分布式相关：

好的设计应该是分布式和集群的结合，先分布式再集群，具体实现就是业务拆分成很多子业务，然后针对每个子业务进行集群部署，这样每个子业务如果出了问题，整个系统完全不会受影响。

分布式理论

CAP 也就是 Consistency（一致性）、Availability（可用性）、Partition Tolerance（分区容错性）这三个单词首字母组合。

BASE 是 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。BASE 理论是对 CAP 中一致性和可用性权衡的结果，其来源于对大规模互联网系统分布式实践的总结，是基于 CAP 定理逐步演化而来的，它大大降低了我们对系统的要求。

相关阅读：CAP 理论和 BASE 理论解读

分布式相关的中间件

1、消息中间件：

消息队列（rabbitMQ,实现异步通讯，ActiveMQ、Rabbit MQ、RocketMQ和Kafk a**）**

(https://blog.csdn.net/luckykapok918/article/details/72528717)

消息中间件的作用：

1、消息中间件主要解决分布式系统之间信息的传递

2、低耦合，不管是程序还是模块之间，使用信息中间件进行间接通信。

3、异步通信能力，使得子系统之间得以充分执行之间的逻辑而无需等待。

4、缓冲能力：消息中间件像是一个巨大的蓄水池，将高峰期大量的请求存储下来，慢慢交给后台进行处理，对于秒杀业务来说尤为重要。

2、缓存中间件：redis。mechache

3、数据库中间件：

- 分布式数据库分表分库（Cobar, MyCAT, TDDL, DRDS, DDB）
- 数据增量订阅与消费（Canal, Erosa）
- 数据库同步（全量、增量、跨机房、复制）（Otter, JingoBus, DRC）
- 跨数据库（数据源）迁移（yugong, DataX）

分布式测试的难点：（https://blog.csdn.net/huazhongkejidaxuezpp/article/details/88992547）

业务方面测试

功能测试

重复提交后的幂等处理；

故障模拟和恢复测试

分布式系统的目标：**任何局部的进程异常都不会影响系统的可用性。**例如：测试时候要关注单节点的故障，是否能够快速侦探到，并且快速隔离故障节点。避免节点故障引发数据库或者队列的堵塞，引起大面积业务异常。

故障模拟操作比如机器宕机，重启，断网，主要模块进程重启，假死、部署中等，这些操作都会按照预先设定比例进行随机组合，并结合分布式系统的目标，进行测试。

具体实施上，可以根据分布式系统本身的故障应对目标而定。 例如，曾经测试过一个Akka的分布式服务，项目本身是迁移到了Akka上。 测试方案制定的时候，除了本身功能、性能、集成等通过后，特意对Akka的故障恢复能力做了测试：

1）目标一:一个组内的一台服务器挂掉，自动切换到该组的其他机器上。

2）目标二:一个组内的全部服务器挂掉，自动切换到另外的备用组。

参考：http://www.51testing.com/html/08/15184608-3725886.html

任务调度，负载均衡策略

思考方向：

每个节点处理的任务数不均匀，那就会导致拖尾现象，系统整体的处理性能不高；

在设计调度规则的时候，考虑默认处理节点，及“兜底”处理；

长时间稳定性测试

这里可以结合压力测试、故障模拟一起进行，评估各个主要模块的读写压力，还有一些诸如CPU，Memory，Network的资源消耗器，以模拟机器资源紧张场景。

异常测试

异常测试按性质分为应用层的业务逻辑异常测试、系统硬件/网络/文件/数据库/缓存/中间件异常测试。

业务逻辑异常测试体现在当上述的第二种异常发生时，是否能根据业务的需要或者架构的设计做出合理的业务处理反应，这是建立在第二种异常测试之上的，因此异常测试的关系也已经非常明确了：

