1. 什么是rpc？

Rpc是远程过度调用，是分布式系统常见的一种通信方法，它允许程序调用另一个地址空间的过程或函数，而不用程序显示编码这个远程调用的细节。

简单来说，rpc就是从一台机器上通过参数传递的方式调用另一台机器上的一个函数或方法并得到的返回结果。

1. 为什么我们要用RPC？

Rpc的主要目标是构建分布式应用更容易，在提供强大的远程调用能力时不损失本地调用的语义简洁性。为实现该目标，prc框架需提供一种透明调用机制让使用者不必显式的区分本地调用和远程调用。

1. Rpc需要解决的三个问题？
2. Call ID映射。在本地调用时，函数体是直接通过函数指针来指定的，我们调用具体函数，编译器会自动帮我们调用相应的函数指针。但是在远程调用中，无法采用这种方法，两个进程的地址空间是完全不一样的。

所以在RPC中，所有的函数都必须有一个自己的ID，这个id在所有的进程中都是唯一确定的。客户端在做远程调用时，必须附上这个id。然后我们还需要在客户端和服务端分别维护一个{函数 Call ID}对应表，两者的表不一定相同，但是相应的函数对应的call id必须相同。当客户端需要进行远程调用时，它就查一下这个表，找出相应的Call Id，然后把它传给服务端，服务端也通过查表，，来确定客户端需要调用的函数，然后执行相应函数的代码。

* **序列化和反序列化**。客户端怎么把参数值传给远程的函数呢？在本地调用中，我们只需要把参数压到栈里，然后让函数自己去栈里读就行。但是在远程过程调用时，客户端跟服务端是不同的进程，**不能通过内存来传递参数**。甚至有时候客户端和服务端使用的都**不是同一种语言**（比如服务端用C++，客户端用Java或者Python）。这时候就需要客户端把参数先转成一个字节流，传给服务端后，再把字节流转成自己能读取的格式。这个过程叫序列化和反序列化。同理，从服务端返回的值也需要序列化反序列化的过程。
* **网络传输**。远程调用往往是基于网络的，客户端和服务端是通过网络连接的。所有的数据都需要通过网络传输，因此就需要有一个网络传输层。网络传输层需要把Call ID和序列化后的参数字节流传给服务端，然后再把序列化后的调用结果传回客户端。只要能完成这两者的，都可以作为传输层使用。因此，它所使用的协议其实是不限的，能完成传输就行。尽管大部分RPC框架都使用TCP协议，但其实UDP也可以，而gRPC干脆就用了HTTP2。Java的Netty也属于这层的东西。

3 实现高可用RPC框架需要考虑到的问题？

采用分布式架构，那么一个服务势必会有多个实例，要解决获取实例的问题，所以需要一个服务注册中心。比如在Dubbo中，就可以使用**Zookeeper作为注册中心**。

**如果每次都去注册中心查询列表，效率很低，那么就要加缓存；**

**客户端总不能每次调用完都等着服务端返回数据，所以就要支持异步调用；**

**服务端的接口修改了，老的接口还有人在用，这就需要版本控制；**

**服务端总不能每次接到请求都马上启动一个线程去处理，于是就需要线程池；**

**Zookeeper是可以集群复制的，集群间通过zab协议来保持数据的一致性。**

该协议包括两个阶段：leader election和atomic broadcast阶段。集群中将选举出一个leader，其他机器则成为follow而，所有的写操作都被传送给leader，并通过broadcast将所有更新告诉follower。当leader崩溃或者leader失去大多数的follower，需要重新选举一个新的leader，让所有的服务器都恢复到一个正确的状态。当leader被选举出来，且大多数服务器完成了和leader的状态同步后，leader election的过程就结束了，将进入atomic broadcast的过程。Actomic broadcast同步leader和follower之间的信息，保证leader和follow而具备相同的系统状态。

具体的消息广播流程：

在zk集群中，数据副本之间的传输就是采用消息广播

1. 客户端发起一个写操作
2. Leader服务器将客户端的请求转为事提案，同时为每一个事务分配一个全局id。
3. Leader为每一个follower服务器分配一个单独的队列，然后将需要广播的事务依次放入队列中去取，并且根据fifo进行消息发送
4. Follower服务器接收到事务后，会首先以事务日志的方式写入到本地磁盘中，写入成功后向leader反馈一个ack响应消息。
5. Leader接收到超过半数以上的follower的ack响应消息后，认为消息发送成功
6. Leader向所有的follwe广播commit消息，同时自身也完成事务的提交。

ZZID是一个64位的事务id，低32位可以看作成一个简单的递增计数器，每一个事务请求，都会对该事务计数器加1；而高32位用于表示每代leader的唯一性；高32位代表了leader本地事务的最大zxid，然后从该zxid中取解析出epoch值，然后+1.

