技術的負債とオープンソース開発

技術的負債と
それを軽減するのにオープンソース開発がどう役立つかを
よりよく理解するための議論

Ibrahim Haddad, Ph.D.、Cedric Bail, M.Sc.（共著）

出版元：The Linux Foundation（電子書籍版） | 2020年7月

www.linuxfoundation.org

THE LINUX FOUNDATION

概要

このホワイトペーパーは、著者である私たちが自宅に閉じ込められていた 2020 年 3 月中旬、日曜日の朝の電話の会話から生まれました。私たちは、Samsung Research のオープンソース グループ内で共に仕事をし、アップストリームのオープンソース プロジェクトを採用することにより、社内の技術的負債 (Technical Debt) を最小化することを直接経験してきました。その経験は、さまざまなレベルでアップストリーム開発に関与した複数の製品やビジネスユニットに使用された数十のオープンソース プロジェクトに及びます。この論文は、広い範囲で、技術的負債の問題の概要を提供します。技術的負債の特定方法、それを最小限に抑える方法、オープンソース開発の役割、および問題に対処するための戦略についての議論が含まれています。

免責事項

このホワイトペーパーで表明している見解は、すべて著者のものであり、必ずしも現在または過去の雇用主の見解を表しているものではありません。内容に誤りや脱落がある場合は事前にお詫びするとともに、フィードバックや更新を受け付けています。

もくじ

技術的負債 4
定義 4
症状 4
技術的負債の種類 4
一時的な技術的負債 5
未知の技術的負債 5
意図的に作り出された技術的負債 5
陳腐化による技術的負債 5
組織の技術的負債 5
技術的負債のさまざまな原因 5
帰結 6
技術的負債はどのように蓄積されるのか 7
技術的負債の取り扱い 8
技術的負債の特定 8
技術的負債の最小化 8
プログラミング言語の選択 8
エコシステムの選択 9
ディペンデンシーの選択 9
例 1：カスタムビルド システムの役割 10
例 2：ユーザーインターフェイス フレームワーク 11
アップストリーム開発の役割 12
アップストリーム — 取り組みの統合 13
大規模な技術的負債への対処 14
ポリシーとプロセスレベル 14
開発レベル 14
手遅れです。技術的負債はすでに存在しています。私たちは何をすべきでしょうか? 15
お勧めのプラクティス 16
結論 17
フィードバック 18
著者について 18

技術的負債

定義

ソフトウェア開発で使用される用語である技術的負債とは、共同開発が行われているメインブランチにマージされないことから生じるソースコード保守コストを指します。技術的負債を構成するものをより広く解釈すれば、以下のようにプロプライエタリ コードそのものです。

- 単一の組織が開発した。
- その組織だけで保持および保守する必要があるソースコードである。
- 場合によっては、組織はパートナーの力を借りてそのコードを保守し、その負債を背負う。

ここで明らかにしておきたいのは、アップストリーム プロジェクトも開発者のコミュニティでコードを保守しているため、リソースや時間がない場合は、アップストリームのコードであっても技術的負債がないわけではないということです。たとえば、プロジェクトに貢献することなく、またプロジェクトがたった 1 人の人間の空き時間にしかメンテナンスされていないことに気づかずに、OpenSSL プロジェクトに依存していた企業が多くありました。これは、The Linux Foundation が現代のインフラストラクチャにとって重要なオープンソース プロジェクトをサポートする Core Infrastructure Initiative を立ち上げる大きな動機づけになった特徴的な例です。

症状

技術的負債の存在を示す症状をどのように特定しますか。また、それらの症状はどのようなものでしょうか。このセクションでは、経験に基づいてそのような症状をいくつか挙げ、それぞれについて簡単に説明します。これは、網羅的で包括的なリストではなく、最も一般的で広く観察されている技術的負債の症状のリストです。

- **リリース速度の低下**
  新機能が提供されるまでのリリース間隔が長くなります。
- **新規開発者がコードを習得する時間の増加**
  新規開発者が仕事を習得する時間はコードの複雑性と深く関係しています。内部開発者だけがコードベースに精通しているコードはその複雑さのために、新規開発者のコードを習得するための時間が大きく増加します。この症状による第 2 の症状は、開発者の維持や新規開発者の採用が困難になることです。
- **セキュリティ問題の増加**
  少なくとも、メインのアップストリーム ブランチよりも多くのセキュリティ問題が発生しています。
- **コードベースを保守するための労力の増加**
  保守作業は、保守するコードの本体が大きく複雑になるほど、より時間がかかるようになります。
- **アップストリームの開発サイクルとの不整合**
  保守のペースが維持できず、アップストリームの開発やリリース サイクルとも整合できていないことを示しています。

