注:这个系列帮助解决项目相关 内容，涉及到项目本身，netty 理 解，rpc 理解,dubbo 理 解，zookeeper 理解，基于该项 目，奠定学生网络通信、分布式 相关的基础基石，打出自己的一 柄剑。面试这个东西，核心就是 区分度，基础大家都会，实践+脱离基础本身，才能大大提高成功 率

项目相关问题:

(1)为何基于 Netty 进行 rpc 网络通信的选型?(简单理解为 为什么使用 netty?)

答案在《Netty 权威指南》中，关键字,高性能，其中涉及到 **NIO**、线程模型、**IO** 模型、Netty 的优势，这些东西全在《Netty 权威指南》中，你懂我意思嘛?

Netty优势:

1. API 使用简单，开发门槛低
2. 功能强大,预置了多种编解码功能,支持多种主流协议
3. 定制能力强,可以通过 ChannelHandler对通信框架进行灵活地扩展
4. 性能高,通过与其他业界主流的NIO框架对比, Netty的综合性能最优
5. 成熟、稳定,Netty修复了已经发现的所有 JDK NIO BUG,业务开发人员不需要再为NIO的BUG而烦恼
6. 社区活跃,版本迭代周期短,发现的BUG可以被及时修复,同时,更多的新功能会加入

高性能:

1. 采用异步非阻塞的I/O类库,基于 Reactor模式实现,解决了传统同步阻塞I/O 模式下一个服务端无法平滑地处理线性增长的客户端的问题。
2. TCP接收和发送缓冲区使用直接内存代替堆内存,避免了内存复制,提升了I/O 读取和写入的性能
3. 支持通过内存池的方式循环利用 ByteBuf,避免了频繁创建和销 ByteBuf带来的性能损耗。
4. 可配置的I/O线程数、TCP参数等,为不同的用户场景提供定制化的调优参数, 满足不同的性能场景。
5. 采用环形数组缓冲区实现无锁化并发编程,代替传统的线程安全容器或者锁。
6. 合理地使用线程安全容器、原子类等,提升系统的并发处理能力。
7. 关键资源的处理使用单线程串行化的方式,避免多线程并发访问带来的锁竞争和额外的CPU资源消耗问题。
8. 通过引用计数器及时地申请释放不再被引用的对象,细粒度的内存管理降低了GC的频率,减少了频繁GC带来的时延增大和CPU损耗。

Reactor 模式：

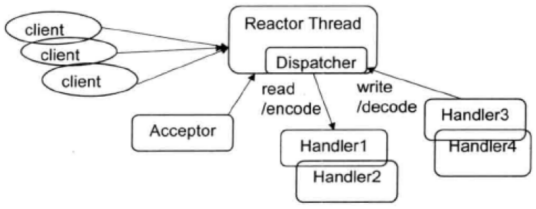


图 18-1　Reactor 单线程模型

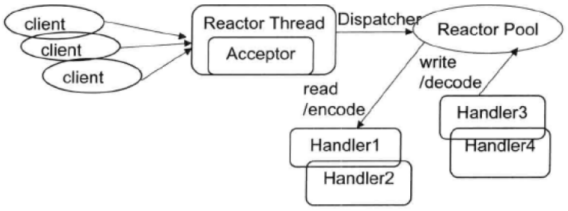


图 18-2　Reactor 多线程模型

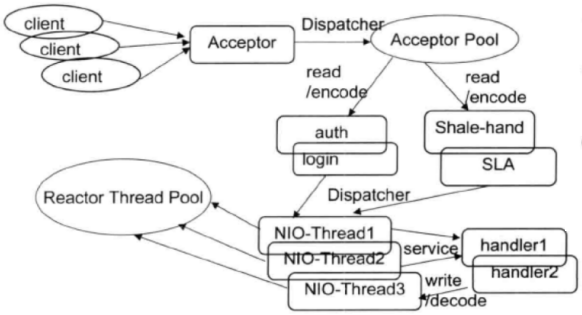


图 18-3　主从 Reactor 多线程模型

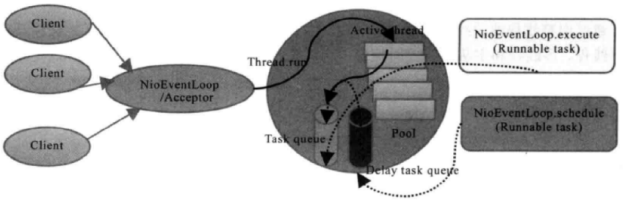


图 18-4　Netty 的线程模型

为了尽可能地提升性能,Netty在很多地方进行了无锁化的设计,例如在I/O线程内部进行串行操作,避免多线程竞争导致的性能下降问题

只要用户不主动切换线程,一直都是由 NioEventLoop调用用户的 Handler, 期间不进行线程切换。这种串行化处理方式避免了多线程操作导致的锁的竞争,从性能角度看是最优的。

IO模型对比：

表 2-1　几种 I/O 模型的功能和特性对比

	同步阻塞 I/O（BIO）	伪异步 I/O	非阻塞 I/O（NIO）	异步 I/O（AIO）
客户端个数：I/O 线程	1：1	*M*：*N*（其中 *M* 可以大于 *N*）	*M*：1（1 个 I/O 线程处理多个客户端连接）	*M*：0（不需要启动额外的 I/O 线程，被动回调）
I/O 类型（阻塞）	阻塞 I/O	阻塞 I/O	非阻塞 I/O	非阻塞 I/O

