- java基础
  - 面向对象及三大特性
  - 多态的实现
  - 注解
  - public，protect，private修饰符及默认default。
  - static，final
  - String，StringBuilder，StringBuffer，intern方法
  - equels与==
  - 错误和异常
  - 接口和抽象类的区别
  - 序列化
  - 不可变类
  - 引用传递与深浅拷贝
  - 反射
  - BIO/NIO
  - java8新特性
  - 负载均衡算法
- 集合
  - collection
  - java7/8中的HashMap（转set机制），HashTable，TreeMap，LinkedHashMap
  - ArrayList和LinkedList，Vector，Stack
  - HashSet、TreeSet、LinkedHashSet
  - collections
  - Comparable和Comparator接口
- 并发
  - java内存模型
  - happens-before
  - volatile
  - CAS
  - 原子类
  - AQS
  - 线程创建方式
  - java7/8的concurrentHashMap
  - synchronized与lock同步锁
  - 线程池
  - ThreadLocal
  - ArrayBlockingQueue和LinkedBlockingQueue（两种阻塞队列）
  - CopyOnWriteArrayList
  - CountDownLatch和CyclicBarrier和Semaphore（同步辅助类）
  - ConcurrentModificationException异常
- JVM
  - 对象
  - 内存泄漏
  - 垃圾回收
  - 内存分配
  - **内存调试工具**
  - 类加载
  - 参数调优
  - 锁优化
- 消息队列
  - 秒杀系统
- JDBC

java基础

面向对象及三大特性

封装：把客观事物抽象成具体的类，并把对类的数据的操作抽象为方法，通过不同的修饰符对内部数据提供不同级别的保护，封装以后也使得信息和操作对用户透明化。

继承：通过继承，子类可以使用父类的非私有变量和方法，实现代码复用，等级划分等，包括接口和类继承，也是支持多态的一种基础。

多态：是指一个类实例的相同方法在不同情形有不同表现形式。通过同一个上级引用指向不同的子类，使得对同一消息做出不同回应。

多态的实现

一个类中的方法重载是编译时多态（传递参数决定执行那个方法，不同参数的同名方法的签名是不一样的，根据签名可以唯一确认执行什么方法）。使用父类（或接口）引用指向子类（或实现类）；调用静态方法看左边；调用非静态方法（只能访问重写的方法或左边引用对象有而右边引用对象没有的方法，右边有左边没有的非静态方法，直接用左边的引用访问会报错，找不到方法），从右边的实例开始找，没有找其父类，逐个往上，直到Object；访问属性都看左边。所以不同的子类或实现类同一个方法对同一个请求有不同的动作，因为找的是右边子类的方法，这是运行时多态。

多态允许具体访问时实现方法的动态绑定。Java对于动态绑定的实现主要依赖于方法表，通过继承和接口的多态实现有所不同。

- 继承：在执行某个方法时，在方法区中找到该类的方法表，再确认该方法在方法表中的偏移量，找到该方法后如果被重写则直接调用（所以继承关系的类的同一个方法，在其自己的方法表上偏移值是一样的，所以可以这样找，继承关系使用`Invokevirtual` 偏移调用方法），否则认为没有重写父类该方法，这时会按照继承关系搜索父类的方法表中该偏移量对应的方法。
- 接口：Java 允许一个类实现多个接口，从某种意义上来说相当于多继承，这样同一个接口的的方法在不同类方法表中的位置就可能不一样了。所以不能通过偏移量的方法，而是通过搜索完整的方法表。

注解

使用@interface作为修饰符进行申请，只有成员变量，注解就像是标签，用作于描述代码，反编译代码中期继承了Annotation接口。创建时使用关键字表明其信息

```
/**
*@Documented:指明该注解是否可以将注解信息添加在java文档中（有就可以）
 *@Inherited：该注解可以被自动继承（子类继承父类时，继承父类方法上的注解等）
 *@Retention:指明在什么级别显示该注解，定义了注解的生命周期
    RetentionPolicy.SOURCE 注解存在于源代码中，编译时会被抛弃
    RetentionPolicy.CLASS 注解会被编译到class文件中，但是JVM会忽略，类加载时忽略
    RetentionPolicy.RUNTIME JVM会读取注解，同时会保存到class文件中
  *@Target:指明该注解可以注解的程序范围
    ElementType.TYPE 用于类，接口，枚举但不能是注解
    ElementType.FIELD 作用于字段，包含枚举值
    ElementType.METHOD 作用于方法，不包含构造方法
    ElementType.PARAMETER 作用于方法的参数
    ElementType.CONSTRUCTOR 作用于构造方法
    ElementType.LOCAL_VERIABLE 作用于本地变量或者catch语句
    ElementType.ANNOTATION_TYPE 作用于注解
    ElementType.PACKAGE 作用于包

    注解只有属性，若只有一个属性时可命名为value，传参就不需要写属性名
    若有其他属性，且无默认值，就只能创建注解时加上，且用逗号分隔
 */

//使用实例
@Documented//注解可以将注解信息添加在java文档（doc）中
@Inherited//注解可以被继承
@Retention(RetentionPolicy.RUNTIME)//生命周期是整个阶段
@Target({ElementType.TYPE, ElementType.FIELD})//次注解作用于类和字段上
public @interface FieldTypeAnnotation {
/**
 *leip 2016年12月3日
 *TODO
**/
String type() default "ignore";
int age() default 27;
String[] hobby(); //没有指定defalut的，需要在注解的时候显式指明
}
```

public，protect，private修饰符及默认default。

public可以被任何类调用，protect同包下及其子类（可以是不同包的子类）访问，default允许同一个包下的类访问，private只能自己访问。

static，final

static可以用于修饰属性，使的属性属于类，只实例化一次，类及其实例化对象都使用这一个，可以用类名直接调用；还可以修饰方法，方法只能使用类的静态变量和调用其他静态方法，方法属于类，可以用类名直接调用，其次还能修饰内部类，做静态块，静态导包，直接使用静态导入包的方法，还不用加类名。static不会影响属性的访问区域，static不能修饰局部变量。

final修饰的类不能继承，所有方法默认使用final修饰，比如Java中的；用final修饰的方法不能被重写；final修饰的变量需要直接赋值或者在静态块里面赋值，final修饰的变量后面不能改变，final修饰的引用后面地址不能改，但地址对应的内部内容可以改。

String，StringBuilder，StringBuffer，intern方法

String直接用引号创建时会在常量池查看是否已经存在，有的话直接返回，没有的话创建后返回；new String创建时会在堆上创建对象并返回，然后会在常量池中看有没有一样的字符串，没有的话在字符串中创建一个；用方法返回值赋值的，不会再常量池生成字符串。String是用final修饰的类，所以是一旦申请后不可再变，这样做可以节约堆空间，因为多个字符串变量都会指向常量池中同一个字符串；因为字符串会被多个线程共享，那么不能变的话就保证了线程安全；在类加载时描述符都是字符串形式的，不可变就提高了安全性；最后就是字符串的不变性让他很适合做hash映射的键，由于本身不变化，所以hashcode也不会变化，那么可以缓存字符串的hashcode提高效率。String在做+运算连接字符串时，效率是很低的，因为它会把所有的字符都转换为StringBuilder后，依次append，最后在toString。

StringBuffer和StringBuilder都是长度可变的，和string一样，他们都用final修饰了，是不可继承的。其扩容机制是变为原来的两倍+2。StringBuffer是线程安全的，因为底层上所有的方法相对于StringBuilder都使用synchronized修饰了。当然没有进行同步的StringBuilder也就具有更高的效率。

使用+运算拼接直接连接使用引号的字符串或者使用final修饰的字符串引用（赋值不是通过方法返回）时，效率是比较高的（编译时会直接拼接，不会转化为StringBuilder拼接），尤其拼接以后产生的字符串在常量池中已经存在的话，效率是最高的。

调用字符串的intern方法时，1.6及之前，会查看常量池中是否存在，若不存在就把字符串复制到常量池中，并返回常量池中字符串的引用；1.7及以后，会先在常量池中是否存在，存在就返回引用，不存在就去堆里面查看是否存在，存在的话就把堆中的引用返回，并把这个引用存到常量池里面。如果都不存在就把当前字符串复制到常量池并返回常量池里面的引用。

equels与==

==对于基本变量比较的是值是否相等，而引用比较的是地址是否相等，equels不能用于比较基本变量类型变量，引用比较默认和==一样，但一般都会重写来比较引用对象的内容是否相等（比如Integer，String，Double）。

错误和异常

异常是能被管理人员处理的，而错误是系统自带的，比如内存溢出等。异常和错误都有共同的父类Throwable，异常又分运行时异常RuntimeException（像类型转换，数组越界，空指针异常，算数异常等），运行时异常又称非受检异常，这种异常可以不处理和抛出。而像Exception，找不到类、文件，IO异常，SQL异常等有称为受检异常，受检异常需要try、catch块进行处理的，否则编译无法通过。子类重写或者实现方法时抛出的异常不能是父类或者接口中对应方法抛出的异常的等级高，只能抛出其一样的或子类异常。

接口和抽象类的区别

- 接口的方法都是抽象的，抽象类的方法可以是抽象的，也可以是非抽象的；
- 抽象类可以有构造方法，但接口没有（构造方法用于子类继承时使用，但不能实例化）；
- 实现接口必须实现实现所有类，但继承接口时可以不用实现所有抽象方法，只要自身依然是抽象类就行；
- 接口的变量都是使用final修饰的，而抽象类中不是的；
- 接口成员函数都是默认public，抽象类有多种修饰符；
- 接口不能被实例化，抽象类也不可以，但抽象类含有main方法时可以被调用；
- 类是多实现，单继承的，所以多态调用中继承使用了偏移直接在方法表找方法，而接口实现类要遍历方法表找方法，所以继承抽象类的方法调用比实现类实现接口的方法调用快；

希望某些方法具有默认实现，或者是基本功能经常改变，就用抽象类，抽象类是用来捕捉子类的通用特性。协调不同的模块间的方法签名的统一或者涉及多继承的情况都用接口，大部分情况下也都使用的是接口。

序列化

往往我们需要将对象进行序列化，这时对象要实现Serializable接口，并指定一个序列号，序列号是通过输入输出流进行的，将数据在对象与字节数组之间转换（要使用同样的编码格式），对象属性中若有属性使用了transient修饰，则这个属性不可序列化，反序列化后结果为null（但如果实现的是Externalizable接口就能序列化），静态变量也不能序列化，反序列化后可以得到结果是因为其从jvm中获得的，而不是反序列化的结果

