目录

[摘要 2](#_Toc11017036)

[Abstract 3](#_Toc11017037)

[第一章 绪论 4](#_Toc11017038)

[1.1 研究背景与意义 4](#_Toc11017039)

[1.2 持续集成国内外研究现状 5](#_Toc11017040)

[1.2.1 持续集成国内外研究现状 5](#_Toc11017041)

[1.3 主要研究内容 7](#_Toc11017042)

[1.4 论文结构安排 7](#_Toc11017043)

[第二章 持续集成相关理论分析 8](#_Toc11017044)

[2.1 软件集成 8](#_Toc11017045)

[2.2 经典软件开发模型的集成方式 9](#_Toc11017046)

[2.2.1 “Big-Bang”集成方式 9](#_Toc11017047)

[2.2.2 “迭代递增”集成方式 11](#_Toc11017048)

[2.2.3 “每日构建”集成方式 13](#_Toc11017049)

[2.2.4 持续集成 13](#_Toc11017050)

[2.3 本章小结 15](#_Toc11017051)

[第三章 持续集成方案设计和系统搭建 16](#_Toc11017052)

[3.1 持续集成系统组成 16](#_Toc11017053)

[3.2 持续集成运行流程 16](#_Toc11017054)

[3.3 Jenkins持续集成系统搭建 17](#_Toc11017055)

[3.4 本章小结 19](#_Toc11017056)

[第四章 基于持续集成在VR领域的实践 20](#_Toc11017057)

[4.1 VR生产环境的自动化配置脚本设计 20](#_Toc11017058)

[4.1.1 设计需求与方案 20](#_Toc11017059)

[4.1.2 设计过程 21](#_Toc11017060)

[4.1.3 结果分析 26](#_Toc11017061)

[4.1.4 总结 30](#_Toc11017062)

[4.2 VR应用测试情况自动报表脚本设计 31](#_Toc11017063)

[4.2.1 设计需求与方案 31](#_Toc11017064)

[4.2.2 设计过程 33](#_Toc11017065)

[4.2.3 结果分析 43](#_Toc11017066)

[4.2.4 总结 46](#_Toc11017067)

[4.3 本章小结 48](#_Toc11017068)

[第五章 总结与展望 49](#_Toc11017069)

[致谢 50](#_Toc11017070)

[参考文献 51](#_Toc11017071)

[附录 52](#_Toc11017072)

[英文文献 52](#_Toc11017073)

[文献翻译 52](#_Toc11017074)

# 摘要

作为新兴的产业，虚拟现实的应用开发目前仍在不断探索当中。为了应对快速发展的虚拟现实应用，本人实习公司采用了敏捷开发的方式进行快速迭代开发，并使用Jenkins作为持续集成系统进行构建和测试。然而在虚拟现实设备生产和应用测试中，环境的配置和测试数据统计仍然是通过人工实现的，不适应于持续集成所提倡的自动化流程。

本文首先对软件集成进行原理分析，描述了集成的概念，进而引出各种经典集成方式。通过分析各集成方式的原理和缺陷，从最基础讲起，重点分析了持续集成的发展情况和具体要求。再通过设计持续集成系统，搭建了Jenkins持续集成服务器。

本文针对公司的需求，分析和研究了生产环境数据库自动化脚本的编写，和代表持续集成反馈机制的虚拟现实测试情况自动报表的脚本两个实践，同时该两个实践都运行在Jenkins服务器环境中。

**关键词：**Jenkins，持续集成，自动化测试，虚拟现实

# 

# Abstract

As an emerging industry, the application development of virtual reality is still being explored. In response to the rapidly evolving virtual reality application, my internship company used agile development for rapid iterative development and built and tested using Jenkins as a continuous integration system. However, in virtual reality device production and application testing, environment configuration and test data statistics are still implemented manually, and are not suitable for the automated processes advocated by continuous integration.

This paper first analyzes the principle of software integration, describes the concept of integration, and then introduces various classic integration methods. By analyzing the principles and shortcomings of each integration method, we focus on the development and specific requirements of continuous integration from the most basic. The Jenkins Continuous Integration Server was built by designing a continuous integration system.

In this paper, based on the needs of the company, the two practices of the production environment database automation script writing and the virtual reality test automatic report script are analyzed and studied, and both practices are run in the Jenkins server environment.

**Key words:** Jenkins, continuous integration, automated testing, virtual reality.

# 绪论

## 研究背景与意义

随着计算机软硬件技术的不断升级，以及互联网的快速发展，软件技术也进行着不断变换，大量软件不断地开发、迭代，软件的开发周期越来越短。自从虚拟现实（Virtual Reality，VR）技术发展以来，基于VR的应用也不断涌现出来。经过了数年的发展，VR的主流从当初的VR盒子（基于手机）到PCVR（基于计算机），再到一体机VR，VR应用也从移动VR应用发展到PCVR应用，再发展到移动VR应用。与VR盒子应用类似的是，一体机VR应用是基于Android系统的，因此大多数手机应用是可以在VR一体机中使用的；不同的是，VR一体机的UI系统是一个虚拟全景系统，而其应用多数是需要支持射线、定位、触控等功能的。得益于如今VR芯片的发展，一体机VR的性能也越来越高，结合各大VR厂商的软件开发工具包（Software Development Kit，SDK），基于一体机VR的应用必将成为VR的主流。

传统的软件构建方法是书序生命周期“瀑布”开发模型，通常都是开发初期制定详细的计划，在计划期间仔细研究、设计最终产品，并且将一切详细资料记录在案。这种方法比较有逻辑性，但是大大增加了开发周期，很难适应如今快速变化的软件开发，无法保证快速开发中的软件质量。这种情况下敏捷开发就诞生了，而持续集成（Continuous Integration，CI）作为敏捷开发的最佳实践被大量开发团队使用，引入自动化测试代替手工测试，测试驱动开发（Test-Driven Development，TDD）成为支撑短周期快速迭代的有效方法。

然而，在一体机VR成为主流前，多数VR厂商将重心都放在了硬件研发和理论研究上，VR应用研发者们只需要考虑Windows系统和显卡的兼容性，之前PC的VR应用研发也都集中在大型的游戏上，开发周期都比较长，“瀑布”开发模型完全可以支撑。对于一体机VR，VR厂商们需要开发自己的VR系统和SDK，各种应用也转变为轻量级应用，开发者们同时需要考虑各种硬件、系统的兼容性问题，敏捷开发成为必然趋势。然而这种转变是很困难的。以本人所实习公司为例，拘泥于传统的开发模式，虽然公司使用了Jenkins对应用自动编译部署并发布，但是作为持续集成的核心，自动化测试并没有落实，研发习惯了简单粗暴的功能编写，而测试则主攻在手动测试上，再加上VR技术的快速发展，自动化测试初期的成本太大，公司只能招入大量的测试人员进行手动测试，增加了人力成本。人员的流动性太大，往往会导致业务了解上的断层，测试者们花大量时间在业务的沟通了解上，过时的测试用例不断累积，新的测试用例却很少创建。对于研发而言，由于没有测试驱动，往往在新功能添加后会引发出旧的功能出现问题，这又增加了测试成本，甚至会造成整个功能块的奔溃。

综合上述问题，需要设计一套完整的自动化测试框架支撑的持续集成方案，一方面解放大量的人工测试，另一方面研发者们在编写功能模块式可以自动测试防止旧功能出现问题。

## 持续集成国内外研究现状

持续集成发展到现在，主要是以美国微软和谷歌等公司带头发展，从概念的提出以到开源工具的使用，国外开发者社区都比国内相对活跃，所以研究国外持续集成的发展状况，并对比国内发展现状以及公司目前使用持续集成的情况是非常必要的。而作为持续集成的核心部分，自动化测试也需要有所体现。由此，本章将对持续集成和自动化测试的发展现状进行研究。

### 持续集成国内外研究现状

自上世纪九十年代以来，软件开发大师Kent Beck一直致力于研究高效的软件开发模式，1996年，他通过编写《Extrme Programming Explained: Embrace Change》一书提出了极限编程（ExtremeProgramming，XP）的概念，该书总结了XP的12个最佳实践，持集成这个概念最初就来源于其中一个最佳实践。然而持续集成实践在当时被当做是XP的一部分，由于XP中的许多实践不为业界所接受，持续集成也未能很好的发展。

2000年，同为XP思想发起者之一的软件大师Martin Fowler专门以《Continuous Integration》（持续集成）为题发布了一篇著名的文章，该文章以他所在公司ThoughtWorks的项目持续集成实践为基础介绍了持续集成的价值。他认为持续集成并不受限于XP的开发模式，其价值在于让项目团队高效地开发出一致的软件。持续集成逐渐得到人们的认可，但是在实际项目中应用还需要大量的研究和时间。

2006年3月，Martin Fowler正式对持续集成实践作出了描述，这里结合个人理解对其进行描述：

（1）维护源码仓库。在软件工程当中，存在大量的源码和其他依赖文件，由于项目修改编译的频率越来越高，大量的文件需要可以对其历史记录和相应的修改者进行回溯，保证在产生错误后能够迅速地上报相应的错误代码和相关文件。由此，需要相应的工具作为支撑，如配置管理工具、源码管理工具、版本控制工具、仓库工具。

（2）自动化构建。项目构建即将代码编译或解释成一个应用程序，整个过程需要对文件、数据库等等进行操作，这些操作往往是复杂且固定的，由此可以使用将这些操作进行自动化。如Java的自动化构建工具Maven和Linux中的自动化构建工具Make等等都可以实现类似的操作。

（3）令构建自测。每次构建完成后，为了保证构建是成功的，需要对构建完成的应用进行基础测试，自动化测试在这时是十分重要的。自测后保证构建好的应用可以正常运行，再交付给相关测试人员进行具体的功能测试。

（4）每人每天向主干提交代码。集成的目的之一是项目沟通，通过集成，项目开发人员在每次提交代码后会产生相应的代码修改记录，其他项目人员可以通过查看代码记录找到相应的负责人员，由此更好地进行代码和功能的交流。当项目存在错误时，尽早、少量的提交变更可以更快地定位到产生错误的变更，降低代码修复或回滚的风险和成本。

（5）每次代码提交前需在集成环境中进行一次构建。每次提交后构建再结合自测可以保证每次的提交是有效的，如无效则自动回退之前的版本。

随着持续集成的不断实践，越来越多的研发者们开始使用持续集成的模式，并基于持续集成开发了各种工具，如CruiseControl、Hudson、LuntBuild、Bamboo等等。如今，开源持续集成工具Jenkins（原Hudson）因其免费、跨平台、丰富的插件成为最受欢迎持续集成工具。在持续集成之后，持续交付和持续部署的概念也逐渐被提出来，Jenkins2.0也由此诞生，其精髓为Pipeline as Code，即任何发布流程都可以表述为一段脚本，实现由持续集成到持续交付的转变。

作为软件开发巨头，Google的持续集成技术是比较领先的。《How Google Tests Software》（Google软件测试之道）一书中详细描述了Google在如今快速变化的互联网时代，为了保证软件质量所做的自动化测试与持续集成的研究与发展，更加证实了持续集成在快速开发中保证质量的重要地位。

2013年，Paas服务提供商dotCloud发布了一个开源项目Docker，它是一个基于LXC容器技术的高级容器引擎，之后大量开发者加入到了这个开源项目当中，开发者们在不断实践中将Docker应用到持续集成当中，使持续集成环境变得更加易于部署和移植。之后，大量的Docker镜像可以被使用，其中就包含Jenkins持续集成环境。

目前国内各大厂都有各自的持续集成方案，阿里、腾讯、华为这几家云服务也引入了Docker技术，华为工程师们也在《Docker进阶与实战》中描述了一些基于Docker的持续集成实践

## 主要研究内容

（1）针对一般软件开发过程的继承问题，分析传统软件开发模式集成方式的不足，引入持续集成思想，对持续集成进行深入的分析与研究。

（2）基于持续集成的理论，研究了持续集成环境的主要结构，搭建了Jenkins持续集成服务器。

（3）研究在VR生产环境中自动配置数据库的脚本，主要涉及到batch语言的语法结构，mysql的安装与表的创建，环境变量的配置。

（4）研究公司VR测试情况的自动报表脚本，主要涉及selenium自动化测试工具，python的图表绘制和对excel表格文件的操作，然后研究了python发送邮件，最后将脚本配置到jenkins服务器上定时运行。

## 论文结构安排

第一章：绪论。主要介绍了本课题的研究背景与意义以及国内外当前的研究现状，之后介绍了本文的主要工作内容和主要结构。

第二章：对持续集成相关理论进行分析。分析和研究了经典软件开发过程中的持续集成方式，重点分析了持续集成思想的发展过程和持续集成的特点，最后分析和总结了持续集成在软件开发过程中的应用价值。

第三章：介绍基于Jenkins的持续集成方案。研究了持续集成环境的主要结构，搭建了Jenkins持续集成服务器

第四章：研究两个基于持续集成在VR领域的实践。其中研究在VR生产环境中自动配置数据库的脚本，主要涉及到batch脚本语法，mysql的安装与表的创建，环境变量的配置。研究公司VR测试情况的自动报表脚本，主要涉及selenium自动化测试工具，python处理数据和发送邮件。最后将脚本配置到jenkins服务器上定时运行。

第五章：对持续集成在VR领域的总结与展望。总结已经完成的工作，并对下一步的工作作出展望。

# 持续集成相关理论分析

## 软件集成

在人类社会发展以来，集成始终贯穿着人类的各个领域，从社会的阶层统治到制造业中的生产流水线，一个庞大的产物总是由各个部分分工合作组成的，不同的领域也赋予了集成不同的定义，但是他们本质上都是一样的，这种模式是目前人类发展至今最有效的组成方式了。它保证了内部的稳定协调，也保证了外部的完整和稳固。

如今已经发展到了信息化社会，不同于传统的工业工程，软件工程是构建于虚拟的，我们可以把它看成虚无缥缈的，但也确实存在的，它由信息构成。当然由信息构成的虚拟的东西并不是只存在于现在，过去的精神文化如小说等就是如此，人们在创作时虽然说是天马行空的，但是一个好的作品是在一个完整的流程创作出来的，它们都少不了构造的过程，我们也可以把它理解成为一个虚拟的工程构造。软件开发也是如此，但是得益于信息的无处不在和信息社会的迅速发展，它们比之传统和精神的工程有了更高的要求和更快的发展，它们的构造流程也更加严格，软件集成由此发展而来。

一个工程的完整和有效是评估工程质量的重要标准，软件集成保证了工程内部的完整统一，也实现了交付前的质量测试。简单的软件可能只是实现需求的功能，而一个软件工程包含了必要的功能代码、调试代码和测试代码，它们通过模块化构成了一个完整的软件产品，这就是软件集成。

但是在软件开发行业发展初期，由于模块众多，只要其中一个模块出现错误，那么整个软件工程就无法正常运行，并且定位出现错误的模块也是非常耗时耗力的，即使问题已经发现，但是如何去修改该模块而不会导致其他功能出现问题也是需要思考的。所以，出奇的软件集成是是非常容易出现问题的，从问题来源来看，主要有以下几类：

（1）模块功能问题。基础模块开发是由开发人员编写的，编写过程难免会产生一些疏忽，从而引发一些Bug，这些Bug可能只是一些简单的语法或格式错误，也可能是由于开发人员对于功能理解错误引起的功能缺失或错误。当然在修复已有缺陷时仍然可能引入新的Bug，对于人来说，这些都是合情合理、无可避免的。

（2）集成时的模块组装问题。在开发单一模块时往往很难发现它与其他模块一起构建时的问题，即使它本身没有任何问题。这些问题可能是由于模块接口产生的，也可能是集成本身的结构存在问题。

（3）合并集成问题。模块开发往往是并行的，当多个模块开发完成后再将其合并，合并前产品本来是没有任何问题的，各个模块的改动也是没有问题的，但是合并后各个改动可能会产生冲突，从而引发一些问题。

（4）操作和设置错误。如不同的开发人员使用的开发环境或版本不统一引发的问题，不同的测试人员使用的测试环境同导致测试结果不一。再如在提交新创建的源文件时未将其添加到版本控制管理工具当中，其他开发人员无法拉取该源文件，导致运行失败。这些错误主要还是体现在开发沟通上。

开发人员们在不断实践中，发现并试图解决这些问题，软件集成方式也在不断改进。软件集成方式往往和软件开发模型密不可分，在软件开发当中存在几个经典的软件开发模型，它们的集成方法大同小异，对比其中各个集成方式的缺陷和优点，我们可以试图得出持续集成是如何发展的。

## 经典软件开发模型的集成方式

### “Big-Bang”集成方式

在软件开发行业发展过程中，为了适应不断增长的软件需求，人们在不断实践中总结出了一些软件生命周期模型，这些模型在软件开发过程起到了很大作用，即使它们存在一些问题。

1970年，Winston Royce在论文《管理大型软件系统开发》（Managing the Development of Larger Software Systems）中，基于软件生命周期提出了著名的“瀑布模型”（water-fall model）。该模型将软件生命周期按固定顺序分为六个基本工程活动，相邻的活动之接是双向的，它们自上而下相互衔接，形如瀑布逐级而下，如图所示。

