Table of Contents

技术债务与开源开发

对于如何更好地理解技术债务以及开源开发如何帮助缓解债务的探讨。

作者：Ibrahim Haddad, Ph.D. & Cedric Bail, M.Sc.

Linux 基金会 2020年7月发布

摘要

本文的灵感来自于2020年3月中旬的一个周末的远程对话，两位作者曾经在三星（Samsung）公司研究院的开源小组的同事，通过与上游开源项目合作，他们对于如何最大限度地减少公司内部的技术债务有着切身体会。这些经验涵盖了跨多个产品和业务部门使用的数十个开源项目，三星在上游开发方面有着不同程度的参与，也有程度不同的体验。

本文概述了关于技术债务的问题。其中包括了对于识别技术债务、如何最小化它、开源开发的作用以及解决问题的策略讨论。

免责声明

本文所表达的观点仅代表作者的观点，并不代表其当前或过去雇主的观点。作者希望提前对任何错误或遗漏道歉，并虚心接受大家的反馈，并会及时更新。

技术债务

定义

技术债务，在软件开发中的定义有点类似于现实世界的经济债务，指的是维护源代码的成本，这些成本主要是由偏离了联合（joint ）开发的主分支引起的。其实，这个词在专有代码中有着更为广泛的解释：

- 由单一的组织所开发
- 源代码的服务和维护主要由这家组织自个承担
- 在某些情况下，组织依赖于合作伙伴维护代码并承担所述债务的能力。

这里需要先澄清一个事实，上游代码也会有技术债务的，尤其是当上游项目没有资源/时间通过其开发人员共同体进行自我维护时，这样的情况是有很好的案例的，例如 OpenSSL 项目，很多公司都严重依赖于这个项目，但是并没有亲力亲为的参与到这个项目，然而当出了问题时才发现这么重要的项目竟然是仅仅只有一个人在业余时间进行维护。当然，由于正是这个案例的发生，促使 Linux 基金会发起了 Core Infrastructure Initiative ，以支持至关重要的现代核心基础设施开源项目。

技术债务的外在表现

我们通常是如何识别哪些技术债务的？它们有什么明显的表现吗？在这个小节中，笔者根据自身的经验，就将这些技术债务一一列出，并将它们的外在表现描述一番。以下所列并没有覆盖所有技术债务相关的详尽清单，而是笔者认为的最为常见的，也是被很多人所观察到的技术债务外在表现。

- **较慢的发布节奏** 新的功能交付周期变长。
- **新人参与的时间变久** ：首要的表现是新来的开发者非常难以参与到项目中，而只有内部开发人员熟悉代码的复杂性；第二种表现则是，老员工留不住，又招不来新员工。
- **安全方面的问题增加**：相比于上游项目，遇到了更多的安全问题。
- **投入代码库的力度加大**：随着要维护的代码体量变得更大、更复杂，维护任务变得更加耗时。
- **与上游渐行渐远**：无法与上游开发、发布周期保持同步和一致。

技术债务的类型

（有关开源的）技术债务类型，并没有一个标准，或者形成普遍的共识，当然也就没有现成的定义，这个任务需要我们自己来完成。

临时技术债务

这里假设是一个正在正在开发的团队，基于上游开源项目做一些研究，可能涉及多个组件、多个系统/子系统等复杂的功能。随着项目的进度，开始有了一些没有和上游进行整合的技术债务，此时，团队里的人也意识到了这些，也明白自己已经欠了

债务，然而，由于赶着上市，团队最终的决策是先将这些债务暂时搁置一旁，不予理会，并承诺在将来再去做和上游合并的事。

未知的技术债务

由于不良的工程实践，在无形中产生的技术债务。这样的情况常见的例子：一段糟糕的代码并没有被上游接受，甚至也不适合在其它地方重复利用。那么这段代码就成了“弃之可惜，食之无味”的债务。

有目的的欠下了技术债务

更多的情况是开发者自己有意为之的。常见的情形是：一个开发团队基于上游项目开发了自己认为先进而独特的功能，然而并不想和上游以及整个开源共同体分享，于是，他们创建了自己的分支，而拒绝和上游合并。随着时间的推移，自行开发的fork版本也会不断累积，这就导致了更多的技术债务，相应的维护成本也在不断的增加。