第一种测试根据业务的不同，范围和流程有不确定性

第二种测试则是在一些明确的规则和约定下进行。

异常测试中最重要的一些场景：

系统资源：cpu、内存使用率过高，能否能将请求切到到资源利用率低的服务器上；

数据库中间件(mybatis,dabus)：锁及慢SQL分析，

缓存(redis)：效率提升、雪崩、击穿等

消息中间件(mafka)：消息加压/过期/重试等处理；

服务发现：服务不可用：是否有其他处理措施；单台不可用：是否能重新选举，重新建立连接；

ESSearch：存储及效率提升

定时调度：数据量多少

分布式的优缺点：

跟单体结构的对比

	传统单体架构	分布式服务化架构
新功能开发	需要时间	容易开发和实现
部署	不经常且容易部署	经常发布，部署复杂
隔离性	故障影响范围大	故障影响范围小
架构设计	难度小	难度级数增加
系统性能	响应时间快，吞吐量小	响应时间慢，吞吐量大
系统运维	运维简单	运维复杂
新人上手	学习曲线大（应用逻辑）	学习曲线大（架构逻辑）
技术	技术单一且封闭	技术多样且开放
测试和查错	简单	复杂
系统扩展性	扩展性很差	扩展性很好
系统管理	重点在于开发成本	重点在于服务治理和调度

缺点：

2、分布式缺点

1、架构设计变得复杂（尤其是其中的分布式事务）

2、部署单个服务会比较快，但是如果一次部署需要多个服务，部署会变得复杂

3、系统的吞吐量会变大，但是响应时间会变长

4、运维复杂度会因为服务变多而变得很复杂

5、架构复杂导致学习曲线变大

6、测试和查错的复杂度增大

7、技术可以很多样，这会带来维护和运维的复杂度
8、管理分布式系统中的服务和调度变得困难和复杂

微服务、集群、分布式三者的关系和区别:

微服务: 微服务是一种架构风格,可以说是一种处理问题的思想,通过这种思想可以将原来一个复杂的系统拆分成多个子系统,多个子系统之间是相互独立的,有自己独立的进程,可以单独部署,每个子系统(微服务)都只关注实现自己的业务功能,这样子就解决了原来所有业务都存放在一个系统上，让系统显得很臃肿，并且难以抽离的问题。

集群: 简单的说是将同一个功能的项目(系统)部署到不同的服务器中,提高了系统的高可用性,就好像我们以前在学校机房里面的备用发电机一样,如果正常的发电机无法使用了,就可以使用备份的，这样你们就不能停止学习了0_0。

分布式: 通俗的将,**是将一个大系统中拆分出来的子系统分别部署到不同的服务器上**,与集群不同之处在于,集群是将同一个业务的项目部署在不同的服务器上,是物理层面的，保障的是系统的高可用性，而分布式是一种工作方式,目的在与将不同业务的子项目部署到不同的服务器上,并不能保障各个项目的高可用性，但是可以对分布式中每个节点做集群处理，从而实现高可用的目的。
微服务是将一个复杂系统根据业务或者其他方式拆分成具有不同功能的子系统的思想或者说架构风格,可以说是实现集群或者分布式的前提，毕竟，没有系统的话集群和分布式也就没有作用的对象了。
集群则是将按照微服务架构风格拆分出来的同一个业务系统部署到不同的服务器上,这样一个系统就存在多份，保障了系统的高可用性。
分布式则是按照微服务架构风格拆分出来的不同业务的系统系统部署到不同的服务器上，可以对分布式上的每一个节点进行集群设置，实现高可用，它是一种工作的方式。
简单说，分布式是以缩短单个任务的执行时间来提升效率的,如一个任务由5个小任务组成，处理完一个任务需要1个小时，那么使用一台服务器的话则需要五个小时，如果使用五台服务器则只需要一个小时，这种也是Hadpood中的一种Map/Reduce 分布式计算模型的工作模式。)

