分布式系统：

高可用性:

故障发现和处理以及恢复 容错处理。在一个正常运作系统中存在一个时间比例的条件。如果一个用户不能访问系统比例增大，它被认为是不可用。可用性公式：Availability = uptime / (uptime + downtime)容错 failover 是指一个系统在错误发生的情况下，仍然一切运行正常。表示这个系统是宽容错误的。

高吞吐量：

异步化方式（SEDA）：针对高并发的分布式系统，NIO 方式很早提出来就是针对这个问题的，包括两部分， 一部分是后端系统之间的异步，典型的是 Netty、Mina、xsocket 等成熟的框架，另外一部分是前端接入 http 的异步 servlet(都有相应的规范)， 在 Http1.1 中支持，对客户端 LCP 来讲是同步调用的的。典型是有 Jetty6、jetty7、tomcat6、tomcat7 以及其他商业服务器都有实现。基本思想都是采用非阻塞方式，接受线程和工作线程分离，提高整 个系统的吞吐量。好处是，整个系统的可用性大大增加，系统之间耦合度大大降低，并且有比较高的吞吐量，各个系统能高速运转，不会相互等待资源，可以充分利用资源。同时整个系统也可以有很好的伸缩性。 弊端是对 CPU 的要求比较高的。

数据存储：

对于数据的高并发的访问，传统的关系型数据库提供读写分离的方案，但是带来的确实数据的一致性问题提供的数据切分的方案；对于越来越多的海量数据，传统的数据库采用的是分库分表，实现起来比较复杂，后期要不断的进行迁移维护；对于高可用和伸缩方面，传统数据采用的是主备、主从、多主的方案，但是本身扩展性比较差，增加节点和宕机需要进行数据的迁移。对于以上提出的这些问题，分布式数据库 HBase 有一套完善的解决方案，适用于高并发海量数据存取的要求。

HBase 基于列式的高效存储降低 IO，通常的查询不需要一行的全部字段，大多数只需要几个字段，对与面向行的存储系统，每次查询都会全部数据取出，然后再从中选出需要的字段，面向列的存储系统可以单独查询某一列，从而大大降低 IO，提高压缩效率，同列数据具有很高的相似性，会增加压缩效率。

一致性：

HBase 的一致性数据访问是通过 MVCC 来实现的。HBase 在写数据的过程中，需要经过好几个阶段，写 HLog，写 memstore，更新 MVCC;只有更新了 MVCC，才算真正 memstore 写成功，其中事务的隔离需要有 mvcc 的来控制，比如读数据不可以获取别的线程还未提交的数据。

HDFS 为分布式存储引擎，一备三，高可靠，0 数据丢失。HDFS 的 namenode 是一个 SPOF。

为避免单个 region 访问过于频繁，单机压力过大，提供了 split 机制，HBase 的写入是 LSM-TREE 的架构方式，随着数据的 append，HFile 越来越多，HBase 提供了 HFile 文件进行 compact，对过期数据进行清除，提高查询的性能。

缓存：

在一些高并发高性能的场景中，使用 cache 可以减少对后端系统的负载，承担可大部分读的压力，可以大大提高系统的吞吐量，比如通常在数据库存储之前增加 cache 缓存。但是引入 cache 架构不可避免的带来一些问题，cache 命中率的问题, cache 失效引起的抖动，cache 和存储的一致性。Cache 中的数据相对于存储来讲，毕竟是有限的，比较理想的情况是存储系统的热点数据，这里可以用一些常见的算法 LRU 等等淘汰老的数据；随着系统规模的增加，单个节点 cache 不能满足要求，就需要搭建分布式 Cache；为了解决单个节点失效引起的抖动 ，分布式 cache 一般采用一致性 hash 的解决方案，大大减少因单个节点失效引起的抖动范围；而对于可用性要求比较高的场景，每个节点都是需要有备份的。数据在 cache 和存储上都存有同一份备份，必然有一致性的问题，一致性比较强的，在更新数据库的同时，更新数据库 cache。对于一致性要求不高的，可以去设置缓存失效时间的策略。Memcached 作为高速的分布式缓存服务器，协议比较简单，基于 libevent 的事件处理机制。Cache 系统在平台中用在 router 系统的客户端中，热点的数据会缓存在客户端，当数据访问失效时，才去访问 router 系统。当然目前更多的利用内存型的数据库做 cache，比如 redis、mongodb；redis 比 memcache 有丰富的数据操作的 API；redis 和 mongodb 都对数据进行了持久化，而 memcache 没有这个功能，因此 memcache 更加适合在关系型数据库之上的数据的缓存。

