学习笔记

一、git

1. SVN与Git的区别？

> 1. SVN是集中式版本控制系统，Git是分布式版本控制系统；
> 2. GIT把内容按元数据方式存储，而SVN是按文件：因为git目录是处于个人机器上的一个克隆版的版本库，它拥有中心版本库上所有的东西，例如标签，分支，版本记录等。
> 3. GIT分支和SVN的分支不同： svn会发生分支遗漏的情况，而git可以同一个工作目录下快速的在几个分支间切换，很容易发现未被合并的分支，简单而快捷的合并这些文件。
> 4. GIT没有一个全局的版本号，而SVN有（Git缺点）；
> 5. GIT的内容完整性要优于SVN:GIT的内容存储使用的是SHA-1哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。

2. 集中式和分布式的区别

> 集中式版本控制系统：版本库是集中存放在中央服务器的，而干活的时候，用的都是自己的电脑，所以要先从中央服务器取得最新的版本，然后开始干活，干完活了，再把自己的活推送给中央服务器。集中式版本控制系统最大的毛病就是必须联网才能工作。
> 分布式版本控制系统：分布式版本控制系统根本没有“中央服务器”，每个人的电脑上都是一个完整的版本库，这样，你工作的时候，就不需要联网了，因为版本库就在你自己的电脑上。比方说你在自己电脑上改了文件A，你的同事也在他的电脑上改了文件A，这时，你们俩之间只需把各自的修改推送给对方，就可以互相看到对方的修改了。

3. 创建版本库

第一步创建一个空目录：

```
$ mkdir learngit
$ cd learngit
$ pwd
/Users/michael/learngit
```

- pwd（print working directory）打印当前的工作目录；
  【注】windows系统注意目录名（父目录）不包含中文以防出现奇怪问题；

第二步，通过`git init`命令吧这个目录变成Git可以管理的仓库（即**初始化Git仓库**）：

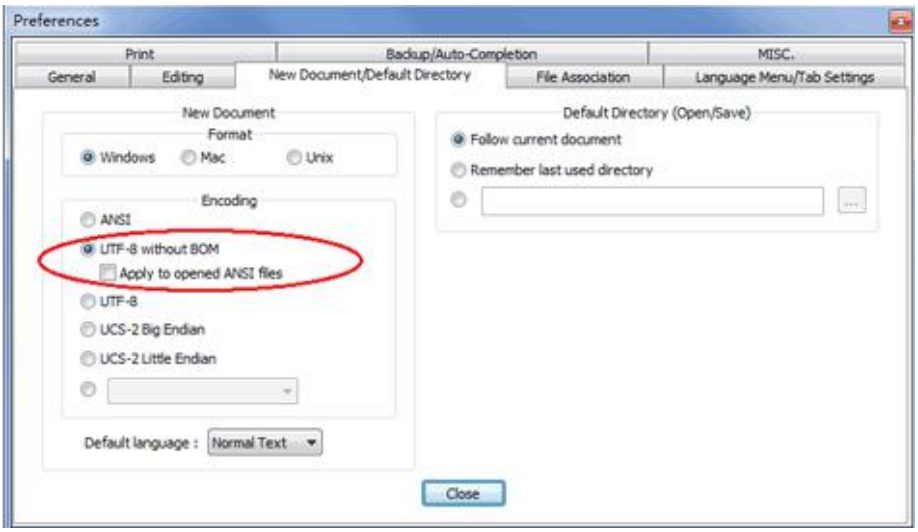
```
$ git init
Initialized empty Git repository in
/Users/michael/learngit/.git/
```

瞬间Git就把仓库建好了，而且告诉我们是一个空的仓库（empty Git repository），细心的读者可以发现当前目录下多了一个.git的目录，这个目录是Git来跟踪管理版本库的，没事千万不要手动修改这个目录里面的文件，不然改乱了，就把Git仓库给破坏了。

如果你没有看到`.git`目录，那是因为这个目录默认是隐藏的，用`ls -ah`命令就可以看见。

使用windows特别注意：

千万不要使用Windows自带的记事本编辑任何文本文件， 原因是Microsoft开发记事本的团队使用了一个非常弱智的行为来保存UTF-8编码的文件，他们自作聪明地在每个文件开头添加了0xefbbbf（十六进制）的字符，你会遇到很多不可思议的问题，比如，网页第一行可能会显示一个“?”，明明正确的程序一编译就报语法错误，等等，都是由记事本的弱智行为带来的。建议你下载Notepad++代替记事本，不但功能强大，而且免费！记得把Notepad++的默认编码设置为UTF-8 without BOM即可：



第三步 添加文件到Git仓库，分两步：

1. 使用命令`git add <file>`，注意，可以多次使用添加多个文件；
    - 添加指定文件到暂存区：
      `$ git add [file1] [file2] ...`
    - 添加指定目录到暂存区，包括子目录:
      `$ git add [dir]`
    - 添加当前目录的所有文件到暂存区
      `$ git add .`
      添加每个变化前，都会要求确认,对于同一个文件的多处变化，可以实现分次提
      `$ git add -p`
2. 使用命令`git commit -m <message>`完成；
    - 提交暂存区到仓库区:
      `git commit -m [message]`
    - 提交暂存区的指定文件到仓库区:
      `$ git commit [file1] [file2] ... -m [message]`

- 提交工作区自上次commit之后的变化，直接到仓库区:
  `$ git commit -a`
- 提交时显示所有diff信息:
  `$ git commit -v`
- 使用一次新的commit，替代上一次提交,如果代码没有任何新变化，则用来改写上一次commit的提交信息:
  `$ git commit --amend -m [message]`
- 重做上一次commit，并包括指定文件的新变化:
  `$ git commit --amend [file1] [file2] ...`

