01 | **到底什么是微服务？**

**单体应用**

在开聊微服务之前，我先要你和介绍下单体应用。如果你不知道单体应用的痛，那也不会深刻理解微服务的价值。

早些年，各大互联网公司的应用技术栈大致可分为 LAMP（Linux + Apache + MySQL + PHP）和 MVC（Spring + iBatis/Hibernate + Tomcat）两大流派。无论是 LAMP 还是 MVC，都是为单体应用架构设计的，其优点是学习成本低，开发上手快，测试、部署、运维也比较方便，甚至一个人就可以完成一个网站的开发与部署。

以 MVC 架构为例，业务通常是通过部署一个 WAR 包到 Tomcat 中，然后启动 Tomcat，监听某个端口即可对外提供服务。早期在业务规模不大、开发团队人员规模较小的时候，采用单体应用架构，团队的开发和运维成本都可控。

然而随着业务规模的不断扩大，团队开发人员的不断扩张，单体应用架构就会开始出现问题。我估计经历过业务和团队快速增长的同学都会对此深有感触。从我的角度来看，大概会有以下几个方面的问题。

**部署效率低下**。以我实际参与的项目为例，当单体应用的代码越来越多，依赖的资源越来越多时，应用编译打包、部署测试一次，甚至需要 10 分钟以上。这也经常被新加入的同学吐槽说，部署测试一次的时间，都可以去楼下喝杯咖啡了。

**团队协作开发成本高**。以我的经验，早期在团队开发人员只有两三个人的时候，协作修改代码，最后合并到同一个 master 分支，然后打包部署，尚且可控。但是一旦团队人员扩张，超过 5 人修改代码，然后一起打包部署，测试阶段只要有一块功能有问题，就得重新编译打包部署，然后重新预览测试，所有相关的开发人员又都得参与其中，效率低下，开发成本极高。

**系统高可用性差**。因为所有的功能开发最后都部署到同一个 WAR 包里，运行在同一个 Tomcat 进程之中，一旦某一功能涉及的代码或者资源有问题，那就会影响整个 WAR 包中部署的功能。比如我经常遇到的一个问题，某段代码不断在内存中创建大对象，并且没有回收，部署到线上运行一段时间后，就会造成 JVM 内存泄露，异常退出，那么部署在同一个 JVM 进程中的所有服务都不可用，后果十分严重。

**线上发布变慢**。特别是对于 Java 应用来说，一旦代码膨胀，服务启动的时间就会变长，有些甚至超过 10 分钟以上，如果机器规模超过 100 台以上，假设每次发布的步长为 10%，单次发布需要就需要 100 分钟之久。因此，急需一种方法能够将应用的不同模块的解耦，降低开发和部署成本。

想要解决上面这些问题，服务化的思想也就应运而生。

**什么是服务化？**

这里我就不谈一些官方的、教条主义的概念了。在我看来，用通俗的话来讲，服务化就是把传统的单机应用中通过 JAR 包依赖产生的本地方法调用，改造成通过 RPC 接口产生的远程方法调用。一般在编写业务代码时，对于一些通用的业务逻辑，我会尽力把它抽象并独立成为专门的模块，因为这对于代码复用和业务理解都大有裨益。

在过去的项目经历里，我对此深有体会。以微博系统为例，微博既包含了内容模块，也包含了消息模块和用户模块等。其中消息模块依赖内容模块，消息模块和内容模块又都依赖用户模块。当这三个模块的代码耦合在一起，应用启动时，需要同时去加载每个模块的代码并连接对应的资源。一旦任何模块的代码出现 bug，或者依赖的资源出现问题，整个单体应用都会受到影响。

为此，首先可以把用户模块从单体应用中拆分出来，独立成一个服务部署，以 RPC 接口的形式对外提供服务。微博和消息模块调用用户接口，就从进程内的调用变成远程 RPC 调用。这样，用户模块就可以独立开发、测试、上线和运维，可以交由专门的团队来做，与主模块不耦合。进一步的可以再把消息模块也拆分出来作为独立的模块，交由专门的团队来开发和维护。

