实例,将其激活并按流程控制顺序及数据依赖执行下一个活动。依此类推,直到整个组合服务流程执行完毕。

流程执行器控制 BPEL4WS 流程的执行,而服务调用代理负责处理流程引擎对 Web 服务的调用,这样就解耦了流程的执行与对服务的调用,使得执行线程不需要等待调用结果而被释放去执行其他的流程实例,提高了流程执行线程的利用率;同时通过 Axis2 的 Non-Blocking API 向服务提供者发送 SOAP 消息,异步调用具体的 Web 服务,使得服务调用线程不需要等待服务响应结果而被释放去处理其他的 invoke 消息,提高了服务调用线程的利用率。

2.2　引擎消息格式设计

在异步调用过程中,客户端与服务端之间要交互两次而且处理周期的长短不确定,同时,在二次交互的过程中,客户端和服务端都会不断地接收或发出新的消息。因此,要实现异步调用,就要解决在不同的传输通道中,客户端和服务端如何将返回结果与请求消息正确地关联起来。引擎在发送 Web 服务请求消息时,必须携带能够惟一标志该请求消息的相关集信息。当服务器处理完该请求后,就可以根据消息相关集来确定该将结果发给哪个请求者。

为此,定义一个二元组的消息相关集 CS = 〈messageID, timeStamp〉。其中:messageID 为 UUID 格式的消息标志符;timeStamp 为消息发送的时间戳。UUID 标志符的极低重复率使得 messageID 值与时间戳组合在一起可以保证惟一地标志一条消息。

WebJetFlow 组合服务执行引擎是由消息驱动的,引擎内核组件之间通过共享四个消息队列的信息来实现交互,因此必须规范定义系统中各种消息的结构。引擎中的消息全部采用标准的 SOAP envelope 格式,在 header 里面加入了表示消息相关集的 messageID 和 timeStamp、流程标志符、流程实例标志符以及可以使引擎支持异步消息流以及跨多个传输协议发送消息的 WS-Addressing 规范中的元素,如〈messageID〉〈replyTo〉和〈relatesTo〉等。引擎消息 header 格式如下(为节省篇幅,命名空间的定义未列出):

```
〈soapenv:Header〉
  〈wjf:ProcessName〉流程标志〈/ wjf:ProcessName〉
  〈wjf:ProcessInstance〉流程实例标志〈/ wjf: ProcessInstance 〉
  〈wjf:TimeStamp〉该消息发送时的时间戳〈/ wjf: TimeStamp 〉
  〈wsa:Action〉指定处理该消息的操作〈/wsa: Action〉
  〈wsa:MessageID〉该消息的 UUID 标志符〈/wsa:MessageID〉
  〈wsa:ReplyTo〉〈wsa: Address〉响应消息接收地址〈/wsa: Address〉〈/wsa:ReplyTo〉
  〈wsa:RelatesTo〉源请求消息的 UUID 标志符〈/wsa:RelatesTo〉
〈/soapenv:Header〉
```

〈Action〉元素用来指定处理该消息的方法,包括服务地址、端口类型以及输入/输出操作名称。

〈ReplyTo〉元素用来指明异步接收结果的地址,对于 WebJetFlow 引擎,可以将该元素设置为引擎对外发布的 response receiver 服务地址,同时这个地址代表的也是一个标准的 Web 服务,服务地址如下:

```
〈wsa:ReplyTo〉〈wsa:Address〉
http://wjf.cslab211.com/ResponseReceiver
〈/wsa:Address〉〈/wsa:ReplyTo〉
```

结果消息不需要设置〈ReplyTo〉元素,但必须设置〈RelatesTo〉元素,以便引擎可以根据〈RelatesTo〉元素中的相关集 CS 将返回结果与已经被缓存或持久化的请求作匹配。

