# 弃用 (Deprecation)

Hyrum Wright (Edited by Tom Manshreck)

2021 年 12 月 19 日

## 目录



## 一、 前言

本章由海鲁姆 · 赖特撰写，由汤姆 · 曼什雷克编辑

I love deadlines. I like the whooshing sound they make as they fly by. —Douglas Adams

我喜欢万事都有一个截止日期。我喜欢它们飞过时发出的嗖嗖声。

-道格拉斯 · 亚当斯如是说。

All systems age. Even though software is a digital asset and the physical bits them- selves don’t degrade, new technologies, libraries, techniques, languages, and other environmental changes over time render existing systems obsolete. Old systems require continued maintenance, esoteric expertise, and generally more work as they diverge from the surrounding ecosystem. It’s often better to invest effort in turning off obsolete systems, rather than letting them lumber along indefinitely alongside the systems that replace them. But the number of obsolete systems still running suggests that, in practice, doing so is not trivial. We refer to the process of orderly migration away from and eventual removal of obsolete systems as deprecation.

所有系统都会老化。虽说软件是一种数字资产，它的字节位本身不会有任何退化。但随着时间的推移，新技术、库、技术、语言和其他环境变化，都有可能使现有的系统过时。旧系统需要持续维护、深奥的专业知识，通常需要花费更多的精力，因为它们与周遭的生态略有不同。投入些精力废弃掉过时的系统通常是个不错的选项，

让它们无限期地与它的替代者共存通常不是明智的选择。从实践的角度出发，那些仍在运行的大量过时系统无不表明，废弃掉过时系统所带来的收益并非微不足道。我们将有序迁移并最终移除过时系统的过程称为弃用。

Deprecation is yet another topic that more accurately belongs to the discipline of software engineering than programming because it requires thinking about how to manage a system over time. For long-running software ecosystems, planning for and executing deprecation correctly reduces resource costs and improves velocity by removing the redundancy and complexity that builds up in a system over time. On the other hand, poorly deprecated systems may cost more than leaving them alone. While deprecating systems requires additional effort, it's possible to plan for depreca- tion during the design of the system so that it's easier to eventually decommission and remove it. Deprecations can affect systems ranging from individual function calls to entire software stacks. For concreteness, much of what follows focuses on code-level deprecations.

“弃用”, 严格意义上说, 它不是一个开发层面的议题, 而应归类于软件工程学范畴。因为它需要考虑如何随着时间的推移来管理系统。对于长期运行的软件生态系统，正确规划执行“弃用”, 可以通过消除系统中随时间累积产生的冗余、复杂性等, 来降低资源成本并提高速度。另一方面，不推荐使用的系统可能比不理会它们的成本更高。虽然“弃用”系统需要额外花费精力，但可以考虑在系统设计期间有计划地“弃用”，可以更容易地实现彻底停用并删除废弃的系统。“弃用”的影响范围可大可小，小到单个函数，大到整个软件生态。接下来的大部分内容我们都将集中在代码层级“弃用”上。

Unlike with most of the other topics we have discussed in this book, Google is still learning how best to deprecate and remove software systems. This chapter describes the lessons we've learned as we've deprecated large and heavily used internal systems. Sometimes, it works as expected, and sometimes it doesn't, but the general problem of removing obsolete systems remains a difficult and evolving concern in the industry.

与我们在本书中讨论的大多数章节不同，Google 仍在学习如何最好地“弃用”和删除软件系统。本章主要介绍我们在“弃用”大型和大量使用的内部系统时学到的经验教训。有时，它能符合预期，有时则不会。毕竟移除过时系统的普遍问题, 仍然是行业中一个困难且须不断探索的问题。

This chapter primarily deals with deprecating technical systems, not end-user prod- ucts. The distinction is somewhat arbitrary given that an external-facing API is just another sort of product, and an internal API may have consumers that consider themselves end users. Although many of the principles apply to turning down a pub- lic product, we concern ourselves here with the technical and policy aspects of depre- cating and removing obsolete systems where the system owner has visibility into its use.

本章主要从技术层面讲“弃用”，而不是从产品层面。考虑到面向外部的 API 也算另一种产品，而内部 API 通常是自产自销，因此这种区别有些武断。尽管许多原则也适用于对外产品，但我们在这里关注的是“弃用”和删除过时的内部系统的技术和政策方面的问题。

## 二、 为什么要“弃用”(Why Deprecate?)

Our discussion of deprecation begins from the fundamental premise that code is a lia- bility, not an asset. After all, if code were an asset, why should we even bother spend- ing time trying to turn down and remove obsolete systems? Code has costs, some of which are borne in the process of creating a system, but many other costs are borne as a system is maintained across its lifetime. These ongoing costs, such as the opera- tional resources required to keep a system running or the effort to continually update its codebase as surrounding ecosystems evolve, mean that it's worth evaluating the trade-offs between keeping an aging system running or working to turn it down.

我们对“弃用”的讨论始于这样一个基本前提，即代码是一种负债，而不是一种资产。毕竟，如果代码是一种资产，我们为什么还要费心去尝试“弃用”它呢？代码有成本，其中有开发成本，但更多的是维护成本。这些持续的成本，例如保持系统运行所需的运营资源或紧跟周围生态而不断更新迭代花费的精力，意味着你需要在继续维护老化的系统运行和将其下线之间做一个权衡。

The age of a system alone doesn't justify its deprecation. A system could be finely crafted over several years to be the epitome of software form and function. Some soft- ware systems, such as the LaTeX typesetting system, have been improved over the course of decades, and even though changes still happen, they are few and far between. Just because something is old, it does not follow that it is obsolete.

“弃用”并不能简单地用项目的年限来定夺。一个系统可以经过数年精心打造，才能稳定成熟。一些软件系统，比如 LaTeX 排版系统，经过几十年的改进，虽然它仍在发生变化，但已经趋向稳定了。系统老旧并不意味着它过时了。

Deprecation is best suited for systems that are demonstrably obsolete and a replace- ment exists that provides comparable functionality. The new system might use resources more efficiently, have better security properties, be built in a more sustaina- ble fashion, or just fix bugs. Having two systems to accomplish the same thing might not seem like a pressing problem, but over time, the costs of maintaining them both can grow substantially. Users may need to use the new system, but still have dependencies that use the obsolete one.

