使用配置管理工具进行持续集成开发

1. 持续集成与现代软件开发

早期的软件开发模式是由程序员负责编写不同的模块，在软件项目完成之前，一次性的把各个模块集成在一起，再进行测试。我们称这种集成方式为“big-bang”的集成方式。使用该种集成方式的项目团队把软件集成安排在开发阶段的后期，一般是应用“瀑布式（Cascade）”开发模式。在项目后期才开始对软件进行集成，会为项目引入很多的未知因素和巨大风险－－程序员往往发现越来越多的 Bug 等待他们去修复。这种集成方式很有可能会威胁到软件项目的成功。随着市场竞争的日益激烈，对软件产品的发布要求越来越高、越来越频繁，这种“big-bang”的集成方式已经不能满足开发团队的需求。取而代之的持续集成的开发方式“Continuous integration”。持续集成可以有效地解决软件开发过程中的许多问题，可以有效的确保软件质量，减小项目的风险，使得软件开发团队从容面对各种各样的变化。

持续集成在现代软件开发中扮演着非常重要的角色。使用持续集成策略的团队，可以发现持续集成可以为软件开发带来如下好处：

第一：可以尽早发现由于软件集成所带来的 Bug，及时进行更正。原因在于由软件集成而引入的 Bug 一般涉及到两个或多个程序员书写的代码，相对来说比较难于 Debug。如果不及早进行集成，就不能发现该 Bug；一旦集成之后，发现该 Bug，就要花费大量的资源来进行 Debug。但是如果使用持续集成，这种 Bug 在被引入的初始阶段就能够被发现，程序员可以检查相对少的源代码就可以更改该 Bug，这时所付出的代价是最小的。

第二：还可以有效的避免程序员在错误的路线上越走越远，以至于在项目后期为其付出巨大的代价。

> 举例来说：Tom 和 Jerry 分别工作在不同的模块 A 和 B 上，其中 A 模块需要依赖 B 模块中的代码。如果 B 模块中的程序有错误，则会影响到 A 模块。如果 Jerry 在开发 B 模块的过程中引入了 Bug，使得 Tom 开发 A 模块是工作在一个错误的基础上。如果两人的程序长时间不进行集成，则 Tom 就会在错误的开发路线上越走越远。

持续集成可以有效的降低由于软件集成所带来的风险。目前已经作为许多流行的软件开发理论的基础组成部分，例如 XP，Staging Delivery，RUP 中的迭代开发等等。许多软件公司都使用“Nightly Build”，或“Daily Build”等方式来强制程序员每天至少进行一次集成（Microsoft 就是使用 Daily Build 的方式进行持续集成的）。实践已经证明，持续集成对于提高软件质量，针对软件开发项目进行有效的风险管理有着不可替代的作用。

持续集成的意义在于使项目随时具有一个明确的“最近状态”。项目团队中的成员的所有工作都是建立在该状态之上的：程序员基于该状态编写代码；测试人员针对该状态进行软件测试。更为重要的是项目管理人员可以根据项目的最新状态对项目的进度、风险、

资源使用情况等进行有效的评估，为最终的项目成功打下坚实的基础。

另外，现代软件开发语言（例如：C、C++，Java 等）决定了持续集成的重要性。在这些语言中，往往是由多个源程序文件来完成某一特定的功能；每一个源代码文件也往往不是只服务于某一个单一系统功能。而这些程序文件往往不是由同一个程序员编写，如果这些源代码文件不是经常的进行集成，则必然在最终的集成过程中为开发团队带来巨大的麻烦，很有可能造成 Build Break。

2. 配置管理工具对持续集成的支持

配置管理工具（版本控制工具）是任何规模的开发团队中必不可少的工具之一。使用合适的配置管理工具可以很好的帮助开发团队进行持续集成开发。在使用持续集成的团队中，代码开发、源程序检入、单元测试的过程一般如下：

Step1: 程序员建立本地工作区。从版本控制工具的存储库中得到项目的所有文件。一般来说，本地工作区和存储库具有相同的目录结构，或是存在某种特定的关联关系。
Step2: 程序员更新本地工作区，使得本地工作区中的文件都更新到最新状态。
Step3: 程序员检出源代码文件。
Step4: 程序员更改源代码，进行单元测试。
Step5: 程序员更新本地工作区，得到其他用户的更改，进行单元测试。如果有冲突发生，需要解决冲突。
Step6: 在单元测试通过之后，程序员检入源代码文件。

