

大连理工大学

Dalian University Of Technology

# 系统持续集成设计

大连理工大学

软件学院

马瑞新



# 目录

1. 系统配置管理
2. 系统代码检查
3. 系统编译构建
4. 系统测试管理

# 系统配置管理

# 版本控制系统简介

- 版本控制系统

- 定义:

- 从狭义上来说，它是软件项目开发过程中用于储存我们所写的代码所有修订版本的软件
    - 事实上我们可以将任何对项目有帮助的文档交付版本控制系统进行管理

- 目的:

- 实现开发团队并行开发、提高开发效率
    - 对软件开发进程中文件或目录的发展过程提供有效的追踪手段，保证在需要时可回到旧版本，避免文件丢失、修改的丢失和相互覆盖

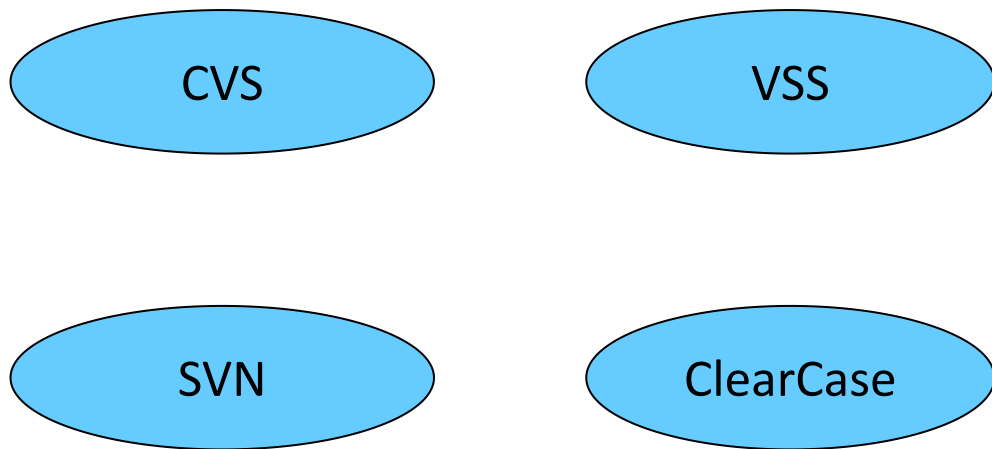
# 版本控制系统简介

- 版本控制系统
  - 好处：减轻开发人员的负担，节省时间，同时降低人为错误
    - 空间
      - 保证集中统一管理，解决一致性和冗余问题
    - 时间
      - 解决冗余、事务性处理并发性问题

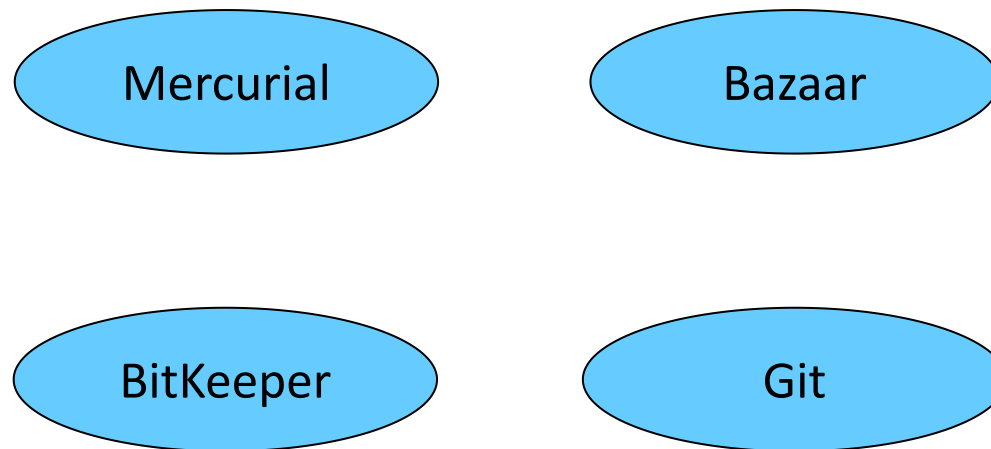
# 版本控制系统简介

- 常见代码版本控制系统

集中式版本控制系统



分布式版本控制系统



# 版本控制系统简介

- 国外常见的云端代码托管服务
  - GitHub
    - 国外最大的社交编程及代码托管网站
    - 提供对源代码的存储和复杂的版本控制
    - 有效支持跨地域协作开发
    - 开源项目免费，而私有仓库要收费
  - Bitbucket
    - 支持git和mercurial两种版本控制
    - 功能上与github不相上下
    - 仓库既可公开也可私有，5人以下仓库免费
  - GitLab
    - 用于仓库管理系统开源项目
    - 与GitHub类似的功能
    - 私有仓库免费

# 版本控制系统简介

- 国内常见的云端代码托管服务
  - 软件开发云配置管理服务
    - 基于Git的云端代码托管服务
    - 支持本地Git客户端操作
    - 支持在线浏览和编辑文件、提交和审核合并请求、查看提交历史、查看仓库统计等操作
    - 暂仅有私有仓库，5人以下仓库免费



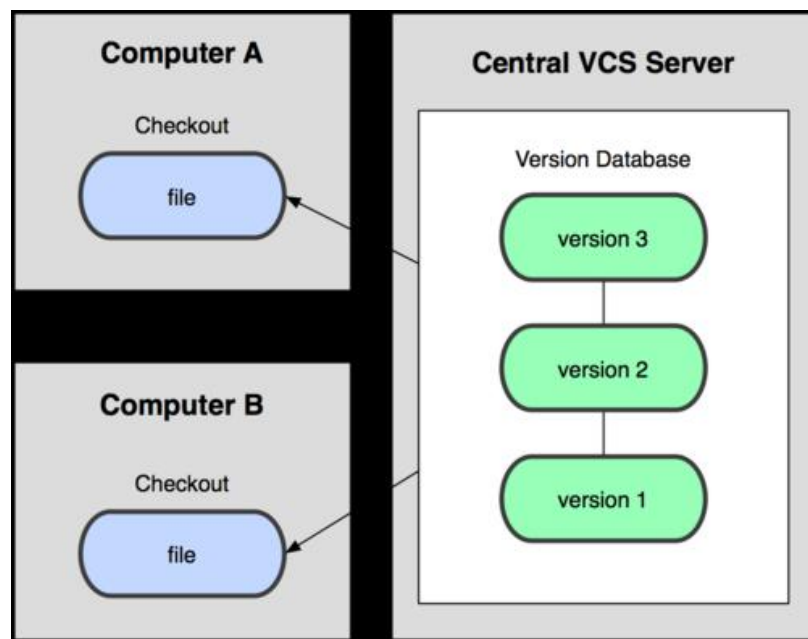
# Git简介

- Git是一款开源的**分布式版本控制系统** (Distributed Version Control System) ，诞生于2002年，由Linux之父Linus Torvalds带领Linux开源社区开发完成。初衷是用其管理Linux内核的庞大的开源代码。
- Git的设计理念包括以下几点：
  - **速度快**
  - **设计简单**
  - **强力支持非线性开发，允许上千分支并行开发**
  - **完全的分布式**
  - **有能力高效管理类似Linux内核一样的超大规模项目（速度和数据量）**

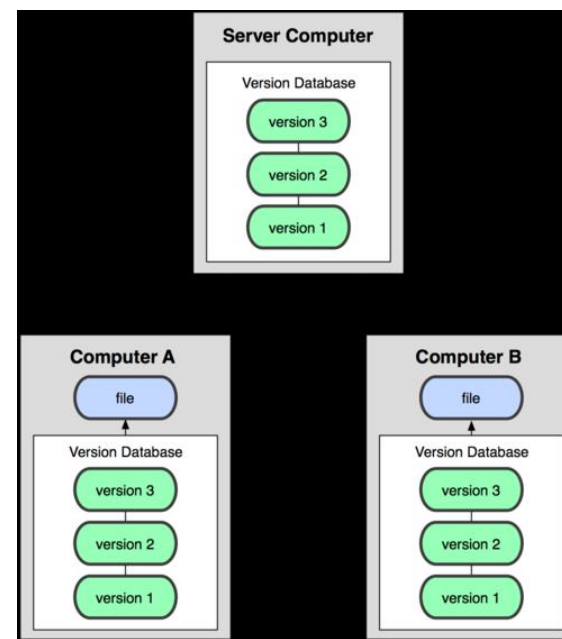
# Git VS SVN(1/7)

- 不同于SVN，Git是一款分布式的版本控制工具。以下两幅图展示了两种工具不同的工作模式。

集中式的版本控制系统工作模型

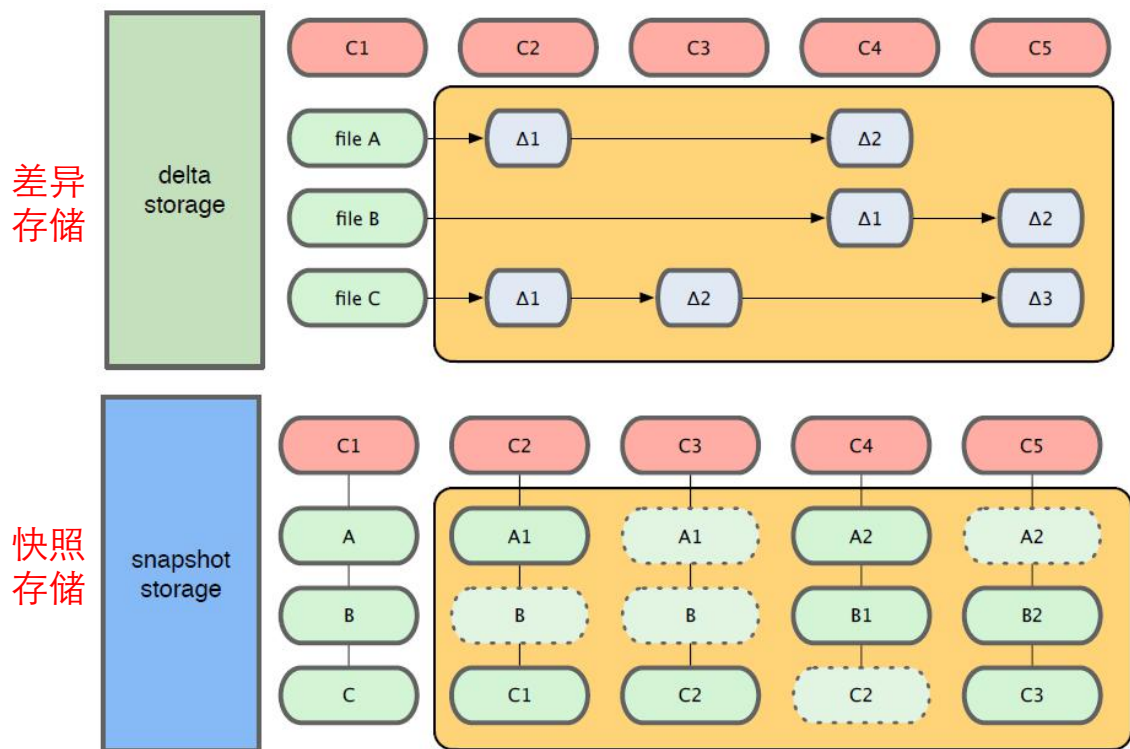


分布式的版本控制系统工作模型



# Git VS SVN(2/7)

- Git 关心整个仓库的文件是否发生变化，而大多数版本控制系统则只关心内容变化了的文件。若文件没有变化，其他版本系统不会记录，Git 而会对未变化的文件的做一链接。



# Git VS SVN(3/7)

- 版本库的安全性高
  - Git
    - 分布式系统，每个用户相当于一个备份
    - 通过SHA1哈希保证数据的完整性，防止恶意篡改
  - SVN
    - 集中式存在单点故障的风险
    - 服务器端历史数据被篡改时，客户端难以发现

# Git VS SVN(4/7)

- 分支功能强大
  - Git
    - 便于查询和追溯分支间的提交历史
    - 支持双向合并
  - SVN
    - 分支不支持提交隔离
    - 一次提交可同时更改主线和分支的内容，无法查询和追溯分支间的提交历史

# Git VS SVN(5/7)

- 更灵活的发布控制
  - Git
    - 设置只有发布管理员才有权限推送的版本库或者分支，用于稳定发布版本的维护
    - 设置只有项目经理、模块管理员才有权限推送的版本库或者分支，用于整合测试

# Git VS SVN(6/7)

- 隔离开发、提交审核
  - Git
    - 支持团队成员自建分支和版本库
    - 通过合并请求或从成员个人版本库、分支获取提交，从提交说明、代码规范等方面对提交逐一审核

# Git VS SVN(7/7)

- 对合并更好的支持：更少的冲突和更好的解决冲突
  - Git
    - 基于DAG（有向非环图）的设计比SVN的线性提交提供更好的合并追踪，避免不必要的冲突，提高工作效率
    - 基于对内容的追踪而非对文件名追踪，所以遇到一方或双方对文件名更改时，Git能够很好进行自动合并或提供工具辅助合并。而SVN遇到同样问题时会产生树冲突，解决起来很麻烦



