如意通科技产品支持 - 说说持续集成(CI)的那点事

这里忆苦思甜，来说说我们的CI是如何演进的，想到哪写到哪吧，如需转载请注明出处和链接。

Hero说应该把SCM单独弄一篇，所以我单独起了一篇， {{article(21)}}

基于脚本的CI

最原始的CI，就是写个编译脚本，想办法自动执行它，基本两个办法

1. 设置一个cron计划任务，每天执行一遍，这就daily build了

2. 最早SVN的时候，可以在里面编写hook
   script，它的前置hook一般用来提醒程序员提交代码的时候必须带上多少个字的注释，后置hook就可以用来跑脚本了

但这样感觉很low是不是，悄悄就把CI这活干了，实际在项目型的公司里，这样差不多就够了

GitLab CI

GitLab的CI，最开始和GitLab的代码管理是两个独立的服务，然后以一种非常别扭的方式集成在一起来用，往事不堪回首，后来的人，你们是幸福的

现在GitLab的CI变成了内置功能了，但真要用起来，还得装 runner
，也就是实际编译代码的那台服务器(可能是独立服务器，可以是虚拟机，也可是Docker容器)

GitLab的CI原理，就是定义了一个固定名字的文件 .gitlab-ci.yml
，如果存在这个文件，那么每次提交代码的时候GitLab服务器就会解析这个脚本里的语法，让它去指定的runner中运行一些指令来实现编译，测试，部署等工作，实际这玩意和SVN里面的后置hook是一个意思，现在封装一下搞得有头有脸一点，画个图意思一下

{{mermaid(GitLab CI 原理)
sequenceDiagram
participant 用户
participant GitLab服务器
participant RUNNER服务器
用户->>GitLab服务器:提交代码
GitLab服务器-->>GitLab服务器: 更新代码仓库
opt 存在 .gitlab-ci.yml
GitLab服务器-->>GitLab服务器: 解析.gitlab-ci.yml获得指令集合
GitLab服务器->>RUNNER服务器:请求执行指令集合
end
RUNNER服务器-->>RUNNER服务器:执行指令集合
RUNNER服务器-->>GitLab服务器:返回执行结果
GitLab服务器-->>GitLab服务器: 更新Web控制台的显示
Note over GitLab服务器: 仓库首页的小图标
Note over GitLab服务器: passed 或 failed
opt 仓库配置了邮件提示
GitLab服务器-->> 用户:配置触发发送邮件
Note left of GitLab服务器: 不一定出错才触发
Note left of GitLab服务器: 取决于仓库配置
end
}}

GitLab CI Runner

GitLab Runner
主要有两类，一类是单机跑一个Runner服务，比如Windows，MacOS，一类是一个Docker主机拖多个Docker容器，下面随便举两个例子

Windows单机单Runner

参考 https://docs.gitlab.com/runner/install/windows.html ,下载一个Runner客户端

```
wget    https://gitlab-ci-multi-runner-downloads.s3.amazonaws.com/latest/binaries/gitlab-ci-multi-runner-windows-amd64.exe
```

把此Runner注册到GitLab服务器上

```
cd C:\Multi-Runner
gitlab-ci-multi-runner register

Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com )
https://gitlab.com
Please enter the gitlab-ci token for this runner
xxx
Please enter the gitlab-ci description for this runner
my-runner
INFO[0034] fcf5c619 Registering runner... succeeded
Please enter the executor: shell, docker, docker-ssh, ssh?
shell
INFO[0037] Runner registered successfully. Feel free to start it, but if it's
running already the config should be automatically reloaded!
```

注意：

1. “ Please enter the gitlab-ci token for this runner ” 这里的 "token" 要去到GitLab服务器上找
2. “ Please enter the executor: shell, docker, docker-ssh, ssh? ” 这里要写shell，单机Runner这里都写shell

另外，最好把Runner安装成Windows服务，这里需要运行服务的帐号和密码，Windows本机的有权限启动服务并执行CI动作的用户

```
gitlab-ci-multi-runner install --user ENTER-YOUR-USERNAME --password ENTER-YOUR-PASSWORD
gitlab-ci-multi-runner start
```

Debian单机多Runner(Docker)

参考 https://docs.gitlab.com/runner/install/linux-repository.html ,以Debian/Ubuntu为例