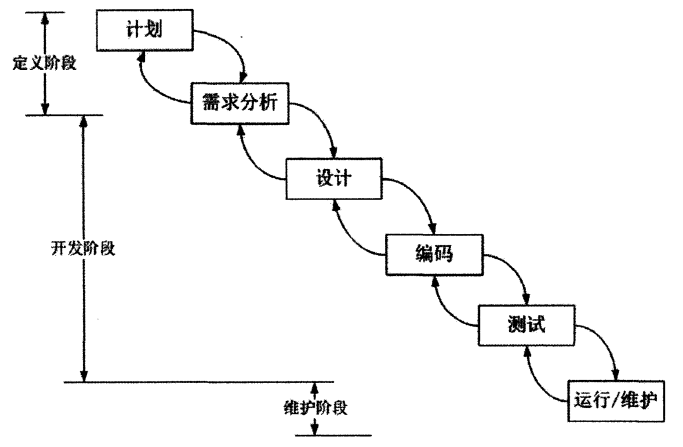


图2-1 瀑布模型

如图，瀑布模型将一个软件生命周期整体上分为三个阶段：定义阶段，开发阶段和维护阶段。定义阶段包含计划和需求分析工作；开发阶段包含设计、编码个测试工作；维护阶段包含运维工作。对于相邻的工作，它们之接的关系并不是单向的，下一阶段需要随时向上一阶段对产生的相关问题进行反馈，以支持项目某一阶段的更改。

根据瀑布模型的相关理论，一种叫“Big Bang”集成的软件集成方式应运而生。从瀑布模型来看，相邻的两个阶段是相关的，且后一阶段的启动必须依赖于前一阶段的产生，但是不相邻的两个阶段直接并未有所关联，这就造成在前一阶段进行规划时无法兼顾其他阶段，当所有阶段集成到一起时，某一阶段可能不符合预期或产生错误。而当对该阶段重新规划时，只能一级一级地向上反馈。由此可以看出，“Big Bang”集成方式意在当项目的进行到最后一个模块时再进行集成，当前面几个模块产生错误时只有在最后集成时发现，需要在最后集成时进行大量修改。在开发初期可能这种集成方式不会出现问题，但是到了软件快要要付时如果出现问题，那么无法很好精确预估修复时间，且各模块在此时已经趋于复杂化，严重地话可能需要对大量模块进行修改，增加时间成本，由此这种集成方式无法满足越来越复杂的软件工程了。

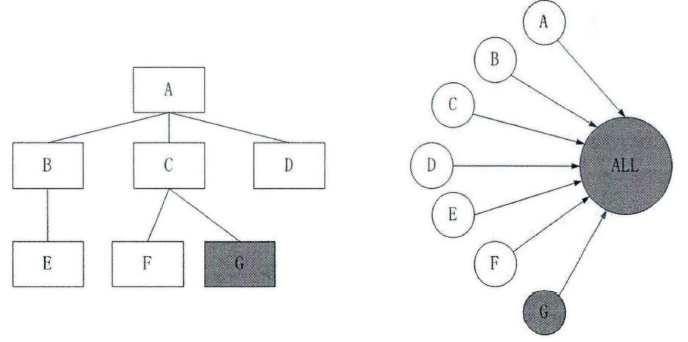


图2-2 Big-Bang 集成方式

### “迭代递增”集成方式

当软件需求在不断改变时，传统的“瀑布模型”无法有效快速地根据需求对软件工程进行修改。由于软件生命周期在不断变化，需要将整个周期按照某种方式划分为几个周期。“迭代递增”就是基于这种情况诞生的，它在Rational公司的Rational统一过程（Rational Unified Process，简称RUP）中提及。

“迭代递增”即存在迭代和递增两个主要方法，每一个迭代都是一个迭代周期，下次迭代都是在上一次迭代趋于稳定后逐渐复杂化，即称为递增。单个迭代周期可以看做是一个瀑布模型，它基本包含了瀑布模型的各个工作，如需求分析、设计、编码、测试工作。当一个迭代趋于结束后，首先要对其进行整体评估分析，确认该迭代周期已达到目标预期，可以结束。之后根据新的客户需求，对下一个迭代周期进行新的目标制定，做整体的规划调整。如图所示。

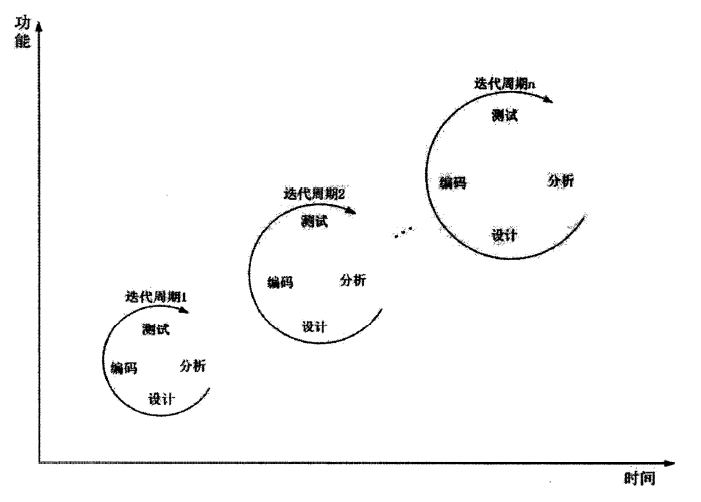


图2-3 迭代递增过程模型

虽然“迭代递增”解决了在客户需求不断改变下的周期问题，但是这种集成方法在不断实践下暴露出两个主要问题，即“迭而不增”和“增而不迭”。

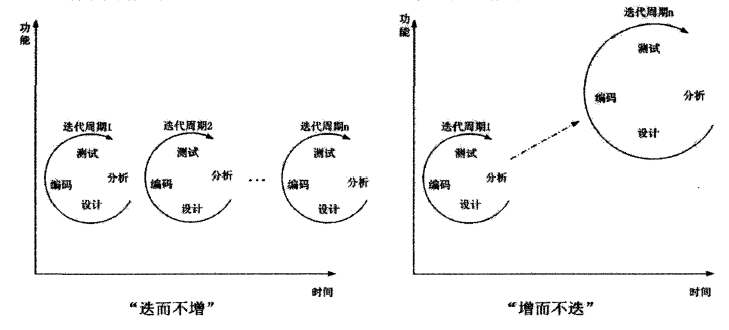


图2-4 迭代递增集成方式的两个问题

如图所示，“迭代递增”主要有两个需要量化的量，即迭代间隔和任务量，如何对它们进行限定成为该集成方式的一个主要问题。当迭代间隔较短时，两个迭代周期的任务量可能不会发生较大改变，仅仅是重复迭代；当迭代间隔较长时，两个迭代周期的任务量差距可能非常大，此时的集成方式其实回归到了“Big Bang”集成。微软公司提倡的“每日构建”集成方式很好的解决了这两个问题。

### “每日构建”集成方式

作为一家世界顶级的软件开发公司，微软无疑拥有者丰富的软件开发经验，他们的工程师在不断实践中根据经验总结出了很多软件开发的方法，其中就包含一套微软解决方案框架（Microsoft Solution Framework，简称MSF），其中就提到了基于里程碑的项目生命周期管理框架。

MSF过程模型中提到了著名的“每日构建”（Daily Build）软件集成方式。这种集成方式将迭代间隔固定唯一天，将迭代增量固定为一天的增量。即在项目组每天完成代码提交后，需要立即进行项目构建，并对构建好的程序进行可执行测试，这种测试被称作“生成验证测试”（Build Verification Test，简称BVT）。为了实现每日构建，微软项目组在项目开除其会通过需求分析，创建一个用于编译、生成、测试的框架，该框架可以实现对所有开发人员每日提交的代码进行合并、构建，最后再进行一次BVT测试。当测试中发现问题后，在下一次迭代周期进行修改，如此往复，保证了软件项目的持续稳定。

### 持续集成

互联网行业的迅速发展使得软件行业的规模变得越来越大，也越来越复杂，各种大小型软件开发相关企业不断出现，不同规模的公司也对持续集成有了不同的要求。由于软件行业的用户量及其庞大，各种需求都是迅速变化且不可预测的，为了应对种种情况，如今的敏捷开发方法被各公司所推崇。本人所在公司即使用这种开发方式。

敏捷开发推崇缩短软件版本开发的周期，简化开发流程，避免多次重复迭代引起的成本问题，持续集成能够很好的应对这种开发方式。持续集成更加注重自动化流程，以保证在一天之内可以有序无误且快速地构建并测试多个软件版本。对比前面的“每日构建”，持续集成并不局限于每天一次构建，“每日”仅仅是它的最低要求。一个完全自动化的持续集成流程包含自动化的构建和自动化测试以及其他项目管理流程，它可以实现每天不止一次的重复构建。从表面上看，虽然流程实现了自动化，多次构建仍然可能加大了开销，但是从长远来看，这种做法其实是节约成本的，并且保证了软件的持续稳定性。

开发人员通过不断实践发现，集成的工作量并不是累加的，而是与两次集成时间间隔的平方近似成正相关。我们假设每天集成一次需要的工作量为C，那么一周（五天）如果采用每天集成一周集成的工作量为5C，而一周集成一次的工作量为25C。从集成工作量来看，缩短集成间隔能够有效地减少总体集成的工作量。当然，对于工作量的计算是要考虑到项目计划、开发人员的感受等等，恰当的集成间隔会使开发人员感到舒适，且不会影响到整个项目的进程。

Martin Fowler提倡在任何可执行的软件开发过程中使用持续集成，在项目不断自动构建的同时保证测试数据的完整，也保证项目是可以溯源的，提高在每次修改的可行性，实现这些目标的持续集成需要包含以下几个环节：

（1）代码管理。由于一个项目的开发人员并不只有一个，为了实现项目的统一管理，且所有参与人员都可以获取、修改、运行他人的提交。当代码未进行统一管理时，开发人员需要随时进行代码的沟通，防止双方所开发部分不会产生冲突，这就需要大量花费时间去进行沟通。所以，首先需要一款代码管理工具将所有参与者的开发代码统一管理在一个地方，如服务器等，这样参与开发人员可以随时查看其他人所提交的代码；当然，代码的提交也要考虑其可行性，此时需要具有相关权限的开发人员对提交的代码进行评审，再决定该部分代码是否能合并到整个代码库中；除此之外，代码库中应不仅仅含有当前版本的代码文件，需要对项目整个过程的大多数版本进行备份，保证更改错误后可以快速回溯代码；当然，为了能保证代码库中的代码在所有参与人员的主机上是可以执行的，需要对整个项目的所有依赖进行同步，可以通过一些配置文件达到此目的。

（2）自动化构建。自动化构建要求项目代码能够通过简单的命令将其自动转化为可执行的应用程序。对于小规模的工程，他的源文件和依赖较少，几条命令即可完成构建；但是对于大型企业项目，源文件构造十分复杂，整体成树状结构，开发人员需要通过文件检索去寻找特定的文件，显然一个一个地对文件进行构建是不切实际的。那么就需要一款自动化构建工具，它能够实现将所有源文件和依赖按照一定的编译顺序，自动有序地编译出来，并且将相应的编译信息输出以供分析。如Java的自动化构建工具Maven和Linux中的自动化构建工具Make等等都可以实现类似的操作。

（3）自动化测试。自动化测试是极限编程（XP）中所提到的实践之一。XP中将测试分为两类：功能测试和单元测试。其中单元测试是由软件开发人员编写的用来测试代码的基本编写单元，这种测试基本都是基于脚本的，基本上都可以实现自动化测试；功能测试的范围较大，多数功能测试都是从客户需求和产品设计上考虑的，且都是基于用例驱动的。功能测试大多数都是依赖手动测试的，因为很多和功能设计需要从用户角度去考虑，这些都是感官上体现的。测试人员在不断实践中发现，有大量的功能测试在每个版本的操作都是重复的，是可以通过脚本自动化的。大多数这些功能都是项目中已经稳定不变的。每次构建完成后，为了保证构建是成功的，需要对构建完成的应用进行这些基础测试，自动化测试在这时是十分重要的。自测后保证构建好的应用可以正常运行，再交付给相关测试人员进行其他无法自动化的功能测试。

（4）主创建。开发人员在提交代码前需要要确保该代码是可以在整个项目中构建的，这就要求开发人员每天在提交代码前将代码服务器中的代码更新到本地，在进行本地构建。然而对于大型的项目，这种将所有代码都更新的方式无疑是浪费时间的，代码的保密性也不是很好。主创建提倡开发人员将自己所负责的模块代码在每次开发前同步到本地，在本地修改后提交到代码库中，并在服务器上进行构建。如果构建成功，那么代码提交是成功的。如果构建失败，代码提交失败，代码库自动回退提交前的状态。为了防止构建冲突，主创建必须保证每个开发人员只能在不同时间进行构建，即同一时间只会给一位开发人员发放构建令牌。

## 本章小结

本章首先对软件集成进行原理分析，描述了集成的概念，进而引出各种经典集成方式。通过分析各集成方式的原理和缺陷，从最基础讲起，重点分析了持续集成的发展情况和具体要求。

# 持续集成方案设计和系统搭建

## 持续集成系统组成

持续集成系统主要由版本控制库、持续集成服务器、构建工具、反馈机制和开发人员构成，他们相互配合并相互依赖，最终完成从开发版本到自动化构建，然后进行自动化测试，最后返回反馈结果。各个环节主要描述如下：

（1）开发人员。开发人员负责代码的最初生成，在开发人员进行需求确认和框架设计后，他会在本地进行一次构建，构建成功后会通过版本控制工具将代码提交到版本控制库中。

（2）版本控制库。版本控制库通过能够通过一个代码管理工具来管理项目的源码和其他依赖资源，所有的源码都集中在存储在代码管理库中。这样版本控制库能够根据版本提交的历史来管理各个版本的代码，为所有开发人员提供统一的源代码版本和环境配置。相关工具如SVN代码管理服务器，Git代码管理工具和Gitlab代码管理服务器等等。

（3）构建工具。构建工具常常用来编译、解释、执行脚本等等。常用的构建工具有Maven、Ant、Make等。构建工具可以通过简单的命令按照一定的流程自动化地执行构建过程。

（4）反馈机制。持续集成的一个很大优势是能够将构建结果进行反馈，及时提醒开发者应对反馈结果。一般来说反馈方式使用的是日志、邮件、报表等等，当构建过程出现问题时，持续集成工具能够及时地通过这些方式通知开发者，开发者可以对详细的反馈信息进行分析，在对项目进行调试和开发，从而及时地解决问题，保证构建的持续稳定。

（5）持续集成服务器。持续集成服务器使用持续集成工具搭建在一个独立的环境中，它可以结合其他项目管理工具，通过配置实现持续集成的整个自动化流程。持续集成服务器类似一个定时装置，可以对需要构建的项目进行定时构建、测试、反馈等。一般来说，目前使用广泛的是Jenkins持续集成工具。

## 持续集成运行流程

持续集成的整个流程的开端是开发人员，通过反馈机制构成一个闭环。在一名开发人员完成自己所属模块的开发工作后，首先需要在本地执行一次构建，构建成功后，将完成的代码通过版本控制公会局提交到代码管理库中。在一个分工明确的项目团队中，开发人员根据自己所负责的模块承担着不同的任务，他们都可以提交自己的变更，通过审批后即可上传到代码库中。开发人员在每次提交变更后都会触发持续集成的整个自动化流程。持续集成的具体过程如下：

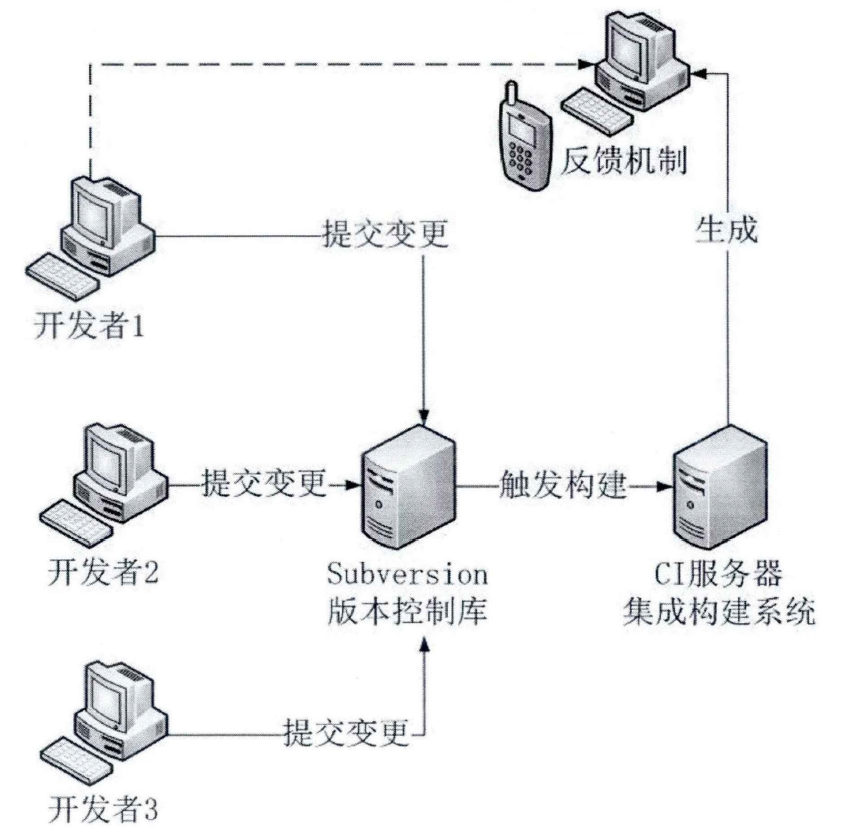


图 3-1 持续集成系统组成部分和运作流程

## Jenkins持续集成系统搭建

1. 使用VMware虚拟机安装Linux发行版Deepin。
2. 安装Docker

（1）首先添加使用HTTPS传输的软件包以及CA证书

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl python-software-properties software-properties-common

