Java基础面试题

一、面向对象 （☆☆☆）

1、谈谈对java多态的理解?

多态是指父类的某个方法被子类重写时，可以产生自己的功能行为，同一个操作作用于不同对象，可以有不同的解释，产生不同的执行结果。

多态的三个必要条件：

- 继承父类。
- 重写父类的方法。
- 父类的引用指向子类对象。

什么是多态

面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

多态的作用：消除类型之间的耦合关系。

现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。

多态的好处：

1.可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆Circle类工作，对其他任何圆形几何体，如圆环，也同样工作。

2.可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

3.接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。

4.灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。

5.简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。

Java中多态的实现方式：接口实现，继承父类进行方法重写，同一个类中进行方法重载。

2、你所知道的设计模式有哪些?

答：Java 中一般认为有23种设计模式，我们不需要所有的都会，但是其中常用的种设计模式应该去掌握。下面列出了所有的设计模式。要掌握的设计模式我单独列出来了，当然能掌握的越多越好。

总体来说设计模式分为三大类：

创建型模式，共五种：

工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：

适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：

策略模式、模板方法模式、观者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

具体可见我的设计模式总结笔记

3、通过静态内部类实现单例模式有哪些优点?

1. 不用 synchronized ， 节省时间。
2. 调用 getInstance() 的时候才会创建对象，不调用不创建，节省空间，这有点像传说中的懒汉式。

4、静态代理和动态代理的区别，什么场景使用?

静态代理与动态代理的区别在于代理类生成的时间不同，即根据程序运行前代理类是否已经存在，可以将代理分为静态代理和动态代理。如果需要对多个类进行代理，并且代理的功能都是一样的，用静态代理重复编写代理类就非常的麻烦，可以用动态代理动态的生成代理类。

```
// 为目标对象生成代理对象
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("开启事务");

                    // 执行目标对象方法
                    Object returnValue = method.invoke(target, args);

                    System.out.println("提交事务");
                    return null;
                }
            });
}
```

- 静态代理使用场景：四大组件同AIDL与AMS进行跨进程通信
- 动态代理使用场景：Retrofit使用了动态代理极大地提升了扩展性和可维护性。

5、简单工厂、工厂方法、抽象工厂、Builder模式的区别?

- 简单工厂模式：一个工厂方法创建不同类型的对象。
- 工厂方法模式：一个具体的工厂类负责创建一个具体对象类型。
- 抽象工厂模式：一个具体的工厂类负责创建一系列相关的对象。
- Builder模式：对象的构建与表示分离，它更注重对象的创建过程。

6、装饰模式和代理模式有哪些区别？与桥接模式相比呢?

- 1、装饰模式是以客户端透明的方式扩展对象的功能，是继承关系的一个替代方案；而代理模式则是给一个对象提供一个代理对象，并由代理对象来控制对原有对象的引用。

- 2、装饰模式应该为所装饰的对象增强功能；代理模式对代理的对象施加控制，但不对对象本身的功能进行增加。
- 3、桥接模式的作用于代理、装饰截然不同，它主要是为了应对某个类族有多个变化维度导致子类类型急剧增多的场景。通过桥接模式将多个变化维度隔离开，使得它们可以独立地变化，最后通过组合使它们应对多维变化，减少子类的数量和复杂度。

7、外观模式和中介模式的区别?

外观模式重点是对外封装统一的高层接口，便于用户使用；而中介模式则是避免多个互相协作的对象直接引用，它们之间的交互通过一个中介对象进行，从而使得它们耦合松散，能够易于应对变化。

8、策略模式和状态模式的区别?

虽然两者的类型结构是一致的，但是它们的本质却是不一样的。策略模式重在整个算法的替换，也就是策略的替换，而状态模式则是通过状态来改变行为。

9、适配器模式，装饰者模式，外观模式的异同?

这三个模式的相同之处是，它们都作用于用户与真实被使用的类或系统之间，作一个中间层，起到了让用户间接地调用真实的类的作用。它们的不同之外在于，如上所述的应用场合不同和本质的思想不同。

代理与外观的主要区别在于，代理对象代表一个单一对象，而外观对象代表一个子系统，代理的客户对象无法直接访问对象，由代理提供单独的目标对象的访问，而通常外观对象提供对子系统各元件功能的简化的共同层次的调用接口。代理是一种原来对象的代表，其它需要与这个对象打交道的操作都是和这个代表交涉的。而适配器则不需要虚构出一个代表者，只需要为应付特定使用目的，将原来的类进行一些组合。

外观与适配器都是对现存系统的封装。外观定义的新的接口，而适配器则是复用一个原有的接口，适配器是使两个已有的接口协同工作，而外观则是为现存系统提供一个更为方便的访问接口。如果硬要说外观是适配，那么适配器有用来适配对象的，而外观是用来适配整个子系统的。也就是说，外观所针对的对象的粒度更大。

代理模式提供与真实的类一致的接口，意在用代理类来处理真实的类，实现一些特定的服务或真实类的部分功能，Facade（外观）模式注重简化接口，Adapter（适配器）模式注重转换接口。

10、代码的坏味道：

1、代码重复：

代码重复几乎是最常见的异味了。他也是Refactoring 的主要目标之一。代码重复往往来自于copy-and-paste 的编程风格。

2、方法过长：

一个方法应当具有自我独立的意图，不要把几个意图放在一起。

3、类提供的功能太多：

把太多的责任交给了一个类，一个类应该仅提供一个单一的功能。

4、数据泥团：

某些数据通常像孩子一样成群玩耍：一起出现在很多类的成员变量中，一起出现在许多方法的参数中.....，这些数据或许应该自己独立形成对象。 比如以单例的形式对外提供自己的实例。

5、冗赘类：

一个干活不多的类。类的维护需要额外的开销，如果一个类承担了太少的责任，应当消除它。

6、需要太多注释：

经常觉得要写很多注释表示你的代码难以理解。如果这种感觉太多，表示你需要Refactoring。

11、是否能从Android中举几个例子说说用到了什么设计模式？

AlertDialog、Notification源码中使用了Bulider（建造者）模式完成参数的初始化：

在AlertDialog的Builder模式中并没有看到Direcotr角色的出现，其实在很多场景中，Android并没有完全按照GOF的经典设计模式来实现，而是做了一些修改，使得这个模式更易于使用。这个的AlertDialog.Builder同时扮演了上下文中提到的builder、ConcreteBuilder、Director的角色，简化了Builder模式的设计。当模块比较稳定，不存在一些变化时，可以在经典模式实现的基础上做出一些精简，而不是照搬GOF上的经典实现，更不要生搬硬套，使程序失去架构之美。

定义：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。即将配置从目标类中隔离出来，避免过多的setter方法。