可见通过服务化，可以解决单体应用膨胀、团队开发耦合度高、协作效率低下的问题。

**什么是微服务？**

从 2014 年开始，得益于以 Docker 为代表的容器化技术的成熟以及 DevOps 文化的兴起，服务化的思想进一步演化，演变为今天我们所熟知的微服务。

那么微服务相比于服务化又有什么不同呢？

在我看来，可以总结为以下四点：

**服务拆分粒度更细**。微服务可以说是更细维度的服务化，小到一个子模块，只要该模块依赖的资源与其他模块都没有关系，那么就可以拆分为一个微服务。

**服务独立部署**。每个微服务都严格遵循独立打包部署的准则，互不影响。比如一台物理机上可以部署多个 Docker 实例，每个 Docker 实例可以部署一个微服务的代码。

**服务独立维护**。每个微服务都可以交由一个小团队甚至个人来开发、测试、发布和运维，并对整个生命周期负责。

**服务治理能力要求高**。因为拆分为微服务之后，服务的数量变多，因此需要有统一的服务治理平台，来对各个服务进行管理。

继续以前面举的微博系统为例，可以进一步对内容模块的功能进行拆分，比如内容模块又包含了 feed 模块、评论模块和个人页模块。通过微服务化，将这三个模块变成三个独立的服务，每个服务依赖各自的资源，并独立部署在不同的服务池中，可以由不同的开发人员进行维护。当评论服务需求变更时，只需要修改评论业务相关的代码，并独立上线发布；而 feed 服务和个人页服务不需要变更，也不会受到发布可能带来的变更影响。

由此可见，微服务化给服务的发布和部署，以及服务的保障带来了诸多好处。

**02 | 从单体应用走向服务化**

**服务化拆分的两种姿势**

那么服务化拆分具体该如何实施呢？一个最有效的手段就是将**不同的功能模块服务化**，独立部署和运维。以前面提到的社交 App 为例，你可以认为首页信息流是一个服务，评论是一个服务，消息通知是一个服务，个人主页也是一个服务。

这种服务化拆分方式是**纵向拆分**，是**从业务维度进行拆分**。标准是按照业务的关联程度来决定，关联比较密切的业务适合拆分为一个微服务，而功能相对比较独立的业务适合单独拆分为一个微服务。

还有一种服务化拆分方式是**横向拆分，**是**从公共且独立功能维度**拆分。标准是按照是否有公共的被多个其他服务调用，且依赖的资源独立不与其他业务耦合。

继续以前面提到的社交 App 举例，无论是首页信息流、评论、消息箱还是个人主页，都需要显示用户的昵称。假如用户的昵称功能有产品需求的变更，你需要上线几乎所有的服务，这个成本就有点高了。显而易见，如果我把用户的昵称功能单独部署成一个独立的服务，那么有什么变更我只需要上线这个服务即可，其他服务不受影响，开发和上线成本就大大降低了。

**服务化拆分的前置条件**

一般情况下，业务系统引入新技术就必然会带来架构的复杂度提升，在具体决策前，你先要认识到新架构会带来哪些新的问题，这些问题你和你的团队是否能够解决？如何解决？是自己投入人力建设，还是采用业界开源方案？

下面几个问题，是从单体应用迁移到微服务架构时必将面临也必须解决的。

**服务如何定义**。对于单体应用来说，不同功能模块之前相互交互时，通常是以类库的方式来提供各个模块的功能。对于微服务来说，每个服务都运行在各自的进程之中，应该以何种形式向外界传达自己的信息呢？答案就是接口，无论采用哪种通讯协议，是 HTTP 还是 RPC，服务之间的调用都通过接口描述来约定，约定内容包括接口名、接口参数以及接口返回值。

