通过系统思考理解业务债务并有序消除

anderszhou 2022.7 v2

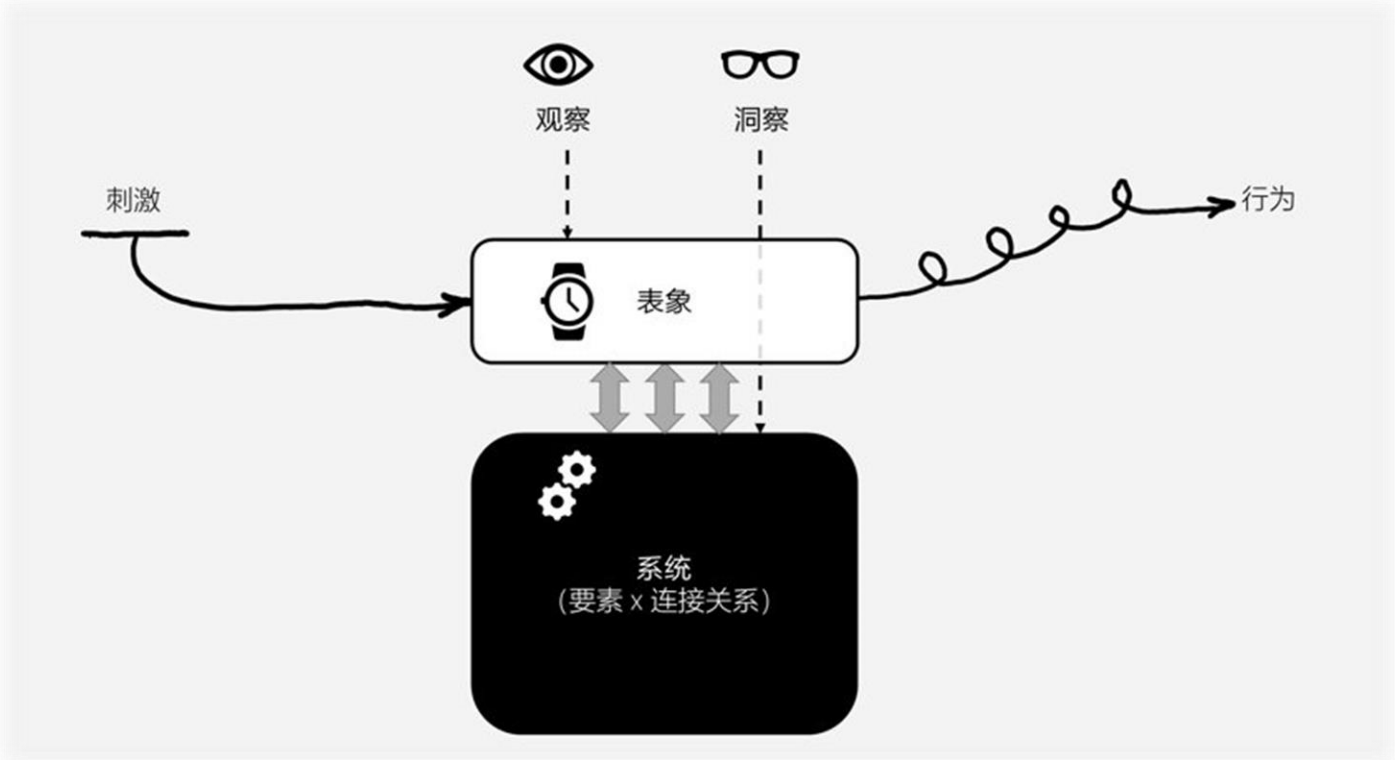
目标

- 了解系统思考方法论
- 了解什么是业务债务
- 了解用系统思考的方法消除业务债务

系统思考

生产软件是一件复杂的事

什么是系统思考



在非线性的世界里，不要用线性的思维模式

功能或者目标

系统 = 要素 X 连接关系；

1.变量

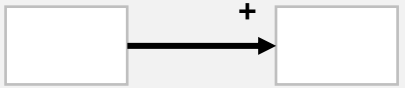
流（入）量

存量

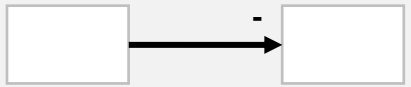
流（出）量

✖

2.因果链



增强的
因果链



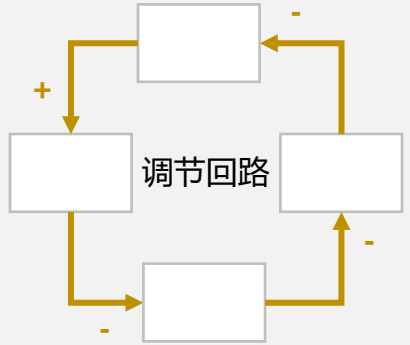
减弱的
因果链

3.增强回路

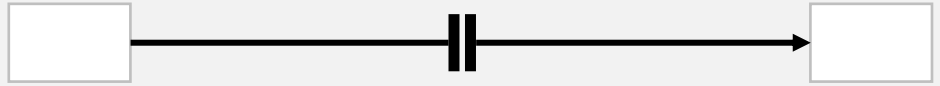
偶数减弱的因果链闭环

4.调节回路

奇数减弱的因果链闭环



5.滞后效应



滞后

雪球为什么不会无限增长（假设雪道无限长）

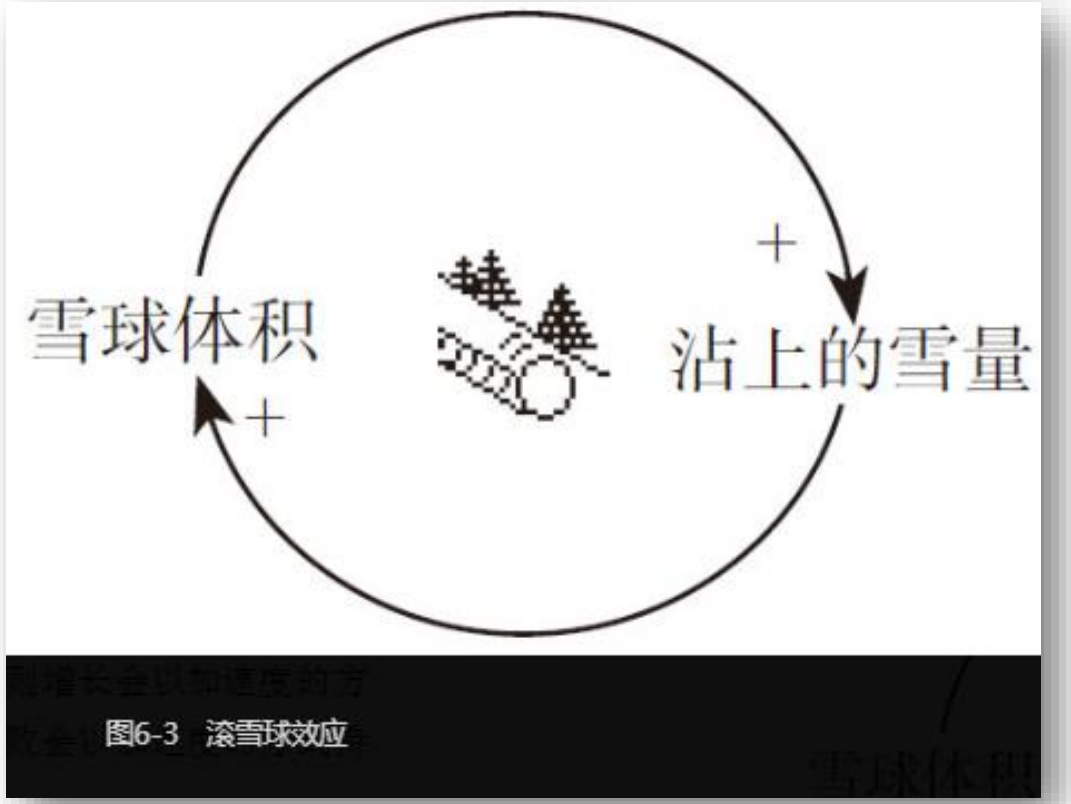


图6-3 滚雪球效应

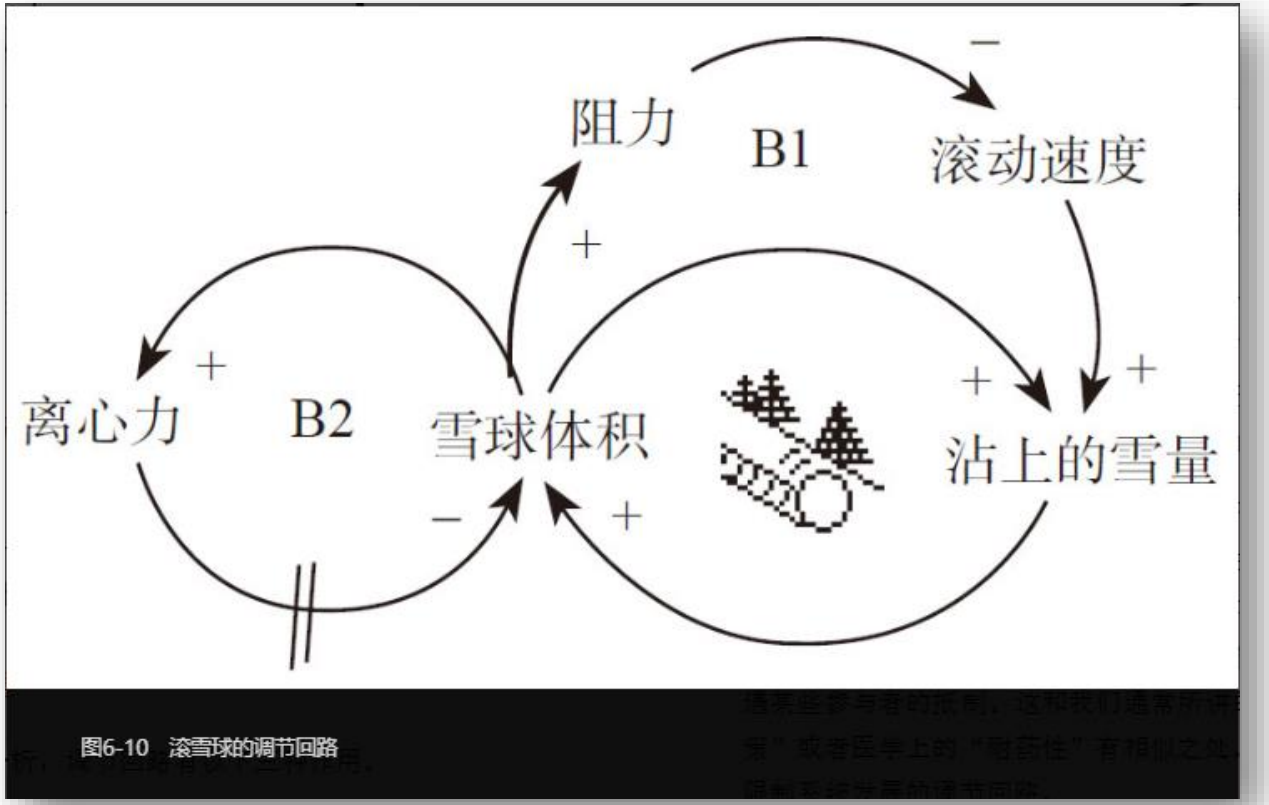
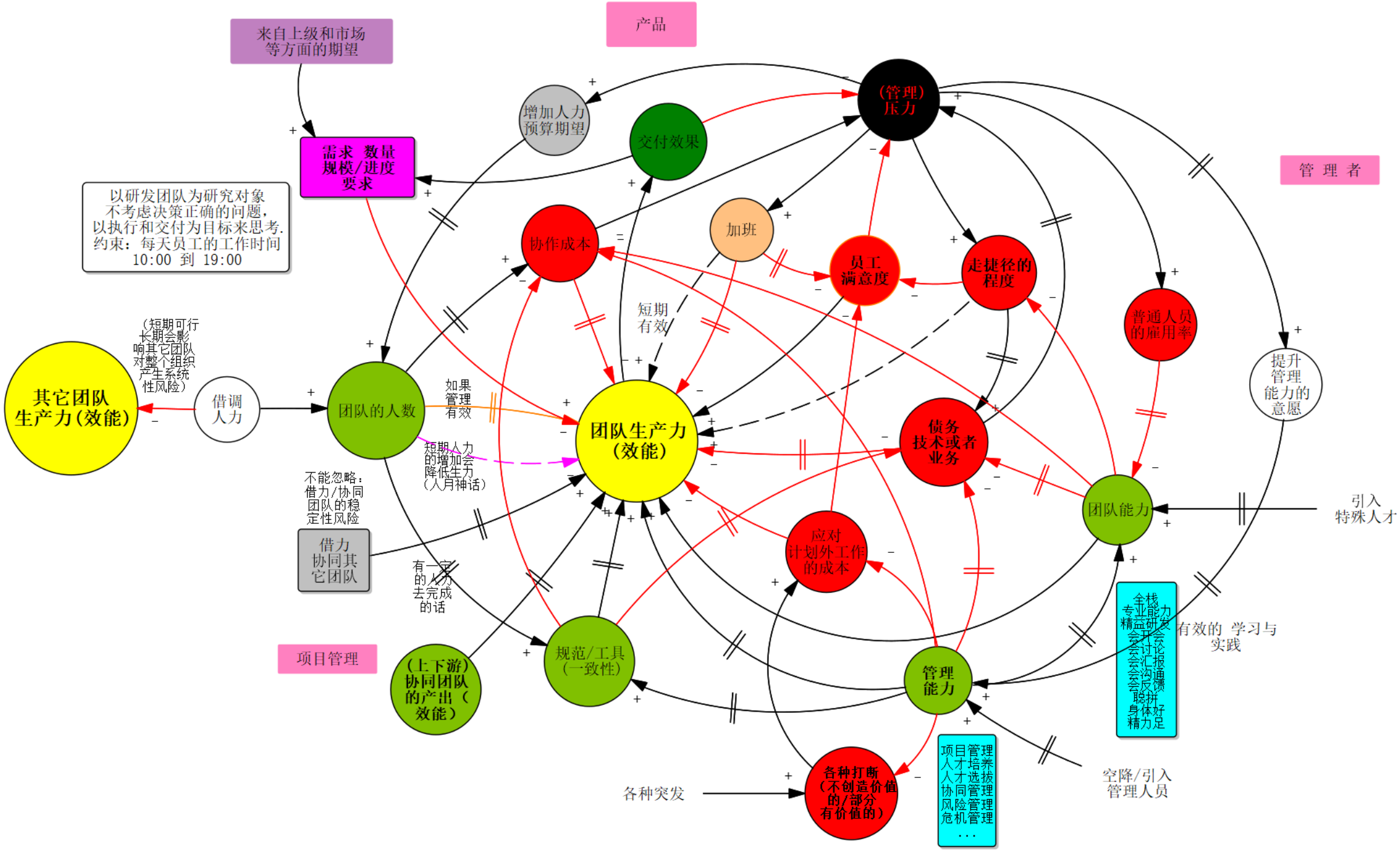