技術的負債の種類

技術的負債にはいくつかの種類があります。 私たちは、これらのさまざまな技技術的負債のタイプを説明するための標準化された、または一般に合意された定義はまだないと理解しています。したがって、このセクションでは、私たちの解釈を提示します。

一時的な技術的負債

チームは、複数のコンポーネント、システム、またはサブシステムに影響を与える可能性のある複雑な機能に取り組んでいる場合があります。開発されたものが、アップストリーム ブランチと統合される必要がある場合、一時的に技術的負債を生み出すことになります。私たちは技術的負債を生み出しているという事実を認識しています。ただし、私たちの目的は製品開発を加速することであり、最終目標は、フォークしたものを別の目的で利用して、後でそれをアップストリーム ブランチとマージすることです。

未知の技術的負債

不適切なエンジニアリングの結果として、無意識のうちに技術的負債を生み出していることがあります。例として、アップストリーム ブランチにも受け入れられず、どこか他のところで再利用する候補にもならない、拙劣なコードです。このようなコードの処置には困ります。

意図的に作り出された技術的負債

この例外的なタイプの技術的負債は意図的に作成されるものです。例としては、特定の機能を、広くコミュニティと共有せず、その組織内でのみ保守するケースが挙げられます。結果として、そのような組織は、アップストリーム ブランチにマージせずに独立性を保つために、このフォークを作成していることになります。時の経過とともに、そのフォーク部分は拡大し、技術的負債も大きくなり、関連する保守コストが増大することになります。

陳腐化による技術的負債

陳腐化による技術的負債は、「独自技術症候群 (Not Invented Here syndrome)」により生み出されたり、または、より広いコミュニティに利益をもたらす可能性のある新しいコンポーネントを独自に開発したりした結果生じるユニークなケースです。技術的な管理ミスやリーダーシップの欠如により、開発されたものを一組織のみで使用している場合です。しかし、この間に、世界では、問題の解決が進み、組織が開発したものと互換性のないデファクト ソリューションや標準が生み出されています。この状況では、独自に開発したものは、陳腐化により技術的負債のもとになります。

組織の技術的負債

企業が開発するソースコードがどのようにその企業の組織によくマッチするかについては、広く議論されています（これは非常に興味深い見解です）。コードが開発されるべきだった組織と、最終的にコードが開発された組織が一致しないこともあります。開発者がマネージャーに反論できず、適切な人がコードを開発できなかったり、適切なコードにすることに寄与できなかったりすると、結果として、そこに存在すべきででもない、誰も使おうとしないコードが作成されてしまうことが起こります。これが技術的負債です。

技術的負債のさまざまな原因

技術的負債を生み出し、増加させる要因は多数あります。このセクションでは、最も一般的な原因について、それぞれ簡単に説明します。

- **低品質のコード**
  対象としているオープンソース プロジェクトによって設定されているコード品質基準を満たさないなど、何らかの理由でアップストリームに取り込まれない低品質のコード。このようなコードは、一般に「スパゲッティ コード」と表現されることもあります。

- **独善的なコード**
  一般的なコミュニティであまり使用されることがなく、コードを提供している特定の企業にのみに有益な独善的なコード。このようなコード（または機能）は、通常はアップストリームに受け入れられず、一般的なユースケースや多くの利用者に利益をもたらすように調整するよう勧告を受けます。

- **連携していない開発体制（開発の断片化）**
  重複した作業や、競合する実装を作り出します。

- **コードをアップストリームに入れる努力の欠如**
  多くの場合、コードを作成するチームの組織には、コミュニティと連携してコードをアップストリーム ブランチに受け入れられるようにするための余力が十分にありません。これは、短期的には効率的かもしれませんが、技術的負債を作り出し、コードの保守と維持管理に長期的な悪影響を生み出します。

- **影響範囲などを考慮しない侵入型コード**
  複数のコンポーネント、システム、サブシステムにわたって、追加の影響調整を必要とします。

- **アップストリームにコードを受け入れてもらうための時間**
  コードが受け入れられ、アップストリーム ブランチにマージされるまでの間、一時的な技術的負債を生じます。長期的には悪影響はありません。

- **テストが不十分**
  完全なテスト カバレッジが妨げられ、提出コードが受け入れられない場合です。

- **ドキュメントが不足している、または最新でない**

- **技術的リーダーシップが欠如し、技術コミュニティとの関わりが不十分**
  コミュニティから外れることになり、後々、先行しているレベルに追いつく作業が必要になります。

- **継続的な要件変更**
  どの組織の社内開発作業でも見られます。

- **非標準のテクノロジー、または既存の標準との整合性欠如**

- **組織としての無頓着さ**
  技術的リーダーシップの欠如とアップストリーム開発の技術的方向性を認識しないという 2 つの側面を組み合わせた「組織としての無頓着さ」。この状況は、開発作業の必要性が増大している非デジタルネイティブ企業でますます広がっています。

