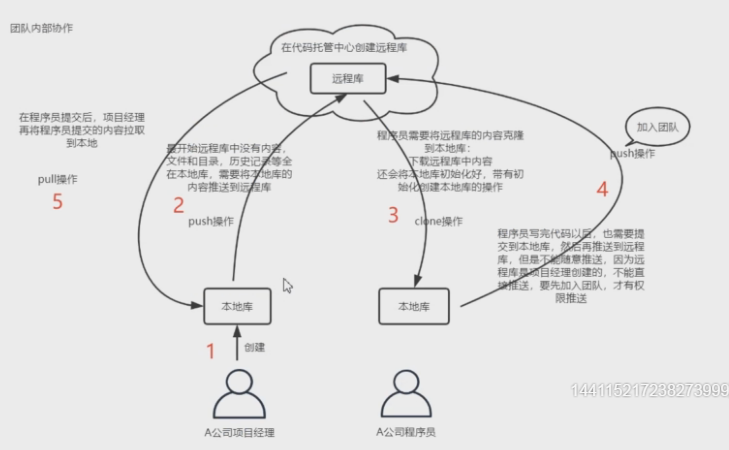
# Git



版本控制系统：

集中化版本控制系统：例如svn、cvs

分布式版本控制系统：客户端不止最新版本文件，而是把代码都镜像下来了。例如Git

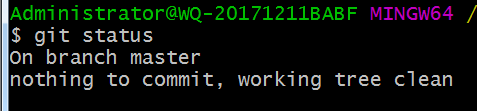
Svn存储的是当前版本和上一版本的差异。

Git储存的是当前版本和上一版本的索引。

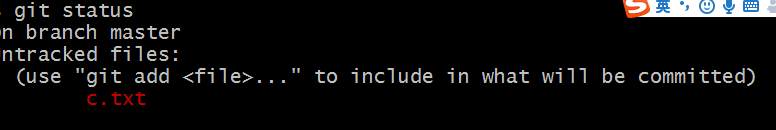
本地结构：工作区（写代码的地方）->Git.add->暂存区（临时存储）-> git.commit->本地库（历史版本信息）

## 使用方法：

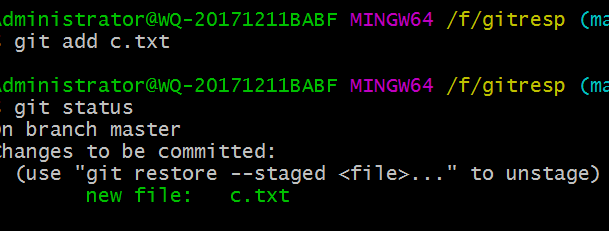
1. 在电脑里新建一个文件夹
2. 桌面右键选择git bash here
3. 然后右键修改字体和编码格式
4. 命令和linux一样：git –vrsion;clear;
5. 设置用户名和邮箱 ：git config --global user.name “niulipeng” ; git config --global user.email [784115501@qq.com](mailto:784115501@qq.com)
6. 本地仓库的初始化：cd命令进入本地文件夹 然后git init 。查看文件ll –la (隐藏目录.git文件此文件下的东西勿修改)
7. 本地新建文件（该文件需要放到本地仓库里）工作区git add test.txt –》暂存区git commit -m “注释” -》本地库。Ps:工作区就是本地磁盘
8. 查看当前状态 git status 如下：



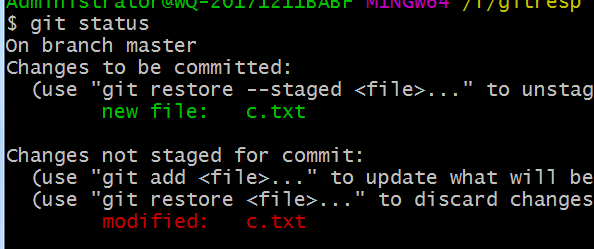
用命令mkdir 新建的文件会同步本地库 直接在文件夹新建的文件用此命令会报如下错误

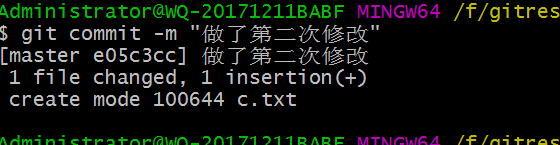


执行完 git add c.txt命令后如下：



修改c.txt文件之后 如下 需要重新添加到本地库





## 查看修改日志

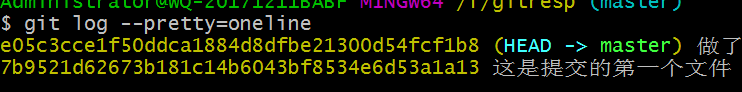
git log 命令

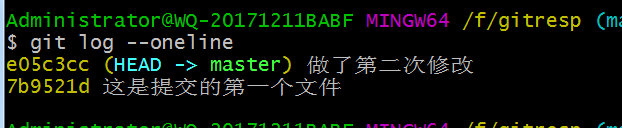


Commit后一连串的是当前提交版本的索引

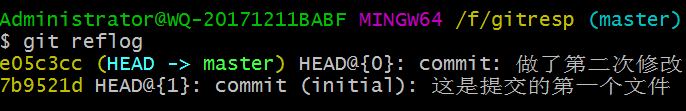
如果提交日志多会出现： 往下查找按 空格 往上查找按B 到尾页显示end 然后按 q退出

其他日志展示方式



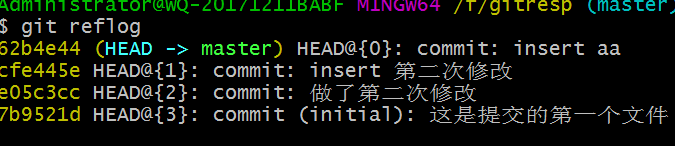


@里的数字表示当前库回退到该版本需要走的步数

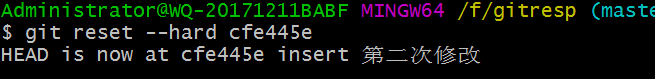


## Reset命令

git reset --hard [索引] 回退或者前进历史版本（工作区文件随着变更）如下



Hard 后面跟要跳转到版本的索引

·

此时当前文件的内容变更为指向的版本内容

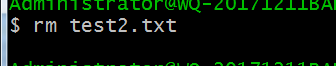
如果要回滚到当前版本需要从当前指向的索引修改到上一个索引

Git reset –mixed [索引] 只修改本地库和暂存区的文件，工作区的文件不变更

Git reset –soft [索引] 只修改本地库文件，暂存区和工作区的文件不变更

## 删除文件

**rm 文件名（只是删除工作区）**

****

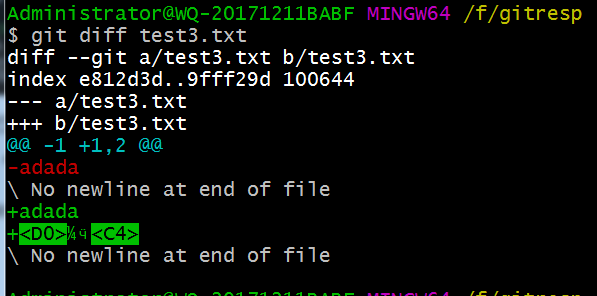
**然后以下操作删除本地库文件： git add test2.txt ； git commit –m “删除文件”**