负载均衡（可由dubbo提供，引入一个负载均衡器）

我们的系统中的某个服务的访问量特别大，我们将这个服务部署在了多台服务器上，当客户端发起请求的时候，多台服务器都可以处理这个请求。那么，如何正确选择处理该请求的服务器就很关键。假如，你就要一台服务器来处理该服务的请求，那该服务部署在多台服务器的意义就不复存在了。负载均衡就是为了避免单个服务器响应同一请求，容易造成服务器宕机、崩溃等问题，我们从负载均衡的这四个字就能明显感受到它的意义。

负载均衡算法

负载均衡算法决定了后端的哪些健康服务器会被选中。几个常用的算法：

- Round Robin（轮询）：为第一个请求选择列表中的第一个服务器，然后按顺序向下移动列表直到结尾，然后循环。
- Least Connections（最小连接）：优先选择连接数最少的服务器，在普遍会话较长的情况下推荐使用。
- Source：根据请求源的 IP 的散列（hash）来选择要转发的服务器。这种方式可以一定程度上保证特定用户能连接到相同的服务器。

分布式协调

ZooKeeper 是一个开源的分布式协调服务，它的设计目标是将那些复杂且容易出错的分布式一致性服务封装起来，构成一个高效可靠的原语集，并以一系列简单易用的接口提供给用户使用
ZooKeeper 为我们提供了高可用、高性能、稳定的分布式数据一致性解决方案，通常被用于实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

微服务背景下，一个系统被拆分为多个服务，但是像安全认证，流量控制，日志，监控等功能是每个服务都需要的，没有网关的话，我们就需要在每个服务中单独实现，这使得我们做了很多重复的事情并且没有一个全局的视图来统一管理这些功能。
综上：一般情况下，网关都会提供请求转发、安全认证（身份/权限认证）、流量控制、负载均衡、容灾、日志、监控这些功能。
上面介绍了这么多功能，实际上，网关主要做了一件事情：请求过滤 。

高性能

消息队列

我们可以把消息队列看作是一个存放消息的容器，当我们需要使用消息的时候，直接从容器中取出消息供自己使用即可。

数据层的扩展

数据层的扩展包括缓存使用、DB读写分离、分库分表等操作。另外对于NoSQL的使用也是减小DB压力的方式，例如针对搜索业务建立单独的搜索引擎

数据库的高性能和高可用

读写分离&分库分表

读写分离作用：（解决并发，高性能）

1、**读写分离主要是为了将数据库的读和写操作分不到不同的数据库节点上。主服务器负责写，从服务器负责读。**另外，一主一从或者一主多从都可以。读写分离可以大幅提高读性能，小幅提高写的性能。因此，读写分离更适合单机并发读请求比较多的场景。

何为分库？（高可用）

分库 就是将数据库中的数据分散到不同的数据库上。

下面这些操作都涉及到了分库：

- 你将数据库中的用户表和用户订单表分别放在两个不同的数据库。
- 由于用户表数据量太大，你对用户表进行了水平切分，然后将切分后的 2 张用户表分别放在两个不同的数据库。

何为分表？

分表 就是对单表的数据进行拆分，可以是垂直拆分，也可以是水平拆分。

何为垂直拆分？

简单来说，垂直拆分是对数据表列的拆分，把一张列比较多的表拆分为多张表。

举个例子：**我们可以将用户信息表中的一些列单独抽出来作为一个表。**

何为水平拆分？

简单来说，水平拆分是对数据表行的拆分，把一张行比较多的表拆分为多张表。

举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。

什么情况下需要分库分表？

遇到下面几种场景可以考虑分库分表：

- 单表的数据达到千万级别以上，数据库读写速度比较缓慢（分表）。
- 数据库中的数据占用的空间越来越大，备份时间越来越长（分库）。
- 应用的并发量太大（分库）。