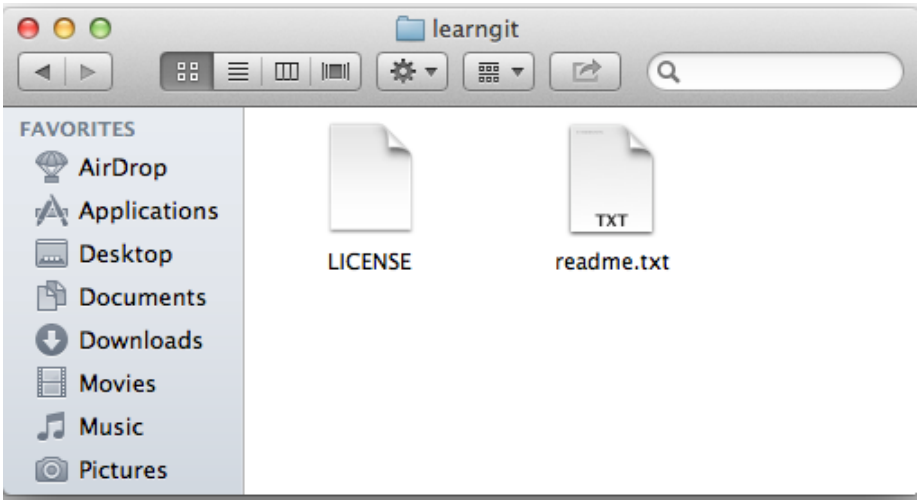
4. 版本回退

1. 查看信息：
   - 显示当前分支的版本历史:
     `$ git log`
   - 显示简短的log：
     `$ git log --pretty=oneline`
   - 显示commit历史，以及每次commit发生变更的文件:
     `$ git log --stat`
   - 搜索提交历史，根据关键词
     `$ git log -S [keyword]`
   - 显示某个commit之后的所有变动，每个commit占据一行：
     `git log [tag] HEAD --pretty=format:%s`
     ...
   - 记录你的每一次命令：
     `$ git reflog`
2. 版本回退：
   - 回退到上一个版本
     `$ git reset --hard HEAD^`
   - 回退到上上个版本
     `$ git reset --hard HEAD^^`
   - 回退到前n个版本
     `$ git reset --hard HEAD~n`
   - 回退到某个版本
     `$ git reset --hard 1094a`(Git的版本回退速度非常快，因为Git在内部有个指向当前版本的HEAD指针，当你回退版本的时候，Git仅仅是把HEAD知道id开头未‘1094a’，然后顺便把工作区的文件更新了。)

5. 工作区和暂存区

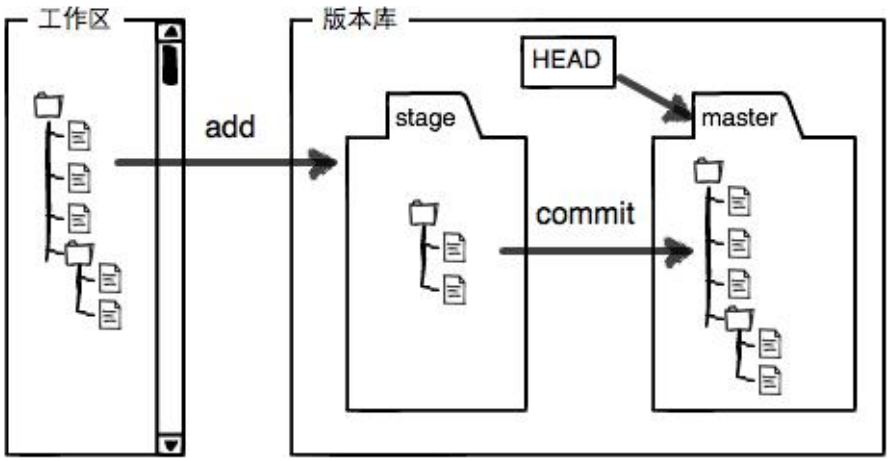
工作区（**Working Directory**）

就是你在电脑里能看到的目录，比如我的learngit文件夹就是一个工作区：



版本库（Repository）

工作区有一个隐藏目录.git，这个不算工作区，而是Git的版本库。
Git的版本库里存了很多东西，其中最重要的就是称为stage（或者叫index）的暂存区，还有Git为我们自动创建的第一个分支master，以及指向master的一个指针叫HEAD。



6.管理修改

`$git commit`只负责把暂存区的修改提交,提交后，用`$git diff HEAD -- readme.txt`命令可以查看工作区和版本库里面最新版本的区别;
【总结】： 不用`git add`到暂存区，那就不会加入到`commit`中；

7. 撤销修改

- 还没有add到暂存区时`$git checkout -- file`可以丢弃工作区的修改(撤销工作区修改)：
  `$git checkout -- reademe.txt`
- 加入到暂存区未commit（撤销暂存区的修改,回到工作区）
  `$git reset HEAD <file>`
  `$git reset HEAD readme.txt`
  用HEAD表示到最新的版本;
  用`$git status`查看，暂存区时干净的，工作区有修改;
- 已经commit了，只能参考4版本回退的方法了;

- 撤销merging状态状态，`git reset --hard HEAD`；
- 撤销(dev|REBASE 1/5)状态， `git rebase --abort` 来取消目前的进程；

---

8. 删除文件

- 在文件管理器中把没用的文件删了，或者用rm命令删了：
  `$ rm test.txt`
  这个时候，Git知道你删除了文件，因此，工作区和版本库就不一致了，git status命令会立刻告诉你哪些文件被删除了：

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be
committed)
  (use "git checkout -- <file>..." to discard changes in
working directory)

    deleted:    test.txt

no changes added to commit (use "git add" and/or "git
commit -a")
```

现在你有两个选择，一是确实要从版本库中删除该文件，那就用命令git rm删掉，并且git commit：

```
$ git rm test.txt
rm 'test.txt'

