**没有失败的变更**

变更对于金融系统来说，意味着代价和风险。变更（特别是）精诚需要按固定的步骤，多个不同的系统之间配合，公司内外的配合。这也是大公司比较害怕变更和不经常做变更的原因。

Devops意味着更快的变更以及经常的变更。这个听起来，增加了变更的风险。但是它的效果是，经常变更能降低这个风险，如果你正确的操作的话。

Puppet 组织了每年的DevOps学习。 我们在2016年DevOps 状态 报表中有如下发现：

DevOps 高效的实施了200+ 变更，远胜他们的低效的竞争对手，with lead times that more than 2000x shorter. 但他们有一个3倍的变更成功率。而且当他们一旦做错后，他们回复速度更快（24倍）。

他们是如何取得这么好的效果呢？把一个变更拆分成很多小步骤，有助于更好的审查和测试。

做一个生产变更 需要将变更拆分成很多可以自动化和标准化的小步骤。认真的控制和观察变更。这样碰到问题时，就能快速的响应。通过从失败和错误中学习，找到根因，并且持续的改进这些系统和操作。降低这些问题重复发生的风险。

所有的这些要求：在变更的时候，不仅考虑你是如何去交付软件，而且要考虑你是如何看待软件以及你是如何怎么决定软件是怎么运行的。运维开发人员需要像操作员一样去思考，并且考虑在生产中如何安全的做变更。操作规范和用户期望一样的重要。

**最小化变更的风险**

在传统的IT企业你通过仔细的规划、详细的文档和纸上演练，请具有综合能力和高超的管理技能审查，来降低变更的风险。在DevOps中，你通过拆分变更步骤，明白每一步操作的含义，使这些步骤简单化和自动化，并且监控这些操作的运行情况，反复的测试这些步骤，直到变更变得可预测和按既定的路线执行。

这也是一个很好的例子：如果这事情让你感到痛苦，反复的做它，直到不痛为止。

它强迫你去简单和标准化每一个步骤，减少人工测试以及审批环节，通过代码替代文档，让你在部署中找到问题的根因。以及增强你的信心。

尽可能的自动化所有的步骤。自动化编译的步骤，自动化测试。用不变的分析去自动审视代码，发现一般的代码漏洞会议及安全风险。自动化提前检查依赖和配置文件。标准化和自动化软件的部署和系统提供。

在部署的之前、之中、之后都采用自动化检查。用自动化替代变更的检查列表以及操作步骤清单。用脚本来代替健康性检查以及自检，发现问题并提前修复他们。

通过持续交付和基础架构代码化去尽可能减少生产和测试环境的差异。使用相同的工具和步骤对测试和生产环境做变更，这样你在生产环境，你就会知道它是怎么运行的。因为你在这之前已经做了无数遍，并且发现了所有的软件错误和缺陷。

如果你把关注点放在减少风险和开销，尽可能的让工作简单和自动化，尽早的发现问题和解决问题，那么你将部署的会越来越快，越来越好。

设计和部署持续交付的流水线需要以下人员参与：架构师、开发人员、测试人员、打包和发布人员、操作和系统管理员，信息安全人员，和风控合规人员。这些越通力合作，越能从彼此学到东西。操作工程师将会提前知道这个软件是如何设计的，并能提供关于如何清楚输入的建议。开发人员可以从中学习到生产环境的建立以及这个系统是怎么运行的。安全合规人员将会了解变更是如何实际进行的以及如何去平分工作流以及使用工具去控制。显而易见的是，他们需要融入到对方的工作中去，假设做的越少，意味着他们将会做出更好和有效的决定，以及翻更少的错误。

**减少批次变更的量**

渐进做变更，更小的变更的好处在IT中的综所周知的。然后，在金融企业，我们将看到大的版本，甚至企业级的变更，这将造成不必要的风险。

自动持续的交付减少了实施的人力。这意味着，我们只需要经常做一些小的变更，是一条减少可能风险的道路。

从整体到微服务

在ING 的工程师团队以及很多其他的组织，遵循着亚马逊和奈飞的方法。将大的，一整块的应用程序分解为小的，灵活组合的微服务，这些微服务将被小组进行设计、编译、测试、和部署、管理。这将还允许小组跑的更快以及独立的做变更，只要他们维护他们自己服务器的API接口即可。

