
Table of Contents

Introduction	1.1
git	1.2
gitbook	1.3
dos	1.4
python	1.5
error-code	1.6
cache	1.7
spring	1.8
java	1.9
leetCode	1.10

Notebook

How to use git in Linux?

- 在同一台电脑上配置多个git账户

在同一台电脑上配置多个git账户

1.首先在`~/.ssh`目录下执行

```
ssh-keygen -t rsa -C "miaoying.new@qq.com"
```

其中 `-C "miaoying.new@qq.com"` 可以不加。如果加上，则在最后生成的`myself_id_rsa.pub`文件内容的末尾会带上`miaoying.new@qq.com`；如果不加，则`myself.id.rsa.pub`文件内容的末尾会加上当前设备的登录用户名和设备名。

根据提示输入文件名（我输入的是`myself_id_rsa`，文件名随意取），之后可以看到生成了两个文件：

```
myself_id_rsa  myself_id_rsa.pub
```

其中，`myself_id_rsa`存放的是私钥，`myself_id_rsa.pub`存放的是公钥。

2.将公钥添加到github的SSH keys列表里，即表示该github账户可以允许含有该SSH的设备进行读写操作，把该SSH文件拷贝到其他设备上，其他设备也可以对项目进行读写操作。

3.配置好后，该设备上就有两个github账户，需要对项目进行账户指定，即允许哪些用户对项目进行git操作，例如项目Demo，只允许用户名为`zhangsan`，邮箱为`zhangsan@qq.com`进行操作，那么在Demo项目根目录下执行（用户名和邮箱随意取，因为git项目信任的是SSH key，而不是用户名）

```
git config user.name zhangsan
git config user.email zhangsan@qq.com
```

另外，同一台设备上可以生成多个SSH，也就是说以上操作可重复执行多次。

How to use gitbook in Linux?

- 安装
- 使用

安装

1. 首先安装nodejs

```
sudo apt-get install nodejs  
sudo apt-get install npm  
npm install gitbook-pdf -g
```

2. 由于生成pdf文件依赖于ebook-convert，故首先安装ebook-convert：<http://calibre-ebook.com/download>

3. 安装gitbook

```
npm install gitbook-cli -g
```

4. 安装pdf转换工具 https://calibre-ebook.com/download_linux

使用

1. 将项目clone到本地后，在项目根目录下执行：gitbook init，则会生成相应的文件：SUMMARY.md 和 README.md 其中，SUMMARY.md是用来存放书的目录的，可以对SUMMARY.md进行编写，但是，对于目录的每一个链接必须有实体文件，否则点击无效。

2. 对md文件进行编辑保存后，使用gitbook pdf命令生成书即可。



DOS命令

- DOS简介
- 批处理 (Batch)
- 批处理语法 (不间断更新)
 - echo
 - rem
 - pause
 - call
 - start
 - goto
 - set
 - 管道符号 |
 - 逻辑命令符

DOS简介

1.dos是磁盘操作系统（Disk Operation System）的缩写，是个人计算机上的一类单用户单任务的操作系统。微软所有的后续版本中，磁盘操作系统仍然被保留着。微软图形界面操作系统Windows NT问世以来，DOS就是以一个后台程序的形式出现，可以通过点击运行CMD进入运行。

2.dos操作系统用户指令不区分大小写。

3.完整的dos组成（5个部分）

- a.引导程序（BOOT）：由格式化程序直接写入磁盘初始扇区。
- b.基本输入/输出管理程序（PC-DOS为IBMBIO.COM、MS-DOS为IO.SYS）。
- c.文件管理和系统调用程序（PC-DOS为IBMDOS.COM、MS-DOS为MSDOS.SYS）。
- d.命令处理程序（COMMAND.COM）。
- e.各种外部命令：完成各种辅助功能的可执行文件

4.dos是命令模式下的人机交互界面，人通过这个界面来运行和控制计算机。另外，dos作为操作系统能有效地管理、调度、运行个人计算机各种软件和硬件资源。

5.Windows在“附件”中有一个“命令提示符”（CMD），其模拟了一个DOS环境，可以使用相关的命令来对计算机和网络进行操作。

批处理 (Batch)

