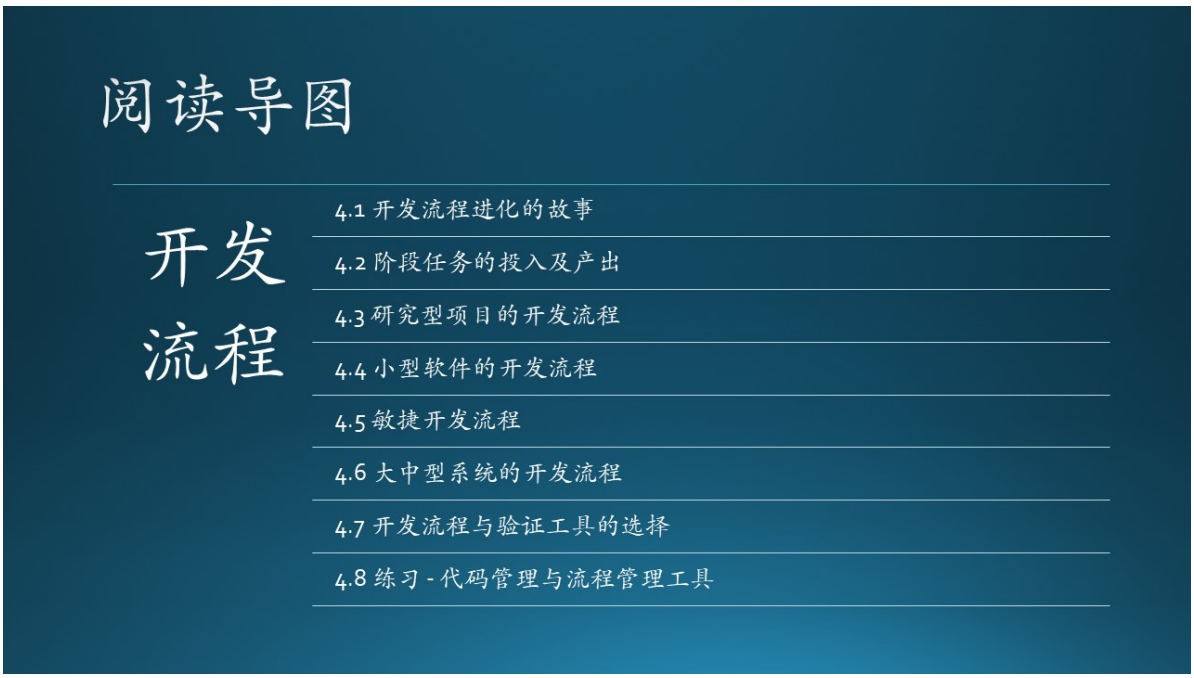
注意我们使用了“流程（Procedure）”，而非“过程（Process）”，强调的对是软件工程的过程中的各个环节的事无巨细的掌控和对关键点的监督。而过程是一种点到为止的叙述，最后注重的是结果。

- 通过对书本知识（比如本书）的学习，读者可以了解到软件工程的**过程**；
- 而**流程**必须通过动手实践才能得到切身体会。

比如那个家喻户晓的笑话：“如何把大象放到冰箱里？”答曰：“1. 把冰箱门打开；2. 把大象牵进去；3. 把冰箱门关上。”这其实就是一种对**过程**的描述，它可以通过忽略具体细节而混淆视听。

还有一个例子，就是木头如何鼓励新手上台表演：“你就背着吉他，眼睛不用看台下以免紧张，然后弹和弦 1645,1645,1645……，鞠躬下台，多简单的**过程**！”但实际上，吉他手上台表演，要考虑站位、站姿是否符合舞台要求，吉他连接到调音台后音量、音色的调节，和弦进行 1645 要克服大横按的一些操作难点，还要听鼓的节奏避免错拍，还要根据歌手的情绪来调节扫弦的力度，等等。这一系列**流程**能顺利走下来都是必须经过长期练习才能做到。



参考资料

- The Chicken and the Pig, Wiki Pedia, https://en.wikipedia.org/wiki/The_Chicken_and_the_Pig
- 《构建之法》，邹欣，人民邮电出版社
- RASCI Responsibility Matrix, https://managementmania.com/en/rasci-responsibility-matrix
- 《卓有成效的敏捷》，史蒂夫·麦克康乃尔，人民邮电出版社
- 什么是MVP? https://tsh.io/blog/mvp-app-and-the-other-validation-methods/
- 《实用软件工程》第二版，郑人杰，清华大学出版社

- Scrum 中文网，https://www.scrumcn.com/agile/scrum-knowledge-library/scrum.html
- Azure DevOps 文档，https://learn.microsoft.com/zh-cn/azure/devops/boards/?view=azure-devops
- 敏捷网站，https://agilemanifesto.org/

4.1 软件工程阶段划分的故事

现代软件工程的阶段划分方法已经非常成熟了，但是读者可能会产生的问题是：这些阶段真的都有必要存在吗？会不会很浪费时间？

在一些小的软件公司中，由于资源的限制，可能会免除其中一个或几个环节；但是在大的软件公司中，还需要在经典的划分方法上更加细化，才能适应复杂软件或复杂团队的需要。

从下面的故事中，读者可以感同身受，从小到大逐步领会到经典划分方法的必要性。

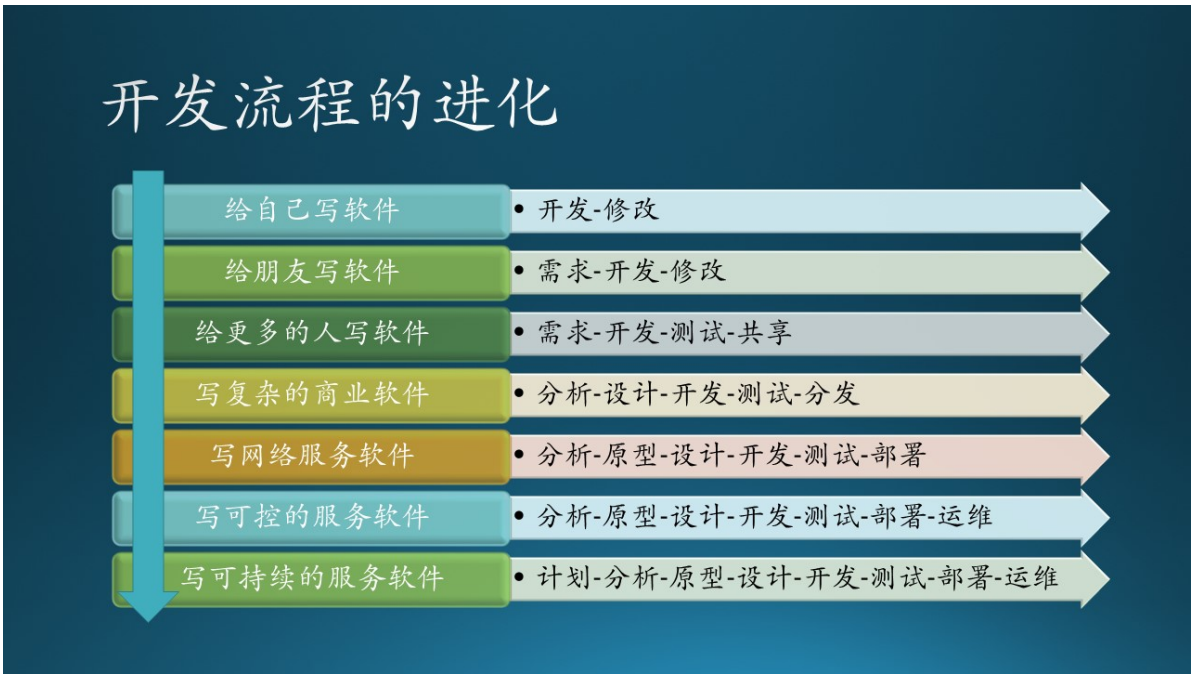


图 4.1.1 软件工程中的开发流程的进化

4.1.1 开发-修改

给自己写软件。Build for Self。

最开始，写软件的人都是为自己写的，为了完成某项繁琐的重复的任务，所以只有"开发-修改"两个步骤。

比如，木头想做一个计算器工具给自己用，因为这个应用场景已经烂熟于心了，所以很快就用 XAML + C# 写了一个粗糙的版本，只能做“加减乘除”。把编译好的可执行程序 Pin 到 Windows 11 的桌面工具条上，双击即可运行，如果发现 bug 就立刻改一改，自己能用就行。

在软件工程还没有出现之前，大家都是处于“开发-修改”的状态，当时的软件也不复杂，代码量不大，所以还是可行的。很多年前，做一个“个人开发者”很时髦，因为很多传统领域都需要软件，那时随便写个软件都能挣钱，木头也确实干过“私活儿”，用挣到的“外快”买了自己的电脑（然后开始打游戏）。

4.1.2 需求-开发-修改

给朋友写软件。Build for Friend。

坐在木头旁边工作的小 D 是个女生，温柔体贴，人也聪明。她觉得那个计算器工具很好，让木头给她拷贝一份。木头心想：“既然是给朋友用，就不要太简陋了吧，别丢脸。” 于是问小 D 都需要什么功能，小 D 想了想，说：“我经常需要用到平方开方计算，你把这两个功能加上吧。”

这就是**需求**的原始形态，面对面简单说两句就能搞清楚了。如果有了 bug 的话，小 D 也是直接和木头说一下，当时花两分钟就改好了。

如果给公司内部做软件或者给团队内部做一个工具，基本上就是这种形式。不太在意什么性能、易用性等等，所以在即使是在微软，内部使用的一些工具类软件也都特别难用。

4.1.3 需求-开发-测试-共享

免费给更多的人写软件。Build for Relationship。

中午大家一起吃饭的时候闲聊，小 D 说起了木头的计算器工具，大家都很感兴趣，都想要一个能满足自己特殊需要的版本。木头也很高兴能帮到大家，而且小 D 也想帮忙一起开发，木头乐得都发芽儿。

两个人简单分了工：

- 小 D 一个一个地询问了同事们的需求，综合到一起；
- 木头写好软件；
- 小 D 按照需求仔细**测试**了每个功能，不然每个人都来报 bug 就太丢脸了；
- 然后木头把这个测试好的统一版本放到一个**共享**文件夹中，大家免费下载使用。

到了这一阶段，出现了多人合作开发的情况，并且出现了测试环节，是一种团队流程的萌芽。由于是免费的，所以没法要求开发者可以提供更多的维护和技术支持，现在很多 GitHub 上的开源项目就是如此，使用的时候需要特别小心，遇到 bug 得自己去解决，不要指望开源者帮你。

4.1.4 分析-设计-开发-测试-分发

写复杂的商业软件。Build for Business。

木头的计算器工具受到了很多人的欢迎。有一个同事的老公的同学的弟弟 B 所在的公司 A 觉得这款软件如果功能再丰富、界面再美观一些，变成一个基于数据库的复合型统计、计算软件，可以做成商业化软件卖钱。所以公司 A 通过弟弟 B 的哥哥的同学的老婆联系到了木头讨论这件事。

木头觉得自己一个人完成不了这件事，就和老板说明了情况，老板也觉得机会很好，就成立了一个项目组，和木头一起完成这个项目。

首先是需求部分，和以前的简单需求不同，需要进行需求**分析**了，要出需求分析文档。

其次，项目结构复杂，涉及到数据库等技术，需要做系统**设计**；而且界面也复杂，需要一个 designer 来做界面**设计**。设计好后，写出系统设计文档，大家才能并行工作。这和木头一个人单枪匹马的情况完全不同，那时设计都在脑子里。

然后，测试也有单独的人来完成，尤其是界面+功能测试，比较繁琐，在发布出去之前需要系统化地测试。

最后一步的**分发**，严格来说不属于软件工程的范畴，但是为了配合后来发展的商业化软件的加密或 Licence 需求，也变得越来越复杂，需要在开发阶段给与配合。形式上，也经历了磁带、磁盘、光盘、远程下载、互联网共享等媒介，变成了一件很复杂的工程。尤其是加密，最初都是用加密盒，接在计算机的串口上，软件定时读取串口中的密钥，检测合法性。

到了这一步，已经正式进入团队开发流程了，形成了传统（经典）的阶段划分方法。很多初创团队或小的软件公司都是处于这种阶段，他们有自己的技术实力和积累，但是需要紧密地结合市场需求才能赚钱。

4.1.5 分析-原型-设计-开发-测试-部署

写商业化的网络服务软件。Build for Service。

上面那个商业软件发布了，公司挣了些小钱儿，老板觉得还可以再挖掘一下商业潜力，就找了几个业务骨干一起商量，觉得如果把这款软件变成一个服务，放在互联网上供大家使用，并收取一定的服务费，这样可以扩大用户范围，更新速度快，还省去了分发的麻烦。

这也是第一次做网络服务软件，木头的团队为了确保能够正确理解需求，先做了一版**原型**，里面除了界面以外，后台的功能都是模拟的，并不进行真正的计算。老板和业务骨干看完后，提了些改进意见和更新的想法，木头又出了一版原型，这才开始后续的开发。

做好软件后，先在本地的 TEST bed（测试环境）进行测试；然后在电信的机房申请了几个机架，自己配好硬件，把软件**部署**到 INT bed（集成环境）中，进行真实环境测试；最后再部署到 PROD bed（生产环境）上，开放给公众使用。

这一步已经用"私有云"的方式来代替传统软件了，好处就是上面所说的"更新快、免分发、用户广、易收费"等。软件公司到了一定规模，就可以用这种方式赚钱了，当然还可以把故事讲得大一些，以便得到风险投资。

4.1.6 分析-原型-设计-开发-测试-部署-运维

写可控的商业化服务软件。Build for Life。

上面的服务发布后，赚了大钱，老板给木头升职加薪。用户量大了，赚钱的压力大了，工作也会更辛苦。木头在公司附近租了一个条件好一些的大房子，免去一些通勤上的辛苦。

运行了一段时间后，也发现了一些小问题，但是由于是网络服务，不能停机，debug 的困难很大。另外，每逢周五，就会发现有很多用户蜂拥而至登录使用，服务器有些承受不住了，而到了周末基本没有用户使用。后来木头才知道那是因为大家都要在周五写总结报告，需要统计数据。

木头和大家商量，在每个关键的业务环节都增加了日志，在硬件资源的使用上也增加了监控，这样就形成了完整的运行维护体系，故障排查时定位很精确。定期检查日志，或通过实时监控**运维**数据，就可以针对资源不足给与报警，再进行人工处理。

针对资源缩放问题，老板觉得自己买很多硬件设备放到电信机房，一个是购买费用高，一个是租金也高，而且一到周末，服务器基本没人用，太浪费了。木头也觉得总跑电信机房做软硬件维护很麻烦，所以建议使用微软的云服务，租用服务器，可远程操作，可自由缩放，不用的话就不花钱。

有了这些基本的运维手段后，系统变得越发的可控，工程师们的日子都好过了些。在工作之余，也可以一起打打台球儿、玩玩儿乐队了，做到 Work Life Balance。一个拥有上百人的中等规模的软件公司可以达到这个程度。

4.1.7 计划-分析-原型-设计-开发-测试-部署-运维

写可持续的商业化服务软件。Build for Future.

在运营过程中，用户经常会发邮件提出一些问题和建议，每天收集整理这些邮件很麻烦。木头就在软件中开通了一个舆情频道，用户可以在里面吐槽。木头的团队还需要对这些舆情进行梳理，分成“问题”和“诉求”两类，是问题的赶快解决，是诉求的记录在团队内部的 DevOps 系统的 backlog 里。另外，团队自己也会经常有一些想法，改善现有的用户体验，或是增加新的功能，同样写在 backlog 里。

与其被用户催着走，不如自己引领方向。积累一段时间后，backlog 里就有了很多内容，木头和同事们把它们都看一遍，挑出其中重要的条目变成新的需求，做出下一轮的开发**计划**。

在这一阶段中，已经进入了自己控制产品方向和开发节奏的良性**循环**，是软件工程的最高级阶段，像微软的 Windows、Office、Azure 等产品都是如此。但是针对不同的公司，或不同规模/性质的软件产品，这个阶段划分还可以进一步细化，我们在“发布与维护”一章中再讨论。

读者可以初步体会到，软件工程不是一个单向的行为，而是像车轮一样向前循环滚动进行的。关于循环流程我们在 4.3 节中再讨论。

4.2 阶段任务的投入与产出

从 4.1 节的故事中，我们最终得到了最理想的阶段划分方法，这种方法可以满足绝大多数软件工程的需要。但是，有可能给读者留下一个误区：“计划结束后做需求分析，需求分析结束后做原型开发，原型开发结束后做系统设计……”但其实没有一个交割点来绝对地分开前后两个阶段，甚至在某个时间段内，有多于 2 个阶段任务的重叠情况。见图 4.2.1。

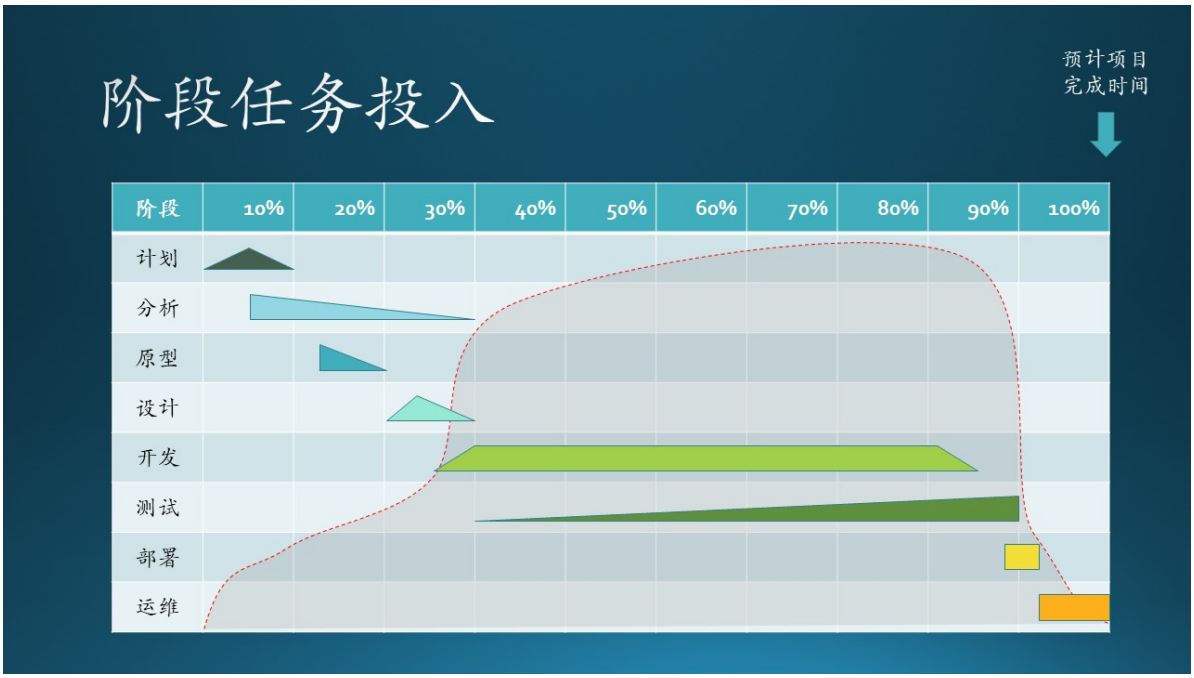


图 4.2.1 阶段任务投入密度曲线

投入密度：关于投入的劳动资源随着时间而形成的密度曲线，越高表示此时刻共同工作的人越多或者劳动强度越大。可以用概率密度曲线做类比理解。

- 斜腰三角形
  逐步热身，积累想法，到尖顶儿时形成初步文档，后期逐步补充完善。
- 左直角三角形
  开始就全部投入，后期工作量逐步减少。
- 右直角三角形
  开始时处于跟随阶段，慢慢熟悉，后期工作量逐步增加。

