基于事件驱动的高性能 WebSocket 服务器的设计与实现

曹文彬 谭新明 刘 备 刘传文
(武汉理工大学计算机科学与技术学院 湖北 武汉 430063)

摘 要 近年来即时通信在 Internet 上得到了越来越广泛的应用,传统服务器所遵循的"拉取"方式并不能很好地满足实际应用中信息获取实时性、用户请求高并发等方面的需求。为了改善上述问题,以"推送"方式进行实时消息传递成为研究热点。通过采用 WebSocket 技术,以 Node. js、Redis、RabbitMQ 等开源项目为基础,设计实现一个 WebSocket 服务器,能够对大量不同用户的订阅请求提供实时消息推送服务,并改进了 Node. js 多进程管理模块。实验结果表明,单个 WebSocket 服务器在处理 1 000 以上的并发连接时,错位率在 1.52% 左右,有效地实现了 WebSocket 服务器的高并发与稳定性。

关键词 WebSocket 服务器 高并发 异步 I/O Node. js 稳定性

中图分类号 TP31 **文献标识码** A DOI:10.3969/j. issn. 1000-386x. 2018. 01. 004

DESIGN AND IMPLEMENTATION OF EVENT-DRIVEN HIGH PERFORMANCE WEBSOCKET SERVER

Cao Wenbin Tan Xinming Liu Bei Liu Chuanwen
(*School of Computer Science and Technology, Wuhan University of Technology, Wuhan 430063, Hubei, China*)

Abstract In recent years, instant messaging has been more and more widely used on the Internet. The Pull model adopted in the traditional servers cannot well meet the requirements of real-time information acquisition and high concurrent using accesses in the practical applications. In order to deal with the aforementioned problem, using the Push model in the real-time message transmission has become a research hotspot. Based on open source projects of Node. js, Redis and RabbitMQ, a WebSocket server is designed and implemented. The WebSocket server can provide real-time message push service to a large number of different users' subscription requests. In the WebSocket server, the Node. js multi-process management module is modified. Experimental results show that when a single WebSocket server handles more than 1 000 concurrent connections, the dislocation rate is around 1. 52%, which effectively achieves the high concurrency and stability of the WebSocket server.

Keywords WebSocket server High concurrent Asynchronous I/O Node. js Stability

0 引 言

随着互联网 Web 2.0 技术的高速发展,基于浏览器的 Web 应用已经成为用户获取互联网信息的主要途径之一。Web 技术在不同领域的深入应用和信息时代高并发连接请求下海量数据的不断产生,使得 Web 应用程序对浏览器与服务器间的数据实时传递有了更高的要求[1],主要体现在数据传递的实时性和高并发带来的海量数据这两个方面。传统的即时通信大多基于"拉取"方式,比如文献[2],通过轮询查看服务器是否有新消息,缺点是定期轮询在低信息率的情况下,可能打开或者关闭许多不必要的连接造成资源的浪费。文献[3]对轮询进行了改进,会对通信双方建立的连接保持一段时间,但是还会存在通信效率低下、资源浪费的情况。文献[4]提出的是基于 Iframe 的流(streaming)技术,缺点是这种方式不可跨域,并且错误处理可控性不强。以上方式不仅没有提高数据传递的

收稿日期:2016 - 11 - 19。湖北省自然科学基金重点项目(2014CFA050)。**曹文彬**,硕士生,主研领域:移动互联网及大数据环境下数据处理平台。**谭新明**,教授。**刘备**,硕士生。**刘传文**,副教授。

实时性,反而制约了实时 Web 应用的性能。与传统方式相比,基于 WebSocket 协议的"推送"方式则是由服务器将消息主动地发送给浏览器用户。这种通信取代了单个 TCP 套接字,通过第一次请求连接就建立起双向通信实时响应的 Socket 连接,可以减少浏览器与服务器之间的交互次数,从而降低网络通信的负担,提高实时通信效率。

目前,主流语言都支持 WebSocket 协议在服务器端的实现,而本文采用基于 Node. js 平台的 JavaScript 语言来实现,主要原因有:1) W3C 标准化的 WebSocket API 和 Node. js 平台支持的语言都是 JavaScript,并且都是基于事件驱动模式编程,可以减低开发难度,缩短开发周期。2) Node. js 的事件驱动和非阻塞 I/O 模型在相对低系统资源耗用下性能突出,负载能力出众,十分适合在分布式设备上运行的数据密集型的实时 Web 应用[5-6]。总的来说,正是因为 Node. js 在处理高并发的用户连接请求时处理能力出色,所以才选择 Node. js 作为 WebSocket 服务器的实现平台,解决高并发所带来的问题。