续表

	同步阻塞 I/O（BIO）	伪异步 I/O	非阻塞 I/O（NIO）	异步 I/O（AIO）
I/O 类型（同步）	同步 I/O	同步 I/O	同步 I/O（I/O 多路复用）	异步 I/O
API 使用难度	简单	简单	非常复杂	复杂
调试难度	简单	简单	复杂	复杂
可靠性	非常差	差	高	高
吞吐量	低	中	高	高

可靠性：

1. 链路有效性检测

为了保证长连接的链路有效性,往往需要通过心跳机制周期性地进行链路检测。使用周期性心跳的原因是:在系统空闲时,例如凌晨,往往没有业务消息。如果此时链路被防火墙Hang住,或者遭遇网络闪断、网络单通等,通信双方无法识别出这类链路异常。
为了支持心跳,Netty提供了如下两种链路空闲检测机制

读空闲超时机制:当连续周期T没有消息可读时,触发超时 Handler,用户可以基于读空闲超时发送心跳消息,进行链路检测:如果连续N个周期仍然没有读取到心跳消息,可以主动关闭链路。

写空闲超时机制:当连续周期T没有消息要发送时,触发超时 Handler,用户可以基于写空闲超时发送心跳消息,进行链路检测;如果连续N个周期仍然没有接收到对方的心跳消息,可以主动关闭链路
为了满足不同用户场景的心跳定制, Netty提供了空闲状态检测事件通知机制,用户可以订阅空闲超时事件、写空闲超时事件、读或者写超时事件,在接收到对应的空闲事件之后,灵活地进行定制。

2. 内存保护机制

通过对象引用计数器对Netty的 ByteBuf等内置对象进行细粒度的内存申请和释放,对非法的对象引用进行检测和保护。
通过内存池来重用 Bytebuf,节省内存
可设置的内存容量上限,包括 Bytebuf、线程池线程数等

3. 优雅停机

优雅停机功能指的是当系统退出时,JVM通过注册的 Shutdown Hook拦截到退出信号量,然后执行退出操作,释放相关模块的资源占用,将缓冲区的消息处理完成或者清空,将待刷新的数据持久化到磁盘或者数据库中,等到资源回收和缓冲区消息处理完成之后,再退出

(2)短连接的问题?为什么使用长连接?

长连接与短连接相关知识普及(看完这篇文章就懂了):https://zhuanlan.zhihu.com/p/ 47074758

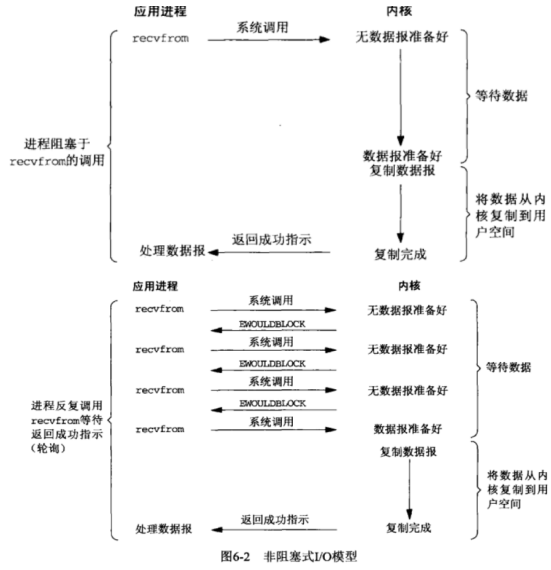
核心回答:基于长连接的优势进行回答即可(核心就是省去大 量 **tcp/ip**建 立过 程 ， 比 如你有 200 万次请求，难道你要进行 200 万次的 tcp/ip 连接?时间+连接字节的浪费是不是 很大?)

(3)同步异步的差距。为何选用异步?

这个问题想战胜别人，比别人答的好，首先回答《Unix 网络编程》中的 **5** 种 **I/o** 模 型 ， 让 面试官了解你并非仅懂同步异步的差距 ， 更知底层，然后开始讲解异步同步概念上的区 别，接下来开始讲解性能方面的差距(参考《netty 权威指南》)。

引申问到 I/O 模型可能会问到 1 个问题:select、poll 和 epoll 的区别 记住:能贴近《UNIX 网络编程》底层的就贴上去回答，展示自己的深度。

5种IO模型:



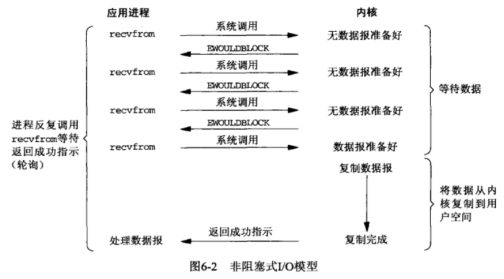


图6-2 非阻塞式I/O模型

图6-2 非阻塞式I/O模型

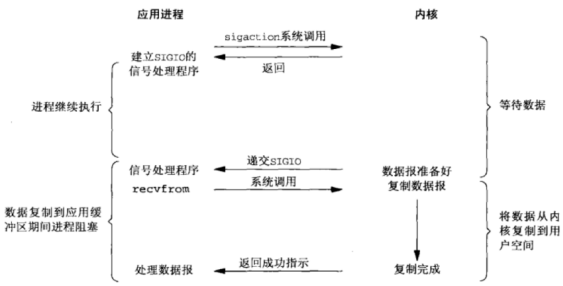


图6-4 信号驱动式I/O模型

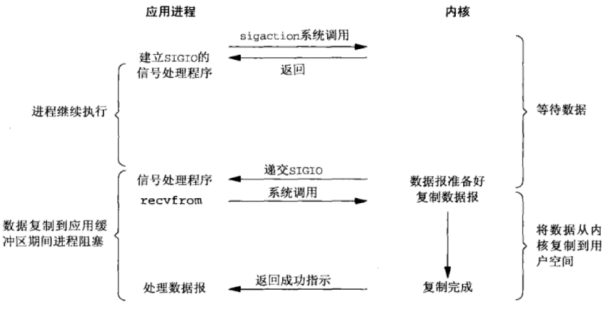


图6-4 信号驱动式I/O模型

同步IO操作(synchronous I/O operation)导致请求进程阻塞,直到IO操作完成
异步IO操作(asynchronous I/O operation)不导致请求进程阻塞。

