



软件工程
第二章 敏捷开发与配置管理
2-5 软件配置管理与Git

徐汉川
xhc@hit.edu.cn

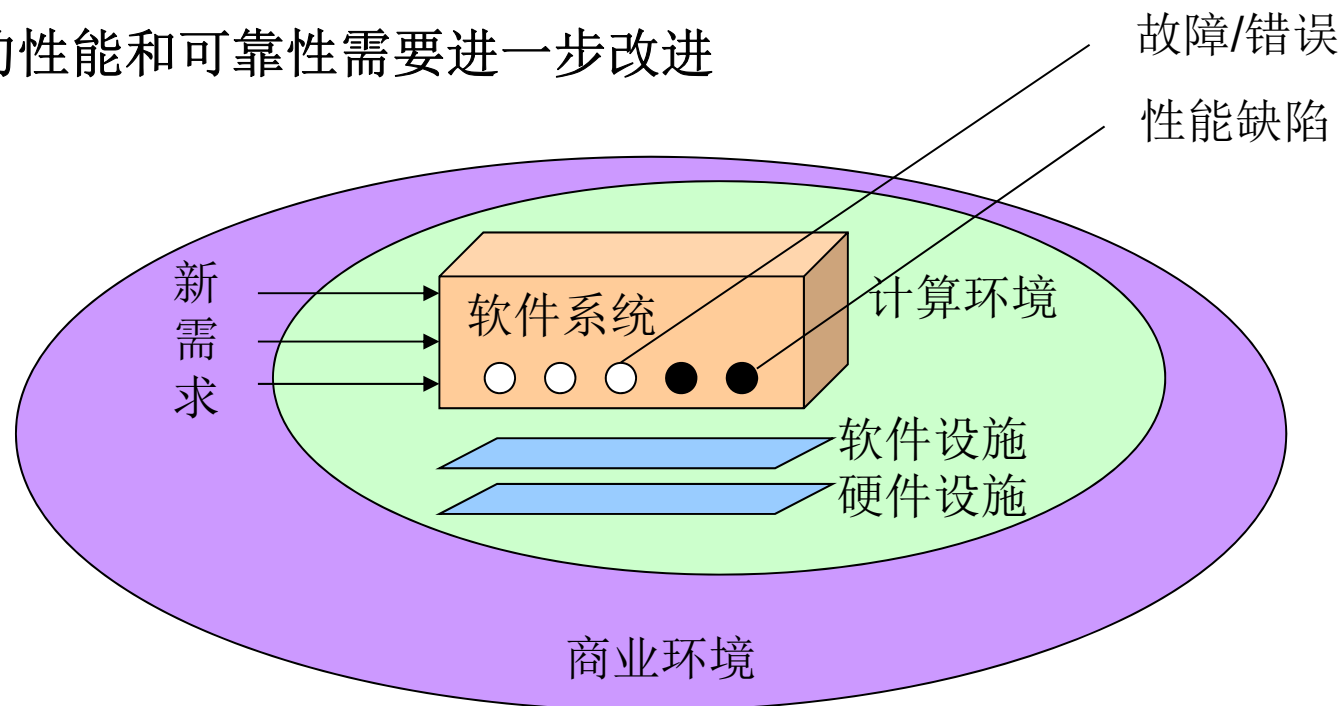
2017年9月25日

主要内容

1. 软件演化
2. 软件维护与软件再工程
3. 软件配置管理(SCM)
4. Git/Github
5. 持续集成

软件演化

- 软件在使用过程中，新的需求不断出现
- 商业环境在不断地变化
- 软件中的缺陷需要进行修复
- 计算机硬件和软件环境的升级需要更新现有的系统
- 软件的性能和可靠性需要进一步改进



软件演化的Lehman定律

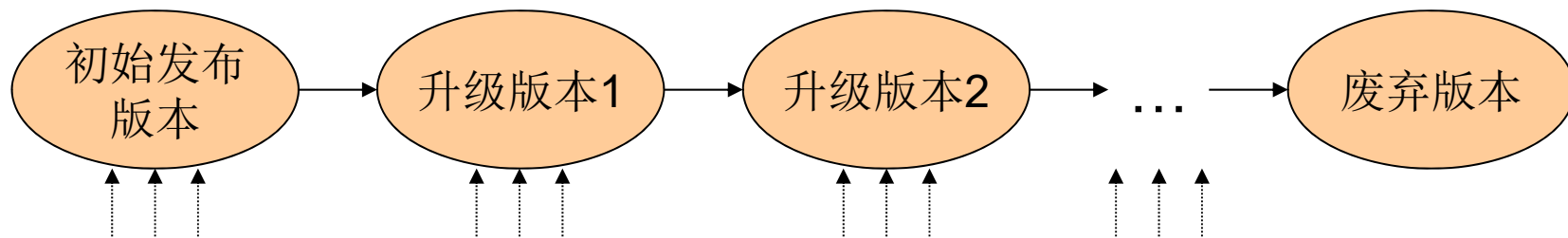
■ 持续变化(continuing change)

- 现实世界的系统要么变得越来越没有价值，要么进行持续不断的变化以适应环境的变化；
- 环境变化产生软件修改，软件修改又继续促进环境变化；

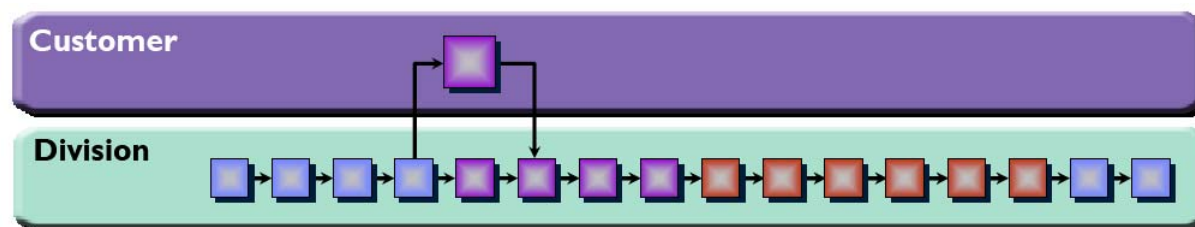
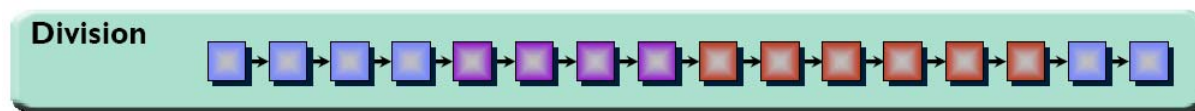
■ 复杂度逐渐增大(increasing complexity)

- 当系统逐渐发生变化时，其结构和功能将变得越来越复杂，并逐渐难以维护并失去控制，直至无法继续演化，从而需要大量额外的资源和维护工作来保持系统的正常运行。
- 软件修改会引入新的错误，造成故障率的升高；

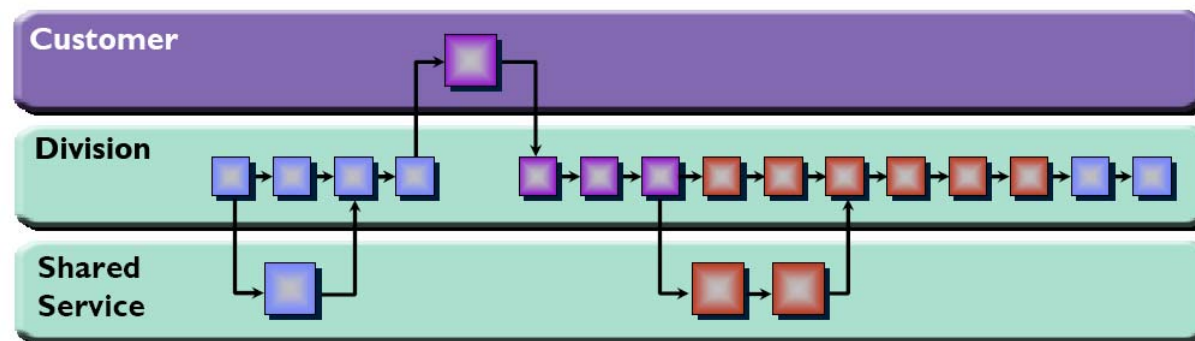
热力学第二定律(熵值理论)



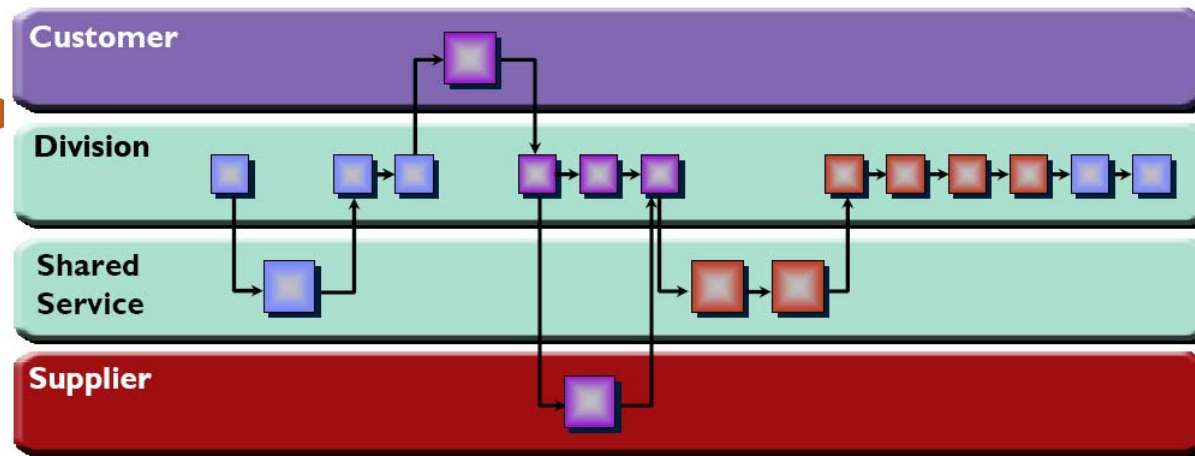
软件演化的典型例子



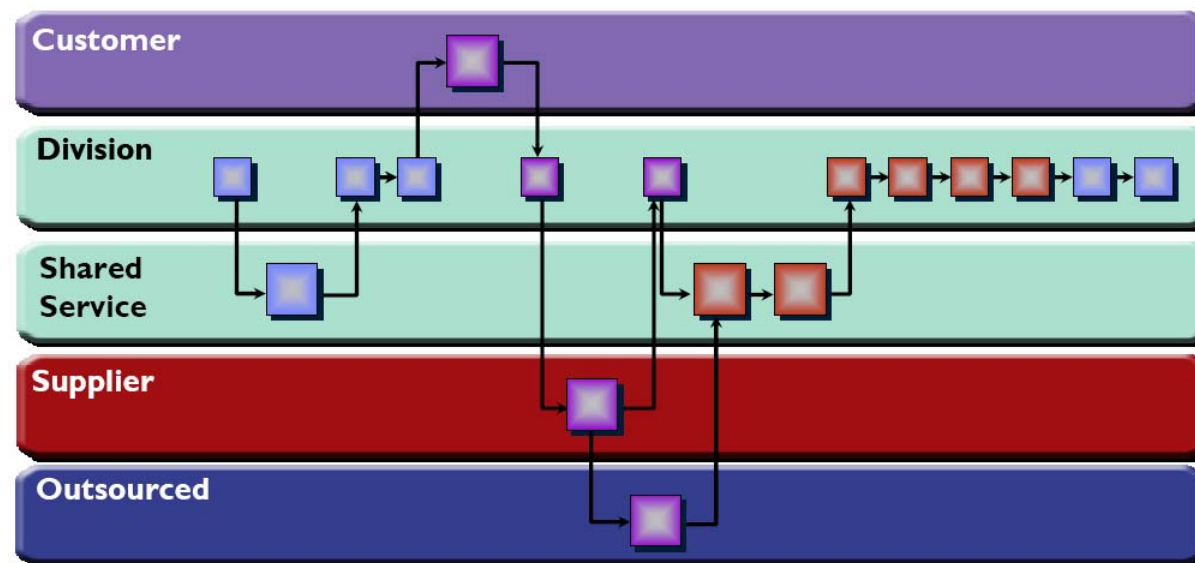
变化1: 客户订单入口



变化2: 提取公共服务



变化3：供应商管理库存



变化4：物流业务外包出去

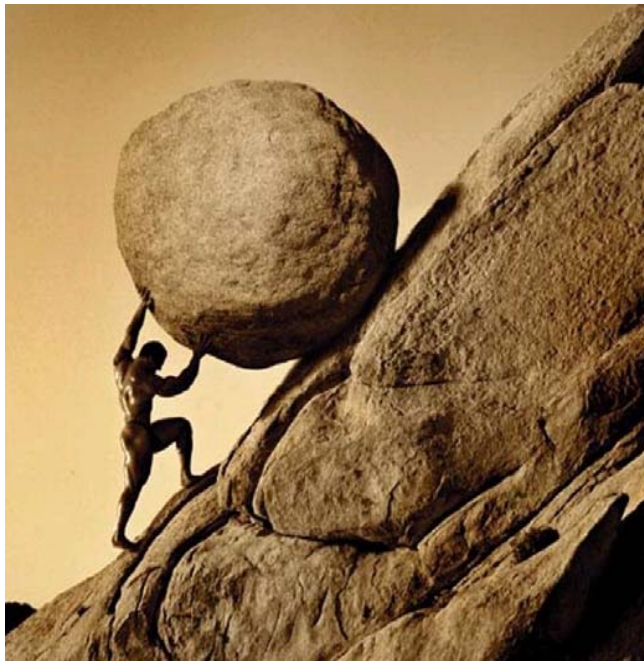
银行业的困境

- 目前不少银行/保险公司所使用的核心业务处理系统甚至仍在使用1980年代所开发的COBOL语言书写的程序，用的还是VSAM文件系统。
- 随着银行/保险行业标准和会计准则的更新和新产品的推出，这些原有的系统已经无法支持这些新变化。
- 但由于现在已经无法找到能够完全理解这些核心系统的程序人员(例如COBOL早已很少使用)，所以不少银行/保险公司往往受困于此却无计可施。



现代版的西西弗斯和吴刚

- 当今的软件开发人员正是现代版的西西弗斯和吴刚，他们面对的是快速变化的技术和无休无止、越来越复杂的需求。
- 必须寻求一种更好的软件研发方法论，以支持软件的持续演化。



软件演化的处理策略

- **软件维护(Software Maintenance)**
 - 为了修改软件缺陷或增加新的功能而对软件进行的变更
 - 软件变更通常发生在局部，不会改变整个结构
- **软件再工程(Software Re-engineering)**
 - 为了避免软件退化而对软件的一部分进行重新设计、编码和测试，提高软件的可维护性和可靠性等
- 前者比后者的力度要小。

主要内容

1. 软件演化

2. 软件维护

2.1 软件维护

2.2 软件再工程

3. 软件配置管理(SCM)

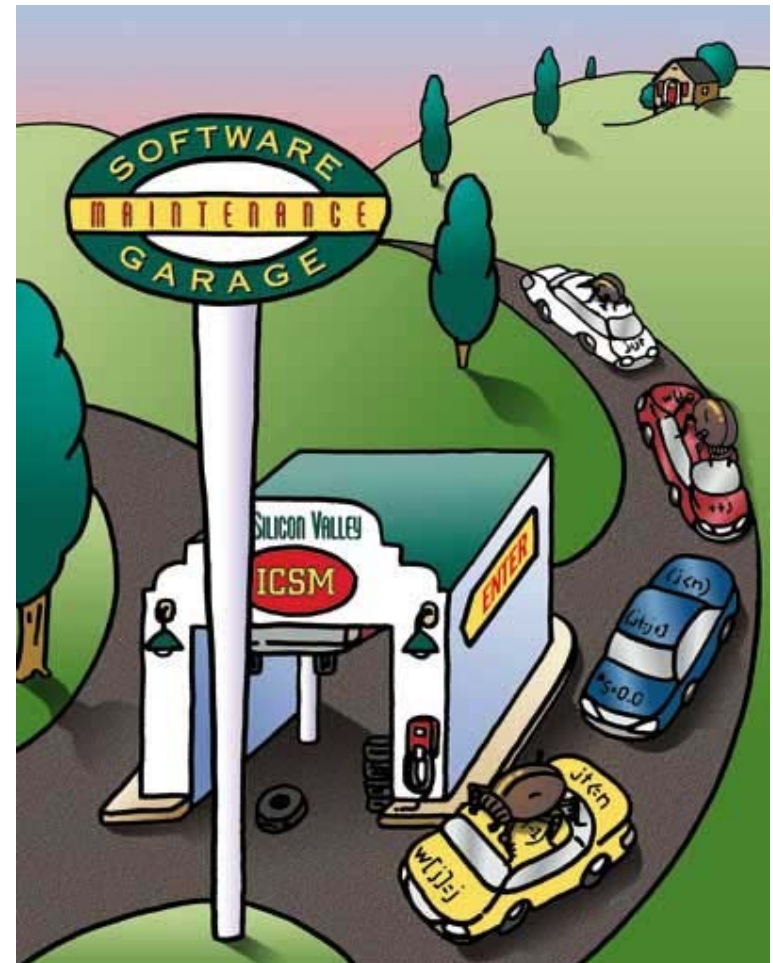
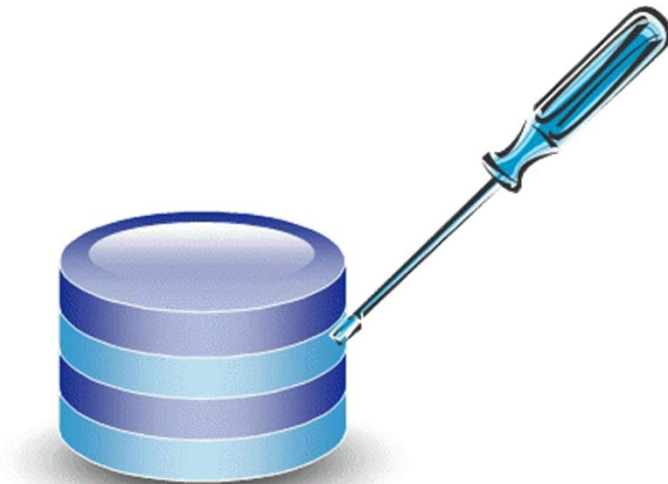
4. Git/Github

5. 持续集成

软件维护

■ 软件维护

- (ANSI/IEEE) 在软件产品发行和被投入运行使用之后对其的修改，以改正错误，改善性能或其他属性，从而使产品适应新的环境或新的需求。



软件维护的类型

- **纠错性维护**：修改软件中的缺陷或不足
- **适应性维护**：修改软件使其适应不同的操作环境，包括硬件变化、操作系统变化或者其他支持软件变化等
- **完善性维护**：增加或修改系统的功能，使其适应业务的变化
- **预防性维护**：为减少或避免以后可能需要的前三类维护而提前对软件进行的修改工作

纠错性维护、适应性维护

■ 纠错性维护(Corrective Maintenance):

- 在软件交付使用后，因开发时测试的不彻底、不完全，必然会有部分隐藏的错误遗留到运行阶段。
- 这些隐藏下来的错误在某些特定的使用环境下就会暴露出来。
- 为了识别和纠正软件错误、改正软件性能上的缺陷、排除实施中的错误使用，应当进行的诊断和改正错误的过程就叫做纠错性维护。

■ 适应性维护(Adaptive Maintenance):

- 在使用过程中，外部环境(新的硬、软件配置)和数据环境(数据库、数据格式、数据输入/输出方式、数据存储介质)可能发生变化。
- 为使软件适应这种变化，而去修改软件的过程就叫做适应性维护。

完善性维护、预防性维护

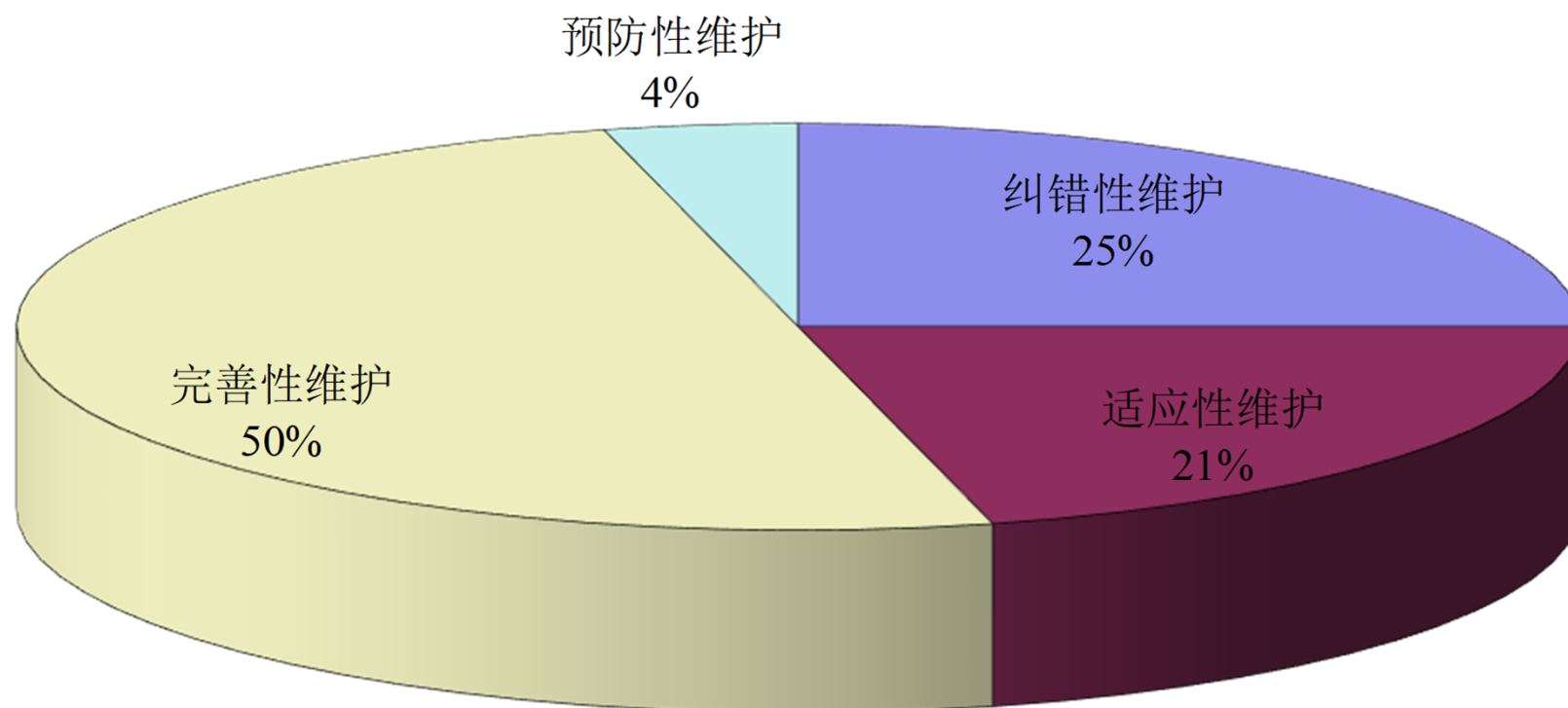
■ 完善性维护(Perfective Maintenance)

- 在软件的使用过程中，用户往往会对软件提出新的功能与性能要求。
- 为了满足这些要求，需要修改或再开发软件，以扩充软件功能、增强软件性能、改进加工效率、提高软件的可维护性。
- 这种情况下进行的维护活动叫做完善性维护。

■ 预防性维护(Preventive Maintenance):

- 为了提高软件的可维护性、可靠性等，为以后进一步改进软件打下良好基础。
- 定义为“采用先进的软件工程方法对需要维护的软件或软件中的某一部分(重新)进行设计、编制和测试”。

软件维护的类型



小结

- 实践表明，在几种维护活动中，**完善性维护所占的比重最大**，即大部分维护工作是改变和加强软件，而不是纠错。
 - 完善性维护不一定是救火式的紧急维修，而可以有计划、有预谋的一种再开发活动。
 - 来自用户要求扩充、加强软件功能、性能的维护活动约占整个维护工作的50%。
- 软件维护活动所花费的工作占整个生存期工作量的70%以上，这是由于在漫长的软件运行过程中需要不断对软件进行修改，以改正新发现的错误、适应新的环境和用户新的要求，这些修改需要花费很多精力和时间，而且有时会引入新的错误。

