# 背景

为了保证其他模块对自己没有影响，那么最好是与其他模块没有任何关系，即去依赖。如果需要依赖，则弱依赖。

弱依赖有需要被依赖方的返回结果和不依赖返回结果两种。需要结果就要请求后回调，不需要就直接异步化。另外要做好超时和重试、蓄洪、限流、熔断、降级。

如果只能强依赖，人家死了，那就我们报错，但是我们不死。这也需要设置合理超时和重试、蓄洪、限流、熔断、降级。人家又复活了，我们也要立即恢复。

单机系统无法满足需求，采用集群，单机房存在问题，采用多机房，整个地区断网，采用多地区，即异地多活。

涉及集群和跨区问题，就需要考虑策略问题：负载均衡，主从切换，优先策略。

**如何保障系统的高可用？**

我们都知道，单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

保证系统高可用，架构设计的核心准则是：冗余。

有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

# 概述

主要涉及故障检测、故障隔离、故障备份和故障恢复。

## 集群

LB（负载均衡）：LVS/Nginx（http/upstream，stream/upstream）

HA（高可用）：SPoF（Single Point Of Failure）

高可用集群（High Availability Cluster，简称HA Cluster），是指以减少服务中断时间为目的的服务器集群技术。它通过保护用户的业务程序对外不间断提供的服务，把因软件、硬件、人为造成的故障对业务的影响降低到最小程度。

HPC

所谓高可用集群，即当前服务器出现故障时，可以将该服务器中的服务、资源、IP等转移到另外一台服务器上，从而满足业务的持续性；这两台或多台服务器构成了服务器高可用集群。

对于客户端来说，集群就像是一台服务器，因为集群运行的是同一种服务，即使其中有的服务器宕机或无法通信时，也不会对业务造成影响。

## 特征

首先高可用架构应该具备如下特征:

数据库对前端业务透明，业务不会因为数据库故障产生中断。

非主节点的数据应该和主节点的数据实时或者最终保持一致。

当业务因高可用机制发生数据库切换时，切换前后的数据库内容应当一致，不会因为数据缺失或者数据不一致而影响业务。

### 主从切换

高可用最主要的就是出现故障时，能够做到业务的无缝迁移，即主备切换。

当其中一台机器的服务宕机后，对于服务调用者来说，能够迅速的切换到其他可用服务，当服务升级为主服务，这种切换速度应当控制在秒级别（几秒钟）。

当宕机的服务恢复之后，自动变为从服务，主从服务角色切换。主从切换一定是要付出代价的，所以当主服务恢复之后，也就不再替换现有的主服务。

### 负载均衡

当服务的请求量比较高的时候，一台服务不能满足需求，这时候需要多台机器提供同样的服务，将所有请求分发到不同机器上。

高可用架构中应该具有丰富的负载均衡策略和易调节负载的方式。甚至可以自动化智能调节，例如由于机器性能的原因，响应时间可能不一样，这时候可以向性能差的机器少一点分发量，保证各个机器响应时间的均衡。

### 易横向拓展

当用户越来越多，已有服务不能承载更多的用户的时候，便需要对服务进行拓展，扩展的方式最好是不触动原有的服务，对于服务的调用者是透明的。

## 衡量指标

由系统可靠性（Availability）和可维护性（maintainabilit）来度量。

计算方式：HA=MTTF（平均无故障事件）/（MTTF+MTTR（平均修复事件））\*100%

99% 全年服务中断时间不超过4天

99.9% 全年服务中断时间不超过10个小时

99.99% 全年服务中断时间不超过1个小时

99.999% 全年服务中断时间不超过6分钟

## 容灾恢复

在灾难恢复方面，目前业界公认有三个目标值得努力。一是恢复时间，企业能忍受多长时间没有 IT，处于停业状态；二是网络多长时间能够恢复；三是业务层面的恢复。整个恢复过程中，最关键的衡量指标有两个：一个是 RTO，另一个是 RPO。

### RTO

RTO (Recovery Time Objective，复原时间目标)是企业可容许服务中断的时间长度。它是指灾难发生后，从 IT 系统当机导致业务停顿之时开始，到 IT 系统恢复至可以支持各部门运作、恢复运营之时，此两点之间的时间段称为 RTO。

比如说灾难发生后半天内便需要恢复，RTO值就是十二小时。

### RPO

RPO (Recovery Point Objective，复原点目标)是指从系统和应用数据而言，要实现能够恢复至可以支持各部门业务运作，系统及生产数据应恢复到怎样的更新程度。这种更新程度可以是上一周的备份数据，也可以是上一次交易的实时数据。

### 选择标准

根据以上两个简单的原则，企业不但可以对现有的数据系统作出，也可以按照既定的RTO及RPO要求，选购最适合的灾备方案。RTO及RPO与方案售价有着密切的关系，然而完美的方案当然是RTO及RPO皆为零，表示当灾难发生后，系统立即恢复，而且完全没有数据丢失，可是其造价是非常昂贵的，而且也不一定有这个必要。因此，最佳方案必需在RTO，RPO，维护及价钱多方面，都能达致平衡。尤其是中小企业，在资源紧拙的情况，应先好好了解对RTO及RPO的要求，然后再看看价钱，那就比较容易找到，适合企业的方案了。

在考虑采用哪个指标之前，IT 人首先要弄清楚一个基本概念，企业的容灾系统预防的是什么灾害，是多少年一遇的，能忍受多少损失，需要算出一个大概的成本，当然不一定很精确。其次，无论企业容灾系统是采用冷备、热备、温备、还是磁盘备份，几分钟恢复业务和几天恢复业务效果是完全不一样的。企业需要明确对恢复时间的容忍底限是多少。再从灾备本身的意义来讲，无论采用哪种衡量指标，最终目的是要能够很好地检验灾备系统的实用性能，否则就失去建立灾备的意义了。而灾备最核心的作用就是确保灾难发生后业务能够连续运行，交易中的数据完整保存，丢失越少越好。因此业务层面的恢复，企业要有一个底限。参考世界范围内一系列灾难恢复经验，国家之间的差别非常大。比如在美国，政府是第一位的，警察局对数据的恢复要求特别高。而在中国，无论什么性质，银行始终是排在第一位的。

作为银行，除开展自身业务之外，更多数据来自上下级银行间的财务汇兑与结算。站在管理者的位置上，一旦灾难发生，最重要的是在尽可能短的时间内排除障碍，恢复业务，保证系统做到连续运行。因此，从这个角度出发，银行容许系统停滞的时间应当越短越好。选择 RTO 刚好合适。但是，RTO 对成本要求太高，与回报似乎不成正比。企业资金不可能无限制地投入到一个灾备系统中。对于银行证券这样的联机交易事故处理非常紧密的金融机构而言，可能每一笔、每一单、每一分钱都很重要，所以都需要恢复。RPO 显然更为合适。许多时候进行选择并不意味着非此即彼，这与现实婚姻中一夫一妻的限制还是有差别的。RTO 和 RPO 对银行来讲都很重要。RTO 越短、RPO 越新，银行面临的损失就越小，但这也意味着系统开发成本将会急剧上升。许多时候，最佳的容灾解决方案却不一定是效益最好的。反之亦是。如何去平衡这中间的关系，不仅是门学问，更像是艺术。

# 集群故障

## 宕机

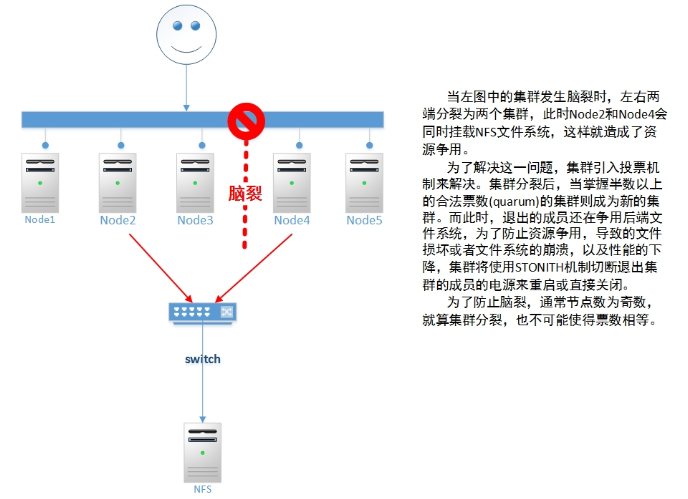
一般来说，“运行环境”是排名第一的宕机类别，大约35%的事件属于这一类。运行环境可以看作是支持数据库服务器运行的系统和资源集合，包括操作系统、硬盘以及网络等。性能问题紧随其后，占大约35%，然后是复制，占20%，最后剩下的10%包含各种类型的数据丢失和损坏，以及其他问题。

以下问题需要注意：

1. 在运行环境的问题中，最普遍的问题时磁盘空间耗尽；
2. 在性能问题中，最普遍的宕机原因却是是运行很糟糕的SQL，但也不一定都是这个原因，比如也有很多问题时由于服务器bug或错误的行为导致的；
3. 糟糕的Schema和索引设计是第二大影响性能的问题；
4. 复制问题通常是由于主备数据不一致导致；
5. 数据丢失问题通常由于DROP TABEL的误操作导致，并总是伴随着缺少可用备份的问题。

## 脑裂

### 概述



脑裂：因某种特殊原因造成集群分裂成两个小集群，而这两个小集群互相不能正常通信，此时，就会发生闹裂（Brain Split）现象。

脑裂是因为集群分裂导致的，集群中有节点因为处理器忙或者其他原因暂时停止响应时，与其他节点间的心跳出现故障，但这些节点还处于active状态，其他节点可能误认为该节点"已死"，从而争夺共享资源（如共享存储）的访问权，分裂为两部分独立节点。