在引擎内核中流动的消息主要有以下四种:

a)流程请求信息。客户端通过 browser 或者以 RPC 方式发出的对引擎上所部署的组合服务流程的请求信息,信息中必须指明所要调用的动作。

b)服务调用信息。从执行器发送过来的被执行线程封装好了的 invoke 请求消息。

c)服务调用响应信息。Web 服务提供方返回的服务响应信息,等待与源服务调用信息的实例匹配。

d)Invoke 活动结果信息。当服务调用响应信息与源服务调用信息的实例匹配找到该 invoke 活动所属的流程实例标志之后,即形成本信息,它被压入 result 队列,等待流程调度器 process scheduler 的调度以用来激活流程实例继续执行,其 body 部分格式和服务调用响应信息相同。

3　WebJetFlow 的异步调用机制设计

3.1　利用 cache 技术提高异步响应结果的匹配效率

服务响应信息异步返回需要与之前已经被序列化的 invoke 实例匹配,invoke 活动结果消息返回需要与之前已经被序列化的流程实例匹配。当外部服务响应时间较长时,可以将 invoke 实例和流程实例序列化到硬盘中,但是如果外部服务响应时间很短,很有可能将流程实例序列化到硬盘中及反序列化到内存中的时间比等待外部服务响应的时间还要长,这等于人为地延长了系统的整体响应时间。

为了进一步提高异步响应结果的匹配效率,参照 CPU 的 cache 技术原理,采用如下设计:

在执行器和调用代理中分别开辟一块长度视系统实际内存大小而定的内存空间作为 executor cache 和 proxy cache,考虑存取速度的因素,采用一维数组结构来实现,分别定义为 cacheE 和 cacheP。在 process executor 中,流程实例 P_i 由于执行 invoke 活动而被序列化到硬盘中的同时,将其一个副本放入 cacheE[i];在 service invoker proxy 中,invoke 实例 I_i 由于需要等待外部服务响应而被序列化到数据库中,同时将其一个副本放入 cacheP[i],以保证 P_i 和 I_i 在各自 cache 中的索引值是一致的。因为一个 BPEL4WS 流程可能会包含多个 invoke 活动,所以在 P_i 的生命周期中可能会对应多个 I_i,但是在同一时刻,它们是一一对应的。

由于 cache 的大小不可能无限扩大,当 cache 中的数据已满的时候,必须将当前等待缓存的 P_i 和 I_i 按照一定的算法将 cache 中的某一条实例替换掉。这里的 cache 替换算法的优劣

将直接影响接下来的结果匹配效率。参考 LFRU[8] 算法,笔者设计了以下 cache 替换算法。

3.2　Cache 替换算法

假设 cacheE 的大小为 n,将 cacheE 中流程实例的访问概率按升序排列,访问概率符合 Zipf 法则[9]。根据 Zipf 法则,排序后第 k 个流程实例的访问概率为 $F_k=\frac{C}{k^{\alpha}}$,$C=\frac{1}{\sum_{i=1}^{n}\frac{1}{i^{\alpha}}}$。其中 $\alpha>0$。

虽然流程实例每次被命中之后都要从 cacheE 中删除,但是一旦执行到该流程的下一个 invoke 活动的时候,该流程实例又会再次被放入 cacheE 中以等待 invoke 活动返回结果。因此,P_k 有一个计数器 C_k(记录 P_k 进入 cacheE 的次数,每进入一次相当于访问 P_k 一次,C_k 的值越大说明 P_k 越受欢迎,访问频率越高,$C_k\geqslant 1$),P_k 还有两个时间戳 t_{first} 和 t_{last},分别记录其第一次和最后一次进入 cacheE 的时刻,假设发生替换动作的时刻为 t,所具有的优先权的值 $W_k=\frac{\Omega-m}{\Omega}\times V_k+\frac{m}{\Omega}\times\overline{V_k}$。其中:$m$ 是执行替换操作的次数;V_k 表示 P_k 上次被访问的时刻距替换时刻的间隔 $V_k=t-t_{\text{last}}$;$\overline{V_k}$ 表示 P_k 被访问的平均时间间隔 $\overline{V_k}=\frac{t-t_{\text{first}}}{C_k}$;$\Omega$ 是一个根据以往替换操作统计的经验值所设立的一个相对较大的常数,所以 $W_k=\frac{\Omega-m}{\Omega}\times(t-t_{\text{last}})+\frac{m}{\Omega}\times\frac{t-t_{\text{first}}}{C_k}$。

