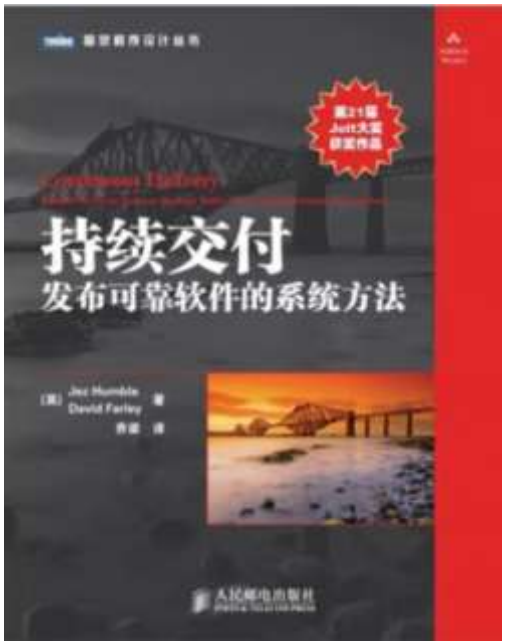
L8 持续交付

发布可靠软件的系统化方法

参考书目

- 持续交付 发布可靠软件的系统方法，Jez Humble, David Farley著，乔梁译，人民邮电出版社，2011
- 持续交付2.0 业务引领的DevOps精要，乔梁著，人民邮电出版社，2019



- 每个追求卓越的科技公司都希望能够随时随地发布，而无需工程师在晚上或者周末进行部署。
- 能够快速、频繁且安全地发布软件，并实现小批量交付，意味着我们可以快速获得对我们的想法的反馈。
- 我们可以构建原型并使用真实用户对其进行测试，从而避免开发那些对用户没有任何价值的功能。

——Jez Humble
《持续交付2.0》序一

持续交付

- 持续交付的核心是**部署流水线**。
- 持续交付的最终目标是，它使团队能够通过一个**完全自动化**的过程，在任意环境上部署和发布软件的任意版本。

大纲

- **软件交付的问题**
  - 一些常见的反模式
  - 反模式的解决
  - 软件交付的原则
- 软件配置管理
- 持续集成
- 安全开发过程
- 持续交付：部署流水线及脚本化
- 应用程序的部署与发布

发布反模式

- 在许多软件项目中，为什么软件发布的当天是最紧张的一天?
  - 软件发布是一个需要**很多手工操作**的过程。
  - 对于一个复杂的发布过程，手工发布需要**多个复杂的环节**，其中只要有一步没有完美地执行，就会导致失败。
  - 手工发布的过程即便是提前进行测试过，也**无法保证再次手工执行的正确性**。

- 反模式
  - 屡见不鲜的糟糕实践

发布反模式（1）

- 手工部署软件
  - 有一份非常详尽的文档，该文档描述了执行步骤及每个步骤中易出错的地方。
  - 以**手工测试**来确认该应用程序是否运行正确。
  - 在发布时常常会**修正**一些在发布过程中发现的问题。
  - 如果是集群环境部署，常常发现在集群中各环境的配置都不相同，比如应用服务器的连接池设置不同或文件系统有不同的目录结构等。
  - 发布过程需要较长的时间（超过几分钟）。
  - 发布结果不可预测，常常不得不回滚或遇到不可预见的问题。
    - 发布之后凌晨两点还绞尽脑汁想着怎么让刚刚部署的应用正常工作

发布反模式（1）

- 手工部署导致了：
  - 部署过程依赖于部署专家
    - 尽管这是个枯燥和重复的过程，但仍需要相当程度的专业知识。
  - 部署过程即不可重复，也不可靠。
  - 依赖于部署文档，而这个文档往往是不完整，未更新，并且维护困难。

- 自动化部署应该是软件部署的唯一方式。

发布反模式（2）

- 开发完成之后才向**类生产环境**部署
  - 测试人员在开发机器上对软件进行测试
  - 运维人员在第一次部署到类生产环境中才接触该软件系统
  - 开发团队与部署团队只是通过最后提交的部署文档来沟通，开发团队与部署、运维团队之间的协作比较少。
  - 在类生产环境（试运行环境）中发现的问题往往被仓促修复以便维持原有交付期限
    - 这些可能成为由项目经理保存的“已知缺陷”，但往往在下一次发布时这些缺陷的优先级还是被排得非常低

- 需要在项目的早期就尽可能将开发、测试、部署和运维团队纳入开发过程。

发布反模式（3）

- 生产环境中使用手工配置管理
  - 如果需要修改配置，比如服务器线程池中的线程数，就登录到生产服务器上手工修改。
  - 升级都由人工修改，包括补丁，并且未做记录。
  - 结果可能是：系统无法回滚到之前部署的某个配置。

- 不应该允许手工修改测试环境，试运行环境和生产环境的配置
  - 系统中的任何配置都应该通过一个自动化的过程进行版本控制。

反模式的解决

- 高度自动化的部署
- 全面的配置管理

自动化部署的威力

曾经有个客户，他们在过去每次发布时都会组建一个较大的专职团队。大家在一起工作七天（包括周末的两天）才能把应用程序部署到生产环境中。他们的发布成功率很低，要么是发现了错误，要么是在发布当天需要高度干预，且常常要在接下来的几天里修复在发布过程中引入的问题或者是配置新软件时导致的人为问题。

我们帮助客户实现了一个完善的自动构建、部署、测试和发布系统。为了让这个系统能够良好运行下去，我们还帮助他们采用了一些必要的开发实践和技术。我们看到的最后一次发布，只花了七秒钟就将应用程序部署到了生产环境中。根本没有人意识到发生了什么，只是感觉突然间多了一些新功能。假如部署失败了，无论是什么原因，我们都可以在同样短的时间里回滚。

反模式的解决

- 自动化
  - 如果构建、部署、测试和发布流程不是自动化的，那它就是不可重复的。
- 频繁做
  - 如果能够做到频繁发布，每个版本之间的差异会比较小，这会大大减少与发布相关的风险。

反模式的解决

- 修改必须触发反馈流程
  - 反馈流程是建立以自动化的方式测试每一项变更的流程。
- 快速反馈的关键是自动化，其中测试自动化是关键。
- 测试用例分为**提交阶段的测试**和**提交阶段之后的测试**两类。这两个阶段对自动化测试用例的要求有所区别

反模式的解决

- 提交阶段的测试
  - 速度快
  - 高覆盖率(75%以上)
  - 测试失败直接触发发布失败
    - 也就是说如果测试失败，则表明有严重问题，无论如何也不能发布。因此检查界面元素颜色是否正确这类测试不应该在这个测试集合中
  - 大部分的测试用例最好做到环境中立，不需要与生产环境一模一样(比如Mock短信发送接口)

反模式的解决

- 提交阶段之后的测试
  - 可以速度慢些
  - 失败后也可能可以继续发布
  - 测试运行环境需要与生产环境相同

软件交付的原则

- 为软件发布创建一个可重复的、可靠的过程
- 将所有的事情自动化
- 把所有的东西都纳入版本控制
- 提前做麻烦的事情
- 只有发布完成了才能算是完成了
- 把交付的压力分担到团队每个人
- 持续改进

大纲

- 软件交付的问题
- **软件配置管理**
- 持续集成
- 安全开发过程
- 持续交付：部署流水线及脚本化
- 应用程序的部署与发布

配置管理

- 配置管理基础概念
- 依赖管理
- 软件配置的类别及管理
  - 部署的配置
  - 启动或运行时的配置
- 环境管理
- 数据管理

配置管理

- 什么是配置管理
  - 配置管理是一个过程，通过该过程，所有与项目相关的产物以及它们之间的关系都被唯一定义、修改和检索。
- 选择版本管理工具仅仅是指定配置管理策略的第一步
  - 你能否完全再现你所需要的任何环境（操作系统，含版本及补丁，以及部署在其上的软件应用及其配置）
  - 你是否能够很容易地看到已被部署到某个具体环境中的某次修改，查看谁、何时、如何进行修改的。
  - 团队成员是否能很容易得到所需的信息

使用版本控制

- 对所有内容进行版本控制
  - 版本控制不是仅仅是源代码控制，还包括测试代码、数据库脚本、文档、库文件、配置文件等。

将所有东西都提交到版本控制库中

许多年前，本书作者之一参与了某个项目开发相关的工作，该项目由三个子系统组成，分别由位于三个不同地点的三支团队开发。每个子系统都使用IBM MQSeries基于某种专用消息协议相互通信。这是在使用持续集成之前，预防配置管理问题的一种手段。

我们对源代码的版本控制一直都非常严格，因为我们在该项目之前就得到过教训。然而，我们的版本控制也仅仅做到了源代码的版本控制而已。

当临近项目的第一个版本发布时间点时，我们要将这三个独立的子系统集成在一起。可是，我们发现其中某个团队使用的消息协议规范与其他两个团队使用的不一致。事实上，该团队所用的实现文档是六个月前的一个版本。结果，为了修复这个问题且保证这个项目不拖期，在之后的很多天里，我们不得不加班到深夜。

假如当初我们把这些文档签入版本控制系统中，这个问题就不会发生，也就不用加班了！假如我们使用了持续集成，项目工期也会大大提前。

使用版本控制

- 对所有内容进行版本控制
  - 可以将编译器、库以及其他相关的二进制镜像也放在版本控制库中。
    - 在有了maven、npm这类工具后，对这些二进制文件的大部分依赖可以变成文本的声明。
  - “删除”的自由
    - 可以随时删除不必要的文件，因为可以随时恢复

使用版本控制

- 频繁提交
  - 使用版本控制时，必须频繁提交代码
- 频繁检出
  - 团队所有成员都尽可能使用最新的版本
- 使用意义明显的提交注释
  - 增加的功能名
  - 修改的bugID
  - 提交本身应该具有良好的原子性（唯一目的）