先安装Docker

```
curl -sSL https://get.docker.com/ | sh
```

然后再安装Multi Runner服务

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-ci-multi-runner/script.deb.sh | sudo bash
sudo apt-get install gitlab-ci-multi-runner
```

如果以后要升级Multi Runner，执行

```
sudo apt-get update
sudo apt-get install gitlab-ci-multi-runner
```

安装完后会有一个 gitlab-ci-multi-runner 服务在跑

参考 https://docs.gitlab.com/ce/ci/docker/using_docker_images.html ， 来使用Multi Runner

把Runner注册到GitLab服务器

```
gitlab-ci-multi-runner register \
  --url "https://gitlab.com/" \
  --registration-token "PROJECT_REGISTRATION_TOKEN" \
  --description "docker-ruby-2.1" \
  --executor "docker" \
  --docker-image ruby:2.1 \
  --docker-postgres latest \
  --docker-mysql latest
```

注意：

1. --registration-token， 这里的 "token" 要去到GitLab服务器上找
2. --docker-image， 这里表示注册的Runner要运行的镜像，编译工作会在这里进行
3. --docker-postgres latest ,
   这里表示Runner镜像在编译时，同时启动的另一个镜像，并且Runner所在镜像会连接到本镜像并使用这上面的postgres服务
4. --docker-mysql latest ,
   这里表示在Runner镜像在编译时，同时启动的另一个镜像，并且Runner所在镜像会连接到本镜像并使用这上面的mysql服务

以下检查此服务以及其中哪些runner在运行