（2）添加软件源的GPG密钥

curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add –

（3）添加docker软件源

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian jessie stable"

（4）安装docker-ce（docker社区版）

sudo apt-get update

sudo apt-get install docker-ce

1. 安装Jenkisn

（1）使用docker安装Jenkins

docker run \

-d \

-p 8080:8080 \

-p 50000:50000 \

-v jenkins:/var/jenkins\_home \

-v /etc/localtime:/etc/localtime \

--name jenkins

jenkinsci/blueocean

如图为各参数含义解释

表3-1 Docker安装Jenkins命令参数解释

|  |  |
| --- | --- |
| -d | 后台运行镜像 |
| -p 8080:8080 | 镜像的8080端口映射服务器的8080端口 |
| -p 50000:50000 | 镜像的50000端口映射服务器的50000端口 |
| -v jenkins-data:/var/jenkins\_home | Jenkins工作目录为docker容器中的/var/Jenkins\_home |
| -v /etc/localtime:/etc/localtime | 同步容器与服务器的时间设置 |
| --name jenkins | 容器命名 |
| jenkinsci/blueocean |  |

（2）在浏览器输入<http://localhost:8080>进入Jenkins登录页面

（3）docker exec -it jenkins bash进入访问jenkins容器，访问/var/jenkins\_home/secrets/initialAdminPassword获取密码

（4）登陆后安装插件完成Jenkins环境搭建

## 本章小结

本章主要研究了持续集成系统的设计组成。首先研究和分析了持续集成系统的组成结构，然后描述了持续集成的具体流程，最后选区Jenkins作为持续集成服务器，并使用docker搭建Jenkins持续集成服务器。

# 基于持续集成在VR领域的实践

## VR生产环境的自动化配置脚本设计

在VR设备生产过程中需要对它的生产校准数据进行保存，涉及到了持续集成中环境配置的问题，这里采用脚本的方式实现自动化环境配置部署。

### 设计需求与方案

公司E3系列VR设备在生产期间需要通过一系列生产工具，对VR设备进行烧录、校准、功能测试、SN号写入等等操作，而这些操作的结果数据需要存储在MySQL数据库中。针对生产环境的各种要求做出如下应对：

（1）环境配置方式简单。设备生产代工厂的工人并不具有任何计算机的相关知识，他们只知道通过相关的说明按照步骤使用生产工具进行生产，而即使有公司人员去进行环境部署，也需要对多台设备进行操作，部署后还要进行环境的稳定测试，工作量巨大，且不易维护。通过脚本将配置步骤有序地实现，只需要通过简单的说明，工厂相关人员即可进行快速地环境配置，且脚本测试只需要进行一次，极大地减少了环境配置的工作量，也保证了环境运行的稳定性。

（2）脚本依赖少。工厂的基础计算机环境都是进行（Win7）初始化过的，没有安装任何编译软件或者相关执行组件，这就需要环境配置脚本必须保证能够在原生的环境中运行。通过分析，在工厂的环境中可执行的脚本是批处理脚本，后缀为.bat，使用batch语言编写。

（3）对各种可能失败的结果进行提示。脚本运行过程必然会因为各种各样的原因出现错误，如旧的数据库进程对安装数据的影响等等，这需要通过大量测试发现相关的问题，并在脚本中做出判断，并作出相关的提示供脚本使用人员进行操作。

（4）易于维护。生产环节的改变以及其他要求的改变会导致数据库环境做出改变，需要在脚本中进行相关修改，这要求脚本中的配置是易于寻找方便修改的。

总结来说，数据库环境的搭建脚本实现方案为：通过batch语言编写自动化脚本，脚本要求最好是一键式的，依赖少，易于维护，并有良好的操作提示。

### 设计过程

数据库安装使用的是MySQL的解压版，再使用脚本实现MySQL的安装，脚本分为安装卸载两个脚本。通过分析脚本主要分为如下模块：

1. 初始化。包含目录、MySQL初始化。
2. 配置参数。该部分参数即在维护中可修改的参数。
3. MySQL配置写入。
4. 依赖安装
5. 数据库安装
6. 环境变量配置
7. 卸载环境

详细步骤和实现方法如下所述：

1. 初始化和卸载环境

初始化是为了保证脚本运行顺利，首先要读取脚本所在目录，以在之后整个脚本中都是基于该目录的。Batch语言中使用%~dp0即可读取脚本所在目录。

在脚本运行前必须保证数据库的安装解压文件是纯净的，本质是将MySQL进行卸载，所以这里将初始化和卸载放在一起。在安装脚本中通过call命令来调用卸载脚本即可实现初始化。

卸载步骤如下：

1. net stop mysql停止MySQL服务
2. sc delete mysql卸载MySQL主要程序
3. taskkill杀掉mysqld.exe进程
4. del删除MySQL的配置文件my.ini
5. rd /s /d data删除MySQL的数据文件

2. 配置参数

为了方便修改脚本参数，将脚本中的常用参数通过set命令来设置，主要参数有：

1. 数据库服务器端口，一般为3306.
2. 数据库密码。
3. 数据库名。
4. 数据库表明
5. 全局变量主表路径。

3. MySQL配置写入

MySQL安装配置主要是通过my.ini文件设置的，通过如echo [mysql] >> my.ini可以将[mysql]写入到文本中，且不会覆盖之前的文本，而使用>是会覆盖的。需要写入的配置主要有：

表4-1 MySQL安装配置

|  |  |  |
| --- | --- | --- |
| 参数 | 解释 | 值 |
| default-character-set | MySQL客户端默认字符集 | utf8 |
| port | MySQL服务运行端口号 | 3306 |
| basedir | MySQL程序存放路径，常用来存放MySQL启动、配置文件、日志等 |  |
| datadir | MySQL数据库存放文件路径 |  |
| max\_connection | MySQL允许的最大连接进程数 | 200 |
| character-server-set | 数据库默认字符集 | utf8 |
| default-storage-engine | MySQL存储引擎 | INNODB |

4. 依赖安装

MySQL程序运行需要依赖Microsoft Visual C++ 2013 Redistributable(x64)，这是一个微软的运行库，该库支持命令行安装，下载vcredist\_x64.exe后，使用命令vcredist\_x64.exe /h可以查看命令行安装方式，如下图

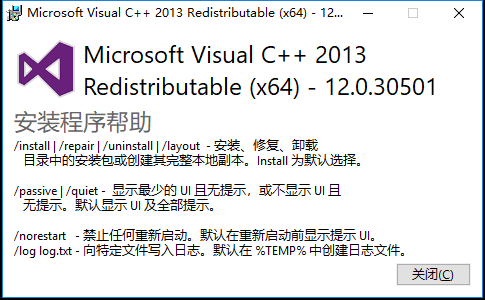


图4-1 vcredist\_x64.exe命令行安装

根据提示，为了快速安装该库采用静默安装，即使用vcredist\_x64.exe /passive命令。

5. 数据库安装

在将数据库配置文件和依赖库安装好后，就可以安装数据库了，使用命令mysqld –initialize和mysqld install即可安装实现数据库安装。但是安装好后的数据库是需要密码登录，为了能够快速进行数据库的各种操作，需要将数据库设置成无密码登录，主要操作如下：

（1）停止数据库服务。使用net stop mysql 2> nul停止数据库服务，由于net stop会在命令行窗口打印输出一些没必要的信息，这里使用2> nul隐藏输出。

（2）修改配置文件。将skp-grant-tables写入到my.ini文件中可以实现无密码登录。

（3）杀掉MySQL命令行工具。使用taskkill命令可以强制杀掉mysqld.exe。

数据库安装成功后，需要判断MySQL是否安装成功，判断依据是MySQL服务是否能够重新启动。由于判断依据不是一条特定的语句，而是服务启动结果，不能使用ifelse语句去进行判断，需要采用管道命令实现，主要使用的管道命令符是&&和||。&&表示前面语句执行成功运行后面语句，||相反为执行失败运行后面的语句。判断语句为net start mysql && goto USAGE || goto FAIL，即当MySQL服务成功启动则跳转到USAGE语句模块，启动失败跳转到FAIL语句模块，各模块描述如下：

（1）:USAGE。该模块为MySQL服务成功启动即成功安装后的操作。由于未进行数据库环境变量设置，数据库安装成功后必须进入到MySQL文件目录中的bin文件夹中才能够执行相关SQL语句和MySQL命令。在脚本中使用如mysql -e “use mysql”语句执行SQL语句，在此时就可以执行SQL语句了。由于之前是采用免密登录MySQL的，首先要设置root用户的密码，需要执行命令

mysql -e "use mysql"

mysql -e "update mysql.user set authentication\_string=password(%password%) where user='root';"

在更新root密码之后，需要使用flush命令刷新MySQL的系统权限相关表，否则会现拒绝访问，即执行

mysql -e "flush privileges;

然而，由于此时root用户已经有权限限制，后续建表将无法继续进行，相较之前而言，root用户的密码是可知的，只需要将root密码写入到my.ini配置文件中，最后通过执行命令

mysql --connect-expired-password -e "SET PASSWORD = PASSWORD('%password%');"

通过MySQL安全策略登录进去，之后即可执行SQL语句创建表和字段了。

到目前为止，关于MySQL的权限设置和表的相关创建都已完成，无需再对数据库进行其他读写操作，通过my.ini中写入root密码的操作并不是安全的。这里需要在配置文件中将密码设置删掉，batch语句无法指定删除文本中的某一行，只能通过重新写入配置对my.ini作出更改。

（2）:FAIL。该模块为MySQL服务启动失败后的操作。服务启动失败即MySQL未安装成功，此时需对可能引发失败的原因做出提示，提醒使用者重新尝试运行脚本，且为了保证安装程序仍然纯净，要使用call调用一次卸载脚本。

6. 环境变量配置

通过前面的步骤执行下去，MySQL数据库已经基本安装完成，但是只能在MySQL的bin文件夹中运行，所以需要将bin添加到Windows的环境变量中，即可在全局环境中运行MySQL数据库服务。

一般手动添加环境变量只需要修改Windows中的path即可，然而脚本中并非如此。使用脚本添加path需要三步：

1. 获取原来的path；
2. 将要填加的path和原来的path通过分号合并在一起；
3. 将合并后的path赋值给环境变量中的path；

总结来说主要需要获取原来的path，然后再替换path。在batch脚本中可以直接通过%Path%来获取path，再用过setx命令重新设置path。但是通过实践发现，通过这种方式获取到的path是包含了用户和系统两个path的，而setx命令只对用户变量做出修改。在这种情况下，修改环境变量的结果是将用户path、系统变量path和添加的的path都作为用户变量的path，修改后的用户变量path将变得十分臃肿，严重破坏了之前的环境变量的值。且path是有长度限制的，如果path足够长，那么限制长度后新添加的path将不会被写入。

通过研究发现，Windows中的系统和应用程序的设置信息都存储在注册表当中，注册表其实是一种windows特有的数据库，类似传统的数据库，注册表中有字段名、类型和相应的值，而path为字段名，修改path需要修改改path对应的值。相应的，系统环境变量位于HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\SessionManager\Environment\，用户环境变量位于HKEY\_CURRENT\_USER\Environment，这里只修改用户环境变量，如图为Windows注册表编辑器查看情况

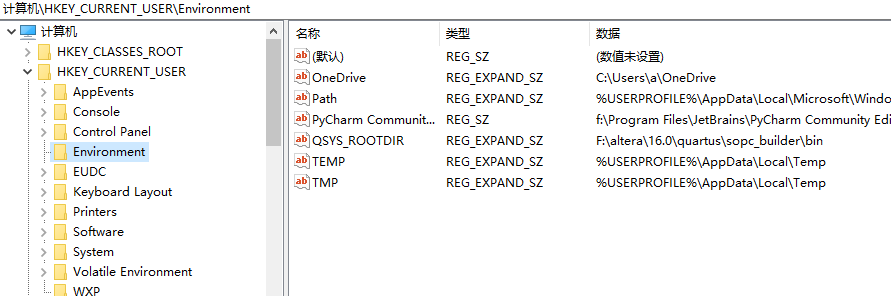


图4-2 Windows注册表编辑器

在batch脚本中，可以使用reg query命令查询注册表的值，查询条件为路径和字段名，即用户环境变量的查询条件为HKEY\_CURRENT\_USER\Environment和Path。查询出来后的输出为字段名+空格+类型+空格+值，但只需要获取path的值，需要对查询输出进行筛选。在batch语句中使用for循环可以按照特定符号分割文本，所以可以根据空格将输出分割为字段名、类型、字段值，最后再获取字段值，具体代码如下

for /f "tokens=1,2,\* delims= " %%a in ('reg query "%regpath\_%" /v %evname\_%') do (

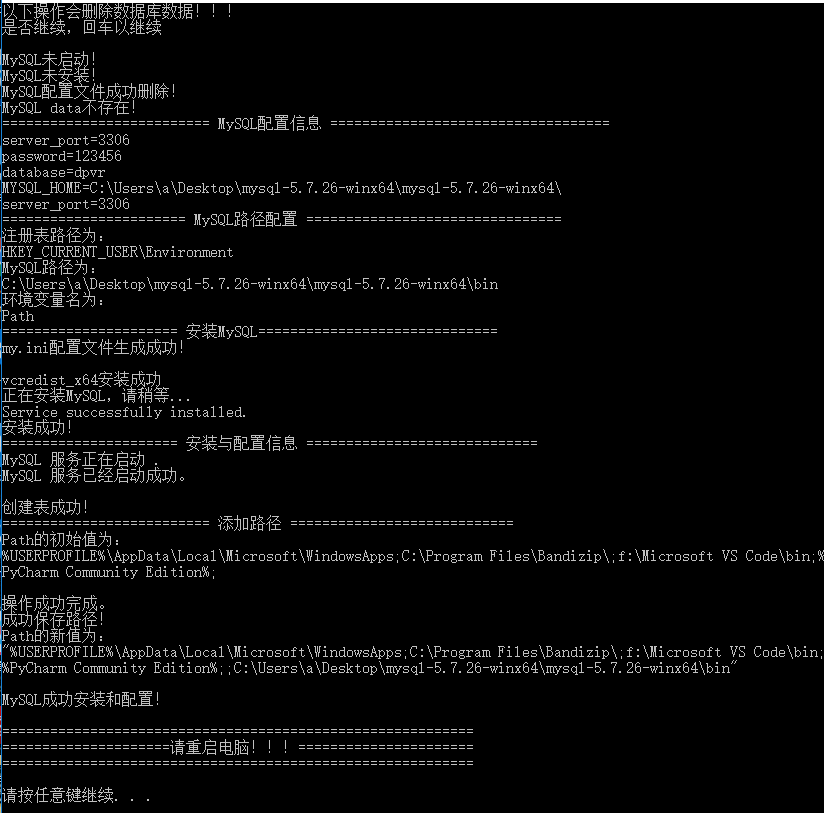
set pathall\_=%%c

)

delims表示分隔符为空格，tokens表示将内容分割为三部分，且将三部分分别赋值给%%a，%%b，%%c，最后再将字段值部分赋值给变量pathall\_。在添加path时需要首先确认该系统中是否已经存在path，如果存在则不添加并作出相应提示；如果不存在则使用reg add 命令修改path即可。至此，关于MySQL的一键配置环境完成。

### 结果分析

经过测试，脚本适用于5.7版本的MySQL数据包安装，低版本会出现无法创建数据文件的错误，从而导致无法成功安装。在安装时，必须保证系统环境中不存在已经安装MySQL数据库环境，否则会因为端口冲突造成数据库无法成功安装。通过注册表添加环境变量多数情况下环境变量不会立即生效，需要重启电脑，脚本成功运行结束后会做出提示。如图为MySQL安装成功的脚本运行情况。



