

# Java基础

# 设计模式

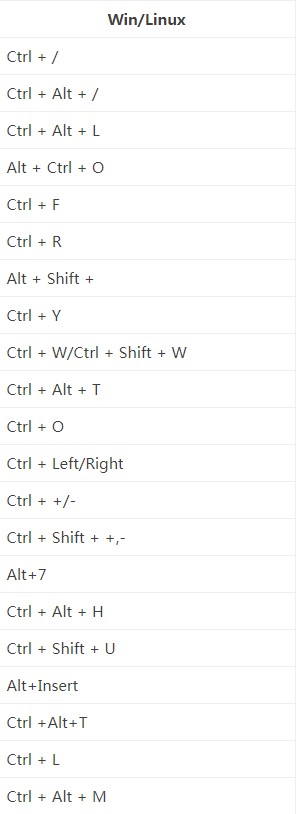
# 数据结构

# Android基础

## 四大组件

## 1.2 四大存储方式

## Android studio快捷键



## Android应用从项目源码到启动过程

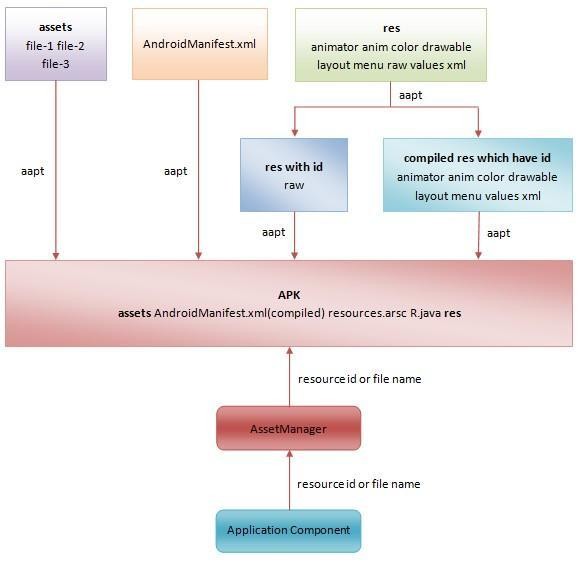
### 1.6.1 apk文件的生成

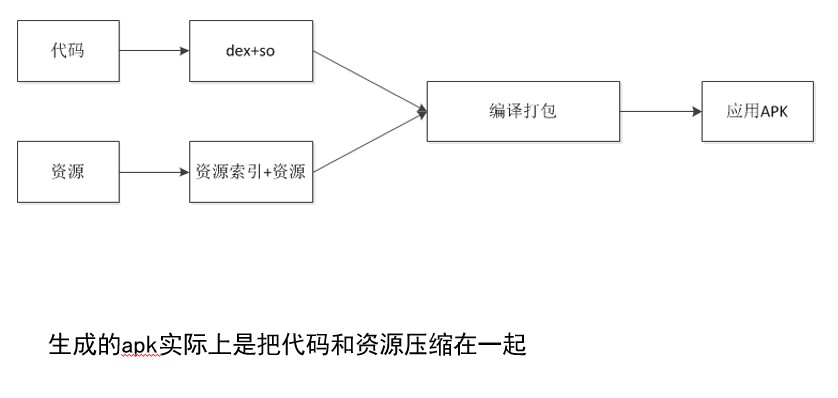


资源打包的过程：

aapt是Android asset packaging tool（Android资源打包工具）

开发APP需要代码和资源。 下图是资源打包过程和查找图：





除了assets资源和res/raw资源被原封不动的打包进apk之外，其他资源都会被编译或者处理，.xml文件会被编译为二进制的xml,解压apk之后无法打开这些xml文件。

除了assets资源外，其他资源都会被赋予一个资源ID

aapt负责编译和打包资源，编译完成后，会生成一个resources.arsc文件和一个R.java，前者是保存一个资源索引库，后者定义了一个资源的ID常量，供代码中索引资源，AndroidManifest.xml文件也会被编译成二进制的xml文件，然后打包到apk中

应用程序在运行时最终通过AssetManager来访问资源，或通过资源ID来访问，或者通过文件名来访问。

res目录和assets目录的区别：

1、引用资源的方式不同

res目录中的所有资源都会在R类的相应子类中生成对应的int类型变量，需要使用这些变量来引用资源。例如res/layout/main.xml是布局文件资源，会在R.layout类中生成一个名为main的变量，所以需要使用R.layout.main来引用main.xml。assets目录中的资源需要直接使用文件名来引用，例如assets/xyz.xml、assets/abc/test.db是assets目录中的两个资源文件，其中abc是assets中的子目录。可以使用如下的代码引用这两个资源文件，并返回与资源文件对应的InputStream对象，剩下的操作就是普通的Java代码输入流操作了。InputStream is1=getResources().getAssets().open("xyz.xml")；InputStream is2=getResources().getAssets().open("abc/test.db");