软件维护的内容

■ 程序维护

- 根据使用的要求，对程序进行全部或部分修改。修改以后，必须书写修改设计报告。

■ 数据维护

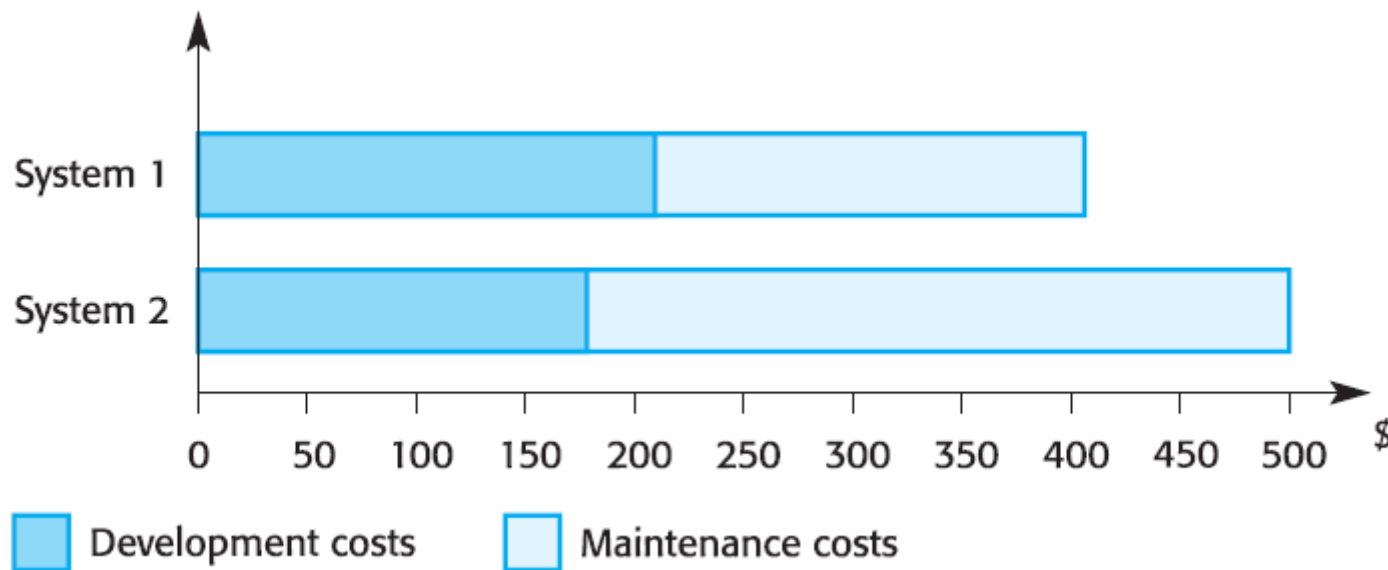
- 数据维护指对数据有较大的变动。如安装与转换新的数据库；或者某些数据文件或数据库出现异常时的维护工作，如文件的容量太大而出现数据溢出等。

■ 硬件维护

- 硬件人员应加强设备的保养以及定期检修，并做好检验记录和故障登记工作。

软件维护的成本

- 软件的维护成本极其昂贵
 - 业务应用系统：维护费用与开发成本大体相同
 - 嵌入式实时系统：维护费用是开发成本的四倍以上



课堂讨论

- 系统维护和升级的困难有哪些、如何规避？ 1组

影响维护成本的因素

■ 团队稳定性

- 系统移交后开发团队会解散，人员分配到其他项目中，负责维护的人员通常不是原开发人员，需要花时间理解系统。

■ 合同责任

- 维护合同一般独立于开发合同，这样开发人员有可能缺少为方便维护而写软件的动力。

■ 人员技术水平

- 维护人员有可能缺乏经验，而且不熟悉应用领域。

■ 程序年龄与结构

- 程序结构随年龄的增加而受到破坏，不易理解和变更。

软件维护的典型困难

- 软件维护中出现的大部分问题都可归咎于软件规划和开发方法的缺陷：
 - 软件开发时采用急功近利还是放眼未来的态度，对软件维护影响极大，软件开发若不严格遵循软件开发标准，维护就会遇到许多困难。
- 例如：
 1. 读懂原开发人员写的程序通常相当困难
 2. 软件人员的流动性，使得软件维护时，很难与原开发人员沟通
 3. 没有文档或文档严重不足
 4. 软件设计时，欠考虑软件的可修改性
 5. 频繁的软件升级，要追踪软件的演化变得很困难，使软件难以修改

造成困难的根本原因

- 很难甚至不可能追踪软件版本的进化过程，软件的变化没在相应文档中反映出来；
- 很难甚至不可能追踪软件的整个创建过程；
- 理解他人的程序非常困难，当软件配置不全、仅有源代码时问题尤为严重；
- 软件人员流动性很大，维护他人软件时很难得到开发者的帮助；
- 软件没有文档、或文档不全、或文档不易理解、或与源代码不一致；
- 多数软件设计未考虑修改的需要(有些设计方法采用了功能独立和对象类型等一些便于修改的概念)，软件修改不仅困难而且容易出错；
- 软件维护不是一项有吸引力的工作，从事这项工作令人缺乏成就感。

不好的维护所造成的代价

- 因为可用的资源必须供维护任务使用，以致耽误甚至丧失了开发的良机；
- 当看来合理的有关改错或修改的要求不能及时满足时将引起用户不满；
- 由于维护时的改动，在软件中引入了潜伏的故障，从而降低了软件的质量；
- 当必须把软件工程师调去从事维护工作时，将在开发过程中造成混乱；
- 生产率的大幅度下降。

遗留系统

■ 遗留系统(legacy system)

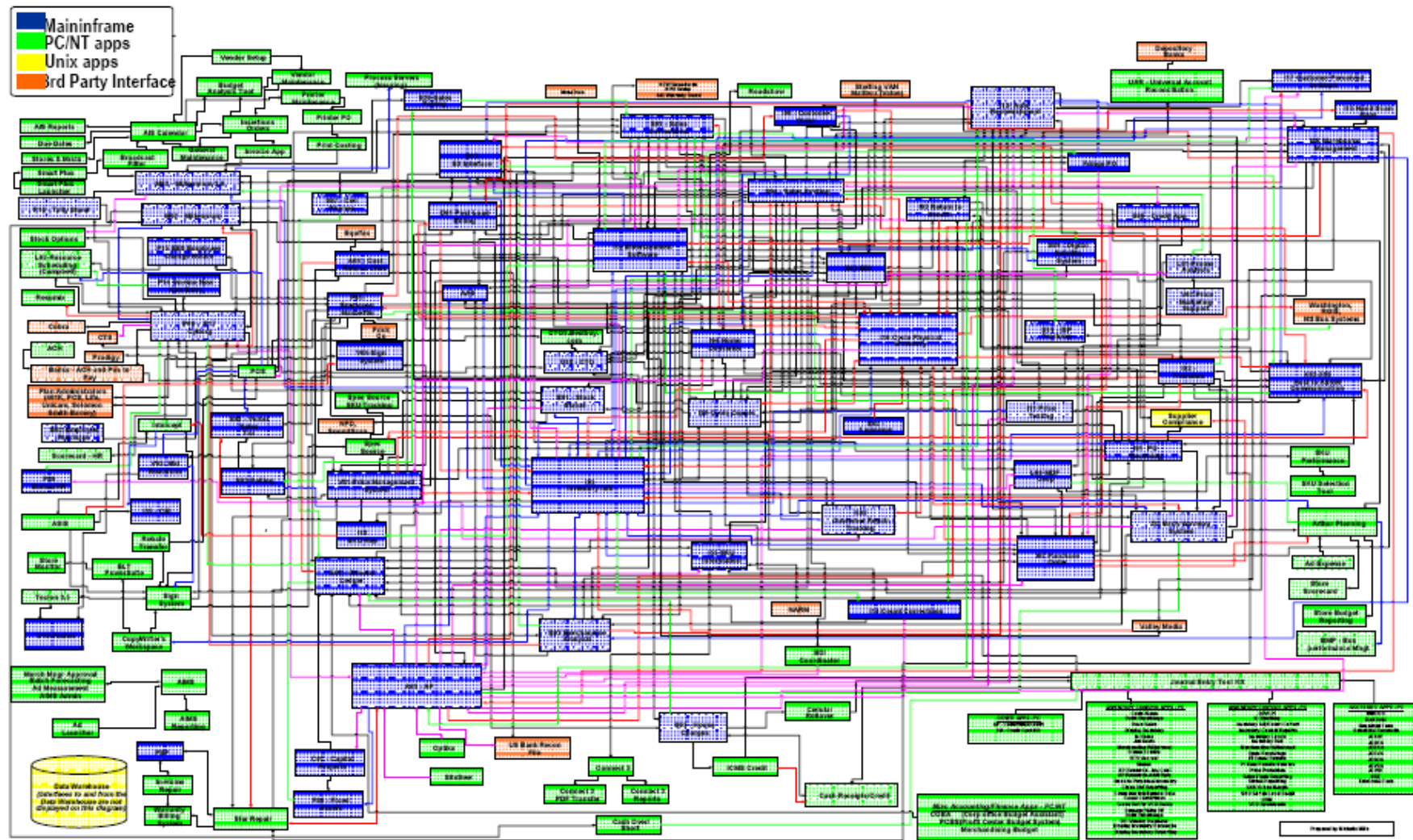
- “已经运行了很长时间的、对用户来说很重要的、但是目前已无法完全满足要求却不知道如何处理的软件系统”。

■ 特点

- 现有维护人员没有参与开发
- 不具备现有的开发规范
- 文档不完整，修改记录简略



遗留系统



遗留系统

- 更换遗留系统(Legacy System)是有风险的
 - 遗留系统几乎没有完整的描述
 - 业务过程和遗留系统的操作方式紧密地“交织”在一起
 - 重要的业务规则隐藏在软件内部
 - 开发新软件本身是有风险的
- 变更遗留系统的问题
 - 系统的不同部分是由不同的团队实现的
 - 系统的部分或全部是用一种过时不用的语言编写
 - 文档不充分或过时
 - 经过多年维护，系统结构可能已经破坏，理解设计难度大

如何维护遗留系统？

- 一种可行的解决方案：软件再工程
 - 采用先进的软件工程方法对整个软件或软件中的一部分重新进行设计、编写和测试，以提高软件的可维护性和可靠性，保证系统的正常运行。
 - “把今天的SE方法学用于昨天的系统以满足明天的需要”

主要内容

1. 软件演化

2. 软件维护

2.1 软件维护

2.2 软件再工程

3. 软件配置管理(SCM)

4. Git/Github

5. 持续集成

重建一所房子...

- 你购买了一所二手房，但你不满意目前房子的装修情况...
- 为此，你考虑重建这所房子。
- 怎么做？
 - 根据一组标准，对房子进行检查
 - 确认其结构是否良好
 - 了解房子最初是如何建造的
 - 采用何种材料进行重建
 - 采用何种方式进行重建



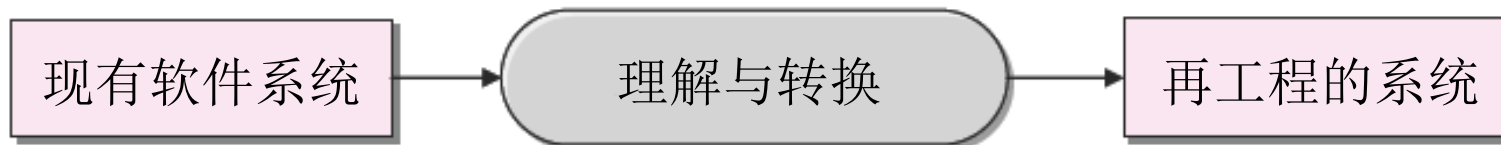
软件再工程

- **软件再工程(Software Re-engineering)**
 - 重新构造或编写现有系统的一部分或全部，但不改变其功能
 - 在大型系统中某些部分需要频繁维护时，可应用软件再工程
 - 目的：努力使系统更加易于维护，系统需要被再构造和再文档化
- **优势**
 - 减少风险：重新开发一个在用的系统具有很高的风险，可能会有开发问题、人员问题和规格说明问题
 - 降低成本：再工程的成本比重新开发软件的成本要小得多

正向工程与再工程

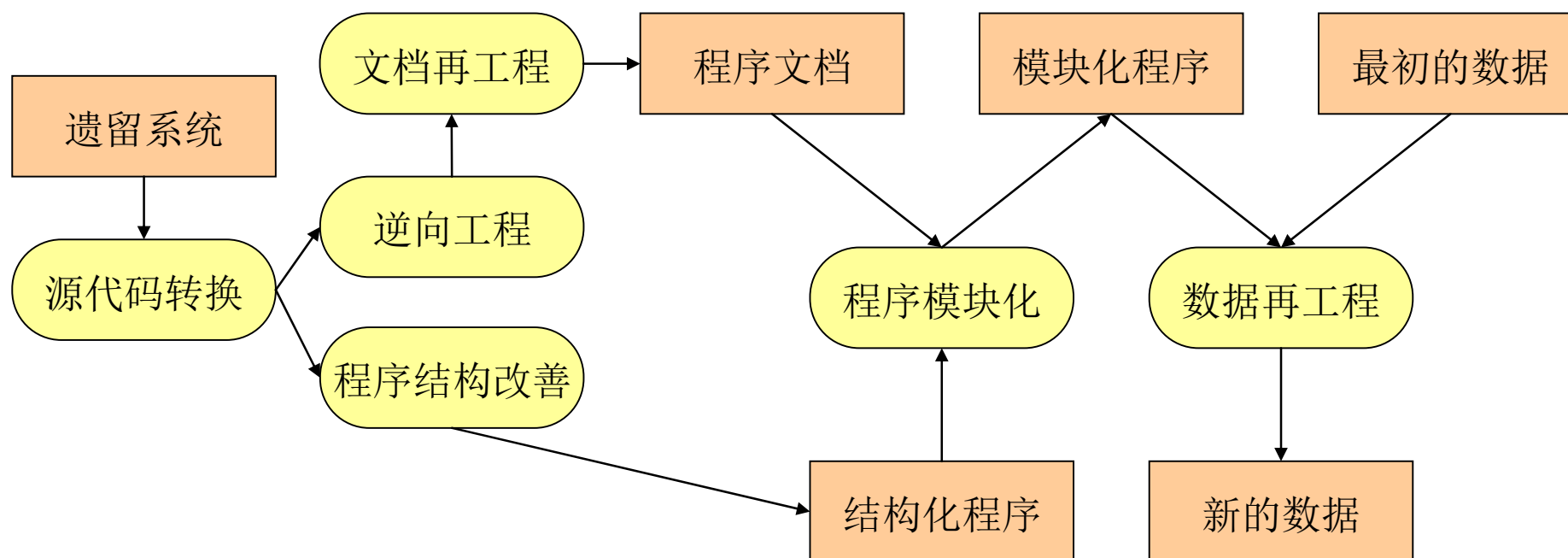


正向工程



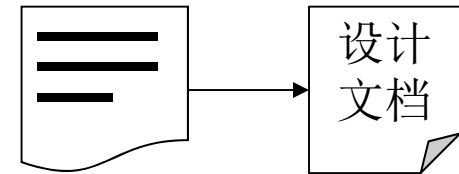
软件再工程

软件再工程的过程

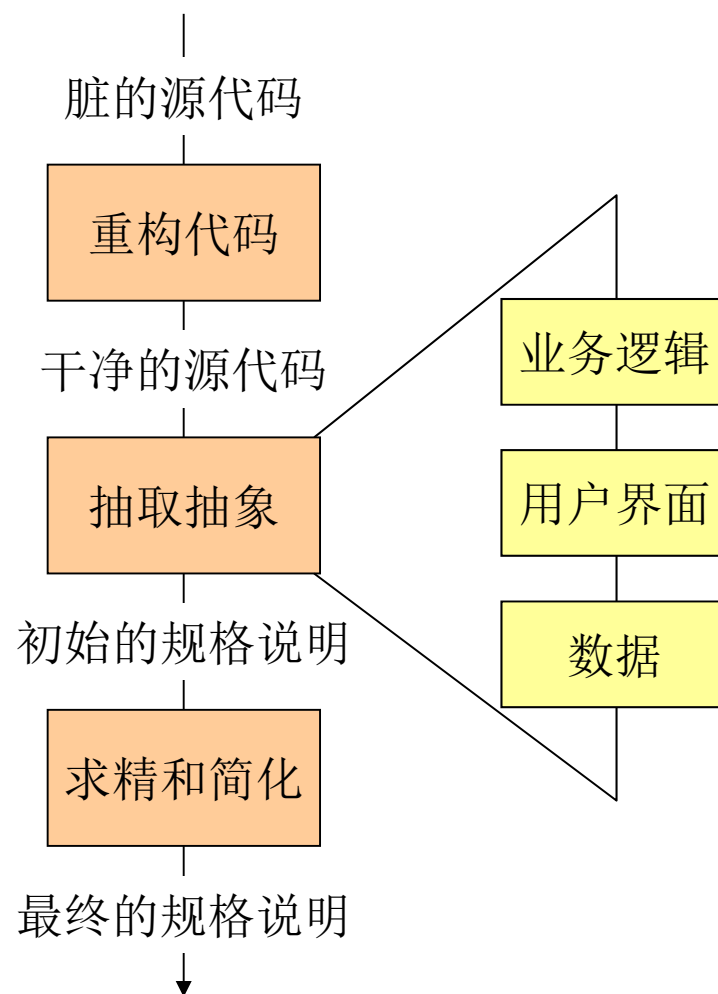


再工程的基本活动

- **源代码转换(source code transformation)**
 - 代码从原有的程序设计语言转换到一种新语言
- **逆向工程(reverse engineering)**
 - 分析程序并抽取信息记录其结构和功能
- **程序结构改善(program restructuring)**
 - 分析和修改程序的控制结构，使其更易读和好理解
- **程序模块化(program modulization)**
 - 重新组织程序的结构
- **数据再工程(data restructuring)**
 - 改变程序处理的数据以反映程序的变更(文件存储→数据库存储，数据格式改变，等等)



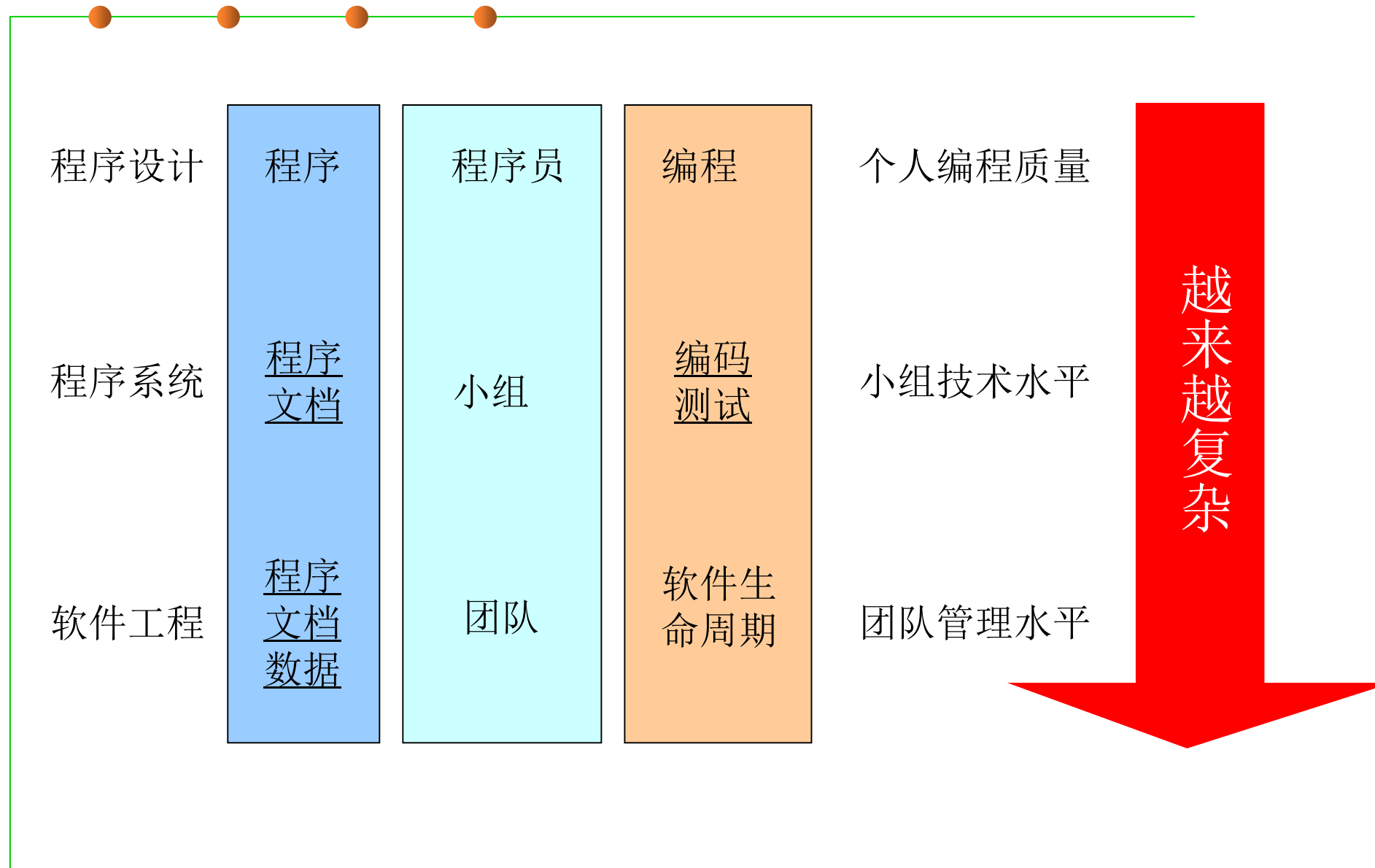
逆向工程



主要内容

1. 软件演化
2. 软件维护与软件再工程
3. 软件配置管理(SCM)
4. Git/Github
5. 持续集成

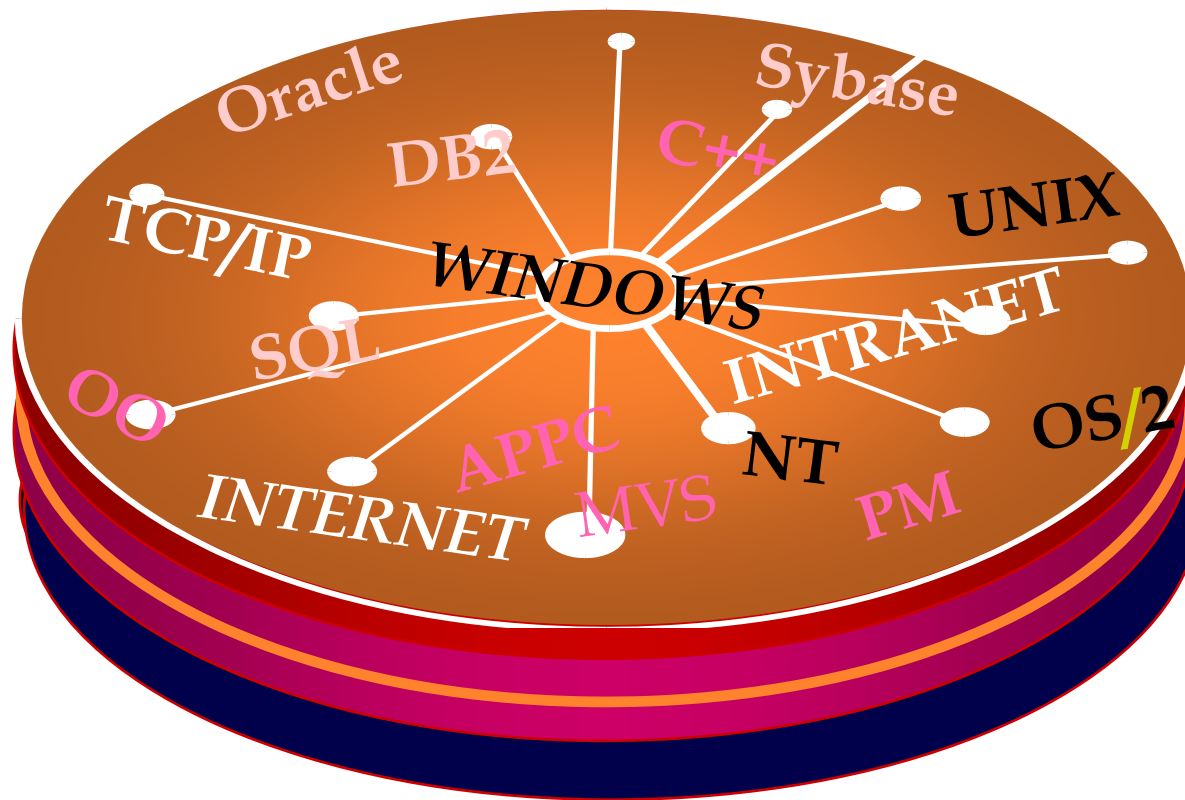
软件配置管理的背景



软件开发过程中面临的困境

- 缺乏对用户需求进行有效的管理和追踪的工具
- 产品升级和维护所必需的程序和文档非常混乱
- 代码可重用性差从而不能对产品进行功能扩充
- 开发过程中的人员流动经常发生
- 软件开发人员之间缺乏必要的交流
- 由于管理不善致使未经测试的软件加入到产品中
- 用户与开发商没有有效的产品交接界面

开发环境的复杂性



- 多操作系统
- 多开发工具
- 网络化
- 团队方式
- 异地开发

缺乏管理所造成的问题

软件生产达不到规模化

缺少有效的通信机制



成员间缺少沟通

人员流动

硬件配置

- 你要购买一台计算机，需要考虑哪些配置？

- CPU类型与主频
- 缓存大小
- 内存容量与频率
- 硬盘容量与转速
- 显示器大小与分辨率
- ...



