# 实战-实现自己的通信框架

## 通信框架功能设计

### 功能描述

通信框架承载了业务内部各模块之间的消息交互和服务调用，它的主要功能如下：

基于 Netty 的 NIO 通信框架，提供高性能的异步通信能力；

提供消息的编解码框架，可以实现 POJO 的序列化和反序列化；

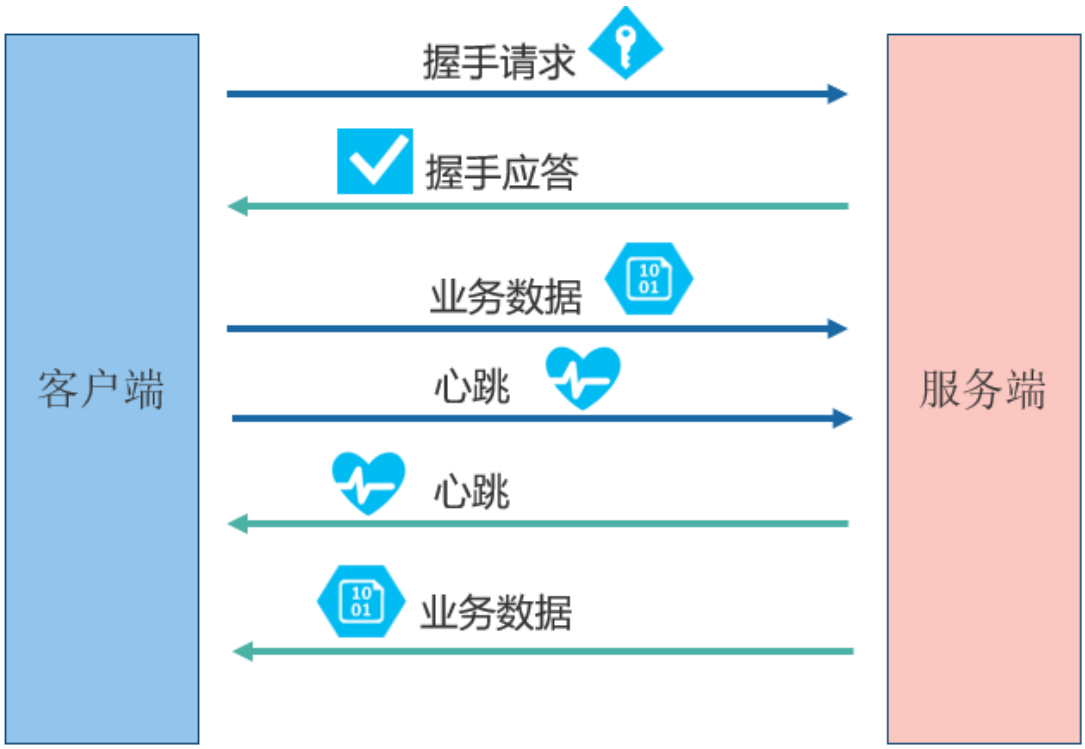
消息内容的防篡改机制

提供基于 IP 地址的白名单接入认证机制；

链路的有效性校验机制；

链路的断连重连机制

### 通信模型




（1）客户端发送应用握手请求消息，携带节点 ID 等有效身份认证信息；

（2）服务端对应用握手请求消息进行合法性校验，包括节点 ID 有效性校验、节点重复登录校验和 IP 地址合法性校验，校验通过后，返回登录成功的应用握手应答消息；

（3）链路建立成功之后，客户端发送业务消息；

（4）链路成功之后，服务端发送心跳消息；

（5）链路建立成功之后，客户端发送心跳消息；

（6）链路建立成功之后，服务端发送业务消息；

（7）服务端退出时，服务端关闭连接，客户端感知对方关闭连接后，被动关闭客户端连接。

备注：需要指出的是，协议通信双方链路建立成功之后，双方可以进行全双工通信，无论客户端还是服务端，都可以主动发送请求消息给对方，通信方式可以是 TWO WAY 或者 ONE WAY。双方之间的心跳采用 Ping-Pong 机制，当链路处于空闲状态时，客户端主动发送 Ping 消息给服务端，服务端接收到 Ping 消息后发送应答消息 Pong 给客户端，如果客户端连续发送 N 条 Ping 消息都没有接收到服务端返回的 Pong 消息，说明链路已经挂死或者对方处于异常状态，客户端主动关闭连接，间隔周期 T 后发起重连操作，直到重连成功。