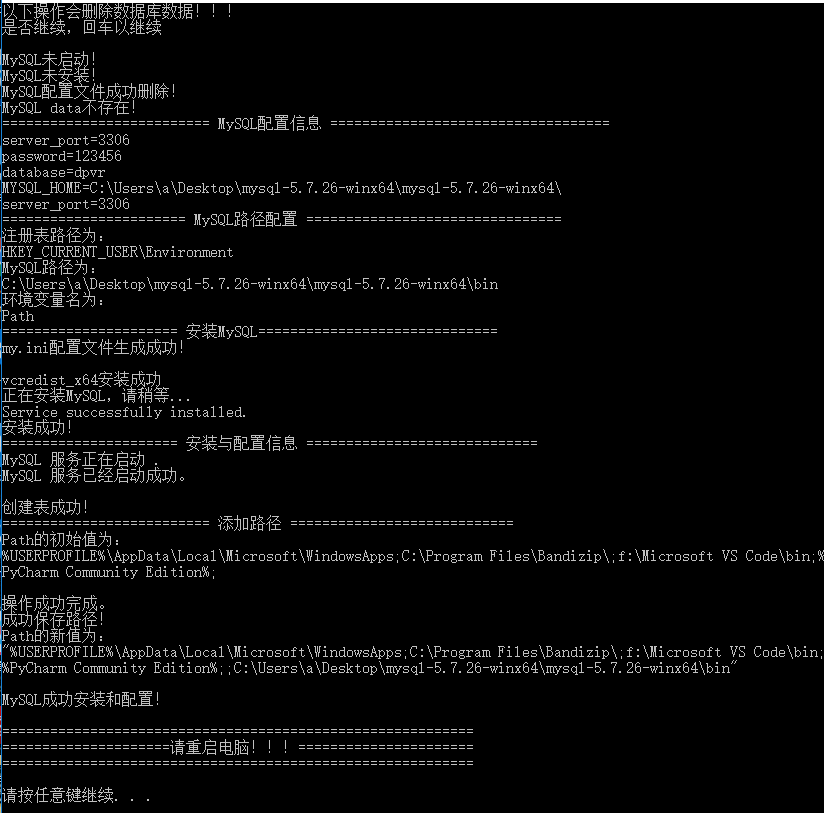


图 4-3 脚本运行成功情况

数据库安装成功后环境变量可以在Windows设置中查看，如图

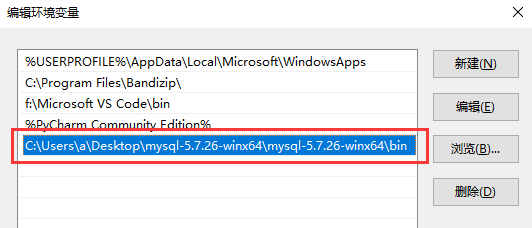
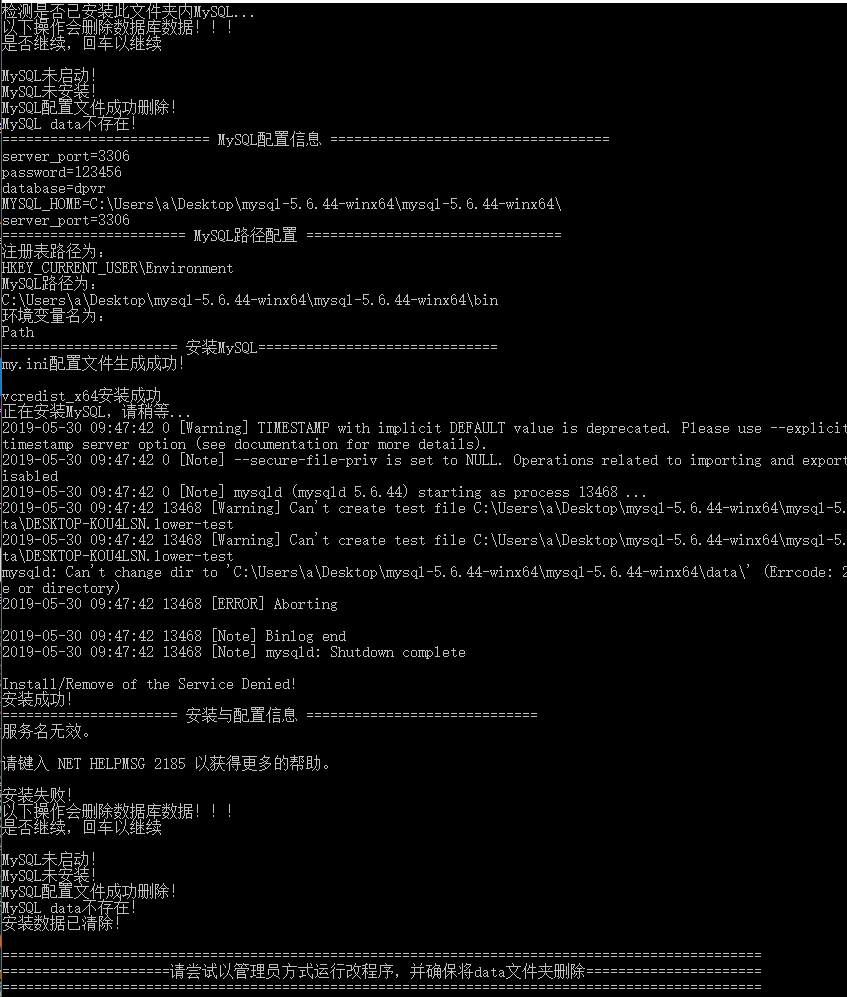


图4-4 环境变量查看

在MySQL服务无法成功启动时，MySQL会提示输入命令查询帮助，之后是脚本做出的相关失败提醒。根据提示可以看出，在安装过程中MySQL服务无法启动，data文件未成功创建，配置文件成功创建，在失败后之前所有创建的文件最后都被清理。如图为脚本运行后安装失败的情况。



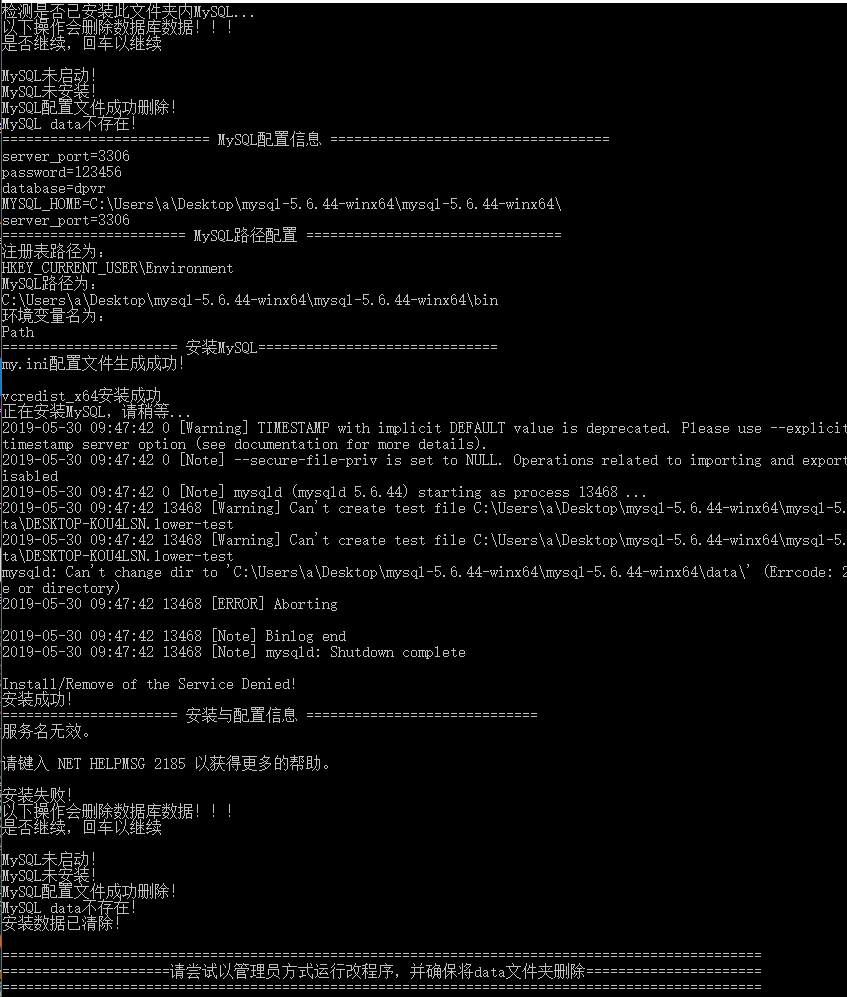


图4-5 脚本运行失败情况

如图，执行命令mysql -uroot -p输入密码后即可登录mysql的root用户。

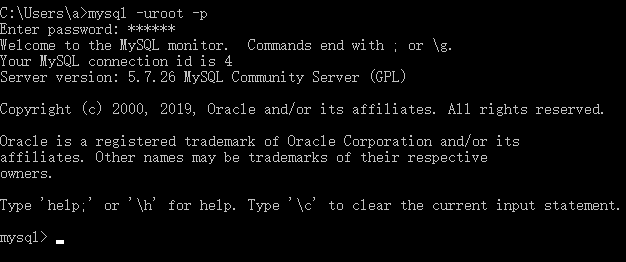


图4-6 登录MySQL

进入MySQL环境后，即可通过SQL命令查询数据库内容，如图可以看到脚本已经自动创建了dpvr的数据库。

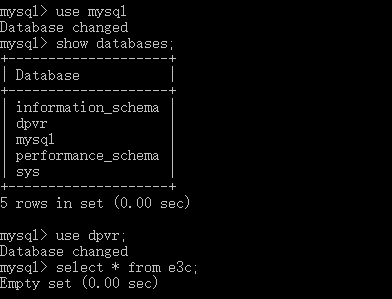
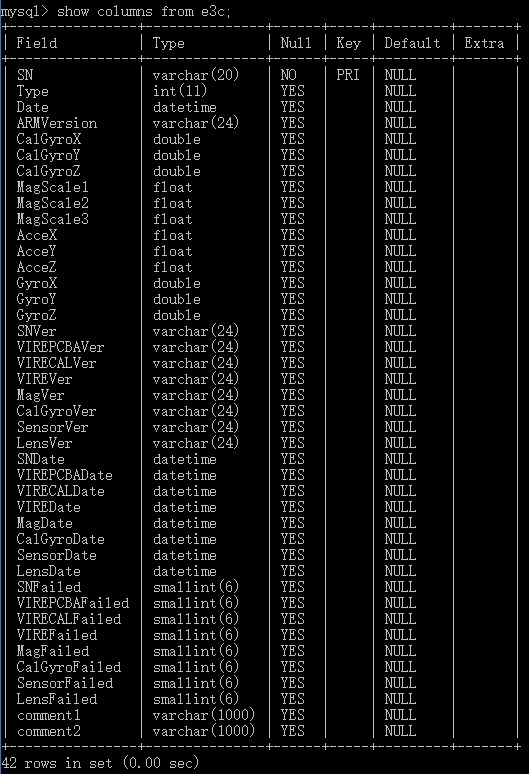


图4-7 数据库查询

如图，脚本自动创建的数据库表文件为e3c，由于目前并未录入任何数据，数据表中数据为空，可以通过show columns from e3c查询表的字段名，如图为脚本自动创建的VR生产数据表的各个字段及类型。



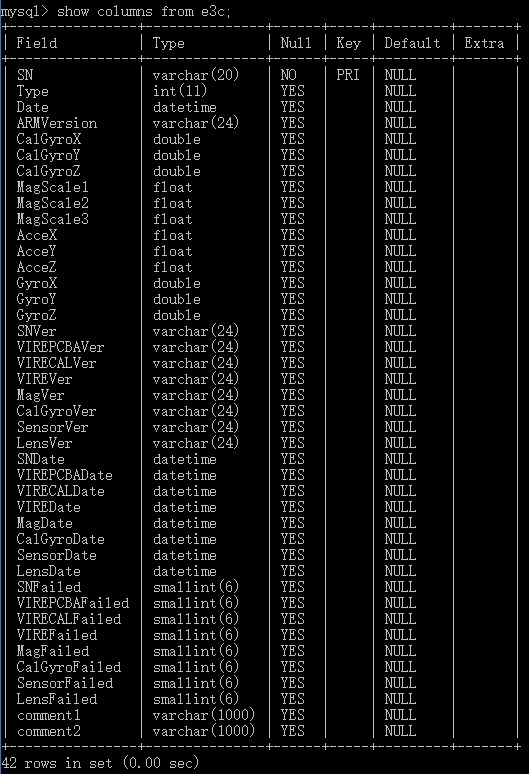


图4-8 数据表字段和类型

### 总结

作为持续集成中的一环，环境配置决定了持续集成的基础运行能否稳定。在VR设备生产过程中，其实就是一个集成的流水线，只是不同于软件开发中的持续集成，生产过程的迭代并无工作量的增加，所以尽可能地迭代更快地生产VR设备才能使收益最大化。zaiVR设备生产过程中，刷机为构建过程，功能测试为测试过程，这些都依赖于生产工人的生产能力、和生产工具的稳定性。保障生产工具的稳定性即保证生产环境的稳定性，通过自动化的环境配置，实现了环境配置的最快部署，加快了生产流程，同时自动化环境配置可以防止配置的遗漏，保证整个生产环境的持续稳定。

在编写自动化配置脚本时，通过深入了解批处理，批处理其实可以应用于很多日常的操作当中。批处理脚本只依赖Windows的原生环境，且仅仅几条简单的命令即可解放大量的手动操作。很多的开发工具都预留了命令行操作，可以通过在批处理脚本中写入命令对开发工具进行快速操作。

在Windows中，注册表中保存了Windows所有的软件配置项，部分软件可能未开放的设置都可以通过修改注册表强制进行修改。对注册表的修改更适用于在脚本中使用，适合自动化，但需要对表中变量值足够了解，否则修改注册表是非常危险的操作。

在作为开发人员编写脚本时，必须对脚本运行的可能性做出各种判断，针对脚本的使用者做出相应的设计，傻瓜式的设计是最能够适用所有人的。无论是普通用户，还是生产工人或者是其他开发人员，快速地自动化流程和良好的提示都能使得脚本的体验得到提升，用户可以通过提示向开发人员寻求解决方案，开发人员也可以根据提示快速调试解决问题，也可以对脚本进行二次开发。当然，任何自动化都是固定的，但是生产流程往往是多变的，自动化脚本必须能够快速适应需求进行更改，在编写过程中发现，将平凡使用、需要修改的脚本命令或参数放在一起，并作出良好的注释是最佳的方式。

## VR应用测试情况自动报表脚本设计

在测试VR设备过程中，开发人员和测试人员是通过项目管理软件进行bug提交与解决沟通的，公司需要一个自动报表系统来更好地对bug数据进行统计。

### 设计需求与方案

本人实习所在公司使用“禅道”项目管理工具对开发的项目进行设计、创建、开发、测试、维护等等，如图为项目各参与人员的工作流程图：



图4-9 禅道项目管理流程

在项目管理过程中，测试人员将bug提交到禅道上，开发人员通过查看bug具体描述对bug进行修复，在禅道上对bug进行状态更新。如图为bug各状态信息。



图4-10 Bug各状态信息

虽然禅道上对bug的信息展示非常具体，但是对于大多数项目参与人员来说，当bug数量越来越多时，关注的重点往往是严重程度最高比如一级和二级bug，大量的三级bug累积下来越积越多。而项目经理在对项目进行可行性评估时，首先是以一级和二级bug的数量来评定版本是否可以发布，其次是三级bug的数量。因而往往存在一个问题，较严重的bug被解决后，在项目经理进行版本代码复核时，往往会因为三级bug过多而将版本打回。所以，除了对严重bug提高关注，还需要bug的整体结构分析。通过每天的代码记录，将各类登记的bug统计下来，并通过图表对bug数量的变化进行可视化的展示，并通过邮件将代码统计情况每天发送给所有的项目参与人员。

开发人员可以根据每天收到的报表查看当前bug的数量与发展趋势，除了着重对较严重bug进行关注外，还可以了解目前三级bug数量的变化情况。这样，开发人员可以通过bug的数量与测试人员进行沟通，了解bug数量变化的原因，从而对出现问题的项目模块快速核查，可以一次性解决大量的低级bug。开发人员也可以通过图表所展示的bug信息，进而分析bug数量变化对自己未来开发工作重心的影响，提前对自己的工作内容进行调整，提高开发的效率。

项目经理可以根据报表中bug的数量统计，大体了解目前项目的发展情况。当bug数量大量产生时，说明目前项目整体仍然处于开发初期时的不稳定状态，那么此时项目经理可以根据bug情况与测试和开发人员进行沟通，了解目前产生大量bug的具体模块和问题所在，项目经理可以根据这些情况对项目参与人员的工作内容进行调整，缩短项目的不稳定期，快速提高项目的开发进程。当新增bug中严重bug数量较少，三级bug数量较多时，说明此时项目已经较为稳定，仍然存在一些影响体验的小问题存在，当然这些小问题无法被忽视，用户的体验往往都是集中在细节的体验上的。这种情况下，项目经理可以开始考虑关注大量的三级bug，通过分析三级bug产生的模块，与产品经理进行沟通，讨论模块设计的可行性，要求产品经理在下一个版本中做出相应的修改和设计。当bug数量持续减少时，说明此时bug的解决速度远远大于bug的产生速度，此时bug已经基本不会产生，项目整体上已经稳定，项目经理可以将关注度放在开发人员身上，开发人员进行沟通，讨论遗留bug的影响情况，对遗留bug的解决优先级进行调整，进而在bug逐渐减少的一定界限内，对项目版本的发布时间进行提前安排。

测试经理可以根据报表中的bug数量统计，了解目前测试人员的测试情况。在项目版本刚刚提测后，测试经理可以根据bug初期的产生情况决定该版本是否可以继续进行测试，如果bug产数量变化十分大，那么测试经理可以与开发人员和项目经理进行沟通，停止测试并打回改版本，要求开发人员对项目版本修改模块进行重新开发，减少测试工作量。在项目进行到后期，bug数量趋于稳定时，测试经理可以对项目测试参与人员的工作进行重新安排，将一部分测试人员安排到其他项目组去，一部分测试人员继续进行项目复核。

整体来说，bug数量的多少也可以侧面体现出项目参与人员的工作情况，可以用于对项目参与人员的工作评定中。项目参与人员也可以通过bug数量的情况相互激励，从精神上提升他们的工作体验。

所以需要一个自动bug统计报表系统，该系统主要具备如下功能：

（1）抓取禅道bug数据。项目的bug情况都记录在禅道上，但是禅道并没有相关的bug趋势相关的统计功能。禅道系统是基于网页的，一般来说，抓取网页的数据最好的方法是使用爬虫，可以通过脚本编写爬虫抓取bug情况信息，并存储下来。

（2）数据存储。将bug数据抓取下来后，需要每天抓取到的数据存储在数据库中，一般来说选取MySQL作为数据库来读写数据是最方便的，但是生产的数据表需要能够在邮件中方便查看，那么需要一种邮件系统支持在线显示的数据文件，Excel作为数据文件符合这种要求。