本文考虑到实时数据传输的效率和可能面对的大量用户,基于 Node. js 设计了一款 WebSocket 服务器,可以应对高并发场景下实时数据的推送问题。在服务器的实现过程中,首先对 WebSocket 技术应用和 Node. js 平台进行了研究分析,根据 WebSocket 服务器应该具有的功能进行关键技术的选择。其次进行了 WebSocket 服务器的框架设计,并对其中的功能模块进行了详细说明。紧接着详细阐述了 WebSocket 服务器的实现细节,并将改善后的 Node. js 应用其中。最后通过功能测试和性能测实验证了 WebSocket 服务器实时数据推送的可行性。

1 相关工作

1.1 WebSocket 服务器的研究

随着 WebSocket 技术的发展,对于 Web 实时通信系统、社交订阅系统、监控系统等实时场景中的数据推送问题,目前已经逐步采用基于 WebSocket 的推送方式实现,取代了传统的拉取方式。通过研究发现,WebSocket 技术在服务器端已经有了广泛的研究应用。文献[7]中建立了一个 Web 应用程序来测量实时风传感器数据的单向传输延迟,该 Web 应用程序用 WebSocket 连接替代了 HTTP 连接,并与 HTTP 轮询和长轮询进行了比较。文献[8]提出了基于 WebSocket 的印刷包装机远程监控方法,并用 MFC 完成了服务端程序,有效提高了印刷包装行业的生产流程数字化程度。文献[9]将 WebSocket 运用在云监控系统中,实验结果表明 WebSocket 监控系统的平均延迟时间通常低于轮询,FlashSocket 和 Socket 解决方案。文献[10]设计了基于 B/S 架构的实时监测 Web 可视化系统,利用 WebSocket 在客户端和服务器端间建立全双工通信,实现生产数据实时推送,有效降低了网络吞吐量,提高了通信效率。文献[11]设计并实现了一种基于 WebSocket 推送技术和 SVG 技术的 B/S 模式下的实时监控系统,提高了监视系统的实时性、稳定性。文献[12]提出了一种基于 WebSocket 协议的服务器推送技术,利用了复杂时间处理技术对数据进行快速处理。

总的来说:1) 上述文献只提及到 WebSocket 技术在数据实时推送方面的优势,对于高并发请求可能带来的大量实时数据,并没有说明如何处理;2) 上述文献只是在服务器端引入了 WebSocket 技术,以实现某个场景下数据实时推送的功能,没有以 WebSocket 技术为核心,从服务器角度去设计、实现一款 WebSocket 服务器。综合以上分析,本文以 Node. js、Redis、RabbitMQ 等开源项目为基础,设计并实现了一个基于事件驱动的 WebSocket 服务器,不仅考虑到了传统的拉取方式难以保障数据的实时性,而且从服务器性能角度考虑,引入第三方框架,解决实时数据的缓存问题。

WebSocket 服务器利用 Node. js 平台是因为 Node. js 本身就适用于高并发的 Web 系统,可以给服务器的处理性能提供保障。利用缓存数据库 Redis 设计了分层的缓存层,实现数据的内部级存储。引入的异步消息队列 RabbitMQ,降低了 WebSocket 服务器的压力,提高了吞吐量。具体来说,本文的主要贡献如下:1) 在 Web 系统中的数据实时推送方面,引入第三方框架,设计并实现了一个 WebSocket 推送服务器;2) 对于实时推送服务器的运行平台 Node. js 在单进程方面存在的不足进行优化改进,使实时推送服务器在稳定和高并发性能方面更加突出。

1.2 Node. js 的高并发和稳定性研究

对于本文研究的高性能 WebSocket 服务器来说,高并发是服务器应该达到的首要性能指标,其次服务器在运行的过程中要保障稳定性。通过 Marc Fasel 从 CouchDB 上读取 JSON 数据来比较 Node. js 和 JavaEE 之间的性能,以及雷凯等对 Node. js 和 Apache 下的 Python 和 PHP 所做的性能对比实验可知,Node. js 在高并发的场景下表现稳定,并且可以对连接进行很好的响应[13-14]。

Node. js 作为 WebSocket 服务器中的关键技术,采

用基于事件驱动的异步 I/O 模型[15]。异步 I/O 是指如果调用线程执行 I/O 操作,线程只是发送 I/O 请求给操作系统,线程本身不被阻塞而继续执行,当 I/O 完成操作系统将会以事件的形式通知线程。事件驱动是指为了能够处理异步 I/O,线程一直在执行一个事件循环,循环检查是否有未处理的事件,并调用相应的事件处理函数。在 Node. js 整个异步 I/O 的执行回调过程中,事件循环类似于一个生产者/消费者模型,首先通过观察者来判断是否有事件需要处理,而不同的底层操作系统所提供的线程池部分正是通过 libuv 来消除差异。Node. js 提供的这一套处理机制,保证了用户的请求能够被整个异步 I/O 模型高效、快速的处理。Node. js 异步 I/O 的回调流程如图 1 所示。