```
sudo gitlab-ci-multi-runner verify
```

```
INFO[0000] adfdf3c5b Veryfing runner... is alive
```

接下来，可以再次注册新的Runner，这就是所谓的Multi Runner，可以注册任意多个Runner

之前安装的gitlab-ci-multi-runner
服务会管理这些Runner，这些Runner平时并不实际运行，只有GitLab服务器触发编译动作时才会启动相应Runner的相关Docker容器，完成后这些Runner容器又会处于停用状态，这就意味着一台Docker主机可以同时跑很多Runner，只要同时运行中的Docker容器没有超过整台Docker主机的最大负载即可

这里也画个时序图稍微演示一下Multi Runner的运作过程吧

{{mermaid(MultiRunner服务原理)
sequenceDiagram
participant 用户
participant GitLab服务器
participant MultiRunner服务器
participant ruby容器A
participant postgres容器A
participant mysql容器A
MultiRunner服务器->>GitLab服务器: 注册一个Runner
Note left of MultiRunner服务器: RunnerA
GitLab服务器-->>MultiRunner服务器: 返回成功
MultiRunner服务器->>ruby容器A: 创建编译容器
activate ruby容器A
Note right of MultiRunner服务器: ruby:2.1镜像

ruby容器A-->>MultiRunner服务器: 返回成功
deactivate ruby容器A
MultiRunner服务器->>postgres容器A: 创建服务容器
activate postgres容器A
Note right of MultiRunner服务器: postgres:latest镜像
postgres容器A-->>MultiRunner服务器: 返回成功
deactivate postgres容器A
MultiRunner服务器->>mysql容器A: 创建服务容器
activate mysql容器A
Note right of MultiRunner服务器: mysql:latest镜像
mysql容器A-->>MultiRunner服务器: 返回成功
deactivate mysql容器A
用户->>GitLab服务器:提交代码

GitLab服务器-->>GitLab服务器: 更新代码仓库
GitLab服务器-->>GitLab服务器: 解析.gitlab-ci.yml
GitLab服务器->>MultiRunner服务器:请求启动postgres服务
MultiRunner服务器->>postgres容器A: 启动postgres容器A
activate postgres容器A
postgres容器A-->>MultiRunner服务器: 返回功
GitLab服务器->>MultiRunner服务器:请求启动mysql服务
MultiRunner服务器->>mysql容器A: 启动mysql容器A
activate mysql容器A
mysql容器A-->>MultiRunner服务器: 返回成功
GitLab服务器->>MultiRunner服务器:请求执行指令
MultiRunner服务器->>ruby容器A:启动ruby容器A
activate ruby容器A
ruby容器A-->>MultiRunner服务器: 返回成功
MultiRunner服务器->>ruby容器A:执行指令
ruby容器A->>postgres容器A:调用postgres服务
postgres容器A-->>MultiRunner服务器: 返回成功
ruby容器A->>mysql容器A:调用mysql服务
mysql容器A-->>MultiRunner服务器: 返回成功
ruby容器A-->>MultiRunner服务器: 返回成功
MultiRunner服务器-->> GitLab服务器: 返回成功
deactivate postgres容器A
deactivate mysql容器A
deactivate ruby容器A
}}

GitLab CI 的使用

有了GitLab CI，确实我们方便很多啊，它的一些优点如下

1. 它是由GitLab的代码提交来触发的，这解决了自动化的问题
2. 它定义了一套脚本规则，可以通过脚本来决定自动化执行的内容
3. 它的控制台可以很方便地观察脚本执行过程和结果
4. 它提供通过webhook把执行结果传递给外部应用的能力
5. 和它配套使用的MultiRunner服务器非常节省资源，这很先进！！！

我们通常拿GitLadb CI
来做组件的自动编译和内部部署，比如Android内部的PDFView组件，它会自动编译并部署到内网的Nexus服务器上

但是它也有一些缺点，使得我们无法用它解决所有CI需求

1. .gitlan-ci.yml
   是在代码仓库中维护的，然而，如何编译和打包，很多时候是由测试或支持人员决定的，但是这些人并没有权限访问代码仓库
2. 有的组件之间存在依赖关系并需要同步更新，也就是说仓库A的代码更新了，那么仓库B也得重新编译，但是仓库B的代码并未更新，只能用webhook干点事情来驱动它，麻烦
3. 有时候我们需要自动编译+自动测试的联动，假如某服务程序(比如是Java程序)的测试脚本(比如Erlang脚本)存在于另一个仓库，如果Java程序更新了，需要触发的是Erlang脚本
4. 当我们编译定制版本的时候，最低限度我们也得从两个不同的仓库获取内容，进行混合处理之后才进行编译, GitLab CI无法跨仓库执行，只能把定制内容也部署到Nexus服务器
5. 当标准版仓库更新之后，定制化版本是否（每一个）都一定随之更新，以及是自动还是手动触发特定客户定制版本的更新，这都是不确定的，需要打包人员随时调整

所以，接下来，就该独立的CI系统Jenkins来帮助我们了

Jenkins CI

这是一个基于Java的独立CI软件，它可以跟市面上大部分的SCM工具（CVS,SVN,Git等）配合，可以通过手工或webhook以及计划任务等多种手段触发，还提供插件机制来扩展各种功能（有很多现成的第三方插件），一个普通的手工触发的Jenkins项目运作如下

{{mermaid(Jenkins CI简单示例)
sequenceDiagram
participant 用户
participant Jenkins服务器
participant Git仓库
用户->>Jenkins服务器:启动项目构建
Jenkins服务器-->>Jenkins服务器: 顺序执行项目指令

opt 存在 SCM配置
Jenkins服务器->>Git仓库: 请求代码
Git仓库-->>Jenkins服务器:返回成功
end
Jenkins服务器-->>Jenkins服务器:执行编译指令
Jenkins服务器-->>Jenkins服务器:Web控制台显示执行结果
opt 项目配置了提示
Jenkins服务器-->> 用户:配置触发发送执行结果提示
Note left of Jenkins服务器:可以是邮件
Note left of Jenkins服务器:也可以是XMPP等
end
}}

Jenkins也可以通过webhook和GitLab配合，使得代码更新可以自动触发Jenkins项目的构建，示例如下

{{mermaid(Webhook触发Jenkins CI示例)
sequenceDiagram
participant 用户
participant GitLab服务器
participant Jenkins服务器
用户->>GitLab服务器:提交代码
GitLab服务器-->>GitLab服务器: 更新代码仓库
opt 配置了webhook
GitLab服务器->>Jenkins服务器:post仓库更新情况
end
Jenkins服务器->>GitLab服务器:请求代码
GitLab服务器-->>Jenkins服务器:返回代码
Jenkins服务器-->>Jenkins服务器:完成编译
Jenkins服务器-->>Jenkins服务器:Web控制台显示执行结果
opt 项目配置了提示
Jenkins服务器-->> 用户:配置触发发送执行结果提示
Note left of Jenkins服务器:可以是邮件
Note left of Jenkins服务器:也可以是XMPP等
end
}}

安装和升级Jenkins

Jenkins要求最低JDK7，Debian 源里有现成的OpenJDK7(建议装这个)，也可以装Oracle JDK7

参考 https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins+on+Ubuntu 文档

