第25章 Android 式多版本库协同

Android 是谷歌（Google）开发的适合手持设备的操作系统，提供了当前最吸引眼球的开源的手持设备操作平台，大有超越苹果（Apple.com）专有的 iOS 系统的趋势。而 Android 的源代码就是使用 Git 进行维护的。Android 项目在使用 Git 进行源代码管理上有两个伟大的创造，一个是用 Java 开发的名为 Gerrit 的代码审核服务器（将在第 5 篇第 32 章专题介绍），另外一个就是本章要重点介绍的 repo。

repo 是一个用 Python 语言开发的命令行工具，可以更方便地进行多版本库的管理。先来看看 Android 到底包含了多少个 Git 库：

- Android 的版本库管理工具 repo：

  git://android.git.kernel.org/tools/repo.git

- 保存 GPS 配置文件的版本库

  git://android.git.kernel.org/device/common.git

- 160 多个其他的版本库（截至 2010 年 10 月）。

如果把 160 多个版本库都列在这里，恐怕各位的下巴都会掉下来。那么为什么 Android 的版本库会有这么多呢？怎么管理这么复杂的版本库呢？

Android 版本库众多的原因，主要是版本库太大，以及 Git 不能部分检出。Android 的版本库有接近 2 个 GB 之多。如果把所有的东西都放在一个库中，而某个开发团队感兴趣的可能就是某个驱动，或者是某个应用，却要下载如此庞大的版本库，是有些说不过去。

好了，既然接受了 Android 有多达 160 多个版本库这一事实，那么 Android 是不是用之前介绍的“子模组”方式组织起来的呢？如果真的用“子模组”方式来管理这 160 多个代码库，可能就需要如此管理：

- 建立一个索引版本库，在该版本库中，通过子模组方式，将目录一个一个地对应到这 160 多个版本库。
- 对此索引版本库执行克隆操作后，再执行 `git submodule init` 命令。
- 当执行 `git submodule update` 命令时，开始分别克隆这 160 多个版本库。
- 如果想修改某个版本库中的内容，需要进入到相应的子模组目录，执行切换分支的操作。因为子模组是以某个固定提交的状态存在的，是不能更改的，必须先切换到某个工作分支后，才能进行修改和提交。

- 如果要将所有的子模组都切换到某个分支（如 master）进行修改，必须自己通过脚本对这 160 多个版本库一一进行切换。
- Android 有多个版本：android-1.0、android-1.5、……、android-2.2_r1.3、……如何维护这么多的版本呢？也许索引库要通过分支和里程碑，与子模组的各个不同的提交状态进行对应。但是由于子模组的状态只是一个提交 ID，如何能够动态地指定到分支，真的给不出答案。

幸好上面只是假设。聪明的 Android 程序设计师一早就考虑到了 Git 子模组的局限性，以及多版本库管理的问题，开发出了 repo 这一工具。

关于 repo 有这么一则小故事：Android 之父安迪·鲁宾在回应乔布斯关于 Android 太开放导致开发维护更麻烦的言论时，在 Twitter[1] 上留了下面这段简短的话：

```
the definition of open: "mkdir android ; cd android ; repo init -u
git://android.git.kernel.org/platform/manifest.git ; repo sync ; make"
```

是的，就是 repo 让 Android 的开发变得如此简单。

25.1 关于 repo

Repo 是 Google 开发的用于管理 Android 版本库的一个工具。Repo 并不是用于取代 Git，而是用 Python 对 Git 进行了一定的封装，简化了对多个 Git 版本库的管理。对于 repo 管理的任何一个版本库，都还是需要使用 Git 命令进行操作。

Repo 的使用过程大致如下：

（1）运行 `repo init` 命令，克隆 Android 的一个清单库。这个清单库和前面假设的“子模组”方式工作的索引库不同，是通过 XML 技术建立的版本库清单。

（2）清单库中的 manifest.xml 文件，列出了 160 多个版本库的克隆方式。包括版本库的地址和工作区地址的对应关系，以及分支的对应关系。

（3）运行 `repo sync` 命令，开始同步，即分别克隆这 160 多个版本库到本地的工作区中。

（4）同时对 160 多个版本库执行切换分支操作，切换到某个分支。

25.2 安装 repo

首先下载 repo 的引导脚本，可以使用 `wget`、`curl` 甚至浏览器从地址

[1] http://twitter.com/Arubin

http://android.git.kernel.org/repo 下载。把 repo 脚本设置为可执行，并复制到可执行的路径中。在 Linux 上可以用下面的指令将 repo 下载并复制到用户主目录的 bin 目录下。

```
$ curl -L -k http://android.git.kernel.org/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

为什么说下载的 repo 只是一个引导脚本（bootstrap）而不是直接称为 repo 呢？因为 repo 的大部分功能代码不在其中，下载的只是一个帮助完成整个 repo 程序继续下载和加载的工具。如果您是一个程序员，对 repo 的执行比较好奇，可以一起来分析一下 repo 引导脚本。否则可以直接跳到下一节。

看看 repo 引导脚本的前几行（为方便描述，把注释和版权信息过滤掉了），会发现一个神奇的魔法：

```
 1  #!/bin/sh
 2
 3  REPO_URL='git://android.git.kernel.org/tools/repo.git'
 4  REPO_REV='stable'
 5
 6  magic='--calling-python-from-/bin/sh--'
 7  """exec" python -E "$0" "$@" """#$magic"
 8  if __name__ == '__main__':
 9    import sys
10    if sys.argv[-1] == '#%s' % magic:
11      del sys.argv[-1]
12  del magic
```

Repo 引导脚本是用什么语言开发的？这是一个问题。

- 第 1 行，有经验的 Linux 开发者会知道此脚本是用 Shell 脚本语言开发的。
- 第 7 行，是这个魔法的最神奇之处。既是一条合法的 shell 语句，又是一条合法的 python 语句。
- 第 7 行如果作为 shell 语句，执行 exec，用 python 调用本脚本，并替换本进程。三引号在这里相当于一个空字串和一个单独的引号。
- 第 7 行如果作为 python 语句，三引号定义的是一个字符串，字符串后面是一个注释。
- 实际上第 1 行到第 7 行，既是合法的 shell 语句又是合法的 python 语句。从第 8 行开始后面都是 python 脚本了。
- Repo 引导脚本无论是用 shell 执行，还是用 python 执行，效果都相当于使用 python 执行此脚本。

Repo 脚本的真正位置在哪里？可以通过分析引导脚本 repo 得到。在引导脚本 `repo` 的 `main` 函数中，首先调用 `_FindRepo` 函数，从当前目录开始依次向上递归查找 `.repo/repo/main.py` 文件。