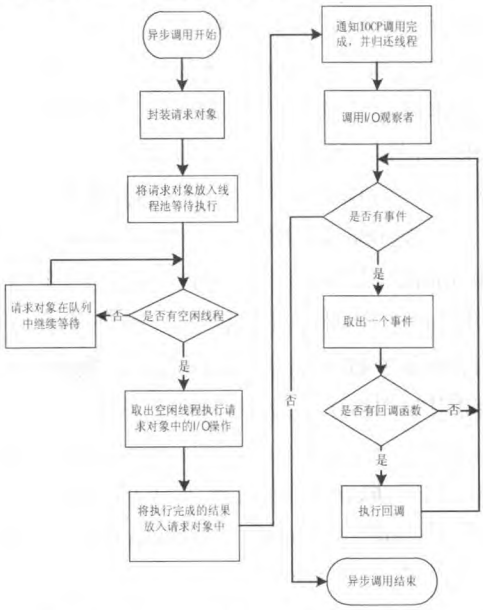

图 1　Node. js 异步 I/O 的回调流程图

图 1 的回调过程是在 Node. js 的单进程模式下完成的,通过研究发现,Node. js 的单进程模式在带来好处的同时也存在一些缺陷。首先,Node. js 的单线程无法利用多核 CPU,造成了多核 CPU 的浪费;其次,一旦单线程停止响应或者崩溃退出,错误会引起整个应用退出,服务器就会立即停止工作。在 Node. js 的现行版本中,官方内置了一个 Cluster 模块,该模块可以直接运行多个进程实现 Node. js 多核处理,能够提高 CPU 利用率。但是通过对 Cluster 模块源码的研究发现了一些问题:1) Cluster 模块在进程管理方面十分有限,判断子进程是否正常工作的方式是:给子进程分配新任务时,查看子进程是否有响应。随着失效的子进程越来越多时,可能出现所有的子进程都无法处理任务的情况。2) Cluster 在任务分配的策略上采用的是 Round-Robin 方式,虽然实现起来简单,但仅提供这一种分派策略在选择方案上显得单一,其次在不同的应用环境中因为存在各种细节上的差异,仅通过这一种方式无法保证每次任务分配达到的效果是合理的。

本文对 Node. js 的多进程模块进行了改进和优化,提高了 Node. js 的并发处理能力和稳定性,为 WebSocket 服务器的高性能提供了保障。

2　WebSocket 服务器的设计

整个服务器采用了三层架构方式,分别是用户服务层、中间层和数据服务层。用户服务层负责接收用户获取实时数据的连接请求,并将用户信息传递给中间层。中间层负责接收用户服务层传递过来的用户信息,并对数据服务层提供的实时数据进行处理。然后再把实时数据通过用户服务层推送给用户,数据服务层则会调用数据中心提供的 API 获取实时数据。用户服务层主要包括连接建立与维护、消息推送模块。中间层主要由任务分派、缓存数据库、进程监控、实时数据处理等功能模块组成。数据服务层包括实时数据的获取和消息队列,具体的分层架构设计如图 2 所示。

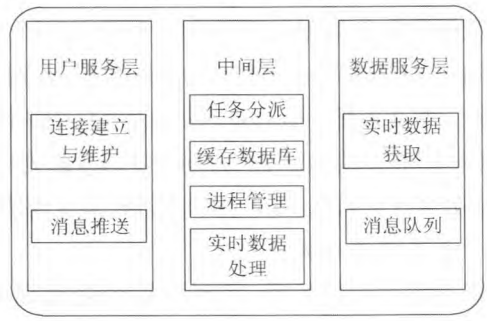

图 2　WebSocket 服务器的分层架构图

结合图 2 中的 WebSocket 服务器架构,对 WebSocket 服务器的处理流程进行了设计,如图 3 所示,主要包括以下部分:

(1) 用户请求连接

服务器在启动后,会一直监听来自用户的 WebSocket 请求。当用户的 WebSocket 请求到来,请求所携带订阅或取消订阅信息将会被 WebSocket 服务器解析,并将解析的请求信息存储到缓存数据库中。

(2) 缓存数据库

主要用来存储用户连接信息以及 worker 子进程的相关信息。目前 Redis, Memcached 都是 key-value 型的高性能缓冲数据库,但是 Redis 相比 Memcached 来说,支持更多服务器端的数据操作和更加丰富的数据结构[16]。本文中将采用 Redis 作为 WebSocket 服务器的缓存数据库,用以减少程序中内存泄露的风险。

(3) 主进程进行任务分派

WebSocket 服务器在启动时就会创建多个 worker

进程,通过负载均衡算法,主进程将 WebSocket 服务器接收到的客户端请求分发到各个 worker 子进程处理。每个 worker 子进程会处理多个任务,在缓存数据库中会有一个“子进程任务信息表”,用来存放子进程和任务之间的对应关系,便于进行消息推送。