软件配置

- 软件配置(**software configuration**): 由在软件工程过程中产生的所有信息项构成, 它可以看作该软件的具体形态(软件配置项)在某一时刻的瞬间影像。
- 软件配置管理(**Software Configuration Management, SCM**)



软件配置管理的含义

- “协调软件开发从而使得混乱减到最小的技术称为软件配置管理。它是对开发团队正在开发软件的修改进行标识、组织和控制的技术，目的是使错误量降至最低，并使生产率最高。”

—— Wayne Babich

《SCM Coordination for Team Productivity》

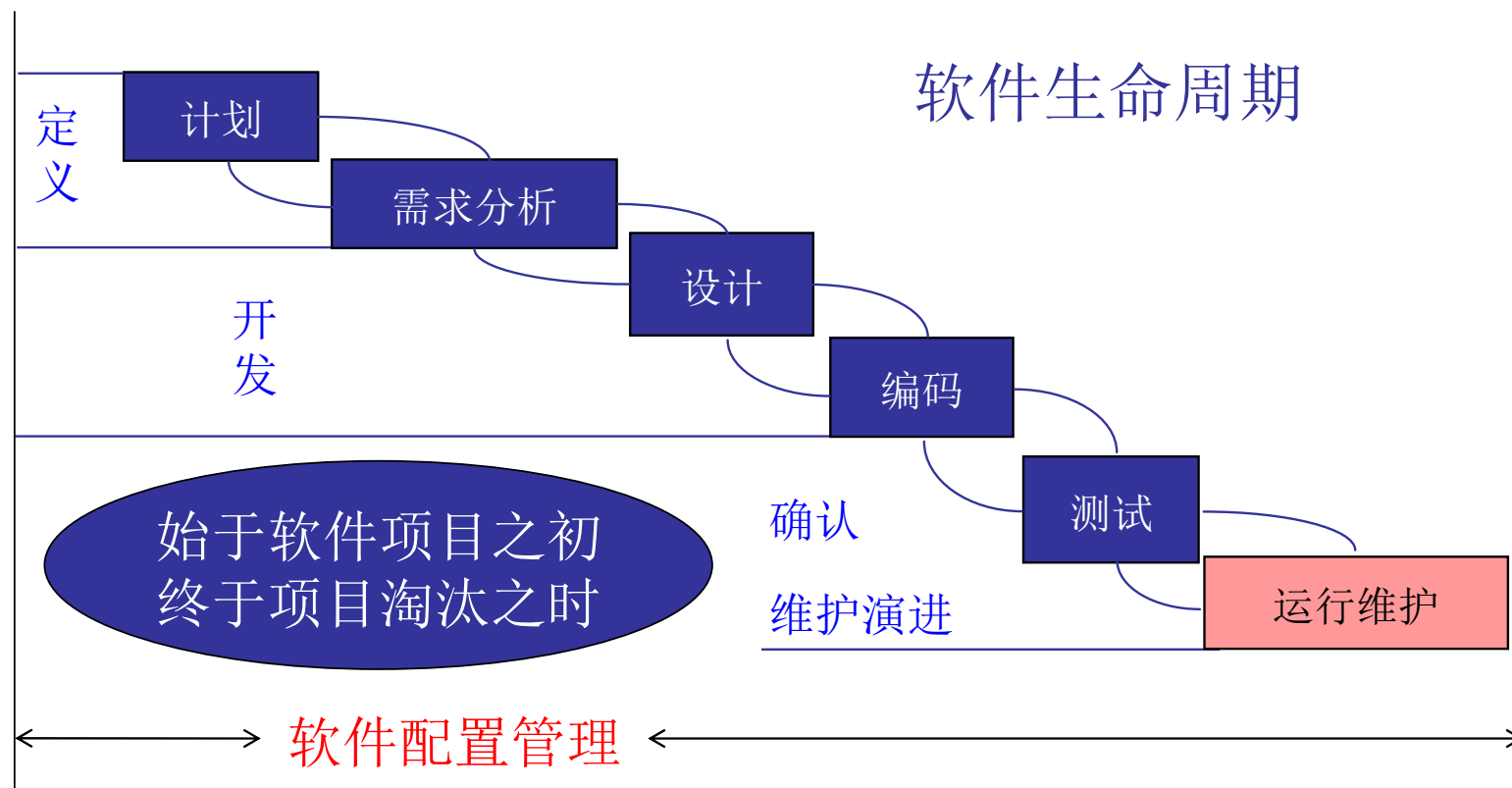
- “配置管理能够系统的处理变更，从而使得软件系统可以随时保持其完整性，因此称为‘变更控制’，可以用来评估提出的变更请求，跟踪变更，并保存系统在不同时间的状态。”

—— Steve McConnell

《Code Complete》

软件配置管理的特点

- SCM贯穿整个软件生命周期与软件工程过程



软件配置管理的目标

- 目标：
 - 标识变更
 - 控制变更
 - 确保变更的正确实现
 - 向开发组织内各角色报告变更

- 总结：当变更发生时，能够提高适应变更的容易程度，并且能够减少所花费的工作量。

SCM的基本元素

- 配置项(Configuration Item, CI)
- 基线(Baseline)
- 配置管理数据库(CMDB)
- 备件库(Definitive Hardware Store, DHS)
- 最终软件库(Definitive Software Library, DSL)

(1) 配置项

- 软件过程的输出信息可以分为三个主要类别：

- 计算机程序(源代码和可执行程序)
- 描述计算机程序的文档(针对技术开发者和用户)
- 数据(包含在程序内部或外部)

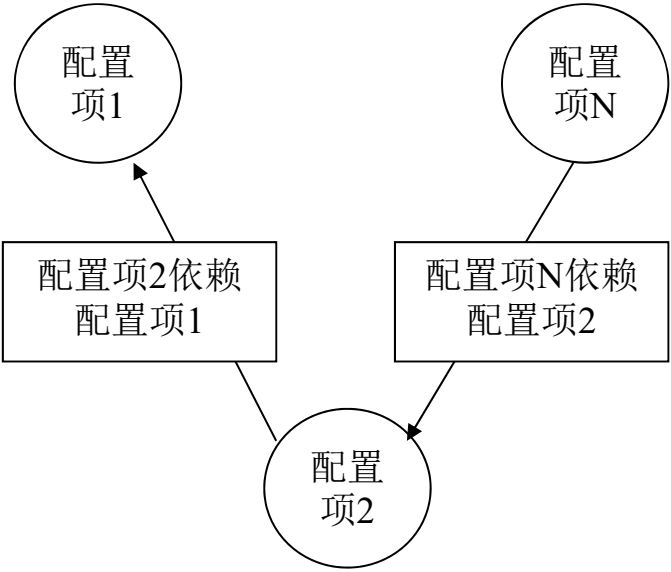
这些项包含了所有在软件过程中产生的信息，总称为软件配置项(SCI)。

- **SCI**是软件全生命周期内受管理和控制的基本单位，大到整个系统，小到某个硬件设备或软件模块。

配置项

- SCI具有唯一的名称标识和多个属性。
 - 名称
 - 描述
 - 类型(模型元素、程序、数据、文档等)
 - 项目标识符
 - 资源表
 - “实现”
 - 变更和版本信息

配置项之间的依赖关系

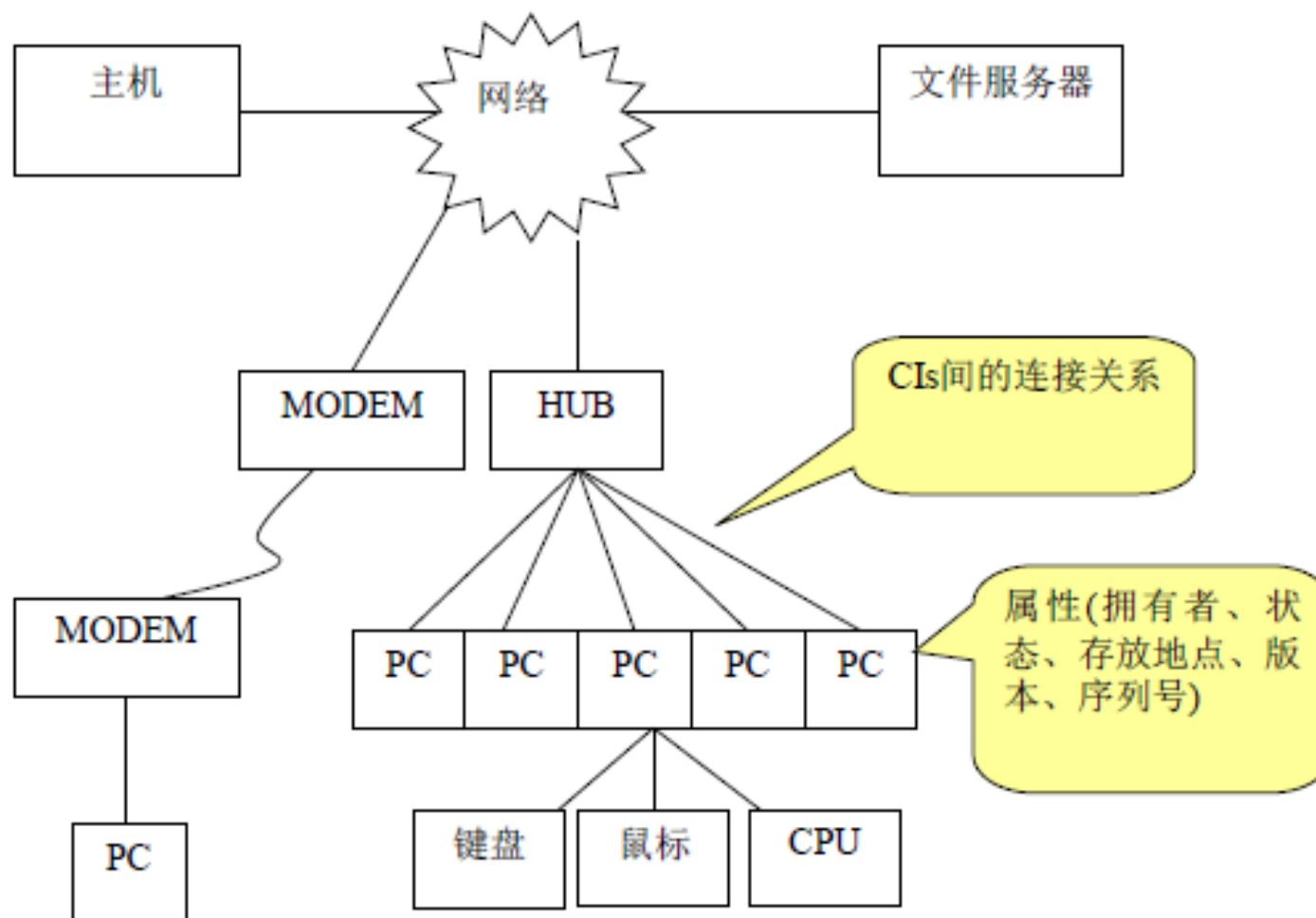


依赖关系	配置项1	配置项2	...	配置项N
配置项1		X		
配置项2				X
...				
配置项N				

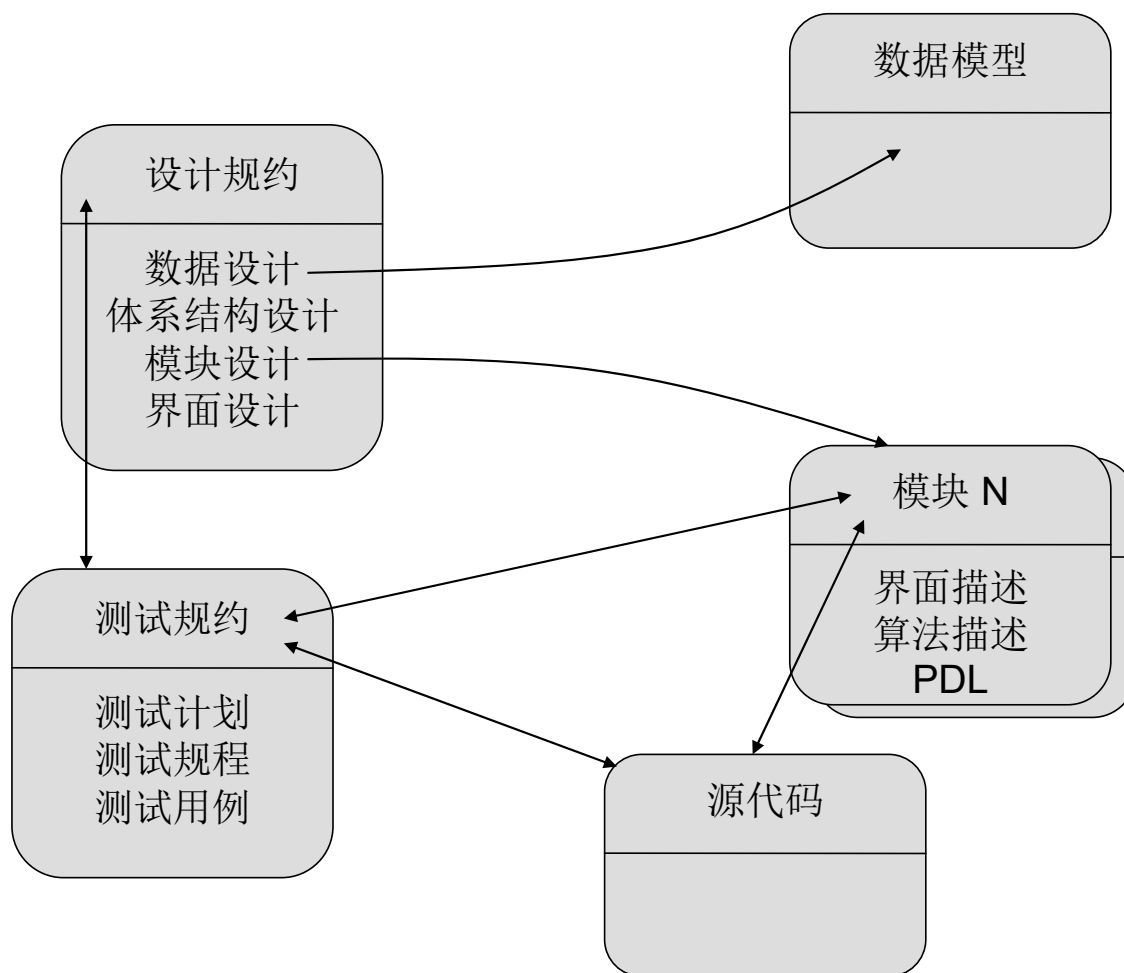
配置项之间的关系

- 配置项之间都可能有哪些类型的依赖关系？
 - 整体-部分关系
 - Data flow diagram (DFD) <part-of> analysis model
 - Analysis model <part-of> requirement specification
 - 关联关系
 - Data model <interrelated> data flow diagram (DFD)
 - Data model <interrelated> test case class m;
 - 还有哪些？

实例：硬件配置项

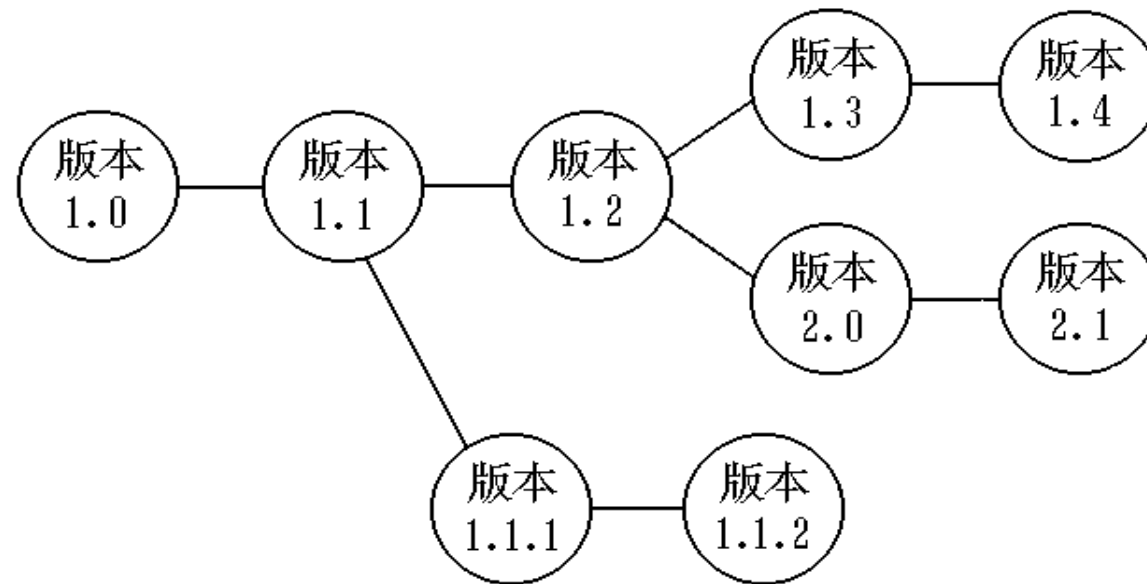


配置项之间的依赖关系



配置项的演变图

- 在对象成为基线以前可能要做多次变更，在成为基线之后也可能需要频繁的变更。
- 对于每一配置项都要建立一个演变图，以记录对象的变更历史。

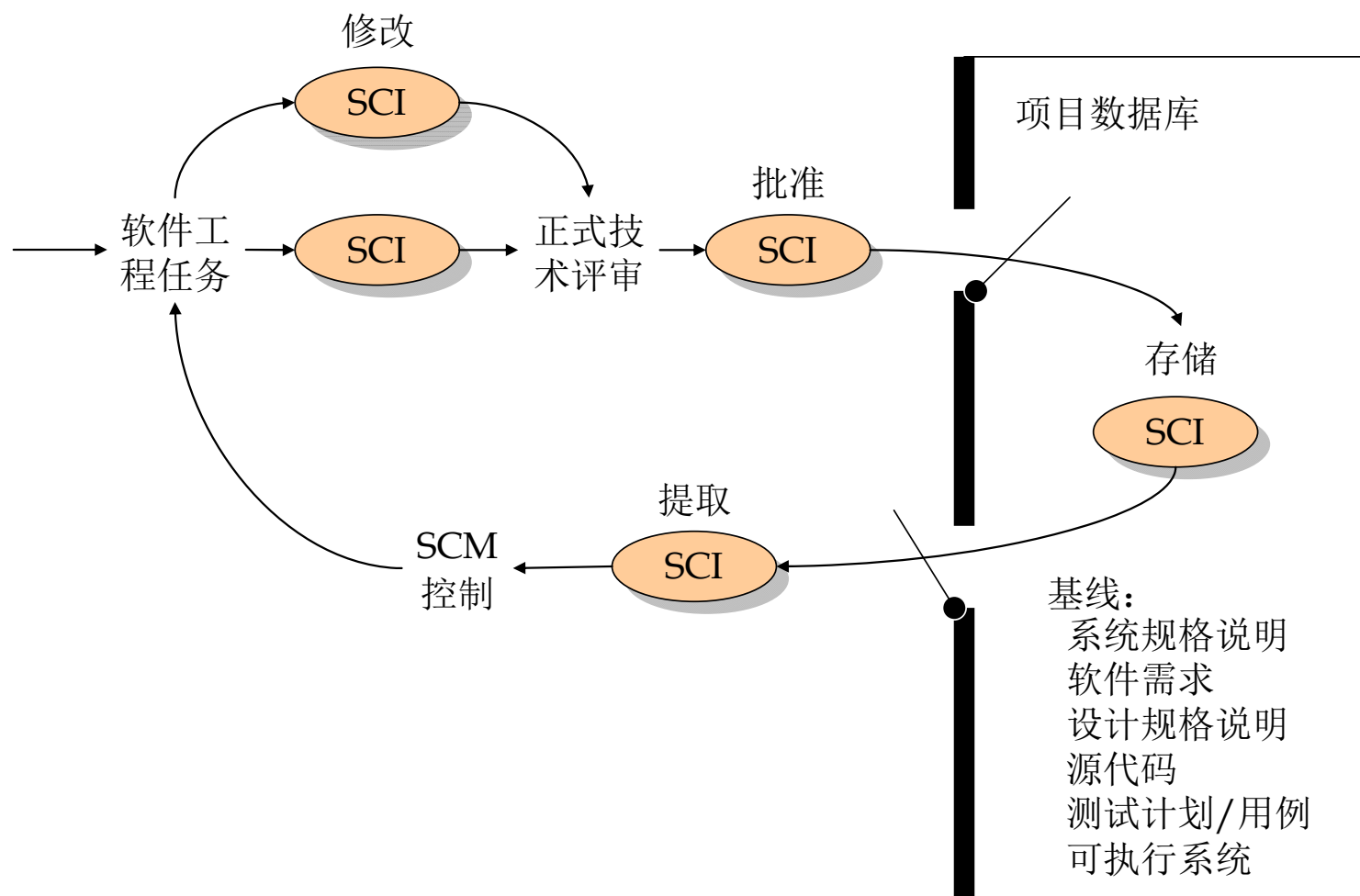


(2) 基线

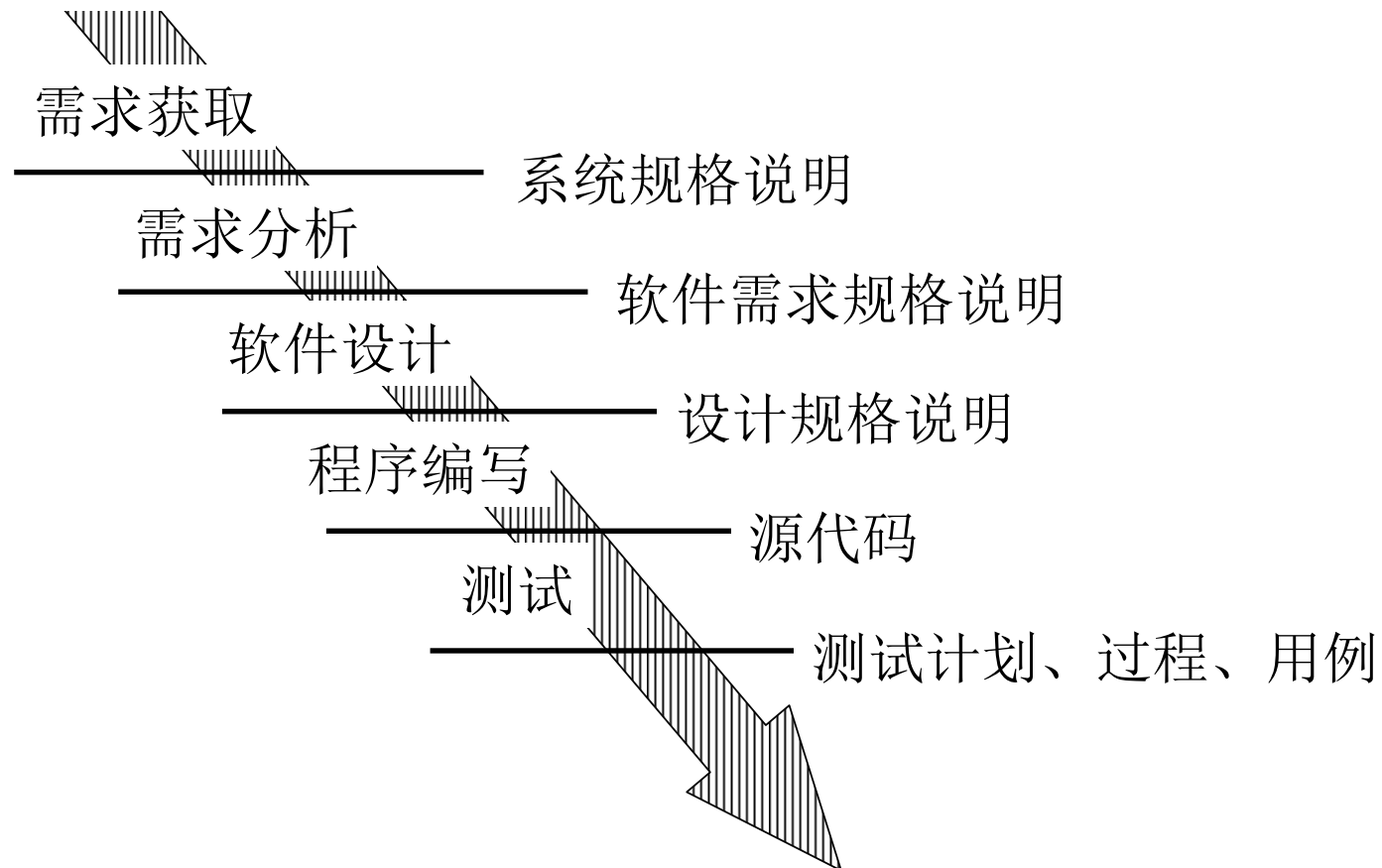
- 基线：已经通过正式评审和批准的软件规格说明或代码，它可以作为进一步开发的基础，并且只有通过正式的变更规程才能修改它。
- 在软件配置项成为基线之前，可以迅速而随意的进行变更；
- 一旦成为基线，变更时需要遵循正式的评审流程才可以变更。
- 因此，基线可看作是软件开发过程中的“里程碑”。