$ git commit -m "remove test.txt"
[master d46f35e] remove test.txt
 1 file changed, 1 deletion(-)
 delete mode 100644 test.txt
```

现在，文件就从版本库中被删除了。
另一种情况是删错了，因为版本库里还有呢，所以可以很轻松地把误删的文件恢复到最新版本： (撤销操作)
`$ git checkout -- test.txt`
`$git checkout`其实是用版本库里的版本替换工作区的版本，无论工作区是修改还是删除，都可以“一键还原”。

小结

命令git rm用于删除一个文件。如果一个文件已经被提交到版本库，那么你永远不用担心误删，但是要小心，你只能恢复文件到最新版本，你会丢失最近一次提交后你修改的内容。

9. 远程仓库

- 关联远程仓库:
  `$ git remote add origin git@github.com:michaelliao/learngit.git`
- 本地库的所有内容推送到远程库上:
  `$ git push -u origin master`
  由于远程库是空的，我们第一次推送master分支时，加上了-u参数，Git不但会把本地的master分支内容推送的远程新的master分支，还会把本地的master分支和远程的master分支关联起来，在以后的推送或者拉取时就可以简化命令，以后直接`git push`进行推送；

10. 从远程库克隆

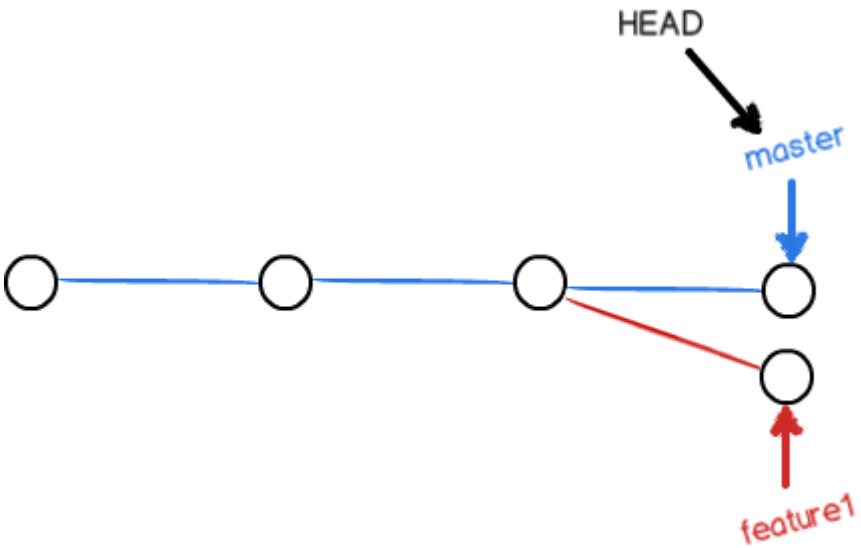
```
$ git clone git@github.com:Grekit-Sun/gitskills.git
```

11. 创建合并

- 创建dev分支，然后切换到dev分支：`$git switch -c dev`;
- 查看当前分支：`$git branch`;
- 切回master分支：`$git switch master`
- 将dev分支工作合并到master:`$git merge dev`
- 合并完成后可以删除分支：`$git branch -d dev`

12. 解决冲突

`master`分支和`feature1`分支各自都分别有新的提交，变成了这样：



这种情况下，Git无法执行“快速合并”，只能试图把各自的修改合并起来，但这种合并就可能会有冲突，我们试试看：

```
$ git merge feature1
Auto-merging readme.txt
CONFLICT (content): Merge conflict in readme.txt
Automatic merge failed; fix conflicts and then commit the
result.
```

Git告诉我们，`readme.txt`文件存在冲突，必须手动解决冲突后再提交。`git status`也可以告诉我们冲突的文件：

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:   readme.txt

no changes added to commit (use "git add" and/or "git
commit -a")
```

我们可以直接查看readme.txt的内容：

```
Git is a distributed version control system.
Git is free software distributed under the GPL.
Git has a mutable index called stage.
Git tracks changes of files.
<<<<<<< HEAD
Creating a new branch is quick & simple.
=======
Creating a new branch is quick AND simple.
>>>>>>> feature1
```

Git用`<<<<<<<`，`=======`，`>>>>>>>`标记出不同分支的内容，我们修改好后保存；

再提交：

```
$ git add readme.txt
$ git commit -m "conflict fixed"
[master cf810e4] conflict fixed
```

最后，删除`feature1`分支：

```
$ git branch -d feature1
Deleted branch feature1 (was 14096d0).
```

分支管理策略

通常，合并分支时，如果可能，Git会用Fast forward模式，但这种模式下，删除分支后，会丢掉分支信息。
如果要强制禁用Fast forward模式，Git就会在merge时生成一个新的commit，这样，从分支历史上就可以看出分支信息。
下面我们实战一下--no-ff方式的git merge：

- 创建并切换dev分支：

```
$ git switch -c dev
Switched to a new branch 'dev'
```

- 现在，我们切换回master：

```
$ git checkout master
Switched to branch 'master'
```

- 准备合并dev分支，请注意--no-ff参数，表示禁用Fast forward：

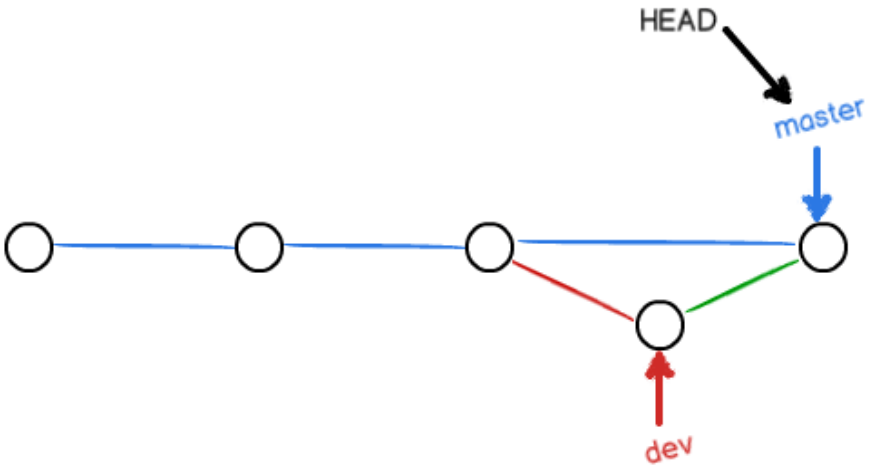
```
$ git merge --no-ff -m "merge with no-ff" dev
Merge made by the 'recursive' strategy.
 readme.txt | 1 +
 1 file changed, 1 insertion(+)
```

因为本次合并要创建一个新的commit，所以加上-m参数，把commit描述写进去。

- 合并后，我们用git log看看分支历史：

```
$ git log --graph --pretty=oneline --abbrev-commit
*   e1e9c68 (HEAD -> master) merge with no-ff
|\
| * f52c633 (dev) add merge
|/
*   cf810e4 conflict fixed
...
```

可以看到，不使用Fast forward模式，merge后就像这样：



小结

Git分支十分强大，在团队开发中应该充分应用。合并分支时，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。

举一反三

git rebase 和merge

`git rebase`使用:

1. `git rebase master`:分支（自己开发的分支），master待合进的分支;
2. `git merge feature(分支)`

使用 `rebase` 和 `merge` 的基本原则：

- 下游分支更新上游分支内容的时候使用 rebase;
- 上游分支合并下游分支内容的时候使用 merge;
- 更新当前分支的内容时一定要使用 --rebase 参数;
  例如现有上游分支 master，基于 master 分支拉出来一个开发分支 dev，在 dev 上开发了一段时间后要把 master 分支提交的新内容更新到 dev 分支，此时切换到 dev 分支，使用 git rebase master。