```
def main(orig_args):
  main, dir = _FindRepo()
```

函数 `_FindRepo` 返回找到的 `.repo/repo/main.py` 脚本文件，以及包含 `repo/main.py` 的 `.repo` 目录。如果找到 `.repo/repo/main.py` 脚本，则把程序的控制权交给 `.repo/repo/main.py` 脚本（省略了在 repo 开发库中执行情况的判断）。

在下载 `repo` 引导脚本后，没有初始化之前，当然不会存在 `.repo/repo/main.py` 脚本，这时必须进行初始化操作。

25.3 repo 和清单库的初始化

下载并保存 repo 引导脚本后，建立一个工作目录，这个工作目录将作为 Android 的工作区目录。在工作目录中执行 `repo init -u <url>`，完成 repo 完整的下载及项目清单版本库（manifest.git）的下载。

```
$ mkdir working-directory-name
$ cd working-directory-name
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

命令 `repo init` 要完成如下操作：

- 完成 repo 这一工具的完整下载，因为现在有的不过是 repo 的引导程序。

  初始化操作会从 android 的代码中克隆 `repo.git` 库到当前目录下的 `.repo/repo` 目录下。在完成 `repo.git` 克隆之后， `repo init` 命令会将控制权交给工作区的 `.repo/repo/main.py` ，这个刚刚从 `repo.git` 库克隆来的脚本文件，继续进行初始化。

- 克隆 android 的清单库 `manifest.git`（地址来自于 `-u` 参数）。
- 克隆的清单库位于 `.repo/manifests.git` 中，本地克隆到 `.repo/manifests` 。清单文件 `.repo/manifest.xml` 只是符号链接，它指向 `.repo/manifests/default.xml` 。
- 询问用户的姓名和邮件地址，如果和 Git 默认的用户名、邮件地址不同，则记录在 `.repo/manifests.git` 库的 `config` 文件中。

- 命令 `repo init` 还可以附带 `--mirror` 参数，以建立和上游 Android 的版本库一模一样的镜像。这会在后面的章节介绍。

1. 从哪里下载 repo.git ？

在 repo 引导脚本的前几行，定义了默认的 `repo.git` 的版本库位置及要检出的默认分支。

```
REPO_URL='git://android.git.kernel.org/tools/repo.git'
REPO_REV='stable'
```

如果不想从默认 URL 地址获取 repo，或者不想获取稳定版（stable 分支）的 repo，可以在 `repo init` 子命令中通过下面的参数覆盖默认的设置，从指定的源地址克隆 repo 代码库：

- 参数 `--repo-url`，用于设定 repo 的版本库地址。
- 参数 `--repo-branch`，用于设定要检出的分支。
- 参数 `--no-repo-verify`，设定不要对 repo 的里程碑签名进行严格的验证。

实际上，完成 `repo.git` 版本库的克隆，这个 repo 引导脚本就江郎才尽了，`repo init` 命令的后续处理（以及其他子命令）都交给刚刚克隆出来的 `.repo/repo/main.py` 来继续执行。

2. 清单库是什么？从哪里下载？

清单库实际上只包含一个 `default.xml` 文件。这个 XML 文件定义了多个版本库和本地地址的映射关系，是 repo 工作的指引文件。所以在使用 repo 引导脚本进行初始化的时候，必须通过 -u 参数指定清单库的源地址。

清单库的下载，是通过 `repo init` 命令初始化时，用 `-u` 参数指定清单库的位置。例如 repo 针对 Android 代码库进行初始化时执行的命令：

```
$ repo init -u git://android.git.kernel.org/platform/manifest.git
```

Repo 引导脚本的 `init` 子命令可以使用下列和清单库相关的参数：

- 参数 `-u`（`--manifest-url`）：设定清单库的 Git 服务器地址。
- 参数 `-b`（`--manifest-branch`）：检出清单库的特定分支。
- 参数 `--mirror`：只在 repo 第一次初始化的时候使用，以和 Android 服务器同样的结构在本地建立镜像。
- 参数 `-m`（`--manifest-name`）：当有多个清单文件时，可以指定清单库的某个清单文件为有效的清单文件。默认为 `default.xml`。

Repo 初始化命令（repo init）可以执行多次：

- 不带参数地执行 repo init ，从上游的清单库获取新的清单文件 default.xml 。
- 使用参数 -u（--manifest-url）执行 repo init ，会重新设定上游的清单库地址，并重新同步。
- 使用参数 -b（--manifest-branch）执行 repo init ，会使用清单库的不同分支，以便在使用 repo sync 时将项目同步到不同的里程碑。
- 但是不能使用 --mirror 命令，该命名只能在第一次初始化时执行。那么如何将已经按照工作区模式同步的版本库转换为镜像模式呢？后面会看到一个解决方案。

25.4 清单库和清单文件

执行完 repo init 之后，工作目录内空空如也。实际上有一个 .repo 目录。在该目录下除了一个包含 repo 实现的 repo 库克隆外，就是 manifest 库的克隆，以及一个符号链接，链接到清单库中的 default.xml 文件。

```
$ ls -lF .repo/
drwxr-xr-x 3 jiangxin jiangxin 4096 2010-10-11 18:57 manifests/
drwxr-xr-x 8 jiangxin jiangxin 4096 2010-10-11 10:08 manifests.git/
lrwxrwxrwx 1 jiangxin jiangxin   21 2010-10-11 10:07 manifest.xml ->
manifests/default.xml
drwxr-xr-x 7 jiangxin jiangxin 4096 2010-10-11 10:07 repo/
```