2、处理方式不同

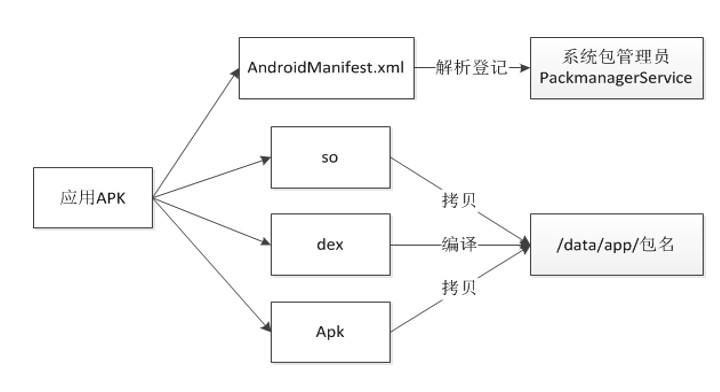
res目录中的资源子目录除了raw外，其他资源目录中的资源文件都会被编译，这也是为什么 将APK文件解压后无法直接查看XML格式资源文件内容的原因。而assets与res/raw目录中的资源文件不会做任何处理，所以将APK解压后，这两个目录中的资源文件都会保持原样.

3、子目录不同

res目录只能有一层子目录，而且这些子目录必须是预定义的，如res/layout、res/values等都是合法的，而res/abc,res/xyz并不是合法的资源目录，在assets目录中可以建任意层次的子目录(只受操作系统的限制)

### 1.6.2 apk文件的安装

安装apk主要包含解析Androidmanifest.xml文件，拷贝，编译dex几个主要步骤：

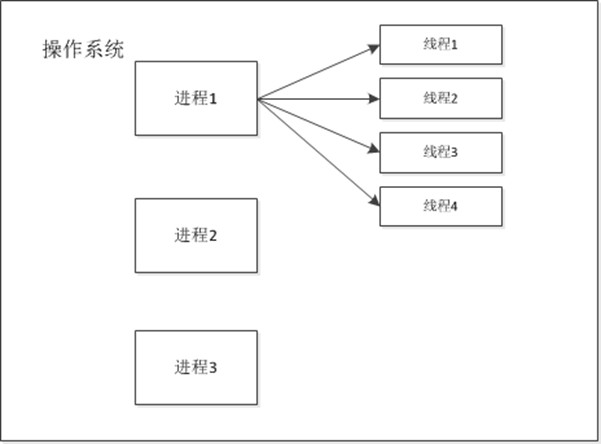


安装完成后base.apk是原始的apk，主要用于加载资源，或者解释执行；lib是apk中所有的C/C++生成的so文件；oat是dex文件(Java代码)编译优化后的虚拟机执行文件

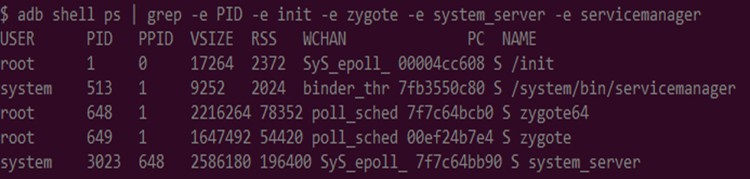
### 1.6.3应用的启动

进程：具有一定独立功能的程序，是系统进行资源分配和调度的一个独立单位，重点是系统调度和单独的单位，也就是说进程是可以独立运行的一段程序，大多数情况下，APP会运行在一个独立的Linux进程中。

线程：是进程的一个实体，是CPU调度和分配的最小单位，是比进程更小的能独立运行的基本单位，线程自己基本上不拥有系统资源。默认情况下APP一开始都会运行在同一个进程中的同一个线程，也就是主线程或UI线程。



/init:开机以后的第一个进程，负责解析init.rc脚本，启动守护进程，创建zygote进程



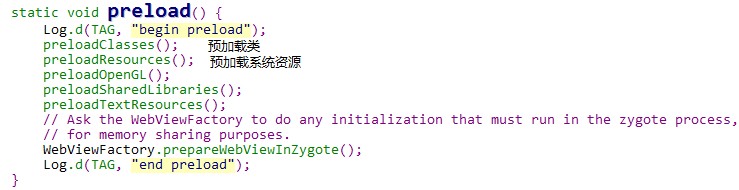
PID是该进程的ID，PPID是父进程的ID，UID是用户ID

zygote/zygote64:所有Android应用进程的父进程

/system/bin/servicemanager:binder通信机制守护进程，总管进程

system\_service:Android系统核心进程，负责创建核心的系统service，包括AMS（Activity Manager Service）/PMS(Package Manager Service)/WMS(Window Manager Service)