等 dev 分支开发完成了之后，要合并到上游分支 master 上的时候，切换到 master 分支，使用 `git merge dev`;

git中merge，rebase，cherry-pick，patch的联系与区别

这些操作都是为了把一个分支上的工作加到另一个分支上。

1. merge
   把另一个分支合并到当前分支上。
2. rebase
   把当前分支的提交在另一分支上重演。（如果可以成功重演，本

分支将会消失）

3. cherry-pick
   把本分支或者其他分支的某一次或某几次提交，在当前分支上重演。
4. patch
   把一次或几次提交，做成补丁文件（可以远程发送给其他人，这是与cherry-pick最大的不同）。这个补丁文件可以被应用到其它分支上。

git rebasse自己理解

- `git rebase <主干>`：就是当前的分支合并入主干版本，解决冲突依次`git add <>`,`git rebase --continue`；之后切换到主干版本用`git merge <分支>`;

---

git rm与git rm --cached 的区别

git rm 是删除暂存区或分支上的文件, 同时也删除工作区中这个文件。
git rm --cached是删除暂存区或分支上的文件,但本地还保留这个文件， 是不希望这个文件被版本控制。

---

git查看本机秘钥

输入命令`$ ssh-keygen -t rsa -C "1024809664@qq.com"`

Git学习心得

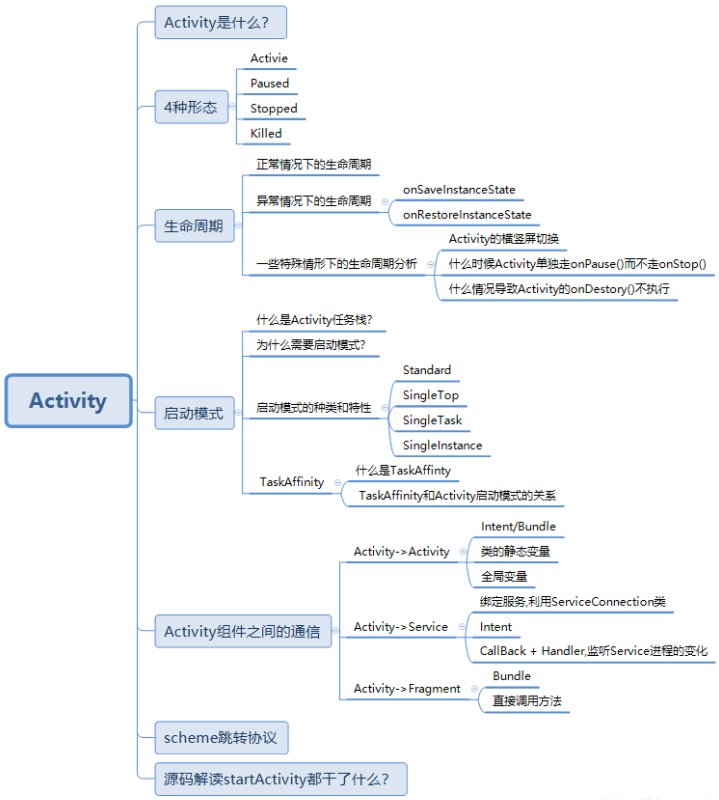
> 由于之前没有用过`git`，一开始学习的时候想要把每条指令都学习理解，都看了一遍之后，发现很多都记不住，然后我就在这边自己练习，模拟真实项目开发环境，练习中遇到各种问题，如连接不上仓库，是因为没有添加秘钥等，通过磊哥的指导，以及自己的再次深入学习，之后对常用的命令得以熟悉，可能后面工作中应用到`git`时还会遇到些小问题，有了这些天的学习基础，我有信心一定可以解决的；