（3）绘制数据图表。将每天的数据存储到Excel表中后，需要将数据再次读取出来，按照时间进行绘制曲线图、柱状图等等，并且绘制出的图表可以保存为图片。

（4）自动发送邮件。生成的数据文件和图表图片以及相关统计情况需要通过邮件发送到项目参与人员中，最后可以通过Jenkins定时任务将每天定时发送。

### 设计过程

通过对自动报表系统的要求和设计的分析，本项目采用Python脚本实现自动抓取bug数据、读写数据、绘制统计表、发送邮件的功能，再通过Jenkins的定时任务功能每天运行一次脚本，实现定时发送的功能。

相应的，本项目的python脚本按照功能分为如下几个python文件：

表4-2 报表脚本文件

|  |  |
| --- | --- |
| 脚本 | 功能 |
| data2file.py | 数据处理 |
| send\_email.py | 邮件发送 |
| settings.py | 脚本相关设置 |
| zentao\_driver.py | 禅道抓取数据 |
| report\_main.py | 主函数 |

本脚本将除了report\_main.py主函数脚本外的其他脚本autoreport文件夹中，作为一个autoreport工具包，其他开发人员可以自己编写主函数调用工具包中的脚本中的类和方法。Python是一门面向对象的脚本语言，工具包中的脚本都创建了类和方法，各脚本中创建的类如下表：

表4-3 报表脚本创建类

|  |  |  |
| --- | --- | --- |
| 脚本 | 类 | 功能 |
| data2file.py | Data2file | 数据处理 |
| send\_email.py | SendEmail | 邮件发送 |
| settings.py | Settings | 脚本相关设置 |
| zentao\_driver.py | XPath | XPath映射 |
| ZenTao | 禅道抓取数据 |

如下按照脚本模块介绍脚本的设计过程。

1. settings.py设置相关

settings.py脚本中创建了Settings类，该类中只创建了类属性，即脚本设置的相关参数，这些参数是脚本使用者可以修改的设置，使用者可以根据自己的情况对各参数进行修改。所有脚本都是基于settings.py脚本中的参数驱动的，必须保证个脚本都对该脚本进行了调用声明。下表为该脚本的Settings类属性介绍：

表4-4 Settings类属性

|  |  |  |
| --- | --- | --- |
| 参数 | 描述 | 备注 |
| time\_now | 目前时间 | strftime('%Y/%m/%d') |
| mail\_host | SMTP服务器 | smtp.mxhichina.com |
| mail\_user | 邮箱地址 |  |
| mail\_pass | 邮箱密码 |  |
| sender | 发送者 | name<usr@mail.com> |
| receivers | 接受者 | name<usr@mail.com> |
| cc | 抄送者 | name<usr@mail.com> |
| subject | 邮件标题 |  |
| sleep\_time | 模拟点击间隔 |  |
| in\_address | 内部禅道地址 |  |
| in\_account | 内部禅道账户 |  |
| in\_password | 内部禅道密码 |  |
| ex\_address | 外部禅道地址 |  |
| ex\_account | 外部禅道账户 |  |
| ex\_password | 外部禅道密码 |  |
| folder | 数据文件储存位置 |  |
| xls | 数据表文件名 |  |
| num\_table\_data | 邮件显示数据条数 |  |
| self.num\_img\_data | 折线图统计数据条数 |  |
| tinify\_key | 图片压缩api的key |  |
| bug\_type | Bug类型 |  |

2．zentao\_driver.py禅道数据抓取相关

zentao\_driver.py中创建了两个类：XPath和ZenTao，这两个类都是基于selenium的 webdriver创建的。

Selenium是以一套web自动化测试框架，在python中拥有selenium相关的脚本库可以调用，除此之外还需要下载chromedriver.exe作为驱动。如图为selenium支持的浏览器，本脚本采用了chrome浏览器。

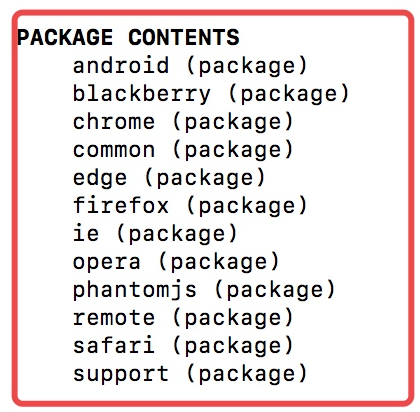


图4-11 selenium支持浏览器

如图为chromedrvier的部分镜像，chromedriver必须与chrome浏览器的版本相对应，否则将无法启用。

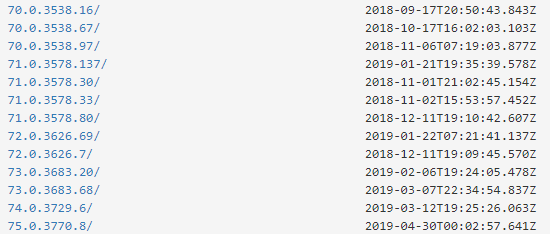


图4-12 webdriver对应chrome版本

Selenium除了用于web自动化测试外，常常用作网络爬虫自动抓取数据。Selenium抓取数据主要的方式是通过定位元素，再对元素进行模拟点击、数据获取等操作。Selenium定位元素有很多方式，如通过元素的id、text以及xpath。这些定位方式并不是一成不变的，需要根据网页的具体情况进行分析，一般来说需要结合起来一起使用进行元素的精准定位。一般来说，网页元素的路径是不变的，那么使用xpath进行元素定位是最为迅速地。Xpath通过特定的语法对元素的路径进行描述，如此看来我们需要了解xpath的语法，看起来是很复杂的。但是chrome浏览器可以自动生成元素的xpath，相对于通过text或id定位元素。xpath是唯一的，不必考虑是否会定位到其他元素去。

部分网站为了反爬虫或者出于其他安全因素会对元素的路径进行实时更新，这种情况下xpath在每次页面更新都会改变，定位无法生效。此时，就需要通过id和text以及其他分析到的变化条件对元素进行综合性精准定位，这种方式是非常复杂的。当然，当页面元素较少时，元素的id一般是唯一的，直接使用id定位即可。

通过测试，禅道的元素路径是不变吧，所以本脚本中大量使用了xpath进行定位。但是，在使用xpath编写脚本后，如图由于xpath中存在大量特殊字符，在脚本中显示十分混乱，对脚本的可读性造成很大影响。

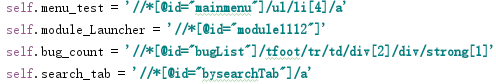


图4-13 脚本中的xpath

如此，脚本中对xpath创建一个单独的类，通过类属性对xpath进行映射，且XPath类的属性变量名通过一定的格式编写，开发人员和使用者只需直接使用映射后的XPath类属性，无需考虑xpath语法。脚本要求其他开发人员在进行脚本修改时在每次使用其他 xpath时将其添加到XPath类中按照格式进行映射。

在zentao\_driver.py脚本中还创建了一个类ZenTao，该类主要创建了几个抓取数据的方法，这些方法抓取禅道不通过项目、模块中的bug数据。下表为ZenTao类的属性和方法的相关描述：

表4-5 Zentao类属性

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | 备注 |
| settings | 设置 | 使用Settings类实例 |
| driver | webdriver | 使用Chrome |
| xpath | XPath实例 |  |
| sleep\_time | 模拟点击间隔 | 使用Settings类属性 |

表4-6 ZenTao类方法

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 | 参数 |
| click\_by\_xpath | 通过xpath定位并点击 | xpath |
| click\_by\_id | 通过id定位并点击 | \_id |
| click\_by\_text | 通过text定位并点击 | text |
| bug\_count | 获取bug数量 |  |
| read\_table | 获取bug详细信息 |  |
| login | 登录外部或内部禅道 | addr, account, password |
| bug\_search\_case1 | 获取内部禅道bug数据 | project, module\_xpath |
| bug\_search\_case2 | 获取一级bug信息 | project, module\_xpath |
| bug\_search\_case3 | 获取外部禅道bug数据 | project |

如表所述，ZenTao类方法分为四类：

（1）click\_by\_xpath、click\_by\_id和click\_by\_text通过定位元素进行点击，这些方法出了点击动作外，还加了延时函数，参数为settings中的sleep\_time，以达到模拟点击直接存在间隔。sleep\_time的时间需要经过特定的测试以达到最好，如果太小，相邻两次的点击间隔太短，造成下一次的元素无法定位，脚本报错；如果太大，响铃两次的点击间隔太久，造成脚本运行时间过长，占用系统资源。通过测试sleep\_time的值设为1s较为恰当。

（2）login方法通过定位元素并使用sendkey发送字符串，可以实现自动登录。

（3）bug\_count和read\_table都是对bug的信息进行统计的方法。bug\_count的获取方式比较简单，只需要定位元素后使用get\_attribute('innerHTML')获取元素内容，获取后将字符串取整用作之后对数据处理。经过测试发现，当bug数量为0时，不会显示bug数量，bug数量无法获取脚本运行错误。这里使用了try…except对运行结果进行判断，当无法和获取bug数量时，打印报错信息，并将bug数量置零。

read\_table是获取bug的详细情况，即获取禅道上的bug表信息，获取到的数据为一个二维数组。通过使用chrome浏览器审查元素，bug表的id为“buglist”，class为“text-center”，通过循环，按照tag为“td”读取每行和每列的数据，最后将读取到的数据存储在一个二维数组中。

（4）bug\_search\_case系列方法是模拟用户点击的流程，通过调用前面编写的方法最终获取到需要的bug数据。该系列方法要求使用者自行调用前面的方法进行编写，当然为了应对目前公司的需求，本脚本创建了三个bug\_search\_case方法。大多数的bug数据获取步骤大同小异，一般步骤为登录，选择的项目，按照需求检索，获取bug数据。根据需求，多数数据的获取步骤只是项目与检索模块不同，所以为了方便函数复用，bug\_search\_case方法编写建议预留项目和模块的参数。

3. datafile.py数据处理相关

data2file.py中创建了Data2file类进行数据转文件的相关处理。数据处理主要有两类：Excel数据表的读写和绘制数据图表。Data2file的属性和方法如下表：

表4-7 Data2file类属性

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | 备注 |
| settings | 设置 | 使用Settings类实例 |
| folder | 文件保存路径 | 使用Settings类属性 |
| report\_name | Excel保存路径 | 使用Settings类属性 |
| row\_type | 行标（时间） | 使用Settings类属性 |
| col\_type | 列标（bug类型） | 使用Settings类属性 |
| tinify.key | Tinify API key | 使用Settings类属性 |

表4-8 Data2file类方法

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 | 参数 |
| bug\_list2xls | 将bug情况表写入Excel | table, xls, sheet |
| data2xls | 将bug数量写入Excel | data, sheet\_index |
| Read\_xls | 读取Excel中的bug数据 | sheet\_index |
| data2img | 绘制bug曲线图 | title, rows, sheet\_index |
| data2img\_bar | 绘制bug指派柱状图 | title, x, y |

当获取数据后，首先要将数据写入到Excel中。根据需求，bug数据主要有两类：每日bug统计量和每日bug详情。其中bug统计量需要每天在原有Excel的基础上进行添加，再对所有数据进行统计分析；bug详情只需要关注每天的bug指派、优先级等情况，需将之前的数据覆盖掉。

在python中使用xlrd、xlwt、xlutils函数库可以实现对Excel数据表的读写。Xlrd主要用来读取Excel中的数据，但是不能修改其中的数据，而xlwt用来将数据保存到Excel中，但是数据会覆盖，所以要使用到xlutils.copy创建副本对其修改并保存。

在修改获取Excel表中的数据后，使用pyecharts绘制数据统计图。pyecharts是基于 echarts.js的api编写的python工具包，相比较而言使用echarts.js需要手动编写js文件来设置图标样式，再通过编写python脚本调用api；Pyechaets的图标样式是固定的，该工具包已经集成了echarts.js的各类型的图标，非常方便调用。更方便的是，Pyecharts存在将生成的图表渲染为html和png的方法，只需要简单掉调用该方法即可。所以脚本中采用了pyecharts作为图表绘制库。

pyecharts渲染后的图片的需要插入到email正文中，通过测试发现，渲染后的图片每张达到5M左右，这导致每天bug统计邮件非常大，所以需要在渲染后对图表文件进行压缩。脚本中使用tinify工具包进行图片压缩。

Tinify工具包通过调用TinyPNG的api进行图片压缩。TinyPNG是一个可以在线压缩PNG格式图片的网站，经过测试，生成的图表文件经过压缩后只有几百K，压缩效果非常好。如图TinyPNG的官方文档中提供了api以供图片的自动上传压缩。

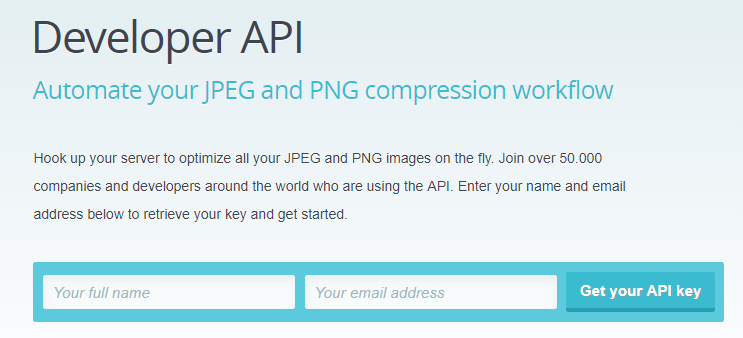


图4-13 TinyPNG API

用户可以通过姓名和邮箱获取一个API key，每个邮箱可以免费获得每个月500次的压缩次数，完全够用于自动压缩图表。在python中Tinify很好的封装了该api，只需要将获取的API key通过 tinify.key导入即可。tinify.from\_file(img\_path).to\_file(img\_path)方法可以将PNG图片直接压缩。

4. send\_email.py发送邮件相关

send\_email.py脚本中创建了SendEmail类实现将文字、图片、附件添加到邮件中并发送。SendEmail类的属性和方法如下表：

表4-9 SendEmail类属性

|  |  |  |
| --- | --- | --- |
| 属性 | 描述 | 备注 |
| settings | 设置 | 使用Settings类实例 |
| msgRoot | 邮件对象 |  |
| sender | 发送者 | 使用Settings类属性 |
| receivers | 接受者 | 使用Settings类属性 |
| msg\_html | 邮件正文 |  |
| msgAlternative | 邮件附件 |  |

表4-10 SendEmail类方法

|  |  |  |
| --- | --- | --- |
| 方法 | 描述 | 参数 |
| read\_addr | 邮件地址格式化 | name\_email\_list |
| text2html | 文字转html | text, weght |
| table2html | 表格转html | data, width |
| img2html | 图片转html | img, tag, width, height |
| add\_html | 将html正文添加到邮件 |  |
| add\_attach | 添加附件 | file |
| send\_email | 发送邮件 |  |

使用python发送邮件主要使用了smtplib、email.mime、email.header和email.utils函数库，其主要思路为创建一个邮件对象msgRoot，该对象包含标题、发件人、收件人、正文和附件，该对象是一个字典类型，其相应的key和value如下表：

表4-11 msgRoot的key和value

|  |  |  |
| --- | --- | --- |
| key | value | 描述 |
| Subject | Header() | 标题 |
| From | Formataddr() | 发件人 |
| to | to\_addrs | 收件人 |
| cc | cc\_addrs | 抄送 |

邮件正文是以html格式编写的，其中包含文字、表格、图片等等。最后将正文和附件一起添加到msgRoot对象中，整个邮件内容就完成了。

邮件发送主要是使用邮箱代理服务器发送的，该脚本使用SMTP SSL验证，请求端口为465，使用SMTP服务器。登录SMTP服务器后，使用sendmail方法即可发送邮件，其参数为发件人、收件人和msgRoot。

当然，邮件发送常常会因为各种原因出现问题，开发者需要通过错误码进行分析出错原因，提出解决方案，smtplib库中对相应的错误码进行了规定，在脚本中使用try…except可以进行错误提示，具体报错对应如下：

表4-12 smtplib报错提示

|  |  |
| --- | --- |
| 错误 | 描述 |
| SMTPConnectError | 连接失败 |
| SMTPAuthenticationError | 认证错误 |
| SMTPSenderRefused | 发件人被拒绝 |
| SMTPRecipientsRefused | 收件人被拒绝 |
| SMTPDataError | 数据接收被拒绝 |
| SMTPException | 邮件发送失败 |

5. report\_main.py主函数

前面已经将脚本的各个模块都完成了，在主函数就要对所有模块进行组装，实现完整的脚本功能。在主脚本中，根据需要统计的项目和模块可以单独划分函数，以防止主函数过于冗长，不利于修改，主脚本中使用bug\_count\_report系列来表示实现需求的响应函数，这些函数都可以单独运行，相互直接不会有所影响，所以建议开发者在添加新的统计方法时将方法按照bug\_count\_report的格式进行编写，以实现脚本的可持续维护。

在主函数需要创建Settings、Data2file和SendEmail实例，并将这些实例传递到各个方法函数中。在每个方法函数中，只需创建ZenTao的实例，使用这些实例可以通过调用其方法快速实现bug数据抓取和数据统计和邮件发送。至此，脚本所有相关功能都已完成。

最后，只需要将脚本放在Jenkins服务器中，通过每日定时构建实现每日统计bug情况并自动发送邮件的功能。

6. 脚本移植

脚本编写测试成功后，需要放在Jenkins服务器中，但此时python缺少相应的函数库从而无法运行。需要在脚本编写环境中运行pip freeze >requirements.txt生成依赖文件，再到Jenkins服务器中运行pip install -r requirements.txt即可将所有依赖安装完成。如图为生成依赖列表。