依赖管理

- 外部库文件的依赖
  - 外部库文件通常以二进制形式存在，也需要纳入配置管理。
  - 使用maven等构建工具可以**在脚本中指定依赖**，但增加了初次build的时候下载的时间。
    - 组织可以通过建立镜像来缓解下载的时间开销
- 内部组件的依赖
  - 一个大的项目会分解为不同的组件，组件之间的依赖建议设计为二进制的依赖
  - 可以把被依赖的组件的二进制文件也放到maven之类构建工具能够识别的库中。

软件配置的类别及管理

- 配置通常独立于源代码，可以在部署的阶段设置，为软件增加一层灵活性。
- 配置的灵活性的代价
  - 使用程序设计语言开发的组件有多种保证质量的方法，比如语法检查，自动审查，甚至自动化的测试。但**配置大多是文本，缺乏语法和语义上的自动检查**，并且往往与环境紧密相关，不会被纳入自动化测试。
  - 如果配置工作变得非常复杂，可能会抵消其灵活性上带来的好处

终极可配置性的危险

我们曾经有个客户，花了三年的时间与一个供应商合作，想在其业务领域使用该供应商提供的软件产品。该产品被设计成具有高灵活性和高可配置性的软件，以便满足客户的需求。然而，最终的结果是只有该产品的产品专家才知道如何配置。

然而，客户担心该系统一时还无法用于生产环境。最后，他找到了我们，而我们的组织花费了八个月的时间，从零开始用Java为其定制了一个满足同样需求的软件。

软件配置的类别及管理

- 配置的分类
  - 构建脚本中的配置
  - 打包脚本中的配置
  - 部署安装时的配置
  - 启动或运行时的配置

软件配置的类别及管理

- 识别出配置信息生效是在软件生命周期的哪个阶段。
- 配置文件与源代码同样管理，但配置文件中的值往往需要设置为变量，通过自动化工具设置。
  - 通过自动化工具，根据不同的环境，将配置文件中配置项进行设置。
- 确保测试可以覆盖到部署或安装时的配置

部署的配置

- 在上述类别中，部署时对软件的配置非常重要，可以在这个阶段定义：
  - 数据库
  - 邮件服务器
  - 消息服务器
  - 线程数
  - ……

启动或运行时的配置

- 应用程序需要以某种方式解释和获取定义在外部的配置信息，包括：
  - 如何描述配置信息
  - 如何获取配置信息
  - 如何管理配置信息的版本

启动或运行时的配置

- 配置的格式
  - 描述配置信息的方法有多种，比如XML文件，java的property文件；对于层次化的配置，可以使用YAML格式的文件。
- 读取配置信息
  - 建议程序中读取配置信息时，不要直接绑定具体的存储形式和格式，而是使用一个Façade接口，这样在测试的时候可以作比较容易Mock
- 为配置项建模
  - 不同的配置通常是为了不同的版本，产生不同版本的原因包括环境、发布功能集等。

环境管理

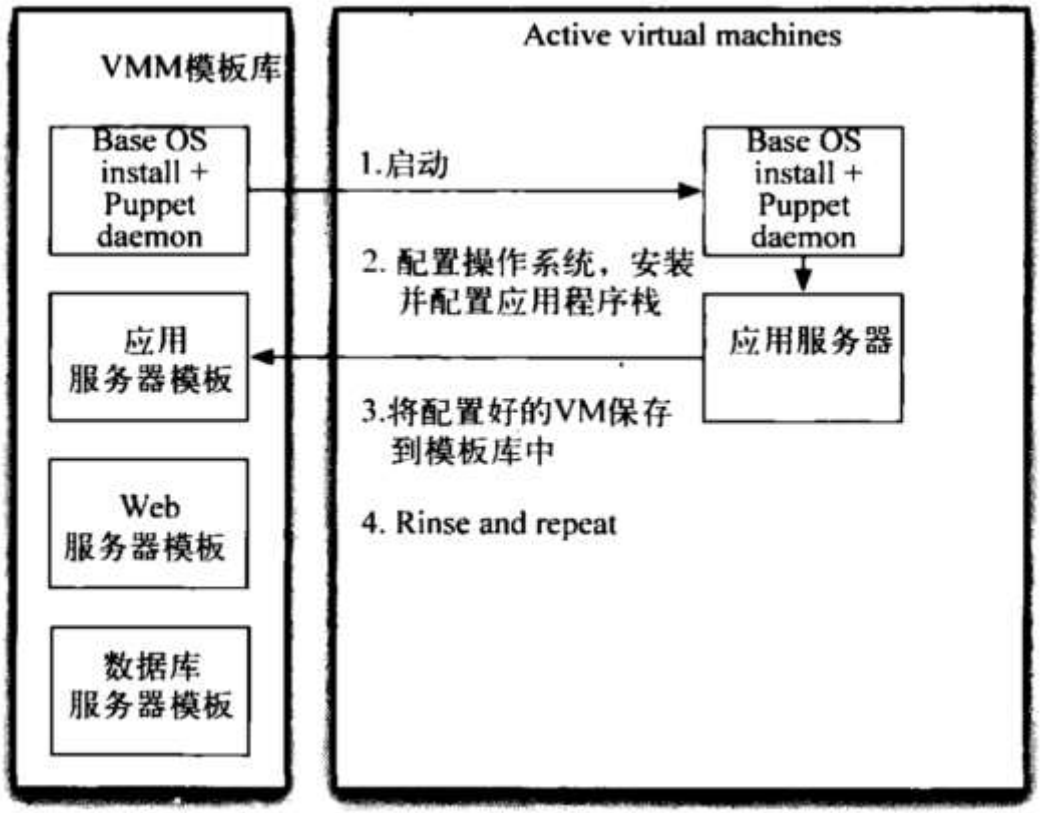
- 环境是指应用程序依赖的基础设施，包括操作系统、数据库等。
- 环境的配置与应用程序的配置同样重要。
- 对环境——特别是生产系统环境——的**变更**过程进行严格的管理控制。
- 以Docker代表的虚拟化技术为自动化环境管理提供了强有力的工具支持。

环境管理：虚拟化技术

- 虚拟化是一种在一个或多个计算机资源上增加了一个抽象层的技术。
  - 通过虚拟化技术，我们可以实现一个受控的，完全可重复的部署和发布流程。
- 虚拟化技术可以通用一个简单的机制来创建系统所需的环境基线
  - 对于Docker，这个**对环境基线的描述**是一个可以很容易放入配置管理系统的**脚本**。
- 使用虚拟化技术使得制作类生产系统的副本变得非常容易

环境管理：虚拟化技术

- 利用虚拟化技术，一个系统的实例可以通过一组预先定义好的模板库构建出来，这样，每次构建得到的实例是高度一致的：



数据管理

- 新版本发布时，应用程序可以完全删除之前的版本，然后用新版本代替现有版本。
- 但是**数据无法以同样的方式处理**
- 对新版本的发布而言，数据通常需要**伴随新版本的升级而变化**，包括：
  - 测试数据的调整
  - 应用数据的保存和迁移
- 数据的迁移是部署的一个重要环节，同时还是一个巨大的挑战。

数据管理

- 数据库的脚本化
  - 任何对数据库的修改都应该通过自动化的过程来修改，避免直接手动修改数据库的任何结构。
  - 相关脚本都必须纳入配置管理。
- 数据库脚本分为两类
  - 初始化数据库脚本
  - 增量式修改脚本

数据管理：初始化脚本

- 如果需要能够以自动化的方式重新建立一个应用程序的运行环境，那必须提供一个**初始化**数据库**结构**以及必要的初始化**数据的脚本**。
- 脚本中的内容应该包括：
  - 清除原有的数据库
  - 创建数据库及数据库模式
  - 加载必要的初始化数据

数据管理：增量式修改

- 对于大多数具有一定生命周期的应用，数据库的演化要复杂得多。
- 以自动化的方式迁移数据最有效的机制是**对数据库进行版本控制**。
- 每次对数据库进行修改时，需要创建两个脚本
  - 对数据库升级的脚本
  - 对数据库降级的脚本
- 部署好的系统中需要有当前部署的版本信息，自动化部署工具可以根据当前的版本信息和目标的版本信息，决定如何调用升级和降级的脚本。

数据管理：增量式修改

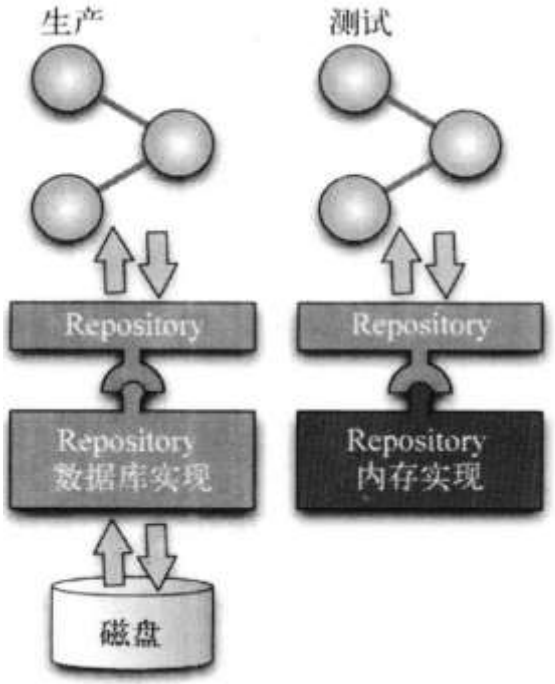
- 通常，升级脚本的难度并不算高，需要的是养成所有修改都是通过纳入配置管理的脚本来修改这样的习惯。
- 但是，降级回滚的脚本有时会困难些
  - 在升级过程中删除了某些数据，造成回滚的困难。
  - 在升级和回滚之间系统又产生了新的数据，回滚后的旧版本无法接受这些数据

