**消息系统该Push/Pull模式分析**

* 参考：<http://blog.csdn.net/pi9nc/article/details/27714745>

    **信息推拉技术简介**

     “智能信息推拉(IIPP)技术”是在网上信息获取技术中加入了智能成份，从而有助于用户在海量信息中高效、及时地获取最新信息，提高了信息系统主动信息服务的能力。如果引入基于IIPP的主动信息服务系统，则可根据用户的特性提供具有针对性的、个性化的信息服务。

     以往在Internet上搜寻信息，最常用的方法就是浏览器发出请求后，Web就将信息传送给用户，此过程用户需要“拉取”信息而被描述为Pull；而将信息直接“推送”到用户的计算机的方法就是信息推送，称之为Push，用户只需要在初次使用时自己设定所需要的信息频道，此后，定制信息将通过Web自动传给用户。

**信息推拉技术智能化**

    在传统的Client/server结构中，信息获取方式是按“拉”(Pull)的模型进行的：服务器根据用户终端发送的服务请求进行处理并返回用户所需的结果。在Push系统中，服务器把信息“推”给用户终端系统。虽然两者数据传输的方向都是从服务器流向用户，但操作的发起者是不同的。从“信源”与“用户”的关系来看，信息的流动可分为两种模式，即信息推送与信息拉取模式。

**Push与Pull之比较**

    推送(Push)技术是根据用户需要，有目的、按时将用户感兴趣的信息主动发送到用户的计算机中。Push技术的主要优点是对用户要求低，普遍适用于广大公众，不要求有专门的技术；二是及时性好，信源及时地向用户“推送”不断更新的动态信息。但是，在随后实际应用中，因为存在以下几方面不足，Push技术并没有取得预期的成功：

 不能确保发送成功。由于Push技术采用广播方式，当网络信息中心发送信息时，只有接收器打开并正好切换到同一频道上，传输才能发生作用，用户才能获取信息。这对于那些要确保能收到信息的应用领域是不太适合的。

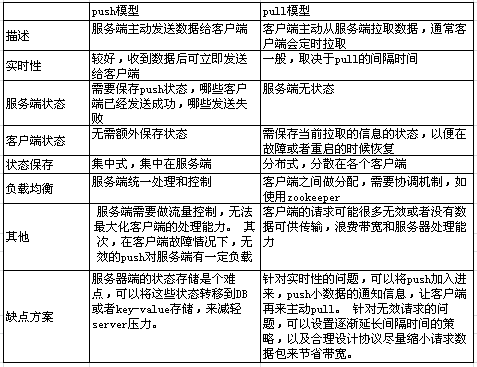
 没有信息状态跟踪。Push技术采用的是“开环控制”模式，一个信息发布以后的状态，如用户是否接收，或客户端收到后是否按信息的提示执行了任务等，这些“反馈信息”发布者无从得知。

 针对性差。推送的信息内容缺乏针对性，不能满足用户的个性化需求。有价值的重要信息，通常都是要针对一些特定的群组来发送的，即只送给相关的人士。Push技术不能满足上述需求。

 信源任务重。信源系统要主动地、快速地、不断地将大量信息推送给用户。

    拉取(Pull)技术指用户有目的地在网络上主动查询信息，用户从浏览器给Web发出请求，由Web获取所需信息。面对拥有海量信息的Internet环境，搜索引擎是有效的网络信息“拉取”(查询)的检索工具。Pull技术的主要优点是针对性强，能满足用户的个性化需求；信息传输量小，网络上所传输的只是用户的请求和服务器针对该请求所作的响应；信源任务轻，信息系统只是被动接受查询，提供用户所需的部分信息。其主要缺点是及时性差，由于用户只会基于自己的知识水平(或专业水平)提出请求，当信源中信息更新变化时，用户难以及时拉取新的动态信息，虽然可以通过定时查询来解决这个问题，但是会浪费大量的网络资源和人力，而且，仍不能保证最好的实时性。对用户要求高，要求用户对信源系统有相应的专业知识，掌握相关的检索技术。

**PUSH和PULL模型对比**



**PUSH和PULL两种模式结合**

 将信息推送与拉取两种模式结合能做到取长补短，使二者优势互补。根据推、拉结合顺序及结合方式的差异，又分以下四种不同推拉模式：

 先推后拉——先由信源及时推送公共信息，再由用户有针对性地拉取个性化信息；

 先拉后推——根据用户拉取的信息，信源进一步主动提供(推送)与之相关的信息；

 推中有拉——在信息推送过程中，允许用户随时中断并定格在感兴趣的网页上，以拉取更有针对性的信息；

 拉中有推——根据用户搜索(即拉取)过程中所用的关键字，信源主动推送相关的最新信息。

    Related posts:

 [IIS日志配置方案](http://blogread.cn/'http:/www.biaodianfu.com/iis-log.html/')

    无论是消息系统，还是配置管理中心，甚至存储系统，你都要面临这样一个选择，push模型 or pull模型?是服务端主动给客户端推送数据，还是客户端去服务器拉数据，一张图表对比如下：

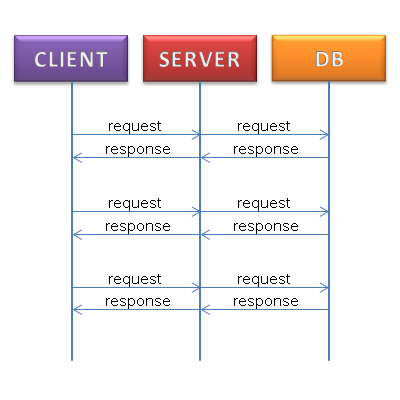
|  |  |  |
| --- | --- | --- |
|  | push模型 | pull模型 |
| 描述 | 服务端主动发送数据给客户端 | 客户端主动从服务端拉取数据，通常客户端会定时拉取 |
| 实时性 | 较好，收到数据后可立即发送给客户端 | 一般，取决于pull的间隔时间 |
| 服务端状态 | 需要保存push状态，哪些客户端已经发送成功，哪些发送失败 | 服务端无状态 |
| 客户端状态 | 无需额外保存状态 | 需保存当前拉取的信息的状态，以便在故障或者重启的时候恢复 |
| 状态保存 | 集中式，集中在服务端 | 分布式，分散在各个客户端 |
| 负载均衡 | 服务端统一处理和控制 | 客户端之间做分配，需要协调机制，如使用zookeeper |
| 其他 | 服务端需要做流量控制，无法最大化客户端的处理能力。  其次，在客户端故障情况下，无效的push对服务端有一定负载。 | 客户端的请求可能很多无效或者没有数据可供传输，浪费带宽和服务器处理能力 |
| 缺点方案 | 服务器端的状态存储是个难点，可以将这些状态转移到DB或者key-value存储，来减轻server压力。 | 针对实时性的问题，可以将push加入进来，push小数据的通知信息，让客户端再来主动pull。      针对无效请求的问题，可以设置逐渐延长间隔时间的策略，以及合理设计协议尽量缩小请求数据包来节省带宽。 |

    在面对大量甚至海量客户端的时候，使用push模型，保存大量的状态信息是个沉重的负担，加上复制N份数据分发的压力，也会使得实时性这唯一的优点也被放小。使用pull模型，通过将客户端状态保存在客户端，大大减轻了服务器端压力，通过客户端自身做流量控制也更容易，更能发挥客户端的处理能力，但是需要面对如何在这些客户端之间做协调的难题。

    客户端和服务端的交互有推和拉两种方式：如果是客户端拉的话，通常就是Polling；如果是服务端[推](http://en.wikipedia.org/wiki/Push_technology)的话，一般就是[Comet](http://en.wikipedia.org/wiki/Comet_(programming))，目前比较流行的Comet实现方式是Long Polling。

    注：如果不清楚相关名词含义，可以参考：[Browser 與 Server 持續同步的作法介紹](http://josephj.com/entry.php?id=358)。

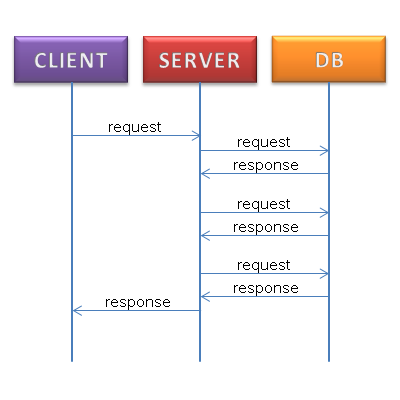
    先来看看Polling，它其实就是我们平常所说的轮询，大致如下所示：



Polling

    因为服务端不会主动告诉客户端它是否有新数据，所以Polling的实时性较差。虽然可以通过加快轮询频率的方式来缓解这个问题，但相应付出的代价也不小：一来会使负载居高不下，二来也会让带宽捉襟见肘。

    再来说说Long Polling，如果使用传统的LAMP技术去实现的话，大致如下所示：

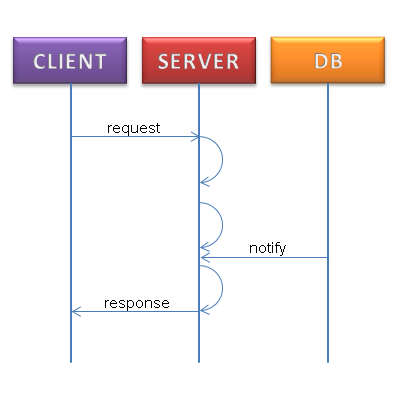


Long Polling

    客户端不会频繁的轮询服务端，而是对服务端发起一个长连接，服务端通过轮询数据库来确定是否有新数据，一旦发现新数据便给客户端发出响应，这次交互便结束了。客户端处理好新数据后再重新发起一个长连接，如此周而复始。