帰結

生成された技術的負債は、開発作業に以下のような悪影響を及ぼします。

- コードの保守コストが膨らむ。
- イノベーションと開発サイクルが遅くなる。
- 負債利子の支払い – 技術的負債の支払いは、メイン ブランチ、競合、および世の中に追いつくために必要な追加の開発の実施という形で行われる。
- メイン ブランチの新機能を見落とす可能性や、フォークしたブランチにすべての新機能を内部的にバックポートしなければならない可能性がある。
- 内部のブランチとパブリック ブランチとの差分が大きすぎるため、メイン ブランチとの重複作業が生じる。

考えられる最悪のケースは、組織が頻繁に保守している自身のフォークのコードベースを長期に保守し続けるための労力が必要になることです。

技術的負債はどのように蓄積されるのか

Arthur Bloch はアメリカ人の作家であり、「マーフィーの法則」の本の著者です。「友は現れては消える（増えも減りもしない）ものだが、敵は増えていく一方だ」と彼は述べています。敵と同じように技術的負債は蓄積されていくので、技術的負債を議論するときに参考になる名言です。技術的負債はどのようにして起こるのでしょうか。図 1 は、技術的負債がどのように生成され、引き継がれていくのかを説明する基本的なシナリオを提示しています。

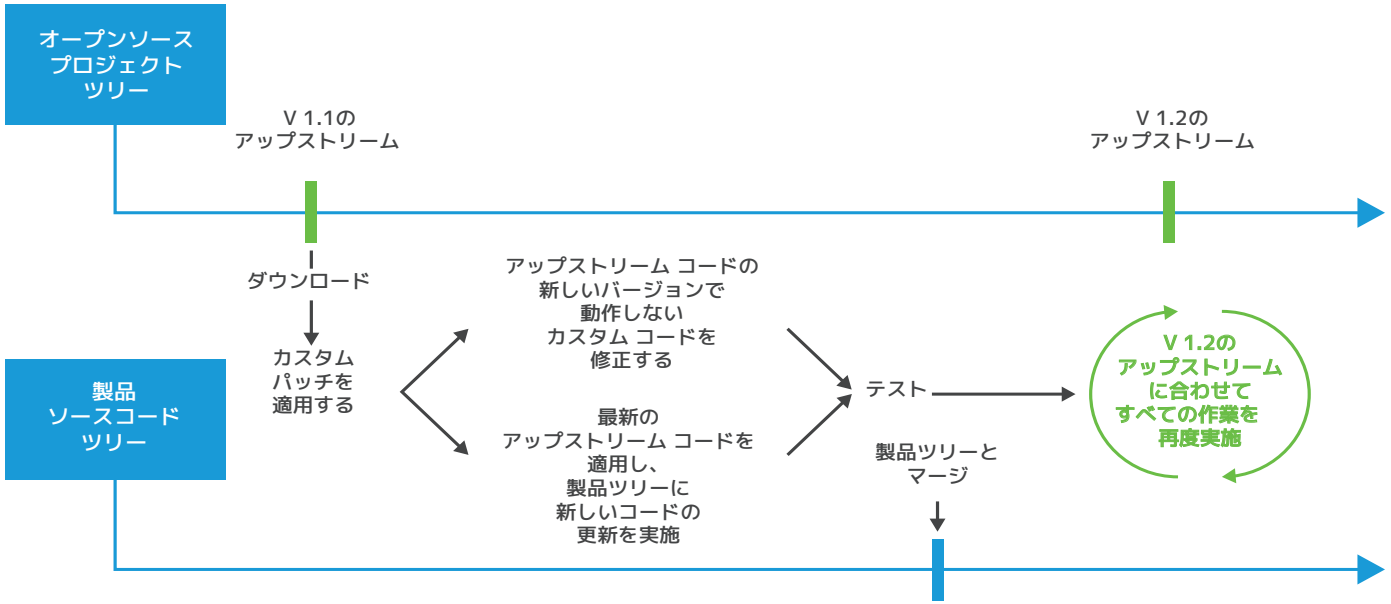

図 1：アップストリームと統合しない開発サイクル

技術的負債の取り扱い

技術的負債の特定

特定のビジネスニーズに最も適したエンジニアリング時間を費やして作成するという点で、コードのすべての行は、将来、技術的負債になる可能性を秘めています。これを理解することは、おもにあなたのビジネス目標とあなたのチームが何に時間を費やしているかを理解することです。以下のいくつかの質問に答えることは、この問題を考える練習に役立つでしょう。