数据管理：增量式修改

- 对于第一种情况，我们需要在升级中保留这些数据，以便于降级的时候将其恢复
- 对于第二种情况
  - 如果是特定于新版本的功能产生的数据，可以将其丢弃。
  - 如果是原有版本的功能，但是在新版本中的结构不同，难以恢复，可以考虑将数据恢复至旧版本的状态，并将升级后用户执行的操作在回滚后的系统上**重新做一遍**。
    - 这需要系统在架构上将用户的命令记录为**可重复执行**的日志

测试数据的管理

- 为单元测试进行数据库的模拟，有两种方式：
  - 对Repository进行Mock
  - 使用基于内存的数据库



测试数据管理

- 测试的独立性
  - 总是将数据库中数据的状态恢复到该测试运行之前的状态。
  - 通过测试完成后回滚事务是达到这种效果的最佳方法

大纲

- 软件交付的问题
- 软件配置管理
- **持续集成**
  - 准备工作与前提
  - 6步提交法
  - 最佳实践
  - 推荐实践
- 安全开发过程
- 持续交付：部署流水线及脚本化
- 应用程序的部署与发布

持续集成

- 很多软件项目都有一个非常奇怪而又常见的特征，即在开发的过程里，应用程序在相当长的一段时间内**无法运行**。
- 持续集成是一种根本的颠覆，让软件经常性地处于可用的状态。
- 这部分的主要内容是讨论如何实现持续集成

持续集成

- 准备工作
  - 版本控制
    - 所有的项目相关内容都应该加到版本控制库中
  - 自动化构建
    - 需要在命令行中启动构建过程，而不是在IDE中
      - 可以被其他工具调用
      - 可以对脚本本身进行测试

持续集成的前提

- 频繁提交
  - 并且需要是提交到主干。在使用生存期很长的分支的情况下，并不可能真正做到持续集成。
- 创建全面的自动化测试套件
  - 自动化构建如果只是编译和打包，那不能给人足够的信心确信构建好的软件能够工作。
  - 三类自动化测试会在持续集成中应用：
    - 单元测试
    - 组件测试
    - 验收测试

持续集成的前提

- 保持较短的构建和测试过程
  - 过长的构建和测试时间会阻碍团队成员提交代码前执行构建和测试，导致失败的概率越来越大
  - 提交的频率会减少
- 管理开发工作区
  - 管理开发环境
  - 管理代码，测试数据，各种脚本
  - 管理依赖
  - 确保自动化测试能够在开发机上运行

持续集成

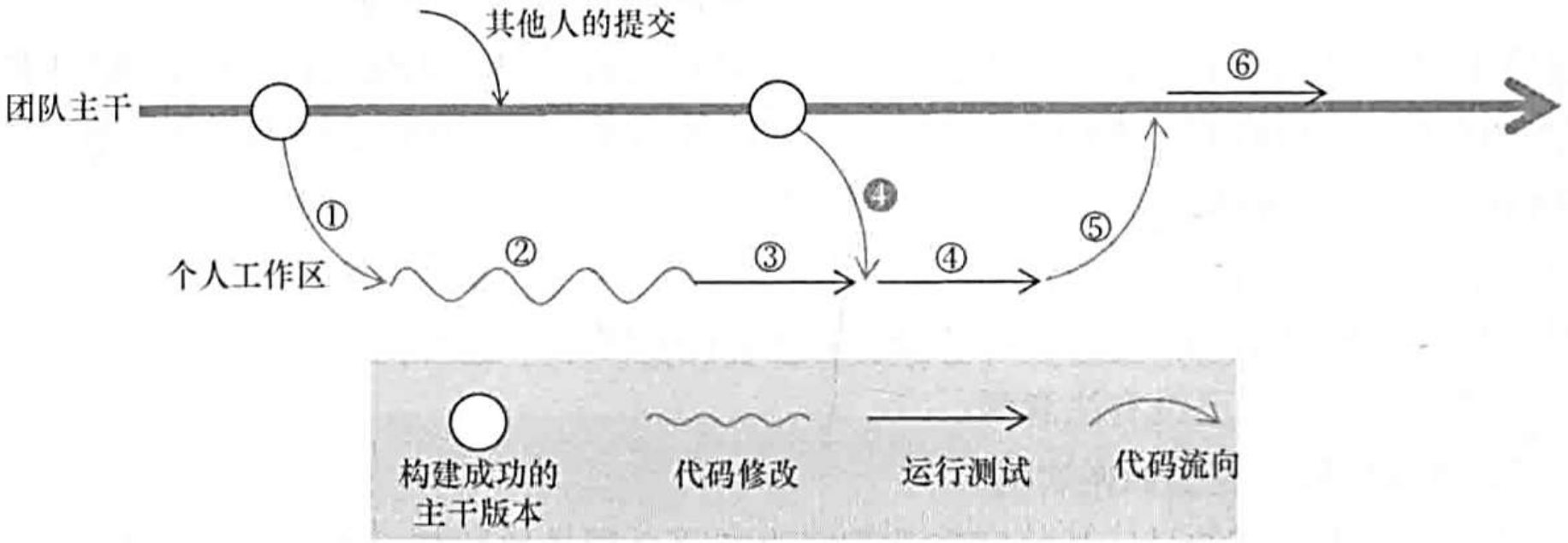
- 6步提交法
  - ① 检出最近成功的代码
  - ② 修改代码
    - 在个人工作区中对代码进行修改，包括实现产品新功能的代码和测试哦用例代码
  - ③ 第一次个人构建
    - 在准备提交前，首先通过自动化测试用例执行一个自动化验证。

持续集成

- 6步提交法
  - ④ 第二次个人构建
    - 在前一次检出代码之后，主干上可能又有了新的代码，需要更新并合并，并再次通过自动化测试用例执行自动化验证。
  - ⑤ 提交代码到团队主干
    - 第二次个人构建成功后，提交代码至团队主干
  - ⑥ 提交构建
    - 持续集成服务器检测到代码变更后运行自动化质量验证，如果构建失败，应该立即着手修复。

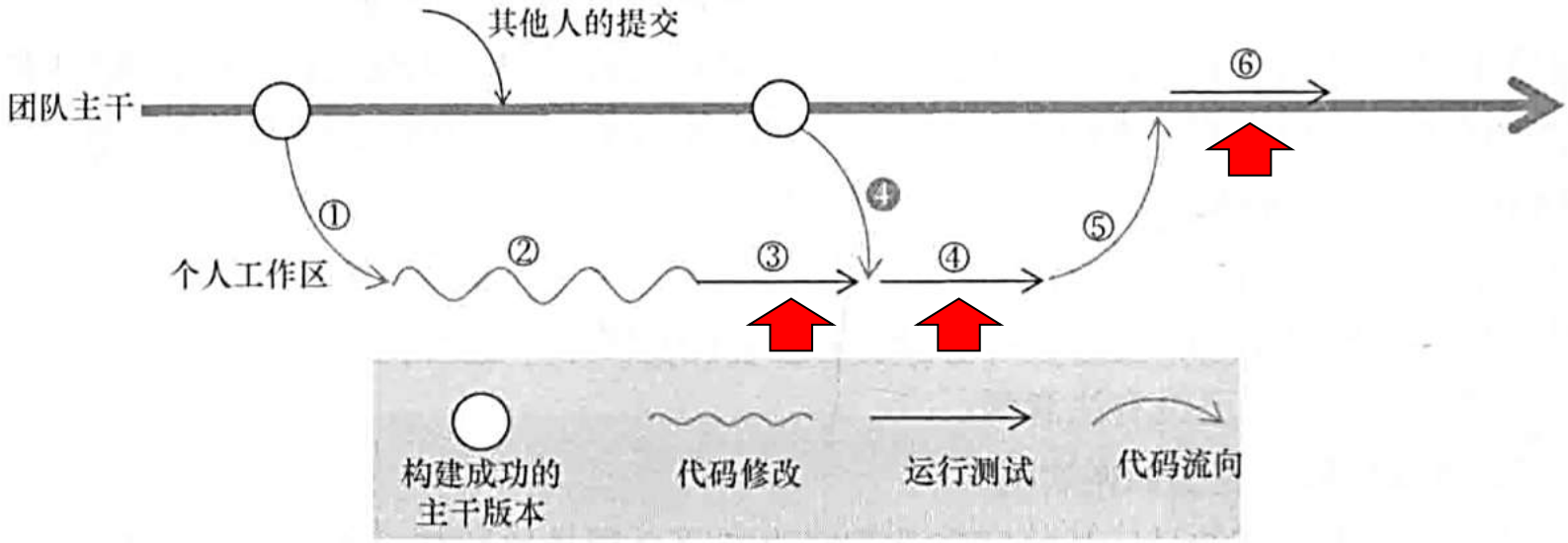
持续集成

- 6步提交法



持续集成

- 在这个过程中，共有3次验证
  - 第③步用于验证自己的增量的质量
  - 第④步用于验证与其他人合并后的质量
  - 第⑥步验证的是在一个干净受控环境中提交的质量



持续集成：最佳实践

- 构建失败后不要提交新的代码
  - 在开发环境中构建失败后，开发人员应该尽快找出失败的原因，修复后再提交
  - 提交前确保在开发环境运行所有提交的测试
    - 每次提交都能够是一个可发布的候选版本
  - 等提交测试通过后再继续工作
  - 回家之前，构建必须处于成功的状态
  - 时刻准备着，并能够回滚到前一个版本，但回滚之前需要规定一个修复时间
  - 不要将失败的测试注释掉
  - 为自己导致的问题负责
  - 测试驱动的开发

