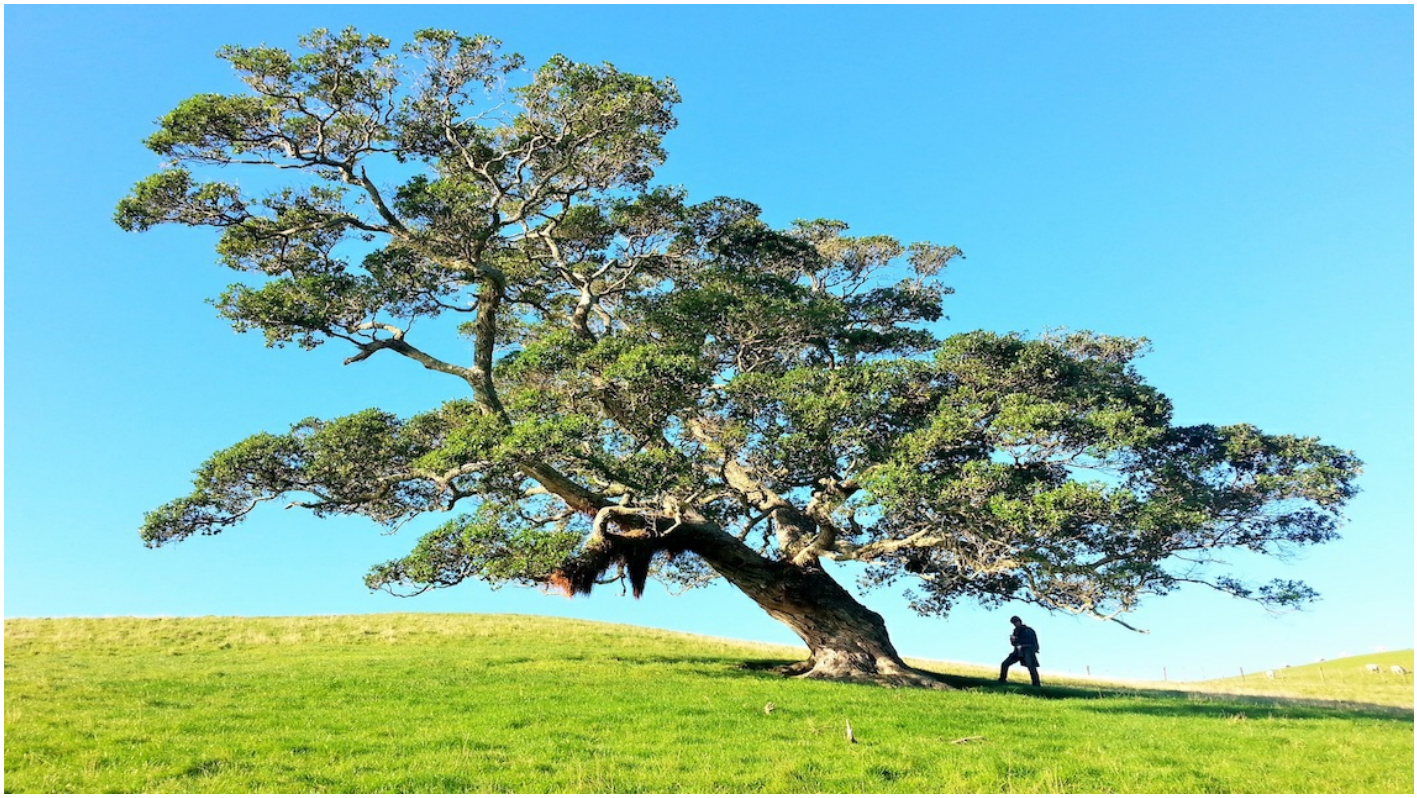
51讲技术债务，有意或无意的选择?