## 消息定义

消息定义包含两部分：

消息头；消息体。

在消息的定义上，因为是同步处理模式，不考虑应答消息需要填入请求消息 ID，所以消息头中只有一个消息的 ID。如果要支持异步模式，则请求消息头和应答消息头最好分开设计，应答消息头中除了包括本消息的 ID 外，还应该包括请求消息 ID，以方便请求消息的发送方根据请求消息 ID 做对应的业务处理。

消息体则支持 Java 对象类型的消息内容。

Netty 消息定义表

| 名称 | 类型 | 长度 | 描述 |
| --- | --- | --- | --- |
| header | Header | 变长 | 消息头定义 |
| body | Object | 变长 | 消息的内容 |

消息头定义（Header）

| 名称 | 类型 | 长度 | 描述 |
| --- | --- | --- | --- |
| md5 | String | 变长 | 消息体摘要，缺省 MD5 摘要 |
| msgID | Long | 64 | 消息的 ID |
| Type | Byte | 8 | 0:业务请求消息<br>1：业务响应消息 |

| | | | |
|---|---|---|---|
| | | | 2：业务 one way 消息<br>3：握手请求消息<br>4：握手应答消息<br>5：心跳请求消息<br>6：心跳应答消息 |
| Priority | Byte | 8 | 消息优先级：0~255 |
| Attachment | Map<String,Object> | 变长 | 可选字段，用于扩展消息头 |

## 链路的建立

客户端的说明如下：如果 A 节点需要调用 B 节点的服务，但是 A 和 B 之间还没有建立物理链路，则有调用方主动发起连接，此时，调用方为客户端，被调用方为服务端。

考虑到安全，链路建立需要通过基于 Ip 地址或者号段的黑白名单安全认证机制，作为样例，本协议使用基于 IP 地址的安全认证，如果有多个 Ip，通过逗号进行分割。在实际的商用项目中，安全认证机制会更加严格，例如通过密钥对用户名和密码进行安全认证。

客户端与服务端链路建立成功之后，由客户端发送业务握手请求的认证消息，服务端接收到客户端的握手请求消息之后，如果 IP 校验通过，返回握手成功应答消息给客户端，应用层链路建立成功。握手应答消息中消息体为 byte 类型的结果，0：认证成功；-1 认证失败；服务端关闭连接。

链路建立成功之后，客户端和服务端就可以互相发送业务消息了，在客户端和服务端的消息通信过程中，业务消息体的内容需要通过 MD5 进行摘要防篡改。

## 可靠性设计

### *心跳机制*

在凌晨等业务低谷时段，如果发生网络闪断、连接被 Hang 住等问题时，由于没有业务消息，应用程序很难发现。到了白天业务高峰期时，会发生大量的网络通信失败，严重的会导致一段时间进程内无法处理业务消息。为了解决这个问题，在网络空闲时采用心跳机制来检测链路的互通性，一旦发现网络故障，立即关闭链路，主动重连。

当读或者写心跳消息发生 I/O 异常的时候，说明已经中断，此时需要立即关闭连接，如果是客户端，需要重新发起连接。如果是服务端，需要清空缓存的半包信息，等到客户端重连。

### 空闲的连接和超时

检测空闲连接以及超时对于及时释放资源来说是至关重要的。由于这是一项常见的任务，Netty 特地为它提供了几个 ChannelHandler 实现。

IdleStateHandler 当连接空闲时间太长时，将会触发一个 IdleStateEvent 事件。然后，可以通过在 ChannelInboundHandler 中重写 userEventTriggered()方法来处理该 IdleStateEvent 事件。

ReadTimeoutHandler 如果在指定的时间间隔内没有收到任何的入站数据，则抛出一个 ReadTimeoutException 并关闭对应的 Channel。可以通过重写你的 ChannelHandler 中的 exceptionCaught()方法来检测该 Read-TimeoutException。

