> 本文从研发规范层面、应用服务层面、存储层面、产品层面、运维部署层面、异常应急层面这六大层面去剖析一个高可用的系统需要有哪些关键的设计和考虑

一、高可用架构和系统设计思想

可用性和高可用概念

可用性是一个可以量化的指标，计算的公式在维基百科中是这样描述的：根据系统损害、无法使用的时间，以及由无法运作恢复到可运作状况的时间，与系统总运作时间的比较。行业内一般用几个9表示可用性指标，对应用的可用性程度一般衡量标准有三个9到五个9；一般我们的系统至少要到 4 个 9（99.99%）的可用性才能谈得上高可用。

高可用（High Availability）的定义：（From 维基百科）是 IT 术语，指系统无中断地执行其功能的能力，代表系统的可用性程度，是进行系统设计时的准则之一。**服务不可能 100% 可用，因此要提高我们的高可用设计，就要尽最大可能的去增加我们服务的可用性，提高可用性指标。一句话来表述就是：高可用就是让我们的服务在任何情况下都尽最大可能能够对外提供服务。**

高可用系统设计思想

高可用系统的设计，需要有一套比较科学的工程管理套路，要从产品、开发、运维、基建等全方位去考量和设计，高可用系统的设计思想包括但不限于：

- **做好研发规范**，系统都是研发人员设计和编码写出来的，因此首先要对研发层面有一个规范和标准
- **做好容量规划和评估**，主要是让开发人员对系统要抗住的量级有一个基本认知，方便进行合理的架构设计和演进。
- **做好服务层面的高可用**，主要是负载均衡、弹性扩缩容、异步解耦、故障容错、过载保护等。
- **做好存储层面的高可用**，主要是冗余备份（热备、冷备）、失效转移（确认，转移，恢复）等。
- **做好运维层面的高可用**，主要是发布测试、监控告警、容灾、故障演练等。
- **做好产品层面的高可用**，主要是兜底策略。
- **做好应急预案**，主要是在出现问题后怎么快速恢复，不至于让我们的异常事态扩大。

二、研发规范层面

方案设计和编码规范

研发规范层面这个是大家容易忽视的一个点，但是，我们所有的设计，都是研发人员来完成的，包括从设计文档到编码到发布上线，因此，研发层面也是有一个规范流程和套路，来让我们更好的去研发和维护一个高可用的系统：

- 设计阶段
  - 规范好相关方案设计文档的模板和提纲，让团队内部保持统一，可以参考我的文章《技术方案设计模板》
  - 方案设计后一定要进行评审，在我们团队中，新项目一定要评审，重构项目一定要评审，大的系统优化或者升级一定要评审，其他的一般研发工作量超过一周的建议要评审的。
- 编码阶段
  - 不要随便打日志
  - 要接入远程日志
  - 要能够分布式链路追踪

    - 代码编写完需要有一定的单测来保证代码的健壮性，同时也能保障我们后续调整逻辑或者优化的时候可以保证代码的稳定
    - 包括增量覆盖率、全量覆盖率，具体的覆盖率要达到多少可以根据团队内部的实际情况来定，在我们团队，定的规则是 50% 的覆盖率。
    - 工程的 layout 目录结构规范，团队内部保持统一，尽量简洁
    - 遵循团队内部的代码规范，一般公司都有对应语言的规范，如果没有则参考官方的规范，代码规范可以大大减少 bug 并且提高可用性。
    - 执行代码规范
    - 单测覆盖率
    - 日志规范
- 发布上线阶段，参考下面运维部署层面那一章节的灰度发布和接口测试相关说明

容量规划和评估

容量评估，是指我们需要评估好，我们这个系统，是为了应对一个什么体量的业务，这个业务请求量的平均值、高峰的峰值大概都在一个什么级别。如果是新系统，那么就需要根据产品和运营同学对业务有一个大体的预估，然后开发同学根据产品给的数据再进行详细的评估。如果是老系统，那么就可以根据历史数据来评估。评估的时候，要从一个整体角度来看全局的量级，然后再细化到每个子业务模块要承载的量级。