**如果想恢复** git reset --hard [索引]

## 对比文件

修改本地文件导致本地区和暂存区不一致：git diff 文件名

比对文件差异git 是按行管理数据

****

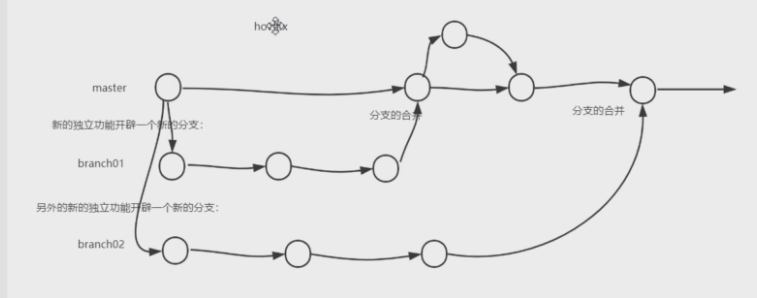
再添加

先删除一行

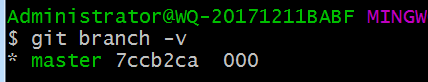
**Git diff 后不加文件名会展示工作区中和暂存区中所有文件的差异**

**Git diff HEAD指针/索引 文件名 比对暂存区和本地库的文件差异**

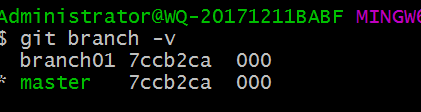
## 分支：

****

**查看分支 \*代表当期在哪个分支上**

****

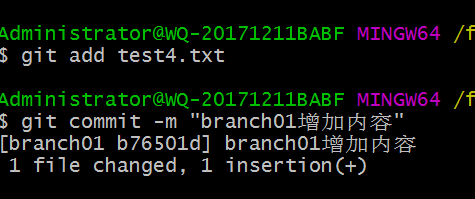
**创建分支**

****

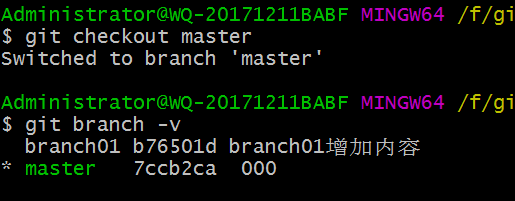
**切换分支**

****

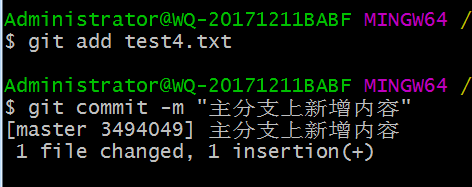
**在01分支下修改文件并提交**

****

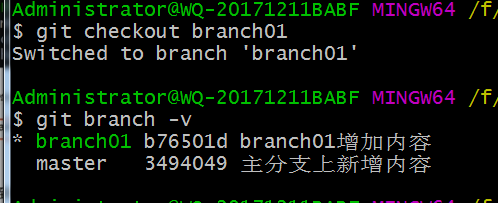
**然后切换到master分支**

****

**然后在主分支新增内容**

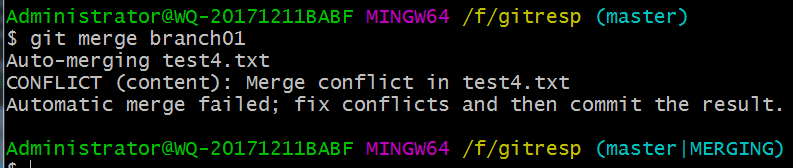
****

**然后再切换到01分支**

****

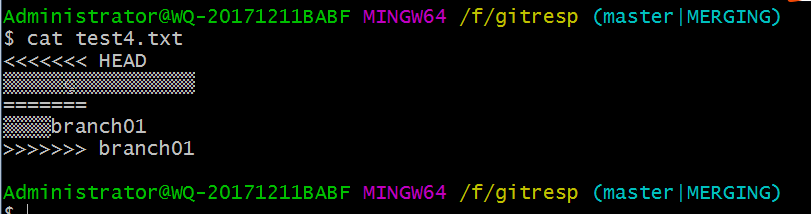
## 合并分支

**将01分支合并到主分支：先切换到主分支**

****

**出现冲突（修改同一个文件的同一个位置）merging表示正在合并中**

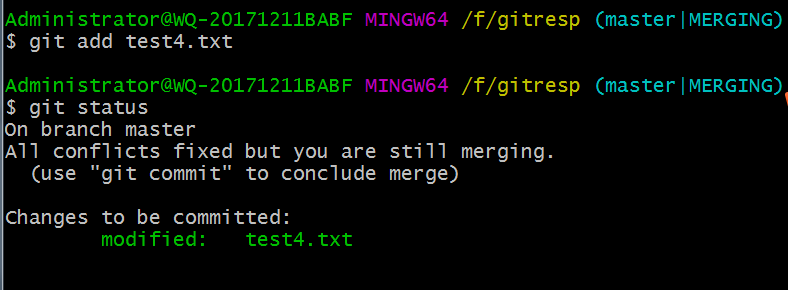
**查看文件（未修改txt的编码格式导致乱码）**

****

**合并修改的内容**

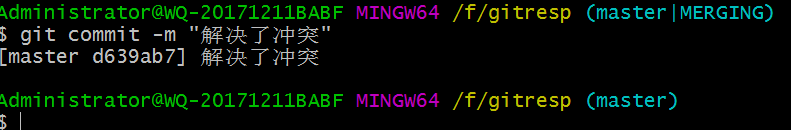
当前分支修改的内容

**去工作区修改最终的文件然后将文件添加到暂存区**

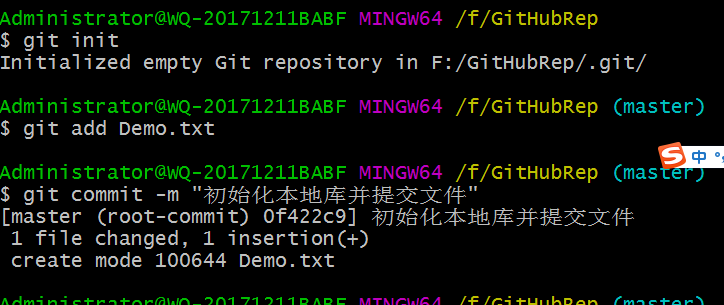
****

表示冲突已经解决但是仍处于合并状态

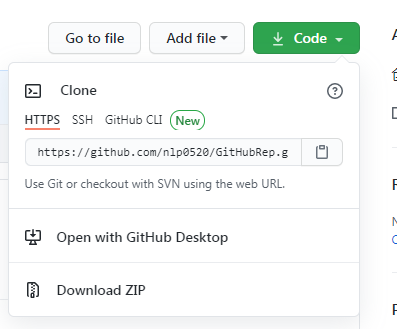
**然后提交到本地库**

****

## 创建GitHub远程库

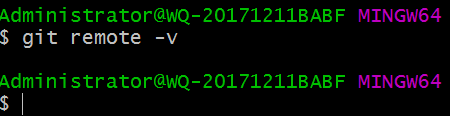
****

**在github上创建一个库，查看地址（新版的在code下 ）**

****

**在git本地将地址，通过别名**

**查看别名 git remote –v**

****

**新增别名 git remote add origin** [**https://github.com/nlp0520/GitHubRep.git**](https://github.com/nlp0520/GitHubRep.git)

**Add 后的是起的别名**

****

## 推送本地文件Git push origin master

Git push origin master (origin->起的别名,master->分支名字)

或者Git push <https://github.com/nlp0520/GitHubRep.git> master

输入完命令会出现登录github的账号和密码



在远程库创建了新的分支

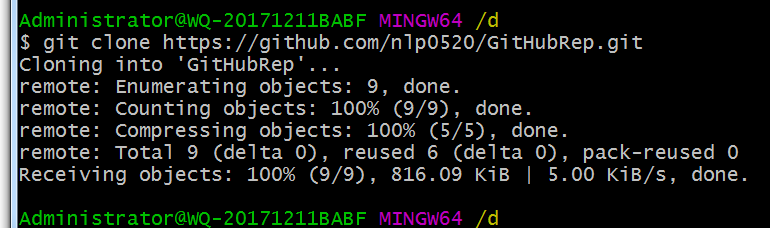
将本地库master分支提交到远程库master分支

## 克隆clone

在setting 下branch 修改默认的分支

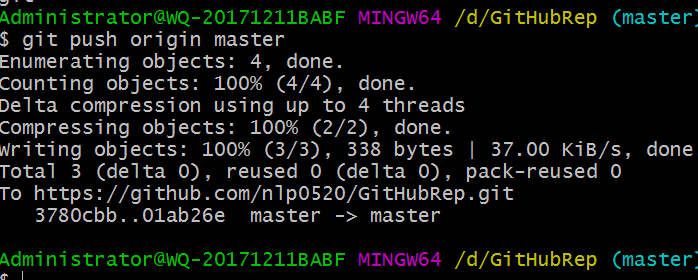


Git clone + github地址



## 推送push

Git push origin(别名) master(要推送的分支)