在工作目录下的 .repo/manifest.xml 文件就是 Android 项目的众多版本库的清单文件。Repo 命令的操作都要参考这个清单文件。

打开清单文件会看到如下内容：

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <manifest>
 3   <remote  name="korg"
 4            fetch="git://android.git.kernel.org/"
 5            review="review.source.android.com" />
 6   <default revision="master"
 7            remote="korg" />
 8
 9   <project path="build" name="platform/build">
10     <copyfile src="core/root.mk" dest="Makefile" />
11   </project>
12
13   <project path="bionic" name="platform/bionic" />
```

```
      ...
181 </manifest>
```

这个文件不太复杂，是么？

- 这个 XML 的顶级元素是 `manifest` ，见第 2 行和第 181 行。
- 第 3 行通过一个 remote 元素，定义了名为 korg（kernel.org 缩写）的远程版本库，其 Git 库的基址为 `git://android.git.kernel.org/` 。还定义了代码审核服务器的地址 `review.source.android.com` 。还可以定义更多的 remote 元素，这里只定义了一个。
- 第 6 行用于设置各个项目默认的远程版本库（remote）为 korg，默认的分支为 `master` 。当然各个项目（project 元素）可以定义自己的 remote 和 revision 覆盖该默认配置。
- 第 9 行定义了一个项目，该项目的远程版本库相对路径为：`platform/build` ，在工作区克隆的位置为：`build` 。
- 第 10 行，即 project 元素的子元素 copyfile，定义了项目克隆后的一个附加动作：从 `core/root.mk` 拷贝文件至 `Makefile` 。
- 第 13 行后后续的 100 多行定义了其他 160 个项目，都是采用类似的 project 元素语法。name 参数定义远程版本库的相对路径，path 参数定义克隆到本地工作区的路径。
- 还可以出现 manifest-server 元素，其 url 属性定义了通过 XMLRPC 提供实时更新清单的服务器 URL。只有当执行 `repo sync --smart-sync` 的时候才会检查该值，并用动态获取的 manifest 覆盖掉默认的清单。

25.5 同步项目

在工作区执行下面的命令，会参照 `.repo/manifest.xml` 清单文件，将项目所有相关的版本库全部克隆出来。不过最好请在读完本节内容之后再尝试执行这条命令。

```
$ repo sync
```

对于 Android，这个操作需要通过网络传递接近 2 个 GB 的内容，如果带宽不是很高的话，可能会花掉几个小时甚至是一天的时间。

也可以仅克隆感兴趣的项目，在 `repo sync` 后面跟上项目的名称。项目的名称来自于 `.repo/manifest.xml` 这个 XML 文件中 project 元素的 name 属性值。例如克隆

platform/build 项目：

```
$ repo sync platform/build
```

Repo 有一个功能可以在这里展示，就是 repo 支持通过本地清单对默认的清单文件进行补充及覆盖。即可以在 .repo 目录下创建 local_manifest.xml 文件，其内容会和 .repo/manifest.xml 文件的内容进行合并。

在工作目录下运行下面的命令可以创建一个本地清单文件。这个本地定制的清单文件来自默认文件，但是删除了 remote 元素和 default 元素，并将所有的 project 元素都重命名为 remove-project 元素。这实际相当于对原有的清单文件“取反”。

```
$ curl -L -k  \
      http://www.ossxp.com/doc/gotgit/download/ch25/manifest-revert.xslt \
      > manifest-revert.xslt
$ xsltproc manifest-revert.xslt .repo/manifest.xml > .repo/local_manifest.xml
```

用下面的这条命令可以看到 repo 运行时实际获取到的清单。这个清单来自于 .repo/manifest.xml 和 .repo/local_manifest.xml 两个文件的汇总。

```
$ repo manifest -o -
```

当执行 repo sync 命令时，实际上就是依据合并后的清单文件进行同步。如果清单中的项目被自定义清单全部“取反”，执行同步就不会同步任何项目，甚至会删除已经完成同步的项目。

本地定制的清单文件 local_manifest.xml 支持前面介绍的清单文件的所有语法，需要注意的是：

- 不能出现重复定义的 remote 元素。这就是为什么上面的脚本要删除来自默认 manifest.xml 的 remote 元素。
- 不能出现 default 元素，因为全局只能有一个。
- 不能出现重复的 project 定义（name 属性不能相同），但是可以通过 remove-project 元素将默认清单中定义的 project 删除然后再重新定义。

试着编辑 .repo/local_manifest.xml，在其中再添加几个 project 元素，然后试着用 repo sync 命令进行同步。

25.6 建立 android 代码库本地镜像

Android 为企业提供一个新的市场，无论企业大小都是处于同一个起跑线上。研究 Android 尤其是 Android 系统核心或驱动的开发，首先要做的就是本地克隆建立一套 Android 版本库管理机

制。因为 Android 的代码库是那么庞杂，如果一个开发团队每个人都去执行 `repo init -u`，再执行 `repo sync` 从 Android 服务器克隆版本库的话，多大的网络带宽恐怕都不够用。唯一的办法是本地建立一个 Android 版本库的镜像。

建立本地镜像非常简单，就是在执行 `repo init -u` 初始化的时候，附带上 `--mirror` 参数。

```
$ mkdir android-mirror-dir
$ cd android-mirror-dir
$ repo init --mirror -u git://android.git.kernel.org/platform/manifest.git
```

之后执行 `repo sync` 就可以安装 Android 的 Git 服务器方式来组织版本库，创建一个 Android 版本库镜像。

实际上附带了 `--mirror` 参数执行 `repo init -u` 命令，会在克隆的 `.repo/manifests.git` 下的 `config` 中记录配置信息：