优点：

- 1、良好的封装性，使用建造者模式可以使客户端不必知道产品内部组成的细节。
- 2、建造者独立，容易扩展。

缺点：

- 会产生多余的Builder对象以及Director对象，消耗内存。

日常开发的BaseActivity抽象工厂模式：

定义：为创建一组相关或者是相互依赖的对象提供一个接口，而不需要指定它们的具体类。

主题切换的应用：

比如我们的应用中有两套主题，分别为亮色主题LightTheme和暗色主题DarkTheme，这两种主题我们可以通过一个抽象的类或接口来定义，而在对应主题下我们又有各类不同的UI元素，比如Button、TextView、Dialog、ActionBar等，这些UI元素都会分别对应不同的主题，这些UI元素我们也可以通过抽象的类或接口定义，抽象的主题、具体的主题、抽象的UI元素和具体的UI元素之间的关系就是抽象工厂模式最好的体现。

优点：

- 分离接口与实现，面向接口编程，使其从具体的产品实现中解耦，同时基于接口与实现的分离，使抽象该工厂方法模式在切换产品类时更加灵活、容易。

缺点：

- 类文件的爆炸性增加。
- 新的产品类不易扩展。

Okhttp内部使用了责任链模式来完成每个Interceptor拦截器的调用：

定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

ViewGroup事件传递的递归调用就类似一条责任链，一旦其寻找到责任者，那么将由责任者持有并消费掉该次事件，具体体现在View的onTouchEvent方法中返回值的设置，如果onTouchEvent返回false，那么意味着当前View不会是该次事件的责任人，将不会对其持有；如果为true则相反，此时View会持有该事件并不再向下传递。

优点：

将请求者和处理者关系解耦，提供代码的灵活性。

缺点：

对链中请求处理者的遍历中，如果处理者太多，那么遍历必定会影响性能，特别是在一些递归调用中，要慎重。

RxJava的观察者模式：

定义：定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

ListView/RecyclerView的Adapter的notifyDataSetChanged方法、广播、事件总线机制。

观察者模式主要的作用就是对象解耦，将观察者与被观察者完全隔离，只依赖于Observer和Observable抽象。

优点：

- 观察者和被观察者之间是抽象耦合，应对业务变化。
- 增强系统灵活性、可扩展性。

缺点：

- 在Java中消息的通知默认是顺序执行，一个观察者卡顿，会影响整体的执行效率，在这种情况下，一般考虑采用异步的方式。

AIDL代理模式：

定义：为其他对象提供一种代理以控制对这个对象的访问。

静态代理：代码运行前代理类的class编译文件就已经存在。

动态代理：通过反射动态地生成代理者的对象。代理谁将会在执行阶段决定。将原来代理类所做的工作由InvocationHandler来处理。

使用场景：

- 当无法或不想直接访问某个对象或访问某个对象存在困难时可以通过一个代理对象来间接访问，为了保证客户端使用的透明性，委托对象与代理对象需要实现相同的接口。

缺点：

- 对类的增加。

ListView/RecyclerView/GridView的适配器模式：

适配器模式把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

使用场景：

- 接口不兼容。
- 想要建立一个可以重复使用的类。
- 需要一个统一的输出接口，而输入端的类型不可预知。

优点：

- 更好的复用性：复用现有的功能。
- 更好的扩展性：扩展现有的功能。

缺点：

- 过多地使用适配器，会让系统非常零乱，不易于整体把握。例如，明明看到调用的是A接口，其实内部被适配成了B接口的实现，一个系统如果出现太多这种情况，无异于一场灾难。

Context/ContextImpl外观模式：

要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行，门面模式提供一个高层次的接口，使得子系统更易于使用。

使用场景：

- 为一个复杂子系统提供一个简单接口。

优点：

- 对客户程序隐藏子系统细节，因而减少了客户对于子系统的耦合，能够拥抱变化。
- 外观类对子系统的接口封装，使得系统更易用使用。

缺点：

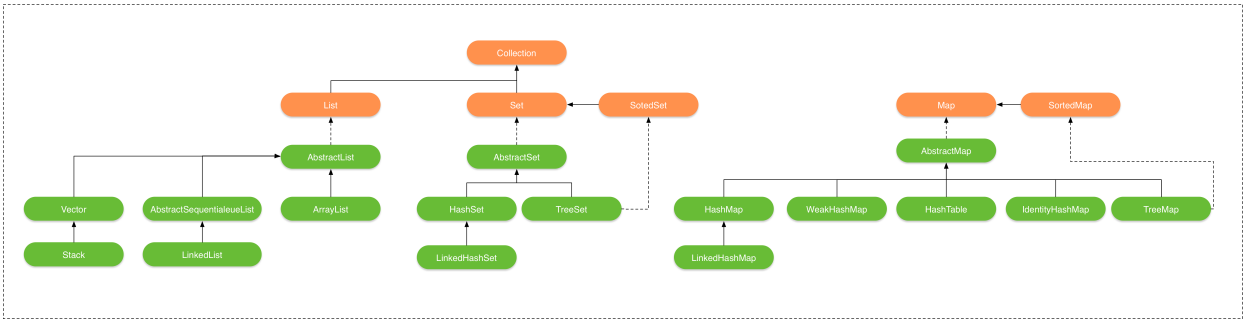
- 外观类接口膨胀。
- 外观类没有遵循开闭原则，当业务出现变更时，可能需要直接修改外观类。

二、集合框架 （☆☆☆）

1、集合框架，list，map，set都有哪些具体的实现类，区别都是什么?

Java集合里使用接口来定义功能，是一套完善的继承体系。Iterator是所有集合的总接口，其他所有接口都继承于它，该接口定义了集合的遍历操作，Collection接口继承于Iterator，是集合的次级接口（Map独立存在），定义了集合的一些通用操作。

Java集合的类结构图如下所示：



List：有序、可重复；索引查询速度快；插入、删除伴随数据移动，速度慢；

Set：无序，不可重复；

Map：键值对，键唯一，值多个；

1.List,Set都是继承自Collection接口，Map则不是;

2.List特点：元素有放入顺序，元素可重复;

Set特点：元素无放入顺序，元素不可重复，重复元素会盖掉，（注意：元素虽然无放入顺序，但是元素在set中位置是由该元素的HashCode决定的，其位置其实是固定，加入Set 的Object必须定义equals()方法;

另外list支持for循环，也就是通过下标来遍历，也可以使用迭代器，但是set只能用迭代，因为他无序，无法用下标取得想要的值）。

3.Set和List对比：

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

4.Map适合储存键值对的数据。

5.线程安全集合类与非线程安全集合类

LinkedList、ArrayList、HashSet是非线程安全的，Vector是线程安全的;

HashMap是非线程安全的，HashTable是线程安全的;

StringBuilder是非线程安全的，StringBuffer是线程安的。

下面是这些类具体的使用介绍：

ArrayList与LinkedList的区别和适用场景

Arraylist：

优点：ArrayList是实现了基于动态数组的数据结构,因地址连续，一旦数据存储好了，查询操作效率会比较高（在内存里是连着放的）。

缺点：因为地址连续，ArrayList要移动数据,所以插入和删除操作效率比较低。

LinkedList：

优点：LinkedList基于链表的数据结构，地址是任意的，其在开辟内存空间的时候不需要等一个连续的地址，对新增和删除操作add和remove，LinedList比较占优势。LikedList适用于要头尾操作或插入指定位置的场景。

缺点：因为LinkedList要移动指针,所以查询操作性能比较低。

适用场景分析：

当需要对数据进行对此访问的情况下选用ArrayList，当要对数据进行多次增加删除修改时采用LinkedList。

ArrayList和LinkedList怎么动态扩容的吗?

