JVM

JVM全名是java虚拟机，是可以运行java字节码的虚拟机。一般写好的java代码先会通过JDK中的javac编译为.class文件，也就是字节码文件，该文件可以被JVM运行，然后由JVM将字节码交给Java解释器，最后由解释器翻译为机器码，由解释器执行，得到结果。这样做的好处是相同的字节码文件（.class）通过JVM运行会得到相同的结果。这样就可以达到一次编译，随处运行的效果。

JDK和JRE

JRE是java的运行时环境，包含了JVM、Java库和一些其他构件。只有安装了jre才能运行Java程序。JRE无法创建新的程序。

- JDK包含了JRE的全部东西，同时还多了编译器（javac）和其他工具（javadoc和jdb）。JDK可以创建和编译程序。

Java语言编译与解释并存

因为java语言是先编译为.class文件然后再由JVM将字节码交给解释器来翻译为机器码。解释器是JVM的一部分。

Java和C++的区别

都是面向对象的语言；

- java没有指针来访问内存，这样内存更加安全；
- java的类是单继承的，C++可以多继承。但是java可以实现多接口；
- java有自动内存管理垃圾回收机制，不需要手动释放内存

字符型常量和字符串常量的区别

字符型常量是单引号，字符串型常量是双引号；

字符型常量对应ASCII码，相当于一个整型的值，可以进行表达式运算。字符串型常量对应一个内存地址；

字符常量占2字节，字符串型常量占若干字节

Java泛型？什么是类型擦除？常用通配符？

泛型是JDK5引入的新特性，泛型提供编译时类型的安全检测。但是java的泛型是伪泛型，因为java在编译期间会将所有的泛型信息擦除，这就是类型擦除。通过反射可以绕过类型擦除的限制。

==和equals（）的区别

如果是基本数据类型，==和equals()没有区别，都比的是值。如果是引用数据类型，==比较的就是内存地址是否相等，而equals()如果没有重写的话，比较的还是内存地址；如果重写了equals()方法，则比较的是具体的内容。

hashCode()和equals()的区别

hashCode()：是获取哈希码，返回一个int整数，以确定该对象在哈希表中的索引位置。该方法在Object类中，所有类都有该方法。

为什么要有hashCode：先使用hashCode判断对象在哈希表中是否已经存在，然后再调用equals方法判断内容，这样大幅减少比较次数，提高效率。

为什么重写equals方法必须重写hashCode方法？

我们理想的效果是，两个对象相等，则hashCode相等，对应的内容也相同。但是如果两个对象的hashCode相同，也是有可能出现两个对象不相等的情况。如果重写了equals方法，而没有重写hashcode方法，会出现equals相等的对象，hashcode不相等的情况，重写hashcode方法就是为了避免这种情况的出现。

为什么两个对象有相同的hashCode，但不一定相等？

哈希是会产生碰撞的。因此可能会出现一个hashCode对应多个对象的情况，这时候再用equals来判断是否真的相等。

解决哈希碰撞的方法：

拉链法、开放寻址法

java的几种基本数据类型，以及包装类

基本数据类型：byte、short、int、long、float、double、char、boolean
对应的包装类：Byte、Short、Integer、Long、Float、Double、Character、Boolean

基本数据类型存放在java的虚拟机栈的局部变量表中
包装类属于对象，对象的实例都在堆中

包装类的常量池

```
Byte、Short、Integer、Long这4中包装类默认创建了[-128,127]的缓存数据。Character创建了[0,127]的缓存数据

Integer i1 = 33;
Integer i2 = 33;
System.out.println(i1 == i2);      //输出true，但是超出缓存数据的值就不相等，因此建议使用equals比较
```

深拷贝和浅拷贝

浅拷贝：对基本数据类型是值传递，对引用数据类型来说，传递的是地址；
深拷贝：对基本数据类型是值传递，对引用数据类型来说，会创建一个新的对象，并复制全部内容到新的对象。

多态

一个对象具有的多种状态。具体表现为父类的引用指向子类的实例；
特点：
    对象类型和引用类型之前具有继承/实现的关系；
    引用类型变量发出的方法具体调用的是哪个类中的方法，必须在运行期间才能确定；
    多态不能调用父类中不存在的方法；
    如果子类重写了父类方法，则调用的是子类方法；否则是调用父类方法。

