# 一.分布式理论

## 分布式和集群的区别?

集群：多个人在一起作同样的事。
分布式：多个人在一起作不同的事。

## 水平拆分和垂直拆分的区别?

垂直拆分是把不同的表拆到不同的数据库中，而水平拆分是把同一个表拆到不同的数据库中。

## 分布式系统面临的问题?

- 通信异常导致消息丢失和消息延迟
- 网络不通导致数据不一致
- 机器有可能经常出现故障
- 出现三态，即成功、失败、超时

## 什么是分布式一致性?

指的是数据在多份副本中存储时，各副本中的数据是一致的

## 一致性分类?

- **强一致性**

  系统写入什么，读出来的也会是什么，当节点A数据被修改时数据没同步到节点B，节点B会被锁从而不能读取。（对性能要求高）

- **弱一致性**

  只能保证在某个时间一致

  弱一致性分类：

  **1.读写一致性**

  A用户更新，A永远看到的是最新的，B用户不一定是最新的

  **解决方案:**

  方案1：一种方案是对于一些特定的内容我们每次都去主库读取。（问题主库压力大）

  方案2：更新时读主库，后面读从库

  方案3：更新上传时间戳，请求时间小于时间戳不给响应

  **2.单调读一致性**

  本次读不能的数据不能比上次读的旧（有可能读的数据库1是刚更改的结果，再一次发起读，读的是数据库2，这时数据库2还没同步成功，于是第二次读的是旧的数据）

  **解决方案:**

  方案1：根据hash映射到机器，保证一直读的是同一个数据库

- **因果一致性**

  如果节点 A 在更新完某个数据后通知了节点 B，那么节点 B 之后对该数据的访问和修改都是基于 A 更新后的值。与此同时，和节点 A 无因果关系的节点 C 的数据访问则没有这样的限制。

- **最终一致性**

  最终一致性是所有分布式一致性模型当中最弱的。可以认为是没有任何优化的“最”弱一致性，它的意思是说，我不考虑所有的中间状态的影响，只保证当没有新的更新之后，**经过一段时间之后，最终系统内所有副本的数据是正确的。它最大程度上保证了系统的并发能力，也因此，在高并发的场景下，它也是使用最广的一致性模型。**

## 分布式理论：CAP定理?

| 选项 | 描述 |
| --- | --- |
| C 一致性 | 分布式系统当中的一致性指的是所有节点的数据一致，或者说是所有副本的数据一致 |
| A 可用性 | Reads and writes always succeed. 也就是说系统一直可用，而且服务一直保持正常 |
| P 分区容错性 | 系统在遇到一些节点或者网络分区故障的时候，仍然能够提供满足一致性和可用性的服务 |

## 为什么CAP不能同时满足?

举例：当Node1更新，同步到Node2出故障时，Node2是继续等待（一致性）还是直接返回给用户（容错性）

大部分分布式系统会选择牺牲一致性保证高可用和分区容错。

## 什么是BASE 理论?

CAP的三个特性基本都满足，就是对CAP理论的协调

BASE：全称：Basically Available(基本可用)，Soft state（软状态）,和 Eventually consistent（最终一致性）三个短语的缩写，来自 ebay 的架构师提出。

**举例：**

- **可用性协调：**Basically Available(基本可用)，当分布式系统出现故障时，允许损失部分可用性，但绝不等价于系统不可用。

  **基本可用例子：**

  - 响应时间上的损失：正常情况下，一个在线搜索引擎需要在0.5秒之内返回给用户相应的查询结果，但由于出现故障（比如系统部分机房发生断电或断网故障），查询结果的响应时间增加到了1~2秒。
  - 功能上的损失：正常情况下，在一个电子商务网站（比如淘宝）上购物，消费者几乎能够顺利地完成每一笔订单。但在一些节日大促购物高峰的时候（比如双十一、双十二），由于消费者的购物行为激增，为了保护系统的稳定性（或者保证一致性），部分消费者可能会被引导到一个降级页面，如下：


- **一致性协调：**Soft state（软状态）：允许系统中的数据存在中间状态，并认为该状态不影响系统的整体可用性，即允许系统在多个不同节点的数据副本之间进行数据同步的过程中存在延迟。
- **一致性协调：**Eventually consistent（最终一致性）：最终一致性强调的是系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。因此最终一致性的本质是需要系统保证最终数据能够达到一致，而不需要实时保证系统数据的强一致性。

## 什么是分布式事务?

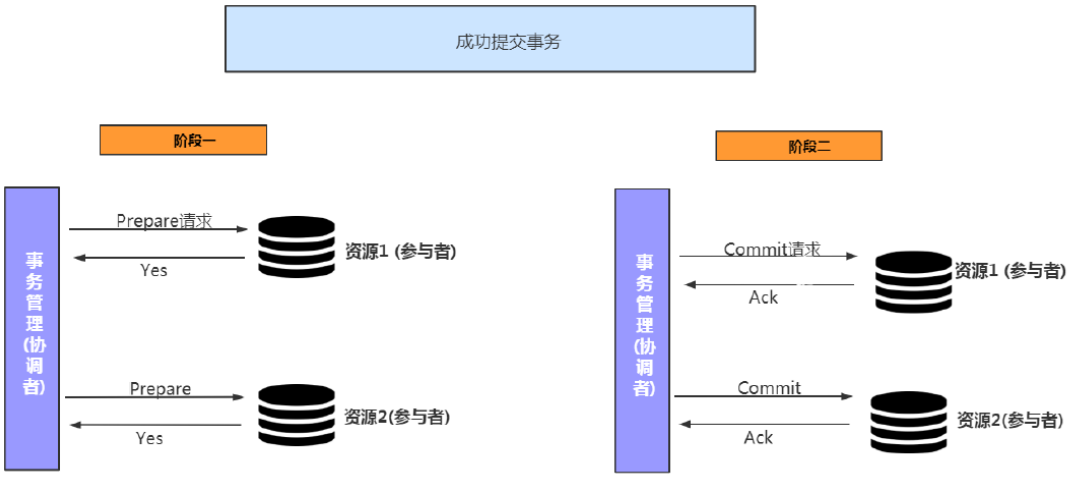
**事务的基本特性:**

- **Atomicity（原子性）**:只要事务中有一个操作出错，回滚到事务开始前的状态的话.
- **Consistency（一致性）**：是说事务执行前后，数据从一个状态到另一个状态必须是一致的，比如A向B转账（A、B的总金额就是一个一致性状态），不可能出现A扣了钱，B却没收到的情况发生。
- **Isolation（隔离性）**：多个并发事务之间相互隔离，不能互相干扰。对于并发事务操作同一份数据的隔离性问题，则是要求不能出现**脏读、幻读**的情况，**即事务A不能读取事务B还没有提交的数据，或者在事务A读取数据进行更新操作时**，不允许事务B率先更新掉这条数据。而为了解决这个问题，**常用的手段就是加锁了，对于数据库来说就是通过数据库的相关锁机制来保证。**
  - **脏读：**一个事务在处理过程中读取了另外一个事务未提交的数据
  - **幻读：**事务A读取与搜索条件相匹配的若干行。事务B以插入或删除行等方式来修改事务A的结果集，然后再提交。

- **不可重复读：**多次查询某个数据，却得到不同的结果。与脏读的区别：脏读是读到未提交的数据，而不可重复读读到的却是已经提交的数据，但实际上是违反了事务的一致性原则
- **Durablity（持久性）**：事务完成后，对数据库的更改是永久保存的。