- 梯形
  短期内熟悉，然后立刻全员投入并持续。
- 矩形
  熟悉时间极短，立刻全员投入并持续。

红色虚线所覆盖的阴影部分，是项目组所有人员的投入密度。

4.2.1 计划

软件项目计划（Software Project Planning）。
人员：商务人员、项目经理、技术负责人。
产出：项目计划书。
时间：不能超过整体的 10%，在 5% 时应该形成初步的文案。

一个简单的 idea 就可以让人躺着赚钱的日子已经不复存在了，现代社会要想成功必须依赖团队力量，而想要说服一个团队一起为一个目标而努力，就要拿出一个看上去漏洞不多的方案来，这个方案就是软件项目计划。

软件项目计划应该包括的主要内容有：

- 目标
  包括软件提供的服务内容、目标用户、预期进度、最终目的。
  比如：
  - 我们要提供一个叫做“必应词典”的英汉词典手机软件；
  - 目标用户是帮助在校学生背单词，帮助想提高英语能力的社会用户练习口语和听力；
  - 预期用半年时间完成大部分功能；
  - 最终在人群中建立“必应词典”的品牌。
- 可行性分析
  - 经济可行性
    需要做成本-效益分析，投入多少人力物力，盈利方案及预期收益。
    比如：在 Android 和 iOS 上各投入三名开发人员，两名后台服务开发人员，一名界面设计人员，两名测试人员，一名项目经理。在软件发布初期不盈利，积累了一定的用户量后，可以在关键模块上按使用量收费，或者增加必应广告。预期收益为每个月 xxx 人民币。
  - 法律可行性
    比如在中国，不能做涉赌涉黄的软件，不能侵权等等。
  - 技术可行性
    所需要的技术与自己所具备的技术之间的差距是否可以弥补，包括软件技术和硬件设备。
    比如：口语练习模块，需要在服务器端部署语音识别服务，把用户上传的口语与标准发音相比对，打分并给出反馈。所以，语音识别技术就是关键技术，目前已经有 xx 公司和 yy 公司提供了服务，我们可以集成。

在图 4.2.1 中的第一行，此项任务的投入密度是一个三角形，开始时大家零散地贡献想法，积累到一定程度后开始形成文案，后期逐渐补充完善。

4.2.2 分析

即软件需求分析（Requirement Analysis）。

人员：项目经理（PM），需求分析人员。
产出：需求调研记录，需求分析文档。
时间：从项目计划书初稿形成开始，投入密度逐步降低，持续时间较长，但是不能超过总时间的 30%。

一个中大型软件系统需要完整的需求调研、需求分析过程，并形成文档并经过共同评审，避免做出来的东西和想要的东西不一致。在初期尽快、尽多地明确需求，在后期可以逐步补充缺失或者不明确的地方，但是不能对已经开始的系统设计产生大的影响。详见第 3 部分。

4.2.3 原型

即原型系统设计及确认（Prototyping）。

人员：视觉设计师（designer）、系统设计师。
产出：原型设计，技术建议书。
时间：在需求分析进行到 1/3 时就可以开始了，越快越好，不要超过总时间的 20%。

力争在较短时间内发布原型，包括设计图或者可执行的软件，并与需求方确认主要流程和关键细节，避免遗漏或误解。该原型系统可以浏览，甚至可以交互（mock up，随机的输入和预定的输出）。系统设计师要给出概念验证原型和技术选型建议。详见第 5 部分。

4.2.4 设计

即系统分析与架构设计（System Analysis and Architect Design）。

人员：系统设计师，视觉设计师，主要的开发人员。
产出：架构设计文档，视觉/交互设计文档，或概要设计文档。
时间：从原型确认后开始，尽量短，不要超过总时间的 30%。

对于大中型系统，需要架构设计，因为要考虑框架的灵活性、可维护性等。在架构设计文档中要包括 4+1 视图：

- 场景视图
- 逻辑视图
- 进程视图
- 开发视图
- 物理视图

任何规模的系统都需要概要设计，详见第 5 部分。有时候概要设计可以放在下一个阶段（开发）。

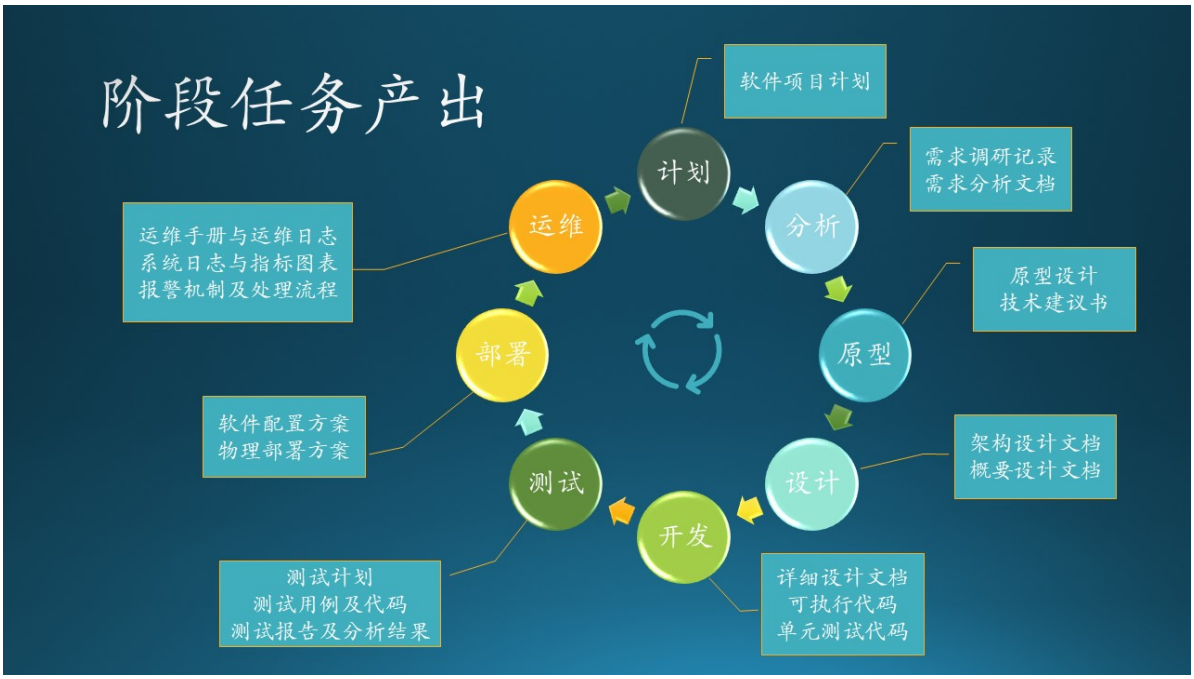


图 4.2.2 每个阶段的任务产出

4.2.5 开发

即软件编码、单元测试（Coding and UnitTest)。

人员：软件开发工程师。
产出：概要设计文档、详细设计文档、可执行代码、单元测试代码。
时间：从架构设计初稿形成后即可开始，开始时间不要晚于 40%，结束时间不要晚于 85%，要给后期的工作留出空间。

在开发初期，根据项目需求做概要设计或详细设计，一般情况下，把概要设计写得稍微详细一些，就可以代替详细设计文档。详见第 5 部分。

在整个开发阶段，软件工程师全力以赴地持续工作，需求不能再做任何较大的修改。那么在前面的 25% 和 后面的 15% 的时间内，软件工程师是处于空闲状态吗？当然不是！他们可能处于以下几种状态：

- 在别的项目内工作；
- 协助做概念验证或原型；
- 协助解决部署或运维中的问题；
- 充电积累，参加培训，学习新技术，为新项目做准备。

4.2.6 测试

即集成测试（Integrated Testing）、系统测试（System Testing）。

人员：测试人员。
产出：测试计划、测试用例、测试代码、测试报告、静态分析、动态分析结果。
时间：随着开发的开始，测试越早借入越好，结束时间不能晚于 90%。

从开发阶段刚开始的时候，测试任务就要启动，只不过开始时密度不大，但是大头在后面，尤其是到了压力测试、可靠性测试等阶段，更是要 24 小时地运行测试代码。

所谓静态分析、动态分析，是用一些工具对程序的安全性进性分析，避免已知的安全漏洞，比如使用了那些已经发现有安全隐患的第三方库。

4.2.7 部署

人员：工程师。
产出：软件配置方案、物理部署方案（包括回滚方案）。
时间：在测试接近确认时就可以开始准备部署。但是，实际的部署方案其实早在系统设计阶段就已经给出了。

部署环境包括两个：集成环境、产品环境。首先在集成环境中部署，经过测试成功后再迁移/切换到产品环境。如果能做组件级别的回滚，就可以直接在产品环境中部署局部节点。如果不能，就要准备独立的产品环境，用于一次性的切换。

4.2.8 运维

人员：运维人员，项目经理。
产出：日志、图表、报警机制、backlog。
时间：从系统上线开始，一直持续到产品生命周期结束。

运维人员要随时监控系统的运行情况，有问题即时处理，定期向上级提供运行情况的总结图表（比如总用户量、并发量、系统负载情况等）。产品经理要引导大家在运维阶段把一些想法和改进意见写入 backlog 中。

4.3 研究型项目的开发流程

所谓“研究型项目”，就是研究员主导的项目，通常是使用一种新的技术或算法解决用传统方法解决不好或者不能解决的问题。

4.3.1 项目 A

MARO - Multi-Agent Resource Optimizaiton 多智能体资源优化平台

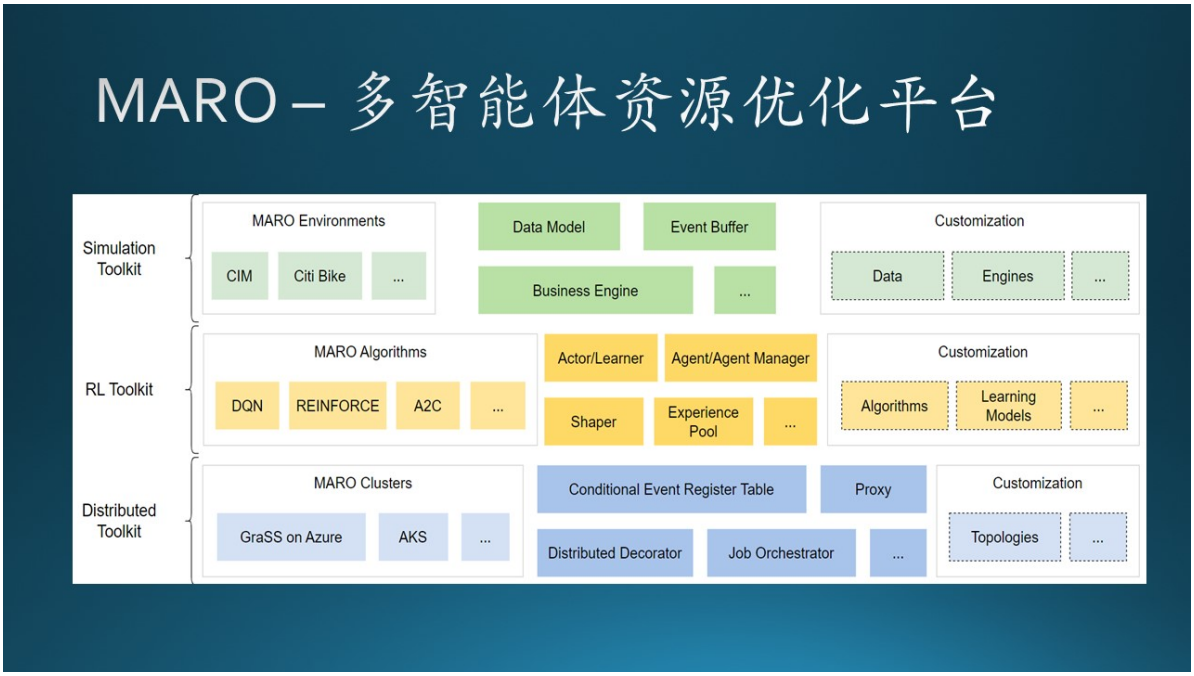


图 4.3.1 多智能体强化学习平台架构图

项目规模

这是一个研究型的项目：一个 dev manager，一个 tech lead，六个 dev（2 FTE + 4 Vendor），两个研究员，一个 vendor designer，没有 PM。

项目背景

一个跨国海运公司在世界各地的港口都有空的集装箱，需要及时地转运到所需要的港口装载货物。如果用传统的优化算法，每次计算需要花费好几天的时间，而且还不能克服一些临时的因素造成的误差，造成空集装箱没有正确地达到需要的港口。

所以研究员们用多智能体强化学习算法，成功地解决了这个问题。

和这个海运公司的合同结束后，研究员和工程师们想把这个算法发展成为多智能体强化学习平台，以便可以方便地解决类似问题。为此，建立了一个开源项目：MARO（Multi Agent Resource Optimizaiton）。

但是只有海运这样一个实例的话，是无法泛化的，于是研究员和工程师们又找了共享单车搬运、虚拟机管理、仓储库存管理等几个例子。

资源调整

这个平台开发了一年多后，那个 tech lead 要离职，老板让木头接替他。一进来就发现 tech lead 定的例会是每周一、三、五，这就很尴尬：因为周五以后就是周末，如果每个 dev 想在周一的例会上说出点儿什么来，就必须在周末花时间做出进展来。大家轮一圈儿后大概要花 30 分钟，而且那个 tech lead 特能说，每次都讲很多。除此之外，在例会之外他还会经常和每个 dev 单独联系，一讨论就是一个小时，语气很强硬。

这位 tech lead 把摊子铺得很大，所以每天都很忙的样子。他和研究员的关系比较密切，对研究员是言听计从。他在离职和木头的交接期间，在他的授意下，有个研究员跳了出来主持例会，给各个 dev 安排任务，还给木头安排了一个文档任务，木头只是不置可否地笑了笑。木头和老板汇报了这件事，老板觉得是不可接受的："我们虽然开放合作，但是还没有开放到可以让别的组的人来指挥我们的 dev 工作。"

所以，木头接手后的第一件事就是把以前的例会取消，然后重新建立了每周二、四两天的例会，每次例会大概 15 分钟左右。没有邀请研究员参加，因为这是个 dev 的例会。

很快，木头发现其实这是个画饼充饥的项目，没有明确的用户，没有产品化目标，没有明确的计划，相当于是一个工程团队在搞研究，只是在 GitHub 上开源了一个项目。

于是木头和领导建议，把部分人力调配到了量化交易的项目上，这边只留三个 dev（1 FTE + 2 Vendor），继续支持研究员关于仓储管理的项目研发。

两个月后，领导和木头说，把这个项目交给 3.4 节中提到的女生 E，木头很痛快地答应了，但是每周的两次 sync 还是几个项目一起开，因为人员上有交叉。木头发现女生 E 作为 tech lead，她自己很少写 code，只做 code review。但这与木头无关，索性忽略。

项目分析

从用户需求上看：

- 没有明确的用户需求，所有需求场景都是研究员和工程师们自己想的，是"伪需求"。这是一个致命的问题。在这一点上不成立的话，就决定了这个项目不会有什么好结果。
- 这是存在于研究员群体的一个通病。在完成了一个研究项目并有了实际的成果转换后，大家都想复制这种成功，于是想把"项目"变成"产品"，把"产品"变成"平台"，把"平台"变成"服务"。这种试图"泛化"的努力是违背了软件项目的基本规律的：还没有找到规律，就想建立模型。

从问题领域上看：

- 基于统计的机器学习，和基于特征的深度学习，都已经被广泛地接受并落地于实际应用场景中。强化学习对于大多数人来说还很陌生，虽然它能解决很多传统机器学习与深度学习不能解决的问题，但是数据如何收集整理，算法如何使用，都是需要专业指导的。所以用户对这个平台的认知程度很低。
- 虽然强化学习的应用范围也很广，但是这方面的领域专家太少了。一方面，我们的研究员没有领域知识，不知道强化学习应该用在哪里；另一方面，有领域知识的人没有强化学习知识，而是寻求传统的优化求解思路，或者把目光转向机器学习和深度学习技术上，强化学习的入门门槛很高。所以，很多人对强化学习的了解还仅限于 Alpha Go 上。
- 即使用户真的熟悉强化学习，而且又有领域知识，这类用户也不会考虑使用这个平台的。因为了解这个框架本身就需要花时间，而且万一这个框架有什么潜在的问题的话，如何能判断出是框架的问题还是问题领域本身的数据或算法问题？所以使用这个框架的风险太大。

从团队组成上看：

- 没有 PM 就代表了没有需求、产品目标、商业计划；
- dev 的配置很齐全，有开发目标，但目标是否正确，全凭 tech lead 一个人说了算；
- 居然还有一个 designer，尽管视觉设计工作非常少。举个例子来说，你觉得在 SK-Learn、PyTorch、Tensor Flow 这种框架上需要有 designer 帮忙吗？

从开发流程上看：

- 因为与前面所说的跨国海运公司签的合同是挣了钱的，所以在那个光环的照耀下，这个项目得以"轰轰烈烈"地进行了一年多。
- 例会在时间上安排得很不专业。
- Tech lead 没有 design doc，不写 code，只做 code review，这是不对的。
- 没有定义时间节点明确说明要有哪些 milestone 发生。

- 实际上是由研究员指定的软件开发相关的工作内容，这是前所未闻的。
- 其实最关键的还是团队的老板，看不清这个项目的未来发展，即不懂技术又不懂管理，被人家画了个饼，却始终吃不到嘴里。
- 还有一点很可笑，所有的文档都是英文的，虽然所有参与人员都是中国人（让木头想起了在美国的墨西哥人为了谋生而开中餐馆）。英文可以有，但是写中文文档不是手到擒来的事情吗？用中文来降低入门的门槛是不是一种合理的考虑呢？

