Lecture11 DevOps & Continuous Integration

1. 开发运维 DevOps

介绍

DevOps 集成了开发人员和运营团队，通过自动化基础设施、自动化工作流和持续测量应用程序性能来提高协作和生产力

- DevOps = Development + IT Operation

2. 持续集成 Continuous Integration

介绍

- 持续集成是一种软件开发实践，团队成员**频繁**地**集成**他们的工作，通常每个人**至少每天集成**到每天多个集成，每个集成都通过**自动构建**（包括**测试**）进行验证，以尽可能快地检测集成错误

持续继承的好处

- 减少集成问题
- 允许团队更快地开发内聚软件

持续集成的 10 个原则

- 维护代码库，进行版本控制
- 自动化的构建
- 让您的构建进行自我测试
- 每个人每天都 Commit 到主分支
- 每个提交都应该在集成机器上构建主分支
- 让构建跟更快
- 在生产环境的克隆中进行测试
- 使任何人都能轻松地获得最新的可执行文件
- 每个人都能看到进度
- 自动部署

集成问题

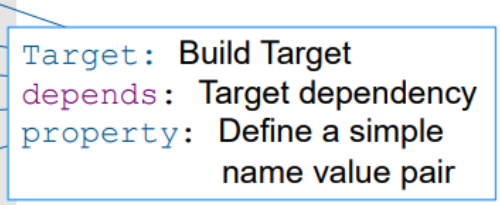
- Merge 冲突
    - 同时修改同一文件
- 编译冲突
    - 修改后的程序不再编译
- 测试冲突
    - 修改后的程序编译，但未能通过测试

自动化的构建 Automate the build

- 每日构建：每日编译可执行文件
  - 开发用于编译和运行系统的脚本
    - 自动构建工具
      - Java: Apache Ant, Maven, Gradle
      - C/C++: make
  - 优点
    - 允许您测试集成的质量
    - 可见的进度
    - 快速捕获/暴露破坏程序的 bug

通过命令行构建

```
<project name="simpleCompile" default="deploy" basedir=".">
   <target name="init">
      <property name="sourceDir" value="src"/ >
      <property name="outputDir" value="classes" />
      <property name="deployJSP" value="/web/deploy/jsp" />
      <property name="deployProperties" value="/web/deploy/conf" />
   </target>
   <target name="clean" depends="init">
      <deltree dir="${outputDir}" />
   </target>
   <target name="prepare" depends="clean">
      <mkdir dir="${outputDir}" />
   </target>
   <target name="compile" depends="prepare">
     <javac srcdir="${sourceDir}" destdir="${outputDir}" />
   </target>
   <target name="deploy" depends="compile,init">
     <copydir src="${jsp}" dest="${deployJSP}"/>
     <copyfile src="server.properties" dest="${deployProperties}"/>
   </target>
</project>
```



- 需要 `build.xml` 进行生成
- 自动构建的一种常见方法是允许从命令行编译项目

让您的构建进行自我测试

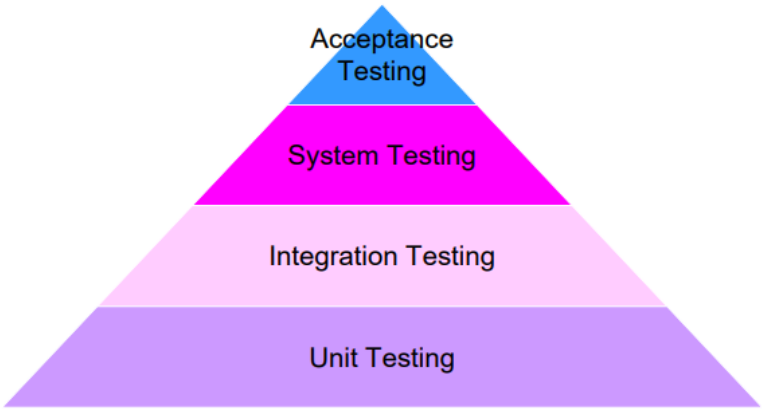
- 自动测试：可以从命令行运行的测试
- 示例
  - 单元测试
  - 集成测试
  - 冒烟测试