自己写的序列化工具：

- 序列化：对象传入对象输出流结果给字节输出流，输出字节数组并返回。
- 反序列化：字节数组传入字节输入流，把输入流结果对对象输入流，得到Object对象并返回。

java集合自定义了序列化工具，其内部真正储存对象的数据结构（比如数组）使用了transient修饰，避免被直接序列化，因为这个数组一般存不满（size一般小于capacity），浪费了空间。集合的序列化是根据size，逐个元素序列化后返回，避免序列化整个数组。

不可变类

类使用final声明；成员全部私有并使用final修饰；不提供setter方法；get方法，返回对象的克隆拷贝

引用传递与深浅拷贝

调用函数传入对象引用时是值传递，此时的值是传入对象引用的拷贝，我们可以在函数中通过引用改变原对象引用指向区域的内容，但在函数中改变引用不影响原引用。

浅拷贝会为拷贝对象新开空间，然后利用被拷贝对象来初始化拷贝对象，对应被拷贝对象的基础变量会在堆中新建一份一样的，但对于对象引用是把对象复制一份后直接给，也就是说被拷贝对象和拷贝对象的属性中的同一个对象引用指向的是同一个区域，clone方法默认为浅拷贝。而深拷贝会把对象引用对应的实际内容在堆中再建一份，并指向，所以深拷贝时，基本数据类型，和对象引用全部不是一样的，实现深拷贝可以使用序列化和反序列化，或者让类内部属性对应的类都实现Cloneable接口。注意存储循环时不要使用深拷贝。

反射

正常使用的类全部是被jvm加载以后的，我们运行的程序在编译器的时候就已经知道需要的类已经被加载了。而反射是在编译时不确定那个类被加载了，而是在运行时才加载，探知自审（可以获得类的方法与属性，并获得属性的类型，对应Type对象；还可以获得生命周期为Runtime和Class的注解），使用编译期并不知道的类就是反射。

通过反射获得类的class对象（Class.forName(全限定名)；类名.class；实例对象.getClass()。）

- 从而可以获得该类的信息，比如访问修饰符，实现了什么接口，父类信息。还可以获得构造方法，打包在哪，导了那些包。

  方法（Method[] 返回本身和父类public方法：getMethods()，返回本身所有方法：getDeclaredMethods()

  ，用getMethods()拿私有方法返回NoSuchMethodException）。

  成员（Field[] 返回本身和父类public变量：getFields()，返回本身所有变量：getDeclaredFields()）。
- 使用class对象的newInstance创建实例。
- 对变量设值或调用对应方法
  - 操作私有变量：获取变量（field）后，field.setAccessible(**true**)获取私有变量的访问权，使用field.set(testClass, 修改成的值)修改变量的值。
  - 操作私有方法：拿到方法后用privateMethod.setAccessible(**true**)获得私有方法访问权，然后用privateMethod.invoke(class对象, 参数1, 参数2...)执行方法。
- 修改私有常量（final修饰）

  jvm优化：对于基本类型和final修饰的类的引用，如果申请时用final修饰，在使用时，jvm会把常量名直接替换为具体对应的值，而不会使用常量名。所以对于这部分常量，无法通过反射去修改，除非改源码。

  但如果申请时没有直接赋值的话，比如常量的值在构造函数中赋值的（用三目运算符赋值等，不直接赋值，编译时无法确定常量的值，必须在运行时才确定就可以，这样就无法优化了），那就能修改常量的值了。

BIO/NIO

BIO：传统IO在读写时会产生阻塞，因为需要等待传输方将数据在缓冲区中准备好,首先创建socket绑定端口以后**死循环等待**连接请求到达，而对于socket接收连接和读写数据都是要阻塞的（阻塞等待可连接，网卡可写，缓冲区可读，等待过程不占用cpu资源，真正操作数据时（比如从网卡把数据读到内存）除了AIO都需要阻塞），所以使用线程池将每个到达的连接从线程池中分配线程进行执行，这样阻塞时可以有其他线程执行，每个线程对于请求，在当前线程被中断或者socket关闭前死循环读取数据并处理。这样做在线程少时是不错的选择，但线程多时，要考虑线程池的设计，不然线程的切换就需要花费大量的功夫。

NIO概念：而NIO是多路复用的，在内核设立了专门的线程去轮询访问所有的socket，而不需要具体操作的线程阻塞等待，没准备好的时候，线程仍然可以进行其他的工作，其在Selector上注册标志位到注册集合表示其对什么事件感兴趣（注册事件处理器），当轮询线程发现准备好时后会在Selector上修改对应的标志位并放到就绪集合，使得工作线程知道数据已经准备好，可以开始操作（Reactor：有事件分发器交给事先约定的事件处理函数或回调函数；Proactor：异步的发起读写请求。Selector.select）。

NIO使用：java使用NIO读写时一般建立文件输入出流，从流对象中getChannel获得通过（FileChannel）实例对象，再建立缓冲区实例对象（如BufferByte.allocate），然后调用channel的读写函数，传入缓冲区实例，进行文件的读写，buffer的flip方法把写模式变成读模式（limit变为position，position归零），读到写用clear，一般用于文件复制,put、get函数写读缓冲区。包含三个重要参数：capacity（包含元素的数量），limit（上限，第一个不能操作的元素的索引），position（下一个要操作的元素的索引），mark用于备份position位置（mark()方法让mark=position，reset()方法反过来）

三个核心组件：NIO的工具包提出了基于**Selector**（选择器）、**Buffer**（缓冲区）、**Channel**（通道用作读写Buffer）的新模式；Selector（选择器）、可选择的Channel（通道）和SelectionKey（选择键）配合起来使用，可以实现并发的非阻塞型I/O能力。

java8新特性

- **Lambda**表达式和Functional接口：Lambda表达式可以允许把函数作为参数传参可以用（类型 参数）->{方法体}替代以前的匿名内部类，方法体中可以调用参数与成员变量和局部变量（隐式转换为final），一般使用参数数组结合forEach或sort等方法时使用，传参也可以不指定类型，参数类型和返回值类型是编译器推测出来的。Functional接口使用@FunctionalInterface注解修饰接口，使得这样的接口可以被隐式转换为lambda表达式。
- **default**修饰符修饰接口中的方法，可以为接口中的方法写默认实现，实现类可以自己决定是否覆盖这个方法
- **方法引用**：直接引用已有Java类或对象（实例）的方法（Class名::method）或构造器（Class名::new）。与lambda联合使用
- **重复注解**
- **Stream**，函数式编程风格，集合的批处理和流式操作，搭配lambda表达式进行去重（distinct）、分类（Collectors.groupingBy）、过滤（filter），加和，最大最小等函数操作。
- **新的Date-Time API**：Clock，LocaleDate与LocalTime
- jdk1.8中JVM的新特性：**抛弃了永久代**，数据转移到堆中，**引入元空间**，元空间作用与永久代类似，都是方法区的实现，元空间不在虚拟机中，而是在本地内存下，字符串不存在元空间，存在字符串常量池里面。

负载均衡算法

轮询分发，每个服务器接收一个请求的来，一台一个依次；按权重，权重越大接收到请求的概率就越高，权重可以按照服务器的性能来划分；哈希分发，哈希分发又分ip哈希和url哈希，使得客户端和服务器对应的服务器总是一致，这样还能在服务器保存客户端的session；按响应时间划分，优先分配给响应时间短的；随机发；发给任务最少的机器。

集合

collection

是set，list的父接口，包含了基本的添加，删除，迭代，判空，判断存在，清空等基本方法。集合中的数据在迭代时，如果被修改可能导致fast-fail，比如并发情况下，一个线程迭代访问，一个线程删除元素会导致这个问题，这个时候建议使用并发的集合；又或者迭代的使用集合的remove在单线程下删除元素也会导致，建议使用迭代器的删除元素的方法，可以避免。

list和set都是实现了collection接口，list可以存null，set可以存一个null；TreeMap和TreeSet在第二次及以上加入元素时会进行比较，所以不能存null键（第一次放null可以放进去，但比较的时候一旦有null就会出错）。

java7/8中的HashMap（转set机制），HashTable，TreeMap，LinkedHashMap

都是实现了Map接口。

HashMap在1.7和1.8中有些许改动，1.7中HashMap是一个长度为2的n次方的数组，这样的好处是可以计算hash时使用位运算代替模运算，并且rehash时增加两倍的特性使得同一个节点下的数据只会分散到两个节点下，每个数组节点都是一个链表，所以采用的是链地址法解决哈希冲突，put操作时先计算hash对应的下标，找到具体的数组项，接着会逐个遍历链表项，有相等的就直接覆盖，没有相等的键或者数组项为空会新插入，值得注意的是插入时会先判断当前负担因子是否超过限额，超过会进行rehash，1.7中采取先扩容后插入的策略，并且为了效率1.7中采用的是头插法，这导致多线程并发访问下，同时触发rehash时可能出现链表结构死循环现象，导致线程死在无休止的get中。get操作也是根据hashcode找到对应下标后遍历。在1.8中发生了一些变化，使用红黑树的结构在一个节点存储数据项大于8时代替链表，提升遍历效率，用尾插代替了头插，但仍然不建议在并发情况下使用hashmap，还有就是数据要先插入再触发rehash。

hashmap有扰动函数，将除null以外的键与键右移16位的结果异或，再将得到的结果与低位掩码进行与操作得到对应的数组下标。

HashTable就是HashMap中所有方法使用synchronized修饰后的类，但在多线程下仍然可能出现问题，比如边删边查导致越界现象等。但HashMap可以存一个null键和很多个null值，放在table[0]里面，put进来处理，HashTable不能存null键和值，因为这是并发集合，如果get时，返回了null，不能确定是键无映射，还是储值为null，而hashmap是单线程的集合，有contains方法判断键是否存在。

TreeMap使用红黑树作为数据结构，以键为排序的依据，按键的自然顺序，或者使用比较器。键与值组成一个节点。

LinkedHashMap中每个节点除了有键和值以为，还存在前后指针，使得每个节点插入时，不仅会根据hash值散列存放起来，还会有前后指针维护成一个双向链表，保证访问顺序