过时的技术债务

这是一种较为独特的技术债务类型，主要是由于强调“自主可控”的执念，以及不想让更多人知道自己的开发所导致。 由于自身的封闭，以及缺乏技术监督，最后只有自己一家在使用这项实现。然而，开源世界在不断的迭代和进化，并优雅的解决了问题并成为了事实上的标准，但是和内部实现并不兼容。这种情况就是发展过时的技术债务的来由。

组织技术债务

人们经常会讨论的一个话题是一家企业的代码和这家企业的组织是高度相关的（康威定律——译者注），这是一个非常有意思的现象。

在某些情况下，尤其是利用开源，会出现一些代码企业内无人理解，也无人能修改和debug的情况，然而这些代码也是没有人负责的，没有人愿意为这些代码负责到底。

当出现人员和代码不相匹配的情况时，请小心，说明这家企业已经欠下了技术债务。

引起技术债务的几个原因

造成技术债务的形成和累积有非常多的原因，在本小节中，尝试列举出一些常见的原因及其概要的描述。

- 低质量的代码，从而无法被上游所接受，例如不符合上游项目所设置的开发质量标准，“意大利面条”式的内部开发的代码通常是引起这样结果的缘由。
- 所开发的代码仅适用于企业自身的业务需求，而非项目的通用，这样的代码，上游通常是不接受的，要进行一定的调整，以使一般用例和更广泛的用户群受益。
- 碎片化的开发导致重复的努力，和同质竞争的出现。
- 缺乏推动上游优先的动力—— 在很多情况下，开发团队所在的组织没有更多的资源、能力参与到共同体，以及驱动开发者基于上游而工作。在短期来看，该

组织获得了一定的红利，获得了不错的效率，但是从代码维护和改进的方面来看则产生了技术债务，长期而言，这是相当糟糕的负面影响。

- 需要跨多个组件或各种系统/子系统进行额外协调的侵入性代码（Intrusive code）。（当然这是一种不好的编码习惯。—— 译者注）
- 从本地提交给上游，到被上游接受，中间是有一个时间差，也就是所谓的临时的技术债务代码，但是代码一旦被接受并合并到上游分支中，这些债务就会被很好的抵消。从长远来看，这么做没有不良影响。
- 测试的缺失，不仅会阻碍完整的测试覆盖率，还会导致提交的失败。
- 文档的缺失，或者是文档的陈旧。
- 技术领导力的缺乏，会导致和技术项目共同体的疏远，团队就会变得边缘化，这样的话，就意味着团队只能另外开辟路径来确保项目的可持续发展。
- 在任何组织中，内部开发的需求都是不断的发生着变化。
- 采用了非标准的技术，或者是标准缺乏一致性。
- 所在组织不够重视，加上技术领导力的不足，会导致对上游缺乏了解，这种情况越发的在非原生数字公司成为了普遍现象，这些公司被迫承担本不属于自身的工作。

总结

欠下技术债务，或者说累积技术债务，对于公司会产生很多的负面影响，例如：

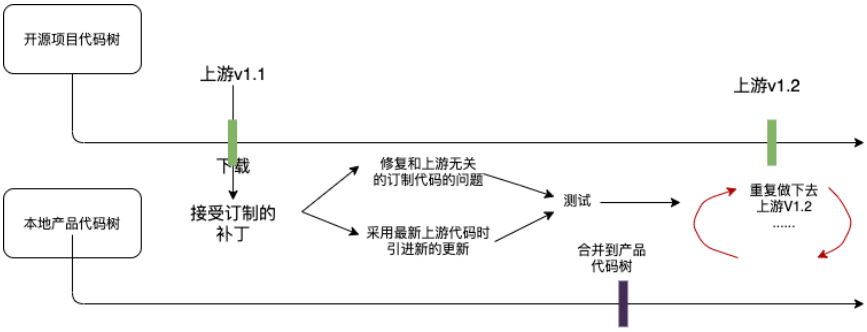
- 代码维护的高昂代价
- 创新的匮乏与拖沓的开发周期
- 需不断交付的债务利息 —— 技术债务不是说可以不还的，而是以另外的形式存在：为了跟上上游的额外开发、无效的恶性竞争、无暇顾及其它。
- 主分支中可能缺少新功能，或者必须将所有新开发内容向后移植到内部分支中。
- 内部版本和上游版本差异太大，因此需要投入额外的精力去做和上游一样重复的工作。