基线



软件开发各阶段的基线

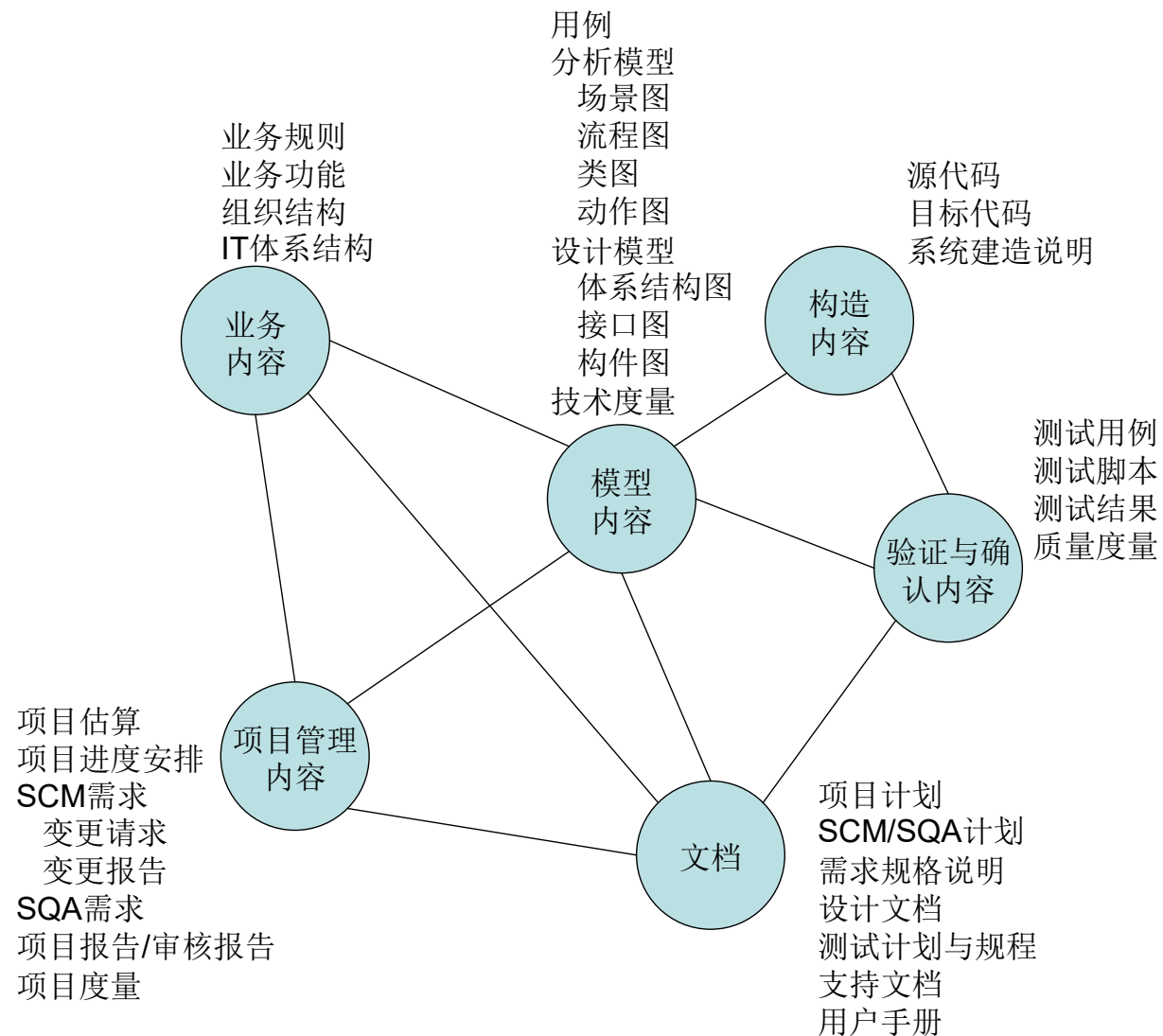


(3) 配置管理数据库

- 配置管理数据库(CMDB)(也称“SCM中心存储库”), 用于保存于软件相关的所有配置项的信息以及配置项之间关系的数据库。
 - 每个配置项及其版本号
 - 变更可能会影响到的配置项
 - 配置项的变更路线及轨迹
 - 与配置项有关的变更内容
 - 计划升级、替换或弃用的配置项
 - 不同配置项之间的关系



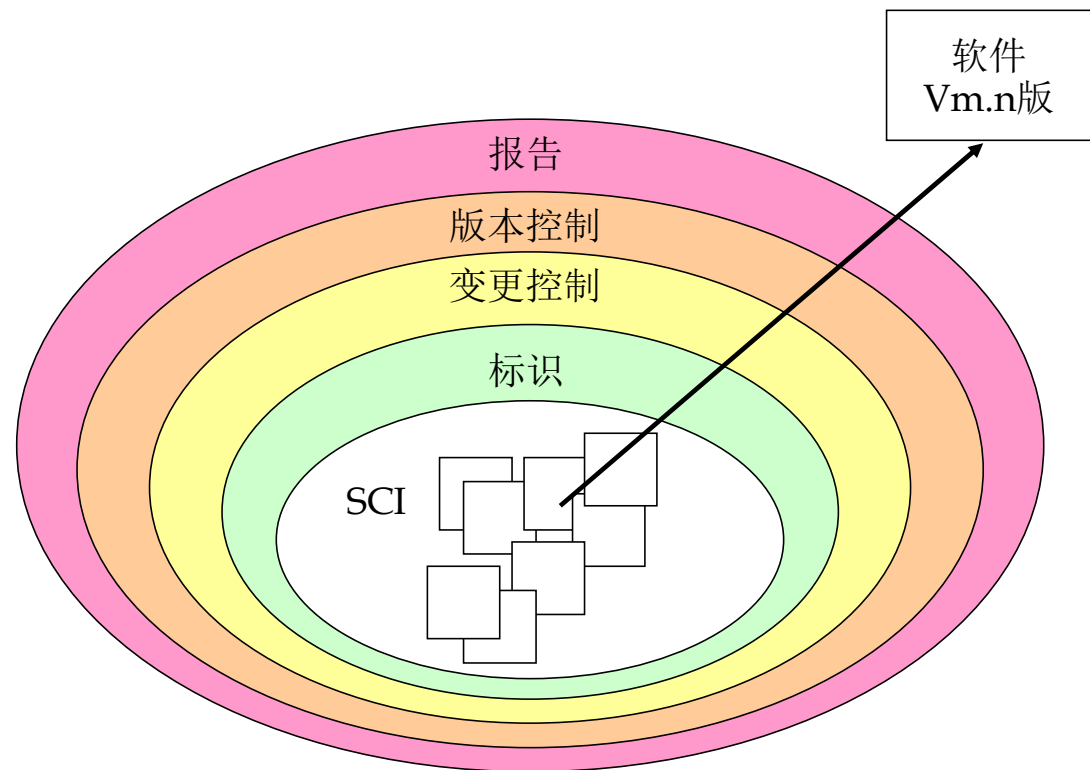
配置管理数据库



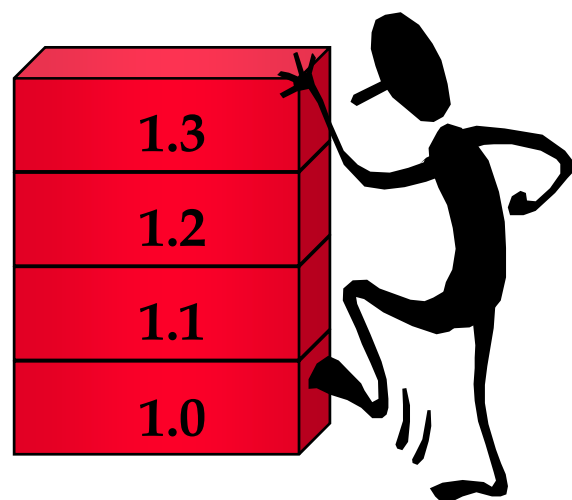
配置管理数据库

■ CMDB的功能:

- 存储配置项及其之间的关系
- 版本控制
- 相关性跟踪和变更管理
- 需求跟踪
- 配置管理
- 审核跟踪



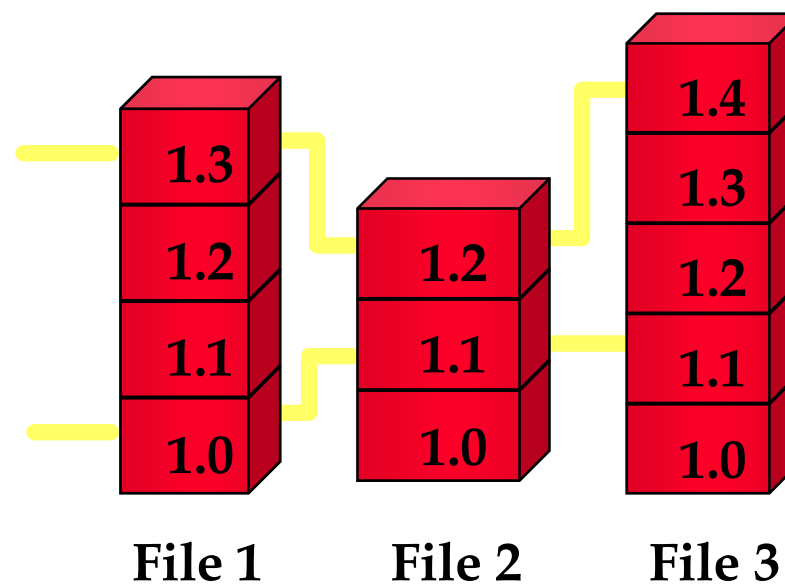
版本控制



正式版

Beta 1

↑
Version
Labels



SCM总结

- 软件开发者的典型心理：“嫌麻烦”、“侥幸”
- 而一旦开发过程中出现变更，就会造成更大的麻烦。
- 任何项目成员都要对其工作成果进行配置管理，应当养成良好的习惯。
- 不必付出过多的精力，最低要求是保证重要工作成果不发生混乱。

主流的版本控制软件

- 版本控制软件(VCS, Version Control System)
 - VSS (Microsoft Visual SourceSafe)
 - CVS (Concurrent Version System)
 - IBM Rational ClearCase
 - SVN (Subversion)
 - Git
 -

课堂讨论

- 根据你开发软件的经历，是否有过对代码进行很大程度修改的现象？你使用什么方法进行代码的维护？ 1组

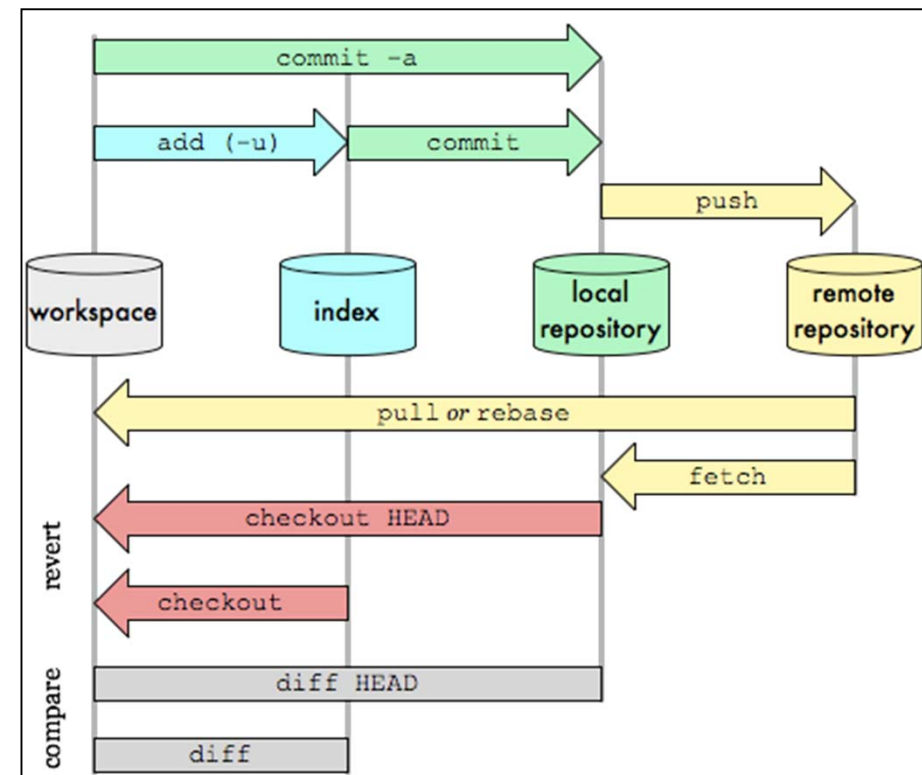
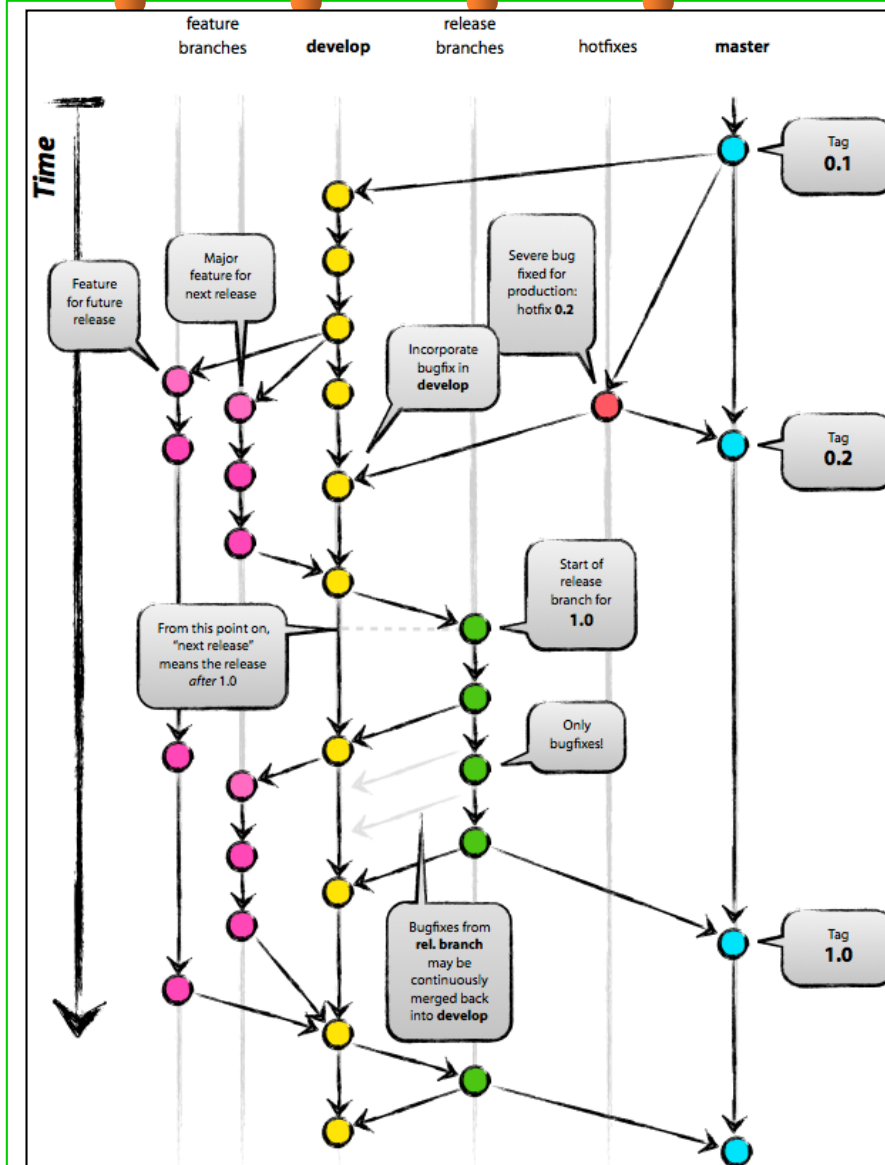
主要内容

1. 软件演化
2. 软件维护与软件再工程
3. 软件配置管理(SCM)
- 4. Git/Github**
5. 持续集成



4. Git/Github

Git / Github



课堂讨论

- 查阅资料，了解Git出现的历史过程，阐述它与传统其他SCM的差异所在。 1组

VCS的发展—手工方式

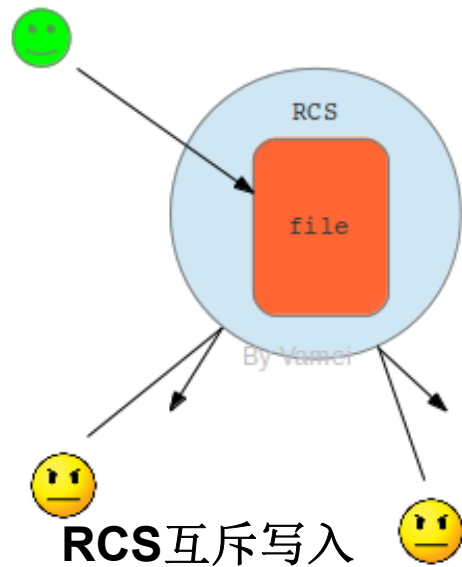
- 早期的软件开发大部分是愉快的**个人创作**。比如UNIX下的sed是L.E.McMahon写的，Python的第一个编译器是Guido写的，Linux最初的内核是Linus写的。
- 这些程序员可以用**手工的方式**进行备份，并以注释或者新建文本文件来记录变动。

VCS的发展—互斥写入方式

- 19世纪70年代末80年代初，VCS的概念已经存在，比如UNIX平台的 **RCS (Revision Control System)**。
- RCS对文件进行集中式管理，主要目的是避免多人合作情况下可能出现的冲突。
 - 如果多用户同时写入同一个文件，其写入结果可能相互混合和覆盖，从而造成结果的混乱。
- RCS允许多个用户同时读取文件，但只允许一个用户锁定(locking)并写入文件。
 - 当一个程序员登出(check-out)某个文件并对文件进行修改的时候。只有在这个程序完成修改，并登入(check-in)文件时，其他程序员才能登出文件。
- 用户可以对原文件进行许多改变 (change, or file delta)。一旦重新登入文件，这些改变将保存到RCS系统中。通过check-in将改变永久化的过程叫做提交(commit)。

VCS的发展--互斥写入方式

- RCS的互斥写入机制避免了多人同时修改同一个文件的可能。
- 但代价是程序员长时间的等待，给团队合作带来不便。如果某个程序员登出了某个文件，而忘记登入，那他就要面对队友的怒火了。(从这个角度来说，RCS造成的问题甚至大于它所解决的问题.....)



VCS的发展--互斥写入方式

- 文件每次**commit**都会创建一个**新的版本(revision)**。
- **RCS**给每个文件创建了一个追踪文档来记录版本的历史。这个文档的名字通常是原文件名加后缀,v (比如**main.c**的追踪文档为**main.c,v**)。
- 追踪文档中包括：最新版本的文件内容，每次**check-in**的发生时间和用户，每次**check-in**发生的改变。在最新文档内容的基础上，减去历史上发生的改变，就可以恢复到之前的历史版本。这样，**RCS**就实现了备份历史和记录改变的功能。



RCS历史版本追踪

VCS的发展--互斥写入方式

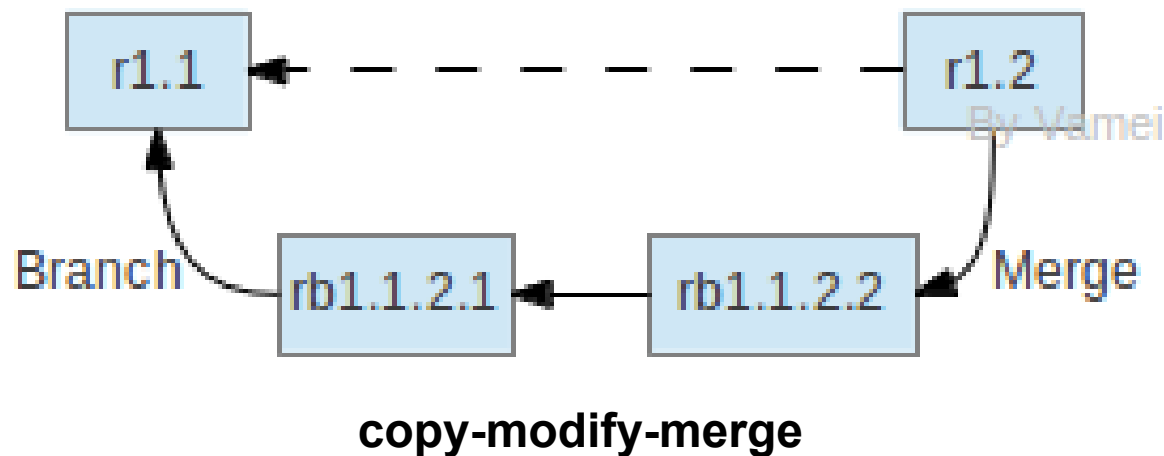
- 相对与后来的版本管理软件，**RCS**纯粹线性的开发方式非常不利于团队合作。但**RCS**为多用户写入冲突提供了一种有效的解决方案。**RCS**的版本管理功能逐渐被其他软件(比如**CVS**)取代。

VCS的发展--CVS

- 1986年，Dick Grune写了一系列的shell脚本用于版本管理，并最终以这些脚本为基础，构成了CVS (Concurrent Versions System)。CVS后来用C语言重写。
- CVS是开源软件。CVS被包含在GNU的软件包中，并因此得到广泛的推广，最终击败诸多商业版本的VCS，呈一统天下之势。
- CVS继承了RCS的集中管理的理念。
- 在CVS管理下的文件构成一个库(repository)。与RCS的锁定文件模式不同，CVS采用复制-修改-合并(copy-modify-merge)的模式，来实现多线开发。
- CVS引进了分支(branch) 的概念。多个用户可以从主干(也就是中心库)创建分支。分支是主干文件在本地复制的副本。用户对本地副本进行修改。用户可以在分支提交(commit)多次修改。用户在分支的工作结束之后，需要将分支合并到主干中，以便让其他人看到自己的改动。

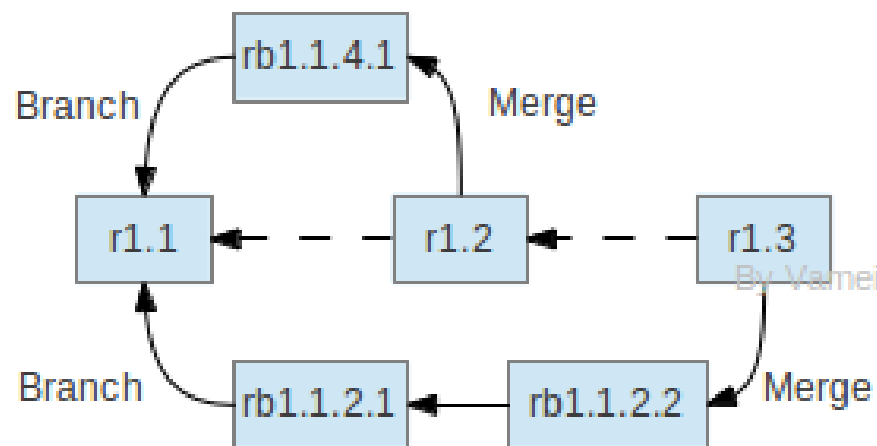
VCS的发展--CVS

- 所谓的合并，就是CVS将分支上发生的变化应用到主干的原文件上。比如下面的过程中，我们从r1.1分支出rb1.1.2.*，并最终合并回主干，构成r1.2。
- 在合并的过程中，CVS将两个change应用于r1.1，就得到了r1.2:
$$r1.2 = r1.1 + \text{change}(rb1.1.2.2 - rb1.1.2.1) + \text{change}(rb1.1.2.1 - r1.1)$$
- 注：箭头顺序表达的为版本追踪方向