可以返回成set集合，键使用keySet，值使用values，键值对一起Set<Map.Entry<X,X>> set=map.entrySet();

只有HashMap中允许键为空

ArrayList和LinkedList，Vector，Stack

ArrayList和Vector底层都是用的数组，包含了对数组基本的添加，删除，获取，清空等操作，数组的扩容每次都扩容为原来的1.5倍，但多线程往arraylist中添加数据时，在临界要扩容时是可能出现数组越界情况的（两个线程交替进入add，先进的判断不会越界后阻塞，后来的也同样可以添加，导致连续添加两个，size+2，在数组边界处越界），LinkedList和Vector底层都是双向链表，理论上无大小限制，也包含对应的各种操作，链表删除新增比数组快，但数组的读取比较快，Vector和Stack是用synchronized修饰了的，是同步的集合，Stack继承了Vector。

HashSet、TreeSet、LinkedHashSet

他们都实现了Set接口，set中不允许出现重复元素的，重复元素指的是对象使用equals比较相等。HashSet是无序的集合，他是基于HashMap实现的，每个元素都是当做键存入了HashMap，然后值使用一个默认值；TreeSet是使用红黑树实现的，键就是存放的值，是有序的集合，在传入值没有自然顺序时，需要实现比较器。LinkedHashSet继承于HashSet，底层使用LinkedHashMap的键存放对应的值，保证了有序性。

collections

集合工具类，维护了很多操作集合的工具，例如反转，排序（转化为Arrays里面的排序方法，数据多的时候使用归并，少的时候使用插排，中间层次使用快排，47,286），二分检索，查找，同步控制（ synchronizedCollection方法可把传入的集合变成同步集合，原理是装饰原来的方法，每一个都用synchronized包起来），还可设置不可变集合。

Comparable和Comparator接口

实现Comparable接口的compareTo方法的对象，在TreeSet和TreeMap中保存时或使用工具类排序时会使用这个方法进行比较，实现方法时用this与传入对象进行比较。也可以自定义类去实现Comparator接口的compare方法，直接传入两个对象进行比较，在新建有序容器时把一个比较器实例对象传入即可。这是使用了策略模式

并发

java内存模型

内存，缓存，寄存器被抽象分为主存和工作内存，每个线程都有工作内存，会从主存中拷贝共享的变量到工作主存中，进行写时，先写工作内存再写会到主存，如此在多线程下就可能导致不一致的现象，物理上是没有分区的，工作内存是逻辑的分区，而每个线程对应位一个内核线程，在内核调度器下使用cpu。

java内存模型需要承诺原子性，可见性，有序性

- 原子性：如对一个32位静态变量赋值，这样的具有原子性的操作，过程中是不允许打断的，synchronized和lock都可保证原子性，CAS操作也可以。
- 指令重排序（没有前后数据依赖时）（导致违背可见性与有序性）（使用volatile）
  - 编译器优化：编译器在编译时，不改变单线程程序语义的前提下，是可以重新安排语句执行顺序的。所以即使写的代码看起来不会有问题，但由于重排导致了问题。
  - 指令并行的重排：现在操作在汇编中对应取址等一系列操作，由于前后数据存在关联导致流水线某一步等待使得流水线需要等待好几个周期才能恢复，所以也有重排来提高效率
  - 内存系统重排：由于缓存和缓冲区的存在，导致load和store操作乱序执行。实际的读取可能出现在修改之前
- 可见性：一个线程在自己的工作内存修改某个共享变量后，其它线程要马上得知。
- 有序性：实际运行与顺序的执行代码结果一致。

happens-before

- 程序顺序原则，保证语句执行串行化，按照代码顺序执行
- 解锁发生在另一个加锁前
- volatile变量写执行完后才会执行后续的读，写完强制写会主存，读从主存直接读
- 线程的start()发生于线程内部代码之前，执行代码先于线程终止
- 传递性，A先于B，B先于C，则A必定先于C
- 线程中断（interrupt()）后可以理解被检测到（interrupted）
- 对象的构造方法先于回收。

volatile

java内存模型中每个线程都有一个高速缓冲区保存主存中某些变量的副本，而内存中的共享变量都需要解决缓存一致性问题。最基础的方法是直接对总线加锁，阻塞其他cpu对其他部件的访问，但这样效率很低。所以后来使用缓存一致性协议来确保效率和一致性，volatile就是这样的，它具备两种特性，可见性和禁止重排序。可见性就是确保数据一致性的体系，它规定在线程对数据进行写时，**立即**从副本写回内存，其他线程工作线程中这个变量会被置为无效，强制读取内存。禁止重排序是指被volatile修饰的变量不会因为重排序而乱序，会确保此变量之前的操作不会出现到后面，后面的操作不会出现在前面，对应变量进行操作时前面的修改都已完成并对后续操作可见。

而这些体现到汇编是编译时多出了内存屏障（lock前缀指令）。屏障分四种。执行遇到内存屏障会确保前面的操作已经完成才会执行后面的，后面的不会出现到屏障前面，避免屏障前后的执行顺序出现乱序；强制将对缓存的修改操作立即写入内存并导致cpu中其他的缓冲行无效。

CAS

比较替换原则，每次根据预期值去比较内存对应变量的值，相等就更新为设定数据，不相等就返回false。比较加替换利用了操作系统的原语操作，也会有加锁存在，但是利用底层硬件实现的锁机制，比synchronized的性能更好，但cas存在ABA问题（后来就有引入了时间戳引用原子类来解决，它比较不仅会比较值，还会比较时间戳），且大部分更新失败的情况下会直旋重复尝试，竞争较大时效率不高，并只能控制单个变量同步

原子类

大部分操作，比如自增等，都是基于CAS来实现的。

AQS

AQS同步器内部维持一个双向队列保存等待线程以及一个头结点一个尾节点，头结点是当前获得同步状态的线程，AQS利用这个队列确保每次只有一个线程可以获得同步状态，几乎所有juc包下的所有锁机制都是基于同步器来实现的，AQS中的包含独占锁与共享锁两种锁，获取锁的时候都会首先判断是否有线程持有同步状态，没有就直接尝试获取，获取成功就直接进入同步状态，否则就尝试把线程放到同步队列尾部，这一步入队会使用CAS自旋来确保入队，入队后是一个自旋锁一直尝试获得同步状态，每次先查看自己前面的节点是不是头节点，是的话就会去尝试获得头部节点，不是或者获取失败或尝试去跳过前面被取消的节点，然后阻塞自己。然后线程接下来能被唤醒或者中断，又会去看前驱节点是不是头结点，并尝试获得同步状态。当头节点释放同步状态时会去唤醒后驱节点，后驱节点拿到同步状态后，排他锁就正常执行，共享锁会主动去传播共享状态。AQS还分公平和不公平锁两种，不公平锁进来就会直接尝试CAS去拿同步状态，在队列里面也不会老实排队，而是只要同步状态释放就直接CAS去抢。

线程创建方式

- 自定义类继承Thread，重写run，创建实例并调用start方法。
- 自定义类实现Runnable接口，实现run方法，创建Thread实例，并传入自定义类，调用Thread实例的start
- 自定义类实现Callable接口，并实现call方法（可以有返回值），新建自定义类的实例对象传给FutureTask创建实现对象task，在把task传给Thread创建实例，调用start方法开始执行，执行是异步的，并且这种创建方式可以有返回值，返回值用task.get()获得。
  - FutureTask继承了Runnable接口和Future接口
    - 用volatile定义了一个整数，对应0-6代表了七种状态，对应创建，任务执行，中断等待，已中断，已取消，成功，异常。取消时任务没被执行就直接变已取消，在执行就变为中断等待，执行完成后变成已中断，正常执行任务根据结果变为成功和异常状态
    - FutureTask既然实现了Runnable接口就有run方法，而run方法就执行传入Callable接口实现类的call方法（若状态为创建才执行任务，不然表示任务已经执行了，**接着用unsafe类把任务执行线程保存在runner字段，保存失败会直接返回，**执行结果用set保存结果，或者setException保存产生的异常，返回唤醒等待线程（Unsafe.unpark）并返回结果）
    - 取消任务时，如果任务不是创建状态，那么表示已经执行过了，直接返回false；若是执行状态，以中断等待作为中间态，执行完后变成已中断；若不是执行状态，直接变为取消；最后唤醒等待任务结果的线程。
    - get用来获得执行结果，有结果就返回，没结果就等待，等待是一个死循环来判断任务的状态，也可以传入时间参数限时等待
      - 等待状态：
        - 线程若被中断，则移除这个等待节点，抛出中断异常
        - 任务完成，直接返回
        - 若未创建等待节点，就新创建节点，让当前线程加入等待队列
        - 通过`LockSupport.park`设置等待时间进行阻塞（用Unsafe类提供的硬件级别原子操作park和unpark）
- Future接口定于了取消任务，查看任务是否取消，查看任务是否执行完成，阻塞获得执行结果，限时获得执行结果方法

java7/8的concurrentHashMap

1.7中的concurrentHashMap使用了多段锁的机制，定义了16个segment，这种结构继承了可重入锁，所以每个segment是一个单独的容器，也是一把锁。初始化时会为16个segment的第一个进行初始化，并用CAS操作放入，初始化第一个的作用在于后面的segment初始化都直接在第一个里面拿值进行初始化。在进行put操作时通过hash计算节点应该放到哪个segment里面，接着查看这个segment是否为空，为空要进行初始化，初始化时线程需要用CAS操作修改某个标志的值，修改成功再进行初始化，不成功就放弃初始化，这保证只有一个线程初始化了segment，初始化时是利用第一个segment的值来初始化的。初始化成功后再进入segment之前要操作获得这个segment的锁，获取失败会进行自旋尝试获得，并且在自旋时还会尝试去初始化节点，拿到锁后在内部再次散列获得数组元素节点，遍历链表，若键存在就更新对应的值，若不存在建立节点后使用头插法插入节点。插入时使用CAS操作保证与get之间的并发性。最后有锁的释放。删除操作同理也要使用CAS保证同步性，插入数据后是可能触发rehash的，rehash会让当前segment下数组长度扩容两倍后依次移动放入，建立的临时数组要用volatile修饰，保证get时是可见的。1.8中抛弃了segment结构，只使用了一个数组，用链地址法解决hash冲突，但单个节点上数据过多会使得链表变成红黑树，这样访问效率就从O(n)变成O(logn)，put时首先检查对应数组节点是不是空的，空的话尝试用CAS插入，然后判断是否当前节点的hash值为MOVED，这表示有线程再进行rehash，那么当前线程要去帮忙rehash，然后前面两种都不满足，说明尝试访问非空的数组节点，这是会首先以链表头作为锁对象利用synchronized关键字建立同步代码块，然后对节点进行访问遍历，有相等的键覆盖，没有就尾插到链表尾，然后计算节点数据个数看要不要转换为红黑树，计算负担因子看要不要扩容。前面说MOVED表示当前有线程再扩容，这是个特殊节点对应的hash值常量，并且此时会发现指向新数组的引用不为空，线程帮忙时会被分配任务，执行对于数组节点上的链表搬运，线程首先检查当前节点的头节点是不是特殊节点，是的话表示这个节点已经被其他节点搬运走了，不是的话，尝试获取头结点的锁，获得成功就锁起来，然后开始搬运链表中的节点，首先把节点根据hash值分成两条链表，再分别放到新数组的两个节点下，最后把久的节点的头节点设置为特殊节点，表示搬运完毕。接着搬运其他，直到结束rehash。