```
[repo]
    mirror = true
```

1. 从 android 的工作区到代码库镜像

在初始化 repo 工作区时，使用不带 `--mirror` 参数的 `repo init -u`，并完成代码同步后，如果再次执行 `repo init` 并附带了 `--mirror` 参数，repo 会报错退出：“fatal: --mirror not supported on existing client”。实际上`--mirror` 参数只能对尚未初始化的 repo 工作区执行。

那么如果之前没有用镜像的方法同步 Android 版本库，难道要为创建代码库镜像再重新执行一次 repo 同步么？要知道重新同步一份 Android 版本库是非常慢的。我就遇到了这个问题。

不过既然有 `manifest.xml` 文件，完全可以对工作区进行反向操作，将工作区转换为镜像服务器的结构。下面就是一个示例脚本，可以 Github 上的本书相关版本库[1]中下载。这个脚本利用了已有的 repo 代码进行实现，所以看着很简洁。 8-)

脚本 `work2mirror.py` 如下：

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import os, sys, shutil
```

[1] https://github.com/ossxp-com/gotgit/raw/master/download/ch25/work2mirror.py

```
cwd = os.path.abspath( os.path.dirname( __file__ ) )
repodir = os.path.join( cwd, '.repo' )
S_repo = 'repo'
TRASHDIR = 'old_work_tree'

if not os.path.exists( os.path.join(repodir, S_repo) ):
    print >> sys.stderr, "Must run under repo work_dir root."
    sys.exit(1)

sys.path.insert( 0, os.path.join(repodir, S_repo) )
from manifest_xml import XmlManifest

manifest = XmlManifest( repodir )

if manifest.IsMirror:
    print >> sys.stderr, "Already mirror, exit."
    sys.exit(1)

trash = os.path.join( cwd, TRASHDIR )

for project in manifest.projects.itervalues():
    #将旧的版本库路径移动到镜像模式下新的版本库路径
    newgitdir = os.path.join( cwd, '%s.git' % project.name )
    if os.path.exists( project.gitdir ) and project.gitdir != newgitdir:
        if not os.path.exists( os.path.dirname(newgitdir) ):
            os.makedirs( os.path.dirname(newgitdir) )
        print "Rename %s to %s." % (project.gitdir, newgitdir)
        os.rename( project.gitdir, newgitdir )

    #将工作区移动到待删除目录
    if project.worktree and os.path.exists( project.worktree ):
        newworktree = os.path.join( trash, project.relpath )
        if not os.path.exists( os.path.dirname(newworktree) ):
            os.makedirs( os.path.dirname(newworktree) )
        print "Move old worktree %s to %s." % (project.worktree, newworktree )
        os.rename( project.worktree, newworktree )

    if os.path.exists ( os.path.join( newgitdir, 'config' ) ):
        # 修改版本库的配置
        os.chdir( newgitdir )
        os.system( "git config core.bare true" )
        os.system( "git config remote.korg.fetch
'+refs/heads/*:refs/heads/*'" )
```

```
        # 删除 remotes 分支，因为作为版本库镜像不需要 remote 分支
        if os.path.exists ( os.path.join( newgitdir, 'refs', 'remotes' ) ):
            print "Delete " + os.path.join( newgitdir, 'refs', 'remotes' )
            shutil.rmtree( os.path.join( newgitdir, 'refs', 'remotes' ) )

# 设置 menifest 为镜像
mp = manifest.manifestProject
mp.config.SetString('repo.mirror', 'true')
```

使用方法很简单，只要将脚本放在 Android 工作区下，执行就可以了。执行完毕会将原有工作区的目录移动到 `old_work_tree` 子目录下，在确认原有工作区没有未提交的数据后，直接删除 `old_work_tree` 即可。

```
$ python work2mirror.py
```

2. 创建新的清单库，或修改原有清单库

建立了 Android 代码库的本地镜像后，如果不对 manifest 清单版本库进行定制，在使用 `repo sync` 同步代码的时候，仍然使用 Android 官方的代码库同步代码，使得本地的镜像版本库形同虚设。解决办法是创建一个自己的 manifest 库，或者在原有清单库中建立一个分支加以修改。如果创建新的清单库，参考 Android 上游的 manifest 清单库进行创建。

25.7 Repo 的命令集

Repo 子命令实际上是 Git 命令的或简单或复杂的封装。每一个 repo 子命令都对应于 repo 源码树中 `subcmds` 目录下的一个同名的 Python 文件。每一个 repo 子命令都可以通过下面的命令获得帮助。

```
repo help <command>
```

通过阅读代码，可以更加深入地了解 repo 子命令的封装。

1. repo init 命令

repo init 子命令主要完成检出清单版本库（manifest.git），以及配置 Git 用户的用户名和邮件地址的工作。

实际上，完全可以进入到 `.repo/manifests` 目录，用 git 命令操作清单库。对 manifests 的

修改不会因为执行 `repo init` 而丢失，除非是处于未跟踪状态。

2. repo sync 命令

repo sync 子命令用于参照清单文件克隆或同步版本库。如果某个项目版本库尚不存在，则执行 `repo sync` 命令相当于执行 `git clone` 。如果项目版本库已经存在，则相当于执行下面的两个命令：

- git remote update

  相当于对每一个 remote 源执行 fetch 操作。

- git rebase origin/branch

  针对当前分支的跟踪分支执行 rebase 操作。不采用 merge 而是采用 rebase，目的是减少提交数量、方便评审（Gerrit）。

3. repo start 命令

repo start 子命令实际上是对 `git checkout -b` 命令的封装。为指定的项目或所有项目（若使用 --all 参数），以清单文件中为项目设定的分支或里程碑为基础，创建特性分支。特性分支的名称由命令的第一个参数指定。相当于执行 checkout -b 。

用法如下：

```
repo start <newbranchname> [--all | <project>...]
```

4. repo status 命令

repo status 子命令实际上是对 `git diff-index`、`git diff-files` 命令的封装，同时显示暂存区的状态和本地文件修改的状态。

用法如下：

```
repo status [<project>...]
```

示例输出：

```
project repo/                                   branch devwork
 -m     subcmds/status.py
 ...
```

上面的示例输出显示了 repo 项目的 devwork 分支的修改状态。

- 每个小节的首行显示项目的名称，以及所在分支的名称。
- 之后显示该项目中文件的变更状态。头两个字母显示变更状态，后面显示文件名或其他变更信息。
- 第一个字母表示暂存区的文件修改状态。

