目录

Java 面试

Java 基础

集合

1、List 和 Set 的区别

Java 中的集合包括三大类，它们是 Set（集）、List（列表）和 Map（映射），它们都处于 java.util 包中，Set、List 和 Map 都是接口，它们有各自的实现类。Set 的实现类主要有 HashSet 和 TreeSet，List 的实现类主要有 ArrayList

Collection 是最基本的集合接口，声明了适用于 JAVA 集合的通用方法，list 和 set 都继承自 collection 接口。

Collection 接口的方法：

boolean add(Object o)　　：向集合中加入一个对象的引用

void clear()：删除集合中所有的对象，即不再持有这些对象的引用

boolean isEmpty()　：判断集合是否为空

boolean contains(Object o) ：　判断集合中是否持有特定对象的引用

Iterartor iterator()　：返回一个 Iterator 对象，可以用来遍历集合中的元素

boolean remove(Object o) ：从集合中删除一个对象的引用

int size()　　：返回集合中元素的数目

Object[] toArray()　：　返回一个数组，该数组中包括集合中的所有元素

关于：Iterator()　和 toArray()　方法都用于集合的所有的元素，前者返回一个 Iterator 对象，后者返回一个包含集合中所有元素的数组。

Collection 没有 get()方法来取得某个元素。只能通过 iterator()遍历元素。

List 的功能方法：

实际上有两种 List：一种是基本的 ArrayList,其优点在于随机访问元素，另一种是更强大的 LinkedList,它并不是为快速随机访问设计的，而是具有一套更通用的方法。

List：次序是 List 最重要的特点：它保证维护元素特定的顺序。List 为 Collection 添加了许多方法，使得能够向 List 中间插入与移除元素(这只推　荐 LinkedList 使用。)一个 List 可以生成 ListIterator,使用它可以从两个方向遍历 List,也可以从 List 中间插入和移除元素。

ArrayList：由数组实现的 List。允许对元素进行快速随机访问，但是向 List 中间插入与移除元素的速度很慢。ListIterator 只应该用来由后向前遍历　ArrayList,而不是用来插入和移除元素。因为那比 LinkedList 开销要大很多。

LinkedList　：对顺序访问进行了优化，向 List 中间插入与删除的开销并不大。随机访问则相对较慢。(使用 ArrayList 代替。)还具有下列方　法：addFirst(), addLast(), getFirst(), getLast(), removeFirst()　和　removeLast(), 这些方法　(没有在任何接口或基类中定义过)使得 LinkedList 可以当作堆栈、队列和双向队列使用。

Set 的功能方法：

Set 具有与 Collection 完全一样的接口，因此没有任何额外的功能。实际上 Set 就是 Collection,只　是行为不同。这是继承与多态思想的典型应用：表现不同的行为。Set 不保存重复的元素(至于如何判断元素相同则较为负责)

Set：存入 Set 的每个元素都必须是唯一的，因为 Set 不保存重复元素。加入 Set 的元素必须定义 equals()方法以确保对象的唯一性。Set 与 Collection 有完全一样的接口。Set 接口不保证维护元素的次序。

HashSet：为快速查找设计的 Set。存入 HashSet 的对象必须定义 hashCode()。

TreeSet： 保存次序的 Set, 底层为树结构。使用它可以从 Set 中提取有序的序列。

LinkedHashSet：具有 HashSet 的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是在使用迭代器遍历 Set 时，结果会按元素插入的次序显示。

list 与 Set 区别：

1、List,Set 都是继承自 Collection 接口

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，重复元素会覆盖掉，（元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的 HashCode 决定的，其位置其实是固定的，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，也可以用迭代器，但是 set 只能用迭代，因为他无序，无法用下标来取得想要的值。）

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。

2、HashSet 是如何保证不重复的

弄清怎么个逻辑达到元素不重复的，源码先上

HashSet 类中的 add()方法：

```
public boolean add(E e) {
      return map.put(e, PRESENT)==null;
   }
```

类中 map 和 PARENT 的定义：

```
private transient HashMap<E,Object> map;
 // Dummy value to associate with an Object in the backing Map 用来匹配 Map 中后面的对象的一个虚拟值
private static final Object PRESENT = new Object();
```

实际上 HashSet 如何保证不重复就是 HashMap 如何保证不重复。

HashMap 的 put 方法：

```
public V put(K key, V value) {
        if (key == null)
            return putForNullKey(value);
        int hash = hash(key.hashCode());
        int i = indexFor(hash, table.length);
        for (Entry<K,V> e = table[i]; e != null; e = e.next) {
            Object k;
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;
                e.value = value;
                e.recordAccess(this);
                return oldValue;
```

```
                }
            }
            modCount++;
            addEntry(hash, key, value, i);
            return null;
        }
```

最重要的就是： if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

可以看到 for 循环中，遍历 table 中的元素，

1，如果 hash 码值不相同，说明是一个新元素，存；

如果没有元素和传入对象（也就是 add 的元素）的 hash 值相等，那么就认为这个元素在 table 中不存在，将其添加进 table；

2（1），如果 hash 码值相同，且 equles 判断相等，说明元素已经存在，不存；

2（2），如果 hash 码值相同，且 equles 判断不相等，说明元素不存在，存；

如果有元素和传入对象的 hash 值相等，那么，继续进行 equles()判断，如果仍然相等，那么就认为传入元素已经存在，不再添加，结束，否则仍然添加。

Hashcode 和 equals 重写的方法：

```
public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + age;
            result = prime * result + ((name == null) ? 0 : name.hashCode());
            return result;
        }
public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Person other = (Person) obj;
            if (age != other.age)
                return false;
            if (name == null) {
                if (other.name != null)
                    return false;
            } else if (!name.equals(other.name))
                return false;
            return true;
        }
```

3、HashMap 是线程安全的吗，为什么不是线程安全的

一直以来都知道 HashMap 是线程不安全的，但是到底为什么线程不安全，在多线程操作情况下什么时候线程不安全？

让我们先来了解一下 HashMap 的底层存储结构，HashMap 底层是一个 Entry 数组，一旦发生 Hash 冲突的的时候，HashMap 采用拉链法解决碰撞冲突，Entry 内部的变量：

```
final Object key;
Object value;
Entry next;
int hash;
```

通过 Entry 内部的 next 变量可以知道使用的是链表，这时候我们可以知道，如果多个线程，在某一时刻同时操作 HashMap 并执行 put 操作，而有大于两个 key 的 hash 值相同，如图中 a1、a2，这个时候需要解决碰撞冲突，而解决冲突的办法上面已经说过，对于链表的结构在这里不再赘述，暂且不讨论是从链表头部插入还是从尾部初入，这个时候两个线程如果恰好都取到了对应位置的头结点 e1，而最终的结果可想而知，a1、a2 两个数据中势必会有一个会丢失。

再来看下 put 方法：

```
public Object put(Object obj, Object obj1)
    {
        if(table == EMPTY_TABLE)
            inflateTable(threshold);
        if(obj == null)
            return putForNullKey(obj1);
        int i = hash(obj);
        int j = indexFor(i, table.length);
        for(Entry entry = table[j]; entry != null; entry = entry.next)
        {
            Object obj2;
            if(entry.hash == i && ((obj2 = entry.key) == obj ||
obj.equals(obj2)))
            {
                Object obj3 = entry.value;
                entry.value = obj1;
                entry.recordAccess(this);
                return obj3;
            }
        }
```

```
            modCount++;
            addEntry(i, obj, obj1, j);
            return null;
        }
```

put 方法不是同步的，同时调用了 addEntry 方法：

```
    void addEntry(int i, Object obj, Object obj1, int j)
    {
        if(size >= threshold && null != table[j])
        {
            resize(2 * table.length);
            i = null == obj ? 0 : hash(obj);
            j = indexFor(i, table.length);
        }
        createEntry(i, obj, obj1, j);
    }
```

addEntry 方法依然不是同步的，所以导致了线程不安全出现伤处问题，其他类似操作不再说明，源码一看便知，下面主要说一下另一个非常重要的知识点，同样也是 HashMap 非线程安全的原因，我们知道在 HashMap 存在扩容的情况，对应的方法为 HashMap 中的 resize 方法：

```
    void resize(int i)
        {
            Entry aentry[] = table;
            int j = aentry.length;
            if(j == 1073741824)
            {
                threshold = 2147483647;
                return;
            } else
            {
                Entry aentry1[] = new Entry[i];
                transfer(aentry1, initHashSeedAsNeeded(i));
```

```
                table = aentry1;
                threshold = (int)Math.min((float)i * loadFactor,
1.073742E+009F);
                return;
            }
        }
```

可以看到扩容方法也不是同步的，通过代码我们知道在扩容过程中，会新生成一个新的容量的数组，然后对原数组的所有键值对重新进行计算和写入新的数组，之后指向新生成的数组。

当多个线程同时检测到总数量超过门限值的时候就会同时调用 resize 操作，各自生成新的数组并 rehash 后赋给该 map 底层的数组 table，结果最终只有最后一个线程生成的新数组被赋给 table 变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的 table 作为原始数组，这样也会有问题。

4、HashMap 的扩容过程

5、HashMap 1.7 与 1.8 的 区别，说明 1.8 做了哪些优化，如何优化的？

（一）Hashmap 的结构，1.7 和 1.8 有哪些区别

不同点：

（1）JDK1.7 用的是头插法，而 JDK1.8 及之后使用的都是尾插法，那么他们为什么要这样做呢？因为 JDK1.7 是用单链表进行的纵向延伸，当采用头插法时会容易出现逆序且环形链表死循环问题。但是在 JDK1.8 之后是因为加入了红黑树使用尾插法，能够避免出现逆序且链表死循环的问题。

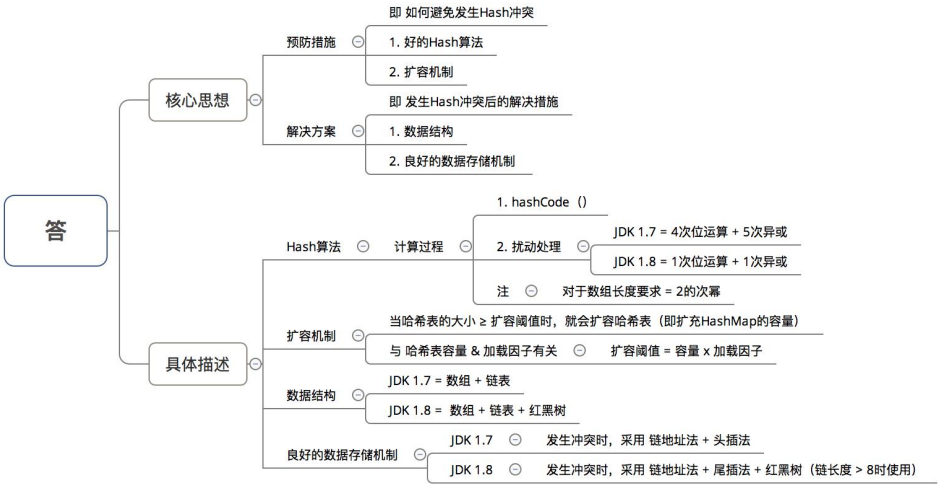
（2）扩容后数据存储位置的计算方式也不一样：

1. 在 JDK1.7 的时候是直接用 hash 值和需要扩容的二进制数进行&（这里就是为什么扩容的时候为啥一定必须是 2 的多少次幂的原因所在，因为如果只有 2 的 n 次幂的情况时最后一位二进制数才一定是 1，这样能最大程度减少 hash 碰撞）（hash 值 & length-1）

2、而在 JDK1.8 的时候直接用了 JDK1.7 的时候计算的规律，也就是扩容前的原始位置+扩容的大小值=JDK1.8 的计算方式，而不再是 JDK1.7 的那种异或的方法。但是这种方式就相当于只需要判断 Hash 值的新增参与运算的位是 0 还是 1 就直接迅速计算出了扩容后的储存方式。

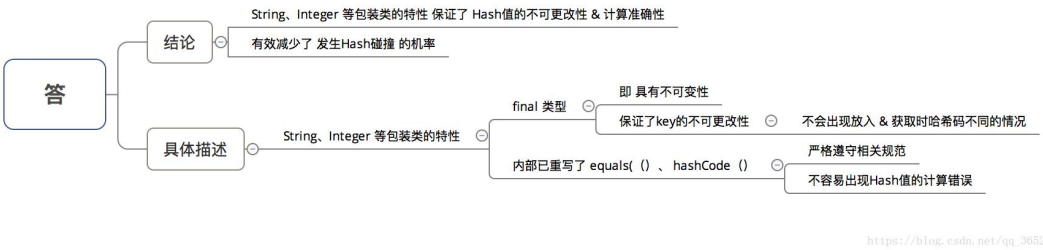
（3）JDK1.7 的时候使用的是数组+ 单链表的数据结构。但是在 JDK1.8 及之后时，使用的是数组+链表+红黑树的数据结构（当链表的深度达到 8 的时候，也就是默认阈值，就会自动扩容把链表转成红黑树的数据结构来把时间复杂度从 O（n）变成 O（logN）提高了效率）

（二）哈希表如何解决 Hash 冲突？

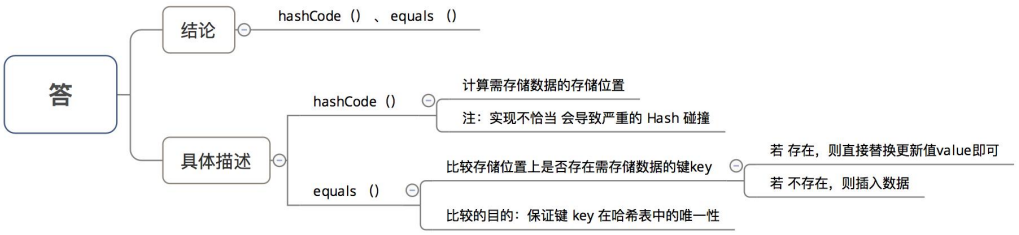


（三）为什么 HashMap 具备下述特点：键-值（key-value）都允许为空、线程不安全、不保证有序、存储位置随时间变化

（四）为什么 HashMap 中 String、Integer 这样的包装类适合作为 key 键



（五）HashMap 中的 key 若 Object 类型， 则需实现哪些方法？



6、final finally finalize

1.简单区别：

final 用于声明属性，方法和类，分别表示属性不可交变，方法不可覆盖，类不可继承。

finally 是异常处理语句结构的一部分，表示总是执行。

finalize 是 Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，供垃圾收集时的其他资源回收，例如关闭文件等。

2.中等区别：

虽然这个单词在 Java 中都存在，但是并没太多关联：

final：java 中的关键字，修饰符。

A).如果一个类被声明为 final，就意味着它不能再派生出新的子类，不能作为父类被继承。因此，一个类不能同时被声明为 abstract 抽象类的和 final 的类。

B).如果将变量或者方法声明为 final，可以保证它们在使用中不被改变.

1)被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。

2)被声明 final 的方法只能使用，不能重载。

finally：java 的一种异常处理机制。

finally 是对 Java 异常处理模型的最佳补充。finally 结构使代码总会执行，而不管无异常发生。使用 finally 可以维护对象的内部状态，并可以清理非内存资源。特别是在关闭数据库连接这方面，如果程序员把数据库连接的 close()方法放到 finally 中，就会大大降低程序出错的几率。

finalize：Java 中的一个方法名。

Java 技术使用 finalize()方法在垃圾收集器将对象从内存中清除出去前，做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没被引用时对这个对象调用的。它是在 Object 类中定义的，因此所的类都继承了它。子类覆盖 finalize()方法以整理系统资源或者执行其他清理工作。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

3.详细区别：

这是一道再经典不过的面试题了，我们在各个公司的面试题中几乎都能看到它的身影。final、finally 和 finalize 虽然长得像孪生兄弟一样，但是它们的含义和用法却是大相径庭。

final 关键字我们首先来说说 final。它可以用于以下四个地方:

1).定义变量，包括静态的和非静态的。

2).定义方法的参数。

3).定义方法。

4).定义类。

定义变量，包括静态的和非静态的。定义方法的参数

第一种情况：

如果 final 修饰的是一个基本类型，就表示这个变量被赋予的值是不可变的，即它是个常量；

如果 final 修饰的是一个对象，就表示这个变量被赋予的引用是不可变的

这里需要提醒大家注意的是，不可改变的只是这个变量所保存的引用，并不是这个引用所指向的对象。

第二种情况：final 的含义与第一种情况相同。

实际上对于前两种情况，一种更贴切的表述 final 的含义的描述，那就是，如果一个变量或方法参数被 final 修饰，就表示它只能被赋值一次，但是 JAVA 虚拟机为变量设定的默认值不记作一次赋值。被 final 修饰的变量必须被初始化。初始化的方式以下几种：

1.在定义的时候初始化。

2.final 变量可以在初始化块中初始化，不可以在静态初始化块中初始化。

3.静态 final 变量可以在定义时初始化，也可以在静态初始化块中初始化，不可以在初始化块中初始化。

4.final 变量还可以在类的构造器中初始化，但是静态 final 变量不可以。

我们运行上面的代码之后出了可以发现 final 变量（常量和静态 final 变量（静态常量被初始化时，编译会报错。

用 final 修饰的变量（常量比非 final 的变量（普通变量拥更高的效率，因此我们在际编程中应该尽可能多的用常量来代替普通变量。

定义方法

当 final 用来定义一个方法时，它表示这个方法不可以被子类重写，但是并不影响它被子类继承。

这里需要特殊说明的是，具有 private 访问权限的方法也可以增加 final 修饰，但是由于子类无法继承 private 方法，因此也无法重写它。编译器在处理 private 方法时，是照 final 方来对待的，这样可以提高该方法被调用时的效率。不过子类仍然可以定义同父类中 private 方法具同样结构的方法，但是这并不会产生重写的效果，而且它们之间也不存在必然联系。

定义类

最后我们再来回顾一下 final 用于类的情况。这个大家应该也很熟悉了，因为我们最常用的 String 类就是 final 的。由于 final 类不允许被继承，编译器在处理时把它的所方法都当作 final 的，因此 final 类比普通类拥更高的效率。而由关键字 abstract 定义的抽象类含必须由继承自它的子类重载实现的抽象方法，因此无法同时用 final 和 abstract 来修饰同一个类。同样的道理，

final 也不能用来修饰接口。 final 的类的所方法都不能被重写，但这并不表示 final 的类的属性（变量值也是不可改变的，要想做到 final 类的属性值不可改变，必须给它增加 final 修饰。

finally 语句

接下来我们一起回顾一下 finally 的用法。finally 只能用在 try/catch 语句中并且附带着一个语句块，表示这段语句最终总是被执行。

运行结果说明了 finally 的作用：

1.程序抛出了异常

2.执行了 finally 语句块请大家注意，捕获程序抛出的异常之后，既不加处理，也不继续向上抛出异常，并不是良好的编程习惯，它掩盖了程序执行中发生的错误，这里只是方便演示，请不要学习。

那么，没一种情况使 finally 语句块得不到执行呢？

return、continue、break 这个可以打乱代码顺序执行语句的规律。那我们就来试试看，这个语句是否能影响 finally 语句块的执行：

很明显，return、continue 和 break 都没能阻止 finally 语句块的执行。从输出的结果来看，return 语句似乎在 finally 语句块之前执行了，事实真的如此吗？我们来想想看，return 语句的作用是什么呢？是退出当前的方法，并将值或对象返回。如果 finally 语句块是在 return 语句之后执行的，那么 return 语句被执行后就已经退出当前方法了，finally 语句块又如何能被执行呢？因此，正确的执行顺序应该是这样的：编译器在编译 return new ReturnClass();时，将它分成了两个步骤，new ReturnClass()和 return，前一个创建对象的语句是在 finally 语句块之前被执行的，而后一个 return 语句是在 finally 语句块之后执行的，也就是说 finally 语句块是在程序退出方法之前被执行的。同样，finally 语句块是在循环被跳过（continue 和中断（break 之前被执行的

finalize 方法

最后，我们再来看看 finalize，它是一个方法，属于 java.lang.Object 类，它的定义如下：protected void finalize()throws Throwable{}众所周知，finalize()方法是 GC（garbagecollector 运行机制的一部分,在此我们只说说 finalize()方法的作用是什么呢？finalize()方法是在 GC 清理它所从属的对象时被调用的，如果执行它的过程中抛出了无法捕获的异常（uncaughtexception，GC 将终止对改对象的清理，并且该异常会被忽略；直到下一次 GC 开始清理这个对象时，它的 finalize()会被再次调用。

序调用了 java.lang.System 类的 gc()方法，引起 GC 的执行，GC 在清理 ft 对象时调用了它的 finalize()方法，因此才了上面的输出结果。调用 System.gc()等同于调用下面这行代码：Runtime.getRuntime().gc();调用它们的作用只是建议垃圾收集器（GC 启动，清理无用的对象释放内存空间，但是 GC 的启动并不是一定的，这由 JAVA 虚拟机来决定。直到 JAVA 虚拟机停止运行，些对象的 finalize()可能都没被运行过，那么怎样保证所对象的这个方法在 JAVA 虚拟机停止运行之前一定被调用呢？

给这个方法传入 true 就可以保证对象的 finalize()方法在 JAVA 虚拟机停止运行前一定被运行了，不过遗憾的是这个方法是不安全的，它会导致有用的对象 finalize()被误调用，因此已不被赞成使用了。由于 finalize()属于 Object 类，因此所类都这个方法，Object 的任意子类都可以重写（override 该方法，在其中释放系统资源或者做其它的清理工作，如关闭输入输出流。通过以上知识的回顾，我想大家对于 final、finally、finalize 的用法区别已经很清楚了。

7、强引用 、软引用、 弱引用、虚引用

Java 四种引用包括强引用，软引用，弱引用，虚引用。

强引用：

只要引用存在，垃圾回收器永远不会回收

```
Object obj = new Object();
```

//可直接通过 obj 取得对应的对象 如 obj.equels(new Object());

而这样 obj 对象对后面 new Object 的一个强引用，只有当 obj 这个引用被释放之后，对象才会被释放掉，这也是我们经常所用到的编码形式。

软引用：

非必须引用，内存溢出之前进行回收，可以通过以下代码实现

```
Object obj = new Object();
SoftReference<Object> sf = new SoftReference<Object>(obj);
obj = null;
```

sf.get();//有时候会返回 null

这时候 sf 是对 obj 的一个软引用，通过 sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回 null；

软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

弱引用：

第二次垃圾回收时回收，可以通过如下代码实现

```
Object obj = new Object();
WeakReference<Object> wf = new WeakReference<Object>(obj);
obj = null;
wf.get();//有时候会返回 null
wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾
```

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回 null。

弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的 isEnQueued 方法返回对象是否被垃圾回收器标记。

虚引用：

垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

```
Object obj = new Object();
PhantomReference<Object> pf = new PhantomReference<Object>(obj);
obj=null;
pf.get();//永远返回 null
pf.isEnQueued();//返回是否从内存中已经删除
```

虚引用是每次垃圾回收的时候都会被回收，通过虚引用的 get 方法永远获取到的数据为 null，因此也被成为幽灵引用。

虚引用主要用于检测对象是否已经从内存中删除。

对象的强、软、弱和虚引用

在 JDK 1.2 以前的版本中，若一个对象不被任何变量引用，那么程序就无法再使用这个对象。也就是说，只有对象处于可触及（reachable）状态，程序才能使用它。从 JDK 1.2 版本开始，把对象的引用分为 4 种级别，从而使程序能更加灵活地控制对象的生命周期。这 4 种级别由高到低依次为：强引用、软引用、弱引用和虚引用。

⑴强引用（StrongReference）

强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。 ps：强引用其实也就是我们平时 A a = new A()这个意思。

⑵软引用（SoftReference）

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存（下文给出示例）。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java 虚拟机就会把这个软引用加入到与之关联的引用队列中。

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

```
String str=new String("abc");                                    // 强引用
SoftReference<String> softRef=new SoftReference<String>(str);    // 软引用
```

当内存不足时，等价于：

```
If(JVM.内存不足()) {
    str = null;   // 转换为软引用
    System.gc(); // 垃圾回收器进行回收
}
```

软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出。这时候就可以使用软引用。

⑶弱引用（WeakReference）

弱引用与软引用的区别在于：只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就会把这个弱引用加入到与之关联的引用队列中。

当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。

⑷虚引用（PhantomReference）

“虚引用”顾名思义，就是形同虚设，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

虚引用主要用来跟踪对象被垃圾回收器回收的活动。虚引用与软引用和弱引用的一个区别在于：虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。

```
ReferenceQueue queue = new ReferenceQueue ();
PhantomReference pr = new PhantomReference (object, queue);
```

程序可以通过判断引用队列中是否已经加入了虚引用，来了解被引用的对象是否将要被垃圾回收。如果程序发现某个虚引用已经被加入到引用队列，那么就可以在所引用的对象的内存被回收之前采取必要的行动。

***：

使用 ReferenceQueue 清除失去了软引用对象的 SoftReference

作为一个 Java 对象，SoftReference 对象除了具有保存软引用的特殊性之外，也具有 Java 对象的一般性。所以，当软可及对象被回收之后，虽然这个 SoftReference 对象的 get()方法返回 null,但这个 SoftReference 对象已经不再具有存在的价值，需要一个适当的清除机制，避免大量 SoftReference 对象带来的内存泄漏。在 java.lang.ref 包里还提供了 ReferenceQueue。如果在创建 SoftReference 对象的时候，使用了一个 ReferenceQueue 对象作为参数提供给 SoftReference 的构造方法，如:

```
ReferenceQueue queue = new
ReferenceQueue();
SoftReference
ref=new
SoftReference(aMyObject, queue);
```

那么当这个 SoftReference 所软引用的 aMyOhject 被垃圾收集器回收的同时，ref 所强引用的 SoftReference 对象被列入 ReferenceQueue。也就是说，ReferenceQueue 中保存的对象是 Reference 对象，而且是已经失去了它所软引用的对象的 Reference 对象。另外从 ReferenceQueue 这个名字也可以看出，它是一个队列，当我们调用它的 poll()方法的时候，如果这个队列中不是空队列，那么将返回队列前面的那个 Reference 对象。

在任何时候，我们都可以调用 ReferenceQueue 的 poll()方法来检查是否有它所关心的非强可及对象被回收。如果队列为空，将返回一个 null,否则该方法返回队列中前面的一个 Reference 对象。利用这个方法，我们可以检查哪个 SoftReference 所软引用的对象已经被回收。于是我们可以把这些失去所软引用的对象的 SoftReference 对象清除掉。常用的方式为:

```
SoftReference ref = null;
while ((ref = (EmployeeRef) q.poll()) != null) {
// 清除 ref
}
```

8、Java 反射

1、什么是 Java 类中的反射？

当程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言。我们认为 Java 并不是动态语言，但是它却又一个非常突出的动态相关的机制，俗称：反射。

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序获取自身的信息，并且可以操作类和对象的内部属性。

通过反射，我们可以在运行时获得程序或程序集中每一个类型成员和成员变量的信息。

程序中一般的对象类型都是在编译期就确定下来的，而 Java 反射机制可以动态的创建对象并调用其属性，这样对象的类型在编译期是未知的。所以我们可以通过反射机制直接创建对象即使这个对象在编译期是未知的，

反射的核心：是 JVM 在运行时 才动态加载的类或调用方法或属性，他不需要事先（写代码的时候或编译期）知道运行对象是谁。

一、Java 反射框架主要提供以下功能：

1.在运行时判断任意一个对象所属的类；

2.在运行时构造任意一个类的对象；

3.在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用 private 方法）；

4.在运行时调用任意一个对象的方法

二、主要用途 ：

1、反射最重要的用途就是开发各种通用框架。

很多框架（比如 Spring）都是配置化的（比如通过 XML 文件配置 JavaBean，Action 之类的），为了保证框架的通用性，他们可能根据配置文件加载不同的对象或类，调用不同的方法，这个时候就必须用到反射——运行时动态加载需要加载的对象。

三、基本反射功能的实现(反射相关的类一般都在 java.lang.relfect 包里)：

1、获得 Class 对象

使用 Class 类的 forName 静态方法

```
Class.forName(driver)
```

直接获取某一个对象的 class

```
Class<?> klass = int.class;
Class<?> classInt = Integer.TYPE;
```

调用某个对象的 getClass()方法

```
StringBuilder str = new StringBuilder("123");
Class<?> klass = str.getClass();
```

2、判断是否为某个类的实例

```
public native boolean isInstance(Object obj);
```

用 instanceof 关键字来判断是否为某个类的实例

3、创建实例

使用 Class 对象的 newInstance()方法来创建 Class 对象对应类的实例。

```
Class<?> c    = String.calss;
Object str = c.getInstance();
```

先通过 Class 对象获取指定的 Constructor 对象，再调用 Constructor 对象的 newInstance()方法来创建实例。

```
//获取 String 所对应的 Class 对象
    Class<?> c = String.class;
  //获取 String 类带一个 String 参数的构造器
  Constructor constructor = c.getConstructor(String.class);
  //根据构造器创建实例
  Object obj = constructor.newInstance("23333");
  System.out.println(obj);
```

4、获取方法

getDeclaredMethods()

5、获取构造器信息

getDeclaredMethods()

getMethods()

getMethod()

6、获取类的成员变量（字段）信息

getFiled: 访问公有的成员变量

getDeclaredField：所有已声明的成员变量。但不能得到其父类的成员变量

getFileds 和 getDeclaredFields 用法

7、调用方法

invoke()

8、利用反射创建数组

反射缺点:

由于反射会额外消耗一定的系统资源，因此如果不需要动态地创建一个对象，那么就不需要用反射。

另外，反射调用方法时可以忽略权限检查，因此可能会破坏封装性而导致安全问题。

9、List 和 map 的区别

List 是继承自 Collection 接口，Map 是 Map 接口。

1、List 是存储单列数据的集合，map 是存储键和值这样的双列数据的集合，List 中存储的数据是有顺序，并且允许重复；

2、Map 中存储的数据是没有顺序的，其键是不能重复的，它的值是可以有重复

List 特点：元素有放入顺序，元素可重复;

Map 特点：元素按键值对存储，无放入顺序 ;

List 接口有三个实现类：LinkedList，ArrayList，Vector;

LinkedList：底层基于链表实现，链表内存是散乱的，每一个元素存储本身内存地址的同时还存储下一个元素的地址。链表增删快，查找慢;

Map 接口有三个实现类：HashMap，HashTable，LinkeHashMap

Map 相当于和 Collection 一个级别的；Map 该集合存储键值对，且要求保持键的唯一性。

10、Arraylist 与 LinkedList 区别

1) 因为 Array 是基于索引(index)的数据结构，它使用索引在数组中搜索和读取数据是很快的。Array 获取数据的时间复杂度是 O(1),但是要删除数据却是开销很大的，因为这需要重排数组中的所有数据。

2) 相对于 ArrayList，LinkedList 插入是更快的。因为 LinkedList 不像 ArrayList 一样，不需要改变数组的大小，也不需要在数组装满的时候要将所有的数据重新装入一个新的数组，这是 ArrayList 最坏的一种情况，时间复杂度是 O(n)，而 LinkedList 中插入或删除的时间复杂度仅为 O(1)。ArrayList 在插入数据时还需要更新索引（除了插入数组的尾部）。

3) 类似于插入数据，删除数据时，LinkedList 也优于 ArrayList。

4) LinkedList 需要更多的内存，因为 ArrayList 的每个索引的位置是实际的数据，而 LinkedList 中的每个节点中存储的是实际的数据和前后节点的位置。

5) 你的应用不会随机访问数据。因为如果你需要 LinkedList 中的第 n 个元素的时候，你需要从第一个元素顺序数到第 n 个数据，然后读取数据。

6) 你的应用更多的插入和删除元素，更少的读取数据。因为插入和删除元素不涉及重排数据，所以它要比 ArrayList 要快。

11、 ArrayList 与 Vector 区别

1） 同步性:Vector 是线程安全的，也就是说是同步的 ，而 ArrayList 是线程序不安全的，不是同步的 数 2。

2）数据增长:当需要增长时,Vector 默认增长为原来一倍 ，而 ArrayList 却是原来的 50% ，这样,ArrayList 就有利于节约内存空间。

如果涉及到堆栈，队列等操作，应该考虑用 Vector，如果需要快速随机访问元素，应该使用 ArrayList 。

12、HashMap 和 Hashtable 的区别

1)HashMap 几乎可以等价于 Hashtable，除了 HashMap 是非 synchronized 的，并可以接受 null(HashMap 可以接受为 null 的键值(key)和值(value)，而 Hashtable 则不行)。

2) HashMap 是非 synchronized，而 Hashtable 是 synchronized，这意味着 Hashtable 是线程安全的，多个线程可以共享一个 Hashtable；而如果没有正确的同步的话，多个线程是不能共享 HashMap 的。Java 5 提供了 ConcurrentHashMap，它是 HashTable 的替代，比 HashTable 的扩展性更好。

3) 另一个区别是 HashMap 的迭代器(Iterator)是 fail-fast 迭代器，而 Hashtable 的 enumerator 迭代器不是 fail-fast 的。所以当有其它线程改变了 HashMap 的结构（增加或者移除元素），将会抛出 ConcurrentModificationException，但迭代器本身的 remove()方法移除元素则不会抛出 ConcurrentModificationException 异常。但这并不是一个一定发生的行为，要看 JVM。这条同样也是 Enumeration 和 Iterator 的区别。

4) 由于 Hashtable 是线程安全的也是 synchronized，所以在单线程环境下它比 HashMap 要慢。如果你不需要同步，只需要单一线程，那么使用 HashMap 性能要好过 Hashtable。

5) HashMap 不能保证随着时间的推移 Map 中的元素次序是不变的。

13、HashSet 和 HashMap 区别

HashSet：

HashSet 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。

HashSet 实现了 Set 接口，它不允许集合中出现重复元素。当我们提到 HashSet 时，第一件事就是在将对象存储在 HashSet 之前，要确保重写 hashCode（）方法和 equals（）方法，这样才能比较对象的值是否相等，确保集合中没有储存相同的对象。如果不重写上述两个方法，那么将使用下面方法默认实现：public boolean add(Object obj)方法用在 Set 添加元素时，如果元素值重复时返回 "false"，如果添加成功则返回"true"

HashMap：

HashMap 实现了 Map 接口，Map 接口对键值对进行映射。Map 中不允许出现重复的键（Key）。Map 接口有两个基本的实现 TreeMap 和 HashMap。TreeMap 保存了对象的排列次序，而 HashMap 不能。HashMap 可以有空的键值对（Key（null）-Value（null））。

HashMap 是非线程安全的（非 Synchronize），要想实现线程安全，那么需要调用 collections 类的静态方法 synchronizeMap（）实现。public Object put(Object Key,Object value)方法用来将元素添加到 map 中。

14、HashMap 和 ConcurrentHashMap 的区别

（1）放入 HashMap 的元素是 key-value 对。

（2）底层说白了就是以前数据结构课程讲过的散列结构。

（3）要将元素放入到 hashmap 中，那么 key 的类型必须要实现实现 hashcode 方法，默认这个方法是根据对象的地址来计算的，具体我也记不太清楚了，接着还必须覆盖对象的 equal 方法。

（4）ConcurrentHashMap 对整个桶数组进行了分段，而 HashMap 则没有

（5）ConcurrentHashMap 在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好，而 HashMap 没有锁机制，不是线程安全的。。。

15、HashMap 的工作原理及代码实现

HashMap 基于 hashing 原理，我们通过 put()和 get()方法储存和获取对象。当我们将键值对传递给 put()方法时，它调用键对象的 hashCode()方法来计算 hashcode，让后找到 bucket 位置来储存值对象。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，当发生碰撞了，对象将会储存在链表的下一个节点中。 HashMap 在每个链表节点中储存键值对对象。

1.HashMap 介绍

HashMap 为 Map 接口的一个实现类，实现了所有 Map 的操作。HashMap 除了允许 key 和 value 保存 null 值和非线程安全外，其他实现几乎和 HashTable 一致。

HashMap 使用散列存储的方式保存 kay-value 键值对，因此其不支持数据保存的顺序。如果想要使用有序容器可以使用 LinkedHashMap。

在性能上当 HashMap 中保存的 key 的哈希算法能够均匀的分布在每个 bucket 中的是时候，HashMap 在基本的 get 和 set 操作的的时间复杂度都是 O(n)。

在遍历 HashMap 的时候，其遍历节点的个数为 bucket 的个数+HashMap 中保存的节点个数。因此当遍历操作比较频繁的时候需要注意 HashMap 的初始化容量不应该太大。 这一点其实比较好理解：当保存的节点个数一致的时候，bucket 越少，遍历次数越少。

另外 HashMap 在 resize 的时候会有很大的性能消耗，因此当需要在保存 HashMap 中保存大量数据的时候，传入适当的默认容量以避免 resize 可以很大的提高性能。 具体的 resize 操作请参考下面对此方法的分析

HashMap 是非线程安全的类，当作为共享可变资源使用的时候会出现线程安全问题。需要使用线程安全容器：

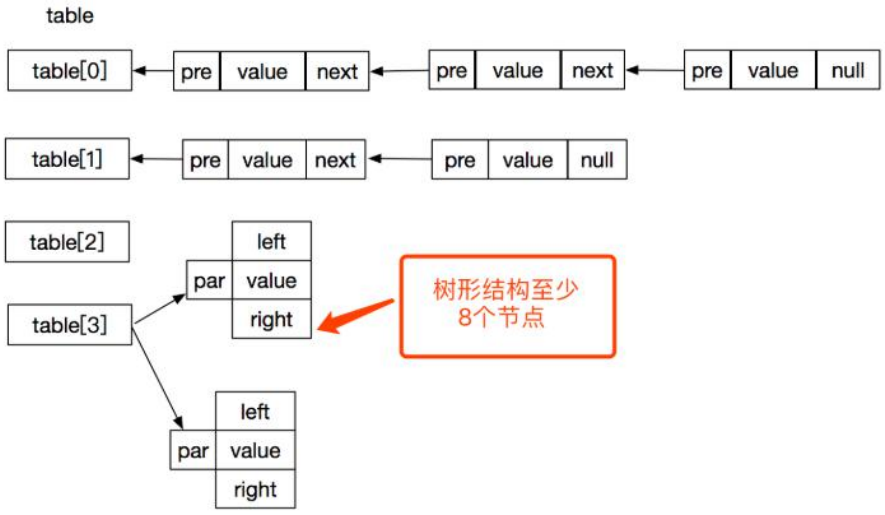
Map m = new ConcurrentHashMap();或者

Map m = Collections.synchronizedMap(new HashMap());