# Git基本概念

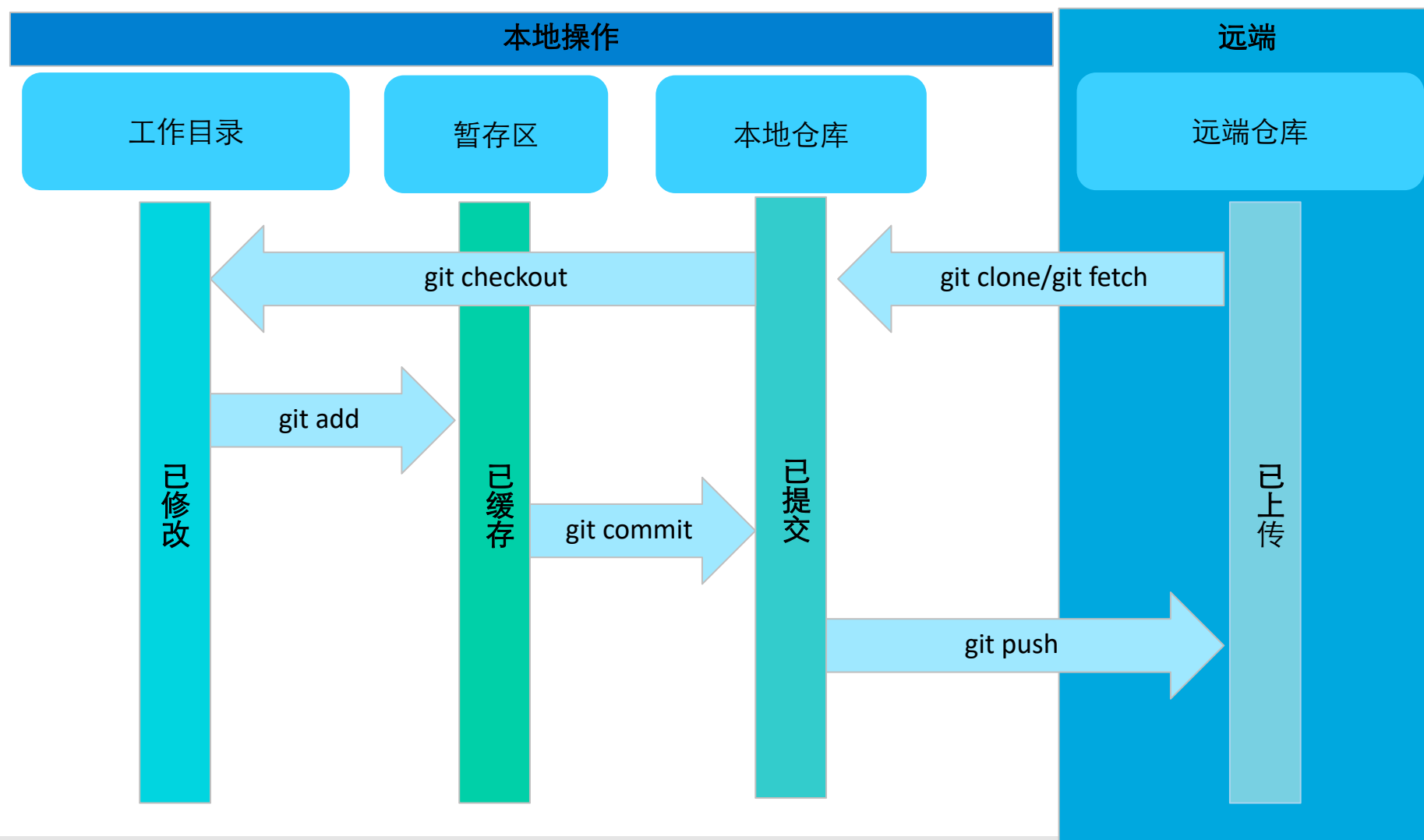
- **工作拷贝（工作目录）**：用于存放产品开发数据本地工作目录。
- **索引（Index）**：用于存放待提交数据的缓存区。
- **本地库**：远端库的一个完整的拷贝，包括所有文件的修改记录，分支等。
- **远端库**：本地库clone来源。
- **中心库**：远端库的一种，公司级存放某个项目所有产品数据的仓库。
- **快照（snapshot）**：版本库某个时间点所有文件集合。
- **全球版本号（commitID）**：Git库的版本号是通过SHA-1算法根据库中的所有内容计算出一个40位的哈希值，这个哈希值是全局唯一的，基本只要前六位就可以唯一标识了。

# Git基本概念 (3/3)

文件状态介绍:

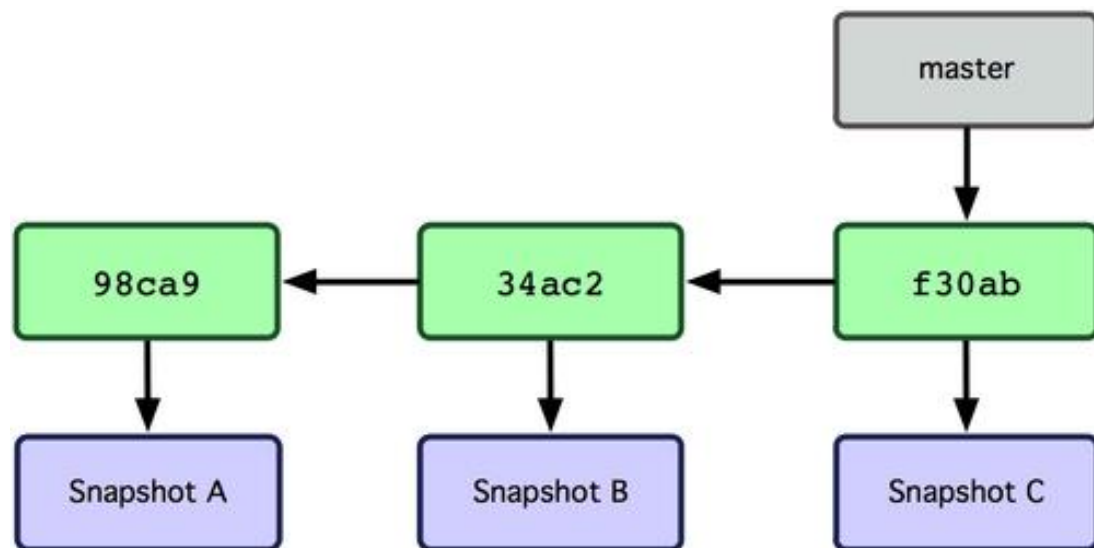
- `Untracked files`: 未被跟踪的文件, 一般指新添加的文件
- `Change not staged for commit`: 已修改的文件, 包括modified和deleted状态
- `Change to be commit`: 已缓冲的文件, 即已add的文件, 包括modified、deleted和new file状态
- `Noting to commit`: 已提交的文件, 即已commit的文件

# Git操作模式



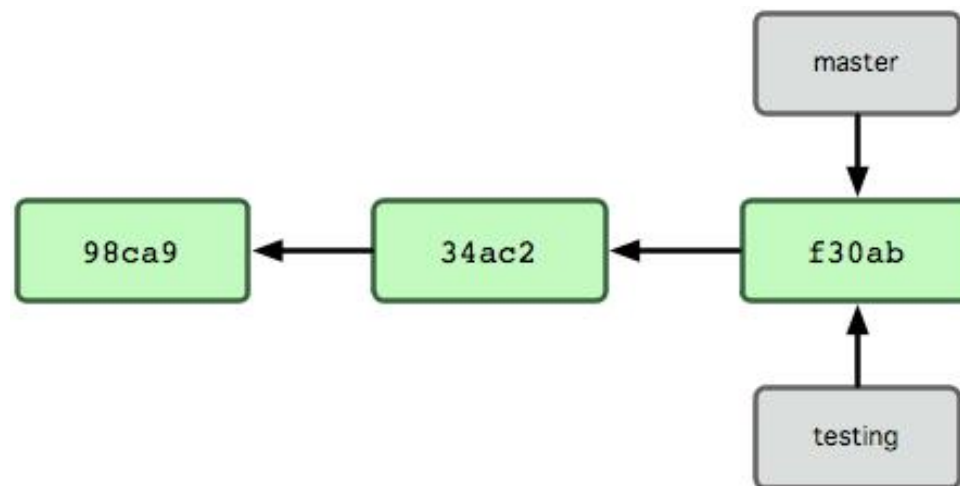
# Git分支(1/17)

- 分支
  - 本质上是个指向commit对象的可变指针
  - 分支的默认名字为master
  - 每次提交的时候都会自动向前移动
  - 作用：从某个提交对象往回看的历史



## Git分支 (2/17)

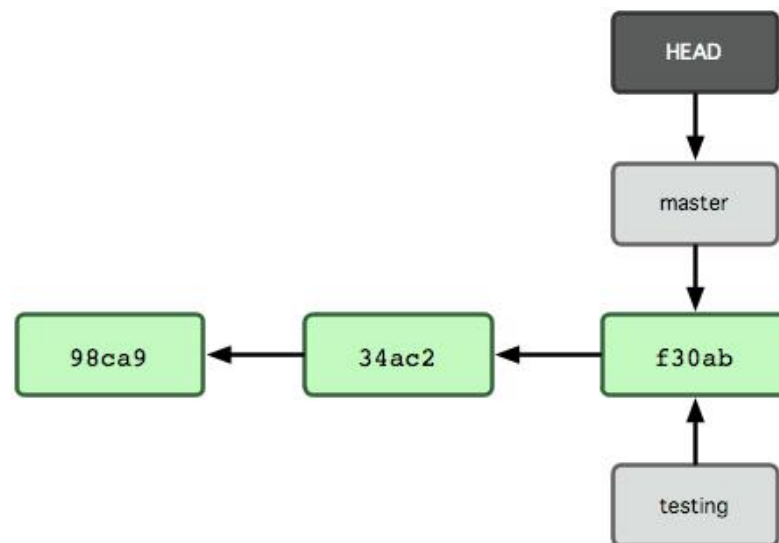
- 新建分支
  - 示例：新建一个testing分支
  - `$ git branch testing`



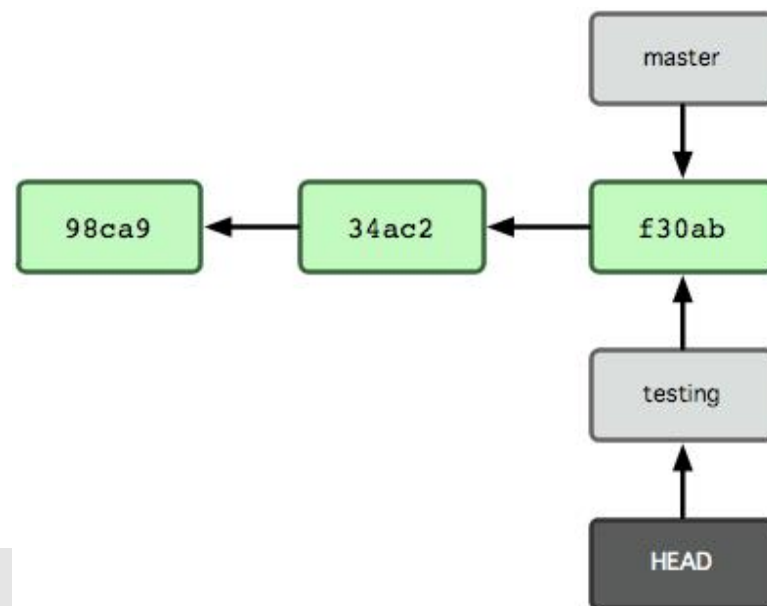
# Git分支 (3/17)

- 如何识别当前分支
  - HEAD指针
  - 示例：切换到当前分支
  - `$ git checkout testing`

切换前

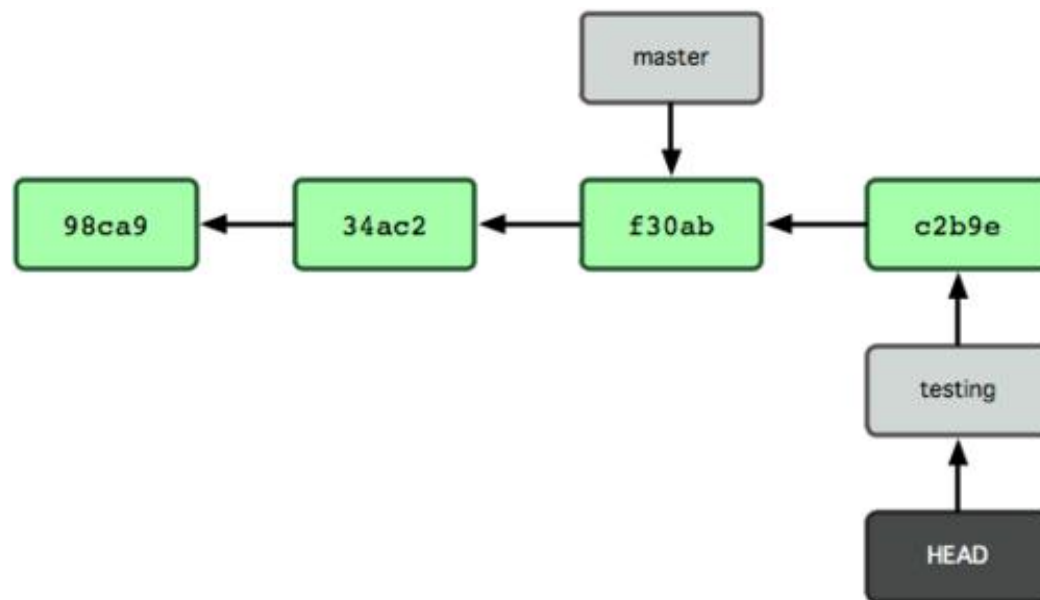


切换后



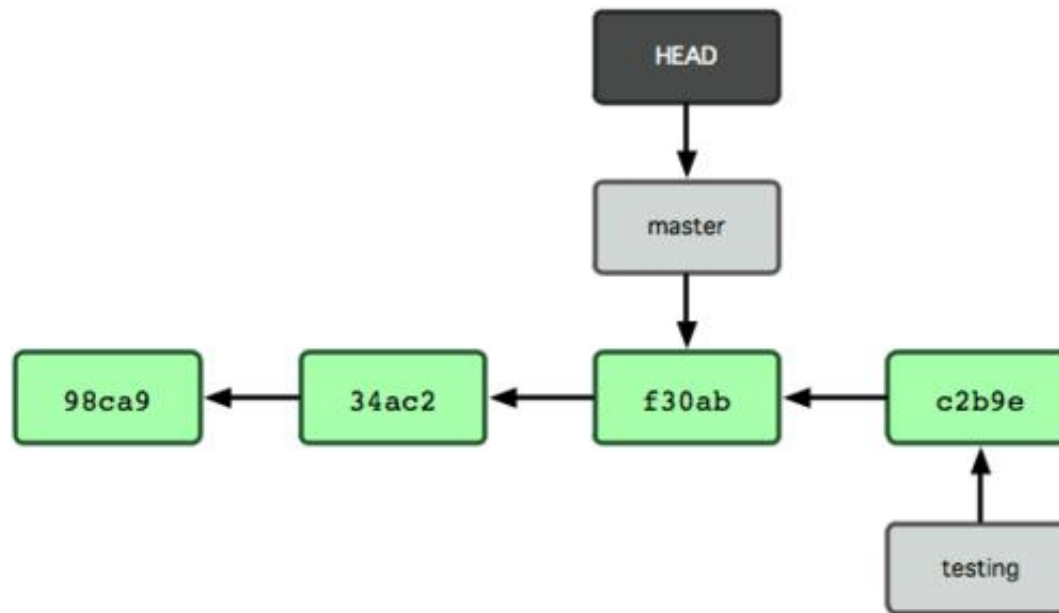
# Git分支(4/17)

- 如何识别当前分支
  - 示例：做一次提交
  - `$ vim test.rb`
  - `$ git commit -a -m 'made a change'`
  - 每次提交后 HEAD 随着分支一起向前移动