**一旦服务器与ZooKeeper集群断开连接，节点也就不存在了，通过注册相应的watcher，服务消费者能够第一时间获知服务提供者机器信息的变更。利用其znode的特点和watcher机制，将其作为动态注册和获取服务信息的配置中心，统一管理服务名称和其对应的服务器列表信息，我们能够近乎实时地感知到后端服务器的状态(上线、下线、宕机)。Zookeeper集群间通过Zab协议，服务配置信息能够保持一致，而Zookeeper本身容错特性和leader选举机制，能保证我们方便地进行扩容。**

Zookeeper中，服务提供者在启动时，将其提供的服务名称、服务器地址、以节点的形式注册到服务配置中心，服务消费者通过服务配置中心来获得需要调用的服务名称节点下的机器列表节点。通过前面所介绍的负载均衡算法，选取其中一台服务器进行调用。当服务器宕机或者下线时，由于znode非持久的特性，相应的机器可以动态地从服务配置中心里面移除，并触发服务消费者的watcher。在这个过程中，服务消费者只有在第一次调用服务时需要查询服务配置中心，然后将（查询到的服务信息缓存到本地），后面的调用直接使用本地缓存的服务地址列表信息，而不需要重新发起请求到服务配置中心去获取相应的服务地址列表，直到服务的地址列表有变更(机器上线或者下线)，变更行为会触发服务消费者注册的相应的watcher进行服务地址的重新查询。这种无中心化的结构，使得服务消费者在服务信息没有变更时，几乎不依赖配置中心，解决了之前负载均衡设备所导致的单点故障的问题，并且大大降低了服务配置中心的压力。

分布式事务

分布式事务要满足CAP定理，

C：一致性，对于某个客户端，读操作能返回最新的写操作

A：可用性：非故障节点在合理时间内返回合理的返回

P：分区容错性：当网络出现故障时候，系统仍然能够继续工作

两阶段提交：

XA协议包括两个角色：事务协调者和事务参与者。

1. 第一阶段

在第一阶段，事务协调者向所有的参与者发送prepare请求。在接到prepare请求后，每一个参与者节点会各自执行与事务有关的数据更新；如果事务参与者更新成功，暂时不提交事务，而是向事务协调者返回“完成信息”

1. 第二阶段

如果事务协调者在之前收到的都是正向返回，那么它将会向所有事务参与者发出commit请求。接收到commit请求后，事务参与者会各自本地提交，并释放锁资源。当本地事务完成提交后，将会向事务协调者返回“完成”信息。

如果失败了，在XA第一阶段，如果某个事务反馈失败消息，说明该节点本地执行不成功，必须回滚。于是在第二阶段，事务协调者会向所有的参与者发送Abort请求，各个事务参与者在本地进行事务回滚操作。

XA三阶段提交

在第二阶段提交上引入了can commit；并引入了超时机制。一旦事务参与者迟迟没有接收到协调者的commit请求，会自动进行本次commit。

一致性hash算法？

场景描述;

假设我们有三台服务器，用于缓存图片，我们现在如果缓存的图片很多，想让其平均的分配到每个服务器中，让各个服务器分摊缓存的压力。如果简单的来实现的话我们可以使用hash取模的方法。Hash（3）%n

但是这个其实有很大的弊端，如果我们当前的服务器新增一台或者崩溃一台，就会导致我们原先的存储位置可能都要发生变化。导致大量缓存在同一个时间失效（缓存雪崩），所有的请求都落在了数据库上，导致整个系统可能崩溃掉。

一致性hash算法不是对服务器的数量取模，而是对2^32取模。钟表的圆有60个节点，我们这个圆有2^32个节点。我们把这个圆环称之为hash环。

当有一个写入缓存的请求到来时，计算Key值k对应的哈希值Hash(k)，如果该值正好对应之前某个机器节点的Hash值，则直接写入该机器节点，如果没有对应的机器节点，则顺时针查找下一个节点，进行写入，如果超过2^32还没找到对应节点，则从0开始查找(因为是环状结构)。

中途就算有的机器挂点，也不会影响别的机器中缓存所在的位置，但是现在还是有一个问题，hash环倾斜。我们的缓存可能会不均匀的分布在不同的机器上，导致某些机器的压力比较大。

在一致性hash算法里面我们使用“虚拟节点”来解决这个问题。如果我们想使缓存均匀地分布在服务器上，最好让我们目前的服务器尽可能多的，均匀的出现在hash环上。我们可以将这些实际节点虚拟复制而来的节点加入到hash环中。