分区：

对于源数据的切分，如果是文件可以根据文件名称设置块大小来切分。对于关系型数据库，由于一般的需求是只离线同步一段时间的数据(比如凌晨把当天的订单数据同步到 HBase)，所以需要在数据切分时(按照行数切分)，会多线程扫描整个表(及时建索引，也要回表)，对于表中包含大量的数据来讲，IO 很高，效率非常低；这里解决的方法是对数据库按照时间字段(按照时间同步的)建立分区，每次按照分区进行导出。

负载均衡：

随着平台并发量的增大，需要扩容节点进行集群，利用负载均衡设备进行请求的分发；负载均衡设备通常在提供负载均衡的同时，也提供失效检测功能；同时为了提高可用性，需要有容灾备份，以防止节点宕机失效带来的不可用问题；备份有在线的和离线备份，可以根据失效性要求的不同，进行选择不同的备份策略。

一个大型的平台包括很多个业务域，不同的业务域有不同的集群，可以用 DNS 做域名解析的分发或轮询，DNS 方式实现简单，但是因存在 cache 而缺乏灵活性；一般基于商用的硬

件 F5、NetScaler 或者开源的软负载 lvs 在 4 层做分发，当然会采用做冗余(比如 lvs+keepalived)的考虑，采取主备方式。4 层分发到业务集群上后，会经过 web 服务器如 nginx 或者 HAProxy 在 7 层做负载均衡或者反向代理分发到集群中的应用节点。选择哪种负载，需要综合考虑各种因素（是否满足高并发高性能，Session 保持如何解决，负载均衡的算法如何，支持压缩，缓存的内存消耗）；下面基于几种常用的负载均衡软件做个介绍。

LVS，工作在 4 层，Linux 实现的高性能高并发、可伸缩性、可靠的的负载均衡器，支持多种转发方式(NAT、DR、IP Tunneling)，其中 DR 模式支持通过广域网进行负载均衡。支持双机热备(Keepalived 或者 Heartbeat)。对网络环境的依赖性比较高。

Nginx 工作在 7 层，事件驱动的、异步非阻塞的架构、支持多进程的高并发的负载均衡器/反向代理软件。可以针对域名、目录结构、正则规则针对 http 做一些分流。通过端口检测到服务器内部的故障，比如根据服务器处理网页返回的状态码、超时等等，并且会把返回错误的请求重新提交到另一个节点，不过其中缺点就是不支持 url 来检测。对于 session sticky，可以基于 ip hash 的算法来实现，通过基于 cookie 的扩展 nginx-sticky-module 支持 session sticky。

HAProxy 支持 4 层和 7 层做负载均衡，支持 session 的会话保持，cookie 的引导；支持后端 url 方式的检测；负载均衡的算法比较丰富，有 RR、权重等。对于图片，需要有单独的域名，独立或者分布式的图片服务器或者如 mogileFS，可以图片服务器之上加 varnish 做图片缓存。