## *重连机制*

如果链路中断，等到 INTEVAL 时间后，由客户端发起重连操作，如果重连失败，间隔周期 INTERVAL 后再次发起重连，直到重连成功。

为了保持服务端能够有充足的时间释放句柄资源，在首次断连时客户端需要等待 INTERVAL 时间之后再发起重连，而不是失败后立即重连。

为了保证句柄资源能够及时释放，无论什么场景下重连失败，客户端必须保证自身的资源被及时释放，包括但不现居 SocketChannel、Socket 等。

重连失败后，可以打印异常堆栈信息，方便后续的问题定位。

## *重复登录保护*

当客户端握手成功之后，在链路处于正常状态下，不允许客户端重复登录，以防止客户端在异常状态下反复重连导致句柄资源被耗尽。

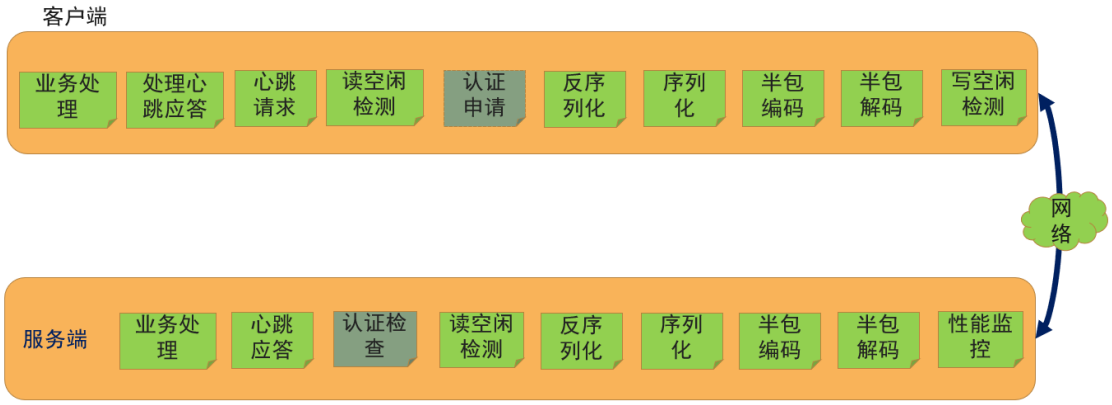
服务端接收到客户端的握手请求消息之后，对 IP 地址进行合法性校验，如果校验成功，在缓存的地址表中查看客户端是否已经登录，如果登录，则拒绝重复登录，同时关闭 TCP 链路，并在服务端的日志中打印握手失败的原因。

客户端接收到握手失败的应答消息之后，关闭客户端的 TCP 连接，等待 INTERVAL 时间之后，再次发起 TCP 连接，直到认证成功。

# 实现

参考 netty-adv 模块下的代码

完成后 Handler 示意图如下：




其中认证申请和认证检查可以在完成后移除。

## 前期准备

cn.tuling.nettyadv.vo 中定义了消息有关的实体类，为了防篡改，消息体需要进行摘要，vo 包下提供了 EncryptUtils 类，可以对消息体进行摘要，目前支持 MD5、SHA-1 和 SHA-256 这三种，缺省为 MD5，其中 MD5 额外提供了加盐摘要。

同时在 cn.tuling.nettyadv.kryocodec 中定义了有关序列化和反序列化的工具类和 Handler，本项目中序列化使用了 Kryo 序列化框架。

## 服务端

服务端中 NettyServe 类是服务端的主入口，内部使用了 ServerInit 类进行 Handler 的安装。

最先安装的当然是解决粘包和半包问题的 Handler，很自然，这里应该用 LengthFieldBasedFrameDecoder 进行解码，为了实现方便，我们也没有在消息报文中附带消息的长度，由 Netty 帮我们在消息报文的最开始增加长度，所以编码器选择了 LengthFieldPrepender。

接下来，自然就是序列化和反序列化，直接使用我们在 kryocodec 下已经准备好的 KryoDecoder 和 KryoEncoder 即可。

服务端需要进行登录检查、心跳应答、业务处理，对应着三个 handler，于是我们分别安装了 LoginAuthRespHandler、HeartBeatRespHandler、ServerBusiHandler。

为了节约网络和服务器资源，如果客户端长久没有发送业务和心跳报文，我们认为客户端出现了问题，需要关闭这个连接，我们引入 Netty 的 ReadTimeoutHandler，当一定周期内（默认值 50s，我们设定为 15s）没有读取到对方任何消息时，会触发一个 ReadTimeouttException，这时我们检测到这个异常，需要主动关闭链路，并清除客户端登录缓存信息，等待客户端重连。

## 客户端

客户端的主类是 NettyClient，并对外提供一个方法 send，供业务使用内部使用了 ClientInit 类进行 Handler 的安装。

最先安装的当然是解决粘包和半包问题的 Handler，同样这里应该用 LengthFieldBasedFrameDecoder 进行解码，编码器选择了 LengthFieldPrepender。

接下来，自然就是序列化和反序列化，依然使用 KryoDecoder 和 KryoEncoder 即可。

客户端需要主动发出认证请求和心跳请求。

在 TCP 三次握手，链路建立后，客户端需要进行应用层的握手认证，才能使用服务，这个功能由 LoginAuthReqHandler 负责，而这个 Handler 在认证通过后，其实就没用了，所以在认证通过后，可以将这个 LoginAuthReqHandler 移除（其实服务端的认证应答 LoginAuthRespHandler 同样也可以移除）。