String、StringBuffer、StringBuilder的区别

String：
    底层使用了字符数组来保存字符串，但是该字符数组有final关键字，所以String不可变。

    虽然该数组有final修饰，但是仅仅保证的是数组的引用地址不可变，但是内容还是可变的。
    所以final关键字的作用是让String类不可继承，这样就避免子类对某些属性进行破坏。
    这个数组还使用了private修饰，并且String没有提供对外修改这个数组的方法，因此初始化后，没有外界方法能修改它。
    String不可变的主要原因在于Java作者在涉及到对这个数组修改的操作的时候，都会去创建一个新的String对象。

StringBuilder：
    底层也是字符数组，但是没有使用final修饰，因此是可变的。继承于AbstractStringBuilder类。StringBuilder没有对方法进行加锁，因此StringBuilder是线程不安全的。

StringBuffer:
    底层也是字符数组，但是没有使用final修饰，因此是可变的。继承于AbstractStringBuilder类。StringBuffer对方法进行了加锁，因此是线程安全的。

Object类的常见方法

```
getClass();      //用于返回当前对象的Class对象
hashCode();      //返回对象的哈希码
equals();        //比较两个对象的内存地址是否相同，String类对此方法进行了重写，比较内容
clone();         //是一个浅拷贝，克隆当前对象，但是没有创建新的对象
toString();      //返回实例对象的16进制哈希码，一般要对其进行重写
notify();        //唤醒一个处于等待状态的线程
notifyAll();     //唤醒所有的等待线程
wait();          //暂停线程的执行。该方法会释放锁
finalize();      //对象被垃圾回收时触发的函数
```

动态代理

JDK动态代理：

必须有一个类B实现接口A；

创建一个代理类，必须实现InvocationHandler接口，重写其invoke方法；还需创建构造方法，参数就是一个Object类；

通过Proxy的newProxyInstance方法来创建我们的代理对象；

```
public class Client
{
    public static void main(String[] args)
    {
        //    我们要代理的真实对象
        Subject realSubject = new RealSubject();

        //    我们要代理哪个真实对象，就将该对象传进去，最后是通过该真实对象来调用其方法的
        InvocationHandler handler = new DynamicProxy(realSubject);

        /*
         * 通过Proxy的newProxyInstance方法来创建我们的代理对象，我们来看看其三个参数
         * 第一个参数 handler.getClass().getClassLoader() ，我们这里使用handler这个类的ClassLoader对象来加载我们的代理对象
         * 第二个参数realSubject.getClass().getInterfaces()，我们这里为代理对象提供的接口是真实对象所实行的接口，表示我要代理的是该真实对象，这样我就能调用这组接口中的方法了
         * 第三个参数handler， 我们这里将这个代理对象关联到了上方的 InvocationHandler 这个对象上
         */
        Subject subject =
(Subject)Proxy.newProxyInstance(handler.getClass().getClassLoader(), realSubject
                .getClass().getInterfaces(), handler);

        System.out.println(subject.getClass().getName());
        subject.rent();
        subject.hello("world");
    }
}
```

final、static、this、super关键字

```
final：最终的、不可修改的，常用来修饰类、方法和变量。
    修饰变量保证该变量属性是只读的，无法修改。
    修饰类保证了该类中方法无法重写，防止子类破坏其不可变性。
static：静态的。常用来修饰成员变量和方法、代码块、内部类。
this：等价于引用类的当前实例。作用于构造方法的时候要放首行。
super：从子类访问父类的变量和方法。作用于构造方法的时候要放首行。
```

保护性拷贝

```
通过创建副本对象来避免共享的方式。
比如String字符串，如果对其改变，则是创建一个新的字符串对象，而不是修改原来的字符串。

带来的问题：对象创建太多

解决方案：享元模式，如果需要重用数量有限的同一对象的时候

比如包装类的常量缓存：
Byte、Short、Long缓存的范围都是-128~127；
Character缓存的范围是0~127；
Integer的默认范围是-128~127，最小值不能变，但是最大值可以通过调整虚拟机参数来改变；
Boolean缓存了TRUE和FALSE

String字符串、BigDecimal、BigInteger都是享元模式，不可变类，线程安全。
```

获取Class对象的方式

```
1.Class alunbarClass = TargetObject.class;
2.Class alunbarClass1 = Class.forName("cn.javaguide.TargetObject");
3.TargetObject o = new TargetObject();
  Class alunbarClass2 = o.getClass();
4.class clazz = ClassLoader.LoadClass("cn.javaguide.TargetObject");
```