## 一致性协议 2PC介绍?

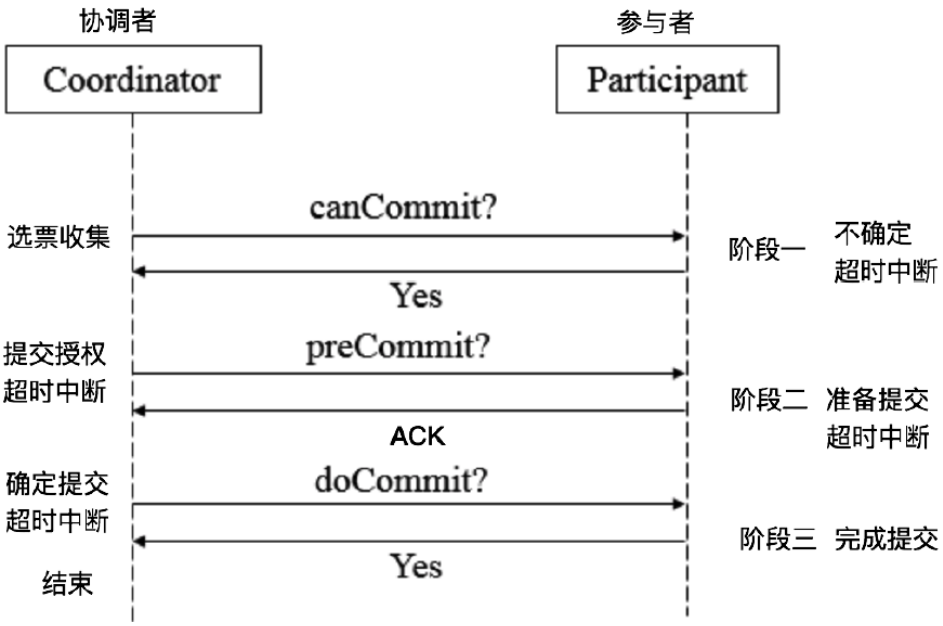
如图，分两个阶段，一阶段询问是否可以执行事务，二阶段提交事务。




缺点：

- 同步阻塞：二阶段所有参与者都在等待其他参与者响应的过程中，无法进行其他操作。
- 单点问题：协调者自身出问题，导致其他参与者一直锁定状态。
- 数据不一致：假如协调者发送commit 请求，有参与者出现故障，这将导致严重的数据不一致问题。
- 过于保守：协调者判断是否需要中断事务只能通过是否超时判断，**任意一个节点失败都会导致整个事务的失败**，过于保守。（参与者故障时，由于自身没有超时机制，协调者只能通过自身的超时机制判断是否中断事务）

## 一致性协议 3PC介绍?




分为三阶段：

1. CanCommit：发送各参与者canCommit请求，返回Yes进入预备状态，返回No则给阶段二发送abort请求
2. PreCommit：协调者在得到所有参与者的响应之后，会根据结果有2种执行操作的情况：执行事务预提交，或者中断事务假如所有参与反馈的都是Yes，那么就会执行事务预提交。**若任一参与者反馈了No响应，或者在等待超时后，协调者尚无法接收到所有参与者反馈，则中断事务。**
3. 该阶段做真正的事务提交或者完成事务回滚。

   **注意：一旦进入阶段三，可能会出现 2 种故障：**

   - 协调者出现问题
   - 协调者和参与者之间的网络故障

     如果出现了任一一种情况，最终都会导致参与者无法收到 doCommit 请求或者 abort 请求，针对这种情况，参与者都会在等待超时之后，继续进行事务提交，导致数据不一致。

**2PC对比3PC：**

1. 协调者和参与者都设置超时机制（2pc只有协调者设置了），有效改善协调者故障，参与者一直锁问题。
2. 多设置了一个缓冲阶段，成功性更高。
3. 由于通信问题协调者的abort请求没发送过去，但是参与者已经运行commit请求，导致数据不一致。（由于参与者自己有超时机制所以这种情况才会发生）

## 什么是一致性算法 Paxos?

分为准备阶段和决议阶段：

**准备阶段：**

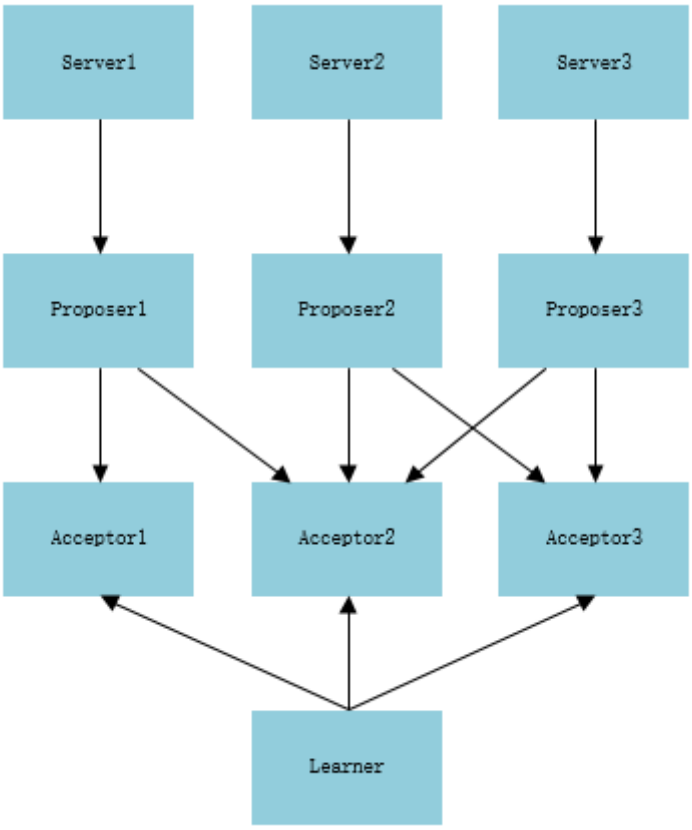
1. Proposer向Acceptor发送消息，Acceptor收到编号后不再回复比这个编号小的Proposer，若Proposer没有收到过半的Acceptor回复，则重新取得大的编号重新发。（这个阶段Acceptor回复的是[编号,null]）。

**决议阶段：**

1. Proposer收到过半回复后提交[编号,serverName]的议案。
2. 这时过半的Acceptor接收了Proposer的议案，则该Proposer选举成为Leader。

## Paxos具体选举流程?

这里具体例子来说明Paxos的整个具体流程：假如有Server1、Server2、Server3这样三台服务器，我们要从中选出leader，这时候Paxos派上用场了。
整个选举的结构图如下：




**Phase1（准备阶段）**

1. 每个Server都向Proposer发消息称自己要成为leader，Server1往Proposer1发、Server2往Proposer2发、Server3往Proposer3发；
2. 现在每个Proposer都接收到了Server1发来的消息但时间不一样，Proposer2先接收到了，然后是Proposer1，接着才是Proposer3；
3. Proposer2首先接收到消息所以他从系统中取得一个编号1，Proposer2向Acceptor2和Acceptor3发送一条，编号为1的消息；
   接着Proposer1也接收到了Server1发来的消息，取得一个编号2，Proposer1向Acceptor1和Acceptor2发送一条，编号为2的消息；
   最后Proposer3也接收到了Server3发来的消息，取得一个编号3，Proposer3向Acceptor2和Acceptor3发送一条，编号为3的消息；
4. 这时Proposer1发送的消息先到达Acceptor1和Acceptor2，这两个都没有接收过请求所以接受了请求返回[2,null]给Proposer1，并承诺不接受编号小于2的请求；
5. 此时Proposer2发送的消息到达Acceptor2和Acceptor3，Acceprot3没有接收过请求返回[1,null]给Proposer2，并承诺不接受编号小于1的请求，但这时Acceptor2已经接受过Proposer1的请求
   并承诺不接受编号小于的2的请求了，所以Acceptor2拒绝Proposer2的请求；
6. 最后Proposer3发送的消息到达Acceptor2和Acceptor3，Acceptor2接受过提议，但此时编号为3大于Acceptor2的承诺2与Accetpor3的承诺1，所以接受提议返回[3,null]；
7. Proposer2没收到过半的回复所以重新取得编号4，并发送给Acceptor2和Acceptor3，然后Acceptor2和Acceptor3都收到消息，此时编号4大于Acceptor2与Accetpor3的承诺3，所以接受提议返回[4,null]；

**Phase2（决议阶段）**

1. Proposer3收到过半（三个Server中两个）的返回，并且返回的Value为null，所以Proposer3提交了[3,server3]的议案；
2. Proposer1收到过半返回，返回的Value为null，所以Proposer1提交了[2,server1]的议案；
3. Proposer2收到过半返回，返回的Value为null，所以Proposer2提交了[4,server2]的议案；