# Git分支 (5/17)

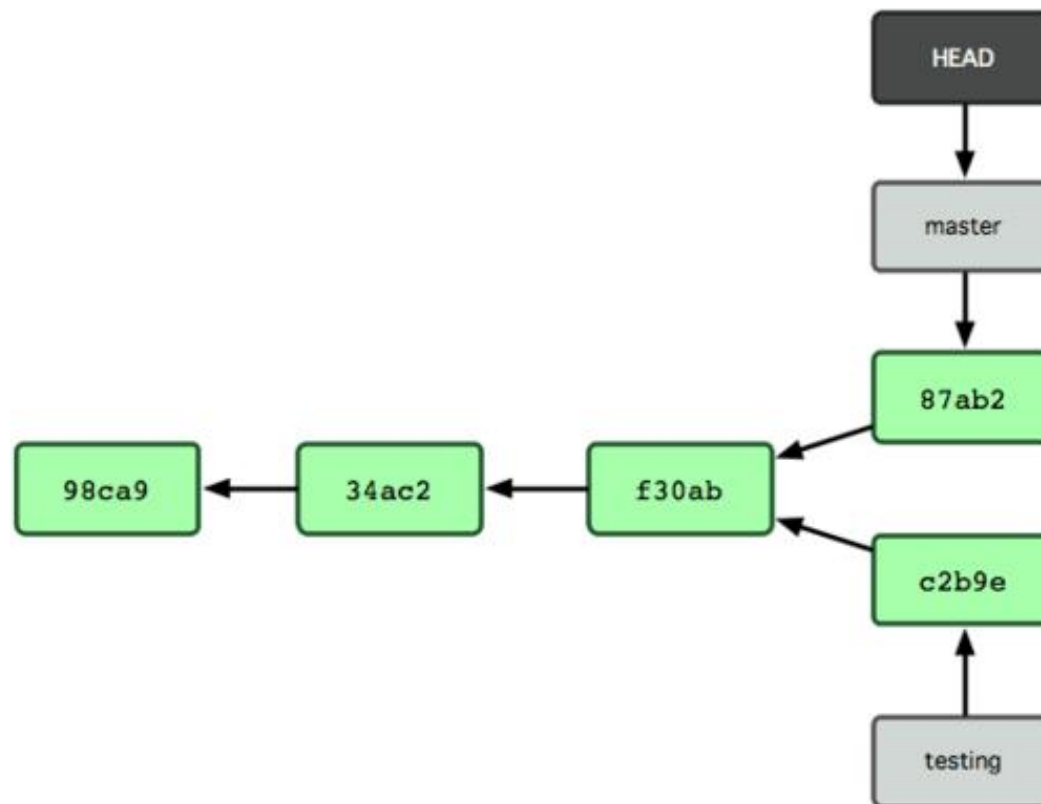
- 如何识别当前分支
  - 示例：切换回master分支
  - `$ git checkout master`





# Git分支 (6/17)

- 如何识别当前分支
  - 示例：在master分支上做一次提交
  - `$ vim test.rb`
  - `$ git commit -a -m 'made other changes'`



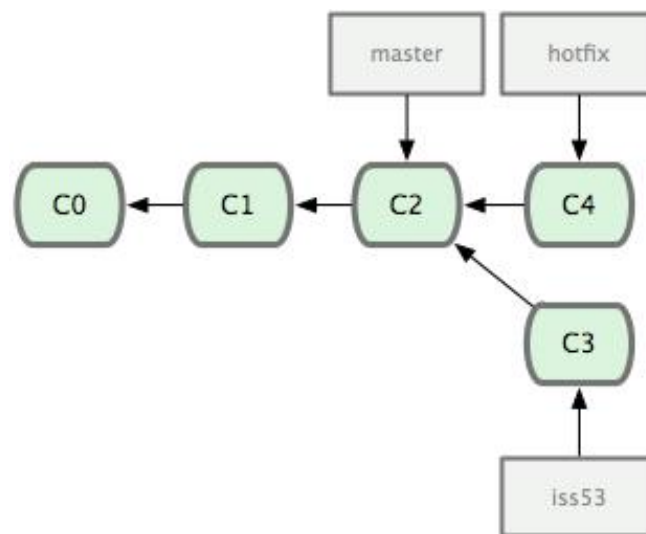
# Git分支(7/17)

- 总结
  - 什么是Git分支：
    - 一个包含所指对象校验和（40 个字符长度 SHA-1 字符串）的文件
    - 新建一个分支就是向一个文件写入 41 个字节（外加一个换行符）
  - Git分支的特点
    - 操作简单
    - 操作速度快
    - 合并分支容易
    - 创建和销毁分支廉价

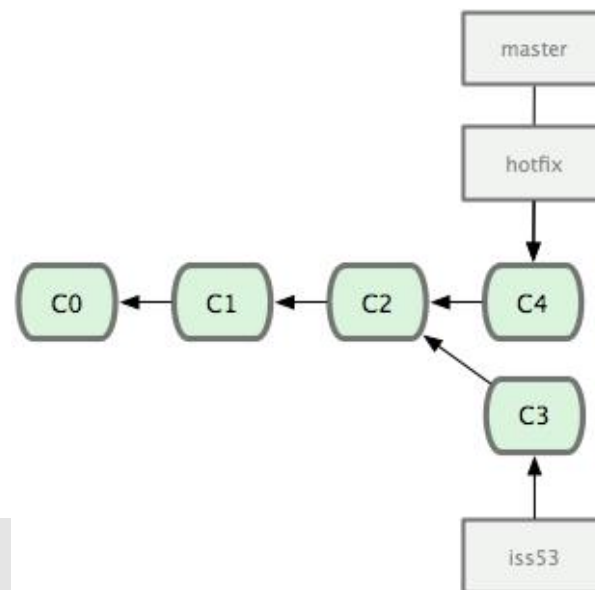
# Git分支(8/17)

- 分支合并
  - 快进式合并(fastforward)
    - 顺着分支走下去可以到达另一个分支的合并过程
    - 示例：右图

合并前



合并后



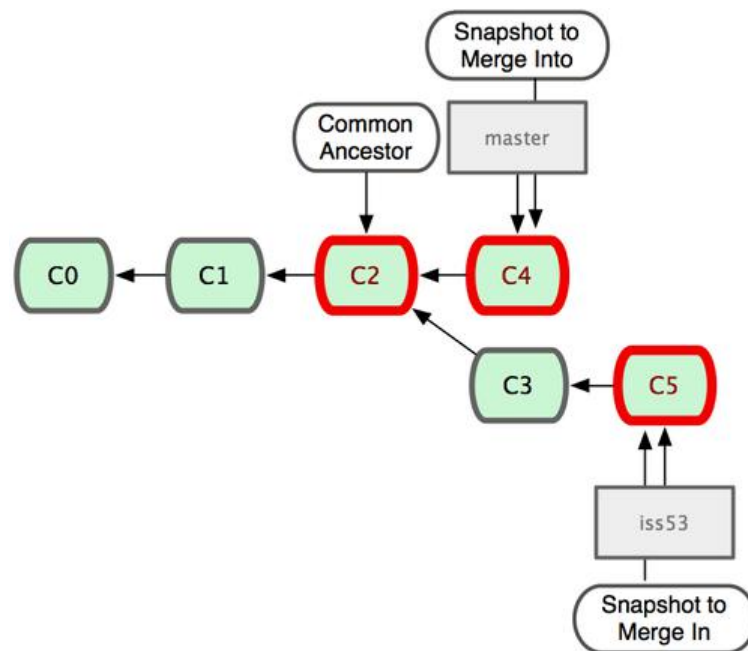
# Git分支 (9/17)

- 分支合并

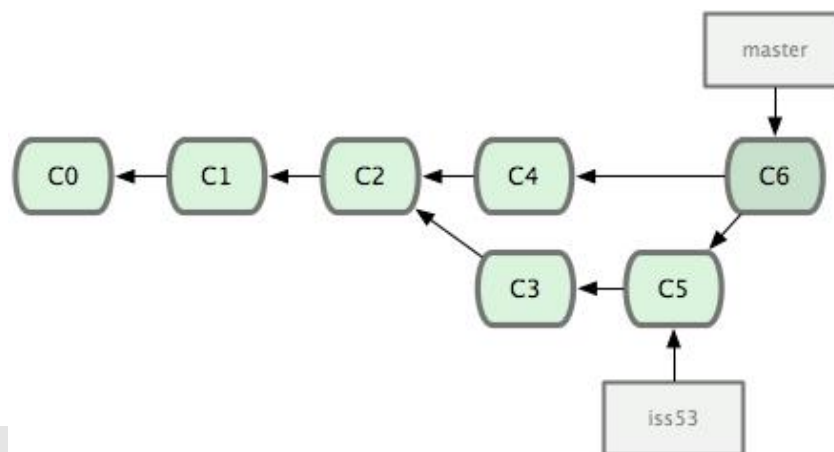
- 非快进式合并 (non-fastforward)

- 使用两个分支的末端以及它们的共同祖先进行一次简单的三方合并计算的合并过程
    - 示例：右图

合并前



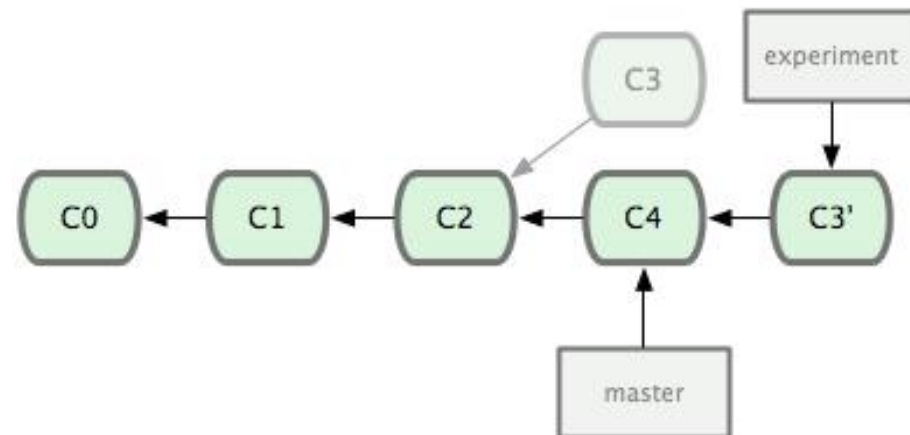
合并后



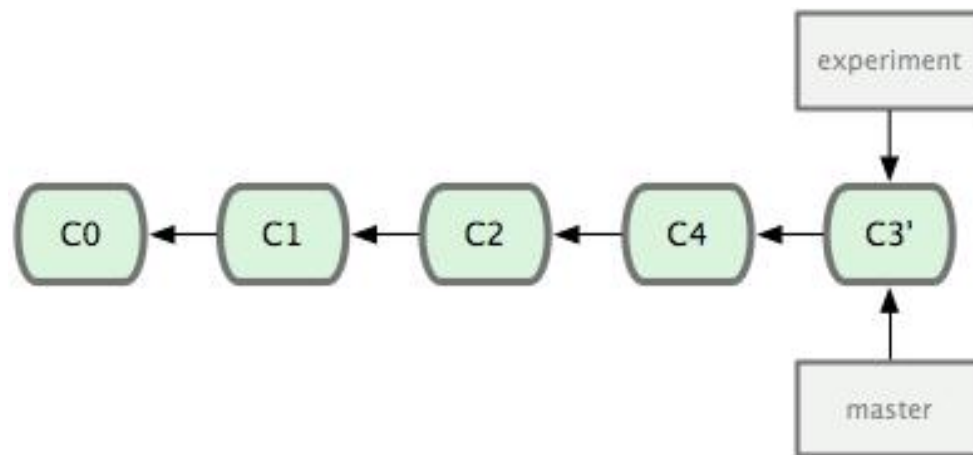
# Git分支(10/17)

- 分支衍合
  - 示例

衍合前



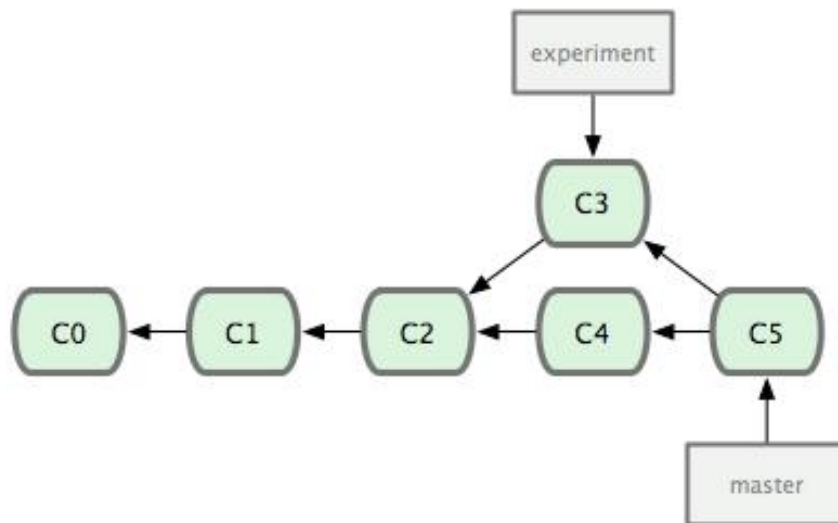
衍合后



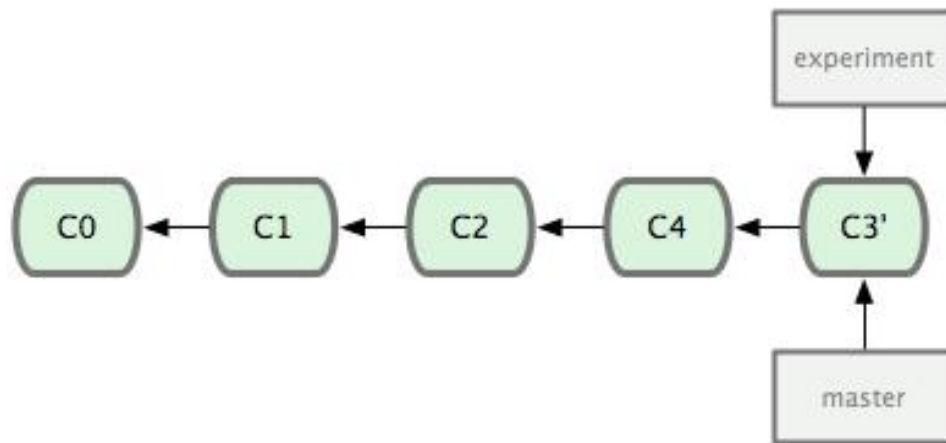
# Git分支(11/17)

- 分支衍合
  - 对比：和三方合并结果一致，但能产生更简洁的提交历史
  - 适用：在主干分支能干净应用的补丁。把解决分支补丁同最新主干代码之间冲突的责任，转化为由提交补丁的人来解决

三方合并



衍合



# Git分支(12/17)

- 合并冲突
  - 类型1: 修改了同一个文件的同一行
    - 解决方法: 确认正确的修改
    - 示例: 命令行解决

```
$ cat doc/README.txt
User1 hacked.
<<<<<< HEAD
Hello, user2.
=====
Hello, user1.
>>>>>> a123390b8936882bd53033a582ab540850b6b5fb
User2 hacked.
User2 hacked again.
```

```
User1 hacked.
Hello, user1 and user2.
User2 hacked.
User2 hacked again.
```

```
$ git add -u
$ git commit -m "Merge completed: say hello to all users."
```

# Git分支(13/17)

- 合并冲突
  - 类型2：文件被重命名为不同的名字（树冲突）
    - 解决办法：确认哪个名字是正确的，删除错误的
    - 示例：命令行解决

```
$ git rm readme.txt
README: needs merge
doc/README.txt: needs merge
readme.txt: needs merge
rm 'readme.txt'
```

```
$ git rm doc/README.txt
README: needs merge
doc/README.txt: needs merge
rm 'doc/README.txt'
```

```
$ git add README
```