(4) 进程管理

主要是对“子进程队列表”中的子进程进行监控、维护。缓存数据库中会存放一张“子进程队列表”,保存的是当前所有有效子进程的相关信息,包括处理任务数、是否有效、运行时间等。对于检测到的失效进程,会删除表中相应的进程信息,而对于重启的新进程,会把相关信息添加到表中。每隔一段时间,会收集子进程的状态信息更新“子进程队列表”,确保表中信息的准确性和时效性。

(5) 实时数据获取

“数据中心”存放的是用户请求的数据,一旦“数据中心”的数据有更新,WebSocket 服务器会根据用户的订阅在“数据中心”获取更新的数据进行响应。

(6) 消息队列

在数据获取和数据处理单元之间设计一个消息队列,主要是应对某时刻出现大量的更新数据,起到消峰减压的效果。本文采用了 RabbitMQ 作为消息中间件,首先将“数据中心”的消息发送到 RabbitMQ 消息队列中,再由用户请求处理子进程从 RabbitMQ 中进行消息消费。

(7) 子进程进行数据处理

根据不同的应用场景,实时数据的处理规则需要进行定制。WebSocket 服务器发送的数据可能是文本、图片、二进制或者是其他数据类型的数据,但是 WebSocket 协议只能处理文本数据消息和二进制数据消息,所以要进行数据类型的转换。

(8) 消息推送

每个 worker 进程都维护了自己服务的所有客户端连接,并且这些连接信息都存储在缓存数据库中,进程只需要将数据处理单元处理的数据发送到对应的用户连接中。

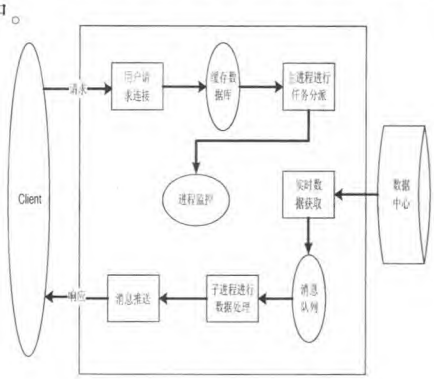

图 3 WebSocket 服务器处理流程示意图

通过对实际应用中的大多数即时通信服务器的研究,本节对 WebSocket 服务器的整体架构进行了设计。在设计的过程中着重考虑了以下几方面:1) 在工作场景中 WebSocket 服务器可能面临的高并发;2) 对于多进程的管理,以保障 WebSocket 服务器的稳定性;3) WebSocket 服务器的跨平台性和扩展性。

3 WebSocket 服务器的实现

3.1 用户请求连接

WebSocket 服务器的用户请求处理包括:用户请求连接和用户发送的消息格式定义。客户端请求 WebSocket 服务器建立连接的过程,就是打开阶段的握手协议,可以选择 Socket. IO 进行实现。

当用户与 WebSocket 服务器建立连接后,用户发送的消息体格式要满足一定规则:一种是数据信息的订阅请求,另外一种就是数据信息的取消订阅请求。订阅请求为“subscribe”,取消订阅请求为“unSubscribe”。当 WebSocket 服务器应用于多应用多租户模式中,用 appID 识别符区分不同的应用,每个请求携带的信息用“msg”识别符进行识别。每个请求可能订阅或取消订阅多个数据信息,若请求多个数据信息,数据信息之间用“,”隔开。最终订阅请求发送的字符串格式为"{appID:" + appID +",cmd:'subscribe',msg:[" + message +"]}",取消订阅请求发送的字符串格式为"{appID:" + appID +", cmd:“unSubscribe”, msg:[" + message +"]}"。

3.2 实时数据获取和数据处理

实时数据获取是指 WebSocket 服务器从数据中心获取实时更新的数据,数据中心提供获取实时数据的接口 getRealTimeData(int appID, List ids),其中 appID 是应用编号,ids 是用户请求订阅的数据 id 的集合。WebSocket 服务器采用每隔一定时间间隔调用一次接口,来保证数据的实时性。在这一部分的实现过程中,采取了“主动通知”模式。所谓主动通知,就是当数据发送更改时,主动通知子进程。在实现的过程中,需要创建一个独立的通知进程,为了不混合业务逻辑,此进程只用来发送数据更新的通知和查询状态是否更改。主动通知模式相比于所有的子进程都去定时轮询,降低了查询状态的开销,并且由于不会涉及多个进程进行状态查询,状态响应处的压力也不会过大,所以可以降低轮询时间,提升数据的准确性。

数据处理部分主要是指数据类型的转换,WebSocket 服务器发送的数据可能是文本、图片、二进制或者是其他数据类型的数据,然而 WebSocket 协议只能处理文