持续集成：推荐实践

- 极限编程开发实践
- 若违背架构原则，就让构建失败
- 若测试运行变慢，就让构建失败
- 若有编译警告或代码风格有问题，就让构建失败

大纲

- 软件交付的问题
- 软件配置管理
- 持续集成
- **安全开发过程**
- 持续交付：部署流水线及脚本化
- 应用程序的部署与发布

安全开发过程

安全规划	安全设计	安全实现	安全测试	安全发布与部署
• 定义安全需求（基于威胁分析、市场准入安全要求和安全认证等） • 确定网络安全目标	• 威胁分析及风险评估 • 使用安全的平台及组件 • 安全设计检视	• 安全编程规范遵从 • 开发者测试 • 代码安全检视 • 静态分析 • 门禁 • 安全编译选项实施	• 安全功能测试 • 白盒安全验证 • 安全扫描测试 • FUZZ测试 • 渗透测试	• 病毒扫描 • 数字签名

开源软件安全

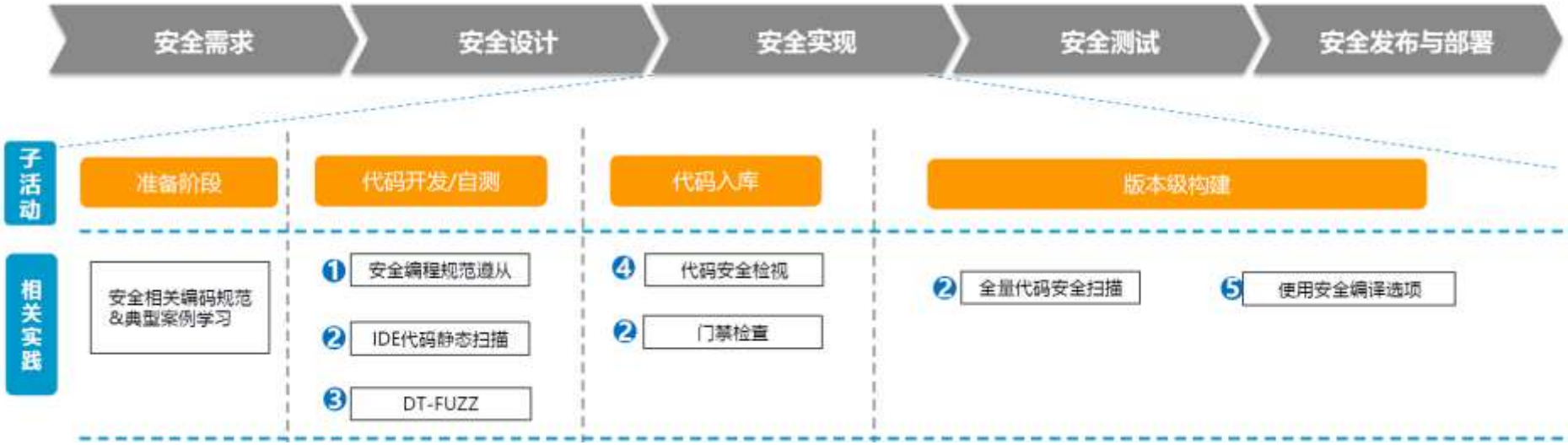
安全选型、规范使用、生命周期维护

漏洞管理

漏洞识别、漏洞响应、防止类似漏洞重现

来源：华为软件安全开发实践

安全实现



- 安全编程规范遵从
- 安全静态扫描
- 开发者测试-模糊测试（DT-Fuzz）
- 代码安全检视
- 安全编译规范遵从

来源：华为软件安全开发实践

安全实现——以缓冲区溢出问题为例

- 开发过程中，缓冲区溢出问题经常出现在内存拷贝、字符串拷贝、数值作为数组索引和指针偏移值使用的场景中。

```
const uint32_t NAME_SIZE = 256;
int main(int argc, char *argv[])
{
    char filename[NAME_SIZE];
    ...
    // 【错误】源字符串长度可能大于目标数组空间，造成缓冲区溢出
    strcpy(filename, argv[1]);
    ...
}
```

```
int main(int argc, char *argv[])
{
    char filename[NAME_SIZE];
    ...
    // 【错误】未考虑filename大小，当args[1]的大小 > (NAME_SIZE + 1 - 4)时可能会导致缓冲区溢出，注意”%s.txt”中的”.txt”4个字符
    sprintf(filename, "%s.txt", argv[1]);
    ...
}
```

安全实现——以缓冲区溢出问题为例

- **安全编程规范遵从**
- 安全静态扫描
- **开发者测试：Fuzz测试**
- 代码安全检视
- **安全编译规范遵从**

安全实现——以缓冲区溢出问题为例

- 安全编程规范遵从

✓ **正确使用安全函数**

```
const uint32_t NAME_SIZE = 256;
int main(int argc, char *argv[])
{
    ...
    char filename[NAME_SIZE];
   // 【错误】源字符串长度可能大于目标数组空间，导致缓冲区溢出
    strcpy(filename, argv[1]);
    ...
}
```

```
const uint32_t NAME_SIZE = 256;
int main(int argc, char *argv[])
{
    ...
    char filename[NAME_SIZE];
   // 【正确】正确使用安全函数，避免缓冲区溢出
    errno_t rc = strcpy_s(filename, NAME_SIZE, argv[1]);
    if  (rc != EOK)  {
       ... // 异常处理
    }
    ...
}
```

✓ **数据作为数组索引时必须确保在数组大小范围内**

```
const uint32_t DEV_SIZE = 128;
void Foo(uint32_t inputIndex, uint32_t id)
{
    static devs[DEV_SIZE] = {0};

   // 【错误】直接使用外部数据作为数组索引，可能导致越界
    devs[inputIndex] = id;
    ...
}
```

```
const uint32_t DEV_NUM = 128;
void Foo(uint32_t inputIndex, uint32_t id)
{
    uint32_t devs[DEV_NUM] = {0};

   // 【正确】对数组索引进行合法性校验，避免数组索引越界
    if (inputIndex >= DEV_NUM) {
       ... // 异常处理，截断执行流
    }
    devs[inputIndex] = id;
    ...
}
```

✓ **校验外部数据中整数值的合法性**（内存拷贝长度）

```
static ssize_t foo_write(..., const char __user *buf, size_t count, ...)
{
    uint32_t knlInfo[2] = {0};
   // 【错误】 当 count>8 字节时，导致缓冲区溢出
    if (copy_from_user(knlInfo, buf, count)){
        ...
    }
    ...
}
```

```
static ssize_t foo_write(..., const char __user *buf, size_t count, ...)
{
    uint32_t knlInfo[2] = {0};
   // 【正确】校验count合法性，避免缓冲区溢出
    if (count > 8) {
        ... // 异常处理，截断执行流
    }
    if (copy_from_user(knlInfo, buf, count)){
        ...
    }
    ...
}
```

安全实现——以缓冲区溢出问题为例

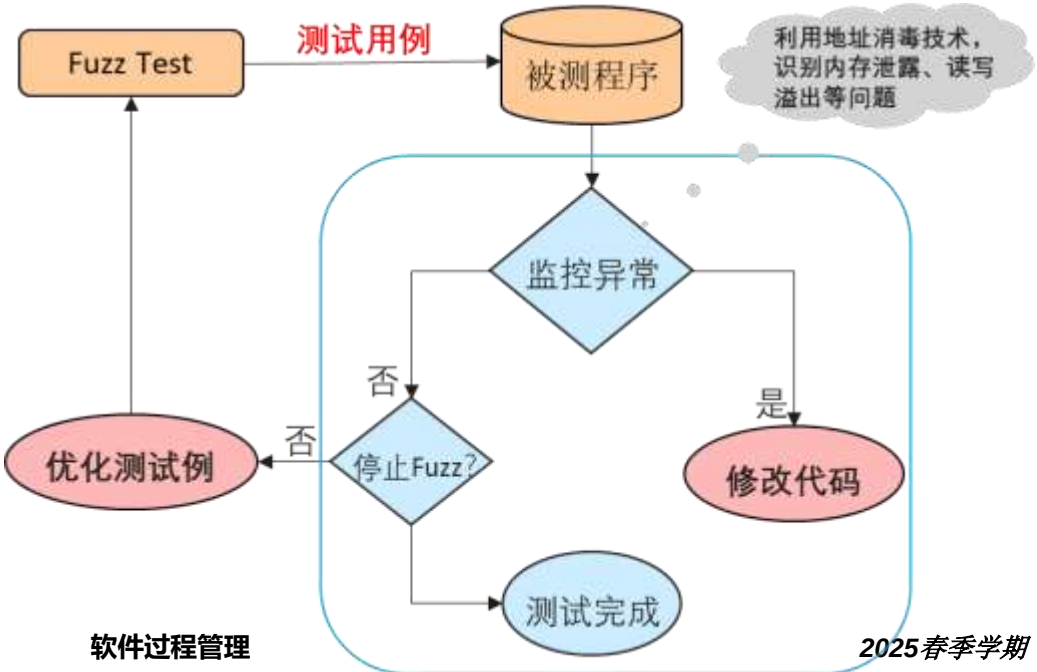
- 开发者测试——模糊测试
  - **模糊测试（Fuzzing）**是一种自动化测试技术
  - Fuzz工具根据用户给定条件，通过算法自动生成测试用例，将大量恶意/随机数据发送到被测试系统，并监视系统运行过程中的异常（断言、异常、deadloop、错误、逻辑错误、倒换、重启等），从而发现应用程序中可能存在的安全问题
  - 常用开源Fuzz测试工具：libFuzzer、syzkaller、AFL

