搜狗实验室技术交流文档

Vol2:2 用java的nio技术实现的异步连接池

摘要

本文重点讲解异步连接池的诞生背景和使用方法，同时介绍 java nio 技术的基础知识。

连接池概述

编写 Web 应用程序时，为了提高性能，往往采取常连接和连接池技术。

所谓常连接，就是一个持久存在的 TCP 连接，连接完成后，可以反复使用，直到连接关闭，或出现通信问题。采用常连接是为了避免两方面的问题：

1. Socket 资源属于系统资源，申请操作需要陷入到内核才能完成，反复申请的话，必然降低程序性能；
2. tcp 连接需要完成连接操作后才能使用，而 tcp 三次握手是个停-等过程，效率很低。

连接池技术是常连接的管理方式。通常情况下，一个 TCP 连接同时只能处理一个请求，一次完整的收发请求操作是不能被其他线程中断的。连接池管理多个连接的复用，保证互斥。常用的连接池算法不需要额外线程参与，算法如下：

1. 从池中取连接；
2. 如果没有空闲链接，陷入 waiting，被唤醒后转 1，否则继续；
3. 发送请求；
4. 接收结果；
5. 将连接放回池中，唤醒在 Step 2 waiting 的其它线程；
6. 结束

其中步骤 1 和步骤 5 需要加锁互斥。

常见连接池的实现中，上述过程是连续的，处理线程不会中途跑开去做别的事情。连接池的这个特性和 Socket 类型有密切的关系，因为，常用 Socket 的读写操作都是阻塞式的，就是说每次 read/write 掉用，在数据被读入/写出前，调用线程都处于阻塞的状态。这样，一方面，连接池中的临界资源—Socket，需要被及时放回到池中；另一方面，使用阻塞式 Socket 的情况下，处理线程没办法知道什么时候响应数据会就位。在这种前提下，处理线程只能是发送完成收马上收取，直到响应结束。

多个后台服务

如果应用程序只需要访问一个后台服务，而且程序的主要功能就是通过连接池所连的后台服务器，完成特定的查询功能的话，上述的“传统连接池”是完全满足要求的，性能也很好。但是，当程序需要访问多个后台服务时，就凸显出来新的问题。

以网页搜索的 FrontWeb 模块为例，它需要连接 Qc，Hint，Cache，TinySearch 等多种服务，对于每次用户查询请求，FrontWeb 要从各个服务器取得查询结果后才能返回搜索页面。

很显然，这里需要多个连接池进行处理，对每个连接池的处理，必须串行完成。即 FrontWeb 必须先向连接池 A 发送请求，等待结果返回后再向连接池 B 请求，依次完成收发操作。这种串联的结构，性能又低，健壮性也不好，任何一种服务出现故障，都会严重影响 FrontWeb 的响应时间。

容易想到，如果把串行改成并行，并发的向多个后台服务发起请求，并收集结果，那么性能和健壮性都会大大改善。但是如果仍然采用阻塞式 I/O，实现的代价会比较大。在阻塞式 I/O 下，每个连接都必须有个专职线程负责执行 I/O 操作。这就意味着，对于 N 个后台服务，每次用户请求，都必须有至少 N 个“专职线程”为它服务。这样系统的能达到的并发度只有最大允许线程数的 1/N，这是不可忍受的。

导致连接池必须串行操作的罪魁祸首就是阻塞式 I/O，如果采用非阻塞式 I/O 的 Socket 调用，可以使单个线程处理多个 tcp 连接，这使我们看到了解决问题的希望。幸运的是，1.4.2 版本以后，java 有了 java.nio 类库的非阻塞 I/O API，这正应了一句古话：有了米，就可以做饭了。

java.nio

在介绍 nio 以前，请先温习一下 A 组通讯第一期中，关于阻塞和非阻塞 I/O 的描述，这里就不再重复了。

Java.nio 包中主要包括下边这些类：

- SelectableChannel
- Buffer、Charset、ByteOrder
- Selector、SelectionKey

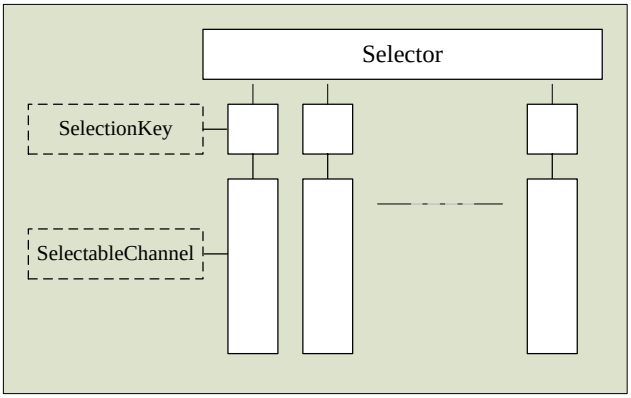
Nio 把它支持的 I/O 对象抽象为 Channel，目前已知的实例类有：SocketChannel、ServerSocketChannel、DatagramChannel、FileChannel 等，从名字大家就能猜出它们是干什么用的。

