# 如何做一个可实施的故障预案?

bd7xzz 于 2022-05-31 20:42:08 发布



### 为何要写这篇博文?

其实写这篇博文的目的有三:

1. 之前写的 《如何写一篇可实施的技术方案》 还是有一些浏览量的,也有很多人私信和留言反馈效果不错。也好久没写博客了,深夜睡不着写一下吧。
2. 小伙伴正巧需要用,问我有没有整理过这方面的资料可借鉴,因此也整理一下吧。
3. 我曾就职于京东物流、美团外卖广告,目前就职于微博视频,也积累了一些诸如:618、双十一备战、突发热点事件的应对经验,也是时候整理一下了。

我认为一个好的系统架构是面向错误和失败的,一个优秀的架构师除了能够给出解决复杂系统问题的方案,更需要给出异常和故障的解决方案。无论何时,我们应该精心设计,通过有限的资源解决复杂的问题,并最大化的预测故障,降低故障的发生概率。

在这里,我希望能够阐述清楚工作中使用的故障分析方法、故障经验总结思路、预案构造过程,以及如何去实施预案。一份合理可实施的预案不是一个人能完全梳理清楚的,也不是一个人能完全实施展开的,这离不开一个团队。如果你的系统并没有什么用户使用,出现故障可以随时下线修复,那这篇文章并不适合你,如果你的系统有大量的用户使用,或者希望你的系统有一天拥有一定规模的用户量,那这篇文章适合你。另外,这篇文章里举的例子都是虚构的,不要认真,不要认真,不要认真。

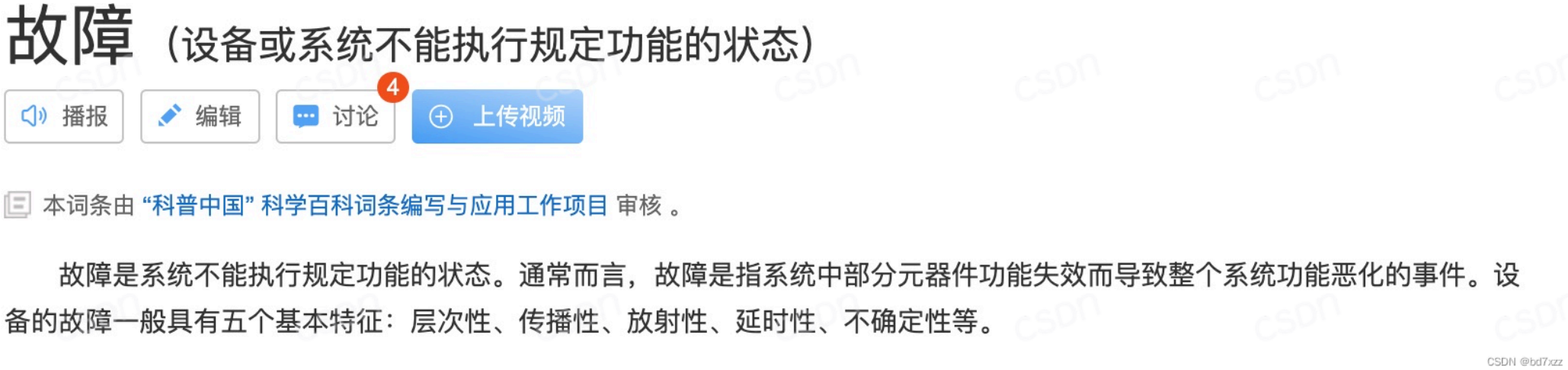

我认为,一个可实施的故障预案建设流程如下:

1. 了解系统整体架构,区分核心业务模块
2. 梳理上下游依赖
3. 评估并制定 SLA

4. 梳理系统薄弱环节
5. 预测可能发生的故障
6. 基于上述评估制定初步预案
7. 针对初步预案实施演练
8. 总结演练经验和结果，完善最终预案
9. 对最终预案进行多次演练
10. 宣讲预案

### 故障的定义?

在产出一份可实施的故障预案之前，我们先来看看系统是如何产生故障的。如果我们能够预测故障（并不能百分之百预测到所有故障），同时能够有清晰的思路认识故障，那就很容易制定一份故障预案。
首先，我们应该先定义什么是故障?
引用百度百科的解释：




