**JAVA装逼指南**

**目录**

# 第一章 基本语法

## （一）关键字

### 1、static

修饰变量、方法、静态块、静态内部内、静态导包

随着类的 加载而加载，只执行一次,并优先于主函数。存在于方法区中。

### 2、[final](https://www.jianshu.com/p/f68d6ef2dcf0)

1.final修饰的部分

类、方法、域、方法参数

2.作用

a.设计上：被final修饰的类不能被继承；被final修饰的方法不能被重写；被final修饰的变量的引用不能被修改。

b.效率上：被final修饰后，内联（inline在Java中就是编译器为程序做的一种优化操作）会起作用。

备注：内联是发生在编译期的一个优化操作。所做的优化操作的意义在于两个字**“替换” 。**

3.为什么Java匿名内部类访问的外部局部变量或参数需要被final修饰？

内部类对象的生命周期会超过局部变量的生命期。

局部变量的生命期：当该方法被调用时，该方法中的局部变量在栈中被创建（诞生），当方法调用结束时（执行完毕），退栈，这些局部变量全部死亡。而：内部类对象生命期，与其它类一样，当创建一个该局部类对象后，只有没有其它人再引用它时，它才能死亡。完全可能：一个方法已调用结束（局部变量已死亡），但该局部类的对象仍然活着。即：局部类的对象生命期会超过局部变量。

### 3、transient

用transient关键字标记的成员变量不参与序列化过程。

### 4、[volatile](https://www.jianshu.com/p/506c1e38a922)

如果一个字段被声明成**volatile**，java线程内存模型确保所有线程看到这个变量的值是一致的。

它比synchronized的使用和执行成本会更低，因为它不会引起线程上下文的切换和调度。

volatile是基于Memory Barrier实现的。如果一个变量是volatile修饰的，JMM会在写入这个字段之后插进一个Write-Barrier指令，并在读这个字段之前插入一个Read-Barrier指令。

Write-Barrier（写入屏障）将刷出所有在Barrier之前写入cache 的数据，因此，任何CPU上的线程都能读取到这些数据的最新版本。

底层基于C++的volatile实现，因为volatile自带了编译器屏障的功能，总能拿到内存中的最新值。

volatile修饰的变量进行写操作的时候会多了一行下面的汇编代码。



lock前缀指令的作用有两个：

1.将当前处理器缓存行的数据会写回到系统内存。

2.这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效。

备注：

测试代码

**public** **class** LazySingleton {

**private** **static** **volatile** LazySingleton *instance* = **null**;

**public** **static** LazySingleton getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** LazySingleton();

}

**return** *instance*;

}

**public** **static** **void** main(String[] args) {

LazySingleton.*getInstance*();

}

}

通过hsdis查看汇编代码

java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly -XX:FreqInlineSize=1 -XX:MaxInlineSize=1 -XX:CompileThreshold=1 -Xcomp -XX:CompileCommand=dontinline,\*LazySingleton.getInstance -XX:CompileCommand=compileonly,\*LazySingleton.getInstance -XX:LogFile=jit.log LazySingleton

lock addl $0x0,(%rsp)

### 5、synchronized

synchronized可以保证方法或代码块在运行时，同一时刻只有一个线程可以进入到临界区（互斥性），同时它还保证了共享变量的内存可见性。

synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。

syncrhoized又叫做内置锁，是通过锁对象的monitor的取用与释放来实现的。

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态。

**monitorenter**

线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

**monitorexit**

执行monitorexit的线程必须是objectref所对应的monitor的所有者。

指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

同步代码块使用了 monitorenter 和 monitorexit 指令实现。

同步方法中依靠方法修饰符上的 ACC\_SYNCHRONIZED 实现。

无论哪种实现，本质上都是对指定对象相关联的monitor的获取，这个过程是互斥性的，也就是说同一时刻只有一个线程能够成功，其它失败的线程会被阻塞，并放入到同步队列中，进入BLOCKED状态。

1．修饰块

**public** **class** BlockTest {

**public** **void** say() {

Object lock = **new** Object();

**synchronized**(lock){

System.***out***.println("");

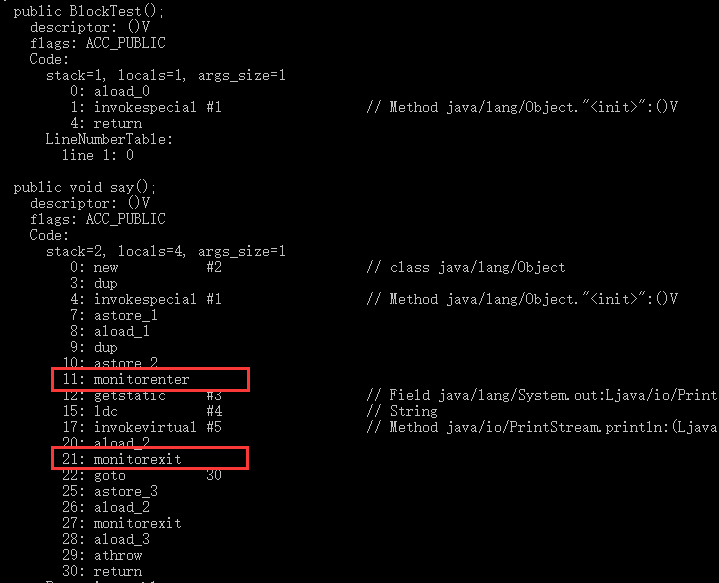
}

}

}

Javac Test.java

javap -l -v Test.class



2.修饰方法

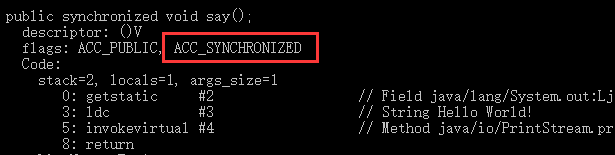
**public** **class** MethodTest {

**public** **synchronized** **void** say() {

System.***out***.println("Hello World!");

}

}



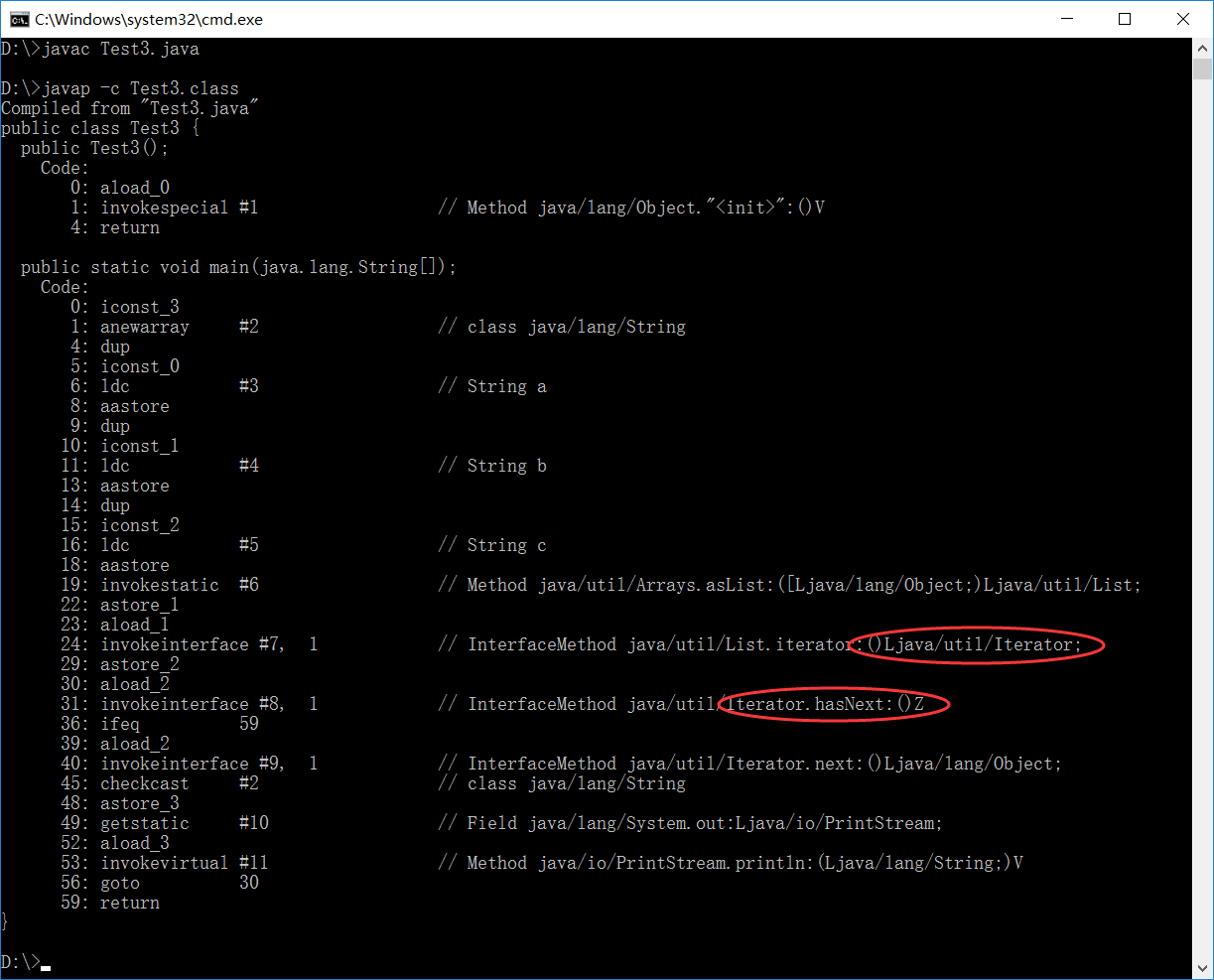
当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。在方法执行期间，其他任何线程都无法再获得同一个monitor对象。 其实本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。

### 6、import

引入包后发生了什么？静态引入？

## （二）foreach循环的原理

通过查看对应的字节码可知，foreach语法最终被编译器转为了对Iterator.hasNext()和对Iterator.next()的调用。



## （三）Serializable

### 1、实现序列化的步骤

File file = **new** File("E:/Serializable.txt");

ObjectOutputStream out = **new** ObjectOutputStream(**new** FileOutputStream(file));

out.writeObject(object);

out.close();

ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream(file));

Object newPerson = in.readObject();

in.close();

System.***out***.println(newPerson);

### 2、为什么一个类实现了Serializable接口，它就可以被序列化呢？

**private** **void** writeObject0(Object obj, **boolean** unshared) **throws** IOException {

......

**if** (obj **instanceof** String) {

writeString((String) obj, unshared);

} **else** **if** (cl.isArray()) {

writeArray(obj, desc, unshared);

} **else** **if** (obj **instanceof** Enum) {

writeEnum((Enum<?>) obj, desc, unshared);

} **else** **if** (obj **instanceof** Serializable) {

writeOrdinaryObject(obj, desc, unshared);

} **else** {

**if** (extendedDebugInfo) {

**throw** **new** NotSerializableException(cl.getName() + "\n" + debugInfoStack.toString());

} **else** {

**throw** **new** NotSerializableException(cl.getName());

}

}

......

}

### 3、默认序列化机制

如果仅仅只是让某个类实现Serializable接口，而没有其它任何处理的话，则就是使用默认序列化机制。

使用默认机制，在序列化对象时，不仅会序列化当前对象本身，还会对该对象引用的其它对象也进行序列化。

### 4、SerialVersionUID的作用

如果不手动指定给SerialVersionUID的值，系统会计算当前类的hash值并赋予给serialVersionUID,

假如序列化处或者反序列化的类被修改了，系统生成的两个serialVersionUID就会不一致，导致反序列化失败。

### 5、影响序列化

#### 1）transient关键字

当某个字段被声明为transient后，默认序列化机制就会忽略该字段。

static、transient修饰的字段不会被序列化。

#### 2）writeObject()方法与readObject()方法

#### 3）Externalizable接口

JDK中提供了另一个序列化接口--Externalizable。

**public** **interface** Externalizable **extends** java.io.Serializable {

**void** writeExternal(ObjectOutput out) **throws** IOException;

**void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException;

}

若使用Externalizable进行序列化，当读取对象时，会调用被序列化类的无参构造器去创建一个新的对象，然后再将被保存对象的字段的值分别填充到新对象中。

#### 4）readResolve()

为了能在序列化过程仍能保持单例的特性，可以在类中添加一个readResolve()方法，在该方法中直接返回类的单例对象。

**总结：**

无论是实现Serializable接口，或是Externalizable接口，当从I/O流中读取对象时，readResolve()方法都会被调用到。

实际上就是用readResolve()中返回的对象直接替换在反序列化过程中创建的对象，而被创建的对象则会被垃圾回收掉。

### 6、常见第三方库？

1）Gson

2）fastjson

## （四）注解的原理

### 1、注解的类型

1.java自带的标准注解

2.元注解（用于定义注解的注解）

3.自定义注解

### 2、原理

注解被编译后的本质就是一个继承Annotation接口的接口，所以@Test其实就是“public interface Test extends Annotation”，当我们通过AnnotationTest.class.getAnnotation(Test.class)调用时，JDK会通过动态代理生成一个实现了Test接口的对象，并把将RuntimeVisibleAnnotations属性值设置进此对象中，此对象即为Test注解对象，通过它的value()方法就可以获取到注解值。

# 第二章 并发相关

## （一）AbstractQueuedSynchronizer

## （二）ReentrantLock

## （三）ReentrantReadWriteLock

## （四）StampedLock

## （五）CopyOnWriteArrayList

## （六）CopyOnWriteArraySet

## （六）SynchronousQueue

## （六）ArrayBlockintQueue

## （七）PriorityQueue

## （八）ConcurrentHashMap

**public** **class** ConcurrentHashMap<K,V> **extends** AbstractMap<K,V>

**implements** ConcurrentMap<K,V>, Serializable

**static** **class** Segment<K,V> **extends** ReentrantLock **implements** Serializable {

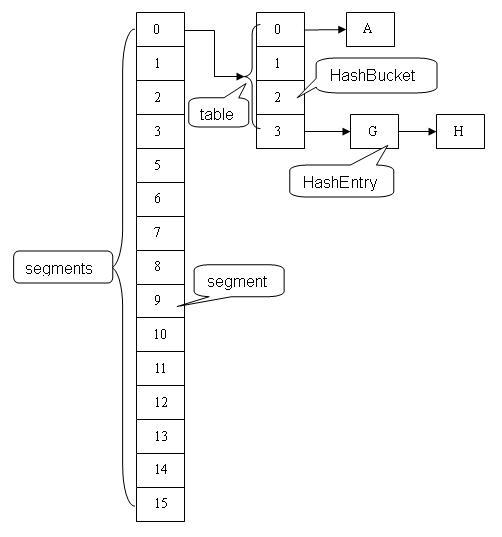
**private** **static** **final** **long** ***serialVersionUID*** = 2249069246763182397L;

