Repo 介绍

目录

25 June 2015
阅读本文之前，需要对 git 有一定的了解。

1. 概要

`repo` 是 Android 为了方便管理多个 git 库而开发的 Python 脚本。repo 的出现，并非为了取代 git，而是为了让 Android 开发者更为有效的利用 git。

Android 源码包含数百个 git 库，仅仅是下载这么多 git 库就是一项繁重的任务，所以在下载源码时，Android 就引入了 repo。 Android 官方推荐下载 repo 的方法是通过 Linux curl 命令，下载完后，为 repo 脚本添加可执行权限：

```
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo

$ chmod a+x ~/bin/repo
```

由于国内 Google 访问受限，所以上述命令不一定能下载成功。其实，我们现在可以从很多第三方渠道找到 repo 脚本，只需要取下来，确保 repo 可以正确执行即可。

---

2. 工作原理

repo 需要关注当前 git 库的数量、名称、路径，才能对这些 git 库进行操作。通过集中维护所有 git 库的清单，repo 可以方便的从清单中获取 git 库的信息。 这份清单会随着版本演进升级而产生变化，同时也一些本地的修改定制需求，所以，repo 是通过一个 git 库来管理项目的清单文件的，这个 git 库名字叫 `manifests`。

当打开 repo 这个可执行的 python 脚本后，发现代码量并不大(不超过 1000 行)，难道仅这一个脚本就完成了 AOSP 数百个 git 库的管理吗？并非如此。 repo 是一系列脚本的集合，这些脚本也是通过 git 库来维护的，这个 git 库名字叫 `repo`。

在客户端使用 repo 初始化一个项目时，就会从远程把 manifests 和 repo 这两个 git 库拷贝到本地，但这对于 Android 开发人员来说，又是近乎无形的(一般通过文件管理器，是无法看到这两个 git 库的)。 repo 将自动化的管理信息都隐藏根目录的**.repo** 子目录中。

2.1 项目清单库(.repo/manifests)

AOSP 项目清单 git 库下，只有一个文件 default.xml，是一个标准的 XML，描述了当前 repo 管理的所有信息。 AOSP 的 default.xml 的文件内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>

<manifest>
```

```
  <remote  name="aosp"
           fetch=".."
           review="https://android-review.googlesource.com/" />
  <default revision="master"
           remote="aosp"
           sync-j="4" />
  <project path="build" name="platform/build" groups="pdk,tradefed" >
    <copyfile src="core/root.mk" dest="Makefile" />
  </project>
  <project path="abi/cpp" name="platform/abi/cpp" groups="pdk" />
  <project path="art" name="platform/art" groups="pdk" />
  ...
  <project path="tools/studio/translation" name="platform/tools/studio/translation"
groups="notdefault,tools" />
  <project path="tools/swt" name="platform/tools/swt" groups="notdefault,tools" />
</manifest>
```

- **<remote>**：描述了远程仓库的基本信息。name 描述的是一个远程仓库的名称，通常我们看到的命名是 origin;fetch 用作项目名称的前缘，在构造项目仓库远程地址时使用到;review 描述的是用作 code review 的 server 地址
- **<default>**：default 标签的定义的属性，将作为**<project>**标签的默认属性，在**<project>**标签中，也可以重写这些属性。属性 revision 表示当前的版本，也就是我们俗称的分支;属性 remote 描述的是默认使用的远程仓库名称，即**<remote>**标签中 name 的属性值;属性 sync-j 表示在同步远程代码时，并发的任务数量，配置高的机器可以将这个值调大
- **<project>**：每一个 repo 管理的 git 库，就是对应到一个**<project>**标签，path 描述的是项目相对于远程仓库 URL 的路径，同时将作为对应的 git 库在本地代码的路径;name 用于定义项目名称，命名方式采用的是整个项目 URL 的相对地址。 譬如，AOSP 项目的 URL 为 https://android.googlesource.com/，命名为 **platform/build** 的 git 库，访问的 URL 就是 https://android.googlesource.com/platform/build

如果需要新增或替换一些 git 库，可以通过修改 default.xml 来实现，repo 会根据配置信息，自动化管理。但直接对 default.xml 的定制，可能会导致下一次更新项目清单时，与远程 default.xml 发生冲突。 因此，repo 提供了一个种更为灵活的定制方式

`local_manifests`:所有的定制是遵循 default.xml 规范的，文件名可以自定义，譬如

local_manifest.xml, another_local_manifest.xml 等， 将定制的 XML 放在新建

的.repo/local_manifests 子目录即可。repo 会遍历.repo/local_manifests 目录下的所有 *.xml 文件，最终与 default.xml 合并成一个总的项目清单文件 manifest.xml。

local_manifests 的修改示例如下：