具体的 HashMap 会出现的线程安全问题分析请参考 9 中的分析。

2.数据结构介绍

HashMap 使用数组+链表+树形结构的数据结构。其结构图如下所示。



3.HashMap 源码分析（基于 JDK1.8）

3.1 关键属性分析

transient Node<K,V>[] table; //Node 类型的数组，记我们常说的 bucket 数组，其中每个元素为链表或者树形结构

transient int size;//HashMap 中保存的数据个数

int threshold;//HashMap 需要 resize 操作的阈值

final float loadFactor;//负载因子，用于计算 threshold。计算公式为：threshold = loadFactor * capacity

其中还有一些默认值得属性，有默认容量 2^4，默认负载因子 0.75 等.用于构造函数没有指定数值情况下的默认值。

3.2 构造函数分析

HashMap 提供了三个不同的构造函数，主要区别为是否传入初始化容量和负载因子。分别文以下三个。

```
//此构造函数创建一个空的 HashMap，其中负载因子为默认值 0.75
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
//传入默认的容量大小，创造一个指定容量大小和默认负载因子为 0.75 的 HashMap
 public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
//创建一个指定容量和指定负载因为 HashMap，以下代码删除了入参检查
public HashMap(int initialCapacity, float loadFactor) {
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}
```

注意：此处的 initialCapacity 为数组 table 的大小，即 bucket 的个数。

其中在指定初始化容量的时候，会根据传入的参数来确定 HashMap 的容量大小。

初始化 this.threshold 的值为入参 initialCapacity 距离最近的一个 2 的 n 次方的值。取值方法如下：

case initialCapacity = 0:

this.threshold = 1；

case initialCapacity 为非 0 且不为 2 的 n 次方:

this.threshold = 大于 initialCapacity 中第一个 2 的 n 次方的数。

case initialCapacity = 2^n：

this.threshold = initialCapacity

具体的计算方法为 tableSizeFor(int cap)函数。计算方法是将入参的最高位下面的所有位都设置为 1，然后加 1

下面以入参为 134217729 为例分析计算过程。

首先将 int 转换为二进制如下：

cap = 0000 1000 0000 0000 0000 0000 0000 0001

另外此处赋值为 this.threshold，是因为构造函数的时候并不会创建 table，只有实际插入数据的时候才会创建。目的应该是为了节省内存空间吧。

在第一次插入数据的时候，会将 table 的 capacity 设置为 threshold，同时将 threshold 更新为 loadFactor * capacity

3.3 关键函数源码分析

3.3.1 第一次插入数据的操作

HashMap 在插入数据的时候传入 key-value 键值对。使用 hash 寻址确定保存数据的

bucket。当第一次插入数据的时候会进行 HashMap 中容器的初始化。具体操作如下：

```
Node<K,V>[] tab;
        int n, i;
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
```

其中 resize 函数的源码如下，主要操作为根据 cap 和 loadFactory 创建初始化 table

```
Node<K, V>[] oldTab = table;
    int oldThr = threshold;   //oldThr 根据传入的初始化 cap 决定 2 的 n 次方
    int newCap, newThr = 0;
    if (oldThr > 0) // 当构造函数中传入了 capacity 的时候
        newCap = oldThr;   //newCap = threshold   2 的 n 次方，即构造函数的时候的初始化容量
     else {                        // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    float ft = (float)newCap * loadFactor;   // 2 的 n 次方  * loadFactory
    newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    threshold = newThr; //新的 threshold== newCap * loadFactory
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap]; //长度为 2 的 n 次方的数组
    table = newTab;
```

在初始化 table 之后，将数据插入到指定位置，其中 bucket 的确定方法为：

```
i = (n - 1) & hash // 此处 n-1 必定为 0000 1111 1111....的格式，取&操作之后的值一定在数组的容量范围内。
```

其中 hash 的取值方式为：

```
 static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

具体操作如下，创建 Node 并将 node 放到 table 的第 i 个元素中

```
if ((p = tab[i = (n - 1) & hash]) == null)
    tab[i] = new Node(hash, key, value, null);
```

3.3.2 非第一次插入数据的操作源码分析

当 HashMap 中已有数据的时候，再次插入数据，会多出来在链表或者树中寻址的操作，和当 size 到达阈值时候的 resize 操作。多出来的步骤如下：

另外，在 resize 操作中也和第一次插入数据的操作不同，当 HashMap 不为空的时候 resize 操作需要将之前的数据节点复制到新的 table 中。操作如下：

3.4Cloneable 和 Serializable 分析

在 HashMap 的定义中实现了 Cloneable 接口，Cloneable 是一个标识接口，主要用来标识 Object.clone()的合法性，在没有实现此接口的实例中调用 Object.clone()方法会抛出 CloneNotSupportedException 异常。可以看到 HashMap 中重写了 clone 方法。

HashMap 实现 Serializable 接口主要用于支持序列化。同样的 Serializable 也是一个标识接口，本身没有定义任何方法和属性。另外 HashMap 自定义了

private void writeObject(java.io.ObjectOutputStream s) throws IOException
private void readObject(java.io.ObjectInputStream s) throws IOException,
ClassNotFoundException
两个方法实现了自定义序列化操作。
注意：支持序列化的类必须有无参构造函数。这点不难理解，反序列化的过程中需要通过反射创建对象。
4.HashMap 的遍历
以下讨论两种遍历方式，测试代码如下：
方法一：
通过 map.keySet()获取 key 的集合，然后通过遍历 key 的集合来遍历 map
方法二：
通过 map.entrySet()方法获取 map 中节点集合，然后遍历此集合遍历 map
测试代码如下：

```
public static void main(String[] args) throws Exception {
        Map<String, Object> map = new HashMap<>();
        map.put("name", "test");
        map.put("age", "25");
        map.put("address", "HZ");
        Set<String> keySet = map.keySet();
        for (String key : keySet) {
            System.out.println(map.get(key));
        }
        Set<Map.Entry<String, Object>> set = map.entrySet();
        for (Map.Entry<String, Object> entry : set) {
            System.out.println("key is : " + entry.getKey() + ".    value is " + entry.getValue());
        }
    }
```

16、ConcurrentHashMap 的工作原理及代码实现

ConcurrentHashMap 采用了非常精妙的"分段锁"策略，ConcurrentHashMap 的主干是个 Segment 数组。Segment 继承了 ReentrantLock，所以它就是一种可重入锁（ReentrantLock)。在 ConcurrentHashMap，一个 Segment 就是一个子哈希表，Segment 里维护了一个 HashEntry 数组，并发环境下，对于不同 Segment 的数据进行操作是不用考虑锁竞争的。

ConcurrentHashMap 是 Java1.5 中引用的一个线程安全的支持高并发的 HashMap 集合类。

1、线程不安全的 HashMap

因为多线程环境下，使用 Hashmap 进行 put 操作会引起死循环，导致 CPU 利用率接近 100%，所以在并发情况下不能使用 HashMap。

2、效率低下的 HashTable

HashTable 容器使用 synchronized 来保证线程安全，但在线程竞争激烈的情况下 HashTable 的效率非常低下。

因为当一个线程访问 HashTable 的同步方法时，其他线程访问 HashTable 的同步方法时，可能会进入阻塞或轮询状态。

如线程 1 使用 put 进行添加元素，线程 2 不但不能使用 put 方法添加元素，并且也不能使用 get 方法来获取元素，所以竞争越激烈效率越低。

3、锁分段技术

HashTable 容器在竞争激烈的并发环境下表现出效率低下的原因，是因为所有访问 HashTable 的线程都必须竞争同一把锁，

那假如容器里有多把锁，每一把锁用于锁容器其中一部分数据，那么当多线程访问容器里不同数据段的数据时，线程间就不会存在锁竞争，

从而可以有效的提高并发访问效率，这就是 ConcurrentHashMap 所使用的锁分段技术。

首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问。

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，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。

这里“按顺序”是很重要的，否则极有可能出现死锁，在 ConcurrentHashMap 内部，段数组是 final 的，并且其成员变量实际上也是 final 的，

但是，仅仅是将数组声明为 final 的并不保证数组成员也是 final 的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

oncurrentHashMap 类中包含两个静态内部类 HashEntry 和 Segment。

HashEntry 用来封装映射表的键 / 值对;Segment 用来充当锁的角色，每个 Segment 对象守护整个散列映射表的若干个桶。

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。

每个 Segment 守护者一个 HashEntry 数组里的元素,当对 HashEntry 数组的数据进行修改时，必须首先获得它对应的 Segment 锁。

5、HashEntry 类

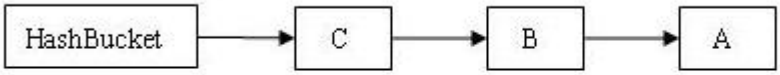
```
static final class HashEntry<K,V> {
        final K key;                        // 声明 key 为 final 型
        final int hash;                     // 声明 hash 值为 final 型
        volatile V value;                   // 声明 value 为 volatile 型
        final HashEntry<K,V> next;   // 声明 next 为 final 型


        HashEntry(K key, int hash, HashEntry<K,V> next, V value)   {
            this.key = key;
            this.hash = hash;
            this.next = next;
            this.value = value;
        }
 }
```

每个 HashEntry 代表 Hash 表中的一个节点，在其定义的结构中可以看到，除了 value 值没有定义 final，其余的都定义为 final 类型，我们知道 Java 中关键词 final 修饰的域成为最终域。

用关键词 final 修饰的变量一旦赋值，就不能改变，也称为修饰的标识为常量。这就意味着我们删除或者增加一个节点的时候，就必须从头开始重新建立 Hash 链，因为 next 引用值需要改变。

由于 HashEntry 的 next 域为 final 型，所以新节点只能在链表的表头处插入。 例如将 A,B,C 插入空桶中，插入后的结构为：



注意：由于只能在表头插入，所以链表中节点的顺序和插入的顺序相反。

6、segment 类

```
static final class Segment<K,V> extends ReentrantLock implements Serializable {
  private static final long serialVersionUID = 2249069246763182397L;
        /**
         * 在本 segment 范围内，包含的 HashEntry 元素的个数
         * 该变量被声明为 volatile 型,保证每次读取到最新的数据
         */
        transient volatile int count;


        /**
         *table 被更新的次数
         */
        transient int modCount;


        /**
         * 当 table 中包含的 HashEntry 元素的个数超过本变量值时，触发 table 的再散列
         */
        transient int threshold;


        /**
         * table 是由 HashEntry 对象组成的数组
         * 如果散列时发生碰撞，碰撞的 HashEntry 对象就以链表的形式链接成一个链表
         * table 数组的数组成员代表散列映射表的一个桶
         * 每个 table 守护整个 ConcurrentHashMap 包含桶总数的一部分
         * 如果并发级别为 16，table 则守护 ConcurrentHashMap 包含的桶总数的 1/16
         */
        transient volatile HashEntry<K,V>[] table;


        /**
         * 装载因子
         */
```

```
        final float loadFactor;
    }
```

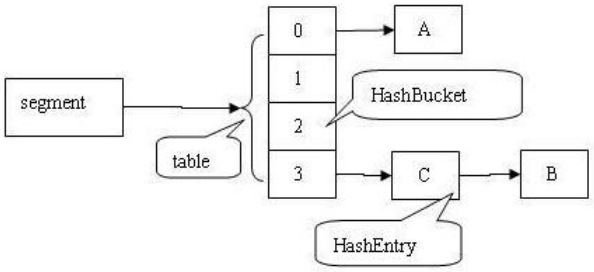
Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 Segment 对象用来守护其（成员对象 table 中）包含的若干个桶。

table 是一个由 HashEntry 对象组成的数组。table 数组的每一个数组成员就是散列映射表的一个桶。

每一个 Segment 对象都有一个 count 对象来表示本 Segment 中包含的 HashEntry 对象的总数。

之所以在每个 Segment 对象中包含一个计数器，而不是在 ConcurrentHashMap 中使用全局的计数器，是为了避免出现“热点域”而影响 ConcurrentHashMap 的并发性。

下图是依次插入 ABC 三个 HashEntry 节点后，Segment 的结构示意图。



7、ConcurrentHashMap 类

默认的情况下，每个 ConcurrentHashMap 类会创建 16 个并发的 segment，每个 segment 里面包含多个 Hash 表，每个 Hash 链都是有 HashEntry 节点组成的。

如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。

```
    public class ConcurrentHashMap<K, V> extends AbstractMap<K, V>
            implements ConcurrentMap<K, V>, Serializable {


         /**
         * 散列映射表的默认初始容量为 16，即初始默认为 16 个桶
         * 在构造函数中没有指定这个参数时，使用本参数
         */
        static final        int DEFAULT_INITIAL_CAPACITY= 16;

        /**
         * 散列映射表的默认装载因子为 0.75，该值是 table 中包含的 HashEntry 元素的个数与
    * table 数组长度的比值
         * 当 table 中包含的 HashEntry 元素的个数超过了 table 数组的长度与装载因子的乘积时，
    * 将触发 再散列
         * 在构造函数中没有指定这个参数时，使用本参数
         */
        static final float DEFAULT_LOAD_FACTOR= 0.75f;
```

```
    /**
     * 散列表的默认并发级别为 16。该值表示当前更新线程的估计数
     * 在构造函数中没有指定这个参数时，使用本参数
     */
    static final int DEFAULT_CONCURRENCY_LEVEL= 16;

    /**
     * segments 的掩码值
     * key 的散列码的高位用来选择具体的 segment
     */
    final int segmentMask;

    /**
     * 偏移量
     */
    final int segmentShift;

    /**
     * 由 Segment 对象组成的数组
     */
    final Segment<K,V>[] segments;

    /**
     * 创建一个带有指定初始容量、加载因子和并发级别的新的空映射。
     */
    public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel) {
        if(!(loadFactor > 0) || initialCapacity < 0 ||
concurrencyLevel <= 0)
            throw new IllegalArgumentException();

        if(concurrencyLevel > MAX_SEGMENTS)
            concurrencyLevel = MAX_SEGMENTS;

        // 寻找最佳匹配参数（不小于给定参数的最接近的 2 次幂）
        int sshift = 0;
        int ssize = 1;
        while(ssize < concurrencyLevel) {
            ++sshift;
            ssize <<= 1;
        }
        segmentShift = 32 - sshift;          // 偏移量值
        segmentMask = ssize - 1;             // 掩码值
        this.segments = Segment.newArray(ssize);    // 创建数组
```

```
            if (initialCapacity > MAXIMUM_CAPACITY)
                initialCapacity = MAXIMUM_CAPACITY;
            int c = initialCapacity / ssize;
            if(c * ssize < initialCapacity)
                ++c;
            int cap = 1;
            while(cap < c)
                cap <<= 1;

            // 依次遍历每个数组元素
            for(int i = 0; i < this.segments.length; ++i)
                // 初始化每个数组元素引用的 Segment 对象
    this.segments[i] = new Segment<K,V>(cap, loadFactor);
        }

        /**
         * 创建一个带有默认初始容量 (16)、默认加载因子 (0.75) 和 默认并发级别 (16)
     * 的空散列映射表。
         */
        public ConcurrentHashMap() {
            // 使用三个默认参数，调用上面重载的构造函数来创建空散列映射表
    this(DEFAULT_INITIAL_CAPACITY,                    DEFAULT_LOAD_FACTOR,
DEFAULT_CONCURRENCY_LEVEL);
    }
```

8、 用分离锁实现多个线程间的并发写操作

插入数据后的 ConcurrentHashMap 的存储形式

（1）Put 方法的实现

首先，根据 key 计算出对应的 hash 值：

```
public V put(K key, V value) {
    if (value == null)              //ConcurrentHashMap 中不允许用 null 作为映射值
        throw new NullPointerException();
    int hash = hash(key.hashCode());          // 计算键对应的散列码
    // 根据散列码找到对应的 Segment
    return segmentFor(hash).put(key, hash, value, false);
}
```

根据 hash 值找到对应的 Segment：

```
/**
* 使用 key 的散列码来得到 segments 数组中对应的 Segment
*/
final Segment<K,V> segmentFor(int hash) {
// 将散列值右移 segmentShift 个位，并在高位填充 0
// 然后把得到的值与 segmentMask 相“与”
// 从而得到 hash 值对应的 segments 数组的下标值
// 最后根据下标值返回散列码对应的 Segment 对象
```

```
        return segments[(hash >>> segmentShift) & segmentMask];
    }
```

在这个 Segment 中执行具体的 put 操作：

```
        V put(K key, int hash, V value, boolean onlyIfAbsent) {
            lock();  // 加锁，这里是锁定某个 Segment 对象而非整个 ConcurrentHashMap
            try {
                int c = count;

                if (c++ > threshold)        // 如果超过再散列的阈值
                    rehash();                    // 执行再散列，table 数组的长度将扩充一倍

                HashEntry<K,V>[] tab = table;
                // 把散列码值与 table 数组的长度减 1 的值相“与”
                // 得到该散列码对应的 table 数组的下标值
                int index = hash & (tab.length - 1);
                // 找到散列码对应的具体的那个桶
                HashEntry<K,V> first = tab[index];

                HashEntry<K,V> e = first;
                while (e != null && (e.hash != hash || !key.equals(e.key)))
                    e = e.next;

                V oldValue;
                if (e != null) {                    // 如果键 / 值对以经存在
                    oldValue = e.value;
                    if (!onlyIfAbsent)
                        e.value = value;       // 设置 value 值
                }
                else {                                  // 键 / 值对不存在
                    oldValue = null;
                    ++modCount;                // 要添加新节点到链表中，所以 modCont 要
加 1
                    // 创建新节点，并添加到链表的头部
                    tab[index] = new HashEntry<K,V>(key, hash, first, value);
                    count = c;                       // 写 count 变量
                }
                return oldValue;
            } finally {
                unlock();                                // 解锁
            }
        `}
```

这里的加锁操作是针对（键的 hash 值对应的）某个具体的 Segment，锁定的是该 Segment 而不是整个 ConcurrentHashMap。

因为插入键 / 值对操作只是在这个 Segment 包含的某个桶中完成，不需要锁定整个

ConcurrentHashMap。
此时，其他写线程对另外 15 个 Segment 的加锁并不会因为当前线程对这个 Segment 的加锁而阻塞。
同时，所有读线程几乎不会因本线程的加锁而阻塞（除非读线程刚好读到这个 Segment 中某个 HashEntry 的 value 域的值为 null，此时需要加锁后重新读取该值）。
（2）Get 方法的实现

```
V get(Object key, int hash) {
    if(count != 0) {            // 首先读 count 变量
        HashEntry<K,V> e = getFirst(hash);
        while(e != null) {
            if(e.hash == hash && key.equals(e.key)) {
                V v = e.value;
                if(v != null)
                    return v;
                // 如果读到 value 域为 null，说明发生了重排序，加锁后重新读取
                return readValueUnderLock(e);
            }
            e = e.next;
        }
    }
    return null;
}
 V readValueUnderLock(HashEntry<K,V> e) {
     lock();
     try {
         return e.value;
     } finally {
         unlock();
     }
 }
```

ConcurrentHashMap 中的读方法不需要加锁，所有的修改操作在进行结构修改时都会在最后一步写 count 变量，通过这种机制保证 get 操作能够得到几乎最新的结构更新。
（3）Remove 方法的实现

```
V remove(Object key, int hash, Object value) {
    lock(); //加锁
    try{
        int c = count - 1;
        HashEntry<K,V>[] tab = table;
        //根据散列码找到 table 的下标值
        int index = hash & (tab.length - 1);
        //找到散列码对应的那个桶
        HashEntry<K,V> first = tab[index];
        HashEntry<K,V> e = first;
        while(e != null&& (e.hash != hash || !key.equals(e.key)))
```

```
                e = e.next;


            V oldValue = null;
            if(e != null) {
                V v = e.value;
                if(value == null|| value.equals(v)) { //找到要删除的节点
                    oldValue = v;
                    ++modCount;
                    //所有处于待删除节点之后的节点原样保留在链表中
                    //所有处于待删除节点之前的节点被克隆到新链表中
                    HashEntry<K,V> newFirst = e.next;// 待删节点的后继结点
                    for(HashEntry<K,V> p = first; p != e; p = p.next)
                        newFirst = new HashEntry<K,V>(p.key, p.hash,
                                                      newFirst, p.value);
                    //把桶链接到新的头结点
                    //新的头结点是原链表中，删除节点之前的那个节点
                    tab[index] = newFirst;
                    count = c;          //写 count 变量
                }
            }
            return oldValue;
        } finally{
            unlock(); //解锁
        }
    }
```

整个操作是在持有段锁的情况下执行的，空白行之前的行主要是定位到要删除的节点 e。

如果不存在这个节点就直接返回 null，否则就要将 e 前面的结点复制一遍，尾结点指向 e 的下一个结点。

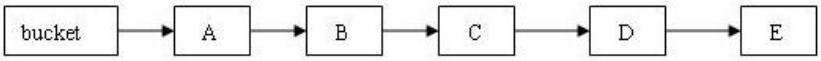
e 后面的结点不需要复制，它们可以重用。

中间那个 for 循环是做什么用的呢？从代码来看，就是将定位之后的所有 entry 克隆并拼回前面去，但有必要吗？

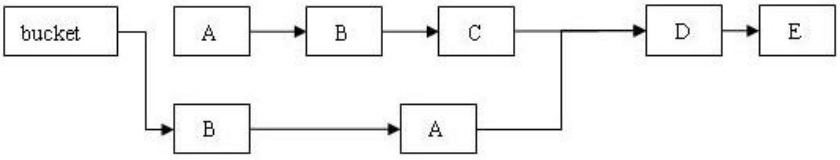
每次删除一个元素就要将那之前的元素克隆一遍？这点其实是由 entry 的不变性来决定的，仔细观察 entry 定义，发现除了 value，其他所有属性都是用 final 来修饰的，

这意味着在第一次设置了 next 域之后便不能再改变它，取而代之的是将它之前的节点全都克隆一次。至于 entry 为什么要设置为不变性，这跟不变性的访问不需要同步从而节省时间有关。

执行删除之前的原链表：



执行删除之后的新链表



注意：新链表在clone的时候。顺序发生反转，A->B变为B->A。

（4）containsKey 方法的实现，它不需要读取值。

```
boolean containsKey(Object key, int hash) {
        if (count != 0) { // read-volatile
            HashEntry<K,V> e = getFirst(hash);
            while (e != null) {
                if (e.hash == hash && key.equals(e.key))
                    return true;
                e = e.next;
            }
        }
        return false;
    }
```

（5）size()

我们要统计整个 ConcurrentHashMap 里元素的大小，就必须统计所有 Segment 里元素的大小后求和。

Segment 里的全局变量 count 是一个 volatile 变量，那么在多线程场景下，我们是不是直接把所有 Segment 的 count 相加就可以得到整个 ConcurrentHashMap 大小了呢？

不是的，虽然相加时可以获取每个 Segment 的 count 的最新值，但是拿到之后可能累加前使用的 count 发生了变化，那么统计结果就不准了。

所以最安全的做法，是在统计 size 的时候把所有 Segment 的 put，remove 和 clean 方法全部锁住，但是这种做法显然非常低效。

因为在累加 count 操作过程中，之前累加过的 count 发生变化的几率非常小，所以 ConcurrentHashMap 的做法是先尝试 2 次通过不锁住 Segment 的方式来统计各个 Segment 大小，如果统计的过程中，容器的 count 发生了变化，则再采用加锁的方式来统计所有 Segment 的大小。

那么 ConcurrentHashMap 是如何判断在统计的时候容器是否发生了变化呢？使用 modCount 变量，在 put , remove 和 clean 方法里操作元素前都会将变量 modCount 进行加 1，那么在统计 size 前后比较 modCount 是否发生变化，从而得知容器的大小是否发生变化。

9、总结

1.在使用锁来协调多线程间并发访问的模式下，减小对锁的竞争可以有效提高并发性。

有两种方式可以减小对锁的竞争：

减小请求同一个锁的频率。

减少持有锁的时间。

2.ConcurrentHashMap 的高并发性主要来自于三个方面：

用分离锁实现多个线程间的更深层次的共享访问。

用 HashEntery 对象的不变性来降低执行读操作的线程在遍历链表期间对加锁的需求。

通过对同一个 Volatile 变量的写 / 读访问，协调不同线程间读 / 写操作的内存可见性。

使用分离锁，减小了请求同一个锁的频率。

17、为什么 Hashtable ConcurrentHashmap 不支持 key 或者 value 为 null

ConcurrentHashmap HashMap 和 Hashtable 都是 key-value 存储结构，但他们有一个不同点是 ConcurrentHashmap、Hashtable 不支持 key 或者 value 为 null，而 HashMap 是支持的。为什么会有这个区别？在设计上的目的是什么？

ConcurrentHashmap 和 Hashtable 都是支持并发的，这样会有一个问题，当你通过 get(k)获取对应的 value 时，如果获取到的是 null 时，你无法判断，它是 put（k,v）的时候 value 为 null，还是这个 key 从来没有做过映射。HashMap 是非并发的，可以通过 contains(key)来做这个判断。而支持并发的 Map 在调用 m.contains（key）和 m.get(key),m 可能已经不同了。

HashMap.class:

// 此处计算 key 的 hash 值时，会判断是否为 null，如果是，则返回 0，即 key 为 null 的键值对

// 的 hash 为 0。因此一个 hashmap 对象只会存储一个 key 为 null 的键值对，因为它们的 hash 值都相同。

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
// 将键值对放入 table 中时，不会校验 value 是否为 null。因此一个 hashmap 对象可以存储
// 多个 value 为 null 的键值对
public V put(K key, V value) {
    return putVal(hash(key), key, value, false, true);
}
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
```

```
                if (p.hash == hash &&
                    ((k = p.key) == key || (key != null && key.equals(k))))
                    e = p;
                else if (p instanceof TreeNode)
                    e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
                else {
                    for (int binCount = 0; ; ++binCount) {
                        if ((e = p.next) == null) {
                            p.next = newNode(hash, key, value, null);
                            if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                                treeifyBin(tab, hash);
                            break;
                        }
                        if (e.hash == hash &&
                            ((k = e.key) == key || (key != null && key.equals(k))))
                            break;
                        p = e;
                    }
                }
                if (e != null) { // existing mapping for key
                    V oldValue = e.value;
                    if (!onlyIfAbsent || oldValue == null)
                        e.value = value;
                    afterNodeAccess(e);
                    return oldValue;
                }
            }
            ++modCount;
            if (++size > threshold)
                resize();
            afterNodeInsertion(evict);
            return null;
        }
Hashtable.class:
public synchronized V put(K key, V value) {
        // 确保 value 不为空。这句代码过滤掉了所有 value 为 null 的键值对。因此 Hashtable
不能
        // 存储 value 为 null 的键值对
        if (value == null) {
            throw new NullPointerException();
        }
        // 确保 key 在 table 数组中尚未存在。
        Entry<?,?> tab[] = table;
        int hash = key.hashCode(); //在此处计算 key 的 hash 值，如果此处 key 为 null，则直
```

接抛出空指针异常。

```
        int index = (hash & 0x7FFFFFFF) % tab.length;
        @SuppressWarnings("unchecked")
        Entry<K,V> entry = (Entry<K,V>)tab[index];
        for(; entry != null ; entry = entry.next) {
            if ((entry.hash == hash) && entry.key.equals(key)) {
                V old = entry.value;
                entry.value = value;
                return old;
            }
        }
        addEntry(hash, key, value, index);
        return null;
    }
```

基本功

18、面向对象的特征

面向对象的三个基本特征是：封装、继承、多态。

封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。

封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

继承

面向对象编程 (OOP) 语言的一个主要功能就是“继承”。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

多态

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

实现多态，有二种方式，覆盖，重载。

19、int 和 Integer 有什么区别

int 是 java 提供的 8 种原始数据类型之一。Java 为每个原始类型提供了封装类，Integer 是 java 为 int 提供的封装类。

int 的默认值为 0，而 Integer 的默认值为 null，是引用类型，即 Integer 可以区分出未赋值和值为 0 的区别，int 则无法表达出未赋值的情况，

Java 中 int 和 Integer 关系是比较微妙的。关系如下：

1、int 是基本的数据类型；

2、Integer 是 int 的封装类；

3、int 和 Integer 都可以表示某一个数值；

4、int 和 Integer 不能够互用，因为他们两种不同的数据类型；

20、重载和重写的区别

重载 Overload 表示同一个类中可以有多个名称相同的方法，但这些方法的参数列表各不相同（即参数个数或类型不同）。

重写 Override 表示子类中的方法可以与父类中的某个方法的名称和参数完全相同，通过子类创建的实例对象调用这个方法时，将调用子类中的定义方法，这相当于把父类中定义的那个完全相同的方法给覆盖了，这也是面向对象编程的多态性的一种表现。子类覆盖父类的方法时，只能比父类抛出更少的异常，或者是抛出父类抛出的异常的子异常，因为子类可以解决父类的一些问题，不能比父类有更多的问题。子类方法的访问权限只能比父类的更大，不能更小。如果父类的方法是 private 类型，那么，子类则不存在覆盖的限制，相当于子类中增加了一个全新的方法。

21、抽象类和接口有什么区别

抽象类是用来捕捉子类的通用特性的。它不能被实例化，只能被用作子类的超类。抽象类是被用来创建继承层级里的子类的模板。

接口是抽象方法的集合，如果一个类实现类某个接口，那么他就继承了这个接口的抽象方法。这就像契约模式。如果实现了这个接口，那么就必须确保使用这些方法。接口只是一种形式，接口自身不能做任何事情。

参数	抽象类	接口
默认的方法实现	它可以有默认的方法实现	接口完全是抽象的。它根本不存在方法的实现
实现	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的方法的实现。	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的方法的实现
构造器	抽象类可以有构造器	接口不能有构造器
与正常 Java 类的区别	除了你不能实例化抽象类之外，它和普通 Java 类没有任何区别	接口是完全不同的类型
访问修饰符	抽象方法可以有 **public**、**protected** 和 **default** 这些修饰符	接口方法默认修饰符是 **public**。你不可以使用其它修饰符。
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。
多继承	抽象方法可以继承一个类和实现多个接口	接口只可以继承一个或多个其它接口
速度	它比接口速度要快	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。
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。

什么时候使用抽象类和接口

1、如果你拥有一些方法想让他们中的一些默认实现，那么使用抽象类。

2、如果你想实现多重继承，那么你必须使用接口。由于 java 不支多继承，子类不能够继承多个类，但可以实现多个接口

3、如果基本功能在不断改变，那么就需要使用抽象类。如果不断改变基本功能并且使用接口 ，那么就需要改变所有实现了该接口的类。

JDK 8 中的默认方法

向接口中引入了默认方法和静态方法，以此来减少抽象类和接口之间的差异。现在我们可以为接口提供默认实现的方法来，并且不用强制来实现它。

22、说说自定义注解的场景及实现

登陆、权限拦截、日志处理，以及各种 Java 框架，如 Spring，Hibernate，JUnit 提到注解就不能不说反射，Java 自定义注解是通过运行时靠反射获取注解。实际开发中，例如我们要获取某个方法的调用日志，可以通过 AOP（动态代理机制）给方法添加切面，通过反射来获取方法包含的注解，如果包含日志注解，就进行日志记录。反射的实现在 Java 应用层面上讲，是通过对 Class 对象的操作实现的，Class 对象为我们提供了一系列方法对类进行操作。在 Jvm 这个角度来说，Class 文件是一组以 8 位字节为基础单位的二进制流，各个数据项目按严格的顺序紧凑的排列在 Class 文件中，里面包含了类、方法、字段等等相关数据。通过对 Claas 数据流的处理我们即可得到字段、方法等数据。

23、HTTP 请求的 GET 与 POST 方式的区别

GET 方法

请注意，查询字符串（名称/值对）是在 GET 请求的 URL 中发送的：

/test/demo_form.asp?name1=value1&name2=value2

请求可被缓存

请求保留在浏览器历史记录中

请求可被收藏为书签

请求不应在处理敏感数据时使用

请求有长度限制

请求只应当用于取回数据

POST 方法

请注意，查询字符串（名称/值对）是在 POST 请求的 HTTP 消息主体中发送的：

POST /test/demo_form.asp HTTP/1.1

Host: w3schools.com

name1=value1&name2=value2

方法	GET	POST
缓存	能被缓存	不能缓存
编码	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或

类型		multipart/form-data。为二进制数据使用多重编码。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）	无限制。
对数据类型的限制	只允许 ASCII 字符	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET	POST 比 GET 更安全，因为参数不会被保存在浏览器历或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

比较 GET 与 POST

其他 HTTP 请求方法
HEAD 与 GET 相同，但只返回 HTTP 报头，不返回文档主体。
PUT 上传指定的 URI 表示。
DELETE 删除指定资源。
OPTIONS 返回服务器支持的 HTTP 方法
CONNECT 把请求连接转换到透明的 TCP/IP 通道。

24、session 与 cookie 区别

cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗考虑到安全应当使用 session。

session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能考虑到减轻服务器性能方面，应当使用 COOKIE。

单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

所以个人建议：

将登陆信息等重要信息存放为 SESSION

其他信息如果需要保留，可以放在 COOKIE 中

25、session 分布式处理

第一种：粘性 session

粘性 Session 是指将用户锁定到某一个服务器上，比如上面说的例子，用户第一次请求

时，负载均衡器将用户的请求转发到了 A 服务器上，如果负载均衡器设置了粘性 Session 的话，那么用户以后的每次请求都会转发到 A 服务器上，相当于把用户和 A 服务器粘到了一块，这就是粘性 Session 机制

第二种：服务器 session 复制

原理：任何一个服务器上的 session 发生改变（增删改），该节点会把这个 session 的所有内容序列化，然后广播给所有其它节点，不管其他服务器需不需要 session，以此来保证 Session 同步。

第三种：session 共享机制

使用分布式缓存方案比如 memcached、Redis，但是要求 Memcached 或 Redis 必须是集群。

原理：不同的 tomcat 指定访问不同的主 memcached。多个 Memcached 之间信息是同步的，能主从备份和高可用。用户访问时首先在 tomcat 中创建 session，然后将 session 复制一份放到它对应的 memcahed 上

第四种：session 持久化到数据库

原理：就不用多说了吧，拿出一个数据库，专门用来存储 session 信息。保证 session 的持久化。 优点：服务器出现问题，session 不会丢失 缺点：如果网站的访问量很大，把 session 存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库。

第五种 terracotta 实现 session 复制

原理：就不用多说了吧，拿出一个数据库，专门用来存储 session 信息。保证 session 的持久化。 优点：服务器出现问题，session 不会丢失 缺点：如果网站的访问量很大，把 session 存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库

26、JDBC 流程

（1）向 DriverManager 类注册驱动数据库驱动程序

```
Class.forName( "com.somejdbcvendor.TheirJdbcDriver" );
```

（2）调用 DriverManager.getConnection 方法， 通过 JDBC URL，用户名，密码取得数据库连接的 Connection 对象。

```
Connection conn = DriverManager.getConnection(
        "jdbc:somejdbcvendor:other data needed by some jdbc vendor", //URL
         "myLogin", //  用户名
        "myPassword" ); //  密码
```

（3）获取 Connection 后， 便可以通过 createStatement 创建 Statement 用以执行 SQL 语句。

```
Statement stmt = conn.createStatement();
 stmt.executeUpdate( "INSERT INTO MyTable( name ) VALUES ( 'my name' ) " );
```

要执行 SQL 语句，必须获得 java.sql.Statement 实例，Statement 实例分为以下 3 种类型：

1、执行静态 SQL 语句。通常通过 Statement 实例实现。

2、执行动态 SQL 语句。通常通过 PreparedStatement 实例实现。

3、执行数据库存储过程。通常通过 CallableStatement 实例实现。

具体的实现方式：

```
Statement stmt = con.createStatement() ; PreparedStatement pstmt = con.prepareStatement(sql) ; CallableStatement cstmt = con.prepareCall("{CALL demoSp(? , ?)}") ;
```

（4）有时候会得到查询结果，比如 select，得到查询结果，查询（SELECT）的结果存放于结果集（ResultSet）中。

```
ResultSet rs = stmt.executeQuery( "SELECT * FROM MyTable" );
```

Statement 接口提供了三种执行 SQL 语句的方法：executeQuery 、executeUpdate 和 execute
1、ResultSet executeQuery(String sqlString)：执行查询数据库的 SQL 语句 ，返回一个结果集（ResultSet）对象。
2、int executeUpdate(String sqlString)：用于执行 INSERT、UPDATE 或 DELETE 语句以及 SQL DDL 语句，如：CREATE TABLE 和 DROP TABLE 等
3、execute(sqlString):用于执行返回多个结果集、多个更新计数或二者组合的 语句。 具体实现的代码：
ResultSet rs = stmt.executeQuery(“ SELECT * FROM … ”) ; int rows = stmt.executeUpdate(“INSERT INTO …”) ; boolean flag = stmt.execute(String sql) ;
获取结果为两种情况：
1、执行更新返回的是本次操作影响到的记录数。
2、执行查询返回的结果是一个 ResultSet 对象。
• ResultSet 包含符合 SQL 语句中条件的所有行，并且它通过一套 get 方法提供了对这些 行中数据的访问。
• 使用结果集（ResultSet）对象的访问方法获取数据：

```
while(rs.next()){
String name = rs.getString(“name”) ;
String pass = rs.getString(1) ; // 此方法比较高效
}
```