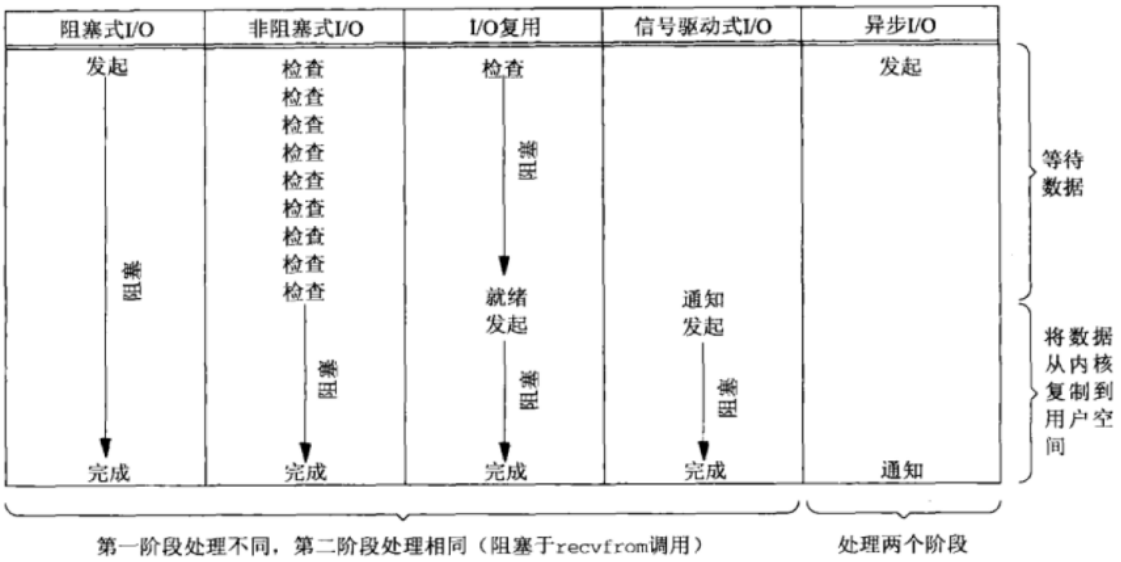


图6-6　5种I/O模型的比较

select、poll和 epoll 的区别：

1、支持一个进程所能打开的最大连接数：

select：单个进程所能打开的最大连接数有FD_SETSIZE宏定义，其大小是32个整数的大小（在32位的机器上，大小就是3232，同理64位机器上FD_SETSIZE为3264），当然我们可以对进行修改，然后重新编译内核，但是性能可能会受到影响，这需要进一步的测试。

poll：poll本质上和select没有区别，但是它没有最大连接数的限制，原因是它是基于链表来存储的。

epoll：虽然连接数有上限，但是很大，1G内存的机器上可以打开10万左右的连接，2G内存的机器可以打开20万左右的连接。

2、FD剧增后带来的IO效率问题

select：因为每次调用时都会对连接进行线性遍历，所以随着FD的增加会造成遍历速度慢的“线性下降性能问题”。

poll：同上

epoll：因为epoll内核中实现是根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

3、 消息传递方式

select：内核需要将消息传递到用户空间，都需要内核拷贝动作

poll：同上

epoll：epoll通过内核和用户空间共享一块内存来实现的。

(1)select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。

(2)poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， 但是它没有最大连接数的限制，原因是它是基于链表来存储的.

(3)epoll==>时间复杂度O(1)

epoll可以理解为event poll，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是事件驱动（每个事件关联上fd）的，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）

(4)心跳机制如何实现?

项目视频中有讲解，再弄懂《Netty 权威指南》当中关于心跳机制即可，这个问题可能引

申到 dubbo 的心跳机制，感兴趣可以搜 1 下相关资料即可。

1. 在pipeline 中添加 IdleStateHandler（）并设置读空闲、写空闲和读写空闲的时间
2. 在ServerHandler中重写userEventTriggered（）方法进行检测，在读空闲的时候关闭channel，在读写空闲时发宋“pong”信息
3. 在ClientHandler中对服务器发送的“pong”信息进行回应

Netty心跳机制：

为了保证长连接的链路有效性,往往需要通过心跳机制周期性地进行链路检测。使用周期性心跳的原因是:在系统空闲时,例如凌晨,往往没有业务消息。如果此时链路被防火墙Hang住,或者遭遇网络闪断、网络单通等,通信双方无法识别出这类链路异常。
为了支持心跳,Netty提供了如下两种链路空闲检测机制

读空闲超时机制:当连续周期T没有消息可读时,触发超时 Handler,用户可以基于读空闲超时发送心跳消息,进行链路检测:如果连续N个周期仍然没有读取到心跳消息,可以主动关闭链路。