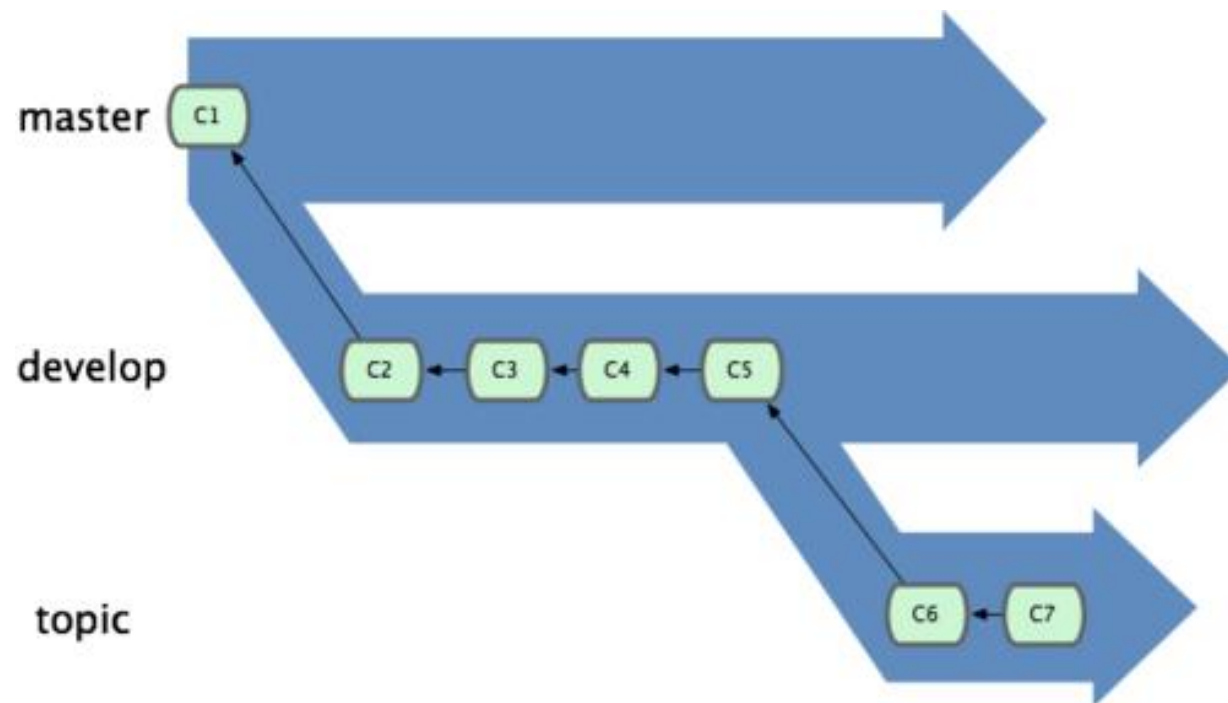


# Git分支(14/17)

- 分支管理
  - 查看所有分支
    - `$ git branch`
  - 查看已合并分支
    - `$ git branch --merged`
  - 查看未合并分支
    - `$ git branch --no-merged`
  - 删除分支
    - `$ git branch -d 分支名`

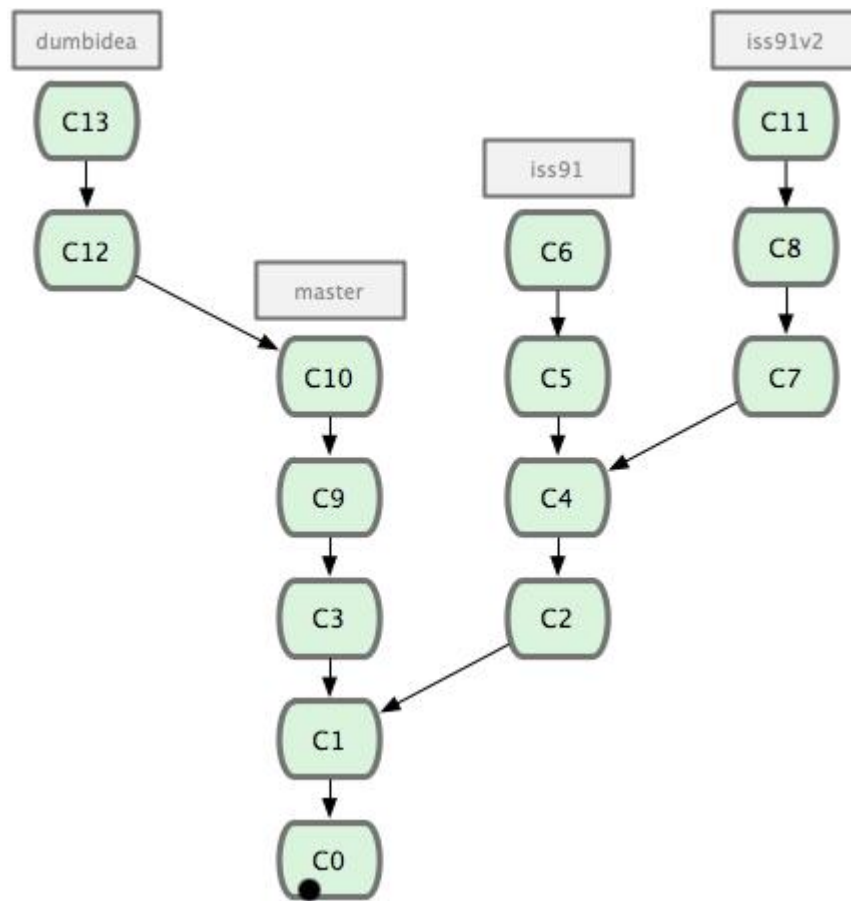
# Git分支 (15/17)

- 分支策略
  - 长期分支
    - 典型示例:
      - master: 稳定分支
      - develop: 测试分支
      - topic: 开发分支



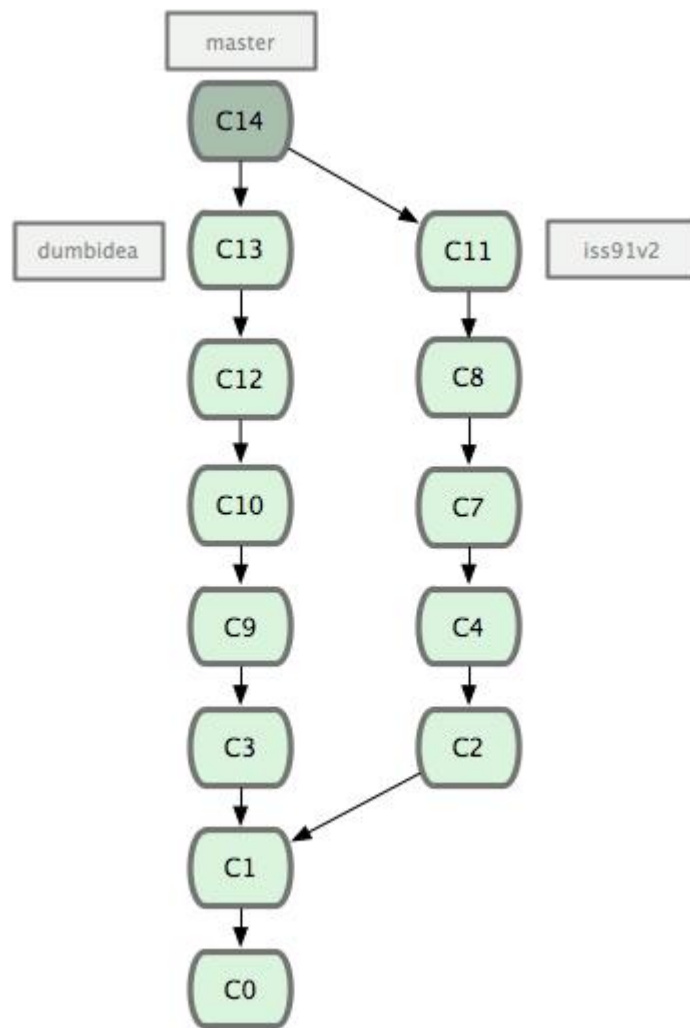
# Git分支(16/17)

- 分支策略
  - 特性分支
    - 适用任何规模的项目
    - 短期的，用来实现单一特性或与其相关工作的分支
    - 示例：见右图



# Git分支(17/17)

- 分支策略
  - 特性分支
    - 示例：见右图



# Git使用协议(1/4)

- HTTP协议
  - 版本控制工具普遍采用的协议，优点：安全（HTTPS）、方便（跨越防火墙）。
- 哑协议
  - Git服务器和客户端的会话过程中只使用了相关协议提供的基本传输功能，而未针对Git的传输特点进行相关优化设计。
  - 优点：配置简单。
  - 缺点：Git客户端和服务端间的通讯缺乏效率，用户使用时最直观的体验之一就是在操作过程没有进度提示，应用会一直停在那里直到整个通讯过程处理完毕。

# Git使用协议(2/4)

- HTTP协议

- 智能协议

- 优点：打破了以前哑传输协议给HTTP协议带来的恶劣印象，让HTTP协议成为Git服务的一个重要选项。
    - 不足：授权管理能力弱。
      - 创建版本库只能在服务器端进行，不能通过远程客户端进行。
      - 配置认证和授权，也只能在服务器端进行，不能在客户端远程配置。
      - 版本库的写操作授权只能进行非零即壹的授权，不能针对分支甚至路径进行授权。

# Git使用协议(3/4)

- SSH协议
  - 唯一一个同时支持读写操作的网络协议
  - 是一个验证授权的网络协议
  - 优点
    - 拥有对网络仓库的写权限
    - 架设相对简单
    - 安全：传输过程加密和授权
    - 高效：压缩数据传输
  - 缺点
    - 无法匿名访问仓库
    - 不利于开源项目

# Git使用协议(4/4)

- Git协议
  - 一个包含在 Git 软件包中的特殊守护进程
  - 监听一个提供类似于SSH服务的特定端口（9418），而无需任何授权
  - 优点
    - 现存最快的传输协议
    - 使用与 SSH 协议相同的数据传输机制，但省去了加密和授权的开销
  - 缺点
    - 缺少授权机制
    - 最难架设的协议：要求有单独的守护进程，需要定制
    - 要求防火墙开放 9418 端口，而企业级防火墙一般不允许对这个非标准端口的访问

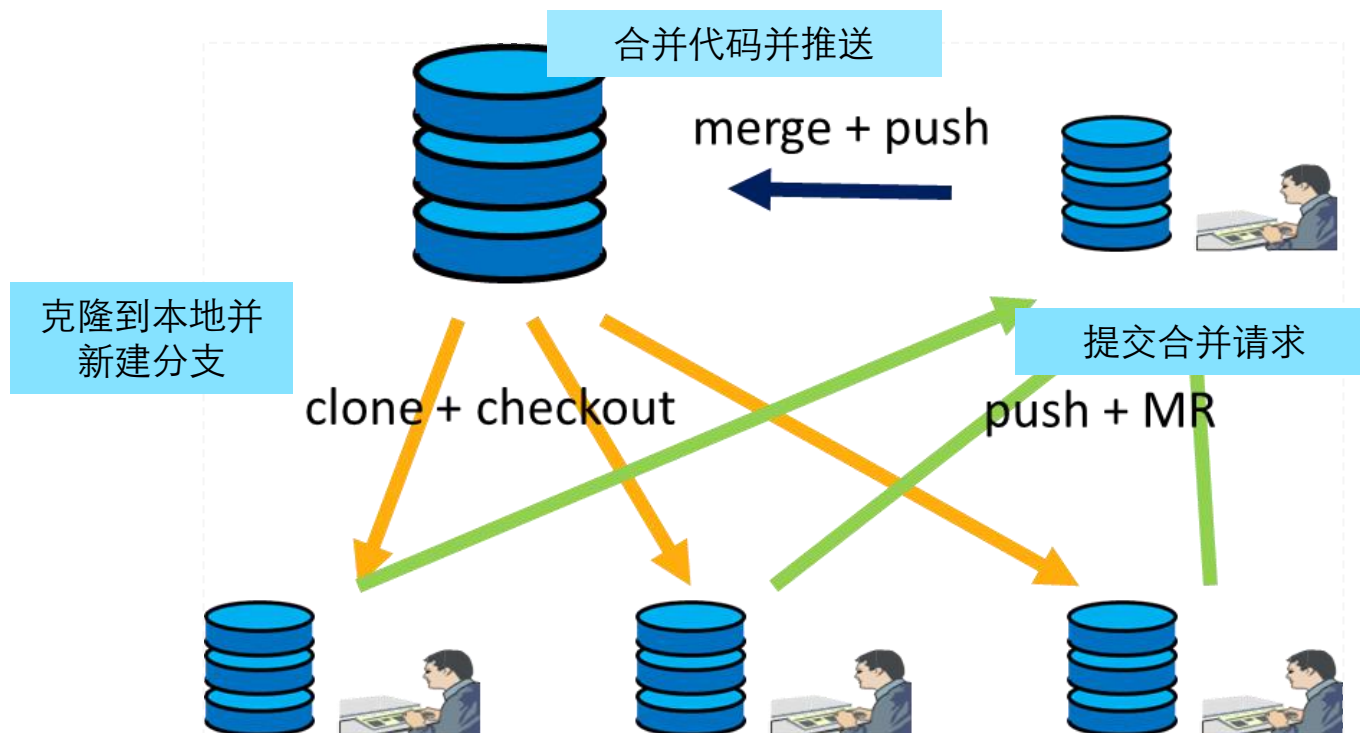


# 配置管理的企业应用

- 配置管理：面向软件开发者提供的基于Git的在线代码托管服务
  - 面向管理员/项目经理，提供仓库管理、权限管理、成员管理、分支保护、安全管控及统计服务。
  - 面向开发者提供代码托管服务、代码仓库、在线客户端等服务。

# 配置管理的企业应用

- 配置管理服务特性一
- 跨地域协同开发
  - 本地离线操作
  - 代码合入评审



# 系统代码检查

# 为什么做代码检查

## 代码交付过程中的常见问题和风险

潜在问题	风险
重复代码过多	造成开发人力的浪费以及后期维护成本增加
编码风格糟糕	代码凌乱、不可读，难于维护与开发修改
圈复杂度过高	造成代码可维护性、可继承性降低，问题定位难度加大
编码安全风险	使用具有安全风险的函数，导致系统的安全性层级降低，加大系统的安全风险

# 代码重复

代码重复 (duplicate code) 在程序设计中表示一段源代码在一个程序，或者一个团体所维护的不同程序中重复出现，是不希望出现的现象。为避免巧合，只有一定数量的代码完全相同才能判定为代码重复。

# 代码重复的原因

产生代码重复可能有以下几个原因：

- 因为某段代码能用就复制粘贴过来，多数情况下代码会有少许不同，如变量名称改变或代码增删。
- 因为要实现与已有功能类似的功能，开发者独立写出与别处相似的代码，研究表明独立撰写的代码在语法上不一定相似。
- 抄袭，即未经允许复制代码，且未列出版权归属。
- 代码自动生成，这时可能需要重复代码以提高性能或方便开发。

# 代码风格

- 代码风格 (Programming style) 即程序开发人员所编写源代码的书写风格，良好代码风格的特点是使代码易读。
- 总结程序设计实践中的经验，代码风格的要素包括（但不限于）以下几点：
  - 名字的使用（驼峰式大小写、标识符命名约定、匈牙利命名法）
  - 表达式与语句
  - 常量的使用
  - 注释的使用
  - 缩进代码的布局