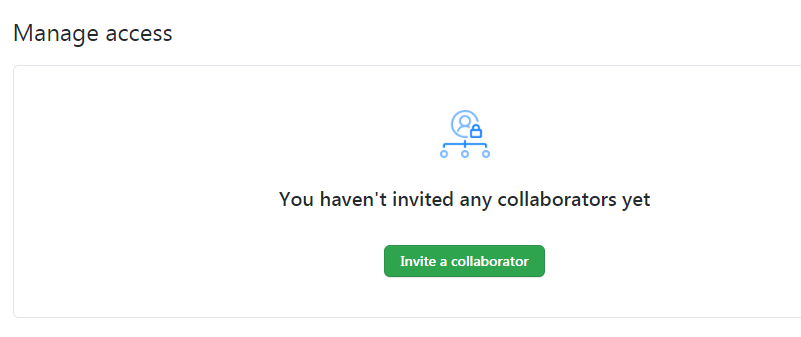




注意：同一台机器git本地有缓存,所以可以提交

凭据里清除

必须要先加入团队，登录项目经理账户settings->manager邀请成员

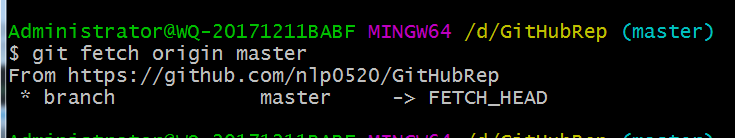


之后页面会出现一个邀请链接的文档，打开将地址输入到地址栏。然后接受邀请即可。

## 拉取pull

Pull相当于 fetch和merge

项目经理先确认项目已经更新，然后拉取。



Fetch 只是将文件下载到本地库，工作区文件未修改。

抓取后可以去本地库查看所有文件



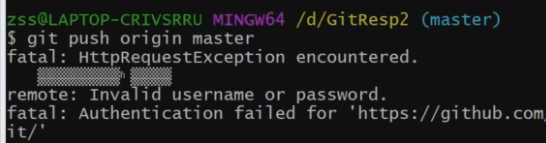
然后合并





或者直接执行 git pull origin master

多项目合作：修改同一个文件的同一行，先push上去的没问题，其他人修改同一行之后push会出现冲突。



需要先拉取下来再推送。人为解决冲突。

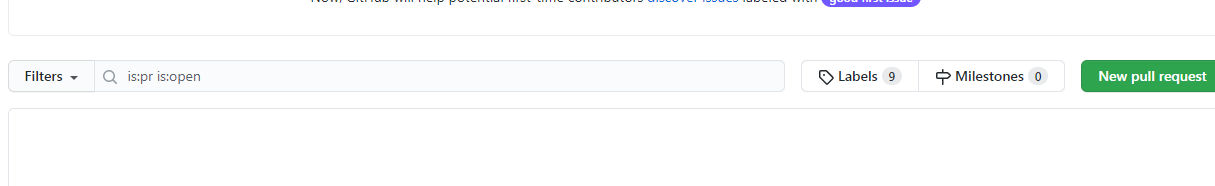
跨团队操作：

<https://github.com/nlp0520/GitHubRep.git>

登录另外账户，复制地址，然后点击下面的fork操作。

然后其他团队clone，push。

然后点击 pull requst



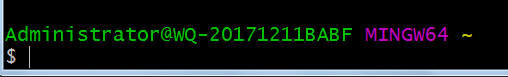
然后发送审核请求

项目经理审核在pull request 下files changed 查看其修改的具体内容

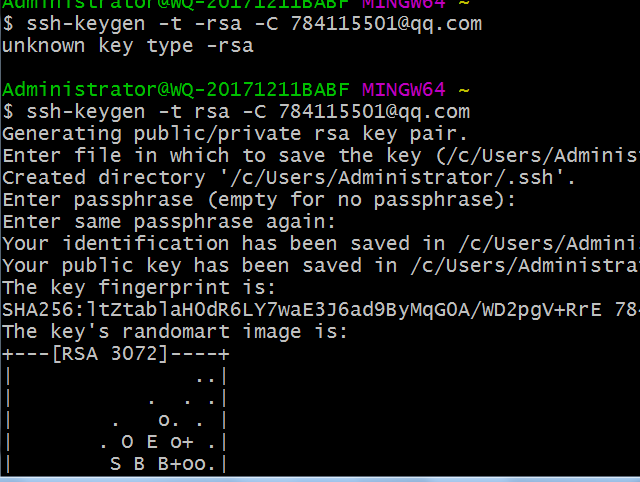
然后merger pull request

## 免密操作

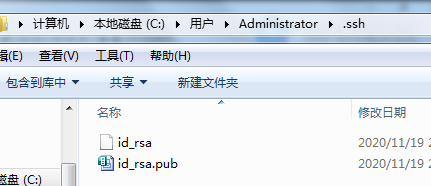
1、目录cd ~



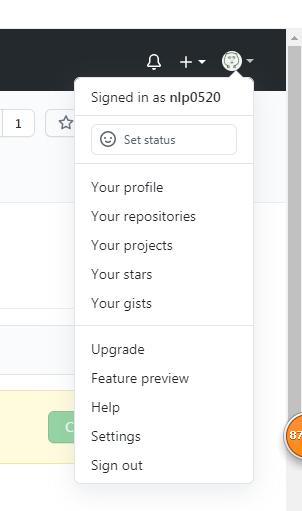
2、执行命令生成一个SSH的文件,三次回车确认

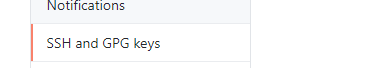


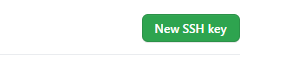
C大写 后跟github注册的邮箱。

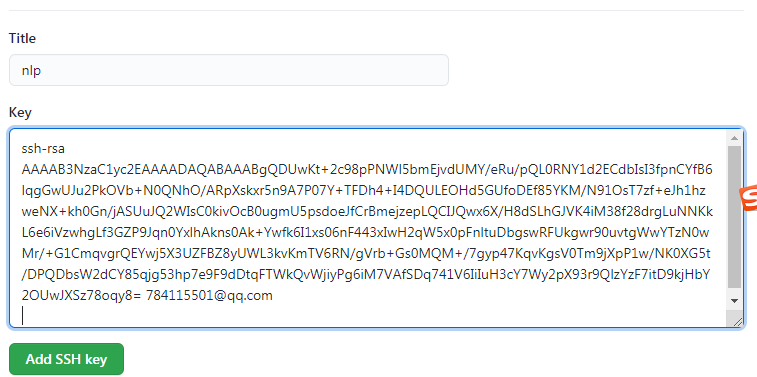


打开pub文件将里面内容复制操作，打开github账户settings

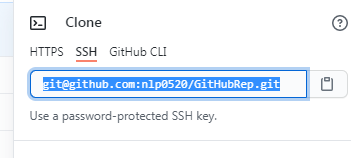








生成秘钥就可以正常操作了。不需要每次操作输入密码



起别名



## IDEA集成git

Settings->version control->git->git executable(git.exe)安装路径

本地库初始化：

Vcs->import into version control->create git rep

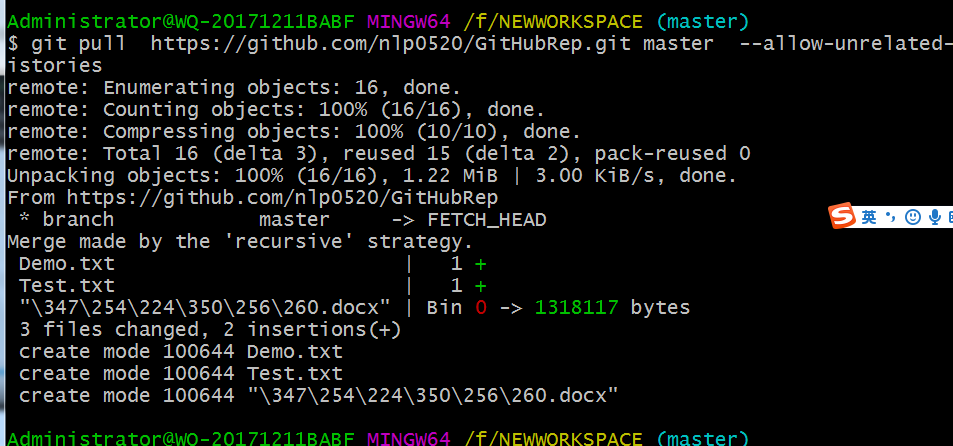
然后创建项目 会提示add操作 ，然后右键项目git->commit操作

提交成功之后左下角有version control查看日志

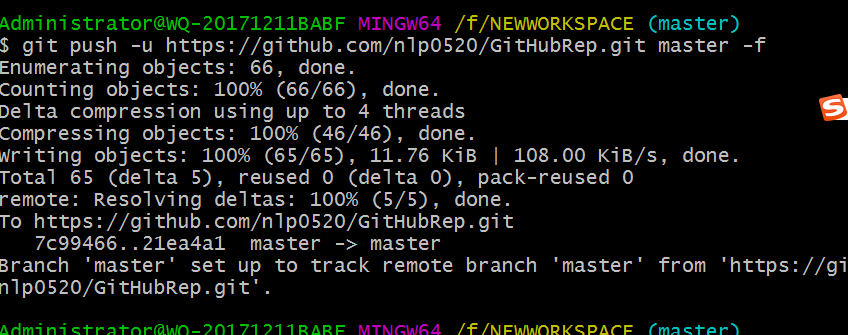
## 使用idea推送和拉取

首先将github上项目拉取到本地然后再将本地库推送上去做同步

Git pull 地址或别名 master --allow-unrelated-histories 将两个不同库的项目合并、



Git push –u 地址或者别名 master –f

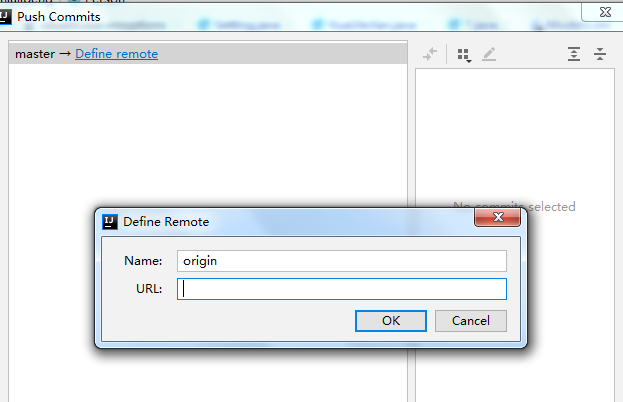


到此就完成本地库和远程库的同步。

然后使用IDEA操作

例如新建类然后右键add、commit

然后右键git->reposity->push

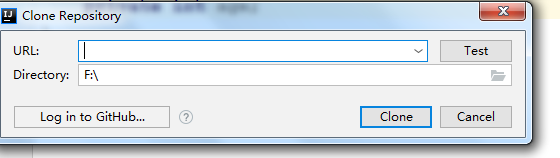


Name为别名 URL为远程库的地址

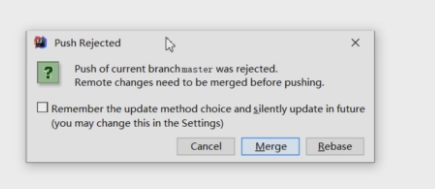