**脑裂后果：**这时两个节点开始争抢共享资源，结果会导致系统混乱，数据损坏。

### 解决

**脑裂解决：**投票和参考节点的方法也能一定程度上解决脑裂的问题，但完全解决还需要资源隔离（Fencing）。

#### 投票/仲裁

当有节点（一个或多个）和另外节点互相接收不到对方心跳信息时，如何决定哪一部分节点是正常运行的，而哪一部分是出现故障需要隔离的呢（避免集群脑裂）？

这时候通过法定票数（quorum）决定，即当有节点故障时，节点间投票决定哪个节点是有问题的，票数大于半数为合法。

**票数：**

每个节点可以设置票数，即决定节点在集群内是否合法（正常）的权限值，这个是可以有多有少的，例如有些节点的性能较好或有其他优势，可以设置较多的票数。

**法定票数（quorum）：**

当一个节点能和另一个节点保持心跳信息，该节点就获取得了另一个节点的票数，该节点获得的所有票数就是法定票数。

关于“投票”，有必要知道著名的Pasox算法和Zookeeper：

**Paxos算法：**

Paxos算法解决的是保证集群中每个节点执行相同的操作序列，可以保证分布式群集中的数据一致性。

例如，通过投票来对写操作进行全局编号，同一时刻，只有一个写操作被批准，同时并发的写操作要去争取选票，只有获得过半数选票的写操作才会被 批准（所以永远只会有一个写操作得到批准）；而其他的写操作竞争失败只好再发起一轮投票，就这样，在日复一日年复一年的投票中，所有写操作都被严格编号排序。

编号严格递增，当一个节点接受了一个编号为100的写操作，之后又接受到编号为99的写操作（因为网络延迟等很多不可预见原因），它马上能意识到自己数据不一致了，自动停止对外服务并重启同步过程。

任何一个节点挂掉都不会影响整个集群的数据一致性（总2n+1台，除非挂掉大于n台）。

**Zookeeper：**

Zookeeper是 Hadoop 大数据生态的一个独立组件，是 Google 的 Chubby一个开源的实现，可以说是Paxos算法（类似）的实现。

Zookeeper主要提供分布式协调服务，分布式应用程序可以基于它实现服务注册（高可用），同步服务，配置维护和命名服务等。

 Zookeeper真正提供的是类似我们普通电脑上的文件系统目录的功能，不过可以原子的进行增/删/改/查操作；具体要实现什么分布式协调服务，需要自己写程序来操作Zookeeper上的“目录”。

Zookeeper为什么可以作为分布式系统提供协调服务？

最主要的是Zookeeper本身运行的是一个由多个Zookeeper节点组成的稳定的高可用集群。

Zookeeper集群的高可用性和各节点“目录”数据的一致性正是基于 类似 Paxos算法实现的投票机制来保证的。

所以 Zookeeper集群节点数量最好也是单数（2n+1），当集群脑裂分区时，分区节点数量不超过一半的（<n+1），会自动停止对外服务。

比如：

 5台ZK节点组成ZK集群，当分成2台和3台的两个分区时，2台的分区会自动停止对外服务，3台的分区会继续提供服务。

另外，如果用6台节点组成ZK集群，当分成3台和3台的两个分区时，这两个分区都自动停止对外服务，所以，容错率和5台节点组成的集群的是一样的，更应该用单数（2n+1）节点数量组成集群。

#### 参考节点

比较特殊的是只有两个节点的集群，或两边票数相等

这时候可以借助另外的参考节点，如ping网关（可以是一个节点），可以和测试点ping通，但不可以和对方通，说明对方节点有问题，或本节点有问题；还有就是通过仲裁设备，如仲裁磁盘，每个节点都间隔一定时间不停往磁盘写数据，若监测到对方不再写入的时候，可能对方节点出故障。

 但最好还是使得组成集群的节点数量为单数台（2n+1），当集群分区脑裂时，节点数量小于一半（>n+1）的分区自动停止对外提供服务。

#### 资源隔离（Fencing）

当不能确定某个节点的状态时，通过fencing把对方干掉，确保共享资源被完全释放，前提是必须要有可靠的fence设备。

STONITH(Shoot The Other Node in the Head，“爆头”)，这种方式直接操作电源开关，当一个节点发生故障时，另一个节点如果能侦测到，就会通过网络发出命令，控制故障节点的电源开关，通过暂时断电，而又上电的方式使故障节点被重启动或者直接断电，这种方式需要硬件支持。

如果备份节点在某一时刻不能收到主节点的心跳信息时，那么如果此时备份节点立刻抢占资源时，而此时主节点正好在执行写操作，备份节点一旦也执行相应的写操作，会导致文件系统错乱或者服务器崩溃，因此在抢占资源的时候可以使用资源隔离机制来防止此类事件发生。而我们常常使用stonithd（即爆头）来使主节点不在抢占资源。

其中资源隔离包括：

**节点级别：**

STONITH（shoot the other node in the head，爆头。硬件方式），直接控制故障节点的电源，绝对彻底。

**资源级别：**

例如：FC SAN switch（软件方式）可以实现在存储资源级别拒绝某节点的访问。

## 资源争用

资源争用：当一个集群中因特殊情况分裂成两个小集群，且这两个集群都不能通信时，这时可能会造成资源争用的情况；分裂情况发生后，如果没有及时的决策，那么可能会因为两个小集群同时使用一个文件系统，而造成后端共享存储中文件损坏，甚至造成整个文件系统的崩溃。显然，这种情况是不允许发生的。

## 资源隔离

资源隔离：主要为了解决资源争用的问题。资源隔离分为节点级别隔离和资源级别隔离。

所谓节点级别隔离指当集群发生分裂时，即发生脑裂现象后，通过STONITH机制将资源隔离，并通过仲裁机制将分裂的票数不足的集群退出集群。STONITH指通过硬件设备，使得退出的主机重启或关机，或者通过交换机阻断退出的集群向外通信和资源通信的能力。

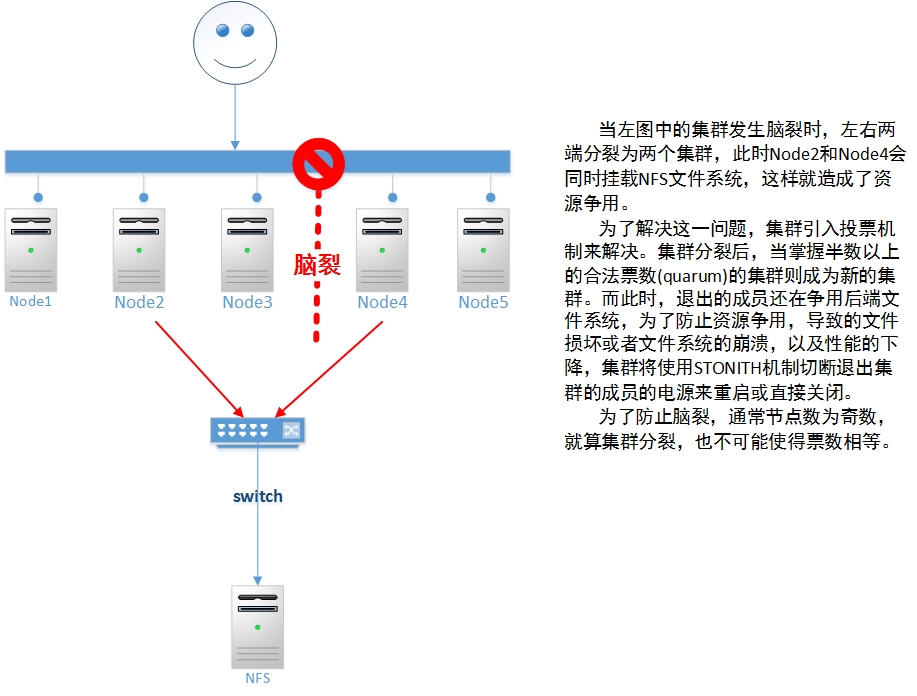
**资源隔离的解决方案：**

1、当集群分裂成两个小集群时会发生资源争用的情况，为避免争用后端存储系统而造成灾难性的系统崩溃，集群系统引入了投票机制，只有拥有半数以上合法票数的集群才能存活，否则就推出集群系统。

2、当集群为偶数时，如果分裂，两边可能都掌握相等的票数；因此，集群系统不应该为偶数，如果是偶数则需要一个额外的ping节点参与投票。

3、票数不足的集群退出集群服务后，为了保证它不会争用资源需要STONITH机制来进行资源隔离。

所以，为了防止脑裂，集群节点数一般为奇数，就算集群分裂，也不可能使得两个集群的票数相等。



# 高可用集群实现方式

实现高可用集群有三种方式：

## 主从方式（非对称）

这种方式组建的高可用集群通常包含2个节点和一个或多个服务器，其中一台作为主节点（active），另一台作为备份节点（standy）。备份节点随时都在检测主节点的健康状况，当主节点发生故障时，服务会自动切换到备份节点上以保证服务正常运行。

这种方式下的高可用集群其中的备份节点平时不会启动服务，只有发生故障时才会有用，因此感觉比较浪费。

## 对称方式

这种方式一般包含2个节点和一个或多个服务，其中每一个节点都运行着不同的服务且相互作为备份，两个节点互相检测对方的健康状况，这样当其中一个节点发生故障时，该节点上的服务会自动切换到另一个节点上去。这样可以保证服务正常运行。