反射的操作实例

```
Class<?> tagetClass = Class.forName("cn.javaguide.TargetObject");        //获取
Class对象
TargetObject targetObject = (TargetObject) tagetClass.newInstance();     //通过
newInstance创建对象
Method method = targetObject.getClass().getMethod("sayHello", String.class);
//通过对象反射方法
method.invoke(targetObject, "张三");       //执行方法
```

集合的底层框架

ArrayList：Object[]数组
Vector：Object[]数组
LinkedList：双向链表(jdk1.6之前还有循环)

HashSet：HashMap
LinkedHashSet：LinkedHashMap
TreeSet：红黑树

HashMap：jdk1.8之前是数组+链表，也就是数组的每个位置上都有一个链表，解决哈希冲突问题(拉链法)。jdk1.8之后采用数组+链表/红黑树。当链表长度大于8的时候，会将链表转成红黑树。另外如果当前数组长度小于64，则会先进行数组扩容。
LinkedHashMap：底层和HashMap一样，还多了个双向链表，使得哈希表插入有序。
HashTable：数组+链表
TreeMap：红黑树

Vector和ArrayList

Vector是List的古老实现类，底层是Object[]数组，线程安全；
ArrayList是List的主要实现类，底层是Object[]数组，线程不安全。

为什么Vector被弃用？
    因为线程安全导致效率低下；
    Vector的扩容是扩容一倍，而ArrayList是扩容一半；
    Vector分配内存需要连续的内存空间，如果数据太多，容易失败；
    Vector只能在尾部进行插入和删除，效率低。

B树、B+树、红黑树

B树：
    一种多路搜索树；
    特点：
        任意非叶节点最多只能有M个儿子，M>2；
        根节点的儿子数为[2，M]；
        所有叶子节点位于同一层。
B+树：
    是B树的变体，主要区别在于：
        所有关键字都出现在叶节点的链表中，所有叶节点都有一个链指针。非叶子节点作为索引。
红黑树：
    自平衡二叉查找树
    特点：
        节点是红色或者黑色；
        根节点是黑色；
        每个叶节点是黑色的null节点；
        每个红色节点的两个子节点是黑色；
        从任一一个节点到每个叶节点的路径会包含相同数量的黑色节点

ArrayList的扩容

    通过无参构造的方法创建ArrayList的时候，一开始是一个空数组，只有在对数组进行元素添加的时候才会分配容量，且初始容量是10。当数组需要扩容是调用底层的grow方法进行扩容。并且更新容量为旧容量的1.5倍。

comparable和Comparator的区别

comparable是在lang包下，有一个compareTo(Object obj)方法来帮助排序；这个相当于内部排序，只要实现这个接口的对象(数组)就相当于有了排序能力。

Comparator在util包下，有一个compare(Object obj1, Object obj2)方法来排序；这是外部排序，被称为比较器。通过它定义比较的方式，再传入Collection.sort()和Arrays.sort()中对目标排序。

HashMap和HashTable的区别

1.线程是否安全：

HashMap是非线程安全的；HashTable是线程安全的，因为其底层使用了synchronized关键字修饰。但HashTable基本不使用了。

2.效率：HashMap效率高于HashTable

3.是否支持null键和null值：

HashMap对null的键和值都支持，但是null的键只能有一个；

HashTable对两个都不支持

4.初始容量和扩容大小：

HashMap初始容量为16，之后每次扩容为原来的2倍；如果初始自定义了大小，则后面HashMap会将其扩充为2的幂次方；

HashTable初始容量为11，每次扩容会变为原来的2n+1；

HashSet如何检查重复

当要把对象加入HashSet的时候，HashSet会先计算对象的hashCode来判断对象的加入位置，同时会与set中其他对象的hashCode进行比较。如果有相同的hashCode，再比较equals()

HashMap的长度为什么是2的幂次方?