（列是从左到右编号的，并且从列 1 开始）
（5）关闭数据库语句，关闭数据库连接。

```
rs.close();
stmt.close();
```

27、equals 与 == 的区别

==与 equals 的主要区别是：==常用于比较原生类型，而 equals()方法用于检查对象的相等性。另一个不同的点是：如果==和 equals()用于比较对象，当两个引用地址相同，==返回 true。而 equals()可以返回 true 或者 false 主要取决于重写实现。最常见的一个例子，字符串的比较，不同情况==和 equals()返回不同的结果。

使用==比较原生类型如：boolean、int、char 等等，使用 equals()比较对象。
==返回 true 如果两个引用指向相同的对象，equals()的返回结果依赖于具体业务实现
字符串的对比使用 equals()代替==操作符
使用==比较原生类型如：boolean、int、char 等等，使用 equals()比较对象。
==返回 true 如果两个引用指向相同的对象，equals()的返回结果依赖于具体业务实现
字符串的对比使用 equals()代替==操作符
其主要的不同是一个是操作符一个是方法，==用于对比原生类型而 equals()方法比较对象的相等性。

28、MVC 设计思想

1：什么是 MVC

MVC(Model View Controller)是一种软件设计的框架模式，它采用模型(Model)-视图(View)-控制器(controller)的方法把业务逻辑、数据与界面显示分离。把众多的业务逻辑聚集到一个部件里面，当然这种比较官方的解释是不能让我们足够清晰的理解什么是 MVC 的。

用通俗的话来讲，MVC 的理念就是把数据处理、数据展示(界面)和程序/用户的交互三者分离开的一种编程模式。

注意！MVC 不是设计模式！

MVC 框架模式是一种复合模式，MVC 的三个核心部件分别是

1：Model(模型)：所有的用户数据、状态以及程序逻辑，独立于视图和控制器

2：View(视图)：呈现模型，类似于 Web 程序中的界面，视图会从模型中拿到需要展现的状态以及数据，对于相同的数据可以有多种不同的显示形式(视图)

3：Controller(控制器)：负责获取用户的输入信息，进行解析并反馈给模型，通常情况下一个视图具有一个控制器

1.2：为什么要使用 MVC

程序通过将 M(Model)和 V(View)的代码分离，实现了前后端代码的分离，会带来几个好处

1：可以使同一个程序使用不同的表现形式，如果控制器反馈给模型的数据发生了变化，那么模型将及时通知有关的视图，视图会对应的刷新自己所展现的内容

2：因为模型是独立于视图的，所以模型可复用，模型可以独立的移植到别的地方继续使用

3：前后端的代码分离，使项目开发的分工更加明确，程序的测试更加简便，提高开发效率。

其实控制器的功能类似于一个中转站，会决定调用那个模型去处理用户请求以及调用哪个视图去呈现给用户

1.3：JavaWeb 中 MVC 模式的应用

在 JavaWeb 程序中，MVC 框架模式是经常用到的，举一个 Web 程序的结构可以更好的理解 MVC 的理念

V：View 视图，Web 程序中指用户可以看到的并可以与之进行数据交互的界面，比如一个 Html 网页界面，或者某些客户端的界面，在前面讲过，MVC 可以为程序处理很多不同的视图，用户在视图中进行输出数据以及一系列操作，注意：视图中不会发生数据的处理操作。

M：Model 模型：进行所有数据的处理工作，模型返回的数据是中立的，和数据格式无关，一个模型可以为多个视图来提供数据，所以模型的代码重复性比较低

C：Controller 控制器：负责接受用户的输入，并且调用模型和视图去完成用户的需求，控制器不会输出也不会做出任何处理，只会接受请求并调用模型构件去处理用户的请求，然后在确定用哪个视图去显示返回的数据

1.4：Web 程序中 MVC 模式的优点

耦合性低：视图(页面)和业务层(数据处理)分离，一个应用的业务流程或者业务规则的改变只需要改动 MVC 中的模型即可，不会影响到控制器与视图

部署快，成本低：MVC 使开发和维护用户接口的技术含量降低。使用 MVC 模式使开发时间得到相当大的缩减，它使程序员（Java 开发人员）集中精力于业务逻辑，界面程序员（HTML 和 JSP 开发人员）集中精力于表现形式上

可维护性高：分离视图层和业务逻辑层也使得 WEB 应用更易于维护和修改

1.5：Web 程序中 MVC 模式的缺点

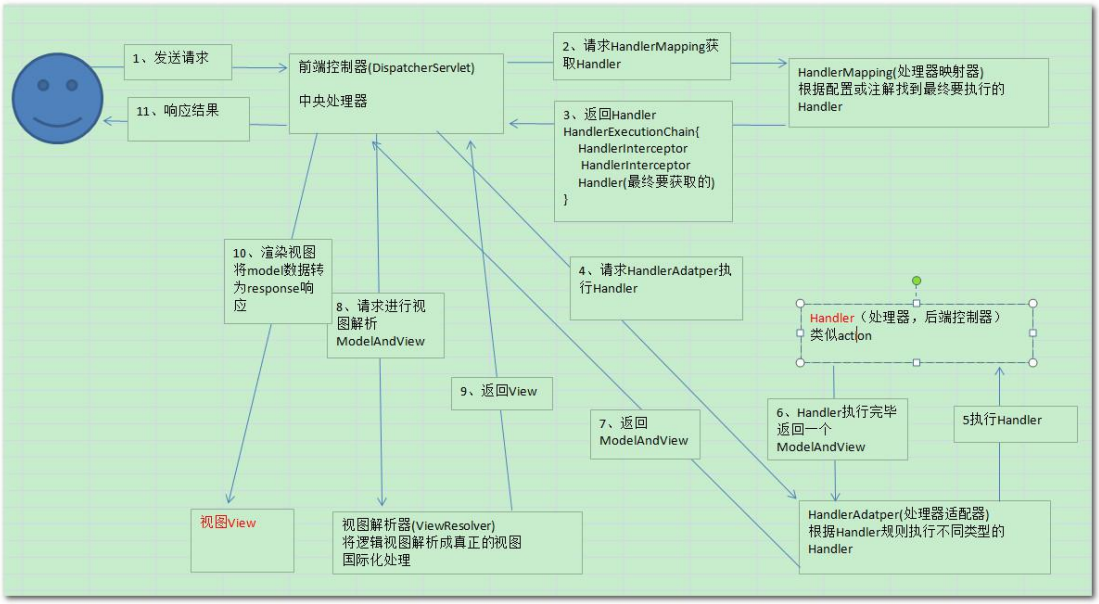
调试困难：因为模型和视图要严格的分离，这样也给调试应用程序带来了一定的困难，每个构件在使用之前都需要经过彻底的测试

不适合小型，中等规模的应用程序：在一个中小型的应用程序中，强制性的使用 MVC 进行开发，往往会花费大量时间，并且不能体现 MVC 的优势，同时会使开发变得繁琐

增加系统结构和实现的复杂性：对于简单的界面，严格遵循 MVC，使模型、视图与控制器分离，会增加结构的复杂性，并可能产生过多的更新操作，降低运行效率

视图与控制器间的过于紧密的连接并且降低了视图对模型数据的访问：视图与控制器是相互分离，但却是联系紧密的部件，视图没有控制器的存在，其应用是很有限的，反之亦然，这样就妨碍了他们的独立重用。依据模型操作接口的不同，视图可能需要多次调用才能获得足够的显示数据。对未变化数据的不必要的频繁访问，也将损害操作性能。

29、springmvc 工作原理图



SpringMVC 流程

1、 用户发送请求至前端控制器 DispatcherServlet。

2、 DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3、 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4、 DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5、 HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller 执行完成返回 ModelAndView。

7、 HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8、 DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9、 ViewReslover 解析后返回具体 View。

10、DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet 响应用户。

组件说明：

以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

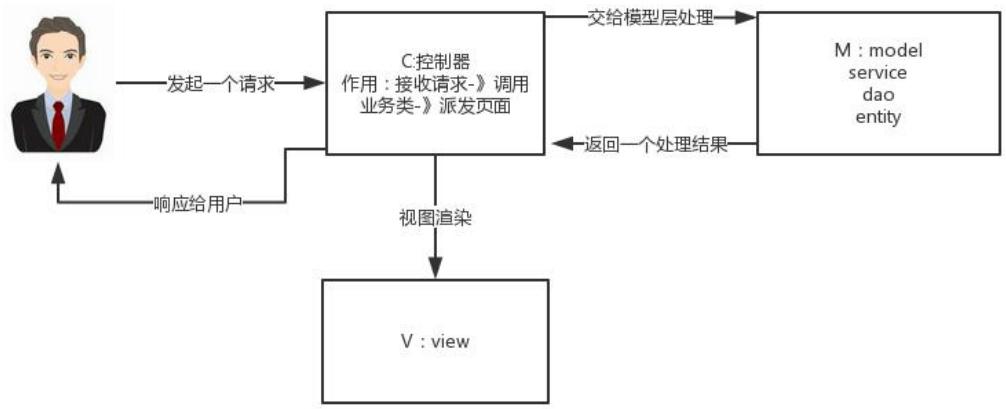
HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。

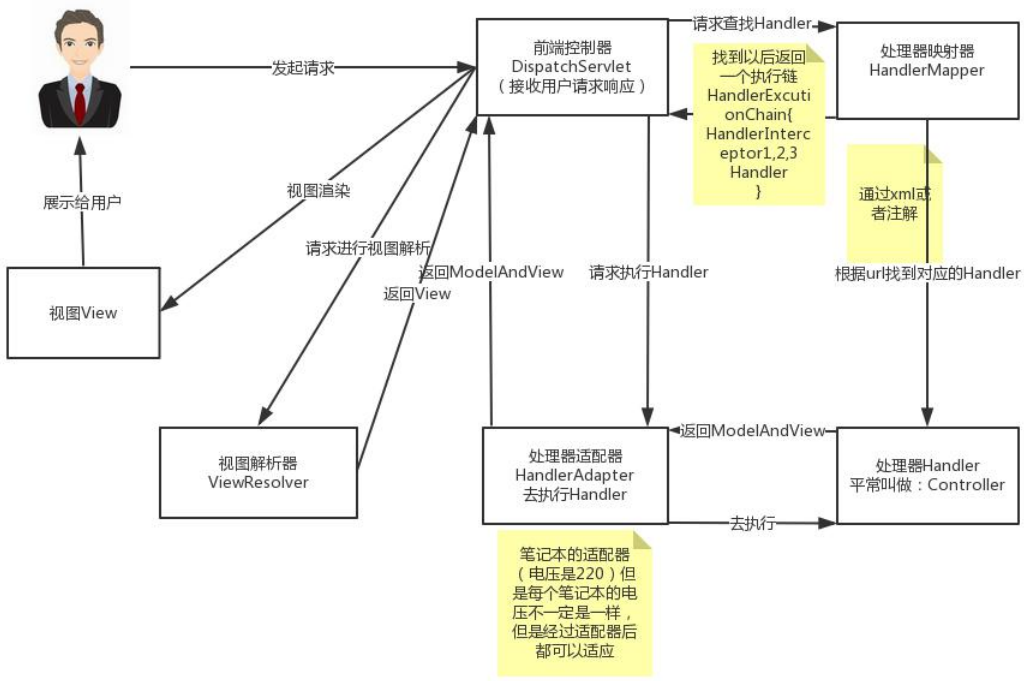
ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、

pdf、excel 等。
组件：
1、前端控制器 DispatcherServlet（不需要工程师开发）,由框架提供
作用：接收请求，响应结果，相当于转发器，中央处理器。有了 dispatcherServlet 减少了其它组件之间的耦合度。
用户请求到达前端控制器，它就相当于 mvc 模式中的 c，dispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet 的存在降低了组件之间的耦合性。
2、处理器映射器 HandlerMapping(不需要工程师开发),由框架提供
作用：根据请求的 url 查找 Handler
HandlerMapping 负责根据用户请求找到 Handler 即处理器，springmvc 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。
3、处理器适配器 HandlerAdapter
作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler
通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。
4、处理器 Handler(需要工程师开发)
注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler
Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。
由于 Handler 涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发 Handler。
5、视图解析器 View resolver(不需要工程师开发),由框架提供
作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）
View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。 springmvc 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。
一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。
6、视图 View(需要工程师开发 jsp...)
View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker、pdf...）
核心架构的具体流程步骤如下：
1、首先用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制；
2、DispatcherServlet——>HandlerMapping，HandlerMapping 将会把请求映射为 HandlerExecutionChain 对象（包含一个 Handler 处理器（页面控制器）对象、多个 HandlerInterceptor 拦截器）对象，通过这种策略模式，很容易添加新的映射策略；
3、DispatcherServlet——>HandlerAdapter，HandlerAdapter 将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；
4、HandlerAdapter——>处理器功能处理方法的调用，HandlerAdapter 将会根据适配的结果调用真正的处理器的功能处理方法，完成功能处理；并返回一个 ModelAndView 对象（包含模型数据、逻辑视图名）；

5、ModelAndView 的逻辑视图名——> ViewResolver， ViewResolver 将把逻辑视图名解析为具体的 View，通过这种策略模式，很容易更换其他视图技术；
6、View——>渲染，View 会根据传进来的 Model 模型数据进行渲染，此处的 Model 实际是一个 Map 数据结构，因此很容易支持其他视图技术；
7、返回控制权给 DispatcherServlet，由 DispatcherServlet 返回响应给用户，到此一个流程结束。
下边两个组件通常情况下需要开发：
Handler：处理器，即后端控制器用 controller 表示。
View：视图，即展示给用户的界面，视图中通常需要标签语言展示模型数据。
在将 SpringMVC 之前我们先来看一下什么是 MVC 模式
MVC：MVC 是一种设计模式
MVC 的原理图：



分析：
M-Model 模型（完成业务逻辑：有 javaBean 构成，service+dao+entity）
V-View 视图（做界面的展示 jsp，html……）
C-Controller 控制器（接收请求—>调用模型—>根据结果派发页面）
springMVC 是什么：
springMVC 是一个 MVC 的开源框架，springMVC=struts2+spring，springMVC 就相当于是 Struts2 加上 sring 的整合，但是这里有一个疑惑就是，springMVC 和 spring 是什么样的关系呢？这个在百度百科上有一个很好的解释：意思是说，springMVC 是 spring 的一个后续产品，其实就是 spring 在原有基础上，又提供了 web 应用的 MVC 模块，可以简单的把 springMVC 理解为是 spring 的一个模块（类似 AOP，IOC 这样的模块），网络上经常会说 springMVC 和 spring 无缝集成，其实 springMVC 就是 spring 的一个子模块，所以根本不需要同 spring 进行整合。
SpringMVC 的原理图：



看到这个图大家可能会有很多的疑惑，现在我们来看一下这个图的步骤：（可以对比 MVC 的原理图进行理解）

第一步:用户发起请求到前端控制器（DispatcherServlet）

第二步：前端控制器请求处理器映射器（HandlerMappering）去查找处理器（Handle）：通过 xml 配置或者注解进行查找

第三步：找到以后处理器映射器（HandlerMappering）像前端控制器返回执行链（HandlerExecutionChain）

第四步：前端控制器（DispatcherServlet）调用处理器适配器（HandlerAdapter）去执行处理器（Handler）

第五步：处理器适配器去执行 Handler

第六步：Handler 执行完给处理器适配器返回 ModelAndView

第七步：处理器适配器向前端控制器返回 ModelAndView

第八步：前端控制器请求视图解析器（ViewResolver）去进行视图解析

第九步：视图解析器像前端控制器返回 View

第十步：前端控制器对视图进行渲染

第十一步：前端控制器向用户响应结果

看到这些步骤我相信大家很感觉非常的乱，这是正常的，但是这里主要是要大家理解 springMVC 中的几个组件：

前端控制器（DispatcherServlet）：接收请求，响应结果，相当于电脑的 CPU。

处理器映射器（HandlerMapping）：根据 URL 去查找处理器

处理器（Handler）：（需要程序员去写代码处理逻辑的）

处理器适配器（HandlerAdapter）：会把处理器包装成适配器，这样就可以支持多种类型的处理器，类比笔记本的适配器（适配器模式的应用）

视图解析器（ViewResovler）：进行视图解析，多返回的字符串，进行处理，可以解析成对应的页面。

线程

30、创建线程的方式及实现

线程被称为轻量级进程，是程序执行的最小单位，它是指在程序执行过程中，能够执行代码的一个执行单位。每个程序程序都至少有一个线程，也即是程序本身。

线程状态：

（1）新建（New）：创建后尚未启动的线程处于这种状态

（2）运行（Runable）：Runable 包括了操作系统线程状态的 Running 和 Ready，也就是处于此状态的线程有可能正在执行，也有可能正在等待着 CPU 为它分配执行时间。

（3）等待（Wating）：处于这种状态的线程不会被分配 CPU 执行时间。等待状态又分为无限期等待和有限期等待，处于无限期等待的线程需要被其他线程显示地唤醒，没有设置 Timeout 参数的 Object.wait()、没有设置 Timeout 参数的 Thread.join()方法都会使线程进入无限期等待状态；有限期等待状态无须等待被其他线程显示地唤醒，在一定时间之后它们会由系统自动唤醒，Thread.sleep()、设置了 Timeout 参数的 Object.wait()、设置了 Timeout 参数的 Thread.join()方法都会使线程进入有限期等待状态。

（4）阻塞（Blocked）：线程被阻塞了，“阻塞状态”与”等待状态“的区别是：”阻塞状态“在等待着获取到一个排他锁，这个时间将在另外一个线程放弃这个锁的时候发生；而”等待状态“则是在等待一段时间或者唤醒动作的发生。在程序等待进入同步区域的时候，线程将进入这种状态。

（5）结束（Terminated）：已终止线程的线程状态，线程已经结束执行。

线程同步方法

线程有 4 中同步方法，分别为 wait()、sleep()、notify()和 notifyAll()。

wait()：使线程处于一种等待状态，释放所持有的对象锁。

sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用它时要捕获 InterruptedException 异常，不释放对象锁。

notify()：唤醒一个正在等待状态的线程。注意调用此方法时，并不能确切知道唤醒的是哪一个等待状态的线程，是由 JVM 来决定唤醒哪个线程，不是由线程优先级决定的。

notifyAll()：唤醒所有等待状态的线程，注意并不是给所有唤醒线程一个对象锁，而是让它们竞争。

创建线程呢个的 4 种方式：

（1）继承 Thread 类

```
//继承 Thread 类来创建线程
public class ThreadTest {
    public static void main(String[] args) {
        //设置线程名字
        Thread.currentThread().setName("main thread");
        MyThread myThread = new MyThread();
        myThread.setName("子线程:");
        //开启线程
        myThread.start();
        for(int i = 0;i<5;i++){
            System.out.println(Thread.currentThread().getName() + i);
        }
```

```
    }
}

class MyThread extends Thread{
    //重写 run()方法
    public void run(){
        for(int i = 0;i < 10; i++){
            System.out.println(Thread.currentThread().getName() + i);
        }
    }
}
```

（2）实现 Runnable 接口

```
//实现 Runnable 接口
public class RunnableTest {
    public static void main(String[] args) {
        //设置线程名字
        Thread.currentThread().setName("main thread:");
        Thread thread = new Thread(new MyRunnable());
        thread.setName("子线程:");
        //开启线程
        thread.start();
        for(int i = 0; i <5;i++){
            System.out.println(Thread.currentThread().getName() + i);
        }
    }
}

class MyRunnable implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(Thread.currentThread().getName() + i);
        }
    }
}
```

（3）实现 Callable 接口

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.FutureTask;
//实现 Callable 接口
public class CallableTest {

    public static void main(String[] args) {
        //执行 Callable 方式，需要 FutureTask 实现实现，用于接收运算结果
```

```
        FutureTask<Integer> futureTask = new FutureTask<Integer>(new MyCallable());
        new Thread(futureTask).start();
        //接收线程运算后的结果
        try {
            Integer sum = futureTask.get();
            System.out.println(sum);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
        return sum;
    }
}
```

（4）线程池

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
//线程池实现
public class ThreadPoolExecutorTest {
    public static void main(String[] args) {
        //创建线程池
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        ThreadPool threadPool = new ThreadPool();
        for(int i =0;i<5;i++){
            //为线程池分配任务
            executorService.submit(threadPool);
        }
        //关闭线程池
        executorService.shutdown();
    }
}

class ThreadPool implements Runnable {
    @Override
```

```
    public void run() {
        for(int i = 0 ;i<10;i++){
            System.out.println(Thread.currentThread().getName() + ":" + i);
        }
    }
}
```

Executors 提供了一系列工厂方法用于创先线程池，返回的线程池都实现了 ExecutorService 接口。

public static ExecutorService newFixedThreadPool(int nThreads)

创建固定数目线程的线程池。

public static ExecutorService newCachedThreadPool()

创建一个可缓存的线程池，调用 execute 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线　程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。

public static ExecutorService newSingleThreadExecutor()

创建一个单线程化的 Executor。

public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)

创建一个支持定时及周期性的任务执行的线程池，多数情况下可用来替代 Timer 类。

31、sleep() 、join（）、yield（）有什么区别

sleep()

sleep()方法需要指定等待的时间，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态，该方法既可以让其他同优先级或者高优先级的线程得到执行的机会，也可以让低优先级的线程得到执行机会。但是 sleep()方法不会释放“锁标志”，也就是说如果有 synchronized 同步块，其他线程仍然不能访问共享数据。

wait()

wait()方法需要和 notify()及 notifyAll()两个方法一起介绍，这三个方法用于协调多个线程对共享数据的存取，所以必须在 synchronized 语句块内使用，也就是说，调用 wait()，notify()和 notifyAll()的任务在调用这些方法前必须拥有对象的锁。注意，它们都是 Object 类的方法，而不是 Thread 类的方法。

wait()方法与 sleep()方法的不同之处在于，wait()方法会释放对象的“锁标志”。当调用某一对象的 wait()方法后，会使当前线程暂停执行，并将当前线程放入对象等待池中，直到调用了 notify()方法后，将从对象等待池中移出任意一个线程并放入锁标志等待池中，只有锁标志等待池中的线程可以获取锁标志，它们随时准备争夺锁的拥有权。当调用了某个对象的 notifyAll()方法，会将对象等待池中的所有线程都移动到该对象的锁标志等待池。

除了使用 notify()和 notifyAll()方法，还可以使用带毫秒参数的 wait(long timeout)方法，效果是在延迟 timeout 毫秒后，被暂停的线程将被恢复到锁标志等待池。

此外，wait()，notify()及 notifyAll()只能在 synchronized 语句中使用，但是如果使用的是 ReenTrantLock 实现同步，该如何达到这三个方法的效果呢？解决方法是使用 ReenTrantLock.newCondition()获取一个 Condition 类对象，然后 Condition 的 await()，signal()以及 signalAll()分别对应上面的三个方法。

yield()

yield()方法和 sleep()方法类似，也不会释放“锁标志”，区别在于，它没有参数，即 yield()方法只是使当前线程重新回到可执行状态，所以执行 yield()的线程有可能在进入到可执行状

态后马上又被执行，另外 yield()方法只能使同优先级或者高优先级的线程得到执行机会，这也和 sleep()方法不同。

join()

join()方法会使当前线程等待调用 join()方法的线程结束后才能继续执行，例如：

```
package concurrent;
public class TestJoin {
    public static void main(String[] args) {
        Thread thread = new Thread(new JoinDemo());
        thread.start();

        for (int i = 0; i < 20; i++) {
            System.out.println("主线程第" + i + "次执行！");
            if (i >= 2)
                try {
                    // t1 线程合并到主线程中，主线程停止执行过程，转而执行 t1 线程，直到 t1 执行完毕后继续。
                    thread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }
    }
}
class JoinDemo implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("线程 1 第" + i + "次执行！");
        }
    }
}
```

32、说说 CountDownLatch 原理

CountDownLatch CountDownLatch 用过很多次了，突然有点好奇，它是如何实现阻塞线程的，猜想是否跟 LockSupport 有关。今天浏览了一下它的源码，发现其实现是十分简单的，只是简单的继承了 AbstractQueuedSynchronizer，便实现了其功能。

实现原理：让需要的暂时阻塞的线程，进入一个死循环里面，得到某个条件后再退出循环，以此实现阻塞当前线程的效果。

下面从构造方法开始，一步步解释实现的原理：

一、构造方法

下面是实现的源码，非常简短，主要是创建了一个 Sync 对象。

```
public CountDownLatch(int count) {
        if (count < 0) throw new IllegalArgumentException("count < 0");
        this.sync = new Sync(count);
```

```
    }
```

二、Sync 对象

```
private static final class Sync extends AbstractQueuedSynchronizer {
        private static final long serialVersionUID = 4982264981922014374L;
        Sync(int count) {
            setState(count);
        }
        int getCount() {
            return getState();
        }
        protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
        }
        protected boolean tryReleaseShared(int releases) {
            // Decrement count; signal when transition to zero
            for (;;) {
                int c = getState();
                if (c == 0)
                    return false;
                int nextc = c-1;
                if (compareAndSetState(c, nextc))
                    return nextc == 0;
            }
        }
    }
```

假设我们是这样创建的：new CountDownLatch(5)。其实也就相当于 new Sync(5)，相当于 setState(5)。setState 我们可以暂时理解为设置一个计数器，当前计数器初始值为 5。

tryAcquireShared 方法其实就是判断一下当前计数器的值，是否为 0 了，如果为 0 的话返回 1（返回 1 的时候，就表明当前线程可以继续往下走了，不再停留在调用 countDownLatch.await()这个方法的地方）。

tryReleaseShared 方法就是利用 CAS 的方式，对计数器进行减一的操作，而我们实际上每次调用 countDownLatch.countDown()方法的时候，最终都会调到这个方法，对计数器进行减一操作，一直减到 0 为止。

稍微跑偏了一点，我们看看调用 countDownLatch.await()的时候，做了些什么。

三、countDownLatch.await()

```
public void await() throws InterruptedException {
        sync.acquireSharedInterruptibly(1);
    }
```

代码很简单，就一句话（注意 acquireSharedInterruptibly（）方法是抽象类：AbstractQueuedSynchronizer 的一个方法，我们上面提到的 Sync 继承了它），我们跟踪源码，继续往下看：

四、acquireSharedInterruptibly(int arg)

```
public final void acquireSharedInterruptibly(int arg)
```

```
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg);
    }
```

源码也是非常简单的，首先判断了一下，当前线程是否有被中断，如果没有的话，就调用 tryAcquireShared(int acquires)方法，判断一下当前线程是否还需要“阻塞”。其实这里调用的 tryAcquireShared 方法，就是我们上面提到的 java.util.concurrent.CountDownLatch.Sync.tryAcquireShared(int)这个方法。

当然，在一开始我们没有调用过 countDownLatch.countDown() 方法时，这里 tryAcquireShared 方法肯定是会返回 1 的，因为会进入到 doAcquireSharedInterruptibly 方法。

五、doAcquireSharedInterruptibly(int arg)

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```


这个时候，我们应该对于 countDownLatch.await()方法是怎么“阻塞”当前线程的，已经非常明白了。其实说白了，就是当你调用了 countDownLatch.await()方法后，你当前线程就会进入了一个死循环当中，在这个死循环里面，会不断的进行判断，通过调用 tryAcquireShared 方法，不断判断我们上面说的那个计数器，看看它的值是否为 0 了（为 0 的时候，其实就是我们调用了足够多次数的 countDownLatch.countDown（）方法的时候），如果是为 0 的话，tryAcquireShared 就会返回 1，代码也会进入到图中的红框部分，然后跳出了循环，也就不再“阻塞”当前线程了。需要注意的是，说是在不停的循环，其实也并非在不停的执行 for 循环里面的内容，因为在后面调用 parkAndCheckInterrupt（）方法时，在这个方法里面是会调用 LockSupport.park(this);，来禁用当前线程的。

五、关于 AbstractQueuedSynchronizer

看到这里，如果各位对 AbstractQueuedSynchronizer 没有了解过的话，可能代码还是看

得有点迷糊，这是一篇我觉得解释得非常好的文章，大家可以看一下：http://ifeve.com/introduce-abstractqueuedsynchronizer/

33、说说 CyclicBarrier 原理

CyclicBarrier 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。线程进入屏障通过 CyclicBarrier 的 await() 方法。

CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier 还提供一个更高级的构造函数 CyclicBarrier(int parties, Runnable barrierAction)，用于在线程到达屏障时，优先执行 barrierAction 这个 Runnable 对象，方便处理更复杂的业务场景。

构造函数

```
public CyclicBarrier(int parties) {
    this(parties, null);
}
public int getParties() {
    return parties;
}
```

实现原理：在 CyclicBarrier 的内部定义了一个 Lock 对象，每当一个线程调用 CyclicBarrier 的 await 方法时，将剩余拦截的线程数减 1，然后判断剩余拦截数是否为 0，如果不是，进入 Lock 对象的条件队列等待。如果是，执行 barrierAction 对象的 Runnable 方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从 await 方法返回，再从 CyclicBarrier 的 await 方法中返回。

实现原理：在 CyclicBarrier 的内部定义了一个 Lock 对象，每当一个线程调用 CyclicBarrier 的 await 方法时，将剩余拦截的线程数减 1，然后判断剩余拦截数是否为 0，如果不是，进入 Lock 对象的条件队列等待。如果是，执行 barrierAction 对象的 Runnable 方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从 await 方法返回，再从 CyclicBarrier 的 await 方法中返回。

await 源码：

```
public int await() throws InterruptedException, BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

dowait 源码：

```
private int dowait(boolean timed, long nanos)
    throws InterruptedException, BrokenBarrierException,
           TimeoutException {
    final ReentrantLock lock = this.lock;
    lock.lock();
```

```
try {
    final Generation g = generation;
    if (g.broken)
        throw new BrokenBarrierException();
    if (Thread.interrupted()) {
        breakBarrier();
        throw new InterruptedException();
    }
    int index = --count;
    if (index == 0) {    // tripped
        boolean ranAction = false;
        try {
            final Runnable command = barrierCommand;
            if (command != null)
                command.run();
            ranAction = true;
            nextGeneration();
            return 0;
        } finally {
            if (!ranAction)
                breakBarrier();
        }
    }
    // loop until tripped, broken, interrupted, or timed out
    for (;;) {
        try {
            if (!timed)
                trip.await();
            else if (nanos > 0L)
                nanos = trip.awaitNanos(nanos);
        } catch (InterruptedException ie) {
            if (g == generation && ! g.broken) {
                breakBarrier();
                throw ie;
            } else {
                // We're about to finish waiting even if we had not
                // been interrupted, so this interrupt is deemed to
                // "belong" to subsequent execution.
                Thread.currentThread().interrupt();
            }
        }
        if (g.broken)
            throw new BrokenBarrierException();
```

```
                if (g != generation)
                    return index;

                if (timed && nanos <= 0L) {
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

当最后一个线程到达屏障点，也就是执行 dowait 方法时，会在 return 0 返回之前调用 finally 块中的 breakBarrier 方法。

breakBarrier 源代码：

```
    private void breakBarrier() {
        generation.broken = true;
        count = parties;
        trip.signalAll();
    }
```

CyclicBarrier 主要用于一组线程之间的相互等待，而 CountDownLatch 一般用于一组线程等待另一组些线程。实际上可以通过 CountDownLatch 的 countDown()和 await()来实现 CyclicBarrier 的功能。即 CountDownLatch 中的 countDown()+await() = CyclicBarrier 中的 await()。注意：在一个线程中先调用 countDown()，然后调用 await()。

其它方法：CycliBarrier 对象可以重复使用，重用之前应当调用 CyclicBarrier 对象的 reset 方法。

reset 源码：

```
    public void reset() {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
            breakBarrier();     // break the current generation
            nextGeneration(); // start a new generation
        } finally {
            lock.unlock();
        }
    }
    import java.util.Random;
    import java.util.concurrent.BrokenBarrierException;
    import java.util.concurrent.CyclicBarrier;
    import java.util.concurrent.ExecutorService;
    import java.util.concurrent.Executors;
    public class CyclicBarrierDemo {
        private CyclicBarrier cb = new CyclicBarrier(4);
```

```
    private Random rnd = new Random();
    class TaskDemo implements Runnable{
        private String id;
        TaskDemo(String id){
            this.id = id;
        }
        @Override
        public void run(){
            try {
                Thread.sleep(rnd.nextInt(1000));
                System.out.println("Thread " + id + " will wait");
                cb.await();
                System.out.println("-------Thread " + id + " is over");
            } catch (InterruptedException e) {
            } catch (BrokenBarrierException e) {
            }
        }
    }
    public static void main(String[] args){
        CyclicBarrierDemo cbd = new CyclicBarrierDemo();
        ExecutorService es = Executors.newCachedThreadPool();
        es.submit(cbd.new TaskDemo("a"));
        es.submit(cbd.new TaskDemo("b"));
        es.submit(cbd.new TaskDemo("c"));
        es.submit(cbd.new TaskDemo("d"));
        es.shutdown();
    }
}
```

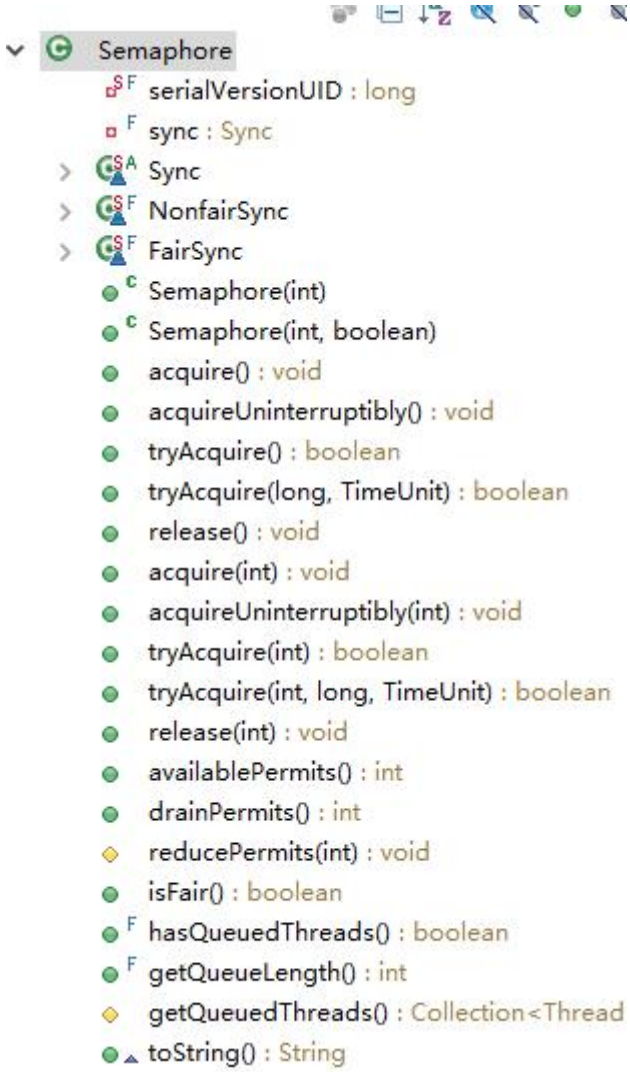
34、说说 Semaphore 原理

Semaphore 直译为信号。实际上 Semaphore 可以看做是一个信号的集合。不同的线程能够从 Semaphore 中获取若干个信号量。当 Semaphore 对象持有的信号量不足时，尝试从 Semaphore 中获取信号的线程将会阻塞。直到其他线程将信号量释放以后，阻塞的线程会被唤醒，重新尝试获取信号量。

Semaphore 使用：

Semaphore 实现了信号量，概念上讲，一个信号量相当于持有一些许可（permits），线程可以调用 Semaphore 对象的 acquire()方法获取一个许可，调用 release()来归还一个许可。

信号量一般用来限制访问资源的线程数量。



1 构造方法：

Semaphore 有两个构造方法 Semaphore(int)、Semaphore(int,boolean)，参数中的 int 表示该信号量拥有的许可数量，boolean 表示获取许可的时候是否是公平的，如果是公平的那么，当有多个线程要获取许可时，会按照线程来的先后顺序分配许可，否则，线程获得许可的顺序是不定的。

2 获取许可

可以使用 acquire()、acquire(int)、tryAcquire()等去获取许可，其中 int 参数表示一次性要获取几个许可，默认为 1 个，acquire 方法在没有许可的情况下，要获取许可的线程会阻塞，而 tryAcquire()方法在没有许可的情况下会立即返回 false，要获取许可的线程不会阻塞。

3 释放许可

线程可调用 release()、release(int)来释放（归还）许可，注意一个线程调用 release()之前并不要求一定要调用了 acquire

35、说说 Exchanger 原理

当一个线程到达 exchange 调用点时，如果它的伙伴线程此前已经调用了此方法，那么它的伙伴会被调度唤醒并与之进行对象交换，然后各自返回。如果它的伙伴还没到达交换点，那么当前线程将会被挂起，直至伙伴线程到达——完成交换正常返回；或者当前线程被中断——抛出中断异常；又或者是等候超时——抛出超时异常。

1.实现原理

Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger 用于进行线程间的数据交换。它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。这两个线程通过 exchange 方法交换数据， 如果第一个线程先执行 exchange 方法，它会一直等待第二个线程也执行 exchange，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。因此使用 Exchanger 的重点是成对的线程使用 exchange()方法，当有一对线程达到了同步点，就会进行交换数据。因此该工具类的线程对象是成对的。

Exchanger 类提供了两个方法，String exchange(V x):用于交换，启动交换并等待另一个线程调用 exchange；String exchange(V x,long timeout,TimeUnit unit)：用于交换，启动交换并等待另一个线程调用 exchange，并且设置最大等待时间，当等待时间超过 timeout 便停止等待。