“弃用”最适合那些明显过时的系统，并且存在提供类似功能的替代品。新系统可能更有效地使用资源，具有更好的安全属性，以更可持续的方式构建，或者只是修复错误。拥有两个系统来完成同一件事似乎不是一个紧迫的问题，但随着时间的推移，维护它们的成本会大幅增加。用户可能需要使用新系统，但仍然依赖于使用过时的系统。

with the old one. Spending the effort to remove the old system can pay off as the replacement system can now evolve more quickly. The two systems might need to interface with each other, requiring complicated transformation code. As both systems evolve, they may come to depend on each other, making eventual removal of either more difficult. In the long run, we’ve discov- ered that having multiple systems performing the same function also impedes the evolution of the newer system because it is still expected to maintain compatibility with the old one. Spending the effort to remove the old system can pay off as the replacement system can now evolve more quickly.

这两个系统可能需要相互连接，需要复杂的转换代码。随着这两个系统的发展，它们可能会相互依赖，从而使最终消除其中任何一个变得更加困难。从长远来看，我们发现让多个系统执行相同的功能也会阻碍新系统的发展，因为它仍然需要与旧的保持兼容性。由于替换系统现在可以更快地发展，因此花费精力移除旧系统会有相关的收益。

> Earlier we made the assertion that “code is a liability, not an asset.”If that is true, why have we spent most of this book discussing the most efficient way to build software systems that can live for decades? Why put all that effort into creating more code when it’s simply going to end up on the liability side of the balance sheet?
>
> Code itself doesn’t bring value: it is the functionality that it provides that brings value. That functionality is an asset if it meets a user need: the code that implements this functionality is simply a means to that end. If we could get the same functionality from a single line of maintainable, understandable code as 10,000 lines of convoluted spaghetti code, we would prefer the former. Code itself carries a cost—the simpler the code is, while maintaining the same amount of functionality, the better.
>
> Instead of focusing on how much code we can produce, or how large is our codebase, we should instead focus on how much functionality it can deliver per unit of code and try to maximize that metric. One of the easiest ways to do so isn’t writing more code and hoping to get more functionality; it’s removing excess code and systems that are no longer needed. Deprecation policies and procedures make this possible.
>
> 前面，我们断言“代码是一种负债，而不是一种资产”。如果这是真的，为什么我们用本书的大部分时间来讨论构建可以存活数十年的软件系统的最有效方法？当它最终会出现在资产负债表的负债方时，为什么还要付出所有努力来创建更多代码呢？代码本身不会带来价值：它提供的功能带来了价值。如果该功能满足用户需求，那么它就是一种资产：实现此功能的代码只是实现该目的的一种手段。如果我们可以从一行可维护、可理解的代码中获得与 10,000 行错综复杂的意大利面条式代码相同的功能，我们会更喜欢前者。代码本身是有成本的——代码越简单，同时保持相同数量的功能越好。与其关注我们可以生产多少代码，或者我们的代码库有多大，我们应该关注每单位代码可以提供多少功能，并尝试最大化该指标。最简单的方法之一就是不要编写更多代码并希望获得更多功能；而是删除不再需要的多余代码和系统。“弃用”政策的存在就是为了解决这个问题。

Even though deprecation is useful, we’ve learned at Google that organizations have a limit on the amount

of deprecation work that is reasonable to undergo simultane- ously, from the aspect of the teams doing the deprecation as well as the customers of those teams. For example, although everybody appreciates having freshly paved roads, if the public works department decided to close down every road for paving simultaneously, nobody would go anywhere. By focusing their efforts, paving crews can get specific jobs done faster while also allowing other traffic to make progress. Likewise, it's important to choose deprecation projects with care and then commit to following through on finishing them.

尽管“弃用”很有用，但我们在 Google 了解到，从执行“弃用”的团队以及这些团队的客户的角度来看，对同时进行的““弃用””是有数量上的限制的。例如，虽然每个人都喜欢新铺设的道路，但如果政府部门决定同时关闭所有道路并进行铺设，那么将会导致大家无路可走。通过集中精力，铺设人员可以更快地完成特定工作，但同时不应该影响其他道路的通行。故同样重要的是要谨慎选择“弃用”项目并付诸实施。

## 三、 为什么“弃用”这么难 (Why Is Deprecation So Hard?)

We've mentioned Hyrum's Law elsewhere in this book, but it's worth repeating its applicability here: the more users of a system, the higher the probability that users are using it in unexpected and unforeseen ways, and the harder it will be to deprecate and remove such a system. Their usage just "happens to work"instead of being "guaran- teed to work."In this context, removing a system can be thought of as the ultimate change: we aren't just changing behavior, we are removing that behavior completely! This kind of radical alteration will shake loose a number of unexpected dependents.

我们在本书的其他地方提到了海拉姆定律，但值得在这里重申一下它的适用性：一个系统的用户越多，用户以意外和不可预见的方式使用它的可能性就越大，并且越难“弃用”和删除这样的系统。它们有可能只是“碰巧可用”而不是“绝对可用”。在这种情况下，“弃用”不是简单的行为变更，而是一次大变革-彻底的“弃用”！这种激进的改变可能会对这样的系统造成意想不到的影响。

To further complicate matters, deprecation usually isn't an option until a newer sys- tem is available that provides the same (or better!) functionality. The new system might be better, but it is also different: after all, if it were exactly the same as the obso- lete system, it wouldn't provide any benefit to users who migrate to it (though it might benefit the team operating it). This functional difference means a one-to-one match between the old system and the new system is rare, and every use of the old system must be evaluated in the context of the new one.