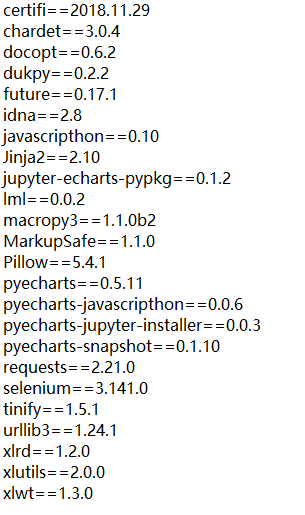


图4-14 脚本依赖

### 结果分析

如图为4月26日自动发送的邮件，邮件中标题、收件人、发件人、抄送均显示成功。



图4-15 邮件信息

该天P1Pro项目一级Bug指派情况如图，邮件中使用表格将一级bug的编号、严重程度、优先级、标题、bug状态和指派人列出，并使用柱状图对各指派人的一级bug进行统计。

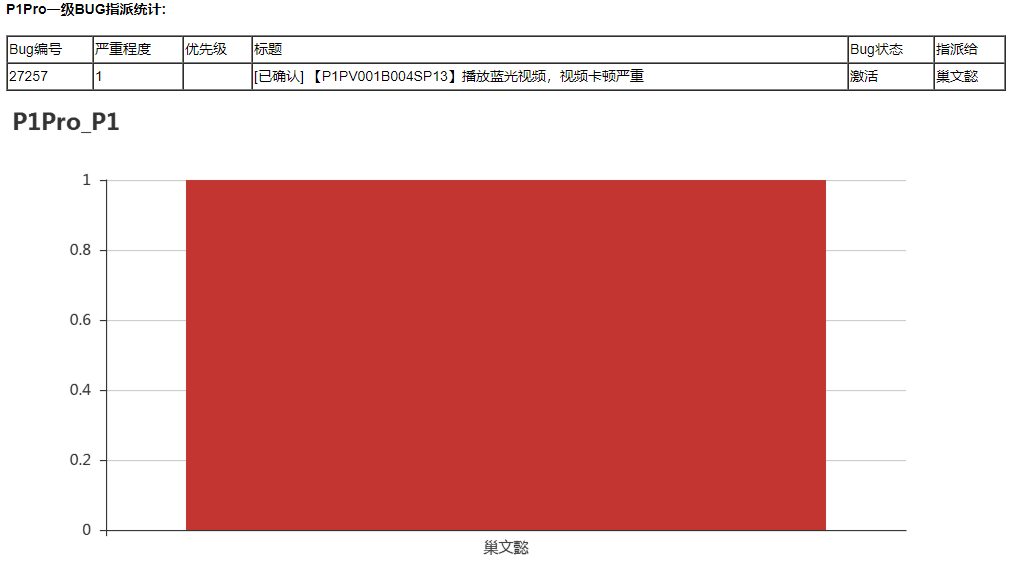


图4-16 一级Bug指派

如图为该天P1pro Launcher模块的Bug情况统计，邮件中的表格取近四天的统计数据，折线图取近30天的统计数据。从表中可以具体看出仅四天各各类型bug的具体数量变化，可以明显看出在4月25日到4月26日期间关闭了100多个bug，说明当天集中工作对遗留的已解决的bug进行了验证，并将其关闭。从折线图中可以看出近30天的bug数量变化情况，在4月12日到4月14日之间三级bug大量增加，此时应该是新版本刚刚发布，且改动较大，新问题较多。而在4月25到4月26日这些三级bug被大量解决和关闭。

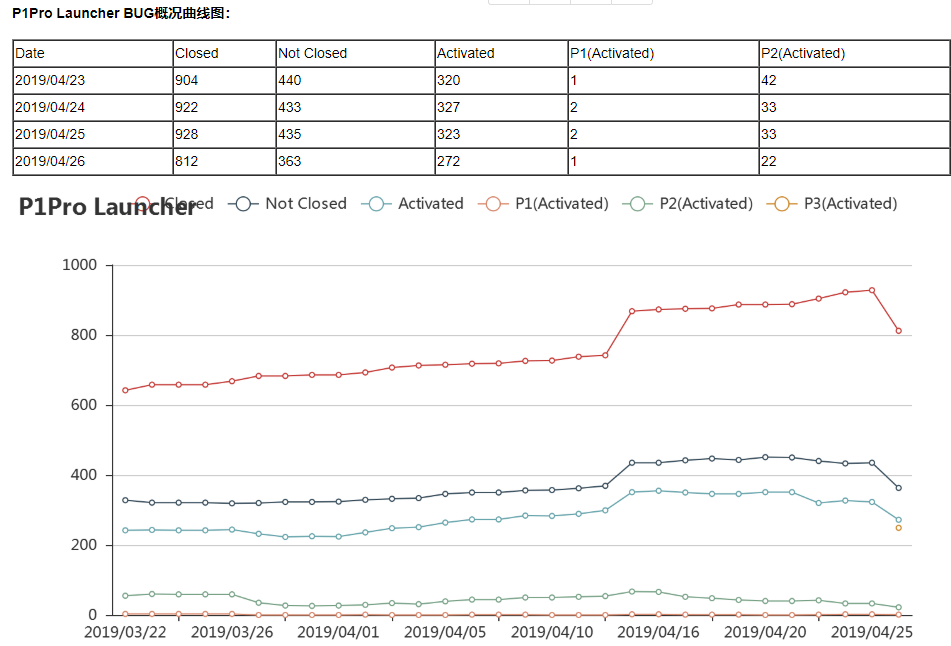


图4-18 P1Pro Launcher Bug统计

如图为P1Pro一体播控Bug概况图，可以看出该项目整体上来看近30天的Bug数量较为平稳，但是每天的Bug数量仍然在上下浮动，说明目前项目整体功能已经完善，但是仍然在持续开发，版本更迭的改动并不是很大。

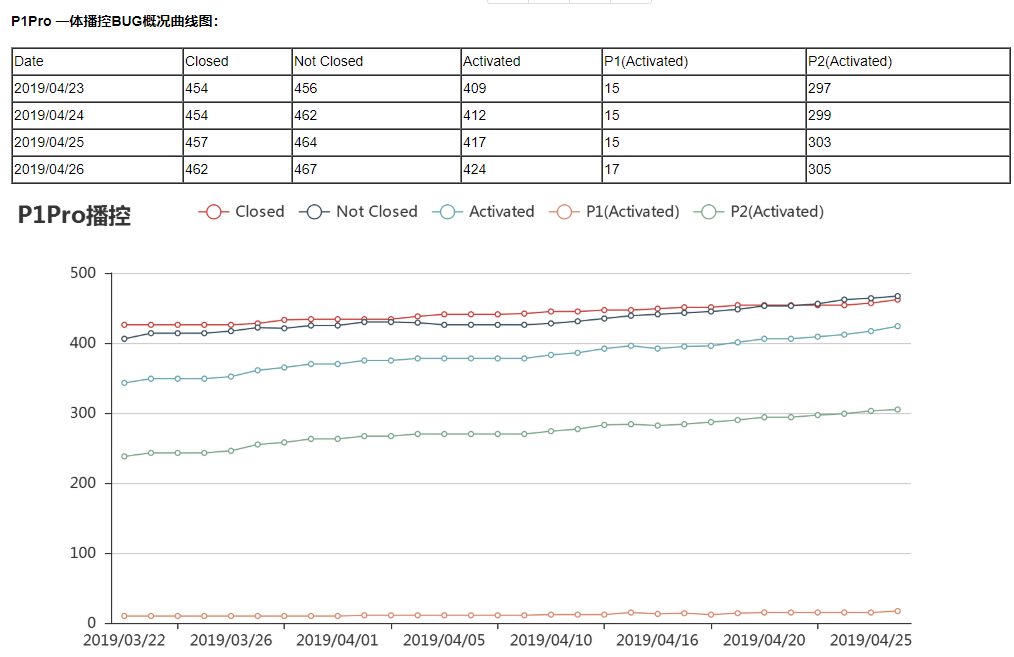


图4-19 P1pro一体播控Bug统计

如图为研发总监在查看邮件后发现二、三级bug数量回升，督促项目组开发人员留意bug的修复。

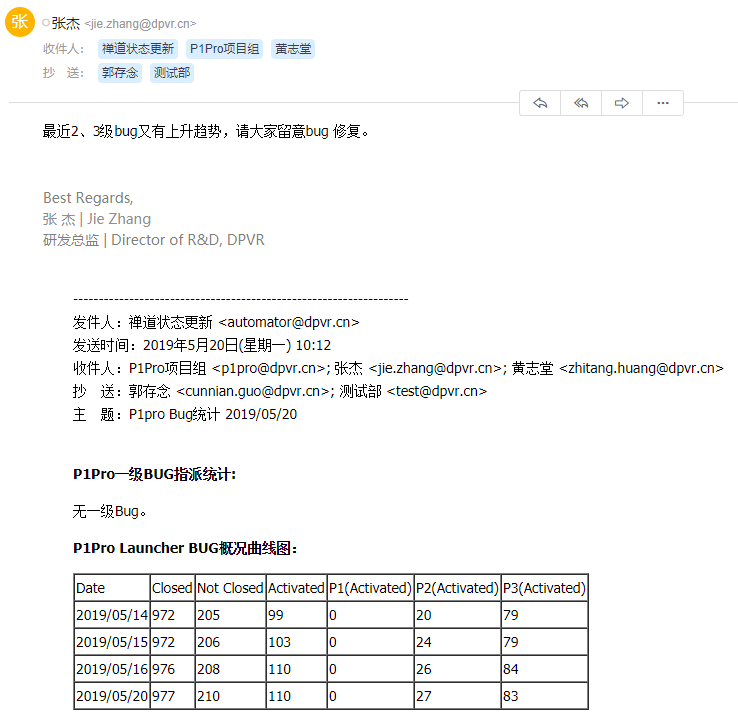


图4-20 研发总监邮件

### 总结

作为持续集成中的一环，通过邮件发送日志信息可以对持续集成后的相关情况进行及时通知，而通过邮件发送每日报告，可以对持续集成的多次迭代情况进行整体性分析。通过分析项目在进行多次迭代后的Bug数量统计情况，可以很好地分析出目前项目的进展，项目的各个参与人员都对项目的大致情况进行了解。

除了对该脚本作为持续集成一环的理解外，通过亲自编写脚本更能体会到持续集成的重要性。在编写脚本过程中，功能的实现其实数十天就可以完成，然而在需求逐渐增加时，不仅仅需要考虑的是功能的实现，更多考虑的是代码的可维护性。在编写代码初期，虽然需求的具体功能已经实现，但是所有代码都集中在单个脚本中，许多功能代码都要重复编写，参数在脚本中分布混乱，如XPath等包含特殊字符的字符串在脚本中随意防止，脚本中参数和功能也没相关的注释。当有新的需求时，简单的一个需求修改如模拟点击间隔的时间调整，需要在脚本中作大量的修改。其他开发人员在读脚本时也很难理解脚本的实现方法，无法进行二次开发。由此可见，这种脚本是十分失败。在之后持续的编写中，考虑更多的是脚本中参数、函数的复用性，以及脚本的易读性，需要随时准备对脚本结构和函数进行重构。具体体现在如下几个方面：

（1）模块化。通过对脚本功能的分析，脚本可以分为数据抓取、数据处理、邮件发送、设置几个模块，最后在主函数中进行调用。

（2）提高代码复用性。通过分析各代码在脚本中的使用情况，当一部分功能代码在脚本中多次被使用，可以考虑将该部分代码封装成函数。这样在进行功能修改时只需要修改功能函数即可。

（3）将可修改参数集中。在脚本中常常存在一些参数需要根据需求进行修改，这些参数分布在脚本各模块的不同位置，在修改脚本时需要对所有的脚本模块进行修改，工作量较大且易遗漏修改。可以通过将所有可以修改的参数集中放在脚本某一位置或单独的脚本中，如本脚本中使用settings模块将所有参数集中在一起，其他模块都调用该模块。

（4）代码整洁美观。整洁美观的代码会让脚本的代码结构更加清晰，提升开发者的便携体验，造成代码混乱的往往是一些字符串，它们可能很长或存在特殊字符。可以通过将这些字符串映射为变量值，并将它们集中编写，这样在使用这些字符串是只需要直接调用这些变量即可。

（5）良好的注释。良好的注释并不是指注释越多越好，对于大多数简单的语句注释并没有意义，常需要注释的是参数、功能函数、类、关键功能语句。

（6）规范代码书写。 规范的代码书写能使再重新读代码时迅速了解代码的含义，代码规范体现在编写时的方方面面，如命名方式、注释方式、换行等等。使用Pycharm编写脚本时会对不规范的语法进行波浪线提醒，可以很好地对语法进行约束。

（7）方便调试。由于该脚本运行时间较长，在对新改动进行测试时需要等待脚本完全运行后才能知道结果。本脚本中通过在每个模块中添加if \_\_name\_\_ == '\_\_main\_\_':判断，可以运行单个模块进行单个模块的调试。

脚本在“每日构建”过程中也显露出了很多问题，目前脚本仍存在如下问题：

（1）运行时间较长。在脚本运行过程中发现，脚本运行的时间主要集中在抓取数据时的模拟点击和图片压缩。前者是由于使用selenium抓取数据是模拟点击时间间隔造成的，后者是由于在使用Tinify调用api过程较长引起的。目前考虑使用python中的多线程使多个webdrvier并行抓取数据，同时对前面生成的图片并行地压缩图片。

（2）缺少日志。虽然在Jenkins的构建中会将脚本构建情况进行保存，但是往往在脚本报错时这些提示是不够的，需要加入日志系统对脚本的各个阶段进行提示和记录。

## 本章小结

本章分析研究了基于Jenkins运行的VR生产环境的自动化脚本和VR应用测试情况自动报表脚本，实现了Jenkins服务器下的功能扩充。首先研究和分析了在VR生产环境中数据库自动化的必要性和在持续集成中的理论体现，再通过分析脚本需求对脚本实现的的代码对脚本的主要结构进行描述，对脚本运行结果进行分析。然后研究和分析了在项目管理中，持续集成中的报表环节的重要性，再通过分析需求对自动报表脚本的实现对脚本的主要结构进行描述，对脚本运行结果进行分析，并总结了编写脚本过程中的感受，侧面体现了持续集成的重要性。

# 总结与展望

1. 工作总结

在软件开发过程中，软件集成阶段是最容易出现问题的阶段，错误的集成方式经常导致项目进行延期交付甚至开发失败。项目生命周期不可控、软件质量无法保证、测试效率低时间久等问题也经常出现在软件开发过程中。在VR行业中，项目的开发仍然避免不了上面的问题。本文基于持续集成基础理论研究，设计和搭建了Jenkins持续集成系统，并实现了在VR企业应用中的两个实践，其中所完成的工作主要有以下几个方面：

（1）对软件集成进行原理分析，描述了集成的概念，进而引出各种经典集成方式。通过分析各集成方式的原理和缺陷，从最基础讲起，重点分析了持续集成的发展情况和具体要求。

（2）研究了持续集成系统的设计组成。首先研究和分析了持续集成系统的组成结构，然后描述了持续集成的具体流程，最后选区Jenkins作为持续集成服务器，并使用docker搭建Jenkins持续集成服务器。

（3）研究了基于Jenkins运行的VR生产环境的自动化脚本和VR应用测试情况自动报表脚本，实现了Jenkins服务器下的功能扩充。首先研究和分析了在VR生产环境中数据库自动化的必要性和在持续集成中的理论体现，再通过分析脚本需求对脚本实现的的代码对脚本的主要结构进行描述，对脚本运行结果进行分析。然后研究和分析了在项目管理中，持续集成中的报表环节的重要性，再通过分析需求对自动报表脚本的实现对脚本的主要结构进行描述，对脚本运行结果进行分析，并总结了编写脚本过程中的感受，侧面体现了持续集成的重要性。

2. 进一步研究工作

本文研究的主要是依赖于Jenkins的持续集成系统并使用脚本实现持续集成的环境配置和反馈环节，并未对VR企业中的应用持续构建作详细研究。本人在对VR应用进行测试期间发现，移动VR应用的测试主要集中在Unity的兼容测试和性能测试上，这些测试往往需要的设备量大、时间久，在今后的工作中可以进行详细的持续集成系统的设计。而docker目前可以作为持续集成环境的容器，具有良好的移植性，在之后的工作也将对docker技术进行研究，实现持续集成环境的快速搭建和移植。

# 致谢

在此论文完成之际，我将对在论文期间对我进行过指导和帮助的老师、同学、同事表示衷心的感谢和诚挚的祝福。

首先要感谢的是我的学校导师朱晓强教授在实习中的关怀和学术上的指导。朱老师在我实习期间，亲自去公司查看工作环境和工作状态，给我提供了很多实习上的指导，在写论文期间，朱老师也不遗余力的对论文相关问题进行解答，使我在工作和毕设上能够很好的协调。然后感谢我的企业导师张新雨和宋艳。在企业中张新雨是我的第一位企业导师兼同事，在我刚刚进入企业将我从学生思维转到工作思维上，帮助我适应了企业的环境。宋艳是我的第二位企业导师兼领导，根据我的毕设方向为我适当的安排工作并提供设备上支持。感谢我的同事和同学在我工作期间给予了陪伴和支持，在毕设上也提出了宝贵的意见。最后，感谢我的父母和姐姐，你们在我忙碌的实习和毕设期间给予了我努力的动力。

最后，再次感谢所有在我本科生涯期间给予我指导和帮助的老师、朋友、同学，谢谢！

# 参考文献

1. 亨布尔. 持续交付 : 发布可靠软件的系统方法 : reliable software releases through bulid, test, and deployment automation. 北京 : 人民邮电出版社, 2011.
2. Beck K, Andres C. Extreme programming explained: embrace change[M]// Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004.
3. Martin Fowler, Matthew Foemmel. Continuous Integration (original version) [EB/OL]. http://martinfowler.com/articles/originalContinuousintegration.html, 2000.
4. Duvall P , SteveMatyas, AndrewGlover. 持续集成软件质量改进和风险降低之道[M]. 电子工业出版社, 2012.
5. 董越. 软件集成策略[M]. 电子工业出版社, 2013.
6. 董越. 未雨绸缪:理解软件配置管理[M]. 电子工业出版社, 2012.
7. Royce W W. Managing the development of large software systems: concepts and techniques[C]// 1987.
8. Conradi H , Fuggetta A . Improving software process improvement[J]. IEEE Software, 2002, 19(4):92-99.
9. IvarJacobson, GradyBooch, JamesRumbaugh. 统一软件开发过程[M]. 机械工业出版社, 2002.
10. 拉尔曼. 敏捷迭代开发:管理者指南[M]. 中国电力出版社, 2004.
11. 惠特克. Google软件测试之道 : 像Google一样进行软件测试. 北京 : 人民邮电出版社, 2013.
12. 贝克. 测试驱动开发 : 实战与模式解析 : by example. 北京 : 机械工业出版社, 2013
13. 巴顿. 软件测试 : 第2版. 北京 : 机械工业出版社, 2006.
14. 王宁. 基于Jenkins的持续集成系统的设计与实现[D]. 北京邮电大学, 2014.
15. 卞孟春. 基于Jenkins的持续集成方案设计与实现[D]. 中国科学院大学（工程管理与信息技术学院）, 2014.
16. 徐仕成. 持续集成在现代软件开发中的应用与研究[D]. 中南大学, 2007.