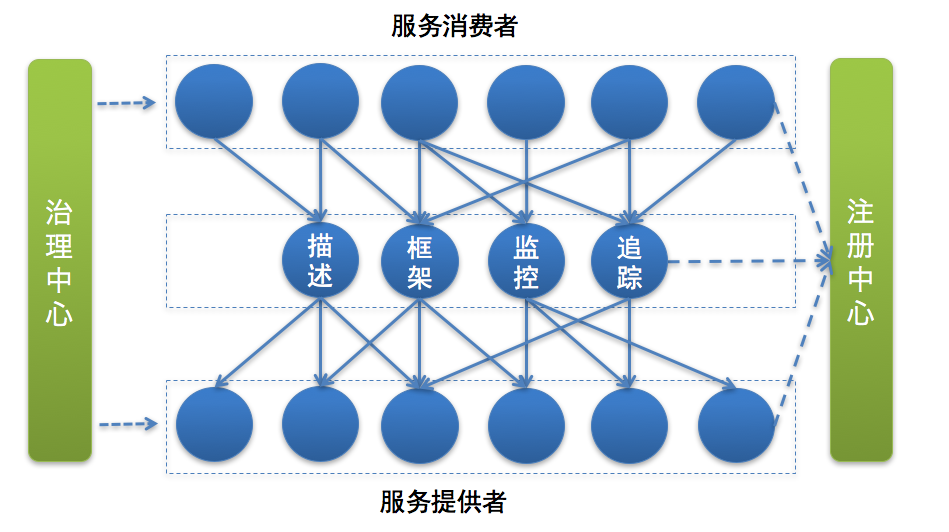
**服务如何发布和订阅**。单体应用由于部署在同一个 WAR 包里，接口之间的调用属于进程内的调用。而拆分为微服务独立部署后，服务提供者该如何对外暴露自己的地址，服务调用者该如何查询所需要调用的服务的地址呢？这个时候你就需要一个类似登记处的地方，能够记录每个服务提供者的地址以供服务调用者查询，在微服务架构里，这个地方就是注册中心。

**服务如何监控**。通常对于一个服务，我们最关心的是 QPS（调用量）、AvgTime（平均耗时）以及 P999（99.9% 的请求性能在多少毫秒以内）这些指标。这时候你就需要一种通用的监控方案，能够覆盖业务埋点、数据收集、数据处理，最后到数据展示的全链路功能。

**服务如何治理**。可以想象，拆分为微服务架构后，服务的数量变多了，依赖关系也变复杂了。比如一个服务的性能有问题时，依赖的服务都势必会受到影响。可以设定一个调用性能阈值，如果一段时间内一直超过这个值，那么依赖服务的调用可以直接返回，这就是熔断，也是服务治理最常用的手段之一。

**故障如何定位**。在单体应用拆分为微服务之后，一次用户调用可能依赖多个服务，每个服务又部署在不同的节点上，如果用户调用出现问题，你需要有一种解决方案能够将一次用户请求进行标记，并在多个依赖的服务系统中继续传递，以便串联所有路径，从而进行故障定位。

**03 | 初探微服务架构**



微服务架构下，服务调用主要依赖下面几个基本组件：

**服务描述:**RESTful API 方式通常用于 HTTP 协议的服务描述，并且常用 Wiki 或者Swagger来进行管理

**注册中心:**

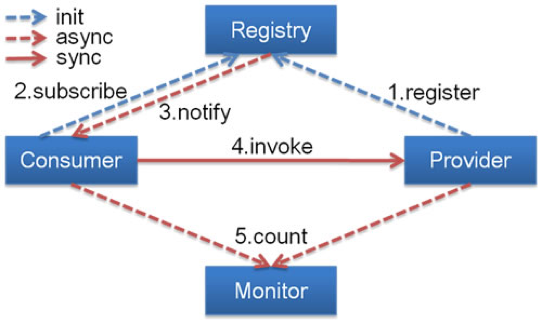
一般来讲，注册中心的工作流程是：

服务提供者在启动时，根据服务发布文件中配置的发布信息向注册中心注册自己的服务。

服务消费者在启动时，根据消费者配置文件中配置的服务信息向注册中心订阅自己所需要的服务。

注册中心返回服务提供者地址列表给服务消费者。

当服务提供者发生变化，比如有节点新增或者销毁，注册中心将变更通知给服务消费者。



**服务框架:**

通过注册中心，服务消费者就可以获取到服务提供者的地址，有了地址后就可以发起调用。但在发起调用之前你还需要解决以下几个问题。

服务通信采用什么协议？就是说服务提供者和服务消费者之间以什么样的协议进行网络通信，是采用四层 TCP、UDP 协议，还是采用七层 HTTP 协议，还是采用其他协议？

数据传输采用什么方式？就是说服务提供者和服务消费者之间的数据传输采用哪种方式，是同步还是异步，是在单连接上传输，还是多路复用。

数据压缩采用什么格式？通常数据传输都会对数据进行压缩，来减少网络传输的数据量，从而减少带宽消耗和网络传输时间，比如常见的 JSON 序列化、Java 对象序列化以及 Protobuf 序列化等。