- チームは何に取り組んでいるのか
- あなたの組織のソフトウェア スタックに関して、新規採用者に何を伝える必要があるか
- そのソフトウェア スタック全体をどれくらいの頻度で更新できるか
- あなたは自分が依存しているすべてのソフトウェアを知っているか
- 通常どのような問題が発生するか
- どのようなディペンデンシー（依存関係）に対処するのが最も困難か
- 使用しているソフトウェア スタックの各コンポーネントの代替になり得るものは何だったか
- それらの代替になり得る各コンポーネントのコミュニティは、どの程度アクティブか
- コミュニティが消滅したり、アクティブでなくなったりした場合、そのコンポーネントを保守していくためにはどれだけの作業が必要か
- そのコンポーネントに切り替えるのに、どの程度の労力が必要か
- 顧客やエンジニアから報告された問題をデバッグするのは、どれくらい大変か

これらの一連の質問は、技術的負債とその盲点がどこにあるかについて議論する上で役に立つでしょう。

技術的負債の最小化

今解くべき核心的な質問は、いかにして技術的負債を最小限に抑え、開発作業への影響を最小限に抑えるかです。

プログラミング言語の選択

製品開発に使用されるプログラミング言語や開発フレームワークは、開発者にある程度の制約を与えます。言語やフレームワークが複雑であるほど、それを扱える要員を確保維持することは難しくなります。これは、ソフトウェアをビルドしているシステムの持つ制約の程度と、仕事を遂行することができるレベルの開発者をどの程度確保維持できるか、との間のバランスになります。一般的には、ほとんどの場合、次の理由で、高水準言語[1]を使用するのがよいでしょう。

- 成果を出せる開発者の採用を促進する。
- サードパーティーによる問題の見極めや解決が容易である。
- コミュニティが参加できるため、移植性があり保守しやすいものになる。
- 「誤り」に対する耐性を高め、より簡単な解決策の提供を可能にする。
- ドキュメント、チュートリアル、既存のソリューションが、有効なオンライン情報源として存在する。

1 どの高水準プログラミング言語を選択するかは状況により異なりますが、たとえば、Go、Python、C#、Typescriptなどをお勧めします。

エコシステムの選択

アプリケーションは常に、言語、フレームワーク、オペレーティングシステムのソフトウェア スタックの上に構築されます。これは、アプリケーションが共存するエコシステムの典型的な構造や構成です。このエコシステムが技術的負債を形成するようになるので、開発者は、エコシステムと自身の目標をどのように協調させていくかについて理解している必要があります。例えば以下のような点です。

- Linux ディストリビューションは異なったサポートの条件を持っており、長期にわたって、さまざまなレベルの API、ABI、セキュリティを保証しています。
- プログラミング言語の選択は、長期にわたってソフトウェアを実行する能力に影響を与えます。その選択はランタイムやそのビルド環境の進化により、コードを実行する能力に影響を与えます。また、それらの変更を管理するための新しい開発者を見つける能力に制約が生じる可能性があります。
- 最新の言語とフレームワークは、Linux ディストリビューションをバイパスしてソフトウェアをパッケージ化する傾向もあります。この要因は、Linux ディストリビューションと同じようにソフトウェアに長期的に影響を与える可能性があります。したがって、選択するものは、API/ABI/ セキュリティのニーズに従っている必要があります。

ディペンデンシーの選択

世の中の進歩に合わせて、より高品質なオープンソース ソフトウェアが利用可能になっていきます。これは継続的なプロセスであり、あなたの組織が開発している各ソフトウェアもこの点から評価することが重要です。これらのディペンデンシー（依存関係）は必要な作業を簡素化し、より複雑な機能やソリューションの構築を可能にします。それでも、関連するコミュニティが十分に健全でなければ、それらも技術的負債になる可能性があります。適切なディペンデンシーを選択し、重要なディペンデンシーのコミュニティに貢献することで、多くの人とそれを共有して技術的負債を削減することができます。

また、数年前のディペンデンシーの価値は、もはや正しくない状況になっているかもしれません。 技術的負債を扱うのと同じように、ディペンデンシーについても、継続的に評価していく必要があるということです。

例 1：カスタム ビルド システムの役割

オペレーティング システムの保守は、フルタイムの仕事であるだけでなく、以下のことを確認する必要があります。

- 長期間にわたって確実かつ安全に動作し、
- 各コンポーネントのビルド作業は、時間が経過しても再現性があり、
- 各コンポーネントは適切に評価およびテストされ、
- ライセンスは尊重され、そして
- 堅牢で安全な更新メカニズムがあること。