**final** **float** loadFactor;

Segment(**float** lf) { **this**.loadFactor = lf; }

}

### 1、JDK1.8之前



ConcurrentHashMap采用分段锁的机制，底层采用数组+链表的存储结构。

其包含两个核心静态内部类 Segment和HashEntry。

Segment继承ReentrantLock用来充当锁的角色，每个 Segment 对象守护每个散列映射表的若干个桶。

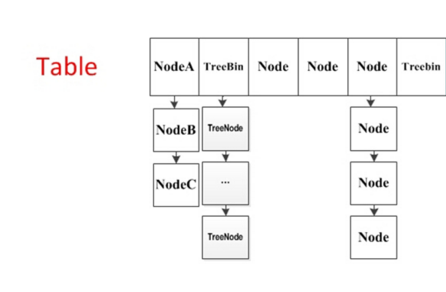
1．HashEntry 用来封装映射表的键 / 值对；

2．每个桶是由若干个 HashEntry 对象链接起来的链表。

一个 ConcurrentHashMap 实例中包含由若干个 Segment 对象组成的数组。

### 2、JDK1.8

1.8的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层采用数组+链表+红黑树的存储结构。



### 3、锁分段技术

### 4、读是否要加锁，为什么？

### 5、迭代器是强一致性的迭代器还是？

## （九）ConcurrentLinkedQueue

## （十）ConcurrentLinkedDeque

## （十）LinkedBlockingQueue

ConcurrentLinkedQueue、LinkedBlockingQueue区别？

## （十）DelayQueue

## （十）LinkedTransferQueue

## （十）PriorityBlockingQueue

## （十一）CountDownLatch

## （十二）CyclicBarrier

## （十三）Exchanger

## （十四）Semaphore

## （十五）分布式锁

# 第三章 多线程

## （一）线程模型

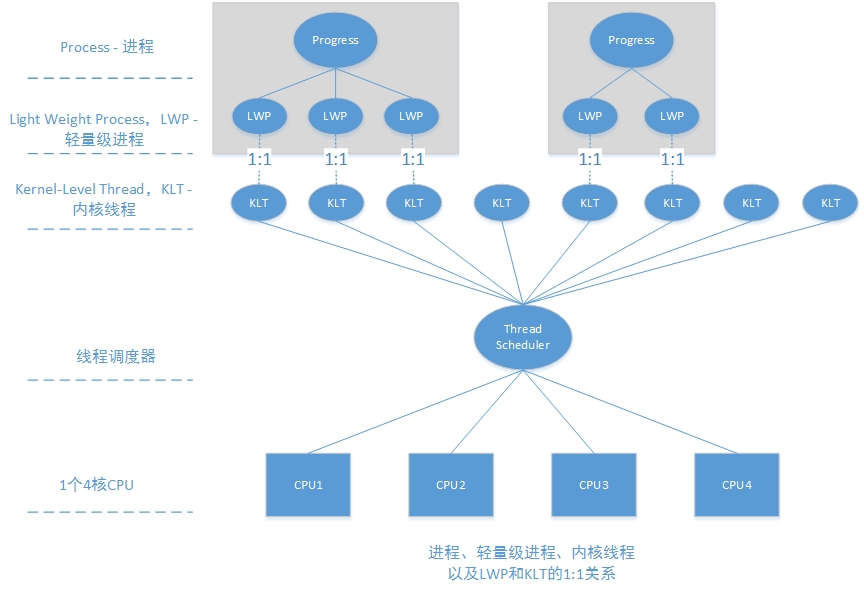
线程是比进程更轻量级的调用单位。实现线程主要有3种方式：

1.使用内核线程（一对一线程模型）实现

2.使用用户线程（一对N线程模型）实现

3.使用用户线程加轻量级进程混合（N对M线程模型）实现

SUN JDK采用了内核线程（一对一线程模型）。这种方式的线程调度流程如下：



**调度流程：**

首先，从OS方面来说：如果操作系统内核中包括了一个线程调度器（Scheduler） ，内核可以操作调度器对内核线程进行调度，将线程任务映射到各个CPU上。那么，这样的内核就叫多线程内核 。

其次，从程序来说：应用程序不能直接使用内核线程，只能去使用内核线程的一种高级接口-轻量级进程（LWP）。对于编程语言来说，它们操作的实际上是这个LWP，LWP就是此时就可以被叫线程 。

最后，轻量级进程和线程之间的映射关系是1：1的，所以，这种方式叫做一对一线程模型 。

**Java线程调度：**

线程调度是指系统为线程分配处理器使用权的过程，主要调度方式有两种

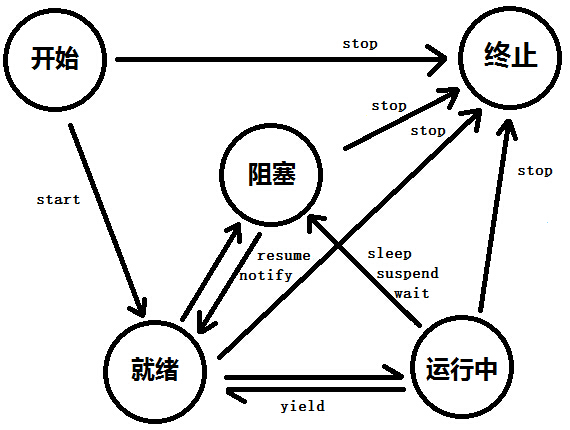
1.协同式线程调度

一个线程执行完通知另一个线程。已经很少使用，很容易造成阻塞。

2.抢占式线程调度

主流方式，由系统来根据一系列复杂的规则为每个线程分配执行时间，线程的切换不是由线程自己做主（Java可以有Thread.yield()来让出执行时间，但是没有获取执行时间的方式）。

## （二）线程有哪些状态？状态转化？



## （三）常用的线程池有几种？这几种线程池间的区别与联系？实现原理？

## （四）synchronized和reentrantLock的区别？synchronized锁普通方法和锁静态方法、死锁的原理及排查方法？

## （五）条件锁、读写锁，分拆锁、分离锁、ThreadLocal、copyOnWrite

## （六）自旋锁、偏向锁、轻量级锁、重量级锁

## （七）乐观锁、悲观锁

**悲观锁：**每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。

**乐观锁：**每次不加锁，假设没有冲突去完成某项操作，如果因为冲突失败就重试，直到成功为止。

## （八）什么是守护线程？线程和进程的差别是什么？

## （九）sleep、yield、suspend和wait区别？notify和notifyAll区别？

### 1、sleep

* Thread.sleep(0) 进入就绪状态
* Thread.sleep(1) 进入阻塞状态

### 2、yield

* 如果调用 Yield，只是把当前线程放入到就绪队列中，而不是阻塞队列
* 如果没有找到其它就绪态的线程，则当前线程继续运行
* 比 Thread.Sleep(0) 速度要快，可以让低于当前优先级的线程得以运行

### 3、suspend

### 4、wait

### 5、notify

### 6、notifyAll

## （十）start与run方法的区别

直接调用run()方法，它不会创建新的线程也不会执行调用线程的代码。

## （十一）CycliBarriar和CountdownLatch有什么区别？

CyclicBarrier可以重复使用已经通过的障碍，而CountdownLatch不能重复使用。

## （十二）Fork/Join框架

## （十三）Callable/Future/FutureTask

## （十四）生产者/消费者模式

最小堆、最大堆

1、FIFO

2、Priority

# 第四章 IO

## （一）File IO

File IO是阻塞IO

## （二）NIO

Buffer

Channel

Selector

MappedByteBuffer

## （三）Socket IO

阻塞IO、非阻塞IO、多路复用IO、异步IO等四种IO模型的区别？

NIO的原理？NIO属于哪一种IO模型？NIO的三大组成？

NIO模型，select/epoll的区别，多路复用的原理

# 第五章 设计模式

## （一）设计模式六大原则

### 1、单一职责原则

一个类只负责一项职责。不要存在多于一个导致类变更的原因

**优点：**

1.降低类的复杂度。

2.提高类的可读性，提高系统的可维护性。

3.降低变更引起的风险。

### 2、里氏替换原则

子类可以扩展父类的功能，但不能改变父类原有的功能。

### 3、依赖倒置原则

高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

### 4、接口隔离原则

类间的依赖关系应该建立在最小的接口上。客户端不应该被强迫地依赖那些根本用不上的方法。

### 5、迪米特法则

类间解耦。一个对象应该对其他对象保持最少的了解。

### 6、开闭原则

对扩展开放，对修改关闭。即不要改以前写的代码，应该加一些代码去扩展原来的功能，来实现新的需求。

**记忆：**

一个单身（单一职责）的里氏（里氏替换原则）拿着一把半开半闭（开闭原则）的扇子，在看接口分离（接口分离）问题，旁边还有一个米老鼠（迪米特）在颠倒（依赖倒置）是非。

## （二）JDK库中的设计模式

### 1）单例模式

* java.lang.Runtime#getRuntime()
* java.awt.Desktop#getDesktop()

### 2）工厂模式

String.*valueOf*(1);

Boolean.*valueOf*(**true**);

### 3）抽象工厂模式

### 4）装饰模式

### 5）责任链模式

### 6）适配器模式

### 7）桥接模式

### 8）组合模式

### 9）装饰者模式

### 10）享元模式

### 11）代理模式

### 12）抽象工厂模式

### 13）建造模式(Builder)

### 14）原型模式

### 15）解释器模式

### 16）迭代器模式

### 17）中介者模式

### 18）备忘录模式

### 19）观察者模式

### 20）状态模式

### 21）模板方法模式

### 22）访问者模式

## （三）项目中用过哪些设计模式，如何使用？

### 1、单利模式

#### 1）枚举实现

#### 2）内部静态类

### 2、代理模式

#### 1）静态代理

静态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类.

缺点：代理类太多，而且不好维护。

#### 2）动态代理

根据传入的参数（class名称，需要代理的接口数组，访问控制符）来动态生成一个新的类，这个类会被JVM动态加载到内存中，然后实例化，最终完成代理的功能。

代理对象不需要实现接口,但是目标对象一定要实现接口。

**public** Object getProxyInstance() {

**return** Proxy.*newProxyInstance*(target.getClass().getClassLoader(), target.getClass().getInterfaces(),

**new** InvocationHandler() {

@Override

**public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {

System.***out***.println("开始事务2");

// 执行目标对象方法

Object returnValue = method.invoke(target, args);

System.***out***.println("提交事务2");

**return** returnValue;

}

});

}

**JDK动态代理原理：**

Proxy.newInstance做了三件事：

1.检查是否拥有对被代理类的访问权限；

2.检查内存中是否存在已经生成好的类，如果没有，就生成一个并返回；

**public** **final** **class** $Proxy0 **extends** Proxy **implements** IUserDao

通过InvocationHandler的invoke方法来调用被代理类的方法。

3.调用生成的代理类的构造方法创建新的类，并返回。

注：可通过设置参数将生成的代理类保存到本地。

System.*setProperty*("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");

#### 3）Cglib代理

**public** **class** ProxyFactory **implements** MethodInterceptor {

**private** Object target;

**public** ProxyFactory(Object target) {

**this**.target = target;

}

**public** Object getProxyInstance() {

// 1.工具类

Enhancer en = **new** Enhancer();

// 2.设置父类

en.setSuperclass(target.getClass());

// 3.设置回调函数

en.setCallback(**this**);

// 4.创建子类(代理对象)

**return** en.create();

}

@Override

**public** Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) **throws** Throwable {

System.***out***.println("开始事务...");

// 执行目标对象的方法

Object returnValue = method.invoke(target, args);

System.***out***.println("提交事务...");

**return** returnValue;

}

}

**Cglib代理原理：**

1.查找类的所有非**final**、非**static**、非**private**类型的方法定义；

2.将这些方法的定义转换成字节码；

3.将组成的字节码转换成相应的代理的class对象；

**public** **class** UserDao$$EnhancerByCGLIB$$c37a2673 **extends** UserDao **implements** Factory

无法代理final类。

每个函数有两个反射方法，一个是通过原生反射获得的，另外一个是通过Cglib构建的。

针对委托类中的每个方法，在代理类中有两个实现，一个实现直接调用委托类方法，而另外一个则通过代理的回调进行调用。

**public** **class** UserDao$$FastClassByCGLIB$$6dd6ee60 **extends** FastClass

对应invoke方法中使用的fastclass。

**public** **class** UserDao$$EnhancerByCGLIB$$c37a2673$$FastClassByCGLIB$$85fd91c0 **extends** FastClass

对应invokeSuper中使用的fastclass。

两个FastClass文件，是为了减少反射调用的时间，将反射调用转化为直接调用。

4.实现MethodInterceptor接口，用来处理对代理类上所有方法的请求（这个接口和JDK动态代理InvocationHandler的功能和角色是一样的）。

注：可通过设置参数将生成的代理类保存到本地。

System.*setProperty*(DebuggingClassWriter.***DEBUG\_LOCATION\_PROPERTY***, "E:/cglib");

**JDK代理与Cglib代理区别：**

1.Java动态代理只能够对接口进行代理，不能对普通的类进行代理（因为所有生成的代理类的父类为Proxy，Java类继承机制不允许多重继承）；CGLIB能够代理普通类；

2.Java动态代理使用Java原生的反射API进行操作，在生成类上比较高效；CGLIB使用ASM框架直接对字节码进行操作，在类的执行过程中比较高效。

### 3、工厂模式

### 4、策略模式

### 5、模板模式

### 6、装饰模式

### 7、观察者模式

## （四）知道设计模式的优缺点？

## （五）能画出常用设计模式的UML图？

# 第六章 JDK源码

## （一）Unsafe

Java最初被设计为一种安全的受控环境。尽管如此，HotSpot还是包含了一个后门sun.misc.Unsafe，提供了一些可以直接操控内存和线程的底层操作。Unsafe被JDK广泛应用于java.nio和并发包等实现中，这个不安全的类提供了一个观察HotSpot JVM内部结构并且可以对其进行修改，但是不建议在生产环境中使用。

### 1、Unsafe的限制

1.私有的构造器

2.工厂方法getUnsafe()的调用器只能被Bootloader加载，否则抛出SecurityException 异常

### 2、API

1. objectFieldOffset

获取指定实例变量在对象内存中的偏移量。

long offset = UNSAFE.objectFieldOffset(Test.class.getDeclaredField("state"));

2. compareAndSwapInt

通过比较并替换的机制，修改指定偏移量内存的值。

UNSAFE.compareAndSwapInt(unsafeTest, offset, 0, 1)

3.park：挂起某一线程

4.unpark：唤醒某一线程

## （二）java.lang.ref

对象的有效性=可达性+引用类型

### 1、ReferenceQueue

**public** **class** ReferenceQueue<T> {

**private** **volatile** Reference<? **extends** T> head = **null**;

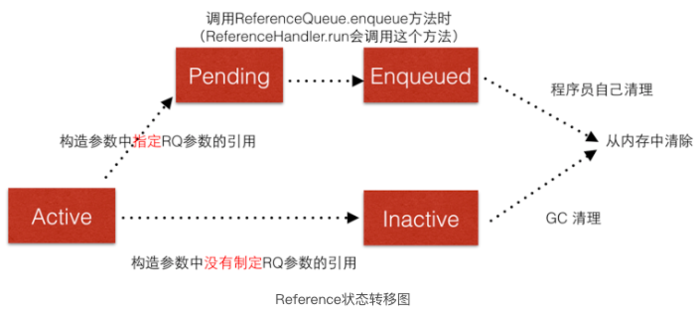
......