最终结果

这个项目开源三年多了，至今虽然还在有人维护，但是处于半死不活的状态，在 GitHub 上只有 660+ 颗星。

做这种决策一定要经验丰富且小心谨慎，因为这种项目是不能快速试错的，花了一年时间，没有用户，会觉得是功能不完善；于是再花一年时间，还是没有用户，会觉得是因为没有好的例子帮助用户理解；于是再花一年时间补充很多例子，仍然没有用户，才发现是个伪需求——我们正在试图生产/销售火车的方向盘。

4.3.2 项目 B

Qlib - Quantitative Library 量化交易平台。命名为一个 Library，意为用户只要调用 API 就可以完成指定的任务。但实际上，它是一个 Framework。也就是说：用户必须按照 Qlib 所规定的流程

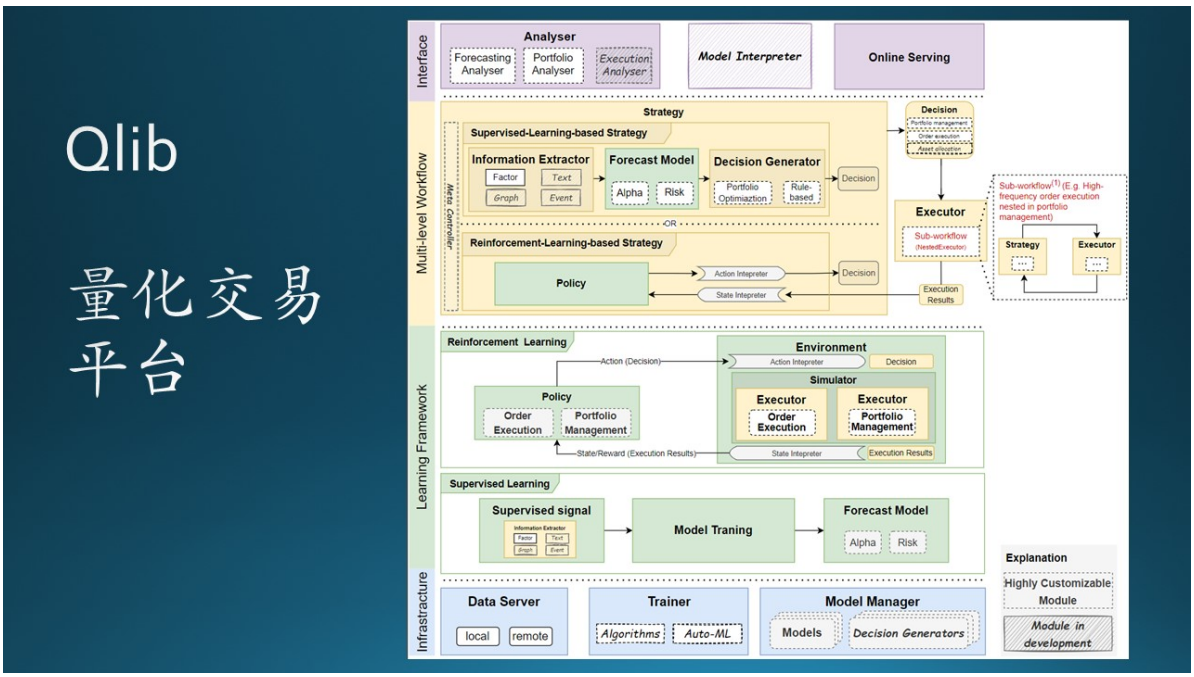


图 4.3.2 Qlib 量化交易平台架构图

项目规模

这是一个研究型的项目：一个 Dev manager，一个 Tech lead，五个 dev（3 FTE + 2 Vendor），三个研究员，一个 PM，两个商务人员，一个外部的长期合作的客户（某基金公司）。

项目背景

研究员们研究股票的量化交易很多年了，并建立了一个开源项目：Qlib。可是缺少实盘的机会。某基金公司考察后决定合作，基金公司提供数据，由研究员们提供算法、模型、预测结果，基金公司参考预测结果再结合以往的投资经验来操盘。

要解决的问题

每个交易日，基金公司都要手工上传昨天的交易数据，用微信叫醒熟睡的研究员，研究员去下载交易数据，处理后用模型做预测，然后微信通知基金公司取结果。整个过程基本没有自动化。

每过 60 个交易日，基金公司就要求研究员使用最近的历史数据重新训练模型，在历史数据上比较新旧模型的差异，然后切换模型。这个过程也都是手工完成，很繁复。

木头的任务就是把上述过程工程化、自动化，省去手工操作的麻烦和误操作。

开发流程

- 每周二、四，木头会安排组内会议，大家 sync 一下当前进度。
- 每周五收盘后，和客户开一次例会，由研究员分享最新的研究成果。开会时客户那边一般是两三个人，我们这边三个工程师、两个研究员、两个商务人员、一个 PM 都要参加。
- 每两周，开发团队要和研究员们做一次内部例会，工程师们汇报系统设计及开发的进展。
- 木头作为 Tech lead 负责开发团队，安排一个 dev 做推理预测部分的工程化，两个 dev 接手改造陈年失修的数据处理 pipeline，两个 dev 做训练部分的工程化。
- 在正式开始之前，木头先把任务做了粗粒度的分割，估算每个部分所需开发时长，并用此时间来对内对外宣布，一是给内部人员一个明确的时间/任务安排，二是给外部人员一个预期。
- 木头自己画了很多图，并配以较少的文字，来说明整个系统的架构，一是用于对内统一思想，一是用于对外汇报。
- 每次做一个新模块，木头会要求相关人员写一个简单的设计。因为木头发现，某些 dev 很快就能做出一个功能模块来，但是在 sync 时发现设计太复杂，不利于理解和后期维护。
- 所有工作都在 Microsoft Azure 上进行，因为客户买了 Azure 的订阅。

项目分析

从用户需求上看：

- 每天的预测要在数据上传后 90 分钟内完成；每隔 60 个交易日要使用历史数据重新进行模型训练；要求在 2 天内训练完 160 个神经网络模型；最新模型的预测结果应该与旧模型匹配程度极高才能上线。
- 用户只有一家，用户说什么就是什么，没有讨价还价的余地。因为涉及到上亿的投资规模，所以用户很谨慎，每上一个新功能都要严格把关，要求我们能够做到自验证。
- 甲方强势，乙方必然弱势。微软这边投入了大量的人力，但是合同金额并不大，只是为了能够把研究成果市场化。因此，甲方会时不常地提出一些临时需求，我们这边就需要从正常的开发流程中抽出人手来帮助分析数据和模型行为。
- 用户才不关心我们用什么平台框架，但是研究员们一心想推广 Qlib，所以想让工程师们使用 Qlib 完成工程任务。这里就会有问题：以往的模型都是用那个陈年失修的 pipeline 来完成数据处理的，然后去训练模型。如果用 Qlib 做数据处理，新训练出来的模型与旧模型差异较大的话，如何向用户解释？基于此，木头拒绝了使用 Qlib 的要求（这令研究员们很不满意），先满足真实的用户需求。

从问题领域上看：

- 量化交易，其实只有大户才能用得上，因为它并不是靠对个股预测的准确性来获利，而是靠对股票组合的抗干扰性预测来指导大户投资的。但是散户往往对此有更浓厚的兴趣，希望通过预测个股来赚钱，但实际上是不可能的，否则研究员们为什么不闷声发大财？
- 大户虽然更合适，但是这种项目有个致命的问题：不能推广。比如，客户 A 已经使用了我们的量化交易系统，我们又找到了客户 B，人家会问：有别的客户使用你们的系统吗？如果你说“还没有其它客户使用”，那客户 B 会觉得不放心；如果你说“已经有客户使用了”，那客户 B 会质疑：如果我们也使用，那么和客户 A 的算法就是完全一致的，在股票市场上就会造成严重的买卖同质性。

- 所以，完成该基金客户的实际需求，是我们工程师应该追求的目标；而使得 Qlib 发扬光大，成为众多量化交易研究者的平台，是研究员才关心的目标。

从团队组成上看：

- 商务人员在整个开发流程中只是处于 RASCI 模型中的 informed 角色，并不实际做事情。
- 虽然有 PM，但是由于这是一个技术导向的不平等的合作项目，所以 PM 的大多数工作都是在组织会议，发会议记录，对于产品的方向没有发言权。
- 研究员想一方面推出 Qlib，另一方面想在实盘交易上有突破，因此在实际的工程项目中也是 informed/Consulted 角色。
- 工程师是完成用户需求的主要力量。

从开发流程上看：

- 每周两次的内部 sync 是必要的，这是项目流程的基本需要。
- 每周一次的和客户交流也是未尝不可的，但是没必要这么频繁，这主要是研究员们过于 push 自己了，其实对于客户来说，每两周一次也没有问题，因为合同的周期是半年。
- 每两周一次的内部交流，纯属是工程师对研究员的单方向汇报，笔者认为跑偏了，相当于工程师是研究员的下属，而不是合作关系。但如果是双方互相交流的话，那就是另外一番融洽的合作场景了。
- 从与客户的关系看，由于以前研究员们对客户过于“尊重”了，真的把客户当成“上帝”了，搞得 PM 根本拦不住也不敢拦客户的随机要求，这会打乱软件工程活动的正常进行。

最终结果

工程师们用了四个月的时间，终于把所有的训练系统、推理系统、模型管理系统做到了自动化，圆满地完成了第一次模型切换，客户很满意。

4.3.3 总结

经过上面两个项目，我们总结一下工程师与研究员合作一些研究项目时需要注意的事项。

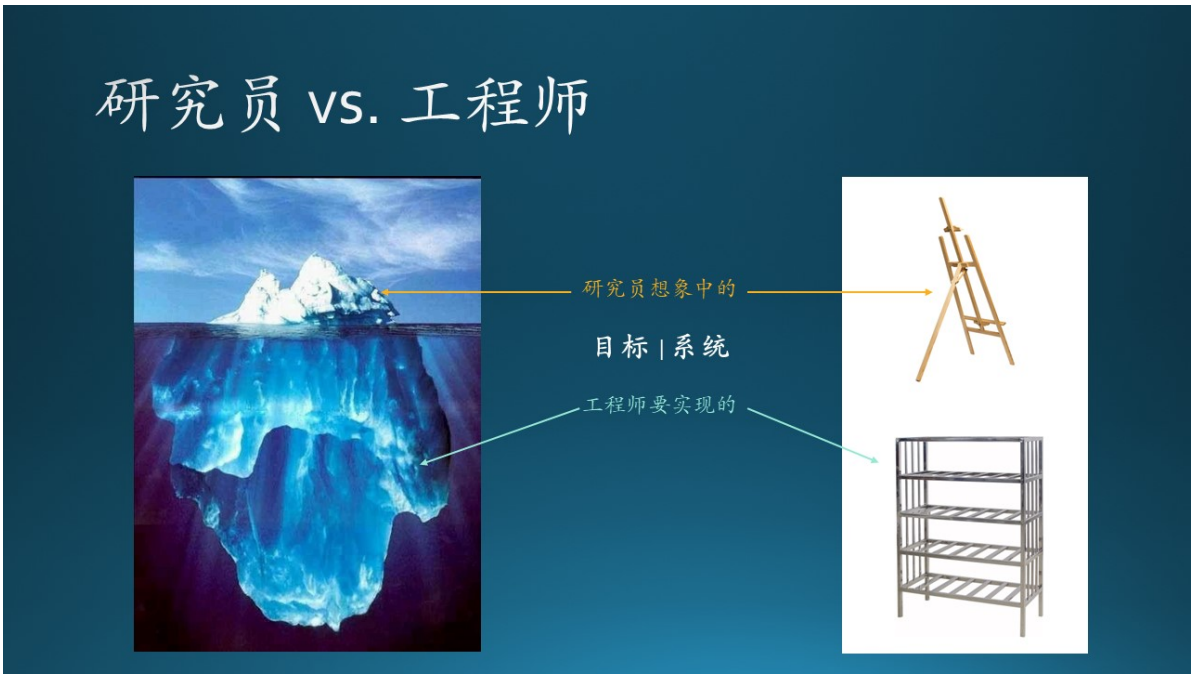


图 4.3.3 研究员与工程师对软件系统的不同理解

目标不同

在强化学习项目中，研究员的目标是做一个能让大家使用的做多智能体强化学习研究的平台，但是它的泛化能力太差，以至于无人问津。而工程师在这个项目中处于辅助位置，在领域知识上处于劣势，基本上要听从研究员的安排。这种项目，成功的概率很小。

在量化交易项目中，研究员的目标是让工程师使用 Qlib，而工程师的目标是要切实完成用户的需求。如果使用 Qlib 的过程遇到什么问题而不能完成需求，那么要承担责任的是工程师，而不是研究员。况且学习 Qlib 还需要时间，而距离下一次模型切换只有 60 个交易日。

背景不同

在量化交易项目中，第一次开会时，研究员提出了上述任务，要求两个工程师 + 两个实习生来完成这些任务。木头心里估算了任务量后，当时就提出：你们的资源要少了！第一，我们工程师和你们研究员不一样，research 领域中主要是实习生写 code，而工程领域中实习生是不能写 production code 的。第二，这些任务的实际工程工作量远超出你们的想象，两个人是不可能完成的，至少需要五个人。

研究员什么时候有能力估算工程师的工作量了？

做法不同

在量化交易项目中，用户买了 100 万的 Azure 订阅，用于训练模型。但是以前的研究员并没有充分使用，而是使用公司内部的 GPU 平台来训练模型。导致用户认为根本不需要购买很多的订阅额。

木头在工程化过程中，充分使用 Azure 上的 GPU 订阅额（quota）建设自动化训练系统，经过了多轮的测试以验证其稳定性，结果用爆了用户的订阅额，促使用户在下一期合同的时候购买了翻倍的订阅额，商务部门很高兴。

研究员在数年的研究过程中积累了一些数据处理代码，都放在一台运算能力超强的计算机上，并不断地以打补丁的方法添加新的功能。这台计算机没有人敢关机，也不敢修改里面的代码。这就是一个典型的研究型黑盒子。

木头在工程化过程中，专门指派了一个 dev 复制了这台计算机，包括环境和代码，然后在副本计算机上做各种探索，终于搞清楚了每一步的输入输出和整个流程，让黑盒变成了白盒，为后期使用 Qlib 成为可能。

最佳实践

- 如果是一个项目处于刚开始的研究阶段，研究员和工程师需要积极配合，研究员是 R 的角色（负责），工程师是 S 的角色（支持），以研究为导向。
- 但是一旦项目成熟了，就应该进入正式的系统开发阶段，以工程为导向，工程师是 R 的角色，研究员是 C 的角色（顾问），就不要再指手画脚了。
- 研究员应该有深度的领域知识传播，而工程师应该有宽度的工程知识介绍，双方各取所长，并非谁领导谁的问题，而是在项目的不同阶段谁主导的问题。在量化交易项目中，每两周一次的内部例会只是工程师单方面输出，搞得气氛很怪异。
- 工程师在与研究员合作的过程中，切记自己的使命，保证代码质量，确保最终产出的不是脆弱的试验室产品。
- 不要轻易去搞什么研究平台，研究员往往陷于“自恋”中不能自拔，这是他们的工作性质所决定的。对于工程师来说，能够判断出哪些需求是真实的，不要拿起来就做。学会正确地做事，也要学会做正确的事，别浪费自己的时间。
- 研究员要给与工程师的合作充分的肯定。在量化交易项目中，工程师们忙活了好几个月，研究员的评价却是“他们只做了一些修修补补的工作”，这是不够负责任的评价。
- 在研究院这种组织中，工程师往往没有 credit，而 research manager 也往往不懂得如何评价工程师的工作绩效。所以工程师们要有心理准备，并且可以用适当的方式给与你的 research manager 一些 education。“教育老板”是微软的一种文化。

4.4 小型软件的开发流程

4.4.1 项目 C

Torch Nebula，基于 PyTorch 的分布式训练进度的云缓存。

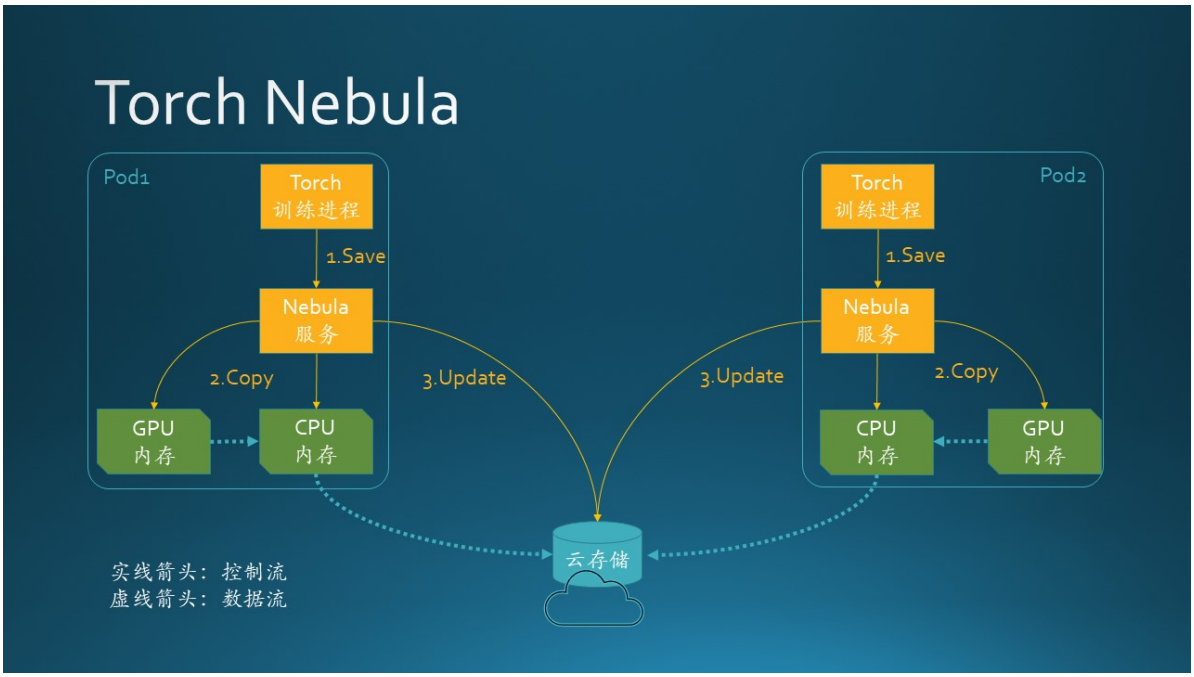


图 4.4.1 Torch Nebula 项目工作原理图

项目规模

这是一个小型项目：一个 dev manager，一个 tech lead，四个 dev，没有 PM 和 Designer，first party（内部）用户。

项目背景

在使用 PyTorch 训练大型神经网络模型时，往往使用分布式训练，以数据分布式居多。这种模型往往需要几天甚至几周的时间才能训练完毕，这就需要在训练过程中间歇性地保存当前的训练进度，以备万一机器掉电或框架软件错误造成中断时，可以把最近一次的保存进度加载进来继续训练。