# 圈复杂度

- 圈复杂度 (Cyclomatic complexity) 是一种代码复杂度的衡量标准。在软件测试的概念里，“圈复杂度”用来衡量一个模块判定结构的复杂程度，数量上表现为独立线性路径条数，即合理的预防错误所需测试的最少路径条数，圈复杂度大说明程序代码可能质量低且难于测试和维护，根据经验，程序的可能错误和高的圈复杂度有着很大关系。



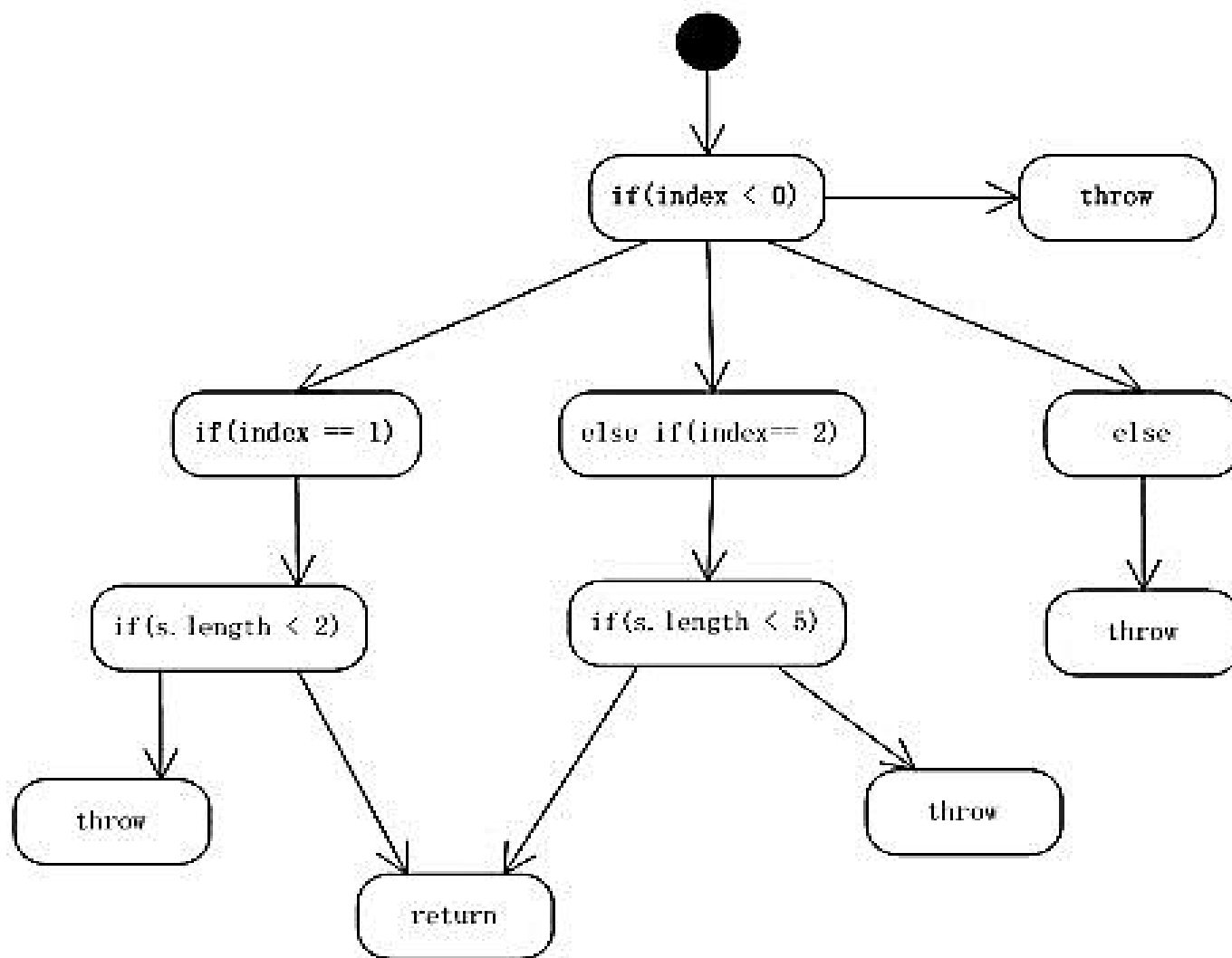
# 圈复杂度的计算

- 通常使用的计算公式是  $V(G) = e - n + 2$  ,  $e$  代表在控制流图中的边的数量（对应代码中顺序结构的部分）,  $n$  代表在控制流图中的节点数量, 包括起点和终点（1、所有终点只计算一次, 即便有多个return或者throw; 2、节点对应代码中的分支语句）

# 圈复杂度的计算实例 (1/3)

```
public String case1(int index, String string) {  
    String returnString = null;  
    if (index < 0) {  
        throw new IndexOutOfBoundsException("exception <0 ");  
    }  
    if (index == 1) {  
        if (string.length() < 2) {  
            return string;  
        }  
        returnString = "returnString1";  
    } else if (index == 2) {  
        if (string.length() < 5) {  
            return string;  
        }  
        returnString = "returnString2";  
    } else {  
        throw new IndexOutOfBoundsException("exception >2 ");  
    }  
    return returnString;  
}
```

## 圈复杂度的计算实例 (2/3)



## 圈复杂度的计算实例 (3/3)

根据公式  $V(G) = e - n + 2 = 12 - 8 + 2 = 6$  。 case1的圈复杂度为6。说明一下为什么  $n = 8$ ，虽然图上的真正节点有12个，但是其中有5个节点为throw、return，这样的节点为end节点，只能记做一个。

# 代码检查的分类

代码检查分为静态分析和动态分析（本课程主要介绍静态分析）

- **静态分析 (Static Program Analysis)**：在不运行计算机程序的条件下，进行程序分析的方法。大部分的静态程序分析的对象是针对特定版本的源代码，也有些静态程序分析的对象是目标代码。
- **动态分析 (Dynamic Program Analysis)**：在运行计算机程序的条件下，进行程序分析的方法。

# 静态分析常用技术 (1/4)

- 词法分析
- 语法分析
- 抽象语法树分析
- 语义分析
- 控制流分析
- 数据流分析
- 污点分析
- 无效代码分析

## 静态分析常用技术 (2/4)

- 词法分析：从左至右一个字符一个字符的读入源程序，对构成源程序的字符流进行扫描，通过使用正则表达式匹配方法将源代码转换为等价的符号(Token) 流，生成相关符号列表。
- 语法分析：判断源程序结构上是否正确，通过使用上下文无关语法将相关符号整理为语法树。

## 静态分析常用技术 (3/4)

- 抽象语法树分析：将程序组织成树形结构，树中相关节点代表了程序中的相关代码。
- 语义分析：对结构上正确的源程序进行上下文有关性质的审查。
- 控制流分析：生成有向控制流图，用节点表示基本代码块，节点间的有向边代表控制流路径，反向边表示可能存在的循环；还可生成函数调用关系图，表示函数间的嵌套关系。



## 静态分析常用技术 (4/4)

- 数据流分析：对控制流图进行遍历，记录变量的初始化点和引用点，保存切片相关数据信息。
- 污点分析：基于数据流图判断源代码中哪些变量可能受到攻击，是验证程序输入、识别代码表达缺陷的关键。
- 无效代码分析：根据控制流图可分析孤立的节点部分为无效代码。

# 代码检查的主流工具 (1/6)

## Meta-Compilation (Coverity)

由Stanford大学的Dawson Engler副教授等研究开发，该静态分析工具允许用户使用一种称作metal的状态机语言编写自定义的时序规则，从而实现了静态分析工具的可扩展性。MC的实际效果非常优秀，号称在Linux内核中找出来数百个安全漏洞。MC目前已经商业化，属于Coverity Inc. 2014年被Synopsys收购。目前学术领域比较认可的静态分析工具，其技术处于领先地位。

## 代码检查的主流工具 (2/6)

### Klocwork K8

Klocwork K8是由 Klocwork 公司开发的，支持 C/C++，JAVA，它能检测缓冲区溢出、内存泄露、安全漏洞等软件缺陷。

# 代码检查的主流工具 (3/6)

## Splint

Splint是开源的静态软件缺陷检测工具，用于检测用标准C实现的软件缺陷。通过在源代码中添加关于函数、参数和类型的格式化注释，实现了未使用的声明、类型不一致、使用未定义变量、内存泄露、不可达代码、函数返回值、无限循环、危险的别名等软件缺陷。

# 代码检查的主流工具 (4/6)

## Findbugs

Findbugs是一个基于 java 语言的静态分析工具，它主要检查类或者 JAR 文件。其主要思想是利用字节码与软件缺陷模式对比来寻找矛盾点，从而发现程序中的软件缺陷。它提供了自定义检测器功能，利用 Byte Code Engineering Library 实现基于 visitor 模式字节码的扫描，并且通过加载 xml 文件的方式进行管理。

# 代码检查的主流工具 (5/6)

## PMD

PMD是一款采用 BSD 协议发布的 Java 程序代码检查工具，其核心是 javacc解析器。该工具可以做到检查 Java 代码中是否含有未使用的变量、是否含有空的抓取块、是否含有不必要的对象、复杂表达式、复杂代码等软件缺陷。而且其易于扩展，利用 xpath 或者 java 语言编写新的缺陷检测器。

# 代码检查的主流工具 (6/6)

## Cppcheck

Cppcheck 是一种开源的 C/C++代码缺陷静态检查工具。不同于 C/C++编译器及其它分析工具，Cppcheck 只检查编译器检查不出来的 bug，不检查语法错误。它能检测出自动变量检查、数组边界检查、class 类检查、过期的函数，废弃函数检查、异常内存使用，释放检查、内存泄露检查、操作系统资源释放检查、异常STL 函数使用检查、代码格式错误以及性能因素检查等软件缺陷。

## 代码实例（Java）－ 编码风格（1/2）

```
OutputStream stream = null;
try{
    stream = new FileOutputStream("myfile.txt");
    for (String property : propertyList) {
        // ...
    }
} catch (Exception e) {
    // ...
}
```



## 代码实例（Java）－ 编码风格（2/2）

保证FileOutputStream在任何情况下都会被关闭，避免资源泄漏甚至更严重的后果。

```
OutputStream stream = null;
try{
    stream = new FileOutputStream("myfile.txt");
    for (String property : propertyList) {
        // ...
    }
} catch (Exception e) {
    // ...
} finally {
    stream.close();
}
```

## 代码实例（Java）－ 编码问题（1/2）

```
public static String decodeString(String method, String content) {  
    if(StrUtils.isEmpty(content)) {  
        return "";  
    }  
    try {  
        content = java.net.URLDecoder.decode(content, "UTF-8");  
    } catch (UnsupportedEncodingException e) {  
        LOGGER.error("decodeString ERROR");  
    }  
    return content;  
}
```

## 代码实例（Java）－ 编码问题（2/2）

避免给函数的参数重新赋值。

```
public static String decodeString(String method, String content) {  
    if(StrUtils.isEmpty(content)) {  
        return "";  
    }  
    try {  
        String decodedContent = java.net.URLDecoder.decode(content, "UTF-8");  
    } catch (UnsupportedEncodingException e) {  
        LOGGER.error("decodeString ERROR");  
    }  
    return decodedContent ;  
}
```

## 代码实例（Java）－ 安全编码（1/2）

```
EntityManager pm = getEM();

TypedQuery q = em.createQuery(
    String.format("select * from Users where name = %s", username),
    UserEntity.class);

UserEntity res = q.getSingleResult();
```

## 代码实例（Java） – 安全编码（2/2）

SQL查询中的输入值必须是安全的， 绑定变量的预编译块可以很容易地减轻SQL注入攻击的风险。

```
TypedQuery q = em.createQuery(  
    "select * from Users where name = usernameParam", UserEntity.class)  
    .setParameter("usernameParam", username);  
  
UserEntity res = q.getSingleResult();
```

# 代码检查的企业实践

- 代码质量在华为受到高度重视。
- 代码检查是华为软件生命周期管理的重要一环。
- 代码检查是华为代码入库的重要依据，代码检查不通过不能入库。
- 代码检查是华为个人及和版本级构建的组成部分，会被自动触发执行。
- 华为在几十年的软件开发历史中积累了大量的代码检查规则。

## 华为公有云DevCloud代码检查服务 (1/7)

- 在线进行多种语言的代码静态检查
- 代码风格检查
- 代码安全检查
- 代码重复检查
- 代码圈复杂度报告
- 代码缺陷改进建议
- 检查规则集配置

# 华为公有云DevCloud代码检查服务 (2/7)

- 只需一次配置任务，重复自动执行

请输入关键字，按Enter进行搜索

代码检查规则库为您提供 **949** 条规则

新建任务

### CSSCodeCheck

最近检查时间：2017-04-18 11:17:42

0.0	0	286
风险指数	未解决问题数	代码行

HuaweiCon... [开始检查](#)

### HTML5CodeCheck

最近检查时间：2017-04-18 10:48:10

3.0	3	40
风险指数	未解决问题数	代码行

HuaweiCon... [开始检查](#)

### JavaCodeCheck

最近检查时间：2017-04-18 10:48:12

34.9	35	109
风险指数	未解决问题数	代码行

HuaweiCon... [开始检查](#)

### JSCodeCheck

最近检查时间：2017-04-18 10:48:12

68.4	84	401
风险指数	未解决问题数	代码行

HuaweiCon... [开始检查](#)



# 华为公有云DevCloud代码检查服务 (3/7)

- 全面分析报告



# 华为公有云DevCloud代码检查服务 (4/7)

- 检查结果展现

急需处理的问题

所有问题

严重

问题描述

Use a logger to log this exc...

检查规则

不要调用Throwable.printSta...

src/main/java/com/huawei/devcloud...

未解决

严重

问题描述

In J2EE, getClassLoader() mi...

检查规则

使用合适的类加载器

src/main/java/com/huawei/devcloud...

未解决

严重

问题描述

Either log or rethrow this ex...

检查规则

异常的处理应该保存原始的异...

src/main/java/com/huawei/devcloud...

未解决

严重

问题描述

Either log or rethrow this ex...

检查规则

异常的处理应该保存原始的异...

src/main/java/com/huawei/devcloud...

未解决

一般

问题描述

System.out.println is used

检查规则

System Println

src/main/java/com/huawei/devcloud...

未解决

一般

问题描述

System.out.println is used

检查规则

System Println

src/main/java/com/huawei/devcloud...

未解决

一般

问题描述

System.out.println is used

检查规则

System Println

src/main/java/com/huawei/devcloud...

未解决

一般

问题描述

System.out.println is used

检查规则

System Println

src/main/java/com/huawei/devcloud...