今では、Linux 環境の構築は数時間で完了できるようになっており、この種の作業は過小評価されています。ただし、この作業は、オペレーティング システムを長期間保守するための最初のステップにすぎません。今日では、ARM がサーバー市場に参入したことで、組み込みデバイス向けのディストリビューションを、より簡単に標準化できるようになりました。Debian、Ubuntu、Redhat、SuSE などが、どの組み込みデバイス上でもそのまま問題なく、またはわずかな微調整で実行できる ARM サーバー ディストリビューションを提供しており、カスタム オペレーティング システムやそれに関連したパッケージ ビルドを保守する必要はなくなっています。また、クラウドという 1 つの環境から組み込み市場に知識を移転するための標準化されたツールも開発者に提供されています。採用担当マネージャーは、より大きな市場である Linux サーバー市場に参入できるため、組み込みシステムで働ける開発者をより簡単に見つけることができるようになりました。

組み込みデバイス開発における重要なトレンドは、何らかの「長期サポート」を備えた ARM サーバー ディストリビューションと、クラウド業界で採用されているのと同じような高水準言語で書かれた小さなサービスを選択することです。Python、Node.js、および Go は、すべて組み込みシステム業界で明るい未来を持っています。組み込み Linux プロジェクトを持つ機会があれば、次回からは、何らかの形で「長期サポート」を提供し、将来の技術的負債を削減してくれる標準の Linux ディストリビューションの使用を検討してください。

例 2：ユーザー インターフェイス フレームワーク

いくつかのオープンソース コンポーネントを収集し、それらを Linux カーネル上で実行し、その集合体を Linux ディストリビューションと名付けることは簡単です。画面にテキストを含む画像を表示し、それを小さな UI フレームワークと呼ぶことも、それほど難しいことではありません。そして、Linux ディストリビューションの保守が終わりのない仕事であるのと同じように、UI フレームワークの作成と保守も終わりのない作業になります。世界中の言語を正しく表示する必要性、アクセシビリティの必要性、スケールアップしてより制約のある環境に適合させる必要性、そして、世の中が、より良いテクノロジーに移行するのに合わせてさまざまなレンダリング システムをサポートする必要性を考えてみてください。UI フレームワークの保守には終わりがありません。既存の UI フレームワークの開発において、これらの状況を容易に知ることができます。Qt、GTK、EFL は、すでに 20 年以上前から存在しています。何百人もの開発者の功績により、これらのフレームワークにたどり着きましたが、今後 20 年間も、同じレベルの努力が必要になると予想されます。React や ReactNative も同様に、何百人もの開発者を必要としており、言語を変更しても、このような外部制約のすべてに対処する必要性は変わりません。UI フレームワークを選択するということは、コミュニティを選択することで、また、コミュニティが技術的負債を負ってくれることを当てにしていることと理解してください。コミュニティが健全な状態を保ち、その負債を肩代わりし続けてくれるようにコミュニティに参加して手助けできることが必要かもしれません。

また、適用されているライセンスや、だれがどのプロジェクトの IP 資産を所有しているかについて、留意してください。その開発に参加したり、ライセンス料を支払ったりすることなく、UI フレームワークに依存している場合は、あなたは、ビジュアルアプリケーションを作成し、技術的負債を増やしているので、本質的にコア ディペンデンシーの 1 つを弱体化させていることになります。通常、フレームワークの切り替えは難しく、アプリケーションが複雑になるほど、フレームワークの変更は困難になります。Linux ディストリビューションに関しては、長期的にあなたのニーズに合わせるために、アップストリームのディペンデンシーに何らかの形で関与していく必要があります。アップストリームへの貢献はさまざまな形（コード、金銭、ドキュメント、マーケティングなど）をとることができ、あなたのビジネスに最も適したものを選択してください。

アップストリーム開発の役割

最終目標がアップストリームのブランチに貢献することであるかぎり、ブランチアウトまたはフォークして開発を進めることになるでしょう。図 2 は、このプロセスを示しています。