# 附录

## 英文文献

**Continuous Integration**

Software development is full of best practices which are often talked about but seem to be rarely done. One of the most basic, and valuable, of these is a fully automated build and test process that allows a team to build and test their software many times a day. The idea of a daily build has been talked about a lot. McConnnell recommends it as a best practice and it's been long known as a feature of the Microsoft development approach. We agree with the XP community, however, in saying that daily builds are a minimum. A fully automated process that allows you to build several times a day is both achievable and well worth the effort.

We are using the term Continuous Integration, a term used as one of the practices of XP (Extreme Programming). However we recognize that the practice has been around for a long time and is used by plenty of folks that would never consider XP for their work. We've been using XP as a touchstone in our software development process and that influences a lot of our terminology and practices. However you can use continuous integration without using any other parts of XP - indeed we think it's an essential part of any competent software development activity.

There are several parts to making an automated daily build work.

* Keep a single place where all the source code lives and where anyone can obtain the current sources from (and previous versions)
* Automate the build process so that anyone can use a single command to build the system from the sources
* Automate the testing so that you can run a good suite of tests on the system at any time with a single command
* Make sure anyone can get a current executable which you are confident is the best executable so far.

All of this takes a certain amount of discipline. We've found that it takes a lot of energy to introduce it into a project. We've also found that once installed, it doesn't take too much effort to keep it up.

**The More Often the Better**

There's a fundamental counter-intuitive effect at the center of continuous integration. It is that it is better to integrate often than to integrate rarely. For those that do it, this is obvious; but for those that haven't this seems like a contradiction to direct experience.

If you integrate only occasionally, such as less than daily, then integration is a painful exercise, one that takes a lot of time and energy. Indeed it's sufficiently awkward that the last thing you want to do is do it more frequently. The comment we often hear is "on a project this big, you can't do a daily build."

Yet there are projects that do. We build a couple of dozen times a day on a code base around two hundred thousand lines of code worked on by a team of fifty people. Microsoft does daily builds on projects with tens of millions of lines of code.

The reason that this is possible is that the effort of integration is exponentially proportional to the amount of time between integrations. Although we're not aware of any measurements for this, this means that integrating once a week does not take five times as long as integrating once a day, but more like twenty-five times as long. So if your integration is painful, you shouldn't take this as a sign that you can't integrate more frequently. Done well, more frequent integration should be painless and you end up spending much less time carrying out the integration.

A key for this is automation. Most of integration can, and should, be done automatically. Getting the sources, compiling, linking, and significant testing can all be done automatically. At end you should be left with a simple indication of whether the build worked: yes or no. If yes you ignore it, if no you should be able to easily undo the last change to the configuration and be certain the build will work this time. No thought should be required to get a working build.

With an automated process like this, you can build as frequently as you like. The only limitation is the amount of time it takes to do the build.

**What Is a Successful Build?**

An important thing to decide is what makes a successful build. It may seem obvious, but it's remarkable how this can get muddy. Martin once reviewed a project. He asked if the project did a daily build and was answered in the affirmative. Fortunately Ron Jeffries was there to probe further. He asked the question "what do you do with build errors?" The response was "we send an e-mail to the relevant person". In fact the project hadn't succeeded in a build for months. That's not a daily build, that's a daily build attempt.

We are pretty aggressive about what we mean by a successful build.

* All the latest sources are checked out of the configuration management system
* Every file is compiled from scratch
* The resulting object files (Java classes in our case) are linked and deployed for execution (put into jars).
* The system is started and suite of tests (in our case, around 150 test classes) is run against the system.
* If all of these steps execute without error or human intervention and every test passes, then we have a successful build

Most people consider a compile and link to be a build. At the least we think a build should include starting the application and running some simple tests against it (McConnnell used the term "smoke test": switch it on and see if smoke comes out). Running a more exhaustive set of tests greatly improves the value of continuous integration, so we prefer to do that as well.

**Single Source Point**

In order to integrate easily, any developer needs to be able to get a full set of current sources easily. There's nothing worse than having to go around to different developers to ask for the latest bits and then having to copy them over, figure out where to put them, all this before you can begin the build.

The standard is simple. Anyone should be able to bring a clean machine, connect it to the network, and with a single command bring down every source file that's needed to build the system under development.

The obvious (we hope) solution is to use a configuration management (source control) system as the source of all code. Configuration management systems are usually designed to be used over a network and have the tools that allow people to easily get hold of sources. Furthermore they also include version management so you can easily find previous versions of various files. Cost shouldn't be an issue as CVS is an excellent open-source configuration management tool.

For this to work all source files should be kept in the configuration management system. All is often more than people think. It also includes build scripts, properties files, database schema DDLs, install scripts, and anything else that's needed to build on a clean machine. Too often we've seen code controlled, but not some other vital file that has to be found.

Try to ensure everything is under a single source tree in the configuration management system. Sometimes people use separate projects in the configuration management system for different components. The trouble with this is then people have to remember which versions of which components work with which versions of other components. For some situations you have to separate the sources, however these situations are much rarer than you think. You can build multiple components from a single tree of sources, such issues should be handled by the build scripts, not by the storage structure.

**Automated Build Scripts**

If you're writing a small program with a dozen or so files, then building the application may be just a matter of a single command to a compiler: javac \*.java. Bigger projects need rather more. In these cases you have files in many directories. You need to ensure the resulting object code is in the proper place. As well as compilations there may be link steps. You have code generated from other files that needs to be generated before you can compile. Tests need to be run automatically.

A big build often takes time, you don't want to do all of these steps if you've only made a small change. So a good build tool analyzes what needs to be changed as part of the process. The common way to do this is to check the dates of the source and object files and only compile if the source date is later. Dependencies then get tricky: if one object file changes those that depend on it may also need to be rebuilt. Compilers may handle this kind of thing, or they may not.

Depending on what you need, you may need different kinds of things to be built. You can build a system with or without test code, or with different sets of tests. Some components can be built stand-alone. A build script should allow you to build alternative targets for different cases.

Once you get past a simple command line, scripts often handle the load. These might be shell scripts or use a more sophisticated scripting language such as perl or python. But soon it makes sense to use an environment designed for this kind of thing, such as the make tools in Unix.

In our Java development we quickly discovered we needed a more serious solution. Indeed Matt put a good bit of time into developing a build tool called Jinx which was designed for Enterprise Java work. Recently, however, we have switched over to the open-source build tool Ant. Ant has a very similar design to Jinx, allowing us to compile Java files and package them into Jars. It also makes it easy for us to write our own extensions to Ant to allow us to do other tasks within the build.

Many of us use IDEs, and most IDEs have some kind of build management process within them. However these files are always proprietary to the IDE and often fragile. Furthermore they need the IDE to work. IDE users set up their own project files and use them for individual development. However we rely on Ant for main builds and the master build is run on a server using Ant.

**Self-Testing Code**

Just getting a program to compile is not enough. While compilers in strongly typed languages can spot many problems, there are too many errors that even a successful compile lets through. To help track these down we put a lot of emphasis on an automated testing discipline - another practice that's advocated by XP.

XP divides tests into two categories: unit tests and acceptance (also called functional) tests. Unit tests are written by developers and typically test an individual class or small group of classes. Acceptance tests are usually written by customers or an outside test group (with the help of developers) and test the whole system end-to-end. We use both kinds of test, and automate both kinds of test as much as possible.

As part of the build we run a suite of tests that we call the "BVT" (Build Verification Tests). All the tests in the BVT must pass in order for us to have a successful build. All of the XP-style unit tests are in the BVT. As this article is about the build process, we'll mostly talk about the BVT here, just bear in mind that there is a second line of testing in addition to what's in the BVT, so don't judge the whole testing and QA effort by the BVT alone. Indeed our QA group doesn't ever see code unless it's passed the BVT since they only work on working builds.

The basic principle is that when developers are writing code they also write tests for that code. When they complete a task, not just do they check in the production code, they also check in the tests for that code. Those that follow XP closely use the test first style of programming: you shouldn't write any code until you have a failing test. So if you want to add a new feature to the system, you first write a test that will only work if the feature is there, then you make that test work.

We write the tests in Java, the same language that we're developing in. This makes writing tests just the same as writing code. We use JUnit as the framework for organizing and writing tests. JUnit is a simple framework that allows you to quickly write tests, organize them into suites, and run suites interactively or in batch mode. (JUnit is the Java member of the xUnit family - for which there are versions for almost every language.)

Developers typically run some subset of the unit tests with every compile as they are writing the software. This actually speeds up the development work since the tests help to find any logic errors in the code you're working on. Then, rather than debugging, you can look at the changes since you last ran the tests. Those changes should be small and thus it's a lot easier to find the bug.

Not everyone works strictly in the XP test-first style, but the key benefit comes from writing the tests at the same time. As well as making an individual task go faster, it also builds up the BVT making it more likely to catch errors. Since the BVT runs several times a day, this means that any problems that the BVT detects are easier to find for the same reason: we can look at a small amount of changed code in order to find the bug. This debugging by looking at the changed code is often much more effective than debugging by stepping though running code.

Of course you can't count on tests to find everything. As it's often been said: tests don't prove the absence of bugs. However perfection isn't the only point at which you get payback for a good BVT. Imperfect tests, run frequently, are much better than perfect tests that are never written at all.

A related question is the issue of developers writing tests on their own code. It's often been said that people should not test their own code, because it's too easy to overlook errors in your own work. While this is true, the self-testing process requires a rapid turn-around of tests into the code base. The value of that fast turn-around is greater than the value of separate testers. So for the BVTs we rely on developer-written tests, but there are separate acceptance tests which are written independently.

Another important part of self-testing is to improve the quality of tests with feedback - a key value of XP. Feedback here comes in the form of bugs that escaped the BVT. The rule here is that you aren't allowed to fix a bug until you have a failing unit test in the BVT. This way every time you fix a bug, you also add a test to ensure it doesn't slip past you again. Furthermore this test should lead you to think of other tests that need to be written to strengthen the BVT.

**The Master Build**

Build automation makes a lot of sense for individual developers, but where it really shines is in producing a master build for the whole team. We've found that having a master build procedure brings the team together and makes it easier to find integration problems early.

The first step is to choose a machine to run the master builds. We use Trebuchet (we played Age of Empires a lot) a four processor server which is pretty much dedicated to the build process. (In the early days when builds took a long time this horse-power was essential.)

The build process is in a Java class that's always running. If there's no build going on the build process sits in a while loop checking the repository every few minutes. If nobody's checked in any code since the last build, it continues to wait. If there's new code in the repository, then it starts building.

The first stage of the build is to do a full check out from the repository. Starteam provides a pretty good API for Java, so it was straightforward to hook into the repository. The build daemon looks at the repository at five minutes before the current time and sees if anyone checked in during that last five minutes. If so it considers it safe to check out the code as of five minutes ago (this prevents checking out in the middle of someone checking in, without locking the repository.)

The daemon checks out to a directory on Trebuchet. Once all is checked out the daemon then invokes the ant script in the directory. Ant then takes over to do a full build. We do a full build from all the sources. The ant script goes as far as compiling and dividing the resulting class files into half a dozen jars for deploying into the EJB server.

Once Ant has done the compiling and deployment, the build daemon starts the EJB server with new jars and executes the BVT test suite. The tests run, if all pass then we have a successful build. The build daemon then goes back into Starteam and labels the sources that were checked out with a build number. It then looks to see if anyone's checked in while it was building, if so it starts another build. If not the daemon goes back into its while loop and waits for the next check in.

At the end of the build, the build daemon sends an e-mail to all developers that had newly checked in code with that build. The e-mail summarizes the status of that build. It's considered bad form to leave the building after checking in code until you've got that e-mail.

The daemon writes a log of all its steps to an XML log file. A servlet runs on Trebuchet which allows anyone to see the state of the build by inspecting the log.

The screen shows if there's a build currently running and if so when it started. Down the left there is a history of all the builds, whether successful or not. Clicking on a build shows the details of that build: whether it compiled, status of tests, what changes were made, etc.

We've found that many developers keep a regular eye on this web page. It gives them a sense of what's going on in the project and on what's getting altered as people check in. At some point we may put other project news on that page, but we don't want it to lose its relevance.

It's important that any developer can simulate the master build locally on their own machine. That way if an integration error does occur, a developer can investigate and debug the problem on their own machine without tying up the master build process. Furthermore the developer can run the build locally before checking in to reduce the chances of the master build failing.

There's a reasonable question here about whether the master build should be a clean build, i.e. all from sources only, or an incremental build. Incremental builds can be a lot faster, but they also introduce the risk of a problem sneaking because something didn't get compiled. It also has the risk of us not being able to recreate a build. Our builds are pretty quick (about 15 minutes for ~200KLOC) so we're happy with doing a clean build each time. However some shops like to do incremental builds most of the time, but a regular clean build (at least daily) in case those odd errors crop up.

**Checking in**

Using an automated build means that the developers follow a certain kind of rhythm as they develop software. The most important part of that rhythm is that they integrate regularly. We've run across organizations where they do a daily build, but people don't check in frequently. If developers only check in every few weeks, then a daily build doesn't do much for you. We follow a general principle that every developer checks code in roughly once a day.

Before starting on a new task, developers should first sync with the configuration management system. This will mean that their local copies of the sources are up to date. Writing code on top of out-of-date sources only leads to trouble and confusion.

The developer then works on the task updating whichever files need changing. A developer may integrate when a task is done, or part way through a task, but all tests need to run fine in order to integrate.

The first part of integration is to resync the developers working copies with the repository. Any files changed on the repository are copied into the working directory and the configuration management system warns the developer of any conflicts. The developer then needs to build to the synchronized working set and run the BVT successfully on those files.

Now the developer can commit the new files to the repository. Once that's done the developer needs to wait for the master build. If the master build succeeds, then the check in succeeded. If not the developer can fix the problem and commit the fix if it's straightforward. If it's more complicated then the developer needs to back out the changes, resync his working directory and get things working locally before committing again.

Some check-in processes force a serialization of the check-in process. In this situation there is a build token which only one developer can take. The developer will take the build token, resync the working copy, commit the changes, and release the token. This stops more than one developer updating the repository between builds. We've found that we rarely run into trouble without the build token, so we don't use one. Frequently more than one person commits into the same master build, but only rarely does that cause a build failure: and then it's usually easy to fix.

We also leave to the developer's judgement how careful they should be before checking in. It's up to developer how likely they think it is that there will be an integration error. If she thinks it is likely, then she does a local build first before checking in, if she thinks an integration error is unlikely then she will just check in. If she's wrong she'll find out once the master build runs and then she'll have to back out her changes and figure out what went wrong. You can tolerate errors providing they are quick to find and easy to remove.

**Summing up**

Developing a disciplined and automated build process is essential to a controlled project. Many software gurus say that, but we've found that it's still a rarity in the field.

The key is to automate absolutely everything and run the process so often that integration errors are found quickly. As a result everyone is more prepared to change things when they need to, because they know that if they do cause an integration error, it's easy to find and fix. Once you have these benefits, you'll find they are such that you'll kick and scream before you give them up.