百度百科中除了给出了故障的基本定义，还给出了故障的分类、基本特征、故障处理。
这里我没有完全 copy 百科中的定义，做一些白话说明和举例。

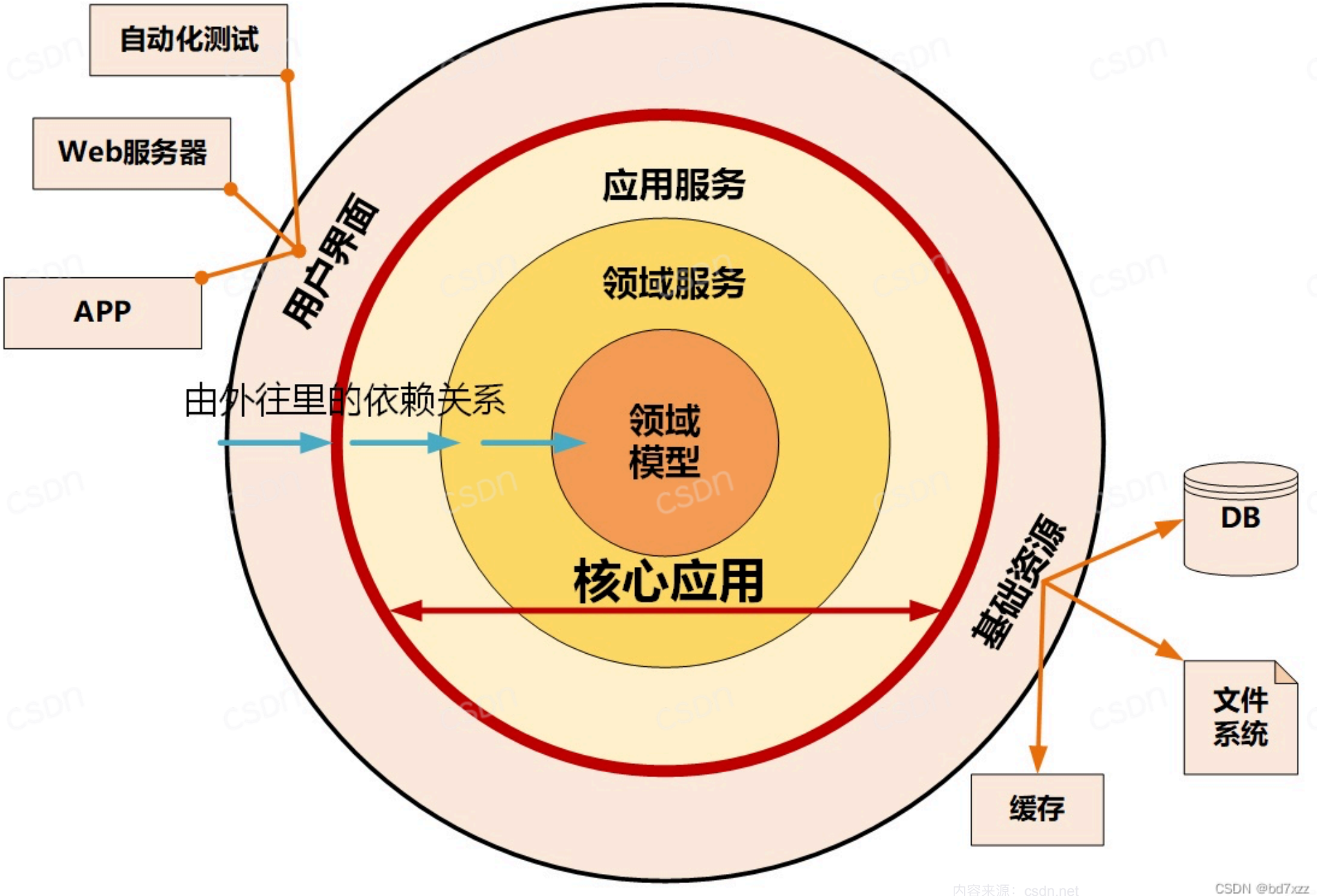
**分类：**

1. 按故障的持续时间分类：对软件系统来说，故障产生，并不一定会立刻被发现或解决，这通常取决于对系统监控覆盖范围、团队对系统的认知程度、团队人员综合技术能力。通常对事故的定级，也与持续时间有直接关系。如：双十一、618 由于下单量较大，生单模块若出现大面积故障，无法创建订单，五分钟内发现并解决为三级故障，十到十五分钟内解决为二级故障，十五分钟以上解决为一级故障。
2. 按故障的发生和发展进程分类：百度百科中给出的分类有 2 种，突发性故障和渐发性故障。对于软件系统，突发性故障可由瞬时流量增高超出系统可承受的最大负载导致，如：赵丽颖在微博宣布离婚了，大量粉丝瞬间涌入微博，超出服务器最大处理能力，导致系统崩溃。渐发性故障可由用户量逐渐递增，产生的数据量逐渐递增，若编码不慎导致了内存泄露，用户量少时内存增长不明显，用户量逐渐增加后，内存泄露被放大，内存不足产生应用崩溃。
3. 按故障发生的原因分类：分为外因故障和内因故障，外因故障如开发人员修改错了线上配置，导致业务代码解析失败，产生了拒绝服务。第二点中内存泄露就是内因故障的一个例子。外因通常由人为产生，内因通常由软件系统自身存在的问题产生。
4. 按故障的部件分类：这一点，对于我们日常的系统来说，可能涉及到软件和硬件俩部分。软件部分，包括代码 bug 产生的问题，人为操作产生的问题，异常流量产生的问题等，硬件部分包括单机故障（如磁盘损坏、断电等）、集群故障（如数据中心地震损坏、专线光缆被挖断等），一些硬件故障也是由人为产生的。
5. 按故障的严重程度分类：对于软件系统来说，破坏性故障通常是突发但非永久的，发生后导致系统拒绝服务，如：流量突增导致的服务器响应不过来，只要扩容就可以解决，或者遭受 DDOS 攻击，接入硬件防火墙、云安全服务就可以解决。非破坏性故障，出现后没有产生拒绝服务，但可能存在系统内部的异常和错误，可快速修复，如：上线新功能因为 bug 或设计存在问题导致的用户数据错乱，需要快速修复并清洗用户数据。
6. 按故障的相关性分类：对于软件系统来说，相关系统故障可能由调用的第三方接口产生，如：创建订单依赖商品信息接口，商品信息接口挂掉后，无法生成订单。非相关系统故障通常就是软件自身产生的问题，与其他依赖和边界无关。

**基本特征：**

1. 层次性：软件系统故障体现出层次性，是因为软件系统自身的复杂性，包括了子系统、子模块、微服务等。故障通常是由点到面的，所以一个复杂系统的故障也是由子系统、子模块、微服务内部产生，并逐层扩散的。
   这里说明一下，软件系统之所以存在层次性（即架构分层），是为了更好的对复杂系统进行拆解，区分出边界、依赖。在数学中，解决复杂问题最好的方法就是拆分出