synchronized与lock同步锁

synchronized用于修饰方法使方法成为同步方法，或者建立同步代码块，他们使用的锁，都是对象头中的锁，修饰方法时，如果不是静态方法，那么使用的是this实例的锁，静态使用类的class对象作为锁，同步代码块需要传入一个对象用作锁，synchronized在发生异常时可以jvm实现锁的释放，多种线程使用synchronized用作同步时，可使用wait，notify，notifyAll，yield，sleep，interrupt进行通信，（过期stop，suspend），等待状态的线程无法被中断，除io阻塞的阻塞线程可以被中断，且线程无法得知自己是否拿到了锁。

Lock接口定义方法lock去获取锁，拿不到就进入队列排队等；tryLock去获取锁，拿不到锁就返回false，不会排队等候；而lockInterruptibly获取锁的话，获取失败会进入阻塞队列等待，但等待期间可以被中断，synchronized在等待期间不能中断退出。lock的实现类ReentrantLock实现了锁的可重入，是lock直接的实现类。ReentrantReadWriteLock锁是ReadWriteLock接口（继承Lock接口）的实现类，可取出读写两种锁来进行同步。

两种锁都是可重入的，lock是可中断的，synchronized不是，都默认是非公平锁，但lock可通过传参变成公平锁，lock同步锁可使用读锁，正常下两种锁效率差距不大，但在读操作频繁的地方，具备读锁的Lock接口明显更高效。

线程池

ThreadPoolExecutor，线程池的重要参数和属性包括核心线程池大小，最大限制创建线程数量，任务缓存队列，空闲时间。如果当前线程数量小于核心线程池的数量限制，那么不管当前线程池中的线程是否空闲，每来一个任务就创建一个线程去执行这个任务；如果线程数量大于等于核心线程池大小，每来一个新的任务就放入任务缓存队列中，线程池中有线程空闲时就会去拿任务执行；如果任务缓存队列满了，线程数量大于等于核心线程池数量但小于最大线程池创建数量就会新建线程去执行任务；如果任务缓存队列满了，线程数量也等于了最大线程创建数量，就会采取任务拒绝策略；拒绝策略有四种：直接丢弃、丢弃后返回异常、丢弃队列首部，新任务再次入队、由尝试放入任务的线程来执行；当线程池数量大于等于核心池数量时，会回收空闲时间大于设定空闲值的线程，直到线程数量小于核心线程池数量；一般线程池设计数量等于cpu数量加1，若进程io，会阻塞线程，可以考虑把线程数量提高。

工具类Executors下有四种方法创建线程池，这是jdk提供的工具类，但不允许使用，四种方法为

- 均采用无界队列来保存任务，可能由于任务堆积导致OOM

    - newSingleThreadExecutor：核心池与线程池最大容量都为1的线程池（该线程不会被回收），并使用无界的阻塞队列，所以就一个线程去执行任务队列的任务。用于任务一个一个执行的情况
    - newFixedThreadPool：核心池大小与线程池最大容量相等，线程不会被回收，任务队列是无界队列，固定了线程数量，任务不会丢失，适用于长任务
- 由于都不限制线程创建上限，均可能由于线程创建过多导致OOM
    - newCachedThreadPool：核心池大小为0，最大线程容量不限，线程空闲回收时间60秒，每个任务都会新建一个线程，空闲线程短时间内会被回收，适用多且短的异步任务。
    - NewScheduledThreadPool：核心池的大小为传入参数，最大线程容量不限，使用一个按超时时间决定顺序的队列，可定时执行
- newSingleThreadScheduledExecutor：核心池与线程池最大容量都为1的线程池（该线程不会被回收），使用一个按超时时间决定顺序的队列，可定时执行

线程池一般不会使用工具类Executors去创建，而是直接调用`ThreadPoolExecutor`的构造函数来自己创建线程池，创建的时候自己定义参数创建。

submit和execute的区别：

- **execute提交的方式只能提交一个Runnable的对象，且该方法的返回值是void**，也即是提交后如果线程运行后，和主线程就脱离了关系了，当然可以设置一些变量来获取到线程的运行结果。并且当线程的执行过程中抛出了异常通常来说主线程也无法获取到异常的信息的，只有通过ThreadFactory主动设置线程的异常处理类才能感知到提交的线程中的异常信息。
- submit可以提交实现了Callable接口的对象，用get可以返回结果，同时也可以提交一个Runnable的对象。

ThreadLocal

每个线程本地维护一个map，这个map是ThreadLocal的内部类，每个线程自己都有一个独立的这种类型的map，而map中以ThreadLocal为键，要对应存储的值为值建立联系，调用ThreadLocal的get方法，其实是在当前线程中取出这个map，在以this（访问的ThreadLocal实例对象）为键取出，所以实际使用中，需要为多个线程设计多少个想要独享的变量，就应该建立多少个ThreadLocal实例对象。ThreadLocal本身与key之间是弱引用，当ThreadLocal被回收后，就导致key对应的value永远无法访问，造成内存泄漏。

ArrayBlockingQueue和LinkedBlockingQueue（两种阻塞队列）

底层一个使用数组，一个使用链表，所以数组队列有界，链表队列理论无界，两者都是一种消费生产者模型，但数组使用一把锁，生产消费都用这把，而链表有两把锁，一个锁头一个锁尾，一个用于同步生产，一个用于同步消费

CopyOnWriteArrayList

在遍历时会直接访问集合，但写时会copy一份副本，修改完以后再将集合引用指向这个副本，且copy副本时会加lock锁，不用担心会在并发情况下copy多份，修改完后将集合引用指向这个副本后，之前开始遍历的线程依然遍历旧的集合。

CountDownLatch和CyclicBarrier和Semaphore（同步辅助类）

CountDownLatch使用一个同步队列加一个计数器count=n（创建实例时传参传入），每来一个线程就让他们进等待队列，等着获取共享锁（同步状态已经被持有），并让count减一，当count归零时，释放同步状态让这n个线程全部获得共享锁一起运行。线程调用其await方法会将自身阻塞，调用countDown让count减1。

而CyclicBarrier也设count=n（创建实例时传参传入）为计数，每来一个就让它阻塞（调用await方法，await可传入参数指定等待时间，时间过了栅栏没开就抛出异常并继续执行），计数减一，当出现了要让计数归零的线程出现时，不阻塞，让它唤醒所有线程，使得多个线程一起运行。他们的不同点在于，前者使用了共享锁，等待的是一个事件的到达，并且不可重置，后者使用阻塞，是等待线程的到达，可重置。且可传入一个Runnable的实现类，在栅栏打开时执行。

Semaphore是信号量机制，它可以掌握n个资源，让多个线程因为这个资源进行PV操作。P操作是acquire()（拿不到就阻塞），V操作是release()，并且都可以传入参数指定获得和释放的资源数量。还可以使用tryAcquire(num，time),指定获得多少资源，拿不到等待多少时间，不传time默认直接返回。

ConcurrentModificationException异常

对同步类进行迭代时修改会产生这个错误，建议在使用iterator迭代的时候使用synchronized或者Lock进行同步；使用并发容器CopyOnWriteArrayList代替ArrayList和Vector。

JVM

jvm在执行java程序时有若干数据区：**方法区（类信息，常量，静态变量，可以不GC），虚拟机栈（存java方法的数据），本地方法栈（存native方法的数据），堆（存对象的地方，GC的主要位置），程序计数器（类似PC）**

对象

- 对象创建：
  - 检测对象是否被加载，解析，初始化过了，如果未加载需要进行加载
  - 为对象分配内存（要考虑并发（原子操作或私有空间）），连续内存指针碰撞，不连续空闲列表（实际）
  - 初始化：分配完后，内存初始值全部为0；然后进行初始化将对象信息赋值
- 对象结构
  - 对象头：有两部分，运行时的数据，像哈希码，GC分代年龄，锁状态标志等；对象的类型指针，指向它的类元数据，标识对象是那个类的实例。
  - 实例数据：存放有效信息的实例数据。
- 访问方式：1.句柄：句柄池两个指针（指向对象实例数据和方法区中对象类型数据）;

  2.直接指针：指针指向堆区一个地方（实例数据+指向方法区中对象类型数据的指针）。

内存泄漏

当栈申请空间，堆满，方法区满都可以产生OOM错误，这时除了考虑空间不足，也需要考虑是不是由于产生了内存泄漏，如果长生命周期的对象持有短生命周期的引用，就很可能会出现内存泄露，比如只用于某个方法的局部变量声明在类属性中，这样局部变量必须要等到类回收时才会回收，所以我们应该尽可能的减少对象的作用域；其次就是容器容易导致内存泄漏，在容器使用完毕后置空比较合理；静态变量的生命周期十分的长，也是可能导致内存泄漏的原因之一，尤其是静态集合；还有就是单例模式生命周期也是十分的长；内部类整个生命周期都会持有外部类的对象，所以内部类的引用没有处理好导致外部类无法回收，可能导致内存泄漏；同理外部模块的引用也是，处理结束后不删除这个引用就导致模块一直持有这个引用；逃逸问题；ThreadLocal（弱引用）被回收，对应线程的map中的键值对无法访问。