ArrayList:

ArrayList 初始化大小是 10 （如果你知道你的arrayList 会达到多少容量，可以在初始化的时候就指定，能节省扩容的性能开支） 扩容点规则是，新增的时候发现容量不够用了，就去扩容 扩容大小规则是，扩容后的大小= 原始大小+原始大小/2 + 1。（例如：原始大小是 10 ，扩容后的大小就是 10 + 5+1 = 16）

LinkedList:

linkedList 是一个双向链表，没有初始化大小，也没有扩容的机制，就是一直在前面或者后面新增就好。

ArrayList与Vector的区别和适用场景

ArrayList有三个构造方法：

```
public ArrayList(intinitialCapacity)// 构造一个具有指定初始容量的空列表。
public ArrayList()// 构造一个初始容量为10的空列表。
public ArrayList(Collection<? extends E> c)// 构造一个包含指定 collection 的元素的列表
```

Vector有四个构造方法：

```
public Vector() // 使用指定的初始容量和等于零的容量增量构造一个空向量。
public Vector(int initialCapacity) // 构造一个空向量，使其内部数据数组的大小，其标准容量增量为零。
public Vector(Collection<? extends E> c)// 构造一个包含指定 collection 中的元素的向量
public Vector(int initialCapacity, int capacityIncrement)// 使用指定的初始容量和容量增量构造一个空的向量
```

ArrayList和Vector都是用数组实现的，主要有这么四个区别：

1)Vector是多线程安全的，线程安全就是说多线程访问代码，不会产生不确定的结果。而ArrayList不是，这可以从源码中看出，Vector类中的方法很多有synchronied进行修饰，这样就导致了Vector在效率上无法与ArrayLst相比；

2)两个都是采用的线性连续空间存储元素，但是当空间充足的时候，两个类的增加方式是不同。

3)Vector可以设置增长因子，而ArrayList不可以。

4)Vector是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

适用场景：

1.Vector是线程同步的，所以它也是线程安全的，而ArraList是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用ArrayList效率比较高。

2.如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用Vector有一定的优势。

HashSet与TreeSet的区别和适用场景

1.TreeSet 是二叉树（红黑树的树据结构）实现的，Treest中的数据是自动排好序的，不允许放入null值。

2.HashSet 是哈希表实现的，HashSet中的数据是无序的可以放入null，但只能放入一个null，两者中的值都不重复，就如数据库中唯一约束。

3.HashSet要求放入的对象必须实现HashCode()方法，放的对象，是以hashcode码作为标识的，而具有相同内容的String对象，hashcode是一样，所以放入的内容不能重复但是同一个类的对象可以放入不同的实例。

适用场景分析:

HashSet是基于Hash算法实现的，其性能通常都优于TreeSet。为快速查找而设计的Set，我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。

HashMap与TreeMap、HashTable的区别及适用场景

HashMap 非线程安全

HashMap：基于哈希表(散列表)实现。使用HashMap要求的键类明确定义了hashCode()和equals()[可以重写hasCode()和equals()]，为了优化HashMap空间的使用，您可以调优初始容量和负载因子。其中散列表的冲突处理主分两种，一种是开放定址法，另一种是链表法。HashMap实现中采用的是链表法。

TreeMap：非线程安全基于红黑树实现。TreeMap没有调优选项，因为该树总处于平衡状态。

适用场景分析：

HashMap和HashTable:HashMap去掉了HashTable的contain方法，但是加上了containsValue()和containsKey()方法。HashTable是同步的，而HashMap是非同步的，效率上比HashTable要高。HashMap允许空键值，而HashTable不允许。

HashMap：适用于Map中插入、删除和定位元素。

Treemap：适用于按自然顺序或自定义顺序遍历键(key)。 (ps:其实我们工作的过程中对集合的使用是很频繁的,稍注意并总结积累一下,在面试的时候应该会回答的很轻松)

2、set集合从原理上如何保证不重复?

1）在往set中添加元素时，如果指定元素不存在，则添加成功。

2）具体来讲：当向HashSet中添加元素的时候，首先计算元素的hashcode值，然后用这个（元素的hashcode）%（HashMap集合的大小）+1计算出这个元素的存储位置，如果这个位置为空，就将元素添加进去；如果不为空，则用equals方法比较元素是否相等，相等就不添加，否则找一个空位添加。

3、HashMap和HashTable的主要区别是什么？，两者底层实现的数据结构是什么？

HashMap和HashTable的区别：

二者都实现了Map 接口，是将唯一的键映射到特定的值上，主要区别在于：

1)HashMap 没有排序，允许一个null 键和多个null 值,而Hashtable 不允许；

2)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；

3)Hashtable 继承自Dictionary 类，HashMap 是Java1.2 引进的Map 接口的实现；

4)Hashtable 的方法是Synchronized 的，而HashMap 不是，在多个线程访问Hashtable 时，不需要自己为它的方法实现同步，而HashMap 就必须为之提供额外的同步。
Hashtable 和HashMap 采用的hash/rehash 算法大致一样，所以性能不会有很大的差异。

HashMap和HashTable的底层实现数据结构：

HashMap和Hashtable的底层实现都是数组 + 链表结构实现的（jdk8以前）

4、HashMap、ConcurrentHashMap、hash()相关原理解析？

HashMap 1.7的原理：

HashMap 底层是基于 数组 + 链表 组成的，不过在 jdk1.7 和 1.8 中具体实现稍有不同。

负载因子：

- 给定的默认容量为 16，负载因子为 0.75。Map 在使用过程中不断的往里面存放数据，当数量达到了 16 * 0.75 = 12 就需要将当前 16 的容量进行扩容，而扩容这个过程涉及到 rehash、复制数据等操作，所以非常消耗性能。
- 因此通常建议能提前预估 HashMap 的大小最好，尽量的减少扩容带来的性能损耗。