系统监控：

监控也是提高整个平台可用性的一个重要手段，多平台进行多个维度的监控；模块在运行时候是透明的，以达到运行期白盒化。

大型分布式系统涉及各种设备，比如网络交换机，普通 PC 机，各种型号的网卡，硬盘，内存等等，还有应用业务层次的监控，数量非常多的时候，出现错误的概率也会变大，并且有些监控的时效性要求比较高，有些达到秒级别；在大量的数据流中需要过滤异常的数据，有时候也对数据会进行上下文相关的复杂计算，进而决定是否需要告警。因此监控平台的性能、吞吐量、已经可用性就比较重要，需要规划统一的一体化的监控平台对系统进行各个层次的监控。

通讯高效可靠：

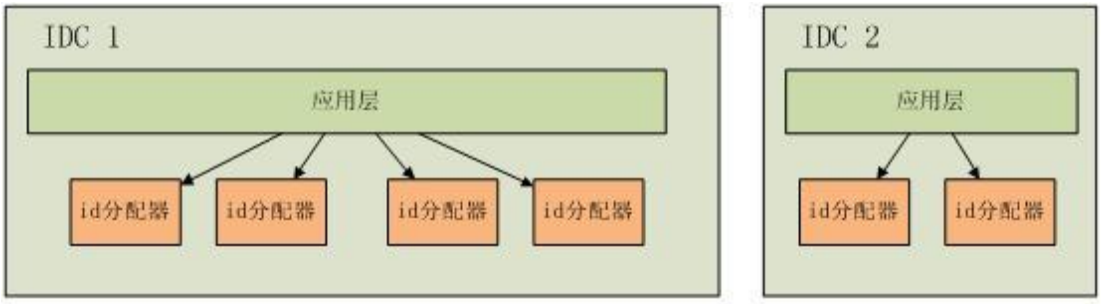
通信组件用于业务系统内部服务之间的调用，在大并发的电商平台中，需要满足高并发高吞吐量的要求。整个通信组件包括客户端和服务端两部分。

客户端和服务器端维护的是长连接，可以减少每次请求建立连接的开销，在客户端对于每个服务器定义一个连接池，初始化连接后，可以并发连接服务端进行 rpc 操作，连接池中的长连接需要心跳维护，设置请求超时时间。

对于长连接的维护过程可以分两个阶段，一个是发送请求过程，另外一个是接收响应过程。在发送请求过程中，若发生 IOException，则把该连接标记失效。接收响应时，服务端返回 SocketTimeoutException，如果设置了超时时间，那么就直接返回异常，清除当前连接中那些超时的请求。否则继续发送心跳包(因为可能是丢包，超过 pingInterval 间隔时间就发送 ping 操作)，若 ping 不通(发送 IOException)，则说明当前连接是有问题的，那么就把当前连接标记成已经失效；若 ping 通，则说明当前连接是可靠的，继续进行读操作。失效的连接会从连接池中清除掉。每个连接对于接收响应来说都以单独的线程运行，客户端可以通过同步(wait,notify)方式或者异步进行 rpc 调用，序列化采用更高效的 hession 序列化方式。服务端采用事件驱动的 NIO 的 MINA 框架，支撑高并发高吞吐量的请求。