  其实是 `git-diff-index` 命令输出中的状态标识，用大写显示。

  - -：没有改变
  - A：添加（不在 HEAD 中，在暂存区）
  - M：修改（在 HEAD 中，在暂存区，内容不同）
  - D：删除（在 HEAD 中，不在暂存区）
  - R：重命名（不在 HEAD 中，在暂存区，路径修改）
  - C：拷贝（不在 HEAD 中，在暂存区，从其他文件拷贝）
  - T：文件状态改变（在 HEAD 中，在暂存区，内容相同）
  - U：未合并，需要冲突解决
- 第二个字母表示工作区文件的更改状态。

  其实是 `git-diff-files` 命令输出中的状态标识，用小写显示。

  - -：新/未知（不在暂存区，在工作区）
  - m：修改（在暂存区，在工作区，被修改）
  - d：删除（在暂存区，不在工作区）
- 两个表示状态的字母后面，显示文件名信息。如果有文件重命名还会显示改变前后的文件名及文件的相似度。

5. repo checkout 命令

repo checkout 子命令实际上是对 `git checkout` 命令的封装。检出之前由 repo start 创建的分支。

用法如下：

```
repo checkout <branchname> [<project>...]
```

6. repo branches 命令

repo branches 读取各个项目的分支列表并汇总显示。该命令实际上通过直接读取 `.git/refs` 目录下的引用来获取分支列表，以及分支的发布状态等。

用法如下：

```
repo branches [<project>...]
```

输出示例：

```
*P nocolor                | in repo
   repo2                  |
```

- 第一个字段显示分支的状态：是否当前分支，分支是否发布到代码审核服务器上？
- 第一个字母若显示星号（*），含义是此分支为当前分支
- 第二个字母若为大写字母 P，则含义是分支的所有提交都发布到代码审核服务器上了。
- 第二个字母若为小写字母 p，则含义是只有部分提交被发布到代码审核服务器上。
- 若不显示 P 或 p，则表明分支尚未发布。
- 第二个字段为分支名。
- 第三个字段为以竖线（|）开始的字符串，表示该分支存在于哪些项目中。
  - in all projects

    该分支处于所有项目中。
  - in project1 project2

    该分支只在特定项目中定义。如：project1、project2。
  - not in project1

    该分支不存在于这些项目中。即除了 project1 项目外，其他项目都包含此分支。

7. repo diff 命令

repo diff 子命令实际上是对 `git diff` 命令的封装，用于分别显示各个项目工作区下的文件差异。

用法如下：

```
repo diff [<project>...]
```

8. repo stage 命令

repo stage 子命令实际上是对 `git add --interactive` 命令的封装，用于挑选各个项目工作区中的改动（修改、添加等）以加入暂存区。

用法如下：

```
repo stage -i [<project>...]
```

9. repo upload 命令

repo upload 相当于 `git push`，但是又有很大的不同。执行 `repo upload` 不是将版本库改动推送到克隆时的远程服务器，而是推送到代码审查服务器（由 Gerrit 软件架设）的特殊引用上，使用的是 SSH 协议（特殊端口）。代码审核服务器会对推送的提交进行特殊处理，将新的提交显示为一个待审核的修改集，并进入代码审查流程。只有当审核通过后，才会合并到官方正式的版本库中。

用法如下：

```
repo upload [--re --cc] {[<project>]... | --replace <project>}

参数:
  -h, --help            显示帮助信息。
  -t                    发送本地分支名称到 Gerrit 代码审核服务器。
  --replace             发送此分支的更新补丁集。注意使用该参数，只能指定一个项
目。
  --re=REVIEWERS, --reviewers=REVIEWERS
                        要求由指定的人员进行审核。
  --cc=CC               同时发送通知到如下邮件地址。
```

确定推送服务器的端口

分支改动的推送是发给代码审核服务器，而不是下载代码的服务器。使用的协议是 SSH 协议，但是使用的并非标准端口。如何确认代码审核服务器上提供的特殊 SSH 端口呢？

在执行 repo upload 命令时，repo 会通过访问代码审核 Web 服务器的 `/ssh_info` 的 Url 获取 SSH 服务端口，默认为 29418。这个端口，就是 `repo upload` 发起推送的服务器的 SSH 服务端口。

修订集修改后重新传送

只有已经通过 `repo upload` 命令在代码审查服务器上提交了一个修订集，才会得到一个修

订号。关于此次修订的相关讨论会发送到提交者的邮箱中。如果修订集有误没有通过审核，可以重新修改代码，再次向代码审核服务器上传修订集。

一个修订集修改后再次上传，如果修订集的 ID 不变是非常有用的，因为这样相关的修订集都在代码审核服务器的同一个界面中显示。

在执行 `repo upload` 时会弹出一个编辑界面，提示在方括号中输入修订集编号，否则会在代码审查服务器上创建新的 ID。有一个办法可以不用手工输入修订集，如下：

```
repo upload --replace project_name
```

当使用 `--replace` 参数后，repo 会检查本地版本库名为 `refs/published/branch_name` 的特殊引用（上一次提交的修订），获得其对应的提交 SHA1 哈希值。然后在代码审核服务器的 `refs/changes/` 命名空间下的特殊引用中寻找和提交 SHA1 哈希值匹配的引用，找到的匹配引用其名称中就所包含有变更集 ID，直接用此变更集 ID 作为新的变更集 ID 提交到代码审核服务器。

Gerrit 服务器魔法

repo upload 命令执行推送，实际上会以类似如下的命令行格式进行调用：

```
git push --receive-pack='gerrit receive-pack --reviewer charlie@example.com' \
         ssh://review.example.com:29418/project HEAD:refs/for/master
```

Gerrit 服务器接收到 git push 请求后，会自动将对分支的提交转换为修订集，显示于 Gerrit 的提交审核界面中。Gerrit 的魔法破解的关键点就在于 git push 命令的 --receive-pack 参数。即交由 gerrit-receive-pack 命令执行提交，进入非标准的 git 处理流程，将提交转换为在 `refs/changes` 命名空间下的引用，而不在 `refs/for` 命名空间下创建引用。

