目录

[语言 2](#_Toc495418790)

[Java 2](#_Toc495418791)

[C++ 2](#_Toc495418792)

[Js 2](#_Toc495418793)

[数据结构 2](#_Toc495418794)

[算法 2](#_Toc495418795)

[算法基础 2](#_Toc495418796)

[1bitmap 2](#_Toc495418797)

[机器学习 3](#_Toc495418798)

[1贝叶斯 3](#_Toc495418799)

[2svm 3](#_Toc495418800)

[web框架 3](#_Toc495418801)

[Spring 3](#_Toc495418802)

[Struct2 4](#_Toc495418803)

[协议 4](#_Toc495418804)

[工具 4](#_Toc495418805)

[Git 4](#_Toc495418806)

[Maven 6](#_Toc495418807)

[数据库 6](#_Toc495418808)

[数据相关操作 6](#_Toc495418809)

[常用命令 8](#_Toc495418810)

[操作系统 8](#_Toc495418811)

[VirtualBox安装centos学习 8](#_Toc495418812)

# 语言

## Java

Jar文件打开方式

$ Java –jar xxx.jar

类的加载过程

1加载

通过一个类的全限定名来获取定义此类的二进制字节流。

将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。

在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

2验证

Java语言本身是相对安全的语言，使用Java编码是无法做到如访问数组边界以外的数据、将一个对象转型为它并未实现的类型等，如果这样做了，编译器将拒绝编译。但是，Class文件并不一定是由Java源码编译而来，可以使用任何途径，包括用十六进制编辑器(如UltraEdit)直接编写。如果直接编写了有害的“代码”(字节流)，而虚拟机在加载该Class时不进行检查的话，就有可能危害到虚拟机或程序的安全。

不同的虚拟机，对类验证的实现可能有所不同，但大致都会完成下面四个阶段的验证：文件格式验证、元数据验证、字节码验证和符号引用验证。

3准备

准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在Java堆中。

public static int value=123;//在准备阶段value初始值为0 。在初始化阶段才会变为123 。

4解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用（Symbolic Reference）：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用与虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。

直接引用（Direct Reference）：直接引用可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄。直接引用是与虚拟机实现的内存布局相关的，如果有了直接引用，那么引用的目标必定已经在内存中存在。

5初始化

类初始化是类加载过程的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

初始化阶段是执行类构造器<clinit>()方法的过程。<clinit>()方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块(static{}块)中的语句合并产生的。

class SingleTon {

    private static SingleTon singleTon = new SingleTon();

    public static int count1;

    public static int count2 = 0;

    private SingleTon() {

        count1++;

        count2++;

    }

    public static SingleTon getInstance() {

        return singleTon;

    }

}

public class Test {

    public static void main(String[] args) {

        SingleTon singleTon = SingleTon.getInstance();

        System.out.println("count1=" + singleTon.count1);

        System.out.println("count2=" + singleTon.count2);

    }

}

结果为1,0分析:

1:SingleTon singleTon = SingleTon.getInstance();调用了类的SingleTon调用了类的静态方法，触发类的初始化  
2:类加载的时候在准备过程中为类的静态变量分配内存并初始化默认值 singleton=null count1=0,count2=0  
3:类初始化化，为类的静态变量赋值和执行静态代码快。singleton赋值为new SingleTon()调用类的构造方法  
4:调用类的构造方法后count=1;count2=1  
5:继续为count1与count2赋值,此时count1没有赋值操作,所有count1为1,但是count2执行赋值操作就变为0

## C++

## Js

# 数据结构

# 算法

## 算法基础

## 1bitmap

所谓的Bit-map就是用一个bit位来标记某个元素对应的Value， 而Key即是该元素。由于采用了Bit为单位来存储数据，因此在存储空间方面，可以大大节省。

来看一个具体的例子，假设我们要对0-7内的5个元素(4,7,2,5,3)排序（这里假设这些元素没有重复）。那么我们就可以采用Bit-map的方法来达到排序的目的。要表示8个数，我们就只需要8个Bit（1Bytes），首先我们开辟1Byte的空间，将这些空间的所有Bit位都置为0

然后遍历这5个元素，首先第一个元素是4，那么就把4对应的位置为1（可以这样操作 p+(i/8)|(0×01<<(i%8)) 当然了这里的操作涉及到Big-ending和Little-ending的情况，这里默认为Big-ending）,因为是从零开始的，所以要把第五位置为1