图6-10 滚雪球的调节回路

- 因为调节回路的存在，雪球的体积不会无限制增长

团队生产力因果循环


来自上级和市场等方面的期望
产品
管 理 者
项目管理
需求 数量 规模/进度 要求
以研发团队为研究对象
不考虑决策正确的问题，
以执行和交付为目标来思考.
约束：每天员工的工作时间
10:00 到 19:00
增加人力预算期望
交付效果
(管理)压力
协作成本
加班
员工满意度
走捷径的程度
普通人员的雇用率
提升管理能力的意愿
短期有效
其它团队生产力(效能)
(短期可行
长期会影
响其它团队
对整个组织
产生系统
性风险)
借调人力
团队的人数
如果管理有效
短期人力的增加会降低生力(人月神话)
团队生产力(效能)
债务技术或者业务
团队能力
引入特殊人才
不能忽略：借力/协同团队的稳定性风险
借力协同其它团队
有一定的人力去完成的话
应对计划外工作的成本
(上下游)协同团队的产出(效能)
规范/工具(一致性)
管理能力
有效的 学习与实践
全栈能力
专业能力
精益研发
会开会
会讨论
会汇报
会通馈
会沟通
会反拼
聪明
身体好
精力足
各种突发
各种打断(不创造价值的/部分有价值的)
项目管理
人才培养
人才选拔
协同管理
风险管理
危机管理
...
空降/引入管理人员