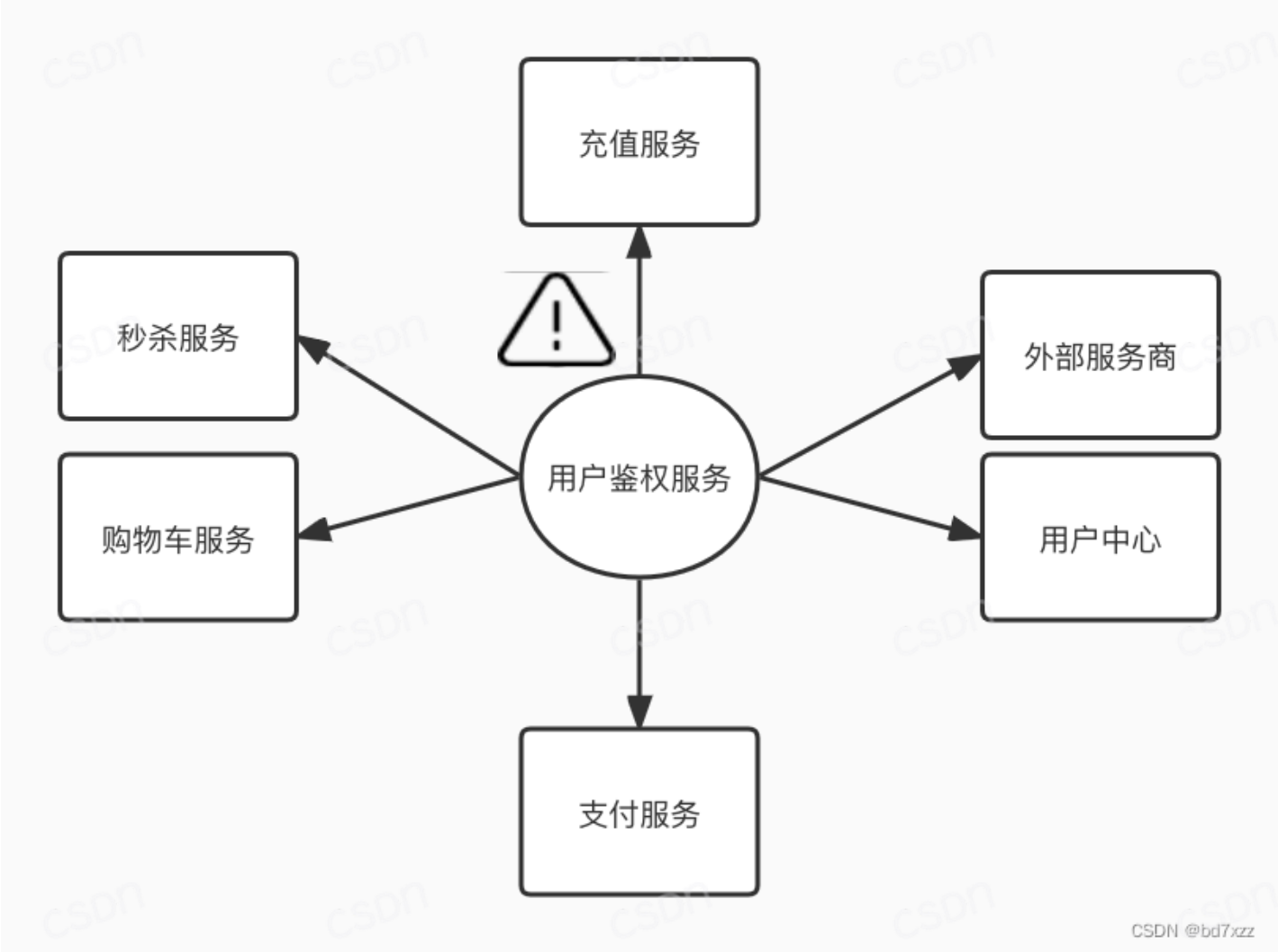
N 个子问题，逐一解决。所以这也是大厂普及领域驱动设计的主要原因。借用欧创新老师的一张图 (点击查看引用)，可以看出领域驱动设计分层的重要性。






2. 传播性：由于软件系统具有层次性，一个点产生了故障就存在逐层传播的风险。最终给用户的反馈可能是拒绝服务、显示错乱等现象。

3. 放射性：由于存在依赖，被依赖方（接口提供者）产生的异常可能传递给依赖方（接口调用者），如：用户权限核心服务产生了崩溃或数据错乱，导致鉴权异常。依赖该服务的多方应用，可能集体出现鉴权失败或越权访问。这时，各业务线可能出现的异常现象不同。




4. 延时性；系统从问题产生，到逐渐扩大，再到形成具有一定规模的故障，通常是有一定时间过程的。如：京东某商品由于运营人员配置错了商品价格，导致被 薅羊毛，血亏一个亿。这种配置错误是由运营人员疏忽导致的（包括事前准备不充分，事中配置错误，事后没有检查），羊毛党通常也不是立刻发现该问题的，一个人发现，广而告之，所以直到官方意识到问题的严重时，已经血亏一个亿。这就是有一定的延时性。

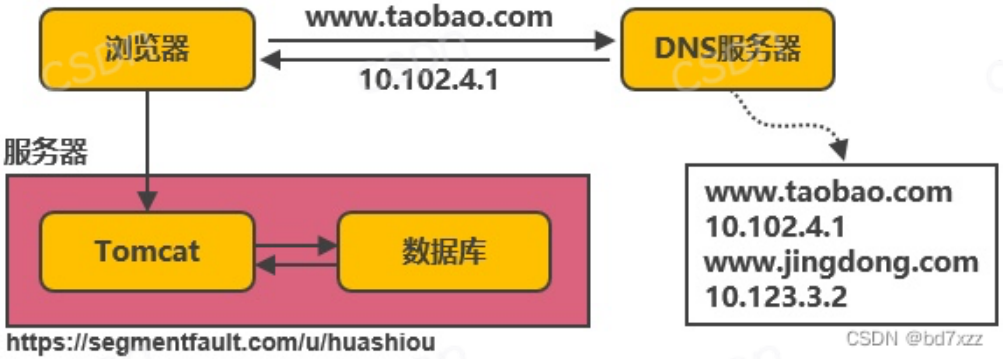
5. 不确定性：故障是不确定的，也是我们所说的风险。人们通常对未知不确定产生恐慌、担忧的心理。所以无论是由外因产生还是内因产生的故障，都会让人措手不及。同时一个复杂的系统，出现故障通无论是突发性还是渐发性都是不易排查的，甚至会产生千奇百怪的不确定性结果。

