15 | 技术债务：那些不可忽视的潜在问题

2019-11-14 石雪峰

DevOps实战笔记 进入课程 >

讲述：石雪峰
时长 15:47 大小 14.47M

你好，我是石雪峰，今天我来跟你聊聊技术债务。

如果要问软件开发人员在项目中最不愿意遇到的事情，答案很可能是接手了一个别人开发了一半的系统。而且，系统开发的时间越长，开发人员的抵触情绪也就越大。那么，既然是同一种代码语言，同一种语法规则，至少还是一个能运行的东西，开发人员为什么要发自内心地抵触呢？我猜，很可能是不想看别人写的代码。之所以会这样，看不懂和怕改错是一个非常重要的原因，而这些，其实都是技术债务的结果。

什么是技术债务？

那么，究竟什么是技术债务呢？它是从哪里来的呢？好好地写个代码，咋还欠债了呢？

试想这样一种场景，老板拍下来一个紧急需求，要求你在 3 天内开发完成上线。在评估需求和设计的时候，你发现，要实现这个功能，有两种方案：

方案 1：采用分层架构，引入消息队列。这样做的好处是结构清晰，功能解耦，但是需要 1 周的时间；

方案 2：直接在原有代码的基础上修修补补，硬塞进去一块逻辑和页面，这样做需要 2 天时间，还有 1 天时间来测试。

那么，你会选择哪个方案呢？

我想，在大多数情况下，你可能都会选择方案 2，因为业务的需求优先级始终是最高的。尤其是当下，市场竞争恨不得以秒来计算，先发优势非常明显。

而技术债务，就是指团队在开发过程中，为了实现短期目标选择了一种权宜之计，而非更好的解决方案，所要付出的代价。这个代价就是团队后续维护这套代码的额外工作成本，并且只要是债务就会有利息，债务偿还得越晚，代价也就越高。

实际上，带来技术债务的原因有很多，除了压力之下的快速开发之外，还包括不明真相的临时解决方案、新员工技术水平不足，和历史债务累积下来的无奈之举等。总之，代码维护的时间越长，引入的技术债务就会越多，从而使团队背上沉重的负担。

技术债务长什么样？

简单来说，你可以把技术债务理解为不好的代码。但是这里的“不好”，究竟是哪里不好呢？我相信，写过代码的人，或多或少都有过这样的经历：

一份代码里面定义了一堆全局变量，各个角落都在引用；

一个脚本仓库里面，一大堆名字看起来差不多的脚本，内容也都差不多；

一个函数里面修修补补写了上千行；

数据表查询各种神奇的关联；

参数传递纯靠肉眼计算顺序；

因为修改一段代码引发了一系列莫名其妙的问题；

……

那么，究竟要如何对代码的技术债务进行分类呢？我们可以借用“Sonar Code Quality Testing Essentials”一书中的代码“七宗罪”，也就是复杂性、重复代码、代码规范、注释有效性、测试覆盖度、潜在缺陷和系统架构七种典型问题。你可以参考一下这七种类型对应的解释和描述：

类型	影响
不能均匀分布的复杂性	较高的圈复杂度需要更多的测试才能覆盖到全路径，导致潜在的功能质量风险。
重复代码	重复代码是最严重的问题，会导致潜在缺陷。另外，重复也会带来维护成本的增加。
不合适的注释	代码的注释没有明确标准。缺少关键环节的注释或者是注释难以理解，都会导致代码的可读性变差。
违反代码规范	影像团队基于共同的规范进行写作，会增加潜在的风险。
缺乏单元测试	单元测试不足会影响团队对代码的信心，增加重构成本，通过测试覆盖率对其进行度量。
缺陷和潜在的缺陷	缺陷和潜在缺陷是最直接影响代码质量的要素，要尽可能地发现并修复。
设计和系统架构	设计和系统架构受限于当时的资源条件，可能无法满足后续产品的发展需求，所以需要持续演进。

除了低质量的代码问题之外，还有很多其他类型的技术债务，比如不合理的架构、过时的技术、冷门的技术语言等等。

比如，我们公司之前基于 Ruby 语言开发了一套系统，但是与 Java、Python 等流行语言相比，Ruby 比较小众，所以很难找到合适的工程师，也影响了系统的进一步发展。再比如，到 2020 年元旦，官方即将停止为 Python 2.x 分支提供任何支持，如果现在你们的新系统还在采用 Python 2 进行开发，那么很快就将面对升级大版本的问题。虽然官方提供