2.实例讲解

通过以上的原理，可以知道使用 Exchanger 类的核心便是 exchange()方法的使用，接下来通过一个例子来使的该工具类的用途更加清晰。该例子主要讲解的是前段时间 NBA 交易截止日的交易。

```
public class ExchangerDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();
        final Exchanger exchanger = new Exchanger();
        executor.execute(new Runnable() {
            String data1 = "克拉克森，小拉里南斯";
            @Override
            public void run() {
                nbaTrade(data1, exchanger);
            }
        });
        executor.execute(new Runnable() {
            String data1 = "格里芬";
            @Override
            public void run() {
                nbaTrade(data1, exchanger);
            }
        });
        executor.execute(new Runnable() {
            String data1 = "哈里斯";
            @Override
            public void run() {
                nbaTrade(data1, exchanger);
            }
        });
        executor.execute(new Runnable() {
            String data1 = "以赛亚托马斯，弗莱";
            @Override
```

```
                public void run() {
                    nbaTrade(data1, exchanger);
                }
            });
            executor.shutdown();
        }
        private static void nbaTrade(String data1, Exchanger exchanger) {
            try {
                System.out.println(Thread.currentThread().getName() + "在交易截止之前把
" + data1 + "  交易出去");
                Thread.sleep((long) (Math.random() * 1000));
                String data2 = (String) exchanger.exchange(data1);
                System.out.println(Thread.currentThread().getName() + "交易得到" + data2);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

当两个线程之间出现数据交换的情况，可以使用 Exchanger 工具类实现数据交换。注意 exchange 方法的含义，以及触发数据交换的条件。

36、说说 CountDownLatch 与 CyclicBarrier 区别

(01) CountDownLatch 的作用是允许 1 或 N 个线程等待其他线程完成执行;而 CyclicBarrier 则是允许 N 个线程相互等待。

(02) CountDownLatch 的计数器无法被重置;CyclicBarrier 的计数器可以被重置后使用,因此它被称为是循环的 barrier。

CountDownLatch	CyclicBarrier
减计数方式	加计数方式
计算为0时释放所有等待的线程	计数达到指定值时释放所有等待线程
计数为0时，无法重置	计数达到指定值时，计数置为0重新开始
调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响	调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞
不可重复利用	可重复利用

CountDownLatch 是一个同步的辅助类，允许一个或多个线程，等待其他一组线程完成操作，再继续执行。

CyclicBarrier 是一个同步的辅助类，允许一组线程相互之间等待，达到一个共同点，再继续执行。

他们都是:Synchronization aid，我把它翻译成同步辅助器，既然是辅助工具，怎么使用啊？哪些场景使用啊？

个人理解：CountDownLatch:我把他理解成倒计时锁

场景还原：一年级期末考试要开始了，监考老师发下去试卷，然后坐在讲台旁边玩着手机等待着学生答题，有的学生提前交了试卷，并约起打球了，等到最后一个学生交卷了，老

师开始整理试卷，贴封条，下班，陪老婆孩子去了。

个人理解：CyclicBarrier:可看成是个障碍，所有的线程必须到齐后才能一起通过这个障碍.

场景还原：以前公司组织户外拓展活动，帮助团队建设，其中最重要一个项目就是全体员工（包括女同事，BOSS）在完成其他项目时，到达一个高达四米的高墙没有任何抓点，要求所有人，一个不能少的越过高墙，才能继续进行其他项目。

37、ThreadLocal 原理分析

ThreadLocal 提供了线程本地变量，它可以保证访问到的变量属于当前线程，每个线程都保存有一个变量副本，每个线程的变量都不同。ThreadLocal 相当于提供了一种线程隔离，将变量与线程相绑定。

ThreadLocal 原理分析:

首先，在每个线程 Thread 内部有一个 ThreadLocal.ThreadLocalMap 类型的成员变量 threadLocals，这个 threadLocals 就是用来存储实际的变量副本的，键值为当前 ThreadLocal 变量，value 为变量副本（即 T 类型的变量）。

初始时，在 Thread 里面，threadLocals 为空，当通过 ThreadLocal 变量调用 get()方法或者 set()方法，就会对 Thread 类中的 threadLocals 进行初始化，并且以当前 ThreadLocal 变量为键值，以 ThreadLocal 要保存的副本变量为 value，存到 threadLocals。 然后在当前线程里面，如果要使用副本变量，就可以通过 get 方法在 threadLocals 里面查找。

1）实际的通过 ThreadLocal 创建的副本是存储在每个线程自己的 threadLocals 中的；

2）为何 threadLocals 的类型 ThreadLocalMap 的键值为 ThreadLocal 对象，因为每个线程中可有多个 threadLocal 变量，就像上面代码中的 longLocal 和 stringLocal；

3）在进行 get 之前，必须先 set，否则会报空指针异常；

如果想在 get 之前不需要调用 set 就能正常访问的话，必须重写 initialValue()方法。

因为在上面的代码分析过程中，我们发现如果没有先 set 的话，即在 map 中查找不到对应的存储，则会通过调用 setInitialValue 方法

返回 i，而在 setInitialValue 方法中，有一个语句是 T value = initialValue()， 而默认情况下，initialValue 方法返回的是 null。

ThreadLocal 的应用场景：

最常见的 ThreadLocal 使用场景为 用来解决 数据库连接、Session 管理等。

```
public class Document {
static ThreadLocal<Long> longLocal = new ThreadLocal<Long>();
    static ThreadLocal<String> stringLocal = new ThreadLocal<String>();
    public static void set() {
        longLocal.set(Thread.currentThread().getId());
        stringLocal.set(Thread.currentThread().getName());
    }
    public static long getLong() {
        return longLocal.get();
    }
    public static String getString() {
        return stringLocal.get();
    }
    public static void main(String[] args) throws InterruptedException {
```

```
        final Document test = new Document();
        Document.set();
        System.out.println("--------------------------1");
        System.out.println(test.getLong());
        System.out.println(test.getString());
        Thread thread1 = new Thread(() -> {
            test.set();
            System.out.println("--------------------------2");
            System.out.println(test.getLong());
            System.out.println(test.getString());
        });
        thread1.start();
        thread1.join();//线程 thread1 加进来，先执行 2，执行完后再执行 3
        System.out.println("--------------------------3");
        System.out.println(test.getLong());
        System.out.println(test.getString());
        /**
         最后打印结果为：
         -------------------------1
         1
         main
         -------------------------2
         11
         Thread-0
         -------------------------3
         1
         main
```

在 main 线程中和 thread1 线程中，longLocal 保存的副本值和 stringLocal 保存的副本值都不一样。

最后一次在 main 线程再次打印副本值是为了证明在 main 线程中和 thread1 线程中的副本值确实是不同的。

```
        */
    }
}
```

38、 讲讲线程池的实现原理

当提交一个新任务到线程池时，线程池的处理流程如下。

1）线程池判断核心线程池里的线程是否都在执行任务。如果不是，则创建一个新的工作线程来执行任务。如果核心线程池里的线程都在执行任务，则进入下个流程。

2）线程池判断工作队列是否已经满。如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3）线程池判断线程池的线程是否都处于工作状态。如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

39、线程池的几种方式

在 Executors 类里面提供了一些静态工厂，生成一些常用的线程池。

1、newFixedThreadPool：创建固定大小的线程池。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

2、newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。

3、newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

4、newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

5、newSingleThreadScheduledExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

40、线程的生命周期

1.线程的生命周期

线程是一个动态执行的过程，它也有一个从产生到死亡的过程。

(1)生命周期的五种状态

新建（new Thread）

当创建 Thread 类的一个实例（对象）时，此线程进入新建状态（未被启动）。

例如：Thread　t1=new Thread();

就绪（runnable）

线程已经被启动，正在等待被分配给 CPU 时间片，也就是说此时线程正在就绪队列中排队等候得到 CPU 资源。例如：t1.start();

运行（running）

线程获得 CPU 资源正在执行任务（run()方法），此时除非此线程自动放弃 CPU 资源或者有优先级更高的线程进入，线程将一直运行到结束。

死亡（dead）

当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

自然终止：正常运行 run()方法后终止

异常终止：调用 stop()方法让一个线程终止运行

堵塞（blocked）

由于某种原因导致正在运行的线程让出 CPU 并暂停自己的执行，即进入堵塞状态。

正在睡眠：用 sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。

正在等待：调用 wait()方法。（调用 motify()方法回到就绪状态）

被另一个线程所阻塞：调用 suspend()方法。（调用 resume()方法恢复）

2.常用方法

void run()　创建该类的子类时必须实现的方法

void start() 开启线程的方法

static void sleep(long t) 释放 CPU 的执行权，不释放锁
static void sleep(long millis,int nanos)
final void wait()释放 CPU 的执行权，释放锁
final void notify()
static void yied()可以对当前线程进行临时暂停（让线程将资源释放出来）
3.（1）结束线程原理：就是让 run 方法结束。而 run 方法中通常会定义循环结构，所以只要控制住循环即可
(2)方法----可以 boolean 标记的形式完成，只要在某一情况下将标记改变，让循环停止即可让线程结束
（3）public final void join()//让线程加入执行，执行某一线程 join 方法的线程会被冻结，等待某一线程执行结束，该线程才会恢复到可运行状态
4. 临界资源：多个线程间共享的数据称为临界资源
（1）互斥锁
a.每个对象都对应于一个可称为“互斥锁”的标记，这个标记用来保证在任一时刻，只能有一个线程访问该对象。
b.Java 对象默认是可以被多个线程共用的，只是在需要时才启动“互斥锁”机制，成为专用对象。
c.关键字 synchronized 用来与对象的互斥锁联系
d.当某个对象用 synchronized 修饰时，表明该对象已启动“互斥锁”机制，在任一时刻只能由一个线程访问，即使该线程出现堵塞，该对象的被锁定状态也不会解除，其他线程任不能访问该对象。

锁机制

41、 说说线程安全问题

线程安全是多线程领域的问题，线程安全可以简单理解为一个方法或者一个实例可以在多线程环境中使用而不会出现问题。

在 Java 多线程编程当中，提供了多种实现 Java 线程安全的方式：

最简单的方式，使用 Synchronization 关键字:Java Synchronization 介绍

使用 java.util.concurrent.atomic 包中的原子类，例如 AtomicInteger

使用 java.util.concurrent.locks 包中的锁

使用线程安全的集合 ConcurrentHashMap

使用 volatile 关键字，保证变量可见性（直接从内存读，而不是从线程 cache 读）

42、volatile 实现原理

在 JVM 底层 volatile 是采用“内存屏障”来实现的。

缓存一致性协议（MESI 协议）它确保每个缓存中使用的共享变量的副本是一致的。其核心思想如下：当某个 CPU 在写数据时，如果发现操作的变量是共享变量，则会通知其他 CPU 告知该变量的缓存行是无效的，因此其他 CPU 在读取该变量时，发现其无效会重新从主存中加载数据。

并发编程中的三个概念：原子性问题，可见性问题，有序性问题。

Java 内存模型：原子性，可见性，有序性。

1. volatile 关键字的两层语义

一旦一个共享变量（类的成员变量、类的静态成员变量）被 volatile 修饰之后，那么就具备了两层语义：

1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

2）禁止进行指令重排序

2. 从上面知道 volatile 关键字保证了操作的可见性，但是 volatile 能保证对变量的操作是原子性吗？、

volatile 关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是 volatile 没办法保证对变量的操作的原子性。

3. 保证原子性：

```
public class Test {
    public    int inc = 0;
    （1）
    public synchronized void increase() {
        inc++;
    }
    （2）
        Lock lock = new ReentrantLock();
        public    void increase() {
            lock.lock();
            try {
                inc++;
            } finally{
                lock.unlock();
            }
        }
    （3）public    AtomicInteger inc = new AtomicInteger();
        public    void increase() {
            inc.getAndIncrement();
        }

    public static void main(String[] args) {
        final Test test = new Test();
        for(int i=0;i<10;i++){
            new Thread(){
                public void run() {
                    for(int j=0;j<1000;j++)
                        test.increase();
                };
            }.start();
        }
        while(Thread.activeCount()>1)    //保证前面的线程都执行完
            Thread.yield();
```

```
        System.out.println(test.inc);
    }
}
```

4.在前面提到 volatile 关键字能禁止指令重排序，所以 volatile 能在一定程度上保证有序性。

volatile 关键字禁止指令重排序有两层意思：

1）当程序执行到 volatile 变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；

2）在进行指令优化时，不能将在对 volatile 变量访问的语句放在其后面执行，也不能把 volatile 变量后面的语句放到其前面执行。

5.volatile 的原理和实现机制

前面讲述了源于 volatile 关键字的一些使用，下面我们来探讨一下 volatile 到底如何保证可见性和禁止指令重排序的。

下面这段话摘自《深入理解 Java 虚拟机》：

“观察加入 volatile 关键字和没有加入 volatile 关键字时所生成的汇编代码发现，加入 volatile 关键字时，会多出一个 lock 前缀指令”

lock 前缀指令实际上相当于一个内存屏障（也成内存栅栏），内存屏障会提供 3 个功能：

1）它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；

2）它会强制将对缓存的修改操作立即写入主存；

3）如果是写操作，它会导致其他 CPU 中对应的缓存行无效。

使用 volatile 关键字的场景

6.synchronized 关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而 volatile 关键字在某些情况下性能要优于 synchronized，但是要注意 volatile 关键字是无法替代 synchronized 关键字的，因为 volatile 关键字无法保证操作的原子性。通常来说，使用 volatile 必须具备以下 2 个条件：

1）对变量的写操作不依赖于当前值

2）该变量没有包含在具有其他变量的不变式中

实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

事实上，我的理解就是上面的 2 个条件需要保证操作是原子性操作，才能保证使用 volatile 关键字的程序在并发时能够正确执行。

下面列举几个 Java 中使用 volatile 的几个场景。

状态标志量：

```
volatile boolean flag = false;
while(!flag){
    doSomething();
}
public void setFlag() {
    flag = true;
}
```

double check：

```
class Singleton{
```

```
    private volatile static Singleton instance = null;
    private Singleton() {
    }
    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

43、synchronize 实现原理

同步代码块是使用 monitorenter 和 monitorexit 指令实现的，同步方法（在这看不出来需要看 JVM 底层实现）依靠的是方法修饰符上的 ACC_SYNCHRONIZED 实现。

44、synchronized 与 lock 的区别

一、synchronized 和 lock 的用法区别

（1）synchronized(隐式锁)：在需要同步的对象中加入此控制，synchronized 可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。

（2）lock（显示锁）：需要显示指定起始位置和终止位置。一般使用 ReentrantLock 类做为锁，多个线程中必须要使用一个 ReentrantLock 类做为对 象才能保证锁的生效。且在加锁和解锁处需要通过 lock()和 unlock()显示指出。所以一般会在 finally 块中写 unlock()以防死锁。

二、synchronized 和 lock 性能区别

synchronized 是托管给 JVM 执行的，而 lock 是 java 写的控制锁的代码。在 Java1.5 中，synchronize 是性能低效的。因为 这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用 Java 提供的 Lock 对象，性能更高一些。但 是到了 Java1.6，发生了变化。synchronize 在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致 在 Java1.6 上 synchronize 的性能并不比 Lock 差。

三、synchronized 和 lock 机制区别

（1）synchronized 原始采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁意味着其 他线程只能依靠阻塞来等待线程释放锁。

（2）Lock 用的是乐观锁方式。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。乐观锁实现的机制就 是 CAS 操作（Compare and Swap）。

1、ReentrantLock 拥有 Synchronized 相同的并发性和内存语义，此外还多了 锁投票，定时锁等候和中断锁等候

线程 A 和 B 都要获取对象 O 的锁定，假设 A 获取了对象 O 锁，B 将等待 A 释放对 O 的锁定，

如果使用 synchronized ，如果 A 不释放，B 将一直等下去，不能被中断

如果 使用 ReentrantLock，如果 A 不释放，可以使 B 在等待了足够长的时间以后，中断等待，而干别的事情

ReentrantLock 获取锁定与三种方式：

a) lock(), 如果获取了锁立即返回，如果别的线程持有锁，当前线程则一直处于休眠状态，直到获取锁

b) tryLock(), 如果获取了锁立即返回 true，如果别的线程正持有锁，立即返回 false；

c)tryLock(long timeout,TimeUnit unit)， 如果获取了锁定立即返回 true，如果别的线程正持有锁，会等待参数给定的时间，在等待的过程中，如果获取了锁定，就返回 true，如果等待超时，返回 false；

d) lockInterruptibly:如果获取了锁定立即返回，如果没有获取锁定，当前线程处于休眠状态，直到或者锁定，或者当前线程被别的线程中断

2、synchronized 是在 JVM 层面上实现的，不但可以通过一些监控工具监控 synchronized 的锁定，而且在代码执行时出现异常，JVM 会自动释放锁定，但是使用 Lock 则不行，lock 是通过代码实现的，要保证锁定一定会被释放，就必须将 unLock()放到 finally{}中

3、在资源竞争不是很激烈的情况下，Synchronized 的性能要优于 ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized 的性能会下降几十倍，但是 ReetrantLock 的性能能维持常态；

5.0 的多线程任务包对于同步的性能方面有了很大的改进，在原有 synchronized 关键字的基础上，又增加了 ReentrantLock，以及各种 Atomic 类。了解其性能的优劣程度，有助与我们在特定的情形下做出正确的选择。

总体的结论先摆出来：

synchronized：

在资源竞争不是很激烈的情况下，偶尔会有同步的情形下，synchronized 是很合适的。原因在于，编译程序通常会尽可能的进行优化 synchronize，另外可读性非常好，不管用没用过 5.0 多线程包的程序员都能理解。

ReentrantLock:

ReentrantLock 提供了多样化的同步，比如有时间限制的同步，可以被 Interrupt 的同步（synchronized 的同步是不能 Interrupt 的）等。在资源竞争不激烈的情形下，性能稍微比 synchronized 差点点。但是当同步非常激烈的时候，synchronized 的性能一下子能下降好几十倍。而 ReentrantLock 确还能维持常态。

Atomic:

和上面的类似，不激烈情况下，性能比 synchronized 略逊，而激烈的时候，也能维持常态。激烈的时候，Atomic 的性能会优于 ReentrantLock 一倍左右。但是其有一个缺点，就是只能同步一个值，一段代码中只能出现一个 Atomic 的变量，多于一个同步无效。因为他不能在多个 Atomic 之间同步。

45、CAS 乐观锁

所谓原子操作类，指的是 java.util.concurrent.atomic 包下，一系列以 Atomic 开头的包装类。例如 AtomicBoolean，AtomicInteger，AtomicLong。它们分别用于 Boolean，Integer，Long 类型的原子性操作。

CAS 是项乐观锁技术，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其它线程都失败，失败的线程并不会被挂起，而是被告知这次竞争

中失败，并可以再次尝试。

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。（在 CAS 的一些特殊情况下将仅返回 CAS 是否成功，而不提取当前值。）CAS 有效地说明了“我认为位置 V 应该包含值 A；如果包含该值，则将 B 放到这个位置；否则，不要更改该位置，只告诉我这个位置现在的值即可。”这其实和乐观锁的冲突检查+数据更新的原理是一样的。

CAS 的缺点：

1.CPU 开销较大

在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给 CPU 带来很大的压力。

2.不能保证代码块的原子性

CAS 机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证 3 个变量共同进行原子性的更新，就不得不使用 Synchronized 了。

46、ABA 问题

CAS 会导致“ABA 问题”。

CAS 算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但是不代表这个过程就是没有问题的。

部分乐观锁的实现是通过版本号（version）的方式来解决 ABA 问题，乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败。因为每次操作的版本号都会随之增加，所以不会出现 ABA 问题，因为版本号只会增加不会减少。

47、乐观锁的业务场景及实现方式

乐观锁（Optimistic Lock）：

每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

框架篇（spring）

1. BeanFactory 和 ApplicationContext 有什么区别

一、BeanFactory 和 ApplicationContext

Bean 工厂（com.springframework.beans.factory.BeanFactory）是 Spring 框架最核心的接口，它提供了高级 IoC 的配置机制。

应用上下文（com.springframework.context.ApplicationContext）建立在 BeanFactory 基础之上。几乎所有的应用场合我们都直接使用 ApplicationContext 而非底层的 BeanFactory。

1.1 BeanFactory 的类体系结构

BeanFactory 接口位于类结构树的顶端， 它最主要的方法就是 getBean(StringbeanName)，该方法从容器中返回特定名称的 Bean，BeanFactory 的功能通过其他的接口得到不断扩展。

ListableBeanFactory：该接口定义了访问容器中 Bean 基本信息的若干方法，如查看 Bean 的个数、获取某一类型 Bean 的配置名、查看容器中是否包括某一 Bean 等方法；

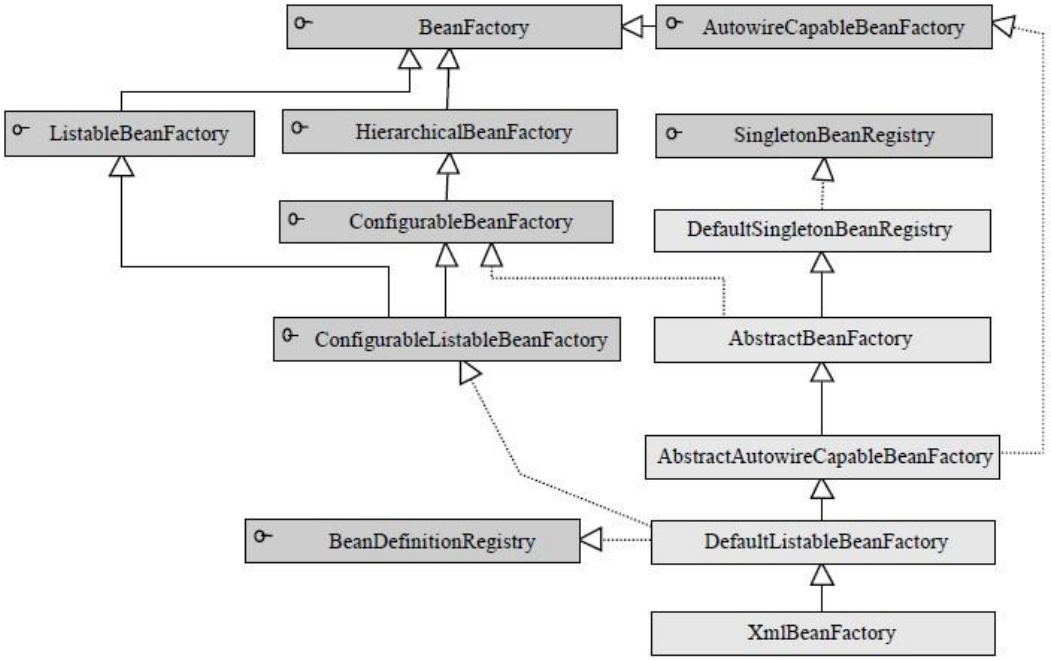
HierarchicalBeanFactory：父子级联 IoC 容器的接口，子容器可以通过接口方法访问父容器；

ConfigurableBeanFactory：是一个重要的接口，增强了 IoC 容器的可定制性，它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法；

AutowireCapableBeanFactory：定义了将容器中的 Bean 按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法；

SingletonBeanRegistry：定义了允许在运行期间向容器注册单实例 Bean 的方法；

BeanDefinitionRegistry：Spring 配置文件中每一个<bean>节点元素在 Spring 容器里都通过一个 BeanDefinition 对象表示，它描述了 Bean 的配置信息。而 BeanDefinitionRegistry 接口提供了向容器手工注册 BeanDefinition 对象的方法。



1.2 ApplicationContext 的类体系结构

ApplicationContext 由 BeanFactory 派生而来，提供了更多面向实际应用的功能。在 BeanFactory 中，很多功能需要以编程的方式实现，而在 ApplicationContext 中则可以通过配置的方式实现。

ApplicationContext 的主要实现类是 ClassPathXmlApplicationContext 和 FileSystemXmlApplicationContext，前者默认从类路径加载配置文件，后者默认从文件系统中装载配置文件。

核心接口包括：

ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了 ApplicationListener 事件监听接口的 Bean 可以接收到容器事件， 并对事件进行响应处理。在 ApplicationContext 抽象实现类 AbstractApplicationContext 中，我们可以发现存在一个 ApplicationEventMulticaster，它负责保存所有监听器，以便在容器产生上下文事件时通知这些事件监听 者。

MessageSource：为应用提供 i18n 国际化消息访问的功能；

ResourcePatternResolver ： 所有 ApplicationContext 实现类都实现了类似于 PathMatchingResourcePatternResolver 的功能，可以通过带前缀的 Ant 风格的资源文件路径装载 Spring 的配置文件。

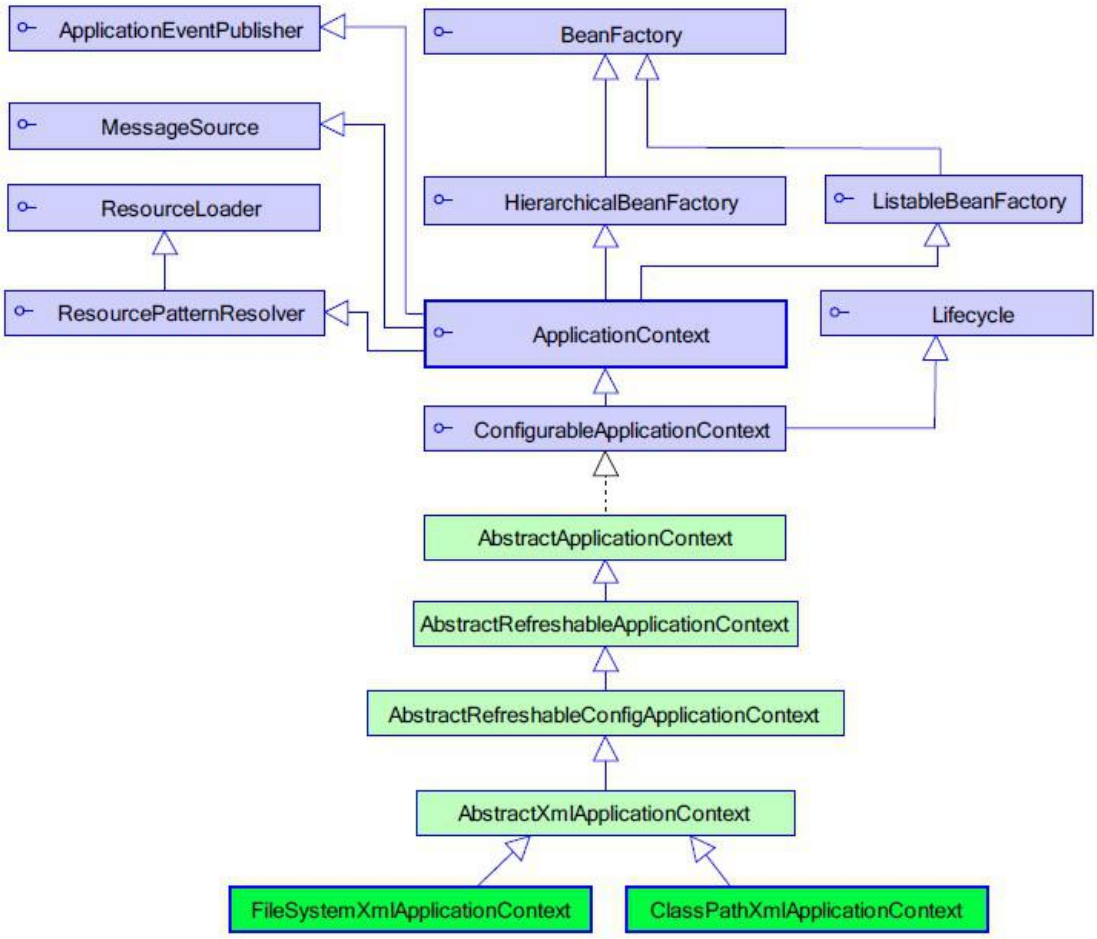
LifeCycle：该接口是 Spring 2.0 加入的，该接口提供了 start()和 stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被 ApplicationContext 实现及具体 Bean 实现，ApplicationContext 会将 start/stop 的信息传递给容器中所有实现了该接口的 Bean，以达到管理和控制 JMX、任务调度等目的。

ConfigurableApplicationContext 扩展于 ApplicationContext，它新增加了两个主要的方法：refresh()和 close()，让 ApplicationContext 具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用 refresh()即可启动应用上下文，在已经启动的状态下，调用 refresh()则清除缓存并重新装载配置信息，而调用 close()则可关闭应用上下文。这些接口方法为容器的控制管理带来了便利。

代码示例：

```
ApplicationContext ctx =new
ClassPathXmlApplicationContext("com/baobaotao/context/beans.xml");
ApplicationContext ctx =new
FileSystemXmlApplicationContext("com/baobaotao/context/beans.xml");
ApplicationContext ctx = new ClassPathXmlApplicationContext(new
String[]{"conf/beans1.xml","conf/beans2.xml"});
```

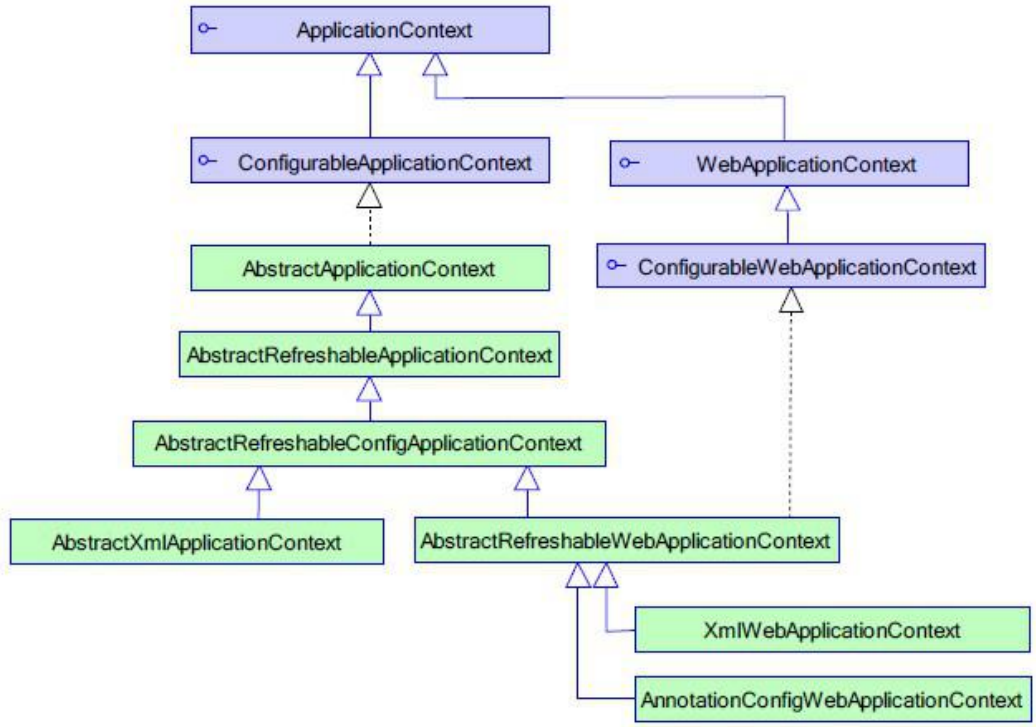
ApplicationContext 的初始化和 BeanFactory 有一个重大的区别：BeanFactory 在初始化容器时，并未实例化 Bean，直到第一次访问某个 Bean 时才实例目标 Bean；而 ApplicationContext 则在初始化应用上下文时就实例化所有单实例的 Bean 。



二、WebApplicationContext 类体系结构

WebApplicationContext 是专门为 Web 应用准备的，它允许从相对于 Web 根目录的路径中装载配置文件完成初始化工作。从 WebApplicationContext 中可以获得 ServletContext 的引用，整个 Web 应用上下文对象将作为属性放置到 ServletContext 中，以便 Web 应用环境可以访问 Spring 应用上下文（ApplicationContext）。Spring 专门为此提供一个工具类 WebApplicationContextUtils，通过该类的 getWebApplicationContext(ServletContext sc)方法，即可以从 ServletContext 中获取 WebApplicationContext 实例。

Spring 2.0 在 WebApplicationContext 中还为 Bean 添加了三个新的作用域：request 作用域、session 作用域和 global session 作用域。而在非 Web 应用的环境下，Bean 只有 singleton 和 prototype 两种作用域。



由于 Web 应用比一般的应用拥有更多的特性，因此 WebApplicationContext 扩展了 ApplicationContext 。 WebApplicationContext 定义了一个常量 ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，在上下文启动 时，WebApplicationContext 实例即以此为键放置在 ServletContext 的属性列表中，因此我们可以直接通过以下语句从 Web 容器中获取

WebApplicationContext：

WebApplicationContext wac = (WebApplicationContext)servletContext.getAttribute(

WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);

2.1 WebApplicationContext 的初启动方式 和 BeanFactory、ApplicationContext 有所区别。

因为 WebApplicationContext 需要 ServletContext 实例，也就是说它必须在拥有 Web 容器的前提下才能完成启动的工作。有过 Web 开发经验的读者都知道可以在 web.xml 中配置自启

动的 Servlet 或定义 Web 容器监听器（ServletContextListener），借助这两者中的任何一个，我们就可以完成启动 Spring Web 应用上下文的工作。

Spring 分别提供了用于启动 WebApplicationContext 的 Servlet 和 Web 容器监听器：

org.springframework.web.context.ContextLoaderServlet；

org.springframework.web.context.ContextLoaderListener。

两者的内部都实现了启动 WebApplicationContext 实例的逻辑，我们只要根据 Web 容器的具体情况选择两者之一，并在 web.xml 中完成配置就可以了。

三、WebApplicationContext 需要使用日志功能

用户可以将 Log4J 的配置文件放置到类路径 WEB-INF/classes 下，这时 Log4J 引擎即可顺利启动。

如果 Log4J 配置文件放置在其他位置，用户还必须在 web.xml 指定 Log4J 配置文件位置。

Spring 为启用 Log4J 引擎提供了两个类似于启动 WebApplicationContext 的实现类：Log4jConfigServlet 和 Log4jConfigListener，

不管采用哪种方式都必须保证能够在装载 Spring 配置文件前先装载 Log4J 配置信息。

四、父子容器

通过 HierarchicalBeanFactory 接口，Spring 的 IoC 容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的 Bean，但父容器不能访问子容器的 Bean。在容器内，Bean 的 id 必须是唯一的，但子容器可以拥有一个和父容器 id 相同的 Bean。父子容器层级体系增强了 Spring 容器架构的扩展性和灵活性，因为第三方可以通过编程的方式，为一个已经存在的容器添加一个或多个特殊用途的子容器，以提供一些额外的功能。

Spring 使用父子容器实现了很多功能，比如在 Spring MVC 中，展现层 Bean 位于一个子容器中，而业务层和持久层的 Bean 位于父容器中。这样，展现层 Bean 就可以引用业务层和持久层的 Bean，而业务层和持久层的 Bean 则看不到展现层的 Bean。

BeanFactory：

是 Spring 里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；

ApplicationContext：

应用上下文，继承 BeanFactory 接口，它是 Spring 的一各更高级的容器，提供了更多的有用的功能；

1) 国际化（MessageSource）

2) 访问资源，如 URL 和文件（ResourceLoader）

3) 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的 web 层

4) 消息发送、响应机制（ApplicationEventPublisher）

5) AOP（拦截器）

两者装载 bean 的区别

BeanFactory：

BeanFactory 在启动的时候不会去实例化 Bean，中有从容器中拿 Bean 的时候才会去实例化；

ApplicationContext：

ApplicationContext 在启动的时候就把所有的 Bean 全部实例化了。它还可以为 Bean 配置 lazy-init=true 来让 Bean 延迟实例化；

我们该用 BeanFactory 还是 ApplicationContent

延迟实例化的优点：（BeanFactory）

应用启动的时候占用资源很少；对资源要求较高的应用，比较有优势；

不延迟实例化的优点： （ApplicationContext）

1. 所有的 Bean 在启动的时候都加载，系统运行的速度快；

2. 在启动的时候所有的 Bean 都加载了，我们就能在系统启动的时候，尽早的发现系统中的配置问题

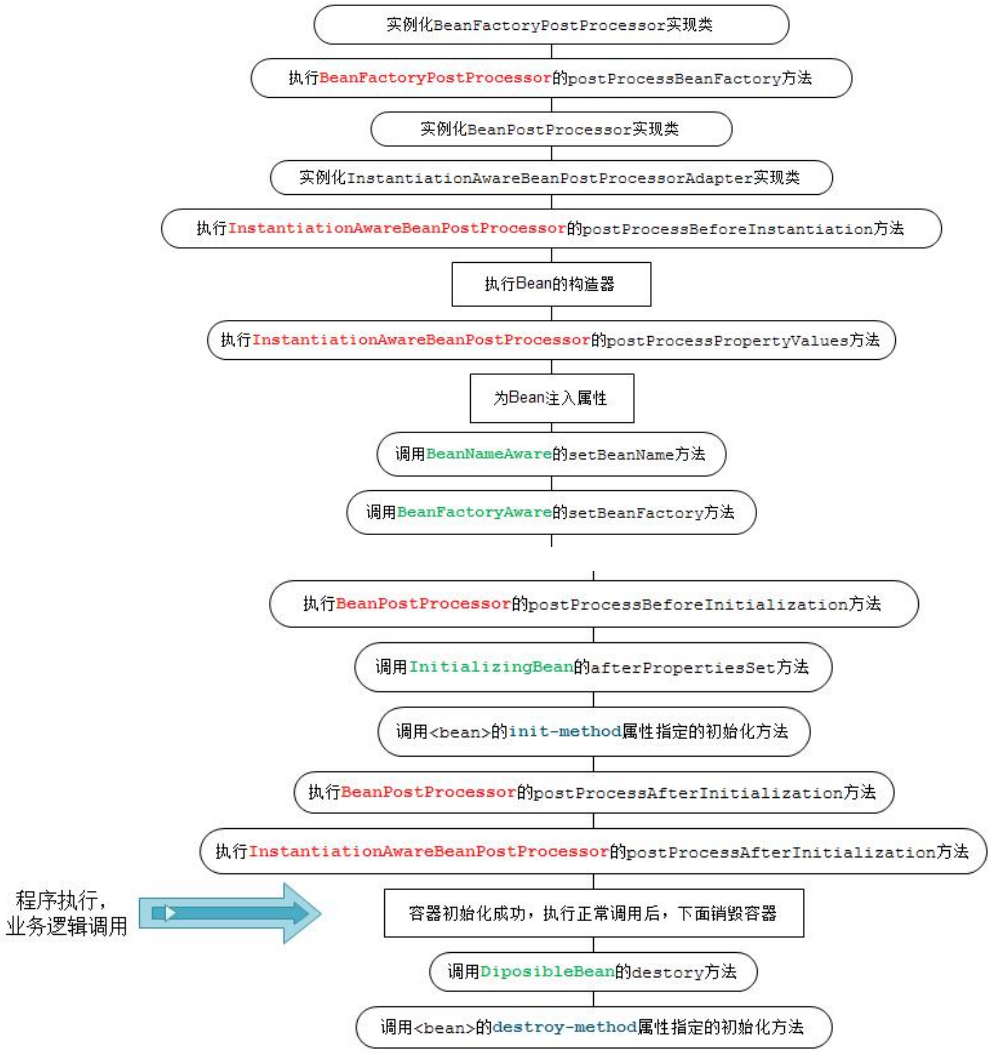
3. 建议 web 应用，在启动的时候就把所有的 Bean 都加载了。（把费时的操作放到系统启动中完成）