其实真正存放数据的是 Entry<K,V>[] table，Entry 是 HashMap 中的一个静态内部类，它有key、value、next、hash（key的hashcode）成员变量。

put 方法：

- 判断当前数组是否需要初始化。
- 如果 key 为空，则 put 一个空值进去。

- 根据 key 计算出 hashcode。
- 根据计算出的 hashcode 定位出所在桶。
- 如果桶是一个链表则需要遍历判断里面的 hashcode、key 是否和传入 key 相等，如果相等则进行覆盖，并返回原来的值。
- 如果桶是空的，说明当前位置没有数据存入，新增一个 Entry 对象写入当前位置。（当调用 addEntry 写入 Entry 时需要判断是否需要扩容。如果需要就进行两倍扩充，并将当前的 key 重新 hash 并定位。而在 createEntry 中会将当前位置的桶传入到新建的桶中，如果当前桶有值就会在位置形成链表。）

get 方法：

- 首先也是根据 key 计算出 hashcode，然后定位到具体的桶中。
- 判断该位置是否为链表。
- 不是链表就根据 key、key 的 hashcode 是否相等来返回值。
- 为链表则需要遍历直到 key 及 hashcode 相等时候就返回值。
- 啥都没取到就直接返回 null 。

HashMap 1.8的原理：

当 Hash 冲突严重时，在桶上形成的链表会变的越来越长，这样在查询时的效率就会越来越低；时间复杂度为 O(N)，因此 1.8 中重点优化了这个查询效率。

TREEIFY_THRESHOLD 用于判断是否需要将链表转换为红黑树的阈值。

HashEntry 修改为 Node。

put 方法：

- 判断当前桶是否为空，空的就需要初始化（在resize方法 中会判断是否进行初始化）。
- 根据当前 key 的 hashcode 定位到具体的桶中并判断是否为空，为空表明没有 Hash 冲突就直接在当前位置创建一个新桶即可。
- 如果当前桶有值（ Hash 冲突），那么就要比较当前桶中的 key、key 的 hashcode 与写入的 key 是否相等，相等就赋值给 e,在第 8 步的时候会统一进行赋值及返回。
- 如果当前桶为红黑树，那就要按照红黑树的方式写入数据。
- 如果是个链表，就需要将当前的 key、value 封装成一个新节点写入到当前桶的后面（形成链表）。
- 接着判断当前链表的大小是否大于预设的阈值，大于时就要转换为红黑树。
- 如果在遍历过程中找到 key 相同时直接退出遍历。
- 如果 e != null 就相当于存在相同的 key,那就需要将值覆盖。
- 最后判断是否需要进行扩容。

get 方法：

- 首先将 key hash 之后取得所定位的桶。
- 如果桶为空则直接返回 null 。
- 否则判断桶的第一个位置(有可能是链表、红黑树)的 key 是否为查询的 key，是就直接返回 value。
- 如果第一个不匹配，则判断它的下一个是红黑树还是链表。
- 红黑树就按照树的查找方式返回值。
- 不然就按照链表的方式遍历匹配返回值。

修改为红黑树之后查询效率直接提高到了 O(logn)。但是 HashMap 原有的问题也都存在，比如在并发场景下使用时容易出现死循环：

- 在 HashMap 扩容的时候会调用 resize() 方法，就是这里的并发操作容易在一个桶上形成环形链表；这样当获取一个不存在的 key 时，计算出的 index 正好是环形链表的下标就会出现死循环：在 1.7 中 hash 冲突采用的头插法形成的链表，在并发条件下会形成循环链表，一旦有查询落到了这个链表上，当获取不到值时就会死循环。

ConcurrentHashMap 1.7原理：

ConcurrentHashMap 采用了分段锁技术，其中 Segment 继承于 ReentrantLock。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理，理论上 ConcurrentHashMap 支持 CurrencyLevel (Segment 数组数量)的线程并发。每当一个线程占用锁访问一个 Segment 时，不会影响到其他的 Segment。

put 方法:

首先是通过 key 定位到 Segment，之后在对应的 Segment 中进行具体的 put。

- 虽然 HashEntry 中的 value 是用 volatile 关键词修饰的，但是并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理。
- 首先第一步的时候会尝试获取锁，如果获取失败肯定就有其他线程存在竞争，则利用 scanAndLockForPut() 自旋获取锁:

  尝试自旋获取锁。 如果重试的次数达到了 MAX_SCAN_RETRIES 则改为阻塞锁获取，保证能获取成功。
- 将当前 Segment 中的 table 通过 key 的 hashcode 定位到 HashEntry。
- 遍历该 HashEntry，如果不为空则判断传入的 key 和当前遍历的 key 是否相等，相等则覆盖旧的 value。
- 为空则需要新建一个 HashEntry 并加入到 Segment 中，同时会先判断是否需要扩容。
- 最后会使用unlock()解除当前 Segment 的锁。

get 方法：

- 只需要将 Key 通过 Hash 之后定位到具体的 Segment，再通过一次 Hash 定位到具体的元素上。
- 由于 HashEntry 中的 value 属性是用 volatile 关键词修饰的，保证了内存可见性，所以每次获取时都是最新值。
- ConcurrentHashMap 的 get 方法是非常高效的，因为整个过程都不需要加锁。

ConcurrentHashMap 1.8原理：

1.7 已经解决了并发问题，并且能支持 N 个 Segment 这么多次数的并发，但依然存在 HashMap 在 1.7 版本中的问题：那就是查询遍历链表效率太低。和 1.8 HashMap 结构类似：其中抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。

CAS：

如果obj内的value和expect相等，就证明没有其他线程改变过这个变量，那么就更新它为update，如果这一步的CAS没有成功，那就采用自旋的方式继续进行CAS操作。

问题：

- 目前在JDK的atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。
- 如果CAS不成功，则会原地自旋，如果长时间自旋会给CPU带来非常大的执行开销。

put 方法：

- 根据 key 计算出 hashcode 。
- 判断是否需要进行初始化。
- 如果当前 key 定位出的 Node为空表示当前位置可以写入数据，利用 CAS 尝试写入，失败则自旋保证成功。
- 如果当前位置的 hashcode == MOVED == -1,则需要进行扩容。
- 如果都不满足，则利用 synchronized 锁写入数据。
- 最后，如果数量大于 TREEIFY_THRESHOLD 则要转换为红黑树。

get 方法：

- 根据计算出来的 hashcode 寻址，如果就在桶上那么直接返回值。
- 如果是红黑树那就按照树的方式获取值。
- 就不满足那就按照链表的方式遍历获取值。

1.8 在 1.7 的数据结构上做了大的改动，采用红黑树之后可以保证查询效率（O(logn)），甚至取消了 ReentrantLock 改为了 synchronized，这样可以看出在新版的 JDK 中对 synchronized 优化是很到位的。

HashMap、ConcurrentHashMap 1.7/1.8实现原理

hash()算法全解析

HashMap何时扩容：

当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即大于当前数组的长度乘以加载因子的值的时候，就要自动扩容。

扩容的算法是什么：

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是使用一个新的数组代替已有的容量小的数组。

Hashmap如何解决散列碰撞（必问）？

Java中HashMap是利用"拉链法"处理HashCode的碰撞问题。在调用HashMap的put方法或get方法时，都会首先调用hashcode方法，去查找相关的key，当有冲突时，再调用equals方法。hashMap基于hasing原理，我们通过put和get方法存取对象。当我们将键值对传递给put方法时，他调用键对象的hashCode()方法来计算hashCode，然后找到bucket（哈希桶）位置来存储对象。当获取对象时，通过键对象的equals()方法找到正确的键值对，然后返回值对象。HashMap使用链表来解决碰撞问题，当碰撞发生了，对象将会存储在链表的下一个节点中。hashMap在每个链表节点存储键值对对象。当两个不同的键却有相同的hashCode时，他们会存储在同一个bucket位置的链表中。键对象的equals()来找到键值对。

Hashmap底层为什么是线程不安全的?