## 文献翻译

**持续集成**

软件开发有很多最佳实践，他们经常被讨论但似乎很少被使用。其中最基本，最有价值的是完全自动化的构建和测试过程，他允许团队每天多次构建和测试他们的软件。关于每日构建的想法已经被讨论了很多，McConnnell建议将其作为最佳实践，并且长期以来一直被称为微软开发方法的一个特性。但是，我们同意XP社区的说法，即每日构建是最低限度的。一个完全自动化的过程，允许您每天建立几次，这是可以实现的，值得为此付出努力。

我们使用续集成用作XP（极限编程）的一种实践。但是我们认识到这种做法已经存在了很长时间，很多人都会使用这种做法，因为他们的工作从未考虑过XP。我们一直在使用XP作为我们软件开发过程中的试金石，这会影响我们的许多术语和实践。但是，您可以在不使用XP的任何其他部分的情况下使用持续集成，事实上我们认为它是任何有能力的软件开发活动的重要组成部分。

制作自动日常构建工作有如下几个部分：

* 保留所有源代码所在的单个位置以及任何人可以从（以及之前的版本）获取当前源的位置
* 自动化构建过程，以便任何人都可以使用单个命令从源构建系统
* 自动化测试，以便可以使用单个命令随时在系统上运行一系列测试
* 确保任何人都可以获得当前可执行文件，确信它是目前为止最好的可执行文件。

所有这些都需要一定的顺序。我们发现将它引入项目需要花费很多精力。我们还发现，一旦安装，就不需要花太多精力来保持它。

**持续集成的优点**

关于持续集成最难表达的一点是，从根本上转向整个开发模式，如果你从未在一个实践它的环境中工作，那么这个模式就不容易看到。事实上，如果他们独自工作，大多数人确实会看到这种氛围 - 因为那时他们只与自己融为一体。对于许多人来说，团队发展只会带来某些问题，这些问题属于领土的一部分。持续集成可以减少这些问题，以换取一定的纪律。

持续整合的根本好处在于它消除了人们花时间寻找错误的会话，其中一个人的工作踩到了别人的工作而没有任何人意识到发生了什么。这些错误很难找到，因为问题不在一个人的区域，而在于两件作品之间的相互作用。这个问题因时间而恶化。通常，集成错误可以在它们首次出现之前的数周或数月内插入。结果他们发现了很多。

通过持续集成，绝大多数此类错误在它们被引入的同一天显现出来。此外，至少有一半的互动在哪里显而易见。这大大减少了搜索bug的范围。如果你找不到这个bug，就可以避免将违规代码放入产品中，所以最糟糕的情况是你不添加同时添加bug的功能。（当然，您可能希望该功能比您讨厌该错误更多，但至少这种方式是一个明智的选择。）

现在无法保证您获得所有集成错误。该技术依赖于测试，并且我们都知道测试并不能证明没有错误。关键点在于，持续集成可以捕获足够的错误以获得成本。

所有这些的最终结果是通过减少追逐集成错误所花费的时间来提高生产力。虽然我们不知道任何接近科学研究的人，但轶事证据非常强烈。持续集成可以减少在集成地狱中花费的时间，事实上它可以将地狱变成非事件。

**构建次数越多越好**

在持续集成的中心存在一种基本的反直觉效应。这是经常整合比很少整合更好。对于那些这样做的人来说，这是显而易见的; 但对于那些没有这个似乎与直接经验相矛盾的人。

如果你只是偶尔整合，比如每天不到，那么整合是一项痛苦的练习，需要花费大量的时间和精力。事实上，你想要做的最后一件事是更频繁地做到这一点是非常尴尬的。我们经常听到的评论是“在一个如此大的项目中，你无法做到日常构建。”

然而，有些项目可以做到。我们每天在代码库上构建几十次，大约有二十万行代码由一个由五十人组成的团队工作。微软每天都会在拥有数千万行代码的项目上构建。

这是可能的原因是积分的努力与积分之间的时间量成指数比例。虽然我们不知道对此进行任何测量，但这意味着每周进行一次积分所需的时间不会是每天进行一次积分的五倍，但更长的时间是二十五倍。因此，如果您的集成很痛苦，那么您不应将此视为无法更频繁集成的标志。做得好，更频繁的集成应该是无痛的，并且最终花费更少的时间进行集成。

关键是自动化。大多数集成都可以而且应该自动完成。获取源代码，编译，链接和重要测试都可以自动完成。最后，您应该留下一个简单的指示，表明构建是否有效：是或否。如果是，则忽略它，如果不是，您应该能够轻松撤消对配置的最后更改，并确保此次构建将起作用。不需要考虑获得有效的构建。

通过这样的自动化流程，您可以根据需要随时进行构建。唯一的限制是进行构建所需的时间。

**什么是成功的构建**

一个重要的事情是成功构建的原因。这似乎是显而易见的，但是如何变得混乱是非常了不起的。马丁曾经审查过一个项目。他询问该项目是否每天都进行了建设并得到了肯定的回答。幸运的是Ron Jeffries在那里进一步探讨。他问了一个问题“你如何处理构建错误？” 回复是“我们向相关人员发送电子邮件”。事实上，该项目几个月来一直没有成功。这不是每日构建，这是每日构建尝试。

成功构建的意义十分重要：

* 从配置管理系统中检出所有最新的源
* 每个文件都是从头开始编译的
* 生成的目标文件（在我们的例子中是Java类）被链接和部署以便执行（放入jar）。
* 启动系统并对系统运行测试套件（在我们的例子中，大约150个测试类）。
* 如果所有这些步骤都在没有错误或人为干预的情况下执行，并且每个测试都通过，那么我们就有了成功

大多数人认为编译和链接是一个构建。至少我们认为构建应该包括启动应用程序并针对它运行一些简单的测试（McConnnell使用术语“冒烟测试”：打开它并查看烟雾是否出来）。运行更详尽的测试集可以大大提高持续集成的价值，因此我们也希望这样做。

**唯一源码库**

为了轻松集成，任何开发人员都需要能够轻松获得全套的当前资源。没有什么比不得不绕过不同的开发人员要求最新的比特，然后不得不复制它们，找出放置它们的地方，所有这一切都可以开始构建之前。

标准很简单。任何人都应该能够使用干净的机器，将其连接到网络，并使用单个命令来删除构建正在开发的系统所需的每个源文件。

显而易见的（我们希望）解决方案是使用配置管理（源代码控制）系统作为所有代码的源。配置管理系统通常设计为通过网络使用，并具有允许人们轻松获取源的工具。此外，它们还包括版本管理，因此您可以轻松找到各种文件的先前版本。成本不应该是一个问题，因为CVS是一个优秀的开源配置管理工具。

为此，所有源文件都应保存在配置管理系统中。所有这些往往超出人们的想象。它还包括构建脚本，属性文件，数据库模式DDL，安装脚本以及在干净机器上构建所需的任何其他内容。我们经常看到代码控制，但不是必须找到的其他重要文件。

尝试确保配置管理系统中的所有内容都位于单个源树下。有时，人们在配置管理系统中为不同的组件使用单独的项目。这样做的麻烦就是人们必须记住哪些组件的哪些版本与哪些版本的其他组件一起使用。在某些情况下，您必须将源分开，但这些情况比您想象的要少得多。您可以从单个源代码树构建多个组件，这些问题应该由构建脚本处理，而不是由存储结构处理。

**自动构建脚本**

如果您正在编写一个包含十几个文件的小程序，那么构建应用程序可能只是对编译器的单个命令：javac \*.java。更大的项目需要更多。在这些情况下，您有许多目录中的文件。您需要确保生成的对象代码位于正确的位置。除了汇编之外，还可能存在链接步骤。您可以在编译之前生成其他需要生成的文件生成的代码。测试需要自动运行。

大型构建通常需要时间，如果您只进行了一些小改动，则不希望执行所有这些步骤。因此，一个好的构建工具可以分析需要在过程中进行更改的内容。执行此操作的常用方法是检查源文件和目标文件的日期，并仅在源日期晚些时进行编译。然后依赖性变得棘手：如果一个目标文件发生了变化，那么依赖它的那些也可能需要重建。编译器可以处理这种事情，或者他们可能不会。

根据您的需要，您可能需要构建不同类型的东西。您可以使用或不使用测试代码或使用不同的测试集来构建系统。某些组件可以独立构建。构建脚本应该允许您为不同的情况构建替代目标。

一旦通过一个简单的命令行，脚本通常会处理负载。这些可能是shell脚本或使用更复杂的脚本语言，如perl或python。但很快就可以使用专为此类设计的环境，例如Unix中的make工具。

在我们的Java开发中，我们很快发现我们需要更严肃的解决方案。事实上，Matt花了很多时间来开发一个名为Jinx的构建工具，该工具是为Enterprise Java工作而设计的。然而，最近我们已经切换到开源构建工具Ant。Ant与Jinx的设计非常相似，允许我们编译Java文件并将它们打包成Jars。它还使我们可以轻松编写自己的Ant扩展，以允许我们在构建中执行其他任务。

我们中的许多人使用IDE，并且大多数IDE在其中都有某种构建管理过程。但是，这些文件始终是IDE专有的，并且通常很脆弱。此外，他们需要IDE才能工作。IDE用户设置自己的项目文件并将其用于个人开发。但是我们依赖Ant来进行主构建，而主构建则使用Ant在服务器上运行。

**自检代码**

刚刚获得编译程序是不够的。虽然强类型语言的编译器可以发现许多问题，但是即使成功编译也会有太多错误。为了帮助跟踪这些问题，我们非常重视自动化测试学科 - 这是XP提倡的另一种做法。

XP将测试分为两类：单元测试和接受（也称为功能）测试。单元测试由开发人员编写，通常测试单个类或一小组类。验收测试通常由客户或外部测试组（在开发人员的帮助下）编写，并对端到端的整个系统进行测试。我们使用两种测试，并尽可能自动化两种测试。

作为构建的一部分，我们运行一系列测试，我们称之为“BVT”（构建验证测试）。BVT中的所有测试必须通过才能使我们成功构建。所有XP风格的单元测试都在BVT中。由于本文是关于构建过程的，我们在这里大多谈论BVT，请记住，除了BVT中的内容之外还有第二行测试，所以不要判断整个测试和QA工作量仅由BVT提供。事实上，我们的QA小组看不到代码，除非它通过了BVT，因为它们只用于工作版本。

基本原则是，当开发人员编写代码时，他们也会为该代码编写测试。当他们完成一项任务时，他们不仅会检查生产代码，还会检查该代码的测试。那些遵循XP的人会密切使用测试的第一种编程方式：在 测试失败之前，不应该编写任何代码。因此，如果要向系统添加新功能，首先要编写一个仅在功能存在的情况下才能运行的测试，然后进行测试。

我们用Java编写测试，这与我们正在开发的语言相同。这使得编写测试与编写代码完全相同。我们使用JUnit作为组织和编写测试的框架。JUnit是一个简单的框架，允许您快速编写测试，将它们组织到套件中，并以交互方式或批处理模式运行套件。（JUnit是xUnit系列的Java成员- 几乎所有语言都有这些版本。）

开发人员通常在编写软件时对每个编译运行单元测试的一些子集。这实际上加快了开发工作，因为测试有助于在您正在处理的代码中找到任何逻辑错误。然后，您可以查看自上次运行测试以来的更改，而不是调试。这些更改应该很小，因此找到错误要容易得多。

不是每个人都严格按照XP测试优先的方式工作，但关键的好处来自于同时编写测试。除了使单个任务更快，它还构建了BVT，使其更容易发现错误。由于BVT每天运行几次，这意味着由于同样的原因，BVT​​检测到的任何问题都更容易找到：我们可以查看少量更改的代码以找到错误。通过查看更改的代码进行调试通常比通过逐步运行代码进行调试更有效。

当然，你不能指望测试找到一切。正如经常说的那样：测试不能证明没有错误。然而，完美并不是获得良好BVT投资回报的唯一方面。经常运行的不完美测试比完全没有编写的完美测试要好得多。

一个相关的问题是开发人员在自己的代码上编写测试的问题。人们经常说人们不应该测试自己的代码，因为在你自己的工作中忽略错误太容易了。虽然这是事实，但自我测试过程需要将测试快速转换为代码库。快速周转的价值大于单独测试人员的价值。因此，对于BVT，我们依赖于开发人员编写的测试，但是有单独的验收测试是独立编写的。

自我测试的另一个重要部分是通过反馈提高测试质量 - 这是XP的关键价值。这里的反馈是以逃脱BVT的错误的形式出现的。这里的规则是，在BVT中进行单元测试失败之前，不允许修复错误。这样每次修复错误时，您还会添加一个测试，以确保它不会再次滑过您。此外，该测试应该引导您考虑需要编写的其他测试以加强BVT。

**主构建**

构建自动化对于个体开发人员来说非常有意义，但是它真正发挥作用的是为整个团队生成主构建。我们发现拥有一个主构建过程可以将团队聚集在一起，并且可以更容易地及早发现集成问题。

第一步是选择一台机器来运行主构建。我们使用Trebuchet（我们玩过很多帝国时代）一个四处理器服务器，它非常专注于构建过程。（在构建需要很长时间的早期阶段，这种马力是必不可少的。）

构建过程位于始终运行的Java类中。如果构建过程中没有构建，则会在while循环中每隔几分钟检查一次存储库。如果自上次构建以来没有人签入任何代码，它将继续等待。如果存储库中有新代码，则会开始构建。

构建的第一阶段是从存储库中进行全面检查。Starteam为Java提供了一个非常好的API，因此可以直接挂钩到存储库。构建守护程序在当前时间前五分钟查看存储库，并查看是否有人在最后五分钟内签入。如果是这样，它认为从五分钟前检查代码是安全的（这可以防止在某人登记时检查，而不会锁定存储库。）

守护进程检出Trebuchet上的目录。一旦检出所有守护进程，然后调用目录中的ant脚本。然后Ant接管完整的构建。我们从所有来源完整构建。ant脚本可以将生成的类文件编译并分成六个用于部署到EJB服务器的jar文件。

一旦Ant完成编译和部署，构建守护程序将使用新jar启动EJB服务器并执行BVT测试套件。测试运行，如果全部通过，那么我们有一个成功的构建。然后，构建守护程序返回到Starteam并标记使用内部版本号签出的源。然后它会查看是否有人在构建时检入，如果是，则启动另一个构建。如果不是，守护进程返回其while循环并等待下一次检入。

在构建结束时，构建守护程序会向所有已使用该构建重新签入代码的开发人员发送电子邮件。电子邮件总结了该构建的状态。在您收到电子邮件之后，在检查代码后离开建筑物被认为是不好的形式。

守护程序将其所有步骤的日志写入XML日志文件。一个servlet在Trebuchet上运行，允许任何人通过检查日志来查看构建的状态。

屏幕显示当前是否正在运行构建，如果是，则在它开始时运行。左下方有所有构建的历史，无论是否成功。单击构建将显示该构建的详细信息：是否已编译，测试状态，所做的更改等。

我们发现很多开发人员都会定期关注这个网页。它让他们了解项目中发生的事情以及人们检查时会发生什么变化。在某些时候我们可能会在该页面上放置其他项目新闻，但我们不希望它失去相关性。

任何开发人员都可以在自己的计算机上本地模拟主构建，这一点非常重要。这样，如果确实发生了集成错误，开发人员可以在自己的计算机上调查和调试问题，而无需占用主构建过程。此外，开发人员可以在签入之前在本地运行构建，以减少主构建失败的可能性。

这里有一个合理的问题，即主构建是否应该是一个干净的构建，即所有来自源，还是增量构建。增量构建可以快得多，但它们也会引入问题偷偷摸摸的风险，因为某些东西没有被编译。它还存在我们无法重新构建构建的风险。我们的构建非常快（大约15分钟，大约200KLOC），所以我们很高兴每次都做一个干净的构建。然而，一些商店喜欢在大多数情况下进行增量构建，而是定期清理构建（至少每天）以防出现奇怪的错误。

**签到**

使用自动构建意味着开发人员在开发软件时遵循某种节奏。这种节奏最重要的部分是他们定期融合。我们遇到了他们每天构建的组织，但人们不经常检查。如果开发人员每隔几周只检查一次，那么每日构建对您来说并没有太大作用。我们遵循一般原则，即每个开发人员大约每天检查一次代码。

在开始新任务之前，开发人员应首先与配置管理系统同步。这意味着他们的本地资源副本是最新的。在过时的资源上编写代码只会导致麻烦和混乱。

然后开发人员处理任务更新，无论哪个文件需要更改。开发人员可以在任务完成时进行集成，也可以在任务中途进行集成，但所有测试都需要运行良好才能进行集成。

集成的第一部分是重新同步开发人员使用存储库工作副本。存储库中更改的所有文件都将复制到工作目录中，配置管理系统会警告开发人员任何冲突。然后，开发人员需要构建同步工作集并在这些文件上成功运行BVT。

现在，开发人员可以将新文件提交到存储库。完成后，开发人员需要等待主构建。如果主构建成功，则检入成功。如果不是，开发人员可以解决问题并提交修复，如果它是直截了当的。如果它更复杂，那么开发人员需要退出更改，重新同步他的工作目录并在再次提交之前让事情在本地工作。

一些签入过程会强制执行签入过程的序列化。在这种情况下，只有一个开发人员可以使用构建令牌。开发人员将获取构建令牌，重新同步工作副本，提交更改并释放令牌。这会阻止多个开发人员在构建之间更新存储库。我们发现没有构建令牌我们很少遇到麻烦，所以我们不使用它。通常不止一个人进入同一个主构建，但很少会导致构建失败：然后通常很容易修复。

我们还会向开发人员判断在签入之前他们应该多么小心。这取决于开发人员他们认为存在集成错误的可能性。如果她认为这很可能，那么她在登记之前首先进行本地构建，如果她认为不太可能出现整合错误，那么她就会检查。如果她错了，她会在主要构建运行后找到，然后她会'我必须退出她的变化并弄清楚出了什么问题。您可以容忍错误，只要它们可以快速找到并且易于删除。

**总结**

开发一个规范和自动化的构建过程对于受控项目至关重要，许多软件专家都这样提到，但我们发现它在该领域仍然很少见。

主要关键是要完全自动化所有事情，并经常运行流程，以便快速找到集成错误。因此，每个人都更愿意在需要时更改内容，因为他们知道当确实会导致集成错误时，很容易找到错误并修复。一旦在这种情况下受益，你会发现项目开发更加容易。