容量规划，是指我们系统在设计的时候，就要能够初步规划好我们的系统大致能够抗多少的量级，比如是十万还是百万级别的请求量，或者更多。不同的量级对应的系统架构的设计会完全不一样，尤其到了千万、亿级别的量级的时候，架构的设计会有很多的考量。当然这里需要注意的是，我们不需要一上来就设计出远超于我们当前业务真实流量的系统，要根据业务实际情况来设计。同时，容量规划还涉及到，我们系统上下游的各个模块、依赖的存储、依赖的三方服务，分别需要多少资源，需要有一个相对可以量化的数据出来。容量规划阶段，更多是要依靠自身和团队的经验，比如要了解我们的 log 的性能、redis 的性能、rpc 接口的性能、服务化框架的性能等等，然后根据各种组件的性能来综合评估自己设计的系统的整体性能情况。

容量评估和容量规划之后，我们还需要做一件事情，就是性能压测，最好是能够做到全链路压测。性能压测的目的是为了确保你的容量规划是准确的，比如我设计的这个系统，我规划的是能够抗千万级别的请求，那么实际上，真的能够抗住吗？这个在上线之前，首先要根据经验来判断，然后是一定要经过性能压测得出准确结论的。**性能压测要关注的指标很多，但是重点要关注是两个指标，一个是 QPS、一个是响应耗时，要确保压测的结果符合预期。**压测的步骤可以先分模块单独压测，最后如果情况允许，那么最好执行全链路压测。

QPS 预估（漏斗型）

QPS 预估（漏斗型），指的是一个真实的请求过来后，从接入层开始，分别经过了我们整个系统的哪些层级、哪些模块，然后每一个层级的 QPS 的量级分别有多少，从请求链路上来看，层级越往下，那么下游层级的量级应该会逐步减少的，因为每经过一个层级，都有可能会被各种条件过滤掉的一部分请求。比如说进入活动页后查看商品详情然后下单这个例子，首先进入活动页，所有的请求都会进入访问；然后只会有部分用户查询商品详情；最后查看商品详情的这些用户又只会有部分用户会下单，因此这里就会有一个漏斗，从上层模块到下层模块的量级一定是逐步减少的。

QPS 预估（漏斗型）就是需要我们按照请求的层面和模块来构建我们的预估漏斗模型，然后预估好每一个层级的量级，包括但不限于从服务、接口、分布式缓存等各个层面来预估，最后构成我们完整的 QPS 漏斗模型。

三、应用服务层面

无状态和负载均衡设计

一般要做到系统的高可用，我们的应用服务的常规设计都是无状态的，这也就意味着，我们可以部署多个实例来提高我们系统的可用性，而这多个实例之间的流量分配，就需要依赖我们的负载均衡能力。无状态 + 负载均衡 既可以让我们的系统提高并发能力，也可以提高我们系统的可用性。

如果我们的业务服务使用的是各种微服务框架来开发的，那么大概率在这个微服务框架里面就会包含了服务发现和负载均衡的能力。这是一整套流程，包括服务注册和发现、负载均衡、健康状态检查和自动剔除。当我们的任何一个服务实例出现故障后会被自动剔除掉，当我们有新增一个服务实例后会自动添加进来提供服务。

如果我们不是使用的微服务框架来开发的，那么就需要依赖负载均衡的代理服务，比如 LVS、Nginx 来帮我们实现负载均衡。

弹性扩缩容设计

弹性扩缩容设计是应对突峰流量的非常有效的手段之一，同时也是保障我们服务可用性的必要手段。弹性扩缩容针对的是我们的无状态的应用服务而言的，因为服务是无状态的，因此可以随时根据请求量的大小来进行扩缩容，流量大就扩容来应对大量请求，流量小的时候就缩容减少资源占用。

怎么实现弹性扩缩容呢？现阶段都是云原生时代，大部分的公司都是采用容器化（K8s）部署，那么基于这个情况的话，弹性扩缩容就非常容易了，只需要配置好 K8s 的弹性条件就能自动根据 CPU 的使用率来实现。