要解决的问题

由于网络参数过大，调用 torch.save() 函数保存进度写入硬盘往往需要一个多小时，然后才能继续训练，是一种阻塞式的行为。我们的目的就是要提供一个可导入的 python 包，当用户调用这个包提供的 nebula.save() 函数时，可以用异步的方式在后台保存进度，只需要 1 分钟左右的延迟就可以继续训练任务。

关于例会

刚一加入这个项目，就发现他们每天上午 11 点都有例会，通过 Teams 开的在线会议，大家 sync 进度，scrum master 是 Tech Lead。他会询问每个人的进度，有问题的话就讨论以下解决办法，或者提醒要注意的关键点。经常进行一些发散性的讨论，造成了一轮 sync 下来会花 40 分钟以上的时间。

初来乍到，木头也不好说什么，就听着呗，轮到自己时就简单说两句。这种每天 sync 的好处就是：Tech Lead 可以清楚地知道每个人的进度及当前状态，而每个 Dev 如果想在第二天的会上说出点儿什么来，必须在前一天花时间完做出质性的进展，很大程度上杜绝了“摸鱼”现象。

但是，每天花 40 分钟有点儿太耽误大家的时间了。另外，项目本身的难度不是很小，所以每天不一定有实质性进展。

开发流程

- 木头是以帮忙的身份来加入这个项目的，从零开始建立一个 CI Pipeline，可以把 dev 们提交的 code 在 Pipeline 里自动编译、打包、部署到测试环境，如果测试成功则允许 merge。
- 团队除了每天上午开一次例会以外，每周五下午向上层的老板做一次回报，本周做了什么，下周准备做什么，用户有什么反馈等等。老板关心的问题就是：有多少用户在用了，使用情况如何。
- 在日常工作中，如果用户或者我们自己发现了什么 bug，就记录在 Azure DevOps 上面；如果有什么新想法，也会以 Feature 的标签记录在 DevOps 上，作为 backlog。
- Tech lead 在例会上看每个人的进度，如果没有任务积累，就会从 backlog 中拿出一项优先级比较高的任务来分配给这个人。当然，像木头这种比较资深的 dev，会自己主动挑任务：或者是自己感兴趣，或者是对了解全局有帮助，或者是自认为比较重要的。
- 每个人要做新功能时，都需要先写一个设计文档，可简可繁，tech lead 找相关人员评审后方可继续。
- 日常的 1:1 的讨论比较多，有时候会直接跑到对方的座位旁对着屏幕讨论。
- 大家都很友善，会帮助新来的同事答疑解惑，这让木头感到很舒服，比起上一个团队来可是强多了。
- 没有外部的需求要求我们做什么功能，都是自己想出来的。
- 每两周会总结一下进度，作为一个 Sprint，但是并不严格，因为很多时候计划的任务两周做不完，就会用三周时间。老板也不会催促。

项目分析

从用户需求上看：

- 需求是客观存在的，而且是刚需，但并不是由用户提出来的，而是由 feature team 自己发掘的。

从开发流程上看：

- 每天的例会是很必要的。
- 每两、三周作为一个 Sprint 是正确的。
- Tech lead 和 Scrum master 是同一个人，这是不正确的做法，导致每天的 Scrum meeting 太长。
- 在项目早期建设了 CI Pipeline 是正确的，但是没有及时建立有效的 Unit Test 机制是不正确的，导致新代码 merge 到主分支的风险很大。
- 实现每个功能前都需要一个简单的设计文档及评审，这是正确的。
- 每周向上级老板汇报一次未尝不可，但是如果能和 Sprint 周期同步就会更好。
- 没有明确的进度要求，只有一个大的方向。

所以，这是一个比较成功的团队流程。

最终结果

当木头离开这个项目组时，已经有不少内部用户使用了 Nebula，反响很好，也得到了领导的夸赞。下一步是想开源这个项目，但不是必须的。

4.4.2 小型软件的开发流程

我们以做一个计算器软件为例，说明开发一个小型软件的开发流程的最佳实践（Best Practice）模型，如图 4.4.2 所示。

图 4.4.2 小型软件开发流程的最佳实践

Make it Work - 能运行

首先，我们做一个比较粗糙的原型，只要能运行就行，结果可以是不正确的。

从功能上看：

1. 提供基本的输入功能，包括数字和运算符，但是只有“0~9,+,=”等 12 个按键，而且排列得七扭八歪；
2. 提供输出显示功能，能把输入显示在屏幕上，但是目前阶段只能显示“1,2,+,=”四种字符；
3. 输入“1+2”后，按“=”键，在屏幕上可以显示结果，但是结果可能是错误的“2”，而不是预期的“3”。

从界面上看：

1. 几个按钮的大小和位置不整齐，但不影响使用；
2. 显示框歪歪扭扭，但还能看清楚数字；
3. 外边框也不是很美观，用了很不协调的红色。

如图 4.4.3 左子图所示。

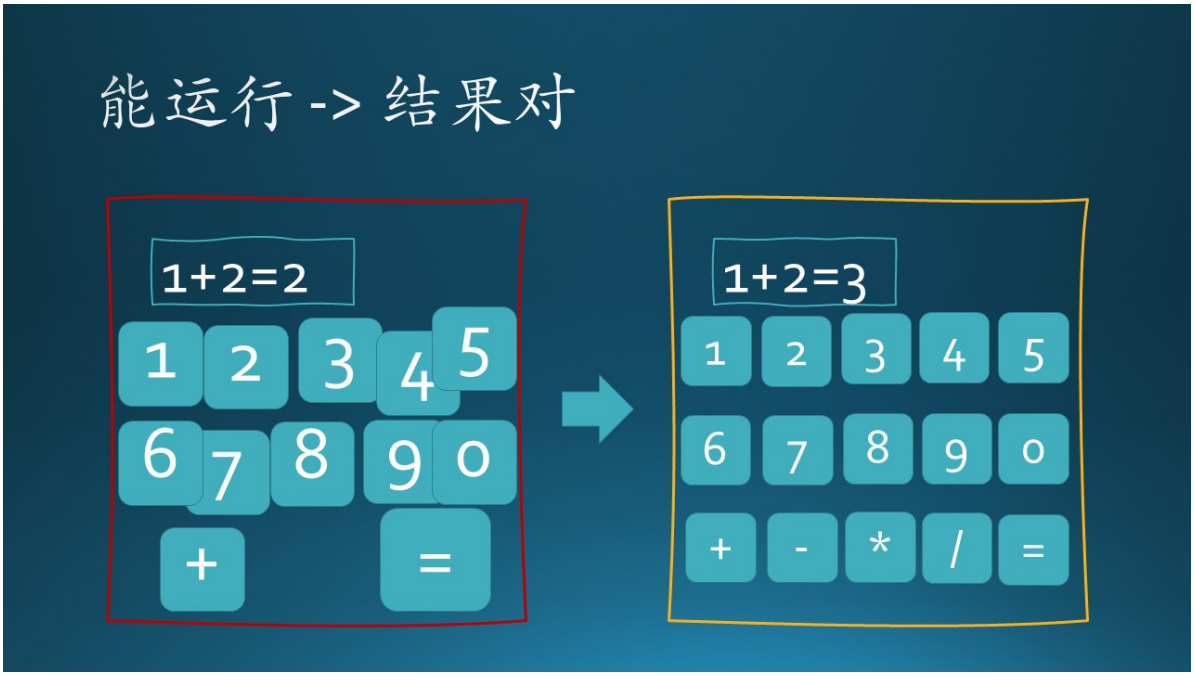


图 4.4.3 从“能运行”到“结果对”

Make it Correct - 结果对

本步中的关键是要求结果正确。

1. 原型可以工作了，但是结果并不正确，所以需要 debug 来排查错误。
2. 顺手把几个按钮的大小和位置调整得好看一些，补全了四则运算的按键，再把边框改成搭配的黄色，显得稍微懂一些美学，别太给咱们程序员丢脸。
3. 再次输入“1+2=”后，可以得到正确的结果“3”了。
4. 能显示更多种类的字符了，但是显示区长度还比较短，只能有 6 个字符。

如图 4.4.3 右子图所示。

Make it Fast - 算得快

本步中的关键是要让结果迅速地出现。

结果虽然正确了，但是发现再输入“=”后，要等 3 秒钟以上才得到结果。这是怎么回事呢?

经过代码检查和 debug 发现:

1. 输入的数字，首先被变成了字符串，在接收到“=”号输入后，从屏幕显示缓冲区把字符串取出来，解析成两组数字“1,2”和“+”号，再进行实际的数学计算。这个设计糟糕透了。
2. 不知道哪一位程序员脑洞大开，把本来可以在本地进行的计算设计成调用网络服务，还号称是可扩展的 B/S 模式，将来可以调用任何新实现的数学运算。
3. 最气人的是，为了显示美观，有人引入了一个巨大的第三方库，而且对该库没有进行任何安全性检查，不知道它都跑了哪些逻辑。
4. 最难以发现的是，每次运算都会触发一次日志记录操作，由于开发过程中的自动测试用例也写日志，导致日志文件变得越来越大，打开、寻址、写入、关闭文件的操作变得很慢。

当然，以上这些都是在原型阶段发现的问题，还有可以纠正的机会。

Make it Rich - 功能全

本步中的关键是要实现所有设计的功能，并保证性能没有明显下降。

在经过需求评审和设计评审后，这个项目进入了正式的实现阶段。已知的功能有:

1. 支持按键数字从0-9;
2. 操作符支持“加减乘除”;
3. 显示区可以显示 10 个字符了，在超过 10 个字符时，用户输入时会有“滴滴”的报警声;
4. 由于还没有收到 Designer 的设计图，界面上还是有些歪歪扭扭的，而且字体的选择也有些问题，上下边界没有对齐。

如图 4.4.4 左子图所示。所以虽然功能都已经实现了，还是没法交给最终用户使用。

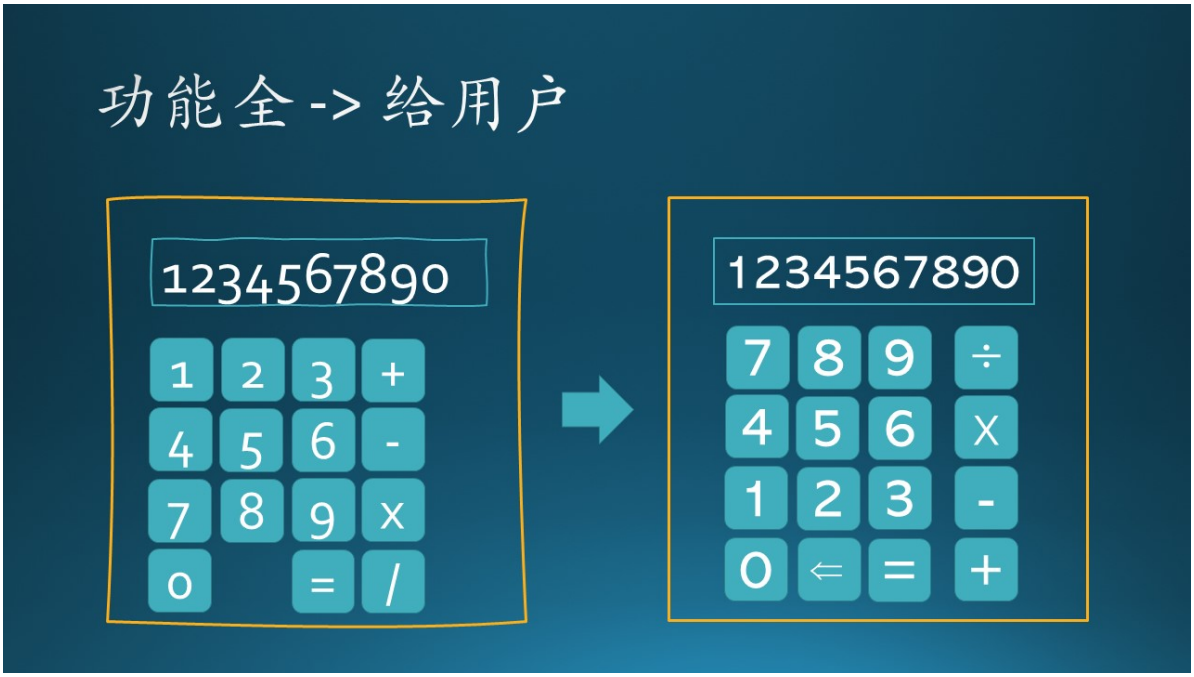


图 4.4.4 从“功能全”到“给用户”

Make it Live - 给用户

终于收到 Designer 的设计图了：

1. 0-9 十个数字的排列顺序让程序员们讨论了半天，为什么 “789” 在上面？找到了一个实际的计算器一看，确实是这样排列的；
2. “加减乘除”四个按钮的顺序也是颠倒过来的，并且与左侧数字去拉开了距离；
3. 字体统一使用等宽形式，上下边界也对齐了；
4. 按键区与显示区的中线对齐，上下左右的边界间距也都合理；
5. 增加了“删除”按钮，可以随时修改输入错误，而在此之前，程序员们测试时都是一通乱按，无所谓对错；
6. 界面边框不再是七扭八歪的了，很规矩。

如图 4.4.4 右子图所示。这样修改之后，再次经过回归测试，功能性能都没问题，这个简单的应用终于可以发布给用户使用了。

4.5 敏捷开发流程

4.5.1 软件体量和形式的变化

在 4.4 节中的例子中，我们采用的就是敏捷开发流程，这种流程是软件开发者在长期的工程实践中通过无数的失败案例总结出来的，虽然它不一定能保证未来的每个软件工程项目都成功。

下面我们来看一看敏捷模型的产生基础。

计算机的应用可以追溯到第二次世界大战前，是名副其实的“国家机器”。但是真正的商业计算机及软件，应该是从上个世纪 50 年代开始的，最初用于构建大型的复杂商业应用，后来发展成为个人计算机，又发展到移动设备。终端设备的变化导致软件的变化，而软件体量、内容、形式上的变化，必然会导致软件开发团队的理念和流程的变化。如图 4.5.1 所示。

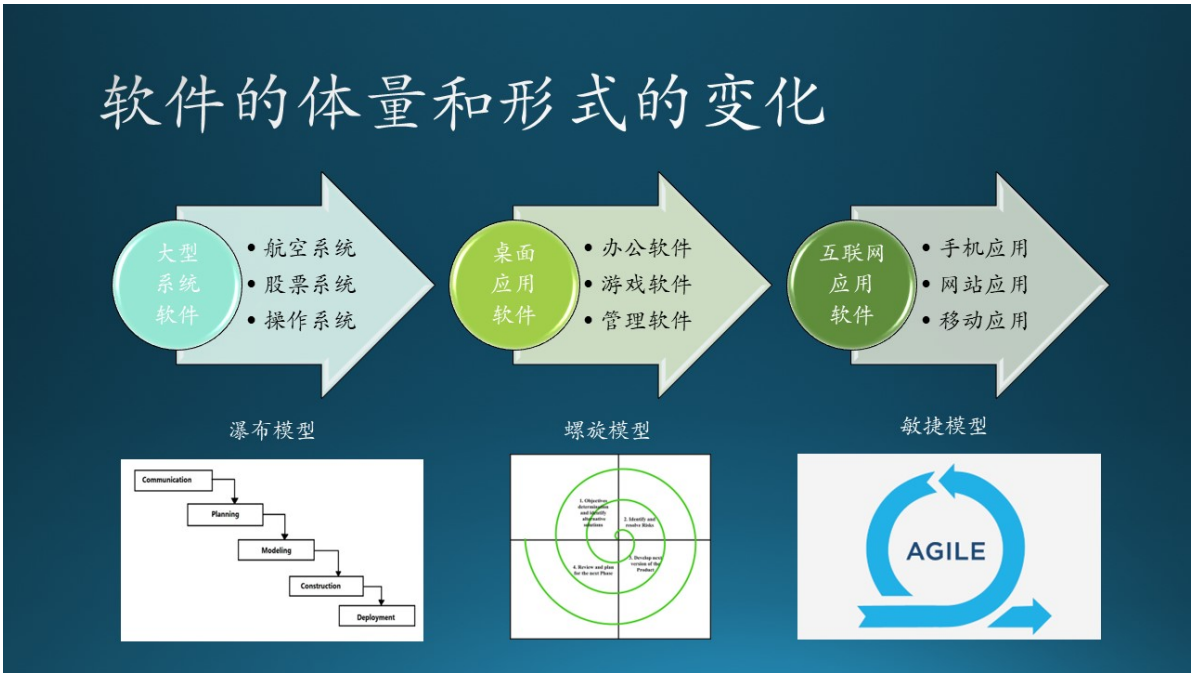


图 4.5.1 软件体量和形式的变化

大型系统软件

50 年前，软件系统是为了解决大型企业的应用场景的刚性需求，比如航空航天系统、金融股票系统、操作系统，可以用微软的 Windows 操作系统作为代表，结构复杂，投入巨大，开发周期长，维护时间也长。这些系统不是所有软件公司都能做的，所以才出现了微软、IBM、Oracle等这些大型软件厂商。像 Windows 操作系统，就是这种类型。

这时的软件开发流程一般采用瀑布模型，上游的任务全部完成后，再开始下游的任务。

桌面应用软件

30 年前，个人电脑普及到办公室，需求场景下移到了中小企业的应用上，比如办公软件、企业管理软件，还有面向大众的游戏软件。这类软件需求多样，需要定制，一般的中型软件公司经过一定的积累后也可以完成。可以用微软的 Office 这类运行在 PC 上的办公套件作为代表。

这时的软件开发流程一般采用螺旋模型，通过对某一阶段的多次重复来保证其完整性、正确性。

互联网应用软件

10年前，随着互联网的全面普及以及智能手机的发展，需求场景进一步下沉到了个人用户上，成百上千的网站、手机应用，以及其它 IoT 设备（如GPS,智能手表,智能手环,智能监控）上的移动应用出现在应用市场上。

此时，一般采用敏捷模型来应对快速的市场变化。

4.5.2 敏捷宣言

敏捷宣言（Mainfesto for Agile Software Development）。

2001年2月11日至13日，在犹他州瓦萨奇山的雪鸟（Snowbird）滑雪胜地洛奇酒店，17位软件开发领域的领军人物聚在一起聊天、滑雪、放松、并试图找到共同点 。

参会者们包括来自于极限编程、Scrum、DSDM、自适应软件开发、水晶系列、特征驱动开发、实效编程的代表们，还包括了希望找到文档驱动、重型软件开发过程的替代品的一些推动者。所有这些方式都是“轻量版”的框架，因为这些方法使用更少，更简单的规则来适应快速变化的环境。不少与会者都觉得“Agile”这个术语非常适用。