    在上面这个Long Polling方案里，我们解决了Polling中客户端轮询造成的负载和带宽的问题，但是依然存在服务端轮询，数据库的压力可想而知，此时我们虽然可以通过针对数据库使用主从复制，分片等技术来缓解问题，但那毕竟只是治标不治本。

    我们的目标是实现一个简单的服务端推方案，但简单绝对不意味着简陋，轮询数据库是不可以接受的，下面我们来看看如何解决这个问题。在这里我们放弃了传统的LAMP技术，转而使用[Nginx与Lua](http://huoding.com/2012/08/31/156)来实现。



Modified Long Polling

    此方案的主要思路是这样的：使用Nginx作为服务端，通过Lua协程来创建长连接，一旦数据库里有新数据，它便主动通知Nginx，并把相应的标识(比如一个自增的整数ID)保存在Nginx共享内存中，接下来，Nginx不会再去轮询数据库，而是改为轮询本地的共享内存，通过比对标识来判断是否有新消息，如果有便给客户端发出响应。

    注：服务端维持大量长连接时内核参数的调整请参考：[http长连接200万尝试及调优](http://blog.lifeibo.com/?p=269)。

    首先，我们简单写一点代码实现轮询(篇幅所限省略了查询数据库的操作)：

lua\_shared\_dict config 1m;

server {

location /push {

local id = 0;

local ttl = 100;

local now = ngx.time();

local config = ngx.shared.config;

if not config:get("id") then

config:set("id", "0");

end

while id >= tonumber(config:get("id")) do

local ttl = math.random(ttl - 10, ttl + 10);

if ngx.time() - now > ttl then

break;

end

ngx.sleep(1);

end

ngx.say("OK");

}

...

}

    注：为了解决服务端不知道客户端何时断开连接的情况，代码中引入超时机制。

    其次，我们需要做一些基础工作，以便操作Nginx的共享内存：

lua\_shared\_dict config 1m;

server {

location /config {

content\_by\_lua '

local config = ngx.shared.config;

if ngx.var.request\_method == "GET" then

local field = ngx.var.arg\_field;

if not field then

ngx.exit(ngx.HTTP\_BAD\_REQUEST);

end

local content = config:get(field);

if not content then

ngx.exit(ngx.HTTP\_BAD\_REQUEST);

end

ngx.say(content);

ngx.exit(ngx.HTTP\_OK);

end

if ngx.var.request\_method == "POST" then

ngx.req.read\_body();

local args = ngx.req.get\_post\_args();

for field, value in pairs(args) do

if type(value) ~= "table" then

config:set(field, value);

end

end

ngx.say("OK");

ngx.exit(ngx.HTTP\_OK);

end

';

}

...

}

    如果要写Nginx共享内存的话，可以这样操作：

shell> curl -d id=123 http:///config

    如果要读Nginx共享内存的话，可以这样操作：

shell> curl http:///config?field=id

    注：实际应用时，应该加上权限判断逻辑，比如只有限定的IP地址才能使用此功能。

    当数据库有新数据的时候，可以通过触发器来写Nginx共享内存，当然，在应用层通过观察者模式来写Nginx共享内存通常会是一个更优雅的选择。

    如此一来，数据库就彻底翻身做主人了，虽然系统仍然存在轮询，但已经从轮询别人变成了轮询自己，效率不可相提并论，相应的，我们可以加快轮询的频率而不会造成太大的压力，从而在根本上提升用户体验。

    突然想起另一个有趣的服务端推的做法，不妨在一起唠唠：如果DB使用Redis的话，那么可以利用其提供的[BLPOP](http://redis.io/commands/blpop)方法来实现服务端推，这样的话，连sleep都不用了，不过有一点需要注意的是，一旦使用了BLPOP方法，那么Nginx和Redis之间的连接便会一直保持下去，从Redis的角度看，Nginx是客户端，而客户端的可用端口数量是有限的，这就意味着一台Nginx至多只能建立五六万个连接(net.ipv4.ip\_local\_port\_range)，有点儿少。

    …

    当然，本文的描述只是沧海一粟，还有很多技术可供选择，比如[Pub/Sub](http://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern)，[WebSocket](http://en.wikipedia.org/wiki/WebSocket)等等，篇幅所限，这里就不多说了，有兴趣的读者请自己查阅。