```
wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key | apt-key add -
sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /etc/apt/sources.list.d/jenkins.list'
apt-get update
apt-get install jenkins
#如果安装困难，则使用tsocks调用代理
tsocks apt-get install jenkins
#升级jenkins
apt-get update
apt-get upgrade
```

启动jenkins

```
/etc/init.d/jenkins start
```

安装jenkins插件

到 系统管理--》插件管理--》可选插件--》Active Directory plugin，安装AD插件

到 系统管理--》插件管理--》可选插件--》GitLab Plugin，安装GitLab 插件插件

它同时安装了很多依赖的插件， 如下：

- Icon Shim Plugin
- Credentials Plugin
- SSH Credentials Plugin
- Git client plugin
- SCM API Plugin
- JUnit Plugin
- Matrix Project Plugin
- Mailer Plugin
- Git plugin
- GitLab Plugin
- Jabber notifier plugin

配置master节点让它构建项目

Jenkins主服务器除了提供控制台，安全，管理等服务，本身可以安装编译环境作为CI服务器节点

因为安装Jenkins的原因，所以master节点很自然地拥有了Java编译环境，不过它还需要安装Git客户端并配置连接GitLab服务器

安装git工具

```
apt-get install git
```

生成密钥用于拉取git仓库

生成本机运行Jenkins服务的用户的密钥

```
cd /var/lib/jenkins
su jenkins
ssh-keygen -t rsa
#一路回车按"y"即可
```

查看生成的密钥

```
cat ~/.ssh/id_rsa.pub
```

在GitLab的仓库中的 Settings --> Deploy Keys 中新增一个key，把公钥的内容完整拷贝进去

然后在本机上以jenkins用户的身份至少执行一次git操作，因为需要手工确认ssh远程登录到git仓库，这样know_hosts文件里缓存下git服务器的公钥，成功后就可把clone下来的东西删掉，仍然交给jenkins自己来更新工作区

```
cd /var/lib/jenkins
su jenkins
cd XXX
git clone git@192.168.0.XXX:test/test.git
正克隆到 'test'...
The authenticity of host '192.168.0.xxx (192.168.0.xxx)' can't be established.
```

ECDSA key fingerprint is eg:cd:25:89:63:vd:ah:ad:c2:6c:9f:e7:c3:20:3d:fa.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.xxx' (ECDSA) to the list of known hosts.
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
接收对象中: 100% (3/3), 完成.
检查连接... 完成。
rm -rf .git
rm -rf test

如果需要访问其他的SCM工具如svn，那么就要安装svn客户端，略

配置系统管理

配置AD集成

到 系统管理--》Configure Global Security，打开“ 启用安全 ”，选择“ Active Directory ”，设置如下

JNLP节点代理的TCP端口 指定端口 ： ？？？？？

Domain Name： ？？？？.local
Domain controller：？？？？.？？.？？.？？:3268
Bind DN： cn=？？,cn=users,dc=？？？,dc=local
Bind Password： ？？？？？？

Enable Slave → Master Access Control

添加用户 XXX,并赋予全部权限 ，作为超级管理员
添加用户 YYY。。。。。。 ，并赋予合适权限

配置邮件和XMPP消息提醒

到 系统管理--》系统设置，设置如下

Jenkins Location
系统管理员邮件地址： ？？？@rooyee.im

Jabber Notification
Jabber ID：？？？@rooyee.im
Password：？？？
Server：rooyee.im
Accept all SSL/TLS certificates：允许

邮件通知
SMTP服务器：？？？.？？？.？？？
使用SMTP认证：
用户名：？？？
密码：？？？

配置全局工具

到 系统管理--》Global Tool Configuration，以Maven为例，点击 "Maven安装"按钮，设置如下

注意：这个设置对从节点也是有效的

Maven Name : Default
MAVEN HOME: /usr/share/maven2

配置从节点

最简单而且通用的产生从节点的方式，就是从节点要通过jnlp连接到主节点上

不过在这之前，从节点所在服务器需要安装jdk7或jre7，
并从Jenkins从节点配置界面下载一个slave.jar包，这样就可以注册到jenkins主节点了

假如以Android用的从节点为例，还要安装git客户端以支持拉取Git仓库（同时需要在GitLab配置Deploy key，此处可参考主节点的类似配置），安装Android SDK环境用来编译，等等

到 系统管理--》管理节点--》新建节点，设置如下

名字：android_build
远程工作目录： c:\jenkins
标签：windows
启动方法： java web start

接下来从节点androi_build机器执行