最糟糕的后果是对代码库的长期可维护性的影响，组织经常会发现只有自己在维护了一个 fork 版。

技术债务是如何累积的

美国作家Arthur Bloch，墨菲定律一书的作者，曾经说过一句话：**“朋友来来去去，但敌人会不断累积。”** 笔者以为这段话用在技术债务中也是非常贴切的。

这一切又是如何发生的了呢？下图一非常形象的说明了技术债务是如何形成以及不断累积的过程：



图一： 未整合上游到开发周期

处理技术债务

识别技术债务

每一行代码都可能是技术债务，这些技术债务会消耗最需要投入在特定业务中的工程时间。弄清楚这一点主要是要弄清楚您的业务目标以及您的团队将时间投入在什么事项中。回答以下几个问题会对这个思考练习有帮助：

- 您的团队在做什么？
- 您需要将您软件栈的哪些注意事项告知新员工？
- 您多久可以对整个软件栈做一次更新？
- 您知道您所依赖的所有软件吗？
- 通常什么软件会发生故障？
- 处理哪些依赖项最痛苦？
- 对所使用软件栈的每个组件会有什么替代方案？
- 这些替代方案所处社区的活跃程度如何？
- 如果社区消失或停滞不前，需要做多少工作来维护该组件？
- 改用这个组件需要花多少精力？
- 调试客户或工程师报告的问题有多难？

这一系列问题将会帮助您开始讨论技术债务以及您的盲点在哪里。

最小化技术债务

现在要解决的核心问题是如何最大限度地减少技术债务并最大限度地减少其对开发工作的影响。

选择编程语言

用于构建产品的编程语言或开发框架对您的开发人员有一定的限制。语言或框架的复杂性越高，就越难维持可以使用它的劳动力。它成为对您正在为其构建软件的系统的约束和为完成工作对必要的开发人员库进行的访问之间的平衡。通常在大多数情况下，使用更高级别[1]的语言会更好，因为它会：

- 促进招聘优秀的开发人员。
- 帮助第三方识别和解决问题
- 因社区参与而便于移植且易于维护。
- 对错误更宽容，并容许更简单的解决方案。
- 包括文档、教程和预制解决方案的重要在线资源。

[1]高级编程语言的选择是基于相应上下文的，然而，作为此类语言的示例，我们建议使用Go、Python、C#和Typescript。

选择上下游

您的应用程序始终构建在由语言、框架和操作系统组成的软件栈之上。这是要存储应用程序的生态系统的典型结构或组成。这个生态系统将塑造技术债务，开发人员应该知道它如何与自己的目标保持一致。例如：

- Linux发行版具有不同的支持条款，并随着时间推移一直保证各种级别的API/ABI/安全性。
- 随着时间的推移，编程语言的选择会影响软件运行的能力。它会影响作为运行时执行代码的能力和在演变中运行的环境的构建。此外，它可能会限制您寻找新开发人员以完成这些更改的能力。
- 现代语言和框架也倾向于绕过Linux发行版来打包软件。这个因素可能会像Linux发行版一样对您的软件产生长期影响，因此它也应该遵循您的API/ABI/安全性需求。

选择依赖项

随着世界的发展，可用的高质量开源软件越来越多了。这是一个持续的过程，且必须评估您的组织正在开发的每个软件。这些依赖项简化了必要的工作，并使它们能够构建更复杂的功能和解决方案。尽管如此，在社区不够健康时它们也会成为技术债务。选择正确的依赖项并为对您重要的依赖项做贡献，再将其与全世界共享，可减少您的技术债务。

此外，几年前的依赖如今可能不再有相同的价值。这意味着就像处理技术债务一样，对您的依赖关系所做的评估需要持续进行。

示例 1：自定义构建系统的作用

维护操作系统不仅仅是一项全职工作。需要确保：

- 随着时间的推移，它一直在安全可靠地运行；
- 随着时间的推移，每个组件都可以重新构建；
- 对每个组件都进行了适当的评估和测试；
- 尊重许可证，并且；
- 拥有强大且安全的更新机制。

这种类型的工作被低估了，因为现在可以在几个小时内完成一个Linux环境的搭建。然而，这只是长期维护操作系统的第一步。如今，随着ARM进入服务器市场，嵌入式设备的标准化分布变得更加容易。如Debian、Ubuntu、Redhat和SuSE等操作系统提供了在任何嵌入式设备上运行良好或稍作调整的ARM服务器发行版，并且无需维护自定义操作系统和相关的软件包构建。它还为开发人员提供标准化工具，将知识从云环境迁移至嵌入式市场中。随着招聘经理可以接入更大的市场——Linux服务器市场，寻找能够通过嵌入式系统工作的开发人员变得更加容易。