更复杂的是，在提供相同（或更好）功能的新系统可用之前，“弃用”通常不是一种选择。新系统可能更好，但也有不同：毕竟，如果它和过时的系统完全一样，它不会为迁移到它的用户提供任何好处（尽管它可能使运行它的团队受益）。这种功能差异意味着旧系统和新系统之间的一对一匹配很少见，新老系统的切换通常需要进行评估。

Another surprising reluctance to deprecate is emotional attachment to old systems, particularly those that the deprecator had a hand in helping to create. An example of this change aversion happens when systematically removing old code at Google: we've occasionally encountered resistance of the form "I like this code!"It can be dif- ficult to convince engineers to tear down something they've spent years building. This is an understandable response, but ultimately self-defeating: if a system is obso- lete, it has a net cost on the organization and should be removed. One of the ways we've addressed concerns about keeping old code within Google is by ensuring that the source code repository isn't just searchable at trunk, but also historically. Even code that has been removed can be found again (see Chapter 17).

另一个令人惊讶的不愿“弃用”的现象是对旧系统的情感依恋，尤其是那些“弃用”者帮助创建的系统。在 Google 系统地删除旧代码时，就会发生这种厌恶更改的一个例子：我们偶尔会遇到“我喜欢这段代码！”这种形式的抵制。说服工程师删除他们花了多年时间建造的东西可能很困难。这是一种可以理解的反应，但最终会弄巧成拙：如果一个系统已经过时，它会给组织带来净成本，应该将其删除。我们解决了将旧代码保留在 Google 中的问题的方法之一是确保源代码存储库不仅可以在主干上搜索，而且可以在历史上搜索。甚至被删除的代码也能再次找到 (见 17 章)

> There's an old joke within Google that there are two ways of doing things: the one that's deprecated, and the one that's not-yet-ready. This is usually the result of a new solution being "almost"done and

is the unfortunate reality of working in a technolog- ical environment that is complex and fast-paced.

Google engineers have become used to working in this environment, but it can still be disconcerting. Good documentation, plenty of signposts, and teams of experts help- ing with the deprecation and migration process all make it easier to know whether you should be using the old thing, with all its warts, or the new one, with all its uncertainties.

谷歌内部有一个古老的笑话，说有两种做事方式：一种已被“弃用”，另一种尚未准备就绪。这通常发生成新解决方案“几乎”完成的时候，并且是在复杂且快节奏的技术环境中工作的不幸现实。谷歌工程师已经习惯了在这种环境中工作，但它仍然令人不安。良好的文档、大量的指引以及帮助“弃用”和迁移过程的专家团队，都可以让您更容易地判断是使用旧的，有缺点，还是新的，有不确定性的。

Finally, funding and executing deprecation efforts can be difficult politically; staffing a team and spending time removing obsolete systems costs real money, whereas the costs of doing nothing and letting the system lumber along unattended are not read- ily observable. It can be difficult to convince the relevant stakeholders that depreca- tion efforts are worthwhile, particularly if they negatively impact new feature development. Research techniques, such as those described in Chapter 7, can provide concrete evidence that a deprecation is worthwhile.

最后，资助和执行“弃用”工作在政治上可能很困难；为团队配备人员并花时间移除过时的系统会花费大量金钱，而无所作为和让系统在无人看管的情况下缓慢运行的成本不易观察到。很难让相关利益相关者相信“弃用”工作是值得的，尤其是当它们对新功能开发产生负面影响时。研究技术，例如第 7 章中描述的那些，可以提供具体的证据证明“弃用”是值得的。

Given the difficulty in deprecating and removing obsolete software systems, it is often easier for users to evolve a system in situ, rather than completely replacing it. Incre- mentality doesn’t avoid the deprecation process altogether, but it does break it down into smaller, more manageable chunks that can yield incremental benefits. Within Google, we’ve observed that migrating to entirely new systems is extremely expensive, and the costs are frequently underestimated. Incremental deprecation efforts accomplished by in-place refactoring can keep existing systems running while mak- ing it easier to deliver value to users.

鉴于“弃用”和删除过时软件系统的难度，用户通常更容易就地改进系统，而不是完全替换它。增量并没有完全避免“弃用”过程，但它确实将其分解为更小、更易于管理的块，这些块可以产生增量收益。在 Google 内部，我们观察到迁移到全新系统的成本非常高，而且成本经常被低估。增量“弃用”工作通过就地重构实现的功能可以保持现有系统运行，同时更容易向用户交付价值。

## 3.1. 设计之初便考虑“弃用”(Deprecation During Design)

Like many engineering activities, deprecation of a software system can be planned as those systems are first built. Choices of programming language, software architec- ture, team composition, and even company policy and culture all impact how easy it will be to eventually remove a system after it has reached the end of its useful life.

与许多工程活动一样，软件系统的“弃用”可以在这些系统首次设计时便进行规划。编程语言、软件架构、团队组成，甚至公司政策和文化的选择都会影响系统在使用寿命结束后最终将其“弃用”的难易程度。

The concept of designing systems so that they can eventually be deprecated might be radical in software engineering, but it is common in other engineering disciplines. Consider the example of a nuclear power plant, which is an extremely complex piece of engineering. As part of the design of a nuclear power station, its eventual decom- missioning after a lifetime of productive service must be taken into account, even going so far as to allocate funds for this purpose.1 Many of the design choices in building a nuclear power plant are affected when engineers know that it will eventu- ally need to be decommissioned.

设计系统以使其最终可以被“弃用”的概念在软件工程中可能是激进的，但它在其他工程学科中很常见。以核电站为例，这是一项极其复杂的工程。作为核电站设计的一部分，必须考虑到其在服务寿命到期后最终退役，甚至为此分配资金。当工程师知道它最终需要退役时，核电站建设中的许多设计，将会随之改变。

Unfortunately, software systems are rarely so thoughtfully designed. Many software engineers are attracted to the task of building and launching new systems, not main- taining existing ones. The corporate culture

of many companies, including Google, emphasizes building and shipping new products quickly, which often provides a dis- incentive for designing with deprecation in mind from the beginning. And in spite of the popular notion of software engineers as data-driven automata, it can be psycho- logically difficult to plan for the eventual demise of the creations we are working so hard to build.