对于发出心跳请求有两种实现方式，一是定时发出，本框架的第一个版本就是这种实现方式，但是这种方式其实有浪费的情况，因为如果客户端和服务器正在正常业务通信，其实是没有必要发送心跳的；所以第二种方式就是，当链路写空闲时，为了维持通道，避免服务器关闭链接，发出心跳请求。为了实现这一点，我们首先在整个 pipeline 的最前面安装一个 CheckWriteIdleHandler 进行写空闲检测，空闲时间定位 8S，取服务器读空闲时间 15S 的一半，然后再安装一个 HearBeatReqHandler，因为写空闲会触发一个 FIRST_WRITER_IDLE_STATE_EVENT 入站事件，我们在 HearBeatReqHandler 的 userEventTriggered 方法中捕捉这个事件，并发出心跳请求报文。

考虑到在我们的实现中并没有双向心跳（即是客户端向服务器发送心跳请求，是服务器也向客户端发送心跳请求），客户端这边同样需要检测服务器是否存活，所以我们客户端这边安装了一个 ReadTimeoutHandler，捕捉 ReadTimeoutException 后提示调用者，并关闭通信链路，触发重连机制。

7、为了测试，单独建立一个 BusiClient，模拟业务方的调用。因为客户端的网络通信代码是在一个线程中单独启动的，为了协调主线程和通信线程的工作，我们引入了线程中的等待通知机制。

## 测试

1、 正常情况
2、 客户端宕机，服务器应能清除客户端的缓存信息，允许客户端重新登录
3、 服务器宕机，客户端应能发起重连
4、在 LoginAuthRespHandler 中进行注释，可以模拟当服务器不处理客户端的请求时，客户端在超时后重新进行登录。

## 功能的增强

作为一个通信框架，支持诊断也是很重要的，所以我们在服务端单独引入了一个 MetricsHandler，可以提供：目前在线 Channel 数、发送队列积压消息数、读取速率、写出速率相关数据，以方便应用方对自己的应用的性能和繁忙程度进行检查和调整。

当然对于一个通信框架还可以提供 SSL 安全访问、流控、I/O 线程和业务线程分离、参数的可配置化等等功能，我们就不一一展现了，同学们可以自行研究后实现，因为 Netty 对上述功能已经提供了很好的支持，大家后面要学习的 Dubbo 框架源码分析中基本都有对应的实现。

# 面试难题分析

## Netty 是如何解决 JDK 中的 Selector BUG 的?

Selector BUG：JDK NIO 的 BUG，例如臭名昭著的 epoll bug，它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK1.6 版本的 update18 修复了该问题，但是直到 JDK1.7 版本该问题仍旧存在，只不过该 BUG 发生概率降低了一些而已，它并没有被根本解决，甚至 JDK1.8 的 131 版本中依然存在。

JDK 官方认为这是 Linux Kernel 版本的 bug，可以参见：

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6403933

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=2147719

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6670302

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6481709

简单来说，JDK 认为 linux 的 epoll 告诉我事件来了，但是 JDK 没有拿到任何事件(READ、WRITE、CONNECT、ACCPET)，但此时 select()方法不再选择阻塞了，而是选择返回了 0，于是就会进入一种无限循环，导致 CPU 100%。

这个问题的具体原因是：在部分 Linux 的 2.6 的 kernel 中，poll 和 epoll 对于突然中断的连接 socket 会对返回的 eventSet 事件集合置为 POLLHUP 或 POLLERR，eventSet 事件集合发生了变化，这就可能导致 Selector 会被唤醒。但是这个时候 selector 的 select 方法返回 numKeys 是 0，所以下面本应该对 key 值进行遍历的事件处理根本执行不了，又回到最上面的 while(true)循环，循环往复，不断的轮询，直到 linux 系统出现 100%的 CPU 情况，最终导致程序崩溃。

Netty 解决办法：对 Selector 的 select 操作周期进行统计，每完成一次空的 select 操作进行一次计数，若在某个周期内连续发生 N 次空轮询，则触发了 epoll 死循环 bug。重建 Selector，判断是否是其他线程发起的重建请求，若不是则将原 SocketChannel 从旧的 Selector 上去除注册，重新注册到新的 Selector 上，并将原来的 Selector 关闭。