嵌入式设备开发的一个重要趋势很可能是选择具有某种长期支持的ARM服务器发行版，以及以更高级别语言编写（与云计算行业大体相同的编写方式）的小型服务。Python、Node.js和Go在嵌入式系统行业都有光明的未来。下次您有嵌入式Linux项目时，请考虑使用标准Linux发行版，该发行版提供某种形式的长期支持并会减少您未来的技术债务。

示例 2：用户界面框架

收集一些开源组件并在Linux内核上运行它们是很容易的，这种组合就是Linux发行版。在屏幕上显示配有文字的图片并不是很困难，这就是所谓的小型UI框架。就像维护Linux发行版是一项永无止境的工作一样，编写和维护您的UI框架也将是一项永无止境的工作。请想一想这些需求：正确显示世界各地的语言、可访问性、扩大规模和适应有更多限制的环境，以及随着技术进步对不同渲染系统的支持。维护任何UI框架都没有止境。您可以很容易地在现有UI框架的发展过程中观察到这些。Qt、GTK和EFL至今已有20多年的历史。他们需要数百名开发人员才能达到他们现在的水平，我们应该预料到，在接下来的20年里，他们需要同样的努力程度。React和React Native也需要数百名开发人员，语言更改不会改变处理所有这些外部约束的需求。当你选择了一个UI框架时，要明白你选择了一个社区，并依靠他们来承担相应的技术债务。为了确保这个社区保持健康并不断帮你摆脱债务，你能够介入社区并提供帮助可能是必要的。

另一个建议是需要注意有效的许可证以及任一具体项目的IP资产的拥有者通常是谁。在依赖于某一个 UI 框架、不参与其开发和未支付任何许可费用的前提下，如果您正在制作可视化应用程序，那么这在本质上会削弱您的核心依赖项之一，从而增加您的技术债务。切换框架通常很困难，而且您的应用程序越复杂，更改框架就越困难。就Linux发行版而言，您应该会希望以某种形式参与上游依赖项，以符合您自己的长远需求。对上游的贡献可以采取多种不同的形式，您应该选择与您的业务最匹配的一种（代码、货币、文档、营销等）。