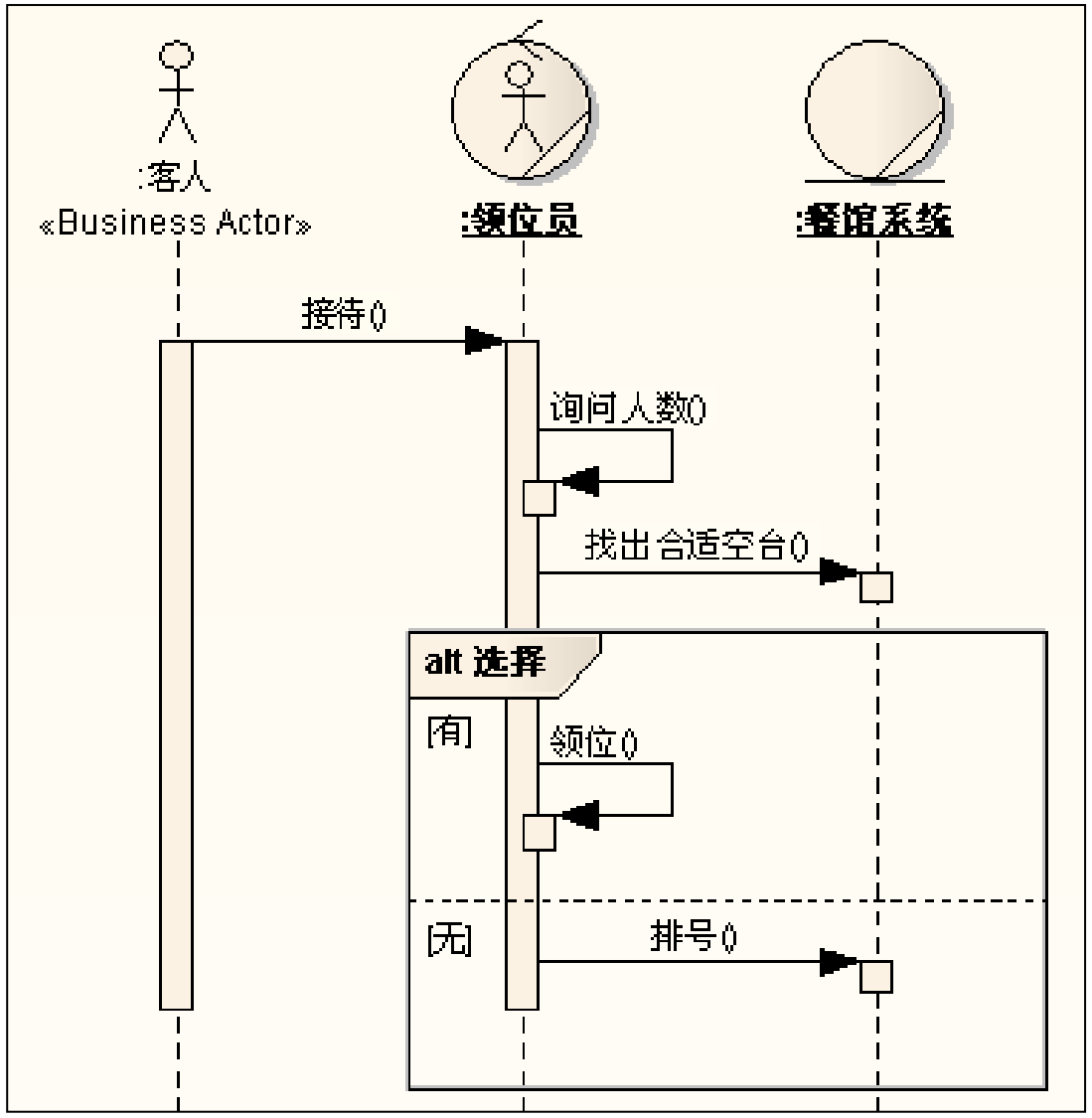
业务债务

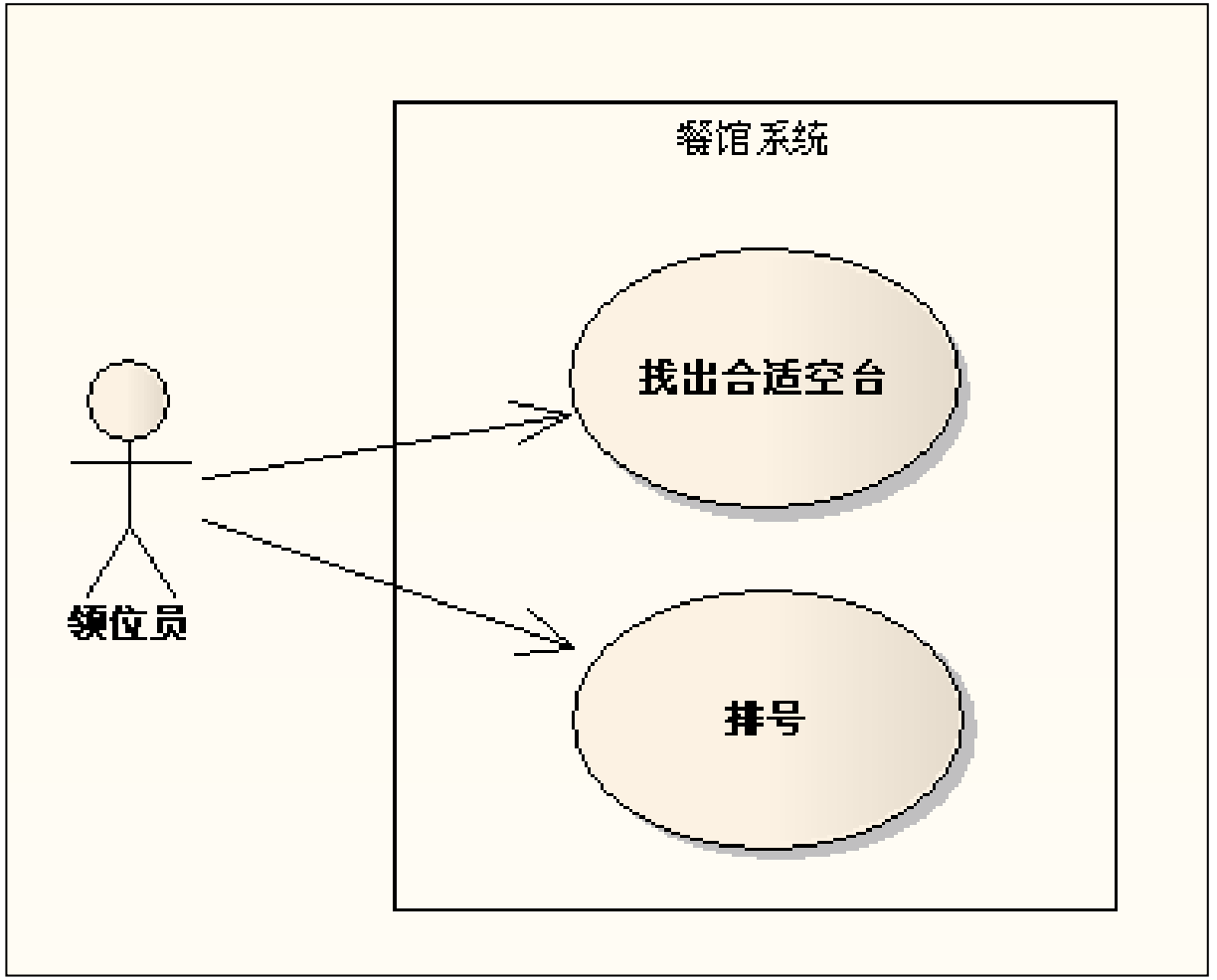
不能忽视的存量

业务=人+系统；系统=功能+非功能

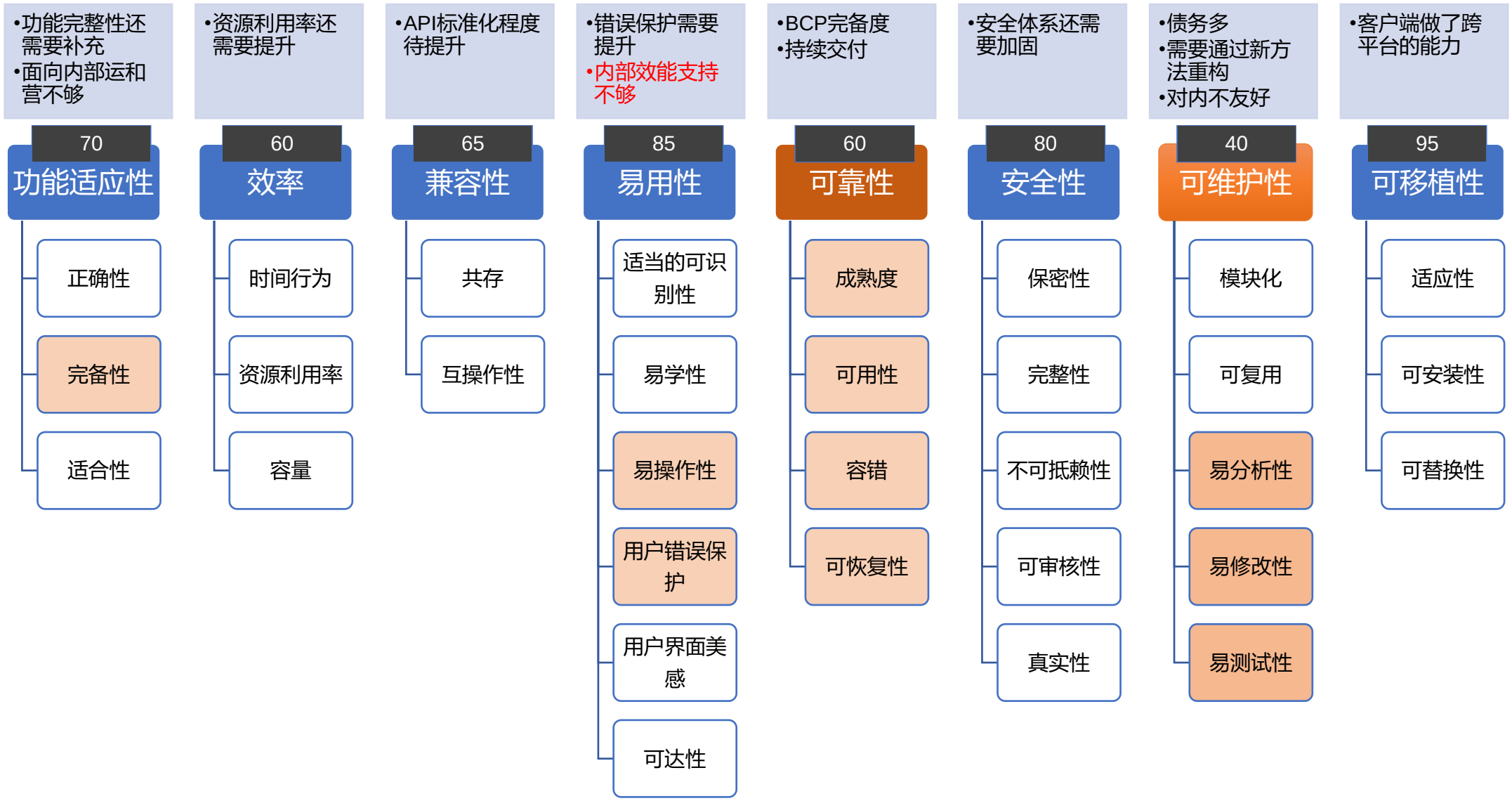
业务

组织内的系统和人通过有序协作给组织外的其它组织提供有价值的服务。

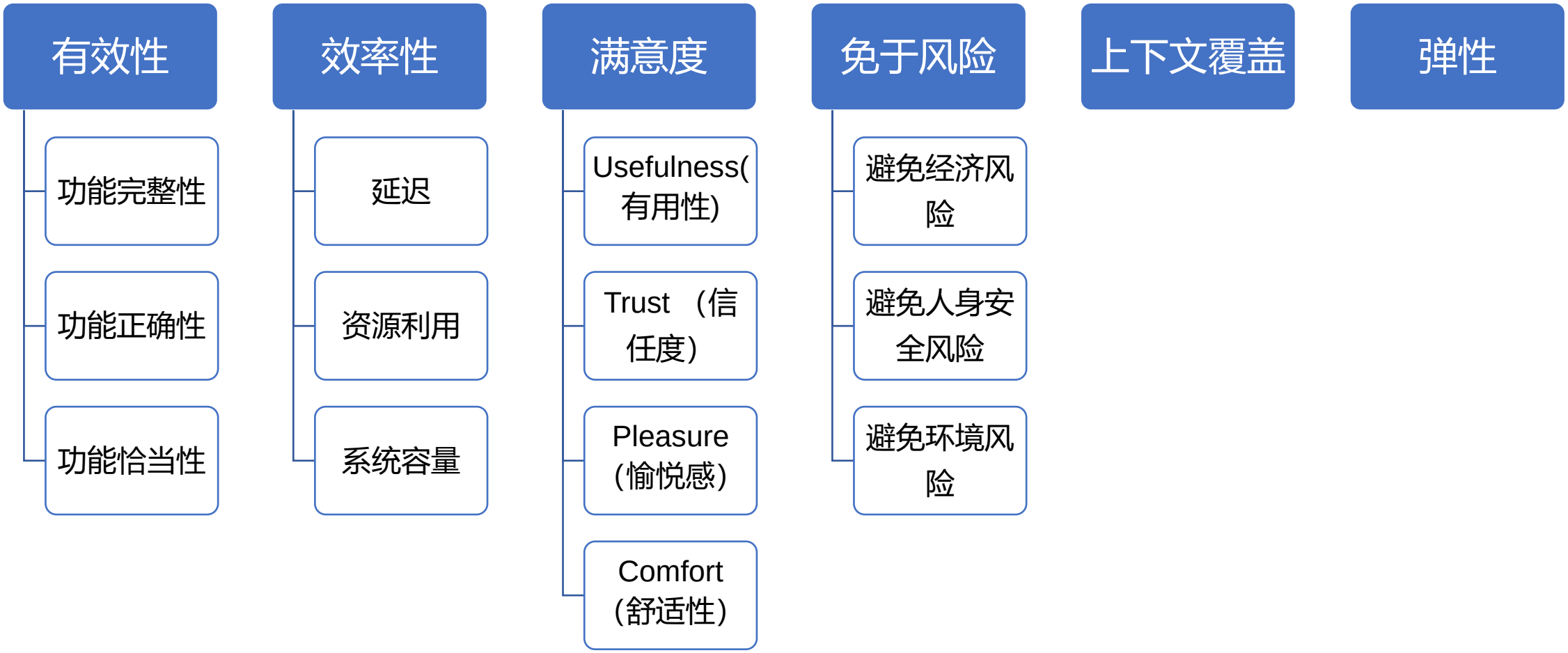




ISO/IEC 25010:2011软件工程-产品质量模型：产品质量



ISO/IEC 25010:2011软件工程-产品质量模型：使用质量



业务债务

定义	特点	危害

• 为获得短期利益而采取的行动所带来的长远**成本**
• 如：
  • 战略债务：为了战略利益（例如首次上市）故意为之，并长期存在。
  • 战术债务：在知情的情况下为了快速收益而产生，适用于短期。
  • 疏忽债务：在不知情的情况由于缺乏知识和意识而产生。
  • 增量债务：定期不慎产生的而导致增量债务。
	• 响应市场
	• 实现快速收益
	• 追求短期利益
		• 长期影响组织的健康运作，带来效能低下，口碑受损，人员发展困难，组织无沉淀等诸多问题
		• 比如：
		  • 降低服务质量
		  • 降低用户体验
		  • 增加运营成本
		  • 降低研发效率
		  • 增加交付成本
		  • 增加交接成本
		  • 增加机器成本

- 这里需要度量

- 故意为之的债务
- 不知道会产生债务

成本

先扛住，再优化
为什么扛住了优化却不见了

目标侵蚀，债务形成的一个来源

系统思考：目标侵蚀

- 为了改变现状，人们通常会设立目标、制订计划，并采取措施去实现目标，这是主动变革的“根本解”
  - 人总是有惰性的，“有志者立长志，无志者常立志”
  - 设定目标固然可以激发人的斗志，也可能只产生“三分钟热度”
- 相对于目标与现状之间的差距所产生的压力而言，后者是“症状解”

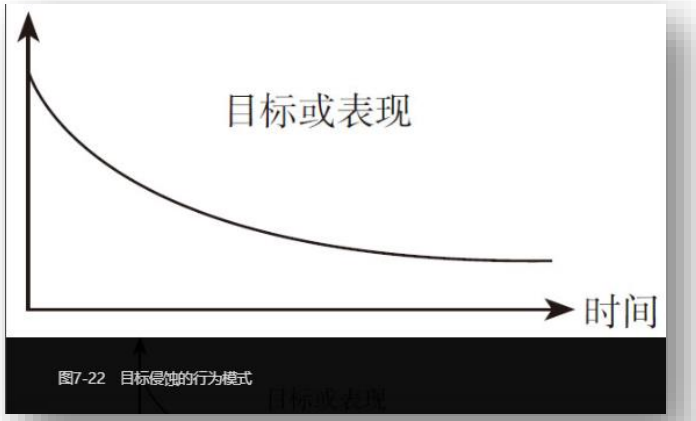


图7-22 目标侵蚀的行为模式

按照人性行为的基本规律“逃避压力”，当目标与现状存在差距时，人们就会产生压力——而这不是人们愿意接受的。
人们就会想办法来纾解压力——办法有两种：要么改变现状，逐渐趋向目标；要么降低目标。

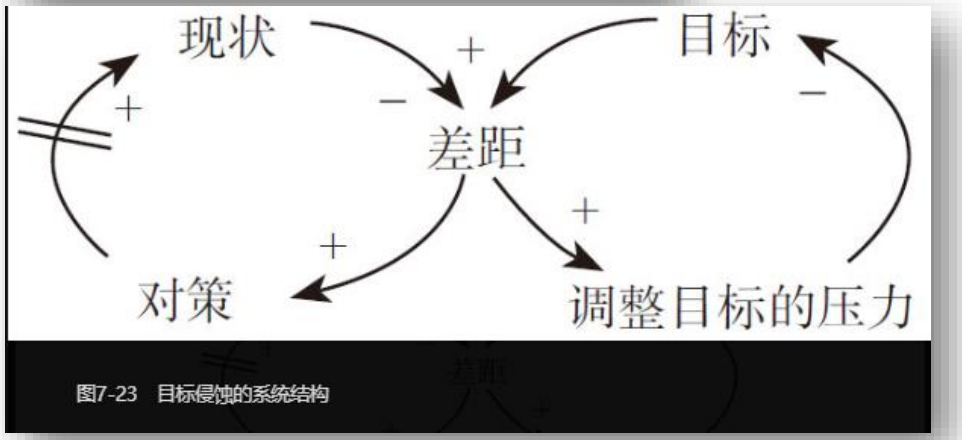


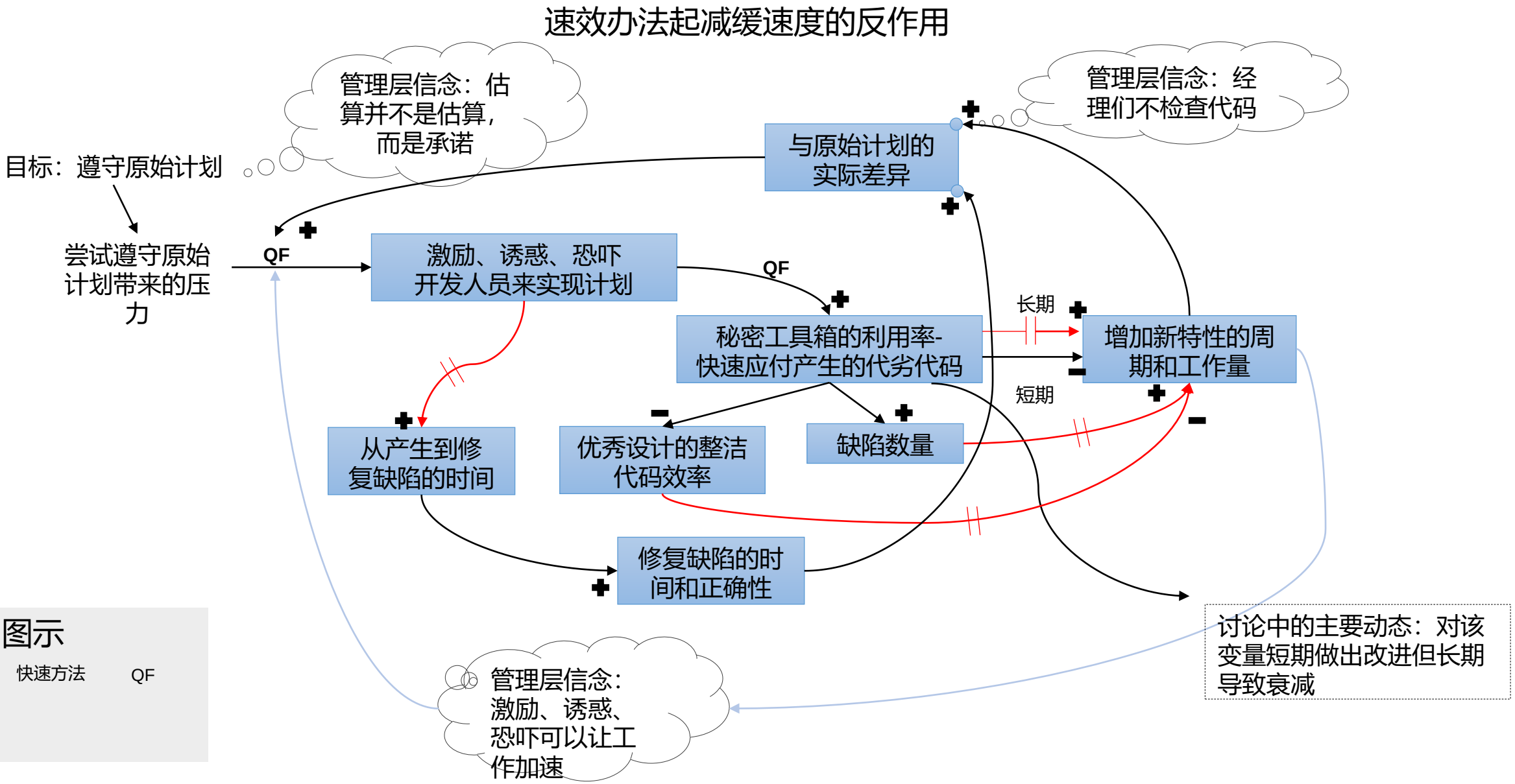
图7-23 目标侵蚀的系统结构

- 根本解
  - 必须关注且只采用“根本解”，即不管绩效如何都要保持一个绝对的标准，自己预期的目标或愿景决不放松；
- 管理
  - 要不断地将目标与过去的最佳标准相对照，而不是和现实的或最差的表现相比。
  - 需要重视这种情况，避免长期对组织产生不好的影响