Buffer 是一种工具类，作为 Channel 收发数据的载体出现。它通过一种 flip 机制，实现了方便的数据读写功能，简单说来，就是每个 Buffer 对象有两个下标变量，分别标识当前的读写位置，使用者只要直接掉用 put 和 get 方法就能实现顺序读写。

Charset 类与 Buffer 类配合，提供了高效的编解码方式，通过简单的 encode/decode 操作就能将数据从字节流编码为字符流，或者从字符流解码为字节流。

ByteOrder 解决了字节序的问题，通过调用 Bufer 类的 order()方法设置一次字节序后，就可以直接调用 Buffer 类的 putInt()、putLong()等方法，免去了 C 语言中 hton() 的烦恼。

上述几种类都是砖瓦，nio 的主体是 Selector，SelectionKey 则是水泥，非阻塞式 I/O 中的 readiness notification 功能，就是通过它来实现的，下边是一个简图：



所有的 Channel 正式使用前都要注册到 Selector 中，运行过程中，管理 Selector 的线程每次调用它的 select()方法时，该方法会把所有 ready 的 SelectionKey 挑出来（或者说 ready 的 channel）。

Selector 是一个内部结构比较复杂的类，且多数 API 没有内部的互斥机制，因此，Selector 必须由一个单独的线程来维护。事实上，写 nio 程序时，处理 Selector 和处理线程间的互斥会花费很大的精力。

现在我们对 nio 有了一个初步的了解，下边，就可以看如何应用这些类来实现异步连接池了。

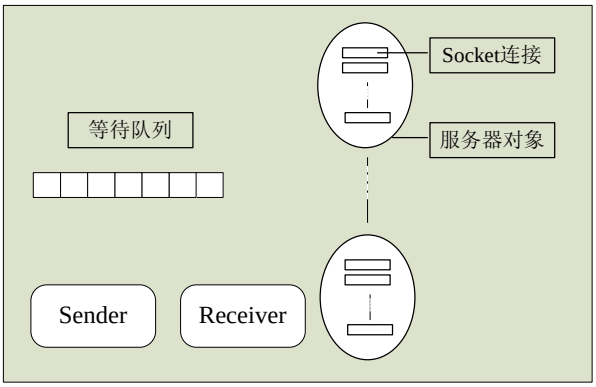
异步连接池

利用 java nio，我们设计了一个高性能的异步连接池。在我的心目中，异步连接池的工作过程应该是这样的：

首先，用户处理线程调用连接池对象的某个方法（比如 sendRequest），把一个能够标识本次请求的 Request 对象扔给连接池。

之后用户处理线程可以去做别的事情，比如，向其他连接池发送请求。

最后当用户线程处理完能做的业务逻辑后，就可以等待连接池返回结果了。



这样的结构下，异步连接池必须提供如下功能：

1. 维护 Selector，Channel 的生命周期。这个是显而易见的。
2. 非阻塞式发送。这里明显需要有个队列，在连接资源耗尽的情况下，缓存一下请求。

3. 完成通知。请求发送后，用户处理线程和该次查询之间唯一的联系就是 Request 对象了，需要在该对象中实现一种 wait/notify 机制，让用户处理线程及时得到通知。