```
$ ls .repo/local_manifests

local_manifest.xml

another_local_manifest.xml


$ cat .repo/local_manifests/local_manifest.xml

<?xml version="1.0" encoding="UTF-8"?>

<manifest>

    <project path="manifest" name="tools/manifest" />

    <project path="platform-manifest" name="platform/manifest" />

</manifest>
```

2.2 repo 脚本库(.repo/repo)

repo 对 git 命令进行了封装，提供了一套 repo 的命令集(包括 init, sync 等)，所有 repo 管理的自动化实现也都包含在这个 git 库中。 在第一次初始化的时候，repo 会从远程把这个 git 库下载到本地。

2.3 仓库目录和工作目录

仓库目录保存的是历史信息和修改记录，工作目录保存的是当前版本的信息。一般来说，一个项目的 Git 仓库目录（默认为.git 目录）是位于工作目录下面的，但是 Git 支持将一个项目的 Git 仓库目录和工作目录分开来存放。 对于 repo 管理而言，既有分开存放，也有位于工作目录存放的:

- **manifests**： 仓库目录有两份拷贝，一份位于工作目录(.repo/manifests)的.git 目录下，另一份独立存放于.repo/manifests.git
- **repo**：仓库目录位于工作目录(.repo/repo)的.git 目录下
- **project**：所有被管理 git 库的仓库目录都是分开存放的，位于.repo/projects 目录下。同时，也会保留工作目录的.git，但里面所有的文件都是到.repo 的链接。这样，即做到了分开存放，也兼容了在工作目录下的所有 git 命令。

既然.repo 目录下保存了项目的所有信息，所有要拷贝一个项目时，只是需要拷贝这个目录就可以了。repo 支持从本地已有的.repo 中恢复原有的项目。

3. 使用介绍

repo 命令的使用格式如下所示：

```
$ repo <COMMAND> <OPTIONS>
```

可选的的有：help、init、sync、upload、diff、download、forall、prune、start、status，每一个命令都有实际的使用场景， 下面我们先对这些命令做一个简要的介绍：

3.1 init

```
$ repo init -u <URL> [<OPTIONS>]
```

- ***-u***：指定 manifests 这个远程 git 库的 URL，manifests 库是整个项目的清单。默认情况，这个 git 库只包含了 default.xml 一个文件，其内容可以参见 Android 的样本
- ***-m， –manifest-name***：指定所需要的 manifests 库中的清单文件。默认情况下，会使用 maniftests/default.xml
- ***-b， –manifest-branch***：指定 manifest.xml 文件中的一个版本，，也就是俗称的“分支”

运行该命令后，会在当前目录下新建一个.repo 子目录：

```
.repo
├── manifests       # 一个 git 库，包含 default.xml 文件，描述 repo 所管理的 git 库的信息
├── manifests.git   # manifest 这个 git 库的实体，manifest/.git 的所有文件都会链接到该目录
├── manifest.xml    # manifests/default.xml 的一个软链接
└── repo            # 一个 git 库，包含 repo 运行的所有脚本
```

这些本地的目录是如何生成的呢？执行 repo 命令时，可以通过`--trace`参数，来看实际发生了什么。

```
$ repo --trace init -u $URL -b $BRANCH -m $MANIFEST

--------------------

mkdir .repo; cd .repo

git clone --bare $URL manifests.git

git clone https://android.googlesource.com/tools/repo

mkdir -p manifests/.git; cd manifests/.git
```

```
for i in ../../manifests.git/*; do ln -s $1 .; done

cd ..