质量 vs 速度

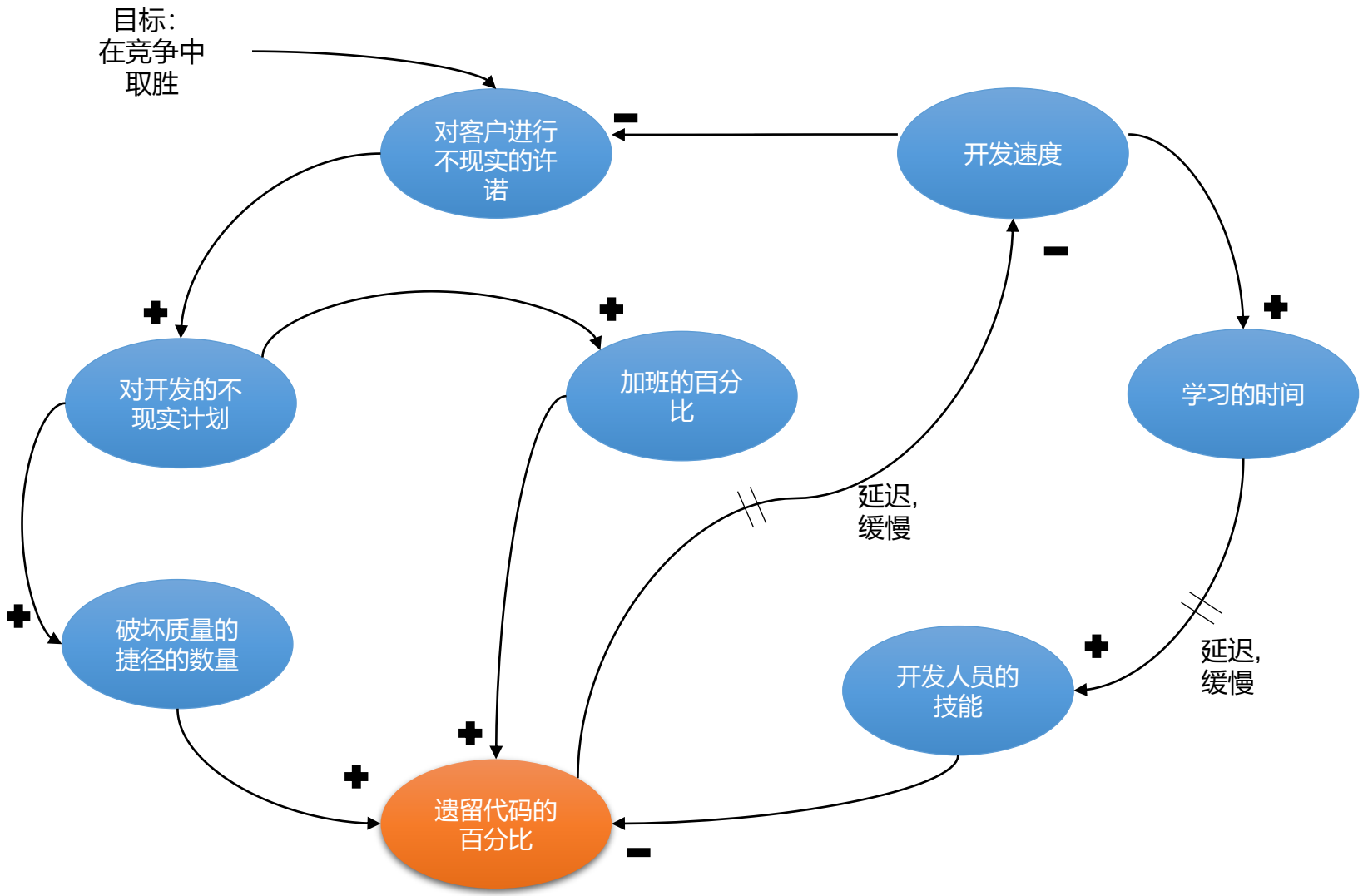
你们怎么看?

欲速则不达： 源《精益和敏捷开发大型应用指南》



欲速则不达不现实期限的承诺，只追求快的因果循环

源《精益和敏捷开发大型应用指南》



欲速则不达，为速形成的债务，谁来买单？

我们在其中的起到什么作用？

功能或者目标

系统 = 要素 X 连接关系；

1.变量

流（入）量

存量

流（出）量

2.因果链

增强的
因果链

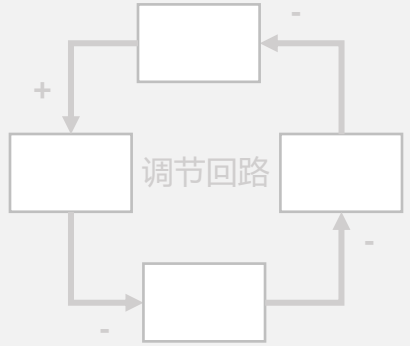
减弱的
因果链

3.增强回路

偶数减弱的因果链闭环

4.调节回路

奇数减弱的因果链闭环



5.滞后效应

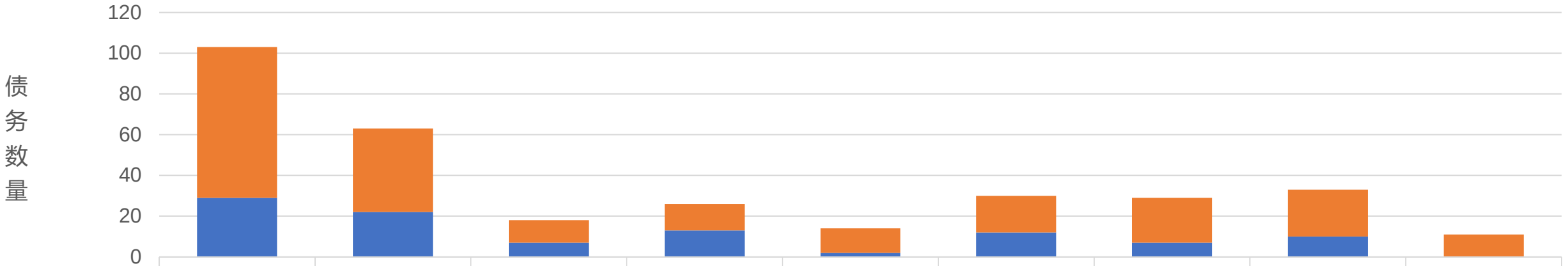
滞后

举例，各个不同阶段的一些典型债务

人肉功能	需求	设计	实现/	测试	运营	流程	基础设施
•业务功能不完备，不完善 •开发人员要参与到业务流程中 •解决投诉 •定位问题 •...	•都在人脑中，人可能已经不在了 •风格各异，难以理解，文法不通（一句话） •Tapd适合作任务管理，不适合作需求管理 •因为啥也没有，工作无法交接，也不用交接， •对异常的关注度少（70%的工作量） •需求只满足部分涉众的利益，涉众覆盖不全 •低ROI的需求占用资源多	•为了快没有设计，一把梭使用改动小的「简单」设计 •没有设计文档，好吓人 •先这么搞，以后再说（还有以后吗） •解决质量需求需要掌握很多架构模式 •质量需求得不到很好的支持，缺少对异常和安全的考虑（读书少，没有时间，项目紧） •历史系统的不合理设计的影响 •康威定律影响下的畸形设计 •业务耦合，到处都是坑 •可测试性基本不会考虑	•需求传递，逻辑问题 •代码管理 •编码 •外部依赖 •不遵守规范，打折扣 •漏实现，改错（偶发问题） •配置文件满天飞，隐患重重 •硬编码 •“各种简单实现“ •缺少积累，代码全部人工码…	•先这样的让用户测试，永远的beta（别人的口号成了我们的借口） •**不知道测试啥** •覆盖不全 •自动化测试不够（基本没有）	•工具缺少，投入不够，处理基本靠人肉手工处理 •人肉运营，熟悉，习惯了 •监控体系不完备 •BCP（业务连续性计划）得不到贯彻执行，忽略小概率事件[3.23南汇断网事件]，不相信墨菲定律） •部署设计缺陷 •告警缺失，上报不够（要加上旷日持久） •遗留系统多	•不好的流程 •没有流程 •执行不到位的流程 •囧：形同虚设的流程	•基础设施简陋 •工具建设滞后 •人肉流程多，风险大

举例：业务债务

债务看板，债务可视化



	设计缺陷	设计过时	安全风险	低质量代码	低质量数据上报	测试保障缺失	运营建设缺失	知识沉淀缺失	工作交接不充分
遗留债务	74	41	11	13	12	18	22	23	11
2019偿还债务	29	22	7	13	2	12	7	10	0

	设计缺陷	设计过时	安全风险	低质量代码	低质量数据上报	测试保障缺失	运营建设缺失	知识沉淀缺失	工作交接不充分
2019偿还的重要债务工作项	• 条带读故障触发屏蔽支付 • 实名中去除对财付通的重复调用 • 付款码和用户认证解耦 • 全局配置滥用优化 30%	• 支持 IPV6 • 32位迁移64位系统 • appplatform迁移 svrkit • memcached迁移 wcache/ttlkv • 无效代码/配置清理 40%	• 提现审计完善 • 业务风控补全	• 渠道收拢代码重构 • 大单分布式事务框架重构 • 代码依赖优化，大库拆小 30% • 代码CR委员会机制落地	• 确定数据上报规范 • 优化部分上报	• 测试流水线建设 • APP支付用例完善 • 993环境治理 90% • 核心交易系统补充大量集成测试用例 • 集成测试及单元测试知识、工具普及，日常落地 80%	• 10+ 运营能力建设	• 领域模型补齐 • 对外契约及业务场景 / 接口使用文档补充	• 无 （人员流动后，暴露出交接问题）
遗留的重要债务工作项	• 继续完成全局配置滥用优化 • 支付生命线共存亡 • 事件中心业务定时器用法消除 • CGI 严格契约检查 • 钱包读扩散优化	• 支付彻底分区 • 数据自动入TDW • 收银台支付模式事件化驱动改造 • 持续清理无效代码/配置 • 大单proxy 从 logicsvr 切换为 svrkit	• 票据全面覆盖 • 模块认证全开	• 代码依赖持续优化 • 消除因实现随意带来的多余热加载 • 支付基础代码重构关注点分离代码架构 *- 可扩展个性业务逻辑* *- 可复用公共业务逻辑* *- 业务流程模式驱动框架* *- 基础设施及可用性*	• 继续完成业务基础数据优化建设	• 全链路容量压测 • 993环境治理完成 • 代码提交主干前质量红线建设	• 继续补完运营能力建设	• 各系统设计沉淀 严禁新增，核心业务补充完整	• 交接中的问题收敛 曾经接收系统的负责人，必须熟悉相关代码，无沉淀的补沉淀并登记债务