**服务监控:**

一旦服务消费者与服务提供者之间能够正常发起服务调用，你就需要对调用情况进行监控，以了解服务是否正常。通常来讲，服务监控主要包括三个流程。

指标收集。就是要把每一次服务调用的请求耗时以及成功与否收集起来，并上传到集中的数据处理中心。

数据处理。有了每次调用的请求耗时以及成功与否等信息，就可以计算每秒服务请求量、平均耗时以及成功率等指标。

数据展示。数据收集起来，经过处理之后，还需要以友好的方式对外展示，才能发挥价值。通常都是将数据展示在 Dashboard 面板上，并且每隔 10s 等间隔自动刷新，用作业务监控和报警等。

**服务追踪:**

除了需要对服务调用情况进行监控之外，你还需要记录服务调用经过的每一层链路，以便进行问题追踪和故障定位。

服务追踪的工作原理大致如下：

服务消费者发起调用前，会在本地按照一定的规则生成一个 requestid，发起调用时，将 requestid 当作请求参数的一部分，传递给服务提供者。

服务提供者接收到请求后，记录下这次请求的 requestid，然后处理请求。如果服务提供者继续请求其他服务，会在本地再生成一个自己的 requestid，然后把这两个 requestid 都当作请求参数继续往下传递。

以此类推，通过这种层层往下传递的方式，一次请求，无论最后依赖多少次服务调用、经过多少服务节点，都可以通过最开始生成的 requestid 串联所有节点，从而达到服务追踪的目的。

**服务治理:**

服务监控能够发现问题，服务追踪能够定位问题所在，而解决问题就得靠服务治理了。服务治理就是通过一系列的手段来保证在各种意外情况下，服务调用仍然能够正常进行。

在生产环境中，你应该经常会遇到下面几种状况。

单机故障。通常遇到单机故障，都是靠运维发现并重启服务或者从线上摘除故障节点。然而集群的规模越大，越是容易遇到单机故障，在机器规模超过一百台以上时，靠传统的人肉运维显然难以应对。而服务治理可以通过一定的策略，自动摘除故障节点，不需要人为干预，就能保证单机故障不会影响业务。

单 IDC 故障。你应该经常听说某某 App，因为施工挖断光缆导致大批量用户无法使用的严重故障。而服务治理可以通过自动切换故障 IDC 的流量到其他正常 IDC，可以避免因为单 IDC 故障引起的大批量业务受影响。

依赖服务不可用。比如你的服务依赖依赖了另一个服务，当另一个服务出现问题时，会拖慢甚至拖垮你的服务。而服务治理可以通过熔断，在依赖服务异常的情况下，一段时期内停止发起调用而直接返回。这样一方面保证了服务消费者能够不被拖垮，另一方面也给服务提供者减少压力，使其能够尽快恢复。