4. Acceptor1、Acceptor2接收到Proposer1的提案[2,server1]请求，Acceptor2承诺编号大于4所以拒绝了通过，Acceptor1通过了请求；
5. Proposer2的提案[4,server2]发送到了Acceptor2、Acceptor3，提案编号为4所以Acceptor2、Acceptor3都通过了提案请求；
6. Acceptor2、Acceptor3接收到Proposer3的提案[3,server3]请求，Acceptor2、Acceptor3承诺编号大于4所以拒绝了提案；
7. 此时过半的Acceptor都接受了Proposer2的提案[4,server2],Larner感知到了提案的通过，Larner学习提案，server2成为Leader；

**一个Paxos过程只会产生一个议案所以至此这个流程结束，选举结果server2为Leader；**

## 一致性算法 Raft?

Raft算法分为两个阶段，首先是选举过程，然后在选举出来的领导人带领进行正常操作，比如日志复制等。

**领导人Leader选举：**

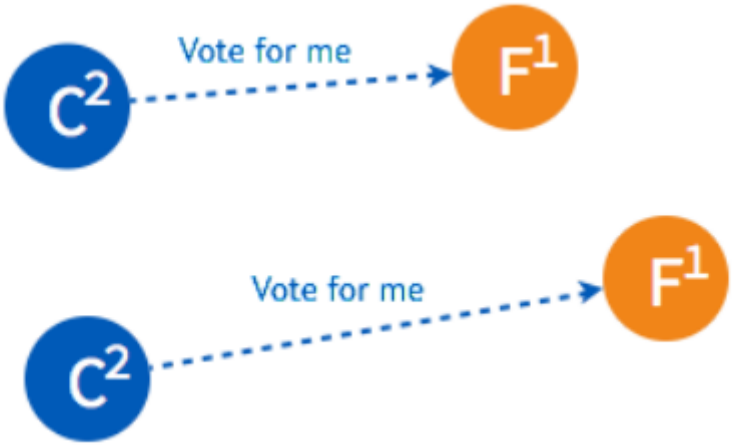
- 领导者(leader)：处理客户端交互，日志复制等动作，一般一次只有一个领导者
- 候选者(candidate)：候选者就是在选举过程中提名自己的实体，一旦选举成功，则成为领导者
- 跟随者(follower)：类似选民，完全被动的角色，这样的服务器等待被通知投票

  Raft使用心跳机制来触发选举。当server启动时，初始状态都是follower。每一个server都有一个定时器，如果follower超时，说明leader不可用了，它就开始一次选举，变为candidate，candidate向其他follower发送选票，绝大多数（**超过半数**）选他就成为leader。选举成功会给follower发送心跳包来建立心跳机制。

**节点异常四种类型：**

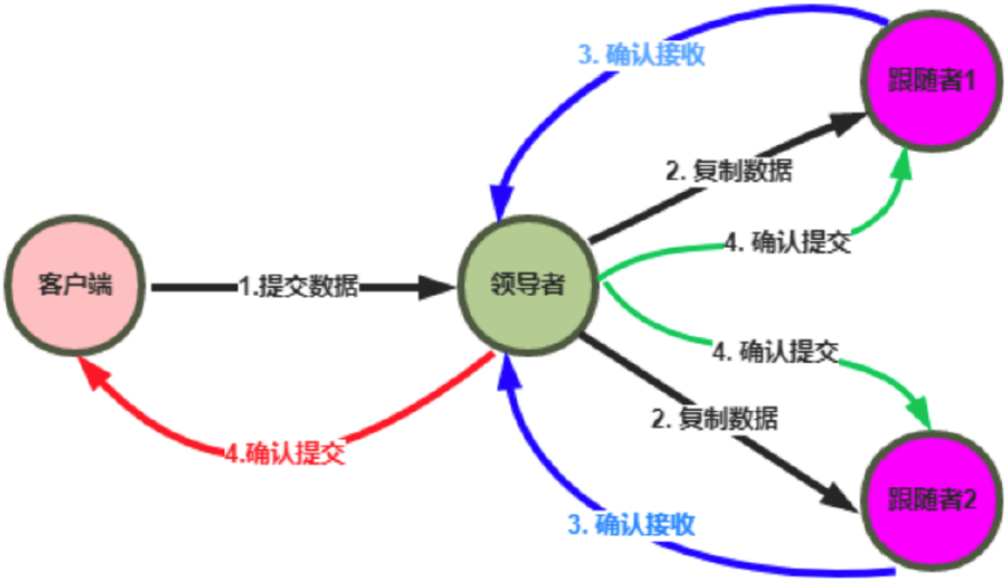
- leader 不可用：重新选举出leader，之前的leader恢复后变为follower，与现有 leader 中的日志保持一致
- follower 不可用：相对简单，恢复时重新同步leader即可。

**多个 candidate选举问题：**




比如上图，两个candidate一直选不出来谁是leader，于是将随机选择一个等待间隔（150ms ~ 300ms）再次发起投票。

- ## 什么是一致性算法 Raft的日志复制（保证数据一致性）？




如上图表示了当一个客户端发送一个请求给领导者，随后领导者复制给跟随者的整个过程。

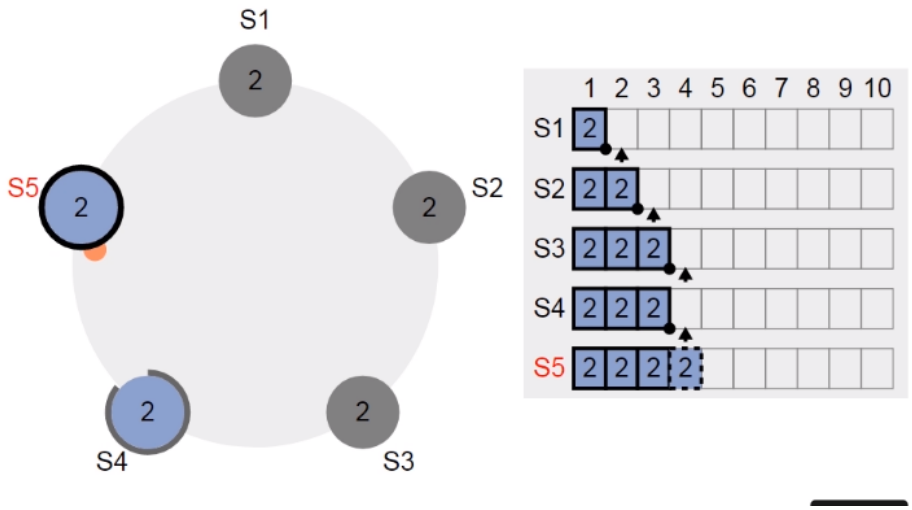
**日志复制四个步骤：**

1. 客户端的每一个请求都包含被复制状态机执行的指令。
2. leader将这个指令作为新的日志条目添加到日志中，然后并行发起 RPC 给其他的follow，让他们复制这条信息。
3. 跟随者响应ACK,如果 follower 宕机或者运行缓慢或者丢包，leader会不断的重试，直到所有的 follower 最终都复制了所有的日志条目。
4. 通知所有的Follower提交日志，同时leader提交这条日志到自己的状态机中，并返回给客户端。

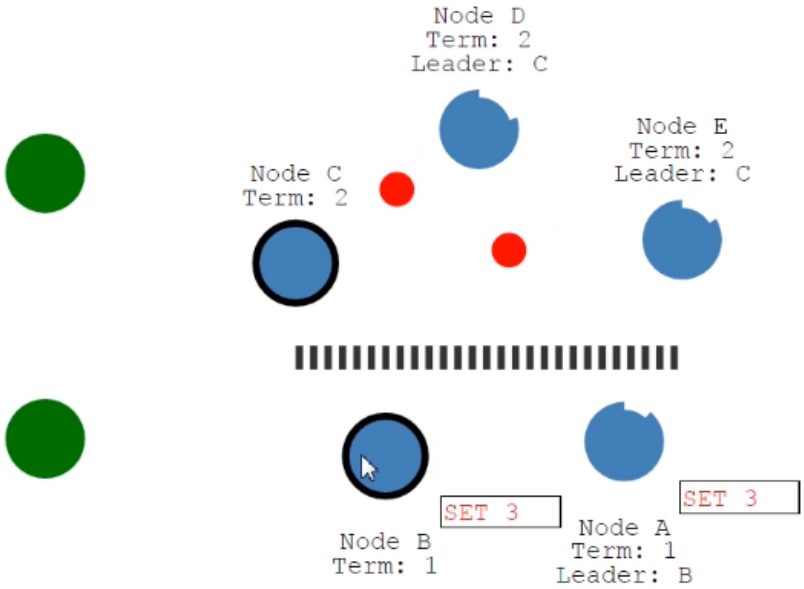
- ## 一致性算法 Raft出现分区、只剩一个follow和一个leader怎么办?