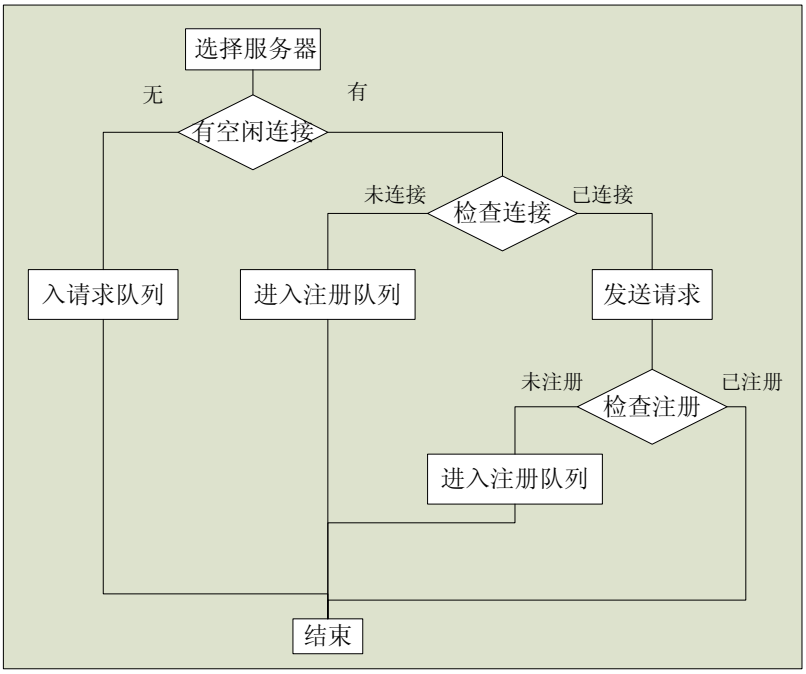
Receiver 线程维护 Selector 以及 SocketChannel，完成 socket 的连接，发送以及通知用户线程的工作，同时负责检查响应超时的请求。

Sender 线程维护等待队列，等有空闲连接时及时完成发送动作，同时负责检查排队超时的请求。

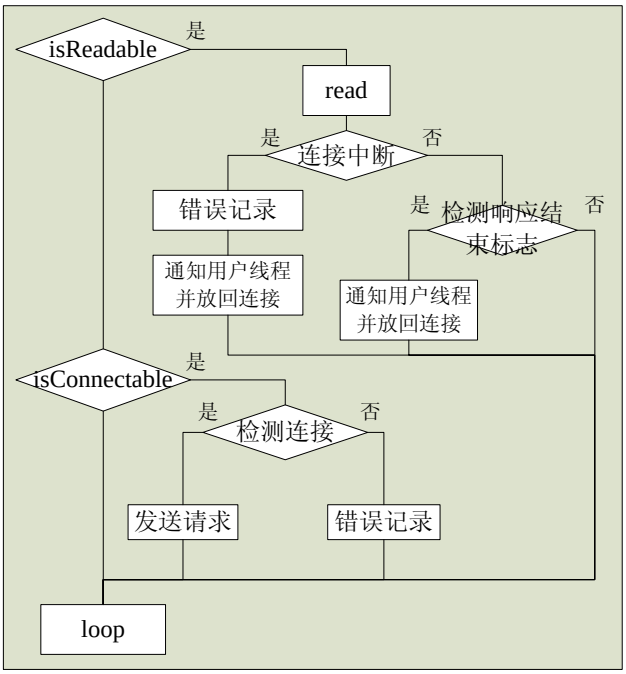
服务器对象抽象为 ServerStatus 类，用一定的数据结构（链表和栈）将 Socket 连接对象组织起来。

Socket 连接对象抽象为 AsyncGenericQueryClient 类，收发数据的逻辑在这个类里实现。

发送过程的逻辑如下图：



Receiver 的工作过程如下图:



关于连接池的内部详细实现机制，这里略过，不过大家一定能想得到，一系列的锁、链表……，有兴趣的同学可参考源码。

事实上，连接池还有个特性，大家可能会喜欢——它是一个类库，你通过比较小的工作量，就能够实例化一个自己的连接池出来。

今后的改进方向：

1. 把 Sender 和 Receiver 的逻辑划分清楚；

现在的 Receiver 线程逻辑不是很清楚，也承担一部分发送请求的工作。今后可能需要把它理清楚一些。等待连接建立发送请求的操作由 Sender 线程自己的 selector 来负责或者把两个线程合一，承担所有的网络 I/O 操作。

2. 使 Sender 能够支持 non-blocking 的发送请求；

目前发送请求的时候实际上是阻塞形式的发送，这样不适应慢速网络，今后要改成 non-blocking 的形式，以支持慢速网络。

异步连接池的用法

异步连接池 lib 主要由以下几个类组成，在 com.sohu.common.connectionpool.async 包中就能找到：

- ***AsyncGenericQueryClient(虚类)***
  - ***handleInput()***
  - ***sendRequest()***
  - ***finishResponse()***
  - ***reset()***
- ***AsyncRequest(虚类)***
  - ***getServerid(int)***
  - ***isValid()***
- ***AsyncClientFactory(接口)***