在编程的路上，我们总会碰到历史系统，接手遗留代码，然后就会忍不住抱怨，那我们是在抱怨什么呢？是债务，技术债务。以前说过，**代码既是资产也是债务**，而历史系统的遗留代码往往是大量技术债务的爆发地。

然而，技术债务到底是如何产生的？它是我们有意还是无意的选择？这里就先从技术债务的认知开始谈起吧。

认知

技术债务，最早源自沃德·坎宁安（Ward Cunningham） 1992 年在一次报告上创造的源自金融债务的比喻，它指的是**在程序设计与开发过程中，有意或无意做出的错误或不理想的技术决策，由此带来的后果，逐步累积，就像债务一样**。

当作为程序员的我们采用了一个非最优或不理想的技术方案时，就已经引入了技术债务。而这个决定，可能是有意的，也可能是无意的。有意产生的债务，一般是根据实际项目情况，如资源与期限，做出的妥协。

而无意产生的债务，要么是因为程序员经验的缺乏引入的，要么是程序员所处的环境并没有鼓励其去关注技术债务，只是不断地生产完成需求的代码。但只要程序员在不断地生产代码，那他们就是在同时创造资产与债务。债务如果持续上升，软件在技术上的风险就不断增加，最后慢慢走向技术破产。

以前看过另一位程序员写的一篇文章，名字就叫《老码农看到的技术债务》，印象还是比较深刻的。文中把技术债务分成了好几类，我记得的大概有如下：

- 战略债务
- 战术债务
- 疏忽债务

战略债务，是为了战略利益故意为之，并长期存在。我理解就是在公司或业务高速发展的阶段，主动放弃了一些技术上的完备与完美性，而保持快速的迭代与试错性。在这个阶段，公司的战略利益是业务的抢占，所以此阶段的公司都有一些类似的口号，比如：先完成，再完美；优雅的接口，糟糕的实现。

这类债务的特点是，负债时间长，但利息不算高且稳定，只要保持长期 “付息”，不还本金也能维持下去。

战术债务，一般是为了应对短期紧急情况采取的折衷办法。这种债务的特点就是高息，其实说高利贷也不为过。

这类债务，一直以来经常碰到。比如，曾经做电信项目时，系统处理工单，主流程上有缺陷，对某一类工单处理会卡住。这时又不太方便停机更新程序，于是就基于系统的动态脚本能力去写了个脚本临时处理这类工单，可以应对当时业务经营的连续性，但缺陷是资源开销大，当超过一定量时 CPU 就会跑满了。这样的技术方案就属于战术债务的应用。为避免“夜长梦多”，当天半夜的业务低谷，我就重新修复上线了新程序，归还了这笔短期临时债务。

疏忽债务，这类债务一般都是无意识的。从某种意义上来说，这就是程序员的成长性债务，随着知识、技能与经验的积累，这类债务会逐步减少。另一方面，如果我们主动创造一个关注技术债务的环境，这类债务就会被有意识地还掉。

从上面的分类可以看出，战略和战术债务都是我们有意识的选择，而疏忽债务正如其名，是无意识的。但不论技术债务是有意的还是无意的，我们都需要有意识地管理它们。

管理

对于技术债务，开发团队中的不同角色关注的债务分类与形态也不太一样。

比如架构师关注的更多是战略债务，保持系统能够健康长期演进的债务平衡。作为架构师，就像 CFO，需要长期持续地关注系统的资产负债表。战略债务可能更多体现为架构、设计与交互方面的形态。而具体某个功能实现层面的代码债务，则更多落在相关开发工程师的关注范围内。测试工程师，会关注质量方面的债务，而一到交接时，各种文档债务就冒出来了。

那对于一个软件系统，我们如何知道技术债务已经积累到了需要去警示并着手计划进行还债的阶段了呢？一般来说，我们直觉都是知道的。

举个例子来说明下，好几年前团队接手继续开发并维护一个系统，系统的业务一开始发展很快，不停地添加功能，每周都要上好几次线。一年后，还是每周都要上好几次线，但每次上线的时间越来越长，回归测试的工作量越来越大。再后来，系统迎来了更多的新业务，我们不得不复制了整个系统的代码，修改，再重新部署，以免影响现有线上系统的正常运行…

到了这样的状况，每个人都知道，债务在报警了，债主找上门了。一次重大的还债行动计划开始了，由于还债的名声不太好听，所以我们喜欢叫：架构升级。架构升级除了还债，还是为未来铺路。当然，前提是要有未来。如果未来还能迎来更大的业务爆发增长，架构升级就是为了在那时能消化更多的长短期债务。

管理债务的目标就是识别出债务，并明了不同类型的债务应该在何时归还，不要让债务持续累积并导致技术破产。一般来说，只要感觉到团队生产力下降，很可能就是因为有技术债的影响。这时，我们就需要识别出隐藏的债务，评估其 “利率” 并判断是否需要还上这笔债，以及何时还。