然后点push完成。

## 使用IDEA克隆

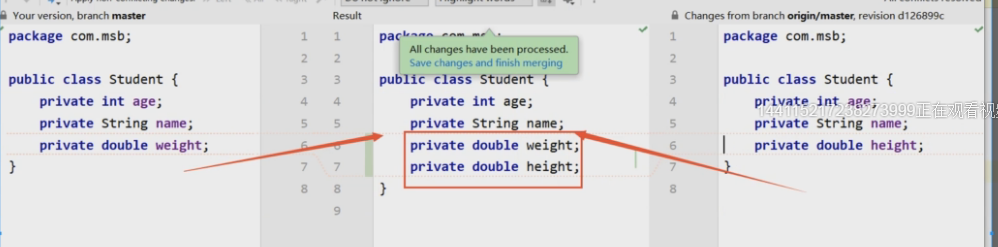
FILE->NEW->Project Version Control->git



冲突解决：尽量先将项目pull下来然后push



首先合并点击merge



# 多线程

线程：一个程序中不同的执行路径。

启动线程的方法：New T().run(); new T().start。

不同点：run是一条执行路径。Start是和主程序分支的一条执行路径。

创建线程的方式： https://blog.csdn.net/m0\_37840000/article/details/79756932

1. 继承Thread类
2. 实现Runnable接口 （代理模式）
3. 使用Callable接口和FutureTask（Lamda表达式）
4. 使用线程池例如Excutor框架（Executors.newCachedThread）

常用方法：sleep、join、yield、wait

Sleep:睡多长时间让给其他线程CPU去执行，不释放锁。

Yield:让出一下CPU进入等待队列。

Join:两个线程，在T1线程上调用T2线程的join，会等待T2线程执行完再执行T1线程。

Wait:进入阻塞状态，释放锁

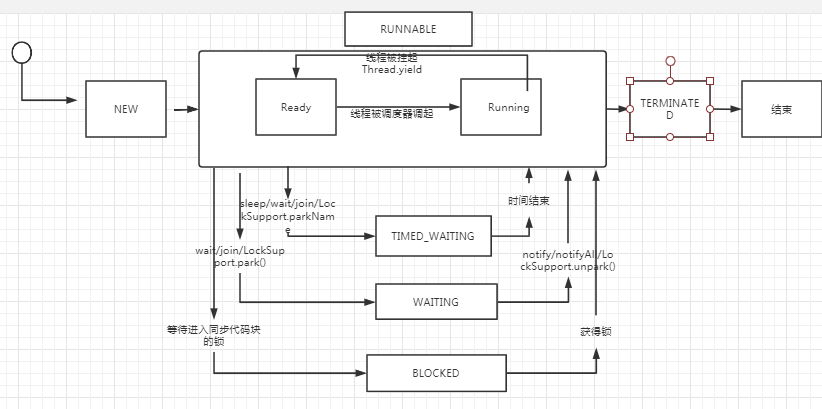
sleep和wait的区别：  
1、sleep是Thread的静态方法，wait是Object的方法，任何对象实例都能调用。  
2、sleep不会释放锁，它也不需要占用锁。wait会释放锁，但调用它的前提是当前线程占有锁(即代码要在synchronized中)。  
3、它们都可以被interrupted方法中断。

线程的优先级设置的大就一定先执行吗？

不一定。

唤醒和等待操作对应的不是Thread而是设置的共享对象或者共享变量。

线程状态图：一共有六个状态***NEW***、***RUNNABLE***、***BLOCKED***、***WAITING***、***TIMED\_WAITING、TERMINATED***



线程执行完毕不能重新start.

## Synchronized

保证了原子性和可见性

锁定是当前对象获得当前对象的锁才能执行内部代码。

Synchronized()和sychronized(this)是一样的。

同步方法和同步代码块。

Synchronized是可重入锁，例如一个同步方法可以调用另外一个同步方法即一个线程已经拥有该对象的锁当再次申请时仍然会得到这个对象的锁。

一个程序抛出异常，锁会被自动释放。所以使用时注意程序异常时的处理。

不能用String常量、Integer、Long

底层实现：

早期的是重量级的需要向OS申请

后来锁升级，syn(obj)先mark word标记这个对象的前两位记录这个线程ID(偏向锁)

如果有线程征用升级为自旋锁（默认10次），得不到就去OS申请。

执行时间短线程多用自旋锁，执行时间长线程少用系统锁。

优化：  
锁的细化，例如将锁不加到方法上，而是加在方法内

锁定的是对象o时，对象o的属性发生变化不影响锁的使用；但是o变成另外一个对象，则锁定的对象发生变化（解决办法将对象定义成final）

## Volatile

保证线程可见性（MESI缓存一致性协议）；禁止指令重排序。

（线程访问堆里的东西会先复制一份出来然后修改之后写到堆里的引用对象。两个线程之间是不可见的）

双重检查例如单例模式的懒汉式可以用volatile修饰对象然后synchronized对类加锁。

## CAS无锁化

三个属性值： P:期望的值 V：实际的值 E：要改变的值

使用AtomXXX类，因为这种类本身都是原子性的。

ABA问题：例如将期望值改了一次又改了回去。解决办法加版本号。

LongAddrer效率比atomic效率更高，其内部用了分段锁。

ReentranctLock可重入锁（同一个线程里syn方法调用另外一个syn方法）可以替代synchronized。注意用这种方式必须在finally释放锁（不是自动释放锁）lock.unlock。