栈溢出(java.lang.StackOverflowError)。

- **减少GC次数的方法**

用StringBuilder代替String，避免常量池在字符串变化过程中产生大量字符串常量；创建对象时避免在短时间内创建大量对象，用时创建；不用的对象，赋值null，方便回收；尽量少的创建静态变量，尤其是静态的集合；监听器不容易被回收；数据库等连接要记得关闭并小心监视器这样不好回收的对象；避免逃逸，比如尽可能的使用基本标量，就不使用封装的基本类；内部类的引用小心处理，否则导致外部类无法回收，同理外部模块的引用。

垃圾回收

- **死亡判断**：**程序计数器**（记录引用），**可达性分析**（从GC root开始能到的对象）
- GC root对象：虚拟机栈和本地方法栈引用的对象，方法区中的静态属性引用对象和常量对象
- 强引用：垃圾收集器永远都不会回收的对象，一般用于程序代码中普遍存在的引用。

- 软引用：描述一些还有用但非必需的对象，发生内存溢出异常之前会把这些对象列入回收范围进行回收
- 弱引用：描述非必需对象，无论是否内存紧张，都会在下一次GC时被回收虚引用
- 虚引用：没有强度的等级，完全不会对生存时间构成影响。甚至无法用来取得一个对象实例，设立虚引用的目的是用来在对象被回收时接收一个系统通知
- **finalize方法**（每个对象的finalize方法只能执行一次，该方法执行代价大，一般不使用。）
  - 第一次GC：如果覆盖finalize方法并没执行过会进行二次回收（F-Queue队列，低优先级执行）
  - 第二次GC：若放入F-Queue队列的对象二次GC的时候还是不可达会被回收，可达被拯救
- **GC触发**
  - Minor GC：新生代回收，eden去空间不足时触发
  - Full GC：当老年代和方法区空间不足时；Minor GC用老年代当做担保时老年代空间不足；新生代中创建大对象直接扔到老年代，老年代存不下时；CMS回收时，由于是并行回收，预留空间不足，放入到了老年代中，老年代空间也不足；主动调用System.gc时
- **垃圾收集算法**
  - 标记-清除：标记回收对象后逐个回收，标记和清除效率都低，且产生很大碎片
  - 复制算法：1:1。老年代做空间担保
  - 标记-整理：标记后，将对象往前移动，效率低，但回收到连续空间
  - 分代收集算法：分老年代和新生代，并进行实际物理分块。结合复制8:1:1，老年代担保
- **垃圾收集器**
  - GC停顿：在进行可达性分析时会进行**GC停顿（停掉所有线程）**确保一致性。不需要遍历整个堆来寻找对象引用再判断（OopMap的数据结构使虚拟机直接得知什么地方存放着对象引用）。
  - 安全点：实际中，许多指令都会导致引用发送变化，每次变化都设置一个OopMap不合理，于是设置安全点，每个安全点出记录引用变化（所以在两个安全点之间记录的引用情况不准确），安全点一般设置在方法调用，循环跳转，异常跳转，进行GC时要保证所有线程到了安全点，确保机制
    - 抢占式中断：停掉所有线程，再唤醒没到安全点的线程跑到安全点停下，用的比较少
    - 主动式中断：安全点设flag，线程过路的时候会主动询问是否需要停下（要gc就设置为真）
  - 安全区域：安全点无法确保阻塞和挂起状态的这些线程，他们不能响应中断，安全区域是一段代码区，整个区域内引用不发生变化。线程进入后标记自己安全，GC时不管他们，出去询问GC是否完成。
  - **垃圾收集器**
    - Serial收集器：复制算法，回收时暂停所有线程。
    - parNew收集器：多线程版本的Serial，cpu数量有影响。
    - Parallel Scavenge收集器：多线程，复制算法，不关注收集效率，设置每次垃圾回收的量来控制吞吐量和GC的时间。若每次回收过少会导致吞吐量降低和空间不足，吞吐量=runtime/(runtime+GCtime)
    - Serial Old收集器：标记-整理算法，单线程，（作用：java5及之前搭配Parallel Scavenge；作为CMS后备）
    - Parallel Old收集器：标记-整理算法，Parallel Scavenge的老年代版本，因为搭配Serial Old拖累性能，java6产生，搭配Parallel Scavenge用于吞吐量和cpu资源敏感（防止cpu使用过高）的地方。
    - **CMS收集器**：标记-清除算法，分四步，两步执行时间较长的步骤运行工作线程一起执行，但只能搭配Serial收集器或者parNew收集器。
      - 初始标记：只标记GC Roots直接关联到的对象，这个过程要使得所有工作线程停顿
      - 并发标记：进行GC Roots Tracing，允许工作线程一起运许
      - 重新标记：工作线程的运行会导致标记发生变动，这一步用于修正这些变动的标记。
      - 并发清除：开始垃圾回收，可以与工作线程一起执行
      - CMS的缺点：
        - 并发标记和并发清除的时候占用了一部分线程从而导致应用程序变慢，cpu数量有影响。

        - 浮动垃圾：并发清除阶段与用户程序一起执行，所以新产生的垃圾在本次无法回收，所以每次需要预留空间给用户线程产生新对象，预留过多使GC频率变高，预留过少可能导致剩余内存不够。此时导致虚拟机启动后备预案使用Serial Old收集器来进行垃圾收集，单线程，停止所有进程，极大拖低效率。
        - 使用标记-清除算法导致可用空间零碎分布，若出现大的对象无法分配内存时，导致提前触发Full GC，CMS收集器在进行Full GC会进行碎片整理。
    - **G1收集器**
        - 特点：
            - 并发与并行：G1利用硬件优势缩短GC停顿的时间，且部分步骤可以与工作进程并发
            - 分代收集：分代概念在G1中仍有保留，独立管理整个GC堆。
            - 空间整合：使用标记-整理算法，回收时不会产生可用空间零碎分布的现象。
            - 可预测的停顿：与CMS一样，都追求降低停顿，但其还能预测停顿的时间模型，预测停顿的时间，
        - G1把java堆分成多个区域，且为区域设置优先级，优先级高的区域先回收，每个区域大小可以配置具体大小，区域有四种，新生代eden，新生代survivor，老年代，大对象区域。
        - 区域之间的对象可能相互引用，为了保持区域独立性又不需要扫描整个堆，每个区域都有自己的Remembered Set，记录引用的其他区域的对象，可达性分析就检查本区域和区域的Remembered Set
        - G1的步骤分为四步
            - 初始标记：类似CMS，标记GC Roots能直接关联到的对象，需要进行GC停顿，然后标记区域使得下一次并发时，用户进程找到正确的Region进行新对象创建。
            - 并发标记：类似CMS，进行可达性分析，寻找要回收的对象，耗时较长，可并发执行用户进程
            - 最终标记：修正并发标记期间，用户进程并发执行引起的引用关系变化，**变化消息记录在线程Remembered Set Log中**，需要把数据合并到Remembered Set中，这阶段可停顿，也可并行执行。
            - 筛选回收：根据优先级来逐个进行区域的回收。
    - 若对象分配过快来不及回收会触发full gc，使用Serial收集器作为回收策略

内存分配

- 对象优先在Eden区分配内存，Eden区空间不足进行GC，GC时若survivor存不下了，则放入老年代（空间担保）
- 大对象（设定参数值）直接在老年代中分配，避免Eden区内存耗费过快，提前GC
- 长期存放在新生代的对象会放入老年代，每个对象维护年龄计数器，每经历一次GC就加1
    - 若年龄大于设置的阈值就放入老年代
    - 若同样年龄的对象占用内存达总内存一半，大于等于这个年龄的对象放入老年代
- GC时空间担保：保证老年代中**最大可用连续空间**大于新生代所有对象空间大小，则可担保进行GC，若不够，jvm允许时会尝试进行冒险（**最大可用连续空间**大于历届晋升老年代对象平均大小）

内存调试工具

- **jps**: 列出正在虚拟机运行的虚拟机进程，并显示虚拟机执行主类的名称，以及这些进程的本地虚拟机的唯一ID。
- **jstat** : 监视虚拟机各种运行状态信息的命令。可以显示本地或远程虚拟机进程中**类装载,垃圾收集，JIT编译,内存**等数据。
- **jinof**: 实时查看和调整虚拟机的各项参数。
- **jmap**（堆）: 生成堆转存储快照，查询fianlize执行队列、java堆和永生代详细信息，如空间使用率，当前用的是那种收集器。
- **Jhat**: 和jmap搭配使用，来分析jmap生成的堆转存储快照。内置一个微型的HTTP/HTML服务器，生成dump文件的分析结果后，可以通过浏览器查看
- **jstack**（线程）:用于生成当前时刻线程快照.线程快照是当前虚拟机内每一条线程正在执行的方法堆栈的集合.生成线程快照的主要目的是为了定位线程长时间停顿的原因.如死锁,死循环,请求外部资源导致的长时间等待.
- **JConsole**:可视化监视和管理工具,几乎包括以上工具的所有功能

类加载

类的加载被分为了七个阶段：

加载–>验证–>准备–>解析–>初始化–>使用–>卸载

- 加载：通过一个类的全限定名（灵活度提高）来获取定义此类的二进制字节流，并把此二进制字节流表示的静态存储结构转化为方法区运行时数据结构，并生成代表这个类的class对象放在方法区（HotSpot放在方法区，其他的可能不一样）中作为类数据的访问入口。
- 验证：确保class文件的自己留包含信息符合jvm的要求，不会危害jvm安全，比如魔术块，版本检测，是否符合java规范，符号引用验证（解析）
- 准备：为类变量（static修饰的变量，不是所有的变量哦）分配内存，并设初值（设初值，全部是零值，真正赋值在初始化那里），但对于像存在final修饰的类变量会使用属性表附带的值直接初始化。
- 解析：将符号引用变为直接引用。符号引用包含如类引用，方法和字段的引用，都要找到对应的类或接口中寻找，并转化为直接引用（指向目标的指针），用全限定名去找。
- 初始化：收集类中所有赋值语句和静态块内容，组合成类构造器，收集顺序按照出现顺序，且静态块里面对于在其后定义的变量可写不可读。然后执行这个类构造器，且执行顺序会先执行最顶级父类，从上往下，所有最先执行的永远是Object（接口（无静态块，但有赋值）与实现类顺序是先子类）
- 类加载的时机是由虚拟机自定义实现的，但初始化过程的触发一般源于以下情况
  - new关键字创建实例；读取设置一个类的静态字段，调用类的静态方法；利用反射进行类的调用；初始化子类时，会先初始化其父类；虚拟机会先初始化带main方法的类
- **类与类加载器**：类加载器只用于实现类的加载动作，但作用却远不止此，判断两个类是否相等，相等的前提条件是两个类由同一个类加载器加载（不然就算两个类来着与同一个class文件，也会判断不相等）
- **双亲委派模型**