VCS的发展--CVS

- 在多用户情况下，可以创建多个分支进行开发。
- 在这样的多分支合并的情况下，有可能出现冲突(colliding)。
- 比如图中，第一次合并和第二次合并都对r1.1文件的同一行进行了修改，那么r1.3将不知道如何去修改这一行 (第二次合并比图示的要更复杂一些，分支需要先将主干拉到本地，合并过之后传回主干)。
- CVS要求冲突发生时的用户手动解决冲突。用户可以调用编辑器，对文件发生合并冲突的地方进行修改，以决定最终版本(r1.3)的内容。

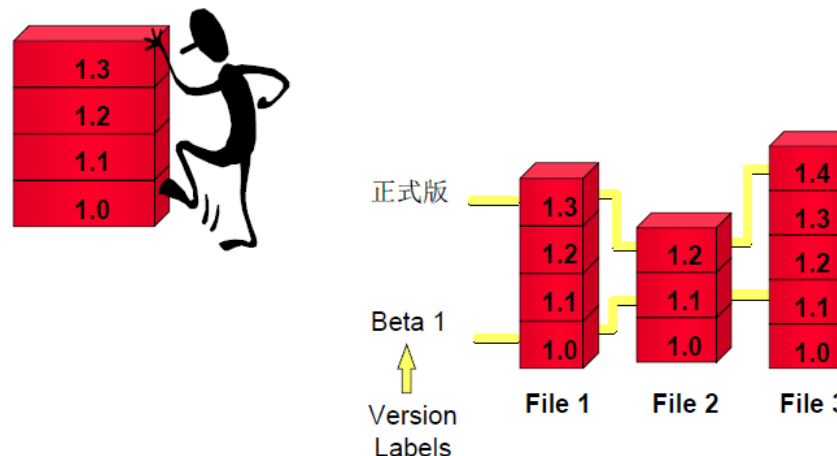


VCS的发展--CVS

- CVS管理下的每个文件都有一系列独立的版本号(比如上面的r1.1, r1.2, r1.3)。但每个项目中往往包含有许多文件。CVS用标签(tag)来记录一个集合, 这个集合中的元素是一对(文件名: 版本号)。比如我们的项目中有三个文件(file1, file2, file3), 我们创建一个v1.0的标签:

```
tag v1.0 (file1:r1.3) (file2:r1.1) (file3:r1.5)
```

- v1.0的tag中包括了r1.3版本的文件file1, r1.1版本的file2…… 一个项目在发布(release)的时候, 往往要发布多个文件。标签可以用来记录该次发布的时候, 是哪些版本的文件被发布。



VCS的发展--CVS

- CVS应用在许多重要的开源项目上。在90年代和00年代初，CVS在开源世界几乎不二选。尽管CVS已经长达数年没有发布新版本，我们依然可以在许多项目中看到CVS的身影。
- CVS也有许多常常被人诟病的地方，比如下面几条：
 - 合并不是原子操作(atomic operation): 如果有两个用户同时合并，那么合并结果将是某种错乱的混合体。如果合并的过程中取消合并，不能撤销已经应用的改变。
 - 文件的附加信息没有被追踪: 一旦纳入CVS的管理，文件的附加信息(比如上次读取时间)就被固定了。CVS不追踪它所管理文件的附加信息的变化。
 - 主要用于管理ASCII文件: 不能方便的管理Binary文件和Unicode文件。
 - 分支与合并需要耗费大量的时间: CVS的分支和合并非常昂贵。分支需要复制，合并需要计算所有的改变并应用到主干。因此，CVS鼓励尽早合并分支。

VCS的发展--SVN

- 随着时间，人们对CVS的一些问题越来越感到不满 (而且程序员喜欢新鲜的东西)，Subversion应运而生。Subversion的开发者Karl Fogel和Jim Blandy是长期的CVS用户。赞助开发的CollabNet, Inc.希望他们写一个CVS的替代VCS。这个VCS应该有类似于CVS的工作方式，但对CVS的缺陷进行改进，并提供一些CVS缺失的功能。
- 总体上说，SVN在许多方面沿袭CVS，也是集中管理库，通过记录改变来追踪历史，允许分支和合并，但并不鼓励过多分支。
- SVN在一些方面得到改善。SVN的合并是原子操作。它可以追踪文件的附加信息，并能够同样的管理Binary和Unicode文件。

VCS的发展--SVN

■ CVS和SVN的主要不同:

- 与CVS的,v文件存储模式不同, SVN采用关系型数据库来存储改变集。VCS相关数据变得不透明。
- CVS中的版本是针对某个文件的, CVS中每次commit生成一个文件的新版本。SVN中的版本是针对整个文件系统的(包含多个文件以及文件组织方式), 每次commit生成一个整个项目文件系统树的新版本。
- Subversion依赖类似于硬连接(hard link)的方式来提高效率, 避免过多的复制文件本身。Subversion不会从库下载整个主干到本地, 而只是下载主干的最新版本。

VCS的发展--SVN

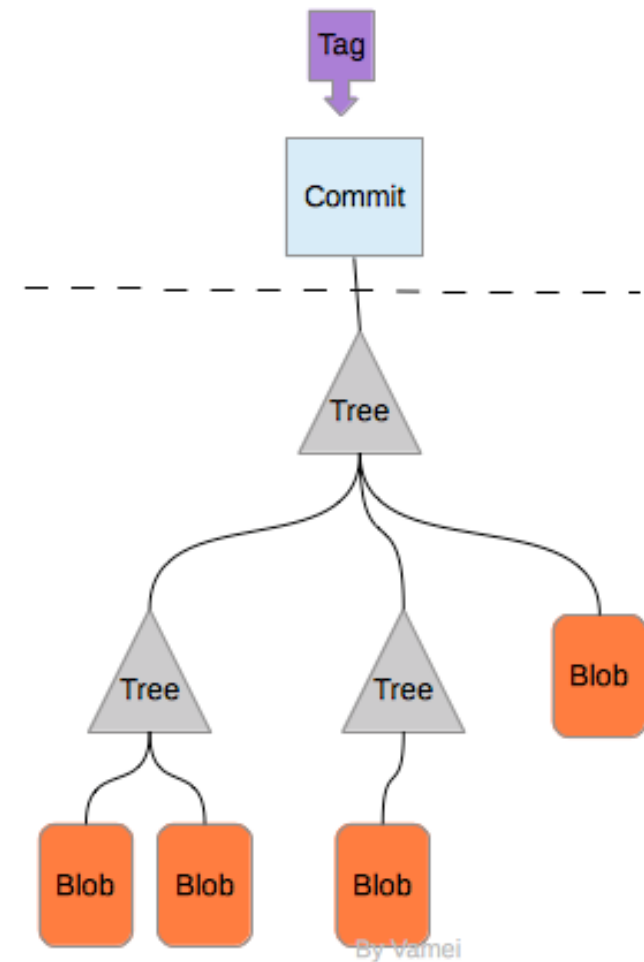
- 在SVN刚刚诞生的时候，来自CVS用户的抱怨不断。但随着时间的推移，SVN逐渐推广 (SVN已经是Apache中自带的一个模块了，SVN应用于GCC、SourceForge，新浪APP Engine等项目)，并依然有活跃的开发，而CVS则逐渐沉寂。事实上，许多UNIX的参考书的新版本中，都缩减甚至删除了CVS的内容。

VCS的发展--git

- git的作者是Linus Torvald，就是写Linux Kernel的人。
- Linus Torvald本人相当厌恶CVS(以及SVN)。然而，操作系统内核是复杂而庞大的代码“怪兽”（2012年的Linux Kernel有1500万行代码，Windows的代码不公开，估计远远超过这一数目）。
- Linux内核小组最初使用.tar文件来管理内核代码，但这远远无法匹配Linux内核代码的增长速度。
- Linus转而使用BitKeeper作为开发的VCS工具。BitKeeper是一款分布式的VCS工具，它可以快速的进行分支和合并。然而由于使用证书方面的争议(BitKeeper是闭源软件，但给Linux内核开发人员发放免费的使用证书)。
- Linus最终决定写一款开源的分布式VCS软件：git。
- 例子：12306的抢票插件拖垮了GitHub (GitHub基于git)

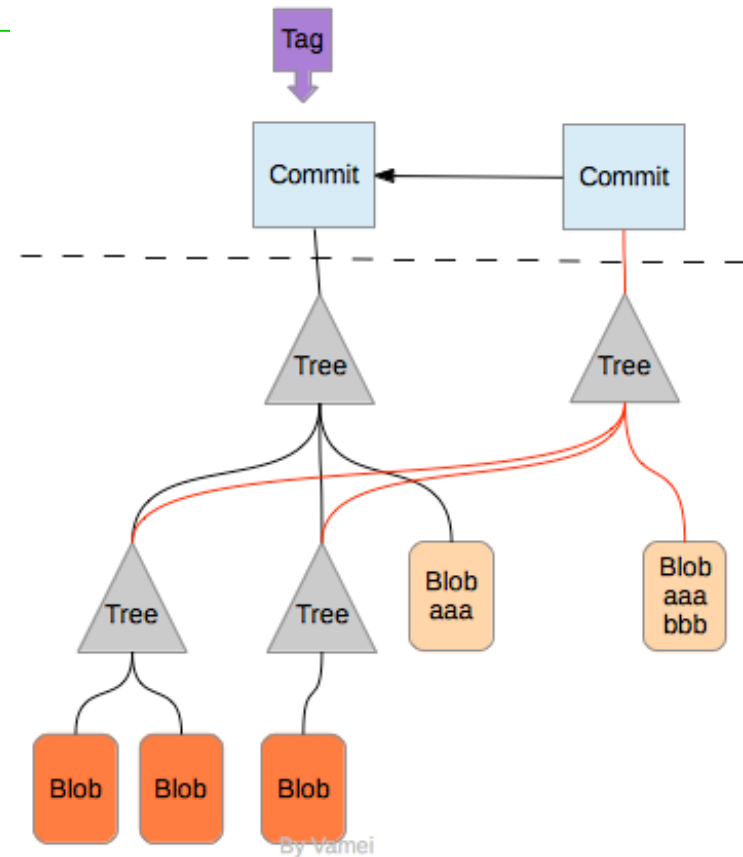
VCS的发展--git

- 对于一个开发项目，**git**会保存**blob**, **tree**, **commit**和**tag**四种对象。
 - 文件被保存为**blob**对象。
 - 文件夹被保存为**tree**对象。**tree**对象保存有指向文件或者其他**tree**对象指针。
- 上面两个对象类似于一个**UNIX**的文件系统，构成了一个文件系统树。
 - 一个**commit**对象代表了某次提交，它保存有修改人，修改时间和附加信息，并指向一个文件树。这一点与**SVN**类似，即每次提交为一个文件系统树。
 - 一个**tag**对象包含有**tag**的名字，并指向一个**commit**对象。
- 虚线下面的对象构成了一个文件系统树。在**git**中，一次**commit**实际上就是一次对文件系统树的快照(**snapshot**)。



VCS的发展--git

- 在整个开发过程中，可能会有许多次提交(commit)。每次commit的时候，git并不总是复制所有的对象。git会检验所有对象的HASH值。如果该对象的HASH值已经存在，说明该对象已经保存过，并且没有发生改变，所以git只需要调整新建tree或者commit中的指针，让它们指向已经保存过的对象就可以了。
- git在commit的时候，只会新建HASH值发生改变的对象。如图所示，我们创建新的commit的时候，只需要新建一个commit对象，一个tree对象和一个blob对象就足够了，而不需要新建整个文件系统树。

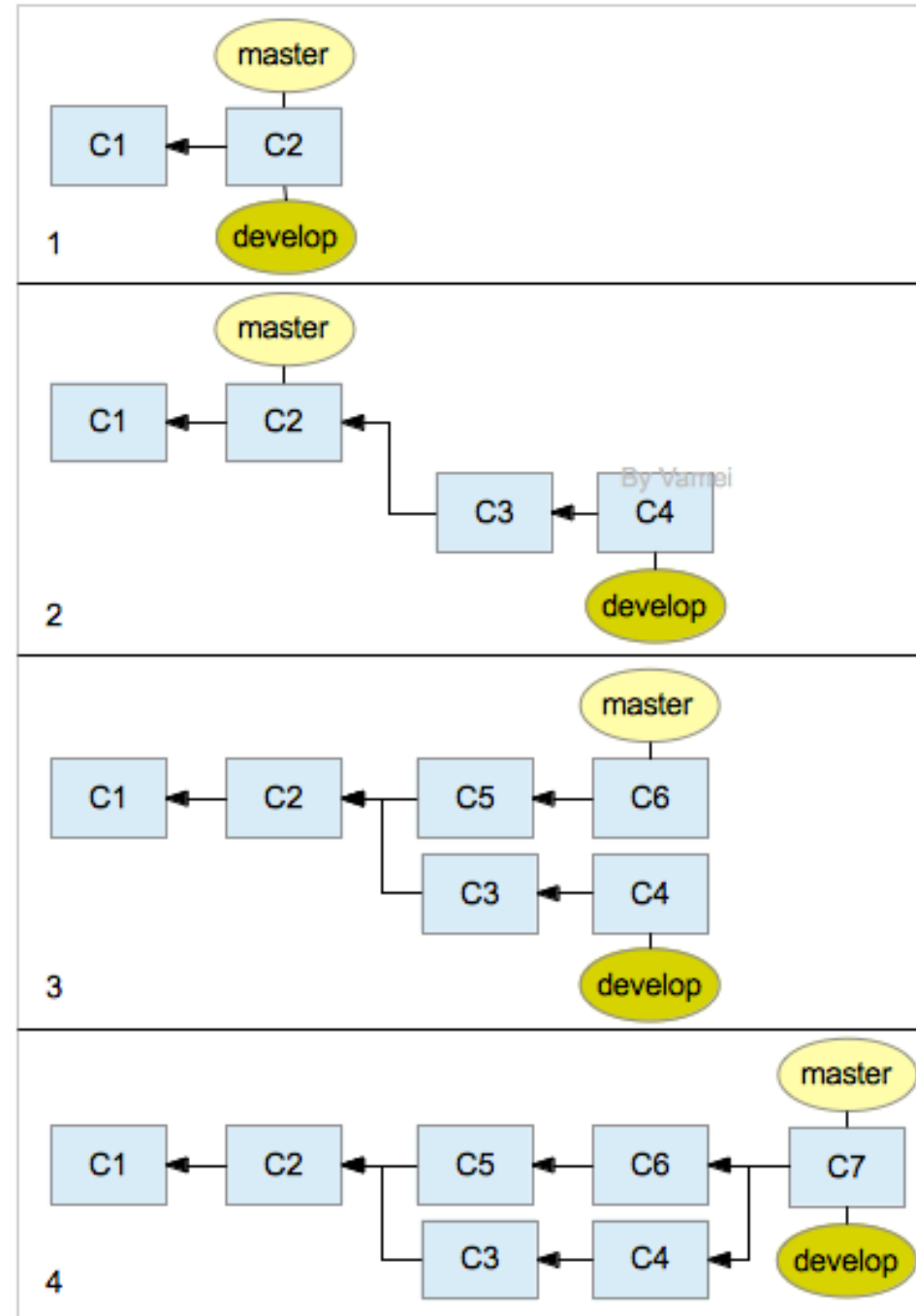


VCS的发展--git

- 可以看到，与CVS，SVN保存改变(file delta)的方式形成对照，git保存的不是改变，而是此时的文件本身。由于不需要遵循改变路径来计算历史版本，所以git可以快速的查阅历史版本。git可以直接提取两个commit所保存的文件系统树，并迅速的算出两个commit之间的改变。
- 同样由于上面的数据结构，git可以很方便的创建分支(branch)。实际上，git的一个分支是一个指向某个commit的指针。合并时，git检查两个分支所指的两个commit，并找到它们共同的祖先commit。git会分别计算每个commit与祖先发生的改变，然后将两个改变合并(同样，针对同一行的两个改变可能发生冲突，需要手工解决冲突)。整个过程中，不需要复制和遵循路径计算总的改变，所以效率提高很多。

VCS的发展--git

- 比如图1中有两个分支，一个 **master** 和一个 **develop**。
- 我们先沿着 **develop** 分支工作，并进行了两次提交(比如修正 **bug1**)，而 **master** 分支保持不变。
- 随后沿着 **master** 分支,进行了两次提交(比如增加输入功能)，**develop** 保持不变。
- 在最终进行图4中的合并时，我们只需要将 **C4-C2** 和 **C6-C2** 的两个改变合并，并作用在 **C2** 上，就可以得到合并后的 **C7**。合并之后，两个分支都指向 **C7**。我们此时可以删除不需要的分支 **develop**。



VCS的发展--git

- 由于git创建、合并和删除分支的成本极为低廉，所以git鼓励根据需要创建多个分支。
- 实际上，如果分支位于不同的站点(site)，属于不同的开发者，那么就构成了分布式的多分支开发模式。每个开发者都在本地复制有自己的库，并可以基于本地库创建多个本地分支工作。开发者可以在需要的时候，选取某个本地分支与远程分支合并。git可以方便的建立一个分布式的小型开发团队。比如我和朋友两人各有一个库，各自开发，并相互拉对方的库到本地库合并(如果上面master，develop代表了两个属于不同用户的分支，就代表了这一情况)。当然，git也允许集中式的公共仓库存在，或者多层的公共仓库，每个仓库享有不同的优先级。git的优势不在于引进了某种开发模式，而是给了你设计开发模式的自由。

VCS的发展--git

- 需要注意的是，GitHub尽管以git为核心，但并不是Linus创建的。事实上，Linus不接收来自GitHub的Pull Request。Linus本人将此归罪于GitHub糟糕的Web UI。但正是GitHub的Web页面让许多新手熟悉并开始使用git。

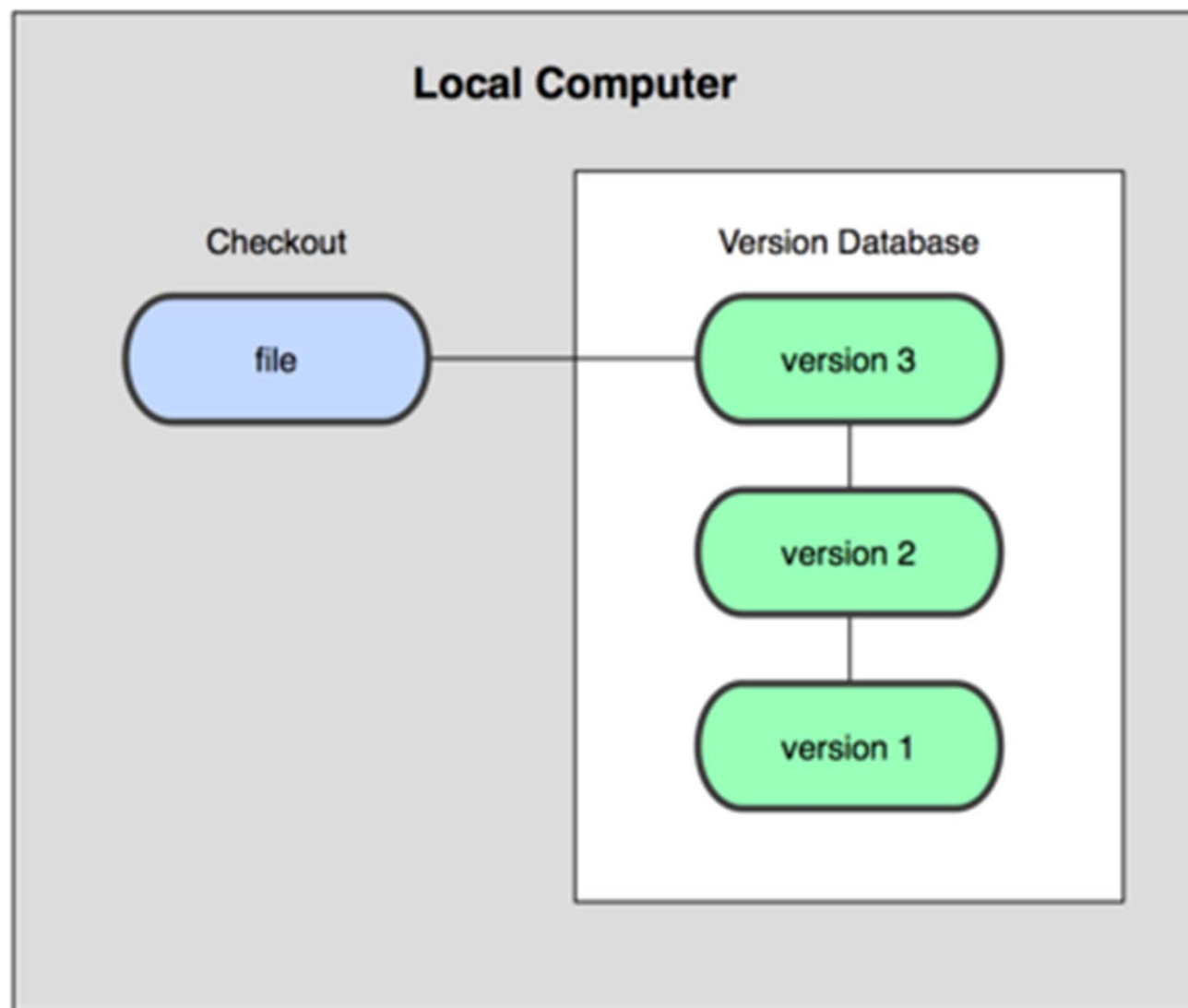
VCS的发展—三者比较总结

- VCS主流的三个工具还没有决出最终胜负
 - 或许SVN会继续在一些重要项目上发挥作用
 - 或许git会最终一统江山
 - 或许CVS可以有新的发展并最终逆袭
 - 又或许，一款新的VCS将取代所有的前辈。
-
- 一款优秀的VCS可以提高了我们管理项目的能力，降低我们犯错所可能支付的代价。随着开发项目越来越庞大和复杂，这一能力变得越来越不可缺少。花一点时间学习VCS，并习惯在工作中使用VCS，将会有意想不到的回报。