未解决

Java

Java

Java

Java

Java

Java

Java

Java

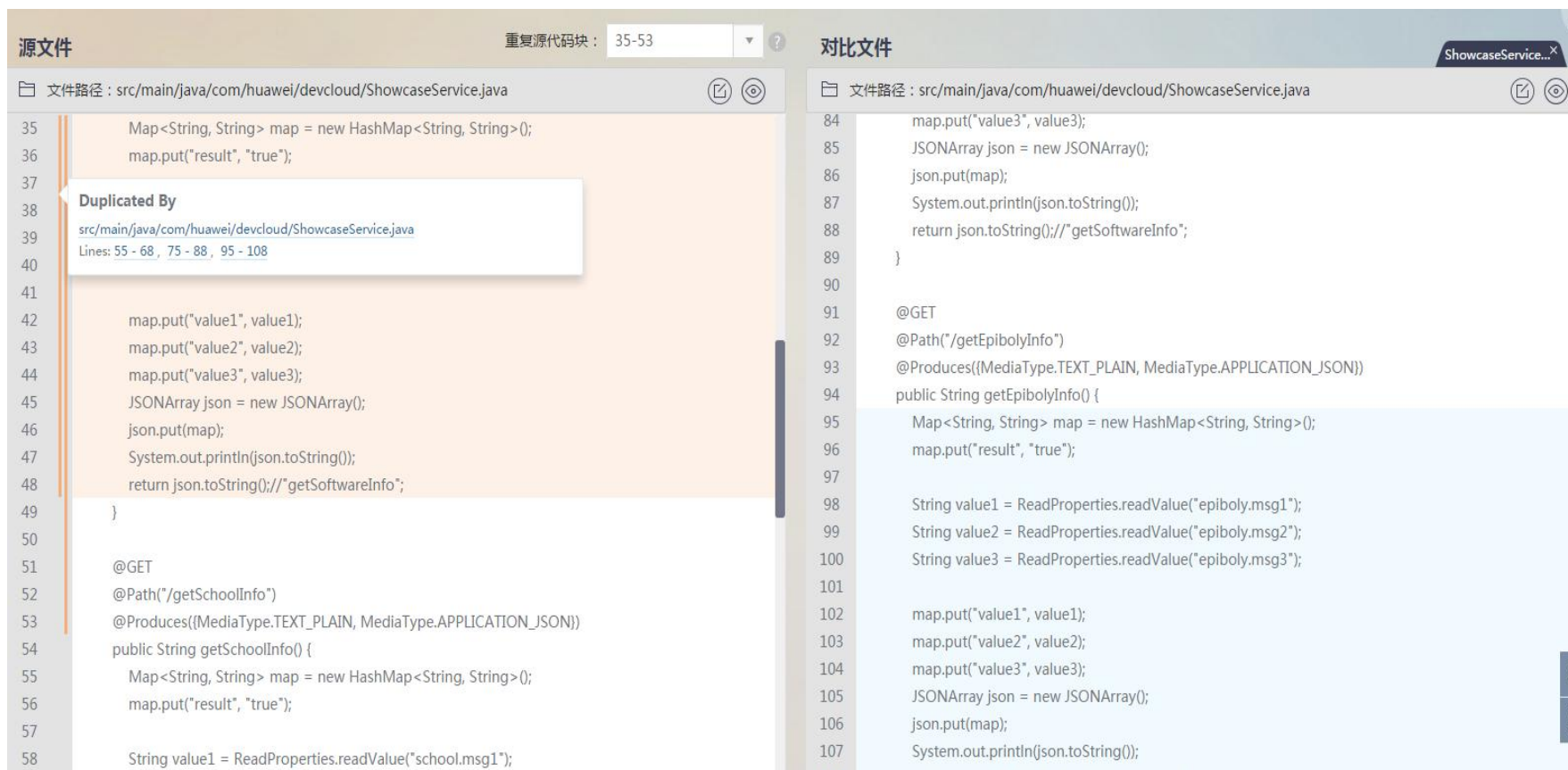
# 华为公有云DevCloud代码检查服务 (5/7)

- 圈复杂度报告



# 华为公有云DevCloud代码检查服务 (6/7)

- 重复代码检测



# 华为公有云DevCloud代码检查服务 (7/7)

- 近1000条规则可供配置检查规则集



# 系统编译构建

# 什么是编译构建

- 软件编译构建是指把软件源码编译成目标文件，并将目标文件和必要的文档制作成软件包的过程。以Maven工程构建为例，包含：从代码仓库拉取源码、从软件仓库拉取依赖包、编译成目标文件、软件打包、上传软件包等步骤。

## 环境搭建耗时费力，且易因环境差异引入问题

- 研发人员耗费大量精力到环境的搭建配置及调测，不能聚焦业务开发。不仅造成开发人员精力的浪费，也由于时间消耗带来等待成本。同时，很难保证本地构建环境的完全同步，引发问题。



## 本地硬件配置不高，编译构建速度慢

- 众所周知，编译构建硬件资源消耗大，但中小型企业 and 创业者受资金投入限制，硬件配置普遍不高，造成编译构建速度慢，影响开发效率。

# 突发项目资源消耗大，结束后闲置

- 企业经常有突发性项目，对编译构建资源的需求突发性增加，让企业陷入两难境地，既不想影响突发项目进度，又不愿投入大量资金购买后期闲置设备。

# 多语言不能并行构建

- 随着互联网的迅猛发展，多语言混合编程成为常态，本地构建环境受硬件限制，难于在同一环境上支持多种构建语言和应用类型并发构建。

# 云端构建优势

- 云端编译构建，在云上集中编译构建资源，通过统一平台和调度，为软件企业或者个人按需分配资源，提供软件编译构建服务。其具有本地构建无可比拟的优势，必将成为软件编译构建的主流方式。

# 把开发人员从繁琐的环境搭建解脱出来

- ▣ 开发人员不再关注构建环境软硬件、安装配置及维护，代码提交后，指定需要的构建类型，系统即可自动分配相应资源完成构建出包。开发人员可完全聚焦于业务实现，降低开发成本。

# 构建成本低

- 软件企业不再需要有自己的独立构建环境，只需按照实际占用的构建资源及时长，支付相应费用。避免企业一次性的大量资金投入，对个人创业者更是具有极大的吸引力。

# 构建环境统一

- 所有构建都基于相同的环境和配置，企业和用户再也不用担心因为环境差异带来的头痛问题。而且云端构建环境，由专业人士统一维护和升级，可快速跟踪和更新为业界主流构建标准和高效工具，快速提升构建质量和效率。

# 资源共享和按需分配

- 云端构建可在构建资源上预装多种构建环境，收到构建请求时，再根据需要动态分配环境和构建任务，任务完成后即刻释放资源，真正做到按需分配和资源共享。



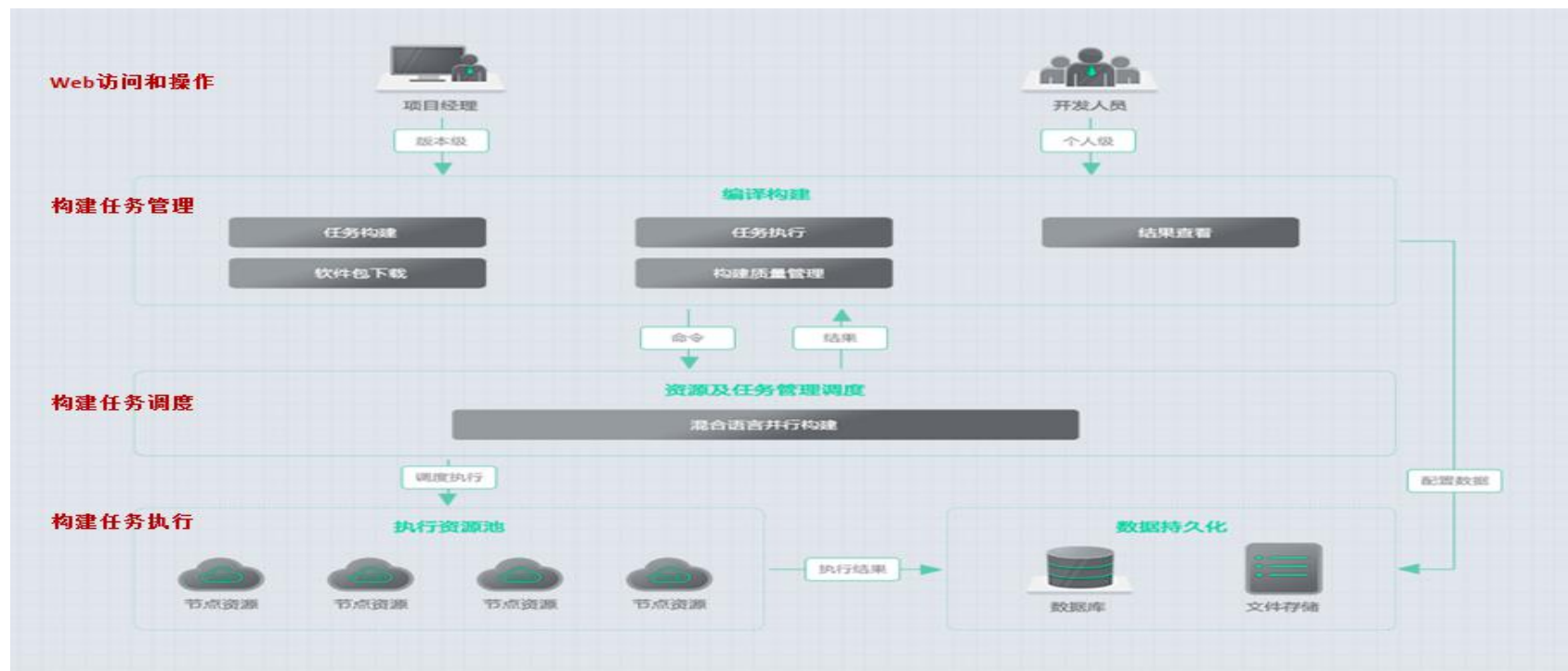
# 混合语言并行构建

- 云端构建具有海量的构建资源，通过统一的调度算法，可指定构建任务到不同的构建资源上同时执行，很好地满足互联网开发混合语言并行构建的需要。

# 构建速度快

- 云端构建不仅可大幅提高硬件配置来提升构建速度，还可利用海量的构建资源，采用分布式构建等技术手段，实现本地构建无法企及的构建速度，大幅提升构建效率，减少人员等待时间。

# 云端构建体系结构 (1/2)



云端构建系统一般分为四层：构建任务执行层；构建任务调度层；构建任务管理层；Web访问和操作层

## 云端构建体系结构（2/2）

- 构建任务执行层负责构建任务的真正执行。
- 构建任务调度层负责构建任务和资源的分配和调度。
- 构建任务管理层负责构建任务的常规管理及构建结果的收集展示。
- Web访问和操作层负责为web访问和操作提供支持 。

# 任务和资源高效调度

- 任务和资源高效调度，是云端编译构建需要具备的基础能力，不仅需要准确匹配构建任务要求和资源，同时其效率和任务的并发度，也是影响构建效率的关键因素。在设计 and 提供云端编译构建服务时，需要重点考虑。

# 构建环境安全

- 云端构建的最大好处就是多用户对构建环境的重用，但在构建时，必然要将用户源码下载到构建环境。对不同企业或者用户来说，就存在安全隐患，如何保证源码（包括编译临时文件）在不同用户和企业间隔离，也是云端构建需要解决的重点问题。

# 构建类型动态扩展

- 由于互联网业务种类繁多，需要支持的构建语言和类型也日新月异，所以云端构建需从架构上具备构建类型动态扩展能力，比如通过插件快速加入新构建类型的的能力。

# 资源弹性伸缩

- 云上承载的构建任务量总是在动态变化，为了避免长期占用宝贵的构建资源，云端构建需要具有资源弹性伸缩能力。构建需求量大时，能够自动申请基础资源并快速可靠创建构建环境；构建需求量减小时，自动释放构建资源给基础云，实现资源的最大化重用。



# 分布式构建

- 为了最大程度地利用云端构建的海量资源来提升构建速度和构建并发能力，需要让构建任务在不同的构建资源中并行执行。如何最大程度地利用资源进行分布式构建，是云端构建的关键技术和核心能力。

# 本地镜像仓库

- 为了提高构建速度，云端构建一般会提供本地镜像仓库，避免构建时再通过网络从中心仓库拉取依赖包。当然，为了保持仓库间软件包的同步，需要镜像仓库与远程中心仓库自动定期同步。

# 构建环境容器化


- 云端构建最好能将构建环境容器化，减少环境依赖，以便快速复制构建环境，并在同一节点执行多个构建任务，提升系统并行构建能力。


# 快速创建构建任务


\* 任务名称：


\* 归属项目： ▼  
※只有项目的创建者、项目经理，开发人员才能在项目下创建构建任务


\* 构建环境：


  
Java ✓













  
python

  
EMBEDDED

  
Other

\* 构建类型：

  
Maven ✓

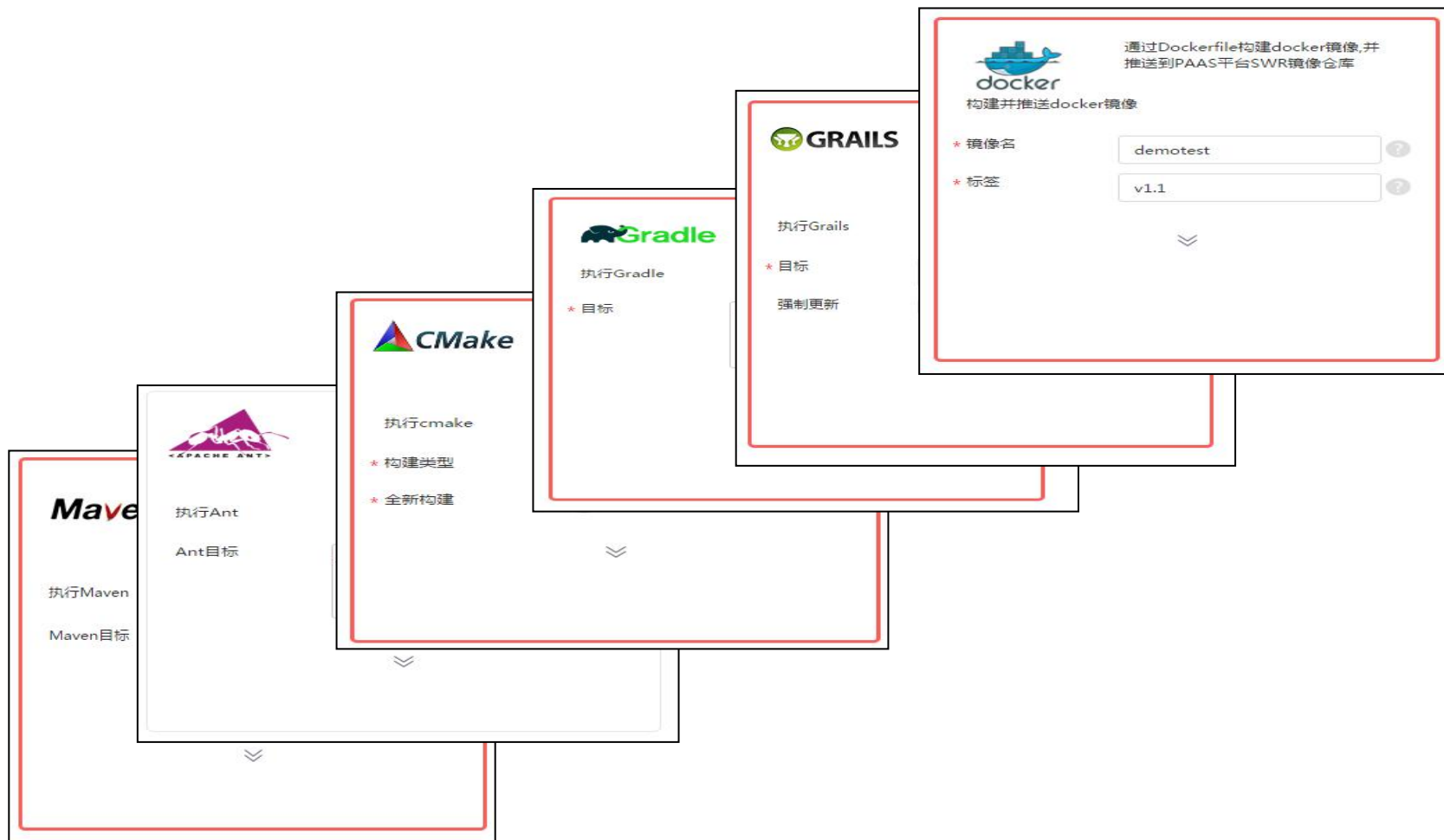
  
APACHE ANT

代码仓库： ▼

分支：

归档：  
※ 归档范例 target/\*.jar，用于归档所有jar文件

# 支持主流构建标准



# 支持多种定时构建方式

## 提交代码自动构建

B3\_JenkinsPipelineAdapter

基本信息 代码配置 构建配置 构建执行配置 构建计划配置

请从Codehub仓库选择分支构建您的任务吧：

您已选择：JenkinsPi... [ \${bra... } ] × standardfil... [ master ] × 清除所有

Jenkins

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

jenkins-2322

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

JenkinsAdapter

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

JenkinsAdapter-API

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

JenkinsAdapterTests

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

jenkinshome

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

JenkinsPipelineAdapter

git@codehub-rnd-devcloud.huawei.com...