  同一个class文件若被不同的类加载器加载会导致两个类不相等，而由于自定义类加载器的存在，这可能导致许多最基本的行为无法保证，应用程序变得混乱（如自定义加载器加载Object类到不同路径下，导致不相等，会导致需要混乱后果）

  于是引入了双亲委派模型，首先要说一下三种系统提供的类加载器

  - 启动类加载器Bootstrap ClassLoader（虚拟机的一部分） 是嵌在JVM内核中的加载器，该加载器是用C++语言写的，主要负载加载JAVA_HOME/lib下的类库和被-Xbootclasspath参数指定路径下由虚拟机识别的类库，启动类加载器无法被应用程序直接使用（涉及虚拟机本地实现细节）。
  - 扩展类加载器Extension ClassLoader： 用JAVA编写，且它的父类加载器是启动类加载器（查看父类为空，因为启动类是c++写的），主要加载JAVA_HOME/lib/ext目录中的类库。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），也称为系统类加载器，默认负责加载应用程序classpath目录下的所有jar和class文件（用户写代码路径下）。它的父加载器为扩展类加载器。

  自定义加载器的父类加载器是应用程序类加载器，但类加载器之间的父子关系不是使用继承来实现的，而是使用组合关系来复用父加载器的代码。

  双亲委派模型： 如果一个类加载器收到了一个类加载请求，它不会自己去尝试加载这个类，它会调用自己的loadClass方法：首先把这个请求转交给父类加载器去完成。每一个层次的类加载器都是如此。因此所有的类加载请求都应该传递到最顶层的启动类加载器中，只有到父类加载器反馈自己无法完成这个加载请求（在它的搜索范围没有找到这个类）时，子类加载器才会调用自己的findClass方法尝试自己去加载。 委派的好处：

  类加载器带有优先级的层次关系，避免有些类被重复加载；保证了Java程序的稳定运行；实现简单。

当我们自己写java包下的类时，并打包到同样的包下，由于双亲委派模型的原因，这个类不会被加载，因为已经有一个一样的类被加载了，这时会返回异常。

线程上下文加载器：第二次破坏时建立，为了打破父级循环调用子类加载器，第三次热部署。

参数调优

- 常用基础参数：-Xms：初始化堆大小；-Xmx：最大堆大小；-Xmn：年轻代大小；-Xss：栈大小参数；-XX:SurvivorRatio：Eden区与幸存区的大小比例；晋升年龄设置；直接在老年代中分配内存的大对象的阈值设置。
- 并行收集器参数：设置年轻代，老年代使用并行收集器；配置并行收集的线程数；设置收集时间与吞吐量
- CMS参数：设置使用CMS收集器；设置利用full GC来整理空间，整理碎片；因为浮动垃圾无法清除的原因，设置进行回收时是在空间使用多少时触发；
- 调优思路：把-Xms和-Xmx设置的一样大，避免初始化很多对象时，多次增加内存；内存分配一般要与处理器个数成正比；对应响应时间敏感和吞吐量敏感的吧年轻代设置大一些，避免minor GC频繁触发；使用CMS时要特别注意空间碎片，合理使用压缩；

锁优化

1.自旋锁：许多应用，共享数据的锁定状态只会持续很短一段时间忙循环（自旋），避免了线程切换的开销；自适应自旋锁，自旋的时间不再固定（自旋容易获得就多次自旋等待，不然就减少或不旋转）

2.锁消除：被检测到不可能存在共享数据竞争的锁进行消除

3.锁粗化：有一串零碎的操作都对同一个对象加锁，将会把加锁同步的范围粗化到整个操作序列的外部。

4.轻量级锁

代码进入同步块时，虚拟机使用CAS操作尝试将对象的Mark word（指向重量级锁的指针）更新为指向线程栈的Lock Record的指针，如果更新成功，则轻量级锁获取成功*，记录锁状态为轻量级锁；*否则，说明已经有线程获得了轻量级锁，目前发生了锁竞争（不适合继续使用轻量级锁），接下来膨胀为重量级锁。（使用CAS代替重锁进行控制，效率更高，但不适用于竞争大的时候）

5.偏向锁：那偏向锁就是在无竞争的情况下把整个同步都消除掉，连CAS都不做。偏向锁的意思是这个锁会偏向第一个获得它的线程，如果在接下来的执行过程中，该锁没有被其他线程获取，则持有偏向锁的线程不需要再进行同步。

消息队列

流量削峰，异步处理，程序解耦。

秒杀系统

将库存预存到redis中，前台首先应该禁止同一个用户重复提交请求，然后前台秒杀时来临的大量请求先过中间件（消息队列）进行缓冲（比如用一个原子变量做key，用户id做值），key对应就买到的库存值，达到额定库存值后，不再进行队列的插入操作。后续用户均不在入队，全部是未抢到商品状态。然后后端再从消息队列中取数据依次处理。（若不使用key对应库存的想法，可对库存进行减操作再比较，使用decr命令）

- 可靠投递（最终一致性）：发送方需要收到接收方的确认消息，确定消息已经到达，才会删除消息，不然都当做接收方未收到，但这样会导致消息重复（如果是单方发送可以利用版本号机制来避免重复）
- 消息重复消费问题：对于本身幂等的消息没事，但对应修改数据库等操作是明显不能重复消费的。消费者更加消息进度去消费消息，还没来得及提交进度，就crash了，还没提交消费进度，重启就会按照没更改的消费进度去重复消费同一条消息，这种只能尽可能保证不发生crash，或者耗费时间在本地记录发生的消费避免重复消费。对于另一种消费者转换消息队列消费时，应当先提交消息进度再切换，避免重复的消费，同时使用新消息队列时也要保证原消费者提交了进度。重要的要保证幂等性（给消息带上id，最近消费过的几条id存入redis，每次消费一条消息之前都在一个储存中查询一下）。
- 消息队列消息积压，一般消息积压说明消费者有问题，可能死在了某个地方，无法继续消费，所以首先解决消费者问题，让消费者恢复正常；接下来要对消息队列进行扩容，避免消息队列爆掉，其次增加消费者，把积压的消息先处理掉。对于可能过期丢失的消息，晚上加班写代码排查重新入队。

JDBC

注册驱动，建立连接，创建Statement进行sql的运输，利用Statement执行sql语句。

事务

通过连接的`setAutoCommit`（设置事务开始点）、`commit`（提交事务）、`rollback`（回滚事务），事务原子性被违背时会返回SQLException，利用try，catch实现，在try最后提交事务，在catch里面回滚事务并关闭连接。

servlet

访问servlet，及生命周期（127.0.0.1:8080/app/hello）

根据ip和端口请求发到tomcat；解析app对应的应用，找到其对应的web.xml文件；在web.xml中根据映射hello找到本地对应那个类的那个方法；在tomcat下找整个类文件；第一次要初始化servlet（调用init）；执行（request包含请求内容）；返回给服务器；服务器读取response；发送http给客户端；servlet会一直存在到服务器被关闭或应用被卸载。

servlet是单例的，web服务器中有线程池，每个请求都会去请求一个线程，由线程去调用servlet，单例的好处在于减少开销，然后用多线程调用单实例。

ServletContext

所有Servlet共享同一个ServletContext对象。ServletContext对象通常也被称之为context域对象

request&response

获取客户机提交的数据找request对象。

- 储存了客户端发送信息的头和参数，数据等；
- 应用：获取表单信息，请求转发（getRequestDispatcher().forward）,封装数据进去让转发的服务器取。

向容器输出数据找response对象。

- 响应头里面指定编码格式，发送http头，控制刷新网页，用输出字节流给客户端输出数据
- 应用：文件下载，通知客户端请求重定向（response.sendRedirect()，302/307状态码）

cookie&session

- Cookie

Cookie是客户端技术，程序把每个用户的数据以cookie的形式写给用户各自的浏览器（程序给浏览器）。用户会带着自己的数据访问服务器中的web资源。

当浏览器第一次访问服务器时，服务器会为用户生成一个cookie（sessionId要通过这个找到），并通过响应发给用户，浏览器收到响应会把自己的cookie存下来，以后发送请求时就会带上。

- HttpSession

Session是服务器端技术，服务器在运行时为每一个用户的浏览器创建一个其独享的HttpSession对象，由于session为用户浏览器独享，所以用户在访问服务器的web资源时，可以把各自的数据放在各自的session中，session为值与key成为映射，而服务器发送给浏览器的cookie就保存了这个key，所以浏览器带cookie访问服务器其他资源时，就能拿到对应的session。

5.**jsp**

JSP实际上就是Servlet，可嵌java代码，JSP既用java代码产生动态数据，又做美化会导致页面难以维护

Listener和Filter

- Listener
  - 监听web常见对象（HttpServletRequest，HttpSession，ServletContext）

    - 监听对象创建销毁，属性变化，session绑定javaBean（配置只需要配置监听器类，监听器就能运行）
    - 步骤：自定义类实现对应监听器接口，重写方法，web.xml中配置监听（绑定javaBean不需要配置）
- Filter
    - 拦截访问web资源的请求与响应操作（Jsp, Servlet, 静态图片文件或静态 html 文件等）
    - 作用：实现URL级别的权限访问控制、过滤敏感词汇、压缩响应信息等一些高级功能
    - 步骤：自定义类实现Filter接口，重写方法，web.xml中配置（配置这个类及要拦截那些请求）
    - 多个Filter时会逐个访问，按照在web.xml里配置的顺序
    - 在监听器内部能拿到request，并进行操作后调用doFilter后才算请求过了这层过滤器，不然不能访问。