Android学习

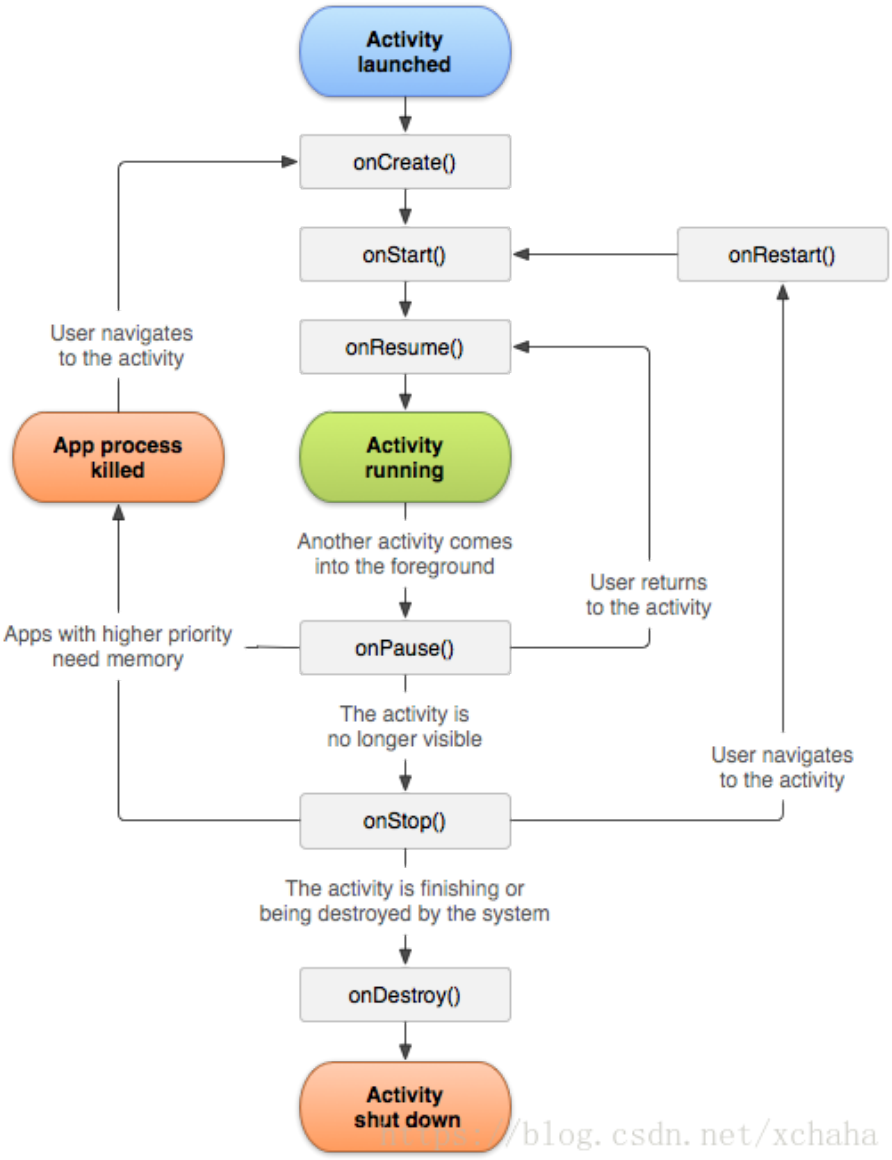
Android 集成Git、Android Studio提交代码

四大组件

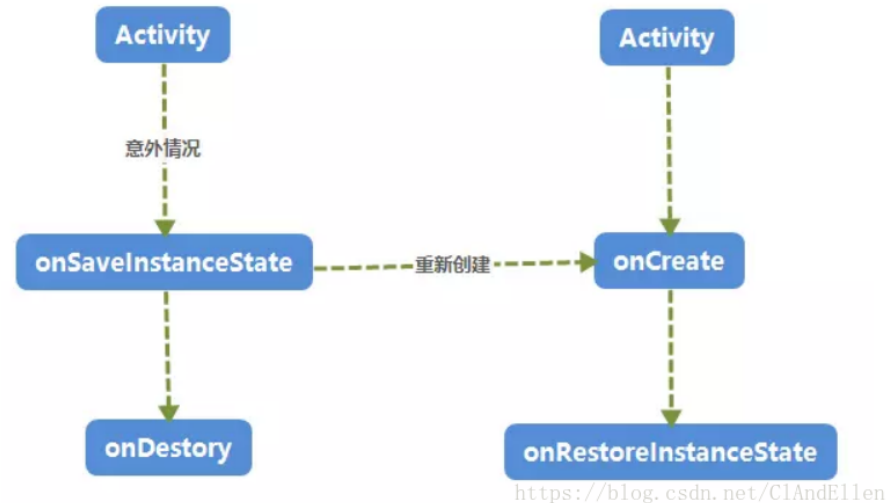
Activity



正常情况下的生命周期：



异常情况下的生命周期(两种)：



1. 情况1：资源相关系统配置发生改变导致Activty被杀死重新创建：手机横竖屏切换、语言切换等等。（可以通过设置"android:configChanges"或相关属性达到不受到情况1的影响）

2. 情况2：资源内存不足导致低优先级的Activity被杀死；（内存不足）

什么时候Activity不执行onDestory()

栈里面的第一个没有销毁的activity会执行ondestroy方法，其他的不会执行。

> 比如说：从mainactivity跳转到activity-A（或者继续从activity-A再跳转到activity-B），这时候，从后台强杀，只会执行mainactivity的onDestroy方法，activity-A（以及activity-B）的onDestroy方法都不会执行；

进程优先级

> 前台>可见>服务>后台>空

前台：与当前用户正在交互的Activity所在的进程。

可见：Activity可见但是没有在前台所在的进程。

服务：Activity在后台开启了Service服务所在的进程。

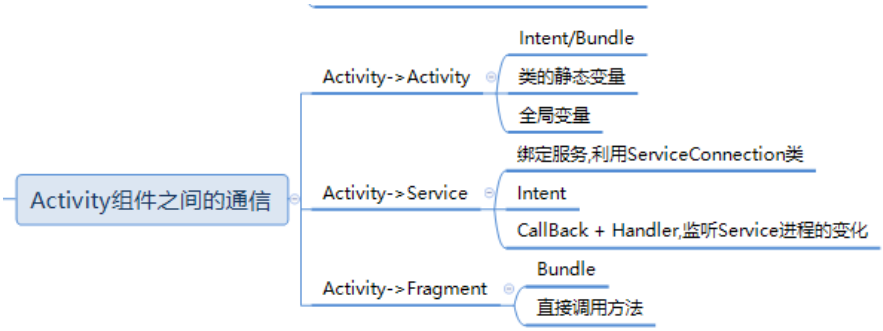
后台：Activity完全处于后台所在的进程。

Activity的启动模式有哪些?

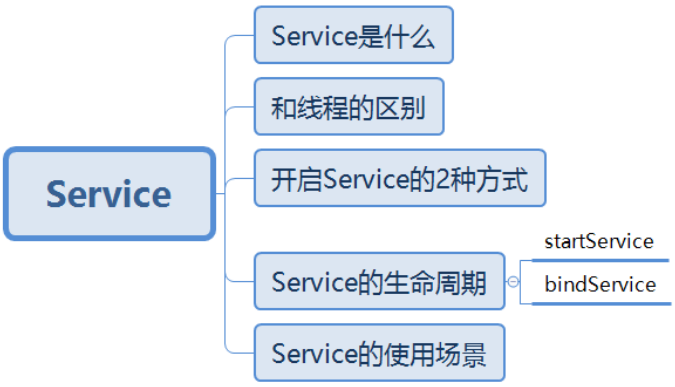
> Activity的启动模式有4种，分别是：standard,singleTop,singleTask和singleInstance。

1. 系统默认的启动模式:Standard,每次启动一个Activity都会重新创建一个新的实例，不管这个实例是否存在。
2. 栈顶复用模式：SingleTop:在这种模式下，如果新的Activity已经位于任务栈的栈顶，那么此Activity不会被重新创建，同时它的onNewIntent方法被回调，通过此方法的参数我们可以取出当前请求的信息。
3. 栈内复用模式：SingleTask:这是一种单例实例模式，在这种模式下，只要Activity在一个栈中存在，那么多次启动此Activity都不会重新创建实例，和singleTop一样，系统也会回调其onNewIntent。
4. 单实例模式：SingleInstance:这是一种加强的singleTask模式，它除了具有singleTask模式所有的特性外，还加强了一点，那就是具有此种模式的Activity只能单独位于一个任务栈中，换句话说，比如Activity A是singleInstance模式，当A启动后，系统会为它创建一个新的任务栈，然后A独自在这个新的任务栈中，由于栈内复用的特性，后续的请求均不会创建新的Activity,除非这个独特的任务栈被系统销毁了。