## Synchronized和ReentranctLock区别

都是可重入锁，一个自动释放锁，一个需要手动释放锁；

ReentranctLock里有特定方法：tryLock、new ReentranctLock(true)公平锁/非公平锁、lockinterruptibly(被打断执行其他线程)

ReentranctLock底层是CAS的实现，synchronized底层是默认锁的四种状态的升级(无锁、偏向锁、自旋锁、重量级锁)

## JUC同步锁

CountDownLatch类(门栓)、CyclicBarrier(循环等待)

Phaser(阶段锁)

ReadWriteLock:共享锁/排它锁（相同的读/写线程可以进行同时操作）

Semaphore(设置信号塔里的锁得到就执行)：用于限流（多少个窗口）

可以起多个线程，但是只有设置的数值的线程数可以执行。

线程池是只能有这么多线程。

Exchanger:两个线程之间交换数据

LockSupport: park方法停止（线程阻塞），unpark方法解封指定线程 可以先于park调用

Condition相当于两个等待队列

AQS:abstractQueuedSynchronized源码解读

双向链表（里面放的是线程）的（node / tail）state控制拿到锁的状态，中间用了CAS 去判断锁的状态

ThreadLocal源码解读

每个线程都会讲值set到当前线程的map里，多用于声明式事务，保证同一个connection。

## 四大引用

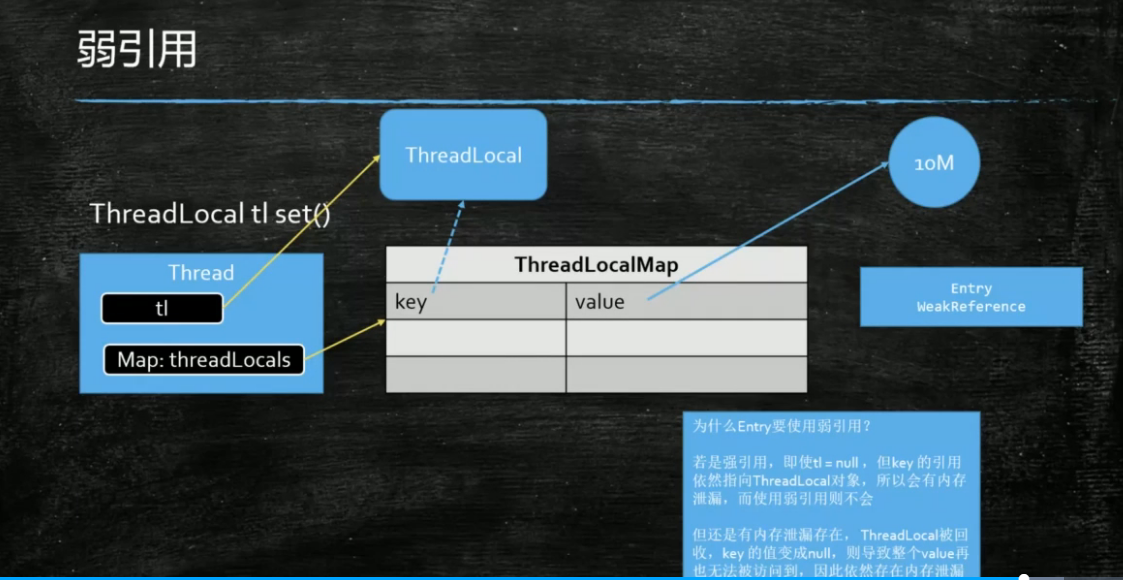
强软弱虚

强引用：普通创建对象都是强引用（将创建对象设为Null手动调用gc可以回收）

软引用：例如SoftReference<**byte** []> m = **new** SoftReference<>(**new byte**[1024\*1024\*10]);当当前堆内存不够时系统会垃圾回收，先回收一次，如果不够，会把软引用干掉。

弱引用：遭遇gc就会被回收，一般用在容器里。WeakReference<M> wr = **new** WeakReference(**new** M()); 了解WeakHashMap应用！！

ThreadLocal<Yinyong> tl = **new** ThreadLocal<>();  
tl.set(**new** Yinyong());  
tl.remove();*//当threadlocal不用时要remove掉防止内存泄漏*



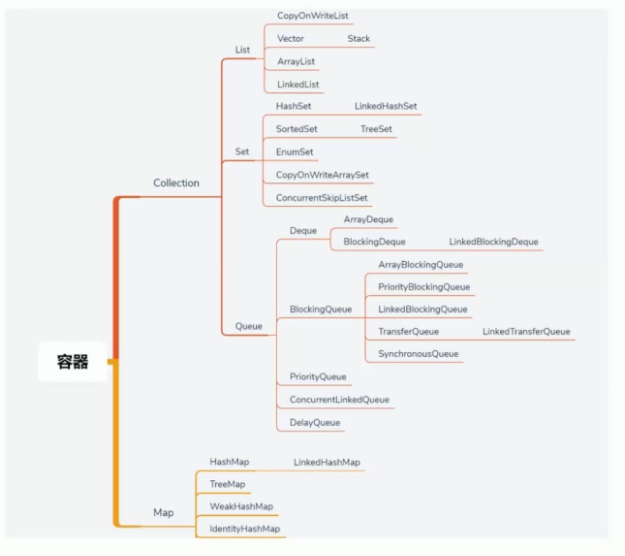
虚引用：管理堆外内存的。用于写JVM的用的。

**private static final** ReferenceQueue<Yinyong> ***QUEUE*** = **new** ReferenceQueue();

PhantomReference pr = **new** PhantomReference(**new** Yinyong (), QUEUE);

当虚引用被垃圾回收时，会把一个值放到队列里 通知去做处理。

## 容器



Collection储存的是单个的元素；Map是键值对存放的。

Vector和HashTale 自带锁

ConcurrentHashMap由HashTable发展来的，底层也是加锁的。读取效率高，写效率一般。

给HashMap手动加锁Map m = Collections.synchronizedMap(new HashMap())；

多线程优先考虑Queue,如果存放的元素是唯一的可以考虑set.。

高并发的常用容器：ConcurrentHashMap、ConcurrentSkipMap(排序)

跳表：从底层链表里拿出关键元素重新组成一个新的链表然后再取出关键元素组成新的链表。。。查询时会从最顶层的链表开始查询。

## Queue属于JUC

Queue和List的区别

添加了put、offer、poll、peak、take等对线程友好的方法（或者阻塞或者等待）而不是抛出异常。

Queue qu = **new** ConcurrentLinkedQueue();

BlockingQueue lq = **new** LinkedBlockingDeque<>();*//实现了阻塞*

BlockingQueue lq1 = **new** ArrayBlockingQueue(10);*//定义初始大小*

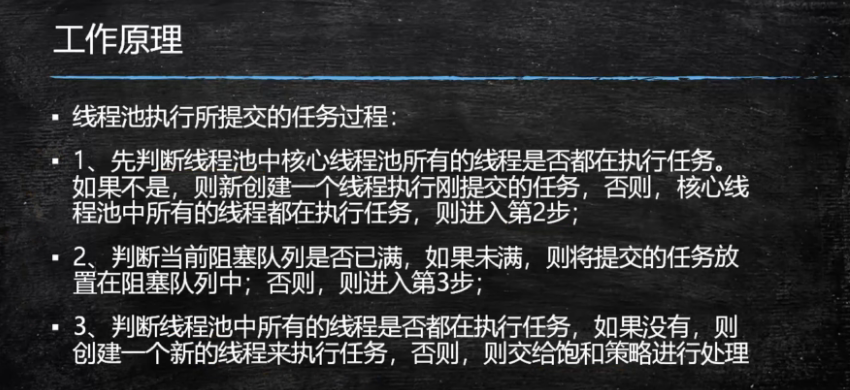
BlockingQueue lq2 = **new** DelayQueue();//底层是PriorityQueue，使用时实Delayed接口，会做一个比较等待时间短的优先执行，用于按时间进行调度的任务。

PriorityQueue p = **new** PriorityQueue();//内部是一个最小堆的树会进行一个排序。

SynchronousQueue *sq* = **new** SynchronousQueue();//先阻塞住当queue里有值后会执行操作，多用于两个线程之间的调度/交换数据/通知任务。

LinkedTransferQueue *lq* = **new** LinkedTransferQueue();//提供的tranfer方法和之前的put方法有区别（put放完数据就离开，transfer会等待消费者取数据才离开）。常用于一个事件需要有一个结果，然后进行之后的操作。