1.批处理也称为批处理脚本，也称作宏，是对某对象进行批量的处理，通常被认为是一种简化的脚本语言，应用于DOS和Windows系统中。类似于Unix的Shell脚本，由DOS和Windows系统内的命令解释器（COMMAND.COM 或者 CMD.EXE）解释运行。批处理文件的扩展名为bat。目前常见的批处理包含两类：DOS批处理（基于DOS命令，用来自动地批量地执行DOS命令以实现特定操作的脚本）和PS批处理（基于图片编辑软件PhotoShop，用来批量处理图片的脚本）。

2.批处理程序虽然是在命令行环境中运行，但不仅仅能使用命令行软件，任何当前系统下可运行的程序都可以放在批处理文件中运行。

3.系统在解释运行批处理程序时，首先扫描整个批处理程序，然后从第一行代码开始向下逐句执行所有的命令，直至程序结尾遇见exit命令或出错意外退出。

批处理语法（不间断更新）

echo

打开回显或关闭回显功能，或显示消息

```
echo [{on|off}] [message]
```

sample:

```
:: 关闭回显  
echo off  
  
:: 开启回显  
echo on  
  
:: 默认开启  
echo  
  
:: 显示消息 "hello world"  
echo "hello world"
```

在实际应用中，我们会把这条命令和重定向符号（也称为管道符号，一般用>>>^）结合起来实现输入一些命令到特定的文件中。

rem

注释命令，类似于在java中的//，但它并不会被执行，只是起到一个注释作用，只有在编辑批处理时才会被看到，主要便于修改。

```
rem [注释消息]
```

sample:

```
rem 这是我的注释消息
```

::也有rem的功能，以下是区别：

在关闭回显时，rem和::后的內容都不会显示
打开回显时，rem后的內容回显示出来，然而::后的內容仍然不会显示出来

pause

暂停命令，运行pause命令时，将显示：Press any key to continue...（或：请按任意键继续...）

sample:

```
@echo off
echo 接下来会显示“请按任意键继续...”
pause
```

call

从一个批处理命令调用另一个批处理命令，并且不终止父批处理程序。如果在脚本或批处理外使用call，他将不会在命令行起作用。

```
:: 指定要调用的批处理的位置和名称
call [路径文件名]
```

start

调用外部程序，所有的dos命令和命令行程序都可以由start命令来调用。

sample:

```
::打开Windows的计算器
start calc.exe
```

start命令的常用参数：

min	开始时窗口最小化
high	在high优先级类别开始应用程序
separate	在分开的空间内开始16位Windows程序
realtime	在realtime优先级类别开始应用程序
wait	启动应用程序并等候它结束
parameters	传送到命令/程序的参数

执行的应用程序是32位 GUI 应用程序时，CMD.EXE 不等应用程序终止就返回命令提示。如果在命令脚本内执行，则新行为则不会发生。

goto

跳转命令。程序指针跳转到指定的标签，从标签后的第一条命令开始继续执行批处理。

sample:

```
:: 跳转到test  
goto test  
  
:test
```

set

显示、设置或删除变量。 1.显示变量：

```
set 或 set s 前者显示批处理当前已定义的所有变量及其值，后者显示所有以s开头的变量及值。
```

2.设置变量：

```
set aa=abcd
```

3.删除变量：

```
:: 若变量已被定义，则删除变量；若aa尚未被定义，则此命令无实际意义  
set aa=
```

批处理中的变量是不区分类型的，比如执行set aa=345后，变量aa的值既可以被视为数字345，也可以被视为字符串345。set命令具有扩展功能，如用作交互输入、字符串处理、数值计算等，属于高级命令范畴。

管道符号 |

将管道符号前面命令的输出结果重定向输出到管道符号后面的命令中去，作为后面命令的输入。使用格式为： command1 | command2

sample:

```
:: 在询问是否删除文件a.txt时，直接显示y（即表示同意删除）  
@echo off  
echo aaa>a.txt  
echo y|del /p a.txt
```