然后再处理第二个元素7，将第八位置为1,，接着再处理第三个元素，一直到最后处理完所有的元素，将相应的位置为1

然后我们现在遍历一遍Bit区域，将该位是一的位的编号输出（2，3，4，5，7），这样就达到了排序的目的

## 机器学习

### 1贝叶斯

### 2svm

# web框架

## Spring

1介绍

2原理

## Struct2

# 协议

# 工具

## Git

1初步介绍

http://www.ruanyifeng.com/blog/2015/12/git-cheat-sheet.html

git status

Untracked files 未跟踪文件

git add abc.txt

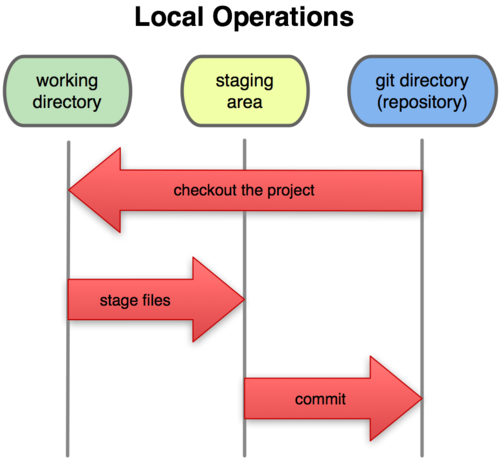
git status

Changes to be committed 在暂存区域生成了快照，等待被提交

Git commit 或git commit -m "first commit"

对于任何一个文件，在 Git 内都只有三种状态：‍‍‍‍**‍**‍**已提交（committed），已修改（modified）和已暂存（staged）**‍**‍。已提交表示该文件已经被安全地保存在本地数据库中了；已修改表示修改了某个文件，但还没有提交保存；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。**‍‍‍‍

由此我们看到 Git 管理项目时，文件流转的三个工作区域：**Git 的工作目录，暂存区域，以及本地仓库**



每个项目都有一个 **Git 目录**（译注：如果 git clone 出来的话，就是其中 .git 的目录；如果 git clone --bare 的话，新建的目录本身就是 Git 目录。），它是 Git 用来保存元数据和对象数据库的地方。该目录非常重要，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。

从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做**工作目录**。这些文件实际上都是从 Git 目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。

所谓的**暂存区域只不过是个简单的文件，一般都放在 Git 目录中**。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。

基本的 Git 工作流程如下：

* 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(commit)，将保存在暂存区域的文件快照永久转储到 Git 目录中。

所以，我们可以从文件所处的位置来判断状态：如果是 Git 目录中保存着的特定版本文件，就属于已提交状态；如果作了修改并已放入暂存区域，就属于已暂存状态；如果自上次取出后，作了修改但还没有放到暂存区域，就是已修改状态。