上面给出了一坨故障的定义，就是为了对故障给出一个宏观框架内的定义，基于这个宏观定义，即使系统再复杂庞大，当产生故障时，也是基于这个框架下进行分析问题、定位问题、解决问题的。

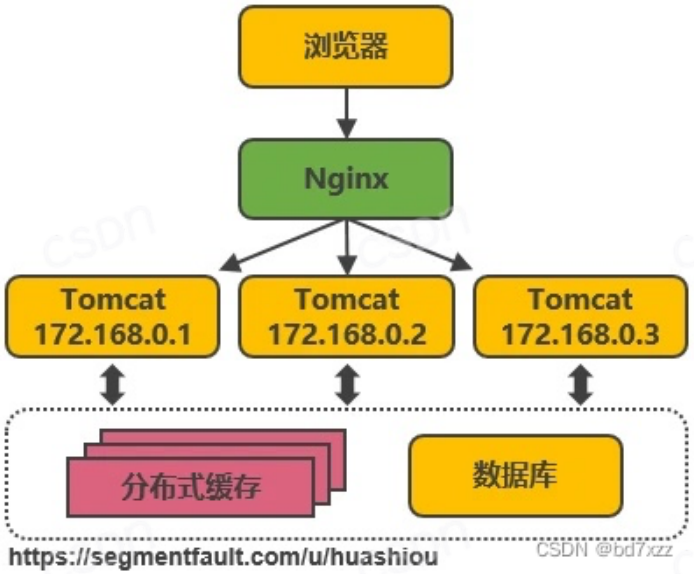
这里，我也给出一些常见的软件系统故障例子，这些例子出现的概率比较大（这些例子可能是内因，也可能是外因产生的）。从中国近 10 年互联网发展来看，服务端所采用的架构会根据系统的复杂度、用户的规模上涨，逐渐演进：单机系统架构 -> 简单多机系统架构 -> 集群系统架构（即分布式、云计算）。三次演进即应对了复杂业务，又解决了用户规模增长带来的问题，但也增加了新的系统复杂度，同时也引入了更多让人头疼的问题（如分布式情况下的副本一致性问题、时间问题等），人们为了解决这些新的问题，可能牺牲了更多的东西（性能、稳定性、一致性）。

借用一些网上的图片（点击查看引用），可以看清演进过程：

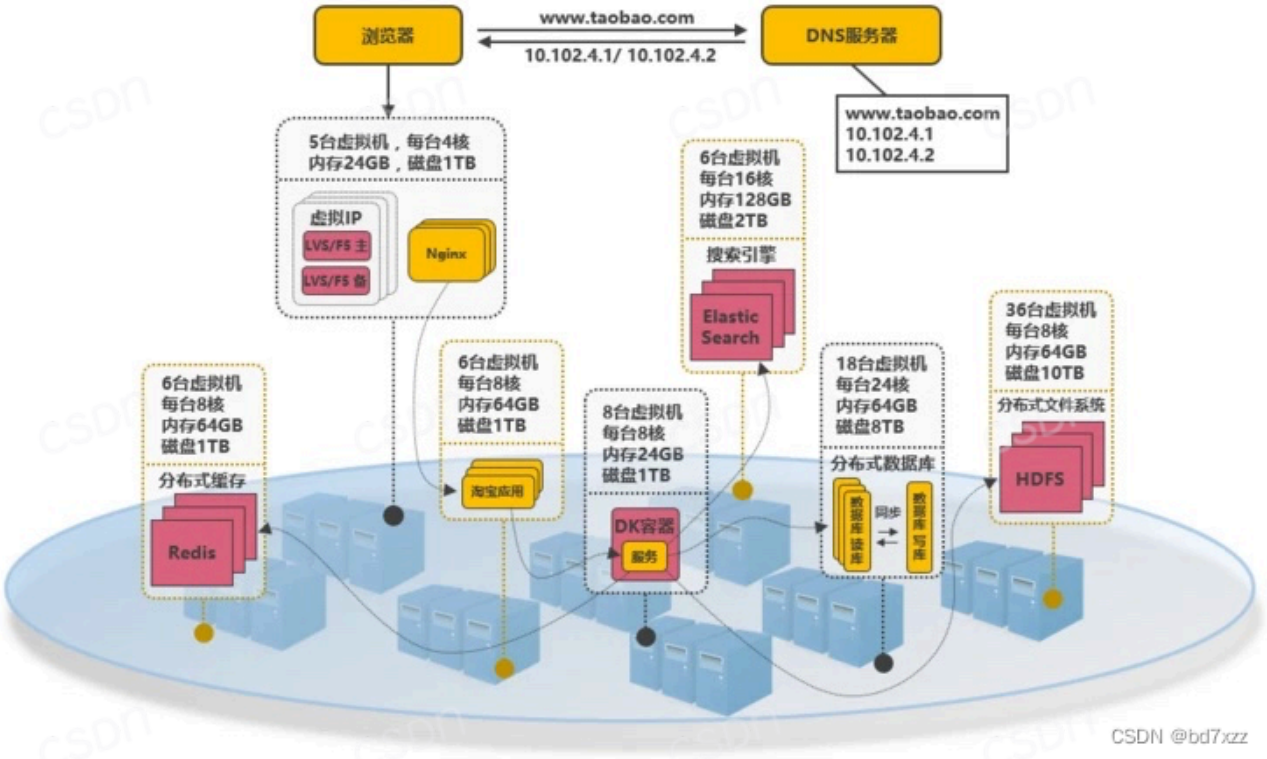
- 单机系统架构：




- 简单的多机系统架构：



- 复杂的集群系统架构：




单机系统可能存在的故障：

1. CPU 飙升或被打满
2. 磁盘 IOPS 飙升或存储空间打满
3. 内存泄露或无法分配内存
4. 网络丢包或大量重传
5. 大量 gc 导致的停顿或虚拟机、操作系统其他机制导致的停顿
6. 进程被 kill 或操作系统自身崩溃
7. 软件自身 bug
8. 硬件故障，包括不限于：磁盘坏道、主板烧毁、机器断电、网络线路被切断
9. 虚拟化 kvm、xen 等技术自身存在的 bug 或资源被抢夺