写空闲超时机制:当连续周期T没有消息要发送时,触发超时 Handler,用户可以基于写空闲超时发送心跳消息,进行链路检测;如果连续N个周期仍然没有接收到对方的心跳消息,可以主动关闭链路
为了满足不同用户场景的心跳定制, Netty提供了空闲状态检测事件通知机制,用户可以订阅空闲超时事件、写空闲超时事件、读或者写超时事件,在接收到对应的空闲事件之后,灵活地进行定制。

dubbo的心跳机制：

目的：检测provider与consumer之间的connection连接是不是还连接着，如果连接断了，需要作出相应的处理。

原理：

provider：dubbo的心跳默认是在heartbeat（默认是60s）内如果没有接收到消息，就会发送心跳消息，如果连着3次（180s）没有收到心跳响应，provider会关闭channel。

consumer：dubbo的心跳默认是在60s内如果没有接收到消息，就会发送心跳消息，如果连着3次（180s）没有收到心跳响应，consumer会进行重连。

(5) 序列化问题-为什么不用 Java 原生序列化，有什么问

题?为什么选用 JSON?还有哪些序列化方式?

这是一个高频问题，首先你要知道什么是序列化，以及序列化在网络传输中的影响， 其次要知道序列化都有哪些选择以及带来的速度/性能问题，最后有 1 些性能对比数据。以 上这些， 《Netty 权威指南》中有 1 个章节讲的全是关于序列化的问题，看透，再搜 1-2 篇 关于序列化的博客即可。

参考:深入理解 RPC—序列化

概念:

网络传输的数据必须是二进制数据，但调用方请求的出入参数都是对象。对象是不能直接在网络中传输的，所以我们需要提前把它转成可传输的二进制，并且要求转换算法是可逆的，这个过程我们一般叫做“序列化”。这时，服务提供方就可以正确地从二进制数据中分割出不同的请求，同时根据请求类型和序列化类型，把二进制的消息体逆向还原成请求对象，这个过程我们称之为“反序列化”。

序列化就是将对象转换成二进制数据的过程，而反序列就是反过来将二进制转换为对象的过程

序列化的影响:

序列化与反序列化过程是 RPC 调用的一个必须过程，那么序列化与反序列化的性能和效率势必将直接关系到 RPC 框架整体的性能和效率。

还有空间开销，也就是序列化之后的二进制数据的体积大小。序列化后的字节数据体积越小，网络传输的数据量就越小，传输数据的速度也就越快，由于 RPC 是远程调用，那么网络传输的速度将直接关系到请求响应的耗时。

兼容性，对于跨进程的服务调用，服务提供者可以能会使用别的语言开发，当需要和异构语言进程交互是，需要强大的兼容性

Java序列化的缺点：

1. 无法跨语言,是Java序列化最致命的问题。
2. 序列化后的码流太大
3. 序列化性能太低

采用JDK序列化机制编码后的二进制数组大小是二进制编码的5.29倍。
Java序列化的性能只有二进制编码的6.17%左右,可见Java 原生序列化的性能实在太差。

JSON

JSON 是典型的 Key-Value 方式，没有数据类型，是一种文本型序列化框架，他在应用上很广泛，无论是前台 Web 用 Ajax 调用、用磁盘存储文本类型的数据，还是基于 HTTP 协议的 RPC 框架通信，都会选择 JSON 格式。

JSON 进行序列化的额外空间开销比较大，对于大数据量服务这意味着需要巨大的内存和磁盘开销；
JSON 没有类型，但像 Java 这种强类型语言，需要通过反射统一解决，所以性能不会太好。
所以如果 RPC 框架选用 JSON 序列化，服务提供者与服务调用者之间传输的数据量要相对较小，否则将严重影响性能

Hessian

相对于 JDK、JSON，由于 Hessian 更加高效，生成的字节数更小，有非常好的兼容性和稳定性，所以 Hessian 更加适合作为 RPC 框架远程通信的序列化协议。
但 Hessian 本身也有问题，官方版本对 Java 里面一些常见对象的类型不支持，比如：

Linked 系列，LinkedHashMap、LinkedHashSet 等，但是可以通过扩展 CollectionDeserializer 类修复；
Locale 类，可以通过扩展 ContextSerializerFactory 类修复；
Byte/Short 反序列化的时候变成 Integer。

Protobuf

Protobuf 是 Google 公司内部的混合语言数据标准，是一种轻便、高效的结构化数据存储格式，可以用于结构化数据序列化，支持 Java、Python、C++、Go 等语言。Protobuf使

用的时候需要定义 IDL（Interface description language），然后使用不同语言的 IDL编译器，生成序列化工具类，它的优点是：
序列化后体积相比 JSON、Hessian 小很多；
IDL 能清晰地描述语义，所以足以帮助并保证应用程序之间的类型不会丢失，无需类似 XML 解析器；
序列化反序列化速度很快，不需要通过反射获取类型；
消息格式升级和兼容性不错，可以做到向后兼容。
Protobuf 非常高效，但是对于具有反射和动态能力的语言来说，这样用起来很费劲，这一点就不如 Hessian，比如用 Java 的话，这个预编译过程不是必须的