经过为期三天的讨论，他们在价值观和原则层面上达成共识，选择了 Agile 一词并为其赋予了特殊的意义，制定并发布了软件行业历史上最为重要的文件之一：敏捷宣言。

参会者将自己命名为“敏捷联盟（The Agile Alliance）”，希望能够帮助软件行业中的其他人以新的、更敏捷的方式思考软件开发、方法和组织。而“敏捷宣言”则被展示在一个网站上，到目前已被翻译成了60多种语言，并作为一种信仰被推广至全球以及非软件行业。

敏捷宣言的原文如下：

我们一直在实践中探寻更好的软件开发方法，身体力行的同时也帮助他人。由此我们建立了如下价值观：

1. 个体和互动高于流程和工具
2. 工作的软件高于详尽的文档
3. 客户合作高于合同谈判
4. 响应变化高于遵循计划

也就是说，尽管右项有一定的价值，我们更重视左项的价值。

误区一：个人英雄主义。

现代的软件规模已经不是几个人就能搞定的事情了，团队合作需要的流程和工具在日常开发管理和大规模作战时很重要。

误区二：多写代码少写文档。

文档是为了形成共识，避免人员流动造成的上下文缺失和误解，从而影响软件质量。

误区三：让合同见鬼去吧！

合同可以保护开发团队的基本利益，维持正常的进度，避免接受不合理的需求变更。

误区四：不需要制定什么计划，反正到时候也不遵守。

计划可以给与团队和客户基本的进度预期，当然是在预先谈好的需求范围内，这样可以避免客户每天都催促。

4.5.3 敏捷宣言遵循的 12 条原则

原文中的 12 条原则是按照一个不规则的顺序列出的，笔者把它们归纳到四大核心价值中，并附加了一些解释，便于大家理解。

图 4.5.2 敏捷宣言及其 12 个原则（分类）

1. 个体与互动高于流程和工具

- **激发个体的斗志，以他们为核心搭建项目。提供所需的环境和支援，辅以信任，从而达成目标。**
  误解：项目成功全都靠一两个大佬儿。
  正解：在团队中提倡以老带新的优良传统，调度积极性，发挥主观能动性。
  重点词：个体。
- **不论团队内外，传递信息效果最好效率也最高的方式是面对面的交谈。**
  误解：君子动口不动手。
  正解：进行必要的谈话避免误解。注意谈话的范围不要太海阔天空。在谈话完成后，形成有效的结论文档，告知整个团队，避免一件事说很多次。
  重点词：交谈。
- **最好的架构、需求和设计出自自组织团队。**
  误解：找合得来的人一起工作。
  正解：在现有人选中，让合适的人承担合适的角色，形成和谐的团队工作模式。所有的功能、特性由团队决定。
  重点词：自组织团队。

2. 工作的软件高于详尽的文档

- **经常地交付可工作的软件，相隔几星期或一两个月，倾向于采取较短的周期。**
  误解：经常更新会让用户更满意。
  正解：根据软件的规模、当前的阶段、用户使用场景，决定更新的频率。一个商业软件用户并不希望不断地得到更新，这反而会降低对软件质量的信任。
  重点词：经常交付。
- **可工作的软件是进度的首要度量标准。**
  误解：软件好用就行了，文档无所谓。
  正解：在交付时，对于用户来说文档和软件同样重要，只是使用的频次不同。联机文档可以作为软件的一部分同期发布。
  重点词：可工作的软件。
- **以简洁为本，它是极力减少不必要工作量的艺术。**
  误解：先把代码完成，测试和文档以后有时间再补。
  正解：设计和实现要简洁，但是该有的测试和文档一个都不能少，养成良好的工作习惯，简洁不等

于缺失。少开一些不必要的会议，减少不必要的流程。
重点词：简洁。

3. 客户合作高于合同谈判

- **我们最重要的目标，是通过持续不断地及早交付有价值的软件使客户满意。**
  误解：用户满意度高于一切。
  正解：满足客户的合理需求，而不是那些异想天开的、改了又改的需求，对额外的需求适当地收费也是合理的。
  重点词：客户满意。
- **业务人员和开发人员必须相互合作，项目中的每一天都不例外。**
  误解：业务人员必须盯紧开发人员的一举一动。
  正解：开发人员当遇到模糊的需求时，应该找业务人员核实，挖掘出客户的真实需求。
  重点词：合作。
- **敏捷过程倡导可持续开发。责任人、开发人员和用户要能够共同维持其步调稳定延续。**
  误解：加个班把进度向前赶一赶，后面还有很多事情要做。
  正解：正确估算开发时间，制定好计划，每天以 90% 的负荷工作，可以持续地、精力充沛地工作。
  重点词：可持续。

4. 响应变化高于遵守计划

- **欣然面对需求变化，即使在开发后期也一样。为了客户的竞争优势，敏捷过程掌控变化。**
  误解：需求发生了变化，前面的工作成果要推倒重来。
  正解：首先要分析为什么会有这种变化发生，其次要检查现有的架构是否可以接受这个变化，再次要分析哪些模块是可以重用的，哪些必须重写。
  重点词：变化。
- **团队定期地反思如何能提高成效，并依此调整自身的举止表现。**
  误解：以后还要一起工作，低头不见抬头见，所以有些不愉快的事情就让它过去吧。
  正解：对事不对人地分析造成项目拖期或影响软件质量的主要因素是什么，提出改进意见。
  重点词：反思与调整。
- **坚持不懈地追求技术卓越和良好设计，敏捷能力由此增强。**
  误解：发现有不理想的设计时，随时修改架构及代码。
  正解：如果设计有缺陷，应立刻改进。但是如果现有设计不影响当期的软件功能，可以先按下不动，下一个阶段再重构。
  重点词：技术和设计。

4.5.4 敏捷开发流程

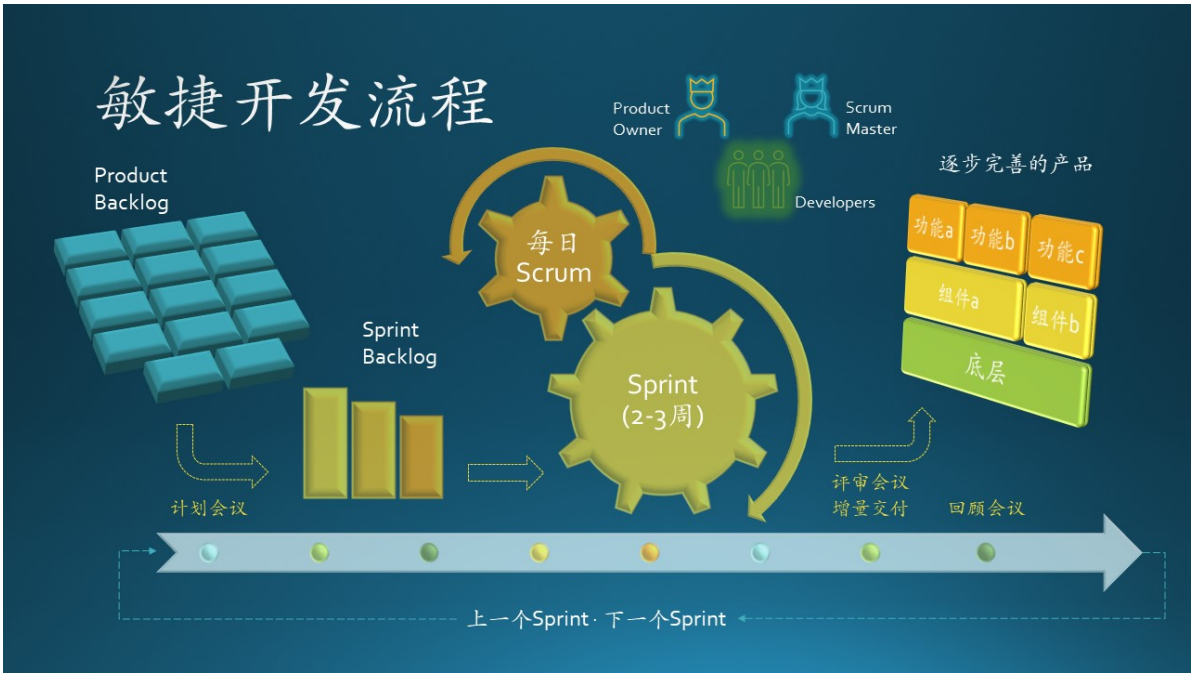


图 4.5.3 敏捷开发流程

三个角色

1. Product Owner 产品负责人

一般是 PM，在没有 PM 时可以是 Dev manager 或者 Tech lead。

2. Scrum Master 流程管理员

是产品负责人以外的任何人。
在 4.4 节的故事中，产品负责人是 tech lead，但他同时又是 Scrum master，这会造成他在每日例会时没有专注在流程执行上，而是陷入了技术细节讨论，最终导致例会时间过长。

3. Developers 开发团队

团队中的开发人员。开发人员可以轮流当 Scrum master。

三个工件

1. Product Backlog 产品（长期）待办事项

- 最初由产品负责人填写原始需求（Requirement），比如：“我们需要一系列数字化工具来提高员工的工作效率”。
  这一般无法实施，需要经过进一步加工。
- 经过粗加工后可以变成史诗（Epic），比如：“我们需要一个能力超强的计算工具来帮助员工省去用纸笔计算的负担”。
  但是这还不够具体，而且过于宏大，无法在屈指可数的时间内完成。
- 把史诗经过分析后，变成特性（Feature），比如：“提供一个可手工操作的电子计算器，和一个可以用程序控制的超级计算模块，可大并发使用”。
  这一级别的产品方向比较明确了，在“计算”这个数字化产品中，需要两个大的特性。
- 把特性经过分解后，变成故事（Story），比如：“作为一个物理研究员，我想要一个手工操作的科学计算器来计算一些物理量，这与就可以降低计算错误率，加快研究的进程。另外还需要一个可编程的计算模块，可以计算微分方程”。
  故事已经可以描述程序员能够看懂的功能了。
- 把故事变为任务（Task），并分配给每一个开发人员，或者由开发人员自己认领。比如：“实现四则运算模块”、“实现三角函数计算模块” 等等。

开发人员领到任务后，估算开发时间，填写到任务中。如果任务还是太大（超过 5 天），就继续分解成更小的任务。

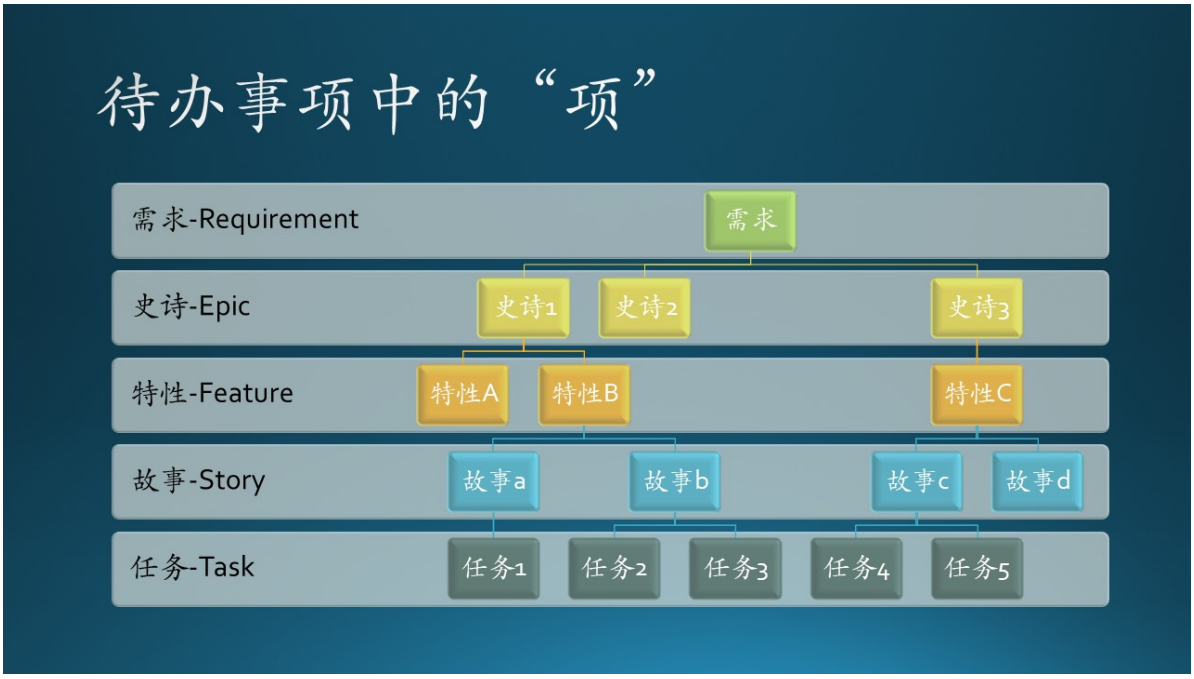


图 4.5.4 Backlog 待办事项中的内容项的层次关系

在这个backlog中，也可以没有特别具体的任务，到特性或者故事级别就可以。另外，在开发过程中，可以随时向里面填写新的事项，比如新的需求，或者列出已发的布产品中需要修补的问题，还可以是一些试验性的想法。所有事项必须有优先级。

2. Sprint Backlog 冲刺期待办事项

团队开计划会议时，把产品待办事项按照优先级提取出来，细化为冲刺期待办事项，必须细化到故事和任务级别，并确定这些冲刺期待办事项可以在一个 Sprint 内完成。

3. 逐步完善的产品

这是一个 Sprint 的增量输出，满足软件质量要求和功能需求后，叠加在已有的产品框架上，发布给用户。

五个事件

1. Sprint 冲刺期

一般为 2-3 周，太长了不好控制进度，太短了不容易完成可交付的产品增量。但是这个冲刺不是一次性的，而是头尾相连的，所以不需要在一个 Sprint 里玩儿命向前冲，而是考虑可持续的开发。

冲刺期可以事先制定好，注意避开节假日（或者相应地缩短），也可以在上一个冲刺期结束后再确定下一个冲刺期的长度。

2. 计划会议

在一个 Sprint 开始时要做的事情。

产品负责人招集大家开会，通过讨论把 Production Backlog 变成 Sprint Backlog，并最终变成任务分配给每个开发人员。

在这个会议中还需要推举 Scrum master。

3. 每日 Scrum

在一个 Sprint 中每个工作日都要做的事情。

在 Scrum master 的主持下，每个人需要说明三件事：

- 昨天干了啥；
- 今天要干啥；
- 有啥困难。

不需要特别多的细节，如果需要别人帮助分析细节的话，可以会后单独讨论。

在微软的工程实践中，绝大部分项目不需要做 Daily Scrum，而是每周两三次的频率，这样就不会显得很“卷”。另外，在跨地区的合作中，大家可以协商开会的时间，如果把会议订在了某个时区的早晨 8 点之前，是可以拒绝的，并要求调整时间。

4. 评审会议

在 Sprint 结束前要做的事情。

产品负责人邀请上级领导或其它利益相关者甚至用户参加评审，阐述在本 Sprint 中完成的任务，演示新添加的功能，得到反馈。然后确定哪些已完成的任务（功能）可以加入到增量交付中。会后由开发人员打包发布，产品负责人写发行说明（release notes）。

5. 回顾会议

在 Sprint 结束时要做的事情。

回顾会议只有团队内部成员参加。需要讨论几件事：

- 哪里做得好，是否需要继续发扬？
- 哪里做得不好，是否需要改进？
- 是否需要做资源、流程调整？如果需要，制定出事项和计划。

也可以叫做 postmortem（事后检讨）。

2、4、5三个会议有点儿罗嗦，可以合并成一个或两个会议，只要大家能分清楚开会时在讨论什么主题即可。

4.6 中大型系统的开发流程

4.6.1 项目 D

ONNX Converter。

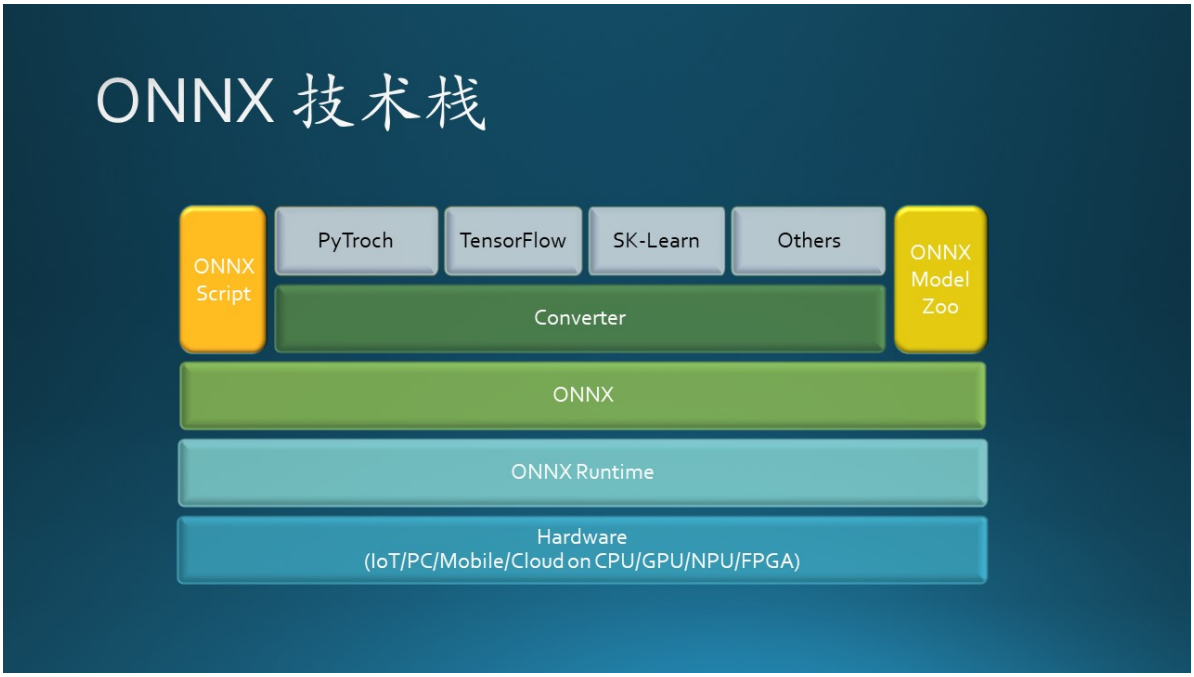


图 4.6.1 ONNX 技术栈

项目规模

这一次的项目个大家伙：ONNX 其实是一个产品，全部的开发人员大概有 200 多人。木头参与的是其中一个子项目 ONNX Converter，算是中等项目吧。一个 dev manager，没有 tech lead，10 几个 dev，而且分布在美国东西部、法国、中国等四个办公地点。

项目背景

ONNX（Open Neural Network Exchange）开源项目，开放的神经网络模型交换格式。作为框架共用的一种模型交换格式，使用 protobuf 二进制格式来序列化模型，可以提供更好的传输性能我们可能会在某一任务中将 Pytorch 或者 TensorFlow 模型转化为 ONNX 模型(ONNX 模型一般用于中间部署阶段)，然后再拿转化后的 ONNX 模型进而转化为我们使用不同框架部署需要的类型，ONNX 相当于一个翻译的作用。