zygote进程：所有应用进程的父进程



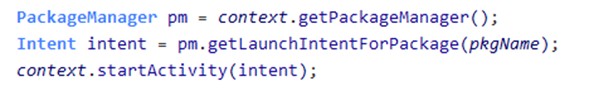
1、预加载Android提供的class：解决编译时框架依赖的原理

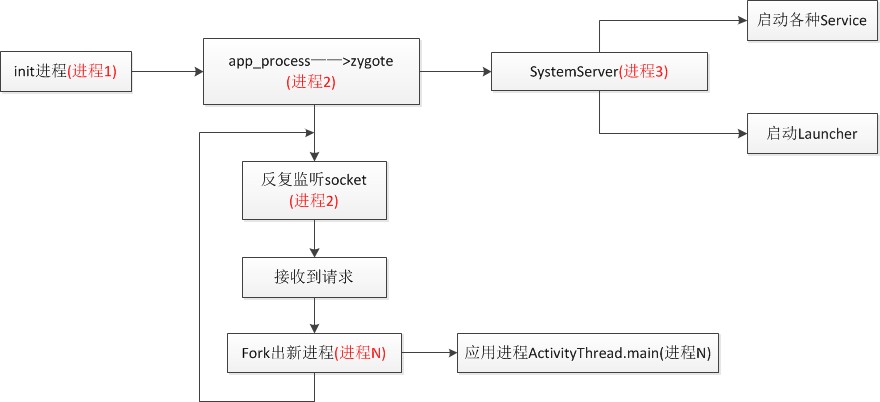
2、预加载系统资源：应用进程中直接访问系统资源的出处

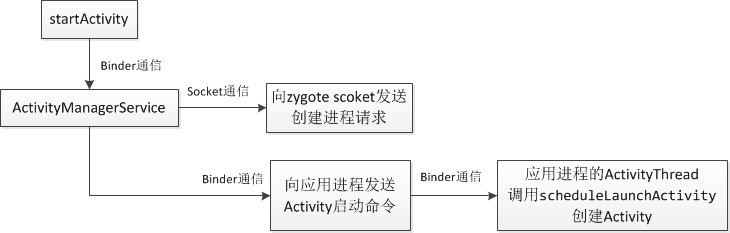
3、死循环监听socket，等待AMS的fork请求（即使请求新建一个进程）

应用进程fork自zygote进程，进入到子进程以后，会将预加载的类和资源一起带到子进程

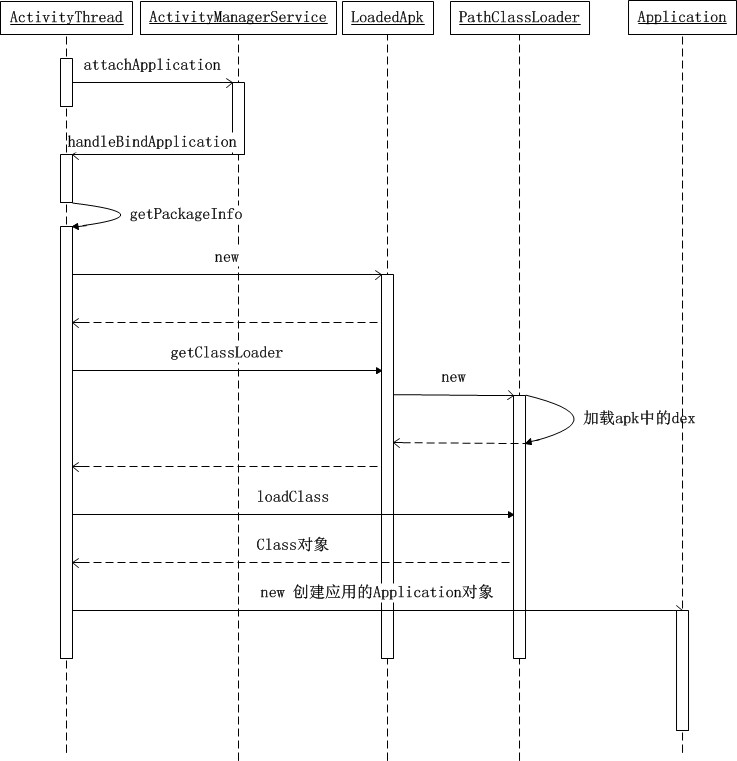
应用启动代码：







进程的启动：是一个通用的步骤，所有的应用在能开始运行之前，都需要经历一次zygote进程fork的过程，fork完成之后该进程和AMS之间没有任何联系，系统无法管理该应用进程，因此进程创建结束以后，应用进程需要主动和AMS建立联系，将其纳入AMS的管理范围。