在这里，我们可以看到程序员在开始工作之前，首先要更新本地工作区，使得自己的开发工作是建立在项目的最新状态之上。在开发工作完成之后，代码检入之前，还要再次更新本地工作区，进行单元测试，其目的是确定本地更改不会和其他程序员的更改发生冲突，能够有效地集成在一起。之后，程序员把本地更改的文件检入到版本控制工具的中央存储库，进行持续集成，项目的最新状态也随之更新，以便其他用户使用。

由以上过程说明我们可以清楚的看到，在使用该开发模式的团队中所使用的配置管理工具必须具有以下几个特点：

- ✧ 整个项目的所有受控文件应该存储在统一的、明确的代码树中（One Source Tree）。
- ✧ 程序员使用版本控制工具可以很方便地得到当前项目的全套最新代码。
- ✧ 程序员能够方便地更新本地工作区（Workspace），得到其它人所作的更改。
- ✧ 代码树记录着当前项目的最新状态，并且可以方便地回溯受控文件的历史版本。
- ✧ 很好的支持三方归并（3-Way Merge）。

现在流行的一些配置管理工具，都可以支持上表中的的功能。这些配置管理工具包括：CVS、VSS、Firefly、ClearCase 等等。

现代软件开发过程中，“并行开发（Parallel Development）”已经成为大部分开发团队所必须接受的现实。开发团队经常需要一边开发新版本的软件，同时还要为旧版本的软件打补丁。在旧版本中已经 Fix 的 Bug 还必须归并（Merge）到新的软件发布版本中。在使用配置管理工具时，就要求该工具能够很好的在不同的开发代码线（Code Line，Code

Stream）之间进行代码的归并、集成。CVS 由于存在“重复归并（Repeat Merge）”的问题，不能很好的支持在代码线之间的归并；VSS 和 ClearCase 在支持并行开发方面，也都存在各自的问题。Firefly 是支持并行开发模式最好的配置管理工具。（注：该问题已经超出本文所要讨论的问题，因此在这里不作过多说明。）

基于“Change Package”的配置管理工具

在配置管理工具的众多产品中，有一类产品是基于“Change Package”概念的。在该类产品中，一个项目的最新状态决定于项目的“基线（Baseline）”加上一组经过挑选的“Change Package”（不同产品中，该对象的名称可能不一样，例如：Change Package、Task 等等）。每一个 Change Package 就是一个变更请求，同时包含有文件的版本变化。程序员在本地的工作过程是这样的：

Step1: 新建 Change Package
Step2: 检出需要更改的文件到 Change Package 中
Step3: 更改文件；进行单元测试
Step4: 检入 Change Package

在程序员检入 Change Package 之后，如果有文件冲突，并不要求程序员立刻解决该冲突。如果要做产品集成的话，需要由***配置管理员***去挑选不同的 Change Package 以组成不同的配置（Configuration），如果这时发现有冲突的话，要求进行解决冲突。之后，程序员再根据项目最新的“配置”来更新本地工作区。这类产品的基本结构如下图所示：