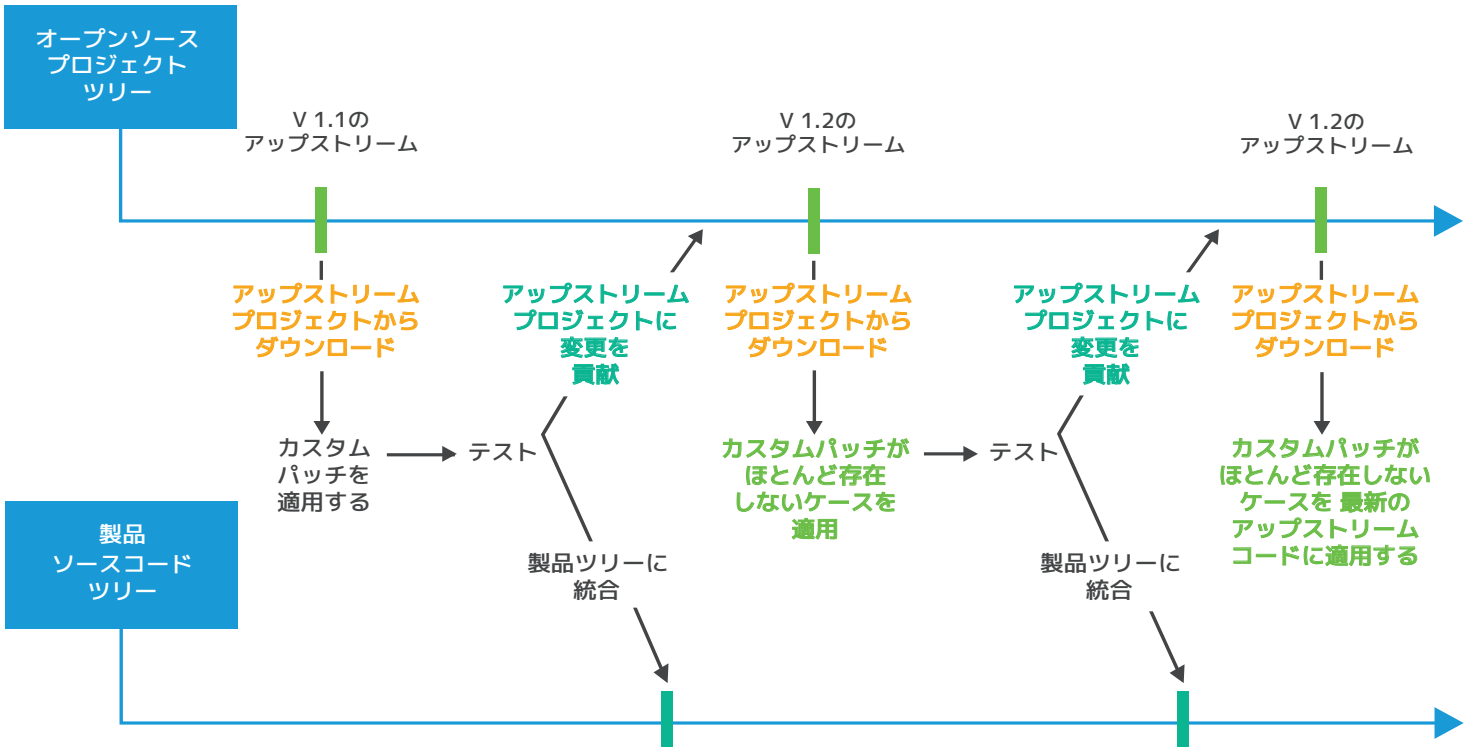

図 2 : アップストリームへの統合を伴う開発サイクル

- **継続的インテグレーションと継続的デリバリー / デプロイメント:** 開発者は機能をより迅速に利用できます。新しいコードが開発者ツリーに迅速に取り込まれます。各コード コミットには、より高いレベルの品質でテストが必要で、リグレッションやバグの発見がより容易になります。

- **早期かつ頻繁にリリース：**早期かつ頻繁にリリースすることは、数多くの利点が実証されています。「早期リリース」することにより、新しいアイデアを取り入れることを容易にし、他の人がフィードバックを提供したり、開発に参加したりすることができるようになります。

  同時に、コードに柔軟性を残し、開発が先に進み過ぎる前に、他のユーザーが問題を報告する時間ができます。一方、「頻繁にリリース」することで、コードベースの変更がわかりやすく、デバッグしやすくなり、成熟度が高まるとともに、開発とイノベーションの速いペースも維持しやすくなります。

- **ピアレビュー：**たとえコードの設計段階であっても、できるだけ早い時期に共有しましょう。RFC: Request for Comments の段階が予想されます。マルチレイヤー階層のサブシステム / メンテナー モデルでは、コードがリリースされるまでに、通常は何度もレビューされます。コードはコミットされる前に必ずレビューを受けます。プロジェクトが、より広範囲の貢献者からコードを受け入れることができるようになります（信頼の輪を構築します）。

- **継続的なテスト：**

  - 早期発見は、より迅速なトリアージと修正を意味します。
  - 小さな変更は、トラブルシューティングを容易にします。
  - リグレッションをより早く発見できます。
  - ・サブシステムのメンテナーが、受け入れるべきコード サブミッションを決定するのに役立ちます。