了一些减少迁移成本的方案，但是，从系统稳定性等方面来讲，依然有着非常大的潜在工作量。

为什么要重视技术债务？

那么问题来了，为什么要重视技术债务呢？或者说，烂代码会有什么问题呢？

从用户的角度来说，技术债务的多少好像并不影响用户的直观体验，说白了就是不耽误使用，应该有的功能都很正常。那么，回到最开始的那个例子，既然 2 天开发的系统，和 1 周开发的系统，从使用的角度来说并没有什么区别，那是不是就意味着，理应选择时间成本更低的方案呢？

显然没有这么简单。举个例子，一个人出门时衣着得体，但是家里却乱成一团，找点东西总是要花很长时间，这当然不是什么值得骄傲的事情。对于软件来说，也是如此。**技术债务最直接的影响就是内部代码质量的高低**。如果软件内部质量很差，会带来 3 个方面的影响：

1. 额外的研发成本

对一个架构清晰、代码规范、逻辑有序、注释全面的系统来说，新增一个特性可能只需要 1～2 天时间。但是，同样的需求，在一个混乱的代码里面，可能要花上 1 周甚至是更长的时间。因为，单是理解原有代码的逻辑、理清调用关系、把所有潜在的坑趟出来就不是件容易的事情。更何况还有大量重复的代码，每个地方都要修改一遍，一不小心就会出问题。

2. 不稳定的产品质量

代码质量越差，修改问题所带来的影响可能就越大，因为你不知道改了一处内容，会在哪个边缘角落引发异常问题。而且，这类代码往往也没有可靠的测试案例，能够保证修改前和修改后的逻辑是正确的。如果新增一个功能，导致了严重的线上问题，这时就要面临是继续修改还是回滚的选择问题。因为如果继续修改，可能会越错越多，就像一个无底洞一样，怎么都填不满。

3. 难以维护的产品

正是由于以上这些问题，研发人员在维护这种代码的时候往往是小心加谨慎，生怕出问题。这样一来，研发人员宁愿修修补补，也不愿意改变原有的逻辑，这就会导致代码质量陷入一

种不断变坏的向下螺旋，越来越难以维护，问题越积累越多，直到再也没办法维护的那一天，就以重构的名义，推倒重来。其实这压根就不是重构，而是重写。

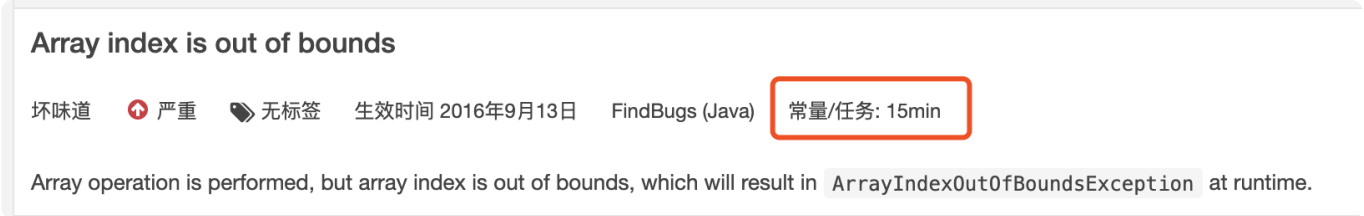
另外，如果研发团队整天跟这样的项目打交道，团队的学习能力和工作积极性都有可能受到影响。可见，技术债务的积累就像真的债务一样，属于“出来混，迟早要还”的那种，只不过是谁来还的问题而已。

如何量化技术债务？

软件开发不像是银行贷款，技术债务看不见摸不着，所以，我们需要一套计算方法把这种债务量化出来。**目前业界比较常用的开源软件，就是 SonarQube**。在 SonarQube 中，技术债是基于 SQALE 方法计算出来的。关于SQALE，全称是 Software Quality Assessment based on Lifecycle Expectations，这是一种开源算法。当然，今天的重点不是讲这个算法，你可以在官网查看更多的内容。同时，我再跟你分享一篇关于 SQALE 算法的文章，它可以帮你更深入地研究代码质量。

Sonar 通过将不同类型的规则，按照一套标准的算法进行识别和统计，最终汇总成一个时间，也就是说，要解决扫描出来的这些问题，需要花费的时间成本大概是多少，从而对代码质量有一种直观的认识。

Sonar 提供了一种通用的换算公式。举个例子，如下图所示，在 Sonar 的默认规则中，数据越界问题被定义为严重级别的问题，换算出来的技术债务等于 15 分钟。这里的 15 分钟，就是根据前面提到的 SQALE 分析模型计算得出的。当然，你也可以在规则配置里面对每一条规则的预计修复时间进行自定义。



计算出来的技术债务会因为开启的规则数量和种类的不同而不同。就像我在上一讲中提到的那样，团队内部对规则达成共识，是非常重要的。因为只有达成了共识，才能在这个基础上进行优化。否则，如果规则库变来变去，技术债务指标也会跟着变化，这样就很难看出团队代码质量的长期走势了。