}

存放引用的队列，保存的是Reference对象。其作用在于Reference对象所引用的对象被GC回收时，该Reference对象将会被加入引用队列中（ReferenceQueue）的队列末尾。

### 2、Reference

引用对象的抽象基类，这个类中定义了所有引用对象的常用操作。



### 3、PlantomReference（虚引用）

该引用必须和引用队列（ReferenceQueue）一起使用，一般用于实现追踪垃圾收集器的回收动作，比如在对象被回收的时候，会调用该对象的finalize方法，在使用虚引用可以实现该动作，也更加安全。

### 4、SoftReference（软引用）

非强引用，但是可以通过软引用对象来访问。软引用的对象，只有在内存不足的时候（抛出OOM异常前），垃圾收集器会决定回收该软引用所指向的对象。

### 5、WeakReference（弱引用）

非强引用和软引用，但是可以通过弱引用对象来访问。弱引用的对象，不管内存是否足够，只要被垃圾收集器发现，该引用的对象就会被回收。

### 6、FinalReference（强引用）

程序中有直接可达的引用，而不需要通过任何引用对象。如果一个对象具有强引用，GC绝不会回收它，当内存空间不足时，JVM宁愿抛出OOM。new出来的对象是典型的强引用。

### 7、Finalizer

## （三）java.util

### 1、ArrayList

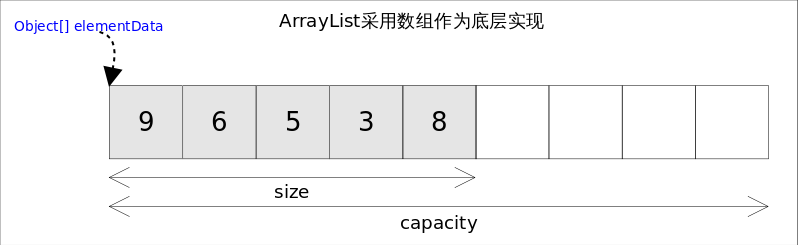
**public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, java.io.Serializable {

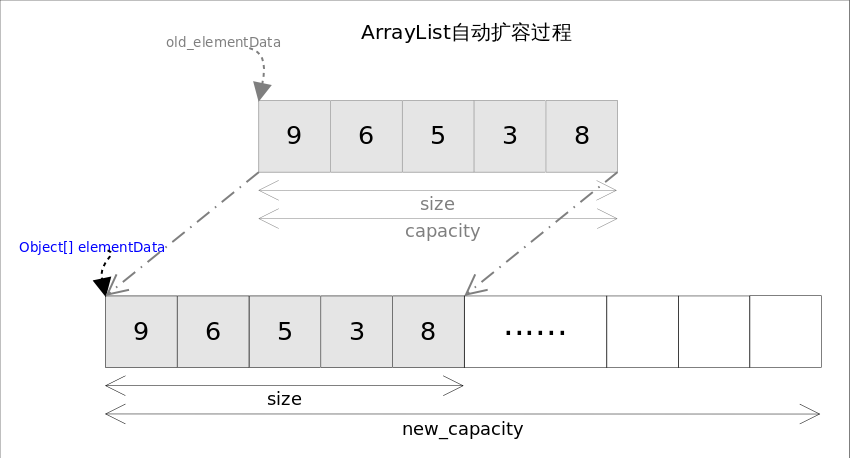
**private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 10;

**transient** Object[] elementData;

......

}





ArrayList底层以数组实现，允许重复，默认第一次插入元素时创建数组的大小为10，超出限制时会增加50%的容量，每次扩容都底层采用System.arrayCopy()复制到新的数组。

**add：**

1.调用ensureCapacityInternal方法，确保数组的容量能放下；如果存储的数量大于数组的长度，则扩容为原来的1.5倍；旧数据赋值通过System.arraycopy实现。

2.将数据放在数组中。

**get：**

检查index范围，如果超出范围，抛异常；否则，返回数据。

**remove：**

检查index范围，如果超出范围，抛异常；否则，通过System.arraycopy将(index+1)后面的数据往前移一位；并将删除的对象的引用标记为null。

**和LinkedList的区别**

1、ArrayList是实现了基于动态数组的数据结构，LinkedList基于链表的数据结构。

2、对于随机访问get和set，ArrayList觉得优于LinkedList，因为LinkedList要移动指针。

3、对于新增和删除操作add和remove（不是在尾部添加删除），LinkedList比较占优势，因为ArrayList要移动数据。

**和Vector的区别**

1、Vector和ArrayList几乎是完全相同的,唯一的区别在于Vector是同步类(synchronized)，属于强同步类。因此开销就比ArrayList要大，访问要慢。

2、Vector每次扩容请求其大小的2倍空间，而ArrayList是1.5倍。

3、Vector还有一个子类Stack.

### 2、LinkedList

**public** **class** LinkedList<E>

**extends** AbstractSequentialList<E>

**implements** List<E>, Deque<E>, Cloneable, java.io.Serializable

**private** **static** **class** Node<E> {

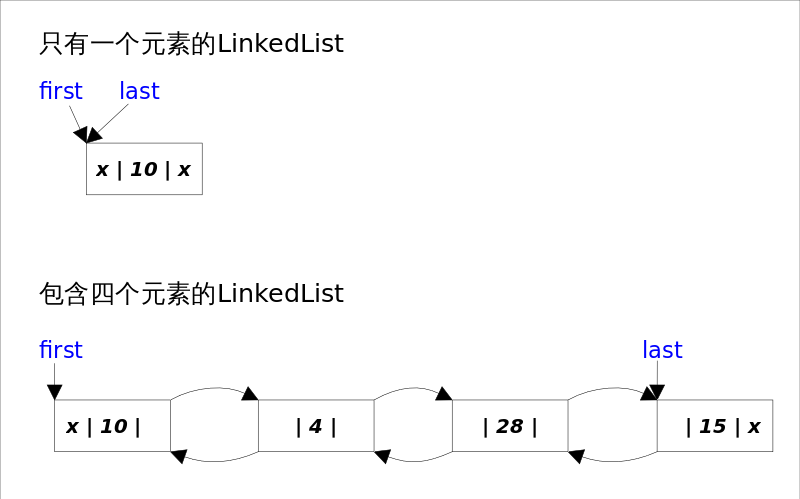
E item;

Node<E> next;

Node<E> prev;

......

}



LinkedList以双向链表实现。

**get：**

检查index范围，如果超出范围，抛异常；否则遍历链表，如果index<size/2，则从链表头开始遍历；否则从链表尾开始遍历。

### 3、Vector

**public** **class** Vector<E>

**extends** AbstractList<E>

**implements** List<E>, RandomAccess, Cloneable, java.io.Serializable

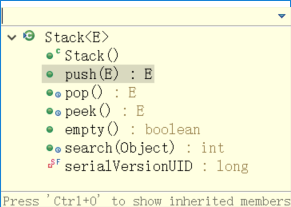
**特征：**

1.初始容量为10；扩容是时，新增为原来的两倍。

2.方法加了**synchronized**。

### 4、Stack

**public class** Stack<E> **extends** Vector<E>



**特征：**

1.last-in-first-out (LIFO)

2.方法加了**synchronized**。

### 5、ArrayDeque

**public** **class** ArrayDeque<E> **extends** AbstractCollection<E> **implements** Deque<E>, Cloneable, Serializable {

**transient** Object[] elements;

**transient** **int** head;

**transient** **int** tail;

// Must be a power of 2.

**private** **static** **final** **int** MIN\_INITIAL\_CAPACITY = 8;

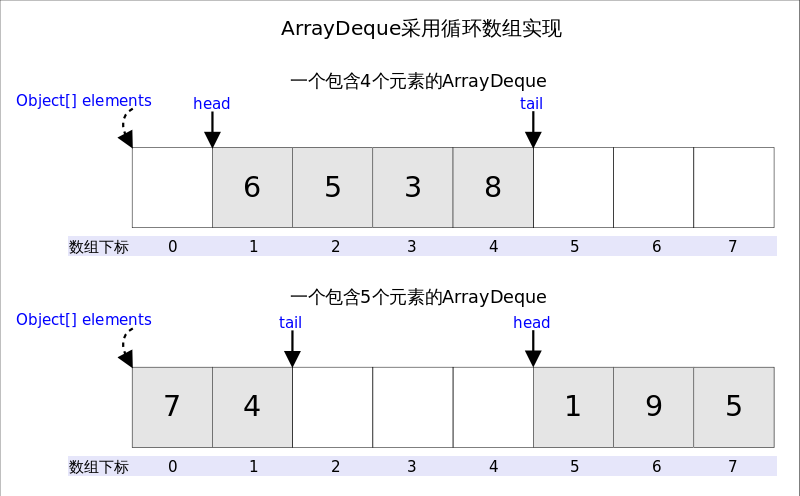
......

**public** **int** size() {

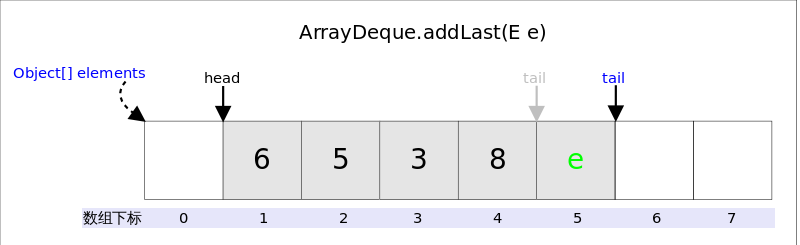
**return** (tail - head) & (elements.length - 1);

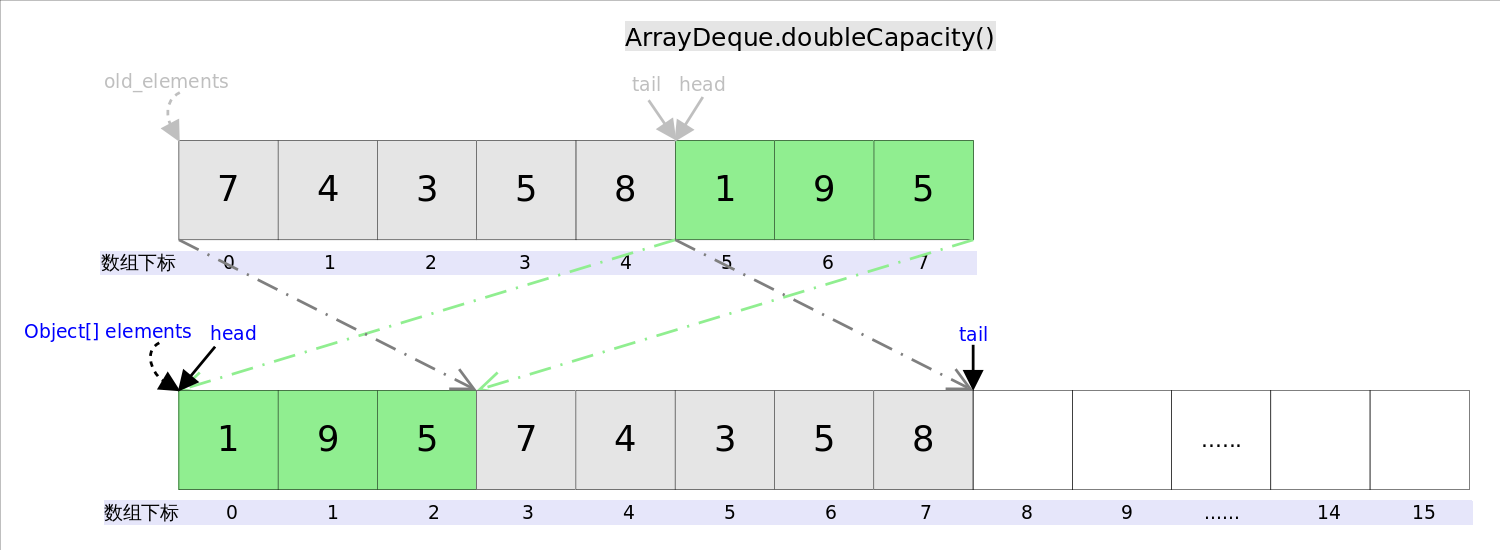
}

}



**head指向首端第一个有效元素，tail指向尾端第一个可以插入元素的空位。**因为是循环数组，所以head不一定总等于0，tail也不一定总是比head大。





**特征：**

ArrayDeque采用了循环数组的方式来完成双端队列的功能。 用两个int的head、tail分别表示队列的头部下标、尾部下标。循环是通过表达式(tail = (tail + 1) & (elements.length - 1))实现的。