如果不是容器化部署，是物理机部署的方式，那么要做到弹性扩缩容，必须要有一个公司内部的基础建设能力，能够在运营平台上针对服务的 CPU 或者 QPS 进行监控，如果超过一定的比例就自动扩缩容，和 K8s 的弹性原理是一样的，只是需要自行实现。

异步解耦和削峰设计（消息队列）

要想我们的系统能够高可用，那么从架构层面来说，要做到分层、分模块来设计，而分层分模块之后，那么各个模块之间，还可以进行异步处理、解耦处理。目的是为了不相互影响，通过异步和解耦可以使我们的架构大大的提升可用性。

架构层面的异步解耦的方式就是采用消息队列（比如常见的 Kafka），并且同时消息队列还有削峰的作用，这两者都可以提高我们的架构可用性：

- 异步解耦：采用消息队列之后，可以把同步的流程转换为异步的流程，消息生成者和消费者都只需要和消息队列进行交互，这样不仅做了异步处理，还讲消息生成者和消费者进行了隔离。异步处理的优势在于，不管消息的后续处理的业务服务是否 ok，只要消息队列还没满，那么就可以执行对外提供服务，而消费方则可以根据自身处理能力来消费消息后进行处理。解耦的优势在于，如果消费方异常，那么并不影响生产方，依然可以对外提供服务，消息消费者恢复后可以继续从消息队列里面消费数据后执行业务逻辑
- 削峰：采用消息队列之后，还可以做到削峰的作用，当并发较高的时候，甚至是流量突发的时候，只要消息生产者能够将消息写入到消息队列中，那么这个消息就不会丢，后续处理逻辑可以慢慢的去消息队列里面消费这些突发的流量数据。这样就不会因为有突发流量而把整个系统打垮。

故障和容错设计

任何服务，一定会存在失败的情况，不可能有 100% 的可用，服务在线上运行过程中，总会遇到各种各样意想不到的问题会让你的服务出现状况，因此业界来评价可用性 SLA 都是说多少个 9，比如 4 个 9(99.99%)的可用性。

为此，我们的设计建议遵循"design for failure"的设计原则，设计出一套可容错的系统，需要做到尽早返回、自动修复，细节如下

- 遵循 fail fast 原则，Fail fast 原则是说，当我们的主流程的任何一步出现问题的时候，应该快速合理地结束整个流程，尽快返回错误，而不是等到出现负面影响才处理。

- 具备自我保护的能力。当我们依赖的其他服务出现问题的时候，要尽快的进行降级、兜底等各种异常保护措施，要避免出现连锁反应导致整个服务完全不可用。比如当我们依赖的数据存储出现问题，我们不能一直重试从而导致数据完全不可用。

过载保护设计（限流、熔断、降级）

系统无法高可用的一个重要原因就在于，我们的系统经常会有突发的流量过来，导致我们的服务超载运行，这个时候，首先要做的当然是快速扩容，并且我们事先就要预留好一定的冗余。另外一个情况下，就算我们扩容了，但是还是会超载，比如超过了下游依赖的存储的最大容量、或者超过了下游依赖的三方服务的最大容量。那么这个时候，我们就需要执行我们的过载保护策略了，主要包括限流、熔断、降级，过载保护是为了保证服务部分可用从而不至于整个服务完全不可用。