具体代码在 NioEventLoop 的 select 方法中：

```
long time = System.nanoTime();
if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >= currentTimeNanos) {
    // timeoutMillis elapsed without anything selected.
    selectCnt = 1;
} else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
        selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
    // The code exists in an extra method to ensure the method is not too big to inline as this
    // branch is not very likely to get hit very frequently.
    selector = selectRebuildSelector(selectCnt);
    selectCnt = 1;
    break;
}
```

## 如何让单机下 Netty 支持百万长连接?

单机下能不能让我们的网络应用支持百万连接？可以，但是有很多的工作要做。

## 操作系统

首先就是要突破操作系统的限制。

在 Linux 平台上，无论编写客户端程序还是服务端程序，在进行高并发 TCP 连接处理时，最高的并发数量都要受到系统对用户单一进程同时可打开文件数量的限制（这是因为系统为每个 TCP 连接都要创建一个 socket 句柄，每个 socket 句柄同时也是一个文件句柄）。

可使用 ulimit 命令查看系统允许当前用户进程打开的句柄数限制：

$ ulimit -n

1024

这表示当前用户的每个进程最多允许同时打开 1024 个句柄，这 1024 个句柄中还得除去每个进程必然打开的标准输入，标准输出，标准错误，服务器监听 socket,进程间通讯的 unix 域 socket 等文件，那么剩下的可用于客户端 socket 连接的文件数就只有大概 1024-10=1014 个左右。也就是说缺省情况下，基于 Linux 的通讯程序最多允许同时 1014 个 TCP 并发连接。

对于想支持更高数量的 TCP 并发连接的通讯处理程序，就必须修改 Linux 对当前用户的进程同时打开的文件数量。

修改单个进程打开最大文件数限制的最简单的办法就是使用 ulimit 命令：

$ ulimit –n 1000000

如果系统回显类似于"Operation not permitted"之类的话，说明上述限制修改失败，实际上是因为在中指定的数值超过了 Linux 系统对该用户打开文件数的软限制或硬限制。因此，就需要修改 Linux 系统对用户的关于打开文件数的软限制和硬限制。

*软限制（soft limit）:是指 Linux 在当前系统能够承受的范围内进一步限制一个进程同时打开的文件数；*

*硬限制（hardlimit）:是根据系统硬件资源状况（主要是系统内存）计算出来的系统最多可同时打开的文件数量。*

第一步，修改/etc/security/limits.conf 文件，在文件中添加如下行：

* soft nofile 1000000

* hard nofile 1000000

'*'号表示修改所有用户的限制；

soft 和 hard 为两种限制方式，其中 soft 表示警告的限制，hard 表示真正限制，nofile 表示打开的最大文件数。1000000 则指定了想要修改的新的限制值，即最大打开文件数（请注意软限制值要小于或等于硬限制）。修改完后保存文件。

第二步，修改/etc/pam.d/login 文件，在文件中添加如下行：

session required /lib/security/pam_limits.so

这是告诉 Linux 在用户完成系统登录后，应该调用 pam_limits.so 模块来设置系统对该用户可使用的各种资源数量的最大限制（包括用户可打开的最大文件数限制），而 pam_limits.so 模块就会从/etc/security/limits.conf 文件中读取配置来设置这些限制值。修改完后保存此文件。

第三步，查看 Linux 系统级的最大打开文件数限制，使用如下命令：

```
[speng@as4 ~]$ cat /proc/sys/fs/file-max
12158
```

这表明这台 Linux 系统最多允许同时打开（即包含所有用户打开文件数总和）12158 个文件，是 Linux 系统级硬限制，所有用户级的打开文件数限制都不应超过这个数值。如果没有特殊需要，不应该修改此限制，除非想为用户级打开文件数限制设置超过此限制的值。

如何修改这个系统最大文件描述符的限制呢？修改 sysctl.conf 文件

```
vi /etc/sysctl.conf
 # 在末尾添加
fs.file_max = 1000000
 # 立即生效
sysctl -p
```

# Netty 调优

## *设置合理的线程数*

对于线程池的调优,主要集中在用于接收海量设备 TCP 连接、TLS 握手的 Acceptor 线程池( Netty 通常叫 boss NioEventLoop Group)上,以及用于处理网络数据读写、心跳发送的 1O 工作线程池(Nety 通常叫 work Nio EventLoop Group)上。

对于 Nety 服务端,通常只需要启动一个监听端口用于端侧设备接入即可,但是如果服务端集群实例比较少,甚至是单机(或者双机冷备)部署,在端侧设备在短时间内大量接入时,需要对服务端的监听方式和线程模型做优化,以满足短时间内(例如 30s)百万级的端侧设备接入的需要。