2. . Spring Bean 的生命周期

一、生命周期流程图：

Spring Bean 的完整生命周期从创建 Spring 容器开始，直到最终 Spring 容器销毁 Bean，

这其中包含了一系列关键点。



容器注册了以上各种接口，程序那么将会按照以上的流程进行。下面将仔细讲解各接口作用。

二、各种接口方法分类

Bean 的完整生命周期经历了各种方法调用，这些方法可以划分为以下几类：

1、Bean 自身的方法 ： 这个包括了 Bean 本身调用的方法和通过配置文件中<bean>的 init-method 和 destroy-method 指定的方法

2、Bean 级生命周期接口方法 ： 这个包括了 BeanNameAware、BeanFactoryAware、InitializingBean 和 DiposableBean 这些接口的方法

3、容器级生命周期接口方法 ： 这个包括了 InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。

4、工厂后处理器接口方法 ： 这个包括了 AspectJWeavingEnabler, ConfigurationClassPostProcessor, CustomAutowireConfigurer 等等非常有用的工厂后处理器接口的方法。工厂后处理器也是容器级的。在应用上下文装配配置文件之后立即调用。

1：Bean 的建立：

容器寻找 Bean 的定义信息并将其实例化。

2：属性注入：

使用依赖注入，Spring 按照 Bean 定义信息配置 Bean 所有属性

3：BeanNameAware 的 setBeanName()：

如果 Bean 类有实现 org.springframework.beans.BeanNameAware 接口，工厂调用 Bean 的 setBeanName()方法传递 Bean 的 ID。

4：BeanFactoryAware 的 setBeanFactory()：

如果 Bean 类有实现 org.springframework.beans.factory.BeanFactoryAware 接口，工厂调用 setBeanFactory()方法传入工厂自身。

5：BeanPostProcessors 的 ProcessBeforeInitialization()

如果有 org.springframework.beans.factory.config.BeanPostProcessors 和 Bean 关联，那么其 postProcessBeforeInitialization()方法将被将被调用。

6：initializingBean 的 afterPropertiesSet()：

如果 Bean 类已实现 org.springframework.beans.factory.InitializingBean 接口，则执行他的 afterProPertiesSet()方法

7：Bean 定义文件中定义 init-method：

可以在 Bean 定义文件中使用"init-method"属性设定方法名称例如：

如果有以上设置的话，则执行到这个阶段，就会执行 initBean()方法

8：BeanPostProcessors 的 ProcessaAfterInitialization()

如果有任何的 BeanPostProcessors 实例与 Bean 实例关联，则执行 BeanPostProcessors 实例的 ProcessaAfterInitialization()方法

此时，Bean 已经可以被应用系统使用，并且将保留在 BeanFactory 中知道它不在被使用。有两种方法可以将其从 BeanFactory 中删除掉(如图 1.2):

1：DisposableBean 的 destroy()

在容器关闭时，如果 Bean 类有实现 org.springframework.beans.factory.DisposableBean 接口，则执行他的 destroy()方法

2：Bean 定义文件中定义 destroy-method

在容器关闭时，可以在 Bean 定义文件中使用"destroy-method"属性设定方法名称，例如：

如果有以上设定的话，则进行至这个阶段时，就会执行 destroy()方法，如果是使用 ApplicationContext 来生成并管理 Bean 的话则稍有不同，使用 ApplicationContext 来生成及管理 Bean 实例的话，在执行 BeanFactoryAware 的 setBeanFactory()阶段后，若 Bean 类上有实现 org.springframework.context.ApplicationContextAware 接口，则执行其 setApplicationContext()方法，接着才执行 BeanPostProcessors 的 ProcessBeforeInitialization()及之后的流程。

在说明前可以思考一下 Servlet 的生命周期：实例化，初始 init，接收请求 service，销毁 destroy；

Spring 上下文中的 Bean 也类似，如下

1、实例化一个 Bean——也就是我们常说的 new；

2、按照 Spring 上下文对实例化的 Bean 进行配置——也就是 IOC 注入；

3、如果这个 Bean 已经实现了 BeanNameAware 接口，会调用它实现的 setBeanName(String)方法，此处传递的就是 Spring 配置文件中 Bean 的 id 值

4、如果这个 Bean 已经实现了 BeanFactoryAware 接口，会调用它实现的 setBeanFactory(setBeanFactory(BeanFactory)传递的是 Spring 工厂自身（可以用这个方式来获取其它 Bean，只需在 Spring 配置文件中配置一个普通的 Bean 就可以）；

5、如果这个 Bean 已经实现了 ApplicationContextAware 接口，会调用 setApplicationContext(ApplicationContext)方法，传入 Spring 上下文（同样这个方式也可以实现步骤 4 的内容，但比 4 更好，因为 ApplicationContext 是 BeanFactory 的子接口，有更多的实现方法）；

6、如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor 经常被用作是 Bean 内容的更改，并且由于这个是在 Bean 初始化结束时调用那个的方法，也可以被应用于内存或缓存技术；

7、如果 Bean 在 Spring 配置文件中配置了 init-method 属性会自动调用其配置的初始化方法。

8、如果这个 Bean 关联了 BeanPostProcessor 接口，将会调用 postProcessAfterInitialization(Object obj, String s)方法、；

注：以上工作完成以后就可以应用这个 Bean 了，那这个 Bean 是一个 Singleton 的，所以一般情况下我们调用同一个 id 的 Bean 会是在内容地址相同的实例，当然在 Spring 配置文件中也可以配置非 Singleton，这里我们不做赘述。

9、当 Bean 不再需要时，会经过清理阶段，如果 Bean 实现了 DisposableBean 这个接口，会调用那个其实现的 destroy()方法；

10、最后，如果这个 Bean 的 Spring 配置中配置了 destroy-method 属性，会自动调用其配置的销毁方法。

以上 10 步骤可以作为面试或者笔试的模板，另外我们这里描述的是应用 Spring 上下文 Bean 的生命周期，如果应用 Spring 的工厂也就是 BeanFactory 的话去掉第 5 步就 Ok 了。

这 Spring 框架中，一旦把一个 bean 纳入到 Spring IoC 容器之中，这个 bean 的生命周期就会交由容器进行管理，一般担当管理者角色的是 BeanFactory 或 ApplicationContext。认识一下 Bean 的生命周期活动，对更好的利用它有很大的帮助。

下面以 BeanFactory 为例，说明一个 Bean 的生命周期活动：

Bean 的建立

由 BeanFactory 读取 Bean 定义文件，并生成各个实例。

Setter 注入: 执行 Bean 的属性依赖注入。

BeanNameAware 的 setBeanName()

如果 Bean 类实现了 org.springframework.beans.factory.BeanNameAware 接口，则执行其 setBeanName()方法。

BeanFactoryAware 的 setBeanFactory()

如果 Bean 类实现了 org.springframework.beans.factory.BeanFactoryAware 接口，则执行其 setBeanFactory()方法。

BeanPostProcessors 的 processBeforeInitialization()

容器中如果有实现 org.springframework.beans.factory.BeanPostProcessors 接口的实例，则任何 Bean 在初始化之前都会执行这个实例的 processBeforeInitialization()方法。

InitializingBean 的 afterPropertiesSet()

如果 Bean 类实现了 org.springframework.beans.factory.InitializingBean 接口，则执行其 afterPropertiesSet()方法。

Bean 定义文件中定义 init-method

在 Bean 定义文件中使用“init-method”属性设定方法名称，如下：

```
<bean id="demoBean" class="com.yangsq.bean.DemoBean" init-method="initMethod">
   .......
 </bean>
```

这时会执行 initMethod()方法，注意，这个方法是不带参数的。

BeanPostProcessors 的 processAfterInitialization()

容器中如果有实现 org.springframework.beans.factory.BeanPostProcessors 接口的实例，则任何 Bean 在初始化之前都会执行这个实例的 processAfterInitialization()方法。

DisposableBean 的 destroy()

在容器关闭时，如果 Bean 类实现了 org.springframework.beans.factory.DisposableBean 接口，则执行它的 destroy()方法。

Bean 定义文件中定义 destroy-method

在容器关闭时，可以在 Bean 定义文件中使用“destory-method”定义的方法

```
<bean id="demoBean" class="com.yangsq.bean.DemoBean" destory-method="destroyMethod">
   .......
</bean>
```

这时会执行 destroyMethod()方法，注意，这个方法是不带参数的。

如果使用 ApplicationContext 来维护一个 Bean 的生命周期，则基本上与上边的流程相同，只不过在执行 BeanNameAware 的 setBeanName()后，若有 Bean 类实现了 org.springframework.context.ApplicationContextAware 接口，则执行其 setApplicationContext()方法，然后再进行 BeanPostProcessors 的 processBeforeInitialization()

实际上，ApplicationContext 除了向 BeanFactory 那样维护容器外，还提供了更加丰富的框架功能，如 Bean 的消息，事件处理机制等。

3.Spring IOC 如何实现

Spring 中的 org.springframework.beans 包和 org.springframework.context 包构成了 Spring 框架 IoC 容器的基础。

BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex 接口对 BeanFactory（是一个子接口）进行了扩展，在 BeanFactory 的基础上添加了其他功能，比如与 Spring 的 AOP 更容易集成，也提供了处理 message resource 的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对 Web 应用的 WebApplicationContext。

org.springframework.beans.factory.BeanFactory 是 Spring IoC 容器的具体实现，用来包装和管理前面提到的各种 bean。BeanFactory 接口是 Spring IoC 容器的核心接口。

1、 IOC 容器就是具有依赖注入功能的容器，IOC 容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。应用程序无需直接在代码中 new 相关的对象，应用程序由 IOC 容器进行组装。在 Spring 中 BeanFactory 是 IOC 容器的实际代表者。

Spring IOC 容器如何知道哪些是它管理的对象呢？这就需要配置文件，Spring IOC 容器通过读取配置文件中的配置元数据，通过元数据对应用中的各个对象进行实例化及装配。一般使用基于 xml 配置文件进行配置元数据，而且 Spring 与配置文件完全解耦的，可以使用其他任何可能的方式进行配置元数据，比如注解、基于 java 文件的、基于属性文件的配置都可以。

那 Spring IOC 容器管理的对象叫什么呢？

2、 Bean 的概念

由 IOC 容器管理的那些组成你应用程序的对象我们就叫它 Bean， Bean 就是由 Spring

容器初始化、装配及管理的对象，除此之外，bean 就与应用程序中的其他对象没有什么区别了。那 IOC 怎样确定如何实例化 Bean、管理 Bean 之间的依赖关系以及管理 Bean 呢？这就需要配置元数据，在 Spring 中由 BeanDefinition 代表，后边会详细介绍，配置元数据指定如何实例化 Bean、如何组装 Bean 等。概念知道的差不多了，让我们来做个简单的例子。

```
public interface HelloService {
    public void sayHello();
}

public class HelloServiceImpl implements HelloService{
    public void sayHello(){
        System.out.println("Hello World!");
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <!-- id 表示组件的名字，class 表示组件类 -->
    <bean id="helloService" class="com.ljq.test.HelloServiceImpl" />
</beans>

public class HelloServiceTest {
    @Test
    public void testHelloWorld() {
        // 1、读取配置文件实例化一个 IOC 容器
        ApplicationContext context = new
ClassPathXmlApplicationContext("helloworld.xml");
        // 2、从容器中获取 Bean，注意此处完全“面向接口编程，而不是面向实现”
        HelloService helloService = context.getBean("helloService", HelloService.class);
        // 3、执行业务逻辑
        helloService.sayHello();
    }
}
```

在 Spring IOC 容器的代表就是 org.springframework.beans 包中的 BeanFactory 接口，BeanFactory 接口提供了 IOC 容器最基本功能；而 org.springframework.context 包下的 ApplicationContext 接口扩展了 BeanFactory，还提供了与 Spring AOP 集成、国际化处理、事件传播及提供不同层次的 context 实现 (如针对 web 应用的 WebApplicationContext)。简单说，BeanFactory 提供了 IOC 容器最基本功能，而 ApplicationContext 则增加了更多支持企业级功

能支持。ApplicationContext 完全继承 BeanFactory，因而 BeanFactory 所具有的语义也适用于 ApplicationContext。

容器实现一览：

• XmlBeanFactory：BeanFactory 实现，提供基本的 IOC 容器功能，可以从 classpath 或文件系统等获取资源；

（1）File file = new File("fileSystemConfig.xml");

Resource resource = new FileSystemResource(file);

BeanFactory beanFactory = new XmlBeanFactory(resource);

（2）

Resource resource = new ClassPathResource("classpath.xml");

BeanFactory beanFactory = new XmlBeanFactory(resource);

ClassPathXmlApplicationContext：ApplicationContext 实现，从 classpath 获取配置文件；

BeanFactory beanFactory = new ClassPathXmlApplicationContext("classpath.xml");

FileSystemXmlApplicationContext：ApplicationContext 实现，从文件系统获取配置文件。

BeanFactory beanFactory = new FileSystemXmlApplicationContext("fileSystemConfig.xml");

4. 说说 Spring AOP

AOP 技术利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个可重用模块，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的逻辑或责任封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。使用“横切”技术，AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如权限认证、日志、事务处理。Aop 的作用在于分离系统中的各种关注点，将核心关注点和横切关注点分离开来。

通知(Adivce)

通知有 5 种类型：

Before 在方法被调用之前调用

After 在方法完成后调用通知，无论方法是否执行成功

After-returning 在方法成功执行之后调用通知

After-throwing 在方法抛出异常后调用通知

Around 通知了好、包含了被通知的方法，在被通知的方法调用之前后调用之后执行自定义的行为

我们可能会问，那通知对应系统中的代码是一个方法、对象、类、还是接口什么的呢？我想说一点，其实都不是，你可以理解通知就是对应我们日常生活中所说的通知，比如‘某某人，你 2019 年 9 月 1 号来学校报个到’，通知更多地体现一种告诉我们（告诉系统何）何时执行，规定一个时间，在系统运行中的某个时间点（比如抛异常啦！方法执行前啦!）， 并非对应代码中的方法！并非对应代码中的方法！并非对应代码中的方法！

切点（Pointcut）

哈哈，这个你可能就比较容易理解了，切点在 Spring AOP 中确实是对应系统中的方法。但是这个方法是定义在切面中的方法，一般和通知一起使用，一起组成了切面。

连接点（Join point）
比如：方法调用、方法执行、字段设置/获取、异常处理执行、类初始化、甚至是 for 循环中的某个点
理论上，程序执行过程中的任何时点都可以作为作为织入点，而所有这些执行时点都是 Joint point
但 Spring AOP 目前仅支持方法执行 (method execution) 也可以这样理解，连接点就是你准备在系统中执行切点和切入通知的地方（一般是一个方法，一个字段）
切面（Aspect）
切面是切点和通知的集合，一般单独作为一个类。通知和切点共同定义了关于切面的全部内容，它是什么时候，在何时和何处完成功能。
引入（Introduction）
引用允许我们向现有的类添加新的方法或者属性
织入（Weaving）
组装方面来创建一个被通知对象。这可以在 编译 时完成（例如使用 AspectJ 编译器），也可以在运行时完成。Spring 和其他纯 Java AOP 框架一样，在运行时完成织入。

5. Spring AOP 实现原理

Spring AOP 中的动态代理主要有两种方式，JDK 动态代理和 CGLIB 动态代理。JDK 动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK 动态代理的核心是 InvocationHandler 接口和 Proxy 类。

如果目标类没有实现接口，那么 Spring AOP 会选择使用 CGLIB 来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB 是通过继承的方式做的动态代理，因此如果某个类被标记为 final，那么它是无法使用 CGLIB 做动态代理的。

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DynaProxyHello implements InvocationHandler {
    private Object target;//目标对象
    /**
     * 通过反射来实例化目标对象
     * @param object
     * @return
     */
    public Object bind(Object object){
        this.target = object;
        return Proxy.newProxyInstance(this.target.getClass().getClassLoader(),
                this.target.getClass().getInterfaces(),this);
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Object result = null;
        Logger.start();
```

```
            result = method.invoke(this.target, args);
            Logger.end();
            return result;
        }
    }
```

6.动态代理（cglib 与 JDK）

JDK 动态代理类和委托类需要都实现同一个接口。也就是说只有实现了某个接口的类可以使用 Java 动态代理机制。但是，事实上使用中并不是遇到的所有类都会给你实现一个接口。因此，对于没有实现接口的类，就不能使用该机制。而 CGLIB 则可以实现对类的动态代理。

（1）静态代理：

```
    package net.battier.dao;
    public interface Count {
        public void queryCount();
        public void updateCount();
    }
    public class CountImpl implements Count {
        @Override
        public void queryCount() {
            System.out.println("查看账户...");
        }
        @Override
        public void updateCount() {
            System.out.println("修改账户...");
        }
    }
    public class CountProxy implements Count {
        private CountImpl countImpl;   //组合一个业务实现类对象来进行真正的业务方法的调用
        public CountProxy(CountImpl countImpl) {
            this.countImpl = countImpl;
        }
        @Override
        public void queryCount() {
            System.out.println("查询账户的预处理————————");
            // 调用真正的查询账户方法
            countImpl.queryCount();
            System.out.println("查询账户之后————————————");
        }
        @Override
        public void updateCount() {
            System.out.println("修改账户之前的预处理————————");
```

```
            // 调用真正的修改账户操作
            countImpl.updateCount();
            System.out.println("修改账户之后————————————————");
        }
    }
        public static void main(String[] args) {
            CountImpl countImpl = new CountImpl();
            CountProxy countProxy = new CountProxy(countImpl);
            countProxy.updateCount();
            countProxy.queryCount();

        }
```

（2）jdk 动态代理

```
    public interface BookFacade {
        public void addBook();
    }
    public class BookFacadeImpl implements BookFacade {
        @Override
        public void addBook() {
            System.out.println("增加图书方法。。。");
        }
    }
    public class BookFacadeProxy implements InvocationHandler {
        private Object target;//这其实业务实现类对象，用来调用具体的业务方法
        /**
         * 绑定业务对象并返回一个代理类
         */
        public Object bind(Object target) {
            this.target = target;   //接收业务实现类对象参数
           //通过反射机制，创建一个代理类对象实例并返回。用户进行方法调用时使用
           //创建代理对象时，需要传递该业务类的类加载器（用来获取业务实现类的元数据，在包装方法是调用真正的业务方法）、接口、handler 实现类
            return Proxy.newProxyInstance(target.getClass().getClassLoader(),
                    target.getClass().getInterfaces(), this); }
        /**
         * 包装调用方法：进行预处理、调用后处理
         */
        public Object invoke(Object proxy, Method method, Object[] args)
                throws Throwable {
            Object result=null;
            System.out.println("预处理操作————————");
            //调用真正的业务方法
            result=method.invoke(target, args);
            System.out.println("调用后处理————————");
```

```
            return result;
        }
    }
    public static void main(String[] args) {
            BookFacadeImpl bookFacadeImpl=new BookFacadeImpl();
            BookFacadeProxy proxy = new BookFacadeProxy();
            BookFacade bookfacade = (BookFacade) proxy.bind(bookFacadeImpl);
            bookfacade.addBook();
        }
```

（3）Cglib 动态代理

```
    public class BookFacadeImpl1 {
        public void addBook() {
            System.out.println("新增图书...");
        }
    }
    public class BookFacadeCglib implements MethodInterceptor {
        private Object target;//业务类对象，供代理方法中进行真正的业务方法调用
        //相当于 JDK 动态代理中的绑定
        public Object getInstance(Object target) {
            this.target = target;  //给业务对象赋值
            Enhancer enhancer = new Enhancer(); //创建加强器，用来创建动态代理类
            enhancer.setSuperclass(this.target.getClass());  //为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）
            //设置回调：对于代理类上所有方法的调用，都会调用 CallBack，而 Callback 则需要实现 intercept()方法进行拦
            enhancer.setCallback(this);
           // 创建动态代理类对象并返回
           return enhancer.create();
        }
        // 实现回调方法
        public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
            System.out.println("预处理——————————");
            proxy.invokeSuper(obj, args); //调用业务类（父类中）的方法
            System.out.println("调用后操作——————————");
            return null;
        }
    public static void main(String[] args) {
            BookFacadeImpl1 bookFacade=new BookFacadeImpl1()；
            BookFacadeCglib   cglib=new BookFacadeCglib();
            BookFacadeImpl1 bookCglib=(BookFacadeImpl1)cglib.getInstance(bookFacade);
            bookCglib.addBook();
        }
```

7.Spring 事务实现方式

一.事务的 4 个特性：

原子性：一个事务中所有对数据库的操作是一个不可分割的操作序列，要么全做，要么全部做。

一致性：数据不会因为事务的执行而遭到破坏。

隔离性：一个事务的执行，不受其他事务（进程）的干扰。既并发执行的个事务之间互不干扰。

持久性：一个事务一旦提交，它对数据库的改变将是永久的。

二.事务的实现方式：

实现方式共有两种：编码方式；声明式事务管理方式。

基于 AOP 技术实现的声明式事务管理，实质就是：在方法执行前后进行拦截，然后在目标方法开始之前创建并加入事务，执行完目标方法后根据执行情况提交或回滚事务。

声明式事务管理又有两种方式：基于 XML 配置文件的方式；另一个是在业务方法上进行 @Transactional 注解，将事务规则应用到业务逻辑中。

三.创建事务的时机：

是否需要创建事务，是由事务传播行为控制的。读数据不需要或只为其指定只读事务，而数据的插入，修改，删除就需要事务管理了。

Spring 配置文件中关于事务配置总是由三个组成部分，分别是 DataSource、TransactionManager 和代理机制这三部分，无论哪种配置方式，一般变化的只是代理机制这部分。

DataSource、TransactionManager 这两部分只是会根据数据访问方式有所变化，比如使用 Hibernate 进行数据访问时，DataSource 实际为 SessionFactory，TransactionManager 的实现为 HibernateTransactionManager。

```
<tx:annotation-driven transaction-manager="transactionManager"/>
    <bean id="sessionFactory"
            class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
        <property name="configurationClass"
value="org.hibernate.cfg.AnnotationConfiguration" />
    </bean>
    <!-- 定义事务管理器（声明式的事务） -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

此时在 Service 类上需加上@Transactional 注解

8.Spring 事务底层原理

a、划分处理单元——IOC

由于 spring 解决的问题是对单个数据库进行局部事务处理的，具体的实现首相用 spring 中的 IOC 划分了事务处理单元。并且将对事务的各种配置放到了 ioc 容器中（设置事务管理器，

设置事务的传播特性及隔离机制）。
b、AOP 拦截需要进行事务处理的类
Spring 事务处理模块是通过 AOP 功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用 TransactionProxyFactoryBean 接口来使用 AOP 功能，生成 proxy 代理对象，通过 TransactionInterceptor 完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。读取 ioc 容器事务配置属性，转化为 spring 事务处理需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为 TransactionAttribute 表示的数据对象。
c、对事物处理实现（事务的生成、提交、回滚、挂起）
spring 委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource 数据源支持、hibernate 数据源事务处理支持、JDO 数据源事务处理支持，JPA、JTA 数据源事务处理支持。这些支持都是通过设计 PlatformTransactionManager、AbstractPlatforTransaction 一系列事务处理的支持。 为常用数据源支持提供了一系列的 TransactionManager。
d、结合
PlatformTransactionManager 实现了 TransactionInterception 接口，让其与 TransactionProxyFactoryBean 结合起来，形成一个 Spring 声明式事务处理的设计体系。

9. 自定义注解实现功能

创建自定义注解和创建一个接口相似，但是注解的 interface 关键字需要以@符号开头。
注解方法不能带有参数；
注解方法返回值类型限定为：基本类型、String、Enums、Annotation 或者是这些类型的数组；
注解方法可以有默认值；
注解本身能够包含元注解，元注解被用来注解其它注解。

注解有什么用？ 注解的作用基本有三个：

生成文档。这是最常见的，也是 java 最早提供的注解。常用的有 @see @param @return 等

跟踪代码依赖性，实现替代配置文件功能。比较常见的是 spring 2.5 开始的基于注解配置。作用就是减少配置。现在的框架基本都使用了这种配置来减少配置文件的数量。也是

在编译时进行格式检查。如@override 放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}
```

用@Retention(RetentionPolicy.CLASS)修饰的注解，表示注解的信息被保留在 class 文件(字节码文件)中当程序编译时，但不会被虚拟机读取在运行的时候；

用@Retention(RetentionPolicy.SOURCE)修饰的注解,表示注解的信息会被编译器抛弃，不会留在 class 文件中，注解的信息只会留在源文件中；

用@Retention(RetentionPolicy.RUNTIME)修饰的注解，表示注解的信息被保留在 class 文件(字节码文件)中当程序编译时，会被虚拟机保留在运行时。

还需要注意的是 java 的元注解一共有四个：

@Document ：一个简单的 Annotations 标记注解，表示是否将注解信息添加在 java 文

档中。

@Target：表示该注解用于什么地方。默认值为任何元素，表示该注解用于什么地方。可用的 ElementType 参数包括：

- ElementType.CONSTRUCTOR: 用于描述构造器
- ElementType.FIELD: 成员变量、对象、属性（包括 enum 实例）
- ElementType.LOCAL_VARIABLE: 用于描述局部变量
- ElementType.METHOD: 用于描述方法
- ElementType.PACKAGE: 用于描述包
- ElementType.PARAMETER: 用于描述参数
- ElementType.TYPE: 用于描述类、接口(包括注解类型) 或 enum 声明

@Inherited：@Inherited 阐述了某个被标注的类型是被继承的。如果一个使用了 @Inherited 修饰的 annotation 类型被用于一个 class，则这个 annotation 将被用于该 class 的子类。

常见标准的 Annotation：

1.）Override

java.lang.Override 是一个标记类型注解，它被用作标注方法。它说明了被标注的方法重载了父类的方法，起到了断言的作用。如果我们使用了这种注解在一个没有覆盖父类方法的方法时，java 编译器将以一个编译错误来警示。

2.）Deprecated

Deprecated 也是一种标记类型注解。当一个类型或者类型成员使用@Deprecated 修饰的话，编译器将不鼓励使用这个被标注的程序元素。所以使用这种修饰具有一定的“延续性”：如果我们在代码中通过继承或者覆盖的方式使用了这个过时的类型或者成员，虽然继承或者覆盖后的类型或者成员并不是被声明为@Deprecated，但编译器仍然要报警。

3.）SuppressWarnings

SuppressWarning 不是一个标记类型注解。它有一个类型为 String[] 的成员，这个成员的值为被禁止的警告名。对于 javac 编译器来讲，被-Xlint 选项有效的警告名也同样对 @SuppressWarings 有效，同时编译器忽略掉无法识别的警告名。

```
@SuppressWarnings("unchecked")
```

自定义注解：

自定义注解类编写的一些规则:

1. Annotation 型定义为 @interface, 所有的 Annotation 会自动继承 java.lang.Annotation 这一接口,并且不能再去继承别的类或是接口.

2. 参数成员只能用 public 或默认(default) 这两个访问权修饰

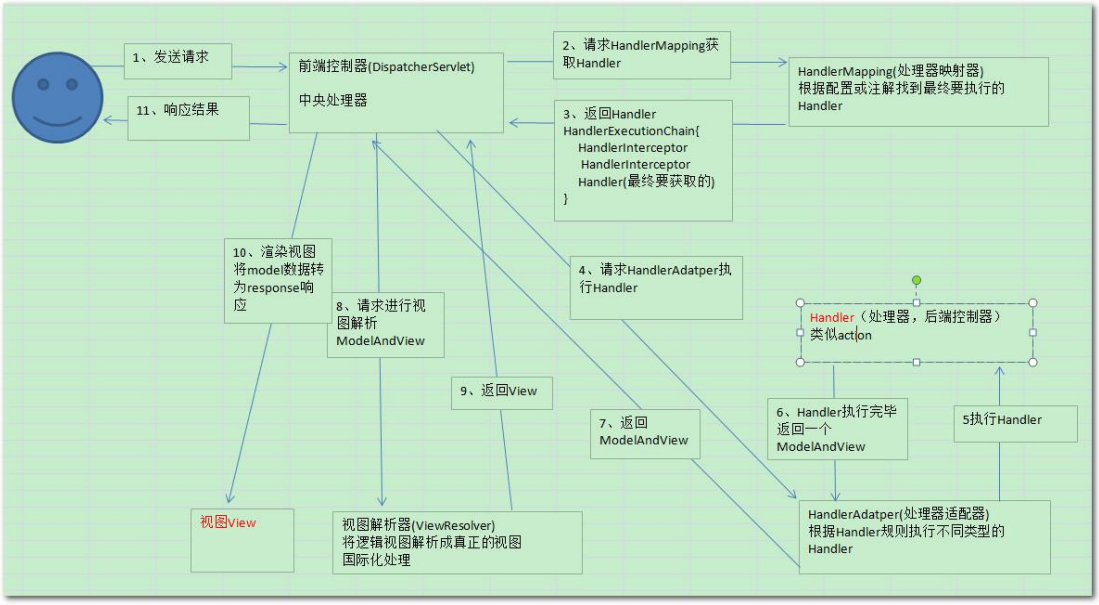
3. 参数成员只能用基本类型 byte、short、char、int、long、float、double、boolean 八种基本数据类型和 String、Enum、Class、annotations 等数据类型，以及这一些类型的数组.

4. 要获取类方法和字段的注解信息，必须通过 Java 的反射技术来获取 Annotation 对象，因为你除此之外没有别的获取注解对象的方法

5. 注解也可以没有定义成员,，不过这样注解就没啥用了

PS:自定义注解需要使用到元注解。

10. Spring MVC 运行流程



SpringMVC 流程:

1、 用户发送请求至前端控制器 DispatcherServlet。

2、 DispatcherServlet 收到请求调用 HandlerMapping 处理器映射器。

3、 处理器映射器找到具体的处理器(可以根据 xml 配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给 DispatcherServlet。

4、 DispatcherServlet 调用 HandlerAdapter 处理器适配器。

5、 HandlerAdapter 经过适配调用具体的处理器(Controller，也叫后端控制器)。

6、 Controller 执行完成返回 ModelAndView。

7、 HandlerAdapter 将 controller 执行结果 ModelAndView 返回给 DispatcherServlet。

8、 DispatcherServlet 将 ModelAndView 传给 ViewReslover 视图解析器。

9、 ViewReslover 解析后返回具体 View。

10、DispatcherServlet 根据 View 进行渲染视图（即将模型数据填充至视图中）。

11、 DispatcherServlet 响应用户。

组件说明：

以下组件通常使用框架提供实现：

DispatcherServlet：作为前端控制器，整个流程控制的中心，控制其它组件执行，统一调度，降低组件之间的耦合性，提高每个组件的扩展性。

HandlerMapping：通过扩展处理器映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

HandlAdapter：通过扩展处理器适配器，支持更多类型的处理器。

ViewResolver：通过扩展视图解析器，支持更多类型的视图解析，例如：jsp、freemarker、pdf、excel 等。

组件：

1、前端控制器 DispatcherServlet（不需要工程师开发）,由框架提供

作用：接收请求，响应结果，相当于转发器，中央处理器。有了 dispatcherServlet 减少了其它组件之间的耦合度。

用户请求到达前端控制器，它就相当于 mvc 模式中的 c，dispatcherServlet 是整个流程控制的中心，由它调用其它组件处理用户的请求，dispatcherServlet 的存在降低了组件之间的耦合性。

2、处理器映射器 HandlerMapping(不需要工程师开发),由框架提供

作用：根据请求的 url 查找 Handler

HandlerMapping 负责根据用户请求找到 Handler 即处理器，springmvc 提供了不同的映射器实现不同的映射方式，例如：配置文件方式，实现接口方式，注解方式等。

3、处理器适配器 HandlerAdapter

作用：按照特定规则（HandlerAdapter 要求的规则）去执行 Handler

通过 HandlerAdapter 对处理器进行执行，这是适配器模式的应用，通过扩展适配器可以对更多类型的处理器进行执行。

4、处理器 Handler(需要工程师开发)

注意：编写 Handler 时按照 HandlerAdapter 的要求去做，这样适配器才可以去正确执行 Handler

Handler 是继 DispatcherServlet 前端控制器的后端控制器，在 DispatcherServlet 的控制下 Handler 对具体的用户请求进行处理。

由于 Handler 涉及到具体的用户业务请求，所以一般情况需要工程师根据业务需求开发 Handler。

5、视图解析器 View resolver(不需要工程师开发),由框架提供

作用：进行视图解析，根据逻辑视图名解析成真正的视图（view）

View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。 springmvc 框架提供了很多的 View 视图类型，包括：jstlView、freemarkerView、pdfView 等。

一般情况下需要通过页面标签或页面模版技术将模型数据通过页面展示给用户，需要由工程师根据业务需求开发具体的页面。

6、视图 View(需要工程师开发 jsp...)

View 是一个接口，实现类支持不同的 View 类型（jsp、freemarker、pdf...）

11. Spring MVC 启动流程

接下来以一个常见的简单 web.xml 配置进行 Spring MVC 启动过程的分析，web.xml 配置内容如下:

```
<web-app>
  <display-name>Web Application</display-name>
  <!--全局变量配置-->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext-*.xml</param-value>
  </context-param>
  <!--监听器-->
  <listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
```

```
    <!--解决乱码问题的 filter-->
    <filter>
      <filter-name>CharacterEncodingFilter</filter-name>
      <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
      <init-param>
        <param-name>encoding</param-name>
        <param-value>utf-8</param-value>
      </init-param>
    </filter>
    <filter-mapping>
      <filter-name>CharacterEncodingFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!--Restful 前端控制器-->
    <servlet>
      <servlet-name>springMVC_rest</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
      <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-mvc.xml</param-value>
      </init-param>
    </servlet>
    <servlet-mapping>
      <servlet-name>springMVC_rest</servlet-name>
      <url-pattern>/</url-pattern>
    </servlet-mapping>
  </web-app>
```

tomcat web 容器启动时会去读取 web.xml 这样的部署描述文件，相关组件启动顺序为: 解析<context-param> => 解析<listener> => 解析<filter> => 解析<servlet>，具体初始化过程如下:

1、解析<context-param>里的键值对。

2、创建一个 application 内置对象即 ServletContext，servlet 上下文，用于全局共享。

3、将<context-param>的键值对放入 ServletContext 即 application 中，Web 应用内全局共享。

4、读取<listener>标签创建监听器，一般会使用 ContextLoaderListener 类，如果使用了 ContextLoaderListener 类，Spring 就会创建一个 WebApplicationContext 类的对象，WebApplicationContext 类就是 IoC 容器，ContextLoaderListener 类创建的 IoC 容器是根 IoC 容器为全局性的，并将其放置在 appication 中，作为应用内全局共享，键名为 WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，可以通过以下两种方法获取:

```
WebApplicationContext applicationContext = (WebApplicationContext)
application.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
```

WebApplicationContext applicationContext1 = WebApplicationContextUtils.getWebApplicationContext(application);

这个全局的根 IoC 容器只能获取到在该容器中创建的 Bean 不能访问到其他容器创建的 Bean，也就是读取 web.xml 配置的 contextConfigLocation 参数的 xml 文件来创建对应的 Bean。

5、listener 创建完成后如果有<filter>则会去创建 filter。

6、初始化创建<servlet>，一般使用 DispatchServlet 类。

7、DispatchServlet 的父类 FrameworkServlet 会重写其父类的 initServletBean 方法，并调用 initWebApplicationContext()以及 onRefresh()方法。

8、initWebApplicationContext()方法会创建一个当前 servlet 的一个 IoC 子容器，如果存在上述的全局 WebApplicationContext 则将其设置为父容器，如果不存在上述全局的则父容器为 null。

9、读取<servlet>标签的<init-param>配置的 xml 文件并加载相关 Bean。

10、onRefresh()方法创建 Web 应用相关组件。

12. Spring 的单例实现原理

单例模式有饿汉模式、懒汉模式、静态内部类、枚举等方式实现，但由于以上模式的构造方法是私有的，不可继承，Spring 为实现单例类可继承，使用的是单例注册表的方式（登记式单例）。

什么是单例注册表呢，

登记式单例实际上维护的是一组单例类的实例，将这些实例存储到一个 Map(登记簿)中，对于已经登记过的单例，则从工厂直接返回，对于没有登记的，则先登记，而后返回

1. 使用 map 实现注册表；

2. 使用 protect 修饰构造方法；

有的时候，我们不希望在一开始的时候就把一个类写成单例模式，但是在运用的时候，我们却可以像单例一样使用他

最典型的例子就是 spring，他的默认类型就是单例，spring 是如何做到把不是单例的类变成单例呢？

这就用到了登记式单例

其实登记式单例并没有去改变类，他所做的就是起到一个登记的作用，如果没有登记，他就给你登记，并把生成的实例保存起来，下次你要用的时候直接给你。

IOC 容器就是做的这个事，你需要就找他去拿，他就可以很方便的实现 Bean 的管理。

懒汉式饿汉式这种通过私有化构造函数，静态方法提供实例的单例类而言，是不支持继承的。这种模式的单例实现要求每个具体的单例类自身来维护单例实例和限制多个实例的生成。可以采用另外一种实现单例的思路：登记式单例，来使得单例对继承开放。

懒汉式饿汉式的 getInstance()方法都是无参的，返回本类的单例实例。而登记式单例是有参的，根据参数创建不同类的实例加入 Map 中，根据参数返回不同类的单例实例

我们看一个例子：