10. repo download 命令

`repo download` 命令主要用于代码审核者下载和评估贡献者提交的修订。贡献者的修订在 git 版本库中以 `refs/changes/<changeid>/<patchset>` 引用方式命名（默认的 patchset 为 1），和其他 Git 引用一样，用 `git fetch` 获取，该引用所指向的最新的提交就是贡献者待审核的修订。使用 repo download 命令实际上就是用 `git fetch` 获取到对应项目的 `refs/changes/<changeid>/patchset>` 引用，并自动切换到对应的引用上。

用法如下：

```
repo download {project change[/patchset]}...
```

11. repo rebase 命令

`repo rebase` 子命令实际上是对 `git rebase` 命令的封装，该命令的参数也作为 `git rebase` 命令的参数，但 `-i` 参数仅当对一个项执行时才有效。

用法如下：

```
命令行: repo rebase {[<project>...] | -i <project>...}

参数:
  -h, --help            显示帮助并退出
  -i, --interactive     交互式的变基（仅对一个项目时有效）
  -f, --force-rebase    向 git rebase 命令传递 --force-rebase 参数
  --no-ff               向 git rebase 命令传递 -no-ff 参数
  -q, --quiet           向 git rebase 命令传递 --quiet 参数
  --autosquash          向 git rebase 命令传递 --autosquash  参数
  --whitespace=WS       向 git rebase 命令传递 --whitespace  参数
```

12. repo prune 命令

`repo prune` 子命令实际上是对 `git branch -d` 命令的封装，该命令用于扫描项目的各个分支，并删除已经合并的分支。

用法如下：

```
repo prune [<project>...]
```

13. repo abandon 命令

相比 `repo prune` 命令，`repo abandon` 命令更具破坏性，因为 `repo abandon` 是对 `git branch -D` 命令的封装。该命令非常危险，将直接删除分支，请慎用。

用法如下：

```
repo abandon <branchname> [<project>...]
```

14. 其他命令

- repo grep

  相当于对 `git grep` 的封装，用于在项目文件中进行内容查找。

- repo smartsync

  相当于用 `-s` 参数执行 `repo sync` 。

- repo forall

  迭代器，可以对 repo 管理的项目进行迭代。

- repo manifest

  显示 manifest 文件内容。

- repo version

  显示 repo 的版本号。

- repo selfupdate

  用于 repo 自身的更新。如果提供 `--repo-upgraded` 参数，还会更新各个项目的钩子脚本。

25.8 Repo 命令的工作流

图 25-1 是 repo 的工作流，每一个代码贡献都起始于 `repo start` 创建的本地工作分支，最终都以 `repo upload` 命令将代码补丁发布到代码审核服务器。

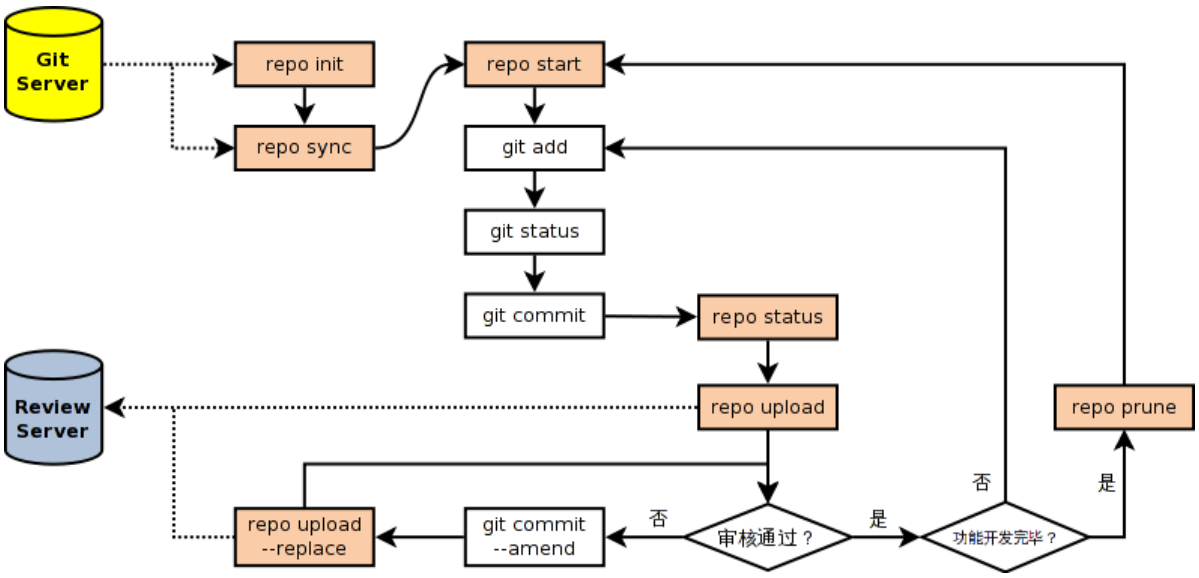

图 25-1：repo 工作流

25.9 好东西不能 android 独享

通过前面的介绍能够体会到 repo 的精巧 —— repo 巧妙地实现了多 Git 版本库的管理。因

为 repo 使用了清单版本库，所以 repo 这一工具并没有被局限于 Android 项目，可以在任何项目中使用。下面就介绍三种 repo 的使用模式，将 repo 引入自己的项目（非 Android 项目）中，其中第三种 repo 使用模式是用我改造后的 repo，实现了脱离 Gerrit 服务器进行推送。

25.9.1 Repo + Gerrit 模式

Repo 和 Gerrit 是 Android 代码管理的两大支柱。正如前面在 repo 工作流中介绍的，部分的 repo 命令从 git 服务器读取，这个 git 服务器可以是只读的版本库控制服务器，还有部分 repo 命令（repo upload、repo download）访问的则是代码审核服务器，其中 repo upload 命令还要向代码审核服务器进行 git push 操作。