应用启动过程总结：

1、AMS向zygote进程发起fork命令

2、zygote进程fork进程

3、进程主线程以ActivityThread.main作为入口开始应用进程的初始化，创建主线程的消息死循环，主线程一切皆是消息

4、应用进程向AMS尽心注册，加入AMS管理

5、应用进程加载odex（代码）和apk（资源）

6、进入应用定义的Application

7、启动Activity

大多数情况下一个APP运行在一个进程中，除非在AndroidManifest.xml文件中配置Android：process属性，或者通过native代码fork进程

应用的主线程：一切皆是消息，应用存活的基础（死循环）



## Context理解

Context是一个抽象类，源码解释：

*/\*\*  
 \* Interface to global information about an application environment. This is  
 \* an abstract class whose implementation is provided by  
 \* the Android system. It  
 \* allows access to application-specific resources and classes, as well as  
 \* up-calls for application-level operations such as launching activities,  
 \* broadcasting and receiving intents, etc.  
 \*/*

翻译：是应用程序环境的全局信息接口，是一个抽象类，它的实现是由Android系统提供，允许访问特定应用程序的资源和类（即是跨应用资源访问，获取已安装应用的context），以及用于应用程序级操作（启动活动、广播、接收intent）的回调。

### Context类中的抽象方法

以下方法的具体实现，要看其实现类中是怎么实现的

#### getAssets()

*/\*\*  
 \* Returns an AssetManager instance for the application's package.  
 \* <p>  
 \* <strong>Note:</strong> Implementations of this method should return  
 \* an AssetManager instance that is consistent with the Resources instance  
 \* returned by {****@link*** *#getResources()}. For example, they should share the  
 \* same {****@link*** *Configuration} object.  
 \*  
 \** ***@return*** *an AssetManager instance for the application's package  
 \** ***@see*** *#getResources()  
 \*/*public abstract AssetManager getAssets();

返回一个应用程序包的AssetManager实例

#### getResource()

*/\*\*  
 \* Returns a Resources instance for the application's package.  
 \* <p>  
 \* <strong>Note:</strong> Implementations of this method should return  
 \* a Resources instance that is consistent with the AssetManager instance  
 \* returned by {****@link*** *#getAssets()}. For example, they should share the  
 \* same {****@link*** *Configuration} object.  
 \*  
 \** ***@return*** *a Resources instance for the application's package  
 \** ***@see*** *#getAssets()  
 \*/*public abstract Resources getResources();

返回一个应用程序包的Resources实例

#### getPackageManager()

*/\*\* Return PackageManager instance to find global package information. \*/*public abstract PackageManager getPackageManager();

返回一个找到全局包信息的PackageManager实例

#### getContentResolver()

*/\*\* Return a ContentResolver instance for your application's package. \*/*public abstract ContentResolver getContentResolver();

返回一个应用程序包的ContentResolver实例

#### getMainLooper()

*/\*\*  
 \* Return the Looper for the main thread of the current process. This is  
 \* the thread used to dispatch calls to application components (activities,  
 \* services, etc).  
 \* <p>  
 \* By definition, this method returns the same result as would be obtained  
 \* by calling {****@link*** *Looper#getMainLooper() Looper.getMainLooper()}.  
 \* </p>  
 \*  
 \** ***@return*** *The main looper.  
 \*/*public abstract Looper getMainLooper();

为当前进程的主线程返回Looper，主线程是用于调度应用程序组件（activities，services等）的线程

#### getApplicationContext()

*/\*\*  
 \* Return the context of the single, global Application object of the  
 \* current process. This generally should only be used if you need a  
 \* Context whose lifecycle is separate from the current context, that is  
 \* tied to the lifetime of the process rather than the current component.  
 \*  
 \* <p>Consider for example how this interacts with  
 \* {****@link*** *#registerReceiver(BroadcastReceiver, IntentFilter)}:  
 \* <ul>  
 \* <li> <p>If used from an Activity context, the receiver is being registered  
 \* within that activity. This means that you are expected to unregister  
 \* before the activity is done being destroyed; in fact if you do not do  
 \* so, the framework will clean up your leaked registration as it removes  
 \* the activity and log an error. Thus, if you use the Activity context  
 \* to register a receiver that is static (global to the process, not  
 \* associated with an Activity instance) then that registration will be  
 \* removed on you at whatever point the activity you used is destroyed.  
 \* <li> <p>If used from the Context returned here, the receiver is being  
 \* registered with the global state associated with your application. Thus  
 \* it will never be unregistered for you. This is necessary if the receiver  
 \* is associated with static data, not a particular component. However  
 \* using the ApplicationContext elsewhere can easily lead to serious leaks  
 \* if you forget to unregister, unbind, etc.  
 \* </ul>  
 \*/*public abstract Context getApplicationContext();