分库分表会带来什么问题呢？

- join 操作 ： 同一个数据库中的表分布在了不同的数据库中，导致无法使用 join 操作。这样就导致我们需要手动进行数据的封装，比如你在一个数据库中查询到一个数据之后，再根据这个数据去另外一个数据库中找对应的数据。
- 事务问题 ：同一个数据库中的表分布在了不同的数据库中，如果单个操作涉及到多个数据库，那么数据库自带的事务就无法满足我们的要求了。
- 分布式 id ：分库之后， 数据遍布在不同服务器上的数据库，数据库的自增主键已经没办法满足生成的主键唯一了。我们如何为不同的数据节点生成全局唯一主键呢？这个时候，我们就需要为我们的系统引入分布式 id 了

- 基本读写分离功能：对sql类型进行判断，如果是select等读请求，就走从库，如果是insert、update、delete等写请求，就走主库。
- 主从数据同步延迟问题：因为数据是从master节点通过网络同步给多个slave节点，因此必然存在延迟。因此有可能出现我们在master节点中已经插入了数据，但是从slave节点却读取不到的问题。对于一些强一致性的业务场景，要求插入后必须能读取到，因此对于这种情况，我们需要提供一种方式，让读请求也可以走主库，而主库上的数据必然是最新的。
- 事务问题：如果一个事务中同时包含了读请求(如select)和写请求(如insert)，如果读请求走从库，写请求走主库，由于跨了多个库，那么本地事务已经无法控制，属于分布式事务的范畴。而分布式事务非常复杂且效率较低。因此对于读写分离，目前主流的做法是，事务中的所有sql统一都走主库，由于只涉及到一个库，本地事务就可以搞定。
- 感知集群信息变更：如果访问的数据库集群信息变更了，例如主从切换了，写流量就要到新的主库上；又例如增加了从库数量，流量需要可以打到新的从库上；又或者某个从库延迟或者失败率比较高，应该将这个从库进行隔离，读流量尽量打到正常的从库上

如何实现

1.代理方式
2、组件（`sharding-jdbc`）

解决方案：ShardingSphere 绝对可以说是当前分库分表的首选

分库分表作用：（解决数据存储的问题）
分库分表是为了解决由于库、表数据量过大，而导致数据库性能持续下降的问题。

高可用

限流
限流是从用户访问压力的角度来考虑如何应对系统故障。
限流为了对服务端的接口接受请求的频率进行限制，防止服务挂掉。比如某一接口的请求限制为 100 个每秒，对超过限制的请求放弃处理或者放到队列中等待处理。限流可以有效应对突发请求过多、

降级
降级是从系统功能优先级的角度考虑如何应对系统故障。
服务降级指的是当服务器压力剧增的情况下，根据当前业务情况及流量对一些服务和页面有策略的降级，以此释放服务器资源以保证核心任务的正常运行。

熔断
当下游的服务因为某种原因突然**变得不可用**或**响应过慢**，上游服务为了保证自己整体服务的可用性，不再继续调用目标服务，直接返回，快速释放资源。如果目标服务情况好转则恢复调用。

排队

另类的一种限流，类比于现实世界的排队。玩过英雄联盟的小伙伴应该有体会，每次一有活动，就要经历一波排队才能进入游戏。

集群

相同的服务部署多份，避免单点故障

超时和重试机制

一旦用户的请求超过某个时间得不到响应就结束此次请求并抛出异常。 如果不进行超时设置可能会导致请求响应速度慢，甚至导致请求堆积进而让系统无法在处理请求。

另外，重试的次数一般设为 3 次，再多次的重试没有好处，反而会加重服务器压力（部分场景使用失败重试机制会不太适合）。

搜索引擎

Elasticsearch（基于lucene ：全文搜索引擎）是一个实时的分布式存储、搜索、分析的引擎。

介绍那儿有几个关键字：

- 实时
- 分布式（核心思想就是在多台机器上启动多个 ES 进程实例，组成了一个 ES 集群。）
- 搜索
- 分析