本数据消息和二进制数据消息[17]。因此 WebSocket 服务器从数据中心获得实时更新的数据后,需要对不同类型的实时数据进行处理统一转换为成二进制。

Node. js 提供了一个全局构造函数 Buffer,可以对二进制数据进行操作,实现 string、int、Json 等格式的转换。具体接口和说明如表 1 所示。

表 1　WebSocket 服务器数据处理接口

接口名称	说明
isBuffer(obj)	判断 obj 是否是 buffer 对象
byteLength(string, [encoding])	判断 string 转为 buffer 的长度
concat(list, [totalLength])	list 是一个数组,将几个 buffer 合为一个
IsEncoding(encoding)	判断当前是否是一个有效的编码格式

3.3　缓存数据库

WebSocket 服务器接收客户端发送的消息体时,能够获取客户端和 WebSocket 服务器之间的连接信息 connection。同时客户端向 WebSocket 服务器发送的是满足一定格式的消息体,WebSocket 服务器通过解析消息体,可以获取到{“appID” + “msg”}信息。客户端可能发送多个消息体,WebSocket 服务器会将客户端发送的消息体进行拆分,以{“appid” + “msg”}为 key,以订阅该 key 的所有 connection 的集合 connections 为 value 存储到 Redis 中。例如,一个用户发送了多个消息体,通过解析得到信息{“appID1” + “UserOneGetMsg1”}、{“appID1” + “UserOneGetMsg2”}、{“appID2” + “UserOneGetMsg1”}。Redis 首先会判断 Key 集合中是否存在这些信息,假如通过判断匹配到了{“appID1” + “UserOneGetMsg1”},就把该用户的 connection 添加到对应的集合 connections 中;假如通过判断没有匹配到{“appID1” + “UserOneGetMsg2”},则会新建一个对应的集合 connections,然后把用户连接信息 connection 再添加到新建集合中。消息体的解析过程以及生成的新数据在 Redis 中的存储流程如图 4 所示。

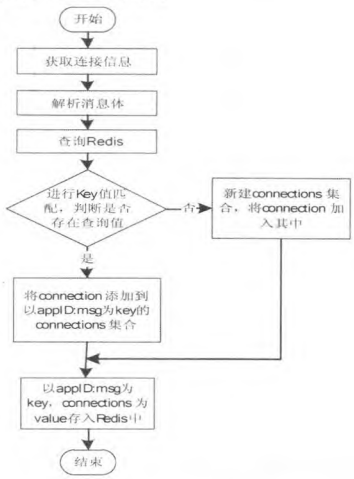

图 4　Redis 中的存储流程图

3.4　消息队列

RabbitMQ 是 AMQP(高级消息队列)协议的一个开源实现,作为老牌的消息中间件不仅功能完善、性能稳定,还有很多监听插件可以选择,提供了丰富的 API。本文选用 RabbitMQ 作为 WebSocket 服务器的消息队列中间件,以期达到以下两个方面的效果:1)减少程序中队列消费不及时造成的内存泄露问题。2)提升服务器的性能。

在 Node. js 中提供了一个 amqp 模块,该模块为所有遵循 AMQP 协议的消息队列服务器提供 Node. js 环境下的客户端。具体的实现步骤:1)安装 RabbitMQ 消息服务器,进行环境变量和权限的设置。2)用浏览器打开 http://localhost:15672,检测是否能够访问 RabbitMQ 管理控制台。3)安装 Node. js 与 RabbitMQ 交互所需要的 node-amqp 模块。4)通过 amqp. createConnection( )函数建立连接,生成队列并进行队列初始化。

3.5　消息的推送

WebSocket 服务器提供了两个消息推送接口,具体接口和说明如表 2 所示,它们分别支持文本类型和二进制类型的数据推送。

表 2　WebSocket 服务器数据推送接口

接口名称	说明
sendTest (data,callback)	当 data 数据类型为文本类型时,调用该接口,data 为要发送的数据信息,callback 为回调函数
sendBinary (data,callback)	当 data 数据类型为二进制时,调用该接口,data 为要发送的数据信息,callback 为回调函数

WebSocket 服务器从数据中心获得实时更新的数据后,执行实时数据的推送,其具体执行流程如图 5 所示。

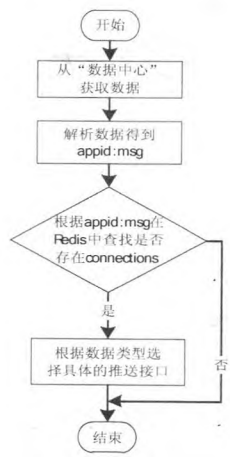

图 5　数据推送流程图

3.6 进程管理

WebSocket 服务器的 Node. js 平台虽然最初版本采用的是单进程工作模式,但在随后的发行版本中引入了多进程模式,以改善单进程对多核 CPU 利用率低、单进程崩溃等问题。进程管理模块就是对 WebSocket 服务器的多个工作子进程进行监控、维护,一旦发现有失效的工作子进程,就会创建新的工作子进程替代失效子进程继续工作,并将任务按负载均衡策略分派给不同的工作子进程处理,为 WebSocket 服务器的稳定性提供保障。在进程管理模块的实现过程中,对原有的进程管理模块进行了优化,分别增加了“心跳机制”和最小连接策略。