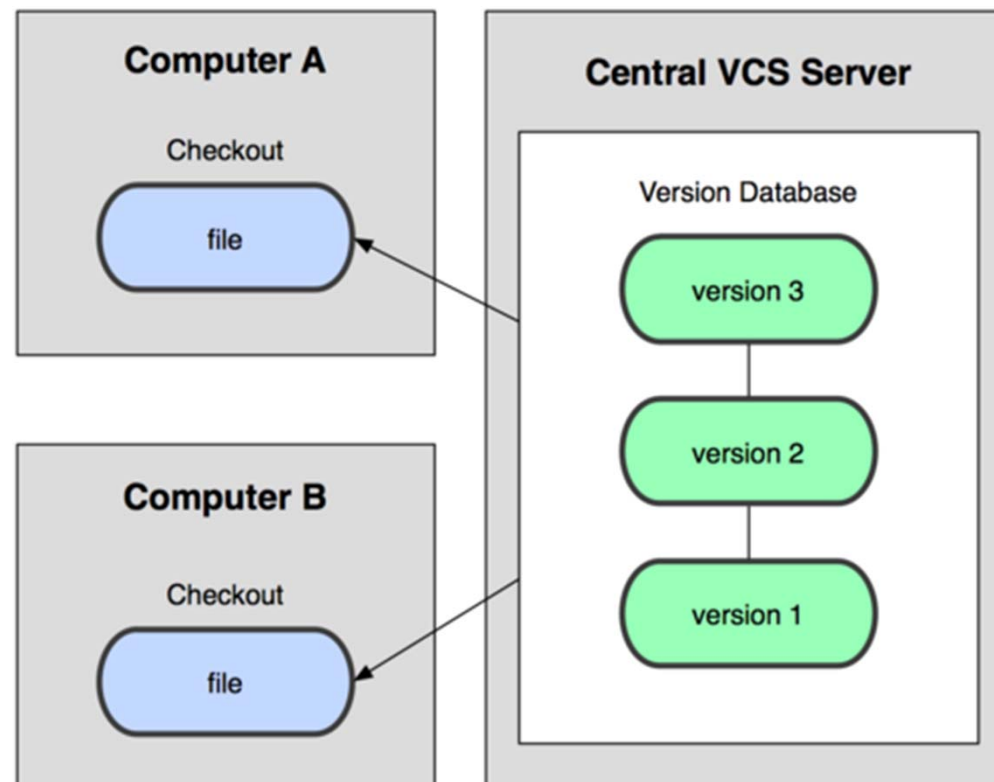
4.1 Git

本地版本控制系统



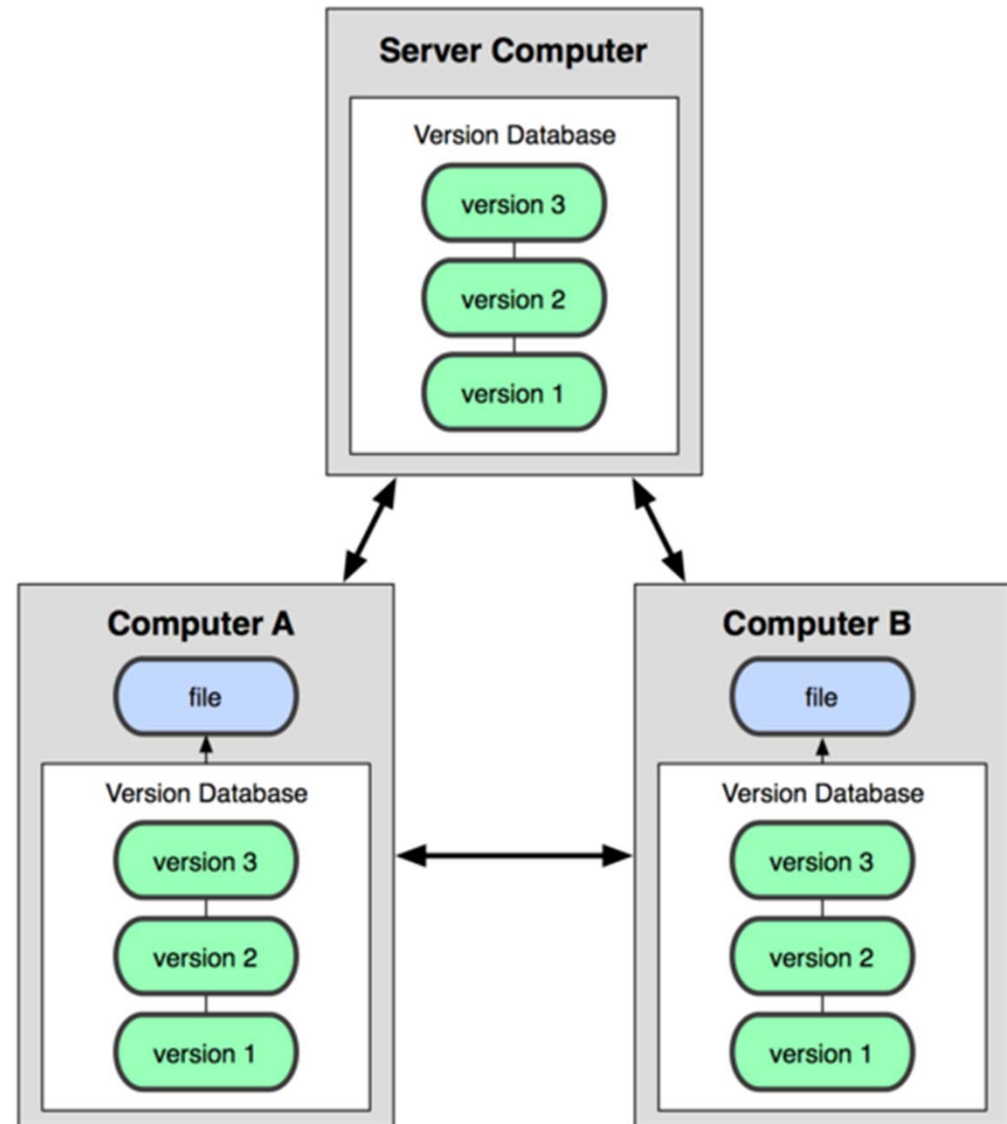
集中化的版本控制系统

- **CVS, Subversion(SVN) 以及 Perforce 等**，有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的开发者通过客户端连到这台服务器，取出最新的文件或者提交更新。



分布式版本控制系统

- 客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。
- 任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜像出来的本地仓库恢复。
- 每一次的提取操作，实际上都是一次对代码仓库的完整备份。



Git如何出现的？

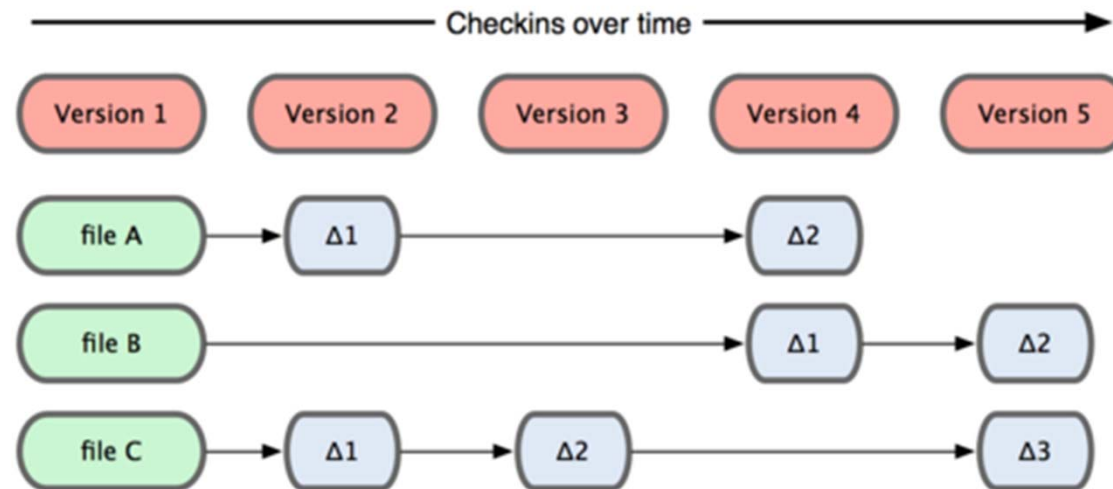
- Git是一个分布式的版本控制工具。
- Linux 内核开源项目有着为数众广的参与者，绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上；
- 最初由Linus Torvalds开发，用于管理Linux内核的开发；

Git的基本思想

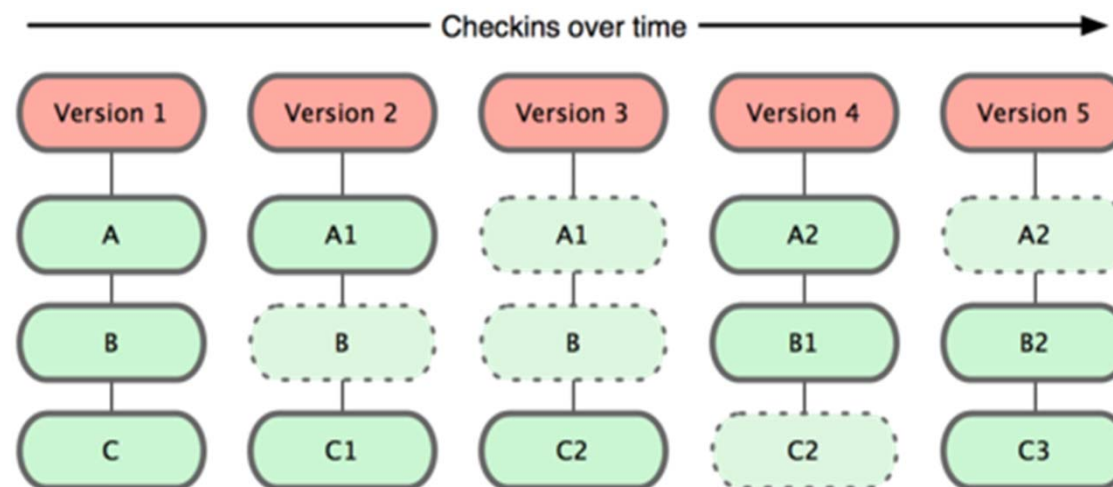
- 其他大多数SCM系统主要关心文件内容的具体差异(每次记录有哪些文件作了更新, 以及都更新了哪些行的什么内容);
- Git关心文件数据的整体是否发生变化, 并不保存这些前后变化的差异数据
 - 把变化的文件作快照后, 记录在一个微型的文件系统中。
 - 每次提交更新时, 它会纵览一遍所有文件的指纹信息并对文件作一快照, 然后保存一个指向这次快照的索引。
 - 为提高性能, 若文件没有变化, Git不会再次保存, 而只对上次保存的快照作一链接。

Git的基本思想

其他SCM软件



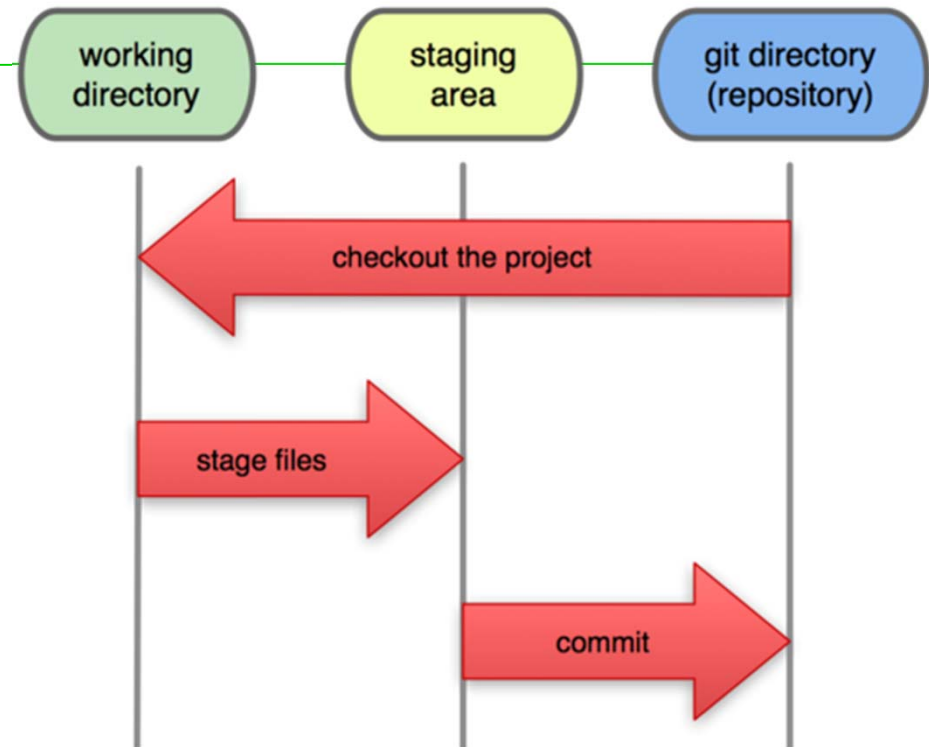
git



Git中文件的三种状态

- **Git**管理项目的三个工作区域：

- 工作目录
- 暂存区域
- 本地仓库



- 对于任何一个文件，在 **Git** 内都只有三种状态：

- 已修改(modified)：表示修改了某个文件，但还没有提交保存；
- 已暂存(staged)：表示把已修改的文件放在下次提交时要保存的清单中；
- 已提交(committed)：表示该文件已经被安全地保存在本地数据库中。

Git的工作区

- 每个项目都有一个Git目录(Git directory)，用来保存元数据和对象数据库。每次clone镜像仓库的时候，实际拷贝的就是这个目录里面的数据。
- 从Git目录中取出某个版本的所有文件和目录，用以开始后续工作，形成工作目录(working directory)，接下来就可以在工作目录中对这些文件进行编辑。
- 所谓的暂存区域(staging area)只不过是个简单的索引文件，是一个包含文件索引的目录树，像是一个虚拟的工作区。在这个虚拟工作区的目录树中，记录了文件名、文件的状态信息（时间戳、文件长度等），文件的内容并不存储其中，而是保存在 Git 对象库中。暂存区域的设计可以实现更有效、灵活的控制要提交的文件对象。

Git的基本工作流程

- 1. 在工作目录中修改某些文件。
- 2. 对修改后的文件进行快照，然后保存到暂存区域。
- 3. 提交更新，将保存在暂存区域的文件快照永久转储到Git目录中。

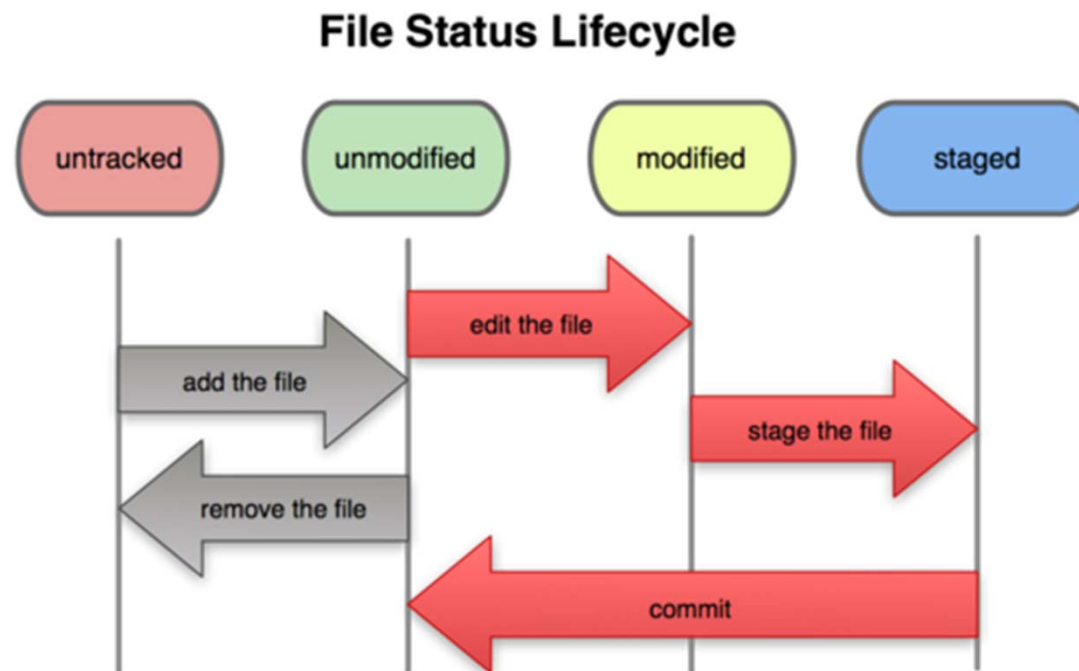
- 可以从文件所处的位置来判断状态：
 - 如果是 Git 目录中保存着的特定版本文件，就属于已提交状态；
 - 如果作了修改并已放入暂存区域，就属于已暂存状态；
 - 如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

基本的Git命令：取得项目的 Git 仓库

- 下载并在本机安装设置Git环境 <http://msysgit.github.io/>
- 在工作目录中初始化新仓库
 - 要对现有的某个项目开始用Git管理，只需到此项目所在的目录，执行**git init**命令，用 **git add** 命令告诉Git开始对这些文件进行跟踪，然后提交：
 - `git add *.c`
 - `git add readme.txt`
 - `git commit -m 'initial project version'`
- 从现有仓库克隆：复制服务器上项目的所有历史信息到本地
 - `git clone [url]`

基本的Git命令：记录每次更新到仓库

- 在工作目录对某些文件作了修改之后，Git将这些文件标为“已修改”，可提交本次更新到仓库。
- 逐步把这些修改过的文件放到暂存区域，直到最后一次性提交所有这些暂存起来的文件，如此重复。



基本的Git命令：跟踪新文件、暂存已修改文件

- 使用**git add**开始跟踪一个新文件（使某个文件纳入到**git**中管理）。
- 一个修改过的且被跟踪的文件，处于暂存状态。
- **git add**后面可以指明要跟踪的文件或目录路径。如果是目录的话，就说明要递归跟踪该目录下的所有文件。
- **git add**的潜台词：把目标文件快照放入暂存区域，也就是 **add file into staged area**，同时之前未曾跟踪过的文件标记为需要跟踪。
- 若对已跟踪的文件进行了修改，使用**git add**命令将其放入暂存区；
- 运行了**git add**之后又对相应文件做了修改，要重新**git add**。

基本的Git命令：检查当前文件状态

- 要确定哪些文件当前处于什么状态，用`git status`命令：
 - # On branch master nothing to commit (working directory clean)
当前没有任何跟踪着的文件，也没有任何文件在上次提交后更改过
 - # On branch master # Untracked files: ...
有未跟踪的文件，使用`git add`开始跟踪一个新文件
 - # On branch master # Changes to be committed:
有处于已暂存状态的文件

基本的Git命令：查看已暂存和未暂存的更新

- **git status**回答：当前做的哪些更新还没有暂存？有哪些更新已经暂存起来准备好了下次提交？
- 如果要查看具体修改了什么地方，可以用**git diff**命令，使用文件补丁的格式显示具体添加和删除的行。
- 要查看尚未暂存的文件更新了哪些部分，不加参数直接输入**git diff**：
 - 比较的是工作目录中当前文件和暂存区域快照之间的差异，也就是修改之后还没有暂存起来的变化内容。
- 若要查看已暂存起来的文件和上次提交时的快照之间的差异，可以用 **git diff --cached** 命令。

基本的Git命令：提交更新

- 在使用`git commit`命令进行提交之前，要确认是否还有修改过的或新建的文件没有`git add`过，否则提交的时候不会记录这些还没暂存起来的变化。
 - 每次准备提交前，先用`git status`进行检查，然后再运行提交命令`git commit`。
- 提交后返回结果：
 - 当前是在哪个分支(master)提交的
 - 本次提交的完整 SHA-1 校验和是什么
 - 在本次提交中，有多少文件修订过、多少行添改和删改过。
- 提交时记录的是放在暂存区域的快照，任何还未暂存的仍然保持已修改状态，可以在下次提交时纳入版本管理。每一次运行提交操作，都是对项目做一次快照，以后可以回到这个状态，或者进行比较。

基本的Git命令：跳过使用暂存区域、移除文件

- Git提供了一个跳过使用暂存区域的方式，只要在提交的时候，给 **git commit** 加上 **-a** 选项，Git 就会自动把所有已经跟踪过的文件暂存起来一并提交，从而跳过 **git add** 步骤；
- 使用 **git rm** 命令从Git中移除某个文件，把它从已跟踪文件清单(暂存区域)中移除，并连带从工作目录中删除指定的文件。

基本的Git命令：对远程仓库的操作

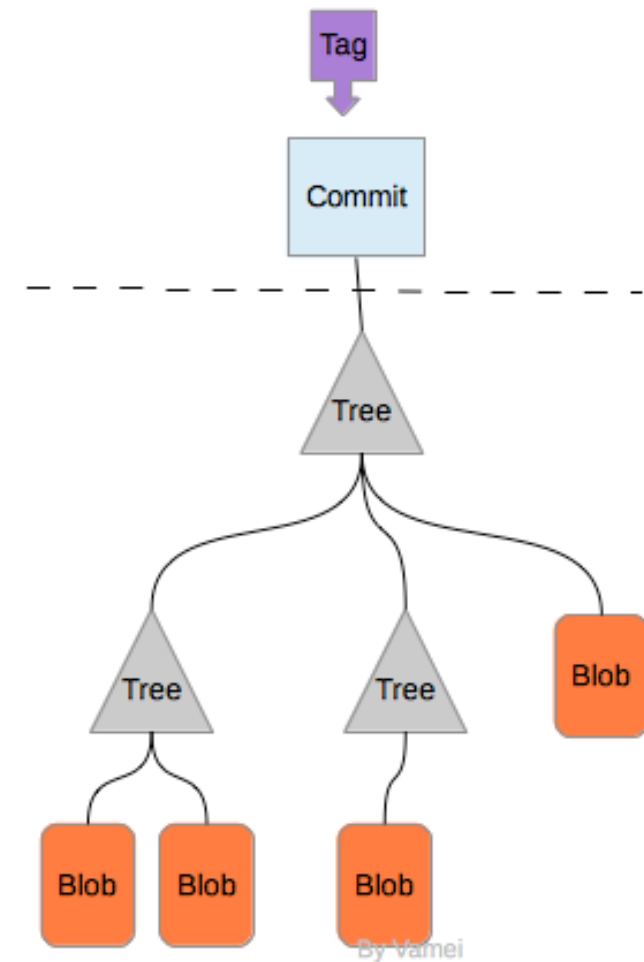
- 远程仓库：托管在网络上的项目仓库；
- 多人协作开发某个项目时，需要管理这些远程仓库，以便推送或拉取数据，分享各自的工作进展。
- 管理远程仓库：添加远程库、移除废弃的远程库、管理各式远程库分支、定义是否跟踪这些分支。
 - git remote：获取当前配置的所有远程仓库；
 - git remote add [shortname] [url]：添加一个远程仓库；
 - git fetch：从远程仓库抓取数据到本地；
 - git push [remote-name] [branch-name]：将本地仓库中的数据推送到远程仓库；
 - git remote show [remote-name]：查看某个远程仓库的详细信息；
 - git remote rm：从本地移除远程仓库；

Commit

- 在 **Git** 中提交时，会保存一个提交(commit)对象，该对象包含一个指向暂存内容快照的指针，包含本次提交的作者等相关附属信息，包含零个或多个指向该提交对象的父对象指针：
 - 首次提交是没有直接祖先的；
 - 普通提交有一个祖先；
 - 由两个或多个分支合并产生的提交则有多个祖先。

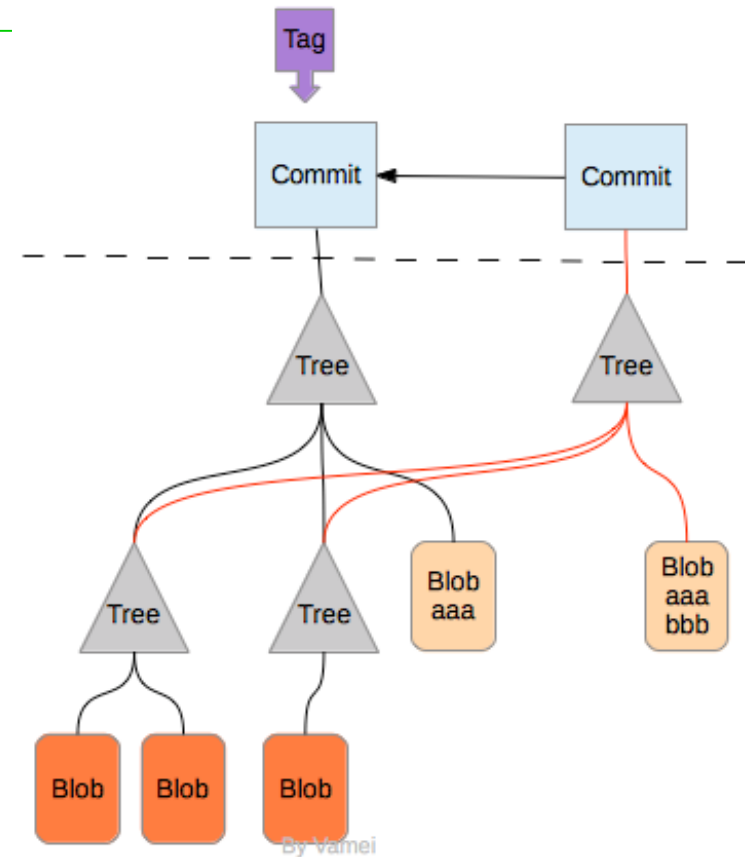
Commit

- 对于一个开发项目，**git**会保存**blob**, **tree**, **commit**和**tag**四种对象。
 - 文件被保存为**blob**对象。
 - 文件夹被保存为**tree**对象。**tree**对象保存有指向文件或者其他**tree**对象指针。
- 上面两个对象类似于一个**UNIX**的文件系统，构成了一个文件系统树。
 - 一个**commit**对象代表了某次提交，它保存有修改人，修改时间和附加信息，并指向一个文件树。这一点与**SVN**类似，即每次提交为一个文件系统树。
 - 一个**tag**对象包含有**tag**的名字，并指向一个**commit**对象。
- 虚线下面的对象构成了一个文件系统树。在**git**中，一次**commit**实际上就是一次对文件系统树的快照(**snapshot**)。