```
java -jar slave.jar -jnlpUrl http://192.168.X.XXX:XX/computer/android_build/slave-agent.jnlp -secret ??????????????????????
```

从我们引入Jenkins的初衷可以看出，我们的单个产品，代码可能存放很多个Git仓库中，编译的时候需要涉及众多的组件，这些编译好的组件乃至我们最终的产品都需要一个可管理的地方来存放，这个地方，我们叫作 ***软件仓库***

接下来我们岔开一点话题，讲讲 Sonatype Nexus
软件仓库服务器，它即是一个私有软件仓库服务器，又可以作为代理和缓存服务器从公共的软件仓库获取内容

Sonatype Nexus

Nexus，一般大家认为这是一个Maven私服，方便Maven编译的时候从私服下载依赖包

实际它的作用远不至于此，它还支持npm仓库，nuget仓库和ruby仓库，这还是Nexus2，我们暂时还没空去对Nexus3一探究竟

安装Nexus2.X

这里只说说Debian/Ubuntu下的安装

```
#安装JDK
apt-get install -y openjdk-7-jdk
# 下载安装包,去到 https://www.sonatype.com/download-oss-sonatype 下载，似乎最新版是2.14.2-01
#解压
cd ~/
tar -xvzf nexus-2.14.2-01-bundle.tar.gz
#把安装目录软链接到系统目录
cd /usr/local && ln -s ~/nexus-2.14.2-01 nexus
#拷贝开机自启动文件
cp /usr/local/nexus/bin/nexus /etc/init.d
#修改开机自启动文件
sed -i "s/NEXUS_HOME=\"..\"/NEXUS_HOME=\"\/usr\/local\/nexus\"/g" /etc/init.d/nexus
sed -i "s/#RUN_AS_USER=/RUN_AS_USER=root/g" /etc/init.d/nexus
sed -i "s/#PIDDIR=\".\"/PIDDIR=\"\/var\/run\"/g" /etc/init.d/nexus
```

启动和停止nexus服务

```
/etc/init.d/nexus start
/etc/init.d/nexus stop
```

第一次启动nexus之后，它会自动产生一个 ~/nexus-work 目录，实际数据都在这个目录里

升级Nexus2.X

因为数据都在 ~/nexus-work 目录，所以升级只要把以前新的代码下载下来，重新做软链接即可

```
#  下载最新安装包,去到  https://www.sonatype.com/download-oss-sonatype
#解压
cd  ~/
tar  -xvzf  nexus-2.xx.x-0x-bundle.tar.gz
#把原来的软链接删掉
cd  /usr/local  &&  rm  -rf  nexus
#把新版的目录软链接到系统目录
cd  /usr/local  &&  ln  -s  ~/nexus-2.xx.x-0x  nexus
```

重新启动nexus即可

```
/etc/init.d/nexus  start
```

Nexus控制台设置

登录 Nexus控制台 http://localhost:8081 后,缺省管理员为 admin ， 密码为 admin123

帐号及安全设置

首先要修改admin密码, 修改邮箱
然后可以新增Role和用户，此处略

服务器设置

修改server的base url，如果放到外网，用到反向代理的时候需要设置这个

仓库设置

公共仓库设置

公共仓库就算公开的开源的那些仓库，当我们编译中首次引用公共仓库中的lib时，我们的Nexus私服就会去到互联网获取这些lib转发给我们，同时它会缓存到自己服务器上，这样下次我们的任何项目再次引用此lib时直接从私服的缓存获取

我们把https的url改成http，这样速度快一些，另外新增一些Nexus没有带的公共仓库（可以根据需要自己加）

- 把Central仓库的remote url地址从https改为http，，允许Download Remote Index
- 把Apache Snapshots的remote url地址从https改为http，然后加入到public仓库组，允许Download Remote Index
- 把Codehaus Snapshots加入到public仓库组，允许Download Remote Index
- 新增ibiblio.org仓库并加入到public仓库组，允许Download Remote Index
- 新增OSS Sonatype Snapshots仓库并加入到public仓库组，允许Download Remote Index
- 新增springsource仓库并加入到public仓库组，允许Download Remote Index

私有仓库设置

这些仓库是用来给公司项目自己进行部署和分发的，所以需要配置权限，以下以如意通科技的部分私有仓库为例

- 新增一个proxy仓库testsnapshots，把Publish URL设为false，这个用来测试的
- 新增一个proxy仓库android_snapshot，把Publish URL设为false，这个用来上传和下载如意通android客户端的组件的snapshot版
- 新增一个proxy仓库android_release，把Publish URL设为false，这个用来上传和下载如意通android客户端的组件的release版
- 新增一个仓库组midea，把android_release和android_snapshot加入其中，这个仓库组是一个容器，专门用来和美的集团协作开发用的，这样美的集团的开发人员只需要面对一个仓库url

Nexus使用

就以gradle的用法为例吧，以下这行命令就是gradle编译组件并上传到Nexus服务器