另外，在 Sonar 中，还有一个更加直观的指标来表示代码质量，这就是 **SQALE 级别**。SQALE 的级别为 A、B、C、D、E，其中 A 是最高等级，意味着代码质量水平最高。级别的算法完全是基于技术债务比例得来的。简单来说，就是**根据当前代码的行数，计算修复技术债务的时间成本和完全重写这个代码的时间成本的比例**。在极端情况下，一份代码的技术债务修复时长甚至比完全推倒重写还要长，这就说明代码已经到了无法维护的境地。所以在具体实践的时候，也会格外重视代码的 SQALE 级别的健康程度。

> 技术债务比例 = 修复已有技术债务的时间 / 完全重写全部代码的时间

将代码行数引入进来，可以更加客观地计算整体质量水平。毕竟，一个 10 万行的代码项目和一个 1 千行的代码项目比较技术债务本身就没有意义。其实，这里体现了一种更加可视化的度量方式。比如，现在很多公司在做团队的效能度量时，往往会引入一大堆的指标来计算，根本看不懂。更加高级的做法，是将各种指标汇总成一组算法，并根据算法给出相应的评级。

当然，如果你想知道评级的计算方法，也可以层层展开，查看详细的数据。比如，持续集成能力，它是由持续集成频率、持续集成时长、持续集成成功率、问题修复时长等多个指标共同组成的。如果在度量过程中，你发现持续集成的整体评分不高，就可以点击进去查看每个指标的数据和状态，以及详细的执行历史。这种数据关联和下钻的能力对构建数据度量体系而言非常重要。

通过将技术债务可视化，团队会对代码质量有更直观的认识，那么接下来，就要解决这些问题了。

解决方法和原则

我走访过很多公司，他们都懂得技术债务的危害，不仅把 Sonar 搭建起来了，还定时执行了，但问题是没时间。的确，很多时候，我们没时间做单测，没时间做代码评审，没时间解决技术债务，但是这样一路妥协，啥时候是个头儿呢？

前几天，我去拜访一家国内最大的券商公司，眼前一亮。这样一家所谓的传统企业，在研项目的技术债务居然是个位数。在跟他们深入交流之后，我发现，公司在这方面下了大力气，高层领导强力管控，质量门禁严格执行，所以才获得了这样的效果。

所以，从来没有一切外部条件都具备的时候，要做的就是先干再说。那么，要想解决技术债务，有哪些步骤呢？

1. 共识：团队内部要对技术债务的危害、解决项目的目标、规则的选择和制定达成一致意见。
2. 可见：通过搭建开源的 Sonar 平台，将代码扫描整合进持续交付流水线中，定期或者按需执行，让技术债务变得可视化和可量化。不仅如此，Sonar 平台还能针对识别出来的问题，给出建议的解决方法，这对于团队快速提升编码水平，大有帮助。
3. 止损：针对核心业务模块，对核心指标类型，比如 vulnerability，缺陷的严重和阻塞问题设定基线，也就是控制整体数量不再增长。
4. 改善：创建技术优化需求，并在迭代中留出一定的时间修复已有问题，或者采用集中突击的方式搞定大头儿，再持续改进。

在解决技术债务的过程中，要遵循 4 条原则。

1. **让技术债务呈良性下降趋势**。一种好的趋势意味着一个好的起点，也是团队共同维护技术债务的一种约定。
2. **优先解决高频修改的问题**。技术债务的利息就是引入新功能的额外成本，那么对于高频修改的模块来说，这种成本会快速累积，这也就意味着修复的产出是最大的。至于哪些代码是高频修改的，只要通过**分析版本控制系统**就可以看出来。
3. **在新项目中启动试点**。如果现有的代码过于庞大，不可能在短时间内完成修复，那么你可以选择控制增长，同时在新项目中试点执行，一方面磨合规则的有效性，另一方面，也能试点质量门禁、IDE 插件集成等自动化流程。
4. **技术债务无法被消灭，也不要等到太晚**。只要还在开发软件项目，技术债务就基本上无法避免，所以不需要一下子把目标定得太高，循序渐进就行了。但同时，技术债务的累积也不是无穷无尽的，等到再也无法维护的时候就太迟了。

在刚开始解决技术债务的时候，最大的问题不是参考指标太少，而是太多了。所以团队需要花大量时间来 Review 规则。关于这个问题，我给你两条建议：第一，参考代码质量平台的默认问题级别。一般来说，阻塞和严重的问题的优先级比一般问题更高，这也是基于代码质量平台长时间的专业积累得出的结论。第二，你可以参考业界优秀公司的实践经验，比如很