- 限流。限流是指对进入系统的请求进行限流处理，如果请求量超过了我们系统最大处理能力或者超过了我们指定的处理能力，那么直接拒绝请求，通过这种丢弃部分请求的方式可以保证整个系统有一定的可用性，从而不至于让整个系统完全不可用。怎么判别超过最大处理能力呢？一般就是针对 QPS 来判别，如果 QPS 超过阈值，那么就直接拒绝请求。
    - 限流有很多细节的策略，比如针对接口限流、针对服务限流、针对用户限流。
- 熔断。熔断，断路（开路）的价值在于限制故障影响范围。我们希望控制、减少或中断和故障系统之间的通信，从而降低故障系统的负载，有利于系统的恢复。一般我们的服务都会有很多下游依赖，如果下游依赖的服务出现问题，比如开始超时甚至响应非常慢的情况下，如果我们不做任何处理，那么会导致我们的整个请求都被卡住从而超时，那么我们的业务服务对外就无法提供任何正常的功能了。为此，熔断策略就可以解决这个问题，熔断就是当我们依赖的下游服务出现问题的时候，可以快速对其进行熔断（不发起请求），这样我们的业务服务至少可以提供部分功能。**熔断的设计至少需要包括 熔断请求判断机制算法、熔断恢复、熔断告警 三部分**
- 降级。降级是指我们划分好系统的核心功能和非核心功能，然后当我们的系统超过最大处理能力之后，直接关闭掉非核心的功能，从而保障核心功能的可用。关闭掉非核心的功能后可以使我们的系统释放部分资源，从而可以有资源来处理核心功能。
    - **熔断和降级这两个策略，看着比较像，字面的意思上来看都是要快速拒绝掉请求。但是他们是两个维度的设计，降级的目的是应对系统自身的故障，而熔断的目的是应对我们系统依赖的外部服务故障的情况。**

四、存储层面

在当前的互联网时代，应用服务基本都是无状态的，因此应用服务的高可用相对会比较简单，但是对于数据存储的高可用，相对来说，会复杂很多，因为数据是有状态的，那具体我们要怎么保障数据存储的高可用，我们来分析下。

存储层面的高可用方案的本质都是通过通过数据冗余的方式来实现高可用，将数据复制到多个存储介质里面，可以有效的避免数据丢失，同时还可以提高并发能力，因为数据是有状态的，因此，这里会比服务的高可用要复杂很多，主要体现在如下几个方面

- 数据如何复制?
- 各个节点的职责是什么?
- 如何应对复制延迟?
- 如何应对复制中断?

常见的解决存储高可用的方案有两种：集群存储和分布式存储。业界大多是围绕这些来构建，或者是做相关衍生和扩展。

集群存储（集中式存储）

集群就是逻辑上处理同一任务的机器集合，可以属于同一机房，也可分属不同的机房。集群存储，就是把多台机器上的存储数据组合在一起对外形成一套统一的系统。**集群存储适合业务存储量规模一般的场景，常规的业务数据存储一般都是集群存储方式就足够了。现在我们一般对于业务数据存储的使用，默认都是集群方式，比如 Redis、MySQL 等存储类型，一般中大型互联网公司，默认肯定都是集群存储的方式。**

集群存储就是我们常说的 1 主多备或者 1 主多从的架构，写数据通过主机，读数据一般通过从机。集群存储主要需要考虑如下几个问题：

- 主机如何将数据复制给备机（从机）
    - 数据的写入都是通过主机，因此数据同步到备机（从机），就是要通过主机进行数据复制到备机（从机）。
    - 还需要考虑主备同步的时间延迟问题。
- 备机（从机）如何检测主机状态
- 主机故障后，备机（从机）怎么切换为主机
    - 主从架构中，如果主机故障，可直接将备机（从机）切换为主机

1，主备复制

主备复制是最常见也是最简单的一种存储高可用方案，几乎所有的存储系统都提供了主备复制的功能，例如 MySQL、Redis、MongoDB 等。

主备架构中的“备机”主要还是起到一个备份作用，并不承担实际的业务读写操作，如果要把备机改为主机，需要人工操作。因此一般使用场景都是在一些内部的后台管理系统中使用。

2，主从复制

主从复制和主备复制虽然只有一字之差，但是两者是不一样的设计思路，“从”意思是“随从、仆从”，“备”的意思是备份。”从“ 的机制是要干活的，因此是承担数据的“读”操作的，一般就是主机负责读写操作，从机只负责读操作，不负责写操作。

3，主从切换

主备复制和主从复制方案存在两个共性的问题：

- 主机故障后，无法进行写操作。
- 如果主机无法恢复，需要人工指定新的主机角色。

主从切换（主备切换）就是为了解决这两个问题而产生的，具体的设计就是在原有方案的基础上增加“自动切换”的能力，当主机异常后，经过系统检测并且自动将备机或者从机切换为主机。这个是实际应用中比较多的一个方案之一，因为我们一定能够有机制保证主机异常后从机能够自动切换为主机。