ID 分配：

将整个 id 空间按取模或分段等分为若干个独立的 id 子空间，每个 id 子空间由一个独立的分配器负责。



优点：简单，各个 id 分配器无需协作，即使发生网络划分时，也可保证可用性和 id 的不冲突。

如果在国际化环境的多 IDC 里进行部署，需要预先将 id 空间划分为 N 份，每个国家里部署若干份。每个 IDC 内应用只连本 IDC 的 id 分配服务。

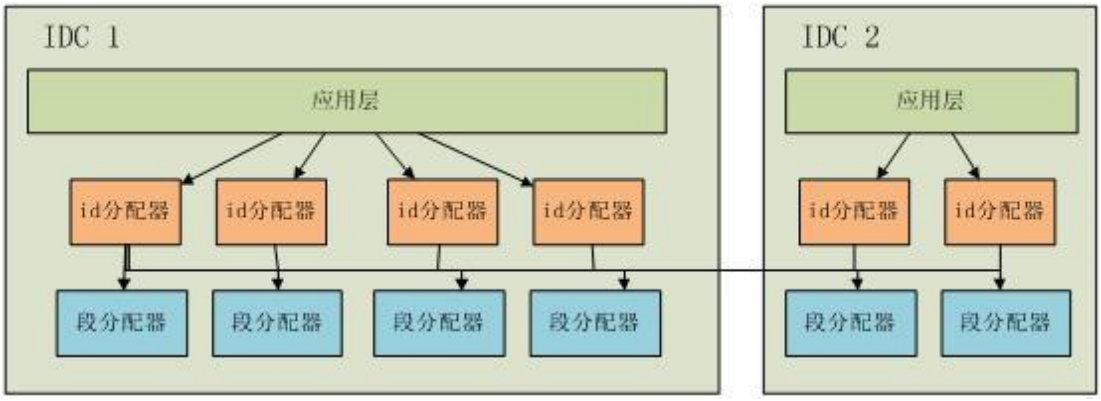
在均衡性上的不足：在同一个 IDC 内，均衡性可以在接入层均衡算法保证，但是在多个 IDC 里，ID 分配器个数的比例和 id 增长的服务往往是不吻合的，因此在多个 IDC 内，id 是无法保证均衡增长的。

均衡性上的改进：

将 id 分配分为两层：

1.上层的“id 分配器”对应用暴露，提供一次申请一个 id 的接口，一般本 IDC 的应用只连本 IDC 的 id 分配器。

2.下层的“段分配器”对“id 分配器”提供服务。id 分配器“知晓”所有 IDC 的所有段分配器的存在，使用均衡策略向段分配器申请一个 id 段，当所持有的 id 段快耗尽时，再请求下一个段。



唯一性：全局中，根据分片规则，每个段分配器会持有不同的 id 段。例如下表中，每个段的大小是 100，段分配器 A 持有分片 0 和分片 1。对于每个分片而言，是一个个跳跃的 id 段。特殊的，当段大小为 1 时，段分配器就是改进前的 id 分配器。

模块	section	段分配器 A		段分配器 B
分片		0	1	15
持有的 id 段	0	000-099	100-199	1500-1599
	1	1600-1699	1700-1799	3100-3199
	2	3200-3299	...	...
	...	...	...	...
	10	16000-16099	16100-16199	...
	...	...	...	...

均衡策略：均衡策略在 id 分配器来实现，简单的讲，是一个轮询策略。每个 id 分配器会轮询下游段分配器的状态，并选中 id 段的最小的那个，然后发起 id 段申请。由于不会加锁，当多个 id 分配器同时竞争时，可能会出现获取的 id 段不是全局最小的，可以附加一些策略来调优，比如再多获取一次，并本地排序。从整体上而言，id 还是比较均衡的，可满足需求。

可用性：当发生网络划分时，本 IDC 的 id 分配器可以只连接本 IDC 的段分配器，成功的申请到 id 段。整个系统可容忍一定时间内不可协作，长时间不可协作的唯一危害是 id 增长不均衡，此时，就退化为改进前的方案。

多 IDC 环境的适应性：id 分配器需要和所有 IDC 的段分配器交互，但是交互频率很低，同时和提供 id 分配服务是两个独立的阶段，不会受到多 IDC 网络环境的干扰。

通讯可靠高效：

在分布式计算环境中，保证计算节点之间数据通信的可靠性是计算机集群需要考察的一个最基本的能力。目前最通用的网络通信协议主要是 TCP 协议和 UDP 协议。在多台计算机协同工作的环境中，采用 TCP 协议可以很方便的达到消息可靠传输的目的，但是 TCP 协议要求计算机在网络中创建点到点之间有状态的 SOCKET 资源，如果需要将同一条消息发送给集群中的每一个节点，采用 TCP 协议将比采用 UDP 协议需要更多的网络流量，这在大规模的网络实时通信中对应用程序而言是一笔不小的开。UDP 协议对网络资源的消耗代价很低，但 UDP 协议本身并不能保证通信过程中消息可以可靠的从消息发送者传递到消息接收者手中，所以如果要在集群环境中使用 UDP 协议达到可靠的消息传输的目的，需要在 UDP 协议之上增加新的协议进行控制。