```
gradlew uploadArchives -PRELEASE_REPOSITORY_URL=http://???.???.??/repositories/android_release -PSNAPSHOT_REPOSITORY_URL=http://???.???.???/repositories/android_snapshot -PNEXUS_USERNAME=??? -PNEXUS_PASSWORD=???
```

RTP客户端CI实战

这里以一下我们的RTP客户端的Android版（代号Unicron）为例，讲讲CI的实操

SCM工具

我们们代码仓库使用的是在GitLab管理下的Git仓库，跟客户协作的时候，客户有用svn的，所以我印象中似乎有用过一个svn和git转换的工具，这个记不太清了，是题外话。

低频Android组件

所谓低频的Android组件，就是基本上你不太会去改它了，这通常是一些外来的开源或者商业组件

这种组件我们一般就用GitLab CI来解决,CI Runner需要预装Anroid编译环境

为了简化，下图只描述了happy path

{{mermaid(低频Android组件GitLab CI过程)
sequenceDiagram
participant 开发人员
participant GitLab服务器
participant Android_Build Runner
participant Nexus服务器
开发人员->>GitLab服务器:提交代码
GitLab服务器-->>GitLab服务器: 更新代码仓库
GitLab服务器-->>GitLab服务器: 解析.gitlab-ci.yml
GitLab服务器->>Android_Build Runner:请求执行指令集合
Android_Build Runner-->>Android_Build Runner:开始编译低频Android组件
Android_Build Runner->>Nexus服务器:请求依赖包
Nexus服务器-->>Android_Build Runner:返回依赖包
Android_Build Runner-->>Android_Build Runner:完成编译
Android_Build Runner->>Nexus服务器:上传编译好的组件
Nexus服务器-->>Android_Build Runner: 返回成功
Android_Build Runner-->>GitLab服务器: 返回成功
GitLab服务器-->>GitLab服务器: 更新Web控制台上的指示图标
GitLab服务器-->> 开发人员:返回成功
Note left of GitLab服务器:发送邮件
}}

低频非Android组件

所谓低频的非Android组件，就是说组件不是Android原生的，比如C++组件之类的，需要再封装成Android组件才可以用

这种组件我们仍然用GitLab CI来解决 , 但是CI Runner需要预装相应的环境

此处以我们的跨平台SIP客户端组件为例，它的Runner是一个MultiRunner服务下的一个Docker容器，
运行于CentOS，预装C和C++c编译器以及Android
NDK，另外它暂时无法纳入Nexus软件仓库管理，所以我们有一个“ 通用存储服务器 ”，这样编译好的组件可以通过ssh传到它上面

为了简化，下图只描述了happy path

{{mermaid(SIP客户端组件GitLab CI MultiRunner过程)
sequenceDiagram
participant 开发人员
participant GitLab服务器
participant MultiRunner服务器

participant SIP客户端专用容器
participant 通用存储服务器
开发人员->>GitLab服务器:提交代码
GitLab服务器-->>GitLab服务器: 更新代码仓库
GitLab服务器-->>GitLab服务器: 解析.gitlab-ci.yml
GitLab服务器->>MultiRunner服务器:请求执行指令集合
MultiRunner服务器->>SIP客户端专用容器:启动专用Runner容器
activate SIP客户端专用容器
SIP客户端专用容器-->>SIP客户端专用容器:编译SIP客户端组件
SIP客户端专用容器->>通用存储服务器:上传编译好的组件
通用存储服务器-->>SIP客户端专用容器: 返回成功
SIP客户端专用容器-->>MultiRunner服务器: 返回成功
deactivate SIP客户端专用容器
MultiRunner服务器-->>GitLab服务器: 返回成功
GitLab服务器-->>GitLab服务器: 更新Web控制台上的指示图标
GitLab服务器-->> 开发人员:返回成功
Note left of GitLab服务器:发送邮件
}}

然后，再封装成Android组件

{{mermaid(封装SIP客户端组件GitLab CI过程)
sequenceDiagram
participant 开发人员
participant GitLab服务器
participant Android_Build Runner