## 多机方式

这种集群包含多个节点和多个服务。每一个节点都可能运行和不运行服务，每台服务器都监视着几个指定的服务，当其中的一个节点发生故障时，会自动切换到这组服务器中的一个节点上去。

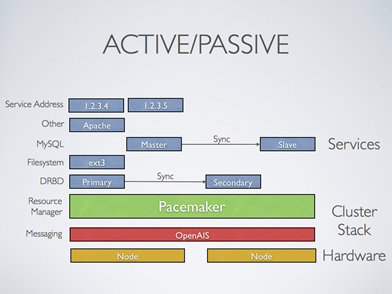
## ----

**工作模型：**

## 主备模型

主备模型（Active/Passive，主从方式（非对称）），一个活动主节点，另一个不活动作为备用节点，当主节点故障，转移到备节点，这时备节点就成为了主节点。备节点完全冗余，造成一定浪费。

mysql、DRBD主从节点间还要进行同步：



## 双主模型

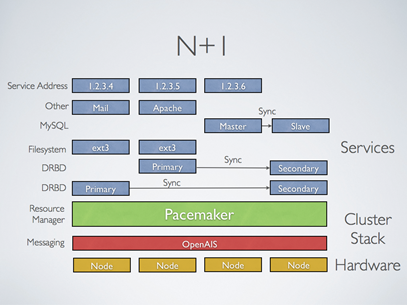
双主模型（对称方式），两个节点都是活动的，两个节点运行两个不同的服务，也互为备用节点。也可以提供同一个服务，比如ipvs，前端基于DNS轮询。这种模型可以使用比较均衡的主机配置，不会造成浪费。

## 多机模式

### N+1

多机模式（N+1）,N个活动主节点N个服务，一个备用节点。这需要额外的备用节点必须能够代替任何主节点，当任何主节点故障时，备节点能够负责它的角色对外提供相应的服务。

如下图，最后一个备用节点可以作为前两台主节点的DRBD和第三台主节点的MYSQL提供备用功能：



### N+M

N个活动主节点，M个备用节点。像上面的N+1模型，一个备用节点可能无法提供足够的备用冗余能力，备用节点的数量M是成本和可靠性要求之间的折衷。

也有一种说法：N-M: N个节点M个服务， N>M， 活动节点为N， 备用节点为N-M。

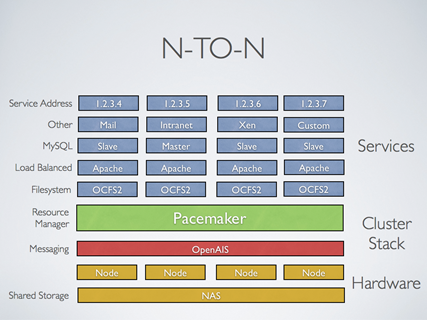
### N-to-1

这和N+1一样，也是N个活动主节点，一个备用节点；不同是的备用节点成为主节点只是暂时的，当原来故障的节点修复后，必须回转才能正常工作。

### N-to-N

N个主节点N个备用节点。这是A/A双主和N + M模型的组合，N节点都有服务，如果一个坏了，剩下的每个节点都可以作为替代提供服务。

如下图，当共享存储是可用的，每一个节点都可能会被用于故障切换。起搏器甚至可以运行服务的多个副本，以分散工作量。



# 高可用集群实现原理

高可用集群主要实现自动侦测(Auto-Detect)故障、自动切换/故障转移(FailOver)和自动恢复(FailBack)。

简单来说就是，用高可用集群软件实现故障检查和故障转移（故障/备份主机切换）的自动化，当然像负载均衡、DNS分发也可提供高可性。

## 实现高可用性

### 提升平均失效时间（MTBF）

### 降低平均恢复时间（MTTR）

提升系统可用性的解决方案之降低MTTR：

手段：冗余redundant

active/passive：主备

active/active：双主

active🡪HEARTBEAT🡪passive

active🡨🡪HEARTBEAT🡨🡪active

## 避免单点失效

单点是系统高可用的大敌，单点往往是系统高可用最大的风险和敌人，应该尽量在系统设计的过程中避免单点。

方法论上，高可用保证的原则是“集群化”，或者叫“冗余”：只有一个单点，挂了服务会受影响；如果有冗余备份，挂了还有其他backup能够顶上。

保证系统高可用，架构设计的核心准则是：冗余。

有了冗余之后，还不够，每次出现故障需要人工介入恢复势必会增加系统的不可服务实践。所以，又往往是通过“自动故障转移”来实现系统的高可用。

**典型互联网架构中，通过冗余+自动故障转移来保证系统的高可用特性。**

### 共享存储或磁盘复制

### MySQL同步复制

### 基于复制的冗余

## 自动侦测

自动侦测阶段由主机上的软件通过冗余侦测线，经由复杂的监听程序，逻辑判断，来相互侦测对方运行的情况。

常用的方法是：集群各节点间通过心跳信息判断节点是否出现故障。

## 故障转移和故障恢复

**自动切换**

自动切换阶段某一主机如果确认对方故障，则正常主机除继续进行原来的任务，还将依据各种容错备援模式接管预先设定的备援作业程序，并进行后续的程序及服务。

通俗地说，即当A无法为客户服务时，系统能够自动地切换，使B能够及时地顶上继续为客户提供服务，且客户感觉不到这个为他提供服务的对象已经更换。

通过上面判断节点故障后，将高可用集群资源（如VIP、httpd等，下面详见）从该不具备法定票数的集群节点转移到故障转移域（Failover Domain，可以接收故障资源转移的节点）。

**自动恢复**

自动恢复阶段在正常主机代替故障主机工作后，故障主机可离线进行修复工作。在故障主机修复后，透过冗余通讯线与原正常主机连线，自动切换回修复完成的主机上。

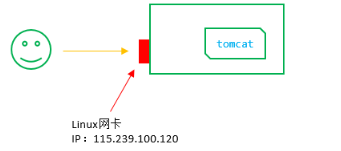
### 提升备库或切换角色

### 浮动IP/虚拟IP地址或IP接管

#### 背景

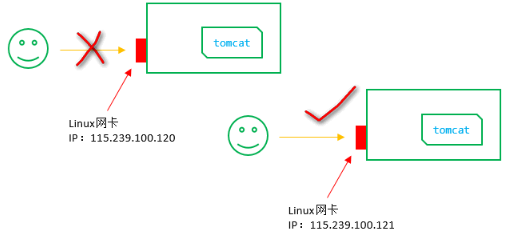
现在有一个场景，在一台Linux上部署一个web应用，应用跑在tomcat里面，linux网卡上的ip是115.239.100.120

大致就是如下的部署关系：



用户通过输入IP地址就能访问这个应用了，但是天有不测风云，有时候因为一些原因，服务会挂掉。于是开发人员就想了一个办法，在另外一个Linux上，部署同样的一个应用，这样这个服务挂了，另外一个顶上，于是架构就变成了下面。

这个样子：



如果用户访问100.120访问不了，就访问100.121就行了，但是这样引入了一个问题，用户需要记住两个IP，很不方便！

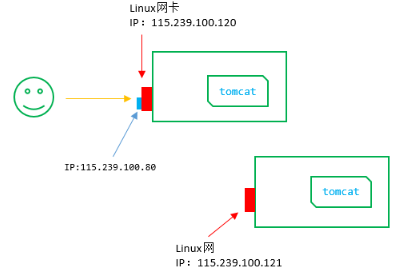
所以这个时候就需要浮动IP了。

#### 概述

虚拟IP地址（浮动IP）就是一个未分配给真实主机的IP，也就是说对外提供数据库服务的主机除了有一个真实的IP外还有一个虚拟IP，使用跟着两个IP中的任意一个都可以连接到这台主机，所有项目中的数据库连接一项配置的都是这个虚拟IP，当服务器发生故障无法对外提供服务时，动态将这个虚拟IP切换到备用主机。

上面我们搭建了两个一模一样的服务，但是用户需要记住两个IP，实在是太麻烦了，于是我们可以采用另外一个策略，

我们给其中一个机器在添加一个IP，让用户访问这个IP， 一个网卡是可以 添加多个IP的。



例如上面，我们给100.120主机添加一个100.80的IP，用户访问115.239.100.80这个IP，如果这台机器的服务挂了，我们就将

这个ip转移到100.121这个机器上，这样对于用户而言，他们始终访问的就只是一个IP而已。

在上面的场景中：115.239.100.80这个ip就是浮动IP，他会随着主机服务挂掉而转移到另外一个能够提供相同服务的主机上，

对于用户而言，他们始终觉得服务是正常的。一般而言，在高可用集群中使用的比较多，例如LVS集群等。

其实一个高可用集群一般而言都有浮动IP的，如果没有，机器宕机了还怎么玩？

#### 配置

利用单个网卡可以绑定多个IP地址的技术。

1、主服务器配置浮动IP

在主服务器上拷贝eth0位eth0:1，并做修改。

cp /etc/sysconfig/network-scripts/ifcfg-eth0 /etc/sysconfig/network-scripts/ifcfg-eth0:1

编辑文件ifcfg-eth0:1：

vim /etc/sysconfig/network-scripts/ifcfg-eth0:1

修改DEVICE为eth0:1，NM\_CONTROLLED设置为no，IPADDR改为浮动IP地址172.24.8.80，去掉网关信息，修改完如下：

DEVICE="eth0:1"

BOOTPROTO="static"

ONBOOT="yes"

NM\_CONTROLLED="no"

TYPE="Ethernet"