1. 数组的容量为2的次方，每次新增1倍。（如果保证2的次方，见[hashmap](#_8、HashMap)部分）

2. 非线程安全的，不支持并发访问和修改。

3. 支持fast-fail。

4. 作为栈使用的话比比栈要快。

5. 当队列使用比LinkedList要快。

6. null元素被禁止使用。

### 6、HashSet

**public** **class** HashSet<E>

**extends** AbstractSet<E>

**implements** Set<E>, Cloneable, java.io.Serializable

{

**private** **transient** HashMap<E,Object> map;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

**public** HashSet() {

map = **new** HashMap<>();

}

**public** HashSet(Collection<? **extends** E> c) {

map = **new** HashMap<>(Math.*max*((**int**) (c.size()/.75f) + 1, 16));

addAll(c);

}

**public** HashSet(**int** initialCapacity, **float** loadFactor) {

map = **new** HashMap<>(initialCapacity, loadFactor);

}

**public** HashSet(**int** initialCapacity) {

map = **new** HashMap<>(initialCapacity);

}

HashSet(**int** initialCapacity, **float** loadFactor, **boolean** dummy) {

map = **new** LinkedHashMap<>(initialCapacity, loadFactor);

}

**public** **boolean** add(E e) {

**return** map.put(e, ***PRESENT***)==**null**;

}

}

HashSet底层用HashMap实现；HashSet的value用map的Key存储，map的value用一个虚拟的**static** **final**修饰的Object代替；是对HashMap的简单包装，对HashSet的函数调用都会转换成合适的HashMap方法。

### 7、BitSet

**public** **class** BitSet **implements** Cloneable, java.io.Serializable {

**private** **final** **static** **int** ***ADDRESS\_BITS\_PER\_WORD*** = 6;

**private** **final** **static** **int** ***BITS\_PER\_WORD*** = 1 << ***ADDRESS\_BITS\_PER\_WORD***;

**private** **final** **static** **int** ***BIT\_INDEX\_MASK*** = ***BITS\_PER\_WORD*** - 1;

**private** **long**[] words;

**private** **void** ensureCapacity(**int** wordsRequired) {

**if** (words.length < wordsRequired) {

**int** request = Math.*max*(2 \* words.length, wordsRequired);

words = Arrays.*copyOf*(words, request);

sizeIsSticky = **false**;

}

}

**public** **void** set(**int** bitIndex) {

**if** (bitIndex < 0)

**throw** **new** IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

**int** wordIndex = *wordIndex*(bitIndex);

expandTo(wordIndex);

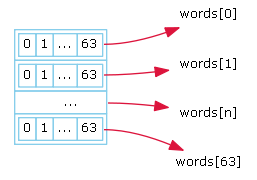
words[wordIndex] |= (1L << bitIndex);

checkInvariants();

}

......

}



底层用一个long[ ]实现，每个数组元素占64位。每次扩容为原来的2倍。

### 8、TreeSet

**public** **class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, java.io.Serializable {

**private** **transient** NavigableMap<E, Object> m;

**private** **static** **final** Object ***PRESENT*** = **new** Object();

**public** TreeSet() {

**this**(**new** TreeMap<E,Object>());

}

**public** **boolean** add(E e) {

**return** map.put(e, ***PRESENT***)==**null**;

}

......

}

TreeSet底层用TreeMap实现；TreeSet的value用map的Key存储，map的value用一个虚拟的**static** **final**修饰的Object代替；是对TreeMap的简单包装，对TreeSet的函数调用都会转换成合适的TreeMap方法。

### 9、LinkedHashSet

**public** **class** LinkedHashSet<E>

**extends** HashSet<E>

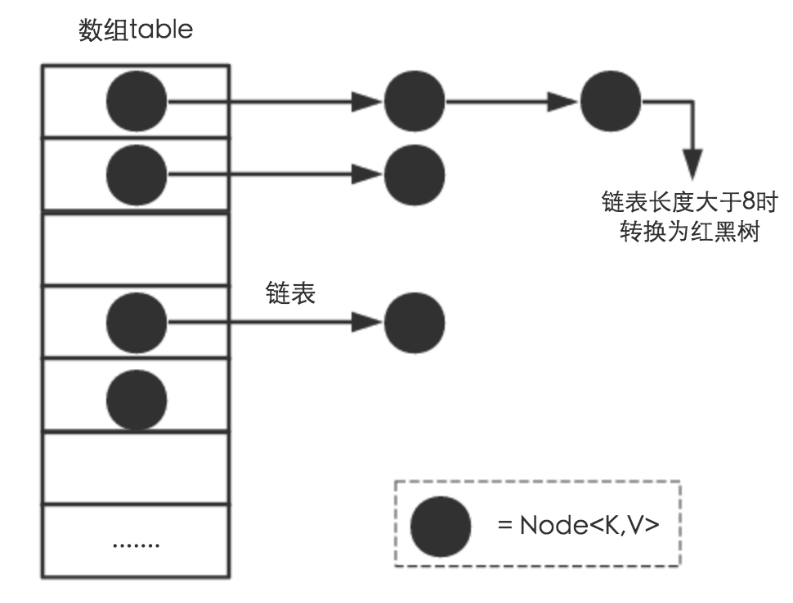
**implements** Set<E>, Cloneable, java.io.Serializable

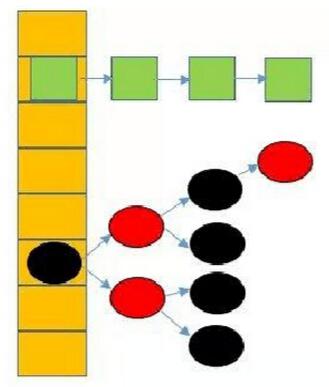
LinkedHashSet是对LinkedHashMap的简单包装，对LinkedHashSet的函数调用都会转换成合适的LinkedHashMap方法。

### 10、HashMap

**public** **class** HashMap<K,V> **extends** AbstractMap<K,V>

**implements** Map<K,V>, Cloneable, Serializable



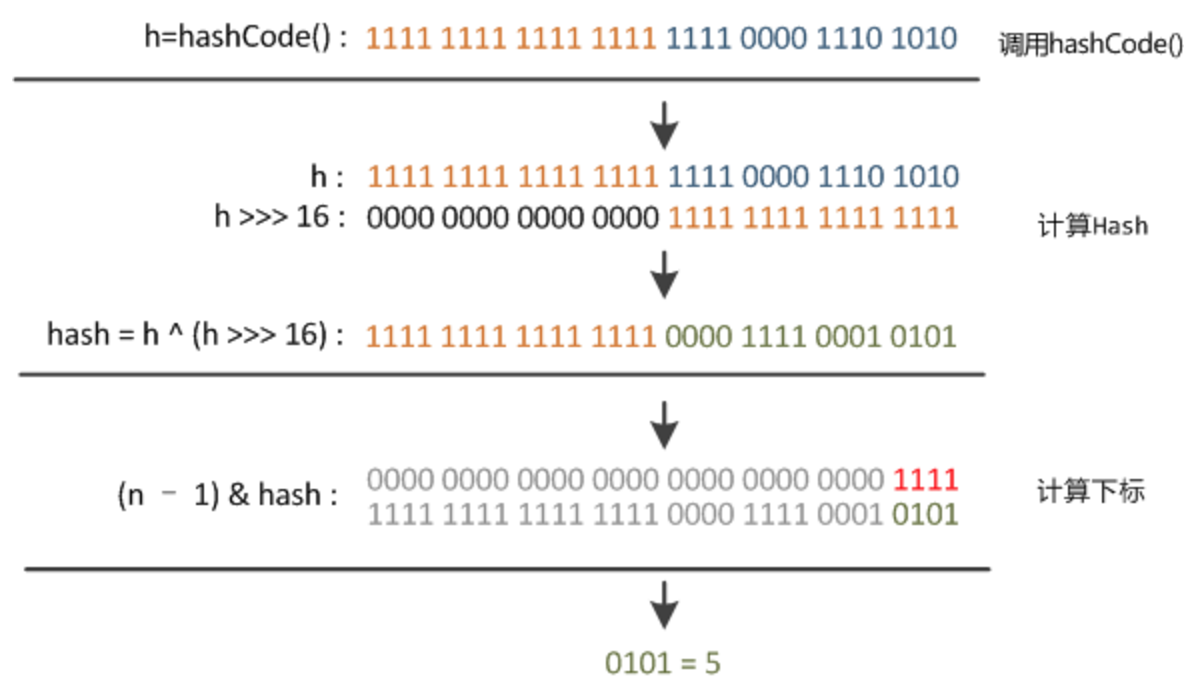


HashMap是基于哈希表实现的，每一个元素是一个key-value对。HashMap底层是一个Node数组，当发生hash冲突的时候，hashmap是采用链表的方式来解决的，在对应的数组位置存放链表的头结点。

对链表而言，新加入的节点会从头结点加入。容量不足（超过了阀值，默认为16）时，会自动增长。

HashMap是非线程安全的，只是用于单线程环境下，多线程环境下可以采用concurrent并发包下的concurrentHashMap。

**Hash函数的实现**：



**值及其作用：**

size：记录HashMap的底层数组中已用槽的数量。

DEFAULT\_LOAD\_FACTOR：默认加载因子为0.75。这是对空间和时间效率的一个平衡选择。（当loadFactor大于1时，会发生碰撞；当loadFactor特别特别大，且插入的元素特别多时，可能会生成红黑树。）

threshold：它是HashMap的阈值，用于判断是否需要调整HashMap的容量（threshold = 容量\*加载因子）。

TREEIFY\_THRESHOLD：链表转化为红黑树的阈值（8）。当链表的个数超过这个值时，需要使用红黑树节点替换链表节点。

UNTREEIFY\_THRESHOLD：红黑树还原为链表的阈值（6）。当扩容时，node的个数小于这个值，就会把树还原（切分）为链表结构。

MIN\_TREEIFY\_CAPACITY：哈希表的最小树形化容量（64）。当哈希表中的数组的长度大于这个值时，才会将链表转红黑树。

**put过程：**

1.如果当前map中无数据，执行resize方法；

2.如果要插入的键值对要存放的这个位置刚好没有元素，那么把他封装成Node对象，并放在这个位置上；

3.如果发生碰撞，判断node的类型是红黑树还是链表；

3.1.如果为红黑树，则将K-V对插在红黑树对应的位置

3.2.如果为链表，遍历链表（有三个if判断）

a.如果为链表最后一个node，则将新的node节点插入到链表尾；

b.插入完后，如果链表的node数量大于8，则将链表转为红黑树的操作；如果当前哈希表为空或数组长度小于64，会扩容；否则转化为红黑树。

转化的过程：先遍历链表，将链表的节点转化为红黑树的节点；然后将链表转化为红黑树。

c.遍历链表时，如果key已存在，则直接break循环。

4.判断是否要扩容；

5.返回。

HashMap中key和value都允许为null。key为null的键值对永远都放在以table[0]为头结点的链表中。

**get过程：**

1．对key的hashCode()做hash，然后再计算index；如果bucket里的第一个节点直接命中则返回；

2.如果有冲突，则通过key.equals(k)去查找对应的entry。

若为树，则在树中通过key.equals(k)查找。

若为链表，则在链表中通过key.equals(k)查找。

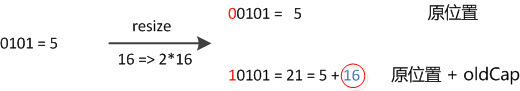
**remove过程：**

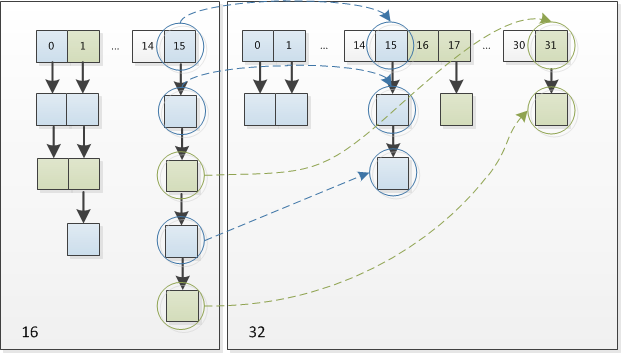
对key的hashCode()做hash，然后再计算index；如果node==null，直接返回；否则，找到key和hash相等的node；如果node为红黑数，则从树中删除；如果为链表，则从链表中删除。

**如何确定key在数组的位置？**

取模算法中的除法运算效率很低，在HashMap中通过h&(n-1)替代取模，得到所在数组位置，效率会高很多。（前提是保证数组的容量是2的整数倍）

**扩容机制：**





1.数组容量扩大一倍

2.遍历旧的数组（有三个if判断）

a.如果node的next为空，则直接将旧的node放在新的数组对应的位置；

b.如果node的next为红黑树

以链表遍历的形式遍历红黑树，将其拆分为两个链表，一个是位置不变的，一个是位置变为(i+oldCap)。如果新链表中的节点数小于等于6，会将红黑树转化为链表；否则，将新生成的链表转红黑树。

c.如果node的next为链表，则根据(hash&oldCap==0)将node放在两个两个新的链表中，一个是位置不变的，一个是位置变为(i+oldCap)，然后将新数组中链表的头节点分别指向它们。

3.返回扩容后的数组。

**hashmap扩容情况：**

1.put操作时，如果map数组的长度为0；

2.发生碰撞，且链表转红黑树判断时，数组的长度小于64时；

3.当K-V对的个数大于加载因子\*数组的长度。

**我们分析下为什么哈希表的容量一定要是2的整数次幂？**

hashmap根据h & (length-1)计算node的位置，如果length不为2的次幂，其对应的二进制数有为0的位，这样会导致二进制为0的位置永远存不了数据，数组可用的位置比真实长度比了很多，会浪费空间，还会增加hash碰撞的几率。