首先,对多进程的管理模式进行了重新设计,主进程根据配置创建若干个子进程,并将创建的子进程信息存入到“子进程队列表”中,然后通过“心跳机制”对子进程进行监控,并根据收集到的子进程信息对“子进程队列表”进行更新。若是在监控中发现有失效进程,主进程会创建新的子进程,并将已失效的子进程从“子进程队列表”中删除。监控子进程是否失效是通过“心跳机制”来实现:worker 进程定时发送心跳给 master 进程,若 master 进程在一定时间内未能收到 worker 进程的心跳,则可以认为此 worker 进程已经停止响应,否则会重启一个新的 worker 进程继续提供服务。通过引入这种监控机制,可以保障一直有工作进程响应客户请求,提升了服务器的稳定性,能够实现持续、健壮地为客户提供服务,改进后的进程管理模块如图 6 所示。

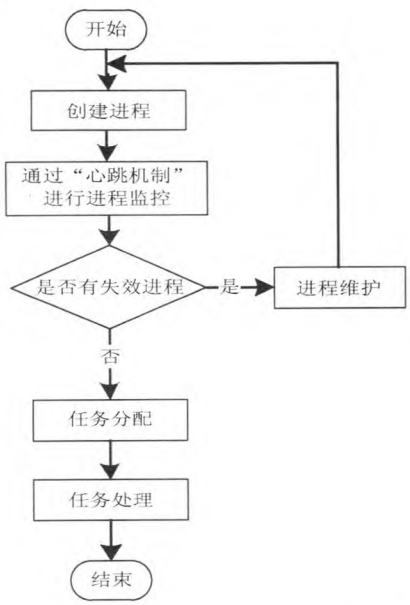

图 6 改进后的进程管理模块功能流程图

WebSocket 服务器接收到的新请求首先会被主进程监听到,然后才会分派给子进程。任务分派的方式是通过进程间的通信来实现的,在进行任务分派时,主进程会按照制定的负载均衡策略分派任务给某一个子进程处理。Node. js 原有的任务分配策略只有静态的轮转算法,本文增加了针对长连接的最少连接策略。最小连接算法解决了静态轮转算法只能保证请求数量均衡的弱点,考虑了各个工作子进程当前的负载信息,相对静态轮转算法具有较好的均衡效果。

最少连接算法原理简单易懂,在本文中主要是通过对比“子进程队列表”中每个子进程的当前连接数目,选择其中连接数最少的子进程接收新来的任务,然后在一个更新周期结束之后,对队列表中的连接数进行更新。例如一个由 n 个子进程组成的进程队列 $P=\{P(0),P(1),P(2),\cdots,P(i),\cdots,P(n-1)\}$,其中变量 $i(0\leqslant i<n)$ 表示子进程的索引,当有新连接到来时,会对 P 中所有子进程的连接数属性进行比较,返回连接数最小的子进程索引 $j(0\leqslant j<n)$,新来的任务就会分配给 $P(j)$ 处理,具体的算法流程如图 7 所示。在具体的实现过程中,为主进程提供了一个 sendHandle(worker,handle)函数,参数 worker 就是连接数最少的子进程,参数 handle 是要发送的 socket 描述符。当用户请求到来时,master 进程循环扫描所有 worker 进程,若遇到 worker 的 count 属性值为零,则直接返回该 worker 的序列号,否则继续进行循环操作,直到选出连接数最少的 worker 进程。连接数最少的 worker 进程选择出来后,就会调用 sendHandle(workers[count],handle)函数进行任务分派。一旦任务分配成功,则将该 worker 进程的连接数加 1。