IPADDR="172.24.8.80"

NETMASK="255.255.255.0"

然后启动该网卡，验证是否生效：

ifup eth0:1

查看是否有浮动IP地址：

ip addr

接着访问浮动IP地址172.24.8.80，能访问成功，即和主服务器172.24.8.55访问的一样，则说明配置成功。

2、从服务器配置浮动IP

配置步骤和主服务器完全一样，eth0:1的IP地址同样设置成浮动IP172.24.8.80，在启动网卡测试之前，需在主服务器关闭浮动IP：

ifdown eth0:1

然后在从服务器启动eth0:1

ifup eth0:1

测试步骤也和主服务器一样。

### 中间件

mycat 实现mysql高可用的中间价。

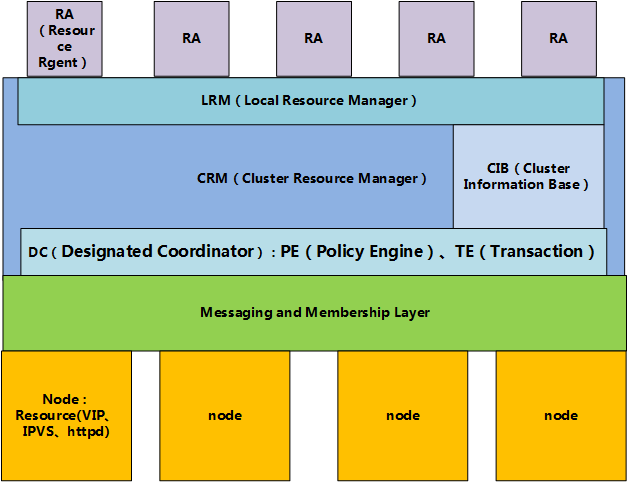
# 高可用集群架构及组件

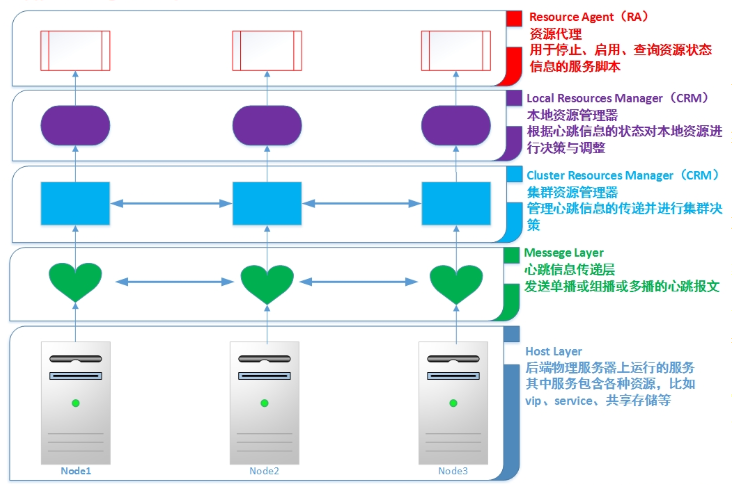
## 架构高可用

架构高可用的手段：

1. 设计无状态
2. 子系统冗余
3. 幂等性设计
4. 异步调用
5. 超时机制
6. 分级管理
7. 服务降级

高可用集群架构层次：





## 节点主机层

这一层主要是正在运行在物理主机上的服务，高可用集群相关的软件运行在各主机上，集群资源也是在各主机上。

## Messaging and Membership Layer

Messaging Layer：可以理解为信息层，主要的作用是传递当前节点的心跳信息，并告知给对方，这样对方就知道其他节点是否在线。如果不在线，则可以实现资源转移，这样另一台节点就可以充当主节点，并正常提供服务。传递心跳信息一般使用一根心跳线连接，该线接口可以使用串行接口也可以是以太网接口来连接。每一个节点上都包含信息层。

可以提供该组件的软件有：

1、heartbeat

heartbeat有三个版本即heartbeat v1、heartbeat v2和heartbeat v3

heartbeat v1是比较老的版本，heartbeat v2是目前稳定的版本，在做实验的时候使用该版本。

2、corosync(openAIS的子项目)

3、keepalive

4、cman

Heartbeat是比较常用的软件，Keepalived配置相对比较简单，而ultramonkey好像不怎么常用，Corosync比heartbeat功能还要强大，功能更加丰富。

信息传递层，传递集群信息的一种机制，通过监听UDP 694号端口，可通过单播、组播、广播的方式，实时快速传递信息，传递的内容为高可用集群的集群事务，例如：心跳信息，资源事务信息等等，只负责传递信息，不负责信息的计算和比较。

成员关系（Membership）层，这层最重要的作用是主节点（DC）通过Cluster Consensus Menbership Service（CCM或者CCS）这种服务由Messaging层提供的信息，来产生一个完整的成员关系。这层主要实现承上启下的作用，承上，将下层产生的信息生产成员关系图传递给上层以通知各个节点的工作状态；启下，将上层对于隔离某一设备予以具体实施。

## CRM（Cluster Resource Manager）

集群资源管理器层，它主要是用来提供那些不具有高可用的服务提供高可用性的。它需要借助Messaging Layer来实现工作，因此工作在Messaging Layer上层。

资源管理器的主要工作是收集messaging Layer传递的节点信息，并负责信息的计算和比较，并做出相应的动作，如服务的启动、停止和资源转移、资源的定义和资源分配。

在每一个节点上都包含一个CRM，且每个CRM都维护这一个CIB（Cluster Information Base，集群信息库），只有在主节点上的CIB是可以修改的，其他节点上的CIB都是从主节点那里复制而来的。

CRM会推选出一个用于计算和比较的节点，叫DC（Designated coordinator）指定协调节点，计算由PE（Policy Engine）策略引擎实现，计算出结果后的动作控制由TE（Transition Engine）事务引擎实现。

在每个节点上都有一个LRM（local resource manager）本地资源管理器，是CRM的一个子功能，接收TE传递过来的事务，在节点上采取相应动作，如运行RA脚本等。

## LRM

LRM：Local Resource Messager，叫做本地资源管理器，它是CRM的一个子组件，用来获取某个资源的状态，并且管理本地资源的。例如：当检测到对方没有心跳信息时，则会启动本地相应服务。

## DC

DC：可以理解为事务协调员，这个是当多个节点之间彼此收不到对方的心跳信息时，这样各个节点都会认为对方发生故障了，于是就会产尘分裂状况（分组）。并且都运行着相关服务，因此就会发生资源争夺的状况。因此，事务协调员在这种情况下应运而生。事务协调员会根据每个组的法定票数来决定哪些节点启动服务，哪些节点停止服务。 例如高可用集群有3个节点，其中2个节点可以正常传递心跳信息，与另一个节点不能相互传递心跳信息，因此，这样3个节点就被分成了2组，其中每一个组都会推选一个DC，用来收集每个组中集群的事务信息，并形成CIB，且同步到每一个集群节点上。同时DC还会统计每个组的法定票数（quorum），当该组的法定票数大于二分之一时，则表示启动该组节点上的服务；否则停止该节点上的服务。对于某些性能比较强的节点来说，它可以投多张票，因此每个节点的法定票数并不是只有一票，需要根据服务器的性能来确定。DC一般位于主节点上。

## PE和TE

PE和TE也是DC的子组件，其中：

PE（Policy Engine）：策略引擎，来定义资源转移的一整套转移方式，但只是做策略者，并不亲自来参加资源转移的过程，而是让TE来执行自己的策略。

TE（Transition Engine）： 就是来执行PE做出的策略的并且只有DC上才运行PE和TE。

## stonithd组件

STONITH(Shoot The Other Node in the Head，”爆头“)， 这种方式直接操作电源开关，当一个节点发生故障时，另 一个节点如果能侦测到，就会通过网络发出命令，控制故障节点的电源开关，通过暂时断电，而又上电的方式使故障节点被重启动或者直接断电， 这种方式需要硬件支持。

如果备份节点在某一时刻不能收到主节点的心跳信息时，那么如果此时备份节点立刻抢占资源时，而此时主节点正好在执行写操作，备份节点一旦也执行相应的写操作，会导致文件系统错乱或者服务器崩溃，因此在抢占资源的时候可以使用资源隔离机制来防止此类事件发生。而我们常常使用stonithd（即爆头）来使主节点不在抢占资源。

其中资源隔离包括：

1、节点级别

使用stonithd设备来实现

2、资源级别

例如：使用FC SAN switch可以实现在存储资源级别拒绝某节点的访问

## 共享存储

对于某些服务如http、mysql等服务，需要将某些数据共享，这样当使用不同的节点来访问存储设备时，都可以返回正确的信息。如果不使用存储设备，假设http服务为例，当某个客户想访问某个图片时，如果这个图片只放在某个指定的服务器上时，一旦该服务器挂了，http服务就会切换到另一台设备上去，而另一台设备上面没有该图片，那么该用户此时就不能访问该图片了，当然这种情况是我们不想看到了。为了解决这类事件发生，可以使用共享存储设备，将相关的数据放在共享设备上，这样无论那一台服务器挂了，都不会影响用户的访问。