节点和leader是少数就会停止发送请求，如图只剩S5（leader）和s4（follow）这时五个节点只有两个运行，则整个请求都停止发送

screencast soon to explain what's going on. This visualization (RaftScope) is still pretty rough around the edges; pull requests would be very welcom




**分区：**如下图，由于网络问题出现分区，后面恢复则服从多数那边的leader




## 分布式系统设计策略是什么?

分布式系统本质是通过低廉的硬件攒在一起以获得更好的吞吐量、性能以及可用性等。

在分布式环境下，有几个问题是普遍关心的，我们称之为设计策略：

- 如何检测当前节点还活着？（心跳检测）
- 如何保障高可用?
- 容错处理
- 负载均衡

## 什么是心跳检测机制?

1. 为什么要进行心跳检测?

   目的是检测节点的能否正常访问，保证系统可靠性和高可用性。

2. 如何进行心跳检测?

   服务端向各个节点发送心跳检测包，节点收到包后给服务端返回信息。

3. 如何应对异常机制?

   周期检测心跳机制、累计失效检测机制

   - 周期检测心跳机制：

     Server端每间隔 t 秒向Node集群发起监测请求，设定超时时间，如果超过超时时间，则判断“死亡”。

   - 累计失效检测机制：

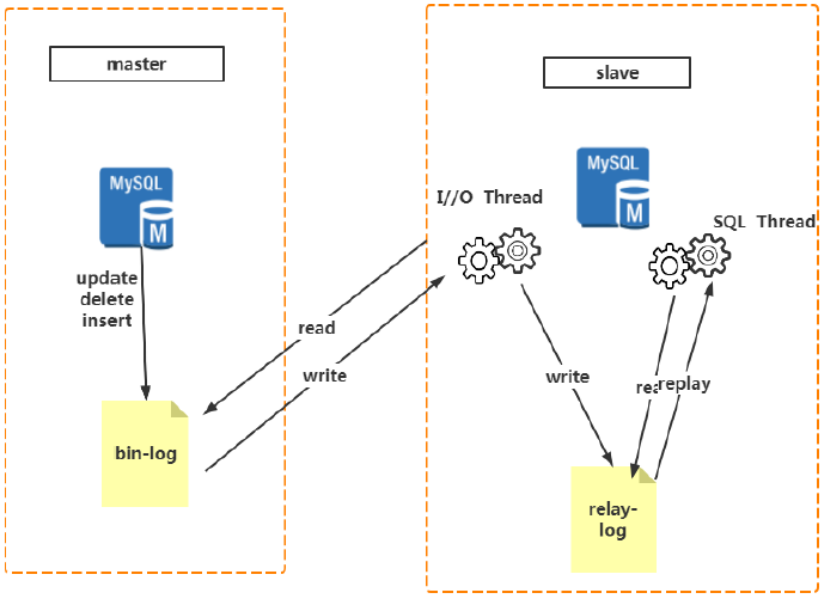
     在周期检测心跳机制的基础上，**统计一定周期内节点的返回情况**（包括超时及正确返回），以此计算节点的“死亡”概率。另外，对于宣告“濒临死亡”的节点可以发起有限次数的重试，以作进一步判断。

## 高可用设计几种方式?

1. **主备模式:**

   主从复制，主机宕机时，备机接管主机的一切工作。主机恢复时，使用者的设定以自动（热备）或手动（冷备）方式将服务切换到主机上运行。

   举例：**MySQL、Redis等就采用MS模式实现主从复制。保证高可用**，如图所示。




上图讲解：一台MySQL数据库一旦启用二进制日志后，它的数据库中所有操作（增删改）都会以“事件”的方式记录在二进制日志中，**其他数据库作为slave通过一个I/O线程与主服务器保持通信。**并监控master的二进制日志文件的变化，如果发现master二进制日志文件发生变化，则会把变化复制到自己的中继日志中，然后slave的一个SQL线程会把相关的“事件”执行到自己的数据库中，以此实现从数据库和主数据库的一致性，也就实现了主从复制。

2. **互备模式（用的比较少）：**

   指两台主机同时运行各自的服务工作且**相互监测情况。**

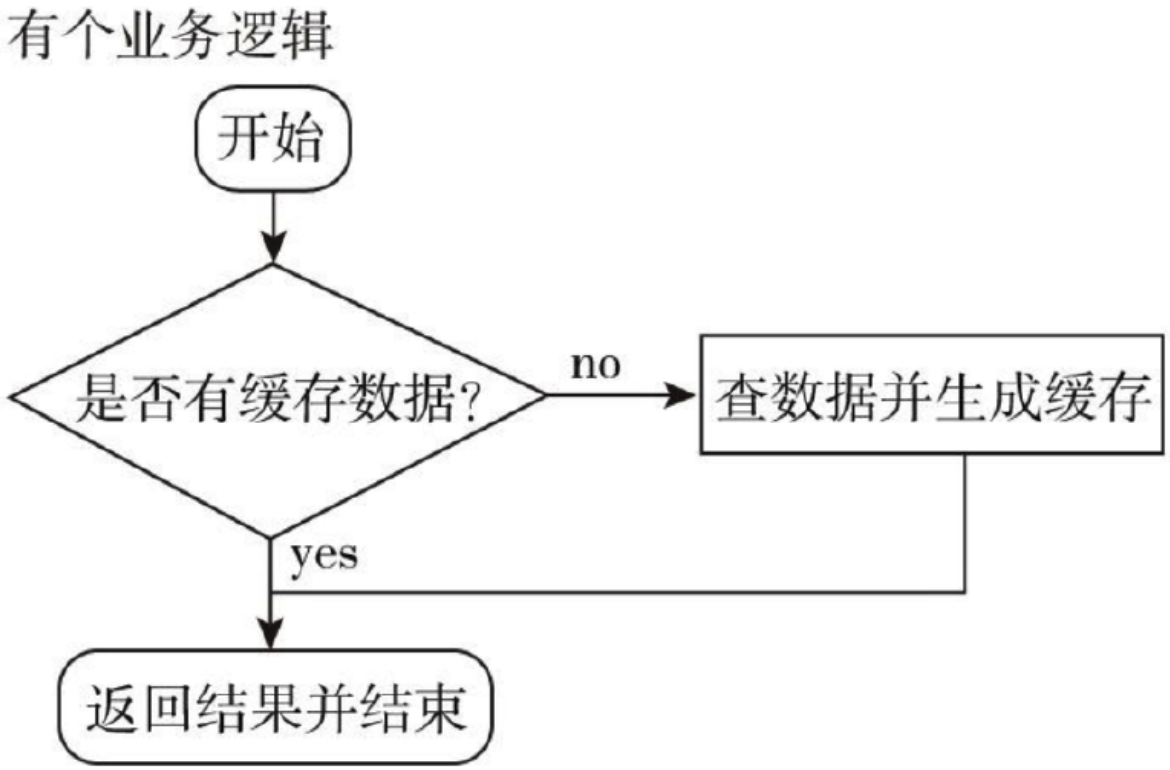
3. **集群模式：**

   集群模式是指有多个节点在运行，同时可以通过主控节点分担服务请求。如Zookeeper。集群模式需要解决主控节点（如Zookeeper）本身的高可用问题，一般采用主备模式。

## 什么是容错性?

容错顾名思义就是IT系统对于错误包容的能力
容错的处理是保障分布式环境下相应系统的高可用或者健壮性，一个典型的案例就是对于缓存穿透问题的解决方案。

我们来具体看一下这个例子，如图所示:




**问题描述：**
我们在项目中使用缓存通常都是先检查缓存中是否存在，有则自己查询缓存，没有则查询数据库后生成缓存。如果我们查询的某一个数据在缓存中一直不存在，就会造成每一次请求都查询DB，这样缓存就失去了意义，在流量大时，或者有人恶意攻击。如频繁发起为id为“-1”的条件进行查询，可能DB就挂掉了。