- ***newInstance()***
- ***AsyncGenericConnectionPool(虚类)***
  - ***构造函数***
  - ***sendRequest(AsyncRequest) (可选)***

不同连接池之间最大的区别是通讯协议，所以异步连接池类库重点实现的连接的管理逻辑，涉及到收发协议的部分，都采用虚函数的方式，由开发人员在实例化的时候自主完成。

开发人员需要实现自己的 request、queryclient。然后继承 pool 类，简单把自己的 request/client 和 pool 绑定起来即可

AsyncGenericQueryClient

前边已经提到，具体的数据收发操作被封装到了 AsyncGenericQueryClient 类中，它有四个主要的虚函数：

1. handleInput()
   由 Receiver 回调，当 Receiver 发现对应的 SocketChannel 有数据可读的时候，就会调用该函数完成数据收取操作。这样做是因为，异步连接池并不关心接收缓冲区如何组织，你可以用多个 Buffer 来组织接收数据，实现的时候根据自己的意愿把数据填充到不同的 Buffer 中，也可以乘此机会做一些其他的工作，只要保证该函数尽可能快的完成就是了。
2. sendRequest()
   回调函数，可能由 Sender，Receiver 或者用户处理线程调用。该函数完成数据的发送逻辑。
3. finishResponse()
   回调函数，由 Receiver 调用。询问该 Client，数据接收工作是否已经完成，即是否已经接收了一组完整的响应数据。该函数完成两个工作：一是确认数据已经接收完成，二是将对数据进行处理，因为缓冲区要腾出来供下次请求使用，数据的解析工作要放在这里执行。
4. reset()
   回调函数，由 Receiver 调用。Client 类是个状态相关的类，具体实例化的时候，你可能会设置一些状态数据，这里允许你对状态数据清零，以备下次使用。

在实现的时候请注意异常的捕获，对于 handleInput，sendRequest 两个函数，只允许抛出 IOException，而且该 Exception 必须是由于读写 SocketChannel 引起的。这这是因为，Receiver 调用这两个函数过程中，如果跑出了 IOException，就会将连接关闭。其他类型的 Exception，Receiver 一概不处理，一旦抛出，Receiver 一定会退出，连接池也一定不能正常工作了。

AscynRequest 虚类

该类用于封装请求数据，保存请求结果，记录查询时间，以及实现前文提到的“结果通知哦”功能。后三个功能是连接池应该提供的功能，所以已经实现好了，使用者通过 getQueueTime()，getTime()方法，就能得到请求的排队时间和服务器处理时间；通过掉用 getResult(long)方法，就能阻塞式获取查询结果。

注意，getResult 方法第一次被掉用时是阻塞式的，参数表示最长等待时间，如果在等待时间内结果已经就绪，或者该方法被调用前结果已经就绪，这个方法会立即返回，返回值就是经 AsyncGenericQueryClient 的 handleInput 方法处理后得到的结果对象。由于连接池不关心结果对象的类型，返回值是 Object 类型的，请自己进行类型转换。

这里也有连个虚函数：getServerId(int)和 isValid()：

1. getServerId(int)
   在用户处理线程掉用线程池的发送方法时被掉用，该函数用于实现分环策略。参数值是当前可用的服务器数量。
2. isValid()
   在请求发送前，允许做一次自我检测，这样可以避免非法请求跟合法请求抢资源。当然了，你的实现中可以令返回值恒为 true，这个是没有关系的，调用 AsyncGenericQueryClient 的 sendRequest 方法还有一次检查的机会。

AsyncClientFacotry

连接池采用 Factory 模式，你在实现 newInstance()方法时，new 一个 AsyncGenericQueryClinet 子类就可以了。

AsyncGenericConnectionPool

该类是连接池对象最外层的类，其中有一系列的 set/get 方法，用于设置连接池的参数，详情可参考源码。由于这是个虚类，且显示指定了构造函数，所以你自己的连接池一定要继承该类，并写一个相应构造函数。

到这里，你的连接池就算完成了，不过这里还有个可选的 sendRequest 方法，它并不是一个虚函数，但是你可以重写它，因为原来的方法实现了一种错误恢复功能，即当某个服务器暂时不可用时，就另外选择一个作为替代。这种功能可能对你的连接池并不适用，那你就得自己写一个了。