返回当前进程的单个、全局Application对象的上下文，通常情况只能在需要一个上下文的情况下使用，该生命周期与当前上下文的生命周期是分开的。这个全局对象的上下文和进程的生命周期相关，而不是当前组件的生命周期。

#### getClassLoader()

*/\*\*  
 \* Return a class loader you can use to retrieve classes in this package.  
 \*/*public abstract ClassLoader getClassLoader();

返回一个可以用来取回此包中的类的类加载器

#### getPackagetName()

*/\*\* Return the name of this application's package. \*/*public abstract String getPackageName();

返回应用程序包的包名

#### getApplicationInfo()

*/\*\* Return the full application info for this context's package. \*/*public abstract ApplicationInfo getApplicationInfo();

返回这个context包的完整的应用程序信息

#### getPackageResourcePath()

*/\*\*  
 \* Return the full path to this context's primary Android package.  
 \* The Android package is a ZIP file which contains the application's  
 \* primary resources.  
 \*  
 \* <p>Note: this is not generally useful for applications, since they should  
 \* not be directly accessing the file system.  
 \*  
 \** ***@return*** *String Path to the resources.  
 \*/*public abstract String getPackageResourcePath();

返回这个Android应用包的完整路径，这个Android包是一个包含了应用resources的一个zip压缩文件。其实就是APK的路径。

#### getPackageCodePath()

*/\*\*  
 \* Return the full path to this context's primary Android package.  
 \* The Android package is a ZIP file which contains application's  
 \* primary code and assets.  
 \*  
 \* <p>Note: this is not generally useful for applications, since they should  
 \* not be directly accessing the file system.  
 \*  
 \** ***@return*** *String Path to the code and assets.  
 \*/*public abstract String getPackageCodePath();