Hash值的范围很大，大概有40亿的映射空间，这无法直接放到内存中，所以Hash值使用之前会先与数组长度进行取模运算，得到的余数才是数组中对应的存放位置。而取余运算如果除数是2的幂次方则等价于与除数-1的与运算：hash%length==hash&(length-1)，并且使用与运算相比于取余运算能提升效率。

get方法中有一步是(n-1) & h，h是哈希值，n是数组长度。如果n是2的幂次方，则效率能有很大提升。

HashMap的多线程死循环

当插入一个新的节点时，如果不存在相同的key，则会判断当前内部元素是否已经达到阈值（默认是数组大小的0.75，初始容量是16），如果已经达到阈值，会对数组进行扩容，将现有元素迁移到已经扩容的数组中去。

rehash后需要将原数据放到扩容后的数组的链表中，1.7以前采用的是头插法，之后采用的是尾插法

头插法会改变顺序，如果是多线程的情况下可能会出现指向原来元素的情况，造成死循环。

1.8虽然采用了尾插法解决了死循环问题，但是并不意味着在多线程下就安全了，还可能会出现其他的问题(如扩容丢失等).

ConcurrentHashMap

jdk1.8版本

采用数组+链表/红黑树的方式来实现，数组和扩容参数都用了volatile修饰，通过CAS保证线程安全。

ConcurrentHashMap不允许key、value有null。

初始化：

使用cas来保证并发安全，懒惰初始化数组。 (在第一次使用的时候才会初始化数组，默认长度16)

链表转红黑树：

如果数组长度大于64且链表长度大于8，则将链表转换为红黑树，提高查找效率。如果数组长度小于64，则会先扩容。
put方法：
如果数组还没有创建，则先通过CAS创建数组；
数组创建后，如果这个位置的链表还没有创建，则通过CAS创建链表；
如果该线程正在进行扩容，如果其他线程需要put，则要判断链表是否有ForwardingNode。如果有则帮助扩容；
如果链表已经存在，则锁住链表头(synchronized)，然后进行后续操作，元素添加到链表尾部(根据哈希码的正负判断是链表插入还是红黑树插入)；
get方法：
这个不需要加锁，仅需要保证可见性，如果是在扩容过程中，其他线程来get，则会判断链表头是否为ForwardingNode，有的话去新数组搜索。
扩容：
新数组长度=旧数组长度*2；
迁移的时候以链表为单位进行移动。如果当前链表已经迁移完成，则将链表头置为ForwardingNode；
如果发现当前链表头已经是ForwardingNode，则继续处理下一个链表；
如果当前链表还未处理，则将链表头加锁(synchronized)，然后根据是链表还是红黑树来处理。

jdk1.7版本
维护一个Segment数组，每一个Segment对应一把锁，每一个Segment里面是一个小的hash表，hash表的结构是数组+链表
优点：多线程访问不同的Segment，则没有冲突
缺点：Segment数组默认长度为16，且无法扩容，并且不支持懒惰初始化

Segment锁是继承于ReentrantLock
链表的插入还是使用的是头插法
扩容还是长度*2
get方法也没有加锁，只用了UNSAFE方法保证可见性
长度计算：
先不加锁计算两次，如果结果一样就正确返回
如果不一样，则进行重试，如果重试次数超过3，则将所有Segment锁住，再重新计算

ConcurrentHashMap和HashTable的区别

ConcurrentHashMap：jdk1.7底层采用分段数组+链表实现；jdk1.8改为数组+链表/红黑树
HashTable：底层采用数据+链表的形式

ConcurrentHashMap：jdk1.7之前采用分段锁来实现线程安全，提高访问率；jdk1.8之后使用synchronized和CAS来实现线程安全。
HashTable：只用了synchronized来保证线程安全。效率很低。

系统调用

当处于用户态的我们想要进行IO操作的时候，因为要访问内核空间的资源，我们没有直接的权限去访问，因此我们必须通过系统调用的方式来间接访问内核空间。应用程序对操作系统发起系统调用，由操作系统的内核来执行对应的IO操作。

当发起IO调用后，一共有两个步骤：

内核等待IO设备准备好数据；

内核将数据从内核空间拷贝到用户空间。

常见的I/O模型

同步阻塞I/O、同步非阻塞I/O、I/O多路复用、信号驱动I/O和异步I/O

BIO

同步阻塞I/O：

应用程序发起read操作的时候，会一直阻塞，直到内核把数据拷贝到用户空间中。

服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销。

NIO

同步非阻塞I/O：

发起请求后不阻塞，而是在内核将数据拷贝到用户空间的时候再阻塞，接收数据。

服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

AIO

异步非阻塞I/O：

请求发起后会直接返回，不产生堵塞，当后台处理完成，操作系统会通知相应的线程进行后续操作。

服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。