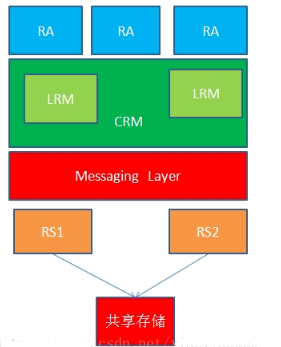
常用的共享存储设备有如下三种：

DAS:Direct Attached Storage,直接附加存储

NAS：Network Attached Storage，网络附加存储

SAN：Storage Area Network，存储区域网络

因此，一个高可用集群服务的组件架构大概是这样子的：



## 资源

其实资源就是启动一个服务需要的子项目。例如启动一个httpd服务，需要ip，也需要服务脚本、还需要文件系统（用来存储数据的），这些我们都可以统称为资源。因此，实现一个高可用集群一般需要ip、服务（脚本）和文件系统（存储数据），当然有些高可用集群不需要存储设备的。

资源也是有类型的，可以分为这样几类：

1、primitive：可以理解为主资源，有时候看到的会是native，都是一个意思，该资源只在主节点上有。（当然备份节点一旦将资源夺过来了，也就成了主节点，因此，主节点是相对来说的）

2、group：组资源，将多个资源绑定在一个同一个组上面且运行在同一个节点上。

3、clone：是将primitive资源克隆n份且运行在每一个节点上

4、master/slave：也是将primitive克隆2份、其中master和slave节点各运行一份，且只能在这2个节点上运行。

对于某些集群服务来说，启动相关的资源是有先后顺序的。例如启动一个mysql集群服务，首先应该先挂载共享存储设备，否则即时mysql服务启动起来了，用户也访问不了数据。因此，一般说来，我们需要将资源进行约束。资源约束有如下几类：

1、位置约束（location）：资源对节点的倾向程度，通常可以使用一个分数（score）来定义，当score为正值时，表示资源倾向与此节点；负值表示资源倾向逃离于此节点。也可以将score定义为-inf(负无穷大)和inf（正无穷大）。例如：有三个节点rs1、rs2、rs3当rs1是主节点且发生故障时，则比较rs2和rs3的score值，谁是正值，则资源将会转移到哪个节点上去。

2、排列约束（colocation）：用来定义资源是否可以在一起，通常也是使用一个score来定义的。当score是正值表示资源可以在一起；否则表示不可以在一起。通过定义资源类型为group也可以来将所有资源绑定在一起。

3、顺序约束（order）：用来定义资源启动和停止的顺序。例如，首先应该先挂载共享存储，在启动httpd或mysqld服务才行吧。

**资源粘性：**用来定义资源是否倾向留在该节点。通常使用score来定义，当score为正数表示乐意留在当前节点，负数表示不乐意留在当前节点。

当某个高可用集群即包含资源粘性又包含位置约束，一旦该节点发生故障后，资源就会转移到另一个节点上去。但是当之前的节点恢复正常时，需要比较所有的资源粘性之和与所有位置约束之和谁大谁小，这样资源才会留在大的一方。

**资源转移**

将有故障节点的VIP设置到另一个节点上去，并在另一个节点启用相应的服务，挂载相应的存储设备等等都可以叫做资源转移。

## RA（Resource Rgent）

资源代理层，简单的说就是能够集群资源进行管理的脚本，如启动start，停止stop、重启restart和查询状态信息status等操作的脚本。LRM本地资源管理器负责运行。

资源代理分为：

1、Legacy heartbeat（heatbeat v1版本的资源管理）；

2、LSB（Linux Standard Base），主要是/etc/init.d/\*目录下的脚本start/stop/restart/status；

3、OCF（Open Cluster Famework），比LSB更专业，更加通用，除了上面的四种操作，还包含monitor、validate-all等集群操作。

4、STONITH：实现节点隔离

# 高可用集群软件

## Messaging Layer集群信息层软件

1、heartbeat (v1, v2)

2、heartbeat v3

可以拆分为：heartbeat, pacemaker, cluster-glue

3、corosync

从OpenAIS分离的项目。

4、cman

5、keepalived

一般用于两个节点的集群。

6、ultramokey

## CRM集群资源管理器软件

1、Haresource

heartbeat v1 v2包含，使用文本配置接口haresources

2、crm

heartbeat v2包含，可以使用crmsh或者heartbeat-gui来进行配置

3、pacemaker

heartbeat v3分离出来的项目，配置接口：CLI：crm、pcs和GUI：hawk(WEB-GUI)、LCMC、pacemaker-mgmt、pcs

4、rgmanager

Cman包含，使用rgmanager(resource group manager)实现管理, 具有Failover Domain故障转移域这一特性，也可以使用RHCS（Redhat Cluster Suite）套件来进行管理：Conga的全生命周期接口，Conga（luci/ricci）先安装后，可用其安装高可用软件，再进行配置。

## LVS

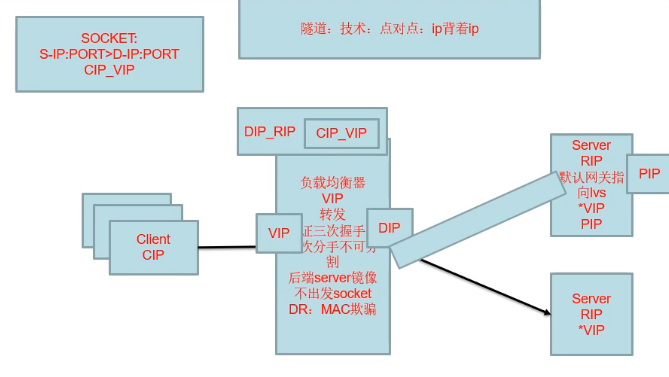
### 简介

LVS是Linux Virtual Server的简写，意即Linux虚拟服务器，是一个虚拟的服务器集群系统。

LVS由用户空间的ipvsadm和内核空间的IPVS组成，ipvsadm用来定义规则，IPVS利用ipvsadm定义的规则工作。

**四层（传输层**，位于内核）：快，**应对更大的流量**。

四层：TUN/VPN



### 特点

高并发连接：LVS基于内核网络层面工作，有超强的承载能力和并发处理能力。单台LVS负载均衡器，可支持上万并发连接。

稳定性强：是工作在网络4层之上仅作分发之用，这个特点也决定了它在负载均衡软件里的性能最强，稳定性最好，对内存和cpu资源消耗极低。

成本低廉：硬件负载均衡器少则十几万，多则几十万上百万，LVS只需一台服务器和就能免费部署使用，性价比极高。

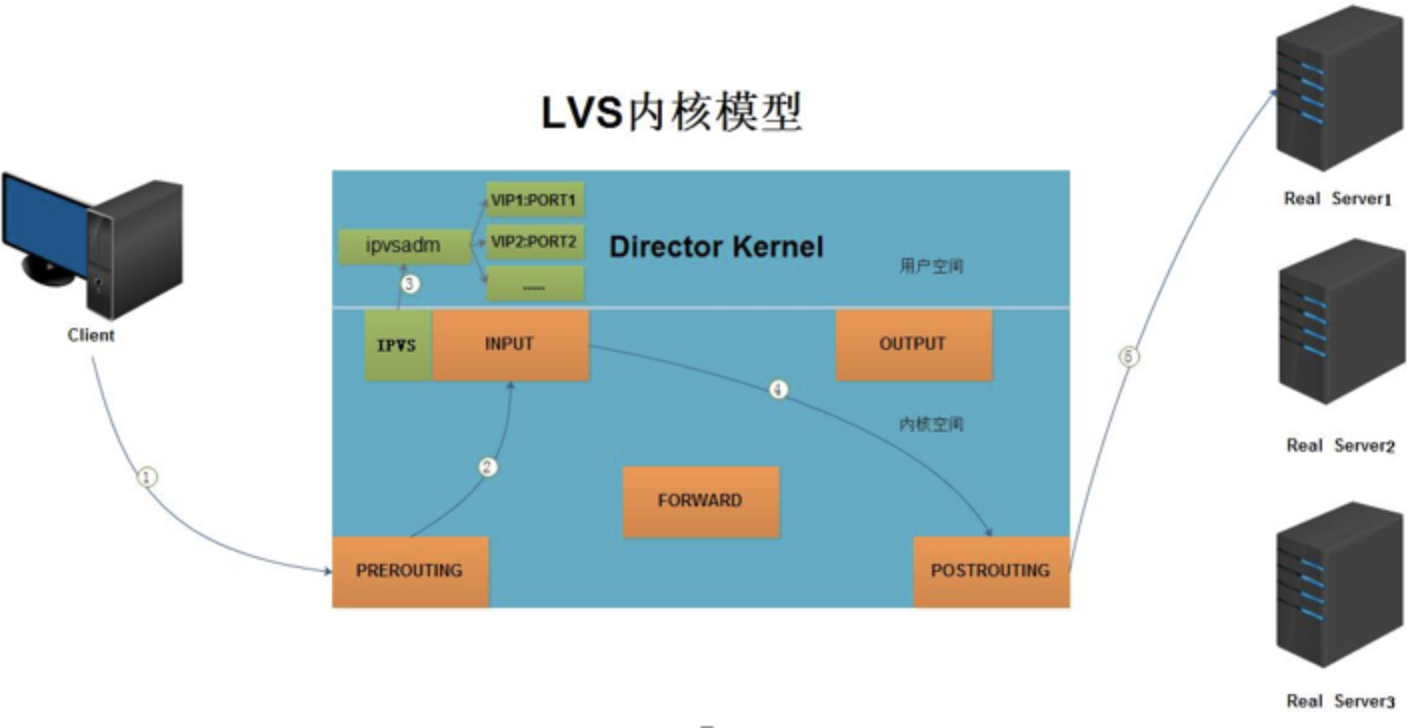
配置简单：LVS配置非常简单，仅需几行命令即可完成配置，也可写成脚本进行管理。

支持多种算法：支持多种调度算法，可根据业务场景灵活调配进行使用。

支持多种工作模型：可根据业务场景，使用不同的工作模式来解决生产环境请求处理问题。

应用范围广：因为LVS工作在4层，所以它几乎可以对所有应用做负载均衡，包括http、数据库、DNS、ftp服务等等。缺点：工作在4层，不支持7层规则修改，机制过于庞大，不适合小规模应用。