- 并发场景下使用时容易出现死循环，在 HashMap 扩容的时候会调用 resize() 方法，就是这里的并发操作容易在一个桶上形成环形链表；这样当获取一个不存在的 key 时，计算出的 index 正好是环形链表的下标就会出现死循环；
- 在 1.7 中 hash 冲突采用的头插法形成的链表，在并发条件下会形成循环链表，一旦有查询落到了这个链表上，当获取不到值时就会死循环。

5、ArrayMap跟SparseArray在HashMap上面的改进?

HashMap要存储完这些数据将要不断的扩容，而且在此过程中也需要不断的做hash运算，这将对我们的内存空间造成很大消耗和浪费。

SparseArray:

SparseArray比HashMap更省内存，在某些条件下性能更好，主要是因为它避免了对key的自动装箱（int转为Integer类型），它内部则是通过两个数组来进行数据存储的，一个存储key，另外一个存储value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间，我们从源码中可以看到key和value分别是用数组表示：

```
private int[] mKeys;
private Object[] mValues;
```

同时，SparseArray在存储和读取数据时候，使用的是二分查找法。也就是在put添加数据的时候，会使用二分查找法和之前的key比较当前我们添加的元素的key的大小，然后按照从小到大的顺序排列好，所以，SparseArray存储的元素都是按元素的key值从小到大排列好的。 而在获取数据的时候，也是使用二分查找法判断元素的位置，所以，在获取数据的时候非常快，比HashMap快的多。

ArrayMap:

ArrayMap利用两个数组，mHashes用来保存每一个key的hash值，mArrray大小为mHashes的2倍，依次保存key和value。

```
mHashes[index] = hash;
mArray[index<<1] = key;
mArray[(index<<1)+1] = value;
```

当插入时，根据key的hashcode()方法得到hash值，计算出在mArrays的index位置，然后利用二分查找找到对应的位置进行插入，当出现哈希冲突时，会在index的相邻位置插入。

假设数据量都在千级以内的情况下:

1、如果key的类型已经确定为int类型，那么使用SparseArray，因为它避免了自动装箱的过程，如果key为long类型，它还提供了一个LongSparseArray来确保key为long类型时的使用

2、如果key类型为其它的类型，则使用ArrayMap。

三、反射 （☆☆☆）

1、说说你对Java反射的理解?

答：Java 中的反射首先是能够获取到Java中要反射类的字节码， 获取字节码有三种方法：

1.Class.forName(className)

2.类名.class

3.this.getClass()。

然后将字节码中的方法，变量，构造函数等映射成相应的Method、Filed、Constructor等类，这些类提供了丰富的方法可以被我们所使用。

深入解析Java反射（1） - 基础

Java基础之—反射（非常重要）

四、泛型 （☆☆）

1、简单介绍一下java中的泛型，泛型擦除以及相关的概念，解析与分派?

泛型是Java SE1.5的新特性，泛型的本质是参数化类型，也就是说所操的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。 Java语言引入泛型的好处是安全简单。

在Java SE 1.5之前，没有泛型的情况的下，通过对类型Object的引用来实现参数的“任意化”，“任意化”带来的缺点是要做显式的强制类型转换，而这种转换是要求开发者实际参数类型可以预知的情况下进行的。对于强制类型换错误的情况，编译器可能不提示错误，在运行的时候出现异常，这是一个安全隐患。

泛型的好处是在编译的时候检查类型安全，并且所有的转换都是自动和隐式的，提高代码的重用率。

1、泛型的类型参数只能是类类型（包括自定义类），不是简单类型。

2、同一种泛型可以对应多个版本（因为参数类型是不确的），不同版本的泛型类实例是不兼容的。

3、泛型的类型参数可以有多个。

4、泛型的参数类型可以使用extends语句，例如。习惯上称为“有界类型”。

5、泛型的参数类型还可以是通配符类型。例如Class<?> classType = Class.forName("java.lang.String");

泛型擦除以及相关的概念

泛型信息只存在代码编译阶段，在进入JVM之前，与泛型关的信息都会被擦除掉。

在类型擦除的时候，如果泛型类里的类型参数没有指定上限，则会被转成Object类型，如果指定了上限，则会被传转换成对应的类型上限。

Java中的泛型基本上都是在编译器这个层次来实现的。生成的Java字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器在编译的时候擦除掉。这个过程就称为类型擦除。

类型擦除引起的问题及解决方法：

1、先检查，在编译，以及检查编译的对象和引用传递的题

2、自动类型转换

3、类型擦除与多态的冲突和解决方法

4、泛型类型变量不能是基本数据类型

5、运行时类型查询

6、异常中使用泛型的问题

7、数组（这个不属于类型擦除引起的问题）

9、类型擦除后的冲突

10、泛型在静态方法和静态类中的问题

五、注解 （☆☆）

1、说说你对Java注解的理解？

注解相当于一种标记，在程序中加了注解就等于为程序打上了某种标记。程序可以利用ava的反射机制来了解你的类及各种元素上有无何种标记，针对不同的标记，就去做相应的事件。标记可以加在包，类，字段，方法，方法的参数以及局部变量上。

六、其它 （☆☆）

1、类的加载过程，Person person = new Person();为例进行说明。

1).因为new用到了Person.class，所以会先找到Person.class文件，并加载到内存中;

2).执行该类中的static代码块，如果有的话，给Person.class类进行初始化;

3).在堆内存中开辟空间分配内存地址;

4).在堆内存中建立对象的特有属性，并进行默认初始化;

5).对属性进行显示初始化;

6).对对象进行构造代码块初始化;

7).对对象进行与之对应的构造函数进行初始化;

8).将内存地址付给栈内存中的p变量。

2、JAVA常量池

Interger中的128(-128~127)

a.当数值范围为-128~127时：如果两个new出来的Integer对象，即使值相同，通过“==”比较结果为false，但两个对直接赋值，则通过“==”比较结果为“true，这一点与String非常相似。

b.当数值不在-128~127时，无论通过哪种方式，即使两对象的值相等，通过“==”比较，其结果为false；

c.当一个Integer对象直接与一个int基本数据类型通过“==”比较，其结果与第一点相同；

d.Integer对象的hash值为数值本身；

为什么是-128-127?

在Integer类中有一个静态内部类IntegerCache，在IntegrCache类中有一个Integer数组，用以缓存当前数值范围为-128~127时的Integer对象。

3、在重写equals方法时，需要遵循哪些约定，具体介绍一下?