Git push 真正推到远程代码服务器

## Maven

1初步介绍

mvn dependency:tree  查看依赖树

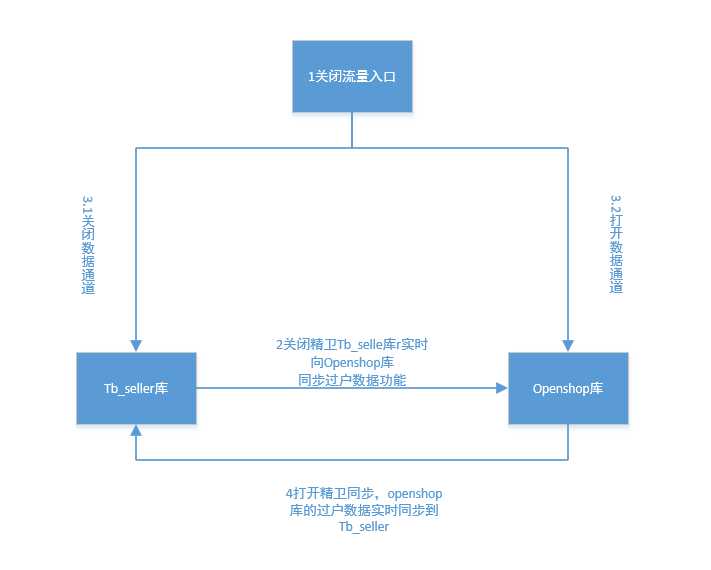
mvn dependency:tree -Doutput=\*.txt

windows 默认settings.xml地址

# 数据库

## 数据相关操作

1切DB



关闭精卫同步任务，完成数据迁移，分成以下几个步骤

1关闭精卫同步任务，开启老DB:TB\_seller向新DB:OpenShop的全量迁移，简称TB和Op

2 TB向Op迁移任务完成

3开始TB向Op的全量校验，

4完成校验，发现Op有46条缺失，12条差异，下载保存。手工抽取几条查看是否正确。

5全量订正Op，选择回源订正

6订正完成

7 Op向TB反向全量校验

8 完成全量校验，发现TB有107条数据缺失，即Op多了107条数据，应该是以前删除老数据库数据时，新数据库没有同步到删除的数据，下载保存，手工抽取几条查看正确

9 全量订正Op，此时Op是源数据。

10 订正完成

11   打开新的精卫同步任务，分为以下两个步骤

    12  时间1:25打开Op向Tb的精卫同步，将Tb作为备数据库

13  时间1:27-1:40预发分别测试过户和升级，数据流入新数据库Op，check发现能同步到老数据库Tb，即同步任务ok

# 常用命令

改完hosts刷新           ipconfig /flushdns

tail –f100 log/production.log

grep  -n 'test'  seller.log

$sed -n '234689,234900p' seller-center.log

sed -n '/2017-07-17 13:47:20/,/2017-07-17 13:47:25/p'  seller-center.log

# 操作系统

## VirtualBox安装centos学习

dky112358

1 知识点

Linux是/根目录，所以目录或文件名区分大小写

已.开头的是隐藏文件

相对路径/usr/log/sa和绝对路径../../usr/log/sa

2 常用命令

$ 当前登录用户

# root用户

命名+选项+参数

Hostname

Uanme or uname –r or uname -a  
firefox 直接打开火狐

Firefox & 后台运行火狐，命令行能继续其他操作

history 查看之前命令

!! 重复前一个命令

！字符 重复前一个已“”字符“”开头的命令

！num 按照历史记录的序号执行命令

！？abc 重复之前包含abc的命令

！-n 重复n个命令之前的那个命令

Ctrl+r 在历史记录中搜索命令

Esc . 按esc后按.键 重复调用前一个命令的参数

Su- 切换到root用户并使用新的运行环境

Su 切换到root

Sudo 使用管理员用户身份运行命令

Id 显示当前用户信息

Passwd 修改当前用户密码

Jobs bg fg管理后台作业

Jobs 当前在后台运行的所以作业

Sleep 5000 停止5000秒

Ctrl+z 暂停当前作业

Bg 1 sleep继续后台运行 1代表作业编号

Fg 1 sleep在前台运行

Pwd 显示当前工作目录（每一个shell或系统进程都有一个当前工作目录）

Touch 创建一个空白文件或者更新已有文件的时间

Ls –a 显示所有文件（包括隐藏文件）

Ls –l 显示详细信息

Ls –R 递归显示子目录结构

Ls –ld 显示目录和链接信息

File abc。Txt 查看文件类型

Cp 源文件 目标文件/目标目录 复制文件或目录

Cp –r 文件夹 目标文件/目标目录

Cp –rv 加v显示详细复制信息

或Cp –r –v

Mv 文件 目录 移动文件 如 mv a opt

Mv 文件 目录/新命名 如 mv a opt/b 将a移到opt下，重命名为b

mv a b a重命名为b，当前目录

rm xx删除当前某文件

rm –r dic 递归删除dic目录

rm –ri dic i交互式删除 每一个删除都询问

rm –rif dic 强制删除 不会有交互式询问 –f

mkdir 创建一个目录

rmdir xx 删除空目录，若目录下有文件，无法删除

rm –r xx 删除非空目录

date用来查看、设置当前设置时间

可以格式化显示 date +%Y--%m--%d 2017—11—16

Date +%Y:%m:%d 2017:11:16

date s “20:20:20”设置时间 需要先su到root

clock hwclock 显示硬件时钟时间

cal 查看日历

uptime 启动时间 启动了多长时间

echo “”xx“” 显示输入内容 xx

cat 显示文件内容 全部显示

more 用于翻页显示文件内容（只能向下）

less 用于翻页显示文件内容（可以上下翻）

head 用于显示前10行（默认）

-n 指定显示行数 如 head –n 3 xx.txt

Tail 显示文件的末尾几行（10）

-n 指定显示行数