测试时要**设置合适的终止条件**，比如采用如下终止条件：

满足：未产生新的bug，且未产生新的测试用例

并满足下面条件之一：

时间已经达到3小时，或数量达到3000万



安全实现——以缓冲区溢出问题为例

- 使用安全编译选项

1.栈保护选项：-fstack-protector

在缓冲区和控制信息间插入一个canary word。当缓冲区被溢出时，在返回地址被覆盖之前canary word会首先被覆盖。通过检查canary word的值是否被修改，就可以判断是否发生了溢出攻击。或有意将局部变量中的数组放在函数栈的高地址，而将其他变量放在低地址，防止由于数组的溢出覆盖到其它变量的值。

Linux	-fstack-protector --param ssp-buffer-size=4	堆栈保护，仅对局部变量中包含字符数组声明并且字符数组长度超过4个字节的函数实施堆栈保护。
	-fstack-protector --param ssp-buffer-size=4 -Wstack-protector	该选项对于包含长度小于4个字节字符串数据组的函数没有进行保护，对这类函数进行告警。
	-fstack-protector-all	对所有函数进行堆栈保护。
Win	/GS	Visual studio编译器通过/GS选项打开栈保护选项。

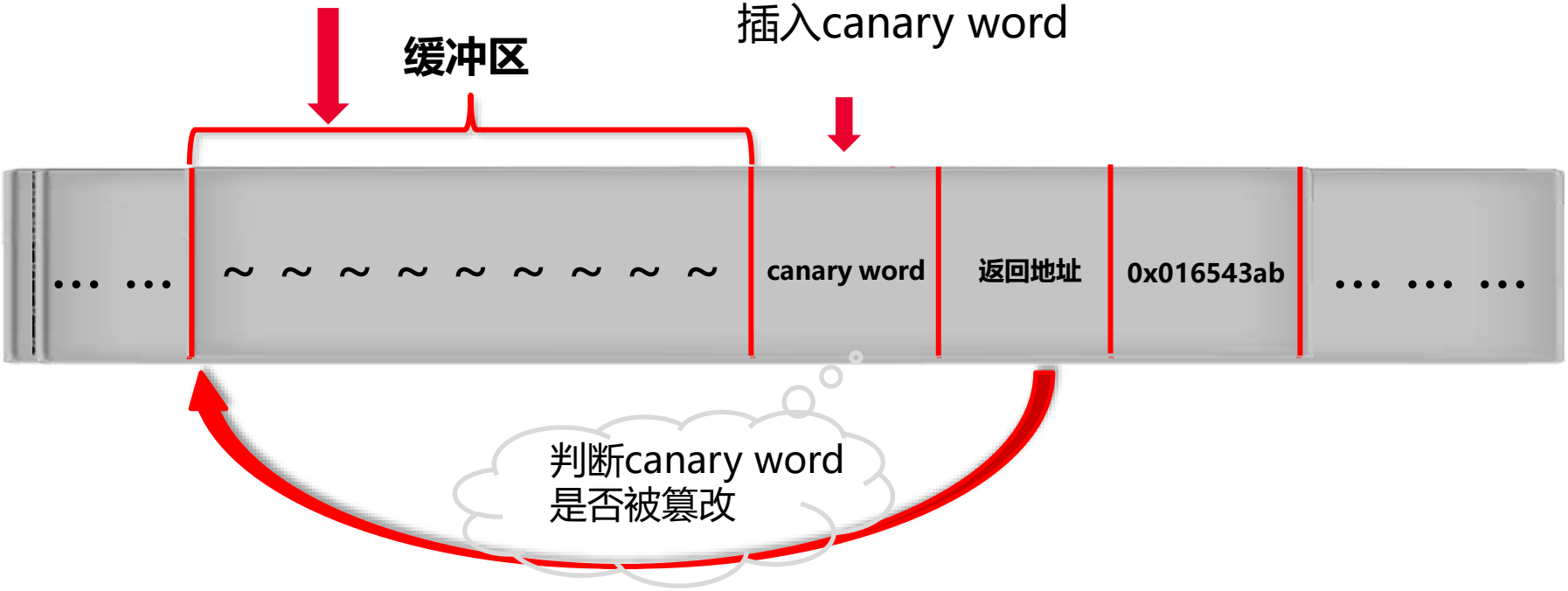
2.堆栈不可执行：DEP\NX

设置堆栈中的数据只可以读和写，不可执行。在程序运行时如果发生溢出，则终止程序。给出错误提示Segmentation fault。

Linux	-noexecstack	设置堆栈中的数据只可以读和写，不可执行。
Win	/NXCOMPAT	Visual studio编译器通过/NXCOMPAT选项打开DEP选项。

安全实现——以缓冲区溢出问题为例

- 使用安全编译选项



大纲

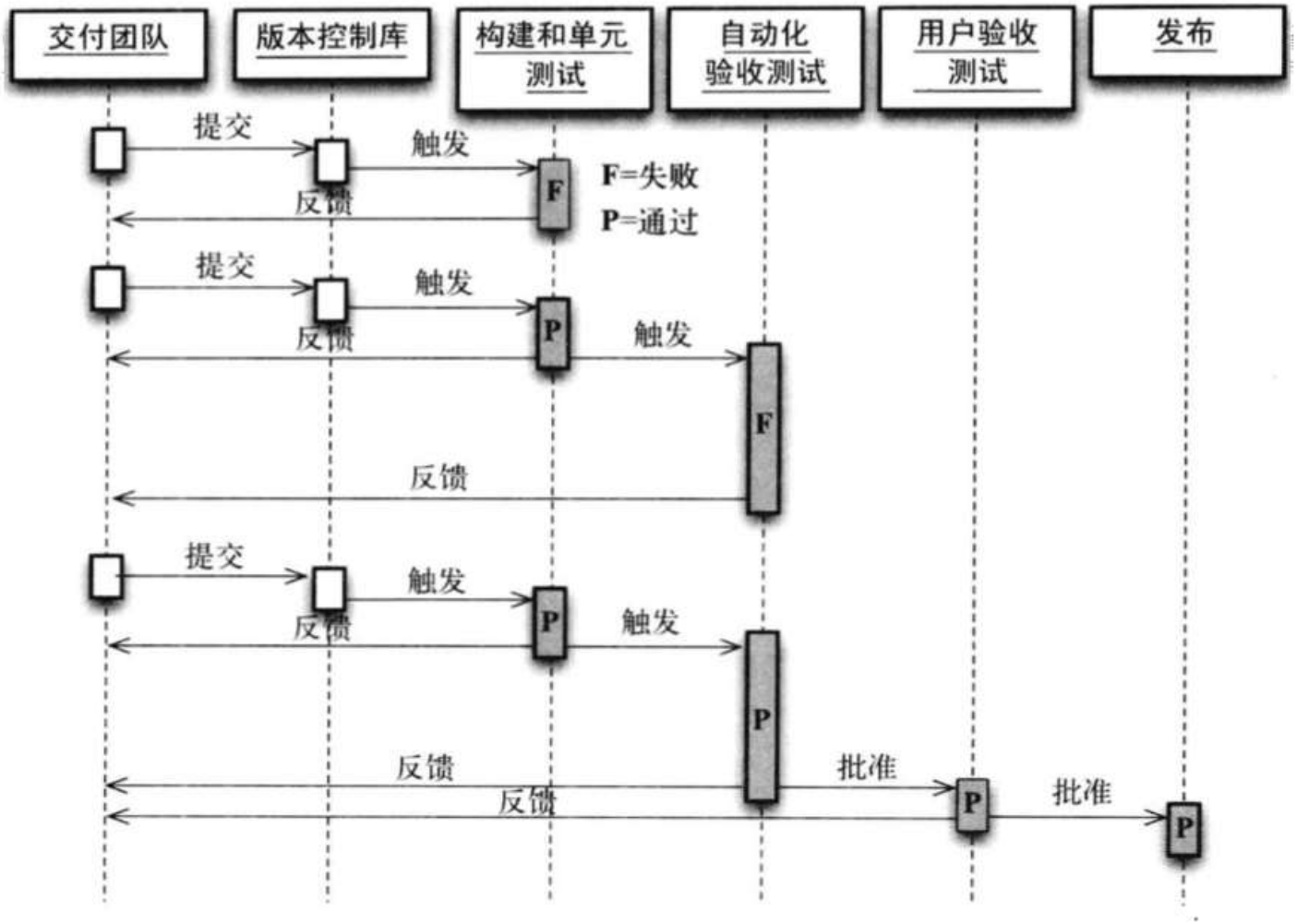
- 软件交付的问题
- 软件配置管理
- 持续集成
- 安全开发过程
- **持续交付：部署流水线及脚本化**
  - 部署流水线
  - 提交阶段、自动化验收测试阶段、发布阶段
  - 相关实践
  - 构建与部署的脚本化
- 应用程序的部署与发布

部署流水线

- 部署流水线是指软件从版本控制库到用户手中这一过程的自动化表现形式。
- 部署流水线是对这一流程的建模，这个模型需要支持控制和查看从**提交到版本库**开始，到**发布给用户**的整个过程。
  - 在软件开发的初期，或者是比较简单的软件系统，这个过程通常比较简单。
  - 但随着系统的复杂度和分布程度的增加，我们需要一个模型帮助我们理解这个过程。同时，模型也是工具化和自动化的前提。