在微软，所有训练出来的模型，最后都会用 ONNX 格式做线上推理，以简化环境部署，提高推理速度。

要解决的问题

用户使用各种深度学习框架搭建的模型，互相之间不能互相转换，而且各个框架着重于训练，而在推理方面则速度较慢。ONNX 作为模型推理平台，可以提升速度，简化环境部署依赖。

木头负责其中的 Scikit-Learn 模型到 ONNX 模型的 converter 的维护，也接触了 PyTorch 模型到 ONNX 模型的 converter。还有人负责 TensorFlow 模型到 ONNX 的 converter。主要框架早已经搭建好了，但是在用户使用过程中会发现不少 bug，以 GitHub issue 的形式记录下来，大家认领属于自己领域的 bug 去修复，而这些 bug 绝大多数是前面的开发者遗留下来的。

关于例会

Dev manager 在美国，每周一下午（中国时间是周二早晨）开例会，有美国人（分布在东部和西部两个时区）、印度人、中国人、法国人，大家用英语，每个人简单说一下自己在干什么，而且要有 GitHub issue 或者 PR 的 link 来证明。

开发流程

- 每半年会重新调整一下工作重点。
- 在这半年中，至少会有 2 次 milestone（里程碑），用于把积累的功能发布为新版本。
- 每周写一次汇报，内容是：上周做了什么，遇到什么困难，本周打算做什么。
- 除了 dev 自己找任务外，Dev manager 也会根据每个人的 workload 分配一些任务。
- 每周例会时，先是很快地 sync 完进度，然后会有一个人（轮班制）讲他是如何解决一个问题的经历，供大家学习参考。
- 代码评审都是找与自己同时区的 dev 来做，避免跨洋交流。至少有一个人做了 code review 并 approve 后才能 check in 代码。
- 有已经搭建好的 CI Pipeline，里面有大量的 UnitTest，每次 check in 代码必须经过 pipeline 的检验，所有 test case 都通过了才能继续。

项目分析

从用户需求上看：

- 这个需求也是需要一种战略性的眼光才能发现的，弥补了微软在 AI Framework 方面的缺憾。
- 在满足了推理方面的需求后，可以向模型训练和优化方向扩展。ONNX Model Zoo 可以提供常用的神经网络模型。
- 在可以训练模型后，可以使用 ONNX Script 来作为 Python 的前端，直接手写自定义模型，这样微软就相当于有了自己的深度学习框架的前端，从而与 PyTorch/TensorFlow 媲美。

从开发流程上看：

- 这是一个维护型项目，框架已经定型，主要的任务是解决 bug，提高 user experience。
- 团队人员分布比较散，没有明确的进度要求，工作全靠自觉。
- 半年一次方向性调整是对的，这种体量和性质的项目，不能像项目 C 那样频繁操作，所以团队流程是正确的。
- 每周一次的 sync 频率也是对的，因为大家时区不同，任务的复杂度也是需要3-5天左右才能完成。
- 在初期采用瀑布式开发，搭建图 4.6.1 中所示的技术栈的各个层面。在中期采用版本升级的方式，不断地添加新的运算符（Operator）和新的平台/硬件支持。
- 每次的 milestone 大概为三个月，符合这个体量的产品的特点，因为用户也不想总跟着升级。
- CI Pipeline 保证了分散开发的灵活性与可靠性。

最终结果

- 现在这个项目还在处于发展与维护阶段，解决了用户的大量典型问题，并不断地增加新的组件，让 ONNX 更好用。

4.6.2 中大型系统的开发流程

面对中大型软件，不可能采用敏捷开发流程，

- 结构复杂，不可能在短期内有大的变动来支持新功能，这种变化大概半年发生一次。
- 功能繁杂，不可能因为新增加了一个局部功能就发布一个版本。
- 团队庞大，人员众多，需要团队间的同步合作，不可能以快节奏前进。
- 用户多为商业用户，而非个人用户，不会经常升级基本支撑软件，避免带来风险。

而且要有不同团队直接的合作，所以工作步骤与小型软件完全不同。基本步骤如下：

确定架构

如图 4.6.2 左子图所示，经过系统分析后，架构师认为应该把系统的技术栈分成以下五层：

- 交互显示层
- 缓存控制层
- 辅助功能层
- 核心功能层
- 基本数据层

并且有了相应的原型代码来验证其可行性。

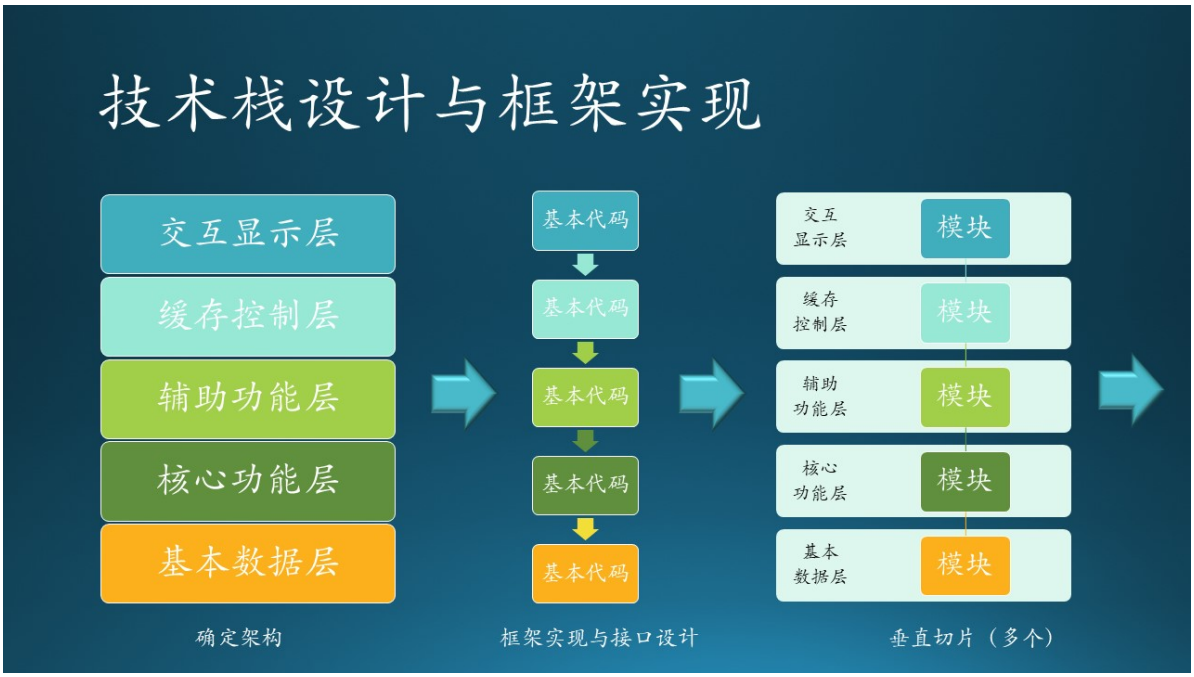


图 4.6.2 中大型系统的开发流程最佳实践

框架实现与接口设计

如图 4.6.2 中子图所示。

- 首先把各个层的基本框架代码写好，可以不做具体实现；
- 然后定义相邻两层之间的接口，可以先按照做原型程序的步骤实现一两个接口。

面向接口的设计可以实现契约式设计，每一层只需要关心对上层提供的接口和需要调用的底层接口。

垂直切片

如图 4.6.2 右子图所示。

- 选择一个功能，依次在技术栈的五个层内做具体的模块实现，并根据实际需要随时调整框架和接口的设计。
- 必要的模块实现好后，把它们从上到下串起来，应该可以形成一个完整的功能。

在做垂直切片的实现时，可以考虑使用大粒度的敏捷开发流程。

后续步骤

重复“垂直切片”的步骤，逐步增加功能。

在开发新的垂直切片功能时，需要注意以前的模块是否可以复用。如果需要修改，切记要经过充分测试，不要破坏已有的垂直切片的功能。

每一块垂直切片就相当于一块电池板，有正负极，有外壳容器，有化学反应剂等等。如果只有一块电池的话，可以提供 12V + 1A 的直流电功率。当电池逐渐多起来时，根据串并联方式，放到一个大盒子（框架）内固定，可以提供更高的电压或更大的电流，整个电池组的输出功率就提高了。

4.6.3 举例

我们仍然用计算器应用这个简单的例子来做说明，并假设它是一个非常复杂的软件，每增加一个按键、一个功能、一个显示字符都很花时间。

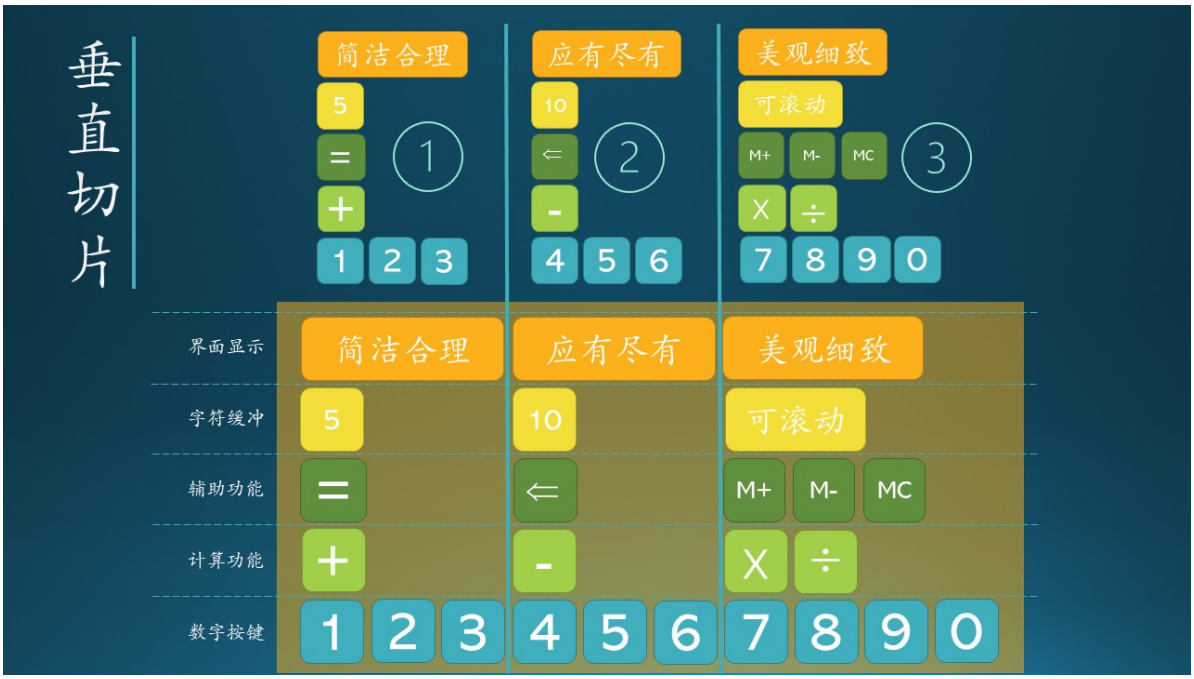


图 4.6.3 中大型系统中的垂直切片

如图 4.6.3 所示，其中：

- 上半部分圆圈内的数字 1,2,3 表示三个阶段的垂直切片的实现。
- 两条竖线严格分开了代表垂直切片的三个阶段的实现功能集。
- 下半部分暗色的矩形内的功能是最终要实现的功能全集。
- 下面的五条横向虚线代表了该系统的技术栈的五层，在表 4.6.1 中描述。

表 4.6.1 垂直切片内容表

技术栈	**第一阶段 （简易加法计算器）**	**第二阶段 （加减法计算器）**	**第三阶段 （四则运算计算器）**
界面显示	简洁合理	应有尽有	美观细致
字符缓存	只能显示5个字符	能显示10个字符	可以显示任意多字符，横向滚动
辅助功能	等号	等号、删除	等号、删除、M+、M-、MC
计算功能	加	加、减	加、减、乘、除
数字按键	123	123456	1234567890

技术栈一共有五层：

- 界面显示

即最终用户可以看到的界面，包括各个界面元素的颜色、尺寸、线宽、间距、位置等等，这些都需要 Designer 的精心设计。
举个例子，0~9 这 10 个按键在界面上应该如何排列？想当年贝尔实验室设计电话按键时，其实排在第一位的按键布局是下面这个：

```
1 2 3 4 5
6 7 8 9 0
```

但是太宽了，所以最终设计成这样：

```
1 2 3
4 5 6
7 8 9
  0
```

原因是最初人们在公共电话亭站着打电话时，“1 2 3”距离人的视线较近。但是计算器的键盘设计却都是相反的：

```
7 8 9
4 5 6
1 2 3
  0
```

原因相同：人们坐着按计算器时，“1 2 3”距离人的视线较近。

- 字符缓冲

  即可以在输出区域显示的字符数量，这涉及到输出缓冲区的设计，及界面的设计。缓冲区的大小不是什么问题，因为计算机的内存足够大。但是如果界面输出框宽度为 8 厘米，字号为 24pt 时只能有 10 个左右的字符可以显示。

- 辅助功能

  包括等号键、回退删除键、存储加键、存储减键，存储清空键。

- 计算功能

  包括“加、减、乘、除”四种操作。当然对于复杂的计算器来说，还会有指数、对数、乘方、开方等操作符，可以在后期继续增加垂直切片。

- 数字按键

  从 0 到 9 的按键输入，属于输入功能界定，并非指的这 10 个按键如何在界面上呈现。更加复杂的功能是可以输入二进制或十六进制的数字。

第一阶段：简易加法计算器

- 数字按键只支持“1,2,3”；
- 计算功能只支持“加法”；
- 辅助功能只有“等号”；
- 显示字符数量最多 5 个；
- 界面显示以“简洁合理”即可。

这个计算器可以给学龄前儿童使用，用于数学启蒙教育。

第二阶段：加减法计算器

- 增加数字按键支持“4,5,6”；
- 增加计算功能“减法”；
- 增加辅助功能“回退删除”；
- 显示字符数量最多 10 个；
- 界面显示可以做到“应有尽有”。

这个计算器可以小学一年级学生使用。如果数字键增加到 “7,8,9,0”，就可以给任何只需要用到加减法的人使用。

第三阶段：四则运算计算器

- 增加数字按键支持“7,8,9,0”；
- 增加计算功能“乘法,除法”；
- 增加辅助功能“M+,M-,MC”；
- 显示字符可以滚动；
- 界面显示“美观细致”。

至此，设计的功能全都实现了。这个计算器可以给会计、销售等行业人员使用。

好处：

1. 尽快出原型产品交付给用户并得到反馈，避免到了最后阶段才发现需求分析阶段出现的偏差。可以说是一种保险措施。
2. 每个开发周期都有明确的、可以实现的目标，做了这种任务分解后，开发进程变得清晰可控。对于负责技术栈中的各个层的软件工程师来说，每个开发周期只集中精力在一个功能上，其它时间可以用于框架优化，也有助于提高产品质量。
3. 验证各层之间的接口设计是否合理，可扩充性是否好。当然在产出第一个原型时，需要先把系统的基本框架搭建好，这需要花一定的时间。通常框架搭建的时间是构建整个产品的总时间的十分之一到五分之一，如果框架设计的好，接口制定得合理，后续的功能实现会非常的容易。

4.7 开发流程与验证工具的选择

4.7.1 采用合理的开发流程

顺序开发与敏捷开发的对比

在前面的各节中我们看到，即使是在微软，也会有多种形式的团队流程。原因多种多样，在图 4.7.1 中用表格形式比较了选择传统的顺序开发流程（瀑布模式）和敏捷开发流程的条件。

顺序开发更像是一个交响乐团，需要严密的组织、严格的排练，而敏捷开发更像一个流行乐队，有时候甚至没有谱子，在排练中找感觉，随时调整编曲中的细节。

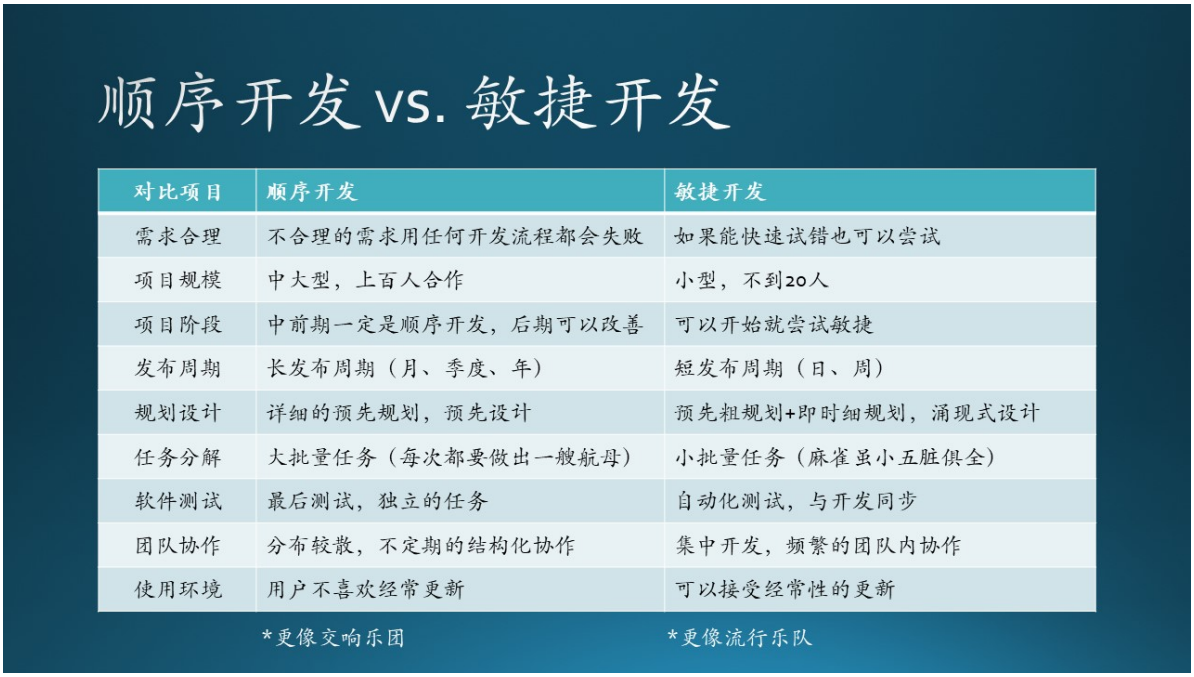


图 4.7.1 选择不同开发流程的条件

1. 需求合理性

   从 4.3.节的故事中，可以看到伪需求是在浪费大家的时间，真正的需求不是臆想出来的，真正的产品也必须是从广泛的用户需求中建模得到的。

2. 项目规模

   中大型系统很少采用敏捷方式，至少在中前期一定是顺序开发模式。

3. 项目阶段

   中大型系统到后期时可以采用敏捷模式，而小型软件从一开始就可以采用敏捷模式。

4. 发布周期

   面向企业的商业软件通常需要较长的发布周期，而面向个人用户的工具类软件可以采用较短的发布周期，当然需要生态环境（如应用商店）的支持，此时可以采用敏捷模式。

5. 规划设计

   预先规划和设计在两种模式中都需要做，只是粒度不同。前者是事无巨细，后者是适可而止。

6. 任务分解

   以制造航母为例，缺乏任何一个功能，都可能导致攻击能力或防御能力降低，此时，虽然可以做任务分解，但是不能小批量交付。而是用敏捷开发，每次交付可以制造出一只小麻雀，愤怒地去攻击猪头。
   另外，任务还有先后顺序，航母上的动力系统肯定需要先期完成，武器系统后期再安装，反过来的话无法安装。