分支：\$(branch)

目录：仅支持数字、字母及下划线。

jenkinsSlaveTemplate

git@codehub-rnd-devcloud.huawei.com...

分支：请选择仓库分支

目录：仅支持数字、字母及下划线。

总数数: 8 < 1 >

## 定时构建

B3\_JenkinsPipelineAdapter

基本信息 代码配置 构建配置 构建执行配置 构建计划配置

计划周期：☐ 不定时 ☐ 每日 ☒ 每周

☒ 周一 ☒ 周二 ☒ 周三 ☒ 周四 ☒ 周五 ☒ 周六 ☒ 周日

(HH:MM)

构建时长限制： (小时)

上一步 保存 取消

# 系统测试管理

# 软件测试的目的：不同组织会有不同的测试目的

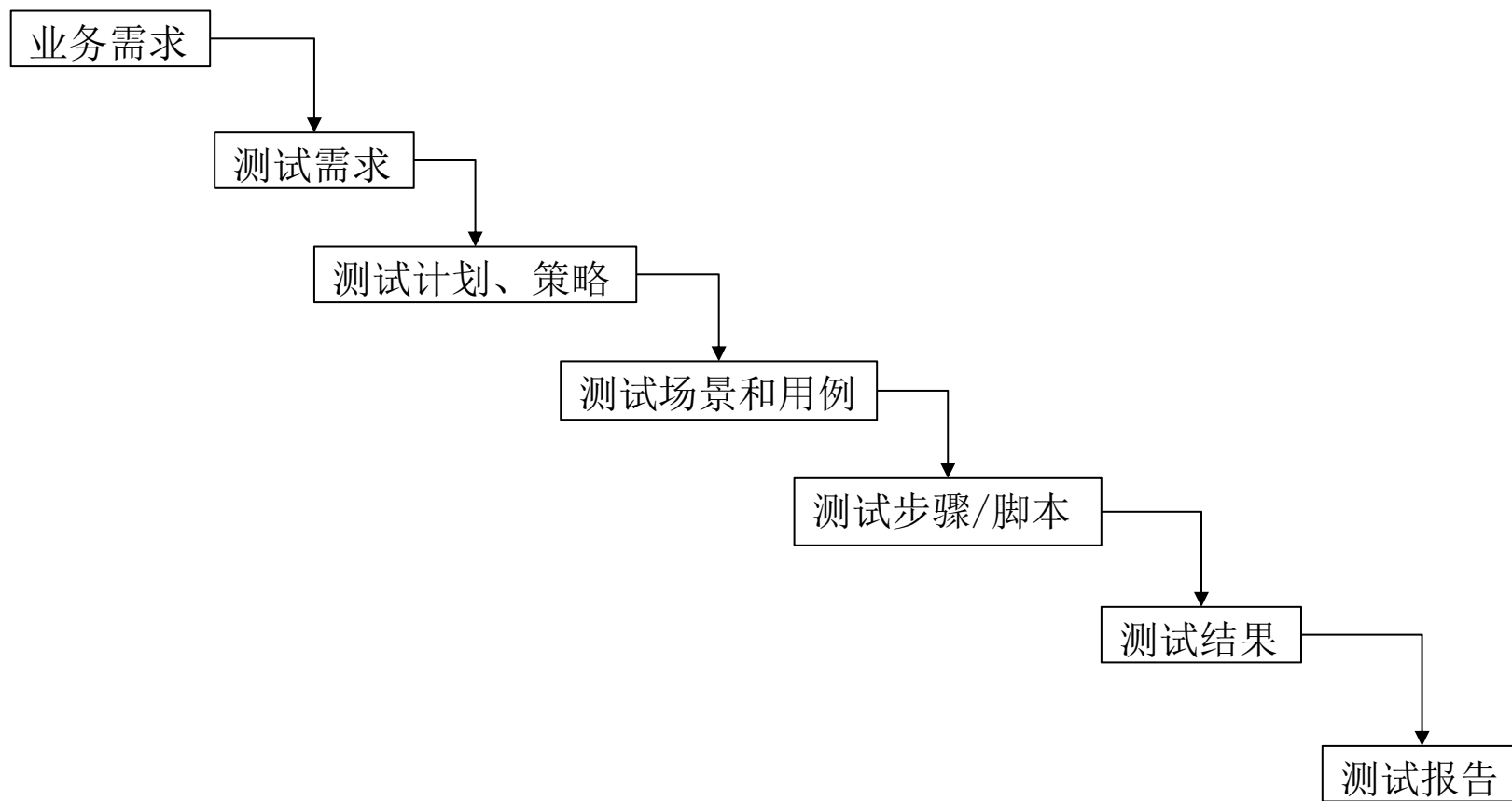
- 确保软件产品符合用户需求
- 发现软件产品中的质量缺陷和风险
- 保证软件产品的高质量
- 帮助经理做发布/不发布决策
- 给出有一定可信度的质量评价
- 找出安全可靠的产品使用方法
- 最小化技术支持投入
- 合规
- 最小化诉讼风险



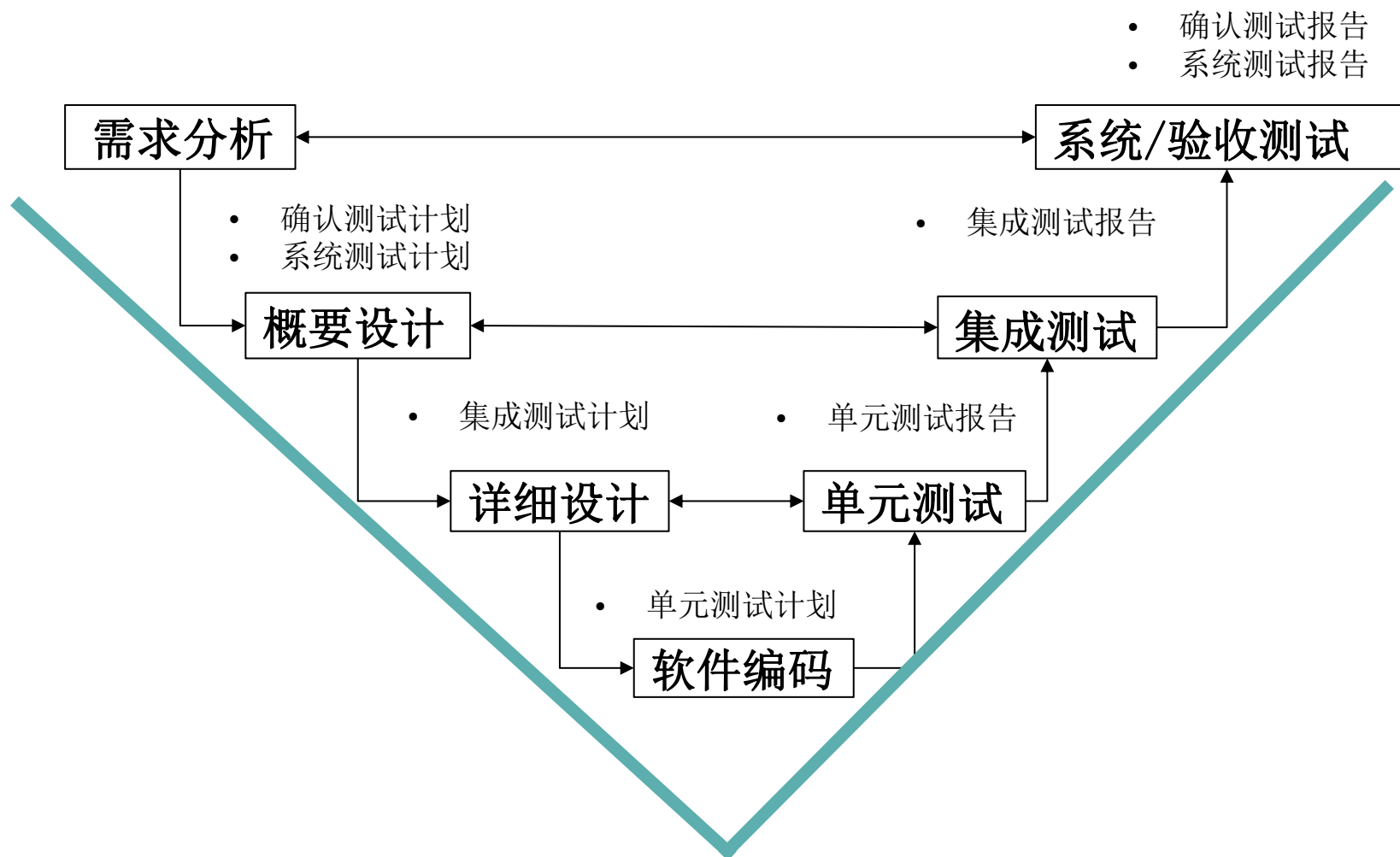
# 软件测试的对象

- 软件源程序
- 需求规格说明
- 概要设计说明
- 详细设计规格说明

# 软件测试过程



# 软件测试V模型



# 软件测试的原则

- 尽早地、不断地进行软件测试，测试和开发过程相结合
- 所有的测试应追溯到用户需求
- 权衡测试投入和产出比
- 测试规模从小到大覆盖，从单元测试到系统测试
- 明确测试输入预制条件和对应的预期输出结果
- 一般避免测试自己编写的程序
- 测试设计时充分考虑异常的输入情况
- 二八原则：分析、设计、实现阶段的复审和测试能发现80%缺陷，系统测试找出其余缺陷中的80%。  
80%问题在20%程序模块。
- 既应该测试软件该做什么，也应该测试软件不该做什么

# 软件测试方法和类型

测试方法	静态测试	动态测试	白盒测试	黑盒测试	灰盒测试
测试级别	单元测试	集成测试	模块接口测试	系统测试	验收测试
测试类型	安装测试	兼容性测试	冒烟测试	回归测试	验收测试
	Alpha测试	Beta测试	功能测试	非功能测试	持续测试
	性能测试	易用性测试	可达性测试	安全测试	国际化
	A/B测试	探索测试	可靠性测试		

# 黑盒测试和白盒测试

- **黑盒测试**，也可以称为功能测试、数据驱动测试或基于规格说明的测试。测试者不了解程序的内部情况，不需具备程序内部知识，只知道程序的输入、输出和系统的功能，这是从用户的角度针对软件界面、功能及外部结构进行测试，而不考虑程序内部逻辑结构。
- **白盒测试**，也称结构测试、逻辑驱动测试或基于程序本身的测试。测试程序的内部结构和运作，而不是测试应用程序的功能。

黑盒测试方法	功能划分	等价类划分	边界值分析	因果分析	错误推测
白盒测试方法	语句覆盖	逻辑覆盖	分支覆盖		

# 动态测试和静态测试

- **动态测试**通过运行被测程序，检查运行结果与预期结果的差异，并分析运行效率、正确性和健壮性等性能。这种方法由三部分组成：构造测试用例、执行程序、分析程序的输出结果。
- **静态测试**是指不运行被测程序本身，仅通过分析或检查源程序的语法、结构、过程、接口等来检查程序的正确性。

动态测试	功能测试	接口测试	覆盖率分析	性能分析	内存分析等
静态测试	代码检查	程序结构分析	代码质量度量等		

# 单元测试

- 测试最小软件设计单元，确保模块编码正确
- 单元测试检查特定一段代码的功能，做功能测试
- 在面向对象编程中，单元测试一般在类级别做检查
- 单元测试由开发者自己编写，检查一些代码分支和边界值，保证功能独立运行正常



# 自动化功能测试

- 自动化功能测试是指使用特定的软件控制测试执行，比对实际结果和期望结果，完成功能测试
- 可以自动录制功能测试脚本，也可以人工编写自动化测试脚本
- 自动录制功能测试脚本过程就是将人在软件上的操作录制，并转换成自动化工具可以识别的脚本的过程
- 自动化功能测试过程就是回放人对软件的操作的过程，并比对实际结果和期望结果，判定测试是否通过。测试过程中可以用截屏、输出日志等形式记录测试过程。
- 自动化功能测试对被测软件变化的容忍性比较差，无法有效应对测试过程中出现的抖动和意外事件。
- 针对不同的测试对象，如网页、手机应用、云服务、桌面软件等，需要不同的自动化功能测试软件

# 性能测试

- 性能测试为系统增加工作负载，收集性能参数，测试响应性和稳定性，用来研究、测量、检查和验证可伸缩性、可靠性和资源消耗等。性能测试测试软件在系统中的运行性能，度量系统与预定义目标的差距。主要指标包括：业务用户量、响应时间、吞吐量；机器CPU、IO、内存等
- 负载测试：通过逐步增加系统负载，确定在满足性能指标的情况下，系统所能承受的最大负载量。
- 压力测试：通过逐步增加系统 负载，确定在什么负载条件下系统处于失效状态，以此来获得系统能提供的最大服务级别。
- 容量测试：确定系统可处理的同时最大在线用户数，给系统增加超额的负载检查它是否能正确处理。