在使用未经改动的 repo 来维护自己的项目（多个版本库组成）时，必须搭建 Gerrit 代码审核服务器。

搭建项目的版本控制系统环境的一般方法为：

- 用 Git 协议或 HTTP 协议搭建 Git 服务器。具体搭建方法参见第 5 篇“搭建 Git 服务器”的相关章节。
- 导入 repo.git 工具库。非必须，只是为了减少不必要的互联网操作。
- 还可以在内部 HTTP 服务器维护一个定制的 repo 引导脚本。非必须。
- 建立 Gerrit 代码审核服务器。会在第 5 篇“第 32 章 Gerrit 代码审核服务器”中介绍 Gerrit 的安装和使用。
- 一一创建相关的子项目代码库。
- 建立一个 manifest.git 清单库，其中 remote 元素的 fetch 属性指向只读 Git 服务器地址，review 属性指向代码审核服务器地址。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote  name="example"
           fetch="git://git.example.net/"
           review="review.example.net" />
  <default revision="master"
           remote="example" />

  ...
```

25.9.2 Repo 无审核模式

Gerrit 代码审核服务器部署比较麻烦，更不要说因为 Gerrit 用户界面的学习和用户使用习惯的更改而带来的困难了。在一个固定的团队内部使用 repo 可能真的没有必要使用 Gerrit，因为团队成员都应该熟悉 Git 的操作，团队成员的编程能力都可信，单元测试质量由提交者保证，集成测试由单独的测试团队进行，即团队拥有一套完整、成型的研发工作流，引入 Gerrit 并非必要。

脱离了 Gerrit 服务器，直接跟 Git 服务器打交道，repo 可以工作么？是的，可以利用 `repo forall` 迭代器实现多项目代码的 PUSH，其中有如下关键点需要重点关注。

- `repo start` 命令创建本地分支时，需要使用和上游同样的分支名。

  如果使用不同的分支名，上传时需要提供复杂的引用描述。下面的示例先通过 `repo manifest` 命令确认上游清单库默认的分支名为 master，再使用该分支名（master）作为本地分支名执行 `repo start` 。示例如下：

```
$ repo manifest -o - | grep default
  <default remote="bj" revision="master"/>

$ repo start master --all
```

- 推送不能使用 `repo upload` ，而需要使用 `git push` 命令。

  可以利用 `repo forall` 迭代器实现批命令方式执行。例如：

```
$ repo forall -c git push
```

- 如果清单库中的上游 git 库地址用的是只读地址，需要为本地版本库一一更改上游版本库地址。

  可以使用 `forall` 迭代器，批量为版本库设置 `git push` 时的版本库地址。下面的命令使用的环境变量 `$REPO_PROJECT` 是实现批量设置的关键。

```
$ repo forall -c \
  'git remote set-url --push bj
android@bj.ossxp.com:android/${REPO_PROJECT}.git'
```

25.9.3 改进的 Repo 无审核模式

前面介绍的使用 `repo forall` 迭代器实现在无审核服务器情况下向上游推送提交，只是权宜之计，尤其是用 `repo start` 建立工作分支要求和上游一致，实在是有点强人所难。

我改造了 repo，增加了两个新的子命令 `repo config` 和 `repo push` ，让 repo 可以脱

离 Gerrit 服务器直接向上游推送。代码托管在 Github 上：http://github.com/ossxp-com/repo 。下面简单地介绍一下如何使用改造之后的 repo。

1. 下载改造后的 repo 引导脚本

建议使用改造后的 repo 引导脚本替换原脚本，否则在执行 `repo init` 命令时需要提供额外的 `--no-repo-verify` 参数、 `--repo-url` 和 `--repo-branch` 参数。

```
$ curl -L -k http://github.com/ossxp-com/repo/raw/master/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
```

2. 用 repo 从 Github 上检出测试项目

如果安装了改造后的 repo 引导脚本，使用下面的命令初始化 repo 及清单库。

```
$ mkdir test
$ cd test
$ repo init -u git://github.com/ossxp-com/manifest.git
$ repo sync
```

如果用的是标准的（未经改造的）repo 引导脚本，使用下面的命令。

```
$ mkdir test
$ cd test
$ repo init --repo-url=git://github.com/ossxp-com/repo.git \
  --repo-branch=master --no-repo-verify \
  -u git://github.com/ossxp-com/manifest.git
$ repo sync
```

当子项目代码全部同步完成后，执行 `make` 命令。可以看到各个子项目的版本及清单库的版本。

```
$ make
Version of test1:    1:0.2-dev
Version of test2:    2:0.2
Version of manifest: current
```

3. 用 repo config 命令设置 pushurl

现在如果进入到各个子项目目录，是无法成功执行 `git push` 命令的，因为上游 Git 库的地址是一个只读访问的 URL，无法提供写服务。可以用新增的 `repo config` 命令设置当执行

`git push` 时的 URL 地址。

```
$ repo config repo.pushurl ssh://git@github.com/ossxp-com/
```

设置成功后，可以使用 `repo config repo.pushurl` 查看设置。

```
$ repo config repo.pushurl
ssh://git@github.com/ossxp-com/
```

4. 创建本地工作分支

使用下面的命令创建一个工作分支 `jiangxin` 。

```
$ repo start jiangxin --all
```

使用 `repo branches` 命令可以查看当前所有的子项目都属于 `jiangxin` 分支。

```
$ repo branches
*  jiangxin                 | in all projects
```

参照下面的方法修改 test/test1 子项目。对 test/test2 项目也作类似修改。

```
$ cd test/test1
$ echo "1:0.2-jiangxin" > version
$ git diff
diff --git a/version b/version
index 37c65f8..a58ac04 100644
--- a/version
+++ b/version
@@ -1 +1 @@
-1:0.2-dev
+1:0.2-jiangxin
$ repo status
# on branch jiangxin
project test/test1/                             branch jiangxin
 -m     version
$ git add -u
$ git commit -m "0.2-dev -> 0.2-jiangxin"
```

执行 `make` 命令，看看各个项目的改变。

```
$ make
Version of test1:    1:0.2-jiangxin
Version of test2:    2:0.2-jiangxin
Version of manifest: current
```

5. PUSH 到远程服务器

直接执行 `repo push` 就可以将各个项目的改动进行推送。