有时，我们会为债务感到焦虑，期望通过一次大规模重构或架构升级还掉所有的债务，从此无债一身轻。其实，这是理想状态，长期负债才是现实状态。

清偿

在产品突进，四处攻城略地时，还需要配合周期性地还债，保持债务平衡，才能保证系统整体健康稳步地发展。

首先，我们认识并理解了技术债务，识别出了系统中的各种债务，并搞清楚了每种债务的类型和利率，这时就需要确定合理的清偿还债方式了。

对于战略债务，长期来说都是持续付利。就像现实中一些大企业从银行借钱经营发展，每年按期付息，但基本不还本金；等公司快速发展到了一定阶段，基本进入成熟期后，市场大局已定，再主动降低负债风险和经营成本。

创业公司从小到大的发展过程中，业务在高速增长，系统服务的实现即使没那么优化，但只要能通过加机器扩展，就暂时没必要去归还实现层面的负债。无非是早期多浪费点机器资源，等业务到了一定规模、进入平稳期后，再一次性清偿掉这笔实现负债，降低运营成本。

这就是技术上的战略债务，**业务高速发展期保持付息，稳定期后一次性归还**。

战术债务，因为利息很高，所以一般都是**快借快还**。而疏忽债务，需要坚持**成长性归还策略**，一旦发现过去的自己写下了愚蠢的代码，就需要主动积极地确认并及时优化，清偿这笔代码实现债务。

其次，还债时，我们主要考虑债务的大小和还债的时机，在不同的时间还债，也许研发成本相差不大，但机会成本相差很大，这一点前面分析战略债务时已有提及。而按不同债务的大小，又可以分为大债务和小债务。一般，我把需要以周或月为单位计算的债务算作大债务，而只需一个程序员两三天时间内归还的债务算作小债务，所以这不是一个精确的定义。

小债务的归还，基本都属于日常的重构活动，局限在局部区域，如：模块、子服务的实现层面。而大债务的归还，比如：架构升级，就需要仔细地考虑和分析机会成本与潜在收益，所以大债务归还要分三步走：

1. 规划：代表愿景，分析哪些债务需要在什么时间还，机会成本的损失与预期收益是多少。
2. 计划：代表路径，细致的债务分期偿还迭代计划。
3. 跟踪：代表过程，真正上路了，确认债务的偿还质量与到位情况。

如今微服务架构的流行，基本把小债务锁定在了一个或几个微服务体内。即使债务累积导致一两个微服务技术破产，这时的还债方式无非就是完全重写一个，在微服务拆分合理的情况下，一个服务的重写成本是完全可预期和可控的。

除了技术债务的管理与清偿，我们还需关注技术债务与作为程序员的我们之间的信用关系，因为毕竟债务也是我们生产出来的。

信用

生产并拥有技术债务的程序员，并不代表其信用就差。

现实生活中，债务依附于借债的主体方，比如金融债务依附于个体或组织，但如果个体死亡或组织破产了，债务就失去了依附体，自然也就消失了。而技术债务的依附体，并不是程序员，而是程序构造的产品或系统，所以当产品或系统的生命周期结束时，相应的技术债务也会消失。