不幸的是，软件系统很少经过精心设计。许多软件工程师更热心于构建和启动新系统，而不是维护现有系统。包括 Google 在内的许多公司的企业文化都强调快速构建和交付新产品，这通常会阻碍从一开始就考虑“弃用”的设计。尽管普遍认为软件工程师是数据驱动的自动机，但在心理上很难为我们辛勤工作的创造物的最终消亡做计划。

So, what kinds of considerations should we think about when designing systems that we can more easily deprecate in the future? Here are a couple of the questions we encourage engineering teams at Google to ask: • How easy will it be for my consumers to migrate from my product to a potential replacement? • How can I replace parts of my system incrementally? Many of these questions relate to how a system provides and consumes dependencies. For a more thorough discussion of how we manage these dependencies, see Chapter 16.

那么，在设计我们将来更容易“弃用”的系统时，我们应该考虑哪些因素？以下是我们鼓励 Google 的工程团队提出的几个问题：

- 我的使用者从我的产品迁移到潜在替代品的难易程度如何？
- 如何逐步更换系统部件？

其中许多问题与系统如何提供和使用依赖项有关。有关我们如何管理这些依赖项的更深入讨论，请参阅第 16 章。

Finally, we should point out that the decision as to whether to support a project long term is made when an organization first decides to build the project. After a software system exists, the only remaining options are support it, carefully deprecate it, or let it stop functioning when some external event causes it to break. These are all valid options, and the trade-offs between them will be organization specific. A new startup with a single project will unceremoniously kill it when the company goes bankrupt, but a large company will need to think more closely about the impact across its port- folio and reputation as they consider removing old projects. As mentioned earlier, Google is still learning how best to make these trade-offs with our own internal and external products.

最后，我们应该指出，是否长期支持项目的决定，是在组织最初决定建立项目时做出的。软件系统存在后，剩下的唯一选择是支持它，小心地“弃用”它，或者在某些外部事件导致它崩溃时让它停止运行。这些都是有效的选项，它们之间的权衡将是特定于组织的。当公司破产时，一个只有一个项目的新创业公司会毫不客气地杀死它，但一家大公司在考虑删除旧项目时需要更仔细地考虑对其投资组合和声誉的影响。如前所述，谷歌仍在学习如何最好地利用我们自己的内部和外部产品进行这些权衡。

In short, don’t start projects that your organization isn’t committed to support for the expected lifespan of the organization. Even if the organization chooses to deprecate and remove the project, there will still be costs, but they can be mitigated through planning and investments in tools and policy.

简而言之，如果你的公司不打算长期支持某个项目，那么轻易不要启动这个项目。即使公司选择“弃用”项目，仍然会有成本，但可以通过规划和投资工具和政策来降低成本。

## 四、 “弃用”的种类 (Types of Deprecation)

Deprecation isn’t a single kind of process, but a continuum of them, ranging from “we’ll turn this off someday, we hope”to “this system is going away tomorrow, cus- tomers better be ready for that.”Broadly speaking, we divide this continuum into two separate areas: advisory and compulsory.

“弃用”不是一种单一的过程，而是一个连续的过程，从“我们希望有一天会关闭它”到“这个系统明天就会消失，客户最好为此做好准备。”从广义上讲，我们将这个连续统一体分为两个独立的领域：建议和强制。

## 4.1. 建议性“弃用”(Advisory Deprecation)

Advisory deprecations are those that don’t have a deadline and aren’t high priority for the organization (and for which the company isn’t willing to dedicate resources). These could also be labeled aspirational deprecations: the team knows the system has been replaced, and although they hope clients will eventually migrate to the new sys- tem, they don’t have imminent plans to either provide support to help move clients or delete the old system. This kind of deprecation often lacks enforcement: we hope that clients move, but can’t force them to. As our friends in SRE will readily tell you: “Hope is not a strategy.”

建议性“弃用”是那些没有截止日期并且对组织来说不是高优先级的（并且公司不愿意为此投入资源）。这些也可能被标记为理想“弃用”：团队知道系统已被替换，尽管他们希望客户最终迁移到新系统，但他们没有近期的计划来提供支持以帮助客户迁移或删除旧系统。这种“弃用”往往缺乏执行力：我们希望客户迁移，但不强迫他们做。正如我们在 SRE 的朋友会很容易告诉你的那样：“希望不是策略。”

Advisory deprecations are a good tool for advertising the existence of a new system and encouraging early adopting users to start trying it out. Such a new system should not be considered in a beta period: it should be ready for production uses and loads and should be prepared to support new users indefinitely. Of course, any new system is going to experience growing pains, but after the old system has been deprecated in any way, the new system will become a critical piece of the organization’s infrastructure.

建议性“弃用”是宣传新系统存在并鼓励早期采用的用户开始尝试的好工具。这样的新系统不应该在测试阶段被考虑：它应该准备好用于生产用途和负载，并且应该准备好无限期地支持新用户。当然，任何新系统都会经历成长的痛苦，但是在旧系统以任何方式被“弃用”之后，新系统将成为组织基础设施的关键部分。

One scenario we’ve seen at Google in which advisory deprecations have strong bene- fits is when the new system offers compelling benefits to its users. In these cases, simply notifying users of this new system and providing them self-service tools to migrate to it often encourages adoption. However, the benefits cannot be simply incremental: they must be transformative. Users will be hesitant to migrate on their own for marginal benefits, and even new systems with vast improvements will not gain full adoption using only advisory deprecation efforts.

我们在谷歌看到的一种情况是，当新系统为其用户提供令人信服的好处时，建议性“弃用”具有强大的好处。在这些情况下，简单地通知用户这个新系统并为他们提供自助服务工具以迁移到它, 通常会鼓励采用。然而，收益不能简单地递增：它们必须具有变革性。否则用户将不愿为了这一点点边际收益而自行迁移，不过对于“建议性“弃用””，即使具有巨大改进的新系统也通常不会被完全采纳。

Advisory deprecation allows system authors to nudge users in the desired direction, but they should not be counted on to do the majority of migration work. It is often tempting to simply put a deprecation warning on an old system and walk away without any further effort. Our experience at Google has been that this can lead to (slightly) fewer new uses of an obsolete system, but it rarely leads to teams actively migrating away from it. Existing uses of the old system exert a sort of conceptual (or technical) pull toward it: comparatively many uses of the old system will tend to pick up a large share of new uses, no matter how much we say, “Please use the new sys- tem.”The old system will continue to require maintenance and other resources unless its users are more actively encouraged to migrate.