**那这种问题有什么好办法解决呢?**

可以将这个不存在的key（如上问题id=-1）预先设定一个值。比如，key=“null”，在返回这个null值的时候，我们的应用就可以认为这是不存在的key，那我们的应用就可以决定是否继续等待访问，还是放弃掉这次操作。如果过了一段时间，再次请求这个key，取到的不是null，可以认为这时候key有值了，从而避免了透传到数据库，把大量的类似请求挡在了缓存之中。

## 什么是负载均衡?

其关键在于使用多台集群服务器共同分担计算任务，把网络请求及计算分配到集群可用的不同服务器节点上，从而达到高可用性及较好的用户操作体验。

以Nginx为例，负载均衡有以下几种策略：

- 轮询：即Round Robin，根据Nginx配置文件中的顺序，依次把客户端的Web请求分发到不同的后端服务器。
- 最少连接：当前谁连接最少（处理请求比较少），分发给谁。
- IP地址哈希：确定相同IP请求可以转发给同一个后端节点处理，以方便session保持。
- 基于权重的负载均衡：配置Nginx把请求更多地分发到高配置的后端服务器上，把相对较少的请求分发到低配服务器。

## 网络通信由什么组成的?

网络通信需要做的就是将流从一台计算机传输到另外一台计算机，**基于传输协议和网络IO来实现**，其中传输协议比较出名的有tcp、udp等等，tcp、udp都是在基于Socket概念上为某类应用场景而扩展出的传输协议，网络IO，主要有bio、nio、aio三种方式。

**网络数据传输的要素**：

**协议：**

- UDP：广播协议，面向无连接，速度快，不安全。（无连接：只会发送，不保证对方能不能接收到）
- TCP：面向连接的协议，速度不快，较为安全。（连接：建立三次握手，确认连通后再发送）

**IO：**

- BIO：同步堵塞IO
- NIO：非堵塞IO
- AIO：非堵塞IO

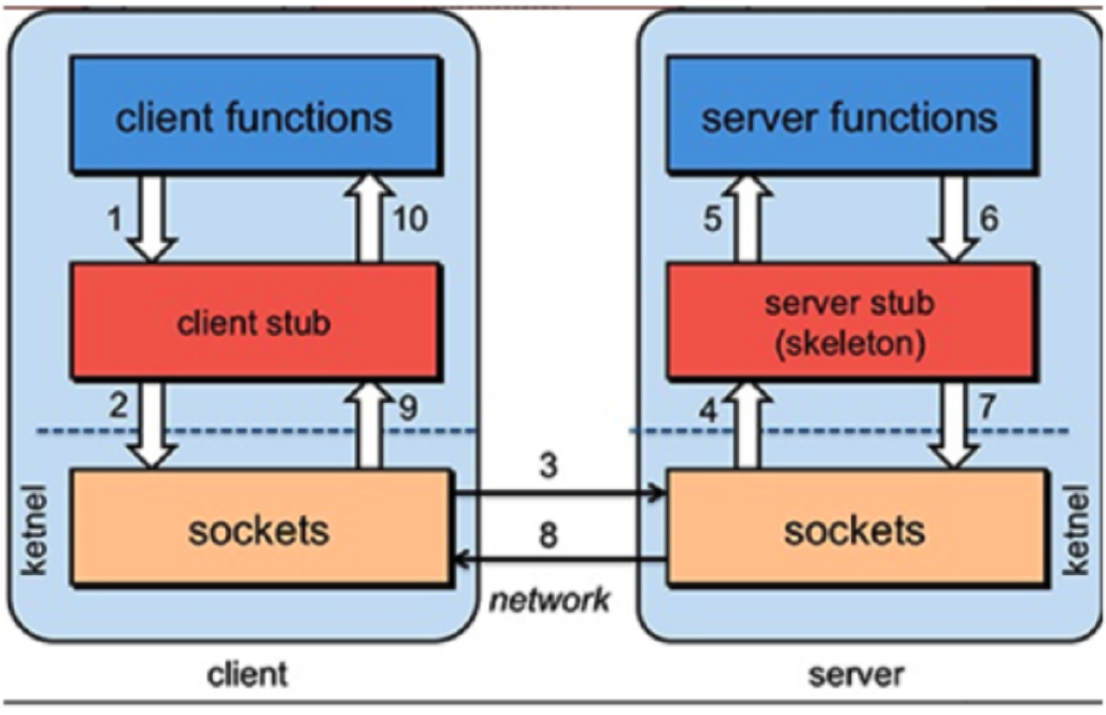
## 什么是RPC?

RPC全称为remote procedure call，即远程过程调用。

需要注意的是RPC并不是一个具体的技术，调用远程服务就像调用本地服务一样，而是指整个网络远程调用过程。

RPC架构：一个完整的RPC架构里面包含了四个核心的组件，分别是Client，Client Stub，Server以及Server Stub，这个Stub可以理解为存根。如下图：

- Client：服务调用方
- Server：服务提供者。
- Client Stub：存放服务端地址消息，将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方
- Server Stub：接收客户端发送过来的消息，将消息解包，并调用本地的方法。