Thrift

Thrift是为了解决 Facebook各系统间大数据量的传输通信以及系统之间语言环境不同需要跨平台的特性,因此 Thrift可以支持多种程序语言,如C++、C#、 Cocoa、 Erlang、 Haskell, Java、 Cami、Perl、PHP、 Python、Ruby 和 Smalltalk。
Thrift适用于静态的数据交换,需要先确定好它的数据结构,当数据结构发生变化时,必须重新编辑IDL文件,生成代码和编译,这一点跟其他IDL工具相比可以视为是 Thrift的弱项。
Thrift适用于搭建大型数据交换及存储的通用工具,对于大型系统中的内部数据传输,相对于JSON和XML在性能和传输大小上都有明显的优势。

JBoss Marshalling

JBoss Marshalling是一个Java对象的序列化API包,修正了JDK自带的序列化包的很多问题,但又保持跟 Java io. Serializable接口的兼容

相比于传统的Java序列化机制,它的优点,如下
可插拔的类解析器,提供更加便捷的类加载定制策略,通过一个接口即可实现定制
可插拔的对象替换技术,不需要通过继承的方式
可插拔的预定义类缓存表,可以减小序列化的字节数组长度,提升常用类型的对象序列化性能
无须实现java.io. Serializable接口,即可实现Java序列化
通过缓存技术提升对象的序列化性能
相比于前面介绍的两种编解码框架, JBOSS Marshalling更多是在 JBOSS 内部使用,应用范围有限。

(6) 如何基于动态代理对请求进行处理?

这个看项目视频如何做的即可，然后面试官一般会多说 1 句:你给我讲讲动态代 理，java 动态代理实现与原理详细分析,然后再讲讲 spring 中怎么用的动态代理(细说 Spring——AOP 详解(动态代理实现 AOP))，这 1 套下来，面试官不得不满意，你懂我意 思嘛?你的知识体系让面试官内心开始欣喜

---

1. 创建代理类实现BeanPostProcessor
2. 重写postProcessBeforeInitialization和postProcessAfterInitialization方法
3. 在postProcessBeforeInitialization方法中获得bean的所有的Field
4. 在遍历Field并判断Field上的注解是否有RemoteInvoke的注解
5. 如果有进行修改，先将Accessible 设置为true，在设置这个field的value
6. value是一个动态代理，同Enhancer 来完成动态代理
7. 先创建一个Enhancer对象，设置需要动态代理的接口 enhancer.setInterfaces(new Class[]{field.getType()})
8. 调用enhancer.esetCallback(new MethodInterceptor(){…})方法,重写里面的intercept 拦截方法，并选择拦截哪些方法
9. 拦截时，采用netty客户端去调用服务器获取结果完成整个动态代理
10. postProcessAfterInitialization方法直接返回bean即可

Java动态代理

代理模式是常用的java设计模式，他的特征是代理类与委托类有同样的接口，代理类主要负责为委托类预处理消息、过滤消息、把消息转发给委托类，以及事后处理消息等

1、静态代理

静态代理：由程序员创建或特定工具自动生成源代码，也就是在编译时就已经将接口，被代理类，代理类等确定下来。在程序运行之前，代理类的.class文件就已经生成
代理模式最主要的就是有一个公共接口，一个具体的类，一个代理类,代理类持有具体类的实例，代为执行具体类实例方法
使用代理模式的一个很大的优点。最直白的就是在Spring中的面向切面编程（AOP），我们能在一个切点之前执行一些操作，在一个切点之后执行一些操作，这个切点就是一个个方法。这些方法所在类肯定就是被代理了，在代理过程中切入了一些其他操作。

2.动态代理

代理类在程序运行时创建的代理方式被成为动态代理。
动态代理，代理类并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的“指示”动态生成的。相比于静态代理， 动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类中的方法。

在java的java.lang.reflect包下提供了一个Proxy类和一个InvocationHandler接口，通过这个类和这个接口可以生成JDK动态代理类和动态代理对象。
实现步骤：

1. 创建一个与代理对象相关联的InvocationHandler
2. 使用Proxy类的getProxyClass静态方法生成一个动态代理类stuProxyClass
3. 获得stuProxyClass 中一个带InvocationHandler参数的构造器constructor
4. 通过构造器constructor来创建一个动态实例stuProxy
5. 上面四个步骤可以通过Proxy类的newProxyInstances方法来简化

动态代理的优势在于可以很方便的对代理类的函数进行统一的处理，而不用修改每个代理类中的方法。是因为所有被代理执行的方法，都是通过在InvocationHandler中的invoke方法调用的，所以我们只要在invoke方法中统一处理，就可以对所有被代理的方法进行相同的操作了。
jdk会为我们的生成了一个叫$Proxy0（这个名字后面的0是编号，有多个代理类会一次递增）的代理类，这个类文件时放在内存中的，我们在创建代理对象时，就是通过反射获得这个类的构造方法，然后创建的代理实例。通过对这个生成的代理类源码的查看，我们很容易能看出，动态代理实现的具体过程。

Spring AOP

通过ProxyFactory类，在类中对代理类进行创建，方法调用及执行Advice

1. 先获取classLoader 和所有接口类型
2. 创建invovationHandler并重写invoke方法
3. 在invoke方法中，在不同的切点进行Advice，并执行目标对象的目标方法，并返回结果
4. 通过Proxy类的newProxyInstance方法获取代理对象，并返回代理对象

(7) 如何解决进程间通信问题?

这块没啥好说的，wait/notify 机制而已，并发那块之前搞定过，往这上答即可。

使用来ReentrantLock和Condition进行控制

1. 创建ReentrantLock和Condition
2. 在主线程获取数据前，先要等待结果
3. 在获取时候结果时，先上锁，防止多个线程同时获取结果
4. 如果还没有结果，则进入Condition的等待室进行等待（condition.await），并在finally块中释放锁

5. 当获取到消息时，通过signal方法唤醒在condition等待室中等待的线程

(8) TCP/IP粘包问题你是如何解决的?