冒烟测试 Smoke Test

- 对每日构建运行一组快速测试
  - 涵盖软件最重要的功能，**但不详尽**
  - 检查代码是否着火或“冒烟”(中断)
  - 尽早暴露集成问题

> 如何进行冒烟测试的?
>
> - 把它装满水。如果有漏水，那就是坏杯子

软件测试层级



- **单元测试**：对软件的单个单元进行测试
  - 单位：应用程序的最小可测试部分
    - 例如：方法

```
public static double div(int x, int y){
      return x/y;
}
```

Input 1	Input 2	Output	Unit Test
1	2	0.5	`assertEquals(0.5, div(1, 2));`
1	1	1.0	`assertEquals(1.0, div(1, 1));`

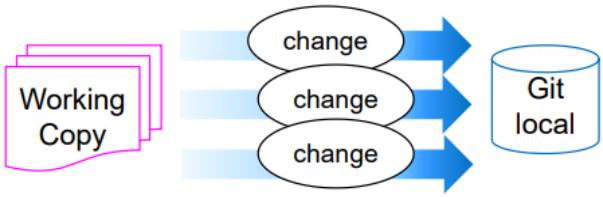
1	0	ArithmeticException	`@Test(expected=java.lang.ArithmeticException.class)`
			`public void testDivideByZero() {`
			`  div(1,0)`
			`}`

```
assertEquals(expected, div(1, 2));
```

- **集成测试**：通过测试两个或多个依赖的软件模块作为一个组来验证软件质量
  - 挑战：
    - 组合单元可能会在更多地方以更复杂的方式失效
    - 如何在不存在所有部分的情况下测试部分系统
    - 如何正确地模拟单位 A 的行为，以便从单位 B 产生一个给定的行为?
  - 大爆炸集成测试
    - 所有的组件被集成在一起
  - 优点
    - 对于小型系统十分方便
  - 缺点
    - 很难发现 bug
    - 由于需要测试的接口数量很多，因此很容易遗漏一些接口
      - 测试团队需要等待，直到所有的东西都集成了，这样测试的时间就会更少
      - 高风险关键模块没有被隔离和优先测试
- **增量集成测试 Incremental Integration Testing**
  - 开发一个功能“骨架”系统
  - 设计，编码，测试，调试一个小的新部件
  - 把这个和骨架结合起来
    - 在添加任何其他馅饼之前测试/调试它

- 优点
  - 错误更容易分离，发现，修改
    - 减少开发人员 bug 修复开销
  - 系统总是处于(相对)工作状态
    - 良好的客户关系，开发人员道德
- 缺点
  - 可能需要创建一些尚未集成的特性的“存根”版本

每天的 Commit



- 每天结束时向主仓库提交工作
  - 想法：减少 Merge 冲突
  - 这是新代码“持续集成”的关键
- 警告：不要仅仅为了维护日常提交实践而**提交错误代码**（不能编译，不能通过测试）
- 如果您的代码在一天结束时还没有准备好提交，要么提交一个一致的子集，要么更加灵活一些，不是完全一定要遵守每天都要 Commit

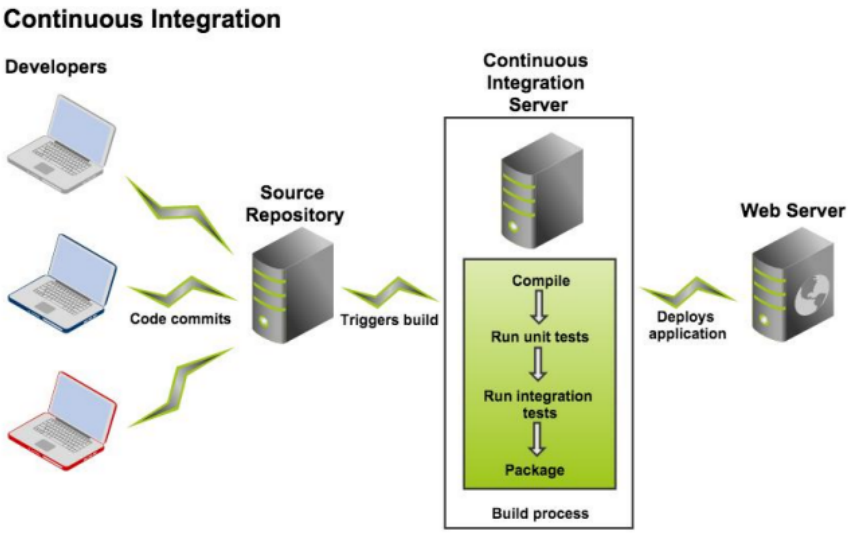
> 这是否意味着程序员需要一直坐着等待构建和运行测试？有没有可能有人自动帮我做这个
>
> - 有的：CI Server