同上，不过这个zip压缩包包含的是这个应用主要的code和assets，这里要明白assets和resources的区别，详见1.6.1 apk文件的生成

#### getSystemService()

*/\*\*  
 \* Return the handle to a system-level service by name. The class of the  
 \* returned object varies by the requested name. Currently available names  
 \* are:  
 \*  
 \* <dl>  
 \* <dt> {****@link*** *#WINDOW\_SERVICE} ("window")  
 \* <dd> The top-level window manager in which you can place custom  
 \* windows. The returned object is a {****@link*** *android.view.WindowManager}.  
 \* <dt> {****@link*** *#LAYOUT\_INFLATER\_SERVICE} ("layout\_inflater")  
 \* <dd> A {****@link*** *android.view.LayoutInflater} for inflating layout resources  
 \* in this context.  
 \* <dt> {****@link*** *#ACTIVITY\_SERVICE} ("activity")  
 \* <dd> A {****@link*** *android.app.ActivityManager} for interacting with the  
 \* global activity state of the system.  
 \* <dt> {****@link*** *#POWER\_SERVICE} ("power")  
 \* <dd> A {****@link*** *android.os.PowerManager} for controlling power  
 \* management.  
 \* <dt> {****@link*** *#ALARM\_SERVICE} ("alarm")  
 \* <dd> A {****@link*** *android.app.AlarmManager} for receiving intents at the  
 \* time of your choosing.  
 \* <dt> {****@link*** *#NOTIFICATION\_SERVICE} ("notification")  
 \* <dd> A {****@link*** *android.app.NotificationManager} for informing the user  
 \* of background events.  
 \* <dt> {****@link*** *#KEYGUARD\_SERVICE} ("keyguard")  
 \* <dd> A {****@link*** *android.app.KeyguardManager} for controlling keyguard.  
 \* <dt> {****@link*** *#LOCATION\_SERVICE} ("location")  
 \* <dd> A {****@link*** *android.location.LocationManager} for controlling location  
 \* (e.g., GPS) updates.  
 \* <dt> {****@link*** *#SEARCH\_SERVICE} ("search")  
 \* <dd> A {****@link*** *android.app.SearchManager} for handling search.  
 \* <dt> {****@link*** *#VIBRATOR\_SERVICE} ("vibrator")  
 \* <dd> A {****@link*** *android.os.Vibrator} for interacting with the vibrator  
 \* hardware.  
 \* <dt> {****@link*** *#CONNECTIVITY\_SERVICE} ("connection")  
 \* <dd> A {****@link*** *android.net.ConnectivityManager ConnectivityManager} for  
 \* handling management of network connections.  
 \* <dt> {****@link*** *#WIFI\_SERVICE} ("wifi")  
 \* <dd> A {****@link*** *android.net.wifi.WifiManager WifiManager} for management of Wi-Fi  
 \* connectivity. On releases before NYC, it should only be obtained from an application  
 \* context, and not from any other derived context to avoid memory leaks within the calling  
 \* process.  
 \* <dt> {****@link*** *#WIFI\_P2P\_SERVICE} ("wifip2p")  
 \* <dd> A {****@link*** *android.net.wifi.p2p.WifiP2pManager WifiP2pManager} for management of  
 \* Wi-Fi Direct connectivity.  
 \* <dt> {****@link*** *#INPUT\_METHOD\_SERVICE} ("input\_method")  
 \* <dd> An {****@link*** *android.view.inputmethod.InputMethodManager InputMethodManager}  
 \* for management of input methods.  
 \* <dt> {****@link*** *#UI\_MODE\_SERVICE} ("uimode")  
 \* <dd> An {****@link*** *android.app.UiModeManager} for controlling UI modes.  
 \* <dt> {****@link*** *#DOWNLOAD\_SERVICE} ("download")  
 \* <dd> A {****@link*** *android.app.DownloadManager} for requesting HTTP downloads  
 \* <dt> {****@link*** *#BATTERY\_SERVICE} ("batterymanager")  
 \* <dd> A {****@link*** *android.os.BatteryManager} for managing battery state  
 \* <dt> {****@link*** *#JOB\_SCHEDULER\_SERVICE} ("taskmanager")  
 \* <dd> A {****@link*** *android.app.job.JobScheduler} for managing scheduled tasks  
 \* <dt> {****@link*** *#NETWORK\_STATS\_SERVICE} ("netstats")  
 \* <dd> A {****@link*** *android.app.usage.NetworkStatsManager NetworkStatsManager} for querying network  
 \* usage statistics.  
 \* <dt> {****@link*** *#HARDWARE\_PROPERTIES\_SERVICE} ("hardware\_properties")  
 \* <dd> A {****@link*** *android.os.HardwarePropertiesManager} for accessing hardware properties.  
 \* </dl>  
 \*  
 \* <p>Note: System services obtained via this API may be closely associated with  
 \* the Context in which they are obtained from. In general, do not share the  
 \* service objects between various different contexts (Activities, Applications,  
 \* Services, Providers, etc.)  
 \*  
 \** ***@param*** *name The name of the desired service.  
 \*  
 \** ***@return*** *The service or null if the name does not exist.  
 \*  
 \** ***@see*** *#WINDOW\_SERVICE  
 \** ***@see*** *android.view.WindowManager  
 \** ***@see*** *#LAYOUT\_INFLATER\_SERVICE  
 \** ***@see*** *android.view.LayoutInflater  
 \** ***@see*** *#ACTIVITY\_SERVICE  
 \** ***@see*** *android.app.ActivityManager  
 \** ***@see*** *#POWER\_SERVICE  
 \** ***@see*** *android.os.PowerManager  
 \** ***@see*** *#ALARM\_SERVICE  
 \** ***@see*** *android.app.AlarmManager  
 \** ***@see*** *#NOTIFICATION\_SERVICE  
 \** ***@see*** *android.app.NotificationManager  
 \** ***@see*** *#KEYGUARD\_SERVICE  
 \** ***@see*** *android.app.KeyguardManager  
 \** ***@see*** *#LOCATION\_SERVICE  
 \** ***@see*** *android.location.LocationManager  
 \** ***@see*** *#SEARCH\_SERVICE  
 \** ***@see*** *android.app.SearchManager  
 \** ***@see*** *#SENSOR\_SERVICE  
 \** ***@see*** *android.hardware.SensorManager  
 \** ***@see*** *#STORAGE\_SERVICE  
 \** ***@see*** *android.os.storage.StorageManager  
 \** ***@see*** *#VIBRATOR\_SERVICE  
 \** ***@see*** *android.os.Vibrator  
 \** ***@see*** *#CONNECTIVITY\_SERVICE  
 \** ***@see*** *android.net.ConnectivityManager  
 \** ***@see*** *#WIFI\_SERVICE  
 \** ***@see*** *android.net.wifi.WifiManager  
 \** ***@see*** *#AUDIO\_SERVICE  
 \** ***@see*** *android.media.AudioManager  
 \** ***@see*** *#MEDIA\_ROUTER\_SERVICE  
 \** ***@see*** *android.media.MediaRouter  
 \** ***@see*** *#TELEPHONY\_SERVICE  
 \** ***@see*** *android.telephony.TelephonyManager  
 \** ***@see*** *#TELEPHONY\_SUBSCRIPTION\_SERVICE  
 \** ***@see*** *android.telephony.SubscriptionManager  
 \** ***@see*** *#CARRIER\_CONFIG\_SERVICE  
 \** ***@see*** *android.telephony.CarrierConfigManager  
 \** ***@see*** *#INPUT\_METHOD\_SERVICE  
 \** ***@see*** *android.view.inputmethod.InputMethodManager  
 \** ***@see*** *#UI\_MODE\_SERVICE  
 \** ***@see*** *android.app.UiModeManager  
 \** ***@see*** *#DOWNLOAD\_SERVICE  
 \** ***@see*** *android.app.DownloadManager  
 \** ***@see*** *#BATTERY\_SERVICE  
 \** ***@see*** *android.os.BatteryManager  
 \** ***@see*** *#JOB\_SCHEDULER\_SERVICE  
 \** ***@see*** *android.app.job.JobScheduler  
 \** ***@see*** *#NETWORK\_STATS\_SERVICE  
 \** ***@see*** *android.app.usage.NetworkStatsManager  
 \** ***@see*** *android.os.HardwarePropertiesManager  
 \** ***@see*** *#HARDWARE\_PROPERTIES\_SERVICE  
 \*/*public abstract Object getSystemService(@ServiceName @NonNull String name);