**04 | 如何发布和引用服务？**

最常见的服务发布和引用的方式有三种：

RESTful API

XML 配置

IDL 文件



**05 | 如何注册和发现服务？**

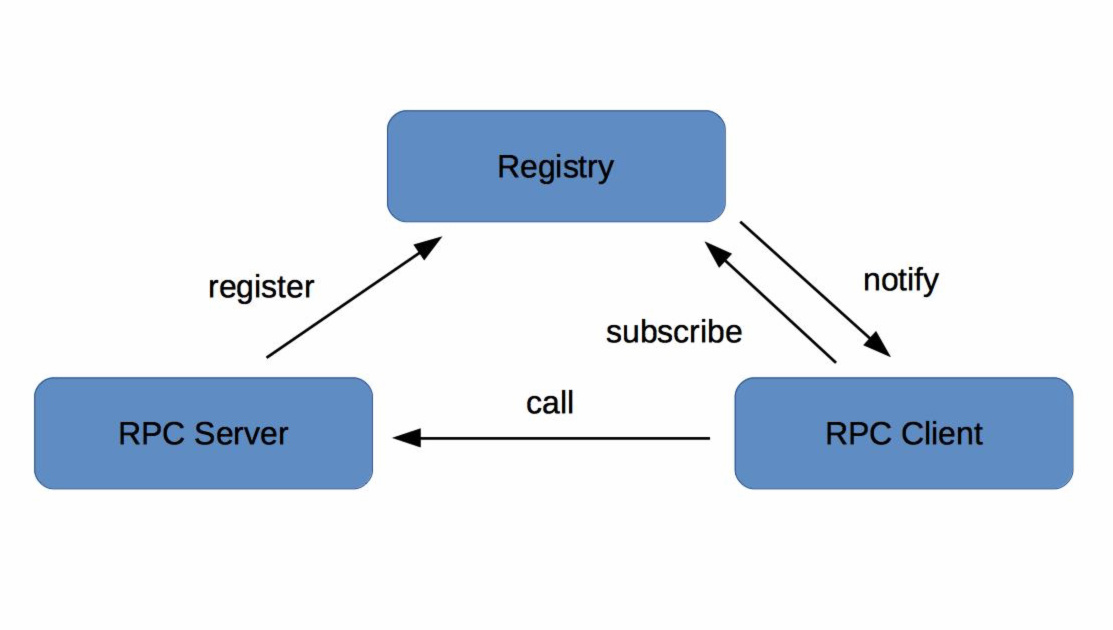
注册中心原理

在微服务架构下，主要有三种角色：服务提供者（RPC Server）、服务消费者（RPC Client）和服务注册中心（Registry），三者的交互关系请看下面这张图，我来简单解释一下。

RPC Server 提供服务，在启动时，根据服务发布文件 server.xml 中的配置的信息，向 Registry 注册自身服务，并向 Registry 定期发送心跳汇报存活状态。

RPC Client 调用服务，在启动时，根据服务引用文件 client.xml 中配置的信息，向 Registry 订阅服务，把 Registry 返回的服务节点列表缓存在本地内存中，并与 RPC Sever 建立连接。

当 RPC Server 节点发生变更时，Registry 会同步变更，RPC Client 感知后会刷新本地内存中缓存的服务节点列表。

RPC Client 从本地缓存的服务节点列表中，基于负载均衡算法选择一台 RPC Sever 发起调

**注册中心实现方式**

注册中心的实现主要涉及几个问题：注册中心需要提供哪些接口，该如何部署；如何存储服务信息；如何监控服务提供者节点的存活；如果服务提供者节点有变化如何通知服务消费者，以及如何控制注册中心的访问权限。

注册中心可以说是实现服务化的关键，因为服务化之后，服务提供者和服务消费者不在同一个进程中运行，实现了解耦，这就需要一个纽带去连接服务提供者和服务消费者，而注册中心就正好承担了这一角色。此外，服务提供者可以任意伸缩即增加节点或者减少节点，通过服务健康状态检测，注册中心可以保持最新的服务节点信息，并将变化通知给订阅服务的服务消费者。

注册中心一般采用分布式集群部署，来保证高可用性，并且为了实现异地多活，有的注册中心还采用多 IDC 部署，这就对数据一致性产生了很高的要求，这些都是注册中心在实现时必须要解决的问题。

**06 | 如何实现RPC远程服务调用？**

在介绍 RPC 调用的原理之前，先来想象一下一次电话通话的过程。首先，呼叫者 A 通过查询号码簿找到被呼叫者 B 的电话号码，然后拨打 B 的电话。B 接到来电提示时，如果方便接听的话就会接听；如果不方便接听的话，A 就得一直等待。当等待超过一段时间后，电话会因超时被挂断，这个时候 A 需要再次拨打电话，一直等到 B 空闲的时候，才能接听。

RPC 调用的原理与此类似，我习惯把服务消费者叫作客户端，服务提供者叫作服务端，两者通常位于网络上两个不同的地址，要完成一次 RPC 调用，就必须先建立网络连接。建立连接后，双方还必须按照某种约定的协议进行网络通信，这个协议就是通信协议。双方能够正常通信后，服务端接收到请求时，需要以某种方式进行处理，处理成功后，把请求结果返回给客户端。为了减少传输的数据大小，还要对数据进行压缩，也就是对数据进行序列化。(序列化---将对象转换成字节码（二进制）存入硬盘或者用于网络传输，反序列--将字节码转换成对象 使用工具io流)