持续集成服务器 Continuous Integration server

- 自动提取最新的 Commit 的代码并完全构建所有资源的外部机器
  - 如果有任何失败，联系您的团队（例如：通过邮件）
  - 确保构建不会被破坏太久
- CI Server 示例
  - Jenkins
  - Bamboo
  - Travis CI

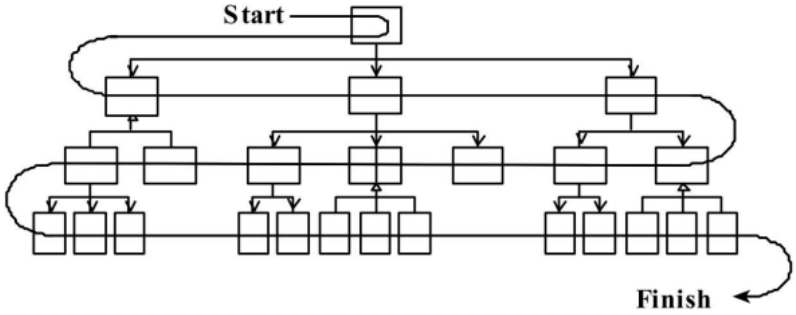
CI 服务器会做些什么

https://confluence.atlassian.com/bamboo/understanding-the-bamboo-ci-server-289277285.html



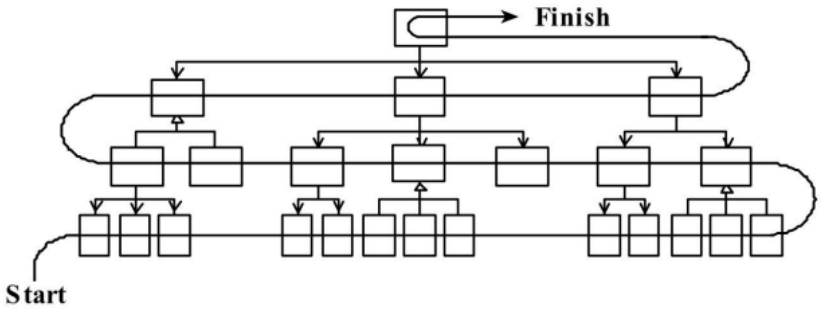
集成测试的种类

自上而下集成 Top-down integration



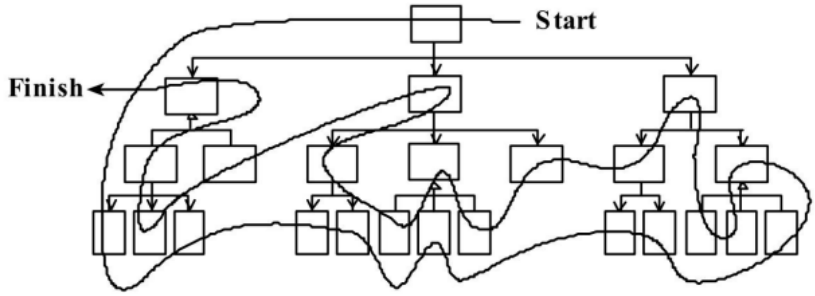
- 从外部 UI 层开始，向内工作
    - 必须编写(大量)存根底层以便 UI 交互
    - 允许推迟艰难的设计/调试决策