git checkout $BRANCH -- .

cd ..

ln -s manifests/$MANIFEST manifest.xml
```

首先，在当前目录下创建.repo 子目录，后续所有的操作都在.repo 子目录下完成;

然后，clone 了两个 git 库，其中一个是***-u*** 参数指定的 manifests，本地 git 库的名称是 manifest.git;另一个是默认的 repo，后面我们会看到这个 URL 也可以通过参数来指定;

接着，创建了 manifest/.git 目录，里面的所有文件都是到 manifests.git 这个目录的链接，这个是为了方便对 manifests 目录执行 git 命令，紧接着，就会将 manifest 切换到***-b*** 参数指定的分支;

最后，在.repo 目录下，创建了一个软链接，链接到***-m*** 参数制定的清单文件，默认情况是 manifests/default.xml。

这样，就完成了一个多 git 库的初始化，之后，就可以执行其他的 repo 命令了。

我们还介绍几个不常用的参数，在国内下载 Android 源码时，会用到：

- ***–repo-url***：指定远程 repo 库的 URL，默认情况是 https://android.googlesource.com/tools/repo，但国内访问 Google 受限，会导致这个库无法下载，从而导致 repo init 失败，所以可以通过该参数指定一个访问不受限的 repo 地址
- ***–repo-branch***：同 manifest 这个 git 库一样，repo 这个 git 库也是有版本差异的，可以通过该参数来指定下载 repo 这个远程 git 库的特定分支
- ***–no-repo-verify***：在下载 repo 库时，会对 repo 的源码进行检查。通过***–repo-url*** 指定第三方 repo 库时，可能会导致检查不通过，所以可以配套使用该参数，强制不进行检查

3.2 sync

```
$ repo sync [PROJECT_LIST]
```

下载远程代码，并将本地代码更新到最新，这个过程称为“同步”。如果不使用任何参数，那么会对所有 repo 管理的进行同步操作;也可以 PROJECT_LIST 参数，指定若干要同步的 PROJECT。 根据本地 git 库代码不同，同步操作会有不同的行为：

- 当本地的 git 库是第一次触发同步操作时，那么，该命令等价于 `git clone`，会将远程 git 库直接拷贝到本地

- 当本地已经触发过同步操作时，那么，该命令等价于 `git remote update && git rebase origin/<BRANCH>`，<BRANCH>就是当前与本地分支所关联的远程分支 代码合并可能会产生冲突，当冲突出现时，只需要解决完冲突，然后执行 `git rebase --continue` 即可。

当 sync 命令正确执行完毕后，本地代码就同远程代码保持一致了。在一些场景下，我们会用到 sync 命令的一些参数：

- ***-j***：开启多线程同步操作，这会加快 sync 命令的执行速度。默认情况下，使用 4 个线程并发进行 sync
- ***-c, –current-branch***：只同步指定的远程分支。默认情况下，sync 会同步所有的远程分支，当远程分支比较多的时候，下载的代码量就大。使用该参数，可以缩减下载时间，节省本地磁盘空间
- ***-d, –detach***：脱离当前的本地分支，切换到 manifest.xml 中设定的分支。在实际操作中，这个参数很有用，当我们第一次 sync 完代码后，往往会切换到 dev 分支进行开发。如果不带该参数使用 sync， 则会触发本地的 dev 分支与 manifest 设定的远程分支进行合并，这会很可能会导致 sync 失败
- ***-f, –force-broken***：当有 git 库 sync 失败了，不中断整个同步操作，继续同步其他的 git 库
- ***–no-clone-bundle***：在向服务器发起请求时，为了做到尽快的响应速度，会用到内容分发网络(CDN, Content Delivery Network)。同步操作也会通过 CDN 与就近的服务器建立连接， 使用 HTTP/HTTPS 的$URL/clone.bundle 来初始化本地的 git 库，clone.bundle 实际上是远程 git 库的镜像，通过 HTTP 直接下载，这会更好的利用网络带宽，加快下载速度。 当服务器不能正常响应下载$URL/clone.bundle，但 git 又能正常工作时，可以通过该参数，配置不下载$URL/clone.bundle，而是直接通过 git 下载远程 git 库

3.3 upload