逻辑命令符

逻辑命令符包括：& && ||

```
& -- 用来连接n个DOS命令，并把这些命令按顺序执行，而不管是否有命令执行失败
&& -- 当前面的命令执行成功时才会执行后续的命令，否则不执行
|| -- 当前面的命令失败时才会执行后续的命令，否则不执行
```

python

- 简介

简介

python 是一种高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

1. 解释性：在开发过程中没有了编译环节。
2. 交互性：可以在一个Python提示符直接互动执行写程序。
3. 面向对象：支持面向对象的风格或代码封装在对象的编程技术。

优点

易于学习
易于维护
易于阅读
一个广泛的标准库
互动模式
可移植
可扩展
数据库（python提供所有主要的商业数据库的接口）
GUI编程
可嵌入（python可嵌入到c/c++程序中，使程序的用户获得脚本化的能力）

error code (记录遇到的错误码)

- 502 Bad Gateway

502 Bad Gateway

情况描述 项目发布时使用同事编写的脚本进行release发布到服务器，由于项目配置文件中端口号一直是8084,(某个地方配置了项目发布到外网的端口一定是8084，具体未知)，然后由于本地有个服务占用了8084，所以把该项目端口换成了8083，发布到服务器后就报错502 Bad Gateway.

错误码描述 502 Bad Gateway是指错误网关，无效网关，在互联网中表示一种网络错误，表现在web浏览器中给出的页面反馈。但这不意味上游服务器已关闭（无响应网关/代理），而是上游服务器和网关/代理不同意的协议交换数据，往往意味着一个或两个机器已不正确或不完全编程。

一般出现了这个问题是由于不良的IP之间的沟通后端计算机，包括可能尝试访问的在web服务器上的网站。
(分析问题前需要完全清除浏览器缓存)

cache (缓存相关)

- 刷新

刷新

1. 基本刷新

点击刷新或者使用F5快捷键

基本刷新有可能只是从本地的硬盘重新拿取数据到浏览器，并不一定重新向服务器发出请求。

2. 从服务器刷新

重新直接点击链接（快捷键Ctrl + F5）

实际上会从服务器重新下载数据。

- 创建一个Spring项目

创建一个**Spring**项目

1. 首先创建一个项目仓库（SpringDemo）

```
mkdir SpringDemo
```

2. 创建src目录，src/main/java, src/main/resources, src/test/java, src/test/resources

```
mkdir src
mkdir -p src/main
mkdir -p src/main/java
mkdir -p src/main/resources
mkdir -p src/test/java
mkdir -p src/test/resources
```

3. 使用gradle进行初始化

```
grdale init
```

4. 最终生成目录如下：□

- final
- java并发基础
- 线程安全的HashMap
- ConcurrentHashMap原理

final

final 关键字可用于修饰类、方法、变量。是Java的一个保留关键字。一旦将引用声明作final，就不能改变这个引用，否则会报编译错误。



final 可修饰本地变量、成员变量。

(本地变量： 在方法中或代码块中的变量)

final变量是只读的。

final 可声明方法

方法前面加上final表示该方法不可被子类的方法重写。一般方法功能比较完善了才会加final。

final方法比非final方法快，因为在编译时就已经静态绑定了，不需要在运行时再动态绑定。

final 可修饰类

final类通常功能完整，不能被继承。Java中很多类是final的，比如String, Integer以及其他包装类。

final 关键字的好处

1. final关键字提高了性能。JVM和Java应用都会缓存final变量。
2. final变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销。
3. 使用final关键字，JVM会对方法、变量、类进行优化。

不可变类

创建不可变类要使用final关键字。不可变类是指它的对象一旦被创建了就不能更改。String是不可变类的代表。

关于final的重要知识点

1. final关键字用于成员变量、本地变量、方法、类。
2. final成员变量必须在声明的时候初始化或在构造器中初始化，否则会报编译错误。
3. 不能对final变量再次赋值。
4. 本地变量必须在声明时赋值。
5. 在匿名类中所有变量都必须是final变量。
6. final方法不能重写。