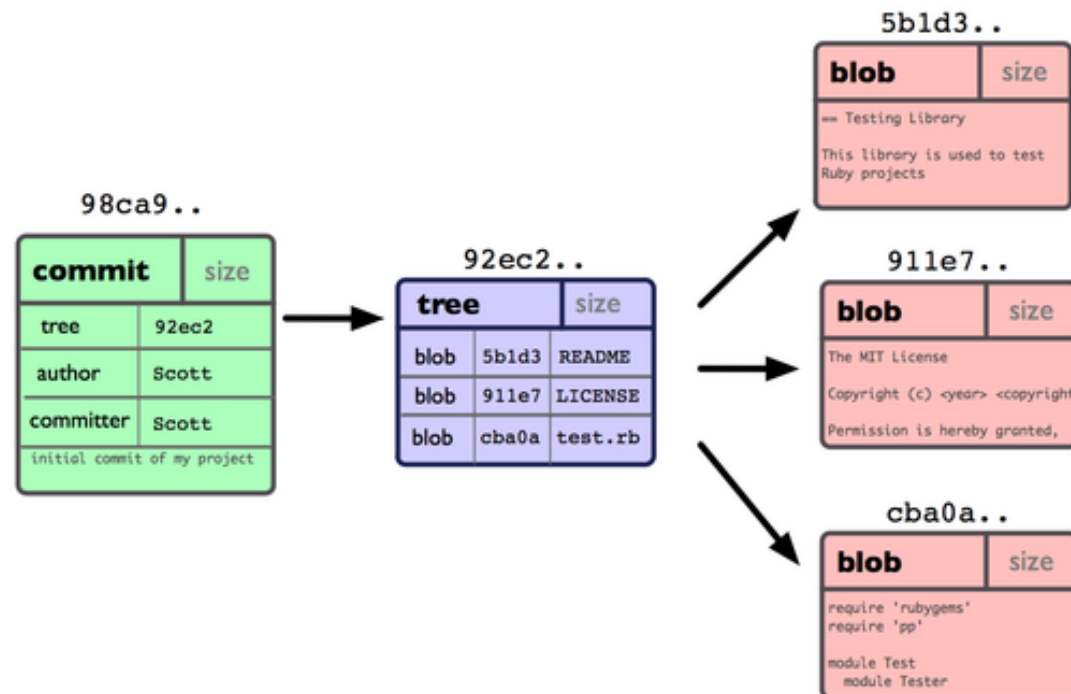
Commit

- 在整个开发过程中，可能会有许多次提交(commit)。每次commit的时候，git并不总是复制所有的对象。git会检验所有对象的HASH值。如果该对象的HASH值已经存在，说明该对象已经保存过，并且没有发生改变，所以git只需要调整新建tree或者commit中的指针，让它们指向已经保存过的对象就可以了。
- git在commit的时候，只会新建HASH值发生改变的对象。如图所示，我们创建新的commit的时候，只需要新建一个commit对象，一个tree对象和一个blob对象就足够了，而不需要新建整个文件系统树。



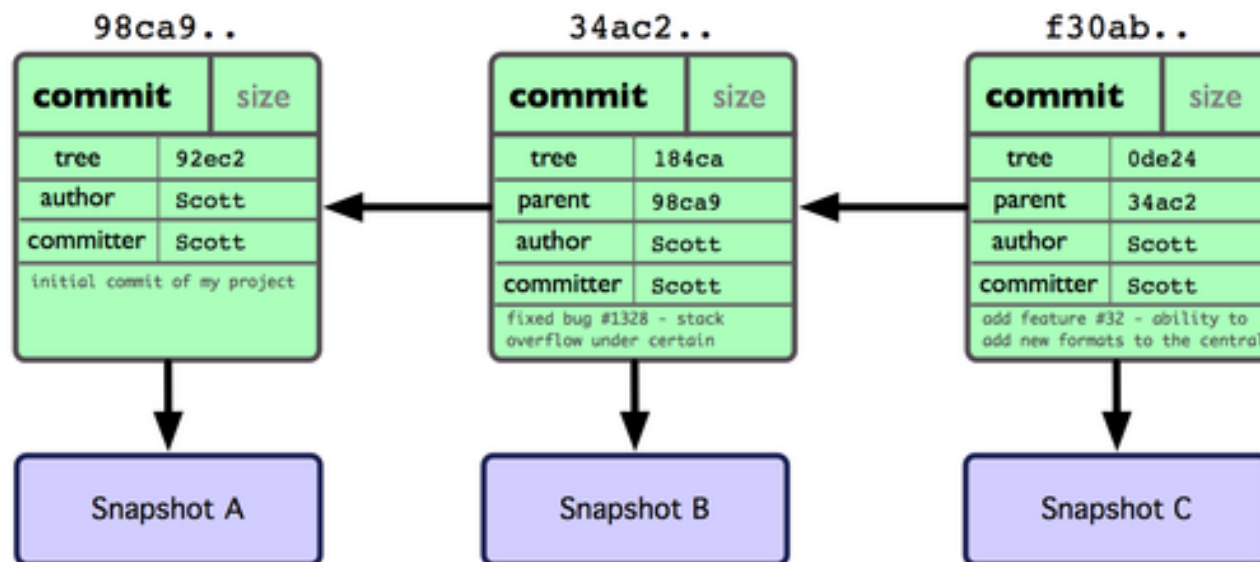
单个提交(Commit)对象在仓库中的数据结构

- 多个表示待提交的文件快照内容的**blob**对象；
- 一个记录着目录树内容及其中各个文件对应 **blob** 对象索引的 **tree** 对象；
- 一个包含指向 **tree**对象(根目录)的索引和其他提交信息元数据的**commit**对象。



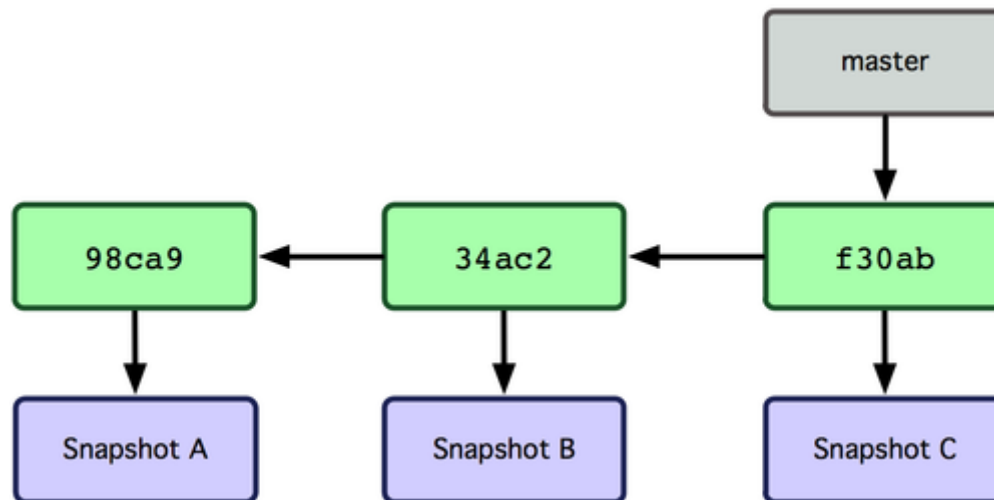
多次提交之后

- 某些文件修改后再次提交，这次的commit对象会包含一个指向上次提交对象的指针。



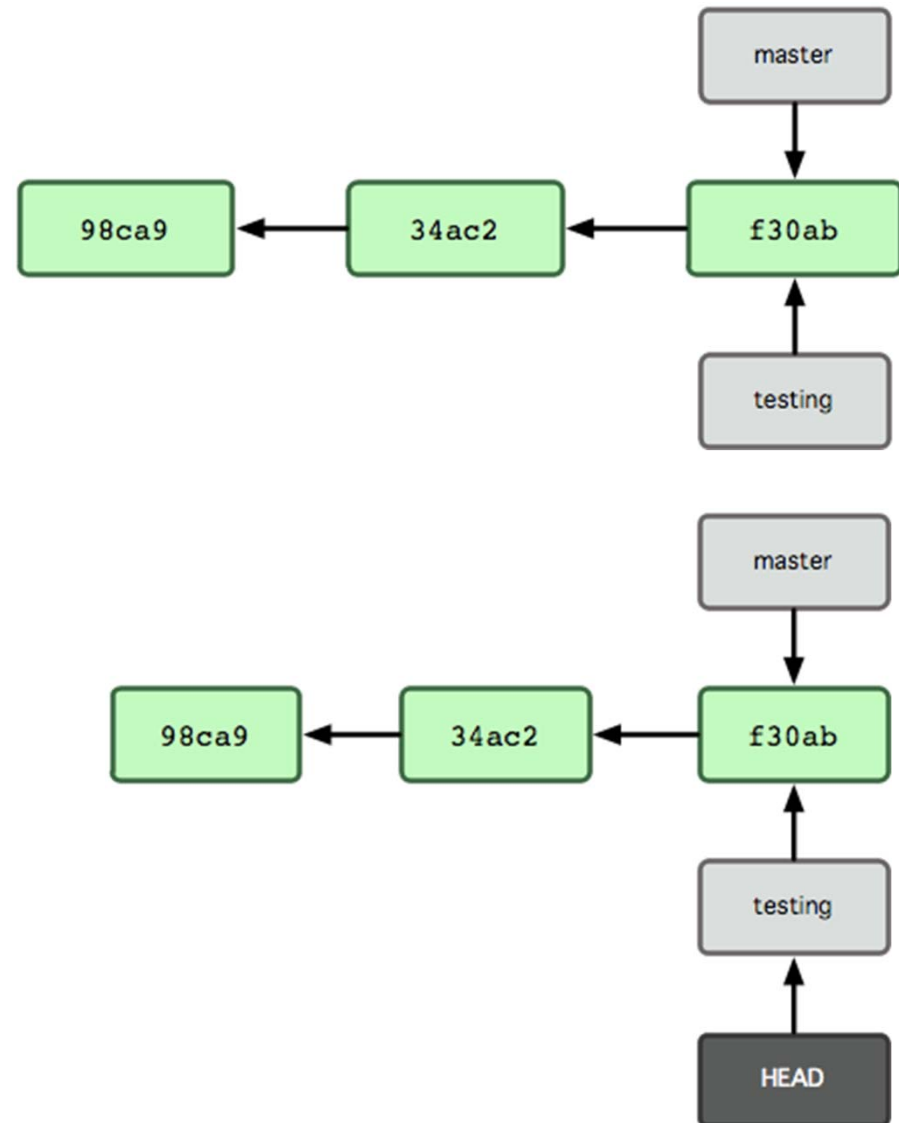
Git的分支

- Git中的分支本质上仅仅是个指向 **commit** 对象的可变指针。
- Git使用**master**作为分支的默认名字。
- 若干次提交后，**master**分支指向最后一次提交对象，它在每次提交的时候都会自动向前移动。



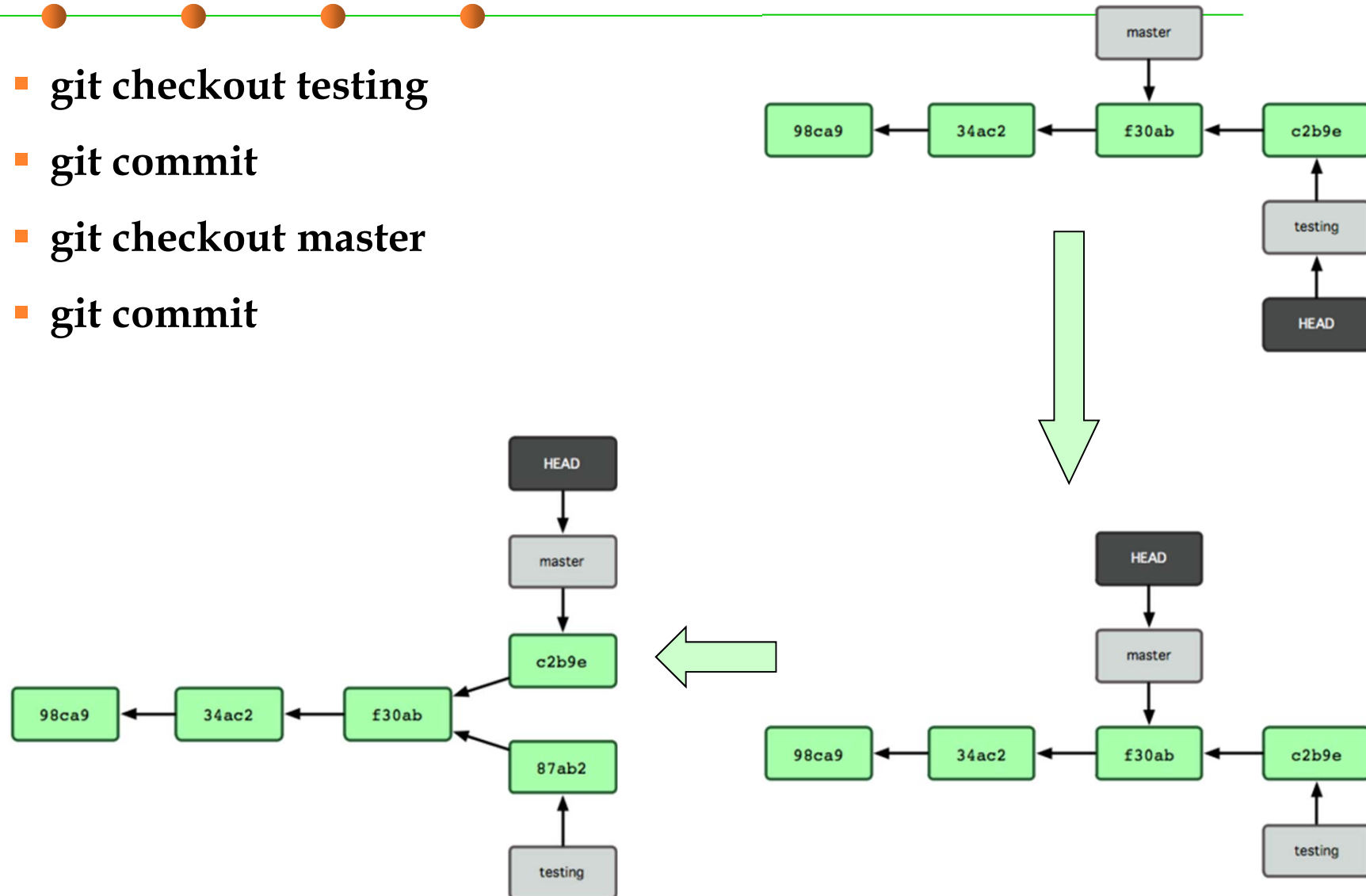
创建一个新的分支

- **git branch**在当前commit对象上新建一个分支指针；
- **Git**使用一个叫做**HEAD**的特别指针来获知你当前在哪个分支上工作。
- 要切换到其他分支，可以执行**git checkout**命令。



在新的分支上提交

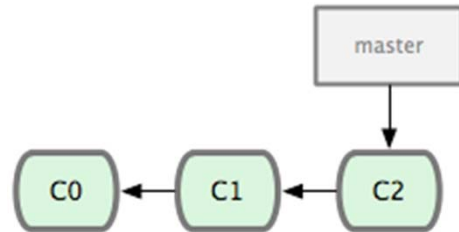
- git checkout testing
- git commit
- git checkout master
- git commit



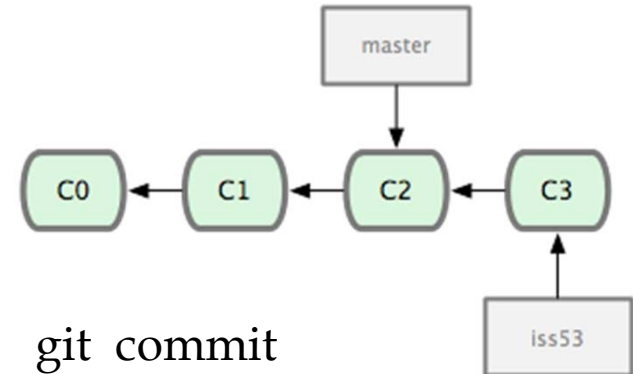
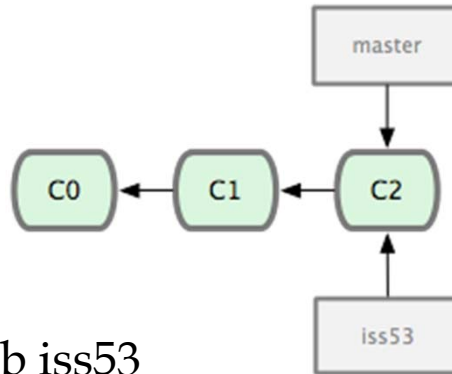
Git的分支机制的优越性

- Git 中的分支实际上仅是一个包含所指对象校验和(40 个字符长度 SHA-1字串)的文件，非常方便；
- 传统的版本控制系统则采用将文件备份到目录的方式。
- Git的分支实现与项目复杂度无关，可以在几毫秒的时间内完成分支的创建和切换。
- 因为每次提交时都记录了祖先信息(即parent 对象)，将来要合并分支时，寻找恰当的合并基础(即共同祖先)，实现起来非常容易。

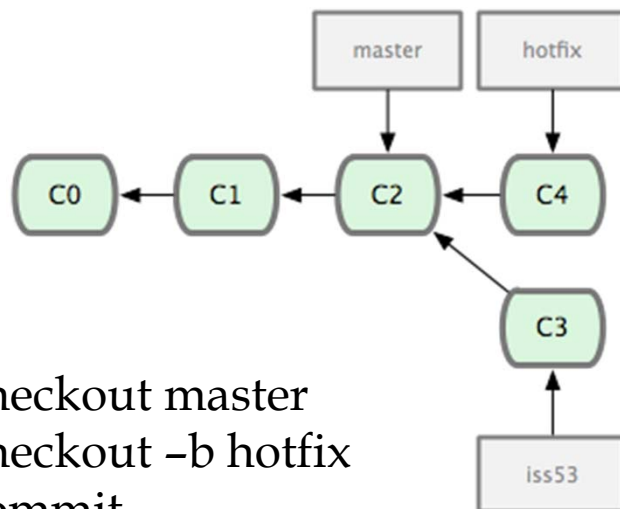
分支的新建与合并：例子



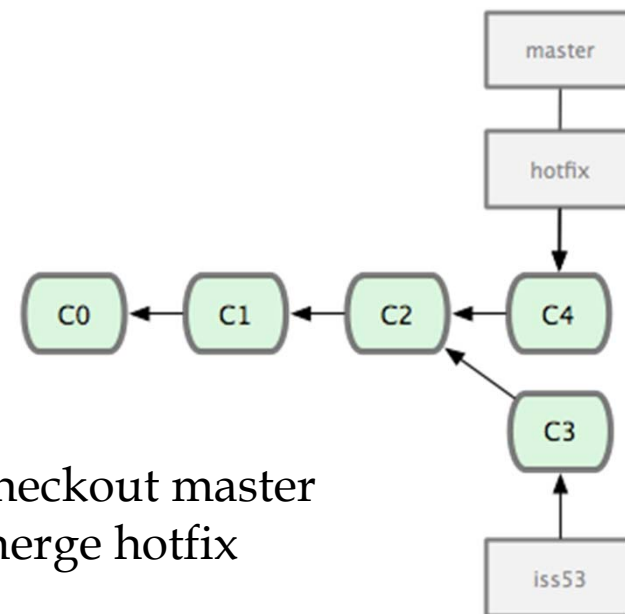
git checkout -b iss53



git commit

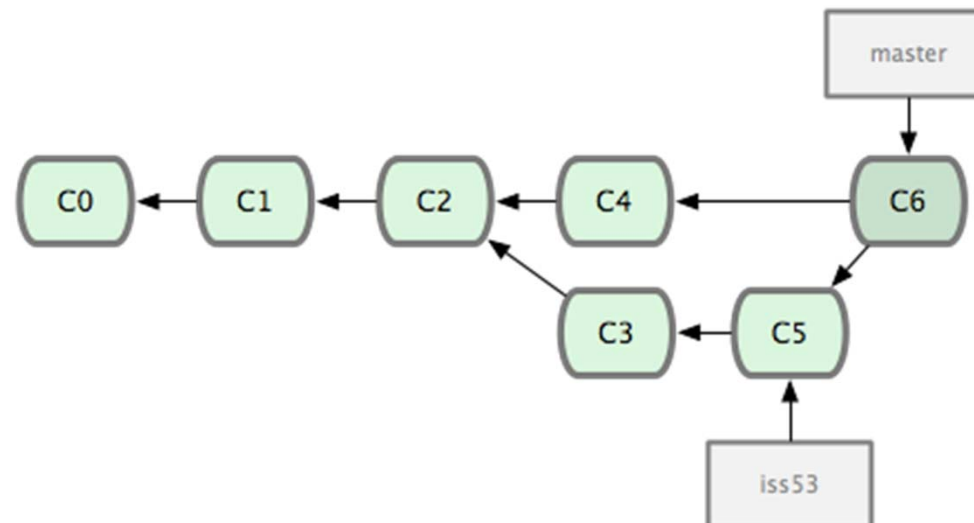
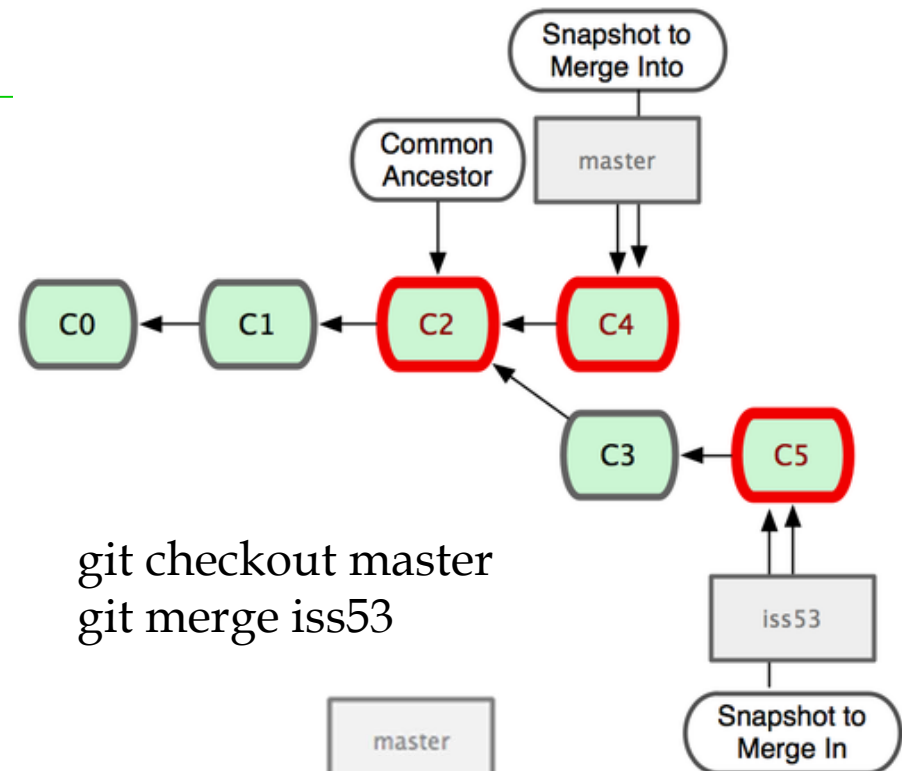
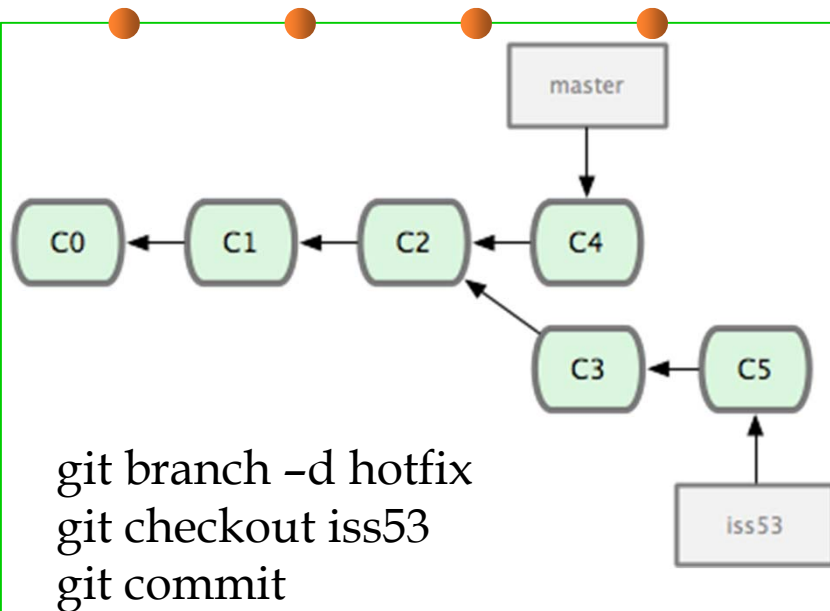