这是一条对走在降低SOA路径的企业很有意义的事情。

然而，微服务的自由和灵活是需要付出代价的。一个单个交易需要很多服务的配合和共享信息。这意味着：

1. 你需要很小心的处理错误和失败的恢复场景，你需要明白在所有依赖的服务会有哪些错误和失败的情况，并且不能遗漏在其他服务变化的时候。
2. 排错和调试将会更加的复杂当你跟踪服务的调用栈。
3. 延迟将会增加，因为服务之间的交接。
4. 尽管一个微服务的功能性测试是简单的，点对点的测试将会变的更加复杂，随着你不断的增加服务。
5. 仅仅停留在任何单个微服务的表面是容易理解的，微服务是可以快速的部署在系统表面。你需要定义和确定这些服务和控制的边界在处理关键数据时。

对生产系统来说，小变更将会更加的安全。其中的一个方法是使用隔断：在一台服务器上发布版本，并且检查这变更是没有问题的。然后把它部署到两台服务器上并且再次检查他们，然后四台，然后扩展所有的服务器都被更新了。

另外减小风险的一个方法是”dark launching”,一个特性只针对部分用户开启，或者针对部分产品，并且在将变更实施到其他更多用户之前仔细监控变更结果。延伸到变更，这在金融系统是一个共识。举例来说，邮件系统经常发布新的特性和规则在字母A-C的金融产品上，然后再是D-F,以此类推。作为逐步发布版本的一部分。

你需要去鼓励开发人员和操作工程师从小的方面去思考，并且通过迭代，做小和安全的变更。

从搭建脚手架开始。保护特性交换机。重构也要从小开始：No “root-canal refactoring” 允许。这对于已经习惯原有开发模式的开发人员来说，是一个负担。甚至对使用灵活方法做开发的开发人员来说也是一个负担。对于运维人员来说，更是困难。因为基础架构的厂商支配了网络、数据库、操作系统、服务器、和存储的变更。

**更早的定位问题**

为了更快的从失败中恢复，你需要尽可能的早的回忆哪里发生了错误，这意味减少失败的定位原因。

DevOps社区一致致力于在一个大的事件中，如何去改进尺度和告警同发现异常处理。以及发现加速系统应用和事件的模型和趋势。

金融系统已经有了很强的监控能力，当有事务排队时，观察异常和延迟，and online surveillance and anti-fraud systems to catch anomalies in system use. 这是另外一个区域 ，在这里金融服务操作更好的和DevOps 社区去学习和改进，通过如Monitorama hackathon 事件。或者通过给开源项目如 Etsy’s的StatsD+Graphite+Carbon或者Elastic Stack ,Graylog 或者Prometheus做贡献。

DevOps 分担了操作和开发团队之间的监控责任，列出生产标准、异常日志、以及告警给开发人员。这个非常重要当变更在提出时。

开发人员编写代码；他们知道当出现一个impssible代码错误出现时意味着什么。脸书有一个实施规则：如果一个开发人员的代码不能部署到生产环境，如果开发人员不能在线的观察和确保这个变更是成功的。

在DevOps工作室，当他们的代码出现问题后开发人员需要按时在线。幸运的是，通过类似PagerDuty的工具很容易将开发人员添加到升级的阶梯里面。

日志、审计、告警和实时的编排必须成为团队操作的一部分：可接受的特性标准和变更。代码审阅和测试需要包括检查告警，日志和错误处理逻辑，审计逻辑，和实时的编排。测试需要保证新的特性和服务器已经正确的加入到监控栈里面并且敏感数据没有暴露在日志里面。

**减少MTTR**

当前大多数金融企业致力于增加平均无故障时间。通过仔细的架构设计，规划，测试和发布编排，这个意味平均恢复时间对于

DevOps 工作来说是一个关键指标。他们假设这失败将会发生，特别是在互联网的范围。正如James Hamilton（亚马逊现卓越工程师）指出：在他的论文”在设计和实施互联网服务” 中，组件故障的事件混合在上亿的事件中是一件非常常见的事情。当你运行上千台服务器提供上亿的服务组件时。当系统达到这种级别时，操作错误将不可避免发生，特别时在做变更的时候。