重写equals方法时需要遵循通用约定：自反性、对称性、传递性、一致性、非空性

1）自反性

对于任何非null的引用值x,x.equals(x)必须返回true。---这一点基本上不会有啥问题

2）对称性

对于任何非null的引用值x和y，当且仅当x.equals(y)为true时，y.equals(x)也为true。

3）传递性

对于任何非null的引用值x、y、z。如果x.equals(y)==true,y.equals(z)==true,那么x.equals(z)==true。

4） 一致性

对于任何非null的引用值x和y，只要equals的比较操作在对象所用的信息没有被修改，那么多次调用x.equals(y)就会一致性地返回true,或者一致性的返回false。

5）非空性

所有比较的对象都不能为空。

4、Java中对异常是如何进行分类的?

异常整体分类：

Java异常结构中定义有Throwable类。 Exception和Error为其子类。

Error是程序无法处理的错误，比如OutOfMemoryError、StackOverflowError。这些异常发生时， Java虚拟机（JVM）一般会选择线程终止。

Exception是程序本身可以处理的异常，这种异常分两大类运行时异常和非运行时异常，程序中应当尽可能去处理这些异常。

运行时异常都是RuntimeException类及其子类异常，如NullPointerException、IndexOutOfBoundsException等， 这些异常是不检查异常，程序中可以选择捕获处理，也可以不处理。这些异常一般是由程序逻辑错误引起的， 程序应该从逻辑角度尽可能避免这类异常的发生。

异常处理的两个基本原则:

1、尽量不要捕获类似 Exception 这样的通用异常，而是应该捕获特定异常。

2、不要生吞异常。

NoClassDefFoundError 和 ClassNotFoundException 有什么区别?

ClassNotFoundException的产生原因主要是： Java支持使用反射方式在运行时动态加载类，例如使用Class.forName方法来动态地加载类时，可以将类名作为参数传递给上述方法从而将指定类加载到JVM内存中，如果这个类在类路径中没有被找到，那么此时就会在运行时抛出ClassNotFoundException异常。 解决该问题需要确保所需的类连同它依赖的包存在于类路径中，常见问题在于类名书写错误。 另外还有一个导致ClassNotFoundException的原因就是：当一个类已经某个类加载器加载到内存中了，此时另一个类加载器又尝试着动态地从同一个包中加载这个类。通过控制动态类加载过程，可以避免上述情况发生。

NoClassDefFoundError产生的原因在于： 如果JVM或者ClassLoader实例尝试加载（可以通过正常的方法调用，也可能是使用new来创建新的对象）类的时候却找不到类的定义。要查找的类在编译的时候是存在的，运行的时候却找不到了。这个时候就会导致NoClassDefFoundError. 造成该问题的原因可能是打包过程漏掉了部分类，或者jar包出现损坏或者篡改。解决这个问题的办法是查找那些在开发期间存在于类路径下但在运行期间却不在类路径下的类。

5、String 为什么要设计成不可变的?

String是不可变的（修改String时，不会在原有的内存地址修改，而是重新指向一个新对象），String用final修饰，不可继承，String本质上是个final的char[]数组，所以char[]数组的内存地址不会被修改，而且String 也没有对外暴露修改char[]数组的方法。不可变性可以保证线程安全以及字符串串常量池的实现。

6、Java里的幂等性了解吗?

幂等性原本是数学上的一个概念，即：f(x) = f(f(x))，对同一个系统，使用同样的条件，一次请求和重复的多次请求对系统资源的影响是一致的。

幂等性最为常见的应用就是电商的客户付款，试想一下如果你在付款的时候因为网络等各种问题失败了，然后去重复的付了一次，是一种多么糟糕的体验。幂等性就是为了解决这样的问题。

实现幂等性可以使用Token机制。

核心思想是为每一次操作生成一个唯一性的凭证，也就是token。一个token在操作的每一个阶段只有一次执行权，一旦执行成功则保存执行结果。对重复的请求，返回同一个结果。

例如：电商平台上的订单id就是最适合的token。当用户下单时，会经历多个环节，比如生成订单，减库存，减优惠券等等。每一个环节执行时都先检测一下该订单id是否已经执行过这一步骤，对未执行的请求，执行操作并缓存结果，而对已经执行过的id，则直接返回之前的执行结果，不做任何操 作。这样可以在最大程度上避免操作的重复执行问题，缓存起来的执行结果也能用于事务的控制等。

7、为什么Java里的匿名内部类只能访问final修饰的外部变量?

匿名内部类用法：

```
public class TryUsingAnonymousClass {
    public void useMyInterface() {
        final Integer number = 123;
        System.out.println(number);

        MyInterface myInterface = new MyInterface() {
            @Override
            public void doSomething() {
                System.out.println(number);
            }
        };
        myInterface.doSomething();

        System.out.println(number);
    }
}
```

编译后的结果

```
class TryUsingAnonymousClass$1
        implements MyInterface {
    private final TryUsingAnonymousClass this$0;
    private final Integer paramInteger;

    TryUsingAnonymousClass$1(TryUsingAnonymousClass this$0, Integer
paramInteger) {
        this.this$0 = this$0;
        this.paramInteger = paramInteger;
    }

    public void doSomething() {
        System.out.println(this.paramInteger);
    }
}
```

因为匿名内部类最终会编译成一个单独的类，而被该类使用的变量会以构造函数参数的形式传递给该类，例如：Integer paramInteger，如果变量不定义成final的，paramInteger在匿名内部类被可以被修改，进而造成和外部的paramInteger不一致的问题，为了避免这种不一致的情况，因次Java规定匿名内部类只能访问final修饰的外部变量。

8、讲一下Java的编码方式?

为什么需要编码

计算机存储信息的最小单元是一个字节即8bit，所以能示的范围是0~255，这个范围无法保存所有的字符，所以要一个新的数据结构char来表示这些字符，从char到byte需要编码。

常见的编码方式有以下几种：

ASCII：总共有 128 个，用一个字节的低 7 位表示，031 是控制字符如换行回车删除等；32126 是打印字符，可以通过键盘输入并且能够显示出来。

GBK：码范围是 8140~FEFE（去掉 XX7F）总共有 23940 个码位，它能表示 21003 个汉字，它的编码是和 GB2312 兼容的，也就是说用 GB2312 编码的汉字可以用 GBK 来解码，并且不会有乱码。

UTF-16：UTF-16 具体定义了 Unicode 字符在计算机中存取方法。UTF-16 用两个字节来表示 Unicode 转化格式，这个是定长的表示方法，不论什么字符都可以用两个字节表示，两个字节是 16 个 bit，所以叫 UTF-16。UTF-16 表示字符非常方便，每两个字节表示一个字符，这个在字符串操作时就大大简化了操作，这也是 Java 以 UTF-16 作为内存的字符存储格式的一个很重要的原因。

UTF-8：统一采用两个字节表示一个字符，虽然在表示上非常简单方便，但是也有其缺点，有很大一部分字符用一个字节就可以表示的现在要两个字节表示，存储空间放大了一倍，在现在的网络带宽还非常有限的今天，这样会增大网络传输的流量，而且也没必要。而 UTF-8 采用了一种变长技术，每个编码区域有不同的字码长度。不同类型的字符可以是由 1~6 个字节组成。

Java中需要编码的地方一般都在字符到字节的转换上，这个一般包括磁盘IO和网络IO。

Reader 类是 Java 的 I/O 中读字符的父类，而InputStream 类是读字节的父类，InputStreamReader类就是关联字节到字符的桥梁，它负责在 I/O 过程中处理读取字节到字符的转换，而具体字节到字符解码实现由 StreamDecoder 去实现，在 StreamDecoder 解码过程中必须由用户指定 Charset 编码格式。

9、String，StringBuffer，StringBuilder有哪些不同?