首先，讲清楚什么叫拆包粘包问题?然后，说 1 下通用方案，其次说下自己的方式。 这个问题面试官更想探讨的是你对拆包粘包的理解，更深 1 层的说就是你对网络通信的一 些理解，这部分《Netty 权威指南》上有单独章节讲这个问题，答好很简单。

TCP/IP粘包和拆包:

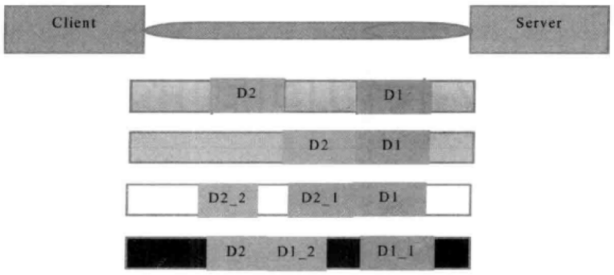


图 4-1 TCP 粘包/拆包问题

(1)服务端分两次读取到了两个独立的数据包,分别是D1和D2,没有粘包和拆包
(2)服务端一次接收到了两个数据包,D1和D2粘合在一起,被称为TCP粘包
(3)服务端分两次读取到了两个数据包,第一次读取到了完整的D1包和D2包的部分内容,第二次读取到了D2包的剩余内容,这被称为TCP拆包
(4)服务端分两次读取到了两个数据包,第一次读取到了D1包的部分内容D1_1,第二次读取到了D1包的剩余内容D1_2和D2包的整包
如果此时服务端TCP接收滑窗非常小,而数据包D1和D2比较大,很有可能会发生第5种可能,即服务端分多次才能将D1和D2包接收完全,期间发生多次拆包。

问题产生的原因有三个,分别如下:
1. 应用程序 write写入的字节大小大于套接口发送缓冲区大小;
2. 进行MSS大小的TCP分段;
3. 以太网帧的 payload大于MTU进行IP分片。

主流解决方案:
1. 消息定长,例如每个报文的大小为固定长度200字节,如果不够,空位补空格
2. 在包尾增加回车换行符进行分割,例如FTP协议
3. 将消息分为消息头和消息体,消息头中包含表示消息总长度(或者消息体长度) 的字段,通常设计思路为消息头的第一个字段使用int32来表示消息的总长度;

项目解决方案:

通过DelimiterBasedFrameDecoder+StringDecoder解决粘包的问题

设置分隔符和获取数据的最大值来解决粘包问题

DelimiterBasedFrameDecode的工作原理是它依次遍历 Bytebuf中的可读字节,判断看是否有设置的分隔符（"\r\n"）,如果有,就以此位置为结束位置,从可读索引到结束位置区间的字节就组成了一行。它是以换行符为结束标志的解码器,支持携带结束符或者不携带结束符两种解码方式,同时支持配置单行的最大长度。如果连续读取到最大长度后仍然没有发现换行符,就会抛出异常,同时忽略掉之前读到的异常码流。

String Decoder的功能非常简单,就是将接收到的对象转换成字符串,然后继续调用后面的Handler 。DelimiterBasedFrameDecode+ String Decoder组合就是按行切换的文本解码器, 它被设计用来支持TCP的粘包和拆包

(9) Beanpostprocessor 机制:

Spirng 揭秘+secret 文档+(https://segmentfault.com/a/1190000015830477)博客，只要讲 明白原理，然后结合项目说清楚怎么使用的，即可(答的过程中肯定是先讲清楚如何使 用，然后涉及到一些原理，面试官如果追问原理，就 blabla，如果不追问，主动告诉面试 官因为我对该部分原理进行深入探讨，技术选型的根源源自 spring 相关机制的理解，然后 你就可以把面试中的一部分精力引道进 spring 相关的了，其他不用我说了把?)

完成对bean对象的动态代理（上面有提及）

在bean初始化之后，将有Controller注解的bean放到Media类中的HashMap里面

1. 先获取有Controller注解的bean
2. 然后获取bena的所有方法
3. bean方法全名的（bean的类+name+"."+方法名）
4. 把全名和方法一起放到Media类中的HashMap中

BeanPostProcessor

BeanPostProcessor会处理容器内所有符合条件的实例化后的对象实例，该接口中有两个方法，一个postProcessBeforeInitialization，另一个postProcessAfterInitialization

通过该接口可以对实例对象进行做更多的处理，例如替换当前的对象实例或者字节码增强当前的对象实例，String的AOP更多地使用BeanPostProcessor来为对象生成相对应的代理对象，BeanPostProcessor是容器提供的对象实例化阶段的强有力的扩展点。

e.g. 假设系统中所有的实现类需要从某个位置取得相应的服务器连接密码,而且系统中保存的密码是加密的,那么在发送这个密码给新闻服务器进行连接验证的时候,首先需要对系统中取得的密码进行解密,然后才能发送。我们将采用 Beanpostprocessor技术, 对所有的实现类进行统一的解密操作

(10)ApplicationListener 机制:

Spirng 揭秘+secret 文档+博客(http://www.kailing.pub/article/index/arcid/33.html)

通过ApplicationListener来监听spring初始化完毕事件（ContextRefreshedEvent），初始化完毕后，启动 netty的server

写一个类继承spring的ApplicationListener监听，并监控ContextRefreshedEvent事件
在spring中,容器管理所有的 bean。是ApplicationEvent 驱动的，一个ApplicationEvent publish了，观察这个事件的监听者就会送到通知
Spring的 ApplicationContext容器内部允许以ApplicationEvent的形式发布事件,容器内注册的ApplicationListener类型的bean定义会被 ApplicationContext容器自动识别,它们负责监听容器内发布的所有 ApplicationEvent类型的事件。也就是说,一旦容器内发布 ApplicationEvent及其子类型的事件, 注册到容器的ApplicationListener就会对这些事件进行处理。
ApplicationListener通过onApplicationEvent（）方法来处理事件

(11)项目其他:

项目性能如何?