每个流程实例都有一个权值 W,执行替换操作时,首先计算 cacheE 中所有流程实例的 W 值,然后将 W 值最大的流程实例替换出 cacheE。

a)在刚开始执行替换操作时,即当 $m\to 0$ 时,$W_k\to V_k$,即将上一次被访问的时刻距替换时刻间隔值最大的实例替换出 cacheE,此时算法等同于 LRU 算法。

b) 当替换操作的次数足够多时,即当 $m\to\Omega$ 时,$W_k\to\overline{V_k}$,即将历次被访问的平均时间间隔最大的实例替换出 cacheE,平均被访问的时间间隔越大说明被访问的频率越小,此时算法等同于 LFU 算法。

在替换次数 $0\to\Omega$ 的过程中,算法从 LRU 逐步过渡到 LFU,很好地模拟了 LFRU 算法。

以上是执行器缓存 cacheE 的替换算法。由于流程实例和 invoke 实例是一一对应的,而且在各自的 cache 中的索引值相同,若 P_k 在 cacheE 中将流程实例 P_i 替换掉,则在调用代理缓存 cacheP 中后续执行替换操作时,应该用 invoke 实例 I_k 替换掉 I_i。

3.3　基于双 cache 的异步响应结果匹配流程

当异步返回的结果信息需要与原来的实例匹配的时候,优先查找 cache,找到相应的实例并与其组装;如果没有找到相应的实例就去数据库中执行查找操作。由于所有的实例只需要匹配一次,对于 cache 的命中率要求较高,否则反而增加了额外的在 cache 中搜索的开销。匹配成功以后,在 cache 中和数据库中被匹配到的实例都要删除。

Response receiver 接收到服务调用响应信息之后,将其放入 response 队列,然后由守护线程开始对结果进行匹配的流程。匹配策略如下:

a)若 response 队列不为空,从 response 队列中取出一条服务调用响应信息,根据其相关集 CS 在 cacheP 中查找被缓存的调用信息实例,否则转入 i)。

b)如果没有在 cacheP 中命中到相应的调用信息实例,转入 c),否则转入 d)。

c)根据相关集 CS 在数据库中查找被持久化的调用信息实例,匹配成功转 d),否则转 h)。

d)把被匹配到的调用信息中所包含的流程实例 ID 组装到服务调用响应信息的 header 中形成 invoke 活动结果信息并放入 result 队列,删除数据库中的该调用信息实例。如果在 cacheP 中命中,从数据库中抽取下一个调用信息实例填补(并非替换)到 cacheP 中,转入 e)。

e)调度 result 队列中的 invoke 活动结果信息,根据流程实例 ID 在 cacheE 中查找被缓存的流程实例。如果没有在 cacheE 中命中到相应的流程实例,转入 f),否则转入 g)。

f)根据流程实例 ID 在数据库中查找被持久化的流程实例,匹配成功转入 g),否则转入 h)。

g)将 invoke 活动结果信息 body 中的有效载荷传递给匹配到的流程实例,删除数据库中的该流程实例,如果在 cacheE 中命中,从数据库中抽取下一个流程实例填补到 cacheE 中,转入 i)。

h)抛出异常,本次匹配失败,转入 a)。

i)本次匹配成功,转入 a)。

为了缩短系统响应时间,响应结果与实例匹配组装成功之后马上将组装结果回送,而对于 cache 填充以及数库中的实例删除等操作由专门的清扫线程 sweeper 来负责完成。