4，主主复制

主主复制指的是两台机器都是主机，互相将数据复制给对方，客户端可以任意挑选其中一台机器进行读写操作，如果采取主主复制架构，必须保证数据能够双向复制。这个相对来说，要求较高。

分布式存储

集群指的是将几台服务器集中在一起，实现同一业务。而分布式是指将不同的业务分布在不同的地方，分布式中的每一个节点，都可以做集群。

分布式存储就是通过网络使用企业中的每台机器上的磁盘空间，并将这些分散的存储资源构成一个虚拟的存储设备，数据分散的存储在企业的各个角落。分布式存储中的每台服务器都可以处理读写请求，因此不存在集中式存储中负责写的主机那样的角色。但在分布式存储中，必须有一个角色来负责执行数据分配算法，这个角色可以是独立的一台服务器，也可以是集群自己选举出的一台服务器。**分布式存储适**

合非常大规模的数据存储，业务数据量巨大的场景可以采用这种方式。常见的分布式存储比如 Hadoop(HDFS)、HBase、Elasticsearch 等。

五、产品层面

产品层面的高可用架构解决方案，基本上就是指我们的兜底产品策略。降级/限流的策略，更多的是从后端的业务服务和架构上的设计来考虑相关解决方案。这里说的兜底策略，也可叫做柔性降级策略，更多则是通过产品层面上来考虑。

- 比如，当我们的页面获取不到数据的时候，或者无法访问的时候，要如何友好的告知用户，比如【稍后重试】之类的。
- 比如 当我们的真实的页面无法访问的时候，那么需要产品提供一个默认页面，如果后端无法获取真实数据，那么直接渲染默认页面。
- 比如服务器需要停机维护，那么产品层面给一个停机页面，所有用户只会弹出这个停机页面，不会请求后端服务
- 比如抽奖商品给一个默认兜底商品
- ...

六、运维部署层面

开发阶段-灰度发布、接口测试设计

灰度发布、接口测试、接口拨测系列设计包括但不限于：

- 灰度发布，我们服务发布上线的时候，要有一个灰度的过程，先灰度 1-2 个服务实例，然后逐步放量观察，如果一切 ok，再逐步灰度，直到所有实例发布完毕
- 接口测试，每次服务发布上线的时候，服务提供的各种接口，都要有接口测试用例，接口测试用例跑过之后，服务才能发布上线，目的是为了查看我们对外提供的接口是否能够正常，避免服务发布上线后才发现有问题

灰度发布和接口测试，一般在大公司里面会有相关的 DevOps 流程来保证。

开发阶段-监控告警设计

监控告警的设计，在大公司来说，根本不是问题，因为一定会有比较专门一拨人去做这种基础能力的建设，会有对应的配套系统，业务开发的同学只需要配置或使用即可。那如果说公司内部没有相关基础建设，那么就需要自己分别来接入对应的系统了。

监控系统

一般在监控系统这方面的开源解决方案包括但不限于这些：

- ELK (Elasticsearch、Logstash、Kibana) 日志收集和分析
    - 我们的日志记录不能都本地存储，因为微服务化后，日志散落在很多机器上，因此必须要有一个远程日志记录的系统，ELK 是不二人选
- Prometheus 监控收集
    - 可以监控各种系统层面的指标，包括自定义的一些业务指标
- OpenTracing 分布式全链路追踪
    - 一个请求的上下游这么多服务，怎么能够把一个请求的上下游全部串起来，那么就要依靠 OpenTracing，可以把一个请求下的所有链路都串起来并且有详细的记录
- OpenTelemetry 可观测系统标准
    - 最新的标准，大一统，集合了跟踪数据（Traces），指标数据（Metrics），日志数据（Logs）来观测分布式系统状态的能力