### 原理



LVS工作原理：

1、当客户端的请求到达负载均衡器的内核空间时，首先会到达PREROUTING链。

2、当内核发现请求数据包的目的地址是本机时，将数据包送往INPUT链。

3、LVS由用户空间的ipvsadm和内核空间的IPVS组成，ipvsadm用来定义规则，IPVS利用ipvsadm定义的规则工作，IPVS工作在INPUT链上,当数据包到达INPUT链时，首先会被IPVS检查，如果数据包里面的目的地址及端口没有在规则里面，那么这条数据包将经过INPUT链送至用户空间，交给用户空间的进程来处理。

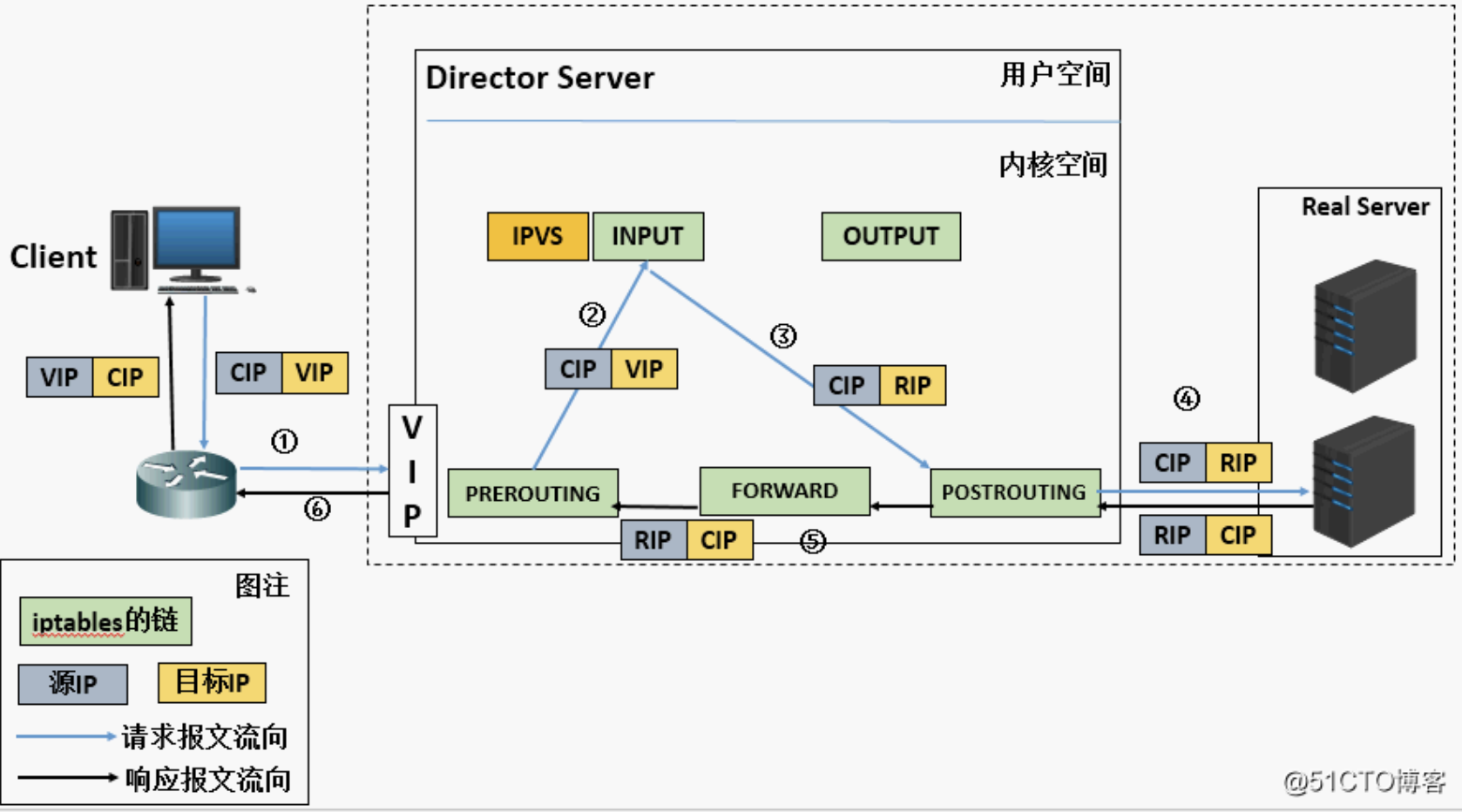
4、如果数据包里面的目的地址及端口在规则里面，那么这条数据报文将被修改目的地址为事先定义好的后端服务器，并送往POSTROUTING链。

5、最后经由POSTROUTING链发往后端服务器。

iptables的五连四表：<https://blog.csdn.net/gui951753/article/details/86619021>

### 工作模式

#### DR



LVS-DR模型的特性

1、保证前端路由将目标地址为VIP报文统统发给Director Server，而不是RS  
解决方案：

在前端路由器做静态地址路由绑定，将对于VIP的地址仅路由到Director Server

存在问题：用户未必有路由操作权限，因为有可能是运营商提供的，所以这个方法未必实用

arptables：在arp的层次上实现在ARP解析时做防火墙规则，过滤RS响应ARP请求。这是由iptables提供的

修改RS上内核参数（arp\_ignore和arp\_announce）将RS上的VIP配置在lo接口的别名上，并限制其不能响应对VIP地址解析请求。（最容易实现）

2、RS可以使用私有地址；也可以是公网地址，如果使用公网地址，此时可以通过互联网对RIP进行直接访问

3、RS跟Director Server必须在同一个物理网络中

4、所有的请求报文经由Director Server，但响应报文必须不能进过Director Server

5、不支持地址转换，也不支持端口映射

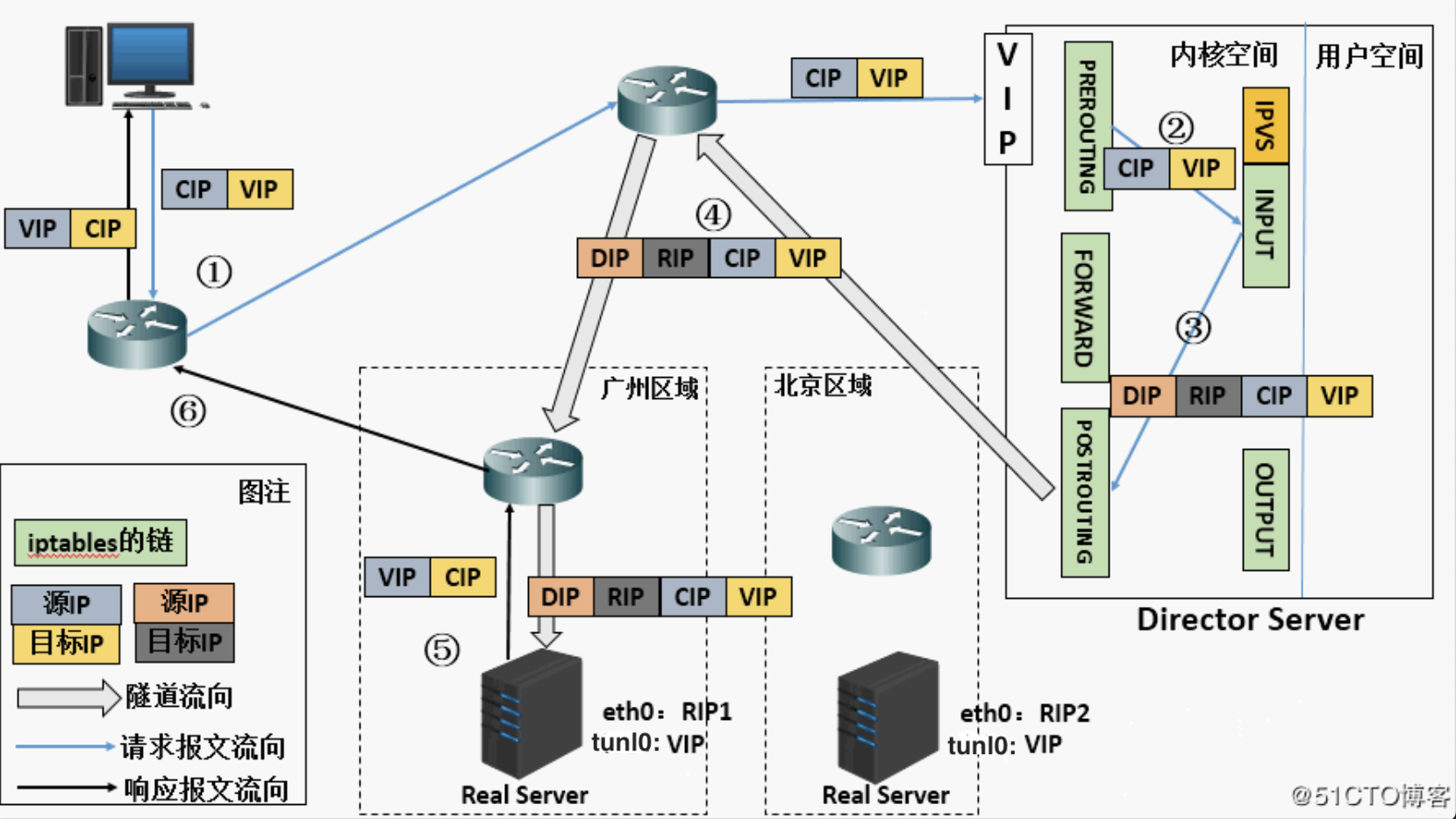
6、RS可以是大多数常见的操作系统

7、RS的网关绝不允许指向DIP(因为我们不允许他经过director)