4　测试结果与分析

4.1　测试目的

对 Web 服务组合执行引擎 WebJetFlow 进行了改进,将非阻塞双传输异步调用以及 cache 机制引入后,引擎的性能是否有了明显的提高。

4.2　测试方案与步骤

a)使用 Mercury 公司的 LoadRunner8.0 同时产生大量的虚拟用户来分别对引擎进行压力测试,引擎通过接收外部消息来启动已经部署的流程或继续执行已经启动的流程实例,当 CPU 利用率稳定在 80% 左右的时候,对比改进前后引擎的最大负载量变化情况。

b)使用 LoadRunner 模拟对引擎进行并发访问,开始使用 10 个虚拟用户进行并发,执行完毕之后再增加 10 个并发数。依此类推,直到并发数目达到 100 个,测试引擎在每个并发量档次的响应时间和成功率。

c)通过 LoadRunner 生成等级为 0、1 和 2 的请求各 100 个,在初始时刻将各个等级的请求同时提交 20 个,然后每隔 5 s 将各个等级的请求同时追加提交 20 个。引擎线程池初始线程数设置为 20 个,线程池的内线程上限设为 40 个。各个等级的线程配额初始值分别为 $u_0(t)=10$,$u_1(t)=6$,$u_2(t)=4$;各个等级的用户期望响应时间分别为 $R_0=7$ s,$R_1=12$ s,$R_2=15$ s。

4.3　测试结果

测试在两台 PC 机(奔腾 3.0 GHz 处理器,1 GB 内存,Nvidia7600 显卡)上完成,测试了引擎的最大负载量变化情况,如图 2 所示。引擎对每个并发量档次的响应时间,如图 3 所

示。引擎对每个并发量档次的成功率,如图 4 所示。

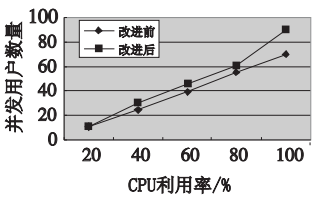

图2　改进前后引擎的最大负载量变化情况

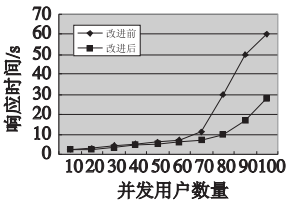

图3　改进前后引擎对每个并发量档次的响应时间

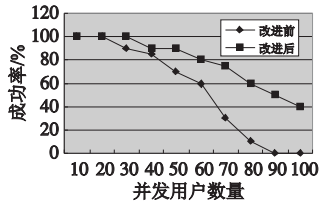

图4　改进前后引擎对每个并发量档次的成功率

通过测试,得到以下结论:

a)当 CPU 利用率稳定在 80% 左右的时候,改进后引擎的最大负载量比改进前大约提高了 13%,而满载负荷(CPU 占用率达到 100%)大约提高了 29%。

b)当虚拟用户的并发数目依次增加达到 100 个的时候,改进后引擎在每个并发量档次的响应时间和成功率分别比改进前都有了明显的提高。改进前引擎的并发用户达到 70 的时候就会出现响应异常的情况,改进之后的引擎对不超过 100 个的并发用户可以正常响应。

以上性能对比测试表明,本文将非阻塞双传输异步调用以及 cache 机制引入到 WebJetFlow 中,显著地提高了引擎的吞吐量,增强了引擎应付高负载请求的能力;引擎在每个并发量档次的响应时间和成功率分别都有了明显的提高。

5　结束语

在 Internet 上,网络流量和距离是导致基于 Web 服务的应用程序性能下降的主要原因之一,因此在遇到任何可能耗时较长的 Web 服务请求时,异步调用模式都是一个很好的方法。

本文讨论了 BPEL4WS 引擎 WebJetFlow 的 Web 服务异步调用机制,在引擎的服务调用代理中,对 Web 服务的调用采用非阻塞双传输异步调用,调用线程无须等待服务的响应,所有响应结果由统一的服务接口接收,调用线程可以继续去处理其他的调用请求,提高了调用线程利用率。同时引入了 cache 机制并设计了相应的 cache 替换算法,保证了那些调用外部响应时间短的服务流程实例可以很快被从内存中取出并往下执行,而对于等待时间较长的流程实例将会被持久化到数据库中,cache 中的实例保证了引擎对异步调用结果消息的匹配效率,数据库中的实例副本提供了数据安全性,即使掉电也不会丢失数据。

在以后的高负载实际测试环境中继续完善该异步调用机制,是笔者下一步要做的工作。

参考文献:

[1] IBM. BPEL4WS, business process execution language for Web services version 1.1[EB/OL]. (2007-02-08)[2009-05-20]. http://ww-w-128.ibm.com/developerworks/library/specification/ws-bpel/.