比如你设置了个 user,rpc 调用 save 方法，1 万次 5s,10 万次 27s,100w 次 173s,机器性能 4 核 16g 笔记本。由于调用量过大，中间发生过 FULL GC，如果问到了就说，没问到就不 说。----注:这个数据我自己测试过，不用担心，你当成你自己的数据

性能优化:几个角度 网络通信框架 netty 已经够好了，那么从测试核性能角度，序列 化协议和 GC 参数(避免频繁 FULL GC)会数据表现更好，最好的办法是提升服务器的性 能，笔记本能提供的能力还是太单薄。

(12) 为什么做这个项目?
因为感觉普通的 curd 项目有时技术含量较低，作为技术人总有这对技术追求的热情，

于是不拘泥与之前的各种系统，决定通过对 rpc 框架和 netty 的学习，运动自己所学，实现 1个rpc框架。

netty相 关：回答:参考 netty 文档

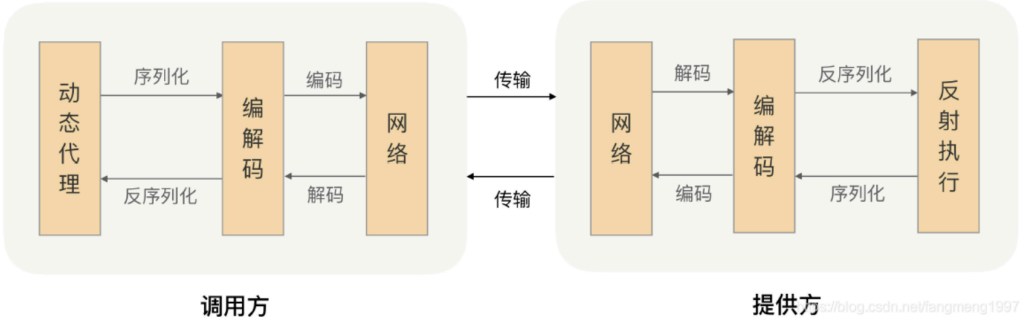
注:校招生掌握这些本身已经够了

rpc相 关：

回答:博客1:深入理解rpc | 博客2:深入理解rpc(2)
注:2 个博客中关于 rpc 的都要看，了解 rpc 框架究竟是什么一回事，哪些模块对 rpc 框架 来说是重点，这 2 个资料足够了

RPC 的全称是 Remote Procedure Call，即远程过程调用。
RPC 是帮助我们屏蔽网络编程细节，实现调用远程方法就跟调用本地一样的体验
RPC 一般默认采用 TCP 来传输
由服务提供者给出业务接口声明，在调用方的程序里面，RPC 框架根据调用的服务接口提前生成动态代理实现类，并通过依赖注入等技术注入到声明了该接口的相关业务逻辑里面。该代理实现类会拦截所有的方法调用，在提供的方法处理逻辑里面完成一整套的远程调用，并把远程调用结果返回给调用方，这样调用方在调用远程方法的时候就获得了像调用本地接口一样的体验。

简易流程



协议：

只有二进制才能在网络中传输，所以 RPC 请求在发送到网络中之前，他需要把方法调用的请求参数转成二进制；转成二进制后，写入本地 Socket 中，然后被网卡发送到网络设备中。为了避免语义不一致的事情发生，我们就需要在发送请求的时候设定一个边界，然后在收到请求的时候按照这个设定的边界进行数据分割。这个边界语义的表达，就是我们所说的协议。

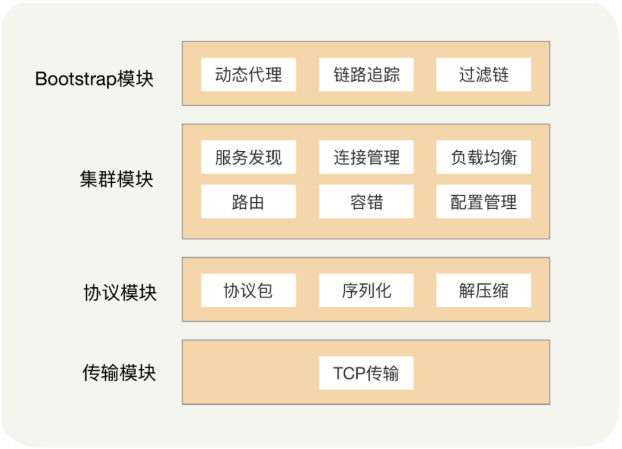
相对于 HTTP 的用处，RPC 更多的是负责应用间的通信，所以性能要求相对更高。但 HTTP 协议的数据包大小相对请求数据本身要大很多，又需要加入很多无用的内容，比如换行符号、回车符等；还有一个更重要的原因是，HTTP 协议属于无状态协议，客户端无法对请求和响应进行关联，每次请求都需要重新建立连接，响应完成后再关闭连接。因此，对于要求高性能的 RPC 来说，HTTP 协议基本很难满足需求，所以RPC 会选择设计更紧凑的私有协议。