**如何保证容量为2的N次方的？**

**static** **final** **int** tableSizeFor(**int** cap) {

**int** n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

**return** (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

1.对于容量 cap ，在转化成二进制之后，总会找到第一个 1 ，如 01xxxxxx；

2.在和自身右移一位做或运算之后，能保证第一个 1 的后面一位也是 1 ，这样就有连续的两位是 1 了，即 01xxxxxx | 001xxxxx = 011xxxxx；

3.同理，依次做或运算之后，能保证最高是 1 的那一位之后的所有位都是 1 ，即 011xxxxx | 00011xxx = 01111xxx ，直至得到 01111111；

4.最后，再进行 n + 1 就得到了大于当前数的下一个2的N次方。

5.第一步的 n = cap - 1 是为了让本身是2的N次方的数，得到的结果还是自己。

**Fail-Fast机制：**

java.util.HashMap不是线程安全的，因此如果在使用迭代器的过程中有其他线程修改了map，那么将抛出ConcurrentModificationException，这就是所谓fail-fast策略。

**HashMap序列化：**

读写Map是根据h&(n-1)来确定，而Object.hashcode()是native方法, 不同的JVM里可能是不一样的，所以要重写序列化方法。

备注：jdk7碰撞时，node插在头部，这种情况在多线程下容易产生死环；jdk8插在尾部。

### 11、TreeMap

**public** **class** TreeMap<K, V> **extends** AbstractMap<K, V> **implements** NavigableMap<K, V>, Cloneable, java.io.Serializable {

**private** **final** Comparator<? **super** K> comparator;

**private** **transient** Entry<K, V> root;

**static** **final** **class** Entry<K, V> **implements** Map.Entry<K, V> {

K key;

V value;

Entry<K, V> left;

Entry<K, V> right;

Entry<K, V> parent;

**boolean** color = BLACK;

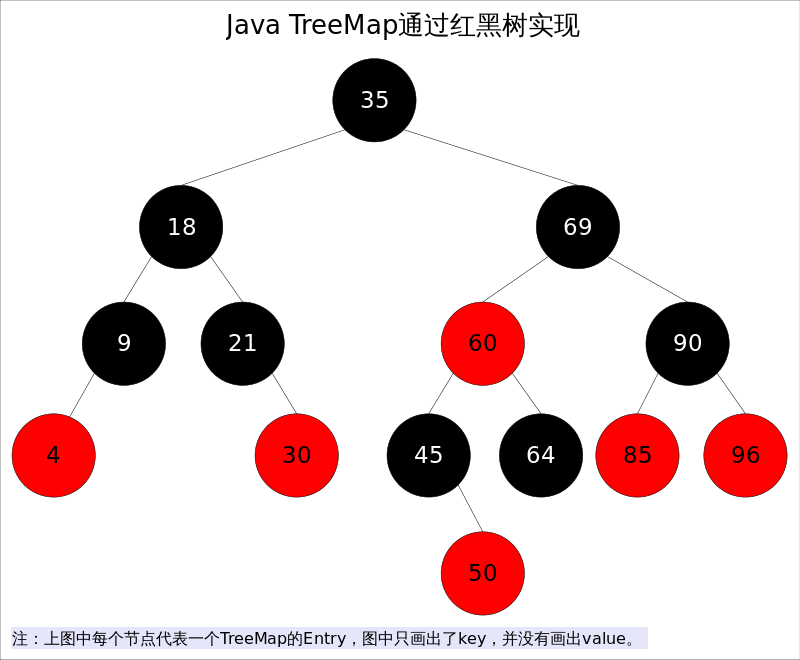
......

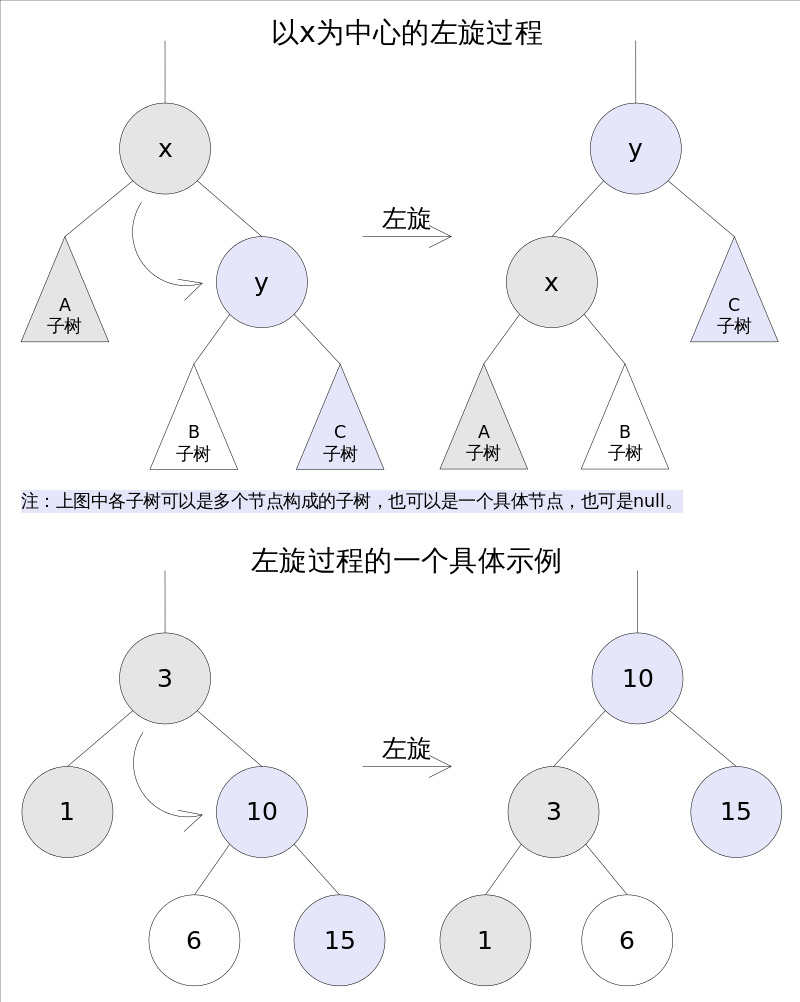
}

......

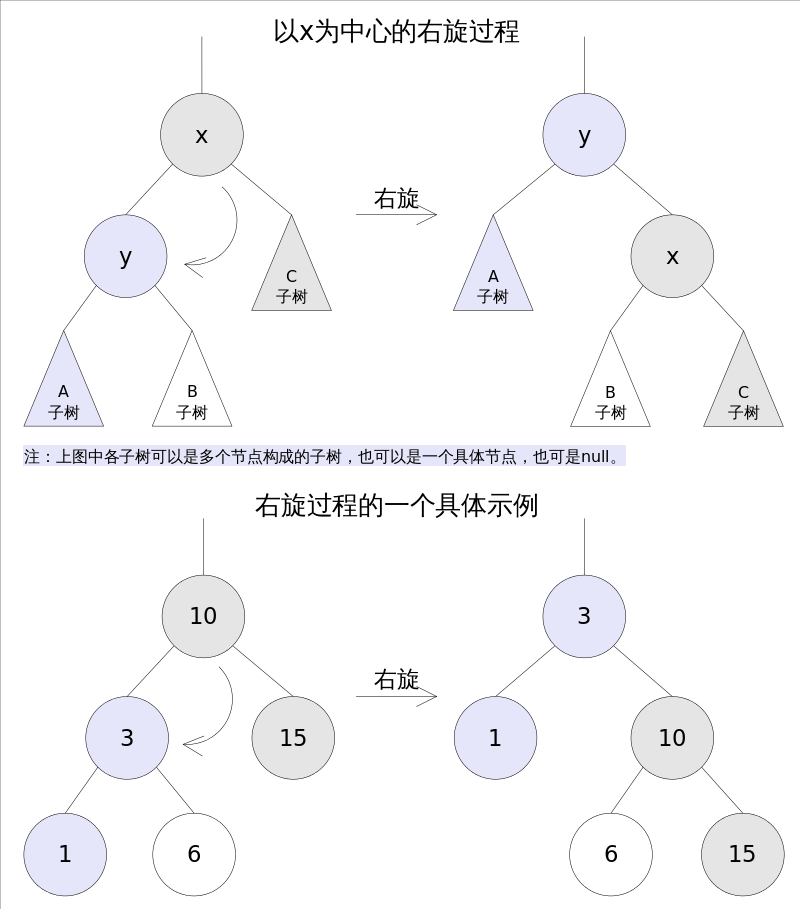
}

TreeMap底层以一颗红黑树实现。key不能为空，非线程安全。以key对应的Comparator排序。





**左旋**的过程是将x的右子树绕x逆时针旋转，使得x的右子树成为x的父亲，同时修改相关节点的引用。旋转之后，二叉查找树的属性仍然满足。



**右旋**的过程是将x的左子树绕x顺时针旋转，使得x的左子树成为x的父亲，同时修改相关节点的引用。旋转之后，二叉查找树的属性仍然满足。

**put过程：**

1.如果root为空，则直接new一个Entry，然后返回null；

2.如果root不为空，且如果Comparator不为空，执行步骤4；

3.如果Comparator为空，将key对应的Comparator赋值给当前TreeMap的Comparator；

4.遍历红黑树，比较key与当前节点key的大小，找到parent。如果相等，则用新的value替换掉旧的value,然后返回旧的Entry；

5.new一个新的Entry，根据key比较的结果，将其放在左或右；

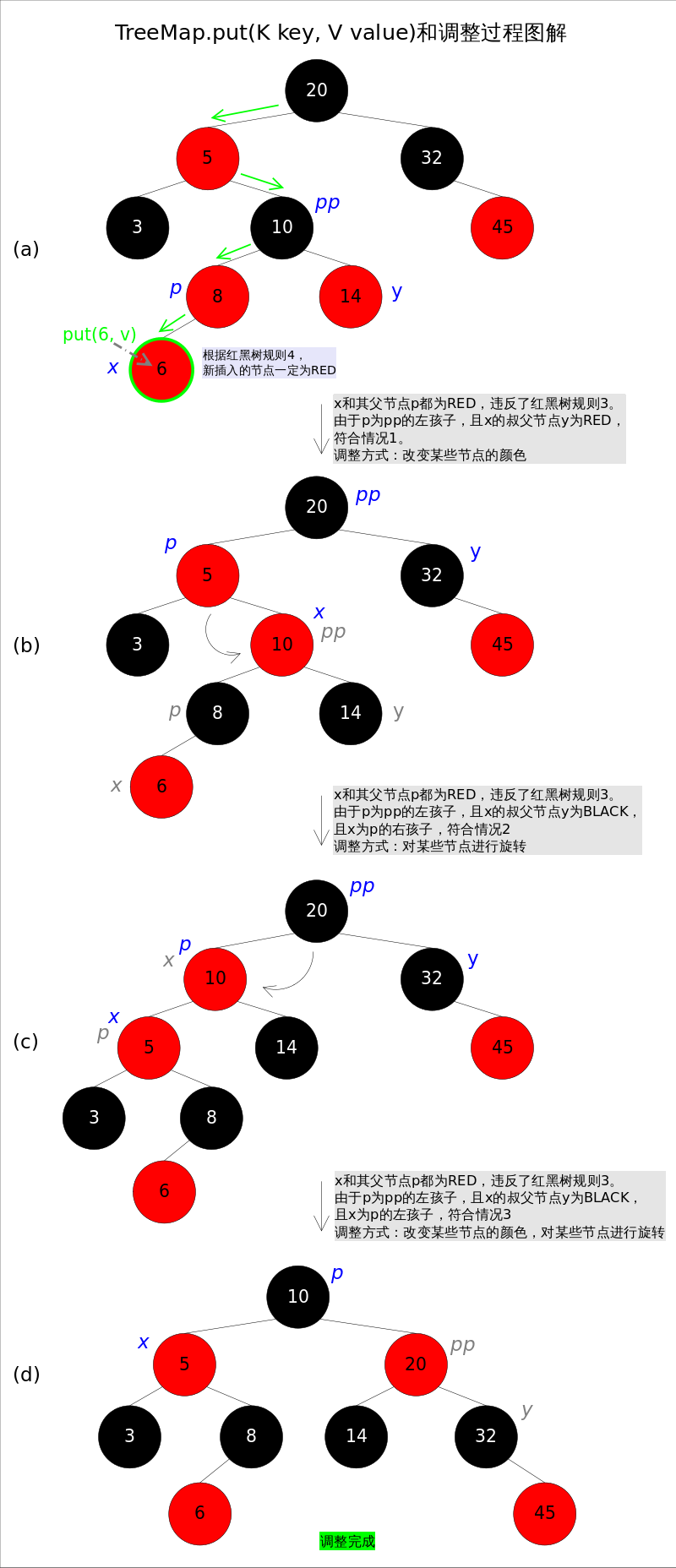
6.调整树

如果Entry不是root节点，且parent为red，则一直调整。

7.返回null。

**get过程：**

比较key，找到对应的节点并返回。



### 12、LinkedHashMap

**public** **class** LinkedHashMap<K, V> **extends** HashMap<K, V> **implements** Map<K, V> {

**static** **class** Entry<K, V> **extends** HashMap.Node<K, V> {

Entry<K, V> before, after;

Entry(**int** hash, K key, V value, Node<K, V> next) {

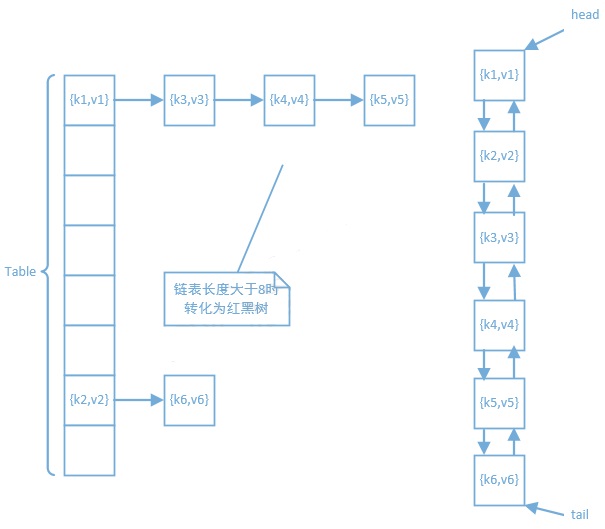
**super**(hash, key, value, next);

}

}

......