participant 通用存储服务器
participant Nexus服务器
开发人员->>GitLab服务器:提交代码
GitLab服务器-->>GitLab服务器: 更新代码仓库
GitLab服务器-->>GitLab服务器: 解析.gitlab-ci.yml
GitLab服务器->>Android_Build Runner:请求执行指令集合
Android_Build Runner-->>Android_Build Runner:开始编译SIP客户端Android组件
Android_Build Runner->>通用存储服务器:请求SIP客户端lib
通用存储服务器-->>Android_Build Runner:返回SIP客户端lib
Android_Build Runner->>Nexus服务器:请求其他依赖包
Nexus服务器-->>Android_Build Runner:返回其他依赖包
Android_Build Runner-->>Android_Build Runner:完成编译
Android_Build Runner->>Nexus服务器:上传编译好的组件
Nexus服务器-->>Android_Build Runner: 返回成功
Android_Build Runner-->>GitLab服务器: 返回成功
GitLab服务器-->>GitLab服务器: 更新Web控制台上的指示图标
GitLab服务器-->> 开发人员:返回成功
Note left of GitLab服务器:发送邮件
}}

高频Android组件

所谓高频的Android组件，就是可能每天都会改它，这通常是我们的核心功能组件

这种组件我们还是用GitLab CI来解决,CI Runner需要预装Anroid编译环境

高频组件的CI，我们的CI脚本可以区分开发分支和master分支，开发分支编译后上传到Nexus的snapshot仓库去用于测试，master分支编译成功后则上传到release仓库去用于发布

实际情况比下图的流程更为复杂，因为组件必须测试通过(而不仅仅是编译通过)才可以release，另外有的组件只有在产品整体编译之后经过交互测试才能确定通过，所以实际应用中我们倾向于在组件层面只上传snapshot版到Nexus，只有在需要和外部客户进行协作开发的时候才发布组件的release版出去给客户使用

为了简化，下图只描述了happy path

{{mermaid(高频Android组件GitLab CI过程)
sequenceDiagram
participant 开发人员
participant GitLab服务器
participant Android_Build Runner
participant Nexus服务器

开发人员->>GitLab服务器:提交代码
GitLab服务器-->>GitLab服务器: 更新代码仓库
GitLab服务器-->>GitLab服务器: 解析.gitlab-ci.yml
alt 非master分支更新代码
GitLab服务器->>Android_Build Runner:请求执行该段指令集合
Android_Build Runner-->>Android_Build Runner:开始编译本分支的snapshot版
Android_Build Runner->>Nexus服务器:请求依赖包
Nexus服务器-->>Android_Build Runner:返回依赖包
Android_Build Runner-->>Android_Build Runner:完成编译
Android_Build Runner-->>Android_Build Runner:自动化测试(如果有的话)
Android_Build Runner->>Nexus服务器:上传组件到snapshot仓库
Nexus服务器-->>Android_Build Runner: 返回成功
else master分支更新代码
GitLab服务器->>Android_Build Runner:请求执行该段指令
Android_Build Runner-->>Android_Build Runner:开始编译本master分支的release版
Android_Build Runner->>Nexus服务器:请求依赖包
Nexus服务器-->>Android_Build Runner:返回依赖包
Android_Build Runner-->>Android_Build Runner:完成编译
Android_Build Runner-->>Android_Build Runner:自动化测试(如果有的话)
Android_Build Runner->>Nexus服务器:上传组件到release仓库
Nexus服务器-->>Android_Build Runner: 返回成功
end

Android_Build Runner-->>GitLab服务器: 返回成功
GitLab服务器-->>GitLab服务器: 更新Web控制台上的指示图标
GitLab服务器-->> 开发人员:返回成功
Note left of GitLab服务器:发送邮件
}}

RTP客户端打包