即便有债务，组织表现出很强的适应性，为什么

温水煮青蛙，我们感觉很良好呀

系统思考：延时

系统思考：成长上限

一个过程开始加速增长，然后成长开始趋缓（系统里面的人往往未察觉），逐渐停滞，或反复震荡；甚至可能逆转，加速下滑并崩溃。

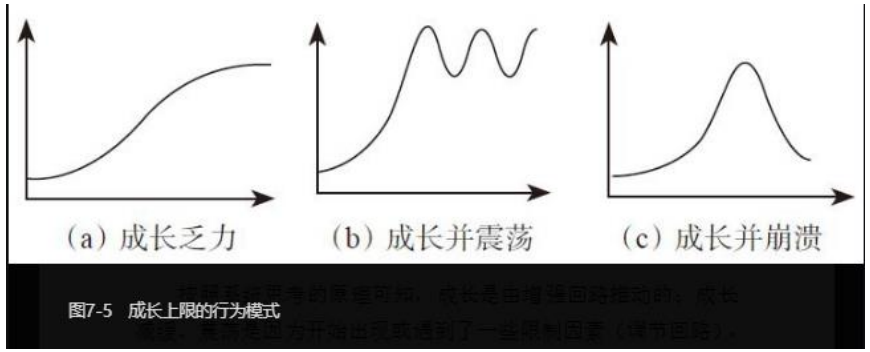


图7-5 成长上限的行为模式

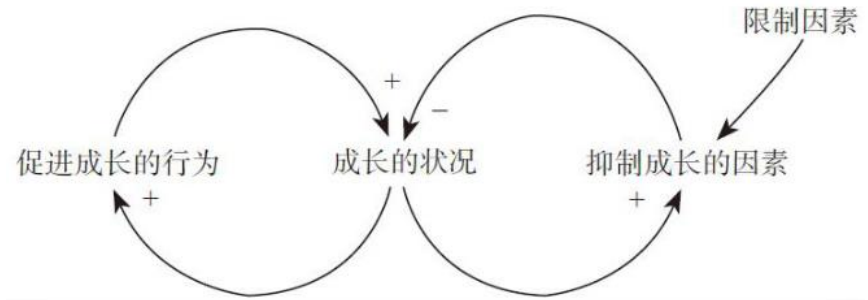


图7-6 成长上限的系统结构

应对

- 第一，预防胜于救火，要预见到可能存在的障碍因素以及哪一项因素居于主导地位，即将开始逐渐发挥作用。周密设计，提前采取措施解除，“防患于未然”。
- 第二，若系统已经处于成长上限的结构时，不要强制推动“成长”，应设法去除或减弱限制性因素的影响，成长就会开始启动。

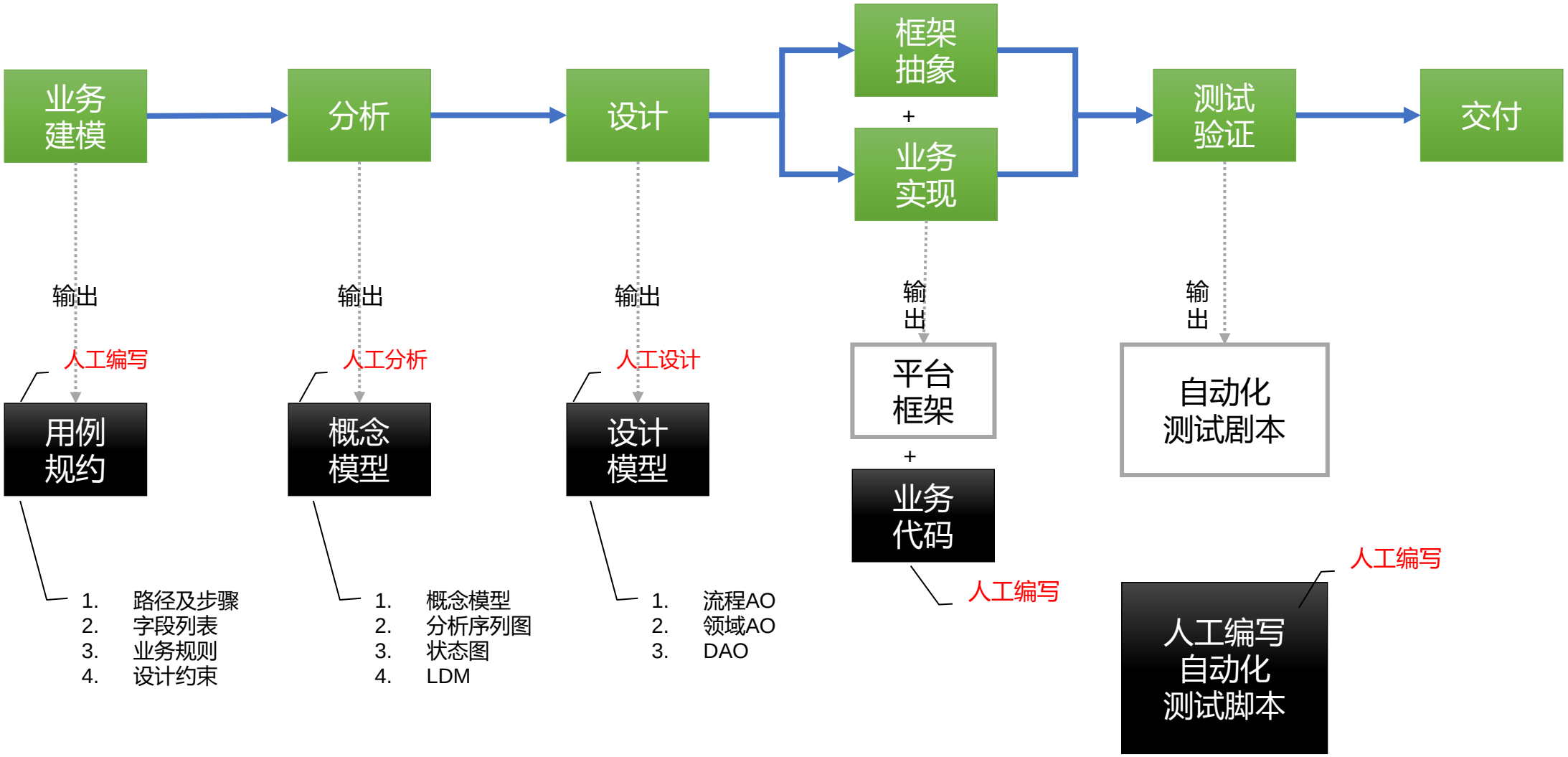
预警信号

- 一开始：“没有什么好担心的。我们正在快速成长。”
- 稍后：“确实是有一些问题，但我们能够应对。”
- 再后来：“我们努力地跑，但却好像一直在原地踏步。”或者“我们越用力推，系统的反弹力量越大。”

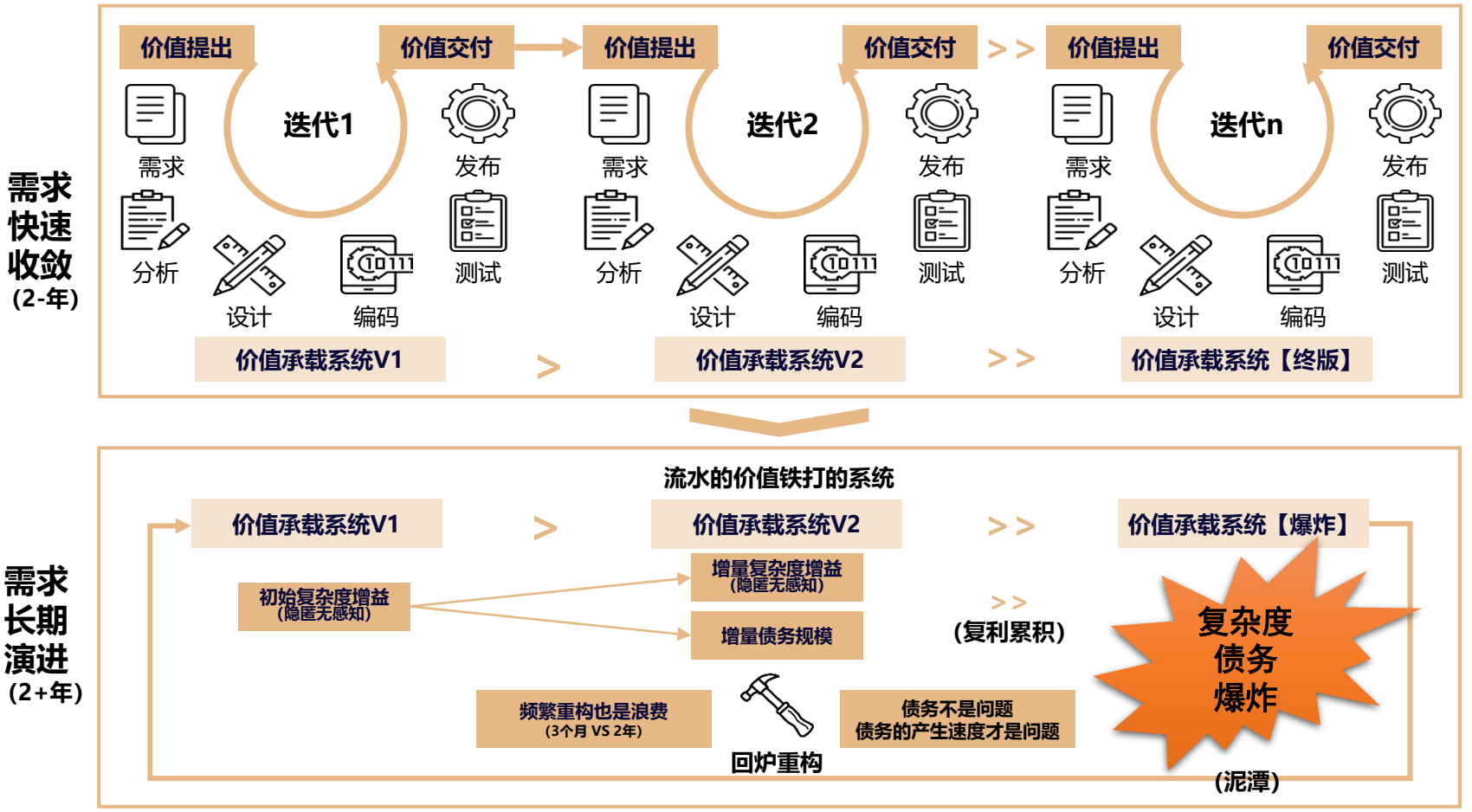
软件生产仍然是手工业

举例，大量的人工处理过程导致效能低下

- 大量为人工处理，规范性差，人难以坚持和稳定输出



持续性难题



为什么价值交付慢?

迭代周期长

迭代WIP多	每一步耗时多	衔接差等待多
• 【流】研发没闲着但每件事都拖长了交付时间 • 【人】人力不开源，全栈性人力不足	• 【器】特别是编码、测试、发布耗时多 • 【人】规范不落地返工多 • 【人】框架、组件、库不知道不熟悉不熟练	• 【管】统筹不足缺乏提前量 • 【流】流程冗长 • 【器】工具繁复难用缺乏集成缺乏自动化

对策

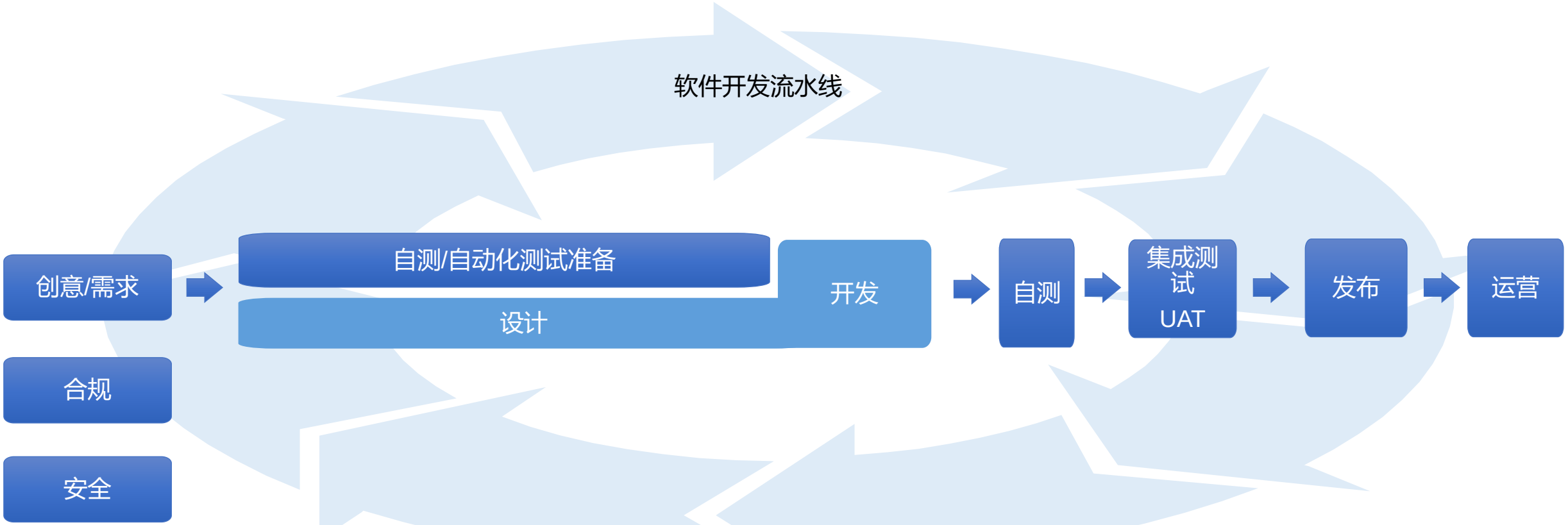
研发流程优化	编码效能工具	研发管理提升
打破前后边界	测试效能工具	研发流程优化
	发布监控优化	工具整合集成
	规范培训	
	框架、组件培训	
	简化省略分析设计	

为什么价值交付越来越慢慢?