[2] CARDOSO J, SHETH A, MILLER J, *et al*. Quality of service for workflows and Web service processes[J]. Journal of Web Semantics, 2004, 1(3):281-308.

[3] NORRIS J, COLEMAN K, FOX A, *et al*. OnCall: defeating spikes with a free-market application cluster[C]//Proc of the 1st International Conference on Autonomic Computing. Washington DC: IEEE Computer Society, 2004: 198-205.

[4] 陈伟安,高春鸣. 一种基于反馈控制的 Web 服务组合执行引擎设计[J]. 计算机工程与应用, 2007,43(29):154-158.

[5] 袁小娟,高春鸣. Web 服务组合执行引擎中服务代理运行机制研究[J]. 计算机工程与应用, 2007, 43(28):103-106.

[6] The Axis2 Development Team. Apache Axis2 DOCS [DB/OL]. (2009-05-08)[2009-05-20]. http://ws.apache.org/axis2.

[7] BOSWORTH A, KALER C, FREY J, *et al*. Web services addressing (WS-Addressing)[EB/OL]. (2003-03-13)[2009-05-20]. http://ww-w.microsoft.com/taiwan/msdn/library/2003/Apr-2003/ws-addressin-ng.htm.

[8] 刘志明,彭宇行. 集群 VOD 系统中磁盘 cache 替换算法研究[J]. 计算机工程, 2004, 30(7):139-140,180.

[9] RONALD R P, CHASE J S, GADDE S, *et al*. The trickle-down effect: Web caching and server request distribution[J]. Computer Communications, 2002, 25(4):345-356.

(上接第 557 页)

参考文献:

[1] BERNERS-LEE T, HENDLER O L J. The semantic Web[J]. Scientific American, 2001,284(5):37-43.

[2] LEVENSHTEIN I V. Binary codes capable of correcting deletions, insertions, and reversals[J]. Soviet Physiscs Doklady, 1966,10(8):707-710.

[3] RUDI L, CILIBRASI P M B V. The Google similarity distance[J]. IEEE Trans on Knowledge and Data Engineering, 2007, 19(3):370-383.

[4] STUDER R, BENJAMINS V R D F. Knowledge engineering, principles and methods[J]. Data and Engineering,1998,25(1-2):161-197.

[5] FAATZ A, STEINMETZ R. Ontology enrichment with texts from the WWW[C]//Proc of WS'02. 2002.

[6] STABB S H A S. Authoring and annotation of Web pages in CREAM [C]//Proc of WWW2002. 2002.

[7] KIRYAKOV A, POPOV B, TERZIEV I, *et al*. Semantic annotation, indexing, and retrieval[J]. Web Semantics: Science, Services and Agents on the World Wide Web, 2004, 2(1):7-14.

[8] PAOLOZZI S, ATZENI P. Interoperability for semantic annotations [C]//Proc of the 18th International Conference on Database and Expert Systems Applications. 2007:445-449.

[9] [EB/OL]. http://annotation.semanticweb.org/tools.

[10] 邹亮,廖述梅. 基于本体的语义标注工具比较与分析[J]. 计算机应用,2004,24(S1):328-330.

[11] PORTOBELLO H C, CUNNINGHAM H, HUMPHREYS K, *et al*. GATE:a general architecture for text engineering[R]. [S.l.]:University of Sheffield, 2002.

[12] BILENKO M, MOONEY R J. Adaptive duplicate detection using learnable string similarity measures[C]//Proc of the 9th ACM SIGKOD International Conference on Knowledge Discovery and Data Mining. New York:ACM Press,2003:39-48.