Bit Offset	0-15	16-47	48-63	64-71	72-79	80-87
0	魔术位	整体长度	消息ID	协议版本	消息类型	序列化方式
协议体	Payload					

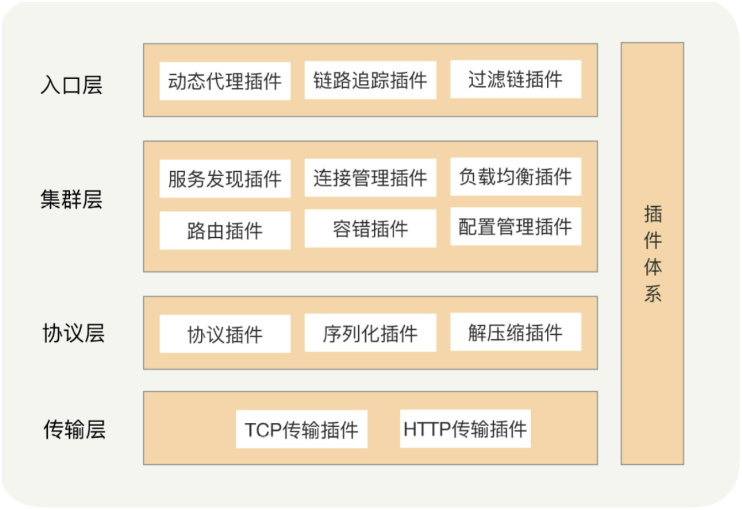
Bit Offset	0-15	16-47	48-63	64-71	72-79	80-87
0	魔术位	整体长度	头长度	协议版本	消息类型	序列化方式
88-103	消息ID					
不固定	协议头扩展字段					
协议体	Payload					

可扩展协议

架构



架构图



插件化RPC

服务发现：

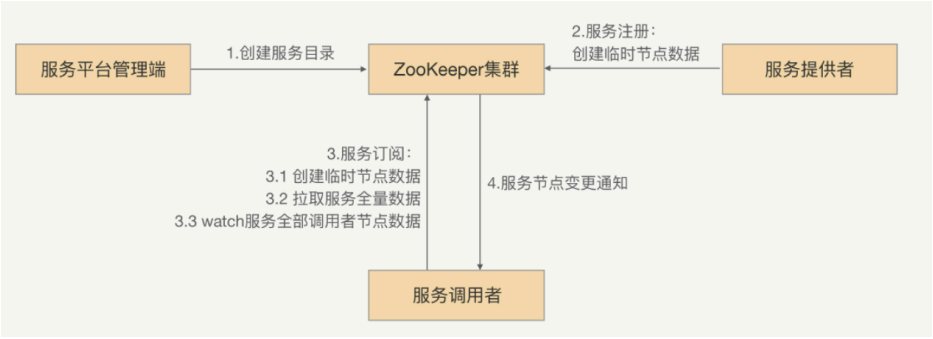
为了高可用，在生产环境中服务提供方都是以集群的方式对外提供服务，集群里面的这些 IP 随时可能变化，我们也需要用一本“通信录”及时获取到对应的服务节点，这个获取的过程我们一般叫作“服务发现”。

服务注册：在服务提供方启动的时候，将对外暴露的接口注册到注册中心之中，注册中心将这个服务节点的 IP 和接口保存下来。

服务订阅：在服务调用方启动的时候，去注册中心查找并订阅服务提供方的 IP，然后缓存到本地，并用于后续的远程调用。

基于 ZooKeeper 的服务发现

整体的思路很简单，就是搭建一个 ZooKeeper 集群作为注册中心集群，服务注册的时候只需要服务节点向 ZooKeeper 节点写入注册信息即可，利用 ZooKeeper 的 Watcher 机制完成服务订阅与服务下发功能



ZooKeeper 的一大特点就是强一致性，ZooKeeper 集群的每个节点的数据每次发生更新操作，都会通知其它 ZooKeeper 节点同时执行更新。它要求保证每个节点的数据能够实时的完全一致，这也就直接导致了 ZooKeeper 集群性能上的下降。

RPC 框架依赖的注册中心的服务数据的一致性其实并不需要满足 CP，只要满足 AP 即可。我们就是采用“消息总线”的通知机制，来保证注册中心数据的最终一致性，来解决这些问题的。

消息总线机制，注册数据可以全量缓存在每个注册中心内存中，通过消息总线来同步数据。当有一个注册中心节点接收到服务节点注册时，会产生一个消息推送给消息总线，再通过消息总线通知给其它注册中心节点更新数据并进行服务下发，从而达到注册中心间数据最终一致性

健康检测

因为有了集群，所以每次发请求前，RPC 框架会根据路由和负载均衡算法选择一个具体的 IP 地址。为了保证请求成功，我们就需要确保每次选择出来的 IP 对应的连接是健康的

可以通过心跳检测来判罚连接是否健康，但是不能从根本解决间歇性失败

可以通过可用率来选择健康的服务器，可用率的计算方式是某一个时间窗口内接口调用成功次数的百分比（成功次数 / 总调用次数）。当可用率低于某个比例就认为这个节点存在问题，把它挪到亚健康列表，这样既考虑了高低频的调用接口，也兼顾了接口响应时间不同的问题。

dubbo相 关：回答:参考 dubbo 文档

注:不难，easy

项目包装:

项目架构图，自己这个图一定好好梳理多遍，烂熟于心，自己能画出来!