7. final类不能被继承。
8. final关键字不同于finally关键字，后者用于异常处理。
9. final关键字容易与finalize()方法搞混，后者是在Object类中定义的方法，是在垃圾回收之前被JVM调用的方法。
10. 接口中声明的所有变量本身是final的。
11. final和abstract这两个关键字是反相关的，final类不可能是abstract的。
12. final方法在编译阶段绑定，称为静态绑定。
13. 没有在声明时初始化final变量的称为空final变量(blank final variable)，它们必须在构造器中初始化，或者调用this()初始化。不这么做的话，编译器会报错“final变量（变量名）需要进行初始化”。
14. 将类、方法、变量声明为final能够提高性能，这样JVM就有机会进行估计，然后优化。
15. 按照Java代码惯例，final变量就是常量，而且通常常量名要大写。

对于集合对象声明为final指的是引用不能被更改，但是可以向其中增加、删除或者改变内容。

java并发基础

当一个对象或变量可以被多个线程共享的时候，就有可能使得程序的逻辑出现问题。

Java内存模型

在java memory model中。Memory分为两类，main memory和working memory。
 main memory为所有线程共享，working memory中存放的是线程所需要的变量的拷贝。
 (线程要对main memory中的内容进行操作的话，首先需要拷贝到自己的working memory。
 一般为了速度，working memory一般是在cpu的cache中的)
 volatile的变量在被操作的时候不会产生working Memory的拷贝，而是直接操作main memory。
 当然volatile虽然解决了变量的可见性问题，但没有解决变量操作的原子性的问题。还需要synchronized或者CAS相关操作配合进行。

可见性

假设一个对象中有一个变量i，那么i是保存在main memory中的。
 当某一个线程要操作i的时候，首先需要从main memory中将i加载到这个线程的working memory中。
 这个时候working memory中就有了一个i的拷贝，这个时候此线程对i的修改都在其working memory中，直到其将i从working memory写回到main memory中，新的i的值才能被其他线程所读取。
 从某个意义上说，可见性保证了各个线程的working memory的数据的一致性。

可见性遵循以下规则：

1. 当一个线程运行结束时，所有的变量会被flush回main memory中。
 2. 当一个线程第一次读取某个变量的时候，会从main memory中读取最新的。
 3. volatile的变量会被立刻写到main memory中。在jsr133中，对volatile的语义进行增强。
 4. 当一个线程释放锁后，所有的变量的变化都会flush到main memory中。
- 然后一个使用了这个相同的同步锁的进程，将会重新加载所有的使用到的变量，这样就保证了可见性。

原子性

当某个线程修改i的值的时候，从取出i到将新的i的值写给i之间不能有其他线程对i进行任何操作。也就是说保证某个线程对i的操作是原子性的，这样就可以避免数据脏读。通过锁机制或者CAS(Compare And Set需要硬件CPU的支持)操作可以保证操作的原子性。

有序性

假设在main memory中存在两个变量i和j，初始值都为0，在某个线程A的代码中依次对i和j进行自增操作(i, j的操作不互相依赖)

```
i++;
```

```
j++;
```

所以i, j修改操作的顺序可能会被重新排序。

那么修改后的i, j写到main memory中的时候，顺序可能就不是按照i, j的顺序了，这就是所谓的reordering，

在单线程的情况下，是按照as-if-serial语义的，即使在实际运行过程中，i, j的自增可能被重新排序，当然计算机也不能帮你乱排序，存在上下逻辑关联的运行顺序肯定还是不会变的。

但是在多线程环境下，问题就不一样了。

这就是reordering产生的不好的后果，所以我们在某些时候为了避免这样的问题需要一些必要的策略，以保证多个线程一起工作时也存在一定的次序。

JMM提供了happens-before的排序策略，这样我们可以得到多线程环境下的as-if-serial语义。



线程安全的HashMap

一个hashMap在运行的时候只有读操作，就不会存在线程安全问题。

但当涉及到同时有改变也有读的时候，就要考虑线程安全问题了。

在不考虑性能问题时，我们可以用HashTable, Collections.synchronizedMap(hashMap)。

这两种方式基本都是对整个hash表结构做锁定操作的，这样在锁表的期间，别的线程就需要等待了，性能低。

ConcurrentHashMap原理

描述

ConcurrentHashMap是在Java1.5作为HashTable的替代选择新引入的，是concurrent包的重要成员。

在Java1.5之前，如果想要实现一个可以在多线程和并发的程序中安全使用的Map，只能在HashTable和synchronized Map中选择，因为HashMap并不是线程安全的。