```
Import java.util.HashMap;
Public class RegSingleton{
    //使用一个 map 来当注册表
   Static private HashMap registry=new HashMap();
   //静态块，在类被加载时自动执行，把 RegistSingleton 自己也纳入容器管理
    Static{
     RegSingleton rs=new RegSingleton();
```

```
            Registry.put(rs.getClass().getName(),rs);
        }
    //受保护的默认构造函数，如果为继承关系，则可以调用，克服了单例类不能为继承的缺点
    Protected RegSingleton(){}
    //静态工厂方法，返回此类的唯一实例
    public static RegSingleton getInstance(String name){
        if(name==null){
            name=" RegSingleton";
        }if(registry.get(name)==null){
            try{
                registry.put(name,Class.forName(name).newInstance());
             }Catch(Exception ex){ex.printStackTrace();}
        }
        Return (RegSingleton)registry.get(name);
    }
    }
```

受保护的构造函数，不能是私有的，但是这样子类可以直接访问构造方法了

解决方式是把你的单例类放到一个外在的包中，以便在其它包中的类（包括缺省的包）无法实例化一个单例类。

```
    public abstract class AbstractBeanFactory implements ConfigurableBeanFactory{
        /**
         * 充当了 Bean 实例的缓存，实现方式和单例注册表相同
         */
        private final Map singletonCache=new HashMap();
        public Object getBean(String name)throws BeansException{
            return getBean(name,null,null);
        }
    ...
        public Object getBean(String name,Class requiredType,Object[] args)throws BeansException{
            //对传入的 Bean name 稍做处理，防止传入的 Bean name 名有非法字符(或则做转码)
            String beanName=transformedBeanName(name);
            Object bean=null;
            //手工检测单例注册表
            Object sharedInstance=null;
            //使用了代码锁定同步块，原理和同步方法相似，但是这种写法效率更高
            synchronized(this.singletonCache){
                sharedInstance=this.singletonCache.get(beanName);
             }
            if(sharedInstance!=null){
                ...
                //返回合适的缓存 Bean 实例
```

```
                bean=getObjectForSharedInstance(name,sharedInstance);
            }else{
                ...
                //取得 Bean 的定义
                RootBeanDefinition
mergedBeanDefinition=getMergedBeanDefinition(beanName,false);
                ...
                //根据 Bean 定义判断，此判断依据通常来自于组件配置文件的单例属性开关
                //<bean id="date" class="java.util.Date" scope="singleton"/>
                //如果是单例，做如下处理
                if(mergedBeanDefinition.isSingleton()){
                    synchronized(this.singletonCache){
                     //再次检测单例注册表
                      sharedInstance=this.singletonCache.get(beanName);
                      if(sharedInstance==null){
                          ...
                        try {
                            //真正创建 Bean 实例
                            sharedInstance=createBean(beanName,mergedBeanDefinition,args);
                            //向单例注册表注册 Bean 实例
                              addSingleton(beanName,sharedInstance);
                        }catch (Exception ex) {
                            ...
                        }finally{
                            ...
                       }
                      }
                    }
                    bean=getObjectForSharedInstance(name,sharedInstance);
                 }
                //如果是非单例，即 prototpye，每次都要新创建一个 Bean 实例
                //<bean id="date" class="java.util.Date" scope="prototype"/>
                else{
                    bean=createBean(beanName,mergedBeanDefinition,args);
                }
        }
        ...
            return bean;
        }
        }
```

13.Spring 框架中用到了哪些设计模式

1. 简单工厂

又叫做静态工厂方法（StaticFactory Method）模式，但不属于 23 种 GOF 设计模式之一。

简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

Spring 中的 BeanFactory 就是简单工厂模式的体现，根据传入一个唯一的标识来获得 Bean 对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

2. 工厂方法（Factory Method）

定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。

Spring 中的 FactoryBean 就是典型的工厂方法模式。

3. 单例（Singleton）

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

Spring 中的单例模式完成了后半句话，即提供了全局的访问点 BeanFactory。但没有从构造器级别去控制单例，这是因为 Spring 管理的是是任意的 Java 对象。

4. 适配器（Adapter）

将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

Spring 中在对于 AOP 的处理中有 Adapter 模式的例子。

由于 Advisor 链需要的是 MethodInterceptor（拦截器）对象，所以每一个 Advisor 中的 Advice 都要适配成对应的 MethodInterceptor 对象。

5.包装器（Decorator）

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

Spring 中用到的包装器模式在类名上有两种表现：一种是类名中含有 Wrapper，另一种是类名中含有 Decorator。基本上都是动态地给一个对象添加一些额外的职责。

6. 代理（Proxy）

为其他对象提供一种代理以控制对这个对象的访问。

从结构上来看和 Decorator 模式类似，但 Proxy 是控制，更像是一种对功能的限制，而 Decorator 是增加职责。

Spring 的 Proxy 模式在 aop 中有体现，比如 JdkDynamicAopProxy 和 Cglib2AopProxy。

7.观察者（Observer）

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

Spring 中 Observer 模式常用的地方是 listener 的实现。如 ApplicationListener。

8. 策略（Strategy）

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

Spring 中在实例化对象的时候用到 Strategy 模式

9.模板方法（Template Method）

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method 模式一般是需要继承的。这里想要探讨另一种对 Template Method 的理解。Spring 中的 JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到 JdbcTemplate 已有的稳定的、公用的数据库连接，那么我们怎么办呢？

我们可以把变化的东西抽出来作为一个参数传入 JdbcTemplate 的方法中。但是变化的东

西是一段代码，而且这段代码会用到 JdbcTemplate 中的变量。怎么办？

那我们就用回调对象吧。在这个回调对象中定义一个操纵 JdbcTemplate 中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到 JdbcTemplate，从而完成了调用。这可能是 Template Method 不需要继承的另一种实现方式吧。

14. Spring 其他产品（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）

目录

核心篇

数据存储

1、MySQL 索引使用的注意事项

EXPLAIN 可以帮助开发人员分析 SQL 问题,explain 显示了 mysql 如何使用索引来处理 select 语句以及连接表,可以帮助选择更好的索引和写出更优化的查询语句。

使用方法,在 select 语句前加上 Explain 就可以了：

索引虽然好处很多，但过多的使用索引可能带来相反的问题，索引也是有缺点的：

虽然索引大大提高了查询速度,同时却会降低更新表的速度,如对表进行 INSERT,UPDATE 和 DELETE。因为更新表时，mysql 不仅要保存数据，还要保存一下索引文件

建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在要给大表上建了多种组合索引，索引文件会膨胀很宽

索引只是提高效率的一个方式，如果 mysql 有大数据量的表，就要花时间研究建立最优的索引，或优化查询语句。

使用索引时，有一些技巧：

1.索引不会包含有 NULL 的列

只要列中包含有 NULL 值，都将不会被包含在索引中，复合索引中只要有一列含有 NULL 值，那么这一列对于此符合索引就是无效的。

2.使用短索引

对串列进行索引，如果可以就应该指定一个前缀长度。例如，如果有一个 char（255）的列，如果在前 10 个或 20 个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和 I/O 操作。

3.索引列排序

mysql 查询只使用一个索引，因此如果 where 子句中已经使用了索引的话，那么 order by 中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引。

4.like 语句操作

一般情况下不鼓励使用 like 操作，如果非使用不可，注意正确的使用方式。like ‘%aaa%’不会使用索引，而 like ‘aaa%’可以使用索引。

5.不要在列上进行运算

6.不使用 NOT IN 、<>、！=操作，但<,<=，=，>,>=,BETWEEN,IN 是可以用到索引的

7.索引要建立在经常进行 select 操作的字段上。

这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

8.索引要建立在值比较唯一的字段上。

9.对于那些定义为 text、image 和 bit 数据类型的列不应该增加索引。因为这些列的数据量要么相当大，要么取值很少。

10.在 where 和 join 中出现的列需要建立索引。

11.where 的查询条件里有不等号(where column != …),mysql 将无法使用索引。

12.如果 where 字句的查询条件里使用了函数(如：where DAY(column)=…),mysql 将无法使用索引。

13.在 join 操作中(需要从多个数据表提取数据时)，mysql 只有在主键和外键的数据类型相同时才能使用索引，否则及时建立了索引也不会使用。

2、说说反模式设计

简单的来说，反模式是指在对经常面对的问题经常使用的低效，不良，或者有待优化的设计模式/方法。甚至，反模式也可以是一种错误的开发思想/理念。在这里我举一个最简单的例子：在面向对象设计/编程中，有一条很重要的原则， 单一责任原则(Single responsibility principle)。其中心思想就是对于一个模块，或者一个类来说，这个模块或者这个类应该只对系统/软件的一个功能负责，而且该责任应该被该类完全封装起来。当开发人员需要修改系统的某个功能，这个模块/类是最主要的修改地方。相对应的一个反模式就是上帝类(God Class)，通常来说，这个类里面控制了很多其他的类，同时也依赖其他很多类。整个类不光负责自己的主要单一功能，而且还负责了其他很多功能，包括一些辅助功能。很多维护老程序的开发人员们可能都遇过这种类，一个类里有几千行的代码，有很多功能，但是责任不明确单一。单元测试程序也变复杂无比。维护/修改这个类的时间要远远超出其他类的时间。很多时候，形成这种情况并不是开发人员故意的。很多情况下主要是由于随着系统的年限，需求的变化，项目的资源压力，项目组人员流动，系统结构的变化而导致某些原先小型的，符合单一原则类慢慢的变的臃肿起来。最后当这个类变成了维护的噩梦(特别是原先熟悉的开发人员离职后)，重构该类就变成了一个不容易的工程。

3、说说分库与分表设计

垂直分表在日常开发和设计中比较常见，通俗的说法叫做“大表拆小表”，拆分是基于关系型数据库中的“列”（字段）进行的。通常情况，某个表中的字段比较多，可以新建立一张“扩展表”，将不经常使用或者长度较大的字段拆分出去放到“扩展表”中。在字段很多的情况下，拆分开确实更便于开发和维护（笔者曾见过某个遗留系统中，一个大表中包含 100 多列的）。某种意义上也能避免“跨页”的问题（MySQL、MSSQL 底层都是通过“数据页”来存储的，“跨页”问题可能会造成额外的性能开销，拆分字段的操作建议在数据库设计阶段就做好。如果是在发展过程中拆分，则需要改写以前的查询语句，会额外带来一定的成本和风险，建议谨慎。

垂直分库在“微服务”盛行的今天已经非常普及了。基本的思路就是按照业务模块来划分出不同的数据库，而不是像早期一样将所有的数据表都放到同一个数据库中。系统层面的“服务化”拆分操作，能够解决业务系统层面的耦合和性能瓶颈，有利于系统的扩展维护。而数据库层面的拆分，道理也是相通的。与服务的“治理”和“降级”机制类似，我们也能对不同业务类型的数据进行“分级”管理、维护、监控、扩展等。

众所周知，数据库往往最容易成为应用系统的瓶颈，而数据库本身属于“有状态”的，相对于 Web 和应用服务器来讲，是比较难实现“横向扩展”的。数据库的连接资源比较宝贵且单机处理能力也有限，在高并发场景下，垂直分库一定程度上能够突破 IO、连接数及单机硬件资源的瓶颈，是大型分布式系统中优化数据库架构的重要手段。

然后，很多人并没有从根本上搞清楚为什么要拆分，也没有掌握拆分的原则和技巧，只是一味的模仿大厂的做法。导致拆分后遇到很多问题（例如：跨库 join，分布式事务等）。

水平分表也称为横向分表，比较容易理解，就是将表中不同的数据行按照一定规律分布到不同的数据库表中（这些表保存在同一个数据库中），这样来降低单表数据量，优化查询性能。最常见的方式就是通过主键或者时间等字段进行 Hash 和取模后拆分。水平分表，能够降低单表的数据量，一定程度上可以缓解查询性能瓶颈。但本质上这些表还保存在同一个库

中，所以库级别还是会有 IO 瓶颈。所以，一般不建议采用这种做法。

水平分库分表与上面讲到的水平分表的思想相同，唯一不同的就是将这些拆分出来的表保存在不同的数据中。这也是很多大型互联网公司所选择的做法。某种意义上来讲，有些系统中使用的“冷热数据分离”（将一些使用较少的历史数据迁移到其他的数据库中。而在业务功能上，通常默认只提供热点数据的查询），也是类似的实践。在高并发和海量数据的场景下，分库分表能够有效缓解单机和单库的性能瓶颈和压力，突破 IO、连接数、硬件资源的瓶颈。当然，投入的硬件成本也会更高。同时，这也会带来一些复杂的技术问题和挑战（例如：跨分片的复杂查询，跨分片事务等）

4、分库与分表带来的分布式困境与应对之策

数据迁移与扩容问题

前面介绍到水平分表策略归纳总结为随机分表和连续分表两种情况。连续分表有可能存在数据热点的问题，有些表可能会被频繁地查询从而造成较大压力，热数据的表就成为了整个库的瓶颈，而有些表可能存的是历史数据，很少需要被查询到。连续分表的另外一个好处在于比较容易，不需要考虑迁移旧的数据，只需要添加分表就可以自动扩容。随机分表的数据相对比较均匀，不容易出现热点和并发访问的瓶颈。但是，分表扩展需要迁移旧的数据。

针对于水平分表的设计至关重要，需要评估中短期内业务的增长速度，对当前的数据量进行容量规划，综合成本因素，推算出大概需要多少分片。对于数据迁移的问题，一般做法是通过程序先读出数据，然后按照指定的分表策略再将数据写入到各个分表中。

表关联问题

在单库单表的情况下，联合查询是非常容易的。但是，随着分库与分表的演变，联合查询就遇到跨库关联和跨表关系问题。在设计之初就应该尽量避免联合查询，可以通过程序中进行拼装，或者通过反范式化设计进行规避。

分页与排序问题

一般情况下，列表分页时需要按照指定字段进行排序。在单库单表的情况下，分页和排序也是非常容易的。但是，随着分库与分表的演变，也会遇到跨库排序和跨表排序问题。为了最终结果的准确性，需要在不同的分表中将数据进行排序并返回，并将不同分表返回的结果集进行汇总和再次排序，最后再返回给用户。

分布式事务问题

随着分库与分表的演变，一定会遇到分布式事务问题，那么如何保证数据的一致性就成为一个必须面对的问题。目前，分布式事务并没有很好的解决方案，难以满足数据强一致性，一般情况下，使存储数据尽可能达到用户一致，保证系统经过一段较短的时间的自我恢复和修正，数据最终达到一致。

分布式全局唯一 ID

在单库单表的情况下，直接使用数据库自增特性来生成主键 ID，这样确实比较简单。在分库分表的环境中，数据分布在不同的分表上，不能再借助数据库自增长特性。需要使用全局唯一 ID，例如 UUID、GUID 等。关于如何选择合适的全局唯一 ID，我会在后面的章节中进行介绍。

5、说说 SQL 优化之道

一、一些常见的 SQL 实践

（1）负向条件查询不能使用索引

```
select from order where status!=0 and stauts!=1
```

not in/not exists 都不是好习惯

可以优化为 in 查询：

select from order where status in(2,3)

（2）前导模糊查询不能使用索引

select from order where desc like '%XX'

而非前导模糊查询则可以：

select from order where desc like 'XX%'

（3）数据区分度不大的字段不宜使用索引

select from user where sex=1

原因：性别只有男，女，每次过滤掉的数据很少，不宜使用索引。

经验上，能过滤 80%数据时就可以使用索引。对于订单状态，如果状态值很少，不宜使用索引，如果状态值很多，能够过滤大量数据，则应该建立索引。

（4）在属性上进行计算不能命中索引

select from order where YEAR(date) < = '2017'

即使 date 上建立了索引，也会全表扫描，可优化为值计算：

select from order where date < = CURDATE()

或者：

select from order where date < = '2017-01-01'

二、并非周知的 SQL 实践

（5）如果业务大部分是单条查询，使用 Hash 索引性能更好，例如用户中心

select from user where uid=?

select from user where login_name=?

原因：B-Tree 索引的时间复杂度是 O(log(n))；Hash 索引的时间复杂度是 O(1)

（6）允许为 null 的列，查询有潜在大坑

单列索引不存 null 值，复合索引不存全为 null 的值，如果列允许为 null，可能会得到“不符合预期”的结果集

select from user where name != 'shenjian'

如果 name 允许为 null，索引不存储 null 值，结果集中不会包含这些记录。

所以，请使用 not null 约束以及默认值。

（7）复合索引最左前缀，并不是值 SQL 语句的 where 顺序要和复合索引一致

用户中心建立了(login_name, passwd)的复合索引

select from user where login_name=? and passwd=?

select from user where passwd=? and login_name=?

都能够命中索引

select from user where login_name=?

也能命中索引，满足复合索引最左前缀

select from user where passwd=?

不能命中索引，不满足复合索引最左前缀

（8）使用 ENUM 而不是字符串

ENUM 保存的是 TINYINT，别在枚举中搞一些“中国”“北京”“技术部”这样的字符串，字符串空间又大，效率又低。

三、小众但有用的 SQL 实践

（9）如果明确知道只有一条结果返回，limit 1 能够提高效率

select from user where login_name=?

可以优化为：

select from user where login_name=? limit 1

原因：你知道只有一条结果，但数据库并不知道，明确告诉它，让它主动停止游标移动

（10）把计算放到业务层而不是数据库层，除了节省数据的 CPU，还有意想不到的查询缓存优化效果

select from order where date < = CURDATE()

这不是一个好的 SQL 实践，应该优化为：

$curDate = date('Y-m-d');

$res = mysqlquery(

'select from order where date < = $curDate');

原因：

释放了数据库的 CPU

多次调用，传入的 SQL 相同，才可以利用查询缓存

（11）强制类型转换会全表扫描

select from user where phone=13800001234

你以为会命中 phone 索引么？大错特错了，这个语句究竟要怎么改？

末了，再加一条，不要使用 select *（潜台词，文章的 SQL 都不合格 ==），只返回需要的列，能够大大的节省数据传输量，与数据库的内存使用量哟。

6、MySQL 遇到的死锁问题

产生死锁的四个必要条件：

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） 不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

下列方法有助于最大限度地降低死锁：

（1）按同一顺序访问对象。

（2）避免事务中的用户交互。

（3）保持事务简短并在一个批处理中。

（4）使用低隔离级别。

（5）使用绑定连接。

7、存储引擎的 InnoDB 与 MyISAM

◆.InnoDB 不支持 FULLTEXT 类型的索引。

◆2.InnoDB 中不保存表的具体行数，也就是说，执行 select count() from table 时，InnoDB 要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count()语句包含 where 条件时，两种表的操作是一样的。

◆3.对于 AUTO_INCREMENT 类型的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中，可以和其他字段一起建立联合索引。

◆4.DELETE FROM table 时，InnoDB 不会重新建立表，而是一行一行的删除。

◆5.LOAD TABLE FROM MASTER 操作对 InnoDB 是不起作用的，解决方法是首先把 InnoDB

表改成 MyISAM 表，导入数据后再改成 InnoDB 表，但是对于使用的额外的 InnoDB 特性(例如外键)的表不适用。

另外，InnoDB 表的行锁也不是绝对的，假如在执行一个 SQL 语句时 MySQL 不能确定要扫描的范围，InnoDB 表同样会锁全表，例如 update table set num=1 where name like “%aaa%”

8、数据库索引的原理

数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树。

9、为什么要用 B-tree

B-tree：B 树属于多叉树又名平衡多路查找树。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘 I/O 消耗，相对于内存存取，I/O 存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘 I/O 操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘 I/O 的存取次数。而 B-/+/*Tree，经过改进可以有效的利用系统对磁盘的块读取特性，在读取相同磁盘块的同时，尽可能多的加载索引数据，来提高索引命中效率，从而达到减少磁盘 IO 的读取次数。

（1）B-tree 利用了磁盘块的特性进行构建的树。每个磁盘块一个节点，每个节点包含了很关键字。把树的节点关键字增多后树的层级比原来的二叉树少了，减少数据查找的次数和复杂度。

B-tree 巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页（每页为 4K），这样每个节点只需要一次 I/O 就可以完全载入。

B-tree 的数据可以存在任何节点中。

（2）B+tree 是 B-tree 的变种，数据只能存储在叶子节点。

B+tree 是 B-tree 的变种，B+tree 数据只存储在叶子节点中。这样在 B 树的基础上每个节点存储的关键字数更多，树的层级更少所以查询数据更快，所有指关键字指针都存在叶子节点，所以每次查找的次数都相同所以查询速度更稳定。

（3）B*tree 每个磁盘块中又添加了对下一个磁盘块的引用。这样可以在当前磁盘块满时，不用扩容直接存储到下一个临近磁盘块中。当两个邻近的磁盘块都满时，这两个磁盘块各分出 1/3 的数据重新分配一个磁盘块，这样这三个磁盘块的数据都为 2/3。

在 B+树的基础上因其初始化的容量变大，使得节点空间使用率更高，而又存有兄弟节点的指针，可以向兄弟节点转移关键字的特性使得 B*树额分解次数变得更少。

10、聚集索引与非聚集索引的区别

聚集索引，就是主键索引。

除了聚集索引以外的索引都是非聚集索引，只是人们想细分一下非聚集索引，分成普通索引，唯一索引，全文索引。

1).聚集索引一个表只能有一个，而非聚集索引一个表可以存在多个

2).聚集索引存储记录是物理上连续存在，而非聚集索引是逻辑上的连续，物理存储并不连续

3).聚集索引:物理存储按照索引排序；聚集索引是一种索引组织形式，索引的键值逻辑顺序决定了表数据行的物理存储顺序

非聚集索引:物理存储不按照索引排序；非聚集索引则就是普通索引了，仅仅只是对数据列创建相应的索引，不影响整个表的物理存储顺序.

4) .索引是通过二叉树的数据结构来描述的，我们可以这么理解聚簇索引：索引的叶节点就是数据节点。而非聚簇索引的叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。

11、limit 20000 加载很慢怎么解决

limit10000,20 的意思扫描满足条件的 10020 行，扔掉前面的 10000 行，返回最后的 20 行，问题就在这里。

mysql 的性能低是因为数据库要去扫描 N+M 条记录，然后又要放弃之前 N 条记录，开销很大。

解决思略：

1、前端加缓存，或者其他方式，减少落到库的查询操作，例如某些系统中数据在搜索引擎中有备份的，可以用 es 等进行搜索

2、使用延迟关联，即先通用 limit 得到需要数据的索引字段，然后再通过原表和索引字段关联获得需要数据。

select a.* from a,(select id from table_1 where is_deleted='N' limit 100000,20) b where a.id = b.id

3、从业务上实现，不分页如此多，例如只能分页前 100 页，后面的不允许再查了

4、不使用 limit N,M,而是使用 limit N，即将 offset 转化为 where 条件。

12、选择合适的分布式主键方案

1.数据库自增长序列或字段：

2.UUID：

对于 InnoDB 这种聚集主键类型的引擎来说，数据会按照主键进行物理排序，这对 auto_increment int 是个好消息，因为后一次插入的主键位置总是在最后。但是对 uuid 来说，这却是个坏消息，因为 uuid 是杂乱无章的，每次插入的主键位置是不确定的，可能在开头，也可能在中间，在进行主键物理排序的时候，势必会造成大量的 IO 操作影响效率，因此不适合使用 UUID 做物理主键。比较适合的做法是把 uuid 作为逻辑主键，物理主键依然使用 自增 ID

3.使用 UUID to Int64 的方法：

4.Redis 生成 ID：

假如一个集群中有 5 台 Redis，可以初始化每台 Redis 的值分别是 1,2,3,4,5，然后步长都是 5。各个 Redis 生成的 ID 为 ：

A : 1,6,11,16,21...

B : 2,7,12,17,22...

C : 3,8,13,18,23...

D : 4,9,14,19,24...

E : 5,10,15,20,25...

优点 : 1> 不依赖于数据库，灵活方便，且性能优于数据库

2> 数字 ID 天然排序，对分页或者需要排序的结果很有帮助

缺点 ：1> 如果系统中没有 Redis，还需要引入新的组件，增加系统复杂度

2> 需要编码和配置的工作量比较大

5. Twitter 的 snowflake 算法：

snowflake 算法 的原理就是用 64 位整数来表示主键。

1 bit 符号位 ： 表示正负，方便使用负数标识不正确的 ID

41 bit 毫秒时间 ：2^41 / (365 * 24 * 3600 * 1000) ≈ 69 年

10 bit 机房 ID + 机器 ID： 最大值为 1023

12 bit 递增序列 ： 最大值为 4095

6. 利用 zookeeper 生成唯一 ID：

zookeeper 主要通过其 znode 数据版本来生成序列号，可以生成 32 位和 64 位的数据版本号，客户端可以使用这个版本号来作为唯一的序列号。

很少会使用 zookeeper 来生成唯一 ID。主要是由于需要依赖 zookeeper，并且是多步调用 API，如果在竞争较大的情况下，需要考虑使用分布式锁。因此，性能在高并发的分布式环境下，也不甚理想

7. MongoDB 的 ObjectId:

MongoDB 的 ObjectId 和 snowflake 算法类似，它设计成轻量型的，不同的机器都能用全局唯一的同种方法方便地生成它。MongoDB 从一开始就设计用来作为分布式数据库，处理多个节点是一个核心要求，使其在分片环境中要容易生成得多。

13、选择合适的数据存储方案

1）关系型数据库 MySQL

MySQL 是一个最流行的关系型数据库，在互联网产品中应用比较广泛。一般情况下，MySQL 数据库是选择的第一方案，基本上有 80%～90% 的场景都是基于 MySQL 数据库的。因为，需要关系型数据库进行管理，此外，业务存在许多事务性的操作，需要保证事务的强一致性。同时，可能还存在一些复杂的 SQL 的查询。值得注意的是，前期尽量减少表的联合查询，便于后期数据量增大的情况下，做数据库的分库分表。

2）内存数据库 Redis

随着数据量的增长，MySQL 已经满足不了大型互联网类应用的需求。因此，Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓存服务降级、缓存穿透、缓存预热等缓存使用问题。

3）文档数据库 MongoDB

MongoDB 是对传统关系型数据库的补充，它非常适合高伸缩性的场景，它是可扩展性的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

此外，日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

MongoDB 还适合存储大尺寸的数据，GridFS 存储方案就是基于 MongoDB 的分布式文

件存储系统。

4）列族数据库 HBase

HBase 适合海量数据的存储与高性能实时查询，它是运行于 HDFS 文件系统之上，并且作为 MapReduce 分布式处理的目标数据库，以支撑离线分析型应用。在数据仓库、数据集市、商业智能等领域发挥了越来越多的作用，在数以千计的企业中支撑着大量的大数据分析场景的应用。

5）全文搜索引擎 ElasticSearch

在一般情况下，关系型数据库的模糊查询，都是通过 like 的方式进行查询。其中，like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。ElasticSearch 作为一个建立在全文搜索引擎 Apache Lucene 基础上的实时的分布式搜索和分析引擎，适用于处理实时搜索应用场景。此外，使用 ElasticSearch 全文搜索引擎，还可以支持多词条查询、匹配度与权重、自动联想、拼写纠错等高级功能。因此，可以使用 ElasticSearch 作为关系型数据库全文搜索的功能补充，将要进行全文搜索的数据缓存一份到 ElasticSearch 上，达到处理复杂的业务与提高查询速度的目的。

ElasticSearch 不仅仅适用于搜索场景，还非常适合日志处理与分析的场景。著名的 ELK 日志处理方案，由 ElasticSearch、Logstash 和 Kibana 三个组件组成，包括了日志收集、聚合、多维度查询、可视化显示等。

14、ObjectId 规则

[0,1,2,3] [4,5,6] [7,8] [9,10,11]

时间戳 |机器码 |PID |计数器

前四位是时间戳，可以提供秒级别的唯一性。

接下来三位是所在主机的唯一标识符，通常是机器主机名的散列值。

接下来两位是产生 ObjectId 的 PID，确保同一台机器上并发产生的 ObjectId 是唯一的。

前九位保证了同一秒钟不同机器的不同进程产生的 ObjectId 时唯一的。

最后三位是自增计数器，确保相同进程同一秒钟产生的 ObjectId 是唯一的。

15、聊聊 MongoDB 使用场景

高伸缩性的场景

MongoDB 非常适合高伸缩性的场景，它是可扩展性的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

日志系统的场景

日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

分布式文件存储

MongoDB 还适合存储大尺寸的数据，之前介绍的 GridFS 存储方案，就是基于 MongoDB 的分布式文件存储系统。

16、倒排索引

倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一

种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构。

有两种不同的反向索引形式：

一条记录的水平反向索引（或者反向档案索引）包含每个引用单词的文档的列表。

一个单词的水平反向索引（或者完全反向索引）又包含每个单词在一个文档中的位置。

17、聊聊 ElasticSearch 使用场景

ElasticSearch 的功能：

分布式的搜索引擎和数据分析引擎

搜索：网站的站内搜索，IT 系统的检索

数据分析：电商网站，统计销售排名前 10 的商家

全文检索，结构化检索，数据分析

全文检索：我想搜索商品名称包含某个关键字的商品

结构化检索：我想搜索商品分类为日化用品的商品都有哪些

数据分析：我们分析每一个商品分类下有多少个商品

对海量数据进行近实时的处理

分布式：ES 自动可以将海量数据分散到多台服务器上去存储和检索

海联数据的处理：分布式以后，就可以采用大量的服务器去存储和检索数据，自然而然就可以实现海量数据的处理了

近实时：检索数据要花费 1 小时（这就不要近实时，离线批处理，batch-processing）；在秒级别对数据进行搜索和分析

ElasticSearch 的应用场景：

维基百科

The Guardian（国外新闻网站）

Stack Overflow（国外的程序异常讨论论坛）

GitHub（开源代码管理）

电商网站

日志数据分析

商品价格监控网站

BI 系统

站内搜索

ElasticSearch 的特点：

可以作为一个大型分布式集群（数百台服务器）技术，处理 PB 级数据，服务大公司；也可以运行在单机上，服务小公司

Elasticsearch 不是什么新技术，主要是将全文检索、数据分析以及分布式技术，合并在了一起

对用户而言，是开箱即用的，非常简单，作为中小型的应用，直接 3 分钟部署一下 ES

Elasticsearch 作为传统数据库的一个补充,比如全文检索，同义词处理，相关度排名，复杂数据分析，海量数据的近实时处理。

缓存使用

18、Redis 有哪些类型

Redis 目前支持 5 种数据类型，分别是：
String（字符串）
List（列表）
Hash（字典）
Set（集合）
Sorted Set（有序集合）

1. String（字符串）

String 是简单的 key-value 键值对，value 不仅可以是 String，也可以是数字。String 在 redis 内部存储默认就是一个字符串，被 redisObject 所引用，当遇到 incr,decr 等操作时会转成数值型进行计算，此时 redisObject 的 encoding 字段为 int。

String 在 redis 内部存储默认就是一个字符串，被 redisObject 所引用，当遇到 incr,decr 等操作时会转成数值型进行计算，此时 redisObject 的 encoding 字段为 int。

应用场景

String 是最常用的一种数据类型，普通的 key/value 存储都可以归为此类，这里就不所做解释了。

2. List（列表）

Redis 列表是简单的字符串列表，可以类比到 C++中的 std::list，简单的说就是一个链表或者说是一个队列。可以从头部或尾部向 Redis 列表添加元素。列表的最大长度为 2^32 - 1，也即每个列表支持超过 40 亿个元素。

Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销，Redis 内部的很多实现，包括发送缓冲队列等也都是用的这个数据结构。

应用场景

Redis list 的应用场景非常多，也是 Redis 最重要的数据结构之一，比如 twitter 的关注列表、粉丝列表等都可以用 Redis 的 list 结构来实现，再比如有的应用使用 Redis 的 list 类型实现一个简单的轻量级消息队列，生产者 push，消费者 pop/bpop。

3. Hash（字典，哈希表）

类似 C#中的 dict 类型或者 C++中的 hash_map 类型。

Redis Hash 对应 Value 内部实际就是一个 HashMap，实际这里会有 2 种不同实现，这个 Hash 的成员比较少时 Redis 为了节省内存会采用类似一维数组的方式来紧凑存储，而不会采用真正的 HashMap 结构，对应的 value redisObject 的 encoding 为 zipmap,当成员数量增大时会自动转成真正的 HashMap,此时 encoding 为 ht。

应用场景

假设有多个用户及对应的用户信息，可以用来存储以用户 ID 为 key，将用户信息序列化为比如 json 格式做为 value 进行保存。

4. Set（集合）

可以理解为一堆值不重复的列表，类似数学领域中的集合概念，且 Redis 也提供了针对集合的求交集、并集、差集等操作。

set 的内部实现是一个 value 永远为 null 的 HashMap，实际就是通过计算 hash 的方式来快速排重的，这也是 set 能提供判断一个成员是否在集合内的原因。

应用场景

Redis set 对外提供的功能与 list 类似是一个列表的功能，特殊之处在于 set 是可以自动排重的，当你需要存储一个列表数据，又不希望出现重复数据时，set 是一个很好的选择，并且 set 提供了判断某个成员是否在一个 set 集合内的重要接口，这个也是 list 所不能提供

的。

又或者在微博应用中，每个用户关注的人存在一个集合中，就很容易实现求两个人的共同好友功能。

5. Sorted Set（有序集合）

Redis 有序集合类似 Redis 集合，不同的是增加了一个功能，即集合是有序的。一个有序集合的每个成员带有分数，用于进行排序。

Redis 有序集合添加、删除和测试的时间复杂度均为 O(1)(固定时间，无论里面包含的元素集合的数量)。列表的最大长度为 2^32- 1 元素(4294967295，超过 40 亿每个元素的集合)。

Redis sorted set 的内部使用 HashMap 和跳跃表(SkipList)来保证数据的存储和有序，HashMap 里放的是成员到 score 的映射，而跳跃表里存放的是所有的成员，排序依据是 HashMap 里存的 score,使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。

使用场景：

Redis sorted set 的使用场景与 set 类似，区别是 set 不是自动有序的，而 sorted set 可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。当你需要一个有序的并且不重复的集合列表，那么可以选择 sorted set 数据结构，比如 twitter 的 public timeline 可以以发表时间作为 score 来存储，这样获取时就是自动按时间排好序的。

又比如用户的积分排行榜需求就可以通过有序集合实现。还有上面介绍的使用 List 实现轻量级的消息队列，其实也可以通过 Sorted Set 实现有优先级或按权重的队列。

19、Redis 内部结构

Redis 内部使用一个 redisObject 对象来表示所有的 key 和 value。type ：代表一个 value 对象具体是何种数据类型。

encoding ：是不同数据类型在 redis 内部的存储方式，比如：type=string 代表 value 存储的是一个普通字符串，那么对应的 encoding 可以是 raw 或者是 int，如果是 int 则代表实际 redis 内部是按数值型类存储和表示这个字符串的，当然前提是这个字符串本身可以用数值表示，比如："123" "456"这样的字符串。

vm 字段：只有打开了 Redis 的虚拟内存功能，此字段才会真正的分配内存，该功能默认是关闭状态的。 Redis 使用 redisObject 来表示所有的 key/value 数据是比较浪费内存的，当然这些内存管理成本的付出主要也是为了给 Redis 不同数据类型提供一个统一的管理接口，实际作者也提供了多种方法帮助我们尽量节省内存使用。

20、聊聊 Redis 使用场景

使用场景说明：

1 计数器

数据统计的需求非常普遍，通过原子递增保持计数。例如，点赞数、收藏数、分享数等。

2 排行榜

排行榜按照得分进行排序，例如，展示最近、最热、点击率最高、活跃度最高等等条件的 top list。

3 用于存储时间戳

类似排行榜，使用 redis 的 zset 用于存储时间戳，时间会不断变化。例如，按照用户关注用户的最新动态列表。

4 记录用户判定信息

记录用户判定信息的需求也非常普遍，可以知道一个用户是否进行了某个操作。例如，用户是否点赞、用户是否收藏、用户是否分享等。

5 社交列表

社交属性相关的列表信息，例如，用户点赞列表、用户收藏列表、用户关注列表等。

6 缓存

缓存一些热点数据，例如，PC 版本文件更新内容、资讯标签和分类信息、生日祝福寿星列表。

7 队列

Redis 能作为一个很好的消息队列来使用，通过 list 的 lpop 及 lpush 接口进行队列的写入和消费，本身性能较好能解决大部分问题。但是，不提倡使用，更加建议使用 rabbitmq 等服务，作为消息中间件。

8 会话缓存

使用 Redis 进行会话缓存。例如，将 web session 存放在 Redis 中。

9 业务使用方式

String(字符串): 应用数, 资讯数等, (避免了 select count(*) from ...)

Hash（哈希表）: 用户粉丝列表, 用户点赞列表, 用户收藏列表, 用户关注列表等。

List（列表）：消息队列, push/sub 提醒。

SortedSet（有序集合）：热门列表, 最新动态列表, TopN, 自动排序。

21、redis 持久化机制与实现

众所周知，redis 是内存数据库，它把数据存储在内存中，这样在加快读取速度的同时也对数据安全性产生了新的问题，即当 redis 所在服务器发生宕机后，redis 数据库里的所有数据将会全部丢失。

为了解决这个问题，redis 提供了持久化功能——RDB 和 AOF。通俗的讲就是将内存中的数据写入硬盘中。

一、持久化之全量写入：RDB（快照）