清晰的明白组件时如何配置的，以及系统每个部分的状态，发挥自动化配置管理的杠杠是非常重要的一步。明白这些，你就可以快速定位变更和回滚变更。

**随时准备回滚**

当一个大型在线金融系统发生严重的问题时，向前推进和快速推出一个批处理并不是一个明智的选择。你需要有足够的信心你能够回滚，并且这是可行，这时安全和快速的，并且不带来更多的问题。这意味这要涉及的更方便的回滚，以及在数据库和API中创建向前和回滚的灵活机制。

这也意味正确的构建回滚步骤和代码正确的运行，这是你实施过程和持续交付的一部分。

**时刻准备事件响应**

在一个金融企业中，一个中断往往被做为一个安全事件，这将会带来法律法规是否遵从的反应。而且要求正式的升级和告知相关人以及监管者。

因为这代价是痛苦和高昂的，大部分的企业都没有充分的测试异常和失败的机制。这也是奈飞的Chaos 猴子引起很多人关注和争议的地方：它自动随机的在生产环境注入失败来测试系统的健壮性，以及团队对失败的处理能力。

游戏日练习-做一个真实，大范围的失败测试（如关闭一个数据中心），这在亚马逊、谷歌和Etsy的DevOps团队是一个常规操作。

这个（如谷歌）让上百名工程师一起工作很多天，去测试灾难恢复场景以及评估企业在对企业处理处理这些真实故障的能力。

在Etsy,游戏日练习在生产环境中运行，甚至关键组件如支付处理也在其中。正如John Allspaw指出：

为什么不在QA或者演练环境来模拟呢？首先，这些环境的任何区别将会对测试带来不确定性 ，其次，在接下来的测试中没有恢复的风险，这将会带来隐形的假设比如失败的容忍度对于设计和恢复。而我们的目标是减少不确定性，而不是增加它。

这些练习将会提前被小心的测试和规划。在小组中通过头脑风暴失败场景并且为它们做准备，通过现在测试环境中运行然后解决可能出现的问题。然后在生产环境中执行，执行人员和开发人员仔细监控并且随时准备恢复，特别是发生了不可预知的错误。

当这演习结束后，团队对整个事件做一个回顾和总结，并且指出需要改进的点。

在生产环境中注入失败并不仅仅是为了测试系统的健壮性，它也可以测试组织在处理这些失败时的能力，并且给你们团队的技术带来信心。

那我们是否希望在证券交易所和大型银行的生产环境中做这种大范围的失败测试呢？不是这样的。对于在生产环境中做这样的测试是否合法的争论一直存在，在金融系统的生产环境中做这样测试的风险是巨大的。

即使你对自己系统在失败的环境中有足够的信心，但你依然不能预测其中参与的系统会有什么样的情况发生，以及对于它们的影响，以及这些系统失败时会如何。

基于这种原因，火灾演习常常在非生产事件，在可控的情况下在生产环境中演习。这个包括工业界广泛的BCP测试，比如美国每年的SIFMA测试。相对来说，他们也许模拟运行，这样不会直接影响生产系统。比如昆腾破晓，这个模拟在工作日全行业的网络攻击，来测试参与者的故障响应能力以及内部的沟通和上报流程。

这些类型的测试，并不是完美和完整的，在评价系统以及建立更好应对事件的能力中依然值得商榷。

**发掘事件最根本的原因**

生产事故代价是巨大和让人有压力的，但是他们也确实提供了有价值的信息。一个核心的想法是在多级别上，DevOps可以用这些信息去学习和改进。

这需要收集尺度去判断哪种类型的变更是有风险的，那个团队的变更是太频繁了，或者不够频繁，通过变更的类型、大小、频率。

并且把这些和生产的问题相关联。在2010年，John Allspaw在他的论文”操作的元标准-你当前需要为变更付出的代价’中对这个想法有个非常好的表述。

事后无责检查是DevOps文化的另外一个重要部分。操作员和开发人员一起针对哪里错了以及明白这个错误是怎么发生的，讨论怎么去做可以避免它再次发生，然后给企业以及客户分享所有信息。

事后分析不是新的提法，特别是在金融企业。在一个重大的事故发生后，监管部门会按照既定流程去公开调查。这个与DevOps最大的区别在于DevOps的重点是去无责的分析这个问题并且在团队中分享这个信息，并且引导回顾流程在小的事故上做为一种学习经验。