7. 软件测试

   中大型系统的测试难点在后期的集成测试阶段。而无论是顺序开发还是敏捷开发，在前期都是要做好自动化测试机制的，以完成单元测试。

8. 团队协作

   地理位置上分布较散的团队，如果职能单一，就不适合做敏捷开发，因为局部敏捷是无效的。

9. 使用环境

   对于用户不喜欢经常更新的使用环境，比如操作系统和办公软件，就不应该使用敏捷方式，避免出现用户正在给领导做幻灯片演示时，忽然遭遇操作系统升级的尴尬场面。

中大型软件中的敏捷

那么，是不是中大型软件就不能使用敏捷开发的思想呢？不是！以必应搜索服务为例，这是一个超大型的项目，最多时有 3000 多人合作开发：

- 2010 年，从 Live Search 改名为 Bing 后，发布周期是 6 个月；
- 3 年后，缩减到 3 个月整体发布一次；
- 再过两年，缩短到 1 个月整体发布一次；
- 质的变化发生在 2016 年，不再整体发布了，而是每个 feature team 可以自主发布，最短周期可以是星期。
- 到了 2018 年，随着系统的成熟，完全取消了发布限制，每个 feature team 可以随时发布新功能。

图 4.7.2 必应搜索服务发布周期的演进

所以，中大型软件想要做到敏捷并非不可能，而是需要一些前提条件。

1. 架构合理

   通过理解布鲁克法则和康威定律，在项目初期就要设计出合理的架构，非常鲁棒地将后续的工作彻底拆分成若干个小项目，使之可以并行工作。
   敏捷和万马奔腾是两回事儿。

2. 团队成熟

   团队具备专业的管理人员来驱动整个项目的开发进程，而普通团队成员的素质较高，接受过相关的培训。
   敏捷和血气方刚是两回事儿。

3. 文档完善

   人多，人员流动大，沟通成本高，所以一些关键性的文档一定要完善，便于所有人能明白上下文。
   敏捷和心有灵犀是两回事儿。

4. 时机恰当

   一开始就敏捷，肯定是行不通的。打好基础后，再实行敏捷。
   敏捷和盲目冒进是两回事儿。

5. 流程完备

比如代码审查、单元测试，这些基本的流程可以保证软件质量，在此基础上敏捷才有意义。
敏捷和偷工减料是两回事儿。

6. 工具先进

   好的工具可以帮助团队降低管理的复杂性，减少错误的发生频率。
   敏捷和吃苦耐劳是两回事儿。

4.7.2 采用合适的验证工具

在开发复杂的软件系统时，尤其是面对一些前所未有的功能时，没有前人的经验可以借鉴，并且没有人可以确定它能做还是不能做。这时就需要一些工具（方法）来帮助人们做快速的验证。经常用到的是：

- 概念验证（PoC，Proof of Concept）
- 原型开发（Prototyping）
- 最小可行产品（MVP，Minimum Viable Product）

其基本概念如图 4.7.3 所示。

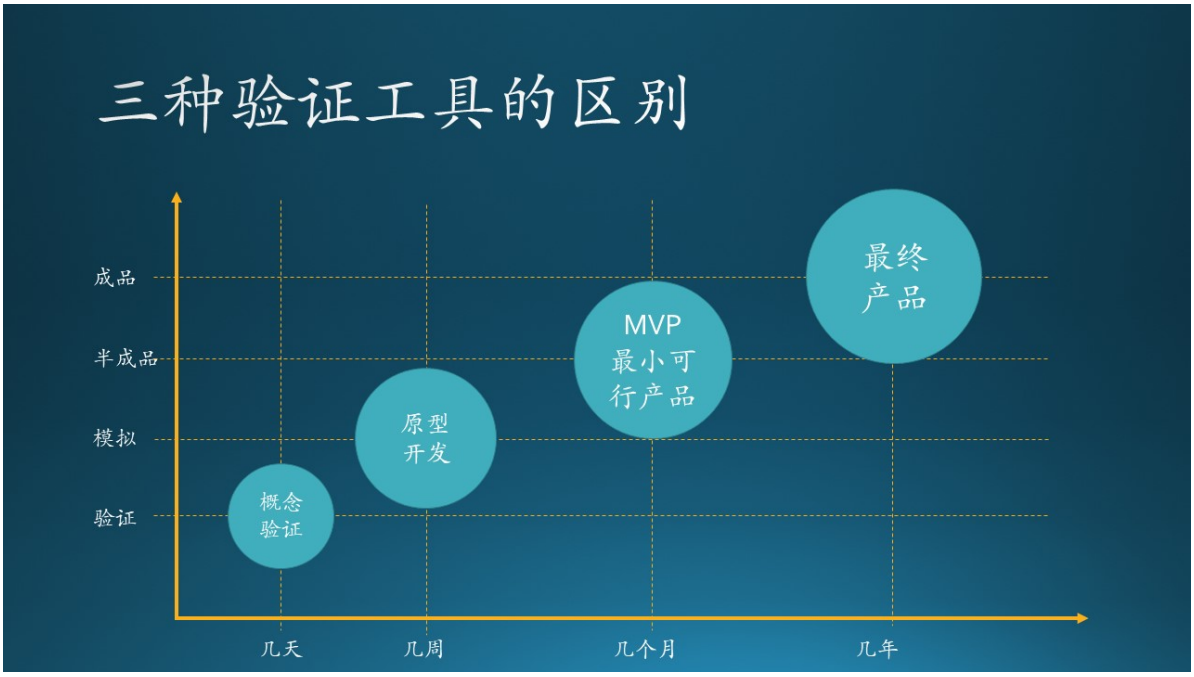


图 4.7.3 三种验证工具的示意图

表 4.7.1 三种验证工具的比较

比较项	概念验证	原型开发	最小可行产品	最终产品
生命周期	极短（日）	短（周）	中（月）	长（年）
成本	极低	较低	中等	最大
形式	简陋的几行代码	代码可沿用到产品	必须是可发布的代码	产品代码
发布	不发布，仅内部使用	定向发布后回收	阶段性发布	公开发布
作用	验证技术难点	确认用户需求	及时得到反馈	获得最大收益

比较项	概念验证	原型开发	最小可行产品	最终产品
要控制	风险	风险	风险	等各种风险

PoC（Proof of Concept）概念验证

目标：验证某个想法是否在技术上是可行的。

PoC 的主要目标是测试这个想法在技术上是否可行，是否值得追求。这不是为了发现用户需要什么，或者是否有解决方案的需求，而是从技术角度来看，是否有可能并且值得追求这个想法。

毕竟，你不想花费大量的时间在一个比你最初期望的要难10倍（而且更昂贵）实现的想法上，所以应该是一次快速的尝试，以开发风险最大的技术假设的测试版，或者只是对现有技术可能性的更深入分析。

比如，想实现的新功能强烈依赖的核心技术是 OCR，而自己临时做一个 OCR 是不可能的，那就需要：

1. 找到现有的商业化 OCR 解决方案去做试验；
2. 多找几个提供商，看看哪一家的识别率更好，价格更低；
3. 有条件的话后期自己做一个，以便降低费用。

概念证明通常不与最终用户共享，而是用作内部工具。

特点：

- 极短的生命周期；
- 极低的成本；
- 为后面的原型、MVP 提供参考；
- 内部使用，有时候也可以分享给甲方；
- 帮助估算后期的开发时间；
- 降低尝试去完成一个不可能完成的任务的风险。

Prototyping 原型开发

目标：以最低成本收集早期反馈。

原型开发方法试图向我们及客户展示产品开发好的样子。以汽车制造举例，PoC 更多地关注“引擎盖下”（用户看不见）部分，但原型设计更多地关注汽车的外观及内饰，即软件的 UX/UI 部分。它的目标是建立一个简单的、实验性的想法模型，以便在实际的产品开发之前测试和验证概念。

几乎任何东西都可以是原型：纸上的线框、数字演示、模仿网站的 PowerPoint 演示等等。尽早获得反馈和发现想法中的漏洞的能力可能是产品成功的决定因素。

有些原型开发也是需要代码的，这些代码可以是粗制乱造的，只要能完成任务即可，当然，也有些代码是可以直接用于最终产品的。

特点：

- 较短的生命周期；
- 低成本；
- 集中在朴素的想法实现和界面/交互部分；
- 可以对整个产品做原型，也可以只针对其中的一部分做原型；
- 分享给甲方，可以尽早得到反馈；
- 降低与预期需求不一致的风险。

MVP（Minimum Viable Product）最小可行产品

目标：尽快、廉价地推出产品，验证假设，收集用户反馈，并迭代。

MVP应用程序开发是关于构建一个产品版本，该版本将在市场上提供，真实用户可以与之交互。

其核心思想是在向最终用户提供价值主张的同时，尽可能少地包含功能，并满足尽可能多的市场需求。

它是关于构建产品的连续循环，然后测量和测试您的假设和猜想是否正确，收集用户的反馈，从中学习并改进产品，然后一次又一次重复。

专注于MVP开发而不是一些成熟的解决方案可以降低交付不需要的产品的风险，并允许您根据市场反应以低成本进行转向。

在真实市场上测试产品总是至关重要的，即使你已经验证了你的原型，主要是因为人们通常不知道他们想要什么，他们只是认为他们知道自己想要什么。

特点：

- 正常的功能开发生命周期；
- 中等成本，大于原型的成本；
- 可以发布到市场面向用户；
- 验证一些局部功能而无需构建整个产品。

MVP 最小可行产品与 4.6 节中讲的“垂直切片”的概念不完全相同：

- 垂直切片，要求**从系统角度看**，为了使得一个功能从上到下可以跑通，而在系统架构的各层提供必要的模块支持。
- MVP 最小可行产品，要求**从用户角度看**，为了使得一个功能令用户满意达到完美标准，而在正确性、可靠性、可用性三个质量维度做出的努力。

所以，最小可行产品是垂直切片的延申。

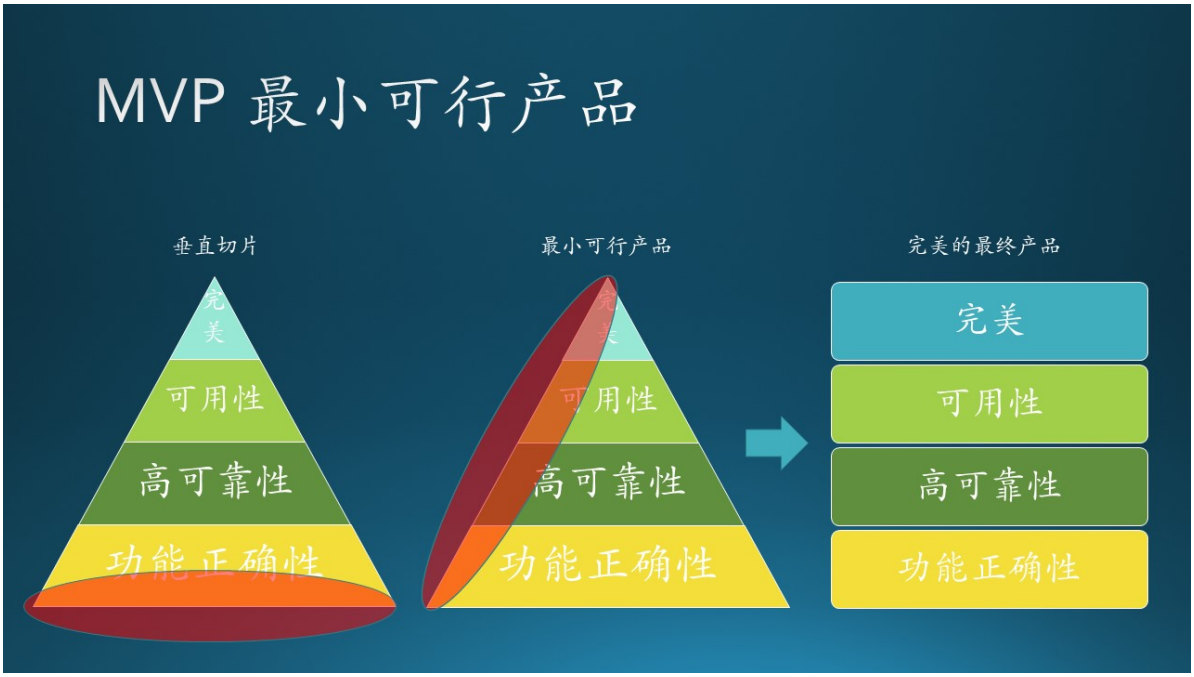


图 4.7.4 MVP 最小可行产品示意图

在图 4.7.4 中，用金字塔形状列出了 4 层，而其内容分别是：

1. 功能正确性（Functional）
2. 高可靠性（Reliable）

3. 可用性（Usable）
4. 完美（Delightful），用户满意

为什么列出了这四个层次呢?

因为 1,2,3 三个层次都是软件质量的评判标准，虽然在软件工程中还有其它一些质量特性，但是对于最终用户来说，只有这三点是可以明显感受到的。关于质量特性我们在第 6 部分再详细介绍。

首先要求功能正确性，并不是说一次性要提供 10 个功能，而是只提供 2 个功能时，要保证这个 2 个功能是完全可以工作的，从输入、计算到输出，都是符合预期的。

其次是高可靠性，即不分时间地点的多次输入，都可以得到预期的输出。

再次是可用性，即界面友好，用户基本上不需要经过特殊培训就可以使用。用户可以方便地输入，比如当打字不方便时可以用语音输入。并且可以得到明确可见的输出。

前三者都具备的情况下，就可以是令用户满意的完美软件。

垂直切片，如图 4.7.4 中的左子图所示，它强调功能的正确性。

最小可行产品，如图 4.7.4 中的中子图所示，在每个层次上都选取一部分进行打磨雕琢，然后发放给用户，而每个层次上的重要性是根据金字塔的形状由大而小的，可以解释为：

- 一定保证功能正确性；
- 有选择地保证其中一部分功能的可靠性；
- 再选择一两个关键的点来打磨其可用性；
- 看看用户在这一两个点上的用户体验是否满意。

而图 4.7.4 的右子图展示了最终的完美产品，它应该在软件质量的各个特性上都是平衡的，而非金字塔形。

从一定程度上来说，最小可行产品的概念更像是 4.4 节中的小型软件开发流程，所以，从中也可以看到，中大型软件和小型软件的开发流程其实是互通的，可以互相借鉴，或者是最终都可以通过系统分解而归结到小型软件上。

4.8 练习 - 代码管理与流程管理工具

4.8.1 团队开发流程代码工具

目前 Git 是绝对的代码管理工具首选。刚刚加入团队的开发人员需要对其有基本的了解。

基本概念：

- Remote 远程仓库
  就是类似 github，码云等网站所提供的仓库，可以理解为远程数据交换的仓库。
- Repository 本地仓库
  你执行 git clone 地址，就是把远程仓库克隆到本地仓库。它是一个存放在本地的版本库，其中 HEAD 指向最新放入仓库的版本。当你执行git commit，文件改动就到本地仓库来了。
- Index/Stage 暂存区
  一般存放在 .git 目录下，即 .git/index，它又叫待提交更新区，用于临时存放你未提交的改动。比如，你执行 git add，这些改动就添加到这个区域。
- Workspace 工作区
  你电脑本地看到的文件和目录，在 Git 的版本控制下，构成了工作区。

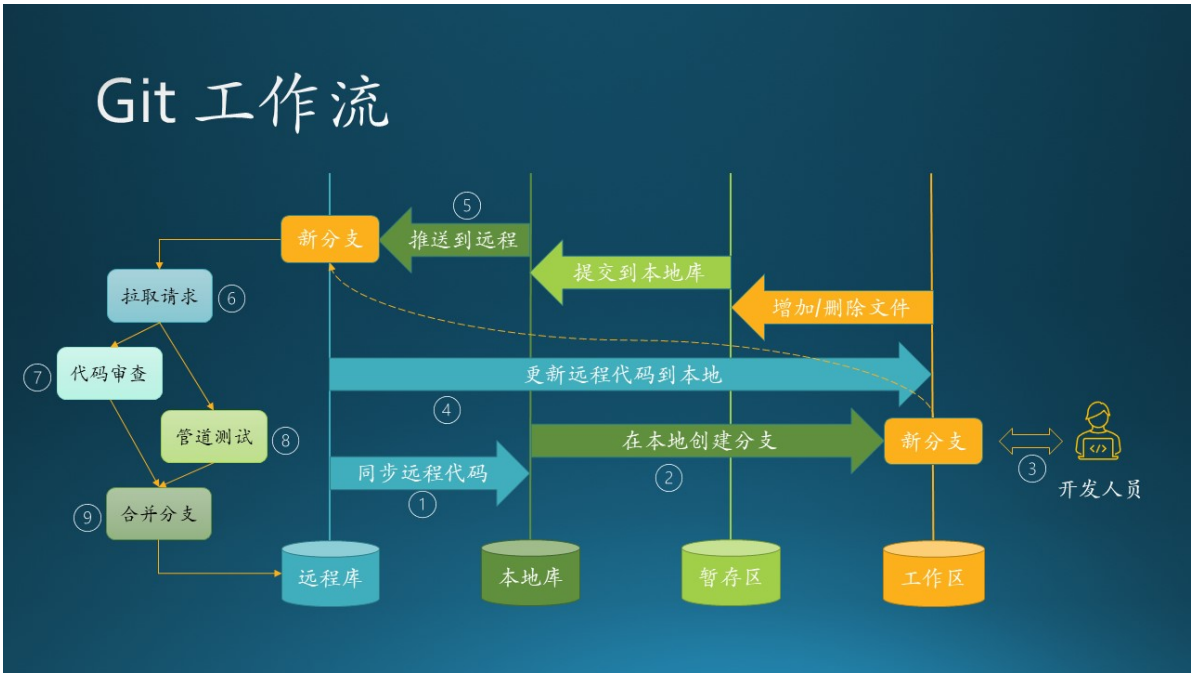


图 4.8.1 Git 工作流

Git 的使用流程如下：

1. 开发人员将源代码从远程存储库签出到本地存储库；
2. 创建一个新分支；
    - 注意每次一个新的任务开始前，都要创建新分支，避免使用旧分支或者与其它不同任务的代码混在一起，便于管理。
3. 开发：在本地编写、运行和调试你的代码；
    - 在本地搭建一个 OneBox 可以跑通系统的基本流程。如果不得不依赖远程运行环境，将会严重影响你的开发、调试进度。
    - 你的代码应该包括对新写的代码的单元测试。
4. 如果在开发期间，有团队内别的开发者提交了代码，开发人员需要从远程更新代码，避免将来的冲突；
    - 如果有很多人同时开发，这种情况将会经常发生，甚至会影响你在本地修改的代码逻辑。
5. 提交代码本地库保存，然后将其推送到远程存储库；
    - 此时的代码对他人没有任何影响。
    - 可以在另外一台计算机上下载代码并继续完善。
6. 创建一个 PR（Pull Request，拉取请求）以将分支合并到目标分支中；
    - 在一个成熟的团队开发环境中，CI Pipeline 和代码审查机制将会被自动触发。
7. 拉取请求由多个审阅者审阅，可以设置为至少有一个审阅者批准后方可；
    - 更严格的设置可以是至少两个人批准。
    - 如果有反馈意见，你需要解释或在本地解决这些质疑，再次推送到远程，此时不需要建立新的 PR，使用原有的 PR 即可。
8. 自动启动测试管道（CI Pipeline）验证该分支；
    - 如果你的代码不能通过自动单元测试，代码审查者将不会给与批准，因为有可能会 break 现有的逻辑。
9. 如果测试通过并且得到批准，可以合并分支到主分支，完成拉取请求。