Unicorn(RTP客户端Android版)的编译打包，我们使用了Jenkins，在预装了Android编译环境的从节点上进行构建，测试人/支持人员手工选择参数启动项目构建

构建的时候，先手工选择是标准版还是客户定制版，然后会编译指定版本并把安装包上传到通用存储服务器备用

为了简化，下图只描述了happy path

{{mermaid(Unicorn打包Jenkins CI过程)
sequenceDiagram
participant 测试人员
participant Jenkins主服务器
participant Android_Build从节点
participant GitLab服务器
participant Nexus服务器
participant 通用存储服务器
测试人员->>Jenkins主服务器:手工启动构建
Note right of 测试人员: 选择指定的标准版或客户定制版
Jenkins主服务器->>Android_Build从节点: 请求执行指令集合
Android_Build从节点->>GitLab服务器: 请求Unicorn仓库的代码
GitLab服务器-->>Android_Build从节点:返回代码
Android_Build从节点->>GitLab服务器:请求Custom_Unicorn仓库的代码
GitLab服务器-->>Android_Build从节点:返回代码
Android_Build从节点-->>Android_Build从节点:根据定制版本合并代码
Android_Build从节点-->>Android_Build从节点:开始编译
Android_Build从节点->>Nexus服务器:请求依赖组件
Note right of Android_Build从节点:包括公共组件和私有组件
Nexus服务器-->>Android_Build从节点:返回依赖组件
Android_Build从节点-->>Android_Build从节点: 完成编译
Android_Build从节点->>通用存储服务器:上传Unicorn指定版本
Note right of Android_Build从节点:包括安装包和自动升级文件
通用存储服务器-->>Android_Build从节点:返回成功
Android_Build从节点-->>Jenkins主服务器:返回成功
Jenkins主服务器-->>Jenkins主服务器: 更新Web控制台上的指示图标
Jenkins主服务器-->> 测试人员:返回成功
Note left of Jenkins主服务器:发送XMPP消息
}}

RTP客户端部署

Unicorn(RTP客户端Android版)的部署，我们仍然使用了Jenkins，它主要是拷贝安装包到指定的位置，所以在Jenkins主服务器上即可进行构建，测试人/支持人员手工选择参数启动项目构建

构建的时候，先手工选择是标准版还是客户定制版，然后填写升级描述，开始构建后会从通用存储服务器上获取安装包和自动升级文件，然后把自动升级文件中的升级描述替换掉，再上传到预定的位置，让最终用户下载安装和自动更新

为了简化，下图只描述了happy path

{{mermaid(Unicorn部署Jenkins CI过程)
sequenceDiagram
participant 测试人员
participant Jenkins主服务器
participant 通用存储服务器
participant 部署服务器
测试人员->>Jenkins主服务器:手工启动构建
Note right of 测试人员: 选择指定的标准版或客户定制版
Note right of 测试人员: 填写升级描述
Jenkins主服务器-->>Jenkins主服务器: 开始执行指令集合
Jenkins主服务器->>通用存储服务器:请求Unicorn指定版本
Note right of Jenkins主服务器:包括安装包和自动升级文件
通用存储服务器-->>Jenkins主服务器:返回Unicorn指定版本
Jenkins主服务器-->>Jenkins主服务器: 替换升级描述文字
Jenkins主服务器->>部署服务器:上传Unicorn指定版本
Note right of Jenkins主服务器:包括安装包和自动升级文件
部署服务器-->>Jenkins主服务器:返回成功
Jenkins主服务器-->>Jenkins主服务器: 更新Web控制台上的指示图标
Jenkins主服务器-->> 测试人员:返回成功
Note left of Jenkins主服务器:发送XMPP消息
}}

尾声

文章前面提到过的，SCM和CI是非常密切，所以另外的文章也稍微说了一点， {{article(21)}}

自动化测试，本来也想提一下，但是这部分情况非常复杂，客户端和服务器，各种语言/平台，都很不同，我们局部用到了一些，但是未成体系，很难弄一篇东西来涵盖这个话题。比如有的组件是可以用Java的自动化测试工具的，有的服务器测试也有自动化测试工具，还有一些组件和服务器需要我们自己写脚本或测试组件来测试。另外UI的交互测试也有一些自动化测试工具了，不过脚本的工作量也相当大。

所以，最终我们这里就只是把持续集成的主要部分，自动(或经人工干预)编译，打包，部署粗略说了说，比较简陋，算是抛砖引玉，有机会的话再和客户，同行当面交流。