- 复杂度累积速度
  - 初始的复杂度增益
    - 初版的分析设计质量 (关键)
      - 缺乏意识 → 意识宣导
      - 不知道方法 → 分析设计技能培训
      - 人的能力 → 招聘和训练
      - 领域知识的可学习性 → 知识的结构化沉淀
  - 迭代的复杂度增益
    - 迭代的分析设计质量
- 债务的增量速度
  - 不彻底的重构
  - 迭代的频率和规模

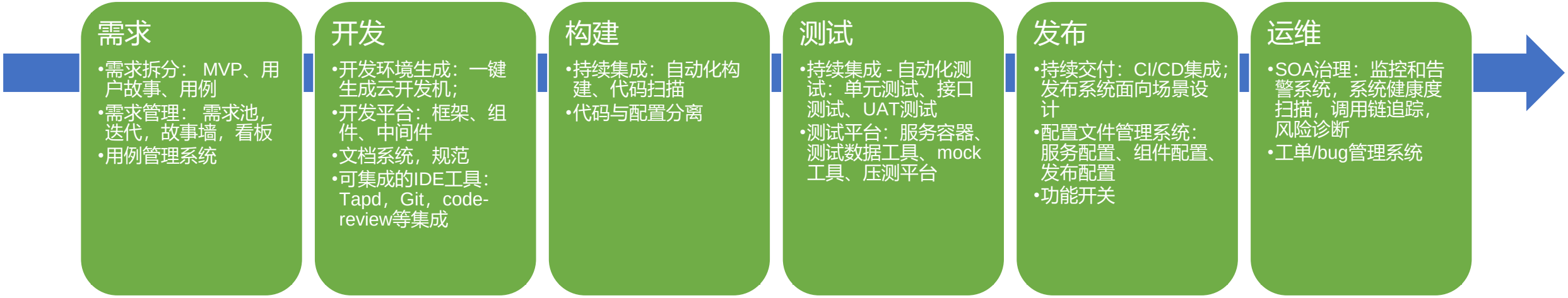
对策

- 快速暴露解决痛点的流程机制
- 鼓励问题暴露解决的管理机制
- 人员培训管理机制
- 遗留债务的治理机制



“别提敏捷，只解决问题！”

完善工具，持续交付 – 构建全流程的工具链和流水线



- Tapd需求管理
- 用例管理系统
- BCP管理系统
- Tapd老板需求集成
- 看板和tapd打通
- Tapd项目管理视图优化
- Tapd与测试、发布等流程集成
- Tapd的数据报表和集成
- Tapd需求状态机定制化

- 开发机生成与管理
- Vim/VScode
- Git代码管理系统
- Tapd与IDE集成
- Git与Tapd关联
- 设计评审管理系统
- 开发资产注册系统
- 代码review跟踪系统

- Blade->Bazel编译
- 蓝盾流水线
- 自动化构建流水线
- 脚本的构建
- 静态扫描和管理
- 动态扫描和管理
- 安全扫描和管理

- UAT测试平台
- 单元测试工具
- 接口测试工具
- 单元测试（试点中）
- UAT测试（试点中）
- 自动化测试流水线
- 测试与需求管理集成
- 环境即代码
- 测试环境监控与事件管理

- 微信发布系统；安全性、易用性待提升
- 配置文件管理系统（建设中）
- 发布DevOps流水线
- 灰度发布/条带发布
- 功能开关
- 面向场景发布流程
- 发布治理(超长单等)

- 监控系统
- 客服工单系统
- 升单管理系统
- 用户反馈ifeedback
- SOA治理系统
- 健康度检查系统
- 可用性检测与记录
- 容量管理与监控
- 混沌系统
- 复盘/故障与tapd打通

消除人脑/零散记录 → 消除人工传递 → 消除人工信息对齐 → 消除人工重复

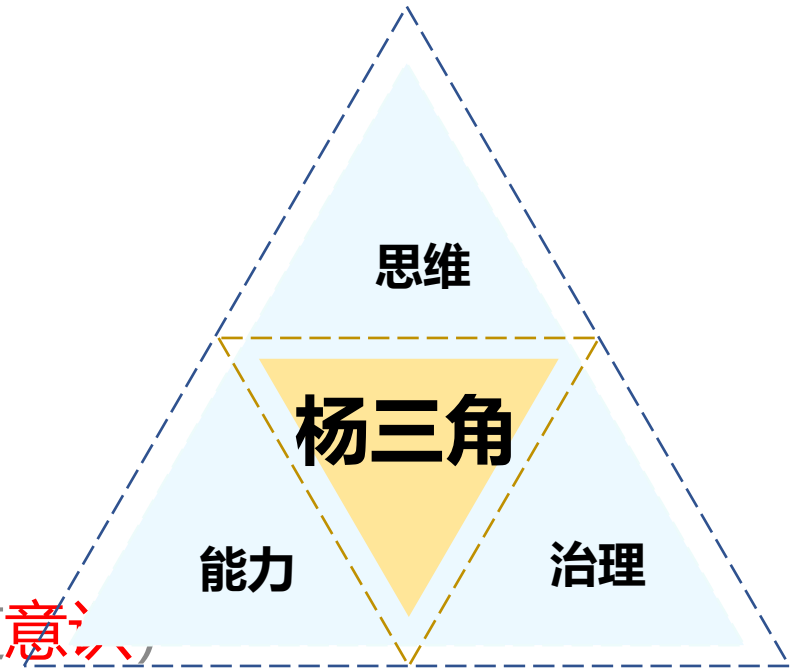
债务多少合适呢?

如人借钱，冷暖自知

人力资源太多的情况，很难作出彻底的改变

偿还债务

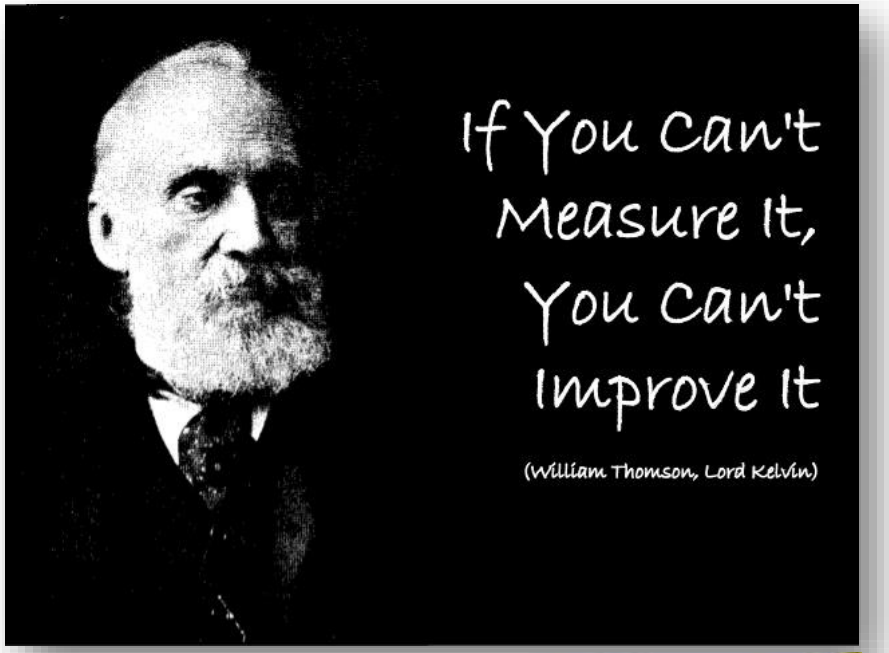
根本解:干干净净，边重构边生活（海量服务之道意义）



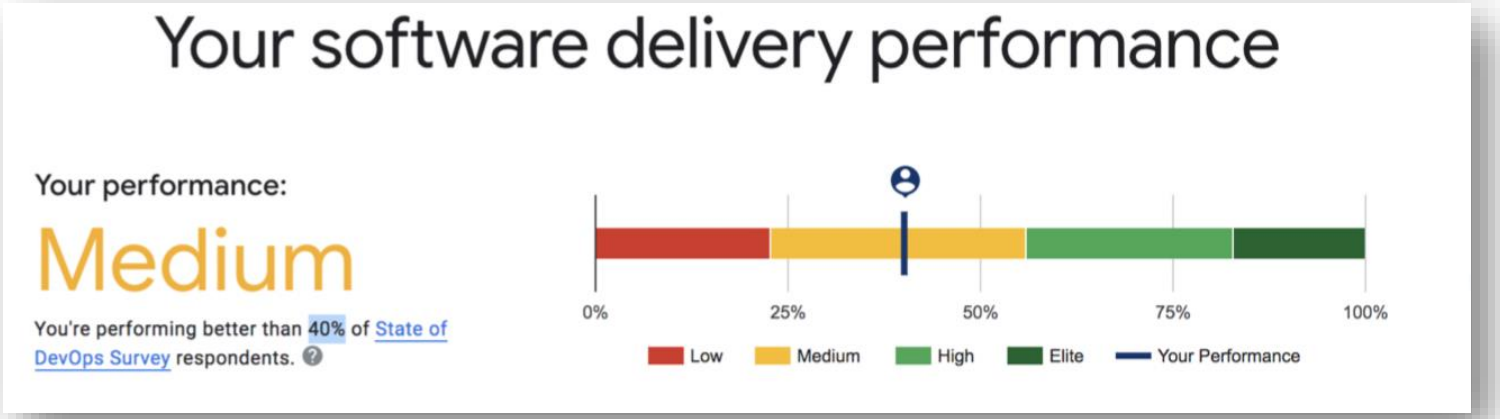
误区：以人为本的治理

要对研发的效能进行度量，进而指导和牵引团队进行债务的消除

对研发的效能进行度量

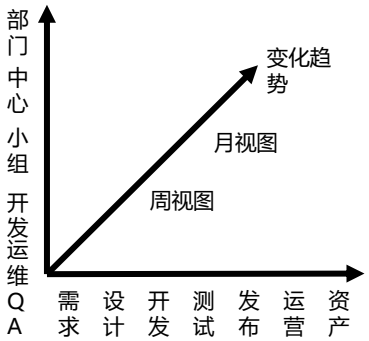


- DORA标准
  - Lead Time、发布频次、恢复时间、变更失败比例



对研发的效能进行度量

- 掌握价值流动全过程
- 价值交付指标
  - 业务特性
  - 成本&改善
- 过程指标维度
  - 质量
  - 速度



- 管理者工作台
  - 可视化管理
  - 透明、及时、一目了然

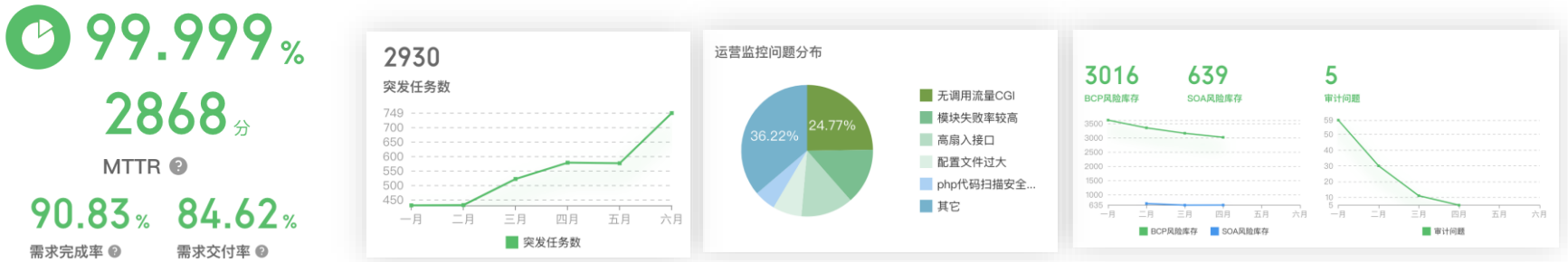
价值交付指标

需求交付周期	需求交付率	复盘率
需求成功率	需求完成率	资源利用率	需求吞吐量

过程指标

开发	测试	部署	发布	运营
任务完成率	BUG单数量	部署成功率	变更失败率	平均故障恢复时长MTTR
突发任务数	集成测试通过率	部署时长	变更次数	故障数量
开发时长	集成测试时长	部署次数	回退次数	平均无故障时间MTBF
函数圈复杂度			发布时长	
代码缺陷数			门禁拦截次数	