通过名称返回系统级服务的句柄，返回对象的类根据请求的名称而异，目前能请求到的系统服务有以上几种。

# Git使用

## Git基本概念

版本控制：记录若干文件变化，以便将来查阅特定版本修订情况的系统。

文件的三种状态：对任何一个文件，在git内都有三种状态：已提交（committed），已修改（modified），已暂存（staged）。已提交：表示该文件已经被安全的保存到了本地数据库中；已修改：表示已经修改了某个文件，但是还没有提交保存；已暂存：表示把已修改的文件放在下次提交时要保存的清单中

每个项目都有一个Git目录（.git）它是Git用来保存元数据和对象数据的地方，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据

Git工作流程：在工作目录中修改某些文件；对修改后的文件进行快照，然后保存到暂存区域；提交更新，将保存在暂存区域的文件快照永久转存到Git目录中。

对现有的某个项目开始使用git管理：git init；.git文件是隐藏的，用于保存下载下来的所有版本记录想要看里面的内容需要在文件夹下显示。

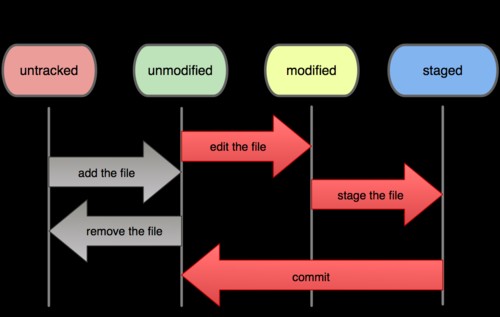
如果当前目录有几个文件想要纳入版本控制，需要git add命令告诉git开始追踪这些文件，然后提交

从现有仓库克隆：git clone [url]；

$ git clone git://github.com/schacon/grit.git mygrit

git支持许多数据传输协议，git：//；http(s)://；user2server：/path.git表示的ssh传输协议

文件状态生命周期：已跟踪和未跟踪，已跟踪的文件时指本来就纳入版本控制管理的文件。



git status命令：用于检查当前文件的状态

git add命令：用于跟踪新文件

git commit命令：用于提交文件

git diff命令：查看不同（difference）

git log命令：显示从最近到最远的提交日志

git reset --hard HEAD^：退回到上一个版本，如果想在退回到上一个版本之后再回到退回之前的版本，git reset --hard [commit id](退回之前的版本的commit ID的前几位即可，通过git log可以查看)，HEAD表示当前版本，HEAD^表示上一个版本，上上一个版本是HEAD^^表示，往上100个版本用HEAD~100表示