建议性“弃用”允许系统作者将用户推向所需的方向，但不应指望他们完成大部分迁移工作。通常只需要在旧系统上简单地发出“弃用”警告，然后弃之不顾即可。我们在 Google 的经验是，这可能会导致（略微）减少对过时系统的使用，但很少会导致团队积极迁移。旧系统的现有功能会有一种吸引力，吸引更多的系统使用它, 无论我们说多少，“请使用新的系统。”除非更积极地鼓励其用户迁移，否则旧系统将需要继续维护。

## 4.2. 强制性“弃用”(Compulsory Deprecation)

This active encouragement comes in the form of compulsory deprecation. This kind of deprecation usually comes with a deadline for removal of the obsolete system: if users continue to depend on it beyond that date, they will find their own systems no longer work.

这种“弃用”通常伴随着删除过时系统的最后期限：如果用户在该日期之后继续依赖它，他们将发现自己的系统不再正常工作。

Counterintuitively, the best way for compulsory deprecation efforts to scale is by localizing the expertise of migrating users to within a single team of experts—usually the team responsible for removing the old system entirely. This team has incentives to help others migrate from the obsolete system and can develop experience and tools that can then be used across the organization. Many of these migrations can be effec- ted using the same tools discussed in Chapter 22.

与直觉相反，推广强制性“弃用”工作的最佳方法是将迁移用户的工作交给一个专家团队——通常是负责完全删除旧系统的团队。该团队有动力帮助其他人从过时的系统迁移，并可以开发可在整个组织中使用的经验和工具。许多这些迁移可以使用第 22 章中讨论的相同工具来实现。

For compulsory deprecation to actually work, its schedule needs to have an enforce- ment mechanism. This does not imply that the schedule can't change, but empower the team running the deprecation process to break noncompliant users after they have been sufficiently warned through efforts to migrate them. Without this power, it becomes easy for customer teams to ignore deprecation work in favor of features or other more pressing work.

为了让强制性“弃用”真正起作用，需要有一个强制执行的时间表。并以警告的形式通知到需要执行迁移的客户团队。没有这种能力，客户团队很容易忽略“弃用”工作，而转而支持其他更紧迫的工作。

At the same time, compulsory deprecations without staffing to do the work can come across to customer teams as mean spirited, which usually impedes completing the deprecation. Customers simply see such deprecation work as an unfunded mandate, requiring them to push aside their own priorities to do work just to keep their serv- ices running. This feels much like the “running to stay in place”phenomenon and creates friction between infrastructure maintainers and their customers. It's for this reason that we strongly advocate that compulsory deprecations are actively staffed by a specialized team through completion.

同时，若没有安排人员协助，可能会给客户团队带来刻薄的印象，这通常会影响迁移的进度。客户只是将它视为一项没有资金的任务，要求他们搁置自己的优先事项，只为保持服务运行而迁移。这会在两个团队间产生摩擦, 故此，我们建议安排人员进行协助迁移。

It's also worth noting that even with the force of policy behind them, compulsory deprecations can still face political hurdles. Imagine trying to enforce a compulsory deprecation effort when the last remaining user of the old system is a critical piece of infrastructure your entire organization depends on. How willing would you be to break that infrastructure—and, transitively, everybody that depends on it—just for the sake of making an arbitrary deadline? It is hard to believe the deprecation is really compulsory if that team can veto its progress.

还值得注意的是，即使有政策支持，强制性“弃用”仍可能面临政治障碍。想象一下，当旧系统的最后一个剩余用户是整个组织所依赖的关键基础架构时，你会愿意为了在截止日期前完成迁移而破坏那个基础设施及所有依赖它的系统吗？如果该团队可以否决其进展，那它的强制性就值得怀疑。

Google's monolithic repository and dependency graph gives us tremendous insight into how systems are used across our ecosystem. Even so, some teams might not even know they have a dependency on an obsolete system, and it can be difficult to dis- cover these dependencies analytically. It's also possible to find them dynamically through tests of increasing frequency and duration during which the old system is turned off temporarily. These intentional changes provide a mechanism for discover- ing unintended dependencies by seeing what breaks, thus alerting teams to a need to prepare for the upcoming deadline. Within Google, we occasionally change the name of implementation-only symbols to see which users are depending on them unaware.

Google 的中心代码仓库和依赖关系图让我们深入了解系统如何在我们的生态系统中使用。即便如此，一些团队甚至可能不知道他们依赖于一个过时的系统，并且很难通过分析发现这些依赖关系。也可以通过增加频率和持续时间的测试来动态找到它们，在此期间旧系统暂时关闭。这些有意的更改提供了一种机制，通过查看中断的内容来发现意外的依赖关系，从而提醒团队需要为即将到来的截止日期做好准备。在 Google 内部，我们偶尔会更改仅变量的名称，来查看哪些用户不知道依赖了它们。

Frequently at Google, when a system is slated for deprecation and removal, the team will announce planned outages of increasing duration in the months and weeks prior to the turndown. Similar to Google's Disaster Recovery Testing (DiRT) exercises, these events often discover unknown dependencies between running systems. This incremental approach allows those dependent teams to discover and then plan for the system's eventual removal, or even work with the deprecating team to adjust their timeline. (The same principles also apply for

static code dependencies, but the semantic information provided by static analysis tools is often sufficient to detect all the dependencies of the obsolete system.)