Spring

IOC

控制反转，把创建对象的权限有java程序交付给spring容器

- DI：依赖注入，为要创建的对象提起绑定属性等（构造函数，set），实现松耦合（面向接口），依赖注入方式有构造函数，setter，注解注入。
- ApplicationContext和BeanFactory：Spring中提供的两种IoC容器。A是B的子类，功能更强大。
    - BeanFactory：无特殊指定采用延迟初始化的机制，即访问容器中的对象时才会对其管理对象进行初始化和依赖注入。启动快，适合资源受限的场景
    - ApplicationContext：提供了许多其他高级特性，且容器初始化时就初始化其管理对象，启动较慢，资源要求较多，更适用于功能要求更高的时候
- 信息加载：容器中每一个对象都对应一个BeanDefinition实例对象，此对象用于保存注入对象的所有信息（class类型，抽象类，构造方法参数等）。用一个专门解析配置文件的类（XmlBeanDefinitionReader），负责读取spring指定格式的xml配置文件并解析，将解析后的文件内容映射为相应的BeanDefinition。
- Bean的注册：容器创造一个对象的过程称为Bean的注册，实现Bean的注册的接口为BeanDefinitionRegistry BeanFactory其实也只是接口，真正的实现类是同时实现这两个接口的，注册利用BeanDefinitionRegistry接口中的registerBeanDefinition(BeanDefinition beandefinition)方法来进行Bean的注册。
- **ApplicationContext容器初始化过程**
    - 调用父类容器的构造方法(super(parent)方法)为容器设置好Bean资源加载器（传参为空，但父类构造方法会加载一个资源加载器）。
    - 调用父类的setConfigLocations(configLocations)方法设置Bean定义资源文件（.xml）的定位路径。
    - IoC容器对Bean定义资源的载入：refresh()函数，refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。refresh的作用类似对IoC容器的重启，在新建立的容器中对容器进行初始化，对Bean定义资源进行载入
        - 调用容器准备刷新的方法，获取容器的当时时间，同时给容器设置同步标识（prepareRefresh()）
        - **加载beanFactory**（最常用实现XmlBeanFactory）
            - 调用本类的obtainFreshBeanFactory()方法去调用子类的refreshBeanFactory()方法
            - refreshBeanFactory()中首先判断BeanFactory是否存在，存在则销毁beans并关闭beanFactory；接着创建DefaultListableBeanFactory，并调用loadBeanDefinitions(beanFactory)装载bean定义。
            - loadBeanDefinitions(beanFactory)里面创建了Bean读取器XmlBeanDefinitionReader（得到要加载的资源，资源加载器就在这里用到的），其负责读取spring指定格式的xml配置文件并解析，将解析后的文件内容映射为相应的BeanDefinition，处理每个Bean元素和元素的值，最后将 beanDefinition 注册进 BeanFactory。
            - 处理每个Bean元素和元素的值，最后将 beanDefinition 注册进 BeanFactory。
        - **加载bean**
            - refresh方法中有个方法会实例化非懒加载的单例Bean
            - 调用`preInstantiateSingletons`方法分析`BeanDefinition`**判断**，如果是自己配置的Bean调用`getBean`方法

- 调用doGetBean方法，这里会发现Spring把实例好的Bean存入singletonObjects中，这是一个ConcurrentHashMap，但是这里bean还未进行实例化，转入下面语句执行
- 在这里拿到RootBeanDefinition检查，并获得**Bean的依赖，循环迭代实例化依赖Bean**，完成之后判断是否是单例的或者非单例，区别在于单例是被缓存起来的，非单例不用缓存，最终调用`createBean(beanName, mbd, args)`
- 调用`instantiate(RootBeanDefiniton, beanName, BeanFactory)`方法，判断如果没有无参构造器则生成CGLIB子类，否则直接反射生成实例，调用`instantiateClass`调用Constructor的`newInstance`方法
- 有了实例对象之后，调用`postProcessPropertyValues`方法委托`InjectionMetadata`对象完成属性注入，依赖关系保存在`checkedElements`集合里，然后执行`inject`方法，反射注入

bean的生命周期

- 实例化：通过new关键字将bean实例化，使用默认构造函数
- 填入属性：spring用bean读取器根据配置的xml文件为每个bean生成了BeanDefinition，BeanDefinition中保存了每个bean的详细信息，spring会调用bean的setter方法，将属性注入（最先）。
  - 首先调用Bean自身的方法，配置文件中⑥bean的init-method（自定义初始化方法）
  - Bean级生命周期接口方法，①BeanNameAware的setBeanName（设置bean的id属性，让bean获取得自身的id属性）、②BeanFactoryAware的setBeanFactory（设置bean对应的工厂让Bean拥有访问beanFactory的能力，但增加了耦合度）、③ApplicationContextAware的setApplicationContext（设置bean对应的容器让Bean拥有访问Spring容器的能力，但增加了耦合度）、⑤InitializingBean的afterPropertiesSet方法（属性初始化后的处理方法）
  - 容器级生命周期接口方法，BeanPostProcessor（初始化前后添加一些自己的逻辑处理） 两个对应实现类的④postProcessAfterInitialization和⑦postProcessAfterInitialization方法。
- bean准备就绪，可以正常使用
- 销毁，销毁时调用bean的destroy-method，根据scope配置的不同销毁时间也不一样
  - singleton单例：spring的bean默认是单例的，此时bean创建的是唯一实例，容器初始化时被创建，容器销毁时销毁
  - prototype原型：每个请求或每次使用getbean方法对应创建一个新的bean，销毁释放由客户端代码决定。
  - request：每一次Http对应一个新的bean，仅在此次请求request中有效
  - session：每一次Http对应一个新的bean，仅在此次http中的session中有效
  - globalsession：类似上面

AOP

为业务功能进行增强，且增强对业务透明。

- 动态代理：窃取消息并装饰，运行时增强
  - JDK动态代理（目标类是实现了接口的类，可以用，也可以用CHLIB）
    - 自定义事件处理器（invoke方法中就能指定方法前后要干嘛了）
      - 事件处理器实现InvocationHandler接口，并在属性中创建Object对象保存要代理的对象实例
      - 重写invoke方法，invoke方法中参数为代理对象实例，方法对象，方法参数，`method.invoke(target,args)`;可以执行传入的方法并返回实际返回值，这段代码前后就加自己的逻辑
      - 通过Proxy.newProxyInstance（classLoder，interfaces，事件处理器实例）返回传入目标的代理类
      - 事件处理器也往往使用匿名类来传入
    - 利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。
    - 代理类继承了Proxy类并且实现了要代理的接口，由于java不支持多继承，所以JDK动态代理不能代理没有接口的类
  - CGLIB代理（对指定的类生成一个子类，覆盖其中的方法）
    - 自定义事件处理器（intercept方法中就能指定方法前后要干嘛了）
    - 事件处理器实现MethodInterceptor接口，实现intercept方法，保存要代理对象的实例

    - 使用Enhancer保存要代理对象的类加载器，设计回调方法的类实例（事件处理器），并创建返回

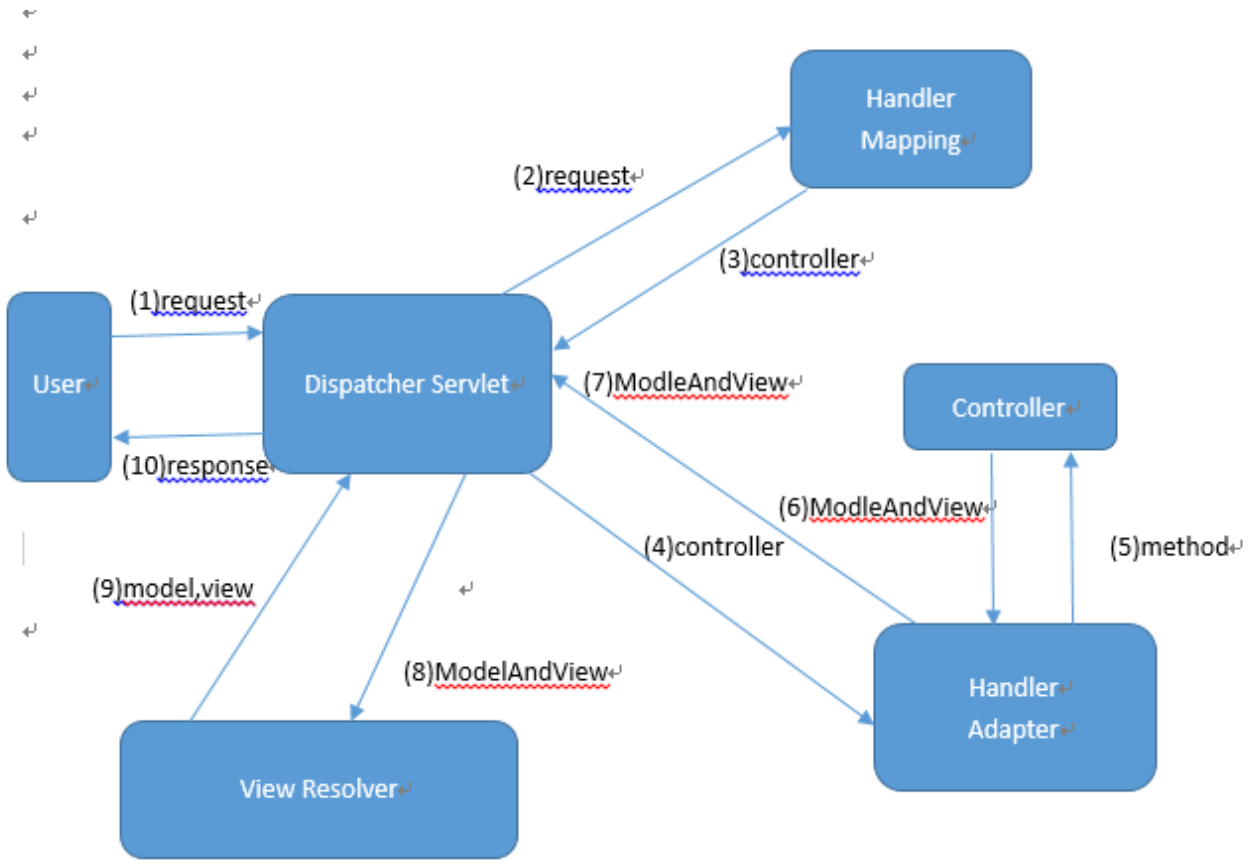
```
Enhancer enhancer=new Enhancer();
enhancer.setSuperclass(target.getClass());
enhancer.setCallback(this);
return enhancer.create();
```