```
[redis@6381]$ more /usr/local/redis/conf/redis.conf
save 900 1
save 300 10
save 60 10000
dbfilename "dump.rdb"                #持久化文件名称
dir "/data/dbs/redis/6381"       #持久化数据文件存放的路径
```

上面是 redis 配置文件里默认的 RDB 持久化设置，前三行都是对触发 RDB 的一个条件，例如第一行的意思是每 900 秒钟里 redis 数据库有一条数据被修改则触发 RDB，依次类推；只要有一条满足就会调用 BGSAVE 进行 RDB 持久化。第四行 dbfilename 指定了把内存里的数据库写入本地文件的名称，该文件是进行压缩后的二进制文件，通过该文件可以把数据库还原到生成该文件时数据库的状态。第五行 dir 指定了 RDB 文件存放的目录。

配置文件修改需要重启 redis 服务，我们还可以在命令行里进行配置，即时生效，服务器重启后需重新配置。

而 RDB 持久化也分两种：SAVE 和 BGSAVE

SAVE 是阻塞式的 RDB 持久化，当执行这个命令时 redis 的主进程把内存里的数据库状态写入到 RDB 文件（即上面的 dump.rdb）中，直到该文件创建完毕的这段时间内 redis 将不能处理任何命令请求。

BGSAVE 属于非阻塞式的持久化，它会创建一个子进程专门去把内存中的数据库状态写入 RDB 文件里，同时主进程还可以处理来自客户端的命令请求。但子进程基本是复制的父进程，这等于两个相同大小的 redis 进程在系统上运行，会造成内存使用率的大幅增加。

而 RDB 持久化也分两种：SAVE 和 BGSAVE

SAVE 是阻塞式的 RDB 持久化，当执行这个命令时 redis 的主进程把内存里的数据库状态写入到 RDB 文件（即上面的 dump.rdb）中，直到该文件创建完毕的这段时间内 redis 将不能处理任何命令请求。

BGSAVE 属于非阻塞式的持久化，它会创建一个子进程专门去把内存中的数据库状态写入 RDB 文件里，同时主进程还可以处理来自客户端的命令请求。但子进程基本是复制的父进程，这等于两个相同大小的 redis 进程在系统上运行，会造成内存使用率的大幅增加。

二、持久化之增量写入：AOF

与 RDB 的保存整个 redis 数据库状态不同，AOF 是通过保存对 redis 服务端的写命令（如 set、sadd、rpush）来记录数据库状态的，即保存你对 redis 数据库的写操作，以下就是 AOF 文件的内容。

```
[redis@iZ]$ more ~/redis/conf/redis.conf
dir "/data/dbs/redis/6381"              #AOF 文件存放目录
appendonly yes                          #开启 AOF 持久化，默认关闭
appendfilename "appendonly.aof"         #AOF 文件名称（默认）
appendfsync no                          #AOF 持久化策略
auto-aof-rewrite-percentage 100         #触发 AOF 文件重写的条件（默认）
auto-aof-rewrite-min-size 64mb          #触发 AOF 文件重写的条件（默认）
```

要弄明白上面几个配置就得从 AOF 的实现去理解，AOF 的持久化是通过命令追加、文件写入和文件同步三个步骤实现的。当 reids 开启 AOF 后，服务端每执行一次写操作（如 set、sadd、rpush）就会把该条命令追加到一个单独的 AOF 缓冲区的末尾，这就是命令追加；然后把 AOF 缓冲区的内容写入 AOF 文件里。看上去第二步就已经完成 AOF 持久化了那第三步是干什么的呢？这就需要从系统的文件写入机制说起：一般我们现在所使用的操作系统，为了提高文件的写入效率，都会有一个写入策略，即当你往硬盘写入数据时，操作系统不是实时的将数据写入硬盘，而是先把数据暂时的保存在一个内存缓冲区里，等到这个内存缓冲区的空间被填满或者是超过了设定的时限后才会真正的把缓冲区内的数据写入硬盘中。也就是说当 redis 进行到第二步文件写入的时候，从用户的角度看是已经把 AOF 缓冲区里的数据写入到 AOF 文件了，但对系统而言只不过是把 AOF 缓冲区的内容放到了另一个内存缓冲区里而已，之后 redis 还需要进行文件同步把该内存缓冲区里的数据真正写入硬盘上才算是完成了一次持久化。而何时进行文件同步则是根据配置的 appendfsync 来进行：

appendfsync 有三个选项：always、everysec 和 no：

1、选择 always 的时候服务器会在每执行一个事件就把 AOF 缓冲区的内容强制性的写入硬盘上的 AOF 文件里，可以看成你每执行一个 redis 写入命令就往 AOF 文件里记录这条命令，这保证了数据持久化的完整性，但效率是最慢的，却也是最安全的；

2、配置成 everysec 的话服务端每执行一次写操作（如 set、sadd、rpush）也会把该条命令追加到一个单独的 AOF 缓冲区的末尾，并将 AOF 缓冲区写入 AOF 文件，然后每隔一秒才会进行一次文件同步把内存缓冲区里的 AOF 缓存数据真正写入 AOF 文件里，这个模式兼顾了效率的同时也保证了数据的完整性，即使在服务器宕机也只会丢失一秒内对 redis 数据库做的修改；

3、将 appendfsync 配置成 no 则意味 redis 数据库里的数据就算丢失你也可以接受，它也会把每条写命令追加到 AOF 缓冲区的末尾，然后写入文件，但什么时候进行文件同步真正把数据写入 AOF 文件里则由系统自身决定，即当内存缓冲区的空间被填满或者是超过了设定的时限后系统自动同步。这种模式下效率是最快的，但对数据来说也是最不安全的，如果 redis 里的数据都是从后台数据库如 mysql 中取出来的，属于随时可以找回或者不重要的数据，那么可以考虑设置成这种模式。

相比 RDB 每次持久化都会内存翻倍，AOF 持久化除了在第一次启用时会新开一个子进程创建 AOF 文件会大幅度消耗内存外，之后的每次持久化对内存使用都很小。但 AOF 也有一个不可忽视的问题：AOF 文件过大。你对 redis 数据库的每一次写操作都会让 AOF 文件里增加一条数据，久而久之这个文件会形成一个庞然大物。还好的是 redis 提出了 AOF 重写的机制，即我们上面配置的 auto-aof-rewrite-percentage 和 auto-aof-rewrite-min-size。AOF 重写机制这里暂不细述，之后本人会另开博文对此解释，有兴趣的同学可以看看。我们只要知道 AOF 重写既是重新创建一个精简化的 AOF 文件，里面去掉了多余的冗余命令，并对原 AOF 文件进行覆盖。这保证了 AOF 文件大小处于让人可以接受的地步。而上面的 auto-aof-rewrite-percentage 和 auto-aof-rewrite-min-size 配置触发 AOF 重写的条件。

Redis 会记录上次重写后 AOF 文件的文件大小，而当前 AOF 文件大小跟上次重写后 AOF 文件大小的百分比超过 auto-aof-rewrite-percentage 设置的值，同时当前 AOF 文件大小也超过 auto-aof-rewrite-min-size 设置的最小值，则会触发 AOF 文件重写。以上面的配置为例，当现在的 AOF 文件大于 64mb 同时也大于上次重写 AOF 后的文件大小，则该文件就会被 AOF 重写。

最后需要注意的是，如果 redis 开启了 AOF 持久化功能，那么当 redis 服务重启时会优先使用 AOF 文件来还原数据库。

22、Redis 集群方案与实现

实现基础——分区:

分区是分割数据到多个 Redis 实例的处理过程，因此每个实例只保存 key 的一个子集。

通过利用多台计算机内存的和值，允许我们构造更大的数据库。

通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

集群的几种实现方式:

1 客户端分片

2 基于代理的分片

3 路由查询

客户端分片:

由客户端决定 key 写入或者读取的节点。

包括 jedis 在内的一些客户端，实现了客户端分片机制。

特性:

优点:

简单，性能高。

缺点:

业务逻辑与数据存储逻辑耦合

可运维性差

多业务各自使用 redis，集群资源难以管理

不支持动态增删节点

基于代理的分片:

客户端发送请求到一个代理，代理解析客户端的数据，将请求转发至正确的节点，然后将结果回复给客户端。

开源方案

Twemproxy

Codis

特性:

透明接入 Proxy 的逻辑和存储的逻辑是隔离的。

业务程序不用关心后端 Redis 实例，切换成本低。

代理层多了一次转发，性能有所损耗

路由查询

将请求发送到任意节点，接收到请求的节点会将查询请求发送到正确的节点上执行。

开源方案:Redis-cluster

集群的挑战

涉及多个 key 的操作通常是不被支持的。涉及多个 key 的 redis 事务不能使用。

举例来说，当两个 set 映射到不同的 redis 实例上时，你就不能对这两个 set 执行交集操作。

不能保证集群内的数据均衡。

分区的粒度是 key，如果某个 key 的值是巨大的 set、list，无法进行拆分。

增加或删除容量也比较复杂。

redis 集群需要支持在运行时增加、删除节点的透明数据平衡的能力。

Redis 集群各种方案原理

1)Twemproxy

Proxy-based

twtter 开源，C 语言编写，单线程。

支持 Redis 或 Memcached 作为后端存储。

Twemproxy 高可用部署架构

Twemproxy 特性:

支持失败节点自动删除

与 redis 的长连接，连接复用，连接数可配置

自动分片到后端多个 redis 实例上支持 redis pipelining 操作

多种 hash 算法：能够使用不同的分片策略和散列函数

支持一致性 hash，但是使用 DHT 之后，从集群中摘除节点时，不会进行 rehash 操作

可以设置后端实例的权重

支持状态监控

支持 select 切换数据库

Twemproxy 不足:

性能低：代理层损耗 && 本身效率低下

Redis 功能支持不完善

不支持针对多个值的操作，比如取 sets 的子交并补等（MGET 和 DEL 除外）

不支持 Redis 的事务操作

出错提示还不够完善

集群功能不够完善

仅作为代理层使用

本身不提供动态扩容，透明数据迁移等功能

失去维护

最近一次提交在一年之前。Twitter 内部已经不再使用。

2)Redis Cluster:

Redis 官网推出，可线性扩展到 1000 个节点

无中心架构

一致性哈希思想

客户端直连 redis 服务，免去了 proxy 代理的损耗

23、redis 为什么是单线程的

1、完全基于内存，绝大部分请求是纯粹的内存操作，非常快速。数据存在内存中，类似于 HashMap，HashMap 的优势就是查找和操作的时间复杂度都是 O(1)；

2、数据结构简单，对数据操作也简单，Redis 中的数据结构是专门进行设计的；

3、采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；

4、使用多路 I/O 复用模型，非阻塞 IO；

5、使用底层模型不同，它们之间底层实现方式以及与客户端之间通信的应用协议不一样，Redis 直接自己构建了 VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。

官方 FAQ 表示，因为 Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且 CPU 不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）。

24、缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

一、缓存雪崩

缓存雪崩我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间 key 是锁着的，这是过来 1000 个请求 999 个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法！

给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。

1、缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际 key 的缓存；

2、缓存数据：它的过期时间比缓存标记的时间延长 1 倍，例：标记缓存时间 30 分钟，数据缓存设置为 60 分钟。 这样，当缓存标记 key 过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

关于缓存崩溃的解决方法，这里提出了三种方案：使用锁或队列、设置过期标志更新缓存、为 key 设置不同的缓存失效时间，还有一各被称为“二级缓存”的解决方法，有兴趣的读者可以自行研究。

二、缓存穿透

缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

缓存穿透解决方案：

（1）采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。

（2）如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓存中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

三、缓存预热

缓存预热就是系统上线后，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

缓存预热解决方案：

（1）直接写个缓存刷新页面，上线时手工操作下；

（2）数据量不大，可以在项目启动的时候自动进行加载；

（3）定时刷新缓存。

四、缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis 默认的有 6 中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的 key 是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

五、缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在 95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于 90%，或者数据库连接池被打爆了，或者访问量突然猛增

到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

25、使用缓存的合理性问题

1 热点数据

对于冷数据而言，读取频率低，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。

对于热点数据，读取频率高。如果不做缓存，给数据库造成很大的压力，可能被击穿。

2 修改频率

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。（读取频率>修改频率）

如果这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到 Redis 缓存，减少数据库压力

3 缓存更新机制

一般情况下，我们采取缓存双淘汰机制，在更新数据库的时候淘汰缓存。此外，设定超时时间，例如 30 分钟。极限场景下，即使有脏数据入 cache，这个脏数据也最多存在三十分钟。

在高并发的情况下，设计上最好避免查询 Mysql，所以在更新数据库的时候更新缓存。

4 缓存可用性

缓存是提高数据读取性能的，缓存数据丢失和缓存不可用不会影响应用程序的处理。因此，一般的操作手段是，如果 Redis 出现异常，我们手动捕获这个异常，记录日志，并且去数据库查询数据返回给用户。

5 服务降级

服务降级的目的，是为了防止 Redis 服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis 出现问题，不去数据库查询，而是直接返回默认值给用户。

6 对于可用性、服务降级实际情况

在大公司，redis 都是 codis 集群，一般整个 codis 是不会挂掉的。所以在程序代码上没去实现可用性、服务降级。（不知我说的对不对，大家参考就好）

7 缓存预热

在新启动的缓存系统中，如果没有任何数据，在重建缓存数据过程中，系统的性能和数据库复制都不太好，那么最好的缓存系统启动时就把热点数据加载好，例如对于缓存信息，在启动缓存加载数据库中全部数据进行预热。一般情况下，我们会开通一个同步数据的接口，进行缓存预热。

26、redis 缓存过期机智

mySQL 里有 2000w 数据，redis 中只存 20w 的数据，如何保证 redis 中的数据都是热点数据

相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略（回收策略）。redis 提供 6 种数据淘汰策略：

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的

数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据。

27、Spring 中 RedisTemplate 操作

1. RedisTemplate 储存 set

```
/**
     * redis 储存 set
     */
    @RequestMapping("/set")
    public void redisSet(){
        Set<String> set=new HashSet<String>();
        set.add("111");
        set.add("222");
        set.add("333");
        redisTemplate.opsForSet().add("set",set);
        Set<String> resultSet =redisTemplate.opsForSet().members("set");
        System.out.println("resultSet:"+resultSet);
    }
```

2. RedisTemplated 储存 map

```
/**
     * redis 储存 map
     */
    @RequestMapping("/map")
    public void redisMap(){
        Map<String,String> map=new HashMap<String,String>();
        map.put("111","111");
        map.put("222","222");
        map.put("333","333");
        map.put("444","444");
        map.put("555","555");
        redisTemplate.opsForHash().putAll("map",map);
        Map<String,String> resultMap= redisTemplate.opsForHash().entries("map");
        List<String> reslutMapList=redisTemplate.opsForHash().values("map");
        Set<String>resultMapSet=redisTemplate.opsForHash().keys("map");
        String value=(String)redisTemplate.opsForHash().get("map","111");
        System.out.println("value:"+value);
        System.out.println("resultMapSet:"+resultMapSet);
        System.out.println("resultMap:"+resultMap);
```

```
            System.out.println("resulreslutMapListtMap:"+reslutMapList);
        }
```

3. RedisTemplate 储存 list

```
/**
     * redis 储存 list
     */
    @RequestMapping("/list")
    public void redisList(){
        List<String> list1=new ArrayList<String>();
        list1.add("111");
        list1.add("222");
        list1.add("333");
        List<String> list2=new ArrayList<String>();
        list2.add("444");
        list2.add("555");
        list2.add("666");
        redisTemplate.opsForList().leftPush("list1",list1);
        redisTemplate.opsForList().rightPush("list2",list2);
        List<String> resultList1=(List<String>)redisTemplate.opsForList().leftPop("list1");
        List<String> resultList2=(List<String>)redisTemplate.opsForList().rightPop("list2");
        System.out.println("resultList1:"+resultList1);
        System.out.println("resultList2:"+resultList2);
    }
```

4. RedisTemplate 储存 key-value

```
/**
     * redis 储存 key-value
     */
    @RequestMapping("/key/value")
    public void redisKeyValue(){
        System.out.println("缓存正在设置。。。。。。。。。");
        redisTemplate.opsForValue().set("111","111");
        redisTemplate.opsForValue().set("222","222");
        redisTemplate.opsForValue().set("333","333");
        redisTemplate.opsForValue().set("444","444");
        System.out.println("缓存已经设置完毕。。。。。。。");
        String result1=redisTemplate.opsForValue().get("111").toString();
        String result2=redisTemplate.opsForValue().get("222").toString();
        String result3=redisTemplate.opsForValue().get("333").toString();
        System.out.println("缓存结果为：result："+result1+"   "+result2+"    "+result3);
    }
```

5. RedisTemplate 储存对象

```
/**
     * redis 储存对象
     */
```

```
@RequestMapping("/Object")
public void redisObject(){
    User user = new User();
    user.setId(11);
    user.setName("yi");
    user.setPhone("123456");
    user.setAge(22);
    redisTemplate.opsForValue().set("user", user);
    User users = (User) redisTemplate.opsForValue().get("user");
    System.out.println(users);
}
```

6. StringRedisTemplate 储存对象

```
/**
 * 存储 json 字符串对象和取出
 */
@RequestMapping("/template")
public void StringRedisTemplate(){
    User user = new User();
    user.setId(11);
    user.setName("yi");
    user.setPhone("123456");
    user.setAge(22);
    template.opsForValue().set("user", JSON.toJSONString(user));
    String str = template.opsForValue().get("user");
    User userName = JSON.parseObject(str, User.class);
    System.out.println(userName);
}
```

7. redis 之 pipeline

```
public String tsetRedis(){
    Long time = System.currentTimeMillis();
    for (int i = 0; i < 10000; i++) {
        stringRedisTemplate.opsForValue().set("yi" + i, "wo" + i);
    }
    Long time1 = System.currentTimeMillis();
    System.out.println("耗时： " + (time1 - time));
    long time4 = System.currentTimeMillis();
    stringRedisTemplate.executePipelined(new SessionCallback<Object>() {
        @Override
        public <K, V> Object execute(RedisOperations<K, V> redisOperations) throws
DataAccessException {
            for (int i = 0; i < 10000; i++) {
                stringRedisTemplate.opsForValue().set("qiang" + i, "wo" + i);
            }
```

```
                return null;    //RedisTemplate 执行 executePipelined 方法是有返回值的
            }
        });
        Long time2 = System.currentTimeMillis();
        System.out.println("耗时： " + (time2 - time4));
        return "redis 正常耗时： " + (time1 - time) + "<br/>" + "redis 管道耗时： " + (time2 - time4);
    }
```

消息队列

28、消息队列的使用场景

校验用户名等信息，如果没问题会在数据库中添加一个用户记录
如果是用邮箱注册会给你发送一封注册成功的邮件，手机注册则会发送一条短信
分析用户的个人信息，以便将来向他推荐一些志同道合的人，或向那些人推荐他
发送给用户一个包含操作指南的系统通知

29、消息的重发补偿解决思路

可靠消息服务定时查询状态为已发送并超时的消息
可靠消息将消息重新投递到 MQ 组件中
下游应用监听消息，在满足幂等性的条件下，重新执行业务。
下游应用通知可靠消息服务该消息已经成功消费。
通过消息状态确认和消息重发两个功能，可以确保上游应用、可靠消息服务和下游应用数据的最终一致性。

30、消息的幂等性解决思路

1 查询操作

查询一次和查询多次，在数据不变的情况下，查询结果是一样的。select 是天然的幂等操作

2 删除操作

删除操作也是幂等的，删除一次和多次删除都是把数据删除。(注意可能返回结果不一样，删除的数据不存在，返回 0，删除的数据多条，返回结果多个)

3 唯一索引，防止新增脏数据

比如：支付宝的资金账户，支付宝也有用户账户，每个用户只能有一个资金账户，怎么防止给用户创建资金账户多个，那么给资金账户表中的用户 ID 加唯一索引，所以一个用户新增成功一个资金账户记录

4token 机制，防止页面重复提交

5 悲观锁

获取数据的时候加锁获取