在集群环境下使用 UDP 协议进行通信可以有效的降低集群网络环境的数据传输压力，JGroups 通过 UNICAST 协议来保证消息的可靠传输，UNICAST 协议在 UDP 的基础上增加了消息响应的步骤，以保证传输过程的完整性。原始的 UDP 协议不要求消息的接收者反馈给消息的发送者是否已经成功接收到消息，UNICAST 在此基础上完善了消息反馈的过程。消息的反馈方式分为 ACK 和 NAKACK 两种：

ACK：消息的发送者不断的重复发送消息，直到所有的接收者都返回了确认消息已经收到，

NAKACK：消息的接收者不断请求消息发送者发送消息，直到消息的接收者确认所有收到的消息是完整的。

UNICAST 协议需要保证消息的传输过程是可靠的，它依赖于一个不断发送消息的时间周期来进行循环，因此 UNICAST 的一项重要的配置属性 timeout 就来源于此：如果 timeout 值被设置为 100，200，400，800，就表示如果消息发送者在等待 100 毫秒还没有接收到消息接收者的 ACK 消息，则消息发送者重新发送消息（第一次重发），消息发送者继续等待 200 毫秒仍然没有接收到 ACK 消息，则消息发送者再次重新发送消息（第二次重发），这样直到等待 800 毫秒进行第四次重发。在多播环境下，NAKACK 协议基于 ACK 协议进行了扩展，在这种协议下，每个消息绑定一个序列号，消息接收者根据序列号确保消息按正确的顺序传递。如果接收者发现了一个序列号的缺失，接收者安排一个周期性的任务去要求发送者重新发送该序列号的消息，当缺失的序列号的消息收到，则请求数据同步的任务取消，并向消息的发送者提供消息完全抵达确认反馈。

消息队列：

消息队列能够提供以下几个方面的帮助：

1，保证消息的传递；

如果发送消息时接收者不可用，消息队列会保留消息，直到成功地传递它；

2，提供异步的通信协议；

消息的发送者将消息发送到消息队列后可以立即返回，不用等待接收者的响应，消息会被保存在队列中，直到接收者取出它；

3，解耦；

只要消息格式不变，即使接收者的接口、位置、或者配置改变，也不会给发送者带来任何改变；

而且，消息发送者无需知道消息接收者是谁，使得系统设计更清晰；

相反的，例如，远程过程调用（RPC）或者服务间通过 socket 建立连接，如果对方接口改变了或者对方 ip、端口改变了，那么另一方需要改写代码或者改写配置；

4，提供路由；

发送者无需与接收者建立连接，双方通过消息队列保证消息能够从发送者路由到接收者，甚至对于本来相互网络不通的两个服务，也可以提供消息路由。

细节方面的关注：

1，支持并发模式和顺序模式

顺序模式：

消息接收后，在消息接收者来主动删除这条消息之前，队列中的其他消息不可被接收。这样保证应用在一个任务完成后再处理下一个任务，提供了强顺序性

并发模式：

消息被接收后，消息接收者主动删除前，队列的下一个消息仍可被其他应用接收，且可以一直并发获取下一个没有被接收的消息。

如果消息被接收后持久（该时间可配）不被删除，则认为消息的原接收者处理失败，消息重新可见，

可被其他应用再次接收（此特性为容错处理，“接收顺序队列消息”的功能也具备该特性）。

2，短暂锁模式：

一个消息不会同时被多个服务接收，这是通过针对消息的短暂锁来保证的，消息的接收者可以指定消息被锁定的时间，

如果接收者处理完消息需要主动将消息显示删除，如果接收者处理消息失败了，那么另一个服务可以在这个消息的锁失效后重新获得这个消息。