在第 6 步中，明明是要把修改的代码合并入主分支（Push），为什么叫拉取请求呢（Pull）？这是相对于主分支说的，是在请求主分支把修改拉进去。

如果是在 Windows/Mac 环境下使用，建议下载 GitHub Desktop 版客户端软件，不需要记忆复杂的 Git 命令，使用鼠标即可轻松完成上述任务。

4.8.2 团队开发流程管理工具

在微软一般使用 Azure DevOps 来做开发流程管理工具，只需要用浏览器访问，所有服务都部署在云端。利用这个工具，开发团队可以方便地把本章中所学到的关于开发流程的知识应用到实践中。

主要步骤如下：

1. 选择流程类型

先登录 Azure DevOps 网站，建立团队和项目。

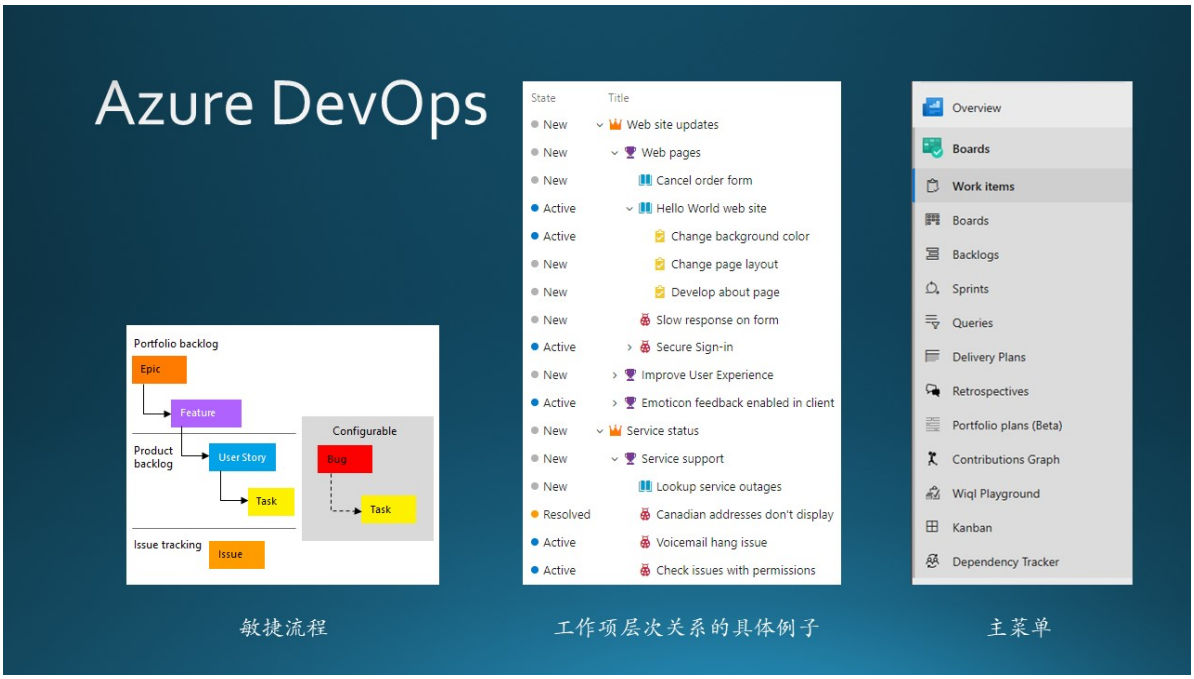


图 4.8.2 项目流程类型与主菜单

一共有四种类型可选：敏捷流程、基本流程、Scrum 流程、CMMI 流程。如果选择“敏捷流程”，则 Backlog 中的层次关系将如图 4.8.2 的左子图所示，具体的例子如中子图所示，而右子图展示了 Azure DevOps 的左边栏的主菜单，我们只关心 Boards 下面的 Boards, Backlogs, Sprints。

2. Backlogs - 创建产品计划

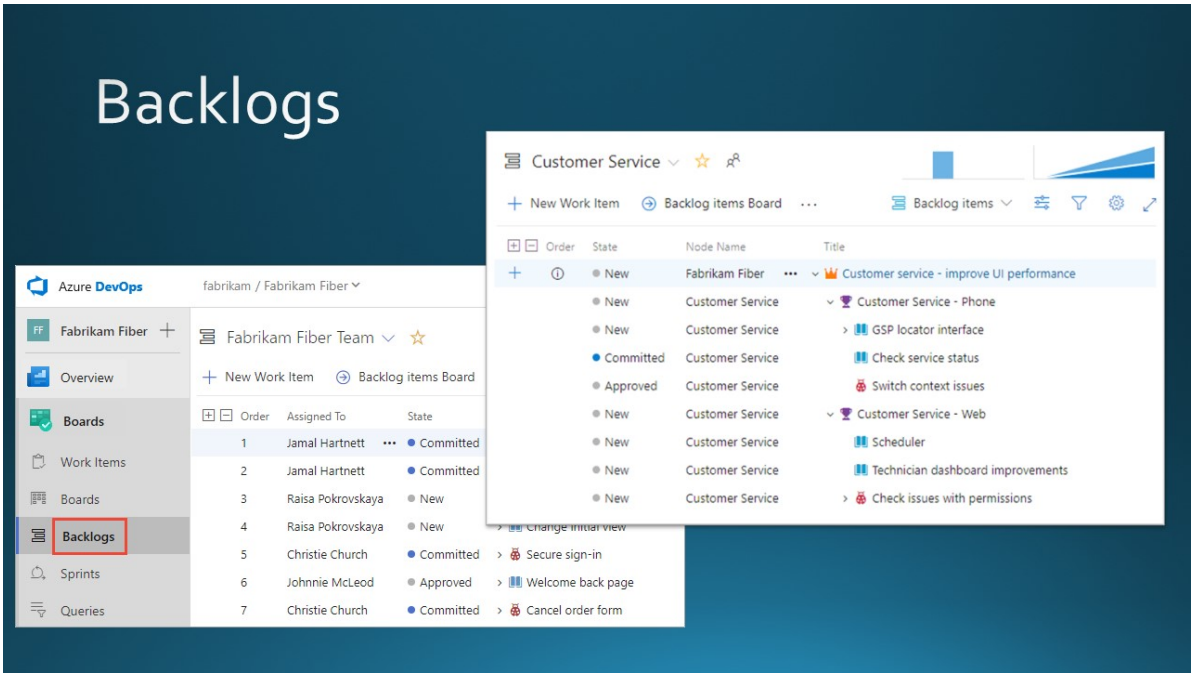


图 4.8.3 待办事项 Backlogs

在 Backlogs 里面，都可以添加以下四种类型的工作项：

- 史诗 Epic
- 特性 Feature
- 用户故事 User Story
- 代码缺陷 Bug

这与图 4.8.2 中左子图的层次关系是对应的。

首先创建 Epic，但是在一个比较小的项目中，也可以没有 Epic，因为整个项目都只为一个 Epic 服务。

其次，创建特性 Feature，有些资料中也称之为功能。特性一般对应系统中的一个子系统。

然后在特性下面创建用户故事 User Story，就是常说的 Scenario（用户场景）。

3. Boards - 添加具体工作内容并管理

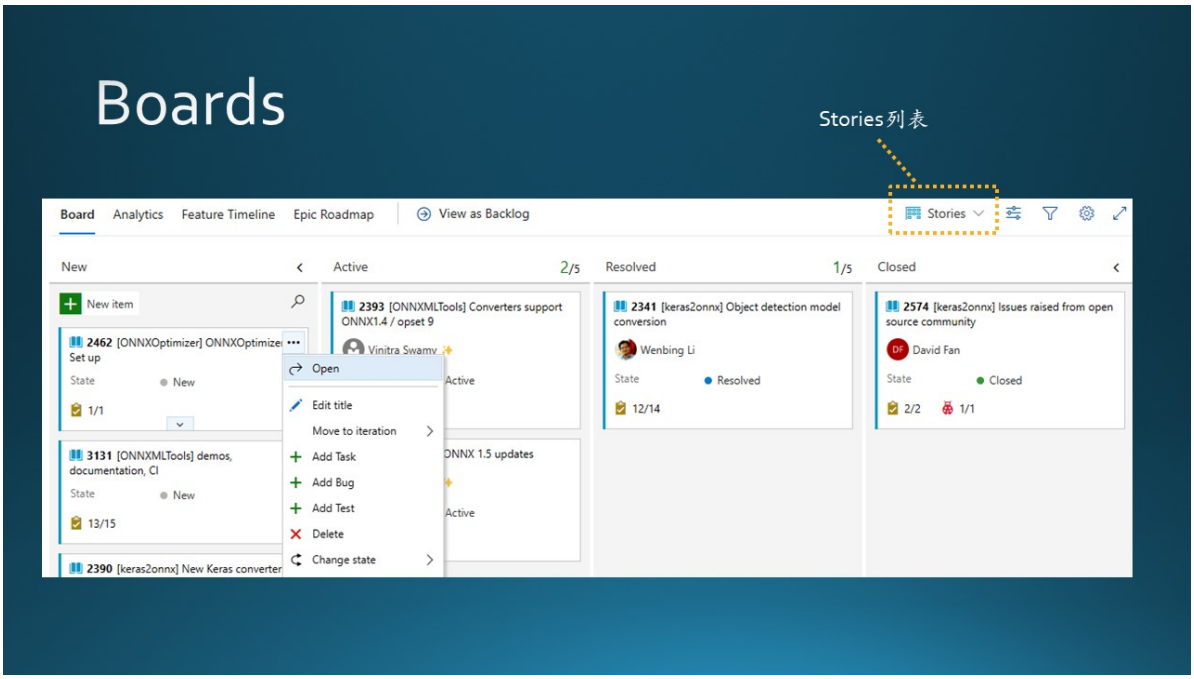


图 4.8.4 工作版内容管理 Boards

Backlogs 和 Boards 是可以相互切换显示的，内容一致，只是显示方式不同。Boards 的显示方式如图 4.8.4 所示。

切换到 Stories 列表（相当于最底层的工作项），就可以为每个 User Story 添加具体的任务（Task）、缺陷（Bug）、测试（Test），然后就可以给每个 Task、Bug、Test 分配 owner 了。在 User Story 及以上级别的项中，一般不分配 owner。

Board 中有四列分栏，分别是 New、Active、Resolved、Closed，状态变化后用鼠标把工作项拖到相应的列即可。

4. Sprints - 制定冲刺计划

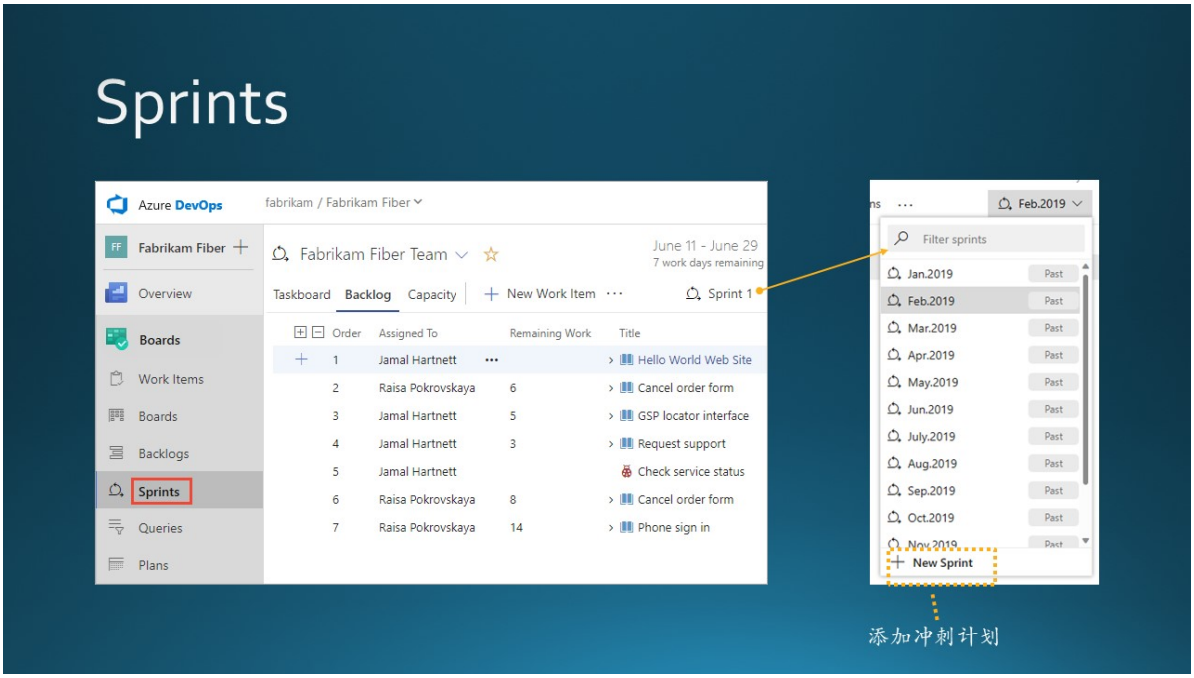


图 4.8.5 冲刺计划 Sprints

在添加好具体工作内容后，点击 Sprints，就可以开始制定冲刺计划了。最开始时没有任何记录，点击【+New Sprint】按钮添加新计划，然后把工作项赋给每个 Sprint。

5. 其它

- Work items - 工作项
  浏览并可以根据条件快速查找工作项。
- Queries - 查询
  用于定义一组筛选条件来列出工作项，便于与他人共享、执行批量更新或导入/导出操作。
- Delivery Plans - 交付计划
  用于查看团队计划交付的故事或功能的计划。
- Dashboards - 仪表板
  在主菜单的 Overview 下面。

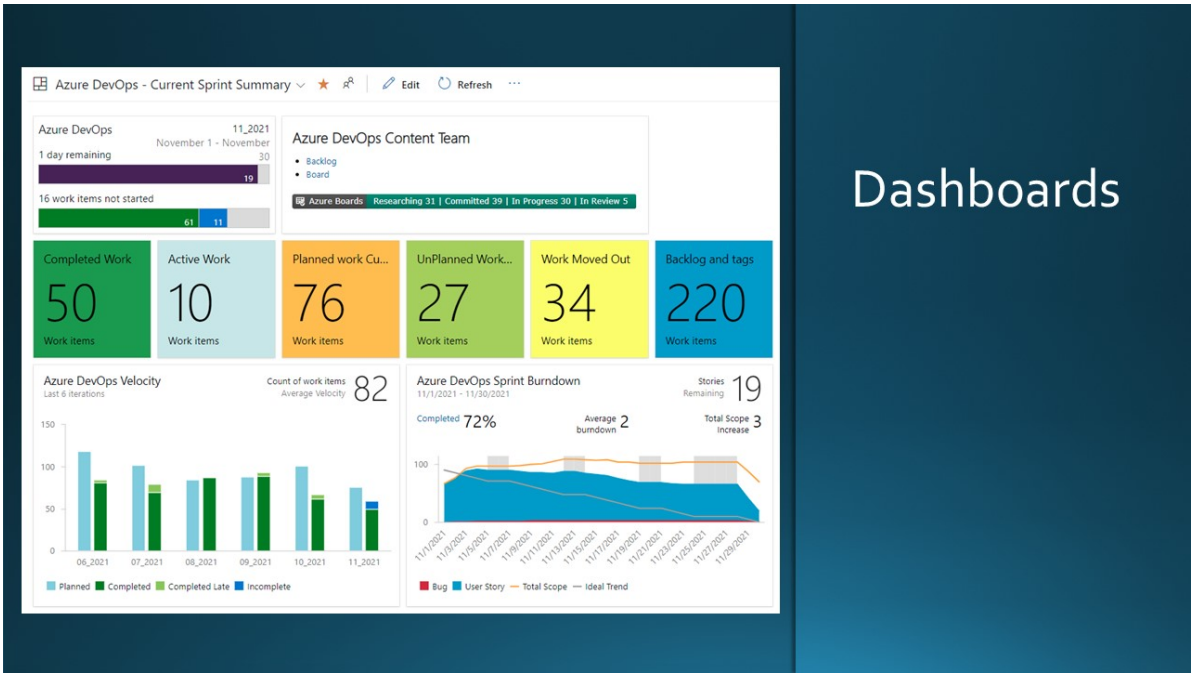


图 4.8.6 仪表板 Dashboards

- Burndown - 燃尽图
  关于燃尽图，几乎所有的资料中都有提及并且建议使用，但是笔者在实际工作中的体会是：这个东西没必要存在。原因如下：

  1. 能够绘制出燃尽图的条件是：每个 task 都需要给出估算时间，这种估算即使是按天估算也是不准确的。而且每个开发人员需要每天更新当前进度，一忙起来就忘记了。
  2. 一个 5-10 人的团队，如果是两周的 Sprint，实际的 task 也没多少，每个人最多两三个。这种短期且粗粒度的统计，没有必要进行。如果具体实践起来，你会发现那个燃尽图根本就不是你想看到的平滑向下，而是有很多断崖。
  3. 如果想显示长期的燃尽图，就需要长期的计划任务及估算，这和敏捷的思想是有矛盾的。
  4. 这东西是给领导看的，对于敏捷团队内部没有多大的指导意义。