```
$ repo upload [PROJECT_LIST]
```

从字面意思理解，upload 就是要上传，将本地的代码上传到远程服务器。upload 命令首先会找出本地分支从上一次同步操作以来发生的改动，然后会将这些改动生成 Patch 文件，上传至 *Gerrit* 服务器。 如果没有指定 PROJECT_LIST，那么 upload 会找出所有 git 库的改动;如果某个 git 库有多个分支，upload 会提供一个交互界面，提示选择其中若干个分支进行上传操作。

upload 并不会直接将改动合并后远程的 git 库，而是需要先得到 Reviewer 批准。Reviewer 查看改动内容、决定是否批准合入代码的操作，都是通过 Gerrit 完成。 Gerrit

服务器的地址是在 manifests 中指定的：打开.repo/manifest.xml，<remote>这个 XML TAG 中的 review 属性值就是 Review 服务器的 URL:

Gerrit 的实现机制不是本文讨论的内容，但有几个与 Gerrit 相关的概念，是需要代码提交人员了解的：

- ***Reviewer***：代码审阅人员可以是多个，是需要人为指定的。Gerrit 提供网页的操作，可以填选 Reviewer。当有多个 git 库的改动提交时，为了避免在网页上频繁的填选 Reviewer 这种重复劳动， upload 提供了***–re, –reviewer*** 参数，在命令行一次性指定 Reviewer
- ***Commit-ID***：git 为了标识每个提交，引入了 Commit-ID，是一个 SHA-1 值，针对当次提交内容的一个 Checksum，可以用于验证提交内容的完整性
- ***Change-ID***：Gerrit 针对每一个 Review 任务，引入了一个 Change-ID，每一个提交上传到 Gerrit，都会对应到一个 Change-ID， 为了区分于 Commit-ID，Gerrit 设定 Change-ID 都是以大写字母 **“I”** 打头的。 Change-ID 与 Commit-ID 并非一一对应的，每一个 Commit-ID 都会关联到一个 Change-ID，但 Change-ID 可以关联到多个 Commit-ID
- ***Patch-Set***：当前需要 Review 的改动内容。一个 Change-ID 关联多个 Commit-ID，就是通过 Patch-Set 来表现的，当通过 `git commit --amend` 命令修正上一次的提交并上传时， Commit-ID 已经发生了变化，但仍可以保持 Change-ID 不变，这样,在 Gerrit 原来的 Review 任务下，就会出现新的 Patch-Set。修正多少次，就会出现多少个 Patch-Set， 可以理解，只有最后一次修正才是我们想要的结果，所以，在所有的 Patch-Set 中，只有最新的一个是真正有用的，能够合并的。

3.4 download

```
$ repo download <TARGET> <CHANGE>
```

upload 是把改动内容提交到 Gerrit，download 是从 Gerrit 下载改动。与 upload 一样，download 命令也是配合 Gerrit 使用的。

- **<TARGET>**：指定要下载的 PROJECT，譬如 *platform/frameworks/base*, *platform/packages/apps/Mms*
- **<CHANGE>**：指定要下载的改动内容。这个值不是 Commit-ID，也不是 Change-ID，而是一个 Review 任务 URL 的最后几位数字。 譬如，AOSP 的一个 Review 任务 https://android-review.googlesource.com/#/c/23823/，其中 **23823** 就是<CHANGE>。

3.5 forall

```
$ repo forall [PROJECT_LIST] -c <COMMAND>
```

对指定的 git 库执行**-c** 参数制定的命令序列。在管理多个 git 库时，这是一条非常实用的命令。PROJECT_LIST 是以空格区分的，譬如：

```
$ repo forall frameworks/base packages/apps/Mms -c "git status"
```

表示对 *platform/frameworks/base* 和 *platform/packages/apps/Mms* 同时执行 `git status` 命令。 如果没有指定 PROJECT_LIST，那么，会对 repo 管理的所有 git 库都同时执行命令。

该命令的还有一些其他参数：

- **-r, –regex**： 通过指定一个正则表达式，只有匹配的 PROJECT，才会执行指定的命令
- **-p**：输出结果中，打印 PROJECT 的名称

3.6 prune