部署流水线

- 一个部署流水线的模型

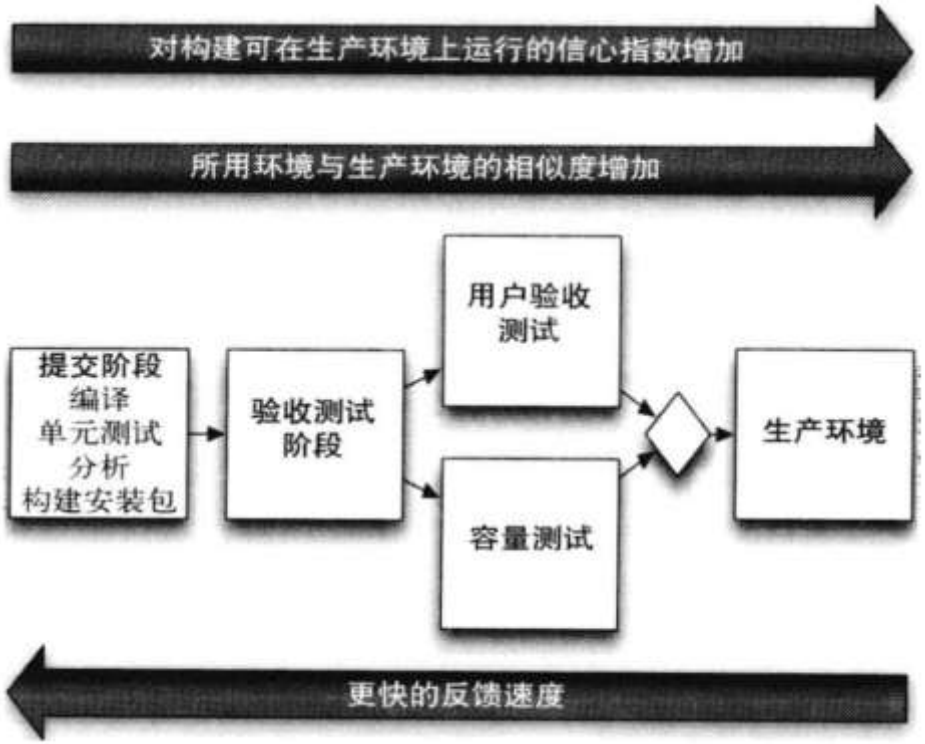


部署流水线

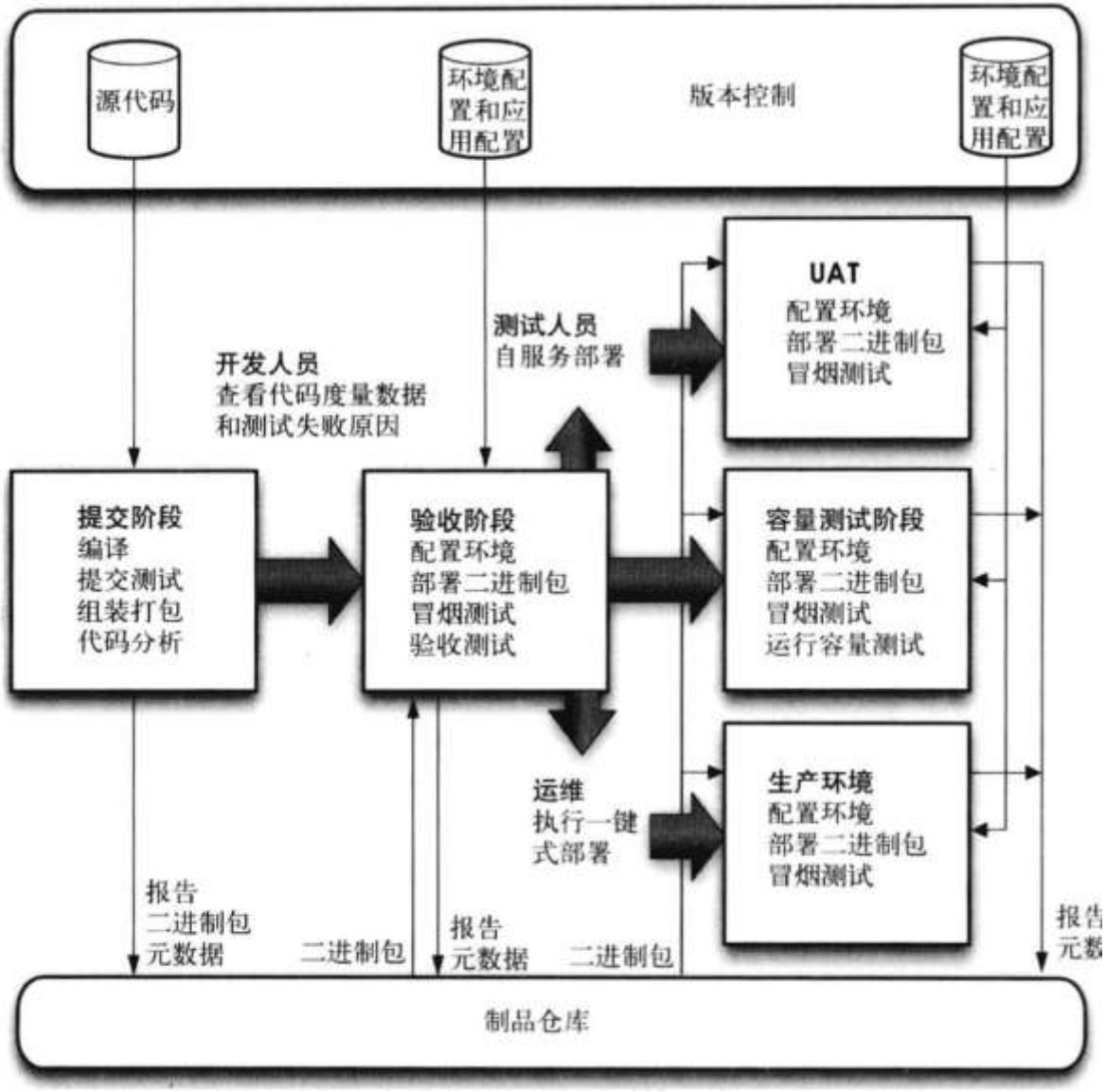
- 部署的过程通常可以分为下面几个阶段
  - 提交阶段
    - 从技术的角度断言整个系统是可以工作的
  - 自动化验收测试阶段
    - 从功能和非功能的角度上断言整个系统是可以工作的
  - 手工测试阶段
    - 断言系统是可用的。
  - 发布阶段
    - 将软件发布给用户，可以是构建好套装软件，也可以是直接发布到生产环境或者试运行环境。

部署流水线

- 通常，在部署流水线靠前的阶段，验证的内容尽可能与具体的环境无关，这样能够适配各种不同的环境，在靠后的阶段，验证时的环境与生产环境接近，这样能够增强在生产环境上正确运行的信心。



一个典型的部署流水线


源代码
环境配置和应用配置
版本控制
环境配置和应用配置
UAT
配置环境
部署二进制包
冒烟测试
测试人员
自服务部署
开发人员
查看代码度量数据
和测试失败原因
提交阶段
编译
提交测试
组装打包
代码分析
验收阶段
配置环境
部署二进制包
冒烟测试
验收测试
容量测试阶段
配置环境
部署二进制包
冒烟测试
运行容量测试
运维
执行一键式部署
生产环境
配置环境
部署二进制包
冒烟测试
报告
二进制包
元数据
二进制包
报告
元数据
二进制包
报告
元数
制品仓库

提交阶段

- 提交阶段的内容包括
  - 编译代码
  - 运行一组提交测试
  - 执行代码分析来检查代码的健康状态
  - 创建二进制包
  - 为后续阶段提供必要的准备，比如数据库

提交阶段

- 可以根据项目设置度量的阈值，超出便强制失败
  - 测试覆盖率
  - 代码重复度
  - 圈复杂度
  - 耦合度
  - 代码风格
  - ……
- 通过提交阶段后，开发人员可以认为被从上一个任务中释放出来，可以开始下一个任务。

自动化验收测试阶段

- 在提交阶段的测试（大多是单元测试）是非常重要，但有些类型的错误是它无法捕获的。
- 单元测试需要与具体环境尽可能解耦，这使得有必要确认代码是否能在接近真实的环境中正常运行。
- 验收测试可能会比较昂贵
  - UI测试
  - 需要模拟生产环境
- 验收测试的典型的方法是BDD中提供的方法。

发布阶段

- 为了安全发布一个系统，我们需要
  - 让尽可能多的人参与到发布的计划中
  - 尽可能自动化
  - 在测试环境、类生产环境中反复演练
  - 在发生意外的情况下，能够撤销发布
    - 撤销发布的流程应该与正常发布的流程一致
  - 需要制定数据迁移的计划

相关实践

- 只生成一次二进制包
  - 如果一个项目被分解为多个组件，每个组件只有在发生变动后才会重新编译，生成一个新的二进制包。
  - 在组件没有发生变化的情况下，构建整个系统时应该直接使用已经构建好的二进制包，而不必重新编译。
  - 这种实践会大大减少编译花费的时间。
  - 二进制包应该尽可能与环境无关，或者说，不建议为每个环境构建一个二进制包。

相关实践

- 对不同的环境采用同样的部署方式
  - 部署会涉及到各种配置和脚本，这些**配置的格式和脚本正确性也需要进行测试。**
    - 如果对不同的环境采用了不同的配置文件和脚本，那么测试的工作量会增加，部署失败的风险（特别是向生产环境上部署的风险）也会大大增加，因为在生产环境上测试的机会最少。
    - 如果使用的是相同的配置文件和脚本，但还是发布失败，那么这种情况下定位失败的原因会比较容易确定：
      - 配置文件中的某个配置项的值有错
      - 依赖的服务不正常
      - 环境本身的配置有错