- **プロジェクトが複数のビルド サイクルやテスト サイクルを持てる：**

  - ビルド サービス ツールでプロセスを自動化できます。
  - 通常は、機能のフリーズと厳密に連携しています。

- **より簡易な保守**（テクノロジーの変更、セキュリティ修正）
- **より優れたテスト、バグレポートの収集、および分析**
- **モジュール設計とアーキテクチャに焦点を当てる：**
  モジュール化にはいくつかの利点があります。つまり、プロジェクトは拡張性を持つことができるようになり、より小さいコアで共通コードの競合を最小限に抑え、プラグインとして実装された機能が衝突を減らし、タスクとスコープを自然に分離し、開発者がより迅速に機能を利用できるようになります。よくできたモジュール設計は、多くの場合、明確なインターフェイスを実装し、相互依存性をほとんど持たないようになっています。

アップストリーム — 取り組みの統合

うまくいけば、アップストリームと連携して開発することは、技術的負債がほとんどゼロになることを保証してくれます。ただし、これにはアップストリームのオープンソース プロジェクトに関与し、積極的に参加する必要があります。アップストリームがあなたに利益をもたらす方向に進むように強要することはできません。それでも、あなたがアップストリームに貢献し、あなたの目標や意志をアップストリームと共有し、他の貢献者にあなたのニーズを理解してもらうことで、アップストリームに影響を与えることができます。

大規模な技術的負債への対処

あなたの組織が数百・数千のオープンソース パッケージを使用しており、それらの多くに変更を加えている場合、そのような戦略的コンポーネントに対する技術的負債を最小限に抑えるために、コア コンポーネントや基本的なコンポーネントにコードを貢献できるような貢献戦略を検討する必要があります。大規模な技術的負債に対処する方法にはどのようなものがあるのでしょうか。「ポリシーとプロセスのレベル」と「開発レベル」でそれを支援する 2 つのプラクティス セットを詳しく見てみましょう。

ポリシーとプロセスのレベル

- オープンソース開発専任者の貢献は包括的承認（blanket approval）とします。
- アップストリーム プロジェクトへの貢献がより早く処理されるような高速承認パスを用意します。
- 柔軟なITサポートを提供することにより、開発者は必要なツールを利用できるようになります（開発者専用の Linux 環境を持つことができない場合でも、今では、Windows Subsystem for Linux や仮想マシンを活用できます）。
- 設定されたプロセス / ポリシーに従って技術的負債の最小化に取り組んでいる開発者が報われるような、パフォーマンス レビュー体制を作ります。
- 開発者に、アップストリーム プロジェクトの作業をするための時間を保証します。

開発レベル

- あなたのビジョンや目標をエンジニアと共有して、エンジニアにビジネス目標を明確に伝えます。最終的に彼らがそれを実装するわけですから、、彼らはあなたが何を考えているのかを知っているべきです。
- 開発者やチームに加わる新しいメンバーに、技術的負債とは何か、そしてすでに持っている技術的負債をなぜ維持しているのかを確実に理解してもらうようにしましょう。
- アップストリームのブランチに貢献をマージすることに関して、現実的な期待を持ちましょう。
- レビュー / フィードバックのサイクルを受け入れましょう。アップストリーム開発のレビュープロセスの一環で、前進したり、後退したりする (back-and-forth) サイクルが複数回生じることもあります。
- 自身の貢献にあまりにも利己的になってはいけません。あなたの組織が短期的に必ずしも直接必要としていなくても、アップストリームの活性化や健全性を高めるようなタスクについては、アップストリームと協力して関わってください。
- 開発者が技術的負債を追加する場合は、コードに明示的にコメントを残すように奨励してください。
- 仕事は短期的な視点で行いますが、計画は長期的な視点で立案しましょう。

手遅れです。技術的負債はすでに存在しています。私たちは何をすべきでしょうか?

あなたが働いている組織には、すでに多くの技術的負債が蓄積されており、あらゆる症状が出ています。今、あなたはそれについて何をすべきでしょうか。解決のための特効薬を望んでも、それはありません。採用すべきアプローチは、要因によって異なります。以下のオプションを試してみてください。

- 保守する必要のある機能を選択する。
- 今も有効活用しているコードを特定する。
- 保守していないコードや使用していないコードを削除する。
- ブランチ / フォークの必要性を減らす。
- アップストリーム化できるコードをリファクタリングし、クリーンにし、アップストリーム化する。

これらの作業は時間がかかるため、開発者がこの作業に専念すると、開発サイクルが遅くなります。この作業実施中は、組織が機能を追加することも困難になり、組織内で物議を醸したり、進行中の作業のブレーキになったりする可能性があります。

あなたが必要としている機能やその一部を提供しているか提供できるオープンソース プロジェクトが、すでに存在しているかもしれません。そのプロジェクトに移行することは、あなたのコードベースを維持するよりも、理にかなっているかもしれません。あなたが知らない間にも世の中は動いていて、今あなたが必要としているものがすでに存在しているかもしれない、と言うことを忘れてはいけません。組織の誇りが他のソリューションを検討することを妨げないようにして、ビジネス上、最も意味のあるソリューションを採用してください。