集群系统可能存在的故障：

1. 配置或设计不当导致的脑裂问题
2. 对等集群下软件 bug 导致的雪崩
3. 网络延迟导致的主从、异地数据中心不一致、数据错乱
4. 负载策略不均衡
5. 分布式的时钟同步问题
6. 单机配置或配置中心被人为修改错误
7. 数据中心遭受不可抗拒性的摧毁（地震、断电、火灾等）

系统依赖可能产生的故障：

1. 基础服务、中间件、框架自身的 bug
2. 第三方 RPC（强依赖）、MQ（弱依赖）产生故障，强弱依赖故障表现不同
3. 上下游系统更新功能产生的故障（如：三方 maven 依赖无法下载、序列化没有考虑到前后兼容）
4. 接口未提供幂等，因为重试、tcp 重传，或因 bug 导致的上下游数据错乱
5. 依赖错综复杂，梳理不清无法控制导致的混乱（包括：数据混乱、系统压力不可控）

### 如何引发故障?

当我们充分了解的故障的分类、基本特征，就可以基于这个宏观的框架，对故障进行模拟并找出解决方法，总结经验形成故障预案并进行常规化的演练。通过不断的模拟故障，并修复故障，迭代最优的预案。同时因为随着时间的推移，系统的复杂性增加、用户规模增加、数据规模增加，产生的故障会不断变化。所以预案、解决方案不是一次性可完成的。
基于上面给出的单机系统、集群系统、系统依赖可能产生的故障，这里聊聊如何引发这些故障。
单机系统可能存在的故障：

1. 通过虚拟化技术，降低 CPU 配置，或使用线上环境相同的配置进行超负载压测，可利用火焰图分析系统在有限的 CPU 资源配置下哪里耗费 CPU 资源最多。通常在 CPU 繁忙情况下由于线程频繁的切换，系统性能会急剧下降，因此会产生故障。另外系统中存在高 CPU 消耗的（时间复杂度高），也可以通过压测引发故障。
   CPU 飙升也可能是由于存在内存泄露，gc 忙于垃圾回收，大量消耗 CPU 资源。通常情况需要分析系统业务流程，针对性构造故障，如：业务从数据库读取数据，加读取的大数据量，就可能导致内存泄露。

2. 对于存在磁盘 IO 操作的业务，可以进行压测，并加大写磁盘文件的字节数，测试磁盘 IOPS 高系统会产生的问题。
   对于不存在磁盘 IO 操作的业务，可能是存在内存泄露，操作系统或者虚拟机在 OOM 瞬间 dump 内存导致的磁盘 IOPS 较高或磁盘空间瞬时被打高，可压测复现。
   对于不存在磁盘 IO 操作的业务，可能由于瞬时流量增高，打印请求日志，或 bug 导致的异常日志增加，频繁写入日志导致的磁盘 IOPS 较高或磁盘空间瞬时被打高，可压测复现，对于异常日志可人为构造异常请求进行压测。
3. 内存泄露，如上面所说，压测或增加数据量复现。内存碎片，对于存在 gc 的系统，可不开启碎片整理、更改 gc 算法、分代调整晋升老年的对象大小（强制大对象申请不到连续的空间）产生故障，观察系统表现。
4. 通过降低物理网络线路带宽，SDN、交换机、路由配置限流，机房内构造广播风暴等手段可构造出网络丢包、重传等故障。
5. 与内存泄露构造故障方案相同，或修改 gc 配置、内存大小配置，让 gc 更忙碌。
   压测业务中存在多线程互斥锁的业务逻辑，增大临界区竞争频率。
   压测业务中存在系统调用、本地方法调用的业务逻辑，通常 native 方法执行会存在 stw 行为。
6. 与内存泄露构造故障方案相同，操作系统在无法分配内存时，oomkiller 会主动 kill 掉用户态进程。
   构造 fork 炸弹，引发操作系统崩溃，或 rm -rf /*（可以在你的线上环境执行试试哦～）。
7. 软件自身 bug 通常程序员是不易复现的，所以可以找到你的 QA 同事，让他们构造出问题，基于他们的构造方式，如：拿到 payload 进行请求，增大异常请求量，观察是否因为请求量放大了 bug 产生的危害。
8. 硬件故障通常可以采用拔字法：拔电源、拔网线、拔磁盘、拔内存、拔 CPU（多颗 CPU 的服务器拔掉一颗会怎样？）

***ps: 这张图各大媒体都报道过，找不到原文了，就不标注了。***

9. 虚拟化异常，这种场景通常很难引发（之前在美团遇到过虚拟化 CPU 出现故障，系统表现为磁盘 IOPS 飙升，导致排查方向出现了问题），可以在相关虚拟化技术官网寻找 buglist，如果有触发方式，可以尝试复现。