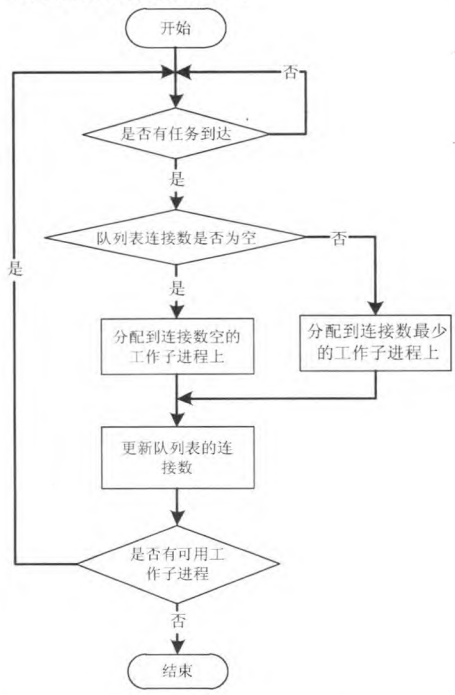

图 7 Node. js 进程管理模块最少连接算法流程图

4　实验及结果

4.1　实验环境

操作系统:64 位 Windows 10;处理器:Core i5 - 4590 @ 3.30 GHz 四核;内存:8 GB。

4.2　功能测试

将 WebSocket 服务器应用在多应用、多租户的实际场景中,由此在实验中设计了两种不同的应用,不同的应用发送不同的数据。数据由 Node.js 自身循环产生,产生的两种数据,用 Json 形式表示如下:模拟的 A 应用的数据格式为{appID:1,sensorNum:'001',data:[" + Math.round(Math.random() * 100) + "]},B 应用的数据格式为{appID:2,deviceNum:'001',lng:'" + lng + "',lat:'" + lat + "'}。原始数据如图 8 所示。

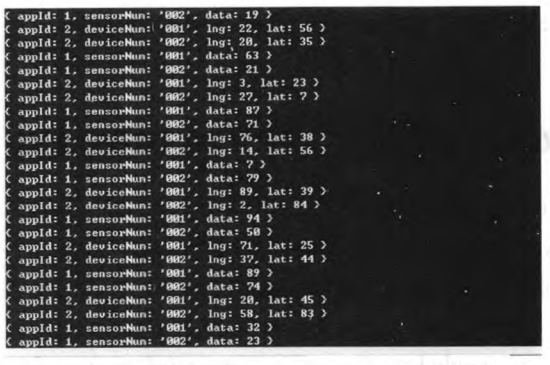

图 8　功能测试部部分原始数据包

这里产生的数据相当于 WebSocket 服务器运行中从数据中心获得的实时数据,将数据直接推送到 RabbitMQ 消息队列中。在 RabbitMQ 网页管理后台,可以查看 RabbitMQ 消息,如图 9 所示,即表示消息已经成功进入了消息队列。

图 9　RabbitMQ 消息队列后台管理

页面一打开就会发送 WebSocket 连接请求到 WebSocket 服务器,WebSocket 服务器接收到用户请求并且连接成功后,会根据用户订阅消息把数据信息实时地推送给客户端浏览器。页面的订阅按钮模拟用户发送 WebSocket 订阅请求,页面的取消按钮模拟用户发送 WebSocket 取消订阅请求,如图 10 所示。

图 10　取消订阅后重新订阅

图 10 中框出表示:用户在取消订阅后,浏览器客户端将不再接收到 sensorNum 为 002 的数据信息,而 sensorNum 为 001 的数据信息会持续接收到。通过实验也表明,WebSocket 服务器是可以达到基本功能需求的。

4.3　性能测试

WebSocket 服务器的性能测试采用 Jmeter2.0.11 作为测试工具,并安装支持 WebSocket 协议的 JMeter-WebSocketSampler - 1.0.2 - SNAPSHOT.jar 插件。安装好插件后创建一个 WebSocket 服务器测试线程组,并在线程组中添加测试线程 WebSocketTest,同时添加察看结果树、Summary-Report、聚合报告等多个监听器。

通过表 3 可知当并发连接数低于 750 时,错误率为 0;当连接数达到 1 500 时,错误率在 1.52% 左右,是可以满足在多应用多租户模式下的实时数据推送需求的。

表 3　系统性能测试

用户并发连接数	平均响应时间/ms	错误率/%
500	1 047	0
600	1 639	0
750	1 852	0
900	2 173	0.15
1 200	2 763	0.25
1 500	3 242	1.52

5　结　语

本文回顾了即时通信领域发展过程中的技术方案,进行了详细的对比分析,归纳总结了 WebSocket 的特性,并推荐在实时系统中采用 WebSocket 技术。分析了国内外 WebSocket 技术在服务器端的应用,总结出目前并没有从服务器角度去设计一款 WebSocket 服务器,用以解决在高并发、稳定性等方面可能存在的问题。研究总结了 Node.js、Redis、RabbitMQ 等开源项目

(下转第 91 页)

据的改变,以状态序列的形式描述故障的演化过程,并能够根据观测数据有效识别飞机所处运行状态趋势,成功验证了该方法的有效性。

参 考 文 献

[1] 曹惠玲,周百政. QAR 数据在航空发动机监控中的应用研究[J]. 中国民航大学学报,2010,28(3):15-19.

[2] Jiang H,Li X,Liu C. Large margin hidden Markov models for speech recognition[J]. IEEE Transactions on Audio Speech & Language Processing,2005,14(5):1584-1595.

[3] 黄晓彬,王春峰,房振明,等. 基于隐马尔科夫模型的中国股票信息探测[J]. 系统工程理论与实践,2012,32(4):713-720.