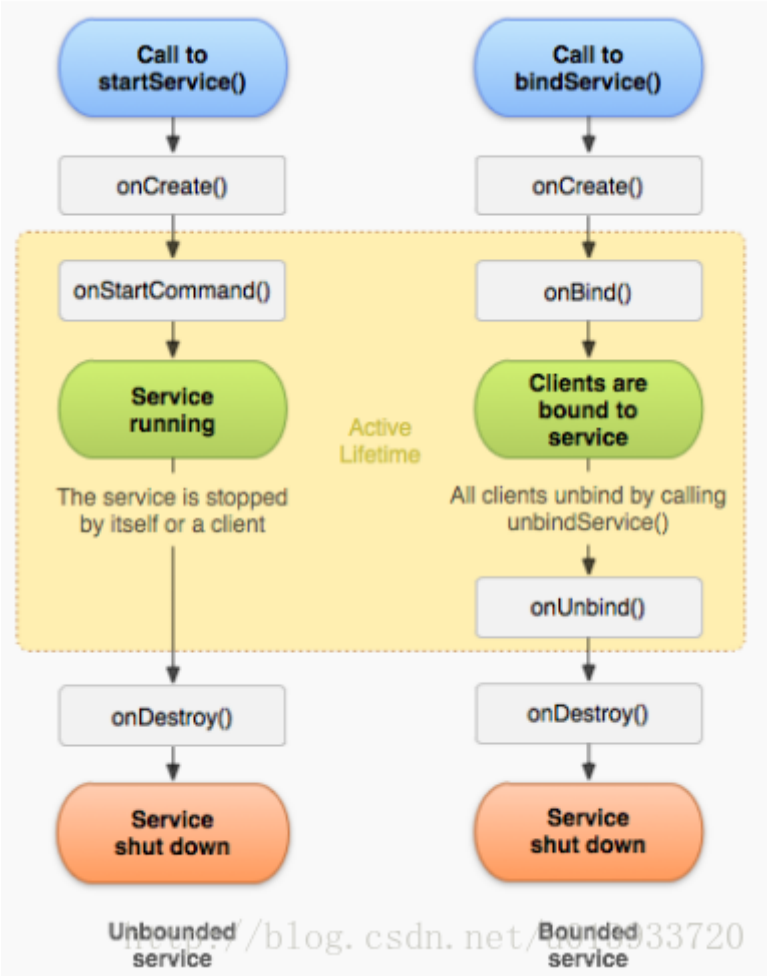
进程之间的通信


Activity组件之间的通信
Activity->Activity
Intent/Bundle
类的静态变量
全局变量
Activity->Service
绑定服务,利用ServiceConnection类
Intent
CallBack + Handler,监听Service进程的变化
Activity->Fragment
Bundle
直接调用方法

Service



生命周期：



a. 被启动的服务的生命周期：如果一个Service被某个Activity 调用Context.startService 方法启动，那么不管是否有Activity使用bindService绑定或unbindService解除绑定到该Service，该Service都在后台运行。如果一个Service被startService 方法多次启动，那么onCreate方法只会调用一次，onStart将会被调用多次（对应调用startService的次数），并且系统只会创建Service的一个实例（因此你应该知道只需要一次stopService调用）。该Service将会一直在后台运行，而不管对应程序的Activity是否在运行，直到被调用stopService，或自身的stopSelf方法。当然如果系统资源不足，android系统也可能结束服务。

b. 被绑定的服务的生命周期：如果一个Service被某个Activity 调用 Context.bindService 方法绑定启动，不管调用 bindService 调用几次，onCreate 方法都只会调用一次，同时onStart方法始终不会被调用。当连接建立之后，Service将会一直运行，除非调用Context.unbindService 断开连接或者之前调用 bindService 的 Context 不存在了（如Activity被finish的时候），系统将会自动停止Service，对应onDestroy将被调用。

c. 被启动又被绑定的服务的生命周期：如果一个Service又被启动又被绑定，则该Service将会一直在后台运行。并且不管如何调用，onCreate始终只会调用一次，对应startService调用多少次，Service的onStart便会调用多少次。调用unbindService将不会停止Service，而必须调用 stopService 或 Service的 stopSelf 来停止服务。

d. 当服务被停止时清除服务：当一个Service被终止（1、调用stopService；2、调用stopSelf；3、不再有绑定的连接（没有被启动））时，onDestroy方法将会被调用，在这里你应当做一些清除工作，如停止在Service中创建并运行的线程。
特别注意：

1、你应当知道在调用 bindService 绑定到Service的时候，你就应当保证在某处调用 unbindService 解除绑定（尽管 Activity 被 finish 的时候绑定会自动解除，并且Service会自动停止）；

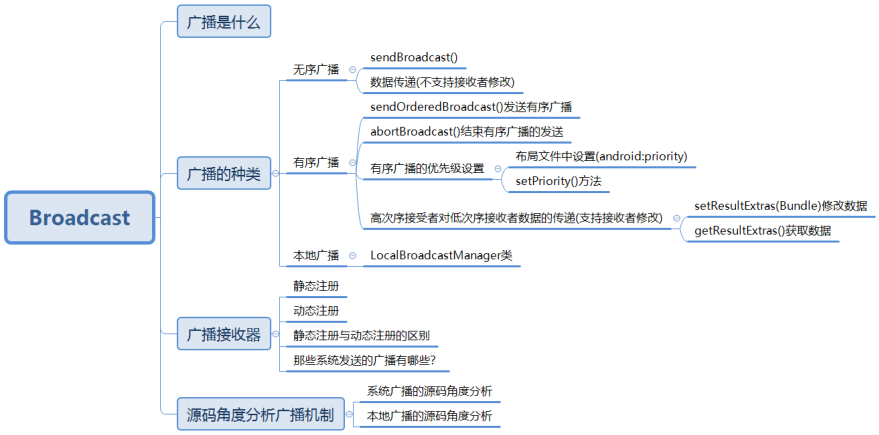
2、你应当注意 使用 startService 启动服务之后，一定要使用 stopService停止服务，不管你是否使用bindService；

3、同时使用 startService 与 bindService 要注意到，Service 的终止，需要unbindService与stopService同时调用，才能终止 Service，不管 startService 与 bindService 的调用顺序，如果先调用 unbindService 此时服务不会自动终止，再调用 stopService 之后服务才会停止，如果先调用 stopService 此时服务也不会终止，而再调用 unbindService 或者 之前调用 bindService 的 Context 不存在了（如Activity 被 finish 的时候）之后服务才会自动停止；

4、当在旋转手机屏幕的时候，当手机屏幕在“横”“竖”变换时，此时如果你的 Activity 如果会自动旋转的话，旋转其实是 Activity 的重新创建，因此旋转之前的使用 bindService 建立的连接便会断开（Context 不存在了），对应服务的生命周期与上述相同。