集群系统可能存在的故障：

1. 故意修改错配置文件，重启集群节点，观察集群对故障的容错性。
   集群若存在主节点角色的情况下，2n+1 可避免脑裂问题的产生（其中 n 代表可容忍故障的失败节点数），可以配置成 n+1，使得集群备选主节点出现偶数，可测试是否存在脑裂现象。
2. 同单机系统引发软件 bug 的方法，观察对等集群的表现。
3. 主从复制结构下，可以向主节点大规模写入、修改、删除数据，可造成从节点复制延时。
   同单机系统引发网络问题，观察主从或跨数据中心复制情况。（跨数据中心构造网络问题比较麻烦）
4. 调整反向代理负载均衡策略，可配置加权，增加单节点负载压力，观察被压节点的表现和集群整体表现（一些分布式数据库会感知到单节点压力，会尝试动态调整流量降低负载过高的节点压力）。
5. 很多软件系统依赖墙上时钟，没有构造合适的分布式逻辑时钟，更没财力建设原子时钟。所以可关闭 NTP，并修改集群内节点的系统时间，进行回拨，观察系统表现（如：订单号用 snowflake 会生成重复）
6. 人为修改单节点配置，观察单节点和集群的表现
   对于采用集中化的配置中心，要测试强行杀死配置中心，修改错配置，两种情况系统的表可能现不同
7. 不可抗拒的事故，如图所示。。。

**我他妈直接拔你充电器**

系统依赖可能产生的故障：

8. 基础服务、中间件、框架的官网都会有 buglist，根据说明进行构造即可产生故障。
9. 对于强依赖的接口，第三方参与故障演练，构造异常。
   微服务接口，需要构造注册中心异常，观察 provider 和 consumer 的异常表现。
   对于消息队列弱依赖情况，可以对生产者、broker 集群、消费者执行强制停机、构造单机硬件资源打满，观察三者的系统表现。

10. 针对序列化没有考虑到前后兼容问题，可根据实际采用的 RPC 技术，开发测试接口，模拟修改接口出入参字段类型、字段名、新增删除字段等行为，观察 RPC 框架的表现（着重观察依赖方的表现）。
11. 删除、查询通常是幂等的，插入、修改不是幂等的，对非幂等接口多次调用必定会产生异常数据，观察异常数据对系统产生的影响。
12. 错综复杂的依赖，理论上没有什么合适的方式构造。是否可能产生故障，或产生什么样的故障是无法预测的。

## 重点是什么?

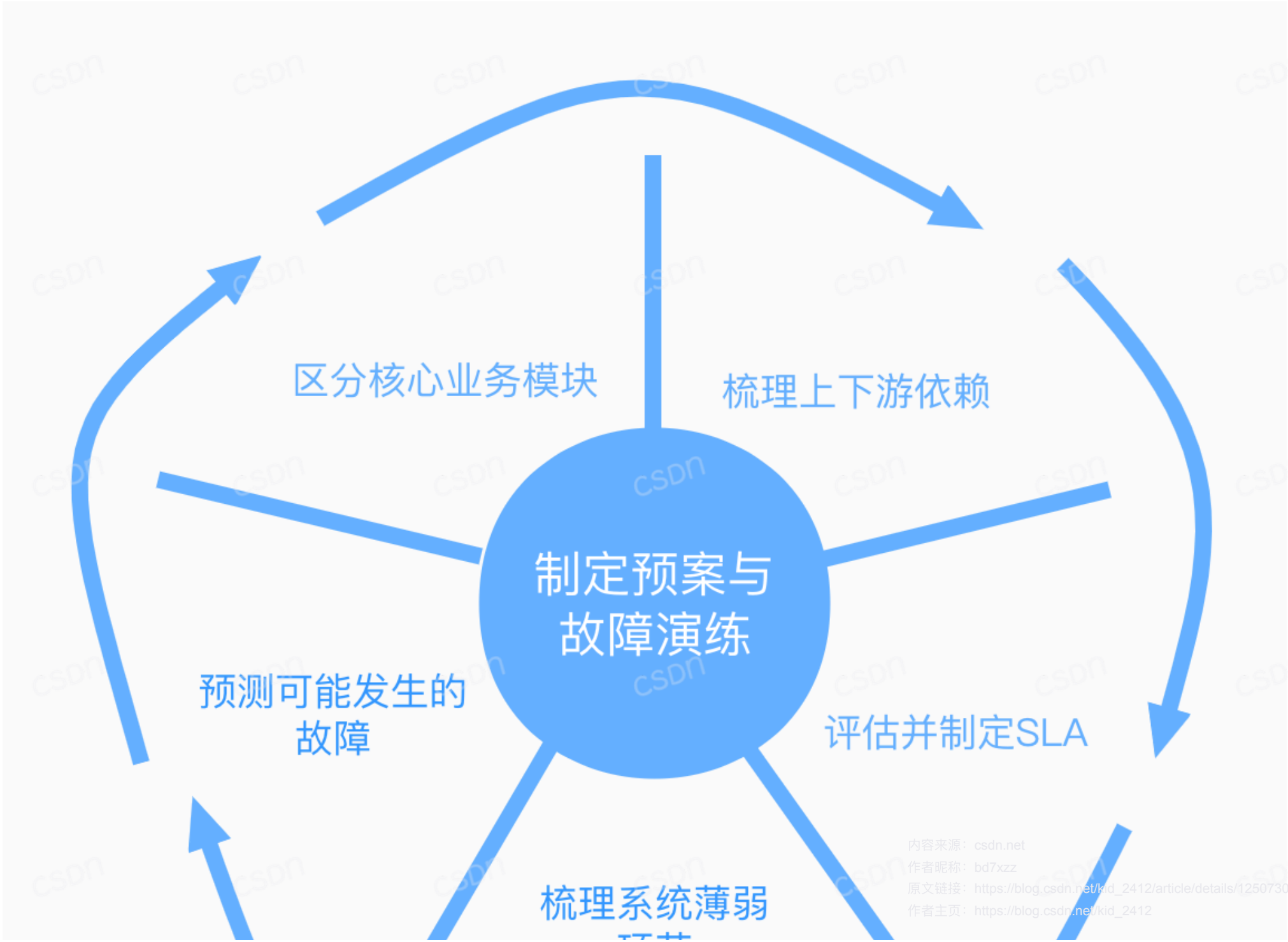
上面，穷举出了我遇到过及常见的故障，并给出了构造故障的方案。理论上，我们就可以基于这些构造方案进行测试演练，并针对实际系统的表现提出解决方案。
我们看看构建预案的重点是什么？拿文章开头说的预案建设流程，拆开细说。