自下而上集成 Bottom-up integration



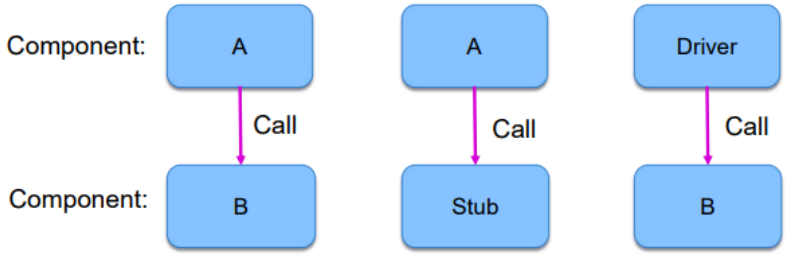
- 从底层数据/逻辑层开始，向外工作
    - 必须编写测试驱动程序来运行这些层
    - 直到很晚才会发现高级/ UI 设计缺陷

“三明治”测试 "Sandwich" integration



- 将顶级 UI 与关键的底层类连接起来
  - 稍后根据需要添加中间层
  - 比自上而下或自下而上更实际

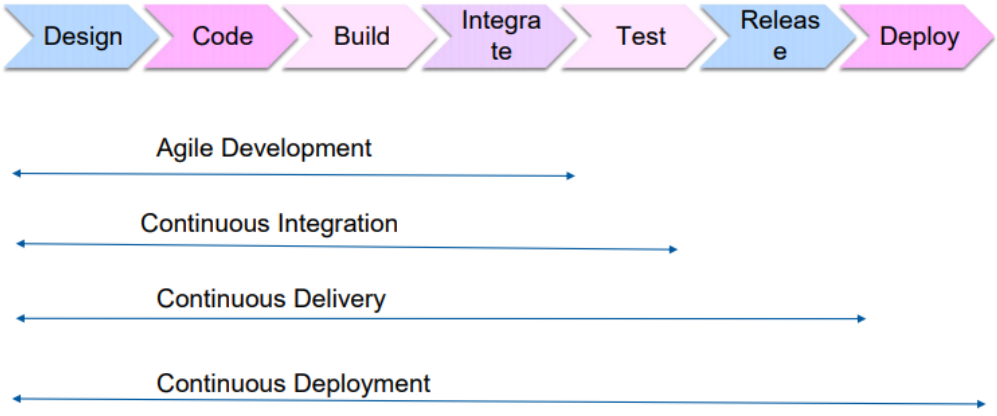
存根 Stub 和驱动 Driver



- 两者都是用来替代缺失的软件和模拟组件之间的接口
  - 创建伪代码
- Stub：被另一个函数调用的假函数
- Driver：调用另一个函数的假函数

概念

持续集成和持续部署 Continuous Deployment

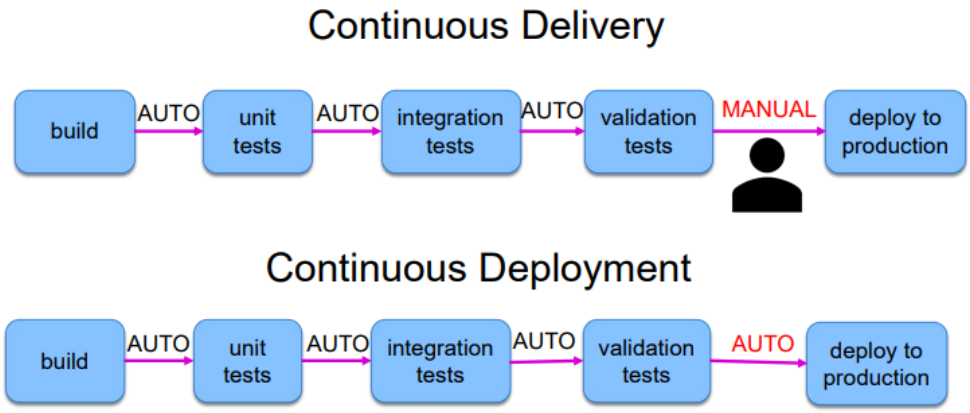


持续交付 Continuous Delivery - CD

- 我们把持续交付称为 CD

我的软件交付哲学的本质是构建软件，使其**始终处于可以投入生产的状态**，我们称之为**持续交付**，因为我们一直在运行一个部署管道来测试该软件是否处于要交付的状态