    可以排除脚本或者配置文件格式错误的可能性。

相关实践

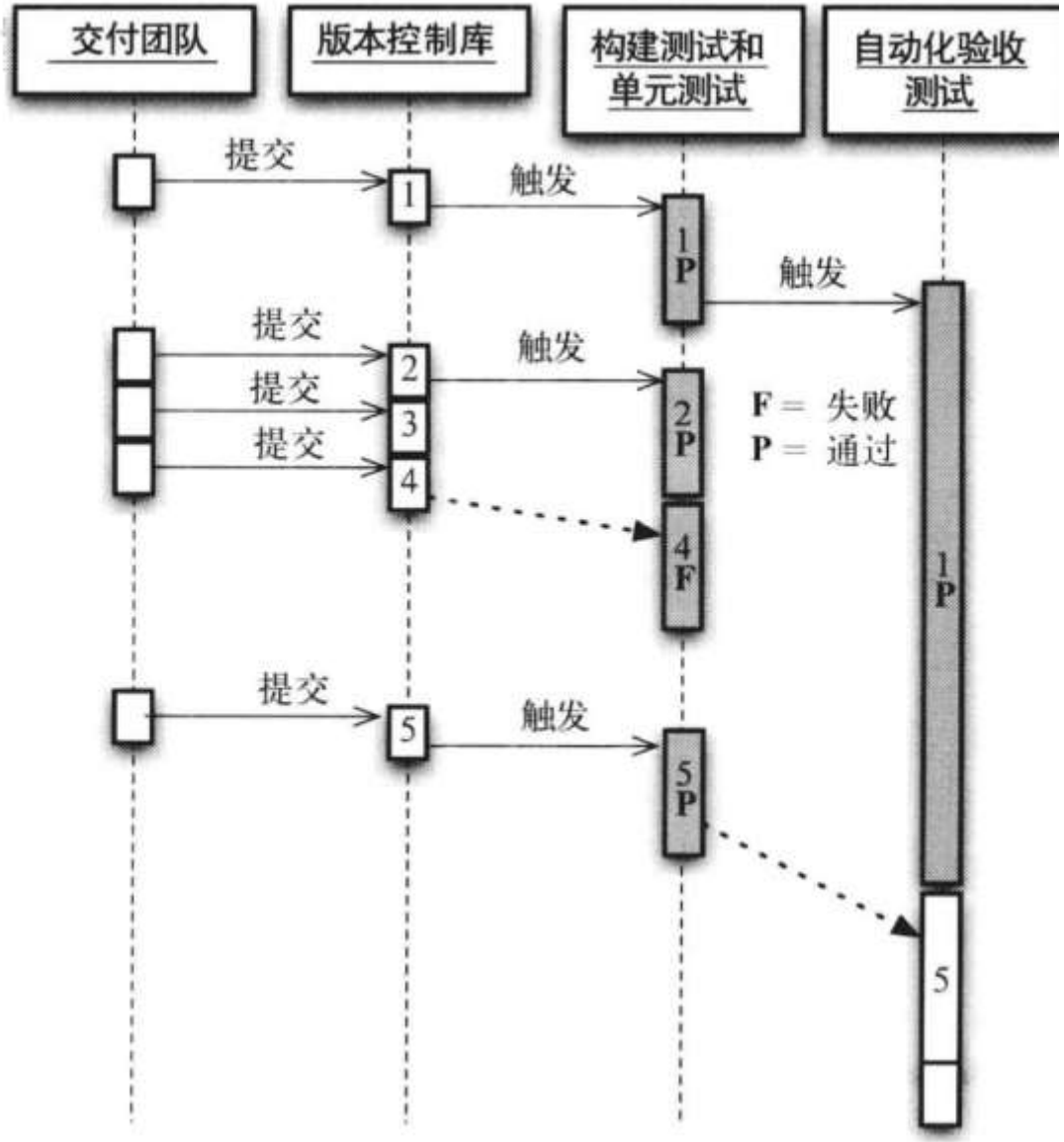
- 对部署的结果进行冒烟测试
  - 简单的冒烟测试可能是
    - 能够登录系统
    - 能够访问几个功能比较复杂的页面
  - 这些也可以通过自动化脚本来验证

相关实践

- 向生产环境的副本中部署
  - 如果生产环境与开发和测试环境有比较大的差异，直接部署到生产环境上的信心不足，在这种情况下，可以构建一个类生产环境：
    - 相同的基础设施，如网络拓扑和防火墙
    - 相同的操作系统和补丁
    - 相同的软件栈
    - 类似的数据状态（模拟数据迁移）

每次变更都立即在流水线中传递

- 自动化流水线需以某种策略检测每次变更，并调度后续的一系列动作，并提供反馈。



只要有环节失败就停止整个流水线

- 对于团队成员，必须通过所有单元测试才能提交
- 对于部署流水线，必须能够完成直至部署的步骤才能认可提交成功
- 一旦失败，整个团队应该优先修复这个问题，在继续进行后续的开发

实现一个部署流水线

- 实现一个部署流水线的过程如下
  - 对价值流建模，创建一个可工作的框架
  - 将构建和部署流程自动化
  - 将代码分析和单元测试自动化
  - 将验收测试自动化
  - 将发布自动化

部署流水线

- 部署流水线需要达到的效果
  - 阻止没有经过充分测试的，或者不满足功能需求的版本进入生产环境
  - 使得发布成为一件平常的事情，只要需要，可以随时发布
  - 如果能够支持自动安全的回滚，会进一步降低发布的风险

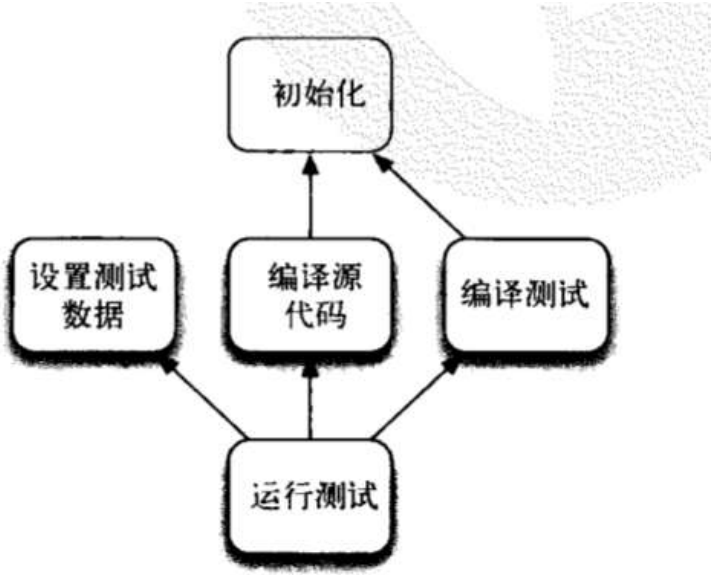
构建与部署的脚本化

- 针对每一个项目，构建与部署实际上也是属于这个项目一部分的，需要开发的软件。
- 这个软件的开发语言通常是各种脚本化的语言。
- 这个软件不仅仅需要从项目的开始就启动开发，而且一直要持续到维护阶段。需要像其他源代码一样精心地设计和维护它，定期测试与使用。

构建与部署的脚本化

- 构建工具
  - 构建工具识别的通常是一种用来描述任务及其依赖关系的DSL（Domain-Specific Language）。使用这个DSL，可以对一个构建需要的任务集及其依赖关系进行建模。
  - 下面是一个典型的依赖关系图：

这种依赖关系用不同的构建工具会有不同的表达方式



构建与部署的脚本化

- 典型的构建工具有：
  - Make
  - Ant
  - NAnt与MSBuild
  - Maven
    - +能够自动管理Java库和项目间的依赖
    - - 定义了一套默认的构建与部署的生命周期，对于例外的情况比较难以定制。

构建与部署的脚本化

- 最佳实践1
  - 为部署流水线的每个阶段创建脚本
- 最佳实践2
  - 使用恰当的技术部署应用程序，比如，使用特定与中间件的部署和配置工具
- 最佳实践3
  - 使用同样的脚本向所有环境部署
- 最佳实践4
  - 确保部署流程是幂等的，即无论开始部署是目标环境处于何种状态，部署流程总是令目标环境达到同样的状态。

大纲

- 软件交付的问题
- 软件配置管理
- 持续集成
- 安全开发过程
- 持续交付：部署流水线及脚本化
- **应用程序的部署与发布**
  - 创建发布策略
  - 应用程序的部署和晋级
  - 部署回滚和零停机发布
  - 最佳实践

应用的部署与发布

- 部署和发布的区别
  - 发布可以认为是一种特殊的，最重要的部署。
  - 发布是直接影响生产环境的一种部署
- 发布与部署尽可能遵循同样的过程，但他们之间还是有不同之处
  - 区别在于对回滚的支持

应用程序的部署和晋级

- 要让软件的部署变得可靠，其关键在于每次部署是都要**用同样的方法**，用**相同的流程**向每个环境进行部署，包括向生产环境的发布。
- 要从第一次向测试环境部署时就开始使用基于脚本的自动化部署。