1. 了解系统整体架构，区分核心业务模块：无论是程序员还是架构师都需要了解自己负责系统的整体架构。架构师熟悉系统架构有助于构造故障演练预案，并针对核心模块和非核心模块评估出合理的故障演练流程和对应的解决方案。
   区分核心与非核心不同的级别，主要目的是针对核心模块投入更多的时间和人力保证能够覆盖足够多的故障，并对每个故障产生的影响（如：拒绝服务，脏数据）提供完善的解决方案。并对方案产出自动化的处理工具。
   程序员了解系统架构有助于对系统有整体的认识，在故障演练中，能充分理解故障产生的连锁反应和影响范围，并实施架构师给出的解决方案。
2. 梳理上下游依赖：由于故障具有传播的特性，且复杂系统存在模块或第三方依赖。无论是对数据库、中间件、第三方接口、内部模块的互相依赖，都需要进行梳理。针对核心模块的依赖，需要三方进行故障演练。着重关注被被依赖（接口提供方）崩溃、异常后依赖（接口调用方）产生什么响应，当依赖方流量瞬时飙升，被依赖方能否正常提供服务。所以全链路压测是必须要实施的。对于核心模块的被依赖方，若出现故障通常采用降级、切换备选通道（备选方案）的方式确保核心服务的可用性，降级需要评估是否对核心业务的数据造成影响，评估降级、切换备选的情况下数据是否可做到最终一致性。
3. 评估并制定 SLA：SLA 是软件服务质量等级，通过制定合理的 SLA 给出明确的目标，才能证明给出的故障预案是有效的。如：接口 P99<20ms、稳定性 99.9 等。试想压测中，没有 SLA 的约束，压测结果表明接口 P99>700ms，并在一定的流量冲击下系统出现崩溃，由于没有强约束研发人员对此并不感到重视，这种异常对用户来说是无法忍受的。即使去做故障预案，也没有任何意义。
4. 梳理系统薄弱环节：薄弱环节通常是系统的边界、依赖、性能最低的部分。比如：生成订单消息消费侧依赖了几个比较重的第三方接口，日常流量比较稳定，单条消息处理最大可容忍在 20 秒内完成。但在双十一、618 的大流量冲击下，单条消息处理可能由于资源繁忙，处理的较慢导致整体消费速度增高，消息大量积压，无法给用户生成订单，甚至出现消息队列 broker 崩溃的现象。所以系统日常中看似不起眼的异常都可能在一个时间点爆发严重的问题。梳理系统薄弱环节并进行提前优化是非常重要的。
5. 预测可能发生的故障：基于单机系统可能存在的故障、集群可能存在的故障、依赖可能存在的故障，我们可以提前预测故障的发生，并基于经验给出解决方案。
6. 基于上述评估制定初步预案：基于上述评估，我们制定初步的预案，初步的预案通常是架构师、系统负责人进行整理的，产出文档进行讨论，针对讨论结果优化预案、更新文档。
7. 针对初步预案实施演练：根据初步预案文档实施演练，通常是在线下进行全面演练，线上对可演练的（不会产生误伤）模块进行演练。演练过程中，要建设完善的监控方案、全链路跟踪方案，保证故障可监控报警、可跟踪排查。监控报警可能是日常逐渐迭代建设起来的，在演练过程中调整监控范围，调整报警阈值，演练时模拟突发

情况，观察监控是否覆盖到位，报警是否及时，报警信息是否全面可快速定位问题。全链路跟踪依赖于基础组件的建设，可以采用 skywalking、pinpoint 等开源链路跟踪系统，在故障产生时，通过异常流量的 trace 编号，在可视化的系统中查询出调用栈，能够快速定位问题。

8. 总结演练经验和结果，完善最终预案：演练完成后总结结论，如：每个模块在什么情况下产生什么反应？异常如何修复？如何优化避免产生故障？通常可以形成一个固定的模板。注意：这里所说的最终预案为阶段性最终预案，系统功能是不断迭代的，复杂度也在不断升高，能产生故障的可能性也在不断升高，每一个阶段都要不断迭代升级预案。
9. 对最终预案进行多次演练：对每个阶段的最终预案，进行反复演练，可以做到出了事不会慌。甚至预案中一些压测用例可作为自动化日常压测，这样可以对系统性能有一个整体把控。
10. 宣讲预案：最后，宣讲预案是很重要的，预案产出了知识，传递知识给团队中每个角色和人员。值班人员要有主备，在后面出现故障时可随时执行预案中的解决方案，避免故障的扩散，及时止损。

我提取出了前面的五个关键步骤作为下图，图中的边代表了循环迭代。我认为，每次预案的迭代，都离不开这五个步骤。




## 拿模板说说

我经过了这些年的故障预案建设经验（参与或主导），总结出了几个基本的模板，这里列出来希望对你有所帮助。

### 故障预案

故障预案与方案设计模板类似，要说明背景、目标、预案负责人、系统负责人、值班联系人。
背景通常简单说明即可，如：保证 xx 系统稳定性，应对 xxx 突发事件，降低 xxx 危害。
目标可参考 SLA 的制定标准给出。
预案负责人要给出负责人的姓名、联系方式，防止出现故障时对预案不明确的人其中的解决方案无从下手。
系统负责人要给出负责人的姓名、联系方式，系统负责人应是对预案最了解的人之一，可以快速指导他人做故障处理。
值班联系人，这应该是一个排班表，根据实际可动态调整。
除了上面几点，整体预案可参考下表：

| 模块 / 接口 | 是否核心 | 监控和报警策略 | 依赖关系 | 是否可降级 | 备选方案 | 异常修复工具 / 解决方案 | 负责人 | 演练记录 |
|---|---|---|---|---|---|---|---|---|
| 写明接口或模块名称和作用 | 标注是否是核心模块或接口 | 给出监控地址、报警配置阈值、报警方式、报警关键信息 | 配备依赖关系图、依赖说明、第三方依赖联系人 | 标注是否可降级，降级执行流程，写明有损无损，有损带来什么问题，如何解决 | 同是否可降级 | 说明异常现象和影响范围，给出解决方案链接或自动化工具 | 给出模块或接口负责人的姓名、联系方式，包括备选负责人 | 是否演练过，演练实施人，演练时间 |