RPC 调用的过程，需要解决四个问题：

客户端和服务端如何建立网络连接？

服务端如何处理请求？

数据传输采用什么协议？

数据该如何序列化和反序列化？

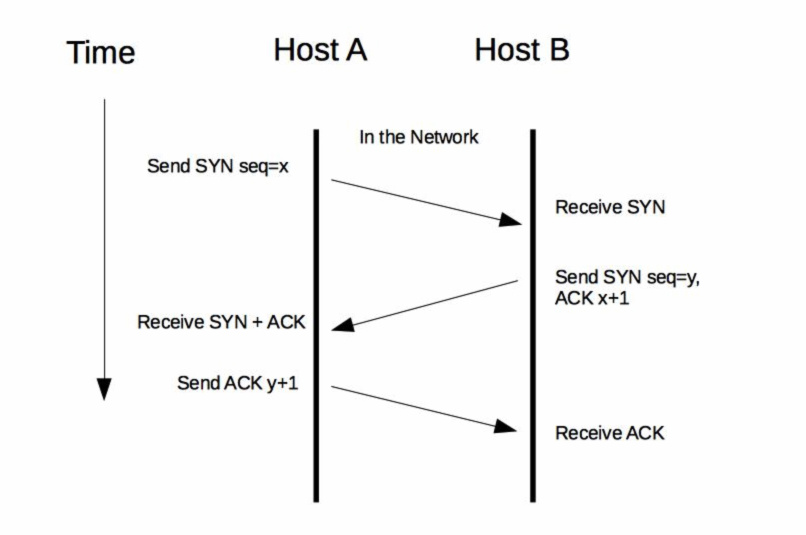
客户端和服务端如何建立网络连接？

根据我的实践经验，客户端和服务端之间基于 TCP 协议建立网络连接最常用的途径有两种。

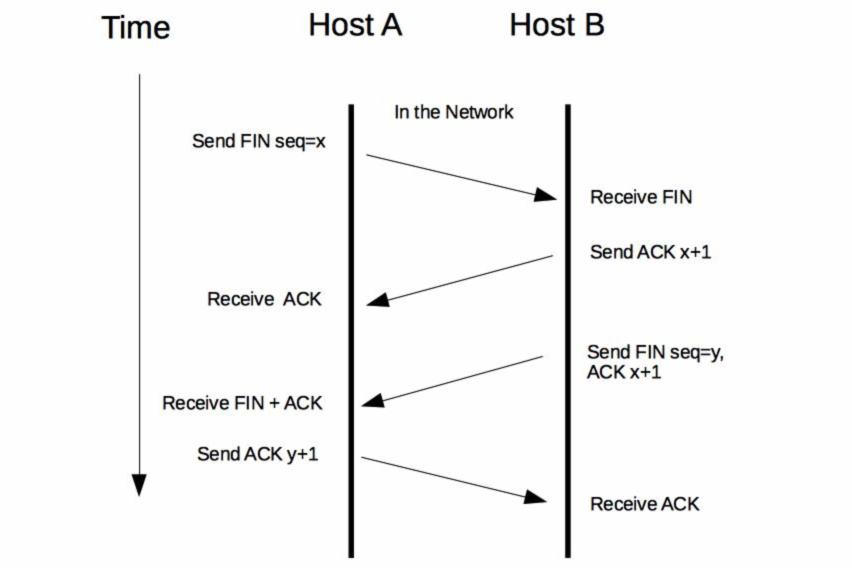
**1. HTTP 通信**

HTTP 通信是基于应用层 HTTP 协议的，而 HTTP 协议又是基于传输层 TCP 协议的。一次 HTTP 通信过程就是发起一次 HTTP 调用，而一次 HTTP 调用就会建立一个 TCP 连接，经历一次下图所示的“三次握手”的过程来建立连接。

“三次握手”（建立tcp连接）（需要百度学习下）



“四次挥手”（断开tcp连接）



**2. Socket 通信**

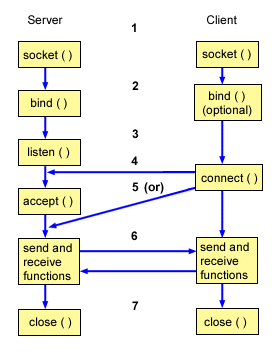
Socket 通信是基于 TCP/IP 协议的封装，建立一次 Socket 连接至少需要一对套接字，其中一个运行于客户端，称为 ClientSocket ；另一个运行于服务器端，称为 ServerSocket 。就像下图所描述的，Socket 通信的过程分为四个步骤：服务器监听、客户端请求、连接确认、数据传输。