在谷歌，当系统计划“弃用”时，团队经常会在关闭前的几个月和几周内宣布计划中断，持续时间会增加。与 Google 的灾难恢复测试 (DiRT) 类似，这些事件通常会发现正在运行的系统之间的未知依赖关系。这种渐进式方法允许那些依赖的团队发现依赖，然后为系统的最终移除做计划，甚至与“弃用”团队合作调整他们的时间表。(同样的原则也适用于静态代码依赖，但静态分析工具提供的语义信息通常足以检测过时系统的所有依赖。)

## 4.3. （弃用警告）Deprecation Warnings

For both advisory and compulsory deprecations, it is often useful to have a program- matic way of marking systems as deprecated so that users are warned about their use and encouraged to move away. It’s often tempting to just mark something as depre- cated and hope its uses eventually disappear, but remember: “hope is not a strategy.”Deprecation warnings can help prevent new uses, but rarely lead to migration of existing systems.

对于建议性和强制“弃用”，以程序化的方式将系统标记为“弃用”通常很有用，这样用户就会及时的发现警告并远离它。将某些东西标记为已“弃用”并希望它的使用最终消失通常很诱人，但请记住：“希望不是一种策略。”“弃用”警告可以减少它的新增用户，但很少导致现有系统的迁移。

What usually happens in practice is that these warnings accumulate over time. If they are used in a transitive context (for example, library A depends on library B, which depends on library C, and C issues a warning, which shows up when A is built), these warnings can soon overwhelm users of a system to the point where they ignore them altogether. In health care, this phenomenon is known as “alert fatigue.”

在实践中通常会发生这些警告随着时间的推移而累积。如果它们在传递上下文中使用（例如，库 A 依赖于库 B，而库 B 又依赖于库 C，而 C 发出警告，并在构建 A 时显示），则这些警告很快就会使系统用户不知所措他们完全忽略它们的点。在医疗保健领域，这种现象被称为“警觉疲劳”。

Any deprecation warning issued to a user needs to have two properties: actionability and relevance. A warning is actionable if the user can use the warning to actually per- form some relevant action, not just in theory, but in practical terms, given the exper- tise in that problem area that we expect for an average engineer. For example, a tool might warn that a call to a given function should be replaced with a call to its updated counterpart, or an email might outline the steps required to move data from an old system to a new one. In each case, the warning provided the next steps that an engi- neer can perform to no longer depend on the deprecated system.2

向用户发出的任何“弃用”警告都需要具有两个属性：可操作性和相关性。如果用户可以使用警告来实际执行某些相关操作，则警告是可操作的，不仅在理论上，而且在实践中，即要提供可操作的迁移步骤，而不仅仅是一个警告。

A warning can be actionable, but still be annoying. To be useful, a deprecation warn- ing should also be relevant. A warning is relevant if it surfaces at a time when a user actually performs the indicated action. Warning about the use of a deprecated func- tion is best done while the engineer is writing code that uses that function, not after it has been checked into the repository for several weeks. Likewise, an email for data migration is best sent several months before the old system is removed rather than as an afterthought a weekend before the removal occurs.

警告可能是可行的，但仍然很烦人。为了有用，“弃用”警告也应该是相关的。如果警告在用户实际执行指示的操作时出现，则该警告是相关的。关于使用已“弃用”函数的警告最好在工程师编写使用该函数的代码时完成，而不是在将其签入存储库数周后。同样，最好在删除旧系统前几个月发送数据迁移电子邮件，而不是在删除前的一个周末之后才发送。

It’s important to resist the urge to put deprecation warnings on everything possible. Warnings themselves are not bad, but naive tooling often produces a quantity of warning messages that can overwhelm the unsus- pecting engineer. Within Google, we are very liberal with marking old functions as deprecated but leverage tooling such as ErrorProne or clang-tidy to ensure that warnings are surfaced in targeted ways. As discussed in Chapter 20, we limit these warnings to newly changed lines as a way to warn people about new uses of the deprecated symbol. Much more intrusive warn- ings, such as for deprecated targets in the dependency graph, are added only for com- pulsory deprecations, and the team is actively moving users away. In either case,

tooling plays an important role in surfacing the appropriate information to the appropriate people at the proper time, allowing more warnings to be added without fatiguing the user.

警告不是越多越好。警告本身并不坏，但不成熟的工具通常会产生大量警告消息，这些消息可能会让工程师不知所措。在 Google 内部，我们会将旧功能标记为已“弃用”，但会利用 ErrorProne 或 clang-tidy 等工具来确保以有针对性的方式显示警告。正如第 20 章中所讨论的，我们将这些警告限制在新更改的行中，以警告人们有关已“弃用”符号的新用法。更具侵入性的警告，例如依赖图中已“弃用”的警告，仅针对强制“弃用”添加，并且团队正在积极地将用户移走。在任何一种情况下，工具都在适当的时间向适当的人提供适当的信息方面发挥着重要作用，允许添加更多警告而不会使用户感到疲倦。

# 五、 管理“弃用”的流程 (Managing the Deprecation Process)

Although they can feel like different kinds of projects because we’re deconstructing a system rather than building it, deprecation projects are similar to other software engi- neering projects in the way they are managed and run. We won’t spend too much effort going over similarities between those management efforts, but it’s worth point- ing out the ways in which they differ.

“弃用”项目尽管与上线一个项目给你的感官不同，但它们的管理和运行方式却是类似的。我们不会花太多精力去讨论他们有何共同点，但有必要指出他们有何不同。

## 5.1. 确定“弃用”的负责人 (Process Owners)

We’ve learned at Google that without explicit owners, a deprecation process is unlikely to make meaningful progress, no matter how many warnings and alerts a system might generate. Having explicit project owners who are tasked with managing and running the deprecation process might seem like a poor use of resources, but the alternatives are even worse: don’t ever deprecate anything, or delegate deprecation efforts to the users of the system. The second case becomes simply an advisory depre- cation, which will never organically finish, and the first is a commitment to maintain every old system ad infinitum. Centralizing deprecation efforts helps better assure that expertise actually reduces costs by making them more transparent.