上游开发的作用

形成分支或Fork并进行开发都在意料之中，只要最终回馈上游分支即可。图2说明了这个过程。

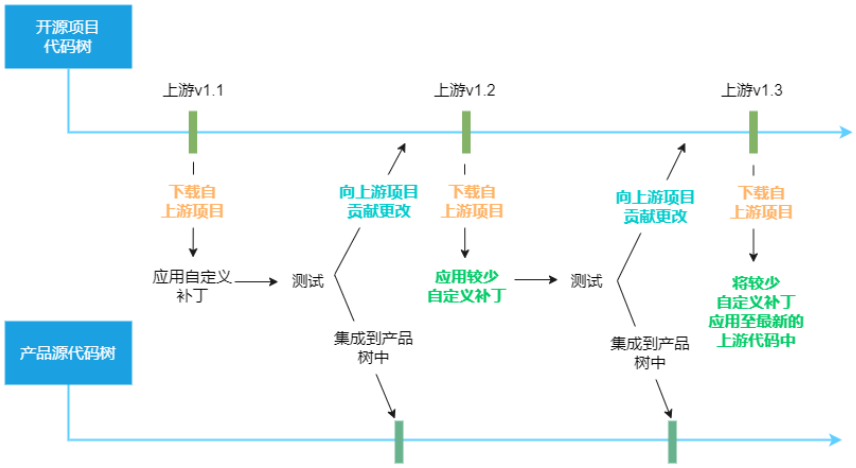


图 2： 整合到上游的开发周期

- **持续集成和持续交付/部署：** 开发人员可以更快地使用功能；新代码更快地出现在开发者树中。以更高的质量水平完成每次代码提交都需要的测试、回归，以及更容易发现漏洞。
- **尽早并经常发布：** 尽早并经常发布实践具有许多已证明的优势。“尽早发布”允许其他人提供反馈并参与开发，鼓励他们提出可能会被吸纳的新想法。同时，代码仍然很灵活，并为其他人在开发进展很久之前就标记问题提供了时间。另一方面，“经常发布”使代码库更改更易于理解、调试和趋于成熟，同时有助于保持快速开发和创新的能力。
- **同行评审：** 尽早分享，即使是在你的代码设计阶段；预计会收到评论请求；具有多层层次结构的子系统/维护模型；代码在发布时通常已经被评审过很多次。代码评审总是在提交之前进行。这能使项目所接受的代码来自更广泛的贡献者（建立信任网络）。
- **正在进行或持续的测试：**
  - 早期发现意味着更快的分类和修复。
  - 较小的更改使故障排除更容易。
  - 较早注意到回归。
  - 帮助子系统维护人员确定接受哪些代码提交。
- **项目可能有多个构建和测试周期。**
  - 构建服务工具可以使过程自动化。
  - 通常与功能冻结密切相关。
- **更易于维护**（不断变化的技术、安全修复）
- **更好的测试、错误报告收集和分析。**
- **专注于模块化设计和架构：** 模块化有几个好处：它允许项目扩展，最大限度地减少对具有较小核心的通用代码的争夺和作为减少冲突的插件实现的功能，引起任务和域的自然分离，允许开发人员更快地使用功能，并且模块化设计（如果设计的好）通常具有较少的相互依赖性和清晰的接口。

上游——努力的统一者

假如做得好，与上游保持一致的开发可以保证您没有或几乎没有技术债务。但是，这需要加入且积极参与上游开源项目。您不能强制上游一直朝着有利于您的方向发展。尽管如此，您仍然可以通过为它做贡献，以及将您的目标/意愿分享给上游并影响其他贡献者来了解您的需求，进而影响上游社区。

大规模解决技术债务

如果您的组织使用成千上百个开源代码包，并对其中许多包进行了修改，那么您将需要考虑一种贡献策略，该策略将允许您将代码贡献回核心或基本组件，以最小化对此类战略组件的技术负债。大规模解决技术债务的方法有哪些？我们将探讨两组能够分别在策略和过程层面以及开发层面上帮助我们实现这一目标的实践。

在策略和过程层面

- 全面批准专门的开源开发人员做出贡献。
- 更快的审批路径使得对上游项目的贡献流动得更快。
- 灵活的 IT 支持使开发人员能够使用所需的工具(如果您无法为开发人员提供专用的 Linux 环境，现在可以使用用于 Linux 和虚拟机的 Windows 子系统)。
- 组织绩效评估，奖励那些遵循既定流程/政策，并致力于最小化技术债务的开发人员。
- 为你保证的时间。

在开发层面

- 通过分享您的愿景和目标，向工程师解释业务目标。最后，他们才是实现它的人，所以他们应该知道你在想什么。
- 确保您的开发人员和加入团队的新成员了解什么是技术债务，以及您为什么选择维护已有的技术债务。
- 对将您的贡献与上游分支合并方面有现实的期望。
- 接受评审/反馈的循环。作为上游开发评审过程的一部分，将会有多个来回的循环。
- 对你的贡献不要太自私。参与到上游的任务中，这些任务在短期内不一定是你的组织直接需要的，但要改善上游的生存能力和健康状况。
- 鼓励您的开发人员在增加技术债务时在代码中明确地留下注释。
- 短期工作，但长远计划。

太晚了，技术债务已经背上了，要‘摆烂’吗？

您所在的组织已经欠下了很多的技术债务，我们上述的情况都有表现，这个时候我们该如何处理？我们都希望有一个能够解决问题的必杀技，可是真实的情况是必杀技并不存在。您要采取的行动，取决于应对不同的情形，不过在进一步之前，可以对照一下下面的情形：

- 摘选出需要保留的功能/特性；
- 确认代码还是有用处的；
- 移除不再使用和维护的代码；
- 减少分支和fork；
- 重构、清理，以符合上游，并合并到上游；

以上这些活动相当的耗时，所以会拖慢您现有的进度，而且还需要专门腾出人手来做这些。在完成上述这些工作的时候，团队也很难添加任何的功能，而且还可能会产生抵触以及更多争议的地方。