ConcurrentHashMap不仅是线程安全的，而且比HashTable和synchronized Map的性能要好。

相对HashTable和synchronized Map锁住了整个Map，ConcurrentHashMap只锁住部分Map。

ConCurrentHashMap允许并发的读操作，同时通过同步锁在写操作时保持数据完整性。



适用场景

1. 读数量超过写数量。

因为当写者数量大于等于读者时，ConcurrentHashMap的性能是低于HashTable和synchronized Map的。

由于当锁住了整个Map时，读操作要等待对同一部分执行写操作的线程结束。

ConcurrentHashMap适用于做cache，在程序启动时初始化，之后可以被多个请求线程访问。

ConcurrentHashMap是HashTable一个很好的替代，但是ConcurrentHashMap比HashTable的同步性弱。

java中concurrentHashMap的实现

数据结构ConcurrentHashMap的目标是实现支持高并发、高吞吐量的线程安全的HashMap。

当然不能直接对整个HashTable加锁，所以在ConcurrentHashMap中，数据的组织结构和HashMap有所区别。

一个ConcurrentHashMap由多个segment组成。

每一个segment都包含了一个HashEntry数组的HashTable，每一个segment包含了对自己的HashTable的操作。

比如get, put, replace等操作，这些操作发生的时候，对自己的HashTable进行锁定，由于每一个segment写操作只锁定自己的HashTable，所以可能存在多个线程同时写的情况，性能比只有一个HashTable锁定的情况好。

在jdk6, jdk7, jdk8中的实现都不同

concurrentHashMap引入了分割，并提供了HashTable支持的所有功能。

在concurrentHashMap中，支持多线程对Map做读操作，并且不需要任何blocking。

这得益于concurrentHashMap将Map分割成了不同的部分，在执行更新操作时只锁住了一部分。

根据默认的并发级别(concurrency level)，Map被分割成16部分，并且由不同的锁控制。

即同时最多可以有16个写线程操作Map，性能的提升显而易见。

但由于一些更新操作，如put(), remove(), putAll(), clear()只锁住操作的部分，所以在检索操作不能保证返回的是最新的结果。

另外

在迭代遍历concurrentHashMap时，keySet返回的iterator是弱一致和failsafe的，可能不会返回某些最近的改变。

并且在遍历过程中，如果已经遍历的数组上的内容变化了，不会抛出ConcurrentModificationException的异常。

concurrentHashMap的并发级别是16，但可以在创建concurrentHashMap时通过构造函数改变。

并发级别代表着并发执行更新操作的数目，所以如果只有很少的线程会更新Map，那么建议设置一个低的并发级别。

另外，concurrentHashMap还使用了ReentrantLock来对segments加锁。

ConcurrentHashMap中的putIfAbsent()

```
synchronized(map){
    if(map.get(key) == null){
        return map.put(key, value);
    } else {
        return map.get(key);
    }
}
```

上面这段代码在HashMap和HashTable中是好用的，但在ConcurrentHashMap中是有出错的风险的。因为ConcurrentHashMap在put操作时并没有对整个Map加锁，所以一个线程正在put(k, v)时，另一个线程调用get(k)会得到null，这就会造成一个线程put的值会被另一个线程put的值所覆盖。当把代码封装到synchronized代码块中，这样虽然线程安全了，但会使你的代码变成单线程。ConcurrentHashMap提供的putIfAbsent(key, value)方法原子性的实现了同样的功能，同事避免了上面的线程竞争的风险。



ConcurrentHashMap总结

1. ConcurrentHashMap允许并发的读和线程安全的更新操作；
2. 在执行写操作时，ConcurrentHashMap只锁住部分的Map；
3. 并发的更新是通过内部根据并发级别将Map分割成小部分实现的；
4. 高的并发级别会造成时间和空间的浪费，低的并发级别在写多线程时会引起线程间的竞争；
5. ConcurrentHashMap的所有操作都是线程安全；
6. ConcurrentHashMap返回的迭代器是弱一致性，fail-safe并且不会抛出ConcurrentModificationException异常；
7. ConcurrentHashMap不允许null的键值；
8. 可以使用ConcurrentHashMap代替HashMap，但ConcurrentHashMap不会锁住整个Map