### 故障复盘

故障事后复盘起到迭代知识的作用，并不是形式主义的内卷和推卸责任。通常采用 5w2h 分析法 + 时间线的方式。包括：事故描述（what)、原因（why)、谁导致的（who)、发生故障的时间线（when)、何地发生的（where)、如何解决并避免再次发生（how)、事故定级和损失（how much)。

引用一张网上的图 （点击查看引用），这篇文章也给出了 5W2H 分析法的由来和实施方法，推荐阅读。


这里不得不说，美团有一套很不错的事故复盘机制，包括事故响应机制，实现了事故发生及时发现并自动化地拉动相关负责人、故障跟踪人建立沟通群，自动在系统中创建事故复盘记录，并由跟踪人员填写相关信息。整个复盘流程采用 5h2w 分析法，并对 when 支持了时间线功能，可记录精确到秒的事故过程和现象。有兴趣可阅读 美团人是如何复盘的？、美团点评酒店后台故障演练系统。

虽然没有像美团那么完善的内卷机制，我们可以采用如下表格进行复盘：

| 时间点（when） | 事件点（what） | 行为（what） | 产生的现象（how mush） | 执行人（who） | 执行根因 （why） | 是否解决问题 （how） |
| --- | --- | --- | --- | --- | --- | --- |
| 具体时间点，精确到时分秒 | 说明这个时间点发生了什么 | 针对事件点做了什么事 | 做了什么事产生了什么后果 | 谁做的这件事 | 为什么要这么做，你的想法是什么 | 是否解决了问题，是否引发了新的问题，新的问题可以作为下一个事件点 |

这个表格给出了 4 个 w 和 2 个 h，其中 what 有两个，分别是：事件点描述发生了什么，行为说明做了什么。2 个 h 分别是产生的现象，即行为产生了什么结果（是好是坏？），是否解决问题，说明了最终解决结果，以及如何解决问题的。没有给出 where，是因为针对不同故障对于 where 的定义有所不同，如数据中心故障的 where 可能是个具体位置，人为修改错配置导致的故障 where 可能是修改了哪个配置。所以 where 应该按需配备的。
这里没给出事故定级，是因为每个公司有自己的定级标准，在中国互联网公司里定级通常与相关人员的绩效挂钩，高级别的事故可能会有人被离职（老油条甩锅，新人成为背锅侠）。

#### 值班表

值班机制是快速应对故障最直接的办法，日常值班人员在没有出现故障时也应该定时对系统进行巡检，防止出现故障。出现故障时，值班人员是第一介入人，自己根据预案进行解决，解决不了则协调相关负责人进行解决。理论上值班人员应该 24 小时在线（美团领导说的）。

值班表可采用如下表格：

| 系统 | 主值班人 | 备值班人 | 主值班人联系方式 | 备值班人联系方式 | 值班时间 |
| --- | --- | --- | --- | --- | --- |
| 写明是哪个系统 | 主值班人姓名 | 备值班人姓名 | 备值班人电话、即时通讯、邮箱 | 主值班人电话、即时通讯、邮箱 | 值班时间（通常是天级） |

### 实施故障演练

基于故障预案中给出的故障行为进行演练，包括故障注入、压测等操作。包括线上演练、线下演练。线上演练要注意评估是否对业务产生误伤。故障注入可人工注入或通过 chaosmonkey 系统自动注入故障，人工注入包括不限于：

1. 拔字法
2. 强制关机、kill 进程、rm -rf、fork 炸弹等
3. 修改负载调度、降低内存、降低网络、CPU 配置等
4. 错误修改系统配置
5. 渗透测试：SQL 注入、XSS 攻击等

chaos 系统可自动化执行上述异常，构造故障。通常，大型互联网公司都会构建或者改造开源的 chaos 系统，进行日常捣乱演练。开源 chaos 系统包括 netflix 的 ChaosMonkey、pingcap 的 Chaos Mesh 。

在故障演练后，总结经验，提出解决方案落实文档，并尝试针对某些具有共性的问题开发自动化工具。

## 总结

1. 从物理学和数学的角度来说，封闭系统是熵增的，宇宙万物无法避免熵增，人类一直在努力尝试控制熵增。故障和问题是无法彻底消灭的，只能基于经验进行预测并尝试解决，迭代认知，传递知识。
2. 永远记住，好的系统设计就是在考虑异常和最坏的情况。
3. 人们会对未知不可控的事情感到害怕和恐慌。问题和故障并不可怕，可怕的是出现问题前后的表现：形式主义的复盘、长篇大论凑字数的技术方案是内卷的根源，过分自信和非理性繁荣是产生问题的根源，过分胆怯不敢思考不敢执行是产生问题的根源，傲慢且不去做认知迭代是产生问题的内因。
4. 解决一个问题会引发更多的问题，需要评估解决这个问题而引发的其他问题是否可控，评估解决问题的投入产出比。
5. 不要满足当下，如果你制定了一分不错的预案，要反复演练，不断迭代。
6. 量变产生质变，无论你写的代码多优秀，系统设计多美妙，从千级别的并发访问到万级别的并发访问，再到亿级别的并发访问，都会产生不同的问题。你需要根据实际的业务需求，评估流量、数据量进行压测。压测是故障预案中必要的。
7. 故障分析、寻找解决办法、整理预案、演练、宣讲每个阶段都需要有相应的产出，如：文档、自动化工具等。