}



LinkedHashMap是HashMap的直接子类，二者唯一的区别是LinkedHashMap在HashMap的基础上，采用双向链表的形式将所有entry连接起来。

**利用LinkedHashMap实现LRU算法缓存：**

**public** **class** LRUCache **extends** LinkedHashMap {

**private** **int** maxElements;

**public** LRUCache(**int** maxSize) {

**super**(maxSize, 0.75F, **true**);

maxElements = maxSize;

}

**protected** **boolean** removeEldestEntry(Map.Entry eldest) {

**return** size() > maxElements;

}

}

LRU即Least Recently Used，最近最少使用。当缓存满了，会优先淘汰那些最近最不常访问的数据。LinkedList提供了一个boolean值accessOrder（true，所有的Entry按照访问的顺序排列）可以让用户指定是否实现LRU。每次访问都把访问的那个数据移到双向队列的尾部去。

### 13、HashTable

**public** **class** Hashtable<K, V> **extends** Dictionary<K, V> **implements** Map<K, V>, Cloneable, java.io.Serializable {

**private** **transient** Entry<?, ?>[] table;

**private** **transient** **int** count;

**private** **int** threshold;

**private** **float** loadFactor;

**private** **transient** **int** modCount = 0;

**public** Hashtable() {

**this**(11, 0.75f);

}

**private** **static** **final** **int** ***MAX\_ARRAY\_SIZE*** = Integer.***MAX\_VALUE*** - 8;

**protected** **void** rehash() {

**int** oldCapacity = table.length;

Entry<?, ?>[] oldMap = table;

**int** newCapacity = (oldCapacity << 1) + 1;

**if** (newCapacity - ***MAX\_ARRAY\_SIZE*** > 0) {

**if** (oldCapacity == ***MAX\_ARRAY\_SIZE***)

**return**;

newCapacity = ***MAX\_ARRAY\_SIZE***;

}

Entry<?, ?>[] newMap = **new** Entry<?, ?>[newCapacity];

modCount++;

threshold = (**int**) Math.*min*(newCapacity \* loadFactor, ***MAX\_ARRAY\_SIZE*** + 1);

table = newMap;

**for** (**int** i = oldCapacity; i-- > 0;) {

**for** (Entry<K, V> old = (Entry<K, V>) oldMap[i]; old != **null**;) {

Entry<K, V> e = old;

old = old.next;

**int** index = (e.hash & 0x7FFFFFFF) % newCapacity;

e.next = (Entry<K, V>) newMap[index];

newMap[index] = e;

}

}

}

......

}

**特征：**

1.key、value不能为null；

2.初始容量为11，加载因子0.75，每次扩容为（2n+1）；

3.线程安全的，大部分方法有**synchronized**修饰。

**hash值获取：**

**int** hash = key.hashCode();

**数组下标获取：**

**int** index = (hash & 0x7FFFFFFF) % tab.length;

“hash & 0x7FFFFFFF”是为了当hash为负数时，转化为正数。

### 14、WeakHashMap

**public** **class** WeakHashMap<K, V> **extends** AbstractMap<K, V> **implements** Map<K, V> {

**private** **static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 16;

**private** **static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30;

**private** **static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;

Entry<K, V>[] table;

**private** **int** threshold;

**private** **final** **float** loadFactor;

**private** **final** ReferenceQueue<Object> queue = **new** ReferenceQueue<>();

**private** **static** **class** Entry<K, V> **extends** WeakReference<Object> **implements** Map.Entry<K, V> {

V value;

**final** **int** hash;

Entry<K, V> next;

Entry(Object key, V value, ReferenceQueue<Object> queue,

**int** hash, Entry<K,V> next) {

**super**(key, queue);

**this**.value = value;

**this**.hash = hash;

**this**.next = next;

}

......

}

......

}

**private** **void** expungeStaleEntries() {

**for** (Object x; (x = queue.poll()) != **null**; ) {

**synchronized** (queue) {

@SuppressWarnings("unchecked")

Entry<K,V> e = (Entry<K,V>) x;

**int** i = indexFor(e.hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> p = prev;

**while** (p != **null**) {

Entry<K,V> next = p.next;

**if** (p == e) {

**if** (prev == e)

table[i] = next;

**else**

prev.next = next;

e.value = **null**;

size--;

**break**;

}

prev = p;

p = next;

}

}

}

}

**private** Entry<K,V>[] getTable() {

expungeStaleEntries();

**return** table;

}

内部是通过弱引用来管理entry的，key为弱引用，value为强引用。当除了自身有对key的引用外，此key没有其他引用，那么key在GC 的时候会被清除，value 在 key 清除后访问 WeakHashMap 时被清除。

### 15、EnumMap

**public** **class** EnumMap<K **extends** Enum<K>, V> **extends** AbstractMap<K, V>

**implements** java.io.Serializable, Cloneable

{

**private** **transient** K[] keyUniverse;

**private** **transient** Object[] vals;

**public** EnumMap(Class<K> keyType) {

**this**.keyType = keyType;

keyUniverse = getKeyUniverse(keyType);

vals = **new** Object[keyUniverse.length];

}

}

一种键为枚举类型的特殊的Map实现。所有的Key也必须是一种枚举类型。底层采用两个数组来实现的，一个存储所有的key，一个存储value；数组大小为枚举类型数量的大小，数组的值为Value。

**put、get方法：**

put方法通过key的ordinal将值存储到对应的地方，get方法则根据key的ordinal获取对应的值。

### 16、IdentityHashMap

**public** **class** IdentityHashMap<K, V> **extends** AbstractMap<K, V> **implements** Map<K, V>, java.io.Serializable, Cloneable {

**private** **static** **final** **int** ***DEFAULT\_CAPACITY*** = 32;

**private** **static** **final** **int** ***MINIMUM\_CAPACITY*** = 4;

**private** **static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 29;

**transient** Object[] table;

**int** size;

**transient** **int** modCount;

**static** **final** Object ***NULL\_KEY*** = **new** Object();

**private** **static** **int** capacity(**int** expectedMaxSize) {

**return** (expectedMaxSize > ***MAXIMUM\_CAPACITY*** / 3) ? ***MAXIMUM\_CAPACITY***

: (expectedMaxSize <= 2 \* ***MINIMUM\_CAPACITY*** / 3) ? ***MINIMUM\_CAPACITY***

: Integer.*highestOneBit*(expectedMaxSize + (expectedMaxSize << 1));

}

**private** **void** init(**int** initCapacity) {

table = **new** Object[2 \* initCapacity];

}

......

}



特征：

1.允许key值相同，key通过“==”比较；

2.底层用一个Object[]实现，数组的前一个元素存key，紧挨着的下一个存value；

3.默认容量为64，所有的容量都要保证是2的幂次；填充因子为2/3，每次扩容为原来的2倍。

说明：NULL\_KEY的存在是为了解决key为null的情况，因为如果为null，key还是要存一个值，否则会破坏数组的存储规则。

### 17、Properties

**public** **class** Properties **extends** Hashtable<Object, Object> {

**protected** Properties defaults;

**public** Properties() {

**this**(**null**);

}

**public** Properties(Properties defaults) {

**this**.defaults = defaults;

}

......

}

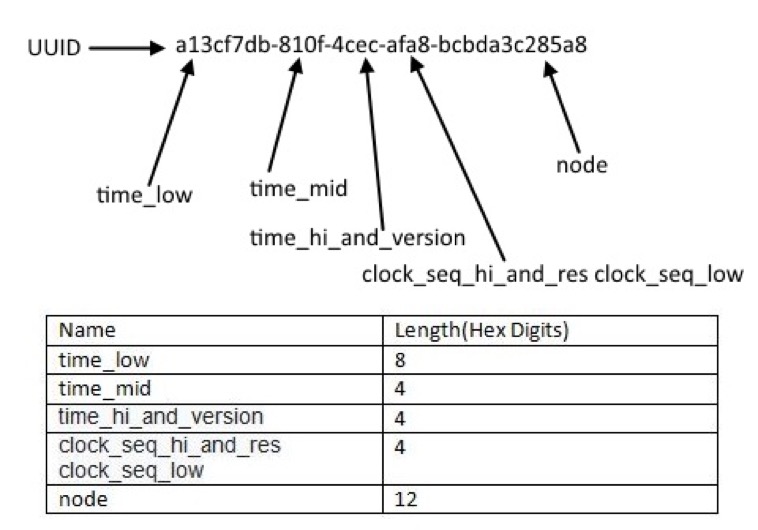
Properties支持文本方式和xml方式的数据存储。在文本方式中，格式为key:value,其中分隔符可以是：冒号(:)、等号(=)、空格。其中空格可以作为key的结束，同时获取的值回将分割符号两端的空格去掉。

Properties只支持1对1模式的属性设置，而且不支持多层多级属性设置。

### 18、UUID

**public** **final** **class** UUID **implements** java.io.Serializable, Comparable<UUID>

UUID是由一组32位数的16进制数字所构成，以连字号分为五段，形式为8-4-4-4-12的32个字符。



## （四）java.lang

### 1、Short

**public** **final** **class** Short **extends** Number **implements** Comparable<Short> {

**public** **static** **final** **short** ***MIN\_VALUE*** = -32768;

**public** **static** **final** **short** ***MAX\_VALUE*** = 32767;

**public** **static** **final** Class<Short> ***TYPE*** = (Class<Short>) Class.*getPrimitiveClass*("short");

**private** **final** **short** value;

**public** **static** **final** **int** ***SIZE*** = 16;

**public** **static** **final** **int** ***BYTES*** = ***SIZE*** / Byte.***SIZE***;

**private** **static** **class** ShortCache {

**private** ShortCache() {

}

**static** **final** Short ***cache***[] = **new** Short[-(-128) + 127 + 1];

**static** {

**for** (**int** i = 0; i < ***cache***.length; i++)

***cache***[i] = **new** Short((**short**) (i - 128));

}

}

**public** **static** Short valueOf(**short** s) {

**final** **int** offset = 128;

**int** sAsInt = s;

**if** (sAsInt >= -128 && sAsInt <= 127) {

**return** ShortCache.***cache***[sAsInt + offset];

}

**return** **new** Short(s);

}

......

}

2字节长度，16位。方法实现基于Integer。缓存了[-128, 127]之间的值。

### 2、Integer

**public** **final** **class** Integer **extends** Number **implements** Comparable<Integer> {

@Native **public** **static** **final** **int** ***MIN\_VALUE*** = 0x80000000;

@Native **public** **static** **final** **int** ***MAX\_VALUE*** = 0x7fffffff;

**public** **static** **final** Class<Integer> ***TYPE*** = (Class<Integer>) Class.*getPrimitiveClass*("int");

**final** **static** **char**[] ***digits*** = {

'0' , '1' , '2' , '3' , '4' , '5' ,

'6' , '7' , '8' , '9' , 'a' , 'b' ,

'c' , 'd' , 'e' , 'f' , 'g' , 'h' ,

'i' , 'j' , 'k' , 'l' , 'm' , 'n' ,

'o' , 'p' , 'q' , 'r' , 's' , 't' ,

'u' , 'v' , 'w' , 'x' , 'y' , 'z'

};

**private** **final** **int** value;

**private** **static** **class** IntegerCache {

**static** **final** **int** ***low*** = -128;

**static** **final** **int** ***high***;

**static** **final** Integer ***cache***[];

**static** {

**int** h = 127;

String integerCacheHighPropValue = sun.misc.VM.*getSavedProperty*("java.lang.Integer.IntegerCache.high");

**if** (integerCacheHighPropValue != **null**) {

**try** {

**int** i = parseInt(integerCacheHighPropValue);

i = Math.*max*(i, 127);

h = Math.*min*(i, Integer.***MAX\_VALUE*** - (-***low***) - 1);

} **catch** (NumberFormatException nfe) {

}

}

***high*** = h;

***cache*** = **new** Integer[(***high*** - ***low***) + 1];

**int** j = ***low***;

**for** (**int** k = 0; k < ***cache***.length; k++)

***cache***[k] = **new** Integer(j++);

**assert** IntegerCache.***high*** >= 127;

}

**private** IntegerCache() {

}

}

**public** **static** Integer valueOf(**int** i) {

**if** (i >= IntegerCache.***low*** && i <= IntegerCache.***high***)

**return** IntegerCache.***cache***[i + (-IntegerCache.***low***)];

**return** **new** Integer(i);

}

......

}

4字节长度，32位。缓存了[-128, high]之间的值，high默认127，可配置（不小于127）。

### 3、Double

**public** **final** **class** Double **extends** Number **implements** Comparable<Double> {

**public** **static** **final** **int** ***SIZE*** = 64;

**public** **static** **final** **int** ***BYTES*** = ***SIZE*** / Byte.***SIZE***;

**public** **static** **final** Class<Double> ***TYPE*** = (Class<Double>) Class.*getPrimitiveClass*("double");

**private** **final** **double** value;

......

}

8字节长度，64位。

### 4、Float

**public** **final** **class** Float **extends** Number **implements** Comparable<Float> {

**public** **static** **final** **int** ***SIZE*** = 32;

**public** **static** **final** **int** ***BYTES*** = ***SIZE*** / Byte.***SIZE***;

**public** **static** **final** Class<Float> ***TYPE*** = (Class<Float>) Class.*getPrimitiveClass*("float");

**private** **final** **float** value;

......

}

4字节长度，32位。

### 5、Long

**public** **final** **class** Long **extends** Number **implements** Comparable<Long> {

**private** **final** **long** value;

@Native

**public** **static** **final** **int** ***SIZE*** = 64;

**public** **static** **final** **int** ***BYTES*** = ***SIZE*** / Byte.***SIZE***;

**public** **static** **final** Class<Long> ***TYPE*** = (Class<Long>) Class.*getPrimitiveClass*("long");

**private** **static** **class** LongCache {

**private** LongCache() {

}

**static** **final** Long ***cache***[] = **new** Long[-(-128) + 127 + 1];

**static** {

**for** (**int** i = 0; i < ***cache***.length; i++)

***cache***[i] = **new** Long(i - 128);

}

}

**public** **static** Long valueOf(**long** l) {

**final** **int** offset = 128;

**if** (l >= -128 && l <= 127) {

**return** LongCache.***cache***[(**int**) l + offset];

}

**return** **new** Long(l);

}

......