1）服务消费方（client）调用以本地调用方式调用服务；

2）client stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；

3）client stub找到服务地址，并将消息发送到服务端；

4）server stub收到消息后进行解码；

5）server stub根据解码结果调用本地的服务；

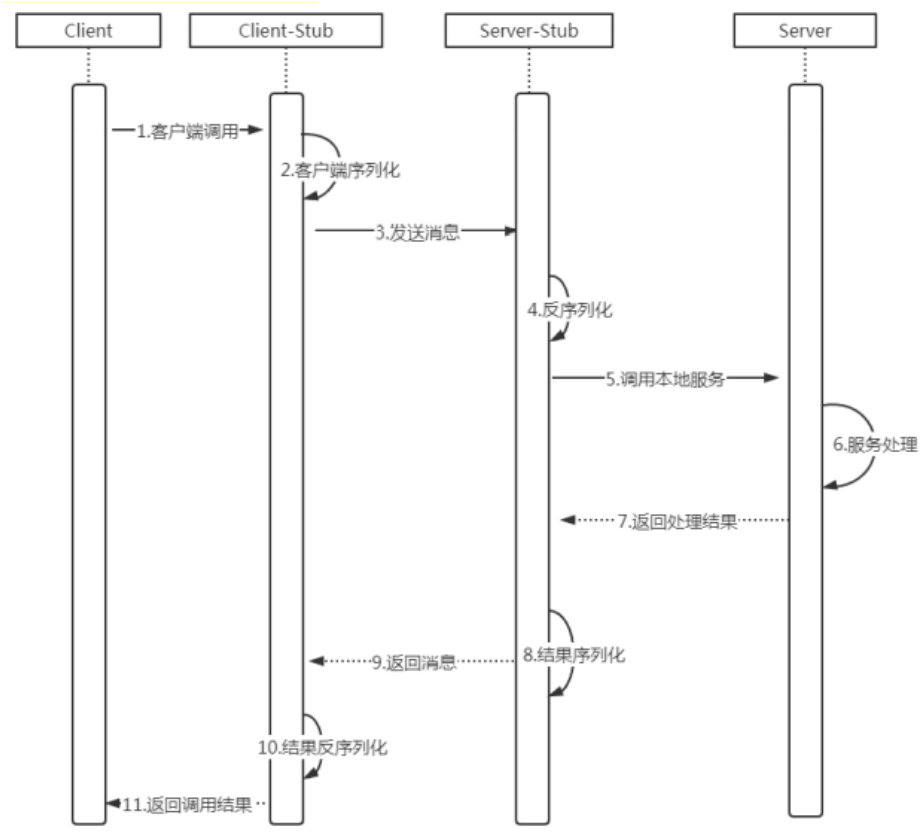
6）本地服务执行并将结果返回给server stub；

7）server stub将返回结果打包成消息并发送至消费方；

8）client stub接收到消息，并进行解码；

9）服务消费方得到最终结果。

RPC的目标就是要2~8这些步骤都封装起来，让用户对这些细节透明。



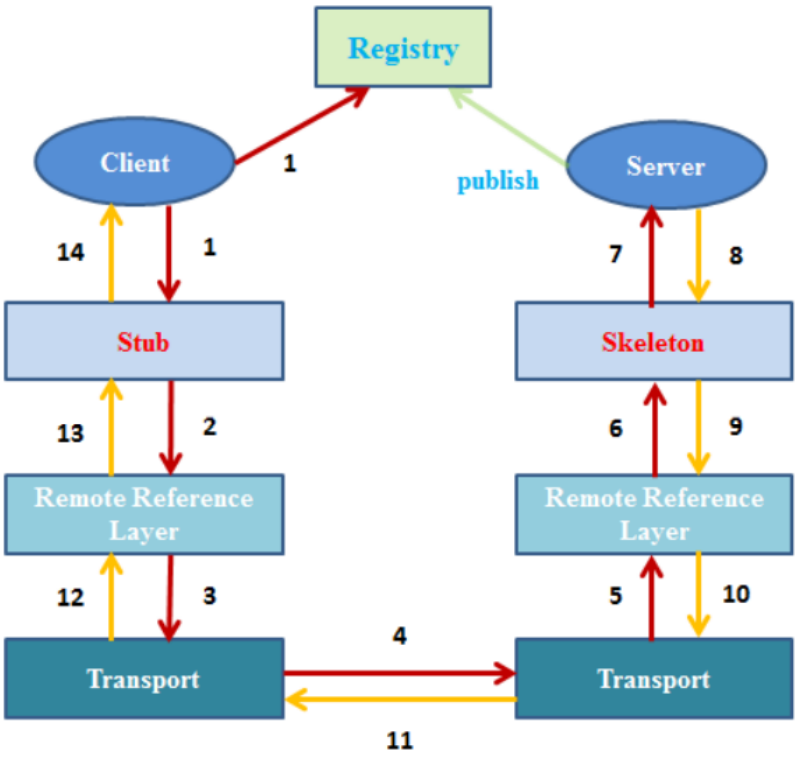

**总结：**

1. 其实rpc底层的流程就是序列化后进行数据传输，接收后反序列化，再返回时序列化数据传输，将结果反序列化。（之所以要序列化是为了将对象转变为二进制流方便传输，传输方式为socket传输）
2. Client Stub：存放服务端地址消息，将客户端的请求参数打包成网络消息，然后通过网络远程发送给服务方
3. Server Stub：接收客户端发送过来的消息，将消息解包，并调用本地的方法。

## RMI介绍：

Java RMI 指的是远程方法调用 (Remote Method Invocation),是java原生支持的远程调用,RMI主要用 于不同虚拟机之间的通信，这些虚拟机可以在不同的主机上、也可以在同一个主机上，这里的通信可以理解为一个 虚拟机上的对象调用另一个虚拟机上对象的方法。

**调用过程：**




1.客户端：

1）存根/桩(Stub)：远程对象在客户端上的代理；

2）远程引用层(Remote Reference Layer)：解析并执行远程引用协议；

3）传输层(Transport)：发送调用、传递远程方法参数、接收远程方法执行结果。

2.服务端：

1）骨架(Skeleton)：读取客户端传递的方法参数，调用服务器方的实际对象方法， 并接收方法执行后的返回值；

2）远程引用层(Remote Reference Layer)：处理远程引用后向骨架发送远程方法调用；

3）传输层(Transport)：监听客户端的入站连接，接收并转发调用到远程引用层。

3.注册表(Registry)：

1）以URL形式注册远程对象，并向客户端回复对远程对象的引用。

## 同步和异步、阻塞和非阻塞概念：

### 同步和异步：

**同步：**指用户进程触发IO操作等待或者轮训的方式查看IO操作是否就绪（就是通过等待或者轮询方式进行确认是否可以执行）

**异步：**当一个异步进程调用发出之后，调用者不会立刻得到结果。而是在调用发出之后，被调用者通过状态、通知来通知调用者，或者通过回调函数来处理这个调用。（就是有通知或回调函数通知你可以执行）

### 阻塞和非阻塞：

**阻塞：** ATM机排队取款，你只能等待排队取款

**非阻塞：**柜台取款，取个号，然后坐在椅子上做其他事，等广播通知，没到你的号你就不能去，但你可以不断的问大堂经理排到了没有。

**例子：**

```
老张煮开水。 老张，水壶两把（普通水壶，简称水壶；会响的水壶，简称响水壶）。
老张把水壶放到火上，站立着等水开。（同步阻塞）
老张把水壶放到火上，去客厅看电视，时不时去厨房看看水开没有。（同步非阻塞）
老张把响水壶放到火上，立等水开。（异步阻塞）
老张把响水壶放到火上，去客厅看电视，水壶响之前不再去看它了，响了再去拿壶。（异步非阻塞）
```

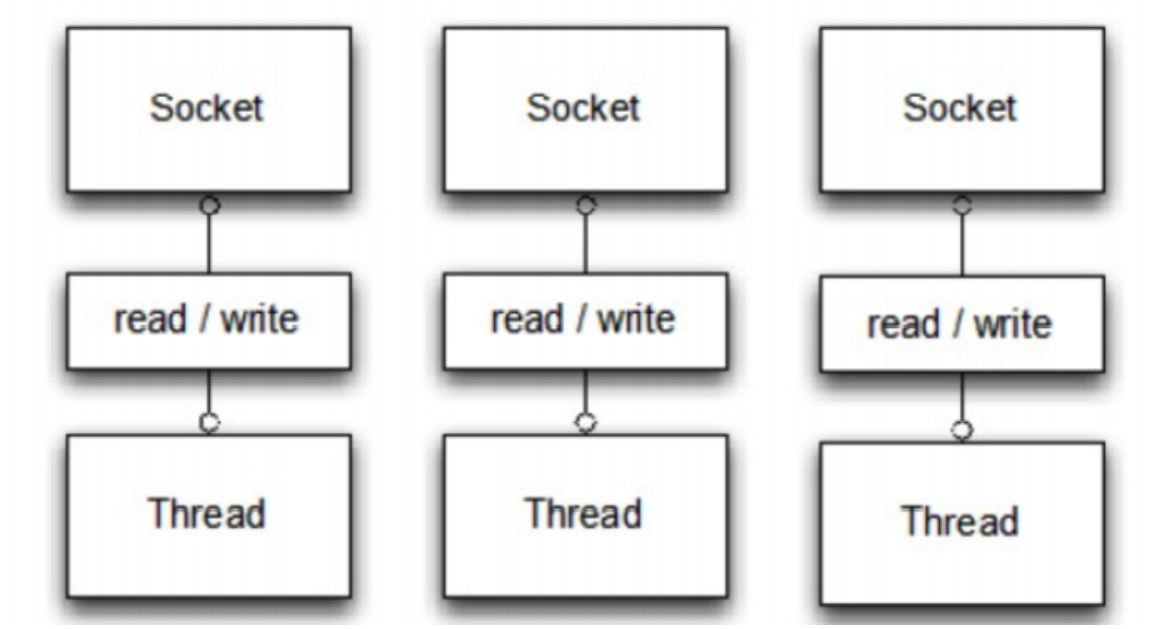
同步和异步强调的是等待的方式，是轮训方式获取（同步）还是对方通知你（异步）。

阻塞和非阻塞强调的是等待的过程，是一直排队等待（阻塞）还是等待时候去干其他的事（非阻塞）。

## BIO介绍：

同步阻塞IO，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。 适用场景：Java1.4之前唯一的选择，简单易用但资源开销太高