git reflog查看提交历史，查看记录每一次命令，前面显示的是commit id，可以通过commit id恢复到以往的任意一个版本

git checkout -- file：恢复到最近一次git commit或者git add之前的状态

cat：查看文档内容

## 建立本地git和远程GitHub仓库的联系

具体怎么创建联系的参照《Git教程By廖雪峰》

git仓库和GitHub之间的传输是通过SSH加密的

git push -u origin master：第一次推送master分支所有内容

git push origin master：推送最新修改

要克隆一个仓库，首先必须知道仓库的地址，然后使用git clone命令克隆

Git支持多种协议，包括https，但通过ssh支持的原生git协议速度最快

.gitignore文件：是列出要忽略的文件模式

所有空行或者以注释符号#开头的行都会被Git忽略

/说明要忽略的是目录；

\*匹配零个或多个任意字符

[abc]匹配任何一个列在方括号中的内容，[0-9]表示所有在这两个字符范围内的都可以匹配

？只匹配一个任意字符

## 分支管理

查看分支：git branch

创建分支：git branch name

切换分支：git checkout name

创建+切换分支：git checkout -b name

合并某分支到当前分支：git merge name

删除分支：git branch -d name

解决冲突

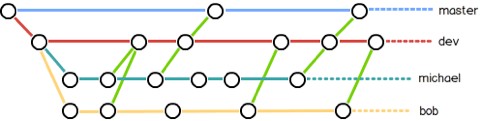
当git无法自动合并分支时，就必须首先解决冲突，解决冲突后，再提交，合并完成

git log --graph命令可以看到分支合并图

## 分支策略

master分支是非常稳定的，仅仅用来发布新版本，平时不能在上面干活，

干活都在dev分支上，dev分支是不稳定的，到某个版本发布时，再把dev分支合并到master分支上，在master分支发布11.0版本



合并分支，加上--no-ff参数就可以用普通模式合并，合并后的历史有分支，能看出来曾经做过合并，而fast forward合并就看不出来曾经做过合并。

## Bug分支

修复bug时，通过创建新的bug分支进行修复，然后合并，最后删除

当手头工作没有完成时，先把工作现场git stash一下，然后去修复bug，修复后，再git stash pop，回到工作现场

## Feature分支

开发新功能，最好建立一个新分支，如果要丢弃一个没有被合并过的分支，可以通过git branch -D name强行删除

## 多人协作

查看远程库信息，使用git remote -v

本地新建的分支如果不推送到远程，对其他人就是不可见的

从本地推送分支，使用git push origin branch-name，如果推送失败，先用git pull抓取远程的新提交

在本地创建和远程分支对应的分支，使用git checkout -b branch-name origin/branch-name，本地分支和远程分支的名字最好一致

建立本地分支和远程分支的关联，使用 gitbranch --set-upstream branch-name origin/branch-name

从远程抓取分支，使用git pull，如果有冲突，要首先处理冲突。

## 创建标签

命令git tag name 用于新建一个标签，默认HEAD，也可制定一个commit id

-a tagname -m "bababababab"可以指定标签信息

命令git tag可以查看所有标签

## 操作标签

命令git push origin tagname可以推送一个本地标签

命令git push origin --tags可以推送全部未推送过的本地标签

命令git tag -d tagname 可以删除一个本地标签

命令git push origin ：refs/tags/tagname可以删除一个远程标签

## 使用github

github上可以任意Fork开源仓库

自己拥有Fork后的仓库的读写权限

可以推送pull request给官方仓库来贡献代码

# adb使用

什么是adb：借助adb工具，可以管理设备或手机模拟器的状态，进行手机操作：安装软件、系统升级、运行shell命令等，简单而言就是连接Android设备和pc端的桥梁，可以让用户在电脑上对手机进行全面的操作。

## 1.1 adb常用基本命令

adb root：获取管理员权限

adb logcat -s 标签名：在命令行中查看LOG信息

adb start-server：开启adb服务

adb kill-server：终止服务

adb devices：列出所有连接到adb的设备，包括真机和模拟器

adb -s <serialNumber>：指定adb指令的目标

adb version：显示adb的版本号

adb install <apk file name>：将指定的APP安装到唯一连接到adb的设备上，后边的参数为安装包的文件名

adb install -r <apk file name>：安装应用程序，但是保留原有应用程序的所有数据

adb uninstall <package name>：将包名指定的app从目标设备上卸载，后面的参数为应用程序包名

adb uninstall -k <package name>：卸载指定包名的APP，但是保留配置文件和缓存

adb push <local> <remote>：将本机的文件或目录拷贝到本机中

adb shell：进入到目标设备的linux环境，在该环境中可以执行Linux命令，linux shell环境中执行ps可以查看Android设备中运行的所有进程。执行exit可以退出linux shell环境

## 1.2 adb logcat命令查看并过滤Android输出log

Android输出的每一条日志都有一个标记和优先级与其关联：就是tag和优先级

ctrl+c中断输出log记录

adb logcat：查看全部日志

adb logcat > c:\test.log ：将日志保存到文件test.log中

例如：仅输出标记为“ActivityManager”且优先级大于等于“Info”和标记为“PowerManagerService”并且优先级大于等于“Debug”的日志

adb logcat ActivityManager：I PowerManagerService：D \*：S

\*：S用于设置所有标记的日志优先级为S，这样可以确保仅输出符合条件的日志

# 待学习

# 专利