应用程序的部署和晋级

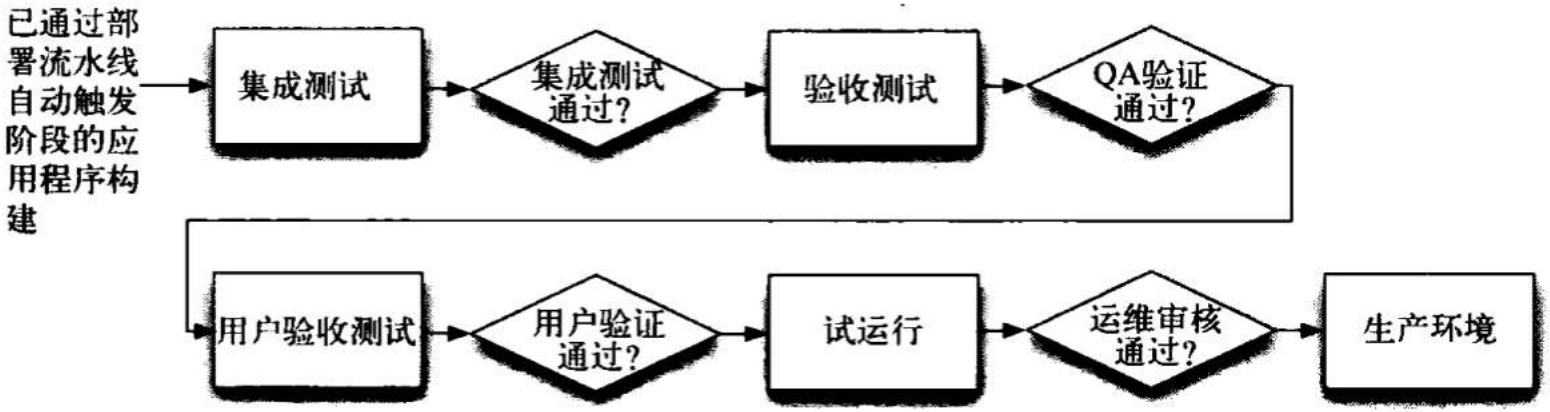
- 首次向类生产环境部署（比如测试环境）
  - 在部署流水线上，有了处于提交阶段的内容。
  - 准备好了一个可用于部署的类生产环境
  - 通过一个自动化的过程获取部署需要的二进制包，并完成自动部署
  - 进行简单的冒烟测试，验证本次部署的正确性。
- 类生产环境应该：
  - 与生产系统的操作系统一致
  - 依赖的软件与生产系统一致（不能有开发工具）
  - 与生产系统环境管理的方式一致
  - 如果软件需要用户自己安装，那么UAT环境要具有一定的代表性

应用程序的部署和晋级

- 晋级(promoting)
  - 构建系统的这样一种能力：项目团队可以选择某个版本，比如测试版、试运行版、发布版，通过一个命令或一个按钮完整这个版本的部署。
- 部署流水线的实现会随着应用程序的复杂而越来越复杂，可能需要设计不同的各种环境之间的晋级门槛，需要考虑的内容有：
  - 设计每个晋级门槛需要通过哪些不同的测试
  - 设计谁来批准每个晋级门槛的通过与否

应用程序的部署和晋级

- 测试、发布流程与晋级门槛



- 自动化部署机制可以使得构建版本的晋级变成一件非常简单的事。

应用程序的部署和晋级

- 不同的晋级中（测试环境，试运行环境，开发环境），一个重要的区别是**配置的不同**。而确认是否正确地设置各自的配置信息是一件复杂的任务。
  - 手工做冒烟测试
  - 同一个的环境中的配置项通常是相关的，成为一个组合版本，部署系统可以强制把这种组合当做一个整体进行部署。

应用程序的部署和晋级

- 部署到联合环境
  - 如果多个应用共享一个环境，需要确保部署的时候不会影响其他应用的正常运行。
  - 可以通过虚拟化技术隔离多个应用
- 部署到试运行和运行环境
  - 项目一开始，就需要准备好试运行环境。如果资源有限，可以将预先准备好的生产环境作为试运行环境。
  - 如果可能的话，发布之前就把系统在生产环境上部署好，发布时通过路由机制将所有访问入口指向这个环境，这有时被称为**蓝绿部署**
  - 如何可能的话，可以把环境先发布给一小撮用户，这个叫**金丝雀部署**

部署回滚和零停机发布

- 万一部署失败，回滚部署是特别重要的，在生产环境上尤其如此。
- 有几种回滚的方法：
  - 重新部署原来的正常版本
  - 零停机发布
  - 蓝绿部署
  - 金丝雀发布

部署回滚和零停机发布

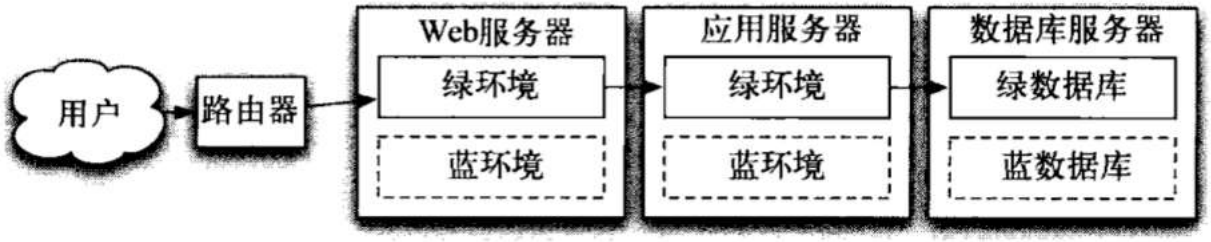
- 重新部署原来的正常版本
  - 好处是简单，相对风险较小
  - 缺点
    - 需要有一段停机的时间
    - 失去了错误的现场，增加了调试的难度
    - 恢复的数据是老的数据，丢失了新版本发布之后产生的数据

部署回滚和零停机发布

- 零停机发布（也成为热部署），是一种将用户从一个版本几乎瞬间转移到另一个版本的方法。
- 零停机发布的关键在于将发布流程中的各个环节解耦，尽量使他们能够独立发生。
- 在升级应用前，就应该能够将应用所依赖的静态资源的新版本放在合适的位置

部署回滚和零停机发布

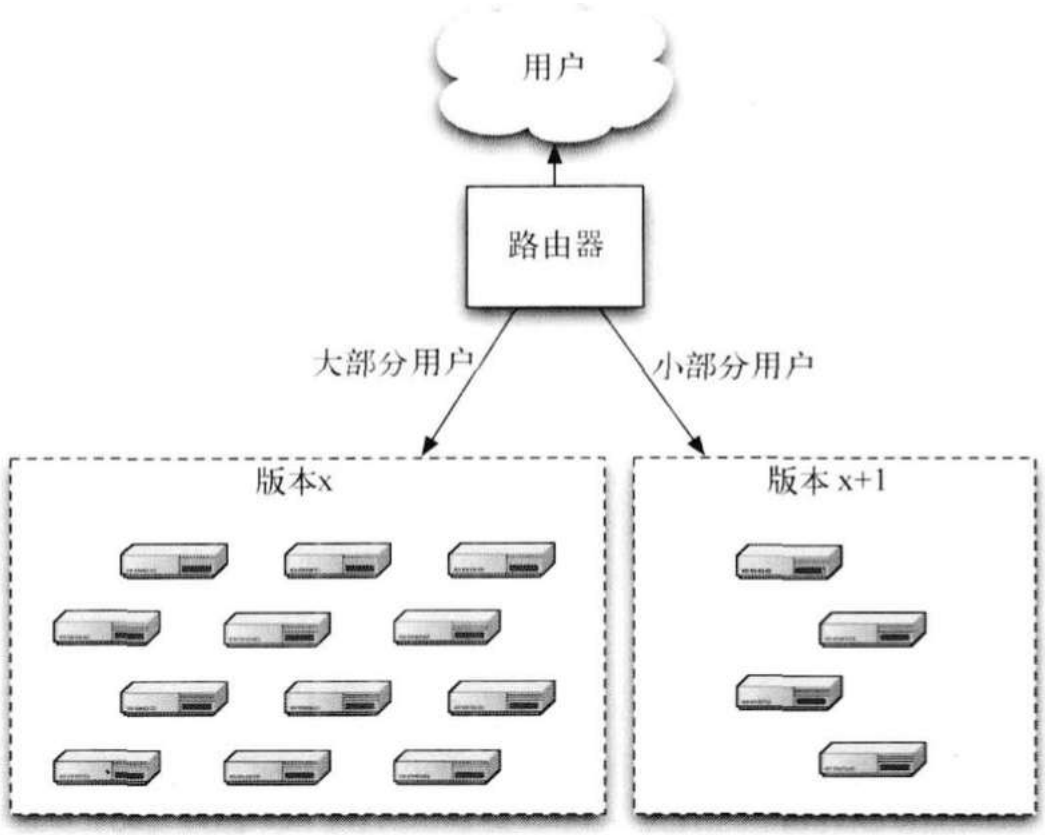
- 蓝绿发布
  - 有两个生产环境：一个叫蓝环境，一个叫绿环境



  - 如果用户当前使用的是绿环境，发布新版本时可以发布到蓝环境，在蓝环境上执行冒烟测试
  - 准备就绪后，可以迅速切换至蓝环境，达到不停机发布的效果
  - 数据的同步是最有挑战性的。在这种情况下可以让系统在一小段时间内处于只读状态。

部署回滚和零停机发布

- 金丝雀发布（灰度发布）
  - 把应用的某个新版本发布到部分服务器中，只提供给一部分人使用，从而快速得到反馈。



最佳实践

- 执行部署的人员参与部署流程与脚本的设计与创建
- 记录部署活动
- 不要删除旧文件，而是移动到别的位置
- 部署是整个团队的职责
- 运行服务器上的服务程序不应该有GUI

最佳实践

- 为新部署预留预热期
- 快速失败
  - 在系统初始化时尽可能自动检查必须的资源和配置，有问题就报错退出
- 不要直接对生产环境进行修改

小结

- 软件交付的问题
  - 一些常见的反模式以及解决方法，自动化地持续交付
- 软件配置管理
  - 版本管理、模块间依赖的管理、环境管理、数据管理
- 持续集成
  - 如何提交?
- 安全开发过程
  - 内建的代码安全保障
- 持续交付：部署流水线及脚本化
  - 自动化
- 应用程序的部署与发布
  - 策略与方法