现在也有可能存在一个开源项目，它确实或可以提供您需要的功能或某些功能。迁移到该开源项目也可能比尝试处理当前的代码库更有意义。但是，请不要忘记，这个世界并不会因为我们不去关注而停止，千万不要盯着眼前的某些可以暂时满足的功能。所以一定不要让贵司阻止你寻找更具长远意义的解决方案，尤其是当前看起来颇具商业意义。

还有另外一个更为激进的做法：连同现有的代码一道将技术债务一股脑扔掉。可以通过多种方式来完成：贵司的业务已经无力承担维护代码的费用，而且客户也没有多少，此时直接丢弃即可；当然，一定要和客户说明即将废弃的代码，这个过程相当于破产清算，即时止损，也就不再有技术债务。

推荐的实践

在这个小节中，我们将为读者随机的提供一些建议，不过请一定记住，本节的内容并不能完全匹配于您现实中遇到的实际情形，请随机应对。（有一些可能适合，有些可能不行。）因此，您需要运用自己的智慧，针对贵司的情况，灵活的借鉴以下所列出的情形，希望您能有一个最佳效果。

- 采用上游优先哲学；
- 请仔细评估不提交上游的自行维护的代码；
- 要去经常将分支的代码合并回主干；
- 保持内部的工作和上游的发布周期尽可能的一致；
- 允许快速批准上游参与。这一点须通过清晰轻量级的策略和流程来完成，从而促进与上游开发人员的交互；
- 更新考核指标，将与技术债务相关的指标纳入总体绩效目标，以确保由于技术债务成本高或无法接受而无法实现开发目标；
- 为开发人员/经理提供培训，使之拥有识别和缓解技术债务情景的知识；
- 要求所有代码都正确记录，以便上游审阅者更好地理解，这样做是有助于更快的接受周期（另外还要记录为什么代码不合并到上游）；
- 遵循今早发布，频繁发布的实践，不要积累了很多，然后才往上游提交，要去分享设计思路、早期代码，以及在有大量贡献的时候，要保持补丁的独立和小巧；
- 对那些没有提交上游的代码要持续跟进：而且在每次的提交窗口期都要进行评估，一旦发现内部维护代码的代价在上升，则就要去讨论和重新评估原先的决策。

总结

开源在此有着非常重要的作用，将您的开发工作与上游保持对齐，可以让贵司所承担的技术债务量有着直接的影响。正如金融债务涉及支付利息一样，技术债务也有另一种需要承担的利息：它不是无息的！

在很多情况下，技术债务是无法避免的短期性的债务。技术债务是技术人员所负担，而且会是自己日常工作的一部分，每次的决策都可能会产生技术债务。任何工程工作的长期目标都应该是尽量减少和消除任何开发工作造成的技术债务。公司可以通过适当的策略、流程、培训和工具，帮助减轻和指导工程工作，从而降低技术债务。

反馈

我们对任何的改进建议都是充满感激之情的。请直接给作者写信提出意见。

关于作者

Cedric Bail (M.Sc.)

是一名高级软件工程师，目前与EASi签约，专门从事嵌入式Linux。Cedric 曾服务过移动运营商、互联网服务提供商以及最近的一家最大的消费电子公司之一担任各种嵌入式 Linux 平台的软件工程师。他的工作重点一直是针对受限环境优化软件，改进其 API，并根据需要在上游软件中引入新组件以满足业务需求。

- Twitter: @cedric_bail
- LinkedIn: linkedin.com/in/cedricbail/
- Email: cedric.bail@free.fr

Ibrahim Haddad (Ph.D.)

Linux AI/Data 基金会的执行总监，Haddad 曾在 Ericsson Research、Open Source Development Labs、Motorola、Palm、Hewlett-Packard、Linux Foundation 和 Samsung Research 担任技术和产品组合管理职务。

- Twitter: @IbrahimAtLinux
- Web: IbrahimAtLinux.com
- LinkedIn: linkedin.com/in/ibrahimhaddad
- Email: ibrahim@linuxfoundation.org

译者及其完成章节

- 适兕 技术债务
- 适兕 技术债务是如何累积的？
- 适兕 技术债务的处理
- JuliaZhou2022 例1:订制构建系统的角色
- JuliaZhou2022 例2:用户接口框架
- JuliaZhou2022 上游开发的角色
- JayFrank 大规模解决技术债务问题
- 适兕 太晚了，技术债务已经背上了，要‘摆烂’吗？
- 适兕 推荐实践
- 适兕 总结
- 适兕 反馈
- 适兕 关于原作者

校对