我们在 Google 了解到，如果没有明确的 Owner，无论系统产生了多少警报,“弃用”过程恐怕都不会太乐观。为了弃用专门指定一个负责人似乎是对资源的浪费，永不“弃用”，或将“弃用”工作完全交给系统的使用者，恐怕会是一个更糟的方案。交给使用者来执行的方案，最多只能应对建议性“弃用”，恐怕它很难做到彻底地“弃用”，而永不“弃用”则相当于无限期地维护着旧系统。集中性的执行“弃用”则更专业更透明，从而真正达到降低成本的目的。

Abandoned projects often present a problem when establishing ownership and align- ing incentives. Every organization of reasonable size has projects that are still actively used but that nobody clearly owns or maintains, and Google is no exception. Projects sometimes enter this state because they are deprecated: the original owners have moved on to a successor project, leaving the obsolete one chugging along in the base- ment, still a dependency of a critical project, and hoping it just fades away eventually.

废弃的项目通常会在确定归属权上存在扯皮的情形。每个小组都存在大量仍在使用却无明确维护人的项目，谷歌也不例外。当一个项目存在这种情形时，通常说明它已被抛弃：即原维护人已参与到新项目开发维护中，老项目则被弃之不顾，但却仍然被某些关键项目所依赖，只寄希望于它慢慢消失在众人视线中。

Such projects are unlikely to fade away on their own. In spite of our best hopes, we’ve found that these projects still require deprecation experts to remove them and pre- vent their failure at inopportune times. These teams should have removal as their pri- mary goal, not just a side project of some other work. In the case of competing priorities, deprecation work will almost always be perceived as having a lower priority and rarely receive the attention it needs. These sorts of important-not-urgent cleanup tasks are a great use of 20% time and provide engineers exposure to other parts of the codebase.

但此类项目不太可能自行消失。尽管我们对之抱有最大的期寄，但我们发现,“弃用”这些项目仍然需要专人负责，否则恐怕会造成意外的损失。负责人应该将废弃他们作为主要目标。在排优先级时，“弃用”通常会有较低的优先级，且少有人关注。但实际上，这些重要但不紧急的清理工作，占用掉程序员 20% 的工作时间，应该是个合适的数字。

## 5.2. 制定里程碑 (Milestones)

When building a new system, project milestones are generally pretty clear: “Launch the frobnazzer features by next quarter.”Following incremental development practi- ces, teams build and deliver functionality incrementally to users, who get a win when- ever they take advantage of a new feature. The end goal might be to launch the entire system, but incremental milestones help give the team a sense of progress and ensure they don’t need to wait until the end of the process to generate value for the organization.

在构建新系统时，项目里程碑通常非常明确：如“在下个季度推出某项功能。”。遵循迭代式开发流程的团队，通常以积小成大的方式构建系统, 并最终交付给用户，只要他们使用了新功能，他们的目的便算得到了。最终目标当然是启用整个系统，但增量迭代式的开发，则能让团队成员更有成就感，因他们无需等到项目结束就可体验项目。

In contrast, it can often feel that the only milestone of a deprecation process is removing the obsolete system entirely. The team can feel they haven’t made any pro- gress until they’ve turned out the lights and gone home. Although this might be the most meaningful step for the team, if it has done its job correctly, it’s often the least noticed by anyone external to the team, because by that point, the obsolete system no longer has any users. Deprecation project managers should resist the temptation to make this the only measurable milestone, particularly given that it might not even happen in all deprecation projects.

相反，对于“弃用”，它常会给人一种只有一个里程碑的错觉，即完全干掉老旧的项目。下班时，团队成员通常会有没取得任何进展的感觉。干掉一个老旧的项目对团队成员来说虽是颇有意义，但对团队之外的人来说却是完全无感，因老旧的系统已不再被任何服务调用。故项目经理不应将完全根除旧项目当作唯一的里程碑。

Similar to building a new system, managing a team working on deprecation should involve concrete incremental milestones, which are measurable and deliver value to users. The metrics used to evaluate the progress of the deprecation will be different, but it is still good for morale to celebrate incremental achievements in the depreca- tion process. We have found it useful to recognize appropriate incremental mile- stones, such as deleting a key subcomponent, just as we’d recognize accomplishments in building a new product.

与新建项目一样，“弃用”一个项目也该渐进的设置多个可量化的里程碑, 用于评估“弃用”进度的指标会有差异，但阶段性的庆祝有助提升士气。

## 5.3. 工具加持 (Deprecation Tooling)

Much of the tooling used to manage the deprecation process is discussed in depth elsewhere in this book, such as the large-scale change (LSC) process (Chapter 22) or our code review tools (Chapter 19). Rather than talk about the specifics of the tools, we’ll briefly outline how those tools are useful when managing the deprecation of an obsolete system. These tools can be categorized as discovery, migration, and backslid- ing prevention tooling.

许多用于管理“弃用”过程的工具在本书的其他地方进行了深入讨论，例如大规模变更 (LSC) 过程（第 22 章）或我们的代码审查工具（第 19 章）。我们不讨论这些工具的细节，而是简要概述如何让这些工具在管理废弃系统的“弃用”时发辉作用。这些工具可以归类为发现、迁移和倒回滚预防工具。

### 5.3.1 发现使用者 (Discovery)

During the early stages of a deprecation process, and in fact during the entire process, it is useful to know how and by whom an obsolete system is being used. Much of the initial work of deprecation is determining who is using the old system—and in which unanticipated ways. Depending on the kinds of use, this process may require revisit- ing the deprecation decision once new information is learned. We also use these tools throughout the deprecation process to understand how the effort is progressing.

在早期阶段，实际上在整个过程中，确认谁在使用及怎样使用我们的废弃项目很有必要。初始工作通常是用于确认谁在用、以及以怎样的方式使用。根据使用的方式不同，有可能会推翻我们“弃用”的推进流程。我们还在整个弃用过程中使用这些工具来了解工作进展情况。

Within Google, we use tools like Code Search (see Chapter 17) and Kythe (see Chap- ter 23) to statically determine which customers use a given library, and often to sam- ple existing usage to see what sorts of behaviors

customers are unexpectedly depending on. Because runtime dependencies generally require some static library or thin client use, this technique yields much of the information needed to start and run a deprecation process. Logging and runtime sampling in production help discover issues with dynamic dependencies.