8、RS上的lo接口配置VIP的IP地址

缺陷：RS和DS必须在同一机房中

#### TUN



重点：在原有的IP报文外再次封装多一层IP首部，内部IP首部（源地址为CIP，目标IP为VIP），外层IP首部（源地址为DIP，目标IP为RIP）

1、当用户请求到达Director Server，此时请求的数据报文会先到内核空间的PREROUTING链。 此时报文的源IP为CIP，目标IP为VIP 。

2、PREROUTING检查发现数据包的目标IP是本机，将数据包送至INPUT链。

3、IPVS比对数据包请求的服务是否为集群服务，若是，在请求报文的首部再次封装一层IP报文，封装源IP为为DIP，目标IP为RIP。然后发至POSTROUTING链。此时源IP为DIP，目标IP为RIP。

4、POSTROUTING链根据最新封装的IP报文，将数据包发至RS（因为在外层封装多了一层IP首部，所以可以理解为此时通过隧道传输）。 此时源IP为DIP，目标IP为RIP。

5、现是自己的IP地RS接收到报文后发址，就将报文接收下来，拆除掉最外层的IP后，会发现里面还有一层IP首部，而且目标是自己的lo接口VIP，那么此时RS开始处理此请求，处理完成之后，通过lo接口送给eth0网卡，然后向外传递。此时的源IP地址为VIP，目标IP为CIP。

6、响应报文最终送达至客户端。

LVS-Tun模型特性

1、RIP、VIP、DIP全是公网地址

2、RS的网关不会也不可能指向DIP

3、所有的请求报文经由Director Server，但响应报文必须不能经过Director Server

4、不持端口映射

5、RS的系统必须支持隧道

### 调度算法

静态的调度算法

RR、WRR、DH、SH

动态的调度算法。

LC、WLC、SED、NQ

LVS有两种类型的调度算法，其一就是静态的调度算法，这种算法一经实现，后续就不会发生变化，是既定的规则，后续数据包的流转都会按照这种规则进行按部就班的流转；其二就是动态的调度算法，这种算法是基于网络状况，或者后端服务器的状况，连接的状况等来进行实时的调整，算法的规则会根据实际情况而发生一定的变化。

常用的静态调度算法有以下几种：

1.RR：轮叫调度（Round Robin）  
调度器通过”轮叫”调度算法将外部请求按顺序轮流分配到集群中的真实服务器上，它均等地对待每一台服务器，而不管服务器上实际的连接数和系统负载｡

2.WRR：加权轮叫（Weight RR）  
调度器通过“加权轮叫”调度算法根据真实服务器的不同处理能力来调度访问请求。这样可以保证处理能力强的服务器处理更多的访问流量。调度器可以自动问询真实服务器的负载情况,并动态地调整其权值。

3.DH：目标地址散列调度（Destination Hash ）  
根据请求的目标IP地址，作为散列键(HashKey)从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，将请求发送到该服务器，否则返回空。

4.SH：源地址 hash（Source Hash）  
源地址散列”调度算法根据请求的源IP地址，作为散列键(HashKey)从静态分配的散列表找出对应的服务器，若该服务器是可用的且未超载，将请求发送到该服务器，否则返回空｡

常用的动态调度算法有下面这些

1、LC：最少链接（Least Connections）

调度器通过”最少连接”调度算法动态地将网络请求调度到已建立的链接数最少的服务器上。如果集群系统的真实服务器具有相近的系统性能，采用”最小连接”调度算法可以较好地均衡负载。

2、WLC：加权最少连接(默认采用的就是这种)（Weighted Least Connections）

在集群系统中的服务器性能差异较大的情况下，调度器采用“加权最少链接”调度算法优化负载均衡性能，具有较高权值的服务器将承受较大比例的活动连接负载｡调度器可以自动问询真实服务器的负载情况,并动态地调整其权值。

3、SED：最短延迟调度（Shortest Expected Delay ）

在WLC基础上改进，Overhead = （ACTIVE+1）\*256/加权，不再考虑非活动状态，把当前处于活动状态的数目+1来实现，数目最小的，接受下次请求，+1的目的是为了考虑加权的时候，非活动连接过多缺陷：当权限过大的时候，会倒置空闲服务器一直处于无连接状态。

4、NQ永不排队/最少队列调度（Never Queue Scheduling NQ）

无需队列。如果有台 realserver的连接数＝0就直接分配过去，不需要再进行sed运算，保证不会有一个主机很空间。在SED基础上无论+几，第二次一定给下一个，保证不会有一个主机不会很空闲着，不考虑非活动连接，才用NQ，SED要考虑活动状态连接，对于DNS的UDP不需要考虑非活动连接，而httpd的处于保持状态的服务就需要考虑非活动连接给服务器的压力。

## Nginx

**七层（应用层）**：慢于四层，看得懂客户端的请求资源。

## Haproxy

## Keepalive

### 概述

Keepalive是集群管理中保证集群高可用（HA：High Available）的服务软件。

1. 需要心跳机制探测后端RS（Resource Server）是否提供服务
2. 探测down，需要从LVS中删除该RS
3. 探测发送从down到up，需要从LVS中再次添加RS
4. LVS DR，需要主备（HA）

**术语：**

虚拟路由器：Virtual Router

虚拟路由器标识：VRID（0~255），唯一标识虚拟路由器

物理路由器：master：主设备backup

备用设备priority：优先级

VIP：Virtual IP

VMAC：Virtual MAC

### 原理

vrrp协议：Virtual Router Redundancy Protocol

IP漂移

## Heartbeat

Keepalived使用的vrrp协议方式，虚拟路由冗余协议 (Virtual Router Redundancy Protocol，简称VRRP)；Heartbeat是基于主机或网络的服务的高可用方式。

keepalived的目的是模拟路由器的双机，heartbeat的目的是用户service的双机。

lvs的高可用建议用keepavlived，业务的高可用建议heartbeat。

**说明：**

1）Keepalived使用更简单：从安装、配置、使用、维护等角度上对比，Keepalived都比Heartbeat要简单得多，尤其是Heartbeat2.1.4后拆分成3个子项目，安装、配置、使用都比较复杂，尤其是出问题的时候，都不知道具体是哪个子系统出问题了；而Keepalived只有1个安装文件、1个配置文件，配置文件也简单很多；

2）**Heartbeat功能更强大**：Heartbeat虽然复杂，但功能更强大，配套工具更全，适合做大型集群管理，而Keepalived主要用于集群倒换，基本没有管理功能；

3）协议不同：Keepalived使用VRRP协议进行通信和选举，Heartbeat使用心跳进行通信和选举；Heartbeat除了走网络外，还可以通过串口通信，貌似更可靠；

4）使用方式基本类似：如果要基于两者设计高可用方案，最终都要根据业务需要写自定义的脚本，Keepalived的脚本没有任何约束，随便怎么写都可以；Heartbeat的脚本有约束，即要支持service start/stop/restart这种方式，而且Heartbeart提供了很多默认脚本，简单的绑定ip，启动apache等操作都已经有了。

使用建议：**优先使用Keepalived，当Keepalived不够用的时候才选择Heartbeat**。

## Zookeeper

## Galera Cluster

## 常用组合

heartbeat v2+haresource(或crm) (说明：一般常用于CentOS 5.X)

heartbeat v3+pacemaker (说明：一般常用于CentOS 6.X)

corosync+pacemaker (说明：现在最常用的组合)

cman + rgmanager (说明：红帽集群套件中的组件，还包括gfs2,clvm)

keepalived+lvs (说明：常用于lvs的高可用)

# 高可用方案

## 客户端实现的高可用方案

以memcache 为例，客户端同时与好几个服务保持连接，按照一定的规则去调用服务，当服务挂掉之后，重新调整规则。当然，如果服务器不做主从备份的话，可能会造成部分数据丢失。

## 服务之间通信实现高可用

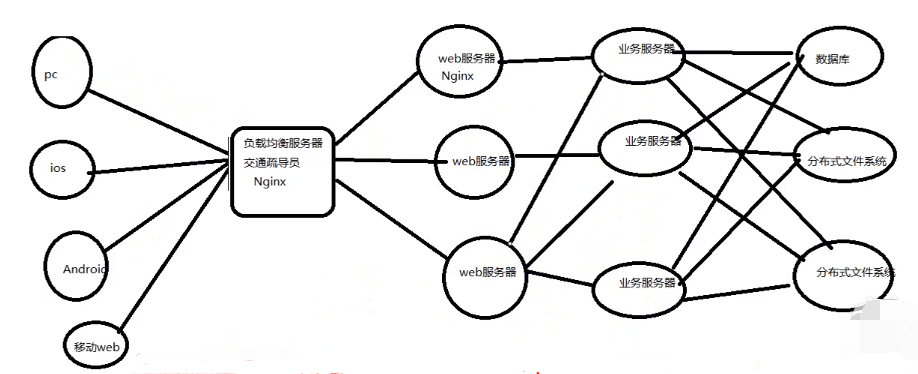
这种经典的案例就是redis了，各个redis之间保持通信，当主服务挂掉之后从服务就会升为主服务。对于客户端来说几乎是透明的。