}

8字节长度，64位。缓存了[-128, 127]之间的值。

### 6、Character

**public** **final** **class** Character **implements** java.io.Serializable, Comparable<Character> {

**private** **final** **char** value;

**public** **static** **final** **int** ***SIZE*** = 16;

**public** **static** **final** **int** ***BYTES*** = ***SIZE*** / Byte.***SIZE***;

**public** **static** **final** Class<Character> ***TYPE*** = (Class<Character>) Class.*getPrimitiveClass*("char");

**private** **static** **class** CharacterCache {

**private** CharacterCache() {

}

**static** **final** Character ***cache***[] = **new** Character[127 + 1];

**static** {

**for** (**int** i = 0; i < ***cache***.length; i++)

***cache***[i] = **new** Character((**char**) i);

}

}

**public** **static** Character valueOf(**char** c) {

**if** (c <= 127) {

**return** CharacterCache.***cache***[(**int**)c];

}

**return** **new** Character(c);

}

......

}

2字节长度，16位。缓存了ASCII码表中的字符。

### 7、Boolean

**public** **final** **class** Boolean **implements** java.io.Serializable, Comparable<Boolean> {

**public** **static** **final** Boolean ***TRUE*** = **new** Boolean(**true**);

**public** **static** **final** Boolean ***FALSE*** = **new** Boolean(**false**);

**public** **static** **final** Class<Boolean> ***TYPE*** = (Class<Boolean>) Class.*getPrimitiveClass*("boolean");

**private** **final** **boolean** value;

......

}

### 8、Byte

**public** **final** **class** Byte **extends** Number **implements** Comparable<Byte> {

**public** **static** **final** **byte** ***MIN\_VALUE*** = -128;

**public** **static** **final** **byte** ***MAX\_VALUE*** = 127;

**public** **static** **final** **int** ***SIZE*** = 8;

**public** **static** **final** **int** ***BYTES*** = ***SIZE*** / Byte.***SIZE***;

**private** **final** **byte** value;

**public** **static** **final** Class<Byte> ***TYPE*** = (Class<Byte>) Class.*getPrimitiveClass*("byte");

**private** **static** **class** ByteCache {

**private** ByteCache(){}

**static** **final** Byte ***cache***[] = **new** Byte[-(-128) + 127 + 1];

**static** {

**for**(**int** i = 0; i < ***cache***.length; i++)

***cache***[i] = **new** Byte((**byte**)(i - 128));

}

}

**public** **static** Byte valueOf(**byte** b) {

**final** **int** offset = 128;

**return** ByteCache.***cache***[(**int**)b + offset];

}

......

}

1字节长度，8位。缓存了所有的值，[-128, 127]之间对应的byte。

### 9、String

**public** **final** **class** String **implements** java.io.Serializable, Comparable<String>, CharSequence {

**private** **final** **char** value[];

**public** String concat(String str) {

**int** otherLen = str.length();

**if** (otherLen == 0) {

**return** **this**;

}

**int** len = value.length;

**char** buf[] = Arrays.copyOf(value, len + otherLen);

str.getChars(buf, len);

**return** **new** String(buf, **true**);

}

......

}

**特征：**

1.底层基于char[ ]实现，且被**final**修饰。

2.用“+”连接字符串，实质上是new了一个StringBuilder，然后通过append方法连接，最后toString。

### 10、StringBuffer

**public** **final** **class** StringBuffer **extends** AbstractStringBuilder **implements** java.io.Serializable, CharSequence {

**private** **transient** **char**[] toStringCache;

**public** StringBuffer() {

**super**(16);

}

@Override

**public** **synchronized** StringBuffer append(String str) {

toStringCache = **null**;

**super**.append(str);

**return** **this**;

}

@Override

**public** **synchronized** String toString() {

**if** (toStringCache == **null**) {

toStringCache = Arrays.*copyOfRange*(value, 0, count);

}

**return** **new** String(toStringCache, **true**);

}

......

}

**abstract** **class** AbstractStringBuilder **implements** Appendable, CharSequence {

**char**[] value;

**int** count;

**void** expandCapacity(**int** minimumCapacity) {

**int** newCapacity = value.length \* 2 + 2;

**if** (newCapacity - minimumCapacity < 0)

newCapacity = minimumCapacity;

**if** (newCapacity < 0) {

**if** (minimumCapacity < 0)

**throw** **new** OutOfMemoryError();

newCapacity = Integer.***MAX\_VALUE***;

}

value = Arrays.*copyOf*(value, newCapacity);

}

**public** AbstractStringBuilder append(String str) {

**if** (str == **null**)

**return** appendNull();

**int** len = str.length();

ensureCapacityInternal(count + len);

str.getChars(0, len, value, count);

count += len;

**return** **this**;

}

......

}

**特征：**

1.底层基于char[ ]实现，没有**final**修饰，初始容量为16，每次扩容为原来的2n+2。

2.线程安全的，大多数方法有**synchronized**修饰。

3.toString()方法缓存了char[ ]。

### 11、StringBuilder

**public** **final** **class** StringBuilder **extends** AbstractStringBuilder **implements** java.io.Serializable, CharSequence {

**public** StringBuilder() {

**super**(16);

}

@Override

**public** StringBuilder append(String str) {

**super**.append(str);

**return** **this**;

}

@Override

**public** String toString() {

// Create a copy, don't share the array

**return** **new** String(value, 0, count);

}

......

}

**特征：**

1.底层基于char[ ]实现，初始容量为16，每次扩容为原来的2n+2。

2.是线程不安全的。

3.toString()方法每次重新new了一个char[ ]。

### 12、Thread

**特征：**

1.底层

### 13、ThreadLocal

内存泄漏

ThreadLocal用于创建线程的本地变量

## （五）java.math

### 1、BigInteger

### 2、BigDecimal

## （六）HashMap、HashTable区别

### 1、继承的父类不同

**public** **class** HashMap<K,V> **extends** AbstractMap<K,V>

**implements** Map<K,V>, Cloneable, Serializable

**public** **class** Hashtable<K,V> **extends** Dictionary<K,V>

**implements** Map<K,V>, Cloneable, java.io.Serializable

### 2、线程安全性不同

Hashtable的大多数方法中都加入了synchronized

### 3、是否提供contains方法

1）HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey，因为contains方法容易让人引起误解。

2）Hashtable则保留了contains，containsValue和containsKey三个方法，其中contains和containsValue功能相同。

**public** **boolean** containsValue(Object value) {

**return** contains(value);

}

### 4、key和value是否允许null值

1）Hashtable中，key和value都不允许出现null值。

2）HashMap中不能由get()方法来判断HashMap中是否存在某个键，而应该用containsKey()方法来判断。

### 5、hash值不同

HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

### 6、内部实现使用的数组初始化和扩容方式不同

1）HashTable在不指定容量的情况下的默认容量为11，而HashMap为16；Hashtable不要求底层数组的容量一定要为2的整数次幂，而HashMap则要求一定为2的整数次幂。

2）Hashtable扩容时，将容量变为原来的2倍加1，而HashMap扩容时，将容量变为原来的2倍。

## （七）AtomicInteger的实现原理？CAS机制？CAS的缺点？

在Java中通过锁和循环CAS的方式实现原子操作。

### 1、AtomicInteger实现原理

### 2、CAS机制

CAS（Compare and Swap即比较并交换），设计并发算法时常用到的一种技术，java.util.concurrent.atomic包下的原子操作类都是基于CAS实现的。

CAS有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做并返回false。

CAS操作是基于处理器的CMPXCHG汇编指令实现的，因此，JVM中的CAS的原子性是处理器保障的。

### 3、CAS缺点

CAS存在三个问题：

1．ABA问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值；如果在这段期间它的值曾经被改成了B，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个漏洞称为CAS操作的”ABA”问题。

2．循环时间长开销大

3．只能保证一个共享变量的原子操作

解决方案：AtomicMarkableReference、AtomicStampedReference

AtomicMarkableReference是利用一个boolean标记记录版本是否更改。

AtomicStampedReference是利用版本戳的形式记录了每次改变以后的版本号。

## （八）Object类中的方法？每个方法的作用？

## （九）Collections类中的方法？每个方法的作用？

Collections.sort()和Arrays.sort()的实现？

## （十）ThreadLocalRandom

## （十一）System.arraycopy(src, srcPos, dest, destPos, length);

## （十二）System.out.println

## （十三）枚举是怎么保证线程安全的？怎么保证只有一个实例？

编译javac

反编译javap

**public** **final** **class** T **extends** Enum

## （十四）反射机制的实现原理？

## （十五）如何用Java分配一段连续的1G的内存空间？

sun.misc.Unsafe.allcateMemory

## （十六）java7与java8的区别

# 第七章 框架

## （一）Spring

### 1、spring的实例化过程？

### 2、Ioc原理？

### 3、Bean工厂实现？

### 4、AOP实现原理？

### 5、过滤器、拦截器的实现原理？

### 6、事务原理、类型、场景、区别、处理机制？

### 7、Bean生命周期？

### 8、SpringMVC请求流程？

### 9、Spring MVC九大组件？

### 10、@Controller、@RequestMapping、@requestbody底层原理

### 11、task的实现原理？

## （二）Spring session

## （三）Spring security

## （四）Spring boot

## （五）Spring cloud

## （六）rabbitmq

## （七）Mybatis

### 1、mybatis加载过程？

### 2、原理？

### 3、缓存实用场景及选择策略？二级缓存？

### 4、Mybatis事务

### 5、Mybatis动态代理的实现原理

## （八）apache mina

## （九）Netty

## （十）hive

## （十一）hbase

## （十二）hadoop

# 第八章 数据库

## （一）数据库的四大特征

### 1、原子性（Atomicity）

原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。

### 2、一致性（Consistency）

一个事务执行之前和执行之后都必须处于一致性状态。

### 3、隔离性（Isolation）

隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

### 4、持久性（Durability）

持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的。

记：一致（原子性）持久（持久性）隔离（隔离性）原子（原子性）

## （二）mysql存储引擎类型？各有优缺点、区别？

## （三）mysql索引类型？对应的数据结构？优缺点？单个索引、联合索引、主键索引？

## （四）常用SQL（select、insert、update、delete、truncate ）优化方式

## （五）SQL什么情况下不会使用索引（不包含，不等于，函数）

## （六）分库？分表？分库分表后的联合查询？分库分表后的分页查询？分表之后想让一个id多个表是自增的，效率实现?

## （七）主从实时备份同步的配置，以及原理(从库读主库的binlog)，读写分离？

## （八）JDBC底层原理

# 第九章 数据结构与算法

## （一）数组、链表、栈、队列、树（二叉树、查找树、AVL树、红黑树、B树）

### 1、二叉树

### 2、查找树

### 3、AVL树

### 4、红黑树

1.根结点和叶结点是黑色的；

2.红色结点的子结点必是黑色的；

3.任何一个结点到其所有后代叶结点的简单路径包含相同数目的黑色结点，并称这个黑色结点的数目（不包含出发结点）为黑高（black-height，用bh(x)表示，红黑树的黑高为根结点的黑高）。