-f 最终显示文件更新 如 tail –f xx.log

Log增加时，终端会不断显示新加入的字段

3linux目录

**/bin**  
bin是binary的缩写。这个目录沿袭了UNIX系统的结构，存放着使用者最经常使用的命令。例如cp、ls、cat，等等。

**/boot**  
这里存放的是启动Linux时使用的一些核心文件。

**/dev**  
dev是device（设备）的缩写。这个目录下是所有Linux的外部设备，其功能类似DOS下的.sys和Win下的.vxd。在Linux中设备和文件是用同种方法访问的。例如：/dev/hda代表第一个物理IDE硬盘。

**/etc**  
这个目录用来存放系统管理所需要的配置文件和子目录。

**/home**  
用户的主目录，比如说有个用户叫wang，那他的主目录就是/home/wang也可以用~wang表示。

**/lib**  
这个目录里存放着系统最基本的动态链接共享库，其作用类似于Windows里的.dll文件。几乎所有的应用程序都须要用到这些共享库。

**/lost+found**  
这个目录平时是空的，当系统不正常关机后，这里就成了一些无家可归的文件的避难所。对了，有点类似于DOS下的.chk文件。

**/mnt**  
这个目录是空的，系统提供这个目录是让用户临时挂载别的文件系统。

**/proc**  
这个目录是一个虚拟的目录，它是系统内存的映射，我们可以通过直接访问这个目录来获取系统信息。也就是说，这个目录的内容不在硬盘上而是在内存里。

**/root**  
系统管理员（也叫超级用户）的主目录。作为系统的拥有者，总要有些特权啊！比如单独拥有一个目录。

**/sbin**  
s就是Super User的意思，也就是说这里存放的是系统管理员使用的管理程序。

**/tmp**  
这个目录不用说，一定是用来存放一些临时文件的地方了。

**/var**  
这个目录中存放着那些不断在扩充着的东西，为了保持/usr的相对稳定，那些经常被修改的目录可以放在这个目录下，实际上许多系统管理员都是这样干的。顺带说一下系统的日志文件就在/var/log目录中。

**/usr**  
这是最庞大的目录，我们要用到的应用程序和文件几乎都存放在这个目录下。其中包含以下子目录；

**/usr/X11R6**  
存放X-Window的目录；

**/usr/bin**  
存放着许多应用程序；

**/usr/sbin**  
给超级用户使用的一些管理程序就放在这里；

**/usr/doc**  
这是Linux文档的大本营；

**/usr/include**  
Linux下开发和编译应用程序需要的头文件，在这里查找；

**/usr/lib**  
存放一些常用的动态链接共享库和静态档案库；

**/usr/local**  
这是提供给一般用户的/usr目录，在这里安装软件最适合；

**/usr/man**  
man在Linux中是帮助的同义词，这里就是帮助文档的存放目录；

**/usr/src**  
Linux开放的源代码就存在这个目录，爱好者们别放过

## Android

dalvik虚拟机定义了一堆的smali指令（256个），这些字节码指令高度抽象，组合这些指令可以完成我们想要的功能。字节码是用简单的二进制数字表示的，与可阅读的smali指令存在对应关系；

smali/baksmali，这个工具可以分析dex文件，解析字节码为对应的smali语法（反汇编），同时也可以将smali语法的文件重新转换为字节码生成dex文件（汇编）。

Java ->class文件->合成dex文件

Dex+资源->打包apk

安装apk： PackageManager优化dex为odex，存放在/data/dalvik-cache目录下。dex优化过程，其实是将一些字节码替换为dalvik相关的、优化后的等价字节码。

dex文件是遵从于dalvik虚拟机标准的文件，它具有跨dalvik虚拟机的特点，而odex是在特定dalvik虚拟机上优化得到的，通常不能跨dalvik虚拟机运行。

在dex文件中，方法体里面的内容最终存储在classData区域，方法体里面存储的是二进制的字节码。

前面已经知道，一个java方法在虚拟机里面对应的Method为directMethod，其insns指向了字节码位置。

Xposed在对java方法进行hook时，先将虚拟机里面这个方法的Method改为nativeMethod(其实就是一个标识字段)，然后将该方法的nativeFunc指向自己实现的一个native方法，这样方法在调用时，就会调用到这个native方法，接管了控制权。

在这个native方法中，xposed直接调用了一个java方法，这个java方法里面对原方法进行了调用，并在调用前后插入了钩子，于是就hook住了这个方法。

Xposed的hook原理就是这么简单，但它有其他的问题要解决：如何将hook的代码注入到目标app的进程中？

作者：difcareer  
链接：https://www.jianshu.com/p/b29a21a162ad  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。