服务端可以监听多个端口,利用主从 Reactor 线程模型做接入优化,前端通过 SLB 做 4 层门 7 层负载均衡。

主从 Reactor 线程模型特点如下:服务端用于接收客户端连接的不再是一个单独的 NO 线程,而是一个独立的 NIO 线程池; Acceptor 接收到客户端 TCP 连接请求并处理后(可能包含接入认证等),将新创建的 Socketchanne 注册到 I/O 线程池(subReactor 线程池)的某个 IO 线程,由它负责 Socketchannel 的读写和编解码工作; Acceptor 线程池仅用于客户端的登录、握手和安全认证等,一旦链路建立成功,就将链路注册到后端 sub reactor 线程池的 IO 线程,由 IO 线程负责后续的 IO 操作。

对于 IO 工作线程池的优化,可以先采用系统默认值(即 CPU 内核数×2)进行性能测试,在性能测试过程中采集 IO 线程的 CPU 占用大小,看是否存在瓶颈， 具体可以观察线程堆栈，如果连续采集几次进行对比,发现线程堆栈都停留在 SelectorImpl. lockAndDoSelect，则说明 IO 线程比较空闲,无须对工作线程数做调整。

如果发现 IO 线程的热点停留在读或者写操作,或者停留在 Channelhandler 的执行处,则可以通过适当调大 Nio EventLoop 线程的个数来提升网络的读写性能。

## *心跳优化*

针对海量设备接入的服务端,心跳优化策略如下。

(1)要能够及时检测失效的连接,并将其剔除,防止无效的连接句柄积压,导致 OOM 等问题

(2)设置合理的心跳周期,防止心跳定时任务积压,造成频繁的老年代 GC(新生代和老年代都有导致 STW 的 GC,不过耗时差异较大),导致应用暂停

(3)使用 Nety 提供的链路空闲检测机制,不要自己创建定时任务线程池,加重系统的负担,以及增加潜在的并发安全问题。

当设备突然掉电、连接被防火墙挡住、长时间 GC 或者通信线程发生非预期异常时,会导致链路不可用且不易被及时发现。特别是如果异常发生在凌晨业务低谷期间,当早晨业务高峰期到来时,由于链路不可用会导致瞬间大批量业务失败或者超时,这将对系统的可靠性产生重大的威胁。

从技术层面看,要解决链路的可靠性问题,必须周期性地对链路进行有效性检测。目前最流行和通用的做法就是心跳检测。心跳检测机制分为三个层面:

(1)TCP 层的心跳检测,即 TCP 的 Keep-Alive 机制,它的作用域是整个 TCP 协议栈。

(2)协议层的心跳检测,主要存在于长连接协议中,例如 MQTT。

(3)应用层的心跳检测,它主要由各业务产品通过约定方式定时给对方发送心跳消息实现。

心跳检测的目的就是确认当前链路是否可用,对方是否活着并且能够正常接收和发送消息。作为高可靠的 NIO 框架,Nety 也提供了心跳检测机制。

一般的心跳检测策略如下。

(1)连续 N 次心跳检测都没有收到对方的 Pong 应答消息或者 Ping 请求消息,则认为链路已经发生逻辑失效,这被称为心跳超时。

(2)在读取和发送心跳消息的时候如果直接发生了 IO 异常,说明链路已经失效,这被称为心跳失败。无论发生心跳超时还是心跳失败,都需要关闭链路,由客户端发起重连操作,保证链路能够恢复正常。

Nety 提供了三种链路空闲检测机制,利用该机制可以轻松地实现心跳检测

(1)读空闲,链路持续时间 T 没有读取到任何消息。

(2)写空闲,链路持续时间 T 没有发送任何消息

(3)读写空闲,链路持续时间 T 没有接收或者发送任何消息

对于百万级的服务器，一般不建议很长的心跳周期和超时时长。

## *接收和发送缓冲区调优*

在一些场景下,端侧设备会周期性地上报数据和发送心跳,单个链路的消息收发量并不大,针对此类场景,可以通过调小 TCP 的接收和发送缓冲区来降低单个 TCP 连接的资源占用率

当然对于不同的应用场景,收发缓冲区的最优值可能不同,用户需要根据实际场景,结合性能测试数据进行针对性的调优

## *合理使用内存池*

随着 JVM 虚拟机和 JT 即时编译技术的发展,对象的分配和回收是一个非常轻量级的工作。但是对于缓冲区 Buffer,情况却稍有不同,特别是堆外直接内存的分配和回收,是一个耗时的操作。