[4] 张璇,周峰. 隐马尔科夫模型应用领域、热点及趋势分析——基于 CiteSpaceII[J]. 现代商贸工业,2015,36(15):63-65.

[5] 于天剑,陈特放,陈雅婷,等. HMM 在电机轴承上的故障诊断[J]. 哈尔滨工业大学学报,2016,48(2):184-188.

[6] 柳姣姣,禹素萍,吴波,等. 基于隐马尔科夫模型的时空序列预测方法[J]. 微型机与应用,2016,35(1):74-76.

[7] 廖俊,于雷,罗寰,等. 基于趋势转折点的时间序列分段线性表示[J]. 计算机工程与应用,2010,46(30):50-53.

[8] 章登义,欧阳黜霏,吴文李. 针对时间序列多步预测的聚类隐马尔科夫模型[J]. 电子学报,2014(12):2359-2364.

[9] 杨慧,王光霞. 基于重要点的飞机发动机异常子序列检测[J]. 计算机工程与设计,2016,37(9):2543-2547.

[10] 钱江波,董逸生. 一种基于广度优先搜索邻居的聚类算法[J]. 东南大学学报(自然科学版),2004,34(1):109-112.

[11] 倪巍伟,陆介平,陈耿,等. 基于 k 均值分区的数据流离群点检测算法[J]. 计算机研究与发展,2006,43(9):1639-1643.

[12] 任江涛,何武,印鉴,等. 一种时间序列快速分段及符号化方法[J]. 计算机科学,2005,32(9):166-169.

(上接第 27 页)

的特点,设计出了 WebSocket 服务器的架构图,详细阐述了各模块的实现方案。为了进一步提高 WebSocket 服务器的高并发和稳定性,对 Node.js 多进程模块进行改进和优化。最后对 WebSocket 服务器进行实验测试,验证结果表明该 WebSocket 服务器在功能和性能上达到了预期设计的目标。在下一阶段的研究中,将会在 WebSocket 服务器中加入日志管理模块,通过日志回放服务器的运行状态。

参 考 文 献

[1] 陆晨,冯向阳,苏厚勤. HTML5 WebSocket 握手协议的研究与实现[J]. 计算机应用与软件,2015,32(1):128-131.

[2] Takagi H,Kleinrock L. Analysis of polling systems[J]. Performance Evaluation,1985,5(3):206.

[3] Loreto S,Saintandre P,Salsano S,et al. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP[J]. Choice Current Reviews for Academic Libraries,2011,4(1846):4.

[4] Loreto S,Saintandre P,Salsano S,et al. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP[J]. Choice Current Reviews for Academic Libraries,2011,4(1846):4.

[5] 许会元,何利力. NodeJS 的异步非阻塞 I/O 研究[C]//全国工业控制计算机技术年会,2014.

[6] Tilkov S,Vinoski S. Node.js:Using JavaScript to Build High-Performance Network Programs[J]. IEEE Internet Computing,2010,14(6):80-83.

[7] Pimentel V,Nickerson B G. Communicating and Displaying Real-Time Data with WebSocket[J]. IEEE Internet Computing,2012,16(4):45-53.

[8] 蔡锦达,蒋振飞. 基于 WebSocket 的印刷包装机械远程监控方法的研究[J]. 包装工程,2013(15):87-90.

[9] Ma K,Zhang W. Introducing browser-based high-frequency cloud monitoring system using WebSocket Proxy[J]. International Journal of Grid & Utility Computing,2015,6(1).

[10] 刘维峰,左泽军,赵利强,等. 基于 HTML5 的生产装置实时监测可视化[J]. 计算机工程与设计,2015(3):809-813.

[11] 金丰,张悦,雍鹏. 基于双服务器的 B/S 模式监测系统的设计与实现[J]. 计算机仿真,2014,31(2):201-205.

[12] 张玲,张翠肖. WebSocket 服务器推送技术的研究[J]. 河北省科学院学报,2014,31(2):49-53.

[13] Lei K,Ma Y,Tan Z. Performance Comparison and Evaluation of Web Development Technologies in PHP,Python,and Node.js[C]//IEEE,International Conference on Computational Science and Engineering. IEEE Computer Society,2014:661-668.

[14] Marc Fasel. Performance Comparison Between Node.js and Java EE For Reading JSON Data from CouchDB[EB/OL]. [2013-10-22]. http://blog.shinetech.com/2013/10/22/performance-comparison-between-node-js-and-java-ee.

[15] O'reilly. Node:up and running:scalable server-side code with javascript[M]. Oreilly Media,2012.

[16] 王心妍. Memcached 和 Redis 在高速缓存方面的应用[J]. 无线互联科技,2012(9):8-9.

[17] Melnikov A. The WebSocket Protocol[OL]. 2011. https://www.rfc-editor.org/rfc/rfc6455.txt.