### 阻塞IO的通信方式：




### 代码实现：

服务端：

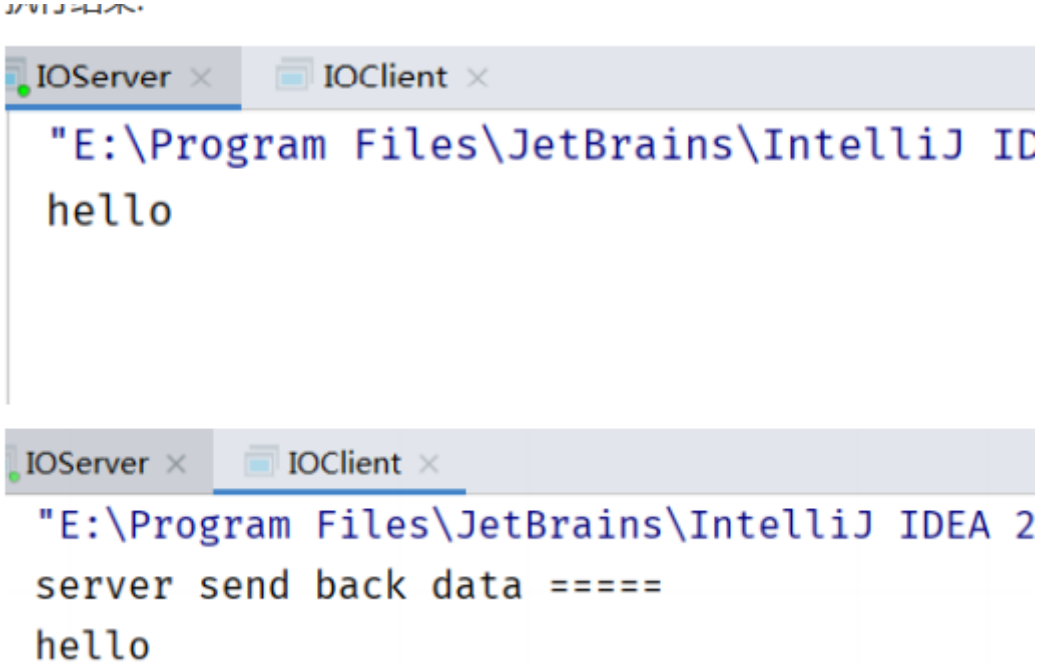
```
public class IOServer {
    public static void main(String[] args) throws Exception {
        //首先创建了一个serverSocket
        ServerSocket serverSocket = new ServerSocket();
        serverSocket.bind(new InetSocketAddress("127.0.0.1",8081));
        while (true){
            Socket socket = serverSocket.accept(); //同步阻塞
```

```
            new Thread(()->{
                try {
                        byte[] bytes = new byte[1024];
                        int len = socket.getInputStream().read(bytes); //同步阻塞
                        System.out.println(new String(bytes,0,len));
                        socket.getOutputStream().write(bytes,0,len);
                        socket.getOutputStream().flush();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
            }).start();
    }
}
```

客户端:

```
public class IOClient {
    public static void main(String[] args) throws IOException {
        Socket socket = new Socket("127.0.0.1",8081);
        socket.getOutputStream().write("hello".getBytes());
        socket.getOutputStream().flush();
        System.out.println("server send back data =====");
        byte[] bytes = new byte[1024];
        int len = socket.getInputStream().read(bytes);
        System.out.println(new String(bytes,0,len));
        socket.close();
    }
}
```

执行结果:




## NIO介绍:

同步非阻塞IO，是指JDK 1.4 及以上版本。

### 通道（Channels）

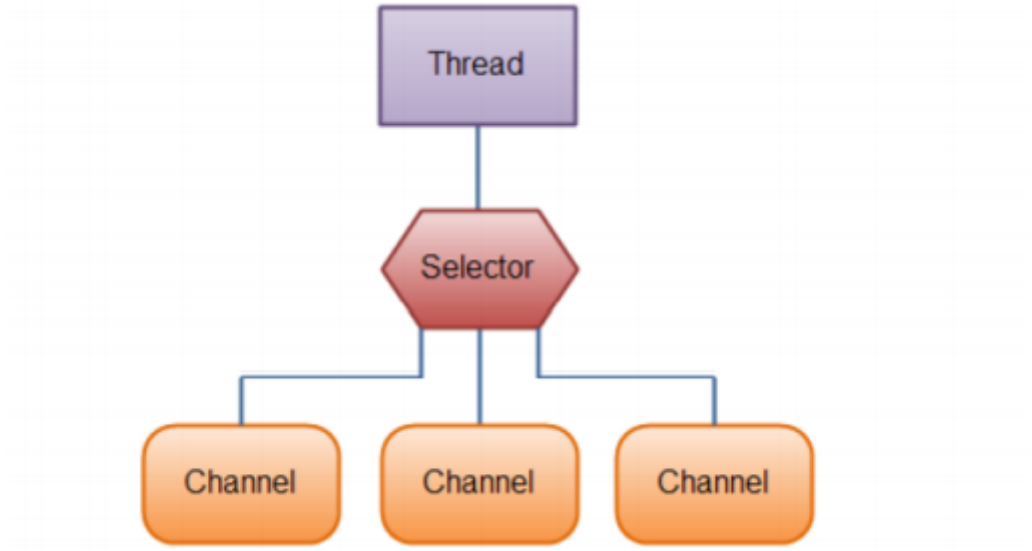
NIO 新引入的最重要的抽象是通道的概念。Channel 数据连接的通道。 数据可以从Channel读到Buffer中，也可以从Buffer 写到Channel中 。

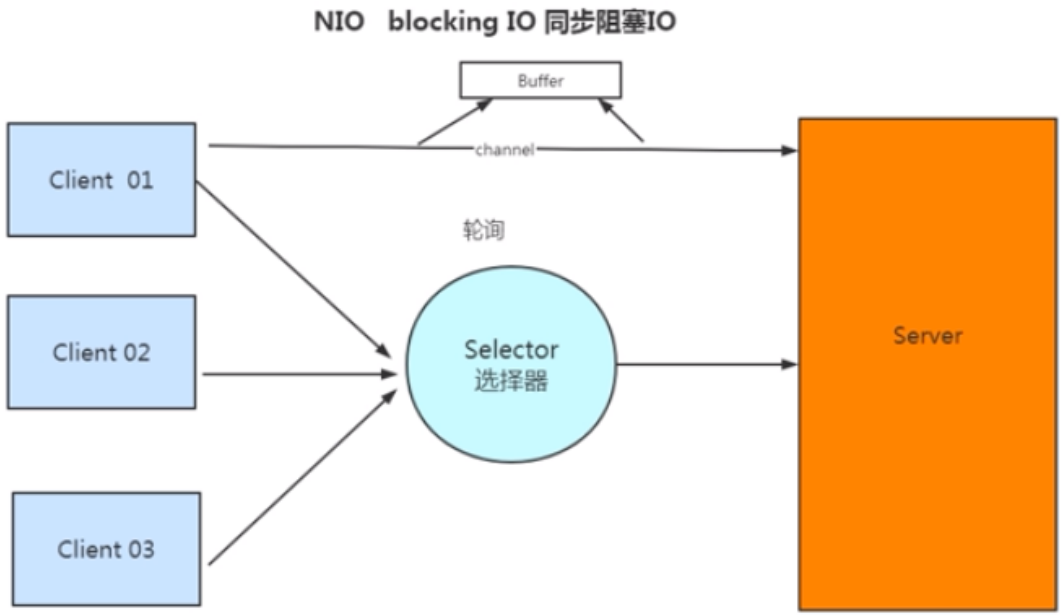
### 缓冲区（Buffers）

通道channel可以向缓冲区Buffer中写数据，也可以向buffer中存数据。

### 选择器（Selector）

使用选择器，借助单一线程，就可对数量庞大的活动 I/O 通道实时监控和维护。




## NIO和BIO对比：

BIO：一个连接来了，会创建一个线程

NIO：由一个线程控制，channel注册到selector，定期检查selector，进而读取数据。

举例：

```
BIO:
每个小朋友配一个老师。每个老师隔段时间询问小朋友是否要上厕所，如果要上，就领他去厕所，100个小朋友就需要100个老师来询问，并且每个小朋友上厕所的时候都需要一个老师领着他去上，这就是IO模型，一个连接对应一个线程。

NIO:
所有的小朋友都配同一个老师。这个老师隔段时间询问所有的小朋友是否有人要上厕所，然后每一时刻把所有要上厕所的小朋友批量领到厕所，这就是NIO模型，所有小朋友都注册到同一个老师，对应的就是所有的连接都注册到一个线程，然后批量轮询。
```