为了尽量重用缓冲区,Nety 提供了基于内存池的缓冲区重用机制。

在百万级的情况下,需要为每个接入的端侧设备至少分配一个接收和发送 ByteBuf 缓冲区对象,采用传统的非池模式,每次消息读写都需要创建和释放 ByteBuf 对象,如果有 100 万个连接,每秒上报一次数据或者心跳,就会有 100 万次/秒的 ByteBuf 对象申请和释放,即便服务端的内存可以满足要求,GC 的压力也会非常大。

以上问题最有效的解决方法就是使用内存池,每个 NioEventLoop 线程处理 N 个链路,在线程内部,链路的处理是串行的。假如 A 链路首先被处理,它会创建接收缓冲区等对象,待解码完成,构造的 POJO 对象被封装成任务后投递到后台的线程池中执行,然后接收缓冲区会被释放,每条消息的接收和处理都会重复接收缓冲区的创建和释放。如果使用内存池,则当 A 链路接收到新的数据报时,从 NioEventLoop 的内存池中申请空闲的 ByteBuf,解码后调用 release 将 ByteBuf 释放到内存池中,供后续的 B 链路使用。

Nety 内存池从实现上可以分为两类:堆外直接内存和堆内存。由于 Byte Buf 主要用于网络 IO 读写,因此采用堆外直接内存会减少一次从用户堆内存到内核态的字节数组拷贝,所以性能更高。由于 DirectByteBuf 的创建成本比较高,因此如果使用 DirectByteBuf,则需要配合内存池使用,否则性价比可能还不如 Heap Byte。

Netty 默认的 IO 读写操作采用的都是内存池的堆外直接内存模式,如果用户需要额外使用 ByteBuf,建议也采用内存池方式;如果不涉及网络 IO 操作(只是纯粹的内存操作),可以使用堆内存池,这样内存的创建效率会更高一些。

### *IO 线程和业务线程分离*

如果服务端不做复杂的业务逻辑操作,仅是简单的内存操作和消息转发,则可以通过调大 NioEventLoop 工作线程池的方式,直接在 IO 线程中执行业务 Channelhandler,这样便减少了一次线程上下文切换,性能反而更高。

如果有复杂的业务逻辑操作,则建议 IO 线程和业务线程分离,对于 IO 线程,由于互相之间不存在锁竞争,可以创建一个大的 NioEvent Loop Group 线程组,所有 Channel 都共享同一个线程池。

对于后端的业务线程池,则建议创建多个小的业务线程池,线程池可以与 IO 线程绑定,这样既减少了锁竞争,又提升了后端的处理性能。

### *针对端侧并发连接数的流控*

无论服务端的性能优化到多少,都需要考虑流控功能。当资源成为瓶颈,或者遇到端侧设备的大量接入,需要通过流控对系统做保护。流控的策略有很多种，比如针对端侧连接数的流控：

在 Nety 中,可以非常方便地实现流控功能:新增一个 FlowControlchannelhandler，然后添加到 ChannelPipeline 靠前的位置,覆盖 channelActive()方法,创建 TCP 链路后,执行流控逻辑,如果达到流控阈值,则拒绝该连接,调用 ChannelHandler Context 的 close(方法关闭连接。

## JVM 层面相关性能优化

当客户端的并发连接数达到数十万或者数百万时,系统一个较小的抖动就会导致很严重的后果,例如服务端的 GC,导致应用暂停(STW)的 GC 持续几秒,就会导致海量的客户端设备掉

线或者消息积压,一旦系统恢复,会有海量的设备接入或者海量的数据发送很可能瞬间就把服务端冲垮。

JVM 层面的调优主要涉及 GC 参数优化,GC 参数设置不当会导致频繁 GC,甚至 OOM 异常,对服务端的稳定运行产生重大影响。

### 1.确定 GC 优化目标

GC(垃圾收集)有三个主要指标。

(1)吞吐量:是评价 GC 能力的重要指标,在不考虑 GC 引起的停顿时间或内存消耗时,吞吐量是 GC 能支撑应用程序达到的最高性能指标。

(2)延迟:GC 能力的最重要指标之一,是由于 GC 引起的停顿时间,优化目标是缩短延迟时间或完全消除停顿(STW),避免应用程序在运行过程中发生抖动。

(3)内存占用:GC 正常时占用的内存量。

JVM GC 调优的三个基本原则如下。

(1) Minor go 回收原则:每次新生代 GC 回收尽可能多的内存,减少应用程序发生 Full gc 的频率。

2)GC 内存最大化原则:垃圾收集器能够使用的内存越大,垃圾收集效率越高,应用程序运行也越流畅。但是过大的内存一次 Full go 耗时可能较长,如果能够有效避免 FullGC,就需要做精细化调优。