管理者工作台现已上线投入使用！助力管理效能提效50%



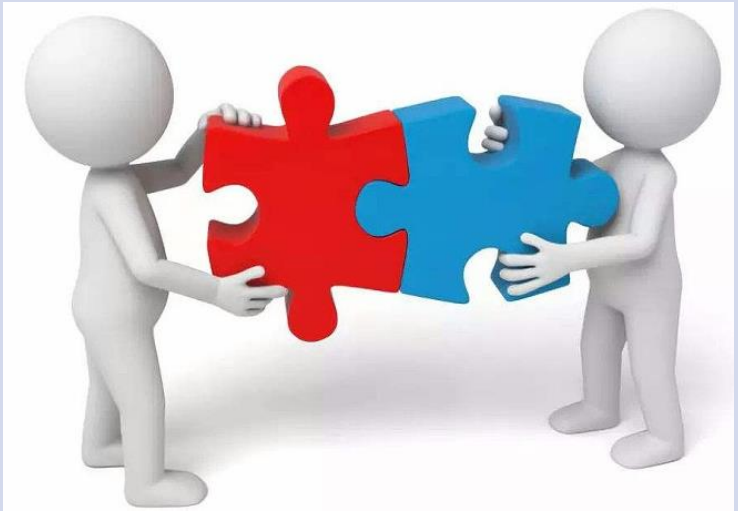
如何修复线上的 Bug?
Hard Code 快速解决问题，但是一定要留给开发解决技术债的时间。 #每日技术# #娱乐#

不要把做业务和还债这事对立，只有选择

建议：调整目标，优化管理机制，分期付款，人员冗余

调整组织目标

- 把效能写入“组织”的目标
  - 质量
  - 速度 （不仅仅只有速度）



改规则

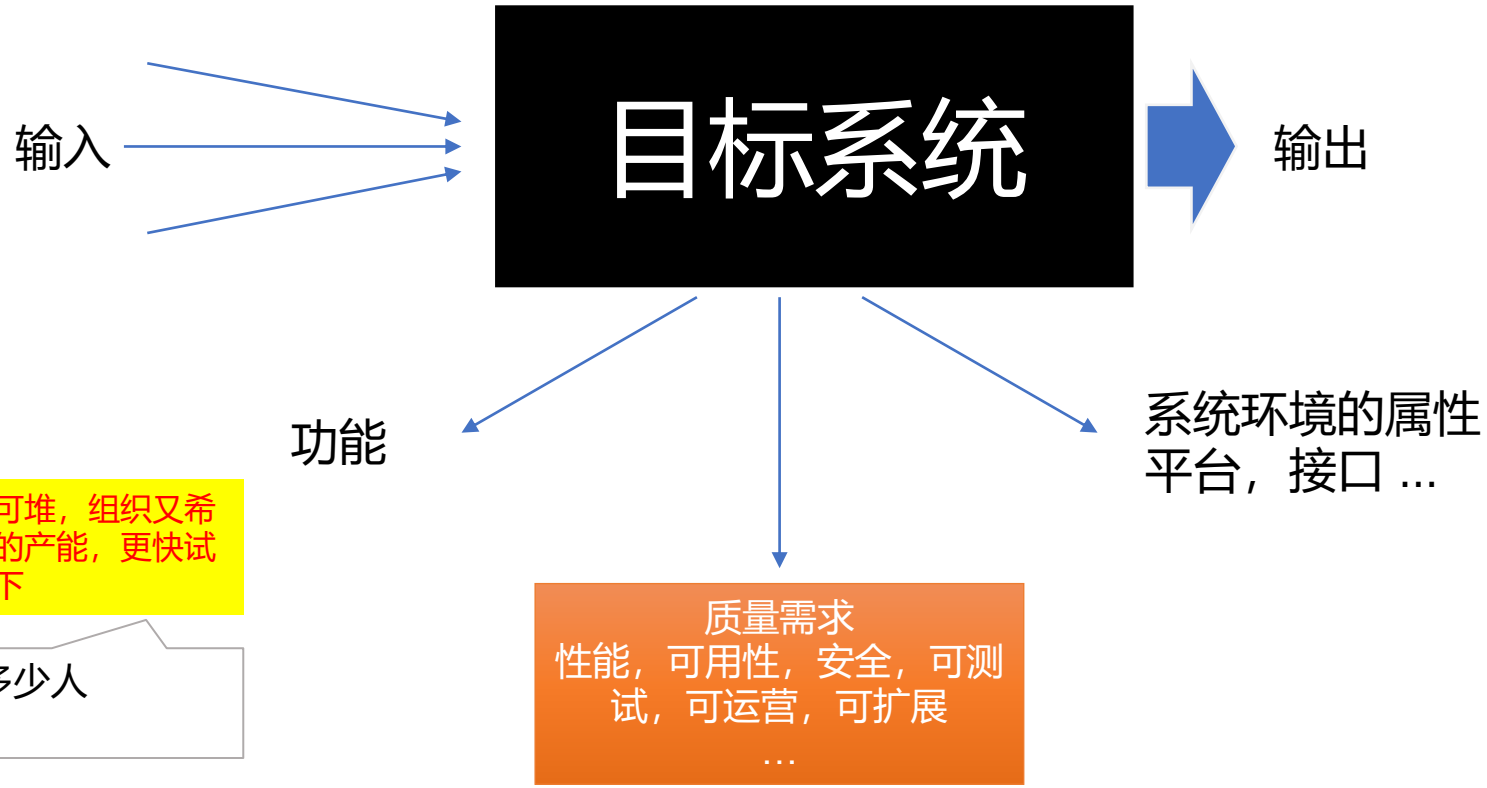
- 分期付款
  - 和产品达成共识，解决“大石头”的同时干掉“小石头”
  - 多批次，小数量迭代偿还债务
  - 慢就是快
- 专职人员
  - 一定要有清洁工（要高级的那种）
- 管理机制
  - 处理脏活也是贡献价值，平衡好新特性和债务的偿还
  - 做好了才能去做新任务，新人做新业务
  - 还债是大家的事，人人都有份

- 偿还存量债务

- 业务都很急，木有办法还债咋办?
  - 都? 有分析过吗?
  - 有没有全流程去看?
  - 团队人力的配置是有问题?
  - 是否上向进行过沟通.....

业务开发只聚焦业务?

功能需求，补充约束 [字段列表，业务规则， 质量需求（质量需求），设计约束]



- 在没有人可堆，组织又希望有更大的产能，更快试错的情况下

- 多久，多少人
- 质量

提升人才密度

尽可能一次性做对，做好，减少债务的形成

建议，引入精益研发，还债也是有价值的

面向价值交付，杜绝浪费，持续改善，带来生产质量和效率（效能）的提升

精益实践框架

工具： TAPD、iWiki、企业微信机器人

关键字： 价值 共识 高质量 低成本 0浪费 透明 持续改善

1)迭代对齐需求价值和OKR

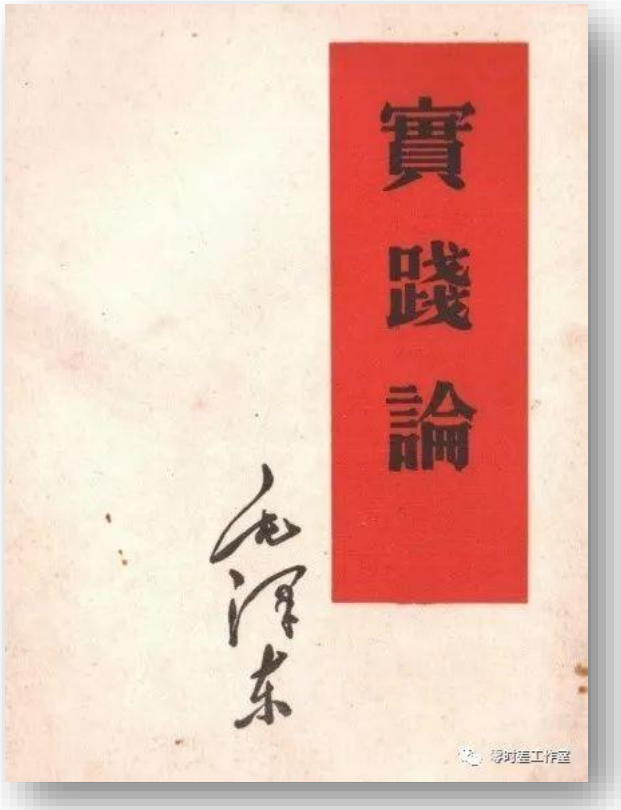
2) 价值流看板，透明价值流动过程

3)拆解任务，小批次拉动生产

使用可靠的系统，低成本、无缝的让价值流动起来

面对增量债务，GO see 管理者要在一线帮助大家解决

管理者的二手信息陷阱



实践是检验真理的唯一标准

精益研发 –复盘和现场改善文化

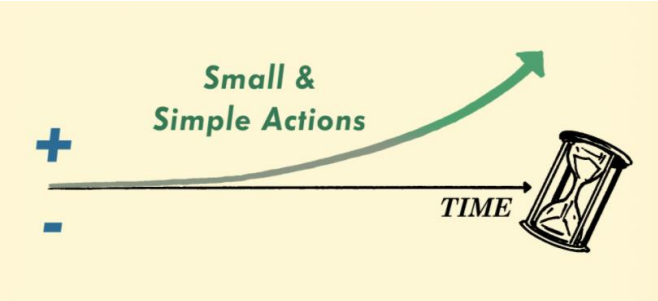
自主发现了972个Lowlight并及时改进(1~5月)，复盘率100%

迭代复盘、项目复盘，可以短暂停下来反思改善的好机会

现场改善，通过观察现场和与现场人员沟通，了解第一手信息，保持增值的，及时去掉不增值的浪费和消除问题。

用脚去“看”