三者在执行速度方面的比较：StringBuilder > StringBuffer > String

String每次变化一个值就会开辟一个新的内存空间

StringBuilder：线程非安全的

StringBuffer：线程安全的

对于三者使用的总结：

1.如果要操作少量的数据用 String。

2.单线程操作字符串缓冲区下操作大量数据用 StringBuilder。

3.多线程操作字符串缓冲区下操作大量数据用 StringBuffer。

String 是 Java 语言非常基础和重要的类，提供了构造和管理字符串的各种基本逻辑。它是典型的 Immutable 类，被声明成为 final class，所有属性也都是 final 的。也由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。由于字符串操作的普遍性，所以相关操作的效率往往对应用性能有明显影响。

StringBuffer 是为解决上面提到拼接产生太多中间对象的问题而提供的一个类，我们可以用 append 或者 add 方法，把字符串添加到已有序列的末尾或者指定位置。StringBuffer 本质是一个线程安全的可修改字符序列，它保证了线程安全，也随之带来了额外的性能开销，所以除非有线程安全的需要，不然还是推荐使用它的后继者，也就是 StringBuilder。

StringBuilder 是 Java 1.5 中新增的，在能力上和 StringBuffer 没有本质区别，但是它去掉了线程安全的部分，有效减小了开销，是绝大部分情况下进行字符串拼接的首选。

10、什么是内部类？内部类的作用。

内部类可以有多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。

在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

创建内部类对象并不依赖于外围类对象的创建。

内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

内部类提供了更好的封装，除了该外围类，其他类都不能访问。。

11、抽象类和接口区别？

共同点

- 是上层的抽象层。
- 都不能被实例化。

- 都能包含抽象的方法，这些抽象的方法用于描述类具备的功能，但是不提供具体的实现。

区别：

- 1、在抽象类中可以写非抽象的方法，从而避免在子类中重复书写他们，这样可以提高代码的复用性，这是抽象类的优势，接口中只能有抽象的方法。
- 2、多继承：一个类只能继承一个直接父类，这个父类可以是具体的类也可是抽象类，但是一个类可以实现多个接口。
- 3、抽象类可以有默认的方法实现，接口根本不存在方法的实现。
- 4、子类使用extends关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明方法的实现。子类使用关键字implements来实现接口。它需要提供接口中所有声明方法的实现。
- 5、构造器：抽象类可以有构造器，接口不能有构造器。
- 6、和普通Java类的区别：除了你不能实例化抽象类之外，抽象类和普通Java类没有任何区别，接口是完全不同的类型。
- 7、访问修饰符:抽象方法可以有public、protected和default修饰符，接口方法默认修饰符是public。你不可以使用其它修饰符。
- 8、main方法:抽象方法可以有main方法并且我们可以运行它接口没有main方法，因此我们不能运行它。
- 9、速度:抽象类比接口速度要快，接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
- 10、添加新方法:如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。如果你往接口中添加方法，那么你必须改变实现该接口的类。

12、接口的意义?

规范、扩展、回调。

13、父类的静态方法能否被子类重写?

不能。子类继承父类后，用相同的静态方法和非静态方法，这时非静态方法覆盖父类中的方法（即方法重写），父类的该静态方法被隐藏（如果对象是父类则调用该隐藏的方法），另外子类可继承父类的静态与非静态方法，至于方法重载我觉得它其中一要素就是在同一类中，不能说父类中的什么方法与子类里的什么方法是方法重载的体现。

14、抽象类的意义?

为其子类提供一个公共的类型，封装子类中的重复内容，定义抽象方法，子类虽然有不同的实现 但是定义是一致的。

15、静态内部类、非静态内部类的理解?

静态内部类：只是为了降低包的深度，方便类的使用，静态内部类适用于包含在类当中，但又不依赖与外在的类，不用使用外在类的非静态属性和方法，只是为了方便管理类结构而定义。在创建静态内部类的时候，不需要外部类对象的引用。

非静态内部类：持有外部类的引用，可以自由使用外部类的所有变量和方法。

16、为什么复写equals方法的同时需要复写hashcode方法，前者相同后者是否相同，反过来呢？为什么？

要考虑到类似HashMap、HashTable、HashSet的这种散列的数据类型的运用，当我们重写equals时，是为了用自身的方式去判断两个自定义对象是否相等，然而如果此时刚好需要我们用自定义的对象去充当hashmap的键值使用时，就会出现我们认为的同一对象，却因为hash值不同而导致hashmap中存了两个对象，从而才需要进行hashcode方法的覆盖。

17、equals 和 hashcode 的关系？

hashcode和equals的约定关系如下：

- 1、如果两个对象相等，那么他们一定有相同的哈希值（hashcode）。
- 2、如果两个对象的哈希值相等，那么这两个对象有可能相等也有可能不相等。（需要再通过equals来判断）

18、java为什么跨平台？

因为Java程序编译之后的代码不是能被硬件系统直接运行的代码，而是一种“中间码”——字节码。然后不同的硬件平台上安装有不同的Java虚拟机(JVM)，由JVM来把字节码再“翻译”成所对应的硬件平台能够执行的代码。因此对于Java编程者来说，不需要考虑硬件平台是什么。所以Java可以跨平台。

19、浮点数的精准计算

BigDecimal类进行商业计算，Float和Double只能用来做科学计算或者是工程计算。

20、final，finally，finalize的区别？

final 可以用来修饰类、方法、变量，分别有不同的意义，final 修饰的 class 代表不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。

finally 则是 Java 保证重点代码一定要被执行的一种机制。我们可以使用 try-finally 或者 try-catch-finally 来进行类似关闭 JDBC 连接、保证 unlock 锁等动作。

finalize 是基础类 java.lang.Object 的一个方法，它的设计目的是保证对象在被垃圾收集前完成特定资源的回收。finalize 机制现在已经不推荐使用，并且在 JDK 9 开始被标记为 deprecated。Java 平台目前在逐步使用 java.lang.ref.Cleaner 来替换掉原有的 finalize 实现。Cleaner 的实现利用了幻象引用（PhantomReference），这是一种常见的所谓 post-mortem 清理机制。利用幻象引用和引用队列，我们可以保证对象被彻底销毁前做一些类似资源回收的工作，比如关闭文件描述符（操作系统有限的资源），它比 finalize 更加轻量、更加可靠。