5、在 sdk 2.0 及其以后的版本中，对应的 onStart 已经被否决变为了 onStartCommand，不过之前的 onStart 任然有效。这意味着，如果你开发的应用程序用的 sdk 为 2.0 及其以后的版本，那么你应当使用 onStartCommand 而不是 onStart。

Broadcast



广播的使用场景

a.同一app内有多个进程的不同组件之间的消息通信。
b.不同app之间的组件之间消息的通信。

广播种类（3种）

1. 无序广播
   context.sendBroadcast(Intent)方法发送的广播，不可被拦截，当然发送的数据，接收者是不能进行修改的。
2. 有序广播
   context.sendOrderBroadcast(Intent)方法发送的广播，可被拦截，而且接收者是可以修改其中要发送的数据，修改和添加都是可以的，这就意味着优先接收者对数据修改之后，下一个接收者接受的数据是上一个接收者已经修改了的，这必须明白。
3. 本地广播
   localBroadcastManager.sendBroadcast(Intent)，只在app内传播。

广播接收器

广播接收器是专门用来接收广播信息的，它可分为静态注册和动态注册：

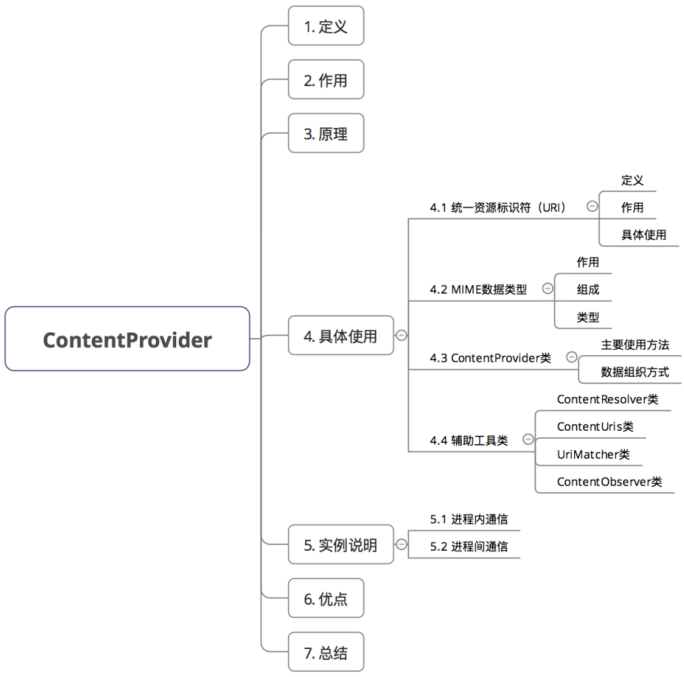
1. 静态注册
   - 首先你要创建一个广播接收器类;
   - 在AndroidManifest.xml文件中注册；
2. 动态注册
   - 新建一个类，让它继承自BroadcastReceiver,并重写父类的onReceive()方法；
   - 代码中创建intentFilter：

```
intentFilter = new IntentFilter();
intentFilter.addAction("android.net.conn.CONNECTIVITY_CHANGE");
networkChangeReceiver = new NetWorkChangeReceiver();
registerReceiver(networkChangeReceiver, intentFilter);//注册广播接收器
```

优点：动态注册的广播接收器可以自由地控制注册与注销，在灵活性方面有很大优势；
缺点：必须要在程序启动之后才能接收到广播，因为注册的逻辑是写在`onCreate()`方法中的。那么有没有广播能在程序未启动的情况下就能接收到广播呢？静态注册的广播接收器就可以做到。

ContentProvider

参考链接2： https://blog.csdn.net/carson_ho/article/details/76101093

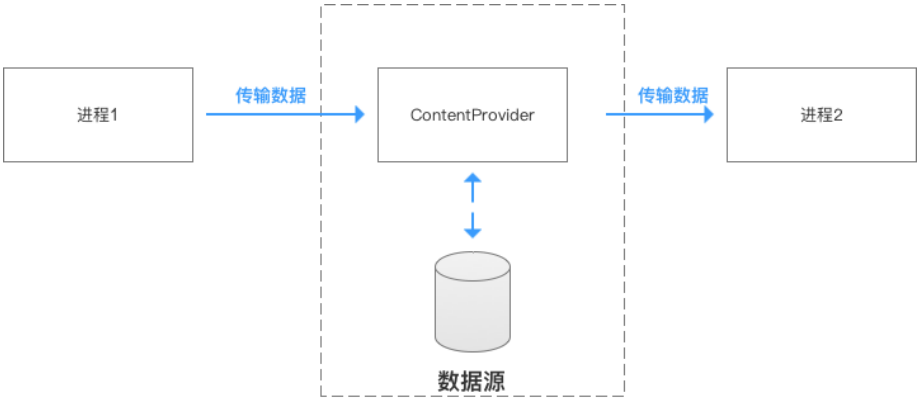


什么是**ContentProvider**：

是Android的四大组件之一；
主要用于不同的应用程序之间实现数据共享功能；

作用

进程间 进行数据交互 & 共享，即跨进程通信



注：
1. ContentProvider = 中间者角色（搬运工），真正 存储&操作数据的数据源还是原来存储数据的方式（数据库、文件、xml或网络）
2. 数据源可以：数据库（如Sqlite）、文件、XML、网络等等

什么是ContentResolver：

是数据调用者，ContentProvider将数据发布出来，通过ContentResolver对象结合Uri进行调用,通过ContentResolver对象可以调用ContentProvider的增删改查;

什么是Uri：

Uri（通用资源标识符 Universal Resource Identifer），代表数据操作的地址，每一个ContentProvider发布数据时都会有唯一的地址。
比如：content：//（固定写法）+com.android.contacts(包名，可变)+/contacts（path路径）

MIME数据类型

- 作用：指定某个扩展名的文件用某种应用程序来打开。如指定.html文件采用text应用程序打开、指定.pdf文件采用flash应用程序打开；

ContentProvider类

组织数据方式

- ContentProvider主要以 表格的形式 组织数据,同时也支持文件数据，只是表格形式用得比较多;
- 每个表格中包含多张表，每张表包含行 & 列，分别对应记录 & 字段,同数据库;

主要方法

- 进程间共享数据的本质是：添加、删除、获取 & 修改（更新）数据;