我们会依托开源系统进行自建或者扩展，甚至直接使用都行，然后我们的监控的指标一般会包括：

- 基础设施层的监控：主要是针对网络、交换机、路由器等低层基础设备，这些设备如果出现问题，那么依托其运行的业务服务肯定就无法稳定的提供服务，我们常见的核心监控指标包括网络流量(入和出)、网络丢包情况、网络连接数等。
- 操作系统层的监控：这里需要包含物理机和容器。常见的核心指标监控包括 CPU 使用率、内存占用率、磁盘 IO 和网络带宽等。
- 应用服务层的监控：这里的指标会比较多，核心的比如主调请求量、被调请求量、接口成功率、接口失败率、响应时间（平均值、P99、P95 等）等。
- 业务内部的自定义监控：每个业务服务自己的一些自定义的监控指标。比如电商系统这里的：浏览、支付、发货等各种情况的业务指标
- 端用户层的监控：前面的监控更多的都是内部系统层面的，但是用户真正访问到页面，中间还有外网的情况，用户真正获取到数据的耗时、打开页面的耗时等这些信息也是非常重要的，但是这个一般就是需要客户端或者前端去进行统计了。

告警系统

这些系统接入完了之后，还只是做到监控和统计，当出现问题的时候，还需要进行实时告警，因此还要有一个实时告警系统，如果没有实时报警，系统运行异常后我们就无法快速感知，这样就无法快速处理，就会给我们的业务带来重大故障和灾难。告警设计需要包括：

- 实时性：实现秒级监控；
- 全面性：覆盖所有系统业务；
- 实用性：预警分为多个级别，监控人员可以方便实用地根据预警严重程度做出精确的决策；
- 多样性：预警方式提供推拉模式，包括短信，邮件，可视化界面，方便监控人员及时发现问题

开发阶段-安全性、防攻击设计

安全性、防攻击设计的目的是为了防刷、防黑产、防黑客，避免被外部恶意攻击，这个一般有几个策略：

- 在公司级别的流量入口做好统一的防刷和鉴权的能力，比如再统一接入层做好封装
- 在业务服务内部，做好相关的业务鉴权，比如登录态信息、比如增加业务鉴权的逻辑

部署阶段-多机房部署（容灾设计）

一般的高可用策略，都是针对一个机房内来服务层面来设计的，但是如果整个机房都不可用了，比如地震、火灾、光纤挖断等。。。。那么这个情况怎么办？这就需要我们的服务和存储都能够进行容灾了，容灾的一个常见方案就是多机房部署了。

- 服务的多机房部署，这个比较容易，因为我们的服务都是无状态的，因此只要名字服务能够发现不同机房的服务，就可以实现调用，这里需要注意的是名字服务（或者说负载均衡服务）要能够有就近访问的能力。
- 存储的多机房部署，这个会比较难搞一点，因为存储是有状态的，部署在不同的机房就涉及到存储的同步和复制问题。

条件不允许的情况下，我们保证多机房部署业务服务就可以了。

线上运行阶段-故障演练（混沌实验）

故障演练在大公司是一个常见的手段；在业界，Netflix 早在 2010 年就构建了混沌实验工具 Chaos Monkey，混沌实验工程对于提升复杂分布式系统的健壮性和可靠性发挥了重要作用。

简单的故障演练就是模拟机房断电、断网、服务挂掉等场景，然后看我们的整个系统运行是否正常。系统的就要参考混沌实验工程来进行详细的规划和设计，这个是一个相对比较大的工程，效果挺好，但是需要有大量人力去开发这种基础建设。

线上运行阶段-接口拨测系列设计

接口拨测，和巡检类似，就是服务上线后，每隔一个固定时间（比如 5s）调用后端的各种接口，如果接口异常则进行告警

针对接口拨测，一般也会有相关配套设施来提供相关的能力去实现，如果没有提供，那么我们可以自己写一个接口拨测（巡检）的服务，定期去调用重要的接口。

七、异常应急层面

前面做了这么多保障，但是终究架不住线上的各种异常情况，如果真出问题了，让我们的服务异常，无法提供服务后，我们还需要最后一根救命稻草，那就是应急预案，将服务异常的损失降低到最小。

应急预案就是我们需要事先规划好，我们业务系统在各个层级出现问题后，我们需要第一时间怎么恢复，制定好相关规则和流程，当出现异常状况后可以按照既有的流程去执行，这样避免出现问题后手忙脚乱导致事态扩大。