要明白，是人就会犯错，安全机制也会失效，事故是会经常发生的，DevOps事后分析让人们在一个开放和安全的环境中去分享发生了什么和明白为什么它发生了，关注在事实上以及问题的解决上，召开一场讨论以及创造一个学习的机会。

这不是去确定谁的责任和摊派责任以及决定谁应该被开除。它也不是去列出一个需要改正的错误清单。而是一个需要去改进的流程。

事后分析是一个展示错误以及它们为什么发生的机会；这样可以去更深层次的去面对技术/企业的问题，如设计上是否具有弹性，培训，决策，以及沟通上的问题。进而去发现这些问题的根因。以及指出如何让这些企业更好。把关注点放在事实上，并且相信彼此，在彼此之间分享信息，事后回顾是给开发和操作人员更加协同的一个方法。

我说过，我们在这里必须找到根因。因为(回归到原点：2011年Velocity论文”高级的事后回顾以及人类101错误”)

在一个复杂的系统中一个给定的事件中，这个没有根因。这更像是有一个巧合导致了这个失败。

在事后看来，这是一个单一的行动，这会阻止事件的发生。当管理者在搜索根因分析时，这只是其中一件事。他们希望去阻止这些事件再次发生。但是在很多事件上，这次事件可能只是一个巧合。这样使得去查找根因是一个没有意义的事情。这会导致对安全的一个错误认识，在一个复杂的系统，有太多造成事故的原因。

#### 从DevOps的角度来看奈特事件

让我们来重新回顾下奈特事件，通过DevOps的视角。从DevOps的角度来看，这说明一系列控制节点的失效。

自动发布/部署

一个操作工程师按照手册去部署一个变更，但是在其中一台服务器上忘记部署代码，不幸的是没有人发现这个错误。

这个就是自动配置管理和发布需要解决的问题。一个审核过，自动化的，并且良好测试的流水线，有事后发布检查，烟囱测试（包括检查所有服务器上的版本是否一致）去检查所有的部署是否成功来规避这类事件的发生。

Dark Launching and Branching in Code

一个减小代码变更的风险是，将变更的代码放在一个特性开关后面，这样操作员就可以通过开启或者关闭某个标志实时的控制实时的控制系统的行为。

在奈特的事故中，新代码在每个顺序的消息中被执行，而不是一个实时的特性开关，这也就意味这没有一个很好的方法去立刻阻止代码的执行。

这个事故告诉我们，可以在代码中使用有条件的开关和分支来控制系统运行的风险。对于新的特性，奈特的开发人员选择重新利用一个标志位，这个意味着和以前的代码版本有很大的不同。而且正如我们上面描述的实施中的错误，原代码依然运行在其中一台服务器上，这个意味着它可能被意外的触发，然后造成不可预知的混乱的结果。

用有条件的逻辑 去 “分割代码“ 可以让你步步为营推进变更，这样可以实时的控制系统的运行。但是它也使得代码变的难懂，难以控制，以及难以测试。如果逻辑条件很长而且在代码中留有分支，随着事件推移，越多的分支加入，这种情况越来越糟糕。过一段时间后，就没有人能懂和明白这是为什么而准备的以及多个开关结合在一起打开的事后会发生什么。这也是在奈特事件中发生的事情。

在代码中使用特性开关和分支在技术上是一个权衡。团队需要遵守纪律的方法去管理这些代码，确保在不需要条件逻辑和标志时，尽快的移除它们。

显而易见 以及监控/反馈 机制

另一个重要的在DevOps中的实践是确保开发人员随时待命，并且在操作变更时，他们能够提供帮助。如果开发人员在凌晨开发代码时发现一个重大错误提示，这样他们就能回想起是什么导致了这种错误，在开市前对这类事情提供帮助。

对失败的应对

我们已经强调了对失败准备，以及对适时的对事件响应能力的重要性。这个包括知道什么时候该回滚代码，以及它是如何执行的。并且要有一个定义良好的上报机制去向一个人汇报，这个人可以拉下中止开关，让事情脱离控制前，将事情关闭。奈特团队在做出关键决策时花了太长的时间。当系统被关闭时，公司的业务已经受到了很大的影响。