- 所以ContentProvider的核心方法也主要是上述4个作用：

```
<-- 4个核心方法 -->
  public Uri insert(Uri uri, ContentValues values)
  // 外部进程向 ContentProvider 中添加数据

  public int delete(Uri uri, String selection, String[]
selectionArgs)
  // 外部进程 删除 ContentProvider 中的数据

  public int update(Uri uri, ContentValues values, String
selection, String[] selectionArgs)
  // 外部进程更新 ContentProvider 中的数据

  public Cursor query(Uri uri, String[] projection, String
selection, String[] selectionArgs,  String sortOrder)
  // 外部应用 获取 ContentProvider 中的数据

// 注：
  // 1. 上述4个方法由外部进程回调，并运行在ContentProvider进程的
Binder线程池中（不是主线程）
 // 2. 存在多线程并发访问，需要实现线程同步
   // a. 若ContentProvider的数据存储方式是使用SQLite & 一个，则
不需要，因为SQLite内部实现好了线程同步，若是多个SQLite则需要，因为
SQL对象之间无法进行线程同步
  // b. 若ContentProvider的数据存储方式是内存，则需要自己实现线程
同步

<-- 2个其他方法 -->
public boolean onCreate()
// ContentProvider创建后 或 打开系统后其它进程第一次访问该
ContentProvider时 由系统进行调用
// 注：运行在ContentProvider进程的主线程，故不能做耗时操作

public String getType(Uri uri)
// 得到数据类型，即返回当前 Url 所代表数据的MIME类型
```

- Android为常见的数据（如通讯录、日程表等）提供了内置了默认的ContentProvider
- 但也可根据需求自定义ContentProvider，但上述6个方法必须重写；
- ContentProvider类并不会直接与外部进程交互，而是通过 ContentResolver 类；

ContentResolver类

作用

统一管理不同 ContentProvider间的操作：
1. 即通过 URI 即可操作 不同的ContentProvider 中的数据；
2.外部进程通过 ContentResolver类 从而与ContentProvider类进行交互；

为什么要使用通过**ContentResolver**类从而与**ContentProvider**类进行交互，而不直接访问**ContentProvider**类？

```
一般来说，一款应用要使用多个ContentProvider，若需要了解每个ContentProvider的不同实现从而再完成数据交互，操作成本高 & 难度大，所以再ContentProvider类上加多了一个 ContentResolver类对所有的ContentProvider进行统一管理。
```

Android 提供了3个用于辅助ContentProvide的工具类：

```
ContentUris
UriMatcher
ContentObserver
```

ContentUris类

作用：操作 URI
具体使用
核心方法有两个：withAppendedId（） &parseId（）

```
// withAppendedId（）作用：向URI追加一个id
Uri uri = Uri.parse("content://cn.scu.myprovider/user")
Uri resultUri = ContentUris.withAppendedId(uri, 7);
// 最终生成后的Uri为：content://cn.scu.myprovider/user/7
// parseId（）作用：从URL中获取ID
Uri uri = Uri.parse("content://cn.scu.myprovider/user/7")
long personid = ContentUris.parseId(uri);
//获取的结果为:7
```

UriMatcher类

作用:

- 在ContentProvider 中注册URI
- 根据 URI 匹配 ContentProvider 中对应的数据表
  具体使用:

```
// 步骤1：初始化UriMatcher对象
    UriMatcher matcher = new
UriMatcher(UriMatcher.NO_MATCH);
    //常量UriMatcher.NO_MATCH  = 不匹配任何路径的返回码
    // 即初始化时不匹配任何东西

// 步骤2：在ContentProvider 中注册URI（addURI（））
    int URI_CODE_a = 1;
    int URI_CODE_b = 2;
    matcher.addURI("cn.scu.myprovider", "user1",
URI_CODE_a);
    matcher.addURI("cn.scu.myprovider", "user2",
URI_CODE_b);
```

```
    // 若URI资源路径 = content://cn.scu.myprovider/user1 ，
则返回注册码URI_CODE_a
    // 若URI资源路径 = content://cn.scu.myprovider/user2 ，
则返回注册码URI_CODE_b

// 步骤3：根据URI 匹配 URI_CODE，从而匹配ContentProvider中相应
的资源（match（））

@Override
    public String getType(Uri uri) {
      Uri uri = Uri.parse("
content://cn.scu.myprovider/user1");

      switch(matcher.match(uri)){
     // 根据URI匹配的返回码是URI_CODE_a
     // 即matcher.match(uri) == URI_CODE_a
      case URI_CODE_a:
        return tableNameUser1;
        // 如果根据URI匹配的返回码是URI_CODE_a，则返回
ContentProvider中的名为tableNameUser1的表
      case URI_CODE_b:
        return tableNameUser2;
        // 如果根据URI匹配的返回码是URI_CODE_b，则返回
ContentProvider中的名为tableNameUser2的表
    }
}
```

ContentObserver类

定义：内容观察者

- 作用：观察 Uri引起 ContentProvider 中的数据变化 & 通知外界（即访问该数据访问者）
  当ContentProvider 中的数据发生变化（增、删 & 改）时，就会触发该 ContentObserver类
  具体使用:

```
// 步骤1：注册内容观察者ContentObserver
    getContentResolver().registerContentObserver（uri）；
    // 通过ContentResolver类进行注册，并指定需要观察的URI

// 步骤2：当该URI的ContentProvider数据发生变化时，通知外界（即访
问该ContentProvider数据的访问者）
    public class UserContentProvider extends
ContentProvider {
      public Uri insert(Uri uri, ContentValues values) {
      db.insert("user", "userid", values);
      getContext().getContentResolver().notifyChange(uri,
null);
      // 通知访问者
   }
```

```
}

// 步骤3：解除观察者
 getContentResolver().unregisterContentObserver（uri）；
    // 同样需要通过ContentResolver类进行解除
```

创建自定义ContentProvider的步骤：

```
1.使用SQLite技术，创建好数据库和数据表
2.新建类继承ContentProvider
3.重写6个抽象方法
4.创建UriMatcher，定义Uri规则
5.在Manifest中注册provider
6.ContentResolver对ContentProvider中共享的数据进行增删改查操作
```

四大组件学习心得

```
1. 更加深入的理解了四大组件的实现机制，以及很多之前不清楚的知识，基础知识得以充实；
2. 其中对ContentProvider这边还是有点模糊，还需要自己练习一下；
3. 最深的是Activity的异常生命周期，之前工作的时候就遇到过类似情况，由于都不是专业的Android开发，语言切换导致走到了异常生命周期，当时那个项目排查此问题花费的很长时间；
4. Android基础还是需要多多看；
```