在谷歌内部，我们使用代码搜索（见第 17 章）和 Kythe（见第 23 章）等工具来静态地确定哪些客户使用给定的库，并经常对现有使用情况进行抽样，以了解客户的使用方式。由于运行时依赖项通常需要使用一些静态库或瘦客户端，因此该技术能提供大部分决策信息。而生产中的日志记录和运行时采样有助于发现动态依赖项的问题。

Finally, we treat our global test suite as an oracle to determine whether all references to an old symbol have been removed. As discussed in Chapter 11, tests are a mecha- nism of preventing unwanted behavioral changes to a system as the ecosystem evolves. Deprecation is a large part of that evolution, and customers are responsible for having sufficient testing to ensure that the removal of an obsolete system will not harm them.

最后，我们将集成测试套件视为预言机，以确定是否已删除对旧变量、函数的所有引用。正如第 11 章所讨论的，测试是一种防止系统随着生态系统发展而发生不必要的行为变化的机制。“弃用”是这种演变的重要组成部分，客户有责任进行足够的测试，以确保删除过时的系统不会对他们造成危害。

### 5.3.2 迁移 (Migration)

Much of the work of doing deprecation efforts at Google is achieved by using the same set of code generation and review tooling we mentioned earlier. The LSC pro- cess and tooling are particularly useful in managing the large effort of actually updat- ing the codebase to refer to new libraries or runtime services.

在 Google “弃用”的大部分工作是通过使用我们之前提到的同一组代码生成和审查工具来完成的, 即 LSC 工具集。它在代码仓库在引入新库或运行时服务时会很有用。

### 5.3.3 避免“弃用”项目被重新启用 (Preventing backsliding)

Finally, an often overlooked piece of deprecation infrastructure is tooling for prevent- ing the addition of new uses of the very thing being actively removed. Even for advi- sory deprecations, it is useful to warn users to shy away from a deprecated system in favor of a new one when they are writing new code. Without backsliding prevention, deprecation can become a game of whack-a-mole in which users constantly add new uses of a system with which they are familiar (or find examples of elsewhere in the codebase), and the deprecation team constantly migrates these new uses. This process is both counterproductive and demoralizing.

最后，一个经常被忽视的问题是新增功能重新使用了废弃的项目。即使对于建议性“弃用”, 警告用户在编写新代码时避免使用已“弃用”的系统而支持新系统也是很有用的。如果没有后退预防机制，“弃用”可能会变成一场打地鼠游戏。按下葫芦浮起瓢是很影响士气的。

To prevent deprecation backsliding on a micro level, we use the Tricorder static anal- ysis framework to notify users that they are adding calls into a deprecated system and give them feedback on the appropriate replacement. Owners of deprecated systems can add compiler annotations to deprecated symbols (such as the @deprecated Java annotation), and Tricorder surfaces new uses of these symbols at review time. These annotations give control over messaging to the teams that own the deprecated sys- tem, while at the same time automatically alerting the change author. In limited cases, the tooling also suggests a push-button fix to migrate to the suggested replacement.

为了防止使用废弃项目，我们使用 Tricorder 静态分析框架来通知用户他们正在调用一个“弃用”的系统中，并提供替代方案。废弃系统的维护者应该将不推荐使用的符号添加编译器注释（例如 @deprecated Java 注释），并且 Tricorder 在审查时会将其发送给废弃项目的维护者。同时自动提醒调用者。在某些情况下，该工具还能一键以替代方案进行修复。

On a macro level, we use visibility whitelists in our build system to ensure that new dependencies are not introduced to the deprecated system. Automated tooling peri- odically examines these whitelists and prunes them as dependent systems are migra- ted away from the obsolete system.

在宏观层面上，我们在构建系统中使用可见性白名单来确保不会将新的依赖项引入已“弃用”的系统。自动化工具会定期检查这些白名单，并在依赖系统从过时系统迁移时对其进行删剪。

# 六、 结论 (Conclusion)

Deprecation can feel like the dirty work of cleaning up the street after the circus parade has just passed through town, yet these efforts improve the overall software ecosystem by reducing maintenance overhead and cognitive burden of engineers. Scalably maintaining complex software systems over time is more than just building and running software: we must also be able to remove systems that are obsolete or otherwise unused.

A complete deprecation process involves successfully managing social and technical challenges through policy and tooling. Deprecating in an organized and well- managed fashion is often overlooked as a source of benefit to an organization, but is essential for its long-term sustainability.

“弃用”感觉就像马戏团刚刚穿过城镇后，清理街道的肮脏工作，但它能通过减少维护开销和工程师的认知负担来改善整个软件生态系统。随着时间的推移，复杂系统的维护, 不仅仅是包含构建和运行那么简单, 还应包含清理过时的老旧系统。

完整的“弃用”过程涉及到管理和技术两个层面的挑战。有效地管理“弃用”通常因不会带来盈利而被轻忽，但它对其长期可持续性维护却至关重要。

# 七、 TL;DRs

- Software systems have continuing maintenance costs that should be weighed against the costs of removing them.
- Removing things is often more difficult than building them to begin with because existing users are often using the system beyond its original design.
- Evolving a system in place is usually cheaper than replacing it with a new one, when turndown costs are included.
- It is difficult to honestly evaluate the costs involved in deciding whether to depre- cate: aside from the direct maintenance costs involved in keeping the old system around, there are ecosystem costs involved in having multiple similar systems to choose between and that might need to interoperate. The old system might implicitly be a drag on feature development for the new. These ecosystem costs are diffuse and difficult to measure. Deprecation and removal costs are often similarly diffuse.
- 软件系统具有持续的维护成本，应与删除它们的成本进行权衡。
- 删除东西通常比开始构建它们更困难，因为现有用户经常使用超出其原始设计意图的系统。
- 如果将停机成本包括在内，就地改进系统通常比更换新系统便宜。
- 很难如实地评估“弃用”所涉及的成本：除了保留旧系统所涉及的直接维护成本外，还有多个相似系统可供选择所涉及的生态成本，互有干涉。旧系统可能会暗中拖累新系统的功能开发。这些不同的生态所带来的成本则分散且难以衡量。“弃用”成本通常同样分散。