git checkout master
git checkout -b hotfix
git commit



git checkout master
git merge hotfix

分支的新建与合并：例子



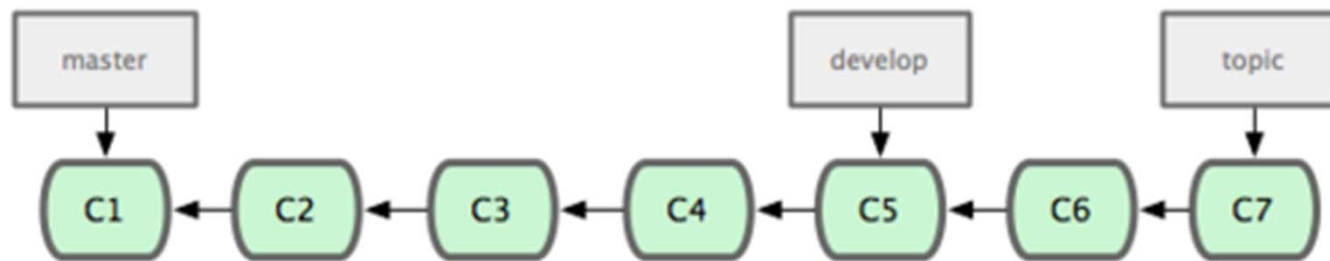
分支的新建与合并：有冲突时如何？

- 如果在不同的分支中都修改了同一个文件的同一部分，**Git**无法干净地把两者合到一起，需要依赖于人的裁决。
- 使用**git status**查看任何包含未解决冲突的文件，在有冲突的文件里加入标准的冲突解决标记，可以通过它们来手工定位并解决这些冲突。

```
<<<<<<< HEAD
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

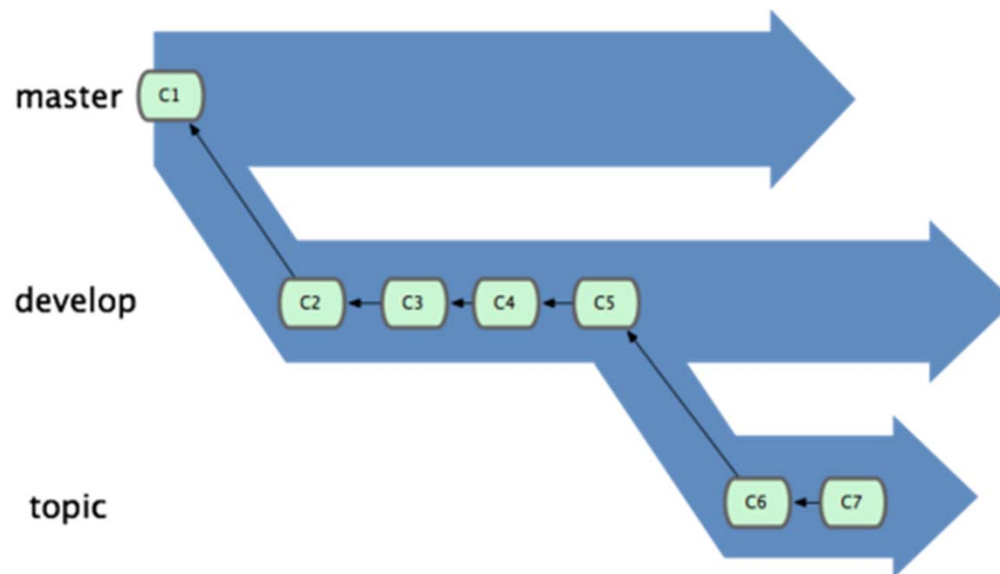
利用分支进行开发的工作流程

- 长期分支：可以同时拥有多个开放的分支，每个分支用于完成特定的任务，随着开发的推进，可以随时把某个特性分支并到其他分支中。
 - 仅在 `master` 分支中保留完全稳定的代码，即已经发布或即将发布的代码。
 - 同时还有一个名为`develop`或`next`的平行分支，专门用于后续的开发，或仅用于稳定性测试，一旦进入某种稳定状态，便可以把它合并到`master`里。
 - 这样，在确保这些已完成的特性分支能够通过所有测试并且不会引入更多错误之后，就可以合并到主干分支中，等待下一次的发布。
 - 随着提交对象不断右移的指针。稳定分支的指针总是在提交历史中落后一大截，而前沿分支总是比较靠前。



长期分支

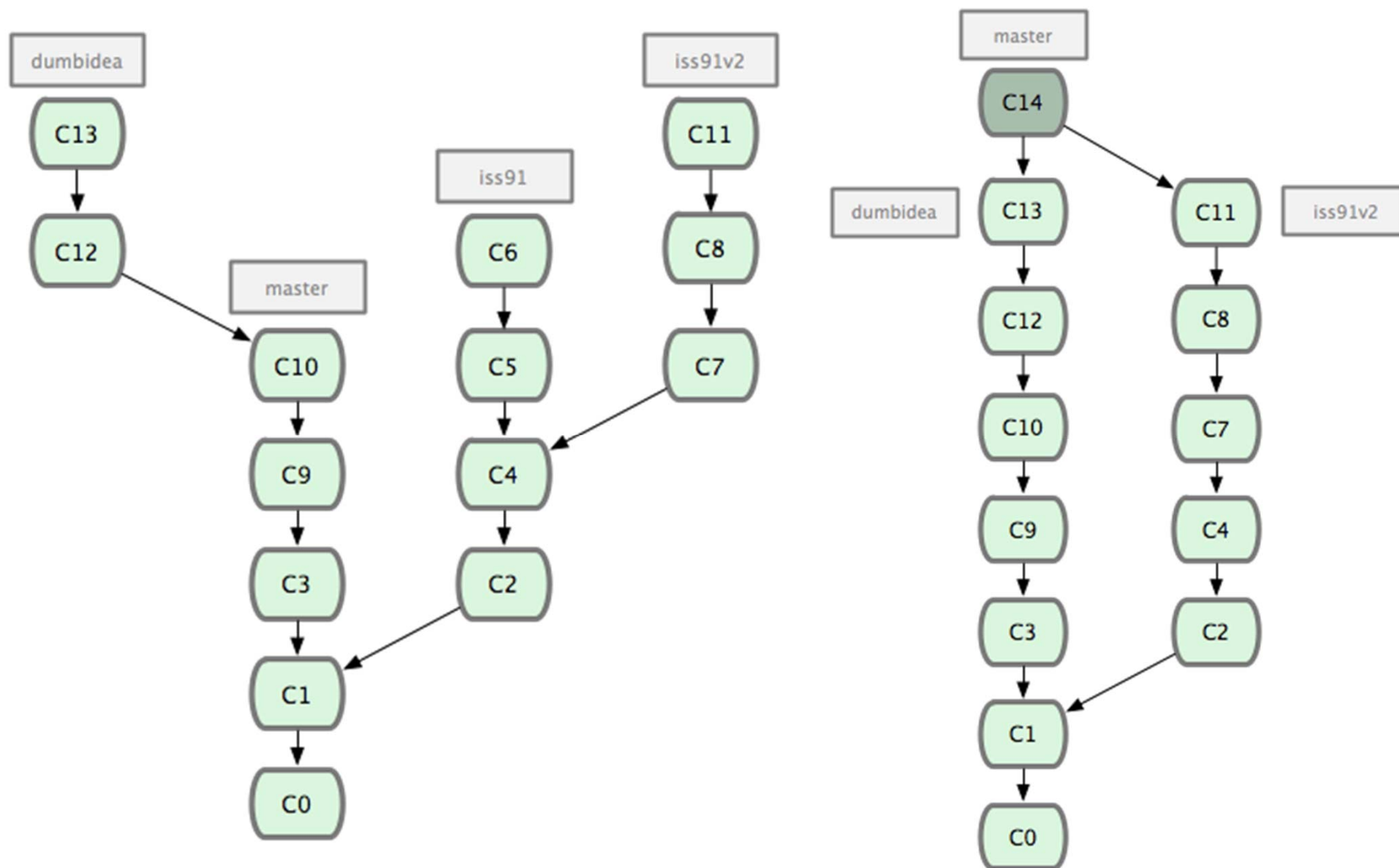
- 这么做的目的是拥有不同层次的稳定性：当这些分支进入到更稳定的水平时，再把它们合并到更高层分支中去。



特性分支

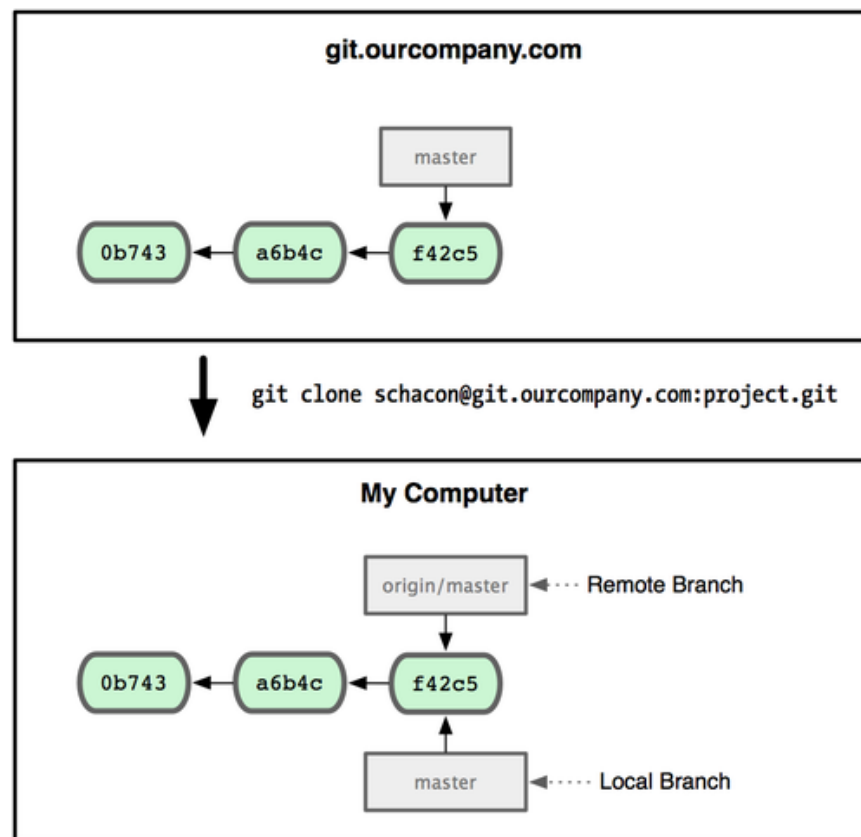
- 特性分支(Topic)是指一个短期的，用来实现单一特性或与其相关工作的分支。
- 创建特性分支，在提交了若干更新后，把它们合并到主干分支，然后删除，从而支持迅速且完全的进行语境切换。
- 因为开发工作分散在不同的流水线里，每个分支里的改变都和它的目标特性相关，浏览代码之类的事情因而变得更简单了。
- 可以把作出的改变保持在特性分支中几分钟，几天甚至几个月，等它们成熟以后再合并，而不用在乎它们建立的顺序或者进度。

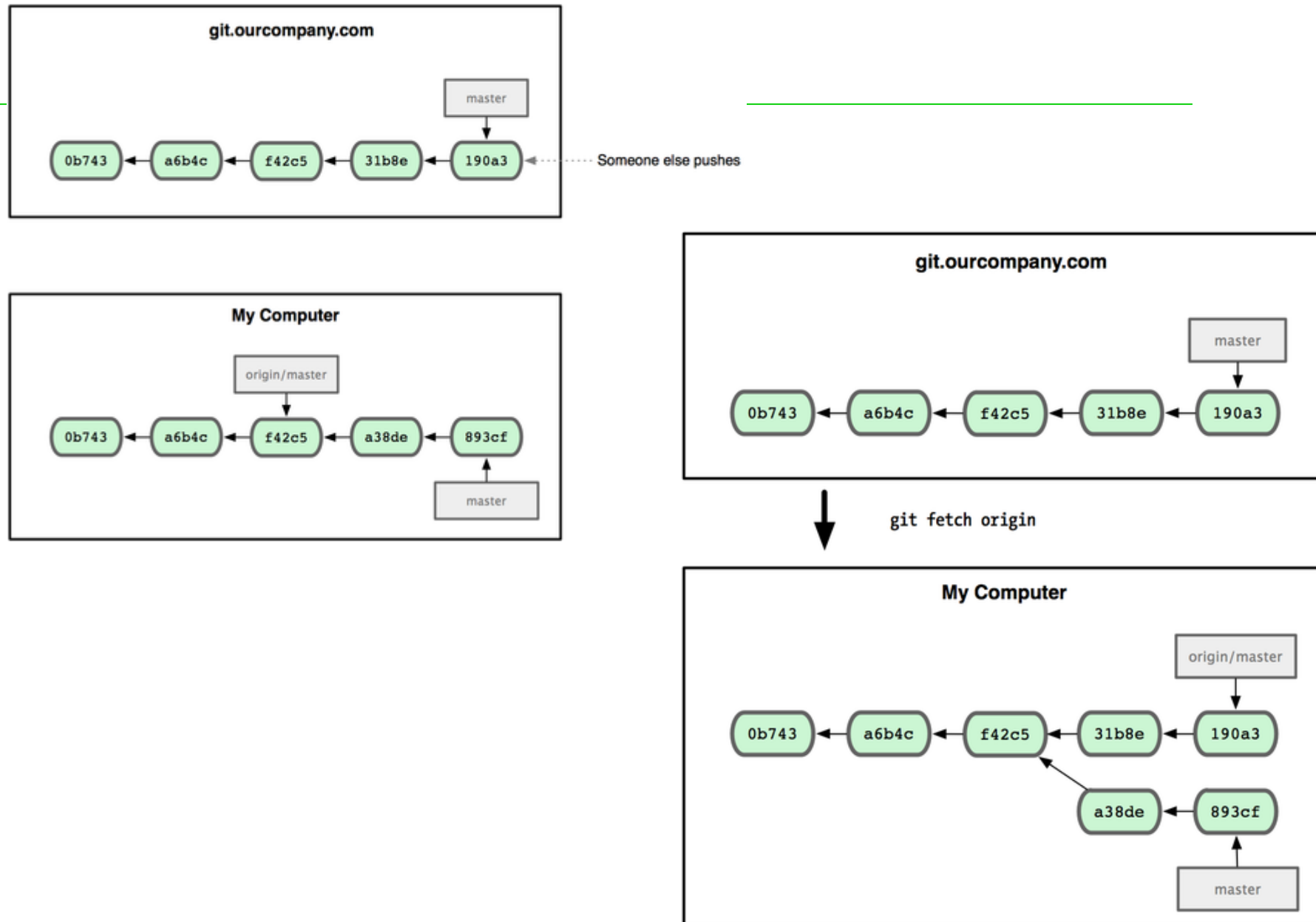
特性分支的例子



远程分支

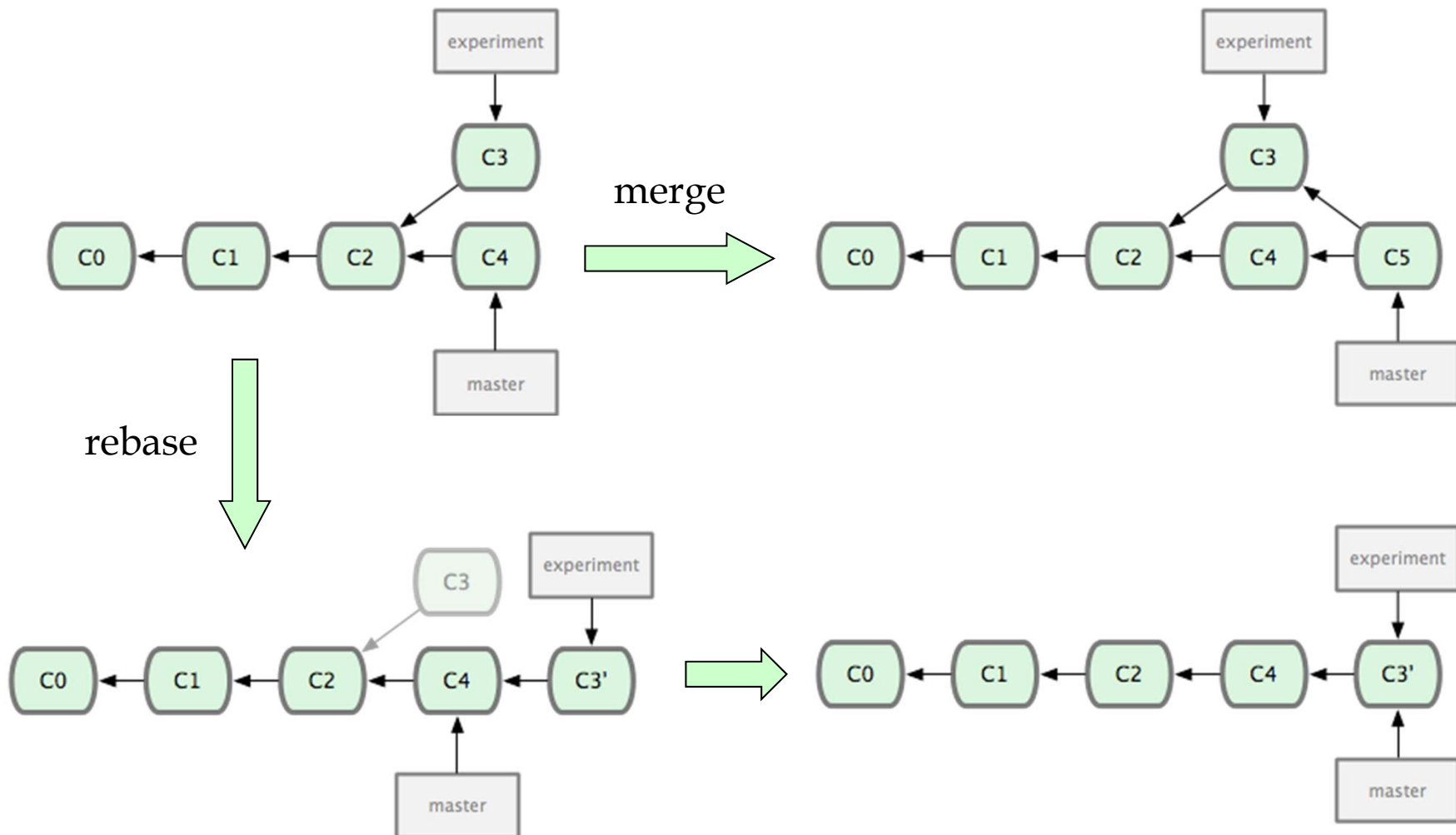
- 远程分支(**remote branch**): 对远程仓库中的分支的索引, 类似于书签, 提醒上次连接远程仓库时上面各分支的位置。
- 使用“(远程仓库名)/(分支名)”表示。





分支的衍合(rebase)

- 除了 `git merge`，另一种进行分支合并的方法：`git rebase`。





4.2 Github

GitHub

Build software
better, together.

- **GitHub:** 共享虚拟主机服务，用于存放使用**Git**版本控制的软件代码和内容项目，是目前最流行的**Git**存取站点，同时提供付费账户和为开源项目提供的免费账户。
- 支持社会化软件开发，允许用户跟踪其他用户、组织、软件库的动态，对软件代码的改动和 **bug** 提出评论等。
- 提供了图表功能，用于显示开发者们怎样在代码库上工作以及软件的开发活跃程度。



Git hub

Build software
better, together.

- GitHub快速入门可参考

<http://www.worldhello.net/gotgithub/index.html>

主要内容

1. 软件演化
2. 软件维护与软件再工程
3. 软件配置管理(SCM)
4. Git/Github
5. 持续集成

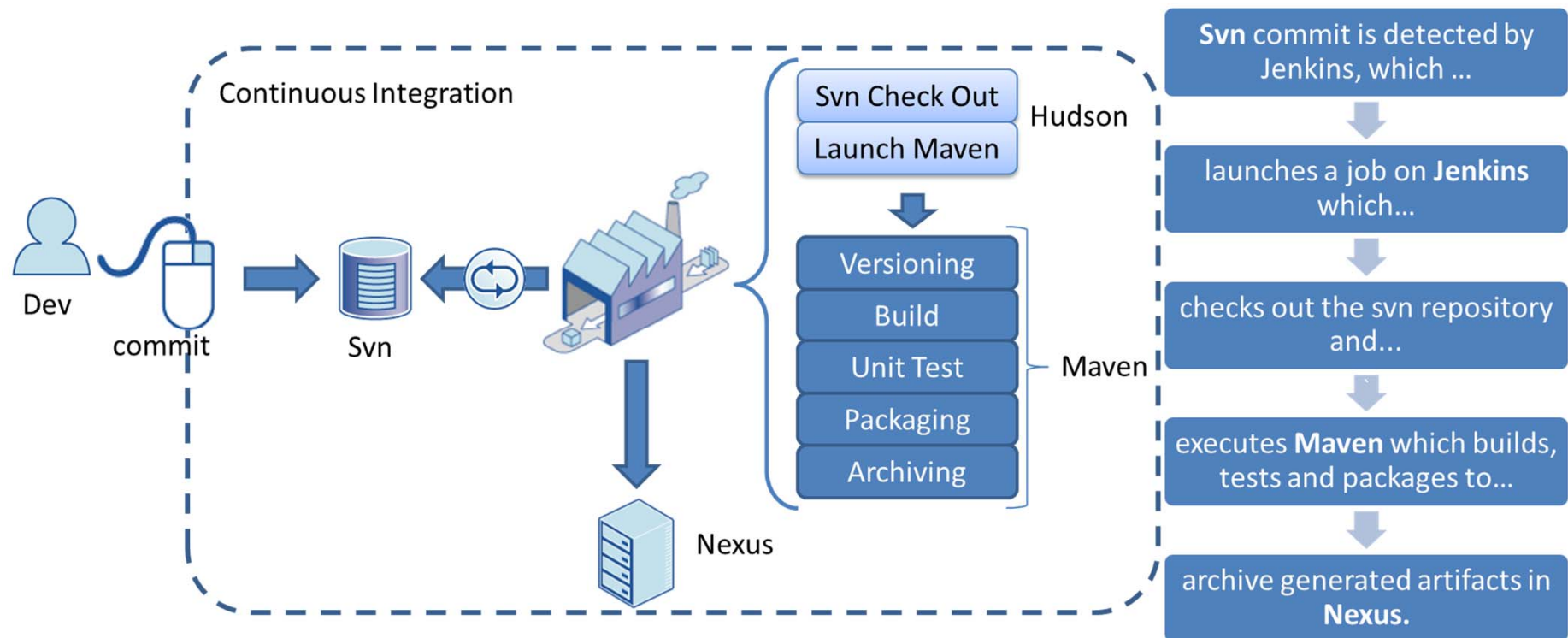
持续集成

- 持续集成：敏捷开发的一项重要实践。
 - Martin Fowler：团队开发成员经常集成他们的工作，每个成员每天至少集成一次，每天可能会发生多次集成。每次集成都通过自动化的构建(包括编译，发布，自动化测试)来验证，从而尽快地发现集成错误，大大减少集成的问题，让团队能够更快的开发内聚的软件。
- 价值：
 - 减少风险：不是等到最后再做集成测试，而是每天都做；
 - 减少重复过程：通过自动化来实现；
 - 任何时间、任何地点生成可部署的软件；
 - 增强项目的可见性；
 - 建立团队对开发产品的信心；

持续集成

- 所有的开发人员需要在本地机器上做本地构建，然后再提交到版本控制库中，从而确保他们的变更不会导致持续集成失败。
- 开发人员每天至少向版本控制库中提交一次代码。
- 开发人员每天至少需要从版本控制库中更新一次代码到本地机器。
- 需要有专门的集成服务器来执行集成构建，每天要执行多次构建。
- 每次构建都要**100%**通过。
- 每次构建都可以生成可发布的产品。
- 修复失败的构建是优先级最高的事情。

持续集成





课外阅读：Pressman教材第22章



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

软件工程

結束

2017年9月25日