**服务器监听**：ServerSocket 通过调用 bind() 函数绑定某个具体端口，然后调用 listen() 函数实时监控网络状态，等待客户端的连接请求。

**客户端请求**：ClientSocket 调用 connect() 函数向 ServerSocket 绑定的地址和端口发起连接请求。

**服务端连接确认**：当 ServerSocket 监听到或者接收到 ClientSocket 的连接请求时，调用 accept() 函数响应 ClientSocket 的请求，同客户端建立连接。

**数据传输**：当 ClientSocket 和 ServerSocket 建立连接后，ClientSocket 调用 send() 函数，ServerSocket 调用 receive() 函数，ServerSocket 处理完请求后，调用 send() 函数，ClientSocket 调用 receive() 函数，就可以得到得到返回结果。



当客户端和服务端建立网络连接后，就可以发起请求了。但网络不一定总是可靠的，经常会遇到网络闪断、连接超时、服务端宕机等各种异常，通常的处理手段有两种。

**链路存活检测**：客户端需要定时地发送心跳检测消息（一般是通过 ping 请求）给服务端，如果服务端连续 n 次心跳检测或者超过规定的时间都没有回复消息，则认为此时链路已经失效，这个时候客户端就需要重新与服务端建立连接。

**断连重试**：通常有多种情况会导致连接断开，比如客户端主动关闭、服务端宕机或者网络故障等。这个时候客户端就需要与服务端重新建立连接，但一般不能立刻完成重连，而是要等待固定的间隔后再发起重连，避免服务端的连接回收不及时，而客户端瞬间重连的请求太多而把服务端的连接数占满。

同步和异步的概念描述的是用户线程与内核的交互方式：同步是指用户线程发起IO请求后需要等待或者轮询内核IO操作完成后才能继续执行；而异步是指用户线程发起IO请求后仍继续执行，当内核IO操作完成后会通知用户线程，或者调用用户线程注册的回调函数。

阻塞和非阻塞的概念描述的是用户线程调用内核IO操作的方式：阻塞是指IO操作需要彻底完成后才返回到用户空间；而非阻塞是指IO操作被调用后立即返回给用户一个状态值，无需等到IO操作彻底完成。

**服务端如何处理请求？**

假设这时候客户端和服务端已经建立了网络连接，服务端又该如何处理客户端的请求呢？通常来讲，有三种处理方式。

同步阻塞方式（BIO），客户端每发一次请求，服务端就生成一个线程去处理。当客户端同时发起的请求很多时，服务端需要创建很多的线程去处理每一个请求，如果达到了系统最大的线程数瓶颈，新来的请求就没法处理了。

同步非阻塞方式 (NIO)，客户端每发一次请求，服务端并不是每次都创建一个新线程来处理，而是通过 I/O 多路复用技术进行处理。就是把多个 I/O 的阻塞复用到同一个 select 的阻塞上，从而使系统在单线程的情况下可以同时处理多个客户端请求。这种方式的优势是开销小，不用为每个请求创建一个线程，可以节省系统开销。

异步非阻塞方式（AIO），客户端只需要发起一个 I/O 操作然后立即返回，等 I/O 操作真正完成以后，客户端会得到 I/O 操作完成的通知，此时客户端只需要对数据进行处理就好了，不需要进行实际的 I/O 读写操作，因为真正的 I/O 读取或者写入操作已经由内核完成了。这种方式的优势是客户端无需等待，不存在阻塞等待问题。

从前面的描述，可以看出来不同的处理方式适用于不同的业务场景，根据我的经验：

BIO 适用于连接数比较小的业务场景，这样的话不至于系统中没有可用线程去处理请求。这种方式写的程序也比较简单直观，易于理解。

NIO 适用于连接数比较多并且请求消耗比较轻的业务场景，比如聊天服务器。这种方式相比 BIO，相对来说编程比较复杂。

AIO 适用于连接数比较多而且请求消耗比较重的业务场景，比如涉及 I/O 操作的相册服务器。这种方式相比另外两种，编程难度最大，程序也不易于理解。

上面两个问题就是“通信框架”要解决的问题，你可以基于现有的 Socket 通信，在服务消费者和服务提供者之间建立网络连接，然后在服务提供者一侧基于 BIO、NIO 和 AIO 三种方式中的任意一种实现服务端请求处理，最后再花费一些精力去解决服务消费者和服务提供者之间的网络可靠性问题。这种方式对于 Socket 网络编程、多线程编程知识都要求比较高，感兴趣的话可以尝试自己实现一个通信框架。但我建议最为稳妥的方式是使用成熟的开源方案，比如 Netty、MINA 等，它们都是经过业界大规模应用后，被充分论证是很可靠的方案。