21、静态内部类的设计意图

静态内部类与非静态内部类之间存在一个最大的区别：非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。

没有这个引用就意味着：

它的创建是不需要依赖于外围类的。 它不能使用任何外围类的非static成员变量和方法。

22、Java中对象的生命周期

在Java中，对象的生命周期包括以下几个阶段：

1.创建阶段(Created)

JVM 加载类的class文件 此时所有的static变量和static代码块将被执行 加载完成后，对局部变量进行赋值（先父后子的顺序） 再执行new方法 调用构造函数 一旦对象被创建，并被分派给某些变量赋值，这个对象的状态就切换到了应用阶段。

2.应用阶段(In Use)

对象至少被一个强引用持有着。

3.不可见阶段(Invisible)

当一个对象处于不可见阶段时，说明程序本身不再持有该对象的任何强引用，虽然该这些引用仍然是存在着的。 简单说就是程序的执行已经超出了该对象的作用域了。

4.不可达阶段(Unreachable)

对象处于不可达阶段是指该对象不再被任何强引用所持有。 与“不可见阶段”相比，“不可见阶段”是指程序不再持有该对象的任何强引用，这种情况下，该对象仍可能被JVM等系统下的某些已装载的静态变量或线程或JNI等强引用持有着，这些特殊的强引用被称为“GC root”。存在着这些GC root会导致对象的内存泄露情况，无法被回收。

5.收集阶段(Collected)

当垃圾回收器发现该对象已经处于“不可达阶段”并且垃圾回收器已经对该对象的内存空间重新分配做好准备时，则对象进入了“收集阶段”。如果该对象已经重写了finalize()方法，则会去执行该方法的终端操作。

6.终结阶段(Finalized)

当对象执行完finalize()方法后仍然处于不可达状态时，则该对象进入终结阶段。在该阶段是等待垃圾回收器对该对象空间进行回收。

7.对象空间重分配阶段(De-allocated)

垃圾回收器对该对象的所占用的内存空间进行回收或者再分配了，则该对象彻底消失了，称之为“对象空间重新分配阶段。

23、静态属性和静态方法是否可以被继承？是否可以被重写？以及原因?

结论：java中静态属性和静态方法可以被继承，但是不可以被重写而是被隐藏。

原因：

1). 静态方法和属性是属于类的，调用的时候直接通过类名.方法名完成，不需要继承机制即可以调用。如果子类里面定义了静态方法和属性，那么这时候父类的静态方法或属性称之为"隐藏"。如果你想要调用父类的静态方法和属性，直接通过父类名.方法或变量名完成，至于是否继承一说，子类是有继承静态方法和属性，但是跟实例方法和属性不太一样，存在"隐藏"的这种情况。

2). 多态之所以能够实现依赖于继承、接口和重写、重载（继承和重写最为关键）。有了继承和重写就可以实现父类的引用指向不同子类的对象。重写的功能是："重写"后子类的优先级要高于父类的优先级，但是“隐藏”是没有这个优先级之分的。

3). 静态属性、静态方法和非静态的属性都可以被继承和隐藏而不能被重写，因此不能实现多态，不能实现父类的引用可以指向不同子类的对象。非静态方法可以被继承和重写，因此可以实现多态。

24、object类的equal 和hashcode 方法重写，为什么?

在Java API文档中关于hashCode方法有以下几点规定（原文来自java深入解析一书）：

1、在java应用程序执行期间，如果在equals方法比较中所用的信息没有被修改，那么在同一个对象上多次调用hashCode方法时必须一致地返回相同的整数。如果多次执行同一个应用时，不要求该整数必须相同。

2、如果两个对象通过调用equals方法是相等的，那么这两个对象调用hashCode方法必须返回相同的整数。

3、如果两个对象通过调用equals方法是不相等的，不要求这两个对象调用hashCode方法必须返回不同的整数。但是程序员应该意识到对不同的对象产生不同的hash值可以提供哈希表的性能。

25、java中==和equals和hashCode的区别?

默认情况下也就是从超类Object继承而来的equals方法与'=='是完全等价的，比较的都是对象的内存地址，但我们可以重写equals方法，使其按照我们的需求的方式进行比较，如String类重写了equals方法，使其比较的是字符的序列，而不再是内存地址。在java的集合中，判断两个对象是否相等的规则是：

```
1.判断两个对象的hashCode是否相等。
2.判断两个对象用equals运算是否相等。
```

50、Java的四种引用及使用场景?

- 强引用（FinalReference）：在内存不足时不会被回收。平常用的最多的对象，如新创建的对象。
- 软引用（SoftReference）：在内存不足时会被回收。用于实现内存敏感的高速缓存。
- 弱引用（WeakReferenc）：只要GC回收器发现了它，就会将之回收。用于Map数据结构中，引用占用内存空间较大的对象。
- 虚引用（PhantomReference）：在回收之前，会被放入ReferenceQueue，JVM不会自动将该referent字段值设置成null。其它引用被JVM回收之后才会被放入ReferenceQueue中。用于实现一个对象被回收之前做一些清理工作。

26、简单谈谈堆外内存以及你的理解和认识。

27、怎么理解栈、堆？堆中存什么？栈中存什么?

28、为什么要把堆和栈区分出来呢？栈中不是也可以存储数据吗?

29、在Java中，什么是是栈的起始点，同是也是程序的起始点?

30、为什么不把基本类型放堆中呢?

31、Java中的参数传递是传值呢？还是传引用?

32、Java中有没有指针的概念?

33、Java中，栈的大小通过什么参数来设置?

34、一个空Object对象的占多大空间?

35、对象引用类型分为哪几类?

36、java里带$的函数见过么，是什么意思

37、为什么jdk8用metaspace数据结构用来替代perm?

38、修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法；

39、静态内部类的设计意图。

40、int、char、long各占多少字节数?

41、int与integer的区别

42、抽象类与接口的应用场景

43、抽象类是否可以没有方法和属性?

44、string 转换成 integer的方式及原理

45、讲一下常见编码方式?

46、utf-8编码中的中文占几个字节，int型几个字节?

47、如何将一个Java对象序列化到文件里?

48、Java 中内部类为什么可以访问外部类?

49、为什么java 7中不能用lambda

50、clone()的默认实现是深拷贝还是浅拷贝?如何让clone()实现深拷贝?

51、Class文件结构（常量池）。

52、运行时栈帧结构（主要是局部变量表，理解栈堆）。

53、Java重排序和顺序一致性。（as-if-serial和happens-before）

54、java 7 8 9 10的区别

55、int,long的取值范围以及BigDecimal，数值越界了如何处理?

56、注解如何获取，反射为何耗性能?

57、Integer类对int的优化

58、深拷贝和浅拷贝的区别