# 探索测试

- 探索测试是在进行软件测试时，同时学习测试对象，探索开发更多不同种类的测试方法，改善测试流程的一种测试方法。
- 区别于即兴测试（Ad hoc测试），探索测试是一个有思考和学习的测试过程。
- 区别于传统软件测试严格地“先设计，后执行”，探索性测试强调测试设计和测试执行的同时性。
- 探索测试过程一般为识别软件产品的目的，识别软件的功能，识别软件潜在问题区，记录学习结果和发现的问题。
- 探索测试可以和传统测试流程结果，在测试执行阶段使用。

# 兼容性测试

软件兼容性测试是检查程序和硬件及其他软件之间的兼容性的测试

- 软件和操作系统兼容性
- 软件之间兼容性
- 软件在不同浏览器兼容性
- 软件和数据库兼容性
- 软件和中间件兼容性
- 软件和硬件兼容性

# 测试工具（开源）

单元测试	Junit	NUnit	xUnit等		
自动化功能测试	Selenium Framework	Robot Framework	Sahi	SoupUI	Gauge等
性能测试	Apache JMeter	Gatling	Grinder	SoupUI等	
持续的测试	Jenkins				

# 测试实例：Swagger Petstore API功能测试(1/4)

- 测试对象：
  - API: Swagger Petstore /pet/{petId} HTTP GET操作。此操作通过ID查找Pet。
  - API描述：  
<http://petstore.swagger.io/v2/swagger.json> 中的 /pet/{petId} get操作部分(右图)。
- 测试目的：验证在不同输入值下/pet/{petId}有正确的返回内容和返回码。
- 测试方法和技术：接口测试、功能测试、黑盒、动态。
- 测试工具：Swagger UI, 或者RestClient, PostMan, SoapUI, cURL等支持API测试或者HTTP操作的工具。

本实例测试对象和测试工具取自Swagger Petstore示例页面  
<http://petstore.swagger.io>



# 测试实例：Swagger Petstore API功能测试(2/4)

- 测试用例设计和执行（测试返回码200场景）
  - 测试准备：
    - 使用Swagger Petstore页面中的Swagger UI通过POST /pet API创建一个新Pet（Post Body使用Example Value，并记录下Example Value中的category、name、photoUrls、status、tags等内容，Parameter content type使用默认的application/json），记录下返回的XML消息中的id，此id即为新创建的Pet的ID。

The screenshot displays the Swagger UI interface for the POST /pet endpoint. The top section shows the endpoint name and description: "POST /pet: Add a new pet to the store". Below this, the "Parameters" section lists the required body parameter. The "Example ValueModel" is shown as a JSON object with fields: id, category, name, photoUrls, tags, and status. The "Parameter content type" is set to "application/json". The "Execute" button is visible.

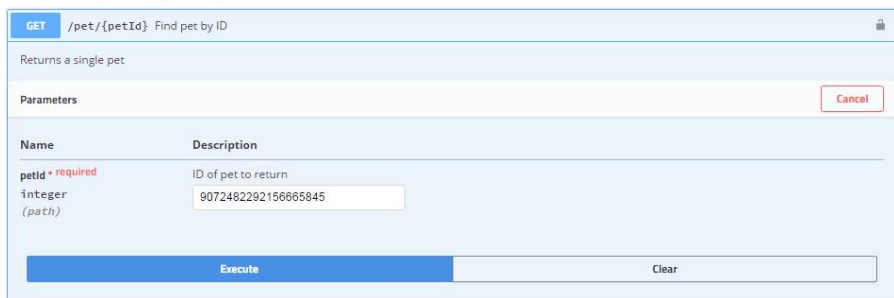
The bottom section shows the "Responses" tab, displaying the server response for a 200 status code. The response body is an XML document with the following structure:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Pet>
  <category>
    <id>0</id>
    <name>string</name>
  </category>
  <id>987248229215665845</id>
  <name>doggie</name>
  <photoUrls>
    <photoUrl>string</photoUrl>
  </photoUrls>
  <status>available</status>
  <tags>
    <tag>
      <id>0</id>
      <name>string</name>
    </tag>
  </tags>
</Pet>
```

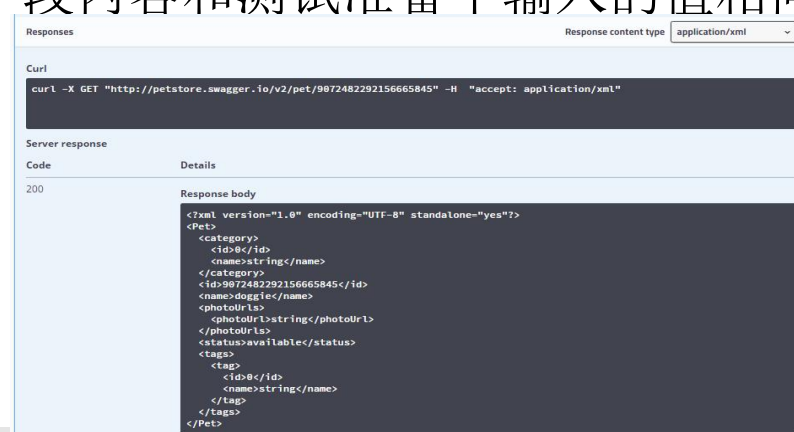
The response headers are also visible, including: access-control-allow-origin: \*, date: Sun, 07 May 2017 15:00:54 GMT, server: Jetty(9.2.9.v20150224), connection: close, access-control-allow-headers: Content-Type, api\_key, Authorization, access-control-allow-methods: GET, POST, DELETE, PUT, and content-type: application/xml.

# 测试实例：Swagger Petstore API功能测试(3/4)

- 测试用例设计和执行（测试返回码200场景）- 测试步骤：
  - ▣ 执行步骤：在Swagger Petstore页面的Swagger UI中打开GET /pet/{petId} 操作区。
  - ▣ 执行步骤：点击Parameters 右边的Try it out, 在出现的petID输入框中输入测试准备中得到的Pet ID。
  - ▣ 执行步骤：点击Execute, 等待返回结果。
  - ▣ 检查步骤：检查返回码正确性：检查Reponse->Server Response->Code 内容是200。
  - ▣ 检查步骤：检查返回Pet字段内容正确性：检查Reponse->Server Response->Details->Response Body中的XML里的Pet字段内容和测试准备中输入的值相同。



The image shows the Swagger UI for the GET /pet/{petId} endpoint. The endpoint is described as "Returns a single pet". Under the "Parameters" section, there is a table with two columns: "Name" and "Description". The first row shows "petId" with a description "ID of pet to return". Below this, there is a text input field containing the value "9072482292156665845". At the bottom, there are two buttons: "Execute" and "Clear".



The image shows the "Server response" details in the Swagger UI. The "Code" is 200. The "Response body" is displayed as XML. The XML content is as follows:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Pet>
  <category>
    <id>0</id>
    <name>string</name>
  </category>
  <id>9072482292156665845</id>
  <name>doggie</name>
  <photoUrls>
    <photoUrl>string</photoUrl>
  </photoUrls>
  <status>available</status>
  <tags>
    <tag>
      <id>0</id>
      <name>string</name>
    </tag>
  </tags>
</Pet>
```



# 测试实例：Swagger Petstore API功能测试(4/4)

- 测试用例设计和执行（测试返回码200场景）－ 测试结束：
  - 使用Swagger Petstore页面中的Swagger UI通过DELETE /pet/{petId}，指定id将测试准备中创建的Pet删除。

**DELETE** /pet/{petId} Deletes a pet

Parameters Cancel

Name	Description
api_key string (header)	api_key
petid * required integer (path)	Pet id to delete 9072482292156665845

Execute Clear

Responses Response content type: application/xml

Curl

```
curl -X DELETE "http://petstore.swagger.io/v2/pet/9072482292156665845" -H "accept: application/xml"
```

Server response

Code	Details
200	Response headers

# 测试对象新发展

- 移动App数量越来越多，Android系统手机碎片化严重。
- Web前端框架丰富，前端功能越来越丰富，代码越来越多。
- 云端服务架构不断演化，微服务架构产品越来越多，使用API接口做服务调用。
- 物联网设备系统平台多，硬件计算存储有限，网络连接性差异大，设备运行环境苛刻，安全性要求高。

# 测试组织和流程新变化

- 业务敏捷化，软件产品研发流程敏捷化，DevOps采纳度越来越高。
- 测试活动的质量目标发生变化，由单纯验证产品功能需求、非功能需求扩展为保证客户满意和业务的可持续。
- QA和开发人员的角色融合。全栈程序员端到端地负责产品的开发、测试和运维，承担了QA人员的角色。
- 持续集成的自动化。持续集成的自动化程度越来越高，实现从代码提交到自动单元测试再到构建打包的持续集成，并和自动化测试环境部署、自动化测试打通。
- 基于BDD（Behavior Driven Development）和TDD（Test Driven Development）的左移测试。BDD和TDD的方式可以帮助开发人员和产品经理对需求达成一致的理解，并保证开发结果满足预期。

# 用例管理

- 用例管理集中管理每个迭代的测试用例，并和需求双向追溯。

<input type="checkbox"/>	用例名称 ▾	用例编号 ▾	用例级别 ▾	基线用例 ▾	归属迭代 ▾	需求编号	需求名称	创建时间 ▾
<input type="checkbox"/>	testttttt	软件开发云00...	L2	否	华为大连...	#3700	【代码检查服务...	2017-01-12 09:48...
<input type="checkbox"/>	扫码关注【华...	软件开发云00...	L2	否	华为大连...	#3700	【代码检查服务...	2017-01-12 09:40...
<input type="checkbox"/>	登录华为软件...	001_1_12	L1	否	华为大连...	#8424	【流水线 CodeP...	2017-01-12 09:40...
<input type="checkbox"/>	登录华为软件...	001_1_1	L1	否	华为大连...	#8425	【部署 Deploy...	2017-01-12 09:40...
<input type="checkbox"/>	登录华为软件...	001_1	L1	否	华为大连...	#8426	【开发客户端 C...	2016-12-19 09:20...
<input type="checkbox"/>	扫码关注【华...	软件开发云001	L2	否	华为大连...	#8427	【扫码关注·精彩...	2016-12-19 09:18...
<input type="checkbox"/>	将本地SVN项...	002	L1	否	里程碑1	#2944	【项目管理 Proj...	2016-08-16 09:31...
<input type="checkbox"/>	登录软件开发...	001	L1	否	华为大连...	#8427	【扫码关注·精彩...	2016-08-16 09:26...

## 需求驱动测试设计

- 根据产品需求创建测试用例，避免需求漏测，规范测试目标。



# 测试执行

- 根据测试用例中的测试步骤执行测试，验证检查点，创建缺陷，设置执行结果。

处理人：

用例等级： L2

状态： + 新建

关联需求： 执行手工测试

前置条件：

测试操作

创建人： xueer

归属迭代： 迭代1

执行结果： --

归属目录： 精简流程Demo

测试步骤：

预期结果：

①

用例详情

设置结果

\* 执行结果：

成功

☒ 同时把用例状态修改为完成

备注：

请在此处输入备注

确定

取消

缺陷 (+)

缺陷编号	标题	处理人	重要程度	状态
------	----	-----	------	----

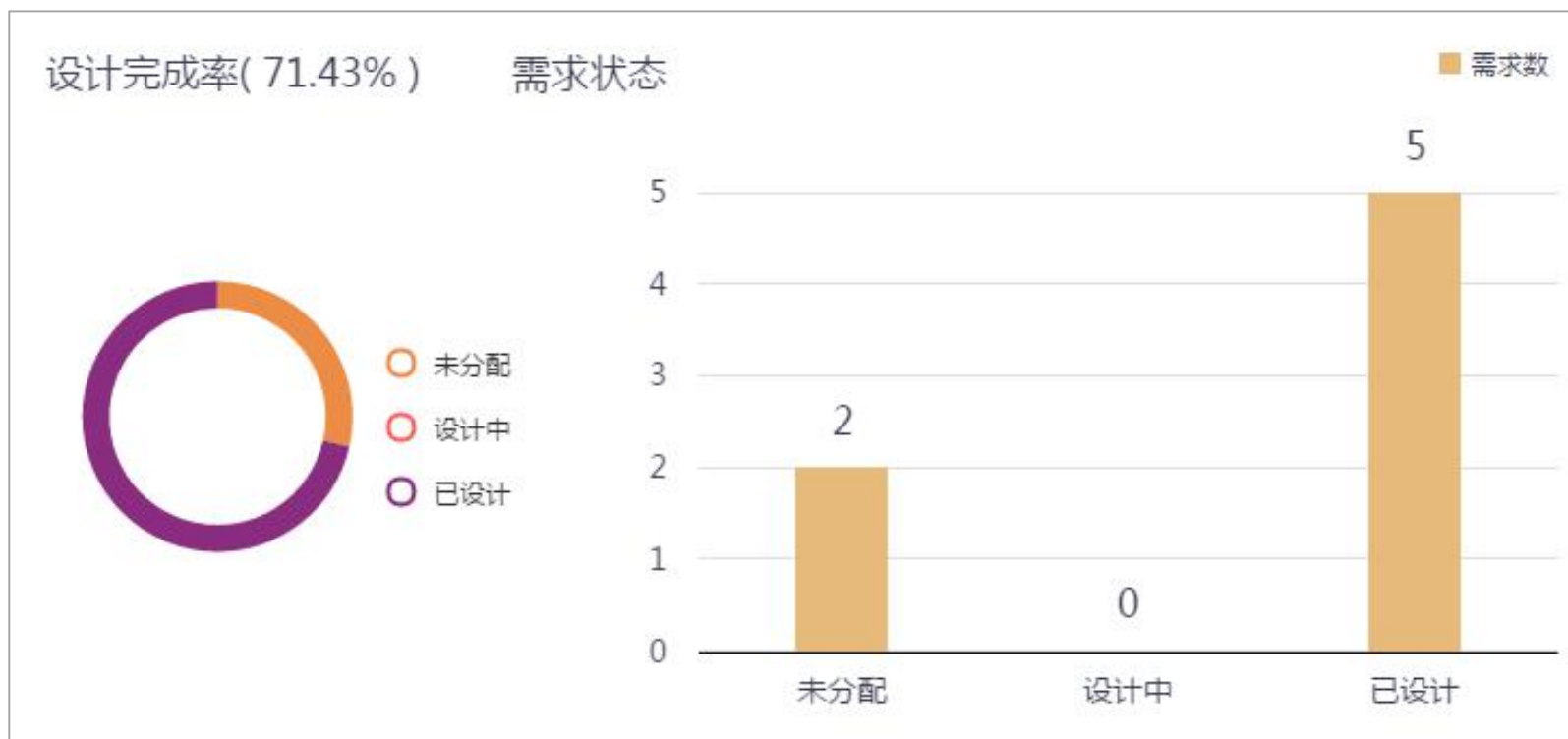
# 测试验收报告

- 自动生成测试验收报告，展示需求完成率、用例通过率、缺陷数目、未测/已测需求详单、缺陷详单。



# 仪表盘监控测试设计完成率

- 通过双向追溯，确保所有需求都被测试，减少漏测。





# 仪表盘监控测试用例通过率

- 展示实时用例执行状态和通过率。