```
$ repo prune [<PROJECT_LIST>]
```

删除指定 PROJECT 中，已经合并的分支。当在开发分支上代码已经合并到主干分支后，使用该命令就可以删除这个开发分支。

随着时间的演进，开发分支会越来越多，在多人开发同一个 git 库，多开发分支的情况会愈发明显，假设当前 git 库有如下分支：

```
* master

  dev_feature1_201501   # 已经合并到 master

  dev_feature2_201502   # 已经合并到 master

  dev_feature3_201503   # 正在开发中，还有改动记录没有合并到 master
```

那么，针对该 git 库使用 prune 命令，会删除 dev_feature1_201501 和 dev_feature2_201502。

定义删除无用的分支，能够提交团队的开发和管理效率。prune 就是删除无用分支的”杀手锏“。

3.7 start

```
$ repo start <BRANCH_NAME> [<PROJECT_LIST>]
```

在指定的 PROJECT 的上，切换到<BRANCH_NAME>指定的分支。可以使用**–all** 参数对所有的 PROJECT 都执行分支切换操作。 该命令实际上是对 `git checkout` 命令的封装，<BRANCH_NAME>是自定义的，它将追踪 manifest 中指定的分支名。

当第一次 sync 完代码后，可以通过 start 命令将 git 库切换到开发分支，避免在匿名分支上工作导致丢失改动内容的情况。

3.8 status

```
$ repo status [<PROJECT_LIST>]
```

status 用于查看多个 git 库的状态。实际上，是对 `git status` 命令的封装。

---

4. 使用实践

Android 推荐的开发流程是：

1. **repo init** 初始化工程，指定待下载的分支
2. **repo sync** 下载代码
3. **repo start** 将本地 git 库切换到开发分支(TOPIC BRANCH)
4. 在本地进行修改，验证后，提交到本地
5. **repo upload** 上传到服务器，等待 review

在实际使用过程中，我们会用到 repo 的一些什么子命令和参数呢？哪些参数有助于提高开发效率呢？下面我们以一些实际场景为例展开说明。

4.1 对项目清单文件进行定制

通过 `local_manifest` 机制，能够避免了直接修改 default.xml，不会造成下次同步远程清单文件的冲突。

CyanogenMod(CM)适配了上百款机型，不同机型所涉及到的 git 库很可能是有差异的。以 CM 对清单文件的定制为例，通过新增 local_manifest.xml，内容如下：

```
<manifest>

    <!-- add github as a remote source -->

    <remote name="github" fetch="git://github.com" />
```

```
    <!-- remove aosp standard projects and replace with cyanogenmod versions -->
    <remove-project name="platform/bootable/recovery" />
    <remove-project name="platform/external/yaffs2" />
    <remove-project name="platform/external/zlib" />
    <project path="bootable/recovery" name="CyanogenMod/android_bootable_recovery" r
emote="github" revision="cm-10.1" />
    <project path="external/yaffs2" name="CyanogenMod/android_external_yaffs2" remot
e="github" revision="cm-10.1" />
    <project path="external/zlib" name="CyanogenMod/android_external_zlib" remote="g
ithub" revision="cm-10.1" />

    <!-- add busybox from the cyanogenmod repository -->
    <project path="external/busybox" name="CyanogenMod/android_external_busybox" rem
ote="github" revision="cm-10.1" />

</manifest>
```

local_manifest.xml 会与已有的 default.xml 融合成一个项目清单文件 manifest.xml，实现了对一些 git 库的替换和新增。 可以通过以下命令导出当前的清单文件，最终 snapshot.xml 就是融合后的版本：

```
$ repo manifest -o snapshot.xml -r
```

在编译之前，保存整个项目的清单，有助于问题的回溯。当项目的 git 库发生变更，需要回退到上一个版本进行验证的时候，只需要重新基于 snapshot.xml 初始化上一个版本即可：