(3)3 选 2 原则:吞吐量、延迟和内存占用不能兼得,无法同时做到吞吐量和暂停时间都最优,需要根据业务场景做选择。对于大多数应用,吞吐量优先,其次是延迟。当然对于时延敏感型的业务,需要调整次序。

### 2.确定服务端内存占用

在优化 GC 之前,需要确定应用程序的内存占用大小,以便为应用程序设置合适的内存,提升 GC 效率。内存占用与活跃数据有关,活跃数据指的是应用程序稳定运行时长时间存活的 Java 对象。活跃数据的计算方式:通过 GC 日志采集 GC 数据,获取应用程序稳定时老年代占用的 Java 堆大小,以及永久代(元数据区)占用的 Java 堆大小,两者之和就是活跃数据的内存占用大小。

### 3.GC 优化过程

1、GC 数据的采集和研读

2、设置合适的 JVM 堆大小

3、选择合适的垃圾回收器和回收策略

当然具体如何做，请参考 JVM 相关课程。而且 GC 调优会是一个需要多次调整的过程，期间不仅有参数的变化，更重要的是需要调整业务代码。

## 什么是水平触发(LT)和边缘触发(ET)?

Level_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完，那么下次调用 epoll_wait()

时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你。

Edge_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完，那么下次调用 epoll_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你。这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！

select()，poll()模型都是水平触发模式，信号驱动 IO 是边缘触发模式，epoll()模型即支持水平触发，也支持边缘触发，默认是水平触发。JDK 中的 select 实现是水平触发，而 Netty 提供的 Epoll 的实现中是边缘触发。

# 请说说 DNS 域名解析的全过程

本题其实是“浏览器中输入 URL 到返回页面的全过程”这个题目的衍生题：

1. 根据域名，进行 DNS 域名解析；

2. 拿到解析的 IP 地址，建立 TCP 连接；

3. 向 IP 地址，发送 HTTP 请求；

4. 服务器处理请求；

5. 返回响应结果；

6. 关闭 TCP 连接；

7. 浏览器解析 HTML；

8. 浏览器布局渲染；

可见 DNS 域名解析是其中的一部分。

DNS 一个由分层的服务系统，大致说来，有 3 种类型的 DNS 服务器:根 DNS 服务器、顶级域(Top-Level Domain，TLD） DNS 服务器和权威 DNS 服务器。

根 DNS 服务器。截止到 2022 年 4 月 22 日，有 1533 个根名字服务器遍及全世界，可到 https://root-servers.org/查询分布情况，根名字服务器提供 TLD 服务器的 IP 地址。

顶级域（DNS）服务器。对于每个顶级域（如 com、org、net、edu 和 gov）和所有国家的顶级域（如 uk、fr、ca 和 jp），都有 TLD 服务器（或服务器集群）。TLD 服务器提供了权威 DNS 服务器的 IP 地址。

权威 DNS 服务器。在因特网上的每个组织机构必须提供公共可访问的 DNS 记录，这些记录将这些主机的名字映射为 IP 地址。一个组织机构的权威 DNS 服务器收藏了这些 DNS 记录。一个组织机构能够选择实现它自己的权威 DNS 服务器以保存这些记录；也可以交由商用 DNS 服务商存储在这个服务提供商的一个权威 DNS 服务器中，比如阿里云旗下的中国万网。

有另一类重要的 DNS 服务器，称为本地 DNS 服务器（ local DNS server）。严格说来，一个本地 DNS 服务器并不属于该服务器的层次结构，但它对 DNS 层次结构是至关重要的。每个 ISP 都有一台本地 DNS 服务器。同时很多路由器中也会附带 DNS 服务。

当主机发出 DNS 请求时,该请求被发往本地 DNS 服务器，它起着代理的作用，并将该请求转发到 DNS 服务器层次结构中，同时本地 DNS 服务器也会缓存 DNS 记录。

所以一个 DNS 客户要决定主机名 www.baidu.com 的 IP 地址。粗略说来，将发生下列事件。客户首先与根服务器之一联系，它将返回顶级域名 com 的 TLD 服务器的 IP 地址。该客户则与这些 TLD 服务器之一联系，它将为 baidu.com 返回权威服务器的 IP 地址。最后，该客户与 baidu.com 权威服务器之一联系，它为主机名 www.baidu.com 返回其 IP 地址。