交付 Delivery 与部署 Deployment



CD != CI

- Continuous Delivery = CI + automated test suite
- 并不是每个更改都是一次发布
  - 手动触发
  - 触发一个核心文件（版本）
  - 为发布版本打上 Tag
- 持续交付的关键是**自动化测试**

> 持续集成和持续交付有什么区别?
>
> - 持续交付需要在发布前进行自动化的测试

Continuous Deployment != CD

- Continuous Deployment = CD + Automatic Deployment
- 通过自动化测试的每个更改都会自动部署到生产环境中
- 部署时间表
  - 当一个特性完成时发布
  - 每天发布

部署 Deployment

部署的策略

- **零宕机部署 Zero-downtime deployment**
  - 部署服务的版本
  - 将数据库迁移到新版本
  - 将服务的版本 2 与版本 1 并行部署
  - 如果版本 2 运行良好，关闭版本 1

    - 部署完成
- **蓝绿部署 Blue-green deployment**
    - 维护生产环境的两个副本（“蓝色”和“绿色”）
    - 将生产 url 映射到蓝色环境
    - 在绿色环境中部署和测试应用程序的任何更改
    - “翻转开关”：将 url 映射到绿色，并从蓝色取消映射
- 优点
    - 不停机/关闭
    - 用户仍然可以使用应用程序没有停机
- 缺点
    - 需要保存 2 份副本
    - 将支持多个副本所需的工作加倍
    - 数据库迁移可能不向后兼容
- 更安全的策略：关闭服务器 -> 迁移代码 -> 部署

持续集成测试的状态 Google

测试范围

- 连续运行 420 万个单独测试
    - 测试在代码提交之前和之后运行
- 1.5 亿次测试执行/天（平均 3500 万个测试/天）
- 分布式使用 bazel 内部版本，IO 到一个大型计算场
- 几乎所有的测试都是自动化的，没有时间进行质量保证
- 13000 多个独立项目团队——全部提交给一个分支
- 驱动谷歌连续交付
- 99% 的测试执行通过

测试文化

- ~10 年的测试文化促进了手工策划的自动化测试
    - 从 2007 年开始进行“马桶测试”和“谷歌测试博客
    - 从 2006 年开始举行 GTAC 会议，分享业界的最佳实践
    - 这是我们新员工培训计划的一部分
- SETI 角色
    - 通常 1-2 名 SETI 工程师/ 8-10 人的团队
    - 开发测试基础设施以支持测试
- 工程师们需要为他们提交的文件编写自动化测试
- 基于模型/自动化测试的有限实验
    - 模糊，UI 通关，突变测试等等
    - 并不是整个测试的很大一部分

预提交测试

- 使用细粒度的依赖关系
- 使用计算资源池
- 避免破坏构建
- 捕获更改的内容并单独进行测试
    - 对 HEAD 进行测试
- 集成了
    - 提交工具—当且仅当测试显示为绿色进行提交
    - 代码评审工具-发布评审结果
- 连续运行 4.2M 测试，提交更改
    - 如果正在被更改的文件存在于测试依赖的传递闭包中，则测试将受到影响。(回归测试选择)
    - 每次测试使用2种不同的 flag 组合(平均)
    - 在分布式后端上并发构建和运行测试
    - 运行容量通常是允许的
- 记录数据库中每个测试的通过/失败结果
    - 每次运行由test + flags + change唯一标识
    - 我们有 2 年的所有测试结果和每次测试发生了什么变化的信息