```
$ cp snapshot.xml .repo/manifests/
$ repo init -m snapshot.xml      # -m 参数表示自定义 manifest
$ repo sync -d                   # -d 参数表示从当前分支脱离，切换到 manifest 中定义的分支
```

4.2 解决无法下载 Android 源码

在 repo init 的时候，会从远程下载 manifests 和 repo 这两个 git 库，默认情况下，这两个 git 库的地址都是写死在 repo 这个 python 脚本里面的。对于 AOSP 而言，这两个 git 库的地址显然是 google 提供的。 但由于 google 访问受限的缘故，会导致 init 时，无法下载

manifests 和 repo。这时候，可以使用 **init** 的**-u** 和**–repo-url** 参数，自定义这两个库的地址，辅以**–no-repo-verify** 来绕过代码检查。

```
$ repo init --repo-url [PATH/TO/REPO] -u [PATH/TO/MANIFEST] -b [BRANCH] --no-repo-verify

$ repo sync
```

4.3 更快更省的下载远程代码

repo 默认会同步 git 库的所有远程分支的代码，但实际开发过程中，用到的分支是有限的。使用 **sync** 的**-c** 参数，可以只下载 manifest 中设定的分支，这会节省代码下载时间以及本地的磁盘空间：

```
$ repo sync -c
```

如果实际开发过程中，需要用到另外一个分支，而又不想被其他分支干扰，可以在已有的工程根目录下，使用如下命令：

```
$ repo manifest -o snapshot.xml -r

$ repo init -u [PATH/TO/MANIFEST] -b [ANOTHER_BRANCH]

$ repo sync -c -d
```

以上命令序列，相当更新了 manifest，而且仅仅只下载 ANOTHER_BRANCH 的代码，这样本地只保存了两个分支的代码。利用保存的 snapshot.xml，还能将所有 git 库方便的切换回原来的分支。

如果本地已经有一份 Android 源码，假设路径为~/android-exsit，想要下载另一份新的 Android 源码，通过**–reference** 参数，在数分钟以内，就能将代码下载完毕：

```
$ mkdir ~/android-new && cd ~/android-new

$ repo init --reference=~/android-exsit -u [PATH/TO/MANIFEST] -b [BRANCH]

$ repo sync -c
```

4.4 避免在匿名分支上工作

在 sync 完代码后，所有 git 库默认都是在一个匿名分支上(no branch)，很容易会由于误操作导致丢失代码修改。可以使用如下命令将所有的 git 库切换到开发分支：

```
$ repo start BRANCH --all
```

4.5 使用 upload 提交代码

开发人员可能同时在多个 git 库，甚至多个分支上，同时进行修改，针对每个 git 库单独提交代码是繁琐的。可以使用如下命令，一并提交所有的修改：

```
$ repo upload
```

不用担心会漏提交或者误提交，upload 会提供一个交互界面，开发人员只需要选择需要提交的 git 库和分支即可。

如果需要省去 Gerrit 上填写 reviewer 的操作，可以使用**–reviewer** 参数指定 Reviewer 的邮箱地址：

```
$ repo upload --reviewer="R.E.viewer@google.com"
```

4.6 定期删除已经合并的开发分支

Git 鼓励在修复 Bug 或者开发新的 Feature 时，都创建一个新的分支。创建 Git 分支的代价是很小的，而且速度很快，因此，不用担心创建 Git 分支是一件不讨好的事情，而应该尽可能多地使用分支。

随着时间的演进，开发分支会越来越多，而一些已经合并到主干的开发分支是没有存在价值的，可以通过 prune 命令定期删除无用的开发分支：

```
$ repo prune [PROJECT_LIST]
```

4.7 同时操作多个 git 库

对于部分开发人员而言，同时操作多个 git 库是常态，如果针对每个 git 库的操作命令都是相同的，那么可以使用如下命令一次性完成所有操作:

```
$ repo forall -c "git branch | grep tmp | xargs git branch -D; git branch"
```

参数**-c** 指定的命令序列可以很复杂，多条命令只需要用“;”间隔。