さらに過激な別のアプローチは、技術的負債に見切りをつけて、すべてのコードを捨て去ることです。これには、複数の方法があります。あなたのビジネスがまだコードを保守していく余裕がなく、そのコードの存在を正当化するのに十分な顧客もいない場合は、そのコードを削除しましょう。それを行うためには、顧客に、そのコードは廃止されることを明確に伝えましょう。これは、技術的負債に対して破産を宣言し、支払いを停止することと同じです。

お勧めのプラクティス

このセクションでは、推奨されるプラクティスのリストを順不同に提示しています。このセクションで提示しているすべてのものが、特定の状況や組織に当てはまるわけではありません。適用できる場合もあれば、適用できない場合もあります。したがって、社内で採用すべき適切なプラクティスを自己評価し、提示されたプラクティスをどのように調整すれば最善の結果が得られるのかも自己評価することが不可欠です。

- 「アップストリーム ファースト」の指針を採用する。
- アップストリームに行かないカスタム コードを慎重に評価する。
- 常に、メイン ブランチにマージすることを計画する。アップストリームにマージする計画があるかぎり、一時的なフォークやサイド ブランチは問題ない。
- 社内の開発作業はアップストリーム ブランチのリリースと歩調を合わせて行う。
- アップストリームへの貢献を迅速に承認する。明確で軽量なポリシーとプロセスがあれば、アップストリームの開発者とのやりとりが容易にできる。
- 技術的負債に関連するメトリックを全体的なパフォーマンス目標の一部として組み込むようにパフォーマンス メトリックを更新し、技術的負債コストが高いまたは許容できなければ開発目標が達成できないことを明確にする。
- 技術的負債の状況を明らかにし、それを軽減するために、開発者 / マネージャーをトレーニングする。
- アップストリームのレビュー担当者がよりよく理解できるように、すべてのコードについて、適切に文書化する必要がある。受け入れサイクルの迅速化にも役立つ(コードがアップストリーム化されない理由についても文書化する)。
- 「リリースは早期に、高頻度で」を実践する。巨大なコードベースを作成した後、それをアップストリーム化することを決定しない。設計や初期コードも共有し、小さく独立性の高いパッチ群で構成された大きな貢献を提出する。
- アップストリーム化しないことを選択したコードを追跡する。あらゆる機会でその選択を再評価する。内部で保守されているコードのサイズに増加傾向があれば、以前の決定についての議論と再評価が必要になる。

結論

オープンソースには重要な役割があり、開発作業をアップストリームのオープンソース プロジェクトに合わせることで、組織が抱える技術的負債の量に直接的なプラスの影響を与えることができます。金融債務には利息の支払いが必要なのと同じように、技術的負債債務には、別の種類の利息があり、それを支払う必要があります。利子がないということはありません。

多くの場合、短期的には、技術的負債は避けられないものです。技術的負債を抱えることは、たいていは開発者が常に決断する必要があることです。エンジニアリング作業の長期的な目標は、開発作業に起因する技術的負債を最小限に抑え、解消することであるべきです。適切なポリシー、プロセス、トレーニング、およびツールを使用することで、組織は技術的負債の削減に向けたエンジニアリングの取り組みを軽減し、指導できるようになります。

フィードバック

改善のための提案をいただければ幸いです。著者に直接コメントを送ってください。

著者について

Cedric Bail (M.Sc.) は、シニア ソフトウェア エンジニアで、現在は、組み込み Linux を専門とする EASi の仕事をしています。さまざまな組み込み Linux プラットフォームのソフトウェア エンジニアとして、モバイルオペレーター、インターネット サービス プロバイダー、そして最近は、世界最大の家電メーカーの 1 つで働いています。彼は、常に、制約のある環境に合わせてソフトウェアを最適化し、API を改善し、ビジネス要件に合わせて、アップストリーム ソフトウェアに必要な新しいコンポーネントを加えることにフォーカスしています。

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

Ibrahim Haddad (Ph.D.) は、LF AI Foundation のエグゼクティブディレクターです。Ericsson Research、Open Source Development Labs、Motorola、Palm、HewlettPackard、The Linux Foundation、Samsung Research でテクノロジーとポートフォリオ マネージメントの仕事をしてきました。

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

The Linux Foundation は、オープンソースがクローズドなプラットフォームと競い合う上で必要な統合されたリソースやサービスを提供し、Linux の普及推進、保護、および標準化に努めています。

The Linux Foundation やそのプロジェクトの詳細については、www.linuxfoundation.org をご覧ください。