多公司都在参考阿里巴巴的 Java 开发手册，京东也有自己的编码规约。最后，我总结了一些影响比较大的问题类型，建议你优先进行处理。

大量重复代码；

类之间的耦合严重；

方法过于复杂；

条件判断嵌套太多；

缺少必要的异常处理；

多表关联和缺少索引；

代码风险和缺陷；

安全漏洞。

总结

在这一讲中，我给你介绍了什么是技术债。而技术债的成本，就是团队后续开发新功能的额外成本。技术债务有很多形态，典型的就是代码“七宗罪”。除此之外，我还跟你聊了下技术债的影响，以及量化技术债务的方法。最后，我给出了一些解决方法和原则，希望能帮你攻克技术债这个难题。

最近这两年，智能研发的声音不绝于耳，其中关于使用人工智能和大数据技术提升代码质量的方法，是目前的一个热门研究领域。通过技术手段，辅助研发解决技术问题，在未来是一种趋势。如果你在公司中从事的是研发辅助和效率提升类的工作，建议你深入研究下相关的学术文章，这对你的工作会大有裨益。

> 参考资料：
> 1. 通过持续监控实现代码克隆的定制化管理
> 2. 基于代码大数据的软件开发质量追溯体系
> 3. 代码克隆那点事：开发人员为何克隆？现状如何改变？

思考题

你遇到过印象深刻的烂代码吗？

欢迎在留言区写下你的思考和答案，我们一起讨论，共同学习进步。如果你觉得这篇文章对你有所帮助，欢迎你把文章分享给你的朋友。

上一篇　14 | 内建质量：丰田和业与迈格我们的启示

下一篇　16 | 环境管理：一切皆代码是一种什么样的体验？

精选留言 (5)

写留言

桃子-夏勇杰
2019-11-14

老师可以分享一下，什么样的技术债还起来比较容易，收益又比较大的？或者分享一个你印象特别的技术债。

作者回复: 你好，我理解你的问题是哪些技术债修复的投入产出比最高，或者最应该优先处理对吧？这个因项目而已，我给你的建议是参考大厂比如阿里巴巴的编码规范，选择其中优先级较高的问题进行修复。另外在我们实际工作中，有以下这些问题的优先级较高，你也可以参考以下：
线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式
多线程并行处理定时任务时，使用ScheduledExecutorService。
在if/else/for/while/do语句中必须使用大括号，即使只有一行代码，避免使用下面的形式：if (condition) statements;
所有的包装类对象之间值的比较，全部使用equals方法比较。
在使用正则表达式时，利用好其预编译功能，可以有效加快正则匹配速度。
不要在foreach循环里进行元素的remove/add操作
避免通过一个类的对象引用访问此类的静态变量或静态方法

2

陈斯佳
2019-11-15

这一周为了解决一个脚本的问题，硬是花了三四天的时间来理解同事写的这个代码，最后修改其实只要改一行代码就够了……我大好的青春啊！

展开

作者回复: 看来这个代码的技术债已经已经达到几十小时的规模了，好人做到底多加点注释吧😂

1

陈□ 争
2019-11-14

之前有一个项目由于时间紧张，没有抽取公共方法，每个人都按自己的逻辑写。比如一个下拉列表查询功能，三个人就会写出三种实现方式，这个不是代码重复的问题，而是完全

不重复，无法复用！！！
今天学习了SonarQube，希望在以后的工作中能够把这个工具用起来，做到技术债直观可视，这样可以更方便的跟领导解释技术债的重要性。

展开

作者回复: 是的，一定要让抽象的事物具象化话，整好我今天遇到一个公司，他们甚至将Sonarqube里面的规则定义了一套自己的算法，并结合多种代码质量类型，比如安全等，最终给出了一个可量化的得分，我个人觉得这个方法很不错，没有比较就没有伤害哈。

1

sugar
2019-11-15

感觉遇到的好多烂代码（小声哔哔），开发人员技术能力良莠不齐，多数都没有一个完整成熟的手册规范，一顿操作，先上线再说，哎

展开

作者回复: 这就是典型的“先上再说”吧，这就看出来代码检查工具的必要性了，降低人与人之间的摩擦（哔哔），有一套客观的标准，谁也不能说啥。

leslie
2019-11-14

其实提及技术债就不得不去提及领导的魄力和远见以及管理能力。

领导有魄力：发现一堆技术债会定期清理，后续人员发现时上报的时候会在维持现有的情况下控制新内容的增速，为此换取陆续解决相关技术债的问题。没有大局观和魄力的：先解决当下，亡羊补牢，能补多少算多少，问题积累数目偏多，老项目就这样通过硬件优化算了，一切的防御改进新项目再说。...

展开

作者回复: 👍希望你能成为“前一种领导”