    - 利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。
    - JDK调用代理方法，是通过反射机制调用，Cglib是通过FastClass机制直接调用方法，Cglib执行效率更高。
- 静态织入：特定语法来创建方面，编译时织入，类加载期织入
    - 自定义类实现各种方法（类前加@Aspect表示这是切面类）
        - 在每个方法前加注解表示把这个方法织入到那个业务方法的那个位置，同时在spring配置这个切面（声明切面的存在）
        - 另一种方法方法前不加注解指定，而是写完方法后直接去配置spring.xml，在配置文件中指定切面类中的方法要织入的那些业务的那些位置。

事务（特性：ACID，原子，一致，隔离，持久）

- 在xml中配置事务管理器（用作管理事务，不由spring负责）和启动事务注解，Spring事务处理模块通过AOP功能来实现声明式事务处理，AOP生成代理对象，然后把事务处理的功能编织到拦截的方法中。
- 事务注解-@Transactional
- **传播行为**
    - REQUIRED：当前方法必须运行在事务中，有事务存在就允许，没有就创建了运行
    - SUPPORTS：不需要事务上下文，存在事务就在当前事务运行
    - MANDATORY：必须在事务上运行，没有事务抛异常
    - REQUIRED_NEW：当前方法必须运行在自己的事务中，事务中调用方法会为这个方法开启一个新事务，并把当前事务挂起，等待新事务提交或回滚再执行自己的事务。
    - NOT_SUPPORTED：当前方法不能运行在事务中，否则导致事务挂起
    - NEVER：当前方法不能运行在事务中，否则抛异常
    - NESTED：允许事务嵌套，嵌套的事务可单独回滚和提交
- **隔离级别**
    - DEFAULT：后端数据库默认隔离级别（mysql：repeatable read；oracle：read commited）
    - read uncommited：最低级别，不加锁，允许读取未提交数据，可能出现脏读，不可重复读，幻读
    - read commited：写的过程行锁锁住数据，避免脏读，可能出现不可重复读，幻读
    - repeatable read：行锁锁住数据整个事务，避免脏读和不可重复读，可能出现幻读
    - serializable：表锁锁住数据整个事务，避免三种情况，效率最低，锁整个表。

SpringMVC



（1）客户端通过url发送请求

（2-3）核心控制器Dispatcher Servlet接收到请求，通过系统或自定义的映射器配置找到对应的handler，并将url映射的控制器controller返回给核心控制器。

（4）通过核心控制器找到系统或默认的适配器

（5-7）找到的适配器（springmvc定义了多种接口实现，所以需要适配器），调用实现对应接口的处理器，并将结果返回给适配器，结果中包含数据模型和视图对象，再由适配器返回给核心控制器

（8-9）核心控制器将获取的数据和视图结合的对象传递给视图解析器，获取解析得到的结果，并由视图解析器响应给核心控制器

（10）核心控制器将结果返回给客户端

Mybatis

动态SQL

- if用作判断

  ```
  <if test="id!=null"> id=#{id}<!--sql语句--> </if>
  ```

- where用作拼接去掉多余前置and和or

  ` id=#{id} AND last_name LIKE #{lastName} AND email=#{email} AND gender=#{gender}
  `

- trim去掉自定义前后缀，添加自定义前后缀，prefix:前缀，trim标签体是整个字符串拼串后的结果， prefix给拼串后的整个字符串加一个前缀 prefixOverrides:前缀覆盖：去掉整个字符串前面多余的字符 suffix：后缀，给整个字符串后缀加一个后缀

suffixOverrides：后缀覆盖：去掉整个字符串后面多余的字符

` id=#{id} AND last_name LIKE #{lastName} AND email=#{email} AND gender=#{gender}
`

类型转换

在使用mybatis时都是直接传入对象，返回也是直接返回对象，所以在传入对象到实际储存之间有一个转换器，我们需要自定义转换器实现TypeHandler接口，比如String数组需要转换为String，中间用某些符号进行分割，才能用varchar存进数据库。

缓存

- 一级缓存：通过配置文件开启一级缓存后`<setting name="localCacheScope" value="SESSION"/>`，通过配置value的值为SESSION或者STATEMENT指定一级缓存当前会话内（ sqlSession ）对整个缓存有效，还是当前的Statement有效。这时在配置范围内的多条一样的sql中（内部使用多个关键字段结合做键存在hashmap中），除了第一条sql语句会去访问数据库后更新缓存，后面的都会直接读缓存。同一个sqlSession更新数据后会导致一级缓存失效，但要注意一级缓存只在单个会话内有效，如果另一个会话更新数据是不会导致缓存失效的，就可能导致了缓存与实际数据的不一致现象。
- 二级缓存：`<setting name="cacheEnabled" value="true"/>`二级缓存的开启使得缓存对于所有会话有效，二级缓存会被多个sqlSession共享。访问顺序：二级缓存、一级缓存、数据库。当一个会话执行修改时，二级缓存会全部失效（缺点，所以适用于多读少写场景），另一个会话会直接访问数据库。

Maven

生命周期

- clean 清理之前的项目
  - pre-clean：执行清理前的准备工作
  - clean：清理上一次构建的项目
  - post-clean：执行清理的后续
- default构建项目
  - 比较多，一般先要处理主资源文件，然后编译项目的主源代码，处理项目的测试资源文件并编译，测试运行程序（test）
  - package：将编译好的主资源文件打包为可发布格式，比如jar
  - install：将包安装到本地仓库工本地其他maven项目使用
  - deploy：将最终的包复制到远程仓库，供其他人员和项目使用
- site建立项目站点
  - pre-site，site，post-site：生成项目站点文档
  - site-depoly：将生成项目站点发布到服务器

解决jar冲突

由于jar包存在传递依赖问题，在导入一个包时同时也会导入其依赖的其他包，然后再导入新导入包的依赖包，这样由于两个jar包依赖同一个jar包的不同版本就产生了包冲突现象。

maven处理方式是两个不同版本的包只导入一个，选择思路为最短路径优先（选择依赖层数少的那个），最先声明优先（谁先要求导入，要那个，后面声明的冲突包就不要了）

设计模式

jdk用的设计模式：Format工厂模式，单例RuntimeException，适配器Arrays.asList，装饰器Reader和Writer，代理Proxy，迭代器模式

单例模式

- 饿汉：线程安全，但单例会被直接创建，如果没使用，会存在浪费（static）
- 懒汉：线程不安全，但单例只有在被使用的时候才会创建（static）
  - 双重校验锁：判空，同步代码块（class对象），判空，初始化（变量static，volatile（避免重排序））
  - 同步方法：同步方法，只创建一次，效率低（变量static，方法synchronized(class对象)）
- 静态内部类：静态内部类中创建静态实例，通过另一个方法返回其静态实例。加载类时不加载静态内部类，调用方法返回静态内部类实例才会初始化，jvm保证只初始化一次。
- 枚举：写法简单，并发安全，加载时，赋值其实是在静态块初始化，类似饿汉模式，jvm保证只会初始化一次，还能防止序列化破坏单例对象（自由序列化，编译器禁用了一些方法防止单例被破坏）。

简单工厂

创建一个工厂类，存在一个创建方法，根据传入的参数来返回对应的类（多态）

```
Produce produce = new Factory.createProduct(1);//对应参数返回各种子类produce
```

工厂方法

使用一个抽象类定义工厂，其中定义一个抽象方法，抽象方法需要返回一个produce，使用另一个方法来返回这个抽象方法的返回值，而每个produce对应的子类都写一个工厂类来继承这个抽象类工厂，并实现创建的抽象方法。

抽象工厂模式

抽象工厂模式一次创建的是多个对象，而不是一个，假设现在有A产品3种，B产品3种，并一一对应是三个组合，那么需要有三个工厂（都具有方法createA()和createB()），保证每种工厂的方法创建出一种组合。思路是首先创建两个类AbsA和AbsB，并让A的3种类继承AbsA，B的三种类继承AbsB（此时就相当于用继承对AB进行了分类）；创建抽象类AbsFactory（具有抽象方法createA()和createB()），此时再创建三种工厂Factory1，Factory2，Factory3都继承AbsFactory，实现其两个抽象方法，分别返回组合A1和B1，A2和B2，A3和B3。

观察者模式

一个主题和多个观察者，主题内维护一个列表，列表中放置观察主题的对象，主题可以设置自己的参数状态（同时更新观察者的状态），可以加入观察者，删除观察者。观察者初始化时传入一个主题，直接让当前对象进入主题维护的队列中，且观察者具备一个update方法，方便主题修改自身状态时，逐个调用列表中所有观察者的update从而实现同步观察者状态的目的。

策略模式

一组算法在运行时动态的选择合适的算法，比如定义了排序算法接口Sort，方法sortMethod()，现在有冒泡，插入，希尔，快速，归并等多种算法都实现这个接口，并实现不同种类的sortMethod()。并自定义类DoSort中用Sort定义一个排序类对象sort，这个对象可以随时通过set方法修改对应的实体类，这样随时动态修改sort，外部创建一个DoSort后，传入不同的实例类就可通过sortMethod()使用不同的排序算法。

适配器模式

假设现在有A和B两个接口，然后对应实现类AI和BI，内部具有方法aaa()和bbb()，现在想要BI的实例类调用aaa()方法（鸡发出鸭叫）。那么可以创建一个适配器实现A接口，内部定义一个B的变量，实例化时传入BI实现类给B创建的变量，然后实现aaa()方法（aaa中调用BI实例的bbb()方法）。

装饰者模式

为基础一直添加装饰，且有多种装饰，每种装饰可以重复添加。例如往煎饼里面加鸡蛋，热狗，鸡柳等，每个材料都有自己的价格，根据不同的配方返回总价。策略是定义一个煎饼接口Pan，接口方法返回价格cost()，接下来定义初始的煎饼类Pancake，并实现cost()返回煎饼初始价格。接下来所有的配料类都实现煎饼类，初始化需要传入当前的煎饼对象，其实现cost()返回自己的价格+传入煎饼实例的价格（n+pan.cost()）。

代理模式

实现同一个接口，代理类和被代理类都实现同一个接口方法，在代理类中定义被代理类的对象，代理类实现接口方法时其实就是调用被代理

静态代理与动态代理的区别主要在：

- 静态代理在编译时就已经实现，编译完成后代理类是一个实际的class文件
- 动态代理是在运行时动态生成的，即编译完成后没有实际的class文件，而是在运行时动态生成类字节码，并加载到JVM中