因而，此种情况下，程序员是有充足的理由不还技术债的，这是技术决策的一种，并不会降低程序员的信用。

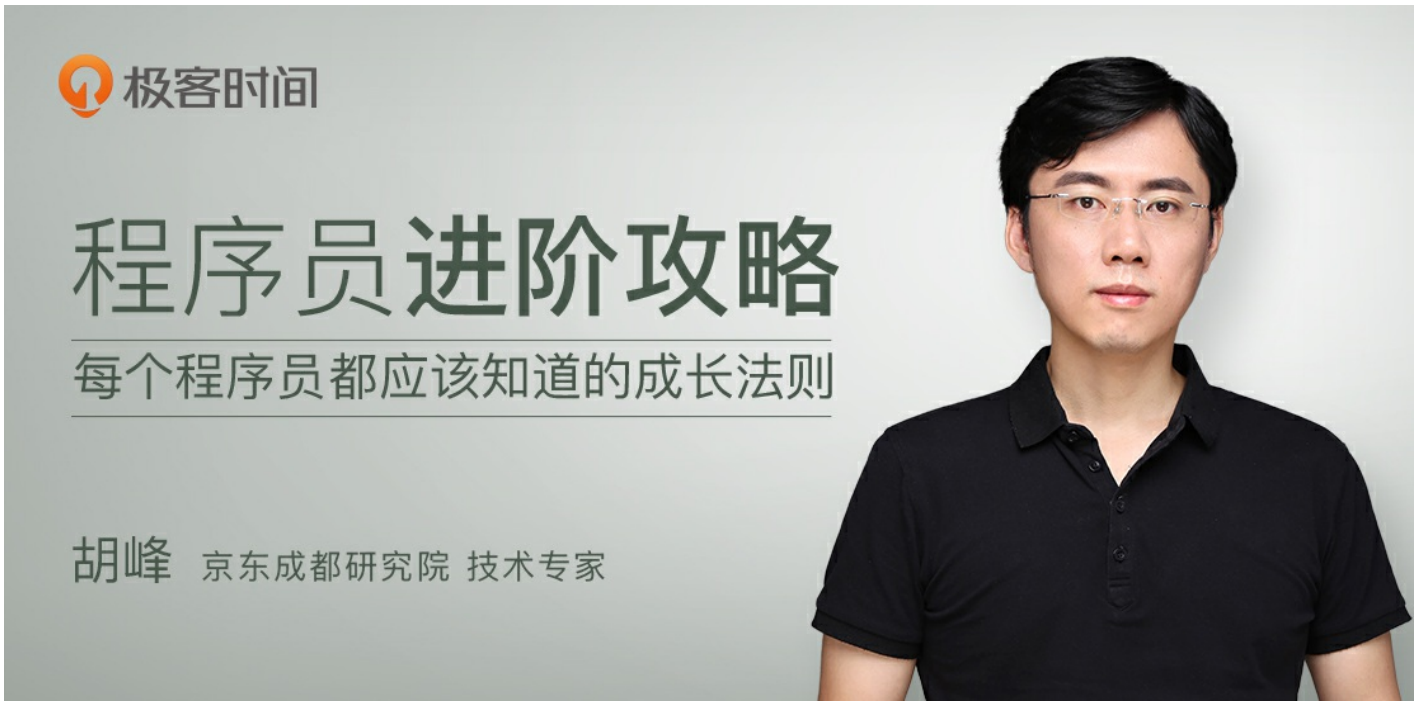
任何一个程序系统或其一部分都会与某一个程序员建立关联，如果这个程序员此时就负责这部分系统，那么他在此基础上继续创造代码时，既增加了资产也可能引入了新的债务。这时他的一个重要职责就是，维持好资产与债务的平衡关系。如果在此期间，系统的债务失衡导致技术破产，需要被迫大规模重构或重写时，那么这个程序员的信用必将受到关联伤害。

所以，**程序员的信用，更多体现在面对技术债务的态度和能力**——有意识地引入债务，并有计划地归还债务；无意识地引入债务，发现之后，有意识地归还。

再有代码洁癖的人也没法写出无债务的代码，而无债务的系统也不存在。面对负债高的系统，我们不必过于焦虑，高负债的系统往往活力还比较强。作为程序员的我们，应把技术债务当作一门工具，而不是一种负担，那么可能就又获得了新的技能，得到了成长。

总之，面对债务，做一个有信用的程序员；将其当作工具，做一个有魄力的技术决策者。

最后，你也可以聊聊你对技术债务的态度和看法，欢迎留言一起讨论。



精选留言

黄蓓

为了加载数据更快，前人在provider进程当中实现了一个内存数据库，随着业务的增长，内存数据库越来越复杂，还出现了数据丢失和不一致的情况，这个债务已经还不起了

2018-11-28 08:54

JohnT3e

知道引入了哪些债务，多少债务，何时偿还是关键，更多情况下往往是债务危机爆发时才发觉。

2018-11-28 08:48

godtrue

现在项目组正在重构我们的核心项目，系统使用每天，不过业务运营比较费劲，特别是特殊时期时，重复操作重复验证工作比较多，我们重构后，希望能够减轻一些运营成本。

2018-11-28 08:40

作者回复

大规模重构一定要提前评估计划，期间的实施成本压力也很大的

2018-11-29 15:15