假设客户端和服务端的连接已经建立了，服务端也能正确地处理请求了，接下来完成一次正常地 RPC 调用还需要解决两个问题，即数据传输采用什么协议以及数据该如何序列化和反序列化。

**数据传输采用什么协议？**

首先来看第一个问题，数据传输采用什么协议？

最常用的有 HTTP 协议，它是一种开放的协议，各大网站的服务器和浏览器之间的数据传输大都采用了这种协议。还有一些定制的私有协议，比如阿里巴巴开源的 Dubbo 协议，也可以用于服务端和客户端之间的数据传输。无论是开放的还是私有的协议，都必须定义一个“契约”，以便服务消费和服务提供者之间能够达成共识。服务消费者按照契约，对传输的数据进行编码，然后通过网络传输过去；服务提供者从网络上接收到数据后，按照契约，对传输的数据进行解码，然后处理请求，再把处理后的结果进行编码，通过网络传输返回给服务消费者；服务消费者再对返回的结果进行解码，最终得到服务提供者处理后的返回值。

通常协议契约包括两个部分：消息头和消息体。其中消息头存放的是协议的公共字段以及用户扩展字段，消息体存放的是传输数据的具体内容。

以 HTTP 协议为例，下图展示了一段采用 HTTP 协议传输的数据响应报文，主要分为消息头和消息体两部分，其中消息头中存放的是协议的公共字段，比如 Server 代表是服务端服务器类型、Content-Length 代表返回数据的长度、Content-Type 代表返回数据的类型；消息体中存放的是具体的返回结果，这里就是一段 HTML 网页代码。

**数据该如何序列化和反序列化？**

再看第二个问题，数据该如何序列化和反序列化。

一般数据在网络中进行传输前，都要先在发送方一端对数据进行编码，经过网络传输到达另一端后，再对数据进行解码，这个过程就是序列化和反序列化。

为什么要对数据进行序列化和反序列化呢？要知道网络传输的耗时一方面取决于网络带宽的大小，另一方面取决于数据传输量。要想加快网络传输，要么提高带宽，要么减小数据传输量，而对数据进行编码的主要目的就是减小数据传输量。比如一部高清电影原始大小为 30GB，如果经过特殊编码格式处理，可以减小到 3GB，同样是 100MB/s 的网速，下载时间可以从 300s 减小到 30s。

常用的序列化方式分为两类：文本类如 XML/JSON 等，二进制类如 PB/Thrift 等，而具体采用哪种序列化方式，主要取决于三个方面的因素。

支持数据结构类型的丰富度。数据结构种类支持的越多越好，这样的话对于使用者来说在编程时更加友好，有些序列化框架如 Hessian 2.0 还支持复杂的数据结构比如 Map、List 等。

跨语言支持。序列化方式是否支持跨语言也是一个很重要的因素，否则使用的场景就比较局限，比如 Java 序列化只支持 Java 语言，就不能用于跨语言的服务调用了。

性能。主要看两点，一个是序列化后的压缩比，一个是序列化的速度。以常用的 PB 序列化和 JSON 序列化协议为例来对比分析，PB 序列化的压缩比和速度都要比 JSON 序列化高很多，所以对性能和存储空间要求比较高的系统选用 PB 序列化更合适；而 JSON 序列化虽然性能要差一些，但可读性更好，更适合对外部提供服务。

**一个完整的 RPC 调用框架**

通信框架。它主要解决客户端和服务端如何建立连接、管理连接以及服务端如何处理请求的问题。

通信协议。它主要解决客户端和服务端采用哪种数据传输协议的问题。

序列化和反序列化。它主要解决客户端和服务端采用哪种数据编解码的问题。