## 线程池



维护着两个集合：一个线程的集合，一个任务队列的集合

Excutor(线程执行器)🡪ExcutorService(完善了线程池生命周期的具体方法)🡪ThreadPoolExcutor()

Callable类似于Runnable，区别是call方法有返回值

Future接口 储存将来执行的结果，有返回值的异步的

FutureTask 即是一个Runnable也是一个Future。

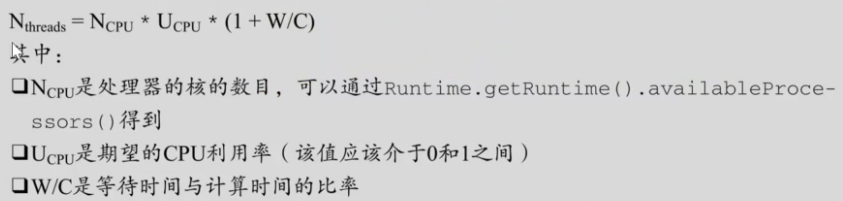
两种线程池

1. ThrealPoolExcutor：多个线程共享同一个队列。
2. ForkJoinPool:分解汇总的任务；用很少的线程可以执行很多的任务(子任务)；CPU密集型;每个线程对应一个队列，当本线程的任务队列执行完成后可以从其他线程的任务队列那任务去帮助执行。

ThreadPoolExecutor *7个参数的属性*

*//核心线程数；最大线程数；最大生存时间；生存时间的单位；任务队列；线程工厂（定义线程的名字，方便出错回溯）；拒绝策略*ThreadPoolExecutor **threadPoolExecutor** = **new** ThreadPoolExecutor(2, 5,  
 600, TimeUnit.***SECONDS***,  
 **new** ArrayBlockingQueue<Runnable>(2),  
 Executors.*defaultThreadFactory*(),  
 **new** ThreadPoolExecutor.AbortPolicy());

线程数的合适比例



## JDK默认提供的线程池

Executors.*newSingleThreadExecutor*();//为什么用单线程池：1、有任务队列2、有线程的生命周期

Executors.*newCachedThreadPool*();//可以起多个线程Integer.Value

Executors.*newFixedThreadPool*(4);//固定线程数

CachedThread VS FixedThread

任务来的忽高忽低用cache ,任务稳定用fix

ScheduledExecutorService executorService4 = Executors.*newScheduledThreadPool*(3);//定时任务线程池

并发：是指任务提交（多个任务同时发送过来）

并行：是指任务执行（多个任务同时被多个CPU执行）

并行是并发的子集。

## ThreadPoolExecutor源码解读

**Ctl 是integer型的共32位，前3位标记线程状态 后29位表示线程数**

**主要方法excutor():每次操作都要get获取线程状态（多线程防止其他线程占用）,判断线程数是否<coreSize 起核心线程/加队列；核心线程数满了起非核心线程。**

addWorker():分成两个步骤：1、两个死循环自旋目的work数加1（使用了CAS）.

2、将work加入到Worker类中执行start（加锁）

Worker类：是Runnable和AQS;记录是哪个thread。

## JMH

测试框架 Java Microbenchmark Harness

官网：http://openjdk.java.net/projects/code-tools/jmh

## Discuptor

内存中用于存储高效的任务队列，单机支持高并发，用于生产者消费者模式，是一个环形的buffer,数组实现，没有头尾指针只有一个sequence指向下一个可用元素，用策略覆盖之前的数据。

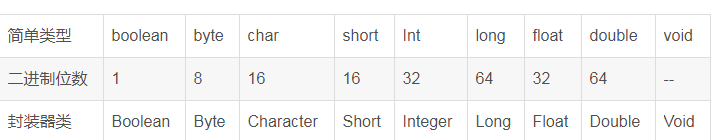


开发步骤

1. 定义Event，队列中需要处理的元素
2. 定义Event工厂，用于填充队列
3. 定义EventHandler(消费者)，处理容器中元素

# java基础

八大基本类型 一个字节8位



数字默认的类型是Int类型

\n换行 \t制表符 \r回车 \\转义符

运算符的优先级 && > ||

原码 1代表负 0代表正 、 反码 保留第一位其余位置取反 、补码 正数的补码还是本身负数的补码是反码+1

Switch 在JDK1.7之前不能判断String

斐波那切数列：第三位为前两位的和 。用递归算法 （有自身调用自身；有结束）getm()-1 +getm()-2

数组：长度确定 一旦创建不可改变 存储相同类型的集合包括基本数据类型和引用数据类型

声明一个数组就是在内存区开辟了一连串的存储空间。

Int[] s = new int[]{1,2} s在栈里指向new时在堆内存中创建的地址

栈stack存储的是变量：基本类型和引用类型 先进后出

堆 heap存储的是new出来的对象 先进先出

数组是引用类型，当创建数组完成以后是在 方法外定义一个变量，此时数组的值是有默认值的

数据结构：线性表 、非线性表、树、图、队列、栈

排序的时间复杂度：衡量一个数组结构是否合适的衡量标准

排序算法的稳定性：排序后值的跟排序之前的位置是否发生变化

二维数组被称为数组的数组 第一个数组指向第二个数组（可以是非固定数量）

OOP: Object Oriented Programming

对象：实实在在的一个东西例如喝水杯 ； 水杯一个名词是对象的抽象就是类

类是对象的一个抽象，对象是类的一个实例化。

对象 ：一组属性和方法构成。

新建对象JVM分三步：1.申请内存2.对成员变量初始化3.把成员变量赋值

Jvaa中方法的传参都是值传递。

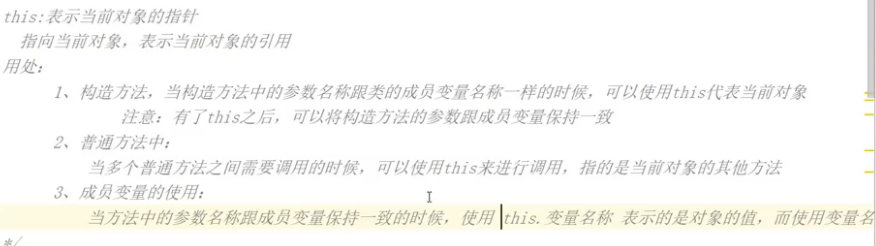
## == 和 equals

字符串在比较的时候 == 比较的是地址 ,equals比较的是值/两个对象的内容是否一致

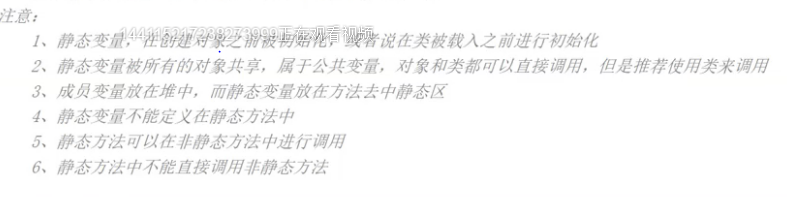
局部变量不包含默认值，使用前必须初始化。

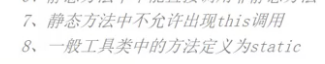
创建对象new时，默认会调用构造方法来创建对象（在heap中开辟空间），可以完成成员变量的初始化操作。

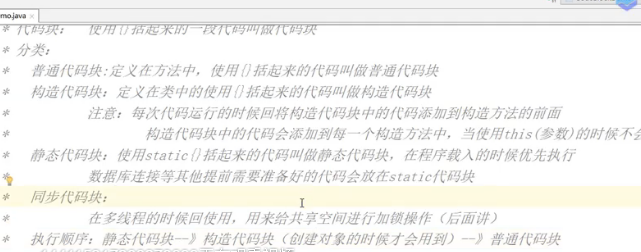
This



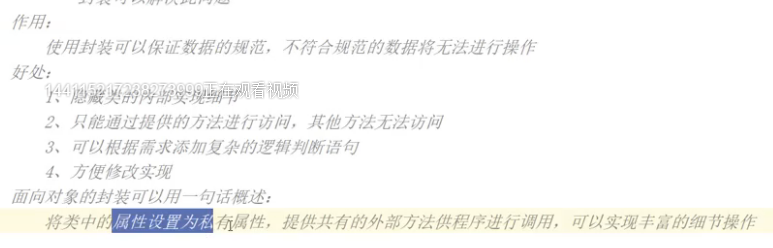
Satic:





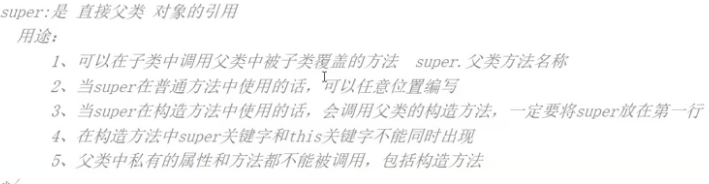
****

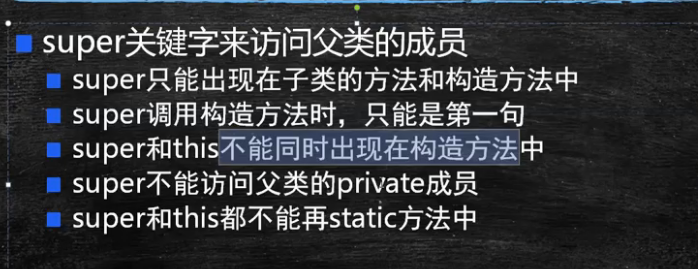
**封装**

****

**访问修饰符**

****

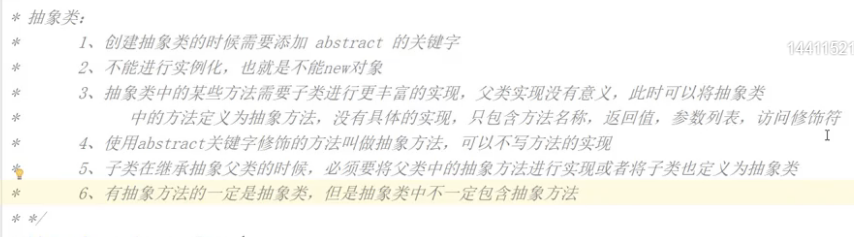
****

****

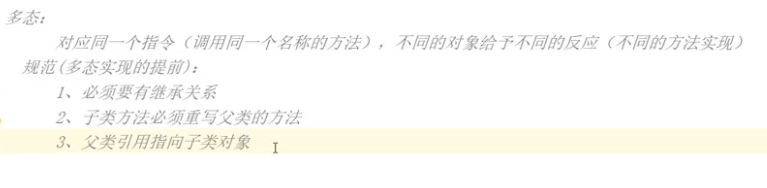
**子类的构造方法都会默认使用super调用父类的无参构造方法。如果构造方法显式的指定super的构造方法，那么无参的构造方法就不会调用。构造方法不能被继承，因此不能被重写。**

****

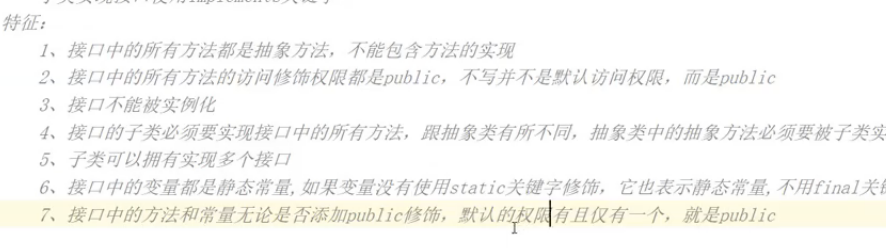
**父类的静态方法子类可以直接调用，但是不能被重写。**

****

**多态**

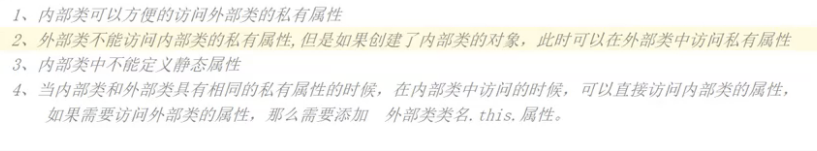
****

**接口**

****

**内部类**

**内部类 对象 = new 外部类.new 内部类。**

****

**匿名内部类:只使用一次**

**例如：new Thread(new Runnable(){})**

**静态内部类：**

**内部类 对象 = new 外部类名.内部类。**

**方法内部类：**

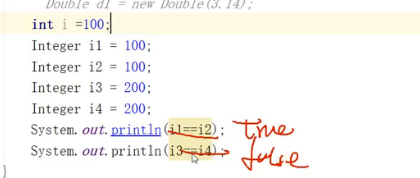
**不能在外部类的方法以为使用，所以不能使用访问控制符和static修饰；**

**Finally代码无论是否有异常都会执行。**

## 包装类与基本数据类型

**包装类是将基本数据类型封装成一个类，包含属性和方法。**

**在使用过程中涉及到装箱和拆箱。**

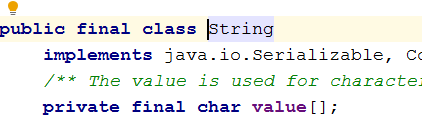
****

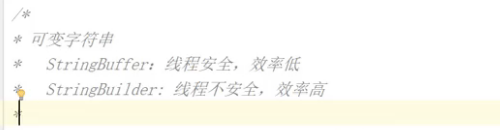
**在范围内从缓存的数组中拿值，否则就每次新建一个对象。**

**JDK1.7之后常量池放到了堆空间里了。**

## 字符串：

**本身是一个字符的数组**

****

****

## List

有续、不唯一

**Arraylist和LinkedList**

**Arraylist底层是数组，在内存中连续分配内存空间。插入慢，查询快。线程不安全。扩容为原来1.5倍。**

**LinkedList底层是链表（数据+指针）。插入快，查询慢。**

**Vector底层是数组，方法都加了synchronized线程是安全的。扩容为原来的2倍。**

## 循环

While /do…while /for / iterator(例如for(object a : list))

所有集合都默认实现了Iterable接口，意味着具备了增强for循环功能

Iterator方法中要求返回一个Iterator的接口子类实例化对象

ListIterator方法针对同时需要遍历和修改元素时。始终通过cursor和lastret的指针获取值及向下的遍历索引。

## Set

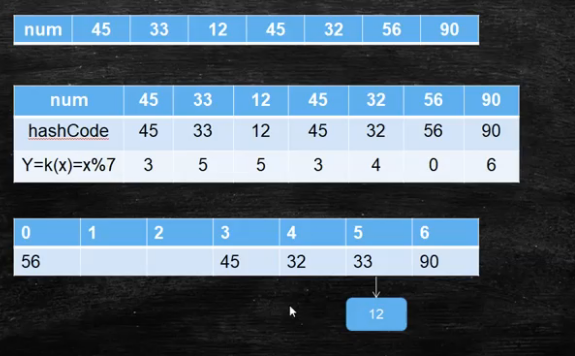
无序、唯一

HashSet和TreeSet

HashSet采用HashTable哈希表存储结构，添加、删除、查询快。无序。

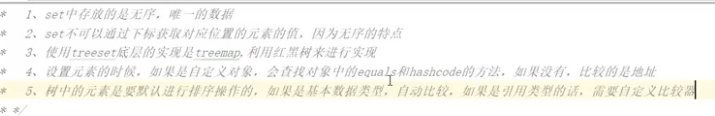
LinkedHashSet采用HashTable哈希表存储结构,同时使用链表维护次序。有序。

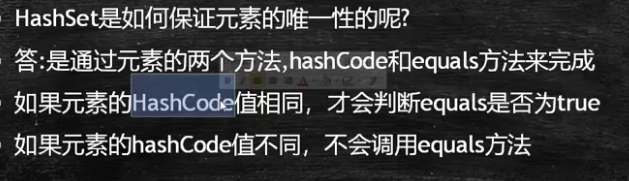
Hash原理(最大散列)