### 5、B树

## （二）最短路径算法（Dijkstra、Floyd）

## （三）深度优先搜索、广度优先搜索

## （四）字符串模式匹配算法（KMP算法）

## （五）大数据处理、排序、查找

## （六）hash算法？hash冲突？冲突解决办法？如何优化？

通过hash计算key，得到的结果作为地址去存储时，发现已经被使用了。

https://www.cnblogs.com/novalist/p/6396410.html

### 1、开放定址法

### 2、链地址法

将所有哈希地址相同的记录都链接在同一个链表中。

### 3、再哈希

产生冲突时计算另一个哈希函数地址，直到冲突不发生为止。

### 4、建立公共溢出区

可以用红黑树提高查找效率

## （七）一致性hash

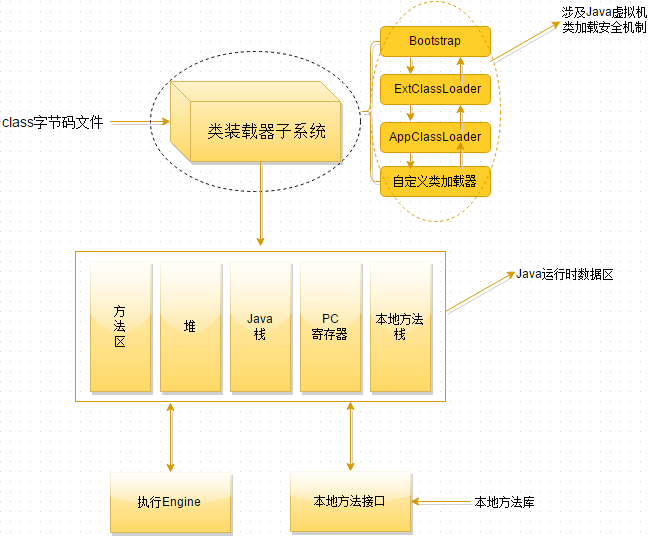
## （八）刷题网站

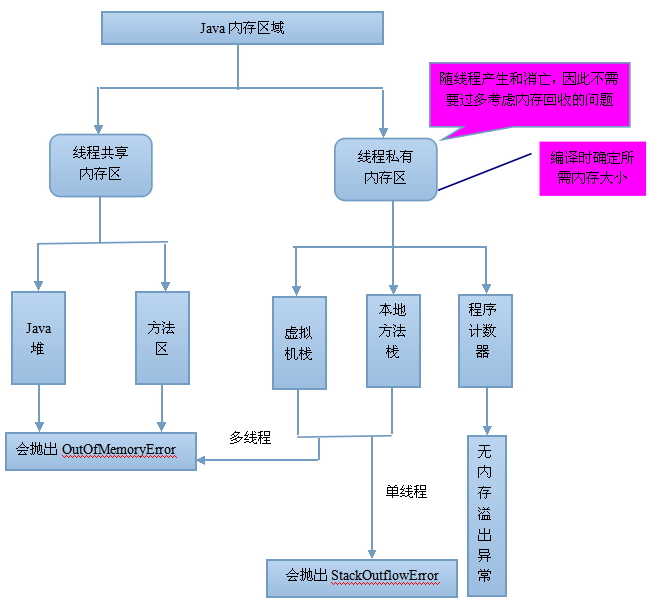
<https://leetcode.com/problemset/all/>

1. 查找第N大的数。
2. 将字符串按出现频率高低输出显示，频率相同的按自然排序显示。
3. 实现LRU缓存算法

# 第十章 Java虚拟机

## （一）jvm内存布局？Class文件结构



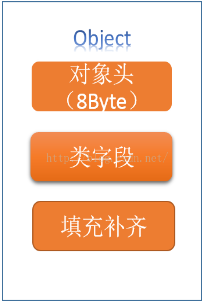


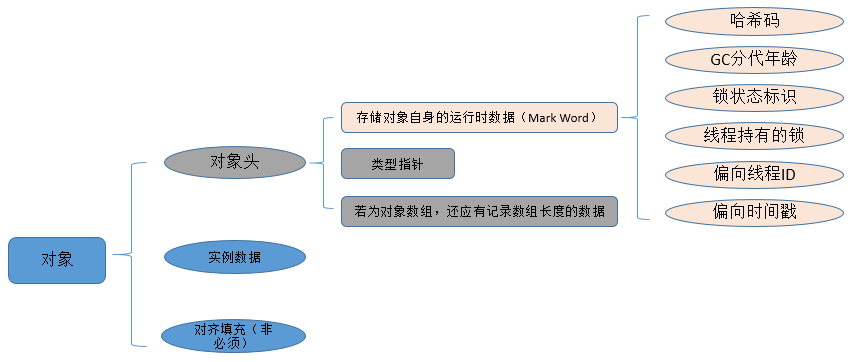
栈和堆的区别？

## （二）Java对象内存模型？对象的创建？

### 1、java对象内存模型

对象在内存的分布分为3个部分：对象头，实例数据，和对齐填充。



**对齐填充：**JVM要求对象的大小必须是8的整数倍；否则需要补位对齐。

2、对象的创建

## （三）六种存储地址

### 1、解释

**寄存器(register)**：

位于处理器内部。是最快的存储区，不能直接控制。

**堆栈(stack)**：

位于通用RAM中，也就是我们常说的内存。但通过它的“堆栈指针”可以从处理器哪里获得支持。堆栈指针若向下移动，则分配新的内存；若向上移动，则释放那些内存。这是一种快速有效的分配存储方法，仅次于寄存器。创建程序时候，JAVA编译器必须知道存储在堆栈内所有数据的确切大小和生命周期，因为它必须生成相应的代码，以便上下移动堆栈指针。这一约束限制了程序的灵活性，所以虽然某些JAVA数据存储在堆栈中——特别是对象引用，但是JAVA对象不存储其中。

**堆(heap)**：

一种通用性的内存池（也存在于RAM中），用于存放所有的JAVA对象。“堆”同时也是一个基本的数据结构，它是一种特殊的完全二叉树。堆不同于堆栈的好处是：编译器不需要知道要从堆里分配多少存储区域，也不必知道存储的数据在堆里存活多长时间。用堆进行存储分配比用堆栈进行存储存储需要更多的时间。

**静态存储(static storage)**：

静态存储里存放程序运行时一直存在的数据。可用关键字static来标识一个对象的特定元素是静态的，但JAVA对象本身从来不会存放在静态存储空间里。使用String类的intern()方法将字符串重新放置在静态存储中。

**常量存储(constant storage)**：

常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会被改变。有时，在嵌入式系统中，常量本身会和其他部分分割离开，所以在这种情况下，可以选择将其放在ROM中。

**非RAM存储**：

如果数据完全存活于程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在。

### 2、存储的内容

堆栈：基本类型的局部变量、对象的引用

堆：new的对象本身

静态存储（方法区）：static修饰的变量、字符串（String str = "world";）

## （四）java内存管理

## （五）GC

### 1、机制（垃圾回收机制）？

### 2、什么时候会触发YGC，什么时候触发FGC？

### 3、强引用、软引用、弱引用、虚引用？

### 4、垃圾对象的判定？

### 5、垃圾收集算法？

### 6、垃圾收集器？

http://blog.csdn.net/ns\_code/article/details/18076173

## （六）调优

### 1、如何查看JVM的内存使用情况？

### 2、JVM有哪些常用启动参数可以调整，描述几个？

### 3、常用的JVM配置和调优参数？

## （七）类的加载机制？双亲委派模型？

### 1、类加载（三种）

1）命令行启动应用时候由JVM初始化加载

2）通过Class.forName()方法动态加载

Class.*forName*(className);

3）通过ClassLoader.loadClass()方法动态加载

ClassLoader.*getSystemClassLoader*().loadClass(name);

### 2、类的加载机制

1）通过一个类的全限定名来获取其定义的二进制字节流。

2）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。

3）在Java堆中生成一个代表这个类的java.lang.Class对象，作为对方法区中这些数据的访问入口。

### 3、双亲委派模型

## （八）概念

java并发采用的是共享内存模型。

### 1、内存屏障

内存屏障，又称内存栅栏，是一个CPU指令，作用：

1．保证特定操作的执行顺序。

2．影响某些数据（或则是某条指令的执行结果）的内存可见性。

编译器和CPU能够重排序指令，在保证最终结果相同的前提下，尝试优化性能。插入一条Memory Barrier会告诉编译器和CPU：不管什么指令都不能和这条Memory Barrier指令重排序。

Memory Barrier所做的另外一件事是强制刷出各种CPU cache，如一个 Write-Barrier（写入屏障）将刷出所有在 Barrier 之前写入 cache 的数据，因此，任何CPU上的线程都能读取到这些数据的最新版本。

### 2、happens-before规则

在JMM中，如果一个操作的执行结果需要对另一个操作可见，那么这两个操作之间必须要存在happens-before关系，这个的两个操作既可以在同一个线程，也可以在不同的两个线程中。

happens-before规则如下：

1．程序顺序规则：一个线程中的每个操作，happens-before于该线程中任意的后续操作。

2．监视器锁规则：对一个锁的解锁操作，happens-before于随后对这个锁的加锁操作。

3．volatile域规则：对一个volatile域的写操作，happens-before于任意线程后续对这个volatile域的读。

4．传递性规则：如果 A happens-before B，且 B happens-before C，那么A happens-before C。

注意：两个操作之间具有happens-before关系，并不意味前一个操作必须要在后一个操作之前执行！仅仅要求前一个操作的执行结果，对于后一个操作是可见的，且前一个操作按顺序排在后一个操作之前。

### 3、指令重排序

在执行程序时，为了提高性能，编译器和处理器会对指令做重排序。但是，JMM确保在不同的编译器和不同的处理器平台之上，通过插入特定类型的Memory Barrier来禁止特定类型的编译器重排序和处理器重排序，为上层提供一致的内存可见性保证。

1．编译器优化重排序：编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

2．指令级并行的重排序：如果不存l在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

3．内存系统的重排序：处理器使用缓存和读写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

### 4、as-if-serial

不管怎么重排序，单线程下的执行结果不能被改变，编译器、runtime和处理器都必须遵守as-if-serial语义。

# 第十一章 JDK工具

## jvisualvm

VisualVM 是一款免费的性能分析工具。

### 1、线程Dump

**线程死锁代码：**

**package** com.hyn.web;

**import** java.util.concurrent.TimeUnit;

**public** **class** ThreadDeadLcokTest {

**static** **final** Object ***LOCKA*** = **new** Object();

**static** **final** Object ***LOCKB*** = **new** Object();

**public** **static** **void** main(String[] args) {

**final** Object[][] locks = **new** Object[][] { { ***LOCKA***, ***LOCKB*** }, { ***LOCKB***, ***LOCKA*** } };

**for** (**int** i = 0; i < locks.length; i++) {

**final** **int** index = i;

**new** Thread(**new** Runnable() {

@Override

**public** **void** run() {

*doSomething*(locks[index][0], locks[index][1]);

}

}).start();

}

}

**public** **static** **void** doSomething(Object lock1, Object lock2) {

**synchronized** (lock1) {

**try** {

TimeUnit.***MILLISECONDS***.sleep(500);

} **catch** (InterruptedException e) {}

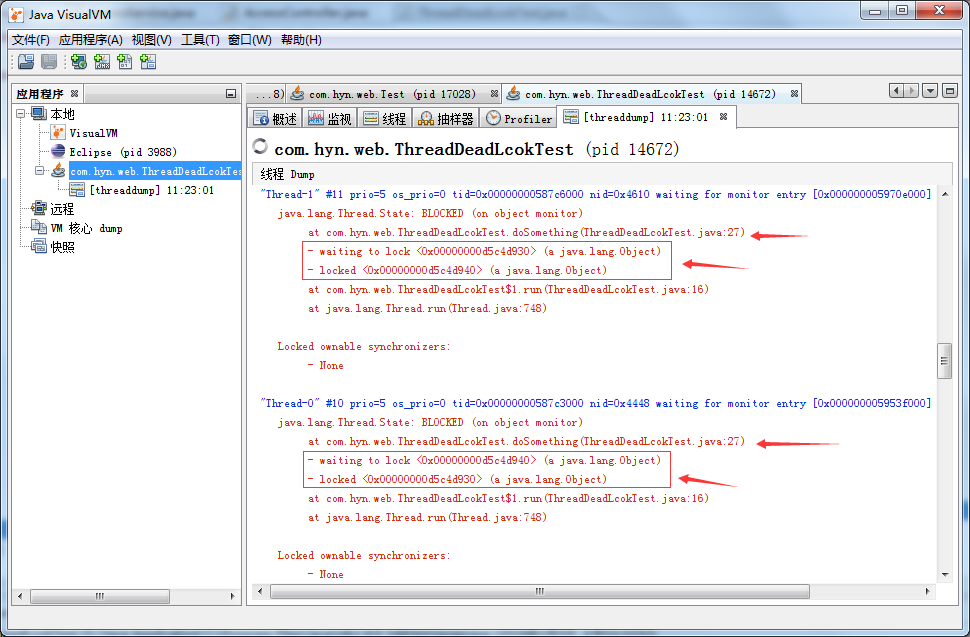
**synchronized** (lock2) {}

}

}

}

**对应的线程dump:**



### 2、堆Dump

（二）

（三）

（四）

（五）

（六）

（七）

# 第十二章 Web相关

## （一）session、cookie

### 1、区别和联系？

### 2、session实现原理？

### 3、session的生命周期？

### 4、分布式session的几种实现方法？

## （二）websocket、http

### 1、原理？区别？

### 2、如何用io、nio实现websocket

## （三）servlet

### 1、实现方式

**class** ServletFirst **implements** Servlet

**class** ServletSecond **extends** GenericServlet

**class** ServletThird **extends** HttpServlet

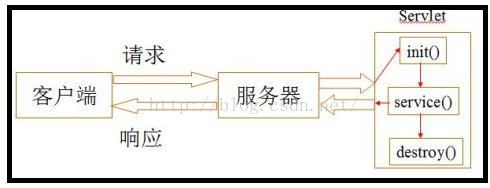
### 2、生命周期

Servlet是单实例多线程的。在第一次被调用时，会被实例化。

1.且仅在初次会调用init()；

2.调用service方法；在service方法中判断调用对应的doGet、doPost、doPut等，返回数据；

3.当容器重启、关闭时，会调用destroy方法。



**解决同步问题的方案**

1.去除实例变量，使用局部变量；

2.使用同步代码块、锁

3.实现SingleThreadModel接口

在同一时刻一个servlet实例的service方法只会被一个线程执行。

### 3、监听器原理

## （四）jsp处理过程

## （五）get、post区别

## （六）webservice

## （七）forward、重定向区别

## （八）https实现原理

## （九）TCP

### 1、TCP建立连接、断开连接的过程

### 2、TCP的滑动窗口协议

### 3、慢启动

### 4、七层模型

## （十）跨域访问

## （十一）Tomcat

### 1、源码分析

### 2、调优

# 第十三 linux

# 第十四章 项目相关

## （一）redis

### 1、redis的数据结构

### 2、实现原理

Redis为单进程单线程模式，采用队列模式将并发访问变为串行访问。Redis本身没有锁的概念。

1）redis是用“单线程-多路复用IO模型”来实现高性能的内存数据服务的。

2）数据在内存中操作

3）I/O多路复用技术通过把多个I/O阻塞复用到同一个select的阻塞上，从而使得系统在单线程的情况下可以同时处理多个客户端请求。

### 3、Redis事务

### 4、Redis的持久化机制是什么？各自的优缺点？

1）RDB

用数据集快照的方式记录redis数据库的所有键值对。

优点：

a.只有一个文件dump.rdb，方便持久化。

b.容灾性好，一个文件可以保存到安全的磁盘。

c.性能最大化，fork子进程来完成写操作，让主进程继续处理命令，所以是IO最大化。

d.相对于数据集大时，比AOF的启动效率更高。

缺点：

a.数据安全性低。

2）AOF

所有的命令行记录以redis命令请求协议的格式保存为aof文件。

优点：

a.数据安全，aof持久化可以配置appendfsync属性，有always，每进行一次命令操作就记录到aof文件中一次。

b.通过append模式写文件，即使中途服务器宕机，可以通过redis-check-aof工具解决数据一致性问题。

c .AOF机制的rewrite模式。

缺点：

a.文件会比RDB形式的文件大。

b.数据集大的时候，比rdb启动效率低。

### 5、缓存穿透？缓存击穿？缓存雪崩？

### 6、redis集群的理解，怎么动态增加或者删除一个节点，而保证数据不丢失。（一致性哈希问题）

## （二）支付宝支付、微信支付、银行卡支付流程？加密方式？

1、ConcurrentHashMap 高并发性的实现机制

https://www.ibm.com/developerworks/cn/java/java-lo-concurrenthashmap/

2、ConcurrentHashMap的Java内存模型

http://www.tengleitech.com/archives/1293

3、读写锁，分拆锁、分离锁，ThreadLocal，copyOnWrite

4、JAVA之Concurrent包书目录

https://zhuanlan.zhihu.com/p/25741785

5、java虚拟机

http://blog.csdn.net/column/details/java-vm.html

ConcurrentHashMap概念

ConcurrentHashMap允许多个修改操作并发的进行，关键在于使用了锁分离的技术。它使用了多个锁来控制 hash 表不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段

CAS（Compare and Swap, 翻译成比较并交换）

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。