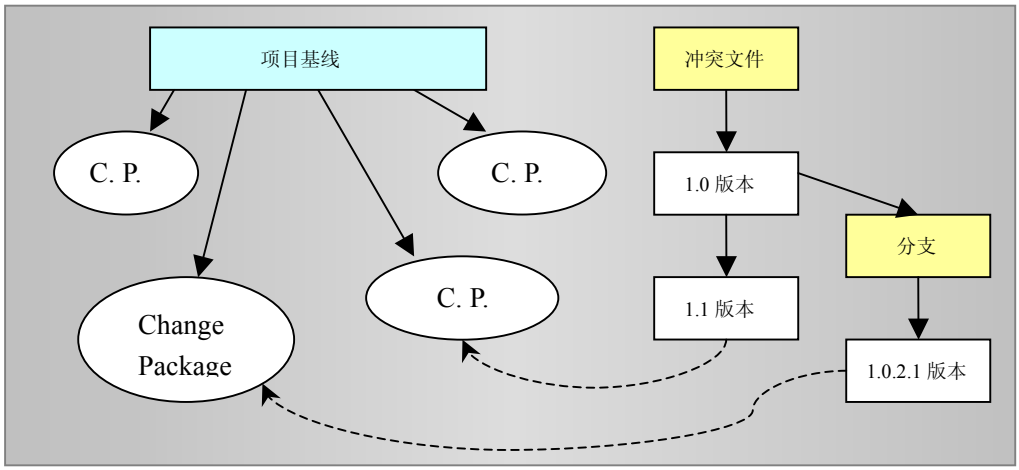

图 1：基于 Change Package 配置管理工具结构示意

这类产品的基础还是文件级别的分支，不能够很好的支持 Stream Line 的开发模式。很明显，该种产品并不能很好的支持持续集成的开发模式。因为整个配置管理工具并没有维护一个代表项目最新状态的 One Source Tree。项目的状态是由项目基线和若干的 Change Package 组成的“配置”来决定的。

Configuration ＝ Baseline + Change Package 1 + … + Change Package N

生成配置的过程是由配置管理员来执行的。要达到持续集成的目的，配置管理员必须要经常手工生成配置（因为 Change Package 是不断变化和增加的），势必要耗费大量的精

力在这项本可以***自动完成***的工作上。配置生成之后，开发者才能依据项目的最新配置更新本地工作区，集成其它程序员的更改。如果不采用持续集成的策略，在产品发布时进行 big-bang 的集成，开发团队会被过多的文件冲突耽误大量的时间，而且会引入很多由于集成引起的 Bug。对于整个项目来说，这无疑是一个灾难。在这种项目中，项目经理如果需要及时了解项目的最新状态的话，很大程度上要依赖于配置管理员的工作！

配置管理员在生成配置，挑选 Change Package 的时候，还要注意到 Change Package 之间的依赖关系。如果两个不同的 Change Package 中包含相同的文件的不同版本，一般情况下，配置管理工具可以自动识别该种依赖管理（见表 1-1）。但是，如果 Change Package 中不包含相同的文件，但是其中的文件具有程序逻辑关系，则 Change Package 还需要管理员手工配置（见表 1-2）。

CP1	CP2
Draw.c [1.0]	Test.c [1.3]
Shape.c[1.2]	Shape.c[1.1]
Circle.c[1.3]	Circle.c[1.1]

表 1-1: 在 Change Package 1 中的 Shape.c 和 Circle.c 分别是 Change Package 2 中相同文件的后续版本。如果用户选择 CP1 生成配置，则自动选择 CP2。

CP1	CP2
Draw.c [1.0]	Draw.h[1.2]
Shape.c[1.1]	Shape.h[1.1]

表 1-2: 在 Change Package 1 中的 Draw.c 和 Shape.c 与 Change Package 2 中的头文件不是同一个文件，但是在程序逻辑上具有紧密的联系，需要同时编译。这时，配置管理工具往往无法判断，需要耗费大量人力进行手工选择。

在 3-2 中，举的还是一个比较简单的例子。在现代的 OO 语言中，程序（源代码）之间存在着各种各样的依赖关系：Extend，Implement，Composite，Reference 等等。不同的文件变更还存在于不同的 Change Package 中（用户很有可能遇到的是一个发散的过程，即: CP1 依赖于 CP2 依赖于 CP3 …. 依赖于 CPn。在生成配置时候需要的 Change Package 越选越多，直到包含了项目中的所有 Change Package），如果要让配置管理员去手工挑选这些 Change Package 去生成配置，何不采用一个能够支持持续集成的配置管理工具呢？

但是，存在必然有其理由。我们发现基于 Change Package 的配置管理工具一般都是在 20 世纪 70 年代发展起来的。当时 PC 还没有流行，软件开发一般都是针对主机（MainFrame），商业软件一般使用 COBOL 语言进行开发。其特点是一个单独的源程序文件完成独立的功能，可以独立开发－编译－运行。在这种环境中发展起来的配置管理工具，使用 Change Package，可以做到非常灵活地生成配置的目的。如果一个源文件有问题，配置管理员可以方便的选择它的历史版本，或者把它排除在配置之外。但是，随着计算机软、硬件的发展，如今的主流开发语言已经是 C/C++/Java/.NET，再继续使用该种配置管理工具，无疑是不合时宜的选择。

注：目前，国内市场上基于 Change Package 的配置管理工具主要有 CCC/Harvest，Synergy/CM，PVCS Dimension。