TreeSet底层是红黑树又叫平衡树（二叉树），要求每个节点存放的类型是一致的。会自动进行一个排序操作。

AVL树：两个树节点相差不超过1.当不平衡时会自动旋转。





## 泛型

当做一些集合的统一操作时，需要保证集合的数据类型的统一。

优点：数据安全；获取数据时效率比较高。

泛型类、泛型接口、泛型方法、泛型上限、泛型下限

## MAP

HashMap: key无序、唯一 value 无序、不唯一

1.7之前是数组+链表 1.8之后数组+链表+红黑树（多了两个值当链表超过8时变成红黑树）

LinkedHashMap: 有序的HashMap，速度快 ，链表储存

TreeMap:没有hash快 ，红黑树储存

Set keys = map.keySet();返回的是一个key的set集合。

Collection cl = map.Values();返回的是value的一个集合。

Map.Entry返回的是一个key-value的键值对。

Hashtable方法是同步的   
HashMap方法是非同步的

Hashtable基于Dictionary类   
HashMap基于AbstractMap，而AbstractMap基于Map接口的实现

Hashtable中key和value都不允许为null，遇到null，直接返回 NullPointerException   
HashMap中key和value都允许为null，遇到key为null的时候，调用putForNullKey方法进行处理，而对value没有处理

Hashtable中hash数组默认大小是11，扩充方式是old\*2+1   
HashMap中hash数组的默认大小是16，而且一定是2的指数

hashMap初始值为2的N次幂：1、方便进行&运算 hash & (inintCapacity-1) 2、在扩容之后设计到元素的迁移过程，迁移的时候只需要判断二进制的前一位是0(新数组和旧数组的下标不变)是1(将索引的位置加上旧的数组的长度即为新数组的下标)。

&运算 转换成二进制取值如21二进制10101 ；15二进制01111 取&之后为0101即为5。高于取模运算。

源码解读主要是put方法。

## Collections工具类

提供的静态方法：

AddAll()批量添加、sort()排序、binarySearch()二分查找、fill()替换、shuffle()随机排序、reverse()逆序

Collection是一个接口，储存一组不唯一，无序的对象，两个子类接口List和Set。

集合总结



## IO

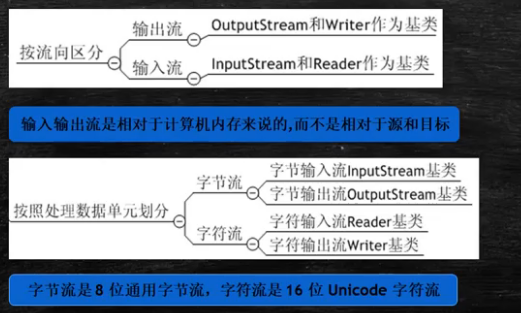
BIO:阻塞

NIO:非阻塞

AIO:

系统预读磁盘最小单位是页，4K的倍数

File:操作文件的类。



实现文件的复制

BufferedReader reader=**new** BufferedReader(**new** FileReader(**"d:/aa.txt"**));

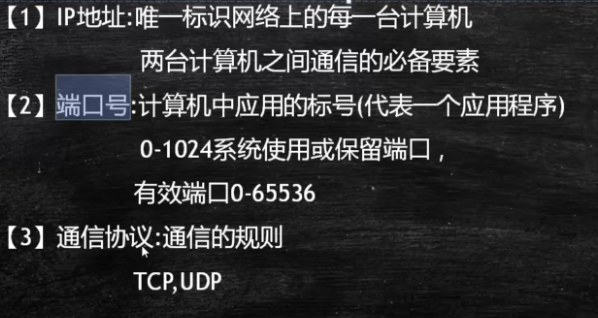
BufferedWriter bufferedWriter= **new** BufferedWriter(**new** FileWriter(**"d:/ab.txt"**));

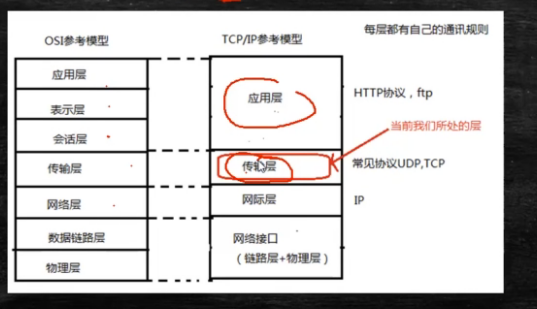
Printf:%s字符串 %d整数 %f浮点

RandomAccessFile 可以将指定文件分块读取.

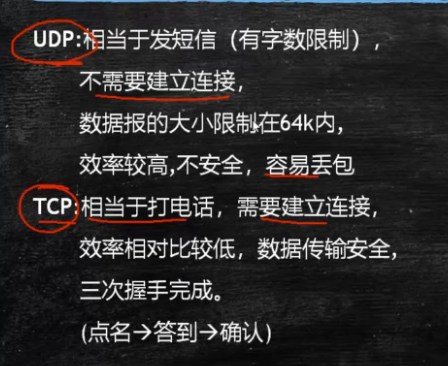
## 《设计模式之禅》

## 网络编程









Lambda表达式是JDK8后出现的替代只有一个抽象函数的接口（匿名内部类）

如创建线程：

**new** Thread(()->{  
 System.***out***.println();  
}).start();

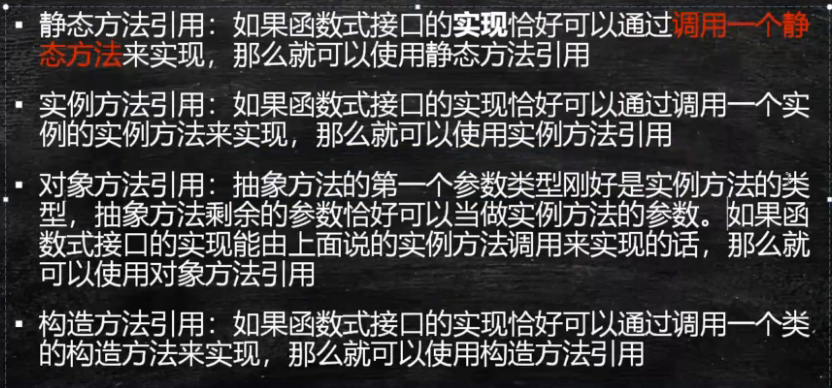
集合排序：

List list = Arrays.*asList*(**"java"**,**"python"**,**"scale"**,**"javascript"**);  
Collections.*sort*(list,(a,b)->a.length()-b.length());  
list.forEach(System.***out***::println);

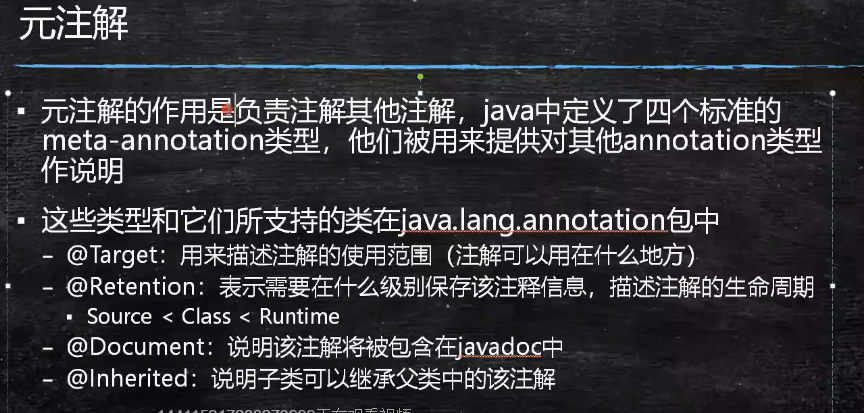
## Lambda表达式

*\* Consumer代表一个输入  
\* BiConsumer代表两个输入  
\* Function代表一个输入一个输出（不同类型）  
\* UnaryOperator代表一个输入一个输出（相同类型）  
\* BiFunction代表两个输入一个输出（不同类型）  
\* BinaryOperator代表两个输入一个输出（相同类型）*





## 注解

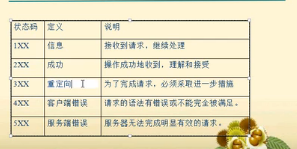


*//target用来声明当前自定义的注解适合适用于什么地方类，方法，变量，包。。。*@Target(ElementType.***METHOD***)  
*//Retention用来表示当前注解适用于什么环境：源码，类，运行环境级别，一般都是运行环境*@Retention(RetentionPolicy.***RUNTIME***)  
*//表示是否将注解展示在javadoc文件内*@Documented  
*//表示当前注解是否能被继承*@Inherited  
@**interface** MyAnnocation{  
  
}

# HTTP

协议：和现实中合同一样，一式多份，双方或者多方遵循规范。

状态码：



# SpringBoot