```
$ repo push
```

如果有多个项目同时进行了改动，为了避免出错，会弹出编辑器显示因为包含改动而需要推送的项目列表。

```
# Uncomment the branches to upload:
#
# project test/test1/:
#  branch jiangxin ( 1 commit, Mon Oct 25 18:04:51 2010 +0800):
#         4f941239 0.2-dev -> 0.2-jiangxin
#
# project test/test2/:
#  branch jiangxin ( 1 commit, Mon Oct 25 18:06:51 2010 +0800):
#         86683ece 0.2-dev -> 0.2-jiangxin
```

每一行前面的井号都是注释会被忽略。将希望推送的分支前的注释去掉，就可以将该项目的分支执行推送动作。下面的操作中把其中的两个分支的注释都去掉了，这两个项目当前分支的改动会 push 到上游服务器。

```
# Uncomment the branches to upload:
#
# project test/test1/:
branch jiangxin ( 1 commit, Mon Oct 25 18:04:51 2010 +0800):
#         4f941239 0.2-dev -> 0.2-jiangxin
#
# project test/test2/:
branch jiangxin ( 1 commit, Mon Oct 25 18:06:51 2010 +0800):
#         86683ece 0.2-dev -> 0.2-jiangxin
```

保存退出（如果使用 vi 编辑器，输入 <ESC>:wq 执行保存退出）后，马上开始对选择的各个项目执行 `git push`。

```
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 293 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test1.git
   27aee23..4f94123  jiangxin -> master
Counting objects: 5, done.
```

```
Writing objects: 100% (3/3), 261 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test2.git
   7f0841d..86683ec  jiangxin -> master


------------------------------------------------
[OK    ] test/test1/     jiangxin
[OK    ] test/test2/     jiangxin
```

从推送的命令输出可以看出来，本地的工作分支 jiangxin 的改动被推送到远程服务器的 master 分支（本地工作分支跟踪的上游分支）上。

再次执行 `repo push`，会显示没有项目需要推送。

```
$ repo push
no branches ready for upload
```

6. 在远程服务器创建新分支

如果想在服务器上创建一个新的分支，该如何操作呢？如下使用 `--new_branch` 参数调用 `repo push` 命令。

```
$ repo start feature1 --all
$ repo push --new_branch
```

经过同样的编辑操作之后自动调用 git push ，在服务器上创建新分支 `feature1` 。

```
Total 0 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test1.git
 * [new branch]      feature1 -> feature1
Total 0 (delta 0), reused 0 (delta 0)
To ssh://git@github.com/ossxp-com/test2.git
 * [new branch]      feature1 -> feature1


------------------------------------------------
[OK    ] test/test1/     feature1
[OK    ] test/test2/     feature1
```

用 `git ls-remote` 命令查看远程版本库的分支，会发现远程版本库中已经建立了新的分支。

```
$ git ls-remote git://github.com/ossxp-com/test1.git refs/heads/*
4f9412399bf8093e88006847720335182 9a6b1fb        refs/heads/feature1
4f9412399bf8093e88006847720335182 9a6b1fb        refs/heads/master
b2b246b99ca504f141299ecdbadb23faf6918973        refs/heads/test-0.1
```

注意到 feature1 和 master 分支引用指向了相同的 SHA1 哈希值，这是因为 feature1 分支是

直接从 master 分支创建的。

7. 通过不同的清单库版本，切换到不同分支

换用不同的清单库，需要建立新的工作区，并且在执行 `repo init` 时，通过 `-b` 参数指定清单库的分支。

```
$ mkdir test-0.1
$ cd test-0.1
$ repo init -u git://github.com/ossxp-com/manifest.git -b test-0.1
$ repo sync
```

当子项目代码全部同步完成后执行 `make` 命令。可以看到各个子项目的版本及清单库的版本不同于之前的输出。

```
$ make
Version of test1:    1:0.1.4
Version of test2:    2:0.1.3-dev
Version of manifest: current-2-g12f9080
```

可以用 `repo manifest` 命令来查看清单库。

```
$ repo manifest -o -
<?xml version="1.0" encoding="UTF-8"?>
<manifest>
  <remote fetch="git://github.com/ossxp-com/" name="github"/>

  <default remote="github" revision="refs/heads/test-0.1"/>

  <project name="test1" path="test/test1">
    <copyfile dest="Makefile" src="root.mk"/>
  </project>
  <project name="test2" path="test/test2"/>
</manifest>
```

仔细看上面的清单文件，可以注意到默认的版本指向到 `refs/heads/test-0.1` 引用所指向的分支 `test-0.1` 。

如果在子项目中修改、提交，然后使用 repo push 会将改动推送到远程版本库的 test-0.1 分支中。

8. 切换到清单库里程碑版本

执行如下命令可以查看清单库包含的里程碑版本：

```
$ git ls-remote --tags git://github.com/ossxp-com/manifest.git
43e5783a58b46e97270785aa967f09046734c6ab        refs/tags/current
3a6a6da36840e716a14d52252e7b40e6ba6cbdea        refs/tags/current^{}
4735d32613eb50a6c3472cc8087ebf79cc46e0c0        refs/tags/v0.1
fb1a1b7302a893092ce8b356e83170eee5863f43        refs/tags/v0.1^{}
b23884d9964660c8dd34b343151aaf968a744400        refs/tags/v0.1.1
9c4c287069e29d21502472acac34f28896d7b5cc        refs/tags/v0.1.1^{}
127d9789cd4312ed279a7fa683c43eec73d2b28b        refs/tags/v0.1.2
47aaa83866f6d910a118a9a19c2ac3a2a5819b3e        refs/tags/v0.1.2^{}
af3abb7ed0a9ef7063e9d814510c527287c92ef6        refs/tags/v0.1.3
99c69bcfd7e2e7737cc62a7d95f39c6b9ffaf31a        refs/tags/v0.1.3^{}
```

可以从任意里程碑版本的清单库初始化整个项目。

```
$ mkdir v0.1.2
$ cd v0.1.2
$ repo init -u git://github.com/ossxp-com/manifest.git -b refs/tags/v0.1.2
$ repo sync
```

当子项目代码全部同步完成后执行 `make` 命令。可以看到各个子项目的版本及清单库的版本不同于之前的输出。

```
$ make
Version of test1:    1:0.1.2
Version of test2:    2:0.1.2
Version of manifest: v0.1.2
```