通过简单的小动作
持续变好的效益



复盘Log， 我们的改善和成长记录

团队反馈：

是的，战斗力变强了

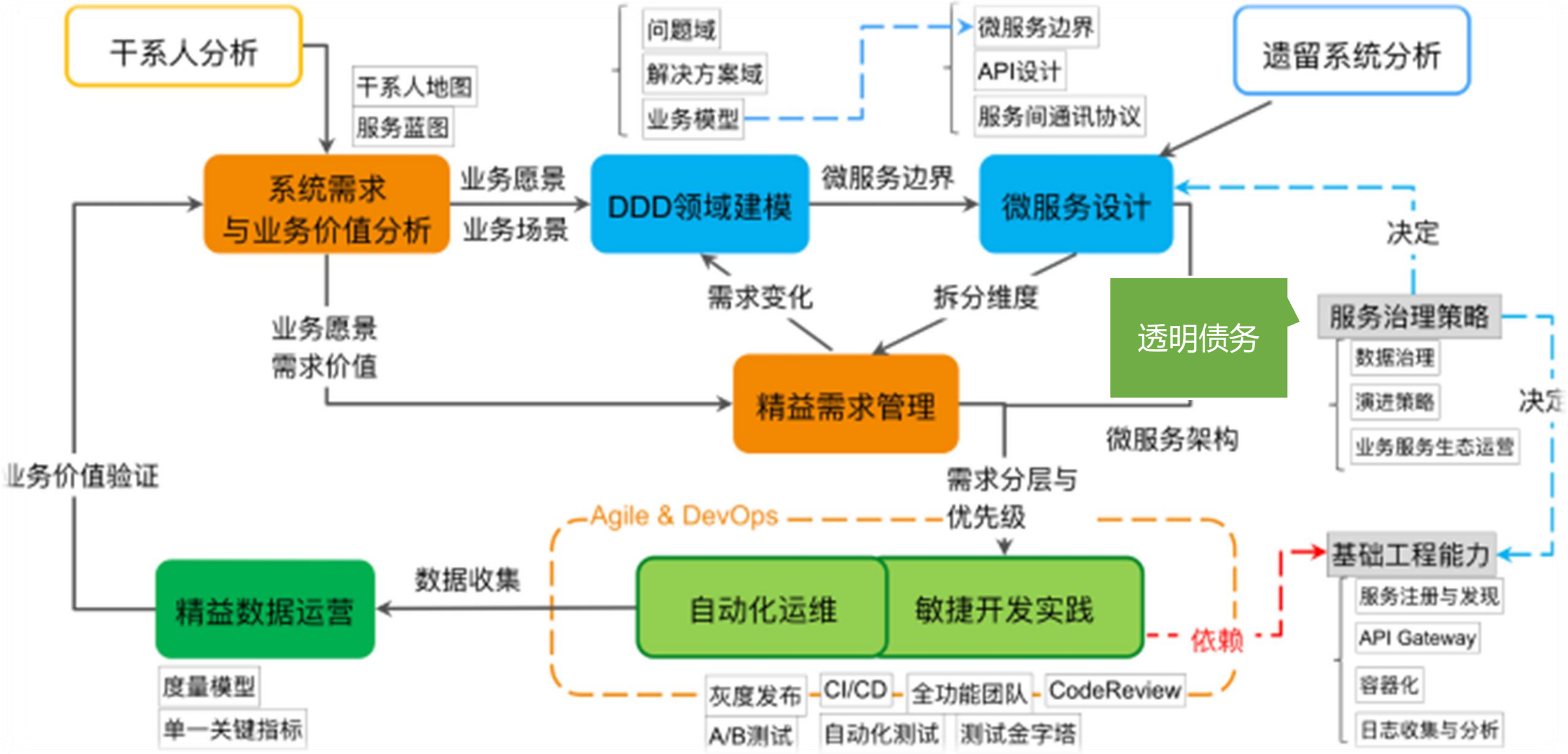
积极性调动起来了 管理投入减少了

挺好的，产品投入了进展就好了。开始有商户使用了。

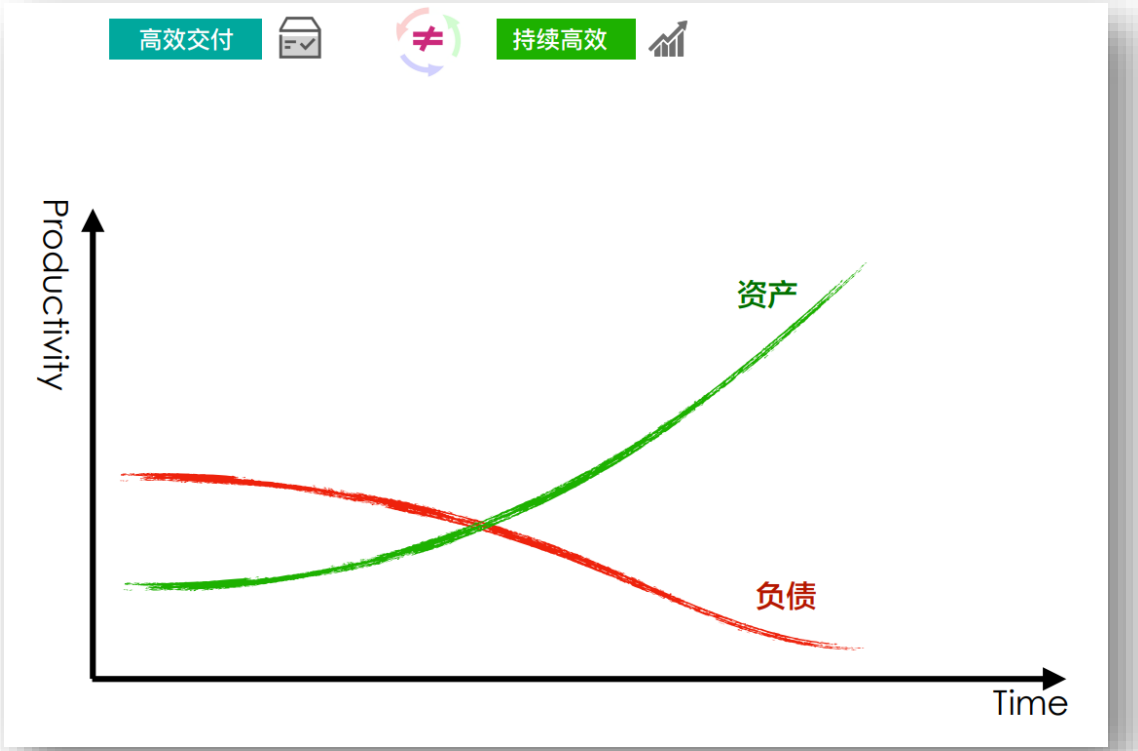
用这个精益迭代，最大的感受是目标感，节奏感增强很多。需要继续提升的是需求的规划和拆分，复盘中如何有效解决lowlight问题。

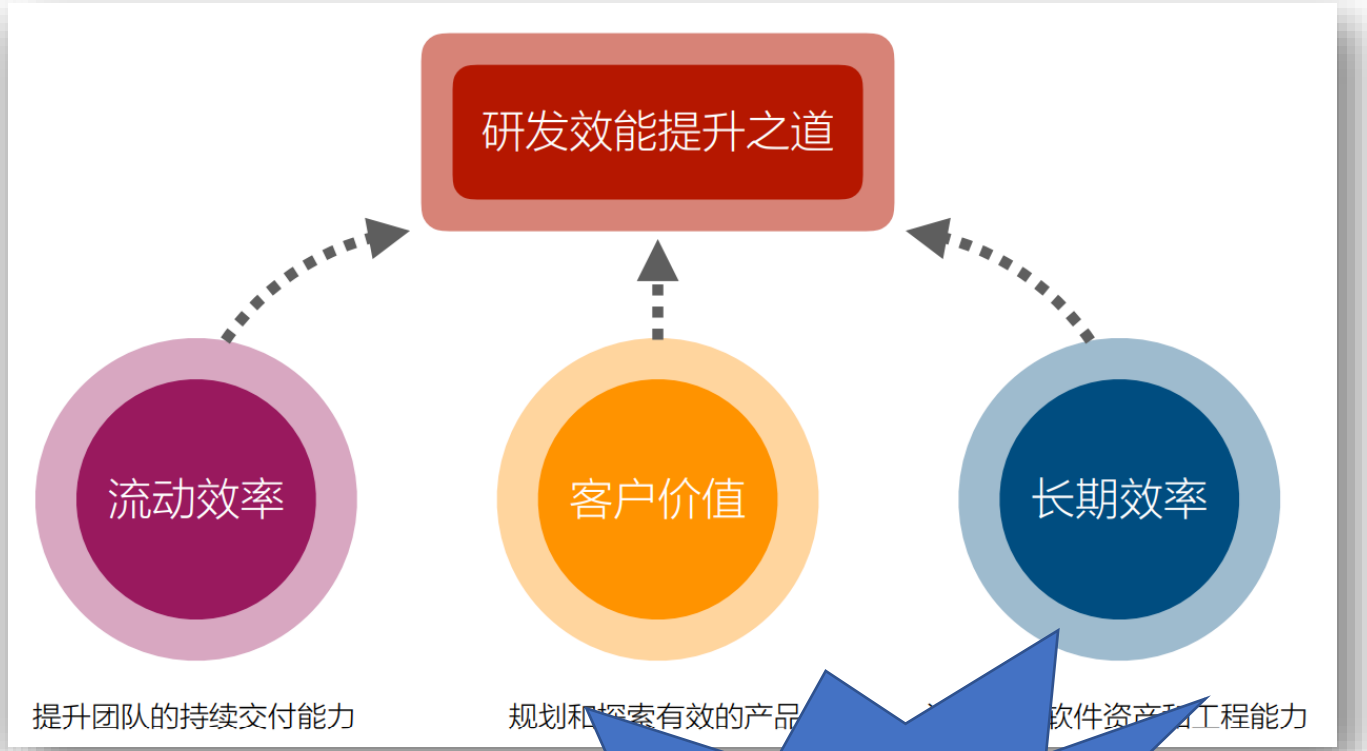
规范——应对重复踩坑，提升一致性

建议：优化研发方法，关键文档不能少，要运营改善



以**长期**效率为核心，沉淀优质软件资产和工程能力，降低债务积累的速度





鼓励长期主义，强化价值创造

- 摘之阿里云效

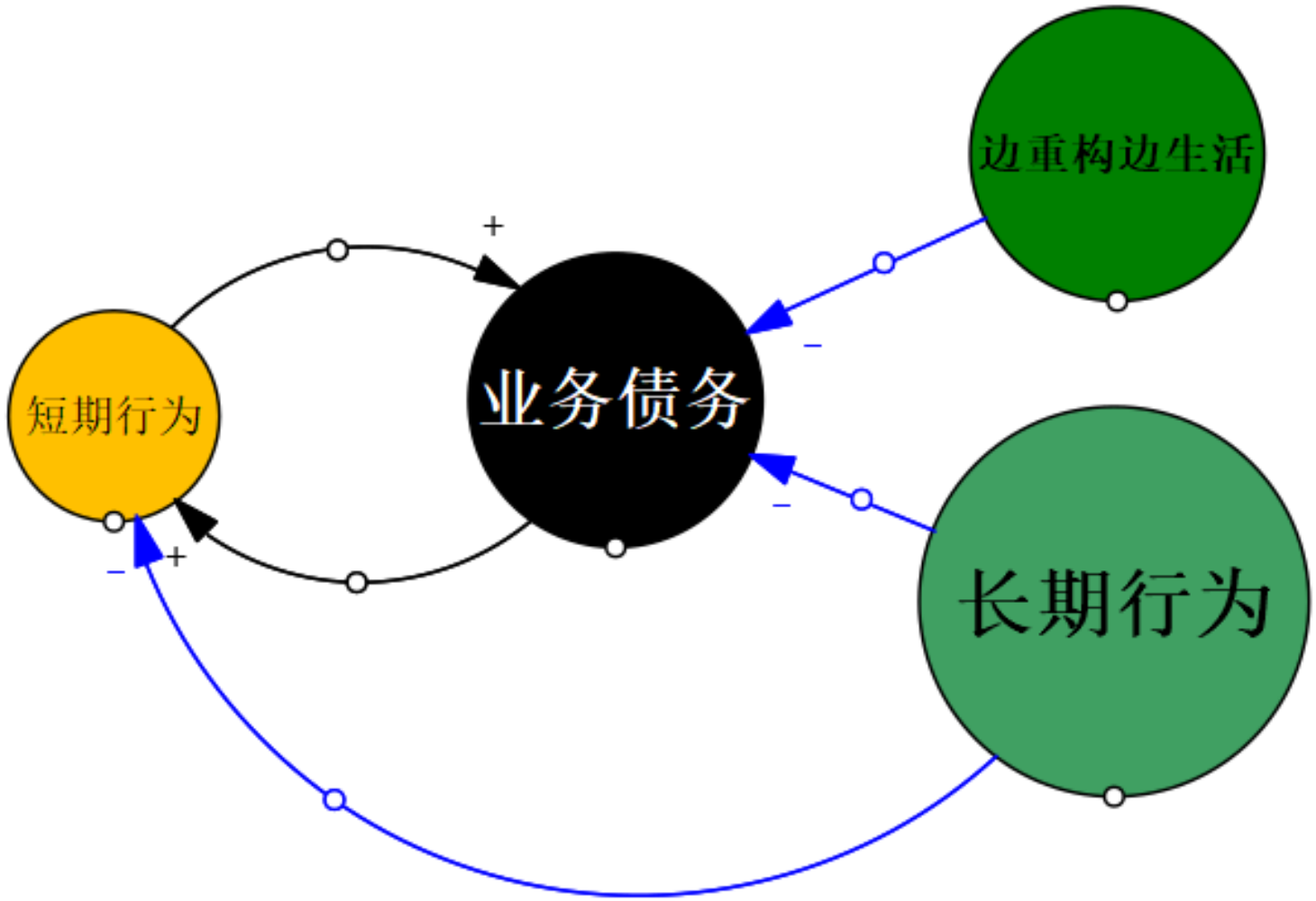
“债是还不完的”

熵增定律：

一个**孤立系统**中，熵永不减小。如果过程是可逆的，则其熵不变，如果过程**不可逆**，则**熵**（无序程度）增加。

我们只能通过有效的手段控制让债务增加的速度变慢，但很难做到不增加债务

鼓励长期主义，强化价值创造



拒絕

破窗

变坏容易，变好难
过日子不易，边重构边生活

在非线性的世界里，不要用线性的思维模式

业务债务和我们每个都相关，我们才是债务的始做俑者

改进，从我们每一个人开始

Q&A