测试结果分析

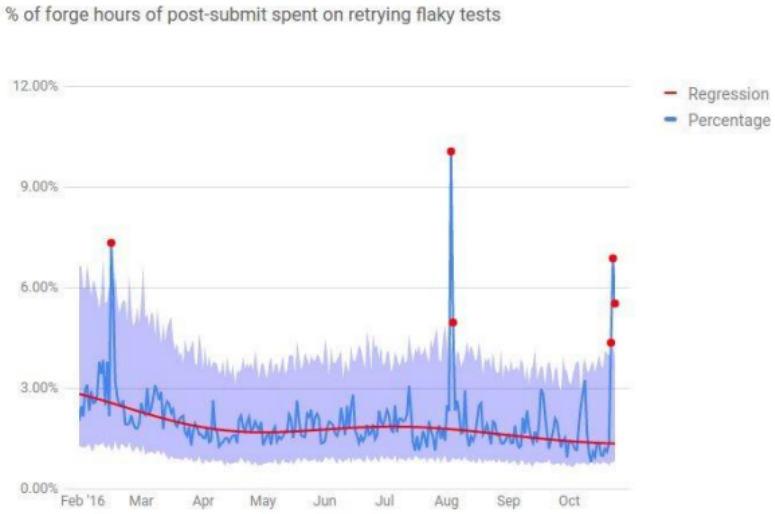
- 大样本试验分析（1 个月）显示：
    - 84%的 pass -> Fail 转换来自“flaky””测试
    - 只有1.23%的测试曾经出现了破损
    - 经常更换的文件更容易出现破损
    - 3名以上的开发人员修改文件更容易导致文件破损
    - 依赖性图中“越近”的变化越容易造成破坏
    - 某些人/自动化更容易造成破损(哎呀!)
    - 某些语言更容易造成破损(抱歉)

CI 问题：Flaky 测试

- Flaky Test（不稳定测试）
    - 对同一代码可能**通过**或**不通过**的测试
- 测试不稳定的来源
    - 并发性
    - 环境/设置问题
    - 不确定或未定义的行为
- 测试不稳定是一个大问题
    - Flakiness 是一种测试，它在同样的代码下可以同时出现通过和不通过的情况

    - 在我们的 4.2M 测试中，几乎 16% 的测试有一定程度的不稳定
    - Flaky 故障经常堵塞和延迟释放
    - 开发人员在提交时忽略了古怪的测试——有时是错误的
    - 我们花费 2% 到 16% 的计算资源来重新运行不稳定的测试
- 缺点
    - 消耗开发人员的调查时间
    - 工期延误
    - 重新运行浪费的计算资源来确认 Flake

重新运行 Flake 所花费的资源百分比



Flakiness 的来源

导致 Flakiness 的因素

- 异步等待
- 并发
- 测试顺序依赖关系
- 资源泄漏
- 网络
- 时间
- IO
- 随机性
- 浮点运算
- 无序集合

Flake 是不可避免的

- 持续 1.5% 的测试执行报告 “flaky” 的结果
- 尽管花了很大的努力来识别和消除 flakiness
    - 目标“搞定”
    - 持续施压于 flake
- 观察到的引入率与修正率大致相同
- 测试系统必须能够处理一定程度的不稳定。最好使开发人员的成本最小化

Flake 测试架构

- 我们重新运行测试失败转换(10次)来验证 flakiness
    - 如果我们观察到一个通过的测试是 flaky 的
    - 为“已知” flaky 的测试保留一个数据库和web UI

3. 回归测试 Regression Testing

介绍

- 什么是回归
    - 软件发生变化
    - 但是做出的修改可能会
        - 提升软件：增加功能/修改 bug
        - 破坏软件：引入新的 bug
    - 我们称这种“破坏性变化”为回归 Regressions
- 什么是回归测试
    - 执行测试，以确保所做的更改不会破坏现有的功能
    - 这意味着重新运行现有测试套件中的测试用例，以确保软件更改不会引入新的故障

如何修复回归错误

- 什么发生了改变
    - 与程序版本有区别吗
    - 使用版本控制系统查看版本之间的变化
        - svn diff, git diff
- 什么改变发生了错误
    - 找出与测试失败相关的更改
    - 使用调试器定位导致失败的更改
        - Debugger
- 修复不正确的改变
- 检查回归 bug 是否被修复
    - 再次运行测试以查看修复是否通过所有测试
    - 添加新的测试以反映更改