```
select * from table_xxx where id='xxx' for update;
```

注意：id 字段一定是主键或者唯一索引，不然是锁表，会死人的

悲观锁使用时一般伴随事务一起使用，数据锁定时间可能会很长，根据实际情况选用

乐观锁

乐观锁只是在更新数据那一刻锁表，其他时间不锁表，所以相对于悲观锁，效率更高。

6 分布式锁

还是拿插入数据的例子，如果是分布是系统，构建全局唯一索引比较困难，例如唯一性的字段没法确定，这时候可以引入分布式锁，通过第三方的系统(redis 或 zookeeper)，在业务系统插入数据或者更新数据，获取分布式锁，然后做操作，之后释放锁，这样其实是把多线程并发的锁的思路，引入多多个系统，也就是分布式系统中得解决思路。

7select + insert

并发不高的后台系统，或者一些任务 JOB，为了支持幂等，支持重复执行，简单的处理方法是，先查询下一些关键数据，判断是否已经执行过，在进行业务处理，就可以了

注意：核心高并发流程不要用这种方法

8 状态机幂等

在设计单据相关的业务，或者是任务相关的业务，肯定会涉及到状态机(状态变更图)，就是业务单据上面有个状态，状态在不同的情况下会发生变更，一般情况下存在有限状态机，这时候，如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

9 对外提供接口的 api 如何保证幂等

如银联提供的付款接口：需要接入商户提交付款请求时附带：source 来源，seq 序列号

source+seq 在数据库里面做唯一索引，防止多次付款，(并发时，只能处理一个请求)

31、消息的堆积解决思路

如果还没开始投入使用 kafka，那应该在设计分区数的时候，尽量设置的多点（当然也不要太大，太大影响延迟，具体可以参考我前面提到的文章），从而提升生产和消费的并行度，避免消费太慢导致消费堆积。

增大批次

瓶颈在消费吞吐量的时候，增加批次也可以改善性能

增加线程数

如果一些消费者组中的消费者线程还是有 1 个消费者线程消费多个分区的情况，建议增加消费者线程。尽量 1 个消费者线程对应 1 个分区，从而发挥现有分区数下的最大并行度。

32、自己如何实现消息队列

大体上的设计是由一条线程 1 执行从等待列表中获取任务插入任务队列再由线程池中的线程从任务队列中取出任务去执行。

添加一条线程 1 主要是防止在执行耗时的任务时阻塞主线程.当执行耗时任务时,添加的任务的操作快于取出任务的操作。

当任务队列长度达到最大值时,线程 1 将被阻塞,等待线程 2,3...从任务队列取出任务执行。

33、如何保证消息的有序性

通过轮询所有队列的方式来确定消息被发送到哪一个队列（负载均衡策略）。订单号相同的消息会被先后发送到同一个队列中，在获取到路由信息以后，会根据算法来选择一个队列，同一个 OrderId 获取到的肯定是同一个队列。

重点篇

MySQL 重点

1. 锁

MyISAM 和 InnoDB 存储引擎使用的锁：
MyISAM 采用表级锁(table-level locking)。
InnoDB 支持行级锁(row-level locking)和表级锁,默认为行级锁
表级锁和行级锁对比：
表级锁： Mysql 中锁定 粒度最大 的一种锁，对当前操作的整张表加锁，实现简单，资源消耗也比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM 和 InnoDB 引擎都支持表级锁。
行级锁： Mysql 中锁定 粒度最小 的一种锁，只针对当前操作的行进行加锁。 行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。
InnoDB 存储引擎的锁的算法有三种：
Record lock：单个行记录上的锁
Gap lock：间隙锁，锁定一个范围，不包括记录本身
Next-key lock：record+gap 锁定一个范围，包含记录本身

2. 大表优化

当 MySQL 单表记录数过大时，数据库的 CRUD 性能会明显下降，一些常见的优化措施如下：

限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。；

读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；

缓存： 使用 MySQL 的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的缓存；

垂直分区：

根据数据库里面数据表的相关性进行拆分。 例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。 如下图所示，这样来说大家应该就更容易理解了。

垂直拆分的优点： 可以使得行数据变小，在查询时减少读取的 Block 数，减少 I/O 次数。此外，垂直分区可以简化表的结构，易于维护。

垂直拆分的缺点： 主键会出现冗余，需要管理冗余列，并会引起 Join 操作，可以通过在应用层进行 Join 来解决。此外，垂直分区会让事务变得更加复杂。

水平分区：

保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或

者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。

水平拆分是指数据表行的拆分，表的行数超过 200 万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。

水品拆分可以支持非常大的数据量。需要注意的一点是:分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升 MySQL 并发能力没有什么意义，所以 水品拆分最好分库 。

水平拆分能够 支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决 ，跨界点 Join 性能较差，逻辑复杂。《Java 工程师修炼之道》的作者推荐 尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度 ，一般的数据表在优化得当的情况下支撑千万以下的数据量是没有太大问题的。如果实在要分片，尽量选择客户端分片架构，这样可以减少一次和中间件的网络 I/O。

3.MySQL 的复制原理以及流程

基本原理流程，3 个线程以及之间的关联；

1. 主：binlog 线程——记录下所有改变了数据库数据的语句，放进 master 上的 binlog 中；

2. 从：io 线程——在使用 start slave 之后，负责从 master 上拉取 binlog 内容，放进 自己的 relay log 中；

3. 从：sql 执行线程——执行 relay log 中的语句。

详解：mysql 主从复制

MySQL 数据库自身提供的主从复制功能可以方便的实现数据的多处自动备份，实现数据库的拓展。多个数据备份不仅可以加强数据的安全性，通过实现读写分离还能进一步提升数据库的负载性能。

在一主多从的数据库体系中，多个从服务器采用异步的方式更新主数据库的变化，业务服务器在执行写或者相关修改数据库的操作是在主服务器上进行的，读操作则是在各从服务器上进行。如果配置了多个从服务器或者多个主服务器又涉及到相应的负载均衡问题，关于负载均衡具体的技术细节还没有研究过，今天就先简单的实现一主一从的主从复制功能。

4. Mysql 中的 myisam 与 innodb 的区别

1、InooDB 支持事务，而 MyISAM 不支持事务

2、InnoDB 支持行级锁，而 MyISAM 支持表级锁

3、InnoDB 支持 MVCC，而 MyISAM 不支持

4、InnoDB 支持外键，而 MyISAM 不支持

5、InnoDB 不支持全文索引，而 MyISAM 支持

<2>InnoDB 引擎的四大特性：插入缓冲，二次写，自适应哈希索引，预读

<3>InooDB 和 MyISAM 的 select count（*）哪个更快，为什么

myisam 更快，因为 myisam 内部维护了一个计算器，可以直接调取。MyISAM 的索引和数据是分开的，并且索引是有压缩的，内存使用率就对应提高了不少。能加载更多索引，而 Innodb 是索引和数据是紧密捆绑的，没有使用压缩从而会造成 Innodb 比 MyISAM 体积庞大不小。

5. InnoDB 的事务与日志的实现方式

错误日志：记录出错信息，也记录一些警告信息或者正确的信息

查询日志：记录所有对数据库请求的信息，不论这些请求是否得到了正确的执行

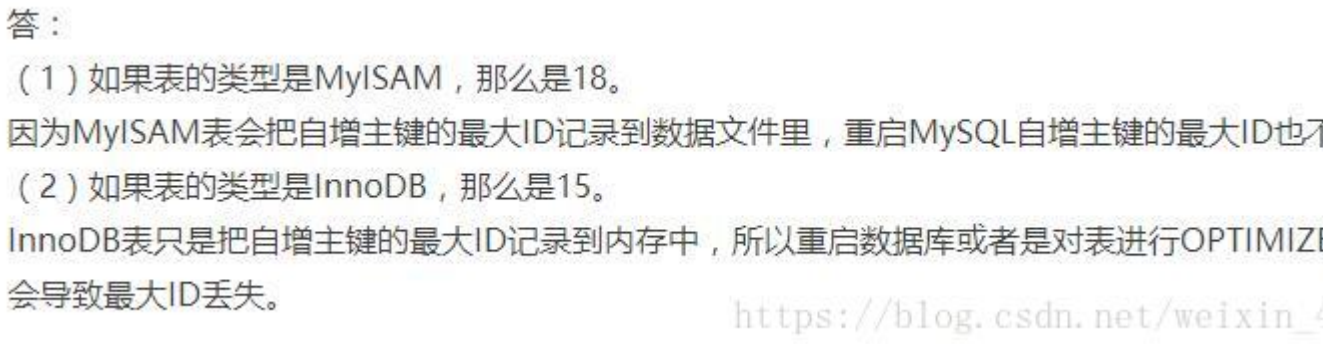
慢查询日志：设置一个阈值，将运行时间超过该值的所有 SQL 语句都记录到慢查询的日志文件中

二进制日志：记录对数据库执行更改的所有操作

中继日志，事务日志。

6. 一张表里面有 ID 自增主键

一张表里面有 ID 自增主键，当 insert 了 17 条记录之后，删除了第 15,16,17 条记录，再把 mysql 重启，再 insert 一条记录，这条记录的 ID 是 18 还是 15 ？



7. 哈希索引的优势

等值查询。哈希索引具有绝对优势（前提是：没有大量重复键值，如果大量重复键值时，哈希索引的效率很低，因为存在所谓的哈希碰撞问题。）

哈希索引不适用的场景：

不支持范围查询

不支持索引完成排序

不支持联合索引的最左前缀匹配规则

通常，B+树索引结构适用于绝大多数场景，像下面这种场景用哈希索引才更有优势：

在 HEAP 表中，如果存储的数据重复度很低（也就是说基数很大），对该列数据以等值查询为主，没有范围查询、没有排序的时候，特别适合采用哈希索引。

而常用的 InnoDB 引擎中默认使用的是 B+树索引，它会实时监控表上索引的使用情况，如果认为建立哈希索引可以提高查询效率，则自动在内存中的“自适应哈希索引缓冲区”建立哈希索引（在 InnoDB 中默认开启自适应哈希索引），通过观察搜索模式，MySQL 会利用 index key 的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快等值查询。

注意：在某些工作负载下，通过哈希索引查找带来的性能提升远大于额外的监控索引搜索情况和保持这个哈希表结构所带来的开销。但某些时候，在负载高的情况下，自适应哈希索引中添加的 read/write 锁也会带来竞争，比如高并发的 join 操作。like 操作和%的通配符操作也不适用于自适应哈希索引，可能要关闭自适应哈希索引。

8. B tree/B+tree

1、B 树，每个节点都存储 key 和 data，所有的节点组成这可树，并且叶子节点指针为 null，叶子节点不包含任何关键字信息

2、B+树，所有的叶子节点中包含全部关键字的信息，及指向含有这些关键字记录的指针，且叶子节点本身依关键字的大小自小到大的顺序链接，所有的非终端节点可以看成是索引部分，节点中仅含有其子树根节点中最大（或最小）关键字

3、为什么说 B+比 B 树更适合实际应用中操作系统的文件索引和数据库索引？

B+的磁盘读写代价更低 B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了。

B+-tree 的查询效率更加稳定 由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

9. 数据库范式

1 第一范式(1NF)

在任何一个关系数据库中，第一范式(1NF)是对关系模式的基本要求，不满足第一范式(1NF)的数据库就不是关系数据库。

所谓第一范式(1NF)是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式(1NF)中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

2 第二范式(2NF)

第二范式(2NF)是在第一范式(1NF)的基础上建立起来的，即满足第二范式(2NF)必须先满足第一范式(1NF)。第二范式(2NF)要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键、主码。

第二范式(2NF)要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。简而言之，第二范式就是非主属性非部分依赖于主关键字。

3 第三范式(3NF)

满足第三范式(3NF)必须先满足第二范式(2NF)。简而言之，第三范式(3NF)要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。例如，存在一个部门信息表，其中每个部门有部门编号(dept_id)、部门名称、部门简介等信息。那么在员工信息表中列出部门编号后就不能再将部门名称、部门简介等与部门有关的信息再加入员工信息表中。如果不存在部门信息表，则根据第三范式(3NF)也应该构建它，否则就会有大量的数据冗余。简而言之，第三范式就是属性不依赖于其它非主属性。(我的理解是消除冗余)

10. 如何设计一个高并发的系统

① 数据库的优化，包括合理的事务隔离级别、SQL 语句优化、索引的优化

② 使用缓存，尽量减少数据库 IO

③ 分布式数据库、分布式缓存
④ 服务器的负载均衡

11.锁的优化策略

① 读写分离
② 分段加锁
③ 减少锁持有的时间
④ 多个线程尽量以相同的顺序去获取资源
等等，这些都不是绝对原则，都要根据情况，比如不能将锁的粒度过于细化，不然可能会出现线程的加锁和释放次数过多，反而效率不如一次加一把大锁。

12.了解 XSS 攻击吗？如何防止？

XSS 是跨站脚本攻击，首先是利用跨站脚本漏洞以一个特权模式去执行攻击者构造的脚本，然后利用不安全的 Activex 控件执行恶意的行为。
使用 htmlspecialchars()函数对提交的内容进行过滤，使字符串里面的特殊符号实体化。

13.SQL 注入漏洞产生的原因？如何防止？

SQL 注入产生的原因：程序开发过程中不注意规范书写 sql 语句和对特殊字符进行过滤，导致客户端可以通过全局变量 POST 和 GET 提交一些 sql 语句正常执行。
防止 SQL 注入的方式：
开启配置文件中的 magic_quotes_gpc 和 magic_quotes_runtime 设置
执行 sql 语句时使用 addslashes 进行 sql 语句转换
Sql 语句书写尽量不要省略双引号和单引号。
过滤掉 sql 语句中的一些关键词：update、insert、delete、select、 * 。
提高数据库表和字段的命名技巧，对一些重要的字段根据程序的特点命名，取不易被猜到的。
Php 配置文件中设置 register_globals 为 off,关闭全局变量注册
控制错误信息，不要在浏览器上输出错误信息，将错误信息写到日志文件中。

14. 存储时期

Datatime:以 YYYY-MM-DD HH:MM:SS 格式存储时期时间，精确到秒，占用 8 个字节得存储空间，datatime 类型与时区无关

Timestamp:以时间戳格式存储，占用 4 个字节，范围小 1970-1-1 到 2038-1-19，显示依赖于所指定得时区，默认在第一个列行的数据修改时可以自动得修改 timestamp 列得值

Date:（生日）占用得字节数比使用字符串.datatime.int 储存要少，使用 date 只需要 3 个字节，存储日期月份，还可以利用日期时间函数进行日期间得计算

Time:存储时间部分得数据

注意:不要使用字符串类型来存储日期时间数据（通常比字符串占用得储存空间小，在进行查找过滤可以利用日期得函数）

使用 int 存储日期时间不如使用 timestamp 类型

JVM 重点

15. 运行时数据区域

Java 虚拟机管理的内存包括几个运行时数据内存：方法区、虚拟机栈、本地方法栈、堆、程序计数器，其中方法区和堆是由线程共享的数据区，其他几个是线程隔离的数据区

1.1 程序计数器

程序计数器是一块较小的内存，他可以看做是当前线程所执行的行号指示器。字节码解释器工作的时候就是通过改变这个计数器的值来选取下一条需要执行的字节码的指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。如果线程正在执行的是一个 Java 方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是 Native 方法，这个计数器则为空。此内存区域是唯一一个在 Java 虚拟机规范中没有规定任何 OutOfMemotyError 情况的区域

1.2 Java 虚拟机栈

虚拟机栈描述的是 Java 方法执行的内存模型：每个方法在执行的同时都会创建一个栈帧用于储存局部变量表、操作数栈、动态链接、方法出口等信息。每个方法从调用直至完成的过程，就对应着一个栈帧在虚拟机栈中入栈到出栈的过程。

栈内存就是虚拟机栈，或者说是虚拟机栈中局部变量表的部分

局部变量表存放了编辑期可知的各种基本数据类型（boolean、byte、char、short、int、float、long、double）、对象引用（refrence）类型和 returnAddress 类型（指向了一条字节码指令的地址）

其中 64 位长度的 long 和 double 类型的数据会占用两个局部变量空间，其余的数据类型只占用 1 个。

Java 虚拟机规范对这个区域规定了两种异常状况：如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常。如果虚拟机扩展时无法申请到足够的内存，就会跑出 OutOfMemoryError 异常

1.3 本地方法栈

本地方法栈和虚拟机栈发挥的作用是非常类似的，他们的区别是虚拟机栈为虚拟机执行 Java 方法（也就是字节码）服务，而本地方法栈则为虚拟机使用到的 Native 方法服务

本地方法栈区域也会抛出 StackOverflowError 和 OutOfMemoryErroy 异常

1.4 Java 堆

堆是 Java 虚拟机所管理的内存中最大的一块。Java 堆是被所有线程共享的一块内存区域，在虚拟机启动的时候创建，此内存区域的唯一目的是存放对象实例，几乎所有的对象实例都在这里分配内存。所有的对象实例和数组都在堆上分配

Java 堆是垃圾收集器管理的主要区域。Java 堆细分为新生代和老年代

不管怎样，划分的目的都是为了更好的回收内存，或者更快地分配内存

Java 堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有完成实例分配，并且堆也无法在扩展时将会抛出 OutOfMemoryError 异常

1.5 方法区

方法区它用于储存已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据

除了 Java 堆一样不需要连续的内存和可以选择固定大小或者可扩展外，还可以选择不实现垃圾收集。这个区域的内存回收目标主要是针对常量池的回收和对类型的卸载

当方法区无法满足内存分配需求时，将抛出 OutOfMemoryErroy 异常

1.6 运行时常量池

它是方法区的一部分。Class 文件中除了有关的版本、字段、方法、接口等描述信息外、还有一项信息是常量池，用于存放编辑期生成的各种字面量和符号引用，这部分内容将在类加载后进入方法区的运行时常量池中存放

Java 语言并不要求常量一定只有编辑期才能产生，也就是可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern()方法

当常量池无法再申请到内存时会抛出 OutOfMemoryError 异常

16. hotspot 虚拟机对象

2.1 对象的创建

1）.检查

虚拟机遇到一条 new 指令时，首先将去检查这个指令的参数是否能在常量池中定位到一个类的符号引用，并且检查这个符号引用代表的类是否已经被加载、解析和初始化过。如果没有，那必须先执行相应的类加载过程

2）分配内存

接下来将为新生对象分配内存，为对象分配内存空间的任务等同于把一块确定的大小的内存从 Java 堆中划分出来。

假设 Java 堆中内存是绝对规整的，所有用过的内存放在一遍，空闲的内存放在另一边，中间放着一个指针作为分界点的指示器，那所分配内存就仅仅是把那个指针指向空闲空间那边挪动一段与对象大小相等的距离，这个分配方式叫做“指针碰撞”

如果 Java 堆中的内存并不是规整的，已使用的内存和空闲的内存相互交错，那就没办法简单地进行指针碰撞了，虚拟机就必须维护一个列表，记录上哪些内存块是可用的，在分配的时候从列表中找到一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式成为“空闲列表”

选择那种分配方式由 Java 堆是否规整决定，而 Java 堆是否规整又由所采用的垃圾收集器是否带有压缩整理功能决定。

3）Init

执行 new 指令之后会接着执行 Init 方法，进行初始化，这样一个对象才算产生出来

2.2 对象的内存布局

在 HotSpot 虚拟机中，对象在内存中储存的布局可以分为 3 块区域：对象头、实例数据和对齐填充

对象头包括两部分：

1). 储存对象自身的运行时数据，如哈希码、GC 分带年龄、锁状态标志、线程持有的锁、偏向线程 ID、偏向时间戳

2). 另一部分是指类型指针，即对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是那个类的实例

2.3 对象的访问定位

1).使用句柄访问

Java 堆中将会划分出一块内存来作为句柄池，reference 中存储的就是对象的句柄地址，而句柄中包含了对象实例数据与类型数据各自的具体地址

优势:reference 中存储的是稳点的句柄地址,在对象被移动(垃圾收集时移动对象是非常普遍的行为)时只会改变句柄中的实例数据指针，而 reference 本身不需要修改

2).使用直接指针访问

Java 堆对象的布局就必须考虑如何访问类型数据的相关信息,而 refreence 中存储的直接

就是对象的地址

优势：速度更快，节省了一次指针定位的时间开销，由于对象的访问在 Java 中非常频繁，因此这类开销积少成多后也是一项非常可观的执行成本

17.OutOfMemoryError 异常

3.1 Java 堆溢出

Java 堆用于存储对象实例，只要不断的创建对象，并且保证 GCRoots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，那么在数量到达最大堆的容量限制后就会产生内存溢出异常

如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄露对象是通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收它们的。掌握了泄漏对象的类型信息及 GC Roots 引用链的信息，就可以比较准确地定位出泄漏代码的位置

如果不存在泄露，换句话说，就是内存中的对象确实都还必须存活着，那就应当检查虚拟机的堆参数（-Xmx 与-Xms），与机器物理内存对比看是否还可以调大，从代码上检查是否存在某些对象生命周期过长、持有状态时间过长的情况，尝试减少程序运行期的内存消耗

3.2 虚拟机栈和本地方法栈溢出

对于 HotSpot 来说，虽然-Xoss 参数（设置本地方法栈大小）存在，但实际上是无效的，栈容量只由-Xss 参数设定。关于虚拟机栈和本地方法栈，在 Java 虚拟机规范中描述了两种异常：

如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError

如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常

在单线程下，无论由于栈帧太大还是虚拟机栈容量太小，当内存无法分配的时候，虚拟机抛出的都是 StackOverflowError 异常

如果是多线程导致的内存溢出，与栈空间是否足够大并不存在任何联系，这个时候每个线程的栈分配的内存越大，反而越容易产生内存溢出异常。解决的时候是在不能减少线程数或更换 64 为的虚拟机的情况下，就只能通过减少最大堆和减少栈容量来换取更多的线程

3.3 方法区和运行时常量池溢出

String.intern()是一个 Native 方法，它的作用是：如果字符串常量池中已经包含一个等于此 String 对象的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用

由于常量池分配在永久代中，可以通过-XX:PermSize 和-XX:MaxPermSize 限制方法区大小，从而间接限制其中常量池的容量。

Intern():

JDK1.6 intern 方法会把首次遇到的字符串实例复制到永久代，返回的也是永久代中这个字符串实例的引用，而由 StringBuilder 创建的字符串实例在 Java 堆上，所以必然不是一个引用

JDK1.7 intern()方法的实现不会再复制实例，只是在常量池中记录首次出现的实例引用，因此 intern()返回的引用和由 StringBuilder 创建的那个字符串实例是同一个。

18.垃圾收集

程序计数器、虚拟机栈、本地方法栈 3 个区域随线程而生，随线程而灭，在这几个区域

内就不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟随着回收了

1.判断对象存活

4.1.1 引用计数器法

给对象添加一个引用计数器，每当由一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器为 0 的对象就是不可能再被使用的

4.1.2 可达性分析算法

通过一系列的成为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径成为引用链，当一个对象到 GC ROOTS 没有任何引用链相连时，则证明此对象时不可用的

Java 语言中 GC Roots 的对象包括下面几种：

1.虚拟机栈（栈帧中的本地变量表）中引用的对象

2.方法区中类静态属性引用的对象

3.方法区中常量引用的对象

4.本地方法栈 JNI（Native 方法）引用的对象

2.引用

强引用就是在程序代码之中普遍存在的，类似 Object obj = new Object() 这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象

软引用用来描述一些还有用但并非必须的元素。对于它在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收，如果这次回收还没有足够的内存才会抛出内存溢出异常

弱引用用来描述非必须对象的，但是它的强度比软引用更弱一些，被引用关联的对象只能生存到下一次垃圾收集发生之前，当垃圾收集器工作时，无论当前内存是否足够都会回收掉只被弱引用关联的对象

虚引用的唯一目的就是能在这个对象被收集器回收时收到一个系统通知

3.Finalize 方法

任何一个对象的 finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的 finalize()方法不会被再次执行，因此第二段代码的自救行动失败了

4.3.1 回收方法区

永久代的垃圾收集主要回收两部分内容：废弃常量和无用的类

废弃常量：假如一个字符串 abc 已经进入了常量池中，如果当前系统没有任何一个 String 对象 abc，也就是没有任何 Stirng 对象引用常量池的 abc 常量，也没有其他地方引用的这个字面量，这个时候发生内存回收这个常量就会被清理出常量池

无用的类：

1.该类所有的实例都已经被回收，就是 Java 堆中不存在该类的任何实例

2.加载该类的 ClassLoader 已经被回收

3.该类对用的 java.lang.Class 对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法

4.垃圾收集算法

4.4.1 标记—清除算法

算法分为标记和清除两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象、

不足:一个是效率问题，标记和清除两个过程的效率都不高；另一个是空间问题，标记清楚之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后再程序运行过程中需要分配较大的对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作

4.4.2 复制算法

他将可用内存按照容量划分为大小相等的两块，每次只使用其中的一块。当这块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可

不足：将内存缩小为了原来的一半

实际中我们并不需要按照 1:1 比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor

当另一个 Survivor 空间没有足够空间存放上一次新生代收集下来的存活对象时，这些对象将直接通过分配担保机制进入老年代

4.4.3 标记整理算法

让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存

4.4.4 分代收集算法

只是根据对象存活周期的不同将内存划分为几块。一般是把 java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记清理或者标记整理算法来进行回收

5.垃圾收集器

a)Serial 收集器：

这个收集器是一个单线程的收集器，但它的单线程的意义不仅仅说明它会只使用一个 COU 或一条收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集时，必须暂停其他所有的工作线程，直到它手机结束

b)ParNew 收集器：

Serial 收集器的多线程版本，除了使用了多线程进行收集之外，其余行为和 Serial 收集器一样

并行：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态

并发：指用户线程与垃圾收集线程同时执行（不一定是并行的，可能会交替执行），用户程序在继续执行，而垃圾收集程序运行于另一个 CPU 上

c)Parallel Scavenge

收集器是一个新生代收集器，它是使用复制算法的收集器，又是并行的多线程收集器。

吞吐量：就是 CPU 用于运行用户代码的时间与 CPU 总消耗时间的比值。即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）

d)Serial Old 收集器：

是 Serial 收集器的老年代版本,是一个单线程收集器，使用标记整理算法

e)Parallel Old 收集器：

Parallel Old 是 Paraller Seavenge 收集器的老年代版本，使用多线程和标记整理算法

f)CMS 收集器：

CMS 收集器是基于标记清除算法实现的，整个过程分为 4 个步骤：

1.初始标记 2.并发标记 3.重新标记 4.并发清除

优点：并发收集、低停顿

缺点：

1.CMS 收集器对 CPU 资源非常敏感，CMS 默认启动的回收线程数是（CPU 数量+3）/4，

2.CMS 收集器无法处理浮动垃圾，可能出现 Failure 失败而导致一次 Full G 场地产生

3.CMS 是基于标记清除算法实现的

g)G1 收集器：

它是一款面向服务器应用的垃圾收集器

1.并行与并发：利用多 CPU 缩短 STOP-The-World 停顿的时间

2.分代收集

3.空间整合：不会产生内存碎片

4.可预测的停顿

运作方式：初始标记，并发标记，最终标记，筛选回收

6.内存分配与回收策略

4.6.1 对象优先在 Eden 分配：

大多数情况对象在新生代 Eden 区分配，当 Eden 区没有足够空间进行分配时，虚拟机将发起一次 Minor GC

4.6.2 大对象直接进入老年代：

所谓大对象就是指需要大量连续内存空间的 Java 对象，最典型的大对象就是那种很长的字符串以及数组。这样做的目的是避免 Eden 区及两个 Servivor 之间发生大量的内存复制

4.6.3 长期存活的对象将进入老年代

如果对象在 Eden 区出生并且尽力过一次 Minor GC 后仍然存活，并且能够被 Servivor 容纳，将被移动到 Servivor 空间中，并且把对象年龄设置成为 1.对象在 Servivor 区中每熬过一次 Minor GC，年龄就增加 1 岁，当它的年龄增加到一定程度（默认 15 岁），就将会被晋级到老年代中

4.6.4 动态对象年龄判定

为了更好地适应不同程序的内存状况，虚拟机并不是永远地要求对象的年龄必须达到了 MaxTenuringThreshold 才能晋级到老年代，如果在 Servivor 空间中相同年龄所有对象的大小总和大于 Survivor 空间的一半，年龄大于或等于该年龄的对象就可以直接进入到老年代，无须登到 MaxTenuringThreshold 中要求的年龄

4.6.4 空间分配担保：

在发生 Minor GC 之前，虚拟机会检查老年代最大可 用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么 Minor DC 可以确保是安全的。如果不成立，则虚拟机会查看 HandlePromotionFailure 设置值是否允许担保失败。如果允许那么会继续检查老年代最大可用的连续空间是否大于晋级到老年代对象的平均大小，如果大于，将尝试进行一次 Minor GC，尽管这次 MinorGC 是有风险的：如果小于，或者 HandlePromotionFailure 设置不允许冒险，那这时也要改为进行一次 Full GC

19. 虚拟机类加载机制

虚拟机吧描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制

在 Java 语言里面，类型的加载。连接和初始化过程都是在程序运行期间完成的

5.1 类加载的时机

类被加载到虚拟机内存中开始，到卸载为止，整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载 7 个阶段

加载、验证、准备、初始化和卸载这 5 个阶段的顺序是确定的，类的加载过程必须按照这种顺序按部就班地开始，而解析阶段则不一定：它在某些情况下可以再初始化阶段之后再开始，这个是为了支持 Java 语言运行时绑定（也成为动态绑定或晚期绑定）

虚拟机规范规定有且只有 5 种情况必须立即对类进行初始化：

1.遇到 new、getstatic、putstatic 或 invokestatic 这 4 条字节码指令时，如果类没有进行过初始化，则需要触发其初始化。生成这 4 条指令的最常见的 Java 代码场景是：使用 new 关键字实例化对象的时候、读取或设置一个类的静态字段（被 final 修饰、已在编译期把结果放入常量池的静态字段除外）的时候，以及调用一个类的静态方法的时候

2.使用 java.lang.reflect 包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化

3.当初始化一个类的时候，如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化

4.当虚拟机启动时候，用户需要指定一个要执行的主类（包含 main()方法的那个类），虚拟机会先初始化这个主类

5.当使用 JDK1.7 的动态语言支持时，如果一个 java.lang.invoke.MethodHandle 实例最后的解析结果 REF_getStatic、REF_putStatic、REF_invokeStatic 的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化

被动引用：

1.通过子类引用父类的静态字段，不会导致子类初始化

2.通过数组定义来引用类，不会触发此类的初始化

3.常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化

接口的初始化：接口在初始化时，并不要求其父接口全部完成类初始化，只有在正整使用到父接口的时候（如引用接口中定义的常量）才会初始化

5.2 类加载的过程

5.2.1 加载

1)通过一个类的全限定名类获取定义此类的二进制字节流

2)将这字节流所代表的静态存储结构转化为方法区运行时数据结构

3)在内存中生成一个代表这个类的 java.lang.Class 对象，作为方法区这个类的各种数据的访问入口

怎么获取二进制字节流？

1)从 ZIP 包中读取，这很常见，最终成为日后 JAR、EAR、WAR 格式的基础

2)从网络中获取，这种场景最典型的应用就是 Applet

3)运行时计算生成，这种常见使用得最多的就是动态代理技术

4)由其他文件生成，典型场景就是 JSP 应用

5)从数据库中读取，这种场景相对少一些（中间件服务器）

数组类本身不通过类加载器创建，它是由 Java 虚拟机直接创建的

数组类的创建过程遵循以下规则：

1)如果数组的组件类型(指的是数组去掉一个维度的类型)是引用类型，那就递归采用上面的加载过程去加载这个组件类型，数组 C 将在加载该组件类型的类加载器的类名称空间上被标识

2)如果数组的组件类型不是引用类型(列如 int[]组数)，Java 虚拟机将会把数组 C 标识为与引导类加载器关联

3)数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为 public

5.2.2 验证

验证阶段会完成下面 4 个阶段的检验动作：文件格式验证，元数据验证，字节码验证，符号引用验证

1.文件格式验证

第一阶段要验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。这一阶段可能包括：

1.是否以魔数 oxCAFEBABE 开头

2.主、次版本号是否在当前虚拟机处理范围之内

3.常量池的常量中是否有不被支持的常量类型(检查常量 tag 标志)

4.指向常量的各种索引值中是否有指向不存在的常量或不符合类型的常量

5.CONSTANT_Itf8_info 型的常量中是否有不符合 UTF8 编码的数据

6.Class 文件中各个部分及文件本身是否有被删除的或附加的其他信息

这个阶段的验证时基于二进制字节流进行的，只有通过类这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面的 3 个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流

2.元数据验证

1.这个类是否有父类(除了 java.lang.Object 之外,所有的类都应当有父类)

2.这个类的父类是否继承了不允许被继承的类（被 final 修饰的类）

3.如果这个类不是抽象类，是否实现类其父类或接口之中要求实现的所有方法

4.类中的字段、方法是否与父类产生矛盾(列如覆盖类父类的 final 字段,或者出现不符合规则的方法重载，列如方法参数都一致，但返回值类型却不同等)

第二阶段的主要目的是对类元数据信息进行语义校验，保证不存在不符合 Java 语言规范的元数据信息

3.字节码验证

第三阶段是整个验证过程中最复杂的一个阶段，主要目的似乎通过数据流和控制流分析，确定程序语言是合法的、符合逻辑的。在第二阶段对元数据信息中的数据类型做完校验后，这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件。

1.保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作，列如，列如在操作数栈放置类一个 int 类型的数据，使用时却按 long 类型来加载入本地变量表中

2.保证跳转指令不会跳转到方法体以外的字节码指令上

3.保证方法体中的类型转换时有效的，列如可以把一个子类对象赋值给父类数据类型，这个是安全的,但是吧父类对象赋值给子类数据类型,甚至把对象赋值给与它毫无继承关系、完全不相干的一个数据类型，则是危险和不合法的

4.符号引用验证

发生在虚拟机将符号引用转化为直接引用的时候,这个转化动作将在连接的第三阶段——解析阶段中发生。

1.符号引用中通过字符串描述的全限定名是否能找到相对应的类

2.在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段

3.符号引用中的类、字段、方法的访问性是否可被当前类访问

对于虚拟机的类加载机制来说，验证阶段是非常重要的，但是不一定必要（因为对程序运行期没有影响）的阶段。如果全部代码都已经被反复使用和验证过，那么在实施阶段就可以考虑使用 Xverify：none 参数来关闭大部分的类验证措施，以缩短虚拟机类加载的时间

5.2.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段,这些变量都在方法区中

进行分配。这个时候进行内存分配的仅包括类变量(被 static 修饰的变量)，而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。其次，这里说的初始值通常下是数据类型的零值。

假设 public static int value = 123；

那变量 value 在准备阶段过后的初始值为 0 而不是 123，因为这时候尚未开始执行任何 Java 方法，而把 value 赋值为 123 的 putstatic 指令是程序被编译后，存放于类构造器<clinit>()方法之中，所以把 value 赋值为 123 的动作将在初始化阶段才会执行，但是如果使用 final 修饰，则在这个阶段其初始值设置为 123

5.2.4 解析

解析阶段是虚拟机将常量池内符号引用替换为直接引用的过

5.2.5 初始化

类的初始化阶段是类加载过程的最后一步，前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才正真开始执行类中定义的 Java 程序代码(或者说是字节码)

5.3 类的加载器

5.3.1 双亲委派模型：

只存在两种不同的类加载器：启动类加载器（Bootstrap ClassLoader），使用 C++实现，是虚拟机自身的一部分。另一种是所有其他的类加载器，使用 JAVA 实现，独立于 JVM，并且全部继承自抽象类 java.lang.ClassLoader.

启动类加载器（Bootstrap ClassLoader），负责将存放在<JAVA+HOME>\lib 目录中的，或者被-Xbootclasspath 参数所制定的路径中的，并且是 JVM 识别的（仅按照文件名识别，如 rt.jar，如果名字不符合，即使放在 lib 目录中也不会被加载），加载到虚拟机内存中，启动类加载器无法被 JAVA 程序直接引用。

扩展类加载器，由 sun.misc.Launcher$ExtClassLoader 实现，负责加载<JAVA_HOME>\lib\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用扩展类加载器。

应用程序类加载器（Application ClassLoader），由 sun.misc.Launcher$AppClassLoader 来实现。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader()方法的返回值，所以一般称它为系统类加载器。负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

这张图表示类加载器的双亲委派模型（Parents Delegation model）. 双亲委派模型要求除了顶层的启动加载类外，其余的类加载器都应当有自己的父类加载器。，这里类加载器之间的父子关系一般不会以继承的关系来实现，而是使用组合关系来复用父类加载器的代码。

5.3.2 双亲委派模型的工作过程是：

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都是应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

5.3.3 这样做的好处就是：

Java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object,它存放在 rt.jar 中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是

同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为 java.lang.object 的类，并放在程序的 ClassPath 中，那系统中将会出现多个不同的 Object 类，Java 类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱

就是保证某个范围的类一定是被某个类加载器所加载的，这就保证在程序中同 一个类不会被不同的类加载器加载。这样做的一个主要的考量，就是从安全层 面上，杜绝通过使用和 JRE 相同的类名冒充现有 JRE 的类达到替换的攻击方式

20. Java 内存模型与线程

6.1 内存间的交互操作

关于主内存与工作内存之间的具体交互协议，即一个变量如何从主内存拷贝到工作内存、如何从工作内存同步到主内存之间的实现细节，Java 内存模型定义了以下八种操作来完成：

lock（锁定）：作用于主内存的变量，把一个变量标识为一条线程独占状态。

unlock（解锁）：作用于主内存变量，把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

read（读取）：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中，以便随后的 load 动作使用

load（载入）：作用于工作内存的变量，它把 read 操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，把工作内存中的一个变量值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收到的值赋值给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中，以便随后的 write 的操作。

write（写入）：作用于主内存的变量，它把 store 操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行 read 和 load 操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行 store 和 write 操作。Java 内存模型只要求上述操作必须按顺序执行，而没有保证必须是连续执行。也就是 read 和 load 之间，store 和 write 之间是可以插入其他指令的，如对主内存中的变量 a、b 进行访问时，可能的顺 序是 read a，read b，load b， load a。

Java 内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：

不允许 read 和 load、store 和 write 操作之一单独出现

不允许一个线程丢弃它的最近 assign 的操作，即变量在工作内存中改变了之后必须同步到主内存中。

不允许一个线程无原因地（没有发生过任何 assign 操作）把数据从工作内存同步回主内存中。

一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化（load 或 assign）的变量。即就是对一个变量实施 use 和 store 操作之前，必须先执行过了 assign 和 load 操作。

一个变量在同一时刻只允许一条线程对其进行 lock 操作，但 lock 操作可以被同一条线程重复执行多次，多次执行 lock 后，只有执行相同次数的 unlock 操作，变量才会被解锁。lock

和 unlock 必须成对出现

如果对一个变量执行 lock 操作，将会清空工作内存中此变量的值，在执行引擎使用这个变量前需要重新执行 load 或 assign 操作初始化变量的值

如果一个变量事先没有被 lock 操作锁定，则不允许对它执行 unlock 操作；也不允许去 unlock 一个被其他线程锁定的变量。

对一个变量执行 unlock 操作之前，必须先把此变量同步到主内存中（执行 store 和 write 操作）。

6.2 重排序

在执行程序时为了提高性能，编译器和处理器经常会对指令进行重排序。重排序分成三种类型：

1.编译器优化的重排序。编译器在不改变单线程程序语义放入前提下，可以重新安排语句的执行顺序。

2.指令级并行的重排序。现代处理器采用了指令级并行技术来将多条指令重叠执行。如果不存在数据依赖性，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。由于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。

从 Java 源代码到最终实际执行的指令序列，会经过下面三种重排序：

为了保证内存的可见性，Java 编译器在生成指令序列的适当位置会插入内存屏障指令来禁止特定类型的处理器重排序。Java 内存模型把内存屏障分为 LoadLoad、LoadStore、StoreLoad 和 StoreStore 四种：

6.3 对于 volatile 型变量的特殊规则

当一个变量定义为 volatile 之后，它将具备两种特性：

第一：保证此变量对所有线程的可见性，这里的可见性是指当一条线程修改了这个变量的值，新值对于其他线程来说是可以立即得知的。普通变量的值在线程间传递需要通过主内存来完成

由于 valatile 只能保证可见性，在不符合一下两条规则的运算场景中，我们仍要通过加锁来保证原子性

1.运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。

2.变量不需要与其他的状态变量共同参与不变约束

第二：禁止指令重排序，普通的变量仅仅会保证在该方法的执行过程中所有依赖赋值结果的地方都能获取到正确的结果，而不能保证变量赋值操作的顺序与程序代码中执行顺序一致，这个就是所谓的线程内表现为串行的语义

Java 内存模型中对 volatile 变量定义的特殊规则。假定 T 表示一个线程，V 和 W 分别表示两个 volatile 变量，那么在进行 read、load、use、assign、store、write 操作时需要满足如下的规则：

1.只有当线程 T 对变量 V 执行的前一个动作是 load 的时候，线程 T 才能对变量 V 执行 use 动作；并且，只有当线程 T 对变量 V 执行的后一个动作是 use 的时候，线程 T 才能对变量 V 执行 load 操作。线程 T 对变量 V 的 use 操作可以认为是与线程 T 对变量 V 的 load 和 read 操作相关联的，必须一起连续出现。这条规则要求在工作内存中，每次使用变量 V 之前都必须先从主内存刷新最新值，用于保证能看到其它线程对变量 V 所作的修改后的值。

2.只有当线程 T 对变量 V 执行的前一个动是 assign 的时候，线程 T 才能对变量 V 执行 store 操作；并且，只有当线程 T 对变量 V 执行的后一个动作是 store 操作的时候，线程 T 才能对变量 V 执行 assign 操作。线程 T 对变量 V 的 assign 操作可以认为是与线程 T 对变量 V 的 store 和 write 操作相关联的，必须一起连续出现。这一条规则要求在工作内存中，每次修改 V 后

都必须立即同步回主内存中，用于保证其它线程可以看到自己对变量 V 的修改。

3.假定操作 A 是线程 T 对变量 V 实施的 use 或 assign 动作，假定操作 F 是操作 A 相关联的 load 或 store 操作，假定操作 P 是与操作 F 相应的对变量 V 的 read 或 write 操作；类型地，假定动作 B 是线程 T 对变量 W 实施的 use 或 assign 动作，假定操作 G 是操作 B 相关联的 load 或 store 操作，假定操作 Q 是与操作 G 相应的对变量 V 的 read 或 write 操作。如果 A 先于 B，那么 P 先于 Q。这条规则要求 valitile 修改的变量不会被指令重排序优化，保证代码的执行顺序与程序的顺序相同。

6.4 对于 long 和 double 型变量的特殊规则

Java 模型要求 lock、unlock、read、load、assign、use、store、write 这 8 个操作都具有原子性，但是对于 64 为的数据类型（long 和 double），在模型中特别定义了一条相对宽松的规定：允许虚拟机将没有被 volatile 修饰的 64 位数据的读写操作分为两次 32 为的操作来进行，即允许虚拟机实现选择可以不保证 64 位数据类型的 load、store、read 和 write 这 4 个操作的原子性

6.5 原子性、可见性和有序性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。Java 内存模型是通过在变量修改后将新值同步会主内存，在变量读取前从主内存刷新变量值这种依赖主内存作为传递媒介的方式来实现可见性，valatile 特殊规则保障新值可以立即同步到祝内存中。Synchronized 是在对一个变量执行 unlock 之前，必须把变量同步回主内存中（执行 store、write 操作）。被 final 修饰的字段在构造器中一旦初始化完成，并且构造器没有吧 this 的引用传递出去，那在其他线程中就能看见 final 字段的值

可见性：可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

有序性：即程序执行的顺序按照代码的先后顺序执行。

6.6 先行发生原则

这些先行发生关系无须任何同步就已经存在，如果不再此列就不能保障顺序性，虚拟机就可以对它们任意地进行重排序

1.程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确的说，应该是控制顺序而不是程序代码顺序，因为要考虑分支。循环等结构

2.管程锁定规则：一个 unlock 操作先行发生于后面对同一个锁的 lock 操作。这里必须强调的是同一个锁，而后面的是指时间上的先后顺序

3.Volatile 变量规则：对一个 volatile 变量的写操作先行发生于后面对这个变量的读操作，这里的后面同样是指时间上的先后顺序

4.线程启动规则：Thread 对象的 start()方法先行发生于此线程的每一个动作

5.线程终止规则：线程中的所有操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join()方法结束、ThradisAlive()的返回值等手段检测到线程已经终止执行

6.线程中断规则：对线程 interrupt()方法的调用先行发生于被中断线程的代码检测到中断时间的发生，可以通过 Thread.interrupted()方法检测到是否有中断发生

7.对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize()方法的开始

8.传递性：如果操作 A 先行发生于操作 B，操作 B 先行发生于操作 C，那就可以得出操作 A 先行发生于操作 C 的结论

6.7 Java 线程调度

协同式调度：线程的执行时间由线程本身控制

抢占式调度：线程的执行时间由系统来分配

6.8 状态转换

1.新建

2.运行：可能正在执行。可能正在等待 CPU 为它分配执行时间

3.无限期等待：不会被分配 CUP 执行时间，它们要等待被其他线程显式唤醒

4.限期等待：不会被分配 CUP 执行时间，它们无须等待被其他线程显式唤醒，一定时间会由系统自动唤醒

5.阻塞：阻塞状态在等待这获取到一个排他锁，这个时间将在另一个线程放弃这个锁的时候发生；等待状态就是在等待一段时间，或者唤醒动作的发生

6.结束：已终止线程的线程状态，线程已经结束执行

21. 线程安全

1、不可变：不可变的对象一定是线程安全的、无论是对象的方法实现还是方法的调用者，都不需要再采取任何的线程安全保障。例如：把对象中带有状态的变量都声明为 final，这样在构造函数结束之后，它就是不可变的。

2、绝对线程安全

3、相对线程安全：相对的线程安全就是我们通常意义上所讲的线程安全，它需要保证对这个对象单独的操作是线程安全的，我们在调用的时候不需要做额外的保障措施，但是对于一些特定顺序的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性

4、线程兼容：对象本身并不是线程安全的，但是可以通过在调用端正确地使用同步手段来保证对象在并发环境中可以安全使用

5、线程对立：是指无论调用端是否采取了同步措施，都无法在多线程环境中并发使用的代码

7.1 线程安全的实现方法

1.互斥同步：

同步是指在多个线程并发访问共享数据时，保证共享数据在同一个时刻只被一个（或者是一些，使用信号量的时候）线程使用。而互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。互斥是因，同步是果：互斥是方法，同步是目的

在 Java 中，最基本的互斥同步手段就是 synchronized 关键字，它经过编译之后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令，这两个字节码都需要一个 reference 类型的参数来指明要锁定和解锁的对象。如果 Java 程序中的 synchronized 明确指定了对象参数，那就是这个对象的 reference；如果没有指明，那就根据 synchronized 修饰的是实例方法还是类方法，去取对应的对象实例或 Class 对象来作为锁对象。在执行 monitorenter 指令时，首先要尝试获取对象的锁。如果这个对象没有被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加 1，对应的在执行 monitorexit 指令时会将锁计数器减 1，当计数器为 0 时，锁就被释放。如果获取对象锁失败，哪当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止

Synchronized，ReentrantLock 增加了一些高级功能

1.等待可中断：是指当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，改为处理其他事情，可中断特性对处理执行时间非常长的同步块很有帮助

2.公平锁：是指多个线程在等待同一个锁时，必须按照申请锁的时间顺序来依次获得锁；非公平锁则不能保证这一点，在锁被释放时，任何一个等待锁的线程都有机会获得锁。

Synchronized 中的锁是非公平的，ReentrantLock 默认情况下也是非公平的，但可以通过带布尔值的构造函数要求使用公平锁

3.锁绑定多个条件是指一个 ReentrantLock 对象可以同时绑定多个 Condition 对象，而在 synchronized 中，锁对象的 wait()和 notify()或 notifyAll()方法可以实现一个隐含的条件，如果要和多余一个的条件关联的时候，就不得不额外地添加一个锁，而 ReentrantLock 则无须这样做，只需要多次调用 newCondition 方法即可

2.非阻塞同步

3.无同步方案

可重入代码：也叫纯代码，可以在代码执行的任何时刻中断它，转而去执行另外一段代码（包括递归调用它本身）而在控制权返回后，原来的程序不会出现任何错误。所有的可重入代码都是线程安全的，但是并非所有的线程安全的代码都是可重入的。

判断一个代码是否具备可重入性：如果一个方法，它的返回结果是可预测的，只要输入了相同的数据，就都能返回相同的结果，那它就满足可重入性的要求，当然也就是线程安全的

线程本地存储：如果一段代码中所需要的数据必须与其他代码共享，那就看看这些共享数据的代码是否能保证在同一个线程中执行？如果能保障，我们就可以把共享数据的可见范围限制在同一个线程之内，这样，无须同步也能保证线程之间不出现数据争用的问题

7.2 锁优化

适应性自旋、锁消除、锁粗化、轻量级锁和偏向锁

7.2.1 自旋锁与自适应自旋

自旋锁：如果物理机器上有一个以上的处理器，能让两个或以上的线程同时并行执行，我们就可以让后面请求锁的那个线程稍等一下，但不放弃处理器的执行时间，看看持有锁的线程是否很快就会释放锁。为了让线程等待，我们只需让线程执行一个忙循环（自旋），这项技术就是所谓的自旋锁

自适应自旋转：是由前一次在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间。如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自过程，以避免浪费处理器资源。

7.2.2 锁消除

锁消除是指虚拟机即时编辑器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行消除。如果在一段代码中。推上的所有数据都不会逃逸出去从而被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行

7.2.3 锁粗化

如果虚拟机检测到有一串零碎的操作都是对同一对象的加锁，将会把加锁同步的范围扩展（粗化）到整个操作序列的外部

7.2.4 轻量级锁

7.2.5 偏向锁

它的目的是消除无竞争情况下的同步原语，进一步提高程序的运行性能。如果轻量级锁是在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量，那偏向锁就是在无竞争的情况下把这个同步都消除掉，CAS 操作都不做了

如果在接下俩的执行过程中，该锁没有被其他线程获取，则持有偏向锁的线程将永远不需要在进行同步。

22. 逃逸分析

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他方法中，成为方法逃逸。甚至还可能被外部线程访问到，比如赋值给类变量或可以在其他线程中访问的实例变量，称为线程逃逸

如果一个对象不会逃逸到方法或线程之外，也就是别的方法或线程无法通过任何途径访问到这个对象，则可能为这个变量进行一些高效的优化

栈上分配：如果确定一个对象不会逃逸出方法外，那让这个对象在栈上分配内存将会是一个不错的注意，对象所占用的内存空间就可以随栈帧出栈而销毁。如果能使用栈上分配，那大量的对象就随着方法的结束而销毁了，垃圾收集系统的压力将会小很多

同步消除：如果确定一个变量不会逃逸出线程，无法被其他线程访问，那这个变量的读写肯定就不会有竞争，对这个变量实施的同步措施也就可以消除掉

标量替换：标量就是指一个数据无法在分解成更小的数据表示了，int、long 等及 refrence 类型等都不能在进一步分解，它们称为标量。

如果一个数据可以继续分解，就称为聚合量，Java 中的对象就是最典型的聚合量

如果一个对象不会被外部访问，并且这个对象可以被拆散的化，那程序正在执行的时候将可能不创建这个对象，而改为直接创建它的若干个被这个方法使用到的成员变量来代替。

23. 什么是 volatile？

关键字 volatile 是 Java 虚拟机提供的最轻量级的同步机制。当一个变量被定义成 volatile 之后，具备两种特性：

保证此变量对所有线程的可见性。当一条线程修改了这个变量的值，新值对于其他线程是可以立即得知的。而普通变量做不到这一点。

禁止指令重排序优化。普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。

为什么基于 volatile 变量的运算在并发下不一定是安全的？

volatile 变量在各个线程的工作内存，不存在一致性问题（各个线程的工作内存中 volatile 变量，每次使用前都要刷新到主内存）。但是 Java 里面的运算并非原子操作，导致 volatile 变量的运算在并发下一样是不安全的。

为什么使用 volatile？

在某些情况下，volatile 同步机制的性能要优于锁（synchronized 关键字），但是由于虚拟机对锁实行的许多消除和优化，所以并不是很快。

volatile 变量读操作的性能消耗与普通变量几乎没有差别，但是写操作则可能慢一些，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

24. 并发与线程

并发与线程的关系？

并发不一定要依赖多线程，PHP 中有多进程并发。但是 Java 里面的并发是多线程的。

什么是线程？

线程是比进程更轻量级的调度执行单位。线程可以把一个进程的资源分配和执行调度分开，各个线程既可以共享进程资源（内存地址、文件 I/O），又可以独立调度（线程是 CPU 调度的最基本单位）。

实现线程有哪些方式？

使用内核线程实现

使用用户线程实现

使用用户线程+轻量级进程混合实现

Java 线程的实现

操作系统支持怎样的线程模型，在很大程度上就决定了 Java 虚拟机的线程是怎样映射的。

25. Java 线程调度

什么是线程调度？

线程调度是系统为线程分配处理器使用权的过程。

线程调度有哪些方法？

协同式线程调度：实现简单，没有线程同步的问题。但是线程执行时间不可控，容易系统崩溃。

抢占式线程调度：每个线程系统来分配执行时间，不会有线程导致整个进程阻塞的问题。

虽然 Java 线程调度是系统自动完成的，但是我们可以建议系统给某些线程多分配点时间——设置线程优先级。Java 语言有 10 个级别的线程优先级，优先级越高的线程，越容易被系统选择执行。

但是并不能完全依靠线程优先级。因为 Java 的线程是被映射到系统的原生线程上，所以线程调度最终还是由操作系统说了算。如 Windows 中只有 7 种优先级，所以 Java 不得不出现几个优先级相同的情况。同时优先级可能会被系统自行改变。Windows 系统中存在一个“优先级推进器”，当系统发现一个线程执行特别勤奋，可能会越过线程优先级为它分配执行时间。

26. 什么是类加载器，类加载器有哪些?

实现通过类的权限定名获取该类的二进制字节流的代码块叫做类加载器。

主要有一下四种类加载器:

1. 启动类加载器(Bootstrap ClassLoader)用来加载 java 核心类库，无法被 java 程序直接引用。

2. 扩展类加载器(extensions class loader):它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

3. 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

4. 用户自定义类加载器，通过继承 java.lang.ClassLoader 类的方式实现。

27. 对象“已死”的判定算法

由于程序计数器、Java 虚拟机栈、本地方法栈都是线程独享，其占用的内存也是随线程生而生、随线程结束而回收。而 Java 堆和方法区则不同，线程共享，是 GC 的所关注的部分。

在堆中几乎存在着所有对象，GC 之前需要考虑哪些对象还活着不能回收，哪些对象已经死去可以回收。

有两种算法可以判定对象是否存活：

1.）引用计数算法：给对象中添加一个引用计数器，每当一个地方应用了对象，计数器加 1；当引用失效，计数器减 1；当计数器为 0 表示该对象已死、可回收。但是它很难解决两个对象之间相互循环引用的情况。

2.）可达性分析算法：通过一系列称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连（即对象到 GC Roots 不可达），则证明此对象已死、可回收。Java 中可以作为 GC Roots 的对象包括：虚拟机栈中引用的对象、本地方法栈中 Native 方法引用的对象、方法区静态属性引用的对象、方法区常量引用的对象。

在主流的商用程序语言（如我们的 Java）的主流实现中，都是通过可达性分析算法来判定对象是否存活的。