## 代码实现：（具体还需了解）

服务端：

```
public class NIOServer extends Thread{

    //1.声明多路复用器
    private Selector selector;
    //2.定义读写缓冲区
    private ByteBuffer readBuffer = ByteBuffer.allocate(1024);
    private ByteBuffer writeBuffer = ByteBuffer.allocate(1024);

    //3.定义构造方法初始化端口
    public NIOServer(int port) {
        init(port);
    }

    //4.main方法启动线程
    public static void main(String[] args) {
        new Thread(new  NIOServer(8888)).start();
    }

    //5.初始化
    private void init(int port) {

        try {
            System.out.println("服务器正在启动.....");
            //1)开启多路复用器
            this.selector = Selector.open();
            //2) 开启服务通道
            ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
            //3)设置为非阻塞
            serverSocketChannel.configureBlocking(false);
            //4)绑定端口
            serverSocketChannel.bind(new InetSocketAddress(port));
            /**
             * SelectionKey.OP_ACCEPT   —— 接收连接继续事件，表示服务器监听到了客户连接，服务器可以接收这个连接了
             * SelectionKey.OP_CONNECT  —— 连接就绪事件，表示客户与服务器的连接已经建立成功
             * SelectionKey.OP_READ     —— 读就绪事件，表示通道中已经有了可读的数据，可以执行读操作了（通道目前有数据，可以进行读操作了）
             * SelectionKey.OP_WRITE    —— 写就绪事件，表示已经可以向通道写数据了（通道目前可以用于写操作）
             */
            //5)注册,标记服务连接状态为ACCEPT状态
            serverSocketChannel.register(this.selector, SelectionKey.OP_ACCEPT);
            System.out.println("服务器启动完毕");
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public void run(){
        while (true){
            try {
                //1.当有至少一个通道被选中,执行此方法
                this.selector.select();
                //2.获取选中的通道编号集合
                Iterator<SelectionKey> keys =
this.selector.selectedKeys().iterator();
                //3.遍历keys
                while (keys.hasNext()) {
                    SelectionKey key = keys.next();
                    //4.当前key需要从动刀集合中移出,如果不移出,下次循环会执行对应的逻
辑,造成业务错乱
                    keys.remove();
                    //5.判断通道是否有效
                    if (key.isValid()) {
                        try {
                            //6.判断是否可以连接
                            if (key.isAcceptable()) {
                                accept(key);
                            }
                        } catch (CancelledKeyException e) {
                            //出现异常断开连接
                            key.cancel();
                        }

                        try {
                            //7.判断是否可读
                            if (key.isReadable()) {
                                read(key);
                            }
                        } catch (CancelledKeyException e) {
                            //出现异常断开连接
                            key.cancel();
                        }

                        try {
                            //8.判断是否可写
                            if (key.isWritable()) {
                                write(key);
                            }
                        } catch (CancelledKeyException e) {
                            //出现异常断开连接
                            key.cancel();
                        }
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    private void accept(SelectionKey key) {
        try {
            //1.当前通道在init方法中注册到了selector中的ServerSocketChannel
```

```
            ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key.channel();
            //2.阻塞方法, 客户端发起后请求返回.
            SocketChannel channel = serverSocketChannel.accept();
            ///3.serverSocketChannel设置为非阻塞
            channel.configureBlocking(false);
            //4.设置对应客户端的通道标记,设置次通道为可读时使用
            channel.register(this.selector, SelectionKey.OP_READ);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    //使用通道读取数据
    private void read(SelectionKey key) {
        try{
            //清空缓存
            this.readBuffer.clear();
            //获取当前通道对象
            SocketChannel channel = (SocketChannel) key.channel();
            //将通道的数据(客户发送的data)读到缓存中.
            int readLen = channel.read(readBuffer);
            //如果通道中没有数据
            if(readLen == -1 ){
                //关闭通道
                key.channel().close();
                //关闭连接
                key.cancel();
                return;
            }
            //Buffer中有游标,游标不会重置,需要我们调用flip重置. 否则读取不一致
            this.readBuffer.flip();
            //创建有效字节长度数组
            byte[] bytes = new byte[readBuffer.remaining()];
            //读取buffer中数据保存在字节数组
            readBuffer.get(bytes);
            System.out.println("收到了从客户端 "+ channel.getRemoteAddress() + " : "+ new String(bytes,"UTF-8"));
            //注册通道,标记为写操作
            channel.register(this.selector,SelectionKey.OP_WRITE);

        }catch (Exception e){

        }
    }

    //给通道中写操作
    private void write(SelectionKey key) {
        //清空缓存
        this.readBuffer.clear();
        //获取当前通道对象
        SocketChannel channel = (SocketChannel) key.channel();
        //录入数据
        Scanner scanner = new Scanner(System.in);

        try {
            System.out.println("即将发送数据到客户端..");
            String line = scanner.nextLine();
            //把录入的数据写到Buffer中
            writeBuffer.put(line.getBytes("UTF-8"));
            //重置缓存游标
```

```
            writeBuffer.flip();
            channel.write(writeBuffer);
            channel.register(this.selector,SelectionKey.OP_READ);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

客户端：

```
public class NIOClient {
    public static void main(String[] args) {
        //创建远程地址
        InetSocketAddress address  = new InetSocketAddress("127.0.0.1",8888);
        SocketChannel channel = null;
        //定义缓存
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        try {
            //开启通道
              channel = SocketChannel.open();
            //连接远程远程服务器
            channel.connect(address);
            Scanner sc = new Scanner(System.in);
            while (true){
                System.out.println("客户端即将给 服务器发送数据..");
                String line = sc.nextLine();
                if(line.equals("exit")){
                    break;
                }
                //控制台输入数据写到缓存
                buffer.put(line.getBytes("UTF-8"));
                //重置buffer 游标
                buffer.flip();
                //数据发送到数据
                channel.write(buffer);
                //清空缓存数据
                buffer.clear();

                //读取服务器返回的数据
                int readLen = channel.read(buffer);
                if(readLen == -1){
                    break;
                }
                //重置buffer游标
                buffer.flip();
                byte[] bytes = new byte[buffer.remaining()];
                //读取数据到字节数组
                buffer.get(bytes);
                System.out.println("收到了服务器发送的数据 : "+ new
String(bytes,"UTF-8"));
                buffer.clear();
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (null != channel){
                try {
                    channel.close();
                } catch (IOException e) {
```

```
                    e.printStackTrace();
                }
            }
        }
    }
}
```

运行结果：


## AIO介绍：

异步非阻塞IO，当有流可以读时,操作系统会将可以读的流传入read方法的缓冲区,并通知应用程序,对于写操作,OS将write方法的流 写入完毕是操作系统会主动通知应用程序。因此read和write都是异步的，完成后会调用回调函数。

使用场景：连接数目多且连接比较长（重操作）的架构，比如相册服务器。重点调用了OS参与并发操作，编程比 较复杂。Java7开始支持

## 怎样做到透明化远程服务调用？

java代理有两种方式：1）jdk动态代理；2）字节码生成？？？。尽管字节码生成方式实现的代理更为强大和高效，但代码维护不易，大部分公司实现RPC框架时还是选择动态代理方式。**用户像以本地调用方式调用远程服务。**

## rpc通信时消息结构为什么有requestID？

**没有使用requestID前存在的问题：**

- 整个远程调用是异步的，远程调用时让客户端“暂停”，等待结果返回时，再往后执行。
- 如果有多个线程同时远程调用，顺序可能是随机的，client收到server处理完结果，怎么知道哪个消息结果是原先哪个线程调用的。

**解决办法：**

1. 客户端生成唯一requestID
2. 将客户端回调对象和requestID存放到全局ConcurrentHashMap里面put(requestID,callback)；

3. 当线程调用channel.writeAndFlush()发送消息后，紧接着执行callback的get()方法试图获取远程返回的结果。在get()内部，则使用synchronized获取回调对象callback的锁，再先检测是否已经获取到结果，如果没有，然后调用callback的wait()方法，释放callback上的锁，让当前线程处于等待状态
4. 服务端接收到请求并处理后，将response结果(此结果中包含了前面的requestID)发送给客户端，客户端socket连接上专门监听消息的线程收到消息，分析结果，取到requestID，再从前面的ConcurrentHashMap里面get(requestID)，从而找到callback对象，再用synchronized获取callback上的锁，将方法调用结果设置到callback对象里，再调用callback.notifyAll()唤醒前面处于等待状态的线程

```
public Object get() {
    synchronized (this) { // 旋锁
        while (!isDone) { // 是否有结果了
            wait(); //没结果是释放锁，让当前线程处于等待状态
        }
    }
}
```

```
private void setDone(Response res) {
    this.res = res;
    isDone = true;
    synchronized (this) { //获取锁，因为前面wait()已经释放了callback的锁了
        notifyAll(); // 唤醒处于等待的线程
    }
}
```

# 项目中rpc使用那些包实现socket

## *socket可以用哪些操作*