## 通过中间件实现高可用

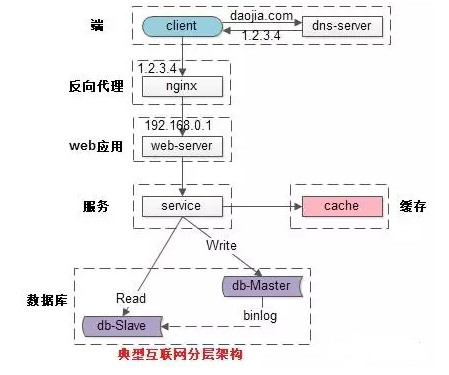
mycat实现mysql高可用的中间价。

早期版本redis不支持集群，那时候redis的高可用也是基于中间件来做的。

# 互联网分层



典型互联网架构中，通过冗余+自动故障转移来保证系统的高可用特性。



常见互联网分布式架构如上，分为：

（1）客户端层：典型调用方是浏览器browser或者手机应用APP

（2）反向代理层：系统入口，反向代理

（3）站点应用层：实现核心应用逻辑，返回html或者json

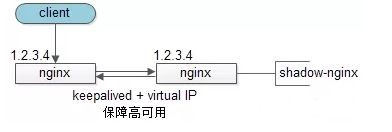
（4）服务层：如果实现了服务化，就有这一层

（5）数据-缓存层：缓存加速访问存储

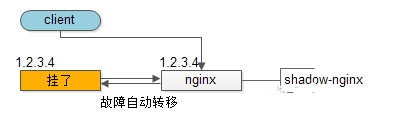
（6）数据-数据库层：数据库固化数据存储

整个系统的高可用，又是通过每一层的冗余+自动故障转移来综合实现的。

## 客户端层->反向代理层

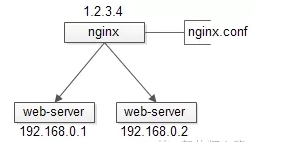


客户端层到反向代理层的高可用，是通过反向代理层的冗余来实现的。以nginx为例：有两台nginx，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP（浮动IP）提供服务。

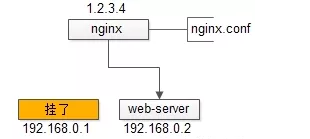


自动故障转移：当nginx挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-nginx，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

## 反向代理层->站点层

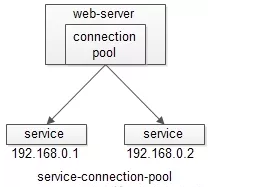


反向代理层到站点层的高可用，是通过站点层的冗余来实现的。假设反向代理层是nginx，nginx.conf里能够配置多个web后端，并且nginx能够探测到多个后端的存活性。

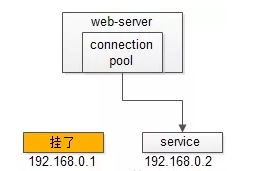


自动故障转移：当web-server挂了的时候，nginx能够探测到，会自动的进行故障转移，将流量自动迁移到其他的web-server，整个过程由nginx自动完成，对调用方是透明的。

## 站点层->服务层

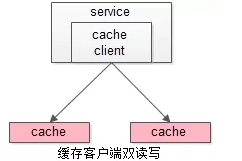


站点层到服务层的高可用，是通过服务层的冗余来实现的。“服务连接池”会建立与下游服务多个连接，每次请求会“随机”选取连接来访问下游服务。



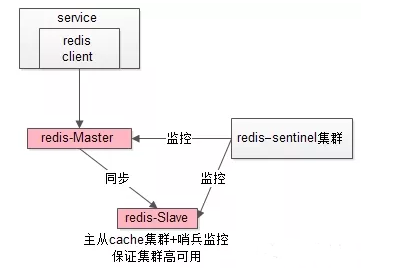
自动故障转移：当service挂了的时候，service-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的service，整个过程由连接池自动完成，对调用方是透明的（所以说RPC-client中的服务连接池是很重要的基础组件）。

## 服务层>缓存层



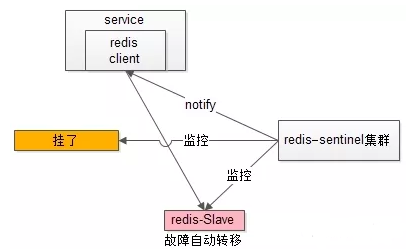
服务层到缓存层的高可用，是通过缓存数据的冗余来实现的。

缓存层的数据冗余又有几种方式：第一种是利用客户端的封装，service对cache进行双读或者双写。



缓存层也可以通过支持主从同步的缓存集群来解决缓存层的高可用问题。

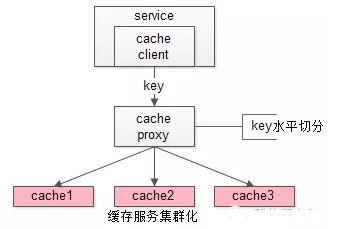
以redis为例，redis天然支持主从同步，redis官方也有sentinel哨兵机制，来做redis的存活性检测。



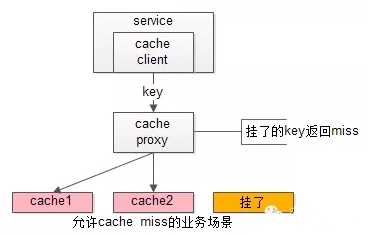
自动故障转移：当redis主挂了的时候，sentinel能够探测到，会通知调用方访问新的redis，整个过程由sentinel和redis集群配合完成，对调用方是透明的。

注意：业务对缓存并不一定有“高可用”要求，更多的对缓存的使用场景，是用来“加速数据访问”：把一部分数据放到缓存里，如果缓存挂了或者缓存没有命中，是可以去后端的数据库中再取数据的。

这类允许“cache miss”的业务场景，缓存架构的建议是：



将kv缓存封装成服务集群，上游设置一个代理（代理可以用集群冗余的方式保证高可用），代理的后端根据缓存访问的key水平切分成若干个实例，每个实例的访问并不做高可用。

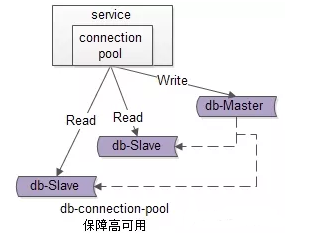


缓存实例挂了屏蔽：当有水平切分的实例挂掉时，代理层直接返回cache miss，此时缓存挂掉对调用方也是透明的。key水平切分实例减少，不建议做re-hash，这样容易引发缓存数据的不一致。

## 服务层>数据库层

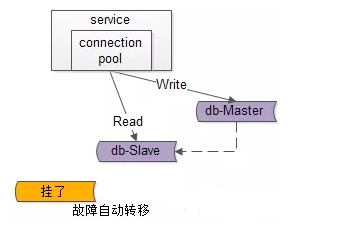
大部分互联网技术，数据库层都用了“主从同步，读写分离”架构，所以数据库层的高可用，又分为“读库高可用”与“写库高可用”两类。

## 服务层>数据库层“读”



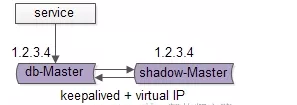
服务层到数据库读的高可用，是通过读库的冗余来实现的。

既然冗余了读库，一般来说就至少有2个从库，“数据库连接池”会建立与读库多个连接，每次请求会路由到这些读库。



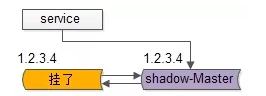
自动故障转移：当读库挂了的时候，db-connection-pool能够探测到，会自动的进行故障转移，将流量自动迁移到其他的读库，整个过程由连接池自动完成，对调用方是透明的（所以说DAO中的数据库连接池是很重要的基础组件）。

## 服务层>数据库层“写”



服务层到数据库写的高可用，是通过写库的冗余来实现的。

以mysql为例，可以设置两个mysql双主同步，一台对线上提供服务，另一台冗余以保证高可用，常见的实践是keepalived存活探测，相同virtual IP提供服务。



自动故障转移：当写库挂了的时候，keepalived能够探测到，会自动的进行故障转移，将流量自动迁移到shadow-db-master，由于使用的是相同的virtual IP，这个切换过程对调用方是透明的。

## 总结

高可用HA（High Availability）是分布式系统架构设计中必须考虑的因素之一，它通常是指，通过设计减少系统不能提供服务的时间。

方法论上，高可用是通过冗余+自动故障转移来实现的。

整个互联网分层系统架构的高可用，又是通过每一层的冗余+自动故障转移来综合实现的，具体的：

（1）【客户端层】到【反向代理层】的高可用，是通过反向代理层的冗余实现的，常见实践是keepalived + virtual IP自动故障转移

（2）【反向代理层】到【站点层】的高可用，是通过站点层的冗余实现的，常见实践是nginx与web-server之间的存活性探测与自动故障转移

（3）【站点层】到【服务层】的高可用，是通过服务层的冗余实现的，常见实践是通过service-connection-pool来保证自动故障转移

（4）【服务层】到【缓存层】的高可用，是通过缓存数据的冗余实现的，常见实践是缓存客户端双读双写，或者利用缓存集群的主从数据同步与sentinel保活与自动故障转移；更多的业务场景，对缓存没有高可用要求，可以使用缓存服务化来对调用方屏蔽底层复杂性

（5）【服务层】到【数据库“读”】的高可用，是通过读库的冗余实现的，常见实践是通过db-connection-pool来保证自动故障转移

（6）【服务层】到【数据库“写”】的高可用，是通过写库的冗余实现的，常见实践是keepalived + virtual IP自动故障转移